



Universidad Nacional  
de Entre Ríos

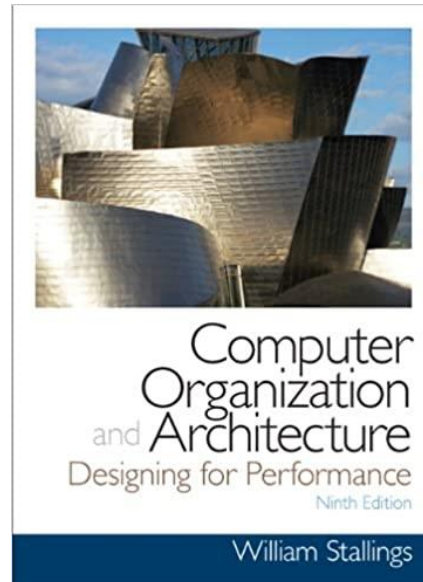
Tecnicatura universitaria en desarrollo web

# Repertorio de instrucciones

Semana 5 – Arquitectura de computadoras

# Esta presentación esta basada en el libro de:

- ❑ William Stallings, Computer Organization and Architecture, 9th Edition, 2017



Archivos presentación y ejemplos se alojan en:

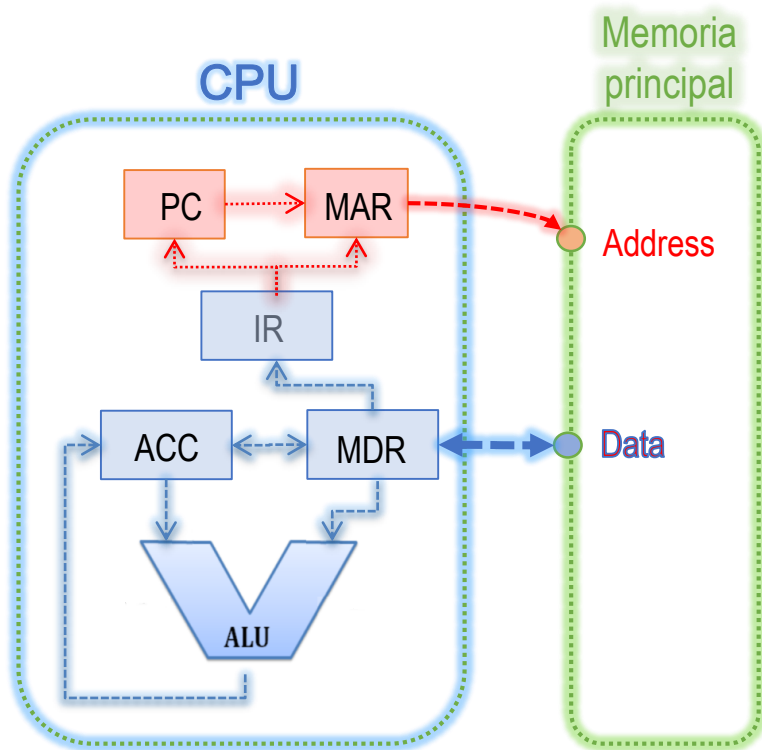


<https://github.com/ruiz-jose/tudw-arq.git>

## Repertorio de instrucciones

- Tipos de instrucciones
- Modos de direccionamiento
- Operandos de la instrucción
- Banderas

