Gabriel Rădulescu Marius Olteanu

Programare în limbaj de asamblare

Lucrări practice

Editura Universităţii Petrol-Gaze din Ploieşti
2007

P R E F A Ț Ă

Programarea în limbaj de asamblare reprezintă activitatea de elaborare a programelor într-un limbaj ce se apropie într-o importantă măsură de resursele primare ale maşinii. Este cunoscut faptul că între limbajul cod maşină şi cel de asamblare există o corespondenţă biunivocă, astfel încât se poate considera că acesta din urmă permite nivelul cel mai avansat de gestionare a disponibilităţilor unui procesor.

Cu toate că în momentul de faţă multe limbaje de nivel înalt oferă posibilitatea interacţiunii directe cu adrese de memorie, porturi, registre etc. limbajul de asamblare continuă să fie folosit, cel puţin sub forma unor proceduri, în situaţiile în care acestea nu pot face faţă unor constrângeri cum ar fi, de exemplu, cele legate de timpul de execuţie.

În ceea ce priveşte procesul de instruire a viitorilor specialişti în domeniul proiectării, programării şi utilizării calculatoarelor, însuşirea programării în limbaj de asamblare contribuie la formarea unei gândiri algoritmice şi permite o abordare non-formală a unor discipline de studiu chemate să contureze profilul profesional al acestora.

Prezenta colecţie de lucrări practice se constituie într-un îndrumar al activităţilor de laborator destinat studenţilor specializării Calculatoare care audiază disciplina **Programarea în limbaje de asamblare**. Lucrările acoperă o gamă largă de aspecte care permit o formare graduală a deprinderilor practice în ceea ce priveşte programarea microprocesoarelor din clasa Intel 80x86. Lucrările se încadrează în programa analitică a disciplinei şi răspund obiectivelor aferente activităţilor practice ale acesteia. De asemenea, lucrarea reprezintă un ghid extrem de util pentru orice altă disciplină de studiu ce presupune – pe anumite segmente de activitate – programare *low-level*: **Calculatoare numerice**, **Sisteme de operare**, **Proiectarea cu microcontroller-e**, **Proiectarea cu microprocesoare** (pentru specializarea Calculatoare), **Arhitectura calculatoarelor**, **Automate programabile**, **Sisteme cu microprocesoare** (pentru specializarea Automatică şi Informatică Aplicată), respectiv **Arhitectura sistemelor de calcul** (pentru studenţii specializării Informatică).

Fiecare lucrare conţine câte un breviar asociat temei respective, o aplicaţie orientată pe tematica aferentă precum şi o temă care trebuie rezolvată pe parcursul şedinţei de laborator. Abordarea acestei teme oferă studenţilor posibilitatea verificării la fiecare şedinţă de laborator a măsurii în care au fost realizate obiectivele lucrării. Este de menţionat faptul că îndrumarul constituie un auxiliar preţios în vederea pregătirii testului de evaluare a activităţii de laborator.

Prin modul de abordare a problemelor şi structurarea logică a materialului prezentat, autorii acestei lucrări îşi exprimă convingerea că ea reprezintă un instrument de lucru deosebit de util pentru studenţii specializărilor amintite, precum şi a tuturor celor interesaţi de o carieră profesională în domeniul tehnicilor şi tehnologiilor IT.

Dr. Ing. Gabriel Rădulescu

Autorii doresc să aducă mulţumiri tuturor celor ce au sprijinit apariţia aceste lucrări, în special Dlui Prof. Univ. Dr. Ing. Liviu Dumitraşcu şi Dlui Prof. Univ. Dr. Ing. Nicolae Paraschiv, pentru grija şi spiritul critic-constructiv cu care s-au aplecat asupra manuscrisului. De asemenea, mulţumirile noastre colegiale se îndreaptă către Dl. Ing. Constantin Stoica, ale cărui propuneri şi observaţii pertinente asupra lucrării noastre au contribuit în mod cert la ridicarea valorii acesteia.

Nu în ultimul rând, autorii sunt recunoscători studenţilor specializărilor Calculatoare, respectiv Automatică şi Informatică Aplicată (grupele 1804, 1805, 1909 şi 1910), cei ce au testat nemijlocit, în cadrul activităţilor de laborator, materialul primar ce a stat la baza prezentului îndrumar de lucrări practice, aducându-şi propria contribuţie la îmbunătăţirea conţinutului său.

CUPRINS

LUCRAREA 1

DESCRIEREA PACHETULUI DE PROGRAME TURBO-ASSEMBLER

1. Obiectivele lucrării

Lucrarea urmăreşte familiarizarea studenţilor cu mediul de dezvoltare a programelor scrise în limbaj de asamblare pus la dispoziţie de către produsul Turbo-Assembler al firmei Borland. Se prezintă pe scurt componentele pachetului de programe şi procedura generală de lucru, cu exemplificare pe clasa programelor în format *EXE*.

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

Pachetul de programe Turbo-Assembler pune la dispoziţia utilizatorului toate instrumentele necesare pentru dezvoltarea programelor în limbaj de asamblare. Macroasamblorul *TASM* produce module obiect relocabile din fişiere sursă în limbaj de asamblare. Pentru aceste module obiect se pot edita legăturile cu programul *TLINK* în scopul creării de programe executabile sub sistemul de operare *DOS*. Pachetul cuprinde:

► macroasamblorul *TASM*;
► editorul de legaturi *TLINK*;
► bibliotecarul *TLIB*;
► depanatorul simbolic *TD*.

Pentru depanare se poate folosi şi depanatorul standard al sistemului de operare *DOS*, *DEBUG*.

3.1. Alegerea tipului de fişier executabil în sistemul de operare DOS

Sunt posibile următoarele formate de fişiere executabile:

► *EXE* este formatul uzual pentru fişiere executabile sub sistemul de operare *DOS*. Programele executabile păstrate în acest format pot avea segmente multiple. Este formatul recomandat pentru programe de dimensiuni mari. Extensia lor implicită este *.EXE*.
► *COM* – programele în acest format sunt limitate pentru un singur segment, din acest motiv ele nedepăşind 64 Ko (exceptând cazurile când nu se specifică segmente). Se recomandă pentru programe mici. Un dezavantaj important este acela că nu conţin informaţii despre sursă şi simbolice pentru depanatorul simbolic *TD*. Ele pot fi depanate însă cu *DEBUG* sau cu *TD* în mod nesimbolic.

3.2. Comenzi necesare dezvoltării unui program

Pentru aflarea opţiunilor la linia de comandă a programelor *TASM* şi *TLINK* se lansează în execuţie aceste programe fără nici un parametru în linia de comandă. Efectul este afişarea sintaxei liniei de comandă împreună cu opţiunile disponibile. Comenzile uzuale sunt expuse în continuare.

► Pentru asamblarea unui program:

TASM nume

unde *nume* este numele fişierului ce conţine programul sursă, implicit cu extensia *.ASM*.

► Pentru asamblarea unui program cu obţinerea fişierului listing:

TASM /l nume

unde *nume* este numele fişierului ce conţine programul sursă, implicit cu extensia *.ASM*.

► Pentru asamblarea unui program cu depunerea de informaţii simbolice în fişierul obiect rezultat, informaţii necesare depanatorului simbolic *TD*:

TASM /zi nume

unde *nume* este numele fişierului ce conţine programul sursă, implicit cu extensia *.ASM*;

► Pentru editarea legăturilor unui program:

TLINK nume

unde *nume* este numele fişierului obiect rezultat în urma asamblării, implicit cu extensia *.OBJ*. Implicit se generează şi harta de alocare a memoriei.

► Pentru editarea legăturilor unui program cu depunerea de informaţii simbolice în fişierul executabil rezultat (necesare depanatorului simbolic *TD*):

TLINK /v nume

unde *nume* este numele fişierului obiect rezultat în urma asamblării cu opţiunea */zi*, implicit cu extensia *.OBJ*. Implicit se generează şi harta de alocare a memoriei.

► Pentru editarea legăturilor unui program în vederea obţinerii unui fişier executabil în format *COM*:

TLINK /t nume

unde *nume* este numele fişierului obiect rezultat în urma asamblării unui program sursă în format *COM*. Implicit are extensia *.OBJ*.

►Pentru depanarea programului executabil obţinut:

TD nume

unde *nume* este numele fişierului executabil rezultat în urma editării legăturilor, implicit cu extensia *.EXE* sau *.COM*.

Depanarea cu *TD* poate fi **obişnuită** sau **simbolică**. Depanarea obişnuită presupune ca în momentul depanării să se cunoască adresele variabilelor din program împreună cu segmentele în care se află aceste variabile. Depanarea simbolică presupune ca, în momentul execuţiei, programul depanator să aibă acces la numele simbolice ale variabilelor din program, depanarea fiind în acest caz mai uşor de realizat.

Pentru **depanarea obişnuită** se parcurg următorii paşi:

► Se asamblează şi link-editează programul;

► Se lansează *TD* în execuţie cu comanda menţionată mai sus. Efectul este apariţia pe ecran a unei ferestre împărţită în 5 subferestre: subfereastra programului, subfereastra de registre, subfereastra de flaguri, subfereastra de stivă şi subfereastra de date. Trecerea dintr-o subfereastra în alta se face apăsând tasta *TAB*;

► Se identifică deplasamentele şi adresele de segment ale variabilelor din program pe baza examinării fişierelor listing şi / sau a subferestrei programului;
► Se vizualizează valorile iniţiale ale variabilelor prin activarea subferestrei de date, apăsarea tastelor *CTRL-G* şi introducerea în căsuţa de dialog a adresei variabilei ce se vrea examinată (sub forma *segment: offset*);
► Se activează subfereastra programului şi se execută apoi în regim pas cu pas (instrucţiune cu instrucţiune) apăsând tastele *F7* sau *F8* şi simultan examinând celelalte subferestre;
► La terminarea depanării se părăseşte programul depanator cu *Alt-X*.

Pentru **depanarea simbolică** se parcurg următorii paşi:

► Se asamblează şi se link-editează programul cu opţiunile de depanare simbolică prezentate mai sus;
► Se lansează *TD* în execuţie prin comanda indicată. Pe ecran apare o fereastră ce conţine programul aşa cum a fost introdus în fişierul sursă;
► Se introduc în fereastra *Watches* variabilele al căror conţinut se doreşte a fi urmărit în timpul execuţiei, prin apăsarea tastelor *Ctrl-F7* şi introducerea în căsuţa de dialog ce apare a numelui acestor variabile. Eventual se deschide fereastra de registre cu comanda *Alt-V R*;
► Se execută programul pas cu pas (instrucţiune cu instrucţiune) prin apăsarea tastelor *F7* sau *F8*, urmărind fereastra *Watches* şi fereastra de registre;
► La terminarea depanării se părăseşte programul depanator cu *Alt-X*.

4. Desfăşurarea lucrării

Procedura generală de lucru este descrisă în continuare.

► Se foloseşte un editor de texte pentru crearea sau modificarea modulelor sursă. Prin convenţie, acestea au extensia implicită *.ASM*. Ele se pot organiza în diverse moduri. Spre exemplu, toate procedurile se pot plasa într-un modul mai mare sau se pot scinda pe mai multe module.

► Se utilizează *TASM* pentru asamblarea fiecărui modul de program. Se pot utiliza opţional fişiere de includere (fişiere ce sunt incluse în modulul asamblat doar pe durata asamblării). Dacă apar erori după executarea acestui pas se revine la pasul anterior. Pentru fiecare fişier sursă *.ASM* se creează în urma asamblării:

- un fişier obiect cu extensia implicită *.OBJ*;
- un fişier listing cu extensia implicită *.LST* (opţional);
- un fişier de referinţe încrucişate cu extensia implicita *.REF* (opţional).

Dacă se doreşte legarea de module scrise în asamblare cu module scrise într-un limbaj de nivel înalt, acestea din urmă se vor compila în fişiere obiect.

► Opţional se foloseşte *TLIB* pentru plasarea mai multor fişiere obiect într-un singur fişier bibliotecă cu extensia implicita *.LIB*. Acest lucru poate fi avut în vedere atunci când se doreşte legarea unor fişiere obiect standard la mai multe programe. De asemenea *TLIB* poate crea un fişier opţional de listare a conţinutului bibliotecii.

► Se foloseşte *TLINK* pentru a combina toate fişierele obiect şi modulele bibliotecă ce formează un program în cadrul unui singur fişier executabil care are implicit extensia *.EXE*. Opţional se poate crea şi o hartă de alocare a memoriei, cu extensia implicită *.MAP*.

► Se poate realiza editarea legăturilor cu opţiunea */l* pentru crearea (dacă este necesară) a unui fişier executabil în format binar. Această operaţie se aplică pentru programele scrise în format *COM* (care vor avea extensia *.COM*) sau pentru fişierele binare.

► Se depanează programul pentru depistarea erorilor de logică. Dintre tehnicile utilizate se amintesc:

- studiul răspunsului programului la diferite seturi de date de intrare;
- studiul fişierelor sursă şi listing;
- utilizarea fişierului listing cu referinţe încrucişate;
- utilizarea depanatorului simbolic *TD* pentru depanare dinamică.

4.1. Exemple de programe în format EXE

Primul program, prezentat în continuare, afişează pe ecran un mesaj de salut.

```
TITLE hello
.MODEL small
.STACK 100h
.DATA
   Message1 DB 'Hello! ',13,10, '$'
.CODE
Start:
      mov ax,@data
      mov ds,ax;
      mov dx, offset  Message1
      mov ah,9h
      int 21h
      mov ah,4ch
      int 21h
END Start
```

Se remarcă următoarele:

► Directiva *.MODEL* este folosită pentru iniţializarea modelului de memorie, semnificând modul de organizare a informaţiei în memoria sistemului;

► Directiva *.STACK* defineşte o stivă de 256 octeţi, suficientă pentru programe mici;

► Directiva *.DATA* marchează începutul segmentului de date. În cazul de faţă acest segment conţine o variabilă şir de caractere, Message1, ce reţine mesajul care va fi afişat în program. Numerele 13, 10 codifică „Enter" (trecerea la rândul următor);

► Eticheta primei instrucţiuni ce urmează directivei *.CODE* marchează începutul porţiunii executabile a programului (numit şi *punctul de intrare în program*). Aceeaşi etichetă este utilizată după directiva *END* ce delimitează sfârşitul programului ea definind astfel punctul de unde va începe execuţia programului;

► Primele două instrucţiuni iniţializează registrul DS. Simbolul *@data* reprezintă numele segmentului creat cu directiva *.DATA*. Registrul DS va fi întotdeauna iniţializat pentru fişiere sursă în format *EXE*;

► Conţinutul variabilei de tip şir de caractere definită în program se afişează pe ecran utilizând funcţia *DOS* cu codul *09h*.

► Funcţia *DOS* cu codul *4Ch* este utilizată pentru terminarea programului şi predarea controlului la sistem. Există şi alte tehnici de ieşire la sistem, dar aceasta este cea recomandată.

Programul următor are ca scop afişarea pe ecran a caracterului ASCII cu codul 1.

```
.MODEL SMALL
.STACK 200H
.CODE
START:

mov ah, 2
mov dl, 1
int 21h           ; Tiparire caracter

mov ah, 4ch
mov al,00h
int 21h           ; Terminare program

end START
```

Se remarcă faptul că plasarea comentariilor în program debutează cu caracterul " ; ".

4.2. Citirea fişierelor de listare a asamblării

TASM va crea un fişier de listare a asamblării sursei ori de câte ori se specifică numele unui astfel de fişier în linia de comandă *TASM* sau se foloseşte opţiunea de asamblare */l*. Fişierul de listare a asamblării conţine atât instrucţiunile din fişierul sursă cât şi codul obiect (dacă există) generat pentru fiecare instrucţiune. Listarea prezintă, de asemenea, numele şi valorile tuturor etichetelor, variabilelor şi simbolurilor din fişierul sursă.

Asamblorul creează tabele pentru macrodefiniţii, structuri, înregistrări, segmente, grupuri şi alte simboluri. Aceste tabele sunt plasate la sfârşitul fişierului de listare a asamblării (exceptând cazul în care ele sunt suprimate cu opţiunea */n*). *TASM* listează numai tipurile de simboluri întâlnite în program. De exemplu, dacă programul nu conţine macrodefiniţii, în tabloul de simboluri nu va exista o secţiune de macrodefiniţii. Toate numele de simboluri vor fi afişate cu majuuscule, exceptând cazul în care sunt utilizate opţiunile */ml* sau */mx* pentru a specifica nume ce conţin si litere mici.

Asamblorul listează codul generat de instrucţiunile unui fişier sursă. Fiecare linie are sintaxa

Număr_linie deplasament cod instrucţiune

unde:

- *număr_linie* - este numărul liniei din listing. Numerotarea începe de la prima instrucţiune din fişierul de listare a asamblării. Numerele liniilor sunt generate numai atunci când se doreşte crearea unui fişier de referinţe încrucişate. Numerele liniilor dintr-un listing nu corespund întotdeauna liniilor din fişierul sursă;
- *deplasament* - reprezintă "distanţa" (în octeţi) de la începutul segmentului curent până la codul corespunzător liniei respective;
- *cod* - indică valoarea numerică în hexazecimal a codului generat dacă această valoare este cunoscută în timpul asamblării. Dacă valoarea este calculată în timpul execuţiei, *TASM* indică acţiunea necesară pentru calcularea valorii;
- *instrucţiune* - este instrucţiunea sursă prezentată exact aşa cum apare în fişierul sursă, sau aşa cum este expandată în urma unei macrosubstituţii.

Dacă apar erori în timpul asamblării, fiecare mesaj de eroare va apărea chiar sub instrucţiunea în care a apărut eroarea. În continuare este prezentat un exemplu de linie ce conţine o eroare şi mesajul de eroare corespunzător.

```
     11    0005  A1 0000      mov  ax,aa
**Error** smiley.ASM(11) Undefined symbol: AA
```

Numărul 11 din mesajul de eroare reprezintă linia sursă în care a apărut eroarea. Listingul model prezentat în continuare este realizat prin asamblarea programului din fişierul *smiley.asm*.

Linia de comandă este

TASM /l smiley.asm

Listingul rezultat, plasat în fişierul *smiley.lst*, este prezentat în continuare.

```
    Turbo Assembler    Version 4.1         05/31/06 23:58:27
Page 1
    smiley.asm



          1     0000                    .MODEL SMALL
          2     0000                    .STACK 200H
          3     0000                    .CODE
          4     0000                    START:
          5
          6     0000  B4 02                   Mov ah, 2
          7     0002  B2 01                   Mov dl, 1
          8     0004  CD 21                   Int 21h
          9
         10     0006  B4 4C                   mov ah, 4ch
         11     0008  B0 00                   mov al,00h
         12     000A  CD 21                   int 21h
         13
         14                             END START
    Turbo Assembler    Version 4.1         05/31/06 23:58:27
```

Segmentele şi grupurile utilizate în fişierul sursă sunt listate la sfârşitul programului, indicându-se dimensiunea lor, tipul de aliniere, tipul de combinare şi clasa. Dacă în fişierul sursă au fost utilizate directive de segment simplificate, numele reale de segmente generate de *TASM* vor fi listate în tabelă. Pentru exemplificare este prezentată tabela de segmente şi grupuri corespunzătoare listingului anterior:

```
    Groups & Segments         Bit Size Align  Combine Class

    DGROUP                    Group
    STACK                     16  0200 Para     Stack   STACK
    _DATA                     16  0000 Word     Public  DATA
    _TEXT                     16  000C Word     Public  CODE
```

Prima coloană indică numele tuturor segmentelor şi grupurilor din modulul sursă. Numele de segmente şi grupuri sunt afişate în ordine alfabetică, exceptând cazul în care numele de segmente ce aparţin unui grup sunt plasate sub numele grupului (în ordinea în care au fost adăugate acestuia).

Coloana *Bit* indică numărul de biţi ce caracterizează entităţile (cuvintele) segmentului respectiv.

Coloana *Size* indică dimensiunea (numărul de octeţi, în hexazecimal) a fiecărui segment. Nu este prezentată dimensiunea grupurilor.

Coloana *Align* indică tipul de aliniere al segmentului, implicit fiind *Para*, adică aliniere la paragraf (multiplu de 16).

Coloana *Combine* indică tipul de combinare a segmentului. Dacă nu este definit un tip de combinare explicit pentru segment, se asociază implicit *none* (tip de combinare special).

Coloana *Class* indică numele clasei segmentului.

Toate simbolurile (cu excepţia numelor pentru macrodefiniţii, structuri, înregistrări şi segmente) sunt prezentate în tabela de simboluri aflată la sfârşitul listingului. Pentru exemplificare, se consideră tabela de simboluri corespunzătoare listingului anterior:

```
Page 2
Symbol Table

Symbol Name                Type       Value

??DATE                     Text       "05/31/06"
??FILENAME                 Text       "smiley "
??TIME                     Text       "23:58:27"
??VERSION                  Number 040A
@32BIT                     Text       0
@CODE                      Text       _TEXT
@CODESIZE                  Text       0
@CPU                       Text       0101H
@CURSEG                    Text       _TEXT
@DATA                      Text       DGROUP
@DATASIZE                  Text       0
@FILENAME                  Text       SMILEY
@INTERFACE                 Text       000H
@MODEL                     Text       2
@STACK                     Text       DGROUP
@WORDSIZE                  Text       2
START                      Near       _TEXT:0000
```

Coloana *Symbol Name* prezintă numele simbolurilor în ordine alfabetică.

Coloana *Type* prezintă tipul fiecărui simbol, fiecare tip având propria semnificaţie:

Tip	**Semnificaţie**
Near	Etichetă cu referinţă apropiată
Far	Etichetă cu referinţă îndepărtată
Number	Etichetă absolută
Alias	Simbol definit cu directiva *EQU*
Text	Echivalare de text
Byte	Un octet
Word	Un cuvânt (doi octeţi)
Dword	Dublu cuvânt (patru octeţi)
Fword	Cuvânt de lungime şase octeţi
Qword	Cuvânt de lungime opt octeţi
Tbyte	Zece octeţi

Tipul unei variabile generate cu operatorul de multiplicare *DUP*, cum ar fi un tablou sau un şir, este tipul elementului de bază din structura variabilei.

Coloana *Value* indică valorile simbolurilor din program. Valoarea unui simbol depinde de tipul simbolului. Spre exemplu, dacă simbolul reprezintă o variabilă, o etichetă sau o procedură, coloana *Value* conţine deplasamentul simbolului (în hexazecimal) faţă de începutul segmentului în care este definit, împreună cu numele segmentului respectiv.

Echivalările de text de la începutul tabelei sunt cele definite automat de asamblor.

5. Modul de lucru

► Se vor asambla şi rula programele prezentate obţinându-se fişiere *.EXE*.
► Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
► Se vor testa cât mai multe dintre opţiunile de lucru ale macroasamblorului *TASM*.
► Se va rula sub *DEBUG* sau *TD* cel puţin unul dintre programele prezentate.

LUCRAREA 2

UTILIZAREA MEDIULUI DE DEPANARE TURBO-DEBUGGER

1. Obiectivele lucrării

Prin efectuarea acestei lucrări de laborator se doreşte atingerea următoarelor obiective:

- ►Cunoaşterea metodologiei de elaborare a fişierelor executabile sub rezidenţa mediului Turbo-Assembler astfel încât acestea să poată fi dezasamblate şi să se poată utiliza codul sursă iniţial;
- ►Însuşirea principalelor comenzi şi facilităţi ale mediului Turbo-Debugger.

2. Aparatura şi suporturile utilizate

- ►PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
- ►Precizările din prezentul îndrumar.

3. Breviar teoretic

În general, în scrierea unui program se deosebesc două tipuri de greşeli: de sintaxă şi de semantică. Cele de sintaxă se referă la regulile de scriere asociate limbajului în care se programează, la setul de instrucţiuni, operatori şi operanzi acceptaţi de acesta şi la corectitudinea scrierii lor. Ele sunt detectate de compilatoare sau asambloare. Erorile de semantică (conceptuale) pot apărea chiar şi atunci când toate greşelile sintactice au fost corectate, fiind greşeli ce privesc concepţia programului. Mai precis, erorile semantice apar în cazul în care se reuşeşte crearea cu succes a unui fişier executabil, însă, la rulare, se observă că acesta nu atinge (total sau parţial) scopul pentru care a fost creat.

3.1. Erori semantice frecvent întâlnite în Turbo-Assembler

Lipsa apelului de întoarcere la promptul DOS

Într-un limbaj de nivel înalt, ca de exemplu C/C++ sau Pascal, execuţia unui program se termină automat după parcurgerea întregului cod sursă, chiar dacă nici o comandă explicită pentru terminarea execuţiei nu a fost scrisă în program, deoarece un limbaj de nivel înalt generează automat aceste comenzi. Situaţia este diferită în limbajul de asamblare, unde numai comenzile scrise explicit sunt executate. Când rulează un program care nu are nici o comandă de revenire în sistemul de operare DOS, execuţia programului continuă după ultima instrucţiune, încărcându-se „codul" aflat la adrese succesive ultimei instrucţiuni, cu rezultate imprevizibile.

Generarea greşită a întoarcerii

Instrucţiunea PROC are două efecte. În primul rând defineşte un nume prin care o procedură este apelată. În al doilea rând stabileşte dacă procedura este de tipul "near" sau "far". Instrucţiunea de întoarcere dintr-o procedură trebuie să corespundă cu tipul acesteia, "near" sau "far", în caz contrar rezultatul întoarcerii nefiind cel dorit.

Inversarea operanzilor

În instrucţiunea MOV AX,BX procesorul scanează linia de la stânga la dreapta, acesta fiind exact modul în care mulţi producători de microprocesoare şi-au dezvoltat limbajele lor de asamblare.

Firma Intel a abordat altfel problema în cazul limbajului de asamblare pentru procesoarele 80x86, astfel linia MOV AX, BX înseamnă copierea conţinutului lui BX în AX. Cu alte cuvinte, ordinea operanzilor în instrucţiunile 80x86 este inversată, ceea ce poate fi o sursă de erori semantice.

Alocarea unei stive prea mici

Dacă nu se alocă explicit spaţiu pentru stivă, există o mare probabilitate ca programul să nu funcţioneze corect. Instrucţiunile PUSH şi POP precum şi apelurile de proceduri sunt exemple ce implică utilizarea stivei. Programele care nu au alocată stiva uneori pot funcţiona, dar nu există siguranţa că acest lucru se întâmplă întotdeauna. Programele DOS pot conţine o directivă ".STACK" care rezervă spaţiu pentru stivă. Pentru fiecare program trebuie alocat spaţiu mai mult decât necesar pentru a preveni erorile.

Apelarea unei subrutine care şterge regiştrii

La scrierea codului în limbajul de asamblare este posibilă interpretarea regiştrilor ca variabile locale, dedicate la un moment dat unei proceduri. Există tendinţa de a presupune că regiştrii rămân neschimbaţi în urma apelurilor altor proceduri. În realitate, regiştrii sunt variabile globale şi orice procedură poate şterge conţinutul unuia sau mai multor regiştri. De aceea subrutinele trebuie să salveze în stivă regiştrii afectaţi (folosiţi în cadrul lor) şi să-i refacă înainte de revenirea în programul care le-a apelat.

Depăşirea cauzată de instrucţiuni

Instrucţiunile de lucru cu şiruri au o proprietate interesantă. După ce sunt executate, pointerii pe care i-au folosit indică o adresă la distanţa de 1 sau 2 octeţi (pentru instrucţiunile ce operează cu byte/word) de ultima adresă. Acest lucru poate crea confuzii în cazul instrucţiunilor repetate, mai ales REP SCAS şi REP CMPS. După execuţia acestor instrucţiuni, registrele index nu mai indică adresa de început a şirului, ci altă adresă, uneori utilă, alteori nu.

Utilizarea incorectă a flag-ului de direcţionare

Când este executată o instrucţiune de lucru cu şiruri, pointerul sau pointerii asociaţi registrelor index sau amândoi sunt incrementaţi sau decrementaţi depinzând de valoarea flag-ului de direcţionare (ce se menţine până la o nouă scriere).

Deşi este relativ uşoară programarea direcţiei o dată şi apoi executarea unei serii de instrucţiuni care operează în acelaşi sens, flag-ul de direcţionare poate fi de asemenea responsabil pentru erori frecvente şi greu de depistat, făcând ca efectul instrucţiunilor să depindă de codul executat anterior.

Ştergerea accidentală a regiştrilor în urma execuţiei unor instrucţiuni

Înmulţirea, fie că este 8 bit cu 8 bit, 16 bit cu 16 bit sau 32 bit cu 32 bit întotdeauna schimbă conţinutul cel puţin unui registru în afara porţiunii de acumulator folosită ca operand sursă. Spre exemplu, la multiplicarea lui AX cu un cuvânt din memorie, rezultatul se găseşte în perechea DX:AX, deci se pierde conţinutul lui DX.

Similar, instrucţiunile precum MOVS, STOS, LODS, CMPS, şi SCAS pot afecta câteva flag-uri şi până la trei regiştri în timpul execuţiei unui singur apel.

Omiterea testării flag-urilor

Durata validităţii flag-urilor este limitată la apelul unei noi instrucţiuni care le modifică, fiind de obicei nu prea lungă. De aceea starea flagurilor trebuie testată imediat după instrucţiunea care le-a modificat, în caz contrar erorile fiind deosebit de dificil de depistat.

Gestiunea incorectă a regiştrilor la apelul întreruperilor

Fiecare întrerupere ar trebui să salveze conţinutul tuturor regiştrilor. Deşi este corect să se păstreze doar conţinutul regiştrilor care se modifică, este de preferat să se salveze toţi regiştrii la începutul unei întreruperi ("PUSH") şi apoi să fie restauraţi ("POP") la terminarea întreruperii. În caz contrar pot exista modificări accidentale de conţinut ce afectează execuţia la revenirea din întrerupere.

3.2. Descrierea mediului Turbo-Debugger

Meniul View

Opţiunea Breakpoints (puncte de întrerupere)

Breakpoints sunt folosite pentru a specifica o acţiune particulară atunci când programul a rulat până la o linie specifică din sursă, linie indicată de numărul ei sau de adresă. Se poate seta un Breakpoint pentru a opri programul la orice linie din sursă sau la orice adresă prin apăsarea tastei F2. O nouă apăsare a lui F2 anulează un Breakpoint. Se poate de asemenea seta un breakpoint prin click cu mouse-ul pe primele două coloane ale liniei respective.

Un breakpoint poate fi „instruit" pentru a realiza una din următorele acţiuni:

- **Break** – această opţiune permite oprirea programului de fiecare dată când rularea a ajuns în dreptul breakpoint-ului. Controlul revine mediului Turbo-Debugger pentru a permite examinarea programului şi a valorilor variabilelor, regiştrilor, memoriei, stivei etc.
- **Log** – permite specificarea unei variabile sau unei expresii referitoare la memorie a cărei valoare să fie notată în fereastra de înregistrare (Log window) ori de câte ori linia de întrerupere este rulată.
- **Execute** – permite specificarea unei expresii ce trebuie evaluată de fiecare dată când se atinge breakpoint-ul.

Opţiunea Watches

Deschide o fereastră în care se pot citi valorile variabilelor sau expresiilor adăugate prin comanda **Add Watch** din meniul **Data**.

Se pot examina date cu ajutorul opţiunii **Inspect** din meniul **Data**, fiind validă orice expresie care converge într-un pointer de memorie, de la nume de variabile simple până la complicate expresii ale unei adrese.

Opţiunea CPU

Aflată în meniul **View,** prin activarea sa se deschide fereastra **CPU** (cea mai importantă din mediul Turbo-Debugger) în care se găsesc instrucţiunile dezasamblate, o zonă cu date în format hexazecimal reprezentând conţinutul memoriei (zona aflată în subsolul ferestrei), registrele şi flagurile procesorului respectiv conţinutul stivei. Fereastra indică tipul procesorului după setul de instrucţiuni dezasamblat (8086, 80286, 80386, 80486, Pentium).

Meniurile locale pentru fereastra CPU conţin:

►Zona cu instrucţiuni: **Go to** (poziţionare la o anumită adresă în segmentul de cod), **Search** (caută o instrucţiune), **Source** (arată cod sursă original, dacă a fost inclus în informaţiile de depanare), **Assemble** (inserează o instrucţiune în locaţia curentă);
►Zona cu imaginea memoriei: **Go to** (poziţionare la o adresă), **Search** (caută o secvenţă de octeţi), **Change** (schimbă octeţii din memorie), **Block** (operaţii cu blocuri de memorie);
►Zona cu imaginea stivei: **Go to**, **Search**, **Change**;
►Zona cu regiştrii procesorului, ce permite prin meniul local incrementarea sau decrementarea lor, setarea pe zero sau orice altă valoare;
►Zona cu flag-uri, ce permite comutarea acestora (**Toggle**).

Principalele flag-uri ale procesorului 80x86 standard sunt prezentate în tabelul 1.

Tabelul 1. Flag-urile procesorului standard

Nume flag	Semnificaţie
C	Carry – transport la rangul următor
Z	Zero – rezultat nul; egalitate
S	Sign – semn rezultat, 1=negativ
O	Overflow – depăşire aritmetică
P	Parity – paritatea rezultatului, 1=par
A	Auxiliary Cary – transport auxiliar, pt. calcule BCD
I	Interrupt – întreruperi activate sau nu
D	Direction – pt. lucrul cu şiruri, direcţia de deplasare

Opţiunea Numeric Processor (coprocesorul matematic)

Se poate deschide fereastra **Numeric Processor** alegând din meniu **View/Numeric Processor**. Linia superioară indică adresa ce conţine instrucţiunea curentă, pointerul OPCODE ce indică tipul instrucţiunii şi pointerul de date. Această fereastră conţine trei secţiuni: secţiunea din stânga (secţiunea regiştrilor-**Register Pane**) ce arată starea curentă a registrelor de calcul în virgulă mobilă, secţiunea din mijloc (**Control Pane**) ce arată flag-urile de control şi secţiunea din dreapta (**Status Pane**) ce indică flag-urile de stare.

Linia aflată deasupra indică următoarele informaţii despre ultima operaţie în virgulă mobilă ce a fost executată:

►**Emulator** indică procesorul numeric care este emulat. Dacă este prezent un procesor numeric (8087,80287,80387 sau 80486) este afişat acesta;
►**IPTR** arată adresa fizică de 20 biţi de la care ultima operaţie în virgulă mobilă a fost executată;
►**OPCODE** arată tipul instrucţiunii;
►**OPTR** indică adresa fizică pe 16 biţi sau 20 biţi a locaţiei de memorie referită de instrucţiune, dacă există.

Secţiunea Registers

Secţiunea regiştrilor indică fiecare dintre regiştrii de calcul în virgulă mobilă, ST(0) până la ST(7), împreună cu starea lor: **valid**, **zero** sau **empty**. Conţinutul este afişat ca un număr pe 80 biţi în virgulă mobilă.

Meniul local al secţiunii "Registers" permite setarea pe zero, pe empty (adică nefolosit) şi schimbarea valorii curente. Acest meniu poate fi activat prin tastarea Alt+F10 sau prin intermediul tastei Ctrl şi al primei litere al comenzii dorite (exemplu: Ctrl+E=Empty).

Secţiunea de Control

Tabelul 2 prezintă diferitele flag-uri de control şi modul în care sunt ele afişate în această secţiune.

Tabelul 2. Flag-urile de control ale coprocesorului matematic

Nume în rubrică	Descrierea flag-ului
im	Operaţie invalidă (invalid operation mask)
dm	Operand anormal (denormalized operand)
zm	Diviziune cu zero (zero divide mask)
om	Depăşire superioară (overflow mask)
um	Depăşire inferioară (underflow mask)
pm	Precizie (precision mask)
iem	Întrerupere activată (interrupt enable mask – doar 8087)
pc	Control precizie (precision control)
rc	Control rotunjire (rounding control)
ic	Control infinit (infinity control)

Flagurile de control pot fi comutate cu opţiunea **Toggle** din meniul local.

Secţiunea de Stare

Tabelul 3 descrie flag-urile de stare şi modul în care apar ele în secţiunea de stare.

Tabelul 3. Flag-urile de stare ale coprocesorului matematic

Nume în panou	Descrierea flagului
ie	Operaţie invalidă (invalid operation)
de	Operand denormalizat (denormalized operand)
ze	Diviziune prin zero (zero divide)
oe	Depăşire superioară (overflow)
ue	Depăşire inferioară (underflow)
pe	Precizie (precision)
ir	Cerere de întrerupere (interrupt request)
cc	Condiţie (condition code)
st	Indicator vârful stivei (stack top pointer)

Flagurile de stare pot fi comutate cu opţiunea **Toggle** din meniul local.

Taste Funcţionale

Există taste funcționale pentru multe din funcţiile mediului Turbo-Debugger, acestea fiind:

►F1 – Ajutor;
►F2 – Breakpoint;
►F3 – Încarcă program;
►F4 – Rulează până la poziţia curentă;
►F5 – Redimensionare ferestre;
►F6 – Următoarea fereastră;
►F7 – Parcurgere prin subrutine;
►F8 – Execută subrutinele direct;
►F9 – Rulează programul;
►F10 – Activează meniu.

Secvența de întrerupere Ctrl-Break poate fi apăsată în orice moment în timp ce programul rulează. Imediat execuţia programului este suspendată şi mediul Turbo-Debugger preia controlul.

4. Desfăşurarea lucrării

Se consideră următorul program:

```
.MODEL  SMALL
.STACK  100h
.DATA

TimePrompt              DB  'E trecut de ora 12? (D/N)?$'
GoodMorningMessage      DB  13,10,'Buna dimineata!',13,10,'$'
GoodAfternoonMessage    DB  13,10,'Buna ziua!',13,10,'$'
DefaultMessage   DB  13,10,'Gandesti deci existi!',10,13,'$'
.CODE
start:
  mov     ax,@data
  mov     ds,ax     ;DS va indica către segmentul de date

  mov     dx,OFFSET TimePrompt
  mov     ah,9      ;functie DOS: afiseaza sir de caractere
  int     21h
  mov     ah,1      ;functie DOS: citeste un caracter
  int     21h
  or      al,20h    ;forteaza caracterul in litera mica
  cmp     al,'d'    ;testeaza daca a fost apasata tasta d
  je      IsAfternoon
  cmp     al,'n'    ;testeaza daca a fost apasata tasta n
  je      IsMorning

  ;salutul implicit
  mov     dx,OFFSET DefaultMessage
  jmp     DisplayGreeting
```

```
IsAfternoon:
  mov     dx,OFFSET GoodAfternoonMessage
  jmp     DisplayGreeting

IsMorning:
  mov     dx,OFFSET GoodMorningMessage

DisplayGreeting:
  mov     ah,9
  int     21h         ;afiseazã mesajul ales
  mov     ah,4ch      ;functie DOS: terminarea programului
  mov     al,0        ;intoarce codul 0
  int     21h

END start
```

În continuare se prezintă succint ciclul de dezvoltare – testare: asamblarea, editarea legăturilor (linkeditarea) şi depanarea, specifice asamblorului TASM.

Asamblare:
tasm /l /zi salut.asm
Opţiunea */l* comandă generarea unei listări a programului în fişierul *salut.lst*. Parametrul */zi* este plasat pentru a da debugger-ului informaţii complete, utile la dezasamblare.

Link-editare:
tlink /v salut.obj
Parametrul */v* are acelaşi scop ca şi parametrul */zi* de la asamblare, furnizând informaţii despre tabela de simboluri.

