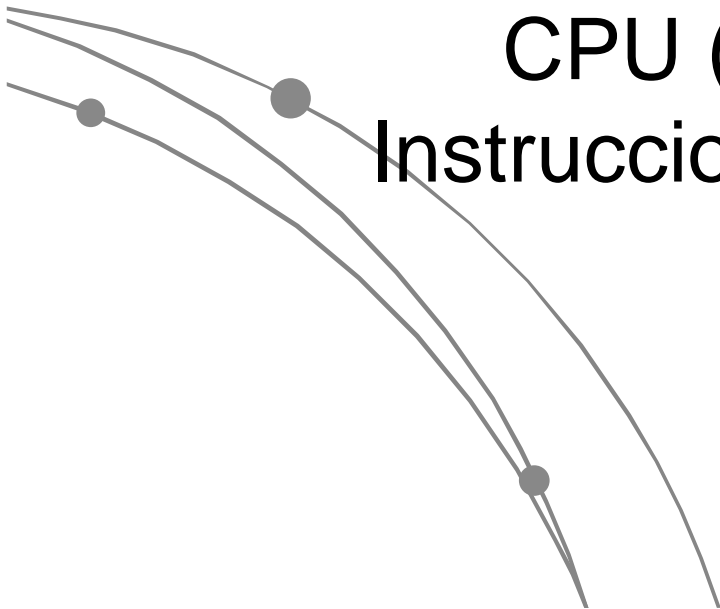
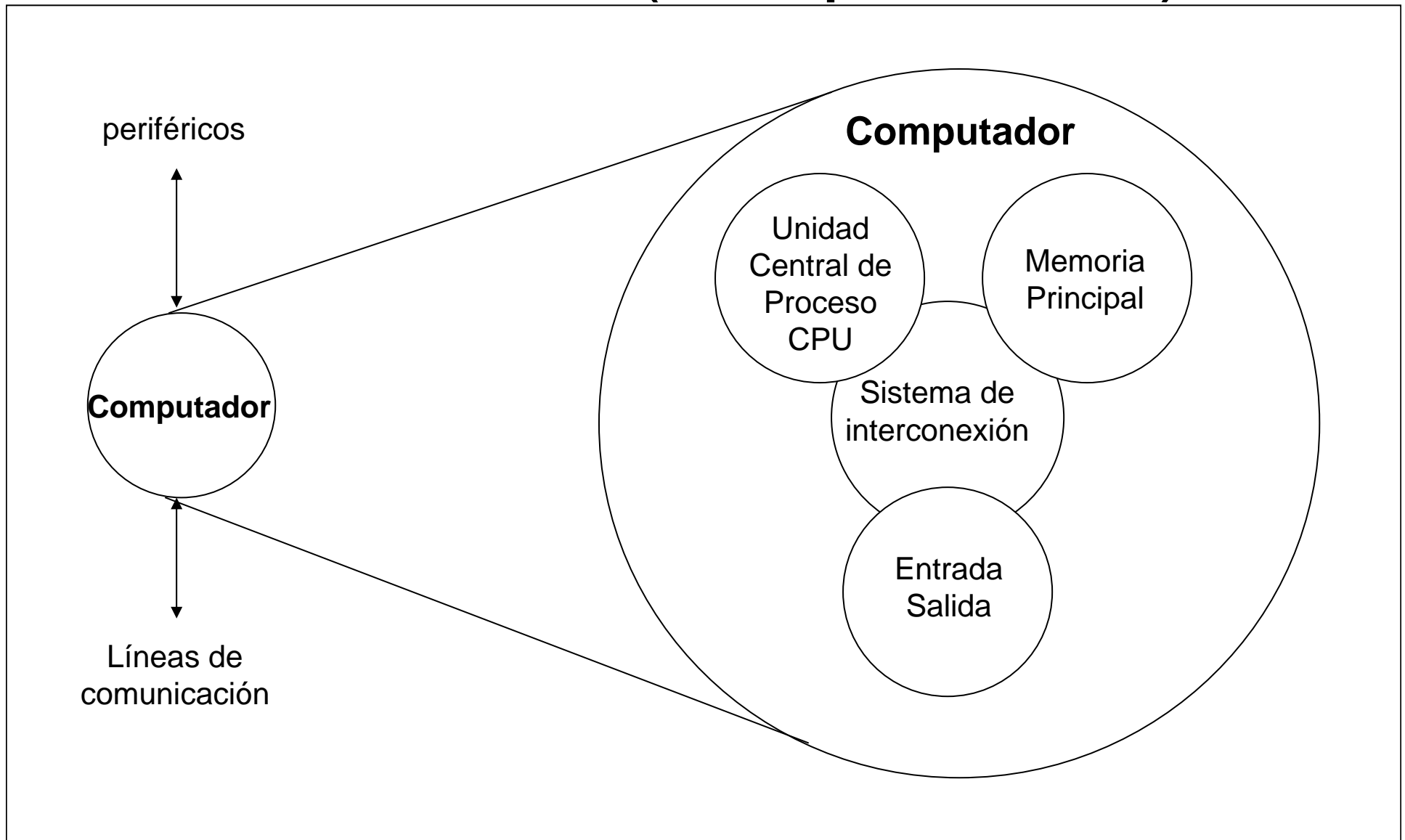


# Organización del Computador

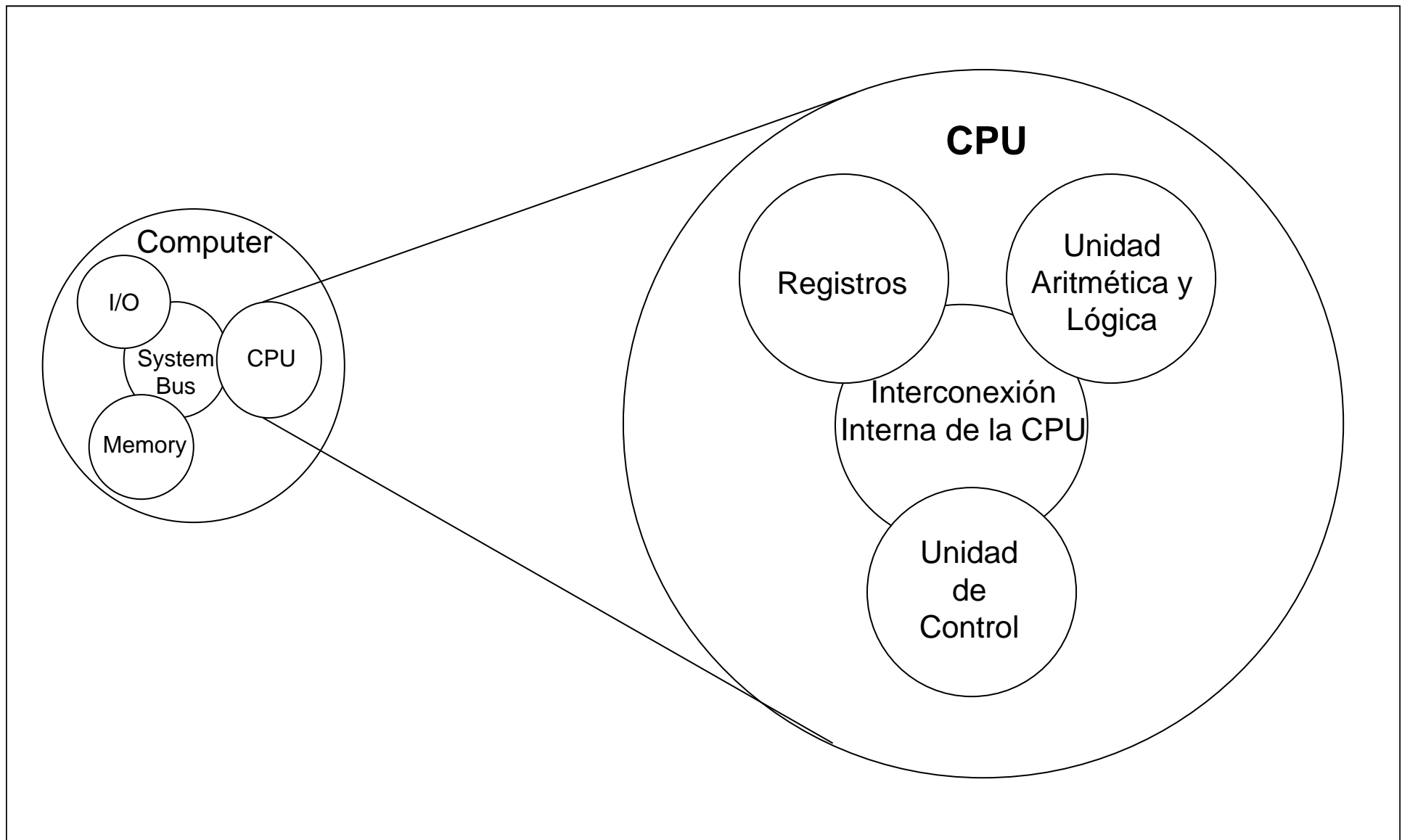
CPU (ISA)– Conjunto de Instrucciones de la Arquitectura



