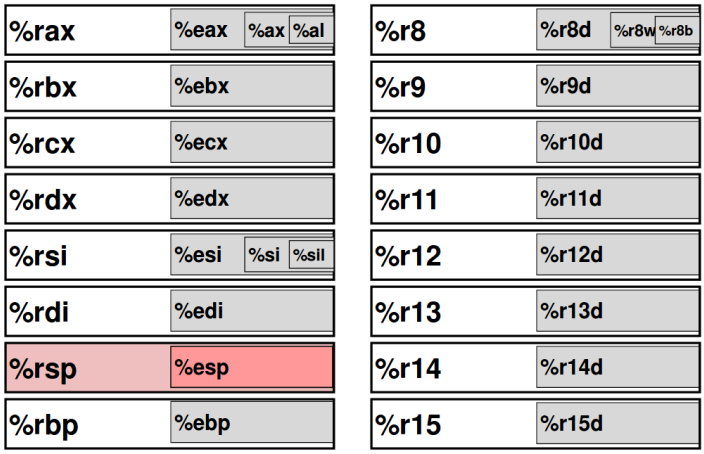
Todo lo que necesitas saber para entender el lenguaje ensamblador

Modos de direccionamiento:
-Inmediato: $10 → Accedes al valor
-Registro: %rax → Accedes al registro, no a lo que guarda
-Memoria: < desplazamiento(%base, %indice, escala) >
→ Accedes a M[desplazamiento+base+indice*escala]
*siempre se direcciona a memoria desde registros (%base)

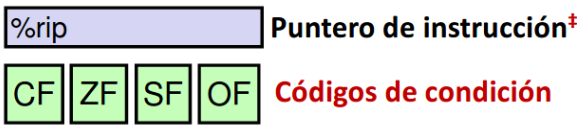
Tamaño de los objetos C (en bytes)

Tipo de Datos C	Normal 32-bit	Intel IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8
– o cualquier otro puntero			

Registros en x86-64 (%rax tiene 64b = 8B)



**Estamos en little endian → %al guarda el bit menos significativos de %rax*
**Modificar un registro de 32b, también pone a 0 los otros 32 MSB*
*%rsp → tope de la pila actual
*%rbp → pumtero de Marco de pila



Códigos de condicion → se ajustan implícitamente con las operaciones aritméticas (ej: t=a+b)
****(leer antes la página de operaciones)***
Flag de Cero: ZF = 1 → El nuevo Dest es 0 (t==0).
Flag de Signo: SF = 1 → El nuevo Dest es negativo (t<0).
Flag de Acarreo: CF = 1 → Hay acarreo del bit más significativo (desbordamiento sin signo)
Flag de Overflow: OF = 1 → Hay desbordamiento en complemento a dos (con signo)
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

Esto quiere decir, básicamente, que el resultado ha cambiado de signo por la cara, seguramente debido al desbordarse un bit (ocurre, por ejemplo, cuando positivo + positivo = negativo o negativo + negativo = positivo)

Funciones:
Flujo de Control en Procedimientos (cómo se les llama)**:**
call label → guarda la dirección de retorno en pila y salta a la etiqueta label. (dir. de retorno = dir. Siguiente a call)
ret → Recupera la dirección de retorno de pila y salta a ella

Flujo de Datos para Procedimientos
(cómo se pasan los parámetos)**:**

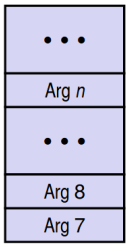
Registros
- **Primeros 6 argumentos**

%rdi
%rsi
%rdx
%rcx
%r8
%r9

- **Valor de retorno**

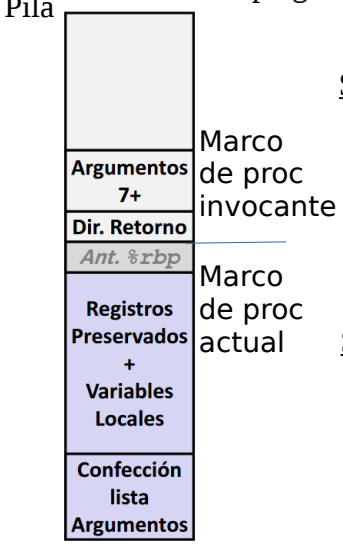
%rax

Pila



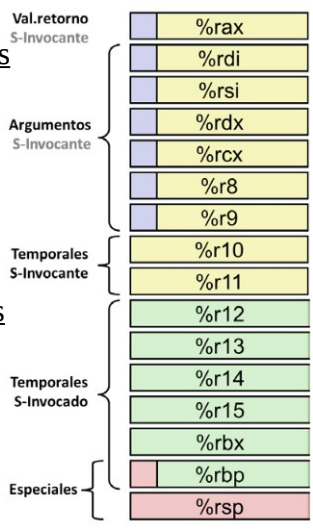
- **Sólo se reserva espacio en la pila cuando se necesita**

Gestión de Datos locales: Marcos de Pila:
%rpb guarda el fin del marco



Salva-Invocantes
Invocante los salva en su marco antes de invocar.

