

# CLASE 3

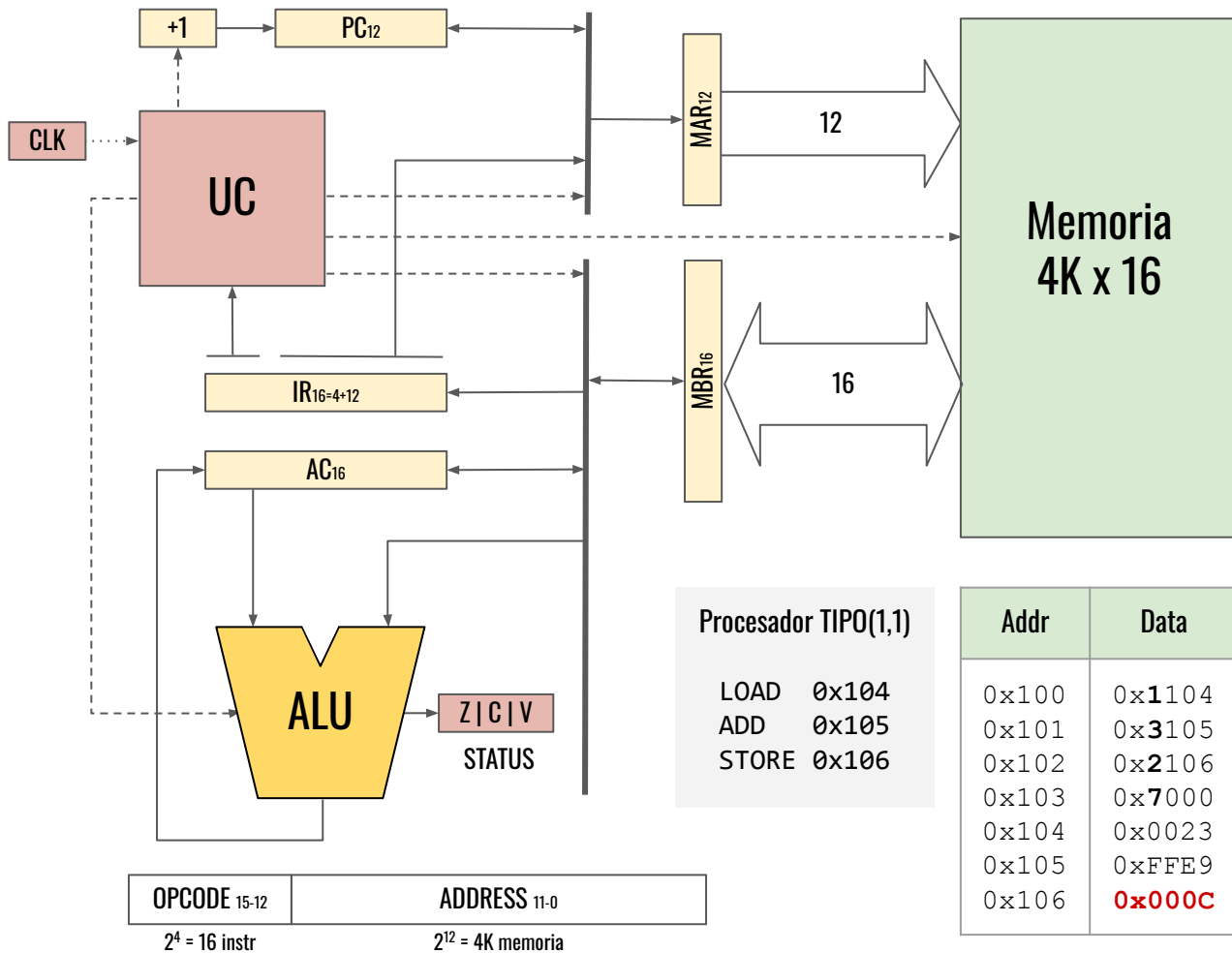
MEDICIÓN DE PERFORMANCE  
EL LENGUAJE ENSAMBLADOR  
PROGRAMACIÓN ESTRUCTURADA

## **Bibliografía**

**Linda Null - Essentials of Computer Organization and Architecture (1a ed. 2003)**

**Capítulo 4: MARIE: An Introduction to a Simple Computer**

**Capítulo 5: A Closer Look at Instruction Set Architectures**



## El ciclo de instrucción

F  
E  
T  
C  
H

MAR  $\leftarrow$  PC  
MBR  $\leftarrow$  M[MAR]  
IR  $\leftarrow$  MBR

RTL

D  
E  
C  
O  
D  
E

PC  $\leftarrow$  PC + 1  
Decode IR[15-12]  
MAR  $\leftarrow$  IR[11-0]

