

# Capitolul 1

## Elemente introductive

### Regiștrii generali ai procesoarelor Intel

	31	15	8 7	0	
EAX		AH	AL		AX
EBX		BH	BL		BX
ECX		CH	CL		CX
EDX		DH	DL		DX
ESI					SI
EDI					DI
EBP					BP
ESP					SP
EIP					IP

Regiștrii generali pot fi folosiți atât pentru a reține date (toți), cât și pentru formarea adreselor locațiilor de memorie (numai cei pe 32 biți). Excepție fac IP/EIP, care nu apar în mod explicit în nici o instrucțiune; rolul lor este exclusiv în formarea adresei următoarei instrucțiuni de executat.

Observații:

1. Regiștrii din tabelul de mai sus nu sînt complet independenți. De exemplu, faptul că registrul EAX îl conține pe AX înseamnă că orice modificare a valorii lui AX afectează automat și valoarea lui EAX și reciproc; similar pentru regiștrii de 1 octet.

2. În practică, regiștrii (E)SP și (E)BP nu sînt disponibili pentru calcule, rolul lor fiind legat de gestionarea stivei. Ca urmare, programatorul are cu adevărat la dispoziție regiștrii EAX, EBX, ECX, EDX, ESI, EDI și regiștrii de dimensiuni mai mici care fac parte din aceștia.

### Indicatorii de condiții

Procesorul permite testarea existenței anumitor situații speciale după execuția unei instrucțiuni. Mai concret, procesorul scrie valorile 0 sau 1 în anumiți biți dedicați, numiți indicatori de condiții (*flags*), în funcție de rezultatul obținut de către instrucțiunea curentă; valorile acestora pot fi apoi testate prin instrucțiuni specifice. Procesorul are mai mulți asemenea indicatori, fiecare corespunzînd unei anumite condiții. Mai jos sînt prezentați cei 4 indicatori care vor fi utilizați în continuare:

- *carry*: ia valoarea 1 dacă se generează transport în urma adunării/scăderii a două numere, 0 în caz contrar; poate fi folosit și de către alte instrucțiuni, pentru indicarea altor situații similare
- *overflow*: ia valoarea 1 dacă se produce o depășire în urma adunării/scăderii a două numere (considerate ca numere cu semn), 0 în caz contrar
- *zero*: ia valoarea 1 dacă și numai dacă rezultatul ultimei instrucțiuni este 0
- *sign*: preia semnul rezultatului obținut de ultima instrucțiune executată

### Formarea adreselor

Sintactic, o adresă trebuie scrisă între paranteze pătrate. Modurile de adresare arată cum se poate preciza o adresă. Nu vom folosi denumirile din terminologia standard, care pot crea confuzie.

a) Adresa este o constantă

Exemplu: [100]

b) Adresa este precizată prin valoarea unui registru general pe 32 biți

Exemplu: [EAX] (se poate scrie și cu litere mici)

c) Adresa este dată ca suma dintre valoarea unui registru general pe 32 biți și o constantă

Exemplu: [EBX+5]

d) Suma a doi regiștri generali pe 32 biți

Exemplu: [ECX+ESI]

e) Combinația celor două variante anterioare: suma a 2 regiștri și a unei constante

Exemplu: [EDX+EBP+14]

f) Suma a 2 regiștri, dintre care unul înmulțit cu 2, 4, sau 8, la care se poate aduna o constantă

Exemple: [EAX+EDI\*2], [ECX+EDX\*4+5]

Toți regiștrii generali pe 32 biți pot fi folosiți la adresare, fără restricții și fără diferențe între ei. Evident, într-un limbaj de nivel înalt pentru accesarea datelor se folosesc numele de variabile; la nivelul limbajului de asamblare trebuie precizate adresele corespunzătoare.

Teoretic, prima formă de adresare ar putea fi suficientă. În practică, o situație de tipul parcurgerii elementelor unui tablou arată deja că nu se poate lucra numai cu valori constante pentru adrese.

### Instrucțiuni în limbaj de asamblare în Visual C++

Scopul acestor laboratoare nu este de a învăța să scriem cod exclusiv în limbaj de asamblare, ci de a scrie porțiuni de cod în limbaj de asamblare inserate în programe C (sau C++).

Pentru a insera cod scris în limbaj de asamblare într-un program C, se folosește cuvântul cheie `_asm`:

```
_asm {  
...                               //comentariile sînt la fel ca în C sau C++  
}
```

Instrucțiunile în limbaj de asamblare pot fi separate prin `;` (ca în limbajul C) sau `Enter`. Pentru claritate, se recomandă scrierea câte unei singure instrucțiuni pe un rînd.

### Instrucțiunea de atribuire

***mov*** *destinație, sursa*

Operanzii pot fi constante, regiștri sau adrese de memorie (variabile în terminologia C). Exemple de combinare a operanzilor:

```
mov eax, ebx  
mov cx, 5  
mov bl, [eax]  
mov [esi], edx
```

Un caz special este atunci cînd un operand este o adresă de memorie, iar celălalt o constantă. În exemplele de mai sus, cel puțin unul dintre operanzi a fost întotdeauna un registru, care impunea dimensiunea celui alt operand. Aici, atât adresa de memorie, cât și constanta pot ocupa 1, 2 sau 4 octeți. Din acest motiv, trebuie ca programatorul să precizeze dimensiunea operanzilor:

```
mov byte ptr [ecx], 5      //operanzi pe 1 octet  
mov word ptr [ecx], 5      //operanzi pe 2 octeți  
mov dword ptr [ecx], 5     //operanzi pe 4 octeți
```

Nu este posibil ca ambii operanzi să fie adrese de memorie:

```
mov byte ptr [eax], [ebx]  //greșit, compilatorul va semnaliza eroare
```

Toate precizările legate de tipul și dimensiunea operanzilor sînt valabile și pentru celelalte instrucțiuni.

