12. Setul extins de instrucţiuni x86

12.1 Generalităţi

Evoluţia arhitecturii microprocesoarelor INTEL de la 16 la 32 de biţi implică anumite modificări ale dimensiunii regiştrilor precum şi adăugarea unor resurse arhitecturale suplimentare.
În figura următoare este prezentată arhitectura simplificată a unui procesor Pentium. Procesorul Pentium are o arhitectură *superscalară* ceea ce îi permite în anumite condiţii să execute două instrucţiuni în acelaşi timp prin cele două pipeline-uri de procesare paralelă a datelor.

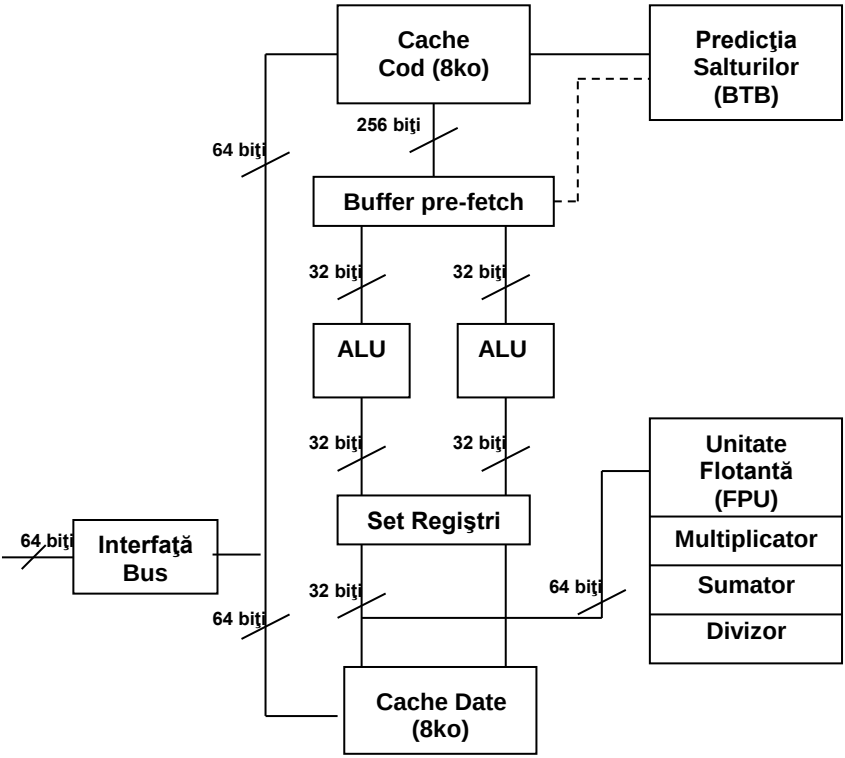

Fig. 12.1. Arhitectura simplificată a procesorului Pentium

Principalele elemente arhitecturale ale schemei sunt: regiştrii, unităţile aritmetice şi logice, memoria cache de date şi cod (8Ko+8Ko), unitatea în virgulă flotantă, blocul pentru predicţia salturilor şi interfaţa cu magistralele. *Setul de regiştri* este cel descris în tabelul 12.1.

Tip regiştri	Biţi	Denumire
Generali	32	EAX, EBX, ECX, EDX
	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	32	ESP, EBP
	16	SP, BP
Index	32	ESI, EDI
	16	SI, DI
Segment	16	CS, DS, SS, ES, FS, GS
Stare şi control	32	EIP, EF, CR0…CR4, DR0…DR7
Alţi regiştri	16	IP, F, MSW, GDTR, LDRT, IDTR, TR

Tabelul 12.1. Setul de regiştri al procesorului Pentium

386/486/Pentium — regiştri pe 32 de biţi

Regiştri generali si de adresă:
EAX — acumulatorul implicit
EBX — conţine implicit o adresă de bază pentru anumite moduri de adresare
ECX — contor implicit
EDX — acumulator extins implicit sau registru de date
ESI — index pentru sursă
EDI — index pentru destinaţie
EBP — indicatorul bazei în stivă
ESP — indicatorul curent în stivă

