

# CAPITOLUL 4

## INSTRUCȚIUNI ALE LIMBAJULUI DE ASAMBLARE

Forma generală a unui program în NASM + scurt exemplu:

```
global start                ; solicităm asamblorului sa confere vizibilitate globală simbolului denumit start
                             (eticheta start va fi punctul de intrare în program)

extern ExitProcess, printf  ; informăm asamblorul că simbolurile ExitProcess și printf au proveniență străină,
                             ; evitând astfel a fi semnalate erori cu privire la lipsa definirii acestora

import ExitProcess kernel32.dll ;precizăm care sunt bibliotecile externe care definesc cele două simboluri:
                             ; ExitProcess e parte a bibliotecii kernel32.dll (bibliotecă standard a sistemului de operare)
import printf msvcrt.dll      ; printf este funcție standard C și se regăsește în biblioteca msvcrt.dll (SO)

bits 32                    ; solicităm asamblarea pentru un procesor X86 (pe 32 biți)

segment code use32 class=CODE ; codul programului va fi emis ca parte a unui segment numit code

start:
    ; apel printf("Salut din ASM")
    push dword string ; transmitem parametrul funcției printf (adresa șirului) pe stivă (așa cere printf)
    call [printf]      ; printf este numele unei funcții (etichetă = adresă , trebuie indirectată cu [])

    ; apel ExitProcess(0), 0 reprezentând "execuție cu succes"
    push dword 0
    call [ExitProcess]

segment data use32 class=DATA ; variabilele vor fi stocate în segmentul de date (denumit data)
string: db "Salut din ASM!", 0
```

## 4.1. MANIPULAREA DATELOR/ 4.1.1. Instrucțiuni de transfer al informației

### 4.1.1.1. Instrucțiuni de transfer de uz general

<b>MOV</b> <i>d,s</i>	<d> <-- <s> (b-b, w-w, d-d)	-
<b>PUSH</b> <i>s</i>	ESP = ESP - 4 și depune <s> în stivă (s – dublucuvânt)	-
<b>POP</b> <i>d</i>	extrage elementul curent din stivă și îl depune în d (d – dublucuvânt) ESP = ESP + 4	-
<b>XCHG</b> <i>d,s</i>	<d> ↔ <s> s,d – trebuie sa fie L-values !!!	-
<b>[reg_segment] XLAT</b>	AL ← < DS:[EBX+AL] > sau AL ← < reg_segment:[EBX+AL] >	-
<b>CMOVcc d, s</b>	<d> ← <s> dacă cc (cod condiție) este adevărat	-
<b>PUSHA / PUSHAD</b>	Depune pe stiva EAX, ECX, EDX, EBX, ESP, EBP, ESI si EDI	-
<b>POPA / POPAD</b>	Extrage EDI, ESI, EBP, ESP, EBX, EDX, ECX si EAX de pe stiva	-
<b>PUSHF / PUSHFD</b>	Depune EFlags pe stivă	-
<b>POPF / POPFD</b>	Extrage vârful stivei și il depune în EFlags	-
<b>SETcc d</b>	<d> ← 1 dacă cc este adevărat, altfel <d> ← 0	-

Dacă operandul destinație al instrucțiunii MOV este unul dintre cei 6 regiștrii de segment atunci sursa trebuie să fie unul dintre cei opt regiștri generali de 16 biți ai UE sau o variabilă de memorie. Încărcătorul de programe al sistemului de operare preinițializează în mod automat regiștrii de segment, iar schimbarea valorilor acestora, deși posibilă din punct de vedere al procesorului, nu aduce nici o utilitate (un program este limitat la a încărca doar valori de selectori ce indică înspre segmente preconfigurate de către sistemul de operare, fără a putea să definească segmente adiționale).

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

**PUSH** *s*    și    **POP** *d*

Operanzii trebuie să fie reprezentați pe dublucuvânt, deoarece stiva este organizată pe dublucuvinte. Stiva crește de la adrese mari spre adrese mici, din 4 în 4 octeți, ESP punctând întotdeauna spre dublucuvântul din vârful stivei.

Funcționarea acestor instrucțiuni poate fi ilustrată prin intermediul unei secvențe echivalente de instrucțiuni MOV și ADD sau SUB:

push eax	⇔	sub esp, 4	; pregătim (alocăm) spațiu pentru a stoca valoarea
		mov [esp], eax	; stocăm valoarea în locația alocată
pop eax	⇔	mov eax, [esp]	; încărcăm în eax valoarea din vârful stivei
		add esp, 4	; eliberăm locația

În perspectiva evaluării efectului unor instrucțiuni ca **PUSH ESP** sau **POP dword [ESP]** trebuie precizat și mai clar ordinea în care se efectuează (sub)operațiile componente ale instrucțiunilor PUSH și POP:

- a). Se evaluează operandul **sursă** al instrucțiunii (ESP pt PUSH sau respectiv elementul din vârful stivei pt POP)
- b). Se actualizează corespunzător ESP (ESP := ESP-4 pt PUSH și respectiv ESP := ESP+4 pt POP)
- c). Se efectuează atribuirea implicată de efectul instrucțiunii asupra operandului **destinație** (noul vârf al stivei pt PUSH și respectiv dword [ESP] (acesta fiind acum după scăderea ESP de la pct b) elementul de sub vârful inițial al stivei) - pt POP

Presupunând ca situația inițială este ESP = 0019FF74, după **PUSH ESP** vom avea ESP = 0019FF70 și conținutul din vârful stivei va fi acum 0019FF74.

