

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Arhitectura microprocesoarelor X86 (IA-32)

---

# Arhitectura Microprocesoarelor x86 (IA-32)

Cu ce categorii de informație ați lucrat la nivel de limbaj de programare?

# Arhitectura Microprocesoarelor x86 (IA-32)

INSTRUCȚIUNI

DATE (tip de date)

ADRESĂ (calcul de adrese)

# Arhitectura Microprocesoarelor x86 (IA-32)

- Limbaj mașină:
  - Un limbaj numeric înțeles în mod specific de procesorul unui computer (CPU)
  - Toate procesoarele x86 înțeleg un limbaj comun al mașinii
- Limbajul mașină – set de numere
- Limbajul de asamblare – instrucțiuni inteligibile
- Fiecare instrucțiune în limbajul de asamblare corespunde unei instrucțiuni unice în limbajul mașină

# Arhitectura Microprocesoarelor x86 (IA-32)

- programarea microprocesoarelor compatibile cu procesoarele Intel IA-32
- procesor x86 => procesor Intel 80386 (1985) => registri pe 32 de biți
- nu este portabil

# Arhitectura Microprocesoarelor x86 (IA-32)

- Avantaje:
  - libertate de declarare sau transfer de date
  - gamă largă de operatori și expresii de adrese
- Dezavantaje:
  - Un număr mare de detalii necesare a fi stabilite înainte de a scrie efectiv cod semnificativ

# Arhitectura Microprocesoarelor x86 (IA-32)

- Microprocesorul
  - Are viteză
  - Combină comenzi de bază (mai multe instrucțiuni de bază pe care procesorul le poate executa)
  - Orice comandă în orice limbaj de programare este *tradusă* într-un set de instrucțiuni pe care procesorul le poate executa
  - Orice program executabil, indiferent de unde provine, se poate vedea în ce se traduce



Astfel putem vedea limitele limbajului de programare

De ce trebuie să declar variabile?

- Orice operație sau instrucțiune este MAXIM binară

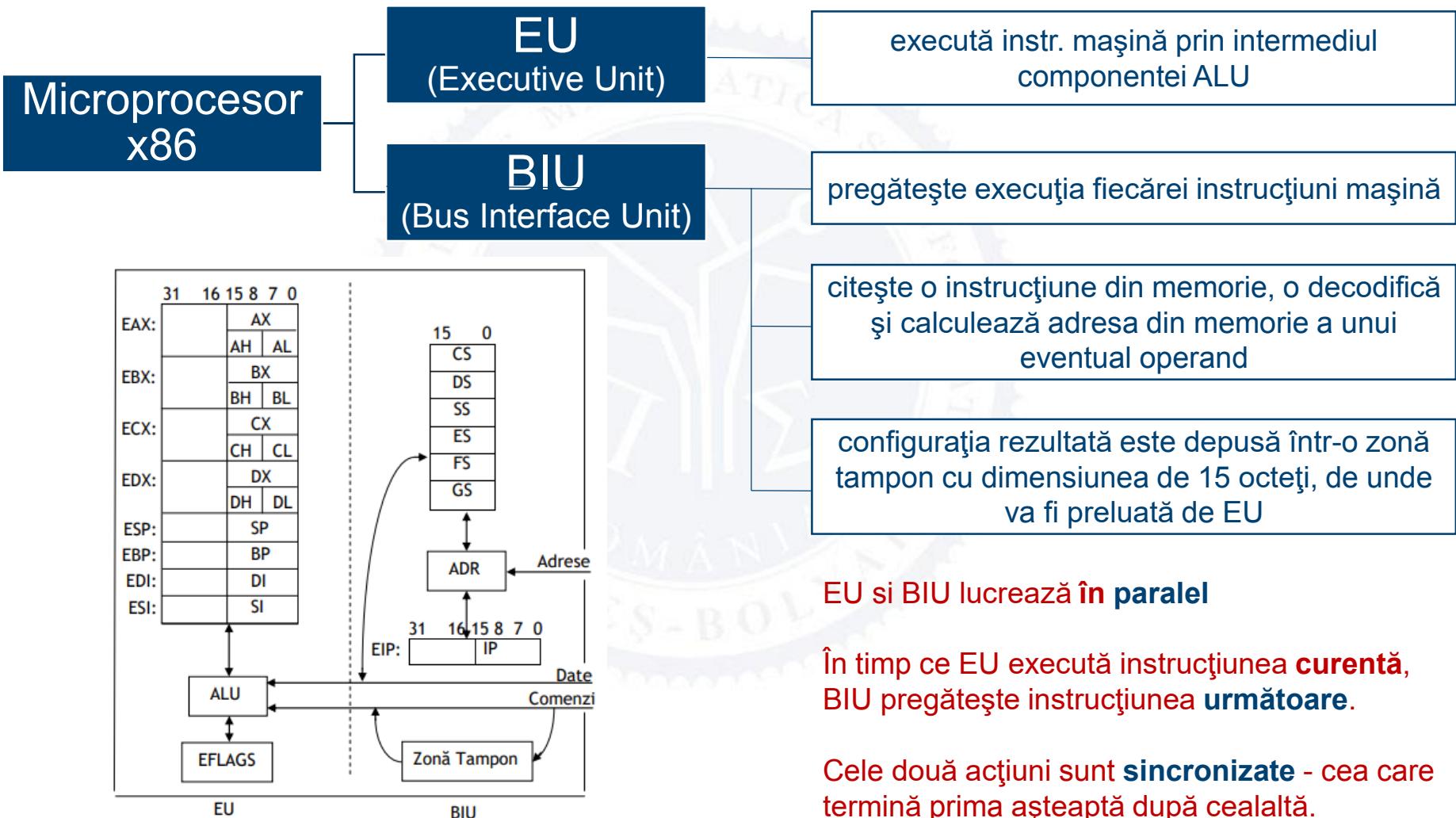
# Structura microprocesorului

Care este unitatea primară de reprezentare a informației?

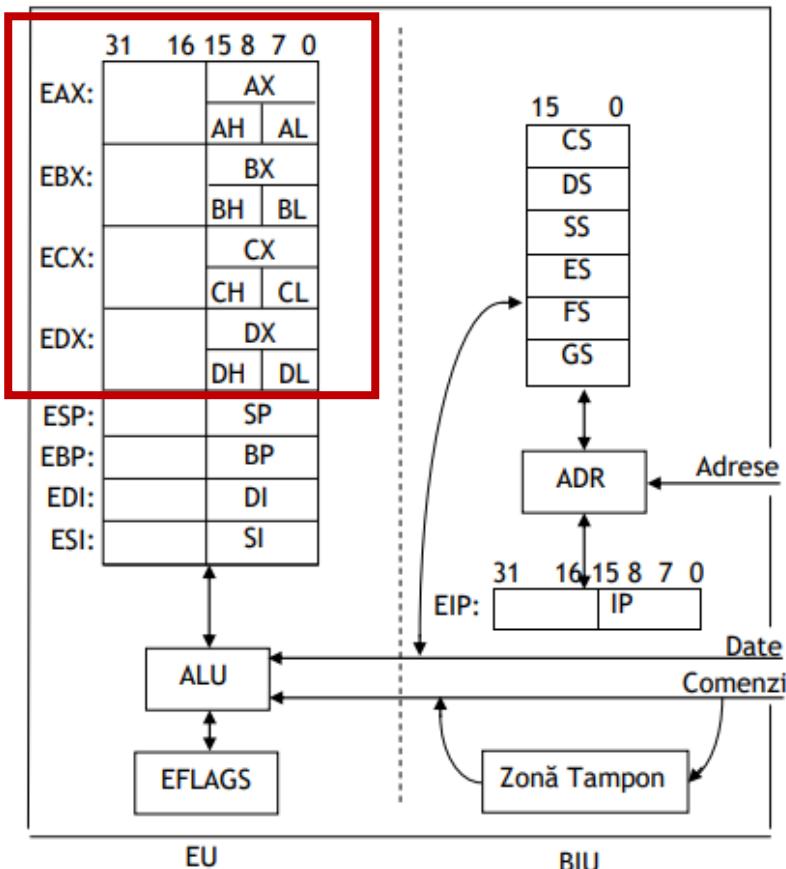
Care este unitatea primară de acces la informație?

Care este diferența dintre REPREZENTARE și INTERPRETARE?

# Structura microprocesorului



# Regiștrii generali EU



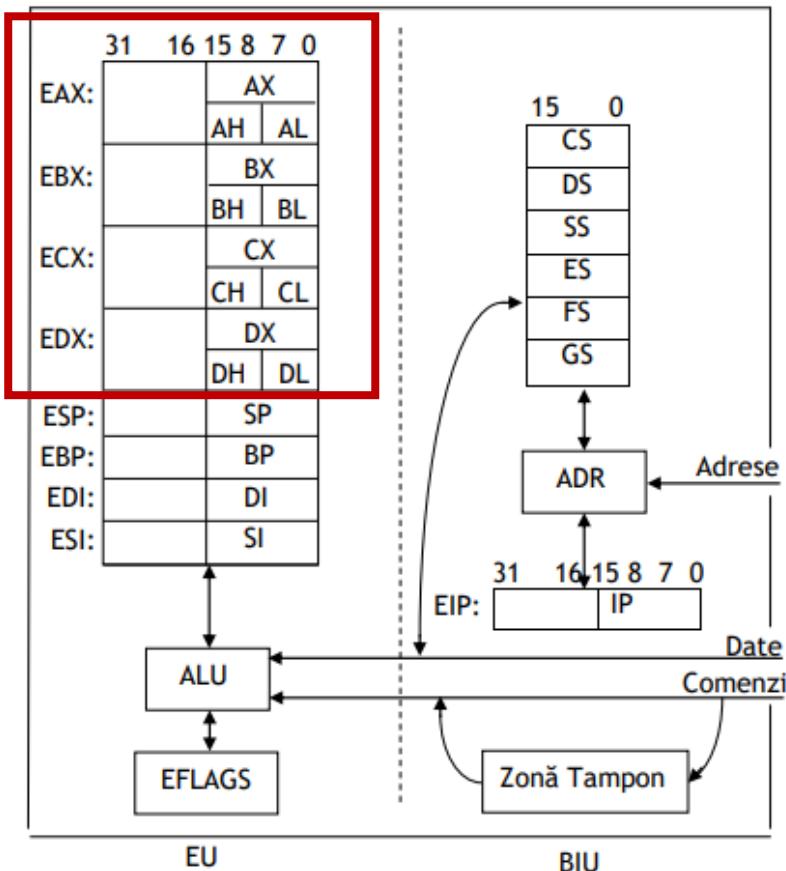
Cum stie procesorul sa distingă între date (variabile / constante), coduri de comenzi sau adrese?

Canale de transmisie (magistrale)

- **A – BUS** – magistrala de adrese
- **C – BUS** – magistrala de comenzi
- **D – BUS** – magistrala de date

Ce înseamnă că un calculator / procesor e pe N biți ?

# Regiștrii generali EU



- **EAX - registru acumulator**
  - este folosit de către majoritatea instrucțiunilor ca unul dintre operanzi.
- **EBX - registru de bază**
  - provine ca denumire de la *Base Register* folosit în aceasta acceptiune la programarea pe 16 biti
- **ECX - registru contor**
  - pentru instrucțiuni care au nevoie de indicații numerice
- **EDX - registru de date**
  - împreună cu EAX se folosește în calculele ale căror rezultate depășesc un dublucuvânt (32 biți).

# Tipuri de date

- BYTE (8 biți)
- WORD (2 octeți, 16 biți)
- DWORD (4 octeți, 32 biți)
- QWORD (8 octeți, 64 biți)

De ce există QUADWORD?

# De ce există QUADWORD?

- La adunare:

op1 (m cifre) +  
op2 (n cifre)

---

suma lor are **max(m,n) + 1 cifre**

- La înmulțire:

op1 (m cifre) \*  
op2 (n cifre)

---

produsul lor are **m+n cifre**

# Consecințe la nivelul arhitecturii

byte + byte => byte

word + word => word

dword + dword => dword

byte \* byte => word

word \* word => dword

dword \* dword => **qword**

## EDX : EAX (qword)

EDX	EAX
Jumătatea superioară (32 biți)	Jumătatea inferioară (32 biți)

# Regiștrii generali EU



De ce începe numerotarea bițiilor de la 0?

De ce se face numerotarea de la dreapta la stânga?

# Regiștrii generali EU



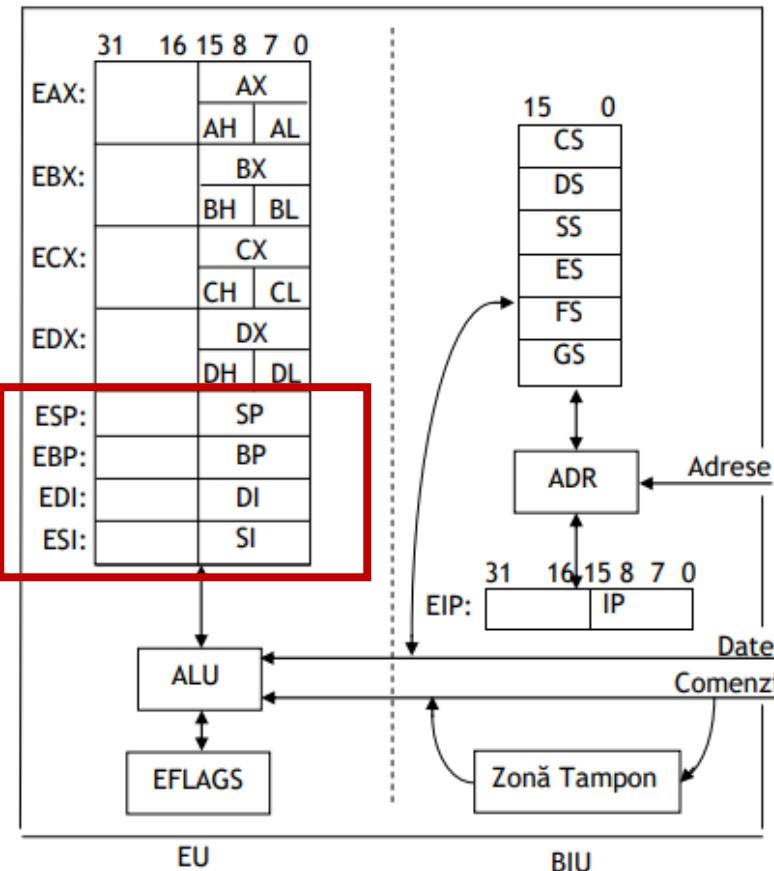
Ce numărăm în cadrul limbajelor de programare?

# Regiștrii generali EU



Cum putem accesa partea superioară a regiștrilor EAX, EBX, ECX EDX?

# Regiștrii generali EU – regiștrii de stivă

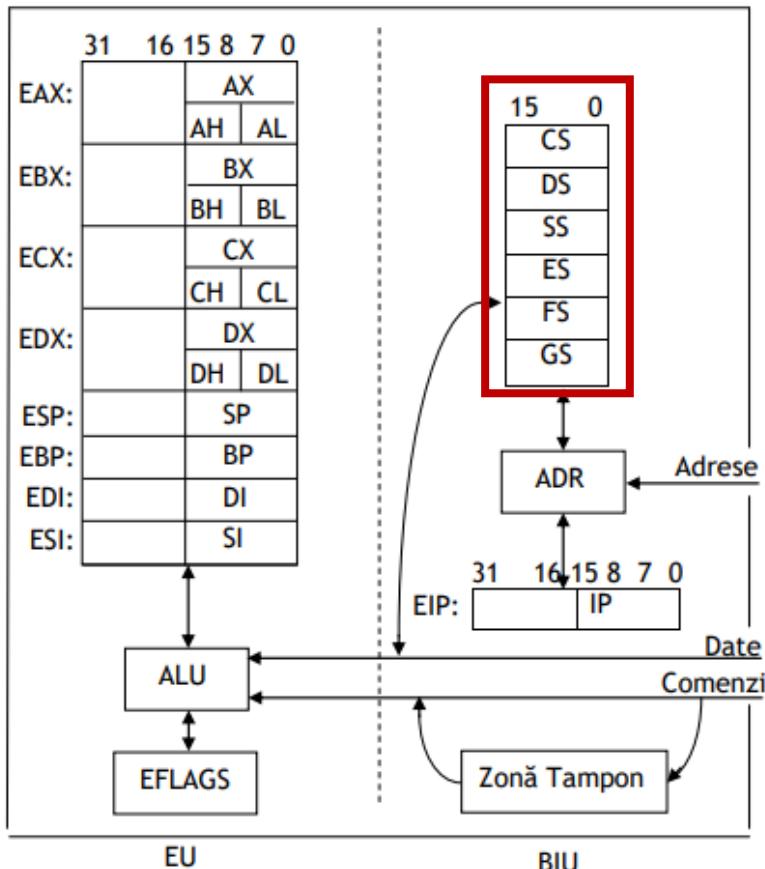


- **ESP – registru de stivă (Stack Pointer)**
  - punctează spre elementul ultim introdus în stivă (elementul din vârful stivei).
- **EBP – registru de stivă (Base pointer)**
  - punctează spre primul element introdus în stivă (indică baza stivei).

Ce este stiva? Prin ce diferă stiva de coadă?

Ce tipuri de date avem?

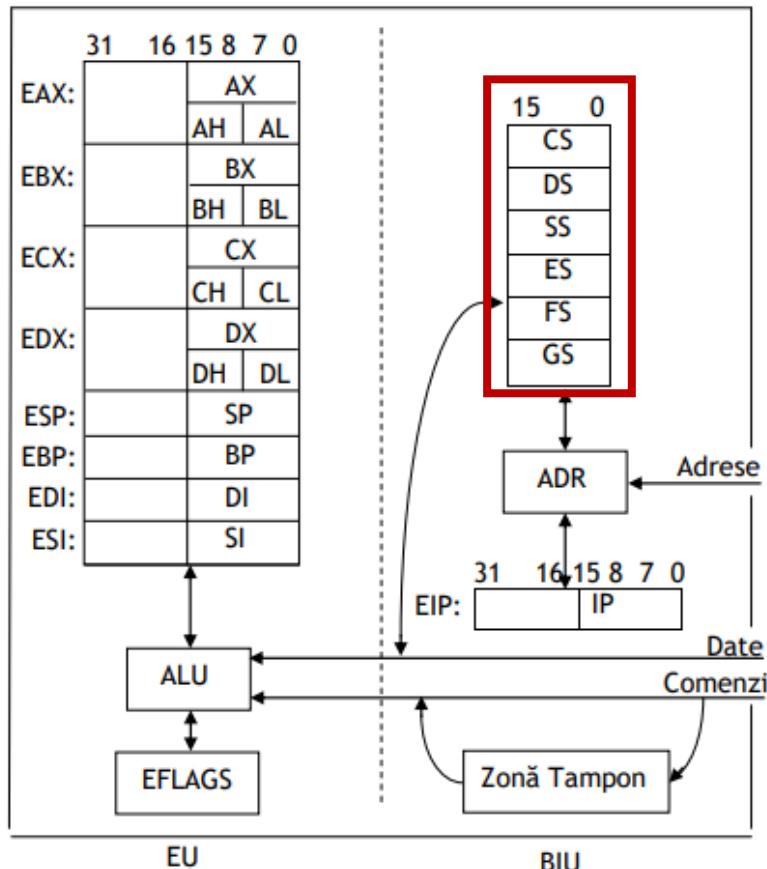
# Regiștrii de segment



- **CS** - *Code segment*
- **DS** - *Data Segment*
- **SS** - *Stack Segment*
- **ES** - *Extra Segment*
- **FS**
- **GS**

De ce procesorul se ocupă de stivă cu 3 regiștrii și nici unul nu face referire la coadă, heap etc.?

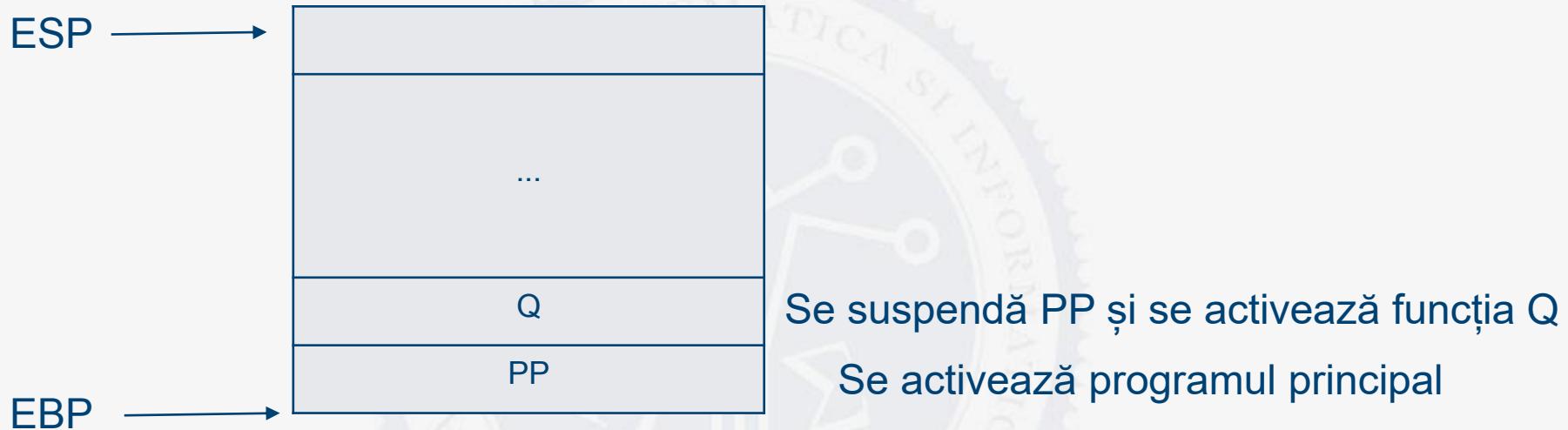
# Regiștrii de segment



- **CS** - *Code segment*
- **DS** - *Data Segment*
- **SS** - *Stack Segment*
- **ES** - *Extra Segment*
- **FS**
- **GS**

De ce procesorul știe de LIFO și nu e interesat de FIFO?

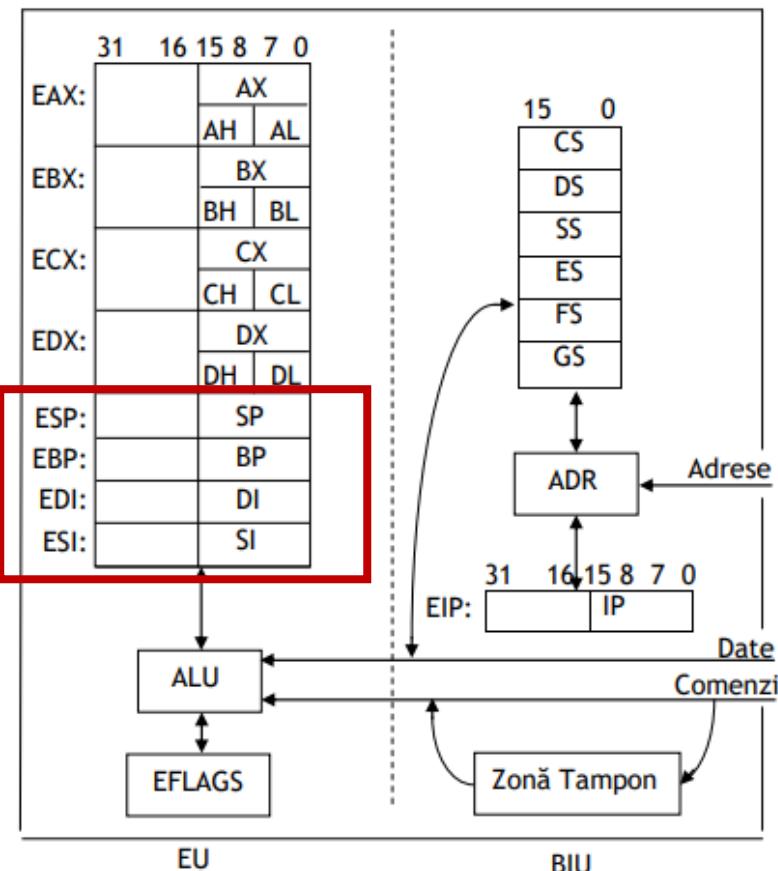
# Stiva de execuție (runtime stack)



Necesar pentru a ști unde a început programul

Rostul limbajelor de asamblare este legătura cu limbajele de programare de nivel înalt (ex. Programare C + ASM)

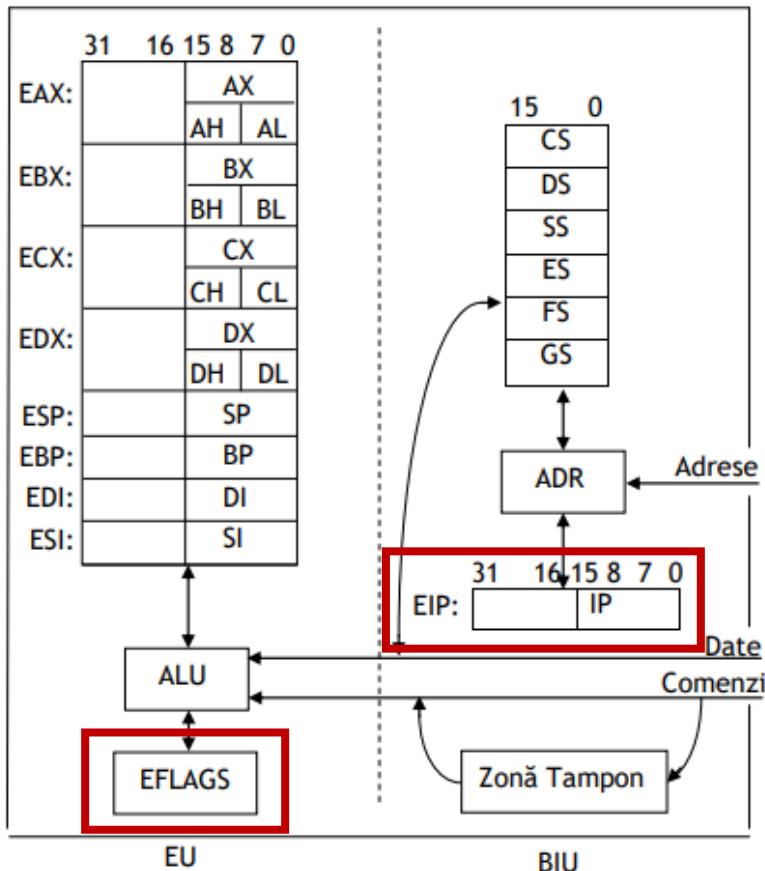
# Regiștrii generali EU – regiștrii de index



- EDI și ESI – **regiștrii de index** (*Destination Index și Source Index*)
  - utilizați de obicei pentru accesarea elementelor din siruri de octeți sau de cuvinte.

a [5432] – cine e indexul?

# Regiștri EFLAGS și EIP



## • EFLAGS

- O configurație a registrului de flaguri indică un rezumat sintetic a execuției fiecărei instrucțiuni.

## • EIP

- conține adresa următoarei instrucțiuni de executat

# Arhitectura Microprocesoarelor x86 (IA-32)

- Reprezentarea datelor:

- Cea mai mică unitate de stocare a informației este BIT-ul (0 sau 1)
- **byte** (octet) = 8 biți
- **word** (cuvânt) = 2 octeți = 16 biți
- **doubleword** (dublucuvânt) = 2 cuvinte = 4 octeți = 32 biți
- **quadword** (tetracuvânt) = 2 dublucuvinte = 4 cuvinte = 8 octeți = 64 biți

# Arhitectura Microprocesoarelor x86 (IA-32)

- Cuvinte rezervate:
  - Numele de instrucțiuni: MOV, ADD, MUL, DIV
  - Numele regiștrilor
  - Numele directivelor
  - Operatori
- Identifieri (variabile, constante, proceduri, etichete):
  - pot contine maxim 247 caractere, litere, cifre și caracterele \_, \$, \$\$, #, @, ~, . și ?
  - în NASM sunt case sensitive
  - primul caracter trebuie să fie o literă (A..Z, a..z), \_ și ?
  - nu poate fi un cuvant rezervat
- Pentru denumirile implicit parte a limbajului, cum ar fi cuvintele cheie, mnemonicile și numele regiștrilor, nu se diferențiază literele mari de cele mici (acestea sunt case insensitive).

# Arhitectura Microprocesoarelor x86 (IA-32)

- Directive:
  - sunt indicații date asamblorului în scopul generării corecte a octetilor

Ex: relații între modulele obiect, definirea unor segmente, indicații de asamblare condiționată, directive de generare a datelor (*pot defini variabile, macro-uri, proceduri sau pot atribui nume segmentelor de memorie*)

## Directive pentru definirea variabilelor:

DB, DW, DD, DQ

## Directive pentru definirea constantelor:

EQU

**Exemplu:** zece EQU 10

# Arhitectura Microprocesoarelor x86 (IA-32)

- Etichete:

- Un identificator care reprezintă adresa unor instrucțiuni sau date
  - Etichete de date: identifică locația unei variabile

```
count DW 100
array DW 1024, 2048
        4096, 8192
```

- Etichete de cod

```
L1: mov AX, BX
L2:
    mov AX, BX
    ...
jmp L2
```

# Arhitectura Microprocesoarelor x86 (IA-32)

- Reprezentarea datelor în memorie: little endian
  - O adresă referă o locație de memorie
  - Procesorul x86:
    - permite ca fiecare octet să fie accesibil prin adresă
    - stochează și furnizează datele din memorie folosind reprezentarea little endian
      - Cel mai puțin semnificativ octet este stocat în memorie la ce mai mică adresă alocată
      - Octetii rămași sunt stocați în memorie în octetii următori

## Exemplu:

a DB 12h	;  12h
b DW 3456h	;  56h 34h
c DD 7890ABCDh	;  CDh Abh 90h 78h
d DQ 1122334455667788h	;  88h 77h 66h 55h 44h 33h 22h 11h

# Arhitectura Microprocesoarelor x86 (IA-32)

OllyDbg - database.exe - [CPU - main thread, module database]

File View Debug Trace Plugins Options Windows Help

Registers (FPU)

EAX	0019FFCC
ECX	00402000 database.<ModuleEntryPoint>
EDX	00402000 database.<ModuleEntryPoint>
EBX	0028B000
ESP	0019FF74
EBP	0019FF80
ESI	00402000 database.<ModuleEntryPoint>
EDI	00402000 database.<ModuleEntryPoint>
EIP	00402000 database.<ModuleEntryPoint>
C	0 ES 002B 32bit 0(FFFFFFFF)
P	1 CS 0023 32bit 0(FFFFFFFF)
A	0 SS 002B 32bit 0(FFFFFFFF)
Z	1 DS 002B 32bit 0(FFFFFFFF)
S	0 FS 0053 32bit 28E000(FFF)
T	0 GS 002B 32bit 0(EFFFFFFF)

Stack [0019FF70]=0  
Imm=0

Address Hex dump

00401000	12 56 34 CD AB 90 78 88 77 66 55 44 33 22 11 00
00401010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0019FF74 77A2F989 éaaaw RETURN to KERNEL32.BaseThreadIn

0019FF78	0028B000 a  (a
0019FF7C	77A2F978 paaw KERNEL32.BaseThreadInitThunk
0019FF80	0019FFDC ma a
0019FF84	77BA74B4  taw RETURN to ntdll.77BA74B4
0019FF88	0028B000 a  (a
0019FF8C	376DF9F9 aa m
0019FF90	00000000 aaaa
0019FF94	00000000 aaaa
0019FF98	0028B000 a  (a
0019FF9C	00000000 aaaa

Data representation in memory

byte (in data segment) a db 12h  
word (in data segment) b dw 3456h  
doubleword (in data segment) c dd 7890ABCDh  
quadword (in data segment) d dq 1122334455667788h

# Arhitectura Microprocesoarelor x86 (IA-32)

- Instrucțiuni:

- Sunt translate de către asamblor în limbaj mașină și apoi sunt încărcate și executate de către CPU la runtime
- Formatul unei linii sursă

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

NUME\_INSTRUCIUNE destinatie, sursa

destinatie = primul operand

sursa = al doilea operand

## Alte formate de instructiuni:

NUME\_INSTRUCIUNE operand1, operand2

NUME\_INSTRUCIUNE operand

NUME\_INSTRUCIUNE

# Arhitectura Microprocesoarelor x86 (IA-32)

- Un program în limbaj de asamblare este format din segmente logice:
  - Segmentul de cod: instrucțiuni
  - Segmentul de date: variabile
  - Stiva: parametrii procedurilor, variabile locale, adrese de revenire
- Instrumente necesare:
  - **Notepad++**: scrierea fișierului sursă în limbaj de asamblare (.asm)
  - **NASM**: asamblor, adică programul care citește fișierul sursă și produce fișierul obiect (.obj)
  - **ALINK**: link-editor, adică programul care citește mai multe fișiere obiect și produce fișierul executabil (.exe)
  - **OillyDbg**: debugger

# Abrevieri uzuale

Abreviere	Semnificatie
reg	Un registru pe 8, 16 sau 32 de biti AH, AL, BH, BL, CH, CL, DH, DL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
reg8, reg16, reg32	Un registru identificat prin numarul de biti reg8: AH, AL, BH, BL, CH, CL, DH, DL reg16: AX, BX, CX, DX, SI, DI, BP, SP reg32: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
mem	Un operand din memorie (o variabila)
mem8, mem16, mem32	Un operand din memorie identificat prin numarul de biti
con	Un operand imediat (constanta)
con8, con16, con32	Un operand imediat identificat prin numarul de biti



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# CURS 1 – INTRODUCERE

---

LECT. Dr. Șotropa Diana-Florina

# CE ESTE ARHITECTURA CALCULATOARELOR?

- Sistem de calcul = dispozitiv care lucrează automat, sub controlul unui program memorat, prelucrând date în vederea producerii unor rezultate ca efect al procesării
- Arhitectura unui Sistem de calcul poate fi analizată la nivel structural (care sunt componentele fizice și căile de comunicare între acestea) sau la nivel logic (funcțiile fiecărei componente în cadrul structurii)

# CE ESTE ARHITECTURA CALCULATOARELOR?

- **Arhitectura calculatoarelor** studiază modul în care sunt proiectate și organizate componente hardware ale unui calculator și cum interacționează acestea cu software-ul.
- Proiectarea logică a unui sistem de calcul: cum sunt structurate CPU-ul, memoria, magistralele și dispozitivele de intrare/iesire.
- Interfața hardware–software: cum scrie un programator cod care ajunge să fie executat de un procesor.
- Modelarea internă a procesorului: registre, unități funcționale, circuite de control, moduri de adresare etc.
- Legătura cu performanța aplicațiilor: ce se întâmplă când rulăm un program, cum influențează arhitectura viteza de execuție.