Regiştri de stare şi control:
EIP — indicator (numărător) de instrucţiuni
EF — registrul de flaguri
CR0…CR4 — regiştri de control
DR0…DR7 — regiştri pentru depanare (debug)

Regiştri segment:
FS — registru segment suplimentar (de date noi)
GS — registru segment suplimentar (de date noi)

Cele două pipeline-uri standard de procesare a instrucţiunilor dispun şi de două unităţi de calcul **ALU** întregi.
Memoriile **cache** sunt separate: pentru date (8Ko) şi pentru instrucţiuni (8Ko). Fiecare memorie cache are câte un modul TLB (Translation Lookaside Buffer) dedicat, care converteşte adresele logice succesive în adrese fizice. Conţine de asemenea şi un coprocesor matematic încorporat **FPU** *(Floating Point Unit).* Se estimează că unitatea de calcul în virgulă mobilă FPU a procesorului Pentium este de 2-10 ori mai rapidă decât cea a procesorului 486.

Modulul numit **BTB** *(Branch Target Buffer)* utilizează ca tehnică predicţia salturilor (*branch prediction*) în scopul reducerii timpului de aşteptare în canalele de procesare. La fiecare salt microprocesorul stochează adresa instrucţiunii de salt şi adresa destinaţiei saltului. BTB încearcă să prevadă apariţia unei instrucţiuni de salt şi să extragă din memorie instrucţiunile corespunzătoare ramurii la care se va face saltul. Salturile anticipate corect nu introduc întârzieri în prelucrare.

Procesorul are o magistrală de adrese pe 32 de biţi şi poate astfel să adreseze 4Go de memorie ca şi procesoarele 386DX şi 486; procesorul Pentium extinde magistrala de date la 64 de biţi, ceea ce înseamnă că poate transfera sistemului de două ori mai multe informaţii, la aceeaşi frecvenţă de ceas. În interior însă procesorul Pentium are regiştri de 32 de biţi care sunt compatibili cu cei ai procesoarelor anterioare.

12.2 Setul extins de instrucţiuni

Utilizarea setului extins de instrucţiuni în aplicaţii trebuie anunţată asamblorului folosind directivele:
.8086 .8087 .186 .286 .287 ;
.286P .386 .387 .386P .486;
.486P .586 .586P
Aceste directive nu acceptă operanzi. Directivele procesor permit utilizarea tuturor instruncţiunilor pe un procesor dat. Directivele .8087, .287, .387 activează setul de instrucţiuni în virgulă flotantă. Scopul acestor directive este să permită instrucţiuni 80287 cu setul 8086 sau 80186, sau instrucţiuni

80387 cu setul 8086, 80186 sau 80286. Directivele ce se termină cu P permit asamblarea unor instrucţiuni pentru privilegii. Acestea sunt utile celor care scriu sisteme de operare, drivere pentru anumite dispozitive şi alte rutine de sistem.

12.2.1 Instrucţiuni de transfer

MOVSX (Move with Sign Extension) – copiază un operand pe 8 biţi la o destinaţie pe 16 sau 32 de biţi, sau un operand pe 16 biţi la o destinaţie pe 32 de biţi prin extensia corespunzătoare a semnului operandului sursă. Flagurile nu se modifică.

MOVSX destinaţie, sursă *destinaţie ← sign_extend(sursă)*

Operanzi: reg,reg
reg,mem

Exemplu:

```
MOVSX       EAX, AL              ;octet → dublucuvânt
MOVSX       EDI, WORD PTR [ESI]  ;cuvânt → dublucuvânt
```

MOVZX (Move with Zero Extension) – copiază un operand pe 8 biţi la o destinaţie pe 16 sau 32 de biţi, sau un operand pe 16 biţi la o destinaţie pe 32 de biţi precum şi extensia cu zero a sursei. Extensia se realizează prin "umplerea" cu zero a biţilor superiori ai destinaţiei. Flagurile nu se modifică.

