

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 1 –

1. Conversia numerelor între bazele de numerație 2, 10, 16
2. Reprezentarea numerelor întregi în memorie
3. Elementele limbajului de asamblare IA-32
4. Primul program în limbaj de asamblare

1. CONVERSIA NUMERELEOR ÎNTRU BAZELE DE NUMERAȚIE 2, 10, 16

Un sistem de numerație este constituit din *totalitatea regulilor de reprezentare a numerelor cu ajutorul unor simboluri denumite cifre*.

Pentru orice sistem de numerație, numărul de cifre ale sistemului este egal cu baza b :

- dacă $b = 2$ (sistemul de numerație binar) cifrele sunt 0 și 1;
- dacă $b = 10$ (sistemul de numerație zecimal) cifrele sunt 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- dacă $b = 16$ (sistemul de numerație hexazecimal) cifrele sunt 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Se observă că pentru numerele scrise într-o bază mai mare decât baza 10 (zecimal) se folosesc și alte simboluri (litere) pe lângă cifrele obișnuite din baza 10. Astfel, în cazul numerelor scrise în hexazecimal, literele A, B, C, D, E, F au ca și valori asociate 10, 11, 12, 13, 14, 15.

1.a. Conversia numerelor din baza 10 într-o bază oarecare

Se realizează prin *împărțirea succesivă* a numărului scris în baza 10 la baza în care se dorește conversia (se împarte numărul la bază, apoi se împarte câtul obținut la bază și.m.d. până când câtul obținut devine 0), după care se iau resturile obținute în ordine inversă, care constituie valoarea numărului în baza cerută.

Exemple:

1. Să se convertească numărul 347 din baza 10 în baza 16.

- împărțim succesiv numărul 347 la baza 16:

$$\begin{array}{r}
 347 \quad | \quad 16 \\
 -32 \quad | \quad 21 \quad | \quad 16 \\
 = 27 \quad | \quad -16 \quad | \quad 1 \quad | \quad 16 \\
 -16 \quad | \quad = 5 \quad | \quad -0 \quad | \quad 0 \\
 = 11 \quad | \quad \quad \quad | \quad = 1
 \end{array}$$

- luăm resturile obținute în ordine inversă și ținând cont că 11 reprezintă cifra B în baza 16, obținem:

$$347_{(10)} = 15B_{(16)}$$

2. Să se convertească numărul 57 din baza 10 în baza 2.

- împărțim succesiv numărul 57 la baza 2:

$$\begin{array}{r}
 57 \quad | \quad 2 \\
 -4 \quad | \quad 28 \quad | \quad 2 \\
 = 17 \quad | \quad -2 \quad | \quad 14 \quad | \quad 2 \\
 -16 \quad | \quad = 8 \quad | \quad -14 \quad | \quad 7 \quad | \quad 2 \\
 = 1 \quad | \quad -8 \quad | \quad = 0 \quad | \quad -6 \quad | \quad 1 \quad | \quad 2 \\
 = 0 \quad | \quad \quad \quad | \quad = 1 \quad | \quad -2 \quad | \quad 3 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad | \quad = 1 \quad | \quad -2 \quad | \quad 1 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad | \quad = 1 \quad | \quad -0 \quad | \quad 0 \quad | \quad 0 \\
 \quad \quad \quad \quad \quad \quad \quad | \quad = 1 \quad | \quad \quad \quad | \quad \quad \quad | \quad 0
 \end{array}$$

- luăm resturile obținute în ordine inversă, obținem:

$$57_{(10)} = 111001_{(2)}$$

1.b. Conversia numerelor dintr-o bază oarecare în baza 10

Dacă considerăm numărul N format din n cifre scrise baza b :

$$N_{(b)} = c_{n-1}c_{n-2}\dots c_2c_1c_0$$

atunci valoarea sa în baza 10 se obține astfel:

$$N_{(10)} = c_{n-1} * b^{n-1} + c_{n-2} * b^{n-2} + \dots + c_2 * b^2 + c_1 * b^1 + c_0 * b^0$$

Exemple:

1. Care e valoarea în baza 10 a numărului întreg hexazecimal $3A8_{(16)}$?

$$3A8_{(16)} = 3 * 16^2 + A * 16^1 + 8 * 16^0 = 3 * 256 + 10 * 16 + 8 * 1 = 936_{(10)}$$

2. Care e valoarea în baza 10 a numărului întreg binar $1101101_{(2)}$?

$$1101101_{(2)} = 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 64 + 1 * 32 + 1 * 8 + 1 * 4 + 1 = 109_{(10)}$$

1.c. Conversia rapidă între bazele de numerație 2 și 16

Putem realiza conversii rapide între numere scrise în bazele de numerație 2 și 16, ținând cont de faptul că *fiecărei cifre hexazecimale îi corespund 4 (patru) cifre binare și invers* (vezi tabelul de mai jos).

Valoare zecimală	Cifra hexazecimală	Numărul binar corespondent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

2. REPREZENTAREA NUMERELEOR ÎNTREGI ÎN MEMORIE

Reprezentarea unui număr se referă la exprimarea/configurația sa în baza 2 în memoria calculatorului.

În arhitectura IA-32 un număr întreg poate fi reprezentat pe 8, 16, 32 sau 64 de biți (1, 2, 4 sau 8 octeți).

Există 2 convenții de reprezentare: *convenția de reprezentare fără semn* și *convenția de reprezentare cu semn*. Astfel, într-o locație de memorie de n biți se află:

- fie un număr natural cuprins între 0 și $2^n - 1$, în *convenția de reprezentare fără semn*;
- fie un număr întreg cuprins între -2^{n-1} și $2^{n-1} - 1$, în *convenția de reprezentare cu semn*.

2.a. Bitul de semn

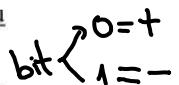
Dacă interpretăm o anumită configurație de biți ca întreg cu semn, atunci, prin convenție, pentru reprezentarea semnului unui număr se folosește un singur bit (numit *bit de semn*).

Bitul de semn = bitul superior (high) din octetul superior (high) al locației în care se reprezintă numărul.

Dacă valoarea acestui bit este 0, atunci numărul este POZITIV. Dacă valoarea acestui bit este 1, atunci numărul este NEGATIV.

2.b. Regula de reprezentare a numerelor întregi

Un număr întreg cuprins între $-2^n - 1$ și $2^{n-1} - 1$ se reprezintă într-o locație de n biți astfel:

bit 

- dacă numărul este pozitiv, atunci în locație se reprezintă numărul respectiv scris în baza 2;
- dacă numărul este negativ, atunci în locație se înscrie complementul în baza 2 a numărului.

2.c. Complementul unui număr întreg

De-a lungul timpului au existat mai multe modalități de reprezentare cu semn a unui număr întreg:

- cod direct: se reprezintă valoarea absolută a numărului pe $n-1$ biți din cei n ai locației, iar în bitul cel mai semnificativ să se pună semnul. Deși această soluție e foarte apropiată de cea naturală, s-a dovedit a fi mai puțin eficientă decât altele. O problemă ar fi faptul că în această reprezentare $-7 + 7 \neq 0$.
- cod invers (complement față de 1): se reprezintă valoarea absolută a numărului pe $n-1$ biți din cei n ai locației, iar în cazul în care numărul este negativ, se inversează toți cei n biți ai reprezentării. și la această reprezentare s-a renunțat, deoarece s-a dovedit neficientă (în plus, apare din nou problema $-7 + 7 \neq 0$).
- cod complementar (complement față de 2)

Numerele întregi cu semn se reprezintă în cod complementar (complementul față de 2).

Principalul avantaj oferit de codul complementar față de alte modalități de reprezentare este că bitul de semn participă la stabilirea valorii numărului.

Definiție Pentru complementarea unui număr întreg reprezentat pe n biți, mai întâi se inversează valorile tuturor bițiilor (valoarea 0 devine 1 și valoarea 1 devine 0) din locația de reprezentare, după care se adaugă 1 la valoarea obținută.

Exemplu:

Se dă numărul 18 în baza 10. Să se obțină complementul acestuia.

Procedăm astfel:

- convertim numărul dat în baza 2:

$$18_{(10)} = 10010_{(2)}$$

- calculăm complementul față de 2 al numărului dat conform definiției:

Reprezentarea inițială

$00010010 \rightarrow$ completăm până la 8 biți

$$\begin{array}{r} 18|2 \\ 18|9|2 \\ -0|8|4|2 \\ \hline -1|4|5|2 \\ -0|2|7|2 \\ =0|0|0|0 \\ \hline -1 \end{array}$$

$$18_{(10)} = 10010_{(2)}$$

După inversarea bițiilor

11101101

Se adună 1

11101101 +

00000001

Complementul față de 2

11101110

$$\begin{array}{r} 238+ \\ 18 \\ \hline 256 \end{array} \checkmark$$

Astfel, complementul numărului $18_{(10)} = 12_{(16)} = 00010010_{(2)}$ este $11101110_{(2)} = EE_{(16)} = 238_{(10)}$.

Reguli alternative de complementare

1. Se lasă neschimbați biții începând din dreapta reprezentării binare până la primul bit 1 inclusiv, iar restul bițiilor se inversează până la bitul $n-1$ inclusiv.
2. Se scade binar conținutul (evidenț binar) al locației de complementat din 100...00, unde numărul de după cifra binară 1 are atâtea zerouri câtă biți are locația de complementat.
3. Se scade hexazecimal conținutul (evidenț hexazecimal) al locației de complementat din 100...00, unde după cifra hexazecimală 1 apar atâtea zerouri câte cifre hexazecimale are locația de complementat.

Exemplu:

Se dă numărul 18 în baza 10. Să se obțină complementul acestuia.

Procedăm astfel:

- convertim numărul dat în baza 2:

$$18_{(10)} = 10010_{(2)}$$

- aplicăm regula de scădere binară (regula nr. 2):

$$\begin{array}{r} 100000000 - \\ 00010010 \\ \hline 11101110_{(2)} \end{array}$$

sau

- convertim numărul dat în baza 16:

$$18_{(10)} = 12_{(16)}$$

- aplicăm regula de scădere hexazecimală (regula nr. 3):

$$\frac{100 -}{\mathbf{EE}_{(16)}}$$

2.d. Dimensiune de reprezentare

Dimensiunea de reprezentare este numărul maxim de cifre binare (numărul de biți) din reprezentarea unui număr întreg.

