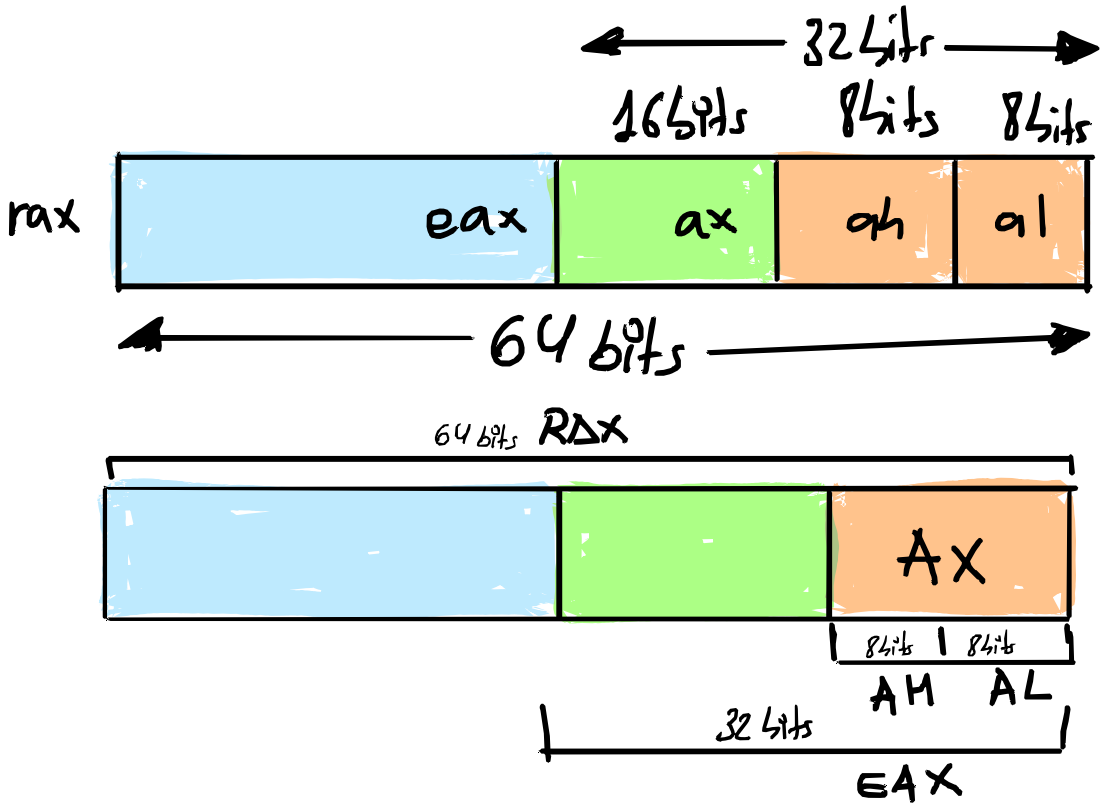


* REGISTROS GENERALISTAS = ①



- **al** → Acceso a los primeros 8 bits
- **ah** → Acceso a los segundos 8 bits
- **ax** → acceso a los primeros 16 bits al mismo tiempo

* ~~acceso~~ a $\rightarrow ah + al = ax$

- **eax** → Acceso a los primeros 32 bits
($ax + ah + al$)
- **rax** → Acceso a los 64 bits

* PROCESSOR : X86-64

(MEMORY CACHE)

