

4.1.2. Instrucțiuni de conversie (distructivă)

CBW	conversie octet conținut în AL la cuvânt în AX (extensie de semn)	-
CWD	conversie cuvânt conținut 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!), conținutul lui s fără semn (zero extension)	-
MOVSX d, s	încarcă în d (REGISTRU !), de dimensiune mai mare decât s (registru sau operand din memorie!), conținutul lui s cu semn (sign extension) http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm !!	-

Instrucțiunea **CBW** convertește 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 conținutul 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, instrucțiunea **CWD** extinde cuvântul cu semn din AX în dublucuvântul cu semn DX:AX. Exemplu:

```
mov ax,-10000   ; AX = 0D8F0h
cwd             ;obține valoarea -10000 în DX:AX (DX = 0FFFFh ; AX = 0D8F0h)
cwde           ;obține 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 obține prin aplicarea instr. MOVZX)

De ce coexistă CWD cu CWDE ? CWD trebuie să rămână din rațiuni de backwards compatibility și din rațiuni de funcționalitate a instrucțiunilor (I)MUL și (I)DIV.

MOV ah, 0c8h

MOVSX ebx, ah ; EBX = FFFFFFFC8h

MOVZX edx, ah ; EDX = 000000C8h

MOVSX ax,[v] ; MOVSX ax, byte ptr DS:[offset v]

MOVZX eax, [v] ; syntax error – op.size not specified

Atenție ! NU sunt acceptate sintactic:

CBD

CWB

CDW

CDB !!! (super-înghesuire!! ☺)

CWDE EBX, BX

CWD EDX, AX

MOVZX AX, BX

MOVSX EAX, -1

MOVSX EAX, [v]

MOVZX EAX, [EBX]

MOVSX dword [EBX], AH

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 octeților drept octeți și nu drept secvențe de octeți interpretate ca și cuvinte sau dublucuvinte, accesarea de cuvinte ca și cuvinte și nu ca perechi de octeți, accesarea de dublucuvinte ca și dublucuvinte și nu ca secvențe de octeți sau de cuvinte) atunci instrucțiunile limbajului de asamblare vor ține cont în mod AUTOMAT de modalitatea de reprezentare little-endian. Ca urmare, dacă se respectă această condiție programatorul nu trebuie să intervină suplimentar în nici un fel pentru a asigura corectitudinea accesării și manipulării datelor utilizate. Exemplu:

a db 'd', -25, 120

b dw -15642, 2ba5h

c dd 12345678h

...

mov al, [a] ;se încarcă în AL codul ASCII al caracterului 'd'

mov bx, [b] ;se încarcă în BX valoarea -15642; ordinea octeților în BX va fi însă inversată față de reprezentarea în memorie, deoarece numai reprezentarea *în memorie* folosește reprezentarea *little-endian*! În regiștri 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 dorește accesarea sau interpretarea datelor sub o formă diferită față de modalitatea de definire atunci trebuie utilizate conversii explicite de tip. În momentul utilizării conversiilor explicite de tip programatorul trebuie să își asume însă întreaga responsabilitate a interpretării și accesării corecte a datelor. În astfel de situații programatorul este obligat să conștientizeze particularitățile de reprezentare little-endian (ordinea de plasare a octeților în memorie) și să utilizeze modalități de accesare a datelor în conformitate cu aceasta **Ex pag.120-122.**

segment data

a dw 1234h ;datorită reprezentării little-endian, în memorie octeții sunt plasați 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) și 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 aceeași adresă și anume adresa octetului 22h.

