

## CAPITOLUL 3

### ELEMENTELE DE BAZA ALE LIMBAJULUI DE ASAMBLARE

**Limbajul mașină** al unui sistem de calcul (SC) este format din totalitatea instrucțiunilor mașină puse la dispoziție de procesorul SC. Acestea se reprezintă sub forma unor siruri de biți cu semnificație prestabilită.

**Limbajul de asamblare** al unui calculator este un limbaj de programare în care setul de bază al instrucțiunilor coincide cu operațiile mașinii și ale cărui structuri de date coincid cu structurile primare de date ale mașinii. **Limbaj simbolic. Simboluri - Mnemonice + etichete.**

Elementele cu care lucrează un **asamblor** sunt:

- \* **etichete** - nume scrise de utilizator, cu ajutorul căror se pot referi date sau zone de memorie.
- \* **instrucțiuni** - scrise sub forma unor mnemonice care sugerează acțiunea. Asamblorul generează octeții care codifică instrucțiunea respectivă.
- \* **directive** - sunt indicații date asamblorului în scopul generării corecte a octetilor. Ex: relații între modulele obiect, definirea unor segmente, indicații de asamblare condiționată, directive de generare a datelor.
- \* **contor de locații** - număr întreg gestionat de asamblor. În fiecare moment, valoarea contorului coincide cu numărul de octeți generați corespunzător instrucțiunilor și directivelor deja întâlnite în cadrul segmentului respectiv (deplasamentul curent în cadrul segmentului). Programatorul poate utiliza această valoare (accesare doar în citire!) prin simbolul '\$'. **Fiecare segment are propriul său contor de locații !!!**

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the \$ and \$\$ tokens. \$ evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using JMP \$.

\$\$ evaluates to the start of the current section; so you can tell how far into the section are by using (\$-\$) .

#### Directiva SECTION

```
section .data
    db 'hello'
    db 'h', 'e', 'l', 'l', 'o'
    data_segment_size equ $-$
```

\$-\$ = the distance from the beginning of the segment AS A SCALAR (constant numerical value)!!!!!!

\$ - is an offset = POINTER TYPE !!!! It is an address !!!!

\$\$ - is an offset =POINTER TYPE !!!! It is an address !!!!

\$ means "address of here".

\$\$ means "address of start of current section".

So \$-\$ means "current size of section".

For the example above, this will be 10, as there are 10 bytes of data given.

Daca nu am definite sectiuni explicite , atunci \$\$=inceputul segmentului respectiv

## Curs 6

Formula de determinarea a offset-ului: (16 biți)

$$\underbrace{[BX/BP]}_{\text{base}} + \underbrace{[SI/DI]}_{\text{index}} + [\text{constantă}]$$

**Asamblator** = program translator, compilator al limbajului assembly

= generația octetii corespunzători directivelor și instrucțiunilor la nivelul codului surșă

**Compilator** = analizarea corectitudinea sintactică a codului surșă

### Contor de locatii

#### Segment data

a db 17, -2, 0ffh, 'xyz',...  
db ....  
db....

### Contorul de locatii

— nu este atrebuită (REGISTER)  
= căți octeti sunt generați de la începutul segmentului

;lga db \$-a (mov [lga],...); ok //aritmetică de pointeri – scăderea a 2 pointeri = scalar (constanță numerică) - lga=variabilă de memorie (mov [lga],...)

;lga dw \$-\$ ; Corect numai DACA a este primul element definit în data segment !!!!

;lga EQU \$-a ; ok ! însă mov [lga],... este syntax error !!! pt ca lga NU este o variabilă alocată... ci o constantă! (fără alocare de memorie)

;lga dw \$-data ; corect în TASM/MASM, INCORRECT în NASM sub 32 biți !!!  
syntax error – “Expression is not simple or relocatable”

;lga dw lga-a !!!!!!!

b EQU 27 ; b NU este un offset !!!!

c dd 12345678h

;lga dw b-a ; syntax error !!!!! b is NOT an address !!!

;lga dw c-a ; ok !!!!

lga dw \$-a-4 ; ok !!!

lg dw \$-a ; length (a) + 4 !!!

Dacă nu se utilizează nici o directivă section în mod explicit, simbolul \$\$ se va evalua implicit la offset-ul începutului de segment.

Elementul sintactic ":" se pune obligatoriu când definim etichete de cod (ex: "start:") însă nu trebuie pus dacă definim o etichetă de date (ex: definirea de variabile "a db 17")

### 3.1. FORMATUL UNEI LINII SURSA

Formatul unei linii sursă în limbajul de asamblare x86 este următorul:

**[etichetă[:]] [prefixe] [mnemonică] [operanzi] [;comentariu]**

Ilustrăm conceptul prin intermediul a câteva exemple de linii sursă:

aici: jmp acolo	; avem etichetă + mnemonică + operand + comentariu
repz cmpsd	; prefix + mnemonică + comentariu
start:	; etichetă + comentariu
	; doar un comentariu (care putea lipsi și el)
a dw 19872, 42h	; etichetă + mnemonică + 2 operanzi + comentariu

Caracterele din care poate fi constituită o *etichetă* sunt următoarele:

- Litere, atât A-Z cât și a-z;
- Cifre de la 0 la 9;
- Caracterele \_, \$, \$\$, #, @, ~, . și ?

