Laborator 0x01

1. Limbaje de asamblare
2. Elemente de programare în x86
3. Regiștii
4. Declararea variabilelor în memorie
5. Instrucțiunea MOV
6. Apeluri de sistem :
   - EXIT
   - WRITE
7. Un prim program în x86
8. Debug
9. Operații aritmetice :
   - ADD
   - SUB

Limbaje de asamblare

- limbajul Intel x 86

Intel — AT&T

Windows — Linux, Mac OS } Unix

- arhitecturi de procesor:

RISC

Reduced Instruction Set Computer

- Software
- instrucțiuni cu lungime fixă
- accesează memoria și o încarcă obligatoriu în regiștrii

CISC

Complex Instruction Set Computer

- Hardware
- instrucțiuni cu lungime variabilă
- putem lucra direct cu memoria

Limbajul de asamblare este o imagine a codului mașină

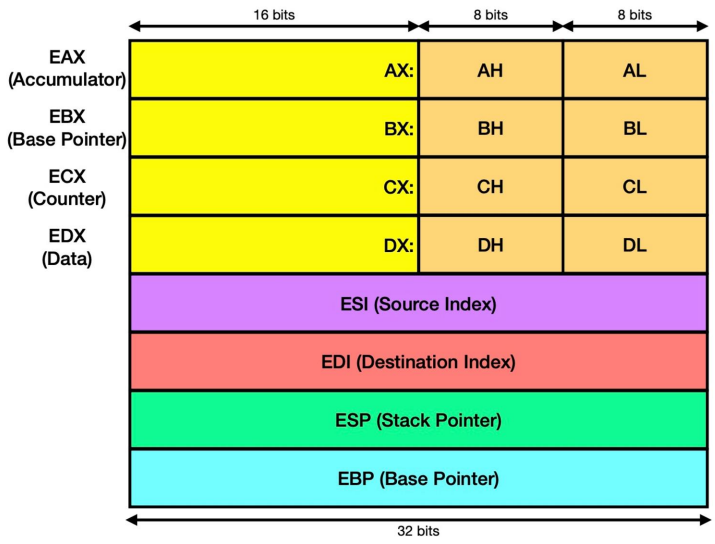
C (medium level)
↓
limbaj de asamblare x 86
↓
cod mașină

Elemente de programare în x86

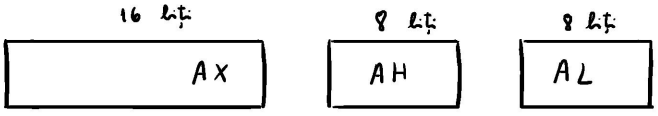
Regiștii

- spații pentru a stoca valori direct pe procesor
- „ variabile pentru procesor "
- sunt în număr limitat
- există convenții de utilizare pentru regiștii

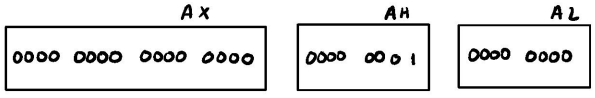
- eax → acumulator, utilizat în operații aritmetice
- ebx → base register, utilizat ca pointer la date
- ecx → counter, utilizat în cadrul instr. repetitive
- edx → data, utilizat în op. aritmetice, op. input / output
- esp → stack pointer // pointează vârful stivei
- ebp → base pointer // pointează baza stivei
- esi → source index
- edi → destination index



EAX



Exemplu



$2^8 = 256$

Notație & registru

$ const_numerice

Declararea variabilelor în memorie

nume : . tip_de_data valoare

Tipuri de date

1) . long - tip de date pentru stocarea intregilor pe 32 de biți (4 bytes)

```
x : . long 15
// long x = 15
```

2) . word - tip de date pentru stocarea întregilor pe 16 biți (2 bytes)

3) . space dim - tip de date pentru a crea un spațiu de dim bytes (octeți)

```
long v[100]   (=)  v : . space 400
!!
4B · 100 elemente = 400 bytes
```

6) byte - ocupa 1 byte (8 biți) în memorie

```
ch : . byte 'b'
```

4) .asciz - tip de date pentru declararea unui șir de caractere fără terminatorul de șir (nu are '\0')

5) .asciz - tip de date pentru declararea unui șir de caractere, dar care se termină cu '\0'

str: .asciz "Hello, world"

Instrucțiunea MOV

mov source, destination

destination ← source

Exemple

```
...
x: .long 15
...
    mov x, %eax
```

// %eax = x

Sufixul ugual ℓ (long) movl x, %eax

mov %ebx, %eax

// %eax = %ebx

mov %eax, x

// x = %eax

mov $5, %ecx ← constantă

// %ecx = 5

Apeluri sistem

return 0 = exit (0)

EXIT - întrerupere pentru încetarea execuției programului

exit (0) sau return 0

```
mov $1, %eax
mov $0, %ebx
int $0x80      ≡ syscall
```

%eax → codifică funcția care trebuie apelată

EXIT are codul 1

%ebx, %ecx, %edx sunt argumente pentru apelurile de sistem

WRITE - apel sistem pentru afișarea șirurilor de caractere

```
mov $4, %eax
mov $1, %ebx
mov $str, %ecx
mov $14, %edx
int $0x80
```

%eax = $4

%ebx = $1 standard output

%ecx = șirul de caractere

%edx = lungimea șirului

int $0x80 → syscall

%eax → 1 → EXIT

→ 4 → WRITE

Un prim program x86

```
.data
    // declararea variabilelor din memorie
    x: .long 15
    str: .asciz "Șir"
.text
    // marchează zona de început a instrucțiunilor în
    // limbaj de asamblare
.global main
    // definește "entry point"-ul
main:
    // instrucțiuni
```

Debug

```
gcc -m32 program.asm -o program
gdb program
b main          // breakpoint la eticheta main
run
i r             // interogate regiștri
step            // abia după step are loc o instrucțiune
```

Operații aritmetice

add op1, op2

calculează op2 = op1 + op2

Obs op2 nu poate fi const. numerică

sub op1, op2 - calculează op2 = op2 - op1

Exemplu de program

```
. data
    x: .long 15
    y: .long 5
    sum: .space 4
. text
. global main
    main
        mov x, %eax          // %eax = x
        mov y, %ebx          // %ebx = y
        add %ebx, %eax       // %eax = x + y
        mov %eax, sum
    et_exit
        mov $1, %eax
        mov $0, %ebx
        int $0x80
```

□