Instrucțiuni x86

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

Modificat: 22-Oct-23

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:

```
(destination)-(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

jz	**jump if zero** (ZF == 1)
jnz	**jump if not zero** (ZF == 0)
jc	**jump if carry** (CF == 1)
jnc	**jump if not carry** (CF == 0)
jo	**jump if overflow** (OF == 1)
js	**jump if sign** (SF == 1)
jp	**jump if parity** (PF == 1)

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

Instrucțiuni de salt (jump)

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

```
je        if equal (ZF==1)
jne       if not equal (ZF==0)
ja        if greater (CF==0 && ZF==0)
jae       if greater or equal (CF==0)
jb        if less (CF==1)
jbe       if less or equal (CF==1||ZF==1)
```

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

```
je        if equal (ZF==1)
jne       if not equal (ZF==0)
jg        if greater (ZF==0 && SF==OF)
jl        if less (SF != OF)
jge       if greater or equal (SF==OF)
jle       if less or equal (ZF==1||SF!=OF)
```

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`**
  - » **`Inainte de apel se salveaza adresa de retur pe stiva`**
  - » Revenirea se face la apelul instructiunii RET care extrage adresa de retur de pe stiva si executa salt la acea locatie

- Exemplu: EBX conține un pointer la o procedură

```
CALL    Calculeaza
…       ;aici se revine dupa call
Calculeaza:
…       ;aici este corpul rutinei
RET     ;aici se termina rutina si se
        ;revine in 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 ≠ 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 ≠ 0 and ZF = 1)