Ca și prim caracter al unei etichete sunt permise doar litere, \_ și ?

Aceste reguli sunt valabile pentru toți *identificatorii* valizi (denumiri simbolice, precum nume de variabile, etichete, macro, etc).

Identifierii NASM sunt *case sensitive*, limbajul diferențiind literele mari de cele mici privitor la denumirile utilizator. Aceasta înseamnă că un identifier Abc este diferit de identifierul abc. Pentru denumirile implicit parte a limbajului, cum ar fi cuvintele cheie, mnemonicile și numele regiștrilor, nu se diferențiază literele mari de cele mici (acestea sunt case insensitive).

La nivelul limbajului de asamblare se întâlnesc două categorii de etichete:

1). **etichete de cod**, care apar în cadrul secvențelor de instrucțiuni cu scopul de a defini destinațiile de transfer ale controlului în cadrul unui program. **Pot apărea și în segmente de date !**

2). **etichete de date**, care identifică simbolic unele locații de memorie, din punct de vedere semantic ele fiind echivalentul noțiunii de *variabilă* din alte limbi. **Pot apărea și în segmente de cod !**

**Valoarea unei etichete în limbaj de asamblare este un număr întreg reprezentând adresa instrucțiunii, directivei sau datelor ce urmează etichetei.**

Distinctia dintre referirea adresei unei variabile sau a conținutului asociat acesteia în NASM se face după regulile:

- Când este specificat între paranteze drepte, numele variabilei desemnează valoarea variabilei, de exemplu [p] specifică accesarea valorii variabilei p, similar cu modul în care \*p semnifică dereferențierea unui pointer (accesul la conținutul indicat prin valoarea pointerului) în C;
- În orice alt context numele variabilei reprezintă adresa variabilei, spre exemplu, p este întotdeauna adresa variabilei p;

Exemple:

mov EAX, et ; încarcă în registrul EAX **adresa** datelor sau a codului marcat cu eticheta et (4 octeți)  
 mov EAX, [et] ; încarcă în registrul EAX **conținutul** de la adresa et (4 octeți)  
 lea eax, [v] ; încarcă în registrul eax **adresa** (offsetul) variabilei v (4 octeți)

Ca generalizare, folosirea parantezelor pătrate indică întotdeauna accesarea unui operand din memorie. De exemplu, mov EAX, [EBX] semnifică un transfer în EAX a conținutului memoriei a cărei adresă este dată de valoarea lui EBX.

Există două tipuri de *mnenomice*: mnemonice de *instrucțiuni* și nume de *directive*. Directivele dirijează asamblorul. Ele specifică modul în care asamblorul va genera codul obiect. Instrucțiunile dirijează procesorul.

*Operanții* sunt parametri care definesc valorile ce vor fi prelucrate de instrucțiuni sau de directive. Ei pot fi **reghiștri, constante, etichete, expresii, cuvinte cheie sau alte simboluri**. Semnificația operanților depinde de

## 3.2. EXPRESII

*expresie* - operanți + operatori. *Operatorii* indică modul de combinare a operanților în scopul formării expresiei. **Expresiile sunt evaluate în momentul asamblării** (adică, valorile lor sunt determinabile la momentul asamblării, cu excepția acelor părți care desemnează conținuturi de regiștri și care vor fi determinate la execuție).

### 3.2.1. Moduri de adresare

Operanții instrucțiunilor pot fi specificați sub forme numite **moduri de adresare**.

Cele trei tipuri de operanți sunt: **operanți imediați, operanți registru și operanți în memorie**.

Valoarea operanților este calculată în momentul asamblării pentru operanții imediați, în momentul încărcării programului pentru adresarea directă (adresa FAR) și în momentul execuției pentru operanții registru și și pt ei adresati în mod direct, (nume de variabila) cei adresăți indirect.

#### 3.2.1.1. Utilizarea operanților imediați

*Operanții imediați* sunt formați din date numerice constante calculabile la momentul asamblării.

Constantele întregi se specifică prin valori binare, octale, zecimale sau hexazecimale. Adițional, este permisă folosirea caracterului \_ (underscore) pentru a separa grupuri de cifre. Baza de numerație poate fi precizată în mai multe moduri:

- Folosind suficele H sau X pentru hexazecimal, D sau T pentru zecimal, Q sau O pentru octal și B sau Y pentru binar; în aceste cazuri numărul trebuie să înceapă obligatoriu cu o cifră între 0 și 9, pentru a nu exista confuzii între constante și simboluri, de exemplu, OABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol
- În stil C, prin prefixare cu 0x sau 0h pentru hexazecimal, 0d sau 0t pentru zecimal, 0o sau 0q pentru octal, respectiv 0b sau 0y pentru binar;

Exemple:

- constanta hexazecimală B2A poate fi exprimată ca 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH, etc;
- valoarea zecimală 123 poate fi specificată ca 123, 0d123, 0d0123, 123d, 123D, ...
- 11001000b, 0b11001000, 0y1100\_1000, 001100\_1000Y reprezintă diferite exprimări ale numărului binar 11001000

**Deplasamentele etichetelor de date și de cod** reprezinta valori determinabile la momentul asamblării care rămân constante pe tot parcursul execuției programului

mov eax, et ; transfer în registrul EAX a adresei (offsetului) asociate etichetei et  
va putea fi evaluată la momentul asamblării drept de exemplu

mov eax, 8 ; distanță de 8 octeți față de începutul segmentului de date

“Constanța” acestor valori derivă din regulile de alocare adoptate de limbajele de programare în general și care statuează că ordinea de alocare în memorie a variabilelor declarate (mai precis distanța față de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanțele salturilor destinație în cazul unor instrucțiuni de tip **goto** sunt valori constante pe parcursul execuției unui program.

