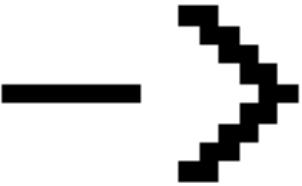


Organización del computador

ISA - Instruction Set Architecture

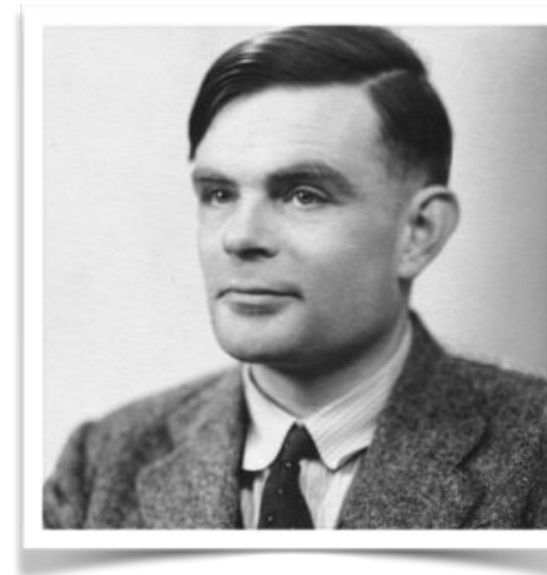
Jerarquía de máquina



Nivel 6	Usuario	Programa ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assembly code
Nivel 3	Software del sistema	Sistema operativo, bibliotecas, etc.
Nivel 2	Lenguaje de máquina	Instruction Set Architecture (ISA)
Nivel 1	Unidad de control	Microcódigo / hardware
Nivel 0	Lógica digital	Circuitos, compuertas, memorias

- ➤ Cada nivel funciona como una máquina abstracta que oculta la capa anterior
- ➤ Cada nivel es capaz de resolver determinado tipo de problemas a partir de comprender un tipo de instrucciones específico
- ➤ La capa inferior es utilizada como servicio

Von Newman / Turing



- * Los programas y los datos se almacenan en la misma memoria sobre la que se puede leer y escribir
- * La operación de la máquina depende del estado de la memoria
- * El contenido de la memoria es accedido a partir de su posición
- * La ejecución es secuencial (a menos que se indique lo contrario)

Arquitectura de von Neumann

— ➤ 3 componentes principales:

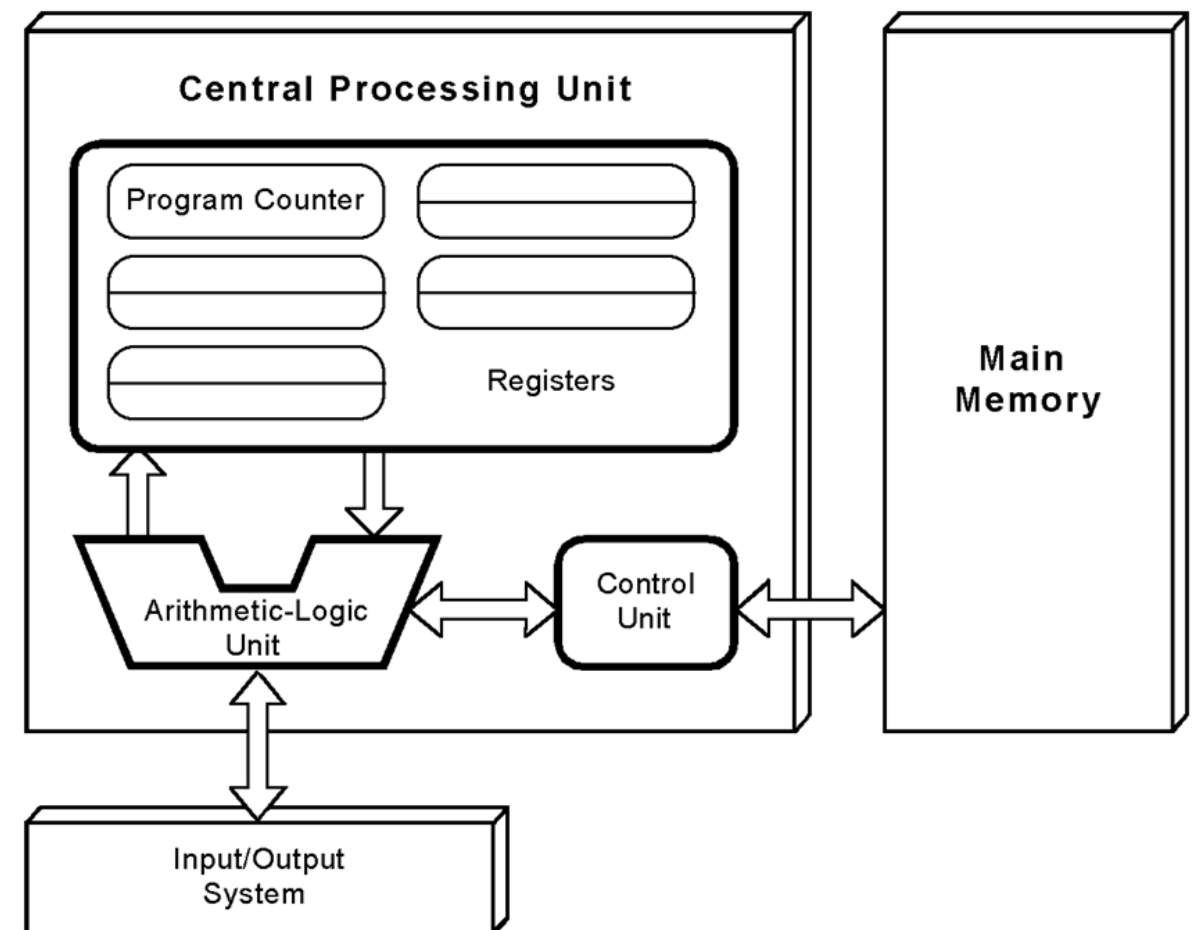
- **CPU:** Unidad de control, Unidad Aritmética lógica, Registros
- **Memoria:** Almacenamiento de programas y datos
- **Sistema** de Entrada y Salida

— ➤ Procesamiento secuencial de instrucciones

— ➤ Datos almacenados en sistema binario

— ➤ Sistema de interconexión de componentes:

- Conecta la Unidad de Control con la Memoria mediante un camino único
- La unicidad del camino fuerza la alternación entre ciclos de lectura / escritura y ejecución
- Esta alternación se llama cuello de botella de von Neumann (von Neumann bottleneck)^[*]



[*] El término “cuello de botella de von Neumann” fue acuñado por John Backus en su conferencia de la concesión del Premio Turing ACM de 1977.

Arquitectura de von Neumann

—> Ciclo de instrucción (UC):

Fetch

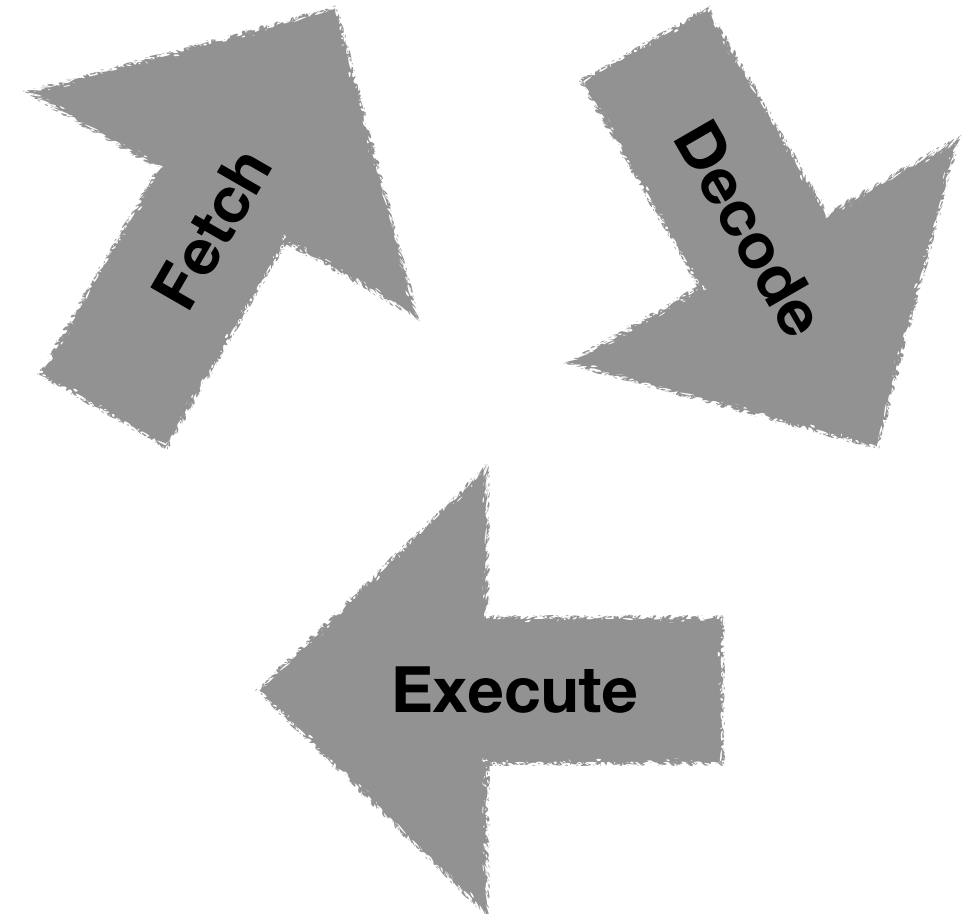
- Se obtiene la instrucción apuntada por el **PC** de la **Memoria**
- La **ALU** incrementa el **PC**