MOVZX destinaţie, sursă *;destinaţie ← sursă*

Operanzi: reg,reg
reg,mem

Exemplu:

```
MOVZX       EAX, AL              ;octet→ dublucuvânt
```

PUSHFD (Push EFLAGS Registers) – copiază registrul de flaguri **EF** în stivă. Instrucţiunea nu are operanzi şi nu afectează nici un flag.

PUSHFD *; ESP ← ESP – 4*
[SS:ESP] ← EF

POPFD (Pop Stack into EFLAGS) – aduce vârful stivei în registrul de flaguri **EF.** Instrucţiunea nu are operanzi, iar flagurile se schimbă în conformitate cu rezultatul operaţiunii.

POPFD *; EF ← [SS:ESP]*
ESP ← ESP + 4

PUSHF (Push 16-bit EFLAGS Registers) – copiază cei mai puţin semnificativi 16 biţi ai registrului de flaguri în stivă. Instrucţiunea nu are

operanzi şi nu afectează nici un flag. Asigură compatibilitatea cu procesoarele pe 16 biţi şi poate produce decalarea stivei.

```
PUSHF                    ;ESP ← ESP – 2
                         [SS:ESP] ← EFlow
```

POPF (Pop Stack into FLAGS) – aduce din stivă cei mai puţin semnificativi 16 biţi ai registrului **EF.** Instrucţiunea nu are operanzi, iar flagurile se schimbă în mod corespunzător.

```
POPFD                    ;EFlow ← [SS:ESP]
                         ESP ← ESP + 2
```

PUSHAD (Push 32-bit General Registers) – copiază toţi cei 8 regiştri pe 32 de biţi, pe stivă. Valoarea lui ESP salvată pe stivă este cea dinaintea execuţiei instrucţiunii PUSHAD. Registrul de flaguri nu se modifică.

```
temp ← ESP
PUSH   EAX
PUSH   ECX
PUSH   EDX
PUSH   EBX
PUSH   temp
PUSH   EBP
PUSH   ESI
PUSH   EDI
```

POPAD (Pop All General Registers) – reface din stivă toţi regiştrii generali pe 32 de biţi cu excepţia lui ESP. Registrul de flaguri nu se modifică.

```
POP    EDI
POP    ESI
POP    EBP
ESP ←  ESP + 4
POP    EBX
POP    EDX
POP    ECX
POP    EAX
```

OBS: Instrucţiunea **PUSHA/ POPA** copiază/ reface toţi cei 8 regiştri de 16 biţi, pe/din stivă.

SAHF (Store AH în EFLAGS) – încarcă conţinutul registrului AH în octetul cel mai puţin semnificativ al registrului EF, cu mascarea biţilor rezervaţi (7,6,4,2 şi 0).

```
EF ← EF or  (AH and 0D5H)
```

Lseg (Load Segment Register) – adresa sursă specifică un pointer pe 48 de biţi conţinând o adresă efectivă pe 32 de biţi urmată de un selector pe 16 biţi. Adresa efectivă este încărcată în registrul destinaţie, iar selectorul în registrul segment specificat prin mnemonica instrucţiunii. Flagurile nu sunt afectate.

LDS	*reg,mem*	*;destinaţie ← [sursă]*
LES	*reg,mem*	*;seg ← [sursă+4]*
LFS	*reg,mem*	
LGS	*reg,mem*	
LSS	*reg,mem*	

BSWAP (Byte Swap) – această instrucţiune se utilizează atunci când se doreşte un schimb de date între procesoare cu arhitecturi diferite. Se realizează conversia între formatele "big-endian" şi "little-endian" schimbând ordinea celor 4 octeţi care compun data conţinută de registrul precizat. (Daca ar fi posibilă folosirea unui operand pe 16 biţi, atunci de exemplu bswap AX ar fi echivalentă cu xchg AH,AL).