`loopne/loopnz target`

Action: ECX = ECX – 1

Jump to target if (ECX ≠ 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
    - **`sal (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
```

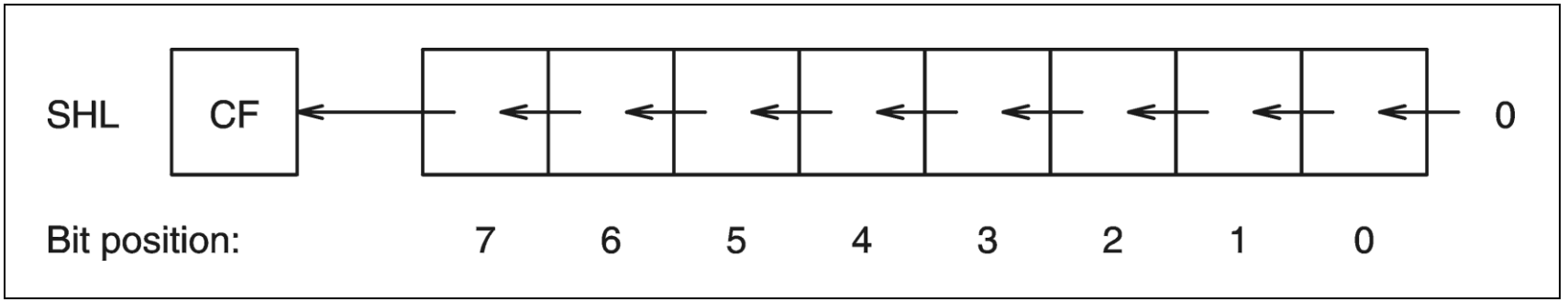
* Semantică: Shiftare stânga/dreapta a **`destination`** cu valoarea din **`count`** sau registrul **`CL`** (nu modifica CL). **`destination`** poate fi pe 8, 16 sau 32 de biți; se poate afla în memorie sau într-un registru
* Exemplu: prelucrare pe biți

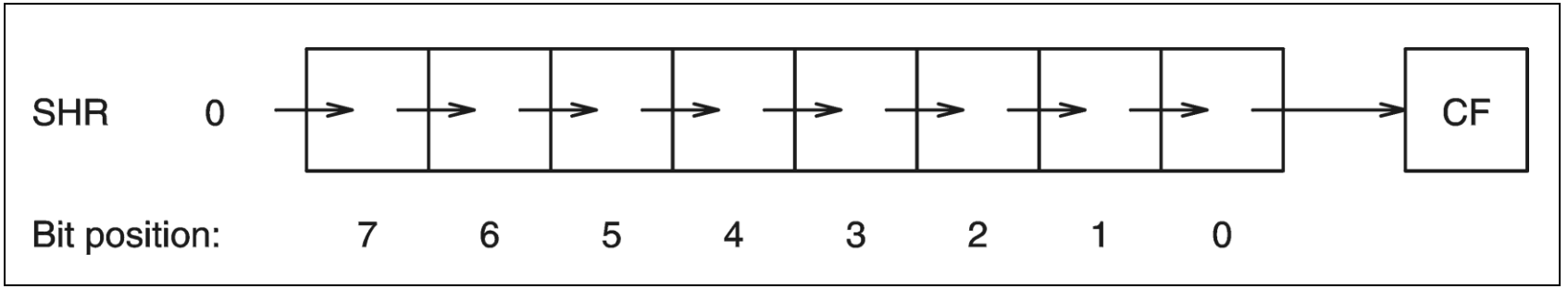
```
; AL contains the byte to be encrypted
mov     AH,AL
