

Organización del Procesador

ASSEMBLY X86 - cont. Arreglos

Departamento de Computación - UNRC

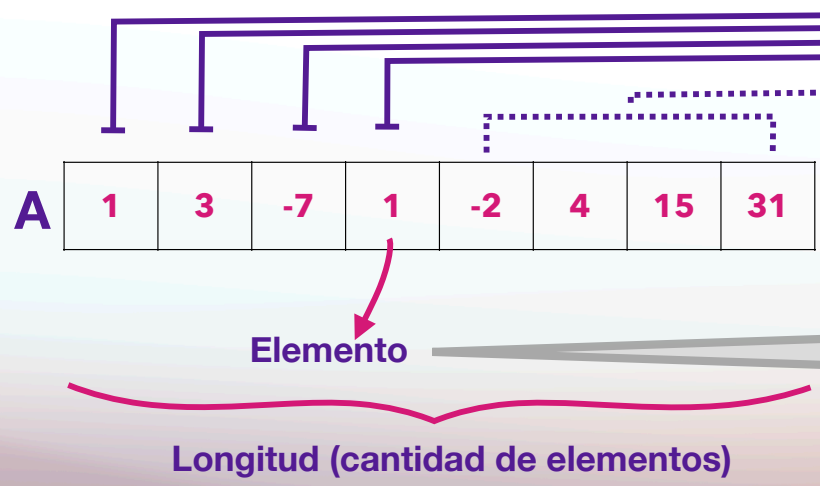
El camino a recorrer

- Un poco de Historia y Sistemas Numéricos
- Introducción a la Electrónica
- Representación de Información
- Cómo computar utilizando la electricidad
- Evolución y funcionamiento abstracto de una computadora
- **Assembly X86**
- Micro-programación (cómo fabricar un procesador)
- Eficiencia
 - Pipelines
 - Memoria Caché
 - Memoria Virtual

Assembly X86 - Arreglos

Un **arreglo** está compuesto por una secuencia homogénea de datos en la memoria. Esta característica permite el acceso directo a sus elementos. Esta dirección la podemos obtener conociendo:

- La dirección del primer elemento
- El tamaño (cantidad de bytes) de cada elemento
- El índice del elemento que necesitamos acceder



MEMORIA															
...															
<i>i</i>	0	0	0	0	0	0	0	0	0						
<i>i</i> +1	0	0	0	0	0	0	0	0	1						
<i>i</i> +2	0	0	0	0	0	0	0	0	0						
<i>i</i> +3	0	0	0	0	0	0	0	1	1						
<i>i</i> +4	1	1	1	1	1	1	1	1	1						
<i>i</i> +5	1	1	1	1	1	0	0	1							
<i>i</i> +6	0	0	0	0	0	0	0	0	0						
<i>i</i> +7	0	0	0	0	0	0	0	0	1						
...															
<i>i</i> +(<i>n</i> *2)	0	0	0	0	0	0	0	0	0						
<i>i</i> +(<i>n</i> *2)+1	0	0	0	1	1	1	1	1							

Suponiendo *n* elementos una representación de 2 bytes (16 bits) por entero y que la primera posición está ubicada en la dirección *i*.

Assembly X86 - Arreglos ¿ cómo podemos acceder a una posición?

Es muy importante tener en cuenta el destino (registro o parte de ellos) donde ubicamos el valor una posición del arreglo.

```
segment .data
```

```
A2 dw 1,2,3,4,5
```

```
segment .text
```

```
...
```

```
mov AX, [A2] ; muevo al AX (16 bits) el primer elemento del arreglo
```

```
mov BX, [A2+3] ;
```

¿ Qué valor queda en BX ?

Assembly X86 - Arreglos ¿ cómo podemos acceder a una posición?