BSWAP reg32	*;temp*	*← reg*
	;reg[0…7]	*←* temp[*24…31]*
	;reg[8…15]	*←* temp[*16…24]*
	;reg[16…23]	*←* temp[*8…15]*
	;reg[24…31]	*←* temp[*0…7]*

Exemplu:

```
MOV EAX, 12345678h     ; EAX=12345678h
BSWAP EAX              ; EAX=78563412h
```

12.2.2 Instrucţiuni de I/O pentru şiruri

INSB, INSW, INSD (Input String from I/O Port) – destinaţia dată de (ES:EDI) sau (ES:DI) primeşte date de la portul de intrare, specificat de către DX. Registrul DI este incrementat dacă flagul DF=0 sau decrementat dacă flagul DF=1 cu dimensiunea operandului (d=1, 2 sau 4). Pot fi folosite împreună cu prefixul REP, iar în acest caz registrul ECX va fi registrul contor. Nu au operanzi.

INSB, INSW, INSD	*(ES:EDI / DI) ← port (DX)*	
	şi (EDI/ DI) = (EDI/ DI) ± (d)	*;d=1 / 2 / 4*

OUTSB, OUTSW, OUTSD (Output String) – se transferă un octet, cuvânt sau un dublucuvânt de la adresa efectivă dată de ESI sau SI la portul specificat prin registrul DX.

OUTSB, OUTSW, OUTSD	*port(DX) ← (ES:ESI/ SI)*	
	şi (ESI/ SI) = (ESI/ SI) ± (d)	*;d=1 / 2 / 4*

12.2.3 Instrucţiuni aritmetice

CDQ (Convert Double Word to Quadword) – converteşte datele pe 32 de biţi aflate în registrul EAX la 64 biţi. Nu este afectat nici un flag. Se utilizează deseori înaintea instrucţiunilor de împărţire când deîmpărţitul este de 64 de biţi.

```
CDQ             ;dacă EAX31 = 1 → EDX = 0FFFFFFFFh
                ;altfel EDX= 0
```

Exemplu:

```
mov EAX, 12345678h     ;EAX=12345678h
cdq                    ;EDX=0h
```

Exemplu:

```
MOV   EAX, [420H]        ;copiază deîmpărţitul în EAX
CDQ                      ;extensie la 64 biţi
IDIV  DWORD  PTR [30H]   ;împărţire cu semn
```

Instrucţiunea **CWDE (Convert Word to Doubleword Extended)** recunoscută de procesoarele de la 80386 şi ulterioare, converteşte cuvântul din AX la dublu-cuvântul din EAX, deci bitul de semn din AX se extinde la tot registrul EAX.

```
CWDE            ; dacă AX15 = 1 → EAX = 0FFFFFFFFh,
                ; altfel EAX= 0
```

Exemplu:

```
mov AX, 0FFA5h         ;AX=0FFA5h
cwde                   ;EAX=0FFFFFFA5h
```

CMPXCHG (Compare and Exchange) – valoarea primului operand este comparată cu valoarea acumulatorului (AX, EAX). Dacă valorile sunt egale, adică ZF=1 valoarea celui de-al doilea operand este copiată în primul. În caz contrar primul operand este copiat în acumulator.

```
CMPXCHG   operand1, operand2     ;dacă   ACC = operand1 atunci
                                 ;ZF=1     operand1 = operand2
                                 ;altfel  ZF=0    ACC = operand1
```

Operanzi: *reg,reg*
mem,reg

Exemplu:

```
cmpxchg [VAR], SI     ;compară AX cu cuvântul din memorie
                       ;adresat de VAR; dacă sunt egale
                       ;=>ZF=1 şi în locaţia adresată de
                       ;VAR se depune conţinutul lui SI
                       ;dacă sunt diferite =>ZF=0 şi în
```

```
                                ;AX se depune conţinutul memoriei
                                ;dat de VAR
```

