LAB 5 – Instrucțiunile procesorului 8086 (partea a doua)

Instructiuni logice (AND, OR, XOR, NOT), de testare (TEST), pt deplasare (SHL=SAL, SHR, SAR), pt rotire (ROL, ROR, RCL, RCR)

CMP op1, op2; salturi ((ne)conditionat-> flagurile, operanzilor) jmp et, JC et, jnC et, jP, jZ, jnZ, jE, jnE

1. Instrucțiuni logice

Instrucțiuni pe biți:

Aceste instrucţiuni consideră operanzii ca simple şiruri de biţi, aplicând o funcţie logică tuturor *biţilor* din reprezentarea numărului, în general fiecare bit fiind considerat independent de ceilalţi (nu există *transport* între poziţiile binare).

La acest tip de instrucţiuni, cum nu există transport între biţi, rolul flagului AF poate fi alterat.
Instrucţiunile logice (cu excepţia NOT) resetează flagurile CF şi OF.

1. **logice:** NOT, AND, OR, XOR
2. **de testare/comparare:** TEST,
3. **de deplasare (shift):** SHL/SAL, SHR, SAR,
4. **de rotire (rotate):** ROL, ROR, RCL, RCR

Instrucţiunile care se execută la nivel de biţi au în general 2 operanzi, iar rezultatul operaţiei e depus în primul dintre ei (în general se numeşte operand destinaţie). Ca operanzi nu se admit regiştri segment, nici IP sau FLAGS.

Instrucţiunile logice NOT, AND, OR, XOR au fost suportate încă de la **8086**↑, acestea fiind folosite la execuţia software (în logică booleană) a operaţiilor corespunzătoare din circuitele logice.

Există ***4 operaţii logice*** majore pe biţi: **NOT**, **AND**, **OR** şi **XOR**, iar Tabelul 5-1.1 furnizează tabelele de adevăr corespunzătoare. Operaţiile logice se realizează asupra şirurilor de biţi, deci se va aplica o funcţie logică fiecărui bit în parte (din reprezentarea numărului), la acest tip de instrucţiuni neexistând transport.

Tabelul 4-1.1. Reguli de obţinere a valorilor la diferite operaţii logice efectuate în binar

NOT a unei cifre binare

NOT	0	1
	1	0

AND între 2 cifre binare

AND	0	1
0	0	0
1	0	1

OR între 2 cifre binare

OR	0	1
0	0	1
1	1	1

XOR între 2 cifre binare

XOR	0	1
0	0	1
1	1	0

Operaţiile **AND** şi **OR** se utilizează în special atunci când se doreşte **mascarea** anumitor biţi (se folosesc biţi de **0**, respectiv de **1** pentru mască), în timp ce instrucţiunea **XOR** se foloseşte atunci când se doreşte **complementarea** anumitor biţi.

Tabelul 5-1.2. Mascarea biţilor folosind operaţiile AND, OR şi XOR

AND

xxxx xxxx operand
0000 1111 masca
0000 xxxx **rezultat**

OR

xxxx xxxx operand
0000 1111 masca
xxxx **1111 rezultat**

XOR

xxxx xxxx operand
0000 1111 masca
xxxx $\overline{\text{xxxx}}$ **rezultat**

4.1.1. Instrucţiunea NOT

Instrucţiunea NOT (***One's Complement Negation***) realizează o negare logică (fiecare valoare de 1 va deveni 0 și fiecare valoare de 0 va deveni 1) aplicată bit cu bit la valoarea care constituie operand; instrucţiunea NOT neagă toţi biţii operandului destinaţie prin calcularea complementului faţă de 1 al acestuia (rezultatul se va depune tot în operandul specificat în instrucţiune).

NOT destinaţie ***; realizează operaţia logică NOT asupra fiecărui bit din operandul destinaţie***
NOT {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$} ***; (echivalent cu complement faţă de 1)***

Figura 5-1.1. Ilustrarea modului de operare al instrucţiunii ***NOT***

Observaţii:

- Instrucţiunea NOT are un singur operand explicit care poate fi registru sau memorie;
- Nu se folosesc ca operand regiștrii segment, nici [-/E/R] IP sau [-/E/R] FLAGS;
- Nici valorile imediate nu sunt acceptate ca operand;
- Această instrucţiune *nu afectează flagurile*.

4.1.2. Instrucţiunea AND

Instrucţiunea **AND** (***Logical AND*** - ŞI logic bit cu bit) efectuează operaţia logică AND (în română SI logic) între operanzii *sursă* și *destinaţie*. Un bit din reprezentare va fi setat (pus în 1) doar dacă biţii corespunzători din *sursă* și *destinaţie* sunt ambii 1; altfel, bitul va fi resetat (pus în 0). Rezultatul se depune în operandul *destinaţie*, iar valoarea din operandul *sursă* nu este afectată.

AND destinaţie, sursă ; realizează operaţia logică AND asupra fiecărei perechi de biţi din cei 2 operanzi

AND $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}, \{reg_{8,16,32,64} | mem_{8,16,32,64} | imed_{8,16,32}\}$

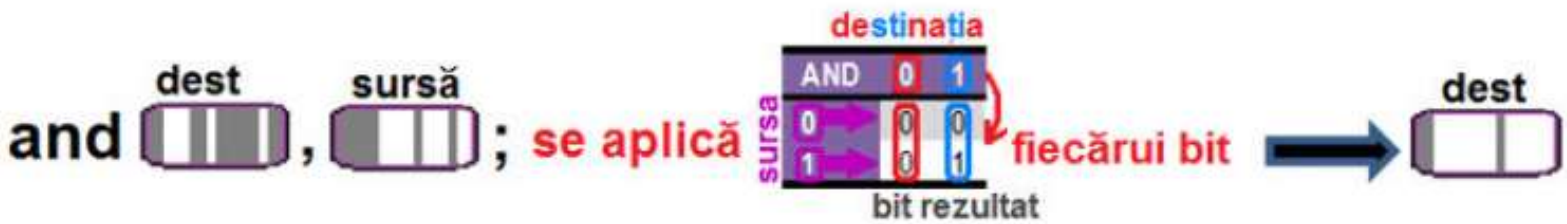

Figura 4-1.2. Ilustrarea modului de operare al instrucţiunii ***AND***

Observaţii:

- Instrucţiunea AND *modifică flagurile aritmetice* SF, ZF, PF conform rezultatului operaţiei,
  - dar flagurile OF și CF sunt zero, OF=CF=0, iar AF este *nedefinit*;
- Operanzii trebuie să aibă dimensiuni identice;
- Este interzis ca ambii operanzi să fie locaţii de memorie;
- Operanzii pot fi atât numere fără semn cât și numere cu semn, dar aici nu se va interpreta valoarea respectivă decât ca o înșiruire de biţi;
- La folosirea regiștrilor pe 64 biţi, valoarea imediată e pe 8 sau 32 biţ și e extinsă cu semn la dimensiunea impusă de operandul destinaţie.

4.1.3. Instrucţiunea OR

Instrucţiunea **OR** (***Logical Inclusive OR***) efectuează operaţia logică OR (în română SAU logic) între operanzii *sursă* și *destinaţie*. Un bit din reprezentare va fi resetat (pus în 0) doar dacă biţii corespunzători din *sursă* și *destinaţie* sunt ambii 0; altfel, bitul va fi setat (pus în 1). Rezultatul se depune în operandul *destinaţie*, iar valoarea din operandul *sursă* nu este afectată.

OR destinaţie, sursă ***; realizează operaţia logică OR asupra fiecărei perechi de biţi din cei 2 operanzi***

OR $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}, \{reg_{8,16,32,64} | mem_{8,16,32,64} | imed_{8,16,32}\}$

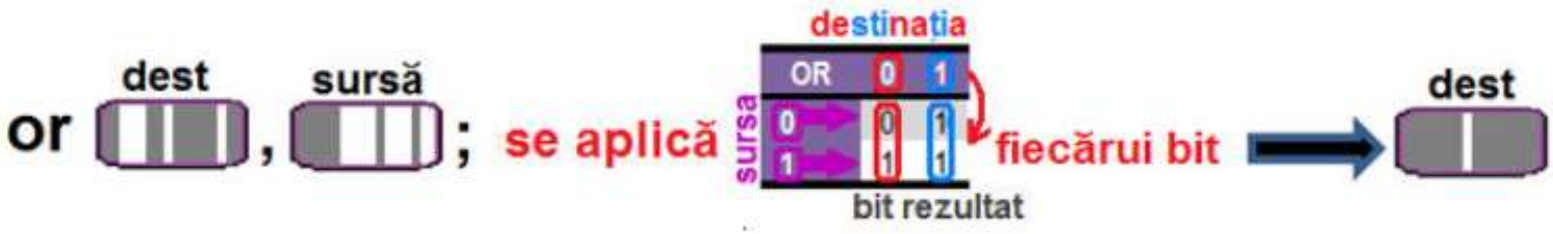

Figura 4-1.3. Ilustrarea modului de operare al instrucţiunii ***OR***

Observaţii: (ca la AND)

- Instrucţiunea OR *modifică flagurile aritmetice* SF, ZF, PF conform rezultatului operaţiei,
  - dar flagurile OF și CF sunt OF=CF=0, iar AF este *nedefinit*;
- Operanzii trebuie să aibă dimensiuni egale;
- Este interzis ca ambii operanzi să fie locaţii de memorie;
- Operanzii pot fi atât numere fără semn cât și numere cu semn, dar aici nu se va interpreta valoarea respectivă decât ca o înșiruire de biţi;
- La folosirea regiștrilor pe 64 biţi, valoarea imediată e pe 8 sau 32 biţi și e extinsă cu semn la dimensiunea impusă de celălalt operand, cel destinaţie.