Fişierul rezultat, "salut.exe", poate fi direct executat sau se poate încărca în Turbo-Debugger:
td salut.exe

5. Modul de lucru

- ►Se editează programul exemplu (în orice mediu de editare) şi se salvează cu extensia .ASM.
- ►Se asamblează şi link-editează programul, obţinându-se forma .EXE (cu respectarea indicaţiilor din secţiunea „Desfăşurarea lucrării").
- ►Se testează programul prin încărcarea sa în depanator, utilizând cât mai multe opţiuni de lucru ale mediului Debugger.
- ►Folosind acelaşi procedeu, se va edita, asambla şi link-edita programul următor, apoi se va încărca în mediul Turbo-Debugger şi, prin rularea sa pas cu pas (F7), se vor urmări modificările survenite asupra regiştrilor şi flagurilor specificate:

```
.model small
.stack 100h
.data
.code

Start:
 mov ax,@data  ;segment date in DS
 mov ds,ax

               ;incarca 1234h in AX:
 mov al,34h    ;mai intai partea low
 mov ah,12h    ;si apoi partea high

               ;scade 1234h din AX
 sub ax,1234h  ;rezultat nul, se observa flagul Z=1

               ;scade 1 din AX
 sub ax,0001h  ;rezultat negativ, se observa flagul S=1

               ;incarca in AX valoarea maxima
 mov ax,7FFFh  ;ptr un numar cu semn pe 16 biti

               ;aduna 1 la AX
 add ax,0001h  ;depasire, se observa flagul O=1

 xor ax,ax     ;sterge continutul lui AX

 mov ax,4C00h  ;revenire in MS-DOS
 int 21h
end Start
```

LUCRAREA 3

REPREZENTĂRI INTERNE ALE DATELOR

1. Obiectivele lucrării

Această lucrare de laborator are ca scop însuşirea de către studenţi a noţiunilor de bază referitoare la:

► sistemele de numeraţie binar, octal şi hexazecimal;
► definirea datelor şi a codurilor;
► codul ASCII;
► reprezentări în virgulă mobilă;
► reprezentări în virgulă fixă.

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

În această lucrare sunt prezentate câteva concepte importante, încluzând reprezentarea datelor numerice (în sistemul binar, octal şi hexazecimal) şi a datelor alfanumerice (setul de caractere ASCII), în contextul utilizării acestora în programe scrise în limbaj de asamblare.

3.1. Sisteme de numeraţie

Atunci când scriem 198, 97788, 10009 sau când efectuăm o operaţie de adunare, scădere, înmulţire sau împărţire, folosim *sistemul de numeraţie zecimal*. El este un sistem de numeraţie poziţional, adică ponderea unei cifre depinde atât de valoarea sa, cât şi de poziţia sa în reprezentarea numărului. Deci, numărul 10 este baza sistemului de numeraţie pe care îl folosim zi de zi şi aceasta se datorează, fără îndoială, faptului că omul are zece degete la mâini. În limba engleză cuvântul "*digit*" desemnează orice număr de la 0 la 9 adică orice cifră a sistemului de numeraţie zecimal.

Alfabetul sistemului de numeraţie zecimal este format din cifrele: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Aceste simboluri definesc numere diferite. Alfabetul conţine 10 simboluri, adică un număr de simboluri egal cu baza sistemului de numeraţie.

Alfabetul este format dintr-un şir de simboluri care reprezintă numere consecutive: 0; 1 = 0 + 1; 2 = 1 + 1; 3 = 2 + 1, …

Primul simbol din şir este 0, adică simbolul care defineşte un număr fără nicio valoare. Simbolul cu valoarea cea mai mare este 9, adică un număr cu o unitate mai mic decât baza 10 (9 = 10 – 1).

Un număr în baza 10 poate fi scris ca o suma de puteri ale lui 10:

$1995 = 1\times10^3+9\times10^2+9\times10^1+5\times10^0$.

deci, în funcţie de poziţia lor în număr, simbolurile au fost înmulţite cu puterile bazei 10.

Astfel, un număr va fi reprezentat de un şir de simboluri:

- un număr de 2 cifre: $N = \overline{ab} = a\times10+b$;
- un număr de 3 cifre: $N = \overline{abc} = a\times100+b\times10+c$;
- un număr de 4 cifre: $N = \overline{abcd} = a\times1000+b\times100+c\times10+d$.

Aceste observaţii sunt valabile pentru orice sistem de numeraţie poziţional, deoarece sistemele de numeraţie diferă între ele doar prin alfabetul de reprezentare şi, implicit, prin baza sistemului.

Prin urmare, un *sistem de numeraţie în baza q* este un sistem de reprezentare a numerelor care are următoarele caracteristici:

- Utilizează un alfabet cu q simboluri diferite între ele, numite cifre, care formează un şir de numere consecutive;
- Prima cifră din şir este 0;
- Cifra cu valoarea cea mai mare este cu o unitate mai mică decât baza sistemului (q– 1);
- În funcţie de poziţia lor în număr, cifrele se înmulţesc cu puteri crescătoare ale bazei q, obţinându-se dezvoltarea numărului după puterile bazei:

$N_{(q)} = \overline{a_n a_{n-1} a_{n-2} \ldots a_2 a_1 a_0} = a_n\times q^n+a_{n-1}\times q^{n-1}+a_{n-2}\times q^{n-2}+\ldots+a_2\times q^2+a_1\times q^1+a_0\times q^0$

Pentru a şti în ce sistem de numeraţie este scris un număr, în dreapta lui, între paranteze rotunde, se scrie valoarea bazei, astfel $N_{(q)}$ înseamnă că numărul este scris în baza q.

În contextul programării în limbaj de asamblare, prezintă interes 3 sisteme de numeraţie:

- Sistemul de numeraţie binar;
- Sistem de numeraţie octal;
- Sistem de numeraţie hexazecimal.

Sistemul de numeraţie binar

Baza acestui sistem de numeraţie este 2, alfabetul fiind format doar din două simboluri (0 şi 1) care alcătuiesc un şir de numere consecutive (0; 1 = 0 + 1), simbolul cu valoarea cea mai mare este 1, el fiind cu o unitate mai mic decât baza 2 (1 = 2 - 1).

În tabelul 1 este prezentată modalitatea de numărare în baza 2 pentru numerele de la 0 la 15 inclusiv.

Tabelul 1. Numărare în baza 2

Zecimal	Binar	Zecimal	Binar	Zecimal	Binar	Zecimal	Binar
0	0	4	100	8	1000	12	1100
1	1	5	101	9	1001	13	1101
2	10	6	110	10	1010	14	1110
3	11	7	111	11	1011	15	1111

Un număr scris în baza 2 va fi dezvoltat după puterile bazei astfel:

$N_{(2)} = \overline{a_n a_{n-1} a_{n-2} ... a_2 a_1 a_0} = a_n \times 2^n + a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$.

Exemple de numere binare: $11_{(2)}$, $101_{(2)}$, $110001_{(2)}$, $1100011_{(2)}$.

Dacă se dezvoltă aceste numere după puterile bazei se obţine:

$11_{(2)} = 1 \times 2^1 + 1 \times 2^0$;

$101_{(2)} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$;

$110001_{(2)} = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$;

$11100011_{(2)} = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$.

Sistemul de numeraţie octal

Baza acestui sistem de numeraţie este 8, iar alfabetul este format din opt simboluri (0, 1, 2, 3, 4, 5, 6 şi 7) care alcătuiesc un şir de numere consecutive (0; 1 = 0 + 1; 2 = 1 + 1; 3 = 2 + 1; 4 = 3 + 1; 5 = 4 + 1; 6 = 5 + 1; 7 = 6 + 1). Simbolul cu valoarea cea mai mare este 7, el fiind cu o unitate mai mic decât baza 8 (7 = 8 – 1).

În tabelul 2 este prezentat felul de numărare în baza 8.

Tabelul 2. Numărare în baza 8

Zecimal	Octal	Zecimal	Octal	Zecimal	Octal	Zecimal	Octal
0	0	4	4	8	10	12	14
1	1	5	5	9	11	13	15
2	2	6	6	10	12	14	16
3	3	7	7	11	13	15	17

Un număr scris în baza 8 va fi dezvoltat după puterile bazei astfel:

$N_{(8)} = \overline{a_n a_{n-1} a_{n-2} ... a_2 a_1 a_0} = a_n \times 8^n + a_{n-1} \times 8^{n-1} + a_{n-2} \times 8^{n-2} + \ldots + a_2 \times 8^2 + a_1 \times 8^1 + a_0 \times 8^0$.

Exemple de numere octale: $11_{(8)}$, $107_{(8)}$, $110022_{(8)}$, $76543210_{(8)}$.

Dacă se dezvoltă aceste numere după puterile bazei se obţine:

$11_{(8)} = 1 \times 8^1 + 1 \times 8^0$;

$107_{(8)} = 1 \times 8^2 + 0 \times 8^1 + 7 \times 8^0$;

$110022_{(8)} = 1 \times 8^5 + 1 \times 8^4 + 0 \times 8^3 + 0 \times 8^2 + 2 \times 8^1 + 2 \times 8^0$;

$76543210_{(8)} = 7 \times 8^7 + 6 \times 8^6 + 5 \times 8^5 + 4 \times 8^4 + 3 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 0 \times 8^0$.

Sistemul de numeraţie hexazecimal

Sistemul de numeraţie hexazecimal este caracterizat prin:

Baza acestui sistem de numeraţie este 16, alfabetul fiind format din şaisprezece simboluri (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E şi F). Deoarece cifrele nu mai sunt suficiente pentru a scrie 16 simboluri diferite s-au utilizat cele 10 cifre şi primele şase litere ale alfabetului). Aceste simboluri alcătuiesc un şir de numere consecutive (0; 1=0+1; 2=1+1; 3=2+1; 4=3+1; 5=4+1; 6=5+1; 7=6+1; 8=7+1; 9=8+1; A=9+1; B=A+1; C=B+1; D=C+1; E=D+1; F=E+1).

Simbolul cu valoarea cea mai mare este F, el fiind şi este cu o unitate mai mic decât baza 16 (F = 16 – 1).

În tabelul 3 este prezentată modalitatea de numărare în baza 16, pentru numere de la 0 la 15 inclusiv.

Tabelul 3. Numărare în baza 16

Zecimal	Hexazecimal	Zecimal	Hexazecimal
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

Un număr scris în baza 16 va fi dezvoltat după puterile bazei astfel:

$N_{(16)} = \overline{a_n a_{n-1} a_{n-2} ... a_2 a_1 a_0} = a_n \times 16^n + a_{n-1} \times 16^{n-1} + a_{n-2} \times 16^{n-2} + \ldots + a_2 \times 16^2 + a_1 \times 16^1 + a_0 \times 16^0$.

Exemple de numere hexazecimale: $11_{(16)}$, $10A_{(16)}$, $AA00BB_{(16)}$, $FEDCBA10_{(16)}$.

Dacă se dezvoltă aceste numere după puterile bazei se obține:

$11_{(16)} = 1 \times 16^1 + 1 \times 16^0$;

$10A_{(16)} = 1 \times 16^2 + 0 \times 16^1 + A \times 16^0$;

$AA00BB_{(16)} = A \times 16^5 + A \times 16^4 + 0 \times 16^3 + 0 \times 16^2 + B \times 16^1 + B \times 16^0$;

$FEDCBA10_{(16)} = F \times 16^7 + E \times 16^6 + D \times 16^5 + C \times 16^4 + B \times 16^3 + A \times 16^2 + 1 \times 16^1 + 0 \times 16^0$.

În exemplele precedente s-a văzut că se poate reprezenta un număr în mai multe moduri, în funcţie de baza sistemului de numeraţie ales.

Astfel, numărul 12 din sistemul de numeraţie zecimal va fi:

► 1100 - sistemul de numeraţie binar;
► 14 - sistemul de numeraţie octal;
► C - sistemul de numeraţie hexazecimal.

3.2. Operaţiile aritmetice într-un alt sistem de numeraţie

Operaţiile aritmetice (adunarea, scăderea, înmulţirea, împărţirea) cu numere în sisteme de numeraţie diferite de baza 10 se efectuează la fel ca şi în bază 10 conform tabelelor de adunare şi înmulţire.

Pentru exemplificare, în tabelul 4 este prezentată tabela adunării în baza 2.

Tabelul 4. Tabela adunării în baza 2

+	0	1
0	0	1
1	1	10

Tot ca exemplu, în tabelul 5 este prezentată tabela înmulţirii în baza 2.

Tabelul 5. Tabela înmulţirii în baza 2

×	0	1
0	0	0
1	0	1

Tabelele 6 şi 7 prezintă aceleaşi operaţii, transpuse în sistemul hexazecimal.

Tabelul 6. Tabela adunării în baza 16

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Tabelul 7. Tabela înmulţirii în baza 16

×	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	7E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Iată un exemplu de operaţii de adunare şi de înmulţire în baza 2:

	1	*1*	*1*	*1*	*1*	← transport		
1	0	1	1	0	0	1	1	+
		1	1	1	1	1	0	
1	1	1	1	0	0	0	1	

		1	0	1 ×
		1	0	1
		1	0	1
1	0	1		
1	1	0	0	1

În continuare avem câte un exemplu de adunare şi înmulţire în baza 16:

1	1	1	← transport	
1	E	F	A	+
9	A	B	C	
B	9	B	6	1

		1	2	3 ×
			A	F
	1	1	0	D +
	B	5	E	
	C	6	E	D

3.3. Principiul reprezentării binare a informaţiei

Teoria informaţiei a arătat că orice informaţie, oricât de complexă ar fi, poate fi exprimată prin cumul de informaţii elementare, reprezentabile prin două stări complementare.

Dacă celor două variante posibile li se asociază cele două cifre binare 0 sau 1, înseamnă că informaţia va putea fi reprezentată din punct de vedere cantitativ în sistemul de numeraţie binar.

Pornind de la teoria informaţiei, s-a ajuns la concluzia că un sistem de calcul va trebui să utilizeze sistemul de numeraţie binar.

Informaţia elementară se mai numeşte şi **BIT (BInary digiT)**, adică una din cele două cifre binare 0 sau 1. *Bitul* reprezintă atomul informaţiei, nivelul de la care informaţia nu mai poate fi descompusă. Biţii se grupează câte 8 şi formează un BYTE sau un OCTET.

Informaţia, modelată şi manipulată de calculator, va fi deci un şir de cifre binare, reprezentarea făcându-se codificat. Cu ajutorul celor 8 biţi (8 cifre binare) care formează octetul se poate construi un cod, care permite 2^8 combinaţii diferite între ele, adică 256 combinaţii. Acestea sunt suficiente pentru a reprezenta literele mari şi mici, cifrele, semnele speciale, comenzile, semnalele, răspunsurile. Astfel, putem să privim octetul ca o "moleculă" de informaţie. Molecula poate să conţină o literă (A, a, …, Z, z), o cifră (0,1,2,….,9), un semn special (*.@,/,:,;,>,…), un semnal (semnal sonor,…), o comandă (retur de car la sfârşitul liniei de text, …). Cu aceste „molecule" se va agrega informaţia complexă, reprezentată de texte, numere, imagini grafice, sunete etc.

Octetul este o unitate de măsură a informaţiei. Pentru măsurarea diferitelor cantităţi de informaţie se pot folosi multiplii octetului. Unităţile de informaţie folosesc ca factor de multiplicare 2^{10}=1024:

1 Koctet = 2^{10} octeţi = 1024 octeţi;

1Moctet = 2^{10} octeţi = 2^{20} octeţi;

1Goctet = 2^{10} Mocteţi = 2^{20} Kocteţi = 2^{30} octeţi.

3.4. Coduri şi date

Orice informaţie (numere, litere, desene, sunete) şi orice instrucţiune pe care trebuie să o execute procesorul calculatorului trebuie să fie codificată într-o secvenţă de cifre binare. Detaliile prin care diferite tipuri de informaţii sunt codificate variază în funcţie de calculator.

Informaţia prelucrată de calculator se numeşte *dată*, reprezentând suportul informaţiei.

Data este un model de reprezentare a informaţiei, accesibil unui utilizator uman, unei unităţi de prelucrare a calculatorului sau unui program, model cu care se poate opera pentru obţinerea de noi informaţii.

Calculatorul este o maşină construită cu elemente fizice care prezintă numai două stări. El înţelege, manipulează şi prelucrează şiruri de cifre binare. Aceste cifre binare pot reprezenta semnale, comenzi (informaţie). Prin adoptarea sistemului de numeraţie binar s-au obţinut următoarele avantaje:

- ► S-a simplificat realizarea fizică a calculatorului;
- ► S-a redus procentul de erori din procesul de prelucrare şi stocare a datelor;
- ► S-a simplificat algoritmul de efectuare a operaţiilor aritmetice şi logice;
- ► S-a simplificat procedeul de transpunere a informaţiilor pe suporturile tehnice de informaţie.

Manifestarea externă a noţiunii de informaţie corespunde limbajului uman. Pentru reprezentare, se folosesc cele 10 cifre (0, 1 , 2,…. 9), cele 26 de litere mici (a – z), cele 26 litere mari (A – Z) şi diferite semne speciale (>, <, ?, \, …). Acest ansamblu de simboluri nu poate fi însă interpretat, prelucrat şi stocat de către calculator, care înţelege numai sistemul de numeraţie binar. Informaţia trebuie să fie transformată astfel încât să fie înţeleasă de către calculator. Ea trebuie adusă într-o formă binară.

Operaţia de transformare a informaţiei din forma de reprezentare externă (care este inteligibilă pentru om), în forma de reprezentare internă (pe care o poate înţelege calculatorul), se numeşte *codificare internă a informaţiei.*

Codul ASCII

Codul ASCII (American Standard Code for Information Interchange) este un cod pe 8 biţi, care a devenit codul calculatoarelor compatibile IBM-PC. El a derivat din codul ASCII pe 7 biţi care permitea numai $2^7 = 128$ cuvinte de cod. Mărindu-se lungimea cuvântului de cod la 8 caractere, s-a putut adăuga setul extins de caractere ASCII, set de încă 128 de caractere.

Codul ASCII este format din setul de caractere de bază (primele 128 de caractere având codurile hexazecimale 00h-7Fh), respectiv setul de caractere extins (ultimele 128 de caractere: 80h-FFh).

Setul de caractere de bază provine din codul ASCII pe 7 biţi şi cuprinde:

- ► 10 caractere pentru cifre (codurile 30h-39h);
- ► 26 caractere pentru literele mari ale alfabetului latin (codurile 41h-5Ahh);
- ► 26 caractere pentru litere mici ale alfabetului latin (codurile 61h-7Ah);
- ► 34 de semne speciale (20h-2Fh), 3Ah-40h, 5Bh-60h, 7Bh-7Fh);
- ► 32 de coduri pentru caracterele de control (00h-1Fh).

Setul de caractere extins cuprinde caracterele adăugate setului de bază pentru a îmbunătăţi codul, aici regăsindu-se următoarele:

- ► 48 de semne şi caractere alfabetice speciale, specifice unor limbi diferite de engleză (codurile 80h-AFh);
- ► 48 de caractere pentru construirea chenarelor (B0h-DFh), prin care se pot trasa liniile şi colţurile chenarelor în diferite maniere: cu linie simplă, cu linie dublă, cu linie îngroşată, cu linie cu o anumită densitate de gri etc;
- ► 32 caractere pentru grupul caracterelor ştiinţifice (E0h-FFh);
- ► literele alfabetului elen care sunt folosite în matematică şi în ştiinţe (E0h-EEh),
- ► simbolurile matematice speciale (EFh-FFh)
- ► caractere de control ASCII

3.5. Noţiuni de reprezentare a numerelor reale

Spre deosebire de numerele întregi, a căror reprezentare se face în mod facil (deoarece ele sunt caracterizate doar de semn şi valoarea numerică întreagă), numerele reale sunt reprezentate prin semn, valoarea părţii întregi, respectiv a părţii fracţionare. Codificarea numerelor reale este diferită, fiind adoptate convenţii de reprezentare în virgulă fixă şi virgulă mobilă.

Reprezentarea numerelor în virgulă fixă

În reprezentarea în virgulă fixă se presupune că partea întreagă este despărţită de partea fracţionară printr-o virgulă imaginară care se găseşte într-o poziţie fixă. În acest caz sunt fixe atât numărul de poziţii ale părţii întregi cât şi numărul de poziţii ale părţii fracţionare. Poziţia virgulei fixe este o caracteristică a tipului de calculator şi a modului în care este construit.

De exemplu, dacă se reprezintă pe 8 biţi un număr fracţionar cu numărul de poziţii întregi 5, automat numărul de poziţii zecimale va fi 3. În cazul în care numărul N care se reprezintă este pozitiv, domeniul de valori al datei va fi:

$00000.000_{(2)} \leq N \leq 11111.111_{(2)}$ adică:$0,0_{(10)} \leq N \leq 31.925_{(10)}$.

Dacă numărul este negativ, se va reprezenta prin complementul său faţă de 2, primul bit de semn şi domeniul de valori al datei va fi:

$11111.111_{(2)} \leq N \leq 01111.111_{(2)}$ adică $-16.925_{(10)} \leq N \leq 15.925_{(10)}$.

Această reprezentare a numerelor reale este dezavantajoasă deoarece nu permite decât reprezentarea unei game restrânse de numere reale. La unele calculatoare, poziţia virgulei este plasată la dreapta numărului, lucrându-se cu numărul real ca şi cum ar fi un număr întreg. Pentru a reprezenta întreaga valoare, în programul care exploatează numărul respectiv trebuie introdus un factor de scară.

Exemplu:

Numărul 12,34 va fi reprezentat ca un numărul întreg $12,34 \times 10^2 = 1234$, iar programul va trebui să actualizeze numărul folosind factorul de scară 10^2.

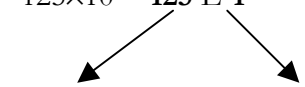
Reprezentarea numerelor în virgulă mobilă

În această reprezentare, numerele sunt reprezentate în format ştiinţific prin exponent şi mantisă.

Se ştie că orice număr poate fi scris explicitând diferite puteri ale lui 10 (exponenţi). În acest mod poate fi controlată poziţia virgulei zecimale, iar reprezentarea obţinută se

numeşte în "virgulă mobilă" deoarece virgula zecimală îşi schimbă poziţia în funcţie de valoarea exponentului.

De exemplu, reprezentarea în notaţie ştiinţifică a numărului zecimal 12,5 este:
$12.5=12.5\times10^{0}=0.125\times10^{2}=125\times10^{-1}=$**125** E**-1**



În mod analog se va reprezenta şi intern numărul, singura deosebire fiind codificarea exponentului şi a mantisei în sistemul binar. În plus, se vor folosi doi biţi pentru reprezentarea semnului mantisei şi a exponentului. Conform acestei convenţii, dacă se consideră reprezentarea în virgulă mobilă pe 32 de biţi, biţii vor fi în principiu folosiţi astfel:

► 1 bit pentru semnul numărului;
► 1 bit pentru semnul exponentului;
► 7 biţi pentru exponent;
► 23 biţi pentru mantisă.

Exemplu:
$12,5_{(10)}=1100,1_{(2)}=0,11001_{(2)}\times2^{4}=0,11001_{(2)}\times10_{(10)}^{100}$.

► mantisa este $11001_{(2)}$;
► exponentul este $4_{(10)}=100_{(2)}$;
► semnul numărului este pozitiv-0 (deci bitul de semn este 0);
► semnul mantisei este pozitiv-0 (bitul de semn fiind de asemena zero).

4. Desfăşurarea lucrării

► Să se efectueze următoarele operaţii de conversie:
a) $12_{(10)}\rightarrow ?_{(2)}\rightarrow ?_{(8)}\rightarrow ?_{(16)}$;
b) $ADE_{(16)}\rightarrow ?_{(2)}\rightarrow ?_{(10)}$.
► Să se efectueze următoarele operaţii în diferite baze de numeraţie:
a) $1111_{(2)}+0100_{(2)}$;
b) $10011_{(2)}+1100_{(2)}$;
c) $1EF_{(16)}+9AB_{(16)}$;
d) $1A_{(2)}\times2F_{(2)}$.
► Se propune un program de afişare a codurilor ASCII tipăribile, program ce va fi adus la forma executabilă:

```
.MODEL small              ;prin model vom intelege un mod de
                          ;dispunere in RAM a segmentelor
                          ;care alcătuiesc un program
.STACK 100h               ;o stiva de 256 (100 hexazecimal)
                          ;octeti este definita prin această
                          ;directiva
.DATA                     ;aceasta directiva marcheaza
                          ;inceputul segmentului de date
 Message DB ,13,10,'Program afisare cod ASCII',13,10,'$'
```

```
.CODE
   Start:
      mov ax,@data              ;incarca locatie
      mov ds,ax                 ;segment in registrul DS
      mov dx, offset Message    ;returneaza deplasamentul fata de
                                ;baza segmentului de date al DATA
      mov ah,9h                 ;incarcă AH cu 9 (codul functiei
                                ;afisare sir)
      int 21h                   ;apel rutina tiparire
     mov bl,48                  ;initializarea registrului BL cu
                                ;48
   Eticheta:
      mov ah,02h                ;incarca AH cu 2 (codul functiei
                                ;afisare caracter)
      mov dl,bl                 ;incarca registrul DL cu codul
                                ;caracterului de afisat
      int 21h
      inc bl                    ;incrementare a registrului BL
      cmp bl,127
      jb Eticheta               ;salt conditionat (dacă BL este
                                ;mai mic decat 127)
      mov ah,4ch                ;functia pentru terminarea
                                ;programului
      int 21h
   END Start
```

5. Modul de lucru

►Se vor rezolva exerciţiile propuse ca exemplu.
►Se va asambla, link-edita şi rula programul dat ca exemplu mai sus.
►Se va crea un program care încarcă în registrul AX numerele zecimale de la 0 la 15.
►Programul va fi executat pas cu pas în *TD* pentru a se vizualiza aceste numere în hexazecimal.

LUCRAREA 4

DEFINIREA ŞI INIŢIALIZAREA DATELOR. OPERATORI

1. Obiectivele lucrării

Obiectivul acestei lucrări este însuşirea modului de utilizare a tipurilor de date în diferite aplicaţii in limbaj de asamblare.

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar de laborator.

3. Breviar teoretic

3.1. Definirea şi utilizarea datelor simple

Asamblorul operează cu trei tipuri sintactice de bază: *constante*, *variabile* şi *etichete*. Ele sunt considerate atomi lexicali – identificaţi prin noţiunea de *date simple*, deoarece nu se pot identifica subdiviziuni ale acestora care să se constituie într-un tip sintactic distinct.

Constante

Constantele pot fi, la rândul lor, de două tipuri: constante simbolice şi constante non-simbolice.

Constantele simbolice se constituie în nume generice asociate unor entităţi valorice. Aceste valori pot reprezenta, de exemplu, un deplasament în cadrul unui segment, o constantă numerică, o adresă, un operand al unei instrucţiuni etc. Un nume simbolic (constantă simbolică) se defineşte ca o succesiune de simboluri literale, numerice şi caractere speciale, supuse unor reguli restrictive.

Iată câteva nume simbolice valide:

A2	Variabila	ConstDoi
Caz_5	Val3	__Eticheta
$7326	@Home	$_@1
Dolar$	EsteMaiMare?	@7326

Următoarele exemple de nume nu sunt însă acceptate de asambloare:

1ConstantaMea	– începe cu un caracter numeric;
Caz.Doi	– conţine un punct în interior;
$	– simbolul $ nu poate fi folosit singular;
LOCAL	– nume simbolic rezervat;
valoare 4 5	– conţine spaţii;
Hello,world!	– conţine caractere în afară de _, $ sau ?.

Constantele non-simbolice se caracterizează prin faptul că valoarea şi semnificaţia lor sunt conferite implicit de conţinut. Exemple de astfel de constante: 254, 3.1415927, "Sir De Caractere", 0AB25FF1h, 'M' etc.

Detaliind, constantele non-simbolice pot fi:

► constante binare, desemnate prin utilizarea sufixelor B sau b: 1010B, 11010010b etc.;

► constante octale, marcate cu sufixele O, o, Q sau q: 17O, 45o, 371Q, 77q etc.;

► constante zecimale întregi, la care se utilizează opţional sufixele D sau d: 2007, 1973D, 33d etc.;

► constante zecimale reale în format simplu (3.142, -0.456) sau ştiinţific (1e7, 1.56e-3, -3.23e+10);

► constante hexazecimale, ce utilizează cifrele de la 0 la 9, simbolurile literale A…F sau a…f şi sufixele H sau h, cu observaţia că, dacă prima cifră a numărului este mai mare decât 9, constanta este prefixată de cifra 0: 1Ah, 23FFH, 0AF2Bh, 45BFH etc.;

► constante tip şir de caractere ASCII, încadrate de semnele apostrof sau ghilimele: 'Text', "text 23".

Definirea constantelor simbolice reprezintă punerea în corespondenţă a acestora cu constantele non-simbolice, ce se realizează prin două forme sintactice de bază:

```
constanta_simbolica EQU expresie_non_simbolica
constanta_simbolica  =  expresie_non_simbolica
```

în care expresia non-simbolică este, de regulă, o valoare numerică sau un şir de caractere. *EQU* şi "=" reprezintă aşa-numitele *directive de definire a constantelor*.

Câteva exemple de definire a constantelor în limbaj de asamblare sunt prezentate în continuare.

```
const_bin1        EQU   1101b
const_octal1      EQU   134q
const_zecimal1    EQU   67
const_zecimal2    EQU   -222
const_zecimal3    EQU   1.1314
const_hex1        EQU   0FF25h
const_sir1        EQU   'Valoare:'
const_simplu1  =   24
const_simplu2  =   "B"
```

Variabile

Pentru definirea şi eventuala iniţializare a variabilelor se foloseşte următoarea construcţie sintactică:

```
nume_variabila  directiva  lista_de_valori
```

unde nume_variabila reprezintă identificatorul variabilei ce se defineşte.

Directiva de definire a variabilelor poate fi :

- ► DB, pentru definirea variabilelor BYTE (Define BYTE);
- ► DW, pentru WORD (Define WORD);
- ► DD, pentru DOUBLE-WORD (Define DOUBLE-WORD);
- ► DQ, pentru QUAD-WORD (Define QUAD-WORD);
- ► DT, pentru TEN-BYTES (Define TEN-BYTES).

lista_de_valori reprezintă lista valorilor iniţiale atribuite variabilei definite, ea putând conţine:

- ► constante absolute sau simbolice (anterior definite);
- ► simbolul "?", pentru locaţii declarate dar neiniţializate;
- ► o adresă (specificată prin numele unei variabile sau al unei etichete) – doar în cazul variabilelor WORD şi DOUBLE-WORD;
- ► un şir ASCII;
- ► operatorul DUP.

Dacă lista asociată definiţiei unei anume variabile conţine un singur element, acea variabilă este de tip *scalar*. Listele multiple, formate din elemente individuale separate prin virgule, definesc variabile *indexate*.

În cazul existenţei unor secvenţe repetitive în lista de valori, pentru simplificarea scrierii se foloseşte operatorul de duplicare DUP, cu următoarea formă generală:

```
n  DUP (expresie)
```

în care expresie poate fi, la rândul ei, o constantă absolută, o listă de valori, simbolul "?" sau un alt operator DUP. Semnificaţia operatorului este de duplicare de n ori a expresiei din paranteză.

În scop ilustrativ, următoarea secvenţă prezintă diferite declaraţii de variabile în limbaj de asamblare:

```
const_int   EQU 2Fh
const_real  EQU 3.14
const_sir   EQU 'abc'

var01  DB   70              ; Constanta absoluta
var02  DB   const_int       ; Constanta simbolica
var03  DB   'Sir'           ; Sir de caractere
var05  DB   ?               ; Declarare fara initializare
var06  DB   71,72,73        ; Lista valori
var08  DB   3 DUP (80)      ; Echivalent 80,80,80
var09  DW 0ABFFh            ; Constanta absoluta
var10  DW const_int         ; Constanta simbolica
var11  DW 2 DUP (1,2)       ; Echivalent 1,2,1,2
```

Variabilelor le sunt asociate o serie de atribute ce le caracterizează ca entităţi ale unui program în limbaj de asamblare:

► adresa de bază a segmentului de date ce conţine variabila;
► adresa relativă în cadrul segmentului (deplasamentul sau offset-ul);
► tipul variabilei (codificat numeric – 1, 2, 4, 8 sau 10 – în funcţie de numărul de octeţi folosiţi la reprezentare);

Aceste informaţii sunt considerate semnificative în anumite situaţii, programatorul putându-le accesa prin aplicarea operatorilor specializaţi ce vor fi prezentaţi în continuare.

Etichete

Etichetele sunt folosite în două situaţii:

►pentru specificarea alternativă a datelor (atunci când sunt plasate în zona de declaraţie a variabilelor);
►pentru specificarea punctelor de salt în cadrul unui program (caz în care ele sunt localizate la nivelul codului).

Principial, indiferent de destinaţie, o etichetă reprezintă asocierea dintre un nume simbolic şi adresa de memorie ce identifică locul în care este plasată eticheta (contorul locaţiei curente). Pentru uşurinţa înţelegerii se poate considera că şi numele variabilelor reprezintă etichete punctând adresa de memorie de la care acestea sunt stocate.

Declararea etichetelor face apel la directiva LABEL, având următoarea formă:

```
nume_eticheta  LABEL  tip
```

în care "nume_eticheta" este numele simbolic asociat etichetei, iar "tip" specifică tipul etichetei, dependent de destinaţia acesteia (pointer la date – BYTE, WORD, DWORD, QWORD, TBYTE – sau pointer la cod – NEAR, FAR).

Pentru exemplificare, în continuare sunt prezentate o serie de definiţii de etichete, atât ca pointeri la date, cât şi la cod.

```
label_byte  LABEL BYTE ; Pointer la date BYTE
label_word1 LABEL WORD ; Pointer la date WORD
...
salt1 LABEL NEAR       ; Pointer la cod, accesibil pentru
                       ; salturi doar din segmentul curent
                       ; de cod
salt1 LABEL FAR        ; Pointer la cod, accesibil pentru
                       ; salturi si din alte segmente de cod
```

Din punct de vedere informaţional, etichetele sunt caracterizate de trei atribute:

► adresa de bază a segmentului ce conţine eticheta;
► adresa relativă în cadrul segmentului (offset-ul);
► tipul etichetei (codificat numeric: 1 pentru BYTE, 2–WORD, 4–DWORD, 8–QWORD, 10–TBYTE, 254–FAR şi 255–NEAR).

3.2. Definirea şi utilizarea datelor compuse

Spre deosebire de datele simple, ce se caracterizează prin indivizibilitatea din punct de vedere lexical, datele compuse sunt agregate din alte entităţi – în general date simple.

Structuri. Operaţii asociate

Ca şi în cazul limbajelor de nivel înalt, în limbaj de asamblare structurile sunt colecţii de date (numite *câmpuri* sau *membri*) grupate sub un nume unic, desemnând o entitate sintactică.

Sintaxa de definiţie a unei structuri este

```
nume_structura  STRUC
     nume_membru_1  definitie_date
     nume_membru_2  definitie_date
     ...
     nume_membru_n  definitie_date
nume_structura  ENDS
```

în care se remarcă prezenţa delimitatorilor STRUC şi ENDS.

De regulă, prin această construcţie se defineşte doar tipul (şablonul) structurii, rezervarea efectivă a spaţiului de memorie făcându-se ulterior, în momentul declarării variabilelor ca având tipul "nume_structura".

Numele membrilor nume_membru_1,…nume_membru_n sunt distincte, chiar în situaţia în care câmpurile desemnate au tipuri diferite. În ceea ce priveşte definiţiile de date, acestea sunt cele uzuale, sub rezidenţa directivelor DB, DW, DD, DQ, DT, aşa cum au fost anterior expuse.

Se consideră următoarea definiţie de structură:

```
TEST_S STRUC
   a DB ?
   b DW ?
   c DD ?
TEST_S ENDS
```

TEST_S a devenit un nou tip de date, ce poate participa la definiţii standard, similar unei directive DB, DW etc. De exemplu, se poate defini o structură de tipul TEST_S, cu numele var_s:

```
var_s  TEST_S  <41h,4243h,44454647h>
```

Această construcţie echivalează conceptual cu succesiunea de definiţii

```
a DB 41h
b DW 4243h
c DD 44454647h
```

Există posibilitatea ca anumitor câmpuri (sau chiar tuturor) să le fie asociate valori implicite, în momentul definiţiei structurii, ca în exemplul următor:

```
TEST_S STRUC
   a DB 41h
   b DW 4243h
   c DD 44454647h
TEST_S ENDS
```

Dacă se doreşte menţinerea valorii implicite asociate unui membru particular al structurii, în poziţia corespunzătoare (între parantezele unghiulare < şi >) se lasă un loc liber:

```
var_s  TEST_S  < , ,0000FFFFh>
```

Accesul la membrii unei structuri se face similar cazului limbajelor de nivel înalt. De exemplu, încărcarea în registrul acumulator a câmpului b al structurii var_s se face prin apelul

```
mov ax,var_s.b
```

orice altă referire la variabilele var_s.a, var_s.b şi var_s.c fiind perfect validă.

Înregistrări. Operaţii asociate

Înregistrările sunt similare structurilor împachetate utilizate într-o serie de limbaje de nivel înalt. În cazul limbajului de asamblare, împachetarea se face pe formate rigide (cu un număr fix de biţi, multiplu de 8), singurul grad de libertate fiind stabilirea numărului de biţi alocaţi fiecărui element individual.

Sintaxa de declarare simplă (fără iniţializare implicită) a unei înregistrări este

```
nume_inreg RECORD n_camp1:lung1,n_camp2:lung2,...
```

în timp ce înregistrările cu iniţializare implicită au sintaxa

```
nume_inreg RECORD n_camp1:lung1=val1,n_camp2:lung2=val2,...
```

În construcţiile de mai sus, nume_inreg este numele sub care înregistrarea este identificată, n_camp1, n_camp2,… sunt numele atribuite câmpurilor individuale, lung1, lung2,… sunt lungimile (în biţi) ale câmpurilor asociate, iar val1, val2,… sunt eventualele expresii care iniţializează implicit membrii înregistrării. Obligatoriu, n_camp1, n_camp2,… trebuie să fie distincte în cadrul programului, chiar dacă fac parte din înregistrări diferite. Iată câteva exemple de declaraţii:

```
R1 RECORD a:4, b:4
R2 RECORD c:7, d:1
R3 RECORD e:3, f:2
R4 RECORD g:1=1,h:6=0Fh,i:4=0Fh
```

Modul în care aceste declaraţii conduc la formarea şabloanelor, la asamblarea standard (8086), este detaliat în figura 1. Se remarcă extensia la valori fixe: 8 biţi în cazul lui R3, respectiv 16 biţi în cazul lui R4. Similar cazului structurilor, definirea înregistrărilor nu are ca efect alocarea de spaţiu în memorie, ci doar crearea şabloanelor. Se pot defini însă acum variabile de tip R1, R2, R3 sau R4, conform construcţiei următoare:

```
var1 R1 <0Fh,00h>
var2 R4 < , , >
```

în care valorile dintre paranteze unghiulare au rolul de iniţializare a câmpurilor de biţi.

Cele două definiţii anterioare sunt echivalente cu definiţiile explicite

```
var1 DB 11110000b
var2 DB 0000010011111111b
```

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R1									a_3	a_2	a_1	a_0	b_3	b_2	b_1	b_0
R2									c_6	c_5	c_4	c_3	c_2	c_1	c_0	d_0
R3												e_2	e_1	e_0	f_1	f_0
R4						g_0	h_5	h_4	h_3	h_2	h_1	h_0	i_3	i_2	i_1	i_0
(R4)	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1

Figura 1. Şabloane pentru înregistrările **R1**, **R2**, **R3** şi **R4**

Deoarece agregarea de tip înregistrare permite lucrul cu entităţi (câmpuri) reprezentate pe un număr variabil de biţi, limbajul de asamblare prevede doi operatori specifici ce permit programatorului manipularea facilă a membrilor înregistrărilor.

Operatorul WIDTH se aplică unui nume de înregistrare sau unui nume de câmp dintr-o înregistrare, întorcând numărul *efectiv* de biţi ai înregistrării (fără extensie la 8 sau 16 biţi), respectiv ai câmpului. De exemplu, secvenţa de definiţii

```
numar_b1 DB WIDTH a
numar_b2 DB WIDTH R3
```

iniţializează variabilele numar_b1 cu 4 şi numar_b2 cu 5.

