

Capitolul 3

Procesoare de 32 de biți

Coprocesoare matematice

3.1 Arhitectura procesoarelor de 32 de biți

3.1.1 Generalități

Apariția și dezvoltarea procesoarelor Intel de 32 de biți a constituit o adevărată revoluție în domeniul calculatoarelor personale, atât prin creșterea performanțelor hardware (în special creșterea spațiului de memorie adresabilă direct) cât și prin perspectivele deschise sistemelor de programe.

Procesoarele de 32 de biți sunt compatibile ca arhitectură cu cele de 16 biți, prin aceea că registrele de 16 biți se regăsesc ca subregistre ale registrelor de 32 de biți. Aceasta a permis ca setul de instrucțiuni al procesoarelor de 16 biți să fie un subset al celui specific procesoarelor de 32 biți, ceea ce face ca orice program dezvoltat pentru procesoarele de 16 biți să poată fi executate și pe mașini de 32 de biți. În plus, toate instrucțiunile care utilizau operanzi de 16 biți se extind și asupra operanzilor de 32 de biți. Această compatibilitate a setului de instrucțiuni a fost unul din scopurile declarate ale noii generații de procesoare, pentru a conserva imensa cantitate de software dezvoltată anterior.

Figura 3.1 descrie registrele procesoarelor de 32 de biți, din punctul de vedere al programelor de aplicație.

Există opt registre generale de 32 de biți, care au numele registrelor generale de 16 biți, cu prefixul E (de la extended). Astfel, registrul EAX se numește acumulator extins, ESP se numește stack pointer extins etc. Primii 16 biți mai puțin semnificativi ai acestor registre sunt disponibili și ca registre de 16 biți (AX, SP etc.), fiind practic identice cu registrele generale 8086.

Aceeași abordare se regăsește și în cazul registrelor EIP (contor program extins) și EFLAGS (registru de flaguri extins). Subsetul de bistabili din registrul FLAGS se regăsește în partea low a registrului EFLAGS.

Registrele de segment au fost păstrate de 16 biți, dar s-au adăugat două noi registre: FS și GS.

Pe lângă registrele generale din Figura 3.1, procesoarele de 32 de biți dispun de registre de control, de gestiune a adresei, de depanare și de test. Acestea diferă de la un tip de procesor la altul și sunt folosite în principal de programele de sistem.

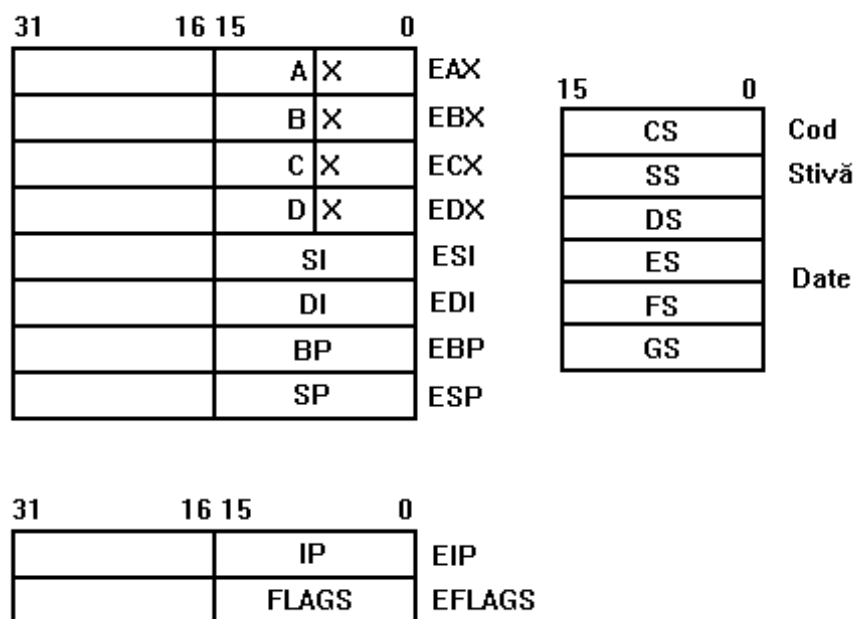


Figura 3.1 Registrele procesoarelor de 32 de biți

Una din modificările majore ale arhitecturii este semnificația registrelor de segment, care acționează în general ca selectoare de segment. Acestea nu mai indică în mod nemijlocit adresa de bază a segmentului de memorie ci un descriptor de segment (încărcat într-un registru special de control) care precizează adresa de bază a segmentului, dimensiunea acestuia și drepturile de acces asociate. Abordarea respectivă permite ca adresa de bază și limita segmentului să poată fi specificate pe 32 de biți, crescând dimensiunea maximă a unui segment până la 4 GB.

3.1.2 Moduri de adresare pe 32 de biți

Modurile de adresare (formarea adresei fizice) au fost mult dezvoltate față de cele pe 16 biți. Se introduc următoarele noțiuni:

- **deplasament** - o valoare imediată pe 8 sau 32 de biți, conținută în instrucțiune;
- **registru de bază** - orice registru general de 32 de biți (spre deosebire adresarea pe 16 biți unde ca registre de bază se utilizează doar BX și BP);
- **registru index** - orice registru general de 32 de biți, cu excepția lui ESP (spre deosebire de adresarea pe 16 biți unde ca registre index se utilizează doar SI și DI);
- **factor de scală** - indexul poate fi înmulțit cu un factor de scală de valoare 1, 2, 4 sau 8 (inexistent în adresarea pe 16 biți).