Los 2 pueden realizarse simultáneamente

E  
X  
E  
C  
U  
T  
E

Si IR[15-12]==0001 (Load)  
MBR  $\leftarrow$  M[MAR]  
AC  $\leftarrow$  MBR

Si IR[15-12]==0010 (Store)  
MBR  $\leftarrow$  AC  
M[MAR]  $\leftarrow$  MBR

Si IR[15-12]==0011 (Add)  
MBR  $\leftarrow$  M[MAR]  
AC  $\leftarrow$  AC + MBR

En principio, todas las instrucciones toman 7 ciclos de reloj (pasos de la UC)

# Medición de performance

$$\frac{\textit{tiempo}}{\textit{programa}} = \frac{\textit{instrucciones}}{\textit{programa}} \times \frac{\textit{ciclos}}{\textit{instruccion}} \times \frac{\textit{segundos}}{\textit{ciclo}}$$

# Medición de performance

$$\frac{\text{tiempo}}{\text{programa}} = \frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instruccion}} \times \frac{\text{segundos}}{\text{ciclo}}$$

$$t = \frac{N \times CPI}{f_{\text{clock}}}$$

# Medición de performance

$$\frac{\text{tiempo}}{\text{programa}} = \frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instruccion}} \times \frac{\text{segundos}}{\text{ciclo}}$$

$$t = \frac{N \times CPI}{f_{clock}}$$

Nuestra máquina utiliza (por ahora) **7** ciclos de reloj por instrucción para todas las instrucciones.

Nuestro primer programa tiene **3** instrucciones (suma de dos números en memoria).

Si la CPU funcionara con un reloj de **100 MHz**, resulta que  $t = 3 \times 7 / 100M = 210 \text{ ns}$

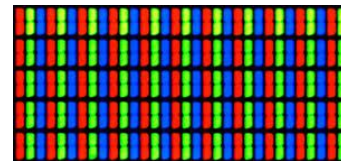
O sea que el programa se podría ejecutar **4.8** millones de veces por segundo.

# Medición de performance

$$\frac{\text{tiempo}}{\text{programa}} = \frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instruccion}} \times \frac{\text{segundos}}{\text{ciclo}}$$

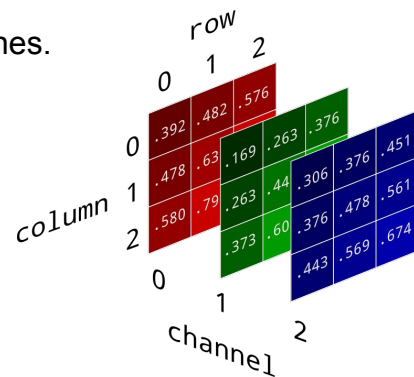
$$t = \frac{N \times CPI}{f_{clock}}$$

LCD



Nuestra máquina utiliza (por ahora) **7** ciclos de reloj por instrucción para todas las instrucciones. Nuestro primer programa tiene **3** instrucciones (suma de dos números en memoria). Si la CPU funcionara con un reloj de **100 MHz**, resulta que  $t = 3 \times 7 / 100M = 210 \text{ ns}$ . O sea que el programa se podría ejecutar **4.8** millones de veces por segundo.