# Structura CURS

1. Reprezentarea datelor: Tipuri de date elementare, reprezentari binare si ordini de plasare, organizarea si memorarea datelor
2. Codificarea caracterelor, codificarea numerelor întregi, convenție cu semn si fara semn, bitul de semn, cod complementar, operatii aritmetice, conceptul de depasire, conversia la o locație de alte dimensiuni
3. Performantele unui SC, dimensiunea unui microprocesor, arhitectura microprocesorului 80x86 – structura, registrii, calculul de adresa, moduri de adresare, adrese FAR si NEAR
4. Unitatea executiva (EU) a microprocesorului 80x86: rolul si functiile registrilor si al flagurilor. Clasificare (Registrii si Flaguri) si studii de caz.
5. Unitatea BIU a microprocesorului 80x86: registrii de adresa, registrii de segment, reprezentarea instrucțiunilor. Formula despecificare offset pe 32 biți si formula de specificare offset pe 16 biți. Aritmetică de pointeri
6. Elementele limbajului de asamblare: formatul unei linii sursa, expresii, tipuri de accesare a operanzilor, operatori aritmetici, operatori pe biti, operatori de tip si tipuri de date asociate operanzilor, contorul de locații. Conversii nondestructive (si operatorii specifici)
7. Directive standard pentru definirea segmentelor. Directive pentru definirea datelor. Directive de generare a datelor. Directivele EQU și INCLUDE

# Structura CURS

8. Instructiuni ale limbajului de asamblare: instructiuni de transfer, conversii, operatii aritmetice cu semn si fara semn, operatii de deplasare si rotire de biti, operatii logice pe biti
9. Impactul reprezentării little endian asupra accesării datelor. Constante de tip string. Reprezentare în memorie și utilizare în cadrul unor instrucțiuni de transfer.
10. Instructiuni de salt conditionat si neconditionat, instructiuni de ciclare, instructiuni pe siruri. Conceptul de depășire și modul în care arhitectura 80x86 reactioneaza la aparitia acestei situatii
11. Reprezentarea instrucțiunilor mașină. Formatul intern al unei instrucțiuni. Prefixe de instrucțiuni.
12. Programarea multimodul ASM-ASM: directivele de import-export GLOBAL si EXTERN. Legarea de module scrise in limbaj de asamblare si modul de comunicare intre acestea
13. Implementarea apelului de subprograme. Convenții de apel: CDECL și STDCALL, cod de apel, cod de intrare, cod de iesire la nivelul limbajelor de nivel inalt vs. limbaj de asamblare
14. Legarea de module NASM cu module scrise în limbi de nivel înalt (studiu de caz – programarea C). Exemple și discuții pentru apeluri recursive

# METODOLOGIE

- Software utilizat: NASM
- Debugger: OLLYDBG – deoarece limbajul de asamblare nu are instrucțiuni de intrare sau de ieșire
  - asm16 – are, dar nu se mai folosesc
  - asm32 – folosește apeluri sistem C

# EVALUARE

Mai multe detalii pe site:

<https://www.cs.ubbcluj.ro/~diana.sotropa/teaching/asc/>

- EVALUARE:

- 15% nota activitate laborator (minim nota 5)
- 15% nota lucrare de control  
**La CURS, 8 decembrie 2025**
- 15% nota probă practică (minim nota 5)  
**5 - 18 ianuarie 2026**
- 55% nota examen scris (minim nota 5)  
**19 ianuarie – 8 februarie 2026,**  
**21 ianuarie 2026**  
**a doua data de examen preferabil în perioada 26-30.01.2026**  
**16 - 22 februarie 2026**  
**16 februarie 2026**

- SEMINAR:

- se admit maximum 2 absențe NEMOTIVATE

- LABORATOR:

- se admit maximum 2 absențe NEMOTIVATE
- teste practice în săptămânilile 5 și 9

# DE CE STUDIEM ASC?

Viziuni de  
predare

Viziune  
hardware  
(politehnica)

Viziune  
software  
(development)

# DE CE ESTE IMPORTANT ACEST CURS?

- De ce limbajele de programare la nivel general (C, C++) au limitările pe care le au?
- Fundamentul tehnic pentru orice programator serios
  - Nu poți optimiza un program dacă nu înțelegi cum funcționează procesorul și memoria.
- Baza pentru programarea de sistem (C, ASM, OS development)
  - Înveți cum interacționează codul cu hardware-ul la nivelul cel mai de jos.
- Înțelegerea performanței aplicațiilor
  - De ce unele bucle sunt lente? Cum afectează accesul la memorie timpul de execuție?
- Control total asupra execuției codului
  - Limbajul de asamblare îți oferă o precizie imposibilă în limbajele de nivel înalt.
- Introducere în domenii precum securitate cibernetică, reverse engineering, exploatari
  - Mulți hackeri etici, analiști de malware și ingineri de securitate încep de aici.
- Aplicabil în embedded systems și IoT
  - Dispozitivele mici (ex: Arduino, ESP32) au nevoie de cod eficient scris într-un limbaj de programare low-level.
- Te ajută să înțelegi limitările hardware-ului
  - Ce înseamnă „buffer overflow”? Ce face un „stack frame”? Acestea nu mai sunt mistere.
- Punte între hardware și software
  - Poți comunica eficient atât cu dezvoltatorii hardware, cât și cu cei software
- Pregătire pentru laboratoare și proiecte complexe
  - Fără această bază, laboratoarele din anii următori (ex: Sisteme de Operare) vor părea mult mai dificile.



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Complementul față de 2

---

# Complementul față de 2

- Matematic, REPREZENTAREA unui număr NEGATIV în **complement față de doi** este valoarea  $2^n - V$ , unde  $V$  este valoarea absolută a numărului reprezentat.

**Motto:**

**Cu complementul față de 2:**

**“Interpretăm reprezentări și reprezentăm interpretări”**

Care sunt situațiile în care se folosește complementul față de 2?

# Complementul față de 2

$1001\ 0011b = 93h = 147$  (în interpretarea FĂRĂ SEMN)

Q1: Fiind un număr care începe cu 1, în interpretarea CU SEMN este un număr negativ. Care este valoarea lui?

Valoarea sa este:

– (complementul față de 2 al configurației binare inițiale) =  $-(2^n - V)$

# Complementul față de 2

$1001\ 0011b = 93h = 147$  (în interpretarea FĂRĂ SEMN)

Q2: Cum se obține complementul față de 2 al unui număr?

**Varianta 1 (Oficială): Se scade binar conținutul (evidenț binar) al locației de complementat din 100 ...00, unde numărul de după cifra binară 1 are atâtea zerouri câți biți are locația de complementat.**

$$\begin{array}{r} 1\ 0000\ 0000b - \quad 2^n - \\ 1001\ 0011b \\ \hline 0110\ 1101b \end{array} \quad \begin{array}{l} V \\ = 6Dh = 6 * 16 + 13 = 96 + 13 = 109 \\ \text{(deci complementul față de 2 pe 8 biți al numărului 147 este 109)} \end{array}$$

Ca urmare,  $1001\ 0011b = -109$  (în interpretarea CU SEMN)

# Complementul față de 2

$1001\ 0011b = 93h = 147$  (în interpretarea FĂRĂ SEMN)

Q2: Cum se obține complementul față de 2 al unui număr?

**Varianta 2 (derivată din definiția complementului față de 2 – mai rapidă dpdV practic): 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.**

Inversăm valorile tuturor bițiilor valorii  $1001\ 0011b$  obținând  $01101100b$  după care adăugăm 1 la valoarea obținută:  $0110\ 1100b + 1 = 0110\ 1101b = 109$

Ca urmare,  $1001\ 0011b = -109$  (în interpretarea CU SEMN)

# Complementul față de 2

$1001\ 0011b = 93h = 147$  (în interpretarea FĂRĂ SEMN)

Q2: Cum se obține complementul față de 2 al unui număr?

**Varianta 3 (MULT mai rapidă dpdv practic, cea mai rapidă pt obținerea configurației binare a complementului față de 2): Se lasă neschimbați biții începând din dreapta reprezentării binare până la primul bit 1 inclusiv, iar restul biților se inversează.**

Plecând de la  $1001\ 0011b$  lăsăm neschimbați biții din dreapta până la primul bit 1 inclusiv, iar restul biților se inversează, deci obținem  $0110\ 1101b = 6Dh = 109$

Ca urmare,  $1001\ 0011b = -109$  (în interpretarea CU SEMN)

# Complementul față de 2

$1001\ 0011b = 93h = 147$  (în interpretarea FĂRĂ SEMN)

Q2: Cum se obține complementul față de 2 al unui număr?

**Varianta 4 (CEA mai rapidă dpdv practic, dacă ne interesează doar valoarea absolută în baza 10 a complementului față de 2): Suma valorilor absolute a celor două valori complementare este cardinalul mulțimii reprezentabile pe acea dimensiune.**

Pe 8 biți se pot reprezenta  $2^8$  valori = 256 valori ( $[0,255]$  sau  $[-128,+127]$ )

Pe 16 biți se pot reprezenta  $2^{16}$  valori = 65536 valori ( $[0,65535]$  sau  $[-32768,32767]$ )

Pe 32 biți se pot reprezenta  $2^{32}$  valori = 4294967296 valori ...

Pe 8 biți complementul față de 2 al lui  $1001\ 0011b = 93h = 147$  este  $256 - 147 = 109$

Ca urmare,  $1001\ 0011b = -109$  (în interpretarea CU SEMN)

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a).  $0110\ 1101b$  b). -109 c). 6Dh d). +147

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a).  $0110\ 1101b$  b). -109 c). 6Dh d). +147

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

Care este valoarea în interpretarea cu semn a lui  $93h$  ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

Care este valoarea în interpretarea cu semn a lui  $93h$  ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a). 0110 1101b **b). -109** c). 6Dh d). +147

Care este valoarea în interpretarea cu semn a lui  $93h$  ?

- a). 0110 1101b **b). -109** c). 6Dh d). +147

Care este valoarea în interpretarea cu semn a lui 147 din baza 10 ?

- a). 0110 1101b b). -109 c). 6Dh d). +147

# Complementul față de 2

Care este valoarea în interpretarea cu semn a lui  $1001\ 0011b$ ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

Care este valoarea în interpretarea cu semn a lui  $93h$  ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

Care este valoarea în interpretarea cu semn a lui 147 din baza 10 ?

- a).  $0110\ 1101b$  b). **-109** c).  $6Dh$  d).  $+147$

Nici unul dintre răspunsuri, deoarece întrebarea este un nonsens – pentru că 147 ESTE DEJA O INTERPRETARE

# Complementul față de 2

- Deci 147 și 109 sunt două valori complementare:
  - $1001\ 0011b = 147$  sau  $-109$   
în funcție de interpretare, adică complementul lui 147 este  $-109$
- Este  $-147$  complementul lui 109?

# Complementul față de 2

- Deci 147 și 109 sunt două valori complementare:
  - $1001\ 0011b = 147$  sau  $-109$   
în funcție de interpretare, adică complementul lui 147 este  $-109$
- Este  $-147$  complementul lui  $109$ ? **NU**

# Complementul față de 2

- Care este complementul lui 109?

**Toată discuția despre valori complementare are sens doar dacă atenția noastră se concentrează pe problematica NUMERELOR NEGATIVE**

Ex1: se pleacă de la o reprezentare în baza 2 care începe cu 1 și ne întrebăm care va fi numărul negativ asociat în interpretarea CU SEMN

Ex2: se pleacă de la o valoare absolută (109 sau 147) și ne întrebăm care este reprezentarea în baza 2 pentru numărul -109 sau pentru -147 !!

→ **Totul se învârte în jurul numerelor NEGATIVE !!**

# Complementul față de 2

- Interpretările cu semn și fără semn ale oricărei configurații binare care începe cu 1 VOR FI INTOTDEAUNA DIFERITE și ele NU vor fi NICIODATA părți ale aceluiași interval de reprezentare admisibil pe N biți !!!
- Valorile absolute ale celor două interpretări reprezintă două valori complementare.

Exemple:

-128, 128 ( $-128 \in [-128, 127]$  (cu semn);  $+128 \in [0, 255]$  (fără semn))

147, -109

-1, 255

-3, 253

-127, 129

Contraexemplu:

127, -129 nu sunt complementare ( $-129 \notin [-128, 127]$  (cu semn);  $+127 \in [0, 255]$  (fără semn))

# Complementul față de 2

Q3: Dacă avem o REPREZENTARE în baza 2 de tip **0xxx....** de valoare **+abc** în interpretarea FARA semn, ce valoare va avea această REPREZENTARE în interpretarea CU SEMN din BAZA 10 ? (**b<sub>2</sub> – b<sub>10</sub>**)

**Tot atâta !**

Un număr care începe cu 0 în baza 2 are aceeași valoare atât în interpretarea cu semn cât și în cea fără semn, fiind un număr pozitiv

109 este tot 109 în ambele interpretări

# Complementul față de 2

Q4: Dacă avem o REPREZENTARE de forma  
 $0xxx\dots$  de valoare  $+abc$ ,

care va fi REPREZENTAREA binară a valorii  $-abc$ ? ( $b_2 - b_2$ )

Ex: dacă plecăm de la **109**, cum se reprezintă binar **-109**?

Abia acum începe “complementul față de 2 să joace un rol”, iar răspunsul este:

**REPREZENTAREA sa va fi  
“complementul față de 2 al configurației binare inițiale”.**

Pentru valoarea **109 = 0110 1101<sub>b</sub>**, complementul față de 2 a lui **0110 1101<sub>b</sub>** este **1001 0011<sub>b</sub>**, deci **-109 = 1001 0011<sub>b</sub>**

Ca urmare putem concluziona că  
**valoarea complementară a unui întreg care începe cu 0, va începe cu 1**  
(excepție făcând doar valoarea 0)  
și **va încăpea ca valoare complementară în interpretarea CU SEMN**  
**pe aceeași dimensiune de reprezentare ca și valoarea inițială**

**-109 este tot un byte ca și 109**

# Complementul față de 2

Q5: Dacă avem o REPREZENTARE de tip  $1xxx\dots$  de valoare  $+abc$  în interpretarea FĂRĂ semn, ce valoare va avea această REPREZENTARE în interpretarea CU SEMN din baza 10 ? ( $b_2 - b_{10}$ )

**Valoarea sa este: – (complementul față de 2 al configurației binare inițiale).**

Pentru exemplul nostru, avem:

$1001\ 0011b = 147$  (fără semn)

- - (complementul față de 2 al configurației  $1001\ 0011b$ )
- -  $(0110\ 1101b) = -109.$

# Complementul față de 2

Q6: Dacă avem o REPREZENTARE de forma  $1xxx\dots$  de valoare  $+abc$ , care va fi REPREZENTAREA binară a valorii  $-abc$ ? ( $b_2 - b_2$ )

Ex: dacă plecăm de la  $1001\ 0011_b = +147$ , cum se reprezintă binar  $-147$ ?

Răspunsul nu poate fi decât unul similar cu cel de la Q4:

**REPREZENTAREA sa va fi “complementul față de 2 al configurației binare inițiale”.**

Numai că: **dacă un număr începe cu 1** în reprezentarea lui în baza 2 și are valoarea  $+abc$  în interpretarea FARA SEMN, atunci tot cu **1** va trebui să înceapă și varianta lui negativă  $-abc$  ca REPREZENTARE (pentru că în caz contrar nu ar mai fi un număr negativ în interpretarea CU SEMN).

Dar, complementarea unei valori binare de forma  $1xxx\dots$  va furniza prin complementare în mod natural o valoare binară CARE INCEPE CU 0 pe o dimensiune de reprezentare identică cu cea de pornire !!! - excepție făcând DOAR valorile de forma  $100\dots$  ( $-128$ ,  $+128$ ,  $-32768$ ,  $+32768$  etc).

**Ca urmare, concluzionăm că dacă plecăm de la o reprezentare de forma  $1xxx\dots$  de valoare  $+abc$  NU PUTEM OBȚINE valoarea  $-abc$  PE O ACEEAȘI DIMENSIUNE DE REPREZENTARE !!!!**

# Complementul față de 2

Q6: Dacă avem o REPREZENTARE de forma **1xxx....** de valoare **+abc**, care va fi REPREZENTAREA binară a valorii **- abc** ? ( $b_2 - b_2$ )

Ex: dacă plecăm de la  $1001\ 0011b = +147$ , cum se reprezintă binar  $-147$  ?

Dovada:

**147 = 1001 0011b** ( $147 \in [0..255]$ , însă  $-147 \notin [-128..+127]$  , deci  $-147$  NU este reprezentabil pe un octet chiar dacă  $+147$  este)

**Deci nu numai că, complementarea nu poate fi făcută corect pe aceeași dim. de reprezentare ca și val. inițiale ca METODOLOGIE, dar și analiza intervalelor de reprezentare admisibile ne confirmă asta dpdv SEMANTIC**

# Complementul față de 2

Q6: Dacă avem o REPREZENTARE de forma **1xxxx...** de valoare **+abc**, care va fi REPREZENTAREA binară a valorii **- abc**? ( $b_2 - b_2$ )

Ex: dacă plecăm de la  $1001\ 0011b = +147$ , cum se reprezintă binar  $-147$ ?

Ca urmare, obținerea lui  $-147$  plecând de la **147 = 1001 0011b**, se face astfel:

i). Reprezentarea binară a lui  $147$  începe cu 1, dar trebuie să ținem cont și că  
 **$-147 \notin [-128..+127]$ , însă  $-147 \in [-32768..+32767]$**

de unde rezultă că  $-147$  NU este reprezentabil ca octet CI DOAR CA ȘI CUVÂNT

ii). Pe o dimensiune de tip WORD,  **$147 = 0000\ 0000\ 1001\ 0011b$**  (deci un număr binar care începe cu 0) și conform răspunsului de la Q5, avem că

**$-147 = \text{"complementul față de 2 al configurației binare inițiale"}$** .

Complementul față de 2 a configurației **0000 0000 1001 0011b** este **1111 1111 0110 1101b**, deci

**$-147 = 1111\ 1111\ 0110\ 1101b = FF6Dh$**

*Verificare:  $1111\ 1111\ 0110\ 1101b = FF6Dh = 65389$  în interpretarea FĂRĂ semn, suma valorilor absolute a celor două valori complementare fiind  $65389 + 147 = 65536$  = cardinalul mulțimii numerelor reprezentabile pe 1 WORD, deci cele 2 interpretări ale configurației binare **11111111 01101101** sunt corecte și consistente*

# Complementul față de 2

Format binar număr	Interpretare	Valoare zecimală	În ce fel este implicat “complementul față de 2”	Răspuns
0xxx	Fără semn	+abc	-	-
	Cu semn (Q4)	+abc	Cum se reprezintă – abc?	Complementul față de 2 a lui 0xxx
1xxx	Fără semn	+def	-	-
	Cu semn (Q5)	+def	Ce valoare va avea 1xxx în interpretarea cu semn?	- (Complementul față de 2 a lui 1xxx)
	Cu semn (Q6)	+def	Cum se reprezinta – def ?	Complementul față de 2 a lui 1xx extins fără semn pe 2 * sizeof (1xxx)

(excepție fac reprezentările de forma de forma 100.... (-128, +128, -32768, +32768 etc)

# Complementul față de 2

Format binar număr	Interpretare	Valoare zecimală	În ce fel este implicat “complementul față de 2”	Răspuns
0110 1101b	Fără semn	+109	-	-
	Cu semn (Q4)	+109	Cum se reprezintă – 109?	1001 0011b
1001 0011b	Fără semn	+147	-	-
	Cu semn (Q5)	+147	Ce valoare va avea 1001 0011b în interpretarea cu semn?	- (0110 1101b) = -109
	Cu semn (Q6)	+147	Cum se reprezinta – 147 ?	Complementul față de 2 a lui 0000 0000 1001 0011b care este 1111 1111 0110 1101b

# Complementul față de 2

Care este nr MINIM de biți pe care se poate reprezenta -147 ?

Pe **n** biți se reprezintă  **$2^n$**  valori

- fie valorile  $[0..2^n - 1]$  în interpretarea fără semn
- fie valorile  $[-2^{n-1}..2^{n-1} - 1]$  în interpretarea cu semn

Pe 8 biți se reprezintă  $2^8 = 256$  valori

- fie valorile  $[0..2^8 - 1] = [0..255]$  în interpretarea fără semn
- fie valorile  $[-2^{8-1}..2^{8-1} - 1] = [-128..+127]$  în interpretarea cu semn

Pe 9 biți se reprezintă  $2^9 = 512$  valori

- fie valorile  $[0..2^9 - 1] = [0..511]$  în interpretarea fără semn
- fie valorile  $[-2^{9-1}..2^{9-1} - 1] = [-256..+255]$  în interpretarea cu semn

Așadar, numărul minim de biți pe care se poate reprezenta -147 este 9, iar reprezentarea lui -147 este 1 0110 1101<sub>b</sub>

Deoarece  $512 - 147 = 365 = 16Dh = 1\ 0110\ 1101b$

# Complementul față de 2

**Care este nr MINIM de biți pe care se poate reprezenta 3 ?**

$$3 = 11b \Rightarrow \text{pe } 2 \text{ biți}$$

**Care este nr MINIM de biți pe care se poate reprezenta -3 ?**

Pe 3 biți se reprezintă  $2^3 = 8$  valori

- fie valorile  $[0..2^3 - 1] = [0..7]$  în interpretarea fără semn
- fie valorile  $[-2^{3-1}..2^{3-1} - 1] = [-4..+3]$  în interpretarea cu semn

Deoarece  $8 - 3 = 5 = 101b$  rezultă că  $101b$  este reprezentarea lui  $-3$  pe 3 biți

**Complementul față de 2 se referă la reprezentări sau la interpretări?**



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# FLAG-urile

---

# Flag-urile

- Un flag este un indicator reprezentat pe un bit.
- O configurație a registrului de flaguri indică un rezumat sintetic a execuției fiecărei instrucțiuni.
- Pentru x86 registrul EFLAGS (*the status register*) are 32 biți dintre care sunt folosiți uzuale numai 9.

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

# Flagurile

$$\begin{array}{r} 1001\ 0011b \\ 0111\ 0011b \\ \hline \textcolor{red}{1}\ 0000\ 0110b \end{array} +$$

## • **CF** (*Carry Flag*) - flagul de transport

- are valoarea 1 în cazul în care în cadrul ultimei operații efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar

### Adunare:

```
add regd, regs; regd ← regd + regs
add reg, mem; reg ← reg + mem
add mem, reg; mem ← mem + reg
add reg/mem, con; reg/mem ← reg/mem + con
```

$$\begin{array}{r} 1001\ 0011b \\ 0111\ 0011b \\ \hline \textcolor{red}{1}\ 0000\ 0110b \end{array} +$$

### Exemplu:

```
MOV AL, 10010011b
ADD AL, 01110011b; AL = 00000110b, CF = 1
```

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

## • CF (Carry Flag) - flagul de transport

- are valoarea 1 în cazul în care în cadrul ultimei operații efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului și valoarea 0 în caz contrar

$$\begin{array}{r} 1001\ 0011b \\ + 0111\ 0011b \\ \hline 1\ 0000\ 0110b \end{array}$$

rezultă un transport de cifră seminificativă și valoarea 1 este depusă automat în CF

$$\begin{array}{r} 147 \\ + 115 \\ \hline 262 \end{array}$$

(fără semn)

$$\begin{array}{r} 93h \\ + 73h \\ \hline 106h \end{array}$$

(hexa)

**Flagul CF semnalează depășirea în cazul interpretării FĂRĂ SEMN**

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

## • PF (Parity Flag)

- valoarea lui se stabilește a.î. împreună cu numărul de biți 1 din octetul cel mai puțin semnificativ al reprezentării rezultatului UOE să rezulte un număr impar de cifre 1

### Scădere:

```
sub regd, regs; regd ← regd - regs
sub reg, mem; reg ← reg - mem
sub mem, reg; mem ← mem - reg
sub reg/mem, con; reg/mem ← reg/mem - con
```

### Exemplu:

```
MOV AL, 10001100b
ADD AL, 00000010b; AL = 10001110b, PF = 1
SUB AL, 10000000b; AL = 00001110b, PF = 0
```

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

## • AF (Auxiliary Flag)

- indică valoarea transportului de la bitul 3 la bitul 4 al rezultatului UOE
- de exemplu, în adunarea de mai jos transportul este 0

Exemplu:

MOV AL, 10010011b

ADD AL, 01110011b; **AL = 00000110b, AF = 0**

$$\begin{array}{r} 93h + \\ 73h \\ \hline 106h \\ (\text{hexa}) \end{array}$$

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

## • ZF (Zero Flag)

- primește valoarea 1 dacă rezultatul UOE este egal cu zero și valoarea 0 la rezultat diferit de zero.

### Incrementare / Decrementare:

INC reg/mem; reg/mem = reg/mem + 1

DEC reg/mem; reg/mem = reg/mem - 1

#### Exemplul 1:

```
MOV ECX, 1  
SUB ECX, 1; ECX = 0, ZF = 1
```

#### Exemplul 2:

```
MOV EAX, 0FFFFFFFh  
INC EAX ; EAX = 0, ZF = 1  
INC EAX ; EAX = 1, ZF = 0  
DEC EAX ; EAX = 0, ZF = 1
```

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

## • SF (Sign Flag)

- primește valoarea 1 dacă rezultatul UOE este un număr strict negativ și valoarea 0 în caz contrar.

### Exemplu:

```
MOV BL, 1; BL = 01h  
SUB BL, 2; BL = -1 = FFh, SF = 1
```

**Flagul SF semnalează care e semnul numărului în cazul interpretării CU SEMN**

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

- **TF (Trap Flag)** - flag de depanare
  - dacă are valoarea 1, atunci mașina se oprește după fiecare instrucțiune
- **IF (Interrupt Flag)** - flag de întrerupere
- **DF (Direction Flag)**
  - pt operare asupra sirurilor de octeți sau de cuvinte.
  - dacă are valoarea 0, atunci deplasarea în sir se face de la început spre sfârșit, iar dacă are valoarea 1 este vorba de deplasări de la sfârșit spre început.

Cât e TF, IF, DF în urma execuției unei adunări?

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

- **OF (Overflow Flag)** - flag pentru depăşire CU SEMN

- dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

## Exemplu:

```
MOV AL, -109; AL = 10010011b
ADD AL, 115; AL = 06 = 00000110b, OF = 0
```

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

- **OF (Overflow Flag)** - flag pentru depășire CU SEMN

- dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

$$\begin{array}{r} 1001\ 0011b \\ + 0111\ 0011b \\ \hline 1\ 0000\ 0110b \end{array}$$

$$\begin{array}{r} -109 \\ + 115 \\ \hline 06 \end{array} \quad \begin{array}{r} 93h \\ + 73h \\ \hline 106h \end{array}$$

(cu semn) (hexa)

Nu rezultă un transport de cifră  
seminificativă deci OF = 0

**Flagul OF semnalează depășirea în cazul interpretării CU SEMN**

# Flagurile

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	...	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

- **OF (Overflow Flag)** - flag pentru depăşire CU SEMN

- dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

$$\begin{array}{r} 1001\ 0011b \\ + \\ 0111\ 0011b \\ \hline 1\ 0000\ 0110b \end{array}$$

$$\begin{array}{r} -109 \\ + \\ 115 \\ \hline 06 \end{array}$$

(cu semn)

$$\begin{array}{r} 93h \\ + \\ 73h \\ \hline 106h \end{array}$$

(hexa)

$$\begin{array}{r} 147 \\ + \\ 115 \\ \hline 262 \end{array}$$

(fără semn)

Nu rezultă un transport de cifră seminificativă deci OF = 0

rezultă un transport de cifră seminificativă deci CF = 1

$6 \in [-128, 127] \Rightarrow OF = 0$

REPREZENTARE vs INTERPRETARE

# Categorii de flag-uri

- a) flag-uri ale căror valori sunt setate ca efect direct al execuției Ulltimei Operații Efectuate (UOE) : **CF, PF, AF, ZF, SF și OF**  
*(unde și CUM se tine cont în program de aceste valori ?)*  
**ADC, salturi CONDITIONATE (cea mai frecventă utilizare)**

## Adunare cu transport:

```
ADC regd, regs; regd ← regd + regs + CF
ADC reg, mem; reg ← reg + mem + CF
ADC mem, reg; mem ← mem + reg + CF
ADC reg/mem, con; reg/mem ← reg/mem + con + CF
```

## Adunare de quadwords: **EDX:EAX + ECX:EBX**

```
ADD EAX, EBX; EAX = dublucuvant low
ADC EDX, ECX; EDX = dublucuvant high
```

# Categorii de flag-uri

- a) flag-uri ale căror valori sunt setate ca efect direct al execuției Ullimeii Operații Efectuate (UOE) : **CF, PF, AF, ZF, SF și OF**  
*(unde și CUM se tine cont în program de aceste valori ?)*  
**ADC, SBB, salturi CONDITIONATE (cea mai frecventă utilizare)**

## Scadere cu imprumut:

```
SBB regd, regs; regd ← regd - regs - CF
SBB reg, mem; reg ← reg - mem - CF
SBB mem, reg; mem ← mem - reg - CF
SBB reg/mem, con; reg/mem ← reg/mem - con - CF
```

## Scadere de quadwords: **EDX:EAX - ECX:EBX**

```
SUB EAX, EBX; EAX = dublucuvant low
SBB EDX, ECX; EDX = dublucuvant high
```

# Categorii de flag-uri

- b) flag-uri ale căror valori sunt setate de către programator în vederea influențării modului de operare al instrucțiunilor care urmează acestor setări : **CF, TF, DF și IF**  
*(CUM le setam => prin instrucțiuni specifice)*

**CLC** – efectul fiind **CF=0**  
**STC** – efectul fiind **CF=1**

**CMC** – complementarea valorii din CF

**CLD** – având ca efect **DF=0**  
**STD** – având ca efect **DF=1**

**CLI** – având ca efect **IF=0**  
**STI** – având ca efect **IF=1**

Având în vedere riscul major de setare accidentală a valorii din TF precum și rolul său absolut special în dezvoltarea de depanatoare, NU există disponibile instrucțiuni de acces direct la valoarea din TF !!

# Operații efectuate de ALU (Arithmetic and Logic Unit)

- Operații aritmetice
  - Adunări și scăderi bit cu bit
  - Înmulțirea și împărțirea sunt operații mai costisitoare, și pot fi înlocuite cu adunări / scăderi repetate
- Operații de shiftare pe biți
  - Implică shiftarea locației unui bit spre stânga sau dreapta cu un anumit număr de poziții
  - Se folosesc pentru operații de înmulțire și împărțire
- Operații logice
  - AND, OR, XOR, NOT

# Operații efectuate de ALU (Arithmetic and Logic Unit)

## Instrucțiunea MOVZX (move with zero-extend):

```
MOVZX reg32, reg/mem8  
MOVZX reg32, reg/mem16  
MOVZX reg16, reg/mem8
```

## Exemple

```
byteVal db 10001111b
```

```
MOVZX AX, [byteVal]; AX = 00000000 10001111b
```

```
MOV BX, 0A69Bh; BH = A6h și BL = 9Bh
```

```
MOVZX EAX, BX; EAX = 0000A69Bh
```

```
MOVZX EDX, BL; EDX = 0000009Bh
```

```
MOVZX CX, BL; CX = 009Bh
```

# Operații efectuate de ALU (Arithmetic and Logic Unit)

## Instrucțiunea MOVSX (move with sign-extend):

```
MOVSX reg32, reg/mem8  
MOVSX reg32, reg/mem16  
MOVSX reg16, reg/mem8
```

## Exemple

```
byteVal db 10001111b
```

```
MOVsx AX, [byteVal]; AX = 11111111 10001111b
```

```
MOV BX, 0F69Bh; BH = F6h și BL = 9Bh
```

```
MOVsx EAX, BX; EAX = FFFFA69Bh
```

```
MOVsx EDX, BL; EDX = FFFFFFF9Bh
```

```
MOVsx CX, BL; CX = FF9Bh
```

# Operații efectuate de ALU (Arithmetic and Logic Unit)

## Instrucțiunea XCHG (exchange data):

```
XCHG reg, reg  
XCHG reg, mem  
XCHG mem, reg
```

## Exemple

```
XCHG AX, BX; interschimbă valorile regiștrilor pe 16 biți  
XCHG AH, AL; interschimbă valorile regiștrilor pe 8 biți  
XCHG [var1], BX  
XCHG EAX, EBX
```

```
MOV AX, [val1]  
XCHG AX, [val2]  
MOV [val1], AX; interschimbă două valori din memorie
```

# Operații efectuate de ALU (Arithmetic and Logic Unit)

**Instrucțiunea MUL (înmulțire în interpretarea fără semn):**

MUL op

op = reg/mem8 => MUL reg/mem8 => AL\*reg/mem8 = AX

op = reg/mem16 => MUL reg/mem16 => AX\*reg/mem16 = DX:AX

op = reg/mem32 => MUL reg/mem32 => EAX\*reg/mem32 = EDX:EAX

## Exemplul 1

```
; 3*4  
MOV AL, 3  
MOV BL, 4  
MUL BL ; AL*BL=AX
```

## Exemplul 2

```
; c*4, c - word  
MOV AX, 4  
MUL word [c]  
; AX*[c]=DX:AX
```

## Exemplul 3

```
; a*b, a,b - doubleword  
MOV EAX, [a]  
MUL dword [b]  
; EAX*[b]=EDX:EAX
```

# Operații efectuate de ALU (Arithmetic and Logic Unit)

Instrucțiunea DIV (împărțire în interpretarea fără semn):

DIV op

op = reg/mem8 => DIV reg/mem8 => AX / reg/mem8 = AL rest AH

op = reg/mem16 => DIV reg/mem16 => DX:AX / reg/mem16 = AX rest DX

op = reg/mem32 => DIV reg/mem32 => EDX:EAX / reg/mem32 = EAX rest EDX

## Exemplul 1

; m/n, m – word, n - byte  
MOV AX, [m]  
DIV byte [n]  
; AX / [n] = AL (cat)  
; AX % [n] = AH (rest)

## Exemplul 2

; 5 / r, r – word  
MOV AX, 5  
MOV DX, 0  
DIV word [r]  
; DX:AX / [r] = AX  
; DX:AX % [r] = DX

## Exemplul 3

; EDX:EAX / 12345678h  
MOV ECX, 12345678h  
DIV ECX  
; EDX:EAX / ECX = EAX  
; EDX:EAX % ECX = EDX

# Operații efectuate de ALU (Arithmetic and Logic Unit)

**Instrucțiunea IMUL (înmulțire în interpretarea cu semn):**

IMUL op

op = reg/mem8 => IMUL reg/mem8 => AL\*reg/mem8 = AX

op = reg/mem16 => IMUL reg/mem16 => AX\*reg/mem16 = DX:AX

op = reg/mem32 => IMUL reg/mem32 => EAX\*reg/mem32 = EDX:EAX

## Exemplul 1

```
; 3*(-4)
MOV AL, 3
MOV BL, -4
IMUL BL ; AL*BL=AX
```

## Exemplul 2

```
; c*(-2), c - word
MOV AX, -2
IMUL word [c]
; AX*[c]=DX:AX
```

## Exemplul 3

```
; a*b, a,b - doubleword
MOV EAX, [a]
IMUL dword [b]
; EAX*[b]=EDX:EAX
```

# Operații efectuate de ALU (Arithmetic and Logic Unit)

Instrucțiunea IDIV (împărțire în interpretarea fără semn):

IDIV op

op = reg/mem8 => IDIV reg/mem8 => AX / reg/mem8 = AL rest AH

op = reg/mem16 => IDIV reg/mem16 => DX:AX / reg/mem16 = AX rest DX

op = reg/mem32 => IDIV reg/mem32 => EDX:EAX / reg/mem32 = EAX rest EDX

## Exemplul 1

; m/n, m – word, n - byte  
MOV AX, [m]  
IDIV byte [n]  
; AX / [n] = AL (cat)  
; AX % [n] = AH (rest)

## Exemplul 2

; (-5) / r, r – word  
MOV AX, -5  
MOV DX, 0  
IDIV word [r]  
; DX:AX / [r] = AX  
; DX:AX % [r] = DX

## Exemplul 3

; EDX:EAX / 0A2345678h  
MOV ECX, 0A2345678h  
IDIV ECX  
; EDX:EAX / ECX = EAX  
; EDX:EAX % ECX = EDX



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Analiza conceptului de depăşire (overflow)

---

# Flag-uri

- Următoarele patru instrucțiuni sunt instrucțiuni de transfer al indicatorilor:
  - Instrucțiunea **LAHF** (Load register AH from Flags) copiază indicatorii SF, ZF, AF, PF și CF din registrul de flag-uri în biții 7, 6, 4, 2 și respectiv 0 ai registrului AH. Conținutul biților 5,3 și 1 este nedefinit. Indicatorii nu sunt afectați în urma acestei operații de transfer (în sensul că instrucțiunea LAHF nu este ea însăși generatoare de efecte asupra unor flag-uri – ea doar transferă valorile flag-urilor și atât).
  - Instrucțiunea **SAHF** (Store register AH into Flags) transferă biții 7, 6, 4, 2 și 0 ai registrului AH în indicatorii SF, ZF, AF, PF și respectiv CF, înlocuind valorile anterioare ale acestor indicatori.
  - Instrucțiunea **PUSHF** transferă toți indicatorii în vârful stivei (conținutul registrului Flags se transferă în vârful stivei). Indicatorii nu sunt afectați în urma acestei operații.
  - Instrucțiunea **POPF** extrage cuvântul din vârful stivei și transferă din acesta indicatorii corespunzători în registrul de flag-uri.
- Limbajul de asamblare pune la dispoziția programatorului niște instrucțiuni de setare a valorii indicatorilor de condiție, pentru ca programatorul să poată influența după dorință modul de acțiune a instrucțiunilor care exploatează flaguri.
- **CLC** ( $CF=0$ ) , **CMC** ( $CF = \sim CF$ ) , **STC** ( $CF=1$ ) , **CLD** ( $DF=0$ ) , **STD** ( $DF=1$ )
- **CLI**, **STI** – actioneaza asupra flagului de intrerupere (IF). Funcționeaza efectiv doar in programarea sub 16 biti, aici la programarea sub 32 biti SO interzicand accesul la flag-ul de intreruperi.