4.1.4. Instrucţiunea XOR

Instrucţiunea **XOR** (***Logical Exclusive OR***) efectuează operaţia logică XOR (în română SAU-exclusiv logic bit cu bit) între operanzii *sursă* și *destinaţie*. Un bit din reprezentare va fi setat (pus în 1) doar dacă biţii corespunzători din *sursă* și *destinaţie* sunt diferiţi; altfel, bitul va fi resetat (pus în 0). Rezultatul se depune în operandul *destinaţie*, iar valoarea din operandul *sursă* nu este afectată.

XOR destinaţie, sursă ***; realizează operaţia logică XOR asupra fiecărei perechi de biţi din cei 2 operanzi***

XOR $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}, \{reg_{8,16,32,64} | mem_{8,16,32,64} | imed_{8,16,32}\}$

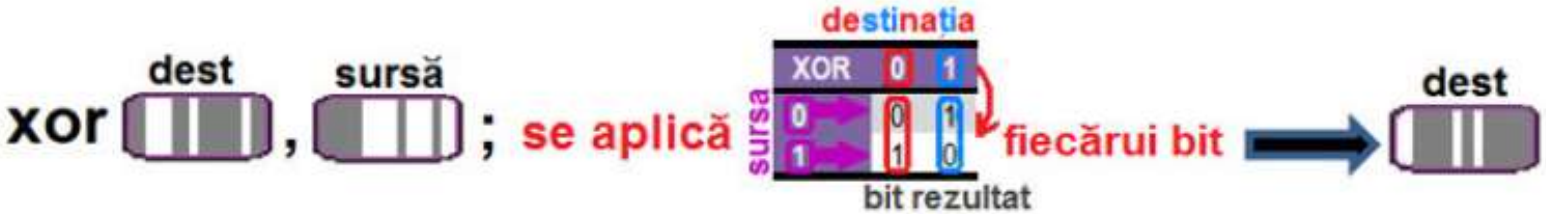

Figura 4-1.4. Ilustrarea modului de operare al instrucţiunii ***XOR***

Observaţii: (ca la AND)

- Instrucţiunea XOR *modifică flagurile aritmetice* SF, ZF, PF conform rezultatului operaţiei,
  - flagurile OF și CF sunt OF=CF=0, iar AF este *nedefinit*;
- Operanzii trebuie să aibă dimensiuni egale;
- Este interzis ca ambii operanzi să fie locaţii de memorie;
- Operanzii pot fi atât numere fără semn cât și numere cu semn, dar aici nu se va interpreta valoarea respectivă decât ca o înșiruire de biţi;
- La folosirea regiștrilor pe 64 biţi, valoarea imediată e pe 8 sau 32 biţi și e extinsă cu semn la dimensiunea impusă de celălalt operand, cel destinaţie.

Exemple de *instrucţiuni ilegale:*

```
not BL, AL          ; sintaxa eronată, NOT are un singur operand
not DS              ; nu se acceptă operand registru segment
and BX, AL          ; dimensiunea operanzilor trebuie să fie aceeaşi
and a, [BX]         ; operanzii să nu fie ambii din memorie
and DS, AX          ; regiştri segment nu pot fi operanzii (nici IP sau FLAGS)
or DX,BL            ; dimensiunea operanzilor trebuie să fie aceeaşi
or [EBX], b         ; operanzii să nu fie ambii din memorie
or DS, BX           ; regiştri segment nu pot fi operanzii (nici IP sau FLAGS)
xor EDX, BX         ; dimensiunea operanzilor trebuie să fie aceeaşi
xor [EBX], [EDX]    ; operanzii să nu fie ambii din memorie
xor DS, BX          ; regiştri segment nu pot fi operanzii (nici IP sau FLAGS)
```

Exemple de *instrucţiuni legale*:

Exemplul 4-1.1 Presupunând EAX=12345678h, EBX=0FFFF0000h, operaţiile logice pot fi realizate la mai multe dimensiuni ale operanzilor:

```
mov EAX, 1234 5678h     ; EAX=   12345678h=0001 0010 0011 0100 0101 0110 0111 1000b
mov EBX, 0FFFF000Fh     ; EBX=0FFFF000Fh=1111 1111 1111 1111 0000 0000 0000 1111b
and EAX, EBX            ; EAX = 1234 0008h=0001 0010 0011 0100 0000 0000 0000 1000b
or  EAX, EBX            ; EAX=0FFFF567Fh=1111 1111 1111 1111 0101 0110 0111 1111b
and AX,BX               ; AX=      0008h=                    0000 0000 0000 1000b
or  AX,BX               ; AX=      567Fh=                    0101 0110 0111 1000b
and AH,BL               ; AH=      06h =                     0000 0110 b
or AL,69h               ; AL=      79h =                               0111 1001b
and AL,69h              ; AL=      68h =                               0110 1000b
```

Exemplul 4-1.2 Obţinerea cifrei zecimale din codul ei Ascii:

```
mov AL, 31h
and AL, 0Fh          ; mască în 0 pe biţii 7...4 => AL = 01h
```

Exemplul 4-1.3 Obţinerea codurilor Ascii ale unui nr de 2 cifre zecimale:

```
mov AL, 3
mov BL, 6
mul BL               ; AX = AL * BL = 3*6=18
aam                  ; instrucţiunea AAM obţine cele 2 cifre în regiştrii AH şi AL: AH=01h, AL=08h
or AX, 3030h         ; se ajustează rezultatul pt obţinerea valorilor Ascii => AX=3138h
```

Exemplul 4-1.4 Obţinerea opusului unui număr, dar fără a folosi instrucţiunea neg.

```
mov AX, 1234h        ; AX=  1234h =0001 0010 0011 0100b
not AX               ; AX=EDCBh=1110 1101 1100 1011b – s-a obţinut complementul faţă de 1 al nr. 4660=1234h,
                     ; adică numărul 0EDCBh=-4661
add AX, 1            ; AX= EDCCh = -4660, adică s-a obţinut complementul faţă de 2 al nr 4660
```

Exemplul 4-1.5 Obţinerea numărului dat de biţii 8...5 în registrul AX.

```
mov AX, 1324h            ; AX=1234h=0001.0011.0010.0100b
and AX, 0000000111100000b ; AX = 0000.0001.0010.0000b – s-au reţinut doar biţii 8...5
mov BL, 32               ; pt a aduce numărul pe biţii 3...0, îl vom împărţi cu 32
div BL                   ; AX = 0000.0000.0000.1001b = 0009h
```

Exemplul 4-1.6 Obţinerea numărului dat de biţii 8...5 în registrul AX în poziţia biţilor 15...12.

```
mov AX, 1324h            ; AX=1234h=0001.0011.0010.0100b
and AX, 0000000111100000b ; AX = 0000.0001.0010.0000b – s-au reţinut doar biţii 8...5
mov BX, 128              ; pt a aduce numărul pe biţii 15...12, îl vom înmulţi cu 2^7
mul BX                   ; AX =1001.0000.0000.0000b = 9000h
```

Instrucţiunea TEST:

Încă de la **8086**↑ a fost suportată instrucţiunea **TEST** pentru testarea anumitor biţi de 1 dintr-un operand pe 8 sau 16 biţi.

Instrucţiunea **TEST** (***Logical Compare***) realizează o operaţie AND fictivă între operandul sursă1 şi operandul sursă2, iar rezultatul nu se va reflecta în destinaţie. Afectează SF, ZF, PF la fel ca instrucţiunea AND.

Această instrucţiune poate fi folosită pt a testa biţii de 1 dintr-un operand (dacă primul bit e bit de 1 sau nu, dacă nu există biţi de 1, dacă e un nr. impar de biţi de 1). Testul se poate realiza asupra tuturor biţilor operandului sau doar asupra unui subset al lor (în funcţie de mască).

