



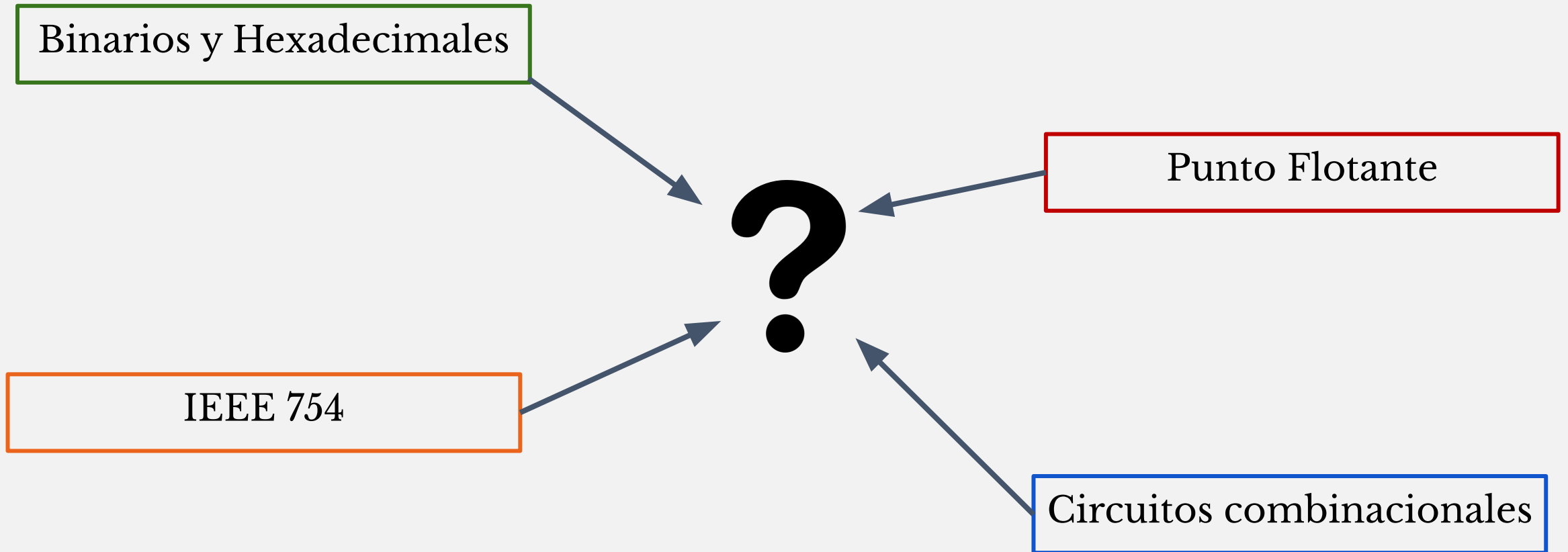
# Organización de computadoras



Lenguaje  
ensamblador

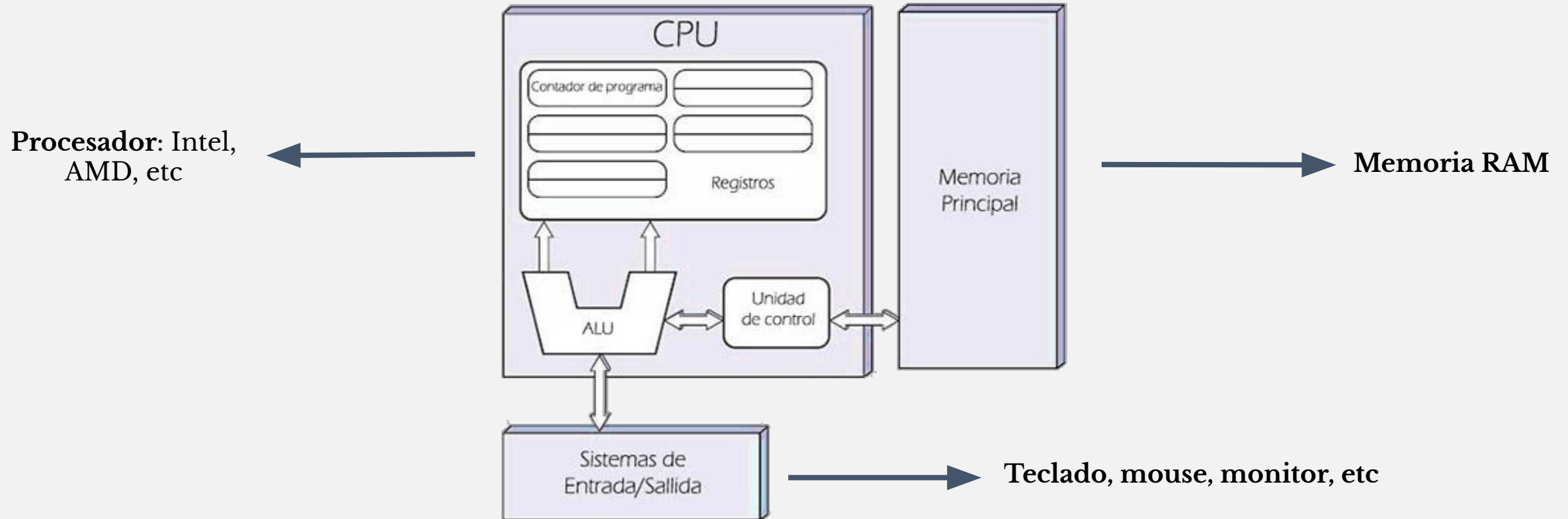
# Esquema global

*Lo que vimos hasta ahora*



# Esquema global

*Arquitectura de Von Neumann*



# Esquema global

*Arquitectura de Von Neumann*

Ahora...



Programamos en un lenguaje de alto nivel (Pascal,  
Python, Ruby, Java, etc)

Los procesadores solo entienden  
binario

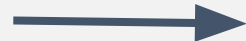


# Compiladores

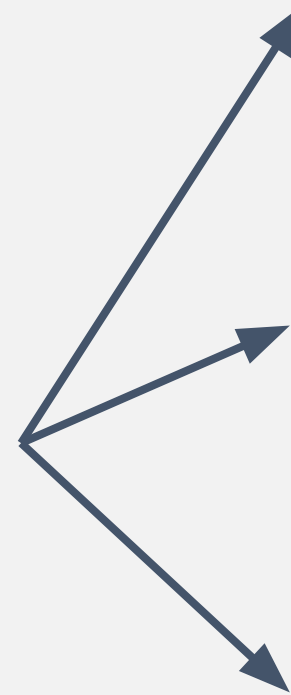
El compilador se encarga de generar código que la máquina entiende (binario) a partir de nuestro código de alto nivel



```
....  
Begin  
    num := 5;  
end  
...
```



Compilador o  
Intérprete



1100111010...

0010101011...

1110110000...



# Assembler

El **Lenguaje Ensamblador**, también llamado **Assembler** o **Assembly**. Es otro lenguaje de programación común y corriente. Pero...

1. Es de bajo nivel y es un mapeo directo de las instrucciones de binario
2. Cada procesador tiene el propio



Assembler

*mov num, 5*

Compilación



Cód. Máquina

*1100111010...*



*move num, 5*



*0010101011...*



*store 5, num*



*1110110000...*



Máquinas de  $N$   
direcciones



# Práctica 5

## *Ejercicio 1*

Por cada **dirección** puedo utilizar una **variable**.

- De la práctica: escribir el programa que implemente  $F = [(A + B) / C] - D$
- Escribir en máquinas de 1, 2 y 3 direcciones.

<u>Nombre</u>	<u>Tamaño</u>	<u>Valor</u>
A:	1 byte	6
B:	1 byte	4
C:	1 byte	2
D:	1 byte	1
F:	1 byte	?