Decode

- Se decodifica la instrucción
- Se obtienen los operandos de la **Memoria** y se los coloca en **Registros**

Execute

- La **ALU** realiza la operación
- Se coloca el resultado en la **Memoria**



Instruction Set Architecture

- ➤ La **ISA** es el límite entre el software y el hardware
- ➤ Está íntimamente relacionada con la organización del sistema pues se apoya sobre su implementación en términos de componentes electrónicas
- ➤ Define la forma en la cual un programador observa la arquitectura del sistema

Instruction Set Architecture

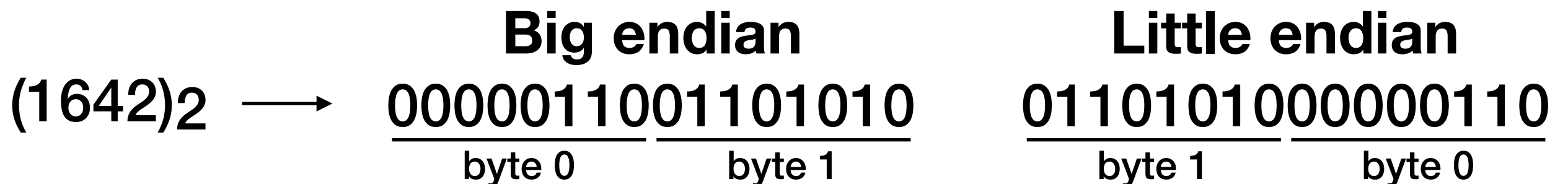
Propiedades de una ISA

- ➤ **Complejidad del conjunto de instrucciones:** CISC vs RISC (Complex / Reduced Instruction Set Computer)
- ➤ **Longitud de las instrucciones:** fija o variable
- ➤ **Cantidad de memoria utilizada por un programa**
- ➤ **Datos:** tipos de datos disponibles y representación de cada tipo de datos, distribución de los nibbles según big o little endian; por ejemplo, enteros en complemento a 2, punto flotante de precisión simple, caracteres en ISO-8859-1, big endian

Instruction Set Architecture

Big endian vs Little endian

- > La convención “endian” refiere a la forma en la que se organizan los datos que requieren de más de un byte para ser almacenados
 - > **Little endian:** el byte conteniendo los bits menos significativos se alojan en la posición de memoria menor
 - > **Big endian:** el byte conteniendo los bits menos significativos se alojan en la posición de memoria mayor
- > **Importante:** No se invierte la representación bit a bit, solo el orden de los bytes que conforman la representación



Instruction Set Architecture

Tipos de arquitectura

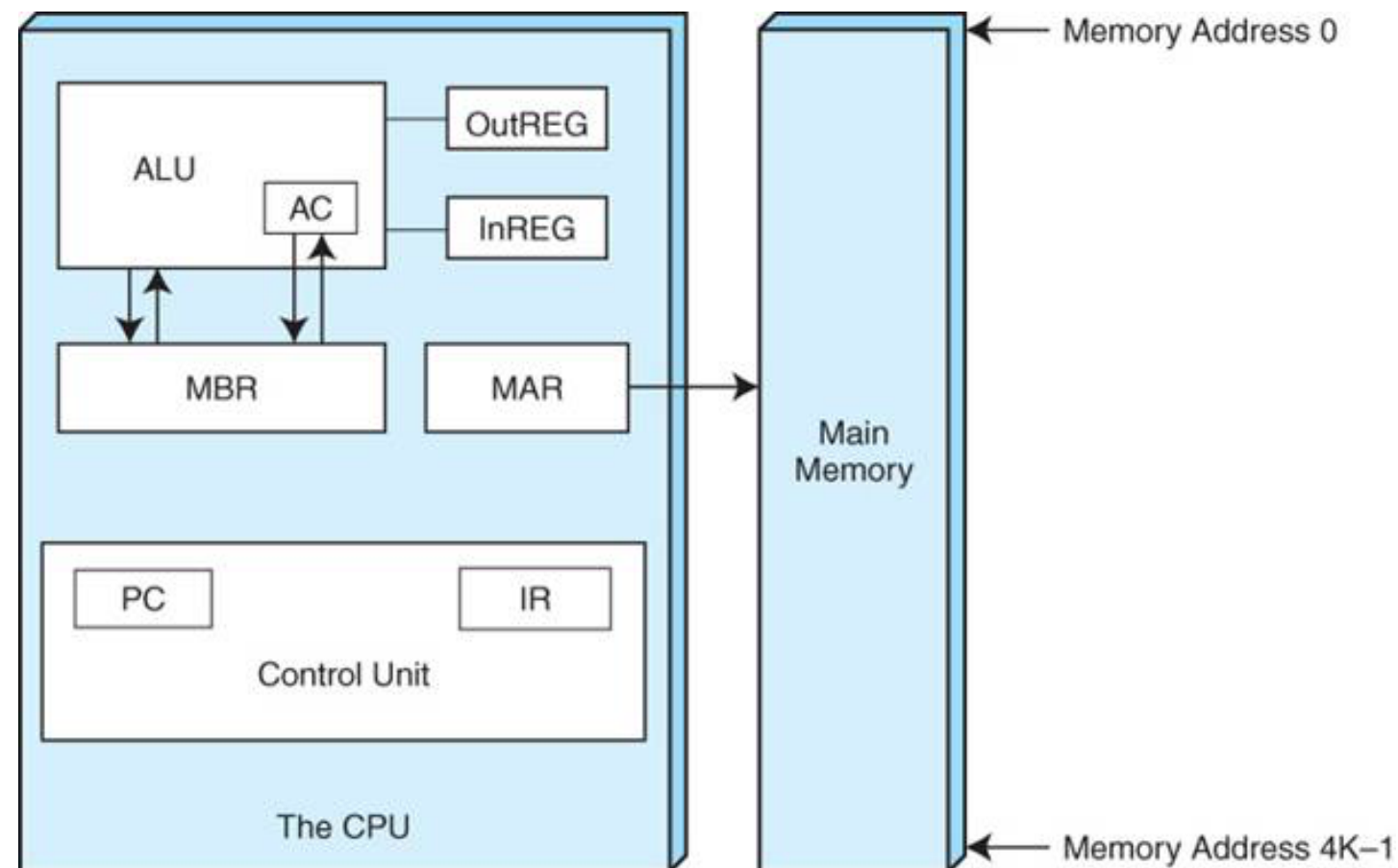
- ➤ **Stack architecture:** se cuenta con una pila y las operaciones se realizan sobre los elementos almacenados en ella accesibles desde el tope
- ➤ **Accumulator:** se cuenta con un registro distinguido (algunas veces dos considerando el multiplier quotient) y las operaciones tienen como operando implícito dicho registro; de ser necesario otro operando se proporciona su dirección en la memoria y el resultado se coloca en el acumulador
- ➤ **General purpose registers:** se cuenta con un banco de registros y las operaciones se realizan entre ellos así que todos los operandos, y el destino del resultado, son explícitos

Instruction Set Architecture

Accumulator (MARIE)

(Machine Architecture that is Really Intuitive and Easy)

- **Representación:** binaria en complemento a 2
- **Memoria:** 4K, accedida por palabras
- **Palabra:** 16 bits
- **Instrucciones:** tamaño fijo 16 bits (4 para el código de operación y 12 para las direcciones)
- **ALU:** 16 bits
- **Registros:** 7 para control y movimiento de datos



Instruction Set Architecture

Accumulator (MARIE)

OpCode	Instrucción	Efecto
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
1001	Jump Dir	$PC = Dir$
1010	Clear	$AC = 0$
1011	Addi X	$AC = AC + [[X]]$
1100	Jumpi X	$PC = [X]$

Instruction Set Architecture

General purpose registers (Orga1)

- ➤ **Representación:** binaria en complemento a 2
- ➤ **Memoria:** 64K, accedida por palabras (16 bits de direcciones)
- ➤ **Palabra:** 16 bits
- ➤ **Instrucciones:** tamaño variable 16/32/48 bits dependiendo de si es de 0, 1 ó 2 operandos (4 para el código de operación y 12 para operando 1, 16 para operando 2 y 16 para operando 3)
- ➤ **ALU:** 16 bits
- ➤ **Registros:** 8 para uso general (R0 — R7), 3 para control (PC-program counter, SP-stack pointer, Flags)

Instruction Set Architecture

General purpose registers (Orga1)

Tipo 1: Instrucciones de dos operandos

<i>4 bits</i>	<i>6 bits</i>	<i>6 bits</i>	<i>16 bits</i>	<i>16 bits</i>
cod. op.	destino	fuelle	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.

