

---

# Instrucțiuni x86

De citit: Dandamudi  
Capitole 4.5, 7, 8, 9, 10

Modificat: 15-Oct-18

# Instrucțiuni aritmetice

---

## INC și DEC

- \* **Format:**

- » **inc destination**

- » **dec destination**

- \* **Semantică:**

- » **destination = destination +/- 1**

- » **destination** poate fi pe 8, 16, sau 32-biți memorie/registru

- » Nu putem folosi operanzi imediați

- » Nu seteaza CF

- **Exemple**

- inc BX ; BX = BX+1**

- dec **word** [value] ; value = value-1 (16 biți)**

# Instrucțiuni aritmetice

---

## ADD/SUB

- \* **Format:**

- » **add destination, source**
- » **sub destination, source**

- \* **Semantică:**

- » **destination = (destination)+/- (source)**

- **Exemple**

```
add    EBX, EAX
add    byte [value], 10H
sub    EBX, EAX
```

- **Observatii:**

- \* **inc EAX** e mai bun decât **add EAX, 1**
- \* **dec EAX** e mai bun decât **sub EAX, 1**
- \* generează mai puțin cod

# Setare indicatori de conditie

---

## CMP

- \* Format:

`cmp destination, source`

- \* Semantică:

$(\text{destination}) - (\text{source})$

- \* **destination și source NU sunt alterate**

- \* Testează relația de ordine între operanzi

- \* Se efectuează scăderea, se setează EFLAGS

- \* Nu se reține rezultatul scăderii

- \* Se folosește cu salturi condiționale

- **Exemple**

`cmp EBX, EAX`

`cmp dword [count], 10`

# Setare indicatori de conditie

---

- Unii indicatori pot fi setati in mod explicit prin operatii specifice:
  - \* **Carry**
    - » STC (Set Carry Flag)
    - » CLC (Clear Carry Flag)
    - » CMP (Complement Carry Flag)
  - \* **Interrupt**
    - » STI (Set Interrupt Flag)
    - » CLI (Clear Interrupt Flag)
  - \* **Direction**
    - » STD (Set Direction Flag)
    - » CLD (Clear Direction Flag)

# Instrucțiuni de salt (jump)

---

## Salt necondiționat

- \* **Format:**

**jmp label**

- \* **Semantică:**

- » Execuția transferată la instrucțiunea identificată de **label**
- » Se retine deplasamentul relativ fata de intructiunea curenta

- **Exemplu:** ciclu infinit

```
    mov     EAX, 1
    [ inc_again:
      inc     EAX
      jmp    inc_again
      mov     EBX, EAX      ; nu se ajunge aici
```

# Instrucțiuni de salt (jump)

---

## Salt condiționat

- \* **Format:**

**j<cond> label**

- \* **Semantică:**

- » Execuția transferată la instrucțiunea referită de **label** doar dacă condiția **<cond>** este îndeplinită

- » **Depinde de o combinație de flag-uri din EFLAGS**

- **Exemplu: Verificare \r (Carriage Return)**

```
GetCh    AL
cmp      AL, 0DH    ; 0DH = ASCII carriage return
[ je      CR_received
  inc     CL
  CR_received:
```

# Instrucțiuni de salt (jump)

---

## Salt condiționat

- \* Acestea depind de valoarea **unui flag** din registrul EFLAGS

|            |                                    |
|------------|------------------------------------|
| <b>jz</b>  | <b>jump if zero</b> (ZF == 1)      |
| <b>jnz</b> | <b>jump if not zero</b> (ZF == 0)  |
| <b>jc</b>  | <b>jump if carry</b> (CF == 1)     |
| <b>jnc</b> | <b>jump if not carry</b> (CF == 0) |
| <b>jo</b>  | <b>jump if overflow</b> (OF == 1)  |
| <b>js</b>  | <b>jump if sign</b> (SF == 1)      |
| <b>jp</b>  | <b>jump if parity</b> (PF == 1)    |

- \* **jz** este echivalent cu **je**
- \* Analog **jnz** și **jne**



# Instrucțiuni de salt (jump)

---

Instrucțiuni ce tratează operanzii ca nr. **fara semn.**

|                  |  |
|------------------|--|
| <code>je</code>  | <code>if equal (ZF==1)</code>                    |
| <code>jne</code> | <code>if not equal (ZF==0)</code>                |
| <code>ja</code>  | <code>if greater (CF==0 &amp;&amp; ZF==0)</code> |
| <code>jae</code> | <code>if greater or equal (CF==0)</code>         |
| <code>jb</code>  | <code>if less (CF==1)</code>                     |
| <code>jbe</code> | <code>if less or equal (CF==1    ZF==1)</code>   |

Instrucțiuni ce tratează operanzii ca nr. **cu semn.**

|                  |   |
|------------------|---|
| <code>je</code>  | <code>if equal (ZF==1)</code>                     |
| <code>jne</code> | <code>if not equal (ZF==0)</code>                 |
| <code>jg</code>  | <code>if greater (ZF==0 &amp;&amp; SF==OF)</code> |
| <code>jl</code>  | <code>if less (SF != OF)</code>                   |
| <code>jge</code> | <code>if greater or equal (SF==OF)</code>         |
| <code>jle</code> | <code>if less or equal (ZF==1    SF!=OF)</code>   |

# Instrucțiuni de salt (exemple)

---

## \* Exemplu 1

mov ah, 1

cmp ah, -1

ja fara ; no jump  $1 < 255$

jg cu ; **jump**  $1 > -1$

## \* Exemplu 2

mov ah, -1

add ah, 10

jo over ; no jump  $(-1+10)=9$

js semn ; no jump  $9 > 0$

jc carry; **jump**  $255+10 > 255$

# Salt indirect

---

- Se face salt la o adresă care nu este specificată direct în corpul instrucțiunii
- Se poate specifica un target printr-o adresă de memorie sau un registru
- Exemplu:
  - » Presupunând că ECX conține adresa targetului

**jmp [ECX]**

  - \* Notă: În cazul acesta, ECX trebuie să conțină adresa în valoare absolută, nu relativă ca în cazul saltului direct.