Salva-Invocados
Invocado debe salvarlos en su marco antes de usarlos



Tipos de Datos Básicos:

Enteros — ***(Puntero → 8 Bytes)** — **Punto Flotante**

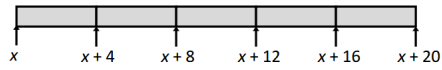
Intel	ASM‡	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int

Intel	ASM	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

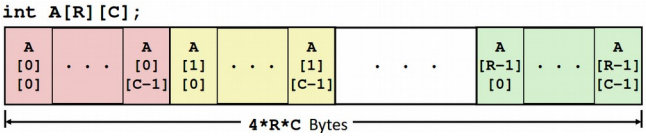
Array → T A[L] reserva una región contigua en memoria de L*sizeof(T) bytes. Ej:

```
int get_digit
† (zip_dig z, size_t digit)
{
  return z[digit];
}
```

```
get_digit:                      # z en %rdi,
†                               # digit %rsi
   movl (%rdi,%rsi,4), %eax     # z[digit]
   ret
```



Arrays Multidimensionales (matrices)
T A[R][C] (analogo a las arrays normales)



ej: int pgh[4][5]; int *get(int index) return pgh[index]; →

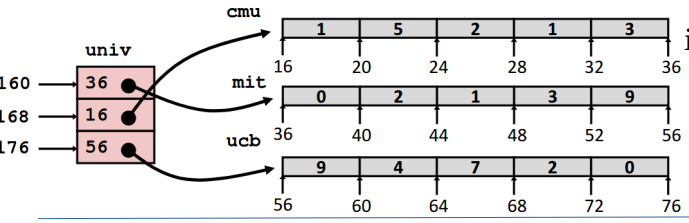
```
get_pgh_zip:
                                # index en %rdi
   leaq  (%rdi,%rdi,4), %rax    # 5 * index
† leaq   pgh(,%rax,4), %rax     # pgh + (20 * index)
  ret
```

```
get_pgh_digit:
                                # digit en %rax
                                # index en %rdi
 leaq   (%rdi,%rdi,4), %rax     # 5*index
 addq   %rax, %rsi              # 5*index+digit

†movl   pgh(,%rsi,4), %eax      # pgh + 4 * (5*index+digit)
 ret
```

← int get(int index, int digit) return pgh[index][digit];

Arrays Multinivel → T A[R][C] pero ya la región de memoria no tiene por qué ser contigua. Ej:



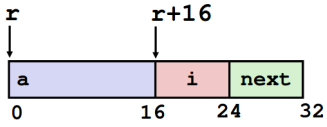
int get(int index, int digit) return univ[index][digit];

```
get_univ_digit:

  movq   univ(,%rdi,8), %rax  # p = *(univ+8*index)
  movl   (%rax,%rsi,4), %eax  # return *(p+4*digit)
  ret
```

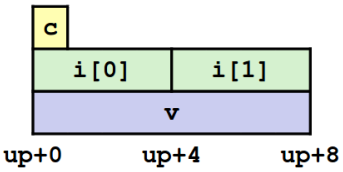
Estructuras: ----------→
(todo bien explicadito en esta diapositiva)--------→

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



Uniones: Solo puede usarse un campo a la vez.
Se reserva conforme al elemento mas grande.

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



- **Accediendo a un Miembro de la Estructura**
  - Puntero indica primer byte de la estructura‡
  - Acceder a los elementos mediante sus desplazamientos†

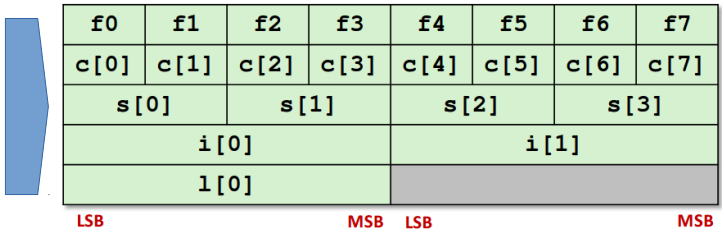
```
void set_i
  (struct rec *r,
      size_t val)
{
‡ r->i = val;
}
```

```
set_i:                  # r en %rdi,
                        # val en %rsi
movq %rsi, 16(%rdi)     # Mem[r+16] = val
ret
```