Adică: o variabilă odată alocată în cadrul unui segment de memorie nu își va schimba niciodată locul alocării (adică poziția sa față de începutul aceluia segment) iar această informație determinabilă la momentul asamblării derivă din ordinea specificării variabilelor la declarare în cadrul textului sursă și din dimensiunea de reprezentare dedusă pe bază informației de tip asociate.

### 3.2.1.2. Utilizarea operanzilor registru

Modul de *invocare/accesare directă* - **mov eax, ebx**

Invocare/accesare *indirectă* - pentru a indica locațiile de memorie - **mov eax, [ebx]**

### 3.2.1.3. Utilizarea operanzilor din memorie

Operanzi din memorie : cu *adresare directă* și cu *adresare indirectă*.

Operandul cu *adresare directă* este o constantă sau un simbol care reprezintă adresa (segment și deplasament) unei instrucțiuni sau a unor date. Acești operanzi pot fi *etichete* (de ex: jmp et, var1 dw 324, add eax,[b]), *nume de proceduri* (de ex: call proc1) sau *valoarea contorului de locații* (de ex: b db \$-a).

**Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării** (assembly time). Adresa fiecărui operand raportată la structura programului executabil (mai precis stabilirea segmentelor la care se raportează deplasamentele calculate) este calculată în momentul editării de legături (linking time). Adresa fizică efectivă este calculată în momentul încărcării programului pentru execuție (loading time – acest proces final de ajustare a adreselor numindu-se RELOCAREA ADRESELOR = Address Relocation).

Un deplasament utilizat ca operand în cadrul unui program este întotdeauna raportat la un registru de segment. Acest registru poate fi specificat explicit sau, în caz contrar, se asociază de către asamblator în mod implicit un registru de segment. Regulile limbajului de asamblare pentru asocierea implicită sunt:

- **CS** pentru etichete de cod destinație ale unor salturi (jmp, call, ret, jz etc);
- **SS** în adresări SIB ce folosesc EBP sau ESP drept bază (indiferent de index sau scală);
- **DS** pentru restul accesărilor de date

Specificarea explicită a unui registru de segment se face cu ajutorul operatorului de prefixare segment (notat ":" și care se mai numește, 'operatorul de specificare a segmentului'). ES poate fi utilizat numai în specificări expuse (ca de exemplu ES:[Var] sau ES:[ebx+eax\*2-a]) sau în cadrul unor instrucțiuni pe siruri (MOVSB)

JMP FAR CS:...

JMP FAR DS:... or JMP FAR [label2]

## Curs 6

*Reguli pentru asociere implicită sunt:*

- **CS** pentru etichete de cod destinație ale unor salturi (jmp, call, ret, jz etc);
- **SS** în adresări SIB ce foloseste EBP sau ESP drept bază (indiferent de index sau scală);
- **DS** pentru restul accesărilor de date

### Exemple ilustrând regulile implicite de prefixare ale unui offset cu segmentul aferent

Mov eax, [v] ; mov eax, DWORD PTR DS:[405000]

Mov eax, [cbx+ebp] ; EAX  $\leftarrow$  ...DS:...

Mov eax, [ebx] ; mov eax, DWORD PTR DS:[ebx]

Mov eax, [cbx\*2+ebp] ; ....SS:....

Mov eax, [ebp] ; mov eax, DWORD PTR SS:[ebp]

Mov eax, [ebp\*2] ; mov eax, DWORD PTR SS:[ebp+ebp]

Mov eax, [cbx\*1+ebp] ; ... SS:....

Mov eax, [ebp\*3] ; mov eax, DWORD PTR SS:[ebp+ebp\*2]

Mov eax, [ebp\*4] ; mov eax, DWORD PTR DS:[ebp\*4]

Mov eax, [cbp\*1+cbx] ; ...DS:....

Mov eax, [cbx+esp] ; eax  $\leftarrow$  dword care incepe la adresa [SS:esp+ebx]

Mov eax, [esp + ebx] ; eax  $\leftarrow$  dword care incepe la adresa [SS:esp+ebx]

Mov eax, [ebx+esp\*2] ; syntax error – ESP nu poate fi index !