# Apel de procedură direct/indirect

---

- \* **Format:**

**CALL label**

- \* **Semantică:**

- » Execuția transferată la instrucțiunea identificată de **label**
- » **Înainte de apel se salvează adresa de retur pe stivă**
- » Revenirea se face la apelul instrucțiunii RET care extrage adresa de retur de pe stivă și execută salt la acea locație

- Exemplu: EBX conține un pointer la o procedură

**CALL      *Calculeaza***

...                    ;aici se revine după call

***Calculeaza:***

...                    ;aici este corpul rutinei

**RET**                ;aici se termină rutina și se  
                      ;revine în programul principal

# Apel de procedură direct/indirect

---

## Call

- apel direct: adresa imediată a procedurii apelate
- apel indirect: adresa procedurii apelate în registru sau memorie

- Exemplu ASM:

dword la **target\_proc\_ptr** conține un pointer  
**call [target\_proc\_ptr]**

- Exemplu C:

```
int inc(int x){ return x+1;}  
int (*fa)(int);  
fa = &inc;  
printf("%d\n", (*fa)(2));
```

# Instrucțiuni de ciclare

---

## LOOP

- \* **Format:**

**loop target**

- \* **Semantică:**

- » Decrementează ECX și face salt la **target** dacă  $ECX \neq 0$
- » ECX trebuie inițializat cu numărul de repetări

- Exemplu (execută <loop body> de 50 de ori):

```
mov    ECX, 50
repeat:
    <loop body>
    loop repeat ...
```

- Exemplul anterior este echivalent cu:

```
mov    ECX, 50
repeat:
    <loop body>
    dec  ECX
    jnz  repeat ...
```

# Instrucțiuni de ciclare

---

- Următoarele două verifică, în plus, și indicatorul ZF

**loope/loopz      target**

Action:  $ECX = ECX - 1$

Jump to target if ( $ECX \neq 0$  and  $ZF = 1$ )

**loopne/loopnz      target**

Action:  $ECX = ECX - 1$

Jump to target if ( $ECX \neq 0$  and  $ZF = 0$ )

# Instrucțiuni logice

---

## AND/OR/XOR (Toate instr. logice actualizează EFLAGS)

**and**            **destination, source**

- \* Prin măști de biți, setează anumiți biți să fie 0

**or**            **destination, source**

- \* Prin măști de biți, setează anumiți biți să fie 1
- \* Copiază anumiți biți dintr-un byte, cuvânt, dublu-cuvânt

**xor**            **destination, source**

- \* Comutare de biți
- \* Inițializare registre la 0, de exemplu:

**xor**            **AX, AX**

**not**            **destination**

- \* Complementul față de 2 al unui număr pe 8 biți, de exemplu:

**not**            **AL**

**inc**            **AL**



# Instrucțiuni logice

---

## TEST

**test      destination, source**

\* Face AND non-destructiv

- » Nu se reține rezultatul, nu se modifică **destination**
- » Similar cu **cmp**

Exemplu: Verificarea parității unui număr reținut în AL

```
test  AL, 01H  ; test the least significant bit
je    even_number
odd_number:
    <process odd number>
    jmp  skip1
even_number:
    <process even number>
skip1:
```

# Instrucțiuni de shiftare

---

- Două tipuri de shiftări

- \* Logică

- » **shl (SHift Left)**

- » **shr (SHift Right)**

- » Altă interpretare:

- Se aplică pe numere fără semn

- \* Aritmetică

- » **sar (Shift Arithmetic Left)**

- » **sar (Shift Arithmetic Right)**

- » Altă interpretare:

- Se aplică pe numere cu semn

# Shiftare logică

---

## Shift left

**shl**      **destination, count**

**shl**      **destination, CL**

## Shift right

**shr**      **destination, count**

**shr**      **destination, CL**

- \* Semantică: Shiftare stânga/dreapta a **destination** cu valoarea din **count** sau registrul **CL** (nu modifica CL). **destination** poate fi pe 8, 16 sau 32 de biți; se poate afla în memorie sau într-un registru

- \* Exemplu: prelucrare pe biți

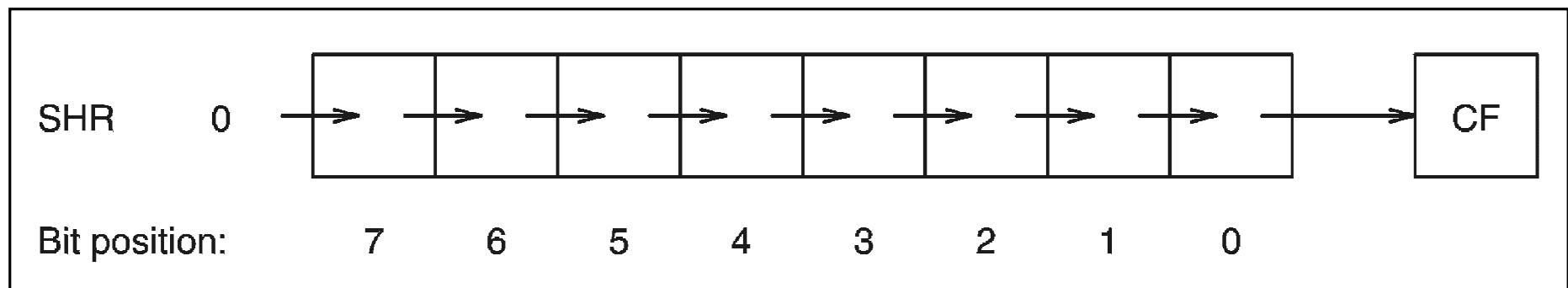
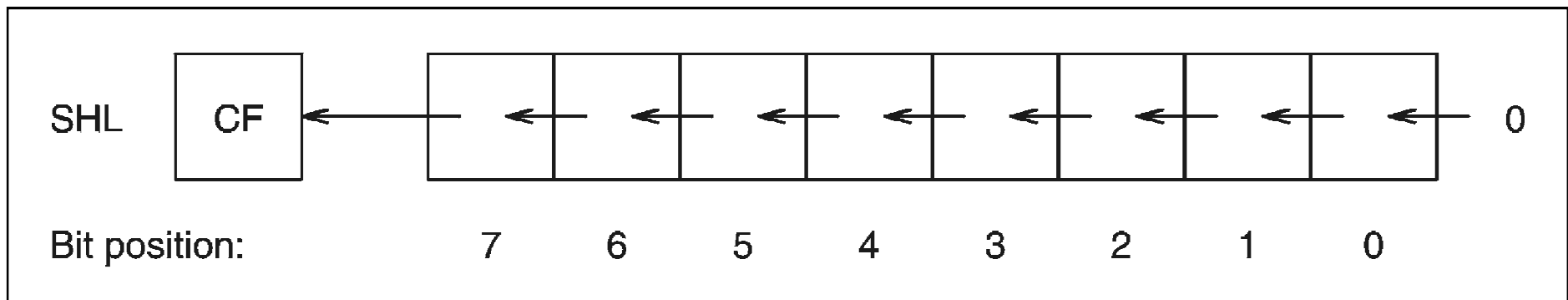
```
    ; AL contains the byte to be encrypted
mov    AH,AL
shl     AL,4      ; move lower nibble to upper
shr     AH,4      ; move upper nibble to lower
or      AL,AH     ; paste them together
    ; AL has the encrypted byte
```