Adresele variabilelor declarate într-un program C nu sînt cunoscute de programator. Pentru ca variabilele să poată fi totuși accesate din limbajul de asamblare, Visual C++ permite folosirea denumirilor simbolice. Cu alte cuvinte, dacă într-un program avem următoarea declarație:

```
int a;
```

putem scrie o instrucțiune de tipul următor în cadrul unui bloc `_asm`:

```
mov a, 3
```

Compilatorul va genera din această linie o instrucțiune de atribuire care poate fi înțeleasă și executată de procesor, înlocuind numele variabilei `a` cu adresa sa. Mai mult, se observă că în acest caz nu mai este necesar să se precizeze dimensiunea operanzilor, deși avem o adresă de memorie și o constantă, deoarece compilatorul o poate afla din declarația variabilei `a` (tipul `int` ocupă 4 octeți). Dacă însă dorim să accesăm doar octetul cel mai puțin semnificativ din `a`, putem scrie:

```
mov byte ptr a, 9
```

Pe de altă parte, următoarea secvență de cod nu este corectă:

```
int a=8, b=9;
```

```
_asm {  
  mov a, b  
}
```

Ambii operanzi fiind adrese de memorie, compilatorul nu poate genera o instrucțiune pentru procesor care să realizeze transferul direct. Trebuie scris:

```
_asm {  
  mov eax, b  
  mov a, eax  
}
```

## Capitolul 2

### Instrucțiuni aritmetice

#### Adunarea

**add op1, op2**

Efect:  $op1 += op2$ ;

Combinații de parametri - la fel ca la atribuire:

```
add eax, ebx
add dl, 3
add si, [...]
add [...], ebp
add byte ptr [...], 14
add word ptr [...], 14
add dword ptr [...], 14
```

În continuare, nu este posibil ca ambii operanzi să fie adrese de memorie:

```
add byte ptr [...], [...] // eroare
```

În exemplele de mai sus, ca și în cele care vor urma, prin [...] se înțelege un operand de tip adresă de memorie, putând fi folosit oricare mod de adresare posibil.

Suma a două numere este calculată corect de instrucțiunea add, indiferent dacă numerele sînt cu semn sau fără semn. Cu alte cuvinte, nu contează dacă operanzii corespund în limbajul C tipurilor int sau unsigned ori corespondenților acestora pe 1 sau 2 octeți.

În cazul adunării a două numere fără semn, este posibil ca rezultatul să nu fie corect, dacă numerele sînt prea mari. Indicatorul *carry* este deci poziționat. În mod similar, pentru numere cu semn este semnificativ indicatorul *overflow*.

O situație care trebuie analizată este calculul sumei a două variabile într-o a treia variabilă:

```
void main()
{
    int a=10, b=5, c;
    _asm {
        mov eax, a
        add eax, b
        mov c, eax
    }
    cout<<c;
}
```

Relativ la indicatorii de condiții, se pot face teste cu tipurile int și unsigned:

```
int main()
{
    int i1=1000000000, i2=2000000000;
    cout<<i1+i2;           // rezultatul este greșit și în C, nu e o eroare de programare
    _asm {
        mov eax, i2
        add i1, eax
    }
    cout<<i1;
}
```

respectiv:

```
int main()
{
    unsigned i1=1000000000, i2=2000000000;
    cout<<i1+i2;           // acum rezultatul este bun
    i1=3000000000;
    cout<<i1+i2;           // acum este greșit
    _asm {
        mov eax, i2
        add i1, eax
    }
}
```

```

}
cout<<i1;
}

```

De remarcat faptul că procesorul poziționează întotdeauna atât *carry*, cât și *overflow* (de fapt toți indicatorii), indiferent dacă lucrăm cu numere cu semn sau fără semn, pentru că nu are de unde ști în care situație ne aflăm. Este sarcina programatorului să testeze indicatorul potrivit.

### Scăderea

**sub op1, op2**

Efect:  $op1 -= op2$ ;

Putem menționa aici relevanța indicatorului *zero*, care în acest caz indică egalitatea celor doi operanzi inițiali. Acest indicator are semnificație și pentru operația de adunare, caz în care însă utilitatea sa practică este mai redusă.

### Incrementarea și decrementarea

**inc op**

**dec op**

Efect:

$op++$ ;

$op--$ ;

Aceste instrucțiuni pot lucra atât cu regiștri, cât și cu adrese de memorie.

### Înmulțirea

**mul op**

**imul op**

Efect: operandul este înmulțit cu un registru implicit, iar rezultatul este depus într-o destinație de asemenea implicită; atât registrul implicit, cât și destinația depind de dimensiunea operandului. Regula generală este că operanzii (cel explicit și cel implicit) trebuie să aibă aceeași dimensiune, iar destinația - dimensiune dublă. Concret:

dimensiune operand explicit	operand implicit	destinație rezultat
1	al	ax
2	ax	(dx, ax)
4	eax	(edx, eax)

În cazul primei instrucțiuni, operanzii sînt considerați numere fără semn, în timp ce a doua instrucțiune consideră operanzii numere cu semn. Spre deosebire de adunare și scădere, la înmulțire și împărțire algoritmi de calcul sînt diferiți pentru numere cu semn și fără semn, deci sînt necesare instrucțiuni diferite.

Dacă dimensiunea operanzilor este 4, rezultatul are 8 octeți. Cum nu există regiștri atât de mari, se folosește perechea  $(edx, eax)$ , în care registrul *edx* conține partea mai semnificativă. În cazul operanzilor pe 2 octeți, deși rezultatul ar încăpea într-un registru de 4 octeți, destinație este perechea  $(dx, ax)$ , pentru a se păstra compatibilitatea cu procesoarele mai vechi, pe 16 biți.

Operandul explicit poate fi un registru sau o locație de memorie, dar nu o constantă:

```

mul ebx           // eax*ebx → (edx, eax)
mul cx
mul al           // se ridică AL la pătrat
mul byte ptr [...] // eax[...] → (edx, eax)
mul word ptr [...]
mul dword ptr [...]
mul byte ptr 10   // eroare

```

Cînd operandul explicit este o locație de memorie, trebuie precizată dimensiunea sa, altfel nu se va ști care este operandul implicit și nici unde trebuie depus rezultatul.