Dependiendo del tamaño de los elementos podemos utilizar el siguiente patrón de acceso indirecto:



; ejemplo de suma de los elemento de un arreglo

segment .data

A2 dw 35,1,17,123,98

segment .text

xor EDX, EDX; *inicializo el acumulador con 0*

mov ECX, 5; *muevo al ECX el tamaño del arreglo*

mov EAX, A2; *muevo al EAX la dirección del comienzo del arreglo*

mov EBX, 0; *inicializo el registro índice con 0*

for1:

add DX, [EAX + 2 * EBX]; *sumo el acumulador con el elemento*

inc EBX ; *incremento el índice*

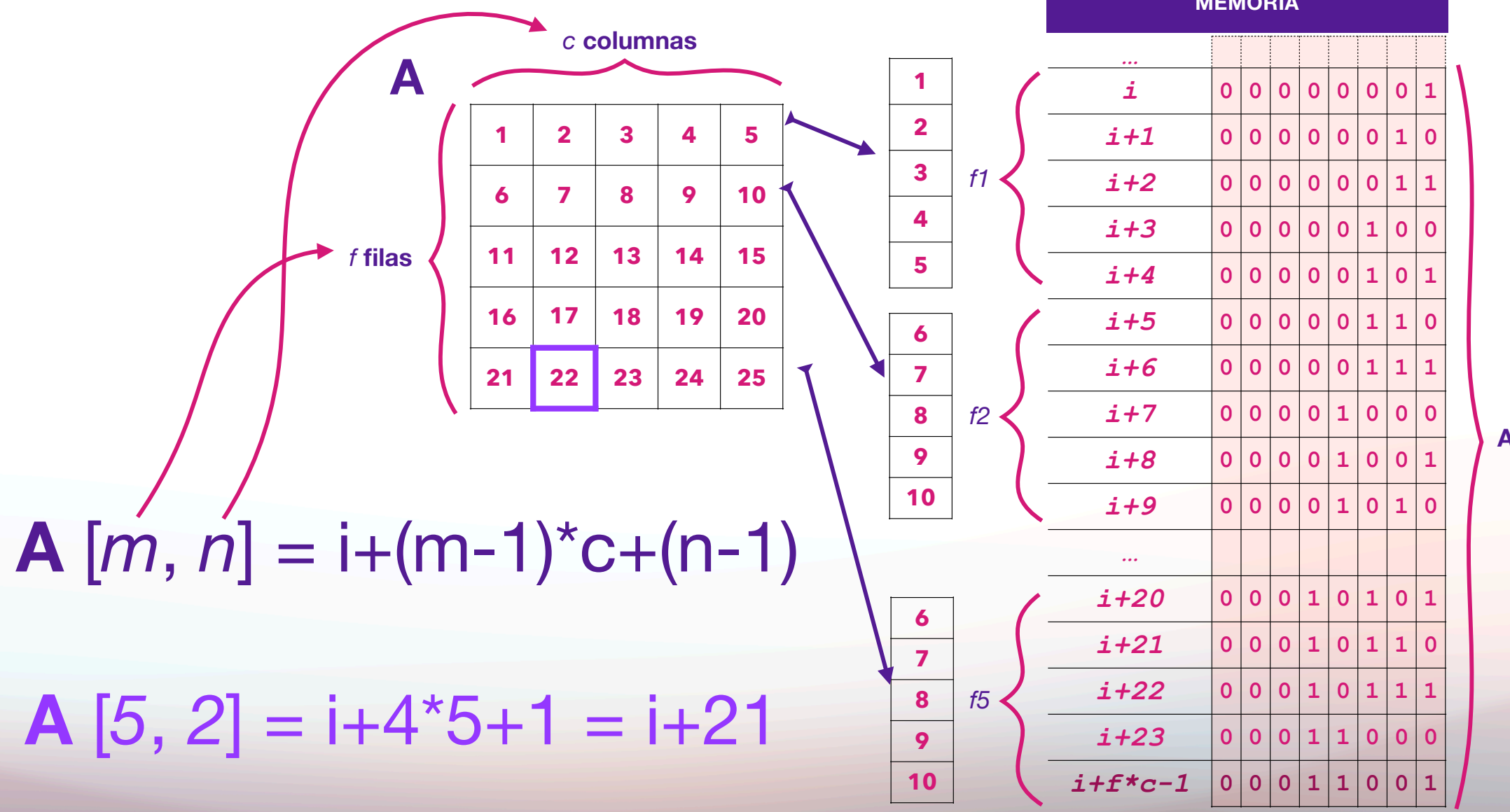
loop for1 ;

Assembly X86 - LEA (Load effective address)

