

Arquitectura

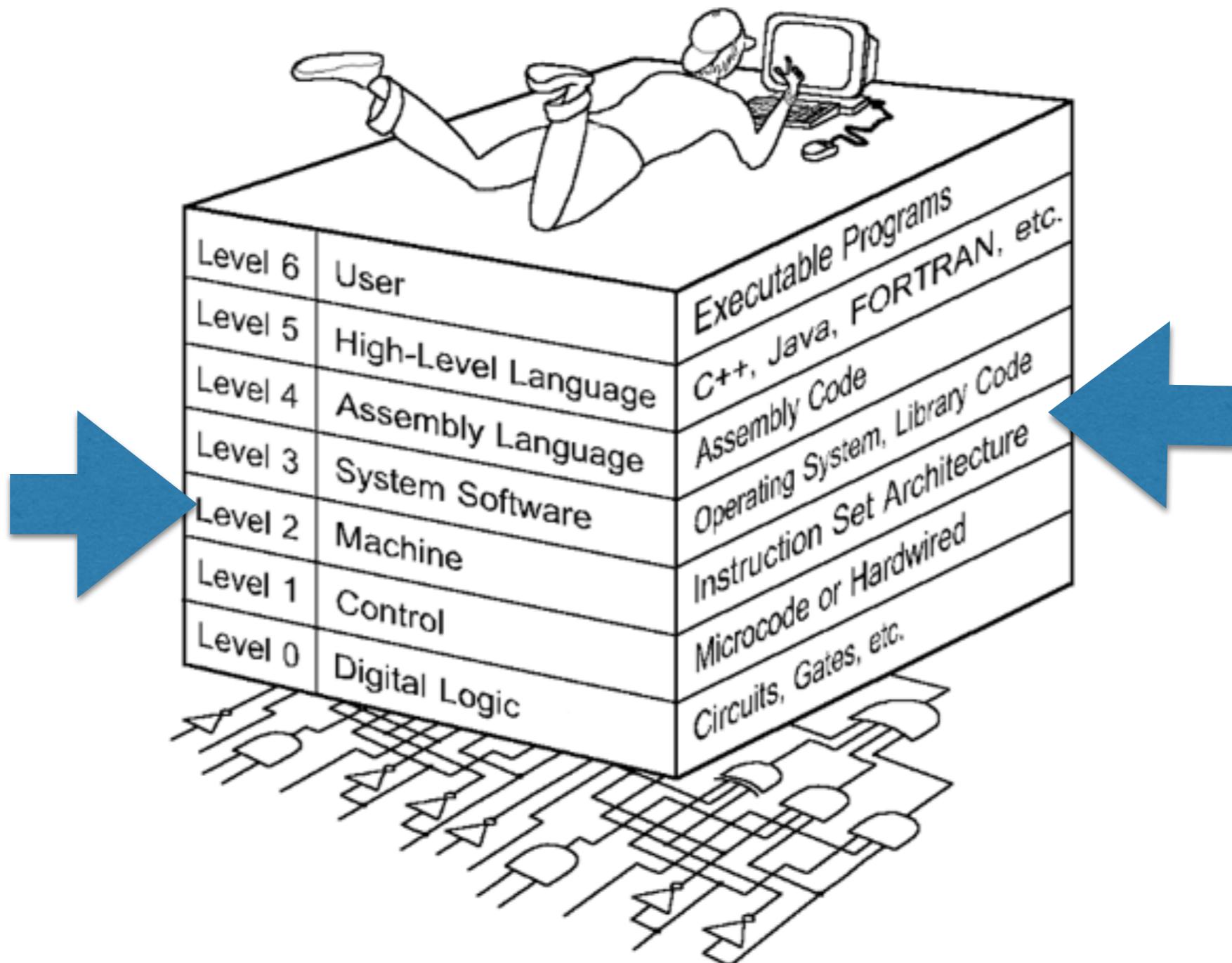
Parte 2/2

Organización del Computador 1



2017

Niveles de Abstracción de una Computadora



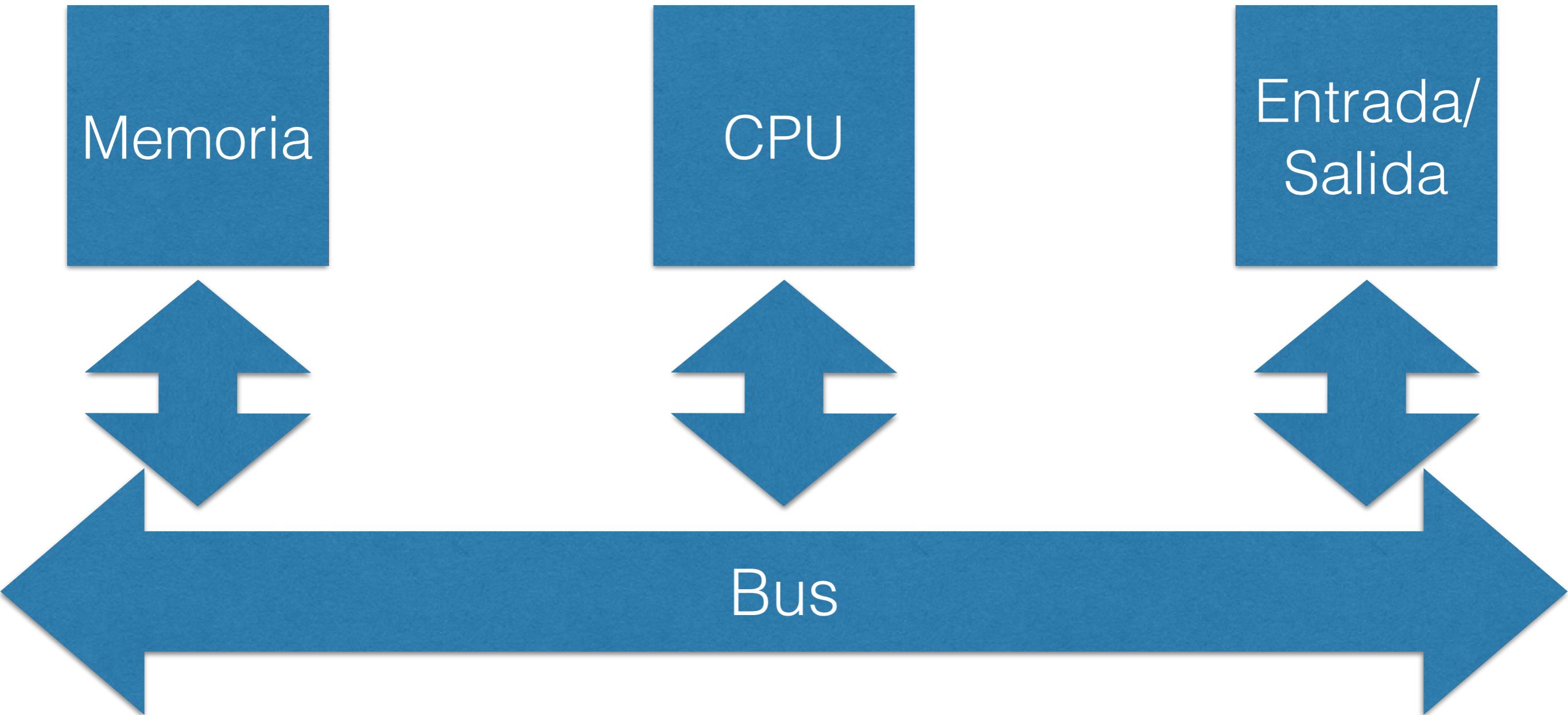
Modelo de Von Neumann

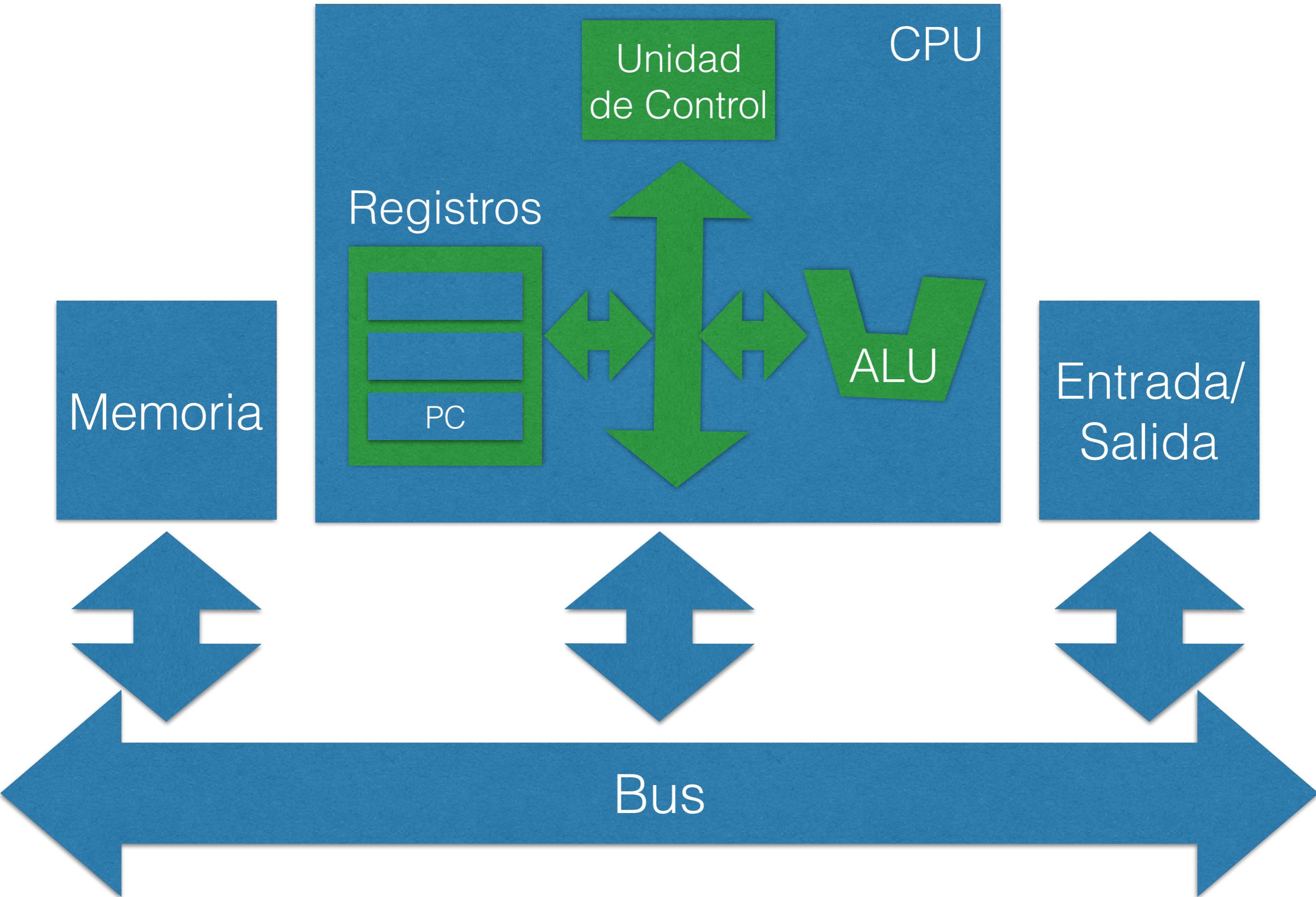
- John Von Neumann
(1903-1957)
- *Primer Informe sobre el EDVAC* (1945)
- Programas son almacenados como datos en memoria
- ¿Qué es un dato? ¿Qué es un programa?



