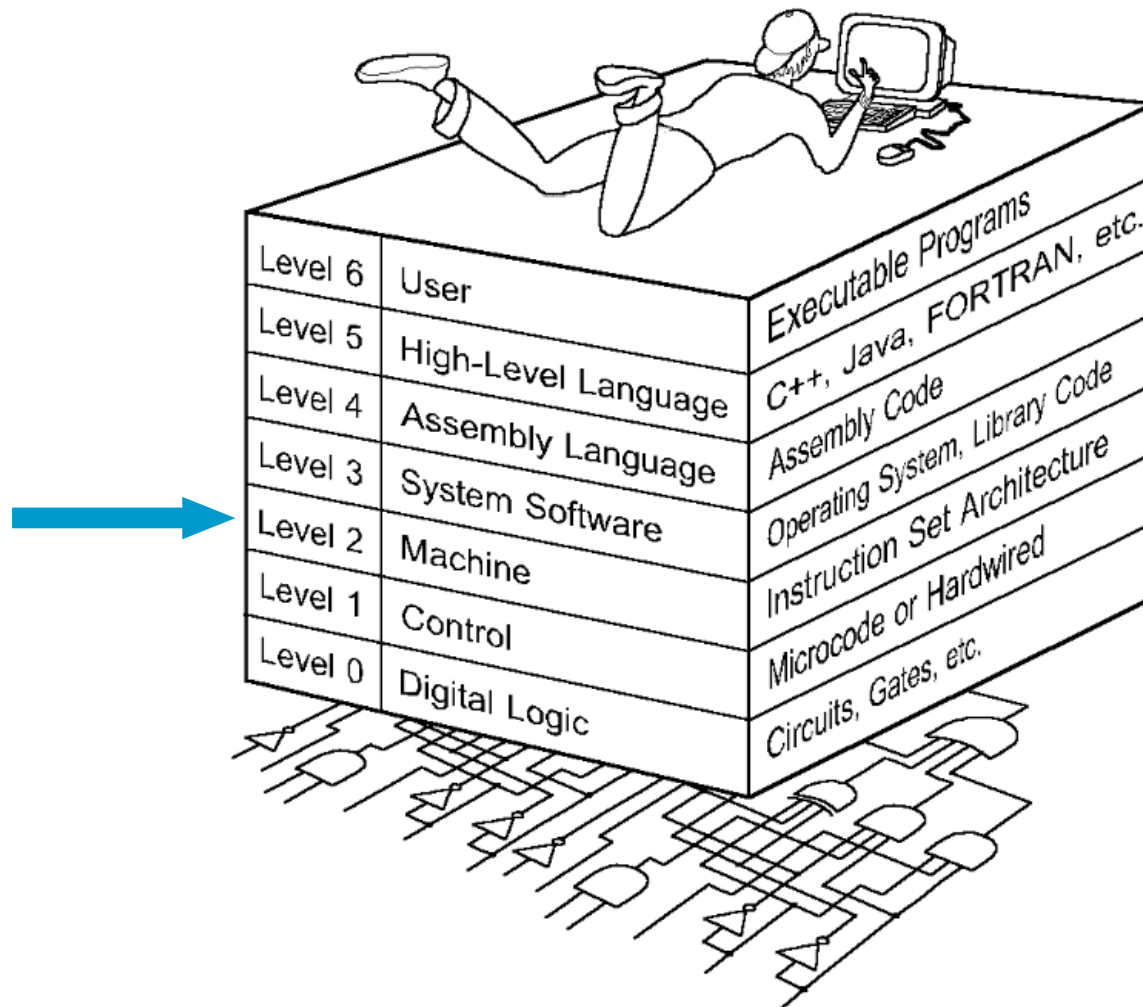


# Organización del Computador 1



CPU (ISA) – Instruction Set Architecture

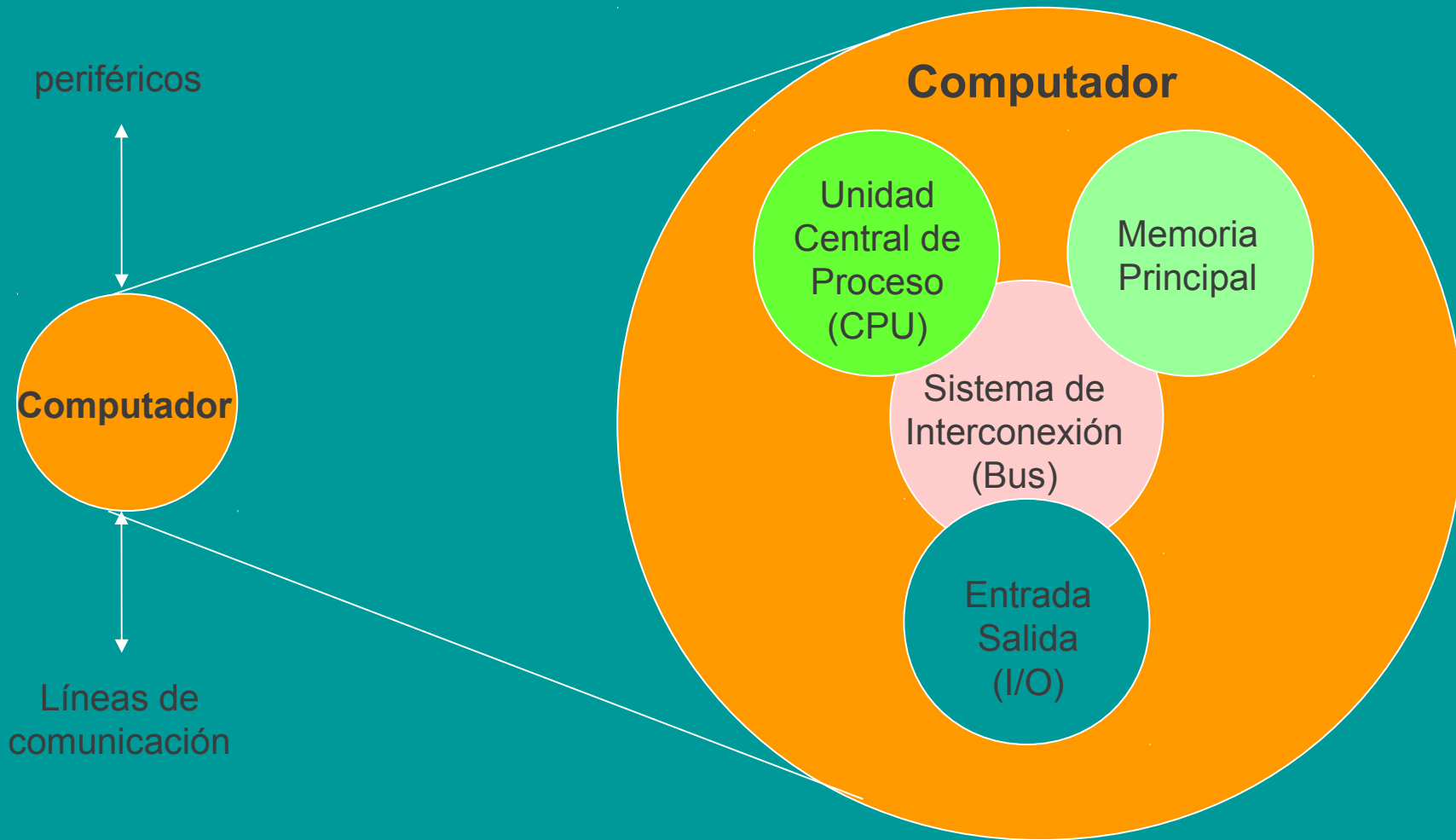
# Niveles de abstracción de una computadora



