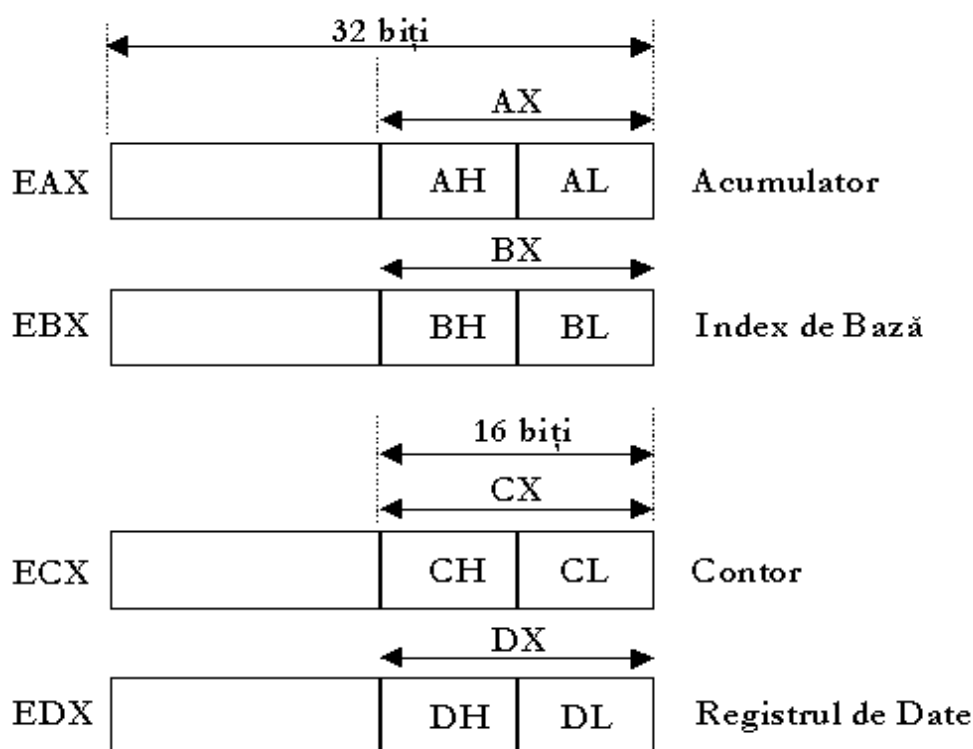


Introducere în limbajul de asamblare

Ne vom referi în cele ce urmează la familia de microprocesoare intitulată iAPx86 ce stau la baza calculatoarelor IBM PC, începând de la procesoarele 8088 și 8086, continuând cu 80286, 80386, 80486, Pentium, ș.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



Notă:

Regiștrii pe 32 de biți nu apar la 8086, 8088, 80286

Figura 1. Regiștrii de uz general – acumulator, index de bază, contor și de date

Regiștrii microprocesorului

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

categorii de regiștri: regiștrii de uz general, registrul indicatorilor de stare (*flags*), regiștrii de segment și registrul pointer de instrucțiune.

Regiștrii de uz general

Regiștrii de uz general (vezi figura 1 și figura 2) sunt implicați în operarea majorității instrucțiunilor, drept operanzi sursă sau destinație pentru calcule, copieri de date, pointeri la locații de memorie sau cu rol de contorizare. Fiecare dintre cei 8 regiștri de uz general AX, BX, CX, DX, SP, BP, DI, SI sunt regiștri pe 16 biți pentru microprocesorul 8086, iar de la procesorul 80386 încolo au devenit regiștri pe 32 de biți, denumiți, respectiv: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI (litera E provine de la *Extended – extins* în engleză). Mai mult, cei mai puțin semnificativi 8 biți ai regiștrilor AX, BX, CX, DX formează respectiv regiștrii AL, BL, CL, DL (litera L provine de la *Low – jos* în engleză), iar cei mai semnificativi 8 biți ai aceluiași regiștri formează regiștrii AH, BH, CH, DH (litera H provine de la *High – înalt* în engleză) (figura 1).

