Universitatea ”Lucian Blaga” din Sibiu
Catedra de Calculatoare și Automatizări

Arpad GELLERT

Rodica BACIU

Programare în Limbaj de Asamblare

Aplicaţii

Programare în limbaj de asamblare
Lucrarea nr. 1.

1. Tipuri de date

BYTE (1 octet)

Acest tip de date ocupă 1 octet şi poate fi reprezentat atât în memoria internă, cât şi într-un registru de 8 biţi al procesorului. Interpretările tipului byte pot fi:

- întreg pe 8 biţi cu sau fără semn;
- caracter ASCII.

Directiva pentru definirea datelor de acest tip este DB (Define Byte).

WORD (2 octeţi)

Acest tip de date ocupă 2 octeţi şi poate fi reprezentat atât în memoria internă, cât şi într-un registru de 16 biţi al procesorului. Interpretările tipului word pot fi:

- întreg pe 16 biţi cu sau fără semn;
- secvenţă de două caractere ASCII;
- adresă de memorie de 16 biţi.

Directiva pentru definirea datelor de acest tip este DW (Define Word). Partea mai puţin semnificativă este memorată la adrese mici. De exemplu dacă presupunem întregul 1234H la adresa 1000H, atunci octetul 34H se va găsi la adresa 1000H, iar octetul 12H la adresa 1001H. Similar, se memorează şi secvenţele de două caractere ASCII.

DOUBLE-WORD (4 octeţi)

Acest tip de date ocupă 4 octeţi şi poate fi reprezentat atât în memoria internă, cât şi într-o pereche de registre de 16 biţi sau într-un registru de 32 de biţi (la procesoarele de 32 de biţi). Interpretările tipului dword pot fi:

- întreg pe 32 de biţi cu sau fără semn;
- număr real în simplă precizie;
- adresă de memorie de 32 de biţi.

Directiva pentru definirea datelor de acest tip este DD (Define Double-Word). Valorile mai puţin semnificative se memorează la adrese mici. În cazul adreselor pe 32 de biţi, adresa de segment este memorată la adrese mari, iar deplasamentul (offset-ul), la adrese mici.

QUAD-WORD (8 octeţi)

Acest tip de date ocupă 8 octeţi şi poate fi reprezentat atât în memoria internă cât şi într-o pereche de registre de 32 de biţi (numai la procesoarele de 32 de biţi). Interpretările tipului qword pot fi:

- întreg pe 64 de biţi cu sau fără semn;
- număr real în dublă precizie;

Directiva pentru definirea datelor de acest tip este DQ (Define Quad-Word).

TEN-BYTES (10 octeţi)

Acest tip de date ocupă 10 octeţi şi poate fi reprezentat atât în memoria internă, cât şi într-unul din registrele coprocesoarelor matematice. Interpretările tipului tbyte pot fi:

- număr întreg reprezentat ca secvenţă de cifre BCD (împachetate), cu semn memorat explicit;
- număr real în precizie extinsă.

Directiva pentru definirea datelor de acest tip este DT (Define Ten-Bytes).

2. Registrele procesorului 8086

Toate registrele procesorului 8086 sunt de 16 biţi. O serie de registre (AX, BX, CX, DX) sunt disponibile şi la nivel de octet, părţile mai semnificative fiind AH, BH, CH şi DH, iar cele mai puţin semnificative, AL, BL, CL şi DL. Denumirile registrelor sunt: AX-registru acumulator, BX-registru de bază general, CX-registru contor, DX-registru de date, BP-registru de bază pentru stivă (base pointer), SP-registru indicator de stivă (stack pointer), SI-registru index sursă, DI-registru index destinaţie. Registrul FLAGS cuprinde flagurile procesorului şi ale bistabililor de condiţie, iar registrul IP (instruction pointer) este contorul program.

Denumirile registrelor de segment sunt: CS-registru de segment de cod (code segment), DS-registru de segment de date (data segment), SS-registru de segment de stivă (stack segment), ES-registru de segment de date suplimentar (extra segment). Perechea de registre (CS:IP) va indica adresa următoarei instrucţiuni, iar perechea (SS:SP) indică adresa vârfului stivei. Registrele DS şi ES sunt folosite pentru a accesa date.

3. Moduri de adresare

Modurile de adresare specifică modul în care se calculează adresa fizică a operandului aflat în memorie. Adresa fizică (AF) se calculează utilizând:

- adresa de segment (AS) adică adresa de început a segmentului în care se află operandul;
- adresa efectivă (AE) adica deplasamentul sau offset-ul operandului în cadrul segmentului.

Adresarea imediată

Operandul apare explicit în instrucţiune:

```
Ex.:    mov     ax, 1           ; pune în AX valoarea 1
        add     bx, 2           ; adună la BX valoarea 2
```

Adresare directă

Adresa efectivă a operandului este furnizată printr-un deplasament. Se consideră registru de segment implicit registrul DS, în cazul în care nu se face o explicitare a registrului de segment.

```
Ex.:    .data
                VAL     dw      1
        .code
                ……………..
                mov     bx, VAL         ; pune în registrul bx valoarea 1
                add     cx, [100]       ; adună la registrul cx ceea ce se află în memorie,
                                        ; în segmentul de date la offset-ul 100.
```

La asamblare, VAL se va înlocui cu deplasamentul în cadrul segmentului de date. Valoarea 100 este offset explicit.

Adresare indirectă (prin registre)

Adresa efectivă a operatorului este dată de conţinutul registrelor BX, SI sau DI. Registrul de segment implicit este DS.

```
Ex.:    mov     ax, [bx]             ; pune în AX conţinutul locaţiei de memorie de la adresa dată de BX
        mov     [di], cx             ; memorează conţinutul lui CX la adresa dată de registrul DI
        add     byte ptr [si], 2     ; adună valoarea 2 la octetul aflat la adresa dată de SI
```

La ultima instrucţiune, operatorul *byte ptr* este absolut necesar altfel nu s-ar cunoaste tipul operandului (octet, cuvânt) la care se adună 2.

Adresare bazată sau indexată

Adresa efectivă a operandului din memorie se obţine adunând la unul din registrele de bază (BP sau BX) sau la unul din regiştrii index (SI sau DI) un deplasament constant de 8 sau 16 biţi. Registrul de segment implicit este DS (pentru BX, SI) şi SS (pentru BP).

```
Ex.:    .data
                VAL dw 10 dup (0)
        .code
                …………….
                mov     bx, 5
                mov     ax, VAL[bx]
                mov     ax, bx[VAL]
                mov     ax, [bx+VAL]
                mov     ax, [bx].VAL
```

Instrucţiunile anterioare pun în registrul AX cuvântul din memorie aflat la offset-ul VAL+5.

Observaţii:

```
        mov     ax, [bx]             ; adresare indirectă
        mov     ax, [bp]             ; adresare bazată cu deplasament nul,
                                     ; deoarece registrul BP nu se foloseşte în adresarea indirectă
```

Adresare bazată şi indexată

Adresa efectivă este formată prin adresarea unuia din registrele de bază (BX sau BP) cu unul din registrele index (SI sau DI) şi cu un deplasament de 8 sau 16 biţi. Registrele de segment implicite sunt DS pentru BX cu SI sau DI şi SS pentru BP cu SI sau DI.

```
Ex.:    mov     ax, [bx][si]
        mov     ax, [bx+si+7]
        mov     ax, [bx+si].7
        mov     ax, [bp][di][7]
```

Observaţii: la toate modurile de adresare se poate utiliza un registru de segment explicit, în felul următor:

```
        mov     bx, ds:[bp+7]        ; adresare bazată
        mov     ax, cs:[si][bx+3]    ; adresare bazată şi indexată
        mov     ax, ss:[bx]          ; adresare indirectă.
```

4. Aplicaţii

Se cere obţinerea fişierului executabil pentru următoarea porţiune de cod şi rularea apoi pas cu pas.

```
.model small
.stack 100
.data
        tabel   db 1, 2, 3, 4, 5, 10 dup(?)
        tabel1  dw 1, 2, 3, 12H, 12
        tabel2  dd 1, 2, 1234H
        tabel3  dq 1, 2, 12345678H
        tabel4  dt 1, 2,, 1234567890H
.code
```

```
start:  mov  ax, @data
        mov  ds, ax
        mov  ax, 14h                      ;adresare imediată
        mov  ax, 14
        mov  al, tabel                    ;adresare directă
        mov  al, tabel[1]                 ;adresare directă
        mov  ax, word ptr tabel           ;adresare directă - operatorul ptr este necesar,
        mov  ax, word ptr tabel[2]        ;tabel fiind definit cu directiva DB
        mov  bx, offset tabel
        mov  al, [bx+1]                   ;adresare indirectă
        mov  al, [bx]                     ;adresare indirectă
        mov  bx, 5
        mov  al, tabel[bx]                ;adresare bazată
        mov  si, 1
        mov  al, [bx][si]                 ;adresare bazată şi indexată
        mov  si, 6
        mov  byte ptr [bx][si],2          ;adresare bazată şi indexată
        mov  bp, offset tabel
        mov  al, [bp]                     ;adresare bazată cu deplasament nul
        mov  byte ptr ds:[bp][si][1], 7
        mov  word ptr ds:[bp][si][1], 19H ;adresare bazată şi indexată
        mov  ah, 4ch
        int  21h
end start
```

Indicaţii

Pentru obţinerea fişierului executabil se va utiliza mediul Borland C sau Borland Pascal de dezvoltare a programelor. Fazele de obţinere a programului executabil din fişierul ASCII salvat cu extensia *.asm sunt:

- asamblarea: tasm *.asm/zi/la
  - *zi* este folosită pentru includerea opţiunilor de depanare;
  - *la* este folosită pentru obţinerea unui fişier listing util în depanare;
- linkeditarea: tlink *.obj/v
  - *v* este folosită pentru includerea opţiunilor de depanare;
- depanarea: td *.exe

După obţinerea fişierului executabil acesta va fi executat pas cu pas în mediul de depanare. Se va vizualiza conţinutul memoriei (View/Dump) şi al registrelor (View/Registers) înainte şi după executarea fiecărei instrucţiuni. Se va verifica pentru fiecare dată definită numărul octeţilor alocaţi în memorie şi respectarea convenţiei Intel pentru memorarea acestora.

Probleme propuse spre rezolvare

1. În aplicaţia de mai sus, care sunt instrucţiunile care modifică conţinutul unei locaţii de memorie?
2. În aplicaţia de mai sus să se determine care este adresa la care se încarcă segmentul de date. Pentru aceasta se vor executa primele două instrucţiuni ale programului.
3. Pentru aplicaţia de mai sus să se specifice pentru fiecare zonă a programului (date, cod, stivă) care este adresa de segment şi care este adresa de offset la începutul execuţiei programului.
4. Pentru aplicaţia de mai sus să se determine care este adresa fizică a locaţiei de memorie la care se memorează valoarea 5 din tabel. Fiecare student va determina această adresă pentru contextul de pe calculatorul pe care lucrează.
5. Pentru aplicaţia de mai sus să se determine câţi octeţi s-au alocat pentru memorarea datelor. Să se verifice şi numărând octeţii în fereastra Dump.
6. Specificaţi adresa efectivă şi adresa fizică a locaţiei de memorie modificată de instrucţiunile:
   - mov byte ptr [bx][si], 2
   - mov byte ptr ds:[bp][si][1], 7
   - mov word ptr ds:[bp][si][1], 19H
7. Motivaţi apariţia valorii 00H (în loc de 01H) în registrul AL, în urma execuţiei instrucţiunii:
   mov al, [bp].

