

## Capitolul 2

### Setul de instrucțiuni 8086

Procesoarele din familia Intel dispun de un set puternic de instrucțiuni aritmetice, logice, de transfer și de control, ceea ce permite încadrarea acestor procesoare în clasa **CISC (Complex Instruction Set Computer)**.

În cadrul acestui capitol, sunt prezentate în detaliu instrucțiunile de bază ale familiei de procesoare Intel. În general, pentru toate tipurile de instrucțiuni, se poate folosi orice mod de adresare (inclusiv cu prefixe de segment) expuse în capitolul anterior. Acolo unde este cazul, se specifică tipurile interzise de adresare.

Setul de instrucțiuni este grupat în 6 clase:

- **instrucțiuni de transfer**, care deplasează date între memorie sau porturi de intrare/ieșire și registrele procesorului, fără a executa nici un fel de prelucrare a datelor;
- **instrucțiuni aritmetice și logice**, care prelucrează date în format numeric;
- **instrucțiuni pentru șiruri**, specifice operațiilor cu date alfanumerice;
- **instrucțiuni pentru controlul programului**, care în esență se reduc la salturi și la apeluri de proceduri;
- **instrucțiuni specifice întreruperilor** hard și soft;
- **instrucțiuni pentru controlul procesorului**.

Această împărțire este realizată după criterii funcționale. De exemplu, instrucțiunile PUSH și POP sunt considerate ca instrucțiuni de transfer, deși, la prima vedere, ar putea fi considerate instrucțiuni specifice procedurilor. Același lucru despre instrucțiunile IN și OUT, care interfațează procesorul cu lumea exterioară: ele sunt considerate instrucțiuni de transfer, deși ar putea fi considerate instrucțiuni de intrare/ieșire. Intrările și ieșirile sunt însă cazuri particulare de transfer.

Fiecare categorie de instrucțiuni este însoțită de specificarea explicită a bistabililor de condiție care sunt modificați în urma execuției.

#### 2.1 Instrucțiuni de transfer

Instrucțiunile de transfer presupun o copiere a unui octet sau unui cuvânt de la o sursă la o destinație. Această copiere (atribuire) va fi desemnată uneori printr-o săgeată, de la sursă către destinație. Destinația poate fi un registru, o locație de memorie sau un port de ieșire iar sursa poate fi un registru, o locație de memorie, date imediate (constante) sau un port de intrare. Cu excepția cazului important al instrucțiunilor PUSH și POP (2.1.1), sursa și destinația nu pot fi simultan locații de memorie.

Instrucțiunile de transfer se clasifică în instrucțiuni generale, specifice acumulatorului, specifice adreselor și specifice flagurilor.

În specificarea destinației și sursei, se vor folosi notațiile segment:offset pentru adrese și notația (x), pentru a desemna "conținutul lui x". Notația cu paranteze se poate extinde pe mai multe niveluri. De exemplu, (BX) înseamnă conținutul registrului BX, iar ((BX)) înseamnă conținutul locație

de memorie adresate de BX (implicit prin DS). Similar ES:((BX)) va însemna conținutul locație de memorie adresate de registrele ES și BX.

Cu excepția instrucțiunilor SAHF și POPF (2.1.4), nici o instrucțiune de transfer nu modifică vreun bistabil de condiție.

### **2.1.1 Instrucțiuni de transfer generale - MOV, PUSH, POP, XCHG**

#### **Instrucțiunea MOV (Move Data - Transferă Date)**

Formatul general este:

```
MOV  destinație, sursă      ; (destinație) <-- sursă
```

În care sursă și destinație (operanzii) sunt octeți sau cuvinte respectând regulile descrise mai sus.

Următoarele operații sunt ilegale:

- sursa și destinația nu pot fi ambele operanzi în memorie;
- nu pot fi folosite registrele FLAGS și IP;
- operanzii nu pot avea dimensiuni diferite;
- registrul CS nu poate apare ca destinație.

Procesorul original 8086 are restricțiile suplimentare:

- nu pot fi transferate date imediate într-un registru de segment;
- operanzii nu pot fi simultan registre de segment.

Aceste ultime două restricții au fost relaxate la variantele ulterioare (80286 și peste). Exemple de instrucțiuni corecte:

```
MOV  AX, BX
MOV  AL, CH
MOV  VAL[BX][SI], AL
MOV  byte ptr [BX+100], 5
```

Ultima instrucțiune de mai sus ar fi fost ambiguă fără utilizarea operatorul ptr: MOV [BX+100], 5 se poate interpreta ca "pune valoarea 5 în octetul de la adresa DS:BX+100", sau, la fel de bine, "pune valoarea 5 la cuvântul de la adresa DS:BX+100". Forma "byte ptr" precizează că este vorba de un transfer pe octet.

Exemple de instrucțiuni incorecte:

```
MOV  AL, BX           ; operanzi de lungime diferită
MOV  [BX], [SI]       ; ambii operanzi în memorie
MOV  CS, AX           ; registrul cs apare ca destinație
```

De reținut că o instrucțiune de forma:

```
.data
    ALFA  DB    1
.code
MOV  AL, ALFA
```

încarcă în AL conținutul locației de memorie ALFA. Dacă se dorește încărcarea adresei efective a variabilei ALFA, se poate folosi operatorul OFFSET:

```
MOV  BX, OFFSET ALFA
```

sau instrucțiunea LEA (2.1.3).

**Instrucțiunea PUSH (Push Data - Salvează date în stivă)**

Forma generală este:

```
PUSH sursă
```

În care sursă este un operand pe 16 biți (registru general de 16 biți, registru de segment sau locație de memorie), iar semnificația este "copiază sursă în vârful stive". Concret, execuția instrucțiunii se face după secvența:

```
(SP) <-- (SP) - 2
SS : ((SP)+1 : (SP)) <-- sursă
```

ceea ce înseamnă că se decrementează SP cu 2 și în octeții de la adresele (SP)+1 și (SP) din segmentul de stivă se copiază operandul sursă. Copierea respectă regula de memorare a cantităților pe mai mulți octeți și anume, partea mai puțin semnificativă (partea low) se memorează la adrese mici. O exprimare detaliată a instrucțiunii ar putea fi:

```
(SP) <-- (SP) - 2
SS : ((SP)+1) <-- high (sursă)
SS : ((SP)) <-- low (sursă)
```

Exemple de instrucțiuni corecte:

```
PUSH BX
PUSH ES
PUSH [BX]
PUSH [BP+5]
PUSH ES:[BX][SI+4]
```

Exemplu de instrucțiuni incorecte:

```
PUSH AL ; Operand pe 1 octet
```

**Instrucțiunea POP (Pop Data - Refă date din stivă)**

Forma generală este:

```
POP destinație
```

În care destinație este un operand pe 16 biți (registru general de 16 biți, registru de segment sau locație de memorie), iar semnificația este "copiază conținutul vârfului stivei în destinație". Registrul CS nu poate apare ca destinație. Concret, execuția instrucțiunii se face după secvența:

```
destinație <-- SS : ((SP)+1 : (SP))
(SP) <-- (SP) + 2
```

ceea ce înseamnă că se transferă octeții de la adresele (SP)+1 și (SP) din segmentul de stivă în operand (destinație) și apoi se incrementează SP cu valoarea 2. O exprimare detaliată a instrucțiunii ar putea fi:

```
high (destinație) <-- SS : ((SP)+1)
low (destinație) <-- SS : ((SP))
(SP) <-- (SP) + 2
```

Exemple de instrucțiuni corecte:

```
POP BX
POP ES
POP ES:[DI]
POP [BP+5]
```

POP SS:[BX+4]

Exemple de instrucțiuni incorecte:

POP AL ; Operand pe 1 octet  
POP CS ; Registrul cs

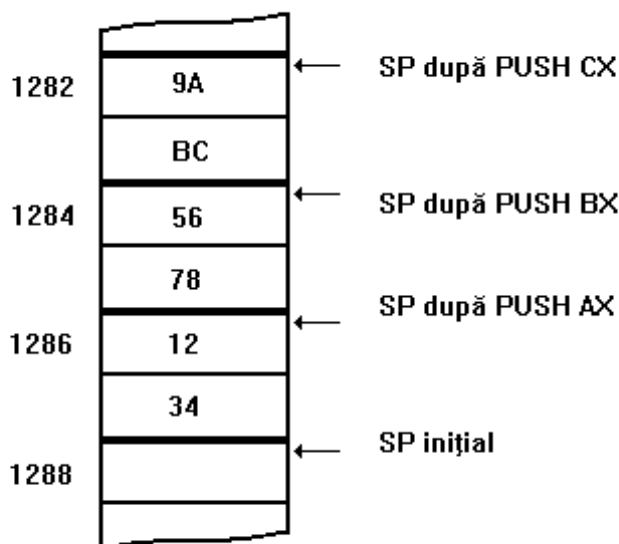
Din analiza instrucțiunilor PUSH și POP, reiese că o secvență de refaceri ale unor cantități salvate în stivă (de exemplu, conținutul unor registre) trebuie scrisă în ordine inversă. Dacă secvența de salvare a fost:

PUSH AX  
PUSH BX  
PUSH CX

atunci secvența de refacere trebuie să fie:

POP CX  
POP BX  
POP AX

Dacă registrele de mai sus conțin valorile AX = 1234H, BX = 5678H și CX = 9ABCH, iar registrul SP conține (înainte de salvări) valoare 1288H, atunci imaginea stivei va fi cea din Figura 2.1.



**Figura 2.1 Imaginea stivei după o secvență de instrucțiuni PUSH și POP**

La operațiile cu stiva, trebui avut grijă ca o secvență de refacere să aducă indicatorul SP la valoarea de dinainte de secvența de salvare. Acest lucru înseamnă că, de regulă, numărul operațiilor POP trebuie să coincidă cu cel al operațiilor PUSH.

Tot ca regulă generală, nu este indicat să se modifice explicit locațiile aflate "în josul stivei", adică la valori mai mari sau egale decât valoarea curentă a registrului SP. În acele locații se pot afla informații a căror alterare ar putea compromite definitiv execuția programului. Trebuie deci folosite cu atenție secvențele de genul:

MOV BX, SP  
MOV SS:[BX], AX  
MOV byte ptr SS:[BX+2], 1

Instrucțiunile PUSH și POP se mai pot folosi la transferul indirect al unor registre. Secvența:

```
PUSH DS
POP ES
```

copiază conținutul registrului DS în ES, lăsând indicatorul SP neschimbat.

Instrucțiunile PUSH și POP realizează de fapt un transfer din memorie în memorie. Dacă operandul din cele două instrucțiuni este o locație de memorie, ceea ce transferă la PUSH este conținutul acelei zone. Secvența:

```
.data
    X      dw 100
.code
    PUSH X
```

va pune în vârful stivei valoarea 100 (conținutul lui X). Dacă se dorește punerea în stivă a adresei efective sau complete a variabilei X, se va folosi instrucțiunea MOV și operatorul OFFSET sau instrucțiunea LEA (2.1.3), respectiv registrul de segment care adresează curent segmentul respectiv. Dacă segmentul în care e definită variabila nu este adresat în mod curent de nici un registru de segment, se poate folosi operatorul SEG, care furnizează segmentul în care este definit operandul:

```
.code
    LEA    AX, X
    PUSH  AX                ; Offset-ul la adrese mici
    PUSH  DS                ; Apoi segmentul
;
;    Sau ...
;
    MOV    AX, OFFSET X
    PUSH  AX                ; Offset-ul lui X
    MOV    AX, SEG X
    PUSH  AX                ; Adresa de segment a lui X
```

### **Instrucțiunea XCHG (Exchange Data - Interschimbă date)**

Forma generală este:

```
XCHG destinație, sursă
```

iar semnificația este cea de interschimbare a sursei cu destinația. Registrele de segment nu pot apărea ca operanzi și, bineînțeles, cel puțin un operand trebuie să fie registru.

Exemple de instrucțiuni corecte:

```
XCHG AL, AH
XCHG BX, SI
XCHG ES:[BX], AX
```

Exemple de instrucțiuni incorecte:

```
XCHG AL, BX      ; Operanzi de lungime diferită
XCHG ES, AX      ; Registru de segment
```

Instrucțiunea XCHG este utilă la interschimbarea a două cantități aflate în memorie. Dacă op1 și op2 sunt doi operanzi aflați în memorie, care trebuie interschimbați, secvența standard de interschimbare (folosind un registru general reg) este:

```
MOV reg, op1
XCHG reg, op2
MOV op1, reg
```

### 2.1.2 Instrucțiuni de transfer specifice acumulatorului - IN, OUT, XLAT

#### Instrucțiunea IN (Input Data - Citește date de la port de intrare)

Forma generală este:

```
IN    destinație, port
```

În care destinație este registrul AL sau AX iar port este fie o constantă cuprinsă între 0 și 255, fie registrul DX. Variantele noi de procesoare acceptă orice registru în locul lui AL sau AX. Semnificația este că se execută o citire de la portul de intrare specificat, pe 8 sau 16 biți, după cum s-a specificat registrul AL sau AX. La variantele noi de procesoare (80286 și peste), registrele AL sau AX pot fi substituite cu orice registre generale.

#### Instrucțiunea OUT (Output Data - Scrie date la port de ieșire)

Forma generală este:

```
OUT    destinație, port
```

În care destinație este registrul AL sau AX iar portul de ieșire este specificat la fel ca la instrucțiunea IN.

Instrucțiunile IN și OUT sunt singurele instrucțiuni propriu-zise care pot realiza interacțiunea procesorului cu alte dispozitive. Unele arhitecturi de calculatoare au organizat memoria în așa fel încât zone din spațiul adresabil sunt dedicate unor echipamente periferice și nu unor zone propriu-zise de memorie. Accesul la zonele respective de memorie va însemna de fapt un acces la echipamentul periferic. Asemenea sisteme de intrări/ieșiri se numesc de tip "memory-mapped" (intrări-ieșiri organizate ca spațiu de memorie).

Să considerăm că un echipament periferic necesită un port de stare și un port de date, ambele pe 8 biți. Într-un sistem de intrări-ieșiri obișnuit, vor exista două porturi de intrare, de exemplu, 0F8H și 0F9H, dedicate perifericului respectiv. Într-un sistem de tip "memory-mapped", vor exista două adrese, de obicei adiacente, de exemplu c800:0000 și c800:0001, corespunzătoare porturilor de stare și de date. Secvențele de citire stare, respectiv date, în cele două tipuri de intrări/ieșiri vor fi:

```
IN    AL, 0F8H    ; Citire stare
IN    AL, 0F9H    ; Citire date
MOV   ES, 0C800H
MOV   AL, ES:[0]   ; Citire stare
MOV   AL, ES:[1]   ; Citire date
```

#### Instrucțiunea XLAT (Translate - Translatează)

Instrucțiunea nu are operanzi iar semnificația este:

```
(AL) <-- DS : ((BX) + (AL))
```

adică se aduce în AL conținutul octetului de la adresa (BX)+(AL). Această instrucțiune este folosită împreună cu tabele de traducere (de unde și numele), utile în conversia unor tipuri de date. De exemplu, dacă dorim să

convertim o valoare numerică între 0 și 15 la cifra hexa corespunzătoare, putem folosi secvența:

```
.data
    TABELA      DB    '0123456789ABCDEF'
.code
    MOV    BX, OFFSET TABELA
    XLAT
```

prin care se încarcă în BX offsetul tabelii de octeți și se face apoi translatarea.

### **2.1.3 Instrucțiuni de transfer specifice adreselor - LEA, LDS, LES**

Aceste instrucțiuni transferă o adresă efectivă într-un registru general, sau o adresă completă pe 32 de biți într-o pereche de registre.

#### **Instrucțiunea LEA (Load Effective Address - Încarcă adresă efectivă)**

Forma generală este:

```
LEA    registru, sursă
```

în care sursă este un operand aflat în memorie, specificat printr-un mod oarecare de adresare. Efectul este copierea adresei efective a operandului (offsetul în cadrul segmentului) în registrul general specificat. Exemple:

```
LEA    BX, ALFA
LEA    DI, ALFA [BX] [SI]
```

Un calcul similar al adresei efective se obține și cu operatorul OFFSET și instrucțiunea MOV, calcul care se face însă la asamblare. Astfel, instrucțiunea LEA permite și moduri de adresare bazată și/sau indexată.

Instrucțiunea LEA se folosește pentru încărcarea registrelor de bază sau de segment cu adresele efective ale unor operanzi din memorie, în vederea unor adresări ulterioare. Secvența:

```
.data
    VECTOR      DW 10, 20, 30, 40, 50
.code
    LEA    BX, VECTOR
    MOV    SI, 4
    MOV    AX, [BX][SI]
```

va încărca în AX al treilea element al tabloului VECTOR.

#### **Instrucțiunile LDS (Load Data Segment - Încarcă DS) și LES (Load Extra Segment - Încarcă ES)**

Forma generală este:

```
LDS    reg, sursă
LES    reg, sursă
```

în care reg este un registru general de 16 biți, iar sursă este un operand de tip double-word aflat în memorie, care conține o adresă completă de 32 de biți. Efectul instrucțiunii este:

```
(reg)      <--  ((sursa))
(DS)/(ES)  <--  ((sursa)+2)
```

adică se încarcă perechea DS:reg, respectiv ES:reg, cu o adresă completă de 32 de biți. De obicei instrucțiunea LDS se folosește împreună cu directiva Define DoubleWord, ca în exemplul următor:

```
.data
    x      db      10
    y      db      15
    adr_x  dd      x
    adr_y  dd      y
.code
    LDS     SI, adr_x
    LES     DI, adr_y
    MOV     byte ptr [SI], 20
    MOV     byte ptr ES:[DI], 30
```

Variabilele adr\_x și adr\_y, care conțin adresele far ale variabilelor x și y, sunt încărcate în DS:SI și ES:DI. Se accesează apoi variabilele x și y, folosind adresarea indexată.

