

Arquitectura

Parte 2/2

Organización del Computador 1
1er Cuatrimestre 2018

Repaso

- Vimos modelo de Von Neumann, ciclo de instrucción y modos de direccionamiento
- Vimos que una instrucción se compone de
 - OpCode
 - Fuente/s
 - Destino/s
- ¿Qué pueden ser estas Fuentes y Destinos?
 - Constantes
 - Registros
 - Direcciones de memoria

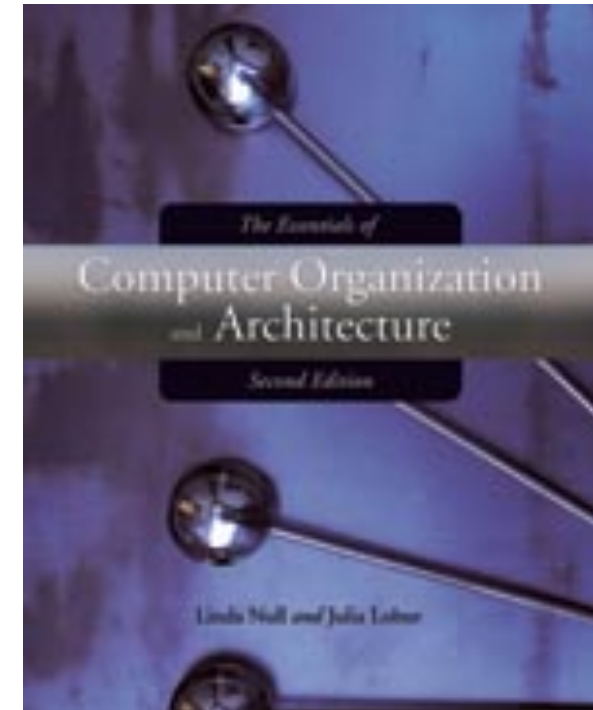


Repaso

- ¿Qué representan?
 - Constantes
 - Referencias a variables
 - Referencias a arreglos (arrays)
 - Referencias a sub-rutinas.
 - Estructuras de datos (listas, colas, etc.)
- Ejemplo de Arquitectura ORGA1

Arquitectura MARIE

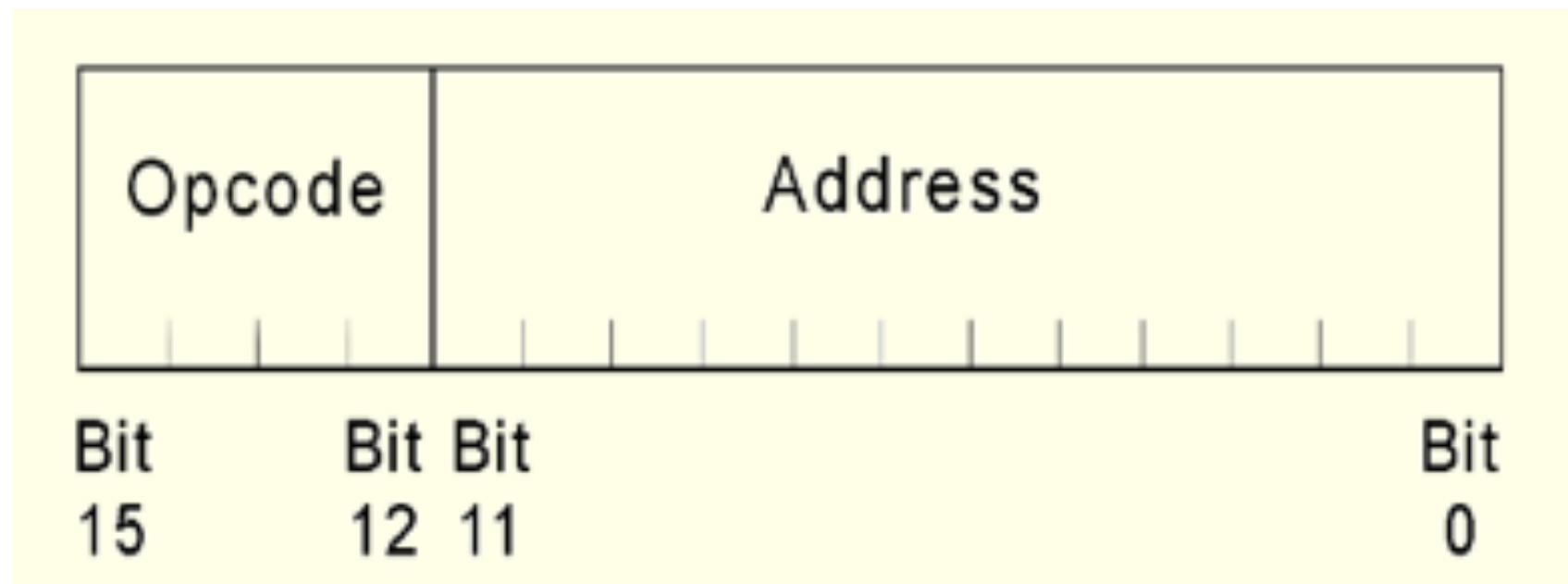
- Registros:
 - AC: Acumulador de 16 bits
 - PC: Program Counter de 12 bits
- Memoria:
 - $2^{12} = 4096$ direcciones
 - En cada dirección se almacenan 16 bits (Total $2^{12} \times 16$ bits = 8 KB)
 - Palabras (unidad de transferencia) = 16 bits
- Máquina de Acumulador (el registro AC es el operando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

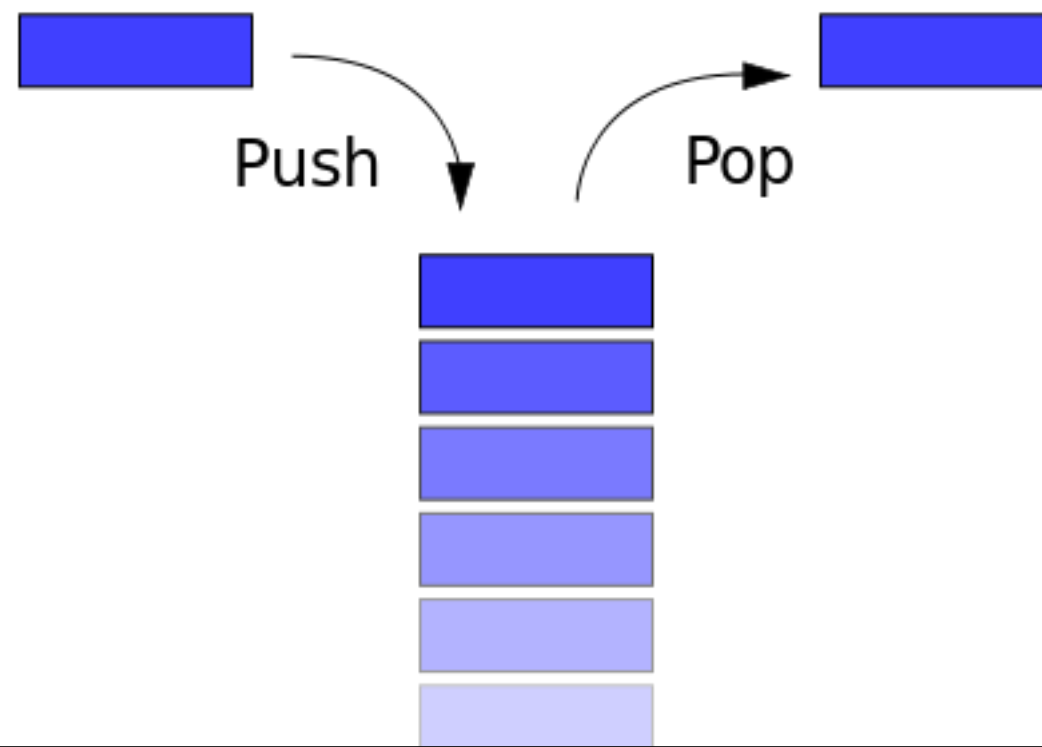
```
STORE D # [D] = ([A] - [B]) + [C]
```