TEST sursă1, sursă2 ***; (sursă1) AND (sursă2) → [-/E/R] FLAGS$_{P,S,Z}$***
TEST {reg$_{8,16,32,64}$|mem$_{8,16,32,64}$}, { reg$_{8,16,32,64}$|data imediată$_{8,16,32}$}

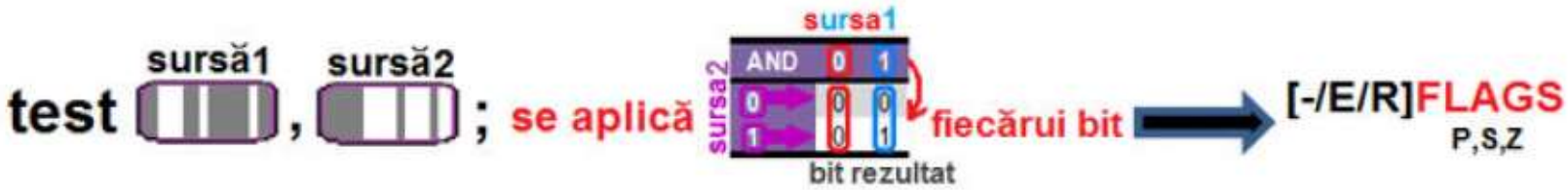

Figura 4-2.1. Ilustrarea modului de operare al instrucţiunii ***TEST***

Observaţii:

- Instrucţiunea TEST *modifică flagurile aritmetice* SF, ZF, PF conform rezultatului operaţiei AND fictive (rezultatul operaţiei e depus într-un registru temporar), dar flagurile OF și CF *sunt zero*, OF=CF=0, iar AF este *nedefinit*;
- Operanzii trebuie să aibă dimensiuni egale; aceştia pot fi atât numere fără semn cât și numere cu semn, dar aici nu se va interpreta valoarea respectivă decât ca o înșiruire de biţi;
- La folosirea regiștrilor pe 64 biţi, valoarea imediată e pe 8 sau 32 biţi și e extinsă cu semn la dimensiunea impusă de celălalt operand, cel destinaţie.

Exemple de *instrucţiuni legale*:

Exemplul 4-2.1

```
mov AX, 9876h
test AX, 1          ; testează bitul LSb, adică b0 dacă este 1, dar acesta nu e 1; astfel, rezultatul este 0 => ZF=1
test AX, 32768      ; testează bitul MSb, adică b15 dacă este 1, iar acesta este 1; astfel, rezultatul este ≠0 => ZF=0
```

Exemplul 4-2.1

```
mov AX, 1234h       ; AX=  1234h = 0001 0010 0011 0100b
mov BX, 0F0Fh       ; BX=EDCBh = 1110 1101 1100 1011b
test AX, BX         ; AX AND BX = 0000h=0000 0000 0000 0000b, deci AX=1234h, BX=EDCBh, SF=0, ZF=1
```

2. Instrucțiuni de deplasare

Operaţiile logice de deplasare şi rotire sunt utile programatorilor în limbaj de asamblare: de exemplu, operaţia de deplasare spre stânga (în binar) cu o poziţie mută/ deplasează fiecare bit din şirul de biţi ce formează numărul cu o poziţie spre stânga, iar rezultatul unei astfel de operaţii este *echivalent cu o înmulţire cu 2 a acelui număr*. În general, dacă se consideră numărul într-o altă bază și prin analogie s-ar muta cifrele spre stânga cu o poziţie, s-ar obţine ca rezultat numărul înmulţit cu acea bază.

Figura 4-3.1. Reprezentarea operaţiilor de deplasare spre stânga și spre dreapta

La o operaţie de deplasare *(logică) spre stânga*, pe locul bitului LSb, adică bitul b0, se va introduce un 0, iar bitul MSb va ajunge în flagul Carry, aşa cum arată Figura 4-3.1 (figura din stânga). Operaţia de deplasare spre stânga cu o poziţie este echivalentă cu înmulţirea valorii cu 2^1.

În general, dacă se consideră numărul într-o altă bază și prin analogie s-ar muta cifrele spre dreapta cu o poziţie, s-ar obţine ca rezultat câtul împărţirii cu acea bază.

O operaţie de deplasare (logică) *spre dreapta* funcţionează în mod asemănător celei spre stânga, doar că datele se deplasează în sens opus, spre dreapta, aşa cum arată Figura 4-3.1 (figura din dreapta).

Există 2 posibilităţi, așa cum reiese și din Figura 4-3.1: pe locul bitului MSb, se poate introduce:

- ori **un 0**, caz în care se spune că s-a realizat o ***deplasare logică spre dreapta***,
- ori **un bit identic cu bitul MSb**, caz în care se spune că s-a realizat o ***deplasare aritmetică spre dreapta***.

În general se folosește mnemonica **SHL** (***shift logic to left***) pentru a desemna o astfel de operație. Instrucțiunea **SAL** (***shift aritmetic to left***) va avea efect identic cu cel obținut prin instrucțiunea SHL, deoarece dinspre bitul 0 se va insera tot 0 (nu ar avea sens să se insereze MSb).

Similar, se folosește mnemonica **SHR** (***shift logic to right***) pentru a desemna o operație de ***deplasare logică*** spre dreapta. Operația de ***deplasare aritmetică*** spre dreapta se poate obține folosind mnemonica **SAR** (***shift aritmetic to right***).

Operația de deplasare spre dreapta rotunjește rezultatul înspre întregul cel mai apropiat, care e mai mic sau egal cu rezultatul. În oricare din cazurile de deplasare spre dreapta, bitul LSb, și anume b0 va ajunge în flagul Carry (CF).

La modul general:

o înmulțire cu 2^n a numărului, înseamnă ***o deplasare spre stânga cu n biți***, iar

o împărțire cu 2^n a numărului, înseamnă ***o deplasare spre dreapta cu n biți***,

și invers, o deplasare spre stânga cu n poziții e echivalent cu o înmulțire cu 2^n, iar

o deplasare spre dreapta cu n poziții e echivalent cu o împărțire cu 2^n.

Există situații când sunt necesare operații de înmulțire/ împărțire (obținute prin deplasare) a numerelor ***fără semn***, iar atunci deplasarea trebuie realizată prin operații de ***deplasare logică***; în situațiile când se dorește deplasarea numerelor ***cu semn***, se vor folosi operații de ***deplasare aritmetică***, întrucât acestea nu vor modifica semnul numerelor, ci doar valoarea lor.

În plus, la deplasarea spre stânga trebuie ținut cont de semnul numărului și de posibilele alterări ale acestuia prin operația de deplasare (pentru a nu obține un rezultat eronat).

Exemple: Numere fără semn: 0011b << 2 = 1100b adică 3 x 4 = 12

Numere cu semn: 1010b >> 1 = 1101b adică – 6 : 2= - 3

În general, instrucțiunea de deplasare sau **shiftare aritmetică** se folosește pentru ***numere cu semn***,
iar instrucțiunea de **shiftare logică** se folosește pentru ***numere fără semn***.

Instrucţiunile de deplasare a biţilor spre stânga sau spre dreapta, logic sau aritmetic (**SHL, SAL, SAL, SAR**) au fost suportate încă de la **8086**↑, cu operand destinaţie de 8 sau 16 biţi, iar operandul folosit ca şi contor putea lua valoarea 1 sau o valoare exprimată în registrul CL.

De la **80286**↑ s-a introdus pentru contor posibilitatea de a fi dată imediată pe 8 biţi, iar de la **80386**↑, dimensiunea operandului a fost extinsă şi la 32 biţi.

Tot de la **80386**↑ s-au adăugat 2 instrucţiuni specifice procesoarelor pe 32 biţi, și anume **SHLD** și **SHRD**. Odată cu apariţia procesoarelor pe 64 biţi, deci de la **Pentium 4**↑ sau **Core 2**↑ aceste instrucţiuni au suportat și operanzi de 64 biţi.

Forma generală a instrucţiunilor SHL, SAL, SAL, SAR

este:

mnemonica destinaţie, contor

mnemonica {reg$_{8,16,32,64}$|mem$_{8,16,32,64}$}, {1| CL|imed$_8$ }

mnemonica={SHL/SAL, SHR,SAR}

unde ***SHL, SHR*** **–>** ***deplasare logică***

SAL, SAR **–>** ***deplasare aritmetică***

Figura 4-3.2. Ilustrarea modului de operare al instrucţiunilor de deplasare

Observaţie

Valoarea din *registrul CL* e mascata cu 1Fh (0001 1111b) la procesoare pe 32 biţi pentru a reduce din timpul maxim de execuţie al instrucţiunii, a.î. **valoarea contorului să fie în gama [0;31];**

În modul pe 64 biţi, valoarea din registrul CL e mascată cu 3Fh a.î. valoarea contorului să fie în gama [0;63].

La procesorul **8086** nu au fost implementate astfel de mascări.

4.3.1. Instrucţiunea SAL/ SHL

Instrucţiunea **SHL / SAL** ***(Shift Logic / Arithmetic Left)*** deplasează logic/ aritmetic la stânga: bitul MSB trece în CF, apoi toţi biţii se deplasează la stânga cu o poziţie (echivalent cu o înmulţire cu doi). Pe poziţia LSB se inserează un 0. Operaţia se repetă de un număr de ori egal cu valoarea din "contor".

Aceste operaţii sunt echivalente cu operaţii de înmulţire:

o deplasare la stânga cu o poziţie e echivalentă cu o înmulţire cu 2^1,
o deplasare cu 2 poziţii e echivalentă cu o înmulţire cu 2^2, și tot așa …

Instrucţiunile **SHL** și **SAL** au același efect, de înmulţire a valorii din operandul destinaţie cu un număr de ori egal cu valoarea din operandul contor. **SHL / SAL** ***(Shift Logic / Arithmetic Left)*** deplasează logic/ aritmetic la stânga: bitul MSB trece în CF, apoi toţi biţii se deplasează la stânga cu o poziţie (echivalent cu o înmulţire cu doi). Pe poziţia LSB se inserează un 0. Operaţia se repetă de un număr de ori egal cu valoarea din "contor".

