Asm 1

Adrese

Orice adresa este formata din doua componente: segment si offset (deplasament), notatia uzuala fiind *segment:offset*. Pentru partea de offset exista mai multe variante:

- Constanta numerica. Exemplu: [100]
- Valoarea unui registru general pe 32 biti. Exemplu: [EAX] (se poate scrie si cu litere mici)
- Suma dintre valoarea unui registru general pe 32 biti si o constanta. Exemplu: [EBX+5]
- Suma a doi registri generali pe 32 biti. Exemplu: [ECX+ESI]
- Combinatia celor 2 variante anterioare: suma a 2 registri si a unei constante. Exemplu: [EDX+EBP+14]
- Suma a 2 registri, dintre care unul inmultit cu 2, 4, sau 8, la care se poate aduna o constanta. Exemple: [EAX+EDI*2], [ECX+EDX*4+5]

Instructiunea de atribuire: mov

Sintaxa: *mov destinatie, sursa*

Efect: pune in destinatie valoarea din sursa.

Destinatia, respectiv sursa, pot fi:

- registru, registru. Exemple: *mov eax, ebx;*, *mov al, bh;*
- registru, adresa de memorie. Exemplu: *mov bl, [eax];*
- adresa de memorie, registru. Exemplu: *mov [esi], esx;*
- registru, constanta numerica. Exemplu: *mov ah, 0;*
- memorie, constanta numerica. Exemplu: *mov [eax], 3;*

Ex. 1:

```
#include <stdio.h>

void main(){
  _asm{
    mov eax, 0;
    mov ah, 1;
  }
}
```

Ex. 2:

```
#include <stdio.h>

void main(){
  int i = 0;
  _asm{
    mov ax, i;
```

```
  }
}
```
Dimensiunea operanzilor:
```
mov byte ptr [eax], 5; //afecteaza 1 octet
mov word ptr [eax], 5; //afecteaza 2 octeti
mov dword ptr [eax], 5; //afecteaza 4 octeti (double word)
```
Instrucţiunea add

Sintaxa: *add op1, op2*

Efect: op1 = op1 + op2

Ex. 1:
```
#include <stdio.h>

void main(){
  int a=10;
  _asm {
    add a,5
  }
  printf("%d\n",a);
}
```
Ex. 2:
```
#include <stdio.h>

void main(){
  _asm {
    mov eax,0xFFFFFFFF;
    add eax,2; // rezultatul este 0x100000001; necesita 33 biti.
               // setare carry
    mov eax,0;
    mov ax, 0xFFFF;
    add ax, 2; // doar ax se modifica!
               // desi rezultatul este 0x10001,
               //  al 17-lea bit din eax nu se modifica.
               // se seteaza carry

  }
  printf("%d\n",c);
}
```
Instrucţiunea sub

Sintaxa: *sub op1, op2*

Efect: op1 = op1 - op2

Ex. 1:

```
#include <stdio.h>

void main()
{
  int a=10,b=14;
  _asm {
    mov eax,b
    sub a,eax
  }
  printf("%d\n",a);
}
```

Instrucţiuni booleene: AND, OR, XOR, NOT
Sintaxa:

- *and* destinatie, sursa
- *or* destinatie, sursa
- *xor* destinatie, sursa
- *not* destinatie

Instrucţiunile *and, or, xor modifică indicatorul ZERO*
Utilitatea principală a acestor instrucţiuni este in lucrul cu măşti. De exemplu, dacă ne interesează valoarea bitului al 5-lea din registrul ax, este suficient să se execute *and* intre *ax* şi valoarea (scrisa binar) 0000000000010000 (aceasta se numeşte mască). Rezultatul operaţiei va fi 0 (iar indicatorul ZERO va deveni 1) dacă bitul al 5-lea din ax are valoarea 0, respectiv va fi diferit de 0 (iar indicatorul ZERO va deveni 0) dacă bitul al 5-lea din ax are valoarea 1.
Dezavantajul abordării de mai sus este acela că instrucţiunea *and* modifică valoarea primului operand, plasind acolo rezultatul operaţiei.
Instrucţiunea *test* are acelaşi efect ca şi *and* (execută AND intre biţii celor doi operanzi, modifică la fel indicatorul ZERO), dar *nu alterează valoarea primului operand*. De exemplu:

```
test ax, 0x0010 // binar: 0000000000010000

modifică indicatorul ZERO ca şi
and ax, 0x0010
fără a altera valoarea din ax.
```

Asm 2
Instrucţiunea mul – înmulţire de numere fără semn
Sintaxa: *mul op*

Efect: *destinatie_implicita = operand_implicit * op*
Operaţia de înmulţire este o operaţie binară. Din moment ce la instrucţiunea *mul* se precizează un singur operand, este evident că celalalt operand va fi implicit. Operandul implicit depinde de dimensiunea operandului explicit *op* (după cum ştim şi de la celelalte instrucţiuni studiate, operanzii trebuie să aibă aceeaşi dimensiune). În tabelul de mai jos sunt precizaţi operanzii impliciţi pentru fiecare din dimensiunile posibile ale operandului explicit.
În plus, trebuie observat faptul că reprezentarea rezultatului operaţiei de înmulţire poate avea lungime dublă faţă de lungimea operanzilor. De exemplu, înmulţind următoarele 2 numere reprezentate pe câte 8 biţi, obţinem un rezultat reprezentat pe 16 biţi:

```
       10110111 *
       11010010
----------------
              0
      10110111
    10110111
  10110111
 10110111
----------------
1001011000011110
```