2.e. Interval de reprezentare admisibilă

Reprezentarea se referă la baza 2, iar interpretările la baza 10, fiind întotdeauna posibile 2 interpretări pentru o reprezentare.

Intervalul de reprezentare admisibil este intervalul de valori în baza 10 care sunt posibil a fi reprezentate în baza 2 pe 1 OCTET, 1 CUVÂNT, 1 DUBLUCUVÂNT etc.

- Având în vedere că orice configurație binară are 2 interpretări posibile, unele dimensiuni de reprezentare a datelor în calculator sunt asociate cu mai multe tipuri de date.

Pentru valorile cele mai frecvent utilizate la nivelul programelor, acestea sunt:

[0, +255]	interval de reprezentare admisibil pentru „întreg <u>fără semn</u> reprezentat pe 1 octet”
[-128, +127]	interval de reprezentare admisibil pentru „întreg <u>cu semn</u> reprezentat pe 1 octet”
[0, +65535]	interval de reprezentare admisibil pentru „întreg <u>fără semn</u> reprezentat pe 2 octeți”
[-32768, +32767]	interval de reprezentare admisibil pentru „întreg <u>cu semn</u> reprezentat pe 2 octeți”

3. ELEMENTELE LIMBAJULUI DE ASAMBLARE IA-32

3.a. Tipuri fundamentale de date

În limbaj de asamblare, **tip de date** = **dimensiune de reprezentare**.

- **bit (binary digit)**: este unitatea elementară de informație
 - octet (byte): o succesiune de 8 biti, numerotată, prin convenție, astfel:

Octet (8 biți)

- cuvânt (word): o succesiune de 16 biți (2 octet), numerotată, prin convenție, astfel:

Cuvânt (16 biți)															
Octetul superior (high)								Octetul inferior (low)							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- dublucuvânt (doubleword); o succesiune de 32 biti (4 octeti), numerotată, prin convenție, astfel:

Dublucuvânt (32 biți)															
Cuvântul superior (16 biți)								Cuvântul inferior (16 biți)							
Octetul superior al cuvântului superior				Octetul inferior al cuvântului superior				Octetul superior al cuvântului inferior				Octetul inferior al cuvântului inferior			
31				24	23			16	15			8	7		0

- quadword: o succesiune de 64 biți (8 octeți), numerotată, prin convenție, astfel:

<i>Quadword (64 biți)</i>	
Dublucuvântul superior (32 biți)	Dublucuvântul inferior (32 biți)

Cuvântul superior al dublucuvântului superior	Cuvântul inferior al dublucuvântului superior	Cuvântul superior al dublucuvântului inferior	Cuvântul inferior al dublucuvântului inferior						
63	...	48 47	...	32 31	...	16	15	...	0

3.b. Directive pentru definirea datelor

- definire date = declarare (specificarea atributelor) + alocare (rezervarea spațiului de memorie necesar)
- tipul de dată = dimensiunea de reprezentare: octet, cuvânt, dublucuvânt, quadword
- definirea unei variabile (declarare + inițializarea valorii acesteia)

[nume] tip_data lista_expresii [;comentariu]

unde:

- **tip_data** este o directivă de definire a datelor, una din următoarele:
DB – date de tip octet (byte – 8 biți)
DW – date de tip cuvânt (word – 16 biți)
DD – date de tip dublucuvânt (dword – 32 biți)
DQ – date de tip 8 octeți (qword – 64 biți)
DT – date de tip 10 octeți (tword – 80 biți)

- declararea unei variabile (allocare fără inițializare)

[nume] tip_alocare factor [;comentariu]

unde:

- **tip_alocare** este o directivă de rezervare de date neinitializate, una din următoarele:

- RESB** – date de tip octet (byte – 8 biți)
- RESW** – date de tip cuvânt (word – 16 biți)
- RESD** – date de tip dublucuvânt (dword – 32 biți)
- RESQ** – date de tip 8 octeți (qword – 64 biți)
- REST** – date de tip 10 octeți (tword – 80 biți)

- **factor** este un număr care indică câți bytes/words/dwords/quadwords/twords vor fi alocați

- definirea unei constante

nume EQU valoare

Exemplu:

```
; segmentul de date
segment data use32 class=data
a db 7      ; 1 byte
b dw 101b   ; 2 bytes (1 word)
c dd 2bfh   ; 4 bytes (1 doubleword)
d dq 3070   ; 8 bytes (1 quadword)
e dt 1024   ; 10 bytes
```

3.c. Reșiștri în arhitectura IA-32

Arhitectura IA-32 pune la dispoziția programatorilor 16 reșiștri. Acești reșiștri pot fi grupați astfel:

Reșiștri de uz general (General-purpose registers)

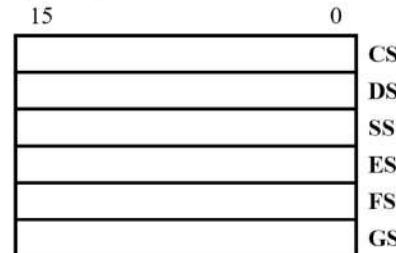
31

0

	EAX
	EBX
	ECX
	EDX
	ESI
	EDI
	EBP



Regiștri de segment (Segment registers)



Registrul cu flag-uri (Program status and control register)



Instruction pointer

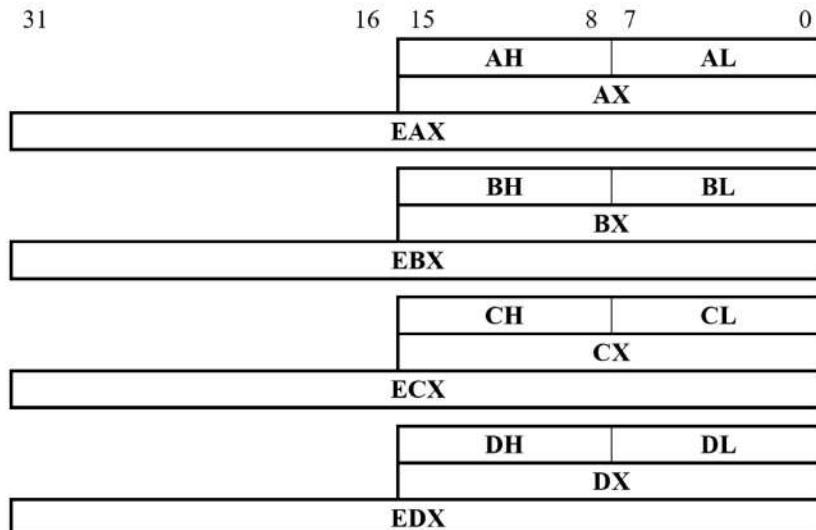


Fig. 1 Regiștri existenți în arhitectura IA-32

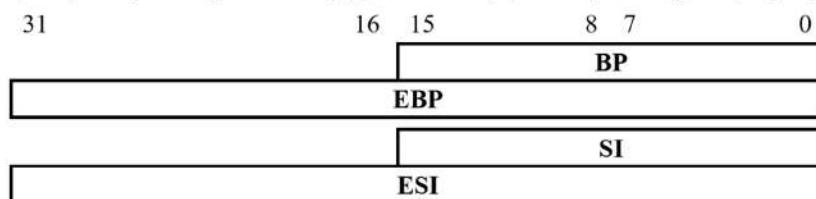
Regiștri de uz general (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) sunt regiștri pe 32 de biți și sunt destinați pentru a stoca: operanzi/rezultate ale operațiilor aritmetice sau logice, operanzi folosiți în calcule de adresă sau pointeri spre locații de memorie.

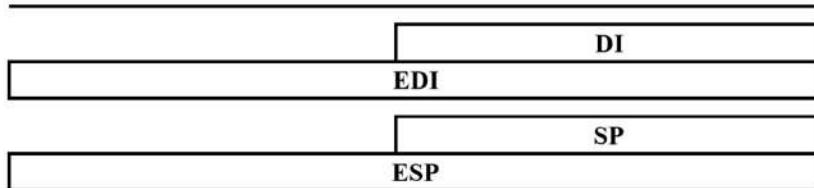
Regiștri EAX, EBX, ECX și EDX (pe 32 de biți) pot fi accesăți și:

- pe 8 biți: AH | AL, BH | BL, CH | CL, DH | DL;
- pe 16 biți: AX, BX, CX, DX.



Regiștri EBP, ESI, EDI și ESP (pe 32 de biți) pot fi accesăți DOAR pe 16 biți: BP, SI, DI, SP.





3.d. Formatul unei linii sursă

Formatul unei linii sursă în limbajul de asamblare este următorul:

[etichetă[:]] [prefixe] [instructiune] [operanzi] [:comentariu]

- *eticheta*: un nume scris de utilizator căreia se pot referi date sau locații de memorie
- *instructiune*: indică simbolic (printr-o mnemonică) acțiunea ce va fi executată de către microprocesor
- *operanzi*: pot fi valori imediate, regiștri sau locații de memorie
- *comentariu*: orice text care începe cu caracterul ;

3.e. Exemple de instrucțiuni în limbaj de asamblare

MOV

Sintaxa: **mov d, s**

unde:

- d poate fi un registru sau o locație de memorie
- s poate fi o valoare imediată (constantă), un registru sau o locație de memorie

Efect: $d \leftarrow s$

Cei doi operanzi d și s trebuie să aibă aceeași dimensiune de reprezentare (ambii sunt fie OCTETI, fie CUVINTE, fie DUBLUCUVINTE).

Ambii operanzi pot fi regiștri, însă CEL MULT un operand poate fi o locație de memorie.

Exemple:

```

mov al, 2
mov ax, bx
mov bl, [a]
mov [a], 22
mov [b], dl

```

ADD

Sintaxa: **add d, s**

unde:

- d poate fi un registru sau o locație de memorie
- s poate fi o valoare imediată (constantă), un registru sau o locație de memorie

Efect: $d \leftarrow d + s$

Cei doi operanzi d și s trebuie să aibă aceeași dimensiune de reprezentare (ambii sunt fie OCTETI, fie CUVINTE, fie DUBLUCUVINTE).

Ambii operanzi pot fi regiștri, însă CEL MULT un operand poate fi o locație de memorie.

Exemple:

```

add al, 10b
add ax, bx
add bl, [a]
add [a], 10h
add [b], dx

```

SUB

Sintaxa: **sub d, s**

unde:

- d poate fi un registru sau o locație de memorie
- s poate fi o valoare imediată (constantă), un registru sau o locație de memorie

Efect: $d \leftarrow d - s$

Cei doi operanzi d și s trebuie să aibă aceeași dimensiune de reprezentare (ambii sunt fie OCTETI, fie CUVINTE, fie DUBLUCUVINTE).