- \* Înmulțire și împărțire

- » Puteri ale lui 2
- » Mai eficient decât mul/div

# Shiftare logică

- Ultimul bitul care iese în afară ajunge în CF
  - » biți de 0 sunt inserați la celălalt capăt



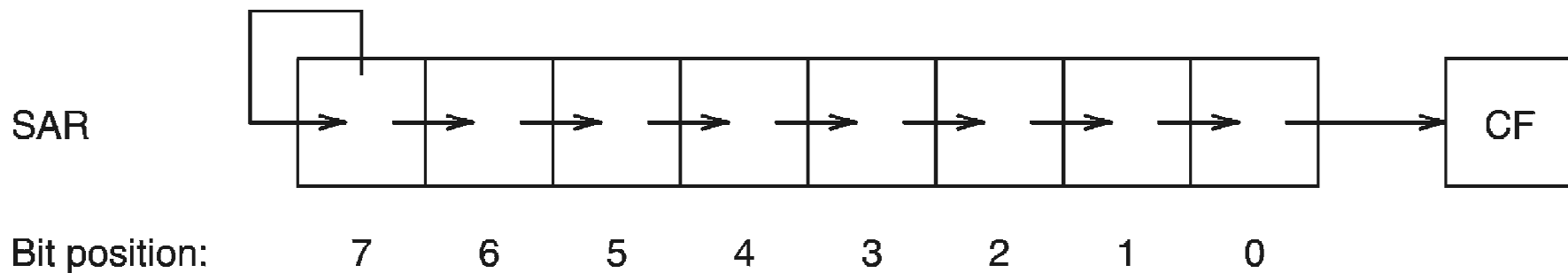
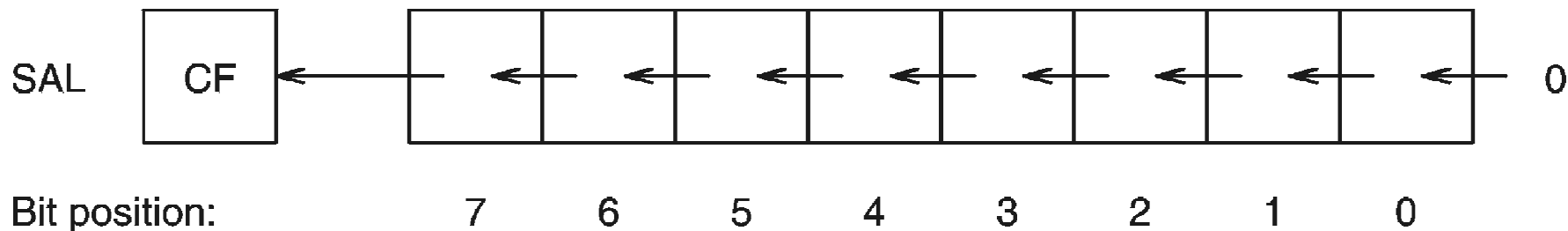
# Shiftare logică

---

- Efect asupra indicatorilor:
  - \* Auxiliary flag (AF): undefined
  - \* Zero flag (ZF) și parity flag (PF) sunt actualizate
  - \* Carry flag
    - » Conține ultimul bit shiftat în afară
  - \* Overflow flag
    - » Pentru shiftări pe mai mulți biți
      - Undefined
    - » Pentru shiftări cu un singur bit
      - Se setează dacă în urma shiftării se schimbă semnul
      - Altfel devine 0

# Shiftare aritmetică

- Două variante, ca la cea logică:  
**sal/sar destination, count**  
**sal/sar destination, CL**



# Instrucțiuni de rotire

---

- Problemă la shiftare
  - \* Biții shift-ați în afară sunt pierduți
  - \* Instrucțiunile de rotire îi inserează la loc
- Două tipuri
  - \* Fără Carry
    - » **rol** (ROtate Left)
    - » **ror** (ROtate Right)
  - \* Cu Carry
    - » **rcl** (Rotate through Carry Left)
    - » **rcr** (Rotate through Carry Right)
  - \* Formatul este similar cu cel al instrucțiunilor de shiftare
    - » Două variante (la fel ca la shiftare)
      - Cu operand imediat
      - Cu numărul de rotiri pasat prin CL

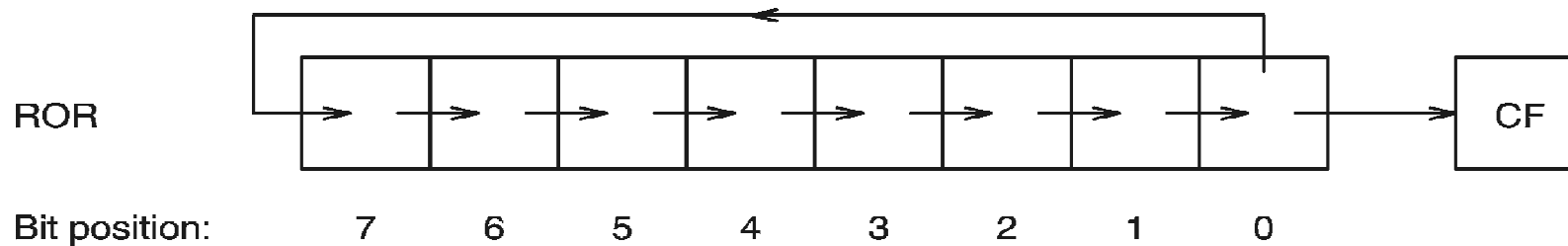
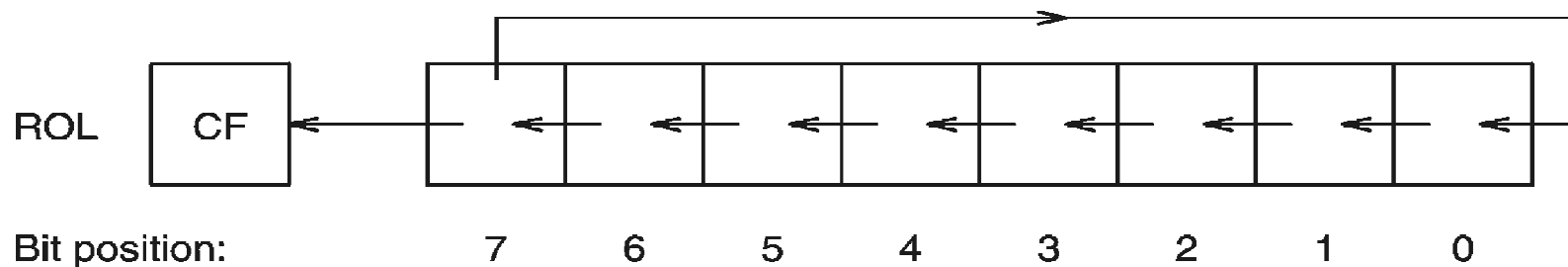
# Rotire fără carry