Instruction Set Architecture

General purpose registers (Orga1)

Tipo 2: Instrucciones de un 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)

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

El formato del operando *destino* responde a la tabla de formatos de operando mostrada más arriba.

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	fuelle	constante fuente (opcional)

operación	cod. op.	efecto
JMP f	1010	$\text{PC} \leftarrow f$
CALL f	1011	$[\text{SP}] \leftarrow \text{PC}, \text{SP} \leftarrow \text{SP} - 1, \text{PC} \leftarrow f$

Instruction Set Architecture

General purpose registers (Orga1)

Tipo 3: Instrucciones sin operandos

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

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

Instruction Set Architecture

General purpose registers (Orga1)

Tipo 4: Saltos condicionales

Las instrucciones en este formato son de la forma *Jxx* (salto relativo condicional). Si al evaluar la condición de salto en los *flags* el resultado es *1*, el efecto es incrementar el PC con el valor de los *8 bits* de desplazamiento, representado en *complemento a 2* de *8 bits*. En caso contrario, la instrucción no produce efectos.

<i>8 bits</i>	<i>8 bits</i>
cod. op.	desplazamiento

Codop	Operación	Descripción	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

Instruction Set Architecture

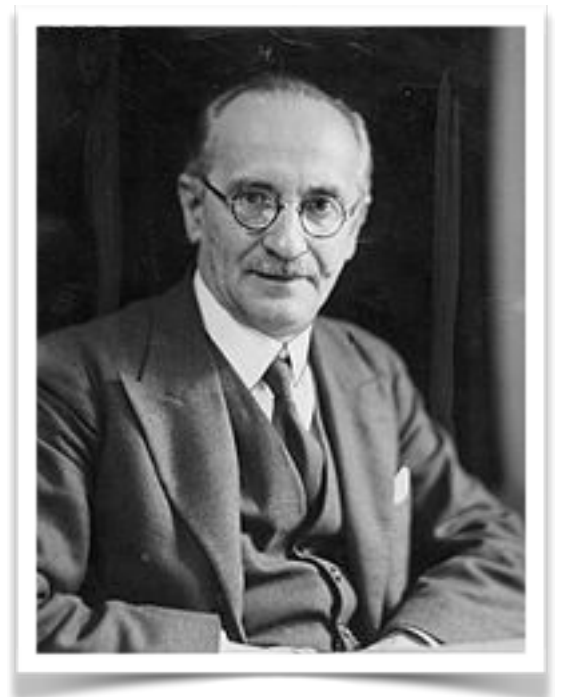
Stack architecture

- ➤ **Jan Łukasiewicz**: Matemático y filósofo polaco, 1878 — 1956. Entre sus grandes contribuciones se cuenta la axiomatización compacta de la lógica proposicional, la lógica trivaluada y la notación polaca inversa
- ➤ La **notación polaca inversa** u **orden posfijo** indica la operación a realizarse detrás de los operandos:

$$X + Y \rightarrow X Y +$$

- ➤ No requiere paréntesis pues no es ambigua

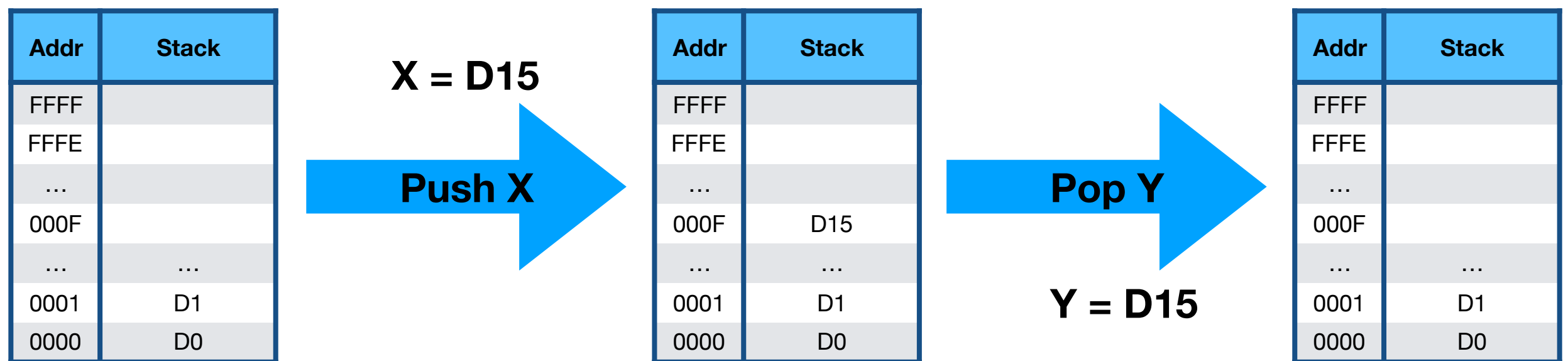
$$X * (Y + Z) \rightarrow X Y Z + *$$



Instruction Set Architecture

Stack architecture

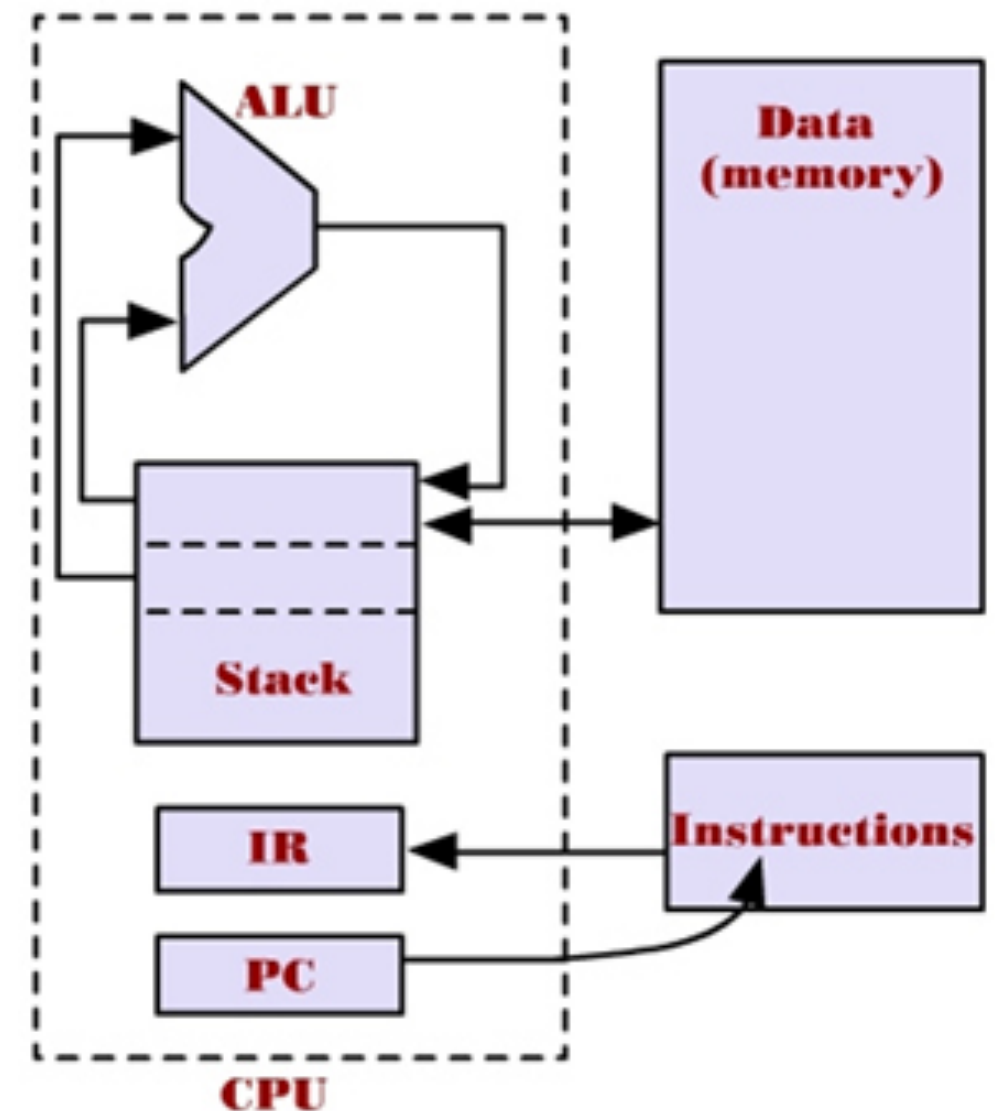
- ➤ Un **stack** o **pila** posee dos operaciones: **Push** que permite colocar un dato en la primera posición libre y **Pop** que permite retirar el último dato que se encuentra en la pila.