Presupunând ca situația inițială este ESP = 0019FF74 și ca în această locație se afla valoarea 7741FA29 (vârful stivei), după **POP dword [ESP]** vom avea ESP = 0019FF78 și conținutul acestei locații (conținutul locației din vârful stivei) va fi 7741FA29 (deci am putea spune că „vârful stivei se mută cu o poziție mai jos”!!).

Instrucțiunile PUSH si POP permit doar depunerea și extragerea de valori reprezentate pe cuvânt și dublucuvânt. Ca atare, **PUSH AL nu reprezintă o instrucțiune validă** (syntax error), deoarece operandul nu este permis a fi o valoare pe octet. Pe de altă parte, secvența de instrucțiuni

```
PUSH ax    ; depunem ax
PUSH ebx   ; depunem ebx
POP  ecx   ; ecx <- dublucuvântul din vârful stivei (valoarea lui ebx)
POP  dx    ; dx <- cuvântul ramas în stivă (deci valoarea lui ax)
```

este corectă și echivalentă prin efect cu

```
MOV  ecx,  ebx
MOV  dx,   ax
```

Adițional acestei constrângeri (inerentă tuturor procesoarelor x86), sistemul de operare impune ca operarea stivei să fie obligatoriu făcută doar prin accese pe dublucuvânt sau multipli de dublucuvânt, din motive de compatibilitate între programele de utilizator și nucleul și bibliotecile de sistem. Implicația acestei constrângeri este că o instrucțiune de forma PUSH operand<sub>16</sub> sau POP operand<sub>16</sub> (de exemplu PUSH word 10), deși este suportată de către procesor și asamblată cu succes de către asamblor, nu este recomandată, putând cauza ceea ce poartă numele de eroare de dezaliniere a stivei: stiva este corect aliniată dacă și numai dacă valoarea din registrul ESP este în permanență divizibilă cu 4!

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

**XCHG** *operand1, operand2*

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

**[reg\_segment] XLAT**

*tabelă de translatare* este **adresa directă** a unui șir de octeți. Instrucțiunea XLAT pretinde la intrare adresa far a tabelii de translatare furnizată sub unul din următoarele două moduri:

- DS:EBX (implicit, dacă lipsește precizarea registrului segment)
- registru\_segment:EBX, dacă registrul segment este precizat explicit

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

De exemplu, secvența

```
mov ebx, Tabela
mov al, 6
ES xlat                AL ← < ES:[EBX+6] >
```

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

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

```
segment data use32
    TabHexa    db    '0123456789ABCDEF'
    numar resb 1
    .          .          .

segment code use32
    mov ebx, TabHexa
    .          .          .
    mov al, numar
    xlat                ; AL ← < DS:[EBX+AL] >
```

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

#### 4.1.1.3. Instrucțiunea de transfer al adreselor LEA

LEA <i>reg_general, continutul unui operand din memorie</i>	reg_general <-- offset(mem)	-
---	-----------------------------	---

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

```
lea  eax,[v]
```

încarcă în EAX offsetul variabilei v, instrucțiune echivalentă cu

```
mov  eax, v
```

Instrucțiunea **LEA** are însă avantajul că operandul sursă poate fi o expresie de adresare (spre deosebire de instrucțiunea **mov** care nu accepta pe post de operand sursă decât o variabilă cu adresare directă într-un astfel de caz). De exemplu, instrucțiunea

```
lea  eax,[ebx+v-6]      avand ca efect „mov eax, ebx+v-6”
```

nu are ca echivalent direct o singură instrucțiune **MOV**, instrucțiunea

```
mov  eax, ebx+v-6
```

fiind incorectă sintactic deoarece expresia *ebx+v-6* nu este determinabilă la momentul asamblării.

Prin utilizarea directă a valorilor deplasamentelor ce rezultă în urma calculelor de adrese (în contrast cu folosirea memoriei indicate de către acestea), **LEA** se evidențiază prin versatilitate și eficiență sporite: versatilă prin combinarea unei înmulțiri cu adunări de regiștri și/sau valori constante și eficiență ridicată datorată execuției întregului calcul într-o singură instrucțiune, fără a ocupa

circuitele ALU care rămân astfel disponibile pentru alte operații (timp în care calculul de adresă este efectuat de către circuite specializate, separate, ale BIU).

Exemplu: înmulțirea unui număr cu 10

```
mov eax, [număr]      ; eax <- valoarea variabilei număr  
lea eax, [eax * 2]     ; eax <- număr * 2  
lea eax, [eax * 4 + eax] ; eax <- (eax * 4) + eax = eax * 5 = (număr * 2) * 5
```

#### **4.1.1.4. Instrucțiuni asupra flagurilor**

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

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

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

Instrucțiunea **PUSHF** transferă toți indicatorii în vârful stivei (conținutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectați în urma acestei operații. Instrucțiunea **POPF** extrage cuvântul din vârful stivei și transferă din acesta indicatorii corespunzători în registrul de flag-uri.

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

<b>CLC</b>	CF=0	CF
<b>CMC</b>	CF = ~CF	CF
<b>STC</b>	CF=1	CF
<b>CLD</b>	DF=0	DF
<b>STD</b>	DF=1	DF

CLI, STI – actioneaza asupra flagului de intrerupere (IF). Functioneaza efectiv doar in programarea sub 16 biti, aici la programarea sub 32 biti SO interzicand accesul la flag-ul de intreruperi.