Dacă valorile stocate sunt cu semn, de câte ori are loc o deplasare și CF e diferit de MSb, se setează OF (s-au pierdut biţi semnificativi, deci se face o atenţionare a acestui fapt).

SAL | SHL destinaţie, contor *; deplasează biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***stânga, inserând 0***
SAL | SHL {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ }

Figura 4-3.2. Ilustrarea modului de operare al instrucţiunilor ***SHL*** și ***SAL***

4.3.2. Instrucţiunea SHR

Instrucţiunea **SHR** ***(Shift Logic Right)*** deplasează logic la dreapta: bitul LSB trece în CF, iar apoi toti biţii se deplasează la dreapta cu o poziţie (sunt echivalente cu o împărţire cu puterile lui 2). Pe poziţia corespunzătoare bitului MSB, se inserează 0. Operaţia se repetă de "contor" ori.

SHR destinaţie, contor *; deplasează biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***dreapta, inserând 0***
SHR {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ }

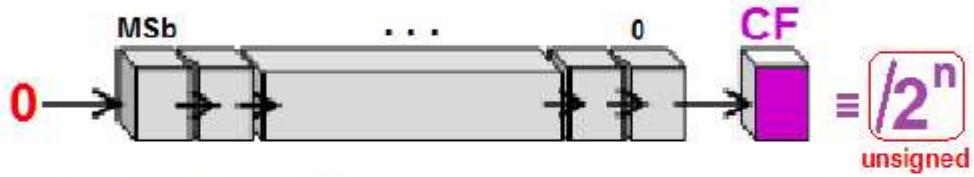

Figura 4-3.3. Ilustrarea modului de operare al instrucţiunii ***SHR***

4.3.3. Instrucţiunea SAR

Instrucţiunea **SAR** ***(Shift Arithmetic Right)*** deplasează aritmetic la dreapta (în CF): diferenţa faţă de SHR este că semnul se păstrează deoarece se completează dinspre stânga cu o valoare a bitului identică valorii MSb (= bitul de semn).

SAR destinaţie, contor *; deplasează biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***dreapta, inserând MSb***
SAR {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ }

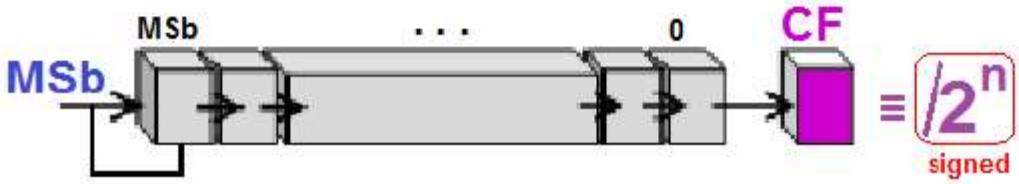

Figura 4-3.4. Ilustrarea modului de operare al instrucţiunii ***SAR***

Exemple de *instrucţiuni ilegale:*

```
shl AX, [SI]        ; al II-lea operand nu poate fi din memorie
shl [DI], CH        ; operandul CH nu e admis ca şi contor; se admite doar reg. CL ca şi contor
shl AX, 1234h       ; operandul imediat 1234h nu e pe 8 biţi
```

(similar, ca la SHL se procedează şi pentru SAL, SHR, SAR)

Exemple de *instrucţiuni legale*:

Exemplul 4-3.1

```
mov AX, 0AAAAh      ; AX=  AAAAh=1010 1010 1010 1010b
shr AX,1            ; AX=  5555h=0101 0101 0101 0101b, CF=0
```

Exemplul 4-3.2

```
mov AX, 0AAAAh      ; AX= AAAAh =1010 1010 1010 1010b
shl AX,1            ; AX= 5554h = 0101 0101 0101 0101b, CF=1
```

Exemplul 4-3.3

```
mov AX, 0AAAAh      ; AX=  AAAAh=1010 1010 1010 1010b
sar AX,1            ; AX=  D555h=1101 0101 0101 0101b, CF=0
```

Exemplul 4-3.8 Împachetarea cifrelor low din registrul AH cu cele similare din registrul AL; fie AH=34h şi AL=37h:

```
and AL, 0Fh         ; AL = 07h
and AH, 0Fh         ; AH = 04h
mov CL, 4           ; CL=contor
shl AH, CL          ; AH = 40h
or AH, AL           ; AH = 47h
```

Exemplul 4-3.9 Înmulţirea numerelor fără semn folosind instrucţiunea shl:

```
mov AX, 23h         ; AX=35
shl AX, 4           ; AX=35*2^4=560
```

Exemplul 4-3.10 Împărţirea numerelor fără semn folosind instrucţiunea shr:

```
mov AX, 2300h       ; AX=8960
shr AX, 5           ; AX=8960/2^5=280
```

Exemplul 4-3.11 Împărţirea numerelor cu semn folosind instrucţiunea sar:

```
mov AX, F300h       ; AX=-3328
sar AX, 5           ; AX=-3328/2^5=-104
```

3. Instructiuni de rotire

Aceste operaţii de rotaţie la nivel de şiruri de biţi se comportă asemănător cu cele de deplasare, cu diferenţa că bitul care iese înafara reprezentării este cel care completează din cealaltă direcţie rezultatul, așa cum arată Figura 4-4.1.

Figura 4-4.1. Reprezentarea operaţiilor de rotaţie spre stânga și spre dreapta

Operaţiile de rotaţie mai au o variantă disponibilă și anume prin implicarea flagului Carry în cadrul operaţiei de rotaţie. Acesta acţionează ca o celulă suplimentară, bitul 0 sau bitul n+1 ca poziţionare, așa cum reiese din Figura 4-4.2.

Figura 4-4.2. Reprezentarea operaţiilor de rotaţie spre stânga și spre dreapta cu CF

Aceste operaţii de rotaţie a şirurilor de biţi, cu sau fără folosirea lui CF în operaţia de rotaţie, pe procesor sunt implementate prin instrucţiunile specifice: **ROL** și **ROR**, respectiv cu implicarea lui Carry Flag: **RCL** și **RCR**.

mnemonica destinaţie, contor *mnemonica* = {**ROL, ROR, RCL, RCR**}

mnemonica $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}$, $\{1 | CL | imediat_8\}$,

Încă de la **80286**↑ s-a introdus pentru contor posibilitatea de a fi dată imediată; de la **80386**↑, dimensiunea operandului a fost extinsă şi la 32 biţi, iar de la **Pentium 4**↑ s-au acceptat și operanzi pe 64 biţi.

Instructiunea STC (Set CF) -> CF=1

Instructiunea CLC (Clear CF)-> CF=0

Instructiunea CMC (complement CF)

Observaţii:

- Cei 2 operanzi la **ROL, ROR, RCL, RCR,** nu trebuie să aibă dimensiuni egale (au semnificaţii diferite).
- Rezultatul rotaţiei se va reflecta în operandul destinaţie (și în CF), destinaţia putând fi un registru de uz general sau o zonă de memorie;

- Flagul CF este afectat de valoarea bitului;
- La operaţiile de rotaţie, flagul OF este definit doar pentru rotaţie cu o poziţie, astfel:
  - pentru rotaţie *spre stânga*, OF va fi definit de operaţia SAU exclusiv între flagul CF (după rotaţie) și MSb al rezultatului;
  - pentru rotaţie *spre dreapta*, flagul OF va fi definit de operaţia SAU exclusiv între cei mai semnificativi 2 biţi ai rezultatului.
- Pentru rotaţie cu mai mult de o poziţie, valoarea lui OF este nedefinită;
- Flagurile SF, ZF și PF nu sunt afectate niciodată;

- Procesorul **8086** *nu folosește mască pentru operandul count*, în schimb
  - procesoarele pe 32 biţi (mai exact de la **286**↑) ***maschează count cu un număr pe 5 biţi***
    (pentru ca numărul maxim acceptat pentru rotaţie să fie 31),
  - procesoarele pe 64 biţi ***maschează count cu un număr pe 6 biţi***
    (pentru ca numărul maxim acceptat pentru rotaţie să fie 63);

4.4.1. Instrucţiunile ROL şi ROR

Instrucţiunea **ROL** ***(Rotate Left)*** roteşte spre ***stânga*** toţi biţii din operandul destinaţie: bitul MSb trece în bitul LSb din operand (dar se va reflecta şi în CF), toţi biţii deplasându-se înspre stânga cu o poziţie. Numărul operaţiilor (rotirilor) este dat de contor. Toţi cei n biţi ai operandului destinaţie îşi schimbă poziţia.