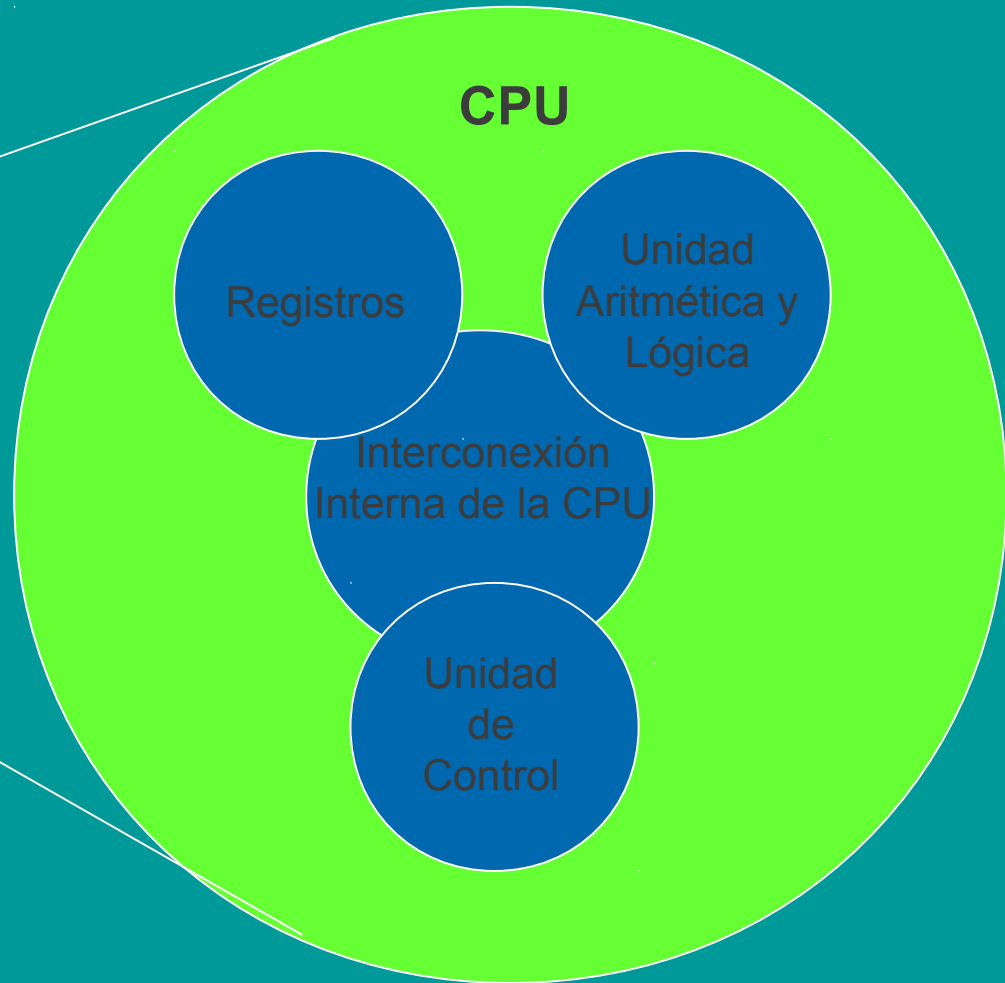
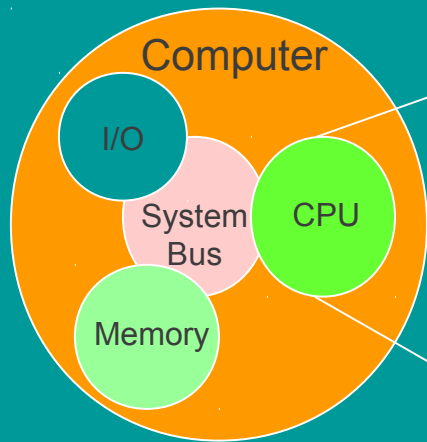
# Modelo Von Neumann