Programare în limbaj de asamblare
Lucrarea nr. 2.

1. Definirea şi iniţializarea datelor

Constante

O constantă este un simplu nume asociat unui număr şi nu are caracteristici distinctive.

Tip	Reguli	Exemple
binare	secvenţe de 0, 1 urmate de B sau b	11B, 1011b
octale	secvenţe de 0÷7 urmate de O sau Q	777Q, 567O
zecimale	secvenţe de 0÷9 opţional urmate de D sau d	3309, 1309D
hexazecimale	secvenţe de 0÷9 şi A÷F urmate de H sau h	55H, 0FEh
ASCII	şir de caractere	'BC'

Constantele pot apare explicit în program:

```
mov     ah, 5
mov     ax, 256
```

sau ca valori asignate unor simboluri (constante simbolice):

```
five    equ 5
mov     ah, five
```

Variabile

Variabilele identifică datele manipulate, formând operanzi pentru instrucţiuni. Se definesc utilizând directivele DB, DW, DD, DQ, DT. Aceste directive alocă şi iniţializează memoria în unităţi de octeţi, cuvinte, dublu cuvinte, etc.

nume directivă lista_de_valori

Numelui variabilei i se asociază următoarele atribute:
- segment: variabilă asociată cu segmentul curent
- offset: variabilă asignată offset-ului curent faţă de începutul segmentului
- tip: 1 octet pentru DB, 2 octeţi pentru DW, etc.

Lista de valori poate cuprinde:
- expresie constantă
- caracterul ? pentru iniţializări nedeterminate
- expresie de tip adresă
- un şir ASCII cu mai mult de două caractere (numai cu directiva DB)
- expresie DUP (expresie1 [, expresie2, …])

Etichete

Etichetele identifică cod executabil, formând operanzi pentru CALL, JMP sau salturi condiţionate. O etichetă poate fi definită:
- prin numele etichetei urmat de caracterul : - se defineşte astfel o etichetă de tip *near*
  nume_etichetă:
  Ex.: etichetea: mov ax, bx
- prin directiva LABEL, cu forma generală
  nume label tip
  Dacă ceea ce urmează reprezintă instrucţiuni (cod), tipul etichetei va fi NEAR sau FAR şi eticheta va fi folosită ca punct ţintă în instrucţiuni de tip JMP/CALL. Daca ceea ce urmează reprezintă definiţii de date, tipul etichetei va fi BYTE, WORD, DWORD, etc.

De exemplu, în urma definiţiei:

```
alfab  label  byte
alfaw  dw  1234H
```

o instrucţiune de forma *mov al, alfab* va pune în AL octetul mai puţin semnificativ al cuvântului (34H).

- prin directiva PROC, numele procedurii fiind interpretat ca o etichetă cu tipul derivat din tipul procedurii (NEAR sau FAR). Forma generală a unei proceduri este:
  nume_proc proc tip
  …
  nume_proc endp

2. Instrucţiuni de transfer

- instrucţiuni de transfer generale: MOV, XCHG, PUSH, POP
- instrucţiuni de transfer specifice acumulatorului: IN, OUT, XLAT
- instrucţiuni de transfer specifice adreselor: LEA, LDS, LES
- instrucţiuni de transfer specifice indicatorilor de condiţii: LAHF, SAHF, PUSHF, POPF

2.1. Instrucţiuni de transfer generale

Instrucţiunea MOV (Move Data - Transferă date)

Forma generală este:

```
mov  dest, sursa          ; dest ← src
```

Următoarele operaţii sunt ilegale:

- sursa şi destinaţia nu pot fi ambele operanzi în memorie;
- nu pot fi folosite registrele FLAGS şi IP;
- operanzii nu pot avea dimensiuni diferite;
- registrul CS nu poate apărea ca destinaţie;
- nu pot fi transferate date imediate într-un registru de segment;
- operanzii nu pot fi simultan registre de segment

Exemple:

```
mov     ax, bx
mov     al, ch
mov     val[bx][si], al
mov     byte ptr [bx+100], 5
```

O instrucţiune de forma:

```
.data
        ALFA  db  1
.code
        mov  al, ALFA
```

încarcă în AL conţinutul locaţiei de memorie ALFA. Dacă se doreşte încărcarea adresei efective a variabilei ALFA, se poate folosi operatorul OFFSET:

```
mov     bx, offset ALFA
```

Instrucţiunea XCHG (Exchange Data - Interschimbă date)

Forma generală este:

```
XCHG dest, sursa          ; sursa ↔ dest
```

Restricţii:

- cel puţin un operand trebuie să fie în registru;
- registrele de segment nu pot fi operanzi.

Exemple:

```
xchg    ax, bx
```

Secvenţă de instrucţiuni pentru schimbarea conţinutului a două locaţii de memorie:

```
mov     ax, alfa1
xchg    ax, alfa2
mov     alfa1,ax
```

Instrucţiunea PUSH (Push Data - Salvează date în stivă)

Forma generală:

push sursa

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

Sursa pote fi un operand pe 16 biţi aflat în:

- registru general de 16 biţi
- registru de segment
- locaţie de memorie

Exemple:

```
push    bx          ; SS:[SP] ← BX
push    beta        ; conţinutul memoriei de la adresa beta se memorează în stivă
push    [bx]        ; SS:[SP] ← DS:[BX]
push    [bp+5]      ; SS:[SP] ← SS:[BP+5]
```

Instrucţiunea POP (Pop Data - Refă date din stivă)

Forma generală:

pop dest

high(dest) ← SS: [SP+1]
low(sursa) ← SS:[SP]
SP ← SP+2

Destinaţia pote fi un operand pe 16 biţi aflat în:

- registru general de 16 biţi
- registru de segment
- locaţie de memorie

Exemple:

```
pop     cx          ; CX ← SS:[SP]
pop     es:[di]     ; ES:DI ← SS:[SP]
pop     [bp+5]      ; SS:[BP+5] ← SS:[SP]
```

Accesul la informaţiile memorate în stivă se poate face fără descărcarea acesteia, utilizăd adresarea bazată:

```
mov     bp, sp      ; baza stivei
push    ax          ; SP ← BP-2
push    bx          ; SP ← BP-4
mov     ax, [bp-2]
mov     bx, [bp-4]
```

2.2. Instrucţiuni de transfer specifice acumulatorului

Instrucţiunea IN (Input Data - Citeşte date de la port de intrare)

Forma generală este:

```
in  dest, port
```

Instrucţiunea transferă un octet sau un cuvânt de la un port de intrare în acumulator (AL sau AX). Port poate fi registrul DX, sau o constantă în intervalul 0÷255.

Instrucţiunea OUT (Output Data - Scrie date la port de ieşire)

Forma generală este:

```
out  port, sursa
```

Instrucţiunea transferă un octet sau un cuvânt din registrul AL sau AX la un port de ieşire.

Exemple:

```
in      al, 0f8h        ; citire stare de la un echipament
in      al, 0f9h        ; citire date de la acelaşi echipament
```

Instrucţiunea XLAT (Translate - Translatează)

Instrucţiunea nu are operanzi, iar semnificaţia este:

AL ← DS:[BX+AL]

Adică aduce în AL conţinutul octetului de la adresa efectivă [BX+AL].

2.3. Instrucţiuni de transfer specifice adreselor

Instrucţiunea LEA (Load Effective Address - Încarcă adresa efectivă)

Forma generală este:

```
LEA  registru, sursa
```

Încarcă adresa efectivă a operandului sursă în registrul specificat. Sursa este un operand aflat în memorie.

Exemple:

```
lea     bx, alfa            ; echivalentă cu instrucţiunea  mov bx, offset alfa
lea     di, alfa[bx][si]
```

Instrucţiunile LDS (Load Data Segment - Încarcă DS) şi LES (Load Extra Segment - Încarcă ES)

Forma generală este:

```
lds     reg, sursa
les     reg, sursa
```

în care *reg* este un registru general de 16 biţi, iar sursa este un operand de tip double-word aflat în memorie, care conţine o adresă completă de 32 de biţi. Această adresă se încarcă în perechea *DS:reg*, respectiv *ES:reg*, astfel încât cuvântul mai puţin semnificativ (adresa efectivă) va fi în registrul *reg*, iar cuvântul mai semnificativ (adresa de segment) va fi în *DS(ES)*.

Exemple:

```
.data
        x  db  10
        y  db  15
        adr_x  dd  x              ; adr_x este adresa valorii x (adresăde tip pointer)
        adr_y  dd  y              ; adr_y este adresa valorii y (adresăde tip pointer)
.code
        …………...
        lds  si,  adr_x
        les  di,  adr_y
        mov  byte  ptr  [si],  20      ; valoarea 20 se memorează în locul valorii 10
        mov  byte  ptr  es:[di],  30  ; valoarea 30 se memorează în locul valorii 15
```

2.4. Instrucţiuni de transfer specifice indicatorilor de condiţie

Instrucţiunea LAHF (Load AH with FLAGS - Încarcă AH cu FLAGS)

$AH \leftarrow FLAGS_{0:7}$

Instrucţiunea SAHF (Store AH into FLAGS - Depune AH în FLAGS)

$FLAGS_{0:7} \leftarrow AH$

Instrucţiunea PUSHF (Push Flags - Salvează FLAGS în stivă)

SP ← SP-2
SS: [SP+1] ← high(FLAGS)
SS:[SP] ← low(FLAGS)

Instrucţiunea POPF (Pop Flags - Reface FLAGS din stivă)

high(FLAGS) ← SS: [SP+1]
low(FLAGS) ← SS:[SP]
SP ← SP+2

3. Aplicaţii

1. Se cere rularea pas cu pas a următorului program:

```
.model small
.stack 100
.data
          tabc db   '0123456789ABCDEF'
.code
    start:  mov     ax,@data
            mov     ds,ax
            mov     al,0                    ;iniţializează AL
    repet:  mov     bx,offset tabc          ;pune în BX adresa efectivă tabc
            push    ax                      ;salvează AL în stivă
            xlat                            ;pune în AL octetul de la adresa efectivă [BX+AL]
            call    afisare                 ;apelul procedurii afisare
            pop     ax
            inc     al
            cmp     al,10H                  ;se verifică dacă s-au afişat toate caracterele
            jz      sfarsit
            jmp     repet
    afisare proc
            mov     dl,al                   ;pune în DL codul caracterului care trebuie afişat
            mov    ah,2h                    ;funcţia DOS pentru afişarea caracterului din DL
            int     21h
            mov     dl,' '
            mov    ah,2h
            int     21h
```

```
        ret
afisare endp
sfarsit: mov    ax,4c00h        ;funcţia DOS de ieşire în sistemul de operare
        int     21h
        end     start
```

2. De ce este necesară salvarea în stivă a registrului AX, iar apoi restaurarea ei?

3. Modificaţi programul de mai sus pentru determinarea pătratului unui număr din intervalul 0-15.

4. Se cere rularea pas cu pas a următorului program:

```
.model small
.stack 100
.data
        adr1    dw 1234h
        adr2    dw 5678h
        adr3    dw 9012h
        adr4    dw 3456h
        tabela  dd adr1, adr2, adr3, adr4
.code
start:  mov     ax, @data
        mov     ds, ax
        mov     cx, 1
        mov     bx, cx
        add     bx, bx
        add     bx, bx
        les     di, tabela[bx]
        mov     ax, es:[di]
        mov     ah, 4ch
        int     21h
        end     start
```

5. Modificaţi programul de mai sus astfel încât să fie pusă în AX valoarea care se află la *adr4*.

6. Care din instrucţiunile următoare sunt corecte?

```
mov     al, si
mov     ds, es
mov     ax, ip
mov     bl, al
mov     cs, 23H
```

7. Care este efectul următoarei secvenţe de instrucţiuni?

```
.data
        alfa    db 14H
        tabel   db  0A1H, 10, 0AFH, 23Q, 1111B, 0011B
.code
        …………………
        lea     bx, tabel
        mov     si, 3
        mov     al, [bx][si]
```

8. Care este conţinutul locaţiei *alfa* după executarea urmoarei secvenţe de instrucţiuni?

```
.data
        alfa    db  23h
        beta    db  43h
.code
        …………………
        mov     al,  alfa
        xchg    al,  beta
        mov     alfa,  al
```

Programare în limbaj de asamblare
Lucrarea nr. 3.