Modelo Von Neumann

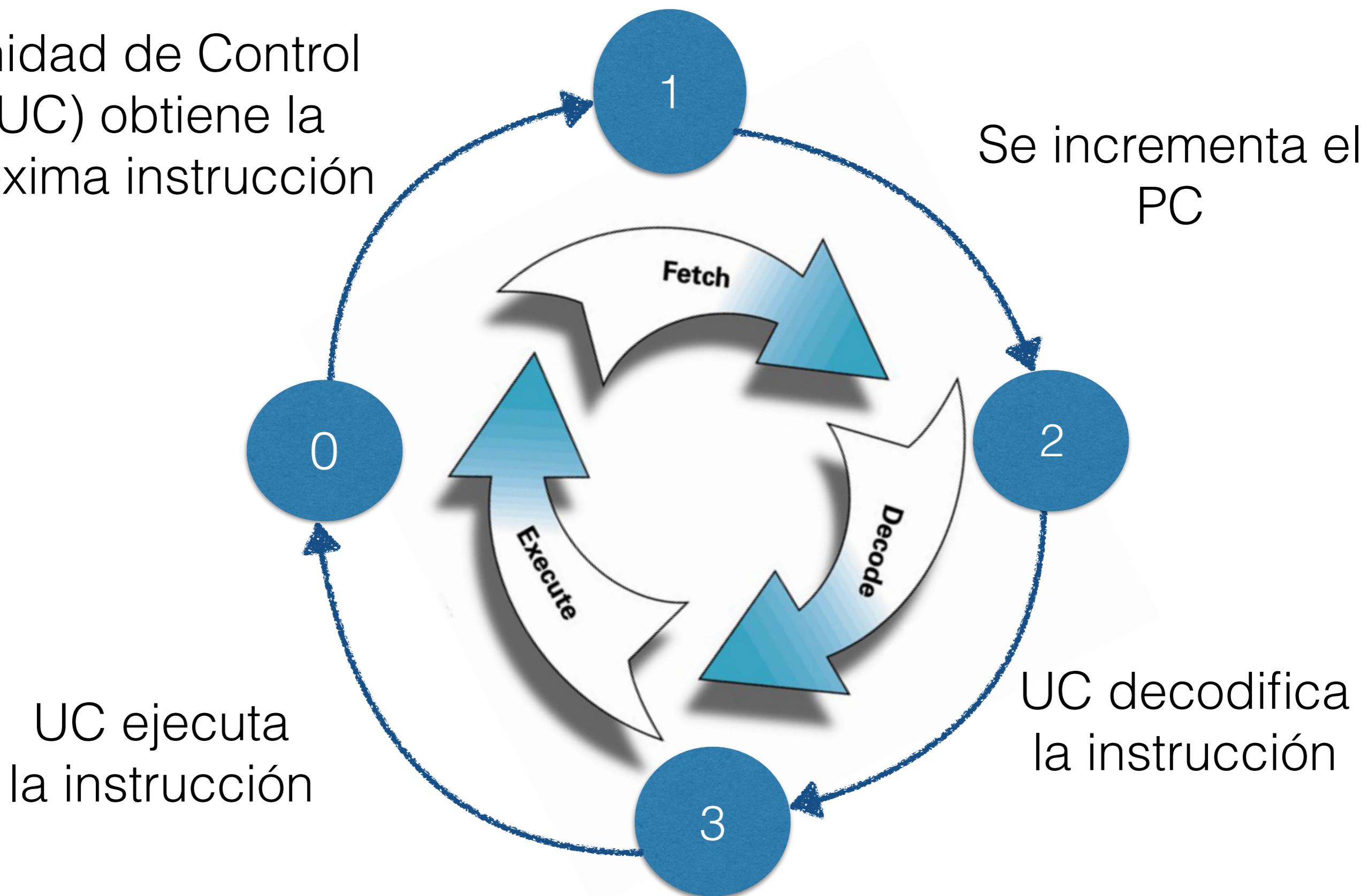
Organización





Ciclo de Instrucción

Unidad de Control
(UC) obtiene la
próxima instrucción



Arquitectura de una Computadora

- ¿Qué tipos de datos puede manejar “nativamente”?
 - ¿Cómo se almacenan?
 - ¿Cómo se acceden?
- ¿Qué operaciones (instrucciones) puede ejecutar?
 - ¿Cómo se codifican estas operaciones?

Arquitectura de una Computadora

- Un conjunto de respuestas a las preguntas anteriores se “llama” **Instruction Set Architecture** (~ISA)
- Cuando estamos estudiando una ISA, la organización se vuelve un detalle que sólo nos interesa para entender el comportamiento

Métricas de una ISA

- Cantidad total de instrucciones disponibles (“largo”).
 - Complejidad del conjunto de instrucciones:
 - RISC: **R**educed **I**nstruction **S**et **C**omputer
 - CISC: **C**omplex **I**nstruction **S**et **C**omputer
 - Longitud de las instrucciones (“ancho”)
 - Cantidad de memoria que un programa requiere (“largo x ancho”)

¿Qué tipos de datos soporta?

- Tipos de datos
 - Enteros (8, 16, 32 bits; complemento a 2)
 - Números Racionales - Punto fijo
 - Números Racionales - Punto flotante
 - Binary Coded Decimal (BCD), American Standard Code for Information Interchange (ASCII)
- Almacenamiento
 - Big Endian
 - Little Endian

LittleEndian vs. BigEndian

- **Endianness** se refiere a la forma en que la computadora guarda los datos cuando el tamaño de la celda de memoria es menor al tamaño del dato.
 - Ejemplo: ¿Cómo almacenamos datos de 32 bits en celdas de memoria de 8 bits?
 - Ejemplo: ¿Cómo almacenamos datos de 64 bits en celdas de memoria de 16 bits?

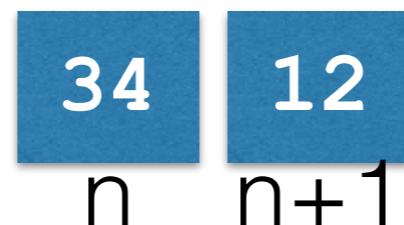
Little Endian vs. Big Endian

- **Endianness** se refiere a la forma en que la computadora guarda los datos cuando el tamaño de la celda de memoria es menor al tamaño del dato.
 - Little endian: la posición de memoria menor contiene el dato **menos** significativo. Ejemplo: Arquitecturas Intel (CISC)
 - Big endian: la posición de memoria menor contiene el dato **más** significativo. Ejemplo: Arquitecturas Motorola (RISC).

LittleEndian vs. BigEndian

- ¿Cómo se almacena 0x1234 (16bits) en 8 bits?

- Little Endian:

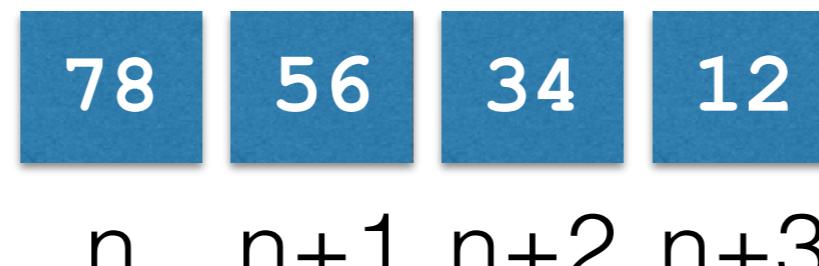


- Big Endian

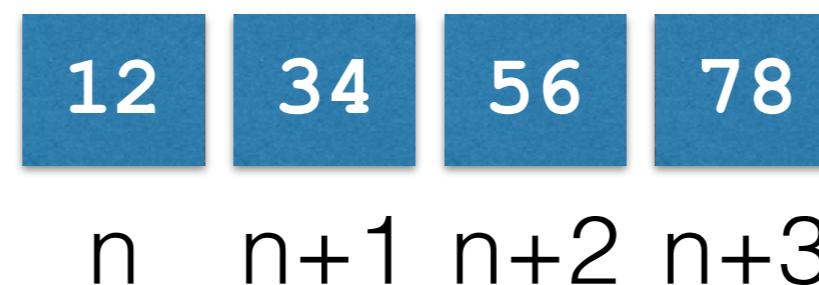


- ¿Cómo se almacena 0x12345678 (32 bits) en 8 bits?

- Little Endian:

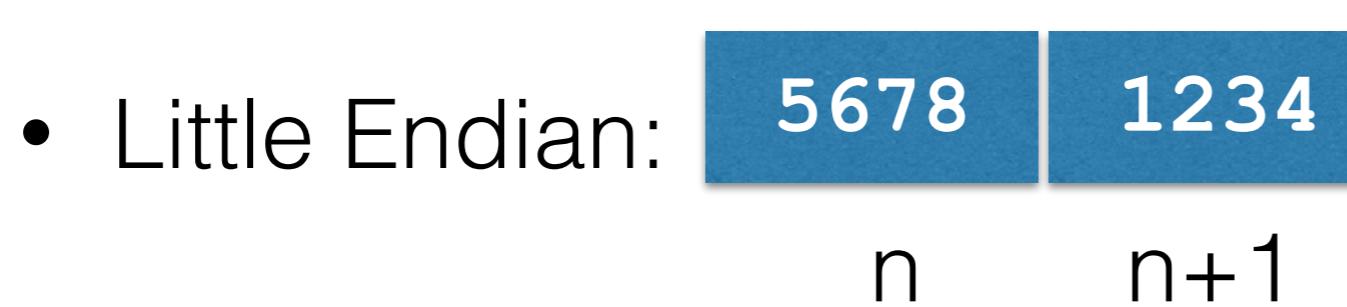


- Big Endian



LittleEndian vs. BigEndian

- ¿Cómo se almacena 0x12345678 (32 bits) en celdas de 16 bits?



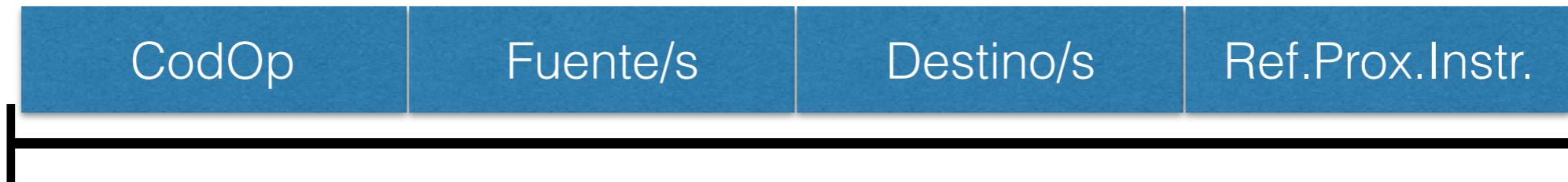
Acceso a los datos

- ¿Dónde se almacenan los datos?
 - ¿Tiene Registros?
 - ¿Cuánta Memoria tiene?
 - ¿Tiene Stack?
 - ¿Tiene Espacio de E/S diferenciado o compartido?
- ¿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)

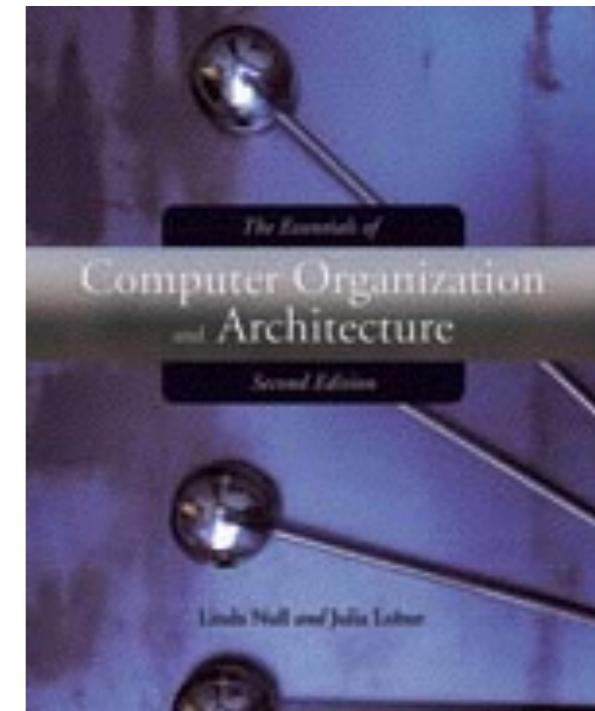
¿Cómo se codifican las operaciones?



- CodOp: representa la operación a realizar
- Fuente/s: provee información suficiente para obtener operandos de origen
- Destino/s: provee información suficiente para obtener la ubicación del resultado de la operación
- Ref. próx. instr.: provee información suficiente para determinar el próximo valor del PC.