shl     AL,4        ; move lower nibble to upper
shr     AH,4        ; move upper nibble to lower
or      AL,AH       ; paste them together
; AL has the encrypted byte
```

* 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





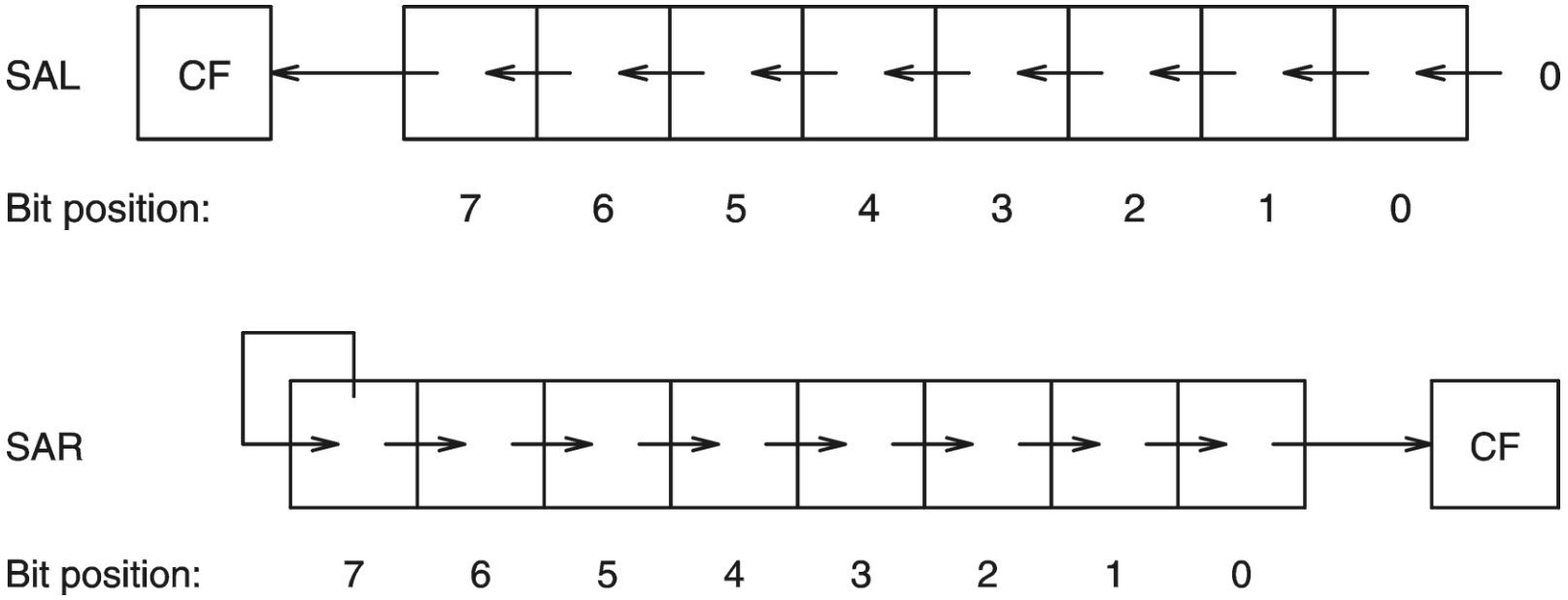
Shiftare logică

- 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

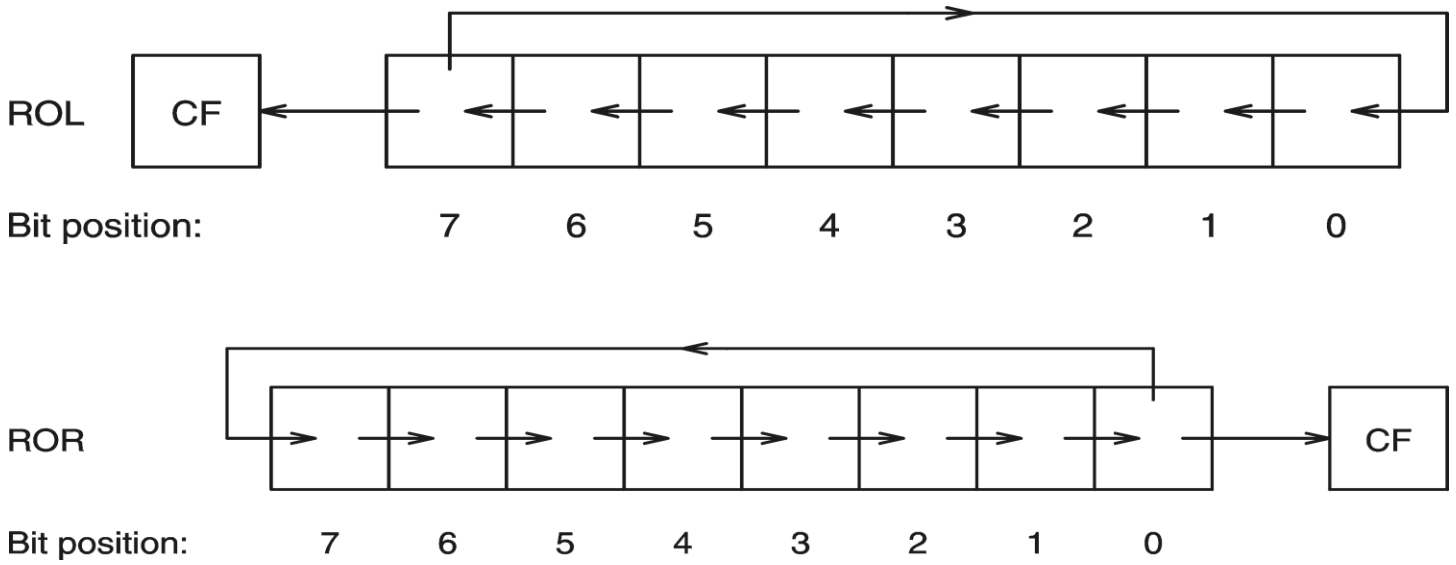
- Problemă la shiftare
  - Biții shift-ați în afară sunt pierduți
  - Instrucțiunile de rotire îi inserează la loc
- Două tipuri
  - Fără Carry
    - **`rol`** (ROtate Left)
    - **`ror`** (ROtate Right)