ROL destinaţie, contor *; roteşte biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***stânga, dinspre LSb***
ROL {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ }

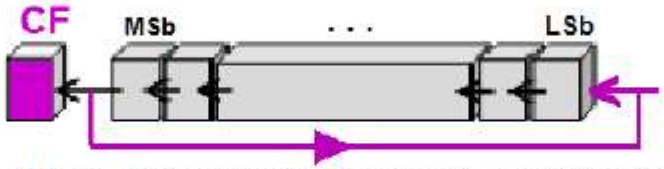

Figura 4-4.3. Ilustrarea modului de operare al instrucţiunii ***ROL***

Instrucţiunea **ROR** ***(Rotate Right)*** roteşte spre ***dreapta*** toţi biţii din operandul destinatie: bitul LSb trece în bitul MSb din operand (dar se va reflecta şi în CF), toţi biţii deplasându-se înspre dreapta cu o poziţie. Numărul de rotiri este dat de contor. Toţi cei n biţi ai operandului destinaţie îşi schimbă poziţia.

ROR destinaţie, contor *; roteşte biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***dreapta, dinspre MSb***
ROR {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ }

Figura 4-4.4. Ilustrarea modului de operare al instrucţiunii ***ROR***

4.4.2. Instrucţiunile RCL şi RCR

Instrucţiunea RCL ***(Rotate Left through Carry)*** roteşte la ***stânga*** prin CF; această instrucţiune seamnănă cu ROL, dar CF participă activ la rotire. În total, n+1 biţi îşi schimbă poziţia (n fiind numărul de biţi al operandului destinaţie). Bitul MSb trece în CF, toţi biţii se deplasează la stânga cu o poziţie, iar CF original trece în bitul LSb. Numărul operaţiilor e dat de *contor*.

RCL destinaţie, contor *; roteşte biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***stânga, dinspre LSb,***
RCL {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ } ***; dar cu participarea lui CF ca celulă suplimentară (înaintea lui LSb)***

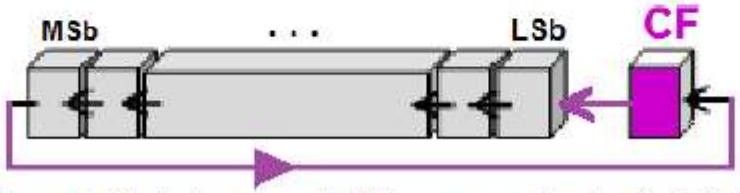

Figura 4-4.5. Ilustrarea modului de operare al instrucţiunii ***RCL***

Instrucţiunea RCR ***(Rotate Right through Carry)*** roteşte la ***dreapta*** prin CF: bitul LSb trece în CF, toţi biţii se deplasează la dreapta cu o poziţie (un număr de n+1 biţi îşi schimbă poziţia, n fiind numărul de biţi al operandului destinaţie), iar CF original trece în bitul MSb. Numărul operaţiilor e dat de *contor*.

RCR destinaţie, contor *; roteşte biţii din* ***destinaţie*** *cu* ***contor poziţii*** *spre* ***dreapta, dinspre MSb,***
RCR {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {1|CL| imed$_8$ } ***; dar cu participarea lui CF ca celulă suplimentară (înaintea lui MSb)***

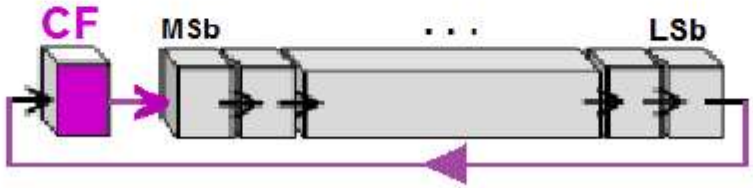

Figura 4-4.6. Ilustrarea modului de operare al instrucţiunii ***RCR***

Exemple de *instrucţiuni ilegale:*

```
rol AX, [SI]        ; al II-lea operand nu poate fi din memorie
roll [DI], CH       ; operandul CH nu e admis ca şi contor; se admite doar reg. CL ca şi contor
roll AX, 1234h      ; operandul imediat 1234h nu e pe 8 biţi
```

(similar, ca la ROL se procedează şi pentru ROR, RCL, RCR)

Exemple de *instrucţiuni legale*:

Exemplul 4-4.1

```
mov AX, 0AAAAh      ; AX=AAAAh=1010 1010 1010 1010b
rol AX,1            ; AX=5555h=0101 0101 0101 0101b, CF=1
```

Exemplul 4-4.2

```
mov AX, 0AAAAh      ; AX=AAAAh=1010 1010 1010 1010b
ror AX,1            ; AX=AAAAh=0101 0101 0101 0101b, CF=0
```

Exemplul 4-4.3

```
clc                 ; şterge flagul CF, adică CF=0
mov AX, 0AAAAh      ; AX=AAAAh=1010 1010 1010 1010b
rcl AX,1            ; AX= 5554h= 0101 0101 0101 0100b, CF=1
```

Exemplul 4-4.4

```
stc                 ; setează flagul CF, adică CF=1
mov AX, 0AAAAh      ; AX= AAAAh=1010 1010 1010 1010b
rcl AX,1            ; AX= 5555h =0101 0101 0101 0101b, CF=1
```

Exemplul 4-4.5

```
clc                 ; şterge flagul CF, adică CF=0
mov AX, 0AAAAh      ; AX= AAAAh=1010 1010 1010 1010
rcr AX,1            ; AX=  5555h =0101 0101 0101 0101b, CF=0
```

Exemplul 4-4.6

```
stc                 ; setează flagul CF, adică CF=1
mov AX, 0AAAAh      ; AX=AAAAh=1010 1010 1010 1010
rcr AX,1            ; AX=D555h =1101 0101 0101 0101b, CF=0
```

4. Instrucțiuni de comparare, salt și buclare

3.4.1. Instrucţiunea CMP

Instrucţiunea **CMP** ***(Compare)*** compară cei doi operanzi sursă menţionaţi explicit în instrucţiune și setează flagurile aritmetice din registrul [-/E/R] FLAGS în concordanţă cu rezultatul unei operaţii fictive de tipul *SUB op1, op2*. Instrucţiunea CMP este asemănătoare cu instrucţiunea SUB, doar că rezultatul operaţiei nu este stocat în destinaţie, ci într-un registru temporar, scăderea fiind una fictivă (se realizează doar pentru a seta flagurile aritmetice).

CMP op1, op ***; op1 ? op2 -> [-/E/R] FLAGS***

CMP $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}, \{reg_{8,16,32,64} | mem_{8,16,32,64} | imed_{8,16,32}\}$

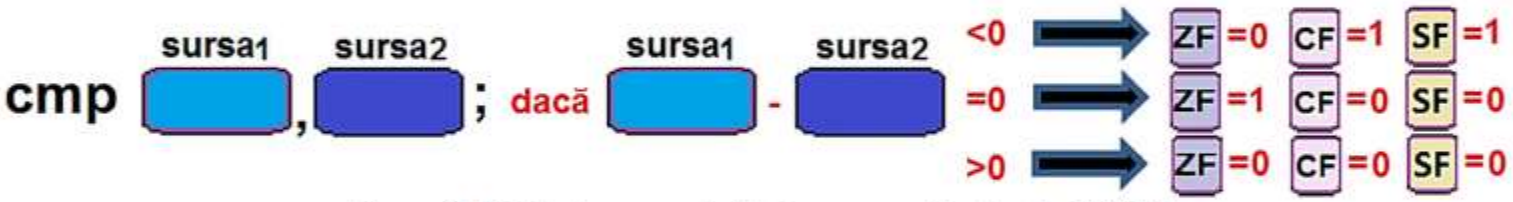

Figura 3-4.1. Ilustrarea modului de operare al instrucţiunii ***CMP***

Instrucţiunea este în general folosită (exact) înaintea unor instrucţiuni de forma Jcc, CMOVcc, SETcc.

Observaţii:

- Instrucţiunea CMP *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, CF, conform rezultatului operaţiei fictive SUB realizată între cei doi operanzi sursă;
- Operanzii trebuie să aibă dimensiuni egale; este interzis ca ambii operanzi să fie locaţii de memorie;
- Operanzii pot fi atât numere fără semn cât și numere cu semn, dar atunci când se folosește un operand imediat, acesta este extins cu semn la dimensiunea operandului destinaţie și apoi se realizează operaţie de scădere.

Instrucţiuni de salt

Locaţia în memorie a următoarei instrucţiuni de executat este dată de perechea de regiştrii CS: (-/E)IP sau RIP (în mod pe 64 biţi). Derularea secvenţială a instrucţiunilor se obţine prin incrementarea registrului (-/E/R) IP în mod automat (utilizatorul nu trebuie să modifice acest registru). Această derulare secvenţială poate fi alterată prin instrucţiuni de salt.

În general, la modurile de lucru pe 16 biţi sau 32 biţi, există următoarele tipuri de salt:
- scurt (SHORT) sau relativ, NEAR *de tip intrasegment*, când saltul se face în interiorul segmentului de cod şi se modifică doar IP, resp. EIP sau
- *de tip intersegment* (FAR), când saltul se face oriunde în memorie şi se modifică atât (-/E)IP cât şi CS.

