ARHITECTURA PROCESOARELOR X86.

Setul de instrucțiuni generale

2016

ANCA APĂTEAN

UT Press
Cluj-Napoca, 2016
ISBN 978-606-737-217-5

Editura U.T.PRESS
Str.Observatorului nr. 34
C.P.42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia materialului:
Camelia Chira, Şef lucrări - UTCN

ISBN 978-606-737-217-5

Prefaţă

Această carte reprezintă un ghid ce se adresează în mod special studenților de la facultăţile de profil Informatică sau Tehnologia Informaţiilor, dar poate fi un instrument util și altor categorii de persoane; a fost conceput pentru cei care doresc să se acomodeze cu programarea în limbaj de asamblare şi deci cunoaşterea instrucţiunilor generale suportate de procesoare x86.

Scopul principal al acestui material îl constituie susținerea procesului de învățare activă, prin studierea și apoi exersarea de exemple (rezolvarea individuală) de exerciţii sau probleme (cu instrucţiunile propuse aici).

Materialul a fost structurat în 7 capitole, acoperind o varietate largă de exemple atât pentru procesoare de 16 biţi cât şi pentru procesoare pe 32 biţi. Acolo unde am considerat necesar, am prezentat şi exemple pentru procesoare pe 64 biţi.

Execuţia programelor sau secvenţelor de instrucţiuni prezentate aici poate fi realizată în mai multe moduri: folosind un simulator de procesor (de exemplu EMU8086), folosind pachetul TASM sub DosBox sau un alt asamblor pe 32 sau chiar 64 biţi (MASM de exemplu), sau folosind Visual C şi asamblare inline.

Cu toate eforturile pe care le-am depus în vederea organizării și prezentării materialului, cu siguranţă că sunt multe îmbunătăţiri care se pot propune; de aceea, aștept cu interes orice propunere, corectură sau pur și simplu comentariu în vederea îmbunătăţirii acestui material la adresa anca.apatean@gmail.com. Pe site-ul meu, la adresa http://users.utcluj.ro/~apateana/ la secţiunea materiale didactice mai puteţi consulta şi alte materiale didactice.

Anca Apătean

"Nimic nu poate înlocui lipsa iubirii,
dar iubirea înlocuieşte toate neajunsurile"
Părintele Arsenie Boca

Dedic acest material mamei mele.

Cuprins

Capitolul 1. Setul de instrucţiuni al procesoarelor din familia x86

1.1. Privire de ansamblu asupra setului de instrucţiuni ISA (Instruction Set Architecture) x86

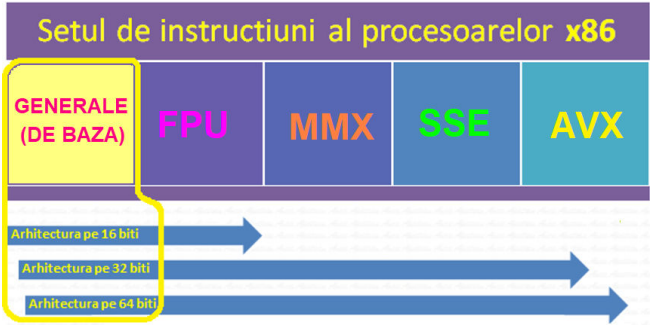
Procesoarele pe 16b şi apoi cele pe 32 biţi au format așa numita familie x86 – cele pe 64 biţi care au urmat - cum ar veni ***x86 pe 64 biţi*** a primit numele de ***x64***; totuşi, în domeniul academic-educaţional încă se foloseşte termenul „x86 pe 64 biţi” pentru a fi lucrurile mai clare celor care întâlnesc astfel de noţiuni pentru prima dată.

Regiştri cu regim special se consideră: regiştrii segment, [R/E]FLAGS şi [R/E]IP. Registrul [-/E/R] FLAGS este un registru de 16, 32 respectiv 64 biţi în care microprocesoarele din familia x86 păstrează starea curentă a procesorului. Regiştrii mai mari păstrează compatibilitatea cu predecesorii lor mai mici; biţii 15…0 ai registrului EFLAGS sunt de fapt biţii registrului FLAGS; similar, biţii 31…0 ai RFLAGS sunt de fapt biţii registrului EFLAGS.
Partea superioară a registrului RFLAGS (în mod pe 64 biţi) este rezervată, iar partea inferioară este identică cu EFLAGS.

Figura 1-1.1. Ilustrarea conţinutului registrului [-/E/R] FLAGS

În modul pe 64 biţi, dacă se folosesc regiştrii R8,...,R15, aceştia pot fi adresaţi la nivel de **byte**, **word**, **doubleword** sau **quadword**, dar doar c.m.p.s. **8**, **16** sau **32** biţi pot constitui un **B**, un **W**, respectiv un **D**; de exemplu, octetul format de biţii b15...b8 nu poate fi accesat direct, așa cum se putea face în cazul procesorului **8086** la *adresarea în mod real* (când se putea accesa și doar porţiunea AH din cadrul registrului AX);

Procesoarele x86, respectiv x64 conţin un set special de regiştri, care nu prea sunt folosiţi în programele uzuale; aceștia sunt regiştri de control (notaţi cu CR0,...,CR15), de depanare (DR0,...,DR7) și de test (TR0,...,TR7), mulţi dintre aceștia fiind rezervaţi. Dimensiunea acestor regiştri se consideră de 32 biţi când se lucrează în mod pe 32 biţi, respectiv de 64 biţi când se lucrează în mod pe 64 biţi. Acești regiştri nu vor fi abordaţi în exemplele prezentate aici. La arhitectura pe 64 biţi a fost introdusă şi noţiunea de ***double quadword*** ca o structură de 128 biţi.



Instrucțiunile ***SIMD*** (Single Instruction Multiple Data) pot creşte foarte mult performanța CPU la execuţie, aplicațiile tipice fiind procesarea semnalelor digitale și prelucrarea grafică (multimedia). Această categorie SIMD pentru procesoare din familia x86 şi x64 a fost divizată în ***mai multe seturi de instrucţiuni***, precum: MMX (1996), 3DNow! (1998), SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2006), SSE4 (2006), SSE5 (2007), AVX (2008), F16C (2009), XOP (2009), FMA4 (2011), FMA3 (2012), AVX2 (2013) AVX-512 (2015), unele fiind comune atât la procesoare Intel cât şi la AMD, altele fiind specifice doar la unul din cei doi producători (de exemplu 3DNow!, SSE5, F16C, XOP, apar la procesoare AMD). [2]

Primul pas în adăugarea acestor seturi de instrucţiuni pe CPU l-a realizat Intel prin introducerea în arhitectura IA-32 a setului MMX (multimedia extension). ***Setul MMX*** a fost însă ulterior perceput ca având două probleme: 1) refolosirea regiştrilor existenţi la acel moment pe CPU (regiştrii FPU), acest lucru împiedicând CPU să lucreze în acelaşi timp şi cu date FP şi cu date SIMD; 2) faptul că a funcţionat doar pentru numere întregi. Astfel, în scurt timp, a apărut ***setul de instrucţiuni SSE*** cu suport pentru execuţia instrucţiunilor cu date FP pe un nou set de regiştri, independent: setul de regiştri **XMM**. În plus, setul SSE a mai adăugat şi câteva instrucţiuni cu numere întregi care foloseau regiştri MMX. SSE a continuat apoi cu SSE2, SSE3, SSSE3 (Supplemental SSE3) şi SSE4. Intel SSE4 s-a constituit dintr-un număr de 54 instrucţiuni, fiind compus de fapt din 2 subseturi: subsetul SSE4.1 (adică 47 instrucţiuni) şi subsetul SSE4.2 (încă 7 instrucţiuni, disponibile prima dată în procesoare core i7, microarhitectură Nehalem). În mod diferit faţă de iteraţiile precedente ale SSE, setul SSE4 conţinea instrucţiuni pentru execuţia de operaţii care nu erau neapărat specifice aplicaţiilor multimedia, unele considerând registrul XMM0 ca operand implicit.

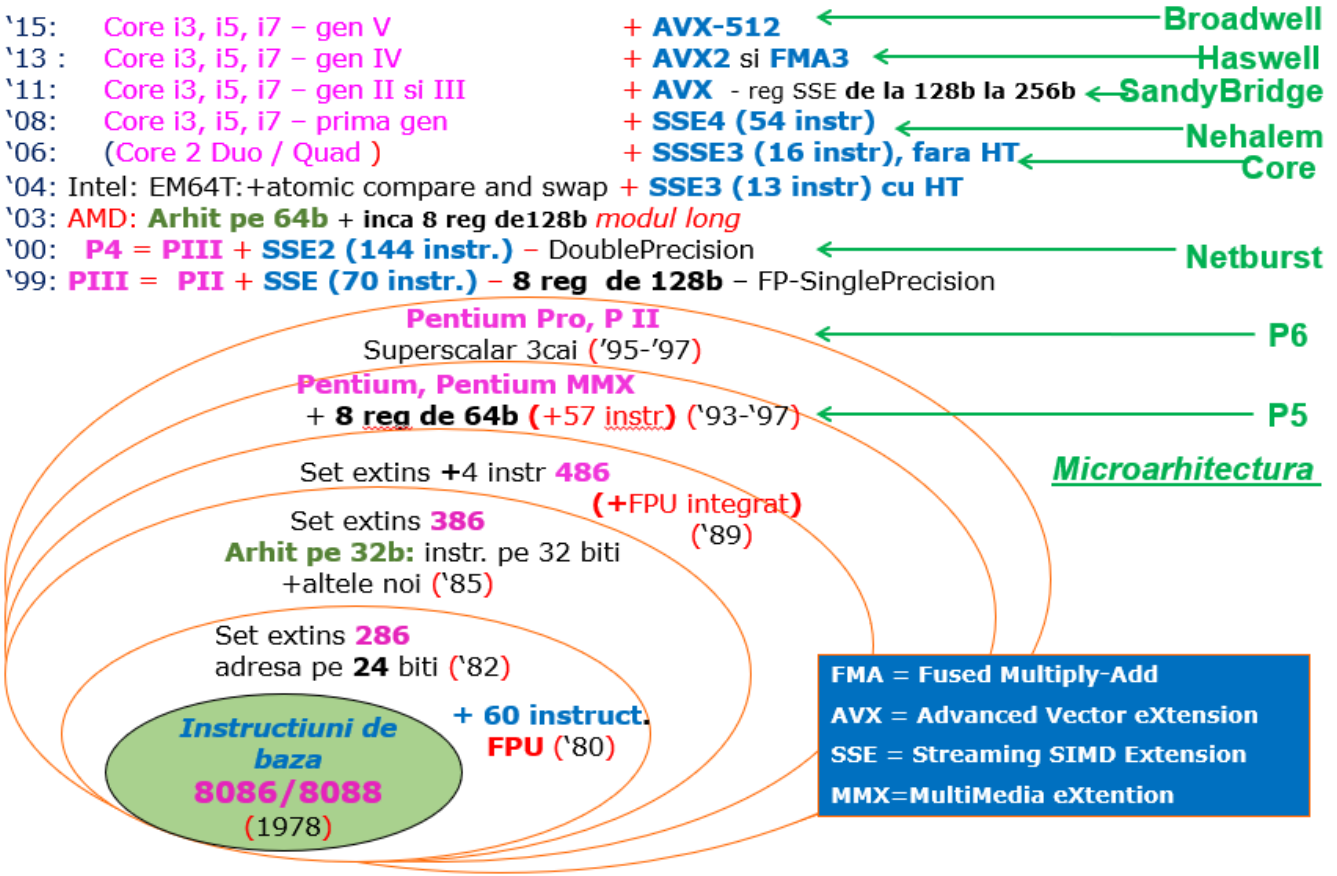


Figura 1-1.2. Evoluţia familiei x86: tipuri de microarhitectură şi seturi de instrucţiuni implementate

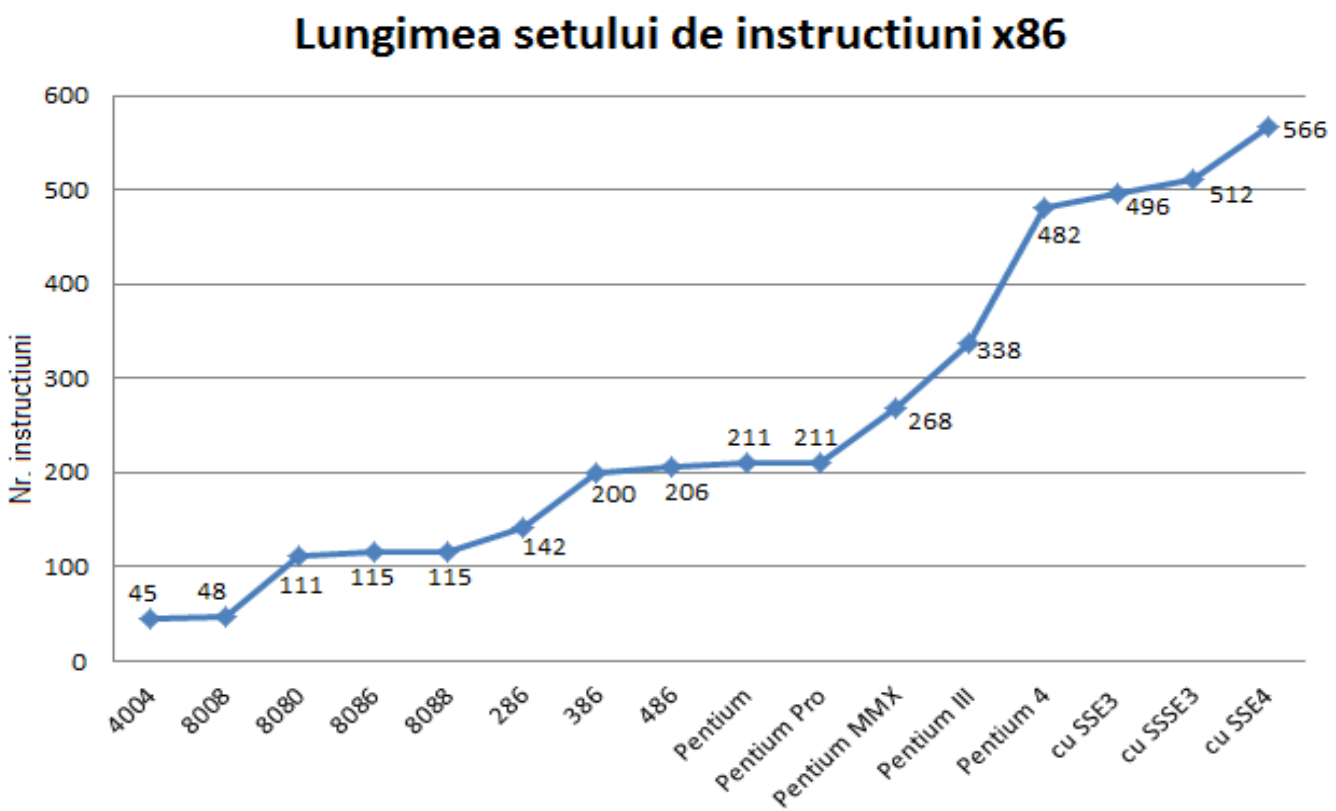


Figura 1-1.3. Numărul de instrucţiuni suportate de un processor x86

Tabel 1.1 Setul de instrucţiuni generale al procesoarelor x86 și x64

Instrucţiuni			Observaţii
de transfer (Cap.2)	**generale**	**MOV**, MOVSX[D], MOVZX, **XCHG**	MOVSX, MOVZX - 80386↑, MOVSXD-P4↑
	condiţionate	CMOVcc, XADD	CMOVcc – Pentium Pro↑
	pt conversie format	BSWAP, MOVBE	BSWAP - 80486↑
	cu stiva	**PUSH**,**POP**,PUSHA,POPA,PUSHAD, POPAD	PUSHA, POPA - 80186↑, PUSHAD, POPAD - 80386↑
	cu acumulatorul	**XLAT**, **IN**, **OUT**	
	pt adrese	**LEA**, **LDS**, **LES**, LFS, LGS, LSS	LFS, LGS, LSS - 80386↑
	pt flaguri	**LAHF**,**SAHF**,**PUSHF**,**POPF**, PUSHFD,POPFD	PUSHFD, POPFD - 80386↑
aritmetice (Cap.3)	**pt adunare**	**ADD**, XADD, **ADC**, **INC**	XADD - 80486↑
	pt scădere, negare	**SUB**, **SBB**, **DEC, NEG**	
	pt înmulţire	**MUL**, **IMUL**	IMUL - alte forme: cu 2,3 operanzi - 80286↑
	pt împărţire	**DIV**, **IDIV**	
	pt comparare	**CMP**, CMPXCHG, CMPXCHG[8,16]B	CMPXCHG-80486↑, CMPXCHG8B-Pentium↑, CMPXCHG16B–P4↑
	pt extinderea ACC	**CWD**, CDQ, CQO, **CBW**, CWDE, CDQE	CWDE , CDQ– 80386↑
	pt corecţia ACC	**DAA**, **DAS**, **AAA**, **AAS**, **AAM**, **AAD**	
pe biţi (Cap.4)	**logice**	**NOT**, **AND**, **OR**, **XOR**	
	de testare/comparare	**TEST**, BT,BT[S/R/C],BS[F/R],SETcc	BT,BT[S/R/C],BS[F/R],SETcc -80386↑
	de deplasare (shift)	**SHL/SAL**, **SHR**, **SAR**, SHLD, SHRD	SHLD,SHRD – 80386↑
	de rotire (rotate)	**ROL**, **RCL**, **ROR**, **RCR**	după cum se specifică la execuţia ***cpuid***
nu sunt suportate în modurile *real* şi *virtual-8086* de funcţionare al CPU	***la nivel de 32 biţi***	*ADCX, ADOX, POPCNT, LZCNT, TZCNT,*	
		ANDN, BEXTR, BLSI, BLSR, BLSMSK,	
		BZHI, MULX, PDEP, PEXT,	
		RORX, SARX, SHLX, SHRX	
de manipulare a şirurilor (Cap.5)	**pt operaţii primitive** movs,cmps,lods,stos,scas	**MOVSB**,**MOVSW**, MOVSD, MOVSQ **CMPSB**,**CMPSW**, CMPSD, CMPSQ **LODSB**,**LODSW**, LODSD, LODSQ **STOSB**,**STOSW**, STOSD, STOSQ **SCASB**,**SCASW**, SCASD, SCASQ	[MOV/CMP/...]SD–80386↑ [MOV/CMP/...]SQ–x64↑
	cu şiruri pe port	INS[B,W,D], OUTS[B,W,D]	[INS/ OUTS]B/W–80186↑, [INS/OUTS]D - 80386↑
	prefixe de repetare	**REP**,**REPE/REPZ**, **REPNE/REPNZ**	

ARHITECTURA PROCESOARELOR X86. SETUL DE INSTRUCŢIUNI GENERALE

Instrucţiuni pentru proceduri, întreruperi, buclare şi salt. Alte instrucţiuni			
(Cap.6)			
	de salt (ne)condiţionat	**JMP**, **J[condiţie]**	
	pt implementare bucle în program	**J[E]CXZ**, **LOOP[-/E/Z/NE/NZ]**	
	pt control indicatori (flags)	**CMC**,**CL[flag]**,**ST[flag]**,flag=C,D,I,CLAC,STAC	
(Cap.7)			
	pt proceduri/ întreruperi	**CALL**, **RET**, **INT**, **IRET**	
	pt controlul procesorului	**HALT, LOCK, WAIT, NOP**	
	pt identificare procesor	**cpuid**	

Capitolul 2. Instrucţiuni de transfer

Această categorie de instrucţiuni, de transfer a datelor, permite copierea, transferul sau chiar interschimbarea datelor între o sursă și o destinaţie; sunt suportate atât *transferuri necondiţionate* cât și *condiţionate*.

Instrucţiunile de transfer cuprind următoarele categorii de instrucţiuni:

1. **de transfer generale:** **MOV**, **MOV[S/Z]X[-/D]**, **XCHG**;
2. **de transfer condiţionat:** **CMOV$_{CC}$**, **XADD**;
3. **de transfer pentru conversie format:** **BSWAP**, **MOVBE**;
4. **de transfer cu stiva:** **PUSH**, **POP**, **PUSHA[-/D]**, **POPA[-/D]**;
5. **de transfer cu accumulatorul:** **XLAT[-/B]**, **IN**, **OUT**;
6. **de transfer pentru adrese:** **LEA**, **LDS**, **LES**, **LFS**, **LGS**, **LSS**;
7. **de transfer pentru flaguri:** **LAHF**, **SAHF**, **PUSHF[-/D/Q]**, **POPF[-/D/Q]**.

2.1. Instrucţiuni de transfer generale

Instrucţiunile de transfer caracterizate ca fiind *generale* cuprind:

- 2 instrucţiuni din setul original al procesorului **8086**, şi anume **MOV** și **XCHG**, dar și
- 2 instrucţiuni adăugate ulterior, la setul de instrucţiuni suportat de procesorul **80386** (**MOVSX** și **MOVZX**).

Tot aici trebuie menţionată și instrucţiunea **MOVSXD**, o continuare pe 64 biţi a instrucţiunii **MOVSX**, suportată deci de **procesoare pe 64 biţi.**

2.1.1. Instrucţiunea MOV

Instrucţiunea **MOV** ***(Data movement)*** realizează un transfer (sau mai bine spus o copiere) a datelor din *sursă* în *destinaţie*. Această instrucţiune face parte din setul de bază al procesoarelor **8086**.

MOV destinaţie, sursă ***; (destinaţie) ← (sursă)***

MOV {$reg_{8,16,32,64}$| $mem_{8,16,32,64}$| *sreg*}, {$reg_{8,16,32,64}$| $mem_{8,16,32,64}$| *sreg*| $imed_{8,16,32,64}$}

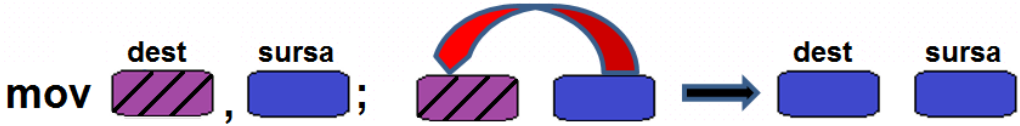


Figura 2-1.1. Ilustrarea modului de operare al instrucţiunii ***MOV***

Observaţii:

- Instrucţiunea MOV *nu afectează flagurile*; nici operandul sursă nu este afectat;
- Sursa şi destinaţia: - trebuie să aibă dimensiune egală,
  - nu pot fi *simultan* operanzi (locaţii) în memorie şi nici regiştri segment;

  Singura modalitate de a transfera date din memorie tot în memorie e prin foosirea instrucţiunii *movs*, vom vedea în *Cap.5*;
- Regiştrii segment nu pot fi încărcaţi cu date imediate (constante) direct; dacă se doreşte încărcarea unui registru segment, aceasta trebuie realizată prin intermediul unui registru GPR (AX de exemplu);
- Registrul CS nu poate fi destinaţie; pentru a încărca registrul CS se pot folosi instrucţiuni *far JMP*, *CALL* sau *RET*;
- Regiştrii [-/E/R] IP şi [-/E/R] FLAGS nu pot fi destinaţie;
- Valorile imediate (sau constantele cum se mai numesc) pot constitui operand sursă, dar niciodată operand destinaţie;
- Operanzii trebuie să aibă dimensiune egală, cei pe ***32 biţi*** au fost suportaţi în instrucţiune de la **80386↑,** iar cei pe ***64 biţi*** de la **Pentium 4** sau **Core 2↑**;

2.1.2. Instrucţiunile MOVSX, MOVZX şi MOVSXD (386↑)

Începând cu **80386↑**, dimensiunea operanzilor a crescut la 32 biţi și acesta a fost unul dintre motivele pentru care au fost adăugate instrucţiunile:

MOVSX ***(Move with Sign Extension)*** şi **MOVZX** ***(Move with Zero Extension).***

Aceste instrucţiuni copiază operandul sursă (din registru sau locaţie de memorie) în operandul destinaţie (obligatoriu registru) și îl extind cu semn sau cu zerouri la dimensiunea de 16 sau 32 biţi. Dimensiunea valorii convertite (dacă e vorba de operand sursă din memorie) se consideră în funcţie de tipul operandului destinaţie.

De la **Pentium 4** şi **Core 2↑**, dimensiunea datelor s-a dublat, a.î. se pot folosi **cvadruplucuvinte**, deci se lucrează şi cu regiştri pe ***64 biţi***, precum RAX, RBX, ... , sau chiar R8, R9, ... (dacă este activată extensia pe 64 biţi).

MOV[S/Z]X[-/D] destinaţie, sursă ; (destinaţie)← extensia cu semn/zero (sursă)

MOV[S/Z]X[-/D] {reg16,32,64}, {reg8,16,32| mem8,16,32}

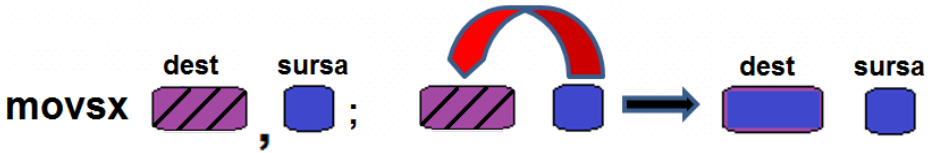


Figura 2-1.2. Ilustrarea modului de operare al instrucţiunii ***MOVSX***

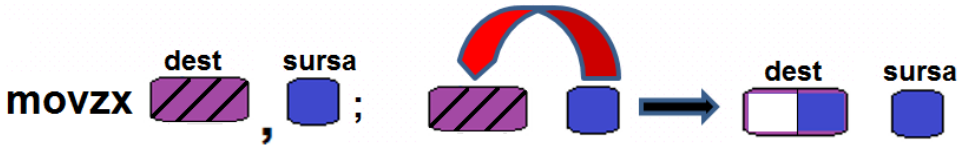


Figura 2-1.3. Ilustrarea modului de operare al instrucţiunii ***MOVZX***

Instrucţiunilor MOVSX și MOVZX le-a fost asigurată în mod diferit continuitatea înspre setul de instrucţiuni x64:
MOVSX a continuat cu **MOVSXD**, instrucţiune care *transferă un doubleword într-un quadword cu extensie de semn* (*MOVSXD* reg_{64}, $reg_{32}|mem_{32}$).
În schimb, instrucţiunea care ar fi fost echivalentă celei **MOVZX**, și anume
MOVZXD reg_{64}, $reg_{32}|mem_{32}$ nu a mai fost implementată, deși documentaţia Intel specifică faptul că se poate folosi
MOVZX reg_{32}, $reg_{16,8}|mem_{16,8}$.

Pentru obţinerea efectului lui MOVZXD, multiple surse recomandă folosirea **MOVSXD** modificată
(prin ștergerea părţii mai semnificative după execuţia instrucţiunii).

Aceste instrucţiuni copiază și extind:
- un operand sursă pe 8 biţi în operandul destinaţie de 16, 32 sau 64 biţi sau
- un operand sursă pe 16 biţi în operandul destinaţie de 32 sau 64 biţi
- un operand sursă pe 32 biţi în operandul destinaţie de 64 biţi (**P4↑**)
  - prin ***extinderea semnului*** (la **MOVSX**) respectiv
  - prin ***adăugarea de zerouri*** la (**MOVZX**).

Următoarele combinaţii sunt permise pentru **MOVSX** și **MOVZX**:

$reg_{16} \leftarrow reg_8|mem_8$, $reg_{32} \leftarrow reg_8|mem_8$, $reg_{64} \leftarrow reg_8|mem_8$,
$reg_{32} \leftarrow reg_{16}|mem_{16}$, $reg_{64} \leftarrow reg_{16}|mem_{16}$,

iar pentru **MOVSXD**: $reg_{64} \leftarrow reg_{32}|mem_{32}$

Observaţii:
- Instrucţiunile MOV[S/Z]X[-/D] *nu afectează flagurile*; nici sursa nu este afectată;
- Destinaţia trebuie să fie un registru și să aibă dimensiune cel puţin dublă comparativ cu cea a sursei; sursa poate fi doar registru sau zonă de memorie;
- Regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Regiștrii de 64 biţi sunt disponibili doar în modul de lucru pe 64 biţi al unui procesor (de la **Pentium 4** sau **Core 2↑**).

2.1.3 Instrucţiunea XCHG

Instrucţiunea **XCHG** ***(Data exchange)*** interschimbă conţinutul a doi operanzi.

Când e folosit registrul [-/E/R] AX, numit şi acumulator, instrucţiunea e codificată pe mai puţini octeţi, fiind și mai rapidă.

XCHG destinaţie, sursă ***; (destinaţie) ↔ (sursă)***

XCHG {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}, {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}

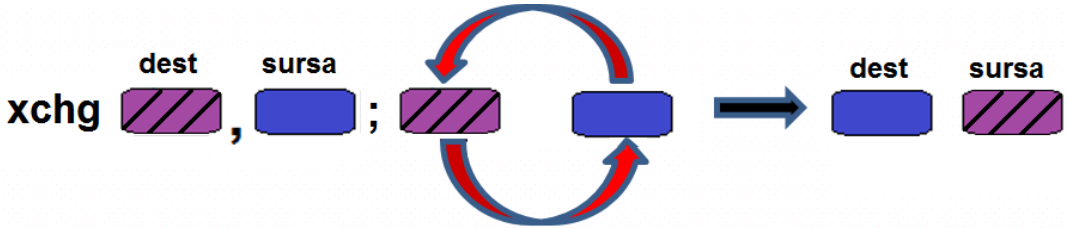


Figura 2-1.4. Ilustrarea modului de operare al instrucţiunii ***XCHG***

Observaţii:

- Instrucţiunea XCHG *nu afectează flagurile*; operandul sursă este afectat;
- Sursa şi destinaţia nu pot fi *simultan* operanzi (locaţii) în memorie;
- Regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Nu se pot folosi valori imediate în cadrul instrucţiunii XCHG;
- Operanzii trebuie să aibă dimensiune egală, cei pe 32 biţi au fost suportaţi în instrucţiune de la **80386↑**, iar cei pe 64 biţi de la **Pentium 4** sau **Core 2↑**.

2.1.4. Exemple

Exemple de *instrucţiuni ilegale:*

```
mov AX, BL          ; sursa şi destinaţia trebuie să aibă dimensiune egală;
mov AL, 400         ; valoarea imediată nu încape în registru
mov a, [BX]         ; nu pot fi ambii operanzi locaţii în memorie;
mov ES, DS          ; nu pot fi ambii operanzi simultan regiştri segment;
mov DS, 200h        ; nu pot fi transferate date imediate (constante) într-un registru segment
mov CS, AX          ; registrul CS nu poate fi destinaţie (pentru a nu altera zona de cod)
mov IP, AX          ; nu poate fi fol. registrul [-/E/R] IP pentru a nu altera starea CPU
mov FLAGS, AX       ; nu poate fi folosit registrul [-/E/R] FLAGS

mov 12h, AH         ; valorile imediate nu pot fi destinaţie
mov AH,1            ; nepermis în mod pe 64 biţi; nu se pot accesa biţii b15 … b8
mov R8B, BH         ; nu se pot accesa biţii b15-8 ai registrului R8 ca byte în mod pe 64 biţi

movsx AH, AL        ; destinaţia trebuie să aibă dimensiunea cel puţin dublă comparativ cu sursa;
movsx IP, AL        ; nu se permite folosirea lui [-/E/R] IP ca operand destinaţie
movsx FLAGS, AH     ; nu se permite folosirea lui [-/E/R] FLAGS ca operand destinaţie
movsx [BX], AL      ; nu e permisă zonă de memorie ca destinaţie
movsxd RAX, AX      ; când se fol. cu sufixul D, operandul destinaţie are dimensiunea de 64 biţi, iar cel sursă de 32 biţi.

xchg a, [SI]        ; cel puţin un operand din instrucţiunea XCHG trebuie să fie registru (nu pot fi ambii din memorie).
xchg DS, AX         ; regiştrii segment nu pot apărea ca operanzi la XCHG (nici [-/E/R] IP sau [-/E/R] FLAGS)
xchg AX, BL         ; sursa şi destinaţia trebuie să aibă dimensiune egală
```

Exemple de *instrucţiuni legale*:

Exemple 2-1.1

```
mov AH, AL      ; se pot accesa separat cei 2 octeţi ai regiştrilor generali de 16 biţi (AX, BX, CX, DX), dar
                ; aceasta nu se respectă şi la ceilalţi regiştri de 16 biţi (precum SI, DI, SP, BP) şi
                ; nici la cuvintele unui registru de 32 biţi sau la dublucuvintele unui registru de 64 biţi
mov BL, AH      ; BL=AH conţinutul registrului AH se copiază în registrul BL; transfer pe octet (8b)
mov AX, BX      ; AX=BX - transfer pe cuvânt (16b)
mov EAX, EBX    ; EAX=EBX-transfer pe dublucuv.(32b)–de la 80386↑
mov RAX, RBX    ; RAX=RBX-transfer pe cvadruplucuvânt (64b)-Pentium 4, Core 2↑
```

Exemplul 2-1.2 Pentru modul pe 64 biţi, de la **P4**, **Core 2↑** în sus sunt suportate:

```
mov RBX, 0123456789ABCDEFh      ; RBX= 01 23 45 67 89 AB CD EFh
mov R8B, BL     ; transfer pe 8 biţi,   R8 =  xx xx xx xx  xx  xx  xx  EFh
mov R8W, BX     ; transfer pe 16 biţi, R8 =  xx xx xx xx  xx  xx  CD EFh
mov R8D, EBX    ; transfer pe 32 biţi, R8 =  xx xx xx xx  89 AB CD EFh
mov R8, RBX     ; transfer pe 64 biţi, R8 = 01 23 45 67 89 AB CD EFh
```

Exemple 2-1.3

```
mov BL, 120     ; se încarcă în registrul BL octetul 78h=120 în zecimal
mov AX, 1234h   ; se încarcă în registrul AX cuvântul 1234h
mov AX, 'AB'    ; în registrul AX se depune 'B' și 'A', AH='B', AL='A' – coduri Ascii[1]
mov EAX,'abcd'  ; în registrul EAX vom avea: EAX=64636261h, adică ‚dcba'[2]
mov ECX, 10b    ; se încarcă în registrul ECX dublucuvântul 10b=0000 0002h
mov AH, 1       ; nu e permisă în modul pe 64 biţi, dar este permisă în modul pe 16 biţi sau pe 32 biţi
mov RCX, 10h    ; se încarcă RCX cu valoarea imediată 10h scrisă pe 64 biţi RCX=00 00 00 00  00 00 00 10h
```

---

[1] In simulatorul EMU8086, instrucţiunea mov AX, 'AB' va încărca: AH='A', AL='B'

[2] În Visual C, instrucţiunea mov EAX, 'abcd' va încărca: EAX=61626364h, adică ‚abcd', dar în Olly DBG va fi EAX=64636261h, deci ‚**dcba**'

Exemplul 2-1.4

```
.data
a dw 1234h          ; se definește variabila a de tip cuvânt în memorie, de valoare 1234h
b dw 0,1,2,3        ; se definește variabila b de tip șir de cuvinte, ca în Figura 2-1.5
.code
mov AX, a           ; se încarcă registrul AX cu valoarea lui a, deci AX=1234h
mov b[2], AX        ; în zona de memorie, la locaţiile b[2] și b[3] se pune cuvântul 1234h după convenţia Little Endian
```

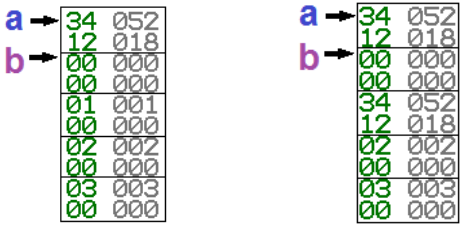


Figura 2-1.5. Ilustrarea exemplului **2-1.4** (înainte și după execuţia celor 2 instrucţiuni)

Exemple 2-1.5 Exemplele următoare se pot considera individuale:

```
mov sir[2], 65          ; în octetul din memorie, adresat de offset sir+2 se depune valoarea 65=41h
mov word ptr sir[2], 65 ; cu specificatorul de tip se poate impune dimensiunea destinaţiei să fie cuvânt, dublucuvânt, etc
```

Diferenţa între cele 2 instrucţiuni de mai sus este că în primul caz e afectată doar locaţia [sir+2], iar în cel de-al doilea caz e afectată şi locaţia [sir+2+1] unde în acest caz se va înscrie 0.

Exemplul 2-1.6

```
mov EDX, 12345678h
mov ECX, [EDX]      ; în ECX se aduce dublucuvântul conţinut în memorie la offsetul 12345678h din segmentul de date;
```

Exemplul 2-1.7

```
mov RDX, 1000b      ; RDX = 00 00 00 00 00 00 00 08h
mov RCX, [RDX]      ; RCX = conţinutul din memorie de la adresa 08h, pe 64 biţi (adică un cvadruplucuvânt)
```

Exemplul 2-1.8 Se urmărește încărcarea unui registru segment la un CPU pe 16 biţi, 8086 de exemplu (adresare segmentată) cu o valoare imediată; astfel, vrem să poziţionăm DS pe valoarea 1200h; se dorește deci obţinerea DS=1200h:

```
mov AX,1200h        ; AX=1200h, se dorește încărcarea unui registru segment,
mov DS, AX          ; DS=AX=1200h–posibil doar prin intermediul unui registru (AX de exemplu),
                    ; nu imediat (nu direct cu o constantă)
```

Exemplul 2-1.9 Lucrul cu regiștri segment la un CPU pe 16 biţi, 8086 de exemplu (adresare segmentată):

```
mov AX, CS          ; se fol. registrul AX ca intermediar în efectuarea transferului, iar valoarea din DS va fi aceeași
mov DS, AX          ; cu valoarea din CS; astfel, adresa de început a segmentului de date și cod va fi aceeași
```

Exemple 2-1.10 Lucrul cu regiștri segment la un CPU pe 16 biţi, 8086 de exemplu (adresare segmentată):

```
mov CS, AX          ; nu e indicată modificarea lui CS deoarece pot apărea erori la execuţia programului
mov val_ds, DS      ; se poate pune valoarea registrului segment și în memorie, într-o variabilă
```

Exemplul 2-1.11 La un CPU pe 16 biţi (adresare segmentată), se dorește ca octetul aflat în memorie în segmentul implicit (DS) la offsetul 24h să fie transferat (copiat) în registrul BL:

```
mov BL, [24h]       ; BL = (DS:24h), DS implicit; octetul din memorie, din segmentul dat de DS, de la offsetul 24h,
                    ; e copiat în registrul BL.
```

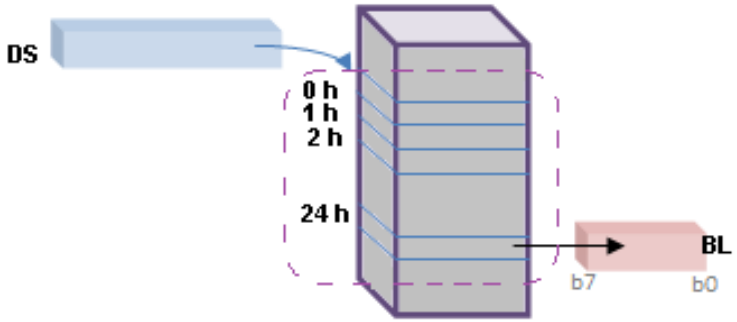


Figura 2-1.6. Ilustrarea exemplului **2-1.11**

Exemplul 2-1.12 Se dorește ca dublucuvântul aflat în memorie în segmentul implicit (DS) începând de la offsetul 24h să fie transferat (copiat) în registrul EBX:

```
mov EBX, [24h]      ; EBX = (DS:27h, DS:26h, DS:25h, DS:24h), DS implicit;
                    ; dublucuvântul din memorie, din segmentul dat de DS, începând cu offsetul 24h, e copiat în
                    ; registrul EBX, după cum spune convenţia Little-END-ian.
```

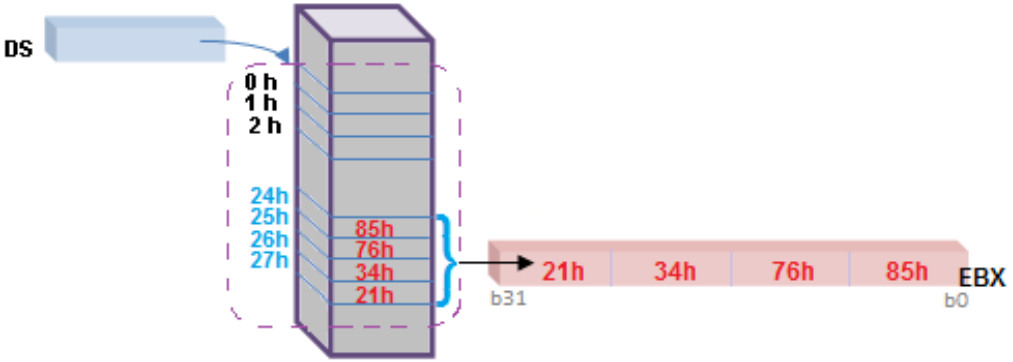


Figura 2-1.7. Ilustrarea exemplului **2-1.12**

Exemple 2-1.13 Deși cele 3 instrucţiuni de mai jos prezintă o oarecare asemănare, codificarea lor se realizează diferit:

```
mov AX,1            ; se codifică b8 01 00h pe 3 octeţi
mov AH,1            ; se codifică b4 01h pe 2 octeţi
mov AL,1            ; se codifică b0 01h pe 2 octeţi
```

în plus, instrucţiunile care folosesc Accumulatorul sunt mai scurte decât cele care folosesc alţi regiştri de uz general:

```
mov AX, [1234h]     ; se codifică A1 34 12h pe 3 octeţi
mov BX, [1234h]     ; se codifică 8B 1E 34 12h pe 4 octeţi
```

Exemple 2-1.14 Se poate copia din memorie în registru, doar dacă dimensiunea operanzilor potrivește; în exemplele de mai jos, e nevoie ca val1 să fie de tip octet, val2 de tip cuvânt, iar val3 de tip dublucuvânt. Instrucţiunile se pot folosi și cu operanzii interschimbaţi, din registru să se copieze într-o variabilă din memorie:

```
mov AL, val1        ; se copiază un singur octet, cel dat de offset val1; AL=[val1]
mov AX, val2        ; se copiază doi octeţi, cei daţi de offset val2+0 și offset val2+1; AX=[val2+1][val2+0]
mov EAX, val3       ; se copiază patru octeţi, cei daţi de offset val3+0, offset val3+1, offset val3+2 și offset val3+3, deci
                    ; EAX = [val3+3][val3+2][val3+1][val3+0]
```

Exemple 2-1.15

```
movsx EAX, AL           ; extinde AL cu bitul de semn în tot registrul EAX (80386↑)
movzx EAX, byte ptr [BX]  ; octetul adresat de BX în segmentul DS este copiat în EAX (prin extindere cu 24 zerouri) (80386↑)
movzx RAX, word ptr[EDI]; cuvântul adresat de EDI e extins cu zerouri în RAX (prin extindere cu 48 zerouri) (P 4, Core 2↑)
```

Exemplul 2-1.16 Se dorește ca octetul aflat în memorie în segmentul implicit (DS) la offsetul 24h să fie transferat (copiat) în registrul AL și apoi extins cu semn la tot registrul EAX:

```
movsx EAX, byte ptr [24h]           ; EAX = (DS:24h), extins cu semn la EAX;
```

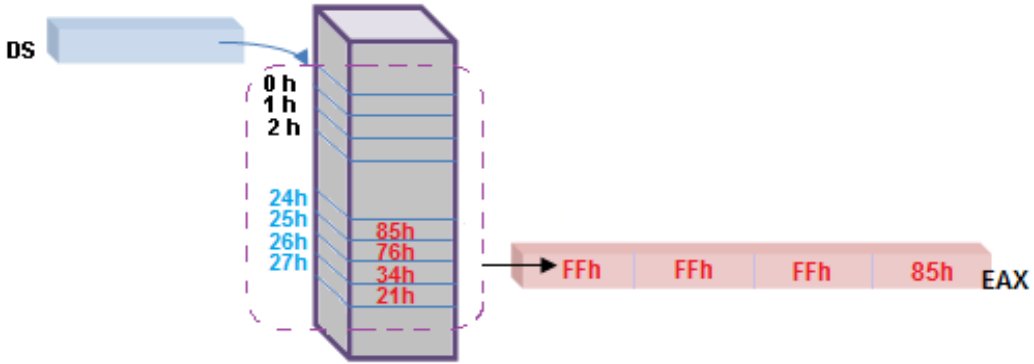


Figura 2-1.8. Ilustrarea exemplului **2-1.16**

Exemplul 2-1.17

```
mov EAX, 12345678h      ; EAX=12345678h
mov AH, 0FFh            ; EAX=1234FF78h
movsx EAX, AH           ; EAX=FFFFFFFFh
movsx EBX, AL           ; EBX=FFFFFFFFh
movzx EAX, AL           ; EAX=000000FFh
```

Exemple 2-1.18

```
xchg BH, CL             ; interschimbă BH cu CL, operaţie la nivel de octet
xchg AX, BX             ; interschimbă AX cu BX, operaţie la nivel de cuvânt
xchg EBX, ECX           ; interschimbă EBX cu ECX, operaţie pe dublucuvânt (de la 80386↑)
xchg RBX, RCX           ; interschimbă RBX cu RCX, operaţie pe cvadruplucuvânt (de la Pentium 4 ↑)
```

Exemplul 2-1.19

```
xchg DL, VAR            ; interschimbă octetul din memorie definit de VAR cu octetul din registrul DL
                        ; (VAR se consideră de tip octet din cauza dimensiunii registrului DL)
```

Exemple 2-1.20

```
mov AX, 1234h           ; AX=1234h
mov BX, 5678h           ; BX=5678h
xchg AX, BX             ; interschimbă AX cu BX
```

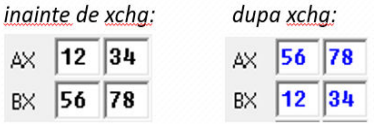


Figura 2-1.9. Ilustrarea exemplului **2-1.20**

Exemplul 2-1.21

```
.data
sir db 10h, 20h, 30h, 40h, 50h, 60h          ; se definește un șir de octeţi în memorie
.code
mov AX, 7654h
xchg AX, word ptr sir[2]    ; interschimbă conţinutul din AX cu cel din memorie adresat de offset sir+2,
                            ; adică în loc de 30h și 40h vom avea 54h și 76h, iar conţinutul din AX va fi 4030h
```

Exemplul 2-1.22

```
.data
sir db 10h, 20h, 30h, 40h, 50h, 60h          ; se definește un șir de octeţi în memorie
.code
mov EAX, 12345678h      ; EAX=12345678h
xchg dword ptr sir, EAX ; interschimbă primii 4 octeţi din memorie începând de la offset sir (considerându-i după convenţia
                        ; Little Endian) cu valoarea din EAX, deci EAX=40302010h, iar sir: 78h, 56h, 34h, 12h, 50h, 60h.
```

Exemplul 2-1.23 Dacă dorim să interschimbăm conţinutul a 2 locaţii din memorie, de exemplu acestea fiind elementele unui șir de octeţi pe care vrem să-l ordonăm crescător în memorie (de la adrese mai mici spre adrese mai mari), se poate folosi instrucţiunea *XCHG* ca în secvenţa de mai jos:

```
; de exemplu în memorie avem 2 octeţi care trebuie interschimbaţi: 75h şi 34h, aşa cum arată Figura 2-1.10
mov AL, sir[2]          ; AL=75h; cele două formulări sir[2] şi sir+2 produc acelaşi efect
xchg AL, sir[3]         ; după execuţia instrucţiunii XCHG, AL=34h, iar sir[3]=75h
mov sir[2], AL          ; după execuţia instrucţiunii MOV, sir[2]=34h
```

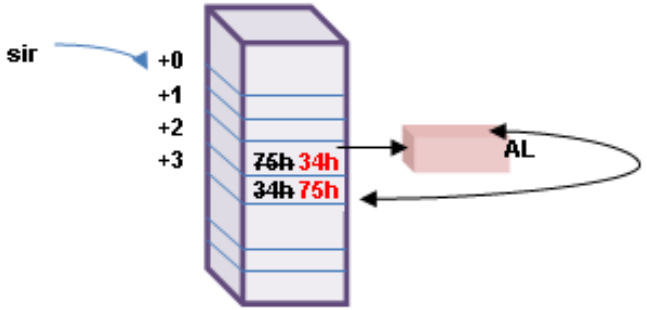


Figura 2-1.10. Ilustrarea exemplului **2-1.23**

2.2. Instrucţiuni de transfer condiţionat

Instrucţiunile de transfer condiţionat **CMOVcc** și **XADD** nu au făcut parte din setul iniţial al **8086**, ci au apărut ulterior, la **procesoare pe 32 biţi**.

2.2.1. Instrucţiunea CMOVcc (P Pro↑)

Începând cu familia de procesoare **P6↑**, a apărut instrucţiunea **CMOVcc** ***(Conditional movement)***, dar aceasta nu e suportată de toate procesoarele pe 32 biţi; pentru a determina dacă un anumit procesor suportă instrucţiunea **CMOVcc**, se poate verifica dacă bitul b15 (bit numit ‚cmov') din registrul EDX este setat după execuţia instrucţiunii **CPUID** cu EAX=00000001h la intrare:

EAX=00000001h ---> **CPUID** ---> EDXb15=1 sau dacă CPUID.00000001h.EDX[15] = 1;

instrucţiunea poate fi folosită în programe doar dacă se specifică directiva .686.

Instrucţiunea CMOVcc are mai multe forme; în funcţie de forma pe care o ia, instrucţiunea CMOVcc va verifica o anumită stare, în concordanţă cu mnemonica – această stare privește unul sau mai multe flaguri de caracteristici din registrul [-/E/R] FLAGS (CF, OF, PF, SF și ZF) și realizează o operaţie de transfer dacă flagul respectiv este într-o anumită stare (sau respectă o condiţie). În cadrul mnemonicii, sufixul ***cc*** arată sau sugerează condiţia care este verificată. Dacă acea condiţie nu e satisfăcută, operaţia de transfer nu se execută și programul continuă cu execuţia instrucţiunilor următoare.

