Introducere în limbajul de asamblare

Prezentăm în cele ce urmează familia de microprocesoare intitulată iAPx86 ce stau la baza calculatoarelor IBM PC, începând cu procesoarele 8088 şi 8086, continuând cu 80286, 80386, 80486, Pentium, ş.a.m.d. Procesorul 8086 reprezintă, de fapt, baza familiei ce este cunoscută pe scurt sub denumirea de familia microprocesoarelor x86. De aceea se vor face referiri în continuare la această arhitectură (8086).

Elementele arhitecturale de bază ale microprocesorului

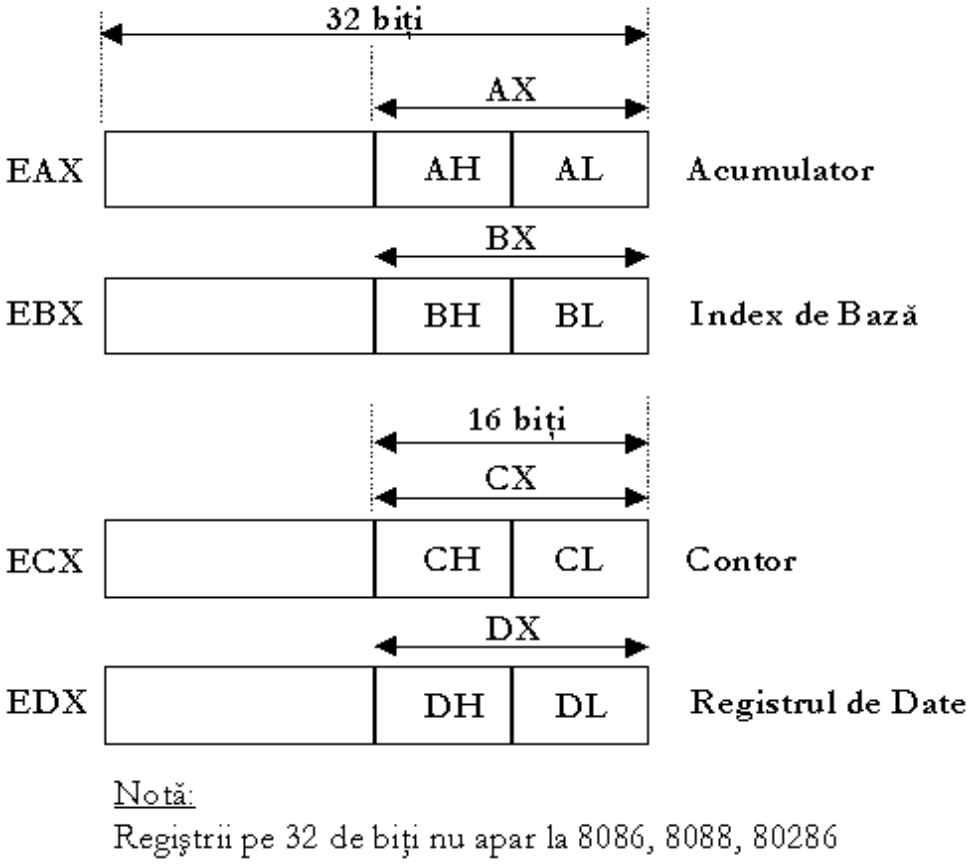

Figura 1. Regiştrii de uz general – acumulator, index de bază, contor şi de date

Regiştrii microprocesorului

Regiştrii (sau registrele) microprocesorului reprezintă locaţii de memorie speciale aflate direct pe cip; din această cauză reprezintă cel mai rapid tip de memorie. Alt lucru deosebit legat de regiştri este faptul că fiecare dintre aceştia au un scop bine precizat, oferind anumite funcţionalităţi speciale, unice. Există patru mari

categorii de regiştri: regiştrii de uz general, registrul indicatorilor de stare (*flags*), regiştrii de segment şi registrul pointer de instrucţiune.

Regiştrii de uz general

Regiştrii de uz general (vezi figura 1 şi figura 2) sunt implicaţi în operarea majorităţii instrucţiunilor, drept operanzi sursă sau destinaţie pentru calcule, copieri de date, pointeri la locaţii de memorie sau cu rol de contorizare. Fiecare dintre cei 8 regiştri de uz general AX, BX, CX, DX, SP, BP, DI,SI sunt regiştri pe 16 biţi pentru microprocesorul 8086, iar de la procesorul 80386 încoace au devenit regiştri pe 32 de biţi, denumiţi, respectiv: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI (litera E provine de la *Extended – extins* în engleză). Mai mult, cei mai puţin semnificativi 8 biţi ai regiştrilor AX, BX, CX, DX formează respectiv regiştrii AL, BL, CL, DL (litera L provine de la *Low – jos* în engleză), iar cei mai semnificativi 8 biţi ai aceloraşi regiştri formează regiştrii AH, BH, CH, DH (litera H provine de la *High – înalt* în engleză) (figura 1).

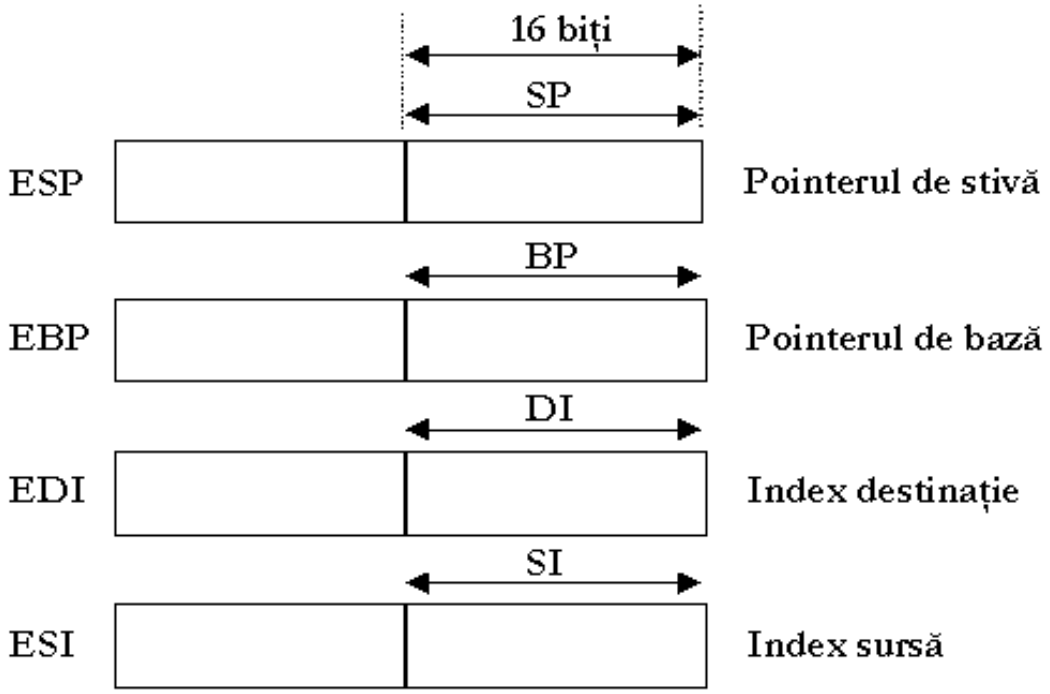

Figura 2. Regiştrii de uz general index şi pointer

Ne vom concentra în continuare atenţia asupra regiştrilor generali pe 16 biţi; fiecare dintre aceştia poate stoca o valoare pe 16 biţi, poate fi folosit pentru stocarea unei valori din memorie sau poate fi utilizat pentru operaţii aritmetice şi logice. Spre exemplu, următoarele instrucţiuni:

```
…
MOV  BX, 2
MOV  DX, 3
```

```
ADD  BX, DX
…
```

încarcă valoarea 2 în registrul BX, valoarea 3 în registrul DX, adună cele două valori iar rezultatul (5) este memorat în registrul BX. În exemplul anterior putem utiliza oricare dintre regiştrii de uz general în locul regiştrilor BX şi DX. În afara proprietăţii de a stoca valori şi de a folosi drept operanzi sursă sau destinaţie pentru instrucţiunile de manipulare a datelor, fiecare dintre cei 8 regiştri de uz general au propria "personalitate". Vom vedea în continuare care sunt caracteristicile specifice fiecăruia dintre regiştrii de uz general.

Registrul AX (EAX)

Registrul AX (EAX) este denumit şi registrul *acumulator*, fiind principalul registru de uz general utilizat pentru operaţii aritmetice, logice şi de deplasare de date. Totdeauna operaţiile de înmulţire şi împărţire presupun implicarea registrului AX. Unele dintre instrucţiuni sunt optimizate pentru a se executa mai rapid atunci când este folosit AX. În plus, registrul AX este folosit şi pentru toate transferurile de date de la/către porturile de Intrare/Ieşire. Poate fi accesat pe porţiuni de 8, 16 sau 32 de biţi, fiind referit drept AL (cei mai puţin semnificativi 8 biţi din AX), AH (cei mai semnificativi 8 biţi din AX), AX (16 biţi) sau EAX (32 de biţi). Prezentăm în continuare alte câteva exemple de instrucţiuni ce utilizează registrul AX. De remarcat este faptul că transferurile de date se fac pentru instrucţiunile (denumite şi *mnemonice*) Intel de la dreapta spre stânga, exact invers decât la Motorola (vom vedea şi alt exemplu asemănător la scrierea datelor în memorie sub format diferit la Motorola faţă de Intel), unde transferul se face de la stânga la dreapta.

Instrucţiunea: **MOV AX, 1234H** încarcă valoarea 1234H (4660 în zecimal) în registrul acumulator AX. După cum spuneam, cei mai puţini semnificativi 8 biţi ai registrului AX sunt identificaţi de AL (A-Low) iar cei mai semnificativi 8 biţi ai aceluiaşi registru sunt identificaţi ca fiind AH (A-High). Acest lucru este utilizat pentru a lucra cu date pe un octet, permiţând ca registrul AX să fie folosit pe postul a doi regiştri separaţi (AH şi AL). Aceeaşi regulă este valabilă şi pentru regiştrii de uz general BX, CX, DX. Următoarele trei instrucţiuni setează registrul AH cu valoarea 1, incrementează cu 1 această valoare şi apoi o copiază în registrul AL:

```
MOV  AH, 1
INC  AH
MOV AL, AH
```

Valoarea finală a registrului AX va fi 22 (AH = AL = 2).

Registrul BX (EBX)

Registrul BX (Base), sau registrul de bază poate stoca adrese pentru a face referire la diverse structuri de date, cum ar fi vectorii stocaţi în memorie. O valoare reprezentată pe 16 biţi stocată în registrul BX poate fi utilizată ca fiind o porţiune din adresa unei locaţii de memorie ce va fi accesată. Spre exemplu, următoarele instrucţiuni încarcă registrul AH cu valoarea din memorie de la adresa 21.

```
MOV  AX,  0
MOV  DS, AX
MOV  BX, 21
MOV  AH, [ BX ]
```

Se observă că am încărcat valoarea 0 în registrul DS înainte de a accesa locaţia de memorie referită de registrul BX. Acest lucru este datorat segmentării memoriei (segmentare discutată mai în detaliu în secţiunea consacrată regiştrilor de segment); implicit, atunci când este folosit ca pointer de memorie, BX face referire relativă la registrul de segment DS (adresa la care face referire este o adresă relativă la adresa de segment conţinută în registrul DS).