Instruction Set Architecture

Stack architecture

- ➤ **Push / Pop:** requieren una dirección de memoria como operando
- ➤ **Add / Mult, LE / GE/ Eq:** realizan la operación con los dos elementos en el tope del stack
- ➤ **JMPT / JMPF:** (jump true/false) requieren una dirección de memoria como operando



Instruction Set Architecture

Asignaciones

- ➤ Permiten depositar valores en posiciones de memoria destinadas a tal fin, tal como en los lenguajes de alto nivel.
- ➤ Las operaciones aritméticas son realizadas utilizando las directivas provistas por la ISA

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

Accumulator	Orga1	Stack
LOAD X	MOV R1, X	PUSH W
ADD Y	ADD R1, Y	PUSH X
STORE TEMP	MOV R2, W	PUSH Y
LOAD W	SUBT R1, R2	ADD
Subt Temp	MOV Z, R1	SUBT
STORE Z		POP Z

Instruction Set Architecture

Control

- ➤ Permiten modificar el flujo natural de la ejecución de un programa decidiendo el lugar a partir del cual se continuará
- ➤ Sirven para implementar If-Then-Else, While, For, etc.

```
if (X < 1) then  
    X := X + 1;  
else  
    Y := Y + 1;
```

Instruction Set Architecture

Control (If-Then-Else)

Accumulator		
If,	100 Load X	; AC := [X]
	102 Skipcond 600	; Si AC <= 0 (X<1): Then
	103 Jump Else	; Caso contrario: Else
Then,	104 Load X	
	105 Add One	
	106 Store X	; X := X + 1
	107 Jump Endif	; Saltea el Else
Else,	108 Load Y	
	109 Add One	
	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

Instruction Set Architecture

Control (If-Then-Else)

Orga1				
If	100	Mov R1, [X]		
	101	CMP R1, 0		
	102	JLE Else		
Then,	103	Add R1, 1		
	104	Mov [X], R1	; X:= X + 1	
	105	Jump Endif	; Saltea ELSE	
Else,	106	Mov R1, [Y]		
	107	Add R1, 1		
	108	Mov [Y], R1	; Y := Y + 1	
Endif,	109	Ret	; Terminar	
X,	10A	DEC ?	; Espacio para X	
Y,	10B	DEC ?	; Espacio para Y	

Instruction Set Architecture

Control (If-Then-Else)

Stack			
If,	100	Push X	
	101	Push 0	
	102	LE	; Si $X \leq 0$: apila 1
	103	JMPT Else	; Si 1: Else. Desapila X y 1
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	Pop Y	; $Y := Y + 1$
Endif,	10D	Halt	; Terminar
X,	10E	DEC ?	; Espacio para X
Y,	10F	DEC ?	; Espacio para Y

Instruction Set Architecture

Control

- ➤ Permiten modificar el flujo natural de la ejecución de un programa decidiendo el lugar a partir del cual se continuará
- ➤ Sirven para implementar If-Then-Else, While, For, etc.

```
if (X < 1) then  
    X := X + 1;  
else  
    Y := Y + 1;
```

Instruction Set Architecture

Control

- ➤ Permiten modificar el flujo natural de la ejecución de un programa decidiendo el lugar a partir del cual se continuará
- ➤ Sirven para implementar If-Then-Else, While, For, etc.

```
int A[5] = {10,15,20,25,30}  
int Sum = 0;  
I := 0;  
while(i < 5){  
    Sum := Sum + A[i];  
    i++;  
}
```

Instruction Set Architecture

Control (While)

Accumulator		
Start,	Load Num	; Cargo el Contador
	SkipCond 800	; Si AC <= 0: Fin
	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	; .
	Dec 20	; .
	Dec 25	; .
	Dec 30	; Num5 = 30
PostAct,	Hex 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	

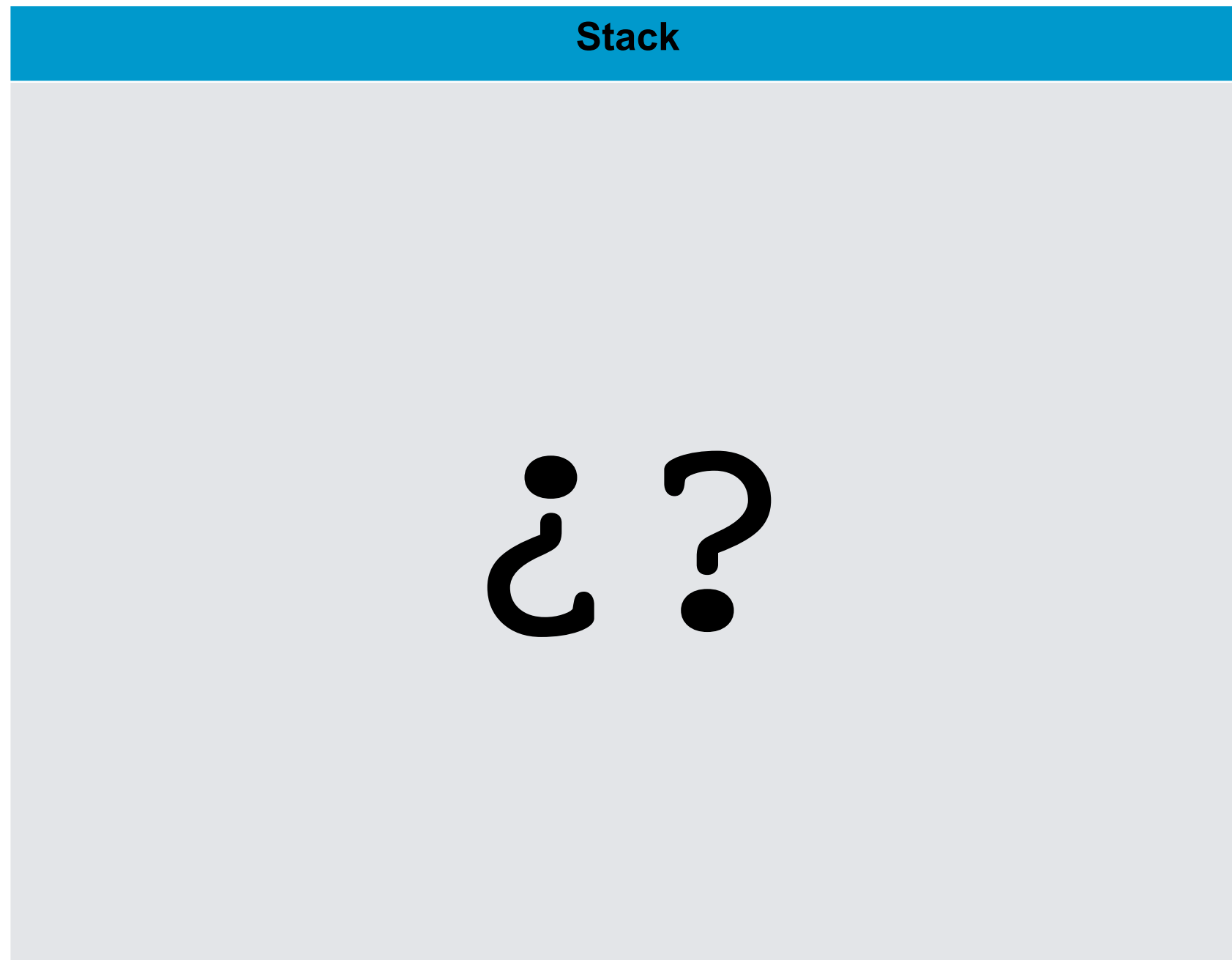
Instruction Set Architecture

Control (While)

Orga1		
Vec	DW 10	; Num1 = 10
	DW 15	; .
	DW 20	; .
	DW 25	; .
	DW 30	; Num5 = 30
Num	DW 5	; Contador para el loop = 5
Sum	DW 0	; Suma = 0
	MOV R2, Vec	; Apunta al Vector
	MOV R1, 0	; Acumulador
	MOV R3, R2	; Inicializa el offset ()
	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