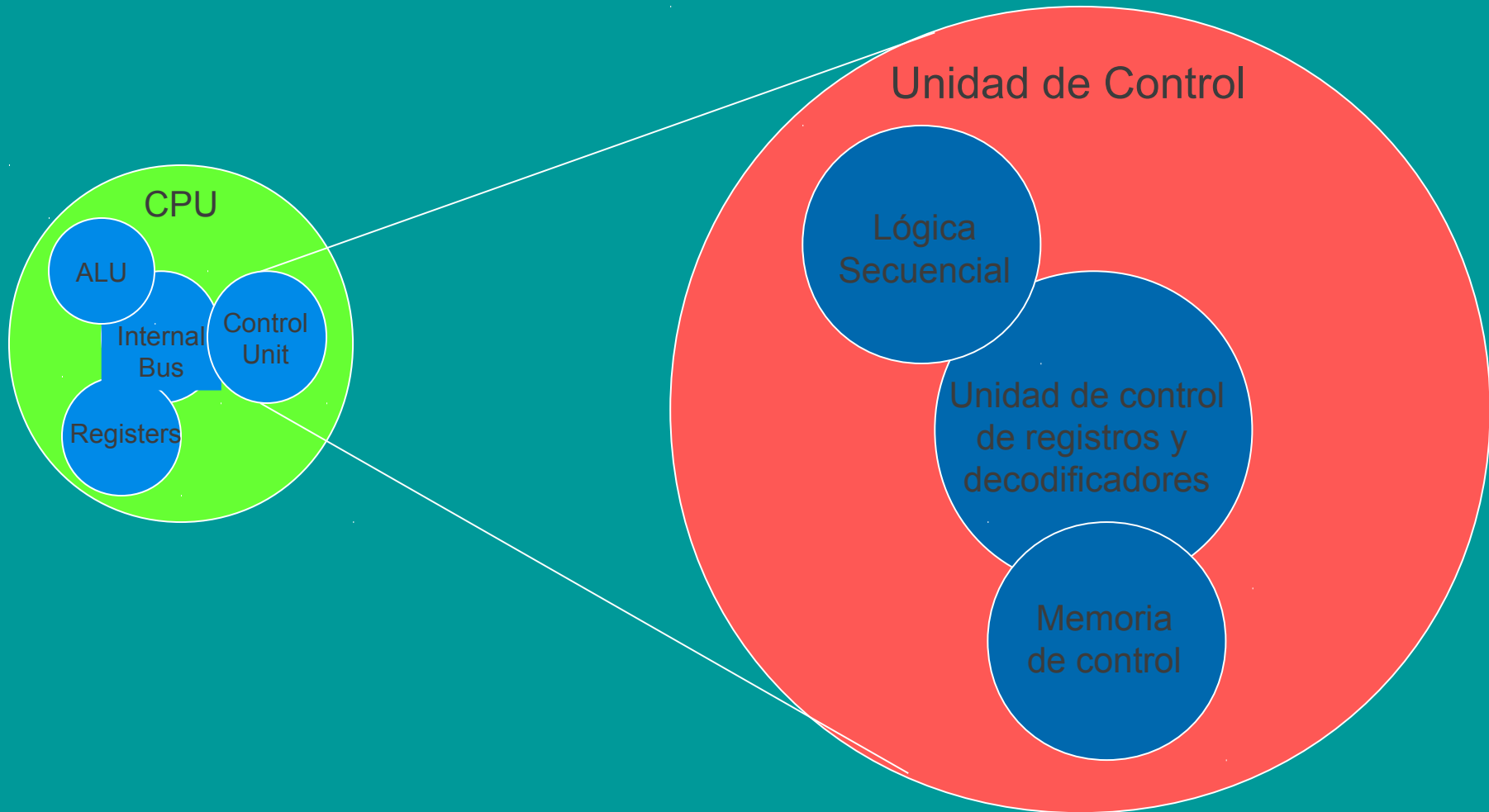
# Estructura (computadora)



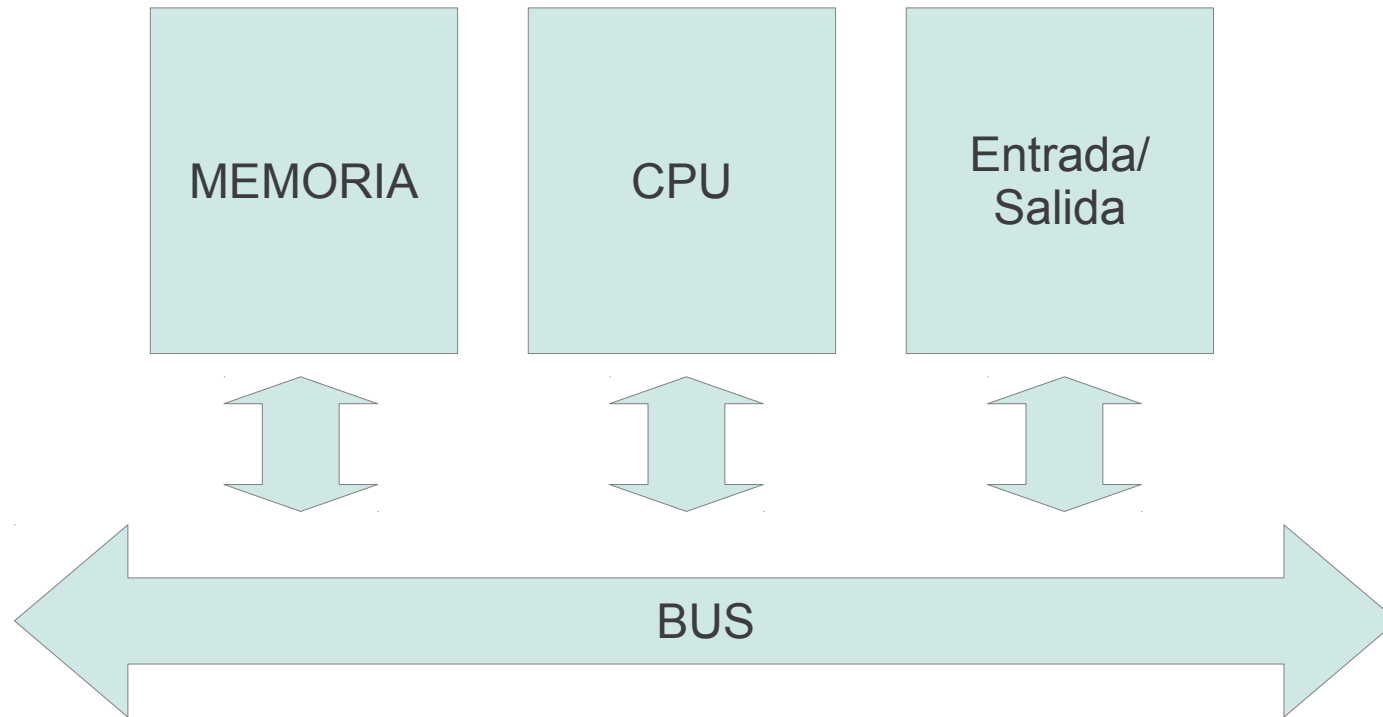
# Estructura (CPU)