CMOVcc destinaţie, sursă; ***destinaţie ← sursă dacă se îndeplinește cc***

CMOVcc {reg$_{16,32,64}$}, { reg$_{16,32,64}$ | mem$_{16,32,64}$}

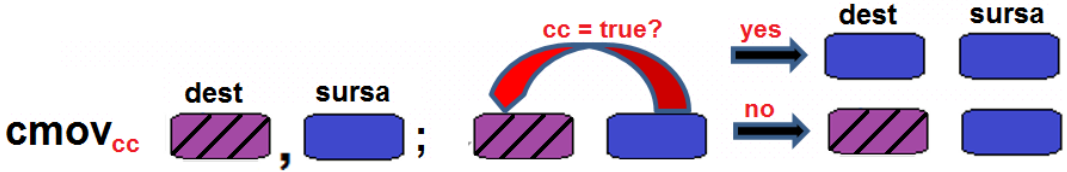


Figura 2-2.1. Ilustrarea modului de operare al instrucţiunii ***CMOV$_{cc}$***

Observaţii:

- Instrucţiunea CMOVcc *nu afectează flagurile*; nu se modifică sursa;
- Regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Nu se pot folosi valori imediate în cadrul instrucţiunii CMOVcc;
- Operanzii trebuie să aibă dimensiune egală, cei pe 16 biţi se pot folosi în modul real de funcţionare al procesorului; în plus, și cei pe 32 biţi, respectiv pe 64 biţi pot fi folosiţi în modurile cu nume corespunzător (mod pe 32 biţi, respectiv mod pe 64 biţi);
- Nu se pot folosi operanzi de 8 biţi;
- Se folosește directiva .686 la scrierea programului în asamblare.

Există **condiţii** care sunt ***verificate de mai multe instrucţiuni***:

- de exemplu CMOVNC, CMOVNB și CMOVAE, sau
- CMOVC, CMOVB și CMOVNAE,

⇨ o anumită stare poate fi interpretată în mai multe moduri
sau altfel spus există ***multiple mnemonici pentru un același cod al operaţiei***.

La evaluarea condiţiei din cadrul instrucţiunii CMOVcc, se folosesc noţiuni diferite, în funcţie de convenţia de reprezentare a numerelor: ***cu semn (signed)*** / ***fără semn (unsigned)***:

noţiunea de „mai mic” (**less**) sau „mai mare” (**greater**) pentru compararea numerelor ***cu semn (signed)***,
iar termenii „deasupra, peste” (**above**) sau „dedesubt, sub” (**below**) se folosesc pentru numere ***fără semn (unsigned)***.

Condiţia pentru fiecare mnemonică CMOVcc este dată în continuare, deci ***cc*** poate lua următoarele forme:

și unsigned și signed	CMOV***E*** sau CMOV***Z***	- equal or zero (ZF=1)
	CMOV***NE*** sau CMOV***NZ***	- not equal/ zero (ZF=0)

(signed)		(unsigned)	
CMOV***S***	- sign (SF=1),	CMOV***C***	- carry (CF=1),
CMOV***NS***	- not sign (SF=0),	CMOV***NC***	- not carry (CF=0),
CMOV***O***	- overflow (OF=1),	CMOV***P*** sau CMOV***PE***	- parity sau parity even (PF=1),
CMOV***NO***	- not overflow (OF=0),	CMOV***NP*** sau CMOV***PO***	- not parity sau parity odd (PF=0),
CMOV***G***	- greater (ZF=0 și SF=OF),	CMOV***B***	- below (CF=1),
CMOV***GE***	- greater or equal (SF=OF),	CMOV***BE***	- below or equal (CF=1 sau ZF=1),
CMOV***NG***	- not greater (ZF=1 sau SF≠OF),	CMOV***NB***	- not below (CF=0),
CMOV***NGE***	- not greater or equal (SF≠OF),	CMOV***NBE***	- not below or equal (CF=0 și ZF=0),
CMOV***L***	- less (SF≠OF),	CMOV***A***	- above (CF=0 și ZF=0),
CMOV***LE***	- less or equal (ZF=1 sau SF≠OF),	COMV***AE***	- above or equal (CF=0),
CMOV***NL***	- not less (SF=OF),	CMOV***NA***	- not above (CF=1 sau ZF=1),
CMOV***NLE***	- not less or equal (ZF=0 şi SF=OF),	CMOV***NAE***	- not above or equal (CF=1),

2.2.2. Instrucţiunea XADD (486↑)

Începând cu familia de procesoare **486↑**, a apărut instrucţiunea **XADD** ***(Exchange and add)*** care interschimbă primul operand (destinaţia) cu cel de-al doilea operand (sursa), iar apoi încarcă suma celor doi operanzi în operandul destinaţie. Operandul destinaţie poate fi un registru sau o locaţie de memorie; operandul sursă este un registru întotdeauna.

XADD destinaţie, sursă ***; (sursa) ↔ (destinaţie)***

XADD { $reg_{8,16,32,64}$ | $mem_{8,16,32,64}$ }, { $reg_{8,16,32,64}$ } ***; (destinaţie) ← (sursa) + (destinaţie);***

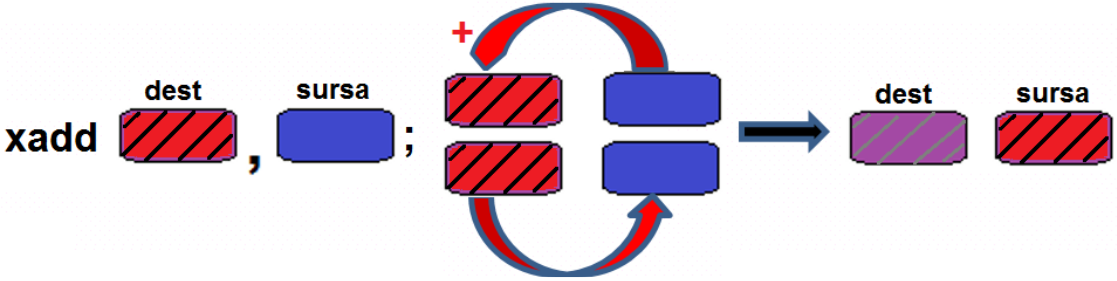


Figura 2-2.2. Ilustrarea modului de operare al instrucţiunii ***XADD***

Observaţii:

- Instrucţiunea XADD *afectează flagurile* în următorul mod: flagurile CF, PF, AF, SF, ZF și OF sunt setate în concordanţă cu rezultatul obţinut după adunare, acest rezultat fiind stocat în operandul destinaţie;
- Sursa nu poate fi dată imediată sau locaţie din memorie (sursa se va modifica);
- Regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Această instrucţiune modifică inclusiv operandul sursă !
- Operanzii trebuie să aibă dimensiune egală, cei pe **8 biţi** și **16 biţi** se pot folosi în modul real de funcţionare al procesorului; în plus, și cei pe **32 biţi**, respectiv pe **64 biţi** pot fi folosiţi în modurile cu nume corespunzător (mod pe 32 biţi, mod pe 64 biţi).

2.2.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
; pentru exemplificare, voi folosi instrucţiunea condiţională CMOVC, dar se poate folosi oricare dintre cele suportate.
cmovc DS, AX          ; nu se pot folosi regiștri segment
cmovc AX, 12h         ; nu se pot folosi valori imediate
cmovc EAX, BX         ; nu se pot folosi operanzi de dimensiuni diferite
cmovc AH, AL          ; operanzii pe 8 biţi nu sunt suportaţi de instrucţiune

xadd AX, 1234h        ; sursa să nu fie dată imediată
xadd AX, var          ; sursa să nu fie operand din memorie
xadd AX, [BX]         ; sursa să nu fie operand din memorie
xadd DS, AX           ; nu se admit regiștri segment ca operanzi
xadd AX, BL           ; operanzii trebuie să aibă dimensiune egală
```

Exemple de *instrucţiuni legale*:

Exemple 2-2.1

```
cmovnz AX, var    ; dacă rezultatul operaţiei anterioare nu a setat ZF, se va transfera conţinutul din memorie al lui var în AX
cmovc EAX, [ESI]      ; dacă CF=1 se va muta dublucuvântul din memorie (de la offsetul dat de ESI
                                                                    ; în cadrul segmentului DS) în EAX
cmovz AX, BX          ; dacă ZF=1 (obţinut din operaţia anterioară), atunci se va transfera conţinutul din BX în AX
```

Exemplul 2-2.2

```
mov EAX, 15           ; EAX=00 00 00 0Fh
mov EBX, 6            ; EBX=00 00 00 06h
xadd EAX, EBX         ; EAX=EAX+EBX=00 00 00 15h și EBX=00 00 00 0Fh
```

Exemplul 2-2.3 Interschimbă primul element al unui şir cu conţinutul lui AL, iar suma lor se depune ca prim element al şirului.

```
.data
a db 1,2,3,4,5
.code
mov AL, 2
xadd [a], AL          ; în memorie se va depune la adresa offset a valoarea 03h, iar în registrul AL va fi valoarea 01h
```

Exemplul 2-2.4

```
.data
a dd 12345678h, 9ABCDEF0h, 0
.code
mov EAX, [a+4]
xadd [a+8], EAX       ; în memorie se va depune în locul lui 0 (deci ca al III-lea element al şirului)
                      ; valoarea sumei 9ABCDEF0h + 0, iar EAX va fi 0
                      ; deci şirul a din memorie va fi acum a = 12345678h, 9ABCDEF0h, 9ABCDEF0h
```

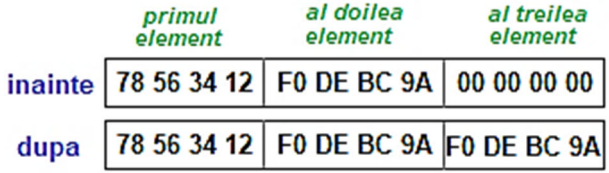


Figura 2-2.3. Ilustrarea modului de aranjare a datelor în memorie pentru exemplul 2-2.4, ca dublucuvinte în format Little Endian (înainte şi după execuţia instrucţiunii XADD) (valorile octeţilor au fost exprimate în hexazecimal)

2.3. Instrucţiuni de transfer pentru conversie format

Aceste instrucţiuni se referă la conversia între formatul Little End-ian (LE) și Big End-ian (BE); se fol. în general atunci când se dorește un schimb de date între procesoare cu arhitecturi diferite. Modificarea formatului din LE în BE sau invers se realiz. prin schimbarea ordinii celor 4 octeţi respectiv 8 octeţi ce compun data din operand. La *BSWAP*, operandul se numește operand destinaţie și poate fi doar registru GPR pe 32 biţi sau 64 biţi, iar la *MOVBE* se numește operand sursă și poate fi și o zonă de memorie. Aceste instrucţiuni, BSWAP și MOVBE, au fost suportate pentru prima oară de procesoare pe 32 biţi.

2.3.1. Instrucţiunea BSWAP (486↑)

Începând cu familia de procesoare **486↑**, a apărut instrucţiunea **BSWAP** ***(Byte swap)*** care inversează ordinea octeţilor într-un dublucuvânt (32 biţi). Ulterior, de la procesoarele pe 64 biţi, instrucţiunea a fost suportată și la nivel de 64 biţi, deci operanzi la nivel de cvadruplucuvânt. Se folosește un registru temporar pentru a realiza inversarea octeţilor. Obţinerea aceluiași efect pe un cuvânt (16 biţi), lucru care ar fi fost necesar la operarea în mod real, poate fi realizată prin instrucţiunea XCHG.

BSWAP destinaţie ***; (destinaţie) ← (destinaţie(octeţi inversaţi ca ordine))***
BSWAP {reg$_{32,64}$}

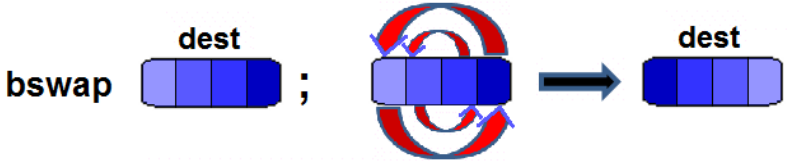


Figura 2-3.1. Ilustrarea modului de operare al instrucţiunii ***BSWAP***

Observaţii:

- Instrucţiunea BSWAP *nu afectează flagurile*;
- Folosește un singur operand (destinaţie) pe care îl modifică; acesta nu poate fi locaţie din memorie sau dată imediată;
- Regiştrii segment nu pot fi operand (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Se poate folosi ca operand doar un registru GPR, iar dimensiunea acestuia poate fi doar de 32 biţi sau de 64 biţi; nu se admit cei pe 8 biţi sau 16 biţi.

2.3.2. Instrucţiunea MOVBE (Intel Atom↑)

Instrucţiunea **MOVBE** ***(Move data after swapping bytes)*** a apărut prima dată la procesoare pe 32 biţi, fiind specifică în general familiei de procesoare **Intel Atom**. Această instrucţiune realizează un transfer după ce octeţii au fost inversaţi în cadrul structurii operandului sursă (pentru inversare se fol. un registru temporar). Pentru a verifica dacă un anumit procesor suportă instrucţiunea **MOVBE**, se poate verifica dacă bitul b22 (bit numit ‚movbe') din registrul ECX este setat după execuţia instrucţiunii **CPUID** cu EAX=00000001h la intrare: EAX=00000001h --->**CPUID**---> $ECX_{b22=1}$ sau dacă CPUID.00000001h.ECX[22] = 1;

MOVBE destinaţie, sursă ***; (destinaţie) ← (sursă(octeţi inversaţi ca ordine))***

MOVBE { $reg_{16,32,64}$ | $mem_{16,32,64}$ }, { $reg_{16,32,64}$ | $mem_{16,32,64}$ }

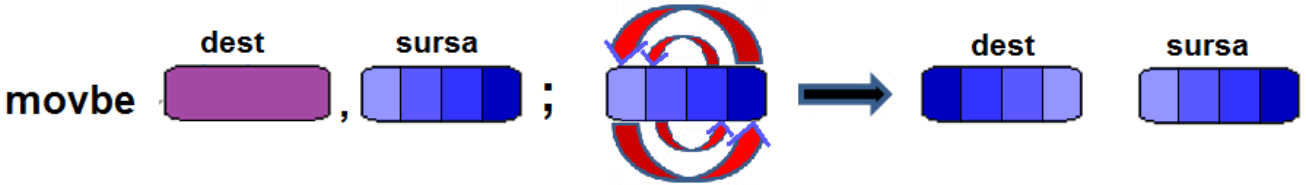


Figura 2-3.2. Ilustrarea modului de operare al instrucţiunii ***MOVBE***

Observaţii:

- Instrucţiunea MOVBE *nu afectează flagurile*;
- Operanzii pot fi doar regiștri sau zone din memorie (nu pot fi constante sau date imediate), dar nu pot fi de același tip – obligatoriu unul va fi registru și celălalt va fi din memorie;
- Sursa nu poate fi dată imediată; această instrucţiune nu modifică operandul sursă;
- Regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, nici [-/E/R] FLAGS);
- Operanzii trebuie să aibă dimensiune egală, cei pe 16 biţi se pot folosi în modul real de funcţionare al procesorului; în plus, și cei pe 32 biţi, respectiv pe 64 biţi pot fi folosiţi în modurile cu nume corespunzător (mod pe 32 biţi, mod pe 64 biţi); nu se pot folosi operanzi pe 8 biţi.

2.3.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
bswap var          ; operandul destinaţie nu poate fi din memorie
bswap [EDI]        ; operandul destinaţie nu poate fi din memorie
bswap 1234h        ; operandul destinaţie nu poate fi dată imediată
bswap DS           ; operandul destinaţie nu poate fi registru segment nici [-/E/R] IP, nici [-/E/R] FLAGS
bswap AX           ; nu se poate folosi registru de 16 biţi, doar de 32 sau 64 biţi

movbe AX, BX       ; operanzii nu pot fi ambii regiştri
movbe var1, [ESI]  ; operanzii nu pot fi ambii locaţii de memorie
movbe AX, 1234h    ; operandul sursă nu poate fi dată imediată;
movbe DS, AX       ; regiştrii segment nu pot fi operanzi (nici [-/E/R] IP, [-/E/R] FLAGS);
movbe EAX, BX      ; operanzii trebuie să aibă dimensiune egală, minim 16 biţi
```

Exemple de *instrucţiuni legale*:

Exemplul 2-3.1

```
mov EAX, 12345678h     ; EAX = 12345678h
bswap EAX              ; EAX = 78563412h
```

Exemplul 2-3.2

```
mov RBX, 0102030405060708h     ; RBX = 0102030405060708h
bswap RBX                      ; RBX = 0807060504030201h
```

Exemplul 2-3.3

```
mov AX, 1234h          ; AX=1234h
mov BX, 5678h          ; BX=5678h
movbe AX, BX           ; AX=7856h, BX=5678h
```

Exemplul 2-3.4

```
mov EAX, 12345678h      ; EAX = 12345678h
movbe EBX, EAX          ; EBX = 78563412h
```

Exemplul 2-3.5

```
.data
a dd 01020304h, 05060708h
.code
mov EAX, a              ; EAX = 01020304h
movbe EAX, a            ; EAX = 04030201h
mov EBX, a+4            ; EBX = 05060708h
movbe EBX, a+4          ; EBX = 08070605h
bswap EBX               ; EBX = 05060708h
movbe a+6, AX           ; a = 01020304h, 01020708h,
                        ; dar în memorie (sub formă LittleEnd-ian) se vor vedea octeţii: 04|03|02|01 şi apoi 08|07|02|01
```

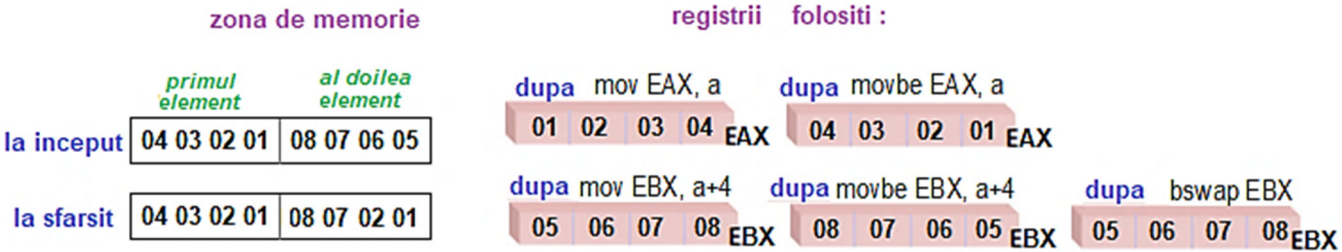


Figura 2-3.3. Ilustrarea modului de aranjare a datelor în memorie pentru exemplul 2-3.5, ca dublucuvinte în format Little Endian (înainte şi după execuţia secvenţei de instrucţiuni) (valorile octeţilor au fost exprimate în hexazecimal)

2.4. Instrucţiuni de transfer cu stiva

Stiva este o zonă specială din memorie în care se pot stoca temporar date. De ex., aici se pot depune temporar valorile unor regiștri care ar urma să fie modificaţi în program; prin folosirea stivei, valorile iniţiale ale regiștrilor respectivi pot fi restaurate. Aceste depuneri (și restaurări) de valori în (din) stivă se realizează cu ***operaţii specifice stivei***, operaţii de tip ***PUSH*** și ***POP***.

De asemenea, în cadrul apelului unei subrutine, cu *CALL* (deci atunci când se intră într-un subprogram nou din programul principal), starea curentă a procesorului (dată de conţinutul regiștrilor din programul principal, dar și valorile curente ale (E)IP și CS) sunt depuse pe stivă; la sfârșitul subprogramului respectiv, toate aceste valori sunt refăcute de pe stivă în mod automat. Astfel, ***stiva*** este definită ca ***o zonă din memorie de tip listă LIFO (Last In First Out)*** şi se află *în segmentul de stivă*, pointată de registrul segment SS. (LIFO - analogie cu un sac în care se stivuiesc articole vestimentare împăturate)

La implementarea stivei, zona de memorie este cu depunere înspre adrese mai mici, ceea ce înseamnă că adresa cea mai mică din segmentul SS este adresa de bază a segmentului și (E)BP (Extended Base Pointer) este egal cu această valoare (acolo se află baza stivei). Pe de altă parte, cea mai mare adresă va fi în registrul (E)SP (Extended Stack Pointer), acesta modificându-și frecvent valoarea, în funcţie de depunerile și extragerile efectuate cu stiva (registrul (E)SP desemnează vârful stivei sau locaţia elementului curent din stivă).

De exemplu, în simulatorul EMU8086, în cadrul programelor de tip .COM, segmentul de stivă (care se suprapune pe cel de date și cod la ***programe .COM***) are alocată ca adresă de bază valoarea 0700h (deci SS=0700h), iar registrul BP are valoarea 0000h (baza stivei). Valoarea registrului SP este iniţializată la începutul programului în mod automat cu valoarea FFFEh (sau în funcţie de dimensiunea impusă pentru stivă), iar la depunerea unei valori de 16 biţi, această valoare se va decrementa cu 2 (va urca în memorie), deci SP va deveni FFFCh, apoi la următoarea depunere va deveni FFFAh ş.a.m.d.

În cadrul unui ***program de tip .EXE***, începutul segmentului de cod, date şi stivă nu mai coincide, însă aceste segmente se pot suprapune. De exemplu, DS=0700h, SS=0712h, CS=0722h. Presupunând că s-a alocat o zonă de 100h octeţi pentru stivă, vom avea SP=0100h, iar BP=0000h. În gen., putem spune că întotdeauna ***diferenţa (E)SP-(E)BP arată câţi octeţi mai încap pe stivă până la umplere***. De exemplu, dacă în stiva de 100h dimensiune (deci cu SP=0100h iniţial) se depune un cuvânt, SP devine 00FEh, iar la depunerea unui al doilea cuvânt, SP devine 00FCh. În plus, se poate afla numărul de operaţii PUSH cu operand de tip cuvânt (sau dublucuvânt) posibile printr-o simplă împărţire cu 2 (respectiv cu 4) a valorii din (E)SP.

Pentru a defini o zonă de stivă în cadrul unui program (de tip .EXE) se poate folosi directiva cu același nume:

.stack 100h ; se va defini o zonă de 100h sau 256 octeţi ca fiind „stivă”.

Directiva *.stack* se specifică la început, după directiva *.model* (definește modelul de memorie dorit). La modelul de memorie tiny, definirea dimensiunii stivei poate lipsi, întrucât DOS va face alocarea automată, însă la modelele de memorie mai mari, dacă se omite această specificare, poate să apară mesaj de avertizare „No stack segment” la linkeditare. Această avertizare poate fi ignorată dacă zona de stivă nu va depăși 128 octeţi, dar altfel se poate ajunge chiar la căderea sistemului (deoarece se suprascrie o zonă din memorie numită PSP). Ce este/ ce conţine aceasta zonă PSP ?

La începutul fiecărui program în memorie se atașează un antet numit ***Program Segment Prefix (PSP)*** care conţine multiple ***informaţii***, unele ***critice pentru rularea programului sau chiar a sistemului***; printre aceste informaţii se numără: descrierea contextului în care rulează programul, numărul de parametri, care sunt acești parametri, etc. Cum stiva crește „în sus”, depășirea dimensiunii alocate (în mod automat de către sistem) pentru stivă va duce la alterarea acestei zone din memorie, deci cu risc de „system crash”. Dacă se folosește modelul de memorie *tiny*, stiva va fi localizată la capătul segmentului (dimensiunea maximă a segmentului fiind 64kB), permiţând astfel mai mult spaţiu de depozitare în stivă.

Instrucţiunile care folosesc stiva generează automat adresele din memorie, prin ***controlarea regiștrilor SS și (E)SP*** ((Extended) Stack Pointer), care indică vârful stivei. Iniţial, stiva nu conţine date, dar pe măsura introducerii acestora în zona de stivă (cu PUSH), dimensiunea stivei creşte, întinzându-se spre adrese mai mici (stiva creşte „în sus” şi (E)SP va descreşte). Deoarece ***stiva se umple înspre adrese ce descresc***, pointerul SS:(E)SP indică adrese tot mai mici şi conţine întotdeauna ***adresa ultimului operand introdus în stivă*** sau unde ar urma să se depună un nou element, numit deseori “*elementul din vârful stivei*”.

Iniţial, stiva nu conţine date, este goală, dar pe măsura introducerii elementelor în stivă, se spune că dimensiunea stivei creşte (sacul se umple), întinzându-se ***spre adrese mai mici*** (stiva creşte „**de jos în sus**”), așa cum sugerează Figura 2-4.1.

În cazul în care stiva s-ar umple (ar ajunge până în vârf în Figura 2-4.1) și în continuare s-ar efectua depuneri pe stivă, aceste elemente depuse “după umplere” vor ajunge de fapt la baza stivei (jos, în Figura 2-4.1). Practic, zona de stivă este implementată ca un buffer circular.

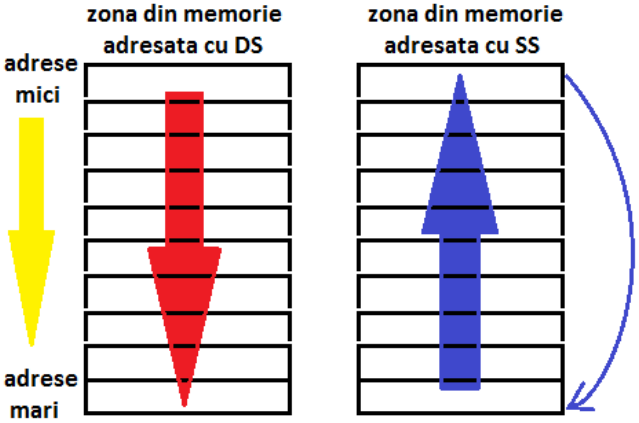


Figura 2-4.1. Depunerea elementelor **în memorie** (stânga) și **în stivă** (dreapta)

Elementele stivei nu sunt niciodată octeţi, doar cuvinte (**8086**↑), dublucuvinte (**80386**↑) sau cvadruplucuvinte (**Pentium 4**↑), așa cum prezintă Figura 2-4.2.

Figura 2-4.2. Tipuri de operanzi suportaţi de stivă, ocupând 2, 4 sau 8 octeţi

Instrucţiunile specifice lucrului cu stiva pot salva (sau pot extrage) în (din) stivă un singur operand (precum instrucţiunile PUSH și POP) sau un set de operanzi (precum instrucţiunile PUSHA[-/D] și POPA[-/D]).

2.4.1. Instrucţiunile PUSH şi POP

Instrucţiunea **PUSH** ***(Push Value onto Stack)*** depune în stivă un operand, iar instrucţiunea **POP** ***(Pop Value off Stack)*** extrage din stivă ultimul operand introdus. Dacă se dorește restaurarea unor valori din mai mulţi regiștri sau zone de memorie, acestea se vor depune pe stivă cu PUSH, iar la preluarea cu POP vor trebui extrași în ordine inversă faţă de cum au fost depuși.

PUSH sursă ***; depune elementul din sursă pe stivă***
PUSH {reg$_{16,32,64}$| mem$_{16,32,64}$| sreg| imed$_{16,32,64}$},

POP destinaţie ***; preia elementul de pe stivă în destinaţie***
POP {reg$_{16,32,64}$| mem$_{16,32,64}$| sreg}

La folosirea lui POP, efectul este invers lui PUSH, elementele extrăgându-se din stivă în operandul specificat în instrucţiune.

Observaţii:

- Instrucţiunile PUSH și POP *nu afectează flagurile*;
- Registrul ES, CS sau o valoare imediată nu poate apărea ca destinaţie;
- Sursa şi destinaţia sunt operanzi pe 16 biţi de la **8086** - **286** (extinse la 32 biţi de la **80386↑** şi apoi la 64 biţi de la **Pentium 4** şi **Core 2↑**), dar *niciodată octet*;
- De la **80386↑,** s-au adăugat regiștrii segment FS și GS, deci și aceștia pot fi folosiţi cu stiva;
- Valorile imediate în cadrul instrucţiunii *PUSH* sunt admise ca operanzi sursă începând de la **80186↑,** dar ca destinaţie (în instrucţiunea *POP*), niciodată;
- La lucrul cu stiva, în general se recomandă ca numărul operaţiilor POP să fie egal cu numărul operaţiilor PUSH în cadrul unui program, pentru ca stiva să nu se decaleze.

La lucrul cu stiva, vorbim de așezarea octeţilor în memorie tot după ***convenţia Little-END-ian***, exact ca în cazul datelor din cadrul segmentului de date.
La PUSH, pointerul (E)SP se actualizează la început şi abia apoi se depune conţinutul sursei pe stivă, în timp ce la POP, prima dată se extrag valorile din stivă și se depun în operandul destinaţie, şi abia apoi se actualizează (E)SP.

Începând cu **80186**, ca operand sursă sunt admise şi datele imediate, iar ***dimensiunea datelor*** imediate salvate pe stivă sau restaurate de pe stivă poate fi de 16 biţi sau începând cu **80386↑** și de 32 biţi (cuvinte sau dublucuvinte) în funcţie de specificatorii ***word*** sau ***dword***. Astfel, registrul SP sau ESP (se va incrementa/ decrementa automat cu 2 sau cu 4. Începând cu **Pentium 4↑**, elementele stivei pot fi şi cvadublucuvinte (cu specificatorul ***qword***), deci RSP cum se va numi acum (E)SP se va incrementa sau decrementa automat cu 8.

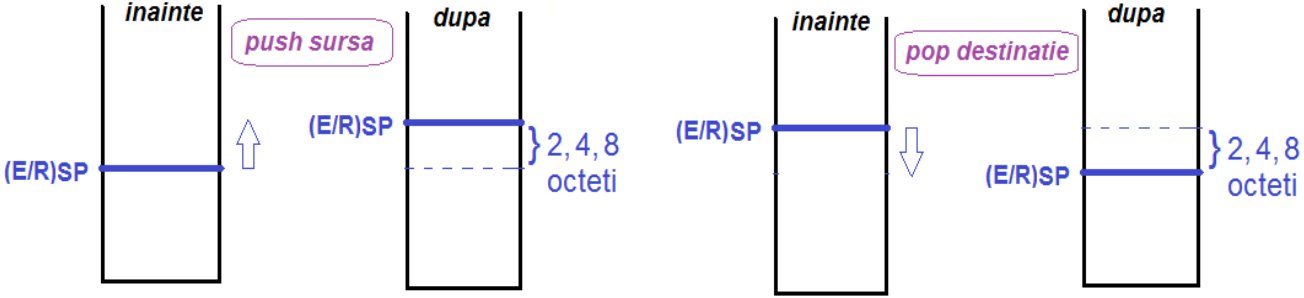


Figura 2-4.3. Ilustrarea modului de operare al instrucţiunii ***PUSH*** (stânga) și al instrucţiunii ***POP*** (dreapta)

PUSH sursă
[-/E/R]SP <- [-/E/R]SP – d
SS:[([-/E/R]SP)+0]<-Lowest8b (sursă)
…
SS:[([-/E/R]SP)+d-1]<-Highest8b (sursă)
d=2 la **8086↑**
d=4 la **386↑**
d=8 la **P4, Core2↑**

POP destinaţie
SS:[([-/E/R]SP)+0]->Lowest8b (destinaţie)
…
SS:[([-/E/R]SP)+d-1]->Highest8b (destinaţie)
[-/E/R]SP <- [-/E/R]SP + d
d=2 la **8086↑**
d=4 la **386↑**
d=8 la **P4, Core2↑**

PUSH sursă:
1. actualizează pointerul (E)SP – sau "îşi face loc"
2. depune conţinutul din sursă pe stivă

POP destinaţie:
1. ia conţinutul de pe stivă şi îl depune în destinaţie
2. actualizează pointerul (E)SP – sau "elimină locul gol"

Când are loc instrucţiunea ***push EAX*** de exemplu, la operarea pe 32 biţi,
1. registrul ESP se decrementează cu 4, iar apoi
2. la adresa SS:ESP+3 se depune octetul de rang 3 (format din biţii 31-24), deci octetul c.m.s.;
   la adresa SS:ESP+2 se depune octetul de rang 2 (format din biţii 23-16);
   la adresa SS:ESP+1 se depune octetul de rang 1 (format din biţii 15-8);
   la adresa SS:ESP+0 se depune octetul de rang 0 (format din biţii 7-0), adică c.m.p.s. octet.

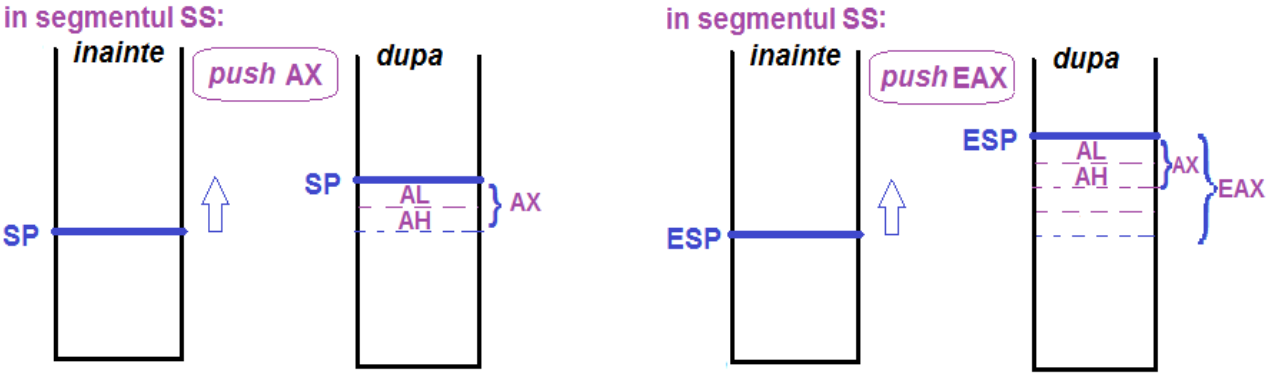


Figura 2-4.4. Ilustrarea modului de operare al instrucţiunii ***PUSH*** în mod pe 16 biţi (stânga) și în mod pe 32 biţi (dreapta)

Când se execută instrucţiunea ***pop EAX***, raţionamentul este invers:
1. de la adresele SS:ESP+3, SS:ESP+2, SS:ESP+1 și SS:ESP+0 se iau octeţii și se depun în EAX, iar apoi
2. se incrementează ESP cu 4.

2.4.2. Instrucţiunile PUSHA şi POPA (186↑)

De la **80186↑** au fost introduse instrucţiunile **PUSHA** ***(Push All 16-bit General-Purpose Registers)*** şi ***POPA*** ***(Pop All 16-bit General-Purpose Registers)*** care salvează pe stivă, respectiv restaurează de pe stivă toţi regiştrii generali de 16 biţi ai CPU. Dacă atributul dimensiunii operandului este 16 biţi, instrucţiunea PUSHA salvează regiştrii în ordinea AX, CX, DX, BX, SP, BP, SI, DI, ocupând 8*2 octeţi în stivă, iar instrucţiunea POPA îi reface în ordine inversă, cu specificarea că registrul SP nu e realmente refăcut (doar se adună 8 la valoarea curentă din registrul SP, continuând apoi cu încă 4 operaţii pop, deci încă +8).

PUSHA ***; depune conţinutul celor 8 regiștri generali de 16 biţi pe stivă***
POPA ***; preia conţinutul de pe stivă și îl depune în cei 8 regiștri generali de 16 biţi***

Observaţii:

- Instrucţiunile PUSHA și POPA *nu afectează flagurile*;

Operaţiile care au loc la execuţia instrucţiunii PUSHA, respectiv POPA sunt:

PUSHA: ***(1) TEMP <- SP; (2) PUSH AX; (3) PUSH CX; (4) PUSH DX; (5) PUSH BX; (6) PUSH TEMP; (7) PUSH BP; (8) PUSH SI; (9) PUSH DI;***

POPA: ***(1) POP DI; (2) POP SI; (3) POP BP; (4) SP=SP+8; (5) POP BX; (6) POP DX; (7) POP CX; (8) POP AX;***

2.4.3. Instrucţiunile PUSHAD şi POPAD (386↑)

De la **80386↑**, cele 2 instrucţiuni prezentate anterior au fost adaptate la tipul double, deci au fost introduse **PUSHAD** ***(Push All 32-bit General Purpose Registers)*** şi **POPAD** ***(Pop All 32-bit General Purpose Registers)*** care realizează aceleaşi operaţii similare lui PUSHA și POPA, dar folosind regiştrii corespunzători pe 32 biţi (EAX, ECX, EDX, EBX, ş.a.m.d.).

Pentru modul pe 64 biţi, instrucţiunile PUSHAD şi POPAD nu au fost definite (nu funcţionează).

PUSHAD ***; depune conţinutul celor 8 regiștri generali de 32 biţi pe stivă***
POPAD ***; preia conţinutul de pe stivă și îl depune în cei 8 regiștri generali de 32 biţi***

Observaţii:

- Instrucţiunile PUSHAD și POPAD *nu afectează flagurile*;
- Instrucţiunea PUSHAD necesită 8*4=32B de spaţiu pe stivă pentru stocarea a 8 regiştri de 32 biţi fiecare;
- Valoarea preluată din stivă pentru ESP este ignorată(se reface ca ESP+4*4, şi apoi se mai adună 4 la fiecare pop).

Operaţiile care au loc la execuţia instrucţiunii PUSHAD, respectiv POPAD sunt:

PUSHAD	POPAD
TEMP <- ESP	
PUSH EAX	POP EDI
PUSH ECX	POP ESI
PUSH EDX	POP EBP
PUSH EBX	ESP=ESP+16
PUSH TEMP	POP EBX
PUSH EBP	POP EDX
PUSH ESI	POP ECX
PUSH EDI	POP EAX

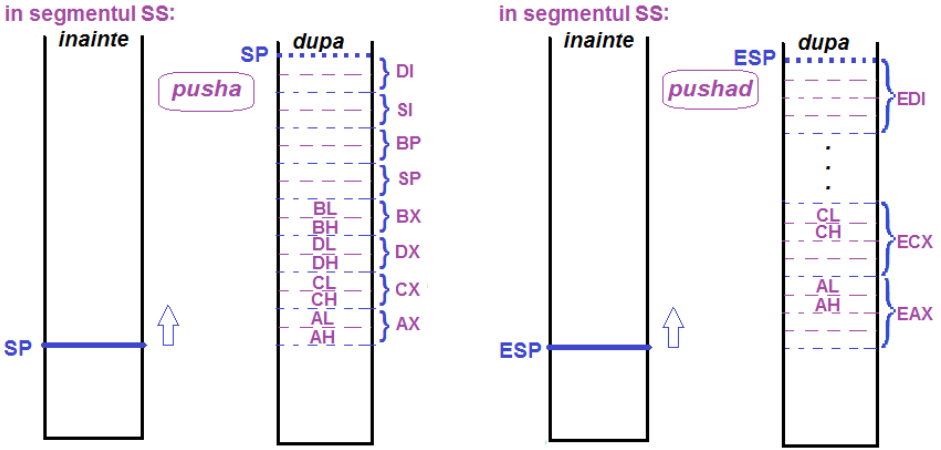


Figura 2-4.5. Ilustrarea modului de operare al instrucţiunii ***PUSHA*** pe 16 biţi (stânga) și al instrucţiunii ***PUSHAD*** pe 32 biţi (dreapta)
Când se operează în modul pe 32 biţi, operanzii din memorie (care nu au specificat explicit tipul) sunt consideraţi de 32 biţi. Similar, la funcţionarea în mod pe 64 biţi, aceștia vor fi consideraţi implicit pe 64 biţi.

2.4.4. Exemple:

Exemple de *instrucţiuni ilegale:*

```
pop ES          ; nu se admite registrul ES ca și destinaţie, nici [-/E/R] IP
pop CS          ; nu e indicat registrul CS ca și destinaţie
push 1234h      ; admisă doar de la 80186↑
pop 1234h       ; nu e admisă niciodată o valoare imediată ca destinaţie
pop AH          ; nu se pot folosi octeţi la lucrul cu stiva
pusha AX        ; instrucţiunea PUSHA nu are operanzi
popa AX         ; instrucţiunea POPA nu are operanzi
pushad EAX      ; instrucţiunea PUSHAD nu are operanzi
popad EAX       ; instrucţiunea POPAD nu are operanzi
```

Exemple de *instrucţiuni legale*:

Exemple 2-4.1

```
push BX         ; SP se decrementează cu 2, iar apoi (SS:[SP]) = BX; deci conţinutul reg. BX este depus în stivă,
push EBX        ; se foloseşte registru pe 32 biţi, din ESP se scade 4 (mod pe 32 biţi)
push RBX        ; mod pe 64 biţi, se folosește registru pe 64 biţi, RSP se decrementează cu 8 (mod pe 64 biţi)
```

Exemple 2-4.2

```
push VAR        ; conţinutul memoriei VAR e depus în stivă la locaţia SS:[SP], VAR obligatoriu de tip
                ; cuvânt la 8086, dar poate fi și dublucuvânt (de la 386↑) sau cvadruplucuvânt (de la P4↑)
push valmem32   ; conţinutul memoriei pe 32 biţi (de la 386↑) e depus în stivă
push valmem64   ; conţinutul memoriei pe 64 biţi (de la P4↑) e depus în stivă
```

Exemple 2-4.3

```
push [BX]       ; cuvântul din memorie, adresat de BX (în segmentul DS) se depune în stivă, dar mai corect ar fi:
push word ptr [BX]  ; cu specificarea tipului: (SS:[SP]) = (DS:[BX])
push qword ptr [RBX]  ; depune în stivă cvadruplucuvântul adresat de RBX
```

Exemple 2-4.4

```
push 'a'        ; depune pe stivă codul Ascii al caracterului a, scris pe 16 biţi[3]
push ','        ; depune pe stivă codul Ascii al caracterului ,, scris pe 16 biţi[3]
push 2          ; depune pe stivă 2, scris ca word, adică 0002h
pushw 10h       ; depune pe stivă 10h, scris ca word, adică 0010h[4]
pushd 10h       ; depune pe stivă 10h, scris ca doubleword: 00000010h
```

Exemple 2-4.5 Lucrul cu regiștrii segment

```
push DS         ; se depune pe stivă valoarea din registrul segment DS
pop DS          ; preia de pe stivă o valoare pe 16 biţi și se stochează în registrul segment DS (de la 8086↑)
pop FS          ; preia de pe stivă o valoare pe 16 biţi și se stochează în registrul segment FS (de la 386↑)
```

Exemple 2-4.6

```
pop valmem64    ; se preiau 8 octeţi de pe stivă și se depun în memorie, la adresa dată de valmem64 (mod pe 64 biţi)
pop valmem32    ; se preiau 4 octeţi de pe stivă și se depun în memorie, la adresa dată de valmem32 (mod pe 32 biţi)
pop word ptr [BX]  ; se preiau 2 octeţi de pe stivă și se depun în memorie, la adresa dată de conţinutul registrului BX
                                                                                    (mod pe 16 biţi)
```

Exemplul 2-4.7

```
pusha           ; se salvează pe stivă AX, CX, DX, BX, SP, BP, SI, DI
pop VAR         ; conţinutul registrului DI e depus în memorie la adresa dată de VAR
```

Exemplul 2-4.8 Fie o secvenţă de cod formată din 4 blocuri de instrucţiuni, aşa cum se poate observa în Figura 2-4.6. Primul bloc conţine doar instrucţiuni ***mov*** (pentru a fixa datele de intrare ale problemei). Interesant de urmărit este blocul II şi IV, unde se subliniază efectul instrucţiunilor ***pusha*** şi ***popa***. După execuţia instrucţiunii ***pusha*** (adică „blocul II"), se observă pe stivă, de jos în sus, următoarele: conţinutul registrului AX, apoi CX, DX, BX, SP, BP, SI şi în final DI.

---

[3] TASM va depune doar octetul c.m.p.s. ca valoare *operand pe 8 biţi*, octetul c.m.s. fiind porţiunea AH din registrul AX; simulatorul EMU8086 nu suportă acest format de instrucţiune

[4] valoarea operandului se extinde cu semn la dimensiunea specificată în instrucţiune;

Pentru SP observăm că s-a depus pe stivă valoarea FFFEh, adică valoarea lui SP dinaintea execuţiei instrucţiunii pusha (sau de după execuţia blocului I). De fapt, valoarea reală a lui SP după execuţia a 5 operaţii push, ar fi trebuit să fie FFF4h.
După execuţia instrucţiunii ***pusha*** (adică după ce s-au depus pe stivă toate cele 8 cuvinte) SP a descrescut cu 16, deci SP=FFFEh-16 = FFEEh. În instrucţiunile blocului III, se curăţă toţi regiştrii (presupunând că valorile acestora au fost alterate de un bloc oarecare din program). După execuţia instrucţiunii ***popa*** (adică după execuţia blocului IV) se observă că valoarea lui SP va fi cea preluată din stivă şi nu cea calculată după 3 operaţii pop, adică FFEEh+6=FFF4h.

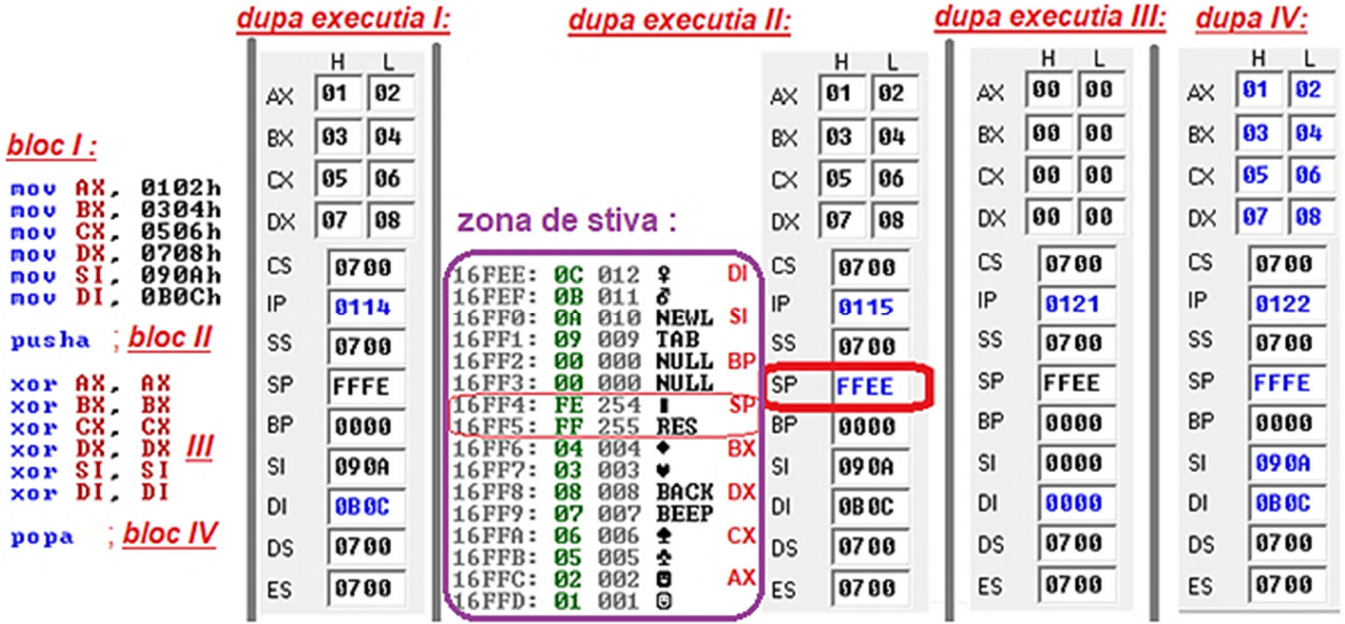


Figura 2-4.6. Ilustrarea modului de execuţie al secvenţei de cod de la exemplul 2-4.8

Exemplul 2-4.9

Să presupunem că stiva a fost definită între locaţiile 10000h şi 1FFFFh pentru un **procesor 8086**; la iniţializarea stivei, se va încărca registrul segment SS cu 1000h şi registrul SP cu 0000h (deci dimensiunea stivei este în acest caz 64KB).

La execuţia primei instrucţiuni de tip PUSH (cu operand de tip word în acest caz) valoarea din SP va deveni SP=FFFEh, ceea ce în convenţia numerelor fără semn înseamnă 65.534.

Se observă astfel că diferenţa între cele 2 adrese 10000h şi 1FFFEh sugerează faptul că se mai pot depune încă 65534 octeţi pe stivă până la umplere sau că se pot efectua 32767 operaţii PUSH (în ipoteza că nu am folosi şi instrucţiuni POP).

Totuşi, ***segmentele de stivă se comportă ciclic***, ca nişte ***buffere circulare***: dacă se atinge extrema (în cazul nostru SP ajunge treptat de la FFFEh, la FFFCh, ..., apoi la 0002h şi apoi la 0000h), SP va relua apoi parcurgerea zonei de memorie (adică a stivei) din cealaltă extremă: SP va deveni din nou FFFEh, şi deci s-ar suprascrie vechiul conţinut.

În practică foarte rar se întâmplă astfel de cazuri, întrucât cu stiva se folosesc şi instrucţiuni POP în alternanţă cu cele PUSH, iar instrucţiunile POP readuc valoarea pointerului înapoi, înspre adrese mai mari.

Valoarea pointerului BP este în general iniţializată cu BP=0000h tocmai pentru a indica acest comportament ciclic al stivei: ***când SP devine egal cu BP se consideră că stiva s-a umplut***.

A. **Ce se întâmplă când stiva se umple? Unde se depune informaţia după ce stiva e plină?**
B. **Ce se întâmplă dacă stiva e goală şi luăm informaţie din ea? Ce informaţie va conţine stiva dacă prima instrucţiune ce se execută cu stiva este o instrucţiune pop ?**

A. În cadrul unui program de tip .COM să presupunem că SS=DS=CS=ES=0700h, iar la început se iniţializează SP=FFFEh şi BP=0000h. Pentru a demonstra comportamentul ciclic al stivei, să presupunem că în cadrul acestui program de tip .COM stiva este aproape plină (pointerul SP a ajuns aproape de limita superioară). Pentru a urmări îndeaproape lucrul cu stiva, în cadrul programului nostru, am ales ca prima instrucţiune să fie una care să modifice pointerul SP, să îl aducă cât mai aproape de partea de sus a stivei. Apoi, se fac mai multe depuneri pe stivă şi se observă că pointerul sare automat de la valoarea 0000h la FFFEh, deci sare din partea de sus (numită şi „vârf"), în partea de jos (numită şi „bază").