# Práctica 5

## *Ejercicio 1*

<u>Nombre</u>	<u>Tamaño</u>	<u>Valor</u>
---------------	---------------	--------------

A:	1 byte	6
----	--------	---

B:	1 byte	4
----	--------	---

C:	1 byte	2
----	--------	---

D:	1 byte	1
----	--------	---

F:	1 byte	?
----	--------	---

Las operaciones son:

- load: carga la variable en contexto (para poder usarla)
- add, sub, div: operación de suma, resta y división
- store: almacena del contexto a una variable

$$F = [(A + B) / C] - D$$

### Maq. de 1 dirección

load A  
add B  
div C  
sub D  
store F

### Maq. de 2 direcciones

mov F, A  
add F, B  
div F, C  
sub F, D

### Maq. de 3 direcciones

add F, A, B  
div F, F, C  
sub F, F, D

# Práctica 5

## *Ejercicio 2*

Maq. de 1 dirección	Maq. de 2 direcciones	Maq. de 3 direcciones
load A add B div C sub D store F	mov F, A add F, B div F, C sub F, D	add F, A, B div F, F, C sub F, F, D

Suponiendo que cada cód. de operación ocupa 6 bits y las dir. son de 10 bits.

	M. de 1 dirección	M. de 2 direcciones	M. de 3 direcciones
<b>Tam del prog en mem (cod.operación + op)</b>	(5 x 6 bits) + (5 x 10 bits) 30 bits + 50 bits = 80 bits = 10 bytes	(4 x 6 bits) + (8 x 10 Bits) 24 bits + 80 bits = 104 bits = 13 bytes	(3 x 6 bits) + (9 x 10 bits) 18 bits + 90 bits = 108 bits = 13,5 bytes
<b>Cant de accesos a mem (instrucciones (MI) + operandos (MD))</b>	MI = 5 accesos MD = 5 accesos = 10 accesos	MI= 4 accesos MD= 1° inst, 2 accesos MD= 3 x 3 = 9 accesos = 15 accesos	MI= 3 accesos MD= 3 x 3 = 9 = 12 accesos