← 32 bits →

16 bits

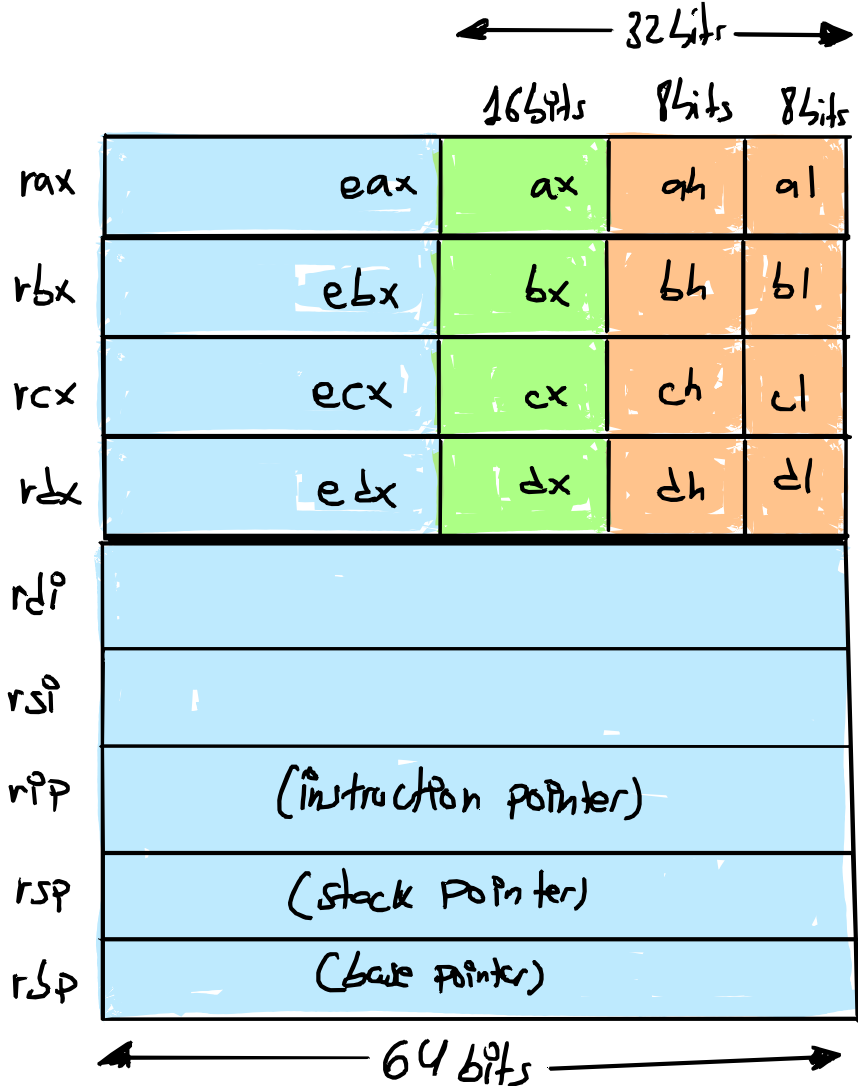
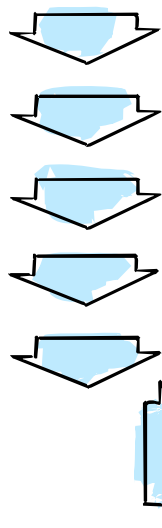
8 bits

8 bits

rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rdi				
rsi				
rip	(instruction pointer)			
rsp	(stack pointer)			
rbp	(base pointer)			

← 64 bits →

DONDE SE REGISTRA
 LOS ARGUMENTOS DE
 FUNCIONES



- LA ABI DE 64 bits (system V)
 PARA LOS PRIMEROS 6 ARGUMENTOS

- | | |
|-------|-------|
| ① RDI | ④ RCX |
| ② RSI | ⑤ R8 |
| ③ RDX | ⑥ R9 |

* A PARTIR DE 6°
 SE DEMANDAN EN
 EL STACK (RSP)

* REGISTROS MAS IMPORTANTES *

- **IP** → INSTRUCTION POINTER

- **ESP** → STACK POINTER

- **EBP** → BASE POINTER

- **eax** → PRIMER REGISTRO

- **RIP** → LE DICE A LA CPU CUAL ES LA SIGUIENTE LÍNEA DEL CÓDIGO

→ ESTE REGISTRO ES MUY IMPORTANTE PARA LOS HALTERS

→ SI TE HACES CON EL CONTROL DE ESTOS REGISTROS, TE HACES CON EL CONTROL DE EJECUCIÓN DEL PROGRAMA.