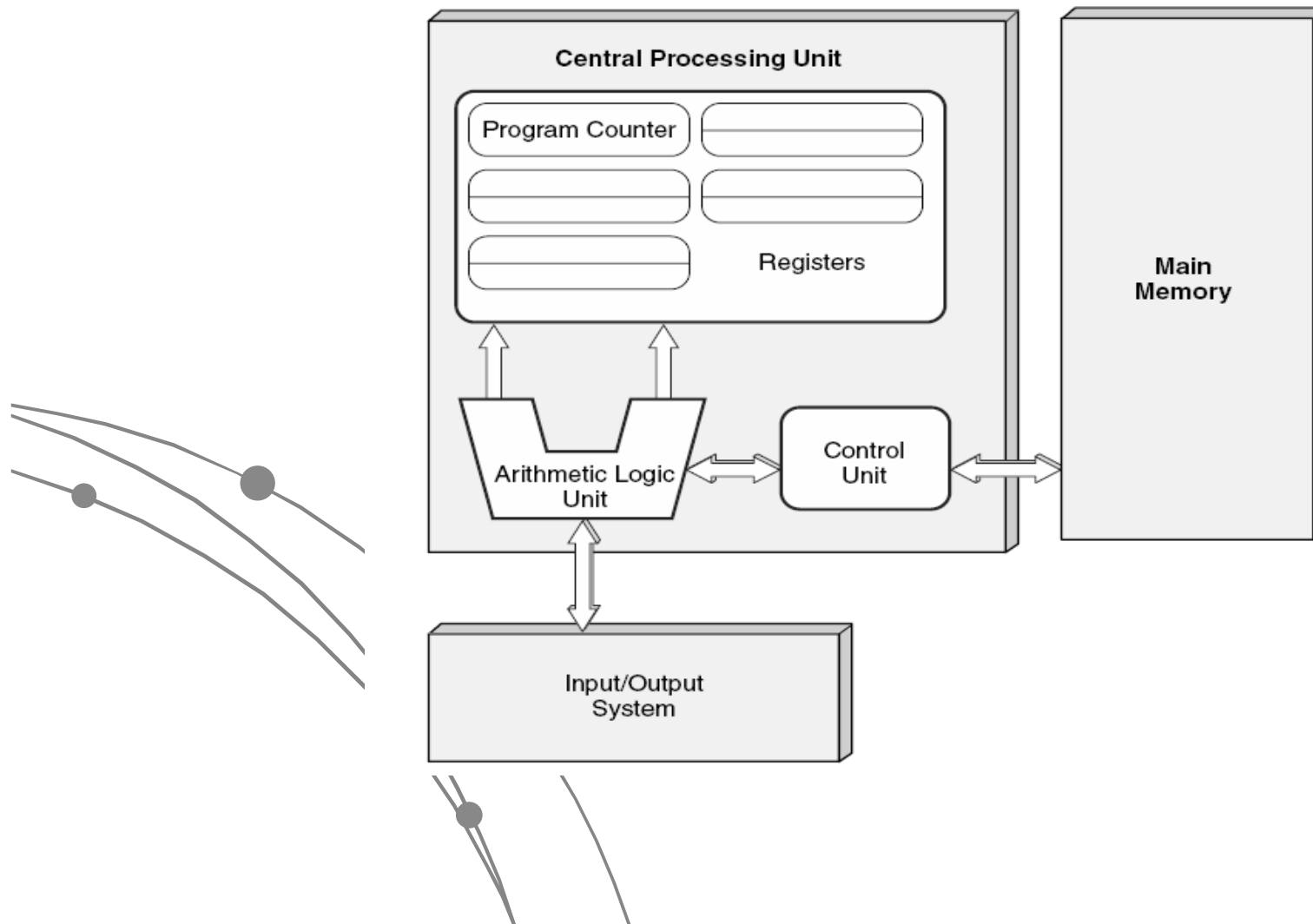
# Estructura (computadora)



# Estructura (CPU)

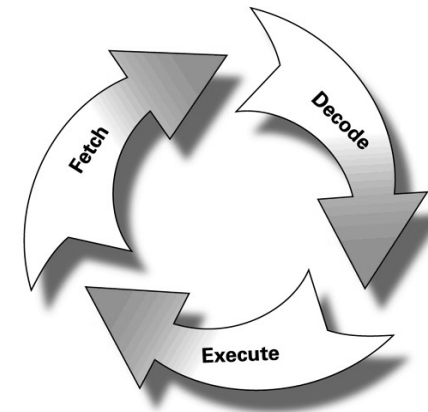


# Estructura de una máquina von Neumann

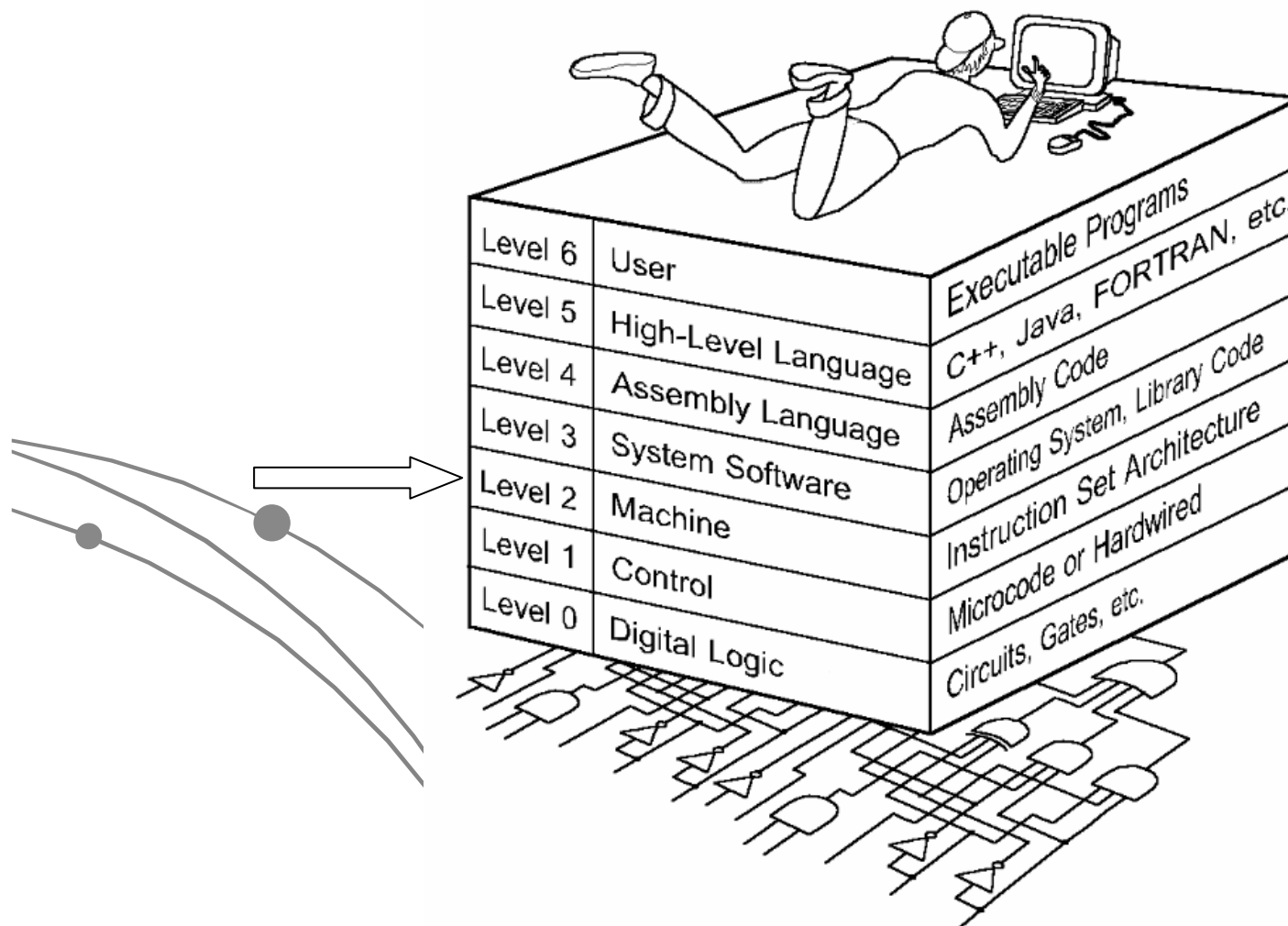


# Ciclo de Ejecución

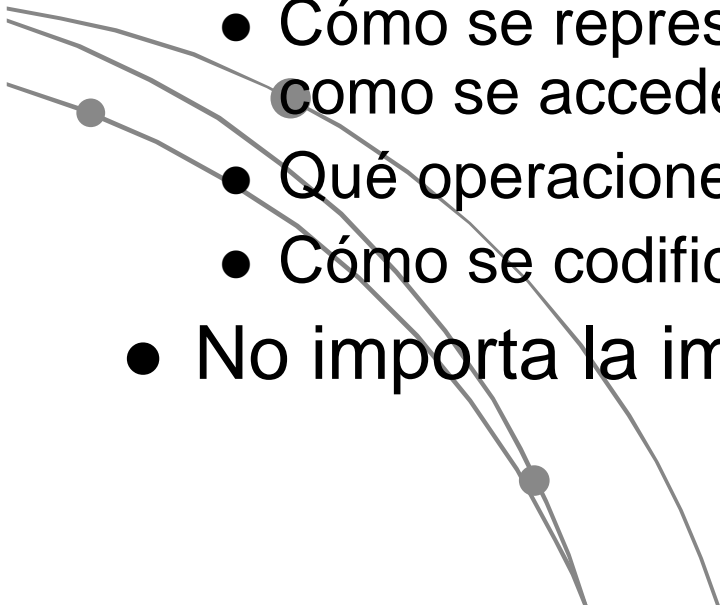
1. UC obtiene la próxima instrucción de memoria (usando el registro PC).
2. Se incrementa el PC.
3. La instrucción es decodificada a un lenguaje que entiende la ALU.
4. Obtiene de memoria los operandos requeridos por la operación.
5. La ALU ejecuta y deja los resultados en registros o en memoria.
6. Repetir paso 1.