Si necesitamos calcular (hacer operaciones aritméticas) para determinar una dirección, podemos utilizar la instrucción **lea**, por ejemplo:

```
lea EBX, [4 * EAX + ECX] ;
```

Assembly X86 - Arreglos bidimensionales



Assembly X86 - arreglos/strings - Registros e instrucciones especiales

En x86, **EDI** y **ESI** son registros de propósito general utilizados para manipular direcciones de memoria durante operaciones con cadenas. **EDI** se usa comúnmente como índice de **destino** para almacenar datos en memoria, mientras que **ESI** se utiliza como índice de **origen** para cargar datos de memoria. Las instrucciones de cadenas, como **MOVS** (mover), **STOS** (almacenar), **CMPS** (comparar) y **SCAS** (escanear), emplean estos registros para efectuar operaciones en *bloques de memoria contiguos*, facilitando así la manipulación y comparación eficiente de cadenas en ensamblador.

Los registros se autoincrementan o autodecrementan según el FLAG **DF** (Direction Flag). La dirección se puede establecer con la siguientes instrucciones:

- CID** Borra la bandera de dirección (los registros se incrementan).
- STD** Establece la bandera de dirección (los registros se decrementan).

Assembly X86 - arreglos/strings - Registros e instrucciones especiales

; ejemplo que copia un arreglo

segment .data

A1 db "hola mundo",0

segment .bss

A2 db 11

segment .text

mov ECX, 11; *muevo al ECX el tamaño del arreglo*

mov ESI, A1; *establecemos como origen la dirección de A1*

mov EDI, A2; *establecemos como destino la dirección de A2*

cld; *establecemos el flag de dirección como incremento*

rep movsb; *copio el arreglo*

La instrucción **movsb/w/d**, mueve un **1/2/4 bytes** desde la dirección que contiene **ESI** a la dirección que contiene **movsb**.
Luego incrementa/decrementa ambos registros.

Instrucción similar a **loop**, repite la instrucción (de manipulación de arreglos/cadenas) tantas veces como indica **ECX**.


Organización del Procesador

ASSEMBLY X86 - cont. Subrutinas

Departamento de Computación - UNRC

Assembly X86 - Subrutinas

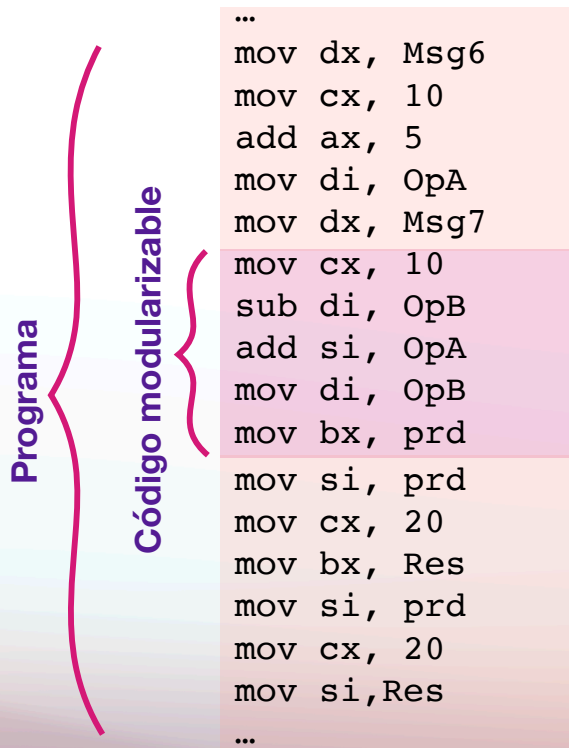
Una **subrutina** es una porción de código **independiente** que realiza una tarea específica dentro de un programa más grande. Las subrutinas se utilizan para **modularizar** el código y mejorar la **organización**, permitiendo **reutilizar** funcionalidades en diferentes partes del programa.



```
...  
mov dx, Msg6  
mov cx, 10  
add ax, 5  
mov di, OpA  
mov dx, Msg7  
mov cx, 10  
sub di, OpB  
add si, OpA  
mov di, OpB  
mov bx, prd  
mov si, prd  
mov cx, 20  
mov bx, Res  
mov si, prd  
mov cx, 20  
mov si, Res  
...
```

