4.1.2. **Instrucţiuni de conversie** (distructivă)

CBW	conversie octet conţinut în AL la cuvânt în AX (extensie de semn)	-
CWD	conversie cuvânt conţinut în AX la dublu cuvânt în DX:AX (extensie de semn)	-
CWDE	conversie cuvânt din AX în dublucuvânt în EAX (extensie de semn)	-
MOVZX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), continutul lui s fară semn	-
MOVSX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), continutul lui s cu semn	-

Instrucţiunea **CBW** converteşte octetul cu semn din AL în cuvântul cu semn AX (extinde bitul de semn al octetului din AL la nivelul cuvântului din AX, modificând distructiv conţinutul registrului AH). De exemplu,

```
mov al, -1      ; AL = 0FFh
cbw             ;extinde valoarea octet -1 din AL în valoarea cuvânt -1 din AX (0FFFFh).
```

Analog, pentru conversia cu semn cuvânt - dublu cuvânt, instrucţiunea **CWD** extinde cuvântul cu semn din AX în dublucuvântul cu semn DX:AX. Exemplu:

```
mov ax,-10000       ; AX = 0D8F0h
cwd         ;obţine valoarea -10000 în DX:AX  (DX = 0FFFFh ; AX = 0D8F0h)
cwde        ;obţine valoarea -10000 în EAX     (EAX = 0FFFFD8F0h)
```

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

De ce coexistă CWD cu CWDE ? CWD trebuie să rămână din raţiuni de backwards compatibility şi din raţiuni de funcţionalitate a intrucţiunilor (I)MUL şi (I)DIV.

```
MOV ah, 0c8h
MOVSX ebx, ah       ; EBX = FFFFFFC8h       MOVSX ax,[v]  ; MOVSX ax, byte ptr DS:[offset v]
MOVZX edx, ah       ; EDX =  000000C8h      MOVZX eax, [v] ; syntax error – op.size not specified
```

Atenţie ! NU sunt acceptate sintactic:

```
CBD                              CWDE  EBX, BX          MOVSX  EAX, [v]
CWB                              CWD  EDX,AX            MOVZX  EAX, [EBX]
CDW                              MOVZX  AX, BX          MOVSX  dword [EBX], AH
CDB !!! (super-înghesuire!! ☺)   MOVSX  EAX, -1         CBW  BL
```

4.1.3. Impactul reprezentării little-endian asupra accesării datelor (pag.119 – 122 – curs).

Dacă programatorul utilizează datele consistent cu dimensiunea de reprezentare stabilită la definire (ex: accesarea octeţilor drept octeţi şi nu drept secvenţe de octeţi interpretate ca şi cuvinte sau dublucuvinte, accesarea de cuvinte ca şi cuvinte şi nu ca perechi de octeţi, accesarea de dublucuvinte ca şi dublucuvinte şi nu ca secvenţe de octeţi sau de cuvinte) atunci instrucţiunile limbajului de asamblare vor ţine cont în mod AUTOMAT de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiţie programatorul nu trebuie să intervină suplimentar în nici un fel pentru a asigura corectitudinea accesării şi manipulării datelor utilizate. Exemplu:

```
a   db   'd', -25, 120
b   dw     -15642, 2ba5h
c    dd   12345678h
…
mov  al, [a]      ;se încarcă în AL codul ASCII al caracterului 'd'
```

mov bx, [b] ;se încarcă în BX valoarea -15642; ordinea octeţilor în BX va fi însă inversată faţă de reprezentarea în memorie, deoarece numai reprezentarea *în memorie* foloseşte reprezentarea *little-endian*! În regiştri datele sunt memorate conform reprezentării structurale normale, echivalente unei reprezentări *big endian*.

mov edx, [c] ;se încarcă în EDX valoarea dublucuvânt 12345678h

Dacă însă se doreşte accesarea sau interpretarea datelor sub o formă diferită faţă de modalitatea de definire atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să îşi asume însă întreaga responsabilitate a interpretării şi accesării corecte a datelor. In astfel de situaţii programatorul este obligat să conştientizeze particularităţile de reprezentare little-endian (ordinea de plasare a octeţilor în memorie) şi să utilizeze modalităţi de accesare a datelor în conformitate cu aceasta **Ex pag.120-122.**

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

    c db -1

segment code

mov al, byte [a+1] ;accesarea lui a drept octet, efectuarea calculului de adresă a+1, selectarea octetului de la
                   adresa a+1 (octetul de valoare 12h) şi transferul său în registrul AL.

    mov dx, word [b+2]    ;dx:=1122h

    mov dx, word [a+4]    ;dx:=1122h  deoarece b+2 = a+4 , în sensul că aceste expresii de tip pointer
                          desemnează aceeaşi adresă şi anume adresa octetului 22h.