6.1. Instrucţiuni de salt (ne)condiţionat

Salturile în program pot fi de 2 tipuri:
- *necondiţionate*, când saltul se execută întotdeauna (instrucţiunea JMP).
- *condiţionate*, când sunt în funcţie de valoarea unui anumit bit din PSW sau în funcţie de conţinutul unui registru (de exemplu poate fi verificat registrul CX care e deseori folosit ca un contor);

Instrucţiunea de salt necondiţionat **JMP** determină întotdeauna un salt la eticheta specificată; acel salt este în spaţiul de 1Moctet pentru procesorul 8086 (modul real de funcţionare), 4G octeţi pentru 80386 (mod protejat).

Instrucţiunile de salt condiţionat implică 2 paşi:
(1) prima dată se testează condiţia, iar apoi
(2) A - se efectuează salt dacă acea condiţie este verificată sau
B - se trece la execuţia instrucţiunii următoare dacă acea condiţie este falsă.

Condiţiile de salt sunt determinate aşa cum se poate urmări în Tabelul 6-1.1 de:
starea anumitor flaguri

Tabelul 6-1.1. Instrucţiuni de salt condiţionat

Mnemonică	Condiţie verificată	Interpretare	Mnemonică	Condiţie verificată	Interpretare
JE, JZ	ZF=1	Equal, Zero	**JNE, JNZ**	ZF=0	NotEqual, NotZero
JL, JNGE	SF≠OF	Less, NotGreater or Equal	**JNL, JGE**	SF=OF	NotLess, Greater or Equal
JLE, JNG	SF≠OF sau ZF=1	Less or Equal, NotGreater	**JNLE, JG**	SF=OF şi ZF=0	NotLess or Equal, Greater
JB,JNAE,JC	CF=1	Below, NotAbove or Equal, Carry	**JNB,JAE, JNC**	CF=0	NotBelow, Above or Equal, NotCarry
JBE, JNA	CF=1 sau ZF=1	Below or Equal, NotAbove	**JNBE, JA**	CF=0 şi ZF=0	NotBelow or Equal, Above
JP, JPE	PF=1	Parity, Parity Even	**JNP, JPO**	PF=0	NotParity, Parity Odd
JO	OF=1	Overflow	JNO	OF=0	NotOverflow
JS	SF=1	Sign	**JNS**	SF=0	NotSign
JCXZ JECXZ	(E)CX=0	CX register is 0 ECX register is 0			

Aşa cum s-a văzut în capitolul 3, instrucţiunea cmp doar execută scăderea fictivă şi setează flagurile corespunzător rezultatului scăderii. Instrucţiunile de salt ce urmează apoi realizează interpretarea valorii flagurilor.

La compararea a 2 ***operanzi numere cu semn***, se folosesc termenii **less/ greater (mai mic/ mai mare)** , iar

La compararea a 2 ***operanzi numere fără semn***, se folosesc termenii **below/ above (inferior, sub/ superior, peste)**

```
Less -> pt interpretarea nr ca signed
mov al,70h      ; 70h = 112 = +112
mov bl,81h      ; 81h = 129 = - 127
cmp al,bl
jl et1          ; 70h > 81h
mov al, 0 ; (c0)
jmp et
et1: mov al,1 ; (c1)
et:
; in AL vom avea AL=0 (deci se obt c0)
```

JL => +112 > -127 -> nu face salt la et1 => **AL=0**

(less, greater -> **nr cu semn** !!!)

```
Below – pt interpretarea nr ca unsigned
mov al,70h      ; 70h = 112 = +112
mov bl,81h      ; 81h = 129 = - 127
cmp al,bl
jb et1          ; 70h < 81h
mov al, 0 ; (c0)
jmp et
et1: mov al,1 ; (c1)
et:
; in AL vom avea AL=1 (deci se obt c1)
```

JB => 112 < 129 -> face salt la et1 => **AL=1**

(below, above -> **nr fara semn** !!!)

6.2. Instrucţiuni pentru controlul buclelor de program

În programe apare deseori necesitatea execuţiei unei secvenţe de instrucţiuni, în mod repetat. Secvenţa care se repetă se numeşte buclă (loop) sau iteraţie, instrucţiunile specifice controlului buclelor fiind prezentate în tabelul 6-2.1:

Tabelul 6-2.1. Instrucţiuni pentru controlul buclelor de program

	Mnemonică	LOOP	LOOPE, LOOPZ	LOOPNE, LOOPNZ
16 biţi	**Cum se interpretează**	CX=CX-1 dacă (CX≠0) atunci execută salt altfel, continuă	CX=CX-1 dacă (CX≠0 şi ZF=1) atunci execută salt altfel, continuă	CX=CX-1 dacă (CX≠0 şi ZF=0) atunci execută salt altfel, continuă

Observaţii:

- ⇨ Trebuie acordată atenţie sporită la valorile CX/ECX/RCX la intrarea în buclă, pentru a evita buclarea de 2^{16}, 2^{32}, resp. 2^{63} ori sau o eventuală buclare infinită (şi nedorită).
- ⇨ Cele 2 de mai jos sunt echivalente semantic, dar nu au acelaşi efect ! (Instrucţiunea DEC afectează flagurile O,Z,S,P, dar LOOP nu le afectează.)

LOOP eti

dec (-/E/R)CX
jnz eti

Mov CX, 5 ; nr de repetari

eti: ;instr1
; instr2
;instr3
LOOP eti ; CX=CX-1, ???(CX=0) daca inca nu este 0, sare la eti
; instr 4

Exercitii rezolvate:

Ex5-1. Să se precizeze rezultatul următoarelor instrucţiuni:
and AX, BX; ***or AX, BX***; ***xor AX, BX***; ***not AX***; dacă AX=C875h, BX = 9A6Dh.
Răspuns:
Mai întâi, se transformă valorile în binar şi se obţine:

AX = 1100.1000.0111.0101b;
BX = 1001.1010.0110.1101b.

a) Pentru operaţia ***AND***, rezultă AX = 1000.1000.0110.0101b = 8865h
b) Pentru operaţia ***OR***, rezultă AX = 1101.1010.0111.1101b = DA7Dh
c) Pentru operaţia ***XOR***, rezultă AX = 0101.0010.0001.1000b = 5218h
d) Pentru operaţia ***NOT***, rezultă AX = 0011.0111.1000.1010b = 378Ah

Ex5-2. Să se precizeze rezultatul următoarelor instrucţiuni:
shl AX,2; ***shr AX,2***; ***sal AX,3***; ***sar AX,3***; dacă AX = A57Dh.
Răspuns:
Se transformă valoarea din AX în binar: AX = 1010.0101.0111.1101b.

a) Instrucţiunea ***shl AX,2*** va da rezultatul:
AX = 1001.0101.1111.01**00**b = 95F4h. În *Carry flag* (CF) va intra al doilea c.m.s. bit (din stânga) al valorii iniţiale a lui AX (**0**), bitul b14 iniţial.
b) Instrucţiunea ***shr AX,2*** va da rezultatul:
AX = **00**10.1001.0101.1111b = 295Fh. În *Carry flag* (CF) va intra al doilea c.m.p.s. bit (din dreapta) al valorii iniţiale a lui AX (**0**), bitul b1 iniţial.
c) Instrucţiunea ***sal AX,3*** va da rezultatul:
AX = 0010.1011.1110.1**000**b = 2BE8h. În *Carry flag* (CF) va intra al treilea c.m.s. bit (din stânga) al valorii iniţiale a lui AX (**1**), bitul b13 iniţial.
d) Instrucţiunea ***sar AX,3*** va da rezultatul:
AX = **111**1.0100.1010.1111b= F4AFh. În *Carry flag* (CF) va intra al treilea c.m.p.s. bit (din dreapta) al valorii iniţiale a lui AX (**1**), bitul b2 iniţial.

Observaţie: În cazul instrucţiunilor de deplasare, deşi flagul OF (*overflow*) se calculează după formula cunoscută, acesta nu are o semnificaţie aparte.

Ex5-3. Să se precizeze rezultatul următoarelor instrucţiuni:
rol AX,2; ***ror AX,2***; ***rcl AX,3***; ***rcr AX,3***; dacă AX = A57Dh.
Răspuns:
Se transformă valoarea din AX în binar: AX = 1010.0101.0111.1101b.

a) Instrucţiunea ***rol AX,2*** va da rezultatul:
AX = 1001.0101.1111.0110b = 95F6h. În *Carry flag* (CF) va intra al doilea c.m.s. bit (din stânga) al valorii iniţiale a lui AX (0).
b) Instrucţiunea ***ror AX,2*** va da rezultatul:
AX = 0110.1001.0101.1111b = 695Fh. În *Carry flag* (CF) va intra al doilea c.m.p.s. bit (din dreapta) al valorii iniţiale a lui AX (0).
c) Instrucţiunea **rcl AX,3** va da rezultatul:
AX = 0010.1011.1110.1**0**10b = 2BEAh, dacă valoarea iniţială a lui CF era **0**.
AX = 0010.1011.1110.1**1**10b = 2BEEh, dacă valoarea iniţială a lui CF era **1**.
În *Carry flag* (CF) va intra al treilea c.m.s. bit al valorii iniţiale a lui AX (1).

d) Instrucţiunea **rcr AX,3** va da rezultatul:
AX = 01**0**1.0100.1010.1111b= 54AFh, dacă valoarea iniţială a lui CF era **0**.
AX = 01**1**1.0100.1010.1111b= 54AFh, dacă valoarea iniţială a lui CF era **1**.
În *Carry flag* (CF) va intra al treilea c.m.p.s. bit al valorii iniţiale a lui AX (1).