**EJEMPLO:** video HD: 720p = 1280 x 720 pixeles (~1M). Cada pixel 3 bytes (RGB)  
1280\*720\*3 = **2.7** millones de operaciones de suma (load-add-store): brillo  
Podría procesar menos de 2 fps



# Medición de performance

$$\frac{\text{tiempo}}{\text{programa}} = \frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos}}{\text{instruccion}} \times \frac{\text{segundos}}{\text{ciclo}}$$

ISA: Repertorio de instrucciones y tipos de datos

Organización interna

Tecnología de implementación

## EJEMPLO: mejora de la capacidad de procesamiento de video HD:

- Optimizar formato de los datos (16 → 24, suma triple)... al ser de 16 bits no se adecuaba bien al problema.
- Organizar mejor: que tome menos de 6/7 ciclos por instrucción. Arquitectura Harvard y técnicas de segmentación ( próx ).
- Aumentar velocidad de reloj, utilizando tecnología más moderna.

Podría resolverse con una única instrucción que tome sólo un ciclo de reloj? Podría procesar 100 fps  
Funcionando a 1 GHz podría procesar 1000 fps. O podría realizar 10 operaciones a 100 fps.  
Cómo se podría expresar la mejora?



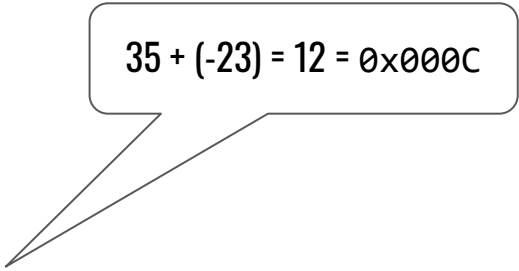
# Medición de performance

$$t = \frac{N \times CPI}{f_{clock}}$$

Este tema volverá a ser abordado cuando se introduzcan nuevos repertorios de instrucciones y sus implementaciones.  
Es una de nuestras herramientas de comparación de arquitecturas.  
Por ahora pausa en este tema.

# Primer programa

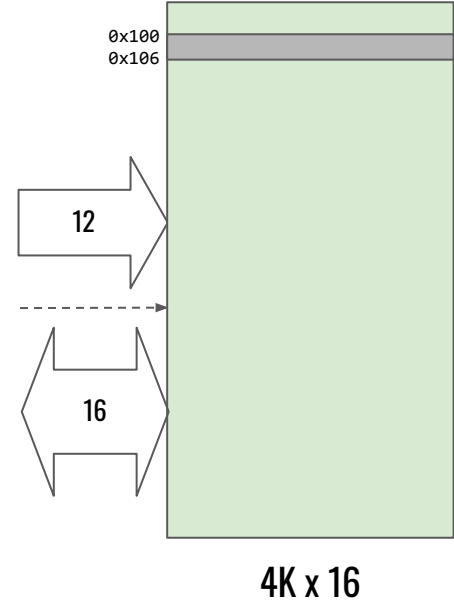
| Addr  | Data            |
|-------|-----------------|
| 0x100 | 0x <b>1</b> 104 |
| 0x101 | 0x <b>3</b> 105 |
| 0x102 | 0x <b>2</b> 106 |
| 0x103 | 0x <b>7</b> 000 |
| 0x104 | 0x0023          |
| 0x105 | 0xFFE9          |
| 0x106 | 0x0000          |



$35 + (-23) = 12 = 0x000C$

# Primer programa

| Dirección de memoria |       | Contenido de la memoria  |                 |
|----------------------|-------|--------------------------|-----------------|
| (12-bit BINARIO)     | (HEX) | (16-bit BINARIO)         | (HEX)           |
| 000100000000         | 0x100 | <b>0001</b> 000100000100 | 0x <b>1</b> 104 |
| 000100000001         | 0x101 | <b>0011</b> 000100000101 | 0x <b>3</b> 105 |
| 000100000010         | 0x102 | <b>0010</b> 000100000110 | 0x <b>2</b> 106 |
| 000100000011         | 0x103 | <b>0111</b> 000000000000 | 0x <b>7</b> 000 |
| 000100000100         | 0x104 | 0000000000 <b>100011</b> | 0x00 <b>23</b>  |
| 000100000101         | 0x105 | <b>1111111111</b> 101001 | 0x <b>FF</b> E9 |
| 000100000110         | 0x106 | 0000000000000000         | 0x0000          |



Necesitamos herramientas para:

- Crear el contenido de la memoria
- Verificar su funcionamiento
- Grabar la memoria.