    mov dx, [a+4]         ;această instrucţiune este echivalentă cu cea de mai sus, nefiind realmente necesară
                          ;utilizarea operatorului de conversie WORD
```

```
mov bx,  [b]          ;bx:=3344h

mov bx,  [a+2]        ;bx:=3344h, deoarece ca adrese b = a+2.

mov ecx, dword [a]    ;ecx:=33441234h, deoarece dublucuvântul ce începe la adresa a este format din octeţii
                      34h  12h  44h  33h  care (datorită reprezentării little-endian) înseamnă de fapt
                      ;dublucuvântul 33441234h.

mov ebx, [b]              ; ebx := 11223344h

mov ax, word [a+1]        ; ax := 4412h

mov eax, word [a+1]       ; ax := 22334412h

mov dx, [c-2]             ; DX := 1122h deoarece c-2 = b+2 = a+4
mov bh, [b]               ;bh := 44h
mov ch, [b-1]             ;ch := 12h
mov cx, [b+3]             ; CX := 0FF11h
```

4.2. OPERAŢII

4.2.1. Operaţii aritmetice

Operanzii sunt reprezentaţi în cod complementar (vezi 1.5.2.). Microprocesorul realizează adunările şi scăderile "văzând" doar configuraţii de biţi şi nu numere cu semn sau fără. Regulile de efectuare a adunării şi scăderii presupun adunarea de configuraţii binare, fără a fi nevoie de a interpreta operanzii drept cu semn sau fără semn anterior efectuării operaţiei! Deci, la nivelul acestor instrucţiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucţiuni separate pentru adunarea/scăderea cu semn faţă de adunarea/scăderea fără semn.

Adunarea şi scăderea se efectuează întotdeauna la fel (adunând sau scăzând configuraţii binare) indiferent de semnul (interpretarea) acestor configuraţii! După cum vom vedea acest lucru nu este valabil şi pentru înmulţire şi împărţire. În cazul acestor operaţii trebuie să ştim apriori dacă operanzii vor fi interpretaţi drept cu semn sau fără semn. De exemplu, fie doi operanzi A şi B reprezentaţi fiecare pe câte un octet:

A = 9Ch = 10011100b (= 156 în interpretarea fără semn şi -100 în interpretarea cu semn)
B = 4Ah = 01001010b (= 74 atât în interpretarea fără semn cât şi în interpretarea cu semn)

Microprocesorul realizează adunarea C = A + B şi obţine

C = E6h = 11100110b (= 230 în interpretarea fără semn şi -26 în interpretarea cu semn)

Se observă deci că simpla adunare a configuraţiilor de biţi (fără a ne fixa neapărat asupra unei interpretări anume la momentul efectuării adunării) asigură corectitudinea rezultatului obţinut, atât în interpretarea cu semn cât şi în cea fără semn.

INSTRUCŢIUNI ARITMETICE – tabel pag.123 (curs)

4.2.1.3. Exemple şi exerciţii propuse – pag.129-130 (curs)

4.2.2. Operaţii logice pe biţi(AND, OR, XOR şi **NOT)**.
Instrucţiunea AND este indicată pentru izolarea unui anumit bit sau pentru forţarea anumitor biţi la valoarea 0.
Instrucţiunea OR este indicata pentru forţarea anumitor biţi la valoarea 1.
Instrucţiunea XOR este indicată pentru schimbarea valorii unor biţi din 0 în 1 sau din 1 în 0.

4.2.3. Deplasări şi rotiri de biţi
Instrucţiunile de *deplasare* de biţi se clasifică în:

- Instrucţiuni de deplasare logică
  - stânga - **SHL**
  - dreapta - **SHR**
- Instrucţiuni de deplasare aritmetică
  - stânga - **SAL**
  - dreapta - **SAR**

Instrucţiunile de *rotire* a biţilor în cadrul unui operand se clasifică în:

- Instrucţiuni de rotire fără carry
  - stânga - **ROL**
  - dreapta - **ROR**
- Instrucţiuni de rotire cu carry
  - stânga - **RCL**
  - dreapta - **RCR**

Pentru a defini deplasările şi rotirile să considerăm ca şi configuraţie iniţială un octet X = abcdefgh, unde a-h sunt cifre binare, h este cifra binară de rang 0, a este cifra binară de rang 7, iar k este valoarea existentă în CF (CF=k). Atunci avem:

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

INSTRUCŢIUNILE DE DEPLASARE ŞI ROTIRE DE BIŢI – tabel – pag.134 (curs)

4.3. RAMIFICĂRI, SALTURI, CICLURI

4.3.1. <u>Saltul necondiţionat</u>

În această categorie intră instrucţiunile JMP (echivalentul instrucţiunii GOTO din alte limbaje), CALL (apelul de procedură înseamnă transferul controlului din punctul apelului la prima instrucţiune din procedura apelată) şi RET (transfer control la prima instrucţiune executabilă de după CALL).

JMP *operand*	Salt necondiţionat la adresa determinată de operand	-
CALL *operand*	Transferă controlul procedurii determinată de operand	-
RET [*n*]	Transferă controlul instrucţiunii de după CALL	-

4.3.1.1. Instrucţiunea JMP

Instrucţiunea de salt necondiţionat **JMP** are sintaxa

JMP *operand*

unde *operand* este o etichetă, un registru sau o variabilă de memorie ce conţine o adresă. Efectul ei este transferul necondiţionat al controlului la instrucţiunea ce urmează etichetei, la adresa dată de valoarea registrului sau constantei, respectiv la adresa conţinută în variabila de memorie. De exemplu, după execuţia secvenţei

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

registrul AX va conţine valoarea 3. Instrucţiunile **inc** şi **jmp** dintre etichetele *AdunaUnu* şi *AdunaDoi* nu se vor executa, decât dacă se va face salt la *AdunaUnu* de altundeva din program.

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