Deci dimensiunea destinaţiei implicite trebuie să fie dublul dimensiunii operanzilor.
Tabelul de mai jos prezintă operanzii impliciţi şi destinaţiile implicite pentru diversele dimensiuni ale operandului implicit:

Dimensiune operand explicit	Operand implicit	Destinaţie implicită
1 octet	al	ax
2 octeţi	ax	(dx, ax)
4 octeţi	eax	(edx, eax)

Operandul explicit *nu poate fi constantă numerică*:

```
mul 10;         //EROARE
mul byte ptr 2;  //EROARE
```

El trebuie să fie ori un *registru de date* (al, ebx, ...), ori o *adresă de memorie*. Dacă adresa de memorie nu este dată prin numele simbolic (de exemplu, numele unei variabile declarate în programul C ce încapsulează codul asm), ci prin modurile de adresare discutate anterior, trebuie precizată dimensiunea în octeţi a zonei de

memorie ce conţine respectivul operand explicit, pentru a se putea stabili operandul implicit şi destinaţia implicită.
De exemplu:

- *mul byte ptr [ebp - 4]*: operandul explicit se află în memorie la adresa [ebp - 4] şi are dimensiunea de 1 octet (valoarea efectivă este cea din octetul de la aceasta adresa)
- *mul word ptr [ebp - 4]*: operandul explicit se află la adresa [ebp - 4] şi are dimensiunea de 2 octeţi (valoarea efectivă este cea compusă din primii 2 octeţi de la aceasta adresa)
- *mul dword ptr [ebp - 4]*: operandul explicit se află la adresa [ebp - 4] şi are dimensiunea de 4 octeţi (valoarea efectivă este cea compusă din primii 4 octeţi de la aceasta adresa)

Câteva *exemple*:

- linia de cod *mul bl* va avea urmatorul efect:
  - se calculează rezultatul înmulţirii dintre *al* şi *bl* (*bl* are dimensiunea de 1 octet, deci operandul implicit la înmulţire este *al*)
  - acest rezultat se pune în *ax*(care este destinaţia implicită pentru înmulţiri cu operandul explicit *op* de 1 octet

Mai concret, *mul bl <=> ax = al * bl*

- linia de cod *mul bx* va avea urmatorul efect:
  - se calculează rezultatul înmulţirii dintre *ax* şi *bx* (*bx* are dimensiunea de 2 octeţi, deci operandul implicit la înmulţire este *ax*)
  - acest rezultat se pune în *(dx,ax)* astfel: primii 2 octeţi (cei mai semnificativi) din rezultat vor fi plasaţi în *dx*, iar ultimii 2 octeţi (cei mai puţin semnificativi) în *ax*
  - rezultatul înmulţirii este, de fapt, $dx*2^{16} + ax$

Desigur, acest rezultat pe 4 octeţi ar fi încăput în *eax*. Se folosesc însă regiştrii *(dx,ax)* pentru compatibilitatea cu maşinile mai vechi, pe 16 biţi.
Exemplu de cod:

```
#include <stdio.h>

void main(){
  _asm {
    mov ax, 60000; //in baza 16: EA60
    mov bx, 60000; //in baza 16: EA60
    mul bx;        //rezultatul inmultirii este 3600000000;
                   //in baza 16: D693A400, plasat astfel:
                   //in registrul dx - partea cea mai semnificativa: D693
                   //in registrul ax - partea cea mai putin semnificativa: A400
```

```
  }
}
```

- linia de cod *mul ebx* va avea urmatorul efect:
  - se calculează rezultatul înmulţirii dintre *eax* şi *ebx* (*ebx* are dimensiunea de 4 octeţi, deci operandul implicit la înmulţire este *eax*)
  - acest rezultat se pune în *(edx,eax)* astfel: primii 4 octeţi (cei mai semnificativi) din rezultat vor fi plasaţi în *edx*, iar ultimii 4 octeţi (cei mai puţin semnificativi) în *eax*
  - rezultatul înmulţirii este, de fapt, $edx*2^{32} + eax$

Exemplu de cod:

```
#include <stdio.h>

void main(){
  _asm {
    mov eax, 60000; //in baza 16: 0000EA60
    mov ebx, 60000; //in baza 16: 0000EA60
    mul ebx;        //rezultatul inmultirii este 3600000000;
                    //in baza 16: D693A400, plasat astfel:
                    //in edx - partea cea mai semnificativa: 00000000
                    //in eax - partea cea mai putin semnificativa: D693A400

  }
}
```

Instrucţiunea imul – înmulţire de numere cu semn (Integer MULtiply)

Sintaxa: *imul op*

După cum am precizat mai sus, la instrucţiunea *mul* operanzii sunt consideraţi numere fără semn. Aceasta înseamnă că se lucrează cu numere pozitive, iar bitul cel mai semnificativ din reprezentare este prima cifră a reprezentării binare, nu bitul de semn.