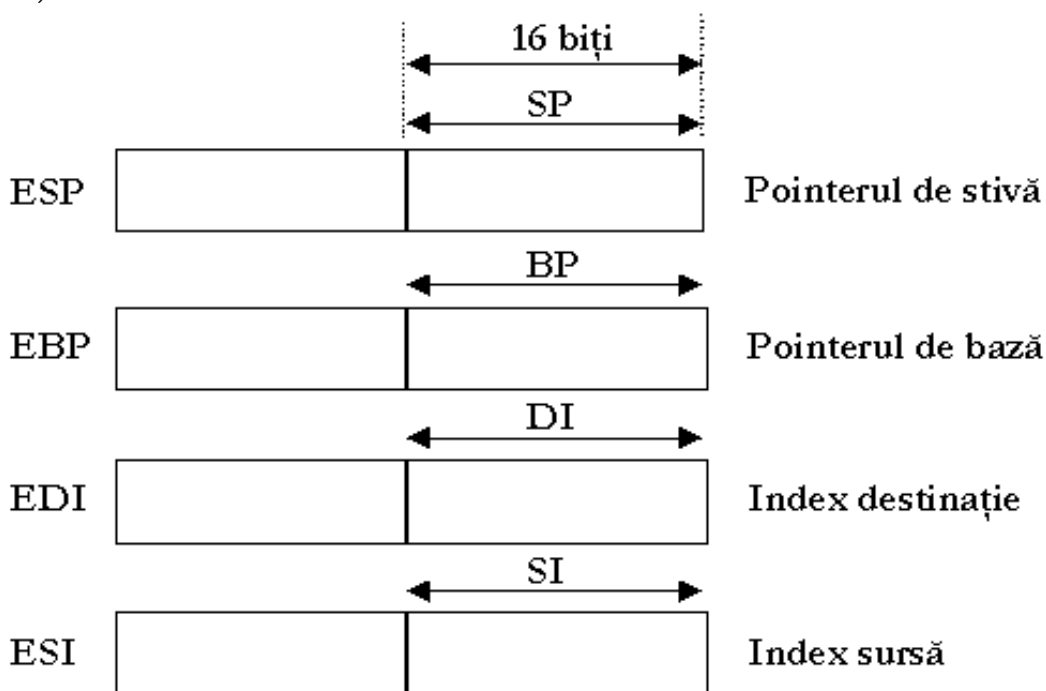


Figura 2. Regiștrii de uz general index și pointer

Ne vom concentra în continuare atenția asupra regiștrilor generali pe 16 biți; fiecare dintre aceștia poate stoca o valoare pe 16 biți, poate fi folosit pentru stocarea unei valori din memorie sau poate fi utilizat pentru operații aritmetice și logice. Spre exemplu, următoarele instrucțiuni:

```
...
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 regiștrii de uz general în locul regiștrilor BX și DX. În afara proprietății de a stoca valori și de a folosi drept operanzi sursă sau destinație pentru instrucțiunile de manipulare a datelor, fiecare dintre cei 8 regiștri de uz general au propria “personalitate”. Vom vedea în continuare care sunt caracteristicile specifice fiecăruia dintre regiștrii de uz general.

Registrul AX (EAX)

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

Instrucțiunea: **MOV AX, 1234H** încarcă valoarea 1234H (4660 în zecimal) în registrul acumulator AX. După cum spuneam, cei mai puțini semnificativi 8 biți ai registrului AX sunt identificați de AL (A-Low) iar cei mai semnificativi 8 biți ai aceluiași registru sunt identificați ca fiind AH (A-High). Acest lucru este utilizat pentru a lucra cu date pe un octet, permițând ca registrul AX să fie folosit pe postul a doi regiștri separați (AH și AL). Aceeași regulă este valabilă și pentru regiștrii de uz general BX, CX, DX. Următoarele trei instrucțiuni setează registrul AH cu valoarea 1, incrementează cu 1 această valoare și 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 stocați în memorie. O valoare reprezentată pe 16 biți stocată în registrul BX poate fi utilizată ca fiind o porțiune din adresa unei locații de memorie ce va fi accesată. Spre exemplu, următoarele instrucțiuni 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 locația de memorie referită de registrul BX. Acest lucru este datorat segmentării memoriei (segmentare discutată mai în detaliu în secțiunea consacrată regiștrilor 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 conținută în registrul DS).

Registrul CX (ECX)

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

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

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

```

MOV CX, 5
start:
...
<instrucțiuni ce se vor executa de 5 ori>
...
LOOP start

```

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

Registrul DX (EDX)

Registrul de uz general DX (Data register), denumit și registrul de date, poate fi folosit în cazul transferurilor de date Intrare/Ieșire sau atunci când are loc o operație de înmulțire sau de împărțire. Instrucțiunea **IN AL, DX** copiază o valoare de tip Byte dintr-un port de intrare, a cărei adresă se află în registrul DX. Următoarele instrucțiuni determină scrierea valorii 101 în portul I/O 1002:

```

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

```

Referitor la operațiile de înmulțire și împărțire, atunci când împărțim un număr pe 32 de biți la un număr pe 16 biți, cei mai semnificativi 16 biți ai deîmpărțitului trebuie să fie în DX. După împărțire, restul împărțirii se va afla în DX. Cei mai puțin semnificativi 16 biți ai deîmpărțitului trebuie să fie în AX iar câtul împărțirii va fi în AX. La înmulțire, atunci când se înmulțesc două numere pe 16 biți, cei mai semnificativi 16 biți ai produsului vor fi stocați în DX iar cei mai puțin semnificativi 16 biți în registrul AX.

Registrul SI

Registrul SI (Source Index) poate fi folosit, ca și BX, pentru a referi adrese de memorie. De exemplu, secvența de instrucțiuni următoare:

```

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

```

Încarcă valoarea (pe 8 biți) din memorie de la adresa 33 în registrul AL. Registrul SI este, de asemenea, foarte folositor atunci când este utilizat în legătură cu instrucțiunile dedicate tipului string (șir de caractere). Secvența 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 secvențial o serie de locații de memorie, cum ar fi șirurile de caractere. Instrucțiunile de tip string se pot repeta de mai multe ori, astfel încât o singură instrucțiune poate avea ca efect sute sau mii de operații.

Registrul DI

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

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

se adună la registrul BL valoarea pe 8 biți stocată la adresa 1000. Registrul DI este puțin diferit față de registrul SI în cazul instrucțiunilor de tip string; dacă SI este întotdeauna pe post de pointer sursă de memorie, registrul DI servește drept pointer destinație de memorie. Mai mult, în cazul instrucțiunilor de tip string, registrul SI adresează memoria relativ la registrul de segment DS, în timp ce DI conține referiri la memorie relativ la registrul de segment ES. În cazul în care SI și DI sunt utilizați cu alte instrucțiuni, ei fac referire la registrul de segment DS.

Registrul BP

Pentru a înțelege mai bine rolul regiștrilor BP și SP, a sosit momentul să spunem câteva lucruri despre porțiunea de memorie denumită stivă (în engleză *stack*). Stiva (vezi figura 3) reprezintă o porțiune specială de locații adiacente din memorie. Aceasta este conținută în cadrul unui segment de memorie și identificată de un selector de segment memorat în registrul SS (cu excepția cazului în care se folosește modelul nesegmentat de memorie în care stiva poate fi localizată oriunde în spațiul de adrese liniare al programului). Stiva este o porțiune a memoriei unde valorile pot

fi stocate și 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 instrucțiuni folosite sunt CALL și RET).

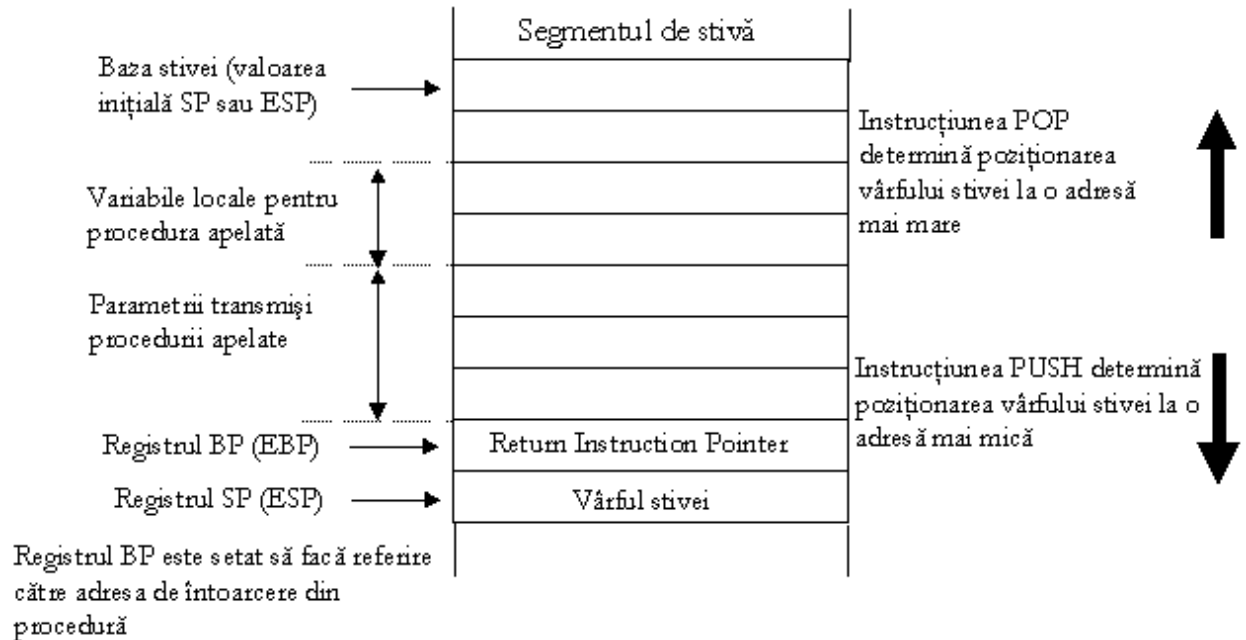


Figura 3. Structura stivei

Registrul pointer de bază, BP (Base Pointer) poate fi utilizat ca pointer de memorie precum regiștrii BX, SI și DI. Diferența este aceea că, dacă BX, SI și DI sunt utilizați 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 obișnuit în limbajele de nivel înalt, C sau Pascal, spre exemplu). Dacă stiva se află în porțiunea 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 și DI se referă la segmentul de date, nu există o modalitate eficientă de a folosi regiștrii BX, SI, DI pentru a face referire la parametrii salvați 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, instrucțiunile:

```
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 și alte necesități legate de accesul la porțiunea de stivă din memorie.