```
(1)   mov  eax, etich

      jmp  eax  ;operand registru
                ; jmp [eax] ?
      etich:    . . .
```

```
(2)   segment data
      Salt   DD   Dest ;Salt := offset Dest
                .    .    .
      segment cod
                .    .    .
      jmp  [Salt]      ;salt NEAR
      .      .;operand variabilă de memorie
      Dest :     . . .
```

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

```
                b   resd  1
(1')            . . .
                mov  [b], DWORD etich    ; b := offset etich
                jmp [b]    ; salt NEAR – operand variabilă de memorie JMP DWORD PTR DS:[offset_b]
```

Exemplul 4.3.1.2. – pag.142-143 (curs) – modul de transfer al controlului la o eticheta.

4.3.2. Instrucţiuni de salt condiţionat

4.3.2.1. Comparaţii între operanzi

CMP *d,s*	comparaţie valori operanzi (nu modifică operanzii) (execuţie fictivă *d* - *s*)	OF,SF,ZF,AF,PF şi CF
TEST *d,s*	execuţie fictivă *d* **AND** *s*	OF = 0, CF = 0 SF,ZF,PF - modificaţi, AF - nedefinit

Instrucţiunile de salt condiţionat se folosesc de obicei în combinaţie cu instrucţiuni de comparare. De aceea, semnificaţiile instrucţiunilor de salt rezultă din semnificaţia operanzilor unei instrucţiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucţiune CMP este de multe ori necesară determinarea relaţiei de ordine dintre două valori. De exemplu, se pune întrebarea: numărul 11111111b (= FFh = 255 = -1) este mai mare decât 00000000b(= 0h = 0)? Răspunsul poate fi şi da şi nu! Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 şi este evident mai mare decât 0. Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 şi este mai mic decât 0.

Instrucţiunea CMP nu face distincţie între cele două situaţii, deoarece aşa după cum am precizat şi în 4.2.1.1. adunarea şi scăderea se efectuează întotdeauna la fel (adunând sau scăzând configuraţii binare) indiferent de semnul (interpretarea) acestor configuraţii. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn ***operanzii*** scăderii fictive *d-s*, ci ***rezultatul*** final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparaţiei revine diverselor instrucţiuni de salt condiţionat prezentate în 4.3.2.2.

4.3.2.2. Salturi condiţionate de flaguri

În tabelul 4.1. (pag.146 – curs) se prezintă instrucţiunile de salt condiţionat împreună cu semnificaţia lor şi cu precizarea valorilor flagurilor în urma cărora se execută salturile respective. Precizăm că pentru toate instrucţiunile de salt sintaxa este aceeaşi, şi anume

<instrucţiune_de_salt> etichetă

Semnificaţia instrucţiunilor de salt condiţionat este dată sub forma ***"salt dacă operand1 <<relaţie>> faţă de operand2"*** (unde cei doi operanzi sunt obiectul unei instrucţiuni anterioare CMP sau SUB) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă şi din condiţiile ce trebuie verificate, instrucţiunile ce se află într-o aceeaşi linie a tabelului sunt echivalente.

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

4.3.2.3. <u>Exemple comentate..... pag.148-162 (curs).</u>

- discutie si analiza comparativa a conceptelor de : reprezentari cu semn vs. fara semn, depasire, modul de actiune al instructiunilor de salt conditionat.

4.3.3. <u>Instrucţiuni de ciclare (pag.162 – 164 – curs).</u>

Ele sunt: **LOOP, LOOPE, LOOPNE** şi **JECXZ**. Sintaxa lor este

<instrucţiune> etichetă

Instrucţiunea **LOOP** comandă reluarea execuţiei blocului de instrucţiuni ce începe la *etichetă*, atâta timp cât valoarea din registrul ECX este diferită de 0. **Se efectuează întâi decrementarea registrului ECX şi apoi se face testul şi eventual saltul.** Saltul este "scurt" (max. 127 octeţi - atenţie deci la "distanţa" dintre LOOP şi etichetă!).

În cazul în care condiţiile de terminare a ciclului sunt mai complexe se pot folosi instrucţiunile **LOOPE** şi **LOOPNE**. Instrucţiunea **LOOPE** (***LOOP*** *while* ***E****qual*) diferă faţă de LOOP prin condiţia de terminare, ciclul terminându-se fie dacă ECX=0, fie dacă ZF=0. În cazul instrucţiunii **LOOPNE** (***LOOP*** *while Not* ***E****qual*) ciclul se va termina fie dacă ECX=0, fie dacă ZF=1. Chiar dacă ieşirea din ciclu se face pe baza valorii din ZF, decrementarea lui ECX are oricum loc. **LOOPE** mai este cunoscută şi sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută şi sub numele de **LOOPNZ**. Se folosesc de obicei precedate de o instrucţiune CMP sau SUB.

JECXZ (***J****ump if* ***ECX*** *is* ***Z****ero*) realizează saltul la eticheta operand numai dacă ECX=0, fiind utilă în situaţia în care se doreşte testarea valorii din ECX înaintea intrării într-o buclă. In exemplul următor instrucţiunea JECXZ se foloseşte pentru a se evita intrarea în ciclu dacă ECX=0:

. . .