Secvenţa de program propusă la secţiunea A:	Iniţial SS=DS=CS =ES=0700h (rămân neschimbate)	După execuţia ***mov SP,0002h***	După execuţia ***push 1234h*** ***(stiva tocmai s-a umplut)***	După execuţia ***push 5678h*** ***(pointerul sare înapoi jos)***	După execuţia ***push 9abcdh***
mov SP,0002h **push 1234h** **push 5678h** **push 9abch**	BP=0000h, **SP=FFFEh,** Deci pointerul la vârful stivei este la locaţia 0700:FFFEh	BP=0000h, **SP=0002h,**	BP=0000h, **SP=0000h,** (la 07000h va depune 34h şi la 07001 va depune 12h)	BP=0000h, **SP=FFFEh,** (la 16FFEh va depune 78h şi la 16FFFFh va depune 56h)	BP=0000h, **SP=FFFCh,** (la 16FFCh va depune BCh şi la 16FFFDh va depune 9Ah)

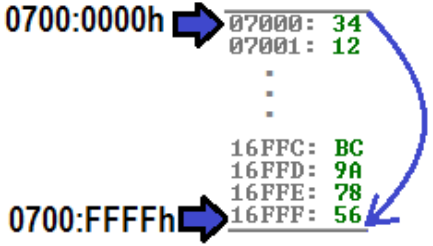


Figura 2-4.7. Ilustrarea modului de operare al stivei ca un buffer circular la PUSH

B. Invers, să presupunem că se foloseşte un program de tip .COM şi că prima instrucţiune care foloseşte stiva din acest program este o instrucţiune POP şi nu una PUSH, sau mai bine că se efectuează o instrucţiune PUSH, urmată de 2 instrucţiuni POP.

Secvenţa de program propusă la secţiunea B:	Iniţial SS=DS=CS =ES=0700h (rămân neschimbate)	După execuţia ***push 1234h***	După execuţia ***pop AX***	După execuţia ***pop BX*** ***(pointerul sare înapoi sus)***	După execuţia ***pop CX***
push 1234h **pop AX** **pop BX** **pop CX**	BP=0000h, **SP=FFFEh,** Deci pointerul la vârful stivei este la locaţia 0700:FFFEh	BP=0000h, **SP=FFFCh,** (la 16FFCh va depune 34h şi la 16FFD va depune 12h)	BP=0000h, în AX va fi 1234h (luat de la 16FFCh şi 16FFDh) şi apoi SP devine **SP=FFFEh**,	BP=0000h, în BX va fi 0000h (luat de la 16FFEh şi 16FFFh) şi apoi SP devine **SP=0000h,**	BP=0000h, în BX va fi 20CDh (luat de la 07000h şi 07001h) şi apoi SP devine **SP=0002h,**

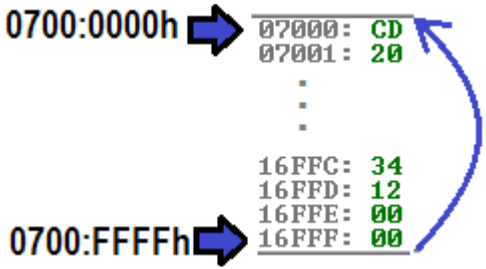


Figura 2-4.8. Ilustrarea modului de operare al stivei ca un buffer circular la POP

Confuzia poate să apară deoarece partea de sus din Figurile 2.4-7 şi 2.4-8 se numeşte „baza” (şi nu partea de jos); totuşi, fiind vorba de un buffer circular, locaţia cea mai de jos e cea „de deasupra” locaţiei de sus şi invers.

Exemplul 2-4.10 La un CPU 8086, presupunând că SS=0700h şi SP=FFFEh, se dă următoarea secvenţă de instrucţiuni:

```
org 100h
mov AX, 1234h        ; AX=1234h
mov BX, 5678h        ; BX=5678h
push AX              ; se decrementează SP cu 2: SP=0FFFCh, iar în locaţiile
                     ; SS:(SP+1) şi SS:(SP) se depune cuvântul din AX;
push BX              ; similar SP=0FFFAh, (SS:SP)=5678h (LittleEnd-ian)
```

în urma execuţiei secvenţei de instrucţiuni, stiva va arăta așa ca în Fig. 2-4.9

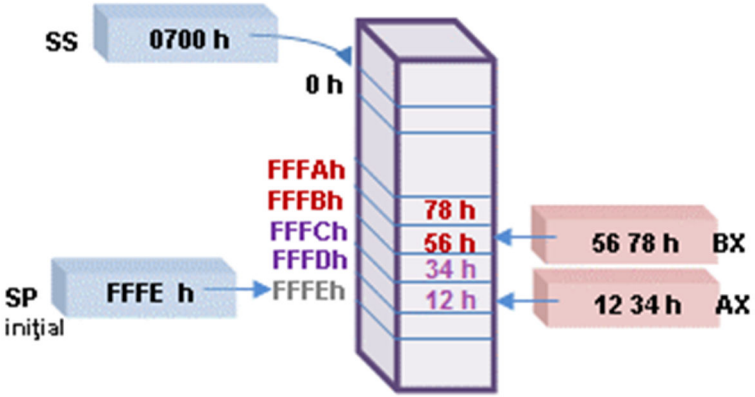


Figura 2-4.9. Ilustrarea exemplului 2-4.10: depunerea pe stivă a două elemente pe cuvânt (1234h și 5678h)

Exemplul 2-4.11 Problema anterioară, adaptată la un CPU pe 32 biţi, se poate transcrie: se dă următoarea secvenţă de instrucţiuni:

```
mov EAX,01234567h      ; EAX=01234567h
mov EBX, 89ABCDEFh     ; EBX=89ABCDEFh
push EAX               ; se decrementează ESP cu 4, iar în locaţiile
                       ; SS:(ESP+3) , SS:(ESP+2), SS:(ESP+1) şi SS:(ESP) se depune dublucuvântul din EAX;
push EBX               ; similar ESPse mai decrementează cu 4, iar (SS:ESP)=89ABCDEFh (LittleEnd-ian)
```

în urma execuţiei secvenţei de instrucţiuni, stiva va arăta așa ca în Fig. 2-4.10

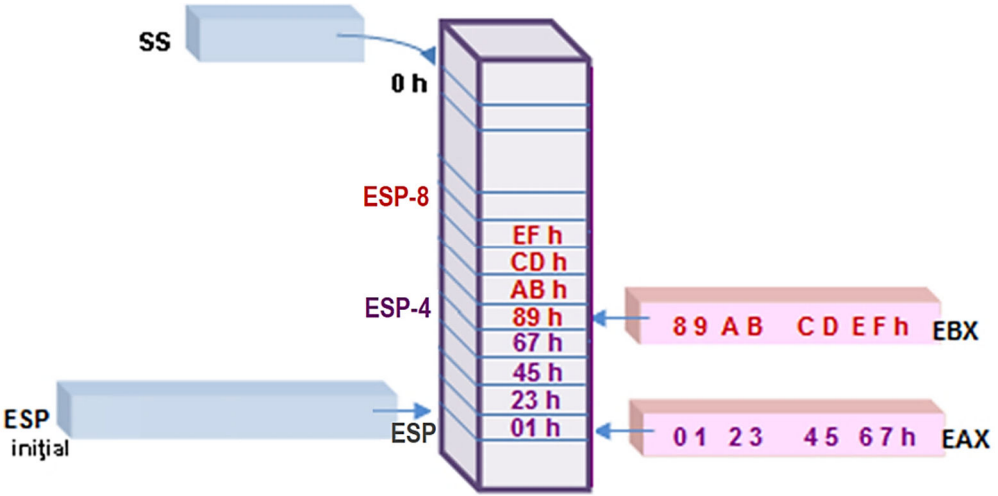


Figura 2-4.10. Ilustrarea exemplului 2-4.11: depunerea pe stivă a două elemente pe dublucuvânt (01234567h și 89ABCDEFh)

Exemplul 2-4.12 Presupunând că în secvenţa de la exemplul 2-4.10 se adaugă și două instrucţiuni pop, ce se va găsi în regiștrii CX și DX și cum va arăta stiva?

```
org 100h
mov AX,1234h        ; AX=1234h
mov BX, 5678h       ; BX=5678h
push AX             ; se decrementează SP cu 2: SP=0FFFCh, iar în locaţiile
                    ; SS:SP+1 şi SS:SP se depune cuvântul din AX;
push BX             ; similar SP=0FFFAh, apoi (SS:SP)=5678h (LittleEnd-ian)
pop CX              ; se depune cuvântul din locaţiile SS:(SP-1) şi SS:SP în CX
                    ; deci CX=5678h iar apoi se incrementează SP cu 2: SP=0FFFCh
pop DX              ; similar, DX=1234h (Little End-ian) și apoi SP=0FFFEh,
```

în urma execuţiei secvenţei de instrucţiuni, stiva va arăta așa ca în Figura 2-4.11:

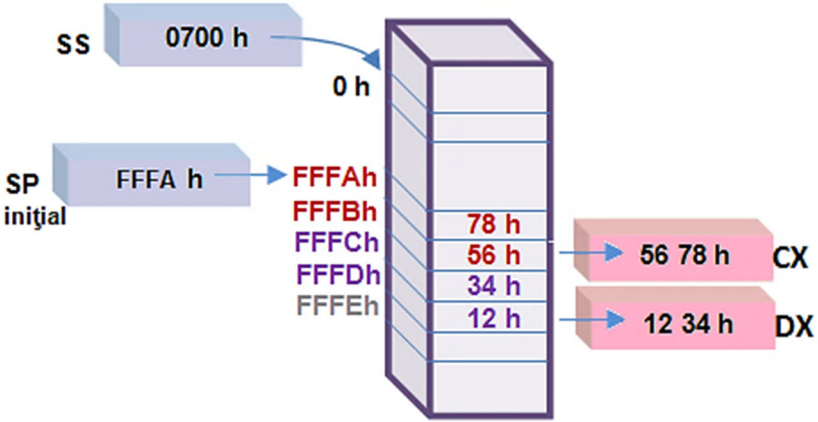


Figura 2-4.11. Ilustrarea exemplului 2-4.12: preluarea de pe stivă a 2 elemente pe cuvânt (5678h și 1234h)

Exemplul 2-4.13 Presupunând că se execută următoarea secvenţă de cod, care vor fi valorile elementelor şirului sird ?

```
org 100h
.data
sirs dw 1,2,3,4
sird dw 4 dup(?)
.code
push sirs[0]
push sirs[2]
push sirs[4]
push sirs[6]
pop sird[0]
pop sird[2]
pop sird[4]
pop sird[6]
```

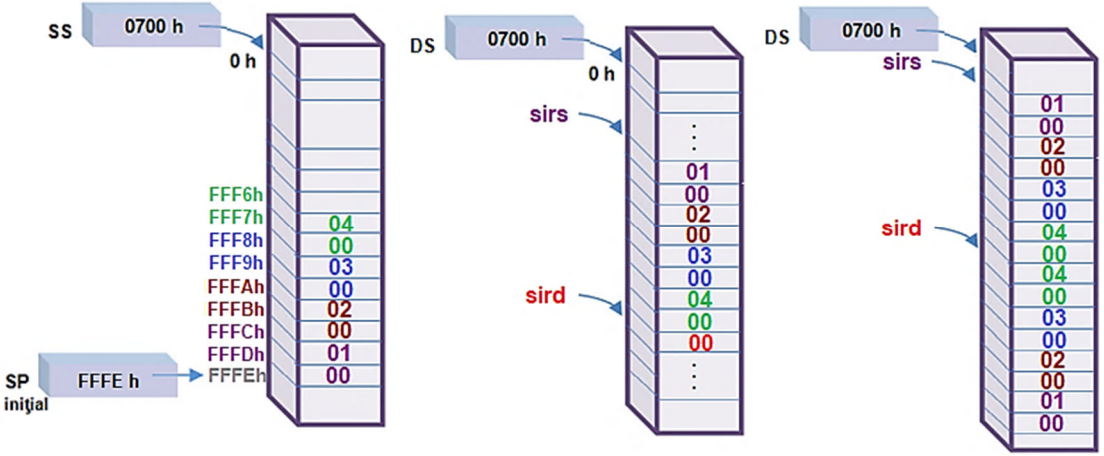


Figura 2-4.12. Ilustrarea exemplului 2-4.13: inversarea unui şir de elemente pe cuvânt prin stivă

Exemplul 2-4.14 Folosind un CPU de 32 biţi, se presupune că se doreşte transferul unui dublucuvânt dintr-un registru de 32 biţi într-o pereche de regiştri de câte 16 biţi fiecare (sau invers, dintr-o pereche de regiştri de 16 biţi într-un registru de 32 biţi).

Transfer din EAX în DX:AX

```
mov EAX, 12345678h  ; EAX = 12345678h
push EAX            ; se depune pe stivă,
        ; de jos în sus: 12h, 34h, 56h, 78h
pop AX              ; AX = 5678h
pop DX              ; DX = 1234h
```

Transfer din DX:AX în EAX

```
mov DX, 1234h       ; DX = 1234h
mov AX, 5678h       ; AX = 5678h
push DX         ; se depune pe stivă 12h, 34h
push AX         ; se depune pe stivă 56h, 78h
pop EAX             ; EAX = 12345678h
```

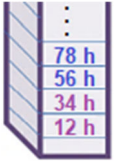


Figura 2-4.13. Ilustrarea exemplului 2-4.14

2.5. Instrucţiuni de transfer cu acumulatorul

Instrucţiunile de transfer cu ACC abordate în această secţiune sunt XLAT(B) și cele pentru lucrul cu porturi IN și OUT.

Tot în această categorie (a instrucţiunilor ce folosesc ACC), s-ar fi putut include și instrucţiunile pentru extinderea sau corecţia ACC, dar acestea vor fi abordate în Capitolul 3.

2.5.1. Instrucţiunea XLAT şi XLATB

Instrucţiunea **XLAT** ***(Translate)*** aparţine setului de instrucţiuni original al **8086**; aceasta translatează, adică aduce în AL conţinutul octetului din memorie, de la adresa fizică sau locaţia DS: [BX+AL].

Este singura instrucţiune care adună un număr pe 8 biţi cu unul pe 16 sau 32 biţi.

XLAT ; AL ← (DS : ((E)BX + AL)) ; aduce în AL conţinutul din memorie de la adresa dată de (E)BX+AL

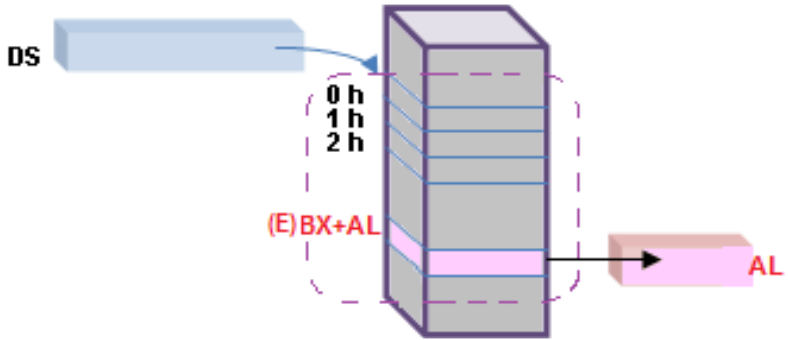


Figura 2-5.1. Ilustrarea modului de operare al instrucţiunii ***XLAT***

Observaţii:

- Instrucţiunea XLAT *nu afectează flagurile*;
- Instrucţiunea nu are operanzi și întotdeauna se folosește registrul acumulator inferior (adică AL) pentru a depune informaţia preluată din memorie;
- Poziţionarea corectă și apoi gestionarea pointerului dat de valoarea BX+AL este în sarcina programatorului.

Instrucţiunea **XLATB** a fost suportată de la **80386↑,** fiind o adaptare a instrucţiunii XLAT: aceasta translatează, adică aduce în AL conţinutul octetului din memorie, de la adresa fizică sau locaţia DS: [EBX+AL],

- de la **Pentium 4** şi **Core 2↑,** aceasta afectează conţinutul din memorie de la locaţia dată de [RBX+AL].

Instrucţiunea **XLAT** sau **XLATB** manipulează întotdeauna un singur octet, cel de la locaţia dată de adresa RBX+AL.

Este utilă în conversia unor tipuri de date (în special între valori binare și coduri Ascii sau valori hexazecimale) şi se foloseşte împreună cu tabele de translatare. În aceste cazuri, poziţia elementelor în şir este de o foarte mare importanţă.

Cea mai cunoscută utilizare este la conversia şi afişarea unui număr din zecimal în hexazecimal (deci obţinerea cifrelor hexazecimale ale numărului ca şi coduri Ascii) sau la diverse codificări, de exemplu codificarea Gray.
O altă posibilă aplicaţie o reprezintă codificarea în binar a segmentelor (ale unui afişaj cu 7 segmente) ce trebuie aprinse pentru a se afişa o anumită cifră pe afişaj.

2.5.2. Instrucţiunile IN şi OUT

Instrucţiunile de transfer cu porturile **IN** ***(Input Data* from Port)** şi **OUT** ***(Output Data to Port)***, așa cum au fost ele implementate în setul original de instrucţiuni al **8086**, oferă posibilitatea de a schimba informaţii cu perifericele prin intermediul porturilor de I/O, transferurile fiind realizate din sau în acumulator, pe octet (AL) sau cuvânt (AX);

de la **80386**↑, instrucţiunile IN şi OUT au fost adaptate astfel încât să se poată folosi şi acumulatorul extins (EAX), deci pentru a transfera pe port date de dimensiune doubleword.

În cadrul instrucţiunilor (IN sau OUT) trebuie specificată adresa portului cu care se lucrează (cu care se dorește transferul). Această adresă poate fi evidentă în cadrul instrucţiunii (deci e dată explicit) sau poate fi mai puţin vizibilă (fiind specificată în registrul DX) – în general, dacă adresa portului e o valoare mai mare decât FFh. Sursa și destinaţia nu pot fi orice tip de registru, ci doar registrul accumulator: AL, AX sau EAX.

În mod pe 64 biţi, instrucţiunea se comportă similar ca în mod protejat pe 32 biţi.

IN destinaţie, port ; citeşte informaţie la nivel de octet/ cuvânt/ dublucuvânt de pe portul de intrare
IN { AL|AX|EAX },{ adresa imediată periferic pe 8 biţi | DX }

OUT port, sursă ; scrie informaţie la nivel de octet/ cuvânt/ dublucuvânt pe portul de ieşire
OUT { adresa imediată periferic pe 8 biţi | DX } , { AL| AX| EAX }

Adresa portului poate fi:

- dată explicit ca o valoare imediată, pentru primele 256 de porturi
  adrese de port între 00h÷FFh (sau în zecimal 0÷ 255) - ***adresare fixă cu portul*** - adresa portului nu se modifică;
- pentru adrese mai mari, dată prin intermediul registrului DX
  adrese de port între 0000h÷FFFFh şi se numeşte ***adresare variabilă cu portul***-adresa din DX se poate modifica

Adresa portului apare pe busul de adresă pe durata unei operaţii I/O.

De exemplu, la execuţia instrucţiunii ***in AL,46h;*** *datele* de la portul cu adresa 46h sunt preluate în registrul AL prin *busul de date*, iar *adresa* apare ca o valoare 0046h pe 16 biţi, pe pinii A15- A0 ai *busului de adresă*.

Biţii A19-A16 (la **8086**/ **8088**), A23-A16 (**80286**/ **80386SX**), A24-A16 (**80386SL**/ **80386SLC**/ **80386EX**) sau A31-A16 (**80386–Core2**) sunt nedefiniţi pentru instrucţiuni IN şi OUT.

Observaţii:

- Instrucţiunile IN și OUT *nu afectează flagurile*;
- Operandul destinaţie (la IN), respectiv cel sursă (la OUT) e întotdeauna ACC, iar celălalt operand e fie o valoare imediată, fie registrul DX;
- La cele 2 instrucţiuni IN și OUT, trebuie acordată o deosebită atenţie întrucât sintaxa lor este inversată.

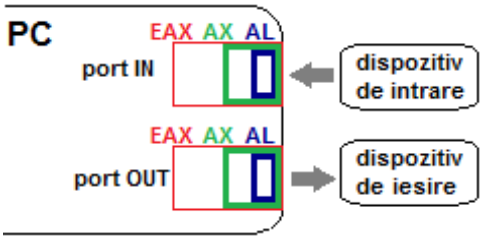


Figura 2-5.2. Ilustrarea modului de operare al instrucţiunilor ***IN*** și ***OUT***

Începând de la **80186**, există instrucţiunile **INS** şi **OUTS** care lucrează cu porturile la nivel de şiruri, însă acestea nu folosesc ACC, deci nu vor fi abordate aici (sunt descrise în Capitolul 5).

2.5.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
xlat AX          ; xlat nu are operanzi

in BX, DX        ; nu se poate folosi alt registru decât ACC
in AH, DX        ; nu se poate folosi registrul AH ca operand destinaţie la instrucţiunea IN
in RAX, DX       ; nu s-a implementat o astfel de instrucţiune pe 64 biţi
in AL, 378h      ; valoarea imediată nu începe pe 8 biţi, deci trebuie specificată prin intermediul registrului DX
in DX, AL        ; sintaxă eronată
out AL, DX       ; sintaxă eronată
```

Exemple de *instrucţiuni legale*:

Exemplul 2-5.1 Dacă se dorește obţinerea codului Ascii al unei litere din alfabet – pe baza numărului de ordine al ei în alfabet, se poate folosi următoarea secvenţă de instrucţiuni:

```
sir  DB  'ABCDEF'    ; se defineşte variabila sir de tip şir de octeţi,
                     ; iniţializată cu codurile Ascii ale caracterelor A,...,F
                     ; codurile Ascii ale literelor sunt valori consecutive:
                     ; 41h-,A', 42h-,B',etc şi 61h-,a',62h-,b', etc
...
mov BX,offset sir    ; se încarcă în BX adresa efectivă a lui sir,
mov AL, 3            ; AL=03h
xlat                 ; la execuţie, xlat e înlocuită cu xlatb =>
                     ; AL=(DS:(BX+AL)), adică 44h=cod Ascii al lui ,D'
```

Se folosește mai ales la tipărire, atunci când e nevoie de codul Ascii al caracterului sau numărului/ valorii numerice întregi de afișat ca cifră hexazecimală; având valoarea numerică într-un registru, de exemplu restul împărţirii la 16, aceasta s-a depus în AL şi apoi s-a executat instrucţiunea xlat; astfel, s-a obţinut codul Ascii ce trebuie folosit la afișare.

Exemplul 2-5.2
În segmentul de date s-a definit variabila varGray cu elemente de tip octet, fiecare octet având o anumită semnificaţie: în funcţie de poziţia lui în şir, acesta reprezintă codificarea Gray a codului binar pe 4 biţi.

Codificarea Gray aparţine codurilor ciclice sau reflexive:
bitul 0 este reflectat în fiecare grup de 4 coduri,
bitul 1 este reflectat în fiecare grup de 8 coduri, iar
bitul 2 este reflectat în fiecare grup de 16 coduri.

Astfel, după cum se poate observa şi din Figura 2-5.3, codificarea lui 4 este 6, a lui 8 este Ch ş.a.m.d.

```
 .data
varGray db 0h, 1h, 3h, 2h, 6h, 7h, 5h, 4h, 0Ch, 0Dh,
0Fh, 0Eh, 0Ah, 0Bh, 9h, 8h   ; variabila din memorie

.code
mov AL,4
mov BX, offset varGray
xlat                         ; în AL se va găsi codul
                             ; Gray corespunzător lui 4,
                             ; adică 6h=0000 0110b
```

Cod zecimal	Cod binar				Cod Gray				
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	1
2	0	0	1	0	0	0	1	1	3
3	0	0	1	1	0	0	1	0	2
4	0	1	0	0	0	1	1	0	6
5	0	1	0	1	0	1	1	1	7
6	0	1	1	0	0	1	0	1	5
7	0	1	1	1	0	1	0	0	4
8	1	0	0	0	1	1	0	0	C
9	1	0	0	1	1	1	0	1	D
10	1	0	1	0	1	1	1	1	F
11	1	0	1	1	1	1	1	0	E
12	1	1	0	0	1	0	1	0	A
13	1	1	0	1	1	0	1	1	B
14	1	1	1	0	1	0	0	1	9
15	1	1	1	1	1	0	0	0	8

Figura 2-5.3. Ilustrarea metodei de codificare Gray pe 4 biţi

Exemplul 2-5.3 În segmentul de date s-a definit variabila var7segm cu elemente de tip octet, fiecare octet având o anumită semnificaţie: în funcţie de poziţia lui în şir, octetul reprezintă codificarea modului în care trebuie activate (adică vor fi aprinse) segmentele unui afişaj cu 7 segmente.

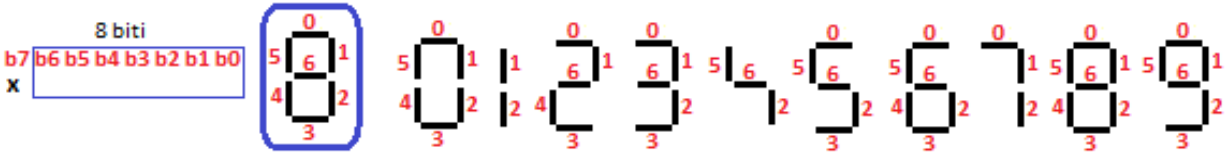


Figura 2-5.4. Ilustrarea modului de lucru cu un afişaj 7-segmente

```
.data
var7segm db 3Fh, 06h, 5Bh, 4Fh, 66h, 6Dh, 7Dh, 07h, 7Fh, 6Fh      ; variabila pt codificarea pe 7 segmente
.code
mov AL,2
mov BX, offset var7segm
xlat                    ; în AL se va găsi codul pentru generarea cifrei 2
```

Exemple 2-5.4

```
in  AL, 60h             ; AL = (port 60h) se preia în registrul AL octetul de la portul cu adresa 60h
in AX, 60h              ; se preia de pe portul cu adresa 60h un cuvânt și se depune în AX
in EAX, 60h             ; se preia de pe portul cu adresa 60h un dublucuvânt și se depune în EAX
```

Exemplul 2-5.5

```
in  EAX, DX             ; se preia de pe portul ce are adresa specificată în reg. DX un dublucuvânt şi se stochează în EAX
```

Exemplul 2-5.6

```
out DX, EAX             ; dublucuvântul din EAX este trimis pe portul ce are adresa specificată în registrul DX
```

Exemplul 2-5.7

```
mov  DX, 37Ah           ; adresa portului este >256, a.î. se foloseşte registrul DX pentru specificarea adresei de port
out  DX, AL             ; (port 37Ah) =AL; trimite octetul din registrul AL pe portul cu adresa 37Ah
```

2.6. Instrucţiuni de transfer pentru adrese

Această categorie de instrucţiuni este utilă atunci când se operează cu adresele operanzilor din memorie. Instrucţiunile LEA, LDS și LES au fost suportate încă de la **8086↑**, dar instrucţiunea generalizată LxS, cu x=F,G,S a fost suportată de la **386↑**.

2.6.1. Instrucţiunea LEA

Instrucţiunea **LEA** ***(Load Effective Address)*** copiază adresa efectivă a sursei (operand aflat în memorie) în registrul general de 16, 32 sau 64 biţi specificat în instrucţiune. Operaţia se realizează în faza de execuţie a programului.

LEA destinaţie, sursă; ***; destinaţie = offset sursă***
LEA {reg$_{16,32,64}$}, {mem$_{16,32}$} unde *mem* este operand aflat în memorie.

Observaţii:

- Instrucţiunea LEA *nu afectează nici un flag*.
- Operandul destinaţie nu poate fi decât registrul de uz general, exclus registru segment, [-/E/R] IP sau [-/E/R] FLAGS.

Începând cu **80386↑**, registrul destinaţie poate fi şi pe 32 biţi (deci EAX, etc), iar de la **P4↑**, acesta poate fi şi de 64 biţi.
Directiva offset realizează aceeaşi operaţie ca instrucţiunea LEA, dacă operandul e doar deplasament.
Adresa efectivă se mai poate obţine și folosind operatorul offset și instrucţiunea mov.

(Atribuirea se realizează în faza de asamblare a programului.)

```
mov DI, offset val
```

2.6.2. Instrucţiunile LDS şi LES, respectiv LxS

Instrucţiunile **LDS** ***(Load Data Segment)*** şi **LES** ***(Load Extra Segment)*** transferă o adresă completă într-o pereche de regiştri: perechea *DS:registru*, respectiv *ES:registru* este încărcată cu adresa completă de 32 sau 48 de biţi (în funcţie de dimensiunea registrului) conţinută în sursă (definită ca operand în memorie). Această adresă se încarcă a.î. cuvântul sau dublucuvântul mai puţin semnificativ (adresa offset) va fi în "registru", iar cuvântul mai semnificativ (adresa de segment) va fi în DS, respectiv ES.

LDS registru, sursă; ***LES registru, sursă;***
LDS { reg$_{16,32}$}, { mem$_{16,32}$} *LES { reg$_{16,32}$}, { mem$_{16,32}$}*

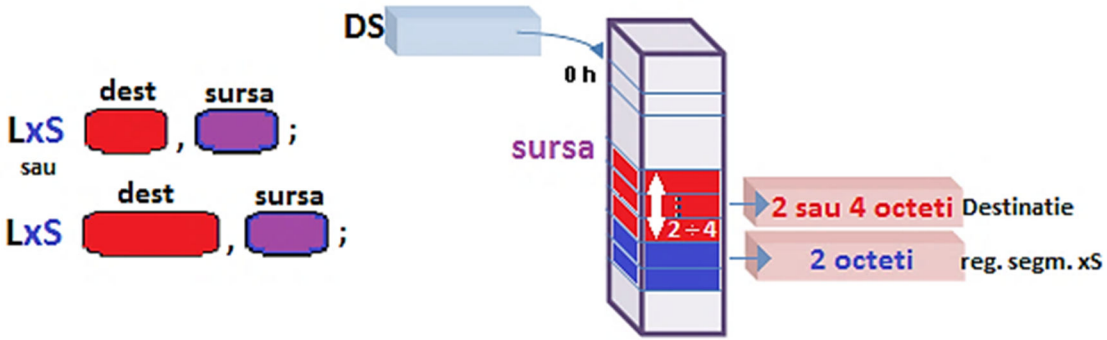


Figura 2-6.1. Ilustrarea modului de operare al instrucţiunii ***LxS***

Începând cu **80386↑**, instrucţiunile LDS şi LES iau forma generală:

LxS dest, sursa, **cu x={D,E,F,G,S}**

(deci se pot folosi și regiştrii SS, FS şi GS), putând încărca perechea de regiştri cu **un pointer pe 4 octeţi** sau cu **un pointer pe 6 octeţi,** în funcţie de dimensiunea offsetului (pe ***16 biţi*** sau ***32 biţi***). Aceste instrucţiuni sunt utile când se salvează în memorie o adresă de tip *near* sau *far* ca **$segment_{16} : offset_{16/32}$**.

La folosirea ***regiştrilor extinşi***, adresa sursă specifică **un pointer pe 48 biţi**, conţinând **o *adresă offset pe 32 biţi*** (adică 4 octeţi), urmată de ***un selector pe 16 biţi*** (primul luat din memorie e offsetul şi abia apoi se ia valoarea pentru segment).

Observaţii:

- Instrucţiunile LDS și LES *nu afectează flagurile*;
- Sintaxa instrucţiunilor trebuie respectată întocmai;
- Instrucţiunea LxS *nu afectează flagurile*;
- La procesoarele de 64 biţi nu se mai folosește noţiunea de segment, deci instrucţiunea LxS nu își mai are rostul.
- Dimensiunea operanzilor trebuie să coincidă: aceştia sunt ori pe 16 biţi, ori pe 32 biţi.

2.6.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
lea DS, var          ; nu se poate folosi registru segment
lea IP, var          ; nu se poate folosi registrul [-/E/R] IP sau [-/E/R] FLAGS
lcs IP, a            ; nu există instrucţiunea LxS cu x=C, adică folosirea registrului CS; nici [-/E/R] IP nu e admis
```

Exemple de *instrucţiuni legale*:

Exemple 2-6.1

```
lea BX,VAR           ; identic cu mov BX, offset  VAR; adresa a lui VAR e depusă în BX; atribuirea- la asamblare
lea EAX,sir          ; încarcă EAX cu offsetul lui sir
lea RBX,sir2         ; încarcă RBX cu offsetul lui sir2
```

Exemplul 2-6.2

```
lea  DI, sir [BX][SI]   ; DI = offset SIR+BX+DI, pentru adresarea elementelor din sir
lea SI, sir [AX][CX]    ; legală doar de la 386↑ datorită adresării cu AX şi CX
```

Exemplul 2-6.3

```
les  DI, sir         ; încarcă ES şi DI cu conţinutul pe 32 biţi din segmentul de date, de la locaţia dată de sir
lfs   SI, sir        ; încarcă FS şi SI cu conţinutul pe 32 biţi din segmentul de date, de la locaţia dată de sir
lgs  DI, sir         ; încarcă GS şi DI cu conţinutul pe 32 biţi din segmentul de date, de la locaţia dată de sir
lss SP, sir          ; încarcă SS şi SP cu conţinutul pe 32 biţi din segmentul de date, de la locaţia dată de sir
```

Exemplul 2-6.4

```
a dw 1234h           ; în segmentul de date se definesc variabilele a, b, c de tip word
b dw 5678h
c dw 9ABCh
lds SI, a            ; încarcă SI=1234h şi DS=5678h ; instrucţiunea operează de la dreapta spre stânga
lds ESI, dword ptr a ; încarcă prima dată pe ESI, deci ESI=56781234h și apoi pe DS; astfel, DS=9ABCh
```

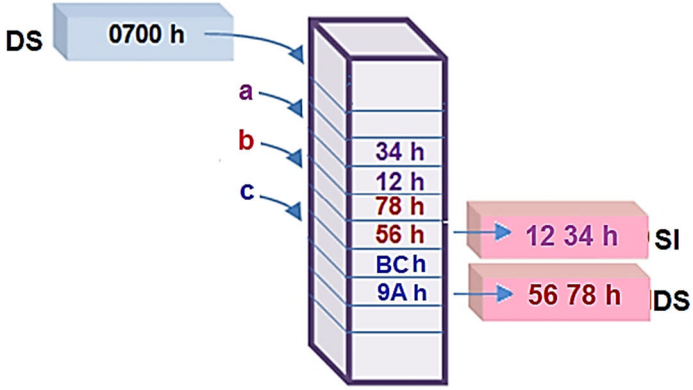


Figura 2-6.2. Ilustrarea instrucţiunii *lds SI, a* din Exemplul 2-6.4

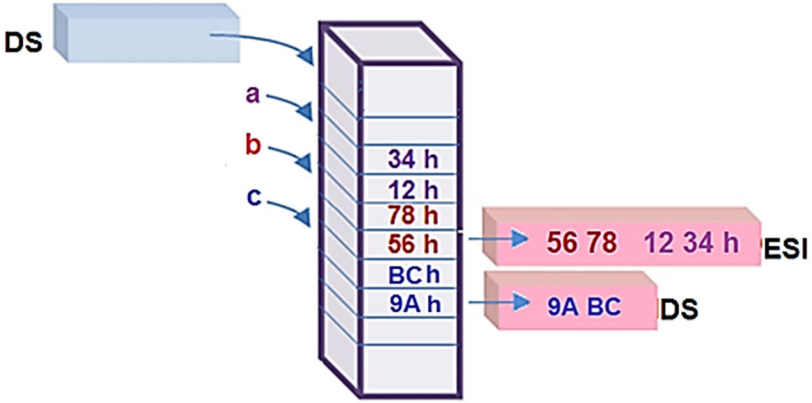


Figura 2-6.3. Ilustrarea instrucţiunii *lds ESI, a* din Exemplul 2-6.4

Exemple 2-6.5

```
lds SI, [0104h]     ; conţinutul locaţiei de memorie 0104h şi 0105h se depune în SI
                    ; iar conţinutul locaţiei de memorie 0106h şi 0107h se depune în  DS
les BX, [1234h]     ; copiază conţinutul din memorie, de la adresa dată de deplasamentul 1234h (din segmentul dat de
                    ; DS) în BL, conţinutul de la adresa 1235h în BH, iar conţinutul de la adresele 1236h și 1237h în ES.
```

Exemplul 2-6.6

```
lds EBX, [DI]       ; conţinutul din memorie, de la adresa dată de DS:[DI+3], DS:[DI+2], DS:[DI+1], DS:[DI+0] se
                    ; copiază în registrul EBX, iar cel de la adresele DS:[DI+5], DS:[DI+4] se copiază în DS
```

Exemplul 2-6.7

De exemplu, există definită o variabilă în memorie denumită *old*, de tip word, cu 2 valori, neiniţializată:

```
.data
old dw ?, ?
```

și se presupune că în această variabilă *old* se dorește să se salveze adresa far a rutinei care tratează o anumită întrerupere (se va prelua din TVI). Presupunem că am preluat această adresă far din TVI în perechea de regiștri ES:BX și apoi vrem ca aceeași adresă far să ajungă în perechea de regiștri DS:DX. Se poate folosi următoarea secvenţă:

```
mov old, BX
mov old+2, ES
lds DX, dword ptr old     ; atunci DS va conține valoarea care înainte a fost în ES,
                          ; iar DX va conține valoarea care înainte a fost în BX.
```

Ne-am putea gândi că totul s-ar fi putut desfășura mult mai simplu, prin secvența:

```
mov AX, ES
mov DS, AX      ; ES -> DS
mov DX, BX      ; BX -> DX
```

Explicaţia alegerii versiunii cu ***lds*** în locul versiunii cu cele 3 instrucţiuni mov este următorul: era necesar ca valorile să rămână undeva în memorie, într-o variabilă, pentru a nu se pierde (din regişti) pe parcursul desfăşurării programului.

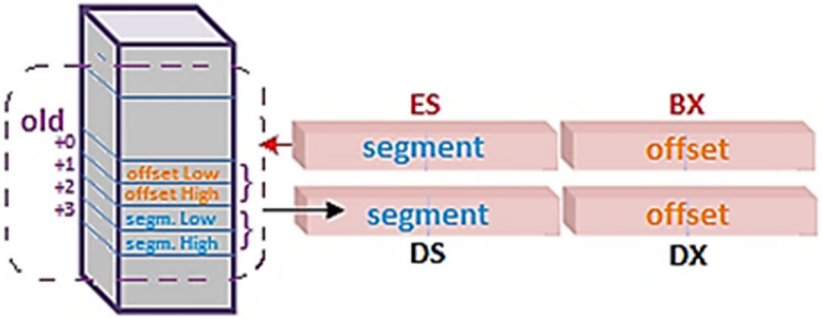


Figura 2-6.4. Ilustrarea instrucţiunii *lds DX, dword ptr old* din Exemplul 2-6.7

De fapt, contextul complet în care se foloseşte acest mecanism este următorul:
Încă de la **8086** a fost definită noțiunea de întrerupere; întreruperea este un semnal transmis în cadrul sistemului de calcul prin care se anunţă procesorul că a apărut un eveniment ce necesită intervenția acestuia; acest semnal ajunge în final la CPU care decide ce este de făcut pe baza unei *rutine de tratare* a acelui eveniment. Astfel de evenimente întrerupătoare pot fi externe sau interne SC, sau pot fi de tip hardware sau software, mascabile sau nemascabile (adică ar putea fi ignorate pe o perioadă de timp sau nu). Fiecare eveniment apărut sub formă de întrerupere are asociată o ***Rutină de Tratare a Întreruperii*** (***RTI***) în engleză termenul fiind ISR (Interrupt Service Routine). În total, în 8086 există 256 posibile surse de întrerupere (0÷FFh), fiecare având asociată câte o RTI specifică. Aceste RTI sunt organizate într-o tabelă numită TVI (Tabela Vectorilor de Întrerupere, în engleză IVT - Interrupt Vector Table).
În zona de memorie TVI (memoria este văzută ca o tabelă), pentru fiecare RTI se păstrează câte 4 octeți care semnifică adresa FAR (sub forma CS:IP) a zonei din memorie unde se găseşte efectiv, instrucțiune cu instrucțiune, rutina respectivă. Astfel, atunci când CPU primeşte un semnal de întrerupere, el trebuie să identifice exact sursa acelei întreruperi şi să cunoască vectorul ei (sau în ce loc se află RTI a ei în tabela TVI). Cunoscând sursa sau vectorul întreruperii, din TVI se poate prelua adresa RTI pe cei 4B, iar apoi cunoscând adresa FAR sub forma CS:IP, CPU poate merge în memorie şi să înceapă execuția codului de acolo, instrucțiune cu instrucțiune (va executa rutina RTI asociată acelui eveniment care a produs întreruperea).

Tabela Vectorilor de Intrerupere (TVI)

La procesoarele **8086** a fost des utilizată o tehnică numită „***redirectare de întrerupere***"; aceasta se practică atunci când se doreşte execuţia unui alt cod (pentru a obţine o facilitate suplimentară sau una total diferită) la apariţia unei întreruperi în sistem.

De exemplu, întreruperea 08h generată de hardware (atunci când apare semnal pe linia IRQ 0 a controllerului de întrerupere) este executată la fiecare bătaie a ceasului de timp real a SC. Această bătaie a ceasului are loc la fiecare 55ms, sau aproximativ 18,2 Hz (bătăi sau impulsuri pe secundă).

În urma apariţiei întreruperii cu tipul 08h, BIOS-ul realizează următoarele:

- actualizează valorile ceasului sistemului,
- opreşte unitatea de dischetă după 2 secunde de inactivitate de citire/scriere şi
- comandă întreruperea int 1Ch (numită şi întrerupere utilizator). Aceasta din urmă conduce înspre o zonă de memorie unde se găseşte RTI 1Ch (care se află în TVI la adresa 0:0070h, adică 1Ch*4) şi care de fapt nu face nimic.

Pentru a putea utiliza întreruperea 1Ch în scop propriu în cadrul unui program (deci să aibă loc o anumită procesare la fiecare 55 msec sau la un anumit multiplu de această bază de timp) se poate realiza aşa numita ***redirectare de întrerupere***, adică înlocuirea în TVI a adresei care este stocată în memorie la 0:0070h cu adresa noii rutine scrise de utilizator. Locul unde va depune procesorul această *nouă rutină* (scrisă de utilizator) în memorie se poate afla simplu, folosind operatorii ***seg*** şi ***offset***.

Redirectarea de întrerupere constă din 3 paşi, denumiți în continuare sugestiv: *pas I, pas II* şi *pas III*.

În *pasul I* se realizează o operație de ***GET (preia din IVT)***, în timp ce în *pașii II și III* se realizează o operaţie de ***SET (depune în IVT)***. Practic, în *pasul I* se preia informaţia originală din IVT și se stochează în perechea de regiștri ES:BX, în timp ce în pașii II și III se depune informaţia dorită (adresa noii rutine în pasul II, respectiv adresa vechii rutine în pasul III) în IVT din cadrul perechii de regiștri DS:DX. *Pasul III* se realizează în vederea refacerii zonei TVI aşa cum era aceasta iniţial (cu adresa rutinei originale), pentru a nu rămâne alterată la ieşirea din program. Procedeul cel mai des întâlnit în practică este următorul:

```
.data
old dw ?,?          ; se defineşte în memorie variabila old cu 2 cuvinte (4 octeți)
.code
```

; secvența de redirectare a întreruperii utilizator 1Ch înspre o nouă rutină numită ***UserRoutine***, cu folosirea acesteia în program și apoi în final, cu refacerea TVI așa cum era inițial – din variabila ***old*** se reface adresa far în TVI

;Pas I (GET–se preia valoarea CS:IP pt **RTI (1Ch originală)** -> variabila locală ***old***):

```
mov AX, 351Ch
int 21h                     ; TVI -> ES:BX
mov old, BX
mov old+2, ES               ; ES:BX -> old
```

;Pas II (SET- se depune adresa ***UserRoutine*** -> în TVI **RTI (în locul 1Ch)**):

```
mov DS, seg UserRoutine
mov DX, offset UserRoutine  ; NOUA RUTINA UserRoutine -> DS:DX
mov AX, 251Ch
int 21h                     ; DS:DX -> TVI
. . .
```

; aici se află **programul principal** unde este posibil ca valoarea registrului BX să se piardă, de aceea nu se putea folosi secvența simplificată propusă inițial

```
. . .
```

;Pas III (SET– se restaurează din variabila locală ***old*** -> în TVI **RTI (1Ch originală)**):

```
lds DX, dword ptr old       ; old -> DS:DX
mov AX, 251Ch
int 21h                     ; DS:DX -> TVI
```

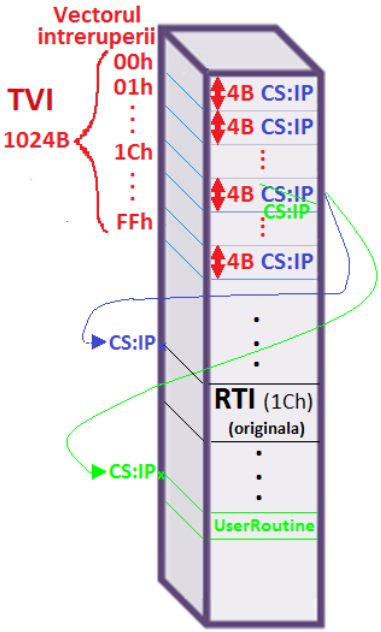


Figura 2-6.5. Ilustrarea TVI şi a procedeului de redirectare a întreruperii

2.7. Instrucţiuni de transfer pentru flaguri (PSW)

Instrucţiunile de transfer pentru flaguri pot afecta tot registrul [-/E/R] FLAGS sau doar o parte a acestuia.

Instrucţiunile **LAHF, SAHF, PUSHF** și **POPF** au fost suportate încă de la **8086↑**.

2.7.1. Instrucţiunile LAHF şi SAHF

Instrucţiunile **LAHF** ***(Load AH with Flags)*** şi **SAHF** ***(Store AH into Flags)*** au fost proiectate cu scopul de a asigura translatarea programelor de pe un procesor de 8 biţi (8085) pe unul de 16 biţi. Aceste instrucţiuni nu au operanzi: LAHF încarcă registrul AH (load AH) cu octetul cel mai puţin semnificativ al registrului [-/E/R] FLAG, respectiv SAHF depune conţinutul registrului AH (store AH) în octetul cel mai puţin semnificativ al [-/E/R] FLAG.

LAHF ; (AH) ← ([-/E/R] FLAG)$_L$ ***SAHF ; ([-/E/R] FLAG)$_L$ ← (AH)***

Procesoarele din primele generaţii x86-64 nu au suportat instrucţiunea LAHF și SAHF în mod de lucru x86-64, dar puteau executa aceste instrucţiuni în mod pe 32 biţi. Aceste instrucţiuni au fost reinstalate ulterior și sunt disponibile în majoritatea procesoarelor de după 2006. Cu o secvenţă de instrucţiuni se poate confirma dacă procesorul suportă instrucţiunile LAHF și SAHF în mod x86-64.

Atât **LAHF** cât și **SAHF** sunt suportate în modul pe 64 biţi doar dacă

bitul b0 (numit *lahf_lm*, denumire care provine de la LAHF/SAHF in long mode) din registrul ECX este setat după execuţia instrucţiunii ***cpuid*** cu EAX=80000001h la intrare, deci se verifică dacă: EAX=80000001h --->***cpuid***---> ECX$_{b0=1}$.

Figura 2-7.1 Ilustrarea modului de operare al instrucţiunii ***LAHF***

Figura 2-7.2 Ilustrarea modului de operare al instrucţiunii ***SAHF***

Observaţii:

- Instrucţiunile LAHF și SAHF nu au operanzi și *nu afectează flagurile*;
- În mod pe 64 biţi, acestea nu mai funcţionează, sunt invalide (depășite).

În mod pe 32 biţi: EFLAGS <- (EFLAGS or (AH and 0D5h)) – mascarea biţilor 7,6,4,2,0 (rezervaţi) din AH => 0D5h = 11_1 . _1_1b

2.7.2. Instrucţiunile PUSHF[-/D/Q] şi POPF[-/D/Q]

Instrucţiunile **PUSHF** ***(Push 16 Low-order Flags)*** şi **POPF** ***(Pop 16 Low-order Flags)***, suportate chiar de la 8086↑, deşi nu au operanzi, ele salvează conţinutul registrului PSW (sau FLAGS) pe stivă (instrucţiunea PUSHF), respectiv îl refac de pe stivă (instrucţiunea POPF).

Începând cu 80386↑, au fost introduse corespondentele lor **PUSHFD** (***Push All 32 Flags***) şi **POPFD** (***Pop All 32 Flags***) - folosesc registrul **EFLAGS** pe 32 biţi, iar de la modul pe 64 biţi au apărut şi **PUSHFQ** şi **POPFQ** care lucrează cu registrul **RFLAGS** de 64 biţi.

PUSHF [-/D/Q] ***;depune pe stivă conţinutul lui [-/E/R] FLAGS***

POPF [-/D/Q] ***;preia de pe stivă conţinutul de acolo şi îl depune în [-/E/R] FLAGS***

Astfel, la PUSHF se ocupă 2 octeţi pe stivă, la PUSHFD se ocupă 4 octeţi, iar la PUSHFQ se ocupă 8 octeţi.

