Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana
diana.sotropa@ubbcluj.ro

Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai

Elementele de bază ale limbajului de asamblare

Elementele de bază ale limbajului de asamblare

Utilizarea operatorilor

- Operatorii se folosesc pentru combinarea, compararea, modificarea şi analiza operanzilor
- Unii operatori lucrează cu constante întregi, alţii cu valori întregi memorate, iar alţii cu ambele tipuri de operanzi.
- **Operatorii efectuează calcule cu valori constante SCALARE determinabile la momentul asamblării (valori scalare = valori imediate), cu exceptia adunarii si scaderii unei constante la un/dintr-un pointer (care va furniza intotdeauna "pointer data type") si cu exceptia formulei de calcul al offset-ului unui operand (care permite operatorul '+').**
- Instrucţiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuţiei. Operatorul de adunare (+) efectuează adunarea în momentul asamblării; instrucţiunea ADD efectuează adunarea în timpul execuţiei.

Elementele de bază ale limbajului de asamblare

Utilizarea operatorilor

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

Elementele de bază ale limbajului de asamblare

Utilizarea operatorilor

Prioritate	Operator	Tip	Rezultat
7	-	Unar, prefixat	Complement faţă de 2 (negare): `-X = 0 - X`
7	+	Unar, prefixat	Fără efect (oferit pentru simetrie cu „-”): `+X = X`
7	~	Unar, prefixat	Complement faţă de 1: `mov AL, ~0 => mov AL, 0xFF`
7	!	Unar, prefixat	Negare logică: `!X = 0` când `X ≠ 0`, altfel `1`
6	*	Binar, infix	Înmulţire: `1 * 2 * 3 = 6`
6	/	Binar, infix	Câtul împărţirii fără semn: `24 / 4 / 2 = 3 (-24/4/2 = 0FDh)`
6	//	Binar, infix	Câtul împărţirii cu semn: `-24 // 4 // 2 = -3 (-24 / 4 / 2 ≠ -3!)`
6	%	Binar, infix	Restul împărţirii fără semn: `123 % 100 % 5 = 3`
6	%%	Binar, infix	Restul împărţirii cu semn: `-123 %% 100 %% 5 = -3`

Elementele de bază ale limbajului de asamblare

Utilizarea operatorilor

Prioritate	Operator	Tip	Rezultat
5	+	Binar, infix	Însumare: `1 + 2 = 3`
5	-	Binar, infix	Scădere: `1 - 2 = -1`
4	<<	Binar, infix	Deplasare pe biţi către stânga: `1 << 4 = 16`
4	>>	Binar, infix	Deplasare pe biţi la dreapta: `0xFE >> 4 = 0x0F`
3	&	Binar, infix	ŞI: `0xF00F & 0x0FF6 = 0x0006`
2	^	Binar, infix	SAU exclusiv: `0xFF0F ^ 0xF0FF = 0x0FF0`
1	|	Binar, infix	SAU: `1 | 2 = 3`

Elementele de bază ale limbajului de asamblare

Operatori de deplasare de biți

```
expresie >> cu_cât
expresie << cu_cât
```

```
mov AH, 01110111b << 3 ; AH = 10111000b
mov BH, 01110111b >> 3 ; BH = 00001110b

mov AX, 01110111b << 3 ; AX = 00000011 10111000b
mov BX, 01110111b >> 3 ; BX = 00000000 00001110b
```

Elementele de bază ale limbajului de asamblare

Operatori logici pe biţi

- Operatorii pe biţi efectuează operaţii logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante.

