Organización del Procesador

ASSEMBLY X86

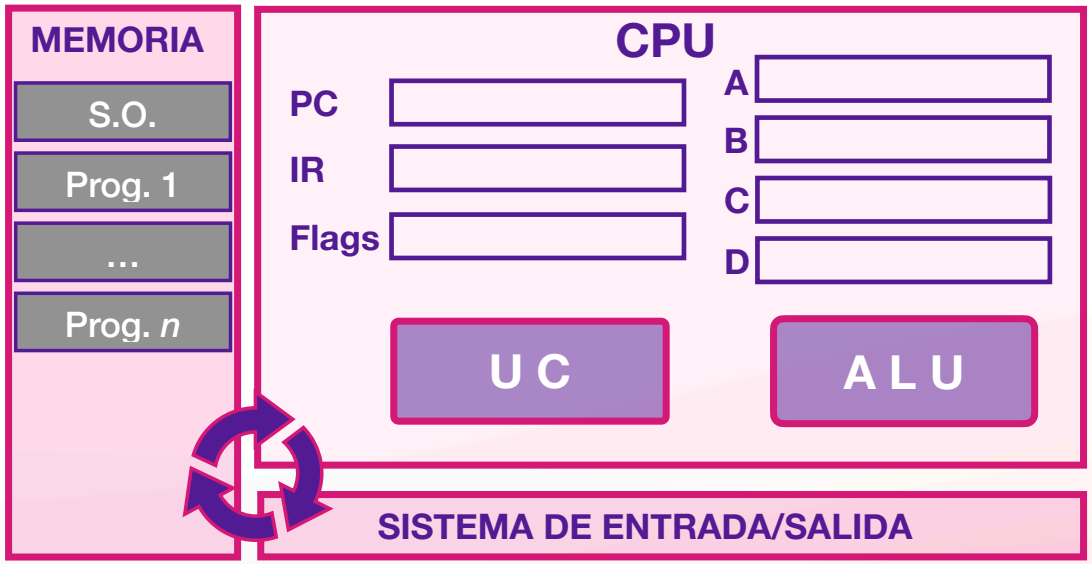
Departamento de Computación - UNRC

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

Funcionamiento abstracto de una computadora

En su gran mayoría las computadoras modernas ejecutan constantemente un programa/proceso particular "**Sistema Operativo**" que se encarga de administrar el uso de los recursos utilizados por **otros programas** que ejecuta el usuario. Estos programas conviven en la memoria mientras son ejecutados.



Evolución de los Procesadores - Otros procesadores RISC vs CISC

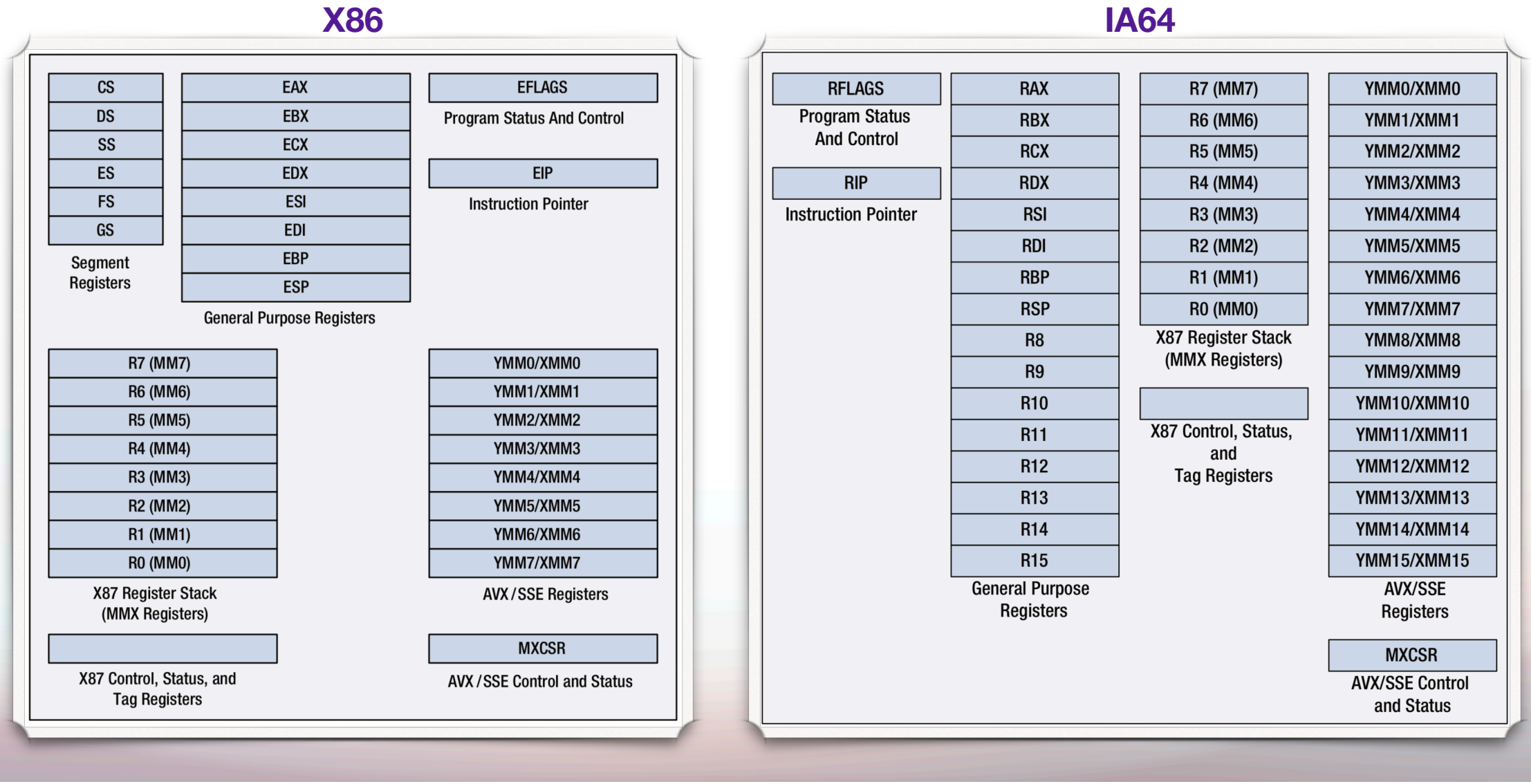
RISC (Reduced Instruction Set Computer)**:**

- Enfoque en simplicidad y eficiencia.
- Utiliza un conjunto de instrucciones reducido y simple.
- Las instrucciones se ejecutan en un ciclo de reloj (generalmente una instrucción por ciclo).
- Mayor cantidad de registros de propósito general para minimizar accesos a memoria.
- Optimizado para ejecución de instrucciones en paralelo.
- Fomenta la optimización del software para explotar características de ejecución eficiente.
- Ejemplos de arquitecturas RISC incluyen MIPS y ARM.

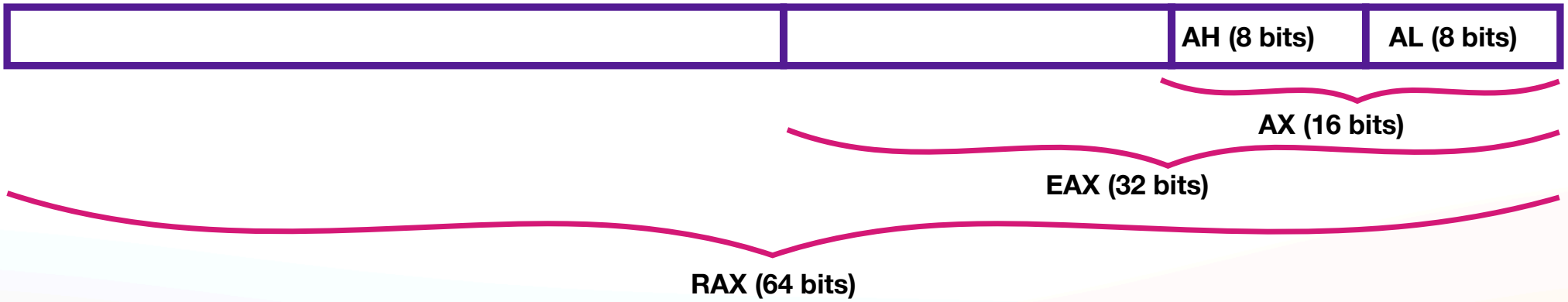
CISC (Complex Instruction Set Computer)**:**

- Enfoque en la facilidad de programación y flexibilidad.
- Ofrece un conjunto de instrucciones más grande y complejo.
- Puede realizar operaciones más completas en una sola instrucción.
- Las instrucciones CISC pueden requerir múltiples ciclos de reloj para ejecutarse.
- Menor cantidad de registros, lo que puede aumentar los accesos a memoria.
- A menudo, utiliza microcódigo para implementar instrucciones complejas.
- Ejemplos de arquitecturas CISC incluyen x86 (Intel), SPARC (Oracle), y algunas versiones de PowerPC.