Se obțin astfel 9 moduri posibile de adresare:

- **adresare directă** - adresa efectivă a operandului face parte din instrucțiune, putând fi pe 8, 16 sau 32 de biți; exemplu:

```
INC dword ptr [1000H]
```

- **adresare indirectă prin registre** - adresa efectivă a operandului este conținută într-unul din registrele de bază; exemplu:

```
MOV [EBX], EAX
```

- **adresare bazată** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se poate adăuga un deplasament; exemplu:

```
ADD ECX, [EAX+32]
```

- **adresare indexată** - adresa efectivă a operandului este formată din conținutul unui registru index la care se poate adăuga un deplasament; exemplu:

```
MUL byte ptr TABLOU [ESI]
```

- **adresare indexată cu factor de scală** - adresa efectivă a operandului este formată din conținutul unui registru index, înmulțit cu un factor de scală, la care se poate adăuga un deplasament; exemplu:

```
MOV EAX, dword ptr TABLOU [EDI*4][100H]
```

- **adresare bazată și indexată** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adună conținutul unui registru index; exemplu:

```
MOV EAX, [ESI][EBX]
```

- **adresare bazată și indexată cu factor de scală** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adugă conținutul unui registru index, înmulțit cu un factor de scală; exemplu:

```
MOV ECX, [EDX*8][EAX]
```

- **adresare bazată și indexată cu deplasament** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adaugă conținutul unui registru index, la care se poate adăuga un deplasament; exemplu:

```
ADD EDX, [ESI] [EBP + 10000H]
```

- **adresare bazată și indexată, cu factor de scală și deplasament** - adresa efectivă a operandului este formată din conținutul unui registru de bază la care se adugă conținutul unui registru index, înmulțit cu un factor de scală, la care se poate adăuga un deplasament; exemplu:

```
MOV EAX, TABLOU [EDI*4] [EBP+800H]
```

Figura 3.2 ilustrează modul cel mai general de adresare (bazată și indexată, cu factor de scală și deplasament), în care adresa efectivă EA se calculează după relația:

$$EA = \text{Registru_Bază} + \text{Registru_Index} * \text{Factor_Scală} + \text{Deplasament}$$

Factorul de scală este foarte util la parcurgerea tablourilor de date simple (pe 2, 4 sau 8 octeți), permițând ca indicele logic al tabloului să coincidă cu conținutul registrului index. De exemplu, o instrucțiune C de forma:

```
long int tablou [1000];
for (i = 0; i < 1000; i++)
    tablou[i] += 10;
```

se implementează prin secvența:

```
MOV    CX, 1000
XOR    EBX, EBX
buc1a:
ADD    _TABLOU [EBX*4], 10
INC    EBX
LOOP   buc1a
```

în care registrul EBX are exact aceeași semnificație ca și indicele i din programul C.

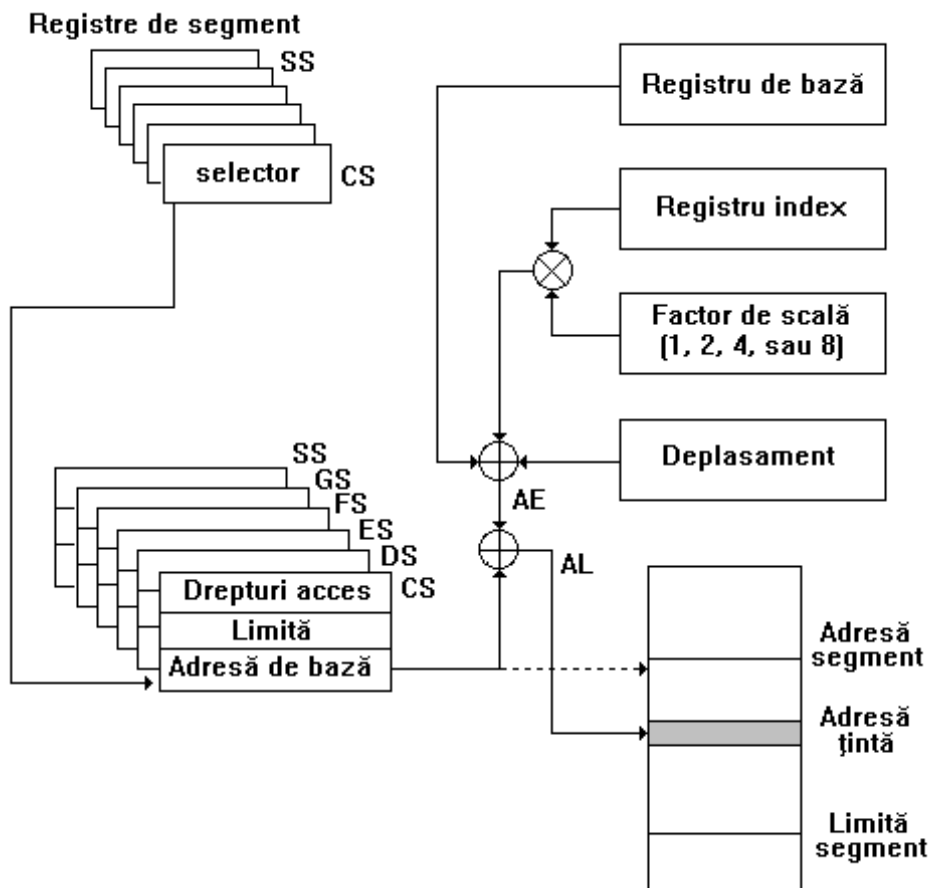


Figura 3.2 Moduri de adresare pe 32 de biți

3.1.3 Modurile Real și Protected

Pentru a păstra compatibilitatea cu programele dezvoltate pentru mașini de 16 biți, procesoarele de 32 de biți dispun de două moduri de operare: modul **Real Address** (pe scurt, modul **Real**) și modul **Protected Virtual Address** (pe scurt, modul **Protected**).

În modul **Real**, procesorul se comportă ca un procesor de 16 biți foarte rapid, permițând însă accesul la date și adrese de 32 de biți.