Mov eax, [cbx+ebp\*2] ; eax  $\leftarrow$  dword care incepe la adresa [DS:ebx + 2\*ebp]

**Mov eax, [cbx\*1+ebp\*1] ; SS... ;** Primul gasit cu \* e considerat index ! EBP - baza

**Mov eax, [ebp\*1+ebx\*1] ; DS....;** Primul gasit cu \* e considerat index! EBX - baza

**Mov eax, [ebp\*1+ebx\*2]; ...SS...**

*Adresa oricărui variabilă alocată într-un program este fixă pe tot parcursul execuției, iar offset-ul său este întotdeauna o constantă determinabilă la momentul asamblării.*

### 3.2.1.4. Operanzi cu adresare indirectă

Operanzii cu *adresare indirectă* utilizează registri pentru a indica adrese din memorie. Deoarece valorile din registri se pot modifica la momentul execuției, adresarea indirectă este indicată pentru a opera în mod dinamic asupra datelor.

Forma generală pentru accesarea indirectă a unui operand de memorie este dată de formula de calcul a offset-ului unui operand:

$$[ \text{registru\_de\_bază} ] + [ \text{registru\_index} * \text{scală} ] + [ \text{constantă} ]$$

*Constantă* este o expresie a cărei valoare este determinabilă la momentul asamblării. De exemplu,  $[ebx + edi + table + 6]$  desemnează un operand prin adresare indirectă, unde atât *table* cât și 6 sunt constante.

Operanzii *registru\_de\_bază* și *registru\_index* sunt folosiți de obicei pentru a indica o adresă de memorie referitoare la un tablou. În combinație cu factorul de scalare, mecanismul este suficient de flexibil pentru a permite acces direct la elementele unui tablou de înregistrări, cu condiția ca dimensiunea în octeți a unei înregistrări să fie 1, 2, 4 sau 8. De exemplu, octetul superior al elementului de tip DWORD cu index dat în ecx, parte a unui vector de înregistrări al cărui adresă (a vectorului) este în edx poate fi încărcat în dh prin intermediul instrucțiunii

```
mov dh, [edx + ecx * 4 + 3]
```

Din punct de vedere sintactic, atunci când operandul nu este specificat prin formula completă, lipsind unele dintre componente (de exemplu lipsește “\* scală”), asamblorul va rezolva ambiguitatea care rezultă printr-un proces de analiză a tuturor formele echivalente de codificare posibile și alegerea celei mai scurte dintre acestea. Cu alte cuvinte, având

```
push dword [eax + ebx] ; salvează pe stivă dublucuvântul de la adresa eax+ebx
```

asamblorul are libertatea de a considera eax drept bază și ebx drept index sau invers, ebx drept bază și eax drept index. Analog, pentru

```
pop DWORD [ecx] ; restaurează vârful stivei în variabila cu adresa dată de ecx
```

asamblorul poate interpreta ecx fie ca bază fie ca index. Ce este realmente important de reținut este faptul că toate codificările luate în considerare de către asamblor sunt echivalente iar decizia finală a asamblorului nu are impact asupra funcționalității codului rezultat.

De asemenea, în plus față de rezolvarea unor astfel de ambiguități, asamblorul permite și exprimări non-standard cu condiția ca acestea să fie transformabile într-un final în forma standard de mai sus.

Alte exemple:

```
lea eax, [eax*2] ; încarcă în eax valoarea lui eax*2 (adică, eax devine 2*eax)
```

În acest caz, asamblorul poate decide între codificare de tip bază = eax + index = eax și scală = 1 sau index = eax și scală = 2.

```
lea eax, [eax*9 + 12] ; eax ia valoarea eax * 9 + 12
```

Deși scală nu poate fi 9, asamblorul nu va emite aici un mesaj de eroare. Aceasta deoarece el va observa posibila codificare a adresei drept: bază=eax + index=eax cu scală=8, unde de această dată valoarea 8 este corectă pentru scală. Evident, instrucțiunea putea fi precizată mai clar sub forma

```
lea eax, [eax + eax * 8 + 12].
```

Să reținem deci că pentru adresarea indirectă, esențială este specificarea între paranteze drepte a cel puțin unuia dintre elementele componente ale formulei de calcul a offsetului.

### 3.2.2. Utilizarea operatorilor

Operatori - pentru combinarea, compararea, modificarea și analiza operanzilor. Unii operatori lucrează cu constante întregi, alții cu valori întregi memorate, iar alții cu ambele tipuri de operanzi.

Este importantă înțelegerea diferenței dintre operatori și instrucțiuni. **Operatorii efectuează calcule cu valori constante SCALARE determinabile la momentul asamblării** (valori scalare = valori immediate), **cu excepția adunării și scaderii unei constantă la un/dintr-un pointer** (care va furniza intotdeauna “pointer data type”) **și cu excepția formulei de calcul al offset-ului unui operand** (care permite operatorul ‘+’). Instrucțiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuției. Operatorul de adunare (+) efectuează adunarea în momentul asamblării; instrucțiunea ADD efectuează adunarea în timpul execuției.

Operatorii disponibili pentru construcția expresiilor sunt asemănători celor din limbajul C, atât ca sintaxă cât și din punct de vedere semantic. **Evaluarea expresiilor numerice se face pe 64 de biți**, rezultatele finale fiind ulterior ajustate în conformitate cu dimensiunea de reprezentare disponibilă în contextul de utilizare al expresiei.

În tabelul de mai jos sunt prezentati în ordinea priorității operatorii ce pot fi folosiți în cadrul expresiilor limbajului de asamblare x86 (NASM !!).