Registros X86 vs IA64



Tamaño de registros de propósitos generales X86 e IA64



Lenguaje ENSAMBLADOR

El Lenguaje Ensamblador (Assembly Language) es un lenguaje de bajo nivel pero que permite programar a nivel de instrucciones de procesador sin tener que hacerlo de bit en bit.

El primer lenguaje ensamblador fue creado por Maurice V. Wilkes en 1950.

Maurice V. Wilkes: (1913-2010) fue un influyente científico de la computación británico conocido por su destacada contribución en el desarrollo de la primera computadora digital de programa almacenado, conocida como EDSAC (Electronic Delay Storage Automatic Calculator), que se puso en funcionamiento en 1949 en la Universidad de Cambridge.

Desempeñó un papel crucial en la popularización del concepto de programas almacenados, que es fundamental para las computadoras modernas. También introdujo la idea de las "*subrutinas*", que permiten la reutilización de código y una estructuración más eficiente de los programas.

Assembly X86

El ensamblador x86 es un lenguaje de bajo nivel utilizado para programar directamente en la arquitectura x86, que es ampliamente utilizada en computadoras personales y servidores.

Instrucciones: El ensamblador x86 opera a nivel de instrucciones de máquina del procesador. Cada instrucción ensamblador se corresponde con una operación específica que puede realizar el procesador, como movimientos de datos, operaciones aritmético-lógicas, saltos condicionales e incondicionales, llamadas a subrutinas, entre otras.

Registros: En el ensamblador x86, se manejan registros de propósito general como EAX, EBX, ECX, EDX, entre otros. Estos registros pueden contener datos temporales, direcciones de memoria y resultados de operaciones.

Memoria: Se pueden utilizar direcciones de memoria directamente en las instrucciones para leer o escribir datos en la memoria principal.

Directivas de ensamblador: Además de las instrucciones de máquina, el ensamblador x86 también incluye directivas que ayudan al ensamblador a compilar el código. Estas directivas pueden ser utilizadas para definir constantes, reservar espacio en memoria, especificar segmentos de código y datos, entre otras cosas.

Assembly X86 - Programa y datos en Memoria

PROGRAMA

```
...
mov EAX,[n]
add EAX, 1
...
```

MEMORIA

...								
i	0	0	0	1	0	0	0	0
i+1	0	0	0	0	0	0	1	0
i+2	0	0	0	0	0	0	1	0
i+3	0	0	1	1	0	0	1	1
i+4	0	0	0	0	0	0	0	0
i+5	0	0	0	0	0	0	0	0
i+6	0	0	0	0	0	0	0	0
i+7	0	0	0	0	0	0	0	1
...								
n	0	0	0	0	0	0	0	1
n+1	0	0	0	0	0	0	1	1
n+2	0	0	0	0	0	1	1	1
n+3	0	0	0	0	1	1	1	1
...								

Programa

Datos del programa

¿ Cómo podemos reservar/definir espacio de memoria (con su tamaño) para almacenar los datos ?

UNIDAD	LETRA
Byte	**B**
Palabra (word)	**W**
Palabra Doble (Double)	**D**
Palabra Cuádruple (Quad)	**Q**
Diez bytes (Ten)	**T**

Ejemplo: `n db 1`

declaro/reservo **1(uno) Byte** para la etiqueta **n**

Assembly X86 - Estructura de programa NASM

Archivos de programa escrito en assembly NASM

```
;estructura de programa
segment .data
   L1 db 8
   L2 db "hola mundo",0
   L3 dw Ah,1001b
   L4 times 10 db 0
   …

segment .bss
   L4 resb 10
   …

segment .text
   xor eax, eax
   mov ax, [L3]
   mov ebx, L4
   …
```

los comentarios comienzan " **;** "

declaro **L1** como un byte inicializado con 8

declaro **L2** como una secuencia de 11 bytes conteniendo "`h,o,l … 0`". **TODA cadena de texto debe finalizar con un byte "0".** Si accedemos por ejemplo a `[L2 + 3]` obtenemos "`a`".

¿cómo accedo al 9?

TIMES me permite repetir, por ej L4 apunta a 10 bytes inicializados con 0

reservar espacios sin inicializar

reserva 10 (diez) espacios de 1 byte "`resb`", `L4` apunta al primer lugar reservado.

Instrucciones del programa

Assembly X86 - Algunas directivas

Archivos de programa escrito en assembly NASM

```
%include "asm_io.inc"

%define SIZE 100

segment .bss
   L4 resb SIZE
   …

segment .text
   xor eax, eax
   mov [L3], SIZE
   mov ebx, SIZE
   …
```

Nos permite incluir otros archivos de código fuente

Nos permite declara constantes, asociar valores a nombres

Luego las constantes las puedo utilizar en las declaraciones de datos

También se pueden utilizar las constantes en el código del programa

Assembly X86 - Instrucciones (ISA)

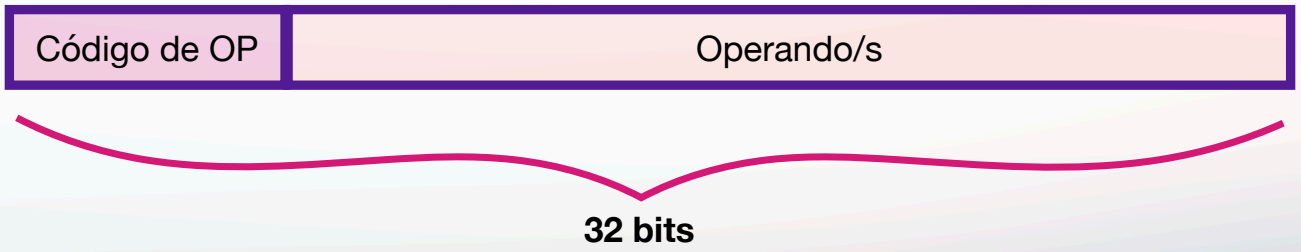
En general todos los procesadores proveen instrucciones para:

- **Transferencia de Datos**: Facilitan la transferencia de información entre la memoria y los registros del procesador, lo que es vital para manejar los datos utilizados en los cálculos y las operaciones.
- **Operaciones Aritmético-Lógicas**: Además de las operaciones básicas de suma, resta, multiplicación y división, los procesadores modernos, en especial, incluyen instrucciones para operaciones más avanzadas, como manipulaciones de bits y operaciones de multimedia, ampliando su capacidad.
- **Control de Programa**: Permiten controlar el flujo de ejecución del programa, mediante instrucciones de salto y ramificación (branching), lo que resulta crucial para implementar bucles y estructuras de control.
- **Instrucciones de Entrada/Salida**: Incluyen instrucciones para gestionar operaciones de entrada/salida (E/S), lo que permite que el procesador interactúe con dispositivos externos como teclados, pantallas, discos duros, y otros periféricos.

Assembly X86 - Instrucciones (ISA)

Una instrucción es una de la posibles operaciones que el procesador puede realizar (hardware). Cada una de ella tiene un único Código de Operación.