# Assembler



*MSX88*

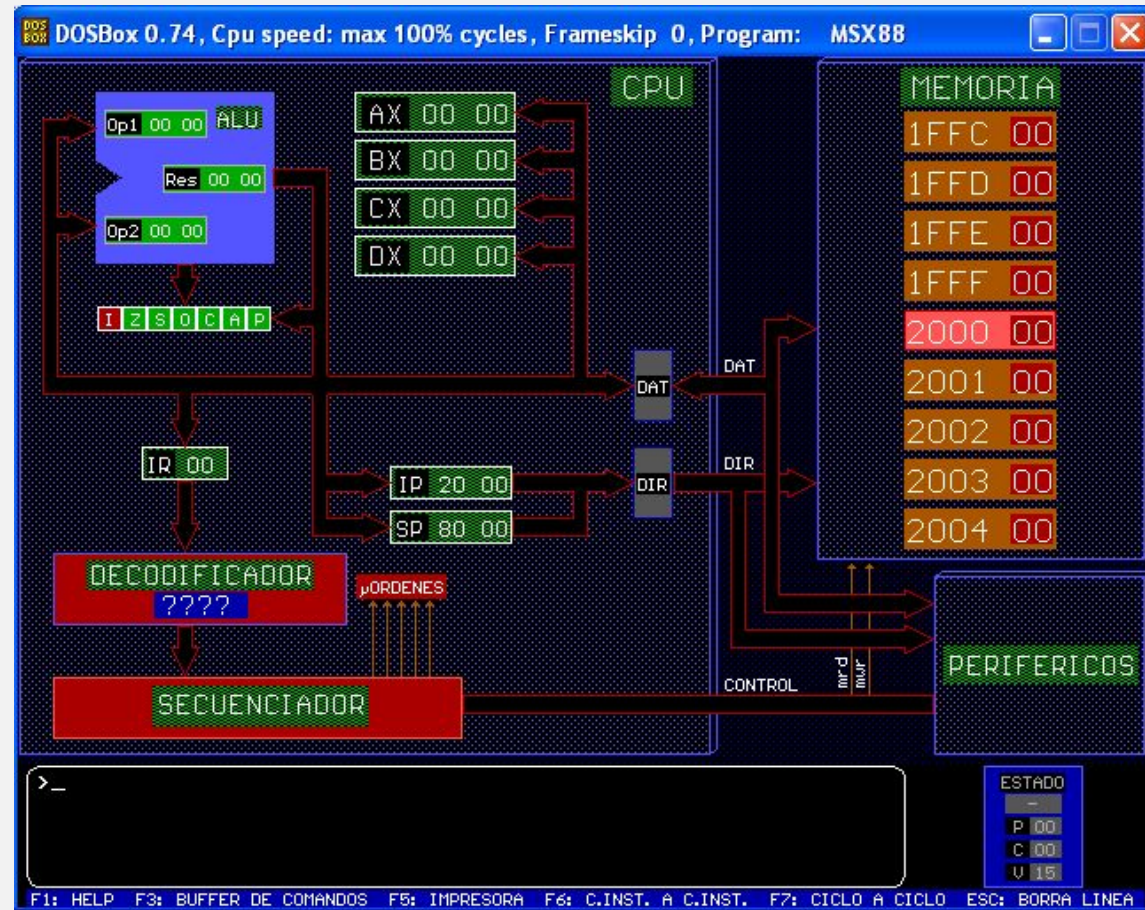
# Assembler

En esta cursada veremos el lenguaje ensamblador de los procesadores de la familia **Intel 8086**.

- Maneja operaciones de **2 direcciones**.
- Posee 4 registros: AX, BX, CX, DX
- Los números negativos se representan en **Ca2**
- Vamos a hacer uso del simulador **MSX88** (solo Windows) que se puede descargar en la plataforma de la materia.

# Assembler

MSX88



# Assembler

## *Almacenamiento*

Los datos pueden estar en memoria (variables) o en registros.

### Variables

<nombre> **DB** <valor inicial> (8 bits/1 byte)

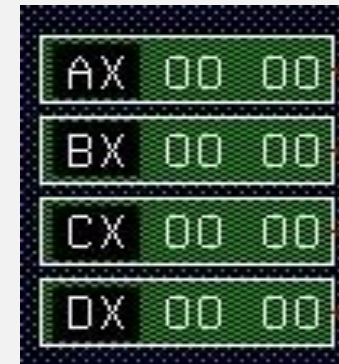
<nombre> **DW** <valor inicial> (16 bits/2 bytes)

### Ejemplos

var1	DB	10
var2	DW	? (No inicializado)
tabla	DB	1, 2, 3, 4, 5
string	DB	"Esto es un string" (ASCII)
var3	DW	0ABCDH (Hexadecimal)
var4	DB	10101010B (Binario)

### Registros

Son 4, cada uno tiene parte baja (L) y parte alta (H). Cada parte puede almacenar 1 byte





# Assembler

## *Memoria y posicionamiento*

Este procesador puede direccionar una memoria de 16 bits. Es decir, que tenemos  $2^{16}$  celdas de memoria que podemos utilizar.

- Por simpleza, la dirección de cada celda se pone en *Hexadecimal*.
- Cada celda de memoria puede almacenar 1 byte de datos

Cuando escribimos código **debemos indicar en qué posición de la memoria** queremos operar.