Instrucţiuni aritmetice şi logice

- instrucţiuni specifice adunării: ADD, ADC, INC, DAA, AAA
- instrucţiuni specifice scăderii: SUB, SBB, DEC, NEG, CMP, DAS, AAS
- instrucţiuni specifice înmulţirii: CBW, CWD, MUL, IMUL, AAM
- instrucţiuni specifice împărţirii: DIV, IDIV, AAD
- instrucţiuni logice: NOT, AND, TEST, OR, XOR
- instrucţiuni de deplasare: SHL, SAL, SHR, SAR
- instrucţiuni de rotaţie: ROL, RCL, ROR, RCR

1. Instrucţiuni specifice adunării

Instrucţiunea ADD (Add - Adună)
ADD dest, sursa
dest ← dest + sursa

Instrucţiunea ADC (Add with Carry - Adună cu transport)
ADC dest, sursa
dest ← dest + sursa + CF

Instrucţiunea INC (Increment - Incrementează)
INC dest
dest ← dest + 1

Instrucţiunea DAA (Decimal Adjust for Addition - Corecţie zecimală după adunare)
Instrucţiunea nu are operanzi şi efectuează corecţia zecimală a acumulatorului AL, după o adunare cu operanzi în format BCD împachetat. Să considerăm, de exemplu, adunarea valorilor BCD 65 şi 17. Aceste valori se reprezintă prin octeţii 65H şi 17H. În urma adunării, se obţine rezultatul 7CH, care este incorect ca rezultat BCD. Operaţia de corecţie conduce la rezultatul 82H, ceea ce reprezintă suma BCD a celor două valori.

Instrucţiunea AAA (ASCII Adjust for Addition - Corecţie ASCII după adunare)
Instrucţiunea nu are operanzi şi efectuează corecţia acumulatorului AX, după operaţii de adunare cu operanzi BCD despachetaţi (o cifră BCD pe un octet). Să considerăm, de exemplu, adunarea valorilor 0309H şi 0104H, care ar corespunde valorilor BCD despachetate 39 şi 14. În urma adunării, se obţine rezultatul 040DH. Instrucţiunea AAA corectează acest rezultat la 0503H care este suma BCD a celor două valori.

2. Instrucţiuni specifice scăderii

Instrucţiunea SUB (Substract - Scade)
SUB dest, sursa
dest ← dest - sursa

Instrucţiunea SBB (Substract with Borrow - Scade cu împrumut)
SBB dest, sursa
dest ← dest - sursa - CF

Instrucţiunea DEC (Decrement - Decrementează)
DEC dest
dest ← dest - 1

Instrucţiunea NEG (Negate - Schimbă semnul)
NEG dest
dest ← 0 - dest

Instrucţiunea CMP (Compare - Compară)
CMP dest, sursa
Execută scăderea temporară *dest - sursa* fără a modifica vreun operand, dar cu poziţionarea bistabililor de condiţie (FLAGS)

Instrucţiunea DAS (Decimal Adjust for Substraction - Corecţie zecimală după scădere)
Instrucţiunea nu are operanzi şi efectuează corecţia zecimală a acumulatorului AL, după o scădere cu operanzi în format BCD împachetat. Să considerăm, de exemplu, scăderea valorilor BCD 52 şi 24. Operaţia de corecţie conduce la rezultatul 28H (52-24=28).

Instrucţiunea AAS (ASCII Adjust for Substraction - Corecţie ASCII după scădere)
Instrucţiunea nu are operanzi şi efectuează corecţia acumulatorului AX, după operaţii de scădere cu operanzi BCD despachetaţi (o cifră BCD pe un octet). Să considerăm, de exemplu, scăderea valorilor 0502H şi 0204H. Instrucţiunea AAS corectează rezultatul la 0208H.

3. Instrucţiuni specifice înmulţirii

Instrucţiunea CBW (Convert Byte to Word - Converteşte octet la cuvânt)
Extinde bitul de semn din AL la întreg registrul AH, obţinându-se astfel o reprezentare a lui AL pe 2 octeţi.

Instrucţiunea CWD (Convert Word to DoubleWord - Converteşte cuvânt la dublu-cuvânt)
Extinde bitul de semn din AX la întreg registrul DX, obţinându-se astfel o reprezentare a lui AX pe 4 octeţi.

Instrucţiunea MUL (Multiply - Înmulţeşte fără semn)
MUL sursa
AH:AL ← AL * sursa (dacă sursa este pe octet)
DX:AX ← AX * sursa (dacă sursa este pe 2 octeţi)

Instrucţiunea IMUL (Integer Multiply - Înmulţeşte cu semn)
IMUL sursa
Este similară cu instrucţiunea MUL. Deosebirea este că operaţia de înmulţire se face considerând operanzii numere cu semn.

Instrucţiunea AAM (ASCII Adjust for Multiply - Corecţie ASCII după înmulţire)
Instrucţiunea nu are operanzi şi efectuează corecţia acumulatorului AX, după o înmulţire pe 8 biţi cu operanzi BCD despachetaţi (o cifră BCD pe un octet). Să considerăm, de exemplu, înmulţirea valorilor 5 şi 9. Instrucţiunea AAM corectează rezultatul 2DH la 0405H.

4. Instrucţiuni specifice împărţirii

Instrucţiunea DIV (Divide - Împarte fără semn)
DIV sursa
Dacă sursa este pe octet:
AL ← AX / sursa
AH ← AX *mod* sursa
Dacă sursa este pe 2 octeţi:
AX ← DX:AX / sursa
DX ← DX:AX *mod* sursa

Instrucţiunea IDIV (Integer Divide - Împarte cu semn)
IDIV sursa
Este similară cu instrucţiunea DIV. Deosebirea este că operaţia de împărţire se face considerând operanzii numere cu semn.

Instrucţiunea AAD (ASCII Adjust for Division - Corecţie ASCII înainte de împărţire))
Instrucţiunea nu are operanzi şi efectuează corecţia acumulatorului AX. Operaţia de corecţie trebuie făcută înainte de împărţirea unui număr BCD pe 2 octeţi la un număr BCD pe un octet.

5. Instrucţiuni logice

Instrucţiunea NOT (Not - Negare logică bit cu bit)
NOT dest

Instrucţiunea AND (And - Şi logic bit cu bit)
AND dest, sursa

Instrucţiunea TEST (Test - Testează)
TEST dest, sursa
Efectul este execuţia unei operaţii AND între cei doi operanzi, fără a se modifica destinaţia, dar cu poziţionarea flagurilor la fel ca la instrucţiunea AND.

Instrucţiunea OR (Or - Sau logic bit cu bit)
OR dest, sursa

Instrucţiunea XOR (Exclusive Or - Sau-exclusiv bit cu bit)
XOR dest, sursa

6. Instrucţiuni de deplasare

Instrucţiunea SHL/SAL (Shift Logic/Arithmetic Left - Deplasează logic/aritmetic la stânga)
SHL operand, contor
SAL operand, contor
Bitul cel mai semnificativ trece în CF, după care toţi biţii se deplasează la stânga cu o poziţie. Bitul cel mai puţin semnificativ devine 0. Numărul operaţiilor este dat de *contor*.

Instrucţiunea SHR (Shift Logic Right - Deplasează logic la dreapta)
SHR operand, contor
Bitul cel mai puţin semnificativ trece în CF, după care toţi biţii se deplasează la dreapta cu o poziţie. Bitul cel mai semnificativ devine 0. Numărul operaţiilor este dat de *contor*.

Instrucţiunea SAR (Shift Arithmetic Right - Deplasează aritmetic la dreapta)
SAR operand, contor
Singura diferenţă faţă de instrucţiunea SHR, este că se conservă bitul de semn, mai precis, completarea dinspre stânga se face cu bitul de semn.

7. Instrucţiuni de rotire

Instrucţiunea ROL (Rotate Left - Roteşte la stânga)
ROL operand, contor
Bitul cel mai semnificativ trece atât în CF, cât şi în bitul cel mai puţin semnificativ, după ce toţi biţii s-au deplasat la stânga cu o poziţie. Numărul operaţiilor este dat de *contor*.

Instrucţiunea RCL (Rotate Left through Carry- Roteşte la stânga prin carry)
RCL operand, contor
Bitul cel mai semnificativ trece în CF, toţi biţii se deplasează la stânga cu o poziţie, iar CF original trece în bitul cel mai puţin semnificativ. Numărul operaţiilor este dat de *contor*.

Instrucţiunea ROR (Rotate Right - Roteşte la dreapta)
ROR operand, contor
Bitul cel mai puţin semnificativ trece atât în CF, cât şi în bitul cel mai semnificativ, după ce toţi biţii s-au deplasat la dreapta cu o poziţie. Numărul operaţiilor este dat de *contor*.

Instrucţiunea RCR (Rotate Right through Carry- Roteşte la dreapta prin carry)
RCR operand, contor
Bitul cel mai puţin semnificativ trece în CF, toţi biţii se deplasează la dreapta cu o poziţie, iar CF original trece în bitul cel mai semnificativ. Numărul operaţiilor este dat de *contor*.

Aplicaţii

1. Care este conţinutul registrului AL după executarea următoarei secvenţe de instrucţiuni:
```
mov     al, 10011110b
mov     cl, 3
rol     al, cl
```
2. Care va fi conţinutul registrului CX după executarea următoarei secvenţe de instrucţiuni:
```
mov     bx, sp
mov     cx, 2134H
push    cx
mov     byte ptr ss:[bx-2], 22H
pop     cx
```
3. Care este conţinutul registrului AX după executarea următoarei secvenţe de instrucţiuni:
```
mov     cl, 4
mov     ax, 0702H
shl     al, cl
shr     ax, cl
```
4. Care este conţinutul registrului AX după executarea următoarei secvenţe de instrucţiuni:
```
mov     cx, 1234H
push    cx
mov     bx, 2359H
push    bx
mov     ax, 4257H
mov     bp, sp
mov     byte ptr [bp], al
pop ax
```
5. Să se scrie un program care adună numerele 145A789FH şi 92457ABCH, afişând rezultatul pe ecran.
6. Să se scrie un program care adună două numere în format BCD împachetat. Fiecare din cele două numere are 4 cifre. Numerele sunt memorate în zona de date. Programul afişează rezultatul adunării.
7. Să se scrie un program care adună două numere în format BCD despachetat. Fiecare din cele două numere are 4 cifre. Numerele sunt memorate în zona de date. Programul afişează rezultatul adunării.
8. Să se scrie un program care afişează pe ecran pătratul unui număr. Se va folosi instrucţiunea XLAT.