Operatorul MASK primeşte un nume de înregistrare sau de câmp al unei înregistrări, furnizând o valoare numerică (numită mască) având 1 în poziţiile biţilor efectiv ocupaţi de înregistrare, respectiv de câmpul desemnat, şi 0 în rest.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R3												e_2	e_1	e_0	f_1	f_0
MASK R3									0	0	0	1	1	1	1	1
MASK e									0	0	0	1	1	1	0	0
R4						g_0	h_5	h_4	h_3	h_2	h_1	h_0	i_3	i_2	i_1	i_0
MASK h	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0

Figura 2. Aplicarea operatorului **MASK**

Folosind definițiile anterioare ale înregistrărilor R3 şi R4, prin aplicarea în diferite situații a operatorului MASK avem următoarele valori returnate:

MASK R3 → 00011111b (1Fh)
MASK e → 00011100b (1Ch)
MASK h → 0000001111110000b (3F0h)

Pentru o mai bună înțelegere, figura 2 detaliază modul în care s-au obținut aceste valori numerice.

3.3. Operatori uzuali

Limbajul de asamblare recunoaşte un set de operatori (aritmetici, logici, relaționali, de tip etc.) prin intermediul cărora se pot forma expresii ale căror valori sunt evaluate *în momentul asamblării* sub forma unor constante. În niciun caz aceşti operatori nu trebuie confundați cu instrucțiunile microprocesorului, care realizează o evaluare a valorilor (numerice, logice etc.) *în timpul execuției programului*.

Tabelul 1 prezintă o prezentare sistematizată a operatorilor (în ordine descrescătoare a precedenței).

Tabelul1. Operatori standard

Operator	**Sintaxă**	**Semnificație** Descriere
1	2	3
( )	**(expresie)**	**Operator de prioritate.** Marchează o expresie pentru a fi evaluată cu prioritate.
[ ]		**Operator de index sau referință la memorie.**
LENGTH	**LENGTH nume**	**Operator de tip.** Se aplică doar variabilelor. Returnează numărul de elemente pentru variabilele vectoriale. În cazul variabilelor scalare, returnează valoarea **1**.
MASK	**MASK inregistrare** **MASK camp_inreg**	**Operator specific înregistrărilor.** Aplicat unui nume de înregistrare returnează o valoare numerică având valoarea **1** în poziția biților efectiv ocupați de aceasta. În cazul câmpurilor, întoarce o valoare numerică având valoarea **1** în poziția biților ocupați de câmpul implicat.
SIZE	**SIZE nume**	**Operator de tip.** Se aplică doar variabilelor. Întoarce dimensiunea (globală, în cazul variabilelor indexate) în octeți, pentru acestea.
WIDTH	**WIDTH inregistrare** **WIDTH camp_inreg**	**Operator specific înregistrărilor.** Aplicat unei înregistrări întoarce numărul efectiv de biți ai acesteia (fără extensie la 8 sau 16 biți). În cel de-al doilea caz întoarce numărul efectiv de biți ocupați de câmp (specificat prin nume).
HIGH	**HIGH expresie**	**Operator de tip.** Se aplică variabilelor de tip WORD. Returnează octetul mai semnificativ.

1	2	3
LOW	**LOW expresie**	**Operator de tip.** Se aplică variabilelor de tip WORD. Returnează octetul mai puţin semnificativ.
+	**+ expresie**	**Operator aritmetic.** Este opţional, indicând faptul că expresia precedată este pozitivă.
-	**- expresie**	**Operator aritmetic.** Schimbă semnul expresiei precedate.
OFFSET	**OFFSET expresie**	**Operator de tip.** Se aplică variabilelor şi etichetelor. Furnizează adresa relativă a unei astfel de entităţi faţă de baza segmentului de date/cod în care este definită (adresa relativă a locaţiei de unde începe stocarea sa în memorie).
PTR	**tip PTR expresie**	**Operator de tip.** Se aplică variabilelor şi etichetelor. Forţează evaluarea expresiei la tipul indicat (**BYTE**, **WORD**, **DWORD**, **QWORD**, **TBYTE**, **NEAR**, **FAR**).
SEG	**SEG expresie**	**Operator de tip.** Se aplică variabilelor şi etichetelor. Furnizează adresa de segment a expresiei (adresa bazei segmentului de date/cod în care este definită).
THIS	**THIS tip**	**Operator de tip.** Creează un operand având asociate o adresă de segment şi un deplasament (în cadrul acestuia) identice cu cele ale locaţiei în care se declară operandul respectiv.
TYPE	**TYPE expresie**	**Operator de tip.** Se aplică variabilelor şi etichetelor. Pentru variabile simple (şi etichete pointer la date) întoarce tipul acestora (numărul de octeţi pe care sunt reprezentate: **1**, **2**, **4**, **8** sau **10**). Pentru variabile tip structură întoarce numărul de octeţi pe este reprezentat în element al structurii. Pentru etichete pointer la cod întoarce tipul acestora (**FFh** sau **FEh**).
*****	**expr1 * expr2**	**Operator aritmetic.** Înmulţeşte două expresii evaluate la întreg.
/	**expr1 / expr2**	**Operator aritmetic.** Împarte două expresii evaluate la întreg.
MOD	**expr1 MOD expr2**	**Operator aritmetic.** Evaluează restul împărţirii a două expresii evaluate la întreg.
SHL	**expresie SHL numar**	**Operator logic.** Deplasează spre stânga (SHift Left) valoarea binară a expresiei, cu numărul de poziţii indicate. Dacă numărul este negativ, deplasarea se face la dreapta.
SHR	**expresie SHR numar**	**Operator logic.** Deplasează spre dreapta (SHift Right) valoarea binară a expresiei, cu numărul de poziţii indicate. Dacă numărul este negativ, deplasarea se face la stânga.
+	**expr1 + expr2**	**Operator aritmetic.** Adună două expresii evaluate la întreg.

1	2	3
-	**expr1 - expr2**	**Operator aritmetic.** Face diferența a două expresii evaluate la întreg.
EQ	**expr1 EQ expr2**	**Operator relațional.** Dacă expresiile sunt egale întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **EQ**.
GE	**expr1 GE expr2**	**Operator relațional.** Dacă prima expresie este mai mare sau egală cu cea de-a doua întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **GE**.
GT	**expr1 GT expr2**	**Operator relațional.** Dacă prima expresie este strict mai mare decât cea de-a doua întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **GT**.
LE	**expr1 LE expr2**	**Operator relațional.** Dacă prima expresie este mai mică sau egală cu cea de-a doua întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **LE**.
LT	**expr1 LT expr2**	**Operator relațional.** Dacă prima expresie este strict mai mică decât cea de-a doua întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **LT**.
NE	**expr1 NE expr2**	**Operator relațional.** Dacă expresiile sunt diferite întoarce TRUE (valoare binară având toți biții de valoare **1**), în caz contrar returnează FALSE (toți biții **0**). Numărul de biți depinde de tipul la care este evaluată expresia ce conține operatorul **NE**.
NOT	**NOT expresie**	**Operator logic.** Complementează bit cu bit valoarea binară a expresiei.
AND	**expr1 AND expr2**	**Operator logic.** Execută operația ȘI logic, la nivel de bit.
OR	**expr1 OR expr2**	**Operator logic.** Execută operația SAU logic, la nivel de bit.
XOR	**expr1 XOR expr2**	**Operator logic.** Execută operația SAU-EXCLUSIV logic, la nivel de bit.
LARGE	**LARGE expresie**	**Operator de tip.** Evaluează deplasamentul expresiei la o valoare pe 32 de biți (operație validă doar dacă la asamblare se generează cod compatibil cu procesorul 80386).

1	2	3
SHORT	**SHORT expresie**	**Operator de tip.** Forţează evaluarea expresiei la un pointer de instrucţiuni de tip scurt (fiind posibile salturi în memoria de program de maxim –128…+127 octeţi faţă de locaţia curentă).
SMALL	**SMALL expresie**	**Operator de tip.** Evaluează deplasamentul expresiei la o valoare pe 16 de biţi (operaţie validă doar dacă la asamblare se generează cod compatibil cu procesorul 80386).
.TYPE	**.TYPE expresie**	**Operator de tip.** Returnează un octet a cărui valoare indică tipul expresiei (variabilă, etichetă sau nume de structură.

4. Desfăşurarea lucrării

Se consideră următorul exemplu de program care ilustrează utilizarea unor directive de declaraţii a datelor, precum şi a unor operatori elementari:

```
.model small
.stack 100h
.data
  m dd 1h
  n dw 2h
  p db 3h
  x db 32 dup(?)
  y dw 16 dup(?)
.code

START:
 mov ax, @data
 mov ds, ax

 A EQU 2
 B EQU 3

 mov ax, (A+B) SHL 3          ;101000b=28h
 mov ax, 32 MOD (A*B)         ;32 mod 6 = 2

 mov ax, A XOR B              ;10b xor 11b = 01b
 mov ax, NOT (A OR B)         ;1111111111111100b=FFFCh

 mov ax, 4 LE (A+B)           ;FFFFh
 mov ax, A EQ B               ;0
 mov al, B GT A               ;FFh

 mov ax, ds
 mov bx, SEG m                ;AX=BX
 mov cx, OFFSET p             ;dd 4 bytes + dw 2 bytes = 6

 mov ax, type m               ;4
 mov bx, type n               ;2
```

```
    mov cx, type p              ;1

    mov ax, LENGTH x            ;32=20h
    mov bx, SIZE y              ;32=20h

    mov ax, 1234h
    mov WORD PTR m, ax          ;m=00001234h
    mov BYTE PTR n, al          ;n=0034h
    mov BYTE PTR n + 1, ah      ;n=1234h

    mov     ax, 4C00h
    int     21h

end     START
```

Se cere aducerea sa la forma executabilă şi analiza acestei forme cu ajutorul Turbo-Debugger.

5. Modul de lucru

- ► Se va asambla şi link-edita programul prezentat ca exemplu, obţinându-se fişierul *.EXE* asociat.
- ► Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
- ► Se va rula sub *TD* programul.
- ► Se propune realizarea unui program în care să se regăsească un număr cât mai mare de definiţii de date, conform celor prezentate în breviarul teoretic. Pentru acest program se recomandă analiza executabilului generat, în *TD*.

LUCRAREA 5

STRUCTURA PROGRAMELOR. SEGMENTE ŞI MODELE DE MEMORIE

1. Obiectivele lucrării

Lucrarea de faţă îşi propune însuşirea modului de gestiune a segmentelor aferente unui program scris în limbaj de asamblare.

2. Aparatura şi suporturile utilizate

- ► PC în configuraţia unitate centrală, monitor, tastatură, mouse.
- ► Precizările din prezentul îndrumar.

3. Breviar teoretic

Pe lângă instrucţiunile propriu-zise, un program ASM poate cuprinde directive, prin intermediul cărora se definesc date, etichete şi proceduri, se structurează segmente, se definesc şi se utilizează macroinstrucţiuni, se controlează în general procesul de asamblare etc. Directivele nu reprezintă instrucţuni, ci comenzi către asamblor, efectul lor manifestându-se exclusiv în faza de asamblare.

Un program realizat în limbaj de asamblare poate fi privit ca fiind alcătuit din unul sau mai multe *module*. Simbolurile definite (etichete, variabile, proceduri etc.) într-un modul şi referite în alte module se declară PUBLIC în modulul de definiţie şi EXTRN în cele în care sunt referite.

Indiferent de modul de dezvoltare a unui program, atât instrucţiunile, cât şi datele trebuie să se găsească în interiorul unor *segmente*. Prin definiţie, un *segment* este o colecţie de instrucţiuni sau date ale căror adrese se calculează relativ la începutul segmentului. Segmentele pot conţine date, cod sau pot fi atribuite stivei. Pentru ca procesorul să localizeze informaţia în segmentele de cod, date şi stivă, la nivelul acestuia se regăsesc registrele speciale d segment, ce memorează adresele de început ale segmentelor referite.

Astfel, cele patru registre segment ale procesorului standard 80x86 sunt:

- ► CS (Code Segment – pentru segmentul care conţine codul programului);
- ► DS (Data Segment – pentru segmentul care conţine datele programului);
- ► ES (Extra Segment – pentru un eventual segment suplimentar de date);
- ► SS (Stack Segment – pentru segmentul în care se realizează operaţiile cu stiva).

Cu cele patru registre de segment principale se pot referi patru segmente logice ce pot corespunde la patru segmente fizice distincte, dar pot exista şi suprapuneri parţiale sau totale ale acestor segmente.

Definirea segmentelor în programele scrise în limbaj de asamblare se poate face în formă completă, respectiv în formă simplificată.

3.1. Forma completă de definire a segmentelor

Definirea extensivă a segmentelor se face cu ajutorul directivei SEGMENT ce permite specificarea următoarelor atribute:

- ► numele segmentului;
- ► combinarea cu alte segmente;
- ► alinierea;
- ► continuitatea (adiacența) segmentelor.

Forma generală a directivei SEGMENT este

```
nume SEGMENT [tip_aliniere] [tip_combinare] ['nume_clasa']
        .
        .
nume ENDS
```

Inițializarea unui registru segment (DS, ES sau SS) cu adresa de bază a unui segment declarat revine utilizatorului, care o va face în mod explicit în cadrul programului, utilizând pentru aceasta numele segmentului respectiv, conform construcției sintactice următoare:

```
mov  ax, <nume_segment>
mov  ds, ax          ; în mod asemănător pentru ES sau SS
```

Parametrii incluși în paranteze drepte sunt opționali la declararea segmentelor. Dacă sunt prezenți, aceştia trebuie să se regăsească în ordinea specificată.

Parametrul *tip_aliniere* specifică la ce tip de adresă limită va fi încărcat (relocat) segmentul în memorie.

Poate avea una din valorile:

- ► PARA (implicit) - aliniere la paragraf (segmentul fizic va fi relocat la prima adresă absolută divizibilă prin 16);
- ► BYTE - fără aliniere (segmentul se va reloca la următorul octet liber);
- ► WORD - aliniere la cuvânt (segmentul se va reloca la următorul octet liber aflat la adresă pară);
- ► DWORD - aliniere la dublu-cuvânt (segmentul se va reloca la prima adresă divizibilă cu 4);
- ► PAGE - aliniere la pagină (segmentul se va reloca la prima adresă absolută divizibilă prin 256).

Parametrul *tip_combinare* specifică dacă şi cum se va combina segmentul respectiv cu alte segmente, în urma operației de link-editare.

Valorile acestui parametru pot fi:

- ► necombinabil (implicit) - nu se scrie nimic;
- ► PUBLIC - segmentul curent va fi concatenat cu alte segmente având acelaşi nume şi atributul PUBLIC. Aceste segmente pot fi în alte module de program. Se va forma, deci, un singur segment final cu numele respectiv, cu o unică adresă de

început; lungimea unui segment cu atributul PUBLIC este suma lungimilor segmentelor cu acelaşi nume;

- ► COMMON - specifică faptul că segmentul curent şi toate segmentele cu acelaşi nume şi tipul COMMON se vor suprapune în memorie (încep la aceeaşi adresă fizică); lungimea unui segment COMMON este cea mai mare lungime a segmentelor componente;
- ► STACK - marchează segmentul curent ca reprezentând stiva programului; dacă sunt mai multe segmente cu tipul STACK, ele vor fi tratate în cazul PUBLIC;
- ► AT EXPRESIE - specifică faptul că segmentul va fi plasat la o adresă fizică absolută de memorie;

Parametrul '*nume_clasă*' - specifică un nume de clasă pentru segment, extinzând astfel numele segmentului; de exemplu, dacă segmentul are si atributul 'nume_clasă', atunci atributele COMMON sau PUBLIC vor acţiona numai asupra segmentelor cu acelaşi nume şi acelaşi nume de clasă; ca nume de clase, se folosesc de obicei 'code', 'data' şi 'stack'.

Următorul exemplu ilustrează definirea şi iniţializarea explicită a unui segment de stivă, respectiv a unui segment de cod:

```
stiva       SEGMENT STACK
            db    256 dup(?)
            stiva_sp label WORD
stiva       ENDS
cod   SEGMENT
            mov   ax, stiva
            mov   ss, ax
            mov   sp, OFFSET stiva_sp
cod   ENDS
```

Directiva ASSUME

Directiva ASSUME este o pseudoinstrucţiune pentru care nu se generează cod, rolul său fiind numai de a informa asamblorul care este intenţia programatorului. Asamblorul nu are nici un fel de mecanism prin care să verifice dacă într-adevăr conţinutul registrului segment este cel declarat în pseudoinstrucţiunea ASSUME. În schimb, această pseudoinstrucţiune este utilizată de asamblor pentru a face referire la date şi instrucţiuni. Când o instrucţiune face referire la o variabilă, fară prefix de segment, asamblorul determină segmentul ce conţine variabila şi apoi examinează pseudoinstrucţiunea ASSUME, pentru a determina registrul ce adresează acel segment.

Directiva are forma generală

```
ASSUME reg_seg : expresie, . . . ;
```

unde reg_seg poate fi:

- ► un nume de segment;
- ► un nume de grup de segmente;
- ► SEG nume segment;
- ► NOTHING.

În aceeaşi directivă ASSUME se pot asocia mai multe registre de segment, de exemplu:

```
ASSUME CS:CSEG, DS:DSEG1, ES:DSEG2, SS:SSEG
```

Directiva ASSUME nu suplineşte de încărcarea registrelor de segment cu adresele de segment. Dacă un nume de segment nu apare într-o directivă ASSUME, datele definite în acel segment pot fi accesate numai prin prefixe de segment.

Directiva GROUP

Directiva *GROUP* serveşte la gruparea mai multor segmente (inclusiv cu nume diferite şi atribute diferite), a căror lungime totală nu depaşeşte 64 Ko, sub acelaşi nume. Are forma generală:

```
nume_grup GROUP nume_seg1, nume_seg2, ...
```

Această directivă reprezintă o altă modalitate de combinare a mai multor segmente, pe lângă cea oferită de atributul PUBLIC. Numele de grup se poate folosi în directiva ASSUME, la iniţializarea unor registre de segment sau la calculul unui offset. Dacă se consideră următorul exemplu:

```
d1 SEGMENT
y       dw    20
d1      ENDS
d2      SEGMENT
x       db    2
d2      ENDS
data    GROUP       d1, d2
```

atunci se pot scrie instrucţiuni de forma

```
mov     ax, data
mov     ds, ax
mov     ax, OFFSET x
mov     ax, OFFSET data:x
```

sau directive de forma

```
ASSUME      ds:data
```

Expresia OFFSET x furnizează offset-ul variabilei x în cadrul segmentului d1, iar expresia OFFSET data:x produce offset-ul variabilei x în cadrul grupului de segmente data.

3.2. Definirea simplificată a segmentelor. Modele de memorie

Această modalitate de definire este introdusă în variantele relativ recente ale asambloarelor. Avantajul major este faptul că se respectă acelaşi format (structură a programului obiect) ca şi la programele dezvoltate în limbaj de nivel înalt. Concret, la definirea simplificată se vor genera segmente cu nume şi atribute identice cu cele generate de compilatoarele de limbaje de nivel înalt.

De aceea, forma simplificată de definire a segmentelor are avantajul, din punct de vedere al utilizatorului, că oferă o gestiune mai simplă a segmentelor.

În această abordare, un program are atribuit un segment de cod (definit cu directive .CODE), un segment de date (definit cu .DATA) şi un segment de stivă (definit cu .STACK).

Formele generale ale acestor directive sunt:

► *.stack*[*dimensiune*]

Această directivă alocă o zonă de memorie de dimensiune specificată pentru segmentul de stivă. Dacă nu se specifică parametrul *dimensiune,* acesta va fi implicit de 1Koctet.

► *.code*[*nume*]

Acestă directivă precede segmentul de cod (program) căruia i se poate opţional atribui un nume.

► *.data*

Această directivă desemnează segmentul de date pentru care utilizatorul trebuie să iniţializeze explicit registrul de segment DS, cu adresa segmentului de date, @DATA.

Adresele de început ale segmentelor şi ale grupurilor de segmente definite cu directivele simplificate sunt disponibile prin simbolurile globale @data, @data?, @fardata, @fardata? şi dgroup.

La încărcarea în memorie a unui program pentru execuţie, sistemul de operare iniţializează registrul de segment CS cu prima adresă de segment disponibilă, iar registrul IP (Instruction Pointer – indicator al locaţiei de memorie în care se află instrucţiunea curentă) cu adresa relativă din cadrul segmentului a primei instrucţiuni ce trebuie executată. Pentru registrele DS şi SS, în principiu, acestea trebuie în mod explicit încărcate de utilizator.

Modul în care sunt organizate în memorie segmentele generate cu directivele de definire simplificată este controlat prin intermediul aşa numitelor modele de memorie.

Directiva pentru specificarea modelului de memorie are forma generală

```
.model tip
```

în care tip poate fi *tiny, small, medium, large* sau *huge*. Această directivă este în mod necesar plasată la începutul codului sursă. Semnificaţia acestor tipuri este :

► *tiny* - toate segmentele (date, cod, stivă) se pot genera într-un spaţiu de 64 Ko şi formează un singur grup de segmente. Se foloseşte la programele de tip .COM. Toate salturile, apelurile şi definiţiile de proceduri sunt implicit de tip NEAR;

► *small* - datele şi stiva sunt grupate într-un singur segment, iar codul în alt segment. Fiecare din acestea nu trebuie să depăşească 64 Ko. Toate salturile, apelurile şi definiţiile de proceduri sunt implicit de tip NEAR;

► *medium* - datele şi stiva sunt grupate într-un singur segment (cel mult egal cu 64 Ko), dar codul poate fi în mai multe segmente separate (nu se grupează), deci poate depăşi 64 Ko. Salturile şi apelurile sunt implicit tip FAR ca şi definiţiile de proceduri;

► *compact* - codul generat ocupă cel mult 64 KO (se grupează), dar datele şi stiva sunt în segmente separate (pot depăşi 64 KO). Apelurile şi salturile sunt implicit de tip NEAR. Se utilizează adrese complete (formate din segment şi offset) atunci când se accesează date definite în alte segmente;

► *large* - atât datele, cât şi codul generat pot depăşi 64 Ko, structurile de date neputând depăşi 64Ko;

► *huge* - este asemănător modelului large, dar structurile de date pot depăşi 64 Ko.

Folosirea directivelor simplificate prezintă si avantajul că nu mai sunt necesare directive ASSUME, deoarece asocierile între segmentele generate şi registrele de segment sunt implicite.

4. Desfăşurarea lucrării

După citirea cu atenţie a breviarului teoretic, se vor realiza următoarele programe în TASM, ilustrând cele două moduri de declarare ale unui segment.

► Forma completă:

```
date segment word public 'data'
mesaj  db    "Salut!",13,10,'$'
lmesaj    EQU  $-mesaj         ;in Dos este echivalent cu
                               ;13,10
date ends
cod segment word public 'code'
assume cs : cod, ds : date, ss : stiva
start:
       mov  ax,date            ;initializare registru segment
       mov  ds,ax              ;pentru date DS
       mov  bx,1
       mov  cx,lmesaj
       mov  dx,OFFSET mesaj    ;initializare adresa mesaj
                               ;in DX
       mov  ah,40h             ;afiseaza continutul
                               ;variabilei de tip
                               ;sir de caractere

       int  21h                ;revenire în DOS

       mov  ax,4C00h
       int  21h
cod ends
       stiva segment word stack ‘stack'
       dw 10 dup (?)          ;rezervare memorie pentru stiva
       stiva  ends
End start
```

► Forma simplificată:

```
.model small                    ;initializare model memorie
.stack 100h                     ;defineste stiva
.data
mesaj  db  "Salut!",13,10,'$'
lmesaj EQU  $-mesaj


.code
Start:
       mov ax , @data           ;inițializare registru segment
       mov ds , ax              ;pentru date DS
       mov bx , 1
       mov cx , lmesaj
       mov dx , OFFSET mesaj    ;inițializare adresă mesaj
                                ;în DX
       mov ah , 40h             ;afiseaza continutul
                                ;variabilei de tip
                                ;sir de caractere
       mov ah , 9               ;apel funcție 9-DOS,
                                ;de tipărire
                                ;a unui text, cu adresa în DX
       int 21h
       mov ax , 4c00h           ;revenire în DOS
       int 21h
End start
```

5. Modul de lucru

► Se vor asambla şi link-edita programele prezentate ca exemplu, obţinându-se fişiere *.EXE*.

► Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.

► Se vor rula sub *DEBUG* sau *TD* programele prezentate.

► Pentru cazul definirii simplificate a segmentelor se vor testa diverse modele de memorie, pentru care se vor observa atât dimensiunea executabilelor generate, cât şi maniera de alocare a memoriei.

LUCRAREA 6

ÎNTRERUPERI BIOS

1. Obiectivele lucrării

Lucrarea urmăreşte prezentarea principială a modului de lucru cu întreruperile pentru microprocesoarele 80x86, punându-se accent pe evidenţierea unor tehnici de implementare la programele în limbajde asamblare ce folosesc rutine ale BIOS. Se are în vedere interacţiunea utilizatorului cu sitemul de intrare/ieşire (adaptor video, tastatură etc.).

2. Aparatura şi suporturile utilizate

► PC în configuraţie unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

3.1 Noţiuni generale

BIOS (Basic Input/Output System) reprezintă nivelul situat la baza ierarhiei resurselor software, interacţionând nemijlocit cu structura hardware a calculatorului. El este practic o colecţie de proceduri de sistem, parte dintre acestea fiind disponibile pentru apelare în programele-utilizator, *indiferent de sistemul de operare*.

Mecanismul prin care o aplicaţie poate invoca serviciile astfel puse la dispoziţie este cel al *întreruperilor software*. În principiu, funcţii similare sunt grupate pe *niveluri* (BIOS-ului fiindu-i caracteristice nivelurile de întrerupere 10h…1Fh şi 40h…5Fh). Suplimentar identificării prin nivel, fiecare funcţie şi/sau sub-funcţie este caracterizată de un cod numeric propriu şi de registrele prin intermediul cărora i se transmit parametrii de intrare, respectiv de registrele în care se vor regăsi rezultatele la ieşire.

Secvenţa de acţiuni corespunzătoare apelării unui serviciu BIOS poate fi sintetizată sub următoarea formă:

► se încarcă argumentele funcţiei în registrele special destinate acestui scop;
► se încarcă în registrul ah codul numeric al funcţiei;
► se încarcă (de obicei în registrul al) un eventual cod de sub-funcţie;
► se lansează cererea de întrerupere pe nivelul specific, prin intermediul instrucţiunii INT;
► se preiau eventualii parametri de ieşire din registrele de rezidenţă asociate funcţiei apelate, în vederea folosirii lor ulterioare.

Pentru exemplificare, tabelul 1 prezintă o serie de servicii BIOS tipice. Sunt precizate, în acest context, nivelele de întrerupere caracteristice, codurile uzuale, denumirea standard şi semnificaţia fiecărei astfel de funcţii.

Tabelul 1. Servicii BIOS

Tip serviciu BIOS	Nivel de întrerupere
Servicii asociate adaptorului video	INT 10h
Servicii asociate tastaturii	INT 16h
Servicii asociate unităţilor de disc	INT 13h
Servicii asociate unităţilor de disc	INT 13h
Servicii asociate imprimantei	INT 17h
Servicii asociate porturilor seriale de comunicaţii	INT 14h
Servicii ale ceasului de timp real	INT 1Ah
Servicii sistem	INT 12h
Servicii sistem	INT 19h

3.2. Servicii BIOS uzuale

În continuare vor fi detaliate o serie de funcţiile frecvent utilizate în cadrul programelor de aplicaţie scrise în limbaj de asamblare.

Servicii video BIOS (INT 10h)
Terminalul grafic poate lucra atât în mod alfanumeric cât şi în mod grafic.
Modurile de lucru alfanumerice sunt caracterizate de următoarele elemente:

- numărul de linii-caracter (25);
- numărul de coloane-caracter (40 sau 80);
- atribuitele de culoare;
- numărul de pagini video.

În contextul acestui mod de lucru, un caracter afişat pe ecran este specificat prin codul ASCII asociat, respectiv prin atributul de culoare.

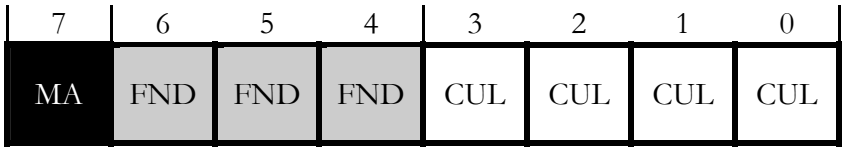


Figura 1. Codificarea atributului de culoare pentru modul alfanumeric
al adaptorului video: CUL – biţii culorii caracterului;
FND – biţii culorii de fond; MA – bitul modului de afişare.

Codificat sub forma unei entităţi pe 8 biţi, la nivelul acestui atribut se identifică trei zone distincte, prezentate în figura 1:

- culoarea caracterului – biţii CUL de rang 3, 2, 1 şi 0;
- culoarea de fond– biţii FND de rang 6, 5 şi 4;
- modul de afişare MA (continuă/intermitentă) – bitul de rang 7.

Tabelul 2 prezintă codurile culorilor. Deoarece culoarea de fond este reprezentată doar pe cei 3 biţi FND, acesteia îi pot fi atribuite doar primele 8 valori numerice, de la 0h la

7h (culorile închise). În ceea ce priveşte bitul MA, dacă valoarea sa este 0 (zero) caracterul va fi afişat normal, în timp ce valoarea 1 conduce la afişarea intermitentă a acestuia.

Tabelul 2. Codificarea culorilor (pentru modul alfanumeric)

Cod	**Culoare**	**Valabil pentru**	**Cod**	**Culoare**	**Valabil pentru**
0h	negru	CUL, FND	8h	gri	CUL
1h	albastru	CUL, FND	9h	albastru intens	CUL
2h	verde	CUL, FND	Ah	verde intens	CUL
3h	cyan	CUL, FND	Bh	cyan intens	CUL
4h	roşu	CUL, FND	Ch	roşu intens	CUL
5h	magenta	CUL, FND	Dh	magenta intens	CUL
6h	maro	CUL, FND	Eh	galben	CUL
7h	alb	CUL, FND	Fh	alb intens	CUL

Modurile de lucru în regim grafic se caracterizează în principal prin rezoluția orizontală şi verticală (exprimată în pixeli), respectiv prin atributele de culoare asociate, toate acestea fiind însă într-o puternică dependență față de capabilitățile adaptorului grafic.

Apelul serviciilor BIOS video se face prin încărcarea registrelor specifice (cu valorile considerate parametri de intrare), lansarea unei cereri de întrerupere software pe nivelul 10h, pentru ca ulterior, după predarea controlului de către rutina de tratare a întreruperii, valorile returnate în registre să fie corespunzător exploatate.

Pentru sistematizare, tabelul 3 oferă o viziune de ansamblu, sintetică, asupra principalelor funcții video accesibile la nivel BIOS.

Tabelul 3. Funcţii video (BIOS) uzuale

Funcţie	**Parametri de intrare**	**Parametri de ieşire**	**Descrierea funcţiei** Detalii
1	2	3	4
AH=00h	AL=mod video.		Selecţie mod video: mod=00h: alfanumeric, 40×25, alb/negru; mod=01h: alfanumeric, 40×25, color; mod=02h: alfanumeric, 80×25, alb/negru; mod=03h: alfanumeric, 80×25, color; mod=04h: grafic, 320×200, color; mod=05h: grafic, 320×200, alb/negru; mod=06h: grafic, 640×200, alb/negru; … mod=13h: grafic, 640×480, color.

1	2	3	4
AH=01h	CH=linie superioară; CL=linie inferioară.		Setare formă / dimensiune cursor. Cursorul are forma unui dreptunghi, cuprins între linia inferioară şi cea superioară. Numerotarea liniilor începe de sus. Valorile liniei sunt în principiu cuprinse în domeniul 0…7, fiind însă dependente de tipul adaptorului video.
AH=02h	BH=pagină; DH=linie; DL=coloană.		Stabilire poziţie cursor. Este actualizată poziţia curentă a cursorului în pagina video selectată.
AH=03h	BH=pagină.	CH:CL=formă cursor DH:DL= coordonate	Citire coordonate şi formă cursor. Se returnează coordonatele curente şi forma cursorului. CH:CL au semnificaţia de la funcţia 01h. DH:DL au semnificaţia de la funcţia 02h.
AH=05h	AL=pagină.		Selecţie pagină video activă. Pagina activă este cea curent afişată. Funcţia are semnificaţie numai pentru modurile alfanumerice.
AH=06h	AL=număr linii de deplasat; BH=atribut de culoare pentru liniile introduse; CH:CL=limite de fereastră, stânga-sus; DH:DL=limite de fereastră, dreapta-jos.		Defilare în sus a conţinutului ferestrei. Fereastra text definită de coordonatele CH:CL şi DH:DL (linie:coloană) este deplasată în sus cu numărul de linii specificat de AL. Dacă AL=0, această funcţie şterge fereastra specificată.
AH=07h	AL=număr linii de deplasat; BH=atribut de culoare pentru liniile introduse; CH:CL=limite de fereastră, stânga-sus; DH:DL=limite de fereastră, dreapta-jos.		Defilare în jos a conţinutului ferestrei. Fereastra text definită de coordonatele CH:CL şi DH:DL (linie:coloană) este deplasată în jos cu numărul de linii specificat de AL. Dacă AL=0, această funcţie şterge fereastra specificată.
AH=08h	BH=pagină.	AL=cod ASCII al caracterului citit; AH=atribut de culoare.	Citire caracter şi atribute asociate. Codul ASCII al caracterului de la poziţia curentă a cursorului va fi regăsit, după apel, în AL, iar atributele de culoare în AH. Funcţia are semnificaţie numai pentru modurile alfanumerice.

1	2	3	4
AH=09h	AL=cod ASCII al caracterului de afişat; BH=pagină; BL=atribut de culoare; CX=număr repetări ale caracterului.		Scriere caracter, cu atribute specifice. Caracterul cu codul ASCII din AL este afişat în poziţia curentă a cursorului, pe pagina selectată de BH. Atributele de culoare sunt specificate în BL. Funcţia are semnificaţie numai pentru modurile alfanumerice.
AH=0Ah	AL=cod ASCII al caracterului de afişat; BH=pagină; CX=număr de repetări ale caracterului.		Scriere caracter, cu păstrare atribute. Caracterul cu codul ASCII din AL este afişat în poziţia curentă a cursorului, pe pagina selectată de BH. Atributele de culoare sunt cele existente în poziţiile de scriere. Funcţia are semnificaţie numai pentru modurile alfanumerice.
AH=0Eh	AL=cod ASCII al caracterului de afişat; BH=pagină; BL=atribut de culoare pt. modurile grafice.		Scriere caracter, cu actualizare poziţie. Caracterul cu codul ASCII din AL este afişat în poziţia curentă a cursorului, pe pagina selectată de BH. Atributele de culoare sunt specificate în BL. Funcţia are semnificaţie atât pentru modurile alfanumerice cât şi pentru cele grafice.
AH=0Fh		AL=mod video; AH=număr coloane; BH=pagina.	Determinare mod de lucru curent. După apel, în registrul AL este returnat codul modului video (cu semnificaţia de la funcţia 00h). AH conţine numărul de coloane pentru modul alfanumeric. Dacă regimul este grafic, BH=0.

Utilizarea tastaturii (INT 16h)

Pentru controlul tastaturii se folosesc două rutine de tratare a întreruperilor: una pe nivelul 09h (care tratează semnalele venite de la tastatură) şi una pe nivelul 16h, aflată la dispoziţia utilizatorului (pentru obţinerea efectivă a unui caracter de la tastatură). Practic, interacţiunea prin program se poate face doar prin rutinele de pe nivelul 16h, conform metodologiei prezentate în continuare.

Funcţia 00h a întreruperii BIOS 16h (citire caracter)

Registre/parametri de intrare:

AH=00h (numărul funcţiei).

Registre/parametri de ieşire:

AL=cod_ASCII (codul ASCII al caracterului citit);

AH=cod_scan (codul de scanare al tastei apăsate).

Descriere:

Dacă INT 16h este apelată cu valoarea 00h depusă în AH, rutina din BIOS va reda controlul programului apelant doar în momentul în care o tastă este disponibilă în buffer-ul

tastaturii. La întoarcere, registrul al conţine codul ASCII pentru tasta citită din buffer, în timp ce ah memorează codul de scanare.

Nu toate tastele au asociat un cod ASCII corespondent (taste ca Home, PgUp, PgDn, End, Alt fiind exemple tipice). În această situaţie, INT 16h întoarce 00h în AL şi codul de scanare în registrul AH. Pe cale de consecinţă, atunci când valoarea corespunzătoare codului ASCII este 00h, programele de aplicaţie vor determina tasta apăsată prin verificarea (suplimentară) a conţinutului lui AH.

Funcţia 01h a întreruperii BIOS 16h (test disponibilitate caracter)
Registre/parametri de intrare:
AH=01h (numărul funcţiei).
Registre/parametri de ieşire:
ZF=0, atunci când există un caracter disponibil în buffer;
ZF=1, dacă nu există caractere în buffer;
AL=cod_ASCII (codul ASCII al caracterului disponibil);
AH=cod_scan (codul de scanare, dacă e disponibil).
Descriere:
Această funcţie permite verificarea existenţei unui caracter în buffer, semnalarea fiind făcută prin poziţionarea corespunzătoare a indicatorului ZF. În orice situaţie (existenţă/inexistenţă) controlul este returnat după apelare. Funcţia *nu şterge caracterul din buffer*, pentru aceasta fiind necesar un apel al funcţiei precedente (00h).

Funcţia 02h a întreruperii BIOS 16h (citire stare taste tip *shift*)
Registre/parametri de intrare:
AH=02h (numărul funcţiei).
Registre/parametri de ieşire:
AX=cod_bit (codificarea la nivel de bit a stării tastelor speciale).
Descriere:
Funcţia 02h întoarce starea tastelor de tip *shift* în registrul AX. Valorile returnate, la nivel de bit, pot fi următoarele:

- Bit 7 = 1: activare mod Insert (prin apăsarea tastei Ins);
- Bit 7 = 0: mod Insert inactiv;
- Bit 6 = 1: activare mod CapsLock (prin tasta CapsLock);
- Bit 6 = 0: mod CapsLock inactiv;
- Bit 5 = 1: activare mod NumLock (prin tasta NumLock);
- Bit 5 = 0: mod NumLock inactiv;
- Bit 4 = 1: activare mod ScrollLock (prin tasta ScrollLock);
- Bit 4 = 0: mod ScrollLock inactiv;
- Bit 3 = 1: tastă Alt apăsată;
- Bit 3 = 0: tastă Alt neapăsată;
- Bit 2 = 1: tastă Ctrl apăsată;
- Bit 2 = 0: tastă Ctrl neapăsată;
- Bit 1 = 1: tastă Shift stânga apăsată;
- Bit 1 = 0: tastă Shift stânga neapăsată;
- Bit 0 = 1: tastă Shift dreapta apăsată;
- Bit 0 = 0: tastă Shift dreapta neapăsată.

4. Desfăşurarea lucrării

În scop ilustrativ sunt prezentate câteva exemple de programe care utilizează funcţiile BIOS anterior detaliate. Studenţii vor aduce aceste programe la forma executabilă, pentru a le rula şi a observa efectul lor.