Ambii operanzi pot fi registri, însă CEL MULT un operand poate fi o locație de memorie.

Exemple:

```
sub al, 0fh
sub ax, bx
sub bl, [a]
sub [a], 1010b
sub [b], edx
```

4. PRIMUL PROGRAM ÎN LIMBAJ DE ASAMBLARE

```
; vom programa pe 32 de biti
bits 32

; declar punctul de intrare in program
; (start este o eticheta care indica prima instructiune din program)
global start

; declar functiile externe pe care le voi folosi
extern exit
import exit msvcrt.dll

; segmentul de date
; aici vom declara datele (constante, variabile etc.)
segment data use32 class=data

; segmentul de cod
; aici vom scrie programul
segment code use32 class=code
start:
    ; incepând de aici vom scrie instructiunile care compun programul

    ...

    ; inchei executia programului
    push dword 0      ; pun pe stiva argumentul functiei
    call [exit]        ; apelez functia exit()
```

EXERCITII

Scrieți un program în limbaj de asamblare care să calculeze expresia aritmetică, considerând domeniile de definiție ale variabilelor:

- a. $1 + 2$
- b. $1 - 2$
- c. $1 + 255$
- d. $a + b$, unde a, b – byte
- e. $a - b$, unde a, b – byte
- f. $a + b$, unde a, b – word
- g. $a - b$, unde a, b – word
- h. $a + b$, unde a, b – dword

- i. $a - b$, unde a, b – dword
- j. $(a + b) - (c + 10)$, unde a, b, c – byte
- k. $(a + b) - (c + 10)$, unde a, b, c – word
- l. $(a + b) - (c + 10)$, unde a, b, c – dword

19 octombrie

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 2 –

1. Obținerea offsetului/valorii unei variabile
2. Ordinea de plasare a octetilor în memorie
3. Instrucțiuni cu/fără semn
4. Instrucțiuni aritmetice pentru înmulțire/împărțire (cu/fără semn)
5. Instrucțiuni de conversie cu/fără semn
6. Instrucțiuni aritmetice care țin cont de transport
7. Instrucțiuni de lucru cu stiva

1. OBȚINEREA OFFSETULUI/VALORII UNEI VARIABILE

Dacă variabila a este un dublucuvânt (a dd 12345678h), atunci:

Instrucțiunea	Efect
mov eax, a	EAX = OFFSET-ul (32 de biți) la care este stocat variabila a
mov eax, [a]	EAX = VALOAREA variabilei a (dublucuvântul care începe de la offset-ul a)

2. ORDINEA DE PLASARE A OCTETILOR ÎN MEMORIE

Reprezentarea în memorie a datelor a căror dimensiune depășește un octet se poate realiza în două moduri distincte:

- placerea **little-endian**, în care octetul cu cea mai mică adresă din locația de memorie respectivă va conține octetul cel mai puțin semnificativ al reprezentării (octetul "end" al reprezentării are adresa cea mai "little");
- placerea **big-endian**, în care octetul cu cea mai mare adresă din locația de memorie respectivă va conține octetul cel mai puțin semnificativ al reprezentării (octetul "end" al reprezentării are adresa cea mai "big").

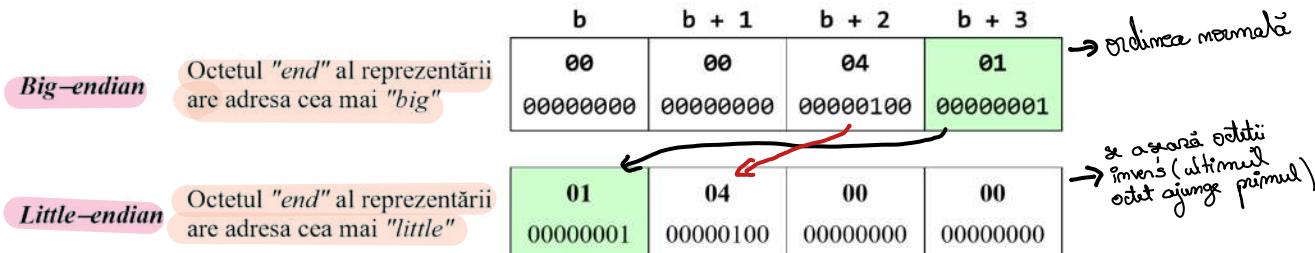
Discuție

Spre exemplu, dacă dorim să reprezentăm numărul $1025_{(10)}$ într-o locație de 4 octeți, procedăm astfel:

- convertim numărul în bazele 16 și 2:

$$1025_{(10)} = 00000401_{(16)} = 00000000 \ 00000000 \ 00000100 \ 00000001_{(2)} \xrightarrow{\text{octet end}}$$

- luând în considerare cele două moduri posibile, ordinea de plasare a octetilor în memorie va fi:



Modul de plasare a octetilor în memorie poate să difere de la un sistem de operare la altul. Familia de sisteme de operare Windows utilizează plasarea **little-endian**.

Exemplu

Se dă următorul segment de date:

```
segment data use32 class=data
a1 db 2, 4, 6, 8
a2 dw 2, 4, 6, 8
a3 dd 2, 4, 6, 8
a4 db '2', '4', '6', '8'
```

```

a5 db 24h, 68h
a6 dw 24h, 68h
a7 dd 24h, 68h
a8 db '24', '68'
a9 dw '24', '68'
a10 dw '2', '4', '6', '8'

a11 db 2468h
a12 dw 2468h
a13 dd 2468h
a14 dd 02040608h, 01030507h

```

Cum va fi reprezentat în memorie segmentul de date de mai sus ?

3. INSTRUCȚIUNI CU/FĂRĂ SEMN

Dacă ținem cont de reprezentarea numerelor cu/fără semn, în arhitectura IA-32, există trei tipuri de instrucțiuni:

- a. instrucțiuni care nu țin cont de reprezentarea cu/fără semn a numerelor: *mov add sub*
- b. instrucțiuni care interprează operanții ca fiind numere fără semn: *div mul*
- c. instrucțiuni care interprează operanții ca fiind numere cu semn: *idiv imul cbw cwd cwde cdq*

Este important ca programatorul să fie consistent atunci când programează în limbajul IA-32:

- dacă consideră toate valorile numerice ca fiind pozitive, atunci trebuie să folosească doar instrucțiuni de tipul **a** și **b**;
- dacă consideră toate valorile numerice ca fiind numere cu semn, atunci trebuie să folosească doar instrucțiuni de tipul **a** și **c**.

Observații

Atunci când se folosesc instrucțiuni cu doi operanzi trebuie să se țină cont de următoarele:

- ambii operanzi trebuie să aibă aceeași dimensiune de reprezentare (de exemplu putem aduna un octet cu un alt octet, dar nu un octet cu un cuvânt sau un octet cu un dublucuvânt);
- cel puțin un operand trebuie să fie un registru de uz general sau o valoare imediată (constantă);
- dacă operandul este constantă, acesta nu poate să fie operandul destinație.

Discuție

Fie segmentul de date și secvențele de instrucțiuni de mai jos:

```

; segmentul de date
segment data use32 class=data
    a db 10
    b db 11

; segmentul de date
segment code use32 class=code
    start:
        add [a], [b]      ; eroare la asamblare
        ...
        mov ax, [a]        ; incorect pentru că AX are 16 biți
        add ax, [b]        ; incorect pentru că se adună AX (16 biți) + b (8 biți)
        ...

```

- instrucțiunea

add [a], [b]

va produce eroare la asamblare, iar fișierul executabil nu va fi creat, deoarece operanții acestei instrucțiuni nu respectă cea de-a doua constrângere de mai sus;

- deși instrucțiunile:

mov ax, [a] → ax-16, a-8
add ax, [b] → ax-16, b-8

sunt incorecte, asamblorul nu va semnaliza eroare de sintaxă deoarece dimensiunea operandului sursă este dedusă din dimensiunea operandului destinație.

Astfel, după execuția primei instrucțiuni, în registrul **AX** vom avea cuvântul din memorie care începe la offset-ul a (cuvântul compus din octeții aflați la offset-ul a și a+1).

Cea de-a doua este incorectă pentru că se efectuează adunarea dintre un cuvânt (în registrul AX) și un octet (valoarea variabilei b).

4. INSTRUCȚIUNI ARITMETICE PENTRU ÎNMULȚIRE/ÎMPĂRTIRE (CU/FĂRĂ SEMN)

MUL – înmulțire fără semn

Sintaxa: `mul op`

unde `op` poate fi un registru sau o variabilă de tip octet, cuvânt sau dublucuvânt

Efect:

Dimensiune op	Registru implicit	Rezultat operație
1 octet (8 biți)	AL (1 octet = 8 biți)	AX (2 octeți = 16 biți)
1 cuvânt (16 biți)	AX (1 cuvânt = 16 biți)	DX (cuvântul superior) AX (cuvântul inferior)
1 dublucuvânt (32 biți)	EAX (1 dublucuvânt = 32 biți)	EDX (dublucuvântul superior) EAX (dublucuvântul inferior)

Exemplu

Instrucțiunea

`mul dx`



va înmulți cuvântul aflat în registrul **DX** cu cuvântul aflat în registrul **AX**.

Rezultatul operației va fi un număr reprezentat pe 32 de biți (1 dublucuvânt) și va fi stocat în doi registri **DX:AX** din motive de compatibilitate cu arhitecturile Intel 8086 precedente.

Dacă presupunem că rezultatul înmulțirii este numărul 12345678h, atunci cuvântul inferior (cel mai puțin semnificativ) va fi stocat în registrul **AX** (**AX** = 5678h), iar cuvântul superior (cel mai semnificativ) va fi stocat în registrul **DX** (**DX** = 1234h).

DIV – împărțire fără semn

Sintaxa: `div op`

unde `op` poate fi un registru sau o variabilă de tip octet, cuvânt sau dublucuvânt

Efect:

Deîmpărțit	Dimensiune op	Cât	Rest
AX (1 cuvânt = 16 biți)	: 1 octet (8 biți)	AL (1 octet = 8 biți)	AH (1 octet = 8 biți)
DX:AX (1 dublucuvânt = 32 biți)	: 1 cuvânt (16 biți)	AX (1 cuvânt = 16 biți)	DX (1 cuvânt = 16 biți)
EDX:EAX (1 quadword = 64 biți)	: 1 dublucuvânt (32 biți)	EAX (1 dublucuvânt = 32 biți)	EDX (1 dublucuvânt = 32 biți)

IMUL și **IDIV** reprezintă varianta cu semn a instrucțiunilor MUL și DIV (operanții sunt interpretați ca numere cu semn).