Modul **Protected** oferă mecanisme sofisticate de gestiune a memoriei (paginare, drepturi de acces la segmente etc.). În acest mod de operare, unitățile de program se numesc taskuri. Se poate executa o comutare de task, pentru a intra într-un regim special de funcționare, denumit mod Virtual 8086. Fiecare asemenea task se comportă (din punct de vedere software) ca o mașină de tip 8086, permițând programelor de 16 biți (aplicații sau chiar un întreg sistem de operare) să poată fi executate ca taskuri.

Atât în modul **Real** cât și în modul **Protected**, procesorul poate executa instrucțiuni de 16 biți, prin examinarea unui bit din descriptorul de segment asociat registrului CS. Acest bit precizează lungimea implicită a operanzilor și adreselor (16 sau 32 de biți).

În afara dimensiunii implicite, se poate forța o anumită dimensiune prin prefixele numite **Operand Size** (dimensiune operand) și **Address Length** (lungime adresă). Aceste prefixe (de valoare 66H și 67H) sunt generate

automat de către macroasamblare. Să presupunem modul Real și să considerăm următoarea secvență de cod sursă:

```
.code
    MOV    AX, word ptr [DI][100H]
    MOV    EAX, dword ptr [DI][100H]
    MOV    AX, word ptr [EDI][100H]
    MOV    EAX, dword ptr [EDI][100H]
```

Fișierul listing generat va conține următoarea descriere:

```
0000 8B 85 0100      MOV AX, word ptr [DI][100H]
0004 66| 8B 85 0100   MOV EAX, dword ptr [DI][100H]
0009 67| 8B 87 00000100 MOV AX, word ptr [EDI][100H]
0010 66| 67| 8B 87 00000100 MOV EAX, dword ptr [EDI][100H]
```

Se observă același cod al instrucțiunii MOV (8BH), același deplasament 100H generat pe 16 sau 32 de biți și prefixele inserate automat. Prima instrucțiune este cu operand și adresare de 16 biți. A doua instrucțiune este cu operand pe 32 de biți și adresare pe 16 biți; se observă prefixul 66H și offsetul 100H generat pe 16 biți. A treia instrucțiune este cu operand pe 16 biți și adresare pe 32 de biți; se observă prefixul 67H și offsetul 100H generat pe 32 de biți. În fine, ultima instrucțiune este cu operand și adresare pe 32 de biți; se observă ambele prefixe și offsetul 100H generat pe 32 de biți.

În modul de operare **Real**, formarea adresei fizice este exact ca la procesoarele de 16 biți (vezi Figura 3.3). Selectorul de segment este unul din cele 4 registre de segment, iar offseturile sunt pe 16 biți, conducând astfel la o lungime maximă a unui segment de 64KB. Pointerii către date sau către instrucțiuni pot fi memorați pe 4 octeți (doi octeți pentru adresa de segment și doi octeți pentru offset).

Asamblarele dispun de directiva DD (Define DoubleWord), care generează pointeri pe 4 octeți, ca în exemplul următor:

```
.data
    TABLOU      DD    256 dup (?)
    ADR_16      DD    TABLOU
.code
    LES    BX, ADR_16
    MOV    EAX, ES:[BX]
```

Instrucțiunea LES BX, ADDR_16 încarcă perechea de regiștri (ES:BX) cu un pointer pe 4 octeți.

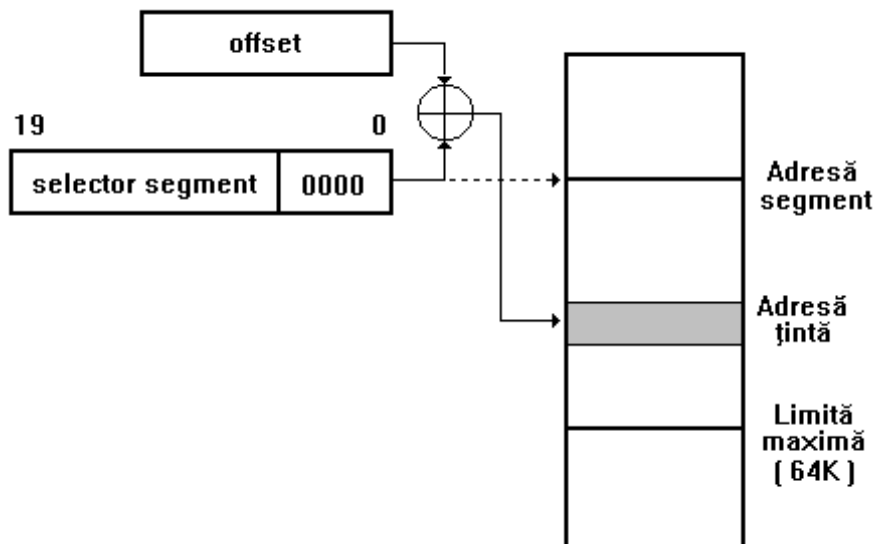


Figura 3.3 Adresarea în modul Real

În modul de operare **Protected**, registrele selectoare de segment (de 16 biți) sunt încărcate cu adresa unui descriptor de segment, iar offseturile pot fi de 16 sau 32 de biți (vezi Figura 3.4), conducând la o lungime maximă a unui segment de 4 GB. Pointerii pot fi pe 4 octeți (doi octeți pentru descriptorul de segment și doi octeți pentru offset) sau pe 6 octeți (patru octeți pentru offset).

Asambleurile dispun de directiva DP (Define Pointer), care generează pointeri pe 6 octeți, ca în exemplul următor:

```
.data
    TABLOU      DD    256 dup (?)
    ADR_32      DP    TABLOU
.code
    LFS    EBX, ADR_32
    MOV    EAX, FS:[EBX]
```

Instrucțiunea `LFS EBX, ADDR_32` încarcă în perechea de registre (FS:EBX) cu un pointer pe 6 octeți.