Următorul exemplu utilizează o tabelă de adrese. Să presupunem că avem 8 variabile word diferite, numite V\_0, ..., V\_7 și dorim să încărcăm în AX variabila a cărei indice este dat de registrul CX. Cu alte cuvinte, dacă CX=0, încărcăm V\_0, dacă CX=1, încărcăm V\_1 etc. Definim în acest scop tabela de adrese TAB\_ADR și folosim adresarea indexată.

```
.data
    TAB_ADR dd      V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7
.code
    MOV     BX, CX
    ADD     BX, BX
    ADD     BX, BX
    LES     DI, TAB_ADR[BX]
    MOV     AX, ES:[DI]
```

Deoarece o poziție din tabelă ocupă 4 octeți, valoarea indicelui din CX trebuie înmulțită cu 4. Acest lucru se obține prin două instrucțiuni ADD BX, BX, care adună BX la el însuși.

#### 2.1.4 Instrucțiuni de transfer specifice flagurilor (LAHF, SAHF, PUSHF, POPF)

##### Instrucțiunea LAHF (Load AH with FLAGS - Încarcă AH cu FLAGS)

Instrucțiunea nu are operanzi, iar semnificația este dată de denumire: se încarcă registrul AH cu partea mai puțin semnificativă a registrului de flaguri:

```
AH <--- FLAGS0:7
```

##### Instrucțiunea SAHF (Store AH into FLAGS - Depune AH în FLAGS)

Este perechea instrucțiunii LAHF, semnificația fiind:

```
FLAGS0:7 <--- AH
```

##### Instrucțiunea PUSHF (Push Flags - Salvează FLAGS în stivă)

Nu are operanzi, iar efectul este plasarea registrului FLAGS în vârful stivei ("salvarea" flagurilor în stivă):

```
(SP) <--- (SP) - 2
SS:((SP) + 2 : (SP)) <--- FLAGS
```



**Instrucțiunea POPF (Pop Flags - Reface FLAGS din stivă)**

Este perechea instrucțiunii PUSHF:

FLAGS	<---	SS:((SP) + 1 : (SP))
(SP)	<---	(SP) + 2

Instrucțiunile LAHF și SAHF citesc și scriu explicit parte mai puțin semnificativă a registrului de flaguri. Partea mai semnificativă poate fi citită și modificată prin mici artificii cu instrucțiunile PUSHF și POPF. Secvența:

PUSHF
POP AX

va încărca în AX registrul de flaguri, deci în AH vom avea partea mai semnificativă, iar secvența:

PUSH AX
POPF

va încărca AX în registrul de flaguri.

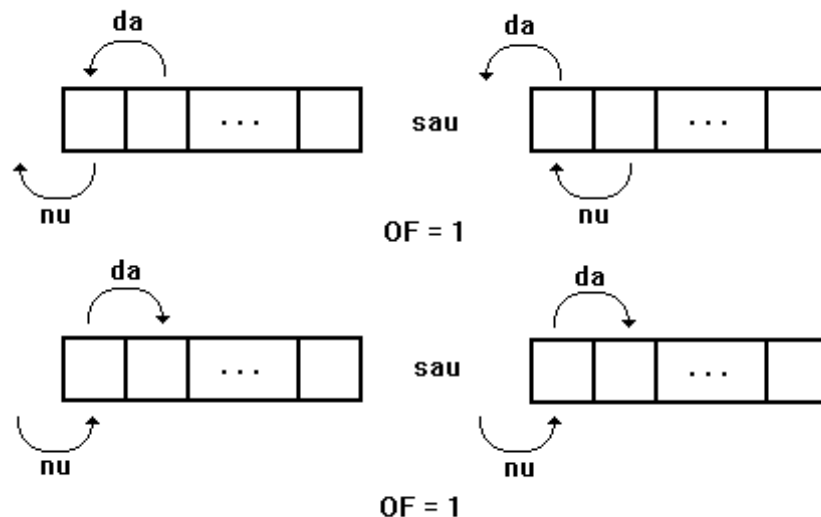
Cu excepția instrucțiunilor explicite SAHF și POPF, nici o instrucțiune de transfer nu modifică bistabilii de condiție.

**2.2 Instrucțiuni aritmetice și logice**

A doua categorie importantă de instrucțiuni o constituie instrucțiunile aritmetice și logice. Rezultatul operației este totdeauna depus într-unul din operanzi. La instrucțiunile cu doi operanzi, rezultatul este depus în primul operand. Instrucțiunile modifică flagurile CF, AF, ZF, SF, PF și OF, motiv pentru care acestea mai sunt numite și flaguri aritmetice.

**2.2.1 Semnificația flagurilor aritmetice**

- **Flagul Carry** = 1 semnifică un transport/împrumut din/în bitul cel mai semnificativ (b.c.m.s.) al rezultatului, unde b.c.m.s. poate fi bitul 7 sau bitul 15. Interpretarea acestui transport sau împrumut este cea de depășire la operațiile de adunare/scădere cu operanzi fără semn.
- **Flagul Auxiliary Carry** = 1 semnifică un transport/împrumut din/în bitul 4 al rezultatului, fiind utilizat la operațiile cu numere BCD.
- **Flagul Zero** este 1 dacă rezultatul operației este nul.
- **Flagul Sign** este 1, dacă bitul de semn al rezultatului (bitul 7 sau 15) este 1.
- **Flagul Parity** este 1 dacă suma modulo 2 a celor 8 biți mai puțin semnificativi ai rezultatului este 0.
- **Flagul Overflow** este 1 dacă operația a condus la un transport înspre b.c.m.s., dar nu din b.c.m.s. al rezultatului, sau invers. Altfel, OF se poziționează la zero. Similar în cazul împrumutului: OF = 1, dacă a avut loc un împrumut dinspre b.c.m.s. dar nu din exterior, sau invers. Situațiile descrise sunt ilustrate în Figura 2.2.



**Figura 2.2 Poziționarea bistabilului Overflow**

Interpretarea flagului OF este aceea de depășire la operațiile de adunare/scădere cu operanzi cu semn.

Să considerăm valoarea pe 1 octet 0FFH și să adunăm 1 la această valoare. Rezultatul este 0 și transport atât din bitul 7 cât și din bitul 6 în bitul 7. Astfel, CF = 1 și OF = 0. Considerând operanzii fără semn (255 și 1), adunarea lor provoacă depășire, ceea ce corespunde cu CF = 1. Dacă se consideră operanzii cu semn, (-1 și 1), adunarea lor nu provoacă depășire, și OF = 0.

Să considerăm, în mod similar, valorile 70H și 10H și suma acestora. Rezultatul este 80H și nu apare transport din bitul 7, dar este transport din bitul 6 în bitul 7. Ca atare, CF = 0 și OF = 1. În interpretarea fără semn, valorile operanzilor sunt 112 și 16, suma lor fiind 128, deci nu apare depășire (CF = 0). În interpretarea cu semn, valorile sunt tot 112 și 16, dar suma lor depășește domeniul admisibil al octeților cu semn și OF = 1.

În fine, să considerăm operanzii 0FDH și 02H și suma lor. La adunare, se obține valoarea 0FFH și nu apare transport nici din bitul 7 nici din bitul 6. Ca atare, CF = 0 și OF = 0. Valorile fără semn sunt 253 și 2, iar suma lor (255) este în domeniul admisibil, deci CF = 0. Valorile cu semn sunt -3 și 2, iar suma lor (-1) este în domeniul de reprezentare al numerelor cu semn, deci OF = 0.

La fiecare instrucțiune aritmetică sau logică, se vor preciza explicit flagurile afectate, adică flagurile care se poziționează conform rezultatului. Un flag neafectat rămâne la vechea valoare. Există și situații în care unele flaguri au valori nedefinite.

### 2.2.2 Instrucțiuni specifice adunării (ADD, ADC, INC, DAA, AAA)

#### Instrucțiunea ADD (Add - Adună)

Are forma generală:

**ADD**    destinație, sursă

În care destinație poate fi un registru general sau o locație de memorie, iar sursă poate fi un registru general, o locație de memorie sau o valoare imediată. Cei doi operanzi nu pot fi simultan locații de memorie. Semnificația este:

(destinație) <--- (destinație) + (sursă)

*Flaguri afectate:* AF, CF, PF, SF, ZF, OF (toate).

Operanzii pot fi pe 8 sau 16 biți și trebuie să aibe aceeași dimensiune. În caz de ambiguitate în ceea ce privește lungimea operanzilor, se folosește operatorul ptr. Iată câteva exemple:

```
ADD  AX, BX
ADD  AX, 1          ; Lungime 16 biți
ADD  AL, 1          ; Lungime 8 biți
ADD  word ptr [DI], -1 ; Dest. în memorie, sursă imediată
ADD  ALFA, 3        ; ALFA este definit cu DW
ADD  byte ptr ALFA, 3 ; Forțare instrucțiune pe octet
```

### **Instrucțiunea ADC (Add with Carry - Adună cu transport)**

Forma generală este:

**ADC**    destinație, sursă

Formatul este similar cu instrucțiunea ADD, ceea ce diferă fiind adunarea bistabilului CF:

(destinație) <-- (destinație) + (sursă) + (CF)

*Flaguri afectate:* AF, CF, PF, SF, ZF, OF (toate).

Adunarea cu Carry se folosește la adunări de operanzi pe mai multe cuvinte, în care poate apare un transport intermediar. De exemplu, secvența de adunare a două numere pe 4 octeți este:

```
.data
    ALFA dd 145A789FH
    BETA dd 92457ABCH
    REZ  dd ?

.code
    MOV  AX, word ptr ALFA ; Cuvintele m.p.s. la adr. mici
    ADD  AX, word ptr BETA ; Aici poate apare transport
    MOV  word ptr REZ, AX
    MOV  AX, word ptr ALFA+2 ; Cuvintele m.s. la adr. mari
    ADC  AX, word ptr BETA+2 ; Se ia în considerare
                                ; transportul precedent
    MOV  word ptr REZ+2, AX
```

### **Instrucțiunea INC (Increment - Incrementează)**

Are forma generală:

**INC**    destinație

În care destinație este un registru sau un operand în memorie, de tip octet sau cuvânt. Semnificația este:

(destinație) <--- (destinație) + 1

*Flaguri afectate:* AF, PF, SF, ZF, OF (fără CF).

## Instrucțiunea DAA (Decimal Adjust for Addition - Corecție zecimală după adunare)

Instrucțiunea nu are operanzi și efectuează corecția zecimală a acumulatorului AL, după o adunare cu operanzi în format BCD despachetat. Semnificația este:

```

dacă (AL0:3) > 9 sau (AF) = 1, atunci {
    (AL) <--- (AL) + 6
    (AF) <--- 1
}
dacă acum (AL4:7) > 9 sau CF = 1, atunci {
    (AL) <--- (AL) + 60H
    (CF) <--- 1
}

```

*Flaguri afectate:* CF, AF, CF, PF, SF, ZF. Flagul OF este nedefinit.

Dacă cifra BCD mai puțin semnificativă (AL<sub>0:3</sub>) este mai mare decât 9, deci este incorectă, sau dacă a avut loc un transport din bitul 3 în bitul 4, se corectează această cifră prin adunarea valorii 6. Dacă cifra BCD mai semnificativă este acum mai mare decât 9, se corectează similar această cifră. În ambele situații, se poziționează corespunzător AF și CF, pentru a indica depășirea care a avut loc.

Să considerăm, de exemplu, adunarea valorilor BCD 65 și 17. Aceste valori se reprezintă prin octeții 65H și 17H. În urma adunării, se obține rezultatul 7CH, care este incorect ca rezultat BCD. Operația de corecție (DAA), conduce la rezultatul 82H, ceea ce reprezintă suma BCD a celor două valori. Secvența de adunare trebuie să fie:

```

.data
    BCD1 db 65H
    BCD2 db 17H
    REZ   db ?

.code
    MOV   AL, BCD1
    ADD   AL, BCD2
    DAA
    MOV   REZ, AL

```

## Instrucțiunea AAA (ASCII Adjust for Addition - Corecție ASCII a acumulatorului)

Instrucțiunea nu are operanzi și efectuează corecția acumulatorului AX, după operații de adunare cu operanzi BCD despachetați (o cifră BCD pe un octet). Semnificația este următoarea:

```

dacă (AL0:3) > 9 sau (AF) = 1, atunci {
    (AL) <--- (AL) + 6
    (AH) <--- (AH) + 1
    (AF) <--- 1
    (CF) <--- 1
    (AL) <--- (AL) AND 0FH
}

```

*Flaguri afectate:* AF, CF, restul nedefinite.

Corecția se face tot prin adaugarea valorii 6 la (AL), dar se face și o incrementare a registrului (AH), în ideea că acolo s-ar putea ține cifra BCD

despachetată mai semnificativă. Totodată se șterg biții 4:7 ai registrului AL, pentru a avea o cifră BCD

despachetată. Să considerăm că registrele AX și BX conțin valorile 0309H și 0104H, ceea ce ar corespunde valorilor BCD despachetate 39 și 14. În urma adunării, se obține rezultatul 040DH, care este incorect. Instrucțiunea AAA, corectează acest rezultat la 0503H, care este suma corectă a celor două valori inițiale:

```
MOV  AX, 0309h
MOV  BX, 0104H
ADD  AX, BX           ; AX = 040DH
AAA                      ; AX = 0503H
```

### 2.2.3 Instrucțiuni specifice scăderii (SUB, SBB, DEC, NEG, CMP, DAS, AAS)

#### Instrucțiunea SUB (Subtract - Scade)

Are forma generală:

```
SUB  destinație, sursă
```

unde destinație și sursă sunt la fel ca la instrucțiunea ADD. Semnificația este:

```
(destinație) <--- (destinație) - (sursă)
```

*Flaguri afectate:* AF, CF, PF, SF, ZF, OF (toate).

Scăderea poate fi privită ca o adunare cu complementul față de 2 al operandului sursă, dar cu inversarea rolului bistabilului CF, în sensul că, dacă la această adunare echivalentă apare transport, atunci CF = 0 și reciproc. Să considerăm secvența de instrucțiuni:

```
MOV  AL, 1
SUB  AL, 05EH
```

Rezultatul este 0A3H și există împrumut deci CF = 1. Dacă luăm complementul față de 2 al sursei, obținem valoarea 0A2H, care, adunată la destinație (adică la 1), conduce la valoarea 0A3H. La această adunare echivalentă nu există transport, deci conform regulii de mai sus, operația de scădere de la care am pornit va conduce la CF = 1.

#### Instrucțiunea SBB (Subtract with Borrow - Scade cu împrumut)

Are forma generală:

```
SBB  destinație, sursă
```

în care destinație și sursă sunt la fel ca la instrucțiunea ADD. Semnificația este:

```
(destinație) <-- (destinație) - (sursă) - (CF)
```

deci se ia în considerare un eventual împrumut anterior.

*Flaguri afectate:* AF, CF, PF, SF, ZF, OF (toate).

Instrucțiunea SBB se utilizează la scăderi de operanzi pe mai multe cuvinte, ca în exemplul următor:

```
.data
ALFA  dd 145A789FH
```

```

BETA dd 92457ABCH
REZ  dd ?
.code
MOV  AX, word ptr ALFA      ; Cuvintele m.p.s. la adr. mici
SUB  AX, word ptr BETA      ; Aici poate apare împrumut
MOV  word ptr REZ, AX
MOV  AX, word ptr ALFA+2    ; Cuvintele m.s. la adr. mari
SBB  AX, word ptr BETA+2    ; Se ia în considerare
                                ; împrumutul precedent
MOV  word ptr REZ+2, AX
    
```

### Instrucțiunea DEC (Decrement - Decrementează)

Are forma generală:

```
DEC  destinație
```

În care destinație este la fel ca la instrucțiunea INC. Semnificația este:

```
(destinație) <--- (destinație) - 1
```

*Flaguri afectate:* AF, PF, SF, ZF, OF (fără CF).

### Instrucțiunea NEG (Negate - Schimbă semnul)

Are forma generală:

```
NEG  destinație
```

În care destinație este un registru sau o locație de memorie, pe 8 sau pe 16 biți. Semnificația este:

```
(destinație) <--- 0 - (destinație)
```

deci se face o schimbare a semnului operandului.

*Flaguri afectate:* AF, PF, SF, ZF, OF (toate).

De observat că schimbarea semnului poate conduce uneori la aceeași valoare, în cazul depășirii domeniului admisibil. De exemplu, secvența:

```
MOV  AL, -128
NEG  AL
```

va lăsa registrul AL neschimbat (80H), deoarece 128 și -128 au aceeași reprezentare internă.

### Instrucțiunea CMP (Compare - Compară)

Are forma generală:

```
CMP  destinație, sursă
```

iar semnificația este execuția unei scăderi temporare (destinație) - (sursă), fără a se modifica vreun operand, dar cu poziționarea bistabililor de condiție.

*Flaguri afectate:* AF, PF, SF, ZF, OF (toate).

Testând bistabilii de condiție, putem deduce relația dintre cei doi operanzi. De exemplu, instrucțiunea CMP AX, BX va provoca o scădere temporară (AX) - (BX). Dacă ZF = 1, înseamnă că (AX) = (BX). Dacă CF = 1, înseamnă că la scăderea a apărut un împrumut, deci (AX) < (BX), dacă sunt considerate ca numere fără semn.

Instrucțiunea CMP se folosește de obicei împreună cu instrucțiuni de salt condiționat (vezi 2.4.3).