† "offset"=compensación, "desplazamiento"

Little Endian

PROBLEMA: FUNCIONAN DIFERENTE EN BIG-ENDIAN Y LITTLE ENDIAN



Big Endian

Por ejemplo, leyendo s:

Little Endian lee
[f1f0, f3f2, f5f4, f6f7]

Big Endian lee
[f0f1, f2f3, f4f5, f6f7]

Instrucciones

movq Src, Dest → copia lo que hay en Src en Dest:

	Source	Dest	Src,Dest	Análogo C
movq	Imm†	Reg	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	Reg	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	Reg	`movq (%rax),%rdx`	`temp = *p;`

*NO SE PUEDE DIRECCIONAR DIRECTAMENTE DE MEMORIA A MEMORIA

***Diferencia mov y leaq**:*
dirección 0x110: [0x8]
%rdx: [0x110]

movq(%rdx), %rdi → %rdi[0x8]
leaq(%rdx), %rdi → %rdi[0x110]

Operaciones aritmetico-lógicas:
Src siempre debe ser expresión.
Dest siempre debe ser registro.

Formato		Operación†
addq	Src,Dest	Dest = Dest + Src
subq	Src,Dest	Dest = Dest – Src
imulq	Src,Dest	Dest = Dest * Src
salq	Src,Dest	Dest = Dest << Src
sarq	Src,Dest	Dest = Dest >> Src
shrq	Src,Dest	Dest = Dest >> Src
xorq	Src,Dest	Dest = Dest ^ Src
andq	Src,Dest	Dest = Dest & Src
orq	Src,Dest	Dest = Dest | Src

leaq Src, Dest → copia el valor Src en Dest

Formato		Operación
incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest – 1
negq	Dest	Dest = – Dest
notq	Dest	Dest = ~Dest

¿Que insertan en los huecos vacío?:

El bit más significativo — Aritméticas
Inserta 0 — Lógicas

Sobre códigos de Condición:
compq b, a → equivale a subq (a-b) pero sin guardar el resultado (solo para ajustar flags)
ZF = 1 → a == b SF = 1 → a < b (con signo)
CF = 1 → Hay acarreo con signo (hacer caso si comparas números sin signo)
OF = 1 → Hay acarreo sin signo (hacer caso si comparas número con signo)

testq b, a → equivale a andq (a&b) pero sin guardar el resultado (solo para ajustar flags) FZ = 1 → a&b == 0 SF = 1 → (a&b) < 0 ***El resto de flags no se modifican***

SetCC Dest → Copia en Dest el código indicado en CC.
Si Dest → registro, debe ser de 1 Byte
Si Dest → memoria, solo modifica el LSB

SetCC	Condición	Descripción	
sete	ZF	Equal / Zero	
setne	~ZF	Not Equal / Not Zero	
sets	SF	Sign (negativo)	
setns	~SF	Not Sign	
setg	~(SF^OF)&~ZF	Greater	(signo)
setge	~(SF^OF)	Greater or Equal	(signo)
setl	(SF^OF)	Less	(signo)
setle	(SF^OF)|ZF	Less or Equal	(signo)
seta	~CF&~ZF	Above	(sin signo)
setb	CF	Below	(sin signo)

LSB = Byte menos significativo
MSB = Byte más significativo

Saltos condicionales jCC Dest ----------→

Dest puede ser un valor absoluto (label) o relativo (un número). En este último caso, se suma Dest a la posición de la **siguiente** instrucción al salto.

jCC	Condición	Descripción
jmp	1	**Incondicional**
je	ZF	**Equal / Zero**
jne	~ZF	**Not Equal / Not Zero**
js	SF	**Sign (negativo)**
jns	~SF	**Not Sign**
jg	~(SF^OF)&~ZF	**Greater (signo)**
jge	~(SF^OF)	**Greater or Equal (signo)**
jl	(SF^OF)	**Less (signo)**
jle	(SF^OF)|ZF	**Less or Equal (signo)**
ja	~CF&~ZF	**Above (sin signo)**
jb	CF	**Below (sin signo)**

Sobre pila:
pushq Src → Guarda Src en pila:
1. Escribe Src en la dir. indicada por %rsp
2. Decrementa %rsp en 8

popq Dest → Saca el último valor de pila:
1. Incrementa %rsp en 8
2. Almacena el valor en Dest.

Hay muchas más operaciones, pero como no vienen explicadas en ningún momento, toca buscarlas en internet o en el libro, si queremos saber cómo funcionan.