Prioritate	Operator	Tip	Rezultat
7	-	Unar, prefixat	Complement față de 2 (negare): $-X = 0 - X$
7	+	Unar, prefixat	Fără efect (oferit pentru simetrie cu „-”): $+X = X$
7	~	Unar, prefixat	Complement față de 1: $\text{mov al}, \sim 0 \Rightarrow \text{mov AL}, 0xFF$
7	!	Unar, prefixat	Negare logică: $!X = 0$ când $X \neq 0$ , altfel 1
6	*	Binar, infix	Înmulțire: $1 * 2 * 3 = 6$
6	/	Binar, infix	Câțul împărțirii fără semn: $24 / 4 / 2 = 3$ ( $-24/4/2 = 0FDh$ )
6	//	Binar, infix	Câțul împărțirii cu semn: $-24 // 4 // 2 = -3$ ( $-24 / 4 / 2 \neq -3!$ )
6	%	Binar, infix	Restul împărțirii fără semn: $123 \% 100 \% 5 = 3$
6	%%	Binar, infix	Restul împărțirii cu semn: $-123 \% \% 100 \% \% 5 = -3$
5	+	Binar, infix	Însumare: $1 + 2 = 3$
5	-	Binar, infix	Scădere: $1 - 2 = -1$
4	<<	Binar, infix	Deplasare pe biți către stânga: $1 << 4 = 16$
4	>>	Binar, infix	Deplasare pe biți la dreapta: $0xFE >> 4 = 0x0F$
3	&	Binar, infix	ȘI: $0xF00F \& 0xFF6 = 0x0006$
2	^	Binar, infix	SAU exclusiv: $0xFF0F ^ 0xFFFF = 0x0FF0$
1		Binar, infix	SAU: $1   2 = 3$

Operatorul de indexare are o utilizare largă în specificarea operanzilor din memorie adresați indirect. Paragraful 3.2.1 a clarificat rolul operatorului [] în adresarea indirectă.

Acest tabel este de o importanță covârșitoare în anumite situații și e bine să îl avem „la îndemână”. Iată de ce:

**5|6+7&8 = (5|6)+(7&8) = 7+0 = 7 ?? NU !!!!!!! datorită precedenței operatorilor avem;**

**5|6+7&8 = 5|(6+7)&8 = 5|13&8 = 5|8=13 = 0Dh !!!**

### 3.2.2.3. Operatori de deplasare de biți

*expresie >> cu\_cât și expresie << cu\_cât*

mov ah, 01110111b << 3 ; desemnează valoarea 10111000b  
add bh, 01110111b >> 3 ; desemnează valoarea 00001110b

AX	00000011 10111000	!!!!
BX	000000000 00001110b	

### 3.2.2.4. Operatori logici pe biți

*Operatorii pe biți* efectuează operații logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante.

OPERATOR	SINTAXA	SEMNIFICAȚIE
~	~ expresie	complementare biți
&	expr1 & expr2	ȘI bit cu bit
	expr1   expr2	SAU bit cu bit
^	expr1 ^ expr2	SAU exclusiv bit cu bit

Instrucțiuni aproxiimatice

not

and

or

xor

Exemple (presupunem că expresia se reprezintă pe un octet):

~ 11110000b ; desemnează valoarea 00001111b ; ~0f0h = ~...  
 01010101b & 11110000b ; are ca rezultat valoarea 01010000b  
 01010101b | 11110000b ; are ca rezultat valoarea 11110101b  
 01010101b ^ 11110000b ; are ca rezultat valoarea 10100101b  
 ! – negare logică (similar cu limbajul C) ; !0 = 1 ; !(orice diferit de zero) = 0

### 3.2.2.6. Operatorul de specificare a segmentului

*Operatorul de specificare a segmentului* (:) comandă calcularea adresei FAR a unei variabile sau etichete în funcție de un anumit segment. Sintaxa este: **segment:expresie**

[ss: ebx+4] ; deplasamentul e relativ la SS // [es:082h] ; deplasamentul e relativ la ES  
 10h:var ; segmentul este indicat de selectorul 10h, iar offsetul este valoarea etichetei var.

### 3.2.2.7. Operatori de tip

Specifică tipurile unor expresii și a unor operanzi păstrați în memorie. Sintaxa pentru aceștia este

**tip expresie**

unde specificatorul de tip este unul dintre cuvintele cheie **BYTE, WORD, DWORD, QWORD**.

Această construcție sintactică forțează ca *expresie* să fie tratată ca având dimensiunea de reprezentare *tip*, fără însă a-i modifica definitiv (destructiv) valoarea în sensul precizat de conversia dorită. De aceea, aceștia sunt considerați **operatori de conversie (temporară) nedestructivă**. Pentru operanzzii păstrați în memorie, *tip* poate fi **BYTE, WORD, DWORD, QWORD** având dimensiunile de reprezentare 1, 2, 4, 8 octeți. Pentru etichetele de cod el poate fi **NEAR** (adresă pe 4 octeți) sau **FAR** (adresă pe 6 octeți). Expressia **byte** [A] va indica doar primul octet de la adresa indicată de A. Analog, **dword** [A] indică dublucuvântul ce începe la adresa A.