# Lenguaje ensamblador

| Dirección memoria | Código máquina | Assembler (mnemonics) | Assembler (labels/etiquetas) | Assembler (directivas) | Assembler (comentarios)        |
|-------------------|----------------|-----------------------|------------------------------|------------------------|--------------------------------|
| 0x100             | 0x1104         | LOAD 0x104            | LOAD X                       | ORG 0x100<br>LOAD X    | ORG 0x100<br>LOAD X / Programa |
| 0x101             | 0x3105         | ADD 0x105             | ADD Y                        | ADD Y                  | ADD Y                          |
| 0x102             | 0x2106         | STORE 0x106           | STORE Z                      | STORE Z                | STORE Z                        |
| 0x103             | 0x7000         | HALT                  | HALT                         | HALT                   | HALT                           |
| 0x104             | 0x0023         | 0x0023                | X, 0x0023                    | X, DEC 35              | X, DEC 35 / Datos              |
| 0x105             | 0xFFE9         | 0xFFE9                | Y, 0xFFE9                    | Y, DEC -23             | Y, DEC -23                     |
| 0x106             | 0x0000         | 0x0000                | Z, 0x0000                    | Z, HEX 0000            | Z, HEX 0000                    |

La “fuente” es generalmente un archivo de texto (codificación ASCII).

El “ensamblador” es una app para un determinado SO que sirve de plataforma de desarrollo.. El resultado se almacena generalmente en un archivo binario, que luego debe ser grabado en la memoria del procesador de destino.

El ensamblador hace dos pasadas: en la primera resuelve direcciones y labels, en la segunda genera el código máquina.

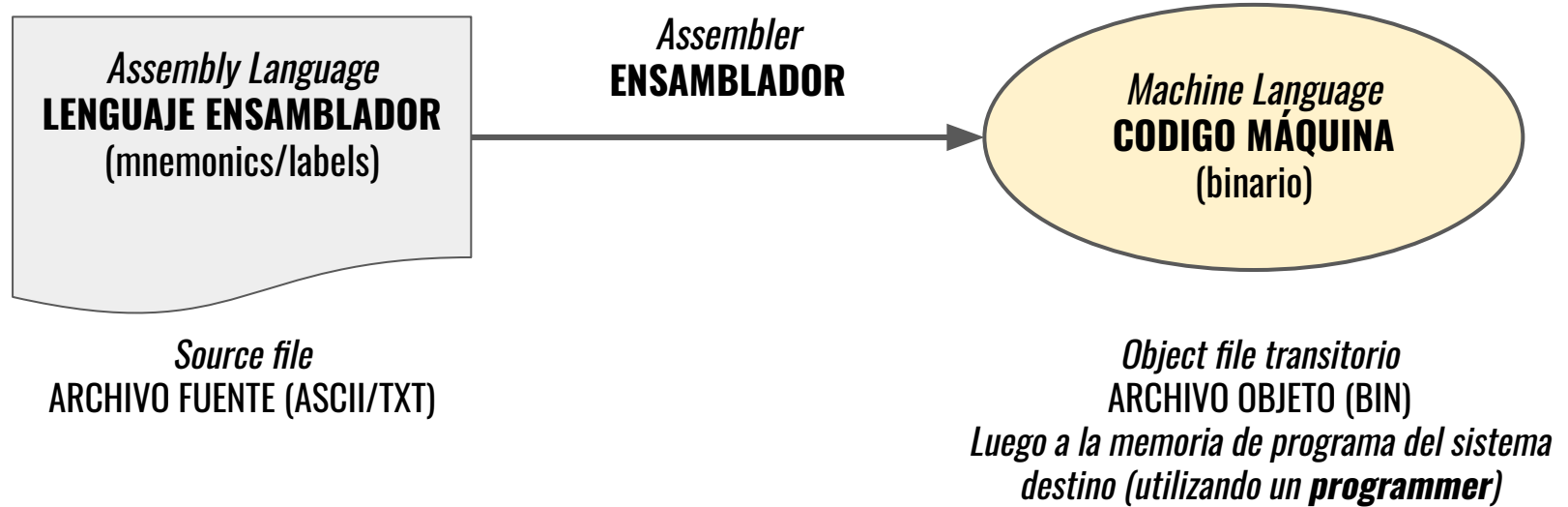
## + MACROS

secuencias de instrucciones con nombre

## + PSEUDO-INSTRUCCIONES

ADD X, Y, Z / Tipo(3,3)