A. Program de utilizare a terminalului video:

```
.MODEL small
.STACK 100h
code segment
     assume cs:code, ds:code
; Procedura afisare text utilizand si atribute de culoare
; Conținut registre la apel:
;          (DH,DL)  = numar linie coloana primul caracter
;                     din text
;          (BH)     = numar pagina in care se face afisarea
;          (BL)     = atribut culoare utilizat pentru toate
;                     caracterele
;          (DS)     = adresa segmentului care conține textul
;          (SI)     = adresa relativa in segment a textului
;
; Textul se termina cu un octet cu valoare 0.
afisare proc
   mov   cx,1   ;repetare caracter
iar:
   mov   ah,2
   int   10h    ;pozitionare cursor
   lodsb
   cmp   al,0   ;comparare sfarsit text
   jz    gata
   cmp   al,0dh ;comparare ENTER
   jz    cr
   mov   ah,9
   int   10h    ;afisare caracter
   inc   dl     ;repozitionare caracter coordonate x
   cmp   dl,80
   jnz   iar
cr:
   inc   dh     ;pozitionare randul următor
   mov   dl,0
   jmp   iar

gata:
   ret
afisare endp
start:
   mov   ax,cs
   mov   ds,ax
   mov   cx,0
   mov   dx,256*24+79
   mov   ah,6
```

```
    mov   bh,36h
    int   10h          ;se sterge ecranul cu o culoare
    mov   dx,10*256+12 ;coordonate cursor (linia=10,
                       ;coloana=12)
    mov   bh,0         ;pagina 0
    mov   bl,00011101b ;atribut culoare
    mov   si,offset mesaj
    call  afisare
    mov   ax,4c00h
    int   21h
  mesaj  db  'Merge !!!',0dh,'Ca daca n-ar fi asa...',0
  code ends
      end start
```

B. Program de utilizare a tastaturii:

```
.MODEL SMALL
.STACK 100h
.CODE
mov ax,@data
mov ds,ax
start:
;Citeste o secventa de tastare pana cand se apasa Enter
ReadLoop:   mov   ah, 0       ;citeste codul tastei
       int   16h
       cmp   al, 0            ;functie speciala
       jz    ReadLoop         ;daca da, nu afisa aceasta
                              ;tasta
       Mov    ah, 0Eh
       int    10h        ;se afiseaza
       cmp    al, 0dh    ;inceput de rand (ENTER)
       jne   ReadLoop
mov ah,4ch
int 21h
END start
```

5. Modul de lucru

► Se editează programele exemplu (în orice mediu de editare) şi se salvează cu extensia .ASM.

► Se asamblează şi link-editează programele, obţinându-se forma .EXE.

► Se execută primul executabil obţinut şi se observă în urma execuţiei acestui program modul de intracţiune cu adaptorul video. Se cere să se creeze un program asemănător care să şteargă întreg ecranul, şi să modifice atributul de culoare al caracterelor la valoarea iniţială.

► Se rulează cel de-al doilea executabil şi se introduce un şir de caractere de la tastatură, finalizat prin apăsarea tastei ENTER. Se cere să se creeze un program asemănător care să citească de la tastatură, în mod succesiv, caractere individuale până se apasă tasta ESC.

LUCRAREA 7

ÎNTRERUPERI DOS

1. Obiectivele lucrării

Lucrarea urmăreşte prezentarea principială a modului de lucru cu întreruperile pentru microprocesoarele 80x86, punându-se accent pe evidenţierea unor tehnici de implementare a programelor în limbaj de asamblare ce folosesc apeluri de sistem DOS. Se au în vedere funcţiile sistem pentru gestiunea operaţiilor de I/E, memoriei şi proceselor.

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

Acţiunea provocată de îndeplinirea unei condiţii (internă sau externă în raport cu programul aflat în execuţie) prin care se realizează automat transferul controlului unei rutine speciale de tratare se numeşte *întrerupere*, iar rutina respectivă se numeşte *rutina de întrerupere*.

Un caz special este reprezentat de rutinele de tip utilizator ("de sistem") ce sunt destinate asistenţei utilizatorului în dialogul cu maşina de calcul şi exploatarea resurselor acesteia. Aşa cum s-a arătat în precedenta lucrare, astfel de utilitare de bază se pot regăsi la nivelul BIOS, însă cele substanţiale sunt conţinute în sistemul de operare.

3.1. Principii de utilizare a rutinelor de întrerupere din sistemul de operare DOS

Un sistem de operare permite utilizatorului controlul asupra sistemului de calcul; el poate fi definit ca o colecţie de proceduri şi programe. Aceste proceduri pot fi apelate din programele utilizatorului şi permit acestuia accesul la resursele calculatorului, memorie şi periferice, precum şi la o serie de informaţii care descriu contextul curent în care funcţionează calculatorul.

Există două tipuri de servicii DOS: întreruperi şi funcţii. Funcţiile DOS se apelează cu întreruperea 21h (INT 21h) şi numărul funcţiei în registrul AH, în timp ce celelalte întreruperi DOS se apelează prin întreruperi software de tip INT n.

Rutinele pe care sistemul de operare DOS le utilizează pentru gestiunea resurselor calculatorului pot fi apelate din programele scrise în limbaj de asamblare. Aceste rutine se mai numesc *apeluri sistem* (system calls). Utilizarea apelurilor sistem de către programele de aplicaţie permite obţinerea de programe independente de maşina fizică, garantându-se compatibilitatea acestora cu versiunile superioare ale sistemului de operare DOS.

Instrucţiunile de întrerupere cu tipurile (nivelurile) 20h,...,3Fh sunt rezervate apelurilor sistem. Majoritatea apelurilor sistem sunt grupate la întreruperea cu tipul 21h.

Aceste apeluri sunt cunoscute sub numele de *funcţii sistem*. Pentru executarea unui apel sistem se procedează astfel:

► se transferă parametrii apelului în registrele procesorului. Parametrii împreună cu registrele corespunzătoare sunt specifici fiecărui apel în parte;
► se execută instrucţiunea INT cu tipul corespunzător apelului respectiv.

Pentru executarea unei funcţii sistem se procedează astfel:
► se transferă parametrii funcţiei în registre;
► se transferă numărul funcţiei în registrul AH;
► se transferă un cod suplimentar, dacă acest lucru este cerut de funcţia respectivă in registrul AL;
► se execută instrucţiunea INT 21h.

Majoritatea funcţiilor sistem setează flagul CF dacă a apărut o eroare, codul de eroare fiind returnat în registrul AX.

3.2 Rutine uzuale DOS apelabile prin sistemul de întreruperi

Principalele funcţii sistem pot fi grupate în următoarele categorii:

Funcţii pentru operaţiile de I/E standard specifice următoarelor dispozitive de I/E: consola, imprimanta, interfaţa serială. Dacă un program foloseşte pentru implementarea operaţiilor de I/E aceste funcţii, intrarea respectiv ieşirea sa pot fi redirectate. Dintre aceste funcţii, de menţionat sunt cele prezentate în continuare.

Funcţia 01h. Această funcţie citeşte un caracter de la intrarea standard, pe care îl trimite în ecou la ieşirea standard. În cazul în care caracterul citit este CTRL-C, se execută INT 23h. La apel AH=01h, şi după execuţie AL=codul ASCII al caracterului citit.
Exemplu:

```
mov ah,1
int 21h
```

Funcţia 02h. Această funcţie scrie un caracter la ieşirea standard. Dacă se tastează CTRL-C, se execută INT 23h. La apel AH=02_{16}, DL=codul ASCII al caracterului care trebuie scris.
Exemplu:

```
mov dl,cod_ASCII_caracter
mov ah,2
int 21h
```

Funcţia 07h. Această funcţie citeşte un caracter de la intrarea standard. Caracterul citit nu este trimis în ecou la ieşirea standard şi nu se efectuează nici un test pentru CTRL-C. La apel AH=07h şi după execuţie AL=codul ASCII al caracterului citit.
Exemplu:

```
mov ah,7
int 21h
```

Funcţia 09h. Această funcţie realizează afişarea unui şir de caractere la ieşirea standard, şirul de caractere fiind terminat cu '$'. Caracterul '$' nu se afişează. La apel AH=09h, DS:DX=adresa de început a şirului de caractere respectiv.

Exemplu:

```
lea dx,sir
mov ah,9
int 21h
```

Funcţia 0Ah. Această funcţie realizează citirea unui şir de caractere de la intrarea standard. La apel AH=0Ah, DS:DX=adresa unei zone tampon cu următoarea structură:

► primul octet al zonei va conţine la apel numărul maxim (fie acesta *n*) de caractere care vor fi citite, inclusiv un caracter CR;

► al doilea octet al zonei va conţine după execuţia funcţiei numărul de caractere efectiv citite, caracterul CR nefiind contorizat;

► restul zonei va trebui să conţină cel puţin atâţia octeţi câţi au fost specificaţi în primul octet. După execuţia funcţiei aici va fi depus şirul citit, primul caracter din şir fiind plasat la adresa cea mai mică.

Caracterele citite sunt plasate în zona tampon specificată, începând cu al treilea octet al zonei. Citirea se termină când se tastează CR. Dacă se încearcă introducerea mai multor caractere decât numărul specificat, adică n-1, caracterele în plus se ignoră, fiind trimis la ieşirea standard caracterul BEL (07h). Ultimul caracter din zona tampon este întotdeauna CR. Daca se tastează CTRL-C, se executa INT 23h.

Exemplu:

```
lea dx,sir
mov sir,nr_car
mov ah,0Ah
int 21h
```

Funcţii pentru gestiunea memoriei

Sistemul de operare DOS ţine evidenţa blocurilor de memorie alocate prin scrierea la începutul fiecăruia a unei înregistrări numită *bloc de control al memoriei* (Memory Control Block, MCB) ce conţine informaţii referitoare la dimensiunea blocului în număr de paragrafe (1 paragraf=16 octeţi) şi un pointer la următorul bloc, dacă există. Dintre funcţiile pentru gestiunea memoriei, cele uzuale sunt prezentate în continuare.

Funcţia 48h. Prin această funcţie, procesul aflat în execuţie poate cere alocarea unei cantităţi de memorie, specificată în număr de paragrafe. La apel AH=48h, BX=numărul de paragrafe cerute. Dacă la retur CF=1, înseamnă că a apărut o eroare, codul de eroare fiind specificat în registrul AX: dacă AX=7 înseamnă că s-a alterat un MCB, iar AX=8 înseamnă că nu există suficientă memorie pentru alocare. În registrul BX se obţine numărul de paragrafe disponibile. Dacă la retur CF=0, înseamnă că alocarea s-a efectuat cu succes, în registrul AX întorcându-se adresa de segment a zonei de memorie alocată.

Exemplu:

```
mov bx,numar_octeti
mov cl,4
shr bx,cl
inc bx
int 21h
```

Funcţia 49h. Această funcţie eliberează un bloc de memorie alocat anterior cu funcţia 48h. La apel AH=49h, ES=adresa de segment a blocului respectiv. Dacă la retur CF=1 înseamnă că a apărut o eroare, codul de eroare fiind specificat în registrul AX: AX=7 înseamnă alterarea MCB-ului, iar AX=9 înseamnă că se doreşte eliberarea unui bloc ce nu a fost alocat corect (cu funcţia 48h). Dacă la retur CF=0 înseamnă că eliberarea s-a făcut cu succes.

Exemplu:

```
mov ax,adresa_segment
mov es,ax
mov ah,9
int 21h
```

Funcţia 4Ah. Această funcţie modifică dimensiunea unui bloc de memorie alocat anterior. La apel AH=$4A_{16}$, BX=noua dimensiune a blocului în număr de paragrafe. Dacă la retur CF=1, înseamnă ca a apărut o eroare, codul de eroare fiind specificat în registrul AX: AX=7 înseamnă alterarea MCB-ului, AX=8 înseamnă memorie insuficientă (în situaţia în care s-a dorit mărirea dimensiunii blocului), iar AX=9 înseamnă că se doreşte modificarea dimensiunii unui bloc de memorie ce nu a fost alocat corect. Dacă AX=8, registrul BX va conţine numărul de paragrafe disponibile. Dacă la retur CF=0 înseamnă că modificarea dimensiunii blocului s-a făcut cu succes.

Exemplu:

```
lea bx,ultimul_octet
mov cl,4
shr bx,cl
add bx,17
mov ah,4Ah
int 21h
mov ax,bx
shl ax,cl
```

Funcţii pentru gestiunea proceselor

Funcţia $4B_{16}$. Această funcţie realizează încărcarea şi execuţia unui program, respectiv încărcarea unui *overlay*, fără executarea acestuia, în funcţie de un cod suplimentar, ce poate avea valoarea 0, respectiv 3.

Un *overlay* este o porţiune de program ce se păstrează pe disc, fiind încărcată în memoria principală pentru execuţie la cererea utilizatorului, datorită spaţiului limitat al acesteia. În cele ce urmează se prezintă doar funcţia de încărcare şi execuţie a unui program, cunoscută şi sub numele de funcţia *load and execute*.

La apel AH=4Bh, AL=0, DS:DX=adresa unui şir de caractere terminat cu caracterul NUL (codul ASCII 0) ce reprezintă calea către un fişier ce conţine programul executabil, iar (es):(bx)=adresa unui bloc de parametri (EXEC Parameter Block, EPB). Când un program utilizează această funcţie pentru încărcarea şi executarea altui program, sistemul de operare DOS alocă memoria necesară şi creează un bloc de date numit *prefixul de segment al programului* (Program Segment Prefix, PSP) la offsetul 0 în cadrul blocului de memorie alocat programului, încarcă noul program şi îi pasează controlul. Adresa de segment a PSP se depune în registrele segment DS şi ES. Noul program are responsabilitatea ca la terminarea sa să execute o funcţie sistem pentru retransmiterea controlului programului apelant. Programul apelant se numeşte

program părinte şi programul încărcat şi executat sub controlul programului părinte se numeşte *program copil*.

Linia de comandă se specifică sub forma unui şir de octeţi cu următoarea strucutură:

► primul octet este lungimea liniei, caracterul CR de terminare nefiind contorizat;
► şirul de caractere ce compune linia de comanda propriu-zisă;
► ultimul octet din şir este caracterul CR.

Spre exemplu, pentru lansarea programului *command.com* cu linia de comandă

command /c dir /w

ce are ca efect executarea unei copii a *command.com* şi sub această copie a unei comenzi *dir /w*, şirul corespunzător liniei de comanda se va defini astfel:

```
cmd_line  DB  9,'/c dir /w',0dh
```

Pentru ca programul copil să poată fi încărcat trebuie să existe suficientă memorie principală disponibilă. Sistemul de operare DOS alocă toată memoria principală disponibilă unui program în momentul încărcării sale. Când un program părinte realizează încărcarea şi executarea unui program copil, trebuie să elibereze o porţiune de memorie utilizând funcţia 4Ah, înaintea apelului pentru încărcarea şi executarea programului copil.

În urma executării apelului toate fişierele deschise devin disponibile noului program încărcat, programul părinte având astfel controlul definirii intrării şi ieşirii standard, auxiliare şi al dispozitivelor de imprimare.

Programul copil primeşte de asemenea un mediu (environment), ce reprezintă o secvenţă de şiruri de caractere (şirurile de caractere din cadrul mediului DOS curent se pot afişa la terminal cu comanda DOS *set*), terminate fiecare cu caracterul NUL, de forma parametru=valoare (spre exemplu VERIFY=ON). Mediul trebuie să înceapă la o adresă de paragraf, să nu depăşească 32 Kocteti şi să fie terminat cu caracterul NUL. După ultimul caracter NUL se află o mulţime de argumente pasate programului copil, care constau dintr-un contor pe cuvânt (în mod normal egal cu 1) urmat de o secvenţă de şiruri de caractere terminate fiecare cu caracterul NUL, numărul de şiruri fiind indicat de contor (în mod normal un sir). Dacă apelul găseşte fişierul cu programul executabil în directorul curent, şirul de caractere conţine unitatea şi calea către programul executabil, aşa cum au fost pasate funcţiei $4B_{16}$. Dacă apelul găseşte fişierul în calea respectiva, apelul concatenează numele fişierului cu calea respectivă. Un program poate utiliza această zonă pentru a determina calea de unde a fost încărcat. Mediul este alocat într-un bloc separat de memorie. Se recomandă ca un program care rămâne rezident (cu funcţia 31h) să îşi elibereze blocul de memorie ce conţine mediul, cu funcţia 49h.

Secvenţa de program necesară pentru încărcarea şi executarea unui program trebuie să conţină în majoritatea situaţiilor următorii paşi:

► Se execută un apel al funcţiei sistem 4Ah cu ES=adresa de segment a PSP şi BX=cantitatea minimă de memorie necesară programului părinte în paragrafe;
► Se pregăteşte un şir de caractere terminat cu caracterul NUL ce conţine numele fişierului cu programul copil şi se depune adresa de început a acestui şir în perechea de registre DS:DX;
► Se pregăteşte un EPB şi se depune adresa sa de început în perechea de registre ES:BX;
► Se salvează valorile curente ale registrelor de segment în variabile relative la CS, acesta fiind singurul punct de referinţă în urma revenirii din programul copil;
► Se execută apelul funcţiei 4Bh;

► Se refac registrele de segment la valorile inițiale;
► Se verifică flagul CF pentru a se testa apariția unei eventuale erori;
► Opțional se poate testa codul de revenire din programul copil, cu ajutorul funcției sistem 4Dh, prezentată în continuare.

Funcția 4Dh: Prin intermediul acestei funcții, un program părinte poate prelua codul de retur întors de un program copil prin funcțiile 31h sau 4Ch. La apel AH=4Dh, iar în urma executării apelului AL=codul de retur și AH=un cod ce indică metoda de ieșire din programul copil, putând avea următoarele valori:

Cod	**Semnificație**
0	Terminare normala
1	Terminare prin CTRL-BREAK (int 23h)
2	Terminare datorita unei erori critice (int 24h)
3	Terminare prin functia 31_{16} (pastrare proces în memorie)

Această funcție trebuie apelată doar o singură dată pentru fiecare proces terminat în parte.

Exemplu:

```
mov ah,4Dh
int 21h
```

Funcția 31h. Apelul acestei funcții are ca efect terminarea programului aflat în execuție, datele și codul programului terminat rămânând rezidente în memorie. La apel AH=31_{16}, AL=codul de retur întors către programul părinte și DX=numărul de paragrafe de memorie necesare programului ce rămâne rezident.

Exemplu:

```
mov al,ind_eroare
lea dx,ultimul_octet
mov cl,4 ; conversie la număr de paragrafe
shr dx,cl
inc dx
mov ah,31h
int 21h
```

Funcția 62h. Această funcție se utilizează pentru determinarea adresei prefixului (Program Segment Prefix) corespunzător programului aflat în execuție. La apel AH=62h adresa fiind întoarsă în registrul BX.

Exemplu:

```
mov ah,62h
int 21h
```

Pentru sistematizare, tabelul 1 prezintă și o serie de alte funcții de sistem ce pot fi apelate în programe scrise în limbaj de asamblare.

Tabelul 1. Alte funcţii de sistem DOS

Cod funcţie (AH)	Descrierea funcţiei	Parametri de intrare si iesire (in AH se pune codul funcţiei)
00h	Terminarea programului (versiunea mai veche)	Intrare : CS- adresa de segment a PSP-ului programului de terminat
03h	Intrare auxiliară (implicit COM1)	Ieşire : AL- caracterul recepţionat
04h	Ieşire auxiliară (implicit COM1)	Intrare : DL- caracterul de transmis
05h	Tipărirea unui caracter	Intrare : DL- caracterul de transmis
06h	Ieşire directă de la consolă	Intrare: DL = caracter (except FFh) Ieşire: AL = caracter de ieşire
06h	Intrare directă de la consola	Intrare: AH = 06h, DL = FFh Ieşire: ZF *set* dacă nu este caracter disponibil şi AL = 00h ZF *clear* dacă este caracter disponibil şi AL = caracter citit
08h	Citirea fară ecou de la tastatura	Ieşire : AL- caracterul citit
0fh	Deschiderea unui fişier	Intrare : DS :DX – pointer la FCB-ul fişierului nedeschis Ieşire : AL=0 – s-a gasit fişierul =FF – eroare
10h	Închiderea unui fişier	idem
14h	Citirea secvenţială a unui fişier folosind FCB (File Control Block)	Intrare : DS :DX – pointer la FCB-ul fişierului deschis Ieşire : AL=0 – citire corectă #0 – eroare
2Ah	Get Date: Returnează data curentă din MS-DOS	Ieşire : AL – Ziua (0=Duminica, 1=Luni, etc.). CX – Anul DH – Luna (1=Ianuarie, 2=Feb, etc.) DL – Ziua din luna (1-31)
2Bh	Set Date: Setează data din MS-DOS	Intrare: CX – Anul (1980 - 2099) DH – Luna (1-12) DL – Ziua (1-31)
2Ch	Get Time: Citeşte timpul curent din MS-DOS	Ieşire: CH – Ora (24h) CL – Minute DH – Secunde DL – Sutimi
2Dh	Set Time: Setează timpul din MS-DOS	Intrare: CH – Ora CL – Minute DH – Secunde DL – Sutimi

Cod funcţie (AH)	Descrierea funcţiei	Parametri de intrare si iesire (in AH se pune codul funcţiei)
3Ch	Crearea unui fişier	Intrare : DS :DX- pointer la numele fişierului (şir ASCIIZ) CX- atributul fişierului (00- normal ;01- numai citire ;02- ascuns ;03-sistem) Ieşire : CF=0- Operaţie reuşita AX- identificatorul logic al fişierului CF=1 Eroare
3Dh	Deschiderea unui fişier folosind identificatorul logic	Intrare : DS :DX- pointer la numele fişierului (şir ASCIIZ) AL-codul de acces Ieşire : CF=0- Operaţie reuşită AX- identificatorul logic al fişierului CF=1 Eroare
3Eh	Închiderea unui fişier folosind identificatorul logic	Intrare : BX – identificatorul logic al fişierului Ieşire : CF=0- Operaţie reuşită CF=1 Eroare
3Fh	Citirea unui fişier folosind identificatorul logic	Intrare : BX – identificatorul logic al fişierului CX- nr. de octeţi citiţi DS :DX- pointer la zona de citire Iesire : CF=0 – Operaţie reuşită AX – numărul de caractere citite CF=1 Eroare
4Ch	Terminarea unui proces	Intrare : AL- codul de revenire

4. Desfăşurarea lucrării

În continuare sunt propuse 2 programe (P1, P2) al căror scop este acela de a ilustra modul de apelare a resurselor DOS prin intermediul întreruperilor.

P1 utilizează funcţia 4Bh pentru a rula o a doua copie a programului *command.com*. Revenirea se face executând comanda DOS *exit*.

```
.model small
.stack 10h
DATA      SEGMENT
     epb_struc STRUC
     seg_env   DW     0
     cmd_lin   DW     2 DUP(?)
     fcb_1     DW     2 DUP(?)
     fcb_2     DW     2 DUP(?)
     epb_struc ENDS
     mesaji    DB 'Se lanseaza un nou COMMAND.COM','$'
     mesajo    DB 'S-a revenit la vechiul COMMAND.COM!','$'
```

```
        pgm_file  DB 'c:\command.com',0
        pgm_cmd   DB 0
        epb       epb_struc <>
        DATA      ENDS
    .model small
    .stack 10h
        COD       SEGMENT
                  ASSUME cs:COD,ds:DATA
        old_ss    DW     ?
        old_sp    DW     ?
        old_ds    DW     ?
        start:    mov    ax,DATA
                  mov    ds,ax
    ; Calculează numărul de paragrafe alocate programului in ax
                  mov    ax,cs
                  mov    bx,es
    ; es contine adresa de segment a PSP
                  sub    ax,bx
                  mov    cl,4
                  shl    ax,cl
                  add    ax,OFFSET ultima_instr
                  xor    dx,dx
                  mov    cx,10h
                  div    cx
                  inc    ax
    ; Se reduce dimensiunea blocului de memorie alocat
                  mov    bx,ax
                  mov    ah,4ah
                  int    21h
    ; Se afişează primul mesaj
                  lea    dx,mesaji
                  mov    ah,09h
                  int    21h
    ; Se aşteaptă o tasta
                  mov    ah,07h
                  int    21h
    ; Se încarcă o noua copie a COMMAND.COM
                  mov    dx,OFFSET pgm_file
                  mov    bx,OFFSET epb
                  mov    [bx].cmd_lin[0],OFFSET pgm_cmd
                  mov    [bx].cmd_lin[2],SEG pgm_cmd
                  mov    [bx].fcb_1[0],5ch
                  mov    [bx].fcb_1[2],es
                  mov    [bx].fcb_2[0],6ch
                  mov    [bx].fcb_2[2],es
                  mov    ax,4b00h
                  push   ds
                  pop    es
                  mov    old_sp,sp
                  mov    old_ss,ss
                  mov    old_ds,ds
                  int    21h
    ; Se afişează al doilea mesaj
```

```
                mov     sp,old_sp
                mov     ss,old_ss
                mov     ds,old_ds
                lea     dx,mesajo
                mov     ah,09h
                int     21h
; Terminare program
                mov     ax,4c00h
                int     21h
ultima_instr:
     COD        ENDS
                END     start
```

Programul **P2** citeşte o tastă, determină dacă este o tastă obişnuită sau funcţională şi apoi afişează codul tastei. Citirea se face cu funcţia 07h. Dacă la primul apel se întoarce codul 0 în registrul AL, înseamnă că tasta apăsată este o tastă funcţională, codul extins al tastei fiind determinat cu un al doilea apel al funcţiei.

```
     DATA       SEGMENT
     cerere     DB 'Apasati o tasta',0ah,0dh,'$'
     sir        DB 3 DUP (?),0ah,0dh,'$'
     mesaj1     DB 'Codul tastei este:',0ah,0dh,'$'
     mesaj2     DB 'Codul extins al tastei
                    este:',0ah,0dh,'$'
     DATA       ENDS
     STIVA      SEGMENT STACK 'STACK'
                DW      30 DUP(?)
     virf       LABEL   WORD
     STIVA      ENDS
     COD        SEGMENT
                ASSUME  cs:COD,ds:DATA,ss:STIVA
     start:     jmp     incep
     conv       PROC    NEAR
; Primeşte în registrul al un număr întreg şi în registrul
; bx adresa unei zone de 3 octeți.Întoarce în această zonă
; şirul de caractere corespunzător scrierii zecimale a
; numărului
push    dx
xor     ah,ah
mov     dl,10
div     dl
add     ah,'0'
mov     [bx+2],ah
xor     ah,ah
div     dl
add     ah,'0'
mov     [bx+1],ah
add     al,'0'
mov     [bx],al
pop     dx
ret
     conv     ENDP
```

```
      incep:   mov     ax,DATA
               mov     ds,ax
               mov     ax,STIVA
               mov     ss,ax
               mov     sp,OFFSET virf
; Se afişează mesajul care cere apasarea unei taste
               lea     dx,cerere
               mov     ah,09h
               int     21h
; Se citeşte o tastă
               mov     ah,07h
               int     21h
; Se testează dacă este tastă funcțională
               cmp     al,0
               jz      cod_extins
; Se determină codul tastei
               lea     bx,sir
               call    conv
; Se afişează mesaj1
               lea     dx,mesaj1
               mov     ah,09h
               int     21h
               jmp     afis
; Se citeşte codul extins
  cod_extins: mov     ah,07h
               int     21h
; Se determină codul extins
               lea     bx,sir
               call    conv
; Se afişează mesaj2
               lea     dx,mesaj2
               mov     ah,09h
               int        21h
; Se afişează codul tastei
      afis:    lea     dx,sir
               mov     ah,09h
               int     21h
               mov     ax,4c00h
               int     21h
      COD    ENDS
               END       start
```

5. Modul de lucru

► Se editează programele exemplu (în orice mediu de editare) şi se salvează cu extensia .ASM.

► Se asamblează şi link-editează programele, obţinându-se forma *.EXE*.

► Se testează programele, inclusiv prin încărcarea în Turbo-Debugger.

► Se cere să se modifice P2 astfel încât la citirea tastei aceasta să fie afişată pe ecran.

► Să se scrie un program care să citească de la tastatura numele unui fişier şi să afişeze conţinutul fişierului pe ecran.

LUCRAREA 8

MODURI DE ADRESARE

1. Obiectivele lucrării

Această lucrare de laborator îşi propune ca obiective familiarizarea studenţilor cu registrele standard ale microprocesoarelor familiei 80x86 şi prezentarea modurilor de adresare specifice acesteia

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitatea centrală, monitor, tastatură, mouse.
► Manuale de prezentare a limbajului de asamblare şi precizările din prezentul îndrumar.

3. Breviar teoretic

3.1. Contextul microprocesorului. Registre şi fanioane

Registrele generale reprezintă practic memoria internă (de lucru) a oricărui microprocesor. În ceea ce priveşte familia Intel 80x86, 16 registre sunt semnificative din punct de vedere al aplicaţiilor dezvoltate de către utilizator, ele fiind evidenţiate în figura 1.

Registrele de transfer al datelor

AX, BX, CX, şi DX sunt direct adresabile pe 8/16 biţi, fiecare putând servi ca sursă/destinaţie a datelor (îndeplinind deci rol de acumulator). La nivel de octet, registrele pot fi referite prin specificarea jumătăţilor inferioare (biţii 0-7: AL, BL, CL, şi DL) sau superioare (biţii 8-15: AH, BH, CH, ŞI DH).

Începând de la microprocesoarele 80386, registrele cu utilizare generală sunt extinse la 32 de biţi, fiind referite ca eax, ebx, ecx, şi edx. Evident, acestea pot fi adresate pe 32 de biţi (de exemplu eax), pe 16 biţi (ax) sau pe 8 biţi (al, ah).

Deşi operarea registrelor cu utilizare generală se face în întregime conform dorinţei utilizatorului, proiectantul le-a asociat totuşi o suită de destinaţii specifice. Astfel:

► AX (Accumulator Register) este utilizat ca acumulator implicit, folosit inclusiv la operaţiile de înmulţire şi împărţire pe 16 biţi, respectiv pentru operaţii de intrare/ieşire pe 16 biţi;
► AL este utilizat în aceleaşi scopuri ca şi AX (cu precizarea că operarea se face pe 8 biţi), în plus el fiind destinat pentru efectuarea operaţiilor în BCD şi a conversiilor de cod;
► AH este folosit pentru înmulţire şi împărţire pe 8 biţi;
► BX (Base Register) se utilizează în conversii de cod, precum şi ca registru de bază la adresare;

► CX (Counter Register) are rolul unui contor de ciclu în cazul structurilor repetitive cu incrementare/decrementare, fiind deasemenea utilizat ca şi contor intern al operaţiilor cu şiruri;
► DX (Data Register) este utilizat ca registru de adresare indirectă pentru porturile de intrare/ieşire, precum şi la operaţiile de înmulţire/împărţire.

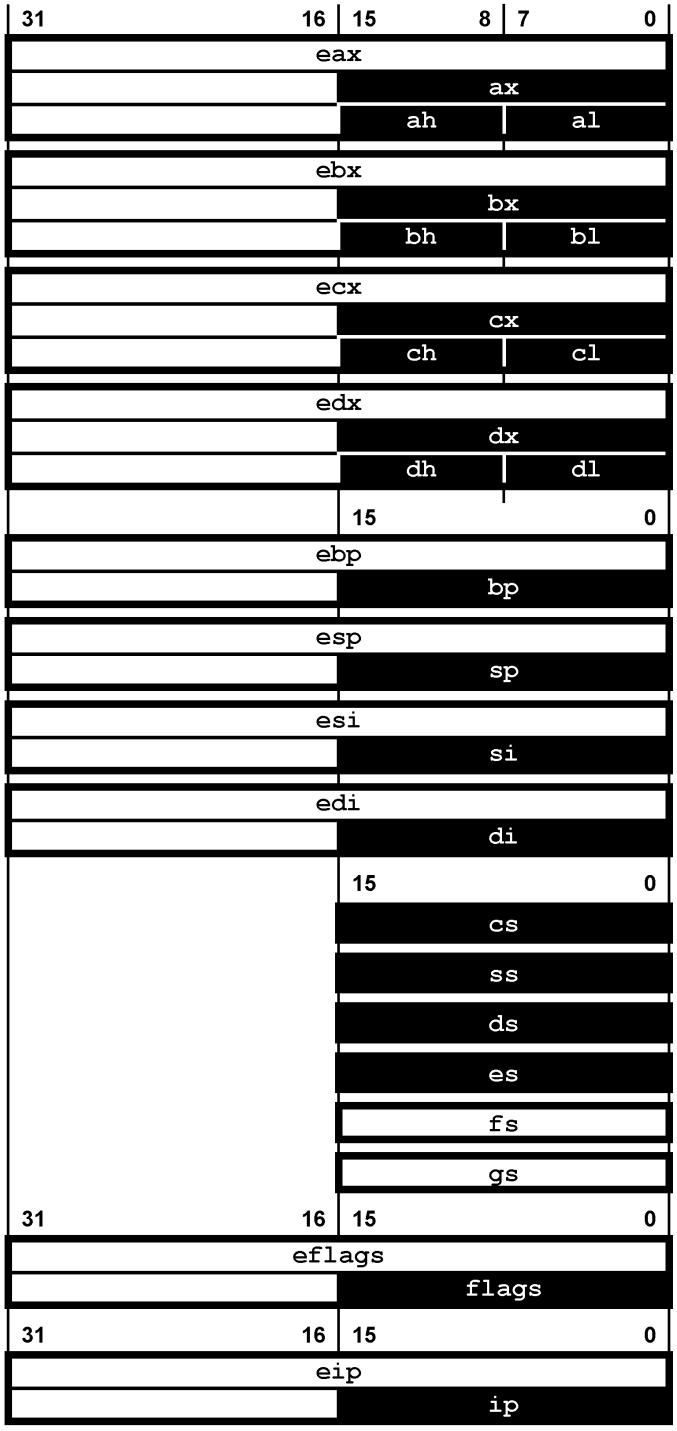


Figura 1. Registrele semnificative ale familiei de procesoare Intel® 80x86: cu negru este figurat setul de bază (al procesoarelor pe 16 biţi), iar cu alb este reprezentată extensia acestuia la procesoarele pe 32 de biţi.

Registrele de tip indicator (pentru lucrul cu stiva)

Pentru organizarea corespunzătoare a unei asemenea structuri, utilizatorului îi sunt puse la dispoziţie două elemente cu semnificaţia unor pointeri, pentru specificarea adreselor bazei stivei, respectiv a vârfului acesteia. În acest sens, baza este indicată de registrul BP (Base Pointer), în timp ce vârful este punctat de registrul SP (Stack Pointer), ambele fiind adresabile doar pe 16 biţi. Începând cu procesoarele 80386 se utilizează şi extensiile EBP/ESP, adresabile numai pe 32 de biţi (figura 1).

Registrele de tip index (pentru gestionarea şirurilor)

Registrele SI (Source Index) şi DI (Destination Index) sunt destinate operării cu două categorii de structuri tip tablou (sursă şi destinaţie). Principial, SI şi DI (şi extensiile lor la 32 de biţi, ESI şi EDI – din figura 1) conţin indecşii necesari pentru parcurgerea celor două structuri, valorile acestora participând la calculul adreselor pentru fiecare element individual din tablourile referite.

Registrele de segment (pentru gestionarea accesului la memorie)

Pentru memorarea adreselor de bază ale segmentelor de memorie asociate unui program aflat în execuţie, la nivelul procesoarelor Intel 80x86 se regăsesc o serie de registre cu destinaţie specială, numite sugestiv *registre de segment*. Indiferent de tipul microprocesorului (pe 16 sau 32 de biţi), un număr de patru registre sunt întotdeauna prezente ca elemente de arhitectură:

- CS (Code Segment) – registrul asociat bazei segmentului de cod;
- DS (Data Segment) – registrul asociat bazei segmentului de date;
- SS (Stack Segment) – registrul asociat bazei segmentului de stivă;
- ES (Extra Segment) – registrul asociat bazei segmentului suplimentar de date.

În plus, procesoarele pe 32 de biţi au posibilitatea declarării a două segmente suplimentare de date, bazele lor fiind referite de registrele FS şi GS. Ca observaţie, FS şi GS, ca de altfel întregul set al registrelor „standard" de segment (CS, DS, SS şi ES) sunt adresabile doar pe 16 biţi.

În calculul adresei unei locaţii este nevoie atât de specificarea bazei segmentului ce o conţine, cât şi de deplasamentul (offset-ul) acesteia faţă de bază. În ceea ce priveşte segmentul de cod, deplasamentul este conţinut de un registru special de 16 biţi, IP (Instruction Pointer), respectiv EIP (Extended Instruction Pointer) – la procesoarele pe 32 de biţi. În acest mod, perechea de registre CS:IP (sau CS:EIP) conţine toată informaţia necesară pentru calculul adresei următoarei instrucţiuni care se va executa.

Referitor la determinarea adreselor în segmentele de date, deplasamentele sunt specificate prin numele simbolice atribuite variabilelor, etichetelor etc. Astfel, calculul adreselor se face pe baza conţinutului registrelor DS, ES, FS şi GS, precum şi a respectivelor deplasamente.

Adresarea în segmentul de stivă nu face excepţie de la regula folosirii adresei de bază combinată cu deplasamentul locaţiei referite. Adresa unui element arbitrar din stivă (în particular a bazei sale logice) se calculează pe baza informaţiei conţinute de perechea de registre SS:BP, în timp ce adresa vârfului stivei este conţinută de perechea SS:SP. Bineînţeles, utilizarea extensiilor la 32 de biţi (EBP, ESP) este similară folosirii registrelor BP şi SP.

În mod firesc, programatorul trebuie să aibă permanent în vedere maniera concretă de calcul al adresei fizice, pentru ca valorile înscrise în registrele de segment şi pointeri să fie corecte şi consistente.

Registrul indicatorilor de condiţii

Indicatorii de condiţie, grupaţi în registrul FLAGS (EFLAGS, la procesoarele pe 32 de biţi) – aşa cum prezintă figura 2 – sunt utilizaţi pentru a memora informaţii despre starea generală a sistemului, proprietăţi ale rezultatului unor operaţii aritmetice sau logice, precum şi pentru exercitarea unor acţiuni de control al activităţii procesorului.

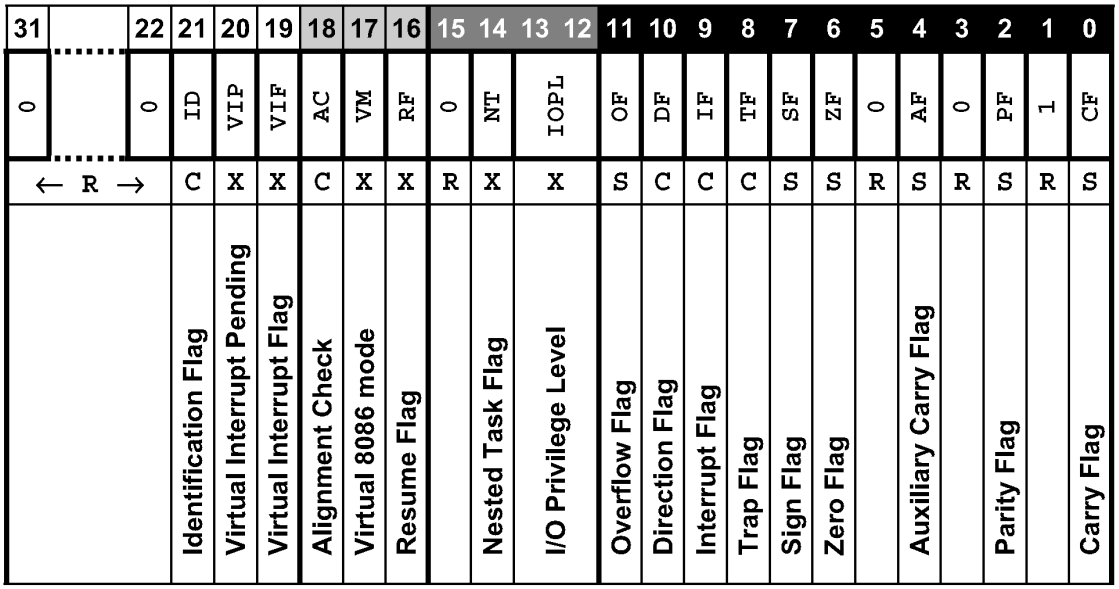


Figura 2. Registrul indicatorilor de condiţii:
X – fanion de sistem; S – fanion de stare;
C – fanion de control; R – poziţie rezervată.