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

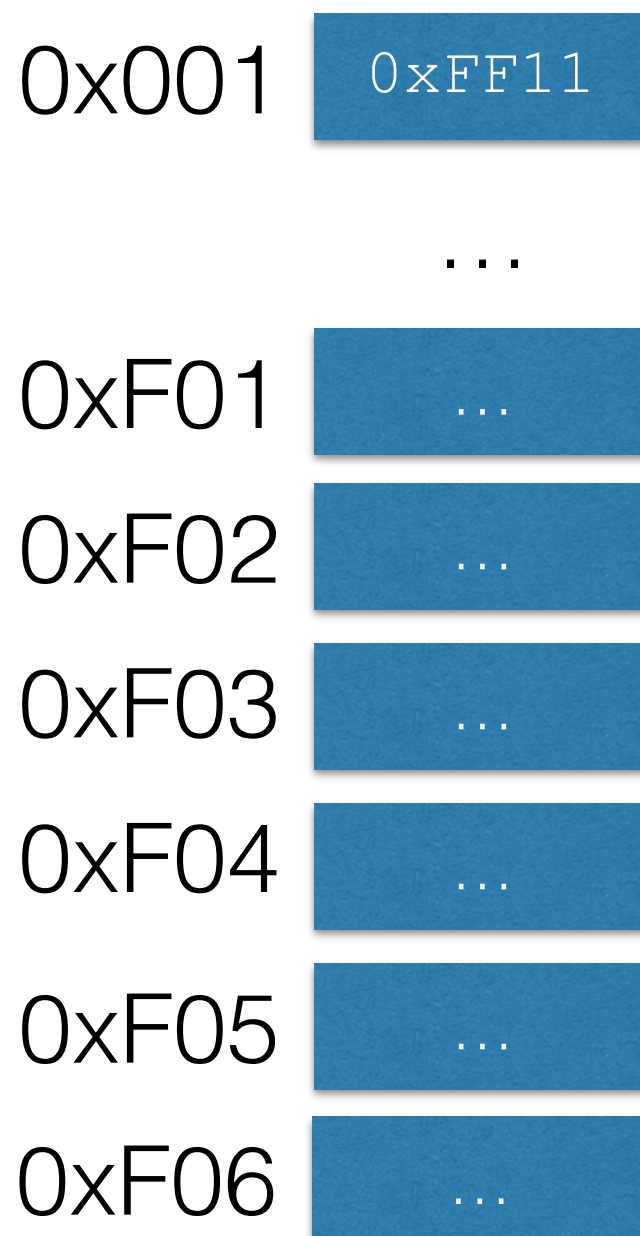
OpCode	Instrucción	Comportamiento
"0000"	PUSH X	push([X])
"0001"	POP X	[X]:=pop()
"0010"	ADD	push(pop()+pop())
"0011"	SUB	push(pop()-pop())
"0100"	AND	push(pop()&pop())
"0101"	OR	push(pop() pop())
"0110"	NOT	push(~pop())
"0111"	LE	push(pop()<=pop())
"1000"	GE	push(pop()>=pop())
"1001"	EQ	push(pop()==pop())
"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