```
    jecxz MaiDeparte        ;dacă ECX=0 se sare peste buclă
Bucla:
    Mov   BYTE [esi],0      ;iniţializarea octetului curent
    inc   esi               ;trecere la octetul următor
    loop  Bucla             ;reluare ciclu sau terminare
MaiDeparte:    .     .     .
```

La întâlnirea instrucţiunii LOOP cu ECX=0, ECX este decrementat, obţinându-se valoarea 0FFFFh (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în ECX, adică de încă 65535 ori!

Este important să precizăm aici faptul că nici una dintre instrucţiunile de ciclare prezentate nu afectează flag-urile.

```
                        dec  ecx
loop Bucla      şi      jnz  Bucla
```

deşi semantic echivalente, nu au exact acelaşi efect, deoarece spre deosebire de LOOP, instrucţiunea DEC afectează indicatorii OF, ZF, SF şi PF.

4.3.4. Instrucţiunile CALL şi RET

Apelul unei proceduri se face cu ajutorul instrucţiunii **CALL**, acesta putând fi *apel direct* sau *apel indirect*. Apelul direct are sintaxa **CALL** *operand*

Asemănător instrucţiunii JMP şi instrucţiunea **CALL** transferă controlul la adresa desemnată de operand. În plus faţă de aceasta, înainte de a face saltul, instrucţiunea CALL salvează în stivă adresa următoarei instrucţiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalenţa

```
CALL operand            push A
A: . . .        ⇔       jmp  operand
```

Terminarea execuţiei secvenţei apelate este marcată de întâlnirea unei instrucţiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucţiunea de la această adresă. Sintaxa instrucţiunii RET este **RET** [*n*] unde *n* este un parametru opţional. El indică eliberarea din stivă a *n* octeţi aflaţi sub adresa de revenire.

Instrucţiunea RET poate fi ilustrată prin echivalenţa

```
                    B  dd   ?
RET n               .    .     .
(revenire near) ⇔   pop  [B]
                    add  esp,[n]
                    jmp  [B]
```

De cele mai multe ori, după cum este şi natural, instrucţiunile CALL şi RET apar în următorul context

```
    etichetă_procedură:
.     .     .
      ret  n
.     .     .
CALL  etichetă_procedură
```

Instrucţiunea CALL poate de asemenea prelua adresa de transfer dintr-un registru sau dintr-o variabilă de memorie. Un asemenea gen de apel este denumit *apel indirect*. Exemple:

```
call  ebx       ;adresă preluată din registru
call  [vptr]    ;adresă preluată din memorie (similar cu apelul funcţiei printf)
```

Rezumând, operandul destinaţie al unei instrucţiuni CALL poate fi:

- numele unei proceduri
- numele unui registru în care se află o adresă
- o adresă de memorie