# Analiza conceptului de depășire (overflow)

## • **CF (Carry Flag)** – flag pentru depășire FĂRĂ SEMN

- are valoarea 1 în cazul în care în cadrul ultimei operatii efectuate (UOE) s-a efectuat transport în afara domeniului de reprezentare a rezultatului si valoarea 0 in caz contrar

$1001\ 0011b +$	$147 +$	$93h +$	$-109 +$
$0111\ 0011b$	$115$	$73h$	$115$
<hr/> $1\ 0000\ 0110b$	<hr/> $262$	<hr/> $106h$	<hr/> $06$
	(fără semn) CF=1	(hexa)	(cu semn) OF=0

## • **OF (Overflow Flag)** - flag pentru depășire CU SEMN

- dacă rezultatul ultimei instrucțiuni în interpretarea CU SEMN a operanzilor nu a încăput în spațiul rezervat operanzilor (intervalul de reprezentare admisibil), atunci acest flag va avea valoarea 1, altfel va avea valoarea 0.

# Analiza conceptului de depășire (overflow)

- Definiție (generală, comprimată și incompletă).
  - O depășire este o condiție/situație matematică ce exprimă faptul că rezultatul unei operații nu a încăput în spațiul rezervat acestuia.  
*(nici -147 nu “încape” în intervalul [-128..+127] și nici pe un byte însă e mai dificil de intuit că definiția cuprinde și se referă și la acest caz...)*
- Definiție mai exactă și completă:
  - La nivelul procesorului și a limbajului de asamblare o depășire este o condiție/situație matematică ce exprimă faptul că:
    - rezultatul UOE nu a încăput în spațiul rezervat acestuia  
SAU
    - că acest rezultat nu aparține intervalului de reprezentare admisibil pe acea dimensiune de reprezentare  
SAU
    - că operația efectuată este un nonsens matematic în respectiva interpretare (cu semn sau fără semn) și nu poate fi astfel acceptată drept o operație matematică corectă.

# Analiza conceptului de depășire (overflow)

$1001\ 0011b +$ $1011\ 0011b$ <hr/> <b>1</b> 0100 0110b (reprezentare binară)	147 + 179 <hr/> 326 (interpretare fără semn)	93h + B3h <hr/> 146h (reprezentare hexa)	-109 + - 77 <hr/> - 186 (interpretare cu semn)
---	--	--	--

Care este nr MINIM de biți pe care se poate reprezenta 326 și -186?  
 $326 \in [0,511]$  și  $-186 \in [-256,255]$

Așadar numărul minim de biți pe care se pot reprezenta numerele este 9.

Ca urmare, TOATE operațiile de mai sus se desfășoară CORECT MATEMATIC pe 9 biți și operanții și rezultatele finale INCAP în spațiul rezervat, DACA operațiile se desfășoară pe 9 biți !!

# Analiza conceptului de depășire (overflow)

$1001\ 0011b +$ $1011\ 0011b$ <hr/> <b>1 0100 0110b</b>	$147 +$ $179$ <hr/> <b>326</b>	$93h +$ $B3h$ <hr/> <b>146h</b>	$-109 +$ $- 77$ <hr/> <b>- 186</b>
(reprezentare binară)	(interpretare fără semn)	(reprezentare hexa)	(interpretare cu semn)

Insă din păcate, adunarea de mai sus se desfășoară la nivel de procesor pe 8 biți (deoarece în limbaj de asamblare avem că ADD b+b = b) și ca urmare dpxv MATEMATIC, aceasta NU se va desfășura corect pe 8 biți, nici 326 și nici -186 neîncăpând pe 1 octet !!

Acest lucru este semnalat SIMULTAN de către flag-urile CF (pt interpretarea fără semn) și respectiv OF (pt interpretarea CU semn), ambele flag-uri fiind setate la valoarea 1.

**Prin setarea flag-urilor CF și OF la valoarea 1, procesorul ne transmite mesajul că ambele interpretări în baza 10 ale operației binare de adunare pe 1 byte sunt operații matematice incorecte !**

# Analiza conceptului de depășire (overflow)

$0101\ 0011b +$ $0111\ 0011b$ <hr/> $1100\ 0110b$ (reprezentare binară)	$83 +$ $115$ <hr/> $198$ (interpretare fără semn)	$53h +$ $73h$ <hr/> $C6h$ (reprezentare hexa)	$83 +$ $115$ <hr/> $198$ (interpretare cu semn)
---	---	---	---

198 este rezultatul corect în baza 10 pentru ambele interpretări ale OPERANZILOR binari din adunarea de mai sus, INSA trebuie să vedem acum dacă rezultatul începe pe 8 biți (DA – începe, de aceea vom avea CF=0) și respectiv dacă rezultatul operației binare în interpretarea cu semn este unul consistent cu corectitudinea operației matematice efectuate (NU este, deoarece  $1100\ 0110b = -58$  NU este un număr pozitiv în interpretarea CU semn !), deci OF=1

# Analiza conceptului de depășire (overflow)

- OF va fi setat la valoarea 1 (*signed overflow*) dacă pentru operația de adunare ne aflăm în una din următoarele două situații (regulile de depășire la adunare pentru interpretarea cu semn). Sunt singurele două situații care provoacă depășire la **adunare** în interpretarea cu semn:

$$\begin{array}{r} 0 \dots \dots + \\ 0 \dots \dots \\ \hline 1 \dots \dots \end{array} \qquad \begin{array}{r} 1 \dots \dots + \\ 1 \dots \dots \\ \hline 0 \dots \dots \end{array}$$

# Analiza conceptului de depășire (overflow)

- În cazul scăderii, avem de asemenea două reguli de depășire în interpretarea cu semn, consecințe a celor două reguli de la depășirea în cazul adunării:

$$\begin{array}{r} 1 \dots \dots - \\ 0 \dots \dots \\ \hline 0 \dots \dots \end{array} \qquad \begin{array}{r} 0 \dots \dots - \\ 1 \dots \dots \\ \hline 1 \dots \dots \end{array}$$

# Analiza conceptului de depășire (overflow)

$$\begin{array}{r} \textcolor{red}{1} \ 0100\ 0110b \\ - \\ 1100\ 1000b \\ \hline 1001\ 1010b \end{array}$$

(reprezentare binară)

$$\begin{array}{r} 98 \\ - \\ 200 \\ \hline -102 \end{array}$$

(interpretare fără semn  
a operanzilor)

$$\begin{array}{r} 62h \\ - \\ C8h \\ \hline 9Ah \end{array}$$

(reprezentare hexa)

$$\begin{array}{r} 98 \\ - \\ 56 \\ \hline 154 \end{array}$$

(interpretare cu semn  
a operanzilor)

- REZULTATELE MATEMATICE CORECTE sunt precizate mai sus-ignorând INTERPRETAREA în vreun fel a configurației binare 1001 1010b
- Însă, dacă luăm în considerare și interpretările REZULTATULUI obținut în program, avem din păcate :

$$\begin{array}{r} \textcolor{red}{1} \ 0100\ 0110b \\ - \\ 1100\ 1000b \\ \hline 1001\ 1010b \end{array}$$

(reprezentare binară)

$$\begin{array}{r} 98 \\ - \\ 200 \\ \hline 154 \end{array}$$

(interpretare fără semn)  
CF=1

$$\begin{array}{r} 62h \\ - \\ C8h \\ \hline 9Ah \end{array}$$

(reprezentare hexa)

$$\begin{array}{r} 98 \\ - \\ 56 \\ \hline -102 \end{array}$$

(interpretare cu semn)  
OF=1

- Ambele interpretări ale rezultatului obținut în baza 2 SUNT INCORECTE MATEMATIC, deci CF și OF vor fi ambele setate la valoarea 1.

# Analiza conceptului de depășire (overflow)

- **Operatia de înmulțire NU furnizează depășire la nivelul arhitecturii 80x86**, spațiul rezervat pt rezultat fiind suficient pentru ambele interpretări
- Pentru a nu rămâne neutilizate flag-urile CF și OF în cazul înmulțirii s-a luat decizia ca:
  - în cazul în care în cadrul operației de înmulțire dimensiunea rezultatului se întâmplă să fie identică cu cea a operanzilor **( $b^*b = b$ ,  $w^*w = w$  sau  $d^*d = d$ )** flag-urile CF și OF să fie setate ambele la valoarea 0 (**« no multiplication overflow »**, **CF = OF = 0**),
  - iar dacă avem în mod real una dintre situațiile  **$b^*b = w$ ,  $w^*w = d$ ,  $d^*d = qword$** , atunci **CF = OF = 1 (« multiplication overflow »)**.

# Analiza conceptului de depășire (overflow)

- Cel mai grav efect al unei situații de depășire se manifestă în cazul împărțirii: în cazul acestei operații, dacă câtul obținut nu începe în spațiul rezervat

(spațiul rezervat de către asamblor fiind byte pentru împărțire word/byte, word pentru împărțire doubleword/word și respectiv doubleword pentru împărțire quadword/doubleword)

→ se va semnala situație de « **depășire la împărțire** » cu efectul '**Run-time error**' și cu emiterea din partea sistemului de operare a unuia dintre cele 3 mesaje echivalente :

- ‘Divide overflow’, ‘Division by zero’ sau ‘Zero divide’.
- În cazul unei împărțiri care se efectuează corect, adică fără a se semnala depășire, CF și OF sunt nedefinite. Dacă avem însă depășire, programul « crapă », execuția lui se încheie, deci practic nu mai are nici un sens pentru nimeni să se întrebe ce valoare au la acel moment flag-urile CF și OF...

# Analiza conceptului de depășire (overflow)

w / b -> b

$1002 / 3 = 334$  = situație de depășire (overflow) în cazul împărțirii => **Division by zero**

Oare DE CE se emite un astfel de mesaj care sugerează împărțirea la zero cu toate că aici am împărțit la 3 ?

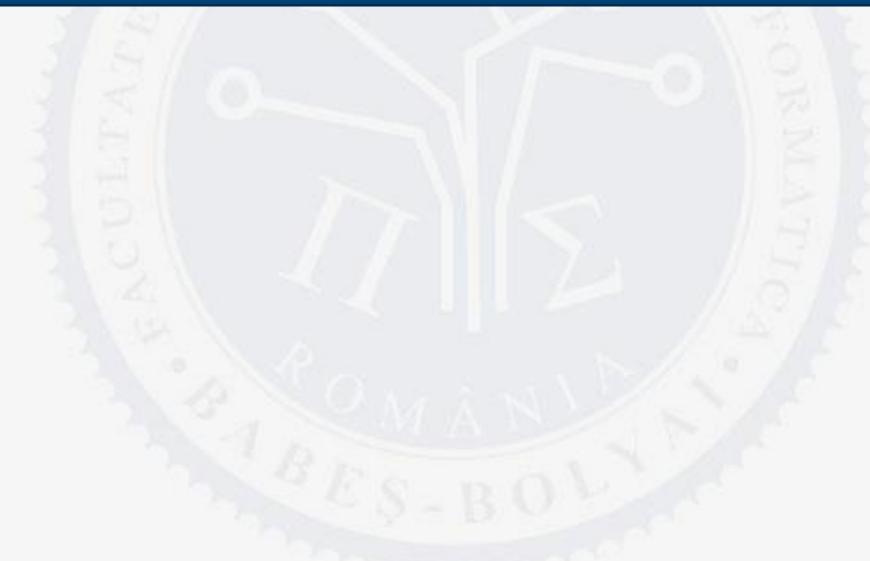
# Analiza conceptului de depășire (overflow)

De ce am nevoie SIMULTAN de CF și OF în EFLAGS ?  
Nu ajunge un singur flag pt a îmi arăta PE RAND daca am sau nu depășire fie în interpretarea cu semn fie în cea fără semn ?



# Analiza conceptului de depășire (overflow)

DE CE AM NEVOIE DE IMUL si IDIV ?





FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Instructiuni

---

# Instrucțiuni

## Instrucțiuni de conversie (distructivă)

Instrucțiune	Efect
CBW	conversie octet conținut în AL la cuvânt în AX (extensie de semn)  <code>mov al, -1 ; AL = 0FFh</code> <code>cbw ; extinde valoarea octet -1 din AL în valoarea cuvânt -1 din AX (0FFFFh).</code>
CWD	conversie cuvânt conținut în AX la dublu cuvânt în DX:AX (extensie de semn)  <code>mov ax, -10000 ; AX = 0D8F0h</code> <code>cwd ; obține valoarea -10000 în DX:AX (DX = 0FFFFh ; AX = 0D8F0h) cwde</code> <code>; obține valoarea -10000 în EAX (EAX = 0FFFFD8F0h)</code>
CWDE	conversie cuvânt din AX în dublucuvânt în EAX (extensie de semn)
CDQ	conversie cuvânt din EAX în dublucuvânt în EDX:EAX (extensie de semn)
MOVZX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), conținutul lui s fară semn (zero extension)
MOVSX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), continutul lui s cu semn (sign extension)

[http://www.c-jump.com/CIS77/ASM/DataTypes/T77\\_0270\\_sext\\_example\\_movsx.htm](http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm)

# Instrucțiuni

## Instrucțiuni de conversie (destructivă)

- Conversia fără semn se realizează prin zerorizarea octetului sau cuvântului superior al valorii de la care s-a plecat. (de exemplu, prin mov ah,0 sau mov dx,0 – efect similar se obține prin aplicarea instr. MOVZX)

### De ce coexistă CWD cu CWDE ?

- CWD trebuie să rămână din rațiuni de backwards compatibility și din rațiuni de funcționalitate a instrucțiunilor (I)MUL și (I)DIV.
- Exemple:**  
MOV ah, 0c8h  
MOVSX ebx, ah ; EBX = FFFFFFFC8h  
MOVZX edx, ah ; EDX = 000000C8h  
MOVSX ax, [v] ; MOVSX ax, byte ptr DS:[offset v]  
MOVZX eax, [v] ; syntax error - op.size not specified

### • Atenție ! NU sunt acceptate sintactic:

CBD	CWDE EBX, BX
CWB	CWD EDX, AX
CDW	MOVZX AX, BX
CDB	MOVSX EAX, -1

MOVSX EAX, [v]
MOVZX EAX, [EBX]
MOVSX dword [EBX], AH
CBW BL

# Instrucțiuni

## Operații aritmetice

- Operanzii sunt reprezentați în cod complementar. Microprocesorul realizează adunările și scăderile "văzând" doar configurații de biți și nu numere cu semn sau fără.
- Regulile de efectuare a adunării și scăderii presupun adunarea de configurații binare, fără a fi nevoie de a interpreta operanzii drept cu semn sau fără semn anterior efectuării operației!
- Deci, la nivelul acestor instrucțiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucțiuni separate pentru adunarea/scăderea cu semn față de adunarea/scăderea fără semn. Adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații! După cum vom vedea acest lucru nu este valabil și pentru înmulțire și împărțire.
- În cazul acestor operații trebuie să stim apriori dacă operanzii vor fi interpretăți drept cu semn sau fără semn. De exemplu, fie doi operanzi A și B reprezentați fiecare pe câte un octet:

A = 9Ch = 10011100b (= 156 în i.f.s și -100 în i.c.s)

B = 4Ah = 01001010b (= 74 atât în i.f.s cât și în i.c.s)

Microprocesorul realizează adunarea  $C = A + B$  și obține

C = E6h = 11100110b (= 230 în i.f.s și -26 în i.c.s)

- Se observă deci că simpla adunare a configurațiilor de biți (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obținut, atât în interpretarea cu semn cât și în cea fără semn.

# Instrucțiuni

## Impactul reprezentării little-endian asupra accesării datelor

- Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octetilor drept octeți și nu drept secvențe de octeți interpretate ca și cuvinte sau dublucuvinte, accesarea de cuvinte ca și cuvinte și nu ca perechi de octeți, accesarea de dublucuvinte ca și dublucuvinte și nu ca secvențe de octeți sau de cuvinte) atunci instrucțiunile limbajului de asamblare vor ține cont în mod AUTOMAT de modalitatea de reprezentare little-endian.
- Ca urmare, dacă se respectă această condiție programatorul nu trebuie să intervenă suplimentar în nici un fel pentru a asigura corectitudinea accesării și manipulării datelor utilizate.
- Exemplu:

```
a db 'd', -25, 120  
b dw -15642, 2ba5h  
c dd 12345678h
```

...  
`mov al, [a]` ; se încarcă în AL codul ASCII al caracterului 'd'  
`mov bx, [b]` ; se încarcă în BX valoarea -15642  
; ordinea octetilor în BX va fi însă inversată față de  
; reprezentarea în memorie, deoarece numai  
; reprezentarea în memorie folosește reprezentarea  
; little-endian! În registri datele sunt memorate  
; conform reprezentării structurale normale,  
; echivalente unei reprezentări big endian.  
`mov edx, [c]` ; se încarcă în EDX valoarea dublucuvânt 12345678h

# Instrucțiuni

## Impactul reprezentării little-endian asupra accesării datelor

- Dacă însă se dorește accesarea sau interpretarea datelor sub o formă diferită față de modalitatea de definire atunci trebuie utilizate **conversii explicite de tip**.
- În momentul utilizării conversiilor explicite de tip programatorul trebuie să își asume însă întreaga responsabilitate a interpretării și accesării corecte a datelor.
- În astfel de situații programatorul este obligat să conștientizeze particularitățile de reprezentare **little-endian** (*ordinea de plasare a octetilor în memorie*) și să utilizeze modalități de accesare a datelor în conformitate cu aceasta

# Instrucțiuni

## Impactul reprezentării little-endian asupra accesării datelor

```
segment data
    a dw 1234h      ;datorită reprezentării little-endian, în memorie octetii sunt plasati astfel:
    b dd 11223344h ;          34h 12h 44h 33h 22h 11h
                      ; adresa a   a+1 b   b+1 b+2 b+3
    c db -1

segment code
    mov al, byte [a+1] ;accesarea lui a drept octet, efectuarea calculului de adresă a+1,
    mov dx, word [b+2]  ;selectarea octetului de la adresa a+1 (octetul de valoare 12h) și
    mov dx, word [a+4]  ;transferul său în registrul AL.
    mov dx, [a+4]       ;dx:=1122h
    mov bx, [b]          ;dx:=1122h deoarece b+2 = a+4 , în sensul că aceste expresii de
    mov bx, [a+2]        ;tip pointer desemnează aceeași adresă și anume adresa oct. 22h.
    mov ecx, dword [a]   ;această instrucțiune este echivalentă cu cea de mai sus, nefiind
                        ;realmente necesară
    mov ebx, [b]          ;utilizarea operatorului de conversie WORD
    mov bx, [a+2]          ;bx:=3344h
    mov eax, word [a+1]   ;bx:=3344h, deoarece ca adrese b = a+2.
    mov dx, [c-2]          ;ecx:=33441234h, deoarece dublucuvântul ce începe la adresa a
    mov bh, [b]            ;este format din octetii 34h 12h 44h 33h care (datorită reprezentării
    mov ch, [b-1]          ;little-endian) înseamnă de fapt ;dublucuvântul 33441234h.
    mov cx, [b+3]          ;ebx := 11223344h
    mov ax, word [a+1]   ;ax := 4412h
    mov eax, word [a+1]   ;eroare de sintaxă. Dacă era dword atunci eax := 22334412h
    mov dx, [c-2]          ;DX := 1122h deoarece c-2 = b+2 = a+4
    mov bh, [b]            ;bh := 44h
    mov ch, [b-1]          ;ch := 12h
    mov cx, [b+3]          ;CX := 0FF11h
```

# Instructiuni

## Constante de tip string. Reprezentare in memorie si utilizare in cadrul unor instructiuni de transfer.

- În cazul initializării unei zone de memorie cu valori de tip constantă string (sizeof > 1) tipul de date utilizat în definire (dw, dd, dq) are rol doar de rezervare a spațiului dorit, ordinea de “umplere” a zonei de memorie respective fiind ordinea în care apar caracterele (octetii) în cadrul constantei de tip string:

a6 dd '123', '345', 'abcd'	; se vor defini 3 dublucuvinte continutul lor fiind   31h 32h 33h 00h   33h 34h 35h 00h   61h 62h 63h 64h
a6 dd '1234'	;   31h 32h 33h 34h
a6 dd '12345' , 'abc'	;   31h 32h 33h 34h   35h 00h 00h 00h   61h 62h 63h 00h
a7 dw '23','45'	;   32h 33h   34h 35h   - 2 cuvinte = 1 doubleword
a7 dw '2345'	;   32h 33h   34h 35h  - 2 cuvinte
a7 dw '23456'	;   32h 33h   34h 35h   36h 00h   - 3 cuvinte
a8 dw '1', '2', '3'	;   31h 00h   32h 00h   33h 00h   - 3 cuvinte
a9 dw '123'	;   31h 32h   33h 00h   - 2 cuvinte

# Instructiuni

Constante de tip string. Reprezentare in memorie si utilizare in cadrul unor instructiuni de transfer.

- Urmatoarele definitii produc aceeasi configuratie de memorie

dd 'ninechars'	; constanta string doubleword
dd 'nine', 'char', 's'	; 3 dublucuvinte
db 'ninechars', 0, 0, 0	; "umplere" zona prin secheta de octeti

- Definitia din documentatia oficiala spune:

*A character constant with more than one byte will be arranged with little-endian order in mind: if you code mov eax, 'abcd' (EAX = 0x64636261) then the constant generated is not 0x61626364, but 0x64636261, so that if you were then to store the value into memory, it would read abcd rather than dcba. This is also the sense of character constants understood by the Pentium's CPUID instruction.*

- Esenta acestei definitii este ca VALOAREA asociata unei constante de tip string 'abcd' este de fapt 'dcba' (adica acesta este modul de STOCARE al acestei constante în TABELA DE CONstanTE)

Instructiune	OllyDbg	Memorie
Mov dword [a], '2345'	mov dword ptr DS:[401000], 35343332	32h 33h 34h 35h

# Instrucțiuni

Constante de tip string. Reprezentare in memorie si utilizare in cadrul unor instructiuni de transfer.

- Dar daca folosim a data definition like `a7 dw '2345'` the corresponding memory layout will be NO little-endian representation, but `| 32h 33h 34h 35h |`
- Astfel, comparativ si in rezumat (constante tip string vs. constante numerice) :

<code>a7 dw '2345'</code>	<code>;   32h 33h   34h 35h  </code>
<code>a8 dd 12345678h</code>	<code>;   78h 56h 34h 12h  </code>
<code>mov eax, '2345'</code>	<code>; EAX = '5432' = 35h 34h 33h 32h</code>
<code>mov ebx, [a7]</code>	<code>; EBX = '5432' = 35h 34h 33h 32h</code>
<code>mov ecx, 12345678h</code>	<code>; ECX = 12345678h (sizeof = 4 bytes)</code>
<code>mov dword [var1], '12345678'</code>	<code>; var1 = '1234' (31 32 33 34) ; (sizeof '12345678' = 8 bytes)</code>
<code>mov edx, [a8]</code>	<code>EDX = 12345678h</code>

# Instructiuni

## Constante de tip string. Reprezentare in memorie si utilizare in cadrul unor instructiuni de transfer.

- In cazul in care se foloseste DB ca directiva de definire a datelor e normal ca ordinea octetilor data in specificarea constantei sa se regaseasca si in memorie in mod similar, deci acest caz nu comporta analiza si discutii suplimentare.

a66 TIMES 4 db '13'	;   31h 33h 31h 33h 31h 33h 31h 33h   echiv cu ...db '1','3'
a67 TIMES 4 dw '13'	;   31h 33h 31h 33h 31h 33h 31h 33h   - cele doua moduri de definire diferite produc acelasi rezultat !!!
a68 TIMES 4 dw '1','3'	;   31h 00h   33h 00h   31h 00h   33h 00h   31h 00h   33h 00h   31h 00 h   33h 00h
a69 TIMES 4 dd '13'	;   31h 33h 00h 00h   31h 33h 00h 00h   31h 33h 00h 00h   31h 33h 00h 00h

- Deci, un sir constant in NASM se comporta ca si cum ar exista o „zonă de memorie” alocată anterior acestor constante (ea exista !! si se numeste TABELA DE CONSTANTE!!), unde acestea sunt stocate folosind reprezentarea little-endian !!
- Dpdv al REPREZENTARII valoarea asociată unei constant de tip string este INVERSA SA !!!! (vezi și “definiția oficială” de mai sus!)
- Practic, dacă inițializăm o zonă de memorie cu o constantă de tip string (fie prin directive de definire a datelor, fie prin mov [zona de memorie], constanta\_string) ordinea în care caracterele vor fi depuse în memorie este ordinea în care ele apar în scrierea pe hârtie a constantei de tip string !!!
- Dacă inițializăm conținutul unui registru cu o constantă de tip string, caracterele se vor depune în ordine inversă apariției lor în scrierea pe hârtie a constantei de tip string !



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Operații pe biți

---

# Operații logice pe biți

- Limbajul de asamblare dispune de un set de patru instrucțiuni pentru realizarea de operații logice la nivel de bit: **AND**, **OR**, **XOR** și **NOT**.

AND d, s	<d> <-> <d> și <s>	CF, OF, PF, SF, ZF modificări; AF - nedefinit
OR d, s	<d> <-> <d> sau <s>	CF, OF, PF, SF, ZF modificări; AF - nedefinit
XOR d, s	<d> <-> <d> sau exclusiv <s>	CF, OF, PF, SF, ZF modificări; AF - nedefinit
NOT s	<s> <-> <s> negat complement față de 1	

# Operații logice pe biți

- Pentru doi biți, unul din sursă, altul din destinație, instrucțiunile logice produc noul bit destinație având valoarea conform următoarelor reguli:

bitul d	bitul s	d AND s	d OR s	d XOR s
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

# Operații logice pe biți: AND

## AND:

```
AND reg,reg  
AND reg,mem  
AND reg,imm  
AND mem,reg  
AND mem,imm
```

## Exemplu:

```
mov al, 10101110b  
and al, 11110110b  
; AL = 10100110b
```

$$\begin{aligned}x \text{ and } 0 &= 0 \\x \text{ and } 1 &= x\end{aligned}$$

bitul d	bitul s	d AND s
0	0	0
0	1	0
1	0	0
1	1	1

OF = CF = 0  
SF, ZF, PF – modificate în funcție de rezultat

# Operații logice pe biți: OR

## OR:

OR reg, reg  
OR reg, mem  
OR reg, imm  
OR mem, reg  
OR mem, imm

$$x \text{ or } 0 = x$$

$$x \text{ or } 1 = 1$$

bitul d	bitul s	d OR s
0	0	0
0	1	1
1	0	1
1	1	1

## Exemplu:

```
mov al,11100011b  
or al,00000100b  
; AL = 11100111b
```

OF = CF = 0

SF, ZF, PF – modificate în funcție de rezultat

# Operații logice pe biți: XOR

## XOR:

XOR reg, reg  
XOR reg, mem  
XOR reg, imm  
XOR mem, reg  
XOR mem, imm

## Exemplu:

```
mov al,11100011b  
xor al,00100100b  
; AL = 11000111b
```

bitul d	bitul s	d XOR s
0	0	0
0	1	1
1	0	1
1	1	0

OF = CF = 0

SF, ZF, PF – modificate în funcție de rezultat

# Operații logice pe biți: NOT

## NOT:

NOT reg  
NOT mem

bitul d	NOT d
0	1
0	0

## Exemplu:

```
mov al,11110000b  
not al  
; AL = 00001111b
```

Flag-urile nu sunt afectate de către instrucțiunea NOT

# Operații logice pe biți

- Pentru doi operanzi instrucțiunile logice realizează aceste operații asupra perechilor corespunzătoare de biți. Operanzii sursă și destinație trebuie să aibă ambii aceeași dimensiune. Fie:

A      DB      10010101b  
B      DB      01011010b

- Atunci avem

```
mov al, [A]  
and al, [B] ; rezultă 00010000b în AL
```

```
mov al, [A]  
or al, [B] ; rezultă 11011111b în AL
```

```
mov al, [A]  
xor al, [B] ; rezultă 11001111b în AL
```

# Operații logice pe biți

- Instrucțiunea AND este indicată pentru **izolarea** unui anumit bit sau pentru forțarea anumitor biți la valoarea 0.
- Astfel, dacă dintr-o configurație pe 8 biți

$a = \underline{x} \ b$

dorim izolarea bitului  $i$  ( $0 \leq i \leq 7$ ), vom crea expresia

AND  $a, 2^i$

Cum izolăm biții 0,2,3 din octetul a?

# Operații logice pe biți

- Instrucțiunea OR poate fi folosită printre altele pentru forțarea unui biț la valoarea 1

Cum forțăm biții 0,2,3 din octetul a să aibă valoarea 1?

# Operații logice pe biți

- Instrucțiunea XOR este indicată pentru schimbarea valorii unor biți din 0 în 1 sau din 1 în 0.

Cum forțăm biții 4-7 din AL să își schimbe valorile în timp ce restul bițiilor rămân neschimbați?

Ce face XOR AX, AX?

# Operații logice pe biți

- Instrucțiunea **NOT** modifică biții operandului în valoarea lor complementară (0 în 1 și 1 în 0)

Ce instrucțiune echivalentă cu **XOR** putem scrie astfel  
încât efectu asupra operandului să fie același cu **NOT**?

# Operații logice pe biți: TEST

**TEST:** AND fictiv (nu modifică operandul, dar modifică flag-urile corespunzător)

```
TEST reg, reg
TEST reg, mem
TEST reg, imm
TEST mem, reg
TEST mem, imm
```

**Exemplu:**

```
mov al, 00100101b
TEST al, 0001001b
; AL - neschimbă 00100101b AND 0001001b = 00000001b
; flag-uri schimbate corespunzător: ZF=0
```

CF = OF = 0

SF, ZF, PF – modificate în funcție de rezultat

# Operații logice pe biți: TEST

**TEST:** AND fictiv (nu modifică operandul, dar modifică flag-urile corespunzător)

```
TEST reg, reg
TEST reg, mem
TEST reg, imm
TEST mem, reg
TEST mem, imm
```

**Exemplu:**

```
mov al, 00100100b
TEST al, 0001001b
; AL - neschimbă 00100100b AND 0001001b = 0000000b
; flag-uri schimbate corespunzător: ZF=1
```

CF = OF = 0

SF, ZF, PF – modificate în funcție de rezultat

# Operații logice pe biți

- Cum putem seta / reseta ZF ?

- $ZF = 1$ , în urma operațiilor TEST <op>, 0 sau AND <op>, 0
- $ZF = 0$ , în urma operației OR <op>, 1

- Cum putem seta / reseta SF ?

- $SF = 1$ , în urma operațiilor OR <op>, 80h
- $SF = 0$ , în urma operației AND <op>, 7Fh

# Deplasări și rotiri de biți

- În cadrul octetilor sau cuvintelor biții pot fi deplasăți aritmetic sau logic sau rotiți la dreapta sau la stânga cu una sau mai multe poziții. Numărul de deplasări este fie 1, fie o valoare cuprinsă între 1 și 255
- Instrucțiunile de *deplasare* de biți se clasifică în:
  - Instrucțiuni de deplasare logică
    - stânga - **SHL**
    - dreapta - **SHR**
  - Instrucțiuni de deplasare aritmetică
    - stânga - **SAL**
    - dreapta - **SAR**
- Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:
  - Instrucțiuni de rotire fără carry
    - stânga - **ROL**
    - dreapta - **ROR**
  - Instrucțiuni de rotire cu carry
    - stânga - **RCL**
    - dreapta - **RCR**

# Deplasări și rotiri de biți

- Pentru a defini deplasările și rotirile considerăm înainte de fiecare linie prezentată mai jos (ca și configurație inițială un octet  $x = abcdefgh$ , unde a-h sunt cifre binare, h este cifra binară de rang 0, a este cifra binară de rang 7, iar k este valoarea existentă în CF ( $CF=k$ )). Atunci:

SHL X,1 ; rezultă  $X = bcdefgh0$  și  $CF = a$   
SHR X,1 ; rezultă  $X = 0abcdefg$  și  $CF = h$   
SAL X,1 ; identic cu SHL  
SAR X,1 ; rezultă  $X = aabcdefg$  și  $CF = h$   
ROL X,1 ; rezultă  $X = bcdefgha$  și  $CF = a$   
ROR X,1 ; rezultă  $X = habcdefg$  și  $CF = h$   
RCL X,1 ; rezultă  $X = bcdefghk$  și  $CF = a$   
RCR X,1 ; rezultă  $X = kabcdefg$  și  $CF = h$

**Important!** Se observă că, în toate cazurile, bitul ce părăsește configurația trece în CF.

Deplasările logice pot fi folosite pentru izolarea anumitor biți în interiorul octetilor (cuvintelor), iar deplasările aritmetice se pot utiliza pentru înmulțirea sau împărțirea numerelor binare cu puteri ale lui 2.

# Deplasări și rotiri de biți

SHL <op>, 1	deplasare logică (aritmetică) stânga cu o poziție. CF conține bitul cel mai semnificativ deplasat. La dreapta se introduc zerouri.	- Flag-ul AF este întotdeauna nedefinit în urma unei operații de deplasare.
SAL <op>, 1		- Indicatorii PF, SF și ZF sunt actualizați ca și în cazul instrucțiunilor logice pe biți.
SHL <op>, cl	deplasare logică (aritmetică) stânga cu cl poziții. CF conține ultimul bit deplasat. La dreapta se introduc zerouri.	- Indicatorul CF conține întotdeauna valoarea ultimului bit deplasat din cadrul operandului.
SAL <op>, cl		- Conținutul indicatorului OF este întotdeauna nedefinit în cazul unor operații de deplasare a mai multor biți. În cazul în care se deplasează un singur bit, OF este setat la valoarea 1 dacă valoarea bitului cel mai semnificativ (semnul) a fost schimbată de operația de deplasare.
SHR <op>, 1	deplasare logică dreapta cu o poziție. La stânga se introduc zerouri. Bitul cel mai puțin semnificativ este deplasat în CF.	
SHR <op>, cl	deplasare logică dreapta cu cl poziții. La stânga se introduc zerouri. CF va conține ultimul bit deplasat.	
SAR <op>, 1	deplasare aritmetică dreapta cu o poziție. La stânga se extinde bitul de semn. Bitul cel mai puțin semnificativ este deplasat în CF.	
SAR <op>, cl	deplasare aritmetică dreapta cu cl poziții. La stânga se extinde bitul de semn. CF va conține ultimul bit deplasat.	
ROL <op>, 1	rotire stânga cu o poziție. CF va conține bitul (cel mai semnificativ) rotit.	
ROL <op>, cl	rotire stânga cu cl poziții. CF va conține ultimul bit rotit.	Instrucțiunile de rotire afectează numai flagurile CF și OF. CF va conține întotdeauna valoarea ultimului bit rotit.
ROR <op>, 1	rotire dreapta cu o poziție. CF va conține bitul cel mai puțin semnificativ al lui s.	Pentru operațiile de rotire cu mai mulți biți valoarea flagului OF este nedefinită. La rotirea unui singur bit OF este setat la valoarea 1 dacă bitul cel mai semnificativ își schimbă valoarea (schimbare de semn).
ROR <op>, cl	rotire dreapta cu cl poziții. CF va conține ultimul bit rotit.	
RCL <op>, 1	rotire stânga cu carry cu o poziție.	
RCL <op>, cl	rotire stânga cu carry cu cl poziții.	
RCR <op>, 1	rotire dreapta cu carry cu o poziție.	
RCR <op>, cl	rotire dreapta cu carry cu cl poziții.	