Arquitectura MARIE

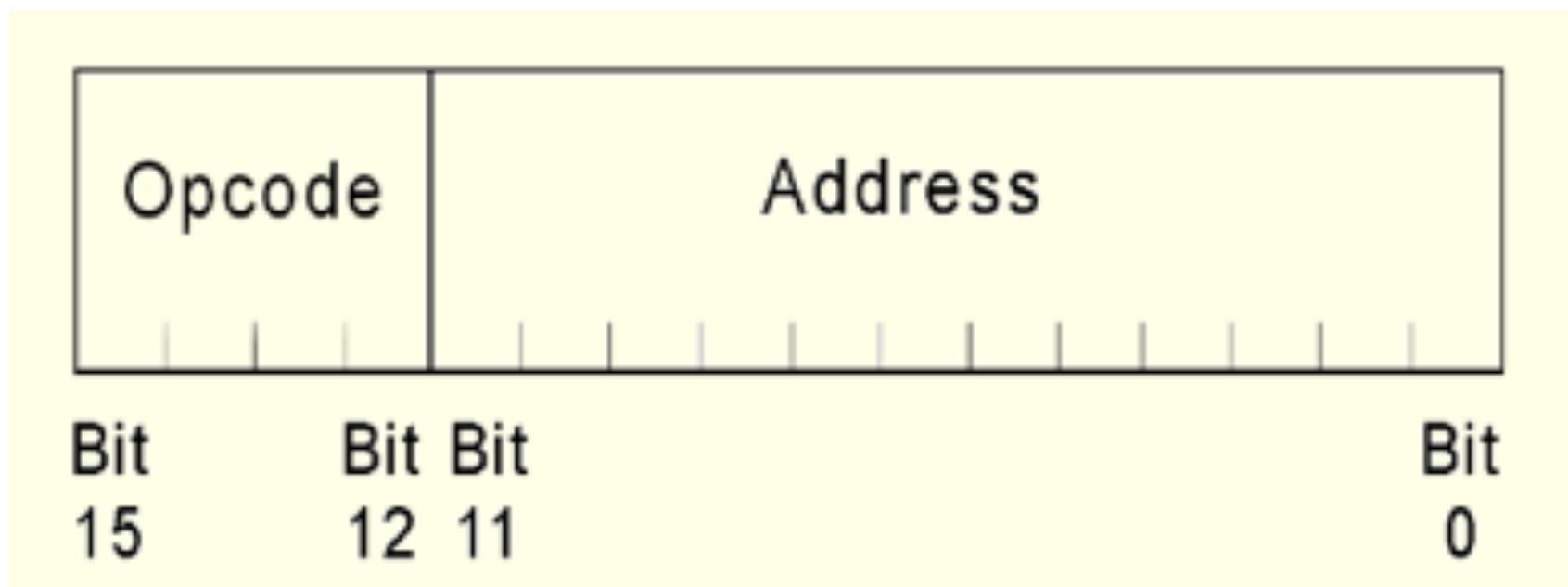
- Registros:
 - AC: Acumulador de 16 bits
 - PC: Program Counter de 12 bits
- Memoria:
 - $2^{12} = 4096$ direcciones
 - En cada dirección se almacenan 16 bits (Total $2^{12} \times 16$ bits = 8 KB)
 - Palabras (unidad de transferencia) = 16 bits
- Máquina de Acumulador (el registro AC es el operando implícito en casi todas las operaciones)



Instrucciones MARIE

Instrucción	Efecto
Load X	AC := [X]
Store X	[X] := AC
Add X	AC := AC + [X]
Subt X	AC := AC - [X]
Clear	AC:=0
Addi X	AC := AC + [[X]]
JnS X	[X] =PC (copia los menos significativos) y luego PC:=X +1
Skip Cond	Si Cond=00 y AC<0, entonces PC:=PC+1 Si Cond=01 y AC=0, entonces PC:=PC+1 Si Cond=10 y AC>0, entonces PC:=PC+1
Jump X	PC:=X
Jumpi X	PC:=[X] (copia los menos significativos)

MARIE: Formato de Instrucción



Instrucciones MARIE

Opcode	Instrucción	Efecto
“0001”	Load X	$AC := [X]$
“0010”	Store X	$[X] := AC$
“0011”	Add X	$AC := AC + [X]$
“0100”	Subt X	$AC := AC - [X]$
“1010”	clear	$AC := 0$
“1011”	Addi X	$AC := AC + [[X]]$
“0000”	JnS X	$[X] = PC$ (copia los menos significativos) y luego $PC := X + 1$
“1000”	Skip Cond	Si Cond=00 y $AC < 0$, entonces $PC := PC + 1$ Si Cond=01 y $AC = 0$, entonces $PC := PC + 1$ Si Cond=10 y $AC > 0$, entonces $PC := PC + 1$
“1001”	Jump X	$PC := X$
“1100”	Jumpi X	$PC := [X]$ (copia los menos significativos)

Arquitectura de Acumulador

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=[A]-[B]+[C]$ en lenguaje ensamblador MARIE

Arquitectura de Acumulador

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=([A]-[B])+[C]$ en lenguaje ensamblador MARIE

LOAD A # AC= [A]

SUBT B # AC= [A] - [B]

ADD C # AC= ([A] - [B]) + [C]

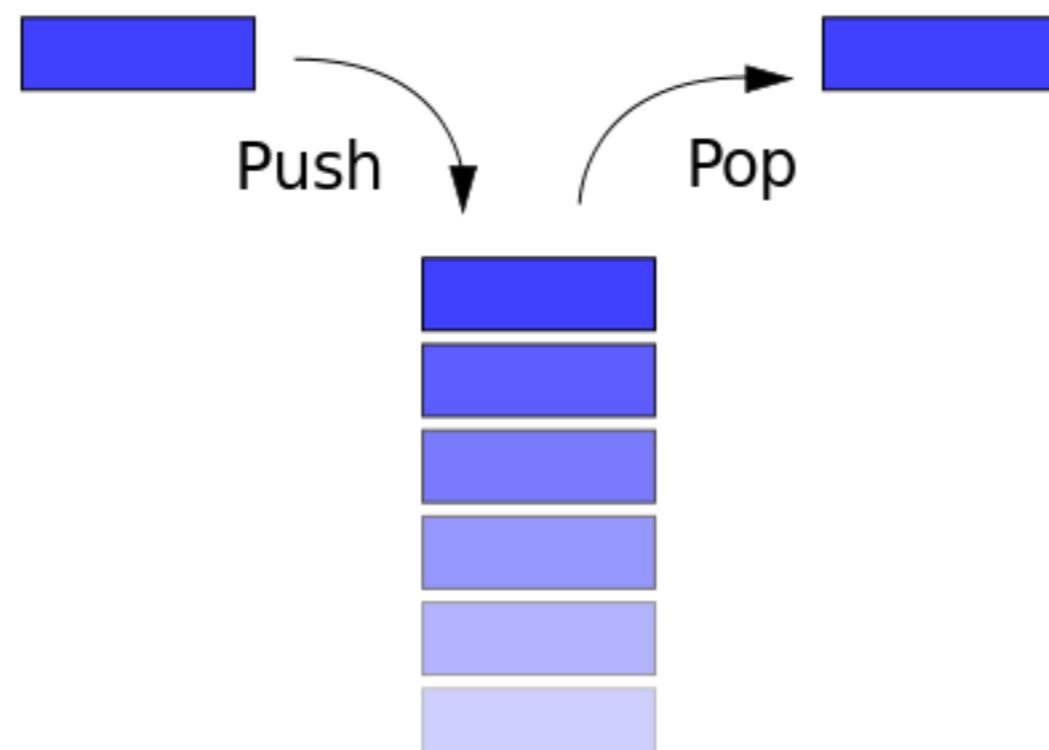
STORE D # [D]= ([A] - [B]) + [C]



La Pila

La Pila (Stack)

- Es una estructura de datos
- Puedo realizar solamente 2 operaciones:
 - PUSH: Agregar un dato al tope de la pila
 - POP: Retirar un dato que está al tope de la pila



Arquitecturas basadas en pila

- ¿Cómo se implementa una pila en memoria?
 - Se utiliza un registro (Stack Pointer) como puntero al **tope** de la pila únicamente
- Las stack machines:
 - Pueblan y despueblan la pila usando PUSH y POP
 - Todas las operaciones aritmético/lógicas obtienen los operando de la pila

Ejemplo: StackMARIE

- Registros:
 - PC (Program Pointer) de 12 bits
 - SP (Stack Pointer) de 12 bits
- Memoria (=MARIE)
- Longitud de Palabra (=MARIE)
- Formato de Instrucción (=MARIE)

Instrucciones StackMARIE

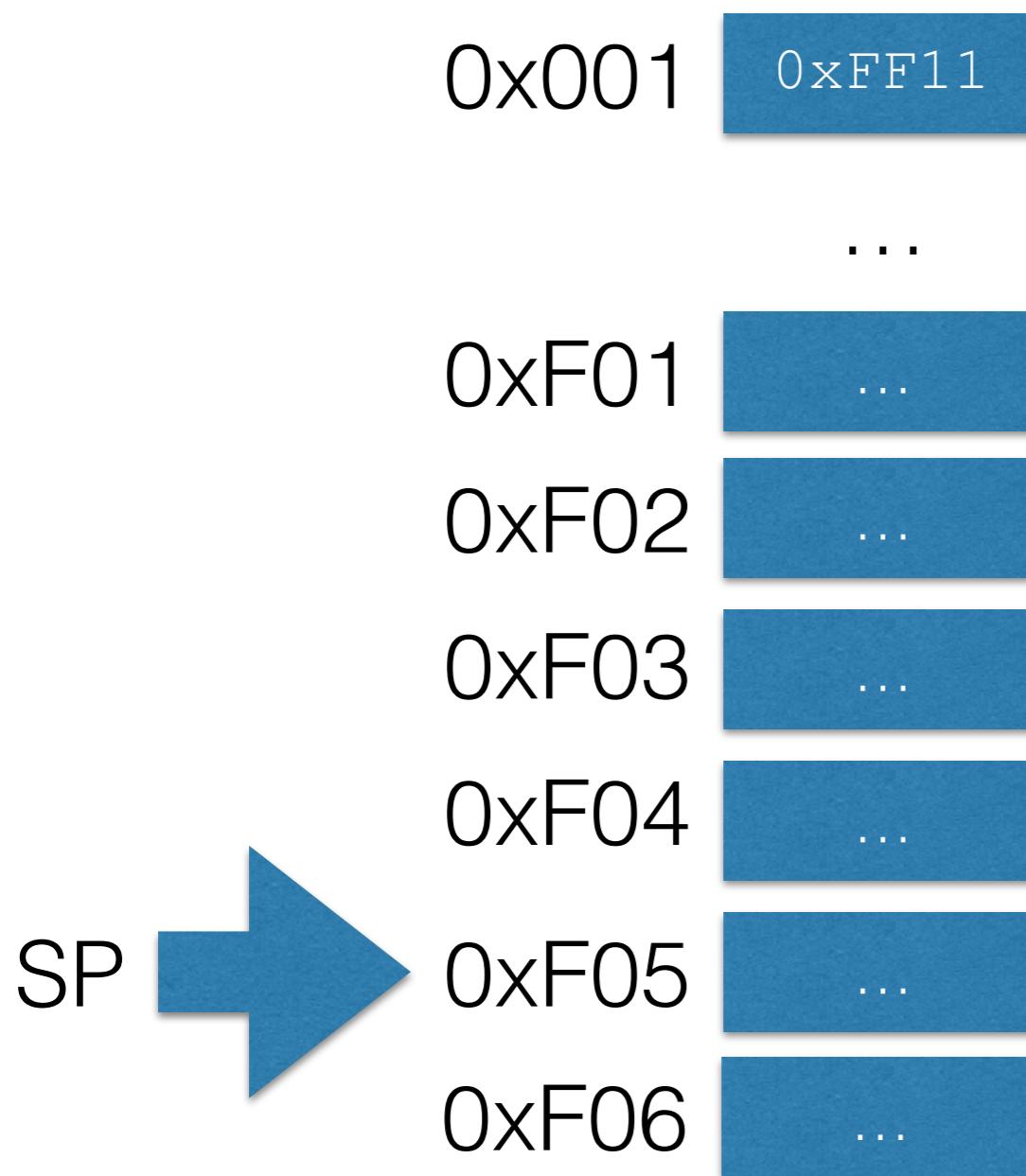
OpCode	Instrucción	Comportamiento
“0000”	PUSH X	push([X])
“0001”	POP X	[X]:=pop()
“0010”	ADD	push(pop()+pop())
“0011”	SUB	push(pop()-pop())
“0100”	AND	push(pop()&pop())
“0101”	OR	push(pop() pop())
“0110”	NOT	push(~pop())
“0111”	LE	push(pop()<=pop())
“1000”	GE	push(pop()>=pop())
“1001”	EQ	push(pop()==pop())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

push()

- PUSH X
 - 1. Lee el contenido de la dirección X
 - 2. Copia el dato leído a la dirección apuntada por el registro SP (Stack Pointer)
 - 3. Decrementa SP

PUSH 0x001

- PUSH X



- 1. Lee el contenido de la dirección X**
2. Copia el dato leído a la dirección apuntada por el registro SP (Stack Pointer)
3. Decrementa SP