5. INSTRUCȚIUNI DE CONVERSIE CU/FĂRĂ SEMN

5.a. Instrucțiuni de conversie fără semn

Nu există instrucțiuni de conversie fără semn.

Conversiile fără semn se realizează prin „zerorizarea” octetului, cuvântului sau dublucuvântului superior.

Exemple

```
; segmentul de date
segment data use32 class=data
    a db 10
    b dw 1122h
    c dd 11223344h

; segmentul de cod
segment code use32 class=code
    start:
        ; BYTE -> WORD
        mov al, [a]      ; AL = 00001010
        mov ah, 0         ; AX = AH:AL = 00000000:00001010 (extindere fără semn)

        ; WORD -> DWORD
        mov ax, [b]      ; AX = 1122h
        mov dx, 0         ; DX:AX = 0000:1122h (extindere fără semn)

        ; DWORD -> QUADWORD
        mov eax, [c]      ; EAX = 11223344h
        mov edx, 0         ; EDX:EAX = 00000000:11223344h (extindere fără semn)
```

5.b. Instrucțiuni de conversie cu semn

CBW

Sintaxa: cbw

Efect: convertește cu semn BYTE-ul din AL la un WORD în AX

Instrucțiunea nu are operanzi, deci va realiza ÎNTOTDEAUNA conversia AL → AX.

Conversia se realizează prin extinderea reprezentării de pe 8 biți pe 16 biți, prin completarea cu bitul de semn în fața octetului inițial.

CWD

Sintaxa: cwd

Efect: convertește cu semn WORD-ul din AX la un DWORD în DX:AX

Instrucțiunea nu are operanzi, deci va realiza ÎNTOTDEAUNA conversia AX → DX:AX.

Conversia se realizează prin extinderea reprezentării de pe 16 biți pe 32 biți, prin completarea cu bitul de semn în fața cuvântului inițial.

CWDE

Sintaxa: cwde

Efect: convertește cu semn WORD-ul din AX la un DWORD în EAX

Instrucțiunea nu are operanzi, deci va realiza ÎNTOTDEAUNA conversia AX → EAX.

Conversia se realizează prin extinderea reprezentării de pe 16 biți pe 32 biți, prin completarea cu bitul de semn în fața cuvântului inițial.

CDQ

Sintaxa: cdq

Efect: convertește cu semn DWORD-ul din EAX la un QUADWORD în EDX:EAX

Instrucțiunea nu are operanzi, deci va realiza ÎNTOTDEAUNA conversia EAX → EDX:EAX.

Conversia se realizează prin extinderea reprezentării de pe 32 biți pe 64 biți, prin completarea cu bitul de semn în fața dublucuvântului inițial.

6. INSTRUCȚIUNI ARITMETICE CARE ȚIN CONT DE TRANSPORT

Există situații în care valorile unor variabile/rezultate se găsesc jumătate într-un registru și jumătate în altul. În aceste cazuri, poate fi convenabil să adunăm/scădem pe două etape: adunăm/scădem mai întâi reședința care conțin parte inferioară a reprezentării, apoi pe cei care conțin parte superioară a acesteia. Însă rezultatul operației nu va fi corect, dacă, în a două etapă, nu ținem cont de un eventual transport/imprumut generat de operația efectuată în prima etapă.

CF (Carry Flag) este flagul de transport. CF va avea valoarea 1 dacă în cadrul ultimei operații efectuate → (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar.

Instrucțiunile aritmetice care țin cont de transport sunt **ADC** și **SBB**.

$$\left\{ \begin{array}{l} CF = 1, \sqrt{T_{\text{transport}}} \\ CF = 0, \times T_{\text{transport}} \end{array} \right.$$

ADC (ADd with Carry)

Sintaxa: **adc d, s**

unde:

- d poate fi un registru sau o locație de memorie
- s poate fi o valoare imediată (constantă), un registru sau o locație de memorie

Efect: $d \leftarrow d + s + \text{CF}$ (Carry Flag)

SBB (SuBtract with Borrow)

Sintaxa: **sbb d, s**

unde:

- d poate fi un registru sau o locație de memorie
- s poate fi o valoare imediată (constantă), un registru sau o locație de memorie

Efect: $d \leftarrow d - s - \text{CF}$ (Carry Flag)

Exemple: sem2_byte+word.asm, sem2_word+dword.asm, sem2_dword+qword.asm

7. INSTRUCȚIUNI DE LUCRU CU STIVA

Orice program care se execută utilizează o stivă de execuție (execution stack, run-time stack).

Stiva este o structură de date care funcționează pe principiul LIFO (Last-In-First-Out). Singurul element direct accesibil este cel aflat în vârful stivei.

Segmentul de memorie în care este localizată stiva este indicat de către registrul SS (Stack Segment), iar offset-ul locației de memorie aflate în vârful stivei se găsește în registrul (E)SP (Stack Pointer).

Cele mai utilizate instrucțiuni de lucru cu stiva sunt **PUSH** (pone un element în stivă) și **POP** (extrag un element din stivă).

PUSH

Sintaxa: **push source**

unde source poate fi o valoare imediată, un registru sau o locație de memorie pe 16 sau 32 biți

Efect:

- a. dacă source este de tip cuvânt (16 biți):

$ESP \leftarrow ESP - 2$

[SS: ((ESP + 1):ESP)] \leftarrow source

- b. dacă source este de tip dublucuvânt (32 biți):

$ESP \leftarrow ESP - 4$

[SS: ((ESP + 2):ESP)] \leftarrow source

POP

Sintaxa: **pop destination**

unde destination poate fi un registru sau o locație de memorie pe 16 sau 32 biți

Efect:

a. dacă destination este de tip cuvânt (16 biți):

```
destination ← [SS:((ESP + 1):ESP)]
```

```
ESP ← ESP + 2
```

b. dacă destination este de tip dublucuvânt (32 biți):

```
destination ← [SS:((ESP + 2):ESP)]
```

```
ESP ← ESP + 4
```

*va scoate ultimii
32 de biți,*

Observații:

- pe/din stivă pot fi puse/extrase doar cuvinte (16 biți) sau dublucuvinte (32 biți)
- stiva crește invers de la adrese de memorie mari către adrese de memorie mici
- (E)SP va indica offset-ul celui mai puțin semnificativ octet al elementului aflat în vârful stivei → offset = poziție

Exemplu: sem2_utilizare_stiva.asm

EXERCITII

Scripti un program în limbaj de asamblare care să calculeze expresia aritmetică, considerând domeniile de definiție ale variabilelor:

1. $x = [(a + b) * c] / d$, unde a, b, c, d – byte
2. $x = (a - b * c) / d$, unde a, b, c, d – byte
3. $x = (a * b) / d - c$, unde a, c, d – byte, b – word

Anhitectura sistemelor de calcul

bits 32

global start

extern exit

import exit msvcrt.dll

segment data use 32 class = data

; $x = (a * b) + (c * d)$ fără semn

a dw 1

b dw 2

c dw 3

d dw 4

x resd 1

segment code use 32 class = code

start:

; $a * b$

mov ax, a

mul word b ; dx:ax = $(a * b)$

mov cx, dx

mov bx, ax ; cx:bx = $(a * b)$; $c * d$

mov ax, c

mul word d ; dx:ax = $(c * d)$

add ax, bx

adc dx, cx ; dx:ax = $(a * b) + (c * d)$

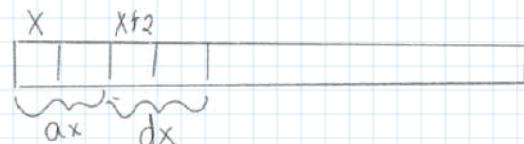
mov [x], ax

mov [x+2], dx

a	b	c
01	00	02

00	03	00
----	----	----

→ setarea CF



→ operațiile aritmetice și ținerea modifică flag-urile

~~add [a], [b]~~~~cmp [a], [b]~~~~mov [a], [b]~~

illegal

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 3 –

1. Instrucțiuni de comparare
2. Salturi
3. Instrucțiuni de ciclare
4. Siruri

1. INSTRUCȚIUNI DE COMPARARE

CMP

Sintaxa: cmp d, s ~~(=)~~ sub d, s

Efect: execută **operația aritmetică d - s** (fără să depună rezultatul în d) și modifică flag-urile în funcție de rezultatul operației efectuate

Flag-uri afectate: OF, SF, ZF, AF, PF, CF

Constrângeri:

- operatorul destinație d poate fi un registru sau o locație în memorie;
- operandul sursă s poate fi un registru, o locație în memorie sau o valoare imediată (constantă);
- operanții trebuie să aibă aceeași dimensiune de reprezentare (octet/cuvânt/dublucuvânt);
- operanții nu pot să fie SIMULTAN locații de memorie (unul dintre ei trebuie să fie un registru).

Observații

Deși numele acestei instrucțiuni este CMP este important de subliniat că, în realitate, această instrucțiune **NU COMPARĂ nimic**, nestabilind niciun criteriu de comparație și neluând de fapt nicio decizie. Ea doar PREGĂTEȘTE decizia corespunzător cu flagurile setate, comparația efectivă și decizia corespunzătoare fiind luată concret de instrucțiunea de salt condiționat care va fi folosită ulterior instrucțiunii CMP !

Dacă nu folosim ulterior nicio instrucțiune de salt condiționat, CMP nu are nici un rol concret în vreo comparație, ea reprezintă doar o simplă scădere fictivă cu rol de afectare a flagurilor și nu își va merita în niciun caz numele de CMP (compare).

TEST

Sintaxa: test d, s

Efect: execută **operația logică d AND s** (fără să depună rezultatul în d) și modifică flag-urile în funcție de rezultatul operației efectuate

Flag-uri afectate: poate modifica doar SF, ZF, PF (CF = 0, OF = 0, AF - nedefinit)

Constrângeri: identice cu cele menționate pentru instrucțiunea CMP

Observații

Deși numele acestei instrucțiuni este TEST este important de subliniat că în realitate această instrucțiune **NU TESTEAZA nimic**, nestabilind niciun criteriu de testare și neluând de fapt nicio decizie, ci ea doar PREGĂTEȘTE decizia corespunzător cu flagurile setate, criteriul de testare, testarea efectivă și decizia corespunzătoare fiind luată concret de instrucțiunea de salt condiționat care va fi folosită ulterior instrucțiunii TEST !