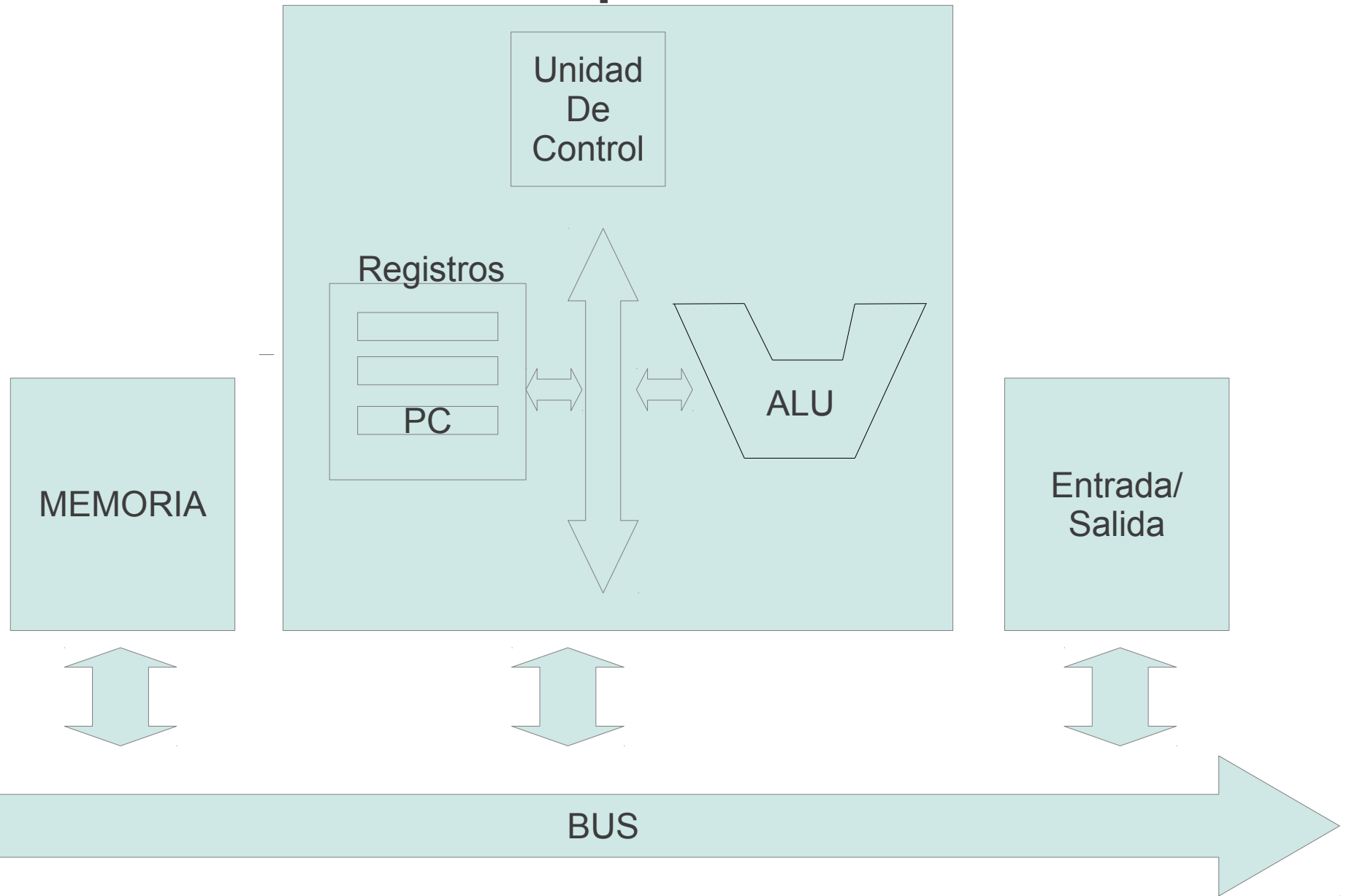
# Estructura (UC)



# Modelo Von Neumann - Organización



# Modelo Von Neumann - Microarquitectura

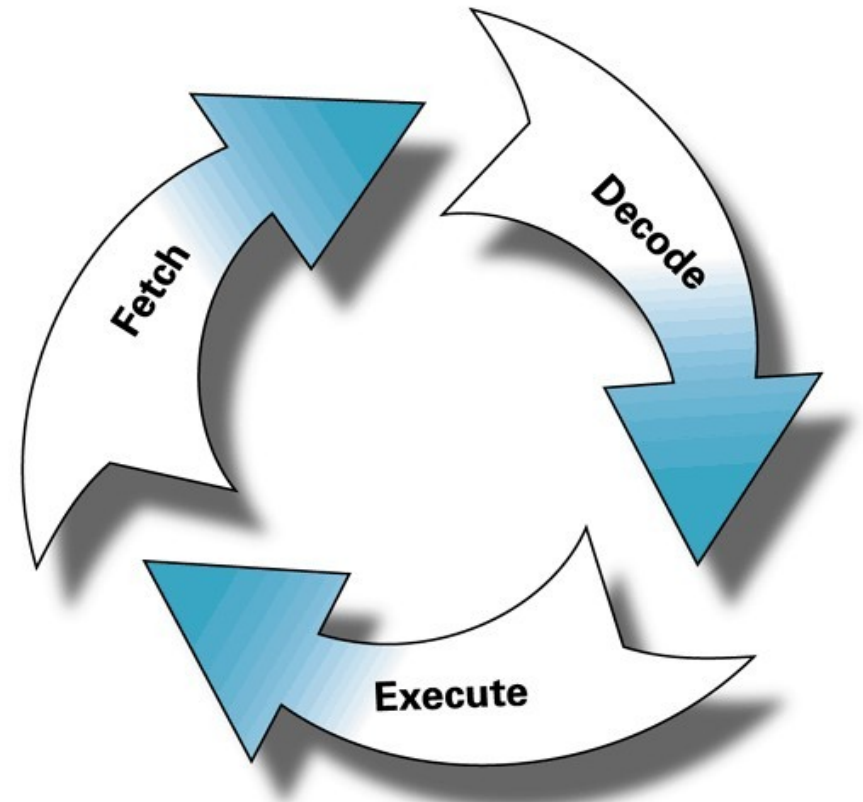


# Modelo Von Neumann - Arquitectura

| Valor de los registros | Valor de la memoria |
|------------------------|---------------------|
| R0 0000                | 0000 AD1B           |
| R1 010A                | 0001 010A           |
| ...                    | ...                 |
| PC 0141                | 0141 11FF           |
| SP FFEF                | ...                 |
|                        | FFFF 0C02           |

# Ciclo de Instrucción

- 1) UC obtiene la próxima instrucción de memoria (usando el registro PC)
- 2) Se incrementa el PC
- 3) La UC decodifica la instrucción
- 4) La UC ejecuta la instrucción (puede usar o no la ALU)
- 5) Vuelve al paso 1)



# Arquitectura

- ¿Qué preguntas debe responder?



# ISA: Instruction Set Architecture

- ¿Qué preguntas debe responder?
  - ¿Qué tipos de datos puedo manejar “nativamente”?
    - ¿Cómo se almacenan?
    - ¿Cómo se acceden?
  - ¿Qué operaciones (instrucciones) puede ejecutar?
    - ¿Cómo se codifican estas operaciones?
- No me importa organización (sólo a efectos de entender el comportamiento de mi arquitectura)

# Métrica de una ISA

- Cantidad total de instrucciones
  - Complejidad del conjunto de instrucciones
    - RISC: Reduced Instruction Set Computer
    - CISC: Complex Instruction Set Computer
- Longitud de las instrucciones
  - Cantidad de memoria que un programa requiere

# ¿Qué tipos de datos soporta?

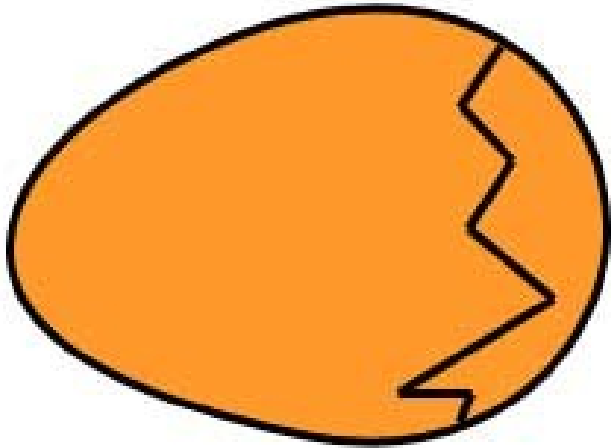
## Tipos de datos

- Enteros (8, 16, 32 bits; complemento a 2)
- Punto Flotante
- Punto Fijo
- ¿BCD, ASCII?