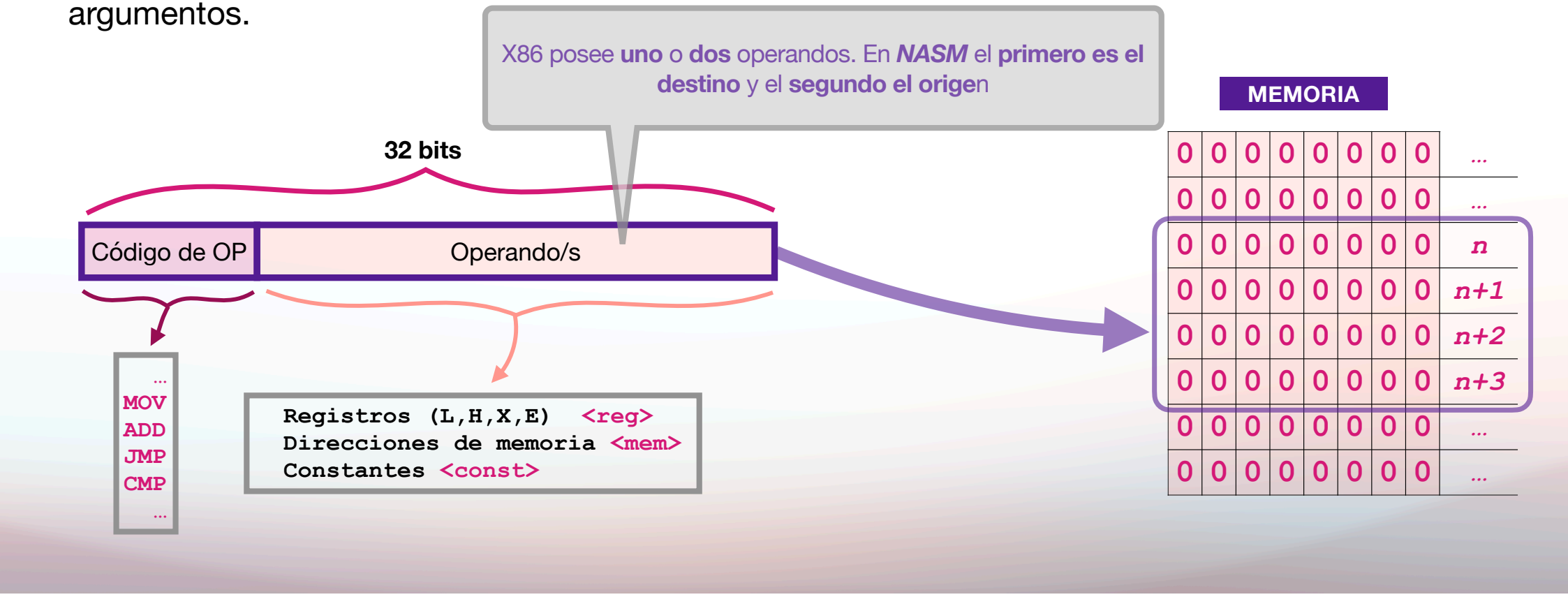
Un dato importante a tener en cuenta es que, dado que los programas deben estar en la memoria, cada procesador tiene su diseño de cómo se codifica una operación con sus argumentos.



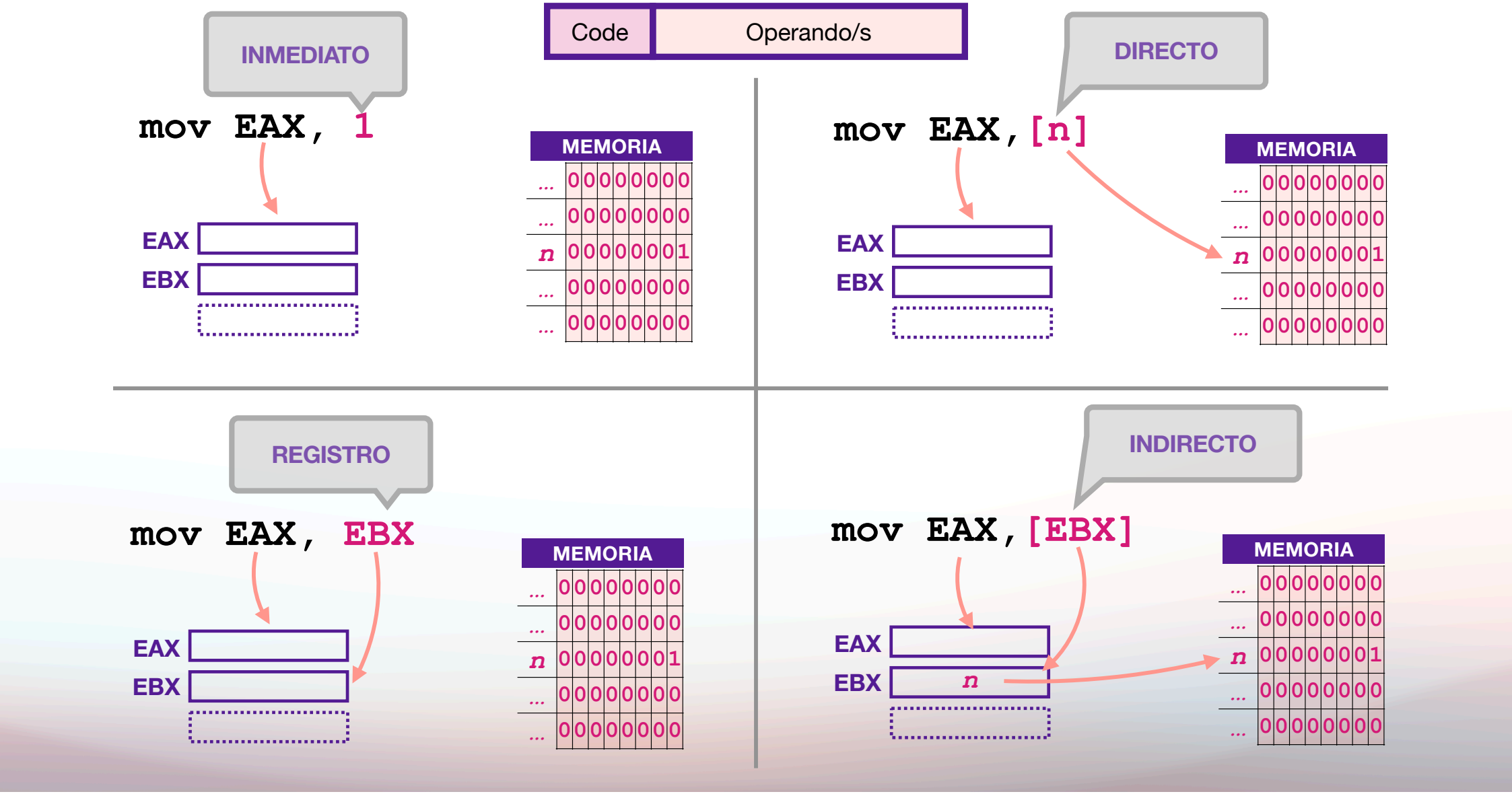
Assembly X86 - Instrucciones (ISA)

Una instrucción es una de la posibles operaciones que el procesador puede realizar (hardware). Cada una de ella tiene un único Código de Operación.

Un dato importante a tener en cuenta es que, dado que los programas deben estar en la memoria, cada procesador tiene su diseño de cómo se codifica una operación con sus argumentos.



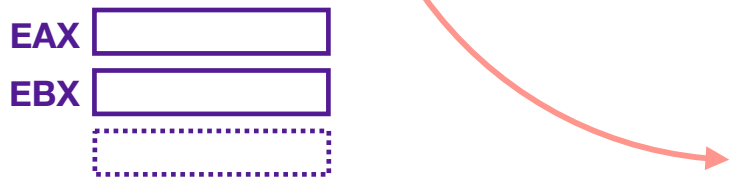
Assembly X86 - Operandos



Assembly X86 - Operandos - acceso indexado

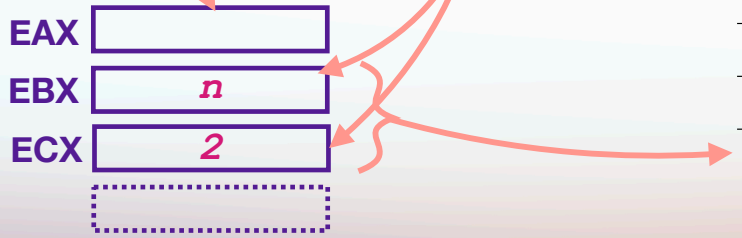
Code	Operando/s

```
mov EAX,[n+2]
```



MEMORIA								
...	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	1
n+1	0	0	0	0	0	0	0	0
n+2	0	0	0	0	0	0	1	0