### Împărțirea

**div op**

**idiv op**

Efectul: analog cu instrucțiunea *mul*. Operandul explicit este împărțitorul. Deîmpărțitul este implicit și depinde de dimensiunea împărțitorului, la fel și destinațiile unde se depun cîțul și restul:

dimensiune împărțitor	deîmpărțit	cît	rest
1	ax	al	ah
2	(dx, ax)	ax	dx
4	(edx, eax)	eax	edx

După cum se observă, în toate cazurile, cîtul este depus în jumătatea mai puțin semnificativă a deîmpărțitului, iar restul în jumătatea mai semnificativă. Acest mod de plasare a rezultatelor permite reluarea operației de împărțire în buclă, dacă este cazul, fără a mai fi nevoie de operații de transfer suplimentare.

Analog cu înmulțirea, operandul explicit (împărțitorul) poate fi un registru sau o locație de memorie, dar nu o constantă:

```
div ebx
div cx
div dh
div byte ptr [...]
div word ptr [...]
div dword ptr [...]
div byte ptr 10           // eroare
```

La fel ca la înmulțire, pentru numere cu semn există instrucțiunea `idiv`.

## Capitolul 3

### Instrucțiuni pe biți

#### Instrucțiuni booleene

Sînt implementate operațiile booleene binare AND, OR, XOR și cea unară NOT. Denumirile instrucțiunilor sînt chiar acestea, iar operandii sînt (la fel ca la atribuire sau adunare) variabile de memorie, regiștri, constante. La fel ca înainte, primul operand este și destinația rezultatului:

```
not eax           // se complementează fiecare bit din EAX
and bx, 16        // se face AND între biții de pe aceeași poziție din cei doi operandi
or byte ptr [...], 100
xor [...], ecx
```

Există aceleași limitări privitoare la combinațiile de operandi: nu se poate ca ambii operandi să fie adrese de memorie și nici ca operandii să aibă dimensiuni diferite. Evident, aceste restricții se aplică operațiilor binare. Ca urmare, instrucțiunile de mai jos sînt greșite:

```
and word ptr [...], [...]
or dx, eax
```

Operatorii echivalenți din limbajul C sînt  $\sim$ ,  $\&$ ,  $|$ ,  $\wedge$ ; de fapt, dat fiind că primul operand este și destinație, mai corect ar fi să spunem  $\sim=$ ,  $\&=$ ,  $|=$ ,  $\wedge=$ .

La ce folosesc acești operatori? La fel ca și în limbajul C, sînt destul de rar folosiți. Utilitatea lor principală constă în lucrul cu măști.

Să considerăm că vrem să testăm bitul de pe poziția 3 din registrul AX. Ne putem folosi de flagul *zero* al procesorului; din păcate, acesta ne dă informații doar despre întreg operandul și nu doar despre un bit al său. Nu ne rămîne deci decît să alterăm registrul AX astfel încît acesta să aibă valoarea 0 dacă și numai dacă bitul său de pe poziția 3 este 0. În acest scop, trebuie să forțăm toți biții din AX pe valoarea 0, cu excepția celui de pe poziția 3, care va păstra valoarea inițială (cea pe care vrem să o testăm). Funcția booleană care poate realiza aceste prelucrări (forțarea pe 0, respectiv conservare) este AND. Va trebui deci să facem AND între AX și un operand construit astfel încît să aibă valoarea 1 pe poziția 3 și 0 în rest:

```
and ax, 8
ax  B15B14B13B12B11B10B9B8B7B6B5B4B3B2B1B0
8   0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
ax  0  0  0  0  0  0  0  0  0  0  0  0  B3 0  0  0
```

În concluzie: dacă  $B_3=0$ , la final  $AX=0$ ; dacă  $B_3=1$ ,  $AX \neq 0$ , deci flagul *zero*, care dă informații despre AX, va indica de fapt valoarea bitului  $B_3$  din AX.

Valoarea constantă cu care se realizează operația AND se numește mască; se observă că, într-adevăr, rolul său este de a "filtra" biții de pe pozițiile care nu interesează, forțându-i la o valoare neutră. Pentru exemplul de mai sus, dacă dorim să testăm bitul de pe poziția  $i$ , masca va avea valoarea  $2^i$ .

Desigur, se pot folosi și celelalte funcții booleene. De asemenea, măștile pot lăsa nemodificați mai mulți biți, nu doar unul singur. Mai mult, putem folosi ca mască un registru sau o variabilă (în locul constantei), astfel încît masca se poate modifica pe parcursul execuției; depinde doar de ceea ce ne propunem să obținem. Totuși, operația cel mai des întîlnită este AND. Ajunși aici, putem face observația că, de fapt, instrucțiunea de mai sus nu este cu adevărat o testare, deoarece modifică operandul testat. Oricît de mult ne-ar interesa valoarea bitului de pe poziția 3, nu ne dorim să pierdem valorile din ceilalți biți. Proiectanții procesorului rezolvă problema punîndu-ne la dispoziție o instrucțiune specială, numită chiar *test*. Astfel, în locul instrucțiunii de mai sus, putem scrie:

```
test ax, 8
```

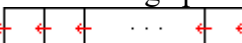
Funcția booleană realizată este tot AND. Diferența constă în faptul că registrul AX nu este modificat - de fapt, rezultatul nu este depus nicăieri. În schimb, indicatorul *zero* este poziționat în același mod. Dacă dorim să facem doar o testare (nedistructivă) și nu o prelucrare a valorii registrului AX, aceasta este exact instrucțiunea necesară.

Nu există instrucțiuni similare pentru celelalte funcții logice; aceasta arată că, într-adevăr, funcția AND este cea mai folosită.

#### Instrucțiuni de deplasare

Sînt instrucțiuni care schimbă pozițiile biților în cadrul operandilor.

Deplasarea la stînga pe o poziție are efectul următor:

Carry  0

Similar pentru deplasarea la dreapta.