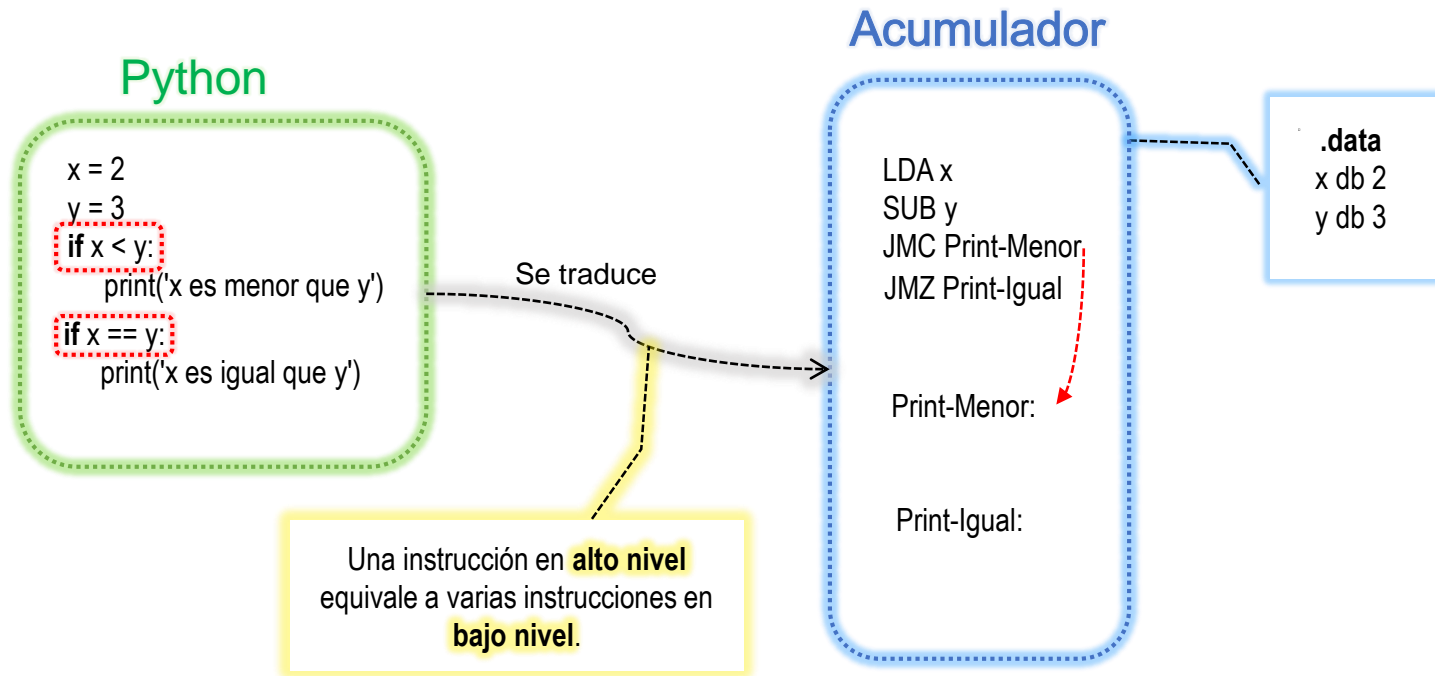
## Arquitectura acumulador (ACC)



### Repertorio de instrucciones

Código operación (C <sub>7</sub> C <sub>6</sub> )	Nemónico	Efecto
0 = 00	LDA dirección	ACC ← Memoria[dirección]
1 = 01	STA dirección	Memoria[dirección] ← ACC
2 = 10	ADD dirección	ACC ← ACC + Memoria[dirección]
3 = 11	HLT	Detiene CPU

## Equivalencia de las estructura de control «if» entre python y ensamblador





Captación

```
1: MAR ← PC
2: MDR ← read(Memoria[MAR]) ;
   PC ← PC + 1
3: IR ← MDR
```

Transferencias:

LDA

```
1: MAR ← IR(dirección)
2: MDR ← read(Memoria[MAR])
3: ACC ← MDR
```

STA

```
1: MAR ← IR(dirección)
2: MDR ← ACC
3: write(Memoria[MAR]) ← MDR
```

Decodificar

Operaciones ALU:

ADD y SUB

```
1: MAR ← IR(dirección)
2: MDR ← read(Memoria[MAR])
3: ACC ← ACC opALU MDR
```

Control:

Saltos condicionales:

JMZ

1: Si ACC = 0

Si

JMC

1: Si  $255 > ACC > 0$ 

Si

Salto incondicional:

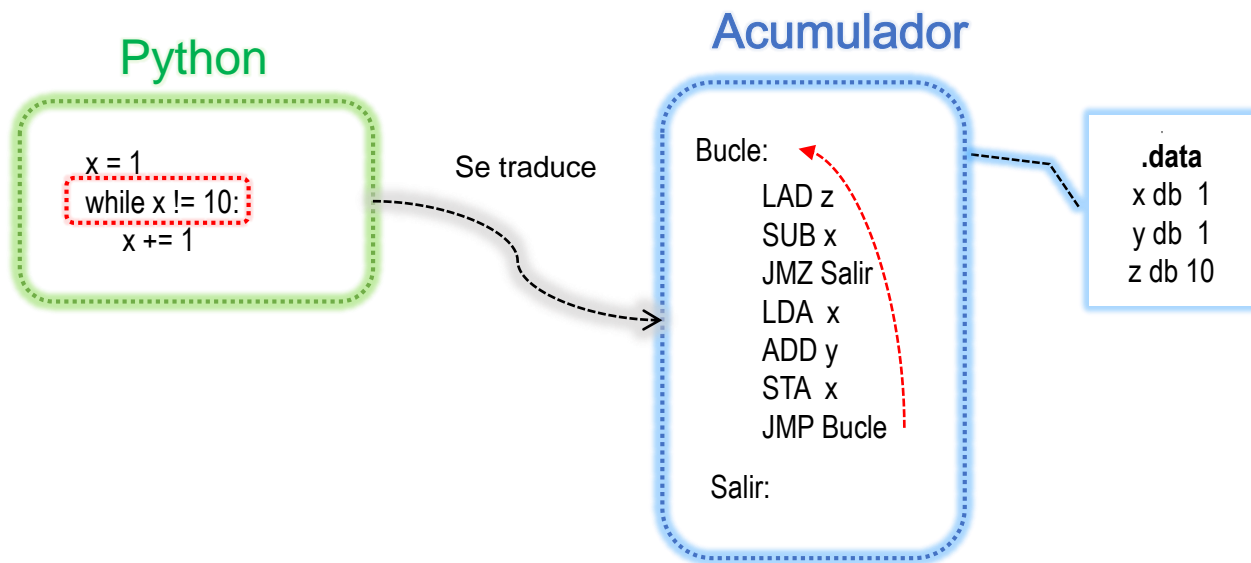
JMP

1: PC ← Dirección[IR]

HLT

Detiene CPU

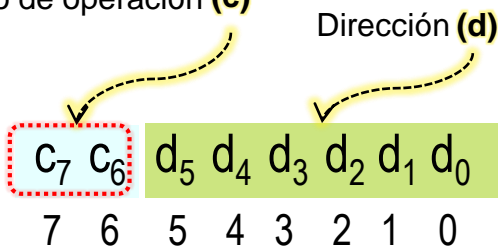
## Equivalencia de las estructura de control «while» entre python y ensamblador





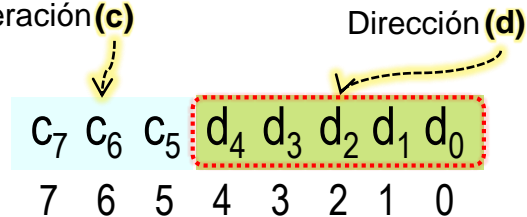
## □ Tipo de instrucciones

Hay que ampliar el  
código de operación (c)



Pero la cantidad de posiciones  
direccionales se reduce  $2^5 = 32$

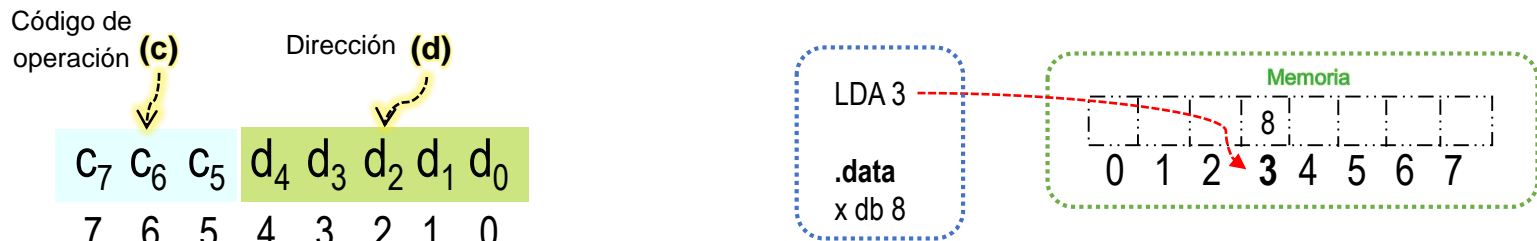
Código de  
operación (c)



Tipo	Código operación / Nemónico	Efecto
Movimiento	0 = 000 → LDA dirección	ACC ← Memoria[dirección]
	1 = 001 → STA dirección	Memoria[dirección] ← ACC
Aritmética	2 = 010 → ADD dirección	ACC ← ACC + Memoria[dirección]
	3 = 011 → SUB dirección	ACC ← ACC – Memoria[dirección]
Control	4 = 100 → JMP dirección	PC ← dirección
	5 = 101 → JMZ dirección	Si ACC == 0 entonces PC ← dirección
	6 = 110 → JMC dirección	Si 255 > ACC < 0 entonces PC ← dirección
	7 = 111 → HLT	Detiene CPU

❑ **Modos de direccionamiento:** Indican la manera que se obtienen los operandos de una instrucción.