\* Format:

**rol**            **destination, count**

**ror**            **destination, count**

count– analog shift, valoare imediată sau prin CL





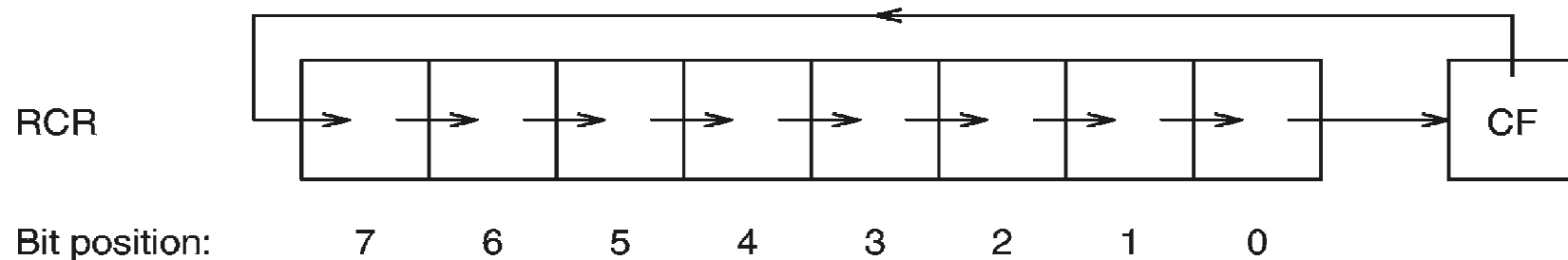
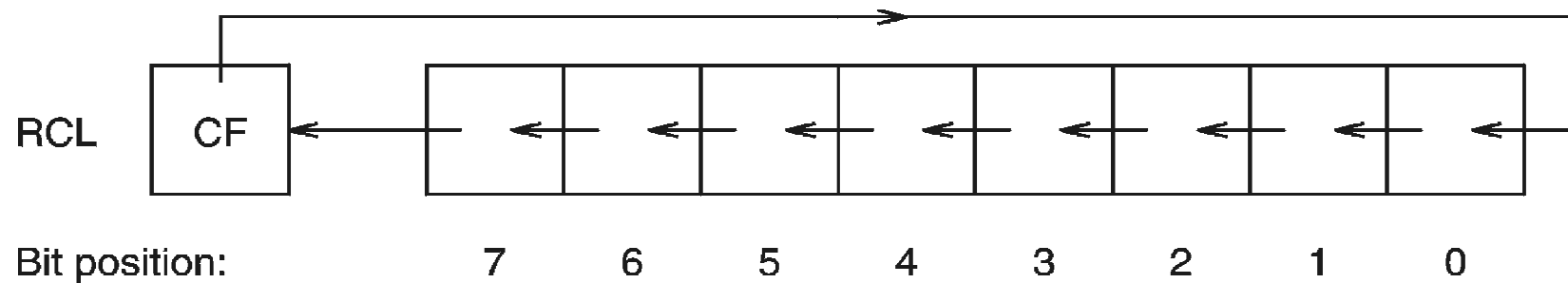
# Rotire cu carry

\* Format

**rcl**      **destination, count**

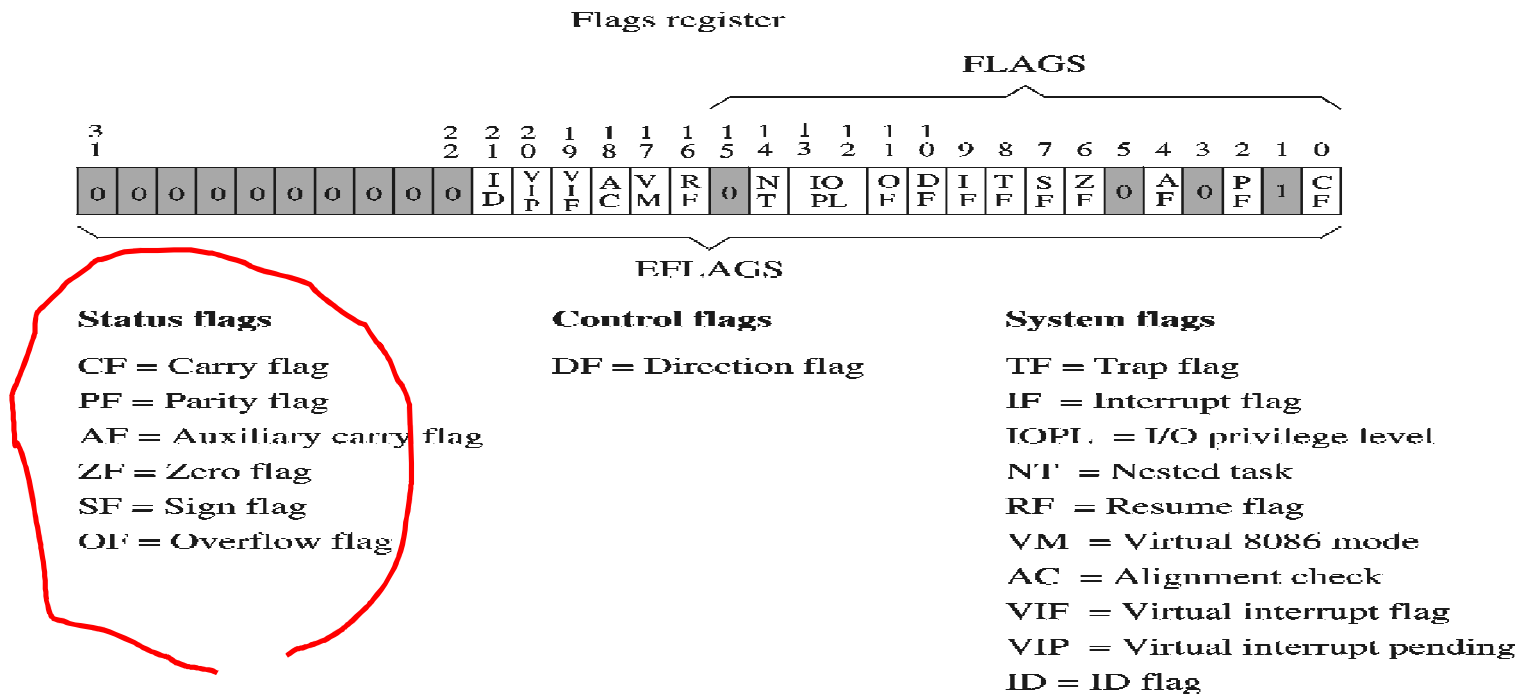
**rcr**      **destination, count**

**count** – analog shift, valoare imediată sau prin CL



# Indicatori de stare

- Șase indicatori ne spun câte o proprietate a rezultatelor instrucțiunilor aritmetice.