Esto se consigue con la sentencia **ORG**

### Sintaxis

*ORG <dirección de memoria>*

1 Byte →  
1 Byte →  
1 Byte →

MEMORIA	
1FFC	00
1FFD	00
1FFE	00
1FFF	00
2000	00
2001	00
2002	00
2003	00
2004	00

Valor de la celda

# Assembler

*Memoria y posicionamiento*

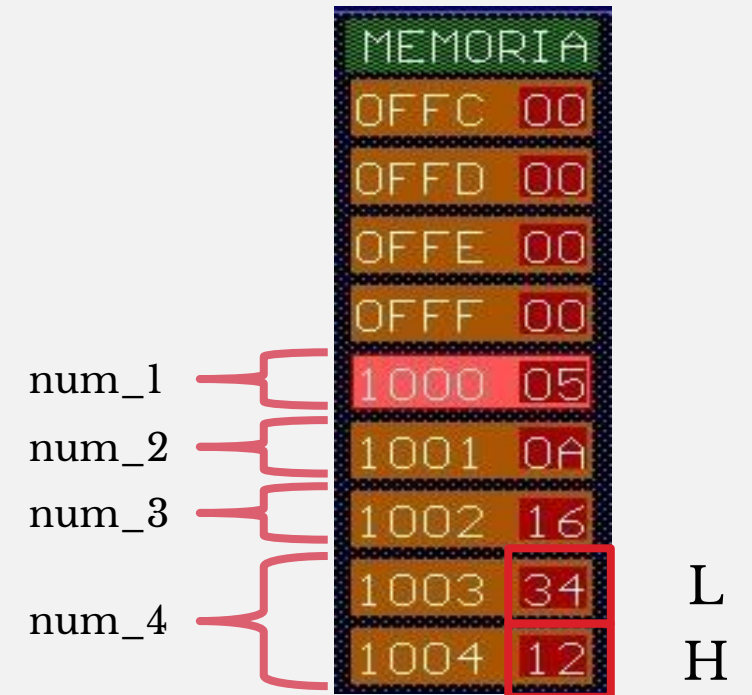
Usualmente definimos las variables a partir de la posición 1000H y el código de nuestro programa a partir de la posición 2000H

**ORG 1000H**

```
num_1 DB 5
num_2  DB 10
num_3  DB 016H
num_4  DW 01234H
```

**ORG 2000H**

```
... ; El código de mi programa principal
HLT ; Esta línea y la de abajo siempre deben ir
END
```



# Assembler

## *Transferencia*

La manera de transferir información es a partir de la sentencia **MOV**

### **ORG 1000H**

```
num_1 DB    5
num_2 DB    10
num_3 DB    016H
num_4 DW    01234H
```

### **ORG 2000H**

```
mov AL, num_1 ; Copio num_1 a la parte baja de AX
mov AH, num_2 ; Copio num_2 a la parte alta de AX
mov DX, num_4 ; Copio los 16 bits de num_4 al registro DX
mov CX, DX ; Copio el registro DX al registro CX
mov num_1, AH ; Copio el valor de AH a la variable num_1
mov num_1, num_2 ; PROHIBIDO
mov num_1, [BX] ; PROHIBIDO
```

Las operaciones donde ambos operandos acceden a memoria están **prohibidos**

# Assembler

*Modos de direccionamiento*

Hay 4 modos de direccionamiento que se pueden utilizar

## Inmediato

*mov AL, 9*

Se utiliza un valor fijo. No requiere acceso a memoria para obtener dicho valor

## Directo

*mov AL, num\_1*

Se requiere un acceso extra a memoria para tomar el valor de la variable

## Directo por registro

*mov AL, AH*

El valor del operando se encuentra en el registro indicado. No requiere acceso a memoria

## Indirecto por reg.

*mov BX, 1000H*  
*mov AL, [BX]*

El registro BX puede servir como puntero. Requiere un acceso extra a memoria.

# Assembler

*Direccionamiento indirecto por registro*

Indirecto por reg.

```
mov BX, 1000H  
mov AL, [BX]
```

El registro BX  
puede servir  
como puntero.  
Requiere un  
acceso extra a  
memoria.

Registros	
AX	...
BX	Dirección del operando
CX	...
DX	...



Memoria
...
operando
...