Instruction Set Architecture

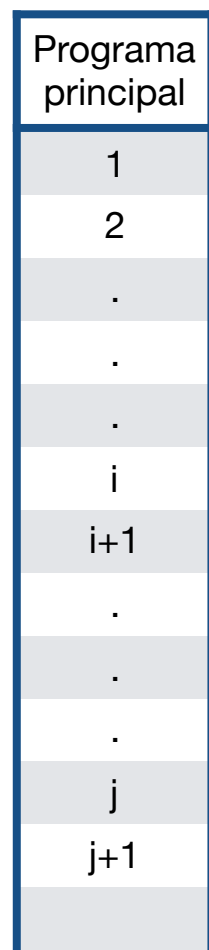
Control (While)



Instruction Set Architecture

Subrutinas

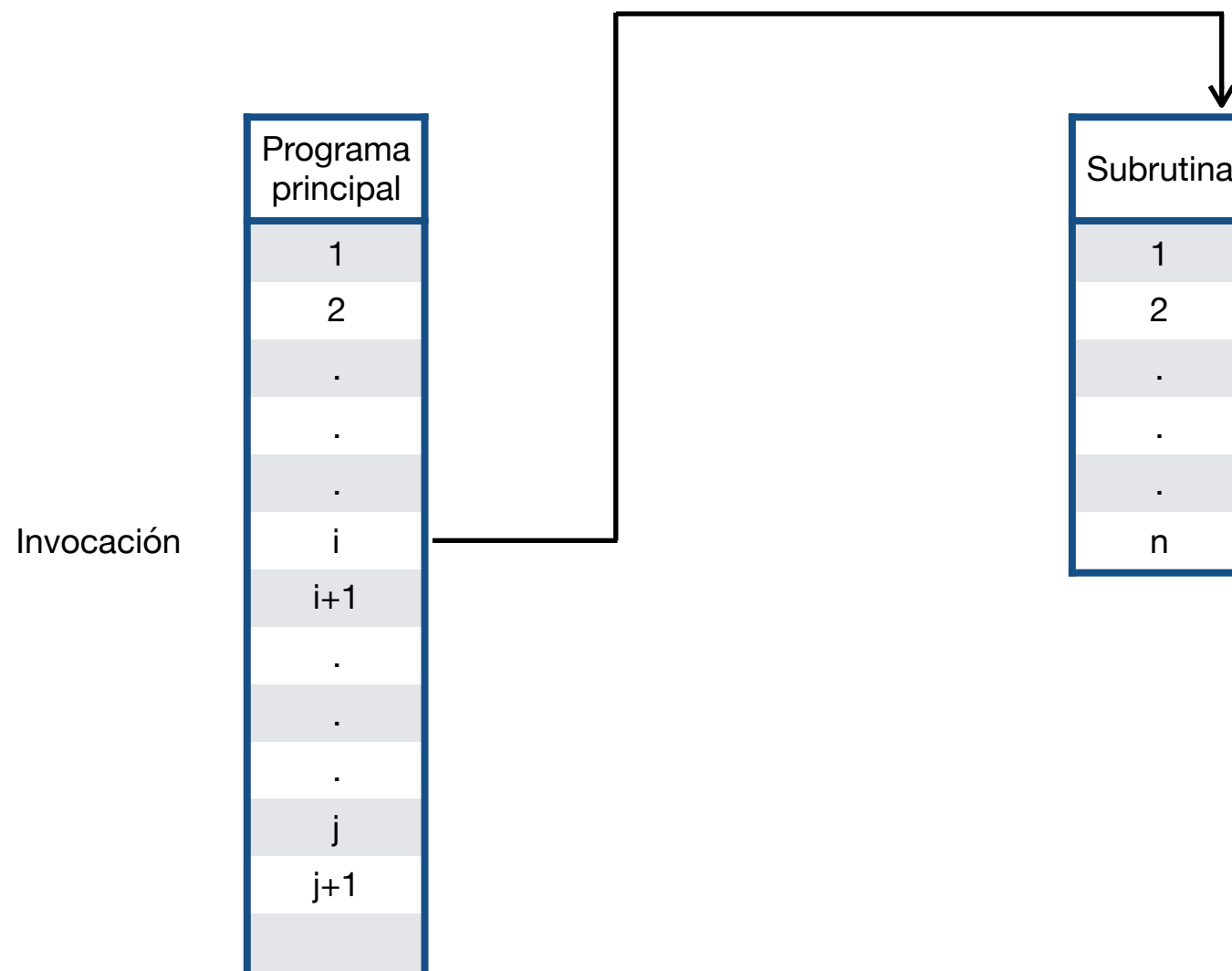
- > Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó



Instruction Set Architecture

Subrutinas

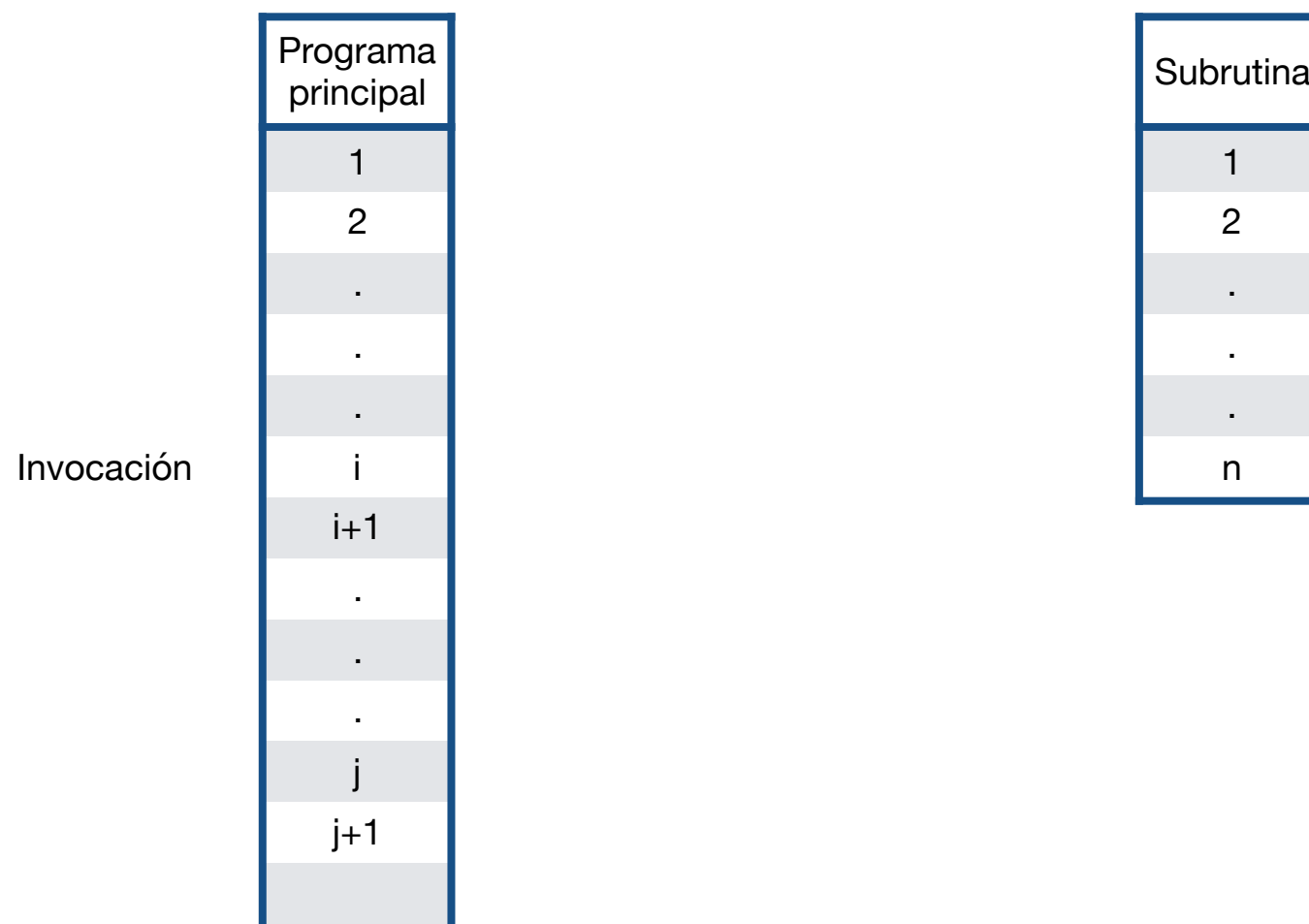
- > Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó



Instruction Set Architecture

Subrutinas

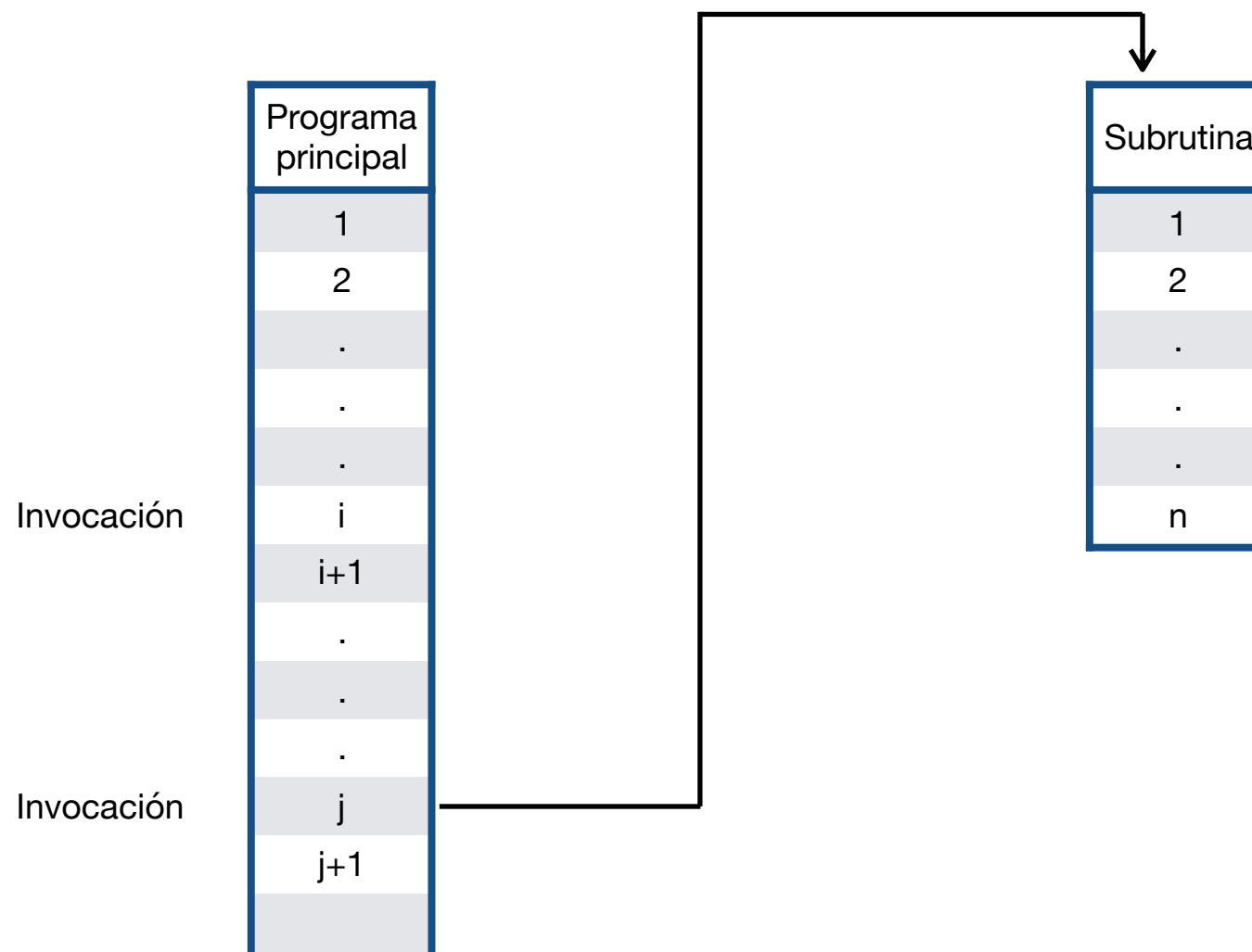
- ➤ Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó



Instruction Set Architecture

Subrutinas

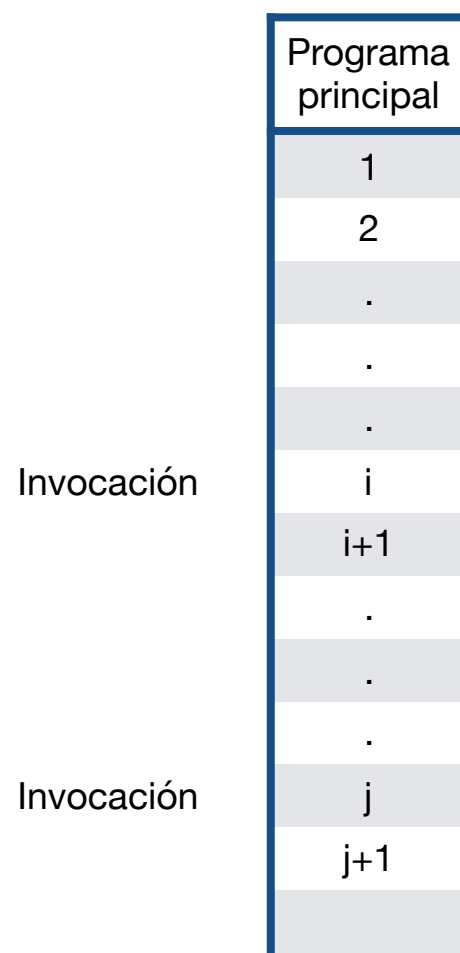
- > Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó



Instruction Set Architecture

Subrutinas

- > Posibilitan la reutilización de código a partir de modularizar fragmentos que son invocados, ejecutados para luego retornar a la ejecución del programa que lo invocó



Instruction Set Architecture

Subrutinas

- ➤ Los sistemas operativos implementan un **stack** o **pila** que posibilita la invocación de **subrutinas** preservando la información del programa que la invocó de forma que la ejecución pueda retornar a la instrucción siguiente a dicha invocación

Addr	Stack
FFFF	
FFFE	
...	...
000F	D15
...	...
0001	D1
0000	D0

- ➤ La instrucción “Call Subr” implícitamente realiza “Push PC”, preservando el punto de retorno a la ejecución del programa invocador y “PC = Subr” indicando que se debe continuar la ejecución dirección de comienzo de la subrutina, “Subr”
- ➤ La instrucción “Ret” implícitamente realiza “Pop PC” restaurando el punto a partir del cual se debe continuar la ejecución de programa invocado

Instruction Set Architecture

Subrutinas

```
[0100] := [0100] + 5  
[0200] := [0200] + 5  
[0204] := [0204] + 5
```

Programa principal

```
MOV R1, 0100  
CALL Sum5  
MOV R1, 0200  
CALL Sum5  
MOV R1, 0204  
CALL Sum5
```

Subrutina

```
Sum5: MOV R2, [R1]  
      ADD R2, 5  
      MOV [R1], R2  
      RET
```

Instruction Set Architecture

Subrutinas

```
[0100] := [0100] + 5  
[0200] := [0200] + 5  
[0204] := [0204] + 5
```

Programa principal

```
MOV R2, CTE  
MOV R1, 0100  
CALL Sum5  
MOV R1, 0200  
CALL Sum5  
MOV R1, 0204  
CALL Sum5
```

Subrutina

```
Sum5:  MOV R2, [R1]  
        ADD R2, 5  
        MOV [R1], R2  
        RET
```

¿Qué ocurre si R2 estaba siendo usado por el programa principal?

Instruction Set Architecture

Subrutinas

```
[0100] := [0100] + 5  
[0200] := [0200] + 5  
[0204] := [0204] + 5
```

Programa principal

```
MOV R2, CTE  
MOV R1, 0100  
CALL Sum5  
MOV R1, 0200  
CALL Sum5  
MOV R1, 0204  
CALL Sum5
```

Subrutina

```
Sum5: Push R2  
      MOV R2, [R1]  
      ADD R2, 5  
      MOV [R1], R2  
      Pop R2  
      RET
```

¿Qué ocurre si R2 estaba siendo usado por el programa principal?

Instruction Set Architecture

Subrutinas

```
[0100] := [0100] + 5  
[0200] := [0200] + 5  
[0204] := [0204] + 5
```

La subrutina es responsable de preservar los valores de todo los registros que utilice en el **stack** y de restaurarlos antes de retornar al programa invocador.

Programa principal

```
MOV R2, CTE  
MOV R1, 0100  
CALL Sum5  
MOV R1, 0200  
CALL Sum5  
MOV R1, 0204  
CALL Sum5
```

Subrutina

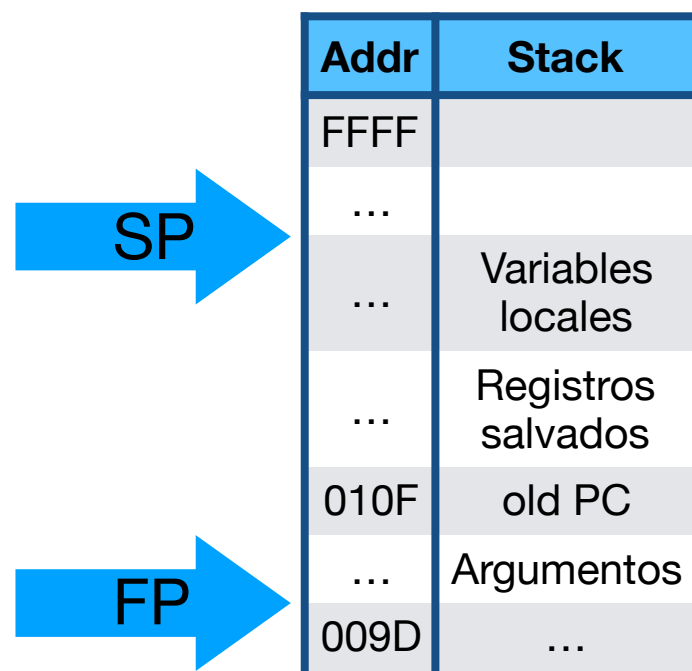
```
Sum5: Push R2  
      MOV R2, [R1]  
      ADD R2, 5  
      MOV [R1], R2  
      Pop R2  
      RET
```

¿Qué ocurre si R2 estaba siendo usado por el programa principal?