→ trebuie să urmărești și înșeuări une de salt

Dacă nu folosim ulterior nicio instrucțiune de salt condiționat, TEST nu are nici un rol concret în vreo testare, ea reprezentând doar o simplă operație AND (bit cu bit) cu rol de afectare a flagurilor și nu își va merita în niciun caz numele de TEST (testarea unei condiții).

2. SALTURI

Salturile pot fi de 2 (două) tipuri: *necondiționate* sau *condiționate*.

2.a. Saltul necondiționat

Există o singură instrucțiune de salt necondiționat:

JMP

Sintaxa: jmp op

unde op poate fi o etichetă, un registru sau o locație memorie ce conține o adresă.

Efect: execuția va continua cu instrucțiunea care urmează după etichetă, care se află la adresa conținută în registru sau la adresa conținută în locația de memorie

Instrucțiunea JMP poate fi utilizată pentru a executa oricare dintre următoarele 3 (trei) tipuri de salturi:

- **short jump:** un salt la o instrucțiune al cărei offset este cuprins între [-128, 127] octeți față de valoarea curentă din registrul EIP
- **near jump** (intrasegment jump): un salt la o instrucțiune aflată în segmentul de cod curent (referit de către registrul de segment CS)
- **far jump** (intersegment jump): un salt la o instrucțiune aflată într-un alt segment de cod

2.b. Instrucțiuni de salt condiționat

Jcc (Conditional jump instructions)

Sintaxa: jcc eticheta

Efect: transferă controlul programului instrucțiunii care urmează după eticheta, dacă condiția specificată de cc (condition code associated with the instruction) este adevărată

Lista completă a instrucțiunilor de salt condiționat este prezentată în "Intel 64 and IA-32 Architectures Software Developer's Manual, volume 1", care poate fi consultată la această adresă (vezi secțiunea 7.3.8):

<https://cdrdv2.intel.com/v1/dl/getContent/671436>

Mnemonica	Semnificație	Condiția verificată
JB	este inferior	
JNAE	nu este superior sau egal	
JC	există transport	
JAE	este superior sau egal	
JNB	nu este inferior	
JNC	nu există transport	
JBE	este inferior sau egal	
JNA	nu este superior	CF=1 sau ZF=1
JA	este superior	
JNBE	nu este inferior sau egal	CF=0 și ZF=0
JE	este egal	
JZ	este zero	ZF=1

JNE	nu este egal	ZF=0
JNZ	nu este zero	
JL	este mai mic decât	SF≠OF
JNGE	nu este mai mare sau egal	
JGE	este mai mare sau egal	SF=OF
JNL	nu este mai mic decât	
JLE	este mai mic sau egal	ZF=1 sau SF≠OF
JNG	nu este mai mare decât	
JG	este mai mare decât	ZF=0 și SF=OF
JNLE	nu este mai mic sau egal	
JP	are paritate	PF=1
JPE	paritatea este pară	
JNP	nu are paritate	PF=0
JPO	paritatea este impară	
JS	nu are semn negativ	SF=0
JO	există depășire	OF=1
JNO	nu există depășire	OF=0

Compararea a două numere poate fi făcută cu semn sau fără semn. De aceea, convenim că:

- atunci când se compară două numere fără semn se folosesc termenii "*below*" (inferior, sub), respectiv "*above*" (superior, deasupra, peste);
- atunci când se compară două numere cu semn se folosesc termenii "*less than*" (mai mic decât) și "*greater than*" (mai mare decât).

Pentru a facilita alegerea corectă de către programator a instrucțiunilor de salt condiționat în raport cu rezultatul unei comparații (cu semn sau fără semn), dăm următorul tabel:

Relația dintre operanzi	Comparație cu semn	Comparație fără semn
$d = s$	je	je
$d \neq s$	jne	jne
$d < s$	jl	jb
$d \leq s$	jle	jbe
$d > s$	jg	ja
$d \geq s$	jge	jae

!

Observații

- nu instrucțiunea CMP este cea care face distincție între o comparație cu semn și una fără semn;
- rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine EXCLUSIV instrucțiunilor de salt condiționat care urmează după instrucțiunea CMP;
- dacă considerăm cazul $s = 0$, tabelul rămâne valabil.

```
segment code use32 class=code
start:
    mov eax, [a]
    cmp eax, [b] ;operatie fictiva a-b, se vor seta flagurile
    je sunt_egale ;se executa doar daca a=b TRUE, else saltul nu se executa
    ;pt fals codul continua in acest bloc de instructiuni

    jmp peste ;sarim peste cod
    sunt_egale
    ;pt a=b codul continua aici
    ;...
    peste
    push dword 0
    call [exit]
```

3. INSTRUCȚIUNI DE CICLARE

LOOP

Sintaxa: loop eticheta

Condiția testată: ECX = 0

Efect: decrementează conținutul registrului ECX, apoi, dacă condiția nu este adevărată, execută salt la instrucțiunea care urmează după eticheta

LOOPE (LOOP while Equal)

Sintaxa: loope eticheta

Condiții testate: ECX = 0 sau ZF = 0

Efect: decrementează valoarea registrului ECX, apoi, dacă niciuna din condiții nu e adevărată, execută salt la instrucțiunea care urmează după eticheta

LOOPNE (LOOP while Not Equal)

Sintaxa: loopne eticheta

Condiția de terminare: ECX = 0 sau ZF = 1

Efect: decrementează valoarea registrului ECX, apoi, dacă niciuna dintre condiții nu e adevărată, execută salt la instrucțiunea care urmează după eticheta

```
segment code use32 class=code
start:
;code
mov ecx, 5
repeta:
;...
;...
;...
loop repeta ;se repeta de cate ori indică ecx, adică de 5 ori
```

→ { dec ecx
 cmp ecx, 0
 } jne repeta / ja repeta
 → dacă ecx = 0 ⇒ loop infinit

jcxzb final → dacă ecx = 0 jump
 ⇌ { cmp ecx, 0
 } jz final

4. řIRURI

4.a. Siruri numerice (de octeti/cuvinte/dublucuvinte/quadwords)

Reprezentarea datelor în memorie (little endian)

s1

s1 db 12, 13, 14

0c	0d	0e					
s2							

little
endian

s2 dw 1234h, 5678h

34	12	78	56				
s3							

s3 dd 12345678h, 1234abcdh

78	56	34	12	cd	ab	34	12	

4.b. Siruri de caractere

Caracterele sunt reprezentate în cod ASCII (American Standard Code for Information Interchange).

Reprezentarea datelor în memorie (little endian)

s1

s1 db 'a', 'b', 'c'

61	62	63						
s2								

s2 dw 'a', 'b', 'c'

61	00	62	00	63	00			
s3								

s3 dd 'a', 'b'

61	00	00	00	62	00	00	00	

4.c. Determinarea lungimii unui sir

Determinarea lungimii unui ſir (in OCTETI) se poate face folosind contorul de locații.

Contorul de locații este un număr întreg gestionat de către asamblor. În orice moment, valoarea acestui contor coincide cu numărul de octeti generati corespunzător instrucțiunilor și directivelor deja întâlnite în cadrul segmentului respectiv (deplasamentul curent în cadrul segmentului).

Programatorul poate să acceseze (poate să citească, dar nu poate să modifice) valoarea curentă a contorului de locații utilizând simbolul \$.

Exemplu

Fie segmentul de date care urmează:

```
...
; segmentul de date
segment data use32 class=data
    s db 'a', 'b', 'c', 'd', 'e'
    len equ $-s           ; lungimea ſirului s (în octeți)

; segmentul de date
segment code use32 class=code
start:
    ...

```

len equ 5

00000000

↓
00000005
(contor de locație)

6A 62 63 64 65

} ⇒ len equ \$-s
(5)cod Ascii
literă = octet

EXERCITII

1. Se dă un sir de caractere S. Se cere sirul de caractere D obținut prin copierea sirului S.
2. Se dă un sir de caractere S format din litere mici. Să se construiască un sir de caractere D care să conțină literele din sirul inițial transformate în majuscule.
3. Se dă un sir de numere întregi S. Să se construiască sirul P care conține toate numerele pare din S și sirul R care conține toate numerele impare din S.
4. Se dă un sir de cuvinte S. Să se determine suma numerele formate din biții 6-9 ai fiecărui cuvânt din sirul S.
5. Se dă un sir de cuvinte S. Să se formeze sirul de octeți D care conține octeții superiori rotiți spre stânga cu valoarea octeților inferiori din fiecare cuvânt al sirului de cuvinte S.

T
o constantele me apăr în memorie

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 4 –

OPERAȚII PE ȘIRURI DE OCTEȚI/CUVINTE/DUBLUCUVINTE

Instrucțiunile pe șiruri nu au operanzi.

Pregătirea execuției instrucțiunilor pe șiruri este obligatorie și constă în:

- stabilirea valorii DF (Direction Flag) utilizând instrucțiunile: **CLD** (DF = 0) sau **STD** (DF = 1);
- încărcarea în registrul ESI (Source Index) a offset-ului șirului sursă;
- încărcarea în registrul EDI (Destination Index) a offset-ului șirului destinație.

a. Instrucțiuni pentru transferul datelor

LODS (LOaD String)
STOS (STOre String)
MOVS (MOVe String)

urmate de o literă care indică dimensiunea operației

$\left\{ \begin{array}{l} \mathbf{B} \text{ (byte)} \\ \mathbf{W} \text{ (word)} \\ \mathbf{D} \text{ (doubleword)} \end{array} \right.$