mov dx, [a+4] ;această instrucțiune 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 octeții
                      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. OPERAȚII

4.2.1. Operații aritmetice

Operanzii sunt reprezentați în cod complementar (vezi 1.5.2.). Microprocesorul realizează adunările și scăderile "văzând" doar configurații de biți și nu numere cu semn sau fără. Regulile de efectuare a adunării și scăderii presupun adunarea de configurații binare, fără a fi nevoie de a interpreta operanzii drept cu semn sau fără semn anterior efectuării operației! Deci, la nivelul acestor instrucțiuni, interpretarea "cu semn" sau "fără semn" rămâne la latitudinea programatorului, nefiind nevoie de instrucțiuni separate pentru adunarea/scăderea cu semn față de adunarea/scăderea fără semn.

Adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații! După cum vom vedea acest lucru nu este valabil și pentru înmulțire și

împărțire. În cazul acestor operații trebuie să știm apriori dacă operanzii vor fi interpretați drept cu semn sau fără semn. De exemplu, fie doi operanzi A și B reprezentați fiecare pe câte un octet:

$A = 9\text{Ch} = 10011100\text{b}$ (= 156 în interpretarea fără semn și -100 în interpretarea cu semn)

$B = 4\text{Ah} = 01001010\text{b}$ (= 74 atât în interpretarea fără semn cât și în interpretarea cu semn)

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

$C = \text{E6h} = 11100110\text{b}$ (= 230 în interpretarea fără semn și -26 în interpretarea cu semn)

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

INSTRUCȚIUNI ARITMETICE – tabel pag.123 (curs)

4.2.1.3. Exemple și exerciții propuse – pag.129-130 (curs)

4.2.2. Operații logice pe biți(AND, OR, XOR și NOT) – pag.131 (curs)

Instrucțiunea AND este indicată pentru izolarea unui anumit bit sau pentru forțarea anumitor biți la valoarea 0.

Instrucțiunea OR este indicată pentru forțarea anumitor biți la valoarea 1.

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

Instrucțiunea NOT este indicată pentru complementarea conținutului operandului (reg/mem).

4.2.3. Deplasări și rotiri de biți

Instrucțiunile de *deplasare* de biți se clasifică în:

- Instrucțiuni de deplasare logică
 - stânga - **SHL**
 - dreapta - **SHR**

- Instrucțiuni de deplasare aritmetică
 - stânga - **SAL**
 - dreapta - **SAR**

Instrucțiunile de *rotire* a biților în cadrul unui operand se clasifică în:

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

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

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

INSTRUCȚIUNILE DE DEPLASARE ȘI ROTIRE DE BIȚI – tabel – pag.134 (curs)

4.3. RAMIFICĂRI, SALTURI, CICLURI

4.3.1. Saltul necondiționat

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

JMP <i>operand</i>	Salt necondiționat la adresa determinată de operand	-
CALL <i>operand</i>	Transferă controlul procedurii determinată de operand	-
RET [<i>n</i>]	Transferă controlul instrucțiunii de după CALL	-

4.3.1.1. Instrucțiunea JMP

Instrucțiunea de salt necondiționat **JMP** are sintaxa

JMP *operand*

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