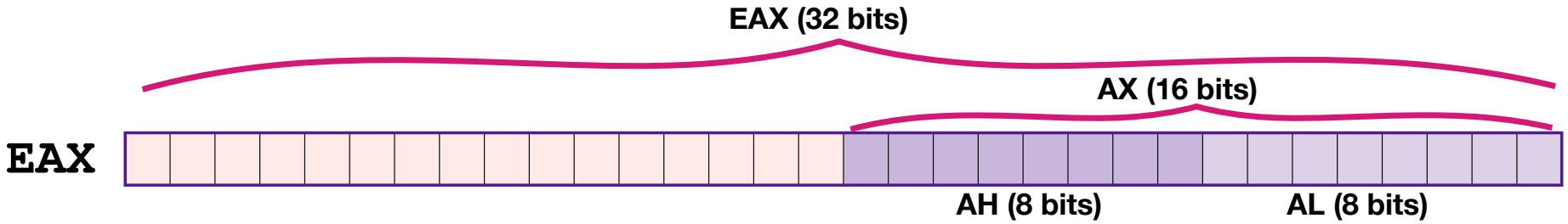
```
mov EAX,[EBX+ECX*2]
```

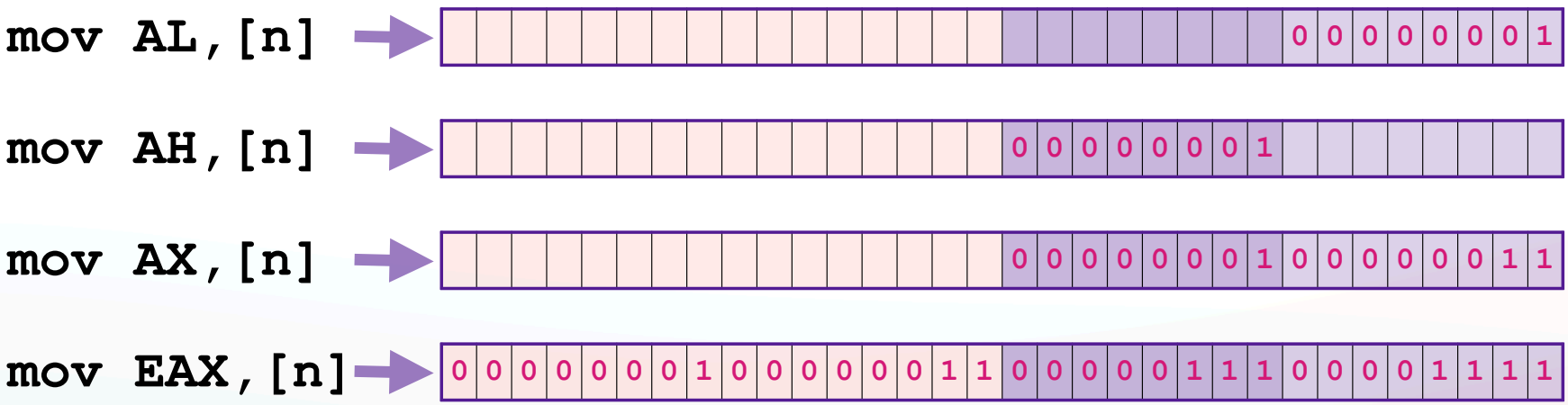


MEMORIA								
n	0	0	0	0	0	0	0	0
n+1	0	0	0	0	0	0	0	0
n+2	0	0	0	0	0	0	0	1
n+3	0	0	0	0	0	0	0	0
n+4	0	0	0	0	0	1	0	0

Generalmente se utiliza para el acceso a **arreglos**

Assembly X86 - acceso a Memoria y Tamaños





MEMORIA								
...	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	1
n+1	0	0	0	0	0	0	1	1
n+2	0	0	0	0	0	1	1	1
n+3	0	0	0	0	1	1	1	1
n+4	0	0	0	0	0	0	0	0
n+5	0	0	0	0	0	0	0	0
n+6	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	0
...	0	0	0	0	0	0	0	0

El **tamaño** del dato traído desde la memoria **depende del operando de destino**.

Assembly X86 - instrucciones aritmético-lógicas

```
add <destino> <valor> ;suma
sub <destino> <valor> ;resta
inc <destino> ;incremento
dec <destino> ;decremento

imul <destino> <valor> ;multiplicación
mul <valor> ;multiplicación
div <destino> <valor> ;división
idiv <valor> ;división
```

combinaciones de tipos de operadores:
<reg>,<reg>
<reg>,<mem>
<mem>,<reg>
<reg>,<const>
<mem>,<const>

Dado que los resultados pueden cambiar de tamaño existen funcionan con diferentes combinaciones y con convenciones (supuestos) acerca del destino de los resultados .

Por ejemplo **imul**, toma dos operando y deja el resultado en el primero. Pero si necesitamos multiplicar enteros de 32 bits, debemos utilizar **mul**, que supone un operando está en el **EAX** y sólo indicamos el segundo operando. El resultado queda almacenado en dos registros **EDX:EAX**

Assembly X86 - instrucciones manipulación de bits

```
shr <reg> <const> ;shift a derecha
shl <reg> <const> ;shift a izquierda
sar <reg> <const> ;shift aritmético a derecha
sal <reg> <const> ;shift aritmético a izquierda
ror <reg> <const> ;rotación a derecha
rol <reg> <const> ;rotación a izquierda
```

Assembly X86 - instrucciones aritmético-lógicas

```
and <destino> <valor> ; "y"
or <destino> <valor> ; "o"
xor <destino> <valor> ; "o exclusivo"
not <destino> ; "no"
```

Assembly X86 - Etiquetas

Las etiquetas se pueden utilizar para **identificar un espacio de la memoria** reservada para almacenar algún valor o para **referenciar instrucciones dentro de un programa**.

En ambas situaciones las etiquetas **representan una dirección de memoria** donde se almacena un valor, o donde está la instrucción del programa.

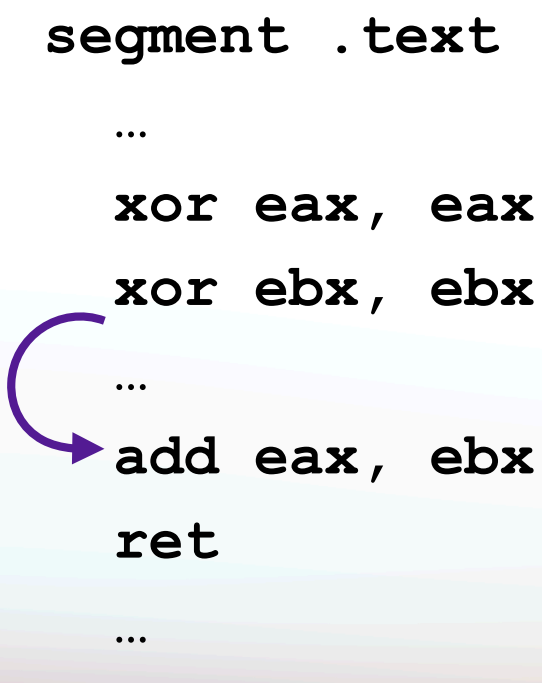
Estas etiquetas son reemplazadas por sus respectivas direcciones en el proceso de compilación.

```
segment .data
  L1 db,0
```

```
segment .text
  ...
Salir:
  ret
  ...
```

Assembly X86 - Saltos

¿**cómo podemos alterar** el **secuenciamiento** por defecto que indica *que la próxima instrucción a ejecutar es la siguiente*?



Assembly X86 - Saltos

¿**cómo podemos alterar** el **secuenciamiento** por defecto que indica *que la próxima instrucción a ejecutar es la siguiente*?

```
segment .text
  …
  xor eax, eax
  xor ebx, ebx
  …
  add eax, ebx
  ret
  …
```

Saltos a etiquetas

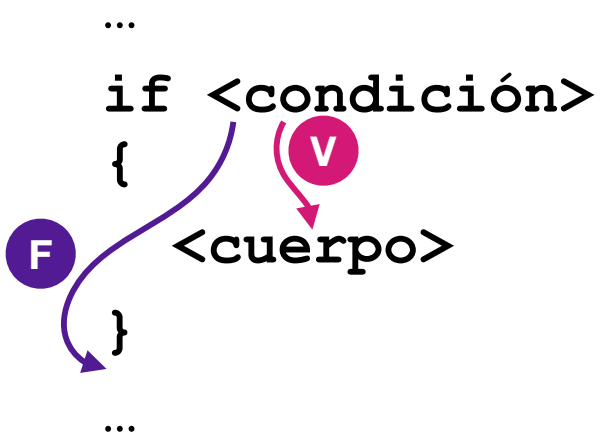
```
segment .text
  …
  xor eax, eax
  xor ebx, ebx
  jmp salir
  …
salir:
  add eax, ebx
  ret
  …
```

En el proceso de compilación las etiquetas son reemplazadas por las direcciones de memoria donde se carga la instrucción correspondiente