O instrucţiune asemănătoare cu CMPXCHG este **CMPXCHG8B** care compară cei 8 octeţi ai perechii EDX:EAX cu o zonă de memorie mem64. 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 in perechea EDX:EAX.

```
CMPXCHG8B  destinaţie      ;dacă EDX:EAX= destinaţie,
                           ;atunci Z=1 şi (destinaţie)=ECX:EBX
                           ;altfel Z=0 şi EDX:EAX =(destinaţie)
```

XADD (Exchange and ADD) – se calculează suma dintre operanzi, iar suma se depune în destinaţie. Valoarea originală a destinaţiei este stocată în sursă.

XADD *destinaţie, sursă* *; destinaţie = destinaţie + sursă*

Operanzi: *reg,reg*
mem,reg

Exemplu:

```
mov EAX,4
mov EBX,6
xadd EAX,EBX   ; EAX=EAX+EBX=> EAX=0000000Ah şi
               ; EBX=00000004h - doar de la 80486↑
```

MUL (*Unsigned* Multiplication) – există un singur operand, înmulţitorul. Se realizează înmulţirea fără semn a numerelor întregi.

În cazul unui operand pe 32 biţi avem:
- deînmulţit: EAX
- produs: EDX | EAX

MUL *sursă* *;acc extins ← acc * sursă*

Operanzi: *reg*
mem

IMUL (*Integer* (*Signed*) Multiplication) – se realizează înmulţirea unor întregi cu semn. Dacă avem un singur operand pe 32 de biţi rezultatul este depus în EDX|EAX. Dacă avem doi sau trei operanzi aceştia trebuie să aibă aceeaşi dimensiune.

```
IMUL    op1
IMUL    op1,op2
IMUL    op1,op2,op3
```

Operanzi:	op1	op2	op3	
	reg			acc ← *acc * reg*
	mem			acc ← *acc * mem*
	reg,	reg		op1 ← *op1*op2*
	reg,	mem		op1 ← *op1*op2*
	reg,	imed		op1 ← *op1*op2*
	reg,	reg,	imed	op1 ← *op2*op3*
	reg,	mem,	imed	op1 ← *op2*op3*

Exemplu:

```
imul CX,DX,10  ;înmulţeşte conţinutul reg. DX cu 10 şi
               ;depunde rezultatul în registrul CX – de
               ;la 80286↑
```

Exemplu:

```
imul EAX,[2]   ;înmulţeşte conţinutul reg. EAX cu
               ;dublucuvântul din locaţia de memorie
               ;(segm DS) care începe la adresa 2,
               ;depunând rezultatul în EAX –de la 80386↑
```

Exemplu:

```
imul ESI,EDI,10        ;înmulţeşte conţ. reg. EDI cu 10
                       ;şi depunde rezultatul în reg.
                       ;ESI – de la 80386↑
```

DIV şi IDIV (Integer (Signed) Division) – valoarea aflată în acumulatorul extins este împărţită la *operand.* Câtul rezultat se stochează în partea mai puţin semnificativă a acumulatorului iar restul în partea cea mai semnificativă a acestuia. Dacă câtul e mai mare decât dimensiunea acumulatorului sau avem împărţire cu zero se produce INT 0.

În cazul unui operand pe 32 biţi avem:
- deîmpărţit: EDX│EAX
- cât: EAX
- rest: EDX

IDIV *operand*

Operanzi: reg
mem

Exemplu:

```
MOV     EAX, [ESP + 16]        ;deîmpărţitul
CDQ                            ;se converteşte la 64 biţi
IDIV    ECX                    ;apoi împarte cu semn
                               ;conţ. perechii EDX:EAX la
                               ;ECX şi depune câtul în
                               ;EAX iar restul în EDX
```

12.2.4 Operaţii logice

BSF (Bit Scan Forward) – se parcurge operandul sursă începând cu bitul cel mai puţin semnificativ şi se scrie poziţia primului bit nenul în destinaţie. Dacă operandul sursă este nul, ZF = 1, iar destinaţia este nedefinită.