  - Cu Carry
    - **`rcl`** (Rotate through Carry Left)
    - **`rcr`** (Rotate through Carry Right)
  - 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

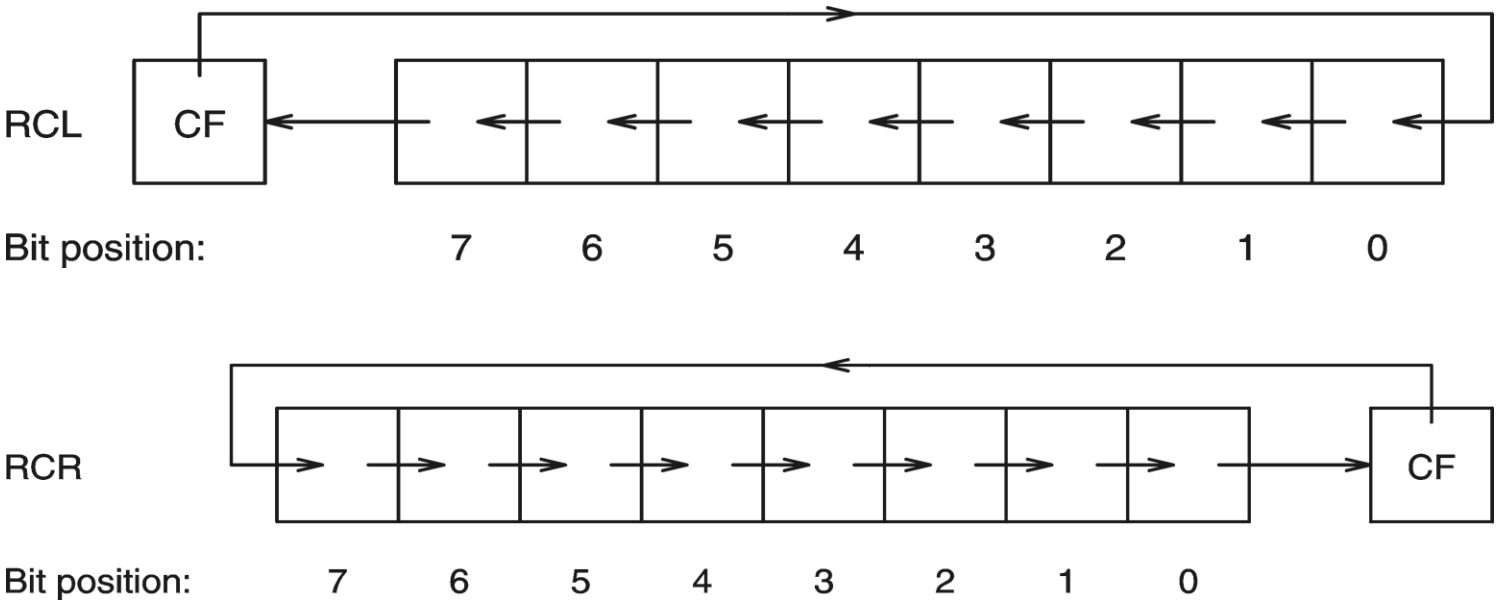


Rotire cu carry

* Format

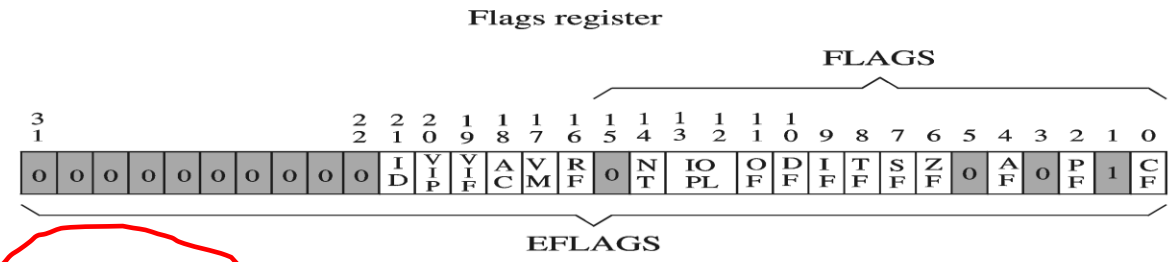
```
rcl     destination,count
rcr     destination,count
```

count– analog shift, valoare imediată sau prin CL



Indicatori de stare

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



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

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

interpretarefărăsemn