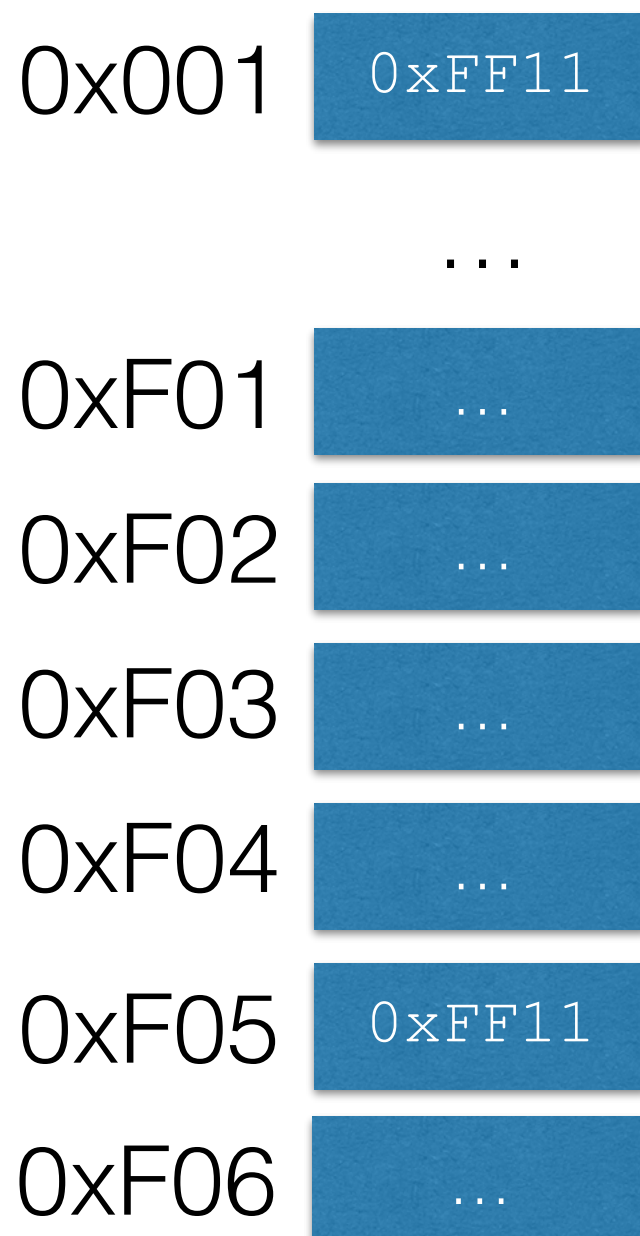
- PUSH X



- 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

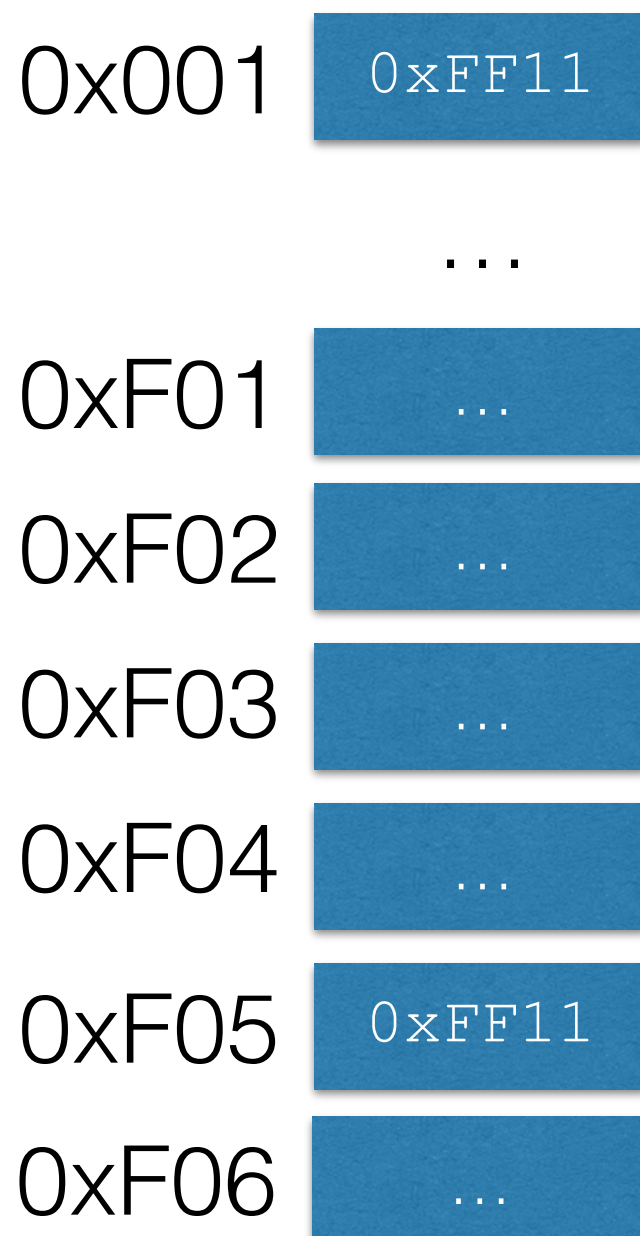
- PUSH X



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

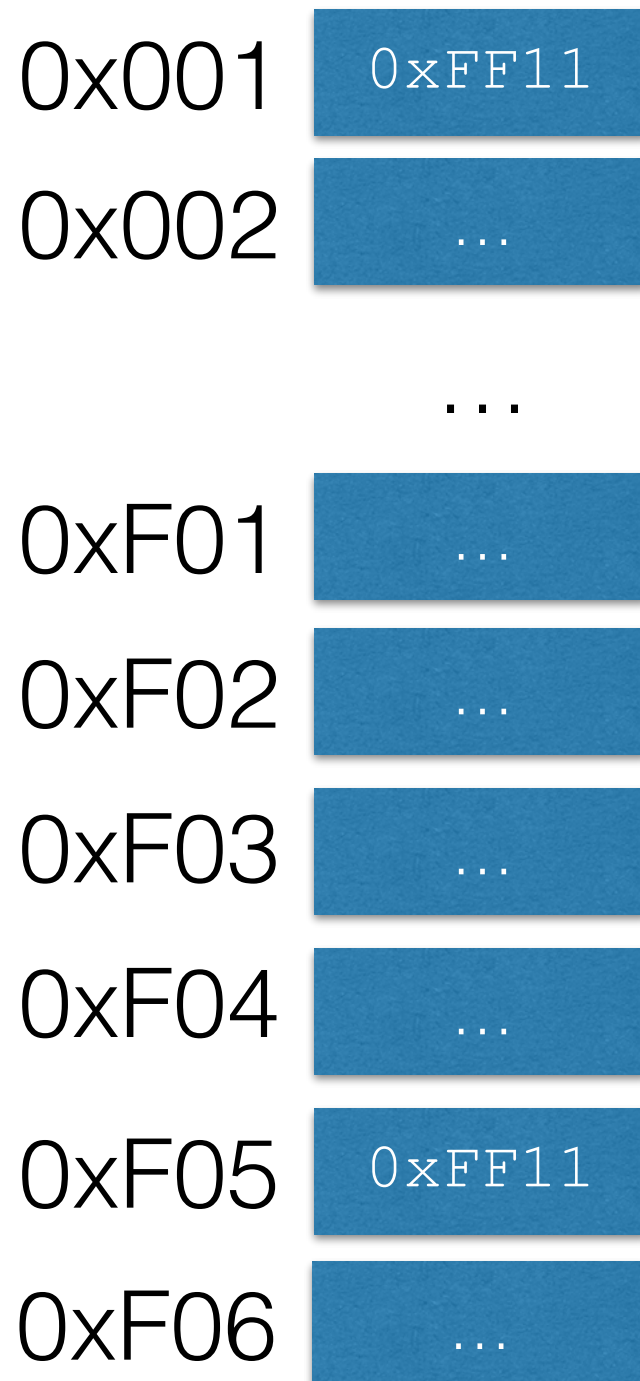


1. Lee el contenido de la dirección X
2. Copia el dato leído a la dirección apuntada por el registro SP (Stack Pointer)
3. **Decrementa SP**