Assembly X86 - Subrutinas

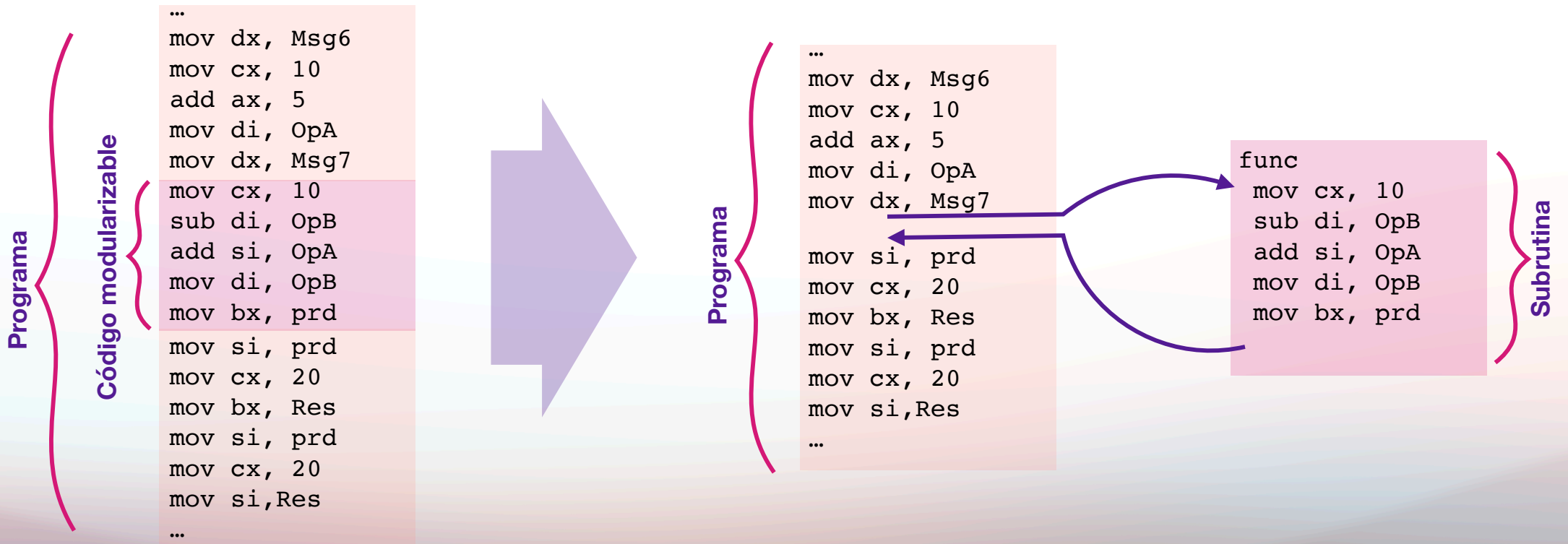
Una **subrutina** es una porción de código **independiente** que realiza una tarea específica dentro de un programa más grande. Las subrutinas se utilizan para **modularizar** el código y mejorar la **organización**, permitiendo **reutilizar** funcionalidades en diferentes partes del programa.