```
    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

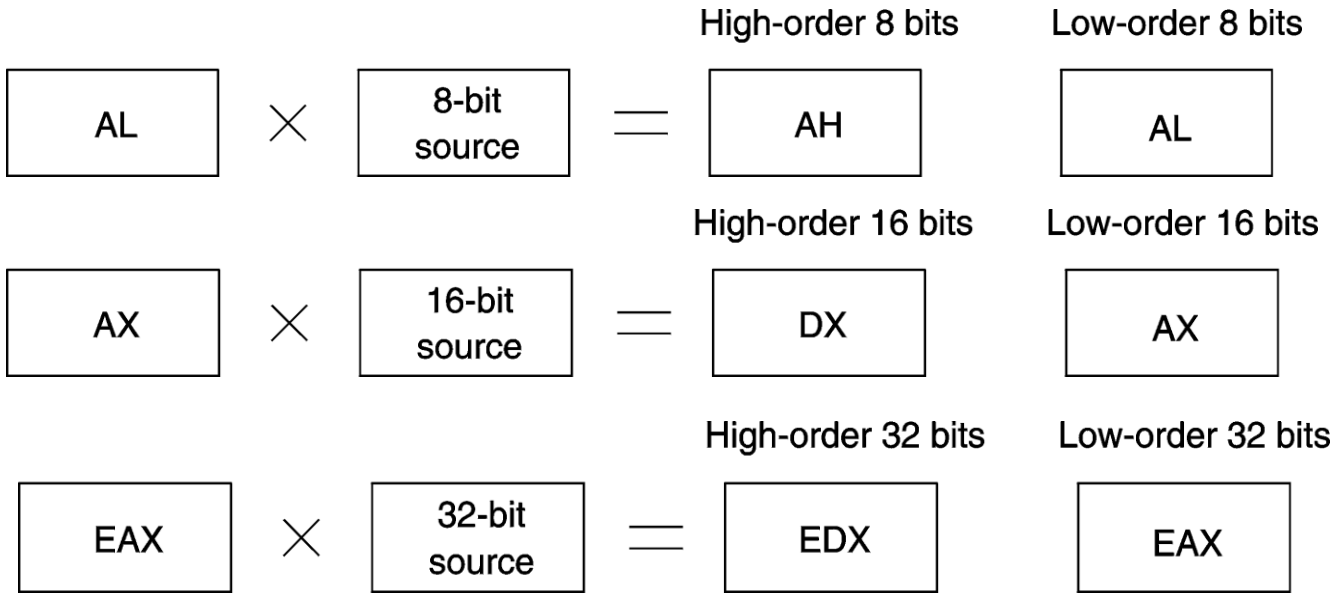
        `imul` numere cu semn

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
```

obținem250D în AX (rezultatulîncape înAL)

- Instrucțiunea **imul** folosește aceeași sintaxă
  * Exemplu

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

Obținem 66D în AL

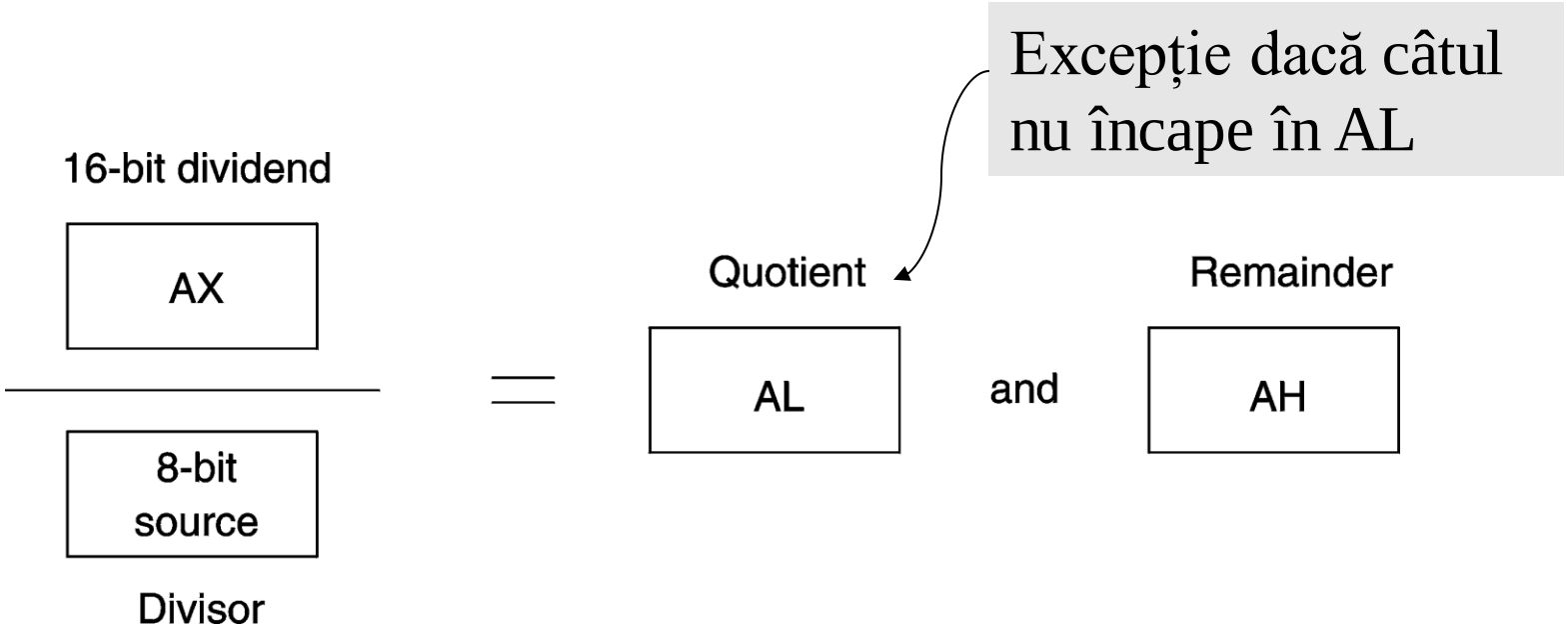
Instrucțiuni aritmetice

- Împărțirea produce două rezultate
  - Câtul
  - Restul
  - La înmulțire se evită situația de overflow cu registre duble
  - întrerupere software ce se declanșează atunci când se produce overflow la împărțire (câtul nu încape)
- 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) => AL, AH
  - DX:AX (pentru împărțitor pe 16 biți) => AX, DX
  - EDX:EAX (pentru împărțitor pe 32 biți) => EAX, EDX



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
  - » **Solutie: 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 16biți, **`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