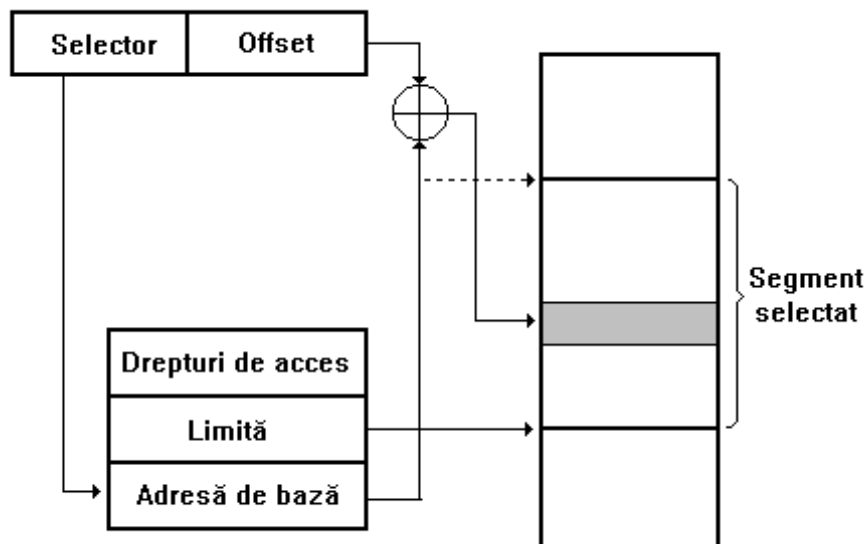


Figura 3.4 Adresarea în modul Protected

Să considerăm următoarea secvență sursă:

```
.data
    TABLOU      DD    256 dup (?)
    ADR_16      DD    TABLOU
    ADR_32      DP    TABLOU

.code
    LEA    BX, TABLOU           ; Acces prin pointer near de 16 biți în
    MOV    EAX, [BX]           ; cadrul segmentului selectat prin DS

    LEA    EBX, TABLOU         ; Acces prin pointer near de 32 de biți în
    MOV    EAX, [EBX]          ; cadrul segmentului selectat prin DS

    LES    AX, ADR_16          ; Acces prin pointer far de 32 de biți în
    MOV    EAX, ES:[BX]        ; cadrul segmentului selectat prin ES

    LFS    EBX, ADR_32         ; Acces prin pointer far de 48 de biți în
    MOV    EAX, FS:[EBX]       ; cadrul segmentului selectat prin FS
```

Listingul generat la asamblare pentru secvența de mai sus este:

```
0000                                .data
0000 0100*(????????)                TABLOU      DD 256 dup (?)
0400 00000000sr                     ADR_16      DD TABLOU
0404 000000000000sr                 ADR_32      DP TABLOU

040A                                .code
0000 BB 0000r                       LEA    BX, TABLOU
0003 66| 8B 07                       MOV    EAX, [BX]

0006 66| 8D 1E 0000r                 LEA    EBX, TABLOU
000B 66| 67| 8B 03                   MOV    EAX, [EBX]

000F C4 1E 0400r                     LES    BX, ADR_16
0013 66| 26: 8B 07                   MOV    EAX, ES:[BX]

0017 66| 0F B4 1E 0404r               LFS    EBX, ADR_32
001D 66| 64: 67| 8B 03               MOV    EAX, FS:[EBX]
```

Se observă prezența a trei categorii de prefixe. Prefixele de operand pe 32 de biți (marcate cu 66|), cele de adresă pe 32 de biți (marcate cu 67|) și prefixele de segment (marcate cu 26: pentru ES: și cu 64: pentru FS:). Instrucțiunea LEA (Load Effective Address) permite încărcarea atât a offseturilor de 16 biți cât și a celor de 32 de biți.

În mod implicit, accesul la date se face prin selectorul DS (la fel ca al 8086), cu excepția modurilor de adresare care implică registrele EBP și ESP, caz în care selectorul implicit este SS. Se poate folosi orice prefix de segment, cu aceleași excepții cunoscute de la 8086:

- instrucțiunile executate sunt accesate totdeauna prin CS;
- operațiile de salvare și restaurare în stivă implicate în instrucțiunile CALL, RET, PUSH și POP se execută totdeauna prin selectorul SS;
- destinația instrucțiunilor pentru șiruri este accesată totdeauna prin selectorul ES.

La punerea sub tensiune, procesorul este în modul **Real**. Trecerea în modul **Protected** se face prin poziționarea unui bit din cuvântul mai puțin semnificativ (numit **Machine Status Word**) al registrului de control CR0.

Așa cum la procesoarele de 16 biți trebuia să încărcăm registrele de segment cu valori cu sens, trecerea în modul **Protected** presupune definirea corespunzătoare a descriptorilor de segment și încărcarea registrelor selectoare. Aceste operații nu vor fi descrise, în ideea că programele de aplicație dezvoltate în limbaj de asamblare vor opera în general în modul **Real**.

3.2 Setul de instrucțiuni al procesoarelor de 32 de biți

3.2.1 Extensia instrucțiunilor de 16 biți

Pentru ca asamblorul să recunoască instrucțiunile specifice, se utilizează directivele .286, .386, .486, care precizează tipul procesorului.

O serie de restricții de la setul de bază 8086 sunt eliminate, începând de la procesorul 80286. De asemenea, se adaugă o serie de instrucțiuni noi.

- încărcarea registrelor de segment se poate face și cu valori imediate; exemplu MOV DS, DGROUP
- Instrucțiunile de deplasare și rotație se pot executa pe un număr oarecare de biți, specificat în instrucțiune; de exemplu: SHL AX, 4

Instrucțiunea de înmulțire cu semn (IMUL) are următoarele forme suplimentare:

- IMUL reg16, im16

prin care se înmulțește registrul de 16 biți reg16 cu o valoare imediată pe 16 biți im16, iar rezultatul se depune în reg16;

- IMUL dreg16, sursa16

prin care se înmulțește dreg16 cu sursa16 (registru sau memorie) și rezultatul se depune în dreg16 (numai la 80386 și următoarele);

- IMUL dreg16, sursa16, im16

prin care se înmulțește sursa16 (registru sau memorie) cu valoarea imediată im16 și rezultatul se depune în registrul dreg16 (destinație).