Programare în limbaj de asamblare
Lucrarea nr. 4.

Instrucţiuni de apel de procedură şi de salt

Forma generală pentru definirea unei proceduri este:

```
nume_proc PROC [FAR | NEAR]
        ……
        RET
nume_proc ENDP
```

unde *nume_proc* este numele procedurii, iar parametrii opţionali FAR sau NEAR indică tipul procedurii. Procedurile sunt de două tipuri: FAR şi NEAR. O procedură FAR poate fi apelată şi din alte segmente de cod decât cel în care este definită, pe când o procedură NEAR poate fi apelată numai din segmentul de cod în care este definită. Dacă se omit parametrii FAR sau NEAR, tipul procedurii este dedus din directivele simplificate de definire a segmentelor (modelul de memorie folosit). De exemplu , modelul LARGE presupune că toate procedurile sunt implicit de tip FAR.

În mod corespunzător, există apeluri de tip FAR, respectiv NEAR, precum şi instrucţiuni de revenire de tip FAR, respectiv NEAR. Instrucţiunea RET (Return) provoacă revenirea în programul apelant; tipul instrucţiunii este dedus din tipul procedurii (NEAR sau FAR). Putem folosi o instrucţiune de revenire explicită: RETN (Return Near) sau RETF (Return Far).

1. Apelul procedurilor şi revenirea din proceduri

Instrucţiunea CALL (Apel de procedură)

```
CALL nume_proc
CALL FAR PTR nume_proc
CALL NEAR PTR nume_proc
```

În primul caz, tipul apelului este dedus din tipul procedurii, iarîn celelalte este specificat explicit (FAR sau NEAR). Tipul apelului trebuie să coincidă cu tipul procedurii şi cu tipul instrucţiunilor Return din interiorul procedurii, altfel se ajunge la funcţionări defectuoase ale programului. În cazul unui apel de procedură de tip NEAR, se salvează în stivă conţinutul registrului IP, care reprezintă adresa de revenire, iar apoi în IP se încarcă adresa primei instrucţiuni din procedură. În cazul unui apel de tip FAR, se salvează în stivă CS:IP, adresa completă de revenire (pe 32 de biţi), iar apoi în CS:IP se încarcă adresa primei instrucţiuni din procedură.

Instrucţiunea RET (Return - Revenire din procedură)

```
RET
RETF
RETN
```

În primul caz, tipul tipul instrucţiunii este dedus din tipul procedurii. În cazul unei reveniri de tip NEAR, se reface registrul IP din stivă, astfel se transferă controlul la instrucţiunea care urmează instrucţiunii CALL care a provocat apelul procedurii. În cazul unei reveniri de tip FAR, se reface din stivă perechea de registre CS:IP.

Instrucţiunea JMP (Jump - Salt)

```
JMP     tinta
JMP     SHORT  PTR     tinta
JMP     NEAR  PTR      tinta
JMP     FAR  PTR       tinta
```

În primul caz, tipul saltului este dedus din atributele expresiei care precizează ţinta. Ţinta specifică adresa de salt şi poate fi o etichetă sau o expresie. Există trei tipuri de instrucţiuni de salt:

- SHORT - adresa ţintă se află la o adresă în domeniul [-127, +127] faţă de adresa instrucţiunii de salt;
- NEAR - adresa ţintă este în acelaşi segment de cod cu instrucţiunea de salt;
- FAR - adresa ţintă poate fi în alt segment de cod faţă de instrucţiunea de salt.

2. Tipuri de salt/apel

JMP/CALL direct
Operandul care se află în formatul instrucţiunii este o etichetă care identifică adresa ţintă. Poate fi de două tipuri:

- salt/apel direct intrasegment (NEAR) - eticheta este în acelaşi segment de cod cu instrucţiunea JMP/CALL;
- salt/apel direct intersegment (FAR) - eticheta poate fi definită şi în alt segment de cod decât cel care conţine instrucţiunea JMP/CALL.

JMP/CALL indirect
Operandul care apare în formatul instrucţiunii reprezintă o adresă de memorie. Poate fi de două tipuri:

- salt/apel indirect intrasegment (NEAR), cu forma generală
  JMP/CALL expr
  în care *expr* precizează adresa efectivă a ţintei şi poate fi un registru, o variabilă de tip WORD, sau un cuvânt din memorie;
- salt/apel indirect intersegment (FAR), cu forma generală
  JMP/CALL expr
  în care *expr* precizează adresa completă a ţintei şi poate fi o variabilă de tip DWORD, sau un dublu-cuvânt din memorie.

3. Instrucţiuni de salt condiţionat

Instrucţiunile din această categorie implementează salturi condiţionate de valoarea unor bistabili (FLAGS). Dacă condiţia nu este îndeplinită, saltul nu are loc, deci execuţia continuă cu instrucţiunea următoare. Toate instrucţiunile de salt condiţionat sunt de tip SHORT, ceea ce înseamnă că adresa ţintă trebuie să fie la o distanţă cuprinsă între -127 şi +127 de octeţi faţă de instrucţiunea de salt. În tabelul următor se prezintă instrucţiunile de salt condiţionat:

Instrucţiune	**Condiţie de salt**	**Interpretare**
JE, JZ	ZF = 1	Zero, Equal
JL, JNGE	SF ≠ OF	Less, Not Greater or Equal
JLE, JNG	SF ≠ OF sau ZF = 1	Less or Equal, Not Greater
JB, JNAE, JC	CF = 1	Below, Not Above or Equal, Carry
JBE, JNA	CF = 1 sau ZF = 1	Below or Equal, Not Above
JP, JPE	PF = 1	Parity, Parity Even
JO	OF = 1	Overflow
JS	SF = 1	Sign
JNE, JNZ	ZF = 0	Not Zero, Not Equal
JNL, JGE	SF = OF	Not Less, Greater or Equal
JNLE, JG	SF = OF şi ZF = 0	Not Less or Equal, Greater
JNB, JAE, JNC	CF = 0	Not Below, Above or Equal, Not Carry
JNBE, JA	CF = 0 şi ZF = 0	Not Below or Equal, Above
JNP, JPO	PF = 0	Not Parity, Parity Odd
JNO	OF = 0	Not Overflow
JNS	SF = 0	Not Sign

La comparaţiile cu semn folosim GREATER şi LESS, iar la comparaţiile fără semn folosim ABOVE şi BELOW.

Uneori este necesar să folosim instrucţiuni de salt condiţionat la etichete care ies în afara domeniului [-127, +127] faţă de instrucţiunea curentă. În această situaţie înlocuim saltul pe o condiţie directă "departe" cu un salt pe condiţia negată "aproape" şi cu un salt necondiţionat "departe". În următorul exemplu, eticheta *et1* se află în afara domeniului, astfel instrucţiunea:

```
        JE  et1
```

se înlocuieşte cu:

```
        JNE  et2
        JMP et1
    et2:
```

4. Instrucţiuni pentru controlul buclelor de program

Instrucţiunea JCXZ (Jump if CX is Zero - Salt dacă CX este zero)

JCXZ eticheta

Eticheta trebuie să se afle în domeniul [-127, +127] faţă de instrucţiunea curentă. Se face salt la eticheta specificată dacă CX conţine valoarea 0.

Instrucţiunea LOOP (Ciclare)

LOOP eticheta

Această instrucţiune este, de fapt, un salt condiţionat de valoarea registrului CX. Cu alte cuvinte, se decrementează CX şi, dacă acesta este diferit de zero, se sare la eticheta specificată. Eticheta trebuie să se afle în domeniul [-127, +127] faţă de instrucţiunea curentă.

Instrucţiunea LOOPZ/LOOPE (Loop While Zero/Equal - Ciclează cât timp este zero/egal)

LOOPZ eticheta
LOOPE eticheta

Se decrementează CX şi, dacă acesta este diferit de zero şi ZF este 1 (rezultatul ultimei operaţii a fost zero), se sare la eticheta specificată.

Instrucţiunea LOOPNZ/LOOPNE (Loop While Not Zero/Equal - Ciclează cât timp nu este zero/egal)

LOOPNZ eticheta
LOOPNE eticheta

Se decrementează CX şi, dacă acesta este diferit de zero şi ZF este 0 (rezultatul ultimei operaţii a fost zero), se sare la eticheta specificată.

Aplicaţii

1. Rulaţi pas cu pas următorul program:

```
.model small
.stack 512
.data
        tab_proc dw proc_1
                 dw proc_2
                 dw proc_3
        tab_procf dd procf_1
                  dd procf_2
                  dd procf_3
        intra    dw etich1
                 dw etich2
                 dw etich3
        inter    dd etif1
                 dd etif2
.code
        proc_1 proc
                push dx
                pop dx
                ret
        proc_1 endp
        proc_2 proc
                push dx
                pop dx
                ret
        proc_2 endp
        proc_3 proc
                push dx
                pop dx
                ret
        proc_3 endp
        procf_1 proc far
                push dx
                pop dx
                ret
        procf_1 endp
        procf_2 proc far
                push dx
                pop dx
```

```
        ret
procf_2 endp
procf_3 proc far
        push dx
        pop dx
        ret
procf_3 endp
start:  mov     ax, @data
        mov     ds, ax
        mov     si, 0
        jmp     inter[si]
etif2   label   far
        jmp     intra
proced: lea     bx, proc_1
        call    bx
        call    tab_proc
        mov     si, 2
        call    tab_proc[si]
        lea     bx, tab_proc
        call    word ptr [bx]
        mov     si, 4
        call    word ptr [bx][si]
        call    tab_procf
        mov     si, 4
        call    tab_procf[si]
        lea     bx, tab_procf
        call    dword ptr [bx]
        mov     si, 4
        call    dword ptr [bx][si]
        jmp     sfarsit
etich1: mov     si, 2
        jmp     intra[si]
etich3  label near
        jmp     proced
etich2  label near
        lea     bx, intra
        jmp     word ptr [bx+4]
etif1   label far
        lea     bx, inter
        mov     si, 4
        jmp     dword ptr [bx][si]
        sfarsit:mov ah, 4ch
        int     21h
end     start
```

2. Să se scrie un program care să afişeze rezultatul obţinut în urma conversiei unui număr hexazecimal în zecimal.
3. Să se scrie un program care să calculeze factorialul unui număr şi apoi să afişeze rezultatul.
4. Care este conţinutul registrului AL după executarea următoarei secvenţe de instrucţiuni:

```
        mov     al, 'A'
        mov     bl, 'B'
        cmp     al, bl
        jne     et1
et2:    mov     al, 1
        jmp     sfarsit
et1:    mov     al, 0
        jmp     sfarsit
```

5. Care este conţinutul registrului AL după executarea următoarei secvenţe de instrucţiuni:

```
        masca   equ   01000000b
        mov     al,   10110101b
        test    al,   masca
        jz      et1
et2:    mov     al,   1
        jmp     sfarsit
et1:    mov     al,   0
        jmp     sfarsit
```

Programare în limbaj de asamblare
Lucrarea nr. 5.