Pentru operaţii de înmulţire care implică numere negative există instrucţiunea *imul* (este nevoie de două instrucţiuni distincte deoarece, spre deosebire de adunare sau scădere, agoritmul de înmulţire este diferit la numerele cu semn). Ceea s-a prezentat la *mul* este valabil şi pentru *imul*. Diferenţa este aceea că numerele care au bitul cel mai semnificativ 1 sunt considerate numere negative, reprezentate în complement faţă de 2.

Exemplu:

```
void main(){
  _asm {
    mov ax, 0xFFFF;
    mov bx, 0xFFFE;
```

```
    mul bx;        //rezultatul inmultirii numerelor FARA SEMN:
                   //65535 * 65534 = 4294770690;
                   //in baza 16: FFFD0002, plasat astfel:
                   //in dx - partea cea mai semnificativa: FFFD
                   //in ax - partea cea mai putin semnificativa: 0002

    mov ax, 0xFFFF;
    mov bx, 0xFFFE;
    imul bx;       //rezultatul inmultirii numerelor CU SEMN:
                   //-1 * -2 = 2;
                   //in baza 16: 00000002, plasat astfel:
                   //in dx - partea cea mai semnificativa: 0000
                   //in ax - partea cea mai putin semnificativa: 0002

  }
}
```

Exerciţiu:

Fie următorul program care calculează factorialul unui număr. Să se înlocuiască linia de cod din interiorul buclei *for* (f = f * i) cu un bloc de cod asm, cu obţinerea aceluiaşi efect. Pentru simplificare, vom considera că rezultatul nu depăşeşte 4 octeţi.

```
#include <stdio.h>

void main(){
  unsigned int n = 10, i, f = 1;
  for(i=1;i<=n;i++){
    f = f * i;
  }
  printf("%u\n",f);
}
```

Împărţirea

Instrucţiunea div – împărţire de numere fără semn

Sintaxa: *div op*

Efect: *cat_implicit, rest_implicit = deimpartit_implicit : op*

Instrucţiunea *div* corespunde operaţiei de împărţire cu rest.

Ca şi la înmulţire, operandul implicit (deîmpărţitul) şi destinaţia implicită (câtul şi restul) depind de dimensiunea operandului explicit *op* (împărţitorul):

Dimensiune operand explicit (împărţitor)	Deîmpărţit	Cât	Rest

1 octet	ax	al	ah
2 octeţi	(dx, ax)	ax	dx
4 octeţi	(edx, eax)	eax	edx

(A se observa similaritatea cu instrucţiunea de înmulţire.)
În toate cazurile, câtul este depus în jumătatea cea mai puţin semnificativă a deîmpărţitului, iar restul în cea mai semnificativă. Acest mod de plasare a rezultatelor permite reluarea operaţiei de nmpărţire nn buclă, dacă este cazul, fără a mai fi nevoie de operaţii de transfer suplimentare.
Analog cu înmulţirea, operandul explicit (împărţitorul) poate fi un registru sau o locaţie 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
```

Operaţia de împărţire ridică o problemă care nu se întâlneşte în alte părţi: împărţirea la 0:

```
#include <stdio.h>

void main(){
  _asm {
    mov eax, 1
    mov edx, 1
    mov ebx, 0
    div ebx
  }
}
```

Programul va semnala o eroare la execuţie (integer divide by zero) şi va fi terminat forţat.
Efectuând următoarea modificare:

```
#include <stdio.h>

void main(){
  _asm {
    mov eax, 1
    mov edx, 1
    mov ebx, 1  //1 in loc de 0
```

```
    div ebx
  }
}
```
se obţine o altă eroare la execuţie: integer overflow.
Motivul este acela că se încearcă împărţirea numărului 0x100000001 la 1, câtul fiind 0x100000001. Acest cât trebuie depus în registrul eax, însă valoarea lui depăşeşte valoarea maximă ce poate fi pusă în acest registru, adică 0xFFFFFFFF.
Mai concret, în cazul în care câtul nu încape în registrul corespunzător, se obţine eroare:
$(edx*2^{32} + eax) / ebx \geq 2^{32} <=>$
$edx*2^{32} + eax \geq ebx*2^{32} <=>$
$eax \geq (ebx - edx) * 2^{32} <=>$
$ebx \leq edx$
Cu alte cuvinte, vom obţine cu siguranţă eroare dacă împărţitorul este mai mic sau egal cu partea cea mai semnificativă a deîmpărţitului. Pentru a evita terminarea forţată a programului, trebuie verificată această situaţie înainte de efectuarea împărţirii.
Instrucţiunea idiv – împărţire de numere cu semn
Sintaxa: *idiv op*
idiv funcţionează ca şi *div*, cu diferenţa că numerele care au bitul cel mai semnificativ 1 sunt considerate numere negative, reprezentate în complement faţă de 2.
Exemple de cod