Se puede usar *OFFSET* para obtener la dirección de una variable  
Ej.: `mov BX, OFFSET num_1`

# Assembler

*Ejemplo*

**ORG 1000H**

NUM0 DB 0CAH  
NUM1 DB 0  
NUM2 DW ?  
NUM3 DW 0ABCDH  
NUM4 DW ?

**ORG 2000H**

MOV BL, NUM0  
MOV BH, 0FFH  
MOV CH, BL  
MOV AX, BX  
MOV NUM1, AL  
MOV NUM2, 1234H  
MOV BX, OFFSET NUM3  
MOV DL, [BX]  
MOV AX, [BX]  
MOV BX, 1006H  
MOV WORD PTR [BX], 0CDEFH

**HLT**

**END**

	AX	BX	CX	DX
L	<del>CA</del> CD	<del>CA</del> <del>04H</del> 06H		CD
H	<del>EF</del> AB	<del>EF</del> <del>10H</del> 10H	CA	

; Copia valor de NUM0 en BL

; Copia valor *FFH* en BH

; Copia BL en CH

; Copia BX en AX

; Copia AL en NUM1

; Copia el valor *1234H* en NUM2

; Copia la dir. de NUM3 en BX

; Copia en DL el contenido de la celda apuntada por BX

; Copia en AX el contenido de la celda apuntada por BX

; Copia el valor *1006H* en BX

; Copia el valor *0CDEFH* a la celda apuntada por BX

Memoria	
NUM0	1000H CA
NUM1	1001H <del>0</del> CA
NUM2	1002H 34H
	1003H 12H
NUM3	1004H CD
	1005H AB
NUM4	1006H EFH
	1007H CDH
	...
	2000H



# Assembler



*Aritmética y lógica*



# Assembler

*Aritmética*

Contamos con varias operaciones aritméticas

**ADD** OP1, OP2  $\longrightarrow$  OP1 := OP1 + OP2

**SUB** OP1, OP2  $\longrightarrow$  OP1 := OP1 - OP2

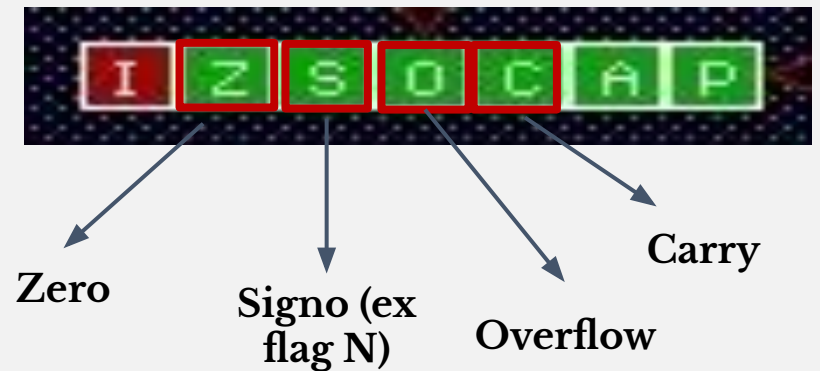
**ORG 1000H**

```
num_1 DW 5
num_2 DW 10
res    DW ?
```

**ORG 2000H**

```
MOV AX, num_1 ; Copio valor de num_1 a AX
ADD AX, num_2 ; Sumo el valor de num_2 a AX
MOV res, AX ; Guardo el resultado en la variable res
HLT
END
```

Estas operaciones setean los flags que conocemos!



Pregunta...

¿ADD num\_1, num\_2?