Instrucţiuni pentru operaţii cu şiruri

Prin şir se înţelege o secvenţă de octeţi sau de cuvinte, aflate la adrese succesive de memorie. Setul de instrucţini cuprinde operaţii primitive (la nivel de octet sau cuvânt) şi prefixe de repetare, care pot realiza repetarea operaţiilor primitive de un număr fix de ori sau până la îndeplinirea unei condiţii de tip comparaţie. Denumirea instrucţiunilor fără operanzi la nivel de octet se termină cu litera B, iar a celor la nivel de cuvânt cu litera W. Aceste instrucţiuni folosesc registrele DS:SI ca adresă sursă şi ES:DI ca adresă destinaţie. Există şi variantele cu operanzi ale acestor instrucţiuni, şi ele se folosesc în situaţia în care se utilizează alt segment pentru şirul sursă, decât segmentul de date implicit (DS). Toate aceste instrucţiuni incrementează (dacă DF este 0) sau decrementează (dacă DF este 1) în mod automat adresele (regiştrii SI, DI) cu 1 sau cu 2, după cum operaţia este la nivel de octet sau de cuvânt. Starea flagului DF poate fi modificată prin instrucţiunile CLD (Clear Direction) şi STD (Set Direction), care şterg, respectiv setează acest bistabil.

1. Operaţii primitive

Instrucţiunile MOVSB, MOVSW (Move String - Copiază şir)

MOVSB
MOVSW

ES:DI ← DS:SI
SI ← SI + delta
DI ← DI + delta

Instrucţiunile CMPSB, CMPSW (Compare String - Compară şiruri)

CMPSB
CMPSW

DS:SI - ES:DI
SI ← SI + delta
DI ← DI + delta

Instrucţiunile LODSB, LODSW (Load String - Încarcă şir)

LODSB
LODSW

acumulator ← DS:SI (*acumulator* este AX sau AL în funcţie de tipul operaţiei)
SI ← SI + delta

Instrucţiunile STOSB, STOSW (Store String - Depune şir)

STOSB
STOSW

ES:DI ← acumulator (*acumulator* este AX sau AL în funcţie de tipul operaţiei)
DI ← DI + delta

Instrucţiunile SCASB, SCASW (Scan String - Căutare în şir)

SCASB
SCASW

acumulator - ES:DI (*acumulator* este AX sau AL în funcţie de tipul operaţiei)
DI ← DI + delta

2. Prefixe de repetare

Prefixele de repetare permit execuţia repetată a unei operaţii primitive cu şiruri de octeţi sau de cuvinte, funcţie de un contor de operaţii sau de un contor şi o condiţie logică. Aceste prefixe nu sunt instrucţiuni în sine, ci participă la formarea unor instrucţiuni compuse, alături de operaţiile primitive descrise mai sus.

Prefixul de repetare REP (Repeat - Repetă)
REP op_primitiva

Repetă operaţia primitivă şi decrementează CX, cât timp conţinutul registrului CX este diferit de zero. De obicei se foloseşte la primitivele de tip MOVS, LODS şi STOS.

Prefixul de repetare REPE/REPZ (Repeat While Zero/Equal - Repetă cât timp este zero/egal)
REPE/REPZ op_primitiva

Repetă operaţia primitivă şi decrementează CX, cât timp conţinutul registrului CX este diferit de zero şi rezultatul operaţiei primitive este 0. De obicei se foloseşte la primitivele de tip CMPS şi SCAS.

Prefixul REPNE/REPNZ (Repeat While Not Zero/Equal - Repetă cât timp nu este zero/egal)
REPNE/REPNZ op_primitiva

Repetă operaţia primitivă şi decrementează CX, cât timp conţinutul registrului CX este diferit de zero şi rezultatul operaţiei primitive este de asemenea diferit de 0. De obicei se foloseşte la primitivele de tip CMPS şi SCAS.

Observaţie: Numărul elementelor unui şir se poate calcula folosind contorul de locaţii. În exemplul următor, expresia *$-text* reprezintă numărul octeţilor de la adresa *text* până la adresa curentă:

```
text    db    'A B C D E F G H J K L M N O P Q R'
ltext   equ   $-text
```

De asemenea, operatorul LENGTH întoarce numărul de elemente, iar operatorul SIZE întoarce dimensiunea în octeţi a unei variabile. Astfel pentru definiţia:

```
tab     dw    100 dup(?)
```

expresia *length tab* are valoarea 100. Pentru aceeaşi definiţie, expresia *size tab* are valoarea 200.

3. Aplicaţii

1. Să se scrie un program care copiază un şir într-un alt şir.
2. Căutarea unui anumit caracter într-un şir. Se va afişa pe ecran poziţia caracterului în şir.
3. Să se scrie un program care dă indicele caracterului începând de la care două şiruri diferă.
4. Să se scrie un program care afişează pe ecran numărul de apariţii a unui şir într-un alt şir.
5. Să se scrie un program care adună două şiruri de cifre zecimale aflate în zona de date. Rezultatul este un şir care conţine pe fiecare poziţie suma cifrelor de pe poziţiile corespunzătoare din cele două şiruri.
6. Precizaţi care este efectul următoarei secvenţe de instrucţiuni:

```
mov    al, 'D'
mov    cx, 12
rep    stosb
```

Programare în limbaj de asamblare
Lucrarea nr. 6.

1. Forma completă de definire a segmentelor

Se utilizează o declaraţie de forma:

```
nume_seg   SEGMENT   [tip_aliniere][tip_combinare][clasa_seg]
           …
nume_seg   ENDS
```

Iniţializarea unui registru de segment (DS, ES sau SS) cu un segment declarat trebuie facută de utilizator în mod explicit în cadrul programului, utilizând pentru aceasta numele segmentului respectiv:

```
mov     ax, nume_seg
mov     ds, ax
```

- *tip_aliniere* - poate fi BYTE, WORD, PARA, PAGE (implicit este PARA), şi arată că adresa de început a zonei de memorie rezervată segmentului este divizibilă cu 1/2/16/256.
- *tip_combinare* - este o informaţie pentru editorul de legături care indică raportul dintre acest segment şi segmente definite în alte module obiect. Acest parametru poate fi:
  PUBLIC - arată că acest segment va fi concatenat cu alte segmente cu acelaşi nume, din alte module obiect, rezultând un singur modul cu acest nume;
  COMMON - precizează că acest segment şi alte segmente cu acelaşi nume din alte module vor fi suprapuse, deci vor începe de la aceeaşi adresă. Lungimea unui astfel de segment va fi lungimea celui mai mare segment dintre cele suprapuse;
  AT <expresie> - segmentul va fi încărcat în memorie la adresa fizică absolută de memorie reprezentată de valoarea expresiei, care este o valoare de 16 biţi;
  MEMORY - segmentul curent de acest tip va fi aşezat în memorie în spaţiul rămas disponibil după aşezarea celorlalte segmente;
  STACK - va concatena toate segmentele cu acelaşi nume, pentru a forma un segment unic; referirea acestui segment se va face prin SS:SP. Registrul SP va fi iniţializat automat cu lungimea stivei.
- *clasa_seg* - este un nume cuprins între apostrofuri. Rolul este de a permite stabilirea modului în care se aşează în memorie segmentele unui program. Două segmente cu aceeaşi clasă vor fi aşezate în memorie la adrese succesive.

Asocierea segmentelor cu registrele de segment se realizează cu pseudoinstrucţiunea ASSUME:

```
ASSUME  reg_seg : nume_seg
```

Unde *reg_seg* poate fi unul dintre registrele CS, DS, ES sau SS, iar *nume_seg* este numele unui segment sau al unui grup de segmente, care va fi adresat de registrul respectiv.
Definirea grupurilor de segmente se realizează cu directiva GROUP, care are următoarea formă:

```
nume_grup GROUP  lista_nume_segmente
```

unde *nume_grup* este numele grupului de segmente, ce va fi folosit pentru a determina adresa de segment utilizată pentru referirea în cadrul grupului de segmente, iar *lista_nume_segmente* poate conţine nume de segmente sau expresii de forma:

```
SEG nume_variabila
SEG nume_eticheta
```

Într-o astfel de listă nu poate să apară numele unui alt grup.

Exemplu:

```
data1 segment
        v1 db 5
data1 ends
```

```
data2 segment
        v2 dw 25
data2 ends
data3 segment
        v3 dw 100
data3 ends
dgrup group data1, data2
cod segment
        assume cs:cod, ds:dgrup, es:data3
  start: mov ax, dgrup
        mov ds, ax
        mov ax, data3
        mov es, ax
        ;referiri la date
        mov bx, v1        ;se utilizează DS pentru ca a fost asociat cu dgrup
        add v3, bx        ;se utilizează ES pentru ca a fost asociat cu segmentul data3
cod ends
end start
```

2. Forma simplificată de definire a segmentelor

Această modalitate de definire a segmentelor respectă acelaşi format ca şi la programele dezvoltate în limbaje de nivel înalt.

.MODEL tip_model
Prin această directivă se specifică dimensiunea şi modul de dispunere a segmentelor în memorie. Modelul de memorie poate fi:

- tiny - toate segmentele (date, cod, stivă) formează un singur segment de cel mult 64KB. Apelurile de procedură şi salturile sunt de tip NEAR şi se folosesc adrese efective (offset) pentru accesarea datelor;
- small - datele şi stiva formează un segment şi codul un alt segment. Fiecare din acestea va avea dimensiunea maxima de 64KB. Apelurile de procedură şi salturile sunt de tip NEAR şi se folosesc adrese efective (offset) pentru accesarea datelor;
- medium - datele şi stiva sunt grupate într-un singur segment (cel mult egal cu 64KB), dar codul poate fi în mai multe segmente, deci poate depăşi 64KB. Apelurile de procedură şi salturile sunt implicit de tip FAR şi se folosesc adrese efective (offset) pentru accesarea datelor;
- compact - codul generat ocupă cel mult 64KB, dar datele şi stiva pot fi în mai multe segmente (pot depăşi 64KB). Apelurile de procedură şi salturile sunt de tip NEAR şi se folosesc adrese complete (segment şi offset) pentru accesarea datelor aflate în alte segmente;
- large - atât datele cât şi codul generat pot depăşi 64KB. Apelurile de procedură şi salturile sunt implicit de tip FAR şi se folosesc adrese complete (segment şi offset) pentru accesarea datelor aflate în alte segmente;
- huge - este asemănător modelului *large*, dar structurile de date pot depăşi 64KB. La modelele *compact* şi *large*, o structură compactă de date (de tip tablou) nu poate depăşi limitele unui segment fizic (64KB); la modelul *huge*, această restricţie dispare.

.STACK [dimensiune]
Această directivă alocă o zonă de memorie de dimensiune specificată pentru segmentul de stivă. Dacă nu se specifică parametrul *dimensiune*, aceasta va fi implicit de 1KB.

.CODE [nume]
Această directivă precede segmentul de cod. Încărcarea adresei acestui segment în registrul CS se face automat de către sistemul de operare, la încărcarea segmentului pentru execuţie. Opţional se pot asocia nume (maxim 6 caractere) pentru segmentele de cod.

.DATA
Această directivă precede segmentul de date. Utilizatorul trebuie să iniţializeze, în mod explicit, registrul de segment DS, cu adresa segmentului de date. Simbolul @*data* primeşte adresa segmentului de date după linkeditare.