# Los niveles de una computadora

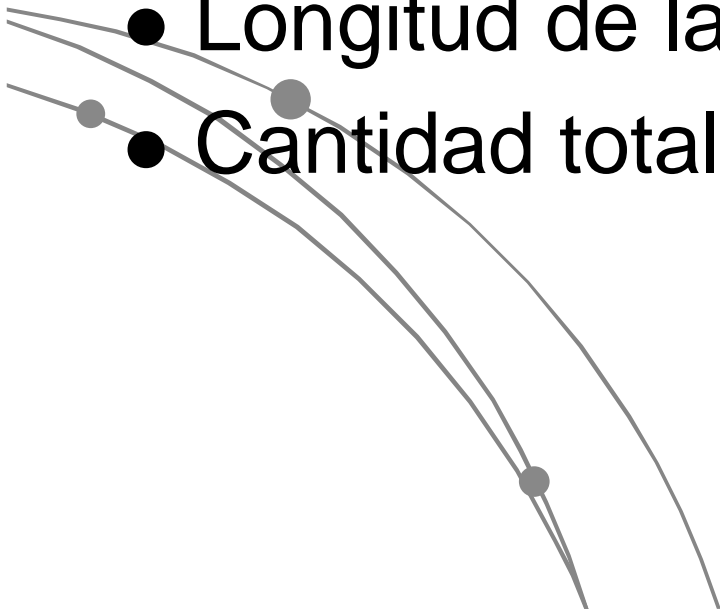


# ISA

- Nivel de Lenguaje de Máquina (Instruction Set Architecture).
  - Límite entre Hardware-Software.
  - Es lo que vemos como programadores
  - Define:
    - Cómo se representan los datos, como se almacenan, como se acceden
    - Qué operaciones se pueden realizar
    - Cómo se codifican estas operaciones
  - No importa la implementación interna.
- 

# Métricas de una ISA

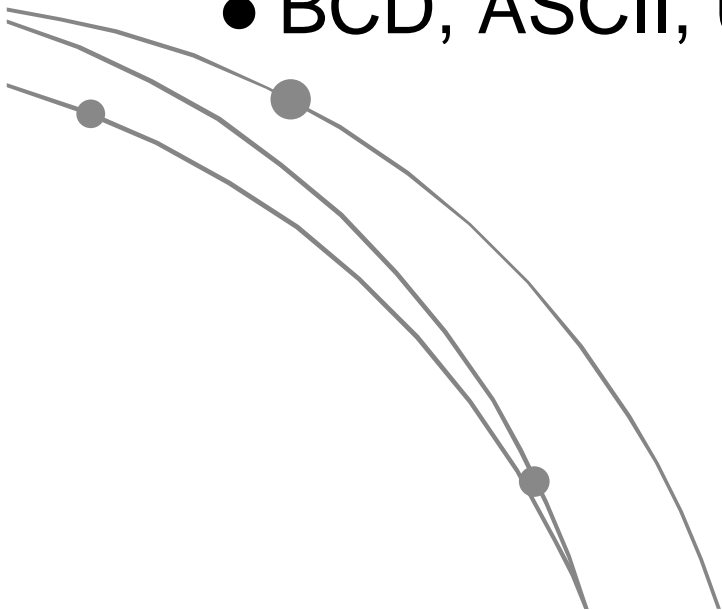
- Cantidad de memoria que un programa requiere.
- Complejidad del conjunto de instrucciones (por ejemplo RISC vs CISC).
- Longitud de las instrucciones.
- Cantidad total de instrucciones.



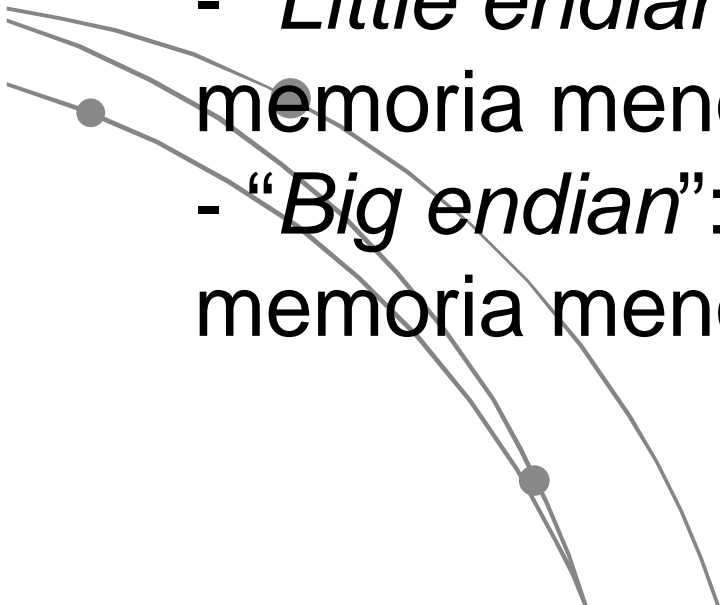


# Cómo se representan los datos?

- Tipos de datos
  - Enteros (8, 16, 32... bits, complemento a 2?).
  - Big-endian, Little endian.
  - Punto Flotante.
  - BCD, ASCII, UNICODE?



# Little 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 vs Big endian

- Ejemplo: entero de dos bytes, Byte 0 menos significativo, Byte 1 más significativo.

- “*Little endian*”:

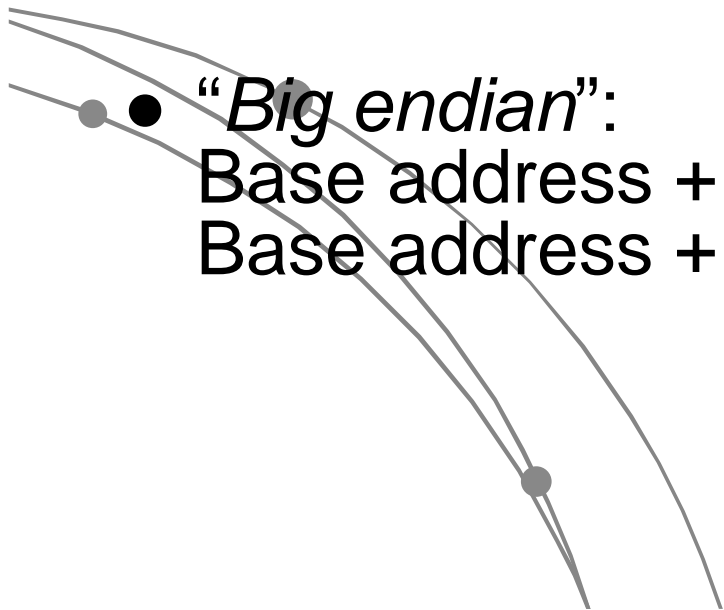
Base address + 0 = Byte 0

Base address + 1 = Byte 1

- “*Big endian*”:

Base address + 0 = Byte 1

Base address + 1 = Byte 0

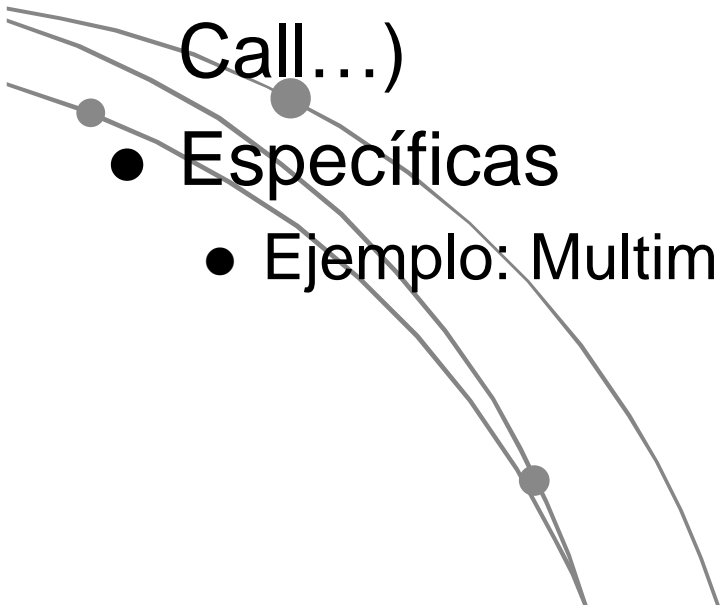


# Acceso a los datos

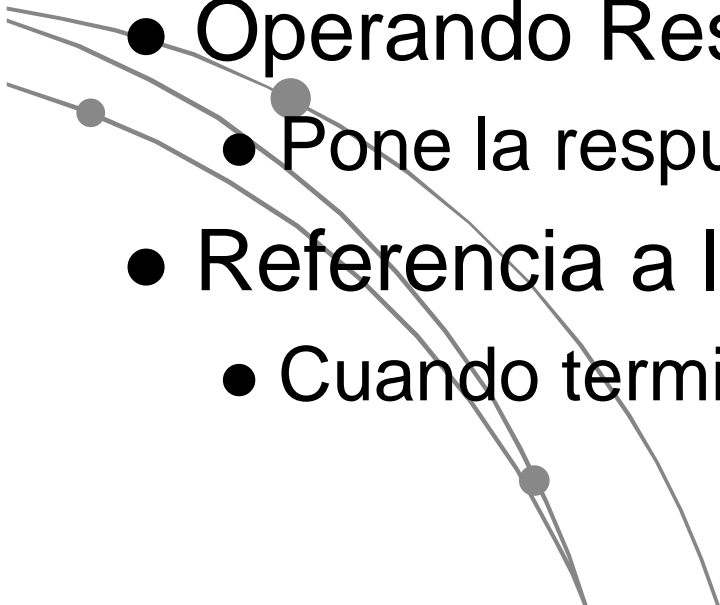
- Donde se Almacenan?
  - Registros
  - Memoria
  - Stack
  - Espacio de I/O
- Como se acceden?
  - Modos de Direcccionamiento



# Operaciones

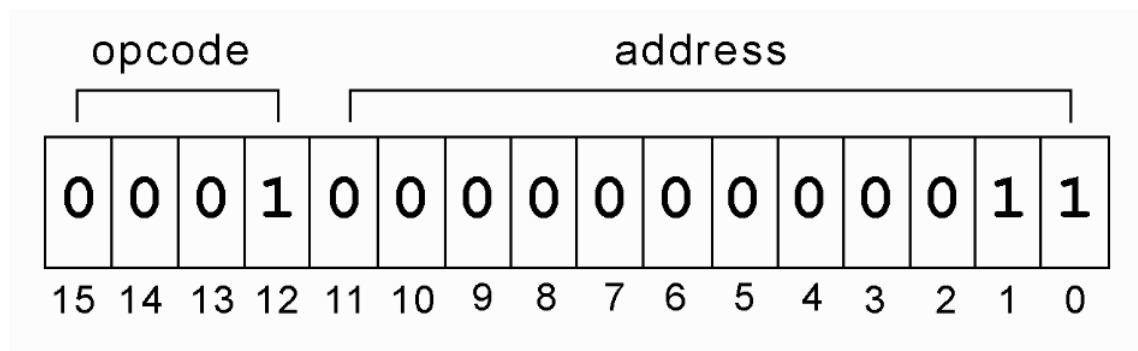
- Movimiento de datos (Move, Load, Store,...)
  - Aritméticas (Add, Sub, ...)
  - Lógicas (And, Xor, ...)
  - I/O.
  - Transferencia de Control (Jump, Skip, Call...)
  - Específicas
    - Ejemplo: Multimedia
- 

# Codificación

- Códigos de operación (OpCode)
    - Representa la operación ...
  - Operando/s Fuente
    - A realizar sobre estos datos ...
  - Operando Resultado
    - Pone la respuesta aquí ...
  - Referencia a la próxima instrucción
    - Cuando termina sigue por aquí ...
- 

# Ejemplo: Marie

- Instrucción LOAD en el IR:



- Opcode=1, Cargar en el AC el dato contenido en la dirección 3.