Instrucțiunile de deplasare sînt:

```
shl eax, 1           // deplasare la stînga a registrului EAX cu o poziție
shl byte ptr [...], 3 // se poate și cu mai multe poziții
shl dx, cl           /* numărul de poziții pe care se face deplasarea poate fi specificat
în CL - singurul registru care poate fi folosit pentru așa ceva (nu este permis nici CX sau ECX) */
shr bh, 1           // deplasare la dreapta
```

Desigur, dacă deplasarea se face pe mai multe poziții, flagul *carry* va reține doar ultimul bit ieșit.

Operatorii C echivalenți sînt << și respectiv >> (mai exact <<= și >>=).

Deplasările sînt folosite cel mai des în scopuri aritmetice. Este bine cunoscut faptul că o deplasare la stînga pe  $n$  poziții este echivalentă cu înmulțirea cu  $2^n$ , în timp ce deplasarea la dreapta reprezintă o împărțire la  $2^n$ . Folosirea deplasărilor cu precădere în acest scop ridică însă și o problemă: cum procedăm dacă numărul este cu semn? Să considerăm deplasarea la dreapta. Dacă inițial bitul cel mai semnificativ avea valoarea 1, iar instrucțiunea de deplasare introduce valoarea 0 pe poziția respectivă, s-a schimbat semnul numărului, ceea ce nu este corect din punct de vedere aritmetic. Din acest motiv, procesorul mai are o instrucțiune de deplasare la dreapta, special pentru numerele cu semn:

```
sar eax, 1
```

Efectul său este asemănător cu al instrucțiunii `shr`, dar bitul cel mai semnificativ își păstrează valoarea inițială (și trece și spre dreapta, desigur). Se poate demonstra că în acest mod se realizează o împărțire corectă la  $2^n$  pentru numere cu semn. Avem deci două instrucțiuni de deplasare spre dreapta; una e potrivită pentru tipurile de date fără semn (`shr`), cealaltă pentru tipurile cu semn (`sar`).

La deplasarea spre stînga, din păcate, nu există o soluție similară. Limbajul de asamblare acceptă instrucțiunea `sal`, dar este de fapt alt nume pentru `shl`. Este ușor de văzut de fapt că putem interveni doar asupra bitului care "intră", care în acest caz este cel mai puțin semnificativ, deci nu are efect asupra semnului.

## Capitolul 4

### Instrucțiuni de salt

Din punct de vedere practic, o instrucțiune de salt modifică valoarea registrului contor program (EIP) și implicit face ca instrucțiunea următoare care se execută să nu fie cea care urmează în memorie. Utilitatea acestor instrucțiuni este clară: orice structură de control (testare sau buclă) se poate implementa la nivelul limbajului procesorului doar prin salturi.

#### Saltul necondiționat

##### **jmp** *adresa*

Deși există și alte forme de exprimare a adresei de salt, în continuare ne vom rezuma la cazul în care adresa este o constantă. Evident, la momentul scrierii codului nu putem cunoaște adresa reală din memorie; ca urmare, vom lăsa gestiunea adreselor în seama compilatorului, prin utilizarea etichetelor, la fel ca în cazul instrucțiunii `goto` din C. Visual C++ este flexibil în această privință: se pot face salturi spre etichete definite în același bloc `_asm`, în alt bloc sau în partea de cod C. De altfel, este posibil și invers: o instrucțiune `goto` să sară la o etichetă definită într-un bloc `_asm`.

Utilitatea saltului necondiționat, luat singur, nu este foarte mare, deoarece nu introduce cu adevărat o ramificație în program - nu avem variante de execuție.

#### Salturi condiționate

În acest caz avem două variante de execuție:

- condiția de salt este adevărată - se face saltul la adresa indicată
- condiția de salt este falsă - se continuă cu instrucțiunea următoare din memorie, ca și cum n-ar fi existat nici o instrucțiune de salt

Cele mai simple sînt instrucțiunile care testează indicatorii de condiții individuali. Vom considera în continuare doar indicatorii cei mai folosiți (deja cunoscuți): *carry*, *overflow*, *zero* și *sign*.

Pentru fiecare indicator există două instrucțiuni de salt condiționat: una care realizează saltul cînd indicatorul testat are valoarea 1 și una care realizează saltul cînd indicatorul are valoarea 0.

indicator testat	salt pe valoarea 1	salt pe valoarea 0
Carry	<code>jc</code>	<code>jnc</code>
Overflow	<code>jo</code>	<code>jno</code>
Zero	<code>jz</code>	<code>jnz</code>
Sign	<code>js</code>	<code>jns</code>

Evident, pentru fiecare instrucțiune de mai sus trebuie precizată și adresa de salt.

În realitate, acești indicatori individuali nu sînt foarte utili. După cum se știe, în limbajul C se fac teste în principal pe baza operatorilor relaționali (<, <=, ==, !=, >, >=). În general, aceste relații între operanzi nu pot fi determinate testînd cîte un singur indicator, ci sînt necesare instrucțiuni mai complexe. În acest sens, considerăm mai întîi instrucțiunea de comparare:

##### **cmp** *op1, op2*

Combinățiile de operanzi care pot fi folosiți sînt aceleași ca la instrucțiunile studiate anterior, începînd cu atribuirea. Această instrucțiune realizează intern o scădere, însă fără a altera operanzii (nu depune rezultatul nicăieri); în schimb, indicatorii de condiții sînt poziționați la fel. Putem deci folosi în același scop și instrucțiunea `sub`, dacă nu ne afectează modificarea primului operand. Cel mai important este faptul că, analizînd toți indicatorii (nu doar unul), putem decide care a fost relația între operanzii instrucțiunii de comparare. Deoarece ar fi prea complicat să scriem mai multe salturi condiționate relaționate între ele, proiectanții procesoarelor Intel au introdus un set de salturi condiționate care fac testele corespunzătoare:

relație	instrucțiune salt
<code>op1 &lt; op2</code>	<code>jb</code>
<code>op1 &lt;= op2</code>	<code>jbe</code>
<code>op1 &gt; op2</code>	<code>ja</code>
<code>op1 &gt;= op2</code>	<code>jae</code>
<code>op1 == op2</code>	<code>je</code>
<code>op1 != op2</code>	<code>jne</code>