PUSH 0x001

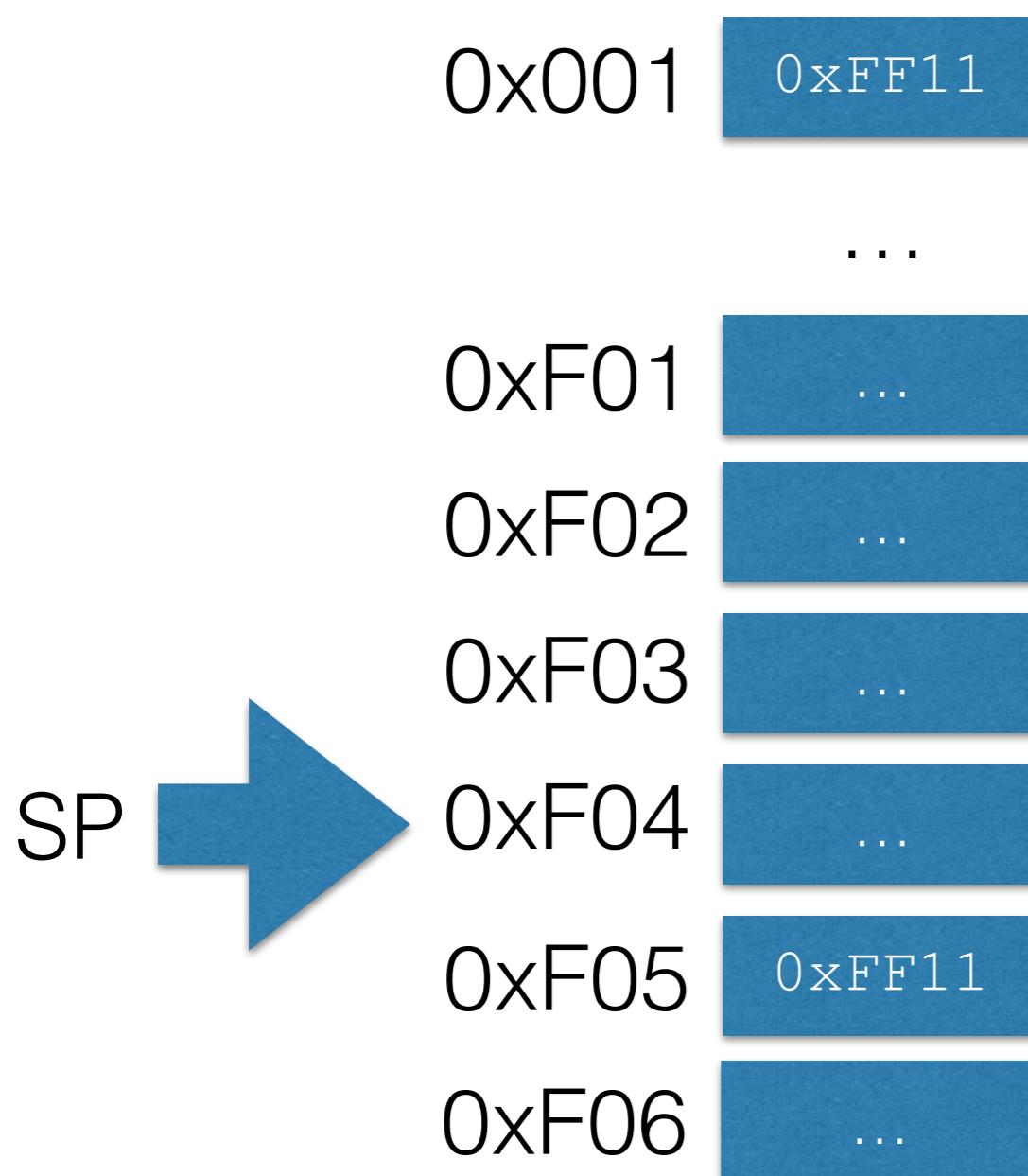
- PUSH X



1. Lee el contenido de la dirección X
2. **Copia el dato leído a la dirección apuntada por el registro SP (Stack Pointer)**
3. Decrementa SP

PUSH 0x001

- PUSH X

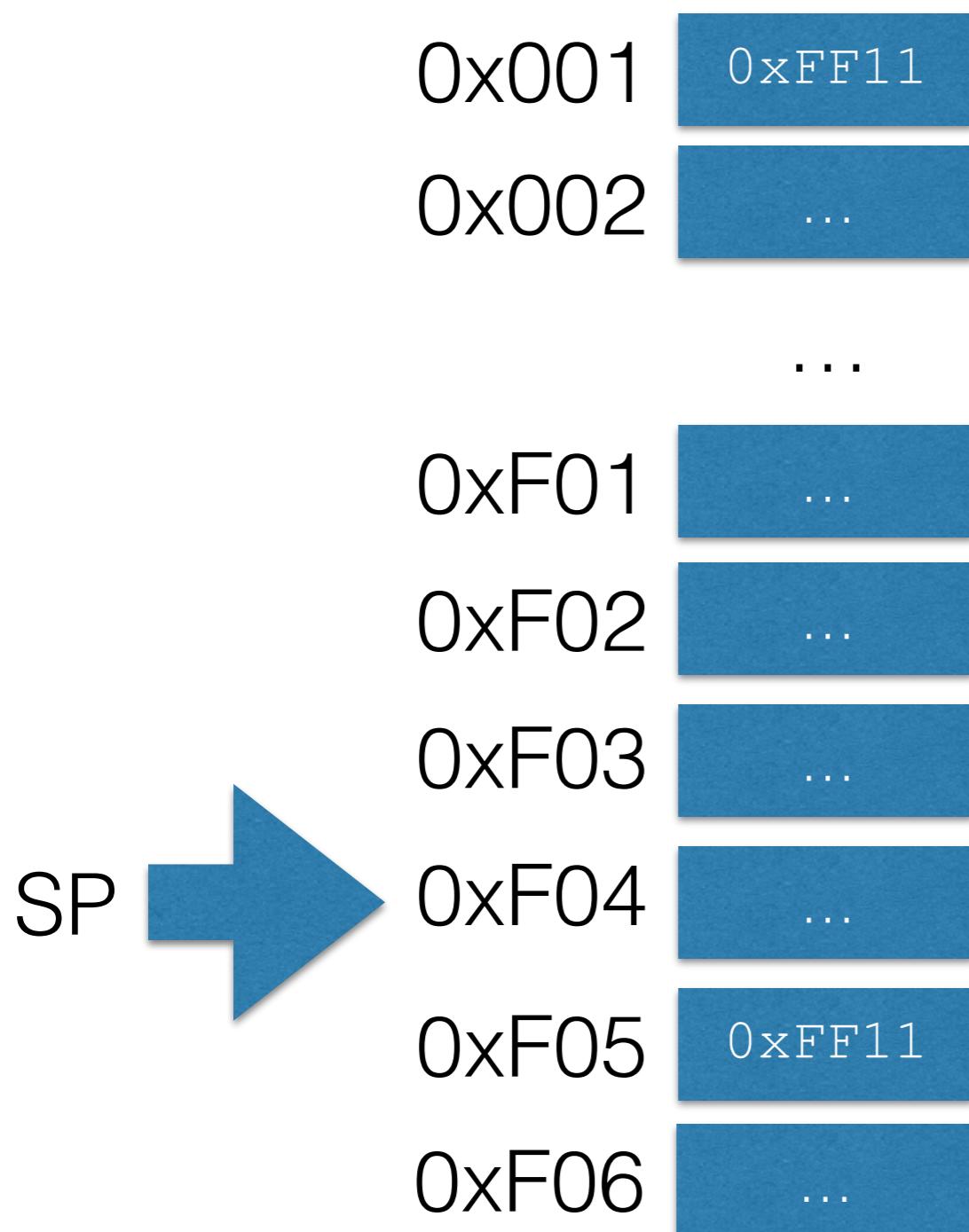


1. Lee el contenido de la dirección X
2. Copia el dato leído a la dirección apuntada por el registro SP (Stack Pointer)
3. **Decrementa SP**

pop()

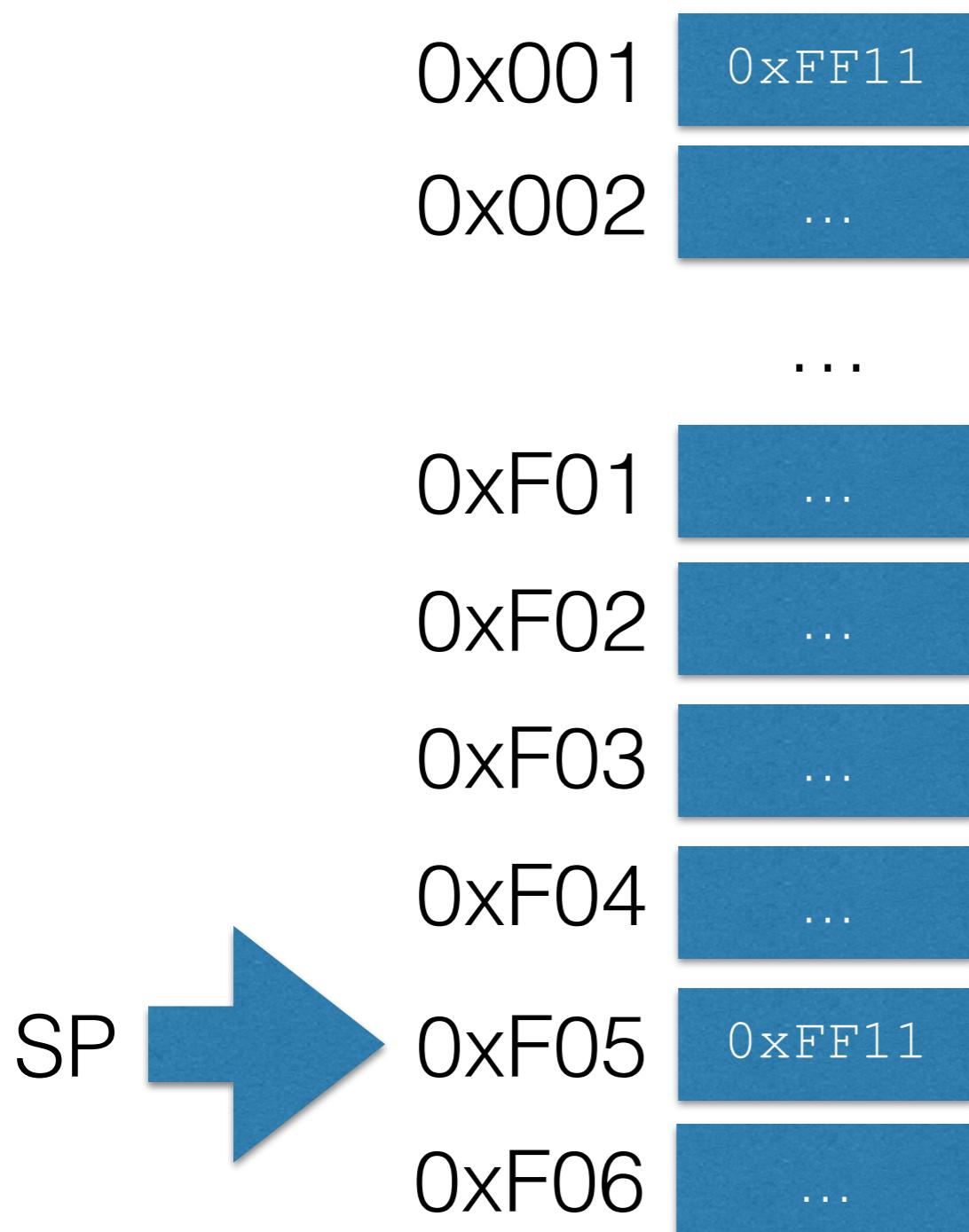
- POP X
 1. Incrementa el SP
 2. Lee el contenido de la dirección SP (Stack Pointer)
 3. Copia el dato leído en la dirección X

POP 0x002



- POP X
 - 1. Incrementa el SP
 - 2. Lee el contenido de la dirección SP (Stack Pointer)
 - 3. Copia el dato leído en la dirección X