```
#include <stdio.h>

void main(){
  _asm {
    mov ax, 35;
    mov dx, 0;  //nu trebuie uitata initializarea lui (e)dx!
                //(in general, initializarea partii celei mai
                //  semnificative a deimpartitului)
    mov bx, 7;
    div bx;    //rezultat: ax devine 5, adica 0x0005 (catul)
               //          dx devine 0 (restul)


    mov ax, 35;
    mov dx, 0;
    mov bx,7
    idiv bx     // acelasi efect, deoarece numerele sunt pozitive
```

```
   mov ax, -35; //in hexa (complement fata de 2): FFDD
   mov dx, 0;
   mov bx,7
   div bx      //deimpartitul este (dx, ax), adica 0000FFDD
               //in baza 10: 65501
               //rezultat: ax devine 0x332C, adica 13100 (catul)
               //          dx devine 0x0001  (restul)

   mov ax, -35; //in hexa (complement fata de 2): FFDD
   mov dx, 0;
   mov bx,7
   idiv bx     //deimpartitul este (dx, ax), adica 0000FFDD
               //este un mumar pozitiv, adica, in baza 10, 65501
               //rezultat: ax devine 0x332C, adica 13100 (catul)
               //          dx devine 0x0001  (restul)
               //(efectul este acelasi ca la secventa de mai sus)

   mov ax, -35; //in hexa (complement fata de 2): FFDD
   mov dx, -1;  //in hexa (complement fata de 2): FFFF
   mov bx,7
   idiv bx     //deimpartitul este (dx, ax), adica FFFFFFDD
               // - numar negativ, reprezentat in complement fata de 2
               //in baza 10: -35
               //rezultat: ax devine 0xFFF9, adica -5 (catul)
               //          dx devine 0  (restul)

   mov ax, -35; //in hexa (complement fata de 2): FFDD
   mov dx, -1;  //in hexa (complement fata de 2): FFFF
   mov bx,7
   div bx      //deimpartitul este (dx, ax), adica FFFFFFDD
               // - numar pozitiv (deoarece folosim div)
               // in baza 10: 4294967261
               //rezultat: EROARE, deoarece FFFF > 0007,
               //   catul (613566751, adica 2492491F) nu incape in ax

  }
}
```

Exerciţiu

Fie următorul program. Să se înlocuiască liniile 4 şi 5 cu un bloc de cod asm, cu obţinerea aceluiaşi efect.

```
1. #include <stdio.h>

2. void main(){
3.   unsigned a=500007,b=10,c,d;
4.   c=a/b;
5.   d=a%b;
6.   printf("%u %u\n",c,d);

7.}
```

Instrucţiuni de deplasare

Sunt instrucţiuni care permit deplasarea biţilor în cadrul operanzilor cu un număr precizat de poziţii.

Deplasările pot fi aritmetice sau logice. Deplasările aritmetice pot fi utilizate pentru a înmulţi sau împărţi numere prin puteri ale lui 2. Deplasările logice pot fi utilizate pentru a izola biţi în octeţi sau cuvinte.

Dintre modificările pe care deplasările le fac asupra indicatorilor:

- Carry Flag (CF) = ultimul bit deplasat în afara operandului destinaţie;
- Sign Flag (SF) = bitul cel mai semnificativ din operandul destinaţie;
- Zero Flag (ZF) = 1 dacă operandul destinaţie devine 0, 0 altfel.

O ilustrare a deplasării logice la stânga cu o poziţie:

Instrucţiunile de deplasare sunt:

- *shr* dest, count
- *shl* dest, count
- *sar* dest, count
- *sal* dest, count

unde:

- *dest* semnifică destinaţia a cărei valoare va fi modificată; poate fi *registru* sau *locaţie de memorie*:
  - *shl* eax, 1
  - *shl* dx, 3
  - *shl* byte ptr [...], 2
- *count* precizează cu cîte poziţii se face deplasarea; poate fi *constantă numerică* sau registrul *cl*:
  - *shl* ebx, cl

Instrucţiunea shr (SHift Right)

Sintaxa: *shr dest, count*

Efect: deplasarea la dreapta a biţilor din *dest* cu numărul de poziţii precizat de *count*; completarea la stânga cu 0; plasarea în Carry a ultimului bit ieşit.
Exemplu:

```
mov bl, 33; //binar: 00100001
shr bl, 3;  //bl devine 00000100
        //Carry devine 0
shr bl, 3   //bl devine 00000000
        //Carry devine 1
```

Instrucţiunea shl (SHift Left)
Sintaxa: *shl dest, count*
Efect: deplasarea la stânga a biţilor din *dest* cu numărul de poziţii precizat de *count*; completarea la dreapta cu 0; plasarea în Carry a ultimului bit ieşit.
Exemplu:

```
mov bl, 33; //binar: 00100001
shl bl, 3;  //bl devine 00001000
        //Carry devine 1
shl bl, 1   //bl devine 00010000
        //Carry devine 0
```

Instrucţiunea sar (Shift Arithmetic Right)
Sintaxa: *sar dest, count*
Efect: deplasarea la dreapta a biţilor din *dest* cu numărul de poziţii precizat de *count*; bitul cel mai semnificativ îşi păstrează vechea valoare, dar este şi deplasat spre dreapta (extensie de semn); plasarea în Carry a ultimului bit ieşit.
Exemplu:

```
mov bl, -36; //binar: 11011100
sar bl, 2;   //bl devine 11110111
        //Carry devine 0
```

Trebuie menţionat că *sar* nu furnizează aceeaşi valoare ca şi *idiv* pentru operanzi echivalenţi, deoarece *idiv* trunchiază toate câturile către 0, în timp ce *sar* trunchiază câturile pozitive către 0 iar pe cele negative către infinit negativ.
Exemplu

```
mov ah, -7; //binar: 11111001
sar ah, 1;  //teoretic, echivalent cu impartirea la 2
        //rezultat: 11111100, adica -4
        //idiv obtine catul -3
```

Instrucţiunea sal (Shift Arithmetic Left)
Sintaxa: *sal dest, count*
Efect: identic cu *shl*.

Asm 3
EFLAGS

Este un registru de 32 de biti care indica "starea" procesorului la un moment dat. Doar o parte din cei 32 de biti sunt folositi pentru a furniza informatii despre rezultatul ultimei operatii executate de procesor. Bitii din EFLAGS se mai numesc si indicatori. Dintre acestia, amintim:

CF - carry flag (transport) - are valoarea 1 (este setat) daca dupa ultima operatie a aparut transport, 0 (nu este setat) altfel.
PF - parity flag (paritate) - are valoarea 1, daca numarul de biti de 1 din rezultatul ultimei operatii este par.
ZF - zero flag - are valoarea 1, daca rezultatul ultimei operatii a fost 0.
SF - sign flag (semn) - are valoarea 1, daca rezultatul ultimei operatii a fost negativ (bitul cel mai semnificativ este 1).
OF - overflow flag (depasire) - are valoarea 1, daca ultima operatie a produs depasire aritmetica.
Instrucţiuni de salt
Instrucţiunile de salt modifică valoarea registrului contor program (EIP), astfel încât următoarea instrucţiune care se execută să nu fie neapărat cea care urmează în memorie. Sunt utile pentru implementarea, la nivel de limbaj de asamblare, a structurilor de control (testări sau bucle).
Salturile pot fi:

- necondiţionate: instrucţiunea *jmp*
- condiţionate: instrucţiuni de forma *j<condiţie>*

Sintaxa: *instructiune_salt adresa*
Vom considera în continuare doar cazul în care adresa este o constantă referită de o etichetă.
Exemplul de mai jos ilustrază modul de definire şi utilizare a etichetelor:

```
#include <stdio.h>

void main(){
  int i;
  _asm{
    mov i, 11;
    jmp eticheta;
```

```
   sub i, 3;     // aceasta instructiune nu se executa
  eticheta:
   add i, 4;
  }
  printf ("%d\n", i);
}
```

Saltul necondiţionat (instrucţiunea jmp)

Nu introduce o ramificaţie în program, neexistînd variante de execuţie. Este util, folosit împreună cu salturi condiţionate, pentru reluarea unei secvenţe de de cod într-o buclă, aşa cum se va vedea într-un exemplu ulterior.

Salturi condiţionate

Introduc o ramificaţie în program, deoarece avem două variante:

- condiţia de salt este adevărată – se face saltul la adresa indicată
- condiţia de salt este falsă – se continuă cu instrucţiunea următoare din memorie ca şi cum nu ar fi existat instrucţiune de salt.

Instrucţiuni care testează indicatorii individuali

Cele mai utile la acest nivel sunt cele care testează indicatorii: Carry, Overflow, Sign, Zero. Pentru fiecare indicator există două instrucţiuni de salt condiţionat: una care face saltul când indicatorul testat are valoarea 1 şi una care face saltul când are valoarea 0.

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

Exemplu:

```
#include <stdio.h>

void main(){
  int a, b, s=0;
  printf("a=");
  scanf("%x", &a);
  printf("b=");
  scanf("%x", &b);

  _asm{
    mov eax, a;
    add eax, b;
    jc semnaleaza_depasire; //in Visual C++,
```

```
                    // putem sari la o eticheta din codul C
    mov s, eax;
    jmp afiseaza_suma;      //sau din alt bloc asm