# Ensamblador y ensamblador ¿?





## Comparando con un lenguaje de alto nivel

VENTAJAS: arquitectura, optimización velocidad o memoria (embedded, drivers), 10%-90%, embedded systems

DESVENTAJAS: difícil de leer y mantener, no portable (cero)

# Simulador MARIE

 Home File Examples Edit View Help G

 Assembly code: File copied as link

```
1 / Suma de dos números en memoria
2
3 ORG 100
4
5 Load X
6 Add Y
7 Store Z
8 Halt
9
10 X, Dec 35
11 Y, Dec -23
12 Z, Dec 0
13
14
```

AC0023

IR1104

MAR104

MBR0023

PC101

IN0000

OUT0000


Output log RTL log Watch list

Input list

OUTPUT MODE:

Performed one step

|     | +0   | +1   | +2   | +3   | +4   | +5   | +6   | +7   | +8   | +9   | +A   | +B   | +C   | +D   | +E   | +F   |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 0E0 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 0F0 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 100 | 1104 | 3105 | 2106 | 7000 | 0023 | FFE9 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 110 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 120 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |

 Assemble Step Microstep Step Back Run Restart Delay:  1 ms

Simular

[MARIE link](#)

**Intervalo 15'**

# Teorema del Programa Estructurado (Böhm–Jacopini)

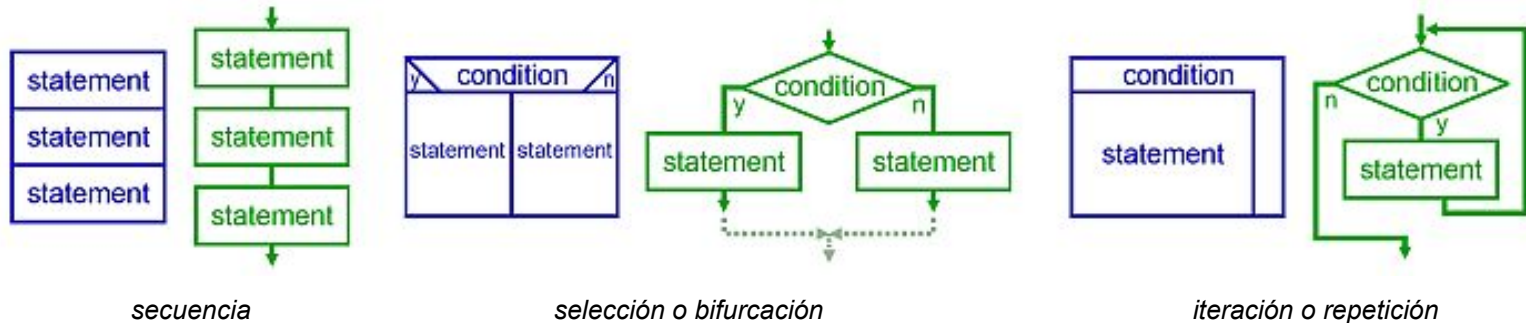
Toda función **computable** puede ser implementada en un lenguaje de programación que combine sólo tres estructuras de control:

**Secuencia:** ejecución de una instrucción (o subprograma, o conjunto de instrucciones) tras otra.

**Selección:** ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.

**Iteración:** repetición de una instrucción (o conjunto) mientras una variable booleana sea 'verdadera'. Esta estructura lógica también se conoce como ciclo o bucle.

Este teorema demuestra que el **salto incondicional** no es estrictamente necesario y que para todo programa que lo utilice existe otro equivalente que no hace uso de dicha instrucción



## Paradigma de Programación Estructurada (lenguaje C)

Utilización de las tres estructuras básicas y **Subrutinas**, prescindiendo del salto incondicional (goto).