În acest context, semnificaţiile biţilor din registrul (e)flags sunt următoarele:

Bitul 0: indicatorul de transport CF (Carry Flag). Are valoarea 1 dacă în execuţia unei instrucţiuni aritmetice care poziţionează acest indicator a apărut un transport sau s-a făcut un împrumut la rangul cel mai semnificativ. De asemenea, instrucţiunile de rotire a conţinutului unui registru pot acţiona asupra acestui indicator.

Bitul 2: indicatorul de paritate PF (Parity Flag). Ia valoarea 1 dacă din execuţia unei instrucţiuni care poziţionează acest indicator s-a obţinut un rezultat conţinând un număr par de biţi cu valoarea 1.

Bitul 4: indicatorul de transport auxiliar AF (Auxiliary Carry Flag). Are valoarea 1 dacă în execuţia unei instrucţiuni ce îl influenţează a apărut un transport de la rangul 3 spre rangul 4 (sau a fost executat un împrumut din rangul 4 spre rangul 3). Acest indicator este intens utilizat pentru implementarea operaţiilor aritmetice cu numere codificate în BCD.

Bitul 6: indicatorul de zero ZF (Zero Flag). Are valoarea 1 dacă, în urma execuţiei unei instrucţiuni ce are efect asupra sa, rezultatul obţinut este nul.

Bitul 7: indicatorul de semn SF (Sign Flag). Acest bit ia valoarea 1 dacă în urma unei operaţii aritmetice s-a obţinut un rezultat negativ (pentru care, conform convenţiei, bitul cel mai semnificativ este de asemenea 1). Practic, în acest caz, SF este o copie a bitului de semn asociat rezultatului.

Bitul 8: bitul de depanare TF (Trap Flag). Acest fanion de control este utilizat pentru execuţia programelor în regim pas cu pas, în scopul testării lor amănunţite. Dacă acest indicator este setat la valoarea 1 de către utilizator, după execuţia fiecărei instrucţiuni se va genera un semnal de întrerupere intern (pe nivelul 1).

Bitul 9: bitul de activare/dezactivare a sistemului de întreruperi IF (Interrupt Flag). Prin valorile date de către utilizator, acest fanion determină acceptarea (dacă IF=1) sau respingerea (dacă IF=1) semnalelor asociate întreruperilor externe. Bineînţeles, indicatorul nu are influenţă asupra întreruperilor nemascabile.

Bitul 10: bitul indicator al direcţiei de parcurgere a şirurilor DF (Direction Flag). Având semnificaţie doar în cazul instrucţiunilor de prelucrare a şirurilor, valoarea 0 (zero) a acestui flag indică parcurgerea şirurilor de la adrese mici spre adrese mai mari, în timp ce valoarea 1 determină inversarea sensului de tratare a acestora.

Bitul 11: indicatorul de depăşire la calcule efectuate în virgulă fixă OF (Overflow Flag). Acest bit ia valoarea 1 dacă din execuţia unei instrucţiuni aritmetice s-a obţinut un rezultat ce depăşeşte capacitatea de reprezentare a destinaţiei precizate implicit sau explicit (numărul de biţi ai acesteia).

Odată cu apariţia microprocesoarelor Intel 80286, capabile de a executa simultan mai multe secvenţe de cod (task-uri), registrul de flags a fost completat cu următorii biţi:

Biţii 12-13: indicatorul nivelului privilegiat de acces la operaţiile de intrare/ieşire IOPL (Input/Output Privilege Level). Valoarea celor doi biţi asociaţi afectează comportarea procesorului la detectarea apelurilor de tip intrare/ieşire.

Bitul 14: indicatorul de task imbricat NT (Nested Task). Valoarea 0 (zero) a acestui bit indică faptul că task-ul curent este independent, revenirea din acesta fiind realizată în mod normal. În caz contrar (NT=1) este semnalată existenţa unui task imbricat, revenirea la task-ul ce anterior cedase controlul fiind posibilă prin apelul instrucţiunii de ieşire din întrerupere (IRET).

La intrarea în scenă a microprocesoarelor pe 32 de biţi, registrul indicatorilor de condiţii a fost completat cu două fanioane de sistem şi un nou flag de control, acesta fiind:

Bitul 16: indicatorul de restaurare a execuţiei RF (Resume Flag). În funcţie de valoarea acestuia, în etapa de depanare a programelor se poate activa sau inhiba generarea aşa-numitelor *excepţii* la execuţia pas cu pas, influenţând revenirea la firul de execuţie normal.

Bitul 17: indicatorul de comutare în mod real/virtual VM (Virtual Mode). Prin intermediul acestui flag, utilizatorul are posibilitatea de a determina un procesor (de clasă superioară) să execute codul ca un procesor virtual 8086 sau, dimpotrivă, să interpreteze instrucţiunile conform propriului mod de acţiune.

Bitul 18: indicatorul de verificare a alinierii AC (Alignment Check). Disponibil doar la procesoarele 80486 şi Pentium, acest bit activează sau inhibă controlul alinierii la efectuarea oricărei operaţii de acces la memorie, conform caracteristicilor impuse de anumite atribute de privilegiere.

Specifice exclusiv procesoarelor de clasă Pentium sunt trei flag-uri:

Bitul 19: indicatorul de întreruperi virtuale VIF (Virtual Interrupt Flag). Are acelaşi rol ca şi bitul IF (acceptarea sau respingerea întreruperilor externe mascabile), atunci când procesorul operează în modul virtual 8086.

Bitul 20: indicatorul de întreruperi virtuale în aşteptare VIP (Virtual Interrupt Pending). Atunci când microprocesorul funcţionează în modul virtual 8086, activarea acestui bit semnalizează existenţa unor cereri de întrerupere în aşteptare.

Bitul 21: indicatorul de activare a detecţiei clasei procesorului ID (IDentification). Principial, dacă utilizatorul este apt să exercite acţiuni asupra acestui bit, atunci repertoriul de instrucţiuni conţine un apel special (CPUID) ce oferă informaţii despre familia/modelul procesorului, fabricantul acestuia etc.

3.2. Moduri de adresare

Procedeele prin care se realizează în mod concret calculul adreselor (pentru instrucţiuni, respectiv operanzi) sunt cunoscute sub numele generic de *moduri de adresare*.

În vederea prezentării modurilor de adresare se va face apel la noţiunile de *adresă de segment* (AS), desemnând adresa de început a segmentului în care se găseşte operandul, respectiv de *adresă efectivă* (AE), specificând deplasamentul operandului în cadrul segmentului ce îl conţine. Pe baza AS şi AE se constituie *adresa fizică* (AF), conform mecanismului anterior descris.

Adresarea imediată

În acest caz, calculul adresei fizice a operandului nu este necesar, deoarece acesta apare chiar în corpul instrucţiunii. Un exemplu este dat de următoarea secvenţă de program în limbaj de asamblare:

```
mov ax,1Ah ; Incarca in registrul ax valoarea
           ; numerica imediata 1Ah
add ax,5   ; Aduna valoarea 5 la conținutul
           ; registrului ax
```

Se remarcă faptul că ambii operanzi sunt specificaţi sub forma unor valori numerice concrete (1Ah, 5) ce se constituie în argumente ale instrucţiunilor MOV şi ADD, localizarea lor fiind implicită (imediată).

Adresarea directă

Este modul de adresare este cel mai utilizat, el fiind caracterizat de specificarea directă a operandului prin intermediul deplasamentului său în cadrul segmentului de date ce îl conţine.

Offset-ul poate fi furnizat fie sub forma unei constante numerice (pe 16 biţi), fie sub forma unui nume simbolic (dacă acesta i-a fost asociat operandului în etapa de definire a datelor).

Următoarea secvenţă de program prezintă diferite apeluri bazate pe adresarea directă:

```
mov ah,ds:[0000h]             ; Incarca registrul ah cu
                              ; valoarea continuta in
                              ; memorie la adresa fizica
                              ; ds:0000 (0000 fiind
                              ; constanta ce defineste
                              ; offsetul)
mov BYTE PTR ds:[0001h],05h   ; Incarca octetul de la
                              ; adresa fizica ds:0001
                              ; cu valoarea 5
mov al,var01                  ; Incarca registrul al cu
                              ; valoarea variabilei
                              ; var01, numele variabilei
                              ; fiind de fapt un offset
                              ; (pointer la locatia
                              ; in care aceasta este
                              ; memorata)
```

Figura 3 prezintă schematic metodologia de determinare a adresei fizice AF în cazul adresării directe.

Ca observaţie, deşi acest mod de adresare foloseşte DS ca registru implicit de segment, menţionarea sa explicită în program este impusă de limitările sintactice ale asambloarelor uzuale.

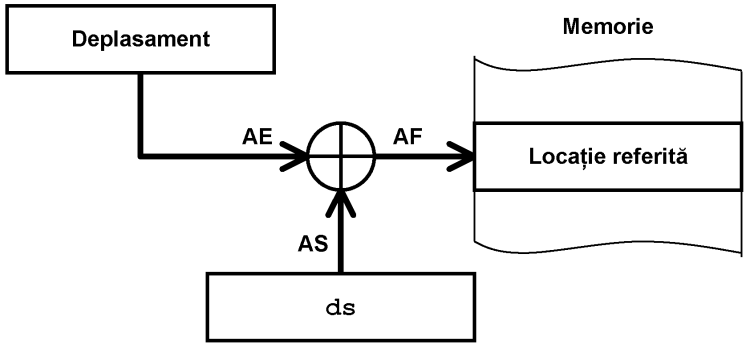


Figura 3. Formarea AF în cazul adresării directe.

Adresarea indirectă (prin registre)

Specificul acestui mod de adresare este reprezentat de faptul că adresa efectivă a operandului este dată de conţinutul registrelor BX, BP, SI sau DI (conform figurii 4). Registrul de segment implicit este considerat DS (dacă se folosesc registrele BX, SI sau DI), respectiv SS (dacă se foloseşte registrul BP). Un exemplu ilustrativ este dat de următoarea secvenţă de cod:

```
mov bx,0000h            ; Incarca registrul bx cu
                        ; valoarea offset-ului (0000h)
mov al,[bx]             ; Incarca registrul al cu
                        ; octetul de la adresa ds:0000
mov bp,0001h            ; Incarca registrul bx cu
                        ; valoarea offset-ului (0001h)
mov al,[bp]             ; Incarca registrul al cu
                        ; octetul de la adresa ss:0001

mov si,0002h            ; Incarca registrul bx cu
                        ; valoarea offset-ului (0002h)
mov al,[si]             ; Incarca registrul al cu
                        ; octetul de la adresa ds:0002

mov di,0003h            ; Incarca registrul bx cu
                        ; valoarea offset-ului (0003h)
mov al,[di]             ; Incarca registrul al cu
                        ; octetul de la adresa ds:0003

mov bx,0004h            ; Incarca registrul bx cu
                        ; valoarea offset-ului (0004h)
mov BYTE PTR [bx],07h   ; Incarca octetul de la adresa
                          ; ds:0004 cu valoarea 7
```

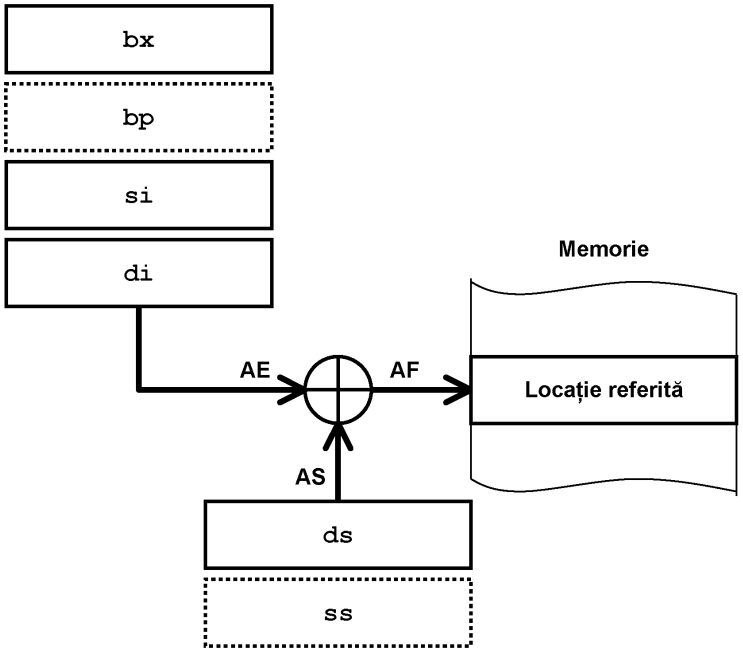


Figura 4. Formarea AF în cazul adresării indirecte (prin registre).

În această situaţie, asambloarele sunt capabile ca, prin analiza sintactică a formatului instrucţiunilor, să identifice registrele de segment implicite. Bineînțeles, utilizatorul este liber să specifice el însuşi un anumit registru de segment (fără a ține cont de cele implicite), astfel încât următoarea secvenţă de instrucţiuni este perfect validă:

```
mov bp,0006h   ; Incarca registrul bx cu valoarea
               ; offset-ului (0006h)
mov al,ds:[bp] ; Incarca registrul al cu octetul de la
               ; adresa ds:0006
```

Adresarea bazată sau indexată

În acest caz, adresa efectivă se obţine prin adunarea unui deplasament specificat – pe 8/16 biţi – la conţinutul unui registru de bază (BX sau BP) sau la conţinutul unui registru index (SI sau DI), conform reprezentării schematice din figura 5.

Şi în acest caz registrul de segment implicit este considerat DS (dacă se folosesc registrele BX, SI sau DI), respectiv SS (dacă se foloseşte registrul BP). Asamblorul determină în mod univoc registrele de segment în utilizare implicită, existând totuşi şi posibilitatea specificării explicite a acestora de către utilizator (prin mijlocirea prefixelor de segment).

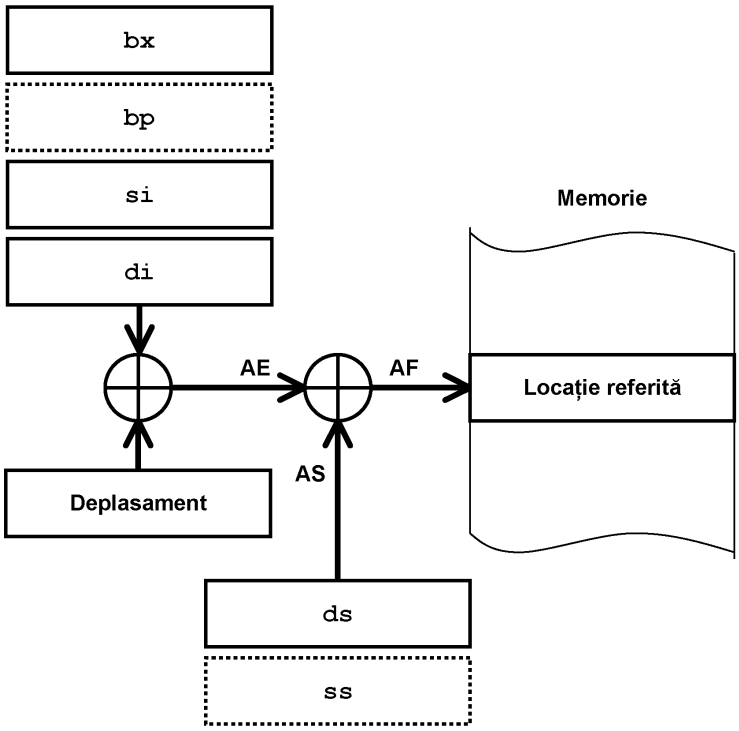


Figura 5. Formarea AF în cazul adresării bazate sau indexate

Ilustrativă în acest sens este următoarea secvenţă de program:

```
mov bx,01h              ; Se incarca registrul de
                        ; baza bx cu valoarea 1
mov ah,string[bx]       ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+1,
                        ; numele variabilei "string"
                        ; fiind de fapt un deplasament
                        ; (pointer la locatia in care
                        ; aceasta este memorata)

mov bx,02h              ; Se incarca registrul de
                        ; baza bx cu valoarea 2
mov ah,bx[string]       ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+2

mov bx,03h              ; Se incarca registrul de
                        ; baza bx cu valoarea 3
mov ah,[bx+string]      ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+3

mov bx,04h              ; Se incarca registrul de
                        ; baza bx cu valoarea 4
mov ah,[bx].string      ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+4

mov bp,05h              ; Se incarca registrul de
                        ; baza bp cu valoarea 5
mov ah,ds:[bp].string   ; Se incarca ah cu valoarea
```

```
                        ; de la adresa ds:string+5,
                        ; cu specificarea explicita
                        ; a registrului de segment ds
```

Determinarea lungimii deplasamentului (pe 8 sau 16 biţi) revine în sarcina asamblorului, programatorul utilizând în codul sursă doar simple valori constante, fără precizări suplimentare.

Exemplul anterior ilustrează diverse forme de scriere acceptate (ce vor fi detaliate, sub aspectul semnificaţiei, în capitolul 3). Se observă că, din punct de vedere al mecanismului de adresare, BX este considerat *adresa de bază*, iar string un *deplasament* (constant), astfel încât accesul la entităţile stocate în memorie se face prin încărcarea lui BX cu diferite valori (modificându-se astfel poziţia bazei).

Este posibilă şi o modalitate alternativă, conform căreia în registrul BX se încarcă un offset constant (baza considerându-se fixă), iar accesul la entităţile memorate se face prin specificarea explicită a unui *indice* variabil, ca în exemplul ce urmează:

```
mov bx,00h              ; Se incarca registrul de
                        ; baza bx cu valoarea 0
mov ah,string[bx+6]     ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+bx+6
mov ah,string[bx].7     ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+bx+7
```

Adresarea bazată şi indexată

Acest caz este cel mai complex şi flexibil, fiind practic o combinaţie între toate celelalte moduri de adresare anterior prezentate. Concret, adresa efectivă se formează prin suma dintre conţinutul unuia dintre registrele de bază (BX sau BP), conţinutul unuia dintre registrele index (SI sau DI) şi deplasamentul 8/16 biţi – dacă acesta este specificat. Figura 6 sugerează modul de determinare a adresei fizice.

Registrele de segment implicite sunt DS (dacă se foloseşte BX cu SI sau DI), respectiv SS (dacă se foloseşte registrul BP în combinaţie cu SI sau DI). Desigur, programatorul poate specifica în mod explicit un anumit registru de segment, gestiunea a ceea ce se regăseşte în locaţiile referite revenindu-i însă în exclusivitate. O exemplificare a principiilor adresării bazate şi indexate este oferită de secvenţa de cod prezentată în continuare:

```
mov bx,0                ; Se incarca registrul de
                        ; baza bx cu valoarea 0
mov si,1                ; Se incarca registrul
                        ; index si cu valoarea 1
mov ah,string[bx][si]   ; Se incarca ah cu valoarea
                        ; de la adresa ds:string+0+1
                        ; (numele variabilei "string"
                        ; fiind un pointer la locatia
                        ; in care aceasta este
                        ; memorata)

mov ah,[string+bx+si]   ; Acelasi efect ca instructiunea
                        ; anterioara
mov ah,[bx][si].5       ; Se incarca ah cu valoarea
```

```
                          ; de la adresa ds:0+1+5

mov bp,0                  ; Se incarca registrul de
                          ; baza bp cu valoarea 0
mov di,0
mov ah,ds:[bp][di][4]     ; Se incarca ah cu valoarea
                          ; de la adresa ds:0+0+4
```

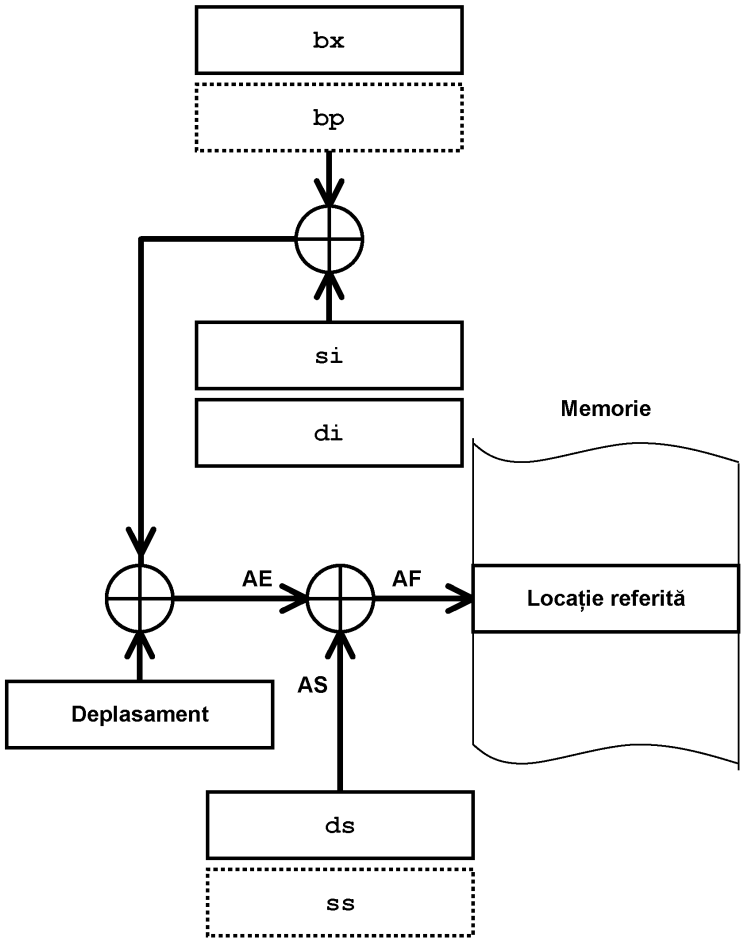


Figura 6. Formarea **AF** în cazul adresării bazate şi indexate.

4. Desfăşurarea lucrării

Pentru familiarizarea cu modurile de adresare prezentate, secvenţele de program ilustrative cuprinse în cadrul breviarului teoretic vor fi incluse în programe complete, funcționale. În sarcina studenţilor revine agregarea acestor programe, precum şi studiul execuției lor în Turbo-Debugger.

5. Modul de lucru

► Se editează programele complete, urmând ca acestea să includă secvențele anterior prezentate.

► Se asamblează şi link-editează sursele .ASM obţinute.

► Se încarcă executabilele generate în depanator. Prin execuţia pas cu pas se va urmări modificarea conţinutului registrelor procesorului şi al locaţiilor de memorie implicate.

LUCRAREA 9

INSTRUCȚIUNI ARITMETICE

1. Obiectivele lucrării

Această lucrare de laborator are ca obiectiv prezentarea instrucţiunilor aritmetice de bază, respectiv familiarizarea studenţilor cu modul de utilizare a acestora în programe.

2. Aparatura şi suporturile utilizate

► PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

În continuare vor fi prezentate instrucţiuni aritmetice ale limbajului de asamblare 80x86 specifice operaţiilor elementare (adunare, scădere, înmulţire, împărţire)..

3.1. Adunarea

Instrucţiunea ADD
Însumează cei doi operanzi, iar rezultatul este depus în operandul destinaţie (valoarea veche a operandului destinaţie fiind înlocuită cu noua valoare).
Sintaxa:

```
ADD destinaţie, sursă
```

Operanzii sursă şi destinaţie pot fi numere binare cu sau fără semn de 8 sau 16 biţi (de 32 de biţi începând cu 80386), dar ambii au aceeaşi dimensiune. Atât sursa cât şi destinaţia pot fi un operand registru sau un operand în memorie. Sursa poate fi şi un operand imediat. Întrucât operaţiile de la memorie la memorie nu sunt admise, operanzii sursă şi destinaţie nu pot fi ambii localizaţi în memorie. Instrucţiunea ADD actualizează indicatorii AF, CF, OF, PF, SF şi ZF.

Exemple:

```
add ax, [bx]    ;Aduna elementul  indicat de BX cu AX,
                ;iar rezultatul il depune in AX
add bx, 4       ;Aduna 4 cu valoarea din BX si depune în BX
add al, mem     ;Aduna mem cu AL si depune în AL
```

Instrucţiunea INC

Adaugă 1 la operandul destinaţie (care poate fi un octet sau un cuvânt), iar rezultatul este depus în operandul destinaţie (valoarea veche a operandului destinaţie fiind înlocuită cu noua valoare).

Sintaxa:

```
INC destinaţie
```

Operandul destinaţie poate fi un număr binar fără semn de 8 sau 16 biţi (respectiv 32 de biţi începând cu 80386). Destinaţia poate fi un operand registru sau un operand în memorie.

Instrucţiunea INC actualizează indicatorii AF, OF, PF, SF şi ZF. Deoarece valoarea operandului destinaţie este preluată ca un întreg fără semn, indicatorul CF nu este afectat.

Exemple:

```
inc  al   ;valoarea din al este incrementată cu 1
inc [bx]  ;valoarea indicată de  bx  este incrementată cu 1
inc mem   ;valoarea lui mem este incrementată cu 1
```

3.2. Scăderea

Instrucţiunea SUB

Operandul sursă este scăzut din destinaţie, rezultatul fiind depus în locaţia destinaţiei (valoarea veche a operandului destinaţie fiind înlocuită cu noua valoare).

Sintaxa:

```
SUB destinaţie, sursă
```

Operanzii sursă şi destinaţie pot fi numere binare cu sau fără semn de 8 sau 16 biţi (respectiv 32 de biţi începând cu 80386), dar ambii trebuie să aibă aceeaşi dimensiune. Atât sursa cât şi destinaţia pot fi un operand registru sau un operand în memorie. Sursa poate fi şi un operand imediat. Întrucât operaţiile de la memorie la memorie nu sunt admise, operanzii sursă şi destinaţie nu pot fi ambii localizaţi în memorie. Instrucţiunea SUB actualizează indicatorii AF, GF, OF, PF, SF şi ZF.

Operaţia de scădere poate fi interpretată ca scădere fie a numerelor cu semn, fie a celor fără semn. Responsabilitatea de a decide modul în care este interpretată scăderea revine programatorului. Tot programatorului îi revine sarcina de a interpreta în mod adecvat semnificaţia poziţionării indicatorilor pentru a deosebi operaţiile corect efectuate de cele ale căror rezultate sunt afectate de depăşirea capacităţii de reprezentare.

Exemple:

```
sub ax, [bx] ;Scade elementul indicat de BX din AX
             ;iar rezultatul îl depune în AX
sub bx,4     ;Scade 4 din valoarea lui BX şi depune în BX
sub al,mem   ;Scade mem din AL şi depune în AL
```

În acest exemplu, primele două instrucţiuni realizează scăderea pe 16 biţi (registrele AX şi BX au 16 biţi), iar cea de-a treia instrucţiune prezintă scăderea pe 8 biţi.

Instrucţiunea DEC

Scade 1 din operandul destinaţie (care poate fi un octet sau un cuvânt), iar rezultatul este depus în operandul destinaţie (valoarea veche a operandului destinaţie fiind înlocuită cu noua valoare).

Sintaxa:

```
DEC destinaţie
```

Operandul destinaţie poate fi un număr binar fără semn de 8 sau 16 biţi (respectiv 32 de biţi începând cu 80386). Destinaţia poate fi un operand registru sau un operand în memorie.

Instrucţiunea DEC actualizează indicatorii AF, OF, PF, SF şi ZF. Deoarece valoarea operandului destinaţie este preluată ca un întreg fără semn, indicatorul CF nu este afectat.

Exemple:

```
dec al   ;Valoarea din AL este decrementată cu unu
dec [bx] ;Valoarea indicata de BX este decrementată cu unu
dec mem  ;Valoarea lui mem este decrementată cu unu
```

Instrucţiunea NEG

Operandul destinaţie (el poate fi un octet sau un cuvânt) este scăzut din 0 iar rezultatul este depus tot în operandul destinaţie (valoarea veche a operandului destinaţie fiind înlocuita cu noua valoare). Astfel se formează complementul faţă de 2 al numărului, schimbându-se de fapt semnul.

Dacă operandul destinaţie este zero, semnul nu se schimbă. Dacă operandul destinaţie este un octet care conţine –128 sau un cuvânt care conţine -32768 sau un cuvânt dublu care conţine 65536, atunci el nu este schimbat şi este poziţionat indicatorul OF.

Sintaxa:

```
NEG destinaţie
```

Operandul destinaţie poate fi un număr binar cu semn de 8 sau 16 biţi (respectiv 32 de biţi începând cu 80386). Destinaţia poate fi un operand registru sau un operand în memorie. Instrucţiunea NEG actualizează indicatorii AF, CF, OF, PF, SF şi ZF.

Exemple:

```
neg al     ;Este schimbat semnul valorii lui AL
neg [bx]   ;Este schimbat semnul valorii indicată de BX
neg mem     ;Este schimbat semnul valorii lui mem
```

3.3. Înmulţirea

Instrucţiunea MUL

Efectuează o înmulţire (MULtiply) fără semn. Operandul sursă este înmulţit cu operandul acumulator. Dacă sursa este un octet, atunci ea este înmulţită cu registrul AL şi

rezultatul este depus în AH şi AL (deci registrul AX). Dacă sursa este un cuvânt, atunci ea este înmulţită cu registrul AX şi rezultatul este depus în regiştrii DX şi AX. Dacă sursa este un cuvânt dublu (începând cu 80386) atunci ea este înmulţită cu registrul EAX şi rezultatul este depus în regiştrii EDX şi EAX.

Sintaxa:

```
MUL sursă
```

Dacă partea cea mai semnificativă a rezultatului (AH pentru operandul sursă pe octet sau DX pentru operandul sursă pe cuvânt sau EDX pentru operandul sursă pe cuvânt dublu) este diferită de 0, atunci indicatorii CF şi OF sunt poziţionaţi (adică vor avea valoarea 1).

Exemple:

```
mul bl      ;Este înmulţit AL cu BL iar rezultatul este
            ;depus în AX
mul [bx]    ;Este înmulţită valoarea indicată de BX cu AX
            ;iar rezultatul este depus în DX:AX
mul mem     ;Este înmulţit mem (presupus pe 16 biţi) cu AX
            ;şi rezultatul este depus în DX:AX
```

Instrucţiunea IMUL

Spre deosebire de instrucţiunea MUL (care efectuează o înmulţire fără semn) instrucţiunea IMUL efectuează o înmulţire (Integer MULtiply) cu semn. Aceasta este singura diferenţă între cele 2 instrucţiuni. Operandul sursă este înmulţit cu acumulatorul. Dacă sursa este un octet, atunci ea este înmulţită cu registrul AL şi rezultatul este depus în AH şi AL (deci registrul AX). Dacă sursa este un cuvânt, atunci ea este înmulţită cu registrul AX şi rezultatul este depus în regiştrii DX şi AX. Dacă sursa este un cuvânt dublu (începând cu 80386), atunci ea este înmulţită cu registrul EAX iar rezultatul este depus în regiştrii EDX şi EAX.

Sintaxa:

```
IMUL sursă
```

Dacă partea cea mai semnificativă a rezultatului (AH pentru operandul sursă pe octet sau DX pentru operandul sursă pe cuvânt sau EDX pentru operandul sursă pe cuvânt dublu) este diferită de zero, atunci indicatorii CF şi OF sunt poziţionaţi (adică au valoarea 1).

Exemple:

```
imul bl    ;Este înmulţit BL cu AL şi rezultatul este depus
           ;în AX
imul [bx]  ;Este înmulţită valoarea indicată de BX cu AX
           ;şi rezultatul este depus în DX:AX
imul mem   ;Este înmulţit mem (presupus pe 16 biţi) cu AX
           ;iar rezultatul este depus în DX:AX
```

Începând cu 80186, IMUL are două sintaxe suplimentare:

```
IMUL registru, operand_imediat
IMUL registru, operand_in_memorie, operand_imediat
```

În această situaţie, cei doi operanzi care se înmulţesc sunt consideraţi pe 16 biţi iar destinaţia în care este depus rezultatul este întotdeauna un registru de 16 biţi. Începând cu 80386, aceste instrucţiuni pot fi extinse la 32 de biţi.

Se observă că aceste două sintaxe suplimentare permit specificarea a doi, respectiv trei operanzi. În situaţia cu doi operanzi produsul se face între operandul sursă care trebuie să fie un operand imediat şi operandul destinaţie care trebuie să fie un registru pe 16 sau 32 biţi. În situaţia folosirii a trei operanzi produsul se face între operandul sursă (operand_imediat) care trebuie să fie un operand imediat şi un operand în memorie (operand_in_memorie) iar rezultatul este depus în operandul destinaţie care trebuie să fie un registru pe 16 sau 32 de biţi.

Exemple:

```
imul dx,137     ;Este înmulţit 137 cu DX iar rezultatul este
                ;depus în DX
imul ax,[bx],8 ;Este înmulţită valoarea indicata de BX cu 8
                ;iar rezultatul este depus în AX
```

Începând cu 80386, IMUL are şi o altă sintaxă suplimentară:

```
IMUL  registru, sursa
```

Operandul destinaţie (registru) nu poate fi decât un registru pe 16 sau 32 biţi iar operandul poate fi atât un registru cât şi o variabilă de memorie de aceeaşi dimensiune cu destinaţia.

Exemple:

```
imul dx,ax    ;este înmulţit AX cu DX iar rezultatul este
              ;depus în DX
imul ax,[bx] ;este înmulţită valoarea indicată de bx cu ax,
              ;rezultatul fiind depus în ax
imul bx,mem  ;este înmulţită valoarea lui mem cu bx iar
              ;rezultatul este depus în bx
```

3.4. Împărţirea

Instrucţiunea DIV

Aceasta efectuează o împărţire fără semn. Acumulatorul şi (eventual) extensia lui vor fi împărţite la operandul sursă. Dacă sursa este un octet, atunci câtul este depus în AL, iar restul în AH. Dacă sursa este un cuvânt dublu (începând cu 80386), atunci câtul e depus în EAX, iar restul în EDX.

Sintaxa:

```
DIV sursa
```

Dacă restul depăşeşte capacitatea registrului destinaţie (FFh pentru sursă pe octet, FFFFh pentru sursă pe un cuvânt, FFFFFFFFh pentru sursă pe un cuvânt dublu) sau dacă a fost încercată o împărţire cu zero, atunci câtul şi restul vor avea valori nedefinite şi este generată o întrerupere. Câtul care nu are valoare întreagă este trunchiat la întreg.

Exemple:

```
div bx      ;este împărţit AX la BX depune câtul în AL şi
            ;restul în AH
div [cx]    ;împărţirea lui AX la valoarea indicată de CX
            ;câtul e depus în AL iar restul în AH
div mem     ;este împărţit AX la mem (pe 16 biţi) câtul e
            ;depus în AL şi restul în AH.
```

Instrucţiunea IDIV

Această instrucţiune efectuează o împărţire cu semn. Acumulatorul şi (eventual) extensia lui vor fi împărţite la operandul sursă. Dacă sursa este un octet, atunci câtul este depus în AL iar restul în AH. Dacă sursa este un cuvânt dublu (începând cu 80386), atunci câtul e depus în AX iar restul în DX.

Sintaxa:

```
IDIV sursa
```

În cazul în care câtul este pozitiv şi depăşeşte valoarea maximă FFh pentru sursa pe octet, FFFFh pentru sursa pe un cuvânt, FFFFFFFFh pentru sursa pe cuvânt dublu sau este negativ şi depăşeşte valoarea minimă 81h pentru sursa pe octet, 8001h pentru sursă pe cuvânt, 80000001h pentru sursă pe cuvânt dublu, atunci câtul şi restul vor avea valori nedefinite şi este generată o întrerupere.

Exemple:

```
idiv bx     ;este împărţit AX la BX depune câtul în AL şi
            ;restul în AH
idiv [cx]   ;împărţirea lui AX la valoarea indicată de CX
            ;câtul e depus în AL iar restul în AH
idiv mem    ;este împărţit AX la mem (pe 16 biti) câtul e
            ;depus în AL şi restul este depus in AH.
```

Instrucţiunea CWD

Dublează spaţiul operandului din registrul AX (sau EAX) prin extinderea semnului şi stocarea rezultatului în registrul DX:AX (sau EDX:EAX). Instrucţiunea CWD copiază semnul valorii din registrul AX (bitul 15) în fiecare bit din registrul DX

Instrucţiunea CWD poate fi utilizată pentru a produce un deîmpărţit dublu-cuvânt dintr-un cuvânt înainte de împărţire.

Precizare:

Ţinând cont de cele de mai sus, se observă faptul că în expresia W=X/Y*Z este mai bine să se calculeze rezultatul înmulţirii Y*Z pentru că instrucţiunea IMUL produce un rezultat pe 32 biţi (presupunând operanzi de 16 biţi). Făcând întâi înmulţirea se extinde automat semnul produsului în registrul DX aşa că nu mai trebuie extins semnul AX înainte de împărţire (cu instrucţiunea CWD).

4. Desfăşurarea lucrării

1. Fie expresia r=((a-b*c)/(d+15))/f.

Pentru a calcula această expresie se cunosc următoarele:

► Toate numerele sunt reprezentate cu semn;

► Variabilele de intrare şi r sunt reprezentate pe 16 biţi;

► b*c si a-b*c sunt reprezentate pe 32 de biţi, iar restul subexpresiilor pe 16 biţi.

Pentru efectuarea acestui calcul, este propus următorul program, pe care studenţii vor trebui să-l analizeze şi să-l aducă la forma executabilă.

```
_DATA segment
a dw 10000
b dw 300
c dw 30
d dw 85
f dw 2
r dw ?
_DATA ends
assume ds:_DATA
assume cs:_TEXT
_TEXT segment
start:
  mov ax, _DATA
  mov ds, ax
  mov ax,b
  imul c
  mov bx,a
  sub bx,ax
  mov ax,bx
  mov bx,d
  add bx,15
  idiv bx
  idiv f
  mov r, ax
  mov ax, 4C00h
  int 21h
_TEXT ends
_STACK segment stack
  db 512 dup (?)
_STACK ends
end start
```

2. Fie expresia r=[(a+b)*c+1)-250]/10 –1 .
Pentru a calcula această expresie se cunosc următoarele:

► Toate numerele sunt reprezentate cu semn;
► Variabilele de intrare şi r sunt reprezentate pe 16 biţi.

Pentru acest calcul este propus următorul program, ce conţine însă (în mod intenţionat) o eroare. Studenţii vor trebui să o corecteze şi apoi să aducă programul la forma executabilă.

```
_DATA segment
a dw 555
b dw 444
c dw 1
f dw 2
r dw ?
_DATA ends
assume ds:_DATA
assume cs:_TEXT
_TEXT segment
start:
  mov ax, _DATA
  mov ds, ax
  mov ax,a
  add ax,b
  imul c
  inc ax
  sub ax,250
  idiv 2
  dec ax
  mov ax, 4C00h
  int 21h
_TEXT ends
_STACK segment stack
  db 512 dup (?)
_STACK ends
end start
```

5. Modul de lucru

► Se vor asambla şi link-edita programele prezentate obţinându-se fişiere .EXE;
► Se vor rula sub TD programele prezentate, pas cu pas, urmărindu-se modificarea conţinutului regiştrilor.
► Se dau expresia W=W-Y-Z şi secvenţa de program propusă pentru rezolvarea ei:

```
mov ax,y
sub ax,z
sub w,ax
```

Acest program este greşit. Se cer identificarea erorii şi corectarea acesteia.

► Se dă expresia r={a+(b*c)-d-(e-20)]/(b/2+c)}+(d*a-100)+1. Se cere alcătuirea unui program pentru efectuarea acestui calcul. Se vor folosi pentru verificare următoarele valori: a=92, b=4, c=2, d=20, e=20. Programul va fi rulat pas cu pas, urmărind cu debuggerul modificarea conținutului regiştrilor.