BSF destinaţie, sursă

dacă (sursa = 0) atunci
ZF = 1 şi destinaţia ←*???*
altfel ZF ← *0*
temp ← *0*
while (bit(sursa, temp) =0)
temp ← *temp+1*
destinaţie ← *temp*

Operanzi: *reg,reg*
reg,mem

Exemplu:

```
MOV     AX, 85C0H
BSF     BX, AX                  ;(BX) ← 6
```

BSR (Bit Scan Reverse) – se parcurge operandul sursă începând cu bitul cel mai semnificativ şi se scrie poziţia primului bit nenul în destinaţie. Dacă operandul sursă este nul, ZF = 1 iar destinaţia este nedefinită.

BSR destinaţie, sursă

dacă (destinaţia este (AX, BX,
CX, DX, SI, DI, BP, SP)) atunci startbit ← 15
altfel
startbit ← 31
dacă (sursa = 0) atunci
ZF = 1 şi destinaţia ← ???
altfel ZF ← 0
temp ← startbit
while (bit (sursa, temp) = 0)
temp ← temp-1
destinaţie ← temp

Operanzi: *reg,reg*
reg,mem

Exemplu:

```
mov AX,1234h            ;AX=1234h=0001 0010 0011 0100b
bsr DX,AX               ;DX=12=0Ch
```

Operaţii cu flaguri

BT (Bit Test) – instrucţiunea testează un singur bit din destinaţie: cel a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată.

BT destinaţie, index ; CF ← destinaţie [index]

Operanzi: reg, data
mem, data
reg,reg
mem,reg

Exemplu:

```
       mov EAX, 23     ;se verifică dacă bitul 23 este setat,
                       ;deci dacă tehnologia MMX este suportată
       mov EBX, 0387F9FFh     ; destinaţia
       bt EBX,EAX             ;EBX=0387F9FFh;
;EBX=0000 0011 1000 0111 1111 1001 1111 1111b cu b23=1,deci CF=1
       jc etSuportaMMX        ;salt la o etich. ce tratează
                              ;cazul în care proc suportă MMX
```

BTC (Bit Test and Complement) – instrucţiunea ajută la testarea unui bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa este apoi inversată.

BTC destinaţie, index ;CF ← destinaţie [index]
*;destinaţie [index] ← **not** destinaţie [index]*

Operanzi: reg, data
mem, data
reg,reg
mem,reg

BTR (Bit Test and Reset) – instrucţiunea testează un bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa devine zero.

BTR destinaţie, index CF ← destinaţie [index]
destinaţie [index] ← 0

Operanzi: reg, data
mem, data
reg,reg
mem,reg

BTS (Bit Test and Set) – instrucţiunea testează un bit din destinaţie a cărui poziţie este precizată prin index. Valoarea acestui bit este încărcată în CF şi ulterior poate fi testată. Valoarea sa devine unu.

BTR	*destinaţie, index*	*CF ← destinaţie [index]* *destinaţie [index] ← 1*

Operanzi: *reg, data*
mem, data
reg,reg
mem,reg

Exemplu:

```
mov EAX, 23    ;se verif. dacă bitul 23 este setat,
mov EBX, 0387F9FFh     ;destinaţia
bts EBX,EAX    ;CF=1,b23 rămâne 1, deci
               ;EBX=038 7F9FFh (neschimbat)
btr EBX,EAX    ;CF=1, b23 devine 0, deci
               ;EBX=030 7F9FFh
btc EBX,EAX    ;CF=0,b23 devine 1 prin complementare
               ;deci EBX=038 7F9FFh (neschimbat)
```