Registrul CX (ECX)

Specializarea registrului CX (Counter) este numărarea; de aceea, el se numeşte şi registrul contor. De asemenea, registrul CX joacă un rol special atunci când se foloseşte instrucţiunea LOOP. Rolul de contor al registrului CX se observă imediat din exemplul următor:

```
MOV CX, 5
start:
…
<instrucţiuni ce se vor executa de 5 ori>
…
SUB CX, 1
JNZ start
```

Deoarece valoarea iniţială a lui CX este 5, instrucţiunile cuprinse între eticheta start şi instrucţiunea JNZ se vor executa de 5 ori (până când registrul CX devine 0). Instrucţiunea SUB CX, 1 decrementează registrul CX cu valoarea 1 iar instrucţiunea JNZ start determină saltul înapoi la eticheta *start* dacă CX nu are valoarea 0. În limbajul microprocesorului există şi o instrucţiune specială legată de ciclare. Aceasta este instrucţiunea LOOP, care este folosită în combinaţie cu registrul CX. Liniile de cod următoare sunt echivalente cu cele anterioare, dar aici se utilizează instrucţiunea LOOP:

```
MOV CX, 5
start:
…
<instrucţiuni ce se vor executa de 5 ori>
…
LOOP start
```

Se observă că instrucţiunea LOOP este folosită în locul celor două instrucţiuni SUB şi JNZ anterioare; LOOP decrementează automat registrul CX cu 1 şi execută saltul la eticheta specificată (*start*) dacă CX este diferit de zero, totul într-o singură instrucţiune.

Registrul DX (EDX)

Registrul de uz general DX (Data register), denumit şi registrul de date, poate fi folosit în cazul transferurilor de date Intrare/Ieşire sau atunci când are loc o operaţie de înmulţire sau de împărţire. Instrucţiunea **IN AL, DX** copiază o valoare de tip Byte dintr-un port de intrare, a cărui adresă se află în registrul DX. Următoarele instrucţiuni determină scrierea valorii 101 în portul I/O 1002:

```
...
MOV  AL, 101
MOV  DX, 1002
OUT  DX, AL
…
```

Referitor la operaţiile de înmulţire şi împărţire, atunci când împărţim un număr pe 32 de biţi la un număr pe 16 biţi, cei mai semnificativi 16 biţi ai deîmpărţitului trebuie să fie în DX. După împărţire, restul împărţirii se va afla în DX. Cei mai puţin semnificativi 16 biţi ai deîmpărţitului trebuie să fie în AX iar câtul împărţirii va fi în AX. La înmulţire, atunci când se înmulţesc două numere pe 16 biţi, cei mai semnificativi 16 biţi ai produsului vor fi stocaţi în DX iar cei mai puţin semnificativi 16 biţi în registrul AX.

Registrul SI

Registrul SI (Source Index) poate fi folosit, ca şi BX, pentru a referi adrese de memorie. De exemplu, secvenţa de instrucţiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
MOV  AL, [ SI ]
```

Încarcă valoarea (pe 8 biţi) din memorie de la adresa 33 în registrul AL. Registrul SI este, de asemenea, foarte folositor atunci când este utilizat în legătură cu instrucţiunile dedicate tipului string (şir de caractere). Secvenţa următoare :

```
CLD
MOV  AX, 0
MOV  DS, AX
MOV  SI, 33
LODSB
```

nu numai că încarcă registrul AX cu valoarea de la adresa de memorie referită de registrul SI, dar adună, de asemenea, valoarea 1 la SI. Acest lucru este deosebit de eficient atunci când se accesează secvenţial o serie de locaţii de memorie, cum ar fi şirurile de caractere. Instrucţiunile de tip string se pot repeta de mai multe ori, astfel încât o singură instrucţiune poate avea ca efect sute sau mii de operaţii.

Registrul DI

Registrul DI (Destination Index) este utilizat în mod asemănător registrului SI. În secvenţa de instrucţiuni următoare:

```
MOV  AX, 0
MOV  DS, AX
MOV  DI, 1000
ADD  BL, [ DI ]
```

se adună la registrul BL valoarea pe 8 biţi stocată la adresa 1000. Registrul DI este puţin diferit faţă de registrul SI în cazul instrucţiunilor de tip string; dacă SI este întotdeauna pe post de pointer sursă de memorie, registrul DI serveşte drept pointer destinaţie de memorie. Mai mult, în cazul instrucţiunilor de tip string, registrul SI adresează memoria relativ la registrul de segment DS, în timp ce DI conţine referiri la memorie relativ la registrul de segment ES. În cazul în care SI şi DI sunt utilizaţi cu alte instrucţiuni, ei fac referire la registrul de segment DS.

Registrul BP

Pentru a înţelege mai bine rolul regiştrilor BP şi SP, a sosit momentul să spunem câteva lucruri despre porţiunea de memorie denumită stivă (în engleză *stack*). Stiva (vezi figura 3) reprezintă o porţiune specială de locaţii adiacente din memorie. Aceasta este conţinută în cadrul unui segment de memorie şi identificată de un selector de segment memorat în registrul SS (cu excepţia cazului în care se foloseşte modelul nesegmentat de memorie în care stiva poate fi localizată oriunde în spaţiul de adrese liniare al programului). Stiva este o porţiune a memoriei unde valorile pot

fi stocate şi accesate pe principul LIFO (Last In – First Out), drept urmare ultima valoare stocată în stivă este prima ce va fi citită din stivă. De regulă, stiva este utilizată la apelul unei proceduri sau la întoarcerea dintr-un apel de procedură (principalele instrucţiuni folosite sunt CALL şi RET).

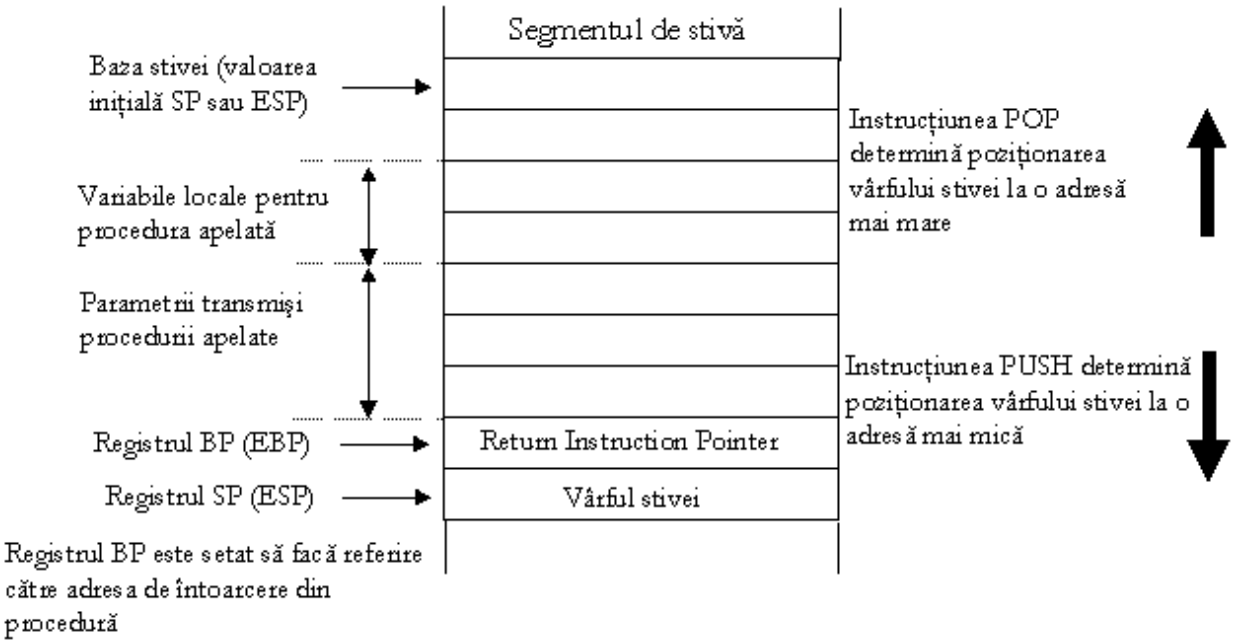

Figura 3. Structura stivei

Registrul pointer de bază, BP (Base Pointer) poate fi utilizat ca pointer de memorie precum regiştrii BX, SI şi DI. Diferenţa este aceea că, dacă BX, SI şi DI sunt utilizaţi în mod normal ca pointeri de memorie relativ la segmentul DS, registrul BP face referire relativ la segmentul de stivă SS. Principiul este următorul: o modalitate de a trece parametrii unei subrutine este aceea de a utiliza stiva (acest lucru se întâmplă în mod obişnuit în limbajele de nivel înalt - C, spre exemplu). Dacă stiva se află în porţiunea de memorie referită de registrul de segment SS (Stack Segment), datele se află în mod normal în segmentul de memorie referit de către DS, registrul segment de date. Deoarece BX, SI şi DI se referă la segmentul de date, nu există o modalitate eficientă de a folosi regiştrii BX, SI, DI pentru a face referire la parametrii salvaţi în stivă din cauză că stiva este localizată într-un alt segment de memorie. Registrul BP oferă rezolvarea acestei probleme asigurând adresarea în segmentul de stivă. Spre exemplu, instrucţiunile:

```
PUSH  BP
MOV  BP, SP
MOV  AX, [ BP+4 ]
```

fac să se acceseze segmentul de stivă pentru a încărca registrul AX cu primul parametru trimis de un apel C unei rutine scrise în limbaj de asamblare. În concluzie, registrul BP este conceput astfel încât să ofere suport pentru accesul la parametri, variabile locale şi alte necesităţi legate de accesul la porţiunea de stivă din memorie.

Registrul SP

Registrul SP (Stack Pointer), sau pointerul de stivă, reţine de regulă adresa de deplasament a următorului element disponibil în cadrul segmentului de stivă. Acest registru este, probabil, cel mai puţin „general" dintre regiştrii de uz general, deoarece este dedicat mai tot timpul administrării stivei.

Registrul BP face în fiecare clipă referire la vârful stivei – acest vârf al stivei reprezintă adresa locaţiei de memorie în care va fi introdus următorul element în stivă. Acţiunea de a introduce un nou element în stivă se numeşte „împingere" (în engleză *push*); de aceea, instrucţiunea respectivă poartă numele de PUSH. În mod asemănător, operaţia de scoatere a unui element din vârful stivei poartă, în engleză, numele de *pop*, iar instrucţiunea echivalentă operaţiei se numeşte POP. În figurile 3 şi 4 sunt ilustrate modificările survenite în conţinutul stivei şi al regiştrilor SP, BX şi CX ca urmare a execuţiei instrucţiunilor următoare (se presupune că registrul SP are iniţial valoarea 1000):