3. Definirea structurilor. Operaţii specifice

Structurile reprezintă colecţii de date plasate succesiv în memorie, grupate sub un unic nume sintactic. Dimensiunea unei structuri este suma dimensiunilor câmpurilor componente. Forma generală este:

```
nume_structura  STRUC
        nume_membru  definitie_date
nume_structura  ENDS
```

Tipul structurii se defineşte fără a se rezerva spaţiu de memorie. Numele membrilor structurii trebuie să fie distincte, chiar dacă aparţin unor structuri distincte. Iată un exemplu de definiţie a unei structuri:

```
alfa  struc
        a  db     ?
        b  dw     1111H
        c  dd     1234
alfa  ends
```

O structură definită este interpretată ca un tip nou de date, care poate apoi participa la definiţii concrete de date, cu rezervare de spaţiu de memorie. De exemplu, putem defini o structură de tipul *alfa*, cu numele *x*:

```
x  alfa  <, , 777>
```

în care *alfa* este tipul de date, *x* este numele variabilei, iar între paranteze unghiulare se trec valorile iniţiale ale câmpurilor structurii *x*. Prezenţa virgulelor din lista de valori iniţiale precizează că primii doi membrii sunt iniţializaţi cu valorile implicite de la definiţia tipului *alfa* , iar al treilea membru este iniţializat cu valoarea 777. Astfel, definiţia variabilei *x* este echivalentă cu secvenţa de definiţii:

```
a  db     ?
b  dw     1111H
c  dd     777
```

Principalul avantaj al structurilor este accesul la membri într-o formă asemănătoare limbajelor de nivel înalt. De exemplu, pentru a încărca în SI câmpul *b* al structurii *x*, putem scrie:

```
mov  si,  x.b
```

Putem acum defini un tablou de 5 structuri de tip *alfa*:

```
tab  alfa  5 dup (<, , 10>)
```

în care primii 2 membri sunt iniţializaţi cu valorile de la definiţia tipului structurii, iar al treilea cu valoarea 10. Având în vedere că o dată de tip *alfa* ocupă 7 octeţi:

- tab[0].a   se referă la câmpul *a* al primei structuri;
- tab[7].c   se referă la câmpul *c* al celei de-a doua structuri;
- tab[14].a   se referă la câmpul *a* al celei de-a treia structuri;
- tab[21].b   se referă la câmpul *b* al celei de-a patra structuri.

4. Definirea înregistrărilor. Operaţii specifice

Înregistrările corespund de fapt unor structuri împachetate din limbajele de nivel înalt. Concret, o înregistrare este o definiţie de câmpuri de biţi de lungime maximă 8 sau 16. Forma generală este:

```
nume_inregistrare  RECORD  nume_camp:valoare [= expresie], …
```

unde *valoare* reprezintă numărul de biţi pe care se memorează câmpul respectiv. Opţional poate fi folosită o expresie a cărei valoare să fie asociată câmpului respectiv, pe numărul de biţi precizat. La fel ca la structuri, numele câmpurilor trebuie să fie distincte, chiar dacă aparţin unor înregistrări diferite.
Să considerăm un exemplu:

```
beta  record  x:7, y:4, z:5
```

prin care se defineşte un şablon de 16 biţi, *x* pe primii 7 biţi (mai semnificativi), *y* pe următorii 4 biţi, şi *z* pe ultimii 5 biţi (mai puţin semnificativi). La fel ca la structuri, putem acum defini variabile de tipul înregistrării *beta*:

```
val  beta  <5, 2, 7>
```

în care valorile dintre parantezele unghiulare iniţializează cele trei câmpuri de biţi. Această definiţie este echivalentă cu definiţia explicită:

```
val  dw  0000101001000111B
```

Operatorul MASK primeşte un nume de câmp şi furnizează o mască cu biţii 1 pe poziţia câmpului respectiv şi 0 în rest. Astfel, expresia MASK y este echivalentă cu constanta binară 0000000111100000B.
Operatorul WIDTH primeşte un nume de înregistrare sau un nume de câmp şi întoarce numărul de biţi ai înregistrării sau ai câmpului respectiv. De exemplu, secvenţa:

```
mov  al,  width  beta
mov  bl,  width  y
```

va încărca în AL 16 şi în BL valoarea 4.

5. Macroinstrucţiuni

Macroinstrucţiunile permit programatorului să definească simbolic secvenţe de program (instrucţiuni, definiţii de date, directive, etc.), asociate cu un nume. Folosind numele macroinstrucţiunii în program, se va genera întreaga secvenţă de program. În esenţă, este vorba de un proces de substituţie (expandare) în textul programului, care se petrece înainte de asamblarea programului. Spre deosebire de proceduri, macroinstrucţiunile sunt expandate la fiecare utilizare. Forma generală a unei macroinstrucţiuni este:

```
nume_macro  MACRO  [p1, p2, …,pn]
        …
ENDM
```

Opţional, macroinstrucţiunile pot avea parametri, în acest caz, *p1, p2, …, pn* sunt identificatori care specifică parametrii formali. Apelul unei macroinstrucţiuni constă în scrierea în program a numelui macroinstrucţiunii. Dacă macroinstrucţiunuea are parametri, apelul se face prin specificarea numelui, urmată de o listă de parametri actuali:

```
nume_macro  x1, x2, …, xn
```

La expandarea macroinstrucţiunii, pe lângă expandarea propriu-zisă, se va înlocui fiecare parametru formal cu parametrul actual corespunzător.

6. Aplicaţii

1. Se consideră următoarea definiţie de date:

   ```
   alfa  struc
      a1  db  ?
      a2  db  21H
      a3  dw 5176H
   alfa  ends
   tab  alfa    10 dup (<3H, ?, 2252H>)
   ```

   Care va fi conţinutul registrului AL după executarea următoarelor instrucţiuni?

   a) mov al, byte ptr tab[12].a2
   b) mov al, byte ptr tab[11].a2

2. Să se definească o macroinstrucţiune care realizează adunarea a două valori pe 32 de biţi. Să se scrie şi un exemplu de apelare al macroinstrucţiunii definite.

Programare în limbaj de asamblare
Lucrarea nr. 7.

1. Întreruperi

Noţiunea de întrerupere presupune intreruperea programului în curs de execuţie şi transferul controlului la o rutină de tratare. Mecanismul prin care se face acest transfer este, în esenţă, de tip apel de procedură, ceea ce înseamnă revenirea în programul întrerupt, după terminarea rutinei de tratare. Pe lângă rutinele de tratare ale sistemului de operare, pot fi folosite şi propriile rutine. Întreruperile software apar ca urmare a execuţiei unor instrucţiuni, cum ar fi INT, INTO, DIV, IDIV. Intreruperile hardware externe sunt provocate de semnale electrice care se aplică pe intrările de întreruperi ale procesorului, iar cele interne apar ca urmare a unor condiţii speciale de funcţionare a procesorului (cum ar fi execuţia pas cu pas a programelor).

Într-un sistem cu procesor 8086, pot exista maxim 256 de întreruperi. Fiecare nivel de întrerupere poate avea asociată o rutină de tratare (procedură de tip far). Adresele complete (4 octeţi) ale acestor rutine sunt memorate în tabela vectorilor de intrerupere, pornind de la adresa fizică 0 până la 1KB. Intrarea în tabelă se obţine înmulţind codul întreruperii cu 4. Rutina de tratare a întreruperii se termină obligatoriu cu instrucţiunea IRET.

2. Redirectarea întreruperii de pe nivelul 0

Împărţirea unui număr foarte mare la un număr mic, poate duce la apariţia întreruperii pe nivelul 0. Pentru obţinerea rezultatului corect, se redirectează întreruperea de pe nivelul 0 spre o rutină care realizează împărţirea în situaţia în care împărţitorul este diferit de 0. Implementaţi programul de mai jos, şi rulaţi-l pas cu pas.

```
.model small
.stack 100
.data
demp  dd   44444444h
imp   dw   1111h
cat   dd   ?
rest  dw   ?
.code
                    ;procedura primeste deimpartitul in (DX, AX) si impartitorul in BX
                    ;returneaza catul in (BX, AX) si restul in DX
divc  proc
      cmp   bx, 0                                   ;daca eroare adevarata
      jnz   divc1
      int   0                                       ;apel tratare impartire cu 0
divc1:
      push  es                                      ;salvare registre modificate de procedura
      push  di
      push  cx
      mov   di, 0
      mov   es, di                                  ;adresa pentru intrarea intrerupere nivel 0
      push  es:[di]                                 ;salvare adresa oficiala
      push  es:[di+2]
      mov   word ptr es:[di], offset trat_0         ;incarcare vector intrerupere
      mov   es:[di+2],cs                            ; pentru noua rutina de tratare
      div   bx                                      ;incercare executie instructiune de impartire
                                                    ;nu a aparut eroare
      sub   bx, bx                                  ;daca impartirea s-a executat corect se pune 0
                    ;in bx ca sa corespunda modului de transmitere al parametrilor stabilit
```

```
revenire:
        pop     es:[di+2]
        pop     es:[di]
        pop     cx
        pop     di
        pop     es
        ret
trat_0 proc far
        push    bp                                              ;salvare bp
        mov     bp, sp
                        ;schimba adresa de revenire din rutina trat_0, adresa care se gaseste in stiva
                                ;a fost pusa in stiva la apelare rutinei de intrerupere (IP-ul)
        mov     word ptr [bp+2], offset revenire
        push    ax                                              ;salvare ax, Y0
        mov     ax, dx                                          ;primul deimpartit Y1
        sub     dx, dx
                                                                ;executie impartire Y1 la X
                                                                ;rezulta (AX) = Q1, (DX) = R1
        div     bx
        pop     cx                                              ;Y0
        push    ax                                              ;salvare Q1
        mov     ax, cx
                                                                ;executie impartire (R1, AX) la X
                                                                ;rezulta AX=Q0, DX=R0
        div     bx
        pop     bx                                              ;Q1
        pop     bp
        iret
trat_0 endp
divc   endp
start:  mov     ax, @data
        mov     ds, ax
        mov     ax, word ptr demp
        mov     dx, word ptr demp+2
        mov     bx, imp
        call    divc
        mov     word ptr cat, ax
        mov     word ptr cat+2, bx
        mov     rest, dx
        mov     ah, 4ch
        int     21h
end start
```

Programare în limbaj de asamblare
Lucrarea nr. 8.

1. Transferul parametrilor către proceduri

1.1. Tipuri de transfer (prin valoare sau prin referinţă)

O primă chestiune care trebuie decisă în proiectarea unei proceduri este tipul de transfer al parametrilor. Se pot utiliza două tipuri de transfer:

- transfer prin valoare, care implică transmiterea conţinutului unei variabile; se utilizează atunci când variabila care trebuie transmisă nu este alocată în memorie, ci într-un registru;
- transfer prin referinţă, care implică transmiterea adresei de memorie a unei variabile; se utilizează atunci când trebuie transmise structuri de date de volum mare (tablouri, structuri, tablouri de structuri, etc.). Se mai foloseşte atunci când procedura trebuie să modifice o variabilă parametru formal alocată în memorie.

Să considerăm o procedură *aduna*, de tip near, care adună două numere pe 4 octeţi, întorcând rezultatul în perechea de registre DX:AX:

```
.data
        n1   dd  12345H
        n2   dd  54321H
        rez  dd  ?
.code
        aduna proc near
                add  ax, bx
                adc  dx, cx
                ret
        aduna endp
                ……
        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 aduna
        mov  word ptr rez, ax
        mov  word ptr rez+2, dx
```

Să considerăm acum varianta transmiterii prin referinţă a parametrilor, în care se transmit către procedură adresele de tip near ale variabilelor n1 şi n1.

```
        aduna proc near
                mov  ax, [si]
                add   ax, [di]
                mov  dx, [si+2]
                adc   dx, [di+2]
                ret
        aduna endp
                ……
        lea  si, n1
        lea  di, n2
        call  near ptr aduna
        mov  word ptr rez, ax
        mov  word ptr rez+2, dx
```

1.2. Transfer prin registre

Avantajul transferului prin registre constă în faptul că, în procedură, parametrii actuali sunt disponibili imediat. Principalul dezavantaj al acestei metode constă în numărul limitat de registre. Pentru conservarea registrelor se foloseşte următoarea secvenţă de apel:

- salvare în stivă a registrelor implicate în transfer;
- încărcare registre cu parametri actuali;
- apel de procedură;
- refacere registre din stivă.