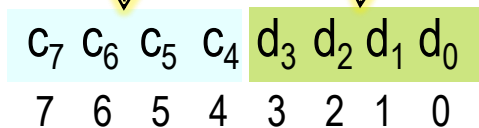
❖ **Direccionamiento directo:** dentro de la instrucción se encuentra la dirección del operando.



❖ **Direccionamiento inmediato:** el valor del operando forma parte de la instrucción.

Vamos a agregar más instrucciones

código de operación (c)



Pero la cantidad de posiciones direccionales se reduce  $2^4 = 16$

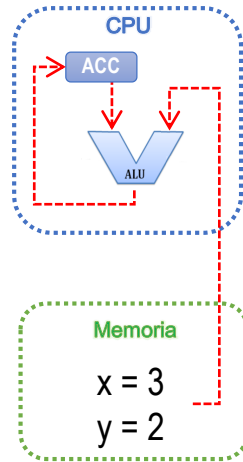
Dirección (d)

Tipo	Código operación / Nemónico		Efecto
Movimiento	0 = 0000 → LDA	dirección	ACC ← Memoria[dirección]
	1 = 0001 → LDI	valor	ACC ← valor
Aritmética	2 = 0010 → ADD	dirección	ACC ← ACC + Memoria[dirección]
	3 = 0011 → ADI	valor	ACC ← ACC + valor

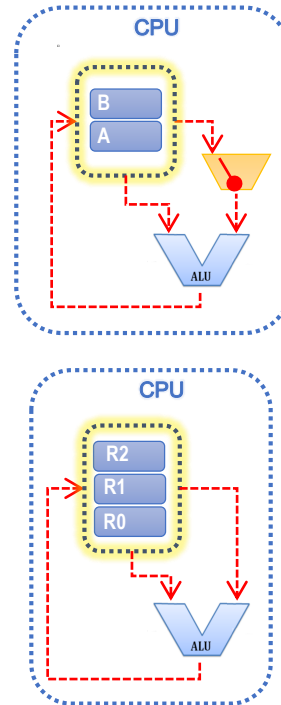
## ■ Modos de direccionamiento

## ❖ Por registro

## Acumulador



## Registros



ADD R, R

ADD A, B

 $A \leftarrow A + B$ 

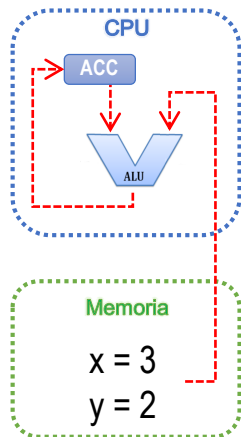
ADD R, R, R

ADD R3, R2, R1

 $R3 \leftarrow R2 + R1$

## Acumulador

Las operaciones tienen como operando implícito el registro (ACC) y para el otro operando se proporciona su dirección en la memoria y el resultado se coloca en el ACC.

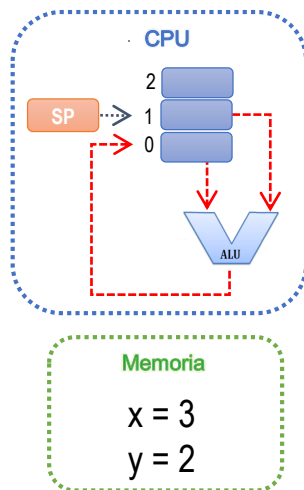


## Stack

Cuenta con una pila y las operaciones se realizan sobre los elementos almacenados en ella accesibles desde el tope (SP).

Un stack o pila posee dos operaciones:

- **Push** que permite colocar un dato en la primera posición libre.
- **Pop** que permite retirar el último dato que se encuentra en la pila.

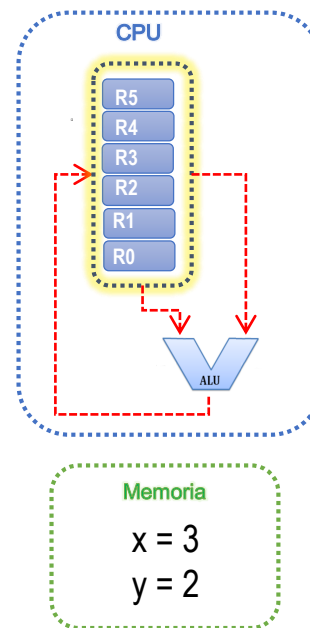


## Registros (R)

Se cuenta con un banco de registros y las operaciones se realizan entre ellos.