POP 0x002



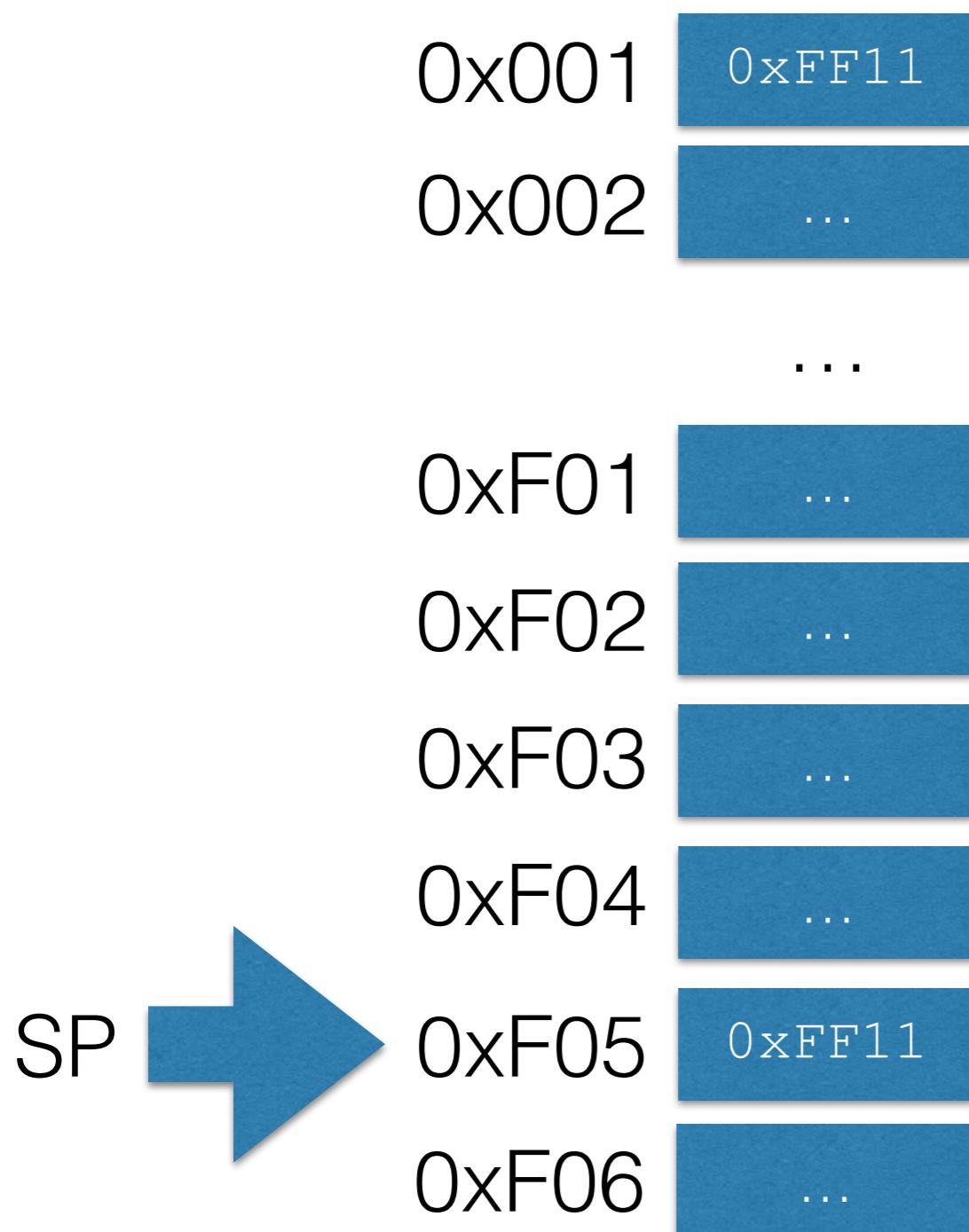
- POP X

1. Incrementa el SP

2. Lee el contenido de la dirección SP (Stack Pointer)

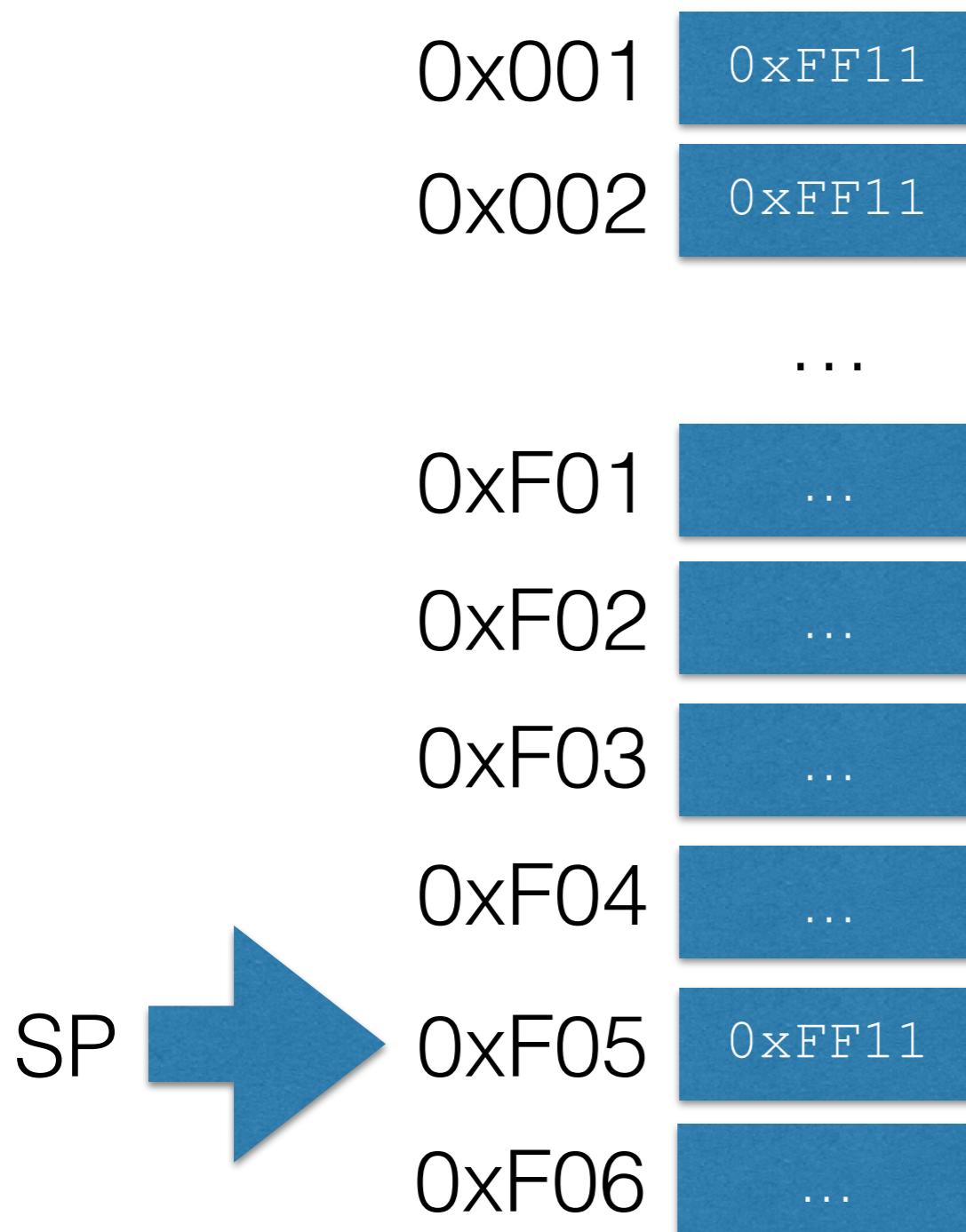
3. Copia el dato leído en la dirección X

POP 0x002



- POP X
1. Incrementa el SP
 2. **Lee el contenido de la dirección SP (Stack Pointer)**
 3. Copia el dato leído en la dirección X

POP 0x002



- POP X
 - 1. Incrementa el SP
 - 2. Lee el contenido de la dirección SP (Stack Pointer)
 - 3. Copia el dato leído en la dirección X**

Arquitectura de Pila

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=[A]-[B]+[C]$ en lenguaje ensamblador StackMARIE

Arquitectura de Pila

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=[A]-[B]+[C]$ en lenguaje ensamblador StackMARIE

PUSH B # pila={ [B] }

PUSH A # pila={ [A] , [B] }

SUB # pila={ [A] - [B] }

PUSH C # pila={ [C] , [A] - [B] }

ADD # pila={ [C] + ([A] - [B]) }

POP D # pila={ } , [D]=[C] + ([A] - [B])

Arquitecturas GPR

- GPR: General Purpose Register
- Arquitecturas de registros de propósito general
- Podemos utilizar varios registros
- Caso típico: arquitecturas RISC
- ... y la mayor parte de las arquitecturas actual
 - Intel, ARM, etc.



Arquitectura Orga1

- Memoria:
 - Direcciones de 16 bits (0x0000 a 0xFFEF)
 - Palabras de 16 bits
 - Direccionamiento a 16 bits



Arquitectura Orga1

- 8 registros de 16 bits de **propósito general**
- 5 registros de **propósito específico**:
 - PC (Program Pointer) 16 bits
 - SP (Stack Pointer) 16 bits
 - IR0,IR1,IR2 (Instruction Register) 16 bits c/u
 - 4 **flags**: Z, N, C y V



Flags

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



Flags: Ejemplo

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

Flags y control de flujo

		<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

- ¿Por qué usar flags en lugar de consultar el valor de registros o memoria?
- ¿Por qué hay JL (Menor) y JCS (Menor sin signo)?

Flags y Control de Flujo

- Sean los valores almacenados en R0 y R1 en notación **sin signo** de 16 bits.
- Escribir un programa en lenguaje ensamblador ORGA1 tal que:

IF R0>R1 then

Then R0 := 0

Endif

Flags y Control de Flujo

- Sean los valores almacenados en R0 y R1 en notación **sin signo** de 16 bits.
- Escribir un programa en lenguaje ensamblador ORGA1 tal que:

IF R0>R1 then	CMP R0, R1
	JCS then
Then R0 := 0	JMP endif
	then: MOV R0, 0
Endif	endif: ...

Flags y Control de Flujo

- Sean los valores almacenados en R0 y R1 en notación **complemento a 2** de 16 bits.
- Escribir un programa en lenguaje ensamblador ORGA1 tal que:

IF R0>R1 then

CMP R0, R1

Then R0 := 0

JCS then

Endif

JMP endif

then: MOV R0, 0

endif: ...

Flags y Control de Flujo

- Sean los valores almacenados en R0 y R1 en notación **complemento a 2** de 16 bits.
- Escribir un programa en lenguaje ensamblador ORGA1 tal que:

IF R0>R1 then

CMP R0, R1

Then R0:=0

JL **then**

Endif

JMP endif

then: MOV R0, 0

endif: ...

Arquitectura GPR

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=[A]-[B]+[C]$ en lenguaje ensamblador ORGA1

Arquitectura GPR

- Sean A,B,C y D direcciones de memoria

Escribir un programa que realice $[D]:=([A]-[B])+[C]$ en lenguaje ensamblador ORGA1

MOV R0, [A] # R0=[A]

MOV R1, [B] # R0=[A], R1=[B]

MOV R2, [C] # R0=[A], R1=[B], R2=[C]

SUB R0, R1 # R0=[A] - [B]

ADD R0, R2 # R0= ([A] - [B]) + [C]

MOV [D], R0 # [D]= ([A] - [B]) + [C]

Arquitectura GPR

- Otra solución para lenguaje ensamblador ORGA1:

```
MOV R0, [A] # R0=[A]
```

```
SUB R0, [B] # R0=[A] - [B]
```