SETcc (Set Byte on Condition) – dacă condiţia este îndeplinită, instrucţiunea setează octetul precizat de destinaţie, iar în caz contrar octetul devine 0.
Condiţia *‚cc'* poate fi: A/ AE/ B/ BE/ C/ E/ G/ GE/ L/ LE/ NA/ NAE/ NB/ NBE/ NC/ NE/ NG/ NGE/ NL/ NLE/ NO/ NP/ NS/ NZ/ O/ P/ PE/ PO/ S/ Z

Exemplu:

```
SETC   destinaţie    ;Set if Carry/ CF=1
SETLE  destinaţie    ;Set if less or equal/ SF≠OF&ZF=1
```

Exemplu:

```
mov AX,2345h
mov BX,0EDCBh
add AX,BX
setnc CL       ; CF=1 -> CL=0
```

Exemplu:

```
mov EAX,2
mov EBX,3
cmp EAX,EBX    ; AX-BX<0
setle CL       ; is less - > CL=1
```

SHLD (Shift Left Double) şi SHRD (Shift Right Double) – *sursa* este concatenată cu *destinaţia* iar valoarea astfel obţinută se deplasează spre stânga respectiv dreapta. Cei mai puţin semnificativi biţi sunt stocaţi în destinaţie. Operandul *count* este mascat cu valoarea 1FH (adică se face şi logic între *count* şi 1FH) astfel că operandul *count* nu depăşeşte valoarea 31.

SHLD *destinaţie, sursă, count*

temp ← max (*count*, 31)
value ← sursă ↑destinaţie
value ← valoare * 2^{temp}
dest ← valoare

Operanzi: reg, reg, imed
mem, reg, imed
reg, reg, CL
mem, reg, CL

SHRD *destinaţie, sursă, count*

temp ← max (*count*, 31)
value ← sursă ↑destinaţie
value ← valoare div 2^{temp}
dest ← valoare

Exemplu:

```
shld AX, BX, 12        ; conţinutul registrului AX este
                       ;deplasat spre stânga cu 12 poziţii şi umplut de
                       ;la dreapta spre stânga cu cei mai semnificativi
                       ;12 biţi ai registrului BX
```

Exemplu:

```
shrd EAX, EBX, 12      ; conţinutul registrului EBX este
                       ;deplasat spre dreapta cu 14 poziţii şi umplut
                       ;de la stânga spre dreapta cu cei mai puţin
                       ;semnificativi 14 biţi ai registrului EAX
```

12.2.5 Alte instrucţiuni

CPUID (CPU Identification) – returnează informaţii de identificare a procesorului sau despre resursele lui. Executarea instrucţiunii pentru diferite valori din EAX dau o imagine completă despre procesor şi posibilităţile sale. Informaţiile returnate la executarea instrucţiunii CPUID sunt: şirul ASCII de identificare a producătorului, semnătura procesorului, numărul de serie al procesorului, flag-urile de caracteristici ale acestuia, informaţii despre memoria cache, etc.

INVD (Invalidate Cache) – invalidează memoria cache internă; instrucţiunea nu are operanzi şi nu modifică nici un flag.

INVLPG (Invalidate TLB Entry) – invalidează intrarea în TLB (Translation Look-aside Buffer) dacă adresa liniară a paginii în care se găseşte adresa *mem* se află în TLB. Registrul de flaguri nu se modifică.

INVLPG mem

dacă adresa_liniară (mem) este în TLB(i)
atunci invalidează TLB(i)

Exemplu:

```
INVLPG     [SI+4]     ;se invalidează PTE pentru această
                      ;adresă
```

MOV (Move Special) – copiază sau încarcă un registru special al CPU în sau dintr-un registru general. Regiştrii speciali sunt CR0, CR2, CR3 (Control Register), respectiv DR0, DR1, DR2, DR3, DR6, DR7 (Debug Register).

MOV destinaţie, sursă; (destinaţie) ← sursă

Operanzi: reg,reg

Exemplu:
```
MOV     EAX, CR0        ;se salvează CR0 în EAX
MOV     DR7, ECX        ;se încarcă DR7 cu conţinutul ECX
```

RDMSR (Read from Model Specific Register) – conţinutul unui registru MSR, precizat de ECX, este copiat în EDX|EAX. Cu ajutorul MSR pot fi observate şi controlate detaliile specifice procesorului.