Instruction Set Architecture

Subrutinas

- ➤ Cuando las subrutinas reciben parámetros de los programas invocadores estos son pasados utilizando el **Stack** del sistema
- ➤ Esto introduce el concepto de **bloque de activación** de un código en ejecución

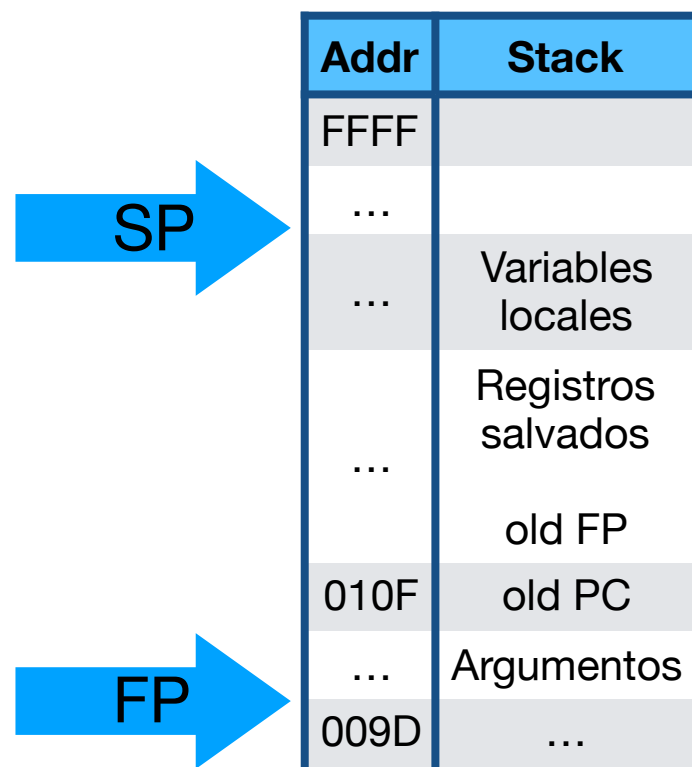


- ➤ FP y SP son registros utilizados para apuntar a los límites del bloque de activación de la rutina que se está ejecutando en este momento
- ➤ Esto obliga a la rutina a salvar, entre los registros el FP anterior para retornar restaurando el bloque de activación del programa invocador

Instruction Set Architecture

Subrutinas

- ➤ Cuando las subrutinas reciben parámetros de los programas invocadores estos son pasados utilizando el **Stack** del sistema
- ➤ Esto introduce el concepto de **bloque de activación** de un código en ejecución



- ➤ FP y SP son registros utilizados para apuntar a los límites del bloque de activación de la rutina que se está ejecutando en este momento
- ➤ Esto obliga a la rutina a salvar, entre los registros el FP anterior para retornar restaurando el bloque de activación del programa invocador

Instruction Set Architecture

Subrutinas

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

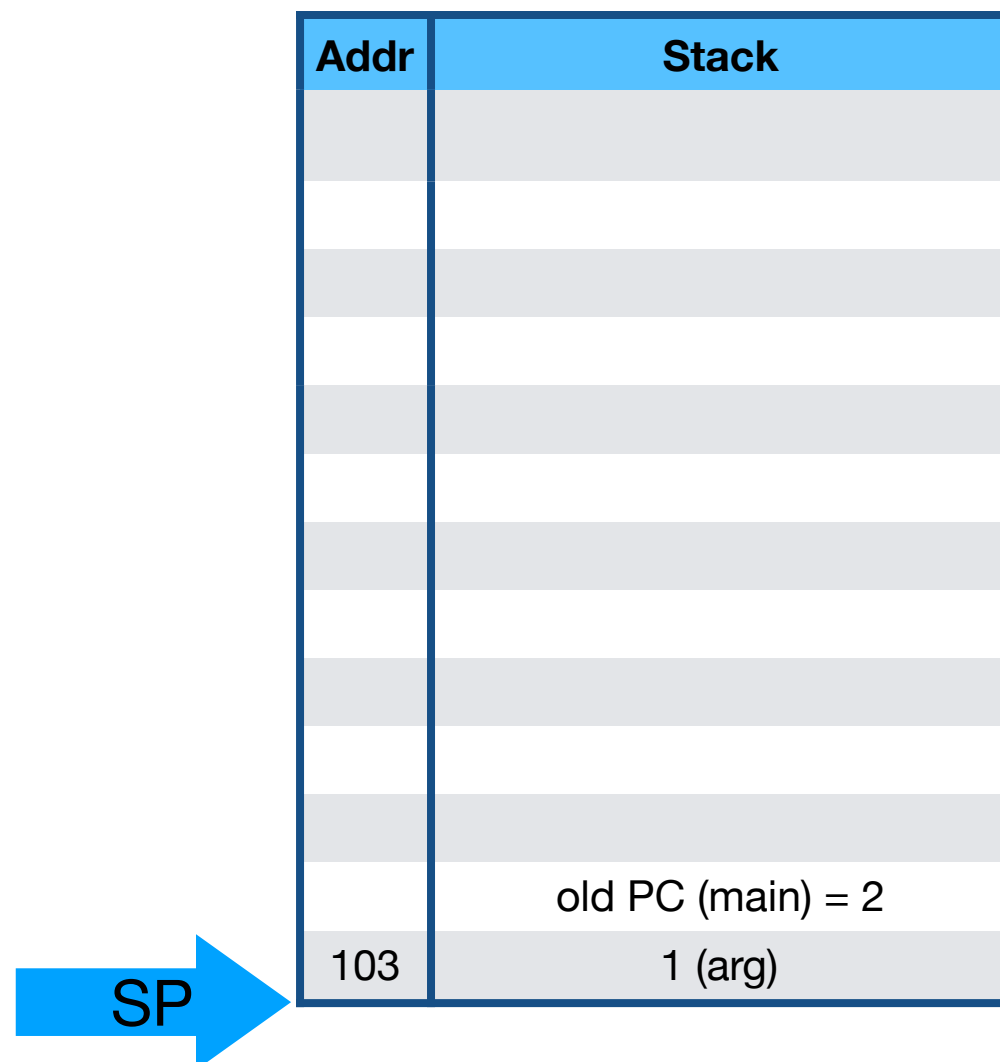
```
void P1 (int n){  
    int i,j;  
    i = 5;  
    int p;  
    ...  
11:  p := P2(10, i);  
12:  j := i + 1;  
    ...  
    return ;  
}
```