1.3. Transfer prin zonă de date

În această variantă, se pregăteşte anterior o zonă de date şi se transmite către procedură adresa acestei zone de date. În exemplul următor transferul parametrilor către procedura *aduna* se face printr-o zonă de date:

```
.data
        zona   label  dword
        n1   dd  12345H
        n2   dd  54321H
        rez  dd  ?
.code
        aduna proc near
                mov  ax, [bx]
                add   ax, [bx+4]
                mov  dx, [bx+2]
                adc   dx, [bx+6]
                ret
        aduna endp
                ……
        lea  bx, zona
        call  near ptr aduna
        mov  word ptr rez, ax
        mov  word ptr rez+2, dx
```

1.4. Transfer prin stivă

Transferul parametrilor prin stivă este cea mai utilizată modalitate de transfer. Avantajele acestei metode sunt uniformitatea şi compatibilitatea cu limbajele de nivel înalt. Tehnica de acces standard la parametrii procedurii se bazează pe adresarea bazată prin registrul BP, care presupune registrul SS ca registru implicit de segment. Accesul se realizează prin următoarele operaţii, efectuate la intrarea în procedură:

- se salvează BP în stivă;
- se copiază SP în BP;
- se salvează în stivă registrele utilizate de procedură;
- se accesează parametrii prin adresare indirectă cu BP.

La încheierea procedurii, se execută următoarele operaţii:

- se refac registrele salvate;
- se reface BP;
- se revine în programul apelant prin RET.

Calculul explicit al deplasamentelor parametrilor reprezintă o sursă potenţială de greşeli, de aceea se utilizează un şablon care conţine imaginea stivei, de la registrul BP în jos. Să considerăm procedura *aduna* de tip near, care primeşte parametrii n1 şi n2 prin stivă

```
.data
        n1   dd  12345H
        n2   dd  54321H
        rez  dd  ?
```

```
        sablon struc
                _bp       dw ?
                _ip       dw ?
                n2_low    dw ?
                n2_high   dw ?
                n1_low    dw ?
                n1_high   dw ?
        sablon ends
.code
        aduna proc near
                push  bp
                mov  bp, sp
                mov  ax, [bp].n1_low
                add   ax, [bp].n2_low
                mov  dx, [bp].n1_high
                adc   dx, [bp].n2_high
                pop   bp
                ret
        aduna endp
                ……
        push  word ptr n1+2
        push  word ptr n1
        push  word ptr n2+2
        push  word ptr n2
        call  near ptr aduna
        add  sp,8
        mov  word ptr rez, ax
        mov  word ptr rez+2, dx
```

În cazul în care procedura este de tip far, pentru adresa de revenire trebuie rezervate în şablon 4 octeţi (adresă completă):

```
        sablon struc
                _bp       dw ?
                _cs_ip    dd  ?
                n2_low    dw ?
                n2_high   dw ?
                n1_low    dw ?
                n1_high   dw ?
        sablon ends
.code
        aduna proc far
                push  bp
                mov  bp, sp
                mov  ax, [bp].n1_low
                add   ax, [bp].n2_low
                mov  dx, [bp].n1_high
                adc   dx, [bp].n2_high
                pop   bp
                ret
        aduna endp
                ……
        push  word ptr n1+2
        push  word ptr n1
        push  word ptr n2+2
        push  word ptr n2
        call  far ptr aduna
        add  sp,8
        mov  word ptr rez, ax
        mov  word ptr rez+2, dx
```

2. Întoarcerea datelor de către proceduri

Există următoarele modalităţi de a întoarce datele de către proceduri:

- în lista de parametri apar adresele rezultatelor sau adresa zonei de date care conţine câmpuri pentru rezultate;
- rezultatele se întorc prin registre;
- rezultatele se întorc în vârful stivei.

Prima tehnică a fost deja descrisă la transmiterea parametrilor prin zonă de date. Practic, în interiorul procedurii se depun explicit rezultatele la adresele conţinute în parametrii formali respectivi. A doua tehnică, transferul rezultatelor prin registre, se foloseşte cel mai frecvent. De obicei se foloseşte registrul acumulator, eventual extins (AL, AX, respectiv DX:AX, după cum rezultatul este pe 1, 2 sau 4 octeţi). A treia tehnică, transferul rezultatelor prin stivă, se foloseşte destul de rar, fiind total nestandard.

3. Aplicaţii

1. Să se scrie un program care determină numărul biţilor 1 dintr-un număr pe 32 de biţi citit de la tastatură (se vor folosi instrucţiunile SHL şi ADC);
2. Să se scrie o procedură care realizează adunarea a două valori citite de la tastatură. Numerele sunt formate din maxim 32 de cifre.
3. Să se scrie un program care converteşte un număr din format BCD împachetat în format BCD despachetat. Numărul este format din 10 cifre. Rezultatul va fi memorat în zona de date. Pentru conversie se va utiliza o procedură care primeşte ca parametri adresa numărului BCD împachetat şi adresa la care va memora numărul BCD despachetat. Cei doi parametri sunt transmişi prin stivă.

Observaţii:

- citirea unui caracter de la tastatură se realizează folosind funcţia 01H a întreruperii 21H. Se încarcă în registrul AH valoarea 01H, se apelează întreruperea 21H, care va pune codul ASCII al caracterului citit în AL;
- afişarea unui caracter pe ecran se realizează folosind funcţia 02H a întreruperii 21H. Se încarcă în registrul AH valoarea 02H. Se încarcă în DL codul ASCII al caracterului care trebuie afişat şi apoi se apelează întreruperea 21H, care va afişa caracterul pe ecran;
- afişarea unui şir de caractere pe ecran (şirul se termină cu caracterul $, care are codul ASCII 24H) se realizează folosind funcţia 09H a întreruperii 21H. Se încarcă în registrul AH valoarea 09H. Se încarcă în DS adresa de segment a şirului care trebuie afişat, şi în DX adresa efectivă (offsetul). Se apelează întreruperea 21H, care va afişa şirul pe ecran.
- pentru trecerea la linie nouă se afişează şirul de caractere:

  ```
  nl  db 13, 10, 24H
  ```

  unde 13, 10 reprezintă codul ENTER-ului.

Programare în limbaj de asamblare
Lucrarea nr. 9.

1. Proceduri recursive

Numim procedură recursivă o procedură care se autoapelează (direct sau indirect). Forma generală a unei funcţii recursive este următoarea:

```
tip f(lista_parametri_formali){
        if (!conditie_de_oprire){
                ...
                ... f(lista_parametri_reali)
                ...
        }
}
```

Recursivitatea poate fi utilizată pentru a rezolva elegant multe probleme, dar procedurile recursive sunt adesea mai lente decât corespondentele lor nerecursive:

- la fiecare apel se depun în stivă valorile parametrilor (dacă există) şi adresa de revenire (vezi lucrarea de laborator nr. 6)
- complexitatea algoritmilor recursivi este de obicei mai mare decât a celor iterativi.

Aplicaţie

Enunţ :
Să se realizeze o funcţie recursivă ce calculează n!

Rezolvare :

Plecăm de la formula recursivă de calcul $n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$ cunoscută din liceu.

Rescriind formula într-o variantă mai apropiată de implementare, avem:

$$fact(n) = \begin{cases} 1, & n = 0 \\ n * fact(n-1), & n > 0 \end{cases}$$

Pentru a înţelege mai uşor programul realizat în limbaj de asamblare, să urmărim pentru început programul C, care declară o funcţie recursivă *fact*, citeşte de la tastatură o valoare, şi foloseşte funcţia pentru a returna factorialul acestei valori:

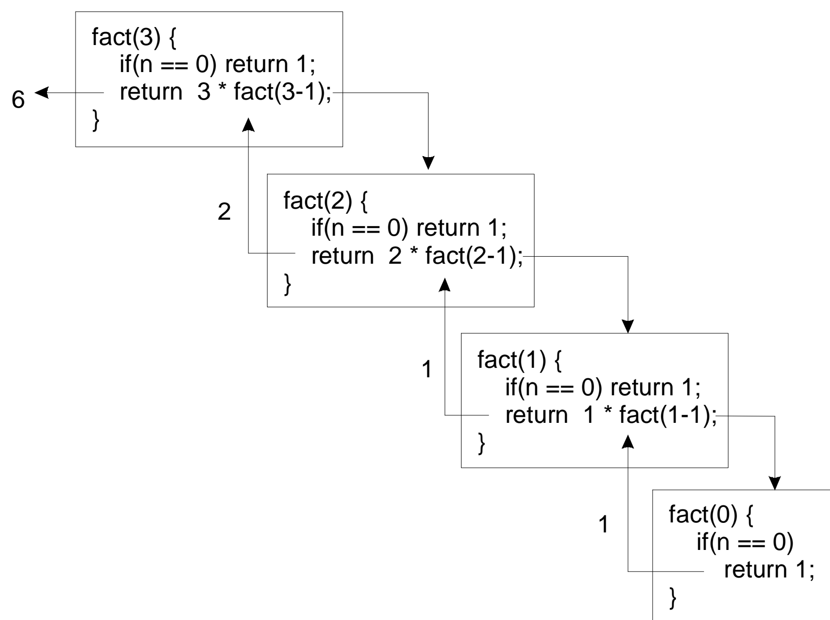
```
#include <stdio.h>

long fact(int n){
        if(n==0) return 1;          //conditia de oprire
        return n*fact(n-1)          //apel recursiv: n!=n(n-1)!
}
void main(){
        int n;
        scanf("%d",&n);             //citire n
        printf("%ld",fact(n));      //afisare n!
}
```

Analiză:

La orice funcţie recursivă trebuie precizată o condiţie de ieşire. În problema factorialului, condiţia de ieşire este 0! care, prin definiţie, este 1. Dacă parametrul primit de *fact* este 0, atunci funcţia returnează 1, altfel returnează rezultatul înmulţirii dintre valoarea parametrului şi factorialul apelat cu valoarea parametrului minus 1.