Problemă: relațiile între operanzi diferă în funcție de tipul acestora (cu semn/fără semn). De exemplu, să presupunem că operanzii arată astfel:

01001011



10011010

Dacă operandii sînt fără semn, evident al doilea operand este mai mare. Dacă însă sînt numere cu semn, primul operand e pozitiv, iar al doilea negativ, astfel încît relația între ei se inversează. Pe de altă parte, instrucțiunea de comparație rămîne aceeași. Instrucțiunile de mai sus sînt de fapt pentru operanzi fără semn. Pentru operanzi cu semn avem alt set de instrucțiuni:

relație	instrucțiune salt
op1 < op2	j1
op1 <= op2	jle
op1 > op2	jg
op1 >= op2	jge
op1 == op2	je
op1 != op2	jne

Se observă (cum era de așteptat) că testele de egalitate și inegalitate sînt identice pentru cele două tipuri de operanzi.

Putem face observația că unele din aceste instrucțiuni testează un singur indicator, deci sînt identice cu unele din primul set. De exemplu, je și jz reprezintă de fapt aceeași instrucțiune. Nu este însă necesar să reținem aceste detalii.

## Structuri de control

### Structura If

În limbajul C, condiția testată într-o structură de control de tip *if* decide dacă blocul de cod subordonat trebuie executat. În schimb, în limbaj de asamblare, instrucțiunile de salt pot fi folosite pentru a evita execuția unei secvențe de cod. Din acest motiv, pentru implementarea în limbaj de asamblare a unei asemenea structuri, condiția testată va fi inversa celei din codul C.

Exemplu: Fie codul C de mai jos:

```
if (x>5)
```

```
    a=2;
```

În limbaj de asamblare, acesta se traduce prin:

```
cmp x, 5
```

```
jle maimare
```

```
// saltul se execută dacă x≥5
```

```
mov a, 2
```

```
// dacă nu s-a executat saltul, se execută a=2;
```

```
maimare:
```

### Structurile For și While

Implementarea în limbaj de asamblare a unei structuri de tip *for* implică două instrucțiuni de salt: una pentru ieșirea din buclă, cealaltă pentru reluarea buclei. Prima instrucțiune de salt testează, la fel ca în cazul structurii *if*, condiția inversă celei exprimate în limbajul C. A doua instrucțiune de salt este necondiționată, deoarece, dacă s-a ajuns cu execuția în punctul respectiv, reluarea buclei este obligatorie. Traducerea în limbaj de asamblare trebuie să țină cont de faptul că instrucțiunea *for* din limbajul C este compusă, deci este necesară multă atenție la ordinea în care se execută părțile componente ale sale.

Exemplu: Fie codul C de mai jos:

```
for (i=0; i<=5; i++)
```

```
    s+=i;
```

În limbaj de asamblare, acesta se traduce prin:

```
mov i, 0
```

```
// inițializare
```

```
buc1a:
```

```
cmp i, 5
```

```
// testul de continuare/ieșire
```

```
jg afara
```

```
// dacă i>5, se iese din buclă
```

```
mov eax, s
```

```
// începe corpul buclei
```

```
add eax, i
```

```
mov s, eax
```

```
inc i
```

```
// actualizare contor
```

```
jmp buc1a
```

```
// reluarea buclei
```

```
afara:
```

Structura de tip *while* este similară structurii *for*, deci implementarea sa se realizează în același mod.

### Structura Do-While

Structura de tip *do-while* diferă de cele anterioare prin faptul că testul de continuare/ieșire se realizează la finalul său. Din acest motiv, în limbaj de asamblare se testează aceeași condiție ca în limbajul C, nemaifiind necesară inversarea condiției de test.

Exemplu: Fie codul C de mai jos:

```
do {  
    s+=i;  
    i++;  
} while(i<=5);
```

În limbaj de asamblare, acesta se traduce prin:

```
buc1a:  
mov  eax,s                // începe corpul buclei  
add  eax,i  
mov  s,eax  
inc  i  
cmp  i,5  
jle  buc1a                // reluarea buclei dacă  $i \leq 5$ 
```

## Capitolul 5

### Lucrul cu stiva

#### Instrucțiuni pentru lucrul cu stiva

Întrucât o structură de tip stivă este necesară în multe situații, procesorul folosește și el o parte din memoria RAM pentru a o accesa printr-o disciplină de tip LIFO. După cum se știe, singura informație fundamentală pentru gestiunea stivei este vârful acesteia. În cazul procesorului, adresa la care se află vârful stivei este memorată în registrul ESP.

Instrucțiunea de introducere în stivă se numește, evident, `push` și are un singur operand. Exemple:

```
push eax
push dx
push dword ptr [...]
push word ptr [...]
push dword ptr 5
push word ptr 14
```

Se observă că stiva lucrează doar cu valori de 2 sau 4 octeți. De fapt, pentru uniformitate, se recomandă a lucra numai cu operanzi pe 4 octeți; varianta cu 2 octeți este prezentă doar pentru compatibilitatea cu procesoarele mai vechi.

Ce se întâmplă când se execută o asemenea instrucțiune? Procesorul scrie valoarea operandului la adresa indicată de ESP (vârful stivei), apoi scade din valoarea ESP dimensiunea în octeți a operandului pus în stivă (2 sau 4); în acest mod, vârful stivei este pregătit pentru următoarea operație de scriere. De exemplu, instrucțiunea `push eax` ar fi echivalentă cu:

```
mov [esp], eax
sub esp, 4
```

Diferența e dată de faptul că procesorul face singur toate aceste acțiuni (și astfel reduce riscul apariției erorilor). Observăm aici că stiva avansează "în jos", adică de la adrese mari spre adrese mici.

Instrucțiunea inversă, de scoatere din stivă, se numește `pop`. După cum era de așteptat, lucrează tot numai cu operanzi de 2 sau 4 octeți:

```
pop eax
pop cx
pop dword ptr [...]
pop word ptr [...]
```

Evident, la execuția acestei instrucțiuni, operandul joacă rolul destinației în care se depune valoarea luată din vârful stivei (ESP); apoi, la valoarea ESP se adună numărul de octeți ai operandului - acesta indică și numărul de octeți scoși din stivă.