Instrucțiunile de lucru cu stiva se extind cu:

- **PUSHA** (Push All)

care salvează în stivă registrele AX, CX, DX, BX, SP, BP, SI, DI (în această ordine), iar registrul SP este scăzut cu 16;

- **POPA** (Pop All)

care reface registrele DI, SI, BP, SP, BX, DX, CX, AX (salvate cu o instrucțiune PUSHA anterioară); registrul SP se reface prin adunarea valorii 16, nu prin extragerea efectivă din stivă;

- **PUSH im16**

care pune în stivă o valoare imediată.

Instrucțiunile pentru șiruri se extind cu:

- **INSB, INSW** (Input String)

cu semnificația:

(ES:(DI)) <-- byte/word citit de la portul specificat de DX;
actualizează DI;

- **OUTSB, OUTSW** (Output String)

cu semnificația:

port specificat prin DX <-- (DS:SI);
actualizează SI;

Instrucțiunile de control se extind cu:

- **BOUND reg16, lim** (Testează limite)

în care lim este adresa unei zone de memorie de doi întregi, care precizează limita minimă și maximă pentru registrul reg16. Dacă $\text{reg16} < \text{DS}:(\text{lim})$ sau $\text{reg16} > \text{DS}:(\text{lim}+2)$, atunci se generează o întrerupere software pe nivelul 5.

- **ENTER dim_loc, nivel** (Creează șablon stivă la intrarea în procedură)

în care dim_loc și nivel sunt întregi pe 16 biți fără semn.

Semnificația este următoarea:

```
PUSH BP
temp <-- SP
dacă (nivel > 0) {
    for (i = 1 to nivel-1) {
        BP <-- BP - 2
        PUSH SS:[BP]
    }
    PUSH temp
}
BP <-- temp
SP <-- SP - dim_loc
```

Primul parametru dim_loc este dimensiunea necesară în stivă pentru parametrii locali ai procedurii. Dacă nivel este 0, atunci semnificația este echivalentă cu:

```
PUSH BP
MOV BP, SP
```

```
SUB    SP, dim_loc
```

care este secvența standard de intrare într-o procedură cu parametri locali în stivă (vezi subcapitolul 6.6).

Dacă nivel este > 0 , se copiază în stivă conținutul zonei de stivă (parametri locali) a procedurii apelante. Dacă procedura curentă este apelată dintr-o altă procedură, care are registrul BP poziționat printr-o instrucțiune ENTER, atunci șablonul stivei curente va conține primele nivel-1 cuvinte locale ale procedurii apelante.

- LEAVE (Pregătire pentru ieșire din procedură)

Instrucțiunea LEAVE reface registrele SP și BP, anulând efectul instrucțiunii LEAVE. Este echivalentă cu secvența:

```
MOV SP, BP
POP BP
```

Instrucțiune ENTER se scrie ca primă instrucțiune din procedură, iar LEAVE imediat înainte de RET.

3.2.2 Instrucțiuni specifice procesoarelor de 32 de biți

Întregul set de instrucțiuni de transfer, aritmetice și logice se extinde cu operanzi de tip DoubleWord (registre, memorie sau valori imediate).

Instrucțiuni specifice de transfer

- PUSHAD/POPAD ; Push/Pop All Double

Salvează/restaurează registrele EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI în stivă.

- PUSHFD/POPF ; Push Flags Double

Salvează/restaurează registrul EFLAGS în stivă.

- CMPXCHG reg/mem, reg ; Compare and Exchange

Compară și apoi interschimbă cei doi operanzi, poziționând flagurile aritmetice.

- BSWAP reg32 (Byte Swap)

Inversează ordinea octeților într-un registru de 32 de biți.

- XADD reg/mem, reg ; Exchange and Add

Este echivalentă cu secvența:

```
XCHG reg/mem, reg
ADD reg/mem, reg
```

Instrucțiuni specifice de conversie

- MOVZX/MOVSX destinație, sursă ; Move with Zero/Sign Extension

Copiază operandul sursă (de 8 sau 16 biți) în operandul destinație (de lungime dublă), completând cu 0 (MOVZX) sau cu bitul de semn al operandului sursă (MOVSX). Destinația poate fi registru sau memorie de 16/32 de biți, iar sursa poate fi registru, memorie sau valoare imediată de 8/16 biți.

- CWDE ; Conver Word to DoubleWord Extended

Convertește AX la un dublu cuvânt în EAX, cu extensie de semn. Este asemănătoare cu CWD (Convert Word to Double), care are ca destinație perechea (DX:AX).

- CDQ ; Convert DoubleWord to QuadWord

Convertește registrul EAX la un cuvânt cvadruplu în perechea (EDX:EAX), cu extensie de semn.

Instrucțiuni specifice adreselor

- LFS/LGS/LSS reg, mem

Încarcă o pereche de registre cu un pointer din memorie, pe 4 sau 6 octeți. Registrul specificat poate fi de 16 sau 32 de biți.

Instrucțiuni aritmetice

Operațiile de înmulțire și împărțire se pot executa pe 32 de biți. Se utilizează registrele EAX și EDX, cu semnificație asemănătoare ca la operațiile de 16 biți. De exemplu, instrucțiunea:

DIV EBX

împarte (EDX:EAX) la EBX, cu câtul în EAX și restul în EDX.

Instrucțiuni de deplasare

- SHLD/SHRD reg/mem, reg, CL/im8 ; Shift Left/Right Double

Se deplasează reg/mem la stânga/dreapta cu numărul de biți specificat în CL sau în operandul imediat im8 (pe 8 biți). Al doilea operand este folosit pentru a completa biții deplasați ai primului operand. Primii doi operanzi trebuie să fie de aceeași dimensiune (Word sau DoubleWord).