```
MOV  BX, 9
PUSH  BX
MOV  CX, 10
PUSH  CX
POP  BX
POP  CX
```

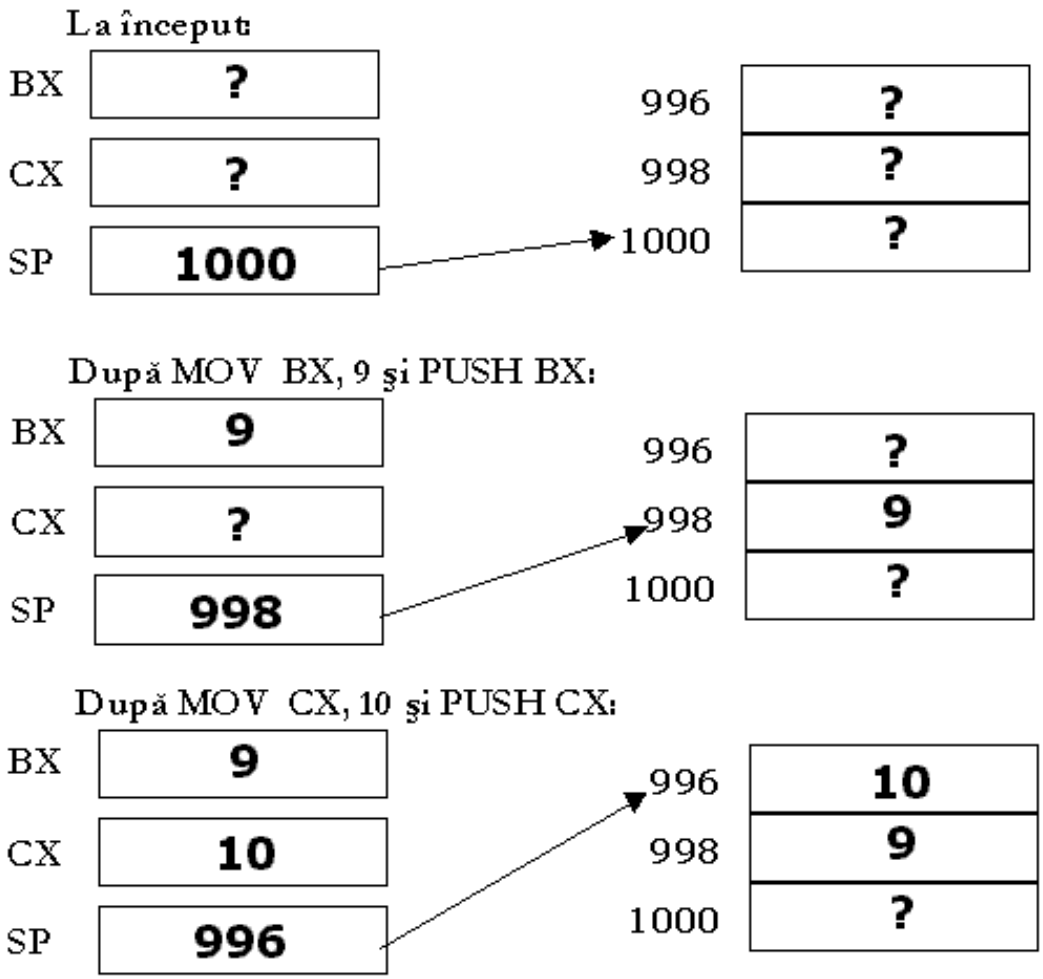

Figura 3. Modalitatea de funcţionare a stivei după execuţia primelor 4 instrucţiuni

Este permisă stocarea valorilor în registrul SP precum şi modificarea valorii sale prin adunare sau scădere la fel ca şi în cazul celorlalţi regiştri de uz general; totuşi, acest lucru nu este recomandat dacă nu suntem foarte siguri de ceea ce facem. Prin modificarea registrului SP, vom modifica adresa de memorie a vârfului stivei, ceea ce poate avea efecte neprevăzute, aceasta pentru că instrucţiunile PUSH şi POP nu reprezintă unicele modalităţi de utilizare a stivei. Indiferent dacă apelăm o subrutină sau ne întoarcem dintr-un astfel de apel de subrutină, fie procedură sau funcţie, în acest caz este folosită stiva. Unele resurse de sistem, precum tastatura sau ceasul de sistem, pot folosi stiva în momentul trimiterii unei întreruperi la microprocesor. Acest lucru presupune că stiva este folosită continuu, deci dacă se modifică registrul SP (adică adresa stivei), datele din noile locaţii de memorie nu vor mai fi cele corecte. În concluzie, registrul SP nu trebuie modificat în mod direct; el este modificat automat în urma instrucţiunilor POP, PUSH, CALL, RET. Oricare dintre ceilalţi regiştri de uz general pot fi modificaţi în mod direct în orice moment.

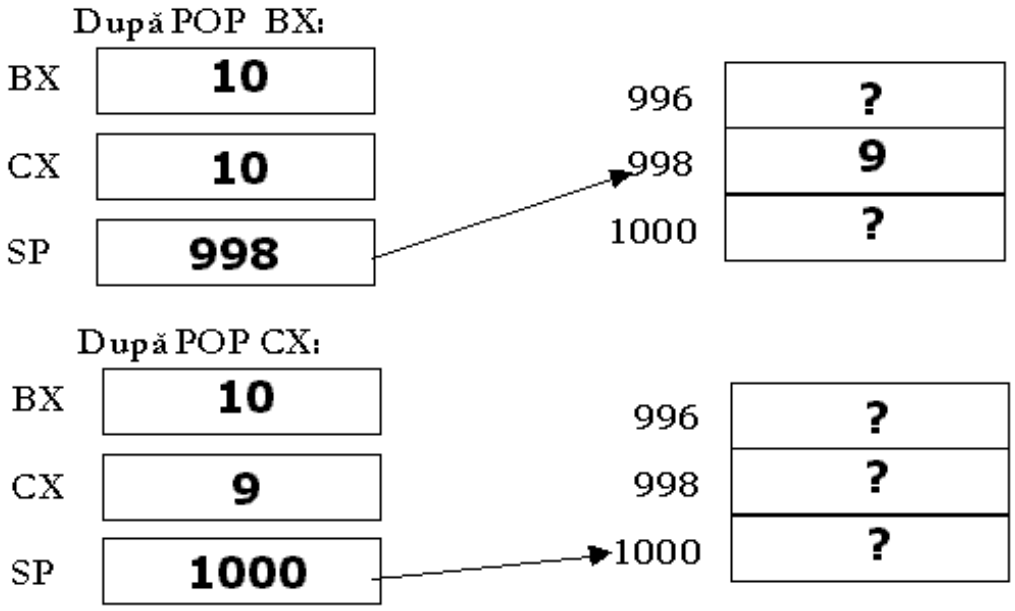

Figura 4. Funcţionarea stivei după ultimile două instrucţiuni POP

Registrul pointer de instrucţiuni (IP)

Registrul pointer de instrucţiuni (IP – Instruction Pointer, vezi figura 5) este folosit, întotdeauna, pentru a stoca adresa următoarei instrucţiuni ce va fi executată de către microprocesor. Pe măsură ce o instrucţiune este executată, pointerul de instrucţiune este incrementat şi se va referi la următoarea adresă de memorie (unde este stocată următoarea instrucţiune ce va fi executată). De regulă, instrucţiunea ce urmează a fi executată se află la adresa imediat următoare instrucţiunii ce a fost executată, dar există şi cazuri speciale (rezultate fie din apelul unei subrutine prin instrucţiunea CALL, fie prin întoarcerea dintr-o subrutină, prin instrucţiunea RET). Pointerul de instrucţiuni nu poate fi modificat sau citit în mod direct; doar instrucţiuni speciale pot încărca acest registru cu o nouă valoare. Registrul pointer de instrucţiune nu specifică pe de-a întregul adresa din memorie a următoarei instrucţiuni ce va fi executată, din aceeaşi cauză a segmentării memoriei. Pentru a aduce o instrucţiune din memorie, registrul CS oferă o adresă de bază iar registrul pointer de instrucţiune indică adresa de deplasament plecând de la această adresă de bază.

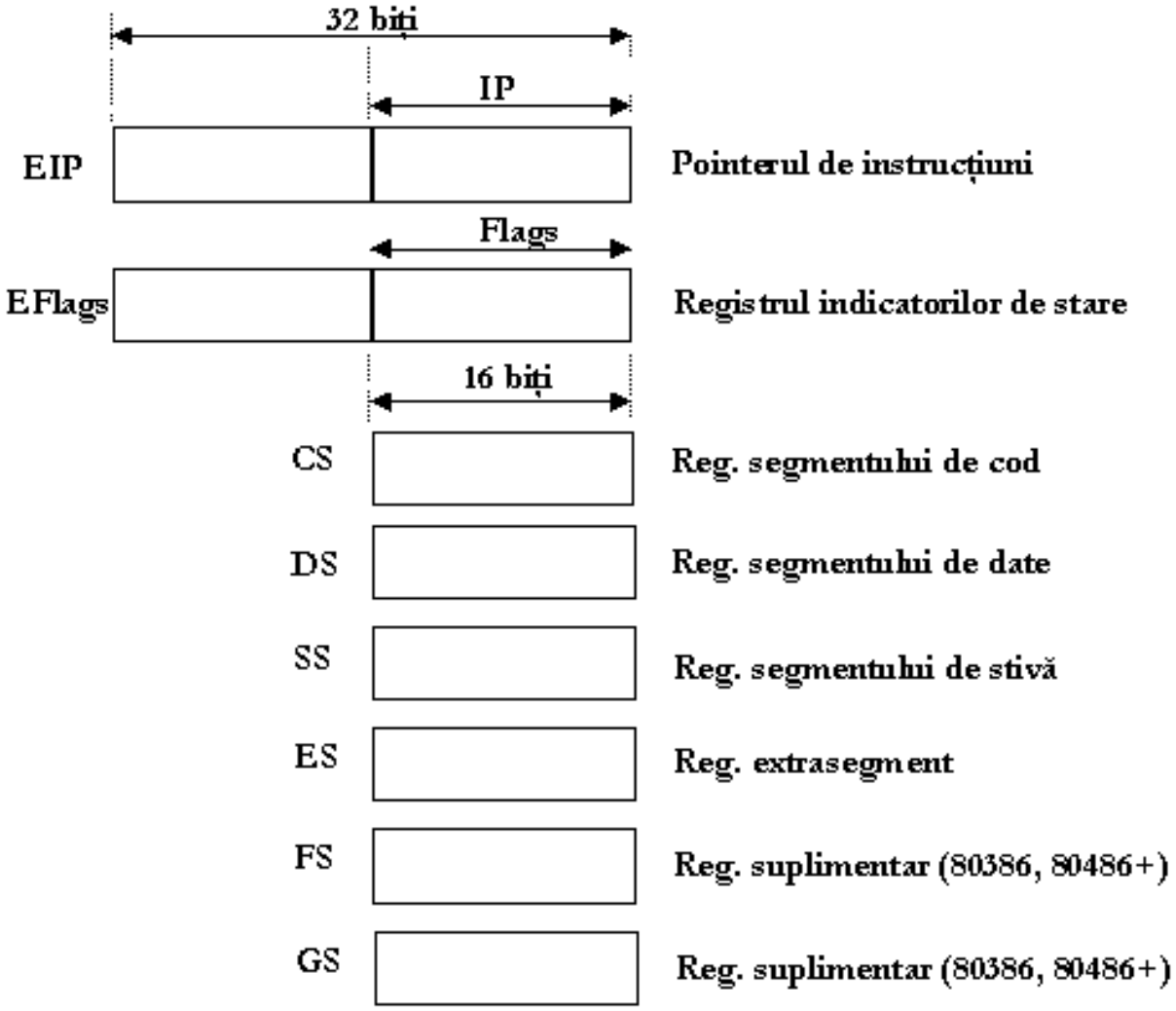

Figura 5. Regiştrii de segment, pointerul de instrucţiuni şi registrul indicatorilor de stare

Registrul indicatorilor de stare (FLAGS)

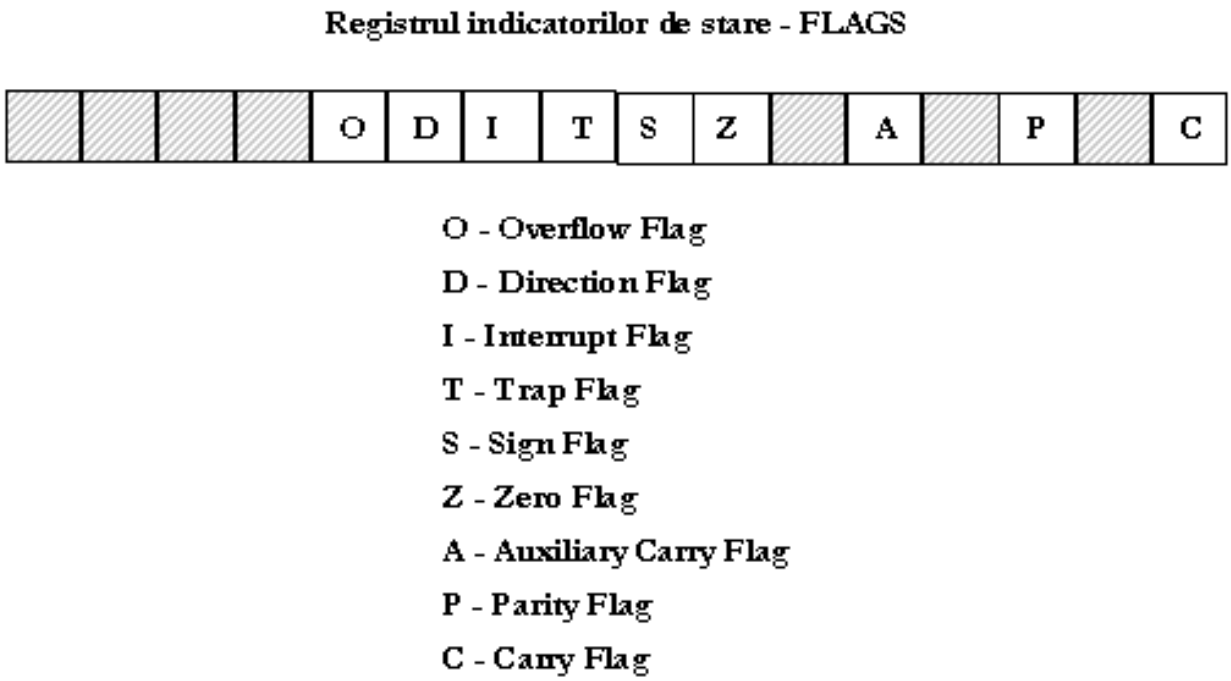

Figura 6. Registrul indicatorilor de stare - detaliu

Registrul indicatorilor de stare (FLAGS) pe 16 biţi conţine informaţii legate de starea microprocesorului precum şi de rezultatele ultimilor instrucţiuni executate. Un indicator de stare (*flag*) este în sine o locaţie de memorie de 1 bit ce indică starea curentă a microprocesorului şi modalitatea sa de operare. Un indicator se spune că "este setat" dacă are valoarea 1 şi "nu este setat" în caz contrar. Indicatorii de stare se modifică după execuţia unor instrucţiuni aritmetice sau logice. Exemple de indicatori de stare (vezi figura 6):

- C (Carry) indică apariţia unei cifre binare de transport în cazul unei adunări sau împrumut în cazul unei scăderi;
- O (Overflow) apare în urma unei operaţii aritmetice. Dacă este setat, înseamnă că rezultatul nu încape în operandul destinaţie;
- Z (Zero) indică faptul că rezultatul unei operaţii aritmetice sau logice ste zero;
- S (Sign) indică semnul rezultatului unei operaţii aritmetice;
- D (Direction) – când este zero, procesarea elementelor şirului se face de la adresa mai mică la cea mai mare, în caz contrar este invers;
- I (Interrupt) controlează posibilitatea microprocesorului de a răspunde la evenimente externe (apeluri de întrerupere);
- T (Trap) este folosit de programele de depanare (de tip debugger), activând sau nu posibilitatea execuţiei programului pas cu pas. Dacă este setat, UCP

întrerupe fiecare instrucțiune, lăsând programul depanator să executet programul respectiv pas cu pas;

- A (Auxiliary carry) suportă operații în codul BCD. Majoritatea programelor nu oferă suport pentru reprezentarea numerelor în acest format, de aceea se utilizează foarte rar;
