C A P I T O L U L 4
INSTRUCŢIUNI 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 şi 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 funcţie standard C şi se regăseşte în biblioteca msvcrt.dll (SO)

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

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 funcţiei printf (adresa şirului) pe stivă (aşa cere printf)
        call   [printf]     ; printf este numele unei funcţii (etichetă = adresă , trebuie indirectată cu [])

        ; apel ExitProcess(0), 0 reprezentând ”execuţie 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. Instrucţiuni de transfer al informaţiei

4.1.1.1. Instrucţiuni de transfer de uz general

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

Dacă operandul destinaţie al instrucţiunii MOV este unul dintre cei 6 regiştrii de segment atunci sursa trebuie să fie unul dintre cei opt regiştri generali de 16 biţi ai UE sau o variabilă de memorie. Încărcătorul de programe al sistemului de operare preiniţializează în mod automat regiştrii de segment, iar schimbarea valorilor acestora, deşi 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 adiţionale).

Instrucţiunile **PUSH** şi **POP** au sintaxa **PUSH** *s* şi **POP** *d*

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

Funcţionarea acestor instrucţiuni poate fi ilustrată prin intermediul unei secvenţe echivalente de instructiuni MOV si ADD sau SUB:

```
push eax  ⇔   sub esp, 4        ; pregătim (alocăm) spaţiu pentru a stoca valoarea
              mov [esp], eax    ; stocăm valoarea in locaţia alocată

pop eax   ⇔   mov eax, [esp]    ; încărcăm în eax valoarea din vârful stivei
              add esp, 4        ; eliberăm locaţia
```

Aceste instrucţiuni permit doar depunerea şi extragerea de valori reprezentate pe cuvânt şi dublucuvânt. Ca atare, PUSH AL nu reprezintă o instrucţiune validă (syntax error), deoarece operandul nu este permis a fi o valoare pe octet. Pe de altă parte, secvenţa de instrucţiuni

```
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 şi echivalentă prin efect cu

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

Adiţional 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 şi nucleul şi bibliotecile de sistem. Implicaţia acestei constrângeri este că o instrucţiune de forma PUSH operand$_{16}$ sau POP operand$_{16}$ (de exemplu PUSH word 10), deşi este suportată de către procesor şi asamblată cu succes de către asamblor, nu este recomandată, putând cauza ceea ce poartă numele de eroare de dezaliniere e stivei: stiva este corect aliniată dacă şi numai dacă valoarea din registrul ESP este în permanenţă divizibilă cu 4!

Instrucţiunea **XCHG** permite interschimbarea conţinutului a doi operanzi de aceeaşi dimensiune (octet, cuvânt sau dublucuvânt), cel puţin unul dintre ei trebuind să fie registru. Sintaxa ei este

XCHG *operand1, operand2*

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

[reg_segment] XLAT

tabelă_de_translatare este adresa directă a unui şir de octeţi. Instrucţiunea XLAT pretinde la intrare adresa far a tabelei de translatare furnizată sub unul din următoarele două moduri:

- DS:EBX (implicit, dacă lipseşte precizarea registrului segment)
- registru_segment:EBX, dacă registrul segment este precizat explicit

Efectul instrucţiunii **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). EXEMPLU: pag.111-112 (curs).

De exemplu, secvenţa

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

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

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

```
segment data use32
        .       .       .
TabHexa     db      '0123456789ABCDEF'
        .       .       .
segment code use32
mov   ebx, TabHexa
        .       .       .
mov   al, numar
xlat                        ; AL ← < DS:[EBX+AL] >
```

O astfel de strategie este des utilizată şi se dovedeşte 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. Instrucţiunea de transfer al adreselor LEA

LEA *reg_general, continutul unui operand din memorie*	reg_general <-- offset(mem)	-

Instrucţiunea **LEA** (***L**oad **E**ffective **A**ddress*) transferă deplasamentul operandului din memorie *mem* în registrul destinaţie. De exemplu

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

încarcă în EAX offsetul variabilei v, instrucţiune echivalentă cu `mov eax, v`

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

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

nu are ca echivalent direct o singură instrucţiune MOV, instrucţiunea

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 (in contrast cu folosirea memoriei indicate de către acestea), LEA se evidenţiază prin versatilitate şi eficienţă sporite: versatilă prin combinarea unei înmulţiri cu adunări de regiştri şi/sau valori constante şi eficienţă ridicată datorată execuţiei întregului calcul într-o singură instrucţiune, fără a ocupa circuitele ALU care rămân astfel disponibile pentru alte operaţii (timp în care calculul de adresă este efectuat de către circuite specializate, separate, ale BIU).

Exemplu: inmulţirea 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. Instrucţiuni asupra flagurilor

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

Instrucţiunea **LAHF** (***Load register AH from Flags***) copiază indicatorii SF, ZF, AF, PF şi CF din registrul de flag-uri în biţii 7, 6, 4, 2 şi respectiv 0 ai registrului AH. Conţinutul biţilor 5,3 şi 1 este nedefinit. Indicatorii nu sunt afectaţi în urma acestei operaţii de transfer (în sensul că instrucţiunea LAHF nu este ea însăşi generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor şi atât).

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

Instrucţiunea **PUSHF** transferă toţi indicatorii în vârful stivei (conţiunutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectaţi în urma acestei operaţii. Instrucţiunea **POPF** extrage cuvântul din vârful stivei şi transferă din acesta indicatorii corespunzători în registrul de flag-uri.

Limbajul de asamblare pune la dispoziţia programatorului nişte *instrucţiuni de setare* a valorii indicatorilor de condiţie, pentru ca programatorul să poată influenţa după dorinţă modul de acţiune a instrucţiunilor care exploatează flaguri.

CLC	CF=0	CF
CMC	CF = ~CF	CF
STC	CF=1	CF
CLD	DF=0	DF
STD	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.