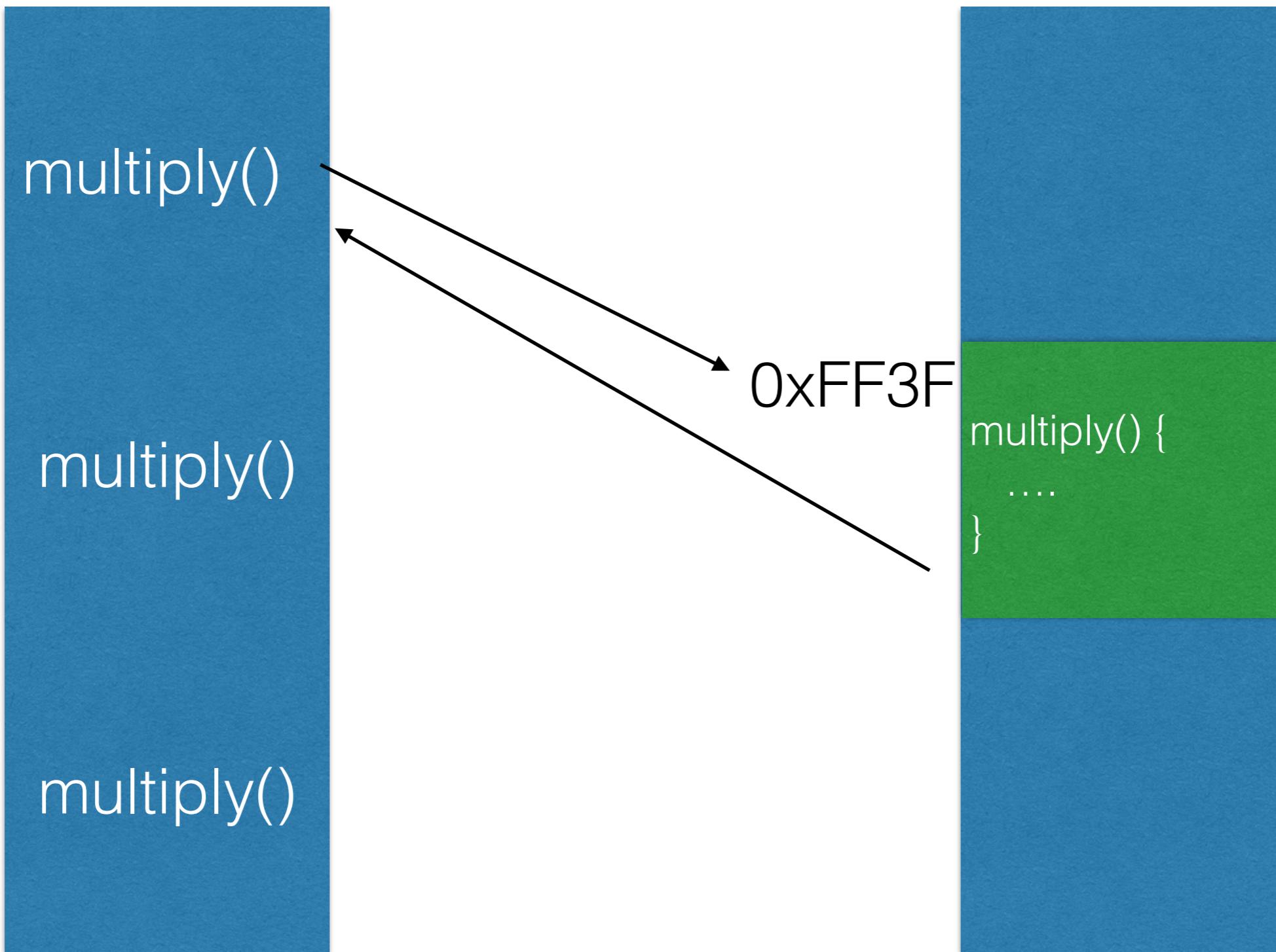
```
ADD R0, [C] # R0= ( [A] - [B] ) + [C]
```

```
MOV [D], R0 # [D]= ( [A] - [B] ) + [C]
```

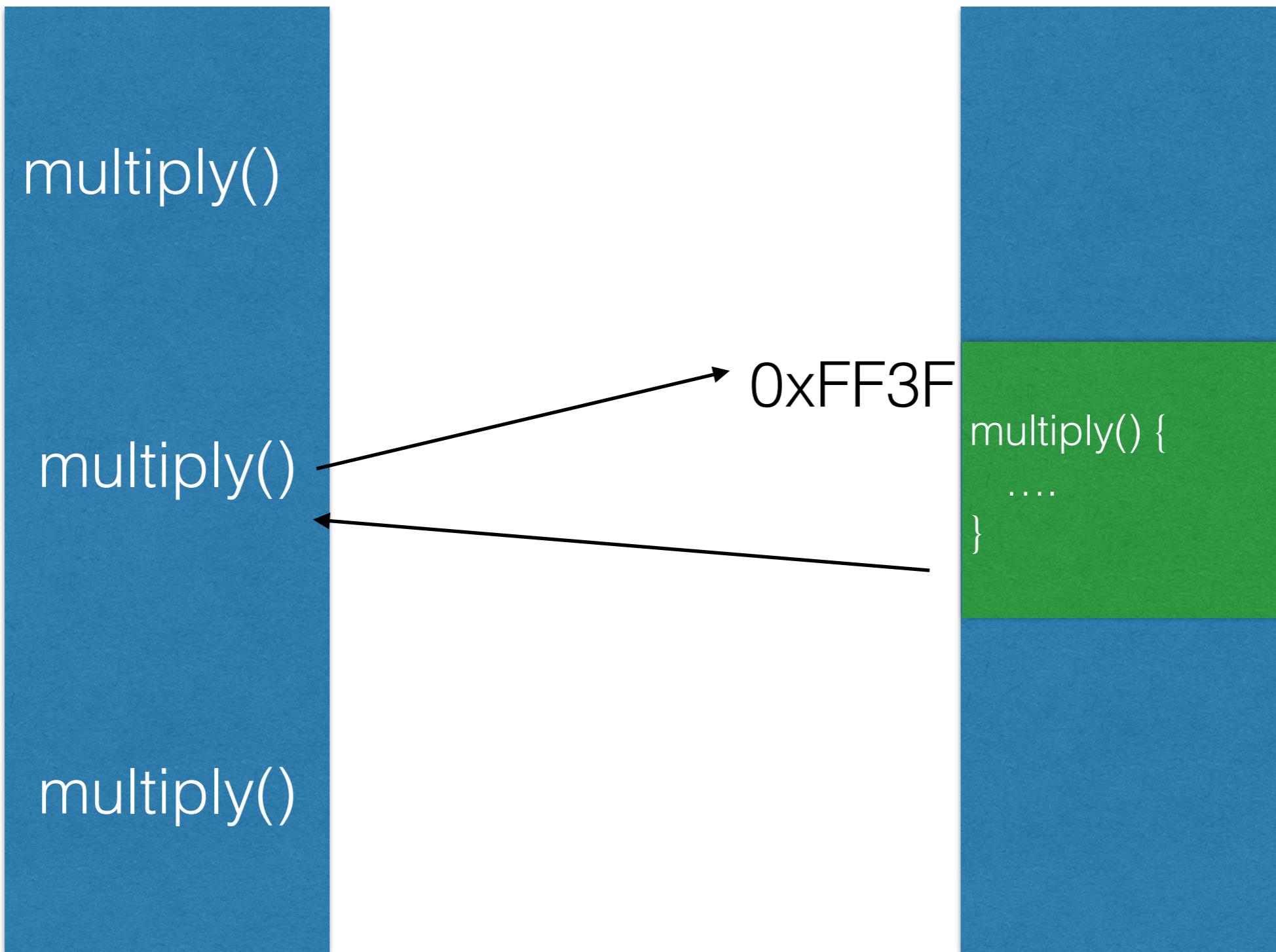
Subrutinas

- Son alteraciones del flujo secuencial que permiten retornar al lugar de donde fueron invocadas.
- Evitan repetir el mismo programa múltiples veces
 - **Ejemplo:** una subrutina que multiplique 2 enteros.
 - Es mejor no tener el mismo programa repetido 1000 veces si se usa 1000 veces.

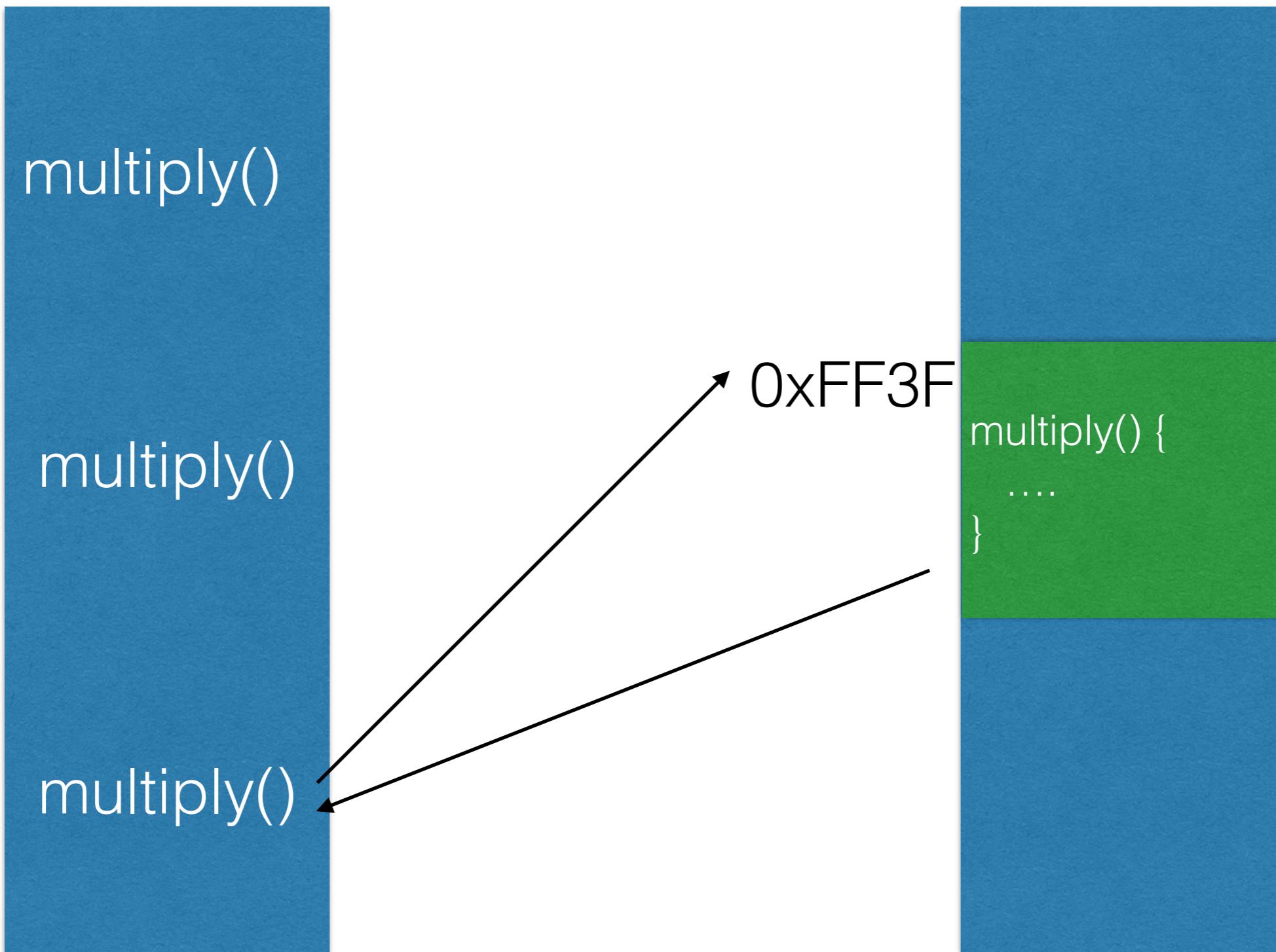
Subrutinas



Subrutinas



Subrutinas



Subrutinas

- Para poder invocar una subrutina necesitamos:
 - Almacenar la dirección de retorno
 - Almacenar los parámetros que la subrutina utiliza (ejemplo: los enteros a ser multiplicados)
 - Almacenar el resultado de la subrutina