### Instrucțiunea DAS (Decimal Adjust for Subtraction - Corecție zecimală după scădere)

Instrucțiunea nu are operanzi și execută corecția zecimală a acumulatorului AL, după operații de scădere cu numere în format BCD împachetat. Semnificația este:

```

dacă (AL0:3) > 9 sau (AF) = 1, atunci {
    (AL) <--- (AL) - 6
    (AF) <--- 1
}
dacă acum (AL4:7) > 9 sau CF = 1, atunci {
    (AL) <--- (AL) - 60H
    (CF) <--- 1
}
    
```

*Flaguri afectate:* CF, AF, CF, PF, SF, ZF. Flagul OF este nedefinit.

Explicația operațiilor de mai sus este similară cu cea de la instrucțiunea DAA. De exemplu, în urma secvenței:

```

.code
    MOV    AL, 52H
    SUB    AL, 24H           ; AL = 2EH
    DAS                    ; AL = 28H
    
```

se obține în AL rezultatul corect 28H (28 = 52 - 24).

### Instrucțiunea AAS (ASCII Adjust for Subtraction - Corecție ASCII după scădere)

Instrucțiunea nu are operanzi și efectuează corecția acumulatorului AX, după operații de scădere cu operanzi BCD despachetați (o cifră BCD pe un octet). Semnificația este următoarea:

```

dacă (AL0:3) > 9 sau (AF) = 1, atunci {
    (AL) <--- (AL) - 6
    (AH) <--- (AH) - 1
    (AF) <--- 1
    (CF) <--- 1
    (AL) <--- (AL) AND 0FH
}
    
```

*Flaguri afectate:* AF, CF, restul nedefinite.

Se observă analogia cu instrucțiunea AAA, specifică adunării în format BCD despachetat.

#### 2.2.4 Instrucțiuni specifice înmulțirii (CBW, CWD, MUL, IMUL, AAM)

Operațiile de înmulțire se fac între acumulator și un al doilea operand. Rezultatul operației este pe 16 sau, respectiv, 32 de biți. Se folosesc următoarele noțiuni, definite diferit la operații de 8 sau 16 biți:

- acumulator - registrul AL, respectiv, AX
- acumulator extins - registrul AX, respectiv perechea de registre DX:AX
- extensia acumulatorului - registrul AH, respectiv DX
- extensia de semn a acumulatorului - conținutul registrului AL, respectiv AX, reprezentat, ca număr cu semn, pe o lungime dublă (16, respectiv 32 de biți).

**Instrucțiunea CBW (Convert Byte to Word - Convertește octet la cuvânt)**

Este fără operanzi și are următoarea semnificație:

```
dacă (AL7) = 0, atunci
    (AH) <--- 0
altfel
    (AH) <--- 1
```

*Flaguri afectate:* nici unul.

Practic, se extinde bitul de semn din AL la întreg registrul AH. Acest lucru este echivalent cu reprezentarea lui AL în complement față de 2, pe un număr dublu de biți. De exemplu, dacă AL = -3 (0FDH), atunci instrucțiunea CBW va forța în AX valoarea 0FFFDH, care este chiar reprezentarea în complement față de 2 a valorii - 3.

**Instrucțiunea CWD (Convert Word to DoubleWord - Convertește cuvânt la dublu-cuvânt)**

Este fără operanzi și are următoarea semnificație:

```
dacă (AX15) = 0, atunci
    (DX) <--- 0
altfel
    (DX) <--- 1
```

*Flaguri afectate:* nici unul.

Se extinde bitul de semn din AX la întreg registrul DX, obținându-se astfel o reprezentare a lui AX pe 32 de biți.

Prin instrucțiunile CBW și CWD, se obțin extensiile de semn ale acumulatorului în acumulatorul extins.

**Instrucțiunea MUL (Multiply - Înmulțește fără semn)**

Are forma generală:

```
MUL    sursă
```

În care sursă poate fi un registru sau o locație de memorie de 8 sau 16 biți. Variantele noi de procesoare acceptă ca și date imediate ca operand sursă. Rezultatul se obține pe un număr dublu de biți (16 sau 32). Semnificația este:

```
(acumulator extins) <--- (acumulator) * (sursă)
```

În care ambii operanzi se consideră numere fără semn.

Mai precis, dacă sursă este pe octet:

```
(AH:AL) <--- (AL) * (sursă)
```

iar dacă sursă este pe cuvânt:

```
(DX:AX) <--- (AX) * (sursă)
```

Flaguri afectate: dacă extensia acumulatorului (adică AH sau DX) este diferită de 0, atunci CF și OF sunt 1, altfel CF și OF sunt 0. Restul flagurilor sunt nedefinite. Exemple:

```
.data
ALFA    db 10H
```



```

        BETA    dw 200H
.code
        MOV     AL, 10H
        MUL     ALFA          ; (AX) <--- (AL) * ALFA
        MOV     AX, 20H
        MUL     BETA          ; (DX:AX) <--- (AX) * BETA
        MOV     AX, 100H
        MOV     BX, 20H
        MUL     BX            ; (DX:AX) <--- (AX) * (BX)
    
```

În situații în care un operand este de tip byte iar celălalt de tip word, se convertește operandul de tip byte la word, ca număr fără semn, deci cu partea mai semnificativă 0. De exemplu, pentru a înmulți valorile ALFA și BETA, se poate scrie:

```

        MOV     AL, ALFA
        MOV     AH, 0          ; (AX) = 0010H
        MUL     BETA          ; (DX:AX) = 2000H
    
```

Se observă că instrucțiunea MUL nu poate conduce la depășiri. Înmulțind fără semn cele mai mari valori posibile pe 8/16 biți se obțin valori corecte pe 16/32 de biți. De exemplu,  $(2^{16} - 1) * (2^{16} - 1) = 2^{32} - 2^{17} + 1$ , care este o valoare reprezentabilă pe 32 de biți.

### **Instrucțiunea IMUL (Integer Multiply - Înmulțește cu semn)**

Are forma generală:

```
IMUL    sursă
```

fiind similară cu instrucțiunea MUL. Deosebirea este că operația de înmulțire se face considerând opearanzii numere cu semn.

*Flaguri afectate:* dacă estensia acumulatorului reprezintă extensia de semn a acumulatorului, atunci CF și OF se poziționează la 0. Altfel, CF și OF devin 1. Restul flagurilor sunt nedefinite.

Nu pot apare situații de depășire. Înmulțind cele mai mari valori în modul, se obțin valori corect reprezentabile. De exemplu,  $127 * 127 = 16129$ , care se reprezintă corect ca număr cu semn pe 16 biți. Similar,  $(-128) * (-128) = 16384$ .

Dacă sunt necesare conversii de la byte la word, se va folosi instrucțiunea CBW, ca în secvența următoare:

```

.data
        ALFA    db -113
        BETA    dw -147
        REZ     dd ?
.code
        MOV     AL, ALFA
        CBW          ; Conversie la word
        MUL     BETA    ; Înmulțire pe 16 biți
        MOV     word ptr REZ, AX ; Partea m.p.s. la adrese mici
        MOV     word ptr REZ+2, DX ; Partea m.s. la adrese mari
    
```

**Instrucțiunea AAM (ASCII Adjust for Multiply - Corecție ASCII după înmulțire)**

Instrucțiunea nu are operanzi și efectuează o corecție a acumulatorului AX, după o înmulțire pe 8biți cu numere în format BCD despachetat. Semnificația este:

```
(AH) <-- (AL)/10
(AL) <-- (AL) MOD 10
```

*Flaguri afectate:* PF, SF, ZF, restul nedefinite.

De exemplu, să considerăm secvența:

```
MOV AL, 5
MOV CL, 9
MUL CL
AAM
```

În urma înmulțirii, registrul AX va conține valoarea 2DH (45). Corecția prin AAM conduce la AH = 4 și AL = 5, deci AX = 0405H, adică reprezentarea BCD despachetat pentru valoarea zecimală 45.

**2.2.5 Instrucțiuni specifice împărțirii (DIV, IDIV, AAD)**

Împărțirea presupune că deîmpărțitul este pe o lungime dublă decât împărțitorul. Prin împărțire se obțin câtul și restul, se lungime egală cu a împărțitorului.

**Instrucțiunea DIV (Divide - Împarte fără semn)**

Are forma generală:

```
DIV sursă
```

În care sursă e un operand (registru sau locație de memorie) pe octet sau pe cuvânt. Procesoarele moderne acceptă și date imediate ca operand sursă. Semnificația este următoarea:

```
(acumulator) <--- (acumulator extins) / (sursă)
(extensia acumulatorului) <--- (acumulator extins) MOD (sursă)
```

Detaliind operațiile, se obține, în cazul în care sursă este pe octet:

```
(AL) <-- (AX) / (sursă)
(AH) <-- (AX) MOD (sursă)
```

iar dacă sursă este pe cuvânt:

```
(AX) <--- (DX:AX) / (sursă)
(DX) <--- (DX:AX) MOD (sursă)
```

*Flaguri afectate:* toate flagurile sunt nedefinite.

Împărțirea poate conduce la depășiri. În situația în care câtul rezultă mai mare decât valoarea maximă reprezentabilă pe 8, respectiv 16 biți, sau dacă împărțitorul este 0, rezultatele sunt nedefinite și se generează o întrerupere soft pe nivelul 0 (Divide Overflow - Depășire la împărțire). De exemplu, în secvența:

```
MOV AX, 1000
MOV BL, 2
DIV BL
```

câtul ar trebui să fie 500, valoare care nu se poate reprezenta pe un octet. Ca atare, apare depășire și se va genera o întrerupere pe nivelul 0. Rutina afectată acestui nivel de întrerupere oprește de obicei programul executabil și afișează un mesaj de eroare la consolă.

Să considerăm câteva exemple de operații de împărțire fără semn, care ilustrează pregătirea operanzilor în situațiile care nu corespund celor două tipuri de bază (împărțire cuvânt la octet și împărțire dublu-cuvânt la cuvânt):

```
.data
    B1      db ?
    B2      db ?
    W1      dw ?
    W2      dw ?
    D1      dd ?

.code
; Împărțire octet la octet
    MOV     AL, B1
    MOV     AH, 0
    DIV     B2                ; AL = cât, AH = rest
;
; Împărțire cuvânt la octet
    MOV     AX, W1
    DIV     B1                ; AL = cât, AH = rest
;
; Împărțire dublu-cuvânt la cuvânt
    MOV     AX, word ptr D1
    MOV     DX, word ptr D1 + 2
    DIV     W1                ; AX = cât, DX = rest
;
; Împărțire cuvânt la cuvânt
    MOV     AX, W1
    MOV     DX, 0
    DIV     W2                ; AX = cât, DX = rest
;
; Împărțire dublu-cuvânt la byte
    MOV     AX, word ptr D1
    MOV     DX, word ptr D1 + 2
    MOV     BL, B1
    MOV     BH, 0
    DIV     BX                ; AX = cât, DX = rest
```

### **Instrucțiunea IDIV (Integer Divide - Împarte cu semn)**

Are forma generală:

```
IDIV  sursă
```

în care sursă este la fel ca la instrucțiunea DIV. Semnificația este aceeași ca la instrucțiunea DIV, cu diferența că operanzii sunt considerați cu semn, iar împărțirea se face cu semn.

*Flaguri afectate:* toate flagurile sunt nedefinite.

În situația în care câtul rezultă în afara domeniului reprezentabil pe 8, respectiv 16 biți, sau dacă împărțitorul este 0, rezultatele sunt nedefinite și se generează o întrerupere soft pe nivelul 0. Concret, câtul trebuie să fie în domeniul [-128, +127] la împărțire pe octet, respectiv în domeniul [-32768, +32767] la împărțire pe cuvânt. De exemplu, în secvența:

```
MOV  AX, 400
MOV  BL, 2
IDIV BL
```

câtul ar trebuie să rezulte 200, valoare care nu se poate reprezenta ca număr cu semn pe un octet. Ca atare, va apare o întrerupere pe nivelul 0.

În ceea ce privește calculul restului la împărțirea cu semn, acesta este totdeauna calculat astfel încât să fie îndeplinită identitatea următoare:

$$a = b \cdot (a/b) + a \text{ MOD } b$$

în care  $a/b$  și  $a \text{ MOD } b$  sunt câtul și restul calculate de instrucțiunea IDIV. De exemplu, în secvența:

```
MOV  AX, -10
MOV  BL, -3
DIV  BL ; AL = 3, AH = 0FFH
```

câtul rezultă +3, iar identitatea de mai sus conduce la restul - 1. Așadar  $(-10) \text{ MOD } (-3) = -1$  și în registrul AH se va găsi 0FFH (-1 reprezentat pe un octet).

Tipurile posibile de împărțiri sunt aceleași ca la instrucțiunea DIV, cu observația că operanzii trebuie pregătiți folosind instrucțiunile CBW și CWD. Considerăm aceleași definiții de date ca la instrucțiunea DIV:

```
.code
; Împărțire octet la octet
    MOV AL, B1
    CBW
    IDIV B2                ; AL = cât, AH = rest
;
; Împărțire cuvânt la octet
    MOV AX, W1
    IDIV B1                ; AL = cât, AH = rest
;
; Împărțire dublu-cuvânt la cuvânt
    MOV AX, word ptr D1
    MOV DX, word ptr D1 + 2
    IDIV W1                ; AX = cât, DX = rest
;
; Împărțire cuvânt la cuvânt
    MOV AX, W1
    CWD
    IDIV W2                ; AX = cât, DX = rest
;
; Împărțire dublu-cuvânt la byte
    MOV AL, B1
    CBW
    MOV BX, AX              ; BX = Deîmpărțit
    MOV AX, word ptr D1
    MOV DX, word ptr D1 + 2 ; DX:AX = împărțitor
    IDIV BX                ; AX = cât, DX = rest
```

O aplicație tipică a instrucțiunilor de împărțire este conversia unui cuvânt de 16 biți, cu sau fără semn, la un șir de caractere (cifre) în baza 10. Dacă presupunem că *n* este variabila care conține numărul iar *sr* este variabila unde se depun cifrele generate, algoritmul poate fi descris (într-o notație specifică limbajului C), în felul următor:

```

adrsir = sir;
do {
    rest = n % 10;
    n = n/10;
    *sir++ = rest + '0';
} while (n != 0);
*sir = 0;
inverseaza(adrsir);
    
```

Interpretăm numărul  $n$  fără semn. În bucla de program cu test la partea inferioară, se calculează restul și câtul împărțirii lui  $n$  la 10. Câtul furnizează cifra curentă, căreia i se adună codul ASCII al cifrei 0. Se avansează adresa `sir` la următorul caracter, bucla terminându-se când  $n$  devine 0.

Se observă că acest algoritm furnizează cifrele numărului în ordine inversă; ca atare, rutina `inverseaza(sir)`, trebui să inverseze ordinea caracterelor din `sir`ul generat. De exemplu, dacă  $n = 1234$ , atunci prima iterație produce  $rest = 4$  și  $n = 123$ , a doua iterație produce  $rest = 3$  și  $n = 12$  etc.

Să implementăm acest algoritm printr-o secvență în limbaj de asamblare.

```

.data
    n      dw ?
    sir     db 20 dup (?)

.code
    lea     si, sir
    mov     ax, n
    mov     dx, 0
    mov     bl, 10
    mov     bh, 0

_do:
    div     bx                ; AX = cât, DX (DL) = rest
    add     dl, '0'
    mov     [si], dl
    inc     si
    mov     dx, 0
    cmp     ax, 0
    jne     _do              ; Salt dacă AX diferit de 0
    mov     byte ptr [si], 0

;
; Inverseaza
;
    mov     di, si
    dec     di                ; DI indică ultimul caracter util
    lea     si, sir           ; SI indică primul caracter

_test:
    cmp     si, di
    jae     _gata            ; Salt dacă SI >= DI
    mov     al, [si]         ; Interschimbă octeții de la
    xchg    al, [di]         ; adresele din SI și DI
    mov     [si], al
    inc     si
    dec     di
    jmp     _test

_gata:
    
```

Declarația **sir db 20 dup (?)** rezervă o zonă de 20 de octeți în segmentul de date. Simbolurile **\_do:**, **\_test:**, **\_gata:** sunt etichete folosite la instrucțiuni de salt.

Deși avem de făcut o împărțire la 10 (deci la o cantitate care se poate reprezenta pe un octet), iar deîmpărțitul este un cuvânt, vom folosi totuși împărțire dublu-cuvânt la cuvânt, pentru a evita depășirile. Se pregătește deci deîmpărțitul în DX:AX și câțul în BX. Pentru accesul la sirul de caractere folosim registrul SI. Așadar, variabilele n și sir din algoritm vor fi ținute în registrele AX și SI.

Prin împărțire se obține câțul în AX și restul în DX. De fapt, fiind o împărțire la 10, restul nu poate depăși valoarea 9, deci îl vom găsi de fapt în DL. Adăugăm '0' și depunem caracterul la adresa dată de SI, incrementăm SI și pregătim deîmpărțitul pentru pasul următor (adică forțăm DX = 0). Se testează acum AX (adică variabila n), prin comparație cu 0. Instrucțiunea JNZ înseamnă "Jump on Not Equal", deci salt dacă operandii din ultima comparație sunt diferiți. Acest salt condiționat reia bucla de program.

Pentru a inversa caracterele generate, se poziționează registrele SI și DI pe primul și, respectiv, pe ultimul caracter util din șir, după care se execută o buclă de inversare, care continuă cât timp SI < DI. Instrucțiunea JAE înseamnă "Jump if Above or Equal",adică, deci salt dacă la ultima comparație, primul operand a fost mai mare sau egal decât al doilea. În bucla respectivă, se interschimbă caracterele de la adresele DI și SI, prin două MOV-uri și un XCHG.

### Instrucțiunea AAD (ASCII Adjust for Division - Corecție ASCII înainte de împărțire)

Instrucțiune nu are operandi și efectuează o corecție a acumulatorului AX, interpretat ca două cifre BCD despachetate. Semnificația este următoarea:

```
(AL) <--- (AH) * 10 + (AL)
(AH) <--- 0
```

*Flaguri afectate:* PF, SF și ZF, restul nedefinite.