- **RSP** → PARA SABER EN QUE POSICIÓN DEL STACK ESTAMOS ACTUALMENTE

↳ **STAMPES** DENTRO DE **INTERIORES** STACK.

- **RBP** → GUARDAR LA POSICIÓN DONDE STACK AL MOMENTO DE ENTRAR EN UNA FUNCIÓN.
 - PARA PODER CREAR LAS VARIABLES LOCALES DE UNA FUNCIÓN Y PODER ACCEDER A ELAS:
int, char, etc.
 - ÚNICO SE MUEVE DENTRO DE UNA FUNCIÓN.

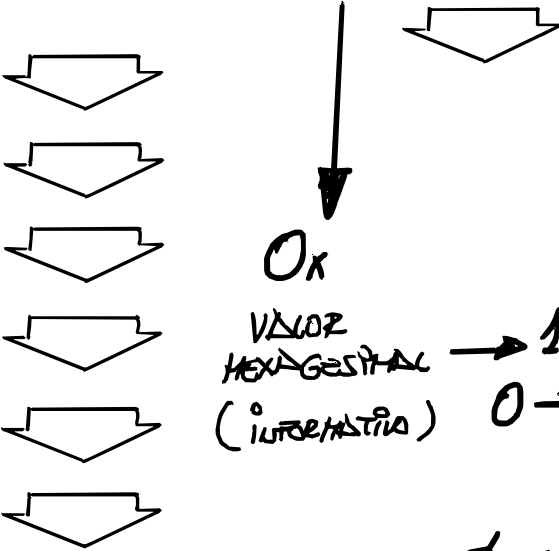
• **RSP + RBP**

— SE USA PARA RESTAURAR LA EJECUCIÓN ENTRE COMIDAS DE FUNCIONES.

- **RDX** → PRIMER REGISTRO DE MANEJO
 - SE USA PARA ALMACENAR EL VALOR QUE DEVUELVEN LAS FUNCIONES.

* EJEMPLOS *

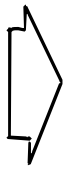
mov eax, 0xdeadbeef



- **MOVIMIENTO DE VALOR:** 0xdeadbeef
AL REGISTRO: eax

* Realmente es el registro RAX
* pero estamos accediendo a los primeros 32 bits

		← 32 bits →		
		16 bits	8 bits	8 bits
rax	00000000	dead	be	ef
rbx				
rcx				
rdx				



mov eax, 0xdeadbeef

INSTRUCTION POINTER



← 32 bits →

16 bits

8 bits

8 bits

rax	00000000	dead	be	ef
rbx	00000000	0000	ba	be
rcx	00000000	dead	be	ef
rdx				

mov bx, 0babe
mov rcx, 0xdeadbeef
mov ecx, 0xdeadbafe



rcx

00000000	dead	ba	fe
----------	------	----	----

¡ESTA PRIMERA VEZ
16 bits en rcx y sobreescribe

* STACK *

2

- EL STACK TIENE DOS FUNCIONES PRINCIPALES
- EL STACK TIENE LA CARACTERÍSTICA DE SER:

- LIFO: Last In First Out

- STACK : LIFO
- QUEUE : FIFO

* FUNCIONES PRINCIPALES *

- PUSH : RECIBE UN DOCUMENTO

push *src* : stack [--rsp] = *src*

push *src* : stack [--rsp] = *src*

1. push RECIBE UN ARGUMENTO : *src*
 2. AUMENTA Δ : stack Y RESTA UNA UNIDAD Δ STACK POINTER (RSP)
 3. UNA VEZ QUE HA RESTADO ESA POSICIÓN ALMACENA EL ARGUMENTO *src*.
-

push *src* : stack [--rsp] = *src*

mov *rax*, 42 → ASIGNAMOS EL VALOR 42 AL REGISTRO RAX

push *rax* → GUARDAMOS EL REGISTRO RAX EN EL STACK

0	
1	
2	
3	
4	42

← RSP (RSP apunta a la 1ª posición)