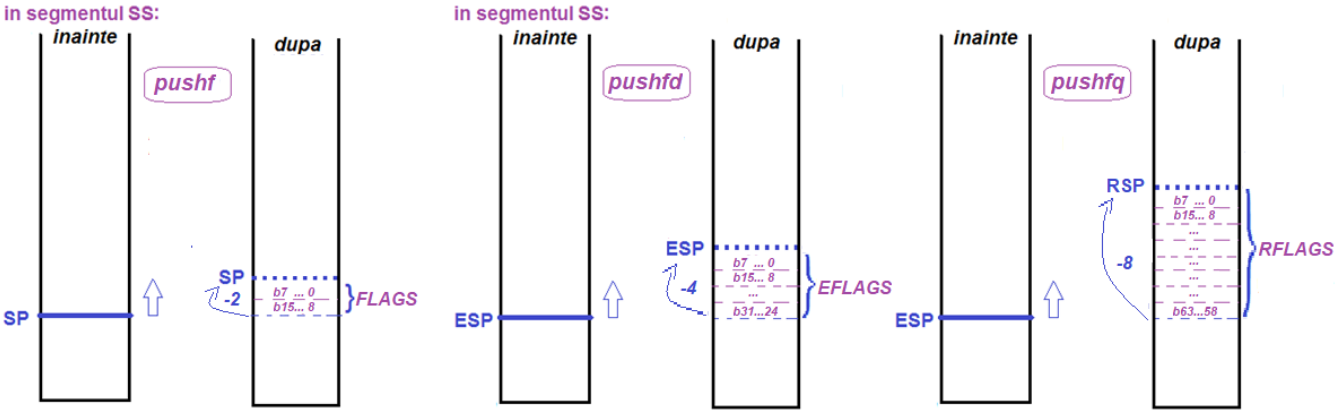


Figura 2-7.2. Ilustrarea modului de operare al instrucţiunii ***PUSHF*** pe 16 biţi (stânga), al instrucţiunii ***PUSHFD*** pe 32 biţi (mijloc) şi al instrucţiunii ***PUSHFQ*** pe 64 biţi (dreapta)

Observaţii:

- Instrucţiunile PUSHF[-/D/Q] și POPF[-/D/Q] *nu afectează flagurile (nu le schimbă valorile)*.
- Aceste instrucțiuni nu au operanzi, lucrează implicit cu [-/E/R] FLAGS.
- Perechile de instrucţiuni PUSHF şi PUSHFD, respectiv POPF şi POPFD au acelaşi cod al operaţiei, diferenţa realizându-se în funcţie de dimensiunea operanzilor utilizaţi (şi modul de lucru al procesorului).

  Unele asambloare pot forţa dimensiunea operandului la 16 biţi pentru PUSHF, resp. POPF şi la 32 biţi pentru PUSHFD, resp. POPFD. Altele pot trata mnemonicile ca fiind identice (sau sinonime) şi pot folosi setarea atributului de dimeniune la 16, resp. 32 biţi, în funcţie de modul de operare al procesorului.

Dacă se execută ***PUSHF în mod pe 32 biți***, se vor depune în stivă doar c.m.p.s. 16 biți ai registrului EFLAGS, aceasta putând duce la decalarea stivei.

Similar, ***POPF în mod pe 32 biți***, va reface doar partea c.m.p.s. a registrului EFLAGS

deci ESP se va decrementa (la PUSHF) / incrementa (la POPF) **cu 2**, nu cu 4.

La execuția PUSHFD şi POPFD, *ESP* se va decrementa/ incrementa *cu 4*, iar

la execuția PUSHFQ şi POPFQ (***în mod pe 64 biți***) *RSP* se va decrementa/ incrementa *cu 8*.

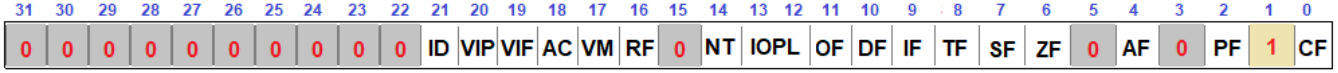


Figura 2-7.3. Ilustrarea conţinutului registrului [-/E/R] FLAGS

La funcţionarea procesorului în mod protejat, de compatibilitate sau în mod pe 64 biţi, ***la un nivel de privilegiu 0*** (echivalent cu modul real de lucru al procesorului), tuturor flagurilor din EFLAGS care nu sunt rezervate (resetate sau setate după cum se poate observa din figură), cu excepţia RF[5], VM, VIF şi VIP, li se permite modificarea stării. Flagurile RF, VM, VIF şi VIP rămân neschimbate. Dacă se operează ***la un nivel de privilegiu mai mare ca 0, dar mai mic decât IOPL***, toate flagurile permit modificarea stării, cu excepţia celor IOPL, RF, VM, VIF, VIP şi ***IF***; acestea rămân neschimbate. Flagul IF poate fi alterat doar la un nivel de privilegiu > IOPL. În plus, flagurile AC şi ID pot fi modificate doar dacă dimensiunea operandului este 32 sau 64 biţi.

[5] Flagul RF este forţat în 0 la execuţia unei instrucţiuni POPF[-/D/Q].

2.7.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
lahf BX          ; sintaxă eronată, LAHF nu are operanzi
sahf BX          ; sintaxă eronată, SAHF nu are operanzi
pushf AX         ; sintaxă eronată, PUSHF[-/D/Q] nu are operanzi
popfd AX         ; sintaxă eronată, POPF[-/D/Q] nu are operanzi
```

Exemple de *instrucţiuni legale*:

Exemplul 2-7.1

```
lahf             ; AH=Low (FLAGS); încarcă în AH valorile curente ale flagurilor: AH=SZ0A0P1C
...              ; se pp. că se execută instrucţiuni care modifică starea vreunuia sau mai multora dintre flaguri
sahf             ; deşi flagurile au fost modificate în concordanță cu operaţiile de mai sus, la folosirea
                 ; instrucţiunii sahf în FLAGS vor fi din nou valorile dinainte de acele operaţii AH -> Low (FLAGS)
```

Exemplul 2-7.2

```
pushf            ; încarcă stiva cu valorile indicatorilor FLAGS
pop AX           ; sunt luate de pe stivă şi depuse în AX
or AX, 01h       ; se setează Carry, prin operația logică bit cu bit OR cu 00000001b
push AX          ; se depune în stivă cu Carry modificat
popf             ; ia din stivă şi depune înapoi în registrul FLAGS, cu CF=1
```

Exemplul 2-7.3

```
pushfd           ; încarcă stiva cu valorile EFLAGS, ocupând 4 octeţi „în sus”
pushfq           ; încarcă stiva cu valorile RFLAGS, ocupând 8 octeţi „în sus”
```

Capitolul 3. Instrucţiuni aritmetice

Instrucţiunile aritmetice sunt cele care se folosesc la realizarea diferitelor operaţii precum cele de adunare, scădere, înmulţire, împărţire dar și cele care ajută la obţinerea acestora (de extindere sau corecţie a Acumulatorului). Tot aici sunt incluse și instrucţiunile de comparare, deoarece operaţia de comparare este văzută ca o scădere fictivă. Aceste instrucţiuni în general ***modifică flagurile OF, SF, ZF, AF, PF, CF***, indicatori care astfel au primit denumirea de "***flaguri aritmetice***".

1. **pentru adunare:** **ADD, ADC, INC**
2. **pentru scădere și negare:** **SUB, SBB, DEC, NEG**
3. **pentru înmulţire și împărţire:** **MUL, IMUL, DIV, IDIV**
4. **pentru comparare:** **CMP, CMPXCHG, CMPXCHG[8/16]B**
5. **pentru extinderea ACC:** **CWD, CDQ, CQO, CBW, CWDE, CDQE**
6. **pentru corecţia ACC:** **DAA, DAS, AAA, AAS, AAM, AAD**

Instrucţiunile aritmetice au în general 2 operanzi, iar rezultatul operaţiei e depus în primul dintre ei. Ca operanzi, nu se admit regiştri segment, nici [-/E/R] IP sau [-/E/R] FLAGS.

3.1. Instrucţiuni pentru adunare

Încă de la procesoarele **8086↑**, au fost suportate instrucţiunile ADD, ADC și INC pentru realizarea operaţiilor de adunare între diferiţi operanzi.

Instrucţiunea XADD, înrudită cu instrucţiunea ADD şi suportată de la **80486↑**, a fost deja prezentată în secţiunea 2.2.2 la instrucţiunile de transfer și nu va mai fi abordată aici.

3.1.1. Instrucţiunea ADD

Instrucţiunea **ADD** ***(Integer Addition)*** adună conţinutul sursei la cel al destinaţiei şi depune rezultatul în destinaţie, vechea valoare pierzându-se.

De la procesoarele **80386↑**, operanzii pot fi şi de dimensiunea dublucuvântului, iar de la procesoarele care suportă modul pe 64 biţi, se pot folosi chiar şi cvadruplucuvinte.

ADD destinaţie, sursă ; (destinaţie) = (destinaţie) + (sursă)
ADD $\{reg_{8,16,32,64} | mem_{8,16,32,64}\}, \{reg_{8,16,32,64} | mem_{8,16,32,64} | imed_{8,16,32}\}$

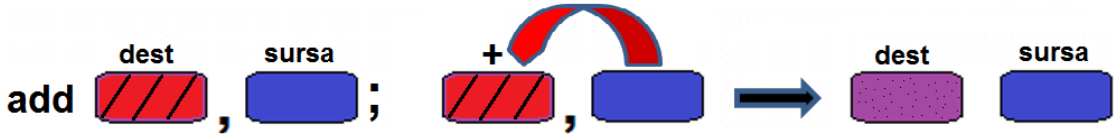


Figura 3-1.1. Ilustrarea modului de operare al instrucţiunii ***ADD***

Observaţii:

- Instrucţiunea ADD *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, CF conform rezultatului operaţiei;
- 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.
- În general, depășirile sunt arătate de OF în cazul numerelor cu semn, respectiv de CF în cazul numerelor fără semn.

3.1.2. Instrucţiunea ADC

Instrucţiunea **ADC** ***(Add with Carry)*** adună cu transport: este identică cu ADD, doar că se adună şi bitul de transport CF (cel obţinut în cadrul operaţiei anterioare) la rezultatul final. Altfel spus, se adună operandul destinaţie cu cel sursă și cu CF, iar rezultatul se depune în operandul destinaţie. Instrucţiunea ADC este utilă la execuţia de adunări pe lungimi mai mari decât cuvântul prelucrat.

ADC destinaţie, sursă; ***;(destinaţie) = (destinaţie) + (sursă) +CF***

ADC {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$},{reg$_{8,16,32,64}$| mem$_{8,16,32,64}$| imed$_{8,16,32}$}

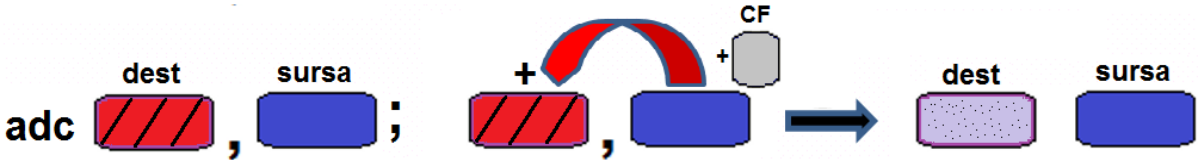


Figura 3-1.2. Ilustrarea modului de operare al instrucţiunii ***ADC***

Observaţii:

- Instrucţiunea ADC *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, CF în conformitate cu rezultatul operaţiei;
- Toate observaţiile sunt similare celor de la instrucţiunea ADD;
- Instrucţiunea ADC este utilă pentru a aduna numere ce ar putea furniza un rezultat mai mare decât se alocă iniţial pentru operanzi (de exemplu, la adunarea de mai mulţi octeţi, rezultatul ar putea avea nevoie de 9 biţi la stocare, nu doar de 8); astfel, se adaugă transportul dintr-o operaţie anterioară. În general se realizează adunarea cu ADD și apoi urmează instrucţiunea ADC pentru a ţine cont și de eventualul transport apărut anterior.

3.1.3. Instrucţiunea INC

Instrucţiunea **INC** ***(Increment)*** incrementează cu 1 destinaţia şi modifică toate flag-urile aritmetice, mai puţin CF. Nu se pot folosi nici regiştri segment, nici [-/E/R] IP sau [-/E/R] FLAGS ca destinaţie.

INC destinaţie ; (destinaţie) = (destinaţie) + 1
INC {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}

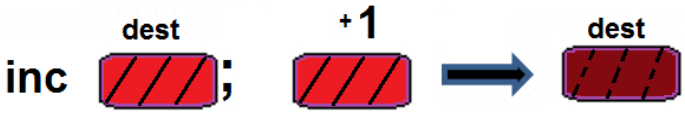


Figura 3-1.3. Ilustrarea modului de operare al instrucţiunii ***INC***

Observaţii:

- Instrucţiunea INC *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, în conformitate cu rezultatul operaţiei, *dar* ***nu modifică valoarea flagului CF***; în mod special a fost astfel concepută încât valoarea lui CF să se păstreze;
  - totuși, dacă se dorește modificarea lui CF la o operaţie de incrementare cu 1, atunci se poate folosi instrucţiunea ADD (aceasta modifică flagul CF);
- Operanzii pot fi atât numere fără semn cât și numere cu semn.

3.1.4. Exemple

Exemple de *instrucţiuni ilegale:*

```
add AX, BL          ; operanzi de dimensiuni diferite
add var1, [BX]      ; ambii operanzi sunt din memorie
add DS, 2           ; nu sunt admiși regiștri segment; nici [-/E/R]IP, nici [-/E/R]FLAGS

adc AX, BL          ; operanzi de dimensiuni diferite
adc var1, [BX]      ; ambii operanzi sunt din memorie
adc ES, 2           ; nu sunt admiși regiștri segment; nici [-/E/R] IP, nici [-/E/R]FLAGS

inc AX, BX          ; sintaxa eronată
inc DS              ; nu se pot folosi regiştri segment
inc FLAGS           ; nu se poate accesa astfel [-/E/R]FLAGS
```

Exemple de *instrucţiuni legale*:

Exemple 3-1.1

```
add AL, BH          ; AL=AL+BH adunarea se reflectă doar în AL, nu trece şi în AH la o eventuală depăşire
add AX, CX          ; AX=AX+CX – se obţine rezultatul în AX
add EBX, EDX        ; EBX=EBX+EDX (pe 32 biţi) (de la 80386↑)
add RAX, RBX        ; RAX=RAX+RBX (pe 64 biţi) (de la Pentium 4↑)
```

Exemple 3-1.2

```
add BX, [sir+DI]        ; BX=BX+cuvântul din memorie dat de DS cu offset sir +DI
add BX, [EAX+2*ECX]     ; cuvântul adresat de ECX de 2 ori + EAX se adună la cuv. din BX şi rezultatul se depune în BX
add EBX, [RAX+RCX]      ; dublucuvântul adresat de RAX+RCX se adună la cel din EBX şi rezultatul se depune în EBX
                                                                                    ; (mod 64biţi)
```

Exemplul 3-1.3

```
add byte ptr [SI],2     ; destinaţia este în memorie, sursa imediată: la octetul adresat de DS, cu offsetul dat de
                        ; conţinutul registrului SI, se adună 2
```

Exemplul 3-1.4

```
mov AX, 9FFEh          ; AX=9FFEh = 40958
mov BX, 0C001h         ; BX=0C001h=49153
add AX, BX             ; AX va fi 5FFFh şi CF=1, care trebuie interpretat ca 10000h+5FFFh= 65536+24575=90111
```

Exemplul 3-1.5

Dacă după instrucţiunea ADD se foloseşte adunarea cu carry a unui 0, adică ADC, atunci vom avea:

```
mov AX, 9FFFh          ; AX=9FFFh
mov BX, 0C001h         ; BX=0C001h
add AX, BX             ; AX va fi 6000h şi CF=1
adc AX, 0              ; AX va fi 6001h şi CF=0
```

operaţia de adunare a lui AX cu BX (echivalent cu primele 3 instrucţiuni) se putea realiza si la nivel de 8 biţi, astfel:

```
mov AX, 9FFFh          ; AX=9FFFh
mov BX, 0C001h         ; BX=0C001h
add AL, BL             ; AL=00h şi CF=1
adc AH, BH             ; AH=60h, deci AX=6000h şi CF=1
```

Exemplul 3-1.6 Pentru un şir de elemente dat, trebuie calculată suma elementelor de pe o anumită poziţie. De exemplu, ne interesează suma elementelor de pe poziţii de ordin par dintr-un şir cu 12 elemente, definite pe octet:

```
.data
sir db 3,2,6,5,7,9,1,4,0,8,1,2          ; s-a definit șirul de date în memorie
.code
mov AL,0                ; valoarea iniţială a sumei va fi 0
mov SI, 0
add AL, sir [SI]        ; adună la 0 pe primul element, cel de ordin 0 (sau +0)
add AL, sir [SI+2]      ; adună la sumă al 3-lea element, cel de pe poziţia +2
add AL, sir [SI+4]      ; adună la sumă al 5-lea element, cel de pe poziţia +4
add AL, sir [SI+6]      ; adună la sumă al 7-lea element, cel de pe poziţia +6
```

```
add AL, sir [SI+8]          ; adună la sumă al 9-lea element, cel de pe poziţia +8
add AL, sir [SI+10]         ; adună la sumă al 11-lea element, cel de pe poziţia +10
```

Exemplul 3-1.7 Pentru un şir de elemente dat, trebuie calculată suma tuturor elementelor, pentru un procesor **386**. Adresarea cu ECX este posibilă doar de la **386↑.**

```
.data
sir dw 305,207,605,5123,4567,9876,1098,4567,0,800        ; s-a definit şirul de date în memorie
.code
mov AX,0                    ; valoarea iniţială a sumei va fi 0, poate stoca până la valoarea maximă 65.535
mov EBX, offset sir         ; adresa unde începe şirul în memorie
mov ECX, 0                  ; index la sir
add AX, [EBX+2*ECX]         ; adună la sumă pe primul element de tip cuvânt din sir
add ECX, 1
add AX, [EBX+2*ECX]         ; adună la sumă pe al doilea element de tip cuvânt din sir
add ECX, 1
add AX, [EBX+2*ECX]         ; adună la sumă pe al treilea element de tip cuvânt din sir
add ECX, 1
. . .                       ; programul repetă ultimele 2 instrucţiuni până la adunarea tuturor elementelor dorite din sir
```

Exemplul 3-1.8

```
adc BX, [BP+2]              ; adună cuvântul din stivă (adresat de registrul segment SS) aflat la offsetul dat de BP+2 la
                            ; valoarea BX+CF
```

Exemplul 3-1.9

```
adc ECX, [EBX]              ; adună dublucuvântul din segm. de date adresat de EBX la cel din reg. ECX,
                            ; adunând şi valoarea lui CF
```

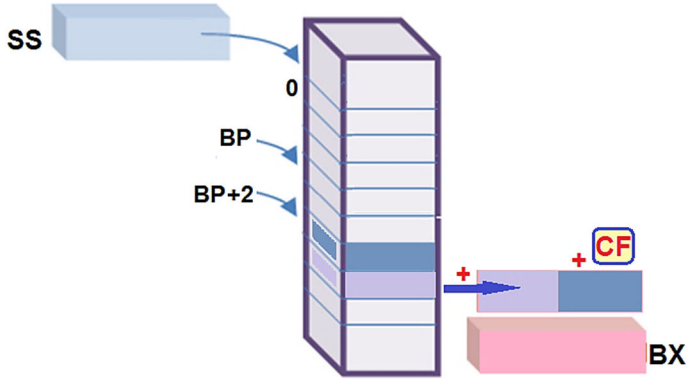


Figura 3-1.4. Ilustrarea instrucţiunii *adc BX,[BP+2]* din Exemplul 3-1.8

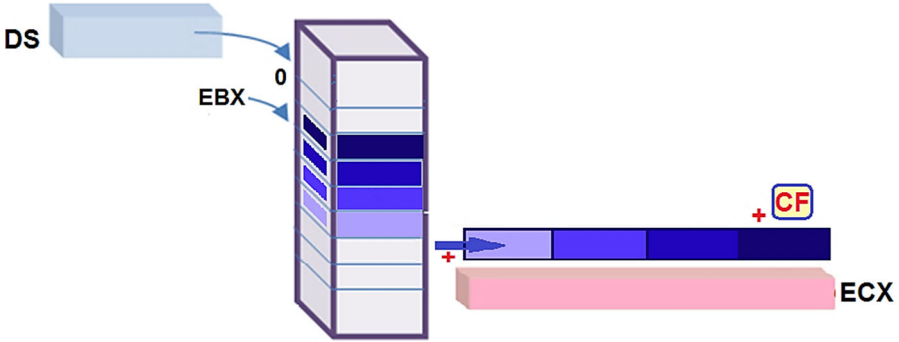


Figura 3-1.5. Ilustrarea instrucţiunii *adc ECX,[EBX]* din Exemplul 3-1.9

Exemplul 3-1.10 Pe un procesor de 16 biţi (ex. **8086-80286**) se doreşte adunarea a 2 numere pe 32 biţi. Dacă ar fi fost disponibili regiştri de 32 biţi, operaţia ar fi fost simplă: add EAX, EBX, presupunând că cele 2 numere se aflau în regiştrii EAX şi EBX. Dar dacă dimensiunea maximă a regiştrilor e de 16 biţi, atunci fiecare număr se va scrie cu ajutorul unei perechi de 2 regiştri de 16 biţi. Astfel, numerele se vor depune în 2 perechi de regiştri, de ex. DX:AX (primul număr) şi BX:CX (cel de-al 2-lea nr). În urma realizării operaţiei, suma se va depune în destinaţie, adică în perechea de regiştri DX:AX.

```
add AX, CX          ; se adună părţile mai puţin semnificative (16b) ale celor 2 operanzi;
                    ; în cazul în care suma e > 0FFFFh, se va seta CF
adc DX, BX          ; apoi se adună la suma părţilor mai semnificative şi valoarea CF
```

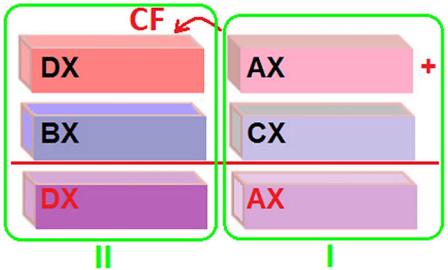


Figura 3-1.6. Ilustrarea instrucţiunilor din Exemplul 3-1.10

Exemplul 3-1.11 Exemplul anterior, transpus pe un procesor de 32 biţi (**80386-Core2**) dar pe care se doreşte însumarea a 2 valori pe 64 biţi, se scrie:

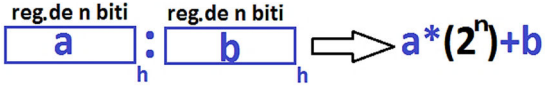
```
add EAX, ECX        ; se adună părţile (32b) mai puţin semnificative ale celor 2 operanzi;
                    ; în cazul în care suma e > 0FFFF FFFFh, se va seta CF
adc EDX, EBX        ; apoi se adună la suma părţilor mai semnificative şi valoarea CF
```

Exemplul 3-1.12 În prezent, operaţia realizată în exemplul anterior, pe un procesor de 64 biţi, se realizează simplu, cu o singură instrucţiune, prin folosirea regiştrilor de 64 biţi:

```
add RAX, RBX    ; se adună deodată toţi cei 64 biţi (disponibilă de la Pentium 4↑)
```

Exemplele 3-1.10, 3-1.11 şi 3-1.12 subliniază faptul că operaţia de adunare a 2 numere de 2n biţi folosind perechi de regiştri (fiecare registru având n biţi), se poate realiza însumând cu ***add*** părţile lor inferioare, iar apoi în al doilea pas însumând cu ***adc*** părţile lor superioare. Operaţia este simplu de realizat pe un procesor de ordin superior, folosind doar 2 regiştri de dimensiune 2n biţi, însumaţi simplu cu ***add***.

Presupunând că într-o pereche de regiştri de n biţi (fiecare registru poate stocate valori unsigned între $[0 \div 2^n-1]$) este stocată o valoare care se observă uşor scrisă în hexazecimal, dar vrem să o convertim în zecimal, se poate utiliza următoarea relaţie:



(relaţia 3-1.1)

De exemplu, în registrul AX avem valoarea 10E1h=4321 care s-ar putea considera scrisă în perechea de regiştrii AH:AL=10h:E1h; aplicând relaţia 3-1.1, vom avea: $10h*2^8+E1h$, adică $16*2^8+225=16*256+225=4321$. Această relaţie poate fi aplicată asupra oricărei perechi de regiştri, de exemplu cele necesare la operaţiile de înmulţire sau împărţire DX:AX, EDX:EAX, RDX:RAX.

Exemplul 3-1.13

```
mov AX, 1234h        ; AX=1234h
inc AX               ; AX=1235h
```

Exemplul 3-1.14

```
clc                  ; CF=0
mov AX, 0FFFFh       ; AX=FFFFh
inc AX               ; AX=0000h și CF=0 (CF nu se va seta, cum s-ar fi întâmplat la instrucţiunea add AX,1)
```

Exemplul 3-1.15

```
inc byte [7]         ; se incrementează octetul din memorie de la adresa DS:7
inc dword ptr [EBX]  ; adună 1 la dublucuvântul din segmentul de date, de la offsetul dat de conţinutul registrului EBX
```

3.2. Instrucţiuni pentru scădere şi negare

Similar operaţiilor de adunare, încă de la **8086↑** au fost implementate și operaţiile de scădere prin instrucţiunile SUB, SBB și DEC. Tot în această categorie a fost inclusă și instrucţiunea NEG care realizează negarea unei valori, adică se obţine opusul unui număr (practic, se folosește tot o operaţie de scădere).

3.2.1. Instrucţiunea SUB

Instrucţiunea **SUB** ***(Subtraction)*** scade conţinutul sursei din destinaţie, rezultatul punându-se în destinaţie. Instrucţiunea SUB este asemănătoare cu ADD deoarece poate fi interpretată ca o adunare a destinaţiei cu complementul faţă de 2 al sursei, inversând deci și rolul flagului CF (se va interpreta ca Borrow, adică împrumut).

SUB destinaţie, sursă ***; (destinaţie) = (destinaţie) – (sursă)***

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

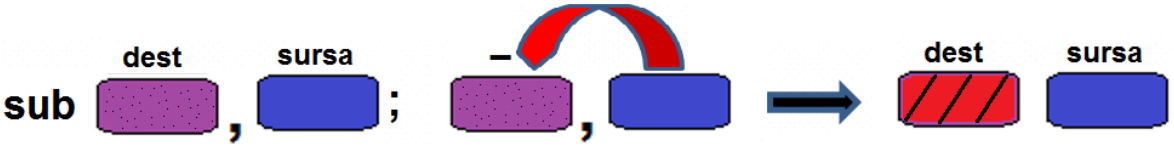


Figura 3-2.1. Ilustrarea modului de operare al instrucţiunii ***SUB***

Observaţii:

- Instrucţiunea SUB *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, CF, conform rezultatului operaţiei;
- 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.

3.2.2. Instrucţiunea SBB

Instrucţiunea **SBB** ***(Subtract with Borrow)*** scade cu împrumut: este identică cu SUB, doar că se ţine cont de un împrumut anterior (intervine în scădere şi CF) şi se foloseşte în general la scăderi de operanzi care se scriu pe mai multe cuvinte. Altfel spus, se adună operandul sursă cu CF și această sumă se scade din operandul destinaţie. Instrucţiunea SBB este instrucţiunea pereche a lui ADC, dar pentru operaţia de scădere, CF va lua rolul lui Borrow.

SBB destinaţie, sursă ***;(destinaţie) = (destinaţie) – (sursă) – CF***

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

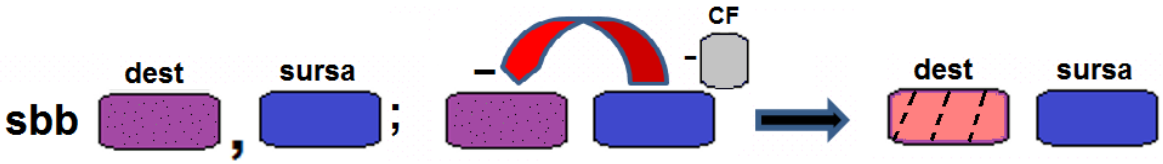


Figura 3-2.2. Ilustrarea modului de operare al instrucţiunii ***SBB***

Observaţii:

- Instrucţiunea SBB *modifică flagurile aritmetice* SF, ZF, PF, AF, CF, OF conform rezultatului operaţiei;
- Toate observaţiile sunt similare celor de la instrucţiunea SUB;
- Întrucât SBB este perechea instrucţiunii ADC, în general se realizează scăderea cu SUB și abia apoi urmează instrucţiunea SBB pentru a ţine cont și de posibilul împrumut apărut anterior.

3.2.3. Instrucţiunea DEC

Instrucţiunea **DEC** ***(Decrement)*** decrementează cu 1 destinaţia şi modifică toate flagurile aritmetice în concordanţă, cu excepţia flagului CF (Carry Flag). Acesta nu e afectat de instrucţiunea DEC.

DEC destinaţie ;(destinaţie) = (destinaţie) – 1
DEC {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}

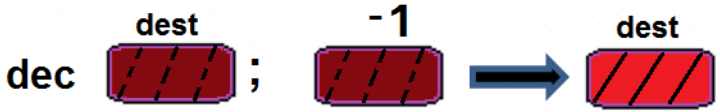


Figura 3-2.3. Ilustrarea modului de operare al instrucţiunii ***DEC***

Observaţii:

- Instrucţiunea DEC *modifică flagurile aritmetice* OF, SF, ZF, AF, PF, în conformitate cu rezultatul operaţiei, *dar nu modifică valoarea flagului CF*; în mod special a fost astfel concepută încât valoarea lui CF să se păstreze;
  - totuși, dacă se dorește modificarea lui CF la o operaţie de decrementare cu 1, atunci se poate folosi instrucţiunea SUB (aceasta modifică flagul CF);
- Operanzii pot fi consideraţi atât numere fără semn cât și numere cu semn.

3.2.4. Instrucţiunea NEG

Instrucţiunea **NEG** ***(Negate)*** schimbă semnul operandului destinaţie: această instrucţiune afectează toate flagurile aritmetice şi realizează complementul faţă de 2 al destinaţiei, fiind echivalentă cu o scădere a operandului destinaţie din valoarea 0.

NEG destinaţie ; (destinaţie) = 0 – (destinaţie)
NEG {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$}

Figura 3-2.4. Ilustrarea modului de operare al instrucţiunii ***NEG***

Observaţii:

- Instrucţiunea NEG *modifică flagurile* SF, ZF, PF, AF, OF conform rezultatului operaţiei, *dar*
  - *flagul CF suferă o excepţie*: dacă operandul este 0, atunci CF este resetat, deci pus în 0;
    altfel, CF va fi setat, deci va fi pus în 1;

(se poate interpreta ca la regulile auxiliare de obţinere a numărului negativ în C2: din 0h se scade val în binar sau hexa a numărului, unde nu se ţine cont dacă acest nr e fără semn sau cu semn, deci nici dacă ar fi pozitiv sau negativ; astfel, din 0 se scade ceva, deci rezultă CF=1 ca borrow sau împrumut)

- Operanzii pot fi consideraţi atât numere fără semn cât și numere cu semn.

3.2.5. Exemple

Exemple de *instrucţiuni ilegale:*

```
sub AX, BL          ; operanzi de dimensiuni diferite
sub var1, [SI]      ; ambii operanzi sunt din memorie
sub DS, 2           ; nu sunt admiși regiștri segment; nici [-/E/R]IP, nici [-/E/R]FLAGS

sbb AX, BL          ; operanzi de dimensiuni diferite
sbb var2, [DI]      ; ambii operanzi sunt din memorie
sbb ES, 2           ; nu sunt admiși regiștri segment; nici [-/E/R]IP, nici [-/E/R]FLAGS

dec AX, BX          ; sintaxa eronată

neg AX, BX          ; sintaxa eronată
```

Exemple de *instrucţiuni legale*:

Exemple 3-2.1

```
sub CL, AL          ; CL=CL-AL operanzi pe 8 biţi (8086↑)
sub AX, BX          ; AX=AX-BX operanzi pe 16 biţi (8086↑)
sub ECX, EBP        ; ECX=ECX-EBP operanzi pe 32 biţi (80386↑)
sub RDX, R8         ; RDX=RDX-R8 operanzi pe 64 biţi (mod pe 64 biţi) (P4↑)
```

Exemplul 3-2.2

```
sub [DI], BH        ; scade conţinutul registrului BH din octetul care se află în segmentul de date la offsetul dat de DI
```

Exemplul 3-2.3

```
mov BL, 5           ; BL= 05h
sub BL, 4           ; BL = 01h  şi  CF = 0
```

Exemplul 3-2.4

```
mov AX, 3000h        ; AX= 3000h = 12288
mov BX, 0B000h       ; BX=0B000h = 45056
sub AX, BX           ; AX= 8000h = 32768 şi CF=1 (interpretat ca Borrow), deci se interpretează în următorul mod:
                     ; 65536+12288-45056=32768
```

Exemplul 3-2.5 Dacă imediat după instrucţiunea SUB din exemplul 3-2.4 se foloseşte SBB, atunci vom avea:

```
mov AX, 3000h        ; AX= 3000h
mov BX, 0B000h       ; BX=0B000h
sub AX, BX           ; AX= 8000h şi CF=1
sbb AX, 1            ; scade din AX=8000h nu 1, ci 1+CF=>AX=7FFEh, iar CF va deveni CF=0
```

Exemplul 3-2.6 Similar exemplului 3-1.5, se poate realiza scăderea elementelor: se va înlocui add cu sub, iar adc cu sbb; în plus, rolul lui Carry Flag trebuie interpretat ca un *Borrow (împrumut)*.

```
sub AX, CX           ; se scad părţile mai puţin semnificative și în cazul în care scăzătorul e > decât descăzutul,
                     ; se va seta CF (un împrumut, adică borrow)
sbb DX, BX           ; la scăderea părţilor mai semnificative, se va ţine cont și de acest împrumut
                     ; prin folosirea instrucţiunii SBB
```

Exemplul 3-2.7

```
mov AX, 1234h        ; AX=1234h
dec AX               ; AX=1233h
```

Exemplul 3-2.8

```
clc                  ; CF=0
mov AX, 0            ; AX=0h
dec AX               ; AX=FFFFh= -1 și CF=0, deși s-a realizat un împrumut;
                     ; dacă s-ar fi folosit instrucţiunea sub AX,1 atunci CF ar fi fost 1
```

Exemplul 3-2.9

```
dec DWORD PTR [var]  ; decrementează dublucuvântul stocat în memorie la adresa dată de var
```

Exemplul 3-2.10

```
mov AX, 0F0Fh          ; AX=0F0Fh (complementul faţă de 2 al lui 0F0Fh este F0F1h)
neg AX                 ; AX=F0F1h şi CF=1 (interpretat ca împrumut - borrow)
```

Exemplul 3-2.11

```
mov AX, 37             ; AX=  37 = 0025h
neg AX                 ; AX= -37 = FFDBh, CF=1
```

Exemplul 3-2.12

```
mov AX, -37            ; AX= -37 = FFDBh
neg AX                 ; AX = 37 = 0025h, CF=1
```

Exemplul 3-2.13

```
mov AX, 0              ; AX = 0 = 0000h
neg AX                 ; AX= -0 = 0000h, CF=0
```

Exemplul 3-2.14

Scăderea a 2 numere

```
mov BL, 4
mov AL, 8
sub BL, AL       ; BL = 4-8, CF=1
```

Operaţia 4 -8 realizată pe biţi:

```
0000 0100b-
0000 1000b
1111 1100b= FCh=-4, CF=1
```

Adunarea cu C2 al scăzătorului în loc de scădere

```
mov BL, 4
mov AL, -8
add BL, AL       ; BL = -4, CF=1
```

Operaţia 4 + (-8) realizată pe biţi:

```
0000 0100b+
1111 1000b
1111 1100b= FCh=-4, /CF=0 (aici CF e cu rol inversat)
```

Exemplul 3-2.15

```
.data
a db 1,-1,2,3
.code
a) neg a            ; prima locaţie e afectată: 01 h devine FFh
b) neg word ptr a ; primele 2 locaţii (adică FF01h=-255) sunt afectate;
în loc de 01h|FFh, în memorie va fi FFh|00h adică 255
```

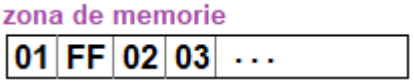


Figura 2-4.13. Ilustrarea zonei de memorie pentru Exemplul 3-2.15 (valorile sunt în hexazecimal)

3.3. Instrucţiuni pentru înmulţire şi împărţire

Încă de la 8086↑, instrucţiunile pentru realizarea operaţiilor de înmulţire (MUL și IMUL) și împărţire (DIV și IDIV) au fost suportate pe procesoarele din familia x86, deși ulterior unele dintre aceste instrucţiuni au primit forme noi.

La înmulţire, operanzii trebuie să aibă aceeaşi dimensiune: ***octet*** sau ***cuvânt*** (de la 8086↑), ***dublucuvânt*** (de la 80386↑), respectiv ***cvadruplucuvânt*** (de la Pentium 4↑).

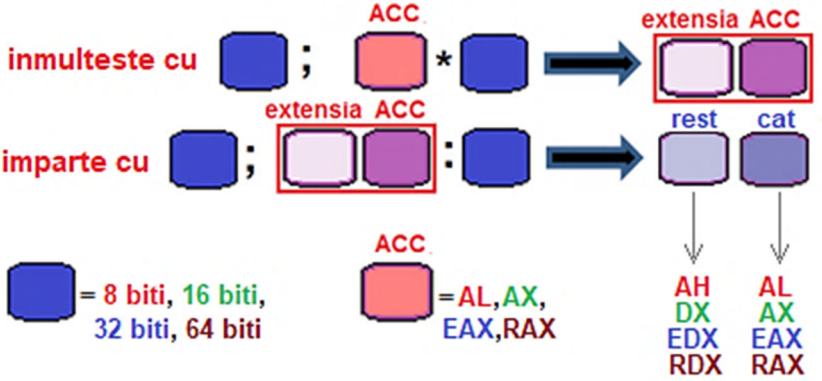


Figura 3-3.1. Ilustrarea modului de operare al instrucţiunilor de înmulţire şi împărţire

Instrucţiunile de înmulţire realizează o înmulţire între doi operanzi: ***operandul sursă (specificat în instrucţiune)*** și ***Accumulator***, adică registrul **AL**, **AX**, **EAX** sau **RAX**.

Rezultatul obţinut este apoi salvat într-o structură de regiștri de dimensiune dublă, precum perechea de regiştri **AH:AL** (adică registrul AX), perechea de regiștri **DX:AX**, perechea de regiștri **EDX:EAX**, respectiv perechea de regiștri **RDX:RAX**.

Instrucţiunile de împărţire realiz. operaţia inversă înmulţirii: deîmpărţitul este ***ACC extins***, exprimat pe **16 biţi**, **32 biţi**, **64 biţi** sau **128 biţi**, împărţitorul va fi ***operandul sursă (specificat în instrucţiune)*** de dimensiune jumătate cât deîmpărţitul, iar ***câtul*** şi ***restul*** obţinute vor avea dimensiunea tot jumătate din cea a deîmpărţitului, fiind exprimate în ***extensia ACC*** sau ***ACC***.

3.3.1. Instrucţiunile MUL şi IMUL

Înmulţirea duce la un rezultat de dimensiune dublă:

în general, înmulţirea a 2 numere scrise fiecare pe ***n biţi*** poate furniza un rezultat pe ***2*n biţi***.

De exemplu, dacă se înmulţesc 2 operanzi pe 8 biţi, pentru a nu apărea depăşiri, rezultatul se va scrie corect pe 16 biţi.
În modul pe 64 biţi (**Pentium 4 ↑**), două numere pe 64 biţi se pot înmulţi şi rezultatul furnizat se va scrie pe 128 biţi.

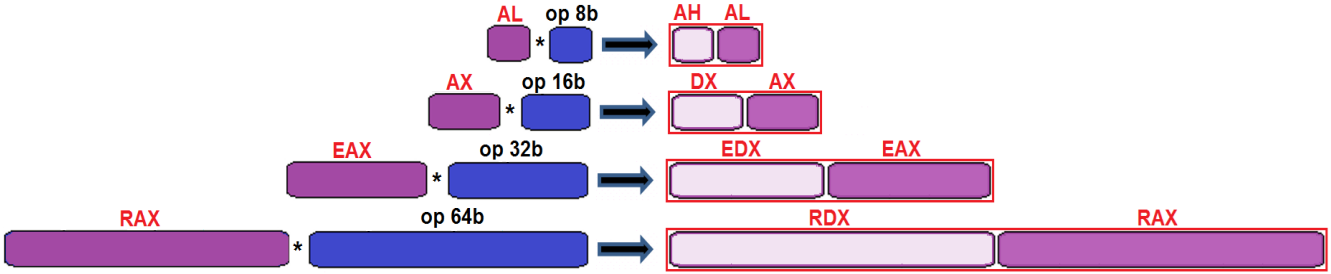


Figura 3-3.2. Ilustrarea modului de operare al instrucţiunii ***MUL*** la nivel de octet, cuvânt, dublucuvânt, cvadruplucuvânt

MUL ***(Multiply***) se fol. pt a realiza o operaţie de înmulţire ***fără semn***, iar **IMUL** ***(Integer Multiply)*** pt înmulţire ***cu semn***.

Operaţia specifică înmulţirii **MUL** (***Unsigned multiply***) se realizează între acumulator şi un alt operand, rezultatul obţinut fiind pe 16 biţi sau pe 32 biţi (de la **8086↑**), pe 64 biţi (de la **80386↑**) sau chiar pe 128 biţi (de la **Pentium 4 ↑**).

Destinaţia (implicită) e ***Acumulatorul*** sau o construcţie extinsă a acestuia (AX, DX:AX, EDX:EAX sau RDX:RAX), iar ***sursa e un registru sau o locaţie de memorie*** pe 8 sau 16 biţi (de la **8086↑**), pe 32 (de la **80386↑**) sau pe 64 biţi (de la **Pentium 4↑**).

Realizarea operaţiei **MUL** de la **8086↑**: Perechea AH:AL = AL * (reg_8/mem_8)
Perechea DX:AX = AX * (reg_{16}/mem_{16})

În plus, de la **386↑** se mai poate realiza şi: Perechea EDX:EAX=EAX*(reg_{32}/mem_{32})

Mai mult, de la **Pentium 4↑** se mai poate realiza şi: Perechea RDX:RAX=RAX*(reg_{64}/mem_{64})

Instrucţiunea **MUL** ***(Unsigned Multiply***) realizează o înmulţire între cei doi operanzi consideraţi ***numere fără semn***.

MUL sursă ; Acc extins = Acc * sursă
MUL { reg$_{8/16/32/64}$ | mem$_{8/16/32/64}$ }

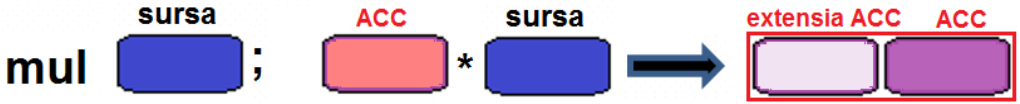


Figura 3-3.3. Ilustrarea modului de operare al instrucţiunii ***MUL***

Observaţii:

- Sunt relevante flagurile OF şi CF, altele (de exemplu ZF) putând furniza valori eronate.
- Instrucţiunea MUL *nu afectează flagurile aritmetice* SF, ZF, PF, AF, dar *resetează flagurile* CF și OF (deci CF=OF=0) *dacă jumătatea superioară a rezultatului este 0*; altfel, aceste două flaguri sunt setate;
- Trebuie subliniat că:
  - instrucţiunea MUL consideră numerele implicate în operaţie ca fiind *fără semn* (indiferent cum au fost ele definite în segmentul de date)
  - dimensiunea Acumulatorului folosit va fi în funcţie de dimensiunea operandului sursă;
  - => dimensiunea operandului destinaţie va fi întotdeauna dublă faţă de cea a operandului sursă, destinaţia fiind o structură de tipul „acumulator extins”.

Instrucţiunea **IMUL** (***Signed Multiply***) realizează o înmulţire între doi operanzi ***numere cu semn***. Faţă de MUL, instrucţiunea IMUL suportă mai multe forme:

1) poate avea ***un singur operand*** (celălalt operand va fi Acc AL, AX, EAX sau RAX și va fi implicit), asemănător cu MUL

2) pot fi menţionaţi ***doi operanzi*** (sursă și destinaţie) în mod explicit, dar chiar și

3) o variantă cu ***trei operanzi***. Rezultatul obţinut este apoi salvat în destinaţie, care poate fi diferită în cele 3 cazuri.

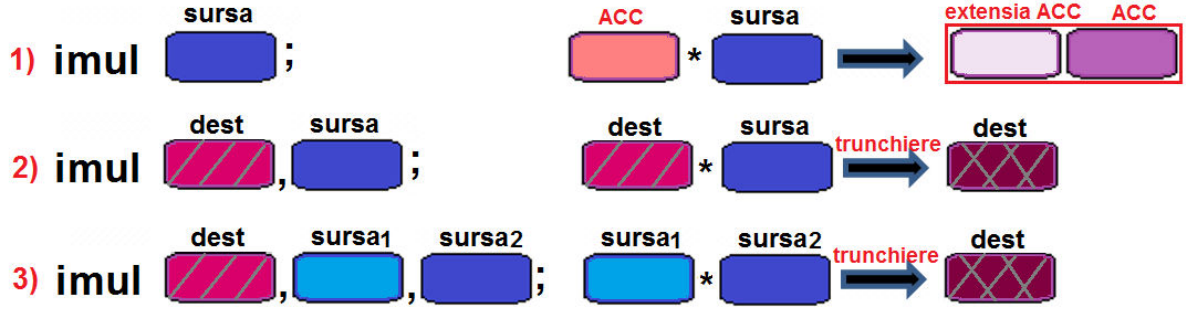


Figura 3-3.4. Ilustrarea modului de operare al instrucţiunii ***IMUL***

D.p.d.v. al tipului operanzilor, instrucţiunea IMUL la **8086** are o formă asemănătoare cu MUL, însă de la **80286**↑ au mai apărut şi alte forme cu 2 sau chiar 3 operanzi, astfel:

IMUL op1,op2 ; op1=op1*op2, cu IMUL {reg_{16}},{$reg_{8,16}$|$mem_{8,16}$|$imed_{8,16}$}
IMUL op1,op2,op3 ; op1=op2*op3, cu IMUL {reg_{16}},{reg_{16}| mem_{16}},{$imed_{8,16}$}

De la **80386**↑ au apărut următoarele forme suplimentare (pe lângă cele „moştenite" de la versiunea **80286**):

IMUL op1,op2 ; op1=op1*op2, cu IMUL {reg_{32}},{reg_{32}|mem_{32}|$imed_{8,32}$}
IMUL op1,op2,op3 ; op1=op2*op3, cu IMUL {reg_{32}},{reg_{32}| mem_{32}}, {$imed_{8,32}$}

Mai mult, de la **Pentium 4**↑ se mai poate realiza și:

IMUL op1,op2 ; op1=op1*op2, cu IMUL {reg_{64}},{reg_{64}| mem_{64}|$imed_{8,32}$}
IMUL op1,op2,op3 ; op1=op2*op3, cu IMUL {reg_{64}},{reg_{64}| mem_{64}}, {$imed_{8,32}$}

Observaţii:

- Trebuie subliniat că instrucţiunea IMUL consideră numerele implicate în operaţie ca fiind *numere cu semn*;
- de la **286**↑, se pot folosi și valori imediate, acestea fiind extinse cu semn la dimensiunea necesară realizării operaţiei;
- Instrucţiunea IMUL suportă 3 forme:
  - 1) *Forma cu un singur operand* - asemănătoare cu cea de la MUL, dimensiunea Acumulatorului folosit fiind în funcţie de dimensiunea operandului sursă; dimensiunea operandului destinaţie va fi întotdeauna dublă faţă de cea a operandului sursă, destinaţia fiind o structură de tipul „acumulator extins";
  - 2) *Forma cu 2 operanzi* - se va realiza produsul dintre cei doi operanzi specificaţi în instrucţiune, dar se va trunchia astfel încât să încapă în operandul destinaţie (op1);
  - 3) *Forma cu 3 operanzi* - se va realiza produsul dintre cei doi operanzi sursă specificaţi (cei mai din dreapta, op2 și op3) dar și acest produs se va trunchia astfel încât să încapă în operandul destinaţie (cel mai din stânga, op1); trebuie subliniat că cel de-al treilea operand trebuie să fie o valoare imediată;
- Instrucţiunea IMUL *nu afectează flagurile aritmetice* SF, ZF, PF, AF, dar
- *resetează flagurile* CF și OF (deci CF=OF=0) dacă:
  - jumătatea superioară a rezultatului este 0, pentru cazul 1);
  - la trunchiere s-au pierdut biţi de 1, pentru cazurile 2) și 3);

  altfel, aceste două flaguri sunt setate (deci CF=OF=1);

3.3.2. Instrucţiunile DIV şi IDIV

Procesoarele din familia 80x86 pot realiza împărţiri de valori pe 16 biţi la valori pe 8 biţi, de valori pe 32 biţi la valori pe 16 biţi, iar cele mai noi (de la **80386↑**) realizează împărţiri de valori pe 64 biţi la valori pe 32 biţi. În modul pe 64 biţi, se pot realiza chiar împărţiri de valori pe 128 biţi la valori pe 64 biţi.

Instrucţiunea **DIV** ***(Divide)*** realizează operaţia de împărţire *fără semn*, iar instrucţiunea **IDIV** ***(Integer Divide)*** realizează operaţia de împărţire *cu semn*. Instrucţiunile **DIV** şi **IDIV** realizează împărţirea ***acumulatorului extins*** (care va constitui *deîmpărţitul*) la operandul sursă specificat în instrucţiune (care va fi pe post de *împărţitor*).

De la **8086↑**, valorile operanzilor pot fi considerate ca numere fără semn, respectiv cu semn, instrucţiunea DIV respectiv IDIV împărţind ***valoarea aflată în registrul AX*** (un număr exprimat pe maxim 16 biţi) sau ***perechea de regiștri DX:AX*** (un număr exprimat pe maxim 32 biţi) ***la operandul sursă menţionat în instrucţiune***.

Rezultatul împărţirii se stochează sub formă de ***cât*** (în AL, resp. AX) și ***rest*** (în AH, resp. DX), așa cum arată Tabelul 3-3.1.

De la **8086↑**, deîmpărţitul poate fi exprimat pe 64 biţi și poate fi plasat în ***perechea de regiștri EDX:EAX***,

iar de la **Pentium 4↑**, poate fi chiar un număr pe 128 biţi, în ***perechea de regiștri RDX:RAX***;

astfel, ***câtul*** se va găsi în EAX, resp. RAX, iar ***restul*** în EDX, resp. RDX.

Tabel 3-3.1. Modul de lucru al instrucţiunilor ***DIV*** și ***IDIV***

Operanzi	*Deîmpărţit*	*Împărţitor*	*Rest*	*Cât*	*Cât la DIV*	*Cât la IDIV*
Word / Byte	**AX**	reg_8|mem_8	**AH**	**AL**	0…255	-128…+127
DoubleWord / Word	**DX:AX**	reg_{16}|mem_{16}	**DX**	**AX**	0…65.535	-32.768…+32767
QuadWord / DoubleWord	**EDX:EAX**	reg_{32}|mem_{32}	**EDX**	**EAX**	$0…2^{32}-1$	$-2^{31}…+2^{31}-1$
DoubleQuadWord / QuadWord	**RDX:RAX**	reg_{64}|mem_{64}	**RDX**	**RAX**	$0…2^{64}-1$	$-2^{63}…+2^{63}-1$

Instrucţiunea **DIV** realizează operaţia de împărţire ***fără semn***, iar **IDIV** realizează operaţia de împărţire ***cu semn***.

(I) DIV sursă *; Acc extins / sursă => Cât -> Acc , Rest -> Extensia Acc*
(I) DIV { reg$_{8/16/32/64}$ | mem$_{8/16/32/64}$ }

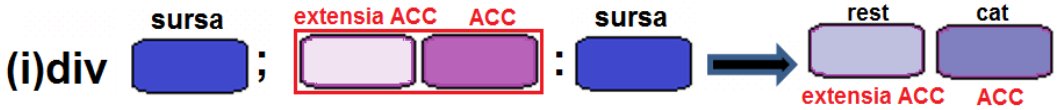


Figura 3-3.5. Ilustrarea modului de operare al instrucţiunii ***(I)DIV***

În urma împărţirii, ca o **regulă generală**:
- ***câtul*** va fi depus ***în acumulator***: (acumulator) = (acumulator extins) / (sursă)
- ***restul*** va fi în ***extensia acumulatorului***: (extensie acumulator) = (acumulator extins) MOD (sursă);

În particular, la realizarea ***împărţirii*** se va ţine cont de următoarele:

- dacă sursa este pe *octet* (reg8/mem8), atunci:
  deîmpărţitul este ***AX***, câtul va fi în registrul ***AL***, iar restul în registrul ***AH***;
- dacă sursa este pe *cuvânt* (reg16/mem16), atunci:
  deîmpărţitul este perechea ***DX:AX***, câtul va fi în ***AX***, iar restul în ***DX***;
- dacă sursa este pe dublucuvânt (reg32/mem32), atunci:
  deîmpărţitul este ***EDX:EAX***, câtul va fi în ***EAX***, iar restul în ***EDX***;
- dacă sursa este pe cvadruplucuvânt (reg64/mem64), atunci:
  deîmpărţitul este ***RDX:RAX***, câtul va fi în ***RAX***, iar restul în ***RDX*** (64 biţi).

Observaţii:

- Instrucţiunile DIV și IDIV folosesc un singur operand, care poate fi registru de uz general sau locaţie de memorie;
  - nu sunt acceptate valori imediate;
- În cadrul operaţiei de împărţire, flagurile nu au fost definite, deci *nu sunt afectate flagurile* CF, OF, SF, ZF, AF, PF.

Împărţirea presupune că lungimea deîmpărţitului este dublă faţă de cea a împărţitorului şi poate duce la depăşiri (CPU nu poate efectua operaţia de împărţire) când:

1) împărţitorul este zero - împărţind orice număr cu 0;

2) câtul depăşeşte dimensiunea rezervată rezultatului - valoarea câtului obţinut este prea mare pentru a putea fi scrisă în registrul alocat;

În urma unei depăşiri, se iniţiază o **întrerupere de nivel 0**, mai nou numită „excepţie". În IBM PC, dacă unul dintre aceste cazuri se întâmplă, va apărea eroarea numită "***Divide error***" sau „***Divide by zero***" cum mai e numită uneori.

La instrucțiunea DIV, dacă valoarea câtului obținut este: ***cât > FFh*** (în AL), deci la împărțirea cu un număr exprimat pe octet sau dacă valoarea câtului obținut este: ***cât > FFFFh*** (în AX) deci la împărțirea cu un nr. exprimat pe cuvânt se generează *întrerupere de nivel 0*.

Similar, la instrucțiunea IDIV, <u>dacă valoarea câtului obținut</u> este:

cât < -127 (81h) sau ***127 (7Fh) < cât***, câtul în AL deci la împărțirea cu un număr exprimat pe octet sau

cât < -32.767 (8001h) sau ***+32767 (7FFFh) < cât***, câtul în AX (la împărțirea cu un nr. exprimat pe cuvânt) se generează ***întrerupere de nivel 0***.

3.3.3. Exemple

Exemple de *instrucţiuni ilegale:*

```
mul 89h          ; MUL nu suportă operand imediat
mul AX, BL       ; MUL nu suportă forma cu mai mult de 1 operand

imul AX, BL      ; dimensiunile operanzilor la IMUL cu 2 operanzi trebuie să coincidă
imul AX, BL, CH  ; dimensiunile operanzilor la IMUL cu 3 operanzi trebuie să coincidă;
                 ; în plus, cel de-al treilea operand trebuie să fie un număr, o constantă sau valoare imediată
div 56h          ; DIV nu suportă operand imediat
idiv 56h         ; IDIV nu suportă operand imediat
div AX, BL       ; nici DIV şi nici IDIV nu suportă forma cu mai mult de 1 operand
```

Exemple de *instrucţiuni legale*:

Exemplul 3-3.1 În secvenţa următoare, deși în prima instrucţiune s-a încărcat registrul AL cu 0F0h ca un număr negativ (-16), valoarea 0F0h va fi interpretată ca un număr *fără semn* (adică 240) din cauza folosirii instrucţiunii MUL.