OPERATOR	SINTAXA	SEMNIFICAŢIE
~	~ expresie	complementare biţi
&	expr1 & expr2	ŞI bit cu bit
|	expr1 | expr2	SAU bit cu bit
^	expr1 ^ expr2	SAU exclusiv bit cu bit
!	!0 = 1, !x = 0, x≠0	Negare logică

Elementele de bază ale limbajului de asamblare

Operatori logici pe biţi

& - operatorul SI bit cu bit AND – instructiune	`x AND 0 = 0` `x AND 1 = x`	`x AND x = x` `x AND ~x = 0`
Operaţie utilă pt forţarea valorii anumitor biţi la 0 !!!!		
| - operatorul SAU bit cu bit OR – instrucţiune	`x OR 0 = x` `x OR 1 = 1`	`x OR x = x` `x OR ~x = 1`
Operaţie utilă pt forţarea valorii anumitor biţi la 1 !!!!		
^ - op. SAU EXCLUSIV bit cu bit; XOR – instrucţiune	`x XOR 0 = x` `x XOR 1 = ~x`	`x XOR x = 0` `x XOR ~x = 1`
Operaţie utilă pt complementarea valorii anumitor biţi !!!		
`XOR AX, AX ; AX=0 !!!`		
`! Negare logică: !X = 0 când X ≠ 0, altfel 1`		
`~ Complement faţă de 1: mov al, ~0 => mov AL, 0ffh`		

Elementele de bază ale limbajului de asamblare

Operatori logici pe biţi

```
~ 11110000b ; desemnează valoarea 00001111b
~ 0f0h ; desemnează valoarea 0fh
 01010101b & 11110000b ; are ca rezultat valoarea 01010000b
 01010101b | 11110000b ; are ca rezultat valoarea 11110101b
 01010101b ^ 11110000b ; are ca rezultat valoarea 10100101b
```

Elementele de bază ale limbajului de asamblare

Operatori logici pe biţi

`MOV EAX, ![a]`	; **syntax error** deoarece [a] NU este o constantă determinabilă la momentul asamblării
`MOV EAX, [!a]`	; ! Poate fi aplicat doar pe valori SCALARE !!!!! a = POINTER
`MOV EAX, !a`	; ! Poate fi aplicat doar pe valori SCALARE !!!!! a = POINTER
`MOV EAX, !(a+7)`	; ! Poate fi aplicat doar pe valori SCALARE !!!!! a(+7) = POINTER
`MOV EAX, !(b-a)`	; Ok! a,b – POINTER, dar b-a = SCALAR
`MOV EAX, ![a+7]`	; **syntax error**
`MOV EAX, !7`	; EAX = 0 ;
`MOV AH, ~7`	; 7 = 00000111b deci ~7 = 11111000b = f8h
`MOV EAX, ~7`	; EAX = -8 (FFFF FFF8h)
`MOV EAX, !ebx`	; **syntax error**
`aa equ 2` `MOV AH, !aa`	; AH = 0
`MOV AH, 17^(~17)`	; AH = 11111111b = 0ffh
`MOV AX, value ^ ~value`	; AX = 0ffffh

Elementele de bază ale limbajului de asamblare

Utilizarea operatorilor

```
5 | 6 + 7 & 8 = ( 5 | 6 ) + ( 7 & 8 ) = 7 + 0 = 7 ??
```

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

- Care dintre variantă e corectă?
  - A doua variantă, datorită precedenței operatorilor
    + (prioritate 5), apoi & (prioritate 3), apoi | (prioritate 1)

Elementele de bază ale limbajului de asamblare

Operații și operatori pe biți

- Atenţie la diferenţa dintre operatori şi instrucţiuni

```
MOV AH, 01110111b << 3 ; AH = 10111000b

Vs.

MOV AH, 01110111b
SHL AH, 3 ; AH = 10111000b
```

Elementele de bază ale limbajului de asamblare

Operatorul de tip

- Specifică tipurile unor expresii şi a unor operanzi păstraţi în memorie. Sintaxa pentru aceştia este `tip expresie` unde specificatorul de tip este unul dintre cuvintele cheie **BYTE, WORD, DWORD, QWORD.**
- Această construcţie sintactică forţează ca `expresie` să fie tratată ca având dimensiunea de reprezentare `tip`, fără însă a-i modifica definitiv (distructiv) valoarea în sensul precizat de conversia dorită.
- De aceea, aceştia sunt consideraţi **operatori de conversie (temporară) nedistructivă.**
- Pentru operanzii păstraţi în memorie, `tip` poate fi **BYTE, WORD, DWORD, QWORD** având dimensiunile de reprezentare 1, 2, 4, 8 octeţi.
- Pentru etichetele de cod el poate fi **NEAR** (adresă pe 4 octeţi) sau **FAR** (adresă pe 6 octeţi).

```
byte [A]; indica primul octet de la adresa indicată de A
dword [A]; indică dublucuvântul ce începe la adresa A
```