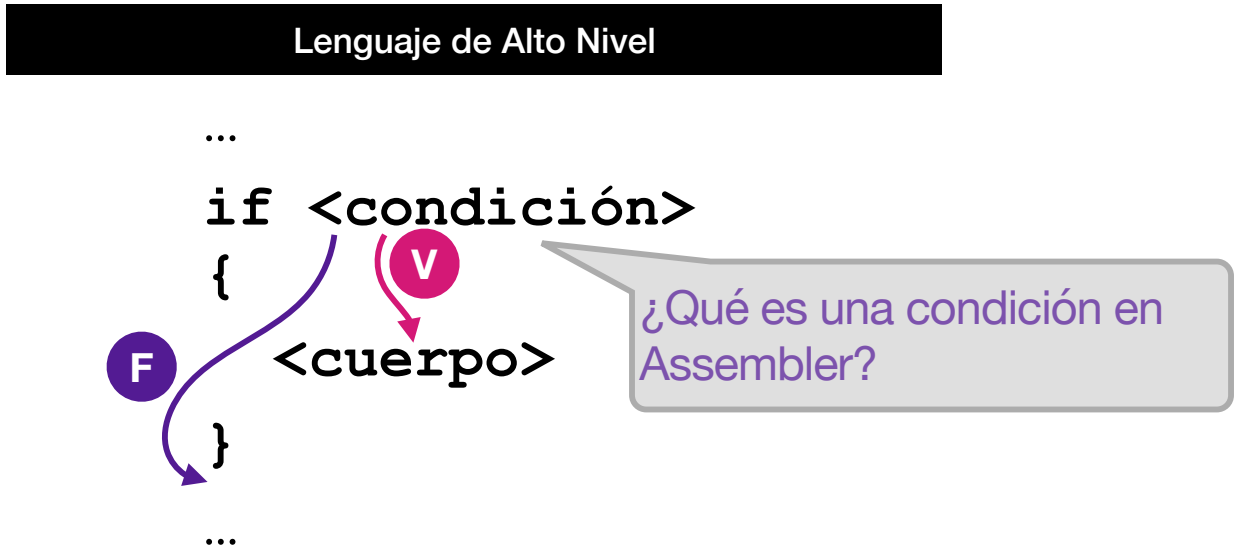
La instrucción `JMP` (jump), **modifica el valor del PC** (program counter) con la *dirección de memoria de la instrucción "etiquetada"*

Cómo simular estructuras de control de lenguajes de alto nivel

Lenguaje de Alto Nivel



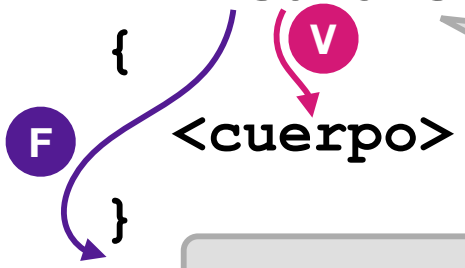
Cómo simular estructuras de control de lenguajes de alto nivel



Cómo simular estructuras de control de lenguajes de alto nivel

Lenguaje de Alto Nivel

```
...
if <condición>
{
    <cuerpo>
}
...
```



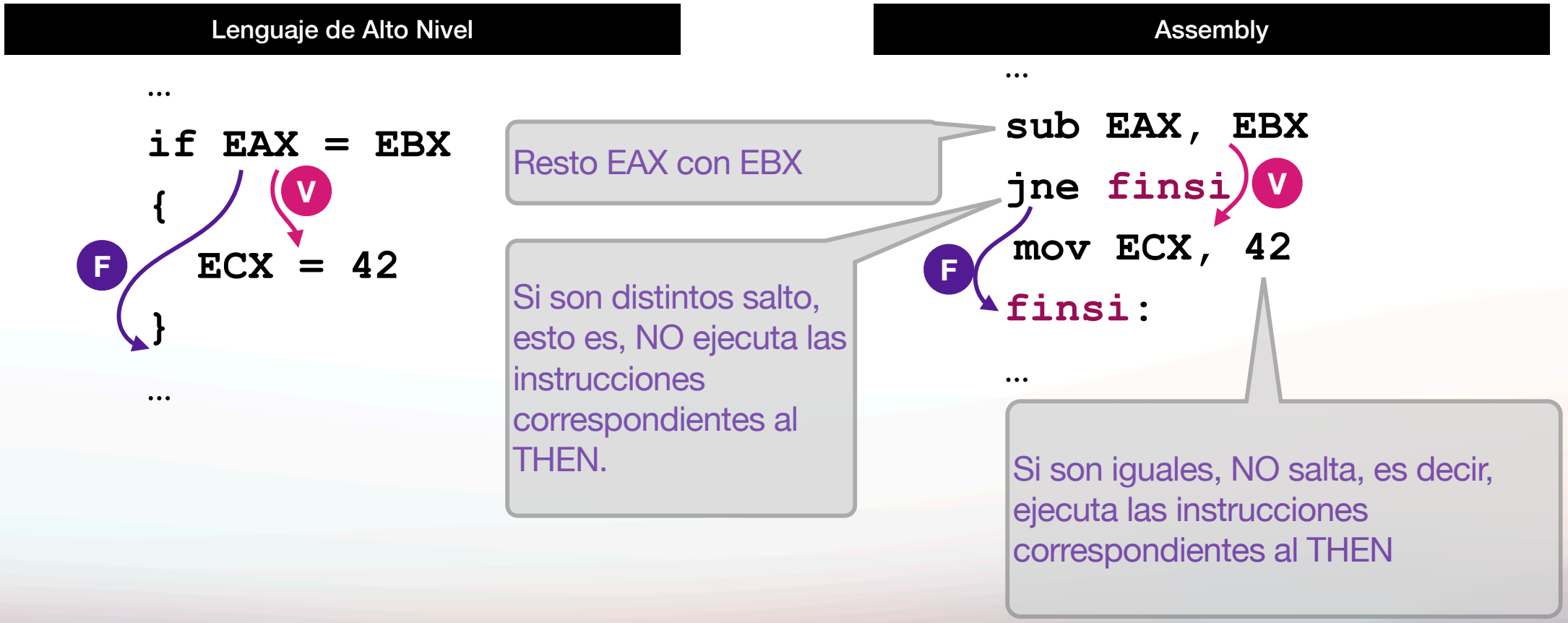
¿Qué es una condición en Assembler?

Tenemos **el registro de FLAGs** (que se actualiza luego de cada instrucción) y una serie de saltos que dependen de sus valores.

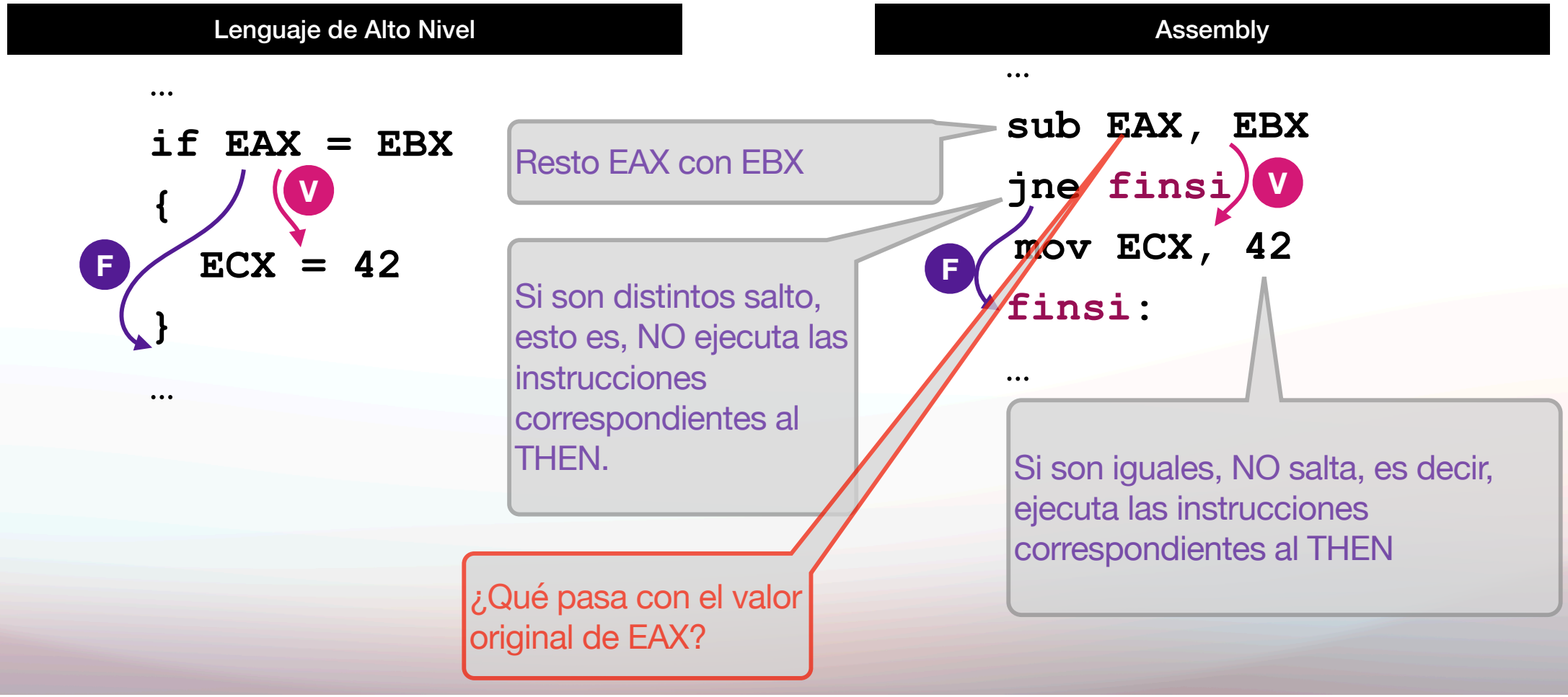
- `je` salta si es igual
- `jne` salta si NO es igual
- `jz` salta si el último resultado fue cero
- `jg` salta si es mayor
- `jge` salta si es mayor o igual
- `jl` salta si es menor
- `jle` salta si es menor o igual
- `jo` salta si hubo overflow
- `jno` salta si NO hubo overflow
- `js` salta si el FLAG de signo está encendido
- `jns` salta si el FLAG de signo NO está encendido
- ...