RDMSR EDX|EAX ← MSR[ECX]

WRMSR (Write to Model Specific Register) – un operand din EDX|EAX este copiat într-un registru MSR

12.3 Exerciţii şi teme

1. Analizaţi exemplele de mai jos şi apoi executaţi-le cu debugger-ul (TD32.exe)

a).
```
INVD                        ;invalidează memoria cache
MOV    EAX, CR0             ;aduce conţinutul registrului de
                            ;control
AND    EAX, 06000000H       ;validează memoria cache
MOV    CR0, EAX             ;rescrie în registrul de control
```

b). Calculul valorii absolute
```
    MOV    EAX, valoare
    OR     EAX, EAX         ;test pentru semn
    JNS    SKIP             ;salt dacă nu există semn
    NEG    EAX              ;negarea numarului dacă e negativ
SKIP:
```

c). Scăderea pe 64 de biţi: EDX|EAX - EBX|ECX
```
    SUB    EAX, ECX         ;se scad biţii de ordin inferior
    SBB    EDX, EBX         ;se scad biţii de ordin superior,
                            ;cu posibilitate de împrumut
```

2. Studiaţi următorul program în care se verifică dacă procesorul suportă tehnologia MMX. Se foloseşte instrucţiunea `CPUID` care returnează informaţii referitoare la procesor: instrucţiunea se execută folosind diferite valori ca intrare în registrul EAX. În cazul parametrului de intrare EAX=1, rezultatul returnat în EDX va conţine flaguri cu informaţii despre procesor. Bitul 23 dă informaţia referitoare la tehnologia MMX: dacă este 1, procesorul supportă MMX.

```
code    segment
        org     100h
        assume  cs:code, ds:code, ss:code, es:code

main:
        jmp     start

        mesaj db "Setul MMX $"
        mesaj_DA db "acceptat.", 13, 10, "$"
        mesaj_NU db "nu este acceptat.", 13, 10, "$"

start:
        ; afişează mesajul de început
        mov ax, seg mesaj
        mov ds, ax
        mov dx, offset mesaj
        mov ah, 09h
        int 21h

.586P   ; selectează set de instrucţiuni 586P

        push    ebx             ; salvează regiştrii
        push    ecx
        push    edx

        ; execută cpuid cu parametrul de intrare eax=1
        mov     eax, 1
        cpuid
        mov     eax,23   ; verific bitul 23
        bt      edx,eax  ; dacă bitul 23 din edx este 1, C=1

        pop     edx      ; reface regiştrii
        pop     ecx
        pop     ebx
.8086                   ; selectează set de instrucţiuni 8086
        jnc nu          ; dacă C=0, nu suportă MMX
        mov     dx, offset mesaj_DA    ; selectez DA
        jmp     afişare

nu:     mov     dx, offset mesaj_NU    ; selectez NU

afisare:
        mov     ah, 09h                ; afişez răspunsul
        int     21h
        mov     ax,4c00h
        int     21h                    ; terminare program
code    ends
        end     main
```

Modificaţi aplicaţia astfel încât să verifice accesul la numărul de serie al procesorului. Această informaţie este dată de bitul 18 din EDX returnat pentru intrare EAX=1, în acelaşi fel: dacă bitul respectiv este 1, avem acces la numărul de serie al procesorului. Testaţi valoarea bitului folosind altă secvenţă decât cea prezentată mai sus.

3. Considerăm următorul şir definit pe octet: 0,1,2,3,4,5,6,7. Scrieţi un program care să realizeze afişarea acestuia în ordinea dată de adresarea bit-reversed. Acest tip de adresare este întâlnit la procesoarele de semnal specializate şi se foloseşte în calculul transformatei Fourier rapide.