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

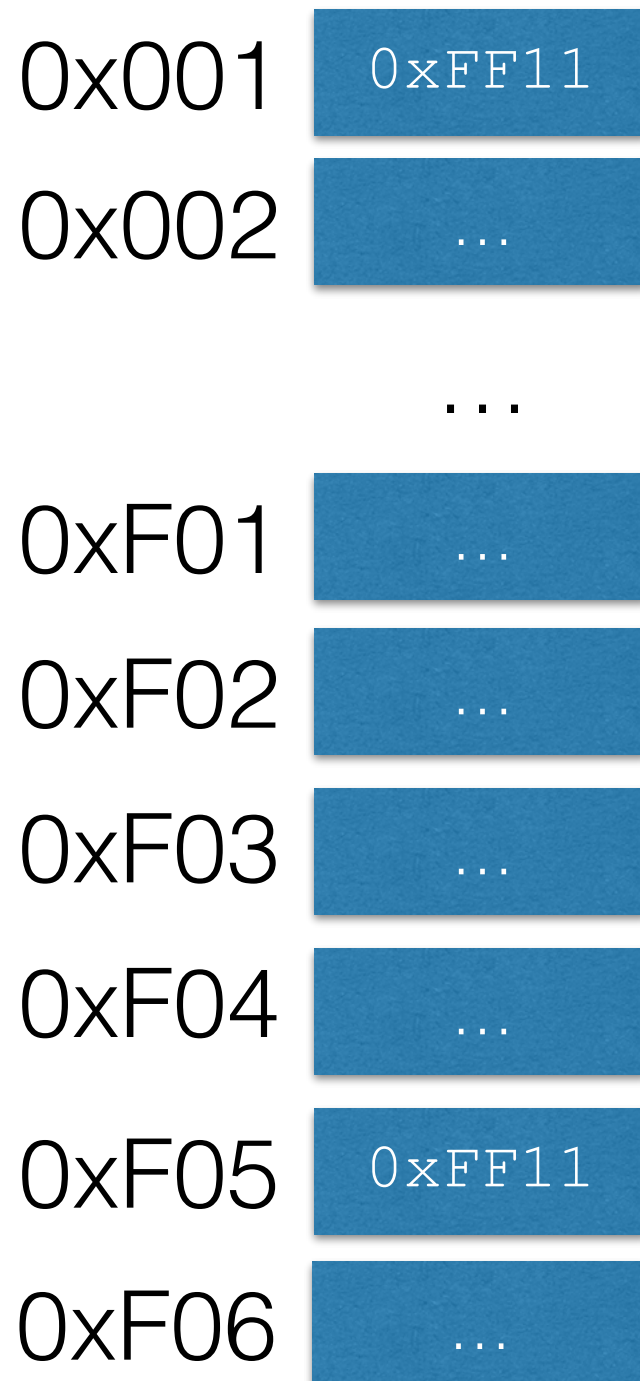
POP 0x002



- 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



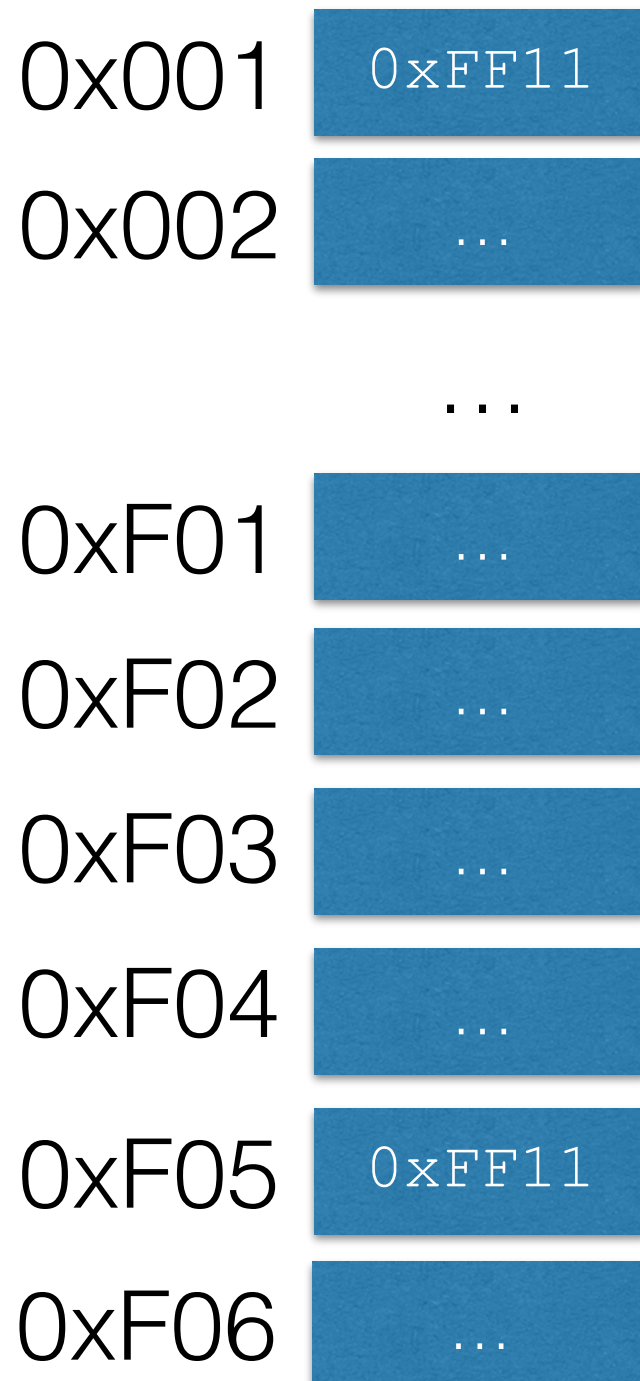
- POP X

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



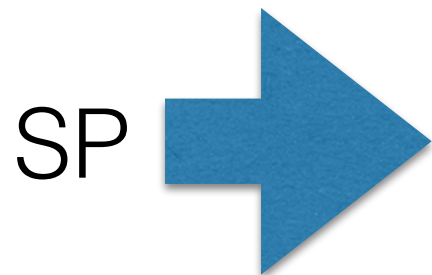
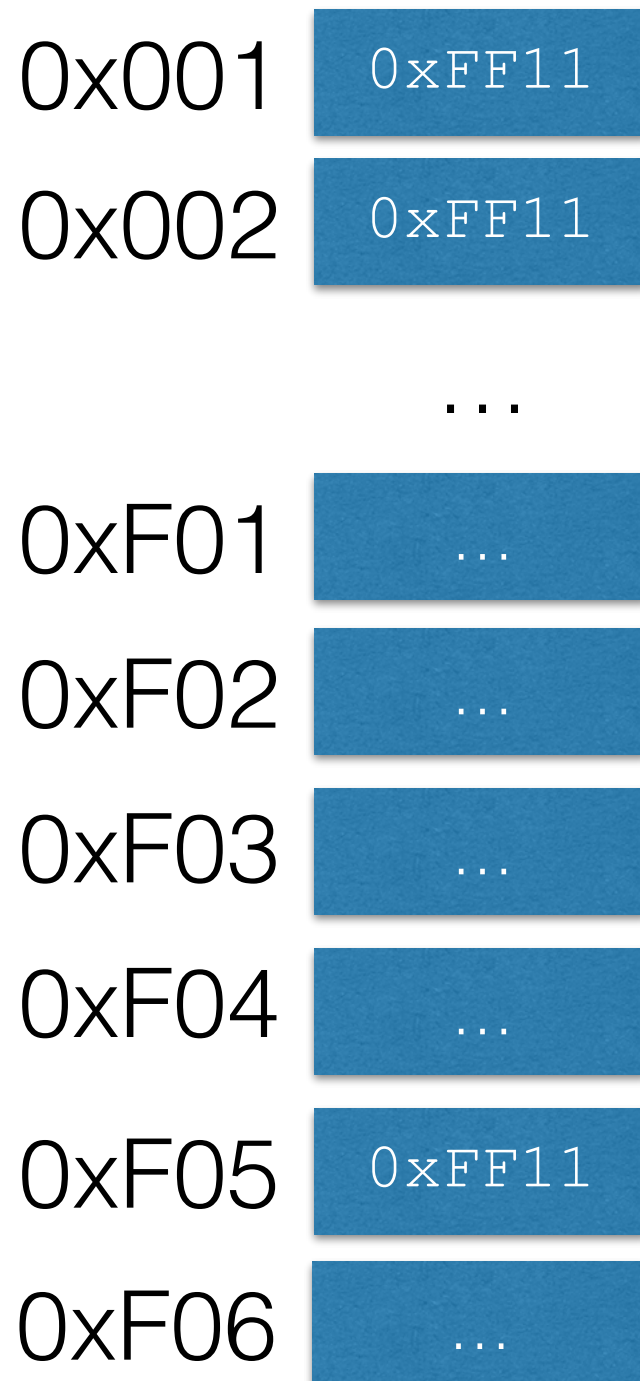
- POP X