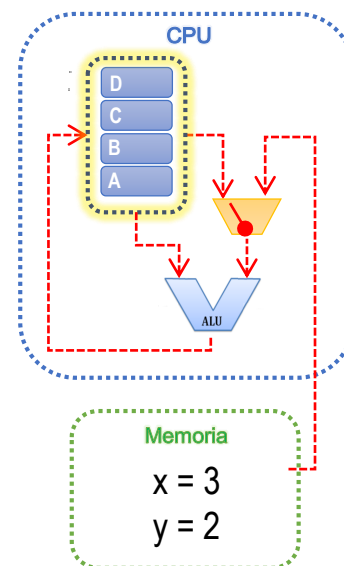
ARM

Registro  
(load-store)

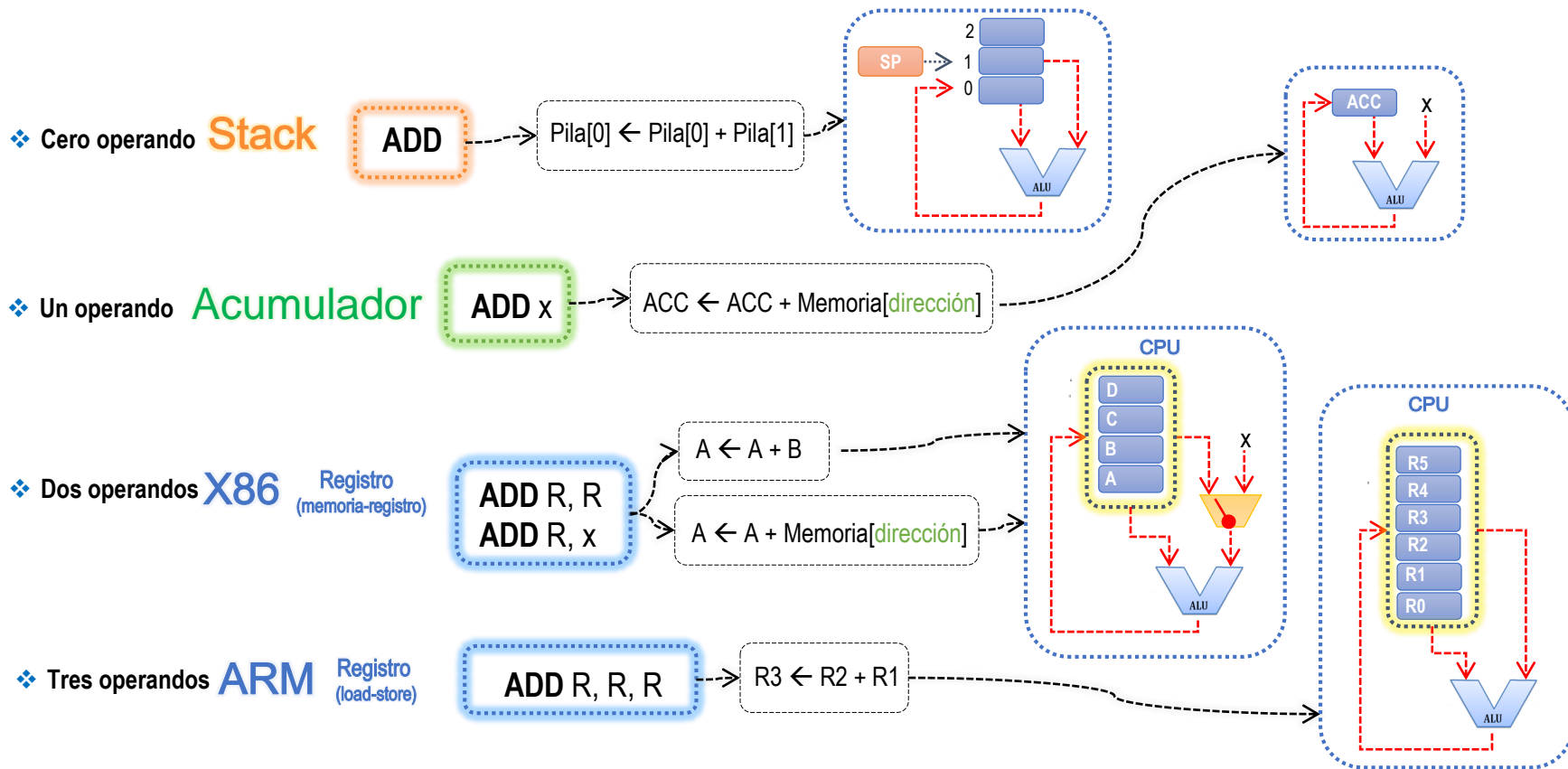


X86

Registro  
(memoria-registro)