Elementele de bază ale limbajului de asamblare

Operatorul de tip

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

```
mov [v],0 ; syntax error - operation size not specified
```

- Specificatorul **QWORD** nu intervine niciodată explicit în cod pe 32 de biţi.

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 inconsistenţă în NASM, asamblorul nu va emite eroare/warning ci va face - push DWORD 15

Elementele de bază ale limbajului de asamblare

Operatorul de tip

```
v d? ...
a d? ...
b d? ...
```

PUSH v; ok, stack <- offset v (pe 32 biţi)

PUSH [v]; syntax error – Operation size not specified ! (*PUSH pe o stivă pe 32 biţi acceptă atât operanzi pe 32 biţi cât şi pe 16 biţi*)

PUSH dword [v]; ok

PUSH word [v]; ok

MOV eax,[v]; ok ; EAX = dword ptr [v], în Olly dbg "mov eax, dword ptr [DS:v]"

PUSH [eax]; syntax error…. Operation size not specified !

PUSH byte [eax]; syntax error….

PUSH word [eax]; ok

Elementele de bază ale limbajului de asamblare

Operatorul de tip

```
v d? ...
a d? ...
b d? ...
```

POP [v]; Op size not specified (a POP from the stack accepts both 16 and 32 bits values as stack operands)

POP (d)word [v]; ok

POP v; sintaxa este POP destinatie; destinatie must be a L-value! Dar v este R-value! Acest pop v este similar ca operatie cu 2:=3!(Invalid combination of opcode and operands)

POP dword b; syntax error

POP [EAX]; Op size not specified

POP (d)word [EAX]; ok

POP 15; 15 is NOT a L-value ! - syntax error

POP [15]; syntax error - Op size not specified

POP dword [15]; syntactic ok, cel mai probabil run-time error deoarece probabil [DS:15] va provoca Access violation

POP byte [v]; Invalid combination of opcode and operands

POP qword [v]; Instruction not supported in 32 bit mode !

Elementele de bază ale limbajului de asamblare

Operatorul de tip

```
v d? ...
a d? ...
b d? ...
```

MOV word [a], b; ok ! - 2 octeţi inferiori din valoarea offset-ului lui b !

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

MOV a, [b]; Invalid comb. of opcode and operands (adresa constanta , a = R-value)

MOV [a], [b]; Invalid comb. Of opcode and operands (NU putem avea 2 operanzi simultan din memorie)

MUL v; syntax error - MUL reg/mem

MUL word v; syntax error - MUL reg/mem

MUL [v]; op. size not specified

MUL dword [v]; ok !

MUL eax; ok !

MUL [EAX]; op. size not specified

MUL byte [EAX]; ok !

MUL 15; Invalid comb. of opcode and operands - MUL reg/mem

Elementele de bază ale limbajului de asamblare

Directive

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

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

- Directiva SEGMENT permite direcţionarea octeţilor de cod sau date emişi de către un asamblor înspre segmentul precizat, segment care poartă un nume şi are asociate diverse caracteristici.

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

- **Numelui segmentului** i se asociază ca valoare **adresa de segment** (32 biţi) corespunzătoare *poziţiei segmentului în memorie în faza de execuţie*.
- În acest sens, asamblorul NASM pune la dispoziţie şi 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.