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] ) }
```

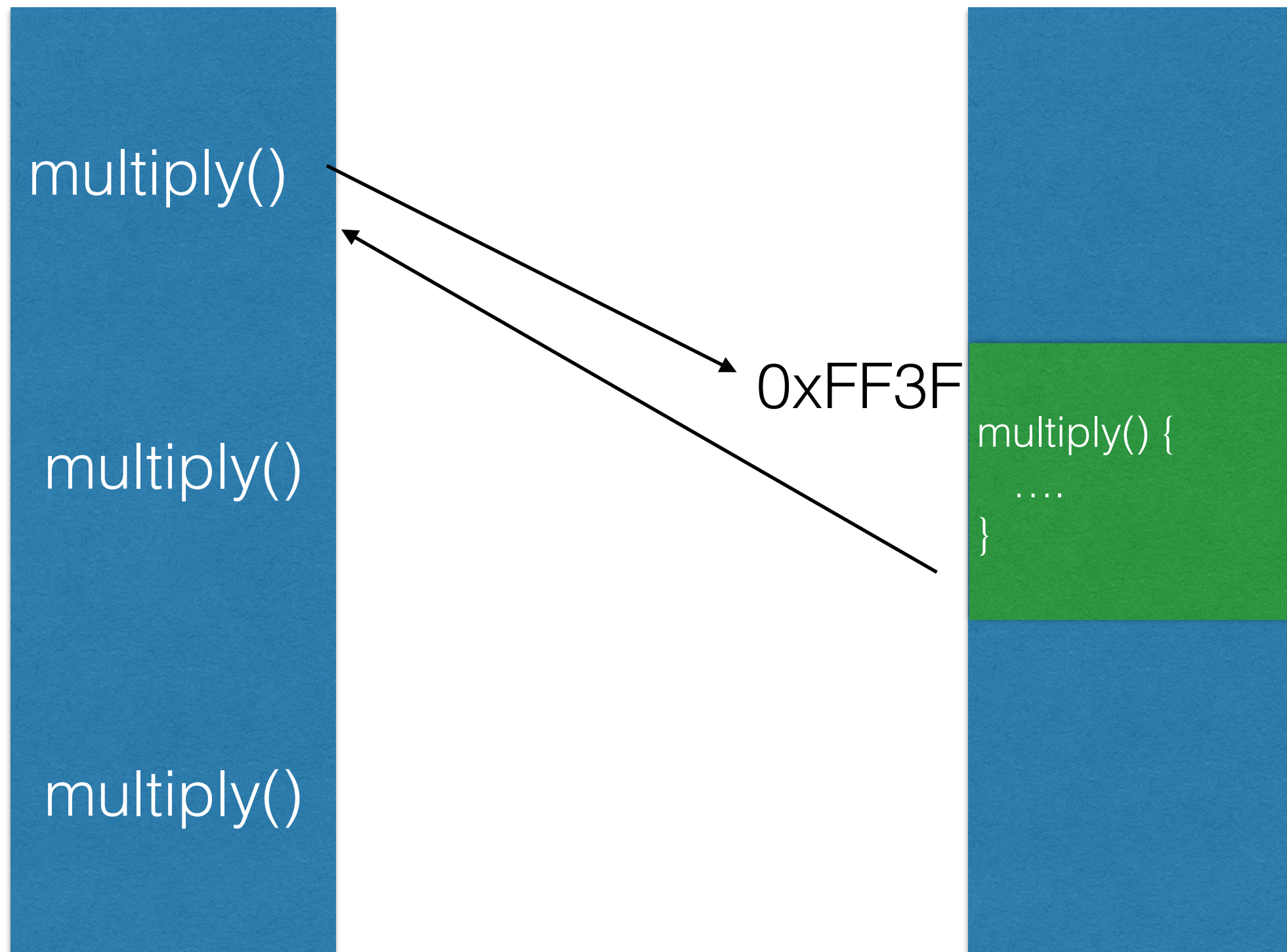
```
POP D # pila={}, [D] = [C] + ( [A] - [B] )
```

Subrutinas

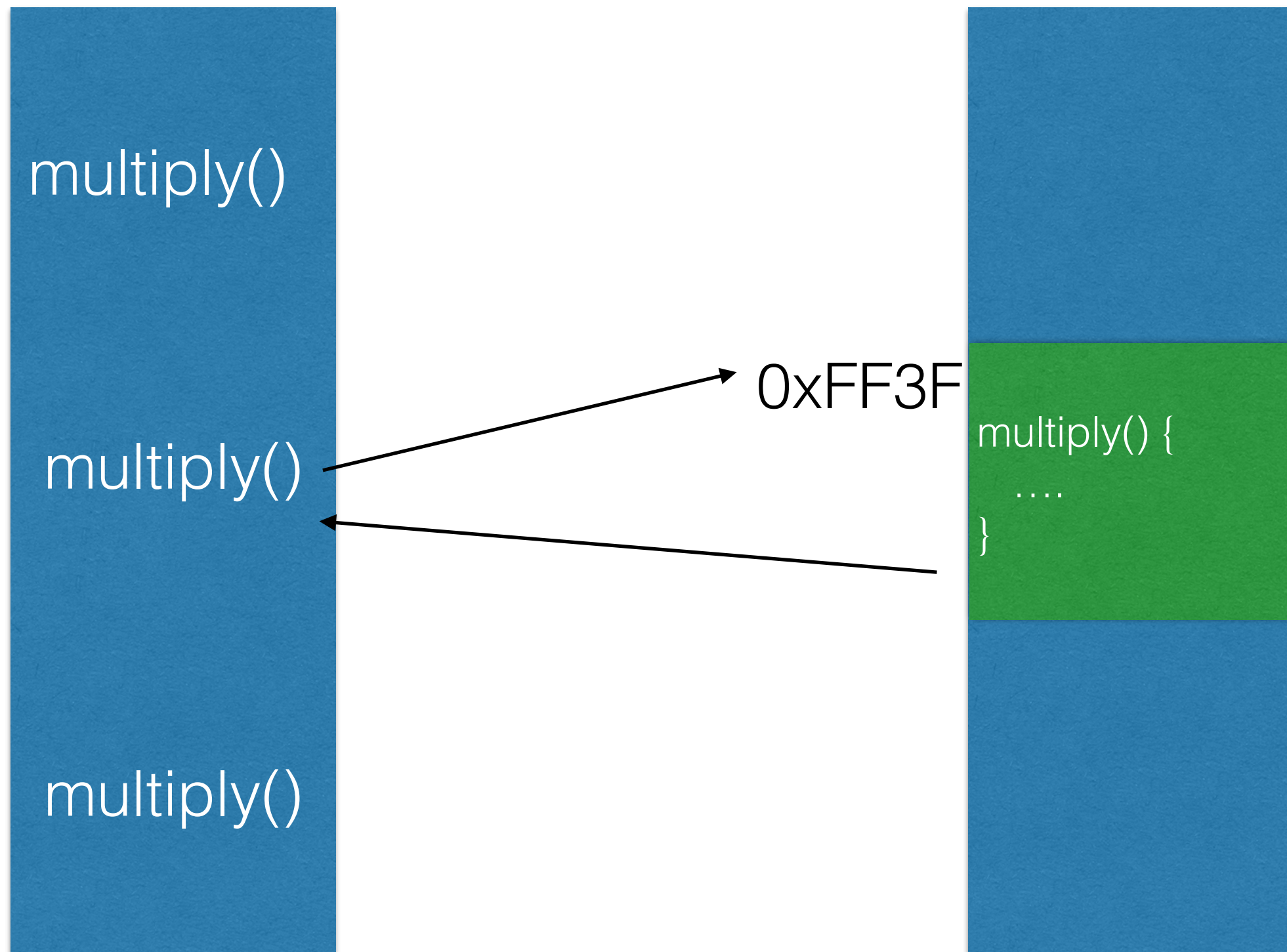
Subrutinas

- 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.

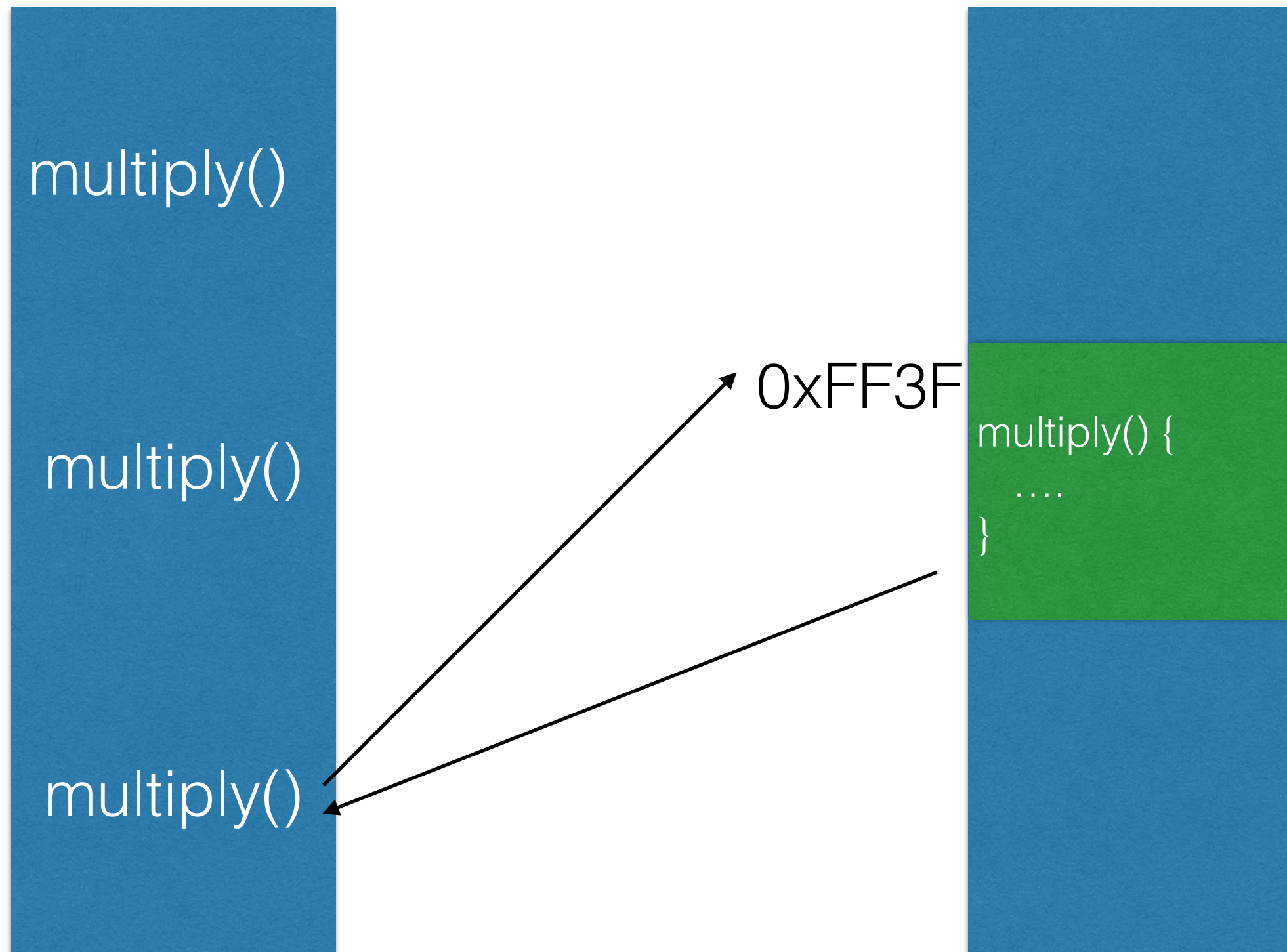
Subroutines



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

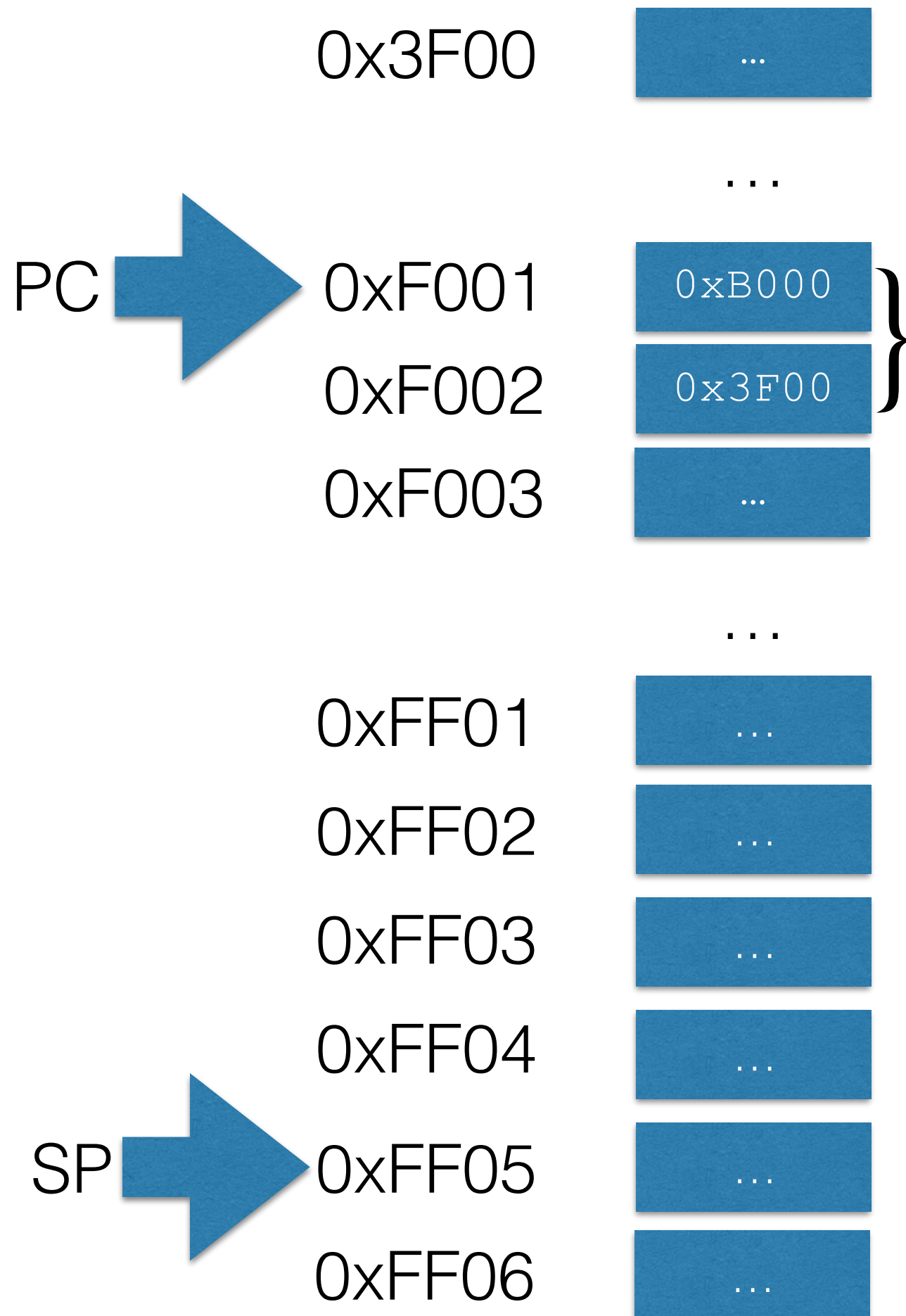
- Para almacenar la dirección de retorno podemos usar:
 - Un registro
 - Una dirección al ppio. de la subrutina (JnS - Jump and Store)
- **Problema:** estas soluciones no permiten recursión (¿Cómo podemos lograr recursión?)

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 fetchear será del programa llamador

Subrutinas en la Arquitectura Orga1

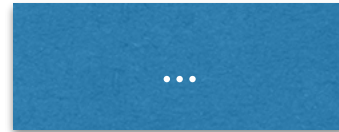
- CALL X
 - X es la dirección del inicio de la subrutina a ejecutar
 - Apila la dirección de retorno (decrementa el SP) y modifica el PC con el valor X
- RET
 - Desapila el tope de la pila (incrementa el SP), y modifica el PC con el tope de la pila



- `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

0x3F00



...

0xF001



0xF002



0xF003



...

0xFF01



0xFF02



0xFF03



0xFF04



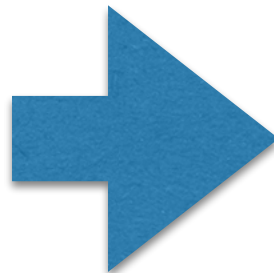
0xFF05



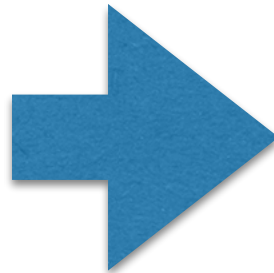
0xFF06



PC



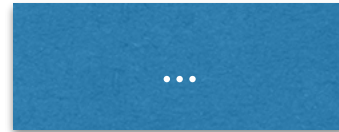
SP



- 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

0x3F00



...

0xF001



0xF002



0xF003



...

0xFF01



0xFF02



0xFF03



0xFF04



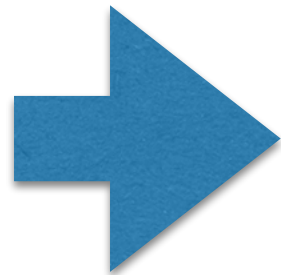
0xFF05



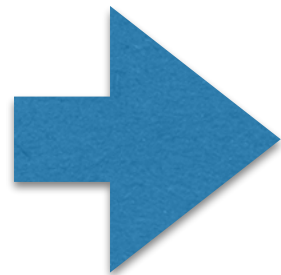
0xFF06



PC



SP



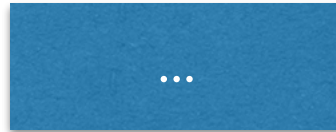
- CALL 0x3F00

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

2. Decrementa SP

3. Modifica el PC con el valor 0x3F00