**NO!** Ambos operandos **no** pueden acceder a memoria!

# Assembler

*Aritmética*

**IMPORTANTE!** Cuando operamos no se tiene en cuenta el carry de operaciones anteriores. Por suerte contamos con:

**ADC** OP1, OP2  OP1 := OP1 + OP2 + **C**

**SBB** OP1, OP2  OP1 := OP1 - OP2 - **C**

Útil para operaciones que podrían “irse de rango”

Contamos con incremento/decremento

**INC** OP1  OP1 := OP1 + **1**

**DEC** OP1  OP1 := OP1 - **1**

# Assembler

*Aritmética*

**IMPORTANTE!** Hay casos en los que solo queremos consultar el estado de los flags sin afectar los operandos

**CMP** OP1, OP2  OP1 - OP2

**Solo** setea los flags. **No** modifica OP1

Veremos que esto resulta **indispensable** cuando veamos los saltos

# Assembler

*Lógica*

También tenemos las operaciones lógicas conocidas!

**AND** OP1, OP2      OP1 := OP1 **AND** OP2

**OR** OP1, OP2      OP1 := OP1 **OR** OP2

**XOR** OP1, OP2      OP1 := OP1 **XOR** OP2

**NOT** OP1      OP1 :=  $\neg$ OP1 (**Ca1**)

**NEG** OP1      OP1 :=  $\neg$ OP1 (**Ca2**)

Todas estas operaciones también setean los flags!

# Assembler

*Ejemplo*

**ORG** 1000H

NUM0 DB 80H

NUM1 DB 200

NUM2 DB -1

BITS0 DB 01111111B

BITS1 DB 10101010B

**ORG** 2000H

MOV AL, NUM0

ADD AL, AL

INC NUM1

MOV BH, NUM1

MOV BL, BH

DEC BL

SUB BL, BH

MOV CH, BITS1

AND CH, BITS0

NOT BITS0

OR CH, BITS0

**HLT**

**END**

	AX	BX	CX	DX
L	<del>80H</del> 00H	<del>C9H</del> <del>C8H</del> FFH		
H		C9H	<del>AA</del> <del>2A</del> AA	

Memoria	
NUM0	1000H 80H
NUM1	1001H <del>C8H</del> C9H
NUM2	1002H FFH
BITS0	1003H <del>7FH</del> 80H
BITS1	1004H AAH
	1005H
	...
	2000H

; Copia valor de NUM0 en AL  
; Suma el valor de AL al de AL  
; Incrementa en 1 NUM1  
; Copia el valor de NUM1 en BH  
; Copia BH en BL  
; Decrementa en 1 el valor de BL  
; Hace BL - BH y lo almacena en BL  
; Copia valor de BITS1 en CH  
; Hace CH AND BITS0 y lo almacena en CH  
; Niega BITS0 utilizando Cal y lo almacena en BITS0  
; Hace CH OR BITS0 y lo almacena en CH



# Assembler



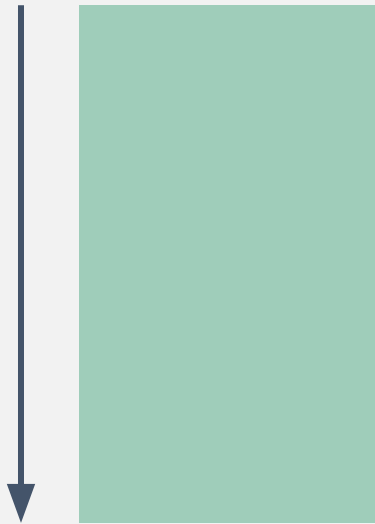
*Salto*

# Assembler

## *Salto*

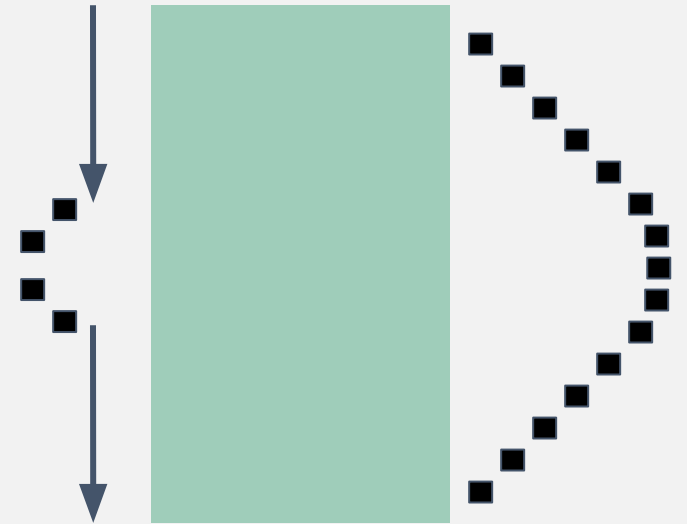
Los saltos permiten ir a un determinado punto del código indicado por una etiqueta