↪ ESTÁ EN LA POSICIÓN 4
QUE ES LA PRIMER DEL
STACK

push *src*: $\text{stack}[\text{--rsp}] = \text{src}$

mov *rax*, 42

push *rax*

mov *rax*, 69

push *rax*

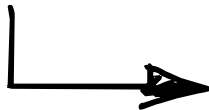
push *rax*

0	
1	
2	69
3	69
4	42

← RSP

- **POP**: RECIBE UN ARGUMENTO
: SACA LO QUE ESTÁ ARRIBA DEL STACK
X SACA LO QUE ESTE PUNTADEO EN
ESTOS MOMENTOS RSP (stack pointer)

pop *dst*: $\text{dst} = \text{stack}[\text{rsp}++]$



PRIMER ARGUMENTO Y
DESPUES MUEVE EL PUNTERO

X UNA VEZ QUE SACA EL VALOR
LO GUARDA EN LA VARIABLE *dst*

pop $dst: dst = stack[rsp++]$

① $pop\ rbp$ $rbp =$

0	
1	
2	69
3	69
4	42

← RSP



② $pop\ rbp$ $rbp = 69$

0	
1	
2	69
3	69
4	42

← RSP



③ $pop\ rbp$ $rbp = 69$

0	
1	
2	69
3	69
4	42

← RSP



$rbp = 42$
 $rsp =$

0	
1	
2	69
3	69
4	42

← RSP

* EJEMPLOS *

pop *dst* : $dst = stack[rsp++]$

push *src* : $stack[--rsp] = src$

0	99	← RSP
1	11	
2	4D	
3	69	
4	42	

MOV *rax*, 42 ; push *rax*

MOV *rax*, 69 ; push *rax*

MOV *rax*, 4D ; push *rax*

MOV *rax*, 11 ; push *rax*

MOV *rax*, 99 ; push *rax*

IMPORTANTE: SI INTRODUCIMOS UN VALOR MÁS:

▲ STACK OVERFLOW ▲

0xdead ← RSP

0	99
1	11
2	4D
3	69
4	42

MOV *rax*, 0xdead ; push *rax*

● LAS SOLUCIONES DE LA MEMORIA QUE ESTÁ RESERVADA PARA ES STACK

* CONCEPTOS TÉCNICOS * (3)

• MEMORIA CPU: (CACHE)

- NO EXISTE UNA MEMORIA SEPARADA
- LOS REGISTROS FORMAN PARTE DE LA ARQUITECTURA INTERNA DEL PROCESADOR

- SE LLAMA REGISTRO INTERNO DEL PROCESADOR.

- Register file → CONJUNTO FÍSICO DE REGISTROS EN CPU

- CPU internal memory. → ENTIDAD GENERAL INCLUYE REGISTROS Y +.

• RAM: MEMORIA PRINCIPAL EXTERNA A LA CPU.

- EN LA RAM ESTÁ EL STACK (pila)

LABORATORY

01

4

STACK

0	
1	
2	
3	
4	
5	
6	
7	&4
-	????????

```

0  int square (int num)
1  {
2      return (num*num);
3
4  int main()
5  {
6      int result = square(10);
7      return (0);
8  }

```

← (rsp)

← (rbp)

0	
1	
2	
3	
4	
5	
6	&5
7	&4
-	????????

```

0  int square (int num)
1  {
2      return (num*num);
3
4  int main()
5  {
6      int result = square(10);
7      return (0);
8  }

```

← (rsp)

← (rbp)

0	
1	
2	
3	
4	
5	
6	& 5
7	& 4
-	?????? ??

```
(rtp) → 0 int square (int num)
          1 { return (num * num)
          2 {
```

← (rsp)

← (VDP)

0	
1	
2	
3	
4	
5	&7
6	&5
7	&4
-	????????

```

0 int square (int num)
1 {
2     return (num * num)
3 }

```

← (rsp)

← (VOP)

0	
1	
2	
3	
4	
5	&7
6	&5
7	&4
-	????????

```

0  int square (int num)
1  {
2      return (num * num);
3  }

```

(rtp) →

```

4  int main()
5  {
6      int result = square(10);
7      return (0);
8  }

```

← (rsp) ← (rbp)

↓ (RBP)

- ESTA ES LA REFERENCIA PARA LAS VARIABLES LOCALS DE LA FUNCIÓN (SQUARE).