# Deplasări și rotiri de biți: SHL

## SHL:

```
SHL reg, imm8
SHL mem, imm8
SHL reg, CL
SHL mem, CL
```

## Exemplu:

```
mov al, 10010110b
SHL al, 1 ; AL = 00101100b, CF = 1
SHL al, 2 ; AL = 10110000b, CF = 0
```

## Multiplicarea rapidă a operandului cu puteri ale lui 2

Deplasarea spre stânga obținută ca efect al instrucțiunii SHL este de fapt o înmulțire cu 2 (presupunând desigur că bitul cel mai semnificativ a fost 0, în caz contrar fiind vorba de trunchiere - pierderea celei mai semnificative cifre binare).

```
shl    dx, 1          ;DX*2
shl    dx, 2          ;DX*4
shl    dx, 3          ;DX*8
shl    dx, 4          ;DX*16
```

# Deplasări și rotiri de biți: SHL

## Exemplu:

```
xor    ah,ah
mov    al,[var]
shl    ax,3          ;înmulțire cu opt
add    al,[var]
adc    ah,0          ;al = var * 9
add    al,[var]
adc    ah,0          ;al = var * 10
```

# Deplasări și rotiri de biți: SHR

## SHR:

```
SHR reg, imm8
SHR mem, imm8
SHR reg, CL
SHR mem, CL
```

## Exemplu:

```
mov al, 10010110b
SHR al, 1 ; AL = 01001011b, CF = 0
SHR al, 2 ; AL = 00010010b, CF = 1
```

## Exemplu:

SHR poate fi folosit pentru a transfera conținutul HIGH al lui EAX în partea LOW a lui EAX

```
mov eax, 0ABCD1234h
mov cl, 16
shr EAX, cl ; EAX = 0000ABCDh, => AX=ABCDh
```

# Deplasări și rotiri de biți: SHR

## Împărțirea rapidă a operandului cu puteri ale lui 2

```
mov dl, 32  
shr dl, 1 ; AL=16
```

```
mov al, 01000000b ; AL = 64  
shr al, 3 ; împărțire la 8, AL = 00001000b
```

# Deplasări și rotiri de biți: SAL

**SAL:** identic cu SHL

```
SAL reg, imm8
SAL mem, imm8
SAL reg, CL
SAL mem, CL
```

**Exemplu:**

```
mov al, 10010110b
SAL al, 1 ; AL = 00101100b, CF = 1
SAL al, 2 ; AL = 10110000b, CF = 0
```

# Deplasări și rotiri de biți: SAR

## SAR:

```
SAR reg, imm8
SAR mem, imm8
SAR reg, CL
SAR mem, CL
```

## Exemplu:

```
mov al, 0F0h
; AL = 11110000b (-16)
sar al, 1
; AL = 11111000b (-8), CF = 0
```

## Exemplu:

SAR are ca efect păstrarea semnului operandului. Acest lucru face ca instrucțiunea SAR să fie indicată pentru efectuarea împărțirilor cu semn la puteri ale lui 2.

```
mov al, 10010110b
SAR al, 1 ; AL = 11001011b, CF = 0
```

```
mov bx, -4
sar bx, 1 ; BX = -2
```

# Deplasări și rotiri de biți: SAR

## Exemplu:

Dacă AX conține un întreg al cărui bit de semn. Dorim să îl extindem în EAX trebuie:

```
mov ax,-128 ; EAX = ????FF80h
shl eax,16 ; EAX = FF800000h
sar eax,16 ; EAX = FFFFFF80h
```

# Deplasări și rotiri de biți: ROL

## ROL:

```
ROL reg, imm8
ROL mem, imm8
ROL reg, CL
ROL mem, CL
```

## Exemplu:

```
mov al, 40h ; AL = 01000000b
rol al, 1 ; AL = 10000000b, CF = 0
rol al, 1 ; AL = 00000001b, CF = 1
rol al, 1 ; AL = 00000010b, CF = 0
```

## Realinierarea biților în cadrul unui operand:

```
mov al, 26h
rol al, 4 ; AL = 62h

mov al, 00100000b
mov cl, 3
rol al, cl ; CF = 1, AL = 00000001b
```

## Rotirea cifrelor hexa:

```
mov ax, 6A4Bh
rol ax, 4 ; AX = A4B6h
rol ax, 4 ; AX = 4B6Ah
rol ax, 4 ; AX = B6A4h
rol ax, 4 ; AX = 6A4Bh
```

# Deplasări și rotiri de biți: ROR

## ROR:

```
ROR reg, imm8  
ROR mem, imm8  
ROR reg, CL  
ROR mem, CL
```

## Exemplu:

```
mov al, 01h ; AL = 00000001b  
ror al, 1 ; AL = 10000000b, CF = 1  
ror al, 1 ; AL = 01000000b, CF = 0
```

## Realinierarea biților în cadrul unui operand:

```
mov si, 49f1h  
mov cl, 4  
ror si, cl ; si = 149Fh
```

# Deplasări și rotiri de biți: RCL

## RCL:

```
RCL reg, imm8
RCL mem, imm8
RCL reg, CL
RCL mem, CL
```

## Exemplu:

```
clc ; CF = 0
mov bl, 88h;CF = 0, BL = 10001000b
rcl bl,1;CF = 1, BL = 00010000b
rcl bl,1;CF = 0, BL = 00100001b
```

## Realizarea de deplasări de biți implicând operanzi reprezentați pe mai multe cuvinte

De exemplu, secvența următoare multiplică cu 4 valoarea din DX:AX

```
shl    ax,1      ;bitul 15 din AX este depus în CF
rcl    dx,1      ;valoarea din CF se depune în bitul 0 din DX
shl    ax,1      ;bitul 15 din AX se depune în CF
rcl    dx,1      ;valoarea din CF se depune în bitul 0 din DX
```

# Deplasări și rotiri de biți: RCR

## RCR:

RCR reg, imm8  
RCR mem, imm8  
RCR reg, CL  
RCR mem, CL

## Exemplu:

```
stc ; CF = 1
mov ah, 10h; AH = 0001 0000,CF = 1
rcr ah, 1 ; AH = 1000 1000, CF = 0
```



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)

---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Ramificări, salturi, cicluri

---

# Instrucțiuni de salt necondiționat

- În această categorie intră instrucțiunile

JMP (echivalentul instrucțiunii GOTO din alte limbaje),

CALL (apelul de procedură înseamnă transferul controlului din punctul apelului la prima instrucțiune din procedura apelată)

RET (transfer control la prima instrucțiune executabilă de după CALL).

Instrucțiune	Efect
JMP operand	Salt necondiționat la adresa determinată de operand
CALL operand	Transferă controlul procedurii determinată de operand
RET [n]	Transferă controlul instrucțiunii de după CALL

# Instrucțiuni de salt necondiționat

## Saltul necondiționat: Instrucțiunea JMP

- Instrucțiunea de salt necondiționat JMP are sintaxa  
JMP operand  
unde operand este o etichetă, un registru sau o variabilă de memorie ce conține o adresă.
- Efectul ei este transferul necondiționat al controlului la instrucțiunea ce urmează etichetei, la adresa dată de valoarea registratorului sau constantei, respectiv la adresa conținută în variabila de memorie.
- De exemplu, după execuția secvenței

```
mov eax,1
jmp AdunaDoi
AdunaUnu: inc eax
jmp urmare
AdunaDoi: add eax,2
urmare: . . .
```

registruul EAX va conține valoarea 3.

- Instrucțiunile inc și jmp dintre etichetele AdunaUnu și AdunaDoi nu se vor executa, decât dacă se va face salt la AdunaUnu de altundeva din program.

# Instrucțiuni de salt necondiționat

## Saltul necondiționat: Instrucțiunea JMP

- După cum am menționat, saltul poate fi făcut și la o adresă memorată într-un registru sau într-o variabilă de memorie. Exemple:

(1) mov eax, etich  
jmp eax ; operand registru . . .  
; jmp [eax] ?

• (2) segment data  
Salt DD Dest ; salt := offset Dest  
segment cod

...  
jmp [Salt] ; salt NEAR  
... ; operand variabilă de memorie  
Dest : . . .

- Dacă în cazul (1) dorim înlocuirea operandului destinație registru cu un operand destinație variabilă de memorie, o soluție posibilă este:

(1') b resd 1  
...  
mov [b], DWORD etich ; b := offset etich  
jmp [b] ; salt NEAR – op. var. de mem. JMP DWORD PTR DS:[offset\_b]

# Instrucțiuni de salt condiționat

- Instrucțiunile de salt condiționat se folosesc de obicei în combinație cu instrucțiuni de comparare.
- De aceea, semnificațiile instrucțiunilor de salt rezultă din semnificația operanzilor unei instrucțiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucțiune, este de multe ori necesară determinarea relației de ordine dintre două valori.
- De exemplu, se pune întrebarea: numărul  $11111111b$  ( $= FFh = 255 = -1$ ) este mai mare decât  $00000000b$  ( $= 0h = 0$ ) ?

**Răspunsul poate fi și da și nu!**

Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 și este evident mai mare decât 0.

Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 și este mai mic decât 0.

- Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine diverselor instrucțiuni de salt condiționat

# Instrucțiunea CMP

CMP dest, sursa ; SUB fictiv care nu modifică dest, dar afectează flag-urile OF, SF, ZF, CF, AF, PF

Rezultat CMP (comparare fără semn)	ZF	CF
dest < sursa	0	1
dest > sursa	0	0
dest = sursa	1	0

Rezultat CMP (comparare cu semn)	Flag-uri
dest < sursa	SF != OF
dest > sursa	SF = OF
dest = sursa	ZF = 1

# Instrucțiuni de salt condiționat

- Precizăm că pentru toate instrucțiunile de salt sintaxa este aceeași, și anume `<instrucțiune_de_salt>` etichetă
- Semnificația instrucțiunilor de salt condiționat este dată sub forma "salt dacă operand1 <> față de operand2" (unde cei doi operanzi sunt obiectul unei instrucțiuni anterioare CMP sau SUB) sau referitor la valoarea concretă setată pentru un anumit flag.
- După cum se observă și din condițiile ce trebuie verificate, instrucțiunile ce se află într-o aceeași linie a tabelului sunt echivalente.
- Când se compară două numere:
  - cu semn se folosesc termenii "**less than**" (mai mic decât) și "**greater than**" (mai mare decât)
  - fără semn se folosesc termenii "**below**" (inferior, sub) și respectiv "**above**" (superior, deasupra, peste).

# Instrucțiuni de salt condiționat

## Salturi condiționate de flag-uri

JB JNAE JC	este inferior nu este superior sau egal există transport	CF=1
JAE JNB JNC	este superior sau egal nu este inferior nu există transport	CF=0
JBE JNA	este inferior sau egal nu este superior	CF=1 sau ZF=1
JA JNBE	este superior nu este inferior sau egal	CF=0 și ZF=0
JE JZ	este egal este zero	ZF=1
JNE JNZ	nu este egal nu este zero	ZF=0
JL JNGE	este mai mic decât nu este mai mare sau egal	SF!=OF

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

# Instrucțiuni de salt condiționat

Relația între operanzi ce se dorește a fi testată	Comparație cu semn	Comparație fără semn
$d = s$	JE	JE
$d \neq s$	JNE	JNE
$d > s$	JG	JA
$d < s$	JL	JB
$d \geq s$	JGE	JAE
$d \leq s$	JLE	JBE

# Instrucțiuni de salt condiționat

```
mov ax,5
cmp ax,10 ; ZF = 0, CF = 1
JC eticheta
```

```
mov ax,1000
mov cx,1000
cmp cx,ax ; ZF = 1, CF = 0
JZ eticheta
```

```
mov si,105
cmp si,0 ; ZF = 0, CF = 0
JNZ eticheta
```

```
mov eax,5
cmp eax,5
JZ L1 ; jump if ZF=1
...
L1:
```

```
mov eax,5
cmp eax,5
JE L1 ; jump if EAX=5
...
L1:
```

# Instrucțiuni de salt condiționat

```
mov eax,1000h
cdq
mov ebx,10h
div ebx ; EAX = 00000100h

mov eax, [deîmpărțit]
cdq
mov ebx, [împărțitor]
cmp ebx,0
je NoDivideZero
div ebx ; împărțitorul nu e 0, se poate continua
.....
NoDivideZero:
; afișare eroare
```

# Instrucțiuni de ciclare

- Ele sunt:  
`LOOP`, `LOOPE`, `LOOPNE` și `JECXZ`.
- Sintaxa lor este  
`<instructiune> etichetă`
- Instrucțiunea `LOOP` comandă reluarea execuției blocului de instrucțiuni ce începe la etichetă, atât timp cât valoarea din registrul ECX este diferită de 0.
- **Se efectuează întâi decrementarea registrului ECX și apoi se face testul și eventual saltul.**
- Saltul este "scurt" (max. 127 octeți - atenție deci la "distanța" dintre `LOOP` și etichetă!).  
(*short jump is out of range!*)
- În cazul în care condițiile de terminare a ciclului sunt mai complexe se pot folosi instrucțiunile `LOOPE` și `LOOPNE`. Instrucțiunea `LOOPE` (LOOP while Equal) diferă față de `LOOP` prin condiția de terminare, ciclul terminându-se fie dacă `ECX=0`, fie dacă `ZF=0`.
- În cazul instrucțiunii `LOOPNE` (LOOP while Not Equal) ciclul se va termina fie dacă `ECX=0`, fie dacă `ZF=1`.
- Chiar dacă ieșirea din ciclu se face pe baza valorii din `ZF`, decrementarea lui `ECX` are oricum loc. `LOOPE` mai este cunoscută și sub numele de `LOOPZ` iar `LOOPNE` mai este cunoscută și sub numele de `LOOPNZ`. Se folosesc de obicei precedate de o instrucțiune `CMP` sau `SUB`.

# Instrucțiuni de ciclare

- `JECXZ` (Jump if ECX is Zero) realizează saltul la eticheta operand numai dacă `ECX=0`, fiind utilă în situația în care se dorește testarea valorii din `ECX` înaintea intrării într-o buclă.
- În exemplul următor instrucțiunea `JECXZ` se folosește pentru a se evita intrarea în ciclu dacă `ECX=0`:

```
jecxz MaiDepartे ;dacă ECX=0 se sare peste buclă
Bucla:
```

```
Mov BYTE [esi],0 ;inițializarea octetului curent
inc esi           ;trecere la octetul următor
loop Bucla       ;reluare ciclu sau terminare
```

```
MaiDepartе:...
```

- La întâlnirea instrucțiunii `LOOP` cu `ECX=0`, `ECX` este decrementat, obținându-se valoarea `0FFFFh` (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în `ECX`, adică de încă 4.294.967.296 ori!
- Este important să precizăm aici faptul că **nici una dintre instrucțiunile de ciclare prezentate nu afectează flag-urile**.

```
        dec ecx
loop Bucla    și    jnz Bucla
```

deși semantic echivalente, nu au exact același efect, deoarece spre deosebire de `LOOP`, instrucțiunea `DEC` afectează indicatorii `OF`, `ZF`, `SF` și `PF`.

# Instrucțiuni de ciclare

## SHORT JMP vs. LONG JMP

Dinnou:

```
Mov eax, 89
...
Jmp Maideparte ; JMP nu are restricții în ceea ce privește distanța
Resd 1000h ; distanța dintre LOOP și eticheta Dinnou este > 127 octeți, deci
             ; nu este short jump
```

Maideparte:

```
Mov ebx, 17
```

...

```
Loop Dinnou ; syntax error: short jump is out of range + warning: byte data
               ; exceeds bounds
```

- Daca inlocuim Loop Dinnou cu echivalentul

```
dec ecx
jnz Dinnou
```

NU mai obtinem eroare deoarece

- In TASM and MASM the short jump condition (ie maximum 127 bytes distance) is imposed both at the level of LOOP type instructions and at the level of conditional jump instructions.
- In NASM the restriction is valid only for LOOP type instructions, the conditional jump instructions are no longer subject to this restriction.

# Instrucțiuni de ciclare

Totusi, exista o diferență importantă între cele 2 variante: Loop NU afectează flag-urile, în timp ce DEC DA !! Dacă vrem o variantă echivalentă care să NU afecteze flag-urile:

Dinnou:

```
Mov eax, 89
...
Jmp Maideparte ; JMP nu are restricții în ceea ce privește distanța,
; Resd 1000h ; distanța dintre LOOP și eticheta Dinnou este > 127 octeți, deci
; nu este short jump
```

MaiAproape:

```
Jmp Dinnou
```

Maideparte:

```
Mov ebx, 17
```

```
...
```

```
Loop MaiAproape ; short jump
```

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplu	Explicație
mov al, 80h	;al := 128 = 10000000b = -128 Remarcăm faptul că datorită regulilor de reprezentare în cod complementar 128 și -128 au aceeași reprezentare binară și anume 10000000b
cmp al, 0	;instrucțiunea CMP nu interpretează în nici un fel valoarea din AL (ca fiind cu semn sau fără semn) ci doar realizează scăderea fictivă AL-0 și afectează corespunzător flagurile: SF=1 , CF=ZF=OF=PF=AF=0 .
jl et	;utilizarea instrucțiunii JL (Jump if Less than) provoacă interpretarea comparației AL < 0 cu semn, adică se testează dacă SF!=OF și cum SF=1 iar OF=0 se decide îndeplinirea condiției și saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JL a decis că dorește să compare -128 cu 0 și cum -128 este “less than” 0 condiția a fost îndeplinită (echiv. cu jnge et). În contrast, jnl et sau jge et (care vor testa dacă SF=OF) NU vor fi îndeplinite și NU vor provoca saltul la eticheta specificată.
jb et	;utilizarea instrucțiunii JB (Jump if Below) provoacă interpretarea comparației AL < 0 fără semn, adică se testează dacă CF=1 și cum CF=0 se decide neîndeplinirea condiției deci nu se va face saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucțiunii JB a decis că dorește să compare 128 cu 0 și cum 128 NU este “below” 0 condiția NU a fost îndeplinită (echivalent cu jnae et sau jc et).

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplu	Explicație
mov al, 80h	
cmp al, 0	
jae et1	;se testează fără semn dacă al $\geq 0$ ( $128 \geq 0?$ ) - CF=0 deci condiție îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la eticheta et1
jbe et2	;se testează fără semn dacă al $\leq 0$ ( $128 \leq 0?$ ) – CF = ZF = 0 deci condiția (CF=1 sau ZF=1) NU este îndeplinită și ca urmare nu se va face saltul la eticheta et2 – rezultat consistent cu jb et, deoarece jbe implică jb (echivalent cu jna et2)
ja et3	;se testează fără semn dacă al $> 0$ ( $128 > 0?$ ) – CF = ZF = 0 deci condiția (CF=0 și ZF=0) este îndeplinită și ca urmare se va face saltul la eticheta et3 (echivalent cu jnbe et3) și rezultat consistent cu jbe et2, deoarece dacă jbe nu este îndeplinită atunci ja trebuie să fie
je et4	;se testează dacă al $= 0$ ( $128 = 0 ?$ ) – nu se pune problema semnului dacă se testează egalitatea! – cum ZF=0, condiția ZF=1 nu este îndeplinită deci nu se va efectua saltul la eticheta et4 (echivalent cu jz et4). În contrast, jne et4 sau jnz et4 (care vor testa dacă ZF=1) vor fi îndeplinite și vor provoca saltul la eticheta specificată.
jle et5	;se testează cu semn dacă al $\leq 0$ ( $-128 \leq 0?$ ) – OF = ZF = 0 și SF=1 deci condiția (ZF=1 sau SF!=OF) este îndeplinită și ca urmare se va face saltul la eticheta et5 (echivalent cu jng et5) și rezultat consistent cu jl et, deoarece jle implică jl.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplu	Explicație
mov al, 80h	
cmp al, 0	
jg et6	;se testează cu semn dacă al > 0 (-128 > 0?) – OF = ZF = 0 și SF=1 deci ;condiția (ZF=0 și SF=OF) NU este îndeplinită și ca urmare NU se va face ;saltul la eticheta et6 (echivalent cu jne et6) și rezultat consistent cu jle ;et5, deoarece dacă jg nu este îndeplinită atunci jle trebuie să fie
jp et7	;se testează dacă PF=1 - PF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (echivalent cu jpe et7 – Jump if Parity Even). În contrast, jnp et7 ;(care testează dacă PF=0 – echivalentă cu jpo et7 – Jump if Parity Odd) va ;fi îndeplinită și saltul se va efectua
jо et8	;se testează dacă OF=1 - OF=0 deci condiție neîndeplinită – nu se efectuează ;saltul (nu există depășire). În contrast, jno et8 (care testează dacă OF=0) va ;fi îndeplinită și saltul se va efectua.
js et9	;se testează dacă în interpretarea cu semn rezultatul comparației are semn ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucțiunii ;CMP, nu este vorba de a interpreta cu semn sau fără semn operanții ;scăderii fictive d-s, ci rezultatul final al acesteia !) adică testăm dacă SF=1 -;condiție îndeplinită în cazul nostru și ca urmare saltul se va efectua !. În ;contrast, jns et9 (care testează dacă SF=0) NU va fi îndeplinită și saltul ;NU se va efectua

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplu	Explicație
mov al, 80h	
cmp 0, al	;eroare de sintaxă : “Illegal immediate” deoarece sintaxa instrucțiunii cmp interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuși dorim forțarea unei 3 comparații de acest tip (0-al) putem utiliza ;pe post de prim operand un registru inițializat cu valoarea 0.
mov bl, 0	
cmp bl, al	;realizează scăderea fictivă bl-al (0-al = 0-80h = 0- 10000000b = ;10000000b) și afectează corespunzător flagurile: CF=SF=OF=1, ;ZF=PF=AF=0.

### Exercițiu propus:

Reluați discuția efectului tuturor instrucțiunilor de salt condiționat de mai sus (analizate pentru cazul comparației (\*) cmp al,0) în condițiile în care această comparație e înlocuită de ultimele două instrucțiuni prezentate, adică în cazul în care se efectuează cmp bl,al cu bl=0.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

**Care ar fi însă justificarea faptului că în cazul  $b1, al$  avem  $CF = OF = SF = 1$  iar în cazul  $cmp\ a1, 0$  doar  $SF=1$  iar  $CF = OF = 0$  ?**

- Pentru a justifica modurile de setare diferite ale flag-urilor trebuie să luăm în discuție regulile practice de setare a acestor flag-uri. Aceste regule generale sunt:
  - SF ia valoarea bitului de semn al rezultatului obținut;
  - CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obținut a provocat ( $CF=1$ ) sau nu ( $CF=0$ ) un transport în afara spațiului de reprezentare; dacă e vorba despre o scădere d-s, avem: dacă  $|d| \geq |s|$  atunci  $CF=0$  (nu e nevoie de cifră de împrumut pentru efectuarea scăderii) iar dacă  $|d| < |s|$  atunci  $CF=1$  (este nevoie de cifră de împrumut pentru efectuarea scăderii și acest lucru se reflectă în CF)
  - OF este setat la valoarea 1 dacă există depășire în interpretarea cu semn a rezultatului (“OF is set if there exists a signed overflow”), adică dacă rezultatul obținut nu se încadrează în intervalul de interpretare admis (acesta fiind  $[-128..+127]$  dacă este vorba despre octeți și respectiv  $[-32768..+32767]$  pentru cuvinte interpretate cu semn).
- Ultimele două regule derivă de fapt din modul de implementare a conceptului de depășire (overflow) la nivelul procesorului 80x86.
- În cazul operațiilor/operanzilor fără semn depășirea va fi semnalată prin setarea indicatorului CF (carry flag). În cazul operațiilor/operanzilor cu semn depășirea va fi semnalată prin setarea indicatorului OF (overflow flag).

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Cum să detectăm însă situațiile de depășire în cazul operațiilor de adunare și scădere ?

- Care sunt regulile practice de aplicat pentru a înțelege și a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ?
- În discuțiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (overflow flag) deoarece și datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situații din partea programatorilor ca fiind depășire sau nu.
- Atragem însă atenția asupra a ceea ce se ignoră de multe ori în acest context și anume faptul că o situație de tipul CF=1 (cu OF=0) semnalează la rândul ei o depășire, însă pentru cazul numerelor interpretate fără semn.
- Pentru ADUNARE: **dacă se adună două numere de același semn și rezultatul este de semn diferit atunci se semnalează depășire (OF=1), în caz contrar nu (OF=0).** Aceasta este deci ceea ce am putea numi **regula depășirii la adunare (RDA)** în cazul interpretării cu semn.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

- De exemplu, la nivel de octet, dacă vom considera adunarea  $100 + 50 = 150$  vom obține depășire (!) cu semn (pare surprinzător, nu-i aşa ?).
- **Justificare:**  
 $100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b)$ .  
Operanții au același semn dar rezultatul este de semn diferit, deci conform RDA vom avea OF=1.
- Intuitiv, depășirea se poate justifica prin faptul că  $150 \notin [-128..127]$  deci se obține o eroare de tip “out of range”. Deși s-ar putea replica faptul că  $150 = 10010110b = -106$  (în interpretarea cu semn), iar  $-106 \in [-128..127]$ , această ultimă interpretare nu poate fi acceptată deoarece operanții (100 și 50) au valori pozitive în ambele interpretări (bitul de semn fiind 0).
- Ca urmare, suma a două numere pozitive nu poate da un număr negativ și astfel singura interpretare ce poate fi acceptată în acest context pentru  $10010110b$  este  $150 \notin [-128..127]$  deci se setează OF=1.
- Pe de altă parte, CF = 0 (nu există cifră de transport în afara spațiului de reprezentare) deci nu avem depășire în interpretarea fără semn: rezultatul adunării  $100 + 50 = 150 \in [0, 255]$  (intervalul de interpretare admis pentru numere fără semn).

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

- Analog, în interpretarea cu semn, **suma a două numere negative nu poate furniza un număr pozitiv.**
- Luăm exemplul:  $10010110b + 10000010b = 1\ 00011000b$   
Se observă din reprezentarea binară că există un transport de cifră 1 în afara spațiului de reprezentare admis al celor 8 biți, deci intuitiv este suficient de justificat depășirea.
- Din punct de vedere al aplicării RDA obținem pe 8 biți în interpretarea cu semn că suma a două numere negative (ele sunt negative deoarece bitul de semn este 1 pentru ambele numere) ar trebui să furnizeze un număr pozitiv:  $00011000b = 18h = 24$ .
- Această valoare este de fapt o trunchiere a valorii binare corecte (pe 9 biți!) ce ar fi trebuit obținută ( $100011000b = 118h$ ), iar trunchierarea are loc tocmai datorită depășirii.
- Ca urmare, nu se poate obține un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depășire!).
- Se observă că o astfel de trunchiere înseamnă întotdeauna și apariția unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat și CF=1.
- $10010110b = 96h = -106$  (în interpretarea cu semn) =  $+150$  (în interpretarea fără semn)  
 $10000010b = 82h = -126$  (în interpretarea cu semn) =  $+130$  (în interpretarea fără semn)

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

- În interpretarea fără semn avem  $150 + 130 = 280 \notin [0..255]$  (justificarea intuitivă a depășirii). Tehnic, am văzut deja că CF = 1 și rezultă astfel clar că avem depășire în interpretarea fără semn.
- Nu putem avea deci  $-106 + (-126) = 24!$  (pentru că  $00011000b = 18h = 24$  în ambele interpretări). Aceasta este sensul în care se aplică RDA aici.
- Un alt mod de justificare intuitivă a depășirii în acest tip de situație este: În interpretarea cu semn avem  $-106 + (-126) = -232 \notin [-128..127]$  deci OF=1.
- Această ultimă motivație este mai intuitivă pentru justificarea depășirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne “cea mai rapid aplicabilă regulă practică din punct de vedere algoritic”
- Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depășire. De asemenea, dacă adunăm două numere de același semn dar rezultatul are același semn cu operanzii nu se va semnala nici în acest caz depășire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeași dimensiune ca și cea a operanzilor). Se poate verifica ușor din punct de vedere matematic că în nici unul din aceste cazuri nu ieșim din intervalul de interpretare admis.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Cum să detectăm însă situațiile de depășire în cazul operațiilor de adunare și scădere ?

- Pentru SCĂDERE: se interpretează operanții respectivi cu semn, se efectuează scăderea solicitată asupra configurațiilor corespunzătoare de biți și dacă rezultatul obținut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul [-128..127] pentru octeții cu semn și respectiv [-32768..32767] pentru cuvinte interpretate cu semn) atunci se semnalează depășire (overflow) și astfel OF=1. Această formulare o putem numi **regula depășirii la scădere (RDS)** pentru cazul interpretării cu semn.
- În cazul depășirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea CF=1, pe care o putem interpreta semnificativ drept “depășire la scădere în interpretarea fără semn”.
- Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum și impactul lor asupra modului de setare al flag-urilor.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 1:

- `mov ah, 82h ;`  $82h = 130$  (interpretarea fără semn) =  $-126$  (interpretarea cu semn)  
`; = 10000010b` (bitul de semn fiind 1 cele două interpretări diferă)  
`mov bh, 2ah ;`  $2ah = 42$  (atât în interpretarea cu semn cât și în cea fără semn)  
`; = 00101010b` (bitul de semn fiind 0 cele două interpretări coincid)  
`cmp ah, bh ;` se realizează scăderea fictivă  $ah-bh=10000010b - 00101010b = 01011000b$   
`; = 58h = 88` (atât în interpretarea cu semn cât și în cea fără semn deoarece  
`; bitul de semn este 0)`

- Această scădere setează flag-urile astfel:

$SF = 1$  (deoarece bitul de semn pentru rezultatul  $A8h = 10101000b$  este 1)

$CF = 1$  (deoarece  $|2ah| < |82h|$  se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine  $42 - 130 = 168$  (!) provenită de fapt din necesitatea unei scăderi de tipul  $(256 + 42) - 130 = 168$  și ca urmare a necesității împrumutului se va semnală depășire în interpretarea fără semn, înțeleasă aici ca “nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut”)

$OF = 1$  (se efectuează scăderea în interpretarea cu semn, adică  $bh-ah = 42-(-126) = +168$  și cum  $+168 \notin [-128..127]$  se semnalează signed overflow și ca urmare  $OF=1$ )

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 2:

- `mov ah, 126 ; echivalent cu mov ah,7eh deoarece 126 = 7Eh = 01111110b (bitul de semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este ;126 atât în interpretarea cu semn cât și în cea fără semn)`  
`mov bh, 2ah ; 2ah = 42 (atât în interpretarea cu semn cât și în cea fără semn)  
; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)`  
`cmp ah, bh ; se realizează scăderea fictivă ah-bh= 01111110b-00101010b = 01010100b  
; = 54h = 84 = 126 - 42 (atât în interpretarea cu semn cât și în cea fără semn  
deoarece bitul de semn al rezultatului este 0)`
- Această scădere setează flag-urile astfel:  
 $SF = 0$  (deoarece bitul de semn pentru rezultatul  $54h = 01010100b$  este 0)  
 $CF = 0$  (deoarece  $|126| > |42|$  nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnaliza depășire în interpretarea fără semn) 7  
 $OF = 0$  (se efectuează scăderea în interpretarea cu semn, adică  $ah-bh = 126 - 42 = 84$  și cum  $84 \in [-128..127]$  NU se semnalează signed overflow și ca urmare  $OF=0$ )

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

### Exemplul 2:

- `mov ah, 126` ; echivalent cu `mov ah, 7eh` deoarece  $126 = 7Eh = 01111110b$  (bitul de semn fiind 0 cele două interpretări coincid, ca urmare conținutul lui AH este ;126 atât în interpretarea cu semn cât și în cea fără semn)  
`mov bh, 2ah` ;  $2ah = 42$  (atât în interpretarea cu semn cât și în cea fără semn)  
; =  $00101010b$  (bitul de semn fiind 0 cele două interpretări coincid)  
`cmp bh, ah` ; se realizează scăderea fictivă  $bh - ah = 00101010b - 01111110b = 10101100b$   
; =  $42 - 126 = ACh = 172$  (în interpretarea fără semn) = -84 (în interpretarea  
; cu semn)
- Această scădere setează flag-urile astfel:  
SF = 1 (deoarece bitul de semn pentru rezultatul ACh =  $10101100b$  este 1)  
CF = 1 (deoarece  $|42| < |126|$  se pune problema unei scăderi cu împrumut de cifră ; în interpretarea fără semn scăderea devine  $42 - 126 = 172$  (!) provenită de fapt din necesitatea unei scăderi de tipul  $(256 + 42) - 126 = 172$  și ca urmare a necesității împrumutului se va semnală depășire în interpretarea fără semn prin setarea flagului carry)  
OF = 0 (se efectuează scăderea în interpretarea cu semn, adică  $bh - ah = 42 - 126 = -84$  și cum  $-84 \in [-128..127]$  NU se semnalează signed overflow și ca urmare OF=0)
- Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii  $a-b \in [-127..127]$  atunci și  $b-a \in [-127..127]$  (situația particulară în care  $a-b = -128$  o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului  $[-32767..32767]$  cu discuție asupra cazului particular  $-32768$ . Ca urmare se poate concluziona faptul că instrucțiunile `cmp a,b` și `cmp b,a` vor furniza întotdeauna aceeași valoare pentru OF.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de saltele condiționat și instrucțiunea cmp

Exemplul 3 - discuție asupra cazurilor cmp 80h,0 și cmp 0,80h:

- `mov ah, 80h ; 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn) ; = 10000000b (bitul de semn fiind 1 cele două interpretări diferă)`  
`mov bh, 0 ; bh:=0`  
`cmp ah, bh ; se realizează scăderea fictivă ah-bh= 10000000b- 00000000b = 10000000b ; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)`
- Această scădere setează flag-urile astfel:  
 $SF = 1$  (deoarece bitul de semn pentru rezultatul  $80h = 10000000b$  este 1)  
 $CF = 0$  (deoarece  $|80h| > |0|$  nu se pune problema unei scăderi cu împrumut de cifră, deci nu poate fi vorba despre depășire în interpretarea fără semn)  
 $OF = 0$  (se efectuează scăderea în interpretarea cu semn, adică  $ah-bh = -128 - 0 = -128$  și cum  $-128 \in [-128..127]$  NU se semnalează signed overflow și ca urmare  $OF=0$ )

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 3 - discuție asupra cazurilor cmp 80h,0 și cmp 0,80h:

- `mov ah, 80h ; 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn) ; = 10000000b (bitul de semn fiind 1 cele două interpretări diferă)`  
`mov bh, 0 ; bh:=0`  
`cmp bh, ah ; se realizează scăderea fictivă bh-ah= 00000000b-10000000b = 10000000b ; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)`
- Această scădere setează flag-urile astfel:  
 $SF = 1$  (deoarece bitul de semn pentru rezultatul  $80h = 10000000b$  este 1)  
 $CF = 1$  (deoarece  $|0h| < |80h|$  se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine  $0 - 128 = 128$  (!) provenită de fapt din necesitatea unei scăderi de tipul  $(256 + 0) - 128 = 128$  și ca urmare a necesității împrumutului se va semnala depășire în interpretarea fără semn prin setarea flagului carry)  
 $OF = 1$  (se efectuează scăderea în interpretarea cu semn, adică  $bh-ah = 0 - (-128) = +128$  și cum  $+128 \notin [-128..127]$  se semnalează signed overflow și ca urmare  $OF=1$ )  
 $CF = 1$  în cazul cmp 0,80h deoarece se efectuează o scădere cu împrumut de tipul :  $0 - 10000000b = 1\ 0000000b - 10000000b = 01000000b$  și cifra de împrumut se transferă în CF.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Să analizăm în acest context ce înseamnă și cum s-a ajuns la domeniul “numerelor cu semn posibil a fi reprezentate pe 1 octet” respectiv domeniul “numerelor cu semn posibil a fi reprezentate pe 1 cuvânt”.