# Características de ISAs

- Tipos típicos de arquitecturas:
  1. Orientada a Stack.
  2. Con acumulador.
  3. Con registros de propósito general.
- Los “tradeoffs”:
  - Simpleza del hardware.
  - Velocidad de ejecución.
  - Facilidad de uso.





# Características de ISAs

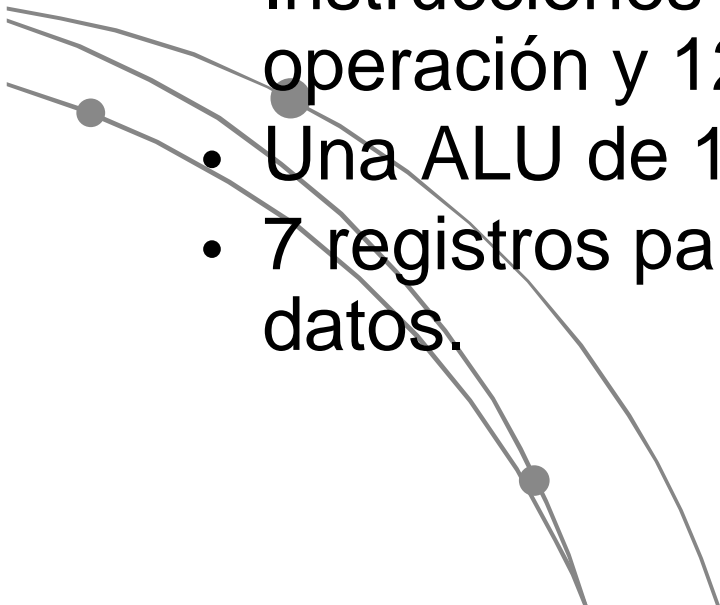
- Arquitectura Stack: las instrucciones y operandos son tomados implícitamente del *stack*
  - El *stack* no puede ser accedido de forma aleatoria (sino a través de un orden).
- Arquitectura acumulador: En cualquier operación binaria un operando está implícito (el acumulador)
  - El otro operando suele ser la memoria, generando cierto tráfico en el bus.
- Arquitectura con registros de propósito general (GPR): los registros pueden ser utilizados en lugar de la memoria
  - Más rápido que la de acumulador.
  - Implementaciones eficientes usando compiladores.
  - Instrucciones más largas... (2 ó 3 operandos).

# Instrucciones (Posibles)

	Stack	Acumulador	Registros
Movimiento	PUSH X POP X	LOAD X STORE X	Mov R1,R2 Load R1,X Store X,R1
Aritméticas	Add Sub	Add X Sub X	Add R1,R2,R3 Sub R1,R2,R3
Lógicas	And Or Not LE GE Eq	And X Or X Not	And R1,R2,R3 Or R1,R2,R3 Not R1, R2 Cmp R1,R2
Control	Jump X JumpT X JumpF X Call/Ret X	Jump X Jump (AC) SkipCond Call X/Ret	Jump X Jump R Jump Cond R Jump Cond X Call X/R Ret

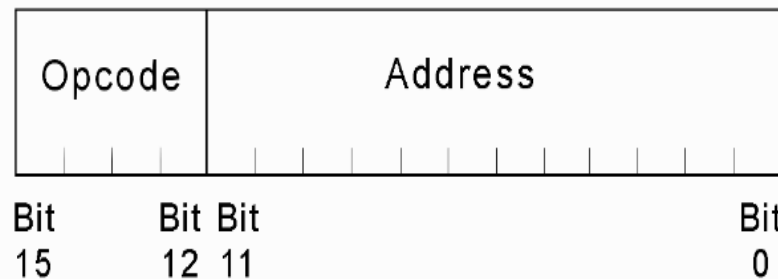
# MARIE

## Máquina de Acumulador:

- Representación binaria, complemento a 2.
  - Instrucciones de tamaño fijo.
  - Memoria accedida por palabras de 4K.
  - Palabra de 16 bits.
  - Instrucciones de 16 bits, 4 para el código de operación y 12 para las direcciones.
  - Una ALU de 16 bits.
  - 7 registros para control y movimiento de datos.
- 

# MARIE

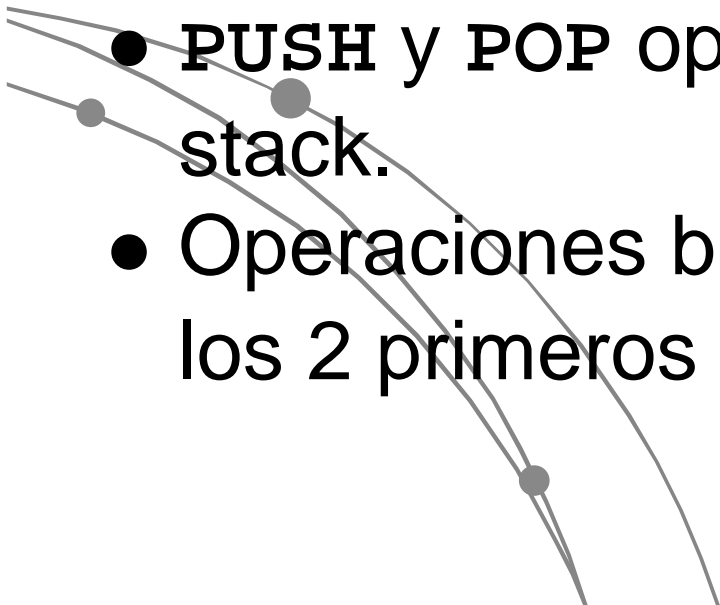
- Registros Visibles:
  - AC: Acumulador
    - 16 bits
    - Operando implícito en operaciones binarias
    - También para condiciones.
  - Formato de instrucción fijo



# MARIE

OpCode	Instrucción	
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	SkipCond Cond	Salta una instrucción si se cumple la condición (00=>AC<0; 01=>AC=0; 10=>AC>0))
1001	Jump Dir	PC = Dir
1010	Clear	AC = 0
1011	Addi X	AC = AC + [ [ X ] ]
1100	Jumpi X	PC = [X]

# Stack Machines

- Las Stack Machines **no usan** operandos en las instrucciones
  - Salvo **PUSH X** y **POP X** que requieren una dirección de memoria como operando.
  - El resto obtienen los operandos del stack.
  - **PUSH** y **POP** operan sólo con el tope del stack.
  - Operaciones binarias (ej. **ADD**, **MULT**) usan los 2 primeros elementos del stack.
- 

# Stack Machines

- Utilizan notación polaca inversa

- Jan Lukasiewicz (1878 - 1956).

- Notación infija:  $Z = X + Y$ .

- Notación postfija:  $Z = X Y +$

- No necesitan paréntesis!

- infija  $Z = (X \times Y) + (W \times U),$

- postfija  $Z = X Y \times W U \times +$

# Comparativa (Asignaciones)

$$Z = X \times Y + W \times U$$

Stack	1 Operando	Registros 2 Operandos	Registros 3 Operandos
PUSH X PUSH Y MULT PUSH W PUSH U MULT ADD POP Z	LOAD X MULT Y STORE TEMP LOAD W MULT U ADD TEMP STORE Z	LOAD R1,X MULT R1,Y LOAD R2,W MULT R2,U ADD R1,R2 STORE Z,R1	MULT R1,X,Y MULT R2,W,U ADD Z,R1,R2



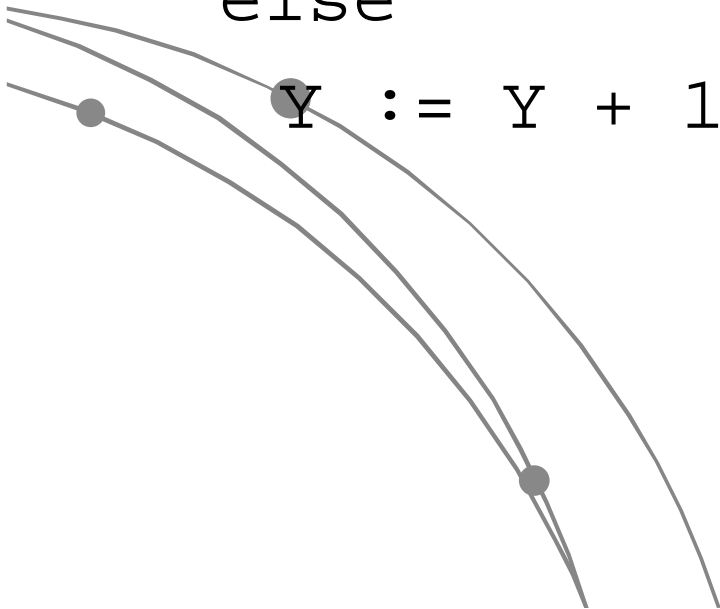
# Comparativa (IF)

- Escribir este programa en diferentes Arquitecturas:

```
if X > 1 do  
    X := X + 1;
```