Observaţie: În cazul instrucţiunilor de rotire, deşi flagul OF (*overflow*) se calculează după formula cunoscută, acesta nu se interpretează.

Ex5-4. Să se calculeze expresia: | AL – BL | (valoarea absolută a diferenţei AL-BL). Să se considere AL=7Fh, iar BL=80h.
Răspuns: Expresia se poate calcula folosind instrucţiuni de comparare şi salt. Secvenţa de program diferă în funcţie de modul în care se consideră valorile din regiştrii AL şi BL, adică numere ***cu semn*** sau ***fără semn***. În fiecare caz, se va executa instrucţiunea scrisă îngroşat:

a) valori ***cu semn***, folosind instr. **jg**

```
      cmp AL, BL   ; 7Fh > 80h
      jg et1
      sub BL, AL  ; dacă AL <= BL
      neg BL
      jmp comun
et1:  sub AL, BL  ; dacă AL > BL
      neg AL
      comun: .........  ; 7Fh-80h=-1
```

b) valori ***fără semn***, folosind instr. **ja**

```
      cmp AL, BL    ; 7Fh < 80h
      ja et1
      sub BL, AL ; dacă AL <= BL
      jmp comun
et1:  sub AL, BL  ; dacă AL > BL
      comun: ......... ; 80h-7Fh=1
```

c) valori ***cu semn***, folosind instr. **jl**

```
      cmp AL, BL   ; 7Fh > 80h
      jl et1
      sub AL, BL ;dacă AL >= BL
      neg AL
      jmp comun
et1:  sub BL, AL ;dacă AL < BL
      neg BL
      comun: .........  ; 7Fh-80h=-1
```

d) valori ***fără semn***, folosind instr. **jb**

```
      cmp AL, BL   ; 7Fh < 80h
      jb et1
      sub AL, BL ;dacă AL >= BL
      jmp comun
et1:  sub BL, AL ;dacă AL < BL
      comun: ......... ; 80h-7Fh=1
```

Ex5-5. Afişaţi un şir pe ecran urmărind specificaţiile:
a) Se dă un text de 20 de caractere (valori Ascii) începând de la offsetul 102h, în segm. de date. **Să se afişeze textul pe ecran, fol.afişarea individuală a valorilor (întreruperea int 10h, cu serv. 0Eh, AL=codul Ascii al caracterului).**
Răspuns:

```
mov SI, 102h    ; se încarcă în SI offsetul de început al şirului de caractere
mov CX, 20      ; se încarcă în CX numărul de caractere de afişat
start:  mov AL, [SI]     ; se încarcă în AL caracterul curent din segm. de date
        mov AH, 0Eh      ; codul funcţiei de afişare
        mov BH,0         ; prima pagină a ecranului
        int 10h          ; afişarea caracterului curent
        inc SI           ; se trece la următorul caracter de afişat
loop start               ; bucla se repetă (de 20 de ori)
```

b) Se dă un şir de 15 octeţi reprezentând cifre zecimale (cu valori între 0..9) începând de la offsetul 102h, în segmentul de date. **Să se afişeze valorile pe ecran, folosind afişarea individuală a valorilor (int 10h, serv. 0Eh, AL=cod Ascii caracter). Nr 6 -> la afisare '6'=36h**

Răspuns:

```
mov SI, 102h        ; se încarcă în SI offsetul de început al şirului de numere
mov CX, 15          ; se încarcă în CX numărul de caractere de afişat
start:  mov AL, [SI]  ; se încarcă în AL cifra curentă din segmentul de date
        add AL, 30h   ; conversie zecimal -> ASCII
        mov AH, 0Eh   ; codul funcţiei de afişare
        mov BH,0      ; prima pagină a ecranului
        int 10h       ; afişarea numărului curent
        inc SI        ; se trece la următorul număr din şir
loop start            ; bucla se repetă (de 15 de ori)
```

Probleme rezolvate:

PR5-1. Să se scrie o secvenţă de instrucţiuni prin care să se verifice dacă ultima cifră hexazecimală a numărului aflat în registrul AL este 6; ?AL=A6h.

Observaţie: la folosirea instrucţiunii AND, operandul destinaţie se suprascrie.

Rezolvare:

```
; p5_01        ; se presupune că AL=36h
mov BL, AL     ; asigurăm o copie întrucât valoarea din AL se va suprascrie
and AL, 0Fh    ; AL =0000 0110b deci s-a păstrat doar cifra hexa de rang 0,
               ; adică în AL avem valoarea 6
cmp AL, 6      ; ZF va fi setat: ZF=1 dacă cifra respectivă este 6
```

PR5-2. Să se scrie o secvenţă de instrucţiuni prin care să se verifice dacă bitul de pe poziţia 5 din registrul AL este 0.

Rezolvare:

```
; p5_02        ; de exemplu, în AL avem AL = 0110 0011b şi vrem să verificăm dacă bitul de pe poziţia 5 este 0
mov BL, AL     ; asigurăm o copie întrucât valoarea din AL se va suprascrie
and AL, 00100000b  ; AL =0010 0000b - s-a păstrat doar cifra binară de rang 5
cmp AL, 0      ; ZF nu va fi setat: ZF=0, întrucât cele 2 valori care se compară
               ; sunt diferite; bitul 5 e 1, nu 0
```

PR5-3. Să se scrie o secvenţă de instrucţiuni prin care să se copieze în registrul AL cifra hexa de rang 3 urmată de cea de rang 0 a numărului aflat în registrul BX.

Exemplu: dacă BX= **98 76** h, în AL să se obţină: AL=**96**h.

Observaţie: la folosirea instrucţiunii OR, operandul destinaţie se suprascrie.

Rezolvare:

```
xor AL, AL     ; p5_03     ; zerorizăm AL, întrucât vom lucra cu instrucţ. OR
mov CX, BX     ; asigurăm o copie: valoarea din BX se va suprascrie
```

```
and BH,0F0h   ; vom reţine doar partea corespunzătoare cifrei hexa de ordin 3      ; BH va fi 90h -> or 00h, 90h => AL=90h
or AL, BH     ; AL=90h, s-a impus cifra din BH deoarece biţii corespunzători
              ; din AL erau 0
and BL,0Fh    ; vom reţine doar partea corespunzătoare cifrei hexa de ordin 0      ; BL va fi 06h -> or 90h,06h => AL=96h
or AL, BL     ; AL=96h, s-a impus cifra din BL deoarece biţii corespunzători
              ; din AL erau 0
```

PR5-4. Se dă un octet în memorie. Să se obţină cifrele componente ale lui şi să se depună în memorie în alte 2 variabile de tip octet (despachetare).
Exemplu: dacă val=47h, va rezulta cifra1=04h, cifra2=07h

Rezolvare:

```
org 100h        ; p5_04
.data
val db 47h          ; valoarea iniţială
cifra1 db ?         ; se alocă spaţiu pt prima cifră
cifra2 db ?         ; se alocă spaţiu pt cea de-a doua cifră
.code
mov AL, val         ; AL = 47h
mov BL, AL          ; BL = 47h
and AL, 0Fh         ; AL = 07h
and BL, 0F0h        ; BL = 40h
mov CL, 4
shr BL, CL          ; BL = 04h
mov cifra1, BL      ; cifra1= 04h
mov cifra2, AL      ; cifra2= 07h
ret
```

PR5-5. Se dă o variabilă în memorie de tip word. a) Să se inverseze poziţia octeţilor în cuvânt; b) Să se inverseze ordinea tuturor cifrelor hexa.
Exemplu: dacă val=1234h, atunci se va obţine: a) 3412h, b) 4321h

Rezolvare:

```
a) org 100h     ; p5_05
.data
val dw 1234h        ; valoarea iniţială
.code
mov AX, val         ; AX = 1234h
mov CL, 8           ; CL = 8
rol AX, CL          ; AX = 3412h
b) se adaugă în plus instrucţiunile:
ror AL,4            ; AL = 21h
ror AH,4            ; AH = 43h
ret                 ; deci AX = 4321h
```

PR5-6. Să se scrie o secvenţă de instrucţiuni prin care să se forţeze în 1 biţii de pe poziţiile 6-3 din registrul AL.

Rezolvare:

; se presupune că AL=56h=**0**101 0**110**b şi se doreşte ca AL= **0**111 1**110**b

Problema se va rezolva în 3 moduri:

a) Folosind operaţia SAU logic

```
; p5_06_a
mov AL, 56h
mov BL, 0111 1000b
or AL, BL
```

b) Cu poziţionarea grupului respectiv în dreapta, umplerea din dreapta cu biţi de 1 şi repoziţionare corectă

```
; p5_06_b
mov AL,56h ;      01010110
ror AL,3 ;      AL=11001010b
shr AL,4 ;      AL=00001100b
mov CX, 4
eti:   stc     ; CF=1
       rCl AL, 1
loop eti   ;AL=11001111b
rol AL, 3 ;AL=01111110b
```

c) Cu poziţionarea grupului respectiv în stânga, umplerea din stânga cu biţi de 1 şi repoziţionare corectă

```
; p5_06_c
mov AL, 56h; 01010110
rol AL, 1 ;AL=10101100b
shl AL, 4 ;AL=11000000b
mov CX, 4
eti:   stc
       rcr AL, 1
loop eti  ;AL=11111100b
ror AL, 1 ;AL=01111110b
```

PR5-7. Fie 2 valori numere fără semn (considerate la nivel de octet) în memorie, aflate în segmentul adresat de DS la offset 0110h și 0111h. Să se aplice o operaţie OR logic, respectiv o operaţie AND logic asupra celor 2 valori cu numărul pe 8 biţi aflat tot în memorie, dar la adresa 0112h. Rezultatele finale se vor stoca tot în memorie, la adresele 0200h şi 0201h.