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

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

- Cu excepţia numelui, toate celelalte câmpuri sunt opţionale atât din punct de vedere a prezenţei cât şi a ordinii în care sunt specificate.
- Argumentele opţionale *tip, aliniere, combinare, utilizare* şi *clasa* dau editorului de legături şi asamblorului indicaţii referitoare la modul de încărcare şi atributele segmentelor.
- **Tip** permite selectarea unui model de folosire al segmentului, având la dispoziție următoarele opțiuni:
  - **code** (sau text) - segmentul va conține cod, conținutul nu poate fi scris dar se poate citi sau executa
  - **data** (sau bss) - segment de date permițând citire și scriere însă nu și execuție (valoare implicită)
  - **rdata** - segment din care se poate doar citi, menit a conține definiții de date constante

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

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

- Argumentul opţional **aliniere** specifică multiplul numărului de octeţi la care trebuie să înceapă segmentul respectiv.
- Alinierile acceptate sunt puteri a lui 2, între 1 și 4096. Dacă argumentul aliniere lipseşte, atunci se consideră implicit că este vorba desprere o aliniere ALIGN=1, adică segmentul poate începe la orice adresă.

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

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

- Argumentul opţional **combinare** controlează modul în care segmente cu acelaşi 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 acelaşi nume, obţinându-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 acelaşi nume. Se obţine un segment având dimensiunea egală cu cea a celui mai mare segment având acelaşi nume.
  - **PRIVATE** - indică editorului de legături că acest segment nu este permis a fi combinat cu altele care poartă același nume.
  - **STACK** - segmentele cu acelaşi nume vor fi concatenate. În faza de execuţie segmentul rezultat va fi segmentul stivă.
- Implicit, dacă nu se specifică o metodă de combinare, orice segment este considerat PUBLIC.

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

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

- Argumentul **utilizare** permite optarea pentru altă dimensiune de cuvânt decât cea de 16 biți, care este implicită in 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 aceeaşi clasă vor fi plasate într-un bloc contiguu de memorie indiferent de ordinea lor în cadrul codului sursă.
- Nu există o valoare implicită de iniţializare pentru acest argument, el fiind nedefinit în lipsa specificării, ducând în consecinţă la evitarea concatenării într-un bloc continuu a tuturor segmentelor definite astfel.

Elementele de bază ale limbajului de asamblare

Directiva SEGMENT

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

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

Elementele de bază ale limbajului de asamblare

Directive pentru definirea datelor

definire date = **declarare** (specificarea atributelor) + **alocare** (rezervarea sp. de memorie necesar)

Unică !!!

NU e unică !!!

Unică !!!

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

Elementele de bază ale limbajului de asamblare

Directive pentru definirea datelor

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

```
[nume] tip_data lista_expresii [;comentariu]
[nume] tip_alocare factor [;comentariu]
[nume] TIMES factor tip_data lista_expresii [;comentariu]
```

- nume este o etichetă prin care va fi referită data
- tipul rezultă din tipul datei (dimensiunea de reprezentare)
- valoarea este adresa la care se va găsi în memorie primul octet rezervat pentru data etichetată cu numele rESPective
- factor este un număr care indică de câte ori se repetă lista de expresii care urmează în paranteză.
- tip_data este o directivă 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 octeţi (QWORD - 64 biţi)

Elementele de bază ale limbajului de asamblare

Directive pentru definirea datelor

```
segment data
        var1 DB 'd' ;1 octet
        .a DW 101b ;2 octeţi
        var2 DD 2bfh ;4 octeţi
        .a DQ 307o ;8 octeţi (1 quadword)
        vartest DW (1002/4+1)
        Tablou DW 1,2,3,4,5
        Tabpatrate DD 0, 1, 4, 9, 16, 25, 36
                   DD 49, 64, 81
                   DD 100, 121, 144, 169
```

Elementele de bază ale limbajului de asamblare

Directive pentru definirea datelor

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

```
[nume] tip_data lista_expresii [;comentariu]
[nume] tip_alocare factor [;comentariu]
[nume] TIMES factor tip_data lista_expresii [;comentariu]
```