# Indicatori de stare

---

- Sunt actualizați pentru a evidenția o proprietate a rezultatelor
  - \* Exemplu: Dacă rezultatul este 0, se setează Zero Flag
- Îndată ce un flag este setat, rămâne setat până când o altă instrucțiune îi modifică starea
- Instrucțiunile afectează în mod diferit indicatorii
  - \* **add** și **sub** le pot schimba pe toate șase
  - \* **inc** și **dec** le pot schimba pe toate mai puțin CF
  - \* **mov**, **push**, și **pop** nu afectează nici un indicator

# Indicatori de stare

---

- Exemplu:

*; initially, assume ZF = 0*

*mov EAX, 55H ; ZF is still zero*

*sub EAX, 55H ; result is 0*

*; ZF is set (ZF = 1)*

*push EBX ; ZF remains 1*

*mov EBX, EAX ; ZF remains 1*

*pop EDX ; ZF remains 1*

*mov ECX, 0 ; ZF remains 1*

*inc ECX ; result is 1*

*; ZF is cleared (ZF=0)*

# Indicatori de stare

---

## Cum utilizam flaguri-le ref. la operatii aritmetice?

- **Studiu de caz: Carry Flag.** Reține dacă în urma unei operații aritmetice pe numere **fără semn** s-a produs depășire

- \* CF e setat în următoarele exemple

|     |          |     |           |
|-----|----------|-----|-----------|
| mov | AL, 0FH  | mov | AX, 12AEH |
| add | AL, 0F1H | sub | AX, 12AFH |

- \* Propagare "împrumut" în adunarea pe mai multe cuvinte

1 ← carry from lower 32 bits

x = 3710 26A8 1257 9AE7H

y = 489B A321 FE60 4213H

7FAB C9CA 10B7 DCFAH

- **Studiu de caz: Overflow flag.** Analog CF, dar pentru **numere cu semn**

- \* Exemple (setează OF, dar nu și CF)

|     |         |              |
|-----|---------|--------------|
| mov | AL, 72H | ; 72H = 114D |
| add | AL, 0EH | ; 0EH = 14D  |

# Indicatori de stare

---

- Cu/fără semn: de unde știe sistemul?

- \* Procesorul nu știe interpretarea
- \* El setează indicatorii carry și overflow pentru ambele

interpretare fără semn

---

```
mov    AL, 72H
add    AL, 0EH
jc     depasire
nu_depasire:
    ....
depasire:
    ....
```

interpretare cu semn

---

```
mov    AL, 72H
add    AL, 0EH
jo     depasire
nu_depasire:
    ....
depasire:
    ....
```

# Instrucțiuni aritmetice

---

- Pentium are o serie de instrucțiuni ce pot lucra la nivel de 8, 16 sau 32 de biți
  - » Adunare: **add, adc, inc**
  - » Scădere: **sub, sbb, dec, neg, cmp**
  - » Înmulțire: **mul, imul**
  - » Împărțire: **div, idiv**
  - » Instrucțiuni ce se folosesc în conjuncție cu cele de mai sus: **cbw, cwd, cdq, cwde, movsx, movzx**

# Instrucțiuni aritmetice

---

- Înmulțire

- \* Mai complexă (costisitoare) ca **add/sub**

- » Produce rezultate de lungime dublă
      - E.g. Înmulțirea a două numere pe 8 biți produce un rezultat ce are nevoie de 16 biți pentru reprezentare
    - » Nu putem folosi o singură instrucțiune de înmulțire atât pentru numere cu semn cât și pentru cele fără semn
      - **add** și **sub** funcționează grație reprezentării în complement față de 2!
      - Pentru înmulțire, avem nevoie de instrucțiuni separate
        - mul** numere fără semn
        - imul** numere cu semn



