

Laborator 0x02

1. Operații aritmetice 2 :

- MUL

- DIV

2. Salturi necondiționate și salturi condiționate

3. Implementarea structurilor repetitive

4. Instrucțiunea LOOP

5. Tablouri unidimensionale

Operații aritmetice

mul op

div op

Obs mul și div au o destinație și o sursă implicită

mul op

(edx, eax) = eax * op

unde edx este completat când se depășește 2^{32}

$$2^{32} \cdot \text{edx} + \text{eax}$$

$$\text{eax} = \text{eax} * \text{op}$$

Exemplu

mov \$9, %eax
mul %ebx

} %eax = 9 * %ebx

$$\text{mul \%eax} \Rightarrow \text{eax}^2$$

$$\text{// \%eax} = \%eax + \%eax$$

div op

$$(\%edx, \%eax) = \%eax / \text{op}$$

restul
împărțirii

cățul
împărțirii

Sol: Exemple de program

. data

x: . long 30

y: . long 7

prod: . space 4

cat: . space 4

rest: . space 4

. text

. global main

main:

mov x, %eax

mov y, %ebx

mul %ebx // %eax := %eax * %ebx = $x * y$

mov %eax, prod

mov x, %eax

div %ebx // (%edx, %eax) := %eax / %ebx = x / y

mov %edx, rest

mov %eax, cat

__exit

mov \$1, %eax

mov \$0, %ebx

int \$0x80




Salteni neconditionate si salteni conditionate

Salteni neconditionate

`jmp et` - sare la eticheta et

Salteni conditionate

- evaluate prin `cmp op2, op1`


< `jl et`

> `jg et`

≤ `jle et`

≥ `jge et`

= `je et`

!= `jne et`

`cmp %eax, %ebx`

`jle et`

// `%ebx ≤ %eax`

Exemple de program

. data

x: .long 5

y: .long 8

str1: .ascii "x < y\n"

str2: .ascii "x > y\n"

. text

. global main

main:

mov x, %eax

mov y, %ebx

if (ebx >= eax)
jmp et_afis

{
cmp %eax, %ebx
jg et_afis1
jmp et_afis2
}

et_afis1:

mov \$4, %eax

mov \$1, %ebx

mov \$str1, %ecx

mov \$7, %edx

int \$0x80

jmp et_exit

et_afis2:

mov \$4, %eax

mov \$1, %ebx

mov \$str2, %ecx

mov \$8, %edx

int \$0x80

et_exit:

mov \$1, %eax

mov \$0, %ebx

int \$0x80

Simulation structure repetitive

```
int sum = 0           ↗ exit cond i = n
for ( int i = 0; i < n; i++)
{
    sum += i
}
```

Sol:

```
mov $0, %eax          # le point de "i"
mov $0, %ecx
```

et_for :

```
cmp %eax, %ecx
```

```
je et_exit
```

```
add %eax, %ecx
```

```
add $1, %ecx          // inc %ecx
```

```
jmp et_for
```

et_exit

```
mov $1, %ecx
```

```
mov $0, %ebx
```

```
int $0x80
```



Instrucțiunea LOOP

- lucrează cu reg. %ecx prin decrementarea lui

mov \$10, %ecx

et_for :

...
loop et_for

↙
verifică dacă %ecx == 0, și dacă da,

merge la linia următoare

altfel

decrementează %ecx

revine la et_for

Tablouri unidimensionale de date

- pentru întregi -

15	27	3	11	17	5
----	----	---	----	----	---

↑
v

- elementele sunt linear așezate în memorie
- v reprezintă un nume pt. adresa de început

long v[6]

I $v + 4 * 0$

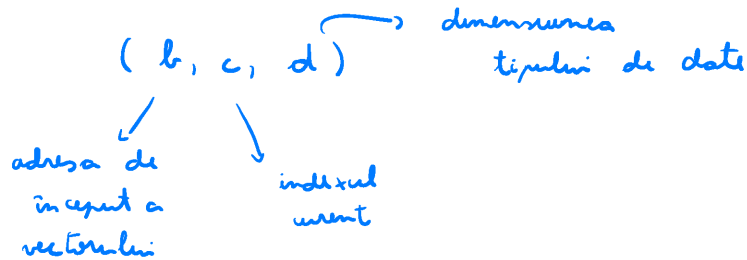
II $v + 4 * 1$

III $v + 4 * 2$

$a(b, c, d) \Rightarrow b + c * d + \underline{a}$ \rightarrow offset de deplasare

i lipseste în general

(mai ales la vectori)



lea = Load
effective
address

lea v, %edi

(%edi, %ecx, 4) \rightarrow dim. tipului
long

adresa
vectorului

se incrementează
la fiecare pas

Declarații în memorie

v: long 15, 27, 3, 11, 17, 5

ex

Suma elementelor dintr-un array

```
for ( %ecx = 0; %ecx < n; %ecx++)  
{  
    %ecx += v[%ecx]  
}
```


. data

v: .long 10, 15, 30

n: .long 3

. text

. global main

main:

lea v, %edi

// %edi retine adresa de inciput a array-ului

mov \$0, %ecx

mov \$0, %eax

et_for:

cmp %ecx, n

je et_out

mov (%edi, %ecx, 4), %ebx

add %ebx, %eax

inc %ecx

jmp et_for

et_out:

mov \$1, %eax

mov \$0, %ebx

int \$0x80

