Ensamblador

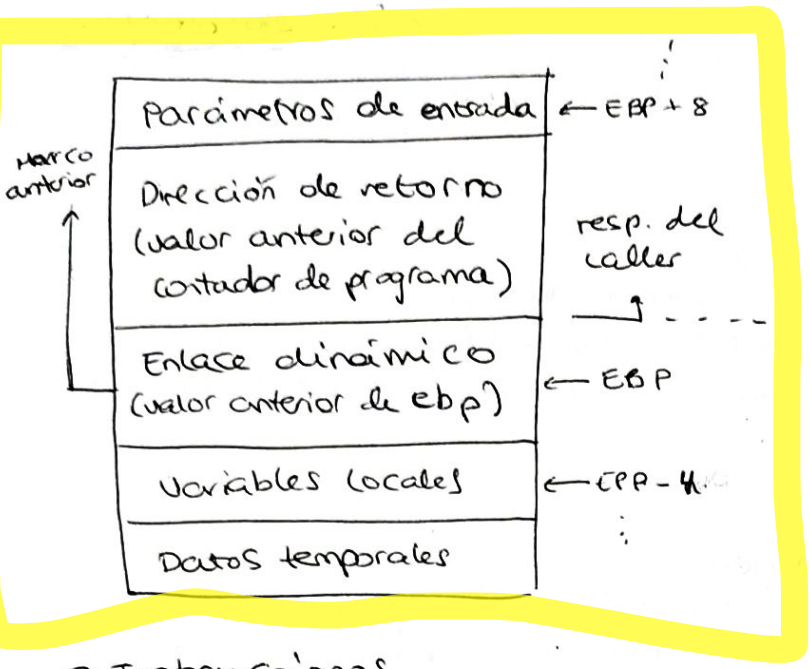
Función (estructura):

```
.text
.globl (nombre)
.type (nombre), @function
(nombre):
    pushl %ebp              #PRÓLOGO
    movl %esp, %ebp         #salvar enlace dinámico
    subl $(4*n), %esp       #establecer nuevo enlace dinámico
                            #reservar espacio para n var. locales.
    :
    {INSTRUCCIONES}
    :
    movl %ebp, %esp         #EPÍLOGO
    popl %ebp               #borra variables locales y temporales
    ret                     #restaura marco de activación anterior
                            #devuelve control.
```

Main (con printf, scanf y llamada a función).

```
.text
.globl main
.type main, @function
main:
    {PRÓLOGO}
    :
    movl $35, p     #var global p = 35
    movl $12, q     #var global q = 12
    pushl q
    pushl p
    pushl $s0       #s0 = "p=%d, q=%d\n"
    call printf     #llamada a printf.
    addl $12, %esp  #sacar parámetros de la pila.
    :
    leal -4(%ebp), %eax     #leer dirección de mem. de var. local
    pushl %eax
    pushl $a
    pushl $s2       #s2 = "%d %d"
    call scanf      #llamada a scanf
    addl $12, %esp  #sacar parámetros de la pila
    pushl -4(%ebp)
    pushl $a
    call media      #llamada a función media
    addl $8, %esp
    movl %eax, -8(%ebp) #guarda el resultado en una var local.
    movl $0, eax    #return 0
    {EPÍLOGO}.
```

Marco de activación



Instrucciones

cdq : extiende bit de signo %eax sobre %edx (hacer antes de la división)

divl %ebx : divide %edx entre %ebx. y guarda en %eax

imull $3, %eax : multiplica 3* %eax y guarde en el 2º par.

addl %ebx, %eax: suma %ebx y %eax y guarde en el 2º par.

subl %ebx, %eax: resta %eax − %ebx y guarde en el 2º par.

Instrucciones float

.LCO:
.float 7.0 : declarar los valores float en la sección de constantes

flds a : pushFl a : push en la pila de registros flotantes

fmulp : pushFL (popFL * popFL)

fdivp : pushFL (popFL / popFL)

faddp : pushFL (popFL + popFL)

fsubp : pushFL (popFL - popFL)

fstpl c : move popFL, c.

fstpl (%esp) : push popFL

⊛ Estructuras de control

→ IF - ELSE

```
//Instrucciones de la condición
cmpl $0, %eax
je false
//instrucciones del if
jmp final
false:
//instrucciones del else
final:
```

→ WHILE

```
start:
//instrucciones de la condición
cmpl $0, %eax
je final
//instrucciones del cuerpo
jmp start
final:
```

⊛ Operaciones lógicas

→ OR (op1 → %eax, op2 → %ebx)

```
cmpl $0, %eax
je 1else
movl $1, %eax
jmp fin
1else:
cmpl $0, %ebx
je 2else
movl $1, %eax
jmp fin
2else:
movl $0, %eax
fin:
```

→ AND (op1 → %eax, op2 → %ebx)

```
cmpl $0, %eax
jne 1else
movl $0, %eax
jmp fin
1else:
cmpl $0, %eax
jne 2else
movl $0, %eax
jmp fin
2else:
movl $1, %eax
fin:
```

→ NOT (op → %eax)

```
cmpl $0, %eax
je else
movl $0, %eax
jmp fin
else:
movl $1, %eax
fin:
```

⊛ Operaciones de comparación

→ MAYOR QUE (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
jle false
movl $1, %eax
jmp fin
false:
movl $0, %eax
fin:
```

→ MENOR QUE (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
jge false
movl $1, %eax
jmp fin
false
movl $0, %eax
fin:
```

→ MAYOR O IGUAL (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
jge true
movl $0, %eax
jmp fin
true:
movl $1, %eax
fin:
```

→ MENOR O IGUAL (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
jle true
movl $0, %eax
jmp fin
true:
movl $1, %eax
fin:
```

→ IGUAL (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
je true
mov $0, %eax
jmp fin
true:
mov $1, %eax
fin:
```

→ DISTINTO (op1 → %eax, op2 → %ebx)

```
cmpl %eax, %ebx
jne true
mov $0, %eax
jmp fin
true:
mov $0, %ecx
fin:
```