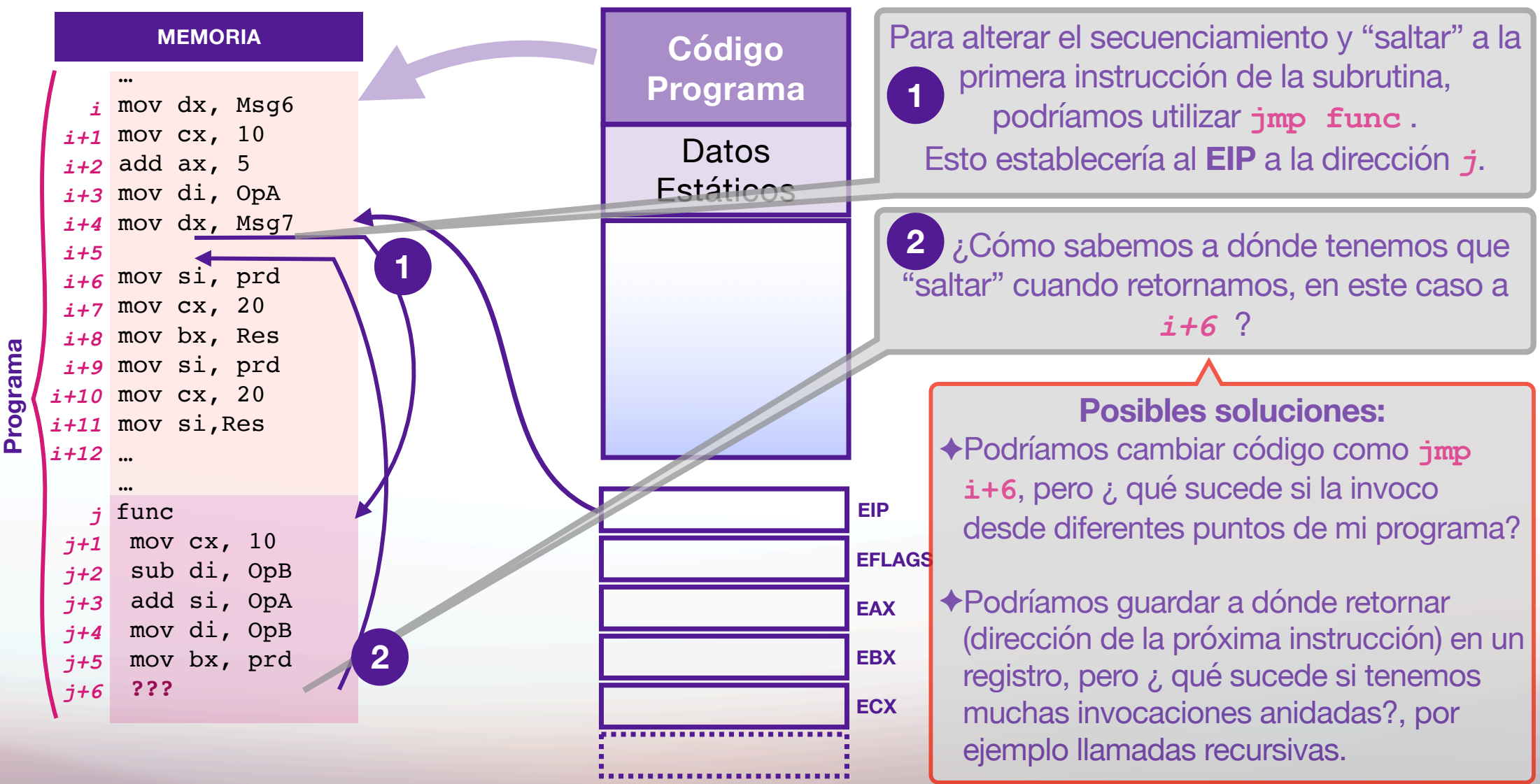
```
...  
mov dx, Msg6  
mov cx, 10  
add ax, 5  
mov di, OpA  
mov dx, Msg7  
mov cx, 10  
sub di, OpB  
add si, OpA  
mov di, OpB  
mov bx, prd  
mov si, prd  
mov cx, 20  
mov bx, Res  
mov si, prd  
mov cx, 20  
mov si, Res  
...
```