Registrul SP

Registrul SP (Stack Pointer), sau pointerul de stivă, reține de regulă adresa de deplasament a următorului element disponibil în cadrul segmentului de stivă. Acest registru este, probabil, cel mai puțin „general” dintre regiștrii 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 locației de memorie în care va fi introdus următorul element în stivă. Acțiunea de a introduce un nou element în stivă se numește „împingere” (în engleză *push*); de aceea, instrucțiunea respectivă poartă numele de PUSH. În mod asemănător, operația de scoatere a unui element din vârful stivei poartă, în engleză, numele de *pop*, iar instrucțiunea echivalentă operației se numește POP. În figurile 3 și 4 sunt ilustrate modificările survenite în conținutul stivei și al regiștrilor SP, BX și CX ca urmare a execuției instrucțiunilor următoare (se presupune că registrul SP are inițial valoarea 1000):

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

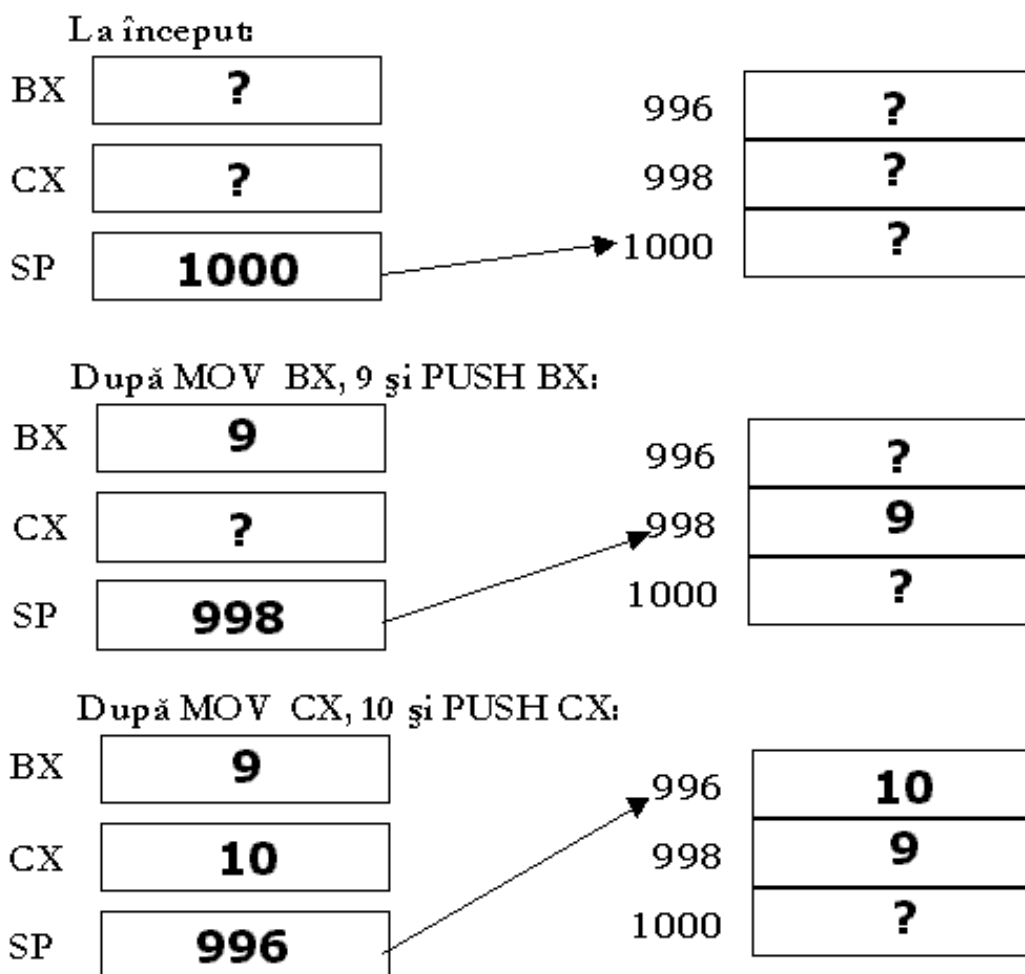



Figura 3. Modalitatea de funcționare a stivei după execuția primelor 4 instrucțiuni

Este permisă stocarea valorilor în registrul SP precum și modificarea valorii sale prin adunare sau scădere la fel ca și în cazul celorlalți regiștri de uz general; totuși, 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 instrucțiunile PUSH și POP nu reprezintă unicele modalități de utilizare a stivei. Indiferent dacă apelăm o subrutină sau ne întoarcem dintr-un astfel de apel de subrutină, fie procedură sau funcție, 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 locații de memorie nu vor mai fi cele corecte. În concluzie, registrul SP nu trebuie modificat în mod direct; el este modificat automat în urma instrucțiunilor POP, PUSH, CALL, RET. Oricare dintre ceilalți regiștri de uz general pot fi modificați în mod direct în orice moment.

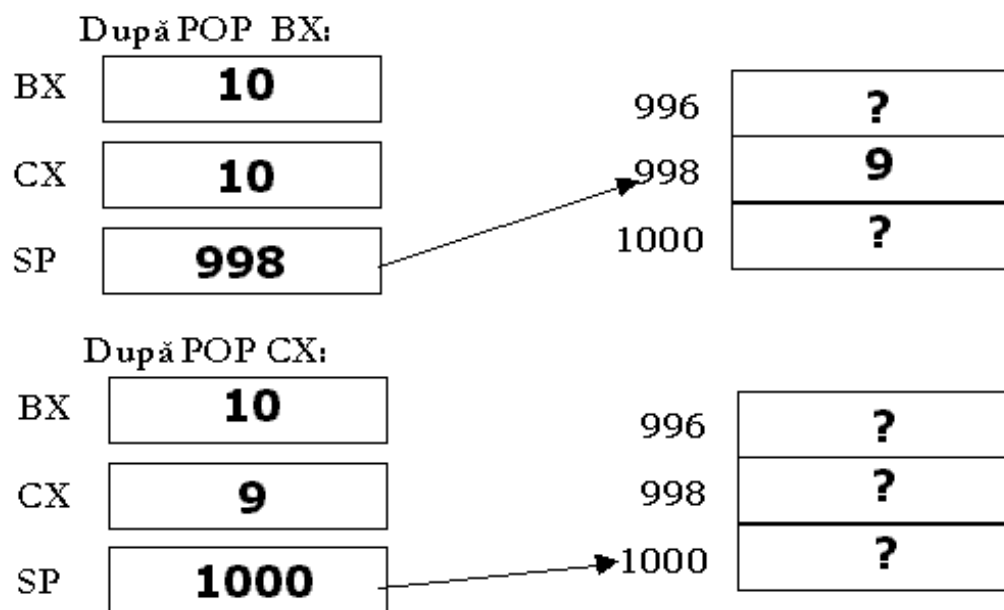


Figura 4. Funcționarea stivei după ultimile două instrucțiuni POP

Registrul pointer de instrucțiuni (IP)