0	
1	
2	
3	
4	10
5	&7
6	&5
7	&4
-	????????

```

0  int square (int num)
1  {
2      return (num * num);
3  }

```

(rtp) →

← (rsp)

← (rbp)

```

4  int main()
5  {
6      int result = square(10);
7      return (0);
8  }

```


* AHORA ES CUANDO REALIZA EL CÁLCULO *

0	
1	
2	
3	
4	10
5	&7
6	&5
7	&4
-	???????

```

0  int square (int num)
1  {
2  }   return (num * num)
3
4  int main()
5  {   int result = square(10);
6     return (0);
7  }

```

(rip) → 2

← (rsp) ← (rbp)

* EL RESULTADO DE LA OPERACIÓN:

100

• se almacena en el registro RAX
de la CPU: 1er registro

* AHORA ENTENDIMOS QUE HABER UN
POP STACK PARA EXTRAER EL RESULTADO
QUE SE ENCUENTRA EN LA POSICIÓN 5.

* LO QUE HAY EN LA POSICIÓN 5 ES
LA DIRECCIÓN 7

`POP rbp : rbp = stack[rsp++]`

1: pop almacena en el registro que le digas el valor de la posición RSP del stack y mueve el puntero 1 posición:

0	
1	
2	
3	
4	10
5	&7
6	&5
7	&4
-	????????

0 int square (int num)
1 {
2 return (num * num)
3 }

← (rsp) ← (rbp)

4 int main()
5 {
6 int result = square(10);
7 return (0);
8 }

⇨ ⇨ ⇨ `POP rbp` ⇩ ⇩ ⇩

- * COMO RESULTADO DEL POP RBP .
- * EL RBP SE MUEVE A SU POSICIÓN INICIAL.
- * RSP SUMA UNA POSICIÓN Y SE MUEVE A LA POSICIÓN 6.

0	
1	
2	
3	
4	10
5	&7
6	&5
7	&4
-	???????

```

0  int square (int num)
1  {
2  }
3  {
4  int main()
5  { int result = square(10);
6    return (0);
7  {

```

(rip) → 2

← (rsp) 6

← (rbp) 7

- * AL HACER Return: SE EJECUTA UN POP
- * ESE POP SE A SIGUE AL RIP:



0	
1	
2	
3	
4	10
5	&7
6	&5
7	&4
-	???????

```

0  int square (int num)
1  {
2      return (num * num);
3  }

```

```

(rbp) -> 4  int main()
5  {
6      int result = square(10);
7      return (0);

```

← (rsp) ← (rbp)

* MOV & ARITHMETICA *

④

instrucción principal  `MOV dst, src`

- `MOV dst, src` : SRC SE LO ASIGNA A DST.

- `MOV rax, 42` : ASIGNA EL VALOR 42 A RAX

↳ EU "C" → RAX = 42.

- `ADD dst, src` : SUMA

- `ADD rax, 42` ⇒ $rax += 42$

- `SUB dst, src` : RESTA

- `SUB rax, 42` ⇒ $rax -= 42$

FLAGS

Las Flags son booleanas:

Posibles valores: 1, 0
(ENCENDIDO o APAGADO)

*ZF - ZERO FLAG

↳ TE DICE SI EL RESULTADO DE UNA OPERACIÓN ES 0

$$7 - 5 = 2$$

ZF = 0 → FLAG APAGADO

$$7 - 7 = 0$$

ZF = 1 → FLAG ENCENDIDO

*SF - Sign FLAG

↳ SE ENCENDE CUANDO EL BIT MÁS A LA IZQUIERDA ESTÁ ENCENDIDO.