Assembly X86 - Subrutinas

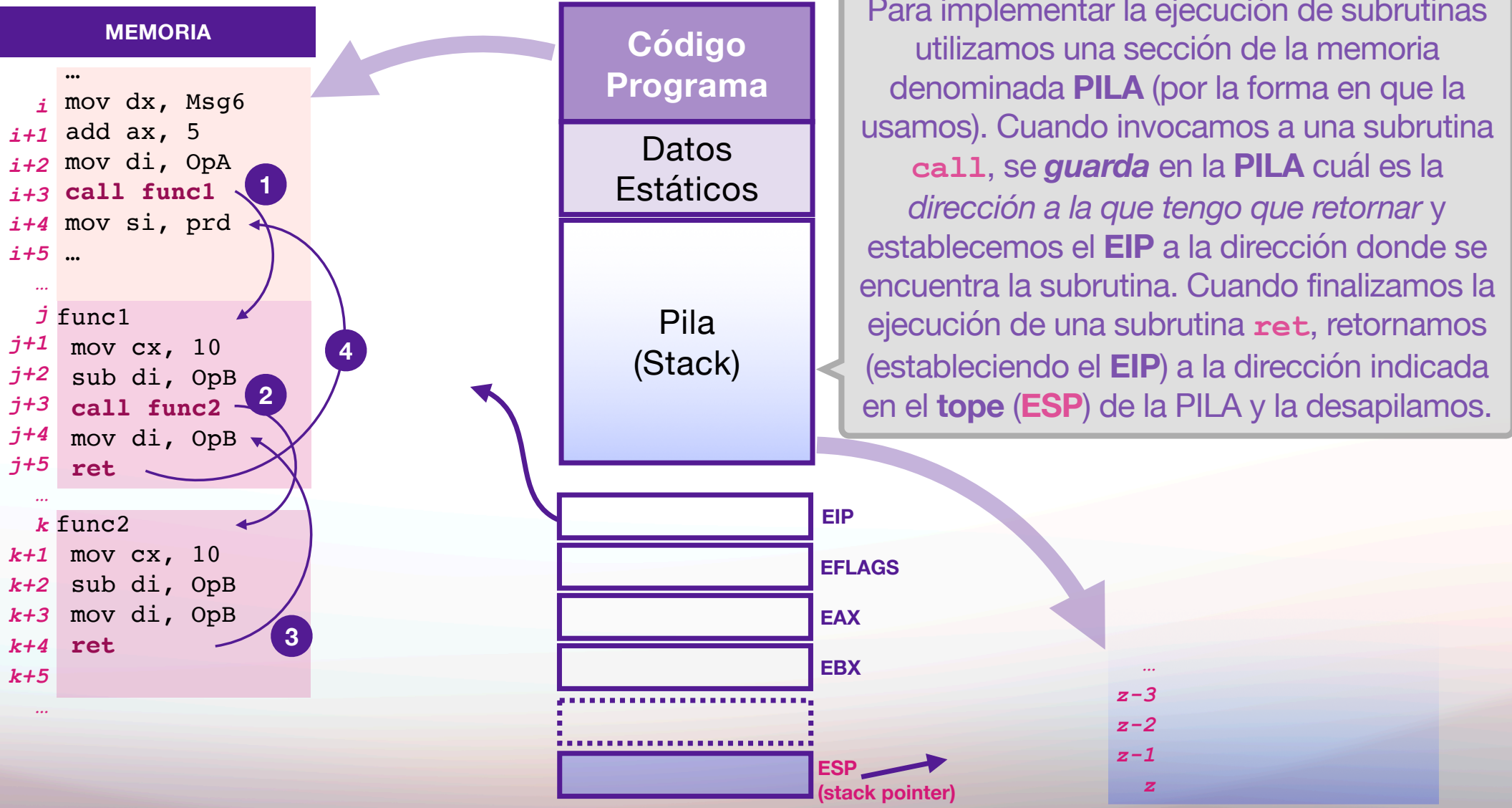
Una **subrutina** es una porción de código **independiente** que realiza una tarea específica dentro de un programa más grande. Las subrutinas se utilizan para **modularizar** el código y mejorar la **organización**, permitiendo **reutilizar** funcionalidades en diferentes partes del programa.



