asm2

MUL - inmultirea numerelor fara 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*:

Operandul explicit *nu poate fi constanta numerica*.


```
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
  }
}
```

IMUL - inmultirea numerelor 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);
}
```

DIV - impartirea numerelor fara 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 impărţire in 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 {
```

```
  _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 ≥ 2^32 <=>
edx*2^32 + eax ≥ ebx*2^32 <=>
eax ≥ (ebx - edx) * 2^32 <=>
ebx ≤ 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.

IDIV - impartirea numerelor 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;
!ⓘ   d=a%b;
6.   printf("%u %u\n",c,d);
```

Instructiuni 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.

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

SHR (SHift Right) - deplasare la dreapta

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 CF (Carry Flag) 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
```

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 CF (Carry Flag) 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
```

SAR (Shift Arithmetic Right) - deplasare aritmetica la dreapta

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
```

SAL - (Shift Arithmetic Left) - deplasare aritmetica la stanga

Sintaxa: *sal dest, count*

Efect: identic cu *shl*.