```
mov AL, -4       ; AL=FCh
mov BL, 2        ; BL=02h
mul BL           ; MUL -> FCh va fi interpretat ca nr fără semn 252, se va înmulţi cu 2 => AX = 01F8h = 504
```

Exemplul 3-3.2

```
mov AL,-4        ; AL=FCh
mov BL,2         ; BL=02h
imul BL          ; IMUL -> FCh va fi interpretat ca nr cu semn -4, se va înmulţi cu 2 => AX = FFF8h = -8
```

Exemplul 3-3.3

```
imul SI,2        ; înmulţeşte cu semn conţinutul registrului SI cu 2 şi depune rezultatul în registrul SI
                 ; rezultatul trunchiat se va găsi în registrul SI (de la 80286↑)
```

Exemplul 3-3.4

```
imul DX, CX, 20        ; înmulţeşte cu semn valoarea din CX cu 20 şi pune rezultatul trunchiat în reg. DX (de la 80286↑)
```

Exemplul 3-3.5

```
imul EAX, [4]          ; înmulţeşte cu semn conţinutul reg. EAX cu dublucuvântul din locaţia de memorie (segm DS) care
                       ; începe la adresa 4 (şi continuă la [5], [6], [7]), depunând rezultatul trunchiat în EAX (de la 80386↑)
```

Exemplul 3-3.6

```
imul ESI, EDI, 20      ; înmulţeşte cu semn conţinutul registrului EDI cu 20 şi depune rezultatul trunchiat în ESI (80386↑)
```

Exemple 3-3.7

```
idiv ECX               ; împarte cu semn conţinutul perechii EDX:EAX la conţinutul din ECX şi depune
                       ; câtul în EAX, iar restul în EDX
div RCX                ; conţinutul perechii RDX:RAX se divide la conţinutul din RCX,
                       ; câtul va fi în RAX, iar restul în RDX
div qword ptr [RDI]    ; conţinutul perechii RDX:RAX se divide cu cvadruplucuvântul din locaţia de memorie
                       ; adresată de registrul RDI, câtul se depune în RAX, iar restul în RDX
```

Exemplul 3-3.8 Se dorește înmulţirea a două valori (numere *fără semn*) aflate în memorie (stocate pe cuvânt) și apoi depunerea produsului obţinut tot în memorie. Pentru realizarea operaţiei, un operand va fi în AX, iar celălalt operand poate fi într-un registru sau în memorie. După realizarea operaţiei, perechea de regiștri DX:AX va conţine rezultatul (exact în această ordine, ca și cum ar fi un singur registru de 32 biţi). Astfel, DX va conţine cuvântul c.m.s. și AX va conţine cuvântul c.m.p.s.

```
val1 dw 8000           ; se defineşte variabila val1 în memorie, de tip word, cu valoarea 8000=1F40h
val2 dw 12000          ; se defineşte variabila val2 în memorie, de tip word, cu valoarea 12000=05B8h
rez dw 2 DUP(?)        ; se alocă spaţiu în memorie pentru variabila rez, 2 instanțe de tip word, adică 4 octeți
...
mov ax,val1            ; AX=val1=8000=1F40h
mul val2               ; se înmulţeşte AX cu val2=12000=2EE0h şi rezultatul se depune
                       ; în perechea de regiştrii DX:AX=05B8h:D800h, deci
                       ; DX:AX = 05B8h*2^16+ D800h= 1464*65536+55296=96.000.000
```

```
mov rez,ax          ; stochează rezultatul D800h la locațiile rez şi rez+1 în format LE
mov rez+2,dx        ; stochează rezultatul 05B8h la locațiile rez+2, rez+3 în format LE
```

Dacă am avea un registru de 32 biți, valoarea rezultată, 05B8D800h se transformă în zecimal ca numărul 96.000.000.

Exemplul 3-3.9

```
a db 7              ; variabila a pe octet inițializată cu 7
mov AX,72h          ; AX=72h=7h*10h+02h=114
div a               ; 114:7=16 rest 2 => AL=10h=16 (câtul), iar AH=02h=2 (restul)
```

Exemplul 3-3.10

```
b dw 300h           ; variabila b pe cuvânt inițializată cu 300h=768
mov AX, 0014h       ; AX=0014h =20
mov DX, 0003h       ; DX=0003h = 3
div b               ; accumulatorul extins este DX:AX, de valoare 3*16^4+20==3*65556+20=196628
                    ; -> se împarte la 768 şi se obține AX=0100h=256 (cât) şi DX=0014h=20 (rest)
```

Exemplul 3-3.11

```
mov AX, 0FFF8h      ; vrem să verificăm operaţia inversă celei executate în exemplul 3-3.2:
mov BL, 2           ; BL=02h = 2
idiv BL             ; va împărţi FFF8h (ca nr. signed, adică -8) la 2 şi depune câtul în AL= -4=FCh şi restul în AH=0
```

Exemplul 3-3.12 Similar exemplului precedent, vrem să verificăm operaţia inversă celei executate în exemplul 3-3.1**:**

```
mov AX, 0FFF8h      ; AX=FFF8h care va fi interpretat de data aceasta ca nr unsigned din cauza instrucţiunii div
mov BL, 2           ; BL=02h = 2
div BL              ; va împărţi FFF8h (ca nr. 65528) la 2 şi încearcă să depună câtul în AL= 32764,
                    ; dar 32764 se scrie ca numărul 7FFCh (nu va încăpea în AL), deci instrucţiunea nu va putea fi
                    ; executată pe procesor şi se va returna eroarea Divide overflow.
```

Pentru a evita astfel de posibile erori, se poate folosi următoarea metodă: se va translata operaţia de împărţire la următorul nivel al regiştrilor (dacă e disponibil pe procesorul respectiv).

În exemplul 3-3.12 se poate folosi ca deîmpărţit perechea de regiştri DX:AX în loc de AH:AL, iar atunci împărţirea se va realiza la BX (cu partea BH zerorizată sau extinsă cu semn – în funcţie de operaţia dorită- fără semn sau cu semn). Exemplul 3-3.12 se rescrie astfel:

Exemplul 3-3.13 Vrem să verificăm operaţia inversă celei executate în exemplul 3-3.1**:**

```
mov AX, 0FFF8h          ; AX=FFF8h care va fi interpretat de data aceasta ca nr unsigned din cauza instrucţiunii div
mov DX, 0               ; ştim că e nr unsigned, deci zerorizăm DX (partea superioară a deîmpărţitului)
mov BX, 2               ; BX=02h = 2 ; se foloseşte BX în loc de BL
div BX                  ; va împărţi 0000FFF8h (ca nr. 65528) la 2 şi va depune câtul în AX= 32764 = 7FFCh,
                        ; (acum încape în AL), deci instrucţiunea  nu va mai returna eroarea Divide overflow.
```

Exemplul 3-3.14 Forma 3) de la imul:

```
mov ECX, 0300 0000h     ; ECX = 0000 0011 0000 0000 0000 0000 0000 0000b =      +50.331.648 ca număr signed
imul EBX, ECX, 32       ; EBX = 011 0000 0000 0000 0000 0000 0000 0000b = +1.610.612.736 ca număr signed, CF=0
```

dar dacă valoarea înmulţitorului ar fi fost mai mare, atunci rezultatul ar fi fost trunchiat şi CF ar fi devenit 1:

```
imul EBX, ECX, 128      ; EBX = 1000 0000 0000 0000 0000 0000 0000 0000b = -2.147.483.648 ca număr signed CF=1
```

Exemplul 3-3.15 Forma 2) de la imul:

```
.data
a dd 3000h,200h
.code
mov EBX, a      ; EBX = 0003 0000h =       +196.608
imul EBX, a+4   ; EBX = 0600 0000h = +100.663.296 (adică +196.608 * 512)
```

Exemplul 3-3.16 Fie următoarea structură a regiştrilor: DX=1234h AX=FDE8h, BX=03E8h
Se presupune că se realizează în mod independent (separat) fiecare instrucţiune:

```
a1) mul BL      ; AL*BL-> AX=E8h*E8h, deci AX=D240h şi se interpretează ca numere unsigned: 232*232= 53.824
b1) mul BH      ; AL*BH-> AX=E8h*03h, deci AX=02B8h şi se interpretează ca numere unsigned: 232*3= 696
c1) mul BX      ; AX*BX-> DX:AX=FDE8h*03E8h, deci DX:AX=03DFh:D240h şi se interpretează ca numere unsigned:
                ; se operează 65.000*1000= 991:53824= 991*65.536+53.824=65.000.000
a2) imul BL     ; AL*BL-> AX=E8h*E8h, deci AX=0240h şi se interpretează ca numere signed: -24*(-24)= 576
b2) imul BH     ; AL*BH-> AX=E8h*03h, deci AX=FFB8h şi se interpretează ca numere signed: -24*3= -72
c2) imul BX     ; AX*BX-> DX:AX=FDE8h*03E8h, deci DX:AX=FFF7h:D240h şi se interpretează ca numere signed:
                ; se operează -536*1000= -53.600
```

Exemplul 3-3.17 Fie următoarea structură a regiştrilor: DX=1234h AX=FDE8h, BX=03E8h.
Se presupune că se realizează în mod independent (separat) fiecare instrucţiune:

```
a1) div BL      ; divide error – întrucât s-a încercat împărţirea valorii din AX ca nr fără semn la valoarea din BL,
                ; deci FDE8h/E8h=65.000/232 şi a rezultat câtul 280 care nu a încăput în AL
b1) div BH      ; divide error – întrucât s-a încercat împărţirea valorii din AX ca nr fără semn la valoarea din BH,
                ; deci FDE8h/03h=65.000/3 şi a rezultat câtul 21666 care nu a încăput în AL
c1) div BX      ; divide error – întrucât s-a încercat împărţirea valorii din DX:AX la BX ca nr fără semn
                ; deci DX:AX=1234h:FDE8h= 4660*65536+65.000=305.462.760/1000=305.462 care nu încape în AX
; cele de mai jos e indicat a fi verificate pe un CPU de 32 biţi, din cauza conversiei în zecimal:
a2) idiv BL
b2) idiv BH     ; divide error
c2) idiv BX
```

3.4. Instrucţiuni pentru comparare

Instrucţiunile de comparare sunt cele care compară doi operanzi și setează diferite flaguri ce indică rezultatele comparării.

De la **8086**↑ a fost disponibilă instrucţiunea **CMP**,

dar **CMPXCHG** și **CMPXCHG[8/16]B** au fost adăugate abia de la **Pentium**↑, respectiv **Pentium 4**↑.

Atât **CMPXCHG8B** cât și **CMPXCHG16B** sunt suportate pe un procesor (în general aput după Pentium) doar dacă bitul b8 (numit *cx8*, denumire care provine de la *Compare and eXchange 8B*, deci de la instrucţiunea mai scurtă) din registrul EDX este setat după execuţia instrucţiunii ***cpuid*** cu EAX=01h la intrare, deci se verifică dacă: EAX=00000001h --->***cpuid***---> $EDX_{b8=1}$.

În general, instrucţiunea CMP este urmată de o instrucţiune de salt condiţionat, de forma ***Jcc***, în vederea testării anumitor condiţii şi luării deciziilor în programe.

Pentru a analiza posibilele condiţii/ stări ce pot fi exploatate în vederea realizării salturilor, recomand revenirea la instrucţiunea CMOVcc (în capitolul precedent) unde au fost prezentate toate aceste situaţii. Singura diferenţă este că în locul prefixului mnemonicii CMOV, se va folosi **J** (de la ***jump if*** *(condiţie)*).

Este absolut necesar ca această instrucţiune de salt condiţionat să fie plasată în program imediat după instrucţiunea de comparare, nu mai târziu; instrucţiunea de comparare să nu fie urmată de o altă instrucţiune (care deci ar putea altera starea flagurilor) înainte de a testa condiţia de salt cu *jump if.*

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$_{8,16,32}$}

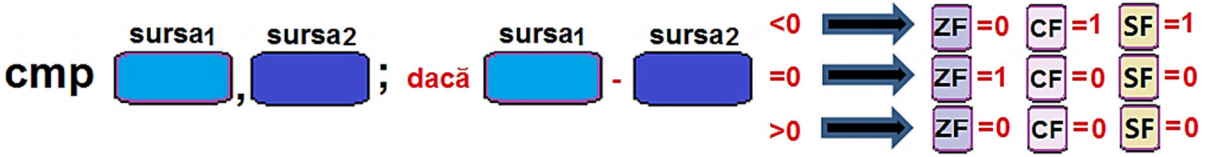


Figura 3-4.1. Ilustrarea modului de operare al instrucţiunii ***CMP***

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.

3.4.2. Instrucţiunea CMPXCHG

Instrucţiunea **CMPXCHG** ***(Compare and exchange***), disponibilă de la **486↑**, compară Acumulatorul cu operandul destinaţie; dacă Acumulatorul şi destinaţia sunt egale, atunci operandul destinaţie va copia valoarea operandului sursă şi va seta ZF=1. Altfel, operandul destinaţie se va încărca în Acumulator și se va reseta flagul Zero: ZF=0.

Operanzii pot avea 8, 16, 32 sau 64 biţi, deci Accumulatorul poate fi AL, AX, EAX sau RAX.

CMPXCHG destinaţie, sursă ***; dacă Acc = (destinaţie), atunci ZF=1 și (destinaţie) <- (sursă)***
CMPXCHG {reg$_{8,16,32,64}$| mem$_{8,16,32,64}$},{reg$_{8,16,32,64}$} ***; dacă Acc ≠ (destinaţie), atunci ZF=0 și ACC <- (destinaţie)***

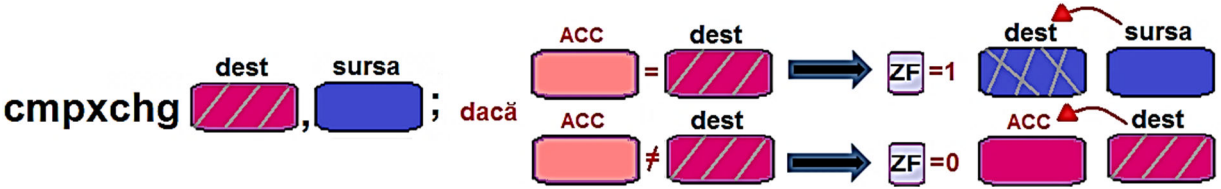


Figura 3-4.2. Ilustrarea modului de operare al instrucţiunii ***CMPXCHG***

Observaţii:

- Instrucţiunea *CMPXCHG modifică flagurile* SF, PF, AF, OF, CF conform rezultatului operaţiei de scădere fictivă care se realizează la comparare, dar *flagului ZF i se aplică regula*; dacă Acc =destinaţie, atunci ZF=1, altfel ZF=0;
- Operanzii pot fi atât numere fără semn cât și numere cu semn;
- Sintaxa și tipul operanzilor instrucţiunii trebuie respectate întocmai; astfel, sursa va fi întotdeauna registru de uz general, dar destinaţia poate fi și o zonă de memorie (nu se acceptă operanzi de tip imediat);
- Operanzii trebuie să aibă aceeași dimensiune: 8, 16, 32 sau 64 biţi.

3.4.3. Instrucţiunile CMPXCHG8B şi CMPXCHG16B

De la **Pentium 4 - Core 2↑** a apărut o instrucţiune asemănătoare cu CMPXCHG și anume **CMPXCHG8B** care compară cei 8 octeţi ai perechii EDX:EAX cu o zonă de memorie pe 64 biţi, denumită *mem64* și specificată în instrucţiune ca operand destinaţie. Dacă sunt egale, ZF=1 şi în acea zonă de memorie se încarcă conţinutul perechii ECX:EBX. Altfel, ZF=0 şi conţinutul zonei de memorie e încărcat în perechea EDX:EAX (care se mai numește și „Accumulator extins").

CMPXCHG8B destinaţie ***; dacă EDX:EAX = (destinaţie), atunci ZF=1 și (destinaţie) <- ECX:EBX***
CMPXCHG8B { mem64} ***; dacă EDX:EAX ≠ (destinaţie), atunci ZF=0 și EDX:EAX <- (destinaţie)***

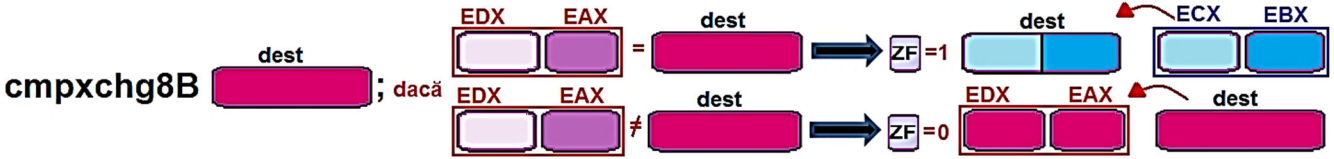


Figura 3-4.3. Ilustrarea modului de operare al instrucţiunii ***CMPXCHG8B***

De la **Pentium 4↑**, a apărut și instrucţiunea **CMPXCHG16B** pentru modul de lucru pe 64 biţi. Această instrucţiune compară cei 16 octeţi ai perechii RDX:RAX cu o zonă de memorie pe 128 biţi (specificată în instrucţiune ca operand destinaţie). Dacă sunt egale, ZF=1 şi în acea zonă de memorie se încarcă conţinutul perechii RCX:RBX. Altfel, ZF=0 şi conţinutul zonei de memorie e încărcat în perechea RDX:RAX (valabilă doar în mod pe 64 biţi).

CMPXCHG16B destinaţie ***; dacă RDX:RAX = (destinaţie), atunci ZF=1 și (destinaţie) <- RCX:RBX***
CMPXCHG16B { mem128} ***; dacă RDX:RAX ≠ (destinaţie), atunci ZF=0 și RDX:RAX <- (destinaţie)***

Observaţii:

- Instrucţiunile CMPXCHG[8/16]B nu afectează flagurile SF, PF, AF, OF, CF;
- în schimb, flagul ZF este setat în 1 dacă EDX:EAX = destinaţie; altfel, ZF=0.

3.4.4. Exemple

Exemple de *instrucţiuni ilegale:*

```
cmp [SI, 2]          ; dimensiunea operanzilor trebuie specificată, altfel e ambiguu => eroare
cmp AX, BL           ; dimensiunea operanzilor trebuie să coincidă
cmp val, [EBX]       ; nu se pot folosi ambii operanzi din memorie
cmpxchg EAX, val     ; operandul sursă nu poate fi din memorie, doar registru
cmpxchg8b EAX, EBX   ; sintaxă eronată, trebuie un singur operand
val1 dw 1234h        ; se definește val1 de tip word în memorie
val2 dd 12345678h    ; se definește val2 de tip doubleword în memorie

...
cmpxchg8b val1       ; val1 e de tip word, iar cmpxchg8b așteaptă operand de tip doubleword din memorie mem64
cmpxchg16b val2      ; val2 e de tip doubleword, iar cmpxchg16b așteaptă operand quadword mem128b
cmpxchg8b EAX        ; dimensiune potrivită, dar nu se acceptă operand registru
cmpxchg16b RAX       ; dimensiune potrivită, dar nu se acceptă operand registru
```

Exemple de *instrucţiuni legale*:

Exemplul 3-4.1

```
mov AX, 2000h        ; AX=2000h
mov BX, 400h         ; BX=400h
cmp AX, BX           ; se realizează scăderea AX-BX=1C00h, dar fictiv deci AX=2000h, BX=400h şi SF=0, CF=0, OF=0;
```

Exemplul 3-4.2

```
mov AX, 2000h        ; AX=2000h
mov BX, 400h         ; BX=400h
cmp BX, AX           ; se realizează scăderea BX-AX=E400h, dar fictiv deci AX=2000h, BX=400h şi SF=1, CF=1, OF=0;
```

Exemplul 3-4.3

```
.data
a db 5,4,3
.code
lea SI, a
mov AL, 2
cmp AL, [SI]            ; compară AX cu octetul adresat de SI în memorie, adică va compara 2 cu 5 şi va seta flagurile
```

astfel: CF=1 (ca borrow), ZF=0, SF=1, AF=1, PF=0, OF=0, deoarece 2-5 = -3 = 0000.0010b-0000.0101b = 1111.1101b=FDh

Exemplul 3-4.4

```
cmpxchg [a+4], EBX      ; compară dublucuvântul din segmentul de date adresat de a+4 cu EBX
                        ; dacă sunt egale =>ZF=1 şi în locaţia adresată de [a+4], se depune conţinutul lui EBX
                        ; dacă sunt diferite => ZF=0 şi în EBX se depune conţinutul memoriei dat de a+4
                        ; (a fost definit ca vector cu elemente de tip doubleword)
```

Exemplul 3-4.5 Fie valorile regiştrilor: EAX=05060708h, EBX=0FFFF0000h, ECX=12345678h şi EDX=01020304h. Se execută următoarea secvenţă (se execută de 2 ori consecutiv instrucţiunea *cmpxchg8b a*):

```
.data
a dq 0102030405060708h
.code
cmpxchg8b a             ; dacă a=EDX:EAX, ZF=1 și valoarea lui a  se va înlocui cu valoarea găsită în ECX:EBX;
                        ; astfel, variabila a va deveni 12345678FFFF0000h şi ZF=1
cmpxchg8b a             ; dacă val e diferită de EDX:EAX, ZF=0 și valoarea lui a se va încărca în perechea EDX:EAX
                        ; astfel, se schimbă valorile regiştrilor: EDX=12345678h, EAX=FFFF0000h
```

Exemplul 3-4.6

```
cmpxchg16b [val]        ; compară RDX:RAX=(128 biţi) din memorie daţi de val
                        ; dacă sunt egali =>ZF=1 şi în locaţia adresată de val se depune conţinutul RCX:RBX
                        ; dacă sunt diferiţi =>ZF=0 şi în RDX:RAX se depune conţinutul memoriei dat de val
```

3.5. Instrucţiuni pentru extinderea semnului ACC

Aceste instrucţiuni nu au nici un operand (lucrează implicit cu acumulatorul (ACC)) şi nu afectează nici un flag. Deși nu este specificat în mod explicit nici un operand, implicit se consideră existenţa unui operand sursă (de la care se pleacă sau asupra căruia se aplică extinderea) și a unui operand destinaţie (în care se va regăsi rezultatul după realizarea operaţiei).

Aceste instrucţiuni se folosesc în general înaintea operaţiilor de împărţire sau atunci când se dorește scalarea unui operand la o dimensiune dublă. Efectul lor este de a extinde cu semn o valoare aflată în registrul AL, AX, EAX sau RAX.

Tabel 3-5.1. Sintetizarea instrucţiunilor ***CWD***, ***CDQ***, ***CQO*** și ***CBW***, ***CWDE***, ***CDQE***

Instrucţiunea		Operaţia
Word to Doubleword	CWD	DX:AX ← extensia cu semn (AX)
Doubleword to Quadword	CDQ	EDX:EAX← extensia cu semn (EAX)
Quadword to Double-Quadword	CQO	RDX:RAX← extensia cu semn (RAX)
Byte to Word	CBW	AX ← extensia cu semn (AL)
Word to Doubleword Extended	CWDE	EAX ← extensia cu semn (AX)
Doubleword to Quadword Extended	CDQE	RAX ← extensia cu semn (EAX)

Instrucţiunile CWD, CDQ și CQO dublează dimensiunea operandului specificat în registrul AX, EAX, respectiv RAX (în funcţie de dimensiunea operandului) prin extensie de semn și stochează rezultatul astfel obţinut în perechea de regiștri DX:AX, EDX:EAX, respectiv RDX:RAX.

Instrucţiunile CBW, CWDE și CDQE dublează dimensiunea operandului sursă prin extinderea semnului în registrul acumulator corespunzător având dimensiune dublă (deci „extins") AX, EAX, respectiv RAX.

Instrucţiunile CBW și CWD au fost disponibile încă de la **8086**↑, dar CWDE și CDQ abia de la **80386** ↑, iar CDQE și CQO doar de la **Pentium 4**↑.

3.5.1. Instrucţiunea CWD

Instrucţiunea **CWD** (***Convert Word to Doubleword***) converteşte cuvântul din registrul AX la dublu-cuvântul din perechea de regiștri DX:AX, deci copiază bitul de semn (bitul 15) al operandului sursă specificat în registrul AX în fiecare bit din cei 16 ai registrului DX; astfel, bitul de semn din AX se extinde şi la registrul DX (numit și extensia acumulatorului).

Instrucţiunea CWD a fost disponibilă încă de la **8086**, iar la procesorul 8086 nu existau regiştri de 32 biţi necesari la stocarea unui doubleword; astfel, modalitatea de a stoca valori de 32 biţi a fost prin concatenarea regiştrilor DX:AX, aşa cum s-a văzut la realizarea înmulţirii cu operanzi pe 16 biţi.

```
CWD             ; dacă AX15 = 1 → DX = 0FFFFh,      altfel DX = 0000h
```

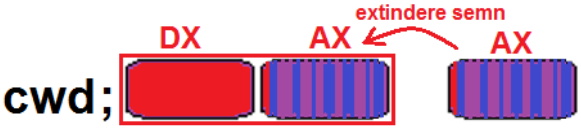


Figura 3-5.1. Ilustrarea modului de operare al instrucţiunii ***CWD***

Observaţii:

- Instrucţiunea CWD *nu afectează nici un flag aritmetic*; instrucţiunea cwd se aseamană mai mult cu instrucţiunile de transfer (din capitolul precedent, care nu afectează [-/E/R]FLAGS);
- CWD nu are operanzi, lucrează implicit cu acumulatorul de tip word, adică AX (ca sursă), iar ca destinaţie se folosește perechea de regiștri ce formează acumulatorul extins (dublu ca dimensiune), deci DX:AX.

3.5.2. Instrucţiunea CDQ

Instrucţiunea **CDQ** (***Convert Doubleword to Quadword***) recunoscută de procesoarele **80386↑**, converteşte dublucuvântul din registrul EAX la cvadruplucuvântul din perechea de regiștri EDX:EAX, deci copiază bitul de semn (bitul 31) al operandului sursă specificat în registrul EAX în fiecare bit din cei 32 ai registrului EDX; astfel, bitul de semn din EAX se extinde şi la EDX.

Instrucţiunea CDQ a fost disponibilă încă de la **80386**, iar la procesorul 80386 nu existau regiştri de 64 biţi necesari la stocarea unui quadword; astfel, modalitatea de a stoca valori de 64 biţi a fost prin concatenarea regiştrilor EDX:EAX, aşa cum s-a văzut la realizarea înmulţirii cu operanzi pe 32 biţi.

CDQ ; dacă EAX_{31} = 1 → EDX = 0FFFFFFFFh, altfel EDX = 00000000h

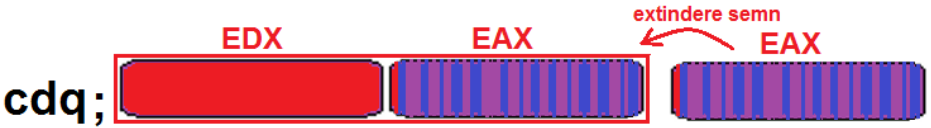


Figura 3-5.2. Ilustrarea modului de operare al instrucţiunii ***CDQ***

Observaţii:

- Instrucţiunea CDQ *nu afectează nici un flag aritmetic*;
- CDQ nu are operanzi, lucrează implicit cu acumulatorul de tip doubleword, adică EAX (ca sursă), iar ca destinaţie se folosește perechea de regiștri ce formează acumulatorul extins (dublu ca dimensiune) EDX:EAX.

3.5.3. Instrucţiunea CQO

Instrucţiunea **CQO** (***Convert Quadword to Doublequadword***) recunoscută de procesoarele de la **Pentium 4↑**, converteşte cvadruplucuvântul din registrul RAX la dublucvadruplucuvântul din perechea de regiștri RDX:RAX, prin copierea bitului de semn (bitul 63) al operandului sursă specificat în registrul RAX în fiecare bit din cei 64 ai registrului RDX; astfel, bitul de semn din RAX se extinde şi la RDX.

Instrucţiunea CQO a fost disponibilă încă de la **procesoare pe 64 biţi**, iar acestea nu dispun de regiştri de 128 biţi necesari la stocarea unui doublequadword; astfel, modalitatea de a stoca valori de 128 biţi a fost prin concatenarea regiştrilor RDX:RAX, aşa cum s-a văzut la realizarea înmulţirii cu operanzi pe 64 biţi.

```
CQO          ; dacă RAX63 = 1 → RDX = 0FFFFFFFFFFFFFFFFh,    altfel RDX = 0000000000000000h
```

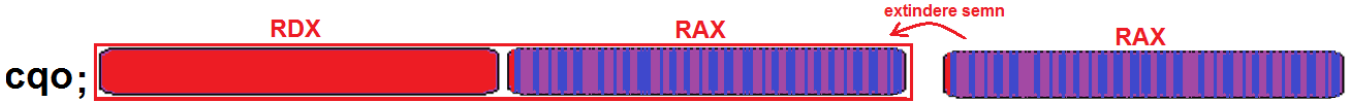


Figura 3-5.3. Ilustrarea modului de operare al instrucţiunii ***CQO***

Observaţii:

- Instrucţiunea CQO *nu afectează nici un flag aritmetic*;
- CQO nu are operanzi, lucrează implicit cu acumulatorul de tip quadword, adică RAX (ca sursă), iar ca destinaţie se folosește perechea de regiștri ce formează acumulatorul extins (dublu ca dimensiune) RDX:RAX.

3.5.4. Instrucţiunea CBW

Instrucţiunea **CBW** (***Convert Byte to Word***) converteşte octetul din registrul AL la cuvântul din AX prin copierea bitului de semn (bitul 7) al operandului sursă specificat în registrul AL în fiecare bit al registrului AH; astfel, bitul de semn din AL se extinde la tot registrul AH, stocând deci valoarea *cu semn* din registrul AL în tot registrul AX.

Instrucţiunea CBW a fost disponibilă încă de la **8086** şi încă este suportată pe procesoarele actuale, deşi au mai apărut şi instrucţiunile MOVSX şi MOVZX (prezentate deja în capitolul precedent şi care realizează extensia cu semn sau fără semn). Spre deosebire de acestea, instrucţiunea CBW consideră întotdeauna operanzii ca valori numere *cu semn*.

CBW ; dacă $AL_7 = 1 \rightarrow AH = 0FFh$, altfel AH = 00h

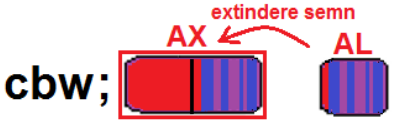


Figura 3-5.4. Ilustrarea modului de operare al instrucţiunii ***CBW***

Observaţii:

- Instrucţiunea CBW *nu afectează nici un flag aritmetic*;
- CBW nu are operanzi, lucrează implicit cu acumulatorul de tip byte, adică AL (ca sursă), iar ca destinaţie se foloseşte acumulatorul (dublu ca dimensiune), deci AX.

3.5.5. Instrucţiunea CWDE

Instrucţiunea **CWDE** (***Convert Word to Doubleword Extended***) copiază bitul de semn (bitul 15) al operandului sursă specificat în registrul AX în fiecare bit al părţii superioare (c.m.s. 16 biţi) a registrului EAX.

Instrucţiunea CWDE a fost disponibilă încă de la **80386** şi încă este suportată pe procesoarele actuale, cu observaţia similară de la instrucţiunea CBW: au mai apărut şi instrucţiunile MOVSX şi MOVZX (prezentate deja în capitolul precedent şi care realizează extensia cu semn sau fără semn). Spre deosebire de acestea, instrucţiunea CWDE consideră întotdeauna operanzii ca valori numere *cu semn*.

CWDE ***; dacă*** $AX_{15} = 1 \rightarrow EAX_{31...16} = 0FFFFh$, ***altfel*** $EAX_{31...16} = 0000h$

Figura 3-5.5. Ilustrarea modului de operare al instrucţiunii ***CWDE***

Observaţii:

- Instrucţiunea CWDE *nu afectează nici un flag aritmetic*;
- CWDE nu are operanzi, lucrează implicit cu acumulatorul de tip word, adică AX (ca sursă), iar ca destinaţie se foloseşte acumulatorul (dublu ca dimensiune), deci EAX.

3.5.6. Instrucţiunea CDQE

Instrucţiunea **CDQE** (***Convert Doubleword to Quadword Extended***) copiază bitul de semn (bitul 31) al operandului sursă specificat în registrul EAX în fiecare bit al părţii superioare (c.m.s. 32 biţi) a registrului RAX.

Instrucţiunea CDQE a fost disponibilă doar la procesoare pe 64 biţi, unde existau regiştri de o astfel de dimensiune; observaţia de la instrucţiunea CBW rămâne totuşi valabilă: deşi există instrucţiunile MOVSX şi MOVZX care realizează extensia cu semn sau fără semn, spre deosebire de acestea, instrucţiunea CDQE consideră întotdeauna operanzii ca valori numere *cu semn*.

CDQE ***; dacă EAX_{31} = 1 → $RAX_{63...32}$ = 0FFFFFFFFh,*** ***altfel $RAX_{63...32}$ = 00000000h***

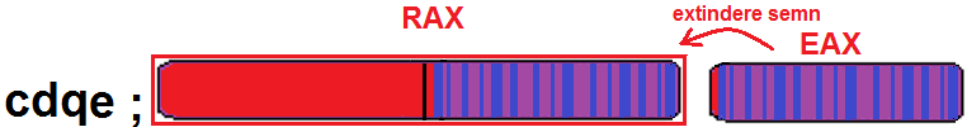


Figura 3-5.6. Ilustrarea modului de operare al instrucţiunii ***CDQE***

Observaţii:

- Instrucţiunea CDQE *nu afectează nici un flag aritmetic*;
- CDQE nu are operanzi, lucrează implicit cu acumulatorul de tip doubleword, adică EAX (ca sursă), iar ca destinaţie se foloseşte acumulatorul (dublu ca dimensiune), deci RAX.

3.5.7. Exemple

Exemple de *instrucţiuni ilegale:*

```
cbw BL          ; sintaxa eronată, CBW nu are operanzi expliciţi
cwd BX          ; sintaxa eronată, CWD nu are operanzi expliciţi
cdq ECX         ; sintaxa eronată, CDQ nu are operanzi expliciţi
cqo RDX         ; sintaxa eronată, CQO nu are operanzi expliciţi
cwde BX         ; sintaxa eronată, CWDE nu are operanzi expliciţi
cdqe EDX        ; sintaxa eronată, CDQE nu are operanzi expliciţi
```

Exemple de *instrucţiuni legale*:

Exemplul 3-5.1

```
mov AX, 04h     ; AX=9804h
mov BL, 0ABh    ; BL=ABh
cbw             ; AX=0004h, adică valoarea din registrul AL s-a extins cu semn la tot registrul AX
                ; deseori am întâlnit confuzia că ar fi vorba despre octetul specificat în instrucţiunea anterioară
                ; s-ar putea crede în mod eronat că instrucţiunea cbw afectează valoarea din BL ! (fals)
```

Exemplul 3-5.2

```
mov DX, 4321h   ; DX=4321h
mov AX, 1285h   ; AX=1285h, deci AH=12h şi AL=85h
cbw             ; AX=FF85h
cwd             ; DX:AX=FFFF FF85h, deci DX=FFFFh şi AX=FF85h
```

Exemplul 3-5.3

```
mov DX, 1035h   ; DX=1035h
mov AX, 7654h   ; AX=7654h
cbw             ; AX=0054h
cwd             ; DX:AX=0000 0054h, deci DX=0000h şi AX=0054h
```

Exemplul 3-5.4

```
mov EAX, 12345678h      ; EAX=12345678h
mov AL, 0A5h            ; AL=0A5h, deci EAX=123456A5h
cbw                     ; AX=FFA5h, deci EAX=1234FFA5h
cwde                    ; EAX=FFFFFFA5h
```

Exemplul 3-5.5

```
mov RAX, 0              ; RAX=0000 0000 0000 0000h
mov EAX, 12345678h      ; EAX=12345678h, RAX=0000000012345678h
mov AX, 0A5h            ; AX=00A5h, EAX=12340056A5h
cwde                    ; EAX=FFFF FFA5h
cdq                     ; EDX=FFFF FFFFh
cdqe                    ; RAX= FFFF FFFF FFFF FFA5h
cqo                     ; RDX=FFFF FFFF FFFF FFFFh
```

Exemplul 3-5.6 Să presupunem că în EBX este o valoare ce se doreşte a fi împărțită la valoarea din ECX. Pentru a putea realiza această împărțire, prima dată trebuie adus deîmpărțitul la dimensiune dublă, adică la 64 biți.

```
mov EAX, EBX            ; valoarea deîmpărțitului se copiază în EAX
cdq                     ; se extinde cu semn la dimensiunea de 64 biți, noua valoare se va găsi în perechea EDX:EAX
idiv ECX                ; valoarea din perechea EDX:EAX se împarte la val. din ECX => în EDX va fi restul, iar în EAX câtul
```

Exemplul 3-5.7