Instrucțiuni cu pentru șiruri

Setul de instrucțiuni pentru șiruri se extinde cu operații specifice șirurilor de întregi pe 4 octeți. Există astfel instrucțiunile MOVSD (Move String Double), STOSD (Store String Double), LODSD (Load String Double), SCASD (Scan String Double). Se utilizează registrul acumulator extins EAX, iar adresele sursă și destinație (SI și DI) sunt incrementate/descrimentate cu 4.

Instrucțiuni la nivel de bit

- BSF/BSR reg, reg/mem ; Bit Scan Forward/Reverse

Instrucțiunea parcurge operandul sursă reg/mem (16 sau 32 de biți), până la primul bit egal cu 1. BSF parcurge de la dreapta la stânga iar BSR invers. Indicele bitului 1 identificat este plasat în primul operand (registru)

- BT reg/mem, im8/reg ; Bit Test

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF.

- BTC reg/mem, im8/reg ; Bit Test and Complement

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi complementează bitul respectiv.

- BTR reg/mem, im8/reg ; Bit Test and Reset

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi forțează bitul respectiv la valoarea 0.

- BTS reg/mem, im8/reg ; Bit Test and Set

Plasează bitul cu numărul dat de im8 sau de reg al operandului reg/mem în CF, apoi forțează bitul respectiv la valoarea 1.

Instrucțiuni de control

- JECXZ etichetă ; Jump if ECX is Zero

Este extensia instrucțiunii JCXZ pentru registrul ECX.

• SETccc reg/mem ; Set Bytes Conditionally

Setează un octet conform unei condiții logice. Octetul poate fi un registru de 8 biți sau o locație de memorie. Condițiile logice și semnificațiile lor sunt aceleași ca la instrucțiunile de salt condiționat. În mnemonicele instrucțiunilor se substituie ccc cu condiția care este codificată sub forma Jccc la instrucțiunile de salt condiționat. Există deci instrucțiuni SETG, SETGE, SETNB, SETPO, SETNLE etc. De exemplu:

SETLE	AL
SETNBE	byte ptr [SI]
SETNC	byte ptr [EBX]

3.3 Coprocesoare matematice

3.3.1 Arhitectura coprocesoarelor matematice

Coprocesoarele matematice sunt circuite integrate dedicate (procesoare specializate), care extind setul de instrucțiuni al procesoarelor centrale cu instrucțiuni specifice operațiilor cu numere reale în virgulă mobilă.

Coprocesorele sunt fie circuite de sine stătătoare (8087, 80287, 80387), fie sunt integrate în procesorul de bază (80486). În cel de-al doilea caz nu se mai face distincție între setul de instrucțiuni al de bază și cel specific formatului în virgulă mobilă.

Tipurile de date recunoscute de coprocesoare sunt:

- număr real în simplă precizie (dword);
- număr real în dublă precizie (qword);
- număr real în precizie extinsă (tbytes);
- întreg pe 2 octeți (word);
- întreg pe 4 octeți (dword);
- întreg BCD pe 10 cifre (tbytes).

Aceste tipyuri de date vor fi notate cu real32, real64, real80, int16, int32 și bcd80. Întregii pe 8 octeți pot fi numai încărcăți în coprocesor. Intern, coprocesoarele lucrează exclusiv cu formatul real în precizie extinsă (10 octeți), descris în Capitolul 1. La încărcarea operanzilor din memorie, toate tipurile de date de mai sus sunt convertite la tipul intern; similar, la depunerea operanzilor în memorie, au loc conversii de la formatul intern la unul din formatele de mai sus.

Arhitectura coprocesoarelor cuprinde 8 registre de câte 80 de biți, numite ST(0), ST(1), ..., ST(7). Aceste registre sunt organizate ca o stivă, registrul ST(0) (care se mai notează simplu cu ST) fiind vârful stivei (vezi Figura 3.5). Coprocesoarele dispun de o serie de registre suplimentare, dintre care cele mai importante sunt registrul de stare (Status Word) și registrul de control (Control Word). Registrul de stare conține flaguri care se poziționează în urma instrucțiunilor de comparație sau în caz de eroare. Registrul de control conține câmpuri care controlează modul de execuție al anumitor acțiuni (de exemplu, cum se face rotunjirea la valori întregi).

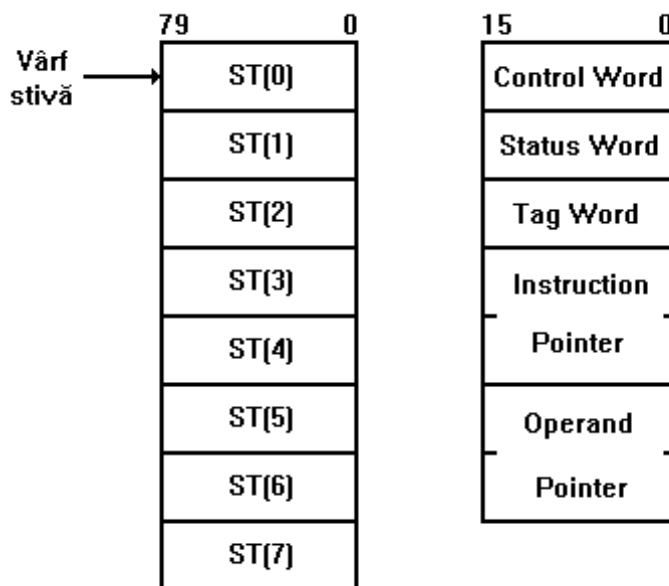


Figura 3.5 Arhitectura coprocesoarelor 80x87

Comunicația dintre procesorul de bază și coprocesor se face exclusiv prin intermediul memoriei. Coprocesorul dispune de instrucțiuni de transfer între memorie și cele 8 registre de lucru, precum și pentru cuvintele de control și de stare.