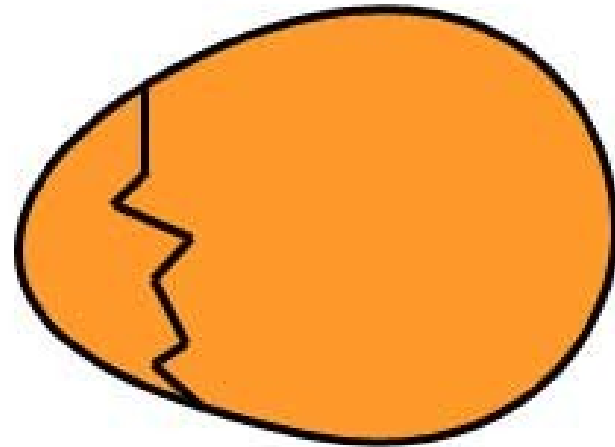
## Almacenamiento

- Big Endian
- Little Endian

# Little Endian vs. Big Endian



**BIG ENDIAN** - The way people always broke their eggs in the Lilliput land



**LITTLE ENDIAN** - The way the king then ordered the people to break their eggs

# Little Endian vs. Big Endian

- “endian” se refiere a la forma en que la computadora guarda datos multibyte.
- Por ejemplo cuando se guarda un entero de dos bytes en memoria:
  - “Little endian”: el byte en una posición de memoria menor, es menos significativo.
  - “Big endian”: el byte en una posición de memoria menor, es el más significativo.

# Little Endian vs. Big Endian

- Ejemplo: entero de dos bytes, Byte 0 menos significativo, Byte 1 más significativo.
- En “Little endian”:
  - Base address + 0 = Byte 0
  - Base address + 1 = Byte 1
- En “Big endian”:
  - Base address + 0 = Byte 1
  - Base address + 1 = Byte 0

# Acceso a los datos

- ¿Dónde se almacenan los datos?
  - Registros
  - Memoria
  - Stack
  - Espacio de E/S
- ¿Cómo se acceden a los datos?
  - Modos de direccionamiento válidos de las instrucciones
    - Directo (memoria), Indirecto (puntero en memoria), indexado a registro

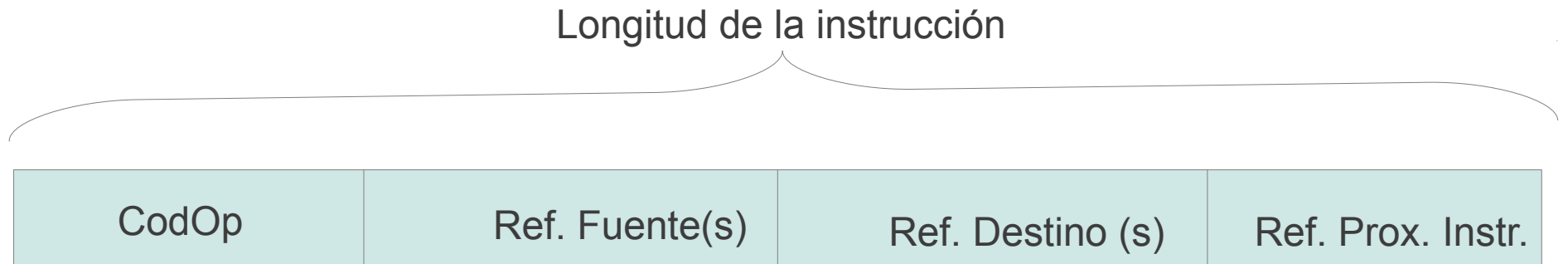
# Operaciones

¿Qué operaciones (instrucciones) puede ejecutar?

- Movimiento de datos (ej: Move, Load, Store, ...)
- Aritméticas (ej: Add, Sub, ...)
- Lógicas (ej: And, Xor,.....)
- E/S (ej: IN, OUT)
- Transferencia de control (ej: Jump, call, ret)
- Específicas (ej: MMX)

# ¿Cómo se codifican las operaciones?

- En general:



Representa la operación  
a realizar

provee información suficiente  
para obtener operandos fuente

provee información suficiente  
para obtener la ubicación del  
resultado de la operación

Provee información suficiente  
para determinar el próximo PC

# Arquitectura ejemplo: MARIE

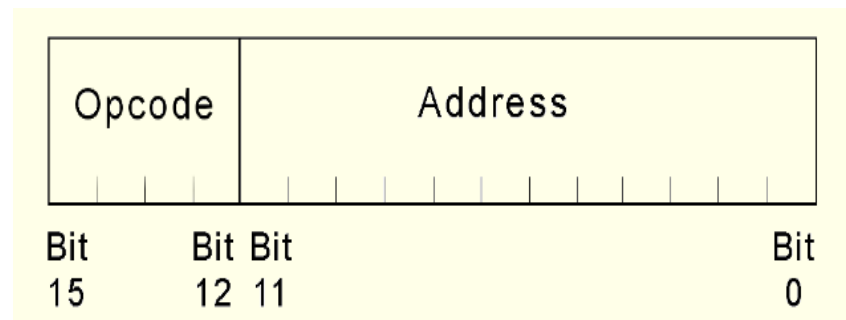
- Representación binaria, complemento a 2
- Memoria de 4K direcciones.
- Cada dirección contiene 16 bits de datos.
- Direcciones de memoria de 12 bits ( $2^{12}=4K$ )
- Máquina de Acumulador:
  - Registro Acumulador (AC) de 16 bits
  - Es el operando implícito en casi todas las operaciones