Guía de problemas resueltos  
Apunte: Resumen de la arquitectura MARIE  
Enunciados de la Práctica 1 (entrega obligatoria)

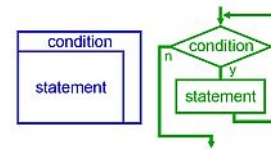


# Otras instrucciones de MARIE

- **Skipcond** para implementar selecciones e iteraciones.
- **JnS** y **JumpI** para implementar llamados a subrutinas.
- **AddI** para implementar punteros.

| Opcode | Instruction | RTN   |
|--------|-------------|---|
| 0000   | JnS X       | $MBR \leftarrow PC$<br>$MAR \leftarrow X$<br>$M[MAR] \leftarrow MBR$<br>$MBR \leftarrow X$<br>$AC \leftarrow 1$<br>$AC \leftarrow AC + MBR$<br>$PC \leftarrow AC$   |
| 0001   | Load X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR], AC \leftarrow MBR$  |
| 0010   | Store X     | $MAR \leftarrow X, MBR \leftarrow AC$<br>$M[MAR] \leftarrow MBR$  |
| 0011   | Add X       | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$   |
| 0100   | Subt X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC - MBR$   |
| 0101   | Input       | $AC \leftarrow InREG$   |
| 0110   | Output      | $OutREG \leftarrow AC$  |
| 0111   | Halt        |   |
| 1000   | Skipcond    | If $IR[11-10] = 00$ then<br>If $AC < 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 01$ then<br>If $AC = 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 10$ then<br>If $AC > 0$ then $PC \leftarrow PC + 1$ |
| 1001   | Jump X      | $PC \leftarrow IR[11-0]$  |
| 1010   | Clear       | $AC \leftarrow 0$   |
| 1011   | AddI X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$MAR \leftarrow MBR$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$  |
| 1100   | JumpI X     | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$PC \leftarrow MBR$  |

# Iteración (for/while)



## Skipcond

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | x | x | x | x | x | x | x | x | x | x |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

0x8400

Tres opciones implementadas en MARIE:

```

1000 00 skip if AC<0 0x8000
1000 01 skip if AC=0 0x8400
1000 10 skip if AC>0 0x8800
  
```

/ Ejemplo de iteración

```

Loop, ... / bloque que repite
...
...
Load Cont
Subt Step
Store Cont
Skipcond 400
Jump Loop
Halt
  
```

Cont--

Pseudo  
JNZ

Se utiliza la posición de memoria **Cont** para llevar la cuenta de las repeticiones. Cuando llega a cero para de repetir y termina.

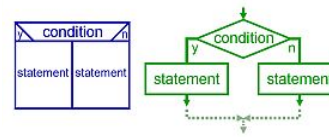
Simular y verificar qué sucede si se utiliza:

Skipcond 000

[MARIE link](#)

| Opcode | Instruction | RTN   |
|--------|-------------|---|
| 0000   | JnS X       | $PC \leftarrow PC$<br>$MAR \leftarrow X$<br>$M[MAR] \leftarrow MBR$<br>$MBR \leftarrow X$<br>$AC \leftarrow 1$<br>$AC \leftarrow AC + MBR$<br>$PC \leftarrow AC$  |
| 0001   | Load X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR], AC \leftarrow MBR$  |
| 0010   | Store X     | $MAR \leftarrow X, MBR \leftarrow AC$<br>$M[MAR] \leftarrow MBR$  |
| 0011   | Add X       | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$   |
| 0100   | Subt X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC - MBR$   |
| 0101   | Input       | $AC \leftarrow InREG$   |
| 0110   | Output      | $OutREG \leftarrow AC$  |
| 0111   | Halt        |   |
| 1000   | Skipcond    | If IR[11-10] = 00 then<br>If AC < 0 then $PC \leftarrow PC + 1$<br>Else If IR[11-10] = 01 then<br>If AC = 0 then $PC \leftarrow PC + 1$<br>Else If IR[11-10] = 10 then<br>If AC > 0 then $PC \leftarrow PC + 1$ |
| 1001   | Jump X      | $PC \leftarrow IR[11-0]$  |
| 1010   | Clear       | AC 0  |
| 1011   | AddI X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$MAR \leftarrow MBR$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$  |
| 1100   | JumpI X     | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$PC \leftarrow MBR$  |

# Selección (if/else/switch)



## Skipcond

|         |     |                     |
|---------|-----|---------------------|
| 1 0 0 0 | 1 0 | x x x x x x x x x x |
|---------|-----|---------------------|