Subrutinas - Retorno

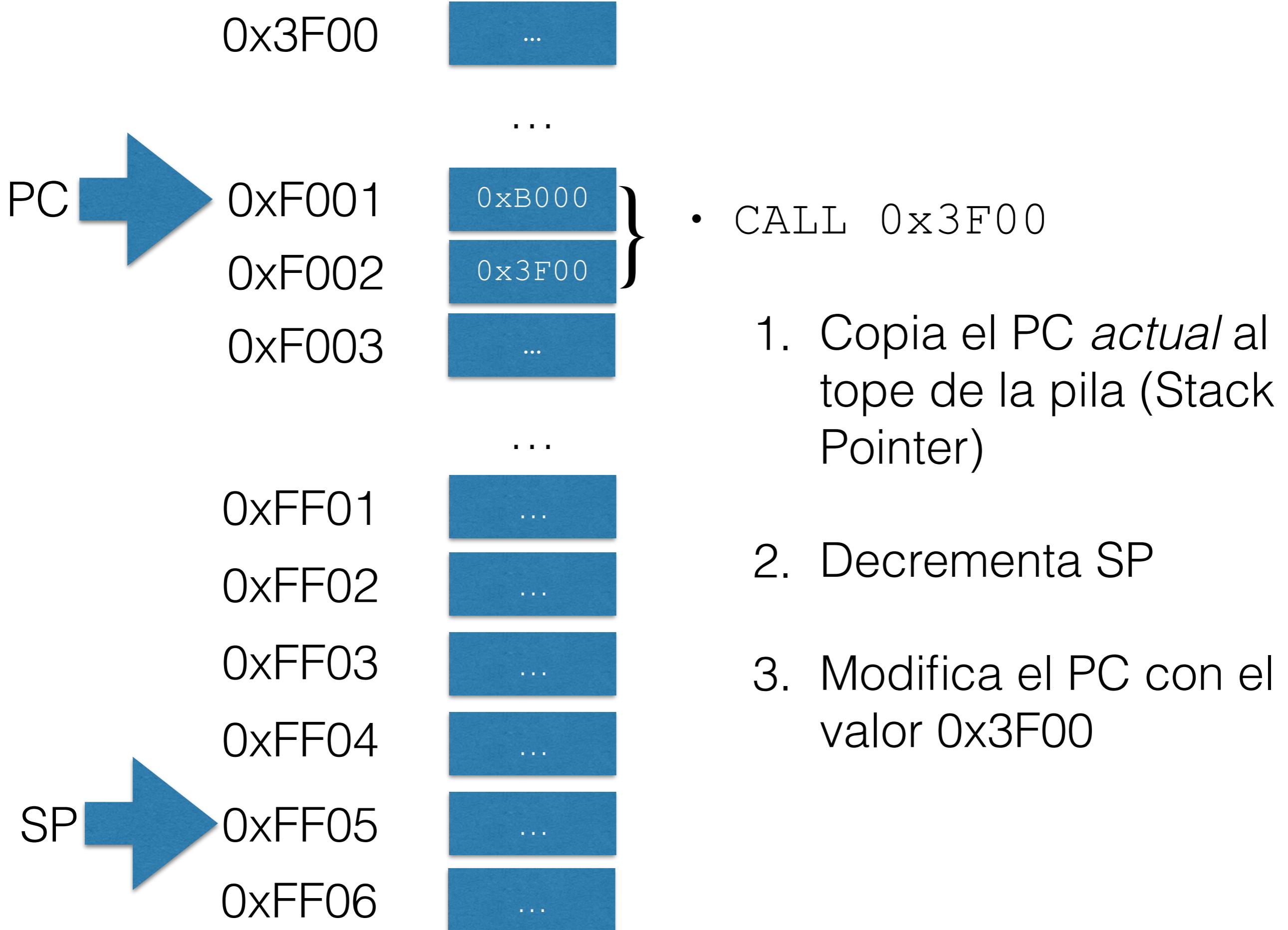
- Para almacenar la dirección de retorno podemos usar:
 - Un registro
 - Una dirección al ppio. de la suburbana (JnS - Jump and Store)
 - **Problema:** estas soluciones no permiten recursión (¿Cómo podemos lograr recursión?)

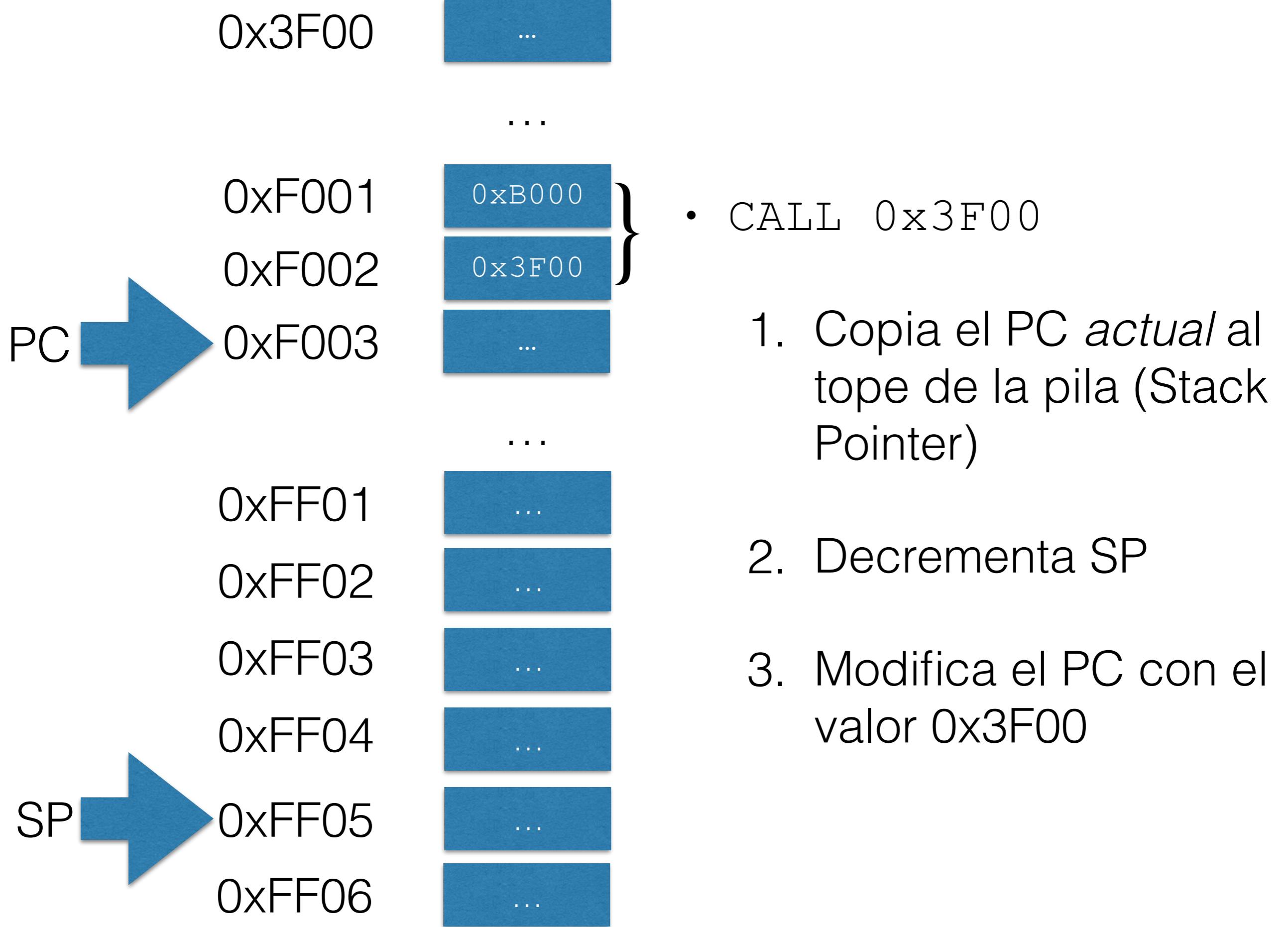
Subrutinas - Recursión

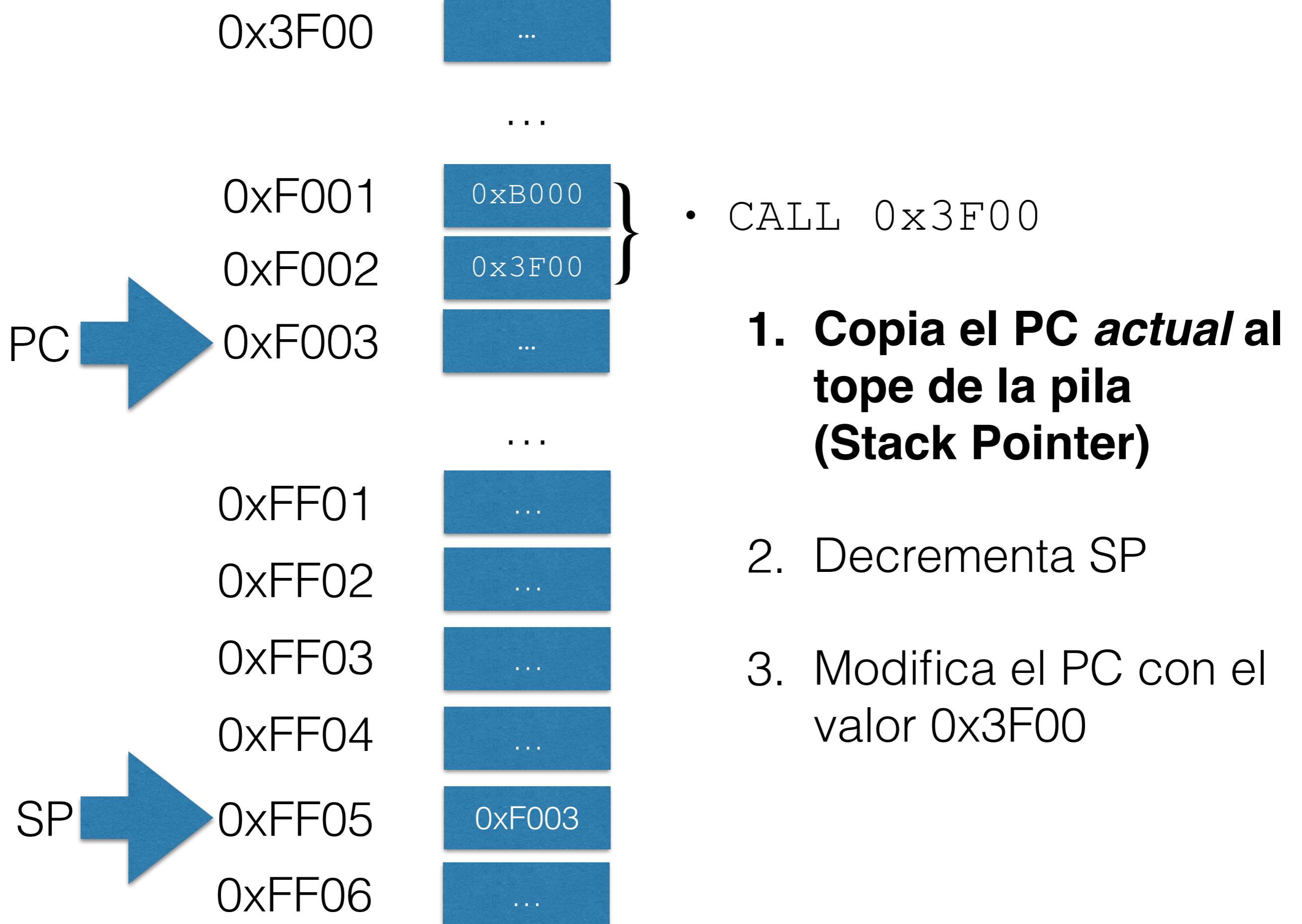
- Para poder soportar recursión podemos usar la **pila** (SP, Stack Pointer)
 - Al invocar la subrutina, **apilamos** la dirección de retorno en el tope de la pila
 - Para retornar, **desapilamos** la dirección de retorno del tope de la pila, y la copiamos al PC.
 - La siguiente instrucción a fatchear será del programa llamador

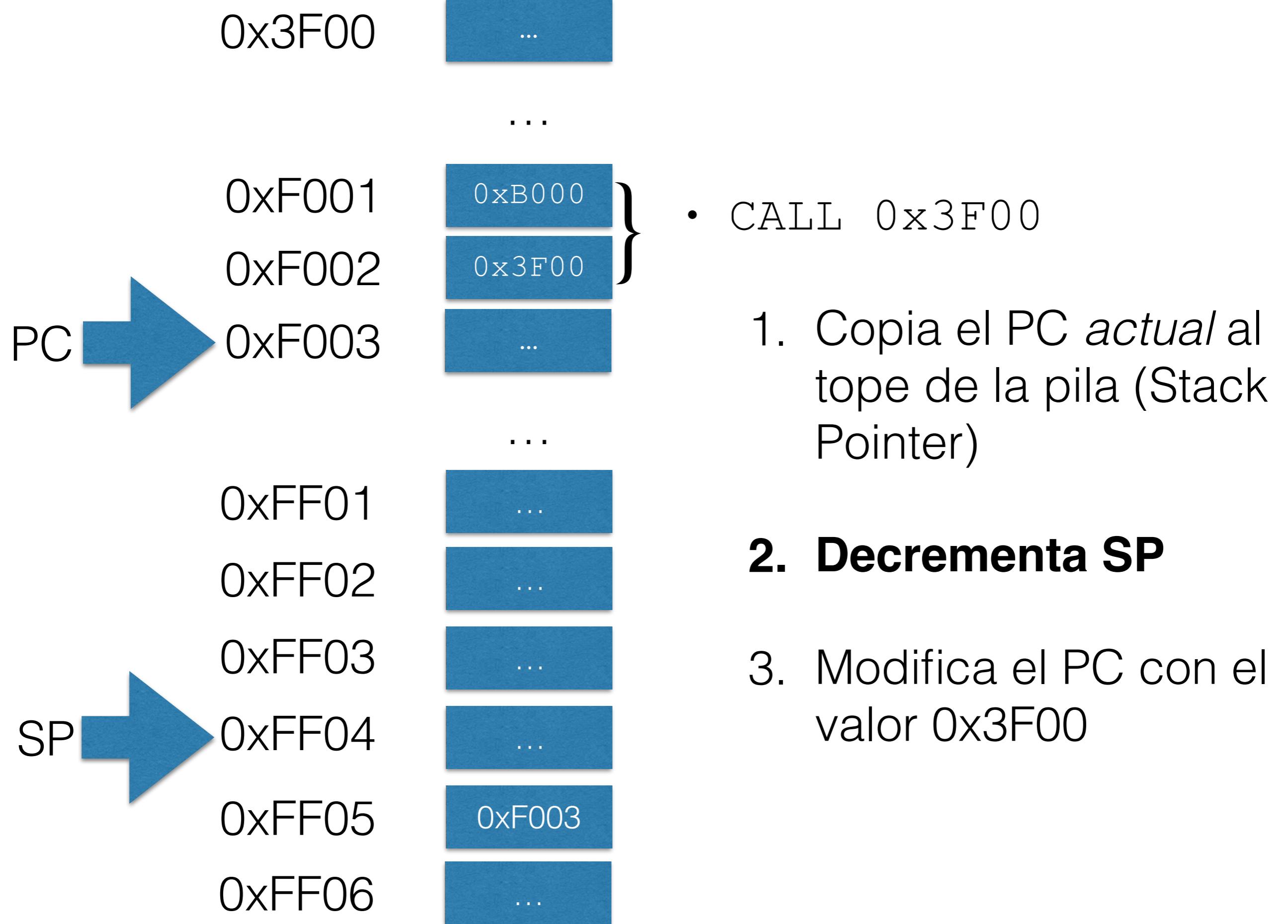
Subrutinas en la Arquitectura Orga1

- CALL X
 - X es la dirección del inicio de la subrutina a ejecutar
 - Apila la dirección de retorno (decrementa el SP) y modifica el PC con el valor X
- RET
 - Desapila el tope de la pila (incrementa el SP), y modifica el PC con el tope de la pila