# MARIE: instrucciones

| Instrucción | Comportamiento   |
|-------------|--|
| JnS X       | $[X]=PC$ y $PC=X+1$  |
| Load X      | $AC = [X]$   |
| Store X     | $[X]= AC$  |
| Add X       | $AC = AC + [X]$  |
| Subt X      | $AC = AC - [X]$  |
| Input       | AC = Entrada de Periférico   |
| Output      | Enviar a un periférico contenido AC  |
| Halt        | Detiene la Ejecución   |
| Skip Cond   | Saltea una instrucción si se cumple<br>Si Cond=00 $\Rightarrow AC<0$<br>Si Cond=01 $\Rightarrow AC=0$<br>Si Cond=10 $\Rightarrow AC>0$ |
| Jump X      | $PC = X$   |
| Clear       | $AC = 0$   |
| Addi X      | $AC = AC + [[X]]$  |
| Jumpi X     | $PC = [X]$   |

# MARIE: Formato de instrucción

- Formato de instrucción fijo



# MARIE: Codificación instrucciones

| OpCode | Instrucción | Comportamiento  |
|--------|-------------|---|
| 0000   | JnS X       | Almacena PC en X y Salta a X+1  |
| 0001   | Load X      | $AC = [X]$  |
| 0010   | Store X     | $[X] = AC$  |
| 0011   | Add X       | $AC = AC + [X]$   |
| 0100   | Subt X      | $AC = AC - [X]$   |
| 0101   | Input       | AC = Entrada de Periférico  |
| 0110   | Output      | Enviar a un periférico contenido AC   |
| 0111   | Halt        | Detiene la Ejecución  |
| 1000   | Skip Cond   | Salta una instrucción si se cumple la condición (00=>AC<0; 01=>AC=0; 10=>AC>0)) |
| 1001   | Jump X      | PC = X  |
| 1010   | Clear       | AC = 0  |
| 1011   | Addi X      | $AC = AC + [[X]]$   |
| 1100   | Jumpi X     | PC = [X]  |

# MARIE: Ejemplo

- Sumar el contenido de 0xFF0 y 0xFF1.  
Almacenar el resultado en 0xFF2

Load 0xFF0    # AC=[0xFF0]

Add 0xFF1    # AC=[0xFF0]+[0xFF1]

Store 0xFF2    # [0xFF2]=[0xFF0]+[0xFF1]

# MARIE: Ejemplo

- Sumar el contenido de 0xFF0 y 0xFF1.  
Almacenar el resultado en 0xFF2

0001 1111 1111 0000 # Load 0xFF0

0011 1111 1111 0001 # Add 0xFF1

0010 1111 1111 0010 # Store 0xFF2



# Instrucciones de Control

- Nos permiten alterar la secuencia de ejecución del programa
  - Fundamentales para implementar IF, WHILE, etc.
  - Llamadas a procedimientos
- Se dividen de acuerdo a su ejecución
  - Saltos condicionales (debe cumplirse alguna condición)
  - Saltos incondicionales (se cambia el flujo SIEMPRE que se ejecuta la instrucción)
  - ¿Ejemplos en MARIE?

# MARIE

¿Cómo podemos escribir este programa con instrucciones de MARIE?

```
If (AC!=0)
```

```
    AC=0
```

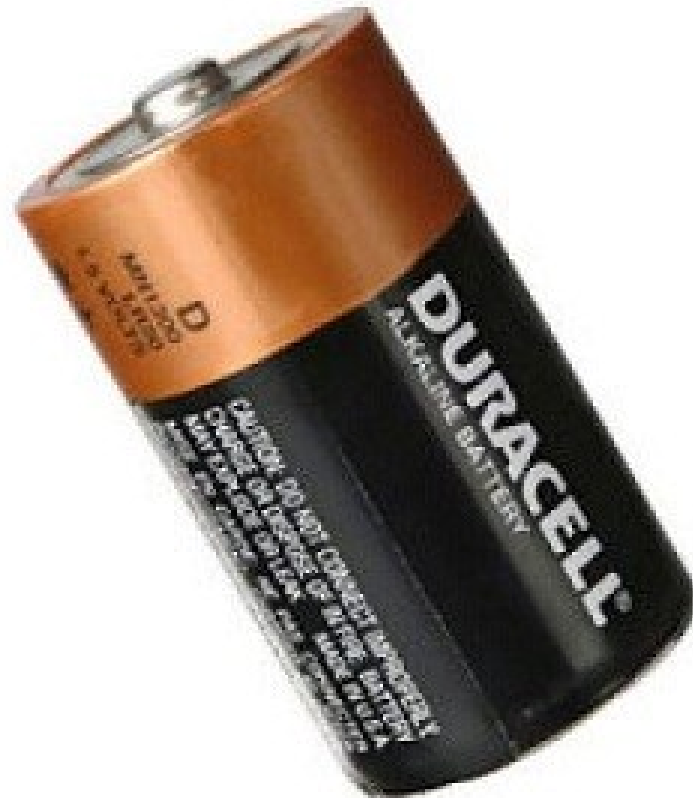
```
Endif
```

# MARIE

¿Cómo podemos escribir este programa con instrucciones de MARIE?

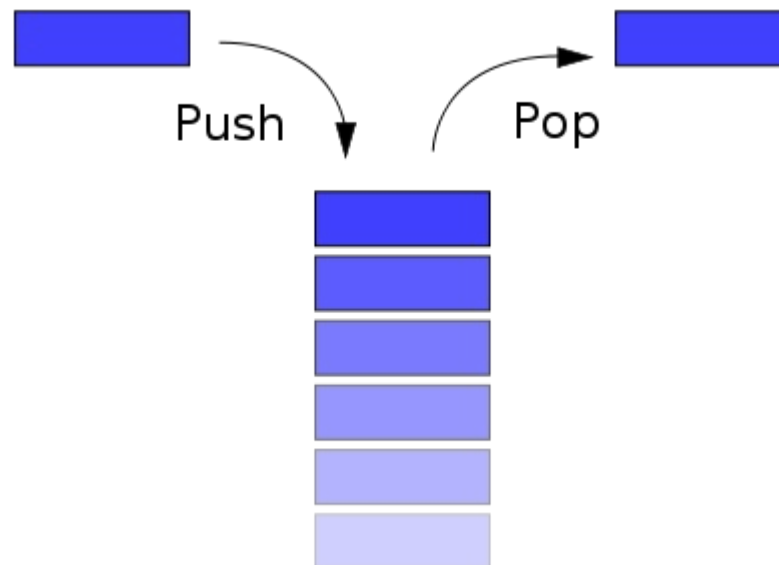
```
If (AC!=0)      SkipCond 01
    AC=0        Clear
Endif
```

# La Pila



# La Pila

- Es una estructura de datos
- Puedo solamente:
  - agregar un dato al tope de la pila (push)
  - Retirar el dato que está al tope de la pila (pop)



# Stack Machines

- Se implementa con un registro que apunta *solamente* al tope de la pila
- Las stack machines
  - Pueblan y despueblan la pila usando:
    - PUSH operando
    - POP operando
  - Las operaciones aritmético/lógicas
    - obtienen los operandos de la pila y
    - almacenan los resultados allí.