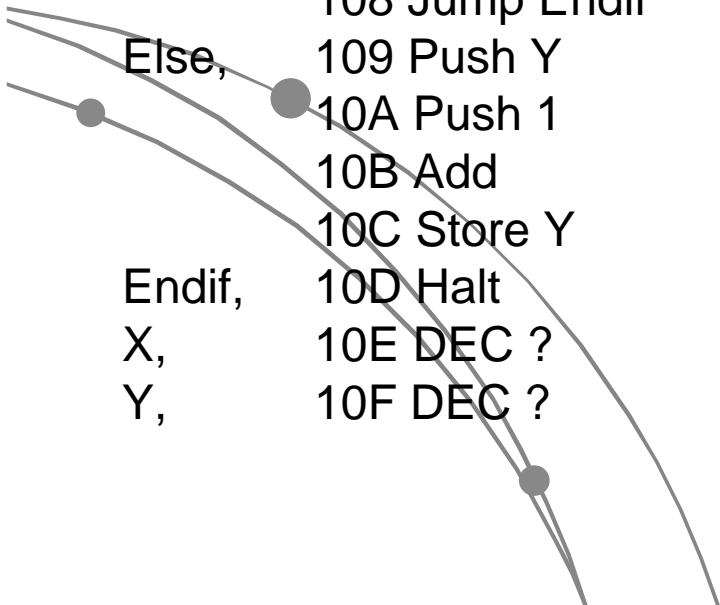
```
else
```

```
    Y := Y + 1;
```



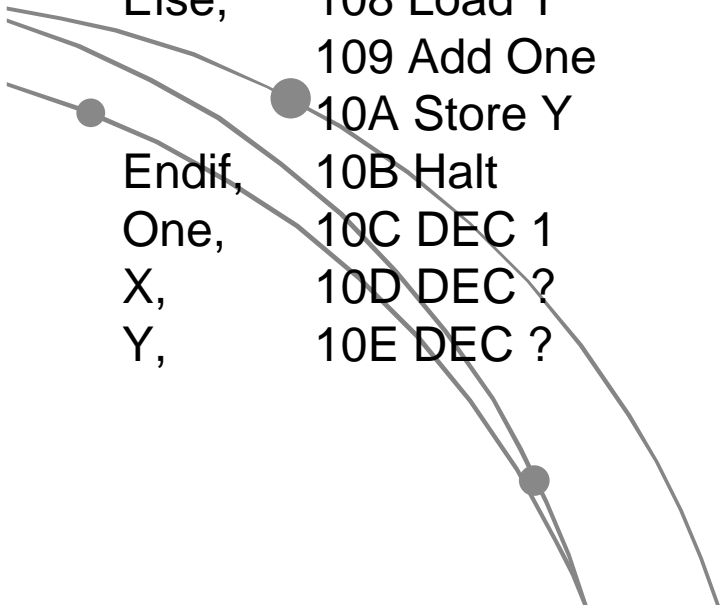
# Solución Stack

If,	100 Push X	
	101 Push 1	
	102 LE	/ Si $X \leq 1$ pone un 1 el el stack
	103 JMPT Else	/ Si 1 va a else. Saca el 1 0 del Stack
Then,	104 Push X	
	105 Push 1	
	106 Add	
	107 Pop X	/ $X := X + 1$
	108 Jump Endif	/Salta la parte ELSE
Else,	109 Push Y	
	10A Push 1	
	10B Add	
	10C Store Y	/ $Y := Y + 1$
Endif,	10D Halt	/Terminar
X,	10E DEC ?	/ Espacio para X
Y,	10F DEC ?	/ Espacio para Y



# Solución MARIE (Acumulador)

If,	100 Load X	/ AC=[X]
	101 Subt One	/ AC=AC-1
	102 Skipcond 600	/ Si $AC \leq 0$ ( $X \leq 1$ ) Saltear
	103 Jump Else	/Salto a ELSE
	104 Load X	/ AC=[X]
Then,	105 Add One	/AC=AC+1
	106 Store X	/X:= X + 1
	107 Jump Endif	/Salta la parte ELSE
Else,	108 Load Y	/Load Y
	109 Add One	/Add 1
	10A Store Y	/Y:= Y + 1
Endif,	10B Halt	/Terminar
One,	10C DEC 1	/ One = 1
X,	10D DEC ?	/ Espacio para X
Y,	10E DEC ?	/ Espacio para Y



# Solución Máquina Práctica

If,        100 Mov R1, [X]  
            101 CMP R1,1  
            102 JLE Else

Then,     105 Add R1,1  
            106 Mov [X],R1  
            107 Jump Endif

Else,      108 Mov R1,[Y]  
            109 Add R1,1  
            10A Mov [Y],R1

Endif,    10B Ret

X, 10C DEC ?

Y, 10D DEC ?

/X:= X + 1

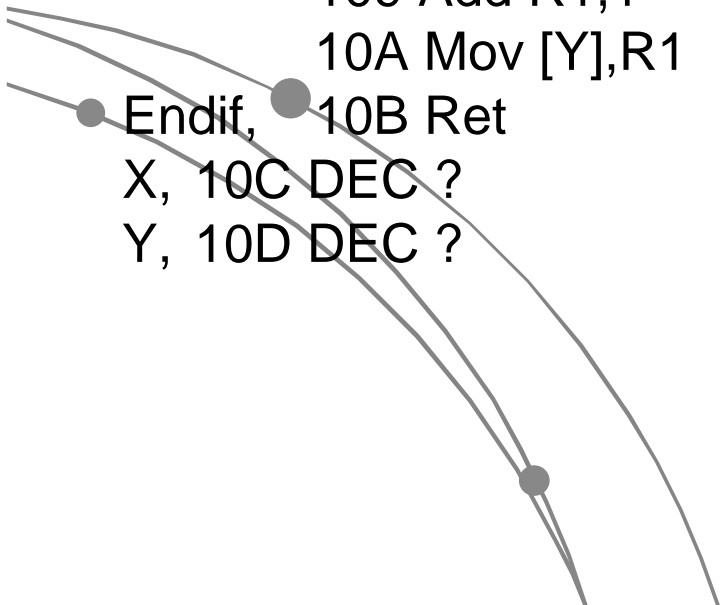
/Salta la parte ELSE

/Y:= Y + 1

/ Aquí termina

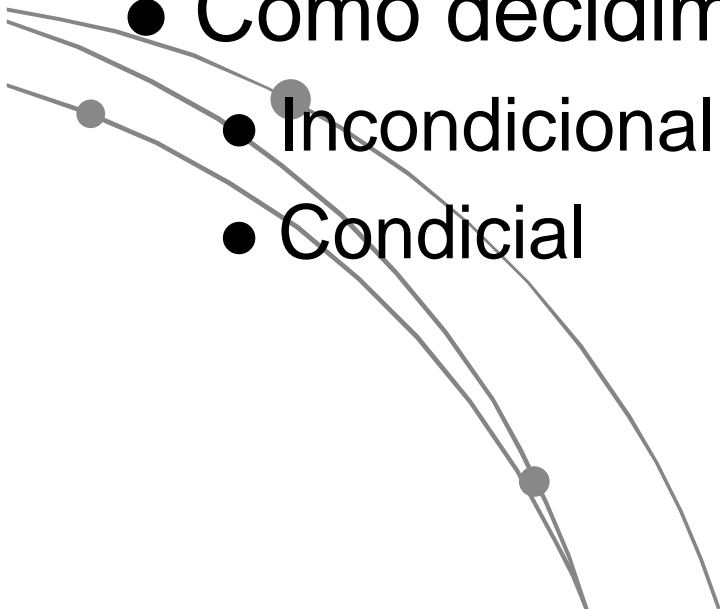
/ Espacio para X

/ Espacio para Y

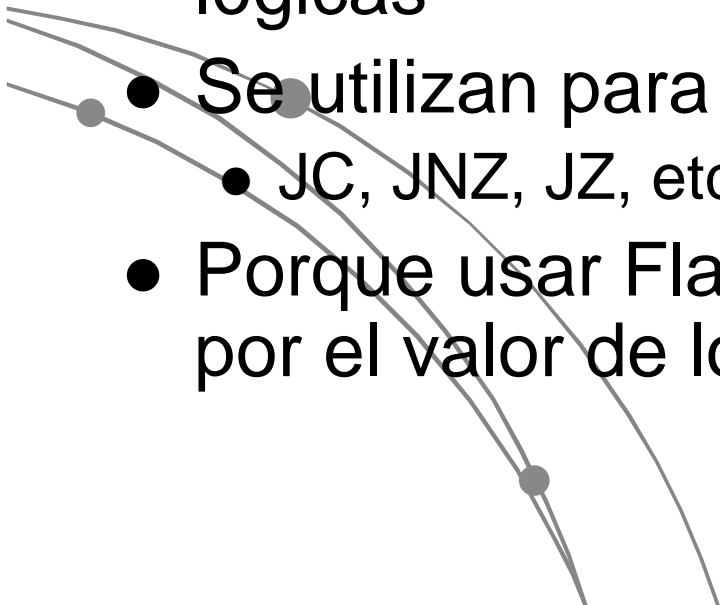


# Control

- Nos permiten alterar la secuencia del programa
  - Fundamental para implementar IF, While, etc
  - Llamadas a procedimientos
- Como decidimos la alteración de control?
  - Incondicional
  - Condicional



# Flags

- Registros que nos dan información de control
    - Carry
    - Overflow, UnderFlow
    - Zero, Negative
  - Se modifican con operaciones aritméticas y lógicas
  - Se utilizan para realizar saltos condicionales
    - JC, JNZ, JZ, etc
  - Porque usar Flags y no directamente preguntar por el valor de los registros o memoria?
- 

# Comparativa (Ciclos)

- Hacer este programa:

```
int A[5]={10,15,20,25,30}
```

```
Sum=0
```

```
i:=0
```

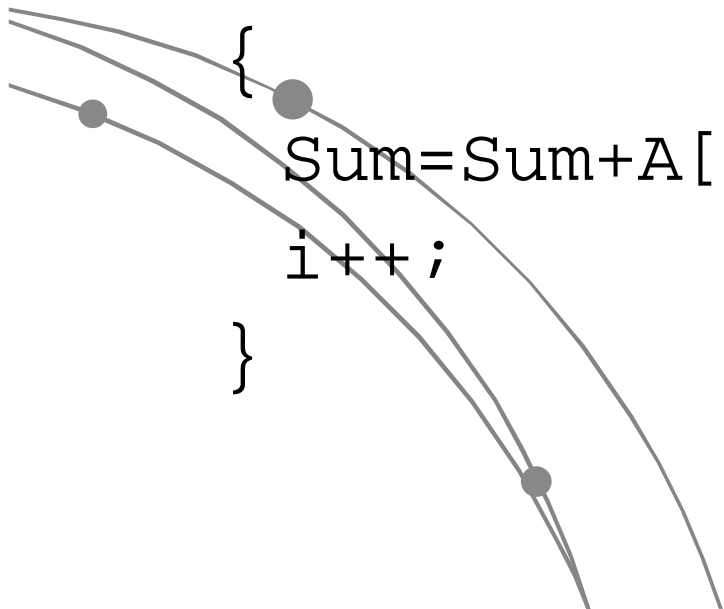
```
While(i<5)
```

```
{
```

```
Sum=Sum+A[i];
```

```
i++;
```

```
}
```