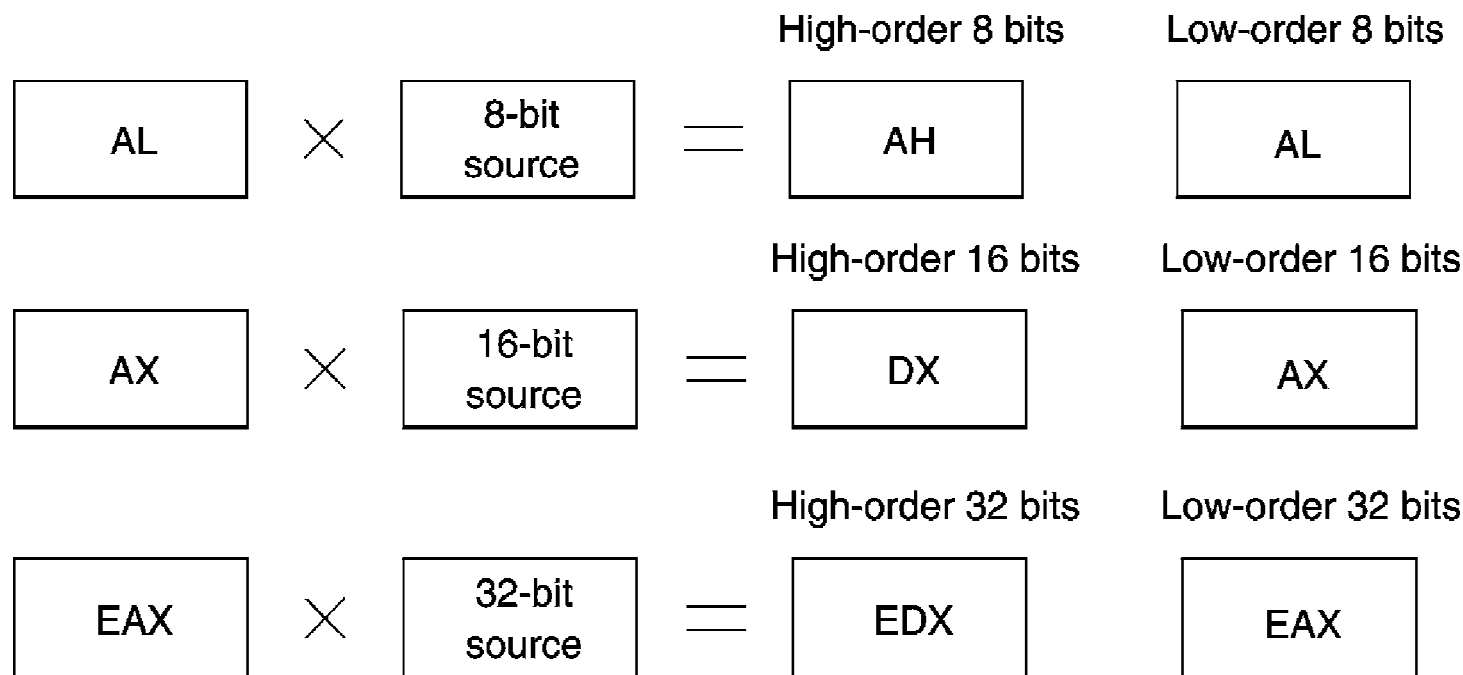
# Instrucțiuni aritmetice

---

- Înmulțire fără semn

**mul** **source**

» Rezultatul și al doilea operand depind de dimensiunea operandului **source**



# Instrucțiuni aritmetice

---

## \* Exemplu

```
mov    AL, 10
mov    DL, 25
mul    DL
```

obținem 250D în AX (rezultatul în AL)

- Instrucțiunea **imul** folosește aceeași sintaxă

## \* Exemplu

```
mov    DL, 0FFH    ; DL = -1
mov    AL, 0BEH    ; AL = -66
imul    DL
```

Obținem 66D în AL

# Instrucțiuni aritmetice

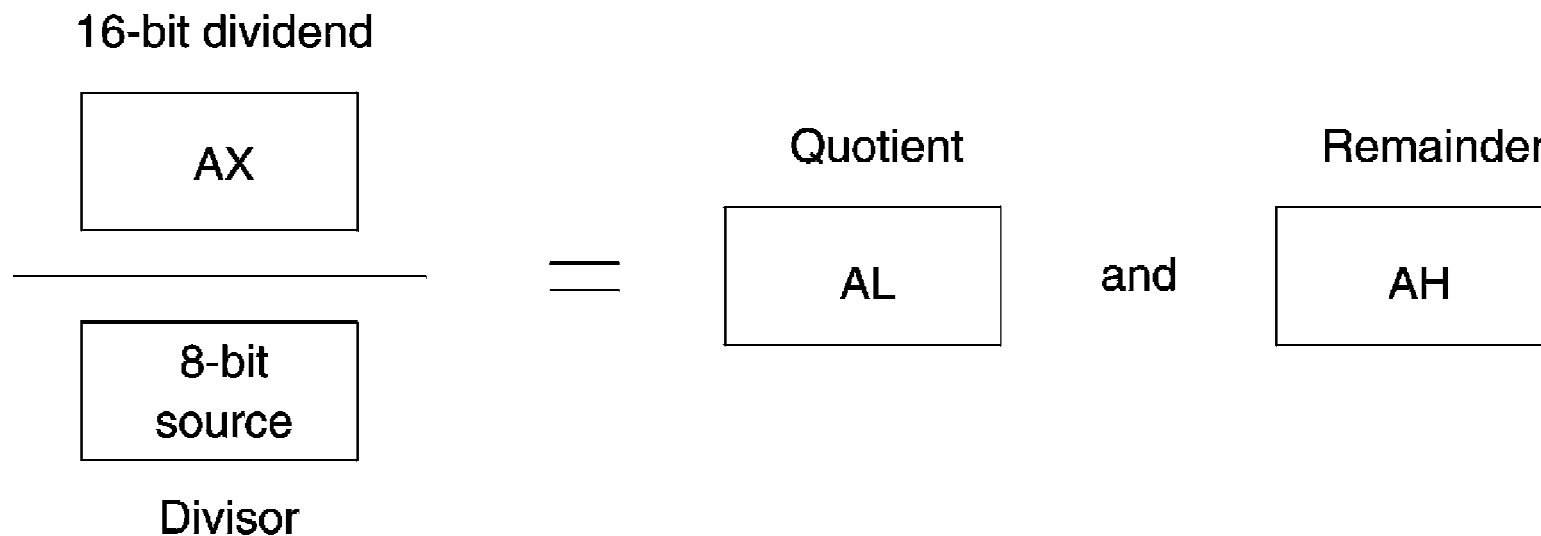
---

- Împărțirea produce două rezultate
  - Câtul
  - Restul
- \* La înmulțire se evită situația de overflow cu registre duble
- \* La împărțire poate să apară!
  - Pentium are o întrerupere software ce se declanșează atunci când se produce overflow la împărțire
- Sintaxa analogă cu înmulțirea
  - div**      **source**    numere fără semn
  - idiv**     **source**    numere cu semn

# Instrucțiuni aritmetice

---

- Deîmpărțitul este de două ori mai lung ca împărțitorul
- Deîmpărțitul se consideră implicit reținut în:
  - \* AX (pentru împărțitor pe 8 biți)
  - \* DX:AX (pentru împărțitor pe 16 biți)
  - \* EDX:EAX (pentru împărțitor pe 32 biți)



# Instrucțiuni aritmetice

---

- Exemplu

```
mov    AX, 251
mov    CL, 12
div    CL
```

avem 20D în AL și 11D drept rest în AH

- Exemplu

```
sub    DX, DX        ; clear DX
mov    AX, 141BH      ; DXAX = 5147D
mov    CX, 012CH      ; CX = 300D
div    CX
```

Avem 17D în AX și 47D rest în DX

# Instrucțiuni aritmetice

---

- La împărțirea cu semn e nevoie de extensie de semn
  - » La numere fără semn, având un număr de 16 biți, făceam completare cu la stânga cu 0-uri până la 32 de biți
  - » Nu mai funcționează în cazul numerelor cu semn
  - » **Soluție: Trebuie făcută extensie la stânga a bitului de semn**
- Instrucțiuni ajutătoare
  - » Două instrucțiuni **mov**
    - movsx dest, src** (move sign-extended **src** to **dest**)
    - movzx dest, src** (move zero-extended **src** to **dest**)
  - » În ambele cazuri, **dest** trebuie să fie un registru
  - » **src** poate fi registru sau locație de memorie
    - Dacă **src** are 8 biți, **dest** trebuie să fie de 16 sau 32 de biți
    - Dacă **src** are 16 biți, **dest** trebuie să fie de 32 de biți

# Structuri de decizie de nivel înalt

---

- Citiți secțiunea 8.5 din carte pentru a vedea cum se implementează:
  - \* if-then-else
  - \* if-then-else with a relational operator
  - \* if-then-else with logical operators AND and OR
  - \* while loop
  - \* repeat-until loop
  - \* for loop