►Următorul program realizează împărțirea -7 la 2

```
_DATA segment
a dw -7
b dw 2
_DATA ends
assume ds:_DATA
assume cs:_TEXT
_TEXT segment
start:
 mov ds,ax
 mov ax,a
 mov bx,b
 idiv bx
 mov ax, 4C00h
 int 21h
_TEXT ends
_STACK segment stack
  db 512 dup (?)
_STACK ends
end start
```

Se regăseşte câtul -4 în AX şi restul 1 in DX? Se cere motivarea răspunsului.

LUCRAREA 10

INSTRUCŢIUNI LOGICE ŞI DE DEPLASARE

1. Obiectivele lucrării

Lucrarea urmăreşte familiarizarea studenţilor cu instrucţiunile logice şi de deplasare. Se vor prezenta instrucţiunile, studentul având astfel posibilitatea de a înţelege şi învăţa modul de lucru, urmând ca în cadrul aplicaţiilor să poată utiliza tehnicile de programare bazate pe aceste instrucţiuni.

2. Aparatura şi suporturile utilizate

►PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
►Precizările din prezentul îndrumar.

3. Breviar teoretic

În următoarele pagini vor fi prezentate instrucţiunile logice si de deplasare însoţite de câteva exemple pentru o mai bună înţelegere a acestora.

3.1. Instrucţiuni logice

Instrucţiunile logice realizează funcţiile logice de bază, pe octet sau pe cuvânt. Operaţiile se fac la nivel de bit, deci se aplică funcţia logică respectivă tuturor biţilor sau perechilor de biţi corespunzători din operanzi. Instrucţiunile logice definite sunt: NOT, AND, OR, XOR şi TEST

Comportamentul lor este asemănător, cu excepţia instrucţiunii NOT care nu afectează niciun indicator (flag) şi a instrucţiunii TEST care nu stochează rezultatul funcţiei logice. Efectul tuturor instrucţiunilor logice asupra indicatorilor (cu excepţia instrucţiunii NOT menţionată mai sus) este următorul:

►indicatorul de depăşire OF şi cel de transport CF sunt întotdeauna poziţionaţi la zero, iar indicatorul de transport auxiliar AF este întotdeauna nedefinit în urma operaţiilor logice;
►indicatorul de semn SF, de paritate PF şi zero ZF sunt întotdeauna poziţionaţi să reflecte rezultatul operaţiei şi pot fi folosiţi în cadrul instrucţiunilor de salt condiţionat.

Instrucţiunea NOT
Sintaxa:

```
NOT destinaţie
```

Instrucţiunea NOT execută o negare logică bit cu bit (inversează biţii) pentru operandul destinaţie, ce poate fi un registru, o locaţie de memorie de 8 sau 16 biţi. Instrucţiunea NOT are un singur operand, celelalte având doi operanzi.

Exemplu:

```
.DATA
masca DB 01000010b
.CODE
...
not masca    ;masca devine 10111101b
```

Instrucţiunea AND

Sintaxa:

```
AND destinaţie, sursa
```

Instrucţiunea AND execută operaţia ŞI logic între cei doi operanzi (regiştri, locaţii de 8 sau 16 biţi) şi returnează rezultatul în operandul destinaţie. Operandul sursă poate fi şi o constantă.

Exemplu:

```
MOV AX,035h  ; AX = 00110101b , muta valoarea 035h in AX
AND AX,0FBh  ; AX = AX AND 11111011b = 00110001b
AND AX,0F8h  ; AX = AX AND 11111000b = 00110000b
```

În acest exemplu se observă că 0FBh este o mască ce are bitul de rang 2 egal cu 0. Prima instrucţiune AND foloseşte această mască pentru a şterge bitul 2. Asemănător, masca 0F8h utilizată de a doua instrucţiune AND şterge biţii 2, 1 şi 0 deoarece ea are biţii 2, 1 şi 0 egali cu 0.

Ca o simplă aplicaţie, instrucţiunea AND poate fi folosită pentru transformarea unui caracter în literă mare prin ştergerea bitului 5 (dacă bitul 5 era deja 0, atunci instrucţiunea AND nu are niciun efect deoarece bitul 5 este şters la literele mari).

```
mov ah, 7          ;funcţia 7: citeste caracter fără ecou
int 21h
and a1, 11011111b ;ştergere bit 5
cmp a1,'Y'         ;este Y?
je da              ;daca da, executa secvenţa da
```

Instrucţiunea TEST

Sintaxa:

```
TEST destinaţie, sursa
```

Instrucţiunea TEST are aceiaşi operanzi şi execută aceeaşi funcţie logică (ŞI) precum instrucţiunea AND, singura diferenţă fiind că instrucţiunea TEST nu modifică destinaţia, actualizând doar indicatorii.

Instrucţiunea OR

Sintaxa:

```
OR destinaţie, sursă
```

Instrucţiunea OR execută operaţia SAU logic a celor doi operanzi (regiştri, locaţii de 8 sau 16 biţi) şi returnează rezultatul în operandul destinaţie. Operandul sursă poate fi şi o constantă.

Exemplu:

Instrucţiunea OR poate fi folosită pentru poziţionarea valorii biţilor specifici, indiferent de poziţionările lor curente. Pentru a realiza acest lucru, se plasează valoarea destinaţiei într-un operand, iar în celălalt operand se plasează masca biţilor ce se şterg. Biţii măştii sunt 1 pentru oricare dintre poziţiile de biţi ce se doresc a fi poziţionaţi si 0 pentru oricare dintre poziţiile ce rămân neschimbate.

```
MOV AX, 035h    ;AX = 00110101b
OR  AX,  08h    ;AX = AX OR 00001000 = 00111101b
OR  AX,  07h    ;AX = AX OR 00000111 = 00111111b
```

Instrucţiunea XOR

Sintaxa:

```
XOR destinaţie, sursa
```

Instrucţiunea XOR execută operaţia SAU EXCLUSIV logic pentru cei doi operanzi (regiştri, locaţii de 8 sau 16 biţi) şi returnează rezultatul în operandul destinaţie. În rezultat se poziţionează un bit la 1 daca biţii corespunzători ai operanzilor iniţiali au valori diferite (unul este 1, celălalt 0), altfel bitul este şters.

Exemplu:

Instrucţiunea XOR poate fi folosită pentru a comuta valoarea unor biţi specificaţi. Acest lucru se poate realiza plasând valoarea destinaţiei într-unul dintre operanzi iar o mască a biţilor ce urmează a fi comutaţi în celălalt operand.

```
MOV AX, 035h   ; AX = 00110101b
XOR AX,  07h   ; AX = AX XOR 00000111b = 00111010b
```

XOR poate fi folosită şi pentru a poziţiona un registru pe 0 (pornind de la constatarea că atunci când cei doi operanzi sunt identici, prin SAU EXCLUSIV pe biţi, fiecare bit se anulează singur, producând 0), ca de exemplu:

```
XOR CX, CX        ; echivalent cu
MOV CX, 0
```

Singurul avantaj al instrucţiunii XOR este acela că nu afectează nici un indicator.

3.2. Instrucţiuni de deplasare

Acest grup de instrucţiuni realizează operaţii de deplasare la nivel de bit. Instrucţiunile au 2 operanzi: primul este operandul propriu-zis, iar al doilea este numărul de biţi cu care se deplasează primul operand. Ambele operaţii se pot face la stânga sau la dreapta. Biţii pot fi deplasaţi aritmetic sau logic în cadrul octeţilor. Numărul de deplasări poate fi 1 sau o valoare intre 1 şi 255 specificată in registrul CL.

Deplasarea cu un bit la stânga este echivalentă cu înmulţirea operandului cu 2, iar deplasarea la dreapta, cu împărţirea operandului la 2, de aceea deplasările aritmetice pot fi folosite pentru înmulţirea sau împărţirea numerelor binare cu puteri ale lui 2. Deplasările logice pot fi folosite pentru a izola biţii in interiorul octeţilor sau cuvintelor.

Ambele tipuri afectează indicatorii după cum urmează:

- ►Indicatorul AF (Auxiliary Carry Flag) este întotdeauna nedefinit în urma unei operaţii de deplasare;
- ►Indicatorii PF, SF şi ZF sunt actualizaţi normal ca în instrucţiunile logice;
- ►CF - conţine întotdeauna valoarea ultimului bit deplasat din operandul destinaţie;
- ►Conţinutul indicatorului OF este întotdeauna nedefinit în cazul unor operaţii de deplasare a mai multor biţi (în cazul în care se deplasează un singur bit, OF este poziţionat la 1 dacă valoarea bitului cel mai semnificativ a fost schimbată după operaţie).

Instrucţiunile SHL şi SAL

Sintaxa:

```
SHL destinaţie, contor
SAL destinaţie, contor
```

Instrucţiunile SHL (Shift logical Left) si SAL (Shift Arithmetic Left) execută aceeaşi operaţie şi fizic reprezintă aceeaşi instrucţiune. Cuvântul (sau octetul) destinaţie este deplasat la stânga cu numărul de biţi specificat de contor. În urma deplasării, în partea dreaptă a operatorului, se introduc biţi de zero.

Instrucţiunea SHR

Sintaxa:

```
SHR destinaţie, contor
```

Instrucţiunea SHR (Shift logical Right) deplasează biţii din operandul destinaţie (octet sau cuvânt) la dreapta, cu numărul de biţi specificaţi de contor. În partea stângă sunt introduşi biţi de zero.

Instrucţiunea SAR

Sintaxa:

```
SAR destinaţie, contor
```

Instrucţiunea SAR (Shift Arithmetic Right) deplasează biţii din operandul destinaţie (octet sau cuvânt) la dreapta, cu numărul de biţi specificaţi de contor. În partea stângă sunt introduşi biţi care au valoarea bitului de semn (bitul cel mai semnificativ), păstrând semnul valorii originale.

Figura 1 sintetizează efectul instruţiunilor prezentate anterior.

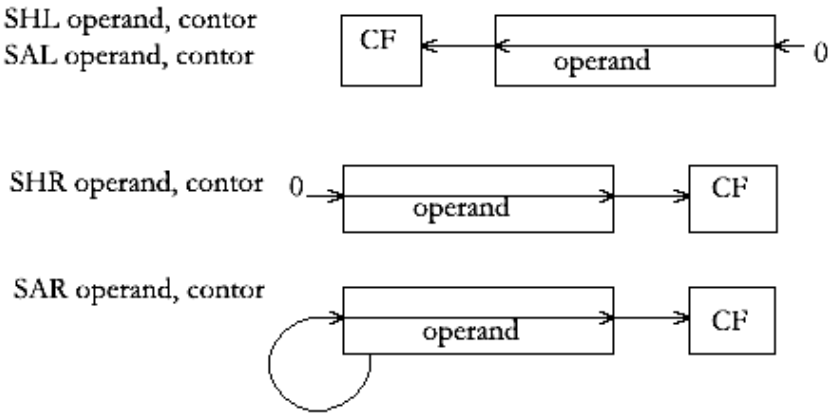


Figura 1. Instrucţiuni de deplasare

Observaţie: în mod asemănător acţionează instrucţiunile de rotire, deosebirea fiind aceea că biţii deplasaţi nu se pierd, ci sunt reintroduşi în câmpul destinaţie (în partea stângă a sensului rotirii).

4. Desfăşurarea lucrării

În cadrul aplicaţiei practice studenţii trebuie să identifice soluţiile câtorva probleme concrete, folosind instrucţiuni logice şi de deplasare.

Spre exemplificare, se cere introducerea, procesarea şi afişarea unor valori binare reprezentate pe cate 8 biţi. O variantă a aplicaţiei este prezentată în cod sursă în cele ce urmează.

```
; Datele de intrare: 2 variabile binare A si B,
; date in siruri de cate 8 de 0 si 1.
; Prelucrare: Se vor executa operatiuni logice
; cu cele 2 variabile, la nivel de octet
; Afisare: Rezultate se vor afisa pe ecran, in format binar
; ---------------------------------------------------------
.model tiny
.stack 100h
.data
 msgintro db "Program Demonstrativ pentru instructiunile logice
si de deplasare", 13, 10, "Datele introduse vor fi in format
binar (de ex: 01011001)", 13, 10, '$'
 msgreada db "Introduceti octetul A: ", '$'
 msgreadb db 13,10, "Introduceti octetul B: ", '$'
 msgcrlf  db 13,10, '$'
 msgaandb db 13,10, "A and B              : ", '$'
 msgaorb  db 13,10, "A or  B              : ", '$'
```

```
 msgaxorb db 13,10, "A xor B               : ", '$'
 msgend   db 13,10, "Ap", 134,"sati orice tast", 134," pentru a
termina...",13,10,'$'
 msgtypo  db 13,10, "V", 134, " rug", 134, "m s", 134, " tastati
0 sau 1!",13,10,'$'
 vara     db 0
 varb     db 0
 aux      db 0

.code
    start:
    mov ax, @data                   ; ds, es -> .data
    mov ds, ax
    mov es, ax
    call cls                        ; apelare procedura stergere
                                    ; ecran

    mov dx, offset msgintro         ; dx -> "Program Dem...
    mov ah, 09h                     ; functia afisare string
                                    ; terminat cu $
    int 21h                         ; apel intrerupere

    mov dx, offset msgreada         ; dx -> "\nIntroduceti oc..
    mov ah, 09h
    int 21h
    call readbinar                  ; citire numar binar in aux
    mov al, aux
    mov vara, al                    ; variabila A = aux

    mov dx, offset msgreadb         ; dx -> "\Introduceti oc..
    mov ah,09h
    int 21h
    call readbinar                  ; citire numar binar in aux
    mov al, aux
    mov varb, al                    ; variabila B = aux

    mov dx, offset msgaandb
    mov ah, 09h
    int 21h                         ; Afisare 'A and B:'
    mov al, vara
    and al, varb
    mov aux, al                     ; Valoarea A and B stocata
                                    ; in AUX
    call printbinar                 ; Afisare AUX

    mov dx, offset msgaorb
    mov ah, 09h
    int 21h                         ; Afisare 'A or B:'
    mov al, vara
    or al, varb
    mov aux, al                     ; Valoarea A or B stocata in
                                    ; AUX
    call printbinar                 ; Afisare AUX
```

```
mov dx, offset msgaxorb
mov ah, 09h
int 21h                          ; Afisare 'A xor B:'
mov al, vara
xor al, varb
mov aux, al                      ; Valoarea A xor B stocata
                                 ; in AUX
call printbinar                  ; afisare AUX
mov dx, offset msgend
mov ah, 09h
int 21h                          ; Afisare mesaj sfarsit
mov ah, 01h                      ; functia citire caracter
                                 ; (valoarea ; este ignorata)
int 21h                          ; apel intrerupere

call cls
mov al, 0h                       ; error code = 0
jmp iesi                         ; salt catre eticheta iesi
                                 ; (se iese din program)
typo: mov dx, offset msgtypo     ; se afiseaza cererea de
                                 ; tastare a 0 si 1
mov ah, 09h                      ; se asteapta o tasta si
                                 ; apoi se iese din program
int 21h
mov ah, 01h                      ; readkey
int 21h
call cls
mov al, 01h                      ; error code = 1
iesi:mov ah, 4ch                 ; dos exit, error code=0(ok)
                                 ; sau 1(input invalid)
int 21h

printbinar proc                  ; Procedura pentru afisare
                                 ; binar
mov cx, 8                        ; lungime 8 biti (octet)
burm: mov dl, '0'
test al, 128                     ; Se afiseaza valoarea
                                 ; B.C.M.S
                                 ; (se testeaza si salt in
                                 ; functie de ZF)
jz  d0                           ; apoi se face shift left
                                 ; bit cu bit pina cx = 0
                                 ; (toti 8 bitii au
mov dl, '1'                      ; fost afisati)
d0: push ax
mov ah, 02h                      ; functia 02, afisare
                                 ; caracter stocat in dl
int 21h
pop ax
shl al, 1
loop burm
ret
```

```
printbinar endp

readbinar proc                  ; Procedura Citire binar
mov aux, 0                      ; valoarea este stocata in
                                ; AUX
mov  cx, 8                      ; lungime 8 biti
bit:  mov  ah, 01h              ; functia 01h, citire
                                ; caracter in al
int  21h
sub  al, 48                     ; se scade 48 din codul
                                ; ascii
                                ; (48 = '0', 49 = '1') -> al
                                ; = 0/1
jb   typo                       ; cod ascii < 48 ->
                                ; caracter invalid
cmp  al, 2                      ; al > 2 -> caracter invalid
jae  typo
shl  ax, cl
shr  ax, 1
or   aux, al                    ; se seteaza bit cu bit AUX
loop bit
ret
readbinar endp

cls   proc                      ; Procedura stergere ecran
mov   ax,0b800h                 ; Ecranul se sterge prin
                                ; scrierea memoriei
                                ; video cu valori de 0
mov   es,ax
mov   ax,0700h
mov   cx,2000
xor   di,di
cld
rep   stosw
mov   dx,10                     ; se repozitioneaza cursorul
                                ; la 10:0
xor   bh,bh
mov   ah,2                      ; cu ajutorul functiei 2,
                                ; intreruperea 10h
int   10h
ret
cls  endp
end start
```

5. Modul de lucru

► Se va asambla şi link-edita programul de mai sus, obţinându-se fişierul .EXE.
► Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
► Se va rula programul sub *DEBUG*, urmărind modificarea valorilor din regiştrii afectaţi.
► Să se realizeze un program care determină şi afişează echivalentul binar al unui octet aflat la adresa „octe" sub forma unui şir de caractere ASCII ('0', '1').

LUCRAREA 11

INSTRUCŢIUNI DE TRANSFER

1. Obiectivele lucrării

Lucrarea urmăreşte familiarizarea studenţilor cu mediul de dezvoltare a programelor scrise în limbaj de asamblare pus la dispoziţie de către produsul Turbo-Assembler al firmei Borland, cu accent pe utilizarea instrucţiunilor de transfer specifice procesoarelor din familia Intel 80X86.

2. Aparatura şi suporturile utilizate

►PC în configuraţia: unitate centrală, monitor, tastatură, mouse.
►Precizările din prezentul îndrumar.

3. Breviar teoretic

3.1. Setul de instrucţiuni de transfer 80x86

Procesoarele din familia Intel dispun de un set puternic de instrucţiuni aritmetice, logice, de transfer şi de control,ceea ce permite încadrarea acestor procesoare în clasa CISC (Complex Instruction Set Computer).

Instrucţiunile de transfer deplasează date între memorie sau porturi de intrare/ieşire şi registrele procesorului, fără a executa niciun fel de prelucrare a datelor.

Instrucţiunile de transfer permit copierea unui octet sau cuvânt de la sursă la destinaţie. Destinaţia poate fi un registru, locaţie de memorie sau un port de ieşire, iar sursa poate fi un registru, o locaţie de memorie, un port de intrare sau o valoare numerică imediată. Ca regulă generală, destinaţia şi sursa nu pot fi ambele locaţii de memorie. În specificarea sursei şi destinaţiei se folosesc uzual notaţiile:

►`segment:offset` pentru adrese fizice;
►`(x)` paranteze rotunde pentru a desemna "conţinutul lui x".

Observaţie: Instrucţiunile de transfer nu modifică indicatorii de condiţie (flagurile) cu excepţia instrucţiunilor SAHF si POPF.

Instrucţiunile de transfer se clasifică în instrucţiuni generale, specifice acumulatorului, specifice adreselor şi specifice flagurilor.

A. Instrucţiuni de transfer generale: MOV, PUSH, POP, XCHG

Instrucţiunea MOV (Move Data - Transferă date)
Transferă un octet sau un cuvânt din operandul sursă în operandul destinaţie.

Forma generală a instrucţiunii MOV este

```
mov  destinație, sursă   ; (destinație)  ← sursa
```

Exemple:

```
mov ax,12          ; Adresare imediata in
                   ; operanzi registru
mov mem,12         ; Adresare imediata in adresare directa
mov mem[bx],12     ; Adresare imediata in adresare
                   ; indirecta
mov mem,ds         ; Registru de segment in memorie
mov mem,ax         ; Registru in adresare directa
mov mem[bx],ax     ; Registru in adresare indirecta
mov ax,mem         ; Adresare directa in registru
mov ax,mem[bx]     ; Adresare indirecta in registru
mov ds,mem         ; Memorie in registru de segment
mov ax,bx          ; Registru in registru
mov ds,ax          ; Registru general in registru
                   ; de segment
mov ax,ds          ; Registru de segment in
                   ; registru general
```

Instrucţiunile de mai sus ilustrează fiecare tip de transfer de memorie ce poate fi realizat cu o singură instrucţiune MOV. Următoarele instrucţiuni ilustrează câteva tipuri uzuale de transferuri care necesită două instrucţiuni MOV.

```
; Transfer imediat la registru de segment
mov ax,@data     ; Incarcare imediata intr-un
                 ; registru general
mov ds,ax        ; Stocare registru general
                 ; intr-un registru de segment

; Transfer memorie la memorie
mov ax,mem1      ; Incărcare memorie intr-un
                 ; registru general
mov mem2,ax      ; Stocare registru general în memorie

; Transfer registru segment la registru segment
mov ax,ds        ; Incarcare registru segment
                 ; intr-un registru general
mov es,ax        ; Stocare registru general în
                 ; registrul de segment dorit
```

Instrucţiunea PUSH (Push Data - Salvează date în stivă)

Decrementează registrul pointer de stivă (SP) cu doi şi apoi transferă un cuvânt din operandul sursă în noul vârf al stivei, adresat de SP.

Forma generală a instrucţiunii PUSH este

```
push  sursa
```

în care sursă este un operand pe 16 biţi (registru general de 16 biţi, registru de segment sau

locaţie de memorie) iar semnificaţia este "copiază sursa în vârful stivei".

Concret, execuţia instrucţiunii se face după secvenţa

```
(SP) ← (SP) - 2
SS : ((SP)+1 : (SP)) ← sursa
```

ceea ce înseamnă că se decrementează SP cu 2 şi în octeţii de la adresele (SP)+1 şi SP din segmentul de stivă se copiază operandul sursă. Copierea respectă regula de memorare a cantităţilor pe mai mulţi octeţi, partea mai puţin semnificativă (partea low) se memoreându-se la adrese mici. O exprimare detaliată a instrucţiunii ar putea fi:

```
(SP) ← (SP) - 2
SS : ((SP)+1) ← high (sursa)
SS : ((SP)) ← low(sursa)
```

Exemple de instrucţiuni corecte:

```
push  bx
push  es
push  [bx]
push  [bp+5]
push  es : [bx]  [si+4]
```

Exemplu de instrucţiune incorectă:

```
push al              ;  Operand pe 1 octet
```

Instrucţiunea POP (Pop Data - Reface date din stivă)

Transferă cuvântul aflat în vârful stivei (adresat de SP) la operandul destinaţie şi apoi incrementează registrul pointer de stivă (SP) cu doi pentru a pointa la noul vârf al stivei.

Forma generală a instrucţiunii POP este:

```
pop  destinaţie
```

unde *destinaţie* este un operand pe 16 biţi (registru general de 16 biţi, registru de segment sau locaţie de memorie), iar semnificaţia este "copiază conţinutul vârfului stivei în destinaţie".

Registrul CS nu poate apărea ca destinaţie. Concret, execuţia instrucţiunii se face după secvenţa

```
destinaţie ← SS : ((SP+1 : (SP))
(SP) ← (SP) + 2
```

ceea ce înseamnă că se transferă octeţii de la adresele (SP)+1 şi (SP) din segmentul de stivă în operand (destinaţie) şi apoi se incrementează SP cu 2. O exprimare detaliată a instrucţiunii ar putea fi

```
high (destinaţie) ← SS : ((SP)+1)
low (destinaţie)  ← SS : ((SP))
(SP) ← (SP) + 2
```

Exemple de instrucţiuni corecte:

```
pop  bx
pop  es
pop  es : [di]
pop  [bp+5]
pop  ss : [bx+4]
```

Exemple de instrucţiuni incorecte:

```
pop  al            ; Operand pe 1 octet
pop  cs            ; Registrul cs
```

Instrucţiunea XCHG(Exchange Data)
Interschimbă sursa cu destinaţia.
Forma generală a instrucţiunii XCHG este:

```
xchg  destinaţie,sursă
```

Restricţii:

► registrele de segment nu pot apărea ca operanzi;
►cel puţin un operand trebuie sa fie un registru general.

Exemple:

```
xchg al, ah
xchg alfa, ax
xchg sir [si], bx
xchg cs: [bx][si], ax
```

Interschimbarea conţinutului a doi operanzi din memorie op1 şi op2 se poate face prin secvenţa de instrucţiuni:

```
mov reg, op1
xchg reg, op2
mov op2, reg
```

B. Instrucţiuni de transfer specifice acumulatorului: XLAT, IN, OUT

Instrucţiunea XLAT (Translate - Translatează)
Forma generală a instrucţiunii XLAT este

```
xlat
```

Instrucţiunea nu are operanzi, semnificaţia fiind

```
(al)← ds: ((bx)+(al))
```

adică se transferă în AL conţinutul octetului de la adresa efectivă: (BX)+(AL).

Instrucţiunea se foloseşte la conversia unor tipuri de date, folosind tabele de conversie; adresa acestor tabele se introduce în BX, iar in AL se introduce poziţia elementului din tabel. De exemplu, conversia unei valori numerice cuprinsă între 0 şi 15 în cifrele hexazecimale corespunzătoare, se poate face prin

```
Tabel DB '0123456789abcdef''
. . . . . .
lea bx                          ; tabel
mov al, 11
xlat
```

În AL se va depune cifra hexazecimală B.

Instrucţiunea IN (Input Data)

Execută o citire de la portul de intrare specificat (pe 8 sau 16 biţi), destinaţia fiind registrul AL sau AX sau alte registre generale, pentru noile procesoare.

Forma generală a instrucţiunii IN este

```
in  destinaţie, port
```

unde:

►**destinaţie** este registrul AL sau AX (noile procesoare acceptă şi alte registre în locul lui AL, respectiv AX);

►**port** este o constantă cuprinsă între 0 şi 255 sau registrul DX.

Instrucţiunea OUT (Output Data)

Execută o scriere la portul de ieşire specificat de 8 sau 16 biţi, după cum se specifică registrul al sau ax.

Forma generală a instrucţiunii OUT este

```
out  port, sursa
```

unde:

►sursa - este registrul AL sau AX;

►port - este o constantă cuprinsă intre 0 şi 255 sau registrul DX.

C. Instrucţiuni de transfer specifice adreselor: LEA, LDS, LES

Instrucţiunea LEA (LOAD Effective Address)

Are ca efect încărcarea adresei efective (offsetul) într-un registru general.

Forma generală a instrucţiunii LEA este

```
lea registru, sursa
```

unde:

►sursa - este un operand aflat în memorie, specificat printr-un mod de adresare;
►registru - este un registru general.

Exemplu:

```
lea  bx, alfa
lea  si, alfa [bx][si]
```

Acelaşi efect se obţine folosind operandul OFFSET in instrucţiunea MOV:

```
mov  bx, offset alfa
mov  si, offset alfa [bx][si]
```

Instrucţiunea LDS (Load Data Segment)
Încarcă DS.
Forma generală a instrucţiunii LDS este:

```
lds  registru, sursa
```

unde:

►registru - este un registru general de 16 biţi;
►sursa - este un operand de tip cuvând dublu – word aflat în memorie, care conţine o adresă completă de 32 biţi.

Are ca efect transferul unei adrese complete în perechea de registre formată din DS şi registrul specificat în instrucţiune, adică:

```
(reg) ← ((sursa))
(ds)  ← ((sursa) + 2)
```

Exemplu:

```
alfa db  25
adr-alfa dd  alfa
……….
lds si, adr-alfa
mov byte ptr [si], 75
……….
```

Instrucţiunea LES (Load Extra Segment)
Încarcă ES.
Forma generală a instrucţiunii LES este

```
les    seg, sursa
```

unde:

- ►reg este un registru general;
- ►sursa este un operand de tip DD.

Efectul instrucţiunii este urmatorul: în registrul specificat se încarcă offsetul operandului sursă iar în registrul ES se încarcă adresa de segment (de la adresa sursă+2), adică:

```
(reg)← ((sursa))
(es) ← ((sursa) + 2)
```

Exemplu:

```
b dw 3467h
adr_b dw b
……....
les di,  adr - b
mov word ptr es: [di] ,                  75
```

Instrucţiunile LDS şi LES pot fi folosite la exploatarea tabelelor de adrese. Fie tabela de adrese:

```
Tab_adr dd R1, R2, R3, R4, R5
```

şi se doreşte încărcarea în AX conţinutului de la adresa Ri, al cărei indice este dat de registrul CX. Secvenţa de instrucţiuni

```
mov  bx, cx
mov  bx, bx     ; se dublează indicele 2 cx
mov  bx, bx     ; se dublează valoarea lui bx, adică 4 cx
lds  si, tab_adr [bx]       ; încarcă în si adresa Ri
mov  ax, ds: [si]           ; încarcă indirect valoarea
                            ; conţinută in Ri
```

încarcă în AX conţinutul adresei Ri.

O adresă fizică ocupă 4 octeţi, ca atare valoarea conţinută în CX trebuie înmulţită cu 4, operaţie realizată prin două instrucţiuni ADD BX, BX.

D. Instrucţiuni de transfer specifice flagurilor: LAHF, SAHF, PUSHF, POPF

Instrucţiunea LAHF (Load AH with Flags)

Încarcă AH cu indicatorii de condiţie, adică

AH ← FLAGS $_{0\div7}$

Instrucţiunea încarcă în registrul AH octetul dat de partea cea mai puţin semnificativă a registrului FLAGS, ce conţine indicatorii. Instrucţiunea nu are operanzi.

Instrucţiunea SAHF (Store AH into Flags)

Depune AH în registrul indicatorilor de condiţii.

Instrucţiunea încarcă în registrul FLAGS, în octetul cel mai puţin semnificativ, conţinutul registrului AH, adică

FLAGS ← $(AH)_{0\div7}$

Instrucţiunea nu are operanzi.

Instrucţiunea PUSHF(Push Flags)

Salvează registrul indicatorilor de condiţii în stivă.

Nu are operanzi, iar efectul este plasarea registrului FLAGS în vârful stivei.

```
(SP) ← (SP) - 2
SS : ((SP) + 2 : (SP)) ← flags
```

Instrucţiunea POPF(Pop Flags)

Este perechea instrucţiunii PUSHF:

```
flags ← SS : ((SP) + 1 : (SP))
(SP)  ← (SP) + 2
```

Instrucţiunile LAHF şi SAHF citesc şi scriu explicit partea mai puţin semnificativă a registrului de flaguri. Partea mai semnificativă poate fi citită şi modificată prin mici artificii cu instrucţiunile PUSHF şi POPF.

4. Desfăşurarea lucrării

Se propune spre analiză (teoretică şi experimentală) următorul program ce ilustrează modul de utilizare a unora din instrucţiunile prezentate în cadrul acestei lucrări.

```
_DATA segment
alfa dw 3
beta dd 3467h
gama dw 5
adrbeta dd beta
_DATA ends
assume ds:_DATA
assume cs:_TEXT
_TEXT segment
start:
mov ax,_DATA
mov ds,ax
mov ax,17           ; Adresare imediata a operandului
                    ; sursa care este o constanta zecimala
mov ax,10001b
mov ax,11h
mov alfa,ax         ; Adresare directa a operandului
                    ; destinatie
xchg ax,bx          ; Acelasi efect cu secventa anterioara
mov bx,OFFSET beta  ; Adresare imediata a operandului
                    ; sursa (adresa variabilei
                    ; operatorului OFFSET
```

```
lea bx,beta         ; Aceasta instructiune are acelasi
                    ; efect cu cea
                    ; anterioara cu toate ca sursa este
                    ; adresata direct
lea bx,gama
lea bx,alfa         ; (alfa are offsetul 0,beta offsetl
                    ; 2,gama offsetul 6.)
mov si,2
mov   cx,[bx][si]   ; Adresare bazata indexata a sursei
                    ; -->in cx se va
                    ; regasi var beta adica 3467h
mov cx,alfa+2       ; Aceasta instructiune are acelasi
                    ; efect cu cea anterioara
                    ; sursa fiind adresata direct
mov   cx,alfa[2]    ; Notatii echivalente pentru
mov cx,[alfa+2]     ; instructiunea anterioara
mov di,3
mov BYTE PTR [bx][di],55h ; Se va folosi aceasta varianta
                    ; cand se doreste o adresare la nivel
                    ; de octet; se scrie in
                    ; octetul 3 adica primul octet al
                    ; variabilei beta 55h astfel ca noua
                    ; variabila beta are valoarea 5567h
mov BYTE PTR alfa+3,55h ; Aceasta instructiune are
                    ; acelasi efect cu cea
                    ;anterioare, destinatia fiind insa
                    ; adresata direct. Desi alfa
                    ; este definit cu DW, operatia este
                    ; realizata la nivel de octet
mov ax, 4C00h
 int 21h
 _TEXT ends
_STACK segment stack
db 512 dup(?)
_STACK ends
end start
```

5. Modul de lucru

► Se va asambla şi rula programul prezentat obţinându-se fişierul *.EXE*.
► Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
► Se va rula sub *DEBUG* sau *TD* programul prezentate.
► Se cere scrierea a două secvenţe de program care să aibă efect echivalent cu cel al instrucţiunii XCHG. Secvenţele vor fi testate în scurte programe demonstrative.

LUCRAREA 12

OPERAŢII CU ŞIRURI

1. Obiectivele lucrării

Prin efectuarea acestei lucrări de laborator se urmăreşte familiarizarea studenţilor cu operaţiile cu şiruri de octeţi, cuvinte şi cuvinte duble.

2. Aparatura şi suporturile utilizate

► PC în configuraţia unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

Instrucţiunile pe şiruri pot opera asupra unor şiruri de maxim 64K, dar pot fi precedate de un prefix care permite repetarea lor la nivel hardware, permiţându-se astfel prelucrarea rapidă a şirurilor lungi. Elementele comune instrucţiunilor pe şiruri vor fi prezentate pe scurt în continuare.

3.1 Localizarea operanzilor

Referitor la acest aspect, se pot face următoarele observaţii

► O instrucţiune pe şiruri poate avea un operand destinaţie şi/sau un operand sursă;
► Şirul sursă se află în segmentul de date curent în mod implicit. Folosind un prefix de segment se poate specifica alt segment pentru şirul sursă;
► Operandul destinaţie se consideră localizat în extrasegmentul de date curent în mod implicit.

3.2 Utilizarea registrelor la operaţiile cu şiruri

În ceea ce priveşte contextul microprocesorului, sunt de amintit următoarele aspecte:

► Registrul SI (registrul sursă index) este folosit ca deplasament pentru a adresa elementul curent al şirului sursă;
► Registrul DI (registrul destinaţie index) este folosit ca deplasament al elementului curent în şirul destinaţie.
►Registrul CX reprezintă contorul pentru numărul de repetări (dacă se utilizează prefixul de repetare).

Observaţii:

► Aceste registre trebuie iniţializate pentru a adresa şirurile sursă/destinaţie înainte

de execuţia instrucţiunilor LDS, LES, LEA;

- ► Instrucţiunile pe şiruri actualizează automat registrele SI şi DI înainte de prelucrarea următorului element al şirului. În funcţie de valoarea indicatorului de direcţie DF, registrele de index se autoincrementează (DF=0) sau se autodecrementează (DF=1);
- ► SI şi DI sunt modificaţi cu 1 după fiecare operaţie dacă elementele şirului sunt octeţi, sau cu 2 dacă elementele sunt cuvinte;
- ► După fiecare instrucţiune, CX va fi decrementat cu 1. Dacă CX=1, instrucţiunea pe şiruri nu se va mai executa şi se predă controlul instrucţiunii următoare.

3.3 Prefixe de segment REP, REPE, REPZ, REPNE şi REPNZ

Prefixele de segment sunt mnemonice folosite în contextul utilizării instrucţiunilor pentru prelucrarea şirurilor. Sunt utilizate mnemonice diferite pentru a mări claritatea programului.

Concret, semnificaţia acestor prefixe este:

- ► REP: Este folosit împreună cu instrucţiunile MOVS şi STOS**.**
  Interpretare: repetă până la sfârşitul şirului (CX diferit de 0).
- ► REPE, REPZ: Sunt folosite cu instrucţiunile CMPS şi SCAS
  Interpretare: identice cu REP (indicatorul ZF este poziţionat la 1).
- ► REPNE, REPNZ: Sunt folosite cu instrucţiunile CMPS şi SCAS.
  Interpretare: identice cu REP (indicatorul ZF poziţionat la 0).

3.4 Instrucţiuni 80x86 pentru lucrul cu şiruri

Există posibilitatea utilizării a 5 tipuri diferite de instrucţiuni primitive cu care se poate opera asupra şirurilor din care pot deriva o serie de alte instrucţiuni utile pentru lucrul asupra unor şiruri pe octeţi, cuvinte sau cuvinte duble. De asemenea, există instrucţiuni dedicate controlului registrului de flag-uri care specifică modul de lucru al procesorului cu şirurile.

A. Instrucţiuni de transfer

Instrucţiunea MOVS (MOVe String)

Transferă o secvenţă de octeţi de la o locaţie de memorie la alta (transferă un octet sau un cuvânt din şirul sursă (adresat de SI) în şirul destinaţie (adresat de DI) şi actualizează SI şi DI să adreseze următorul element al şirului).

Sintaxa:

```
MOVS sir_destinaţie, sir_sursa
```

Observaţii:

- ► În cazul în care se utilizează şi un prefix de repetare, instrucţiunea MOVS poate fi folosită pentru mutarea unor blocuri de memorie;
- ► Dacă se utilizează instrucţiunea MOVS, asamblorul este cel care detectează tipul operanzilor angajaţi în operaţiunea de transfer.

Următorul exemplu ilustrează folosirea MOVS:

```
;Programul realizează copierea unui sir intr-o alta zona de
;memorie

.MODEL small
.STACK 100h
.DATA
    sir1 LABEL byte
                              ;Sirul este de tip octet
    DB 'ABC'
    lsir1 EQU ($-sir1)
    sir2 DB 3 DUP(?)

.CODE
    start:
    mov ax, @data
    mov ds,ax
    mov es,ax                 ;Registrele DS si ES vor avea
                              ;aceeasi valoare
    mov si,OFFSET sir1
    mov di,OFFSET sir2
    mov cx,lsir1
    cld
    iar:
         movs es: [sir2], [sir1]
    loop iar
                              ;Bucla
    mov ah, 4ch
    int 21h
    END start
```

Observaţie: Secvenţa de instrucţiuni

```
iar:
    movs es: [sir2],[sir1]
loop iar
```

poate fi înlocuită prin

```
    rep movs es: [sir2],[sir1]
```

REP indică repetarea executării instrucţiunii pentru adresarea şirului care îi urmează, de un număr de ori egal cu valoarea înscrisă în registrul CX.

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea MOVS se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

- ► MOVSB(MOVe String, Bytes) - specifică transfer pe octet;
- ► MOVSW(MOVe String, Words) - specifică transfer pe cuvânt;
- ► MOVSD(MOVe String, Double Words) - specifică transfer pe dublu cuvânt (valabilă începând cu 80386).