# Ejemplo Marie

Start,	Load Num	; Cargo el Contador
	SkipCond 800	; Si AC<=0 Fin (Si
	Jump End	
	Load Sum	; AC=[Sum]
	Addi PostAct	; AC+=[[PostAct]]
	Store Sum	; [SUM]=AC
	Load PostAct	; AC=[PostAct]
	Add Index	;
	Store PostAct	; [PosAct]=[PosAct]+Index
	Load Num	; AC=[Num]
	Sub One	
	Store Num	; [Num]=[Num]-1
	Jump Start	
End,	Halt	
Vec,	Dec 10	; Num1 =10
	Dec 15	; Cada palabra del array es inicializada
	Dec 20	
	Dec 25	
	Dec 30	
PostAct,	Hex OffSet Vec	; Aquí hay que poner una dirección
Num,	Dec 5	; contador para el loop=5
Sum,	Dec 0	; Suma=0
Index,	Dec 1	
One,	Dec 1	



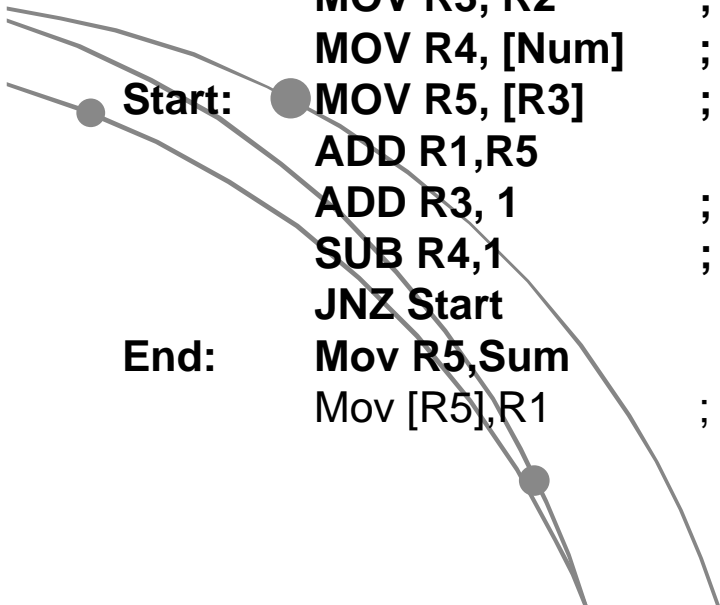
# Ejemplo Máquina Práctica

**Vec**      **DW 10**    ; Num1 =10  
            **DW 15**    ; Cada palabra del array es inicializada  
            **DW 20**  
            **DW 25**  
            **DW 30**

**Num**      **DB 5**      ; contador para el loop=5  
**Sum**      **DB 0**      ; Suma=0

**MOV R2, Vec**            ; Apunta al Vector  
            **MOV R1, 0**            ; Acumulador  
            **MOV R3, R2**            ; inicializa el offset (Dentro del array)  
            **MOV R4, [Num]**        ; R4 es el registro contador  
**Start:**    **MOV R5, [R3]**            ; El dato del vector (R5 = [ [ Vec [i] ] ] )  
            **ADD R1,R5**  
            **ADD R3, 1**            ; Avanza dentro del vector  
            **SUB R4,1**            ; Decrementa el contador  
            **JNZ Start**

**End:**      **Mov R5,Sum**  
            **Mov [R5],R1**        ; Grabo el resultado en Sum



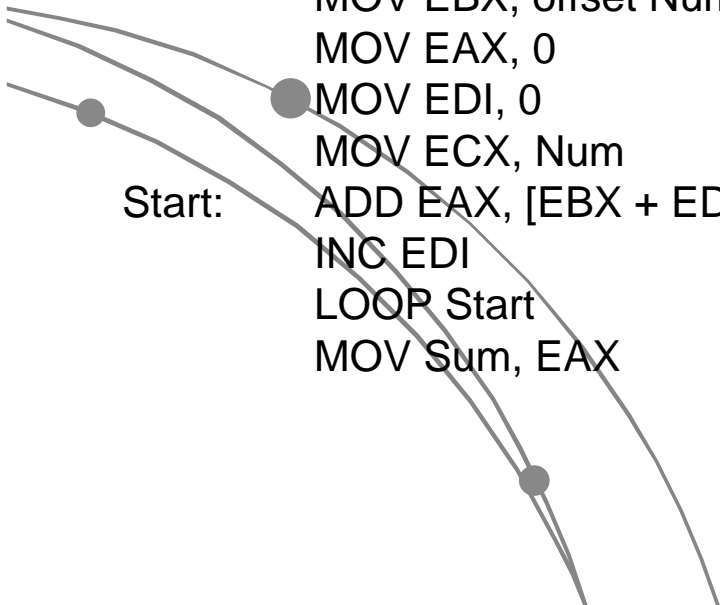
# Ejemplo Intel

## .DATA

```
Num1    EQU 10 ; Num1 =10
        EQU 15 ; Cada palabra del array es inicializada
        EQU 20
        EQU 25
        EQU 30
Num      DB 5 ; contador para el loop=5
Sum      DB 0 ; Suma=0
```

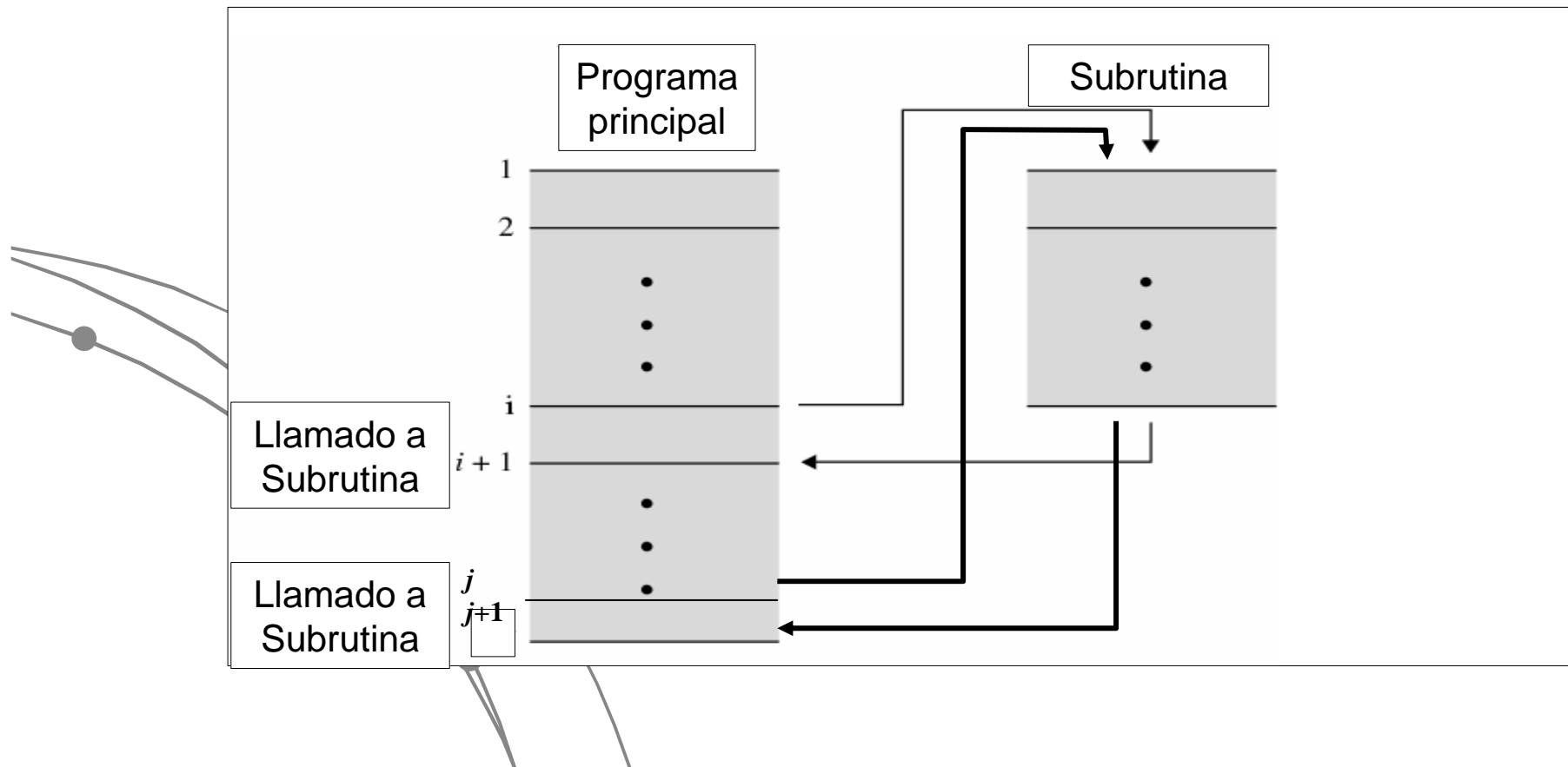
## .CODE

```
Start:   MOV EBX, offset Num1    ; Carga la direccion de Num1 en EBX
        MOV EAX, 0               ; inicializa la suma
        MOV EDI, 0               ; inicializa el offset (Dentro del array)
        MOV ECX, Num             ; ECX es el registro contador
        ADD EAX, [EBX + EDI * 4] ; Suma el EBX-esimo numero a EAX
        INC EDI                  ; Incrementa el offset
        LOOP Start                ; CX=CX-1, SI CX>0 Vuelve a Start
        MOV Sum, EAX             ; Guarda el resulatdo en Sum
```



# Subrutinas

- Reutilización de código
  - Funciones, Procedimientos, Métodos



# Subrutinas

- CALL subrutina
  - PC= subrutina
- Pero como guardamos la dirección de retorno? (el viejo PC). Y Los argumentos??
  - En registros?
    - Y si lo usamos?
  - En memoria? Por ejemplo al comienzo de la rutina...
    - Recursión?
  - Mejor usar un Stack!
  - Otro enfoque: Ventana de registros

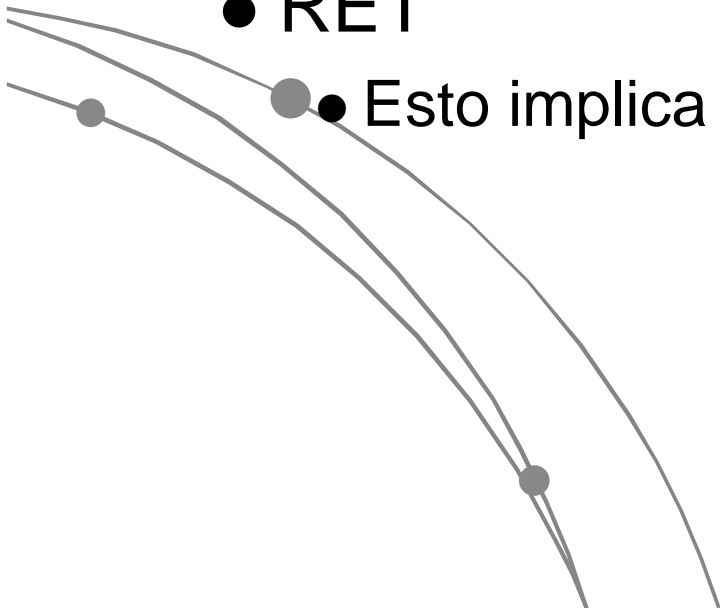
# Subrutinas

## Código en llamador:

- CALL subrutina
  - Esto implica (PUSH PC, PC = rutina)

## Código en subrutina

- RET
  - Esto implica (POP PC)



# Subrutinas Ejemplo

- Se desea hacer un programa que realice las siguientes operaciones:

[0100]  $\leftarrow$  [0100] + 5  
[0200]  $\leftarrow$  [0200] + 5  
[0204]  $\leftarrow$  [0204] + 5

¿¿Que pasa si estábamos usando R2??

