Registri:
= variabile specifice procesorului
- sunt spatii de stocare direct pe procesor, care este rapid accesabil
fata de variabilele din RAM

Intel x86 proc pe 32b => Registrii pot stoca pe 32b maxim

Registrii sunt finiti, limitati ca numar, si au nume specifice si o conventie
de utilizare.

Registru Conventie

%eax accumulator: registru acumulator, este utilizat in op. aritmetice
  si retine codurile apelurilor de sistem

%ebx base: utilizat in operatii aritmetice

%ecx counter: utilizat ca index in structurile repetitive

%edx data: utilizat ca pointer la date si in op aritmetice

%esi source index: sunt utilizati pt stocarea pointerilor*
%edi destination index: *la tablourile de date

%ebp base pointer: utilizati pt gestiunea stivei
%esp stack pointer:

%eip instruction pointer: retine mereu adresa urmatoarei instructiuni de executat

Variabile in RAM
sintaxa: nume_variabila: .tip valoare

Tipurile De Date
.long - tip de date pt stocarea intregilor pe 32b (4B/octeti)
 long x = 15 <=> x: .long 15
.word - top de date pt stocarea intregilor pe 16b (2B)
 y: .word 34
.byte - tip de date pt stocarea valorilor pe 8b (1B)
 (echivalent cu char din C): char ch = 'b'; <=> ch: .byte 'b' sau cb: .byte 8
Pentru siruri de caractere:
-ascii - pt definirea sirurilor de caractere fara '\0' (NULL)
 hw: .ascii "Hello, world!" marime: 13
.asciz - pt definirea sirurilor de caractere cu '\0' (NULL) - Vom lucra doar cu .ASCIZ !!!!!!!
 hw2: .asciz "Hello, world!" marime: 14 (13 + NULL)
.space dim -> declara un spatiu liber de dim B in memorie
 long v[100]; <=> v: .space 400 (un long ocupa 32b = 4B deci 4 * 100 = 400B)
 spatiu pentru un sir de maxim 100 de caractere: str: .space 101 (un caracter are 1B deci 100 * 1 si 1B pt '\0')

Instructiuni
[eticheta:] nume [operanzi]
Instructiunea MOV: mov source, destination (val din source -> destination)
 mov reg, reg: mov %eax, %ebx (%ebx = %eax)
 x: .long 27
 mov x, %ecx

mov %edx, x
mov $1, %eax (%eax = 1; constantele au prefix cu $; $x = &x in C, daca x e variabila declarata, $ ia adresa din me morie)
mov $2, x (x = 2)
mov %edx, $1 - NU SE POATE
Toate instructiunile pot fi sufixate cu tipul de date: movl = mov long; movW = mov word; movB = mov byte
sufixul se foloseste ca sa ma asigur ca operatia e executata pe doua tipuri de date ca cel din sufix ex: movl long, lon g

Apeluri de sistem
EXIT: return 0; <=> exit(0);
WRITE
READ
In limbajele de asamblare, pt fiecare apel de sistem am un cod asociat:
EXIT - 1
WRITE - 4
READ - 5
Instructiunea specifica intreruperii hardware: int $0x80; oblig sistemul de aperare sa verifice tabela de registri
Codul este retinut in reg. %eax!!!!
------- pt executarea EXIT:
mov $1, %eax // Dupa ce am inregistrat codul in %eax, arg sunt stocate in ordine in reg. %ebx, %ecx, %edx
mov $0, %ebx // 0 este argumentul din exit(0) <=> return 0;
int $0x80
------- pt executarea WRITE: are 3 arg (exit are doar 1):
1. STD OUTPUT = 1 // locul UNDE se face afisarea
2. // CE vreau sa scriu (adresa din memorie a sirului de afisat)
3. // CAT vreau sa scriu (nr de caractere)
exemplu la pg2.asm

Structura unui program
** pg1.asm:
-> .data // aici se declara variabilele din memorie exp: x: .long 15 sau g: .space 4
-> .text // aici se pot scrie instructiuni
-> .global main // precizeaza entry pointul din program
-> main: // de aici incepe executarea propriu-zisa
// EXIT
mov $1, %eax
mov $0, $ebx
int $0x80

** pg2.asm:
```
.data
 hw: .asciz "Hello, world!"
.text
.global main
main:
 mov $4, %eax // mut codul sistem pt write in eax
 mov $1, %ebx // EBX = 1 => arg 1 = 1 => STDOUT
 mov $hw, %ecx // arg 2: mut adresa sirului in ecx
 mov 14, %edx // arg 3: cat vreau sa scriu (hw ocupa 14 explicat mai sus)
 int $0x80

 mov $1, %eax
 mov $0, %ebx
```

int $0x80 // EXIT(0)

SETUP PT LAB:
gcc, g++ multilib
in terminal pt exec: gcc -m32 pg2.asm -o pg2
  ./pg2 // ca sa execute
  SAU
  gdb pg2 // pt debug
DEBUG: gdb pg2
b main // pt breakpoint in main (se opreste cand ajunge la main
run // incepe executarea
i r // information registers afiseaza tabela de registri
i r eax // daca ma intereseaza un registru anume afiseaza ce valoare e in el la linia la care a ajuns
stepi // step instruction care muta debuggerul la urmatoarea linie

Instructiunile aritmetice
add // are 2 operanzi: add op1, op2 <=> op2 += op1
 add %eax, %eax // <=> eax += eax
 add %eax, %ebx // <=> ebx += eax
 add x, %eax
 add $1, %ecx
sub // are 2 operanzi: sub op1, op2 <=> op2 -= op1 in rest ca la add
mul
div