Pentru fact(3) avem următoarea succesiune de operaţii:

(1) apelează recursiv fact(2);
(2) apelează recursiv fact(1);
(3) apelează recursiv fact(0);
(4) returnează 1 – revenire din fact(0), calculează 1*fact(0);
(5) returnează 1 – revenire din fact(1), calculează 2*fact(1);
(6) returnează 2 – revenire din fact(2), calculează 3*fact(2);
(7) returnează 6 – revenire din fact(3);



Implementarea programului în limbaj de asamblare este următoarea:

```
.model small
.stack
sablon struc
        _bp dw ?
        _cs_ip dw ?
        _n dw ?
sablon ends
.data
        n dw 7
        rez dd ?
.code
        fact proc near
                push bp                     ;salvare bp
                mov bp, sp                  ;initializare cu varful stivei
                pushf                       ;salvare indicatori
                push bx
                mov bx, word ptr [bp]._n    ;preluarea parametrului
                cmp bx, 0                   ;conditia de oprire
                jne rec
                mov ax, 1                   ;0!=1
```

```
          mov dx, 0
          jmp stop
     rec:dec bx                    ;termenul urmator
          push bx                  ;transferul parametrului
          call near ptr fact       ;apel recursiv, cu rezultat in DX:AX
          add sp, 2
          mul word ptr [bp]._n
    stop:pop bx                    ;refacerea registrului bx
          popf                     ;refacere indicatori
          pop bp
          retn
fact endp
afis proc near
          push ax                  ;salvarea registrelor
          push bx
          push cx
          push dx
          mov dx, word ptr rez+2   ;preluare din rez
          mov ax, word ptr rez
          mov cx, 0                ;initializarea contorului
          mov bx, 10
    next:div bx                    ;se obtine pe rand in dx fiecare cifra zecimala
          push dx                  ;salvarea in stiva e necesara pentru afisarea in ordinea corecta
          mov dx, 0
          inc cx
          cmp ax, 0
          jne next
   print:pop dx                    ;preluare din stiva
          add dl, 30h              ;conversie la codul ASCII
          mov ah, 02h
          int 21h                  ;afisare
          loop print
          pop dx                   ;refacerea registrelor
          pop cx
          pop bx
          pop ax
          retn
afis endp
start:
          mov ax, @data            ;initializare registru segment
          mov ds, ax
          mov ax, n
          push ax                  ;transferul parametrului  prin stiva
          call near ptr fact       ;DX:AX<--rezultatul
          add sp, 2
          mov word ptr rez+2, dx   ;rezultatul se depune in rez
          mov word ptr rez, ax
          call near ptr afis       ;afisarea rezultatului
          mov ah, 4ch              ;revenire DOS
          int 21h
end start
```

Pentru preluarea parametrului din stivă, procedura *fact* foloseşte următoarea structură şablon:

```
sablon struc
          _bp dw ?
          _cs_ip dw ?
          _n dw ?
sablon ends
```

Deoarece în problema factorialului, condiţia de ieşire este 0! care, prin definiţie, este 1, după preluarea parametrului, valoarea acestuia se compară cu 0. În caz de egalitate, în DX:AX se depune valoarea 1 şi se face salt la eticheta *stop* (procedura întoarce valoarea 1). Dacă valoarea parametrului nu este 0, se returnează rezultatul înmulţirii dintre valoarea parametrului şi factorialul apelat cu valoarea parametrului minus 1.
Să urmărim din nou succesiunea de operaţii pentru acelaşi exemplu (3!), apelăm deci procedura fact cu valoarea 3 trimisă ca parametru:

(1) apelează recursiv procedura *fact*, valoarea parametrului este 2;
(2) apelează recursiv procedura *fact*, valoarea parametrului este 1;
(3) apelează recursiv procedura *fact*, valoarea parametrului este 0;
(4) returnează 1 în DX:AX – revenire din *fact(0)*;
(5) returnează 1 în DX:AX – revenire din fact(1);
(6) returnează 2 în DX:AX – revenire din fact(2);
(7) returnează 6 în DX:AX – revenire din fact(3);

Procedura *afis* preia rezultatul din *rez* (rezultatul se depune în rez înainte de a apela procedura *afis*), converteşte această valoare în zecimal şi o afişează.

2. Aplicaţii

1. Modificaţi programul prezentat, înlocuind procedura nerecursivă *afis* cu o procedură recursivă

Să se realizeze funcţii care rezolvă recursiv următoarele probleme:

2. Calculaţi $2^n = \begin{cases} 1, n = 0 \\ 2 \cdot 2^{n-1}, n > 0 \end{cases}$

3. Calculaţi cel de-al n-lea număr Fibonacci $fibo(n) = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ fibo(n-1) + fibo(n-2), n > 1 \end{cases}$

4. Calculaţi $C_n^k = \begin{cases} 1, n = k \\ 1, k = 0 \\ C_{n-1}^{k-1} + C_{n-1}^{k}, \; in\, rest \end{cases}$

Programare în limbaj de asamblare
Lucrarea nr. 10.

1. Aplicaţii mixte C-ASM

1.1. Transferul parametrilor

- compilatorul de Borland C realizează tra nsferul parametrilor spre funcţii prin stivă (în ordinea dreapta-stânga), apelul unei funcţii C din ASM presupune plasarea parametrilor în stivă şi apoi apelarea funcţiei C;
- deoarece în C descărcarea stivei este făcută de modulul apelant, un program ASM trebuie să descarce stiva după apelarea unei funcţii C.

1.2. Întoarcerea rezultatelor

După executarea unei funcţiiC, rezultatul se va întoarce în funcţie de dimensiunea zcestuia, în felul următor:

- 1 octet: în registrul AL (char);
- 2 octeţi: în registrul AX (int şi short);
- 4 octeţi: în perechea de registre DX:AX (long şi float);
- float (4 octeţi), double (8 octeţi), long double (10 octeţi): într-o zonă specială a bibliotecii de virgulă mobilă sau în vârful stivei coprocesorului matematic. O soluţie pentru rezultatele de tip real ar fi ca funcţia să întoarcă pointeri la aceste valori.

1.3. Numele simbolurilor externe

În C numele simbolurilor externe (nume de variabile, nume de funcţii) sunt generate implicit cu caracterul "_" precedând numele simbolului. Cu alte cuvinte, parametrii din C, utilizaţi şi în modulul scris în limbaj de asamblare, vor fi precedaţi de caracterul "_". În modulul în care simbolurile sunt definite, ele se declară PUBLIC, iar în modulele în care ele sunt referite se declară EXTRN. În unul din module (C sau ASM), trebuie să existe funcţia main, care în ASM se declară în felul următor:

```
public  _main
_main  proc
      ……….
      ret
_main  endp
```

Următoarea aplicaţie mixtă calculează recursiv factorialul unui număr citit de la tastatură:

Modulul C:

```
#include <stdio.h>
#include <conio.h>
```

Modulul ASM:

```
.model tiny
extrn _clrscr:near, _scanf:near, _printf:near, _getch:near
public _main
.stack 1024
sablon struc
        _bp    dw ?
        _ip    dw ?                                   ;_cs_ip  dd ?  - modificare I
        _n     dw ?
sablon ends
.data
    n          dw ?
    MesRd      db 'Introduceti un intreg: ',0
    MesWr      db 'Factorialul este %d...',0
    scan       db '%d',0
```

```
.code
    Factorial proc near                                   ;Factorial proc far  - modificare I
        push bp
        mov bp,sp
                                                          ;+ push ds - modificare II
                                                          ;+ push es - modificare II
        push dx
        push bx
        mov bx,[bp]._n
        cmp bx,1
        jne et1
        mov ax,1
        jmp et2
        et1: dec bx
        push bx                                           ; parametru salvat in stiva
        call near ptr Factorial                           ; call far ptr Factorial - modificare I
        add sp,2
        mul [bp]._n
        et2:    pop bx
        pop dx                                            ; ax - contine rezultatul deci nu se restaureaza
                                                          ;+ pop es - modificare II
                                                          ;+ pop ds - modificare II
        pop bp                                            ; ===> ax nu trebuie salvat in stiva la inceputul proc
        retn                                              ;retf  - modificare I
    Factorial endp
    _main proc near                                       ;_main proc far - modificare I
        call near ptr _clrscr                             ; call far ptr _clrscr - modificare I
                                                          ;+ mov ax, seg MesRd - modificare II
                                                          ;+ push ax - modificare II
        lea ax, MesRd
        push ax
        call near ptr _printf                             ; call far ptr _printf - modificare I
        add sp,2                                          ; add  sp, 4 - modificare II
                                                          ;+ mov ax, seg n - modificare II
                                                          ;+ push ax - modificare II
        lea ax, n
        push ax
                                                          ;+ mov ax, seg scan - modificare II
                                                          ;+ push ax - modificare II
        lea ax, scan
        push ax
        call near ptr _scanf                              ; call far ptr _scanf - modificare I
        add sp,4                                          ; add sp, 8 - modificare II
        mov ax,n
        push ax
        call near ptr Factorial                           ; call far ptr Factorial - modificare I
        add sp,2
        push ax
                                                          ;+ mov ax, seg MesWr - modificare II
                                                          ;+ push ax - modificare II
        lea ax,MesWr
        push ax
        call near ptr _printf                             ; call far ptr _printf - modificare I
        add sp,4                                          ; add sp, 6 - modificare II
        call near ptr _getch                              ; call far ptr _getch  - modificare I
        retn                                              ; retf  - modificare I
    _main endp
end
```

Observaţie: Modificările I trebuie făcute în cazul utilizării modelelor de memorie cu apel de procedură de tip far, iar modificările II trebuie făcute atunci când se folosesc modele de memorie cu adrese fizice (complete). În cel de-al doilea caz, simbolul @data trebuie modificat cu simbolul DGROUP.

2. Aplicaţii

Să se realizeze funcţii care rezolvă recursiv următoarele probleme:

1. Calculaţi suma cifrelor unui număr $sumac(n) = \begin{cases} 0, & n = 0 \\ n\%10 + sumac(n/10), & n > 0 \end{cases}$

2. Calculaţi $cmmdc(a,b) = \begin{cases} a, & b = 0 \\ cmmdc(b, a\%b), & b > 0 \end{cases}$

3. Se dă un şir ordonat crescător. Realizaţi funcţia ce implementează căutarea binară.

$$cauta(s,d,val) = \begin{cases} 0, & s > d \\ 1, & a[m] = val,\ unde\ m = (s+d)/2 \\ cauta(s, m-1, val), & a[m] > val \\ cauta(m+1, d, val), & a[m] < val \end{cases}$$

4. Suma elementelor unui vector $sumas(i) = \begin{cases} a[0], & i = 0 \\ a[i] + sumas(i-1), & i > 0 \end{cases}$

5. Verificarea dacă un vector e palindrom $pali(s,d) = \begin{cases} 1, & s >= d \\ 0, & a[s] != a[d] \\ pali(s+1, d-1), & in\ rest \end{cases}$

6. Descompunerea unui număr în factori primi $desc(n,d) = \begin{cases} 1, & n = 1 \\ scrie\ d,\ desc(n/d, d), & daca\ d \mid n \\ desc(n, d+1), & in\ rest \end{cases}$

7. Conversia unui număr din baza 10 în baza 2.

8. Calculul coeficientului de grad *n* al polinomului Cebîşev de speta I.

$$T_n(x) = \begin{cases} 1, & n=0 \\ x, & n=1 \\ 2 \cdot T_{n-2}(x) - T_{n-1}(x) \end{cases}$$

9. Calculaţi valoarea functiei Ackermann $A(m,n) \begin{cases} n+1, & m = 0 \\ A(m-1, 1), & n = 0 \\ A(m-1, A(m, n-1)), & in\ rest \end{cases}$

10. Cautarea minimului şi maximului intr-un sir de *n* valori.

11. Determinarea numărului de apariţii a unei valori într-un şir.

Bibliografie

[1] G. Muscă, *Programare în limbaj de asamblare*, Editura Teora, 1998.
[2] V. Lungu, *Procesoare INTEL - Programare în limbaj de asamblare*, Editura Teora, 2001.
[3] R. Baciu, *Programarea în limbaje de asamblare*, Editura ALMA MATER, Sibiu, 2003.
[4] A. Florea, L. Vinţan, *Simularea şi optimizarea arhitecturilor de calcul în aplicaţii practice*, Editura MATRIX ROM, Bucureşti, 2003.