- Pe 1 octet se pot reprezenta 256 de valori, indiferent că vorbim despre interpretarea cu semn sau interpretarea fără semn. În interpretarea fără semn aceste valori sunt cele din intervalul [0..255]. Care sunt însă cele 256 de valori reprezentabile în interpretarea cu semn ? Este vorba despre intervalul [-128..127] sau despre intervalul [-127..128] ? Pentru că nu poate fi vorba despre intervalul [-128..128] deoarece în acest interval sunt 257 de valori ! Cu alte cuvinte cineva a trebuit să aleagă una dintre cele două variante și totodată să facă precizarea că numerele -128 și +128 nu pot coexista între limitele aceluiași interval de reprezentare al aceluiași tip de dată ! (reamintim că în limbaj de asamblare tip de dată = dimensiune de reprezentare)
- În acest sens este de observat și impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât shortint cât și byte în Turbo Pascal acceptă valoarea 80h (-128 ca shortint și +128 ca byte) însă 80h nu poate avea două interpretări distincte în cadrul aceluiași tip de dată ! Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 și +128 ca fiind prezente în cadrul aceluiași tip de dată !
- Ca urmare, s-a luat decizia ca intervalul acceptat al valorilor cu semn reprezentabile pe 1 octet să fie intervalul [-128..+127] (care este exact domeniul de valori și a tipului de dată shortint din Turbo Pascal): deci +128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet !

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Să analizăm în acest context ce înseamnă și cum s-a ajuns la domeniul “numerelor cu semn posibil a fi reprezentate pe 1 octet” respectiv domeniul “numerelor cu semn posibil a fi reprezentate pe 1 cuvânt”.

- Totuși, după cum putem verifica foarte ușor, instrucțiunile mov ah, 128 și mov ah,- 128 sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în ah a configurației binare 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fără semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea cu semn. Simplă încărcare a unui registru cu o anumită configurație binară nu presupune și necesitatea interpretării respectivei configurații într-un anumit fel. Sarcina interpretării acelei configurații drept cu semn sau fără semn va cădea în sarcina instrucțiunilor ce urmează și care vor folosi ca operanzi aceste valori. De exemplu, utilizarea lui IMUL în loc de MUL va provoca interpretarea configurației binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui DIV în loc de IDIV va provoca interpretarea aceluiași operand ca fără semn și.a.m.d.
- În cazul cmp 80h,0 se efectuează  $80h - 0 = 80h = 10000000b$  ( $128 - 0 = 128$  în interpretarea fără semn) fără a fi nevoie de o cifră de transport împrumutată pentru a putea efectua scăderea, deci nu avem depășire în interpretarea fără semn și astfel CF = 0. În interpretarea cu semn a operanzilor și a rezultatului final avem  $-128 - 0 = -128 \in [-128..127]$  deci nu avem depășire nici în interpretarea cu semn și astfel OF = 0. Pe de altă parte, avem evident în ambele cazuri SF=1. Justificarea intuitivă: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea tehnică: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 4 - Să analizăm în continuare modurile în care putem compara valorile 0 și 1 (și apoi 0 și -1) și ce efecte are asupra flagurilor instrucțiunea cmp în fiecare dintre situații:

- Situația cmp 1,0 (evidențiată la nivelul unui text sursă de exemplu prin `cmp ah,0 cu ah=1`) va efectua scăderea fictivă  $1-0 = 1 = 00000001b$ . Efectul asupra flag-urilor va fi  $CF = SF = OF = ZF = PF = AF = 0$ . Justificările sunt evidente pe baza discuțiilor din exemplele anterioare.
- Situația cmp 0,1 (evidențiată la nivelul unui text sursă de exemplu prin `cmp ah,1 cu ah=0`) va efectua scăderea fictivă  $0-1 = -1 = 11111111b$ :  $0 - 00000001b = 1\ 00000000b - 00000001b = 0\ 111111b$ . Efectul asupra flag-urilor va fi  $CF = SF = PF = AF = 1$  și  $ZF = OF = 0$ . Justificarea valorilor din CF și SF este și aici evidentă pe baza discuțiilor din exemplele anterioare iar  $OF=0$  deoarece rezultatul în interpretarea cu semn este -1, iar  $-1 \in [-128..127]$ .
- Situația cmp -1,0 (evidențiată la nivelul unui text sursă de exemplu prin `cmp ah,0 cu ah = -1`) va efectua scăderea fictivă  $-1-0 = -1 = 11111111b$ . Efectul asupra flag-urilor va fi  $SF = PF = 1$  și  $CF = OF = ZF = AF = 0$ . SF=1 deoarece bitul de semn este 1. OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar  $-1 \in [-128..127]$ . CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.
- Situația cmp 0,-1 (evidențiată la nivelul unui text sursă de exemplu prin `cmp ah,-1 cu ah = 0`) va efectua scăderea fictivă  $0 - (-1) = +1 = 00000001b$ :  $0 - 11111111b = 1\ 00000000b - 11111111b = 0\ 00000001b$ . Efectul asupra flag-urilor va fi  $CF = AF = 1$  și  $OF = SF = ZF = PF = 0$ . SF = 0 deoarece bitul de semn este 0. OF=0 deoarece  $0 - (-1) = +1 \in [-128..127]$ . CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica și așa: în interpretarea fără semn această scădere înseamnă de fapt  $0 - 255 = 1$  (!), care trebuie justificată prin  $(256+0) - 255 = 1$ , deci e nevoie de cifră de împrumut și astfel se semnalează depășire în cazul interpretării fără semn, deci CF = 1.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 5: Cazurile studiate anterior (1-4) s-au referit la operații de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucțiunii cmp. Să analizăm în continuare și cazul unei depășiri furnizate de operația de adunare revenind astfel la discuția asupra aplicării regulii RDA:

- `mov ah, 126 ;126 = 01111110b = 7eh` (aceeași valoare 126 în ambele interpretări)  
`add ah, 2 ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b = 7eh + 02h = ;`  
`10000000b = 80h (= 128 fără semn = -128 cu semn)`
- Flag-urile sunt setate astfel:
  - CF = 0 deoarece:  $01111110b + 00000010b = 10000000b$  - nu există transport în afara spațiului de reprezentare al rez.
  - SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operației efectuate este strict negativ = -128).
  - OF = 1 deoarece: - justificare tehnică - conform RDA se adună două numere de același semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).
  - justificare intuitivă - adunăm două numere fără semn a căror sumă este  $126 + 2 = 128$ . Însă numărul  $+128 \notin [-128..127]$  deci se semnalează signed overflow și ca urmare OF=1.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 6: Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situația în care programatorul își initializează operanții cu anumite valori inițiale dorite (cu semn sau fără semn, conform necesităților problemei în cauză) și se așteaptă la obținerea unor rezultate sau reacții în conformitate cu valorile furnizate. Atenție însă! De obicei aceste valori au o dublă interpretare posibilă și nu vor fi interpretate în orice situație sub forma furnizată la inițializare!

- Utilizarea ulterioară a unor instrucțiuni care forțează prin modul de lor de acțiune interpretarea complementară (cu semn/fără semn) celei de la inițializare poate provoca apariția unor situații în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare... Aceasta se întâmplă dacă nu se ține cont în permanență de dubla interpretare posibilă a configurațiilor binare manipulate. Să luăm un exemplu:

```
mov al, 200 ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (cu semn)  
mov bl, -1 ; bl = 11111111b = 0FFh = 255 (fără semn) = -1 (cu semn)  
cmp al, bl ; al-bl = 11001001b = C9h = -55 (cu semn) = 201 (fără semn) (și se setează corespunzător OF=ZF=0 și CF=SF=1)
```

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 6:

- Deci pe cine comparăm de fapt aici? Pe 200 cu -1 sau cum precizează valorile de la initializare? Sau poate pe 200 cu 255? Sau pe -56 cu -1 ? Sau pe -56 cu 255?
- Răspuns: comparăm întotdeauna pe 0C8h cu 0FFh sau în exprimare binară pe 11001000b cu 11111111b. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparației efectuate în baza 10 nu este dedus din acțiunea instrucțiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucțiuni ulterioare care vor avea ele rolul de a interpreta în unul din cele 4 moduri de mai sus comparația efectuată.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 6:

- Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucțiunilor corespunzătoare de salt condiționat:

Exemplu	Explicație
jl et1	; evident că $200 < -1$ deci la prima vedere pare că nu este îndeplinită condiția necesară pentru efectuarea saltului... să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparației ca fiind cu semn (deci -55) aceasta însemnând implicit și faptul că scăderea este interpretată ca $(-56 - (-1))$ deci și operanții vor fi amândoi interpretați cu semn... cum $-56 < -1$ iată că și intuitiv condiția se verifică (pe lângă justificarea tehnică a îndeplinirii condiției de salt SF ≠ OF) și deci saltul se va efectua! Deci chiar dacă programatorul a furnizat la inițializare valorile 200 și -1, utilizarea instrucțiunii JL a provocat interpretarea comparației ca fiind între -55 și -1 și nu între 200 și -1! (explicația de aici și faptul că saltul se va efectua vă poate ajuta să "demonstrați" unor colegi cum 200 poate fi mai mic decât -1 !!!)
ja et2	; deoarece $200 > -1$ în acest caz ne-am așteptă ca saltul să se efectueze... însă utilizarea instrucțiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparație corectă aici este comparația lui 200 cu 255 și cum $200 > 255$ condiția nu este îndeplinită și deci saltul nu se va efectua (iata deci cum se poate "demonstra" că 200 nu este superior valorii -1 !!!). Ca o confirmare, se poate vedea că nici condiția tehnică impusă de JA nu este îndeplinită: ar trebui să avem CF=ZF=0, însă în cazul nostru CF=1 deci saltul nu se va efectua.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 6:

- Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucțiunilor corespunzătoare de salt condiționat:

Exemplu	Explicație
jb et3	; intuitiv $200 < 255$ , iar tehnic CF=1 deci saltul se efectuează
jg et4	; intuitiv $-56 > -1$ , iar tehnic deși ZF=0 nu este îndeplinită și condiția SF = OF deci ; saltul nu se va efectua

- Ca urmare din cele 4 situații teoretic posibile de mai sus, vom întâlni concret numai două: - comparație fără semn (200 cu 255) - impusă de “above” sau “below” - comparație cu semn (-56 cu -1) – impusă de “less than” sau “greater than”
- Nu putem aşadar compara de fapt pe 200 cu -1 aşa cum au fost specificate valorile la inițializare și nici pe -56 cu 255 deoarece interpretarea este ori cu semn ori fără semn pentru ambii operanzi!

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 7: Am studiat în exemplele anterioare modalitatea de reacție (de interpretare) a procesorului 80x86 legată de noțiunea de depășire în cazul operațiilor de adunare și de scădere. Când și cum semnalează însă procesoarele din familia 80x86 depășirea la înmulțire și respectiv la împărțire ?

- “Depășirea” la înmulțire. Instrucțiunile MUL și IMUL setează CF=1 și OF=1 dacă “jumătatea” superioară a produsului (octetul superior dacă este vorba despre produscuvânt sau cuvântul superior dacă este vorba despre produs-dublucuvânt) este o valoare diferită de zero. Aceasta este definiția noțiunii de “depășire la înmulțire” în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincție între MUL și IMUL și de aceea nici între CF și OF. Ori vor fi amândouă flag-urile setate la valoarea 1 cu semnificația de “depășire la înmulțire” în sensul precizat mai sus, ori vor primi amândouă valoarea 0.
- Iată un exemplu pe 8 biți:  

```
mov al, 5
mov bl, 170
mul bl ;AX := AL * BL = 5 * 170 = 850 = 0352h și vom avea CF=1 și OF=1
;deoarece octetul superior AH = 03 ≠ 0.
```

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 7:

- Varianta cu IMUL va furniza:

```
mov al, 5
mov bl, 170 ;170 = 0aah = -86 în interpretarea cu semn
imul bl ;AX := AL * BL = 5 * (-86) = - 430 = 0fe52h și vom avea CF=1 și
          ;OF=1 deoarece octetul superior AH = 0feh ≠ 0.
```

- În cazul unor operanzi pe 16 biți putem avea de exemplu:

```
val1 DW 2000h
val2 DW 0100h ...
mov ax, val1
mul val2 ;DX:AX = 00200000h și vom avea CF=1 și OF=1 deoarece jumătatea
          ;sup. a produsului DX:AX, adică registrul DX conține valoarea 0020h ≠ 0.
```

- Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potențială pierdere de informație ca și în cazul celorlalte depășiri - adunare, scădere sau împărțire. Aceasta deoarece chiar dacă înmulțim valorile maximale posibil a fi reprezentate pe dimensiunea operanzilor ( $255 * 255$  pentru octeți și respectiv  $65535 * 65535$  pentru cuvinte) tot nu se depășește dublul dimensiunii de reprezentare a operanzilor, adică spațiul pe care îl avem oricum la dispoziție prin definiție, deoarece  $255 * 255 = 65025 < 65535$  (numărul maximal fără semn reprezentabil pe un cuvânt) iar  $65535 * 65535 = 4\ 294\ 836\ 225 < 4\ 294\ 967\ 295$  (numărul maximal fără semn reprezentabil pe un dublucuvânt).
- În cazul înmulțirii cu semn (instrucțiunea IMUL) justificarea este similară:  $127 * 127 = 16129 < 32767$  (numărul maximal cu semn ce poate fi reprezentat pe 1 cuvânt), iar  $32767 * 32767 = 1\ 073\ 676\ 289 < 2\ 147\ 483\ 647$  (numărul maximal cu semn reprezentabil pe un dublucuvânt).

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

Exemplul 7:

- Depășirea în cazul înmulțirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanzi octeți (respectiv cuvinte) produsul nu începe tot într-un octet (respectiv într-un cuvânt) ci este realmente nevoie de o dimensiune dublă pentru memorarea rezultatului.
- În acest sens, din punct de vedere matematic s-a specificat clar că înmulțirea nu provoacă de fapt depășire, tocmai din cauza alocării unui spațiu suficient pentru reprezentarea produsului.
- În concluzie, se poate spune că din punct de vedere matematic singura operație care nu provoacă depășire este înmulțirea, însă procesoarele 80x86 promovează totuși noțiunea de “depășire la înmulțire” pentru a diferenția între situațiile în care produsul începe într-un spațiu de dimensiunea operanzilor și în care nu.
- Situațiile în care produsul începe pe dimensiunea operanzilor vor fi caracterizate de setările CF = OF = 0 (nu avem deci depășire la înmulțire).
- Iată un exemplu:

```
mov al, 5
mov bl, 51
mul bl ; AX := AL * BL = 5 * 51 = 255 = 00ffh și vom avea CF=0 și OF=0
          ; deoarece octetul superior AH = 0.
```

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

### Exemplul 7 - Depășirea la împărțire.

- În cazul împărțirii, specificarea acestei operații sub forma  
(I) DIV operand presupune că operandul specificat este împărțitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biți) iar deîmpărțitul este considerat implicit în AX (dacă operand este octet) sau în DX:AX (dacă împărțitorul este cuvânt).
- Efectuarea operației are ca efect: AX : operand pe 8 biți = câtul în AL și restul în AH; DX:AX / operand pe 16 biți = câtul în AX și restul în DX;
- În cazul împărțirii depășirea apare atunci când rezultatul împărțirii nu începe în spațiul rezervat conform definiției pentru reprezentare, mai exact, când câtul nu începe în AL sau respectiv AX. Într-o astfel de situație, procesorul 80x86 emite o întrerupere 0, execuția terminându-se cu un mesaj furnizat de către rutina de tratare a întreruperii 0, de genul “Divide by zero”, “Zero divide” sau “Divide overflow” (în funcție de tipul de procesor și/sau de SO instalat). Pare ciudat la prima vedere că o împărțire prin 0 (de genul div bh cu bh = 0) ce practic nu se poate efectua din punct de vedere matematic este tratată similar ca efect din punct de vedere al limbajului de asamblare cu o împărțire care matematic se poate efectua.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

### Exemplul 7 - Depășirea la împărțire.

- Secvența

```
mov ax, 60000
mov bl, 2
div bl
```

ar trebui să furnizeze din punct de vedere matematic câtul 30000. Însă conform definiției împărțirii DIV acest cât trebuie memorat în registrul AL, de dimensiune octet. Cum cea mai mare valoare reprezentabilă pe 1 octet este 255, este evident astfel că din punct de vedere al limbajului de asamblare împărțirea de mai sus nu se poate efectua (similar cu o situație de tip div 0) și ca urmare înțelegem acum decizia proiectanților de a trata tot prin emiterea unei întreruperi 0 și o situație de genul celei de mai sus. Să remarcăm în acest sens și faptul că mesajul “Divide overflow” (depășire la împărțire) este acceptat în acest context ca similar unui “Divide by zero”.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

### Exemplul 8

- Una dintre erorile logice frecvente pe care o fac programatorii neexperimentați este de a confunda exprimările “numere cu semn” și “numere fără semn” cu exprimările “numere negative” și respectiv “numere pozitive”.
- Numere cu semn nu înseamnă automat numere negative ! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.
- Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunțul unei probleme care cere efectuarea unei anumite acțiuni “dacă numărul v este (strict) negativ”? În primul rând vom concluziona că este vorba despre interpretarea cu semn.
- Se pune însă întrebarea: cum vom testa practic dacă un număr cu semn este negativ sau nu? (să presupunem că v este octet). Fiind vorba despre interpretarea cu semn, dacă primul bit al configurației binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra primului bit din reprezentarea numărului.

# Instrucțiuni

## Manipularea datelor - Modul de acțiune a instrucțiunilor de salt condiționat și instrucțiunea cmp

### Exemplul 8

- Iată două alternative pentru realizarea unui astfel de test:
  - a) Realizăm o deplasare a primului bit în CF și testăm valoarea sa printr-o instrucțiune adecvată de salt condiționat. Secvența

```
mov al, [v] ;pentru a nu afecta destructiv conținutul variabilei v
shl al,1      ;shift stânga cu 1 poziție pentru ca primul bit să treacă în CF.
jc negativ   ;dacă CF=1 atunci salt la eticheta este_negativ asigură testarea faptului dacă variabila v este sau nu un număr negativ.
```
  - b) Utilizăm instrucțiunea cmp pentru o comparație în raport cu 0:

```
cmp byte [v], 0 ;scădere fictivă v-0
jl negativ ;dacă vv atunci salt la eticheta este_negativ
```



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai

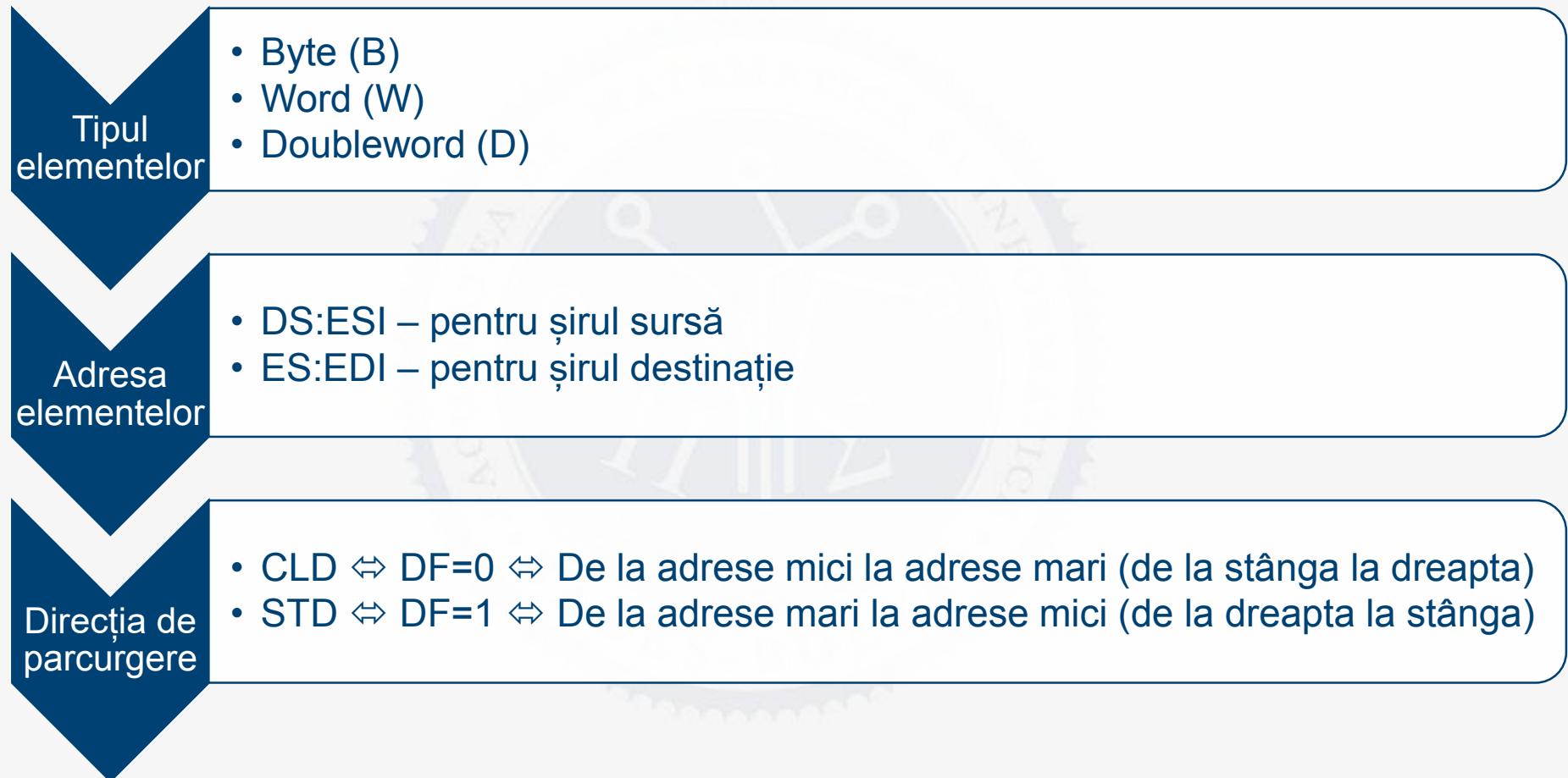




# Instructiuni pe siruri

---

# Caracteristici ale șirurilor



# Caracteristici ale șirurilor

LODS (Load from string) – încarcă B / W / D din memorie de la adresa DS:ESI în registrul AL, AX sau EAX

STOS (Store string) – stochează conținutul registrului AL, AX sau EAX în memorie la adresa ES:EDI

MOVS (Move string data) – copiază B / W / D din memorie de la adresa DS:ESI în B / W / D din memorie de la adresa ES: EDI

CMPS (Compare strings) – compară conținutul B / W / D din memorie de la adresa DS:ESI cu B / W / D din memorie de la adresa ES: EDI

SCAS (Scan string) – compară registrul AL, AX sau EAX cu B / W / D din memorie de la adresa ES: EDI

# Caracteristici ale șirurilor

LODS (Load from string) – încarcă B / W / D din memorie de la adresa DS:ESI în registrul AL, AX sau EAX

- **LODSB** – octetul de la adresa **DS:ESI** este încărcat în **AL**
  - Dacă DF = 0 atunci ESI = ESI + 1 altfel ESI = ESI – 1
- **LODSW** – cuvântul de la adresa **DS:ESI** este încărcat în **AX**
  - Dacă DF = 0 atunci ESI = ESI + 2 altfel ESI = ESI - 2
- **LODSD** – dublucuvântul de la adresa **DS:ESI** este încărcat în **EAX**
  - Dacă DF = 0 atunci ESI = ESI + 4 altfel ESI = ESI - 4

# Caracteristici ale șirurilor

STOS (Store string) – stochează conținutul registrului AL, AX sau EAX în memorie la adresa ES:EDI

- **STOSB** – octetul din registrul **AL** este stocat în octetul de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 1 altfel EDI = EDI – 1
- **STOSW** – cuvântul din registrul **AX** este stocat în cuvântul de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 2 altfel EDI = EDI – 2
- **STOSD** – dublucuvântul din registrul **EAX** este stocat în dublucuvântul de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 4 altfel EDI = EDI – 4

# Caracteristici ale șirurilor

MOVS (Move string data) – copiază B / W / D din memorie de la adresa DS:ESI în B / W / D din memorie de la adresa ES: EDI

- **MOVSB** – copiază **octetul** din memorie de la adresa **DS:ESI** în **octetul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 1 și EDI = EDI + 1,  
altfel ESI = ESI – 1 și EDI = EDI – 1
- **MOVSW** – copiază **cuvântul** din memorie de la adresa **DS:ESI** în **cuvântul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 2 și EDI = EDI + 2,  
altfel ESI = ESI – 2 și EDI = EDI – 2
- **MOVSD** – copiază **dublucuvântul** din memorie de la adresa **DS:ESI** în **dublucuvântul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 4 și EDI = EDI + 4,  
altfel ESI = ESI – 4 și EDI = EDI – 4

# Caracteristici ale șirurilor

CMPS (Compare strings) – compară conținutul B / W / D din memorie de la adresa DS:ESI cu B / W / D din memorie de la adresa ES: EDI

- **CMPSB** – compară **octetul** din memorie de la adresa **DS:ESI** cu **octetul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 1 și EDI = EDI + 1,  
altfel ESI = ESI – 1 și EDI = EDI – 1
- **CMPSW** – compară **cuvântul** din memorie de la adresa **DS:ESI** cu **cuvântul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 2 și EDI = EDI + 2,  
altfel ESI = ESI – 2 și EDI = EDI – 2
- **CMPSD** – compară **dublucuvântul** din memorie de la adresa **DS:ESI** cu **dublucuvântul** din memorie de la adresa **ES:EDI**
  - Dacă DF = 0 atunci ESI = ESI + 4 și EDI = EDI + 4,  
altfel ESI = ESI – 4 și EDI = EDI – 4

# Caracteristici ale șirurilor

SCAS (Scan string) – compară registrul AL, AX sau EAX cu B / W / D din memorie de la adresa ES: EDI

- **SCASB** – compară **octetul** din registrul **AL** cu **octetul** de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 1 altfel EDI = EDI – 1
- **SCASW** – compară **cuvântul** din registrul **AX** cu **cuvântul** de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 2 altfel EDI = EDI – 2
- **SCASD** – compară **dublucuvântul** din registrul **EAX** cu **dublucuvântul** de la adresa **ES:EDI**
  - Dacă DF = 0 atunci EDI = EDI + 4 altfel EDI = EDI – 4

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0 ⇔ directia de parcurgere de la stanga la dreapta
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir; ESI = offset sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB; AL = 88h; ESI = ESI + 1
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW; AX = 6677h, ESI = ESI + 2
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD; EAX = 22334455h, ESI = ESI + 4

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r; EDI = offset r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah; AL = 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB; byte [EDI]=[r]=1Ah, EDI = EDI + 1
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h; AX = 1234h
STOSW
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW; word [EDI] = [r+1] = 1234h, EDI = EDI + 2
MOV EAX, 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh; EAX = 567890cdh
STOSD
```

# Caracteristici ale șirurilor

```
segment data ...
sir dq 1122334455667788h, 1a2b3c4d5e6faabbh
lung_sir equ ($-sir)/8
r times lun_sir dq 0
segment code...
CLD; DF = 0
MOV ESI, sir
LODSB
LODSW
LODSD

MOV EDI, r
MOV AL, 1Ah
STOSB
MOV AX, 1234h
STOSW
MOV EAX, 567890cdh
STOSD; dword [EDI] = [r+3] = 567890cdh
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1
STD
CMPSB
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1; ESI = offset a + 2
MOV EDI, b
ADD EDI, lb
STD
CMPSB
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1; EDI = offset b + 2
STD
CMPSB
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1
STD; DF = 1
CMPSB
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1
STD
CMPSB; "CMP 11h, 56h", ESI = ESI - 1, EDI = EDI-1
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1
STD
CMPSB
JE equal
JNE nonEqual
```

**MOV AX, 0FFFFh; AX = FFFFh**

```
SCASW
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Caracteristici ale șirurilor

```
segment data ...
a db 10h, 12h, 11h
la equ $-a
b db 98h, 34h, 56h
lb equ $-b
segment code...
MOV ESI, a
ADD ESI, la - 1
MOV EDI, b
ADD EDI, lb - 1
STD
CMPSB
JE equal
JNE nonEqual
```

```
MOV AX, 0FFFFh
SCASW; "CMP AX, 5634h", EDI =
EDI - 2
JG maiMare
JLE maiMic
equal:
...
nonEqual:
...
maiMare:
...
maiMic:
...
```

# Prefixe de instrucțiuni repetitive

REP – Repeat while ECX > 0

REPZ, REPE – Repeat while  
ZF = 1 and ECX > 0

REPNZ, REPNE – Repeat  
while ZF = 0 and ECX > 0

# Exemplu: copierea unui sir de cuvinte într-un alt sir

```
segment data ...
s dw 1234h, 5678h
ls equ ($-s)/2
d times ls dw 0
segment code ...
MOV ECX, ls
MOV ESI, 0
MOV EDI, 0
JECXZ final
repeta:
    MOV AX, [s+ESI]
    MOV [d+EDI], AX
    add ESI, 2
    add EDI, 2
LOOP repeta
final:
```

Varianta 1

# Exemplu: copierea unui sir de cuvinte într-un alt sir

```
segment data ...
s dw 1234h, 5678h
ls equ ($-s)/2
d times ls dw 0
segment code ...
MOV ECX, ls
MOV ESI, s
MOV EDI, d
JECXZ final
CLD
repeta:
    LODSW
    STOSW
LOOP repeta
final:
```

Varianta 2

# Exemplu: copierea unui sir de cuvinte într-un alt sir

```
segment data ...
s dw 1234h, 5678h
ls equ ($-s)/2
d times ls dw 0
segment code ...
MOV ECX, ls
MOV ESI, s
MOV EDI, d
JECXZ final
CLD
repeta:
    MOVSW
LOOP repeta
final:
```

Varianta 3

## Exemplu: copierea unui sir de cuvinte într-un alt sir

```
segment data ...
s dw 1234h, 5678h
ls equ ($-s)/2
d times ls dw 0
segment code ...
MOV ECX, ls
MOV ESI, s
MOV EDI, d
JECXZ final
CLD
REP MOVSW
final:
```

Varianta 4



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai

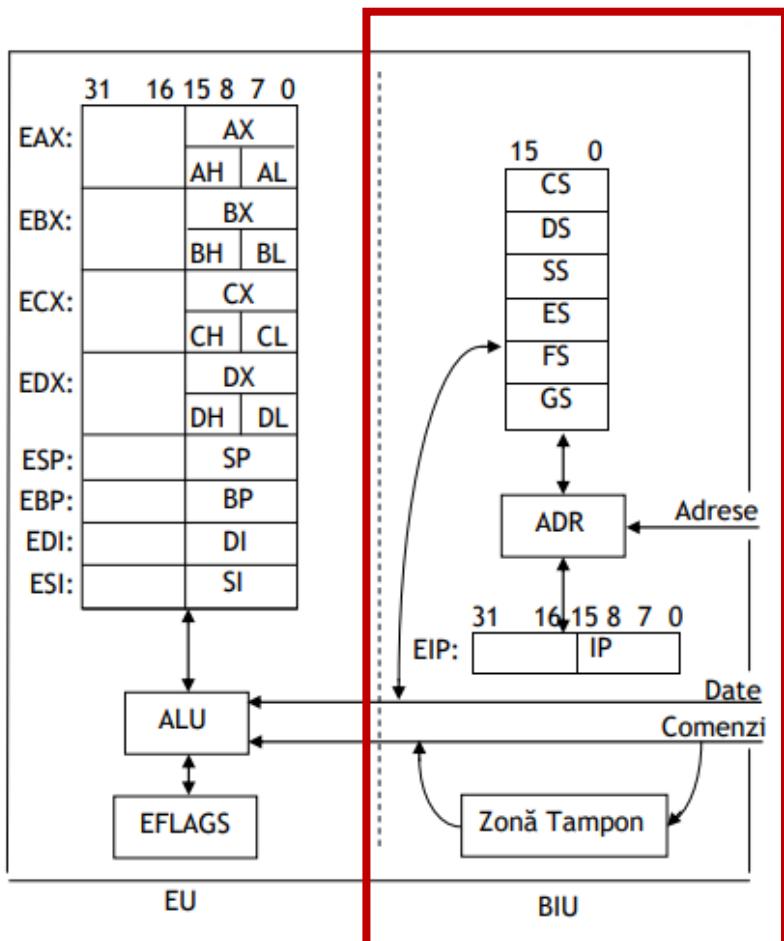




# Sistemul de adresare al Arhitecturii x86

---

# Sistemul de adresare al Arhitecturii x86



- Nici o instrucțiune din ASM nu permite folosirea ambilor operanzi expliciti din memorie
- Componența ADR din BIU se folosește pentru calcul de adrese
- Adresarea se face în 2 pași (de aceea există registri de segment)

## SEGMENTARE

Adresa de segment

Offset / deplasament în cadrul segmentului

# Regiștrii de adresă și calculul de adresă

- **ADRESA UNEI LOCAȚII** = numărul de octeți consecutivi dintre începutul memoriei RAM și începutul locației respective
- **SEGMENT** = o succesiune continuă de locații de memorie menită să deservească scopuri similare în timpul execuției unui program

# Regiștrii de adresă și calculul de adresă

## CLASIFICARE:

- dpdv logic (*logical segment*)

**SEGMENT** = diviziune logică a memoriei unui program, caracterizată prin **ADRESA DE BAZĂ** (început), **LIMITĂ** (dimensiune, sizeof) și **TIPUL** acesteia (cod, date, stivă, extra)

- dpdv fizic (*physical segment*)

**SEGMENT** = un bloc de memorie de dimensiune fixă

- 64 KB pentru procesoare pe 16 biți
- 4 GB pentru procesoare pe 32 biți

# Regiștri de adresă și calculul de adresă

- **OFFSET sau DEPLASAMENT** = adresa unei locații față de începutul unui segment

Un offset e valid  $\Leftrightarrow$  valoarea sa numérica pe 32 biți nu depășește LIMITA segmentului în care se raportează

- **SPECIFICARE DE ADRESĂ** = pereche formată din **SELECTOR DE SEGMENT** și un **OFFSET**

specificare de adresă = adresă segment : offset

(16 biți) (32 biți)

$s_3 s_2 s_1 s_0 : o_7 o_6 o_5 o_4 o_3 o_2 o_1 o_0$

- **SELECTOR DE SEGMENT** = valoare pe 16 biți care identifică în mod unic segmentul accesat și caracteristicile lui  
= un descriptor de segment

# Regiștri de adresă și calculul de adresă

- $S_3S_2S_1S_0$  :

$O_7O_6O_5O_4O_3O_2O_1O_0$

Furnizat de SO

baza segmentului este  $b_7b_6b_5b_4b_3b_2b_1b_0$  și are limita  $I_7I_6I_5I_4I_3I_2I_1I_0$ .

# Regiștri de adresă și calculul de adresă

- **ADRESA DE SEGMENTARE** (ADRESĂ LINIARĂ) = baza + offset  
 $a_7a_6a_5a_4a_3a_2a_1a_0 = b_7b_6b_5b_4b_3b_2b_1b_0 + o_7o_6o_5o_4o_3o_2o_1o_0$ .