0x3F00



...

0xF001



0xF002



0xF003



...

0xFF01



0xFF02



0xFF03



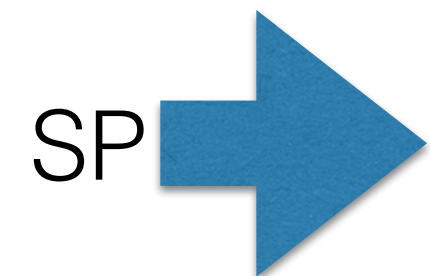
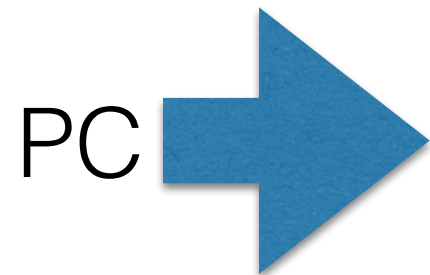
0xFF04



0xFF05



0xFF06

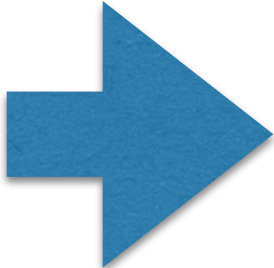


- `CALL 0x3F00`

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

...

...

0xF001

0xB000

0xF002

0x3F00

0xF003

...

...

0xFF01

...

0xFF02

...

0xFF03

...

SP  0xFF04

...

0xFF05

0xF003

0xFF06

...

- 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**



...

0xF001

0xB000

0xF002

0x3F00

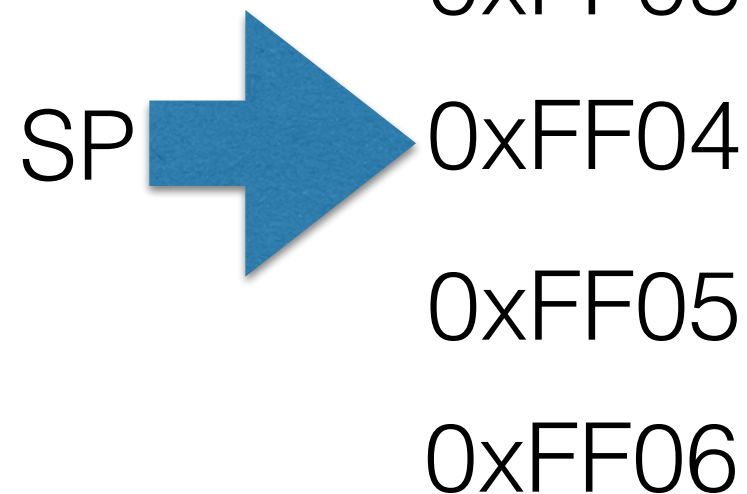
0xF003

...

• RET

1. Incrementa SP

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



...

0xFF01

...

0xFF02

...

0xFF03

...

0xFF04

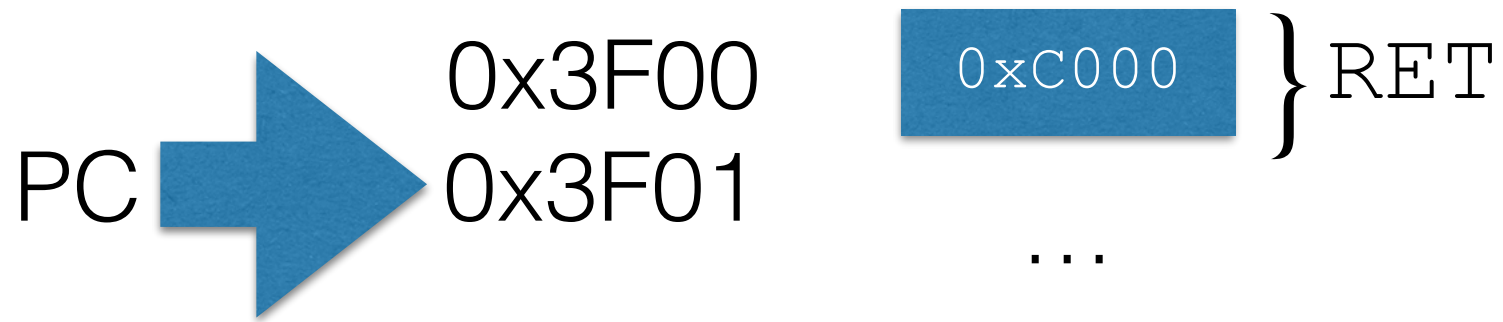
...

0xFF05

0xF003

0xFF06

...



0xF001

0xB000

0xF002

0x3F00

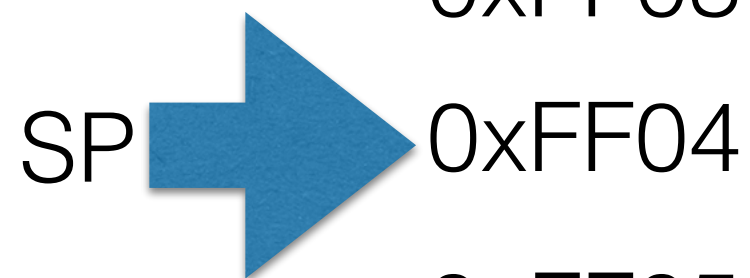
0xF003

...

• RET

1. Incrementa SP

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

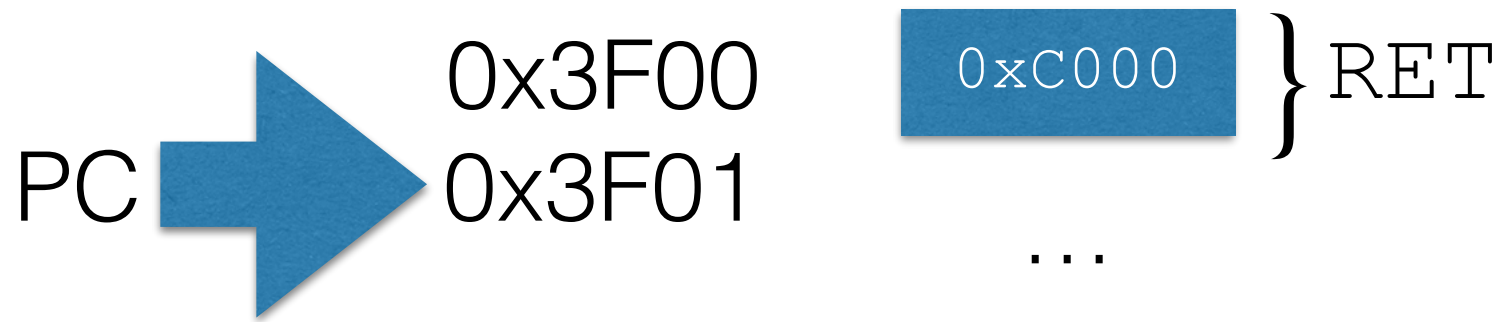


0xFF05

0xF003

0xFF06

...



0xF001

0xB000

0xF002

0x3F00

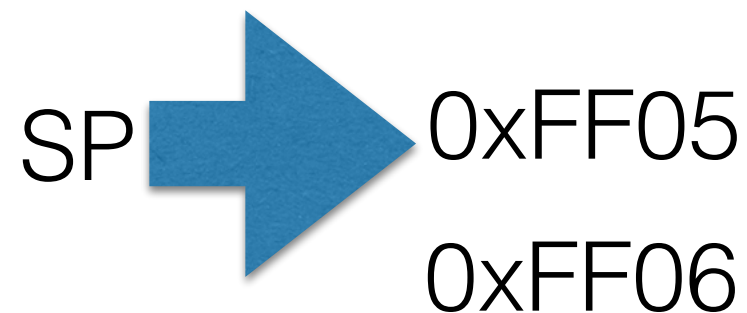
0xF003

...

• RET

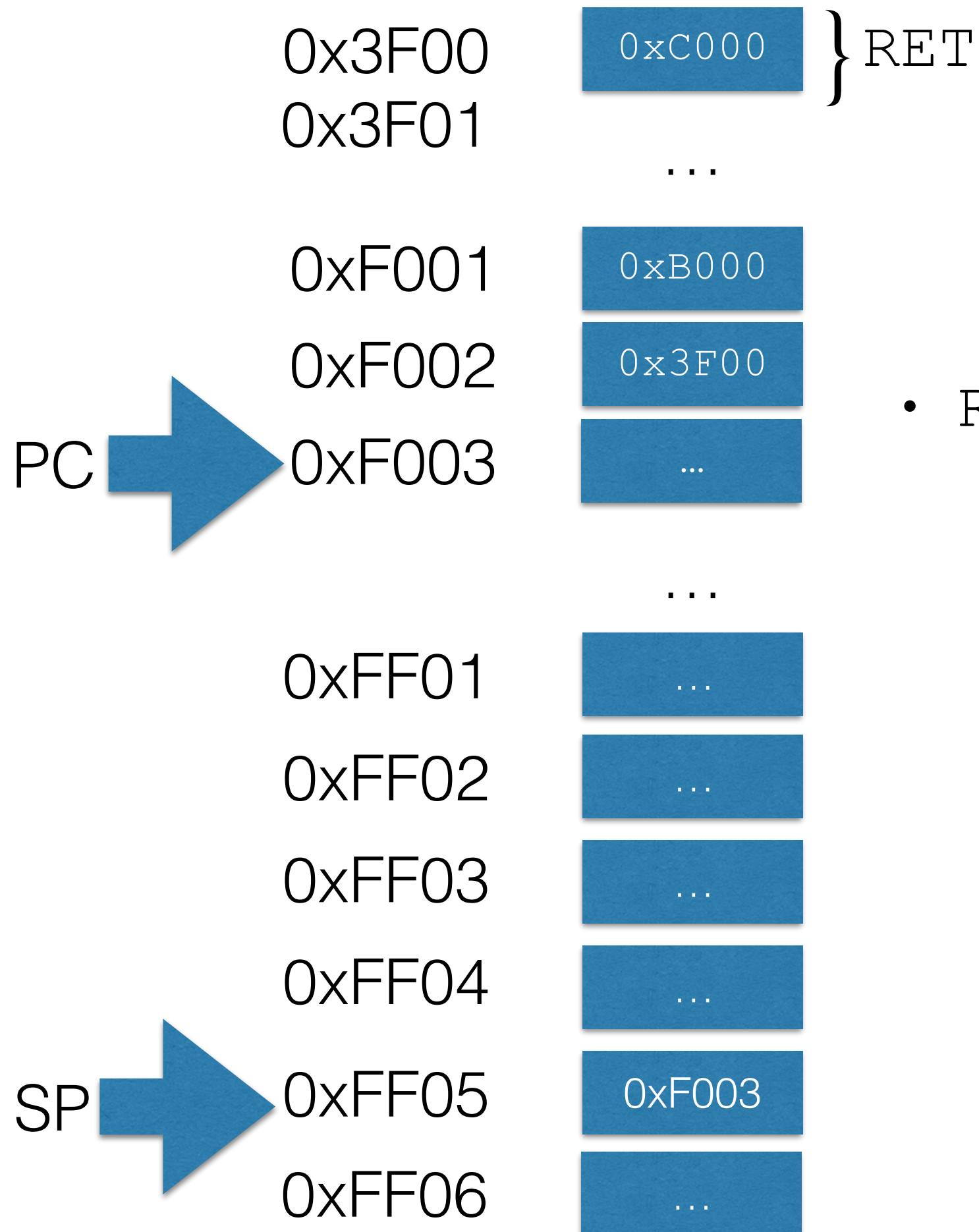
1. Incrementa SP

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



0xF003

...



• RET

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)

Diseño de una Arquitectura (ISA)

Diseño de ISA

- Tenemos que considerar:
 - Tipos de operaciones
 - Números de bits por instrucción
 - Número de operandos por instrucción
 - Operandos implícitos y explícitos
 - Ubicación de los campos
 - Tamaño y tipo de los operandos
 - Uso de stack, registros, etc.
 - Cómo almacenar los datos

Algunas métricas

- Memoria principal ocupada por el programa
- Tamaño de la instrucción (en bits)
- Code density: tratar de que las instrucciones ocupen lo menos posible