```

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

```

registru AX va conține valoarea 3. Instrucțiunile **inc** și **jmp** dintre etichetele *AdunaUnu* și *AdunaDoi* nu se vor executa, decât dacă se va face salt la *AdunaUnu* de altundeva din program.

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

<pre> (1) mov eax, etich jmp eax ;operand registru ; jmp [eax] ? etich: . . . </pre>	<pre> (2) segment data Salt DD Dest ;Salt := offset Dest . . . segment cod . . . jmp [Salt] ;salt NEAR . ;operand variabilă de memorie Dest : . . . </pre>
--	---

Dacă în cazul (1) dorim înlocuirea operandului destinație registru cu un operand destinație variabilă de memorie, o soluție 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. Instrucțiuni de salt condiționat

4.3.2.1. Comparații între operanzi

CMP <i>d,s</i>	comparație valori operanzi (nu modifică operanzii) (execuție fictivă <i>d - s</i>)	OF,SF,ZF,AF,PF și CF
TEST <i>d,s</i>	execuție fictivă <i>d AND s</i>	OF = 0, CF = 0 SF,ZF,PF - modificați, AF - nedefinit

Instrucțiunile de salt condiționat se folosesc de obicei în combinație cu instrucțiuni de comparare. De aceea, semnificațiile instrucțiunilor de salt rezultă din semnificația operanzilor unei instrucțiuni de comparare. În afara testului de egalitate pe care îl poate efectua o instrucțiune CMP este de multe ori necesară determinarea relației de ordine dintre două valori. De exemplu, se pune întrebarea: numărul 11111111b (= FFh = 255 = -1) este mai mare decât 00000000b (= 0h = 0)? Răspunsul poate fi și da și nu! Dacă cele două numere sunt considerate fără semn, atunci primul are valoarea 255 și este evident mai mare decât 0. Dacă însă cele două numere sunt considerate cu semn, atunci primul are valoarea -1 și este mai mic decât 0.

Instrucțiunea CMP nu face distincție între cele două situații, deoarece așa după cum am precizat și în 4.2.1.1. adunarea și scăderea se efectuează întotdeauna la fel (adunând sau scăzând configurații binare) indiferent de semnul (interpretarea) acestor configurații. Ca urmare, nu este vorba de a interpreta cu semn sau fără semn *operanții* scăderii fictive *d-s*, ci **rezultatul** final al acesteia! Rolul de a interpreta în mod diferit (cu semn sau fără semn) rezultatul final al comparației revine diverselor instrucțiuni de salt condiționat prezentate în 4.3.2.2.

4.3.2.2. Salturi condiționate de flaguri

În tabelul 4.1. (pag.146 – curs) se prezintă instrucțiunile de salt condiționat împreună cu semnificația lor și cu precizarea valorilor flagurilor în urma cărora se execută salturile respective. Precizăm că pentru toate instrucțiunile de salt sintaxa este aceeași, și anume

<instrucțiune_de_salt> etichetă

Semnificația instrucțiunilor de salt condiționat este dată sub forma "***salt dacă operand1 <<relație>> față de operand2***" (unde cei doi operanzi sunt obiectul unei instrucțiuni anterioare CMP sau SUB) sau referitor la valoarea concretă setată pentru un anumit flag. După cum se observă și din condițiile ce trebuie verificate, instrucțiunile ce se află într-o aceeași linie a tabelului sunt echivalente.

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

4.3.2.3. Exemple comentate..... pag.148-162 (curs).

- 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. Instructiuni de ciclare (pag.162 – 164 – curs).

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

<instructiune> etichetă

Instructiunea **LOOP** comandă reluarea execuției blocului de instrucțiuni ce începe la *etichetă*, atâta timp cât valoarea din registrul ECX este diferită de 0. **Se efectuează întâi decrementarea registrului ECX și apoi se face testul și eventual saltul.** Saltul este "scurt" (max. 127 octeți - atenție deci la "distanța" dintre LOOP și etichetă!). (short jump is out of range!)

În cazul în care condițiile de terminare a ciclului sunt mai complexe se pot folosi instrucțiunile **LOOPE** și **LOOPNE**. Instructiunea **LOOPE** (*LOOP while Equal*) diferă față de LOOP prin condiția de terminare, ciclul terminându-se fie dacă ECX=0, fie dacă ZF=0. În cazul instrucțiunii **LOOPNE** (*LOOP while Not Equal*) ciclul se va termina fie dacă ECX=0, fie dacă ZF=1. Chiar dacă ieșirea din ciclu se face pe baza valorii din ZF, decrementarea lui ECX are oricum loc. **LOOPE** mai este cunoscută și sub numele de **LOOPZ** iar **LOOPNE** mai este cunoscută și sub numele de **LOOPNZ**. Se folosesc de obicei precedate de o instrucțiune CMP sau SUB.

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

. . .

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

La întâlnirea instrucțiunii LOOP cu ECX=0, ECX este decrementat, obținându-se valoarea 0FFFFh (= -1, deci o valoare diferită de 0), ciclul reluându-se până când se va ajunge la valoarea 0 în ECX, adică de încă 4.294.967.296 ori!

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

		dec ecx
loop Bucla	și	jnz Bucla

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

4.3.4. Instrucțiunile CALL și RET

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

CALL *operand*

Asemănător instrucțiunii JMP și instrucțiunea **CALL** transferă controlul la adresa desemnată de operand. În plus față de aceasta, înainte de a face saltul, instrucțiunea CALL salvează în stivă adresa următoarei instrucțiuni de după CALL (adresa de revenire). Cu alte cuvinte, avem echivalența

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

Terminarea execuției secvenței apelate este marcată de întâlnirea unei instrucțiuni **RET**. Aceasta preia din stivă adresa de revenire depusă acolo de CALL, predând controlul la instrucțiunea de la această adresă. Sintaxa instrucțiunii RET este **RET** [*n*] unde *n* este un parametru opțional. El indică eliberarea din stivă a *n* octeți aflați sub adresa de revenire.

Instrucțiunea RET poate fi ilustrată prin echivalența

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

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

etichetă_procedură:

. . .
ret n

. . .
CALL *etichetă_procedură*

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

call ebx	;adresă preluată din registru
call [vptr]	;adresă preluată din memorie (similar cu apelul funcției <i>printf</i>)

Rezumând, operandul destinație al unei instrucțiuni CALL poate fi:

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