E făcută de ADR, nu se poate accesa de către programatori

- Adresă FAR = adresă completă, specificare de adresă
- Adresă NEAR = adresă care se precizează doar prin offset

# Regiștrii de adresă și calculul de adresă

8:1000h

Specificare de adresă

Pentru a calcula adresa liniară ce-i corespunde acestei specificări, procesorul va proceda după cum urmează:

1. **Verifică dacă segmentul ce corespunde valorii de selector 8 a fost definit de către sistemul de operare și se blochează accesul dacă nu a fost definit un astfel de segment;**
2. **Extrage adresa de bază (B) și limita acestui segment (L),** de exemplu, ca rezultat am putea avea  $B = 2000h$  și  $L = 4000h$ ; (este o operație la ale cărei detalii NU avem acces, ea derulându-se exclusiv între procesor și SO)
3. **Verifică dacă offsetul depășește limita segmentului:**  $1000h > 4000h$ ? În caz de depășire accesul ar fi fost blocat (memory violation error);
4. **Adună offsetul cu B,** obținând în cazul nostru adresa liniară  $3000h$  ( $1000h + 2000h$ ). Acest calcul este efectuat de către componenta ADR din BIU.

Acest mecanism de adresare poartă numele de **segmentare**, vorbind astfel despre **modelul de adresare segmentată**.

# Regiștrii de adresă și calculul de adresă

În cazul în care segmentele încep la adresa 0 și au dimensiunea maximă posibilă (4GB), orice offset este automat valid și segmentarea nu contribuie efectiv în calculul adreselor.

Astfel, având  $b_7b_6b_5b_4b_3b_2b_1b_0 = 00000000$ , calculul de adresă pentru adresa logică  $s_3s_2s_1s_0 : o_7o_6o_5o_4o_3o_2o_1o_0$  va rezulta în adresa liniară:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + o_7o_6o_5o_4o_3o_2o_1o_0$$

$$a_7a_6a_5a_4a_3a_2a_1a_0 := o_7o_6o_5o_4o_3o_2o_1o_0$$

Acet mod particular de utilizare a segmentării, folosit de către majoritatea sistemelor de operare moderne poartă numele de **model de memorie flat**.

# Regiștrii de adresă și calculul de adresă

- Procesoarele x86 suportă și un mecanism de control al accesului la memorie numit **paginare**, independent de adresarea segmentată.
- Atât calculul de adrese cât și folosirea mecanismelor de segmentare și paginare sunt influențate de modul de execuție al procesorului, procesoarele x86 suportând următoarele moduri de execuție mai importante:
  - **mod real**, pe 16 biți (folosind cuvânt de memorie de 16 biți și având memoria limitată la 1MB);
  - **mod protejat** pe 16 sau 32 biți, caracterizat prin folosirea paginării și segmentării;
  - **mod virtual** 8086, permite rulare programelor de tip mod real alături de cele de mod protejat;
  - **long mode**, pe 64 sau 32 biți, unde paginarea este obligatorie în timp ce segmentarea este dezactivată.

În cadrul cursului nostru ne vom concentra asupra arhitecturii și comportamentului procesoarelor din familia Intel x86 în **modul protejat pe 32 de biți**.

# Regiștrii de adresă și calculul de adresă

Arhitectura x86 permite folosirea a patru tipuri de segmente cu roluri diferite:

- **segment de cod**, care conține instrucțiuni mașină;
  - **segment de date**, care conține date asupra cărora se acționează în conformitate cu instrucțiunile;
  - **segment de stivă**;
  - **segment suplimentar de date** (extrasegment).
- 
- În fiecare moment este ACTIV cel mult câte un segment din fiecare tip
  - CS, DS, SS, ES – contin valorile selectorilor segmentelor active corespunzătoare fiecărui tip
  - Doar segmentul de cod este obligatoriu!

# Regiștrii de adresă și calculul de adresă

Noțiune	Reprezentare	Descriere
Specificare de adresă, adresă logică, adresă FAR	Selector <sub>16</sub> :offset <sub>32</sub>	Definește complet atât segmentul cât și deplasamentul în cadrul acestuia
Selector de segment	16 biți	Identifică unul dintre segmentele disponibile. Ca valoare numerică acesta codifică poziția descriptorului de segment selectat în cadrul unei tabele de descriptori.
Offset, adresă NEAR	Offset <sub>32</sub>	Definește doar componenta de offset (considerând segmentul cunoscut ori folosirea modelului de memorie flat)
Adresă liniară (adresă de segmentare)	32 biți	Inceput segment + offset, reprezintă rezultatul calculului de adresă
Adresă fizică efectivă	Cel puțin 32 biți	Rezultatul final al segmentării plus, eventual, paginării. Adresa finală obținută de către BIU, indicând în memoria fizică (hardware)

# Regiștrii de adresă și calculul de adresă

- Pentru a adresa o locație din memoria RAM sunt necesare două valori: una care să indice segmentul, alta care să indice offsetul în cadrul segmentului.
- Pentru a simplifica referirea la memorie, microprocesorul derivă, în lipsa unei alte specificări, adresa segmentului din **unul dintre regiștrii de segment CS, DS, SS sau ES**.
- Alegerea implicită a unui registru de segment se face după niște reguli proprii instrucțiunii folosite.

# Regiștrii de adresă și calculul de adresă

## Adrese NEAR

- Prin definiție, o adresă în care se specifică doar offsetul, urmând ca segmentul să fie preluat implicit dintr-un registru de segment poartă numele de **adresă NEAR** (adresă apropiată).
- O adresă NEAR se află întotdeauna în interiorul unuia din cele patru segmente active.

# Regiștrii de adresă și calculul de adresă

## Adrese FAR

- O adresă în care programatorul indică explicit un selector de segment poartă numele de **adresă FAR** (adresă îndepărtată).
- O adresă FAR este deci o SPECIFICARE COMPLETA DE ADRESĂ și ea se poate exprima la nivelul unui program în trei moduri:
  - $s_3s_2s_1s_0$  : specificare\_offset, unde  $s_3s_2s_1s_0$  este o constantă;
  - registru\_segment : specificare\_offset, registru segment fiind CS, DS, SS, ES, FS sau GS;
  - **FAR [variabilă]**, unde variabilă este de tip QWORD și conține cei 6 octeți constituind adresa FAR (ceea ce numim variabila pointer in limbajele de nivel înalt)
- **Formatul intern al unei adrese FAR** este: la adresa mai mică se află offsetul, iar la adresa mai mare cu 4 (dublucuvântul care urmează după dublucuvântul curent) se află cuvântul ce conține selectorul care indică segmentul.
- Reprezentarea adreselor respectă principiul **reprezentării little-endian**: partea cea mai puțin semnificativă are adresa cea mai mică, iar partea cea mai semnificativă are adresa cea mai mare.

# Regiștrii de adresă și calculul de adresă

## Calculul offsetului unui operand. Moduri de adresare

- În cadrul unei instrucțiuni există 3 moduri de a specifica un operand pe care aceasta îl solicită:

- **modul registru**, dacă pe post de operand se află un registru al mașinii;

mov eax, 17



E specificat în mod registru

# Regiștrii de adresă și calculul de adresă

## Calculul offsetului unui operand. Moduri de adresare

- În cadrul unei instrucțiuni există 3 moduri de a specifica un operand pe care aceasta îl solicită:
    - **modul registru**, dacă pe post de operand se află un registru al mașinii;  
`mov eax, 17`
    - **modul imediat**, atunci când în instrucțiune se află chiar valoarea operandului (nu adresa lui și nici un registru în care să fie conținut);  
`mov eax, 17`
- 
- E specificat in mod imediat

# Regiștrii de adresă și calculul de adresă

## Calculul offsetului unui operand. Moduri de adresare

- În cadrul unei instrucțiuni există 3 moduri de a specifica un operand pe care aceasta îl solicită:
  - **modul registru**, dacă pe post de operand se află un registru al mașinii;  
`mov eax, 17`
  - **modul imediat**, atunci când în instrucțiune se află chiar valoarea operandului (nu adresa lui și nici un registru în care să fie conținut);  
`mov eax, 17`
  - **modul adresare la memorie**, dacă operandul se află efectiv undeva în memorie. În acest caz, offsetul lui se calculează după următoarea formulă:

$$\text{adresa\_offset} = [ \text{bază} ] + [ \text{index} \times \text{scală} ] + [ \text{constanta} ]$$

# Regiștrii de adresă și calculul de adresă

Calculul offsetului unui operand. Moduri de adresare

$$\text{adresa\_offset} = [ \text{bază} ] + [ \text{index} \times \text{scală} ] + [ \text{constanta} ]$$

- Deci **adresa\_offset** se obține din următoarele (maxim) patru elemente:
  - conținutul unuia dintre regiștrii EAX, EBX, ECX, EDX, EBP, ESI, EDI sau **ESP** ca **bază**;
  - conținutul unuia dintre regiștrii EAX, EBX, ECX, EDX, EBP, ESI sau EDI drept **index**;
  - factor numeric (**scală**) pentru a înmulți valoarea registrului index cu 1, 2, 4 sau 8 - valoarea unei constante numerice, pe octet, cuvant sau dublucuvânt.
- De aici rezultă următoarele moduri de adresare la memorie:
  - **Directă**, atunci când apare numai constanta;
  - **Indirectă**, atunci când apare unul dintre regiștrii de bază sau de index;
    - **bazată**, dacă în calcul apare unul dintre regiștrii bază;
    - **scalat-indexată**, dacă în calcul apare unul dintre regiștrii index;
- Cele trei moduri de adresare a memoriei pot fi combinate. De exemplu, poate să apară adresare directă bazată, adresare bazată și scalat-indexată etc

# Regiștrii de adresă și calculul de adresă

Calculul offsetului unui operand. Moduri de adresare

- Adresarea care NU este **directă** se numește **adresare indirectă** (bazată și/sau indexată). Deci o **adresare indirectă** este cea pt care avem specificat cel puțin un registru între parantezele drepte.
- La instrucțiunile de salt mai apare și un alt tip de adresare numit **adresare relativă**.
- Adresa relativă indică poziția următoarei instrucțiuni de executat, în raport cu poziția curentă. Poziția este indicată prin numărul de octeți de cod peste care se va sări.
- Arhitectura x86 permite atât adrese relative scurte (SHORT Address), reprezentate pe octet și având valori între -128 și 127, cât și adrese relative apropriate (NEAR Address), pe dublucuvânt cu valori între -2147483648 și 2147483647.

# Regiștri de adresă și calculul de adresă

```
;EBX = 17152
MOV EAX, EBX; EAX = ?
MOV EAX, [EBX]; EAX = ?
MOV EAX, [EBX + 4*ESI - 17]
```

```
a dw 21712
```

```
...
```

```
MOV AX, [a]; AX = ?
```

# Regiștri de adresă și calculul de adresă

```
;EBX = 17152
MOV EAX, EBX; EAX = 17152 - ambii operanzi de tip regisztr
MOV EAX, [EBX]; EAX = ?
MOV EAX, [EBX + 4*ESI - 17]

a dw 21712
...
MOV AX, [a]; AX = ?
```



# Regiștri de adresă și calculul de adresă

```
;EBX = 17152
```

```
MOV EAX, EBX; EAX = 17152
```

**MOV EAX, [EBX];** EAX = dublucuvântul din memoria RAM de la adresa 17152, adică EAX = 793456F0h (! Little endian)

- op sursă este op din memorie cu adresare indirectă
- op destinație este op registru

```
MOV EAX, [EBX + 4*ESI - 17]
```

```
a dw 21712
```

...

```
MOV AX, [a]; AX = ?
```



# Regiștri de adresă și calculul de adresă

```
;EBX = 17152
MOV EAX, EBX; EAX = 17152
MOV EAX, [EBX]; EAX = dublucuvântul din memoria RAM de la
adresa 17152, adică EAX = 793456F0h (! Little endian)
MOV EAX, [EBX + 4*ESI - 17]

a dw 21712
...
MOV AX, [a]; AX = 21712
```



# Regiștri de adresă și calculul de adresă

```
;EBX = 17152
MOV EAX, EBX; EAX = 17152
MOV EAX, [EBX]; EAX = dublucuvântul din memoria RAM de la
adresa 17152, adică EAX = 793456F0h (! Little endian)
MOV EAX, [EBX + 4*ESI - 17]

a dw 21712
...
MOV AX, [a]; AX = 21712
```

Ce este o variabilă? Ce fel de operand este [a]? Mod registru, mod imediat, mod adresare la memorie?



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

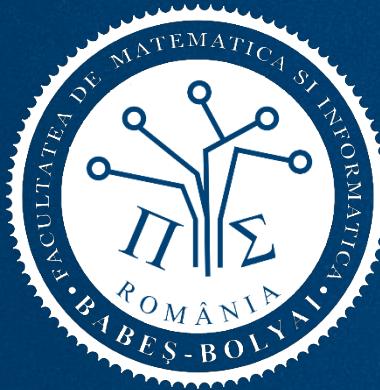
Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Aritmetica de pointeri

---

# Aritmetică de pointeri

Care sunt operațiile ARITMETICE cu pointeri permise IN INFORMATICA ?

**MOV EAX, [v1]; []** = operatorul de derefențiere

**a[7] = \* (a+7); \*** = operatorul de derefențiere

- Aritmetică de pointeri/adrese = utilizarea de expresii aritmetice, care au ca operanzi adrese

# Aritmetică de pointeri

Care sunt operațiile ARITMETICE cu pointeri permise IN INFORMATICA ?

- adunări și scăderi de adrese ?
  - Adunare de adrese = ? CE reprezinta ? **NIMIC !!!!**
  - Scădere de adrese = ? CE reprezinta ?  $q-p =$  nr octeți dintre cele două adrese de memorie (niciodată nu depăşim dim memoriei ; valoarea obținută este o **CONSTANTA NUMERICA** )
- adunări și scăderi de constante la o/dintr-o adresă = necesare și utile pt accesarea elementelor dintr-un array
- Înmulțirea a două adrese ? - nepermisă (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata).
- Inmultirea cu o constantă ? - (in majoritatea cazurilor valoarea obtinuta este dincolo de limita maxima a memoriei posibile a fi accesata). In plus, CE reprezinta valoarea obtinuta ?... Nimic util !
- Impărțire ? ... No way !

# Aritmetică de pointeri

- Singura excepție de la regulile aritmeticii de pointeri o constituie **formula de calcul a offsetului unui operand** unde sunt permise adunări de valori de registri (NU adunări de pointeri!!)... În rest nu există excepții

```
MOV AX, a[7]
```

Ce e a[7]?

```
a[7]=*(a+7)=*(7+a)=7[a]; atât în C cât și în asamblare
```

# Aritmetică de pointeri

- DOAR 3 operații sunt permise cu **POINTERI**:

- **Scăderea a două adrese**

adresa – adresa = ok

$q-p$  = scadere de 2 pointeri = `sizeof(array)` sau nr de elemente (in C) / octeti (asamblare) dintre două adrese de memorie

=> valoare SCALARĂ !!! (valoare numerică constantă imediată)

- **Adunarea unei constante numerice la o adresă**

adresa + constanta numerică

identificarea unui element prin indexare –  $a[7]$  ,  $q+9$

=> **POINTER**

- **Scăderea unei constante numerice dintr-o adresă**

adresa – constanta numerică –  $a[-4]$  ,  $p-7$

=> **POINTER**

# Aritmetică de pointeri

**v db 17**

**ADD EDX, [EBX + ECX \* 2 + v - 7];** – ok!

**MOV EBX, [EBX + ECX \* 2 - v - 7];** – Syntax error !

Invalid effective address – impossible segment base multiplier

**ADC ECX, [EBX + ECX \* 2 + a + b - 7];** – Syntax error din cauza "a+b"; invalid effective address – impossible segment base multiplier

**SUB [EBX + ECX\*2 + a - b - 7], EAX;** – ok! pentru că a-b este o operație corectă cu pointeri

# L-value vs R-value

## Valoare stângă vs. valoare dreaptă a unei atribuiri.

- Atribuire:  $i := i + 1$

Q: i-ul din stanga este tot una cu i-ul din dreapta?

# L-value vs R-value

## Valoare stângă vs. valoare dreaptă a unei atribuiri.

- Atribuire:  $i := i + 1$   
adresa lui  $i <- \text{valoarea lui } i + 1$
- LHS (valoarea stanga a unei atribuiri este o L-value = adresa)
- RHS (valoarea dreapta a unei atribuiri este o R-Value = continut)
- Sintaxele majorității limbajelor de programare prevăd că:  
 $\text{Symbol} := \text{expression\_value}$ , adică  $\text{Identifier} := \text{expresie}$
- În fapt, sunt limbaje (C++, ASAMBLARE) care permit mai general sintaxa:  
 $\text{Expresie\_calcul\_de\_adresa} := \text{valoare\_expresie\_aritmetică}$

```
mov dword [ebx+2*EDX+v-7], a+2 ; este corect?  
mov dword [ebx+2*EDX+v-7], [a+2] ; este corect?
```

# L-value vs R-value

## Valoare stângă vs. valoare dreaptă a unei atribuiri.

- Atribuire:  $i := i + 1$   
adresa lui  $i <- \text{valoarea lui } i + 1$
- LHS (valoarea stanga a unei atribuiri este o L-value = adresa)
- RHS (valoarea dreapta a unei atribuiri este o R-Value = continut)
- Sintaxele majorității limbajelor de programare prevăd că:  
 $\text{Symbol} := \text{expression\_value}$ , adică  $\text{Identifier} := \text{expresie}$
- În fapt, sunt limbaje (C++, ASAMBLARE) care permit mai general sintaxa:  
 $\text{Expresie\_calcul\_de\_adresa} := \text{valoare\_expresie\_aritmetică}$

```
mov dword [ebx+2*EDX+v-7], a+2 ; este corect? - DA
mov dword [ebx+2*EDX+v-7], [a+2] ; este corect? - NU
```

## L-value vs R-value

### Valoare stângă vs. valoare dreaptă a unei atribuiriri.

Q: Operatorul condițional ternar este L-value sau R-value?

- $(a + 2 ? b : c) = x + y + z ;$  este corect?
- $(a + 2 ? 1 : c) = x + y + z ;$  este corect?

# L-value vs R-value

## Valoare stângă vs. valoare dreaptă a unei atribuiri.

Q: Operatorul condițional ternar este L-value sau R-value?

- $(a + 2 ? b : c) = x + y + z$ ; este corect? - DA
- $(a + 2 ? 1 : c) = x + y + z$ ; este corect? - NU! Syntax error

# L-value vs R-value

## Valoare stangă vs. valoare dreaptă a unei atribuiriri.

- Address computation Expression := expression\_value  
 $\text{In C++ } f(\bar{a}+3, \ b-2, \ 2) = x+y+z$
- Variabilele “referință C++” au 3 utilizări:
  - `Int& j = i;` // j devine ALIAS pt i
  - Transmiterea de variabile prin referință la apelul de subprograme  
`float f(int&x, y)`
  - Returnarea de L-valori prin intermediul funcțiilor  
`Int& f(x, i) { ...return v[i]; }`  
Funcția f returnează o LHS (valoare stângă)  
 $f(a, 7) = 79$ ; înseamnă că  $v[7] = 79$
- De asemenea, separat de acestea se permite și utilizarea operatorului condițional ternar pe post de valoare stângă:  
 $(a+2?b:c) = x+y+z$  ; - correct  
 $(a+2?1:c) = x+y+z$ ; - syntax error !!! 1:=n !!!!



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto: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

- Elementele cu care lucrează un asamblor sunt:
  - Etichete
    - Nume scrise de utilizator cu ajutorul cărora se pot referi date sau zone de memorie
    - JMP **label**  
    a DB 2
  - Instrucțiuni
  - Directive
  - Contor de locații

# Elementele de bază ale limbajului de asamblare

- Elementele cu care lucrează un asamblor sunt:
  - Etichete
  - **Instrucțiuni**
    - Scrise sub forma unor mnemonice care sugerează acțiunea; ASAMBLORUL generează octetii care codifică instrucțiunea respectivă
    - Sunt menite pentru a ajunge la procesor
    - În EU -> ALU
    - Dacă operanții sunt din memorie intervine BIU -> ADR
  - Directive
  - Contor de locații

# Elementele de bază ale limbajului de asamblare

- Elementele cu care lucrează un asamblor sunt:
  - Etichete
  - Instrucțiuni
  - Directive
    - Sunt indicații date asamblorului în scopul generării corecte a octetilor
    - Cel mai important task pentru asamblor este generarea octetilor (vezi coloana 2 din debugger)
    - Relația dintre modulele obiect, definirea unor segmente, indicații de asamblare condiționată, directive de generare a datelor
    - Intersecția cu instrucțiunile este vidă
  - Contor de locații

# Elementele de bază ale limbajului de asamblare

- Elementele cu care lucrează un asamblor sunt:
  - Etichete
  - Instrucțiuni
  - Directive
  - Contor de locații
    - număr întreg gestionat de asamblor.
    - valoarea contorului de locatii exprima deplasamentul curent în cadrul segmentului.
    - mai exact, ca și valoare concreta \$ reprezinta offset-ul începutului de linie care conține expresia respectiva.
    - programatorul poate utiliza această valoare (accesare doar în citire!) prin simbolul '\$'.
    - fiecare segment are propriul său contor de locații
    - Este low level, specific limbajului de asamblare

# Elementele de bază ale limbajului de asamblare

- Particularitate NASM

- \$ - pointează către “aici” (POINTER)
- \$\$ - pointează către începutul secțiunii curente (POINTER)  
  
(\$-\$) – distanța de la începutul segmentului / secțiunii (SCALAR)  
= dimensiunea curentă a secțiunii  
= numărul de octeți generați corespunzător instrucțiunilor și directivelor deja întâlnite în cadrul segmentului sau sectiunii respective.

```
section .data
db 'hello'
db 'h', 'e', 'l','l','o'
data_segment_size equ $-$; 10 octeți scriși
```

# Elementele de bază ale limbajului de asamblare

Formatul unei linii sursă

**[eticheta[:]] [prefixe] [mnemonică] [operanzi] [:comentariu]**

**aici: jmp acolo;** avem etichetă + mnemonică + operand + comentariu

**repz cmpsd;** prefix + mnemonică + comentariu

**start:** ; etichetă + comentariu

**;** doar un comentariu (care putea lipsi și el)

**a dw 19872, 42h ;** etichetă + mnemonică + 2 operanzi + comentariu

# Elementele de bază ale limbajului de asamblare

- La nivelul limbajului de asamblare se întâlnesc două categorii de etichete:
  - **etichete de cod**, care apar în cadrul secvențelor de instrucțiuni cu scopul de a defini destinațiile de transfer ale controlului în cadrul unui program.  
Pot apărea și în segmente de date !
  - **etichete de date**, care identifică simbolic unele locații de memorie, din punct de vedere semantic ele fiind echivalentul noțiunii de variabilă din alte limbaje.  
Pot apărea și în segmente de cod !
- Valoarea unei etichete în limbaj de asamblare este un număr întreg reprezentând adresa instrucțiunii, directivei sau datelor ce urmează etichetei.

# Elementele de bază ale limbajului de asamblare

- Distincția dintre referirea adresei unei variabile sau a conținutului asociat acesteia în NASM se face după regulile:
  - Când este specificat **între paranteze drepte, numele variabilei** desemnează **valoarea variabilei**, de exemplu [p] specifică accesarea valorii variabilei p
  - În orice alt context **numele variabilei** reprezintă **adresa variabilei**, spre exemplu, p este întotdeauna adresa variabilei p;

```
MOV EAX, et ; încarcă în registrul EAX adresa datelor sau a codului marcat cu eticheta et (4 octeți)
```

```
MOV EAX, [et] ; încarcă în registrul EAX conținutul de la adresa et (4 octeți)
```

```
lea EAX, [v] ; încarcă în registrul EAX adresa (offsetul) variabilei v (4 octeți)
```

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

# Elementele de bază ale limbajului de asamblare

- Moduri de adresare:
  - operanzi imediați, operanzi registru și operanzi în memorie
- Valoarea operanzilor este calculată:
  - **în momentul asamblării** pentru *operanzii imediați* și pentru offset-urile reprezentând *adresarea directă*,
  - **în momentul încărcării programului** pentru *adresarea directă* (adresa FAR)
  - **în momentul execuției** pentru *operanzii registru* și *cei adresati indirect*.

?:offset (assembly time)

0708h:offset (loading time)

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor imediați

- Operanzii imediați sunt formați din date numerice constante calculabile la momentul asamblării.
- Constantele întregi se specifică prin valori binare, octale, zecimale sau hexazecimale.
- Adițional, este permisă folosirea caracterului \_ (underscore) pentru a separa grupuri de cifre.
- Baza de numerație poate fi precizată în mai multe moduri:
  - Folosind sufixele H sau X pentru hexazecimal, D sau T pentru zecimal, Q sau O pentru octal și B sau Y pentru binar; în aceste cazuri numărul trebuie să înceapă obligatoriu cu o cifră între 0 și 9, pentru a nu exista confuzii între constante și simboluri, de exemplu, OABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol
  - În stil C, prin prefixare cu 0x sau 0h pentru hexazecimal, 0d sau 0t pentru zecimal, 0o sau 0q pentru octal, respectiv 0b sau 0y pentru binar;

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

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor imediați

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

MOV EAX, et ; transfer în registrul EAX a adresei asociate etichetei et va putea fi evaluată la momentul asamblării drept de exemplu

MOV EAX, 8 ; distanță de 8 octeți față de începutul segmentului de date

- “Constanța” acestor valori derivă din regulile de alocare adoptate de limbaje de programare în general și care statuează că **ordinea de alocare în memorie** a variabilelor declarate (mai precis distanța față de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanțele salturilor destinație în cazul unor instrucțiuni de tip **goto** sunt valori constante pe parcursul execuției unui program.
- Adică: o variabilă odată alocată în cadrul unui segment de memorie nu își va schimba niciodată locul alocării (adică poziția sa față de începutul aceluia segment) iar această informație determinabilă la momentul asamblării derivă din **ordinea specificării variabilelor la declarare** în cadrul textului sursă și din **dimensiunea de reprezentare** dedusă pe bază **informației de tip asociate**.

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor registru

- Modul de invocare /accesare directă

MOV EAX, EBX

- Invocare/accesare indirectă - pentru a indica locațiile de memorie

MOV EAX, [EBX]

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor din memorie

- Operanții din memorie: cu adresare **directă** și cu adresare **indirectă**.
- Operandul cu **adresare directă** este o constantă sau un simbol care reprezintă adresa (*segment și deplasament*) unei instrucțiuni sau a unor date.
- Acești operanți pot fi etichete (de ex: jmp et, var1 dw 324, add EAX, [b]), nume de proceduri (de ex: call proc1) sau valoarea contorului de locații (de ex: b db \$-a).
- **Deplasamentul unui operand** cu adresare directă este calculat **în momentul asamblării (assembly time)**.
- **Adresa fiecărui operand** raportată la structura programului executabil (mai precis stabilirea segmentelor la care se raportează deplasamentele calculate) este calculată **în momentul editării de legături (linking time)**.
- **Adresa fizică efectivă** este calculată **în momentul încărcării programului pentru execuție (loading time** – acest proces final de ajustare a adreselor numindu-se **RELOCAREA ADRESELOR** = Address Relocation).

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor din memorie

- Un deplasament utilizat ca operand în cadrul unui program este întotdeauna raportat la un registru de segment.
- Acest registru poate fi specificat explicit sau, în caz contrar, se asociază de către asamblor în mod implicit un registru de segment.
- Regulile limbajului de asamblare pentru asociările implicate sunt:
  - **CS** pentru etichete de cod destinație ale unor salturi (`jmp`, `call`, `ret`, `jz` etc);
  - **SS** în adresări SIB ce foloseste EBP sau ESP drept bază (indiferent de index sau scală);
  - **DS** pentru restul accesărilor de date.

# Elementele de bază ale limbajului de asamblare

## Operatorul de specificare a segmentului

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

segment:expresie

**ss: [EBX+4];** deplasamentul e relativ la SS

**ES: [082h] ;** deplasamentul e relativ la ES

**10h:var ;** segmentul este indicat de selectorul 10h, iar offsetul este valoarea etichetei var.

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor din memorie

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

JMP FAR CS:....

JMP FAR DS:....

JMP FAR [label2]

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor din memorie

```
MOV EAX, [v] ; MOV EAX, DWORD PTR DS:[405000]
MOV EAX, [EBX] ; MOV EAX, DWORD PTR DS:[EBX]
MOV EAX, [EBP] ; MOV EAX, DWORD PTR SS:[EBP]
MOV EAX, [EBP*2] ; MOV EAX, DWORD PTR SS:[EBP+EBP]
MOV EAX, [EBP*3] ; MOV EAX, DWORD PTR SS:[EBP+EBP*2]
MOV EAX, [EBP*4] ; MOV EAX, DWORD PTR DS:[EBP*4]
MOV EAX, [EBX+ESP] ; EAX <- dword care incepe la adresa SS:[ESP+EBX]
MOV EAX, [ESP+EBX] ; EAX <- dword care incepe la adresa SS:[ESP+EBX]
MOV EAX, [EBX+ESP*2] ; syntax error - ESP nu poate fi index !
MOV EAX, [EBX+EBP*2] ; EAX <- dword care incepe la adresa DS:[EBX+2*EBP]
MOV EAX, [EBX+EBP] ; EAX <- ...DS:...
MOV EAX, [EBP+EBX] ; EAX <- ...SS:...
MOV EAX, [EBX*2+EBP] ; EAX <- ...SS:...
MOV EAX, [EBX*1+EBP] ; EAX <- ...SS:...
MOV EAX, [EBP*1+EBX] ; EAX <- ...DS:...
MOV EAX, [EBX*1+EBP*1] ; EAX <- ...SS... ; Primul gasit cu * e index ! EBP - baza
MOV EAX, [EBP*1+EBX*1] ; EAX <- ...DS....; Primul gasit cu * e index! EBX - baza
MOV EAX, [EBP*1+EBX*2] ; EAX <- ...SS:..._
```

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor cu adresare indirectă

- Operanzii cu *adresare indirectă* utilizează regiștri pentru a indica **adrese din memorie**.
- Deoarece valorile din regiștri se pot modifica la momentul execuției, adresarea indirectă este indicată pentru a opera în mod dinamic asupra datelor.
- Forma generală pentru accesarea indirectă a unui operand de memorie este dată de formula de calcul a offset-ului unui operand:

```
[ registru_de_bază ] + [ registru_index * scală ] + [ constantă ]
```

- **Constanta** este o expresie a cărei valoare este determinabilă la momentul asamblării

```
[EBX + edi + table + 6]; table, 6 sunt constante
```

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor cu adresare indirectă

- Operanzii **registru\_de\_bază** și **registru\_index** sunt folosiți de obicei pentru a indica o adresă de memorie referitoare la un tablou.
- În combinație cu factorul de scalare, mecanismul este suficient de flexibil pentru a permite acces direct la elementele unui tablou de înregistrări, cu condiția ca dimensiunea în octeți a unei înregistrări să fie 1, 2, 4 sau 8.

```
mov DH, [EDX + ECX * 4 + 3]; octetul superior al cuvântului  
superiorului elementului de tip DWORD cu index dat în ECX, parte a  
unei vector de înregistrări al cărui adresă (a vectorului)  
este în EDX este încărcat în DH
```

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor cu adresare indirectă

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

**push dword [EAX + EBX]**; salvează pe stivă dublucuvântul de la adresa EAX + EBX, asamblorul având libertatea de a considera EAX drept bază și EBX drept index sau invers, EBX drept bază și EAX drept index

**pop DWORD [ecx]**; restaurează vârful stivei în variabila cu adresa dată de ECX, asamblorul putând interpreta ECX fie ca bază fie ca index.

Toate codificările luate în considerare de către asamblor sunt echivalente iar decizia finală a asamblorului nu are impact asupra funcționalității codului rezultat.

# Elementele de bază ale limbajului de asamblare

## Utilizarea operanzilor cu adresare indirectă

- asamblorul permite și exprimări non-standard cu condiția ca acestea să fie transformabile într-un final în forma standard de mai sus.

**lea EAX, [EAX\*2]**; încarcă în EAX valoarea lui EAX\*2 (adică, EAX devine  $2 \times \text{EAX}$ )

- asamblorul poate decide între codificare de tip  $\text{bază} = \text{EAX} + \text{index} = \text{EAX}$  și  $\text{scală} = 1$  sau  $\text{index} = \text{EAX}$  și  $\text{scală} = 2$ .

**lea EAX, [EAX\*9 + 12]**; EAX ia valoarea  $\text{EAX} * 9 + 12$

- Deși scală nu poate fi 9, asamblorul nu va emite aici un mesaj de eroare. Aceasta deoarece el va observa posibila codificare a adresei drept:  $\text{bază} = \text{EAX} + \text{index} = \text{EAX}$  cu  $\text{scală} = 8$ , unde de această dată valoarea 8 este corectă pentru scală. Evident, instrucțiunea putea fi precizată mai clar sub forma  
`lea EAX, [EAX + EAX * 8 + 12]`.

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



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Sistemul de adresare al Arhitecturii x86

---

# Regiștrii de adresă și calculul de adresă

**MOV EAX, [2\*EDX - 7];** - ok!

**MOV EAX, [ECX + 4\*EDX + 18];** - ok!

**MOV EAX, 23;** - respectă formula? NU!

**MOV EAX, [23];** - ok! Dar se asamblează "aiurea" în OllyDbg

**MOV EDX, [ECX + 2\*ESP + 7];** - syntax error! ESP nu poate fi index

**MOV EAX, [EBX \* 2];** - ok!

**MOV EAX, [EBX \* 3];** - ok!

**MOV EAX, [EBX + EBX \* 2];** - ok!  $\Leftrightarrow$  **MOV EAX, [EBX \* 3]**

# Regiștrii de adresă și calculul de adresă

```
PUSH dword [EAX*9 + 12] ; - ok!
PUSH dword [EAX + EAX * 8 + 12] ; - ok! ⇔ PUSH dword
[EAX*9 + 12]

MOV EAX, [ESP * 5] ; - syntax error! ESP nu poate fi
index
MOV EAX, [EBP * 7] ; - syntax error!

MOV EAX, [EBX + 2] ; - ok!
MOV EAX, EBX + 2; - syntax error!

MOV EAX, [EBX + EDX] ; - ok!
MOV EAX, EBX + EDX; - syntax error!
MOV EAX, [EBX - EDX] ; - syntax error! Invalid effective
address
```



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto: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 și analiza operanzilor
- Unii operatori lucrează cu constante întregi, alții cu valori întregi memorate, iar alții cu ambele tipuri de operanzi.
- **Operatorii efectuează calcule cu valori constante SCALARE determinabile la momentul asamblării (valori scalare = valori immediate), cu excepția adunării și scaderii unei constante la un/dintr-un pointer (care va furniza în totdeauna “pointer data type”) și cu excepția formulei de calcul al offset-ului unui operand (care permite operatorul ‘+’).**
- Instrucțiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuției. Operatorul de adunare (+) efectuează adunarea în momentul asamblării; instrucțiunea ADD efectuează adunarea în timpul execuției.