### Flujo normal de ejecución



### Flujo de ejecución con saltos

instruccion 1  
**etiqueta:** instruccion 2  
instruccion 3  
instruccion 4  
instruccion 5  
instruccion 6  
instruccion 7  
**JMP etiqueta**  
instruccion 9  
instruccion 10





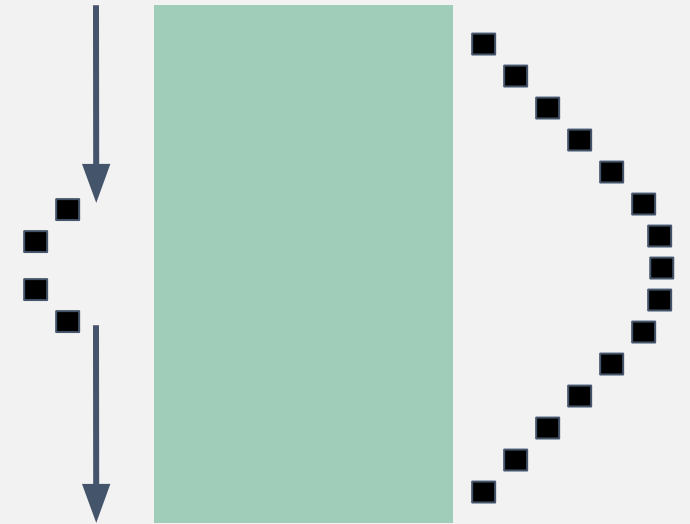
# Assembler

## *Salto*

Las sentencias de saltos son las siguientes. Las condiciones se basan sobre los flags

- **JMP** etiq      ☐ Salto incondicional
- **JS** etiq        ☐ Salto si S == 1
- **JNS** etiq      ☐ Salto si ~ S (S == 0)
- **JZ** etiq        ☐ Salto si Z == 1
- **JNZ** etiq      ☐ Salto si ~ Z (Z == 0)
- **JC** etiq        ☐ Salto si C == 1
- **JNC** etiq      ☐ Salto si ~ C (C == 0)
- **JO** etiq        ☐ Salto si O == 1
- **JNO** etiq      ☐ Salto si ~ O (O == 0)

Flujo de ejecución con saltos



# Assembler

*Ejemplo saltos*

```
ORG 1000H
    INI  DB 0
    FIN  DB 15
ORG 2000H
    MOV AL, INI
    MOV AH, FIN
SUMA: INC AL
      CMP AL, AH
      JNZ SUMA
HLT
END
```

	AX	BX	CX	DX
L	<del>0</del> <del>1</del> <del>2</del> <del>3</del> ... 15			
H	15			

; Copia valor de INI en AL  
; Copia valor de FIN en AH  
; Incremento en 1 el valor de AL  
; Comparo AL con AH  
; Si la comparación termina con el flag Z = 0 salta a *SUMA*

INI

FIN

Memoria

1000H	0
1001H	15
1002H	
1003H	
...	
2000H	

Flag Z: ~~0~~ ~~0~~ ~~0~~ ... 1

$$\begin{array}{r} 0000\ 0001 \\ -\ 0000\ 1111 \\ \hline 1111\ 0010 \end{array}$$

...

$$\begin{array}{r} 0000\ 1111 \\ -\ 0000\ 1111 \\ \hline 0000\ 0000 \end{array}$$