## Programa principal

**MOV R1,0100**  
**CALL SUM5**  
**MOV R1,0200**  
**CALL SUM5**  
**LOAD R1,0204**  
**CALL SUM5**

## Subrutina

**SUM5: MOV R2,[X]**  
**ADD R2,5**  
**MOV [X],R2**  
**RET**

# Subrutinas Ejemplo

- Se desea hacer un programa que realice las siguientes operaciones:

$[0100] \leftarrow [0100] + 5$

$[0200] \leftarrow [0200] + 5$

$[0204] \leftarrow [0204] + 5$

## Programa principal

**MOV R1,0100**

**CALL SUM5**

**MOV R1,0200**

**CALL SUM5**

**LOAD R1,0204**

**CALL SUM5**

## Subrutina

**SUM5: PUSH R2**

**MOV R2,[X]**

**ADD R2,5**

**MOV [X],R2**

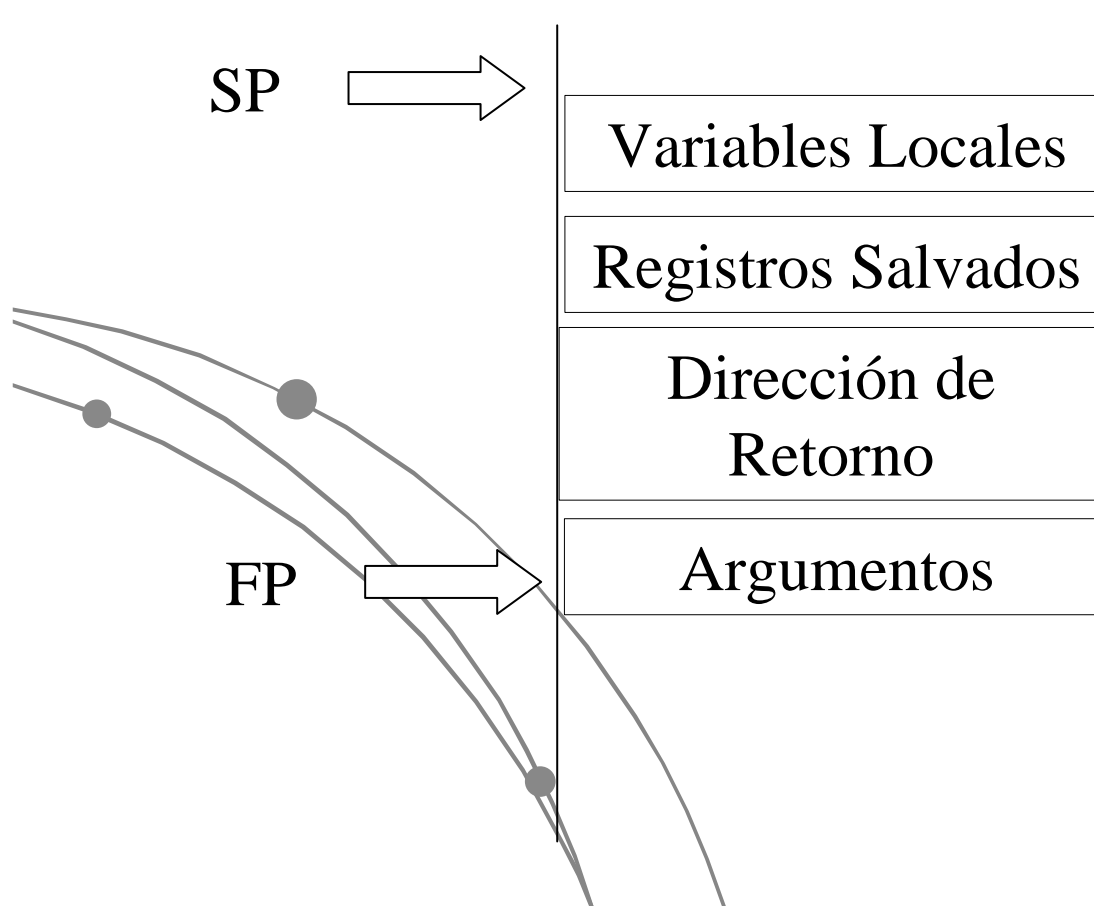
**POP R2**

**RET**

En general, los registros que usan las rutinas deben ser salvados previamente en el Stack

# Bloque de Activación

- Contiene toda la información necesaria para la ejecución de la subrutina
- Generalmente armado por el compilador de un lenguaje de alto nivel



El puntero FP, a diferencia de SP, se mantiene fijo durante toda la ejecución del procedimiento, brindando un registro base estable para acceder a parámetros y variables.

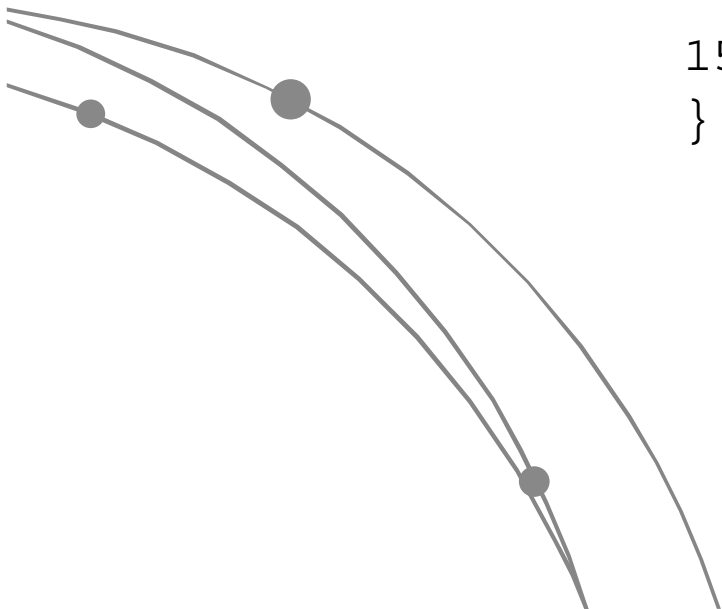


# Ejemplo

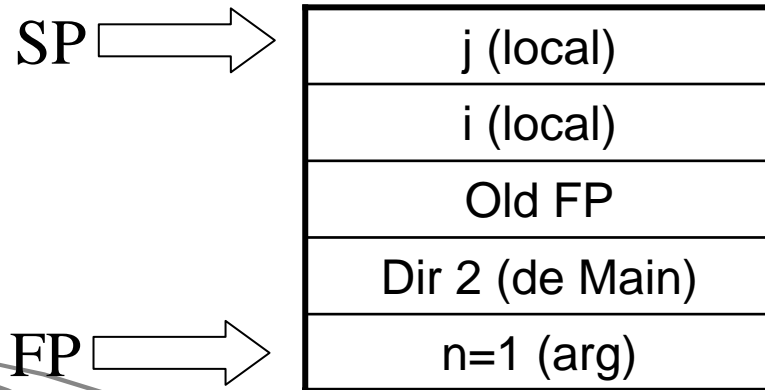
```
void main()  
{  
1: P1(1)  
2: return  
}
```

```
void P1(int n)  
{  
    int i,j;  
    i = 5;  
    ...  
11: P2(10,j);  
12: j=i+1;  
    ...  
15: return;  
}
```

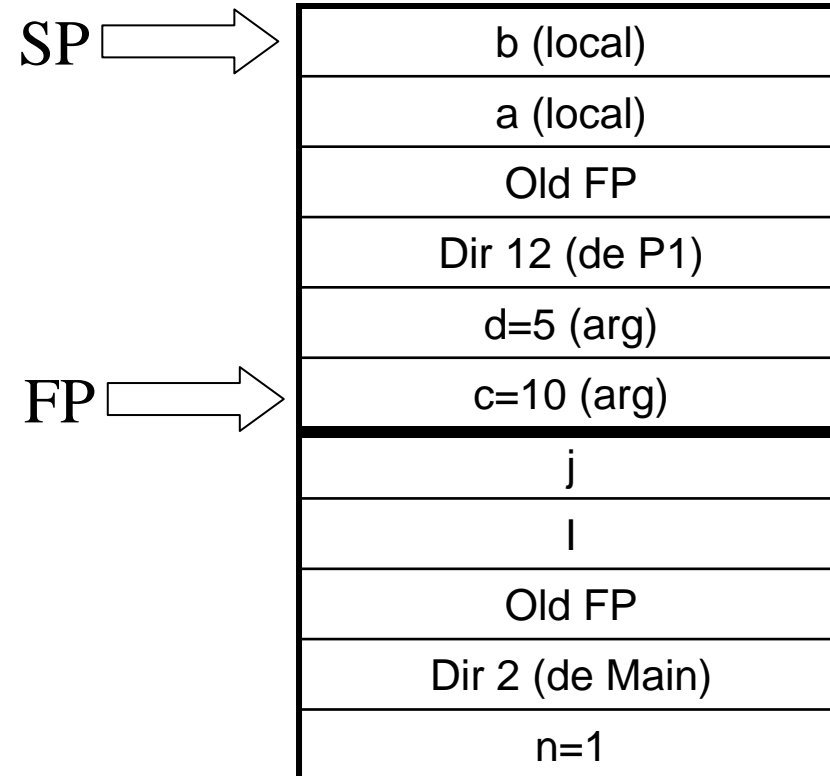
```
void P2(int c,d)  
{  
    int a,b;  
    ...  
21: return;  
}
```