Operația de corecție trebuie făcută înainte de împărțirea unui număr pe două cifre BCD, reprezentat pe un cuvânt, la o cifră BCD reprezentată pe un octet.

De exemplu, dacă AX = 0305H, adică valoarea BCD 35 și BL = 2, secvența de împărțire va fi:

```
AAD          ; AX = 23H
DIV BL       ; AL = 11H, AH = 1
```

Ininstrucțiunea AAD încarcă AX cu valoarea 23H (35) și împărțirea se face corect, obținându-se câțul 11H (17) și restul 1.

### 2.2.6 Instrucțiuni logice (NOT, AND, TEST, OR, XOR)

Instrucțiunile logice realizează funcțiile logice de bază, pe octet sau pe cuvânt. Operațiile se fac la nivel de bit, deci se aplică funcția logică respectivă tuturor biților sau perechilor de biți corespunzători din operandi.

Instrucțiunea NOT re un singur operand celelalte având fiecare doi operanzi.

### **Instrucțiunea NOT (Not - Negare logică bit cu bit)**

Forma generală este:

NOT    destinație

În care destinație poate fi un registru sau o locație de memorie de 8 sau 16 biți. Instrucțiunea provoacă negarea tuturor biților operandului, deci calculul complementului față de 1.

*Flaguri afectate:* nici unul.

### **Instrucțiunea AND (And - Și logic bit cu bit)**

Are forma generală:

AND    destinație, sursă

În care destinație poate fi un registru sau o locație de memorie de 8 sau 16 biți, iar sursă poate fi un registru, o locație de memorie sau o constantă, pe 8 sau 16 biți. Semnificația este:

(destinație) <--- (destinație) AND (sursă)

*Flaguri afectate:* SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

### **Instrucțiunea TEST (Test - Testează)**

Are forma generală:

TEST    destinație, sursă

În care destinație și sursă sunt la fel ca la instrucțiunea AND. Efectul este execuția unei operații AND între cei doi operanzi, fără a se modifica destinație, dat cu poziționarea flagurilor la fel ca la instrucțiunea AND.

*Flaguri afectate:* SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Această instrucțiune și raportul ei cu instrucțiunea AND sunt similare cu instrucțiunea CMP și raportul cu instrucțiunea SUB.

### **Instrucțiunea OR (Or - Sau logic bit cu bit)**

Are forma generală:

OR    destinație, sursă

În care destinație și sursă sunt la fel ca la instrucțiunea AND. Semnificația este:

(destinație) <--- (destinație) OR (sursă)

*Flaguri afectate:* SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

### **Instrucțiunea XOR (Exclusive Or - Sau-exclusiv bit cu bit)**

Are forma generală:

XOR    destinație, sursă

În care destinație și sursă sunt la fel ca la instrucțiunea AND. Semnificația este:

(destinație) <--- (destinație) XOR (sursă)

*Flaguri afectate:* SF, ZF, PF, CF = 0, OF = 0, AF nedefinit.

Funcția sau-exclusiv este 1 când operandii săi sunt unul 0 iar celălalt 1 și este 0 când operandii sunt ambii 0 sau ambii 1. Din acest motiv, funcția sau-exclusiv se mai numește și anticoincidență.

Instrucțiunile logice sunt folosite frecvent pentru anumite operații tipice, cum ar fi:

- ștergerea rapidă a unui registru, cu poziționarea flagurilor:

```
XOR  AX, AX
XOR  CL, CL
```

- forțarea unor biți la valoarea 1, restul rămânând neschimbați

```
MASCA EQU 01101101B
OR     AL, MASCA
```

Pseudoinstrucțiunea EQU definește constante simbolice, iar prefixul B înseamnă număr scris în baza 2. În secvența de mai sus, biții cu valoarea 1 din MASCA vor fi forțați la 1 în registrul AL, iar biții cu valoarea 0 din MASCA vor rămâne neschimbați.

- forțarea unor biți la valoarea 0 cu ceilalți neschimbați

```
AND    AL, MASCA
```

Biții cu valoarea 0 din MASCA vor deveni 0 în AL, iar cei cu valoarea 1 în MASCA vor rămâne neschimbați.

- testarea unui singur bit dintr-un operand

```
TEST  AL, 01000000B
JZ     etichetă          ; sau JNZ
```

- poziționarea flagurilor fără a modifica operandul

```
OR     AX, AX           ; Poziționează FLAGS conform AX
AND    AX, AX           ; Similar
TEST   AX, AX           ; Similar
```

- complementarea unui grup de biți, cu ceilalți neschimbați

Presupunem constanta MASCA, definită ca mai sus, și operandul aflat în AL. Dorim ca biții cu valoarea 1 din MASCA să fie complementați în AL, iar ceilalți să rămână neschimbați.

```
MASCA EQU 01101101B
MOV    BL, AL           ; Salvare
AND    AL, MASCA        ; Selecție biți care se modifică
NOT    AL, MASCA        ; Complementare
XCHG   AL, BL           ; Refacere
AND    AL, NOT MASCA    ; Selecție biți care nu se modifică
OR     AL, BL           ; Rezultat final
```

Expresia NOT MASCA este evaluată la asamblare, producându-se o constantă cu toți biții negați.

### **2.2.7 Instrucțiuni de deplasare (SHL, SAL, SHR, SAR) și de rotație (ROL, RCL, ROR, RCR)**

Acest grup de instrucțiuni realizează operații de deplasare și de rotație la nivel de bit. Instrucțiunile au doi operanzi: primul este operandul propriu-zis, iar al doilea este numărul de biți cu care se deplasează sau se rotește primul operand. Ambele operații se pot face la dreapta sau la



stânga. Deplasare înseamnă translatarea tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în dreapta/stânga și cu pierderea biților din stânga/dreapta. Deplasarea cu un bit la stânga este echivalentă cu înmulțirea operandului cu 2, iar deplasarea la dreapta, cu împărțirea operandului la 2.

Rotație înseamnă translatarea tuturor biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă.

Ambele operații se fac cu modificarea bistabilului CF, care poate chiar participa la operațiile de rotație.

Forma generală a instrucțiunilor este:

OPERAȚIE    OPERAND, CONTOR

În care OPERAND este un registru sau o locație de memorie de 8 sau 16 biți, iar CONTOR (numărul de biți), este fie constanta 1, fie registrul CL, care conține numărul de biți cu care se deplasează/rotește operandul. Procesoarele de generație nouă (80286 și peste) acceptă un număr oarecare de biți, specificat și printr-o constantă întreagă.

Flagurile sunt afectate în felul următor. La operațiile de deplasare, se modifică toate flagurile conform rezultatului, în afară de AF, care este nedefinit. La operațiile de rotație, se modifică numai CF și OF.

Modificarea flagului OF se face printr-un algoritm destul de complicat. Pentru a nu repeta acest algoritm, considerăm următoarele secvențe definite în pseudo-cod:

```
pozit_OF_left {
    dacă CONTOR = 1, atunci
        dacă b.c.m.s. din OPERAND este diferit de CF, atunci
            OF <--- 1
        altfel
            OF <--- 0
    altfel
        OF nedefinit
}

pozit_OF_right {
    dacă CONTOR = 1, atunci
        dacă cei doi biți mai semnif. din OPERAND sunt diferiți
            OF <--- 1
        altfel
            OF <--- 0
    altfel
        OF nedefinit
}
```

Se vede deci că flagul OF este poziționat numai dacă se face o operație de deplasare/rotație de 1 bit.

Descrierea instrucțiunilor se va face într-un format de tip pseudo-cod, cu evidențierea operațiilor aritmetice echivalente.

La instrucțiunile de deplasare, se consideră deplasări logice și aritmetice, care se pot utiliza după natura operanzilor.

**Instrucțiunea SHL/SAL (Shift Logic/Arithmetic Left - Deplasează logic/aritmetic la stânga)**

Are forma generală:

```
SHL/SAL    OPERAND, CONTOR
```

Deși există două mnemonice (SHL și SAL), în fapt este vorba de o unică instrucțiune. Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    CF <--- b.c.m.s din OPR
    OPR <--- 2*OPR (operație fără semn)
    temp <--- temp - 1
}
pozit_OF_left
```

Această descriere nu spune altceva decât că bitul cel mai semnificativ al operandului trece în CF, după care toți biții se deplasează la stânga cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

**Instrucțiunea SHR (Shift Logic Right - Deplasează logic la dreapta)**

Are forma generală:

```
SHR    OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    CF <--- b.c.m.p.s din OPR
    OPR <--- OPR/2 (operație fără semn)
    temp <--- temp - 1
}
pozit_OF_right
```

Descrierea de mai sus spune că bitul cel mai puțin semnificativ din OPERAND trece în CF, după care se deplasează toți biții cu o poziție la dreapta (împărțire la 2). Faptul că operația de împărțire se execută fără semn înseamnă că se completează cu un bit 0 dinspre stânga (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

**Instrucțiunea SAR (Shift Arithmetic Right - Deplasează aritmetic la dreapta)**

Are forma generală:

```
SAR    OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    CF <--- b.c.m.p.s din OPR
    OPR <--- OPR/2 (operație cu semn)
    temp <--- temp - 1
}
pozit_OF_right
```

Singura diferență față de deplasarea logică la dreapta este realizarea împărțirii la 2, luând în considerare semnul operandului. Aceasta înseamnă că se conservă bitul de semn, mai precis, completarea dinspre stânga se face cu bitul de semn (vezi Figura 2.3).

Aceste două tipuri de deplasare la dreapta se reflectă și în implementările limbajelor de nivel înalt. De exemplu, standardul ANSI al limbajului C lasă neprecizat faptul că deplasarea la dreapta a unui întreg de tip signed se face cu completare dinspre stânga cu 0 sau cu bitul de semn (deci operație de tip SHR sau SAR); această alegere revine implementării. Totuși, la deplasarea cantităților de tip unsigned, se completează întotdeauna dinspre stânga cu 0 (deci se face deplasare logică). Acest fapt este important mai ales în cazul tipului char, care nu este specificat în standardul ANSI ca fiind cu semn sau fără semn.

### **Instrucțiunea ROL (Rotate Left - Rotește la stânga)**

Are forma generală:

```
ROL    OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    CF <--- b.c.m.s. din OPERAND
    OPR <--- 2*OPERAND + CF
    temp <--- temp - 1
}
pozit_OF_left
```

Descrierea de mai sus spune că bitul cel mai semnificativ din OPERAND trece atât în CF, cât și în bitul cel mai puțin semnificativ din OPERAND, după ce toți biții acestuia s-au deplasat la stânga cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

### **Instrucțiunea RCL (Rotate Left through Carry - Rotește la stânga prin carry)**

Are forma generală:

```
RCL    OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    temp_cf <--- CF
    CF <--- b.c.m.s. din OPERAND
    OPR <--- 2*OPERAND + temp_cf
    temp <--- temp - 1
}
pozit_OF_left
```

Descrierea de mai sus spune că bitul cel mai semnificativ din OPERAND trece în CF, se deplasează toți biții din OPERAND cu o

poziție la stânga, iar CF original trece în bitul cel mai puțin semnificativ din OPERAND. Cu alte cuvinte, CF participă efectiv la rotație (vezi Figura 2.3).

Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

### Instrucțiunea ROR (Rotate Right - Rotește la dreapta)

Are forma generală:

```
ROR  OPERAND, CONTOR
```

Semnificația este:

```
temp <--- CONTOR
cât timp temp != 0 {
    CF <--- b.c.m.p.s din OPERAND
    OPR <--- OPR/2 (operație fără semn)
    b.c.m.s. din OPERAND <--- CF
    temp <--- temp - 1
}
pozit_OF_right
```

Descriere de mai sus spune că bitul cel mai semnificativ din OPERAND trece atât în CF, cât și în bitul cel mai puțin semnificativ, după ce toți biții s-au deplasat la dreapta cu o poziție (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

### Instrucțiunea RCR (Rotate Right through Carry - Rotește la dreapta prin carry)

Are forma generală:

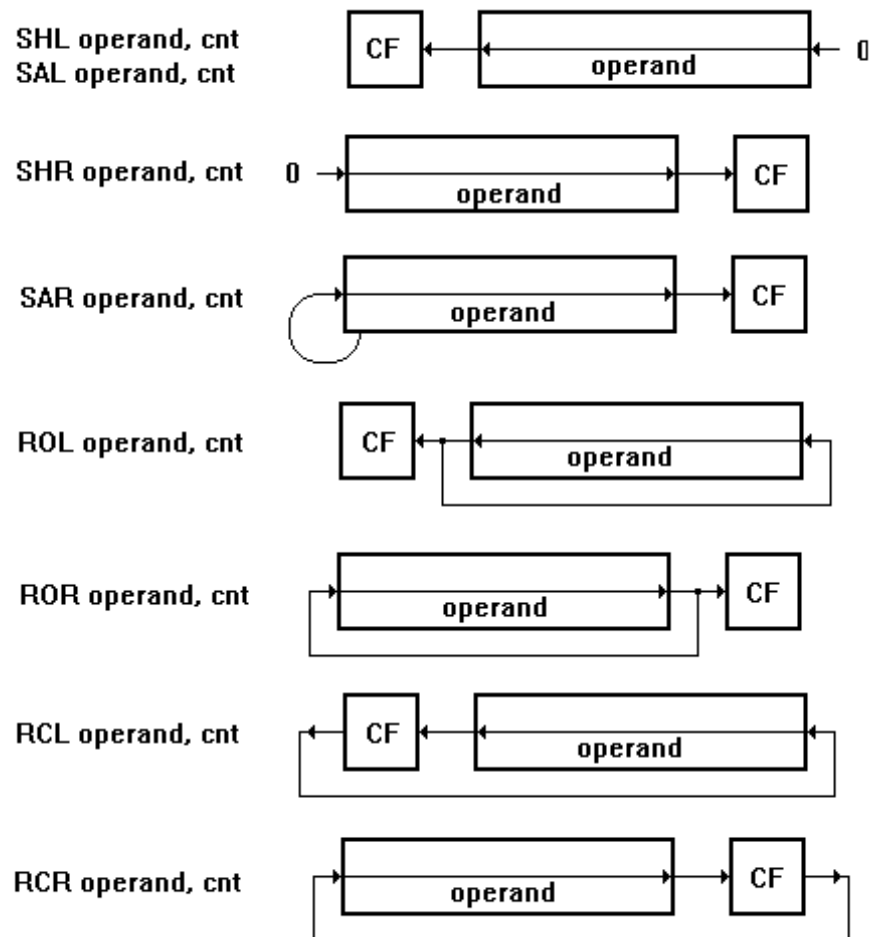
```
RCR  OPERAND, CONTOR
```

Semnificația este următoarea:

```
temp <--- CONTOR
cât timp temp != 0 {
    temp_cf <--- CF
    CF <--- b.c.m.p.s. din OPERAND
    OPR <--- OPR/2 (operație fără semn)
    b.c.m.s. din OPERAND <--- temp_cf
    temp <--- temp - 1
}
pozit_OF_right
```

Descrierea de mai sus spune că bitul cel mai puțin semnificativ din OPERAND trece în CF, se deplasează toți biții din OPERAND cu o poziție la dreapta, iar CF original trece în bitul cel mai semnificativ din OPERAND. Cu alte cuvinte, CF participă efectiv la rotație (vezi Figura 2.3). Operația se repetă de atâtea ori cât este valoarea lui CONTOR (1 sau conținutul registrului CL).

Semnificațiile celor 7 instrucțiuni de deplasare și rotație sunt ilustrate în Figura 2.3.



**Figura 2.3 Semnificația operațiilor de deplasare și rotație**

Să considerăm câteva exemple de rotații și deplasări.

Înmulțirea/împărțirea cu puteri ale lui 2. Operațiile de asemenea tip se execută mult mai eficient prin deplasări la stânga/dreapta. Secvența:

```
MOV CL, 4
MOV AH, 0
SHL AX, CL
```

realizează înmulțirea lui AL cu valoarea 16. Rezultatul se regăsește în AX. Secvența:

```
MOV CL, 3
SAL BX, CL
```

realizează împărțirea lui BX (considerat număr cu semn) la valoarea 8.

Similar se realizează și înmulțiri cu valori care se pot exprima prin sume cu un număr redus de termeni de puteri ale lui 2. Secvența următoare realizează înmulțirea unei valori N presupuse inițial în AL cu valoarea 13, prin deplasări și adunări repetate. Rezultatul se regăsește în BX.