# Elementele de bază ale limbajului de asamblare

## Utilizarea operatorilor

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

# Elementele de bază ale limbajului de asamblare

## Utilizarea operatorilor

Prioritate	Operator	Tip	Rezultat
7	-	Unar, prefixat	Complement față de 2 (negare): $-X = 0 - X$
7	+	Unar, prefixat	Fără efect (oferit pentru simetrie cu „-”): $+X = X$
7	$\sim$	Unar, prefixat	Complement față de 1: <code>mov AL, ~0 =&gt; mov AL, 0xFF</code>
7	!	Unar, prefixat	Negare logică: $!X = 0$ când $X \neq 0$ , altfel 1
6	*	Binar, infix	Înmulțire: $1 * 2 * 3 = 6$
6	/	Binar, infix	Câtul împărțirii fără semn: $24 / 4 / 2 = 3$ ( $-24/4/2 = 0\text{FDh}$ )
6	//	Binar, infix	Câtul împărțirii cu semn: $-24 // 4 // 2 = -3$ ( $-24 / 4 / 2 \neq -3!$ )
6	%	Binar, infix	Restul împărțirii fără semn: $123 \% 100 \% 5 = 3$
6	%%	Binar, infix	Restul împărțirii cu semn: $-123 \% \% 100 \% \% 5 = -3$

# 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 biți către stânga: $1 << 4 = 16$
4	>>	Binar, infix	Deplasare pe biți la dreapta: $0xFE >> 4 = 0x0F$
3	&	Binar, infix	ȘI: $0xF00F \& 0xFFFF = 0x0006$
2	^	Binar, infix	SAU exclusiv: $0xFF0F ^ 0xFFFF = 0x0FF0$
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 biți

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

OPERATOR	SINTAXA	SEMNIFICATIE
$\sim$	$\sim$ expresie	complementare biți
$\&$	expr1 & expr2	ȘI bit cu bit
$ $	expr1   expr2	SAU bit cu bit
$^$	expr1 ^ expr2	SAU exclusiv bit cu bit
!	$!0 = 1, !x = 0, x \neq 0$	Negare logică

# Elementele de bază ale limbajului de asamblare

## Operatori logici pe biți

& - operatorul SI bit cu bit AND – instrucțiune	$x \text{ AND } 0 = 0$ $x \text{ AND } 1 = x$	$x \text{ AND } x = x$ $x \text{ AND } \sim x = 0$
Operație utilă pt forțarea valorii anumitor biți la 0 !!!!		
- operatorul SAU bit cu bit OR – instrucțiune	$x \text{ OR } 0 = x$ $x \text{ OR } 1 = 1$	$x \text{ OR } x = x$ $x \text{ OR } \sim x = 1$
Operație utilă pt forțarea valorii anumitor biți la 1 !!!!		
^ - op. SAU EXCLUSIV bit cu bit; XOR – instrucțiune	$x \text{ XOR } 0 = x$ $x \text{ XOR } 1 = \sim x$	$x \text{ XOR } x = 0$ $x \text{ XOR } \sim x = 1$
Operație utilă pt complementarea valorii anumitor biți !!!		
XOR AX, AX ; AX=0 !!!		
! Negare logică: $\neg x = 0$ când $x \neq 0$ , altfel 1		
$\sim$ Complement față de 1: mov al, $\sim 0 \Rightarrow$ mov AL, 0ffh		

# Elementele de bază ale limbajului de asamblare

## Operatori logici pe biți

~ **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 biți

MOV EAX, ! [a]	; <b>syntax error</b> 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]	; <b>syntax error</b>
MOV EAX, ! 7	; EAX = 0 ;
MOV AH, ~7	; 7 = 00000111b deci ~7 = 11111000b = f8h
MOV EAX, ~7	; EAX = -8 (FFFF FFF8h)
MOV EAX, !ebx	; <b>syntax error</b>
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

$$5 \mid 6 + 7 \& 8 = (5 \mid 6) + (7 \& 8) = 7 + 0 = 7 ??$$

$$\begin{aligned}5 \mid 6 + 7 \& 8 &= 5 \mid (6 + 7) \& 8 \\&= 5 \mid 13 \& 8 = 5 \mid 8 = 13 = \text{0Dh} !!!\end{aligned}$$

- 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

- Atenție la diferența dintre operatori și instrucțiuni

```
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 și a unor operanzi păstrați în memorie. Sintaxa pentru aceștia este **tip expresie** unde specifikatorul de tip este unul dintre cuvintele cheie **BYTE**, **WORD**, **DWORD**, **QWORD**.
- Această construcție sintactică forțează ca **expresie** să fie tratată ca având dimensiunea de reprezentare **tip**, fără însă a-i modifica definitiv (destructiv) valoarea în sensul precizat de conversia dorită.
- De aceea, aceștia sunt considerați **operatori de conversie (temporară nedestructivă)**.
- Pentru operanzii păstrați în memorie, **tip** poate fi **BYTE**, **WORD**, **DWORD**, **QWORD** având dimensiunile de reprezentare 1, 2, 4, 8 octeți.
- Pentru etichetele de cod el poate fi **NEAR** (adresă pe 4 octeți) sau **FAR** (adresă pe 6 octeți).

**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 intervene niciodată explicit în cod pe 32 de biți.

Exemple unde e necesar un specificator de dimensiune al operanzilor:

```
mov [mem], 12  
push [mem]
```

```
(i) div [mem]  
pop [mem]
```

```
(i) mul [mem]
```

`push 15` – aici este o inconsistență 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 biți)

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

**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 octeți 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 constantă , 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 și datele în momentul asamblării.

# Elementele de bază ale limbajului de asamblare

## Directiva SEGMENT

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

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

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

# Elementele de bază ale limbajului de asamblare

## Directiva SEGMENT

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

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

# Elementele de bază ale limbajului de asamblare

## Directiva SEGMENT

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

- Argumentul opțional **aliniere** specifică multiplul numărului de octeți la care trebuie să înceapă segmentul respectiv.
- Alinierile acceptate sunt puteri a lui 2, între 1 și 4096. Dacă argumentul aliniere lipsește, atunci se consideră implicit că este vorba despre 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 optional **combinare** controlează modul în care segmente cu același nume din cadrul altor module vor fi combinate cu segmentul în cauză la momentul editării de legături.
- Valorile posibile sunt:
  - **PUBLIC** - indică editorului de legături să concateneze acest segment cu alte eventuale segmente cu același nume, obținându-se un unic segment a cărei lungime este suma lungimilor segmentelor componente.
  - **COMMON** - specifică faptul că începutul acestui segment trebuie să se suprapună peste începutul tuturor segmentelor ce au același nume. Se obține un segment având dimensiunea egală cu cea a celui mai mare segment având același nume.
  - **PRIVATE** - indică editorului de legături că acest segment nu este permis a fi combinat cu altele care poartă același nume.
  - **STACK** - segmentele cu același nume vor fi concatenate. În fază de execuție 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 în lipsa precizării acestui argument.
- Argumentul **clasa** are rolul de a permite stabilirea ordinii în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeași clasă vor fi plasate într-un bloc contiguu de memorie indiferent de ordinea lor în cadrul codului sursă.
- Nu există o valoare implicită de inițializare pentru acest argument, el fiind nedefinit în lipsa specificării, ducând în consecință la evitarea concatenării într-un bloc continuu a tuturor segmentelor definite astfel.

# 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 declarații 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 octeți (QWORD - 64 biți)

# Elementele de bază ale limbajului de asamblare

## Directive pentru definirea datelor

```
segment data
    var1 DB 'd' ;1 octet
    .a DW 101b ;2 octeți
    var2 DD 2bfh ;4 octeți
    .a DQ 307o ;8 octeți (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 declarații 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 neinițializate:

- RESB - date de tip octet (BYTE)
- RESW - date de tip cuvânt (WORD)
- RESD - date de tip dublucuvânt (DWORD)
- RESQ - date de tip 8 octeți (QWORD - 64 biți)

- **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 declarații 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 instrucțiuni sau definiții de dată:

```
segment data  
    Tabchar TIMES 80 DB 'a'; crează un tablou de 80 de octeți  
    inițializați fiecare cu codul ASCII al caracterului 'a'  
    matrice10x10 times 10*10 dd 0; va furniza 100 de dublucuvinte dispuse  
    continuu în memorie începând de la adresa asociată etichetei  
    matrice10x10.
```

- TIMES poate fi aplicat și pe instrucțiuni: **TIMES factor instrucțiune**

```
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 sir de caractere unei etichete fără alocarea de spațiu de memorie sau generare de octeți. 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 conține 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', ...

<b>lga db \$-a</b>	- scăderea a 2 pointeri = scalar (constanta numerică) - lga = variabila de memorie (mov [lga],...)
<b>lga dw \$-\$</b>	ok !
<b>lga EQU \$-a</b>	ok ! însă mov [lga],... este syntax error !!! pt că lga NU este o variabilă alocată... ci o constantă! (fără alocare de memorie)
<b>lga dw \$-data</b>	corect în TASM/MASM, INCORRECT în NASM sub 32 biți !!! syntax error – “Expression is not simple or relocatable”
<b>lga dw lga-a</b>	ok !
<b>b EQU 27</b>	ok! Atenție b NU este un offset !
<b>c dd 12345678h</b>	ok !
<b>lga dw b-a</b>	syntax error ! b NU este un offset !
<b>lga dw c-a</b>	ok !
<b>lga dw \$-a-4</b>	ok !
<b>lg dw \$-a</b>	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 același 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 obținem 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 definiție recursivă ! Dacă a este primul element definit în segment consideră a=0 (offset-ul față de începutul segmentului) și 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 <b>aici</b> , start definit <b>altundeva</b> ) expression is not simple or relocatable !
dd start-a	; ok !(start definit <b>altundeva</b> și a definit <b>aici</b> ) - rez=POINTER, deoarece etichetele NU fac parte din același segment și este interpretată ca scădere FAR = pointer
dd start-start1	; ok ! (ambele etichetele sunt definite în același segment) => rez=SCALAR pt că avem scădere de offset-uri definite în același 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 începe 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 și a din data)
    mov cx, a-$            ; OK (a definit altundeva și $ 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 începe 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 definiție de date ca expresie de
                  ; inițializare, 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  $et1$  și  $et2$  sunt etichete – fie de cod, fie de date) sunt acceptate sintactic de către NASM,
    - Fie dacă ambele sunt definite în același segment
    - Fie dacă  $et1$  aparține unui segment diferit față de cel în care apare expresia, iar  $et2$  este definită în segmentul în care apare expresia. Într-un astfel de caz, TD asociat expresiei  $et1 - et2$  este POINTER și NU SCALAR (constantă numerică) ca și în cazul etichetelor ce fac parte din același segment. (Deci ALTUNDEVA – AICI merge, însă AICI – ALTUNDEVA NU)
  - Scădere de offset-uri ce se raportează la același 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 în memorie în faza de execuție*" însă aceasta nu ne este disponibilă/accesibilă, fiind decisă la momentul încărcării programului pt execuție de către încărcătorul de programe al SO.
  - De aceea, dacă folosim nume de segmente în expresii din program în mod explicit vom obține fie **eroare de sintaxă** (în situații de tipul \$/a-data - "AICI – ALTUNDEVA") fie de **link-editare** (în situații 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" și 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 biți de exemplu, unde `nume_segment` în faza de asamblare = offset-ul său = 0). Ca urmare, se constată că în progr. sub 32 de biți 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 biți 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 și la programarea sub 16 biți. Din acest punct de vedere sub 32 biți este o mare diferență între numele de variabile (al căror offset poate fi determinat la asamblare) și numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca și o constantă, această valoare fiind cunoscută la fel ca și adresa de segment doar la momentul încărcării programului pt execuție - loading time).
- Dacă avem mai mulți operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinație validă de operații 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 biți 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 și la programarea sub 16 biți. Din acest punct de vedere sub 32 biți este o mare diferență între numele de variabile (al căror offset poate fi determinat la asamblare) și numele de segmente (al căror offset NU poate fi determinat la momentul asamblării ca și o constantă, această valoare fiind cunoscută la fel ca și adresa de segment doar la momentul încărcării programului pt execuție - loading time).
- Dacă avem mai mulți operanzi pointeri, asamblorul va încerca să încadreze expresia într-o combinație validă de operații cu pointeri:

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



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto: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

## Directive de definire de date

<b>a1 db 0,1,2,'xyz'</b>	; 00h 01h 02h 'x' 'y' 'z' ⇔ 00h 01h 02h 78h 79h 7Ah ; offset(a1) - determinat la încărcarea lui OllyDbg) = 00401000h ; offset(a1) - determinat la asamblare de către NASM = 0
<b>db 300, "F"+3</b>	; 2Ch 49h -Warning - byte data (300 = 1 2Ch) exceeds bounds!
<b>a2 TIMES 3 db 44h</b>	; 44h 44h 44h ; offset a2 = 00401008h, însă offsetul determinat la asamblare de către NASM va fi 8
<b>a3 TIMES 11 db 5,1,3</b>	; 05h 01h 03h ... de 11 ori (33 octeți)
<b>a4 dw a2+1, 'bc'</b>	; offset(a2)=00401008h; a2+1=00401009h; deci se generează 09h 10h 'b' 'c' = 09h 10h 62h 63h (2 cuvinte - words) 09 10 (corect, DAR... această valoare particulară 10h este calculabilă doar DUPA INCARCAREA PROGRAMULUI (LOADING) - deci offset-ul începuturilor de segmente este și el determinabil doar la momentul încărcării programului - LOADING TIME) Offset-ul variabilelor față de începutul segmentelor în care apar sunt constante (de tip pointer, NU scalar !) determinabile la momentul asamblării
<b>a41 dw a2+1, 'b', 'c'</b>	; 09h 10h 'b' 00h 'c' 00h = 09h 10h 62h 00h 63h 00h (3 cuvinte - words)
<b>a42 db a2+1</b>	; - syntax error - OBJ format can only handle 16 or 32 bits relocation !
<b>a44 dw 1009h</b>	; 09h 10h

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

a5 dd a2+1, 'bcd'	; 09h 10h 40h 00h 62h 63h 64h 00h
a6 TIMES 4 db '78'	; 37h 38h 37h 38h 37h 38h 37h 38h
a61 TIMES 4 db '7' , '8'	; 37h 38h 37h 38h 37h 38h 37h 38h
a62 TIMES 4 dw '78'	; 37h 38h 37h 38h 37h 38h 37h 38h
a7 db a2	; syntax err. OBJ format can only handle 16- or 32- relocation (echiv. cu mov ah,a2)
a8 dw a2	; 08h 10h
a9 dd a2	; 08h 10h 40h 00h
a10 dq a2	; 08h 10h 40h 00h 00h 00h 00h 00h
a11 db [a2]	<ul style="list-style-type: none"><li>- expression syntax error - pt că [a2] NU este o expresie validă acceptată de către asamblor, nereprezentând "o valoare constantă determinabilă la momentul asamblării" ! Dereferențierea implicată aici este ceea ce deranjează asamblorul</li><li>- Conținutul unei zone de memorie sau al unui registru NU sunt valori constante determinabile la momentul asamblării. Acestea sunt accesibile și determinabile DOAR la momentul execuției !! (run-time).</li></ul>

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

a12 dw [a2]	; expression syntax error
a13 dd dword [a2]	; expression syntax error
a14 dq [a2]	; expression syntax error
a15 dd eax	; expression syntax error
a16 dd [eax]	; expression syntax error
mov ax, v	; Warning - 32 bit offset in 16 bit field

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

- Pașii urmați de un program de la codul sursă până la rulare:
  - Verificarea sintactică (realizată de *asamblor/compiler/interpreter*)
  - Fișierele **.OBJ** sunt generate de către *asamblor/compiler*
  - Faza de linking (realizată de LINKER = o unealtă furnizată de sistemul de operare, care verifică posibilele DEPENDENȚE dintre aceste fișiere/module OBJ); Rezultatul → fișier **.EXE**
  - Tu (utilizatorul) activezi fișierul exe (prin clic sau apăsarea tastei Enter...)
  - LOADER-ul sistemului de operare caută spațiul necesar în memoria RAM pentru fișierul EXE. Când îl găsește, încarcă fișierul EXE și realizează RELOCALIZAREA ADRESELOR
  - La final, loader-ul cedează controlul procesorului prin specificarea PUNCTULUI DE INTRARE al programului (ex: eticheta start)
  - Faza de run-time începe ACUM...

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date - Tipuri de date asociate operanzilor

- Directivele de definire a datelor în NASM NU sunt mecanisme de definire a tipurilor de date
- a db 17,19  
b dw 1234h ; 34h 12h  
c dd....
- Rolul directivelor de definire a datelor NU este în NASM de a preciza tipul de dată al variabilelor definite, ci DOAR de a genera octeții corespunzători acelor zone de memorie pe care le ocupă în conformitate cu directiva specifică aleasă și respectând ordinea de plasare de tip little-endian
- Deci a NU este de tip byte – ci doar un offset/deplasament și atât... un simbol desemnând începutul unei zone de memorie FARA VREUN TIP ASOCIAT
- Iar b NU este de tip word – ci doar un offset/deplasament și atât ... un simbol desemnand începutul unei zone de memorie FARA VREUN TIP ASOCIAT
- Si nici c NU este de tip doubleword – ci doar un offset/deplasament și atât ... un simbol desemnand începutul unei zone de memorie FARA VREUN TIP ASOCIAT

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date - Tipuri de date asociate operanzilor

- Atunci DE CE mai asociem în definiție o directivă de tip de dată ? Pt a INDICA ASAMBLORULUI CUM să populeze cu date/initializeze zona de memorie respectivă (fie ca o secvență de bytes, fie ca una de words, fie ca una de doublewords)
- Directivele de date se referă la modalitatea de initializare concretă a unei zone de memorie și nu asociază atributul de tip de dată cu un simbol
- Deci: **directive de definire** a unei date NU este un mecanism de asociere de tip de dată pentru o variabilă, ci doar un mecanism de initializare a zonei de memorie alocate variabilei cu valorile dorite

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date - Tipuri de date asociate operanzilor

- Numele unei variabile este asociat, în limbaj de asamblare, cu offset-ul ei relativ la segmentul în care apare definiția acesteia.
- Offset-urile variabilelor definite într-un program sunt întotdeauna valori constante, determinabile în momentul asamblării/compilării.
- Limbajul de asamblare și C sunt limbaje orientate pe valori, ceea ce înseamnă că totul se reduce, în final, la o valoare numerică — aceasta fiind o caracteristică de nivel scăzut.
- Într-un limbaj de programare de nivel înalt, programatorul poate accesa memoria doar folosind nume de variabile; în schimb, în limbajul de asamblare, memoria este/poate/trebuie accesată DOAR prin folosirea formulei de calcul a offset-ului („formula de la două noaptea”), unde este utilizată și aritmetica pointerilor (aceasta fiind folosită și în C!).
- `mov ax, [ebx]` – operandul sursă nu are un tip de date asociat (reprezintă doar începutul unei zone de memorie) și, din acest motiv, în cazul instrucțiunii MOV, operandul destinație este cel care decide tipul de date al transferului (un word în acest caz), iar transferul se va face în conformitate cu reprezentarea little endian.

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

- Segment code (incepe intotdeauna la offset 00402000 - CINE decide asta ?)
- Linkeditorul ia deciziile de acest tip. Adresa de baza pentru incarcarea PE-urilor, cel putin cea implicita (setata de catre linkeditorul de la Microsoft si nu numai) este 0x400000 in cazul executabilelor (respectiv 0x10000000 pentru biblioteci).
- Alink respecta aceasta conventie si completeaza in campul ImageBase al structurii IMAGE\_OPTIONAL\_HEADER din fisierul P.E. nou construit valoarea 0x400000.
- Cum fiecare “segment”/sectiune din program poate prevedea drepturi diferite de acces (codul este executabil, putem avea segmente read-only etc...), acestea sunt planificate sa inceapa fiecare la adresa unei noi pagini de memorie (4KiB, deci multiplu de 0x1000), fiecare pagina de memorie putand fi configurata cu drepturi specifice de catre incarcatorul de programe.

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

- În cazul unor programe de mici dimensiuni, implicatia este ca se va obține urmatoarea harta a programului în memorie (la execuțare):
  - programul este planificat să fie încărcat în memorie la adresa 0x4000000 (însă aici vor ajunge structurile de metadate ale fișierului, nu codul sau datele programului în sine)
  - primul “segment” va fi încărcat la 0x401000 (punând ghilimele deoarece nu este un segment propriu-zis ci doar o diviziune logică a programului, nu este asociat direct “segmentul” unui registru de segment – din aceasta se preferă de multe ori denumirea de secțiune în loc de segment)
  - al doilea “segment” va fi încărcat la 0x402000 (segmentul pe care îl va folosi procesorul pentru segmentare începe la adresa 0 și are limită de 4GiB, indiferent de adresele și dimensiunile secțiunilor)
  - va fi pregătit “segment” (secțiune) de importuri, “segment” de exporturi și “segment” de stack în ordinea decisă de către îmbeditor (și de dimensiuni prevăzute tot de către acesta), segmente ce vor fi încărcate de la 0x403000, 0x404000 și astăzi mai departe (incremente de 0x1000 cât timp au dimensiune suficient de mică, în caz contrar fiind nevoie să se folosească pentru increment cel mai mic multiplu de 0x1000 care permite suficient spațiu pentru continutul întregului segment)

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

- Conform logicii de decizie a adreselor de inceput ale sectiunilor, putem concluziona ca aici avem o sectiune (de date probabil) inaintea celei de cod, continand sub 0x1000 octeti, motiv pentru care codul porneste imediat dupa, de la 0x402000, harta programului fiind la final: metadate (antete) de la 0x400000, cod la 0x401000 si date la 0x402000 (urmat bineintele de alte “segmente” pentru stiva, importuri si, optional, exporturi).

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

Segment code ;starts always at offset 00402000

Start:

```
Jmp Real_start ; offset(instr. JMP) = 00402000 => + 2 octeti
a db 17 ; offset(a) = 00402002 => + 1 octet
b dw 1234h ; offset(b) = 00402003 => + 2 octeti
c dd 12345678h ; offset(c) = 00402005 => + 4 octeti
```

Real\_start:

.....

```
Mov eax, c ; EAX = 00402005
```

```
Mov edx, [c] ; mov edx, DWORD PTR DS:[00402005]
```

.....

```
Mov edx, [CS:c] ; mov edx, DWORD PTR CS:[402005]
```

```
Mov edx, [DS:c] ; mov edx, DWORD PTR DS:[402005]
```

```
Mov edx, [SS:c] ; mov edx, DWORD PTR SS:[402005]
```

```
Mov edx, [ES:c] ; mov edx, DWORD PTR ES:[402005]
```

- Efectul fiind în toate cele 5 cazuri EDX:=12345678h DE CE ?

# Elementele de bază ale limbajului de asamblare

## Directive de definire de date

- Explicația este direct legată de modelul de memorie flat – toate segmentele descriu în realitate întreaga memorie, începând de la 0 și până la capătul primilor 4GiB ai memoriei. Ca atare, [CS:c] sau [DS:c] sau [SS:c] sau [ES:c] vor accesa aceeași locație de memorie însă cu drepturi de acces potențial diferite.
- Deși toți selectorii (potențiali DIFERITI – vezi OllyDbg) indică segmente IDENTICE ca adresă și dimensiune, aceștia pot avea diferențe în cum le sunt completeate alte câmpuri de control și de acces ale descriptorilor de segment indicați de către ei.
- Modelul flat ne asigura ca mecanismul de segmentare este transparent pentru noi, noi nu sesizăm diferențe între segmente și, ca atare, scapam complet de grija segmentării (însă ne interesează împărțirea logică în segmente a programului, motiv pentru care folosim secțiuni/"segmente" separate pentru cod / date).
- Acest lucru este valabil însă doar cât timp ne limităm la CS/DS/ES și SS!
- Selectorii FS și GS indică înspre segmente speciale care nu respectă întotdeauna modelul flat (rezervate interacțiunii programului cu S.O.-ul), mai precis, [FS:c] nu garantează indicarea aceleiași zone de memorie ca și [CS:c]!

# Elementele de bază ale limbajului de asamblare

## Clasificarea erorilor în Informatică

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



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Instructiuni ale limbajului de asamblare

---

# Instructiuni ale limbajului de asamblare

- Forma generală a unui program în NASM + scurt exemplu:

```
bits 32 ; solicităm asamblarea pentru un procesor X86 (pe 32 biți)
global start ; solicităm asamblorului sa confere vizibilitate globală simbolului denumit start
(exetica start va fi punctul de intrare în program)

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

import ExitProcess kernel32.dll ; precizăm care sunt bibliotecile externe care definesc cele
două simboluri: ExitProcess e parte a bibliotecii kernel32.dll (bibliotecă standard a
sistemului de operare)

import printf msrvct.dll ; printf este funcție standard C și se regăsește în biblioteca
msrvct.dll (SO)

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

segment code use32 class=CODE ; codul progr. va fi emis ca parte a unui segment numit code
start:

; apel printf("Salut din ASM")

push dword string ; transmitem param. funcției printf (adresa sirului) pe stivă (așa cere printf)
call [printf] ; printf este numele unei funcții (etichetă = adresă , trebuie indirectată cu [])
; apel ExitProcess(0), 0 reprezentând "execuție cu succes"

push dword 0
call [ExitProcess]
```

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Instrucțiuni de transfer de uz general

Instrucțiune	Descriere / Efect
MOV d, s	$\langle d \rangle \leftarrow \langle s \rangle$ (b-b, w-w, d-d)
PUSH s	$ESP = ESP - 4$ și depune $\langle s \rangle$ în stivă (s – dublucuvânt)
POP d	Extrage elementul curent din stivă și îl depune în d (d – dublucuvânt), $ESP = ESP + 4$
XCHG d, s	$\langle d \rangle \leftrightarrow \langle s \rangle$ ; s,d trebuie să fie L-values
[reg_segment]XLAT	$AL \leftarrow DS:[EBX+AL]$ sau $AL \leftarrow \text{reg\_segment}:[EBX+AL]$
CMOVcc d, s	$\langle d \rangle \leftarrow \langle s \rangle$ dacă cc (cod condiție) este adevărat
PUSHA / PUSHAD	Depune pe stivă: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
POPA / POPAD	Extrage de pe stivă: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX
PUSHF / PUSHFD	Depune EFlags pe stivă
POPF / POPFD	Extrage vârful stivei și îl depune în EFlags
SETcc d	$\langle d \rangle \leftarrow 1$ dacă cc este adevărat, altfel $\langle d \rangle \leftarrow 0$

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Dacă operandul destinație al instrucțiunii MOV este unul dintre cei 6 registrii de segment atunci sursa trebuie să fie unul dintre cei opt registri generali de 16 biți ai UE sau o variabilă de memorie. Încărcătorul de programe al sistemului de operare preinițializează în mod automat registrii de segment, iar schimbarea valorilor acestora, deși posibilă din punct de vedere al procesorului, nu aduce nici o utilitate (un program este limitat la a încărca doar valori de selectori ce indică înspre segmente preconfigurate de către sistemul de operare, fără a putea să definească segmente adiționale).
- Instrucțiunile PUSH și POP au sintaxa PUSH s și POP d
- Operanții trebuie să fie reprezentați pe dublucuvânt, deoarece stiva este organizată pe dublucuvinte. Stiva crește de la adrese mari spre adrese mici, din 4 în 4 octeți, ESP punctând întotdeauna spre dublucuvântul din vârful stivei.

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Funcționarea acestor instrucțiuni poate fi ilustrată prin intermediul unei secvențe echivalente de instrucțiuni MOV și ADD sau SUB:
  - push eax  $\Leftrightarrow$  sub esp, 4; alocăm spațiu pentru a stoca valoarea mov [esp], eax; stocăm valoarea în locația alocată
  - pop eax  $\Leftrightarrow$  mov eax, [esp]; încărcăm în eax valoarea din vârful stivei add esp, 4; eliberăm locația

# Instructiuni

## Manipularea datelor – Instructiuni de transfer al informației

- În perspectiva evaluării efectului unor instructiuni ca **PUSH ESP** sau **POP dword [ESP]** trebuie precizat și mai clar ordinea în care se efectuează (sub)operatiile componente ale instructiunilor PUSH și POP:
  - a) **Se evaluatează operandul sursă al instrucțiunii**  
(ESP pt PUSH sau respectiv elementul din vârful stivei pt POP)
  - b) **Se actualizează corespunzător ESP**  
(ESP := ESP-4 pt PUSH și respectiv ESP := ESP+4 pt POP)
  - c) **Se efectuează atribuirea implicită de efectul instrucțiunii asupra operandului destinație**  
(noul vârf al stivei pt PUSH și respectiv dword [ESP] (acesta fiind acum după scăderea ESP de la pct b) elementul de sub vârful initial al stivei) - pt POP)
- Presupunând că situația initială este ESP = 0019FF74, după PUSH ESP vom avea ESP = 0019FF70 și continutul din varful stivei va fi acum 0019FF74.
- Presupunând că situația initială este ESP = 0019FF74 și că în aceasta locație se află valoarea 7741FA29 (varful stivei), după POP dword [ESP] vom avea ESP = 0019FF78 și continutul acestei locații (continutul locației din varful stivei) va fi 7741FA29 (deci am putea spune că „vf.stivei se mută cu o pozitie mai jos”!!).

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Instrucțiunile PUSH si POP permit doar depunerea și extragerea de valori reprezentate pe cuvânt și dublucuvânt.
- Ca atare, PUSH AL nu reprezintă o instrucțiune validă (syntax error), deoarece operandul nu este permis a fi o valoare pe octet. Pe de altă parte, secvența de instrucțiuni
  - PUSH ax ; depunem ax
  - PUSH ebx ; depunem ebx
  - POP ecx ; ecx <- dublucuvântul din vârful stivei (valoarea lui ebx)
  - POP dx ; dx <- cuvântul ramas în stivă (deci valoarea lui ax)este corectă și echivalentă prin efect cu
  - MOV ecx, ebx
  - MOV dx, ax

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Adițional acestei constrângeri (inerentă tuturor procesoarelor x86), sistemul de operare impune ca **operarea stivei să fie obligatoriu făcută doar prin accese pe dublucuvânt sau multipli de dublucuvânt**, din motive de compatibilitate între programele de utilizator și nucleul și bibliotecile de sistem. Implicația acestei constrângeri este că o instrucțiune de forma PUSH operand16 sau POP operand16 (de exemplu PUSH word 10), deși este suportată de către procesor și asamblată cu succes de către asamblor, nu este recomandată, putând cauza ceea ce poartă numele de **eroare de dezalinierie e stivei: stiva este corect aliniată dacă și numai dacă valoarea din registrul ESP este în permanență divizibilă cu 4!**

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Instrucțiunea XCHG permite interschimbarea conținutului a doi operanzi de aceeași dimensiune (octet, cuvânt sau dublucuvânt), cel puțin unul dintre ei trebuind să fie regisztr. Sintaxa ei este  
`XCHG operand1, operand2`

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- Instrucțiunea XLAT "traduce" octetul din AL într-un alt octet, utilizând în acest scop o tabelă de corespondență creată de utilizator, numită tabelă de translatare. Instrucțiunea are sintaxa  
`[reg_segment] XLAT`
- tabelă\_de\_translatare este **adresa directă** a unui sir de octeți.
- Instrucțiunea XLAT pretinde la intrare adresa fară a tabelei de translatare furnizată sub unul din următoarele două moduri:
  - DS:EBX (implicit, dacă lipsește precizarea registrului segment)
  - registru\_segment:EBX, dacă registrul segment este precizat explicit
- **Efectul instrucțiunii XLAT este înlocuirea octetului din AL cu octetul din tabelă ce are numărul de ordine valoarea din AL (primul octet din tabelă are indexul 0).**

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

- De exemplu, secvența

mov ebx, Tabela

mov al, 6

ES xlat; AL = < ES:[EBX+6] >

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

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al informației

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

```
segment data use32
    .
    .
    .
    TabHexa db '0123456789ABCDEF'
    numar resb 1
    .
    .
    .
    segment code use32
    mov ebx, TabHexa
    .
    .
    .
    mov al, numar
    xlat ; AL = < DS:[EBX+AL] >
```

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

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al adreselor LEA

- LEA `reg_general`, continutul unui operand din memorie ; `reg_general <- offset(mem)`
- Instrucțiunea LEA (**L**oad **E**ffective **A**ddress) transferă deplasamentul operandului din memorie *mem* în registrul destinație. De exemplu  
`lea eax, [v]`  
încarcă în EAX offsetul variabilei v, instrucțiune echivalentă cu  
`mov eax, v`
- Instrucțiunea LEA are însă avantajul că operandul sursă poate fi o expresie de adresare (spre deosebire de instrucțiunea `mov` care nu acceptă pe post de operand sursă decât o variabilă cu adresare directă într-un astfel de caz). De exemplu, instrucțiunea  
`lea eax, [ebx+v-6]`  
având ca efect  
„`mov eax, ebx+v-6`”

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al adreselor LEA

- LEA reg\_general, continutul unui operand din memorie ; reg\_general <- offset(mem)
- Instrucțiunea LEA (**L**oad **E**ffective **A**ddress) transferă deplasamentul operandului din memorie mem în registrul destinație. De exemplu  
`lea eax, [v]`  
încarcă în EAX offsetul variabilei v, instrucțiune echivalentă cu  
`mov eax, v`
- Instrucțiunea LEA are însă avantajul că operandul sursă poate fi o expresie de adresare (spre deosebire de instrucțiunea mov care nu acceptă pe post de operand sursă decât o variabilă cu adresare directă într-un astfel de caz). De exemplu, instrucțiunea  
`lea eax, [ebx+v-6]`  
având ca efect  
`“mov eax, ebx+v-6”`  
nu are ca echivalent direct o singură instrucțiune MOV, instrucțiunea  
`mov eax, ebx+v-6`  
fiind incorectă sintactic deoarece expresia `ebx+v-6` nu este determinabilă la momentul asamblării.

# Instrucțiuni

## Manipularea datelor – Instrucțiuni de transfer al adreselor LEA

- Prin utilizarea directă a valorilor deplasamentelor ce rezultă în urma calculelor de adrese (în contrast cu folosirea memoriei indicate de către acestea), LEA se evidențiază prin versatilitate și eficiență sporite:
  - versatilă prin combinarea unei înmulțiri cu adunări de registri și/sau valori constante
  - eficiență ridicată datorată execuției întregului calcul într-o singură instrucțiune, fără a ocupa 7 circuitele ALU care rămân astfel disponibile pentru alte operații (timp în care calculul de adresă este efectuat de către circuite specializate, separate, ale BIU).
- Exemplu: înmulțirea unui număr cu 10

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

# Instrucțiuni

**Ramificări, salturi, cicluri** (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidență deosebirile dintre un transfer direct și unul indirect.

segment data

aici DD here ;echivalent cu aici := offsetul etichetei here din segmentul de cod

segment code

...

here:

...

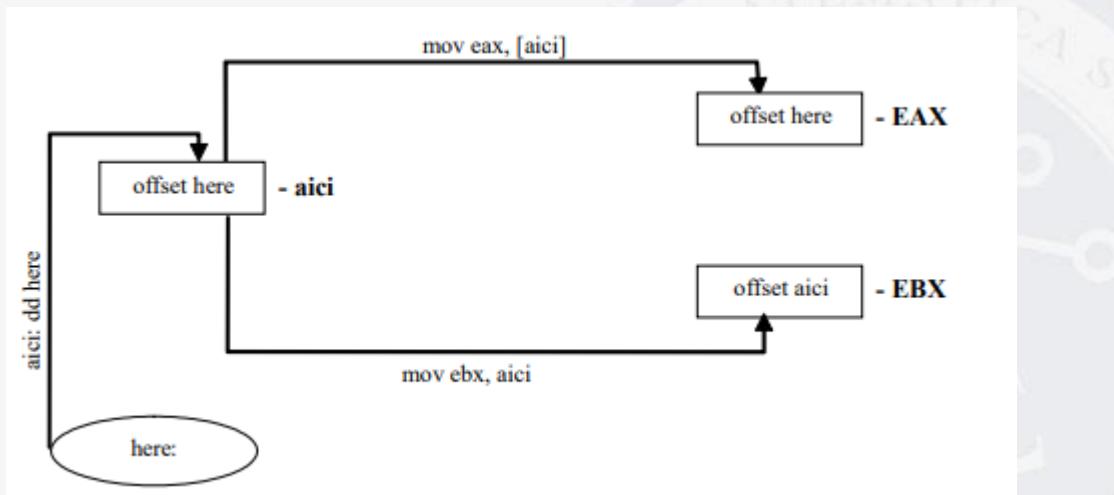
mov eax, [aici] ;se încarcă în EAX conținutul variabilei aici (adică  
deplasamentul lui here în cadrul segmentului code – echivalent  
deci ca efect cu: mov eax, here)

mov ebx, aici ;se încarcă în EBX deplasamentul lui aici în cadrul segmentului  
data (probabil 00401000h)

# Instrucțiuni

Ramificări, salturi, cicluri (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.



Initializarea variabilei `aici` și a regiștrilor `EAX` și `EBX`

# Instrucțiuni

**Ramificări, salturi, cicluri** (modul de transfer al controlului la o etichetă, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidență deosebirile dintre un transfer direct și unul indirect.

jmp [aici] ; salt la adresa desemnată de valoarea variabilei **aici** (care este adresa lui **here**),  
deci instrucțiune echivalentă cu jmp here  
; ce face jmp aici? – același lucru ca și jmp ebx ! salt la CS:EIP cu EIP=offset  
aici) din SEGMENT DATA (00401000h)  
; salt la niste instr. care merg pana la primul access violation

jmp here ; salt la adresa lui **here** (sau, echivalent, salt la eticheta **here**);  
; ce face jmp [here] ? – JMP DWORD PTR DS:[00402014] – cel mai probabil  
Access violation....

jmp eax ; salt la adresa conținută în **EAX** (adresare registru în mod direct), adică la **here**  
; ce face prin comparatie jmp [eax]? - JMP DWORD PTR DS:[EAX] – cel mai  
; probabil Access violation....

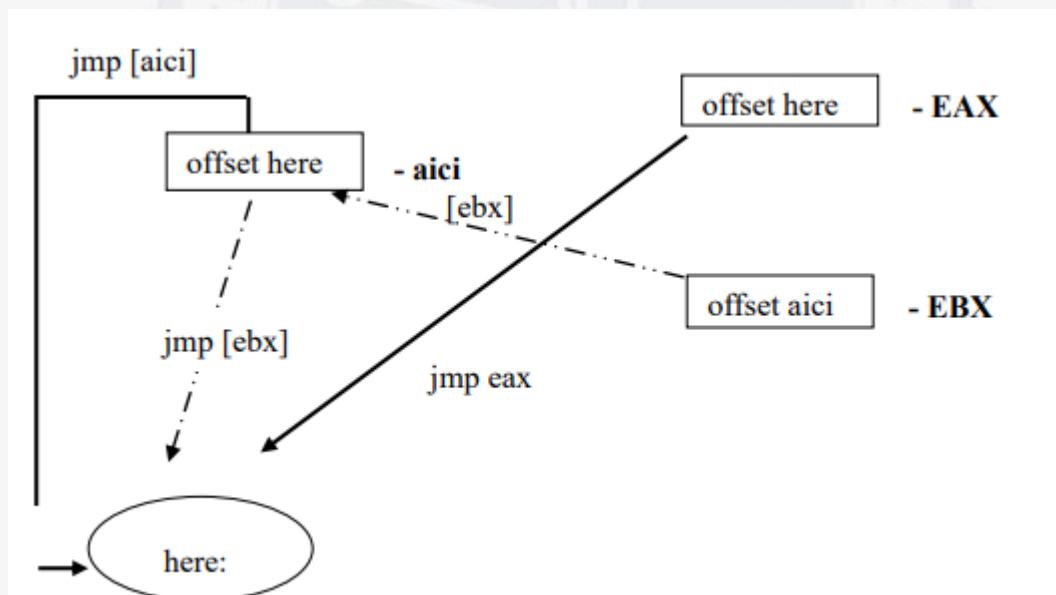
jmp [ebx] ; salt la adresa conținută în locația de memorie a cărei adresă este conținută în **EBX**  
adresare registru în mod indirect - singura situație de apel indirect din acest exemplu) – ce  
face prin comparatie jmp ebx ? - salt la CS:EIP cu EIP=offset (aici) din  
SEGMENT DATA (00401000h)  
; salt la niste instr. care merg pana la primul access violation

# Instrucțiuni

Ramificări, salturi, cicluri (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Prezentăm în continuare un exemplu edificator pentru modul de transfer al controlului la o etichetă, punând în evidență deosebirile dintre un transfer direct și unul indirect.
- În EBX se află adresa variabilei aici, deci se accesează conținutul acestei variabile
- În această locație de memorie se găsește offset-ul etichetei here, deci se va efectua saltul la adresa here - ca urmare, ultimele 4 instrucțiuni de mai sus sunt toate echivalente cu jmp here



Modalități alternative de efectuare a salturilor la eticheta here.

# Instrucțiuni

Ramificări, salturi, cicluri (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Explicații asupra interacțiunii dintre regulile de asociere implicită a unui offset cu registrul segment corespunzător și efectuarea corespunzătoare a saltului la offset-ul precizat