Ne vom referi în continuare la coprocesorul 8087, deoarece setul de instrucțiuni este practic același la toate coprocesoarele. La procesorele mai vechi decât 80486, se pune și problema comunicației dintre procesorul de bază și coprocesor.

Instrucțiunile specifice 8087 se scriu în textul sursă la fel ca instrucțiunile procesorului de bază. Asamblorul recunoaște mnemonicele acestor instrucțiuni și generează cod mașină corespunzător. Acest cod va fi executat de către coprocesor, în urma generării unui semnal de către procesorul de bază 8086.

Coprocesorul 8087 monitorizează permanent fluxul de instrucțiuni și sesizează prezența unei instrucțiuni specifice 8087 în memorie. În acest caz, el semnalează intenția de a intra în execuție prin semnalul electric TEST. Execuția începe numai după ce 8086 intră în așteptare, ca urmare a unei instrucțiuni WAIT. Coprocesorul recunoaște starea de așteptare și începe execuția instrucțiunii matematice, anulând în același timp cererea efectuată prin semnalul TEST. Ca urmare, 8086 iese din starea de așteptare și își continuă execuția. Asamblarele inserează automat o instrucțiune WAIT înaintea fiecărei instrucțiuni 8087, deci nu este necesară codificarea lor explicită.

La coprocesoarele de generație mai nouă (80387), sincronizarea cu procesorul de bază se face prin semnale specializate, ceea ce elimină necesitatea instrucțiunilor WAIT.

În mod corespunzător, există instrucțiunea 8087 FWAIT, destinată sincronizării reciproce (8087 "așteaptă" după 8086). Instrucțiunea FWAIT este necesară numai după operațiile de depunere în memorie executate de 8087.

Înainte de începerea operației propriu-zise, este necesară o instrucțiune FINIT, care șterge registrele interne, condițiile de eroare etc.

3.3.2 Setul de instrucțiuni 8087

Mnemonicile instrucțiunilor 8087 încep toate cu litera F. Majoritatea instrucțiunilor are ca operanzi vârful ST al stivei coprocesorului și un alt registru ST(i) sau un operand în memorie. Cele mai multe instrucțiuni matematice actualizează stiva, prin operațiile descrise mai jos (ST și ST(0) reprezintă același registru):

```
PUSH_ST:
    for (i = 7 to 1)
        ST(i) <--- ST(i-1)
POP_ST:
    for (i = 1 to 7)
        ST(i-1) <--- ST(i)
```

Instrucțiuni pentru încărcarea datelor

FLD	ST(i)	(Încarcă ST(i) în ST, cu deplasarea stivei)
	temp <-- ST(i)	
	PUSH_ST	
	ST <-- temp	
FLD	mem (real32/64/80)	(Încarcă număr real din memorie în ST)
	PUSH_ST	
	ST <-- mem	
FILD	mem (int16/32/64)	(Încarcă număr întreg din memorie în ST)
	PUSH_ST	
	ST <-- mem	
FBLD	mem (bcd80)	(Încarcă număr BCD din memorie în ST)
	PUSH_ST	
	ST <-- mem	
FLDZ		(Încarcă constanta 0.0 în ST)
	PUSH_ST	
	ST <-- 0.0	
FLD1		(Încarcă constanta 1.0 în ST)
	PUSH_ST	
	ST <-- 1.0	
FLDPI		(Încarcă constanta π în ST)
	PUSH_ST	
	ST <-- π	
FLDL2E		(Încarcă constanta $\log_2(e)$ în ST)
	PUSH_ST	
	ST <-- $\log_2(e)$	
FLDL2T		(Încarcă constanta $\log_2(10)$ în ST)
	PUSH_ST	
	ST <-- $\log_2(10)$	
FLDLG2		(Încarcă constanta $\log_{10}(2)$ în ST)
	PUSH_ST	
	ST <-- $\log_{10}(2)$	

Instrucțiuni pentru depunerea datelor

FST	ST(i)	(Depune ST în ST(i) fără descărcarea stivei)
	ST(i) <-- ST	
FSTP	ST(i)	(Depune ST în ST(i) cu descărcarea stivei)
	ST(i) <-- ST	
	POP_ST	
FST	mem (real32/64/80)	(Depune ST ca real în memorie fără descărcare)
	mem <-- ST	
FSTP	mem (real32/64/80)	(Depune ST ca real în memorie cu descărcare)

	mem <-- ST	
	POP_ST	
FIST	mem (int16/32)	(Depune ST ca întreg în memorie fără descărcare)
	mem <-- ST	
FISTP	mem (int16/32)	(Depune ST ca întreg în memorie cu descărcare)
	mem <-- ST	
	POP_ST	
FBSTP	mem (bcd80)	(Depune ST ca BCD în memorie cu descărcare)
	mem <-- ST	
	POP_ST	
FXCH		(Schimbă ST cu ST(1))
FXCH	ST(i)	(Schimbă ST cu ST(i))

Instrucțiuni de adunare

FADD		(Adună ST cu ST(1), cu descărcare)
	ST(1) <- ST(1) + ST	
	POP_ST	
FADD	ST(i), ST	(Adună ST la ST(i))
	ST(i) <-- ST(i) + ST	
FADD	ST, ST(i)	(Adună ST(i) la ST)
	ST <-- ST + ST(i)	
FADD	mem (real32/64/80)	(Adună real din memorie la ST)
	ST <-- ST + mem	
FIADD	mem (int16/32)	(Adună întreg din memorie la ST)
	ST <- ST + mem	
FADDP	ST(i)	(Adună ST la ST(i) cu descărcarea stivei)
	ST(i) <-- ST(i) + ST	
	POP_ST	