0x8400

Tres instrucciones?

|      |    |              |     |
|------|----|--------------|-----|
| 1000 | 00 | skip if AC<0 | SKN |
| 1000 | 01 | skip if AC=0 | SKZ |
| 1000 | 10 | skip if AC>0 | SKP |

```

if x == y then
    x = x * 2;
else
    y = y - x;
endif
    
```

/ Example 4.2

ORG 100

If, Load X

/Load the first value

Subt Y

/Subtract the value of Y, store result in AC

**Skipcond 400**

/If AC=0, skip the next instruction

**Jump Else**

/Jump to Else part if AC is not equal to 0

Then, Load X

/Reload X so it can be doubled

Add X

/Double X

Store X

/Store the new value

**Jump Endif**

/Skip over the false part (else)

Else, Load Y

/Start the else part by loading Y

Subt X

/Subtract X from Y

Store Y

/Store Y-X in Y

Endif, Halt

/Terminate program

X, Dec 12

Y, Dec 20

[MARIE Link](#)

Simular

En este ejemplo, la condición necesaria es la igualdad. La resta hace la traducción al estado del AC. Agregando **Skipcond** y **Jump** se implementa el equivalente a **JNE** (saltar si no iguales) en tres instrucciones.

| Opcode | Instruction | RTN   |
|--------|-------------|---|
| 0000   | JnS X       | $MBR \leftarrow PC$<br>$MAR \leftarrow X$<br>$M[MAR] \leftarrow MBR$<br>$MBR \leftarrow X$<br>$AC \leftarrow 1$<br>$AC \leftarrow AC + MBR$<br>$PC \leftarrow AC$   |
| 0001   | Load X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR], AC \leftarrow MBR$  |
| 0010   | Store X     | $MAR \leftarrow X, MBR \leftarrow AC$<br>$M[MAR] \leftarrow MBR$  |
| 0011   | Add X       | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$   |
| 0100   | Subt X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC - MBR$   |
| 0101   | Input       | $AC \leftarrow InREG$   |
| 0110   | Output      | $OutREG \leftarrow AC$  |
| 0111   | Halt        |   |
| 1000   | Skipcond    | If IR[11-10] = 00 then<br>If AC < 0 then PC $\leftarrow$ PC + 1<br>Else If IR[11-10] = 01 then<br>If AC = 0 then PC $\leftarrow$ PC + 1<br>Else If IR[11-10] = 10 then<br>If AC > 0 then PC $\leftarrow$ PC + 1 |
| 1001   | Jump X      | $PC \leftarrow IR[11-0]$  |
| 1010   | Clear       | AC 0  |
| 1011   | AddI X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$MAR \leftarrow MBR$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$  |
| 1100   | JumpI X     | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$PC \leftarrow MBR$  |

# Indirección (puntero)

puntero

/ Example 4.1

```

ORG 100
Load Addr
Store Next
Load Num
Subt One
Store Cont
Load Sum
AddI Next
Store Sum
Load Next
Add One
Store Next
Load Cont
Subt One
Store Cont
Skipcond 000
Jump Loop
Halt
Addr, Hex 117
Next, Hex 0
Num, Dec 5
Sum, Dec 0
Cont, Hex 0
One, Dec 1
Dec 10
Dec 15
Dec 20
Dec 25
Dec 30
    
```

Loop,

```

/Load address of first number to be added
/Store this address is our Next pointer
/Load the number of items to be added
/Decrement
/Store this value in Ctr to control looping
/Load the Sum into AC
/ Add the value pointed to by location Next
/Store this sum
/Load Next
/Increment by one to point to next address
/Store in our pointer Next
/Load the loop control variable
/Subtract one from the loop control variable
/Store this new value in loop control variable
/If control variable < 0, skip next instruction
/Otherwise, go to Loop
/Terminate program
/Numbers to be summed start at location 0x118
/A pointer to the next number to add
/The number of values to add
/The sum
/ The loop control variable
/Used to increment and decrement by 1
/The values to be added together
    
```

M[X] y  
M[[X]]