Sintaxa:

```
MOVSB
MOVSW
MOVSD
```

În continuare este prezentat un exemplu de utilizare a instrucţiunilor MOVSW şi a prefixelor de repetare.

```
;Programul realizeaza copierea unui bloc de 2000 octeti intr-o
;alta zona de memorie

.MODEL small
.STACK 100h
    bloc EQU 2000
.DATA
    blocs DB 2000 DUP(0)
    blocd DB 2000 DUP(?)
.CODE
    start:
    cld
                              ;Sensul crescator
    mov cx,bloc
                              ;Incarca in CX lungimea sirului
    mov si, OFFSET blocs
                              ;Incarca deplasarea blocului sursa
    mov ax,SEG blocs
                              ;Incarca adresa segmentului sursa
    mov ds,ax
                              ;Adresarea blocului sursa
    mov di,OFFSET blocd
                              ;Incarca deplasarea blocului
                              ;destinatie
    mov ax,SEG blocd
                              ;Incarca adresa segmentului
                              ;destinatie
    mov es,ax
                              ;Adresarea blocului destinatie
    rep movsw
                              ;Muta un cuvant
    mov ah, 4ch
    int 21h
    END start
```

Instrucţiunea LODS (LOad String)

Transferă un element (octet sau cuvânt) al şirului sursă în registrul acumulator (elementul adresat de SI este transferat în registrul AL (sau AX) şi este actualizat SI pentru a adresa următorul element al şirului).

Sintaxa:

```
LODS sir_sursa
```

Observaţii:

- ► În cazul instrucţiunii LODS nu este necesară folosirea prefixelor de repetare pentru a încărca registrul AL (sau AX) succesiv deoarece se păstrează de fapt doar ultima valoare.
- ► Dacă se utilizează instrucţiunea LODS, asamblorul este cel care detectează tipul operanzilor angajaţi în operaţiunea de transfer.

Programul următor este un exemplu ilustrativ de utilizare a funcţiei LODS.

```
.MODEL small
.STACK 100h
.DATA
    sir LABEL byte
    DB 0,1,2,3,4,5,6,7,8,9
    CODE
    start:
    cld
                              ;Reseteaza registrul de flag-uri
    mov cx, 10
                              ;Incarca lungimea
    mov si, OFFSET sir
                              ;Incarca deplasamentul sursei
    mov ah, 2
                              ;Functie de afisare caracter
    cit: lods sir
                              ;Obtine caracter
    add al, 48
                              ;Transfera caracterul in ASCII
    mov dl, al
                              ;Transfera in DL
    int 21h
    loop cit
    mov ah, 4ch
    int 21h
    END start
```

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea LODS se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

- ► LODSB(LOad String, Byte) - specifică transfer pe byte;
- ► LODSW(LOadString, Words) - specifică transfer pe cuvânt;
- ► LODSD(LOad String, Double Words) - specifică transfer pe cuvânt dublu (valabilă începând cu 80386).

Sintaxa:

```
LODSB
LODSW
LODSD
```

Instrucţiunea STOS (STOre String)

Principalul său rol este de a iniţializa şiruri şi tablouri cu o valoare constantă (este transferat conţinutul registrului acumulator într-un şir destinaţie).

Sintaxa:

```
STOS sir_destinatie
```

Utilizând prefixele de repetare, această instrucţiune se poate folosi pentru iniţializarea unui şir cu o valoare constantă.

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea STOS, se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

► STOSB(STOreString, Byte) - specifică transfer pe octet;
► STOSW(STOre String, Words) - specifică transfer pe cuvânt;
► STOSD(STOre String, Double Word) - specifică transfer pe cuvânt dublu (valabilă începând cu 80386).

Sintaxa:

```
STOSB
STOSW
STOSD
```

Exemplu de utilizare a funcţiei STOSB:

```
;Programul realizeaza initializarea unui sir de 10 octeti cu
;caracterul 'x'

.MODEL small
.STACK 100h
.DATA
    sir LABEL byte
    DB 100 DUP (?)
    lung EQU $ - sir
.CODE
    start:
                              ;Presupune ES=DS
    cld
                              ;Reseteaza registrii de flag-uri
    mov ax, 'x'
                              ;Incarca caracterul 'x'
    mov cx, lung
```

```
                                ;Incarca lungimea şirului
    mov di, OFFSET sir
                                ;Incarca adresa destinatie in
                                ;toti octetii
    rep  stosb
    mov ah, 4ch
    int 21h
    END start
```

B. Instrucţiuni de comparare

Instrucţiunea CMPS (CoMPare String)

Compară două blocuri de memorie unul cu celălalt, scăzând cuvântul (sau octetul) destinaţie, adresat de DI, din cuvântul (sau octetul) sursă, adresat de SI.

Sintaxa:

```
    CMPS sir_destinatie, sir_sursa
```

Observaţii:

► Utilizată împreună cu prefixele de repetare condiţionale CMPS, se poate executa până când este satisfăcută o anumită condiţie;
► CMPS afectează indicatorii, dar nu alterează niciun operand, actualizând SI şi DI să adreseze următorul element.

Exemplu de utilizare a funcţiei CMPS şi a prefixelor de repetare:

```
;Programul compara cate 1000 de elemente de cate 16 octeti
;(cuvinte), aflate in sirul sursa (sirs), respectiv destinatie
;(sird)

.MODEL small
.STACK 100h
    lung EQU 1000
.DATA
    sirs LABEL word
                                ;"sirs" este de tip cuvant
    DW lung DUP(?)
    sird LABEL word
    DW lung DUP(?)
.CODE
    start:
    mov ax,@data
    mov ds, ax
    mov ax,SEG sirs
                                ;Incarca adresa segment sursa
    mov ds,ax
                                ;Adresarea sirului sursa
    mov si,OFFSET sirs
                                ;Incarca deplasarea cuv. sursa
```

```
    mov ax,SEG sird
                              ;Incarca adresa segment destinatie
    mov es,ax
                              ;Adresarea sirului destinatie
    mov di,OFFSET sird
                              ;Incarca deplasarea sirului
                              ;destinatie
    cld
                              ;Reseteaza registrii de flag-uri
    mov cx,lung
                              ;Incarca in CX lungimea sirului
    repe cmps sird, es :sirs
                              ;Compara in mod repetat cate un
                              ;cuvant iar compararea inceteaza
                              ;la detectarea primei diferente
    jne diferit
    diferit:
    dec si
                              ;Scaderea cu 1 se face cu scopul
                              ;punctarii corecte a cuvintelor
                              ;diferite
    dec di
    mov ah, 4ch
    int 21h
    END start
```

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea CMPS se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

- ► CMPSB(CoMPare String, Bytes) - specifică comparare pe octet;
- ► CMPSW(CoMPareString,Words) - specifică comparare pe cuvânt;
- ► CMPSD(CoMPare String, Double words) - specifică comparare pe dublu cuvânt (valabilă începând cu 80386).

Sintaxa:

```
    CMPSB
    CMPSW
    CMPSD
```

C. Instrucţiuni de căutare

Instrucţiunea SCAS (SCAn String)

Caută într-un bloc de memorie un element particular, scăzând elementul din şirul destinaţie (octet sau cuvânt) adresat de DI din conţinutul registrului AL sau AX şi actualizând indicatorii (dar nu modifică conţinutul celor doi operanzi).

Sintaxa:

```
    SCAS sir_sursa
```

Observaţii:

- ► Instrucţiunea SCAS poate fi utilizată cu prefixele de repetare condiţionale pentru a compara elementele şirului cu AL (sau AX) până când este satisfăcută o anumită condiţie;
- ► În urma instrucţiunii este modificat şi DI pentru a adresa următorul element al şirului şi sunt actualizaţi indicatorii AF, CF, OF, PF, SF, ZF pentru a reflecta relaţia dintre elementul şirului şi AL (sau AX).

Exemplul următor ilustrează utilizarea funcţiei SCAS.

```
;Programul realizeaza cautarea unui caracter intr-un sir

.MODEL small
.STACK 100h
.DATA
    sir LABEL byte
    DB 'Textul in care se caută'
    lsir EQU $-sir
                              ;Lungime sir
    psir DD sir
                              ;Pointer far la sir
.CODE
    start:
    cld
                              ;Resetarea registrului de
                              ;flag-uri
    mov cx,lsir
                              ;Incarca lungime sir
    les di,psir
                              ;Incarca adresa sirului
    mov al,'x'
                              ;Incarca caracterul care trebuie
                              ;gasit
    repne scas ES:sir
                              ;Cauta caracterul 'x' din AL în
                              ;sirul 'sir'
    jne negasit
    negasit:                  ;CX este 0 daca nu este gasit
                              ;ES:DI refera caracterul dupa
                              ;primul 'x' negasit:
   mov ah, 4ch
    int 21h
    END start
```

Cînd nu se găseşte caracterul căutat, această situaţie se tratează ca un caz special: s-a presupus că ES nu este identic cu DS, însă adresa şirului este memorată într-o variabilă de tip pointer. Instrucţiunea LES este folosită pentru a încarcă adresa FAR a şirului în ES:DI.

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea SCAS, se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de

transfer:

- ► SCASB(SCAn String, Bytes) - specifică cautare pe octet;
- ► SCASW(SCAn String, Words) - specifică căutare pe cuvânt;
- ► SCASD(SCAn String, Double Words) - specifică pe dublu cuvânt (valabilă începând cu 80386).

Sintaxa:

```
SCASB
SCASW
SCASD
```

D. Instrucţiuni pentru regiştrii de flag:

Sunt instrucţiuni care manipulează flag-uri individuale, asigură servicii speciale din partea procesorului sau manevrează moduri privilegiate pentru operaţii.

Instrucţiunea CLD (CLear Direction flag)

Poziţionează indicatorul de direcţie (DF=0) pentru ca instrucţiunile pe şiruri să autoincrementeze registrele index SI şi/sau DI.

Sintaxa:

```
CLD
```

Instrucţiunea STD (SeT Direction flag)

Poziţionează indicatorul de direcţie (DF=1) pentru ca instrucţiunile pe şiruri să autodecrementeze registrele index SI şi/sau DI.

Sintaxa:

```
STD
```

Observaţie: CLD şi STD nu afectează ceilalţi indicatori.

E. Instrucţiuni pentru transferul şirurilor la şi de la porturi:

Instrucţiunile care urmează permit citirea unui şir de la un port şi depunerea sa în memorie sau scrierea unui şir din memorie la un port.

Instrucţiunea INS (INput String to port)

Citeşte un octet, un cuvânt sau un cuvânt dublu de la un port de intrare şi îl depune în memorie. Instrucţiunea se termină când CX=0. Pentru transferul datelor, în registrul CX se încarcă iniţial numărul de iteraţii. După decrementarea registrului CX, procesorul decrementează registrul DI cu 1, 2 sau cu 4 dacă indicatorul de control DF este egal cu 0. Dacă acest indicator DF este egal cu 1, registrul DI se decrementează cu 1, 2 sau cu 4.

Sintaxa:

```
INS destinatie, sursa
```

în care destinaţie este şirul ce urmează a fi transferat, iar sursa este indicată de registrul DS.

Observaţii:

► Combinată cu prefixul de repetare (REP), instrucţiunea INS va transfera un bloc de date de la portul specificat în memorie, în locaţii succesive;

► Dacă pe măsură ce şirul este transferat se doreşte şi prelucrarea sa, atunci nu se mai foloseşte prefixul de repetare a instrucţiunii, ci se realizează o buclă (de exemplu cu instrucţiunea LOOP);

► Numărul portului trebuie să se afle în registrul DX (portul nu poate fi specificat ca o valoare imediată).

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea INS se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

► INSB(Input String, Bytes) - specifică transfer pe octet;
► INSW(Input String, Words) - specifică transfer pe cuvânt;
► INSD(Input String, Double words) - specifică transfer pe dublu cuvânt (valabilă începând cu 80386).

Sintaxa:

```
INSB
INSW
INSD
```

Exemplu de utilizare a funcţiilor INSW:

```
.MODEL small
.STACK 100h
.486
                                ;directiva de procesor MASM
                                ;(Microsoft Macro Assembler) care
                                ;permite utilizarea in modul real
                                ;a unor functii ce opereaza in
                                ;modul protejat asupra unor
                                ;segmente de 16 biti sau 32
                                ;biti,disponibile doar pe tipuri
                                ;de procesoare mai
                                ;noi(80386,80486,80586,etc.)
.DATA
    nr EQU 1
    nport DW 300h
    t LABEL word

    .CODE
    start:
                                ;Se considera ES=DS
    cld
                                ;Reseteaza registrii de flag-uri
    mov cx, nr
                                ;Incarca nr octetilor de
```

```
                                ;transferat
    mov di, OFFSET t
                                ;Incarca adresa destinatiei
    mov dx, nport
                                ;Incarca numarul portului
    rep insw
                                ;Transfera sirul de la port la
                                ;zona tampon
    mov ah, 4ch
    int 21h
    END start
```

Instrucţiunea OUTS (OUTput String to port)

Transferă un octet, un cuvânt sau un cuvânt dublu din memorie la un port de ieşire. După fiecare transfer al unui octet, cuvânt sau cuvânt dublu, se decrementează registrul CX. Instrucţiunea se termină când CX=0. Prin urmare, pentru transferul datelor, în registrul CX se încarcă contorul pentru numărul de iteraţii. După decrementarea registrului CX, procesorul decrementează registrul DI cu 1, 2 sau cu 4 dacă indicatorul de control DF este egal cu 0. Dacă acest indicator DF este egal cu 1, registrul DI se decrementează cu 1, 2 sau cu 4.

Sintaxa:

```
    OUTS destinatie, sursa
```

Destinaţie: registrul DX;
Sursa : şirul ce urmează a fi transferat.

Observaţii:

► Combinată cu prefixul de repetare (REP) instrucţiunea INS va transfera un bloc de date din memorie (din locaţii succesive) la portul de ieşire specificat;

► Numărul portului trebuie să se afle in registrul DX, portul neputând fi specificat ca o valoare imediată;

► Şirul care urmează să fie transferat de OUTS este considerat şir sursă, astfel încât el este referit de DS:SI.

Exemplu de utilizare a funcţiei OUTS:

```
;Programul realizeaza transferul unui sir de caractere la un
;port de iesire

.MODEL small
.STACK 100h
.486
                                ;directiva de procesor MASM
                                ;(Microsoft Macro Assembler)
                                ;care permite utilizarea în
                                ;modul real a unor functii
                                ;ce opereaza in modul protejat
```

```
                                ;asupra unor segmente de 16 biti
                                ;sau 32 biti, disponibile doar
                                ;pe tipuri de procesoare mai noi
                                ;(80386,80486,80586, etc.)
.DATA
    nr EQU 100
    sir LABEL byte
    DB nr DUP(?)
    nport DW 300h
.CODE
    start:
                                ;Se considera ES=DS
    cld ;
                                ;Reseteaza registrii de flag-uri
    mov cx, nr
                                ;Incarca nr. octetilor de
                                ;transferat
    mov si, OFFSET sir
                                ;Incarca adresa sursei
    mov dx, nport
                                ;Incarca numarul portului
    rep outs dx,sir
                                ;Transfera sir din memorie la port
    mov ah, 4ch
    int 21h
    END start
```

Dacă nu se poate determina tipul elementelor, pentru instrucţiunea INS, se utilizează mnemonicele care specifică explicit tipul operanzilor implicaţi în operaţiunea de transfer:

- ► OUTSB(Output String, Bytes) - specifică transfer pe octet;
- ► OUTSW(Output String, Words) - specifică transfer pe cuvânt;
- ► OUTSD(Output String, Double Words) - specifică transfer pe cuvânt dublu (valabilă începând cu 80386).

Sintaxa:

```
    OUTSB
    OUTSW
    OUTSD
```

4. Desfăşurarea lucrării

Efectuarea acestei lucrări constă în analiza teoretică şi experimentală a programelor prezentate ca exemple ilustrative pentru utilizarea instrucţiunilor de lucru cu şiruri de octeţi şi cuvinte.

5. Modul de lucru

►Se vor asambla şi link-edita programele prezentate ca exemplu, obţinându-se fişiere *.EXE*.
►Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
►Se vor rula sub *TD* cel puţin două dintre programele prezentate.
►Să se scrie un program care să transfere un element al unui şir – octet, cuvânt sau dublu-cuvânt în registrul acumulator.
► Să se scrie un program care compară câte 1000 de elemente de câte 16 octeţi (cuvinte), aflate în şirul sursă (blocs), respectiv destinaţie (blocd) – utilizând funcţia CMPSW.
►Să se scrie un program care să realizeze căutarea unui caracter într-un şir utilizând funcţia SCASB.
►Să se scrie un program care să realizeze transferul unui bloc de caractere de la o zonă de memorie la un port de ieşire OUTSB.

LUCRAREA 13

DECIZIA SIMPLĂ ŞI COMPUSĂ. CICLURI

1. Obiectivele lucrării

Lucrarea prezentă îşi propune însuşirea unor tehnici uzuale de programare în limbaj de asamblare pentru implementarea structurilor decizionale, respectiv a celor ciclice.

2. Aparatura şi suporturile utilizate

►PC cu configuraţia unitate centrală, monitor, tastatură, mouse;
►Precizările din prezentul îndrumar de laborator.

3. Breviar teoretic

Majoritatea operaţiilor de bază în programarea structurată (decizia, selecţia, ciclurile cu test la partea inferioară şi superioară) implică în mod inerent evaluarea unor condiţii logice. În ASM, aceste condiţii sunt în general de tip comparaţie între valori numerice.

3.1. Decizia simplă şi decizia compusă

Decizia simplă, codificată în pseudo-cod prin

```
if (condiție)
    Ramura_if
```

se implementează în ASM prin şablonul

```
Evaluează condiție
    Salt condiționat (pe condiție falsă) la eticheta_1
   ; Ramura_if
eticheta_1:
```

Decizia compusă, codificată în pseudo-cod prin

```
if (condiție)
    Ramura_if
Else
  Ramura_else
```

se implementează după şablonul

```
Evaluează condiție
    Salt condiționat (pe condiție falsă) la et_1
```

```
        ; Ramura_if
        jmp et_2
et_1:
        ; Ramura_else
et_2:
```

Evaluarea *condițiilor logice simple* se realizează prin instrucțiuni de comparație, aritmetice etc., care poziționează bistabilii de condiție. De exemplu, secvența de pseudo-cod

```
if (ax  < bx)
        ; Ramura_if
else
        ; Ramura_else
```

se implementează prin

```
cmp ax,bx
jge et1:
; Ramura_if
jmp et2
et1:
; Ramura_else
et2:
```

Evaluarea *condițiilor complexe* se abordează într-o manieră graduală. Primul caz e cel care în subcondițiile sunt conectate prin operatorul logic AND. Astfel, se consideră o condiție logică de forma

C = C1 AND C2 AND C3 AND … AND Cn

şi decizia compusă

```
if (C)
        ; Ramura_if
else
        ; Ramura_else
```

Implementarea este următoarea:

```
Evaluează C1
  Salt condiționat (pe condiție falsă) la et_1
Evaluează C2
  Salt condiționat (pe condiție falsă) la et_1
……..
Evaluează Cn
  Salt condiționat (pe condiție falsă) la et_1
  ;
  ; Ramura_if
  ;
  jmp et_2
```

```
et_1:
  ;
  ; Ramura_else
  ;
et_2:
```

Se consideră, ca exemplu, secvența de pseudo-cod

```
if (car>='0' AND car<='9')
    {
       şir[i]=car-'0';
       i=i+1;
    }
```

în care se presupune că variabila se află în AL, iar indicele i în registrul BX. Implementarea e următoarea:

```
 cmp al,'0'            ; Evaluare al>='0'
 jb  et_1              ; Salt pe condiție falsă (al<'0')
 cmp al,'9'            ; Evaluare al<='9'
 ja  et_1              ; Salt pe condiție falsă (al>'9')
 sub al,'0'            ; Calcul car-'0'
 mov şir[bx],al        ; Depunere in şir[i]
et_1:
```

Al doilea caz de bază este cel în care subcondiţiile sunt conectate prin operatorul OR. Se consideră condiția compusă

C = C1 OR C2 OR C3 OR … OR Cn.

Implementarea este următoarea:

```
Evaluează C1
  Salt condiționat (pe condiție adevărată) la et_1
  Evaluează C2
  Salt condiționat (pe condiție adevărată) la et_1
……..
Evaluează Cn
  Salt condiționat (pe condiție adevărată) la et_1
  ;
  ; Ramura_else
  ;
  jmp et_2
et_1:
  ;
  ; Ramura_if
  ;
et_2:
```

A treia schemă de dezvoltare se referă la implementarea condiţiilor negate. O secvență de pseudo-cod de forma

```
if (NOT condiție)
    ; Ramura_if
else
    ; Ramura_else
```

se implementează la fel cu schema if-else obişnuită, dar cu inversarea saltului condiţionat:

```
Evaluează condiție
  Salt condiționat(pe condiție adevărată) la et_1
  ;
  ; Ramura_if
  ;
  jmp et_2
et_1
  ;
  ; Ramura_else
  ;
et_2:
```

Cu aceste trei operații de bază (AND, OR, NOT) se poate evalua orice tip de condiţie logică. Se consideră secvența pseudo-cod

```
if ((C1 AND C2) OR C3)
    ; Ramura_if
else
    ; Ramura_else
```

În acest caz se consideră subcondiţiile C1 AND C2 şi C3 şi se aplica pentru început şablonul de la operația OR:

```
Evaluează (C1 AND C2)
  Salt condiționat (pe condiție adevărată) la et_1
et_3:
Evaluează C3
  Salt condiționat (pe condiție adevărată) la et_1
  ;
  ; Ramura_else
  ;
  jmp et_2
et_1:
  ;
  ; Ramura_if
  ;
et_2:
```

În continuare, se detaliază evaluarea subcondiţiei (C1 AND C2), observând că se sare la eticheta et_1 dacă ambele subcondiţii C1 şi C2 sunt adevărate, altfel se sare la eticheta et_3:

```
Evaluează C1
  Salt condiționat (pe condiție falsă) la et_3
Evaluează C2
  Salt condiționat (pe condiție adevărată) la et_1
et_3:
Evaluează (C3)
  Salt condiționat (pe condiție adevărată) la et_1
  ;
  ; Ramura_else
  ;
  jmp et_2
et_1:
  ;
  ; Ramura_if
  ;
et_2:
```

După aceste modele, se pot evalua pas cu pas condiţii oricât de complicate. Şabloanele de evaluare se utilizează şi la celelalte operaţii din programarea structurată.

3.2. Cicluri cu test la partea superioară şi inferioară

Ciclul cu test la partea superioară, descris în pseudo-cod prin

```
while (condiție)
    ; Bloc
```

se implementează după şablonul

```
et_1:
  Evaluează condiție
  Salt condiționat (pe condiție falsă) la et_2
  ;
  ;
  ; Bloc
jmp et_1
```

Se consideră secvenţa următoare în limbajul C:

```
while (*s>='0' && *s<='9')
    {
       n=10 * n+*s - '0';
       s++;
    }
```

unde s este un pointer la char, iar n un întreg. Această secvenţă e tipică pentru conversia ASCII-întreg . Se presupune că variabila n se află în AX, iar că pointerul s este asignat în registrul SI şi se aplică şablonul de implementare

```
et_1:
  Evaluează ([SI]>='0')
  Salt pe condiție falsă la et_2
  Evaluează ([SI]<='9')
  Salt pe condiție falsă la et_2
  AX=AX*10+[SI]-'0'
  SI=SI+1
  Jmp et_1
et_2:
```

În codificarea propriu-zisă, se va încărca [SI] în registrul AL pentru o operare mai eficientă, conform exemplului următor:

```
  mov bx,10
  lea si,s
et_1:
  mov ax,n
  mov cl,[si]
  xor ch,ch       ; CX ← caracterul de la adresa SI
  cmp cl,'0'      ; Prima subcondiție
  jb  et_2        ; Salt pe condiție falsă
  cmp cl,'9'      ; A doua subcondiție
  ja  et_2        ; Salt pe condiție falsă
  mul bx          ; n=n*10
  sub cl,'0'      ; *s -'0'
  add ax,cx       ; Valoare finală n
  mov n,ax
  inc si          ; s++
  jmp et_1
```

Ciclul cu test la partea inferioară, descris în pseudo-cod prin una din formele

```
do                                    repeat
    ; Bloc              sau              ; Bloc
while (condiție)                      until (condiție)
```

se implementează după şabloanele

```
et:
  ; Bloc
Evaluează condiție
  Salt pe condiție adevărată la et
```

respectiv

```
et:
  ; Bloc
Evaluează condiție
  Salt pe condiție falsă la et
```

Se consideră un algoritm tipic pentru conversia întreg-ASCII, descris în limbajul C de secvența

```
do
{
    *s++=n%10+'0';
    n=n/10;
} while (n!=0);
    *s=0;
```

Prin împărțiri succesive la 10 se generează cifrele corespunzătoare întregului n şi se depun în şirul de caractere s (cifrele rezultă în ordine inversă). Implementarea e următoarea (se consideră n memorat în registrul AX, iar pointerul s în registrul index SI):

```
  mov bx,10
  lea si,s
et:
  xor dx,dx             ; Deimparțit=DX:AX
  div bx                ; AX=n/10, DX (DL)=n%10
  add dl,'0'            ; n%10+'0'
  mov [si],dl           ; Depunere in *s
  inc  si               ; si apoi incrementare s
  cmp ax,0              ; Compară n cu 0
  jnz  et               ; Reluare ciclu
  mov dl,'$'
  mov [si],dl
```

Un caz particular de ciclu cu test la partea superioară e ciclul cu contor ascendent, descris în pseudo-cod prin

```
for (contor=vi to vf step pas)
    ; Bloc
```

în care se consideră că pas e o valoare strict pozitivă. Dacă specificația pasului lipşeste, se consideră implicit pasul 1. Această descriere se poate detalia într-un ciclu de tip while şi o inițializare, conform schemei:

```
contor=vi;
while (contor<=vf) {
    ; Bloc
    contor=contor+pas; }
```

ceea ce arată că se poate aplica şablonul de implementare de la ciclul while. Ciclul cu contor descendent este descris în pseudo-cod prin:

```
for (contor=vi downto vf step pas)
    ; Bloc
```

în care, de asemenea, se consideră că pas e pozitiv şi implicit 1. Şi această formă se poate detalia în

```
contor=vi;
while (contor>=vf) {
    ; Bloc
    contor=contor-pas;
}
```

Se consideră, de exemplu, o secvență tipică de translatare la dreapta cu o poziție a elementelor unui tablou de întregi:

```
for (i=99 downto 1)
    TAB[i]=TAB[i-1];
```

Se asignează variabila i la registrul SI. Trebuie observat că indicii variază după elementele tabloului, dar adresele variază cu câte 2 octeți la fiecare element. Pentru o implementare ordonată, se vor alterna valorile variabilei indice SI exact ca în specificarea algoritmului şi se va ajusta temporar prin înmulţire cu 2 înaintea accesării elementelor tabloului. Detalierea ciclului este

```
i=99;
et_1:
  Evaluează (i>=1)
  Salt pe condiție falsă la et_2
  TAB[i]=TAB[i-1];
  i=i-1;
et_2:
```

iar implementarea este de forma

```
 mov di,99             ; i=99;
et_1:
  cmp di,1             ; Evaluează (i>=1)
  jl et_2              ; Salt pe condiție falsă (i<1)
  shl di,1             ; Temporar, DI← 2*DI
  mov ax,TAB[si]       ; TAB[i]
  mov TAB[si-2],ax     ; TAB[i-1]
  shr di,1             ; Refacere DI
  dec di               ; i=i-1
et_2:
```

Secvența de mai sus ar putea fi implementată şi prin instrucțiuni specifice şirurilor de cuvinte (se presupune că registrele DS şi ES indică segmentul în care e memorat TAB),

```
lea si, TAB[98*2]          ; Sursa initială
lea di, TAB[99*2]          ; Destinatie initială
```

```
std                             ; Sens descendent
mov cx, 99                      ; Numar iteraţii
rep movsw                       ; Transfer
```

O altă formă posibilă, care permite generalizări interesante e copierea la nivel de octet:

```
lea si,TAB[98*2+1]              ; începem de la
lea di,TAB[99*2+1]              ; ultimul octet
std
mov cx,99*2                     ; Numar de octeţi
rep movsb                       ; Copiere
```

În general, prelucrarea tablourilor de tip oarecare presupune înmulţirea indicilor de acces cu numărul de octeţi ai tipului de bază al tabloului. În acest sens, iniţializările registrelor SI şi DI din ultima secvenţă se pot generaliza astfel:

```
lea si, TAB[98*(TYPE TAB)+(TYPE TAB-1)]
lea di, TAB[99*(TYPE TAB)+(TYPE TAB-1)]
```

Dacă secvenţa de copiere propriu-zisă se înlocuieşte cu

```
mov cx, 99*(TYPE TAB)
rep movsb
```

atunci se obţine o secvenţă care implementează algoritmul dat, indiferent de tipul tabloului. Se consideră un tablou de structuri de tipul

```
MY_STRUC struc
    n dw ?
    text db 10 dup(0)
ENDS
```

tabloul având un număr de 20 de structuri:

```
.data
tab MY_REC 20 dup(<,>)
```

Secvenţa de translatare se poate scrie:

```
lea si, tab[(LENGTH tab-1)*(TYPE MY_REC)-1]
lea di, tab[(LENGTH tab)*(TYPE MY_REC)-1]
std
mov cx, (LENGTH tab-1)*(TYPE MY_REC)
rep movsb
```

prin care se poziţionează DI (destinaţia) pe ultimul octet al ultimei înregistrări din tablou, iar SI (sursa) pe ultimul octet al penultimului element din tablou. Numărul de iteraţii la

nivel de elemente e numărul de elemente al tabloului, micşorat cu o unitate; la nivel de octeţi se înmulţeşte această valoare cu dimensiunea unui element.

O altă formă posibilă de scriere foloseşte legătura dintre operatorii LENGTH, TYPE şi SIZE. De exemplu, iniţializarea contorului CX s-ar mai putea scrie

```
mov cx, SIZE tab-TYPE tab
```

În cazurile în care ciclul cu contor se poate descrie prin:

```
for (contor=n downto 1)
    ; Bloc
```

sau atunci când se repetă de n ori o anumită operaţie, iar valoarea curentă a contorului nu contează, ciclul se poate implementa prin instrucţiunea LOOP:

```
  mov cx, n
et:
  ; Bloc
  loop et
```

Se consideră o secvenţă de determinare a maximului şi a minimului unui tablou de întregi cu semn, definit prin

```
.data
TABLOU dw 100 dup(?)
n      dw ($-TABLOU)/2
val_max     dw ?
val_min     dw ?
i_max       dw ?
i_min       dw ?
```

Se doreşte determinarea atât a valorilor maxime şi minime, cât şi a indicilor pe care apar aceste elemente. Secvenţa se poate descrie în pseudo-cod prin

```
val_max=TABLOU[0];
val_min=TABLOU[0];
i_max=0;
i_min=0;
for(i=1 to n-1) {
    if (TABLOU[i]>val_max) {
        val_max=TABLOU[i];
        i_max=i;
    }
    else if(TABLOU[i]<val_min) {
        val_min=TABLOU[i];
        i_min=i;
    }
}
```

Pentru implementare, se va presupune că adresa elementului TABLOU[i] e alocată în registrul BX. Se va incrementa direct această adresă, iar ciclul va fi implementat printr-o instrucţiune LOOP. Indicele elementului curent se obţine printr-o diferenţă între adresa curentă (BX) şi adresa de început a tabloului (SI) şi o împărţire la 2.

```
.code
  lea bx, TABLOU  ; Adresa elementului TABLOU[0]
  mov si, bx      ; Copiată şi in SI
aici_1:
  mov cx, n
  dec cx          ; Sunt n-1 iteraţii
aici_2:
  mov ax, [bx]
  mov val_max, ax ; val_max=TABLOU[0]
  mov val_min, ax ; val_min=TABLOU[0]
  mov i_max, 0    ; i_max=0
  mov i_min, 0    ; i_min=0
  add bx, 2       ; Adresa elementului TABLOU[1]
et_1:
  mov ax, [bx]    ; AX←TABLOU[i]
  cmp ax, val_max ; Evaluează (TABLOU[i]>val_max)
  jle et_2        ; Salt pe condiţie negată
  mov val_max, ax ; val_max=TABLOU[i]
  push bx         ; Salvare adresă curentă
  sub bx, si      ; Adr. curentă-adr. de inceput
shr bx, 1; / 2
  mov i_max, bx   ; =indicele i_max
  pop bx          ; Refacere adresă curentă
et_2:
  cmp ax, val_min ; Evaluează (TABLOU[i]<val_min)
  jge et_3        ; Salt pe condiţie negată
  mov val_min, ax ; val_min=TABLOU[i]
  push bx         ; Salvare adresă curentă
  sub bx, si      ; Adr. curentă-adr. de inceput
shr bx, 1
  mov i_max, bx   ; =indicele i_min
  pop bx          ; Refacere adresă curentă
et_3:
  add bx, 2       ; Adresa elementului următor
  loop et_1       ; Ciclu dupa CX
```

Se observă că implementarea ciclului prin instrucţiunea LOOP complică determinarea indicelui elementului curent. Dacă s-ar fi implementat un ciclu ascendent obişnuit, înlocuind secvenţa dintre etichetele aici_1 şi aici_2 prin

```
aici_1:
  mov cx, 1
  cmp cx, n
  jl et_4
aici_2:
```

iar secvența de după eticheta et_3 prin

```
et_3:
  add bx, 2
  inc cx
  jmp et_1
et_4:
```

atunci, la fiecare iterație, indicele elementului curent ar fi fost disponibil în registrul CX, iar secvențele de determinare a indicilor i_max şi i_min s-ar fi redus la simple transferuri de forma

```
mov i_min, cx
```

Exemplul de mai sus arată că implementarea ciclurilor cu contor prin instrucțiunea LOOP, aparent mai simplă, poate conduce la complicații în interiorul ciclului.

Există şi cicluri cu mai multe puncte de ieşire (o ieşire normală şi una sau mai multe ieşiri forțate), ca în secvența pseudo-cod următoare, în care prin break s-a marcat ieşirea forțată din ciclu:

```
while (condiție_1) {
    ; Bloc_1
    if (condiție_2)
       break;
    ; Bloc_2
}
```

O asemenea situație se implementează combinând şabloanele de la while şi if:

```
et_1:
  Evaluează (condiție_1)
  Salt pe condiție falsă la et_2
  ; Bloc_1
  Evaluează (condiție_2)
  Salt pe condiție adevărată la et_2
  ; Bloc_2
  jmp et_1
et_2:
```

4. Desfăşurarea lucrării

Se consideră următorul program, în care e prezentat un algoritm tipic pentru conversia întreg-ASCII. Prin împărţiri succesive la 10 se generează cifrele corespunzătoare întregului n (memorat în AX) şi se depun în şirul de caractere s.

```
TITLE conversie.

.MODEL large
.STACK 1024h
.DATA
            n dw 112h
            s db 10 dup(?)
.CODE
start:
             mov ax,@data
             mov ds,ax
             mov ax,n
             mov bx,10
             lea si,s
et:
             xor dx,dx        ; Deîmpărțit = DX:AX
             div bx           ; AX = n/10, DX(DL)=n%10
             add dl,'0'       ; n%10+ '0'
             mov [si],dl      ; Depunere în *s
             inc si           ; şi apoi incrementare s
             cmp ax,0         ; Compară n cu 0
             jnz et           ; Reluare ciclu
             add dl,'$'
             mov [si],dl
             lea dx,s
             mov ah,9h
             int 21h
             mov ah,4ch
             int 21h
END start
```

Se propune analiza teoretică şi experimentală a acestui program.

5. Modul de lucru

- ►Se editează programul exemplu (în orice mediu de editare) şi se salvează cu extensia .ASM.
- ►Se asamblează şi link-editează programul, oţinându-se forma .EXE.
- ►Se testează programul, inclusiv prin încărcarea sa în depanator.
- ►Se cere elaborarea unui program care face conversia literelor mari dintr-un text în litere mici şi invers, memorând rezultatul într-o zonă de memorie. Textul iniţial va fi definit de programator.
- ► Se cere elaborarea unui program care realizează sortarea unui vector prin metoda bulelor. Vectorul este definit de programator în secţiunea de date.

LUCRAREA 14

MACROINSTRUCŢIUNI

1. Obiectivele lucrării

Această lucrare are ca scop însuşirea tehnicilor de utilizare a macroinstrucţiunilor în limbajul de asamblare asociat familiei de microprocesoare 80x86.

2. Aparatura şi suporturile utilizate

► PC în configuraţia unitate centrală, monitor, tastatură, mouse;
► Precizările din prezentul îndrumar.

3. Breviar teoretic

Macroinstrucţiunile permit programatorului să definească simbolic secvenţe de program (instrucţiuni, definiţii de date, directive etc.), asociate cu un nume. Prin folosirea macroinstrucţiunilor se simplifică scrierea codului sursă şi se măreşte lizibilitatea sa.

3.1. Macroinstrucţiuni. Definire şi expandare

Prin folosirea numelui macroinstrucţiunii în program, se va genera întreaga secvenţă de program. În esenţă, este vorba de un proces de substituţie (expandare) în textul programului sursă, care se petrece înainte de asamblarea programului. Un asamblor care dispune de suport pentru macroinstrucţiuni se numeşte macroasamblor.

Sunt două etape de lucru cu macroinstrucţiuni: definirea macroinstrucţiunilor şi utilizarea lor. Utilizarea se mai numeşte invocare sau chiar apel de macroinstrucţiune, dar ultima denumire poate produce confuzie, termenul utilizându-se în cadrul procedurilor.

Spre deosebire de proceduri, macroinstrucţiunile sunt expandate la fiecare utilizare, adică fiecare apariţie a numelui macroinstrucţiunii este înlocuită cu secvenţa corespunzătoare de cod. Avantajul este că textul sursă scris devine mai clar şi mai scurt, fără a se apela la utilizarea procedurilor.

Macroinstrucţiunile pot fi cu parametri sau fără parametri. Din punct de vedere sintactic, ele sunt asemănătoare cu directivele de tip DEFINE din limbajele de nivel înalt, dispunând, în general, de mecanisme mai evoluate decât acestea.

Definirea unei macroinstrucţiuni fără parametri are forma generală

```
nume_macro MACRO
;
; Corp macroinstructiune
;
ENDM
```

Invocarea constă în scrierea în textul sursă a numelui macroinstrucţiunii.

Exemple:

Macroinstrucţiunea **exit_to_dos** este definită prin:

```
exit_to_dos macro
 mov ah, 4Ch   ;nr. funcție în AH
 mov al, 0h    ;cod returnat sistemului de operare
 int 21h       ;întrerupere DOS
endm
```

O pereche de macroinstrucţiuni care salvează şi refac registrele generale folosind stiva este utilă pentru scrierea procedurilor sau a întreruperilor, pentru a păstra valorile registrelor dinainte de apelul procedurii sau a întreruperii:

```
save macro
 push ax
 push bx
 push cx
 push dx
 push si
 push di
endm
rest macro
 pop di
 pop si
 pop dx
 pop cx
 pop bx
 pop ax
endm
```

Din punct de vedere simbolic, setul de instrucţiuni al maşinii a fost extins cu două instrucţiuni asemenea cu *save* şi *rest*.

3.2. Macroinstrucţiuni cu parametri

Macroinstrucţiunile importante sunt cele cu parametri. Definiția unei macroinstrucţiuni cu parametri are forma generală

```
nume_macro MACRO P1, P2, …, Pn
;
; Corp macroinstrucțiune
;
ENDM
```

în care P1, P2, …, Pn sunt identificatori care specifică parametrii formali, adică cei folosiţi în definiția macroinstrucţiunii. Apelul (invocarea) unei macroinstrucţiuni cu parametri se face prin specificarea numelui, urmată de o listă de parametri actuali:

```
nume_macro X1, X2, …, Xn
```

La expandarea macroinstrucţiunii, se înlocuieşte numele ei cu corpul acesteia din definiţie şi în plus fiecare parametru formal se înlocuieşte cu parametrul actual respectiv.