POR LO TANTO ES UN NÚMERO NEGATIVO

mov al, 0; al = 8-bit register

sub al, 1; 0b11111111 (two's complement)

SF = 1

* GUARDAMOS EL VALOR DE 0 EN AL

* RESTAMOS 1 EN AL

* REPRESENTACION BINARIA DE -1

0b11111111

* SE ENCUEDE SF

* **CF** - Carry Flag

→ SE ENCUEDE CUANDO TENEMOS UN OVERFLOW DE UN NÚMERO QUE NO TIENE SIGNO.

→ TENEMOS UN NÚMERO PRESTADO EN UNA RESTA

→ AFINCEJAR EL ÚLTIMO BIT QUE SALIÓ EN UN SHIFT

● UNSIGNED OVERFLOW:

mov al, 255; 8-bit register

* MOVEMOS EL VALOR 255 A UN REGISTRO EN MEMORIA CPU DE 8bit.

* 1 byte = 8bits solo PUEDE CODIFICAR EN BINARIO 256 VALORES SIN SIGNO (positivos) del 0 al 255

add al, 1;

al = 0, CF = 1 (unsigned overflow)

* AL SOBREPASA EL VALOR MÁXIMO Y VA A VALER A EMPETAR A CODIFICAR

● NÚMERO PRESTADO EN RESTA:

mov a1, 5; 8-bit register
sub a1, 10; $5 < 10$, CF=1,
(NECESITAMOS NÚMERO PRESTADO DEL
SIGUIENTE NIVEL)

● EL ÚLTIMO BIT QUE SE SACÓ:

mov a1 0b10000001 → 0b REPRESENTACIÓN
GUARDA
shr a1, 1; shift right, CF=1
(EL ÚLTIMO BIT QUE SE SACÓ ES 1),

0b01000000 | 1

→ con shift right, movemos los bits a la
derecha. EL 1 QUE SE SACÓ SE
GUARDA EN EL Carry Flag • CF=1

X. **OF** - Overflow Flag.

↳ OVERFLOW DE ARITHMÉTICA DE NÚMEROS COM SINAL.

mov al, 100;

add al, 50; 127 (máximo com sinal 8-bits)

; Resultado de 150 volta a -106

; OF = 1 (Signed overflow)

(DESDE -128 a 127)

(256 VALORES DISTINTOS)

#####*#####

RESUMEN

#####*#####

ZF - ZERO FLAG : Resultado es 0

SF - SIGNED FLAG : bit mas grande es 1 (-)

CF - CARRY FLAG : prestado, bit se sale,
unsigned overflow

OF - OVERFLOW FLAG : overflow con signo (-)

* INSTRUCCIONES CONDICIONALES *

cmp $a, b \rightarrow b - a$, active flags

* si se activa ZF es que el valor es cero. Por lo tanto son iguales.

test $a, b \rightarrow b \& a$, active flags

* USAMOS LA PUERTA LÓGICA DE AND.

AND \rightarrow MULTIPLICACIÓN

0 0 0

0 1 0

1 0 0

1 1 1

jmp function \rightarrow salte function

je function \rightarrow salte a function

* si $ZF = 1$

jne function \rightarrow salte function

* si $ZF = 0$

jc function \rightarrow salte si CF se active (variable $\neq 0$ o \emptyset)

* FUNCIONES *

call : push **nip** , jmp fn

↳ ES UNA LLAMADA A UNA FUNCIÓN

↳ HACE UN PUSH AL Register Instruction Pointer
RIP

↳ SALTA A LA FUNCIÓN.
CAMBIA EL INSTRUCCIÓN POINTER AL
AL REGISTRO DONDE CONTIENE EL PUNTO DE LA
FUNCIÓN

(LO HEMOS VISTO EN EL LABORATORIO 1)

ret : pop **nip**

↳ FINAL DE UNA FUNCIÓN Y HACE
POP A RIP:

- POPA el último elemento del STACK
Y LO ASIGNA A RIP.