# if-then-else

---

|                                |                   |                   |
|--------------------------------|-------------------|-------------------|
| <b>if (value1 &gt; value2)</b> | <b>mov</b>        | <b>AX, value1</b> |
| <b>bigger = value1;</b>        | <b>cmp</b>        | <b>AX, value2</b> |
| <b>else</b>                    | <b>jle</b>        | <b>else_part</b>  |
| <b>bigger = value2;</b>        | <b>then_part:</b> |                   |
|                                | <b>mov</b>        | <b>bigger, AX</b> |
|                                | <b>jmp</b>        | <b>end_if</b>     |
|                                | <b>else_part:</b> |                   |
|                                | <b>mov</b>        | <b>AX, value2</b> |
|                                | <b>mov</b>        | <b>bigger, AX</b> |
|                                | <b>end_if:</b>    |                   |
|                                | <b>. . .</b>      |                   |



# if-then-else cu operatori logici

---

```
if (ch>= 'a' && ch<= 'z')  
    ch = ch - 32;
```

```
    cmp DL, 'a'  
    jb not_lower_case  
    cmp DL, 'z'  
    ja not_lower_case  
lower_case:  
    mov     AL, DL  
    add     AL, 224  
    mov     DL, AL  
not_lower_case:  
    . . .
```

# Bucle while, for

---

```
while(total < 700)
{
    <loop body>
}
```

```
    jmp while_cond
while_body:
    < loop body >
    . . .
while_cond:
    cmp BX,700
    jl while_body
end_while:
    . . .
```

```
for(i = SIZE-1; i>= 0; i--
)
{
    <loop body>
};
```

---

```
    mov SI,SIZE-1
    jmp for_cond
loop_body:
    < loop body >
    dec SI
for_cond:
    or SI,SI
    jge loop_body
    . . .
```

# Reprezentarea sirurilor de caractere

---

- Două moduri de reprezentare

- \* Stocarea explicită a lungimii (PASCAL)

**string          DB          'Error message'**  
**str\_len        DW          \$-string**

– \$ reprezintă valoarea curentă a contorului de alocare

- \* Caracter santinelă (zero folosit în C)

» Stringuri terminate cu NULL se numesc string-uriASCIIZ

» Atenție zero în ASCII este 48 = 0x30, NULL este octetul 0x00

**string          DB          'Error message', 0**

# Instrucțiuni pe string-uri

---

- Fiecare instrucțiune
  - \* Poate opera cu operanzi pe 8-, 16-,sau 32 de biți
  - \* Actualizează registrele index în mod implicit
    - » Operanzii pe Byte: incrementează/decrementează cu 1
    - » Operanzii pe Word: incrementează/decrementează cu 2
    - » Operanzii pe Dword: incrementează/decrementează cu 4
- Direction flag
  - \* DF = 0: înainte (incrementează registrele index)
  - \* DF = 1: înapoi (decrementează registrele index)
- Cum modificăm DF?
  - std**          set direction flag (DF = 1)
  - cld**          clear direction flag (DF = 0)

# Prefixe de repetare

---

- Fără prefix se execută o singură dată

- \* Repetare necondiționată

**rep** REPEAT

- \* Repetare condiționată

**repe/repz** REPEAT while Equal

REPEAT while Zero

**repne/repnz** REPEAT while Not Equal

REPEAT while Not Zero

# Prefixe de repetare

---

**rep**

**while** ( $ECX \neq 0$ )

Execută instrucțiunea

$ECX := ECX - 1$

**end while**

- ECX este verificat la început
  - \*  $ECX == 0$  ? instrucțiunea nu se execută deloc
  - \* Seamănă cu **JECXZ**

# Prefixe de repetare

---

## **repe/repz**

**while** ( $ECX \neq 0$ )

Execută instrucțiunea

$ECX := ECX - 1$

**if** ( $ZF = 0$ )

**then**

exit loop

**end if**

**end while**

- Se folosește cu **cmps** și **scas**

# Prefixe de repetare

---

## **repne/repnz**

**while** ( $ECX \neq 0$ )

Execută instrucțiunea

$ECX := ECX - 1$

**if** ( $ZF = 1$ )

**then**

exit loop

**end if**

**end while**



# Instrucțiuni pentru string-uri

---

- MOVS, LODS, STOS

- Move string (movs)

- \* Format

- `movsb dest_string, source_string`

- `movsb` ; operands are bytes

- `movsw` ; operands are words

- `movsd` ; operands are doublewords

- \* sursa= DS:ESI

- \* destinația = ES:EDI

- \* În Linux, DS = ES, ambele pregătite de kernel

# Instrucțiuni pentru string-uri

---

**movsb** --- move a byte string

ES:EDI:= (DS:ESI) ; copy a byte

if (DF=0) ; forward direction

then

ESI := ESI+1

EDI := EDI+1

else ; backward direction

ESI := ESI-1

EDI := EDI-1

end if

Nici un flag nu este modificat

# Instrucțiuni pentru string-uri

---

## Exemplu MOVSB:

```
.section data
string1      db      'The original string',0
strLen       EQU     $ - string1

.section bss
string2      resb     80

.section text
Start_:
mov     ECX, strLen      ; strLen includes NULL
mov     ESI, string1
mov     EDI, string2
cld                          ; forward direction
rep     movsb
```

# Instrucțiuni pentru string-uri

---

- **Load String (LODS)**

- \* Copiaza valoarea indicată de DS:ESI în:

- » AL (**lods b**), AX (**lodsw**), EAX (**lodsd**)

- \* Prefixul de repetare nu are sens

**lods b** ---load a byte string

AL := (DS:ESI) ; copy a byte

**if** (DF=0) ; forward direction

**then**

ESI := ESI+1

**else** ; backward direction

ESI := ESI-1

**end if**

Nici un flag nu este modificat

# Instrucțiuni pentru string-uri

---

## Store String (STOS) [Operația complementară LODS]

- \* Copiaza valoarea din
  - » AL (**lods**b), AX (**lod**sw), EAX (**lod**sd)
- \* În stringul pointat de ES:EDI
- Prefixul de repetare poate fi folosit pentru inițializarea unui bloc de memorie

```
stosb --- store a byte string
      ES:EDI := AL          ; copy a byte
if (DF=0)                  ; forward direction
then
      EDI := EDI+1
else                      ; backward direction
      EDI := EDI-1
end if
```

Nici un flag nu este modificat

# Instrucțiuni pentru string-uri

---

## Exemplu STOSW: Inițializare array1 cu -1

```
.bss
array1    resw    100

.text
mov     ECX, 100
mov     EDI, array1
mov     AX, -1
cld                      ; forward direction
rep     stosw
```