**Specificatorii BYTE / WORD / DWORD / QWORD au intotdeauna doar rol de a clarifica o ambiguitate (inclusiv cand este vorba despre o variabilă de memorie, faptul de a preciza **mov BYTE [v], 0** sau **mov WORD [v], 0** este tot o clarificare a ambiguității, cum nasm nu asociază faptul ca v este byte/ word / dword).**

**mov [v],0** ; syntax error – operation size not specified

**Specificatorul QWORD nu intervene niciodată explicit** în cod pe 32 de biți.

Exemple unde e necesar un specificator de dimensiune al operanzilor:

- **mov [mem], 12**

- **(i)div [mem] ; (i)mul [mem]**

- **push [mem] ; pop [mem]**

- **push 15** - aici este o inconsistență în NASM, asamblorul nu va emite eroare/warning ci va face
- **push DWORD 15**

Exemple de operanzi IMPLICITI efectiv pe 64 biți (în cod pe 32):

- **mul dword [v]** ; înmulțește eax cu dword-ul de la adresa v și depune în EDX:EAX rezultatul
- **div dword [v]** ; împărțire EDX:EAX la v

### 3.3. DIRECTIVE

Directivele indică modul în care sunt generate codul și datele în momentul asamblării.

#### 3.3.1.1. Directiva SEGMENT

Directiva SEGMENT permite direcționarea octetilor de cod sau date emisi de către un asamblor înspre segmentul precizat, segment care poartă un nume și are asociate diverse caracteristici.

**SEGMENT nume [tip] [ALIGN=aliniere] [combinare] [utilizare] [CLASS=clasă]**

Numelui segmentului i se asociază ca valoare adresa de segment (32 biți) corespunzătoare pozitiei segmentului în memorie în faza de execuție. În acest sens, asamblorul NASM pune la dispoziție și simbolul special \$\$ care este echivalent cu adresa segmentului curent, acesta având însă avantajul că poate fi utilizat în orice context, fără a fi necesar să fie cunoscut numele segmentului în care ne aflăm.

Cu excepția numelui, toate celelalte câmpuri sunt opționale atât din punct de vedere a prezenței cât și a ordinii în care sunt specificate.

Argumentele opționale *tip*, *aliniere*, *combinare*, *utilizare* și 'clasa' dau editorului de legături și asamblorului indicații referitoare la modul de încărcare și atributele segmentelor.

Tip permite selectarea unui model de folosire al segmentului, având la dispoziție următoarele opțiuni:

- **code** (sau **text**) - segmentul va contine cod, conținutul nu poate fi scris dar se poate citi sau executa
- **data** (sau **bss**) - segment de date permitând citire și scriere însă nu și execuție (valoare implicită)
- **rdata** - segment din care se poate doar citi, menit a conține definiții de date constante

Argumentul opțional *aliniere* specifică multiplul numărului de octeți la care trebuie să înceapă segmentul respectiv. Alinierile acceptate sunt puterii a lui 2, între 1 și 4096.

Dacă argumentul *aliniere* lipsește, atunci se consideră implicit că este vorba despre o alinieră ALIGN=1, adică segmentul poate începe la orice adresă.

Argumentul opțional *combinare* controlează modul în care segmente cu același nume din cadrul altor module vor fi combinate cu segmentul în cauză la momentul editării de legături. Valorile posibile sunt:

- **PUBLIC** - indică editorului de legături să concateneze acest segment cu alte eventuale segmente cu același nume, obținându-se un unic segment a cărei lungime este suma lungimilor segmentelor componente.
- **COMMON** - specifică faptul că începutul acestui segment trebuie să se suprapună peste începutul tuturor segmentelor ce au același nume. Se obține un segment având dimensiunea egală cu cea a celui mai mare segment având același nume.
- **PRIVATE** - indică editorului de legături că acest segment nu este permis a fi combinat cu celelalte care poartă același nume.
- **STACK** - segmentele cu același nume vor fi concatenate. În faza de execuție segmentul rezultat va fi segmentul stivă.

Implicit, dacă nu se specifică o metodă de combinare, orice segment este considerat PUBLIC.

Argumentul *utilizare* permite optarea pentru altă dimensiune de cuvânt decât cea de 16 biți, care este implicită în lipsa precizării acestui argument.

Argumentul 'clasa' are rolul de a permite stabilirea ordinii în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeași clasă vor fi plasate într-un bloc contigu de memorie indiferent de

ordinea lor în cadrul codului sursă. Nu există o valoare implicită de inițializare pentru acest argument, el fiind nedefinit în lipsa specificării, ducând în consecință la evitarea concatenării într-un bloc continuu a tuturor segmentelor definite astfel.

```
segment code use32 class=CODE      segment data use32 class=DATA
```

### 3.3.2. Directive pentru definirea datelor

*definire date = declarare* (specificarea atributelor) + *alocare* (rezervarea sp. de memorie necesar).  
 (UNICA !!!) (NU e unica !!!) (UNICA !!!)

In C – 17 module (fisiere separate) ; A1- variabilă globală (int A1 + 16 declarații de data extern int A1)  
 LINKER-ul este responsabil pt verificarea DEPENDENTELOR dintre module !

Structura unei Variable = ([nume], set\_de\_atribute, [adresa/referință, valoare])

Variabilele dinamice NU AU NUME !!!!