Para determinar cuándo realizar el salto, las instrucciones analizan el estado de los FLAGs (combinaciones de ellos). Por ejemplo `je` revisa si el FLAG de cero (`ZF`) está encendido.

Cómo simular estructuras de control de lenguajes de alto nivel



Cómo simular estructuras de control de lenguajes de alto nivel



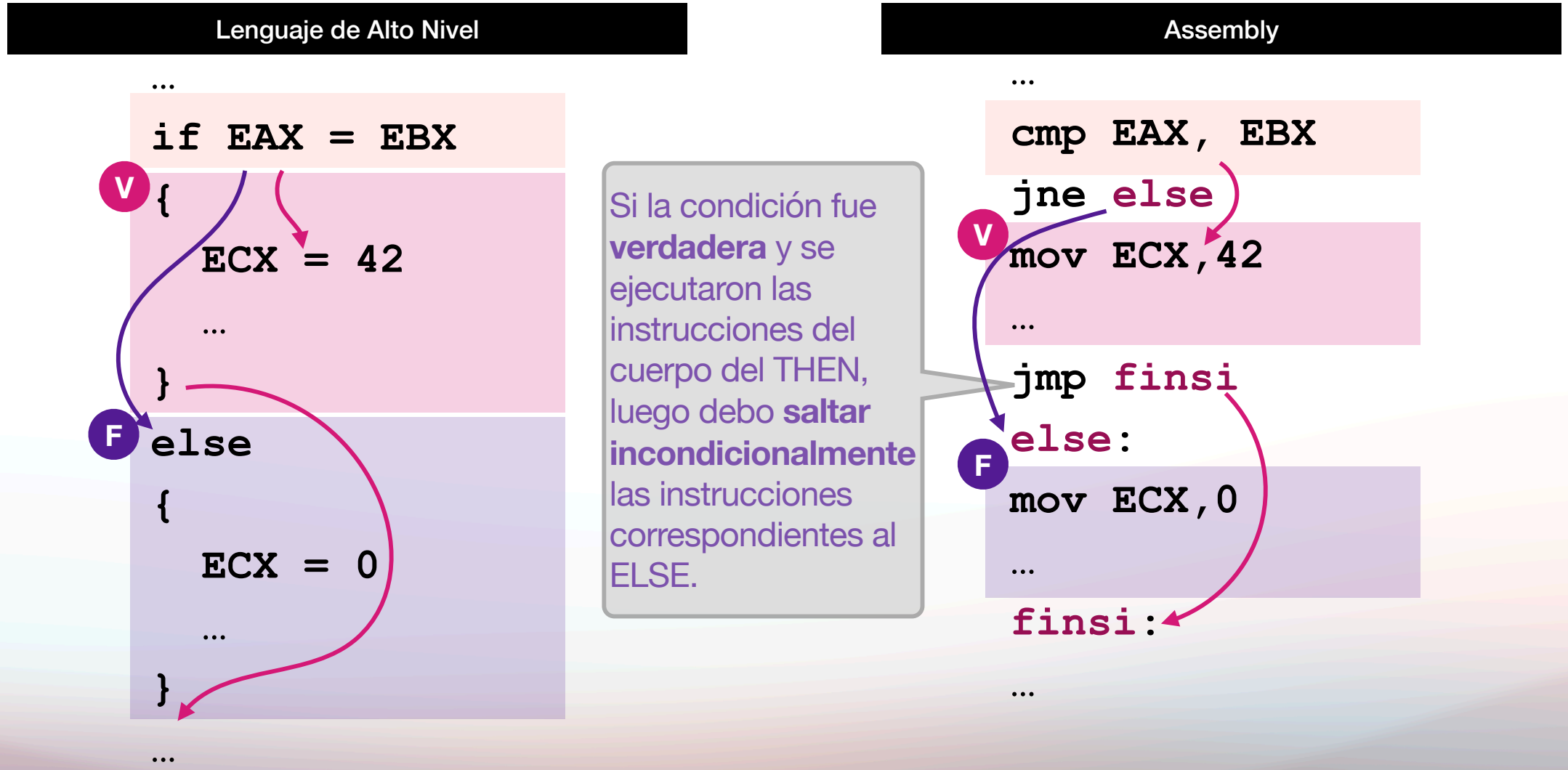
Cómo comparar sin destruir y cómo construir

Considerando que los saltos condicionales se basan en los estados del registro de FLAGS, que se actualizan tras operaciones específicas, y dado que la evaluación de condiciones NO debe modificar los valores de los elementos bajo observación, se introduce la instrucción de comparación **CMP**.

Esta instrucción realiza una resta entre sus operandos (como si fuese SUB) actualizando los valores del registro de FLAGs, pero SIN alterar los valores de dichos operandos.

```
...
cmp EAX, EBX
jne finsi
 mov ECX, 42
finsi:
...
```

Cómo simular estructuras de control de lenguajes de alto nivel



Cómo evaluar condiciones complejas

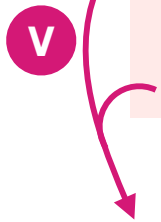
Lenguaje de Alto Nivel

```
...
if (EAX=EBX) or (EAX=ECX)
...
```

¿Cómo podemos evaluar condiciones más complejas?

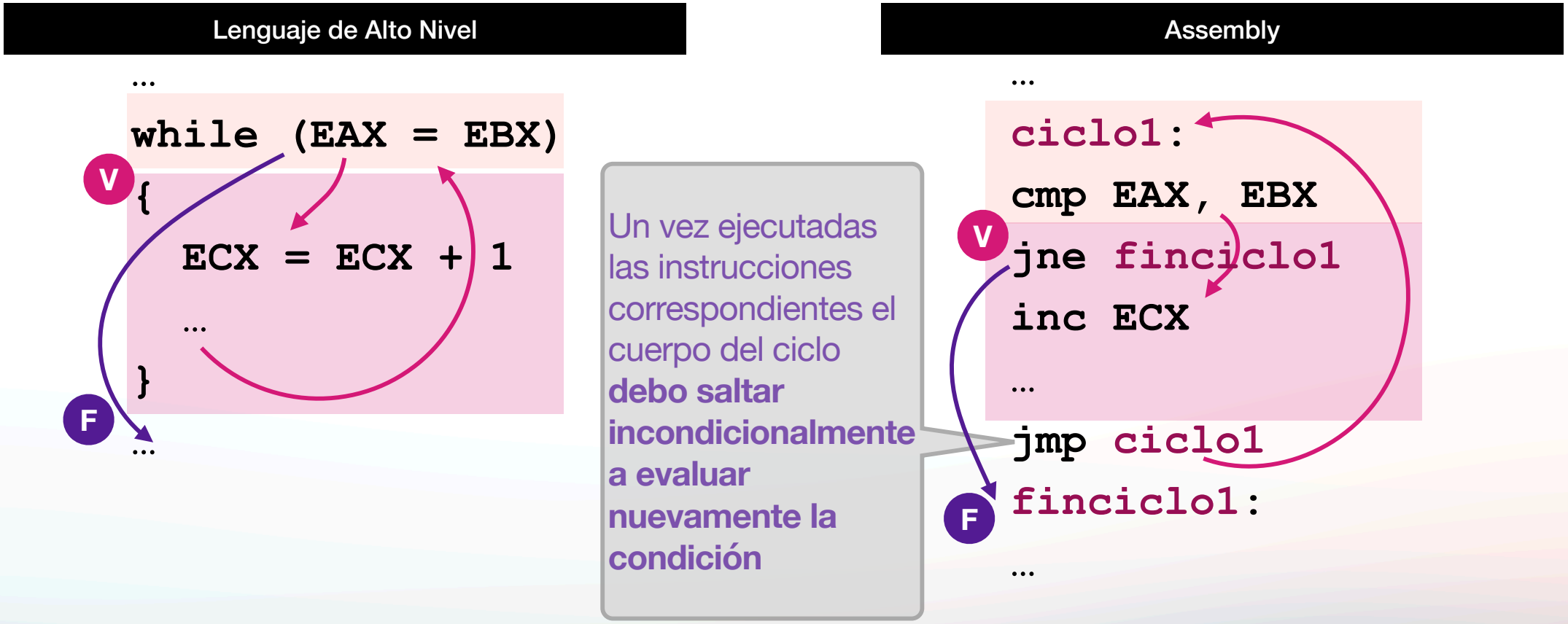
Assembly

```
...
cmp EAX, EBX
je then
cmp EAX, ECX
je then
...
```



Debemos ir evaluando **cada parte de la condición de manera independiente** e ir saltando dependiendo de la condición que debamos satisfacer.