Instrucțiuni de scădere

FSUB		(Calculează ST(1) - ST cu descărcare)
	ST(1) <- ST(1) - ST	
	POP_ST	
FSUB	ST(i), ST	(Scade ST din ST(i))
	ST(i) <-- ST(i) - ST	
FSUB	ST, ST(i)	(Scade ST(i) din ST)
	ST <-- ST - ST(i)	
FSUB	mem (real32/64/80)	(Scade real în memorie din ST)
	ST <-- ST - mem	
FISUB	mem (int16/32)	(Scade întreg în memorie din ST)
	ST <- ST - mem	
FSUBP	ST(i)	(Scade ST din ST(i) cu descărcare)
	ST(i) <-- ST(i) - ST	
	POP_ST	
FSUBR		(Calculează ST - ST(1) cu descărcare)
	temp <-- ST - ST(1)	
	POP_ST	
	ST <-- temp	
FSUBR	ST(i), ST	(Scade ST(i) din ST, rezultat în ST(i))
	ST(i) <-- ST - ST(i)	
FSUBR	ST, ST(i)	(Scade ST din ST(i), rezultat în ST)
	ST <-- ST(i) - ST	
FSUBR	mem (real32/64/80)	(Scade ST din real în memorie, rezultat în ST)
	ST <-- mem - ST	
FISUBR	mem (int16/32)	(Scade ST din întreg în memorie, rez. în ST)
	ST <- mem - ST	
FSUBPR	ST(i)	(Scade ST(i) din ST cu descărcarea stivei)
	ST(i) <-- ST - ST(i)	
	POP_ST	

Instrucțiuni de înmulțire

FMUL		(Înmulțește ST cu ST(1), cu descărcare)
	ST(1) <- ST(1) * ST	
	POP_ST	
FMUL	ST(i), ST	(Înmulțește ST cu ST(i), rezultat în ST(i))
	ST(i) <-- ST(i) * ST	
FMUL	ST, ST(i)	(Înmulțește ST(i) cu ST, rezultat în ST(i))
	ST <-- ST * ST(i)	
FMUL	mem (real32/64/80)	(Înmulțește real din memorie cu ST)
	ST <-- ST * mem	
FIMUL	mem (int16/32)	(Înmulțește întreg din memorie cu ST)
	ST <- ST * mem	
FMULP	ST(i)	(Înmulțește ST cu ST(i), cu descărcare)
	ST(i) <-- ST(i) * ST	
	POP_ST	

Instrucțiuni de împărțire

FDIV		(Calculează ST(1) / ST cu descărcare)
	ST(1) <-- ST(1) / ST	
	POP_ST	
FDIV	ST(i), ST	(Împarte ST(i) la ST, rezultat în ST)
	ST(i) <-- ST(i) / ST	
FDIV	ST, ST(i)	(Împarte ST la ST(i), rezultat în ST(i))
	ST <-- ST / ST(i)	
FDIV	mem (real32/64/80)	(Împarte ST la real din memorie)
	ST <-- ST / mem	
FIDIV	mem (int16/32)	(Împarte ST la întreg din memorie)
	ST <- ST / mem	
FDIVP	ST(i)	(Împarte ST(i) la ST cu descărcare)
	ST(i) <-- ST(i) / ST	
	POP_ST	
FDIVR		(Calculează ST / ST(1) cu descărcare)
	temp <-- ST / ST(1)	
	POP_ST	
	ST <-- temp	
FDIVR	ST(i), ST	(Împarte ST la ST(i), rezultat în ST(i))
	ST(i) <-- ST / ST(i)	
FDIVR	ST, ST(i)	(Împarte ST(i) la ST, rezultat în ST)
	ST <-- ST(i) / ST	
FDIVR	mem (real32/64/80)	(Împarte real în memorie la ST, rezultat în ST)
	ST <-- mem / ST	
FIDIVR	mem (int16/32)	(Împarte întreg în memorie la ST, rez. în ST)
	ST <- mem / ST	
FDIVPR	ST(i)	(Împarte ST la ST(i) cu descărcare)
	ST(i) <-- ST / ST(i)	
	POP_ST	

Instrucțiuni de comparație

Instrucțiunile de comparație poziționează indicatorii din cuvântul de stare prin efectuarea unei operații temporare de scădere între operanzii care se compară.

FCOM		(Compară ST cu ST(1))
FCOM	ST(i)	(Compară ST cu ST(i))
FCOM	mem (real32/64/80)	(Compară ST cu real în memorie)
FICOM	mem (int16/32)	(Compară ST cu întreg din memorie)
FTST		(Compară ST cu 0.0)
FCOMP		(Compară ST cu ST(1), cu descărcare)
	temp <-- ST - ST(1)	
	POP_ST	
FCOMP	ST(i)	(Compară ST cu ST(i), cu descărcare)
FCOMP	mem (real32/64/80)	(Compară ST cu real, cu descărcare)
FICOMP	mem (int16/32)	(Compară ST cu întreg, cu descărcare)

FCOMPP	(Compară ST cu ST(1), cu descărcare dublă)
temp <-- ST - ST(1)	
POP_ST	
POP_ST	

Instrucțiuni diverse

FABS	(Ia valoarea absolută)
ST <-- ST	
FCHS	(Schimbă semnul)
ST <-- - ST	
FSQRT	(Radical)
ST <-- sqrt (ST)	

Instrucțiuni de control

FINIT	(Inițializează coprocesor)
FWAIT	(Sincronizează cu procesorul de bază)
FSTSW mem (int16)	(Depune cuvânt de stare în memorie)
FSTCW mem (int16)	(Depune cuvânt de control în memorie)
FLDCW mem (int16)	(Încarcă cuvânt de control din memorie)
FNOP	(Nici o operație)

Pe lângă instrucțiunile de mai sus, coprocesoarele dispun de instrucțiuni transcendente, pentru calculul funcțiilor trigonometrice directe și inverse, exponențială, logaritm etc. Aceste instrucțiuni nu vor fi prezentate, deoarece este puțin probabil că se vor dezvolta programe cu funcții transcendente în limbaj de asamblare.

În Capitolul 6 vor fi prezentate tehnici de programare specifice coprocesoarelor matematice.