Rolul stivei procesorului este de a stoca informații cu caracter temporar. De exemplu, dacă avem nevoie să folosim un registru pentru unele operații, dar nu avem la dispoziție nici un registru a cărui valoare curentă să ne permitem să o pierdem, procedăm ca mai jos:

```
push eax
... // utilizare EAX
pop eax
```

În acest mod am putut folosi temporar registrul EAX, dar valoarea sa inițială a fost păstrată.

De asemenea, variabilele locale sînt plasate tot în stivă. Reamintim că o variabilă locală este creată la momentul apelului funcției în care este declarată și distrusă la terminarea acelei funcții, deci are tot un caracter temporar.

Evident, lucrul cu stiva necesită multă atenție; instrucțiunile de introducere în stivă trebuie riguros compensate de cele de scoatere din stivă - aici este vorba atît de numărul acestor instrucțiuni, cît și de dimensiunea operanzilor. Orice eroare afectează în general mai multe date. De exemplu, să considerăm secvența următoare:

```
push eax
push edx
mov eax, [esi]
mov edx, 5
mul edx
mov [esi], eax
pop eax
```

Înmulțirea variabilei cu 5 se realizează corect. În schimb, pentru că o instrucțiune `pop` a fost uitată, sînt afectate valorile ambilor regiștri implicați: `EAX` primește altă valoare decît avea inițial, iar valoarea `EDX` rămîne cea rezultată din înmulțire, fără a fi restaurată cea veche. Similar stau lucrurile dacă se fac prea multe instrucțiuni `pop`. În majoritatea cazurilor, operandul depus în stivă printr-o instrucțiune `push` este și destinația instrucțiunii `pop` corespunzătoare (dar nu este 100% obligatoriu, depinzînd de natura programului).

O altă eroare poate apărea atunci cînd registrul `ESP` este manipulat direct. De exemplu, pentru a alocă spațiu unei variabile locale (neinițializată), e suficient a scădea din `ESP` dimensiunea variabilei respective. Similar, la distrugerea variabilei, valoarea `ESP` este crescută. Aici nu se folosesc în general instrucțiuni `push`, respectiv `pop`, deoarece nu interesează valorile implicate, ci doar ocuparea și eliberarea de spațiu. Se preferă adunarea și scăderea direct cu registrul `ESP`; evident că o eroare în aceste operații are consecințe de aceeași natură ca și cele de mai sus.

### Apelul funcțiilor

Un apel de funcție arată la prima vedere ca o instrucțiune de salt, în sensul că se întrerupe execuția liniară a programului și se sare la o cu totul altă adresă. Diferența fundamentală constă în faptul că la terminarea funcției se revine la adresa de unde s-a făcut apelul și se continuă cu instrucțiunea următoare. Din moment ce într-un program se poate apela o funcție de mai multe ori, din mai multe locuri, și întotdeauna se revine unde trebuie, este clar că adresa la care trebuie revenit este memorată și folosită atunci cînd este cazul. Cum adresa de revenire este în mod evident o informație temporară, locul său este tot pe stivă.

Apelul unei funcții se realizează prin instrucțiunea `call`, care are următoarea sintaxă:

**call** *adresa*

Evident, în Visual C++ vom folosi nume simbolice pentru a preciza adresa, cu singura mențiune că de data aceasta nu este vorba de etichete, ca la salturi, ci chiar de numele funcțiilor apelate.

Efectul instrucțiunii `call` este următorul: se introduce în stivă adresa instrucțiunii următoare (adresa de revenire) și se face salt la adresa indicată. Aceste acțiuni puteau fi realizate și cu instrucțiuni `push` și `jmp`, dar din nou procesorul face singur totul și astfel ne ferește de erori.

Revenirea dintr-o funcție se face prin instrucțiunea `ret`, preia adresa de revenire din vârful stivei (similar unei instrucțiuni `pop`) și se face saltul la adresa respectivă.

Acum devine clar că o eroare în lucrul cu stiva are consecințe și mai grave decît s-a văzut anterior. O instrucțiune `pop` omisă (sau una în plus) are ca efect faptul că, la execuția unei instrucțiuni `ret`, se va prelua din stivă o cu totul altă valoare decît adresa corectă de revenire. Efectul practic al acestui salt greșit este blocarea programului sau terminarea sa forțată - și întotdeauna există apeluri de funcții într-un program (cel puțin funcția `main`).

### Funcții și parametri

Cum se transmit parametrii unei funcții? Parametrii sînt tot variabile locale, deci se găsesc pe stivă. Cel care face apelul are responsabilitatea de a-i pune pe stivă la apel și de a-i scoate de pe stivă la revenirea din funcția apelată. Avem la dispoziție instrucțiunea `push` pentru plasarea în stivă. Evident, această operație trebuie realizată imediat înainte de apelul propriu-zis. În plus, în limbajul C/C++, parametrii trebuie puși în stivă în ordine inversă celei în care se găsesc în lista de parametri. La revenire, parametrii trebuie scoși din stivă, nemaifiind necesari. Cum nu ne interesează preluarea valorilor lor, nu se folosește instrucțiunea `pop`, ci se adună la `ESP` numărul total de octeți ocupat de parametri (reamintim faptul că pe stivă se lucrează în general cu 4 octeți, chiar și atunci cînd operanzii au dimensiuni mai mici).

Să luăm ca exemplu funcția următoare:

```
void dif(int a,int b)
{
    int c;
    c=a-b;
    cout<<c;
}
```

Apelul `dif(9,5)` se traduce prin secvența de mai jos:

```
push dword ptr 9
push dword ptr 5
call dif
add esp,8
```

## Valori returnate de către funcții

Cum poate o funcție să returneze o valoare? Convenția în Visual C++ (și la majoritatea compilatoarelor) este că rezultatul se depune într-un anumit registru, în funcție de dimensiunea sa:

- pentru tipurile de date de dimensiune 1 octet - în registrul AL
- pentru tipurile de date de dimensiune 2 octeți - în registrul AX
- pentru tipurile de date de dimensiune 4 octeți - în registrul EAX
- pentru tipurile de date de dimensiune 8 octeți - în regiștii EDX și EAX

Evident, la revenirea din funcție, cel care a făcut apelul trebuie să preia rezultatul din registrul corespunzător.