Cómo simular estructuras de control - cliclos



Cómo simular estructuras de control - ciclos con repeticiones predefinidas

Lenguaje de Alto Nivel

```
...
for i desde 1 hasta 10
{
   EAX = EAX + 1
   ...
}
...
```

Si bien podemos simularlo con un *clico condicional evaluando y actualizando* el valor de `i`, podemos utilizar una instrucción particular `loop`.

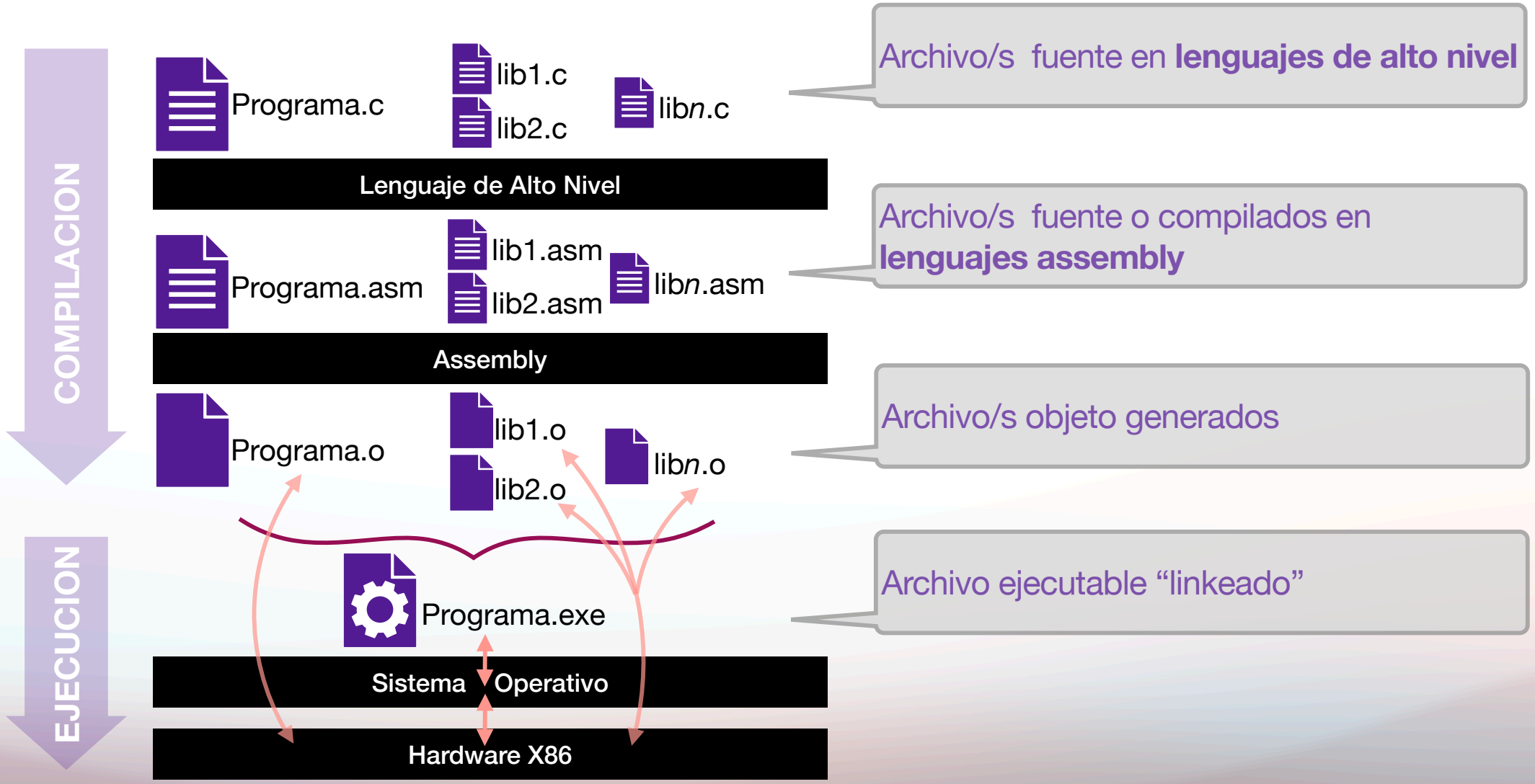
Assembly

```
...
mov ECX,10
for1:
inc EAX
...
loop for1
...
```

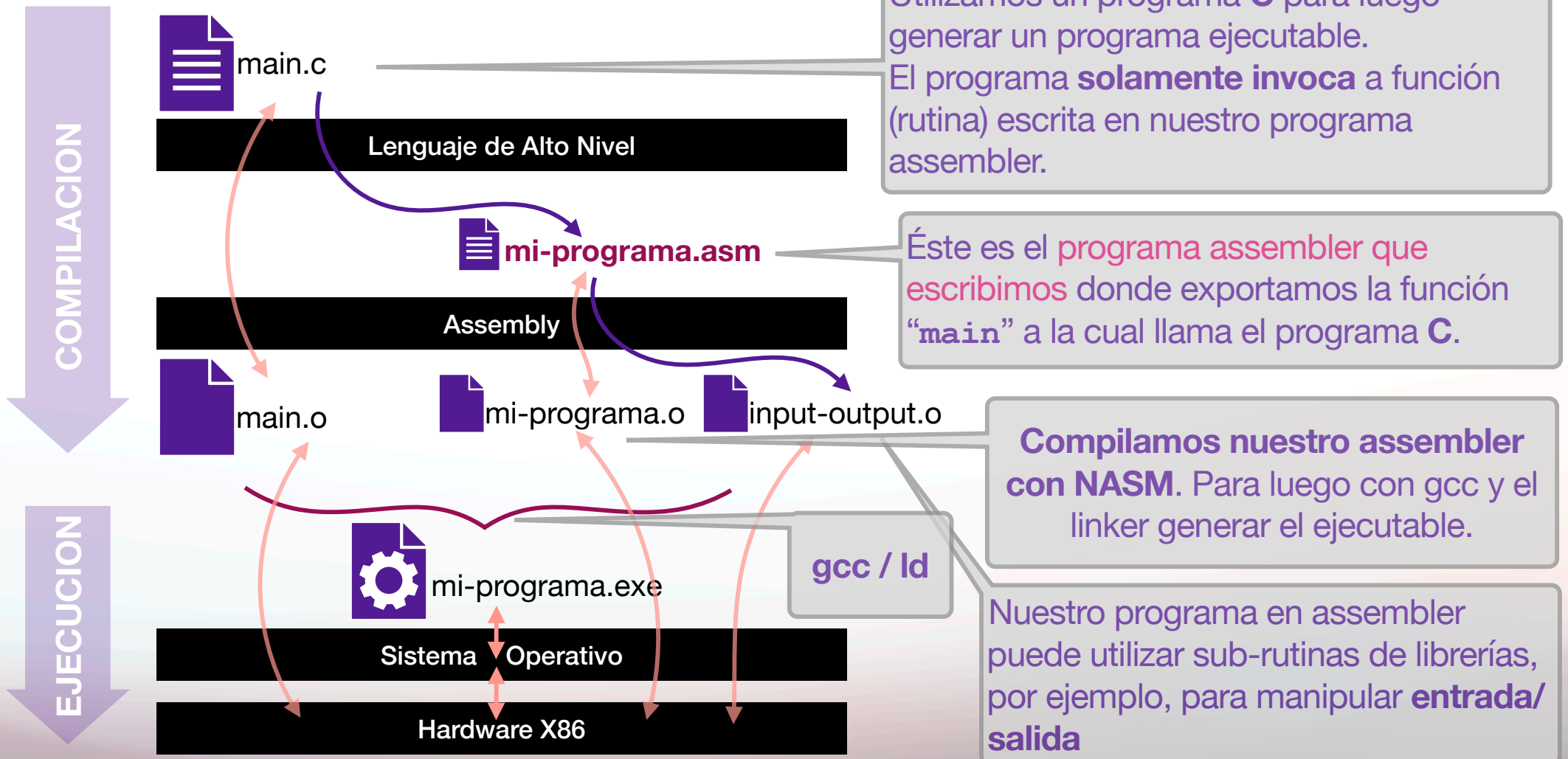
La instrucción `loop` utiliza el **registro `ECX` *como índice***. Debemos mover allí cuántas veces necesitamos ciclar.