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

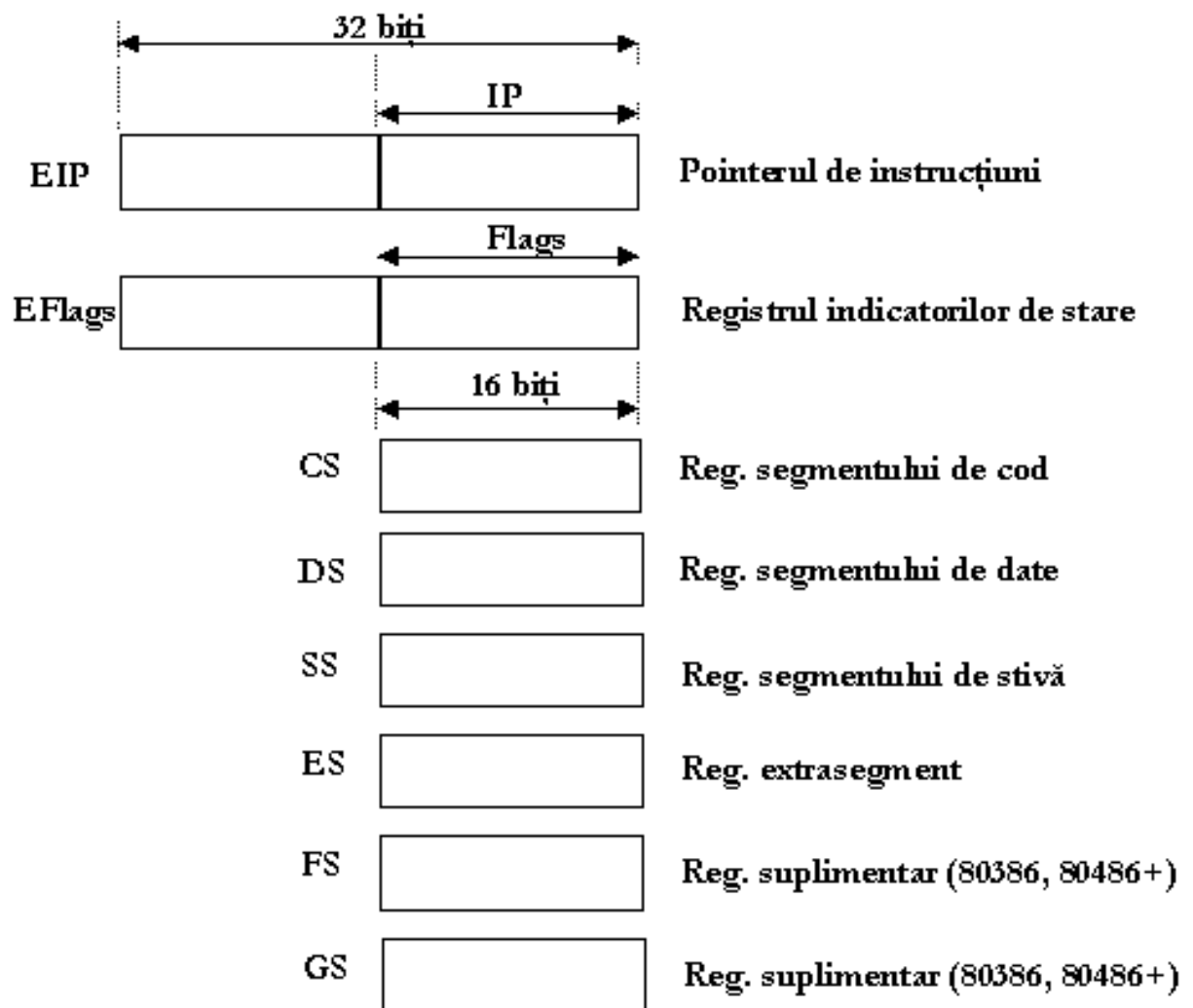
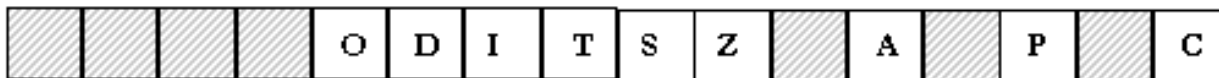


Figura 5. Regiștii de segment, pointerul de instrucțiuni și registrul indicatorilor de stare

Registrul indicatorilor de stare (FLAGS)

Registrul indicatorilor de stare - FLAGS



O - Overflow Flag

D - Direction Flag

I - Interrupt Flag

T - Trap Flag

S - Sign Flag

Z - Zero Flag

A - Auxiliary Carry Flag

P - Parity Flag

C - Carry Flag

Figura 6. Registrul indicatorilor de stare - detaliu

Registrul indicatorilor de stare (FLAGS) pe 16 biți conține informații legate de starea microprocesorului precum și de rezultatele ultimilor instrucțiuni executate. Un indicator de stare (*flag*) este în sine o locație de memorie de 1 bit ce indică starea curentă a microprocesorului și modalitatea sa de operare. Un indicator se spune că “este setat” dacă are valoarea 1 și “nu este setat” în caz contrar. Indicatorii de stare se modifică după execuția unor instrucțiuni aritmetice sau logice. Exemple de indicatori de stare (vezi figura 6):

- C (Carry) indică apariția unei cifre binare de transport în cazul unei adunări sau împrumut în cazul unei scăderi;
- O (Overflow) apare în urma unei operații aritmetice. Dacă este setat, înseamnă că rezultatul nu încapă în operandul destinație;
- Z (Zero) indică faptul că rezultatul unei operații aritmetice sau logice este zero;
- S (Sign) indică semnul rezultatului unei operații aritmetice;
- D (Direction) – când este zero, procesarea elementelor șirului 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 execuției programului pas cu pas. Dacă este setat, UCP

întrerupe fiecare instrucțiune, lăsând programul depanator să execute 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 setat 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.

Regiștrii 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 ($16 \cdot 64 \text{ KB} = 1024 \text{ KB} = 1 \text{ MB}$).

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).