  }
semnaleaza_depasire:
  printf ("S-a produs depasire!\n");
  return;
  _asm{
    afiseaza_suma:
  }
  printf ("%x + %x = %x\n", a, b, s);
}
```

Instrucţiuni corespunzătoare operatorilor relaţionali

În practică, utilizăm mai des ramificări dictate de operatori relaţionali: <, <=, !=, etc. În acest sens este utilă instrucţiunea de comparare *cmp*:

cmp funcţionează ca şi *sub* (aceleaşi restricţii pentru operanzi, *aceiaşi indicatori modificaţi*), însă nu modifică primul operand (destinaţia). Prin verificarea indicatorilor se poate stabili în urma aceste operaţii relaţia dintre operanzi.

Instrucţiunile care fac aceste verificări sunt:

relaţie	instrucţiune	Comentariu
op1 < op2	jb	"Jump if Below"
op1 <= op2	jbe	"Jump if Below or Equal"
op1 > op2	ja	"Jump if Above"
op1 >= op2	jae	"Jump if Above or Equal"

pentru numere *fără semn*, respectiv

relaţie	instrucţiune	Comentariu
op1 < op2	jl	"Jump if Less than"
op1 <= op2	jle	"Jump if Less than or Equal"
op1 > op2	jg	"Jump if Greater than"
op1 >= op2	jge	"Jump if Greater than or Equal"

pentru numere *cu semn*.

Sunt necesare instrucţiuni diferite pentru numere fără semn, respectiv cu semn, deoarece indicatorii ce trebuie verificaţi diferă. De exemplu, comparând 00100110 şi 11001101, ar trebui să obţinem relaţia 00100110 < 11001101 dacă sunt numere fără semn, şi 00100110 > 11001101 dacă sunt numere cu semn.

Independent de statutul bitului celui mai semnificativ (semn sau cifră) funcţionează instrucţiunile:

relaţie	instrucţiune	Comentariu
op1 == op2	je	"Jump if Equal" (identic cu jz)
op1 != op2	jne	"Jump if Not Equal" (identic cu jnz)

Asm 4
Lucrul cu stiva. Apelul funcţilor
Lucrul cu stiva
Procesorul foloseşte o parte din memoria RAM pentru a o accesa după o disciplină de tip LIFO (ca în cazul unei structuri de stivă). După cum se ştie, singura informaţie fundamentală pentru gestiunea stivei este *vârful* acesteia. În cazul procesorului, adresa la care se află vârful stivei este memorată în perechea de regiştri *SS* şi *ESP*; deoarece regiştrii segment au, pe maşinile pe 32 de biţi, doar scop de "validare" a adresei, vom lucra în continuare numai cu ESP.
Instrucţiunile care permit lucrul cu stiva sunt *push* şi *pop*.
Instrucţiunea push
Realizează *introducerea* unei valori în stivă.
Sintaxa: *push* operand;
Operandul poate fi registru, locaţie de memorie sau constantă numerică. Stiva lucrează doar cu valori de 2 sau 4 octeţi, pentru uniformitate preferându-se numai operanzi de 4 octeţi (varianta cu 2 se păstrază pentru compatibilitate cu procesoarele mai vechi).
Exemple:
push eax
push dx
push dword ptr [...]
push word ptr [...]
push dword ptr 5
push word ptr 14
Introducerea valorii în stivă se face astfel: se *scade* din ESP dimensiunea, în octeţi, a valorii care se vrea depusa în stivă, dupa care procesorul *scrie* valoarea operandului la adresa indicată de registrul ESP (vârful stivei); dimensiunea poate fi 2 sau 4 (se observă că se avansează "în jos", de la adresele mai mari la adresele mai mici); în acest mod, vârful stivei este pregătit pentru următoarea operaţie de scriere.
De exemplu, instrucţiunea
push eax;
ar fi echivalentă cu:

sub esp, 4;
mov [esp], eax;
Prin folosirea lui *push* în locul secvenţei echivalente se reduce, însă, riscul erorilor.
Instrucţiunea pop
Extrage vârful stivei într-un operand destinaţie.
Sintaxa: *pop* operand;
Operandul poate fi registru sau locaţie de memorie, de 2 sau 4 octeţi.
Exemple:
pop eax
pop cx
pop dword ptr [...]
pop word ptr [...]
Extragerea valorii din stivă se face prin depunerea în destinaţie a valorii aflate în vârful stivei (la adresa [ESP]) şi adunarea, la ESP, a numărului de octeţi ai operandului (acesta indică, practic, numărul de octeţi scoşi din stivă).
Rolul stivei
Rolul stivei procesorului este acela de a stoca informaţii cu caracter temporar. De exemplu, dacă avem nevoie să folosim un registru pentru nişte operaţii, dar nu avem la dispoziţie nici un registru a cărui valoare curentă să ne permitem să o pierdem, putem proceda ca mai jos:
push eax; //se salveaza temporar valoarea lui eax pe stiva
... // utilizare eax
pop eax //restaurare
Variabilele locale (cu excepţia celor statice) sunt plasate de asemenea în stivă (deoarece au caracter temporar: sunt create la intrarea în funcţie şi distruse la ieşire).
În lucrul cu stiva, *instrucţiunile de introducere în stivă trebuie riguros compensate de cele de scoatere*, din punctul de vedere al numărului de instrucţiuni şi al dimensiunii operanzilor. Orice eroare poate afecta mai multe date, din cauza decalajelor.
push edx;
push eax;
... //utilizare registri
pop ax //se recupereaza doar 2 octeti din valoarea anterioara a lui eax
pop edx //nu se recupereaza edx, ci 2 octeti din eax, 2 din edx
  //decalajul se poate propaga astfel pana la capatul stivei
O altă eroare poate apărea atunci când registrul ESP este manipulat direct. De exemplu, pentru a aloca spaţiu unei variabile locale (neiniţializată), e suficient a scădea din ESP dimensiunea variabilei respective. Similar, la distrugerea variabilei, valoarea ESP este crescută. Aici nu se folosesc în general instrucţiuni

push, repectiv pop, deoarece nu interesează valorile implicate, ci doar ocuparea şi eliberarea de spaţiu. Se preferă adunarea şi scăderea direct cu registrul ESP; evident că o eroare în aceste operaţii are consecinţe de aceeaşi natură ca şi cele de mai sus.

Apelul funcţiilor

Un apel de funcţie arată la prima vedere ca o instrucţiune de salt, în sensul că se întrerupe execuţia liniară a programului şi se sare la o altă adresă. Diferenţa fundamentală constă în faptul că la terminarea funcţiei se revine la adresa de unde s-a făcut apelul şi se continuă cu instrucţiunea următoare. Din moment ce într-un program se poate apela o funcţie de mai multe ori, din mai multe locuri, şi întotdeauna se revine unde trebuie, este clar că adresa la care trebuie revenit este memorată şi folosită atunci când este cazul. Cum adresa de revenire este în mod evident o informaţie temporară, locul său este tot pe stivă.

Instrucţiunea call

Apelul unei funcţii se realizează prin instrucţiunea *call*.

Sintaxa: *call* adresa

În Visual C++ vom folosi *nume simbolice* pentru a preciza adresa, cu menţiunea că de data asta nu este vorba de etichete, ca la salturi, ci chiar de numele funcţiilor apelate.

Efectul instrucţiunii *call*: se introduce în stivă adresa instrucţiunii următoare (*adresa de revenire*) şi se face *salt la adresa indicată*. Aceste acţiuni puteau fi realizate şi cu instrucţiuni push şi jmp, dar din nou se preferă call pentru evitarea erorilor.

Instrucţiunea ret

Revenirea dintr-o funcţie se face prin instrucţiunea *ret*, care poate fi folosită fără operand. În acest caz, se preia adresa de revenire din vârful stivei (similar unei instrucţiuni pop) şi se face saltul la adresa respectivă. Din motive de conlucrare cu Visual Studio, nu vom folosi această instrucţiune.

Transmiterea parametrilor

Parametrii sunt tot nişte variabile locale, deci se găsesc pe stivă. Cel care face apelul are responsabilitatea de a-i pune pe stivă la apel şi de a-i scoate de pe stivă le revenirea din funcţia apelată. Avem la dispoziţie instrucţiunea *push* pentru plasarea în stivă. Evident, această operaţie trebuie realizată imediat înainte de apelul propriu-zis. În plus, în limbajul C/C++ (nu în toate), parametrii trebuie puşi în stivă în ordine inversă celei în care se găsesc în lista de parametri. La revenire, parametrii trebuie scoşi din stivă, nemaifiind necesari. Cum nu ne interesează preluarea valorilor lor, nu se foloseşte instrucţiunea pop, care ar putea altera inutil un registru, de exemplu, ci se adună la ESP numărul total de octeţi ocupat de parametri (atenţie, pe stivă se lucrează în general cu 4 octeţi, chiar dacă operanzii au dimensiuni mai mici).

Să luăm ca exemplu funcţia următoare:

```
void show_dif(int a,int b){
  int c;
  c=a-b;
  printf("%d\n",c);
}
```

Apelul dif(5,9) se traduce prin secvenţa care se poate vedea mai jos:

```
void main(){
  _asm{
    push dword ptr 9
    push dword ptr 5
    call show_dif
    add esp,8
  }
}
```

Returnarea unei valori

Convenţia în Visual C++ (şi la majoritatea compilatoarelor) este că rezultatul se depune într-un anumit registru, în funcţie de dimensiunea sa:

- pentru tipurile de date de dimensiune 1 octet - în registrul AL
- pentru tipurile de date de dimensiune 2 octeţi - în registrul AX
- pentru tipurile de date de dimensiune 4 octeţi - în registrul EAX
- pentru tipurile de date de dimensiune 8 octeţi - în regiştii EDX şi EAX

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

Vom modifica exemplul de mai sus astfel încât funcţia să returneze diferenţa pe care o calculează într-o secvenţă de instrucţiuni în limbaj de asamblare:

```
#include <stdio.h>

int compute_dif(int a,int b){
  _asm{

    mov eax, a;
    sub eax, b;
            //in eax ramane rezultatul, care
            //  va fi preluat la termiarea functiei
  };
}

void main(){
  int c;
```

```
  _asm{
    push dword ptr 9
    push dword ptr 5
    call compute_dif  //se salveaza adresa de revenire pe stiva
    mov c, eax;
    add esp,8         //"stergerea" parametrilor din stiva
  }
  printf("Diferenta este %d.\n", c);
}
```

Parametri

Exista o alta modalitate de accesa in cadrul unei functii parametrii acesteia in cadrul unui bloc limbaj de asamblare. Ei se gasesc pe stiva incepand cu adresa [ebp+8] si ocupa numarul de octeti al tipului de date respectiv.

Exemplu, o functie cu 3 parametri de tip int (o variabila de tip int are 4 octeti):

```
void functie(int a, int b, int c){
  _asm{

        mov eax, [ebp+8] // muta in eax valoarea lui a
        mov ebx, [ebp+12] // muta in ebx valoarea lui b
        mov ecx, [ebp+16] // muta in ecx valoarea lui c

  };
}
```

Scris in aceasta maniera, exemplul de mai sus ar arata in felul urmator:

```
#include <stdio.h>

int compute_dif(int ,int ){ // nu mai este nevoie sa punem nume variabilelor, deoarece vom lucra direct cu stiva
  _asm{

    mov eax, [ebp+8];
    sub eax, [ebp+12];
            //in eax ramane rezultatul, care
            //  va fi preluat la termiarea functiei
  };
}

void main(){
  int c;
  _asm{
```

```
   push dword ptr 9
   push dword ptr 5
   call compute_dif  //se salveaza adresa de revenire pe stiva
   mov c, eax;
   add esp,8         //"stergerea" parametrilor din stiva
  }
  printf("Diferenta este %d.\n", c);
}
```

Asm 5
Pointeri.

Pentru a intelege cum se folosesc tablourile in ASM, trebuie inteles mai intai conceptul de pointer. Pointer-ul reprezinta o variabila ce pastreaza o adresa de memorie a unei date ("pointeaza" spre o adresa de memorie). Un pointer poate fi utilizat pentru referirea a diferite tipuri de date (tablouri de tip int, siruri de caracetere, matrici etc.) sau structuri de date. Schimband adresa memorata in pointer, pot fi manipulate informatii situate la diferite locatii de memorie.
Legatura dintre tablouri si pointeri

Numele unui tablou este un pointer constant spre primul sau element. Expresiile de mai jos sunt deci echivalente:
nume_tablou
&nume_tablou
&nume_tablou[0]
*nume_tablou
nume_tablou[0]

// Ex.1 Interschimbarea a 2 valori

```
#include <stdio.h>

void swap (int *a, int *b)
{
        _asm{
                mov eax, a;       // punem in eax adresa data de pointerul *a
                mov ecx, [eax];   // punem in ecx valoarea efectiva a lui *a (valoarea
de la adresa pointerului)
                mov ebx, b;       // analog pt b
```

```
                mov edx, [ebx];
                mov [eax], edx;   // mutam la adresa lui a valoarea lui *b
                mov [ebx], ecx;   // analog invers
        }
}

void main()
{
        int a=2, b=3;
        swap(&a,&b);
        printf("%d %d", a, b);
}

// Ex.2 Suma elementelor dintr-un vector

#include <stdio.h>

int suma_vector (int *, int )
{
        _asm
        {
                mov eax, 0            // suma
                mov ebx, [ebp+8]  // primul parametru, pointer la vectorul de
elemente
                mov ecx, 0            // contor
bucla:
                cmp ecx, [ebp+12] // al 2-lea parametru, lungimea vectorului de
elemente
                jae stop
                add eax, [ebx+ecx*4]      // elementul vectorului de pe pozitia ecx
                inc ecx
                jmp bucla
stop:
        }
}

void main()
{
        int v[5]={5,1,2,3,6};
        int *p=v;
```

```
	int s;

	_asm{
		push 5
		push p
		call suma_vector
		add esp, 8
		mov s, eax
	}

	printf("Suma: %d", s);
}

// Ex.3 Lungimea unui sir de caractere (un sir de numere se termina cu valoarea 0)

#include <stdio.h>
int lungime(char *)
{
	_asm{
		mov eax, 0
		mov ebx, [ebp+8]  // adresa de inceput a sirului de caractere
bucla:
		cmp [ebx+eax], 0   // comparam caracterul curent cu 0
		je stop
		inc eax
		jmp bucla
stop:
	}
}

void main()
{
	char *sir="zigyzagy";
	int l;

	_asm{
		push sir
		call lungime
		add esp, 4
		mov l, eax
```

```
	}

	printf("Lungime: %d %d\n", l, strlen(sir));
}
```

Pentru matrice de a[n][m] (n x m elemente), pentru avea acces la elementul de pe pozitia [i][j] (linia i, coloana j), va trebui sa aflam adresa acestuia. "a[i][j]" este echivalent cu: "&a + (i*m+j)*4" (adresa primului element la care adaugam i x nr_coloane + j, totul inmultit cu dimensiunea elementelor, in cazul nostru 4 octeti pentru int)

```
// Ex. 4 - Construirea matricii unitate (1 pe diagonala, 0 in rest)

#include <stdio.h>

void matrice_unitate(int *, int )
{
	_asm{
		mov edi, [ebp+8]		// adresa la primul element din matrice
		mov ebx, 0
for_i:
		cmp ebx, [ebp+12]		// dimensiunea matricii
		jae exit1

		mov ecx, 0
for_j:

		cmp ecx, [ebp+12]
		jae exit2

		mov eax, [ebp+12]		// construim adresa de pe pozitia [i][j]
		mul ebx
		add eax, ecx

		cmp ebx, ecx
		jne not_eq
		mov dword ptr [edi+eax*4], 1	// i == j, deci vom pune 1
		jmp inside
not_eq:
		mov dword ptr [edi+eax*4], 0	// altfel, 0
inside:
```

```
                inc ecx
                jmp for_j
exit2:
                inc ebx
                jmp for_i
exit1:
        }
}

void main()
{
        int n=5;
        int mat[5][5];
        int *p = mat[0];

        _asm
        {
                push n
                push p
                call matrice_unitate
                add esp, 8
        }

        for(int i=0; i<n; i++)
        {
                for(int j=0; j<n; j++)
                        printf("%d ", mat[i][j]);
                printf ("\n");
        }
}
```