# Ejemplo: StackMARIE

- Tiene la arquitectura de MARIE pero orientada a pila

| OpCode | Instrucción | Comportamiento            |
|--------|-------------|---------------------------|
| 0000   | PUSH X      | push([x])                 |
| 0001   | POP X       | [x]=pop()                 |
| 0010   | ADD         | push(pop()+pop())         |
| 0011   | SUB         | push(pop()-pop())         |
| 0100   | AND         | push(pop() & pop())       |
| 0101   | OR          | push(pop()   pop())       |
| 0110   | NOT         | push ( ~pop())            |
| 0111   | LE          | push(pop()<=pop())        |
| 1000   | GE          | push(pop()>=pop())        |
| 1001   | EQ          | push(pop()==pop())        |
| 1010   | JUMP X      | PC=X                      |
| 1011   | JumpT X     | Si pop()==T Entonces PC=X |
| 1100   | JumpF X     | Si pop()==F Entonces PC=X |

# StackMARIE

- Sumar las direcciones 0xFF0 y 0xFF1.  
Almacenar el resultado en 0xFF2

# StackMARIE

- Sumar las direcciones 0xFF0 y 0xFF1.  
Almacenar el resultado en 0xFF2

PUSH [0xFF0]

PUSH [0xFF1]

ADD

POP [0xFF2]

# Arquitecturas GPR

- GPR: General Purpose Register
- Arquitecturas de registros de propósito general
- Podemos utilizar varios registros

# Arquitectura Máquina Orga1

## Memoria:

- Direcciones de 16 bits (de 0x0000 a 0xFFFF)
- Palabras de 16 bits
- Direccionamiento a palabra (65.535 palabras)

## CPU:

### Registros

- 8 registros de **propósito general** de 16 bits (R0 a R7)
- 3 registros de **propósito específico** de 16 bits
  - PC (program counter)
  - IR (instruction register)
  - SP (stack pointer)
- 4 **Flags**: Z, N, C y V

# Flags

- Son registros (de 1 bit) que nos dan información de control.
- En la arquitectura Orga1:
  - (Z)ero
  - (N)egative
  - (C)arry
  - o(V)erflow
- Se modifican con todas las operaciones aritmético-lógicas
  - todas salvo MOV, CALL, RET, JMP y Jxx.



# Flags: Ejemplo

- `ADD R0, R1 # R0 := R0+R1` (suma bit a bit)
  - $Z=1$  iff el resultado es `0x0000`
  - $N=1$  iff el resultado es negativo complemento a 2 de 16 bits
  - $C=1$  iff la suma bit a bit produjo acarreo
  - $V=1$  iff el resultado no es representable en complemento a 2 de 16 bits
    - `NEG + NEG = POS`
    - `POS + POS = NEG`

# Flags: Utilidad

- Los flags se utilizan para realizar saltos condicionales.

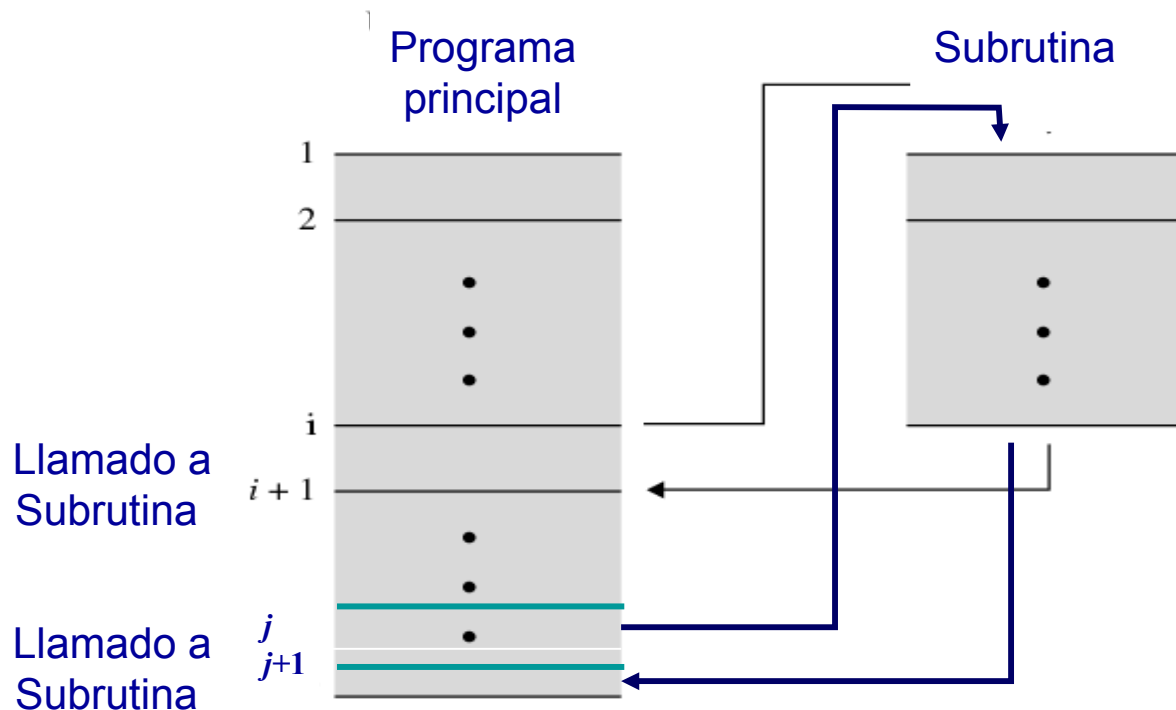
| Salto | Descripción             | Condición                |
|-------|-------------------------|--------------------------|
| JE    | Igual / Cero            | Z                        |
| JNE   | Distinto                | Not Z                    |
| JLE   | Menor o igual           | Z or ( N xor V )         |
| JG    | Mayor                   | Not ( Z or ( N xor V ) ) |
| JL    | Menor                   | N xor V                  |
| JGE   | Mayor o igual           | Not ( N xor V )          |
| JLEU  | Menor o igual sin signo | C or Z                   |
| JGU   | Mayor sin signo         | Not ( C or Z )           |
| JCS   | Carry / Menor sin signo | C                        |
| JNEG  | Negativo                | N                        |
| JVS   | Overflow                | V                        |

# Flags: Preguntas

- ¿Por qué usar Flags y no preguntar por el valor de los registros o la memoria?
- ¿Por qué hay un “Menor” y un “Menor sin signo”?