* SECCIONES *

5

NAME	DESCRIPTION
.text	exe

• .text : EJECUTABLE

● EJEMPLO ●

section .text
global _start

; Code segment
; Entry point for linker

_start:

; exit program

mov rax, 60 ; syscall: exit

mov rdi, 0 ; exit code 0

syscall

NAME	DESCRIPTION
.text	exe
.rodata	read-only



• rodata → DADOS DE GUARDA COMO CONSTANTES

● EXEMPLO ●

seção .rodata

const1 db "Sou Constante", 10, 0 ; 10 = \n, 0 = \0

const1_len equ \$ - const1

NAME	DESCRIPTION
.text	exe
.rodata	read-only
.data	init vars



• data → VALORES PREDEFINIDOS
 → VALORES ESTÁTICOS
 → VALORES GLOBALES
 * SI SE PUEDE MODIFICAR

● EJEMPLO ●

section .data

msg1 db "Hola data data", 10, 0

msg1_len equ \$ - msg1

NAME	DESCRIPTION
.text	exe
.rodata	read-only
.data	init vars
.bss	uninit vars



• bss → VARIABLES NO INICIALIZADAS.
 * COMO NO ESTAN INICIALIZADA, NO GUARDA
 EL VALOR.

• EJEMPLO 1

section .bss

buffer resb 64 ; reserve de 64 bytes

number resq 1 ; reserve de 8 bytes

* **RESB** → 64 bytes

* **RESQ** → QUARTER (8 bytes)

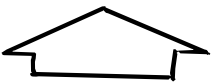
* COMPILER *

6

EMSEMBLER: `nasm -f elf64 ./codigo.asm`

ELDER Y
COMPILER: `ld -o ./codigo ./codigo.o`

EJECUTOR: `./codigo`

Linux 

Macos 

EMSEMBLER:

`nasm -f macho64 ./code.s -o ./code.o`

ELDER Y
COMPILER:

`gcc code.o -o my-code -nostartfiles -e
_start -Wl,-Uerror -Wl,-Uextra`

EJECUTOR: `./my-code`

* DEBUGGER *

gdb ./codigo

gef condps context+layout "regs stack code"

* LINUX *

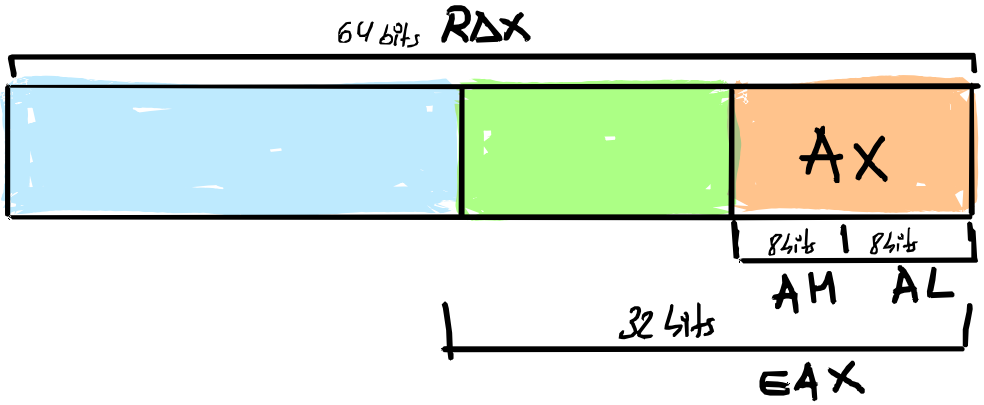
* DEMO *

VER : lab01.s

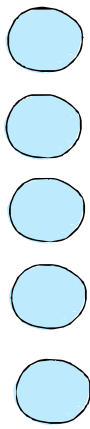
* MOSTRAR SALIDA *

- DESPUÉS DE EJECUTAR NUESTRO PROGRAMA
- **echo \$?** → NOS MUESTRA LA SALIDA DE NUESTRO PROGRAMA EN EL TERMINAL
- **strace ./codigo** → MUESTRA INFO

* TABLES *



&

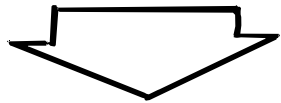


≡

≡

0

#####



1