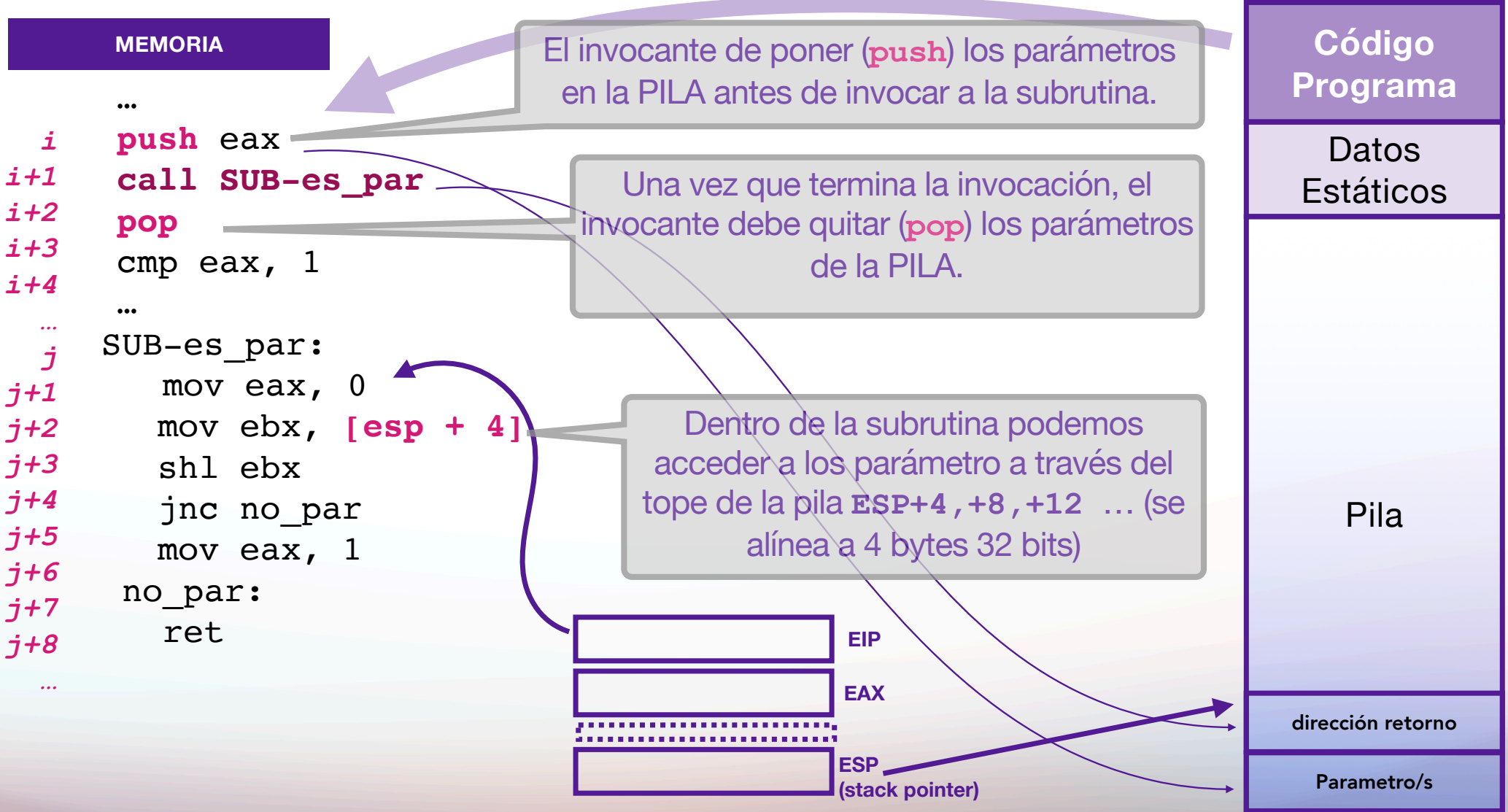
Assembly X86 - Subrutinas - ¿ cómo podemos implementar la idea ?



Assembly X86 - Subrutinas - la PILA, call y ret



Assembly X86 - Subrutinas - Parámetros (primer intento)



Assembly X86 - Subrutinas - parámetros y variables locales

```
...
push A1; apilo la dirección del primer parámetro
push A2; apilo la dirección del segundo parámetro
call SUB-swap ; invoco a la subrutina
pop; quito el segundo parámetro
pop; quito el primer parámetro
...
```

SUB-swap:

<pre>push ebp ;<i>apilo el ebp anterior (para restaurar)</i> mov ebp, esp ;<i>actualizo el actual ebp</i> sub esp, 4 ; <i>reservo espacio para var locales</i> mov [ebp - 4], [ebp + 8] mov [ebp + 8], [ebp + 12] mov [ebp + 12], [ebp -4] add esp, 4 ; <i>quito el espacio reservado</i> pop ebp; <i>restablezco el ebp anterior</i> ret</pre>	<div>Prólogo</div> <div>Cuerpo de la Subrutina</div> <div>Epílogo</div>
--	---

Utilizamos el **Base Pointer EBP** para poder referenciar de igual manera a las variables y parámetros (independientemente de cuántos hay)