- tip_alocare este o directivă de rezervare de date neiniţializate:
  - RESB - date de tip octet (BYTE)
  - RESW - date de tip cuvânt (WORD)
  - RESD - date de tip dublucuvânt (DWORD)
  - RESQ - date de tip 8 octeţi (QWORD - 64 biţi)
- factor este un număr care indică de câte ori se repetă tipul alocării precizate

```
segment data
        Tabzero RESW 100h; rezervă 256 de cuvinte
        buffer: resb 64 ; rezervă 64 octeți
        wordvar: resw 1 ; rezervă 1 cuvânt
        realarray resq 10 ; rezervă 10 quadword
```

Elementele de bază ale limbajului de asamblare

Directive pentru definirea datelor

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

```
[nume] tip_data lista_expresii [;comentariu]
[nume] tip_alocare factor [;comentariu]
[nume] TIMES factor tip_data lista_expresii [;comentariu]
```

- Directiva TIMES permite asamblarea repetată a unei instrucţiuni sau definiţii de dată:

```
segment data
        Tabchar TIMES 80 DB ‘a’; crează un tablou de 80 de octeţi
iniţializaţi 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 poate fi aplicat si pe instrucțiuni: `TIMES factor instrucţiune`

```
TIMES 32 add EAX, EDX ; are ca efect EAX = EAX + 32*EDX
```

Elementele de bază ale limbajului de asamblare

Directiva EQU

- Directiva EQU permite atribuirea, *în faza de asamblare*, unei valori numerice sau şir de caractere unei etichete fără alocarea de spaţiu de memorie sau generare de octeţi. Sintaxa directivei EQU este:

```
nume EQU expresie
```

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

- Expresia pentru echivalarea unei etichete definite prin directiva EQU poate conţine la rândul ei etichete definite prin EQU

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

Elementele de bază ale limbajului de asamblare

Contor de locații

```
Segment data
 a db 17, -2, 0ffh, 'xyz',…
```

lga db $-a	- scăderea a 2 pointeri = scalar (constanta numerică) - lga = variabila de memorie (mov [lga],…)
lga dw $-$$	ok !
lga EQU $-a	ok ! însa mov [lga],… este syntax error !!! pt ca lga NU este o variabilă alocata… ci o constantă! (fără alocare de memorie)
lga dw $-data	corect in TASM/MASM, INCORECT in NASM sub 32 biţi !!! syntax error – "Expression is not simple or relocatable"
lga dw lga-a	ok !
b EQU 27	ok! Atenţie b NU este un offset !
c dd 12345678h	ok !
lga dw b-a	syntax error ! b NU este un offset !
lga dw c-a	ok !
lga dw $-a-4	ok !
lg dw $-a	ok !

Elementele de bază ale limbajului de asamblare

Contor de locații

```
Segment data
 a db 1, 2, 3, 4 ; | 01h | 02h | 03h | 04h |
```

`lg db $-a`	`04h`
`lg db $-data`	`; syntax error – expression is not simple or relocatable (în TASM $-data=4 – deci acelaşi efect ca mai sus, deoarece în TASM, MASM offset(nume_segment)=0 !!!; în NASM NU, offset(data) = 00401000 !!!)`
`lg db a-data`	`; syntax error – expression is not simple or relocatable`
`lg2 dw data-a`	`; asamblorul ne lasă, însă obţinem eroare la link-editare – "Error in file at 00129 – Invalid fixup recordname count…."`
`db a-$`	`; = -5 = FBh`
`c equ a-$`	`; 0-6 = -6 = FAh`
`d equ a-$`	`; 0-6 = -6 = FAh`
`e db a-$`	`; 0-6 = -6 = FAh`
`x dw x`	`; 07h 10h !!!!!`
`x db x`	`; syntax error !`

Elementele de bază ale limbajului de asamblare

Contor de locații