❑ **Operandos de la instrucción** Se clasifican según el numero de operandos **explícitos** en la instrucción:



**Z = X+Y**

**.data**

x db 3

y db 2

z db 0

**Acumulador**

LDA x  
STA x  
ADD x

LDA x  
ADD y  
STA z

**Stack**

PUSH x  
POP x  
ADD

PUSH x  
PUSH y  
ADD  
POP z

PUSH x => Pila[0] = Memoria[x]  
PUSH y => Pila[1] = Memoria[y]  
ADD => Pila[0] = Pila[0] + Pila[1]  
POP z => Memoria[z] = Pila[0]

**Registros (R)**

**X86**

Registro  
(memoria-registro)

MOV R, x  
MOV x, R  
ADD R, x

MOV A, x  
ADD A, y  
MOV w, A

**ARM**

Registro  
(load-store)

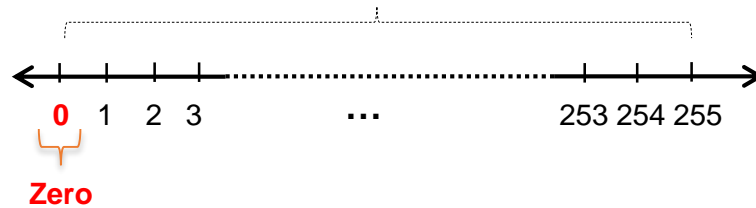
LD R, x  
ST R, x  
ADD R, R, R

LD R1, x  
LD R2, y  
ADD R3, R2, R1  
ST R3, z

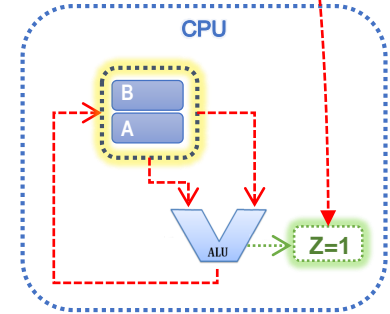
# Flags - Banderas

Cero - (resultado=0) – Zero – Flag Z

1 byte = 8 bits, se puede representar  $2^8 = 256$  valores (0-255)

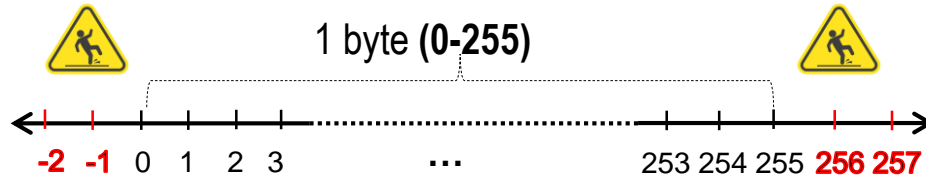


```
MOV A, 2  
MOV B, 2  
SUBA, B
```



Si el resultado de la ALU es cero → el flag Zero = 1

## Acarreo – (o me llevo) – Carry – Flag C



**Se activa el Carry = 1:** si el resultado es mayor o menor al rango soportado por el registro de 8 bits (0-255) → indica que el resultado es erróneo.



$$W = Z - (X + Y)$$

.data

x db 3

y db 2

z db 7

w db 0

### Acumulador

LDA x  
STA x  
ADD x  
SUB x

LDA x  
ADD y  
STA t  
LDA z  
SUB t  
STA w

### Stack

PUSH x  
POP x  
ADD  
SUB

PUSH w  
PUSH x  
PUSH y  
ADD  
SUB  
POP w

### ARM Registro (load-store)

LD R, x  
ST x, R  
ADD R, R, R  
SUB R, R, R

LD R1, x  
LD R2, y  
ADD R3, R2, R1  
LD R1, z  
SUB R2, R1, R3  
ST R2, w

### X86 Registro (memoria-registro)

MOV R, x  
MOV x, R  
ADD R, x  
SUB R, x  
**MOV R, R**

MOV A, x  
ADD A, y  
MOV B, A  
MOV A, z  
SUB A, B  
MOV w, A

# Preguntas?