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

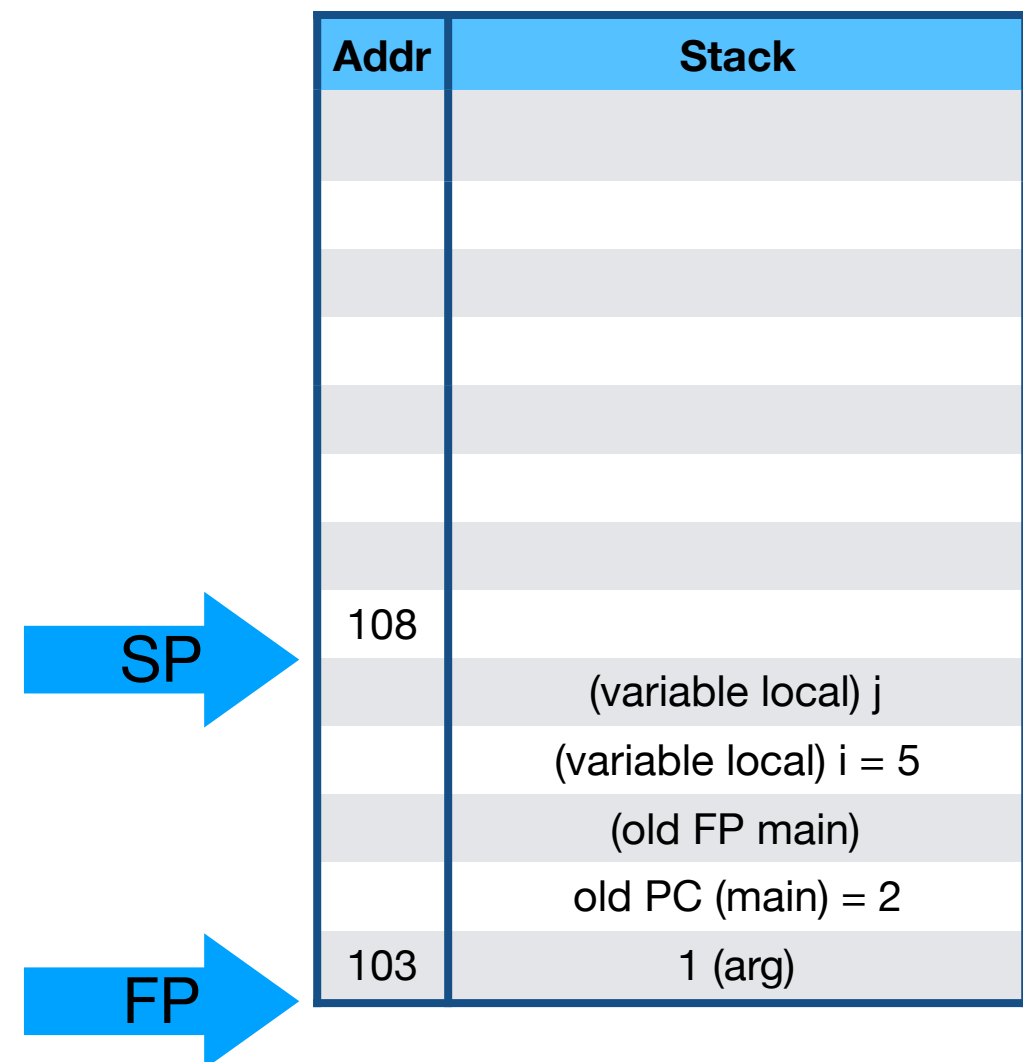
Instruction Set Architecture

Subrutinas

Al momento de Call P1



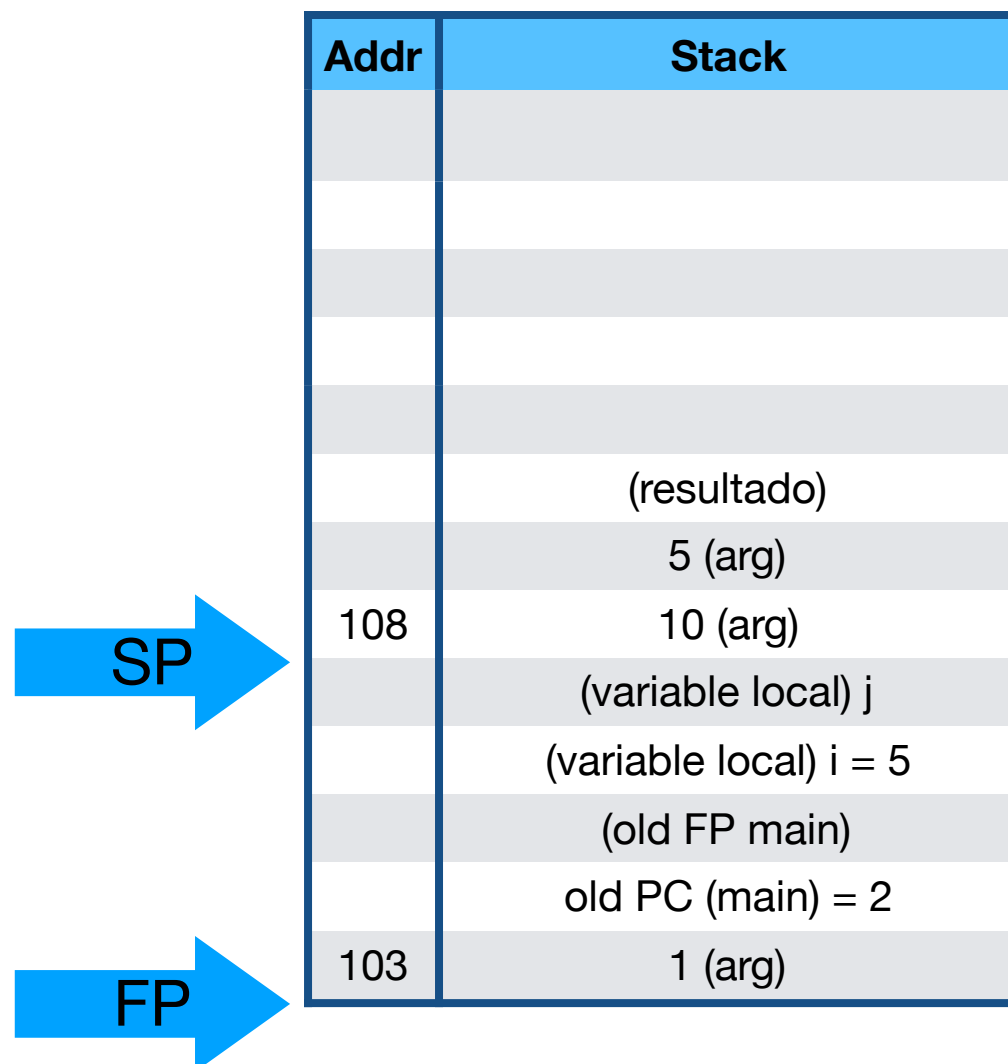
En “11” de la ejecución de P1



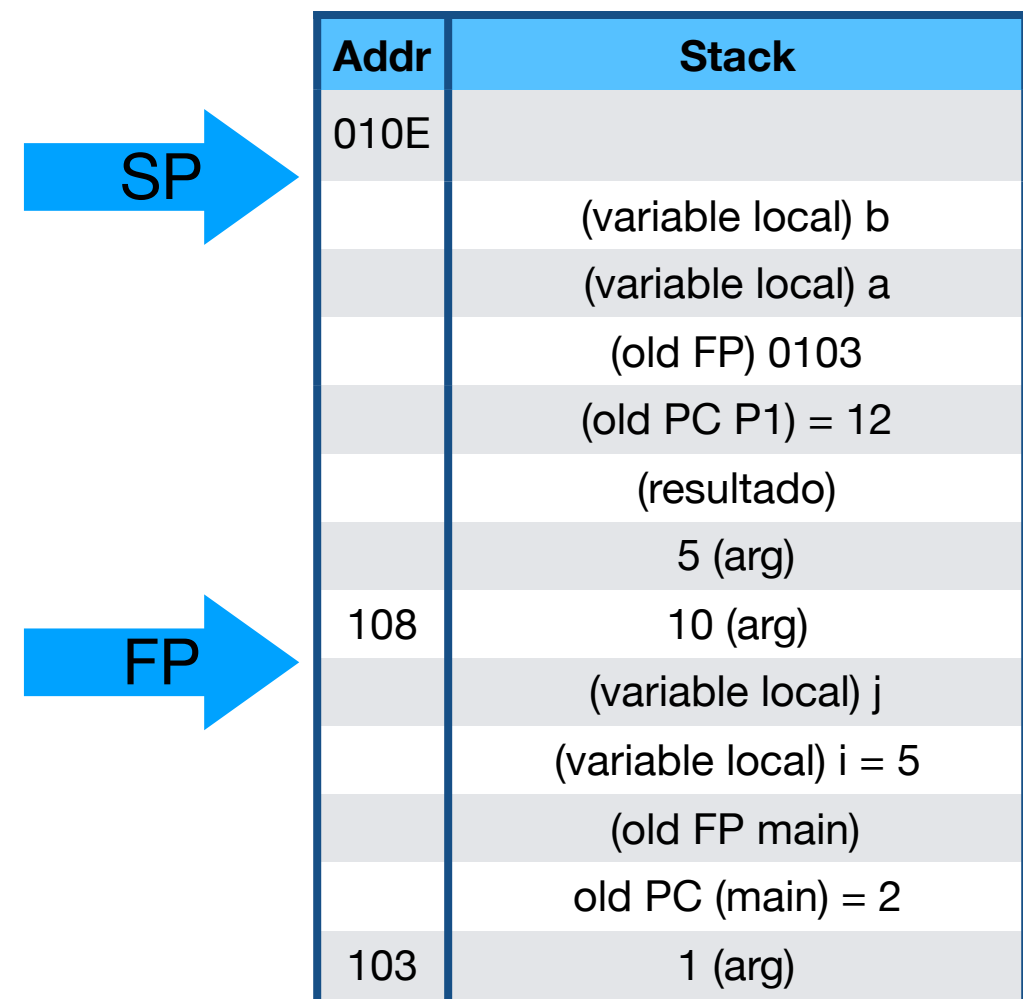
Instruction Set Architecture

Subrutinas

En “11” de la ejecución de P1



En “21” de la ejecución de P2



Addr	Stack
103	1 (arg)

SP →

Instruction Set Architecture

- ➤ **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