- P (Parity) este stetat în conformitate cu paritatea biților cei mai puțin semnificativi ai unei operații cu date. Astfel, dacă rezultatul unei operații conține un număr par de biți 1, acest indicator este setat. Dacă numărul de biți 1 din rezultat este impar, atunci indicatorul PF este zero. Este folosit de regulă de programe de comunicații, dar Intel a introdus acest indicator nu pentru a îndeplini o anumită funcționalitate, ci pentru a asigura compatibilitatea cu vechile microprocesoare ale familiei x86.

Regiştrii de segment

Proprietățile regiștrilor de segment (vezi figura 5) sunt în strânsă legătură cu noțiunea de segmentare a memoriei. Premiza de la care se pleacă este următoarea: 8086 este capabil să adreseze 1MB de memorie, astfel că sunt necesare adrese pe 20 de biți pentru a cuprinde toate locațiile din spațiul de 1 MB de memorie. Totuși, registrele utilizate sunt registre pe 16 biți, deci a trebuit să se găsească o soluție pentru această problemă. Soluția găsită se numește *segmentarea memoriei*; în acest caz memoria de 1MB este împărțită în 16 segmente de câte 64 KB (16*64 KB = 1024 KB = 1 MB).

Noțiunea de segmentare a memoriei presupune utilizarea unor adrese de memorie formate din două părți. Prima parte reprezintă adresa segmentului iar cea de-a doua porțiune reprezintă adresa de deplasament, sau offset-ul (figura 7).

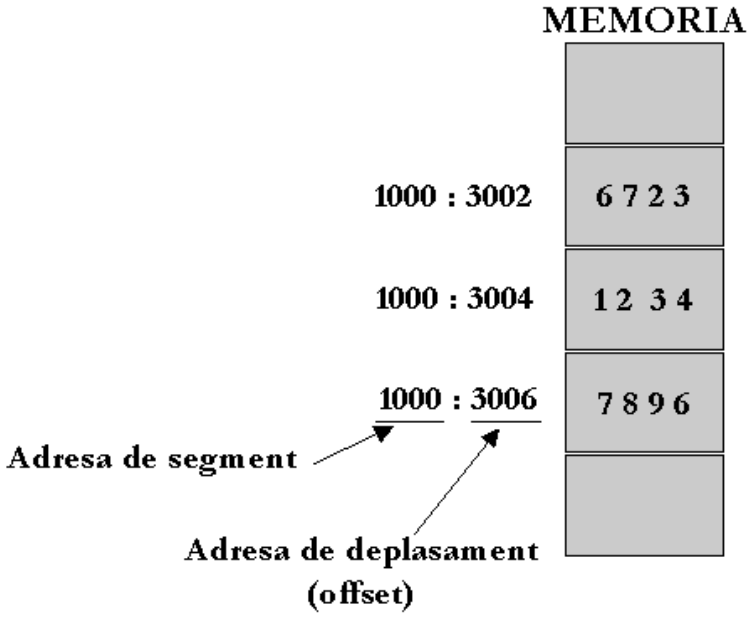

Figura 7. Cele două porțiuni ale unei adrese segmentate

Fiecare pointer de memorie pe 16 biţi este combinat cu conţinutul unui registru de segment pe 16 biţi pentru a forma o adresă completă pe 20 de biţi. Adresa de segment împreună cu adresa de deplasament sunt combinate în felul următor: valoarea de segment este deplasată la stânga cu 4 biţi (înmulţită cu 16 = 2^4) şi apoi adunată cu valoarea adresei de deplasament. Adresa astfel construită se numeşte adresă efectivă; fiind o adresă pe 20 de biţi poate accesa 2^{20} octeţi de memorie, adică 1 MB de memorie. Construirea unei adrese efective este prezentată în figura 8.

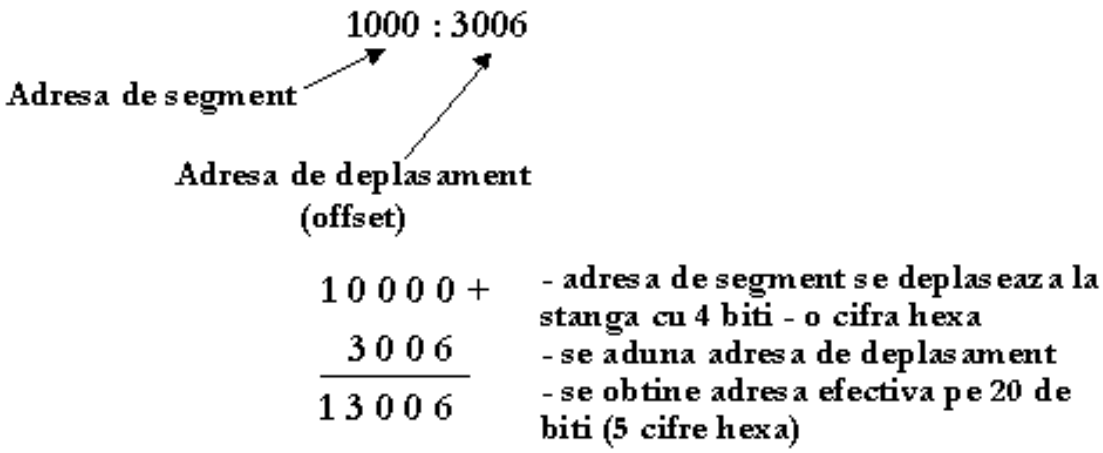

Figura 8. Exemplu de calcul al adresei efective

Registrul CS – acest registru face referire la începutul blocului de 64 KB de memorie în care se află codul programului (segmentul de cod). Microprocesorul 8086 nu poate aduce altă instrucţiune pentru execuţie decât cea definită de CS. Registrul CS poate fi modificat de un număr de instrucţiuni, precum instrucţiuni de salt, apel sau de întoarcere. El nu poate fi încărcat în mod direct cu o valoare, ci doar prin intermediul unui alt registru general.

Registrul DS – face referire către începutul segmentului de date, unde se află mulţimea de date cu care lucrează programul aflat în execuţie.

Registrul ES – face referire la începutul blocului de 64KB cunoscut sun denumirea de extra-segment. Acesta nu este dedicat nici unui scop anume, fiind disponibil pentru diverse acţiuni. Uneori acesta poate fi folosit pentru creearea unui bloc de memorie de 64 KB adiţional pentru date. Acest extra-segment lucrează foarte bine în cazul instrucţiunilor de tip STRING. Toate instrucţiunile de tip STRING ce scriu în memorie folosesc adresarea ES : DI ca adresă de memorie.

Registrul SS – face referire la începutul segmentului de stivă, care este blocul de 64 KB unde se află stiva. Toate instrucţiunile ce folosesc implicit registrul SP (instrucţiunile POP, PUSH, CALL, RET) lucrează în segmentul de stivă deoarece registrul SP este capabil să adreseze memoria doar în segmentul de stivă.

Formatul general al unei instrucţiuni în limbaj de asamblare

O linie de cod scrisă în limbaj de asamblare are următorul format general:

<nume> <instructiune/directiva> <operanzi> <;comentariu>

unde:

- **<nume>** - reprezintă un nume simbolic opţional;
- **<instructiune/directiva>** - reprezintă mnemonica (numele) unei instrucţiuni sau a unei directive;
- **<operanzi>** - reprezintă o combinaţie de unul, doi sau mai mulţi operanzi (sau chiar nici unul), care pot fi constante, referinţe de memorie, referinţe de regiştri, şiruri de caractere, în funcţie de structura particulară a instrucţiunii;
- **<;comentariu>** - reprezintă un comentariu opţional ce poate fi plasat după caracterul „;" până la sfârşitul liniei respective de cod.

Nume de variabile şi etichete

Numele folosite într-un program scris în limbaj de asamblare pot identifica variabile numerice, variabile şir de caractere, locaţii de memorie sau etichete. Spre exemplu, următoarea secvenţă de cod, care calculează valoarea lui trei factorial (3!=1x2x3=6) cuprinde câteva nume de variabile şi etichete:

```
.MODEL  small
.STACK  200h
.DATA
Valoare_Factorial  DW  ?
Factorial  DW  ?
.CODE

Trei_Factorial  PROC
MOV  ax, @data
MOV ds, ax
MOV  [Valoare_Factorial], 1
MOV [Factorial], 2
MOV  cx, 2
Ciclare:
MOV ax, [Valoare_Factorial]
MUL [Factorial]
MOV  [Valoare_Factorial], ax
INC [Factorial]
LOOP Ciclare
```

```
RET
Trei_Factorial ENDP
END
```

Numele *Valoare_Factorial* şi *Factorial* sunt utilizate pentru definirea a două variabile de tip word (pe 16 biţi), *Trei_Factorial* identifică numele procedurii (subrutinei) ce conţine codul pentru calculul factorialului, permiţând apelul său din altă parte a programului. *Ciclare* reprezintă un nume de etichetă, identificând adresa instrucţiunii MOV ax, [Valoare_Factorial], astfel încât instrucţiunea LOOP folosită mai jos să poată face un salt înapoi la această instrucţiune. Numele de variabile pot conţine următoarele caractere: literele a-z şi A-Z, cifrele de la 0-9 precum şi caracterele speciale _ (*underscore* – liniuţă de subliniere), @ (*„at”* în engleză – citit şi *„a rond”* sau *„coadă de maimuţă”*), $ şi ?. Se poate folosi si caracterul punct (“.”) drept prim caracter al numelui unei etichete. Cifrele 0-9 nu pot fi utilizate pe prima poziţie a numelui; de asemenea, nu pot fi folosite nume care să conţină un singur caracter $ sau ?. Fiecare nume poate fi definit *o singură dată* (numele sunt *unice*) şi pot fi utilizate ca operanzi de oricâte ori se doreşte într-un program. Un nume poate să apară într-un program singur pe o linie (linia respectivă nu mai conţine altă instrucţiune sau directivă). În acest caz, valoarea numelui este dată de adresa instrucţiunii sau directivei de pe linia următoare din program. De exemplu, în secvenţa următoare:

```
...
JMP scadere
...
scadere:
SUB AX, CX
...
```

următoarea instrucţiune care va fi executată după instrucţiunea **JMP scadere** va fi instrucţiunea **SUB AX, CX**. Exemplul anterior este echivalent cu secvenţa:

```
...
JMP scadere
...
scadere: SUB AX, CX
...
```

Există unele avantaje atunci când scriem instrucţiunile pe linii separate. În primul rând, atunci când scriem un nume de etichetă pe o singură linie, este mai uşor să folosim nume lungi de etichete fără a strica „forma” programului scris în limbaj de asamblare. În al doilea rând, este mai uşor să adăugăm ulterior o nouă instrucţiune în dreptul etichetei dacă aceasta nu este scrisă pe aceeaşi linie cu instrucţiunea.

Numele variabilelor sau etichetelor folosite într-un program nu trebuie să se confunde cu numele *rezervate* de asamblor, cum ar fi numele de directive şi instrucţiuni, numele regiştrilor, etc. De exemplu, o declaraţie de genul:

```
...
ax  DW  0
BYTE:
...
```

nu poate fi acceptată, deoarece AX este numele registrului acumulator, AX, iar BYTE reprezintă un cuvânt cheie rezervat.

Orice nume de etichetă ce apare pe o linie fără instrucţiuni sau apare pe o linie cu instrucţiuni trebuie să aibă semnul „:” după numele ei. Tototdată, se încearcă să se dea un nume sugestiv etichetelor din program. Fie următorul exemplu:

```
...
CMP AL, 'a'
JB  Nu_este_litera_mica
CMP AL, 'z'
JA  Nu_este_litera_mica
SUB AL, 20H         ; se transforma in litera mare
Nu_este_litera_mica:
…
```

comparativ cu:

```
...
CMP AL, 'a'
JB  x5
CMP AL, 'z'
JA  x5
SUB AL, 20H         ; se transforma in litera mare
x5:
…
```

Dacă în primul caz am folosit un nume sugestiv de etichetă (Nu_este_litera_mica), în cazul al doilea, identic din punct de vedere al funcţionalităţii cu primul, eticheta a fost denumită x5, absolut nesugestiv!

Observaţie:

Limbajul de asamblare nu este *case sensitive*. Aceasta semnifică faptul că, într-un program scris în limbaj de asamblare, numele de variabile, etichete, instrucţiuni, directive, mnemonice, etc., pot fi scrise atât cu litere mari cât şi cu litere mici, nefăcându-se diferenţa între ele (*Nu_este_litera_mica* este acelaşi lucru cu *nu_este_litera_mica* sau *Nu_Este_Litera_Mica*, etc.).

Directive de segment simplificate

Datorită faptului că regiştrii microprocesorului 8086 sunt regiştri pe 16 biţi, s-a impus folosirea unor segmente de memorie de câte 64Ko (maxim cât se poate adresa având la dispoziţie 16 biţi - 64Ko=2^16=65536). Într-un program scris în limbaj de asamblare (vom folosi în continuare prescurtarea ASM) există trei segmente: segmentul de cod, segmentul de date şi segmentul de stivă.

Directivele de segment (fie sub formă standard, fie sub formă simplificată) sunt necesare în orice program scris în limbaj de asamblare pentru a defini şi controla utilizarea segmentelor iar directiva END este folosită întotdeauna pentru a încheia codul programului.

Exemple de directive de segment simplificate sunt:

```
.STACK
.CODE
.DATA
.MODEL
DOSSEG
END
```

.STACK,.CODE,.DATA definesc, respectiv, segmentele de stivă, de cod şi de date.

De exemplu, **.STACK 200H** defineşte o stivă de 512 octeţi (în ASM valorile ce sunt încheiate cu litera H semnifică faptul că este vorba despre hexazecimal). O astfel de valoare pentru stivă este suficientă în mod normal; unele programe, însă (îndeosebi cele recursive) pot necesita dimensiuni mai mari ale stivei.

Directiva **.CODE** marchează începutul segmentului de cod.

Directiva **.DATA** marchează începutul segmentului de date, adică locul în care vom plasa variabilele de memorie. Reprezentativ aici este faptul că trebuie încărcat în mod explicit registrul de segment DS cu valoarea "@data" înaintea accesării locaţiilor de memorie în segmentul definit de .DATA. Având în vedere că un registru de segment poate fi încărcat fie dintr-un registru general fie dintr-o locaţie de memorie dar nu poate fi încărcat direct cu o constantă, registrul de segment DS este încărcat în general printr-o secvenţă de 2 instrucţiuni:

```
...
mov ax, @data
mov ds, ax
...
```

(se poate folosi şi alt registru general în locul lui AX).

Secvenţa anterioară semnifică faptul că DS se va referi către segmentul de date ce începe cu directiva .DATA.

Considerăm în continuare un exemplu de program ce afişează textul memorat în DataString pe ecran:

```
;Program p01.asm
.MODEL small          ;se specifică modelul de memorie SMALL
.STACK 200H           ;se defineşte o stivă de 512 octeţi
.DATA                 ;se specifică începutul segmentului de
                      ;date
DataString DB 'Hello!$'  ;se      defineşte      variabila
                      ;DataString, iniţializată cu valoarea
                      ;"Hello!"
.CODE                 ;începutul segmentului de cod al
                      ;programului
ProgramStart:         ;orice program are o etichetă de
                      ;început
mov bx,@data          ;secvenţa ce setează registrul DS să
                      ;facă referire la segmentul de date ce
                      ;începe cu .DATA
mov ds,bx
mov dx, OFFSET DataString ;se încarcă în DX adresa
;variabilei DataString
mov ah,09             ;codul funcţiei DOS de afişare a unui
                      ;string
int 21H               ;apelul DOS de afişare a string-ului
mov ah, 4cH           ;codul funcţiei DOS de terminare a
                      ;programului
int 21H               ;apelul DOS de terminare a programului
END ProgramStart      ;directiva de terminare a codului
                      ;programului
```

Explicaţii:

1. Se pot introduce comentarii într-un program ASM prin folosirea ";". Tot ce urmează după ";" şi până la sfârşitul liniei este considerat comentariu.

2. Nu are importanţă dacă programul este scris folosind litere mari sau mici (nu este "case sensitive").

3. Fără cele două instrucţiuni care setează registrul DS către segmentul definit de .DATA, funcţia de afişare a string-ului nu ar fi funcţionat cum trebuie. Variabila DataString se află în segmentul .DATA şi nu poate fi accesată dacă DS nu este poziţionat către acest segment. Acest lucru se explică în modul următor: atunci când facem apelul DOS de afişare a unui string, trebuie să parcurgem întreaga adresă de tipul segment:offset a string-ului în DS:DX. De aceea, de abia după ce am încărcat DS cu segmentul .DATA şi DX cu adresa (offset-ul) lui DataString avem o referinţă completă segment:offset către DataString.

Observaţii.

Nu trebuie să încărcăm în mod explicit registrul de segment CS deoarece DOS face acest lucru automat în momentul când rulăm un program. Astfel, dacă CS nu ar fi deja setat la momentul execuţiei primei instrucţiuni din program, procesorul nu ar şti unde să găsească instrucţiunea şi programul nu ar rula niciodată. În mod asemănător, registrul de segment SS este setat de DOS înainte de execuţia programului şi de regulă rămâne nemodificat pe perioada execuţiei programului.

Cu registrul de segment DS lucrurile stau altfel. În timp ce registrul CS se referă la intrucţiuni (cod), SS se referă ("pointează") la stivă, DS "pointează" la date. Programele nu manipulează direct instrucţiuni sau stive dar au de-a face în mod direct cu date. De asemenea, programele vor acces la date situate în segmente diferite în orice moment. Se poate dori încărcarea în DS a unui segment, accesarea datelor din acel segment şi apoi încărcarea lui DS cu un alt segment pentru a accesa un bloc diferit de date. În programe mici sau medii nu vom avea nevoie de mai mult de un segment de date dar programe mai complexe folosesc deseori segmente de date multiple.

Următorul program va afişa un caracter pe ecran, folosind încărcarea registrului ES în locul lui DS.

```
;Program p02.asm
.MODEL small
.STACK 200H
.DATA
OutputChar DB 'B'        ;definirea variabilei OutputChar
                         ;iniţializată cu valoarea "B"
.CODE
ProgramStart:
mov dx, @data
mov es, dx          ;spre deosebire de programul anterior,
                    se foloseşte ES pentru specificarea
                    segmentului de date
mov bx, offset OutputChar     ;se încarcă BX cu adresa
                              ;variabilei OutputChar
mov dl, es:[bx]          ;se încarcă AL cu valoarea de la
                         ;adresa explicită es:[bx]
                         ;(adresare indexată)
mov ah,02                ;codul funcţiei DOS de afişare a
                         ;unui caracter
int 21H             ;apelul DOS de execuţie a afişării
mov ah, 4cH         ;codul funcţiei DOS de terminare a
                    ;programului
int 21H             ;apelul DOS de terminare a programului
END ProgramStart    ;directiva de terminare a codului
                    ;programului
```

DOSSEG este directiva ce face ca segmentele dintr-un program să fie grupate conform convenţiilor Microsoft de adresare a segmentelor.

Directiva **.MODEL**

Este directiva ce specifică modelul de memorie pentru un program ASM ce foloseşte directive de segment simplificate.

Definiţii: "near" înseamnă adresa (offset-ul) pe 16 biţi din cadrul aceluiaşi segment, în timp ce "far" înseamnă o adresă completă de tip segment:offset, din cadrul altui segment decât cel curent.

Modelele de memorie ce se pot specifica prin intermediul directivei .MODEL sunt:

- ***tiny*** - atât codul cât şi datele programului încap în acelaşi segment de 64Ko. Atât codul cât şi datele sunt de tip near.
- ***small*** - codul programului trebuie să fie într-un singur segment de 64Ko şi datele într-un bloc separat de 64Ko; codul şi datele sunt near
- ***medium*** - codul programului poate fi mai mare decât 64Ko dar datele trebuie să fie într-un singur segment de 64 Ko. Codul este far, datele sunt near.
- ***compact*** - codul programului poate fi într-un singur segment, datele pot fi mai mari de 64 Ko. Codul este near, datele sunt far.
- ***large*** - atât codul cât şi datele pot depăşi 64Ko, dar nici un masiv de date nu poate depăşi 64 Ko. Atât codul cât şi datele sunt far.
- ***huge*** - atât codul cât şi datele pot depăşi 64Ko şi masivele de date pot depăşi 64 Ko. Atât codul cât şi datele sunt far. Pointerii la elementele dintr-un masiv sunt far.

În continuare sunt prezentate câteva exemple legate de modalităţile de declarare a variabilelor şi de adresare a memoriei.