# Subrutinas

- Son alteraciones del flujo secuencial que permiten “retornar” al lugar desde donde fueron invocadas



# Instrucciones para subrutinas

- CALL dir\_subrutina
  - PC = dir\_subrutina
- ¿Pero cómo guardamos la dirección de retorno?
  - Opción a) en registros
  - Opción b) en una dirección de memoria (ejemplo: al inicio de la rutina)
  - Opción c) se escuchan opciones...

# Instrucciones para subrutinas

- ¿Pero cómo guardamos la dirección de retorno?
  - Opción a) en registros
    - ¿Y si usamos los registros? Perdemos el valor de retorno
  - Opción b) en una dirección de memoria (ejemplo: al inicio de la rutina)
    - ¿Cómo soportamos recursión/varios llamados a la misma subrutina?
  - Opción c) se escuchan opciones...
    - ¡La pila al rescate!

# Subrutinas en la pila

- CALL dir\_subrutina
  - Sirve para comenzar a ejecutar una subrutina
  - El CPU hace push(PC), y luego PC:=dir\_subrutina
- RET
  - Sirve para terminar de ejecutar una subrutina
  - El CPU hace PC:=pop()
- Adicionalmente, con PUSH/POP podríamos almacenar los argumentos y el retorno de la subrutina (**Spoiler Alert**: orga2 ad-nauseaum)

# Orga1: Instrucciones de 2 operandos

|               |               |               |                              |                             |
|---------------|---------------|---------------|------------------------------|-----------------------------|
| <i>4 bits</i> | <i>6 bits</i> | <i>6 bits</i> | <i>16 bits</i>               | <i>16 bits</i>              |
| cod. op.      | destino       | fuente        | constante destino (opcional) | constante fuente (opcional) |

| operación   | cod. op. | efecto  |
|-------------|----------|---|
| MOV $d, f$  | 0001     | $d \leftarrow f$  |
| ADD $d, f$  | 0010     | $d \leftarrow d + f$ (suma binaria)                                     |
| SUB $d, f$  | 0011     | $d \leftarrow d - f$ (resta binaria)                                    |
| AND $d, f$  | 0100     | $d \leftarrow d \text{ and } f$   |
| OR $d, f$   | 0101     | $d \leftarrow d \text{ or } f$  |
| CMP $d, f$  | 0110     | Modifica los <i>flags</i> según el resultado de $d - f$ (resta binaria) |
| ADDC $d, f$ | 1101     | $d \leftarrow d + f + \text{carry}$ (suma binaria)                      |

Formato de operandos destino y fuente.

| Modo               | Codificación | Resultado    |
|--------------------|--------------|--------------|
| Inmediato          | 000000       | c16          |
| Directo            | 001000       | [c16]        |
| Indirecto          | 011000       | [[c16]]      |
| Registro           | 100rrr       | Rrrr         |
| Indirecto registro | 110rrr       | [Rrrr]       |
| Indexado           | 111rrr       | [Rrrr + c16] |

c16 es una constante de *16 bits*.

Rrrr es el registro indicado por los últimos tres *bits* del código de operando.

Las instrucciones que tienen como destino un operando de tipo *inmediato* son consideradas como inválidas por el procesador, excepto el CMP.

# Orga1: instrucciones de 1 operando

*Tipo 2a:* Instrucciones de un operando destino.

|               |               |               |                              |
|---------------|---------------|---------------|------------------------------|
| <i>4 bits</i> | <i>6 bits</i> | <i>6 bits</i> | <i>16 bits</i>               |
| cod. op.      | destino       | 000000        | constante destino (opcional) |

| operacin | cod. op. | efecto                                   |
|----------|----------|--|
| NEG $d$  | 1000     | $d \leftarrow 0 - d$ (resta binaria)     |
| NOT $d$  | 1001     | $d \leftarrow \text{not } d$ (bit a bit) |

*Tipo 2b:* Instrucciones de un operando fuente.

|               |               |               |                             |
|---------------|---------------|---------------|-----------------------------|
| <i>4 bits</i> | <i>6 bits</i> | <i>6 bits</i> | <i>16 bits</i>              |
| cod. op.      | 000000        | fuentes       | constante fuente (opcional) |

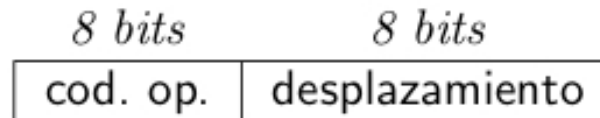
| operacin | cod. op. | efecto  |
|----------|----------|---|
| JMP $f$  | 1010     | $PC \leftarrow f$   |
| CALL $f$ | 1011     | $[SP] \leftarrow PC, SP \leftarrow SP - 1, PC \leftarrow f$ |

# Orga1: instrucciones sin operandos

|               |               |               |
|---------------|---------------|---------------|
| <i>4 bits</i> | <i>6 bits</i> | <i>6 bits</i> |
| cod. op.      | 000000        | 000000        |

| operacin | cod. op. | efecto                                       |
|----------|----------|--|
| RET      | 1100     | $PC \leftarrow [SP+1], SP \leftarrow SP + 1$ |

# Orga1: saltos condicionales



| Codop     | Operacin | Descripcin              | Condición de Salto       |
|-----------|----------|-------------------------|--------------------------|
| 1111 0001 | JE       | Igual / Cero            | Z                        |
| 1111 1001 | JNE      | Distinto                | not Z                    |
| 1111 0010 | JLE      | Menor o igual           | Z or ( N xor V )         |
| 1111 1010 | JG       | Mayor                   | not ( Z or ( N xor V ) ) |
| 1111 0011 | JL       | Menor                   | N xor V                  |
| 1111 1011 | JGE      | Mayor o igual           | not ( N xor V )          |
| 1111 0100 | JLEU     | Menor o igual sin signo | C or Z                   |
| 1111 1100 | JGU      | Mayor sin signo         | not ( C or Z )           |
| 1111 0101 | JCS      | Carry / Menor sin signo | C                        |
| 1111 0110 | JNEG     | Negativo                | N                        |
| 1111 0111 | JVS      | Overflow                | V                        |

Si se cumple la condición  $PC := PC + \text{signExt}(\text{desplazamiento})$

# ISAs

- La mayoría de las arquitecturas actuales son GPR
  - Memoria-Memoria: donde 2 o 3 operandos pueden estar en memoria
  - Registro-Memoria: donde al menos un operando es un registro (en el CPU)
  - Load-Store: donde las operaciones son sólo entre registros

# Orga1: Seguimiento del Ciclo de Instrucción



# Tarea

- Sean A,B,C,D direcciones de memoria de la arquitectura correspondiente, escribir un programa que realice  $D := (A - B) + C$  en
  - MARIE (acumulador)
  - StackMARIE (pila)
  - Orga1 (GPR)

# Bibliografía

- Capítulos 4 y 5 del Null
  - Van a servir para más temas...