- Complejidad de la instrucción
- Número total de instrucciones disponibles

Algunas preguntas a responder

- Tamaño de la instrucción: ¿corto? ¿largo? ¿variable?
- ¿Cuántos bits para el OpCode?
 - Limita la cantidad de instrucciones
- ¿Cuántos bits para acceder a los operandos?
 - Limita la cantidad de memoria accesible
- ¿Cuántos registros?
 - Más registros es más cómodo para el programador, pero más costoso para la construcción

Algunas preguntas a responder

- Memoria:
 - ¿Direccionamiento por word o byte?
 - ¿Endianness?
 - ¿Cuántos/cuáles son los modos de direccionamiento?

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.
- 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).

Little Endian vs. Big Endian

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

- Little Endian:

34	12
n	n+1

- Big Endian

12	34
n	n+1

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

- Little Endian:

78	56	34	12
n	n+1	n+2	n+3

- Big Endian

12	34	56	78
n	n+1	n+2	n+3

Little Endian vs. Big Endian

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

- Little Endian:



- Big Endian



Tamaño Memoria

- Tamaño de la memoria=

Tamaño unidad direccionable x cant. direcciones

- Ejemplo:
 - Tamaño unidad direccionable = 1Byte
 - Cant. direcciones = $2^{16} = 65536$ direcciones
 - Tamaño Memoria = $65536 \times 1 \text{ B} = 65536 \text{ B} = \mathbf{64 \text{ KB}}$

Cantidad de Direcciones de Memoria

- Cantidad de direcciones =

$\text{Tam. de la Memoria} / \text{Tam. Unidad Direccionable}$

- Ejemplo:
 - Tamaño de la Memoria = 8 MB
 - Tamaño Unidad Direccionable = 2 B
 - Cant. de Direcciones = $8 \text{ MB} / 2 \text{ B} = 4 \text{ M}$
 $= 4 \times 1024 \times 1024 = \mathbf{4.194.304 \text{ direcciones}}$

Cantidad de bits para direcciones de Memoria