```
JMP [var_mem]      ; JMP DWORD PTR DS:[00401704]
                      ; salt NEAR la offset-ul din DS:[00401704]

JMP [EBX]           ; JMP DWORD PTR DS:[EBX]
                      ; salt NEAR la offset-ul din DS:[EBX]

JMP [EBP]           ; JMP DWORD PTR SS:[EBP]
                      ; salt NEAR la offset-ul din SS:[EBP]

JMP here            ; JMP [SHORT] 00402024
                      ; va face saltul DIRECT la offsetul respectiv în code segment conform
                      ; regulilor de asociere implicită a unui offset cu registrul segment coresp.
```

- Operandul cu adresare **INDIRECTĂ** de după JMP ne indică DE UNDE să luăm OFFSET-ul la care să se efectueze saltul **NEAR** (salt în interiorul segmentului de cod curent).
- Ultima instrucțiune exprimă un salt DIRECT la offset-ul calculat ca valoare asociată etichetei here (salt la CS:here).

# Instrucțiuni

**Ramificări, salturi, cicluri** (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Chiar dacă folosim prefixarea explicită cu un registru segment a operandului destinație saltul NU va fi unul FAR. FAR va fi doar ADRESA de la care se va prelua OFFSET-ul unde se va face saltul NEAR.

`jmp [ss: ebx + 12]`

`jmp [ss:ebp +12] ⇔ jmp [ebp +12]`

- Saltul rămâne în continuare unul NEAR și se va efectua în cadrul aceluiași segment de cod, adică la CS : valoare offset preluată din SS:[ebp+12]
- Dacă dorim însă efectuarea saltului într-un segment diferit (salt FAR) trebuie să specificăm explicit acest lucru prin intermediul operatorului de tip FAR, care va impune tratarea operandului destinație a instrucțiunii JMP drept O ADRESĂ FAR:

`jmp far [ebx + 12] => CS : EIP <- adresa far (48 biți = 6 octeți)`

`⇒`

`jmp far [DS: ebx + 12] => CS : EIP <- adresa far (48 biți = 6 octeți)`

`(9b 7a 52 61 c2 65) -> EIP = 61 52 7a 9b ; CS= 65 c2`

`„Mov CS:EIP, [adr_far]”`

- sau prin prefixare explicită:

`jmp far [ss: ebx + 12] => CS : EIP <- adresa far (48 biți)`

# Instrucțiuni

Ramificări, salturi, cicluri (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

## Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR.

- Operatorul FAR precizeaza aici faptul că nu doar EIP trebuie populat cu ce se află la adresa de memorie indicată de operandul destinație (adresă NEAR = offset) ci și CS-ul trebuie încărcat cu o nouă valoare (CS:EIP = adresă FAR).
- Pe scurt avem:
  - valoarea pointer-ului poate fi stocată oriunde în memorie, rezultând că orice specificare de adresă validă la un mov poate apărea la fel de bine și la jmp (de exemplu jmp [gs:ebx + esi\*8 - 1023])
  - pointer-ul în sine (deci octetii luati de la respectiva adresă de memorie) poate fi FAR sau NEAR, fiind, după caz, aplicat fie lui EIP (dacă e NEAR), fie perechii CS:EIP dacă saltul e FAR.
- Saltul poate fi imaginat ca fiind echivalent, ipotetic, cu instrucțiuni MOV de forma:

jmp [gs:ebx + esi \* 8 - 1023]  $\Leftrightarrow$  mov EIP, [gs:ebx + esi \* 8 - 1023]

jmp FAR [gs:ebx + esi \* 8 - 1023]  $\Leftrightarrow$  mov EIP, DWORD [gs:ebx + esi \* 8 - 1023] +  
mov CS, WORD [gs:ebx + esi \* 8 - 1023 + 4]

- La adresa [gs:ebx + esi \* 8 - 1023] am gasit în exemplul concret utilizat configurația de memorie: 7B 8C A4 56 D4 47 98 B7.....

Mov CS:EIP, [memory]  $\rightarrow$  EIP = 56 A4 8C 7B, CS = 47 D4

# Instrucțiuni

**Ramificări, salturi, cicluri (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)**

**Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR. JMP prin etichete – întotdeauna NEAR!**

```
segment data use32 class=DATA
a db 1,2,3,4
start3:
    mov edx, eax      ; ok ! – controlul este transferat la start3 și această instrucțiune este executată !

segment code use32 class=code ; offset code segment = 00402000
start:
    mov edx, eax
    jmp start2        ; ok – salt NEAR - JMP 00403000 (offset code1 segment = 00403000)
    jmp start3        ; ok – salt NEAR – JMP 00401004 (offset data segment = 00401000)
    jmp far start2   ; Segment selector relocations are not supported in PE file – syntax error !
    jmp far start3   ; Segment selector relocations are not supported in PE file– syntax error !
                    ; Cele două salturi de mai sus jmp start2 si respectiv jmp start3 se vor efectua la etichetele
                    ; menționate, start2 si start3 însă ele nu vor fi considerate salturi FAR (dovada este că precizarea
                    ; acestui atribut mai sus în celelalte două variante ale instrucțiunilor va furniza syntax error !). Ele
                    ; vor fi considerate salturi NEAR datorita modelului de memorie FLAT utilizat de catre SO
    add eax,1

final:
    push dword 0
    call [exit]

segment code1 use32 class=code
start2:
    mov eax, ebx
    push dword 0
    call [exit]
```

**De ce ?... Din cauza “Flat memory model”**

# Instrucțiuni

**Ramificări, salturi, cicluri** (modul de transfer al controlului la o eticheta, pag. 142-143 – carte curs)

**Analiza instrucțiunii JMP. Salturi NEAR și salturi FAR. JMP prin etichete – întotdeauna NEAR!**

**Concluzii finale.**

- Salturi NEAR – se pot realiza prin oricare dintre cele 3 tipuri de operanzi (etichetă, regisztr, operand cu adresare la memorie)
- Salturi FAR (asta însemnând modificarea și a valorii din CS, nu numai a valorii din EIP) – se pot realiza DOAR prin intermediul unui operand adresare la memorie pe 48 de biți (pointer FAR = 6 octeți). De ce doar aşa și prin etichete sau registri nu ?
- Prin etichete, chiar dacă se sare într-un alt segment nu se consideră ca este salt FAR deoarece nu se modifica CS-ul (din cauza modelului de memorie implementat – Flat Memory Model). Se va modifica doar EIP-ul și saltul se consideră dpdv tehnic ca fiind un salt NEAR.
- Prin registri nu este posibil deoarece registri sunt pe 32 de biți și se poate astfel specifica drept operand al unui JMP doar un offset (salt NEAR), deci practic suntem în imposibilitatea de a preciza un salt FAR cu un operand limitat la 32 de biți

# Instrucțiuni

## Ramificări, salturi, cicluri

### Instrucțiuni CALL și RET

- Apelul unei proceduri se face cu ajutorul instrucțiunii CALL, acesta putând fi apel direct sau apel indirect.
- Apelul direct are sintaxa  
`CALL operand`  
Asemănător instrucțiunii JMP și instrucțiunea CALL transferă controlul la adresa desemnată de operand.
- În plus față de aceasta, înainte de a face saltul, instrucțiunea CALL salvează în stivă adresa următoarei instrucțiuni de după CALL (adresa de revenire).
- Cu alte cuvinte, avem echivalența

<code>CALL operand</code>	$\Leftrightarrow$	<code>push A</code>
<code>A: . . .</code>		<code>jmp operand</code>

# Instrucțiuni

## Ramificări, salturi, cicluri

### Instrucțiuni CALL și RET

- Terminarea execuției secvenței apelate este marcată de întâlnirea unei instrucțiuni RET.
- Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucțiunea de la această adresă.
- Sintaxa instrucțiunii RET este

RET [n]

unde n este un parametru optional. El indică eliberarea din stivă a n octeți aflați sub adresa de revenire.

- Instrucțiunea RET poate fi ilustrată prin echivalență

B dd ?

. . .

pop [B]

add esp, [n]

jmp [B]

RET n

(revenire near)  $\Leftrightarrow$

# Instrucțiuni

## Ramificări, salturi, cicluri

### Instrucțiuni CALL și RET

- De cele mai multe ori, după cum este și natural, instrucțiunile CALL și RET apar în următorul context

etichetă\_procedură :

...

ret n

...

CALL etichetă\_procedură

- Instrucțiunea CALL poate de asemenea prelua adresa de transfer dintr-un registru sau dintr-o variabilă de memorie.
- Un asemenea gen de apel este denumit **apel indirect**.

Exemple:

call ebx ;adresă preluată din registru

call [vptr] ;adresă preluată din memorie (similar cu apelul funcției printf)

- Rezumând, operandul destinație al unei instrucțiuni CALL poate fi:

- numele unei proceduri
- numele unui registru în care se află o adresă
- o adresă de memorie



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**

# Arhitectura Sistemelor de Calcul

Lect. Dr. Șotropa Diana  
[diana.sotropa@ubbcluj.ro](mailto:diana.sotropa@ubbcluj.ro)



---

Facultatea de Matematică și Informatică  
Universitatea Babeș-Bolyai





# Programarea multimodul

---

Material creat de către Lect. Dr. Vancea Alexandru împreună cu Marius Vanță  
Bitdefender 2017

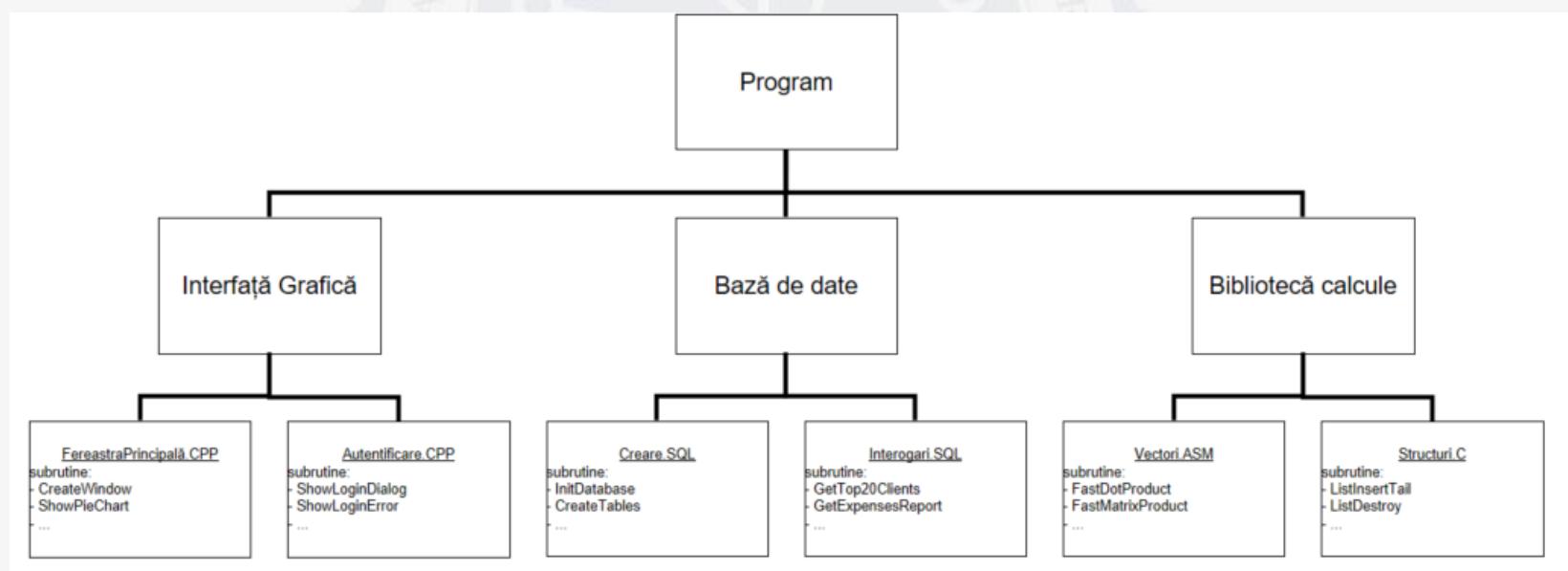
# Arhitecturi modulare

## Programare modulară

- Cum împărțim problema în sub-probleme?

- Modularizare

- Program -> unități logice
- Cod (al unităților) -> fișiere distincte
- Fișiere -> subroutines



# Arhitecturi modulare

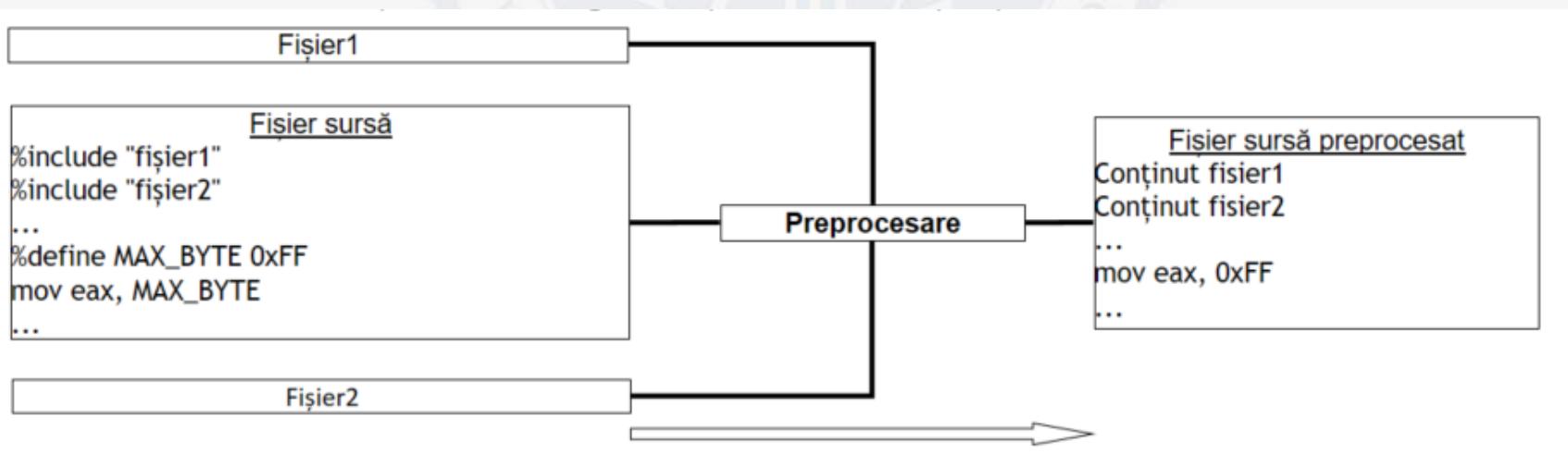
## Programare modulară

- Pentru care sub-probleme există deja rezolvări disponibile?
  - Reutilizare:
    - Fișiere sursă
      - Refolosire cod și date din asamblare
        - Directiva `%include`  
**(NU este programare multi modul, deoarece la compilare ajunge DOAR UN SINGUR MODUL obținut prin concatenarea textuală a fișierelor incluse)**
      - Fișiere binare
        - Refolosire cod și date din asamblare
        - Refolosire cod și date din limbaje de nivel înalt
        - Biblioteci
    - Existența de fișiere binare separate implică ASAMBLARE / COMPILEARE SEPARATĂ

# Tehnici și instrumente

## Includerea statică la compilare / asamblare: directiva %include

- Specifică limbajului (dar are echivalent și în alte limbi)
- Modularizare: permite doar divizarea codului scris în acel limbaj
  - NU este programare multimodul, deoarece aceasta necesită COMPILEARE SEPARATĂ
- Reutilizare: expune codul sursă
- Pericolos și problematic:
  - Mecanism de preprocesor -> concatenare textuală a fișierelor
  - Expune cu vizibilitate globală toate denumirile -> conflicte (redefiniții / redeclarări)
  - Include fișierul în întregime - și ce se folosește, și ce nu

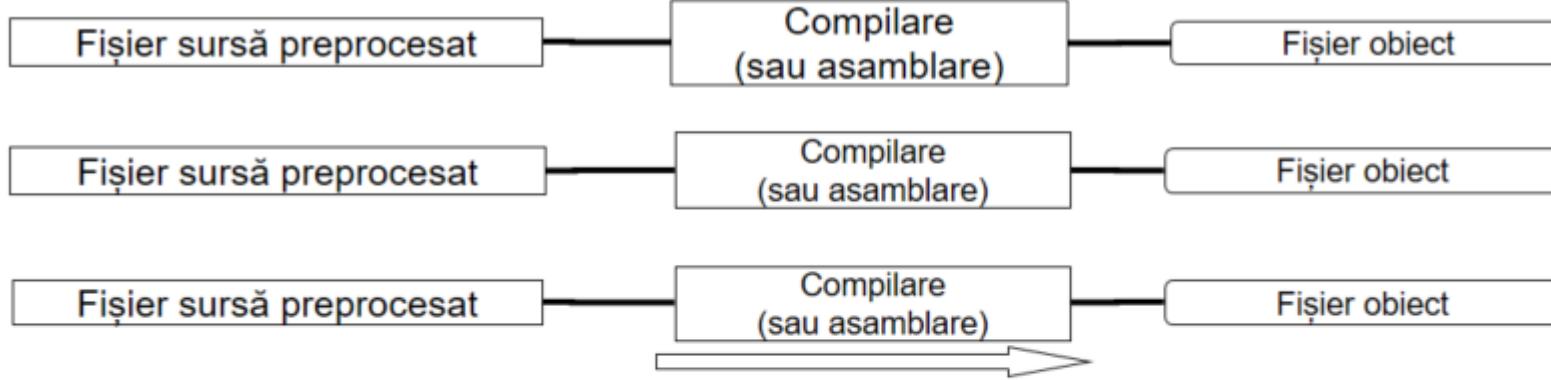


# Tehnici și instrumente

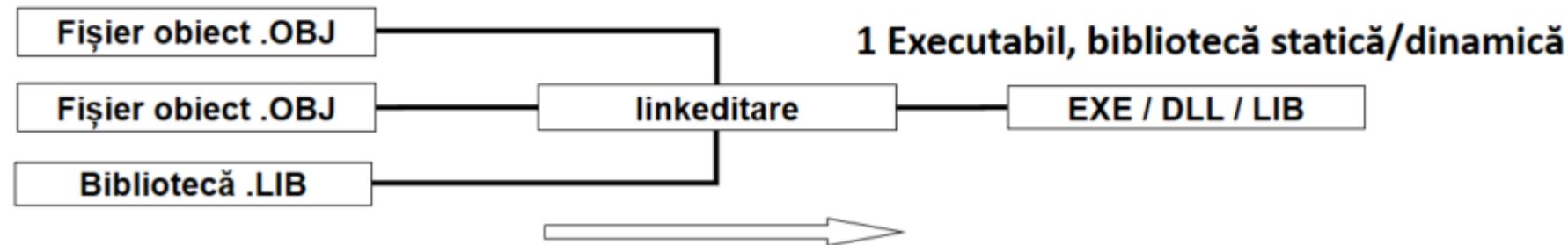
## Legarea statică la linkeditare:

- Pas realizat de către un linkeditor după asamblare / compilare

**N unități de compilare, compilate SEPARAT !!!**



**N module**



# Tehnici și instrumente

## Legarea statică la linkeditare – sumar responsabilități

- **Preprocesor: text => text**

- Efectuează prelucrări asupra textului sursă, rezultând un text sursă intermediar
- Se poate imagina ca fiind o componentă a compilatorului sau asamblorului
- Poate lipsi, multe limbaje nu au un preprocesor

- **Asamblor: instrucțiuni (text) => codificare binară (fișier obiect)**

- Codifică instrucțiunile și datele (variabilele) din textul sursă preprocesat și construiește un fișier obiect ce conține cod mașină și valori de variabile alături de informații despre conținut (denumiri de variabile, subroutines, informații despre tipul și vizibilitatea acestora, etc.)

# Tehnici și instrumente

## Legarea statică la linkeditare – sumar responsabilități

- **Compilator: instrucțiuni (text) => codificare binară (fișier obiect)**

- Identifică secvențe de instrucțiuni de procesor prin care se pot obține funcționalitățile descrise în textul sursă, iar apoi, precum un asamblor, generează un fișier obiect ce conține codificarea binară a acestora și a variabilelor din program
- Asamblarea este un caz special de compilare, unde instrucțiunile de procesor sunt gata oferite direct în textul programului și ca atare nu necesită să fie alese de către compilator

- **Linkeditor: fișiere obiect => bibliotecă sau program**

- Construiește rezultatul final, adică un program (.exe) sau bibliotecă (.dll sau .lib) în care leagă împreună (include) codul și datele binare prezente în fișierele obiect
- Nu este interesat în ce compilatoare sau ce limbaje au fost folosite! Legarea necesită doar ca fișierele de intrare să respecte formatul standard al fișierelor obiect!

# Tehnici și instrumente

## Exemplu folosire %include

; fișierul constante.inc

; gardă dublă-includere

```
%ifndef _CONSTANTE_INC_ ; la prima includere _CONSTANTE_INC nu este definit
#define _CONSTANTE_INC_ ; definim _CONSTANTE_INC_ -> condiție falsă la
viitoare includeri
```

; recomandat ca astfel de fișiere (incluse de către altele) să conțină (doar) declarații

```
MAX_BYTE equ 0xFF
MAX_WORD equ 0xFFFF
MAX_DWORD equ 0xFFFFFFFF
MAX_QWORD equ 0xFFFFFFFFFFFFFF
%endif ; _CONSTANTE_INC_
```

# Tehnici și instrumente

## Exemplu folosire %include

- impachetare EAX într-un BYTE / WORD / DWORD, conform magnitudinii valorii acestuia

;fișierul program.asm

```
%include "constante.inc"
```

```
CMP EAX, MAX_BYTE
```

```
JA .nu_incape_in_octet
```

```
.incape_in_octet:
```

```
    MOV [rezultat_octet], AL
```

```
    JMP .gata
```

```
.nu_incape_in_octet:
```

```
    CMP EAX, MAX_WORD
```

```
    JA .nu_incape_in_cuvant
```

```
.incape_in_cuvant:
```

```
    MOV [rezultat_word], AX
```

```
    JMP .gata
```

```
.nu_incape_in_cuvant:
```

```
    MOV [rezultat_dword], EAX
```

```
.gata:
```

; incape valoarea din EAX într-un BYTE?

; dacă da, salvăm AL în rezultat octet

; altfel verificăm dacă ajunge un WORD

; dacă da, salvăm AX în rezultat\_word

; dacă nu ajunge un WORD, salvăm întreg EAX

# Tehnici și instrumente

## Legarea statică la linkeditare – cerințele nasm

- Resursele sunt partajate de comun acord
- Export prin **global nume1, nume2, ...**
  - Ofer disponibilitate oricărui fișier ar fi interesat
- Import prin **extern nume1, nume2, ...**
  - Solicit acces, indiferent din ce fișier va fi oferită resursa
- Solicitare fără disponibilitate = eroare!
  - Nu se pot importa decât resurse ce sunt exportate undeva
- Însă disponibilitate fără solicitare este caz permis. De ce?
  - Răspuns: chiar dacă nici un modul din program nu solicită / folosește, poate se va utiliza într-o versiune viitoare sau de către alt program
- Limbajele de programare de nivel mai înalt oferă și ele la rândul lor construcții sintactice cu rol echivalent!
  - Exemplu: În limbajul C
    - Disponibilitatea este automată / implicită, putându-se însă opta pentru a bloca accesul prin folosirea cuvântului cheie **static**
    - Solicitarea de acces se face (tot) prin intermediul cuvântului cheie **extern**

# Tehnici și instrumente

## Legarea statică la linkeditare

- Permite unirea mai multor **module binare** (fișiere obiect sau biblioteci statice) într-un singur fișier
  - Intrări: oricâte fișiere obiect (.obj) și/sau biblioteci statice (.lib)
    - Atenție! Nu toate fișierele .LIB sunt biblioteci statice!
    - Ieșire: .EXE sau .LIB sau .DLL (Dynamic-Link Library)
- Multimodul: oricâte fișiere pot fi asamblate / compilate separat și linkeditate împreună
  - Pas realizat de linkeditor după compilare / asamblare -> **nu depinde de limbaj!**
- Reutilizare:
  - În formă binară – nu expune codul sursă!
  - Permite inter-operabilitate între limbaje diferite!
- Alte avantaje și dezavantaje:
  - Editorul de legături poate identifica și elibera resurse neutilizate sau efectua alte optimizări
  - Dimensiune mare a programului: programul înglobează resursele externe reutilizate
  - Dimensiune mare a programelor: bibliotecile populare duplicate în multe programe
- NASM: directivele **global (mecanism export)** și **extern (mecanism import)**
  - Global nume – oferirea posibilității de utilizare din exterior a acestei resurse date prin nume
  - Extern nume – solicitare de acces la resursa specificată; necessită să fie publică

# Tehnici și instrumente

- Modularizarea codului în asamblare
  - Procedură – secvență de instrucțiuni care să poată fi apelată din alte zone ale programului și care după terminarea ei returnează controlul programului apelant
  - Este necesar ca la apelul unei proceduri să se salveze **adresa de revenire** în **stiva de execuție**
  - Revenirea din procedură este de fapt o instrucțiune de salt la adresa de revenire

**call eticheta**

**ret [n]**

# Tehnici și instrumente

## Legarea statică la linkeditare - cerințele NASM

- Folosirea în practică a directivelor global și extern

; FIŞIER1.ASM

**global** Var1, Subrutina2

**extern** Var3, Subrutina3

Subrutina1:

...  
Apel (Subrutina3)

...  
Operatii (Var3)

Subrutina2:

...  
Var1 dd ...  
Var2 db ...

; FIŞIER2.ASM

**extern** Var1, Subrutina2

**global** Var3, Subrutina3

Subrutina3:

...  
Apel (Subrutina2)

...  
Operatii (Var1)

Subrutina1:

...  
Var2 dd ...  
Var3 db ...

# Tehnici și instrumente

- Exemplu program multimodul nasm + nasm
  - Pașii necesari construirii programului executabil final
    - Se asamblează fișierul main.asm  
`nasm.exe -fobj main.asm`
    - Se asamblează fișierul sub.asm  
`nasm.exe -fobj sub.asm`
    - Se editează legăturile dintre cele două module  
`alink.exe main.obj sub.obj -oPE -entry start -subsys console`
  - Observație: cele două module pot fi asamblate în orice ordine! Abia în timpul linkeditării este necesar ca simbolurile referite să aibă toate implementare disponibilă în unul dintre fișierele obiect oferite linkeditorului
  - Linkeditarea, în mod evident, este posibilă doar după asamblare / compilare

# Tehnici și instrumente

## Legarea statică la linkeditare - cerințele NASM

- Folosirea în practică a directivelor global și extern

; FIŞIER1.ASM

**global** Var1, Subrutina2

**extern** Var3, Subrutina3

Subrutina1:

...  
Apel (Subrutina3)

...  
Operatii (Var3)

...

Subrutina2:

...

Var1 dd ...

Var2 db ...

; FIŞIER2.ASM

**extern** Var1, Subrutina2

**global** Var3, Subrutina3

Subrutina3:

...  
Apel (Subrutina2)

...  
Operatii (Var1)

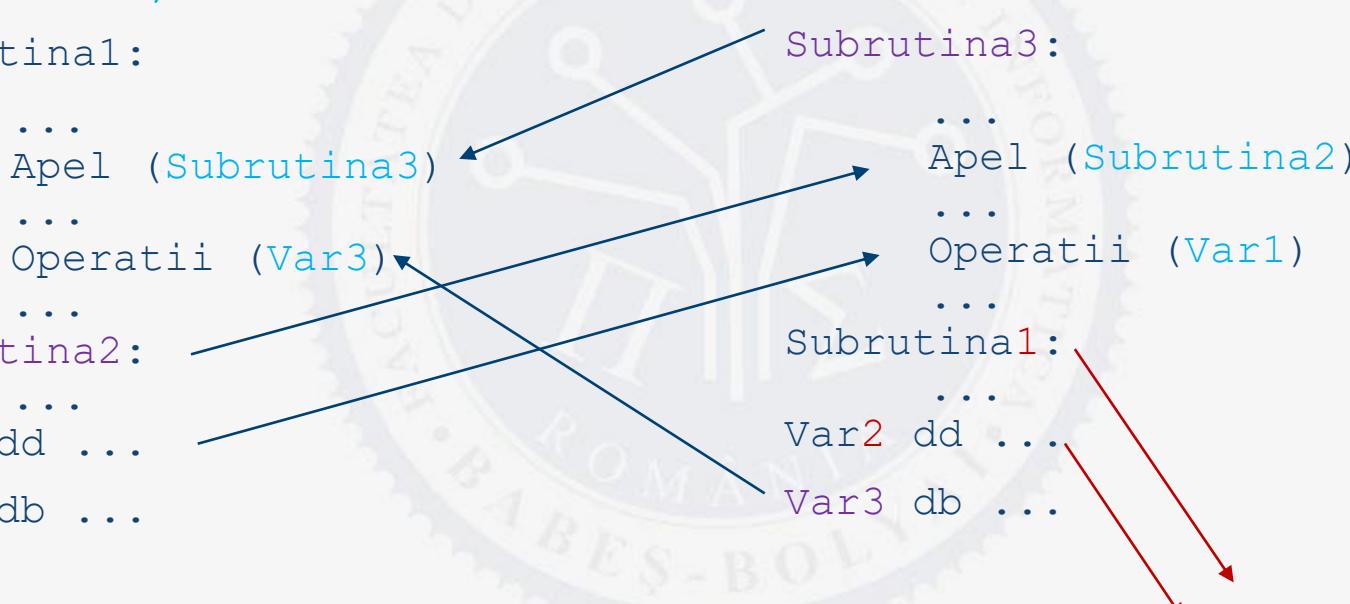
...

Subrutina1:

...

Var2 dd ...

Var3 db ...



*Pot fi refolosite denumirile,  
cât timp nu sunt globale!*

# Tehnici și instrumente

## Legarea statică la linkeditare - cerințele NASM

- MODALITĂȚI DE TRANSMITERE A ARGUMENTELOR ȘI DE OBȚINERE A REZULTATULUI
  - Argumente: registru, Rezultat: registru
  - Argumente: stivă, Rezultat: registru
  - Argumente: stivă, Rezultat: stivă

# Tehnici și instrumente

## Exemplu program multimodul nasm + nasm

```
; MODULUL MAIN.ASM
global SirFinal
extern Concatenare

import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit
global start
```

```
segment code use32 public code class='code'
start:
```

```
    mov eax, Sir1
    mov ebx, Sir2
    call Concatenare
    push dword SirFinal
    call [printf]
    add esp, 1*4
    push dword 0
    call [exit]
```

```
segment data use32
```

```
Sir1 db 'Buna ',0
Sir2 db 'dimineata!',0
SirFinal resb 1000 ; spațiu pentru rezultat
```

```
; MODULUL SUB.ASM
extern SirFinal
global Concatenare

segment code use32 public code class='code'
; eax = adresa primului sir
; ebx = adresa sirului secund

Concatenare:
    mov edi, SirFinal ; destinație = sirFinal
    mov esi, eax ; sursa = primul sir

.sir1Loop:
    lodsb ; luăm octetul următor
    test al,al ; este terminatorul de sir (=0)?
    jz .sir2 ; dacă da trecem la sirul al doilea
    stosb ; altfel copiem în destinație
    jmp .sir1Loop ; și continuăm până la nul

.sir2:
    mov esi, ebx ; sursa = sirul al doilea

.sir2Loop:
    lodsb ; același proces pentru noul sir
    test al,al
    jz.gata
    stosb
    jmp .sir2Loop

.gata:
    stosb ; adăugăm terminatorul de sir din al
    ret
```



FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
UNIVERSITATEA BABEŞ-BOLYAI

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

**[www.cs.ubbcluj.ro](http://www.cs.ubbcluj.ro)**