Ver utilización del  
puntero **Next**, posición  
de memoria que almacena  
una dirección -> **AddI**

| Opcode | Instruction | RTN   |
|--------|-------------|---|
| 0000   | JnS X       | $MBR \leftarrow PC$<br>$MAR \leftarrow X$<br>$M[MAR] \leftarrow MBR$<br>$MBR \leftarrow X$<br>$AC \leftarrow 1$<br>$AC \leftarrow AC + MBR$<br>$PC \leftarrow AC$   |
| 0001   | Load X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR], AC \leftarrow MBR$  |
| 0010   | Store X     | $MAR \leftarrow X, MBR \leftarrow AC$<br>$M[MAR] \leftarrow MBR$  |
| 0011   | Add X       | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$   |
| 0100   | Subt X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC - MBR$   |
| 0101   | Input       | $AC \leftarrow InREG$   |
| 0110   | Output      | $OutREG \leftarrow AC$  |
| 0111   | Halt        |   |
| 1000   | Skipcond    | If $IR[11-10] = 00$ then<br>If $AC < 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 01$ then<br>If $AC = 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 10$ then<br>If $AC > 0$ then $PC \leftarrow PC + 1$ |
| 1001   | Jump X      | $PC \leftarrow IR[11-0]$  |
| 1010   | Clear       | $AC \leftarrow 0$   |
| 1011   | AddI X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$MAR \leftarrow MBR$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$  |
| 1100   | JumpI X     | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$PC \leftarrow MBR$  |

# Subrutinas

| Opcode | Instruction | RTN   |
|--------|-------------|---|
| 0000   | JnS X       | $MBR \leftarrow PC$<br>$MAR \leftarrow X$<br>$M[MAR] \leftarrow MBR$<br>$MBR \leftarrow X$<br>$AC \leftarrow 1$<br>$AC \leftarrow AC + MBR$<br>$PC \leftarrow AC$   |
| 0001   | Load X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR], AC \leftarrow MBR$  |
| 0010   | Store X     | $MAR \leftarrow X, MBR \leftarrow AC$<br>$M[MAR] \leftarrow MBR$  |
| 0011   | Add X       | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$   |
| 0100   | Subt X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC - MBR$   |
| 0101   | Input       | $AC \leftarrow InREG$   |
| 0110   | Output      | $OutREG \leftarrow AC$  |
| 0111   | Halt        |   |
| 1000   | Skipcond    | If $IR[11-10] = 00$ then<br>If $AC < 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 01$ then<br>If $AC = 0$ then $PC \leftarrow PC + 1$<br>Else If $IR[11-10] = 10$ then<br>If $AC > 0$ then $PC \leftarrow PC + 1$ |
| 1001   | Jump X      | $PC \leftarrow IR[11-0]$  |
| 1010   | Clear       | $AC \leftarrow 0$   |
| 1011   | AddI X      | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$MAR \leftarrow MBR$<br>$MBR \leftarrow M[MAR]$<br>$AC \leftarrow AC + MBR$  |
| 1100   | JumpI X     | $MAR \leftarrow X$<br>$MBR \leftarrow M[MAR]$<br>$PC \leftarrow MBR$  |

/ Example 4.3

```

ORG 100
Load X           / Load the first number to be doubled.
Store Temp       / Use Temp as parameter to pass value to Subr.
JnS Subr         / Store return address, jump to the procedure.
Store X          / Store the first number, doubled
Load Y           / Load the second number to be doubled.
Store Temp
JnS Subr         / Store return address, jump to the procedure.
Store Y          / Store the second number doubled.
Halt             / End program.

```

```

X,    DEC 20
Y,    DEC 48
Temp, DEC 0

```

```

Subr, HEX 0      / Store return address here.
Load Temp       / Actual subroutine to double numbers.
Add Temp        / AC now holds double the value of Temp.
JumpI Subr      / Return to calling code.
Halt

```

## HAY QUE ALMACENAR LA DIRECCIÓN DE "RETORNO"

Jns almacena el contenido del PC en la dirección de destino y salta a destino +1.  
 JumpI es un salto indirecto: salta a la dirección almacenada en el argumento (retorno de subrutina).

## ¿RECURSIVIDAD?

costo y beneficios de agregar un registro de uso específico que almacene la dirección de retorno  
 LA PILA