Rezolvare:

```
org 100h          ; p5_07
.data
.code
mov AL, [0110h]       ; în AL va fi octetul din mem. de la offset 0110h
or AL, [0112h]        ; se realiz. SAU logic cu octetul de la offset 0112h
mov [0200h], AL       ; rezultatul se depune în mem. la offset 0200h
mov AL, [0111h]       ; în AL va fi octetul din mem. de la offset 0111h
and AL, [0112h]       ; se realiz. SAU logic cu octetul de la offset 0112h
mov [0201h], AL       ; rezultatul se depune în mem. la offset 0201h
ret
```

PR5-8. Se dau 8 valori pe cuvânt, stocate în memorie la adrese succesive, începând de la offset 0102h. Să se realizeze o operaţie logică între aceste valori şi un anumit număr a.î. să se păstreze doar biţii de 1 aflaţi pe o poziţie pară; rezultatele finale obţinute se vor depune într-o altă zonă din memorie, începând cu adresa 0202h, iar zona iniţială din memorie nu se va altera.

Rezolvare:

Pentru a păstra doar biţii de pe poziţie pară, trebuie aplicată o mască de forma 0x0x...0x0xb pe 16 biţi sau AAAAh. Operaţia logică pt mascare este SI logic.

```
org 100h          ; p5_08
.data
```

```
.code
mov AX, 0
mov DI, 0                ; index folosit la adresarea valorilor din mem.
mov CX, 8                ; nr de repetări ale buclei
eti: mov AX, 0102h [DI] ; AX = cuvântul din mem. indexat cu 0, apoi 2, ...
     mov BX, AX          ; copie a lui AX
     and BX, 0AAAAh      ; se maschează (resetează) biţii de pe poz. impară
     mov 0202h [DI], BX ; se depune rezultatul în zona din mem. coresp.
   add DI, 2             ; se actualizează indexul pentru următorul element
loop eti                 ; asigură reluarea buclei de 8 ori
ret
```

PR5-9. a) Să se scrie o secvenţă de program care să implementeze operaţia de înmulţire cu 4 la nivel de octet, dar fără a folosi instrucţiunile *(i)mul* sau *add*; se vor folosi doar operaţii pe biţi şi se va considera că numerele sunt în convenţia de reprezentare *fără semn*.
Exemplu: n=5 şi se doreşte obţinerea lui 5*4 =20 în variabila *rez*.
b) Propuneţi o metodă de a afla numărul maxim (putere a lui 2) cu care se poate realiza înmulţirea.

Rezolvare:

a) Pentru a implementa operaţia de înmulţire fără semn, echivalentă cu instrucţiunea ***mul***, se va folosi instrucţiunea ***shl***: o înmulţire cu 2^m este echivalentă cu deplasarea spre stânga a numărului cu m poziţii.

```
.data            ; p5_09
n db 5
rez dw ?
.code
mov AX,0
mov AL, n                ; AL = 5 = 00000101b
shl AX, 2                ; implementarea operaţiei de înmulţire cu 2^2 prin shl
mov rez, AX              ; rez = 00010100b = 20
ret
```

b) Se poate găsi poziţia bitului c.m.s. setat din registrul AL şi se poate realiza calculul: *15-acea poziţie*. Numărul maxim cu care se poate înmulţi numărul este $2^{15\text{-acea poziţie}}$.

PR5-10. Să se scrie o secvenţă de program care să implementeze operaţia de împărţire cu 4 (culegând doar câtul operaţiei), dar fără a folosi instrucţiunile *(i)div* sau *sub*; astfel, se vor folosi doar operaţii pe biţi. Se va considera că se folosesc numere *fără semn*, respectiv *cu semn*.
Exemplu: n=768=300h şi se doreşte obţinerea lui 768/4 =192 în variabila *rez* pentru numere *fără semn*, respectiv n= -768 şi se doreşte obţinerea lui -768/4 = -192 în variabila *rez* pentru numere *cu semn*.
b) Propuneţi o metodă de a afla numărul minim (putere a lui 2) cu care trebuie să se realizeze împărţirea a.î. *rez* să se poată scrie doar pe octet.

Rezolvare:

a) Pentru a implementa operaţia de împărţire fără semn, echivalentă cu instrucţiunea ***div***, se va folosi instrucţiunea ***shr***, iar pentru a implementa operaţia de împărţire cu semn, echivalentă cu instrucţiunea ***idiv***, se va folosi instrucţiunea ***sar***; o împărţire cu 2^m este echivalentă cu deplasarea spre dreapta a numărului cu m poziţii, deplasare care să păstreze bitul de semn în cazul nr. cu semn.

```
.data     ; p5_10_a
n dw 768
rez dw ?
.code
mov AX,n  ; AX=768=0300h
shr AX, 2 ; implementarea operaţiei
          ; de împărţire cu 2^2 prin shr
mov rez, AX   ; AX=00C0h=192
ret
```

```
.data    ; p5_10_b
n dw -768
rez dw ?
.code
mov AX,n  ; AX= -768=FD00h
sar AX, 2 ; implementarea operaţiei
          ; de împărţire cu 2^2 prin sar
mov rez, AX   ; AX=FF40= -192
ret
```

b) Se poate găsi poziţia bitului c.m.s. setat din registrul AH şi se poate realiza calculul: *1+acea poziţie*. Numărul minim cu care se poate împărţi numărul n pentru a se putea scrie rezultatul pe doar 8 biţi este $2^{1+\text{acea poziţie}}$, în acest caz 2^3.

PR5-11. Se dă un octet A definit în memorie. Să se obţină cuvântul B, a.î. biţii 13-9 ai lui să fie identici cu biţii 5-1 ai octetului A, iar ceilalţi biţi să fie 0.

Rezolvare:

```
org 100h        ; p5_11
.data
A db 56h        ; A = 56h = 01010110b
B dw ?          ; B neiniţializat la început, dar va fi 0001.0110.0000.0000b
.code
mov AX, 0               ; registru intermediar pt formarea cuvântului B
mov AL, A               ; AL = 01010110b
mov BL, 00111110b       ; masca pt izolarea biţilor 5-1
and AL, BL              ; AL = 00010110b
shl AX, 8       ; deplasare spre stânga cu 8, pt ca b1 să ajungă pe poziţia b9
mov B, AX       ; B = 0001.0110.0000.0000b = 1600h
ret
```

PR5-12. Se dă un octet A definit în memorie.

a) Să se obţină numărul întreg n reprezentat de biţii 5-2 ai lui A.

b) Să se obţină apoi în variabila B produsul dintre A şi 2^n (fără a folosi instrucţiuni aritmetice).

Rezolvare:

```
a) org 100h     ; p5_12
.data
A db 64h        ; A = 64h = 01100100b
n db ?          ; n neiniţializat la început, dar va fi 1001b=9
.code
mov AL, A               ; AL = 01100100b
mov BL, 00111100b       ; masca
and AL, BL              ; AL = 00100100b
shr AL, 2       ; deplasare spre dreapta cu 2, pt ca b2 să ajungă pe poziţia b0
mov n, AL       ; n = AL = 00001001b = 9
```

b) În segmentul de date se mai adaugă directiva:
b dw ?
iar zona de cod se va completa cu următoarea secvenţă:

```
mov AX, 0
mov AL, A       ; AL = 64h = 100
mov CL, n       ; CL = 9
shl AX, CL      ; AX = C800h = 512000 (adică 100*2^9)
```

PR5-13. a) Scrieţi următoarea secvenţă în simulator și urmăriţi valoarea din registrul BX pe măsură ce executaţi bucla. De câte ori se execută această buclă?
b) Modificaţi secvenţa astfel încât să se execute de 6 ori. Ce se va găsi în AX?

```
          mov AX,0
          mov BX,2
 Aduna:  add AX,BX
           inc BX                ; BX = ___; ___; ___; ___; ___; ___;
jmp Aduna
```

Rezolvare:

a) *; p5_13*

```
Aduna:  add AX,BX               ; AX = 2; 5; 9; 14; 20; 27;
          inc BX                ; BX = 3; 4; 5;   6;  7;   8;
jmp Aduna
```

b) În locul instrucţiunii de salt *jmp* se va pune *loop* şi pentru ca loop să funcţioneze, e nevoie ca CX să fie înscris cu nr dorit de repetări; astfel, instrucţiunea *mov CX,6* va trebui plasată înainte de eticheta *Aduna*.