```
MOV AH, 0
MOV BX, AX                ; Salvare N
MOV DX, AX                ; Salvare N
MOV CL, 3
SHL AX, 3                 ; AX <--- N * 8
ADD BX, AX                 ; BX <--- N * 8 + N
MOV AX, DX                ; Refacere N
```

```
MOV    CL, 2
SHL    AX, CL           ; AX <--- N * 4
ADD    BX, AX           ; BX <--- N * 8 + N * 4 + N
```

### 2.3. Instrucțiuni pentru operații cu șiruri de caractere/cuvinte

Acest grup de instrucțiuni implementează un set puternic de operații cu șiruri de octeți sau de cuvinte. Prin șir se înțelege o secvență de octeți sau de cuvinte, aflate la adrese succesive de memorie. Setul de instrucțiuni cuprinde operații primitive (la nivel de un octet sau un cuvânt) și prefixe de repetare, care pot realiza repetarea operațiilor primitive de un număr fix de ori sau până la îndeplinirea unei condiții de tip comparație.

#### 2.3.1 Operații primitive

Operațiile primitive se grupează în patru categorii, fiecare putând fi pe octet sau pe cuvânt. Denumirea instrucțiunilor la nivel de octet se termină cu litera B, iar a celor la nivel de cuvânt cu litera W. Aceste instrucțiuni nu au operanzi.

- Move String (Copiază șir) - MOVSB, MOVSW
- Compare String (Compară șiruri) - CMPSB, CMPSW
- Load String (Încarcă șir în AL/AX) - LODSB, LODSW
- Store String (Depune AL/AX în șir) - STOSB, STOSW
- Scan String (Compară șir cu AL/AX) - SCASB, SCASW

Toate operațiile primitive folosesc registrele DS:SI ca adresă sursă și/sau ES:DI ca adresă destinație. De asemenea, toate operațiile primitive actualizează adresele implicate în operație, în felul următor:

- dacă flagul DF = 0, adresele (registrul SI și/sau DI) sunt incrementate cu 1 sau cu 2, după cum operația primitivă este la nivel de octet sau de cuvânt;
- dacă flagul DF = 1, adresele (registrul SI și/sau DI) sunt decrementate cu 1 sau cu 2, după cum operația primitivă este la nivel de octet sau de cuvânt.

Vom utiliza notațiile:

```
(SI) <-- (SI) + delta
(DI) <-- (DI) + delta
```

unde delta este +-1 sau +-2, după starea bistabilului DF sau a tipului operației (octet sau cuvânt).

Starea flagului DF poate fi controlată prin instrucțiunile (fără operanzi) CLD (Clear Direction) și STD (Set Direction), care șterg, respectiv setează acest bistabil.

#### Instrucțiunile MOVSB, MOVSW

Semnificația este:

```
((ES:DI) <--- ((DS:SI))
(SI) <-- (SI) + delta
(DI) <-- (DI) + delta
```

Se transferă deci un octet (MOVSB) sau un cuvânt (MOVSW) de la adresa dată de (DS:SI) la adresa dată de (ES:DI), după care se actualizează adresele.

**Instrucțiunile CMPSB, CMPSW**

Semnificația este:

```
((DS:SI) - ((ES:DI))
(SI) <-- (SI) + delta
(DI) <-- (DI) + delta
```

Se calculează temporar (fără a se modifica vreun operand) diferența dintre octeții (cuvintele) de la adresele (DS:SI) și (ES:DI), după care se actualizează adresele. Bistabilii de condiție se poziționează conform operației de scădere. Instrucțiunile se folosesc la testarea egalității/inegalității șirurilor.

**Instrucțiunile LODSB, LODSW**

Semnificația este:

```
(acumulator) <--- ((DS:SI))
(SI) <--- (SI) + delta
```

Se încarcă deci în AL, respectiv AX, octetul, respectiv cuvântul de la adresa (DS:SI), după care se actualizează aceasta adresă.

**Instrucțiunile STOSB, STOSW**

Semnificația este:

```
((ES:DI)) <--- (acumulator)
(DI) <-- (DI) + delta
```

Se depune conținutul registrului AL, respectiv AX în octetul, respectiv cuvântul de la adresa (ES:DI), după care se actualizează această adresă.

**Instrucțiunile SCASB, SCASW**

Semnificația este:

```
(acumulator) - ((ES:DI))
(DI) <-- (DI) + delta
```

Se calculează temporar (fără a se modifica vreun operand) diferența dintre registrul AL, respectiv AX și octetul, respectiv cuvântul de la adresa (ES:DI), după care se actualizează această adresă. Bistabilii de condiție se poziționează conform operației de scădere. Instrucțiunile se folosesc la testarea căutarea unui anumit octet (cuvânt într-un șir).

Pe lângă formele fără operanzi, descrise mai sus, asamblorul mai recunoaște și forme în care operanzii apar explicit. Semnificația adreselor implicate în aceste instrucțiuni nu poate fi însă schimbată (se vore folosi tot registrele SI și DI). Singura rațiune pentru aceste forme este specificarea unui prefix de segment pentru adresa sursă. În acest caz mnemonica instrucțiunii se scrie fără litera B sau W de la sfârșit, dar este obligatorie specificarea tipului operației prin operatorul PTR. Nu se recomandă utilizarea acestor prefixe de segment din următoarele motive:

- nu se poate schimba registrul de segment al adresei destinație
- (acesta este tot timpul ES);
- în condițiile folosirii prefixelor de repetare (vezi 2.3.2), instrucțiunea rezultată ar avea atât prefix de repetare cât și prefix de segment, ceea ce poate conduce la funcționări defectuoase, într-un context în care pot apare întreruperi externe.

### 2.3.2 Prefixe de repetare (REP/REPE/REPZ, REPNE/REPNZ)

Prefixele de repetare permit execuția repetată a unei operații primitive cu șiruri de octeți sau cuvinte, funcție de un contor de operații sau de un contor și o condiție logică. Aceste prefixe nu sunt instrucțiuni în sine, ci participă la formarea unor instrucțiuni compuse, alături de operațiile primitive descrise mai sus.

#### Prefixul de repetare REP/REPE/REPZ (Repeat - Repetă)

Cele trei mnemonice identifică de fapt un unic prefix de repetare. Forma generală a unei instrucțiuni cu acest prefix este:

```
REP/REPE/REPZ op_primitivă
```

Semnificația este:

```
cât timp CX != 0 {
    tratează o eventuală întrerupere externă
    op_primitivă
    CX <--- CX - 1
    dacă op_primitivă este CMPS/SCAS și ZF = 0, se iese din buclă
}
```

Se vede deci că operația primitivă se execută de un număr maxim de ori dat de conținutul registrului CX. Dacă CX este inițial 0, operația nu se execută nici o dată. În cazul operațiilor primitive CMPS și SCAS (care poziționează ZF), se iese forțat din buclă dacă ZF = 0, adică dacă rezultatul operației primitive este nenul. Bucla se execută deci cât timp acest rezultat este 0.

De obicei, scrierea cu REP se folosește la primitivele de tip MOVS, LODS și STOS, iar scrierea cu REPE sau REPZ la primitivele de tip CMPS și SCAS.

Să considerăm două exemple.

Secvența de mai jos transferă 100 de octeți de la adresa SURSA la adresa DESTINATIE, ambele presupuse în segmentul curent adresat prin DS.

```
.data
    SURSADB 100 dup (?)
    DEST    db 100 dup (?)

.code
    CLD                      ; Direcție ascendentă
    MOV     AX, DS           ; Pregătire
    MOV     ES, AX           ; adrese
    LEA     SI, SURSADB      ; Adresă sursă
    LEA     DI, DEST         ; Adresă destinație
    MOV     CX, 100          ; Contor
    REP     MOVSB            ; Instrucțiune compusă
```

Secvența următoare identifică primul octet diferit de un octet dat dintr-un șir de lungime 200 de octeți.

```
.data
    SIR     DB 200 dup (?)

.code
    MOV     AX, DS
    MOV     ES, AX
    LEA     DI, SIR
    CLD
    MOV     AL, 'A'          ; Se caută primul octet diferit de 'A'
```



```
MOV    CX, 200           ; Număr maxim de iterații
REPE   SCASB             ; Buclă de căutare
```

Examinând bistabilul ZF la ieșirea din această secvență, putem deduce rezultatul căutării. Dacă  $ZF = 0$ , înseamnă că a avut loc o ieșire forțată din buclă, deci comparația a dat rezultatul 0. Registrul DI, decrementat cu o unitate, furnizează adresa primului octet diferit de octetul din AL. Dacă  $ZF = 1$ , înseamnă că toți octeții parcurși sunt identici cu octetul dat în AL.

La prima vedere, s-ar părea că, la fel, de bine, am putea examina conținutul registrului CX la ieșirea din buclă (0 sau diferit de 0), pentru a deduce dacă. Acest test este incorect, deoarece s-ar putea ca primul octet diferit de cel din AL să fie chiar ultimul. În acest caz se iese din buclă cu  $CX = 0$ , la fel că în situația în care toți octeții parcurși sunt identici cu cel din AL.

### **Prefixul de repetare REPNE/REPNZ (Repeat While Not Equal/Not Zero - Repetă cât timp diferit/diferit de zero)**

Cele două mnemonice identifică un singur prefix de repetare. Forma generală este:

```
REPNE/REPNZ operație_primitivă
```

Semnificația este:

```
cât timp CX != 0 {
    tratează o eventuală întrerupere externă
    op_primitivă
    CX <--- CX - 1
    dacă op_primitivă este CMPS/SCAS și ZF = 1, se iese din buclă
}
```

Diferența față de prefixul precedent este condiția de ieșire forțată din buclă: se iese dacă  $ZF = 1$ , deci dacă rezultatul operației primitive a fost 0. Ca atare, bucla se execută cât timp acest rezultat este nenul, dar nu mai mult de CX ori.

Practic, acest prefix de repetare se folosește numai cu operațiile primitive CMPS și SCAS. Pentru celelalte operații primitive, în care nu se ia în considerare bistabilul ZF, se preferă scrierea cu prefixul REP. La ieșirea din buclă, se poate examina bistabilul ZF, deducându-se dacă a fost sau nu o ieșire forțată.

Următoarea secvență determină ultimul caracter egal cu un caracter dat, prin parcurgerea șirului în sens invers.

```
.data
SIR    db 30 (?)

.code
LEA    DI, SIR
ADD    DI, 29
MOV    CX, 30
STD
MOV    AL, '?'           ; Se caută primul octet = 'x', de la
                        ; dreapta la stânga
REPNE SCASB
```

### 2.3.3 Operații complexe asupra șirurilor

Să considerăm unele operații complexe asupra șirurilor de caractere. Vom considera că șirurile au, ca ultimul caracter, octetul 0, pe post de terminator de șir.

#### *Compararea lexicografică a două șiruri*

Acest algoritm primește ca date de intrare adresele a două șiruri de caractere terminate cu 0 și întoarce diferența primilor octeți care diferă în cele două șiruri sau 0 dacă cele două șiruri coincid. Următoarea secvență de program întoarce în AL rezultatul cerut. (Presupunem că DS și ES sunt inițializate corespunzător.)

```
.data
    SIR_1 DB 'Un exemplu de sir terminat cu zero', 0
    SIR_2 DB 'Un exemplu de sir terminat cu octetul nul', 0

.code
    CLD
    LEA SI, SIR_1
    LEA DI, SIR_2

IAR:
    LODSB                    ; (AL) = (SIR_1)
    TEST AL, AL              ; Test terminator
    JZ GATA
    SCASB                    ; Compara (AL) cu (SIR_2)
    JE IAR                   ; Dacă sunt egale, se reia bucla
    DEC DI                   ; Dacă nu, se revine pe caracterul

GATA:
    SUB AL, ES:[DI]          ; Se face diferența
```

#### *Copierea unui șir în alt șir*

Următoarea secvență copiază un șir în alt șir. Copierea încetează după copierea terminatorului din șirul sursă. Se presupune că la adresa destinație se găsește suficient spațiu rezervat. Se presupune că registrele DS și ES indică segmentul curent de date.

```
.data
    SURSADB 'Un sir care trebuie copiat',0
    DEST DB 80 DUP (?)

.code
    LEA SI, SURSA
    LEA DI, DEST
    CLD

BUCLA:
    LODSB                    ; (AL) <--- (SURSA)
    STOSB                    ; (DEST) <--- (AL)
    TEST AL,AL               ; Test terminator
    JNZ BUCLA                ; Reluare
```

#### *Calculul lungimii unui șir*

Următoarea secvență determină în AX numărul de caractere utile din șirul SURSA. Terminatorul 0 nu se numără.

```
.code
    CLD
    LEA DI, SURSA
    MOV CX, 0FFFFH           ; Numărul maxim posibil
    MOV BX, DI                ; Salvare adresă început
    XOR AL, AL                ; Octet căutat
```

REP NZ SCASB	; Repetă cât timp e diferit de 0
DEC DI	; Înapoi pe terminator
SUB DI, BX	; Adresă terminator - Adresă de
MOV AX, DI	; Început = Lungime

#### **Copierea a N caractere dintr-un șir în alt șir**

Se copiază un număr de N caractere din șirul SURSA în șirul DEST. Dacă SURSA are mai puțin de N caractere, se completează DEST cu 0, până la N caractere.

```
.code
    CLD
    LEA SI, SURSA
    LEA DI, DEST
    MOV CX, N
    OR CX, CX
    JZ GATA
BUCLA:
    LODSB                ; Buclă de copiere
    STOSB                ; până la terminator
    DEC CX               ; inclusiv, dar nu mai
    JZ GATA              ; mult de CX
    TEST AL, AL           ; caractere
    JNZ BUCLA
    REP STOSB            ; Completare eventual cu 0
GATA:
```

Dacă N este 0, nu se copiază nimic. Se execută o buclă de copiere, în care se decrementează CX. Dacă CX = 0, totul se termină. Dacă s-a copiat terminatorul, se iese din bucla de copiere. În acest moment, AL = 0 și, dacă CX > 0, se execută depunerea lui AL în DEST, de atâtea ori cât este valoarea curentă a lui CX.

### **2.4 Instrucțiuni de apel de procedură și de salt (CALL, RET, JMP)**

Procedurile se definesc în textul sursă după șablonul:

```
nume_proc PROC [ FAR | NEAR ]
    .
    .
    .
    RET
nume_proc ENDP
```

unde `nume_proc` este numele procedurii, iar parametrii FAR sau NEAR (opționali) indică tipul procedurii.

Procedurile sunt de două tipuri: FAR și NEAR. O procedură FAR poate fi apelată și din alte segmente de cod decât cel în care este definită, pe când o procedură NEAR poate fi apelată numai din segmentul de cod în care este definită.

Dacă se omit parametrii FAR sau NEAR, tipul procedurii este dedus din directivele simplificate de definire a segmentelor (modelul de memorie folosit). De exemplu, modelul LARGE presupune că toate procedurile sunt implicit de tip FAR.

În mod corespunzător, există apeluri de tip FAR, respectiv NEAR, precum și instrucțiuni de revenire de tip FAR, respectiv NEAR.

Instrucțiunea RET (Return) provoacă revenirea în programul apelant. Putem scrie o instrucțiune return explicită, în formele RETN (Return Near) sau RETF (Return Far), sau RET pur și simplu, caz în care tipul instrucțiunii Return este dedus din tipul procedurii (FAR sau NEAR).

### **2.4.1 Apelul procedurilor și revenirea din proceduri**

#### **Instrucțiunea CALL (Apel de procedură)**

Poate avea una din formele:

- CALL nume\_proc
- CALL FAR PTR nume\_proc
- CALL NEAR PTR nume\_proc

În primul caz, tipul apelului este dedus din tipul procedurii, iar în celelalte, este specificat explicit (FAR sau NEAR).

Tipul apelului trebuie să coincidă cu tipul procedurii și cu tipul instrucțiunilor Return din interiorul procedurii, altfel se ajunge la funcționări defectuoase ale programului.

Semnificația instrucțiunii CALL este următoarea:

```
CALL de tip NEAR
  (SP) <--- (SP) - 2
  SS:((SP) + 1 : (SP)) <-- (IP)
  (IP) <--- offset-ul primei instrucțiuni din procedură
```

Descrierea de mai sus spune că se salvează în stivă contorul program curent, într-o manieră similară instrucțiunii PUSH (se decrementează registrul SP cu 2 și se înscrie conținutul lui IP în vârful stivei). Registrul IP conține totdeauna adresa instrucțiunii care urmează (în memorie), după instrucțiunea care se execută în mod curent. Practic, se salvează în stivă adresa instrucțiunii de după instrucțiunea CALL. Această adresă este numită adresă de revenire.

După această salvare, se încarcă în IP adresa (deplasamentul) primei instrucțiuni din procedură, ceea ce înseamnă un transfera al controlului către procedură.

Este important de reținut că, în momentul intrării în procedură, în vârful stivei există adresa de revenire. Această adresă nu trebuie modificată în nici un fel, în caz contrar, revenirea în programul apelant nemaifiind posibilă.

De observat că, în secvența de apel a procedurii, registrul CS nu se modifică, ceea ce înseamnă că se rămâne în același segment de cod.

```
CALL de tip FAR
  (SP) <--- (SP) - 2
  SS:((SP) + 1 : (SP)) <-- (CS)
  (SP) <--- (SP) - 2
  SS:((SP) + 1 : (SP)) <-- (IP)
  (CS) <--- adresa de segment a primei instrucțiuni din procedură
  (IP) <--- offset-ul primei instrucțiuni din procedură
```

Ceea ce este diferit față de apelul de tip NEAR este faptul că se salvează adresa completă de revenire (pe 32 de biți), prin plasarea în stivă atât a

regitrului IP cât și a registrului CS. Similar, transferul controlului se face prin modificarea explicită perechii de registre (CS:IP).

De observat că instrucțiunea CALL de tip FAR este ua din puținele instrucțiuni care modifică explicit registrul CS.

### **Instrucțiunea RET (Return - Revenire din procedură)**

Există Return de tip FAR și Return de tip NEAR. Formele posibile ale instrucțiunii sunt:

- RETN [ N ]
- RETF [ N ]
- RET [ N ]

În care parantezele drepte spun că N este o constantă întreagă opțională.

În cea de-a treia formă, tipul instrucțiunii (NEAR sau FAR) este dedus din tipul procedurii.

Semnificația este următoarea:

```
Return de tip NEAR
(IP) <--- SS:((SP) + 1 : (SP))
(SP) <--- (SP) + 2
[ (SP) <--- (SP) + N ]
```

Se observă că se reface registrul (IP), prin copierea vârfului stivei și incrementarea registrului SP cu 2. Dacă SP are aceeași valoare ca la intrarea în procedură și conținutul stivei nu a fost alterat între timp, atunci se copiază practic în IP adresa de revenire, ceea ce provoacă transferul controlului la instrucțiunea care urmează instrucțiunii CALL care a provocat apelul procedurii.

Dacă în formatul instrucțiunii RET există constanta opțională N, atunci se adună această constantă la registrul SP. Acest tip de Return se numește Return cu descărcarea stivei.