P=new(...); p=malloc(...); ...free... (Diferența between POINTER and DYNAMIC variables !!!!)

(Nume, set\_de\_atribute) = parametrii formalii ai funcțiilor și procedurilor !!!

Set\_de\_atribute = (TD, Domeniu\_de\_vizibilitate (scope), durata de viață (lifetime/extent), clasa de memorie)  
 Memory class (in C) = (auto, register, static, extern)

*tipul de date = dimensiunea de reprezentare* – octet, cuvânt, dublucuvânt, quadword

Forma generală a unei linii sursă în cazul unei declarații de date este:

[nume] tip\_data lista\_expresii ;comentariu  
 sau [nume] tip\_alocare factor ;comentariu

sau [nume] TIMES factor tip\_data lista\_expresii ;comentariu

unde *nume* este o etichetă prin care va fi referită data. Tipul rezultă din tipul datei (dimensiunea de reprezentare) iar valoarea este adresa la care se va găsi în memorie primul octet rezervat pentru data etichetată cu numele respectiv.

*factor* este un număr care indică de câte ori se repetă lista de expresii care urmează în paranteză.

*Tip\_data* este o directive de definire a datelor, una din următoarele:

DB - date de tip octet (BYTE)

DW - date de tip cuvânt (WORD)

DD - date de tip dublucuvânt (DWORD)

DQ - date de tip 8 octeți (QWORD - 64 biți)

DT - date de tip 10 octeți (TWORD - utilizate pentru memorarea constantelor BCD sau constantelor reale de precizie extinsă)

De exemplu, secvența următoare definește și inițializează cinci variabile de memorie:

```
segment data
  var1 DB  'd' ;1 octet
        .a DW  101b ;2 octeți
  var2 DD  2bfh ;4 octeți
        .a DQ  3070 ;8 octeți (1 quadword)
        .b DT  100  ;10 octeți
```

NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing DW ? or similar things: this is what it does instead. The operand to a RESB-type pseudo-instruction is a **critical expression** (toți operanții care intervin în calcul trebuie să fie cunoscute în momentul în care expresia este întâlnită). Ex:

```
buffer:    resb  64 ; reserve 64 bytes
wordvar:   resw  1  ; reserve a word
realarray: resq  10 ; array of ten reals
```

**Directiva TIMES** permite asamblarea repetată a unei instrucțiuni sau definiții de date:

**TIMES** *factor tip\_data expresie*

De exemplu                                      Tabchar TIMES 80 DB 'a'

crează un tablou de 80 de octeți inițializați fiecare cu codul ASCII al caracterului 'a'.

```
matrice10x10                    times 10*10 dd 0
```

va furniza 100 de dublucuvinte dispuse continuu în memorie începând de la adresa asociată etichetei matrice10x10.

TIMES can also be applied to instructions:

**TIMES** *factor instrucțiune*

```
TIMES 32 add eax, edx ; having as effect EAX = EAX + 32*EDX
```

### 3.3.3. **Directiva EQU**

**Directiva EQU** permite atribuirea, în faza de asamblare, unei valori numerice sau sir de caractere unei etichete fără alocarea de spațiu de memorie sau generare de octeți. Sintaxa directivei EQU este

*nume EQU expresie*

Exemplu:

```
END_OF_DATA        EQU '!'
BUFFER_SIZE       EQU 1000h
INDEX_START       EQU (1000/4 + 2)
VAR_CICLARE       EQU i
```

Prin utilizarea de astfel de echivalări textul sursă poate deveni mai lizibil. Se observă asemănarea etichetelor echivalente prin directiva EQU cu constantele din limbajele de programare de nivel înalt.

Expresia pentru echivalarea unei etichete definite prin directiva EQU poate conține la rândul ei etichete definite prin EQU:

```
TABLE_OFFSET      EQU 1000h
INDEX_START       EQU (TABLE_OFFSET + 2)
DICTIONAR_STAR    EQU (TABLE_OFFSET + 100h)
```

## Curs 7

### Operări și operatori pe biti

Operatorii și instrucțiunile noastre sunt la fel:

`mov AH, 0111011b < 3 ; AH: 10111000b`

VS

`mov AH, 0111011b`

`shl AH, 3`

**B - operatorul și bit cu bit**

**AND - instrucție**

!  $X \text{ AND } 0 = 0 \rightarrow$  înlocușă valoarea anumitor biti la valoarea 0

$X \text{ AND } 1 = X$

$X \text{ AND } X = X$

$X \text{ AND } \bar{X} = 0$

**I - operatorul SAV bit cu bit**

**OR - instrucție**

$X \text{ OR } 0 = X$

!  $X \text{ OR } 1 = 1 \rightarrow$  înlocușă valoarea anumitor biti la valoarea 1

$X \text{ OR } X = X$

$X \text{ OR } \bar{X} = 1$

**N - operatorul SAV EXCLUSIV bit cu bit**

**XOR - instrucție**

$X \text{ XOR } 0 = X$

$X \text{ XOR } 1 = \bar{X} \rightarrow$  operatie utilă pt complementarea valorii anumitor biti