```
.data
a dw 9876h, 5678h
.code
mov AX, a               ; AX = 9876h = -26506
mov BX, a+2             ; BX = 5678h = 22136
cwd                     ; DX:AX = FFFF:9876h = -26506
idiv BX                 ; se împarte DX:AX la BX, deci -26506 la 22136 şi se obţine
                        ; câtul în AX = -1=FFFFh, iar restul în DX =EEEEh = -4370
```

3.6. Instrucţiuni pentru corecţia ACC

BCD (Binary Coded Decimal) a fost unul dintre primele coduri folosite în sistemele de calcul; inventat de IBM, a fost folosit în calculatoarele lor încă din anii '50-'60 (sistemele 704, 7040, 709, 7090).

În anumite aplicaţii (precum afişaje LCD, reprezentări pe digiţi), în general în sisteme embedded (autovehicule, cuptoare cu afişaj electronic, ceasuri cu alarmă, etc) datele numerice se folosesc *în formă zecimală* şi se preferă utilizarea codului (codificării) ***zecimal codificat în binar BCD* (Binary Coded Decimal).** Astfel, pot fi evitate conversiile repetate din zecimal în binar şi invers. De câte ori se menţionează termenul de *valori BCD* în cadrul instrucţiunilor implementate în CPU (cu referire la cele originale ale 8086) trebuie folosite doar cifrele zecimale, de la 0 la 9.

Codificarea BCD folosește 4 biţi pentru a reprezenta cele 10 cifre zecimale {0, 1, ... 9}, asemănător cu reprezentarea hexazecimală, dar se folosesc numai primele 10 combinaţii de biţi (câte cifre sau numere zecimale există).

Acestea sunt: 0_{10} -> 0000_2, 1_{10} -> 0001_2, 2_{10} -> 0010_2, ... 9_{10} -> 1001_2

celelalte combinaţii fiind nepermise (precum 1010, ... 1111).

În formatul BCD se folosesc doar 10 combinaţii în loc de 16 cât ar fi posibile pe un nibble (un digit), ducând astfel la o stocare ineficientă; un alt dezavantaj este că toate calculele în format BCD sunt mai lente decât cele în binar.

În sistemele electronice care realizează prelucrări de numere, este des întâlnită ***codificarea binary-coded decimal (BCD)*** adică ***zecimal codificat ca binar***. Aceasta este un tip de codificare a numerelor zecimale în formă binară, în care fiecare digit zecimal se reprezintă pe un număr fix de biţi: **4 (forma împachetată)** sau **8 (forma despachetată)**. BCD a fost utilizat pe scară largă în trecut, dar sistemele mai noi nu au mai implementat instrucţiunile specifice (de exemplu cele cu CPU de tip ARM).

Familia de procesoare x86 are implementate încă aceste instrucţiuni, deși nu au mai fost optimizate pentru viteză. Utilizarea cea mai des întâlnită este în aplicaţiile din domeniul financiar, comercial și industrial, unde sunt necesare diverse calcule.

De exemplu, într-un sistem electronic unde trebuie afișată o valoare numerică, manipularea datelor numerice este mult simplificată (versus exploatarea valorilor în binar) prin tratarea fiecărui digit ca un subcircuit electronic separat, implementat de exemplu cu afișaje de tip 7-segmente; această situaţie este mai apropiată de realitatea fizică sau hardware-ul sistemului.

⇨ în SC unde calculele sunt relativ simple (adunări, scăderi, etc), lucrul cu valori BCD poate simplifica mecanismul de implementare al întregului sistem (vs. conversia din zecimal în binar, efectuarea de calcule și apoi conversia înapoi în zecimal pentru afișare). Calculatorul de buzunar e un exemplu sugestiv în acest sens.

Exemplu: un inginer proiectant sau un programator va lucra cu memoria internă, cu CPU, deci va reprezenta valorile în regiștri pentru a le opera; deci intern, în interiorul SC vom lucra cu valori binare, deși utilizatorul va vedea aceste valori în zecimal; problema majoră care se pune este dacă putem simplifica modul de lucru și de implementare al sistemului, pentru a realiza cât mai puţine din conversiile necesare.

Multiplele astfel de conversii din formatul extern SC (de tip Ascii) și cel intern SC (de tip binar) pot fi evitate folosind instrucţiuni BCD. Aceste instrucţiuni pot ajuta mult atunci când reprezentăm valorile pe digiţi zecimali – din interiorul SC, pe un sistem care reprezintă doar valori zecimale sau doar preia valori zecimale pe care apoi intern le prelucrează.

Procesoarele Intel din familia x86 au instrucţiuni în limbaj de asamblare care suportă operaţii aritmetice în reprezentarea BCD, adică valorile pot fi considerate numerele cu care ne-am obișnuit noi oamenii din școala primară: pe un digit: 0...9, pe 2 digiţi: 00..99, și așa mai departe.

Valorile cifrelor zecimale pot fi codificate BCD:

- **individual - fiecare cifră pe câte un octet** (forma ***despachetată***)
  Exemplu: nr **53**h se va scrie **0000 0101 0000 0011**b
- **împreună, câte 2 cifre pe un octet** (forma ***împachetată***)
  Exemplu: nr **53**h se va scrie **0101 0011**b

Aritmetica BCD: operaţiile realizate în aritmetică BCD se efectuează pentru valori exprimate în forma împachetată, adică 2 digiţi BCD pe un octet. Astfel, se poate realiza corecţia zecimală („***Decimal adjust***") după ***adunarea***, respectiv după ***scăderea*** a două valori exprimate pe octet în formă BCD împachetată; operaţiile corespunzătoare sunt implementate pe CPU folosind instrucţiunile **DA*A*** și **DA*S***.

Tabelul 3-6.1. Exemplificarea operaţiilor de *ajustare Zecimală*

Operaţia	**Adunare** BCD împachetat	**Scădere** BCD împachetat
Exemplificare	54h+	54h-
	19h	19h
	6Dh	3Bh
După corecţie	73h	35h

Aritmetica Ascii: operaţiile în aritmetică Ascii se realizează pentru valori exprimate în forma despachetată, adică 1 singur digit BCD se va exprima pe un octet (doar nibble-ul c.m.p.s.). Este posibilă corecţia Ascii („**Ascii adjust**") după **adunarea**, **scăderea**, **înmulţirea** și respectiv înainte de **împărţirea** valorilor exprimate pe octet în forma BCD împachetată; operaţiile corespunzătoare sunt implementate în EMU folosind instrucţiunile **AAA**, **AAS**, **AAM** și **AAD**.

Tabelul 3-6.2. Exemplificarea operaţiilor de *ajustare Ascii*

Operaţia	**Adunare** BCD despachetat	**Scădere** BCD despachetat	**Înmulţire** BCD despachetat	**Împărţire** BCD despachetat
Exemplificare	0504h+ 0109h 060Dh	0504h- 0109h 03FBh	06h* 07h 42	0208h pregătește pt împărţire
După corecţie	0703h	0205h+0100h= 0305h	0402h	28 = 001Ch

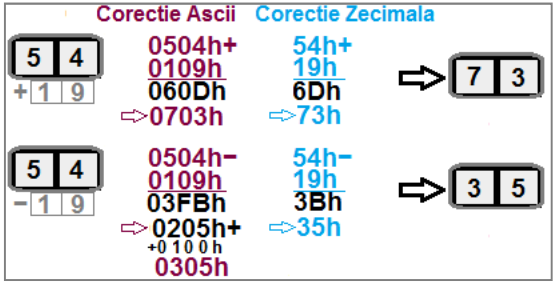


Figura 3-6.1. Exemplificarea operaţiilor de corecţie Ascii și Zecimală

Dacă, de exemplu la scădere, realizăm operaţia 28h-19h, după corecţia zecimală se va obţine 09h, dar dacă se va scădea 18h-19h, după corecţia zecimală se va obţine 99h; este ca și cum ar fi avut loc un împrumut din exteriorul domeniului de reprezentare al valorii – Borrow; în loc de FFh, în zecimal avem 99.

Folosirea operaţiilor de corecţie Ascii:

Exemplu: Să presupunem că a fost efectuată o adunare care a dus la obţinerea rezultatului 15 în zecimal care se scrie 0Fh sau 00F0h; de exemplu s-au adunat două valori ca format împachetat 9+6 și am depășit cifra BCD; ca să putem reprezenta tot cu BCD, vom folosi o operaţie de *corecţie Ascii după adunare*, astfel:

000**F**h = **15**d (prin operaţia de corecţie Ascii după adunare)-> **0105**h

Exemplu: Se presupune că am preluat de la tastatură 2 valori zecimale, de exemplu am preluat tastele 1 și 5, adică vom avea ‚1' și ‚5' și dacă scădem din ele ‚0', obţinem 1 și 5 -> scrise în hexazecimal acestea vor furniza valoarea 0105h (care se va scrie pe 16 biţi); pentru a forma această valoare ca numărul 15d =0Fh (adică scris pe 8 biţi numărul cincisprezece), putem proceda în următorul mod: **0105**h (prin operaţie de *ajustare Ascii înainte de împărţire*)->se obţine **15** = **0F**h

adică l-am transformat într-o valoare ce se poate scrie intern pe un singur octet.

Exemplu: **0205**h (prin operaţie de *ajustare Ascii înainte de împărţire*)-> **25** = **19**h

Acest tip de prelucrări ne poate folosi atunci când preluăm valori zecimale de la tastatură și apoi vrem să le prelucrăm; de ex., l-am întrebat pe utilizator vârsta sa și vrem să o verificăm; în funcţie de aceasta, îi cerem sau nu informaţii suplimentare

Exemplu: Invers decât în exemplele anterioare, dacă de exemplu, în funcţie de datele de la angajator am calculat vârsta la care un utilizator poate intra la pensie și vrem să afișăm aceste informaţii pe ecran, atunci intern avem valoarea 0041h =65 și ne pregătim să apelăm o funcţie de afișare care să scrie pe ecran un 6 urmat de un 5, adică vom folosi codurile Ascii ale lor, mai exact: ‚6','5', fiecare scris pe câte 8 biţi, ca 36h 35h.

Astfel: 0041h=**65** (prin operaţie de *corecţie Ascii după înmulţire*) -> **0605**h și apoi vom aduna 3030h, obţinând 3635h, adică ‚65' scris pe 16 biţi; nu recomand această scriere întrucât în general se comit erori la interpretarea și scrierea valorilor în memoria sistemului ca și coduri Ascii.

Operaţii cu numere în format BCD împachetat (2 cifre pe octet): numărul 1234, în format BCD împachetat este stocat ca 12h 34h. Deşi operanzii şi rezultatele obţinute după realizarea operaţiei (adunare, scădere, înmulţire, împărţire) sunt stocate în regiştrii în format hexazecimal, valorile şi operarea trebuie văzute în format zecimal.

39h+ 39+
58h 58
91h -> 97 (obţinut prin adunarea lui 6)

16h+ 16+
45h 45
5Bh -> 61 (obţinut prin adunarea lui 6)

73h+ 73+
41h 41
B4h -> 14 (obţinut prin adunarea lui 60), CF=1

În loc de 91h, în format BCD împachetat trebuie obţinut 97 (cât este suma în zecimal a valorilor văzute în zecimal); analog, în loc de 5Bh trebuie obţinut 61, iar în loc de B4h trebuie obţinut 114 (corect dacă se ţine cont de CF=1).

Operaţii cu numere în format BCD despachetat (o cifră pe octet): numărul 1234, în format BCD despachetat este stocat ca 01h 02h 03h 04h, dar se aplică aceleaşi reguli la realizarea operaţiilor, asemănător cazului BCD împachetat.

3.6.1. Instrucţiunea DAA

Instrucţiunea **DAA** (***Decimal Adjust AL after BCD Addition***) ajustează conţinutul registrului AL (conţinut presupus a fi obţinut în urma unei instrucţiuni **ADD** asupra ***a 2 valori BCD împachetate***, deci fiecare scrise cu 2 digiţi) pentru a produce un rezultat corect (legal) în format BCD împachetat.

Instrucţiunea nu are operanzi expliciţi; se folosește registrul AL atât ca operand sursă (din AL se ia valoarea asupra căreia se aplică acea corecţie zecimală) cât și ca operand destinaţie (tot în AL se depune rezultatul obţinut).

DAA ; dacă $AL_{3...0}$ > 9 sau AF = 1 → AL=AL+6h și AF = 1, altfel → AF = 0
; dacă $AL_{7..4}$ > 9 sau CF = 1 → AL=AL+60h și CF = 1, altfel → CF = 0

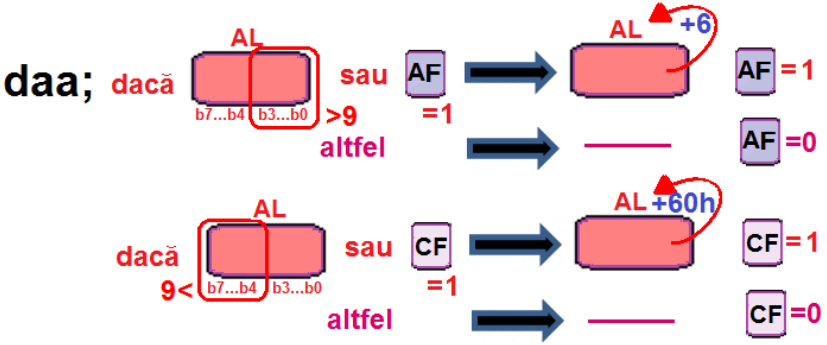


Figura 3-6.1. Ilustrarea modului de operare al instrucţiunii ***DAA***

Observaţii:

- Instrucţiunea DAA nu are operanzi expliciţi, lucrează implicit cu registrul AL;
- Această instrucţiune *setează flagurile* CF și AF dacă la ajustare a apărut un transport zecimal la vreunul din digiţii rezultatului; flagurile SF, ZF, PF *se setează în funcţie de rezultat*; flagul OF *nu este definit*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.2. Instrucţiunea DAS

Instrucţiunea **DAS** (***Decimal Adjust AL after Subtraction***) ajustează conţinutul registrului AL (conţinut presupus a fi obţinut în urma unei instrucţiuni **SUB** între ***2 valori BCD împachetate***, deci fiecare scrise cu 2 digiţi) pentru a produce un rezultat corect în format BCD împachetat. Instrucţiunea nu are operanzi expliciţi; se folosește registrul AL atât ca operand sursă (din AL se ia valoarea asupra căreia se aplică acea corecţie zecimală) cât și ca operand destinaţie (tot în AL se depune rezultatul obţinut).

DAS *; dacă $AL_{3...0}$ > 9 sau AF = 1 → AL=AL-6h și AF = 1, altfel → AF = 0*
; dacă $AL_{7..4}$ > 9 sau CF = 1 → AL=AL-60h și CF = 1, altfel → CF = 0

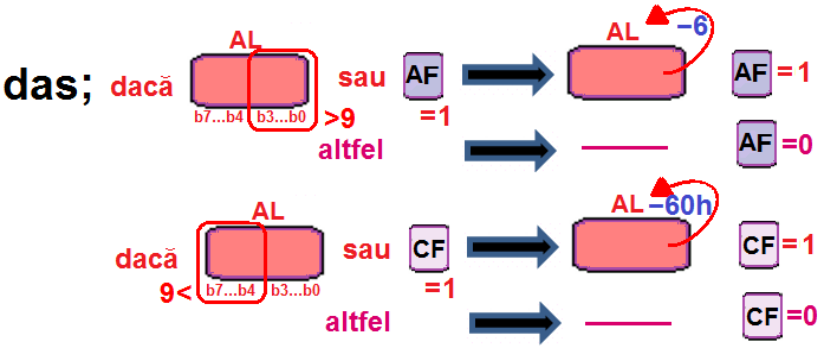


Figura 3-6.2. Ilustrarea modului de operare al instrucţiunii ***DAS***

Observaţii:

- Instrucţiunea DAS nu are operanzi expliciţi, lucrează implicit cu registrul AL;
- Această instrucţiune *setează flagurile* CF și AF dacă la ajustare a apărut un transport ca împrumut zecimal la vreunul din digiţii rezultatului; flagurile SF, ZF, PF *se setează în funcţie de rezultat*; flagul OF *nu este definit*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.3. Instrucţiunea AAA

Instrucţiunea **AAA** (***ASCII Adjust AL after Addition***) ajustează conţinutul registrului AL (conţinut presupus a fi obţinut în urma unei instrucţiuni **ADD** asupra ***a 2 valori BCD despachetate***, deci fiecare scrise cu 1 digit, pe octet) pentru a produce un rezultat corect în format BCD despachetat. Instrucţiunea nu are operanzi expliciţi; se folosește registrul AL atât ca operand sursă (din AL se ia valoarea asupra căreia se aplică acea corecţie ASCII) cât și ca operand destinaţie (tot în AL se depune rezultatul obţinut). Dacă adunarea produce transport (carry), atunci registrul AH este incrementat cu 1 și flagurile CF și AF sunt setate. Dacă nu apare un astfel de transport, CF=AF=0 și AH rămâne neschimbat. Biţii 7...4 ai registrului AL sunt întotdeauna în 0, indiferent de caz.

AAA ; dacă $AL_{3...0}$ > 9 sau AF = 1 → AH=AH+1, AL=(AL+6h)&0Fh și AF=1, CF=1 , altfel → AF=0, CF=0

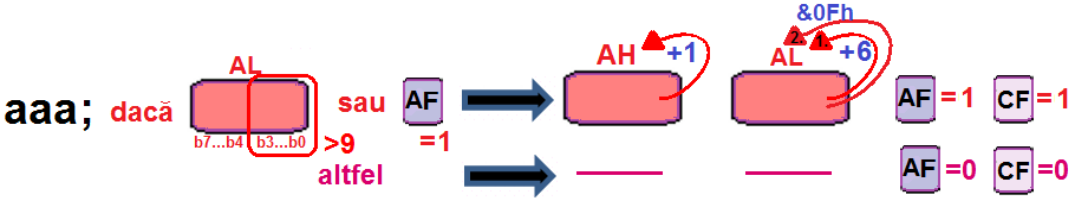


Figura 3-6.3. Ilustrarea modului de operare al instrucţiunii ***AAA***

Observaţii:

- Instrucţiunea AAA nu are operanzi expliciţi, lucrează implicit cu registrul AL, dar poate afecta și registrul AH (dacă apare depășire la corecţie);
- Această instrucţiune *setează flagurile* CF și AF dacă la ajustare a apărut transport zecimal; altfel, sunt puse în 0. Flagurile SF, ZF, PF, OF sunt *nedefinite*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.4. Instrucţiunea AAS

Instrucţiunea **AAS** (***ASCII Adjust AL after Subtraction***) ajustează conţinutul registrului AL (conţinut presupus a fi obţinut în urma unei instrucţiuni **SUB** între ***2 valori BCD despachetate***, deci fiecare scrise cu 1 digit, pe octet) pentru a produce un rezultat corect în format BCD despachetat. Instrucţiunea nu are operanzi expliciţi; se folosește registrul AL atât ca operand sursă (din AL se ia valoarea asupra căreia se aplică acea corecţie ASCII) cât și ca operand destinaţie (tot în AL se depune rezultatul obţinut).

Dacă scăderea a necesitat un împrumut (carry ca borrow), atunci registrul AH este decrementat cu 1 și flagurile CF și AF sunt setate. Dacă nu apare un astfel de transport, CF=AF=0 și AH rămâne neschimbat. Biţii 7...4 ai registrului AL sunt întotdeauna în 0, indiferent de caz.

AAA ; dacă $AL_{3...0}$ > 9 sau AF = 1 → AH=AH+1, AL=(AL+6h)&0Fh și AF=1, CF=1 , altfel → AF=0, CF=0

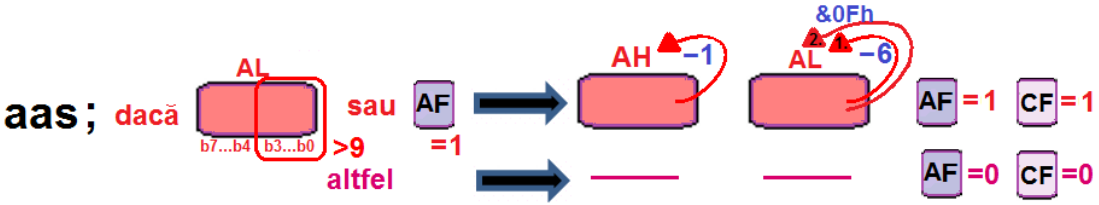


Figura 3-6.4. Ilustrarea modului de operare al instrucţiunii ***AAS***

Observaţii:

- Instrucţiunea AAS nu are operanzi expliciţi, lucrează implicit cu registrul AL, dar poate afecta și registrul AH (dacă apare depășire la corecţie);
- Această instrucţiune *setează flagurile* CF și AF dacă la ajustare a apărut transport (împrumut) zecimal; altfel, aceste flaguri sunt puse în 0. Flagurile SF, ZF, PF, OF sunt *nedefinite*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.5. Instrucţiunea AAM

Instrucţiunea **AAM** (***ASCII Adjust AX after Multiply***) ajustează conţinutul registrului AX (conţinut presupus a fi obţinut în urma unei instrucţiuni **MUL** între ***2 valori BCD despachetate***, deci fiecare scrise cu 1 digit, pe octet) pentru a produce un rezultat corect în format BCD despachetat, scris pe 2 octeţi, în AX.

Instrucţiunea nu are operanzi expliciţi în forma iniţială, dar ulterior s-a adăugat forma generalizată cu operand valoare imediată pe 8 biţi; această valoare imediată se folosește pentru a specifica baza în care se dorește exprimarea rezultatului după corecţie (baza 10 sau 0Ah uzual pentru BCD, 08h pentru octal, etc).

Se folosește registrul AX atât ca operand sursă (din AX se ia valoarea asupra căreia se aplică acea corecţie ASCII) cât și ca operand destinaţie (tot în AX se depune rezultatul obţinut).

AAM [sursă] ***; AH = (AX) / 10, AL = (AX) MOD 10***
AAM {imediat8}

Figura 3-6.5. Ilustrarea modului de operare al instrucţiunii ***AAM***

Observaţii:

- Instrucţiunea AAM în mod uzual nu are operanzi expliciţi, deoarece lucrează implicit cu registrul AX, dar poate avea un operand explicit ca valoare imediată pe 8 biţi, operand care va specifica baza în care se dorește exprimarea rezultatului după aplicarea corecţiei;
- Instrucţiunea *setează flagurile* SF, ZF, PF în funcţie de rezultatul obţinut, dar flagurile OF, CF și AF *sunt nedefinite*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.6. Instrucţiunea AAD

Instrucţiunea **AAD** (***ASCII Adjust AX before Division***) ajustează conţinutul registrului AX (conţinut presupus a fi obţinut înaintea unei instrucţiuni **DIV**, deci cele ***2 valori BCD despachetate*** din AX) astfel încât o operaţie DIV asupra rezultatului corecţiei să conducă la o valoare BCD despachetată corectă.

Această instrucţiune trebuie să preceadă o instrucţiune DIV care va realiza împărţirea rezultatului corecţiei (din AX) la o valoare BCD despachetată. Instrucţiunea nu are operanzi expliciţi în forma iniţială, dar ulterior s-a adăugat forma generalizată cu operand valoare imediată pe 8 biţi; această valoare imediată se folosește pentru a specifica baza în care se dorește exprimarea rezultatului după corecţie (baza 10 sau 0Ah uzual pentru BCD, 08h pentru octal, etc).

Se folosește registrul AX atât ca operand sursă (din AX se ia valoarea asupra căreia se aplică acea corecţie ASCII) cât și ca operand destinaţie (tot în AX se depune rezultatul obţinut).

AAD [sursă] ***; AH = 0 , AL = AH*10 + AL***
AAD {imediat8}

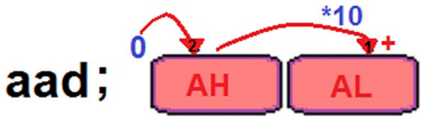


Figura 3-6.6. Ilustrarea modului de operare al instrucţiunii ***AAD***

Observaţii:

- Instrucţiunea AAD în mod uzual nu are operanzi expliciţi, deoarece lucrează implicit cu registrul AX, dar poate avea un operand explicit ca valoare imediată pe 8 biţi, operand care va specifica baza în care se dorește exprimarea rezultatului după aplicarea corecţiei;
- Instrucţiunea *setează flagurile* SF, ZF, PF în funcţie de rezultatul obţinut, dar flagurile OF, CF și AF sunt *nedefinite*;
- Această instrucţiune nu este validă în modul de lucru pe 64 biţi.

3.6.7. Exemple

Exemple de *instrucţiuni ilegale:*

```
daa BL          ; sintaxa eronată, DAA nu are operanzi expliciţi
das BX          ; sintaxa eronată, DAS nu are operanzi expliciţi
aaa ECX         ; sintaxa eronată, AAA nu are operanzi expliciţi
aas DX          ; sintaxa eronată, AAS nu are operanzi expliciţi
aam BL          ; sintaxa eronată, AAM suportă operand explicit doar ca valoare imediată pe 8 biţi
aad EBX         ; sintaxa eronată, AAD suportă operand explicit doar ca valoare imediată pe 8 biţi
```

Exemple de *instrucţiuni legale*:

Exemplul 3-6.1 Adunarea valorilor 35 şi 58, reprezentate prin octeţii 35h şi 58h în BCD împachetat: mov AL, 35h ; AL=35h

```
mov AH, 58h     ; AH=58h
add AL, AH      ; AL=8Dh, care este incorect ca rezultat BCD, Dh>9
daa             ; corecţia 8Dh+6=93h la rezultat: AL=93h (suma BCD a valorilor), iar CF=0, ceea ce
                ; se interpretează ca 35+58=93
```

Exemplul 3-6.2 Adunarea valorilor 88 şi 49, reprezentate prin octeţii 88h şi 49h în BCD împachetat:

```
mov AL, 88h     ; AL=88h
mov AH, 49h     ; AH=49h
add AL, AH      ; AL=D1h, care e incorect ca rezultat BCD, Dh>9; în plus, AF=1
daa             ; deoarece AF=1, se corectează D1h+6h=D7h şi apoi se trece la corecţia D7h+60h=37h la rezultat:
                ; AL=37h iar CF=1 şi se va interpreta ca 1*100 +37 =137 (corect ca BCD), s-a adunat 88+49=137
```

Exemplul 3-6.3 Adunarea valorilor 58 şi 31, reprezentate prin octeţii 58h şi 31h în BCD împachetat: mov AL, 58h ; AL=58h

```
mov AH, 31h     ; AH=31h
add AL, AH      ; AL=89h, care este corect ca rezultat BCD, 58+31=89
daa             ; corecţia nu va avea sens, AL=89h
```

Exemplul 3-6.4 Scăderea valorilor 57 şi 29, reprezentate prin octeţii 57h şi 29h în BCD împachetat:

```
mov AL, 57h          ; AL=57h
mov AH, 29h          ; AH=29h
sub AL, AH           ; AL= 2Eh, care este incorect ca rezultat BCD, Eh>9-> 2Eh-6
das                  ; face corecţia la rezultatul AL=28h (diferenţa BCD), CF=0, 57-29=28
```

Exemplul 3-6.5 Scăderea valorilor 57 şi 76, reprezentate prin octeţii 57h şi 76h în BCD împachetat:

```
mov AL, 57h          ; AL=57h
mov AH, 76h          ; AH=76h
sub AL, AH           ; AL= E1h, care este incorect ca rezultat BCD, Eh>9-> E1h-60h
das                  ; face corecţia la rezultatul AL=81h (diferenţa BCD), CF=1 şi se interpretează ca în
                     ; clasele primare: am împrumutat 10 dinafară (de la cifra de rang următor):
                     ; se interpretează ca 57-76=81 (cu împrumut de la 100)
```

Exemplul 3-6.7 Scăderea valorilor 57 şi 78, reprezentate prin octeţii 57h şi 78h în BCD împachetat:

```
mov AL, 57h          ; AL=57h
mov AH, 78h          ; AH=78h
sub AL, AH           ; AL= DFh, care este incorect ca rezultat BCD, Fh>9-> DFh-6h
das                  ; face corecţia la rezultatul AL=D9h (diferenţa BCD), iar apoi
                     ; corectează Dh>9 cu D9h-60h; rezultă AL=79h şi CF=1 şi se
                     ; interpretează ca în clasele primare: am împrumutat 10 dinafară (de la cifra de rang următor)
                     ; se interpretează ca 57-78=79 (cu împrumut de la 100)
```

Exemplul 3-6.8 Adunarea valorilor 24 şi 49, reprezentate prin 0204h şi 0409h în BCD despachetat:

```
mov AX, 0204h        ; AX=0204h
mov BX, 0409h        ; BX=0409h
add AX, BX           ; AX=060Dh, care este incorect ca rezultat BCD
aaa                  ; corectează 060Dh+F06h=0703h la val. AX=0703h (suma BCD): _2_4 + _4_9 = _7_3
```

Exemplul 3-6.9 Adunarea valorilor 24 şi 94, reprezentate prin 0204h şi 0904h în BCD despachetat:

```
mov AX, 0204h        ; AX=0204h
mov BX, 0904h        ; BX=0904h
add AX, BX           ; AX=0B08h, care este corect dpdv al instrucțiunii aaa (se verifică doar AL) ca rezultat BCD,
                     ; deci nu corectează nimic
aaa                  ; AX=0B08h
```

Exemplul 3-6.10 Scăderea valorilor 44 şi 28, reprezentate prin 0404h şi 0208h în BCD despachetat:

```
mov AX, 0404h        ; AX=0404h
mov BX, 0208h        ; BX=0208h
sub AX, BX           ; AX= 010Ch, care este incorect ca rezultat BCD
aas                  ; face corecţia la rezultatul AX=0106h (diferenţa BCD): 44-28=16
```

Exemplul 3-6.11 Scăderea valorilor 44 şi 82, reprezentate prin 0404h şi 0802h în BCD despachetat:

```
mov AX, 0404h        ; AX=0404h
mov BX, 0802h        ; BX=0802h
sub AX, BX           ; AX= FC02h, care este corect dpdv al instrucțiunii aaa (se verifică doar AL) ca rezultat BCD,
aas                  ; deci nu corectează nimic AX= FC02h, deci _ 4 -_2 = _2
```

Exemplul 3-6.12

```
mov AL,6             ; AL=06h
mov BL,7             ; BL=07h
mul BL               ; AX=AL*BL=42 => AX = 2Ah
aam                  ; AX = 0402h, 6*7=42
```

Exemplul 3-6.13

```
mov AX, 0208h        ; AX=0208h în BCD, adică 20+8=28
aad                  ; după corecţie AL=1Ch, iar AH=0h;
                     ; deci AX=001Ch=28, ca şi cum pe 28 l-ar pregăti pentru o următoare operaţie în binar (o împărţire)
```

Capitolul 4. Instrucţiuni pe biţi

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, BT, BT[S/R/C], BS[F/R], SETcc**
3. **de deplasare (shift):** **SHL/SAL, SHR, SAR, SHLD, SHRD**
4. **de rotire (rotate):** **ROL, ROR, RCL, RCR**

5. **alte instrucţiuni pe 32 biţi:** **ADCX, ADOX, POPCNT, LZCNT, TZCNT, ANDN, BEXTR, BLSI, BLSR, BLSMSK, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX**

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 numește 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.

4.1. Instrucţiuni 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** și **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 și 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)***

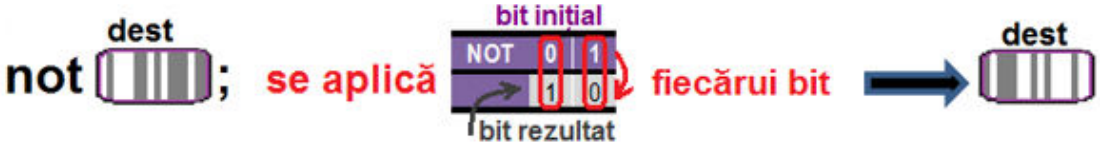


Figura 5-1.1. Ilustrarea modului de operare al instrucţiunii ***NOT***

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$_{8,16,32}$}

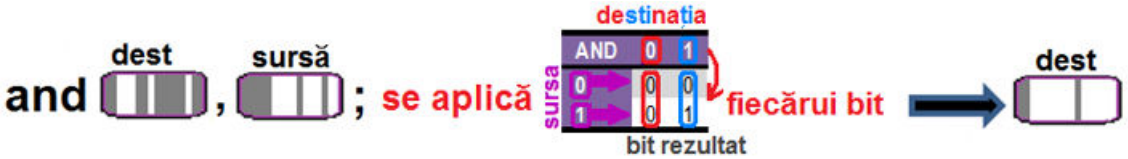


Figura 4-1.2. Ilustrarea modului de operare al instrucţiunii ***AND***

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 biţi 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.

4.1.5. Exemple

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
```

4.2. Instrucţiuni de testare pe biţi

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

Ulterior, de la **80386↑** au fost adăugate mai multe instrucţiuni de testare a biţilor, inclusiv cu posibilitate de testare și modificare a unui singur bit: **BT, BTS, BTR, BTC, BSF, BSR, SETcc;** în plus, dimensiunea operanzilor a fost extinsă la 32 biţi.

Instrucţiunea **TEST** şi instrucţiunea **SETcc** pot fi folosite pentru a modifica sensul de curgere al programului, în funcţie de starea flagurilor din [-/E/R] FLAGS. Instrucţiunea TEST este urmată în general de o instrucţiune de salt condiţionat (Jcc).

Instrucţiunea **BT** ***(Bit Test)*** nu afectează destinaţia; aceasta doar testează şi setează/ resetează apoi CF după cum e cazul.

Instrucţiunile **BTS** ***(Bit Test and Set)***, **BTR** ***(Bit Test and Reset)*** şi **BTC** ***(Bit Test and Complement)*** sunt asemănătoare cu **BT**, dar mai realizează în plus, după setarea CF: setarea, resetarea, respectiv complementarea valorii bitului testat din operandul destinaţie, deci aceste instrucţiuni modifică destinaţia.

La aceste instrucţiuni, operanzii nu trebuie să aibă dimensiuni egale, întrucât au semnificaţie diferită.

Unele dintre instrucţiunile tratate aici au fost îmbunătăţite pe procesoarele mai noi, prin implementarea instrucţiunilor din categoria „**alte instrucţiuni pe 32 biţi”.**

Dintre acestea, cele mai apropiate de instrucţiunile din această categorie sunt: **LZCNT, TZCNT, BLSI, BLSR, BEXTR, BLSMSK, BZHI.**

4.2.1. Instrucţiunea TEST

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$_{8,16,32}$}

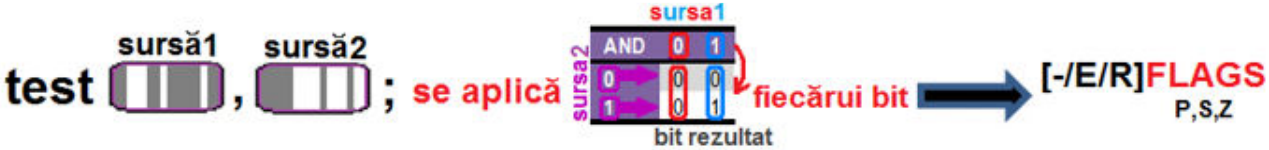


Figura 4-2.1. Ilustrarea modului de operare al instrucţiunii ***TEST***

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.

4.2.2. Instrucţiunea BT

Instrucţiunea **BT** ***(Bit Test)*** testează un singur bit din operandul destinaţie: cel specificat de operandul index. Acest bit este stocat în CF.

BT destinaţie, index ***; dacă (destinaţie [index]) = 1 → CF=1***
BT { $reg_{16,32,64}$| $mem_{16,32,64}$}, { $reg_{16,32,64}$| $imed_8$} ***altfel → CF=0***

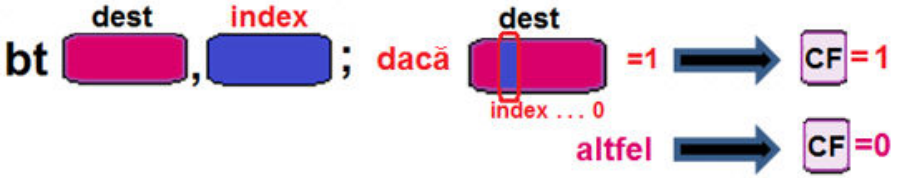


Figura 4-2.2. Ilustrarea modului de operare al instrucţiunii ***BT***

Observaţii:

- Instrucţiunea BT determină încărcarea în CF a bitului supus testării, flagul ZF *nefiind afectat* de această operaţie;
  - flagurile OF, SF, AF, PF sunt *nedefinite*;
- Operanzii nu trebuie să aibă dimensiuni egale (au semnificaţie diferită);
- Instrucţiunea BT nu afectează destinaţia, ci doar flagul CF.

4.2.3. Instrucţiunea BTS

Instrucţiunea **BTS** ***(Bit Test and Set)*** testează un singur bit din operandul destinaţie: cel specificat de operandul index. Acest bit este stocat în CF și apoi bitul respectiv din destinaţie este setat (pus în 1).

BTS destinaţie, index ***; dacă (destinaţie [index]) = 1 → CF=1***
BTS { reg$_{16,32,64}$| mem$_{16,32,64}$}, { reg$_{16,32,64}$| imed$_8$} ***altfel → (destinaţie [index]) = 1 și CF=0***

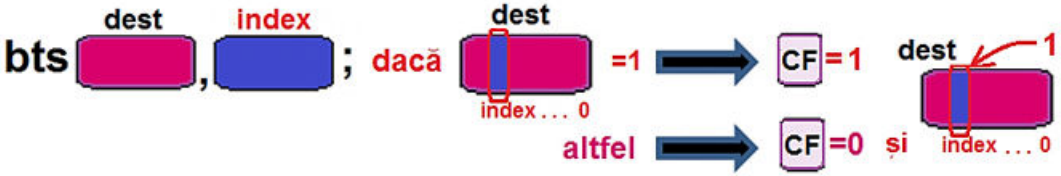


Figura 4-2.3. Ilustrarea modului de operare al instrucţiunii ***BTS***

Observaţii:

- Instrucţiunea BTS determină încărcarea în CF a bitului supus testării, flagul ZF *nefiind afectat* de această operaţie;
  - OF, SF, AF, PF sunt *nedefinite*;
- Operanzii nu trebuie să aibă dimensiuni egale (au semnificaţie diferită);
- Instrucţiunea BTS poate afecta destinaţia, deoarece bitul respectiv se va pune pe 1 (destinaţie[index]=1);
  - valoarea care se va regăsi în flagul CF va fi cea dinainte de această setare.

4.2.4. Instrucţiunea BTR

Instrucţiunea **BTR** ***(Bit Test and Reset)*** testează un singur bit din operandul destinaţie: cel specificat de operandul index. Acest bit este stocat în CF și apoi bitul respectiv din destinaţie este resetat (pus în 0).

BTR destinaţie, index ***; dacă (destinaţie [index]) = 0 → CF=0***
BTR { reg$_{16,32,64}$| mem$_{16,32,64}$}, { reg$_{16,32,64}$| imed$_8$} ***altfel → (destinaţie [index]) = 0 și CF=1***

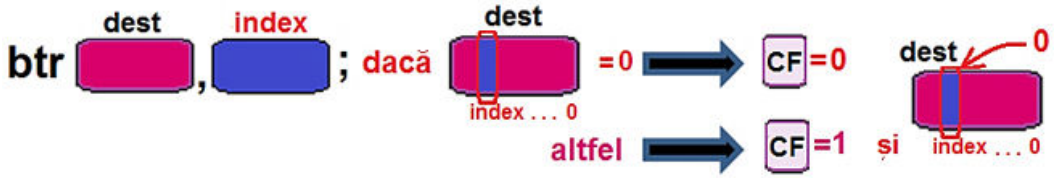


Figura 4-2.4. Ilustrarea modului de operare al instrucţiunii ***BTR***

Observaţii:

- Instrucţiunea BTR determină încărcarea în CF a bitului supus testării, flagul ZF *nefiind afectat* de această operaţie;
  - OF, SF, AF, PF sunt *nedefinite*;
- Operanzii nu trebuie să aibă dimensiuni egale (au semnificaţie diferită);
- Instrucţiunea BTR poate afecta destinaţia, deoarece bitul respectiv se va pune pe 0 (destinaţie[index]=0);
  - valoarea care se va regăsi în flagul CF va fi cea dinainte de această resetare.

4.2.5. Instrucţiunea BTC

Instrucţiunea **BTC** ***(Bit Test and Complement)*** testează un singur bit din operandul destinaţie: cel specificat de operandul index. Acest bit este stocat în CF și apoi bitul respectiv din destinaţie este setat (pus în 1).

BTS destinaţie, index ***; dacă (destinaţie [index]) = 1 → CF=1***
BTS { $reg_{16,32,64}$| $mem_{16,32,64}$}, { $reg_{16,32,64}$| $imed_8$} ***altfel → (destinaţie [index]) = 1 și CF=0***

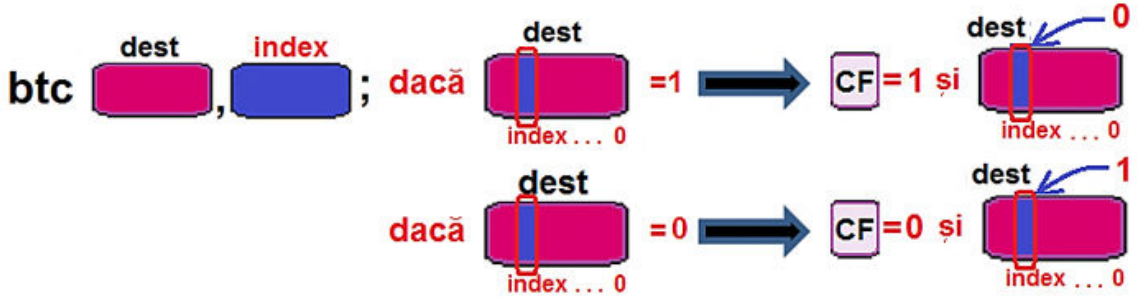


Figura 4-2.5. Ilustrarea modului de operare al instrucţiunii ***BTC***

Observaţii:

- Instrucţiunea BTC determină încărcarea în CF a bitului supus testării, flagul ZF *nefiind afectat* de această operaţie;
  - OF, SF, AF, PF sunt *nedefinite*;
- Operanzii nu trebuie să aibă dimensiuni egale (au semnificaţie diferită);
- Instrucţiunea BTC afectează destinaţia, deoarece bitul respectiv se va complementa; se realizează de fapt operaţia: (destinaţie[index]) = ***not*** (destinaţie[index]));
  - valoarea care se va regăsi în flagul CF va fi cea dinainte de această complementare.

4.2.6. Instrucţiunea BSF

Instrucţiunea **BSF** ***(Bit Scan Forward)*** parcurge pe biţi operandul sursă înspre MSb ← de la bitul LSb (bit0) şi scrie în *operandul destinaţie* poziţia primului bit nenul întâlnit. Dacă operandul sursă este nul (și deci nu se găsește nici un bit de 1), atunci ZF=1, iar destinaţia este nedefinită.

BSF destinaţie, sursă ; caută în sens "←" (sursă [primul indice]) = 1 → (destinaţie) = indice și ZF=0
BSF { reg16,32,64}, { reg16,32,64| mem16,32,64} ***altfel, dacă nu găsește → (destinaţie) = nedefinit și ZF=1***

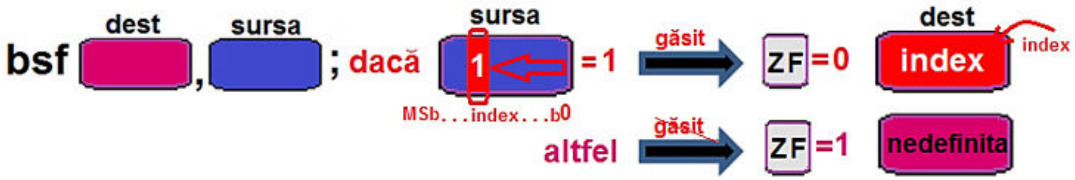


Figura 4-2.6. Ilustrarea modului de operare al instrucţiunii ***BSF***

Observaţii:

- Instrucţiunea BSF determină încărcarea în operandul destinaţie a poziţiei bitului cel mai puţin semnificativ găsit de valoare 1 în sursă. Dacă s-a găsit un astfel de bit, ZF=0; altfel, ZF=1 și destinaţia va fi nedefinită;
  - flagurile CF, OF, SF, AF și PF sunt *nedefinite*;
- Operanzii trebuie să aibă dimensiuni egale;
- Instrucţiunea BSF nu modifică sursa.

4.2.7. Instrucţiunea BSR

Instrucţiunea **BSR** ***(Bit Scan Reverse)*** parcurge pe biţi operandul sursă dinspre MSb → la bitul LSb (bit0) şi scrie în *operandul destinaţie* poziţia primului bit nenul întâlnit. Dacă operandul sursă este nul (și deci nu se găsește nici un bit de 1), atunci ZF=1, iar destinaţia este nedefinită.

BSR destinaţie, sursă ; caută în sens "→" (sursă [primul indice]) = 1 → (destinaţie) = indice și ZF=0
BSR { $reg_{16,32,64}$}, { $reg_{16,32,64}$| $mem_{16,32,64}$} ***altfel, dacă nu găsește → (destinaţie) = nedefinit și ZF=1***

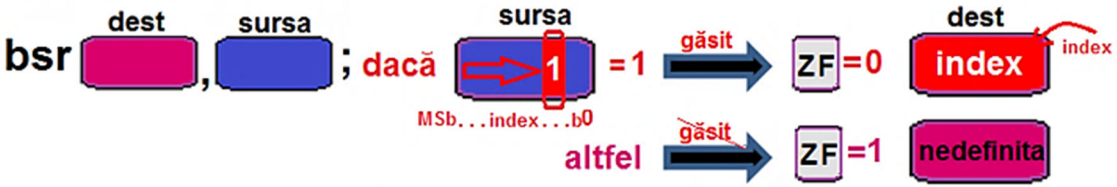


Figura 4-2.7. Ilustrarea modului de operare al instrucţiunii ***BSR***

Observaţii:

- Instrucţiunea BSR determină încărcarea în operandul destinaţie a poziţiei bitului cel mai semnificativ găsit de valoare 1 în sursă. Dacă s-a găsit un astfel de bit, ZF=0; altfel, ZF=1 și destinaţia va fi nedefinită;
  - flagurile CF, OF, SF, AF și PF sunt *nedefinite*;
- Operanzii trebuie să aibă dimensiuni egale;
- Instrucţiunea BSR nu modifică sursa.

4.2.8. Instrucţiunea SETcc

Instrucţiunea **SETcc** (***SET (Byte) on Condition***) verifică o anumită condiţie în funcţie de sufixul „**cc**” (asupra flagurilor sau numerelor cu semn/ fără semn) şi depune în octetul precizat ca operand destinaţie:

- valoarea 1 (dacă se verifică acea condiţie) sau
- valoarea 0 (dacă nu se verifică acea condiţie).

SETcc destinaţie ***; dacă cc = adevărat → (destinaţie) = 1***
SETcc { reg$_8$| mem$_8$} ***altfel → (destinaţie) = 0***

Figura 4-2.8. Ilustrarea modului de operare al instrucţiunii ***SETcc***

La evaluarea condiţiei din cadrul instrucţiunii SETcc, se folosește noţiunea de „mai mic” (**less**) sau „mai mare” (**greater**) pentru compararea numerelor ***cu semn (signed)***, iar termenii „deasupra, peste” (**above**) sau „dedesubt, sub” (**below**) se folosesc pentru numere ***fără semn (unsigned)***, exact așa cum s-a procedat în secţiunea 2.2.1. la instrucţiunea CMOVcc. Condiţia pentru fiecare mnemonică a fost prezentată detaliat acolo și nu se va mai relua şi aici.

Observaţii:

- Instrucţiunea SETcc se foloseşte în general în urma unei operaţii aritmetice sau a unei comparări;
- Instrucţiunea SETcc *nu afectează flagurile*;
- Operandul trebuie să aibă dimensiunea unui octet, indiferent că este registru sau operand din memorie;
  - nu se pot folosi operanzi de tip imediat;

4.2.9. Exemple

Exemple de *instrucţiuni ilegale:*

```
test BX, AL        ; dimensiunea operanzilor nu potriveşte
test AL, 32768     ; 32768 nu încape pe 8 biţi
bt AX, a           ; al II-lea operand nu poate fi din memorie
(similar la btr, bts, btc)
bsf a, AX          ; primul operand nu poate fi decât registru
bsf AL, BL         ; instrucţiunea e suportată doar cu operanzi de 16, 32 sau 64 biţi
(similar la bsr)
setl AX            ; instrucţiunea foloseşte operand de tip byte, nu word (din registru sau din memorie)
```

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
```

Exemplul 4-2.2

```
mov EAX, 23         ; se verifică bitul 23: dacă e setat, atunci tehnologia MMX e suportată
mov EBX, 0387F9FFh  ; destinaţia EBX=0387F9FFh
bt EBX, EAX         ; EBX=0000.0011.1000.0111.1111.1001.1111.1111b cu  b23=1, deci CF=1
jc etSuportaMMX     ; salt la o etichetă care tratează cazul în care procesorul suportă setul de instrucțiuni MMX
```

Exemplul 4-2.3 Se doreşte urmărirea efectului execuţiei instrucţiunilor BTS, BTR şi BTC într-o secvenţă de program:

```
mov EAX, 23          ; se verifică dacă bitul 23 este setat,
mov EBX, 0387F9FFh   ; registrul EBX joacă rol de destinaţie, EBX = 0000.0011.1000.0111.1111.1001.1111.1111b
bts EBX, EAX         ; CF=1, b23 rămâne 1->EBX=0387F9FFh (neschimbat)
btr EBX, EAX         ; CF=1 (CF păstrează valoarea iniţială a bitului), b23 devine 0 după BTR, deci EBX=0307F9FFh
btc EBX, EAX         ; CF=0, b23 devine 1 prin complementare (după BTC), deci EBX=0387F9FFh
```

Exemplul 4-2.4

```
mov AX, 1234h        ; AX=1234h=0001 0010 0011 0100b
bsf CX, AX           ; CX=2, deoarece găseşte dinspre dreapta spre stânga bitul b2 setat
bsr DX, AX           ; DX=12=0Ch, deoarece găseşte dinspre stânga spre dreapta bitul b12 setat
```

Exemplul 4-2.5

```
mov AX, 2345h        ; AX =  2345h  = 9029
mov BX, 0EDCBh       ; BX = EDCBh = 60875
add AX, BX           ; AX=AX+BX= 1111h şi CF=1
setnc CL             ; se testează dacă CF=0, deci CL va fi 0
```

Exemplul 4-2.6

```
mov EAX, 2           ; EAX=2
mov EBX, 3           ; EBX=3
cmp EAX, EBX         ; se testează 2 faţă de 3 şi rezultă EAX-EBX=2-3=-1<0
setle CH             ; rezultatul este mai mic decât 0, deci CH=1
```

Exemplul 4-2.7

```
mov AX, -2           ; AX= -2 = FFFEh
mov BX, 2            ; BX=  2 = 0002h
cmp AX, BX           ; se testează -2 faţă de 2, de fapt numerele nu sunt interpretate ci sunt văzute ca FFFEh şi 0002h
setbe CH             ; dacă se foloseşte below/above, nr vor fi văzute ca fiind fără semn => 65.534 nu este sub 2=> CH=0
setle CH             ; dacă se foloseşte less/greater, nr vor fi văzute ca fiind cu semn => -2 este mai mic decât 2=> CH=1
```

4.3. Instrucţiuni 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 foloseşte mnemonica **SHL** (***shift logic to left***) pentru a desemna o astfel de operaţie. Instrucţiunea **SAL** (***shift aritmetic to left***) va avea efect identic cu cel obţinut prin instrucţiunea SHL, deoarece dinspre bitul 0 se va insera tot 0 (nu ar avea sens să se insereze MSb).
Similar, se foloseşte mnemonica **SHR** (***shift logic to right***) pentru a desemna o operaţie de ***deplasare logică*** spre dreapta.
Operaţia de ***deplasare aritmetică*** spre dreapta se poate obţine folosind mnemonica **SAR** (***shift aritmetic to right***).

Operaţia de deplasare spre dreapta rotunjeşte 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 înmulţire cu 2^n a numărului, înseamnă ***o deplasare spre stânga cu n biţi***, iar

o împărţire cu 2^n a numărului, înseamnă ***o deplasare spre dreapta cu n biţi***,

și invers, o deplasare spre stânga cu n poziţii e echivalent cu o înmulţire cu 2^n, iar

o deplasare spre dreapta cu n poziţii e echivalent cu o împărţire cu 2^n.

Există situaţii când sunt necesare operaţii de înmulţire/ împărţire (obţinute prin deplasare) a numerelor ***fără semn***, iar atunci deplasarea trebuie realizată prin operaţii de ***deplasare logică***; în situaţiile când se dorește deplasarea numerelor ***cu semn***, se vor folosi operaţii de ***deplasare aritmetică***, întrucât acestea nu vor modifica semnul numerelor, ci doar valoarea lor.

În plus, la deplasarea spre stânga trebuie ţinut cont de semnul numărului și de posibilele alterări ale acestuia prin operaţia de deplasare (pentru a nu obţine 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, instrucţiunea de deplasare sau **shiftare aritmetică** se folosește pentru ***numere cu semn***,
iar instrucţiunea 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***

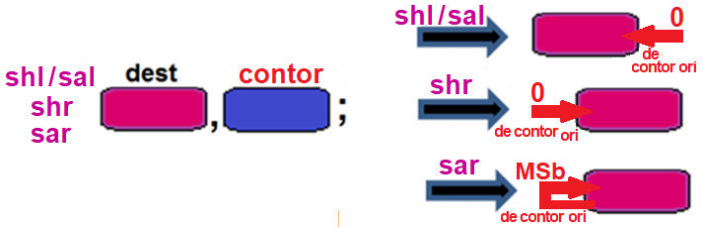


Figura 4-3.2. Ilustrarea modului de operare al instrucţiunilor de deplasare

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$_8$ }

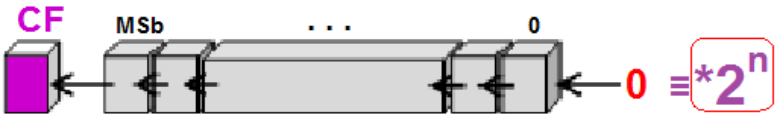


Figura 4-3.2. Ilustrarea modului de operare al instrucţiunilor ***SHL*** și ***SAL***

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$_8$ }

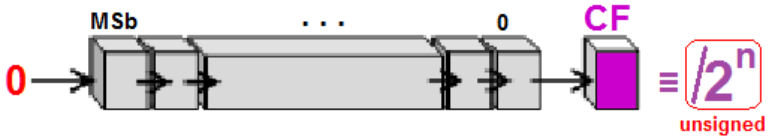


Figura 4-3.3. Ilustrarea modului de operare al instrucţiunii ***SHR***

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$_8$ }

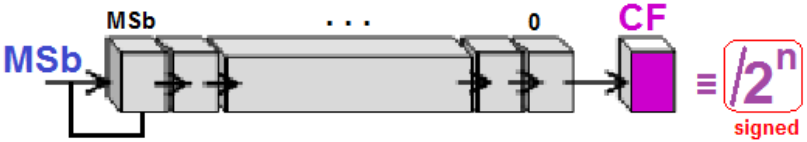


Figura 4-3.4. Ilustrarea modului de operare al instrucţiunii ***SAR***

La instrucţiunile **SHL/SAL, SHR, SAR**, operanzii nu trebuie să aibă dimensiuni egale, întrucât au semnificaţii diferite.

Observaţii (la deplasarea spre stânga prin **SHL/ SAL***):*

- Instrucţiunile SAL / SHL afectează flagurile CF, OF, SF, ZF, PF. Flagul CF conţine valoarea ultimului bit deplasat spre stânga, înafara operandului destinaţie. Flagul OF este afectat doar pentru o singură deplasare: va avea valoarea OF=0 dacă flagul CF este identic cu bitul MSb (adică primii 2 biţi c.m.s. ai operandului erau identici), sau OF=1 în caz contrar. În caz că sunt mai multe astfel de deplasări, OF va fi nedefinit. SF, ZF și PF sunt afectate în funcţie de rezultatul obtinut, după regulile uzuale. Flagul AF este nedefinit;
- Aceste instrucţiuni sunt echivalente cu o înmulţire cu 2, de contor ori;

Observaţii (la deplasarea spre dreapta prin **SHR***):*

- Instrucţiunea SHR afectează flagurile CF, OF, SF, ZF, PF. Flagul CF conţine valoarea ultimului bit deplasat spre dreapta, înafara operandului destinaţie. Flagul OF este setat cu valoarea bitului MSb al valorii iniţiale a destinaţiei, iar SF, ZF și PF sunt afectate în funcţie de rezultatul obtinut, după regulile uzuale. Flagul AF este nedefinit;
- Această instrucţiune este echivalentă cu o împărţire cu 2, de contor ori, dacă se folosește pentru numere *fără semn*; produce același cât ca și DIV;

Observaţii (la deplasarea spre dreapta prin **SAR***):*

- Instrucţiunea SAR afectează flagurile CF, OF, SF, ZF, PF. Flagul CF conţine valoarea ultimului bit deplasat spre dreapta, înafara operandului destinaţie. Flagul OF este afectat doar pentru o singură deplasare: va fi resetat la deplasarea unui singur bit. Flagurile SF, ZF și PF sunt afectate în funcţie de rezultatul obtinut, după regulile uzuale. Flagul AF este nedefinit;
- Aceste instrucţiuni sunt echivalente cu o împărţire cu 2, de contor ori, dacă se folosește pentru numere *cu semn*;
  - Deplasarea spre dreapta este asemănătoare unei împărţiri, însă SAR nu produce același rezultat ca IDIV: la IDIV, rotunjirea câtului (valoarea din Acc) are loc înspre 0, în timp ce valoarea care rămâne în destinaţie la SAR (câtul) e rotunjit înspre -∞. Diferenţa se constată în special la numerele negative.
  - Exemplu: -9: 4 => cât=-2, rest =-1 cu IDIV, iar cu SAR: -9 deplasat spre dreapta cu 2 poziţii, deci -9:4 => -3 (cât), iar restul ar trebui să fie +3 !, dar SAR nu stochează decât MSb al restului în CF (nu se va vedea 3, ci 1).

4.3.4. Instrucţiunile SHLD / SHRD

De la **80386↑** au fost adăugate instrucţiunile **SHLD** ***(Shift Left Double)*** şi **SHRD** ***(Shift Right Double)***; acestea realizează deplasări spre stânga, respectiv spre dreapta, a unor valori în dublă precizie. Deplasarea biţilor destinaţiei (primul operand) se realizează prin umplerea cu biţi din cadrul unui operand sursă (al doilea operand). Se realizează atâtea deplasări cât specifică operandul contor (al treilea operand).

La **SHLD**, al doilea operand (operandul sursă) furnizează biţii care se vor insera dinspre dreapta, începând de la bitul 0 al destinaţiei, în timp ce la **SHRD**, al doilea operand (operandul sursă) furnizează biţi care se vor insera dinspre stânga, începând de la bitul MSb al destinaţiei.

SHLD/SHRD destinaţie, sursă, contor *; deplasează biţii din **destinaţie** cu **contor poziţii** spre **stânga/ dreapta,***
; inserând biţi din copia temporară a sursei

*mnemonica {reg$_{16,32,64}$|mem$_{16,32,64}$}, {reg$_{16,32,64}$-5biti} , {CL| imed$_8$}, mnemonica={**SHLD, SHRD**}*

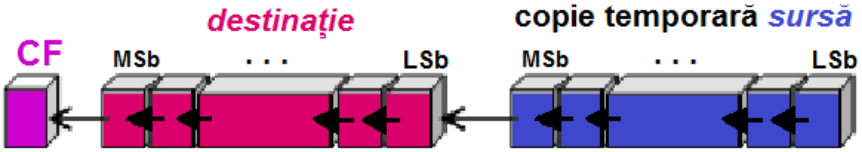


Figura 4-3.5. Ilustrarea modului de operare al instrucţiunii ***SHLD***

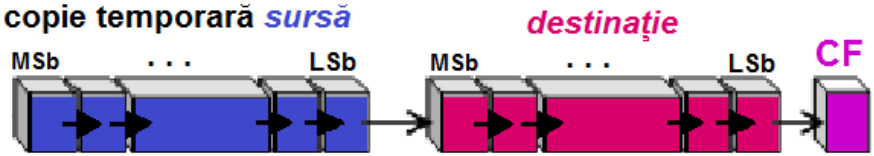


Figura 4-3.6. Ilustrarea modului de operare al instrucţiunii ***SHRD***

Observaţii:

- La instrucţiunile **SHLD** şi **SHRD,** dimensiunea primilor 2 operanzi trebuie să fie aceeaşi, dar nu e obligatoriu să coincidă şi cu dimensiunea celui de-al III-lea operand (au semnificaţii diferite).

- Instrucţiunea **SHLD** afectează flagurile CF, OF, SF, ZF, PF. Flagul CF este identic cu bitul MSb care a fost eliminat anterior (ultimul bit deplasat spre stânga, înafara operandului destinaţie).

- Instrucţiunea **SHRD** afectează flagurile CF, OF, SF, ZF, PF. Flagul CF este identic cu bitul bo care a fost eliminat anterior (ultimul bit deplasat spre dreapta, înafara operandului destinaţie).

- Flagul OF este afectat doar pentru o singură deplasare: va avea valoarea OF=1 dacă a apărut o schimbare de semn (adică primii 2 biţi c.m.s. ai operandului au fost diferiţi).
  - În caz că sunt mai multe astfel de deplasări, OF va fi nedefinit. SF, ZF și PF sunt afectate în funcţie de rezultatul obtinut, după regulile uzuale.
  - Flagul AF este nedefinit;
- 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);

Dacă valoarea operandului count este mai mare decât dimensiunea primilor 2 operanzi, rezultatul este nedefinit.

4.3.5. Exemple

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 biți
```
(similar, ca la SHL se procedează şi pentru SAL, SHR, SAR)
```
shld AL, BL, 10       ; nu sunt suportați operanzi pe 8 biți
shld AX, [BX], 10     ; nu se suportă operandul2 din memorie
shld AX, BL           ; sintaxă eronată, trebuie specificați 3 operanzi
shld AX, BL, 2        ; dimensiunea primilor 2 operanzi trebuie să fie identică
shld AX, BX, CX       ; al III-lea operand poate fi doar CL sau imediat
```
(similar, ca la SHLD se procedează şi pentru SHRD)