```
Return de tip FAR
(IP) <--- SS:((SP) + 1 : (SP))
(SP) <--- (SP) + 2
(CS) <--- SS:((SP) + 1 : (SP))
(SP) <--- (SP) + 2
[ (SP) <--- (SP) + N ]
```

Se reface din stivă perechea de registre (CS:IP), cu actualizarea registrului SP și, dacă este prezentă constanta N, se adună N la SP.

Este important de reținut că instrucțiunile CALL și RET sunt instrucțiuni pereche: ele salvează, respectiv refac adresa de revenire. Pentru ca mecanismul de apel/revenire să funcționeze corect, trebuie îndeplinite condițiile:

- tipul instrucțiunii CALL și tipul instrucțiunii RET trebuie să coincidă (FAR sau NEAR);
- registrul SP din momentul execuției instrucțiunii RET să aibe aceeași valoare ca la intrarea în procedură (să indice adresa de revenire);
- adresa de revenire salvată în stivă să nu fi fost alterată de către procedură.

Încălcarea unora din cele trei condiții de mai sus constituie o eroare frecventă de programare. În astfel de cazuri, funcționarea programului

este compromisă, deoarece se plasează în CS și/sau IP o adresă incorectă, unde probabil nici nu există vreun program cu sens. Acest tip de eroare se numește "execuție de date", adică procesorul ajunge să execute nu instrucțiuni cu sens (definite într-un segment de cod) ci o zonă oarecare de memorie (date).

Recunoașterea acestei erori este destul de ușoară: se blochează tastatura, pe ecranul calculatorului apar tot felul de caractere ciudate, difuzorul începe să țiuie etc. Singura soluție este în acest caz un reset general al calculatorului.

Putem deci enunța o regulă de aur a programării în limbaj de asamblare: **Verificați controlul stivei!**

#### **2.4.2 Instrucțiunea de salt (JMP)**

Are forma generală:

```
JMP    țintă
```

În care țintă specifică adresa de salt (punctul în care se va da controlul). Specificarea țintei se poate face printr-o etichetă sau printr-o expresie (vezi 2.4.3). Etichetele au asociat un tip (NEAR sau FAR) și pot fi:

- un nume de procedură;
- o etichetă definită cu : (de tip NEAR);
- o etichetă definită cu directiva LABEL.

Exemple de etichete:

```
et1:  
aici:  
et3    LABEL FAR  
gata    LABEL NEAR
```

Există trei tipuri de instrucțiuni JMP:

- de tip SHORT - adresa țintă este situată la o adresă în domeniul [-128, +127] față de adresa instrucțiunii JMP;
- de tip NEAR - adresa țintă este în același segment de cod cu instrucțiunea JMP;
- de tip FAR - adresa țintă poate fi în alt segment de cod față de instrucțiunea JMP.

Tipurile de salt pot fi explicitate prin operatorul PTR, într-una din formele:

```
JMP    SHORT PTR țintă  
JMP    NEAR PTR țintă  
JMP    FAR PTR țintă
```

sau se deduc din atributele expresiei care precizează ținta. În cazul etichetelor, tipul etichetei (FAR sau NEAR) furnizează tipul saltului.

Semnificația instrucțiunii JMP este:

```
JMP de tip SHORT  
    (IP) <--- (IP) + distanța dintre offset-ul curent și cel țintă  
JMP de tip NEAR  
    (IP) <--- offset-ul adresei țintă  
JMP de tip FAR  
    (IP) <--- offset-ul adresei țintă  
    (CS) <--- segmentul adresei țintă
```

Din punctul de vedere al programatorului, faptul că la saltul de tip SHORT se adună la IP o diferență este similar cu modificarea explicită a lui IP de la JMP de tip NEAR.

Ceea ce diferă este codificarea internă a celor două instrucțiuni:

- la salt de tip SHORT, codul instrucțiunii conține diferența dintre offset-ul curent și cel al țintei. Această diferență se memorează intern pe un octet, de unde restricția de domeniu [- 128, +127];
- la salt de tip NEAR, codul instrucțiunii conține explicit offsetul țintei.

Se observă că salturile de tip NEAR și FAR sunt similare cu apelurile de procedură de tip NEAR sau FAR, fără însă a se salva adresa de revenire și identificând numele procedurii cu o etichetă.

### **2.4.3 Tipuri de salt/apel**

Tipurile de instrucțiuni de salt și de apel specifică modul de determinare al adresei țintă, respectiv al adresei procedurii. Deoarece aceste tipuri sunt identice pentru instrucțiunile JMP și CALL, vor fi prezentate împreună. Prin instrucțiuni de salt înțelegem aici cele de tip NEAR sau FAR (salturile de tip SHORT sunt implicit directe). De asemenea, în cele ce urmează identificăm etichetele cu numele de proceduri.

#### **JMP/CALL de tip direct**

Operandul care apare în formatul instrucțiunii este o etichetă care identifică punctul în care se dă controlul (adresa țintă). Salturile/apelurile directe pot fi:

- salturi/apeluri directe intrasegment (NEAR) - eticheta este în același segment de cod cu instrucțiunea JMP/CALL;
- salturi/apeluri intersegment (FAR) - eticheta poate fi definită și în alt segment de cod decât cel care conține instrucțiunea JMP/CALL.

Exemplu:

```
.code
ALFA:
    .....
BETA LABEL FAR
    .....
GAMMA PROC FAR
    .....
GAMMA ENDP
    .....
JMP/CALL ALFA           ; NEAR
JMP/CALL BETA           ; FAR implicit
JMP/CALL FAR PTR GAMMA ; FAR explicit
```

#### **JMP/CALL de tip indirect**

Operandul care apare în formatul instrucțiunii JMP/CALL reprezintă un cuvânt (sau un dublu-cuvânt) din memorie, care conține adresa NEAR (sau FAR) unde se va da controlul.

Salturile/apelurile indirecte pot fi:

- Salt/apel indirect intrasegment (NEAR)

Forma generală este:

JMP/CALL      expr

În care expr poate fi:

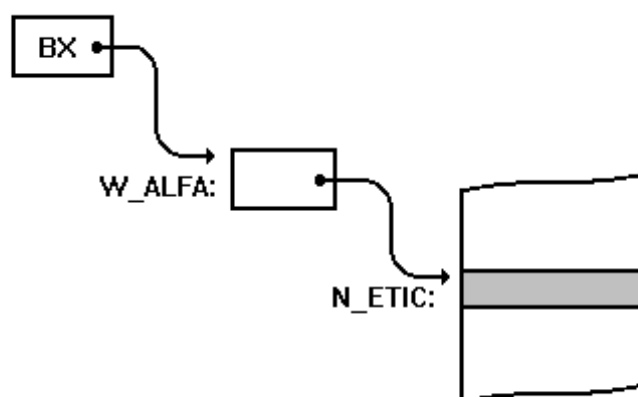
- a) un registru care conține offset-ul țintei;
- b) o variabilă de tip WORD care conține offset-ul țintei;
- c) o expresie cu indici reprezentând un cuvânt din memorie care conține offset-ul țintei;
- d) o referire anonimă la un cuvânt din memorie care conține offset-ul țintei.

Să considerăm câteva exemple:

```
.data
    W_ALFA      dw    N_ETIC
    W_TAB_PROC  dw    N_PROC_0
                    dw    N_PROC_1
                    dw    N_PROC_2

.code
N_ETIC:                                ; Etichetă NEAR
    .....
N_PROCi PROC NEAR                      ; i = 0, 1, 2
    .....
N_PROCi ENDP
    .....
    LEA        BX, N_PROC1
    JMP/CALL   BX                       ; Tipul a)
    JMP/CALL   W_ALFA                   ; Tipul b)
    JMP/CALL   W_TAB_PROC [SI]          ; Tipul c); SI = 0, 2, 4
    LEA        BX, W_TAB_PROC
    JMP/CALL   WORD PTR [BX] [SI]       ; Tipul d); SI = 0, 2, 4
    LEA        BX, W_ALFA
    JMP/CALL   WORD PTR [BX]            ; Tipul d)
```

Formele cu referiri anonime la memorie trebuie însoțite de operatorul WORD PTR. Ultimele două exemple de salt/apel (tipul d) pun în evidență două niveluri de indirectare (vezi Figura 2.4): de la BX la variabila W\_ALFA și de la W\_ALFA la eticheta N\_ETIC.



**Figura 2.4 Cele două niveluri de indirectare la CALL WORD PTR [BX]**

- Salt/apel indirect intersegment (FAR)

Forma generală este:

JMP/CALL      expr

În care expr poate fi:



- a) o variabilă de tip DWORD care conține adresa completă a țintei;
- b) o expresie cu indici reprezentând un dublu cuvânt din memorie care conține adresa completă a țintei;
- c) o referire anonimă la un dublu cuvânt de memorie care conține țintei.

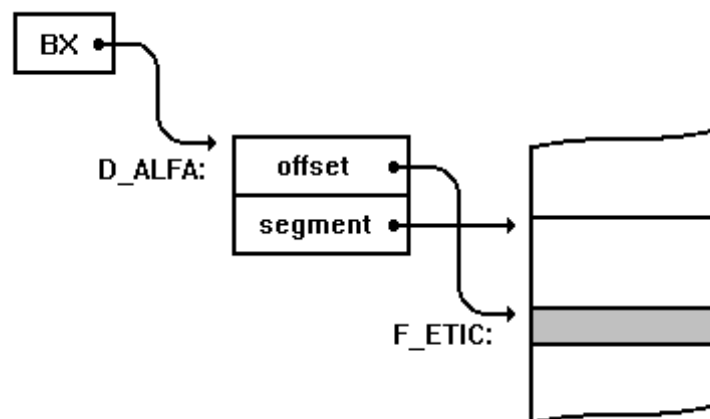
Să considerăm câteva exemple:

```
.data
    D_ALFA      dd F_ETIC
    D_TAB_PROC  dd F_PROC_0
                  dd F_PROC_1
                  dd F_PROC_2

.code
D_ETIC LABEL FAR           ; Eticheta FAR

.....
F_PROCi PROC FAR           ; i = 0, 1, 2
.....
F_PROCi ENDP
.....
JMP/CALL      D_ALFA           ; Tipul a)
JMP/CALL      D_TAB_PROC [SI]   ; Tipul b); SI = 0, 2, 4
LEA           BX, D_TAB_PROC
JMP/CALL      DWORD PTR [BX] [SI] ; Tipul c); SI = 0, 2, 4
LEA           BX, D_ALFA
JMP/CALL      DWORD PTR [BX]    ; Tipul c)
```

Formele cu referiri anonime la memorie trebuie însoțite de operatorul DWORD PTR. Ultimele două exemple de salt/apel (tipul c) pun în evidență două niveluri de indirectare (vezi Figura 2.5): de la BX la variabila DALFA și de la DALFA la eticheta ETIC.



**Figura 2.5 Cele două niveluri de indirectare la CALL DWORD PTR [BX]**

Apelurile indirecte de proceduri trebuie să respecte tipul procedurii. Astfel, o instrucțiune de forma CALL WORD PTR [BX] trebuie folosită numai cu proceduri de tip NEAR, iar una de tip CALL DWORD PTR [BX] numai cu proceduri de tip FAR.

Să considerăm un exemplu de utilizare a tabelor de adrese. Presupunem dezvoltarea unui meniu de comenzi, fiecare comandă fiind reprezentată de un caracter alfabetic (de la 'A' la 'Z') Unele caractere pot să nu fie utilizate în meniu. De asemenea, presupunem caracterul în registrul AL și existența, pentru fiecare comandă, a unei proceduri de tratare.

O soluție evidentă dar neeficientă este comparare lui AL cu toate caracterele meniului și apeluri corespunzătoare de proceduri, deci ceva de genul:

```
CMP    AL, 'A'
JZ     et_A
CMP    AL, 'B'
JZ     et_B ; etc.
```

Evident că, dacă meniul este mare, o asemenea secvență de program este greu de întreținut. Adăugarea sau eliminarea unei comenzi presupune parcurgerea cu atenție a cosului sursă.

Soluția adecvată a problemei constă în definirea unei tabele cu adresele procedurilor de tratare, în ordinea alfabetică a comenzilor. Pentru literele de alfabet care nu sunt alocate nici unei comenzi, se scrie o procedura specială PROC\_NULL. La execuție, se va calcula poziția respectivă din tabelă, chiar pe baza caracterului din AL și se va genera un apel indirect de procedură.

```
.data
    TAB_CMD    DW PROC_A
                DW PROC_B
                DW PROC_NULL    ; Comanda 'C' nu exista
                DW PROC_D
                                ; etc.
                DW PROC_Z

.code
    SUB    AL, 'A'                ; AL = 0...27
    ADD    AL, AL                ; AL <-- 2*AL
    ; ADD    AL, AL                ; Se pune în cazul procedurilor FAR
    MOV    BL, AL
    XOR    BH, BH                ; BX = intrare în tabelă
    CALL   WORD PTR TAB_CMD[BX]
```

În cazul procedurilor far, se definește tabela TAB\_CMD cu directiva Define DoubleWord și se calculează BX <-- 4\*AL, în loc de BX <-- 2\*AL.

Această implementare este foarte ușor de întreținut. Eliminarea unei comenzi se face punând adresa procedurii PROC\_NULL pe poziția corespunzătoare. Adăugarea unei comenzi se face înlocuind procedura PROC\_NULL cu procedura de tratare a noii comenzi.

Nici un tip de instrucțiune de salt sau de apel de procedură nu modifică vreun bistabil de condiție.

#### 2.4.4 Instrucțiuni de salt condiționat

Instrucțiunile din această categorie implementează salturi condiționate de valoarea unor bistabili de condiție. Prin extensie, instrucțiunile de salt obișnuite (JMP) se mai numesc și salturi necondiționate. Instrucțiunile de salt condiționat au următoarele caracteristici:

- toate instrucțiunile de salt condiționat sunt de tip SHORT (deci directe), ceea ce înseamnă că adresa țintă trebuie să fie la o distanță cuprinsă între -127 și +128 de octeți față de instrucțiunea de salt;
- există mai multe mnemonice pentru aceeași instrucțiune;
- dacă condiția nu este îndeplinită, saltul nu are loc, deci execuția continuă cu instrucțiunea următoare;

- există instrucțiuni atât pe condiția directă cât și pe condiția negată;
- bistabilii de condiție nu sunt afectați.

În Tabelul 2.6 se prezintă instrucțiunile de salt condiționat. Prima coloană listează mnemonicele instrucțiunilor, coloana a doua listează condiția de salt iar coloana a treia interpretarea condiției respective. Primele opt linii reprezintă salturile pe condițiile directe iar celelalte opt, salturile pe condițiile corespunzătoare negate.

Denumirile instrucțiunilor se citesc în forma:

### **Jump (Salt) on (pe) <Interpretare>**

unde <Interpretare> este textul din coloana a treia.

Instrucțiunile de salt condiționat se utilizează după o instrucțiune care modifică bistabilii de condiție. Cea mai des întâlnită situație este instrucțiunea de comparație CMP.

Se observă că există două categorii de instrucțiuni pentru noțiunile de "mai mic" și "mai mare": cele care conțin cuvintele "above" sau "bellow" și cele care conțin cuvintele "less" sau "greater". Primele se folosesc în situația comparației a două cantități fără semn, iar ultimele, în situația comparației unor cantități cu semn.

Să considerăm următoarele secvențe de program, în care se compară valorile 0FFH și 1:

MOV	AL, 0FFH	MOV	AL, 0FFH
MOV	BL, 1	MOV	BL, 1
CMP	AL, BL	CMP	AL, BL
JA	ET_1	JG	ET_1

Se observă că  $(AL) > (BL)$  dacă interpretăm cele două valori fără semn ( $255 > 1$ ) și că  $(AL) < (BL)$  dacă interpretăm cele două valori cu semn ( $-1 < 1$ ). Astfel, în primul exemplu, saltul la eticheta ET\_1 are loc, pe când în cel de-al doilea nu.

Condițiile asupra bistabililor în cele două tipuri de instrucțiuni și semnificația bistabililor respectivi (vezi 2.2.1) explică cele două tipuri de condiții.

Reținem deci următoarele reguli:

- La comparații cu semn, folosim "GREATER" și "LESS"
- La comparații fără semn, folosim "ABOVE" și "BELLOW"

Instrucțiune (mnemonică)	Condiție de salt	Interpretare
JE, JZ	ZF = 1	Zero, Equal
JL, JNGE	S F $\neq$ OF	Less, Not Greater or Equal
JLE, JNG	SF $\neq$ OF sau ZF = 1	Less or Equal, Not Greater
JB, JNAE, JC	CF = 1	Bellow, Not Above or Equal, Carry
JBE, JNA	CF = 1 sau ZF = 1	Bellow or Equal, Not Above
JP, JPE	PF = 1	Parity, Parity Even
JO	OF = 1	Overflow
JS	SF = 1	Sign
JNE, JNZ	ZF = 0	Not Zero, Not Equal
JNL, JGE	S F = OF	Not Less, Greater or Equal
JNLE, JG	SF = OF și ZF = 0	Not Less or Equal, Greater
JNB, JAE, JNC	CF = 0	Not Bellow, Above or Equal, Not Carry
JNBE, JA	CF = 0 și ZF = 0	Not Bellow or Equal, Above
JNP, JPO	PF = 0	Not Parity, Parity Odd
JNO	OF = 0	Not Overflow
JNS	SF = 0	Not Sign

**Tabelul 2.6 Instrucțiunile de salt condiționat**

Uneori este necesar să scriem instrucțiuni de salt condiționat la etichete care ies în afara domeniului [-128, +127] față de instrucțiunea curentă. În această situație, folosim următoarea schemă, prin care înlocuim saltul pe o condiție directă "departe" cu un salt pe condiția negată "aproape" și cu un salt necondiționat "departe".