```
if (value1 > value2)
  bigger = value1;
else
  bigger = value2;
```

```
    mov     AX, [value1]
    cmp     AX, [value2]
    jle     else_part
then_part:
    jmp     end_if
else_part:
    mov     AX,value2
end_if:
    mov     [bigger], AX
    . . .
```

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
   . . .
```

SET condițional

SET{cond} reg8/mem8

- Condiție = testează biți din EFLAGS (z, ge, c, o, …)
- destinația trebuie să fie de 8biți
- dacă condiția este îndeplinită, se setează la 1
- dacă condiția NU este îndeplinită, se setează la 0
- EFLAGS nu este afectat

```
cmp edx,ecx
setge al
```

al = ((int)edx >= (int)ecx) ? 1 : 0

MOV condițional

CMOV{cond} dst, src

- Condiție = testează biți din EFLAGS (z, ge, c, o, …)
- destinația trebuie să fie un registru
- dacă condiția este îndeplinită, se execută MOV
- dacă condiția NU este îndeplinită, dst rămâne neschimbat
- EFLAGS nu este afectat

MOV condițional

Mnemonic	Required Flag State	Description
CMOVO	OF = 1	Conditional move if overflow
CMOVNO	OF = 0	Conditional move if not overflow
CMOVB CMOVC CMOVNAE	CF = 1	Conditional move if below Conditional move if carry Conditional move if not above or equal
CMOVAE CMOVNB CMOVNC	CF = 0	Conditional move if above or equal Conditional move if not below Conditional move if not carry
CMOVE CMOVZ	ZF = 1	Conditional move if equal Conditional move if zero

- CMOVNE, CMOVBE, CMOVA, CMOVG, ...

```
int adjust(int x){
  if(x > max){
     max = x;
     return 1;
  }
  return 0;
}
```

```
adjust:
    enter 0,0
    mov edx, [ebp+8];x
    cmp edx, [max]
    jg update_max
    mov eax, 0
    jmp ret_adj
update_max:
    mov [max], edx
    mov eax, 1
ret_adj:
    leave
    ret
```

```
adjust_nojump:
    enter 0,0
    mov ecx, [max]
    mov eax, 0
    cmp [ebp+8], ecx
    cmovg ecx, [ebp+8]
    mov [max], ecx
    setg al
    leave
    ret
```

MOV condițional

Avantaje performanță

- Mai bun pentru pipeline (fetch, decode, execute)
- Permite execuția secvențială pură
- Benchmarks https://github.com/xiadz/cmov

Limitări

- Doar pentru atribuiri simple – operatorul ?:
- nu poate înlocui secvența if..then..else - sunt necesare salturi condiționale
- doar pentru registre, doar 16, 32, 64 biți