```
Segment data
 a db 1, 2, 3, 4 ; | 01h | 02h | 03h | 04h |
```

`x1 dw x1`	`; x1 = offset(x1) 09 00 la asamblare si in final 09 10`
`db lg-a`	`; 04`
`db a-lg`	`; = -4 = FC`
`db [$-a]`	`; expression syntax error`
`db [lg-a]`	`; expression syntax error`
`lg1 EQU lg1`	`; lg1 = 0 NASM consideră că e corect ! (IT IS A NASM BUG !!!)`
`lg1 EQU lg1-a`	`; lg1 = 0 - DE CE ? Bug NASM ! Orice se evalueaza la zero în dreapta este acceptat chiar daca este definiţie recursivă ! Dacă a este primul element definit în segment consideră a=0 (offset-ul faţă de începutul segmentului) şi va furniza lg1=0; dacă a este definit altundeva (offset ≠ 0), atunci va furniza syntax error = "Macro abuse, recursive EQUs"`
`g34 dw c-2`	`; -8 = F8h`
`b dd a-start`	`; syntax error ! (a definit` **`aici`**`, start definit` **`altundeva`**`) expression is not simple or relocatable !`
`dd start-a`	`; ok !(start definit` **`altundeva`** `şi a definit` **`aici`**`) - rez=POINTER, deoarece etichetele NU fac parte din acelasi segment şi este interpretată ca scădere FAR = pointer`
`dd start-start1`	`; ok ! (ambele etichetele sunt definite în acelaşi segment) => rez=SCALAR pt că avem scădere de offset-uri definite în acelaşi segment`

Elementele de bază ale limbajului de asamblare

Contor de locații

```
segment code use32
 start:
        mov ah, lg1         ; AH = 0
        mov bh, c           ; BH = -6 = FAh
        mov ch, lg          ; OBJ format can handle only 16 or 32 byte
                              relocation (offset NU încape în 1 byte)
        mov ch, lg-a        ; CH = 04
        mov ch, [lg-a]      ; mov ch, byte ptr DS:[4] - Mem. access violation
        mov cx, lg-a        ; CX = 4
        mov cx, [lg-a]      ; mov WORD ptr DS:[4] - Mem. access violation
        mov cx, $-a         ; invalid operand type($ generat aici, a altundeva)
        mov cx, $$-a        ; invalid operand type($$ din code şi a din data)
        mov cx, a-$         ; OK (a definit altundeva şi $ generat aici)
        mov ch, $-a         ; invalid operand type($ generat aici, a altundeva)
        mov ch, a-$         ; OBJ format can handle only 16/32 byte relocation
                              a-$ e OK, dar a-$ = POINTER - syntax error
                             (pt că offset NU încape în 1 byte !!)
        mov cx, $-start     ; ok - pt că etichetele sunt din ac. seg.
        mov cx, start-$     ; ok - pt că etichetele sunt din ac. seg.
        mov ch, $-start     ; ok - pt ca REZ este scalar
        mov ch, start-$     ; ok
        mov cx, a-start     ; ok (a definit altundeva si start definit aici)
        mov cx, start-a     ; invalid operand type(start definit aici, a nu)
```

Elementele de bază ale limbajului de asamblare

Contor de locații