$X \text{ XOR } X = 0$

$X \text{ XOR } \bar{X} = 1$

!  $\text{XOR AH, AH} ; AH = 0$

**Utilizarea operatorilor ! și ~**

**!** - negare logică  $!x = \begin{cases} 0, & x \neq 0 \\ 1, & \text{altfel} \end{cases}$

**~** - complementul față de 1: `mov AL, ~0  $\Rightarrow$  mov AL, OFFh`

## Lexempe

a d? 1# /2# /...

b d? -3 /...

mov eax, ! [a] ; expression **syntax error** pt că [a] nu este o constantă determinabilă la momentul asamblării  
 mov eax, [!a] ; ! can only be applied to scalar values , a - pointer  
 mov eax, ! a ; \_\_\_\_\_ || \_\_\_\_\_  
 mov eax, !(a + #) ; \_\_\_\_\_ || \_\_\_\_\_  
 mov eax, !(b - a) ; a,b pointers, dan b-a scalar  
 mov eax, ! [a + #] ; expression **syntax error**  
 mov eax, ! # ; EAX = 0  
 mov ah, # ; # = 00000111b deci # = 11111000b = F8h  
 mov eax, ! 0 ; eax = 1  
 mov eax, ! ebx ; **syntax error** → val medeterminabilă la momentul asamblării  
 aa equ 2 { OK  
 mov ah, ! aa  
 mov ah, #^ (n #) ; AH = 11111111b = OFFh  
 mov ax, value ^ n value ; AX = 0FFFh

## Operatori de tip și tipuri de date asociate operandelor

Operatorii efectuează calcule cu valori constante **scalare** determinabile la momentul asamblării (valori scalare = valori immediate) cu excepția adunării și scăderii unei constante la un/dintre-un pointer (care va furniza întotdeauna „pointer data type”) cu excepția formulei de calcul al offset-ului unui operand (care permite operatorul +)

Specificatorii **BYTE/WORD/DWORD/QWORD** au rolul de a clarifica o ambiguitate

v d?

a d?

b d?

push v , ok , se va pune offset-ul pe stivă pe 32 de biti

push [v] ; **syntax error (ambiguitate)** : operator size not specified → push pe stivă acceptă operanșe și de 32 și de 16 biti

push dword [v] ✓

push word [v] ✓

mov eax,[v] ; ok ✓ EAX = dword ptr [v] ; în Ollydbg , mov eax, dword ptr [DS:v]

push [eax] ; **syntax error** : Operation size not specified !

push byte [eax] ; **syntax error**

push word [eax] ; ok ✓

push 15 ; push DWORD 15

pop [v]; operation size not specified (a pop from the stack accepts both 16 and 32 bits value)

pop word / dword [v]; ok ✓

Pop v; syntaxa este pop destinatie; destinatie must be L-value

↳ R-value; acest pop este similar ca operatie cu 2:=3

(invalid combination of opcode and operands)

push op-sursă  
pop op-dest

pop dword b; syntax error

pop [eax]; operand size not specified

pop (d)word [eax]; ok ✓

pop 15; 15 is NOT a L-value syntax error

pop [15]; syntax error → operand size not specified

pop dword [15]; syntactic ok dar cel mai probabil run-time error pt că [DS:15] va provoca access violation

mov [v], 0, syntax error - operand size not specified

mov byte [v], 0; ok ✓

mov [v], byte 0; ok

div [v]; operand size?

div word [v]; ok ✓

imul [v+2]; operand size?

imul word [v+2]; DX:AX = AX \* word de la adresa v+2

a d?

b d?

mov a, b; syntax error pt că NU este L-value, ci R-value fără un offset determinabil ca și constantă la momentul asamblării

mov [a], b; syntax error - operand size not specified! bcs an offset is either a 16 bits value or a 32 bits value never a 8 bits value (the same effect as mov ah, v)

mov word [a], b; ok ✓ → 2 octetii inferiori din valoarea offset-ului lui b!

mov dword [a], b; full offset 32 bits

mov a, [b]; invalid combination of opcode and operands (a = R-value)

mov [a], [b]; invalid combination of opcode and operands (NU putem avea 2 operanze simultan din memorie)

mul v; syntax error MUL reg/mem not offset

mul word v; — — —

mul [v]; op. size not specified

mul dword [v]; ok ✓

mul eax ; ok ✓  
 mul [eax] ; operand size not specified  
 mul byte [eax] ; ok ✓  
 mul 15 ; invalid comb of opcode and operands - MUL reg/mem , not constantă  
 pop byte [v] ; invalid comb of opcode and operands  
 pop ~~word~~ [v] ; instruction not supported in 32 bit mode

## Clasificarea erorilor în informatică

- **Eroare de sintaxă** – ea este diagnosticată de asamblator/compilator ! (eroare de asamblare)
- **Run-time error (eroare la execuție)** – programul “crapă” – programul se oprește = program crash !!
- **Eroare logică** = programul funcționează până la capăt sau rămâne blocat în ciclu infinit, însă GRESIT dpu LOGIC obținându-se cu totul alte rezultate decât cele așteptate...
- **Fatal: Linking Error !!!** (de ex în cazul unei definiții duble de variabilă... 17 module și o variabilă trebuie să fie DEFINITĂ DOAR într-un singur modul ! Dacă ea este definită în 2 sau mai multe module se va obține Fatal: Linking Error !!! – Duplicate definition for symbol A1 !!!)