LODSB	În AL se încarcă <u>octetul</u> de la adresa <DS:ESI> Dacă DF=0 atunci inc ESI, altfel dec ESI
LODSW	În AX se încarcă <u>cuvântul</u> de la adresa <DS:ESI> Dacă DF=0 atunci ESI=ESI+2, altfel ESI=ESI-2
LODSD	În EAX se încarcă <u>dublucuvântul</u> de la adresa <DS:ESI> Dacă DF=0 atunci ESI=ESI+4, altfel ESI=ESI-4
STOSB	La adresa <ES:EDI> se încarcă <u>octetul</u> din AL Dacă DF=0 atunci inc EDI, altfel dec EDI
STOSW	La adresa <ES:EDI> se încarcă <u>cuvântul</u> din AX Dacă DF=0 atunci EDI=EDI+2, altfel EDI=EDI-2
STOSD	La adresa <ES:EDI> se încarcă <u>dublucuvântul</u> din EAX Dacă DF=0 atunci EDI=EDI+4, altfel EDI=EDI-4
MOVSB	La adresa <ES:EDI> se încarcă <u>octetul</u> de la adresa <DS:ESI> Dacă DF=0 atunci inc ESI și inc EDI, altfel dec ESI și dec EDI
MOVSW	La adresa <ES:EDI> se încarcă <u>cuvântul</u> de la adresa <DS:ESI> Dacă DF=0 atunci ESI=ESI+2 și EDI=EDI+2, altfel ESI=ESI-2 și EDI=EDI-2
MOVSD	La adresa <ES:EDI> se încarcă <u>dublucuvântul</u> de la adresa <DS:ESI> Dacă DF=0 atunci ESI=ESI+4 și EDI=EDI+4, altfel ESI=ESI-4 și EDI=EDI-4

b. Instrucțiuni pentru consultarea/compararea datelor

SCAS (SCAn String)
CMPS (CoMPare String)

urmate de o literă care indică dimensiunea operației

$\left\{ \begin{array}{l} \mathbf{B} \text{ (byte)} \\ \mathbf{W} \text{ (word)} \\ \mathbf{D} \text{ (doubleword)} \end{array} \right.$

SCASB	CMP AL, <ES:EDI> Dacă DF=0 atunci inc EDI, altfel dec EDI
SCASW	CMP AX, <ES:EDI> Dacă DF=0 atunci EDI=EDI+2, altfel EDI=EDI-2

SCASD	CMP EAX, <ES:EDI> Dacă DF=0 atunci EDI=EDI+4, altfel EDI=EDI-4
CMPSB	CMP <DS:ESI>, <ES:EDI> Dacă DF=0 atunci inc ESI și inc EDI, altfel dec ESI și dec EDI
CMPSW	CMP <DS:ESI>, <ES:EDI> Dacă DF=0 atunci ESI=ESI+2 și EDI=EDI+2, altfel ESI=ESI-2 și EDI=EDI-2
CMPSD	CMP <DS:ESI>, <ES:EDI> Dacă DF=0 atunci ESI=ESI+4 și EDI=EDI+4, altfel ESI=ESI-4 și EDI=EDI-4

c. Prefixe pentru execuția repetată a unei instrucțiuni pe siruri

Construcția:

prefix_instrucțiune instrucțiune_pe_siruri

este echivalentă cu:

```
repeta:
    instrucțiune_pe_siruri
loop repeta
```

unde **prefix_instrucțiune** este unul din următoarele prefixe:

REP (REPeat)	Repetă cât timp ECX ≠ 0
REPE (REPeat while Equal) REPZ (REPeat while Zero)	Repetă cât timp ECX ≠ 0 și ZF = 1
REPNE (REPeat while Not Equal) REPNZ (REPeat while Not Zero)	Repetă cât timp ECX ≠ 0 și ZF = 0

Observație

Prefixele **REPE/REPZ** și **REPNE/REPNZ** sunt destinate pentru a fi utilizate DOAR cu instrucțiunile **STOSx**, **MOVsx**, **SCASx** sau **CMPSx**.

Exemple

1. Copierea unui sir într-un alt sir

```
...
segment data use32 class=data
s db 'a', 'b', 'c', 'd', 'e'
len equ $-s           ; lungimea sirului s (în octeți)
d times len db 0

segment code use32 class=code
start:
    mov ecx, len      ; numărul de elemente ale sirului
    cld                ; DF = 0
    mov esi, s          ; ESI = offset-ul sirului sursă
    mov edi, d          ; EDI = offset-ul sirului destinație
    rep movsb          ; execuție repetată până când ECX = 0
...
```

2. Căutarea unei valori într-un sir

```
...
segment data use32 class=data
s db 2, 1, 3, 5, 7
len equ $-s           ; lungimea sirului s (în octeți)
```

```
segment code use32 class=code
start:
    mov ecx, len      ; numărul de elemente ale șirului
    cld                ; DF = 0
    mov edi, s         ; ESI = offset-ul șirului săursă
    mov al, 5          ; AL = 5 (5 este valoarea căutată)
    repe scasb        ; execuție repetată până când ECX = 0 sau ZF=1
...
```

3. Cea mai rapidă cale de inițializare a unui bloc de memorie de mare dimensiune: **rep stosb**

EXERCITII

1. Se dă un șir de caractere S. Să se copieze elementele șirului S într-un alt șir de caractere D, folosind instrucțiuni pe șiruri.
2. Se dă un șir de caractere S format din litere mici. Să se construiască un șir de caractere D care să conțină literele din șirul inițial transformate în majuscule, folosind instrucțiuni pe șiruri.
3. Se dă un șir de octeti S. Să se construiască șirul de octeți D, care conține pe fiecare poziție numărul de biți care au valoarea 1 din octetul de pe poziția corespunzătoare din S.

Exemplu:

S: 5, 25, 55, 127

S în baza 2: 101, 11001, 10111, 1111111

D: 2, 3, 5, 7

4. Se dă un șir de cuvinte S. Să se construiască două șiruri de octeți:

- B1 care are ca elemente partea superioară a cuvintelor din s;
- B2 care are ca elemente partea inferioară a cuvintelor din s.

5. Se dau două șiruri de octeți S1 și S2 de lungimi egale. Să se determine poziția p în care elementele ambelor șiruri sunt egale.

6. Se dă un șir de octeți S. Să se ordoneze crescător elementele șirului.

7. Se dă un unui șir de octeți reprezentând un text (o succesiune de șiruri de caractere separate prin spații).

Să se determine cuvintele din șir care sunt palindroame (ex. cojoc, capac etc.).

- saltwater conditional sumt mean

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 5 –

1. Apeluri de funcții
2. Operații cu fișiere de tip text

1. APELURI DE FUNCȚII

O *bibliotecă de funcții* (*a function library*) este un fișier care conține definițiile (în cod mașină) unui grup de funcții. Un exemplu de bibliotecă de funcții este *msvcrt.dll* (*Microsoft Visual C Runtime*).

Lista completă a funcțiilor definite în biblioteca de funcții *msvcrt.dll* poate fi consultată la această adresă:
<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference>

1.a. Instrucțiunea **CALL**

Apelul unei funcții de bibliotecă definite într-o bibliotecă de funcții externă se realizează cu ajutorul instrucțiunii **CALL** astfel:

```
call [nume_funcție]
```

Instrucțiunea **CALL** execută un „*salt cu revenire*”, prin urmare, execuția sa va produce următoarele efecte:

- se pune în vârful stivei offset-ul instrucțiunii imediat următoare (valoarea curentă din registrul EIP);
- se execută un salt de tip FAR în segmentul de cod care conține definiția funcției apelate.

Observații

Înainte de a efectua apelul unei funcții definite într-o bibliotecă de funcții externă este **OBLIGATORIU** să indicăm asamblorului unde se găsește definiția funcției apelate. În cazul apelului în NASM a unor funcții definite în libraria *msvcrt.dll* (*Microsoft Visual C Runtime*) acest lucru se realizează cu ajutorul directivelor **extern** și **import**, astfel:

- indicăm asamblorului că aceste funcții (simboluri) sunt definite într-o librărie externă (nu sunt definite în segmentul de cod curent):

```
extern nume_funcție1, nume_funcție2, nume_funcție3
```

- pentru FIECARE dintre funcțiile enumerate mai sus, indicăm asamblorului de unde va „*importa*” acele funcții (care este numele librăriei în care sunt definite):

```
import nume_funcție1 msvcrt.dll
import nume_funcție2 msvcrt.dll
import nume_funcție3 msvcrt.dll
```

1.b. Convenții de apel

O *convenție de apel* (*a calling convention*) definește un ansamblu de reguli referitoare la:

- modul și ordinea de transmitere a argumentelor funcției;
- unde va fi returnat rezultatul execuției;
- care sunt regiștri care pot fi modificați pe timpul execuției;
- cine are obligația să „*șteargă*” argumentele transmise.

Apelul unei funcții din biblioteca *msvert.dll* (*Microsoft Visual C Runtime*) necesită utilizarea convenției de apel *_cdecl*. Această convenție de apel specifică următoarele reguli:

- argumentele funcției apelate se pun pe stivă începând cu ultimul argument din lista de argumente (de la dreapta spre stânga listei);
- dacă funcția apelată returnează un rezultat, acesta va fi depus implicit în registrul EAX;
- funcția apelată nu va „sterge” argumentele puse pe stivă (este sarcina funcției apelante);
- pe timpul execuției funcției apelate conținutul regiștrilor EAX, ECX și EDX poate fi modificat.

1.c. Funcții pentru afișare pe ecran

Pentru afișarea rezultatelor pe ecran (la consolă) vom utiliza funcția:

```
printf(const char* format, val_1, val_2, ...)
```

unde:

format este offset-ul șirului de caractere care specifică formatul de afișare a datelor, astfel:

Specificator	Ce se afișează	Exemplu	Dimensiune de reprezentare
"%c"	Caracter	<i>a</i>	octet
"%d" "%i"	Număr cu semn în baza 10	<i>392</i> <i>-1024</i>	dublucuvânt
"%u"	Număr fără semn în baza 10	<i>7235</i>	dublucuvânt
"%x"	Număr fără semn în baza 16	<i>7fa</i>	dublucuvânt
"%s"	Șir de caractere (terminat cu 0)	<i>"ana are mere"</i>	șir de octeți terminat cu 0

1.d. Funcții pentru citirea de la tastatură

Pentru citirea datelor de la tastatură putem utiliza funcțiile:

```
gets(char* buffer)  
scanf(const char* format [, argument]...)
```

unde:

buffer este offset-ul zonei de memorie în care vor fi citite datele

format este specificatorul de format (vezi tabelul de mai sus)

2. OPERAȚII CU FIȘIERE DE TIP TEXT

Lucrul cu fișiere implică execuția următoarelor operații:

- a. deschiderea/crearea fișierului;
- b. prelucrarea datelor (citirea și/sau scrierea datelor din/in fișier);
- c. închiderea fișierului.

2.a. Deschiderea/Crearea unui fișier text

Pentru deschiderea/crearea unui fișier de tip text vom utiliza funcția **fopen()**.