## Structura generală a unei funcții

La apelul unei funcții, adresa de revenire este plasată în stivă după parametri. Ca urmare, în timpul execuției funcției, parametrii nu pot fi scoși din stivă prin instrucțiuni `pop`, deoarece ar trebui mai întâi extrasă adresa de revenire, ceea ce nu este permis.

Soluția este de a accesa parametrii pe baza adreselor lor din memorie. La prima vedere, adresele din stivă sînt legate de valoarea registrului ESP. Totuși, orice instrucțiune de tip `push/pop` executată după intrarea în funcție alterează valoarea acestui registru, astfel încît nu ar mai fi posibil să exprimăm adresa unui parametru printr-o formulă fixă. Din acest motiv, la intrarea în funcție, valoarea registrului ESP este copiată în registrul EBP, acesta din urmă fiind utilizat pentru accesul la parametri.

Un aspect important: programul pe care îl scriem este întotdeauna C/C++, deci nu putem ști cum folosește compilatorul regiștrii. Ca urmare, dacă o funcție scrisă de noi în limbaj de asamblare folosește unii regiștri, aceștia trebuie salvați în stivă la începutul execuției funcției și refăcuți la terminare. Excepție fac regiștrii EAX și EDX, care, după cum am văzut, sînt folosiți pentru a returna valori, deci îi putem modifica întotdeauna fără a fi necesară salvarea/restaurarea lor.

La modul general, structura unei funcții este deci următoarea:

```
push ebp           // se salvează în stivă valoarea registrului EBP (necesară funcției apelante)
mov ebp, esp       // se memorează valoarea curentă a vârfului stivei în registrul EBP
push ...           // se salvează în stivă regiștrii care vor fi modificate în interiorul funcției
.....           // se execută prelucrările corespunzătoare funcției
pop ...            // se refac din stivă valorile regiștrilor care au fost modificate
pop ebp            // se reface din stivă valoarea anterioară a registrului EBP
ret
```

Se observă că primele două instrucțiuni, la fel ca și ultimele două, sunt mereu aceleași. Din acest motiv, ele sînt generate automat de către compilator, deci nu este sarcina programatorului de a le scrie.

Datorită modului de a lucra cu stiva, parametrii funcției pot fi accesați pe baza următoarelor adrese:

- primul parametru din listă - adresa EBP+8
- următorul parametru - adresa EBP+12
- etc.

În Visual C++ parametrii pot fi accesați și prin intermediul numelor lor, ceea ce este adesea mai simplu.

## Capitolul 6

### Tablouri și pointeri

#### Tablouri unidimensionale

Pentru a accesa un anumit element dintr-un tablou, este necesar să fie determinată adresa sa. Aceasta se realizează cu ajutorul următoarelor informații:

- adresa de început a tabloului
- indicele elementului în cadrul tabloului
- dimensiunea unui element al tabloului

În timp ce dimensiunea elementelor tabloului rezultă direct din tipul acestora, iar indicele este ușor de gestionat, adresa de început a tabloului poate fi exprimată în mai multe forme.

a) În cazul cel mai simplu, adresa de început a tabloului este reprezentată chiar prin numele variabilei respective. De exemplu, considerăm următoarea declarație:

```
int t[50];
```

Presupunem că indicele elementului pe care dorim să îl accesăm este memorat în registrul EAX. În acest caz, elementul respectiv poate fi accesat printr-o instrucțiune de tipul celei de mai jos:

```
mov edx, t[eax*4]
```

Instrucțiunea, care depune valoarea elementului dorit în registrul EDX, poate fi scrisă și în modul următor (cele două moduri de exprimare fiind echivalente):

```
mov edx, t+eax*4
```

Compilatorul cunoaște adresa la care este plasat tabloul, deci va înlocui numele `t` cu adresa corespunzătoare. La această adresă de început se adună valoarea indicelui, înmulțită cu dimensiunea unui element (4 în acest caz).

b) Uneori adresa de început a unui tablou nu este disponibilă direct prin numele variabilei, ci prin intermediul unui pointer. Cel mai adesea (dar nu exclusiv), această situație apare atunci când un tablou este transmis ca parametru al unei funcții, ca în exemplul de mai jos:

```
void f(int t[]);
```

După cum se cunoaște, deși declarația de mai sus pare a indica faptul că întreg tabloul a fost copiat pe stivă, în realitate parametrul este reprezentat doar de adresa de început a tabloului. Astfel, varianta de mai jos a antetului funcției este absolut similară:

```
void f(int *t);
```

În acest caz, evident, încercarea de a accesa un element al tabloului prin modul de adresare indicat mai sus nu va produce rezultate corecte. În schimb, valoarea pointerului trebuie copiată într-un registru, care va fi apoi folosit ca valoare de bază în exprimarea adresei elementului căutat. Reluând exemplul anterior pentru situația în care `t` este pointer, soluția este cea de mai jos:

```
mov esi, t
```

```
mov edx, [esi+eax*4]
```

Observație: În limbajul C, pentru accesul la un element al unui tablou se utilizează aceeași sintaxă, indiferent dacă avem la dispoziție direct numele tabloului sau un pointer către începutul acestuia. Această similaritate, trebuie subliniat, nu este naturală, ci reprezintă o excepție impusă, din anumite motive, de către creatorii limbajului. În realitate, după cum se observă atunci când se lucrează la nivelul limbajului de asamblare, accesul este realizat în moduri diferite.