```
var1 DW 01234h      ;se defineste o variabila word cu
                    ;valoarea 1234h
var2 DW 01234       ;se defineste o variabila word cu
                    ;valoarea zecimala 1234 (4D2 in hexa)
var3 RESW 1         ;se rezerva spatiu pentru o variabila
                    ;word (de valoare 0)
var4 DW ABCDh       ;atribuire ilegala!

 mesajsco2 DB 'SCO 2 este cursul preferat!'

...start:

  mov ax,cs         ; setarea  segmentului de date
  mov ds,ax         ; DS=CS
```

```
     ; orice referinta de memorie se presupune ca este relativa
la segmentul DS

        mov ax,[var2]           ; AX <- var2
                                ; == mov ax,[2]

        mov si,var2      ;se  foloseste  SI  ca  pointer  catre
                         var2 (cod C echivalent SI=&var2)

        mov ax,[si]      ;se citeste din memorie valoarea lui
                         ;var2 (*(&myvar2))
                         ;(referinta indirecta)

        mov bx,mesajsco2 ; BX este pointer la un string
                         ; (cod C echivalent:  BX=&mesajsco2)

        dec BYTE [bx+1]  ; transforma 'C' in 'B' !

        mov si, 1        ; Foloseste SI cu rol de index
        inc byte [mesajsco2+SI]  ; == inc byte [SI + 8]
                                 ; == inc byte [9]

     ; Memoria poate fi adresata folosindu-se 4 registri:
     ; SI  -> Implica DS
     ; DI  -> Implica DS
     ; BX  -> Implica DS
     ; BP  -> Implica SS !  (nu este foarte des utilizat)
     ;
     ;Exemple:

     mov ax,[bx]    ; ax <- word in memorie referit de BX
     mov al,[bx]    ; al <- byte in memorie referit de BX
     mov ax,[si]    ; ax <- word referit de SI
     mov ah,[si]    ; ah <- byte referit de SI
     mov cx,[di]    ; di <- word referit de DI

     mov ax,[bp]    ; AX <- [SS:BP] Operatie cu stiva!

     ; In plus, sunt permise BX+SI si BX+DI:

     mov ax,[bx+si]
     mov ch,[bx+di]

     ; Deplasamente pe 8 sau 16 biti:

     mov ax,[23h]        ; ax <- word in memorie DS:0023
     mov ah,[bx+5]       ; ah <- byte in memorie [DS:BX+5]
     mov ax,[bx+si+107]  ; ax <- word la adresa [DS:BX+SI+107]
     mov ax,[bx+di+47]   ; ax <- word la adresa [DS:BX+DI+47]
```

```
; ATENTIE: copierea din memorie in memorie este ilegala!
;Totdeauna trebuie sa se treaca valoarea copiata printr-un
;registru

mov [bx],[si]   ;Ilegal
mov [di],[si]   ;Ilegal

; Caz special: operatiile cu stiva!

pop word [var]         ; var <- [SS:SP]
```

Adrese de memorie şi valori

Un program scris în limbaj de asamblare se poate referi fie la o adresă de memorie (OFFSET = DEPLASAMENT), fie la o valoare stocată de variabilă în memorie. Din păcate, limbajul de asamblare nu este nici strict, nici intuitiv cu privire la modurile în care aceste două tipuri de referire sunt făcute şi, drept urmare, referirile la OFFSET sau la valoare sunt deseori confundate. În figura 9 sunt ilustrate conceptele de adresă de deplasament (offset) şi valoare stocată în memorie.

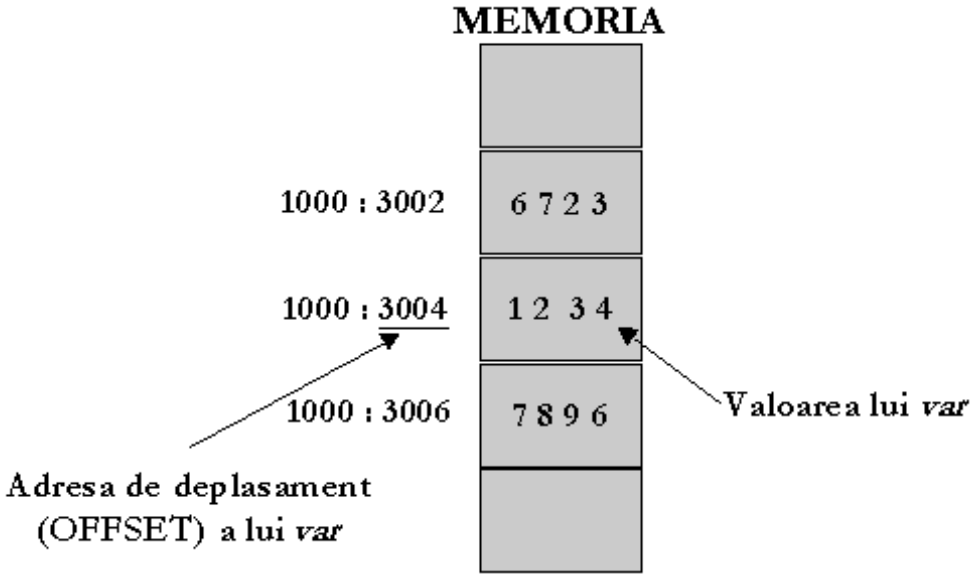

Figura 9. Ilustrarea noţiunilor de adresă de deplasament şi valoare stocată în memorie

Deplasamentul unei variabile de memorie *var* de dimensiune word este valoarea constantă 5004H, obţinută cu operatorul OFFSET. Spre exemplu, instrucţiunea:

MOV BX, OFFSET var

Încarcă valoarea 5004H în registrul BX. Valoarea 5004H nu se modifică; ea este construită în cadrul instrucţiunii. Valoarea lui *var* este 1234H, citită din memorie la adresa dată de offset-ul 5004H din segmentul de date. O modalitatea de citire a acestei valori este de a încărca registrele BX, SI, DI sau BP cu offset-ul lui *var* şi apoi folosirea registrului respectiv pentru adresarea memoriei. Instrucţiunile:

```
MOV BX, OFFSET var
MOV AX, [ BX ]
```

Au ca efect încărcarea valorii lui *var* (1234H) în registrul AX.

De asemenea, se poate încărca valoarea lui *var* direct în AX folosind:

```
MOV AX, var
Sau
MOV AX, [ var ]
```

În timp ce valoarea deplasamentului rămâne constantă, valoarea 1234H nu este permanent asociată cu *var*. De exemplu, instrucţiunile:

```
MOV [ var ], 5555H
MOV AX, [ var ]
```

Au ca efect încărcarea valorii 5555H în registrul AX.

Cu alte cuvinte, în timp ce deplasamentul lui *var* este o valoare constantă ce descrie o adresă fixă dintr-un segment de date, valoarea variabilei *var* este un număr ce poate fi modificat şi care se află memorat la adresa (de memorie) respectivă. Instrucţiunile:

```
MOV [ var ], 1
ADD [ var ], 2
```

Modifică valoarea lui var la 3, dar instrucţiunea:

ADD OFFSET var, 2 este echivalentă cu **ADD 5002H, 2**, ceea ce este un lucru fără sens, deoarece este imposibil să se însumeze o constantă cu alta.

O problemă ce poate apărea adesea în timpul programării este aceea a omiterii lui OFFSET; de exemplu, dacă scriem **MOV SI, var** atunci când, de fapt, dorim încărcarea în SI a deplasamentului lui *var*. Nu va fi semnalată nici o eroare în acest caz, având în vedere că *var* este o variabilă de tip word. Totuşi, în momentul execuţiei programului, registrul SI va fi încărcat cu valoarea lui *var* (1234H), în loc de OFFSET, ceea ce poate conduce la rezultate imprevizibile. În acest caz, referirile la constantele de adresă vor fi precedate de OFFSET iar referirile la valori din memorie să fie cuprinse între paranteze drepte („[” şi „]”), eliminând astfel ambiguitatea.

Instrucţiuni ale microprocesorului Intel

Microprocesoarele din familia Intel x86 dispun de o serie impresionantă de instrucţiuni, asemeni tuturor procesoarelor din clasa procesoarelor CISC (Complex Instruction Set Computer). Instrucţiunile se pot împărţi în: instrucţiuni logice, aritmetice, de transfer şi de control. Prezentăm în continuare câteva exemple din fiecare clasă de instrucţiuni.

Instrucţiuni logice

Instrucţiunile logice implementează funcţiile logice de bază, pe un octet sau pe cuvânt. Ele acţionează bit cu bit, deci se aplică funcţia logică respectivă tuturor biţilor sau perechilor de biţi corespunzători operanzilor. Instrucţiunile logice sunt următoarele:

- **NOT**: A =~A
- **AND**: A &= B
- **OR**: A |= B
- **XOR**: A ^= B
- **TEST**: A & B

De regulă, instrucţiunile logice au efect asupra indicatorilor de stare, cu excepţia instrucţiunii NOT, care nu are efect asupra nici unui flag (indicator de stare). Aceste efecte sunt următoarele:

- Se şterge indicatorul carry (C)
- Se şterge indicatorul overflow (O)
- Se setează zero flag (Z) dacă rezultatul este zero, sau îl şterge în caz contrar
- Se copiază bitul mai "înalt" al rezultatului în indicatorul sign (S)
- Se setează bitul de paritate (P) conform cu *paritatea* rezultatului

Instrucţiunea NOT

Este o instrucţiune cu un singur operand (instrucţiune *unară*), cu forma generală:

NOT *destinatie*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi. Instrucţiunea are ca efect inversarea (negarea) tuturor biţilor operandului, adică aducerea în forma codului invers - complement faţă de 1.

Instrucţiunea AND

Este o instrucţiune cu doi operanzi (instrucţiune *binară*), cu forma generală:

AND *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi, iar *sursa* poate fi registru, locaţie de memorie sau o constantă pe 8 sau 16 biţi. Instrucţiunea are ca efect operaţia: <destinatie> == <destinatie> AND <sursa>. Indicatorii de stare modificaţi sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucţiunea TEST (AND "non-distructiv")

Este o instrucţiune cu doi operanzi (instrucţiune *binară*), cu forma generală:

TEST *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi, iar *sursa* poate fi registru, locaţie de memorie sau o constantă pe 8 sau 16 biţi. Instrucţiunea are acelaşi efect ca şi instrucţiunea AND, cu deosebirea că nu se modifică operandul destinaţie, iar indicatorii de stare sunt modificaţi în acelaşi mod ca şi în cazul instrucţiunii AND.

Instrucţiunea OR

Este o instrucţiune cu doi operanzi, cu forma generală:

OR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi, iar *sursa* poate fi registru, locaţie de memorie sau o constantă pe 8 sau 16 biţi. Instrucţiunea are efectul: <destinatie> == <destinatie> OR <sursa>. Indicatorii de stare modificaţi sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucţiunea XOR (SAU-Exclusiv)

Este o instrucţiune cu doi operanzi, cu forma generală:

XOR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locaţie de memorie pe 8 sau 16 biţi, iar *sursa* poate fi registru, locaţie de memorie sau o constantă pe 8 sau 16 biţi. Instrucţiunea are efectul: <destinatie> == <destinatie> XOR <sursa>. Indicatorii de stare modificaţi sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit. Funcţia XOR, denumită SAU-Exclusiv (sau *anti-coincidenţă*) are valoarea logică 1 atunci când operanzii săi sunt diferiţi (unul are valoarea 0 iar celălalt valoarea 1) şi valoarea logică 0 când ambii operanzi au aceeaşi valoare (fie ambii au valoarea 0, fie ambii au valoarea 1).

Observaţie:

De cele mai multe ori, instrucţiunile AND şi OR sunt folosit pe post de „mascare" a datelor; în acest sens, o valoarea de tip „mască" (*mask*) este utilizată pentru a forţa anumiţi biţi să ia valoarea zero sau valoarea 1 în cadrul altei valori. O

astfel de „mască” logică are efect asupra anumitor biţi, în timp ce pe alţii îi lasă neschimbaţi. Exemple:

- Instrucţiunea **AND CL, 0Fh** – face ca cei mai semnificativi 4 biţi să ia valoarea 0, în timp ce biţii mai puţin semnificativi sunt lăsaţi neschimbaţi; astfel, dacă registrul CL are valoarea iniţială **1001 1101**, după execuţia instrucţiunii **AND CL, 0Fh** va avea valoarea **0000 1101**.
- Instrucţiunea **OR CL, 0Fh** – face ca cei mai puţin semnificativi 4 biţi să ia valoarea 1, în timp ce biţii mai semnificativi să rămână nemodificaţi. Dacă registrul CL are valoarea iniţială **1001 1101**, după execuţia instrucţiunii **OR CL, 0Fh** va avea valoarea **1001 1111**.

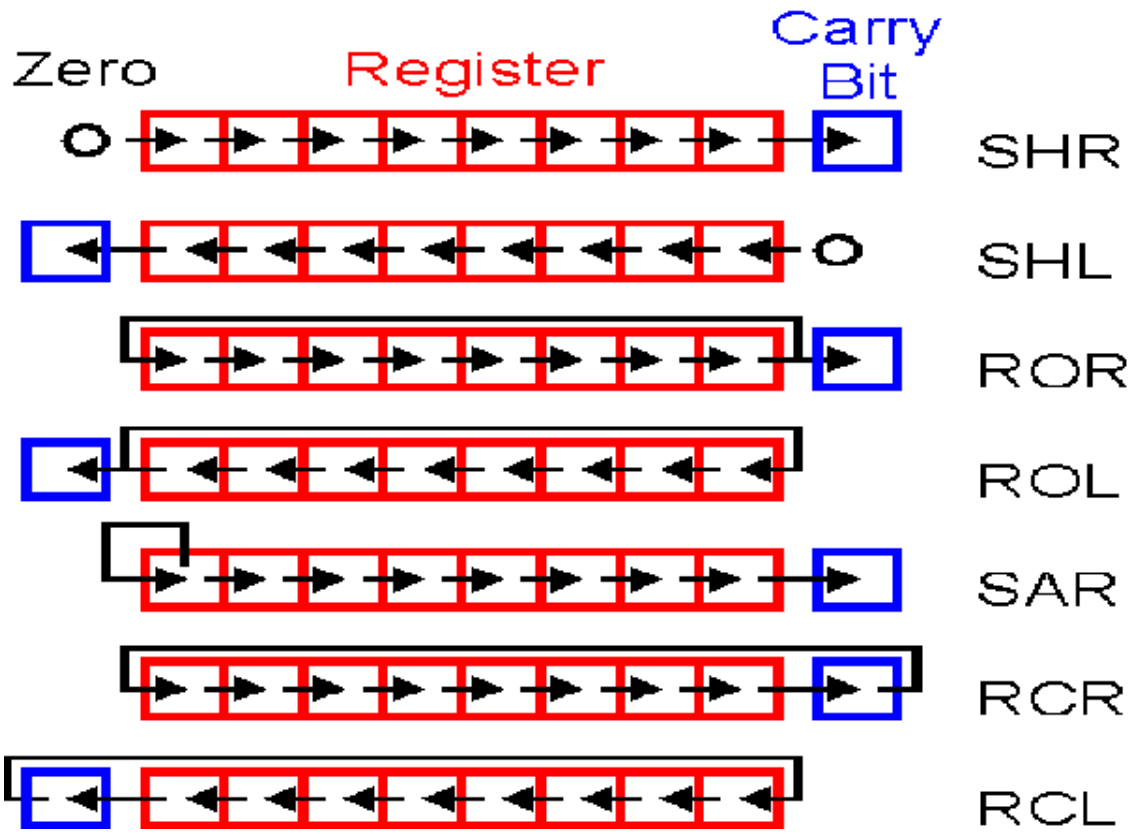

Figura 10. Instrucţiuni de deplasare şi de rotaţie

Instrucţiuni de deplasare şi de rotaţie

Acest tip de instrucţiuni (vezi figura **10**) permit realizarea operaţiilor de deplasare şi de rotaţie la nivel de bit. Ele au doi operanzi, primul operand fiind cel asupra căruia se aplică operaţia de deplasare pe biţi, iar cele de-al doilea (operandul *numărător* sau *contor*) semnifică numărul de biţi cu care se face această deplasare. Operaţiile se pot face de la dreapta spre stânga sau invers. Deplasarea înseamnă

translatarea tuturor biţilor din operand la stânga/dreapta, cu completarea unei valori fixe în poziţia rămasă liberă şi cu pierderea biţilor din dreapta/stânga. Rotaţia presupune translatarea biţilor din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biţii care se pierd în partea opusă. Sintaxa generală a instrucţiunilor de deplasare şi rotaţie este următoarea:

INSTR <operand> , <contor>

Unde **INSTR** reprezintă numele instrucţiunii, **<operand>** reprezintă un registru sau o locaţie de memorie pe 8 sau 16 biţi, iar **<contor>** semnifică numărul de biţi cu care se face deplasarea, adică fie o constantă, fie registrul CL (care îşi confirmă astfel rolul de numărător).

Observaţie.

Totdeauna există două modalităţi de deplasare:

- Prin folosirea unui contor efectiv – de exemplu: SHL AX, 1
- Prin folosirea registrului CL pe post de contor – de exemplu: SHL AX, CL

Instrucţiunea SHL/SAL (Shift Left/Shift Arithmetic Left)

Această instrucţiune translatează biţii operandului o poziţie la stânga de câte ori specifică operandul numărător. Poziţiile rămase libere prin deplasarea la stânga sunt umplute cu zerouri la bitul cel mai puţin semnificativ, în timp ce bitul cel mai semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de înmulţire cu o putere a lui 2 (în funcţie de numărul de biţi pentru care se face deplasarea la stânga).

Exemple:

1. Înmulţirea lui AX cu 10 (1010 în binar) (înmulţim cu 2 şi cu 8, apoi adunăm rezultatele)

```
shl     ax, 1       ; AX ori 2
mov     bx, ax      ; salvăm 2*AX în BX
shl     ax, 2       ; 2*AX(original) * 4 = 8*AX(original)
add     ax, bx      ; 2*AX + 8*AX = 10*AX
```

2. Înmulţirea lui AX cu 18 (10010 în binar) (înmulţim cu 2 şi cu 16, apoi adunăm rezultatele)

```
shl     ax, 1       ; AX ori 2
```

```
mov   bx, ax        ; salvăm 2*AX
shl   ax, 3         ; 2*AX(original) ori 8 = 16*AX(original)
add   ax, bx        ; 2*AX + 16*AX = 18*AX
```

Instrucţiunea SHR (Shift Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai puţin semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *fără semn* la o putere a lui 2 (dacă deplasarea se face cu *o poziţie* la dreapta, operaţia este echivalentă cu o împărţire la 2, dacă deplasarea se face cu două poziţii, operaţia este echivalentă cu o împărţire la 2^2, etc.). Operaţia de împărţire se execută *fără semn*, completându-se cu un bit 0 dinspre stânga (bitul cel mai semnificativ).

Instrucţiunea SAR (Shift Arithmetic Right)

Această instrucţiune translatează biţii operandului o poziţie la dreapta de câte ori specifică operandul numărător. Bitul cel mai semnificativ rămâne neschimbat, în timp ce bitul cel mai puţin semnificativ este copiat în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de împărţire *cu semn* la o puterea a lui 2 (în funcţie de numărul de biţi cu care se face deplasarea la dreapta).

Instrucţiunea RCL (Rotate through Carry Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga prin intermediul lui CF (Carry Flag). Astfel, cel mai semnificativ bit trece din operand în CF, apoi se deplasează toţi biţii din operand cu o poziţie la stânga iar CF original trece în bitul cel mai puţin semnificativ din operand.

Instrucţiunea ROL (Rotate Left)

Această instrucţiune determină o rotaţie a biţilor operandului către stânga. Astfel, cel mai semnificativ bit trece din operand în bitul cel mai puţin semnificativ.

Exemplu:

După execuţia instrucţiunilor:
ROL AX, 6
AND AX, 1Fh
Biţii 10-14 din AX se mută în biţii 0-4.

Instrucţiunea RCR (Rotate through Carry Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta prin intermediul lui CF (Carry Flag). Astfel, bitul din CF este scris înapoi în bitul cel mai semnificativ al operandului.

Instrucţiunea ROR (Rotate Right)

Această instrucţiune determină o rotaţie a biţilor operandului către dreapta. Bitul cel mai puţin semnificativ trece în bitul cel mai semnificativ.

Exemple:

```
MOV ax,3    ; Valori iniţiale             AX = 0000 0000 0000 0011
MOV bx,5    ;                             BX = 0000 0000 0000 0101
OR ax,9     ;     ax <- ax | 0000 1001    AX = 0000 0000 0000 1011
AND ax,10101010b  ; ax <- ax & 1010 1010  AX = 0000 0000 0000 1010
XOR ax,0FFh       ; ax <- ax ^ 1111 1111  AX = 0000 0000 1111 0101
NEG ax            ; ax <- (-ax)           AX = 1111 1111 0000 1011
NOT ax            ; ax <- (~ax)           AX = 0000 0000 1111 0100
OR ax,1           ; ax <- ax | 0000 0001  AX = 0000 0000 1111 0101
SHL ax,1    ; depl logică la stg cu 1 bit AX = 0000 0001 1110 1010
SHR ax,1    ; depl logică la dr cu 1 bit  AX = 0000 0000 1111 0101
ROR ax,1    ; rotaţie stg (LSB=MSB)       AX = 1000 0000 0111 1010
ROL ax,1    ; rotaţie dr (MSB=LSB)        AX = 0000 0000 1111 0101
MOV cl,3    ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHR ax,cl   ; împărţim AX la 8            AX = 0000 0000 0001 1110
MOV cl,3    ; folosim CL pt depl cu 3 biţi           CL = 0000 0011
SHL bx,cl   ; înmulţim BX cu 8            BX = 0000 0000 0010 1000
```

Instrucţiuni aritmetice

Instrucţiunea ADD (ADDition)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Cei doi operanzi nu pot fi însă în acelaşi timp locaţii de memorie. Rezultatul operaţiei este

următorul: <destinatie> == <destinatie> + <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Dacă apare ambiguitate la modul de exprimare al operanzilor (8 sau 16 biţi) se va folosi operatorul PTR.

Exemple:

```
ADD  AX, BX        ; adunare între regiştri – AX ← AX + BX
ADD  DL, 33h       ; adunare efectivă - DL ← DL + 33h
MOV DI, NUMB       ; adresa lui NUMB
MOV AL, 0          ; se şterge suma
ADD  AL, [DI]      ; adună [NUMB]
ADD AL, [DI + 1]   ; adună [NUMB + 1]
ADD word ptr [DI], -2     ; destinatie in memorie, sursa imediata
ADD byte ptr VAR, 5       ; fortarea instructiunii pe un octet, VAR fiind
                          ;declarat DW
```

Instrucţiunea INC (Increment addition)

Instrucţiunea INC are formatul general:

INC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biţi iar semnificaţia operaţiei este incrementarea valorii destinaţie cu 1. Toţi indicatorii de stare sunt afectaţi, cu excepţia lui CF (Carry Flag).

Exemplu:

```
MOV DI, NUMB       ; adresa lui NUMB
MOV AL, 0          ; şterge suma
ADD  AL, [DI]      ; adună [NUMB]
INC   DI           ; DI = DI + 1
ADD AL, [DI]       ; adună [NUMB + 1]
```

Instrucţiunea ADC (ADdition with Carry)

Instrucţiunea ADD are formatul general:

ADD <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată.

Instrucţiunea acţionează întocmai ca ADD, cu deosebirea că la rezultat este adăugat şi bitul CF. Este utilizat, de regulă, pentru a aduna numere mai mari de 16 biţi (8086-80286) sau mai mari de 32 de biţi la 80386, 80486, Pentium.

Exemplu:

Adunarea a două numere pe 32 de biţi se poate face astfel (BXAX) + (DXCX):

```
ADD  AX, CX
ADC  BX, DX
```

Instrucţiunea SUB (SUBstract)

Instrucţiunea SUB are formatul general:

SUB <destinatie> <sursa>

Unde <*destinatie*> poate fi un registru general sau o locaţie de memorie, iar <*sursa*> poate fi registru general, locaţie de memorie sau o valoare imediată. Rezultatul operaţiei este următorul: <destinatie> == <destinatie> - <sursa>. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biţi şi trebuie să aibă aceeaşi dimensiune. Scăderea poate fi văzută ca o adunare cu reprezentarea în complementul faţă de 2 al operandului sursă şi cu inversarea bitului CF, în sensul că, dacă la operaţie (adunarea echivalentă) apare transport, CF=0 şi dacă la adunarea echivalentă nu apare transport, CF=1.

Pentru instrucţiunile:

```
MOV CH, 22h
SUB CH, 34h
```

Rezultatul este **-12 (1110 1110)**, iar indicatorii de stare se modifică astfel:

ZF = 0 (rezultat diferit de zero)
CF = 1 (împrumut)
SF = 1 (rezultat negativ)
PF = 0 (paritate pară)
OF = 0 (fără depăşire)

Instrucţiunea DEC (DECrement substraction)

Instrucţiunea DEC are formatul general:

DEC <destinatie>

Unde <*destinatie*> este un registru sau un operand în memorie, pe 8 sau pe 16 biţi iar semnificaţia operaţiei este decrementarea valorii destinaţie cu 1. Toţi indicatorii de stare sunt afectaţi, cu excepţia lui CF (Carry Flag).

Instrucţiunea SBB (SuBstract with Borrow)

Instrucţiunea SBB are formatul general:

SBB <destinatie>, <sursa>

Unde <*destinatie*> şi <*sursa*> pot fi registru sau operand în memorie, pe 8 sau pe 16 biţi. Rezultatul operaţiei este următorul: <destinatie> == <destinatie> - <sursa> - CF, deci la fel ca şi în cazul instrucţiunii SUB, dar din rezultat se scade şi bitul CF. Indicatorii de stare modificaţi în urma acestei operaţii sunt: AF, CF, PF, SF, ZF, OF. Această instrucţiune este utilizată, de regulă, pentru a scădea numere mai mari de 16 biţi (la 8086 - 80286) sau de 32 de biţi (la 80386, 80486, Pentium).

Exemplu

Scăderea a două numere pe 32 de biţi se poate face astfel (BXAX) - (SIDI):

```
SUB   AX, DI
SBB   BX, SI
```

Exemple de programe

1. Program care citeşte un număr de la tastatură şi afişează dacă numărul este par sau nu:

```
; Programul citeste un numar si afiseaza un mesaj referitor la paritate
dosseg
.model small
.stack
.data

mesaj  db 13,10,'Introduceti numarul:(<=9)$'
mesg_par  db 13,10,'Numarul introdus este par!$'
mesg_impar  db 13,10,'Numarul introdus este impar!$'
```

```
.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          mov bx,2
          div bx ; se imparte AX la BX, catul va fi in AX, restul in DX
          cmp dx,0
          jnz impar
          mov ah,09
          mov dx,offset mesg_par
          int 21h
          jmp sfarsit
impar:    mov ah,09
          mov dx,offset mesg_impar
          int 21h
sfarsit:
          mov ah,4ch
          int 21h ; sfarsitul programului

END pstart
```

2. Program care calculează pătratul unui număr introdus de la tastatură.

```
; Programul calculeaza patratul unui numar (<=256) introdus de la tastatura
; Valoarea patratului se calculeaza in registrul AX (valoare maxima 2^16 =
65536)

dosseg
.model small
.stack
.data

nr DB 10,10 dup(0)
r DB 10, 10 dup(0)
```

```
mesaj  db 13,10,'Introduceti numarul:(<=256)$'
patrat db 13,10,'Patratul numarului este:$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov cl,nr[1]  ; incarc in CL numarul de cifre al numarului introdus
          inc cl        ; in sir se va merge pana la pozitia cl+1
          mov si,1      ; folosesc registrul SI pe post de contor
          xor ax,ax     ; initializez AX cu valoarea 0
          mov bl,10     ; se va inmulti cu valoarea 10 care este stocata in BL
inmultire:
          mul bl
          inc si
          mov dl,nr[si]
          sub dl,30h
          add ax,dx
          cmp si,cx
          jne inmultire

          mul ax

          xor si,si
          mov bx,10

cifra: ; aici incepe afisarea rezultatului din AX
          div bx
          add dl,30h
          mov r[si],dl
          inc si
          xor dx,dx
```

```
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset patrat
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter
        jmp sfarsit


        mov ah,9
        mov dx,offset patrat
        int 21h

sfarsit:
        mov ah,4ch
        int 21h ; stop program

END pstart
```

3. Program care calculează valoarea unui număr ridicat la o putere. Atât numărul cât şi exponentul (puterea) sunt introduse de la tastatură.

```
; Programul calculeaza un numar ridicat la o putere

; Observatie. Deoarece rezultatul se calculeaza in registrul AX care este un
; registru pe  16 biti, valoarea maxima calculata corect este 2^16= 65536

.model small
.stack
.data

mesaj1  db 13,10,'Introduceti numarul:(<=9)$'
mesaj2  db 13,10,'Introduceti puterea:(<=9)$'
mesaj_final  db 13,10,'Rezultatul este: $'
mesaj_putere_0 db 13,10, 'Orice numar ridicat la puterea 0 este 1! $'
```

```
r db 30 dup(0) ; in variabila r se va stoca rezultatul

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj1
        int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
          ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h ; se obtine valoarea numerica
                    ; scazandu-se codul lui 0 in ASCII (30H)
          push ax ; se salveaza valoarea lui ax in stiva

          mov ah,09
          mov dx,offset mesaj2
          int 21h
          mov ah,01h ; se citeste un caracter de la tastatura
                    ; codul ASCII al caracterului introdus va fi in AL
          int 21h
          and ax,00FFh
          sub ax, 30h   ; se obtine valoarea numerica
                        ; scazandu-se codul lui 0 in ASCII (30H)
          mov cx,ax     ; registrul CX contorizeaza numarul de inmultiri
          cmp cx,0
          jne putere_0
          mov ah,09
          mov dx, offset mesaj_putere_0
          int 21h
          jmp sfarsit

putere_0:
          pop bx          ;se salveaza in BX valoarea cu care inmulteste
          mov ax,0001
inmultire:
          mul bx
          loop inmultire
```

```
        xor si,si
        mov bx,10
cifra:
        div bx
        add dl,30h
        mov r[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx, offset mesaj_final
        int 21h

caracter:
        dec si
        mov ah,02    ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,r[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter



sfarsit:
        mov ah,4ch
        int 21h ; sfarsitul programului

END pstart
```

4. Program care verifică dacă un număr este palindrom (un număr se numeşte palindrom dacă scris de la dreapta la stânga sau invers are aceeaşi valoare).

```
; Programul verifica daca un numar sau sir de caractere este palindrom


dosseg
.model small
.stack
.data
```

```
nr DB 10,10 dup(0)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_nu db 13,10,'Numarul nu este palindrom!$'
mesaj_da db 13,10,'Numarul este palindrom!$'

.code

pstart:
        mov ax,@data
        mov ds,ax

        mov ah,09
          mov dx,offset mesaj
        int 21h

          mov ah,0ah
          mov dx,offset nr
          int 21h

          mov si,1
          mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
          and cx,00FFh

          mov ax,cx
          mov bl,2
          div bl          ; in AL este catul impartirii lui AX la 2
          and ax,00FFh
          inc ax
          inc cx
          mov di,cx

urmatorul_caracter:
          inc si          ; SI creste de la inceputul sirului spre mijloc
          mov bl,nr[di]
          cmp nr[si],bl
          jne nu_este
          dec di          ; DI scade de la sfarsitul sirului spre mijloc
          cmp si,ax       ; in sir se va merge pana la pozitia cl+1
          jne urmatorul_caracter
          mov ah,9
          mov dx,offset mesaj_da
          int 21h
```

```
        jmp sfarsit

nu_este:
        mov ah,9
        mov dx,offset mesaj_nu
        int 21h


sfarsit:
        mov ah,4ch
        int 21h ; stop program

END pstart
```

5. Program care calculează suma cifrelor unui număr introdus de la tastatură.

```
; Programul calculeaza suma cifrelor unui numar introdus de la tastatura


dosseg
.model small
.stack
.data

nr DB 10,10 dup(?)
rezultat DB 10,10 dup(?)

mesaj  db 13,10,'Introduceti numarul:$'
mesaj_suma db 13,10,'Suma cifrelor numarului este: $'

.code

pstart:
      mov ax,@data
      mov ds,ax

      mov ah,09               ; aici se afiseaza mesajul initial de introducere
      mov dx,offset mesaj  ; a numarului
      int 21h

        mov ah,0ah            ; functia 10(0ah) citeste un sir de caractere de la
                              ; tastatura intr-o variabila de memorie
        mov dx,offset nr
```

```
        int 21h

        mov si,1
        mov cl,nr[si] ; incarc in CL numarul de cifre al numarului introdus
        and cx,00FFh
        inc cx        ; CX stocheaza acum ultima pozitie din sirul de cifre
        xor ax,ax     ; stocam rezultatul in AX, pe care il initializam cu zero

urmatorul_caracter:
        inc si        ; SI creste de la inceputul sirului spre sfarsit
        add al,nr[si]

        sub al,30h    ; scadem codul ASCII al lui zero

        cmp si,cx     ; in sir se va merge pana la pozitia cl+1
        jne urmatorul_caracter

        xor si,si     ; SI este indicele din sirul care va contine rezultatul

cifra: ; aici incepe afisarea rezultatului din AX

        mov bx,0ah
        div bx
        add dl,30h
        mov rezultat[si],dl
        inc si
        xor dx,dx
        cmp ax,0
        jne cifra

        mov ah,9
        mov dx,offset mesaj_suma
        int 21h

caracter:
        dec si
        mov ah,02             ;apelarea functiei 02 pentru afisarea unui caracter
        mov dl,rezultat[si]   ;al carui cod ASCII este in DL
        int 21h
        cmp si,0
        jne caracter

        mov ah,4ch
```

```
        int 21h                 ; terminarea programului

END pstart
```