Exemple:

Apelurile de funcţii DOS (întreruperea 21h) presupun plasarea numărului funcţiei în registrul AH. Se poate, deci, defini o macroinstrucţiune de forma:

```
dosint macro N
 mov ah, N
 int 21h
endm
```

şi invoca macroinstrucţiunea prin linii de forma:

```
dosint 9 ;funcţia 9 (afişare şir)
```

3.3. Controlul numărului de parametri actuali

O problemă importantă este controlul coincidenţei dintre numărul parametrilor formali şi cel al parametrilor actuali. Se vor utiliza următoarele directive specifice:

- ►Directiva *%OUT* (mesaj la consolă la momentul asamblării). Are forma generală **%OUT TEXT** efectul fiind afişarea textului la consolă la momentul asamblării; este utilizată pentru mesaje de eroare la asamblare;
- ►Directiva *.ERR* (forţează eroare de asamblare). Aceasta directivă forţează o eroare la asamblare, astfel încât să nu se mai genereze fişier obiect; este utilizată în cazul neconcordanţei dintre numărul parametrilor actuali şi cel al parametrilor formali.
- ►Directiva *EXITM* (ieşire forţată din expandare).

Pentru controlul efectiv al parametrilor actuali, se mai utilizează directivele de asamblare condiţionată *IFNB* (If Not Blank) şi *IFB* (If Blank) care testează existenţa sau non-existenţa unui parametru actual.

Se poate formula o regulă generală de scriere a unei macroinstrucţiuni cu controlul numărului de parametri. Dacă macroinstrucţiunea are N parametri utili, se declară un al (N+1) -lea parametru suplimentar şi se testează dacă al N-lea parametru este vid (cu directiva IFB); în caz afirmativ numărul parametrilor actuali este prea mic. Succesiv se testează dacă al (N+1) -lea parametru este nevid (cu directiva IFNB), cazul afirmativ semnificând faptul că sunt prea mulţi parametri actuali. O posibilă implementare este sugerată de secvenţa următoare:

```
test macro P1, ..., Pn, Pm
 ;testare număr de parametri
 IFB <Pn>  ;dacă nu există al n-lea parametru
 %OUT Prea putini parametri!
 .ERR
 ENDIF
 IFNB <Pm> ;dacă există un parametru în plus
 %OUT Prea multi parametri!
```

```
 .ERR
ENDIF
;corp macroinstrucţiune
endm
```

Exemplul de mai sus ajută la utilizarea (apelarea din program) corectă a unei macroinstrucţiuni cu n parametri. Orice încercare de a folosi mai mulţi sau mai puţini parametri decât numărul necesar întrerupe asamblarea.

3.4. Macroinstrucţiuni repetitive

Aceste macroinstrucţiuni au nume predefinite, iar utilizatorul defineşte corpul macroinstrucţiunii. Scopul lor este de a genera secvenţe repetate de program, prin expandare.

Macroinstrucţiunea *REPT* (Repeat - Repetă).

Forma generală de invocare este

```
rept N
;
; Corp macroinstructiune
;
endm
```

în care N este o constantă întreagă. Efectul este repetarea corpului macroinstrucţiunii de N ori.

Un exemplu de utilizare este dat de următoarea secvenţă care generează un şir de caractere cu litere de la 'A' Ia 'Z':

```
n=0
alfabet label byte
rept 26
db 'A'+n
n=n+1
endm
```

Macroinstrucţiunea *IRP* (Indefinite Repeat - Repetă nedefinit).

Forma generală de invocare este

```
irp p_formal, <lista_par_act>
;
; Corp macroinstructiune
;
endm
```

în care *p_formal* este un parametru formal, iar *lista_par_act* este o listă de parametri actuali, separaţi prin virgulă. Efectul este repetarea cosrpului macroinstrucţiunii de atâtea ori câte elemente conţine lista de parametri actuali. La fiecare repetare, se substituie parametrul formal cu câte un parametru actual.

4. Desfăşurarea lucrării

Se consideră următorul exemplu de utilizare a unei macroinstrucţiuni care realizează calculul factorialului unui număr n. Acest exemplu urmează a fi asamblat şi link-editat în cadrul şedinţei de laborator, în vederea studiului teoretic şi experimental.

```
.model small
.stack 100h
.data
.code

;Calculeaza N!, rez. in AX
fact macro N
 mov ah,0h  ;pune 1 in AX
 mov al,1h
 mov dl,1h  ;pune 1 in dl
 rept N     ;repeta de N ori
  mul dl    ;inmulteste AL cu DL
  inc dl    ;incrementeaza DL pt. urmatoarea iteratie
 endm
endm

Start:
 mov ax,@data ;segment date in DS
 mov ds,ax

 fact 5       ;5!=120

 mov ax,4C00h ;revenire in MS-DOS
 int 21h
end Start
```

5. Modul de lucru

- ►Se editează programul de mai sus (în orice mediu de editare, spre exemplu *Notepad*) şi se salvează cu extensia .ASM.
- ►Se asamblează şi link-editează programul, utilizând TASM şi TLINK cu opţiunile prezentate în lucrările anterioare, obţinându-se forma .EXE.
- ►Se testează programul, inclusiv prin încărcarea sa în depanator, urmărind expandarea macroinstrucţiunii în secvenţa echivalentă, inclusiv prin examinarea fişierului-listing generat la asamblare.
- ►Se cere elaborarea unui program care să calculeze S=1+2+…+n.

LUCRAREA 15

INSTRUCŢIUNI DE SALT ŞI APELURI DE PROCEDURĂ

1. Obiectivele lucrării

Lucrarea îşi propune atingerea ca obiective familiarizarea cu instrucţiunile de salt şi cu apelurile de procedură precum şi elaborarea unor programe în limbaj de asamblare utilizând aceste instrucţiuni.

2. Aparatura şi suporturile utilizate

► PC în configuraţia unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

Secvenţa de execuţie a instrucţiunilor într-un program este determinată de conţinutul registrului de segment de cod (CS) şi de pointerul de instrucţiuni (IP). Instrucţiunile de transfer al controlului programului operează asupra lui IP şi CS. La apariţia unui astfel de transfer, coada de instrucţiuni nu mai conţine instrucţiunea următoare şi unitatea de execuţie a procesorului va transmite noua adresă la unitatea de interfaţă cu magistrala, care va obţine instrucţiunea următoare direct din memorie, utilizând noile valori pentru CS : IP; instrucţiunea va fi transferată pentru execuţie, iar apoi începe reumplerea cozii de la noua locaţie.

Există patru tipuri de instrucţiuni de transfer al controlului:

► instrucţiuni de transfer necondiţionat;
► instrucţiuni de transfer condiţionat;
► instrucţiuni de ciclare;
► instrucţiuni de întrerupere.

3.1 Instrucţiuni de transfer necondiţionat al controlului

Transferul controlului se poate face la o locaţie (ţintă) în segmentul de cod curent (transfer intrasegment sau NEAR) sau la un segment de cod diferit (transfer intersegment sau FAR). Dacă modelul de organizare a memoriei utilizat într-o aplicaţie nu face segmentele vizibile programatorului, atunci nu se vor utiliza transferuri de control intersegment.

Instrucţiunea CALL (Apel de procedură)
Sintaxă:

```
CALL nume_procedura
CALL FAR  PTR nume_procedura
```

```
CALL NEAR PTR nume_procedura
```

Această instrucţiune activează o procedură, salvând adresa de revenire în stivă, pentru a permite unei instrucţiuni RET (return) din procedură să transfere controlul înapoi la instrucţiunea următoare lui CALL.

Şablonul dedefinire al unei proceduri este:

```
nume_proc PROC [ FAR | NEAR ]
              .
              .
     RET
nume_proc ENDP
```

unde nume_proc este numele procedurii, iar parametrii FAR şi NEAR (opţionali) indică tipul procedurii.

Asamblorul generează tipuri diferite de instrucţiuni maşină pentru CALL, după modul cum programatorul a definit numele procedurii, cu atributul NEAR sau FAR. Pentru returnarea corectă a controlului, tipul instrucţiunii CALL trebuie să fie potrivit cu tipul instrucţiunii RET. Diferitele forme ale instrucţiunii CALL permit ca adresa procedurii ţintă să fie obţinută direct din instrucţiune (apel direct), de la o locaţie de memorie sau dintr-un registru referit de instrucţiune (apel indirect). Trebuie reţinut faptul că procesorul ajustează automat pointerul IP pentru a referi instrucţiunea următoare, ce trebuie executată înainte de a-l salva în stivă.

Instrucţiunile de apel de procedură pot fi:

a) CALL direct intrasegment (NEAR) – eticheta este în acelaşi segment de cod cu instrucţiunea CALL

b) CALL indirect intersegment (FAR) – eticheta poate fi definită şi în alt segment de cod decât cel care conţine instrucţiunea CALL

Exemple:

```
.CODE
UNU:
    ...
DOI LABEL FAR
    ...
TREI PROC FAR
    ...
TREI ENDP
    ...
CALL UNU                    ;NEAR
CALL DOI                    ;FAR implicit
CALL FAR PTR TREI           ;FAR explicit
```

c) CALL indirect intrasegment

Exemple:

```
call cx;
call bx;
call word ptr nume_var
call word ptr var[bp][di]
call word ptr [bx][si]
call mem_word
```

d) CALL indirect intersegment – poate fi făcut numai prin memorie.

Exemple:

```
call dword ptr [bx]
call dword ptr nume_var[bp][si]
call mem_dublu_cuv
```

Apelurile şi salturile pot fi făcute numai la etichete relative la CS, nu la variabile. Atributul NEAR este implicit, în afara cazurilor când FAR este precizat în instrucţiune sau în declaraţia etichetei ţintă. Apelurile indirecte utilizând registre presupun apeluri NEAR. Registrul segment implicit este DS, la un apel indirect prin registru, în afara cazului când se utilizează BP sau se specifică un prefix:, ca în exemplul

```
call word ptr es:[bp][di]
```

Când se utilizează apeluri indirecte, trebuie ca tipul lui CALL să fie potrivit cu tipul instrucţiunii de revenire RET, altfel vor apărea erori (ce pot duce la pierderea controlului asupra calculatorului).

Pentru exemplificare, se consideră următorul program care evaluează expresia rez=n1+n2.

```
.MODEL small
.STACK  1000h
.DATA
        n1      dd 10000h
        n2      dd 20000h
        rez     dd ?
.CODE
Start:
        mov ax,@data                ;initializare registru de
                                    ;segment
        mov ds,ax
        jmp go                      ;sare la go
     pro_add proc near              ;procedura
        add ax,bx
        adc dx,cx
     ret                            ;return  fara descarcare
     pro_add endp                   ;incheiere procedura
```

```
go:
         mov ax, word ptr n1
         mov dx, word ptr n1+2
         mov bx, word ptr n2
         mov cx, word ptr n2+2
       call near ptr pro_add          ;apelul procedurii
         mov word ptr rez, ax
         mov word ptr rez+2,dx
         mov ah,4ch
         int 21h
END Start
```

Instrucţiunea RET (Revenire din procedură)

Sintaxă:

```
RETN [val_pop]
RETF [val_pop]
RET [val_pop]
```

În cazul RETN instrucţiunea reface registrul IP prin copierea vârfului stivei şi incrementarea registrului SP cu 2. Practic se copiază în IP adresa de revenire, ceea ce provoacă transferul controlului la instrucţiunea care urmează instrucţiunii CALL care a provocat apelul procedurii.

În cazul RETF instrucţiunea reface perechea CS:IP cu actualizarea registrului SP şi dacă este prezentă constanta val_pop se aduna val_pop la SP.

Asamblorul generează două tipuri de instrucţiuni maşină: RET intrasegment, dacă procedura a fost definită NEAR, şi RET intersegment, dacă procedura a fost definită de tip FAR.

Valoarea val_pop, care este opţională, este prevăzută pentru a descărca parametrii din stivă, depuşi în aceasta înainte de apelul procedurii. Această valoare reprezintă numărul de octeţi/cuvinte descărcate din stivă.

Instrucţiunea JMP (Instrucţiunea de salt)

Sintaxă:

```
JMP <ţinta>
```

Spre deosebire de CALL, instrucţiunea JMP nu salvează în stivă nicio informaţie, ci doar transferă controlul la o nouă adresă, specificată de instrucţiune. Instrucţiunea este asemănătoare cu CALL şi în ceea ce priveşte obţinerea adresei ţintă: direct din instrucţiune sau indirect prin registru sau memorie), potrivit următoarelor considerente:

a) JMP direct intrasegment modifică IP prin adunarea deplasamentului relativ al ţintei, conţinut în instrucţiune, la valoarea actualizată a lui IP. Dacă deplasamentul este în intervalul [–128 , +127] octeţi faţă de JMP, se va genera o formă de instrucţiune numai pe 2 octeţi, denumită short JMP; astfel se generează un near JMP cu un deplasament de 16 biţi (la 386 şi de 32 biţi). Aceste instrucţiuni sunt autorelative şi deci independente de poziţie (relocabile dinamic).

```
    jmp near_etich        ; salt direct intrasegment
    jmp short near_etich  ; salt de tip short
```

b) JMP direct intersegment va înlocui valorile lui CS:IP cu valorile conţinute în instrucţiune.

```
    jmp far_etich        ; salt în alt segment
    jmp far ptr etich   ;'etich' poate fi şi în acelaşi segment
```

c) JMP indirect intrasegment poate fi făcut prin intermediul unui registru general sau prin memorie, asemănător cu CALL:

```
    jmp ax
    jmp si
    jmp tabela[bx]
    jmp word  ptr [bp][di]
```

Exemplu de utilizare, pentru evaluarea expresiei

e = a+b daca i=1
a×b daca i=5
7 în rest

Valorile variabilelor a şi b aparţin intervalului (0,255].

```
.model small
.data
        a dw 3
        b dw 10
        i dw 5                    ;exemplu
        e dw ?
.stack 100h
.code
start:
        mov ax,@data
        mov ds,ax
        mov ax,i                ;pune  i in ax
        cmp ax,1                ;compară ax cu 1
        jne continui1           ;salt la continui1 daca
                                ;egalitatea ax<>1
        jmp calcul1             ;salt la calcul1
continui1:
        cmp ax,5
        jne continui7
        jmp calcul2
continui7:
        jmp calculREST
calcul1:
        mov ax,a
        add ax,b                ;aduna b la ax
        jmp final
calcul2:
        mov ax,a
```

```
        mul b
        jmp final
calculREST:
        mov ax,7
final:
        mov e,ax
        mov ax,4c00h
        int 21h
end start
```

Se observă implementarea structurii alternative multiple care testează pe i şi determină salturi către secvenţe unde se calculează expresii cu valori în AX. După aceea conţinutul lui AX este mutat în e.

Pentru a realiza salt la o etichetă sau un apel de procedură dintr-un alt modul, în modulul de definiţie eticheta trebuie să fie de tip PUBLIC, iar în modulul în care se utilizează, aceasta va fi definită prin EXTRN, conform schemei generale:

Modul1.asm	**Modul2.asm**
`PUBLIC etich1`	`EXTRN etich1:far`
`........`	`........`
`........`	`Jmp etich1`

3.2 Instrucţiuni de transfer condiţionat al controlului

Forma generală a unei astfel de instrucţiuni este

```
JCON <ţinta>
```

Instrucţiunile de transfer condiţionat pot transfera sau nu controlul, în funcţie de indicatorii procesorului, în momentul execuţiei instrucţiunii. Cele 18 tipuri de instrucţiuni de salt condiţionat testează o anumită configuraţie de indicatori. Când condiţia este îndeplinită (true) controlul este transferat la ţinta specificată de instrucţiune. Toate salturile condiţionale sunt de tip SHORT, adică ţinta trebuie să fie în segmentul de cod curent, în intervalul de [–128,+127] octeţi faţă de instrucţiunea de transfer (de exemplu JMP 00 realizează saltul la primul octet al următoarei instrucţiuni).

Deoarece adresa de salt este determinată prin adunarea deplasamentului relativ al ţintei la pointerul IP actualizat, toate salturile condiţionale sunt autorelative. Instrucţiunea de salt condiţional se utilizează după o instrucţiune aritmetică sau logică ce modifică indicatorii conform rezultatului. De cele mai multe ori, ele se utilizează după o instrucţiune de comparare de tip CMP d, s.

Tabelele 1, 2, 3, 4, şi 5 sintetizează utilizarea acestor instrucţiuni de salt condiţional.

Instrucţiunile listate ca perechi reprezintă aceleaşi instrucţiuni; asamblorul furnizează mnemonici alternative pentru o mai mare claritate în cadrul programului. Aceste mnemonici pot fi determinate relativ uşor deoarece ele utilizează anumite litere cu o semnificaţie predefinită astfel:

A – above – adică mai mare (deasupra)

B – below – adică mai mic (dedesubt)

G – greater – deci mai mare

L – less – deci mai mic

Primele două (A,B) se referă la relaţia dintre două valori fără semn, în timp ce ultimele două (G,L) se referă la relaţia dintre două valori cu semn.

Tabelul 1. Instrucţiunile de salt

Mnemonica	Condiţia de salt
JZ/ZE	ZF=1, dacă egalitatea (d)=(s)
JNZ/JNE	ZF=0, dacă egalitatea (d)<>(s)
JL/JNGE	SF<>OF, dacă (d)<(s), pentru numere cu semn
JLE/JNG	SF<>OF sau ZF=1, (d)<=(s), pentru numere cu semn
JNL/JGE	SF=OF, dacă (d)>=(s) pentru numere cu semn
JNLE/JG	SF=OF şi ZF=0, (d)>(s) pentru numere cu semn
JB/JNAE/JC	CF=1, dacă (d)< (s) pentru numere fără semn
JBE/JNA	CF=1 sau ZF=1, (d)<=(s) pentru numere fără semn
JNB/JAE/JNC	CF=0, dacă (d)>=(s), pentru numere fără semn
JNBE/JA	CF=0 şi ZF=0, (d)>(s), pentru numere fără semn
JP/JPE	PF=1, dacă numărul are paritate pară
JNP/JPO	PF=0, dacă numărul are paritate impară
JO	OF=1, dacă este depăşire de reprezentare
JNO	OF=0, dacă nu este depăşire de reprezentare
JS	SF=1, dacă numărul este negativ
JNS	SF=0, dacă numărul este pozitiv
J(E)CX	(E)CX =0 test asupra registrului ECX sau CX

Ultima instrucţiune J(E)CX (salt dacă (E)CX sau CX=0) se utilizează de obicei la începutul unei bucle software pentru a sări peste ciclul respectiv, în cazul în care (E)CX =0, altfel ciclul respectiv (cu instrucţiunea „loop") s-ar efectua de 64k ori.

Instrucţiunea de comparare a două valori poziţionează, bineînţeles, toţi indicatorii, dar, în funcţie de interpretarea dată de utilizator celor două valori, sunt semnificativi numai anumiţi indicatori, în funcţie de tipul numerelor:

Tabelul 2. Indicatorii pentru numere fără semn

Condiţie	ZF	CF
d>s	0	0
d=s	1	0
d<s	0	1

Tabelul 3. Indicatorii pentru numere cu semn

Condiţie	OF	SF	ZF
d>s	0/1	0/1	0
d=s	0	0	1
d<s	0/1	1/0	0

În compararea a două numere, în scopul de a realiza saltul pentru o anumită condiţie dată se vor folosi următoarele instrucţiuni:

Tabelul 4. Instrucţiuni de salt după comparare

Salt pentru	Numere fără semn	Numere cu semn
d=s	JE/JZ	JE/JZ
d<>s	JNE/JNZ	JNE/JNZ
d<s	JB/JNAE/JC	JL/JNGE
d>s	JA/JNBE	JG/JNLE
d<=s	JBE/JNA	JLE/JNG
d>=s	JAE/JNB/JNC	JGE/JNL

În scop ilustrativ, în cele ce urmează este prezentată o aplicaţie ce realizează însumarea elementelor unui vector

```
.model small
.data
    n equ 10
    x dw 1,2,3,4,5,6,7,8,9,10
    s dw ?
.stack 100h
.code
start:
    mov ax,@data
    mov ds,ax
    xor cx,cx
    xor ax,ax
    xor si,si
salt:
    add ax,[x+si]
    add si,2
    inc cx                  ;incrementeaza cx
    cmp cx,n                ;compara cx cu n
    jne continui            ;salt la continui daca cx<>n
    jmp final               ;salt la final
continui:
    jmp salt                ;salt la final
final:
    mov s,ax
    mov ax,4c00h
    int 21h
end start
```

4. Desfăşurarea lucrării

Efectuarea acestei lucrări constă în executarea programelor prezentate ca exemple ilustrative pentru utilizarea instrucţiunilor de salt şi apeluri de procedură.

5. Modul de lucru

- ►Se vor asambla şi link-edita programele prezentate ca exemplu, obţinându-se fişiere *.EXE*.
- ►Se vor obţine şi se vor analiza fişierele listing şi de referinţe încrucişate.
- ►Se vor rula sub *TD* programele prezentate.
- ►Se cere elaborarea unui program care, bazat pe o procedură cu numele *to_uppercase*, să modifice toate caracterele ASCII ale unui şir dat la majuscule.

LUCRAREA 16

INSTRUCŢIUNI ALE COPROCESOARELOR MATEMATICE

1. Obiectivele lucrării

Lucrarea urmăreşte familiarizarea studenţilor cu arhitectura coprocesoarelor matematice şi cu instrucţiunile specifice la nivel de limbaj de asamblare. Se va prezenta coprocesorul matematic din punct de vedere fizic şi funcţional şi se va urmări dezvoltarea unor aplicaţii cu instrucţiuni specifice coprocesoarelor matematice în limbajul de asamblare.

2. Aparatura şi suporturile utilizate

► PC în configuraţia unitate centrală, monitor, tastatură, mouse.
► Precizările din prezentul îndrumar.

3. Breviar teoretic

3.1. Introducere

Coprocesoarele matematice sunt circuite integrate dedicate (procesoare specializate), care extind setul de instrucţiuni ale procesoarelor centrale cu instrucţiuni specifice operaţiilor cu numere reale în virgulă mobilă.

Coprocesoarele sunt fie circuite de sine stătătoare (8087, 80287, 80387), fie sunt integrate în procesorul de bază (80486). În cel de-al doilea caz, nu se mai face distincţie între setul de instrucţiuni al procesorului de bază şi cel specific formatului în virgulă mobilă.

Tipurile de date recunoscute de coprocesoare sunt :

► Număr real în simplă precizie (dword);
► Număr real în dublă precizie (qword);
► Număr real în precizie extinsă (tbytes);
► Întreg pe 2 octeţi (word);
► Întreg pe 4 octeţi (dword);
► Întreg pe 8 biţi (qword);
► Întreg BCD pe 10 cifre (tbytes).

Aceste tipuri de date vor fi notate cu real32, real64, real80, int16, int32 şi bcd80. Întregii pe 8 octeţi pot fi numai încărcaţi în coprocesor. Intern, coprocesoarele lucrează exclusiv cu formatul real în precizie extinsă (10 octeţi). La încărcarea operanzilor din memorie, toate tipurile de date de mai sus sunt convertite la tipul intern. Similar, la depunerea operanzilor în memorie, au loc conversii de la formatul intern la unul din formatele de mai sus.

3.2. Arhitectura coprocesoarelor matematice

Arhitectura coprocesoarelor cuprinde 8 registre de câte 80 de biţi, numite ST(0), ST(1),…,ST(7). Aceste registre sunt organizate ca o stivă, registrul ST(0) (care se mai notează simplu ST) fiind vârful stivei.

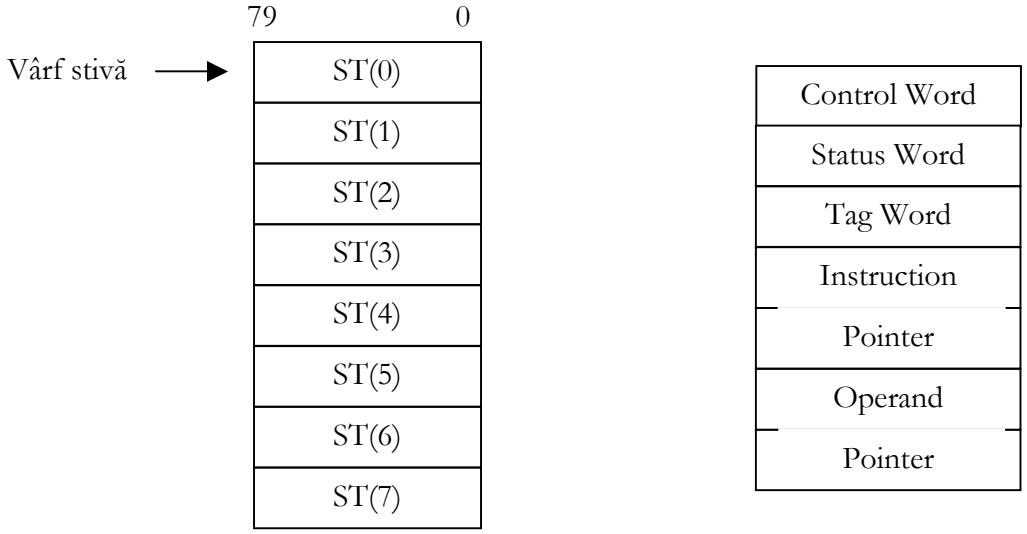


Fig. 1. Arhitectura Coprocesoarelor 80x87

Coprocesoarele dispun de o serie de registre suplimentare, dintre care cele mai importante sunt registrul de stare (Status Word) şi registrul de control (Control Word). Registrul de stare conţine indicatoare care se poziţionează în urma instrucţiunilor de comparaţie sau în caz de eroare. Registrul de control conţine câmpuri care controlează modul de execuţie al anumitor acţiuni (de exemplu cum se face rotunjirea la valori întregi).

Comunicaţia dintre procesorul de bază şi coprocesor se face exclusiv prin intermediul memoriei. Coprocesorul dispune de instrucţiuni de transfer între memorie şi cele 8 registre de lucru, precum şi pentru cuvintele de control şi de stare.

Vom prezenta coprocesorul 8087, deoarece setul de instrucţiuni este practic acelaşi la toate coprocesoarele. La procesoarele mai vechi decât 80486, se pune şi problema comunicaţiei dintre procesorul de bază şi coprocesor.

Coprocesorul 8087 monitorizează permanent fluxul de instrucţiuni şi sesizează prezenţa unei instrucţiuni specifice 8087 în memorie. În acest caz, el semnalează intenţia de a intra în execuţie prin semnalul electric TEST. Execuţia începe numai după ce 8086 intră în aşteptare, ca urmare a unei instrucţiuni WAIT. Coprocesorul recunoaşte starea de aşteptare şi începe execuţia instrucţiunii matematice, anulând în acelaşi timp cererea efectuată prin semnalul TEST. Ca urmare, 8086 iese din starea de aşteptare şi îşi continuă execuţia. Asambloarele inserează automat o instrucţiune WAIT înaintea fiecărei instrucţiuni 8087, deci nu este necesară codificarea lor explicită.

La coprocesoarele de generaţie mai nouă (80387), sincronizarea cu procesorul de bază se face prin semnale specializate, ceea ce elimină necesitatea instrucţiunilor WAIT.

În mod corespunzător, există instrucţiunea 8087 FWAIT, destinată sincronizării reciproce (8087 „aşteaptă” după 8086). Instrucţiunea FWAIT este necesară numai după operaţiile de depunere în memorie executate de 8087.

Înaintea începerii operării propriu-zise, este necesară o instrucțiune FINIT, care șterge registrele interne, condițiile de eroare, etc.

3.3. Instrucțiuni ale coprocesorului 8087

Instrucțiunile specifice 8087 se scriu în textul sursă la fel ca instrucțiunile procesorului de bază. Asamblorul recunoaște mnemonicele acestor instrucțiuni și generează cod mașină corespunzător. Acest cod va fi executat de către coprocesor, în urma generării unui semnal de către procesorul de bază 8086. Mnemonicele instrucțiunilor 8087 încep toate cu litera F. Majoritatea instrucțiunilor au ca operanzi vârful ST al stivei coprocesorului și un alt registru ST(i) sau un operand din memorie. Cele mai multe instrucțiuni matematice actualizează stiva, prin operațiile PUSH_ST și POP_ST (echivalentele operațiilor PUSH și POP, operându-se pe stiva coprocesorului).

Tabelele 1, 2, 3, 4 și 5 prezintă o sistematizare a acestor instrucțiuni.

Tabelul 1. Instrucțiuni pentru încărcarea datelor

Instrucțiune	Parametri	Descriere
FLD	ST(i)	Încarcă ST(i) în ST cu deplasarea stivei
FLD	Mem (real32/64/80)	Încarcă număr real din memorie în ST
FILD	Mem (int16/32)	Încarcă număr întreg din memorie în ST
FBLD	Mem (bcd80)	Încarcă număr BCD din memorie în ST
FLDZ		Încarcă constanta 0.0 în ST
FLD1		Încarcă constanta 1.0 în ST
FLDPI		Încarcă constanta pi în ST
FLDL2E		Încarcă constanta $\log_2(e)$ în ST
FLDL2T		Încarcă constanta $\log_2(10)$ în ST
FLDLG2		Încarcă constanta $\log_{10}(2)$ în ST

Tabelul 2. Instrucțiuni pentru depunerea datelor

Instrucțiune	Parametri	Descriere
FST	ST(i)	Depune ST în ST(i) fără descărcarea stivei
FSTP	ST(i)	Depune ST în ST(i) cu descărcarea stivei
FST	Mem (real32/64/80)	Depune ST real în memorie fără descărcarea stivei
FSTP	Mem (real32/64/80)	Depune ST real în memorie cu descărcarea stivei
FIST	Mem (int16/32)	Depune ST întreg în memorie fără descărcarea stivei
FISTP	Mem (int16/32)	Depune ST întreg în memorie cu descărcarea stivei
FBSTP	Mem (bcd80)	Depune ST BCD în memorie cu descărcarea stivei
FXCH		Schimbă ST cu ST(1)
FXCH	ST(i)	Schimbă ST cu ST(i)

Tabelul 3. Instrucţiuni de adunare, scădere, înmulţire şi împărţire

Instrucţiune	Parametri	Descriere
Adunare		
FADD		Adună ST cu ST(1)
FADD	ST(i),ST	Adună ST la ST(i)
FADD	Mem (real32/64/80)	Adună real din memorie la ST
FIADD	Mem (int16/32)	Adună întreg din memorie la ST
FADDP	ST(i)	Adună ST la ST(i) cu descărcarea stivei
Scădere		
FSUB		Calculează ST(1)-ST cu descărcarea stivei
FSUB	ST(i),ST	Scade ST din ST(i)
FSUB	Mem (real32/64/80)	Scade realul din memorie din ST
FISUB	Mem (int16/32)	Scade întregul din memorie din ST
FSUBP	ST(i)	Calculează ST(i)-ST cu descărcarea stivei
FSUBR		Calculează ST-ST(1) cu descărcarea stivei
FSUBR	ST(i),ST	Scade ST(i) din ST, rezultat în ST(i)
FSUBR	Mem (real32/64/80)	Scade ST din realul din memorie, rezultat în ST
FISUBR	Mem (int16/32)	Scade ST din întregul din memorie, rezultat in ST
FSUBPR	ST(i)	Scade ST(i) din ST cu descărcarea stivei
Înmulţire		
FMUL		Înmulţeşte ST cu ST(1) cu descărcarea stivei
FMUL	ST(i),ST	Înmulţeşte ST cu ST(i) rezultat în ST(i)
FMUL	Mem (real32/64/80)	Înmulţeşte realul din memorie cu ST
FIMUL	Mem (int16/32)	Înmulţeşte întregul din memorie cu ST
FMULP	ST(i)	Înmulţeşte ST cu ST(i) cu descărcarea stivei
Împărţire		
FDIV		Împarte ST(1) la ST cu descărcarea stivei
FDIV	ST(i),ST	Împarte ST(i) la ST rezultat în ST(i)
FDIV	Mem (real32/64/80)	Împarte ST la realul din memorie
FIDIV	Mem (int16/32)	Împarte ST la întregul din memorie
FDIVP	ST(i)	Împarte ST(i) la ST cu descărcarea stivei
FDIVR		Împarte ST la ST(1) cu descărcare
FDIVR	ST(i),ST	Împarte ST la ST(i) rezultat în ST(i)
FDIVR	Mem (real32/64/80)	Împarte realul din memorie la ST, rezultat în ST
FIDIVR	Mem (int16/32)	Împarte întregul din memorie la ST, rezultat în ST
FDIVPR	ST(i)	Împarte ST la ST(i) cu descărcare

Tabelul 4. Instrucţiuni de comparaţie

Instrucţiune	Parametri	Descriere
FCOM		Compară ST cu ST(1)
FCOM	ST(i)	Compară ST cu ST(i)
FCOM	Mem (real32/64/80)	Compară ST cu realul din memorie
FICOM	Mem (int16/32)	Compară ST cu întregul din memorie
FTST		Compară ST cu 0.0
FCOMP		Compară ST cu ST(1) cu descărcarea stivei
FCOMP	ST(i)	Compară ST cu ST(i) cu descărcarea stivei
FCOMP	Mem (real32/64/80)	Compară ST cu realul din memorie cu descărcarea stivei
FICOMP	Mem (int16/32)	Compară ST cu întregul din memorie cu descărcarea stivei
FCOMPP		Compară ST cu ST(1) cu descărcare dublă

Tabelul 5. Instrucţiuni diverse

Instrucţiune	Parametri	Descriere
Instrucţiuni diverse		
FABS		ST ia valoarea absolută
FCHS		ST ia valoarea -ST
FSQRT		ST ia valoarea sqrt(ST)
Instrucţiuni de control		
FINIT		Iniţializează coprocesorul
FWAIT		Sincronizează cu procesorul de bază
FSTSW	Mem (int16)	Depune cuvântul de stare în memorie
FSTCW	Mem (int16)	Depune cuvânt de control în memorie
FLDCW	Mem (int16)	Încarcă cuvânt de control din memorie
FNOP		Nici o operaţie

În afară de instrucţiunile prezentate, coprocesoarele dispun de instrucţiuni transcendente pentru calculul funcţiilor trigonometrice directe şi inverse, exponenţială, logaritmică, etc. Aceste instrucţiuni nu vor fi prezentate deoarece este puţin probabil să se dezvolte aplicaţii cu funcţii transcendente în limbaj de asamblare.

4. Desfăşurarea lucrării

În scopul analizei teoretice şi experimentale, este propus un program în limbajul de asamblare pentru calculul volumului unui trunchi de con, folosindu-se instrucţiunile prezentate în breviarul teoretic. Se dau raza bazei mari (r1), raza bazei mici (r2) şi înălţimea trunchiului de con h.

```
TITLE trunchidecon
.MODEL small
.STACK 100h
.8087              ; directiva specifica coprocesorului 8087
.DATA
 cst   dd    3.0         ; constanta 3
 r1    dd    12.0        ; raza mare a trunchiului de con
```

```
  r2    dd    4.0          ; raza mica a trunchiului de con
  h     dd    9.0          ; inaltimea trunchiului de con
  tmp1  dd    ?            ; variabila temporara de lucru
  tmp2  dd    ?            ; variabila temporara de lucru
.code
Start:

     mov ax,@data
     mov ds,ax

     finit                ; initializare coprocesor matematic
     fld cst              ; incarcam st cu 3
     fldpi                ; incarcam st cu pi
     fld h                ; incarcam st cu inaltimea trunchiului
     fld r1               ; incarcam st cu r1
     fld r2               ; incarcam st cu r2

;calculam r1 la patrat
     fld r1               ; incarcam st cu r1
     fmul ST,ST(2)        ; inmultim st cu st(2), adica r1*r1
     fstp tmp1            ; extragem rezultatul in tmp1
                          ; cu descarcarea stivei

;calculam r2 la patrat
     fld r2               ; incarcam st cu r2
     fmul ST,ST(1)        ; inmultim st cu st(1), adica r2*r2
     fstp tmp2            ; extragem rezultatul in tmp2
                          ; cu descarcarea stivei

;calculam r1*r2
     fmulp ST(1),ST       ; inmultim st(1)=r1 cu st=r2,
                          ; rezultat in st(1) cu descarcarea
                          ; stivei
     fadd tmp1            ; adunam la st pe tmp1, st va deveni
                          ; r1*r2+r1*r1
     fadd tmp2            ; adunam la st pe tmp2,
                          ; st va deveni r1*r2+r1*r1+r2*r2
     fmul                 ; inmultim st cu st(1),
                          ; rezultat in st(1) cu descarcarea
                          ; stivei
                          ; in st va fi valoarea
                          ; (r1*r2+r1*r1+r2*r2)*h
     fmul                 ; inmultim st cu st(1),
                          ; rezultat in st(1) cu descarcarea
                          ; stivei, in st va fi valoarea
                          ; (r1*r2+r1*r1+r2*r2)*h*pi
     fdivr                ; impartim st cu st(1),
                          ; rezultat in st(1) cu descarcarea
                          ; stivei
;----- rezultat final ----
     mov ah,4ch
     int 21h
END Start
```

Se foloseşte directiva .8087 care instruieşte asamblorul să accepte mnemonicele specifice coprocesorului 8087. În funcţie de tipul coprocesorului şi al procesorului de bază se pot utiliza directivele .287, .387, .486.

În exemplul prezentat, volumul trunchiului de con se va afla în vârful stivei ST a coprocesorului matematic.

5. Modul de lucru

► Se va edita fişierul *tr.asm*.
► Se va asambla fişierul *tr.asm* folosind *tasm tr.asm*.
► Se va genera *tr.exe* folosind *tlink tr.obj*.
► Se va verifica funcţionarea programului cu ajutorul TD, alegând din meniul "View" opţiunea "Numeric Processor". Se va rula folosind tasta F8 (Step) pentru a urmări toţi paşii execuţiei programului.
► Se cere crearea unui program în care să se calculeze media aritmetică, media geometrică şi media armonică a două numere.

BIBLIOGRAFIE

1. Athanasiu, A., Pănoiu, A. *Microprocesoarele 8086, 286, 386. Programarea în limbaj de asamblare*. Editura Teora, Bucureşti, 1992.

2. Hyde, R. *The Art of Assembly Language*. No Starch Press, San Francisco, 2003.

3. Lungu, V. *Procesoare Intel. Programare în limbaj de asamblare*. Editura Teora, Bucureşti, 2000.

4. Muscă, G. *Programare în limbaj de asamblare*. Editura Teora, Bucureşti, 1996.

5. Rădulescu, G. *Elemente de arhitectură a sistemelor de calcul. Programare în limbaj de asamblare*. Editura MATRIX ROM, Bucureşti, 2007.

6. Ţăpuş, N., Z. Racoviţă, Z. *Programarea în limbaj de asamblare*. Universitatea „Politehnica" Bucureşti, 1994.

7. *** *8086 16-BIT HMOS MICROPROCESSOR 8086/8086-2/8086-1. Data Sheet*. Intel Corporation, 1990. (**http://www.datasheet4u.com/html/8/0/8/8086_Intel.pdf.html**)

8. *** *80C186XL/80C188XL Microprocessor User's Manual*. Intel Corporation, 1995. (**http://www.intel.com/design/intarch/manuals/27216403.pdf**)

9. *** *Intel® Architecture Software Developer's Manual*. Intel Corporation, 1999. (**http://www.intel.com/design/pentiumii/manuals/24319002.pdf**)

10. *** *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2006. (**http://www.intel.com/design/processor/manuals/253665.pdf**)

ANEXA 1. TABELA CODURILOR ASCII

– setul de bază –

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

În mod curent, pentru codurile ASCII de la **0** la **31** (zecimal) se folosesc următoarele abrevieri standard:

00-NUL 01-SOH 02-STX 03-ETX 04-EOT 05-ENQ 06-ACK 07-BEL
08-BS 09-HT 10-LF 11-VT 12-FF 13-CR 14-SO 15-SI
16-DLE 17-DC1 18-DC2 19-DC3 20-DC4 21-NAK 22-SYN 23-ETB
24-CAN 25-EM 26-SUB 27-ESC 28-FS 29-GS 30-RS 31-US

ANEXA 2. TABELA CODURILOR ASCII

– setul extins –

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	∑
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	