Prototipul acestei funcții în limbajul C este:

```
FILE* fopen(const char* nume_fisier, const char* mod_acces)
```

unde:

nume_fisier este offset-ul unui șir de caractere care specifică numele fișierului (calea absolută/relativă)

mod_acces este offset-ul unui șir de caractere care specifică drepturile de acces, astfel:

Mod acces	Drepturi de acces	Descriere
"r"	citire (read)	Deschide un fișier text pentru citire. <u>Fișierul trebuie să existe pe disc</u> . Dacă acest fișier nu există, funcția va returna valoarea 0 (eroare la deschiderea fișierului).
"w"	scriere (write)	Dacă nu există un fișier cu acel nume, va crea fișierul și îl va deschide pentru scriere. Dacă există deja un fișier cu acel nume, îl va deschide pentru scriere, însă conținutul inițial al fișierului <u>va fi suprascris</u> (scrierea se va face de la începutul fișierului).
"a"	adăugare (append)	Dacă nu există un fișier cu acel nume, va crea fișierul și îl va deschide pentru scriere. Dacă există deja un fișier cu acel nume, îl va deschide pentru scriere, însă conținutul inițial al fișierului <u>va fi păstrat</u> (scrierea se va face de la sfârșitul fișierului).
"r+"	citire și scriere	Deschide un fișier text pentru citire și scriere. <u>Fișierul trebuie să existe pe disc</u> . Dacă acest fișier nu există, funcția va returna valoarea 0 (eroare la deschiderea fișierului).
"w+"	citire și scriere	Dacă nu există un fișier cu acel nume, va crea fișierul și îl va deschide pentru citire și scriere. Dacă există deja un fișier cu acel nume, îl va deschide pentru citire și scriere, însă conținutul inițial al fișierului <u>va fi suprascris</u> (scrierea se va face de la începutul fișierului).
"a+"	citire și adăugare	Dacă nu există un fișier cu acel nume, va crea fișierul și îl va deschide pentru citire și scriere. Dacă există deja un fișier cu acel nume, îl va deschide pentru citire și scriere, însă conținutul inițial al fișierului <u>va fi păstrat</u> (scrierea se va face de la sfârșitul fișierului).

Dacă apelul funcției s-a realizat cu succes, `fopen()` va returna un descriptor de fișier (un pointer către structura FILE). Dacă apelul a eşuat, funcția va returna valoarea 0.

Observații

1. Este NECESAR să stocăm descriptorul de fișier într-o variabilă definită în segmentul de date, deoarece acest descriptor va fi folosit de către funcțiile de citire/scriere din/in fișier.
2. Având în vedere că există situații în care apelul funcției `fopen()` poate să eșueze, este OBLIGATORIU să verificăm dacă fișierul a fost deschis cu succes sau nu:

```
cmp eax, 0
je eroare_deschidere_fisier
```

2.b.1. Citirea dintr-un fișier text

Citirea dintr-un fișier care a fost deschis anterior cu `fopen()` se poate realiza cu ajutorul funcțiilor:

```
int fread(void* buffer, int size, int count, FILE* stream)
int fscanf(FILE* stream, const char* format, ...)
```

unde:

`buffer` este offset-ul zonei de memorie în care vor fi citite datele

`size` este dimensiunea tipului de date care vor fi citite

`count` este numărul de elemente care vor fi citite

`stream` este descriptorul de fișier (un pointer către structura FILE)

`format` specifică formatul operației de citire din fișier

2.b.2. Scrierea într-un fișier text

Scrierea într-un fișier care a fost deschis anterior cu `fopen()` se poate realiza cu ajutorul funcțiilor:

```
int fwrite(void* buffer, int size, int count, FILE* stream)
int fprintf(FILE* stream, const char* format, ...)
```

unde:

buffer este offset-ul zonei de memorie din care vor fi scrise datele

size este dimensiunea tipului de date care vor fi scrise

count este numărul de elemente care vor fi scrise

stream este descriptorul de fișier (un pointer către structura FILE)

format specifică formatul operației de scriere în fișier

2.c. Închiderea unui fișier text

Închiderea unui fișier care a fost deschis anterior cu **fopen()** se realizează în mod obligatoriu cu ajutorul funcției:

```
int fclose(FILE* descriptor_fisier)
```

unde:

descriptor_fisier este descriptorul de fișier (un pointer către structura FILE) returnat după execuția cu succes a apelului funcției **fopen()**

EXERCITII

1. Scrieți un program care citește de la tastatură un număr întreg și îl afișează în baza 2.
2. Scrieți un program care citește de la tastatură un număr întreg și afișează câte cifre are numărul citit în baza 10.
3. Scrieți un program care citește de la tastatură un sir de numere întregi și îl afișează în ordine crescătoare/descrescătoare.
4. Scrieți un program care citește de la tastatură cuvinte (șiruri de caractere), verifică și, apoi, afișează dacă aceste cuvinte sunt de tip palindrom.
5. Se dă un nume de fișier (definit în segmentul de date). Fișierul conține numerele întregi cu semn separate prin spații. Să se calculeze și să se afișeze suma numerelor din fișier.
6. Se dă un nume de fișier (definit în segmentul de date). Fișierul conține litere mici, litere mari, cifre și caractere speciale. Să se înlocuiască toate caracterele speciale din fișier cu caracterul 'X'.
7. Se dă un nume de fișier (definit în segmentul de date). Fișierul conține cuvinte (șiruri de caractere separate prin spații). Să se determine și să se afișeze numărul de cuvinte din fișier.
8. Se dă un nume de fișier (definit în segmentul de date). Fișierul conține propoziții (șiruri de caractere separate prin spații sau punct). Să se determine și să se afișeze numărul de propoziții din fișier.

ARHITECTURA SISTEMELOR DE CALCUL

– Seminar 6 –

1. Proceduri scrise în asamblare (subprograme)
2. Programe din mai multe module

1. PROCEDURI SCRISE ÎN ASAMBLARE (SUBPROGRAME)

Limbajul de asamblare nu recunoaște noțiunea de *subprogram*. Putem însă să creăm o secvență de instrucțiuni, care să poată fi apelată din alte zone ale programului și care, la finalul execuției, să returneze controlul programului apelant. O astfel de secvență se numește *procedură*.

1.a. Apelul unei proceduri scrise în asamblare

Apelul unei proceduri se poate realiza, teoretic, și printr-o instrucțiune de salt necondiționat **JMP**. Problema care apare în acest caz este că procesorul nu știe de unde va trebui să reia execuția programului după terminarea procedurii.

De aceea, pe timpul apelului unei proceduri, este necesar să salvăm undeva offsetul instrucțiunii de la care se va relua execuția programului. Locul unde se salvează offsetul de revenire este stiva de execuție.

Apelul unei proceduri scrise în asamblare se realizează cu ajutorul instrucțiunii **CALL** astfel:

call nume_procedura

Instrucțiunea **CALL** execută un „*salt cu revenire*”, prin urmare, execuția sa va produce următoarele efecte:

- se pune în vârful stivei offset-ul instrucțiunii imediat următoare (valoarea curentă din registrul EIP);
- se execută un salt de tip NEAR la eticheta **nume_procedura** (la începutul sevenței de instrucțiuni).

După execuția sevenței de instrucțiuni proprii, orice procedură trebuie să transfere controlul programului apelant. De aceea, în orice procedură, ultima instrucțiune este, în mod obligatoriu, instrucțiunea **RET**.

Execuția instrucțiunii **RET** va produce următoarele efecte:

- se extrage offset-ul de revenire din vârful stivei;
- se execută un salt de tip NEAR la offset-ul extras anterior din stivă.

Exemplu

Considerăm următorul program scris în limbajul C:

```

...
// definitia functiei 1
void functia_1() {
    ...           // liniile de cod care compun corpul functiei
}

// definitia functiei 2
int functia_2() {
    ...           // liniile de cod care compun corpul functiei
    return x;
}

// programul principal
int main(int argc, char* argv[]) {
    ...
    functia_1();           // apelez functia 1
}

```

```

int r = functia_2();    // apelez functia 2
...
return 0;
}

```

În limbaj de asamblare, programul arată astfel:

```

...
segment code use32 class=code

; programul principal
start:

...          ; transmit argumentele
call functia_1 ; apelez functia_1
...          ; eliberez stiva

...          ; transmit argumentele
call functia_2 ; apelez functia_2
...          ; eliberez stiva
...          ; stochez valoarea returnata

push dword 0
call [exit]

; definitia functiei1
functia_1:
...          ; secventa de instructiuni
ret         ; revin in programul principal

; definitia functiei 2
functia_2:
...          ; secventa de instructiuni
ret         ; revin in programul principal

```

1.b. Modalități de transmitere a argumentelor și de returnare a rezultatului

Atunci când apelăm în limbaj de asamblare o funcție din librăria C (librăria de funcții **msvcrt.dll**) este obligatoriu să respectăm convenția de apel **_cdecl**. În cazul procedurilor scrise în asamblare, nu mai suntem constrâniți să respectăm o anumită convenție de apel.

Cel care definește procedura este obligat să stabilească:

- modul și ordinea de transmitere a argumentelor funcției;
- cum va fi returnat rezultatul execuției;
- care sunt regiștri care pot fi modificați pe timpul execuției;
- cine are obligația să „steargă” argumentele (în cazul în care acestea sunt puse pe stivă).

Transmiterea argumentelor unei proceduri scrise în asamblare se poate face în regiștri sau pe stivă.

Procedura poate să returneze rezultatul într-un registru sau pe stivă.

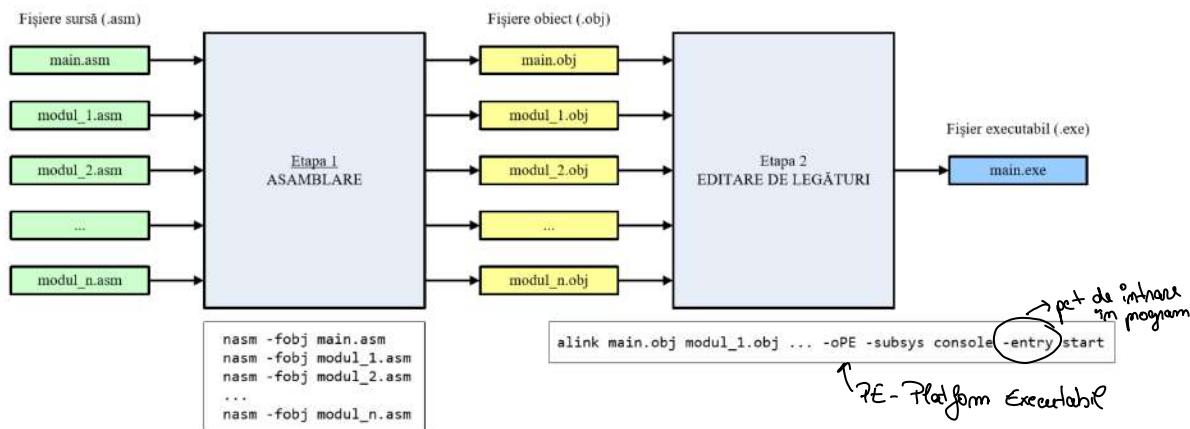
2. PROGRAME DIN MAI MULTE MODULE

Un program scris în limbaj de asamblare poate fi împărțit în mai multe fișiere sursă (*module*), fiecare fiind asamblat separat în *fișiere obiect* (*.obj*).