#### Tablouri bidimensionale

După cum s-a observat deja, limbajul de asamblare permite accesarea elementelor unui tablou printr-o sintaxă în mare măsură similară celei din limbajul C. Acest fapt nu mai este valabil în cazul tablourilor bi- sau multidimensionale, deoarece limbajul de asamblare nu permite utilizarea directă a doi sau mai mulți indici. Din nou, abordarea este diferită în funcție de modul în care este disponibilă adresa de început a tabloului.

a) Considerăm un exemplu de declarație a unei matrice:

```
int m[10][10];
```

Așa cum se cunoaște, elementele matricii sînt plasate liniar în memorie, în ordinea dată de linii și coloane: mai întîi elementele liniei 0 (în ordinea `m[0][0]`, `m[0][1]`, ..., `m[0][9]`), urmate imediat de elementele liniei 1 (`m[1][0]`, `m[1][1]`, ..., `m[1][9]`) ș.a.m.d. Ca urmare, pentru a accesa un element `m[i][j]`, adresa sa este calculată prin formula:

```
&m + (i * 10 + j) * 4
```

unde `&m` reprezintă adresa de început a matricii, 10 este numărul de coloane al matricii, iar 4 este dimensiunea unui element al matricii.

Exemplu: considerăm că indicele liniei (corespunzător variabilei `i` în expresia de mai sus) este reținut în registrul `EBX`, iar indicele coloanei (`j`) în registrul `ECX`. Pentru a copia în registrul `EDX` valoarea elementului `m[i][j]` putem proceda ca mai jos:

```
mov eax, 10
mul ebx
add eax, ecx
mov edx, m[eax*4]
```

Se observă că, având la dispoziție numele variabilei `m`, care indică adresa de început a matricii, modul de adresare este practic similar celui utilizat la tablouri unidimensionale.

b) Considerăm o funcție care primește ca parametru o matrice de felul celei de mai sus:

```
void f(int m[][10]);
```

La fel ca în cazul anterior, parametrul nu reprezintă întreaga matrice, ci este doar un pointer către începutul său. Trebuie deci procedat similar ca la tablourile unidimensionale date ca parametru, în sensul că adresa de început va fi preluată într-un registru, care va fi apoi utilizat ca valoare de bază în exprimarea adresei elementului căutat. Astfel, codul de mai sus suferă o ușoară transformare:

```
mov eax, 10
mul ebx
add eax, ecx
mov edi, m
mov edx, [edi+eax*4]
```

c) Considerăm acum cazul unei matrici alocate dinamic:

```
int **m;
m=new int*[10];
for(i=0;i<10;i++)
    m[i]=new int[10];
```

În acest caz, structura matricii în memorie este total diferită de situațiile anterioare. De fapt, în memorie se găsesc un tablou de pointeri cu 10 elemente (a cărui adresă de început este reținută în variabila `m`) și 10 tablouri de numere întregi cu câte 10 elemente (adresele lor de început reținute respectiv în variabilele `m[0]`, `m[1]`, ..., `m[9]`). Pentru a accesa elementul `m[i][j]`, se procedează astfel:

- folosind adresa `m`, se obține `m[i]` - care conține adresa de început a tabloului corespunzător liniei `i`
- cu ajutorul adresei `m[i]` se accesează `m[i][j]`

Pentru același exemplu din cazurile anterioare, o posibilă implementare în limbaj de asamblare este următoarea:

```
mov edi, m
mov eax, [edi+ebx*4]
mov edx, [eax+ecx*4]
```

Observație: În limbajul C, o matrice definită static, la fel ca în cazul a), poate fi transmisă ca parametru unei funcții precum cea de la cazul b). Sintaxa prin care se accesează elementele matricii este aceeași în ambele cazuri, deși, așa cum s-a văzut, implementarea la nivelul procesorului este ușor diferită. În schimb, o matrice definită dinamic, la fel ca în cazul c), nu poate fi transmisă ca parametru unei asemenea funcții (cazul b), nici nu se poate realiza o atribuire între o matrice alocată static (cazul a) și una alocată dinamic (cazul c). Diferențele de structură ale matricilor sînt prea mari, astfel încât compilatorul nu acceptă operații între aceste tipuri.

## Capitolul 7

### Structuri

Variabilele care formează o structură se găsesc în memorie la adrese consecutive. Ca urmare, pentru a accesa un anumit membru al unei structuri, sînt necesare două elemente:

- adresa de început a structurii
- deplasamentul în cadrul structurii al membrului vizat; acest deplasament este egal cu suma dimensiunilor membrilor anteriori

Exemplu: Considerăm o structură definită ca mai jos:

```
struct Point {  
    int x,y;  
};  
Point p;
```

Membrii structurii `p` pot fi accesați astfel:

```
mov eax,dword ptr p      // citirea p.x (deplasament 0, fiind primul membru)  
mov ebx,dword ptr p+4    // citirea p.y (deplasament 4)
```

La a doua instrucțiune, adresa `p.y` poate fi scrisă și sub forma `p[4]`. Totuși, deoarece aici nu este vorba de tablouri, sintaxa de mai sus este mai sugestivă.

Observație: Utilizarea calificativului `dword ptr` în aceste instrucțiuni este obligatorie. Omiterea sa provoacă o eroare, deoarece, din punct de vedere al compilatorului, se încearcă atribuirea unei variabile de 8 octeți (dimensiunea tipului `Point` și deci a variabilei `p`) către un registru de 4 octeți. Prin sintaxa de mai sus, compilatorul este înștiințat de faptul că al doilea operand este citit din memorie pe dimensiunea de 4 octeți.

În cazul în care se dorește transmiterea unei structuri ca parametru al unei funcții, trebuie menționat că limbajul C nu permite așa ceva în mod direct (în schimb este posibil în C++). Prin urmare, în limbajul C nu există decît posibilitatea de a transmite ca parametru adresa unei structuri:

```
void f(Point *p);
```

Accesul la membrii structurii se face într-un mod similar accesului la elementele unui tablou transmis ca parametru. Diferența constă în faptul că deplasamentul față de adresa de început este determinat prin regulile specifice structurilor și nu tablourilor.

```
mov edi,p                // copierea adresei de început a structurii în registrul EDI  
mov eax,[edi]            // citirea p.x (deplasament 0)  
mov ebx,[edi+4]          // citirea p.y (deplasament 4)
```