```

Salt condiționat pe condiție directă la ET_1
;      Eticheta ET_1 e prea departe
ET_1:
```

Forma echivalentă este:

```

Salt condiționat pe condiție negată la ET_2
Salt necondiționat la ET_1
ET_2:
```

De exemplu instrucțiunea:

```
JE    ET_1
```

se substituie cu:

```

JNE   ET_2
JMP   ET_2
ET_2:
```

În schema echivalentă, eticheta ET\_1 poate fi la orice distanță față de punctul de salt.

#### **2.4.5 Instrucțiuni pentru controlul buclilor de program (JCXZ, LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE)**

##### **Instrucțiunea JCXZ (Jump if CX is Zero - Salt dacă CX este zero)**

Are forma generală:

JCXZ etic

unde etic este o etichetă aflată în domeniul [-128, +127] față de instrucțiunea curentă (la fel ca la salturile de tip SHORT).

Semnificația este evidentă:

dacă (CX) = 0  
(IP) <--- (IP) + distanța dintre offset-ul curent și cel țintă

adică se sare la eticheta specificată dacă CX conține valoarea 0.

##### **Instrucțiuni de ciclare (LOOPxx - Ciclează)**

Aceste instrucțiuni sunt de fapt salturi condiționate de valoarea registrului CX și de bistabilul ZF. La fel ca prefixele de repetare, cu care, de altfel, se aseamănă oarecum, există câte două mnemonice pentru aceeași instrucțiune. Forma lor generală este:

LOOPxx etic

unde etic este o etichetă aflată în domeniul [-128, +127] față de instrucțiunea curentă.

Toate aceste instrucțiuni utilizează registrul CX ca număr de iterații.

##### **Instrucțiunea LOOP**

Semnificația este următoarea:

CX <--- CX - 1  
dacă CX != 0, atunci  
(IP) <--- (IP) + distanța dintre offset-ul curent și cel țintă

Cu alte cuvinte, se decrementează CX și, dacă acesta este diferit de zero se sare la eticheta specificată.

Următoarea secvență calculează suma (pe 16 biți) a elementelor unui tablou TAB de 100 de întregi și o depune în variabila SUMA.

```
.data
    TAB    DW 100 DUP (0)
    SUMA   DW ?

.code
    XOR     AX, AX                ; Inițializează sumă
    MOV     CX, 100              ; Număr de iterații
    MOV     SI, AX               ; Inițializează indice
NEXT:
    ADD     AX, TAB [SI]         ; Calcul sumă
    ADD     SI, 2                ; Actualizare indice
    LOOP    NEXT                ; Ciclează
    MOV     SUMA, AX             ; Depune rezultat
```

Forma generală a unei bucle de program realizată cu instrucțiunea LOOP este:

```
start_bucla:
    ; ;
    ; Corp buclă ;
    ; ;
LOOP start_bucla
```

Corpul buclei se va executa de atâtea ori cât este conținutul inițial al registrului CX (presupunând că acesta nu este modificat explicit în interiorul buclei).

Trebuie observat că registrul CX este mai întâi decrementat și apoi testat. Ca atare, dacă (CX) este inițial 0, bucla de program se va executa de 65535 de ori. Protecția față de o asemenea situație se realizează prin instrucțiunea JCXZ. Forma corectă a buclei de program va fi:

```
JCXZ end_bucla
start_bucla:
    ; ;
    ; Corp buclă ;
    ; ;
LOOP start_bucla
end_bucla:
```

În forma de mai sus, dacă (CX) este inițial 0, corpul buclei nu se execută nici o dată.

În situația în care corpul buclei ocupă mai mult de 128 de octeți, trebuie să renunțăm la JCXZ și LOOP și să le înlocuim cu comparații și salturi explicite. Bucla de program va avea forma generală:

```
TEST CX, CX
JNZ start_bucla
JMP end_bucla
start_bucla:
    ; ;
    ; Corp buclă ;
    ; ;
    DEC CX
    TEST CX, CX
    JZ end_bucla
    JMP start_bucla
end_bucla:
```

Următorul exemplu calculează primele 20 de numere Fibonacci (fib(n), n = 1,...,20), definite recursiv prin:

```
f(0) = 0, f(1) = 1
f(k) = f(k-1) + f(k-2), dacă n > 1
```

și le depune în tabloul FIBO. Cele trei valori (f(k), f(k-1) și f(k-2)) care intervin în calcul sunt ținute în registrele AX, BX și DX.

```
.data
    FIBO DW 20 DUP (?)
.code
    MOV BX, 0          ; f(0)
    MOV DX, 1          ; f(1)
    MOV CX, 20         ; Contor
    MOV DI, 0          ; Indice în tabloul
bucla:
    MOV AX, BX
```

```

ADD AX, DX      ; f(k) <-- f(k-1) + f(k-2)
MOV FIBO [DI], AX ; Depune rezultat
ADD DI, 2       ; Actualizează indice
MOV DX, BX      ; f(k-1) devine f(k-2)
MOV BX, AX      ; f(k) devine f(k-1)
LOOP bucla      ; Ciclează

```

### Instrucțiunea LOOPZ/LOOPE (Loop While Zero/Equal - Ciclează cât timp este zero/egal)

Semnificația este următoarea:

```

CX <--- CX - 1
dacă CX != 0 și ZF = 1, atunci
    (IP) <--- (IP) + distanța dintre offset-ul curent și cel țintă

```

Se decrementează deci CX și, dacă acesta este diferit de zero și bistabilul ZF este 1 (adică rezultatul ultimei operații aritmetice a fost zero), se sare la eticheta specificată. De obicei, pentru poziționarea bistabilului ZF se utilizează o instrucțiune de comparație.

Exemplul următor determină primul întreg diferit de zero dintr-un tablou TABL de 100 de întregi.

```

.code
    MOV     CX, 100
    LEA     BX, TABL
    MOV     SI, -2
next:
    ADD     SI, 2
    CMP     WORD PTR [BX][SI], 0
    LOOPZ   next
    JNZ     gasit

```

Dacă bistabilul ZF este 0 la ieșirea din buclă, înseamnă ca s-a identificat primul întreg diferit de 0, iar SI conține adresa acestui întreg. În caz contrar, toate cele 100 de elemente ale tabloului sunt nule.

### Instrucțiunea LOOPNZ/LOOPNE (Loop While Not Zero/Not Equal - Ciclează cât timp este diferit de zero/diferit)

Semnificația este următoarea:

```

CX <--- CX - 1
dacă CX != 0 și ZF = 0, atunci
    (IP) <--- (IP) + distanța dintre offset-ul curent și cel țintă

```

Condiția asupra bistabilului ZF este negata condiției de la LOOPZ/LOOPE. Efectul este că se ciclează cât timp rezultatul ultimei operații aritmetice este diferit de zero, dar nu de mai multe ori cât este conținutul inițial al lui CX.

De remarcat asemănările și deosebirile care există între instrucțiunile de ciclare și prefixele de repetare de la operații cu șiruri (vezi 2.3.2).

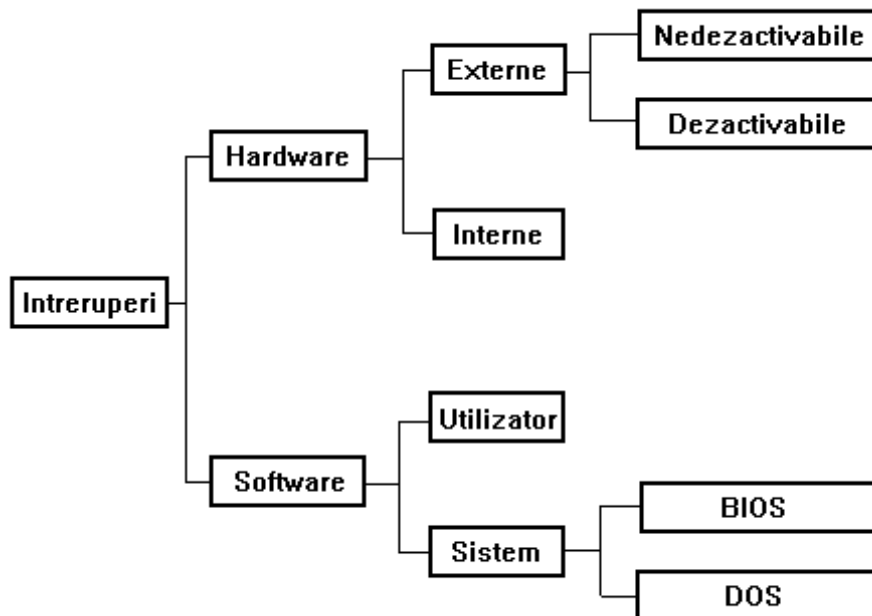
## 2.5. Întreruperi

Noțiunea de întrerupere presupune (așa cum îi arată și numele) întreruperea programului în curs de execuție și transferul controlului la o anumită rutină specifică, numită rutină de tratare, dictată de cauza care a generat întreruperea. Mecanismul prin care se face acest transfer este, în

esență, de tip apel de procedură, ceea ce înseamnă revenirea în programul întrerupt, după terminarea rutinei de tratare.

### 2.5.1 Întreruperi hardware și software. Tabela de întreruperi

Întreruperile se pot clasifica după mai multe criterii, rezultând tipurile ilustrate în Figura 2.7.



**Figura 2.7 Clasificarea întreruperilor**

Întreruperi software apar ca urmare a execuției unor instrucțiuni, cum ar fi INT, INTO, DIV, IDIV;

Întreruperi hardware externe sunt provocate de semnale electrice care se aplică pe intrările de întreruperi INT și NMI ale procesorului;

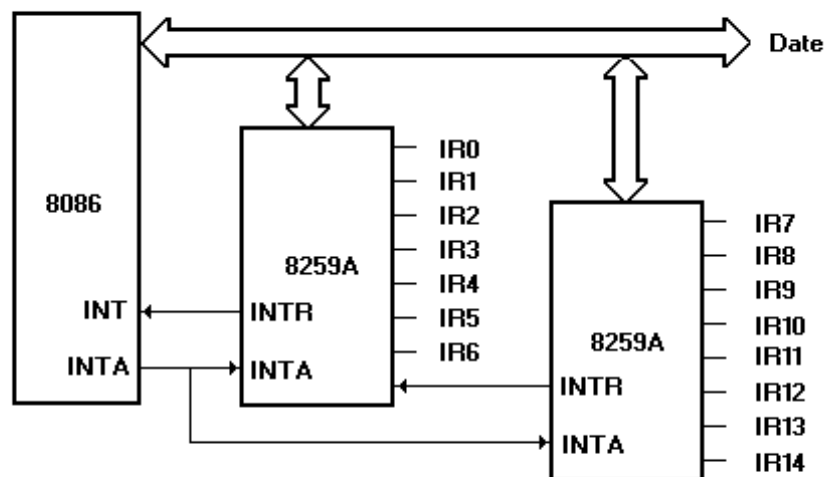
Întreruperile hardware interne apar ca urmare a unor condiții speciale de funcționare a procesorului (cum ar fi execuția pas cu pas a programelor);

Întreruperile dezactivabile sunt provocate de semnalul electric aplicat pe linia (intrarea) INT și sunt luate în considerare numai dacă bistabilul IF este 1.

Întreruperile nedezactivabile sunt provocate de semnalul electric aplicat pe linia NMI și sunt totdeauna luate în considerare.

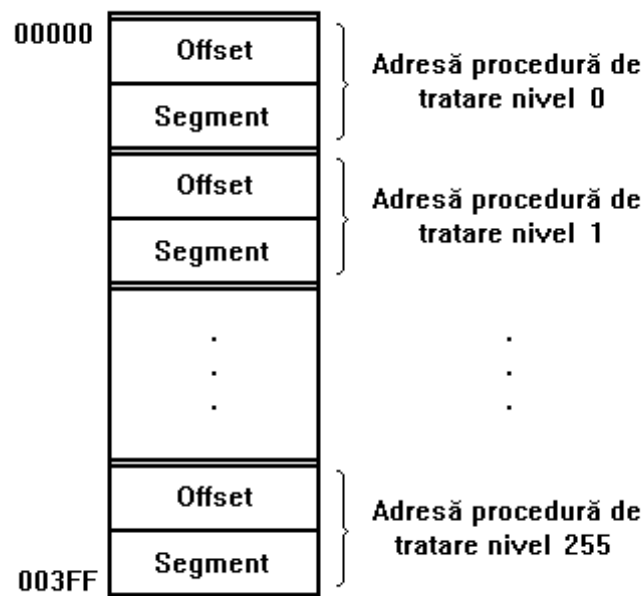
Întreruperile hardware dezactivabile sunt controlate de unul sau mai multe circuite specializate (controlere de întreruperi 8259A), care acceptă fiecare cel mult opt cereri de întreruperi, pe care le transmit către linia INT a procesorului. Dacă bistabilul IF este 1, procesorul răspunde printr-o secvență de semnale electrice INTA (Interrupt Acknowledge), asemănătoare unei secvențe de citire a codului unei instrucțiuni. Pe durata unuia din aceste semnale, controlerul care a inițiat cererea de întrerupere plasează pe magistrala de date octetul care identifică nivelul întreruperii. Un asemenea sistem se numește sistem vectorizat de întreruperi (vezi Figura 2.8).





**Figura 2.8 Întreruperi hardware vectorizate**

Într-un sistem cu procesor 8086 pot exista maxim 256 de întreruperi (niveluri) distincte. Fiecare din aceste niveluri poate avea asociată o procedură de tip far, numită rutină de tratare. Adresele acestor rutine sunt trecute într-o așa numită tabelă de întreruperi, aflată la adresele fizice 00000 - 003FFH, ocupând deci 1024 de octeți. Fiecare nivel ocupă 4 octeți, primii reprezentând offset-ul iar următorii adresa de segment a procedurii (vezi Figura 2.9).



**Figura 2.9 Tabela de întreruperi**

La apariția unei întreruperi, au loc următoarele acțiuni:

- se salvează în stivă registrele FLAGS, CS și IP (în această ordine);
- se șterg bistabilii IF și TF;
- se furnizează procesorului un întreg pe 8 biți (deci în gama 0- 255), numit și vector de întrerupere, care identifică nivelul asociat întreruperii;

- se execută un salt indirect intersegment la adresa de început a rutinei de tratare, prin intermediul tabelului de întreruperi.

Vectorul de întrerupere poate fi furnizat procesorului în următoarele moduri:

- în cazul întreruperilor hardware interne, nivelul este implicit; de exemplu, pentru întreruperea de execuție pas cu pas (generată dacă bistabilul TF = 1), se utilizează totdeauna nivelul 1;
- în cazul întreruperilor hardware externe, nivelul este plasat pe magistrala de date, în cadrul ciclului mașină (Interrupt Acknowledge), care urmează după luarea în considerare a întreruperii, de către dispozitivul care a generat întreruperea;
- în cazul întreruperilor software, nivelul este conținut în instrucțiune.

Se observă că, exceptând salvarea flagurilor și ștergerea bistabililor IF și TF, secvența de tratare a unei întreruperi se reduce la un apel indirect de procedură far, prin intermediul tabelului de întreruperi.

### **2.5.2 Instrucțiuni specifice întreruperilor (INT, IRET, INTO)**

#### **Instrucțiunea INT (Interrupt - Întrerupere software)**

Are forma generală:

INT n

unde n este o constantă întreagă în gama 0-255.

Semnificația este următoarea:

```
(SP) <-- (SP) - 2
SS:((SP) + 1 : (SP)) <-- (FLAGS)
IF <-- 0, TF <-- 0
(SP) <-- (SP) - 2
SS:((SP) + 1 : (SP)) <-- (CS)
(CS) <-- (4*n + 2)
(SP) <-- (SP) - 2
SS:((SP) + 1 : (SP)) <-- (IP)
(IP) <-- (4*n)
```

Se încarcă deci în CS și IP, conținutul de la adresele fizice  $4*n+2$  și  $4*n$ , adică cei patru octeți corespunzători nivelului n din tabela de întreruperi.

#### **Instrucțiunea IRET (Interrupt Return - Revenire din întrerupere)**

Datorită acțiunilor care au loc la apariția unei întreruperi, o procedură de tratare nu se poate încheia cu o instrucțiune RETF, deoarece trebuie refăcut și registrul de flaguri. Ca atare, procedurile de tratare a întreruperilor se încheietotdeauna cu instrucțiunea IRET.

Instrucțiunea, care este fără operanzi, are semnificația următoare:

```
(IP) <-- SS:((SP) + 1 : (SP))
(SP) <-- (SP) + 2
(CS) <-- SS:((SP) + 1 : (SP))
(SP) <-- (SP) + 2
(FLAGS) <-- SS:((SP) + 1 : (SP))
(SP) <-- (SP) + 2
```

Se observă că bistabilii IF și TF, care au fost șterși la apariția întreruperii, sunt refăcute așa cum erau înainte de întrerupere, odată cu ceilalți bistabili din registrul FLAGS.

## Instrucțiunea INTO (Interrupt if Overflow - Întrerupere în caz de depășire)

Instrucțiunea, care este fără parametri, are semnificația următoare:

dacă  $OF = 1$ , atunci se execută secvența corespunzătoare unei instrucțiuni INT 4

Codificarea instrucțiunilor INT este pe doi octeți, cu excepția instrucțiunii INT 3, la care codificarea este pe un singur octet. Acest fapt nu este întâmplător, deoarece nivelurile 2 și 3 sunt în general utilizate de programele de depanare (debuggere).

Astfel, rularea pas cu pas a unui program se face setând bistabilul TF și scriind pe nivelul 1 al tabelii de întreruperi adresa unei rutine proprii de tratare. La execuția fiecărei instrucțiuni, se va da controlul rutinei proprii, care va afișa corespunzător starea programului. Rularea până o adresă dată (breakpoint) se realizează prin înlocuirea octetului de la adresa respectivă cu codul unei instrucțiuni INT 3. Astfel, când programul care se execută ajunge la acea adresă, se dă controlul rutinei de tratare pe nivelul 3.

Nivelurile predefinite de întrerupere sunt deci:

- 0 - depășire la împărțire (cauze posibile: instrucțiunile DIV sau IDIV);
- 1 - execuție pas cu pas (cauză posibilă: bistabilul TF = 1);
- 2 - întrerupere externă nedeactivabilă (cauză posibilă: senzal electric pe linia de întrerupere nedeactivabilă NMI);
- 3 - execuție pas cu pas (cauză posibilă: instrucțiunea INT 3);
- 4 - depășire (cauză posibilă: instrucțiunea INTO).

La calculatoarele de tip IBM-PC, se mai pot cita întreruperile hardware de la ceasul de timp real (nivelu 8) și de la tastatură (nivelul 9).

Întreruperile software în gama 20H-2FH sunt folosite de sistemul de operare DOS, iar cele în gama 10H-1AH de către subsistemul de intrări-ieșiri BIOS.

### 2.6 Instrucțiuni pentru controlul procesorului

Toate instrucțiunile de acest gen sunt fără operanzi.

O primă categorie de instrucțiuni se referă la controlul explicit al unor bistabili de condiție (CLC, STC, CMC, CLD, STD, CLI și STI).

Instrucțiunile **CLC (Clear Carry Flag)**, **STC (Set Carry Flag)** și **CMC (Complement Carry Flag)** efectuează operațiile de ștergere, setare și respectiv, complementare a bistabilului CF.

Instrucțiunile **CLD (Clear Direction Flag)** și **STD (Set Direction Flag)** șterg, respectiv setează bistabilul DF.

Instrucțiunile **CLI (Clear Interrupt Flag)** și **STI (Set Interrupt Flag)** șterg, respectiv setează bistabilul IF. Când  $IF = 0$ , întreruperile externe nedeactivabile nu sunt luate în considerare. Astfel, o așa numită secvență critică de program (care nu dorim să fie întreruptă) se protejează printr-o instrucțiune CLI înainte și o instrucțiune STI după. O secvență critică tipică este modificarea explicită a tabelii de întreruperi.

O altă serie de instrucțiuni (HALT, LOCK, WAIT) realizează unele operații speciale asupra procesorului.

Astfel, instrucțiunea **HALT (Oprire procesor)**, forțează procesorul într-o stare specială de inactivitate, din care se poate ieși doar prin întreruperi externe sau prin reset general.

Prefixul **LOCK (Blocare magistrală)** se poate folosi înaintea oricărei instrucțiuni, efectul fiind că pe durata instrucțiunii, accesul la magistrala sistemului hardware al unui alt dispozitiv este blocat. Într-un sistem multiprocesor, nu este suficient să protejăm secvențele critice prin instrucțiuni CLI/STI, deoarece controlul magistralei ar putea fi preluat de un alt procesor. De aici, utilitatea prefixului LOCK.

Instrucțiunea **WAIT (Așteaptă)** este folosită pentru a sincroniza activitatea procesorului cu cea a unui alt dispozitiv, de obicei un coprocesor matematic. Execuția unei instrucțiuni matematice este preluată de coprocesor; totuși, procesorul de bază nu poate continua programul până nu primește un semnal de la coprocesor, prin care se anunță sfârșitul operației executate. Această sincronizare se realizează printr-un semnal electric aplicat pe linia TEST al procesorului de bază. Ca atare, instrucțiunea WAIT va forța procesorul de bază într-o stare de inactivitate, până la apariția unui semnal electric pe linia TEST.

În fine, există și o instrucțiune care nu face nimic: instrucțiunea NOP (No Operation). Scopul unei asemenea instrucțiuni este de a introduce o întârziere în program. Asamblorul recunoaște mnemonica NOP, dar codificarea internă este aceeași cu a instrucțiunii XCHG AX, AX, care, evident, nu face nimic.

## **2.7 Dezvoltarea programelor în limbaj de asamblare**

Acum, după ce am parcurs setul de instrucțiuni al procesorului, putem trece la prezentarea unui cadru de dezvoltare al programelor în limbaj de asamblare. Contextul ales este cel al calculatorului de tip IBM-PC sub sistemul de operare DOS și al produselor software Borland: asamblorul **TASM** (Turbo Assembler), link-editorul **TLINK** (Turbo Linker), bibliotecarul **TLIB** (Turbo Librarian) și depanatorul interactiv **TD** (Turbo Debugger).

Vom utiliza extensiile implicite ale fișierelor: **.ASM** pentru fișiere sursă, **.OBJ** pentru fișiere obiect, **.EXE** sau **.COM** pentru fișiere executabile.

O problemă specifică limbajului de asamblare este absența unor instrucțiuni de intrare-ieșire de nivel înalt. Pentru asemenea operații, va trebui să ne scriem propriile rutine. În această lucrare, vom utiliza o serie de proceduri și macroinstrucțiuni, implementate în fișierul **IO.H** (fișier header, listat în Anexa A) și în fișierul **IO.ASM** (fișier sursă, listat în Anexa B). Aceste fișiere asigură (între altele) suport pentru următoarele operații de bază:

- introducerea de caractere de la consolă;
- afișarea de caractere la consolă;
- introducerea de șiruri de caractere de la consolă;
- afișarea de șiruri de caractere la consolă;

- afișarea unor mesaje imediate la consolă;
- introducerea de întregi pe 16 biți cu sau fără semn de la consolă;
- afișarea de întregi pe 16 biți cu sau fără semn la consolă;
- inițializarea registrelor DS și ES la intrarea în program;
- terminarea programului cu ieșire în sistemul DOS.

Se vor utiliza directivele de definire simplificată a segmentelor. Țăblonul de dezvoltare al unui program ASM care conține un modul executabil va fi următorul:

```
.model MMMMM
includeio.h
.stack NNNN
.data
    ; Definiții de date
.code
    ; Definiții de proceduri
start:
    init_ds_es
        ; Program principal
    exit_dos
end    start
```

unde:

- **MMMMM** este modelul de memorie, care poate fi **tiny**, **small**, **medium**, **compact**, **large** sau **huge**; modelele uzuale sunt small și large, în care toate adresele, procedurile, apelurile, salturile și revenirile din procedură sunt implicit de tip NEAR, respectiv de tip FAR;
- **NNNN** este dimensiunea rezervată segmentului de stivă; O valoare uzuală este 1024;
- **include io.h** este o directivă care include în textul sursă fișierul io.h, care trebuie să se găsească în același catalog (director) cu fișierul sursă;
- **init\_ds\_es** este o macroinstrucțiune (definită în io.h) care inițializează registrele DS și ES cu adresa segmentului de date; registrele SS și CS sunt inițializate automat la încărcarea programului executabil de pe disc.
- **exit\_dos** este o macroinstrucțiune (definită în io.h) care provoacă terminarea programului și revenirea în sistemul DOS;
- **start** este o etichetă care marchează punctul de început al programului principal;
- **end start** este o directivă care marchează sfârșitul logic al modului și precizează totodată că modulul curent este modul de program principal, având punctul de intrare la eticheta start.

O altă variantă este scrierea programului principal sub forma unei proceduri (având numele, de exemplu, `_main`) și precizarea punctului de start prin numele procedurii:

```
.model MMMMM
include    io.h
.stack NNNN
.data
    ; Definiții de date
.code
    ; Definiții de proceduri
_main proc
    init_ds_es
        ; Program principal
    exit_dos
```

```
_main endp  
end _main
```

Un modul de program care nu este modul de program principal (deci conține definiții de date și/sau proceduri) nu are etichetă în directiva end. Într-o aplicație dezvoltată modular (în mai multe fișiere sursă), un singur modul poate fi modul de program principal.

Asamblorul nu face distincție între simbolii scriși cu litere mici sau mari. De acum încolo, vom scrie de regulă programele cu litere mici, iar cu litere mari unele directive și tipuri de date definite de utilizator, pentru a le scoate în evidență.

Asamblarea unui fișier sursă NUME.ASM se face cu una din comenzile:

```
C:\> tasm nume.asm  
C:\> tasm nume
```

În urma căreia rezultă un fișier obiect NUME.OBJ.

Dacă se dorește și fișier listing, se dă comanda:

```
C:\> tasm nume „nume
```

În urmă căreia rezultă și un fișier NUME.LST.

Dacă se dorește o depanare simbolică ulterioară, trebuie introdusă opțiunea /zi:

```
C:\> tasm nume /zi
```

Operația de asamblare se face pentru fiecare fișier sursă în parte.

Legarea modulelor se face cu comanda:

```
C:\> tlink nume_1 nume_2 ... [, nume_bin ] [/v]
```

În care parantezele drepte indică parametri opționali; nume\_1, nume\_2 sunt numele modulelor obiect, nume\_bin este numele fișierului executabil rezultat (dacă lipsește, se consideră numele primului modul obiect) iar /v este o opțiune de depanare simbolică (se introduce dacă vrem să executăm programul sub controlul depanatorului TD). În urma comenzii rezultă un fișier executabil.

O situație tipică este cea a unui singur modul sursă, caz în care comanda de legare va fi:

```
C:\> tlink nume io
```

prin care se leagă și modulul io.obj (care conține procedurile de intrare-ieșire descrise mai sus).

Peste tot în cuprinsul cărții vom considera că șirurile de caractere sunt terminate cu octetul 0 binar. Definiția unui șir constant sau rezervarea de spațiu pentru un șir variabil se pot face prin directiva Define Byte (constantele simbolice cr și lf sunt definite în fișierul io.h). Definiția unui întreg sau rezervarea de spațiu pentru un întreg se poate face cu directiva Define Word. Definirea de spațiu la nivel de caracter se face cu directiva Define Byte.

```
.data  
sir_1 db 'Un sir de caractere', cr, lf, 0
```

```

sir_2  db      80 dup (0)
i_1    dw     -200
i_2    dw      ?
u_1    dw     0FFFFH
u_2    dw     -1
c_1    db     'A'
c_2    db      ?

```

Afișarea unui șir de caractere la consolă se poate face prin macroinstructiunea **puts (Put String)**, în una din formele:

```

puts  sir_1      ; O primă formă de invocare
lea   si, sir_1  ; O a doua
puts  [si]       ; formă de invocare

```

Citirea unui șir de caractere de la consolă se poate face cu macroinstructiunea **gets (Get String)**, în una din formele:

```

gets  sir_2      ; O primă formă de invocare
lea   bx, sir_2  ; O a doua
gets  [bx]       ; formă de invocare

```

Afișarea unui șir constant de caractere (mesaj la consolă) se poate face cu macroinstructiunea **putsi (Put String Immediate)**, care nu necesită definirea șirului și nici prezența explicită a terminatorului 0:

```

putsi  <'Introduceți un șir de caractere', cr, lf>
gets   sir_2
putsi  <'Șirul introdus este:', cr, lf>
puts   sir_2
putsi  <cr,lf>

```

Introducerea unui întreg cu sau fără semn se poate face cu macroinstructiunile **geti (Get Integer)**, fără parametri, care întoarce întregul citit în registrul AX.

Afișarea unui întreg cu sau fără semn se poate face cu macroinstructiunile **puti (Put Integer)** sau **putu (Put Unsigned)**, în una din formele:

```

geti                                ; Citește întreg în AX
puti  ax                            ; Afișează întregul din AX
mov   i_1, ax                       ; Depune
puti  i_1                           ; Afișează întreg cu semn din memorie
putu  u_1                           ; Afișează ca număr fără semn
lea   di, i_2
puti  [di]                          ; Afișează ca număr cu semn
putu  [di]                          ; Afișează ca număr fără semn

```

Introducerea unui singur caracter de la consolă se poate face cu macroinstructiunea **getc (Get Character)**, care întoarce caracterul în registrul al, iar afișarea unui caracter se poate face cu macroinstructiunea **putc (Put Character)**, în una din formele:

```

putc  'A'
putc  car_1
lea   bx, car_2
putc  [bx]

```

Toate macroinstructiunile de mai sus conservă registrele procesorului, deci nu e nevoie de salvări și restaurări explicite.

Să considerăm un exemplu de program, în care se fac următoarele operații:

- se citesc de la consolă cel mult 20 de întregi cu semn;
- se afișează valorile introduse;
- se sortează crescător aceste valori;
- se afișează valorile sortate.

Considerăm modelul de memorie **large**, adică toate adresele sunt implicit de 32 biți. Pentru sortare vom folosi un algoritm elementar (metoda bulelor), descris mai jos (se consideră că indicii din tabloul a variază de la 0 la n-1):

```
for i = 1 to n-1
  for j = n - 1 downto i
    if ( a[j-1] > a[j] )
      schimba a[j] cu a[j-1]
```

Implementarea întregului program este următoarea:

```
.model large
includeio.h
.stack 1024
.data
    vec    dw    20 dup (?)
    n      dw    ?
.code
tipvec proc far
;   Procedură de afișare vector.
;   Date de intrare:
;   ds:si = adresă vector de intregi
;   cx = număr de elemente

    jcxz   tipend                ; Nu avem ce afișa
tip:
    puti   [si]                  ; Afișare întreg cu semn
    putsi  <' '>                  ; Spațiu
    add    si, 2                  ; Actualizare adresă
    loop   tip                    ; Buclă după CX
tipend:
    ret
tipvec endp

bubble proc far
;   Procedură de sortare
;   Date de intrare
;   ds:bx = adresă tablou de intregi
;   cx = dimensiune tablou
;   Variabila i : asignată la si
;   Variabila j : asignată la di
;   Adresa de început a tabloului: asignată la bx

    cmp    cx, 1
    jbe    algend                 ; Nu avem ce sorta
    mov    si, 1                  ; i = 1
fori:
    mov    di, cx
    dec    di                     ; j = n-1
forj:
    shl    di, 1                  ; Întregii pe 2 octeți
```



```

        mov     ax, [bx][di-2]           ; a[j-1]
        cmp     ax, [bx][di]            ; Compara cu a[j]
        jle     nextj                   ; Mai mic sau egal
        xchg    ax, [bx][di]            ; Schimba a[j]
        mov     [bx][di-2], ax          ; cu a[j-1]
nextj:
        shr     di, 1                   ; Refacere indice
        dec     di                       ; Ciclu downto
        cmp     di, si
        jae     forj                     ; Cât timp j >= i
nexti:
        inc     si                       ; Ciclu to
        cmp     si, cx
        jb      fori                     ; Cât timp i < n
algend:
        ret
bubble endp

; Programul principal

start:
        init_ds_es
        putsi   <'Introduceti datele',cr,lf>
        mov     cx, 20                   ; Număr maxim de elemente
        lea     bx, vec                  ; Adresă tablou
iar:
        geti
        test    ax, ax                   ; Este 0 ?
        jz      gata                     ; Dacă da, gata
        mov     [bx], ax                 ; Depunere în tablou
        add     bx, 2                     ; Actualizare adresă
        loop    iar                      ; Buclă după CX
gata:
        mov     ax, 20                   ; Calcul număr de
        sub     ax, cx                     ; elemente introduse
        mov     n, ax

        putsi   <'Vector nesortat', cr, lf>
        lea     si, vec                  ; Adresă tablou
        mov     cx, n                     ; Număr de elemente
        call    tipvec                   ; Afișare tablou nesortat

        lea     bx, vec                  ; Adresă tablou
        mov     cx, n                     ; Număr de elemente
        call    bubble                   ; Sortare tablou

        putsi   <cr,lf,'Vector sortat', cr, lf>
        lea     si, vec
        mov     cx, n
        call    tipvec                   ; Afișare tablou sortat
        exit_dos                          ; ieșire în DOS
end start

```

În segmentul de date se rezerva spațiu pentru tabloul vec de cel mult 20 de întregi și pentru dimensiunea n a acestuia.

Procedura TIPVEC primește în SI adresa unui tablou și în CX numărul de elemente, realizând afișarea la consolă a elementelor, considerate întregi

cu semn. Se observă forma standard a unei bucle de program implementată cu instrucțiunea LOOOP.

Procedura BUBBLE implementează algoritmul de sortare. Indicii *i* și *j* din descrierea algoritmului sunt asigurați la regiștrii SI și SI. Indicii sunt incrementați sau decrementați cu 1, dar întregii sunt pe doi octeți. De aceea, în secvența de adresarea a memoriei, se înmulțește temporar DI cu 2, pentru a accesa corect elementele tabloului. Astfel, elementul de indice 0 se va afla la deplasament 0, elementul de indice 1 la deplasament 2 etc. Înmulțirea și refacerea se realizează prin deplasări logice. Se observă secvența standard de interschimbare a două elemente din memorie (două MOV-uri și un XCHG). Trebuie remarcat modul în care se fac comparațiile diverselor elemente: indicii sunt considerați fără semn, deci testele se fac cu instrucțiuni de salt condiționat de tip "Above" sau "Below". În schimb, tabloul este cu semn, deci comparațiile între elemente se fac cu instrucțiuni de salt condiționat de tip "Greater" sau "Less".

Programul principal începe printr-o buclă de citire a datelor. Introducerea se poate opri mai devreme de 20 de elemente, dacă se introduce valoarea 0. La ieșirea din buclă, se calculează umărul de elemnte efectiv introduse, ca diferență dintre numărul maxim admis (20) și valoarea curentă a lui CX. Acest număr se depune în variabila *n*, pentru utilizări viitoare.

În continuare, vom avea secvențe de apel ale procedurilor TIPVEC și BUBBLE. Se observă încărcarea corespunzătoare a parametrilor (adresa tabloului și numărul de elemente) în registrele desemnate în acest scop pentru fiecare procedură.

Exemplul prezentat ilustrează modul de utilizare a macroinstrucțiunilor prezentate, ca și schema generală de dezvoltare a unui program.