Pentru scrierea unui program compus din mai multe module trebuie să respectăm următoarele reguli:

→ în NASM sunt public predefinit!

- toate segmentele vor fi declarate cu specificatorul **public**, pentru că în programul final segmentul de cod va fi construit prin concatenarea segmentelor de cod din fiecare modul (la fel și segmentul de date);
- etichetele și numele variabilelor declarate într-un modul și care trebuie „*exportate*” în alte module trebuie să declarate folosind directiva **global**;
- etichetele și variabilele care sunt declarate într-un modul și sunt folosite în alt modul trebuie să fie „*importate*” prin directiva **extern**;
- o variabilă trebuie declarată în întregime într-un modul (nu poate fi jumătate într-un modul și jumătate într-altul);
- transferul controlului execuției dintr-un modul în altul se face cu instrucțiunile: **jmp**, **call** sau **ret**;
- punctul de intrare în program (**global start**) trebuie declarat doar în modulul care conține „*programul principal*”.



Construcția unui program din mai multe module se realizează în două etape:

1. etapa de *asamblare*:

- se realizează cu ajutorul unui program numit *asamblor*;
 - fiecare fișier sursă (modul) va fi asamblat folosind asamblorul *NASM (The Netwide Assembler)*:
- ```

nasm -fobj modul_1.asm
nasm -fobj modul_2.asm
...
nasm -fobj modul_n.asm

```

- în urma procesului de asamblare se va genera câte un fișier obiect (.obj): **modul\_1.obj**, **modul\_2.obj**, ..., **modul\_n.obj**

### 2. etapa de *editare de legături (linkeditare)*:

- se realizează cu ajutorul unui program numit *editor de legături (linkeditor)*;
- toate fișierele obiect (.obj) rezultate în urma asamblării vor fi concatenate într-un singur fișier executabil (.exe):

```
alink modul_1.obj modul_2.obj ... modul_n.obj -oPE -subsys console -entry start
```

La finalul procesului de construcție, se obține un singur fișier executabil pe 32 de biți (.exe) care poate fi rulat de către sistemul de operare Windows.

Cum arată un executabil?  
Căuta linkeditor

EXERCITII

1. Scrieți un program care calculează valoarea expresiei  $x = a+b$ .
2. Scrieți un program care calculează factorialul unui număr natural  $n$  citit de la tastatură.
3. Scrieți un program care concatenează două siruri de caractere citite de la tastatură.

copiată ALINK —  
exemplu sămănătoare

un batch se execută  build.bat  
comandă e comandă

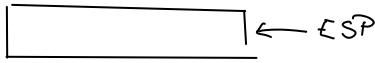
# Seminar — motif —

```
void function1()
{
 int maxm()
 {
 function1()

 function2()

 return 0;
 }
}
```

segment code use 32b class = code  
start:



call function

mov eax,[esp+8]

advise baseata

```
push dwond()\n call [exit]
```

functional:

- 1 -

nut

segment code use 32b class = code  
start:

```
; calculate a+b
 mov eax,[a]
 mov ebx,[b]
 call sum_1
 mov [sum],eax
```

```
push dword [a]
push dword [b]
call sum-2
mov [sum], eax
```

```
push dword 0
push dword [a]
push dword [b]
call sum_3
pop dword [sum]
```

; input: EAX=a, EBX=b  
; output: EAX=a+b

Sum-1 :

```
add eax, edx
ret
```

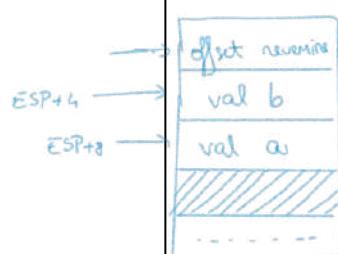
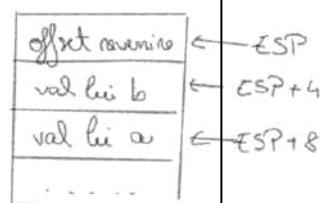
; input pe stivă  
; output  $EAX = a+b$

sum\_2:  
mov eax,[esp+8]  
add eax,[esp+4]

steige zu panam du  
pe steiva

jimpud pe givā  
joutpud pe givā

```
 mov eax,[esp+4]
 add eax,[esp+8]
 mov [esp+12],eax
 ret 2*4
```



ex 2

; concatenare (S1, S2, d)

concatenare:

cld

mov edi, [esp + 4]  
mov esi, [esp + 12]

loop1:

lodsb

cmp al, 0

je maiDepart

stosb

jmp loop1

maiDepart:

rmov edi, [esp + 8]

loop2:

lodsb

cmp al, 0

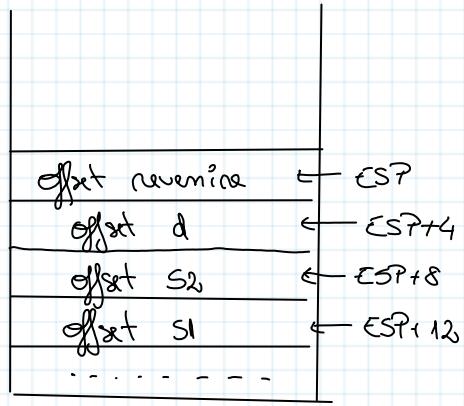
je regine

stosb

jmp loop2

regine:

ret 3\*4



Obs

shl bl, 1 → immultare cu 2

shr bl, 1 → Impartire cu 2

# Multimodul asm+asm vs multimodul asm+C

Page 1 of 2

## Programare multimodul asm+asm versus Programare multimodul asm+C

### Programare multimodul asm+asm

```
Fisier: main.asm
bits 32
// declar procedura definita in asamblare
extern suma

...
segment code use32 class=code
start:
...
; apelul procedurii definite in asamblare
push dword [b]
push dword [a]
call suma
add esp, 2*4
...
push dword 0
call [exit]
```

Stiva de execuție (la intrarea în procedura suma)

|                 |        |
|-----------------|--------|
| offset revenire | <- ESP |
| valoarea lui a  | ESP+4  |
| valoarea lui b  | ESP+8  |
| ...             |        |
| ...             |        |

### Programare multimodul asm+C

```
Fisier: main.c
#include <stdio.h>
// declar procedura definita in asamblare
int suma(int, int);

...
int main(void)
{
 ...
 // apelul procedurii definite in asamblare
 int sum = suma(a, b);
 ...
 return 0;
}
```

Stiva de execuție (după crearea noului cadru de stivă în procedura suma)

|                                    |        |
|------------------------------------|--------|
| Cadrul de stivă al procedurii suma | <- ESP |
| vechea valoare din EBP             | ESP    |
| offset revenire în main()          | EBP+4  |
| valoarea lui a                     | EBP+8  |
| valoarea lui b                     | EBP+12 |
| ...                                |        |
| ...                                |        |

main() → suma() →

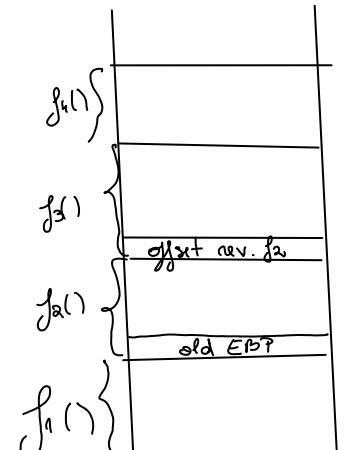
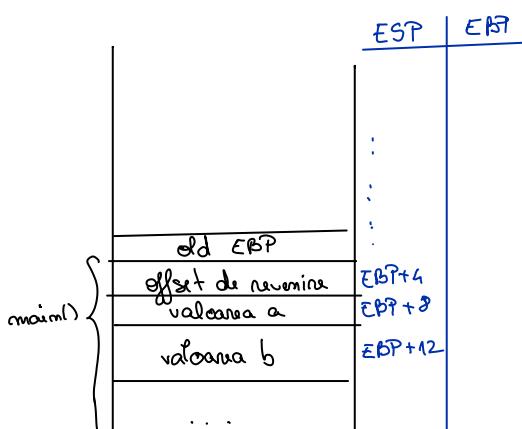
f<sub>1</sub>() → f<sub>2</sub>() → f<sub>3</sub>() → f<sub>4</sub>()

int main()

} ...  
int sum = suma(a,b)

push dword [b]  
push dword [a]  
call \_suma  
add esp, 2\*4

- suma:  
; cod de intrare  
push ebp  
mov ebp, esp  
; facem suma  
mov eax,[ebp + 8]  
add eax,[ebp + 12]  
; cod de ieșire  
mov esp,ebp  
pop ebp  
ret



# Multimodul asm+asm vs multimodul asm+C

Page 2 of 2

Fisier: modul.asm

```
bits 32

; facem vizibila procedura pentru toate modulele
global suma

segment data use32 class=data

segment code use32 class=code
 suma:
 ; obtin valorile argumentelor de pe stiva
 mov eax, [esp+4] ; EAX = a
 mov edx, [esp+8] ; EDX = b

 add eax, edx ; EAX = a+b

 ret
```

Fisier: modul.asm

```
bits 32

; facem vizibila procedura pentru toate modulele
global _suma

segment data use32 class=data

segment code use32 class=code
 _suma:
 ; creez noul cadru de stiva
 push ebp
 mov ebp, esp

 ; obtin valorile argumentelor de pe stiva
 mov eax, [ebp+8] ; EAX = a
 mov edx, [ebp+12] ; EDX = b

 add eax, edx ; EAX = a+b

 ; refac cadrul de stiva al functiei main()
 mov esp, ebp
 pop ebp

 ret
```

? linked din ?  
asm + C ?