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 4-3.4 Fie AX=1234h, BX=5678h:
```
shld AX, BX, 8        ; conţinutul reg. AX este deplasat spre stânga cu 8 poziţii şi umplut de la dreapta spre stânga cu
                      ; cei mai semnificativi 8 biţi ai registrului BX - 80386↑; astfel, AX va fi AX=3456h.
```

Exemplul 4-3.5 Fie EAX=12345678h, EBX=9ABCDEF0h:

```
shrd EAX, EBX, 20        ; conţinutul reg.EAX e deplasat spre dreapta cu 20 poziţii şi umplut de la stânga spre dreapta cu
                         ; cei mai puţin semnificativi 20 biţi ai registrului EBX - 80386↑; astfel, EAX va fi: EAX=CDEF0123h
```

Exemplul 4-3.6 Fie EAX=23456789h:

```
shl EAX, 31      ; conţinutul lui EAX e deplasat spre stânga cu 31 poziţii şi umplut de la stânga spre dreapta cu 0,
                 ; EAX=80000000h
```

Exemplul 4-3.7 Fie EAX=23456789h:

```
shl EAX, 32      ; teoretic, conţinutul lui EAX e deplasat spre stânga cu 32 poziţii şi umplut de la stânga spre dreapta cu 0,
                 ; practic, EAX rămâne neschimbat întrucât există o mască de 5 biţi care se aplică peste contor, deci biţii de
                 ; pe poziţiile 31...6 ai contorului sunt ignoraţi: EAX=23456789h
```

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
shl AX, 5        ; AX=8960/2^5=280
```

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

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

4.4. Instrucţiuni de rotaţie

Aceste operaţii de rotaţie la nivel de șiruri 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 șirurilor 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.

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$_8$ }

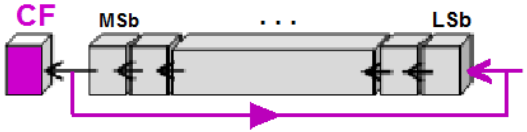


Figura 4-4.3. Ilustrarea modului de operare al instrucţiunii ***ROL***

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$_8$ }

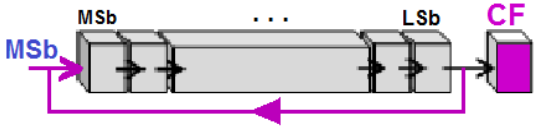


Figura 4-4.4. Ilustrarea modului de operare al instrucţiunii ***ROR***

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)***

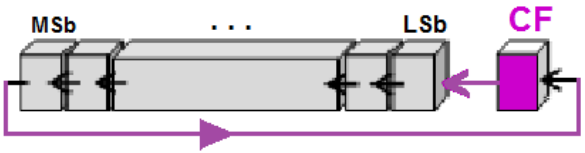


Figura 4-4.5. Ilustrarea modului de operare al instrucţiunii ***RCL***

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)***

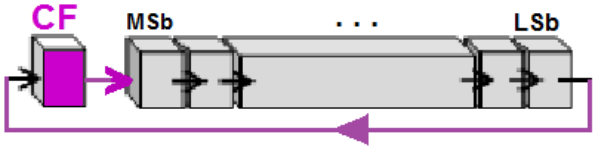


Figura 4-4.6. Ilustrarea modului de operare al instrucţiunii ***RCR***

4.4.3. Exemple

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 biți
```
(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, CF=1
rcr AX,1            ; AX=D555h =1101 0101 0101 0101b, CF=0
```

Exemplul 4-4.7 Fie EAX=23456789h:

```
rol EAX, 8          ; conţinutul lui EAX e rotit spre stânga cu 8 poziţii; astfel, EAX=45678923h
```

Exemplul 4-4.8 Fie EAX=23456789h:

```
ror EAX, 8          ; conţinutul lui EAX e rotit spre dreapta cu 8 poziţii; astfel, EAX=89234567h
```

Exemplul 4-4.9 Fie EAX=23456789h:

```
clc                 ; CF=0
rcl EAX, 8          ; conţinutul lui EAX e rotit spre stânga cu 8 poziţii, cu CF; astfel, EAX=45678911h
```

Exemplul 4-4.10 Fie EAX=23456789h:

```
stc                 ; CF=1
rcl EAX, 8          ; conţinutul lui EAX e rotit spre stânga cu 8 poziţii, cu CF; astfel, EAX=45678991h
```

Exemplul 4-4.11 Fie EAX=23456789h:

```
clc                 ; CF=0
rcr EAX, 8          ; conţinutul lui EAX e rotit spre dreapta cu 8 poziţii, cu CF; astfel, EAX=12234567h
```

Exemplul 4-4.12 Fie EAX=23456789h:

```
stc                 ; CF=1
rcr EAX, 8          ; conţinutul lui EAX e rotit spre dreapta cu 8 poziţii, cu CF; astfel, EAX=13234567h
```

4.5. Alte instrucţiuni cu GPR, introduse de la procesoare pe 32 biţi

Deşi în ultimii ani au apărut foarte multe seturi de instrucţiuni SIMD, procesoarele bazate pe microarhitecturi recente, precum Broadwell şi Haswell includ şi câteva instrucţiuni noi cu regiştrii GPR (General-Purpose Register), şi deci care nu folosesc setul de regiştri MMX, SSE, FPU, etc.

Aceste instrucţiuni se folosesc în general pentru a realiza:

- ***operaţii de înmulţire de numere întregi fără semn***,
- ***deplasări*** (în variantă care ***nu afectează flagurile***, fiind astfel mai rapide),
- ***manipulări de biţi*** sau
- ***extensii pentru adunare cu carry***.

De foarte multe ori însă se confundă (deoarece au apărut deodată) faptul că în setul SSE4 ar exista încă 2 instrucţiuni excepţie (care operează cu regiştri de numere întregi şi nu cu cei SSE, dar care de fapt nu sunt instrucţiuni SIMD): instrucţiunea POPCNT şi instrucţiunea LZCNT.

Intel a implementat instrucţiunea POPCNT începând cu microarhitectura Nehalem, iar pe cea LZCNT începând cu Haswell. AMD a denumit această pereche de instrucţiuni ca fiind setul *Advanced Bit Manipulation (ABM)*.

Aceste instrucţiuni se numesc instrucţiuni GPR deoarece folosesc regiştri de uz general în execuţie (nu folosesc regiştri SIMD, precum MMX, XMM, FPU, etc); astfel, despre aceste instrucţiuni se spune că sunt *non-SIMD*, ele fiind organizate în următoarele seturi: ***ADX***, ***BMI1*** şi ***BMI2***.

Intel ADX (*Multi-Precision Add-Carry Instruction Extensions*) a fost implementat pentru prima dată în microarhitectura Broadwell şi conţine doar 2 instrucţiuni: **ADCX** şi **ADOX**. Aceste instrucţiuni nu sunt altceva decât variante mai eficiente ale instrucţiunii ADC; diferenţa majoră constă în faptul că acestea afectează un singur flag: ori Carry (la ADCX), ori Overflow (la ADOX), creând astfel posibilitatea de a executa cele 2 instrucţiuni în paralel. Reamintesc aici că instrucţiunea ADC putea realiza operaţia la nivel de numere cu semn, ceea ce putea să ducă la setarea ambelor flaguri: şi Carry şi Overflow. Faptul că setul de instrucţiuni **ADX** este suportat de un procesor se poate verifica prin execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare şi verificarea stării bitului 19 din registrul EDX, care ar trebui să fie setat.

Restul instrucţiunilor abordate în această secţiune fac parte din setul de instrucţiuni ***BMI*** *(Bit Manipulation Instructions Set)* (cu excepţia **POPCNT** şi **LZCNT**: POPCNT face parte din setul SSE4.2, iar LZCNT este parte a setului BMI1 [2]. Aceste instrucţiuni sunt în general dedicate operaţiilor de manipulare a biţilor:

- POPCNT, LZCNT;
- **BMI1**: ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT;
- **BMI2**: BZHI, MULX, PDEP, PEXT, RORX, SARX, SHRH, SHLX.

Instrucţiunile ce conţin litera X ca sufix sunt de tipul „flagless", ceea ce înseamnă: „fără setarea flagurilor după realizarea operaţiei" şi din acest motiv sunt mai rapide; pentru înmulţire: MULX, iar pentru deplasare/ rotire: SARX, SHRX, SHLX, RORX.

Cele 2 seturi BMI1 şi BMI2 au fost introduse de Intel pentru prima dată în microarhitectura Haswell;

AMD a folosit alte 2 seturi similare, denumite ABM (Advanced Bit Manipulation) şi TBM (Trailing Bit Manipulation).

Faptul că setul de instrucţiuni **BMI1** este suportat de un procesor se poate verifica prin execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare şi verificarea stării bitului 3 din registrul EBX, care ar trebui să fie setat;

similar, setul **BMI2** este validat de bitul 8 din acelaşi registru.

Suportul pt execuţia instrucţiunii LZCNT[6] poate fi verificat prin consultarea flagului „*abm*”: EAX=80000001h ->cpuid-> ECX|b5=1. Instrucţiunea POPCNT are un flag separat: EAX=1 ->cpuid-> ECX|b23=1.

Tabel 4.5-1. Setul de instrucţiuni BMI1 şi BMI2 [1]

Set BMI1 (EAX=7-> EBX|b3=1)			Set BMI2 (EAX=7-> EBX|b8=1)	
Instrucţ.	**Descriere**		**Instrucţ.**	**Descriere**
POPCNT*	Population count *Numără biţii de 1*		**BZHI**	Zero high bits starting with specified bit position *Zerorizează biţii superiori începând de la o anumită poziţie*
LZCNT	Count the no. of leading zero bits *Numără biţii de 0 superiori*		**MULX**	Unsigned MUL without affecting flags and arbitrary destination registers *Înmulţire MUL fără afectarea flagurilor şi registru detinaţie arbitrar*
TZCNT	Count the no. of trailing zero bits *Numără biţii de 0 inferiori*		**PDEP**	Parallel bits deposit *Depune biţi în paralel*
ANDN	Logical and not *Operaţie logică AND cu NOT*		**PEXT**	Parallel bits extract *Extrage biţi în paralel*
BEXTR	Bit field extract (with register) *Extrage un câmp de biţi*		**RORX**	Rotate right logical without affecting flags *Rotire logică spre dreapta fără afectarea flagurilor*
BLSI	Extract lowest set isolated bit *Extrage biul inferior setat*		**SARX**	Shift arithmetic right without affecting flags *Rotire aritmetică spre dreapta fără afectarea flagurilor*
BLSR	Reset lowest set bit *Resetează bitul inferior setat*		**SHRX**	Shift logical right without affecting flags *Deplasare logică spre dreapta fără afectarea flagurilor*
BLSMSK	Get mask up to lowest set bit *Maschează până la bitul inf. setat*		**SHLX**	Shift logical left without affecting flags *Deplasare logică spre stânga fără afectarea flagurilor*

[6] Posibilitatea execuţiei instrucţiunilor LZCNT şi POPCNT pe un CPU de tip AMD poate fi verificată prin consultarea bitului b5 (bit numit ‚abm') din registrul ECX după execuţia instrucţiunii cpuid cu EAX=80000001h la intrare (acesta trebuie să fie setat).

4.5.1. Instrucţiunea ADCX

Instrucţiunea **ADCX** ***(Unsigned Integer Addition of Two Operands with Carry Flag)*** face parte din setul de instrucţiuni **ADX**; pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b19 din registrul EDX (bit numit ,***adx***') să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare, deci se verifică dacă: EAX=7 --->***cpuid***---> $EDX|_{b19=1}$.

Instrucţiunea **ADCX** realizează o adunare de numere văzute ca fiind fără semn: se adună fără semn operandul destinaţie cu operandul sursă şi cu flagul Carry şi stochează rezultatul în operandul destinaţie.

Operandul destinaţie este un registru de uz general pe 32 sau 64 de biți, în timp ce operandul sursă poate fi un registru general sau o locație de memorie, de asemenea, pe 32 sau 64 de biți. Starea lui CF reprezintă un transport apărut de la o adunare anterioară. Instrucțiunea setează CF în caz că a apărut transport la adunarea fără semn a operanzilor.

Instrucțiunea ADCX este destinată realizării operaţiilor de adunare cu Carry în contextul adunării unei serii de operanzi.

ADCX destinaţie, sursă *; adună* ***fără semn*** *sursa şi* ***CF*** *(Carry flag) la destinaţie şi* ***actualizează CF***
ADCX {$reg_{32,64}$}, {$reg_{32,64}|mem_{32,64}$}

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri de tip GPR şi nu sunt suportate valori imediate ca operanzi; operandul sursă poate fi şi din memorie;
- Instrucţiunea ADCX nu afectează flagul OF: valoarea anterioară a OF va rămâne neafectată;
  - În caz că e nevoie de modificarea flagurilor CF şi OF:
    se pot folosi instrucţiuni precum STC (CF=1), CLC (CF=0), XOR (CF=0, OF=0);
- Instrucţiunea ADCX modifică flagurile: CF e actualizat pe baza rezultatului obţinut după adunare,
  dar flagurile OF, SF, ZF, AF şi PF sunt neafectate.

4.5.2. Instrucţiunea ADOX

Instrucţiunea **ADOX** ***(Unsigned Integer Addition of Two Operands with Overflow Flag)*** face parte din setul de instrucţiuni **ADX**; pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b19 din registrul EDX (bit numit ,***adx***') să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare: EAX=7 --->***cpuid***---> $EDX|_{b19=1}$.

Instrucţiunea **ADOX** realizează o adunare de numere văzute ca fiind fără semn: se adună fără semn operandul destinaţie cu operandul sursă şi cu flagul Overflow şi stochează rezultatul în operandul destinaţie.

Operandul destinaţie este un registru de uz general pe 32 sau 64 de biți, în timp ce operandul sursă poate fi un registru general sau o locație de memorie, de asemenea, pe 32 sau 64 de biți. Starea lui OF reprezintă un transport apărut de la o adunare anterioară. Instrucțiunea setează flagul OF în caz că a apărut transport la adunarea fără semn a operanzilor.

ADOX este destinată realizării operaţiilor de adunare cu Carry (văzut ca overflow) în contextul adunării unei serii de operanzi.

ADOX destinaţie, sursă *; adună **fără semn** sursa şi **OF** (Overflow flag) la destinaţie şi **actualizează OF***
ADOX {$reg_{32,64}$}, {$reg_{32,64}|mem_{32,64}$}

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri de tip GPR şi nu sunt suportate valori imediate ca operanzi; operandul sursă poate fi şi din memorie;
- Instrucţiunea ADOX nu afectează flagul CF: valoarea anterioară a lui CF va rămâne neafectată;
  - în caz că e nevoie de modificarea flagurilor CF şi OF, se pot folosi instrucţiuni precum: STC (CF=1), CLC (CF=0), XOR (CF=0, OF=0);
- Instrucţiunea ADOX modifică flagul OF: acesta este actualizat pe baza rezultatului obţinut după adunare, dar după o altă regulă decât ADD sau ADC: flagul OF va fi setat dacă a apărut un transport la adunare (transport care la ADD sau ADC ar fi setat flagul CF, deci ar fi apărut "Carry");
- Flagurile CF, SF, ZF, AF şi PF sunt neafectate.

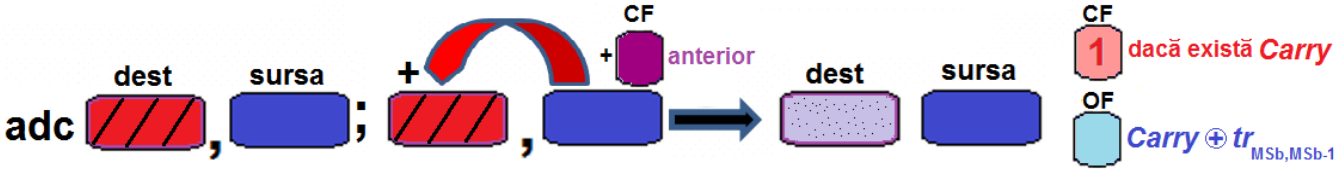


Figura 4-5.1. Ilustrarea modului de operare al instrucţiunii ***ADC***

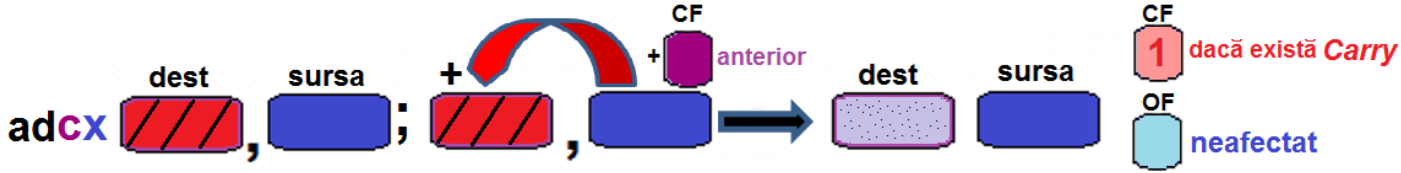


Figura 4-5.2. Ilustrarea modului de operare al instrucţiunii ***ADCX***

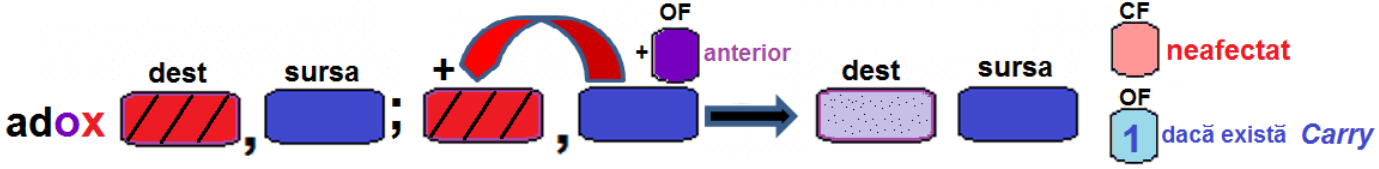


Figura 4-5.3. Ilustrarea modului de operare al instrucţiunii ***ADOX***

4.5.3. Instrucţiunea POPCNT

Instrucţiunea **POPCNT** (***Population Count*** sau ***Return the Count of Number of Bits Set to 1***) găseşte numărul de biţi de 1 din operandul sursă (al doilea operand) şi returnează această valoare în operandul destinaţie.

POPCNT destinaţie, sursă ***; numără biţii de 1 din sursă***
POPCNT {reg$_{16,32,64}$}, {reg$_{16,32,64}$|mem$_{16,32,64}$}

Pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b23 din registrul ECX (bit numit ,***popcnt***') să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=1 la intrare: EAX=1 --->***cpuid***---> ECX|$_{b23=1}$.

Figura 4-5.4. Ilustrarea modului de operare al instrucţiunii ***POPCNT***

Observaţii:

- Această instrucţiune este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 16, 32 sau 64 biţi;
- Suportă doar regiştri GPR; nu sunt suportate valori imediate ca operanzi, dar operandul sursă poate fi şi din memorie;
- Instrucţiunea POPCNT afectează flagurile: OF, SF, ZF, AF, CF, PF sunt toate resetate;
- Flagul ZF va fi setat dacă sursa nu conţine nici un bit de 0, caz în care şi destinaţia va fi 0.

4.5.4. Instrucţiunile LZCNT şi TZCNT

Pentru ca instrucţiunea **LZCNT** să fie suportată de procesor, e nevoie ca bitul b5 (bit numit ,***abm***') din registrul ECX să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=80000001h la intrare: EAX=80000001h --->***cpuid***---> $ECX|_{b5=1}$.

Pentru ca instrucţiunea **TZCNT** să fie suportată de procesor, e nevoie ca bitul b3 (bit numit ,***BMI1***') din registrul EBX să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare: EAX=7h --->***cpuid***---> $EBX|_{b3=1}$.

Instrucţiunea **LZCNT** (***Count the Number of Leading Zero Bits*** sau ***Leading Zero Bits Count)*** numără biţii de zero cei mai semnificativi din operandul sursă (al doilea) (sau numără - dinspre MSb înspre LSb - biţii de 0 până la primul bit de 1) şi depune acest nr ca rezultat în operandul destinaţie (primul operand). Instrucţiunea LZCNT este asemănătoare cu instrucţiunea BSR; cele două diferă prin faptul că LZCNT returnează numărul de biţi de 0 găsiţi, în timp ce BSR returnează poziţia primului bit de 1 găsit. În plus, când sursa este 0, LZCNT returnează în destinaţie dimensiunea operandului sursă, în timp ce BSR returnează o valoare nedefinită (poate fi diferită de 0) şi ZF=1.

LZCNT destinaţie, sursă ***; numără biţii de zero de la începutul sursei şi depune rezultatul în destinaţie***
LZCNT {$reg_{16,32,64}$}, {$reg_{16,32,64}|mem_{16,32,64}$}

Figura 4-5.5. Ilustrarea modului de operare al instrucţiunii ***LZCNT***

Instrucţiunea **TZCNT** (***Count the Number of Trailing Zero Bits*** sau ***Trailing Zero Bits Count)*** numără biţii de zero cei mai puţin semnificativi din operandul sursă (al doilea) (sau numără - dinspre LSb înspre MSb - biţii de 0 până la primul bit de 1) şi depune rezultatul în operandul destinaţie (primul). Instrucţiunea TZCNT este o extensie a lui BSF: pe un procesor pe care instrucţiunea TZCNT nu este suportată, aceasta este înlocuită cu BSF. Diferenţa cea mai mare între TZCNT şi BSF este că în destinaţie, TZCNT returnează dimensiunea operandului sursă dacă sursa este 0, în timp ce BSF lasă destinaţia nedefinită (poate fi diferită de 0) şi returnează ZF=1.

TZCNT destinaţie, sursă *; numără biţii de zero de la sfârşitul sursei şi depune rezultatul în destinaţie*
TZCNT {reg16,32,64}, {reg16,32,64|mem16,32,64}

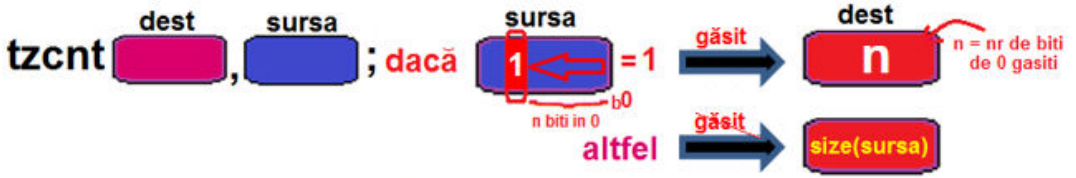


Figura 4-5.6. Ilustrarea modului de operare al instrucţiunii ***TZCNT***

Observaţii:

- Instrucţiunile **TZCNT** şi **LZCNT** sunt suportate în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 16, 32 sau 64 biţi;
- Instrucţiunea **LZCNT** ***(Leading Zeros Count)*** este aproape identică cu instrucţiunea ***BSR (Bit Scan Reverse)***, dar în loc să seteze ZF atunci când nu găseşte vreun bit de 1 în sursă, va seta ZF=1 dacă rezultatul este 0 şi CF=1 dacă sursa e 0; în plus, returnează o valoare definită (dimensiunea operandului sursă în biţi) dacă sursa este zero; flagurile OF, SF, PFşi AF sunt nedefinite;
- Instrucţiunea **TZCNT** ***(Trailing Zeros Count)*** este aproape identică cu instrucţiunea ***BSF (Bit Scan Forward)***, dar în loc să seteze ZF atunci când nu găseşte vreun bit de 1 în sursă, va seta ZF=1 dacă rezultatul este 0 şi CF=1 dacă sursa e 0; în plus, returnează o valoare definită (dimensiunea operandului sursă în biţi) dacă sursa este zero; ZF este setat la 1 în caz că LSb=1 (cel mai puţin semnificativ bit din sursă este 1), deci rezultatul este 0 (există 0 biţi găsiţi); CF este setat la 1 dacă sursa este plină cu 0; flagurile OF, SF, PF şi AF sunt nedefinite.

4.5.5. Instrucţiunea ANDN

Instrucţiunea **ANDN** (***Logical AND NOT***) realizează o operaţie logică AND între inversul valorii conţinute în sursă1 şi valoarea din sursă2, stocând rezultatul în destinaţie. Dintre cei 3 operanzi ai instrucţiunii, doar ultimul poate fi şi zonă de memorie pe 32 sau 64 biţi, înafară de posibilitatea de a fi regiştri GPR de 32 sau 64 biţi.

ANDN destinaţie, sursă1, sursă 2 *; operaţie logică AND între invers(sursă1) şi (sursă2) şi pune rezultatul în destinaţie*
ANDN $\{reg_{32,64}\}$, $\{reg_{32,64}\}$, $\{reg_{32,64}|mem_{32,64}\}$

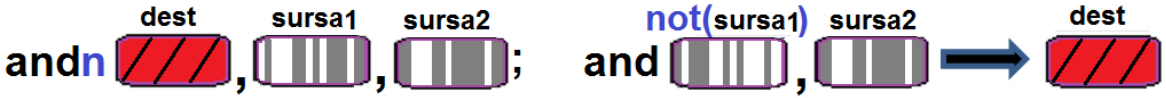


Figura 4-5.7. Ilustrarea modului de operare al instrucţiunii ***ANDN***

Pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b3 (bit numit ‚***BMI1***') din registrul EBX să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare: EAX=7h --->***cpuid***---> $EBX|_{b3=1}$.

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR şi nu sunt suportate valori imediate ca operanzi; doar cel de-al treilea operand poate fi şi din memorie;
- Flagurile SF şi ZF sunt actualizate pe baza rezultatului obţinut;
- Flagurile OF şi CF sunt resetate, iar flagurile AF şi PF sunt nedefinite;
- Operaţie echivalentă în C: ~x & y;

4.5.6. Instrucţiunea BEXTR

Instrucţiunea **BEXTR** (***Bit Field Extract***) extrage un câmp de biţi (biţi contigui) din operandul sursă folosind un index şi o lungime specificate în operandul *control*: biţii 7…0 precizează poziţia bitului de la care se începe extracţia (START), iar biţii 15…8 specifică LUNGIMEA câmpului ce se va extrage (ca număr de biţi). La folosirea unei valori START mai mare decât dimensiunea operadului, nu se va extrage nici un bit, iar dacă LUNGIMEA determină depăşirea operandului sursă, se vor lua doar biţii până la poziţia dată de dimensiunea operandului -1. Biţii extraşi vor fi înscrişi în destinaţie, începând de la bitul LSb, iar pe poziţiile din destinaţie care nu vor conţine biţi extraşi vor fi zerouri. Instrucţiunea BEXTR face parte din setul BMI1.

BEXTR destinaţie, sursă, control *; extrage un câmp de biţi (contigui) din (sursă) în funcţie de control*
BEXTR {reg32,64}, {reg32,64|mem32,64}, {reg32,64} *şi pune rezultatul în destinaţie*

Figura 4-5.8. Ilustrarea modului de operare al instrucţiunii ***BEXTR***

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR şi nu sunt suportate valori imediate ca operanzi;
- Flagul ZF este actualizat pe baza rezultatului obţinut (dacă nu se extrag biţi şi destinaţia e 0, atunci ZF=1).
- AF, SF şi PF sunt nedefiniţi, iar CF şi OF sunt resetaţi;
- Operaţie echivalentă în C: (src >> start) & ((1 << len)-1);

4.5.7. Instrucţiunea BLSI

Instrucţiunea **BLSI** (***Extract Lowest Set Isolated Bit***) extrage bitul c.m.p.s. de valoare 1 sau altfel spus, caută dinspre LSb înspre MSb primul bit de 1 şi pe poziţia respectivă depune în destinaţie un bit de 1, toţi ceilalţi biţi din destinaţie fiind 0.

Dacă nu găseşte nici un bit de 1 în sursă, instrucţiunea va pune toţi biţii din destinaţie în 0 şi va seta ZF şi CF. Face parte din setul BMI1.

BLSI destinaţie, sursă *; caută primul bit de 1 dinspre LSb şi depune în destinaţie pe acea poziţie un 1*
BLSI {reg$_{32,64}$}, {reg$_{32,64}$|mem$_{32,64}$}

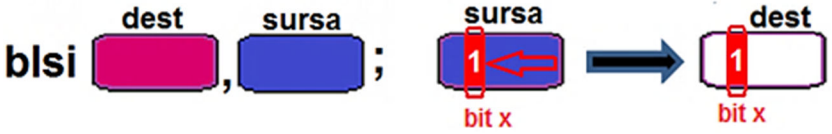


Figura 4-5.9. Ilustrarea modului de operare al instrucţiunii ***BLSI***

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR şi nu sunt suportate valori imediate ca operanzi; cel de-al doilea operand poate fi şi din memorie;
- Flagurile ZF şi SF sunt actualizate pe baza rezultatului obţinut. Flagul CF se setează dacă sursa este 0. Flagul OF este resetat, dar dacă sursa este 0, se setează. Flagurile AF şi PF sunt nedefinite.
- Operaţie echivalentă în C: x & -x.

4.5.9. Instrucţiunea BLSR

Instrucţiunea **BLSR** (***Reset Lowest Set Bit***) resetează bitul c.m.p.s. de valoare 1 sau altfel spus, caută dinspre LSb înspre MSb primul bit de 1 şi pe poziţia respectivă depune în destinaţie un bit de 0, toţi ceilalţi biţi din destinaţie fiind identici cu cei din sursă. Dacă operandul sursă este 0, instrucţiunea setează CF. Face parte din setul BMI1.

BLSR destinaţie, sursă *; caută primul bit de 1 dinspre LSb şi îl resetează în destinaţie, restul biţilor fiind copiaţi din sursă*
BLSR $\{reg_{32,64}\}$, $\{reg_{32,64}|mem_{32,64}\}$

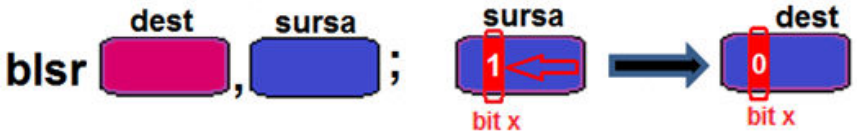


Figura 4-5.10. Ilustrarea modului de operare al instrucţiunii ***BLSR***

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR şi nu sunt suportate valori imediate ca operanzi;
- Flagurile ZF şi SF sunt actualizate pe baza rezultatului obţinut. Flagul CF este setat dacă sursa este 0. Flagul OF este resetat, iar AF şi PF sunt nedefinite;
- Operaţie echivalentă în C: x & (x - 1)

4.5.8. Instrucţiunea BLSMSK

Instrucţiunea **BLSMSK** (***Get Mask Up to Lowest Set Bit***) caută dinspre LSb înspre MSb primul bit de 1 şi pe poziţia respectivă depune în destinaţie un bit de 1, toţi biţii de pe poziţie inferioară fiind setaţi (mascaţi), iar cei de pe poziţie superioară fiind copiaţi din sursă. Dacă operandul sursă este 0, instrucţiunea setează toţi biţii din destinaţie la 1, iar CF=1. Instrucţiunea BLSMSK face parte din setul BMI1.

BLSMSK destinaţie, sursă *; setează toţi biţii din destinaţie la 1, începând de la b0 până la bitul c.m.p.s. setat din sursă*
BLSMSK {reg$_{32,64}$}, {reg$_{32,64}$|mem$_{32,64}$}

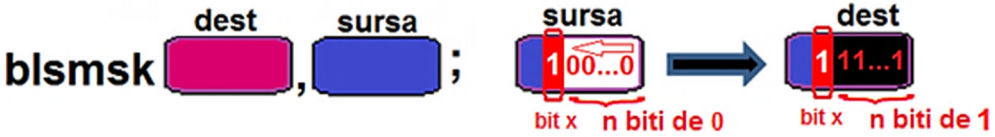


Figura 4-5.11. Ilustrarea modului de operare al instrucţiunii ***BLSMSK***

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 2 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR şi nu sunt suportate valori imediate ca operanzi;
- Flagul SF este actualizat pe baza rezultatului obţinut. CF este setat dacă sursa este 0, flagurile ZF şi OF sunt resetate, iar AF şi PF sunt nedefinite;
- Operaţie echivalentă în C: x ^ (x - 1).

4.5.10. Instrucţiunea BZHI

Instrucţiunea **BZHI** (***Zero High Bits Starting with Specified Bit Position***) copiază biţii din sursă (al doilea operand) în destinaţie (primul operand), curăţând biţii de ordin superior în concordanţă cu valoarea INDEX specificată de operandul cu acelaşi nume (al treilea); biţii forţaţi în 0 sunt de la poziţia dată de INDEX, inclusiv, înspre MSb). Acest INDEX este specificat de biţii 7…0, permiţând astfel o valoarea maximă de 255. Cum dimensiunea operanzilor sursă şi destinaţie este de maxim 32, respectiv 64 biţi, valoarea specificată pt INDEX poate depăşi dimensiunea operanzilor; în cazul în care INDEX este mai mare decât dimensiunea operanzilor -1, CF va deveni 1; altfel, CF=0.

BZHI destinaţie, sursă, index ***; Zerorizează biţii din destinaţie, începând de la poziţia dată de INDEX inclusiv,***
BZHI {reg32,64},{reg32,64|mem32,64},{reg32,64} ***; restul biţilor fiind copiaţi din sursă (de pe poziţiile INDEX-1 ...LSb)***

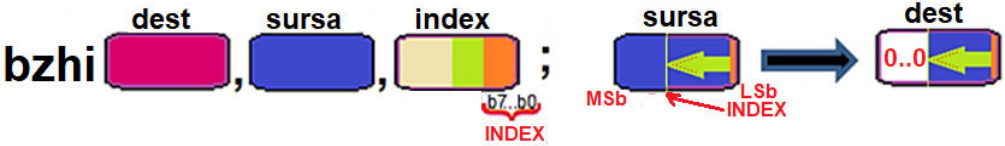


Figura 4-5.12. Ilustrarea modului de operare al instrucţiunii ***BZHI***

Pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b8 (bit numit ,***BMI2'***) din registrul EBX să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare: EAX=7h --->***cpuid***---> $EBX|_{b8=1}$.

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: 32 sau 64 biţi;
- Nu sunt suportaţi decât regiştri GPR (şi memorie pt sursă) şi nu sunt suportate valori imediate ca operanzi;
- Flagul CF este setat în cazul în care valoarea INDEX a fost mai mare decât dimensiunea operanzilor -1 , altfel CF=0; Flagurile ZF şi SF sunt actualizate pe baza rezultatului obţinut, iar OF este curăţat. Flagurile AF şi PF sunt nedefinite.

4.5.11. Instrucţiunea MULX

Instrucţiunea **MULX** (***Unsigned Multiply Without Affecting Flags***) realizează o înmulţire considerând valorile fără semn şi fără să afecteze vreun flag (sufixul X de la sfârşit indică faptul că nu afectează (-/E/R)FLAGS). Instrucţiunea MULX are un operand implicit: EDX sau RDX (depinde de execuţia pe 32 sau 64 biţi), operand care se va înmulţi cu cel de-al treilea operand specificat în instrucţiune (sursa). Rezultatul obţinut (pe 64 sau 128 biţi) se va depune în perechea de regiştrii formată din primii 2 operanzi: destinaţie2 va conţine jumătatea low a rezultatului (prima care se scrie), iar destinaţie1 va conţine jumătatea high a rezultatului (înscrisă a doua oară).

MULX destinaţie1, destinaţie2, sursă ***; realizează înmulţirea fără semn a registrului. EDX sau RDX cu sursa***
MULX{reg$_{32,64}$},{reg$_{32,64}$},{reg$_{32,64}$|mem$_{32,64}$} ***; depune rezultatul în perechea de regiştrii destinaţie1 : destinaţie2***

Figura 4-5.13. Ilustrarea modului de operare al instrucţiunii ***MULX***

Pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca bitul b8 (bit numit ,***BMI2'***) din registrul EBX să fie setat după execuţia instrucţiunii ***cpuid*** cu EAX=7 la intrare: EAX=7h --->***cpuid***---> EBX|$_{b8=1}$.

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: 32 sau 64 biţi; sursa poate fi şi din memorie, pe 32 sau 64 biţi.
- Operandul implicit la înmulţire aici este EDX resp. RDX şi nu Acc (cum a fost în cazul înmulţirii clasice);
- Rezultatul se va înscrie astfel: prima parte care se înscrie în registru este cea low, deci *destinaţie2*, iar a doua oară se înscrie şi jumătatea high a rezultatului în *destinaţie1* deci. Astfel, dacă cei 2 regiştri destinaţie sunt unul şi acelaşi, acest registru va conţine doar jumătatea high a rezultatului înmulţirii;
- Instrucţiunea MULX nu afetează nici un flag, aceasta permiţându-i să fie mai eficientă (rapidă) la execuţie.

4.5.12. Instrucţiunile PDEP şi PEXT

Instrucţiunile **PDEP** ***(Parallel bit deposit)*** şi **PEXT** ***(Parallel bit extract)*** sunt instrucţiuni generalizate pentru comprimarea şi respectiv extensia unei valori, fără afectarea vreunui flag. Ambele instrucţiuni folosesc 2 operanzi: sursă şi selector. Selectorul este o hartă ce va specifica modul cum se vor selecta biţii ce trebuie supuşi împachetării, resp. despachetării.

Instrucţiunea **PEXT** ***(Parallel bit extract)*** copiază biţii selectaţi de selector (folosind biţi de 1 se consideră selectaţi) din sursă într-o zonă continguă din destinaţie, începând de la LSb (deci în partea low a destinaţiei vor fi depuşi biţii extraşi); în partea high a destinaţiei (cea rămasă) se depun zerouri.

Instrucţiunea **PDEP** ***(Parallel bit deposit)*** realizează operaţia inversă: ia biţii din zona low contiguă a sursei (atâţia biţi câţi biţi de 1 are selectorul) şi-i distribuie în destinaţie pe poziţiile indicate de selector (sau mască); celelalte poziţii din destinaţie vor avea biţi de 0.

Pentru ca aceste instrucţiuni să fie suportate de procesor, e nevoie ca b8 (bit numit ‚***BMI2***') din registrul EBX după execuţia instrucţiunii cpuid cu EAX=7 la intrare să fie setat: EAX=7 -> ***cpuid*** -> EBX|b8=1.

Tabelul 4-5.1. Exemple de operare la nivel de 8 biţi a instrucţiunilor PDEP şi PEXT [1]

Sursa	Selectorul	PEXT - Parallel bit extract (selectează biţii de pe poziţia măştii (1), dinspre MSb) *zerourile din mască sunt depuse la stânga, iar biţii extraşi sunt depuşi la dreapta*	PDEP - Parallel bit deposit (selectează toţi biţii dinspre LSb), *dar îi depune pe poziţiile măştii (1); structura destinaţiei e la fel cu a măştii*
abcd'efgh	1111'0000	0000'abcd	efgh'0000
abcd'efgh	0000'1111	0000'efgh	0000efgh
abcd'efgh	1111'1100	00ab'cdef	cdef'gh00
abcd'efgh	0001'1111	000d'efgh	000d'efgh
abcd'efgh	1101'1101	00ab'defh	cd0e'fg0h
abcd'efgh	0101'0101	0000'bdfh	0e0f'0g0h

Instrucţiunea **PDEP** (***Parallel Bits Deposit***) folosește o mască (sau selector) în cel de-al treilea operand din instrucţiune pentru a transfera sau „împrăștia” pe acele poziţii selectate în destinaţie (pornind dinspre LSb) biții contigui luaţi din operandul sursă (al doilea operand). PDEP ia biți de ordin mic din sursă și îi depozitează în destinație pe locațiile corespunzătoare care sunt setate în selector (mască). Toți ceilalți biți (biții care nu sunt setați în mască) în destinaţie vor fi puşi în zero.

PDEP destinaţie, sursă, mască ***; transferă sau împrăştie biţii contigui c.m.p.s. din sursă în destinaţie***
PDEP {reg32,64}, {reg32,64},{reg32,64| mem32,64} ***;pe poziţiile date de mască sau selector***

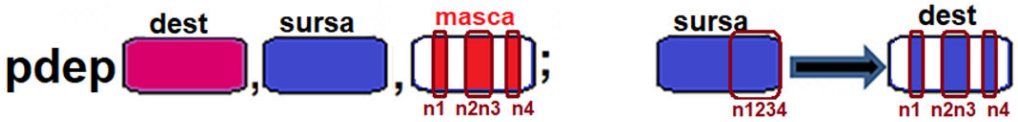


Figura 4-5.14. Ilustrarea modului de operare al instrucţiunii ***PDEP***

Instrucţiunea **PEXT** (***Parallel Bits Extract***) folosește o mască (sau selector) în cel de-al treilea operand din instrucţiune pentru a selecta anumite poziţii din sursă (al doilea operand). Biţii de pe acele poziţii selectate se vor depune în destinaţie (pornind dinspre LSb) ca biți contigui. Toți ceilalți biți (biții superiori) în destinaţie vor fi puşi în zero.

PEXT destinaţie, sursă, mască ***; transferă în destinaţie (plecând de la LSb), ca biţi contigui,***
PEXT {reg32,64}, {reg32,64},{reg32,64| mem32,64} ***; biţii extraşi din sursă de pe poziţiile date de mască sau selector***

Figura 4-5.15. Ilustrarea modului de operare al instrucţiunii ***PEXT***

Observaţii:

- Aceste instrucţiuni nu sunt suportate în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: 32 sau 64 biţi; masca poate fi şi din memorie, pe 32 sau 64 biţi;
- Instrucţiunile PDEP şi PEXT nu afectează niciun flag din (-/E/R)FLAGS);
- Ambele instrucţiuni fac parte din setul BMI2.

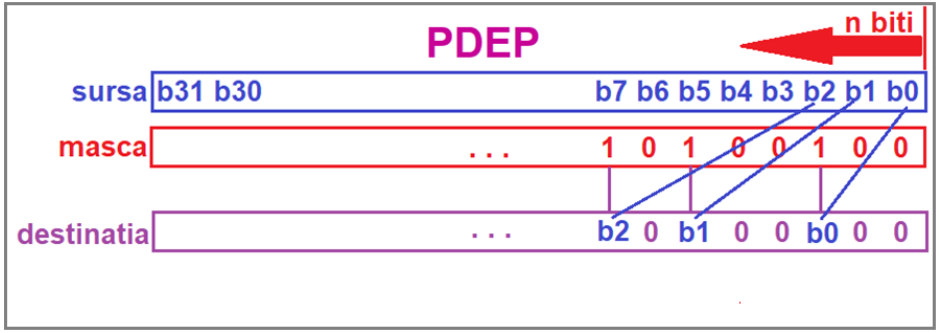


Figura 4-5.16. Ilustrarea schematică a modului de operare al instrucţiunii ***PDEP***

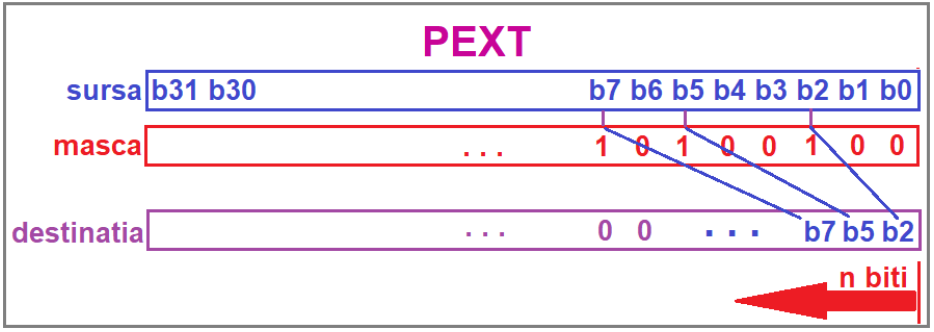


Figura 4-5.17. Ilustrarea schematică a modului de operare al instrucţiunii ***PEXT***

4.5.13. Instrucţiunea RORX

Instrucţiunea **RORX** ***(Rotate Right Logical Without Affecting Flags)*** roteşte operandul sursă (al doilea operand din instrucţiune) înspre dreapta de atâtea ori cât specifică operandul contor (al treilea) fără să afecteze flagurile aritmetice şi depune rezultatul în operandul destinaţie (primul operand specificat în instrucţiune).

RORX - pentru ca această instrucţiune să fie suportată de procesor, e nevoie ca b8 (bit numit ‚BMI2') din registrul EBX după execuţia instrucţiunii cpuid cu EAX=7 la intrare să fie setat: EAX=7 ->***cpuid*** -> EBX|b8=1

RORX destinaţie, sursă, contor *; roteşte biţii din sursă cu* ***contor poziţii*** *spre* ***dreapta, dinspre MSb***
ROR {reg$_{32,64}$}, {reg$_{32,64}$| mem$_{32,64}$}, { imed$_8$ } *; rezultatul se depune în* ***destinaţie***

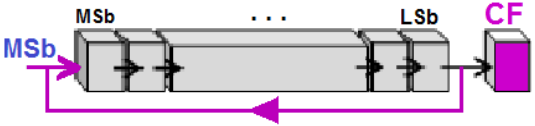


Figura 4-5.18. Ilustrarea modului de operare al instrucţiunii ***RORX (identic cu ROR)***

Observaţii:

- Această instrucţiune nu este suportată în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi nu trebuie să fie identică, decât primii 2: aceştia pot fi de 32 sau 64 biţi; operandul contor este o valoare imediată pe 8 biţi;
- Pentru operanzi de 32 biţi, se realizează o mască AND între valoarea lui contor şi 1Fh, iar pentru operanzi de 64biţi, se foloseşte ca mască 3Fh;
- Instrucţiunea RORX nu afectează niciun flag din (-/E/R)FLAGS);
- Instrucţiunea face parte din setul BMI2.

4.5.14. Instrucţiunile SARX, SHLX şi SHRX

Instrucţiunile **SARX/ SHLX/ SHRX** ***(Shift Without Affecting Flags)*** funcţionează în mod similar celor SAR, SHL, SHR, doar că ele nu afectează niciun flag aritmetic, fiind astfel mai rapide la execuţie. Ca sintaxă sunt diferite, deoarece operandul sursă (al doilea operand specificat în instrucţiune) este deplasat spre stânga/ dreapta cu un nr de poziţii specificat de openrandul contor (al treilea operand), iar rezultatul se depune în destinaţie (primul operand).

RORX destinaţie, sursă, contor *; roteşte biţii din sursă cu **contor poziţii** spre **dreapta, dinspre MSb***
ROR {reg32,64}, {reg32,64| mem32,64}, {reg32,64} *; rezultatul se depune în **destinaţie***

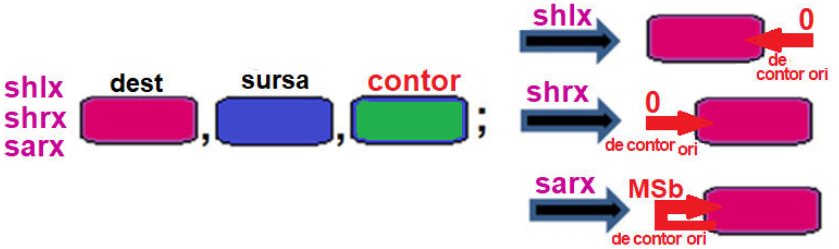


Figura 4-5.19. Ilustrarea modului de operare al instrucţiunilor ***SHLX, SHRX, SARX***

Pentru ca aceste instrucţiuni să fie suportate de procesor, e nevoie ca b8 (bit numit ‚BMI2') din registrul EBX după execuţia instrucţiunii cpuid cu EAX=7 la intrare să fie setat: EAX=7 ->***cpuid*** -> EBX|b8=1.

Observaţii:

- Aceste instrucţiuni nu sunt suportate în mod real sau mod virtual 8086;
- Dimensiunea celor 3 operanzi trebuie să fie identică: aceştia pot fi de 32 sau 64 biţi; pentru operanzi de 32 biţi, se realizează o mască AND între valoarea lui contor şi 1Fh, iar pentru operanzi de 64biţi, se foloseşte ca mască 3Fh;
- Instrucţiunile nu afectează niciun flag aritmetic din (-/E/R)FLAGS);
- Instrucţiunile fac parte din setul BMI2.

4.5.15. Exemple

Exemple de *instrucţiuni legale*:

Exemple 5.4-1 Diferenţa între ADC, ADCX, ADOX