Cuando main llama a P1



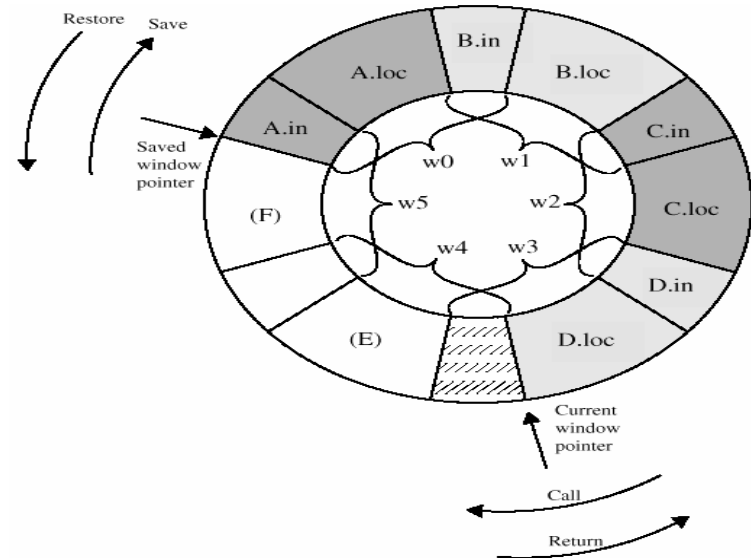
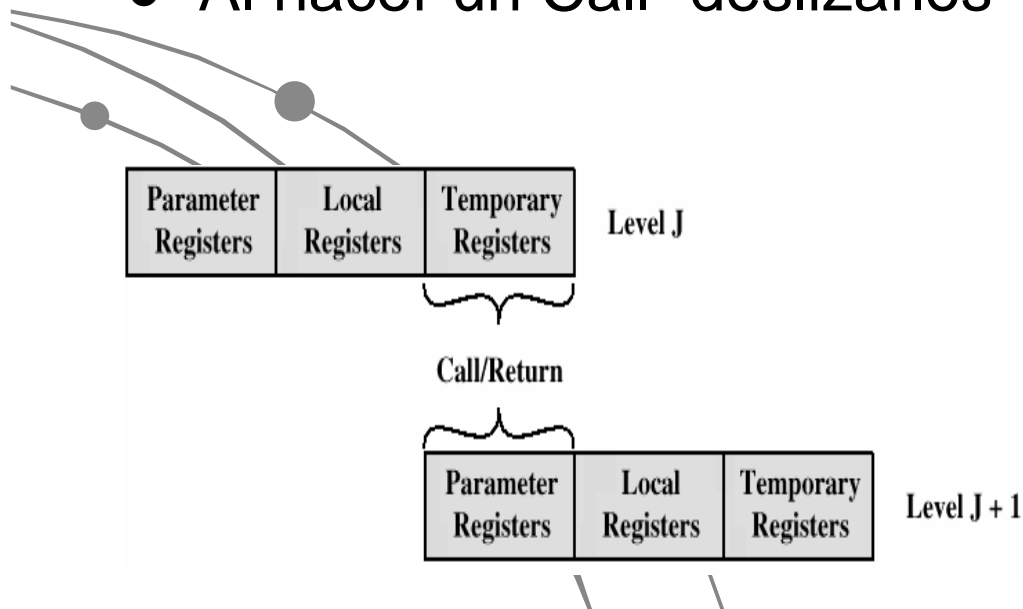
Cuando P1 llama a P2



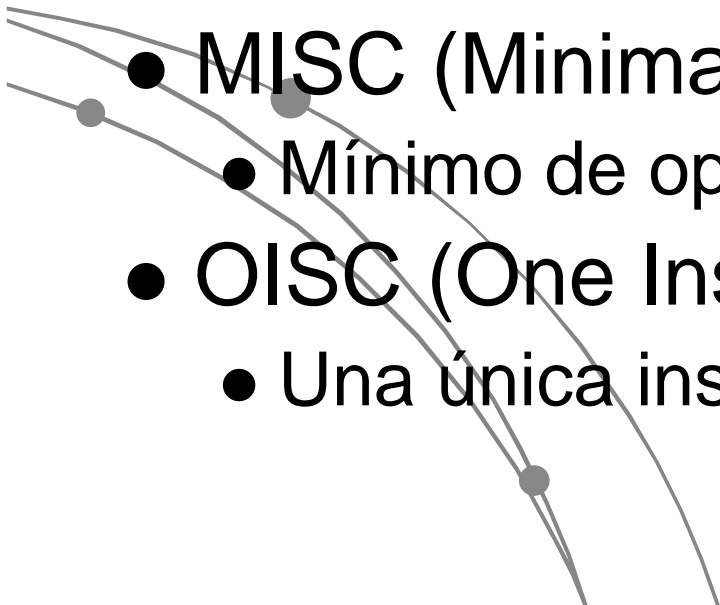
1. El procedimiento llamador se encarga de pushear los parámetros en el Stack
2. El call pushea el PC en el Stack
3. Antes de salir el llamado se ocupa de limpiar del Stack las variables locales y el FP
4. El return asigna al PC=la dirección que esta al tope del stack y la elimina
5. El llamador libera los parámetros

# Subrutinas (Ventana de registros)

- Evitar el uso del Stack
- Idea: tener muchos registros, pero permitir usar unos pocos
- Asignar registros a parámetros, argumentos y variables locales
- Al hacer un Call “deslizarlos”



# Enfoques

- CISC (Complex Instruction Set Computer)
    - Instrucciones que realizan tareas complejas
  - RISC (Reduced Instruction Set Computer)
    - Instrucciones que realizan operaciones sencillas
  - MISC (Minimal Instruction Set Computer)
    - Mínimo de operaciones necesarias
  - OISC (One Instruction Set Computer)
    - Una única instrucción
- 

# Ejemplo CISC (Intel)

Prefijo	OpCode
0xF3	0xA4

Instrucción: **REP MOVSB**

Copia CX bytes de DS:SI, a ES:DI.

MOVSB: Copia el dato en DS:SI, a ES:DI.

Dependiendo de un flag, SI y DI son incrementados (+1) o decrementados (-1)

REP decrementa CX y hace que se repita la operación hasta que CX llegue a 0

# Ejemplo RISC

La misma instrucción implementada en una MIPS:

Asumamos que en \$s3 esta el fuente, \$s4 el destino y \$s5 es el contador

bucle:

lb \$t0, 0(\$s3) ; t0 = mem[s3]

sb \$t0, 0(\$s4) ; mem[s4] = t0

add \$s3,\$s3,1

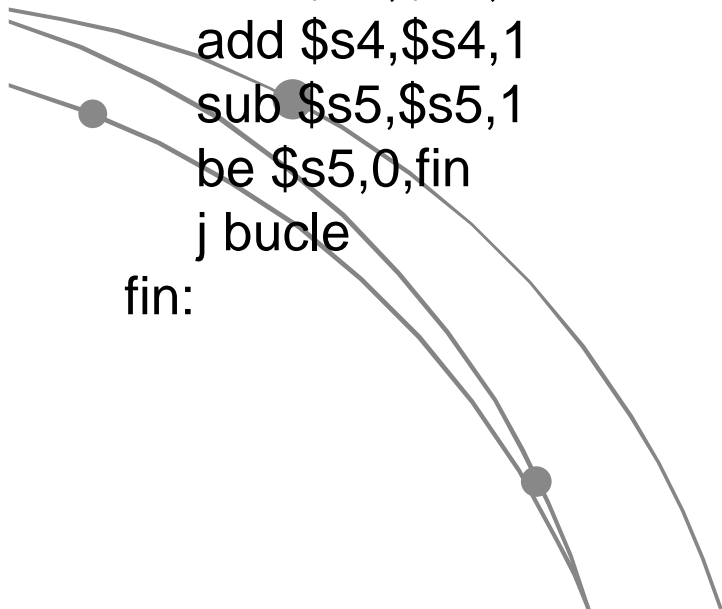
add \$s4,\$s4,1

sub \$s5,\$s5,1

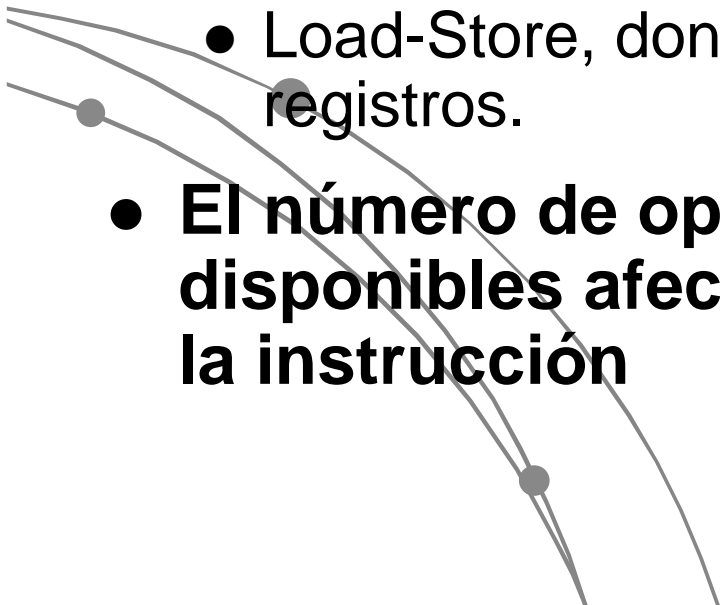
be \$s5,0,fin

j bucle

fin:



# ISAs

- La mayoría de los sistemas actuales son **GPR (general purpose registers)**.
  - 3 tipos de sistemas:
    - Memoria-Memoria donde 2 o 3 operandos pueden estar en memoria.
    - Registro-Memoria donde al menos un operando es un registro
    - Load-Store, donde las operaciones son sólo entre registros.
  - **El número de operandos y de registros disponibles afectan directamente el tamaño de la instrucción**
- 

# Referencias

- Capítulo 5 – Tanenbaum
- Capitulo 4 y 5 – Null
- Capitulo 10 – Stallings
- <http://www.cs.uiowa.edu/~jones/arch/risc/>
  - Artículo sobre OISC