Debemos etiquetar el comienzo de las instrucciones del cuerpo del ciclo, luego al final del mismo, indicar con la instrucción `loop` la etiqueta del comienzo.
El **valor del registro se actualiza automáticamente** y cuando ***llega a cero***, la instrucción deja de realizar el salto al comienzo. Podemos utilizar el valor de ECX, pero debemos tener en cuenta que se ***actualiza de manera decreciente***.

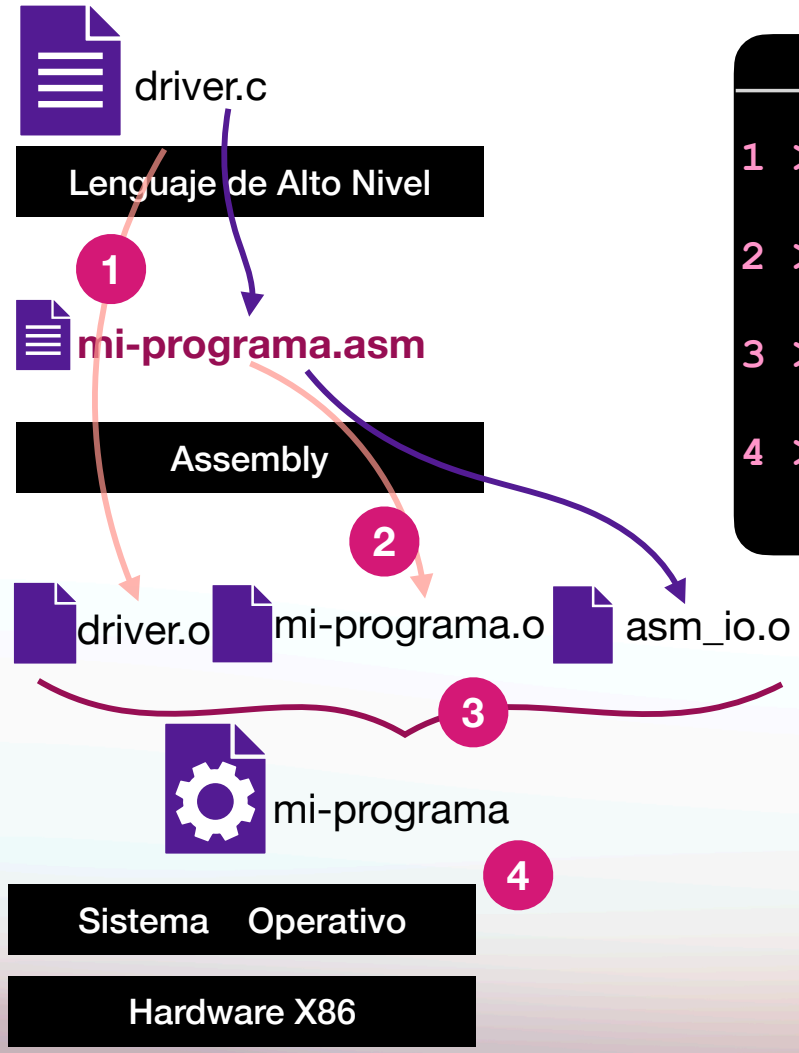
Practicando programar Assembly X86 - proceso general de compilación/ejecución



Practicando programar Assembly X86 - (compilación y ejecución “manual”)



Practicando programar Assembly X86 - (compilación y ejecución “manual”)

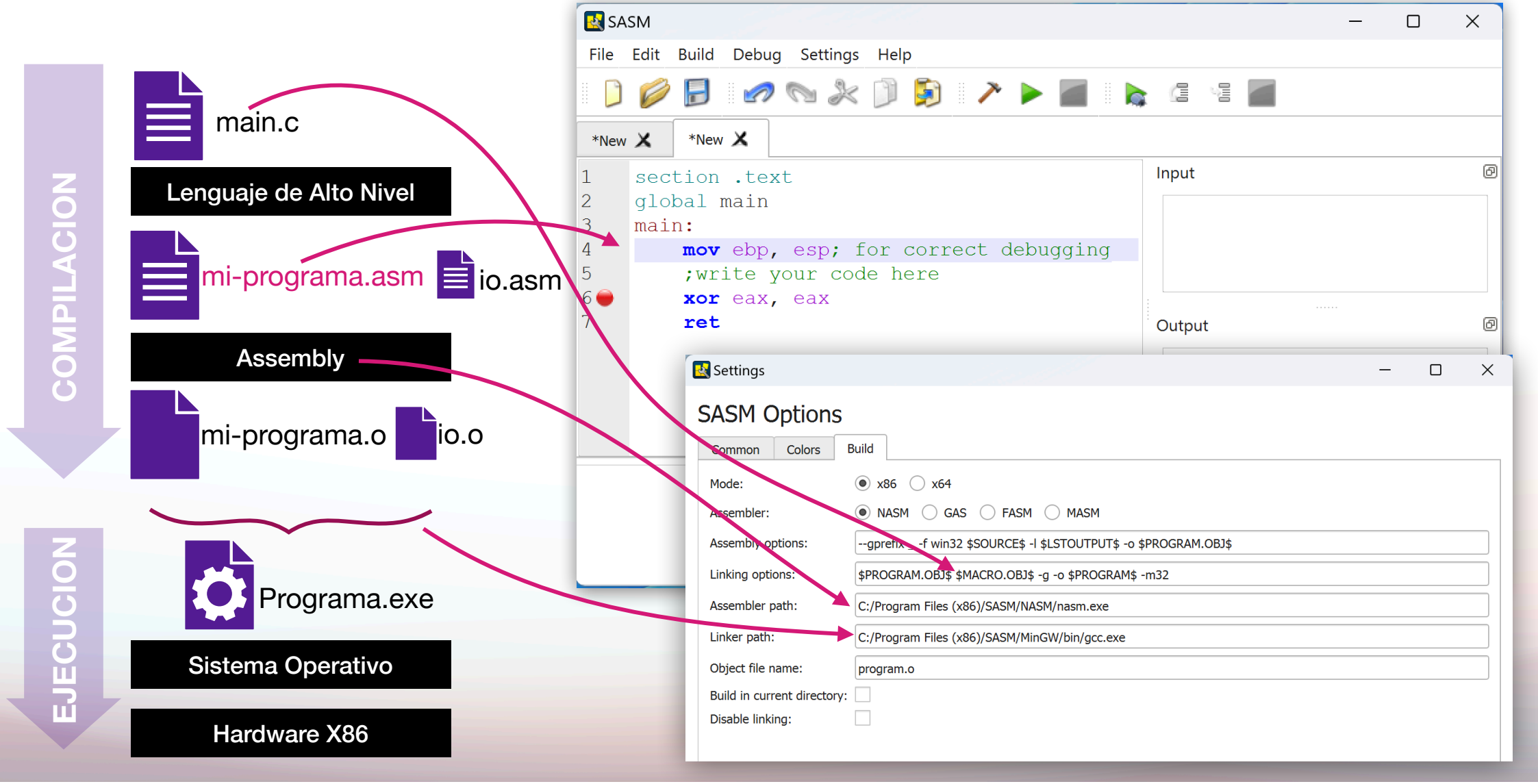


Consola

```
1 > gcc -c driver.c
2 > nasm -f formato-de-objeto mi-programa.asm
3 > gcc -o mi-programa driver.o mi-programa.o asm_io.o
4 > ./mi-programa
```

Windows: **COFF** (iCommon Objet File Format)
Linux: **ELF** (Executable an Linkable Format)

Practicando programar Assembly X86 (usando SASM “push-button”)



Practicando programar Assembly X86 (usando SASM “push-button”)

“Breakpoints” (puntos de pausa)

Ejecución en modo “depuración”.

Acceso a información de etiquetas o direcciones de memoria

interfaz de entrada de información

visualización de salida

Monitor de actividad

Acceso a los valores de todos los registros de procesador con la visualización configurable (binario, hexadecimal, etc.)

Comandos específicos de depuración