```
xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 6B000000h   ; EBX = 6B 00 00 00h
adc EAX, EBX         ; EAX = E5 00 00 01h, CF=0, OF=1, SF=1 (din operaţia curentă => carry=0, tr=1, deci OF=1)

xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 6B000000h   ; EBX = 6B 00 00 00h
adcx EAX, EBX        ; EAX = E5 00 00 01h, CF=0, OF=0, SF=0 (din operaţia curentă => carry=0, tr=1, deci OF=1 dar
                     ; instrucţiunea ADCX nu afectează flagul OF, rămâne aşa cum a fost, deci OF va fi 0)

xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 6B000000h   ; EBX = 6B 00 00 00h
adox EAX, EBX        ; EAX = E5 00 00 00h, CF=1, OF=0, SF=0 (din operaţia curentă => carry=0, tr=1, dar
                     ; instrucţiunea ADOX nu afectează flagul CF, deci rămâne aşa cum a fost, iar
                     ; un eventual Carry care ar apărea va fi văzut ca Overflow, deci OF = 0)
```

Exemple 5.4-2

```
xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 9B000000h   ; EBX = 9B 00 00 00h
adc EAX, EBX         ; EAX = 15 00 00 01h, CF=1, OF=0, SF=0 (din operaţia curentă => carry=1, tr=1, deci OF=0)

xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 9B000000h   ; EBX = 9B 00 00 00h
adcx EAX, EBX        ; EAX = 15 00 00 01h, CF=1, OF=0, SF=0 (din operaţia curentă => carry=1, tr=1
                     ; instrucţiunea ADCX nu afectează flagul OF, deci OF va fi 0 – rămâne din valoarea anterioară aşa)

xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 7A000000h   ; EAX = 7A 00 00 00h
mov EBX, 9B000000h   ; EBX = 9B 00 00 00h
adox EAX, EBX        ; EAX = 15 00 00 00h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=1, dar
                     ; instrucţiunea ADOX nu afectează flagul CF, deci rămâne aşa cum a fost, iar
                     ; un eventual Carry care ar apărea va fi văzut ca Overflow, deci OF = 1)
                     ; la ADOX, când apare Carry, e văzut ca Overflow şi acel tr_MSb-1,MSb nu contează
```

Exemple 5.4-3

```
xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 8A000000h   ; EAX = 8A 00 00 00h
mov EBX, 9B000000h   ; EBX = 9B 00 00 00h
adc EAX, EBX         ; EAX = 25 00 00 01h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=0, deci OF=1)
```

```
; adc EAX, EBX          ; EAX = 25 00 00 01h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=0, deci OF=1)

xor EAX,EAX            ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                    ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 8A000000h     ; EAX = 8A 00 00 00h
mov EBX, 9B000000h     ; EBX = 9B 00 00 00h
adcx EAX, EBX          ; EAX = 25 00 00 01h, CF=1, OF=0, SF=0 (din operaţia curentă => carry=1, tr=0;
                       ; instrucţiunea ADCX nu afectează flagul OF, deci OF va fi 0 – rămâne din operaţia anterioară aşa)

xor EAX,EAX            ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                    ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 8A000000h     ; EAX = 8A 00 00 00h
mov EBX, 9B000000h     ; EBX = 9B 00 00 00h
adox EAX, EBX          ; EAX = 25 00 00 00h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=0, dar
                       ; instrucţiunea ADOX nu afectează flagul CF, deci rămâne aşa cum a fost, iar
                       ; un eventual Carry care ar apărea va fi văzut ca Overflow, deci OF = 1)
                       ; la ADOX, când apare Carry, e văzut ca Overflow şi acel tr_MSb-1,MSb nu contează
```

Exemplul 5.4-4

```
xor EAX,EAX            ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                    ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 8A000000h     ; EAX = 8A 00 00 00h
mov EBX, 9B000000h     ; EBX = 9B 00 00 00h
adox EAX, EBX          ; EAX = 25 00 00 00h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=0)
mov ECX, 0E4000000h    ; ECX = E4 00 00 00h
adox EAX, ECX          ; EAX = 09 00 00 01h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=1,
```

Exemplul 5.4-5

```
xor EAX,EAX          ; curăţă toate flags care ne interesează: CF=0, OF=0, SF=0
stc                  ; CF=1 – presupunem că o instrucţiune de adunare anterioară a generat acest CF=1
mov EAX, 8A000000h   ; EAX = 8A 00 00 00h
mov EBX, 9B000000h   ; EBX = 9B 00 00 00h
adox EAX, EBX        ; EAX = 25 00 00 00h, CF=1, OF=1, SF=0 (din operaţia curentă => carry=1, tr=0)
mov ECX, 64000000h   ; ECX = 64 00 00 00h
adox EAX, ECX        ; EAX = 89 00 00 01h, CF=1, OF=0, SF=0 (din operaţia curentă => carry=0, tr=1, dar
                     ; instrucţiunea ADOX nu afectează flagul CF, deci rămâne aşa cum a fost, iar
                     ; un eventual Carry care ar apărea va fi văzut ca Overflow, deci OF = 0)
```

Exemplul 5.4-6

```
stc                  ; CF=1
mov eax,8B203040h    ; EAX= -1960824768 (s) = 2.334.142.528 (uns)
mov ebx, 87654321h   ; EBX= -2023406815 (s) = 2.271.560.481 (uns)
adcx eax, ebx        ; EAX =12857362h (s) = 310.735.714 (uns), CF=1, OF=0
```

Exemplul 5.4-7 Exemplificare POPCNT

```
mov EAX,12345678h    ; EAX = 12345678h
popcnt BX, AX        ;  BX =       0008h
popcnt EBX, EAX      ; EBX = 0000000Dh
```

Exemplul 5.4-8 Exemplificare LZCNT (versus BSR)

```
 mov EAX, 01234h     ; EAX = 00001234h
 mov EBX, 0412300h   ; EBX = 00412300h
 lzcnt EAX, EBX      ; EAX=00000009h
 ; a găsit 9 biţi superiori în 0
```

```
mov EAX, 01234h      ; EAX = 00001234h
mov EBX, 0412300h    ; EBX = 00412300h
bsr EAX, EBX         ; EAX=16h (adică b22=1)
; indexul sau rangul primului bit ≠0
```

Exemplul 5.4-9 Exemplificare TZCNT (versus BSF)

```
mov EAX, 01234h          ; EAX = 00001234h
mov EBX, 0412300h        ; EBX = 00412300h
tzcnt EAX, EBX           ; EAX=00000008h
; a găsit 8 biţi inferiori în 0
```

```
mov EAX, 01234h          ; EAX = 00001234h
mov EBX, 0412300h        ; EBX = 00412300h
bsf EAX, EBX             ; EAX=8h (adică b8=1)
; indexul sau rangul primului bit ≠0
```

Exemplul 5.4-10

```
mov EBX, 0
lzcnt EAX, EBX           ; EAX=20h, avem 32 biţi în 0, iar CF=1, ZF=0
tzcnt EAX, EBX           ; EAX=20h, avem 32 biţi în 0, iar CF=1, ZF=0

bsf EAX, EBX             ; EAX=???, dar ZF=1 deşi destinaţia e diferită de zero, CF=0
bsr EAX, EBX             ; EAX=???, dar ZF=1 deşi destinaţia e diferită de zero, CF=0
; de fapt, BSF şi BSR returnează destinaţia nedefinită (va fi atât cât a fost înainte de execuţia instrucţiunii resp.)
```

Exemplul 5.4-11

```
mov EAX, 01020304h       ; EAX = 01020304h -> /EAX = FE FD FC FBh
mov EBX, 11223344h       ; EBX = 11223344h
andn ECX, EAX, EBX       ; ECX = and (/EAX, EBX) = 10203040h
```

Exemplul 5.4-12

```
mov EAX,87654321h        ; EAX = 87654321h
mov EBX,11220817h        ; EBX = 11220817h
bextr ECX, EAX, EBX      ; ECX = 000000Eh ; extrage începând de la bitul 23 o lungime de 8 poziţii înspre MSb,
                         ; adică  EAX=1000 0111 0110 0101 0100 0011 0010 0001b, rezultă ECX = 0000000Eh
```

Exemplul 5.4-13

```
mov eax, 0D7654321h
mov ebx, 11220C10h
bextr ECX,EAX,EBX        ; ECX=00000765h, extrage de la bitul 16 o lungime de 12 biţi înspre MSb
```

Exemplul 5.4-14

```
mov EAX, 87654320h
blsi EBX, EAX          ; EBX=00000020h, adică bitul 5
```

Exemplul 5.4-15

```
mov EAX,87654320h
blsr EBX, EAX          ;  cel mai LSb setat e bitul 5 şi pe acesta îl resetează => EBX = 87654300h
```

Exemplul 5.4-16

```
mov ECX, 15            ; ECX = 15
mov EAX,12345678h      ; EAX = 12345678h
mov EBX, 87654321h     ; EBX = 87654321h
bzhi EAX, EBX, ECX     ; EAX = 00004321h, CF=0
```

Exemplul 5.4-17

```
mov EAX,12345678h      ; EAX = 12345678h
mov EBX, 9ABCDEF0h     ; EBX = 9ABCDEF0h
mov ECX, 34            ; ECX = 34
bzhi EAX, EBX, ECX     ; EAX = 9ABCDEF0h, CF=1 (s-a depăşit dimensiunea operandului)
```

Exemplul 5.4-18

```
mov EAX, 12345678h
mov EBX, 0F0F0F0Fh
pdep ECX, EAX, EBX     ; ECX=05060708h
pdep ECX, EBX, EAX     ; ECX=02340078h
```

Exemplul 5.4-19

```
mov EAX, 12345678h
mov EBX, 0FFFFh
pdep ECX, EAX, EBX     ; ECX=00005678h
pdep ECX, EBX, EAX     ; ECX=12345678h
```

Exemplul 5.4-20

```
mov EAX, 12345678h
mov EBX, 0F0F0F0Fh
pext ECX, EAX, EBX          ; ECX=00002468h
pext ECX, EBX, EAX          ; ECX=00000931h
```

Exemplul 5.4-21

```
mov EAX, 12345678h
mov EBX, 0FFFFh
pext ECX, EAX, EBX          ; ECX=00005678h
pext ECX, EBX, EAX          ; ECX=000000FFh
```

Exemplul 5.4-22

```
mov EAX, 1234567h           ; EAX = 1234567h
mov EBX, 89ABCDEFh          ; EBX = 89ABCDEFh
mov ECX, 8                  ; ECX = 8
; se vor executa independent :
a) sarx EAX, EBX, ECX       ; EAX = FF89ABCDh
b) shlx EAX, EBX, ECX       ; EAX = ABCDEF00h
c) shrx EAX, EBX, ECX       ; EAX = 0089ABCDh
d) rorx EAX, EBX, 8         ; EAX = EF89ABCDh
```

Capitolul 5. Instrucţiuni de manipulare a şirurilor

5.1. Instrucţiuni pentru operaţii primitive

Există 7 operaţii primitive specifice șirurilor, suportate încă de la **8086↑**, acestea sunt date de instrucţiunile
MOVS, LODS, STOS, CMPS, SCAS; **INS, OUTS**
Aceste instrucţiuni operează asupra unui ***octet (8 biţi)*** sau ***cuvânt (16 biţi)*** din memorie (deci asupra unor locaţii succesive). Începând cu **386↑**, acestea au fost suportate și la nivel de ***dublucuvânt (32 biţi)***, iar de la modul pe 64 biţi, pentru procesoare **Pentium 4↑** sau **Core 2↑,** sunt suportate chiar şi la nivel de ***cvadruplucuvânt (64 biţi)***.

Tipul operanzilor (adică la ce nivel se operează: de byte, de word, de doubleword sau de quadword) poate fi:

- *detectat automat de către asamblor prin specificarea operanzilor în mod explicit* (la instrucţiunile MOVS, CMPS, SCAS, etc de exemplu o instrucţiune MOVS la nivel de octet va fi automat înlocuită cu MOVSB)
- *precizat în mod explicit în mnemonica instrucţiunii* (MOVS**B**, CMPS**B,** etc);

Distincţia între dimensiunea operanzilor se realizează prin folosirea sufixelor:

B (byte), ***W (word)***, ***D (doubleword)***, respectiv ***Q (quadword)***.

Sintaxa instrucțiunii movs de exemplu poate lua formele:

movs {mem$_{8,16,32,64}$},{mem$_{8,16,32,64}$} sau ***movsb*** sau ***movsw*** sau ***movsd*** sau ***movsq***

Adresa din memorie (unde începe şirul) va fi specificată de :
ES:(E)DI pt operandul destinație şi **DS:(E)SI** pt operandul sursă în ***mod pe 32 biți***
(E/R)DI pt operandul destinație şi **(E/R)SI** pt operandul sursă în ***mod pe 64 biți.***

Astfel, ***adresele*** fiind deja stabilite pentru *destinație* şi *sursă*, mai e necesară doar precizarea ***dimensiunii operanzilor*** sau *numărul de locații afectate - aceasta* poate fi: un octet - va fi vizată o singură locație, un cuvânt - vor fi vizate 2 locații, un dublucuvânt - vor fi vizate 4 locații, un cvadruplucuvânt - vor fi vizate 8 locații începând de la ***acele adrese***.

Dimensiunea operanzilor se va putea deduce **fie din operanzii expliciți** - *byte* pt mem_8
- *word* pt mem_{16}
- *doubleword* pt mem_{32}
- *quadword* pt mem_{64}

fie din tipul mnemonicii folosite: - *byte* pt MOVSB, LODSB, STOSB, CMPSB, SCASB
- *word* pt MOVSW, LODSW, STOSW, CMPSW, SCASW
- *doubleword* pt MOVSD, LODSD, STOSD, CMPSD, SCASD
- *quadword* pt MOVSQ, LODSQ, STOSQ, CMPSQ, SCASQ

Instrucţiunile pentru şiruri realizează inclusiv actualizarea adreselor, însă acestea nu ştiu să repete operaţia de un număr de ori câte elemente are şirul de prelucrat. Astfel, se folosește combinarea instrucţiunii pt lucrul cu şiruri cu algoritmi de repetare, gen: ***prefixe de repetare*** *(rep/repe/repz/repne/repnz)* sau
bucle cu salt condiţionat *(cu loopz, loopnz, loope, etc)*;
uneori se poate folsi ***loop*** pentru repetarea operației de ECX ori.

În mod implicit, la instrucţiunile pe şiruri, se consideră următoarele:

Observații:

- Sursa şi destinația sunt implicite:
  - perechea de regiştrii DS:[-/E] SI sau registrul RSI se foloseşte pentru adresarea sursei;
  - perechea de regiştrii ES: [-/E] DI sau registrul RDI se foloseşte pentru adresarea destinaţiei;
- Segmentul implicit ES nu poate fi modificat prin specificarea explicită a unui alt registru segment în instrucţiune (de exemplu DS:DI), dar DS poate fi schimbat.

- Şirul poate fi stocat în memorie:

- în ***sens crescător*** (de la adrese mai mici -> adrese mai mari) dacă **DF=0**, mod auto-incrementare
- în ***sens descrescător*** (de la adrese mai mari -> adrese mai mici) dacă **DF=1**, mod auto-decrementare;

- La parcurgerea şirului în sens crescător
  => (-/E/R)SI sau/ şi (-/E/R)DI vor fi actualizaţi prin incrementare,
  iar la parcurgerea şirului în sens descrescător
  => (-/E/R)SI sau/ şi (-/E/R)DI vor fi actualizaţi prin decrementare;

- Numărul de octeţi cu care se incrementează/ decrementează regiştrii (-/E/)SI, resp. RSI şi (-/E)DI resp. RDI este dat de dimensiunea elementelor şirului: ***d=1*** pt *octeţi*, ***d=2*** pt *cuvinte*, ***d=4*** pt *dublucuvinte* şi ***d=8*** pt *cvadublucuvinte*;

- Când se operează în modul pe 32 biţi, de la **80386↑**, regiștrii EDI și ESI se folosesc în locul regiștrilor DI și SI (registrul acumulator va fi în acest caz EAX în loc de AX), permiţând folosirea oricărei locaţii de memorie în spaţiul de memorie protejat de 4GB al microprocesorului.
- Când se operează în modul pe 64 biți, de la **Pentium 4** și **Core 2↑**, regiștrii RDI și RSI se folosesc în locul regiștrilor (E)DI și (E)SI (registrul ACC va fi în acest caz RAX în loc de (E)AX). În acest caz, nu mai există noţiunea de segment ca în modurile de operare inferioare (cele pe 16 biți sau pe 32 biți), deci DS şi ES îşi pierd rolul avut la segmentare. În acest mod, se pot muta date de tip doubleword, dar adresa va fi specificată tot pe 64 biți, cu RDI şi RSI.

Astfel, în funcție de atributul de dimensiune al instrucțiunii pe 32 sau 64 biți, se poate folosi:

La **8086**:	ES:DI	sau	DS:SI
La **386**:	ES:EDI	sau	DS:ESI
La **P4**:	RDI	sau	RSI

5.1.1. Instrucţiunile MOVS şi MOVS B/W/D/Q

Instrucţiunile de copiere sau transfer ***(MOVe String)*** **MOVS[-B/W/D/Q]** transferă un ***octet***, un ***cuvânt***, un ***dublucuvânt*** sau un ***cvadruplucuvânt*** din şirul sursă (adresat de DS:(E)SI în mod pe 32 biți sau (E/R)SI în mod pe 64 biţi) în şirul destinaţie (adresat de ES:(E)DI în mod pe 32 biți sau RDI în mod pe 64 biţi), transfer urmat de actualizarea adreselor.

mod pe 16biți, mod pe 32 biți	mod pe 64 biți
mov ES:[(E)DI], DS:[(E)SI]	***mov [ES:EDI*** sau ***RDI], [DS:ESI*** sau ***RSI]*** *şi apoi*

(-/E/R) DI = (-/E/R) DI ± d şi
(-/E/R) SI = (-/E/R) SI ± d,

unde operaţia e cu "+" dacă DF=0, respectiv cu "-" dacă DF=1,

iar d = {1-octet, 2-cuvânt, 4-dublucuvânt, 8-cvadruplucuvânt}

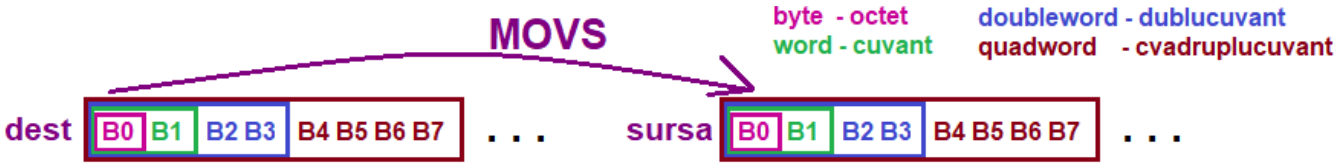


Figura 5-1.1. Ilustrarea modului de operare al instrucţiunii ***MOVS*[-B/W/D/Q]**

Observaţii:

Instrucţiunile MOVS şi MOVS B/W/D/Q nu afectează flagurile.

5.1.2. Instrucţiunile LODS şi LODS B/W/D/Q

Instrucţiunile de încărcare a elementelor din şir ***(LOaD String)*** **LODS[-/B/W/D/Q]** încarcă în **AL, AX, EAX** (la **80386↑**) sau chiar **RAX** (de la **Pentium 4** sau **Core 2↑**) un ***octet***, ***cuvânt***, ***dublucuvânt*** sau ***cvadruplucuvânt*** (cel de la adresa DS:(E)SI în mod pe 32 biți sau (E/R)SI în mod pe 64 biţi), iar apoi se actualizează adresa pentru a accesa următorul element al şirului.

mod pe 16biți, mod pe 32 biți	mod pe 64 biți
mov AL/AX/EAX, [DS:(E)SI]	***mov AL/AX/EAX/RAX, [DS:ESI*** sau ***RSI]*** *şi apoi*
(-/E/R) SI = (-/E/R) SI ± d,	

*unde operaţia e cu "**+**" dacă DF=0, respectiv cu "**-**" dacă DF=1,*

iar d = {1-octet, 2-cuvânt, 4-dublucuvânt, 8-cvadruplucuvânt}

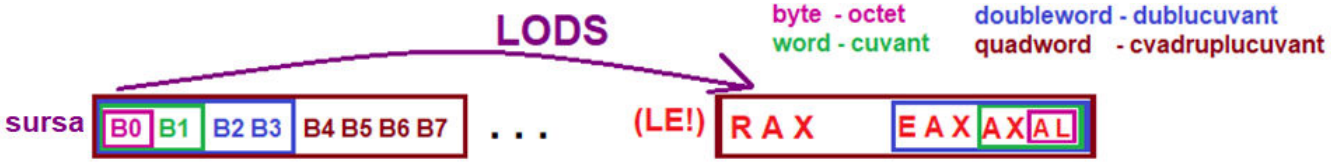


Figura 5-1.2. Ilustrarea modului de operare al instrucţiunii ***LODS*[-B/W/D/Q]**

Observații:

Instrucţiunile LODS şi LODS B/W/D/Q nu afectează flagurile.

5.1.3. Instrucţiunile STOS şi STOS B/W/D/Q

Instrucţiunile de memorare şir ***(STOre String)*** **STOSB/W/D/Q** încarcă valoarea din **AL, AX, EAX** (**la 80386↑**) sau chiar **RAX** (de la **Pentium 4** sau **Core 2↑**) în ***octetul***, ***cuvântul***, ***dublucuvântul*** sau ***cvadruplucuvântul*** de la adresa DS:(E)DI în mod pe 32 biți sau (E/R)DI în mod pe 64 biţi, iar apoi se actualizează adresa.

mod pe 16biți, mod pe 32 biți	mod pe 64 biți
mov [ES:(E)DI], AL/AX/EAX	***mov [ES:EDI*** sau ***RDI], AL/AX/EAX/RAX*** *şi apoi*
(-/E/R) DI = (-/E/R) DI ± d,	

*unde operaţia e cu "**+**" dacă DF=0, respectiv cu "**-**" dacă DF=1,*

iar d = {1-octet, 2-cuvânt, 4-dublucuvânt, 8-cvadruplucuvânt}

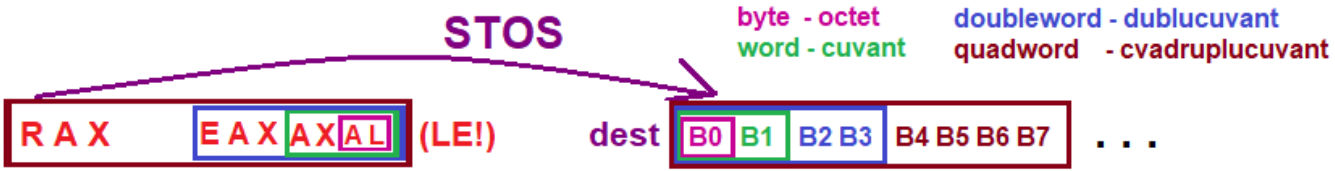


Figura 5-1.3. Ilustrarea modului de operare al instrucţiunii ***STOS*[-B/W/D/Q]**

Observaţii:

Instrucţiunile STOS şi STOS B/W/D/Q nu afectează flagurile.

5.1.4. Instrucţiunile CMPS şi CMPS B/W/D/Q

Instrucţiunile de comparare şir ***(CoMPare String)*** **CMPSB/W/D/Q** testează egalitatea şirurilor prin comparare element cu element. Se execută o scădere *fictivă* între ***octeții***, ***cuvintele***, ***dublucuvintele*** sau ***cvadruplucuvintele*** de la adresele DS:(E)SI şi ES:(E)DI în mod pe 32 biți sau (E/R)SI şi (E/R)DI în mod pe 64 biţi, fără modificarea operanzilor, dar cu poziţionarea tuturor flagurilor.

mod pe 16biți, mod pe 32 biți	mod pe 64 biți
cmp [DS:(E)SI], [ES:(E)DI] *setează (-/E/R)FLAGS şi apoi* ***(-/E/R) DI = (-/E/R) DI ± d,*** ***(-/E/R) SI = (-/E/R) SI ± d,***	***cmp [DS:ESI*** sau ***RSI], [ES:EDI*** sau ***RDI]***

unde operaţia e cu "+" dacă DF=0, respectiv cu "-" dacă DF=1,

iar d = {1-octet, 2-cuvânt, 4-dublucuvânt, 8-cvadruplucuvânt }

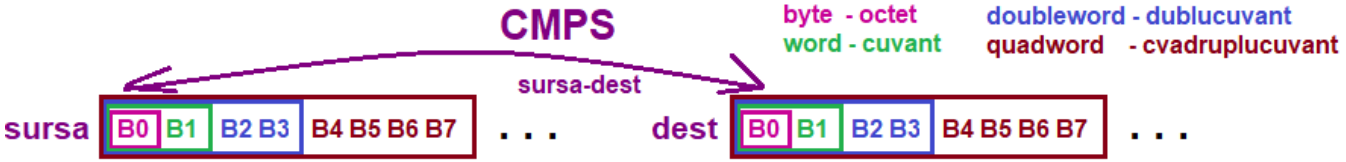


Figura 5-1.4. Ilustrarea modului de operare al instrucţiunii ***CMPS*[-B/W/D/Q]**

Observații:

Instrucţiunile CMPS şi CMPS B/W/D/Q afectează flagurile OF, SF, ZF, AF, PF, CF.

5.1.5. Instrucţiunile SCAS şi SCAS B/W/D/Q

Instrucţiunile de scanare ***(SCAn String)*** **SCASB/W/D/Q** testează/ caută un anumit ***octet***, ***cuvânt***, ***dublucuvânt*** sau ***cvadruplucuvânt*** într-un şir. Se execută diferenţa fictivă dintre **AL, AX, EAX** sau **RAX** şi ***octetul***, ***cuvântul***, ***dublucuvântul*** sau ***cvadruplucuvântul*** de la adresa DS:(E)DI în mod pe 32 biți sau (E/R)DI în mod pe 64 biţi, fără modificarea operanzilor, dar cu poziţionarea tuturor flagurilor.

mod pe 16biți, mod pe 32 biți	mod pe 64 biți
cmp AL/AX/EAX, [ES:(E)DI] *setează (-/E/R)FLAGS şi apoi* ***(-/E/R) DI = (-/E/R) DI ± d,***	***cmp AL/AX/EAX/RAX, [ES:EDI*** sau ***RDI]***

unde operaţia e cu "+" dacă DF=0, respectiv cu "-" dacă DF=1,

iar d = {1-octet, 2-cuvânt, 4-dublucuvânt, 8-cvadruplucuvânt}

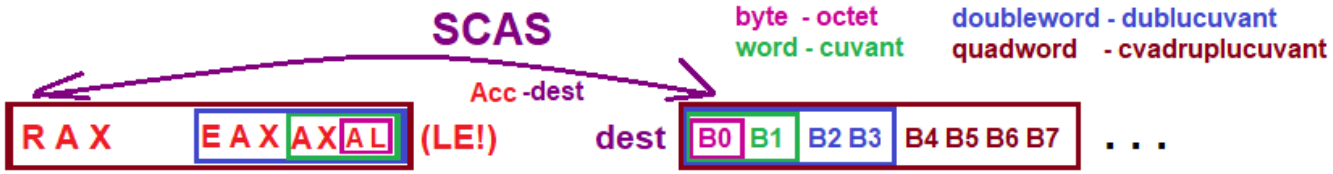


Figura 5-1.5. Ilustrarea modului de operare al instrucţiunii ***SCAS*[-B/W/D/Q]**

Observaţii:

Instrucţiunile SCAS şi SCAS B/W/D/Q afectează flagurile OF, SF, ZF, AF, PF, CF.

5.2. Instrucţiuni cu şiruri pe port

Instrucţiunile **INS** ***(Input String)*** şi **OUTS** ***(Output String)*** (implementate de la procesoarele 80186↑) preiau/ trimit un şir de elemente (octet, word sau doubleword) de la/ la un port de intrare/ieşire.

Instrucţiunile **INSB/W/D** şi **OUTSB/W/D** nu au operanzi şi sunt formele scurte ale instrucţiunilor INS şi OUTS, furnizând implicit dimensiunea datelor de pe port: octet, cuvânt sau dublucuvânt.

Registrul DX în mod implicit este *adresa portului*, iar *destinaţia* implicită este dată de perechea de regiştri ES:(E)DI, respectiv DS:(E)SI.

În modul pe 64 biţi nu există intrare pe 64 biţi, dar adresa memoriei este scrisă pe 64 biţi și specificată în RDI pt instrucţiunea INS, respectiv în RSI pentru instrucţiunea OUTS.

Exemplele prezentate cu şiruri se vor considera la execuţia pe 16 biţi.

Exemple de *instrucţiuni ilegale:*

```
movsb EAX, EBX          ; instrucţiunile pe şiruri au operanzii impliciţi, aceştia nu trebuie precizaţi în instrucţiune
```

Exemple de *instrucţiuni legale*:

Exemplele care urmează au rolul de a fixa anumite noţiuni ale instrucţiuilor pe şiruri şi de a se însuşi deprinderea lucrului cu acestea. In secţiunile următoare aceste exemple vor fi reluate în cadrul unor exemple de programe complete.

Exemple 5-1.1:
În mod pe 16 biți:

```
movsb                   ; ES:[DI]=DS:[SI]; DI=DI±1, SI=SI±1 (transfer la nivel de byte)
movsw                   ; ES:[DI]=DS:[SI]; DI=DI±2, SI=SI±2 (transfer la nivel de word)
```

În mod pe 32 biți: s-a adăugat în plus şi:

```
movsb                   ; ES:[DI]=DS:[SI]; DI=DI±1, SI=SI±1 (transfer la nivel de byte)
movsw                   ; ES:[DI]=DS:[SI]; DI=DI±2, SI=SI±2 (transfer la nivel de word)
movsd                   ; ES:[EDI]=DS:[ESI];EDI=EDI±4,ESI=ESI±4 (similar, doubleword)
```

În mod pe 64 biți:

Adresele pot fi văzute cu ESI şi EDI

```
movsb          ; ES:[EDI]=DS:[ESI]; DI=DI±1, SI=SI±1 (transfer la nivel de byte)
movsw          ; ES:[EDI]=DS:[ESI]; DI=DI±2, SI=SI±2 (transfer la nivel de word)
movsd          ; ES:[EDI]=DS:[ESI];EDI=EDI±4,ESI=ESI±4 (similar, doubleword)
movsq          ; ES:[EDI]=DS:[ESI];EDI=EDI±8,ESI=ESI±8 (transfer la nivel de quadword)
```

Adresele pot fi văzute cu RSI şi RDI

```
movsb          ; [RDI]=[RSI]; RDI=RDI±1, RSI=RSI±1 (transfer la nivel de byte)
movsw          ; [RDI]=[RSI]; RDI=RDI±2, RSI=RSI±2 (transfer la nivel de word)
movsd          ; [RDI]=[RSI]; RDI=RDI±4, RSI=RSI±4 (similar, doubleword)
movsq          ; [RDI]=[RSI]; RDI=RDI±8, RSI=RSI±8 (transfer de quadword)
```

Exemple 5-1.2:

```
stosb          ; ES:[DI]=AL, DI=DI±1
lodsb          ; AL=DS:[SI], SI=SI±1
insb           ; ES:[DI]=[DX], DI=DI±1
outsb          ; [DX]=DS:[SI], SI=SI±1
```

Exemplul 5-1.3 Secvenţa următoare defineşte 2 şiruri: unul sursă şi unul destinaţie, poziţionează regiştrii index pe zonele de început ale şirurilor (vor pointa spre primul element din fiecare şir), iar apoi execută instrucţiunea MOVSB:

```
SIRs DB 1,2,3,4,5 ;se def. şirul destinaţie cu 5 elem. octet, iniţilizat cu 1,2,3,4,5
SIRd DB 5 DUP(0) ;se def şirul sursă cu 5 elemente pe octet, neiniţilizat
...
lea SI, SIRs       ; SI=adresa de început a SIRs
lea DI, SIRd       ; DI=adresa de început a SIRd
cld                ; DF=0, şirurile vor fi parcurse în sens crescător
movsb              ; se mută doar primul elem. din şir şi se poziţionează
                   ; pe cel de-al doillea element din cadrul fiecărui şir
```

Exemplul 5-1.4 Exemplul anterior poate fi rescris fol. instrucţiunile LODSB şi STOSB, iar astfel elementul din şirul sursă este preluat în registrul acumulator, şi abia apoi depus în şirul destinaţie.

```
lodsb                ; AL = element curent din SIRs
stosb                ; din AL se depune elementul curent în SIRd
```

Exemplul 5-1.5 Secvenţa următoare verifică în cadrul unui şir cu 10 elemente definite pe octet dacă primul element este egal cu valoarea din registrul AL, prin utilizarea instrucţiunii SCASB:

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul destinaţie cu 10 elem. pe octet, iniţilizat cu 0,1,2,3,2,4,2,5,2,6
...
mov AL, 2            ; numărul (elementul) de căutat
lea DI, SIRd         ; DI=adresa de început a SIRd
cld                  ; DF=0, şirul va fi parcurs în sens crescător
scasb                ; se verifică egalitatea elem: cel din şir şi cel căutat şi se poziţionează pe următorul elem din şir
```

Exemplul 5-1.6 Exemplul anterior s-ar putea transpune pentru verificarea egalităţii primului element din 2 şiruri prin utilizarea instrucţiunii CMPSB:

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul destinaţie cu 10 elemente pe octet, iniţilizat cu 0,1,2,3,2,4,2,5,2,6
SIRs DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul sursă cu 10 elemente pe octet, iniţilizat cu 0,1,2,3,2,4,2,5,2,6
                ...

lea SI, SIRs         ; SI=adresa de început a SIRs
lea DI, SIRd         ; DI=adresa de început a SIRd
cld                  ; DF=0, şirul va fi parcurs în sens crescător
cmpsb                ; se verifică egalitatea primului element din cele 2 şiruri şi se poziţionează pe următ. elem. din şir
```

Exemplul 5-1.7

```
insb        ; preia un şir de octeţi de pe portul ce are adresa specificată în registrul DX şi îl depune în memorie
            ; la adresa dată de ES:DI, DI=DI±1 din memorie, de la adresa dată de DS:SI trimite
outsw       ; un şir de cuvinte pe portul cu adresa specificată în registrul DX, SI=SI±1
```

Capitolul 6. Instrucţiuni de salt

Locaţia în memorie a următoarei instrucţiuni de executat este dată de perechea de regiștrii 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 așa cum se poate urmări în Tabelul 6-1.1 de:
starea anumitor flaguri
conţinutul registrului CX

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)
```

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)
```

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
32 biţi		ECX=ECX-1 dacă (ECX≠0) atunci execută salt altfel, continuă	ECX=ECX-1 dacă (ECX≠0 şi ZF=1) atunci execută salt altfel, continuă	ECX=ECX-1 dacă (ECX≠0 şi ZF=0) atunci execută salt altfel, continuă
64 biţi		RCX=RCX-1 dacă (RCX≠0) atunci execută salt altfel, continuă	RCX=RCX-1 dacă (RCX≠0 şi ZF=1) atunci execută salt altfel, continuă	RCX=RCX-1 dacă (RCX≠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

Exemplul 6-2.1
Pentru a realiza copierea unui şir (SIRs) de 5 elemente într-un alt şir (SIRd) se poate pleca de la secvenţa de instrucţiuni din Exemplul 6-1.3 (inserate în culoare mai deschisă în exemplul curent) şi se poate adapta pentru repetarea acţiunii la numărul de elementeal şirului.

Pentru aceasta, va trebui la început să poziţionăm regiştrii index pe zonele de început ale şirurilor (să pointeze spre primul element din fiecare şir), iar apoi pentru fiecare element să execute instrucţiunea MOVSB/W/D, să decrementeze numărul de elemente pentru care se repetă algoritmul şi să reia algoritmul până când nr. de elemente pentru care se repetă algoritmul a ajuns la zero. Astfel, vom obţine secvenţa:

```
SIRs DB 1,2,3,4,5 ;se def. şirul destinaţie cu 5 elem. octet, iniţilizat cu 1,2,3,4,5
SIRd DB 5 DUP(0) ;se def şirul sursă cu 5 elemente pe octet, neiniţilizat
...
lea SI, SIRs      ; SI=adr de început a SIRs
lea DI, SIRd      ; DI=adr de început a SIRd

mov CX,5          ; CX=5 numărul de elemente
cld               ; DF=0, şirurile vor fi parcurse în sens crescător

et: movsb         ;eticheta et se fol. pt a asigura revenirea în acest punct
dec cx            ;după execuţia mosvb (mută element şi
                  ; actualizează adresa), CX=CX-1
jnz et            ; dacă încă CX nu a ajuns să fie zero
                  ; (jump if not ZF-> dacă ZF=0, face salt), se reia bucla
```

Exemplul 6-2.2 Exemplul precedent poate fi rescris folosind instrucţiunile LODSB şi STOSB, iar astfel fiecare element al şirului sursă este preluat în registrul acumulator, şi abia apoi depus în şirul destinaţie. Bucla de repetiţie se va rescrie astfel:

```
et: lodsb       ; AL = element curent din SIRs
stosb           ; din AL se depune elementul curent în SIRd
dec cx          ; după execuţia mosvb (mutare element şi actualizare adrese), CX=CX-1
jnz et          ; dacă încă CX nu a ajuns să fie zero, se reia bucla
```

Exemplul 6-2.3 Pentru a realiza căutarea (prin scanare) unui anumit caracter într-un şir de elemente va trebui la început să repetăm raţionamentul descris în primul exempl adaptat la operaţia SCASB: condiţia de repetare nu se mai referă la un anumit număr de elemente, ci la o anumită condiţie (egalitatea a 2 elemente). Astfel, vor interveni flagurile, mai exact ZF care se setează doar în momentul când cei 2 operanzi ce se compară sunt egali. In secvenţa următoare se caută primul caracter ‚2':

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul destinaţie cu 10 elem. pe octet,
lungSIRd equ ($-SIRd)
...
mov AL, 2       ; numărul (elementul) de căutat
lea DI, SIRd    ; DI=adr de început a SIRd
mov CX, lungSIRd ; CX= nr. de elemente, determinat cu operatorul $
cld             ; DF=0, şirul va fi parcurs în sens crescător
et:  scasb      ;eticheta et se fol. pt a asigura revenirea în acest punct
jz gasit        ; se verifică egalitatea elementelor (cel din şir cu cel căutat) şi în funcţie de ZF se face salt
dec CX
jnz et          ; dacă s-a găsit,sare la eticheta gasit, altfel ajunge aici şi face salt la et, atât timp cât CX încă nu a ajuns la
                ; sfârşit, deci reia căutarea pt următorul element din şir
mov DI,0        ; dacă elementul nu există în şir, pun 0 în registrul DI
jmp exit        ; sar la eticheta care duce la sfârşitul programului
gasit:  dec DI  ; din cauză că instrucţ. scasb a actualizat deja adresa pe următorul element al şirului,
                ; trebuie să ne întoarcem o poziţie înapoi
exit:  ...
```

Exemplul 6-2.4 Pentru a testa egalitatea a 2 şiruri vor trebui comparate şirurile element cu element, iar dacă înainte de a ajunge la sfârşitul celor 2 şiruri (se pp. că au acelaşi nr de elem.) se întâlneşte o nepotrivire, concluzionăm că şirurile nu sunt egale (şi în registrul DI se va obţine poziţia primei nepotriviri). Astfel, asemănător cu exemplul precedent, vom obţine:

```
SIRd DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul destinaţie cu 10 elemente pe octet,
lungSIRd equ ($-SIRd)     ; se def. imediat după variabila corespunzătoare
SIRs DB 0,1,2,3,2,4,2,5,2,6 ; se def şirul sursă cu 10 elemente pe octet,
                          ; iniţializat cu 0,1,2,3,2,4,2,5,2,6
...
lea SI, SIRs        ; SI=adr de început a SIRs
lea DI, SIRd        ; DI=adr de început a SIRd
mov CX, lungSIRd          ; CX= numărul de elemente, determinat cu operatorul $
cld                       ; DF=0, şirul va fi parcurs în sens crescător

et:  cmpsb                ;eticheta et se fol. pt a asigura revenirea în acest punct
jnz nepotrivire           ; se verifică egalitatea elementelor din cele 2 şiruri şi
                          ; în funcţie de ZF se face salt
dec CX
jnz et                    ;dacă s-a găsit o inegalitate, sare la eticheta
                          ; nepotrivire, altfel ajunge aici şi face salt la et,
                          ; doar dacă încă nu s-a ajuns la sfârşitul şirurilor;
potrivire: call AfisMesaj ; dacă s-a ajuns aici, înseamnă că s-au parcurs şirurile până la sfârşit şi nu s-a găsit nici o nepotrivire,
                          ; deci şirurile sunt egale, astfel că se afişează mesajul corespunzător
nepotrivire:  dec DI      ; deoarece instrucţ. scasb a actualizat deja adresa pe următ. elem al şirului,
                          ; trebuie să ne întoarcem o poziţie
```

Exemplul 6-2.5 Secvenţa de buclare din exemplul 6-2.3:

```
et: movsb           ; eticheta et se fol. pt a asigura revenirea în acest punct
dec CX              ; după execuţia mosvb (mută element şi
                    ; actualizează adresa), CX=CX-1
jnz et              ; dacă încă CX nu a ajuns să fie zero
                    ; (jump if not ZF-> dacă ZF=0, face salt),
                    ; se reia bucla
;se poate înlocui cu:
et: movsb
loop et             ; reg. CX e automat decrementat şi verificat prin folosirea loop
```

Exemplul 6-2.6 Secvenţa de buclare poate fi rescrisă:

```
et:  scasb          ; eticheta et se fol. pt a asigura revenirea în acest punct
jz gasit            ; se verifică egalitatea elementelor (cel din şir cu cel
                    ; căutat) şi în funcţie de ZF se face salt
dec CX
jnz et              ; dacă s-a găsit,sare la eticheta gasit, altfel ajunge aici
                    ; şi face salt la et, atât timp cât CX încă nu a  ajuns la
                    ; sfârşit, deci reia căutarea pt următ. element din şir
;se poate înlocui cu:
et: scasb
je gasit
loop et             ; reg.CX e automat decrementat şi verificat prin folosirea loop
```

6.3. Prefixe de repetare

Prefixele de repetare se folosesc pentru execuţia repetată a unor operaţii primitive cu şiruri, în funcţie de: (1) un contor sau (2) un contor şi o condiţie logică. Aceste prefixe nu sunt instrucţiuni în sine, ci participă la formarea de instrucţiuni compuse, alături de operaţiile primitive descrise anterior.

REP/ REPE/ REPZ ***(Repeat while equal/zero)***

- operaţia primitivă se execută până când:
  - CX devine 0 sau
  - apare o nepotrivire (ZF=0 pentru instrucţiunile SCAS sau CMPS).

REP/REPE/REPZ operaţie_primitivă;

REPNE/ REPNZ ***(Repeat while not equal/zero)***

- operaţia primitivă se execută până când:
  - CX devine 0 sau
  - apare o potrivire (ZF=1 pentru instrucţiunile SCAS sau CMPS).

REPNE/REPNZ operaţie_primitivă;

Exemplul 6-3.1 Secvenţa de buclare din exemplul 6.2-1:

```
et: movsb       ;eticheta et se fol. pt a asigura revenirea în acest punct
dec cx          ; după execuţia mosvb (mută element şi actualizează adrese), CX=CX-1
jnz et          ; dacă încă CX nu a ajuns să fie zero (jump if not ZF-> dacă ZF=0, face salt), se reia bucla
;se poate înlocui cu:
rep movsb       ; registrul CX este automat decrementat şi verificat prin folosirea prefixului rep.
```

Exemplul 6-3.2 Secvenţa de buclare din exemplul 6-2.3:

```
et:  scasb      ;eticheta et se fol. pt a asigura revenirea în acest punct
jz gasit        ; se verifică egalitatea elementelor (cel din şir cu cel căutat) şi în funcţie de ZF se face salt
dec CX
jnz et          ; dacă s-a găsit,sare la eticheta gasit, altfel ajunge aici şi face salt la et, atât timp cât CX încă nu a  ajuns la
                ; sfârşit, deci reia căutarea pt următ. element din şir
mov DI,0        ; dacă elementul nu există în şir, pun 0 în registrul DI
jmp exit        ; sar la eticheta care duce la sfârşitul programului
;se poate înlocui cu:
repnz scasb     ; prefixul repnz face ca instrucţ. scasb să se repete până când CX=0 sau până găseşte elementul căutat
jcxz exit       ; dacă CX a ajuns în 0, am terminat şi nu s-a găsit
```

6.4. Instrucţiuni pentru control indicatori (flags)

Instrucţiunile pentru controlul indicatorilor sau flagurilor – se folosesc în general atunci când programatorul doreşte să modifice / controleze modul de execuţie al unor instrucţiuni care exploatează acele flaguri (de exemplu direcţia de parcurgere a elementelor octet/ cuvânt, etc la instrucţiunile pe şiruri).

Instrucţiune	*Efect*	*Instrucţiune*	*Efect*
CLC	CF=0	**STC**	CF=1
CMC	CF=/CF		
CLD	DF=0	**STD**	DF=1
CLI	IF=0	**STI**	IF=1
CLAC	ACF=0 (EFLAGS|b18)	**STAC**	ACF=1 (EFLAGS|b18)

7. Alte instrucţiuni

Se propune studiul individual (din [1]) al următoarelor instrucţiuni:

CALL – pentru apelul procedurilor
RET – pentru revenirea din procedură
INT n – pentru apelul (execuţia) întreruperilor software
INTO – pentru apel excepţie overflow
IRET – pentru revenire din întrerupere
HLT – pentru a opri execuţia instrucţiunii curente (va intra în starea „halt")
LOCK – utilă în mediu multiprocesor pt a asigura acces exclusiv la memoria partajată
WAIT – verifică dacă sunt excepţii FP în aşteptare
CPUID – identifică procesorul şî caracteristicile lui

Alte instrucţiuni pentru controlul procesorului:

INVD
INVLPG
RDTSCMP
RDMSR
WRMSR

Bibliografie:

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 1: Basic Architecture

[3] A. Apătean, "Aspecte de bază în programarea în limbaj de asamblare folosind simulator de microprocesor 8086", Ed. UTPress, Cluj-Napoca, 2016, carte online.

[4] E. Lupu, S. Emerich, A. Apătean, "Iniţiere în limbajul de asamblare x86", Ed. Galaxia Gutenberg, Târgu Lăpuş, 2012.

[5] G. Toderean, A. Căruntu, O. Buza, A.Nica, "Sisteme cu Microprocesoare- Îndrumător de laborator", Ed. Risoprint, Cluj-Napoca, 2007.

[6] Al. Vancea, F. Boian, D. Bufnea, A. Gog, A. Darabant, A. Sabău, "Arhitectura calculatoarelor. Limbajul de asamblare 80x86", Ed.Risoprint, Cluj Napoca, 2005.

[7] A. Gog, A. Sabău, D. Bufnea, A. Sterca, A. Darabant, Al. Vancea, "Programarea în limbaj de asamblare 80x86. Exemple şi aplicaţii", Ed.Risoprint, Cluj Napoca, 2005.

[8] V. Lungu, "Procesoare Intel. Programare în limbaj de asamblare", ed.Teora, 2004.

[9] Gh. Muscă, "Programare în limbaj de asamblare", Ed. Teora, Bucureşti, 1998.

[10] Assembly Language - Norton Guide, http://www.ousob.com/ng/asm/

[11] Introduction to x86 Assembly Language, http://www.c-jump.com/CIS77/ASM/Assembly/ lecture.html

[12] Randal Hyde, Art of assembly, 32 bits edition