# Instrucțiuni pentru string-uri

---

- Prefixele de repetare **nu se folosesc** cu **lods /stos**

- Exemplu procesare text:

```
    mov     ECX, strLen
    mov     ESI, string1
    mov     EDI, string2
    cld     ; forward direction
loop1:
    lodsb
    or      AL, 20H
    stosb
    loop    loop1
```

# Comparare string-uri

---

- **Compare String (CMPS)**

- \* Dimensiuni disponibile: CMPSB, CMPSW, CMPSD

**cmpsb** --- compară octeții de la DS:ESI și ES:EDI, setează FLAGS

```
if (DF=0)           ; forward direction
then
    ESI := ESI+1
    EDI := EDI+1
else                 ; backward direction
    ESI := ESI-1
    EDI := EDI-1
end if
```

Setează FLAGS ca și instrucțiunea **cmp**  
[DS:ESI] - [ES:EDI]



# Comparare string-uri

---

## Exemplu 1 CMPSB (FORWARD):

```
.data
string1    db    'abcdfghi',0
strLen     EQU   $ - string1
string2    db    'abcdefgh',0

.text
mov     ECX, strLen
mov     ESI, string1
mov     EDI, string2
cld          ; forward direction
repe    cmpsb
dec     ESI
dec     EDI    ;ESI & EDI pointing to the last
               ;character that differs
```

# Comparare string-uri

---

## Exemplu 2 CMPSB (**BACKWARD**):

```
.data
string1    db    'abcdfghi',0
strLen     EQU   $ - string1 - 1
string2    db    'abcdefgh',0

.text
mov     ECX, strLen
mov     ESI, string1 + strLen - 1
mov     EDI, string2 + strLen - 1
std     ; backward direction
repne   cmpsb
inc     ESI ; ESI & EDI pointing to the first character that matches
inc     EDI ; in the backward direction
```

# Scanare string-uri

---

- **Scan String (SCAS)**

- \* Dimensiuni disponibile: SCASB, SCASW, SCASD

**scasb** --- Scan a byte string

Compară AL cu octetul [ES:EDI], set FLAGS

**if** (DF=0) ; forward direction

**then**

EDI := EDI+1

**else** ; backward direction

EDI := EDI-1

**end if**

Setează FLAGS ca și instrucțiunea **cmp**

[DS:ESI] - [ES:EDI]

- **scasw** folosește AX; **scasd** folosește EAX

# Scanare string-uri

---

## EXEMPLUL 1 SCASB

```
.data
string1 db 'abcdefgh',0
strLen EQU $ - string1

.text
mov ECX,strLen
mov EDI,string1
mov AL,'e'
;character to be searched

cld ;forward direction
repne scasb

dec EDI
; leaves EDI pointing to
  'e' in string1
```

## EXEMPLUL 2 SCASB

```
.data
string1 db '      abc',0
strLen EQU $ - string1

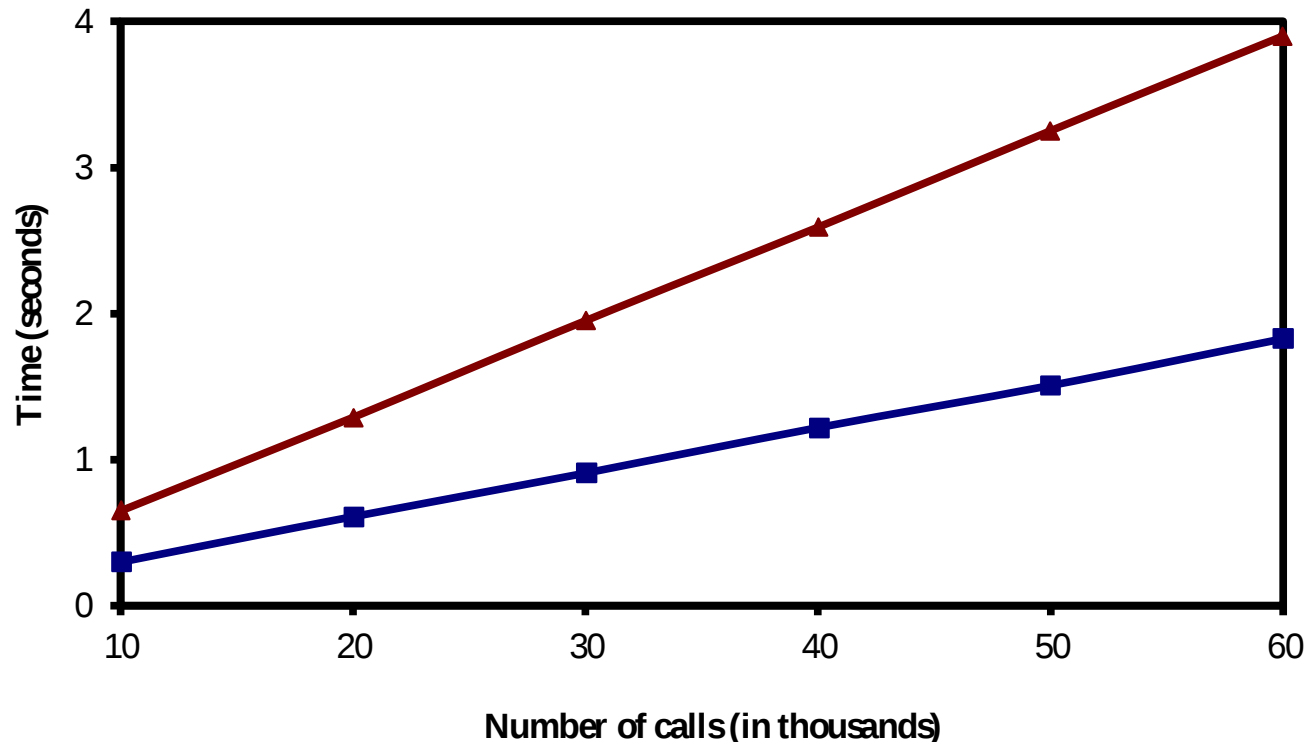
.text
mov ECX,strLen
mov EDI,string1
mov AL,' '
; character to be searched

cld
; forward direction
repe scasb
dec EDI
; EDI pointing to the first
  non-blank character 'a'
```

# Performanță instrucțiuni pentru string-uri

---

- Două avantaje pentru instrucțiunile dedicate:
  - \* Registre index actualizate automat
  - \* Pot folosi doi operanzi direct din memorie
- Masuratori efectuate pentru mutarea a 50000 de elemente utilizand MOVSD:



# Performanță instrucțiuni pentru string-uri

---