```
start1:
        mov ah, a+b       ; MERGE, DAR … NU ESTE ADUNARE DE POINTERI !
                            E adunare de scalar: a+b = (a-$$) + (b-$$)
        mov ax, b+a       ; AX = (b-$$) + (a-$$) - adunare de SCALARI !!!!
        mov ax, [a+b]     ; INVALID EFFECTIVE ADDRESS - ASTA CHIAR E ADUNARE
                            DE POINTERI, deci e INTERZISA - syntax error
        var1 dd a+b       ; syntax error ! - expression is not simple or
                            relocatable  (deci NASM nu permite ca "a+b" să
                            apară într-o definiţie de date ca expresie de
                            iniţializare, ci NUMAI ca OPERAND AL UNEI
                            INSTRUCTIUNI - cum se observă mai sus !)
```

Elementele de bază ale limbajului de asamblare

Contor de locații

- Concluzie:
  - Expresiile de tip `et1 - et2` (unde `et1` si `et2` sunt etichete – fie de cod, fie de date) sunt acceptate sintactic de catre NASM,
    - Fie dacă ambele sunt definite in acelaşi segment
    - Fie dacă `et1` aparţine unui segment diferit faţă de cel în care apare expresia, iar `et2` este definită în segmentul în care apare expresia. Intr-un astfel de caz, TD asociat expresiei `et1-et2` este POINTER şi NU SCALAR (constantă numerică) ca şi în cazul etichetelor ce fac parte din acelasi segment. (Deci ALTUNDEVA – AICI merge , însă AICI – ALTUNDEVA NU)
  - Scădere de offset-uri ce se raportează la acelaşi segment = SCALAR
  - Scădere de pointeri din segmente diferite = POINTER

Elementele de bază ale limbajului de asamblare

Contor de locații

- Concluzie:
  - Numele unui segment este asociat cu *"Adresa segmentului in memorie în faza de execuţie"* însă aceasta nu ne este disponibilă/accesibilă, fiind decisă la momentul încărcării programului pt execuţie de către încărcătorul de programe al SO.
  - De aceea, dacă folosim nume de segmente în expresii din program în mod explicit vom obţine fie **eroare de sintaxă** (în situaţii de tipul $/a-data - “AICI – ALTUNDEVA”) fie de **link-editare** (în situaţii de tipul data-a - ALTUNDEVA – AICI), deoarece practic numele unui segment este asociat cu adresa FAR al acestuia (adresă care nu va fi cunoscută decât la momentul încărcării programului, deci va fi disponibilă doar la run-time;
  - observăm astfel că `adresă_segment` este considerată ca “ALTUNDEVA”) şi NU este asociat cu *"Offset-ul segmentului în faza de asamblare, fiind o constantă determinată la momentul asamblării"* (aşa cum este în programarea sub 16 biţi de exemplu, unde nume_segment în faza de asamblare = offset-ul său = 0). Ca urmare, se constată că în progr. sub 32 de biţi numele de segmente NU pot fi folosite in expresii !!

Elementele de bază ale limbajului de asamblare

Contor de locații

```
Mov eax, data      ; Segment selector relocations are not supported in PE files (relocation
                     error occurred) - syntax error!
```

- Sub 32 biţi offset (data) = vezi OllyDbg – DATA segment începe la offset-ul 00401000, iar CODE segment la offset 00402000 (sau eventual invers, depinde de link-editor, versiunea de SO etc).
- Deci offset-urile începuturilor de segment nu sunt 0 la fel ca şi la programarea sub 16 biţi. Din acest punct de vedere sub 32 biţi este o mare diferenţă între numele de variabile (al căror offset poate fi determinat la asamblare) şi numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca şi o constantă, această valoare fiind cunoscută la fel ca şi adresa de segment doar la momentul încărcării programului pt execuţie - loading time).
- Dacă avem mai mulţi operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinaţie validă de operaţii cu pointeri:

```
Mov bx,[(v3-v2) ± (v1-v)]
```

Elementele de bază ale limbajului de asamblare

Contor de locații

```
Mov eax, data      ; Segment selector relocations are not supported in PE files (relocation
                     error occurred) - syntax error!
```

- Sub 32 biţi offset (data) = vezi OllyDbg – DATA segment începe la offset-ul 00401000, iar CODE segment la offset 00402000 (sau eventual invers, depinde de link-editor, versiunea de SO etc).
- Deci offset-urile începuturilor de segment nu sunt 0 la fel ca şi la programarea sub 16 biţi. Din acest punct de vedere sub 32 biţi este o mare diferenţă între numele de variabile (al căror offset poate fi determinat la asamblare) şi numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca şi o constantă, această valoare fiind cunoscută la fel ca şi adresa de segment doar la momentul încărcării programului pt execuţie - loading time).
- Dacă avem mai mulţi operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinaţie validă de operaţii cu pointeri:

```
Mov bx,[(v3-v2) ± (v1-v)]; OK - adunarea şi scăderea de valori imediate este corectă
```

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
UNIVERSITATEA BABEȘ-BOLYAI

Str. Mihail Kogălniceanu nr. 1
Cluj-Napoca, Cluj, România

www.cs.ubbcluj.ro