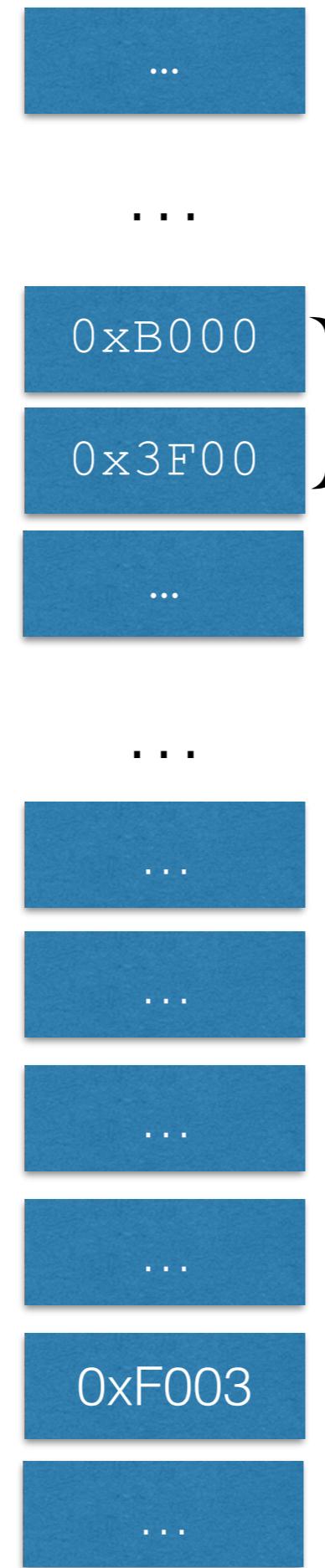






PC → 0x3F00

SP → 0xFF04



- CALL 0x3F00

1. Copia el PC *actual* al tope de la pila (Stack Pointer)

2. Decrementa SP

3. **Modifica el PC con el valor 0x3F00**

PC → 0x3F00
0x3F01

0xC000 } RET

...

0xF001

0xB000

0xF002

0x3F00

0xF003

...

- RET

...

0xFF01

...

0xFF02

...

0xFF03

...

SP → 0xFF04

...

0xFF05

0xF003

0xFF06

...

1. Incrementa SP
2. Modifica el PC con el valor al que apunta SP

PC →

0x3F00
0x3F01
...

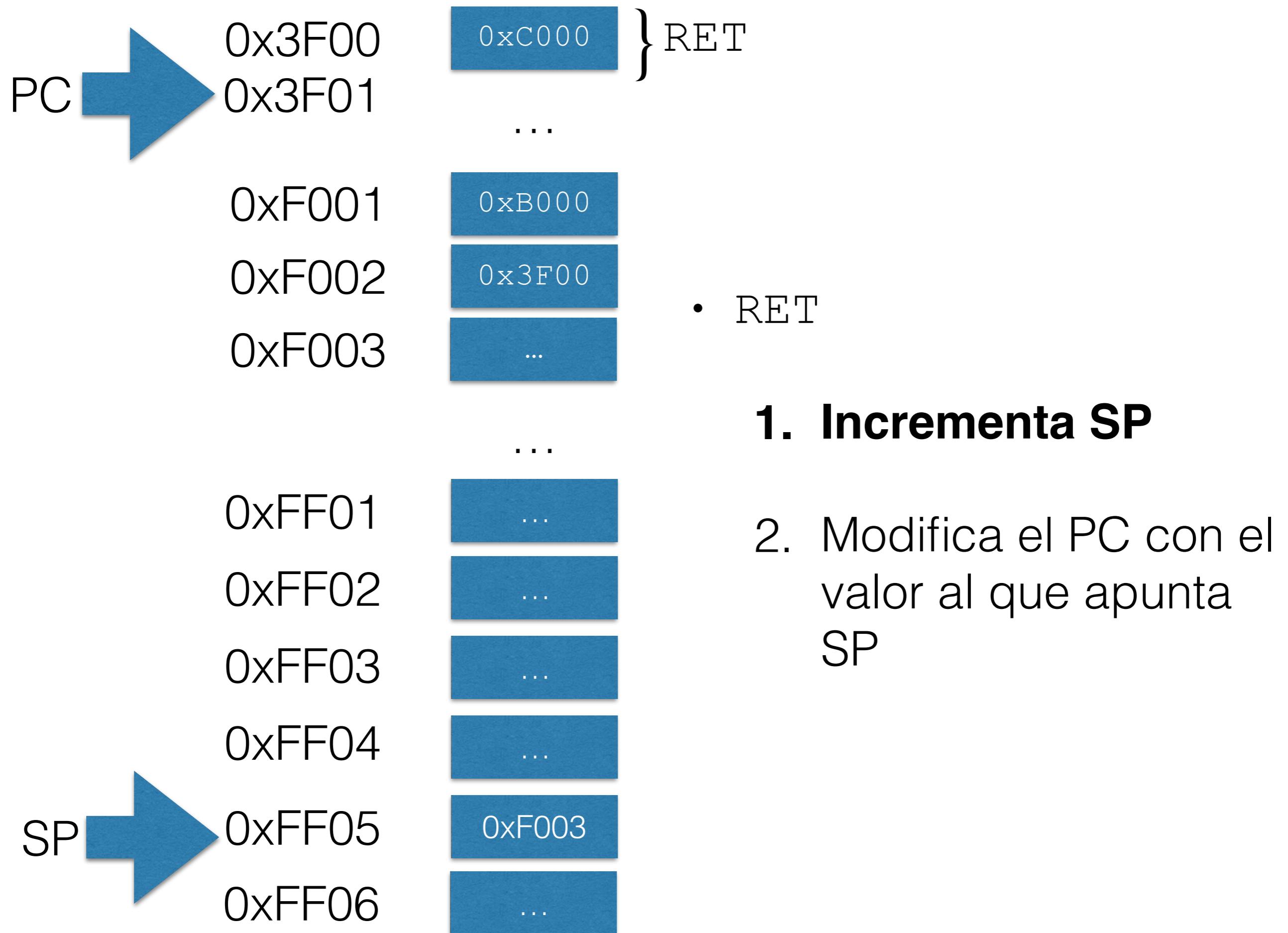
0xC000 } RET
0xB000
0x3F00
...

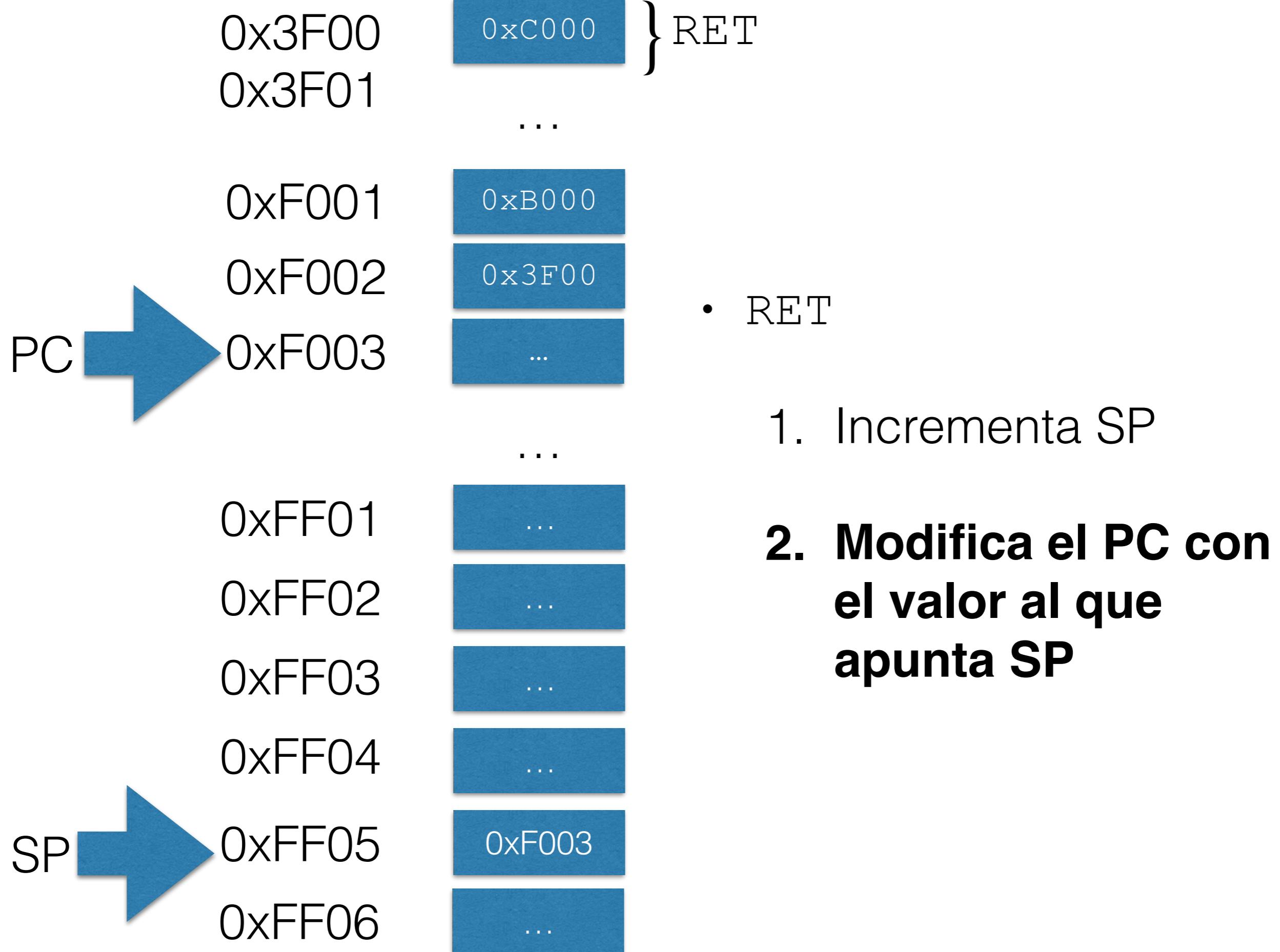
- RET

SP →

0xF001
0xF002
0xF003
0xF004
0xF005
0xF006
...

1. Incrementa SP
2. Modifica el PC con el valor al que apunta SP





CALL/RET

- CALL: permite invocar una subrutina
- RET: return/permite retornar a la instrucción siguiente al llamado
- Adicionalmente, se deben pasar los parámetros
 - Por registro/memoria (si no hay instrucciones PUSH/POP)
 - Por pila (si hay instrucciones PUSH/POP)

Resumen

- Arquitecturas:
 - De Acumulador (MARIE)
 - De pila (StackMARIE)
 - De GPR (ORGA1)
- Flags/Saltos condicionales
- Llamados a subrutinas

Bibliografía

- Capítulos 4 y 5 del Libro de Null.