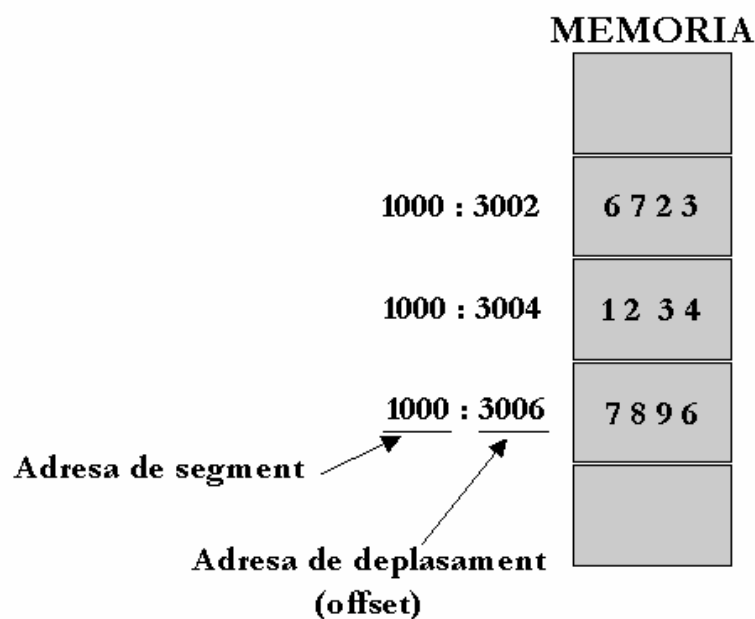


Figura 7. Cele două porțiuni ale unei adrese segmentate

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

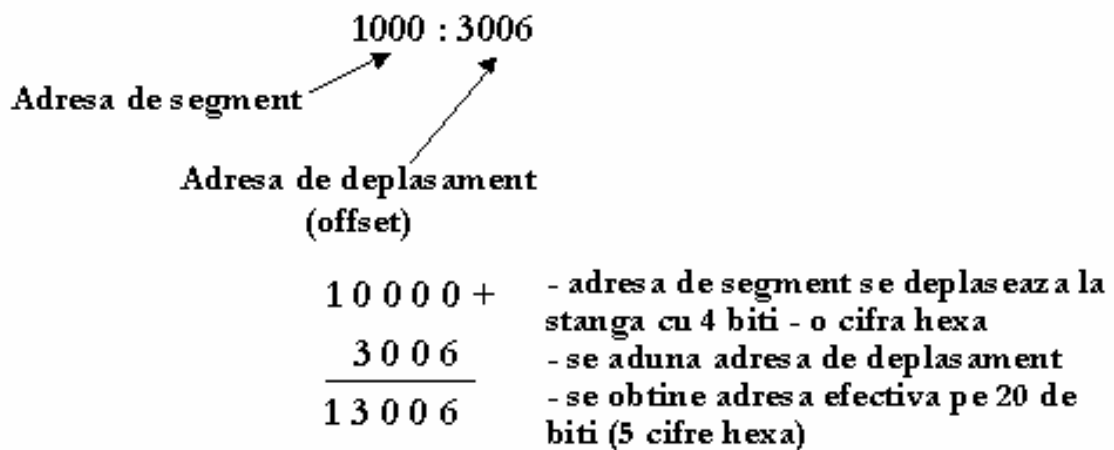


Figura 8. Exemplu de calcul al adresei efective

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 instrucțiune pentru execuție decât cea definită de CS. Registrul CS poate fi modificat de un număr de instrucțiuni, precum instrucțiuni 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 mulțimea de date cu care lucrează programul aflat în execuție.

Registrul ES – face referire la începutul blocului de 64KB cunoscut sub denumirea de extra-segment. Acesta nu este dedicat nici unui scop anume, fiind disponibil pentru diverse acțiuni. Uneori acesta poate fi folosit pentru crearea unui bloc de memorie de 64 KB adițional pentru date. Acest extra-segment lucrează foarte bine în cazul instrucțiunilor de tip STRING. Toate instrucțiunile 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 instrucțiunile ce folosesc implicit registrul SP (instrucțiunile 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 instrucțiuni în limbaj de asamblare

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

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

unde:

- **<nume>** - reprezintă un nume simbolic opțional;
- **<instrucțiune/directiva>** - reprezintă mnemonica (numele) unei instrucțiuni sau a unei directive;
- **<operanzi>** - reprezintă o combinație de unul, doi sau mai mulți operanzi (sau chiar nici unul), care pot fi constante, referințe de memorie, referințe de regiștri, șiruri de caractere, în funcție de structura particulară a instrucțiunii;
- **<;comentariu>** - reprezintă un comentariu opțional ce poate fi plasat după caracterul „;” până la sfârșitul liniei respective de cod.

Nume de variabile și etichete

Numele folosite într-un program scris în limbaj de asamblare pot identifica variabile numerice, variabile șir de caractere, locații de memorie sau etichete. Spre exemplu, următoarea secvență de cod, care calculează valoarea lui trei factorial ($3! = 1 \times 2 \times 3 = 6$) cuprinde câteva nume de variabile și 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
Ciclar:
MOV ax, [Valoare_Factorial]
MUL [Factorial]
MOV [Valoare_Factorial], ax
INC [Factorial]
LOOP Ciclar
```

```
RET
Trei_Factorial ENDP
END
```

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

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

...
următoarea instrucțiune care va fi executată după instrucțiunea **JMP scadere** va fi instrucțiunea **SUB AX, CX**. Exemplul anterior este echivalent cu secvența:

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

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

Numele variabilelor sau etichetelor folosite într-un program nu trebuie să se confunde cu numele *rezervate* de asamblor, cum ar fi numele de directive și instrucțiuni, numele regiștrilor, etc. De exemplu, o declarație 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 instrucțiuni sau apare pe o linie cu instrucțiuni trebuie să aibă semnul „:” după numele ei. Totodată, 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 funcționalității cu primul, eticheta a fost denumită x5, absolut nesugestiv!

Observație:

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

Directive de segment simplificate

Datorită faptului că regiștrii microprocesorului 8086 sunt regiștri pe 16 biți, s-a impus folosirea unor segmente de memorie de câte 64Ko (maxim cât se poate adresa având la dispoziție 16 biți - $64Ko=2^{16}=65536$). Într-un program scris în limbaj de asamblare (vom folosi în continuare prescurtarea ASM) există trei segmente: segmentul de cod, segmentul de date și 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 și 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 și de date.

De exemplu, **.STACK 200H** definește o stivă de 512 octeți (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 locațiilor 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 locație de memorie dar nu poate fi încărcat direct cu o constantă, registrul de segment DS este încărcat în general printr-o secvență de 2 instrucțiuni:

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

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

Secvența 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 afișează textul memorat în `DataSource` pe ecran:

```
;Program p01.asm
.MODEL small          ;se specifică modelul de memorie SMALL
.STACK 200H          ;se definește o stivă de 512 octeți
.DATA                ;se specifică începutul segmentului de
                    ;date
DataSource DB 'Hello!$' ;se definește variabila
                    ;DataSource, inițializată cu valoarea
                    ;"Hello!"
.CODE                ;începutul segmentului de cod al
                    ;programului
ProgramStart:       ;orice program are o etichetă de
                    ;început
mov bx,@data        ;secvența ce setează registrul DS să
                    ;facă referire la segmentul de date ce
                    ;începe cu .DATA

mov ds,bx
mov dx, OFFSET DataSource ;se încarcă în DX adresa
;variabilei DataSource
mov ah,09           ;codul funcției DOS de afișare a unui
                    ;string
int 21H            ;apelul DOS de afișare a string-ului
mov ah, 4cH        ;codul funcției DOS de terminare a
                    ;programului
int 21H            ;apelul DOS de terminare a programului
END ProgramStart   ;directiva de terminare a codului
                    ;programului
```

Explicații:

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

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

3. Fără cele două instrucțiuni care setează registrul DS către segmentul definit de `.DATA`, funcția de afișare a string-ului nu ar fi funcționat cum trebuie. Variabila `DataSource` se află în segmentul `.DATA` și nu poate fi accesată dacă DS nu este poziționat către acest segment. Acest lucru se explică în modul următor: atunci când facem apelul DOS de afișare 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` și DX cu adresa (offset-ul) lui `DataSource` avem o referință completă `segment:offset` către `DataSource`.

Observații.

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 execuției primei instrucțiuni din program, procesorul nu ar ști unde să găsească instrucțiunea și programul nu ar rula niciodată. În mod asemănător, registrul de segment SS este setat de DOS înainte de execuția programului și de regulă rămâne nemodificat pe perioada execuției programului.

Cu registrul de segment DS lucrurile stau altfel. În timp ce registrul CS se referă la instrucțiuni (cod), SS se referă ("poințează") la stivă, DS "poințează" la date. Programele nu manipulează direct instrucțiuni 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 și 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 afișa 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
                               ;inițializată cu valoarea "B"

.CODE
ProgramStart:
mov dx, @data
mov es, dx                   ;spre deosebire de programul anterior,
                               se folosește 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 funcției DOS de afișare a
                               ;unui caracter
int 21H                      ;apelul DOS de execuție a afișării
mov ah, 4cH                  ;codul funcției 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 convențiilor Microsoft de adresare a segmentelor.

Directiva **.MODEL**

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

Definiții: "near" înseamnă adresa (offset-ul) pe 16 biți din cadrul aceluiași 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 și datele programului încap în același segment de 64Ko. Atât codul cât și datele sunt de tip near.

- **small** - codul programului trebuie să fie într-un singur segment de 64Ko și datele într-un bloc separat de 64Ko; codul și 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 și datele pot depăși 64Ko, dar nici un masiv de date nu poate depăși 64 Ko. Atât codul cât și datele sunt far.

- **huge** - atât codul cât și datele pot depăși 64Ko și masivele de date pot depăși 64 Ko. Atât codul cât și datele sunt far. Pointerii la elementele dintr-un masiv sunt far.

În continuare sunt prezentate câteva exemple legate de modalitățile de declarare a variabilelor și 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!

mesajsc02 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,mesajsc02      ; BX este pointer la un string
                       ; (cod C echivalent: BX=&mesajsc02)

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

mov si, 1             ; Foloseste SI cu rol de index
inc byte [mesajsc02+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 și 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 și, drept urmare, referirile la OFFSET sau la valoare sunt deseori confundate. În figura 9 sunt ilustrate conceptele de adresă de deplasament (offset) și valoare stocată în memorie.

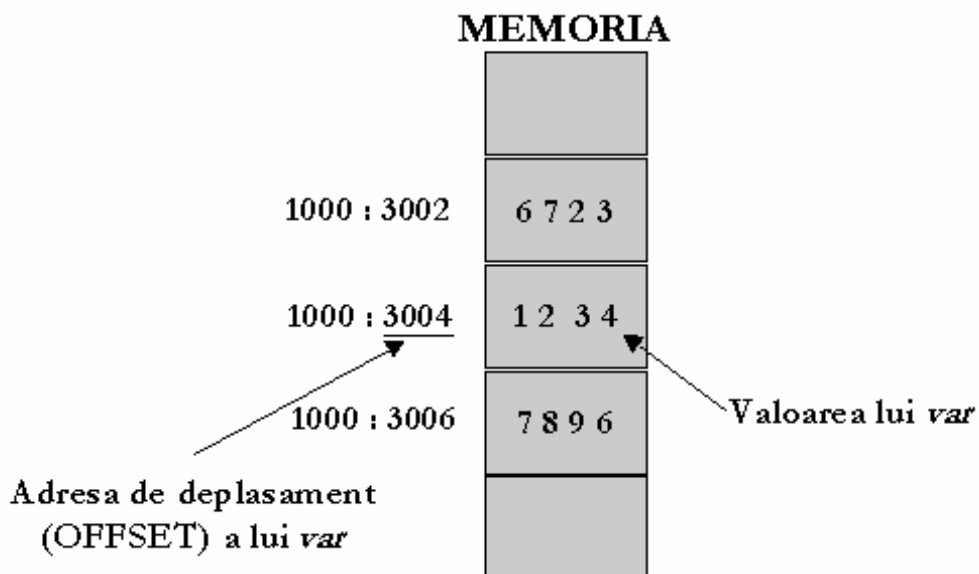


Figura 9. Ilustrarea noțiunilor de adresă de deplasament și valoare stocată în memorie

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

```
MOV BX, OFFSET var
```

Încarcă valoarea 5004H în registrul BX. Valoarea 5004H nu se modifică; ea este construită în cadrul instrucțiunii. 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* și apoi folosirea registrului respectiv pentru adresarea memoriei. Instrucțiunile:

```
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, instrucțiunile:

```
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 și care se află memorat la adresa (de memorie) respectivă. Instrucțiunile:

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

Modifică valoarea lui *var* la 3, dar instrucțiunea:

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. Totuși, în momentul execuției 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 („[” și „]”), eliminând astfel ambiguitatea.

Instrucțiuni ale microprocesorului Intel

Microprocesoarele din familia Intel x86 dispun de o serie impresionantă de instrucțiuni, asemeni tuturor procesoarelor din clasa procesoarelor CISC (Complex Instruction Set Computer). Instrucțiunile se pot împărți în: instrucțiuni logice, aritmetice, de transfer și de control. Prezintă în continuare câteva exemple din fiecare clasă de instrucțiuni.

Instrucțiuni logice

Instrucțiunile logice implementează funcțiile logice de bază, pe un octet sau pe cuvânt. Ele acționează bit cu bit, deci se aplică funcția logică respectivă tuturor biților sau perechilor de biți corespunzători operanzilor. Instrucțiunile logice sunt următoarele:

- **NOT:** $A = \sim A$
- **AND:** $A \&= B$
- **OR:** $A |= B$
- **XOR:** $A ^= B$
- **TEST:** $A \& B$

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

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

Instrucțiunea NOT

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

NOT *destinație*

Unde *destinație* este fie un registru, fie o locație de memorie pe 8 sau 16 biți. Instrucțiunea are ca efect inversarea (negarea) tuturor biților operandului, adică aducerea în forma codului invers - complement față de 1.

Instrucțiunea AND

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

AND *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are ca efect operația: $\langle \text{destinatie} \rangle == \langle \text{destinatie} \rangle \text{ AND } \langle \text{sursa} \rangle$. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea TEST (AND “non-distructiv”)

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

TEST *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are același efect ca și instrucțiunea AND, cu deosebirea că nu se modifică operandul destinație, iar indicatorii de stare sunt modificați în același mod ca și în cazul instrucțiunii AND.

Instrucțiunea OR

Este o instrucțiune cu doi operanzi, cu forma generală:

OR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: $\langle \text{destinatie} \rangle == \langle \text{destinatie} \rangle \text{ OR } \langle \text{sursa} \rangle$. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit.

Instrucțiunea XOR (SAU-Exclusiv)

Este o instrucțiune cu doi operanzi, cu forma generală:

XOR *destinatie, sursa*

Unde *destinatie* este fie un registru, fie o locație de memorie pe 8 sau 16 biți, iar *sursa* poate fi registru, locație de memorie sau o constantă pe 8 sau 16 biți. Instrucțiunea are efectul: $\langle \text{destinatie} \rangle == \langle \text{destinatie} \rangle \text{ XOR } \langle \text{sursa} \rangle$. Indicatorii de stare modificați sunt: SF, ZF, PF, CF, OF = 0, AF nedefinit. Funcția XOR, denumită SAU-Exclusiv (sau *anti-coincidență*) are valoarea logică 1 atunci când operandii săi sunt diferiți (unul are valoarea 0 iar celălalt valoarea 1) și valoarea logică 0 când ambii operanzi au aceeași valoare (fie ambii au valoarea 0, fie ambii au valoarea 1).

Observație:

De cele mai multe ori, instrucțiunile AND și OR sunt folosite pe post de „mascare” a datelor; în acest sens, o valoare de tip „mască” (*mask*) este utilizată pentru a forța anumiți biți să ia valoarea zero sau valoarea 1 în cadrul altei valori. O

astfel de „mască” logică are efect asupra anumitor biți, în timp ce pe alții îi lasă neschimbați. Exemple:

- Instrucțiunea **AND CL, 0Fh** – face ca cei mai semnificativi 4 biți să ia valoarea 0, în timp ce biții mai puțin semnificativi sunt lăsați neschimbați; astfel, dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **AND CL, 0Fh** va avea valoarea **0000 1101**.
- Instrucțiunea **OR CL, 0Fh** – face ca cei mai puțin semnificativi 4 biți să ia valoarea 1, în timp ce biții mai semnificativi să rămână nemodificați. Dacă registrul CL are valoarea inițială **1001 1101**, după execuția instrucțiunii **OR CL, 0Fh** va avea valoarea **1001 1111**.

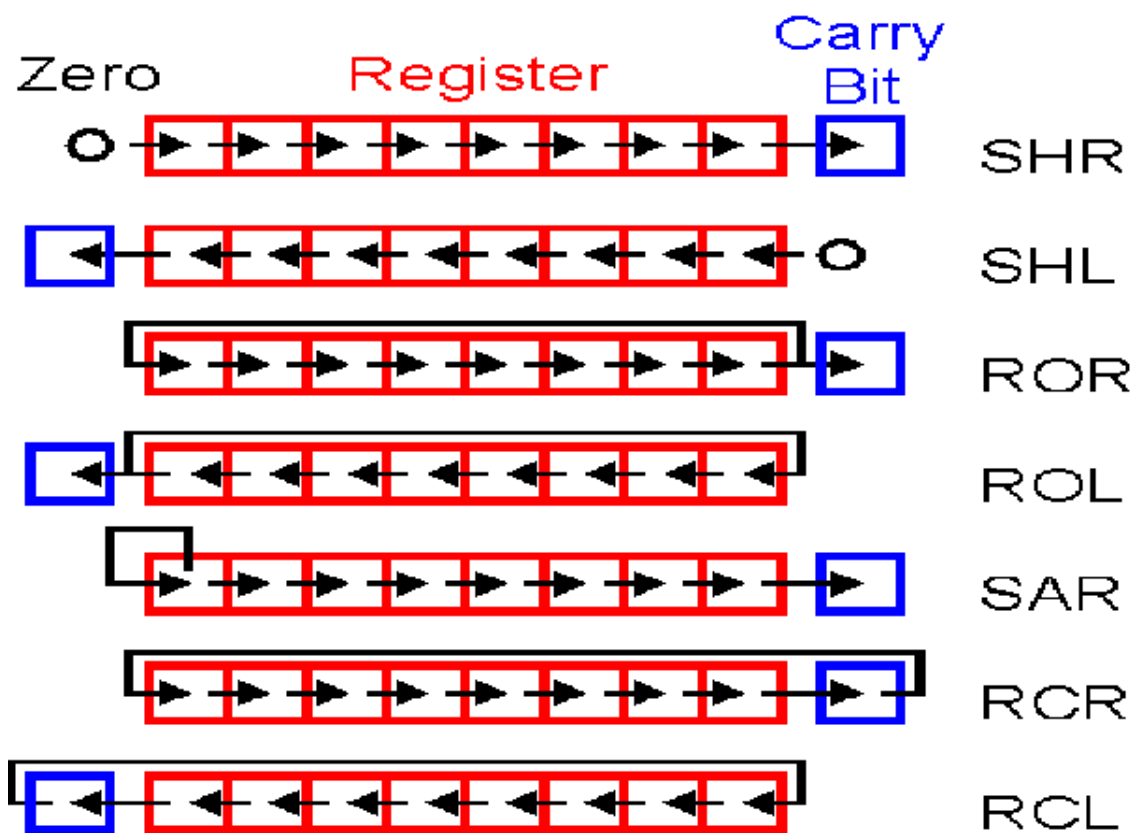


Figura 10. Instrucțiuni de deplasare și de rotație

Instrucțiuni de deplasare și de rotație

Acest tip de instrucțiuni (vezi figura 10) permit realizarea operațiilor de deplasare și de rotație la nivel de bit. Ele au doi operanzi, primul operand fiind cel asupra căruia se aplică operația de deplasare pe biți, iar cele de-al doilea (operandul *numărător* sau *contor*) semnifică numărul de biți cu care se face această deplasare. Operațiile se pot face de la dreapta spre stânga sau invers. Deplasarea înseamnă

translatarea tuturor biților din operand la stânga/dreapta, cu completarea unei valori fixe în poziția rămasă liberă și cu pierderea biților din dreapta/stânga. Rotația presupune translatarea biților din operand la stânga/dreapta, cu completarea în dreapta/stânga cu biții care se pierd în partea opusă. Sintaxa generală a instrucțiunilor de deplasare și rotație este următoarea:

INSTR <operand> , <contor>

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

Observație.

Totdeauna există două modalități 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

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

Această instrucțiune translatează biții operandului o poziție la stânga de câte ori specifică operandul numărător. Pozițiile rămase libere prin deplasarea la stânga sunt umplute cu zerouri la bitul cel mai puțin semnificativ, în timp ce bitul cel mai semnificativ se deplasează în indicatorul CF (Carry Flag).

Reprezintă o modalitate rapidă de înmulțire cu o putere a lui 2 (în funcție de numărul de biți pentru care se face deplasarea la stânga).

Exemple:

1. Înmulțirea lui AX cu 10 (1010 în binar) (înmulțim cu 2 și 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. Înmulțirea lui AX cu 18 (10010 în binar) (înmulțim cu 2 și 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

Instrucțiunea SHR (Shift Right)

Această instrucțiune translatează biții operandului o poziție la dreapta de câte ori specifică operandul numărător. Bitul cel mai puțin semnificativ se deplasează în indicatorul CF (Carry Flag).

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

Instrucțiunea SAR (Shift Arithmetic Right)

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

Reprezintă o modalitate rapidă de împărțire *cu semn* la o puterea a lui 2 (în funcție de numărul de biți cu care se face deplasarea la dreapta).

Instrucțiunea RCL (Rotate through Carry Left)

Această instrucțiune determină o rotație a biților operandului către stânga prin intermediul lui CF (Carry Flag). Astfel, cel mai semnificativ bit trece din operand în CF, apoi se deplasează toți biții din operand cu o poziție la stânga iar CF original trece în bitul cel mai puțin semnificativ din operand.

Instrucțiunea ROL (Rotate Left)

Această instrucțiune determină o rotație a biților operandului către stânga. Astfel, cel mai semnificativ bit trece din operand în bitul cel mai puțin semnificativ.

Exemplu:

După execuția instrucțiunilor:

ROL AX, 6

AND AX, 1Fh

Biții 10-14 din AX se mută în biții 0-4.

Instrucțiunea RCR (Rotate through Carry Right)

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

Instrucțiunea ROR (Rotate Right)

Această instrucțiune determină o rotație a biților operandului către dreapta. Bitul cel mai puțin semnificativ trece în bitul cel mai semnificativ.

Exemple:

```
MOV ax,3      ; Valori inițiale          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      ; rotație stg (LSB=MSB)     AX = 1000 0000 0111 1010
ROL ax,1      ; rotație dr (MSB=LSB)     AX = 0000 0000 1111 0101
MOV cl,3      ; folosim CL pt depl cu 3 biți CL = 0000 0011
SHR ax,cl     ; împărțim AX la 8          AX = 0000 0000 0001 1110
MOV cl,3      ; folosim CL pt depl cu 3 biți CL = 0000 0011
SHL bx,cl     ; înmulțim BX cu 8          BX = 0000 0000 0010 1000
```

Instrucțiuni aritmetice

Instrucțiunea ADD (ADDition)

Instrucțiunea ADD are formatul general:

ADD <destinație> <sursa>

Unde <destinație> poate fi un registru general sau o locație de memorie, iar <sursa> poate fi registru general, locație de memorie sau o valoare imediată. Cei doi operanzi nu pot fi însă în același timp locații de memorie. Rezultatul operației este

următorul: <destinație> == <destinație> + <sursa>. Indicatorii de stare modificați în urma acestei operații sunt: AF, CF, PF, SF, ZF, OF. Operanții pot fi pe 8 sau pe 16 biți și trebuie să aibă aceeași dimensiune. Dacă apare ambiguitate la modul de exprimare al operanzilor (8 sau 16 biți) se va folosi operatorul PTR.

Exemple:

```
ADD AX, BX      ; adunare între regiștri – AX ← AX + BX
ADD DL, 33h     ; adunare efectivă - DL ← DL + 33h
MOV DI, NUMB    ; adresa lui NUMB
MOV AL, 0       ; se șterge suma
ADD AL, [DI]    ; adună [NUMB]
ADD AL, [DI + 1] ; adună [NUMB + 1]
ADD word ptr [DI], -2 ; destinație în memorie, sursa imediată
ADD byte ptr VAR, 5 ; fortarea instrucțiunii pe un octet, VAR fiind
                   ; declarat DW
```

Instrucțiunea INC (Increment addition)

Instrucțiunea INC are formatul general:

INC <destinație>

Unde <destinație> este un registru sau un operand în memorie, pe 8 sau pe 16 biți iar semnificația operației este incrementarea valorii destinație cu 1. Toți indicatorii de stare sunt afectați, cu excepția lui CF (Carry Flag).

Exemplu:

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

Instrucțiunea ADC (ADdition with Carry)

Instrucțiunea ADD are formatul general:

ADD <destinație> <sursa>

Unde <destinație> poate fi un registru general sau o locație de memorie, iar <sursa> poate fi registru general, locație de memorie sau o valoare imediată.

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

Exemplu:

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

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

Instrucțiunea SUB (SUBstract)

Instrucțiunea SUB are formatul general:

SUB <destinatie> <sursa>

Unde <destinatie> poate fi un registru general sau o locație de memorie, iar <sursa> poate fi registru general, locație de memorie sau o valoare imediată. Rezultatul operației este următorul: <destinatie> == <destinatie> - <sursa>. Indicatorii de stare modificați în urma acestei operații sunt: AF, CF, PF, SF, ZF, OF. Operanzii pot fi pe 8 sau pe 16 biți și trebuie să aibă aceeași dimensiune. Scăderea poate fi văzută ca o adunare cu reprezentarea în complementul față de 2 al operandului sursă și cu inversarea bitului CF, în sensul că, dacă la operație (adunarea echivalentă) apare transport, CF=0 și dacă la adunarea echivalentă nu apare transport, CF=1.

Pentru instrucțiunile:

```
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 depășire)

Instrucțiunea DEC (DECrement subtraction)

Instrucțiunea DEC are formatul general:

DEC <destinatie>

Unde <destinatie> este un registru sau un operand în memorie, pe 8 sau pe 16 biți iar semnificația operației este decrementarea valorii destinație cu 1. Toți indicatorii de stare sunt afectați, cu excepția lui CF (Carry Flag).

Instrucțiunea SBB (SuBstract with Borrow)

Instrucțiunea SBB are formatul general:

SBB <destinatie>, <sursa>

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

Exemplu

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

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

Exemple de programe

1. Program care citește un număr de la tastatură și afișează 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 citește un caracter de la tastatură
    ; codul ASCII al caracterului introdus va fi în AL
    int 21h
    mov bx,2
    div bx ; se împarte AX la BX, câtul va fi în AX, restul în 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 ; sfârșitul programului

END pstart

```

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

; Programul calculează pătratul unui număr (≤ 256) introdus de la tastatură
; Valoarea pătratului se calculează în registrul AX (valoare maximă $2^{16} = 65536$)

```

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 și 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 numește palindrom dacă scris de la dreapta la stânga sau invers are aceeași 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