- Cantidad de bits para direcciones = $\log_2(\text{Cantidad de direcciones})$
- Ejemplo:
 - Cant. de Direcciones = 4.194.304
 - Cant. de bits para codificar direcciones = $\log_2(4.194.304) = \mathbf{22 \text{ bits}}$

Operandos

- 3 operandos: RISC y Mainframes
 - $A = B + C$
- 2 operandos: Intel, Motorola
 - $A = A + B$
 - (Al menos uno debe ser un registro)
- 1 operando: arquitecturas de acumulador
 - $AC = AC + A$ (+operando implícito: registro acumulador)
- 0 operandos: arquitecturas de pila
 - $\text{push}(\text{pop}() + \text{pop}())$ (+operando implícito: pila)

Algunas ISAs

	CISC			RISC		Superescalares	
	IBM 370/168	VAX 11/780	Intel 80486	88000	R4000	RS/6000	80960
Año	1973	1978	1989	1988	1991	1990	1989
Número de instrucciones	208	303	235	51	94	184	62
Tamaño de instrucción (bytes)	2 - 6	2 - 57	1 - 11	4	4	4	4, 8
Modos de direccionamiento	4	22	11	3	1	2	11
Número de GRPs	16	16	8	32	32	32	32 - 256
Tamaño mem de control (K bits)	420	480	246	-	-	-	-
Tamaño caché (KB)	64	64	8	16	128	32 - 64	0.5

Ortogonalidad

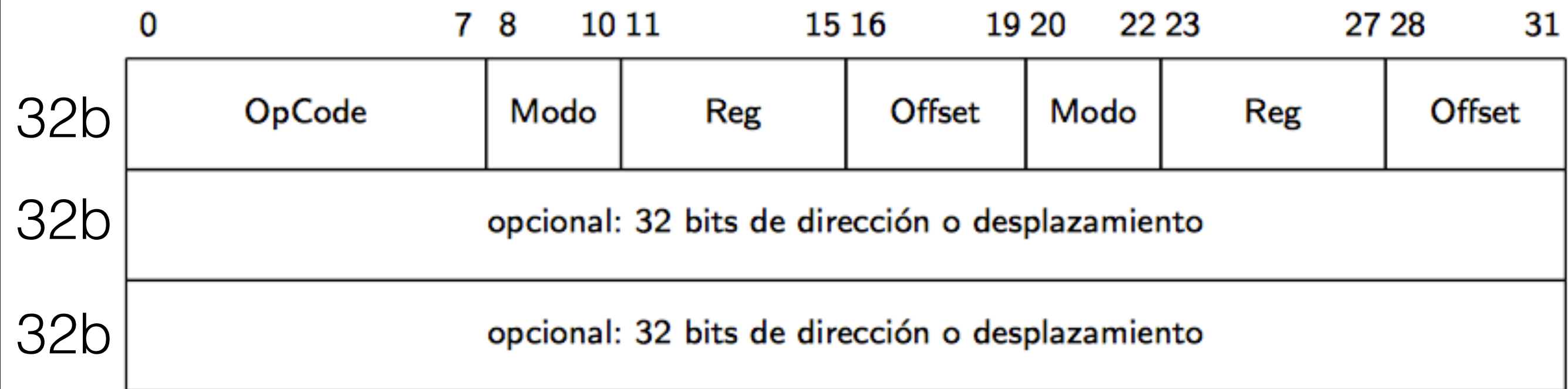
- **Máxima ortogonalidad:** cualquier instrucción puede ser usada con cualquier modo de direccionamiento
- Es una característica “elegante” pero muy costosa:
 - Implica tener muchas instrucciones
 - Algunas quizás poco usadas o fácilmente reemplazables

Ejemplo

- ¿Qué podemos deducir de este formato de instrucción?

0	7	8	10	11	15	16	19	20	22	23	27	28	31
OpCode		Modo	Reg		Offset		Modo	Reg		Offset			
opcional: 32 bits de dirección o desplazamiento													
opcional: 32 bits de dirección o desplazamiento													

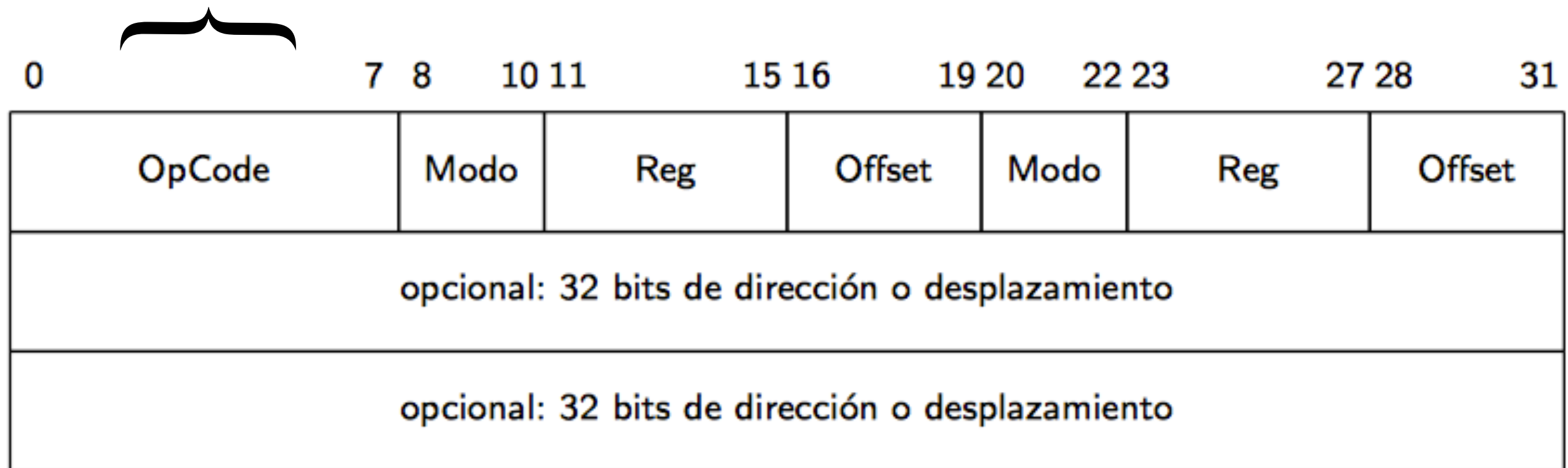
Ejemplo



- Instrucciones de longitud variable
 - Tamaño: 32 bits, 64bits o 96bits

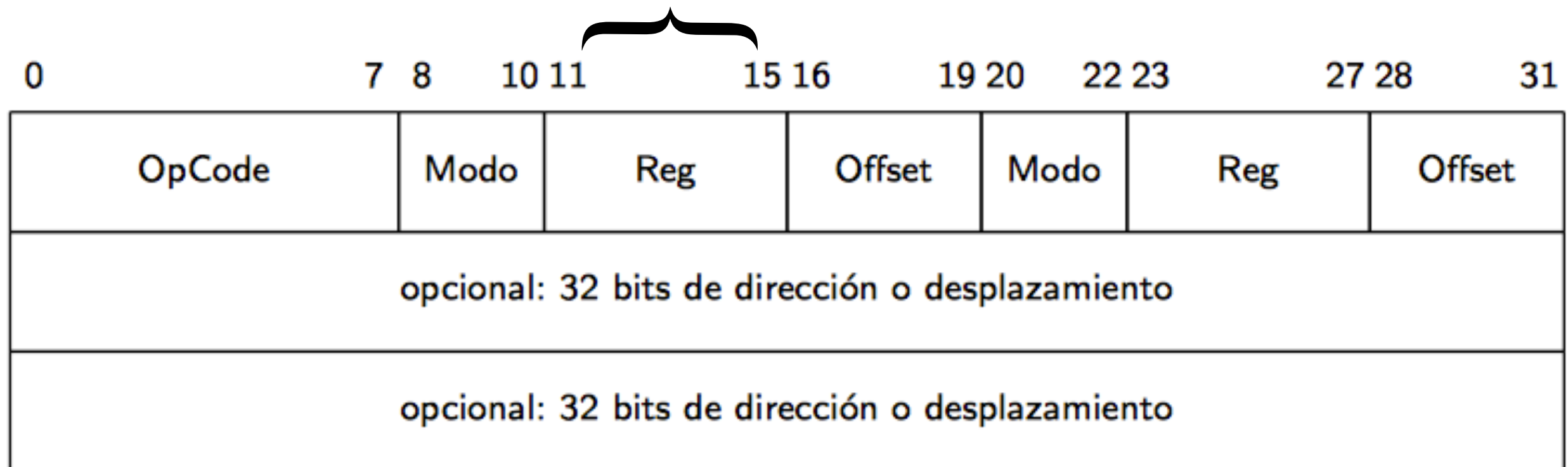
Ejemplo

8 bits para Opcode = 2^8 instrucciones = 256



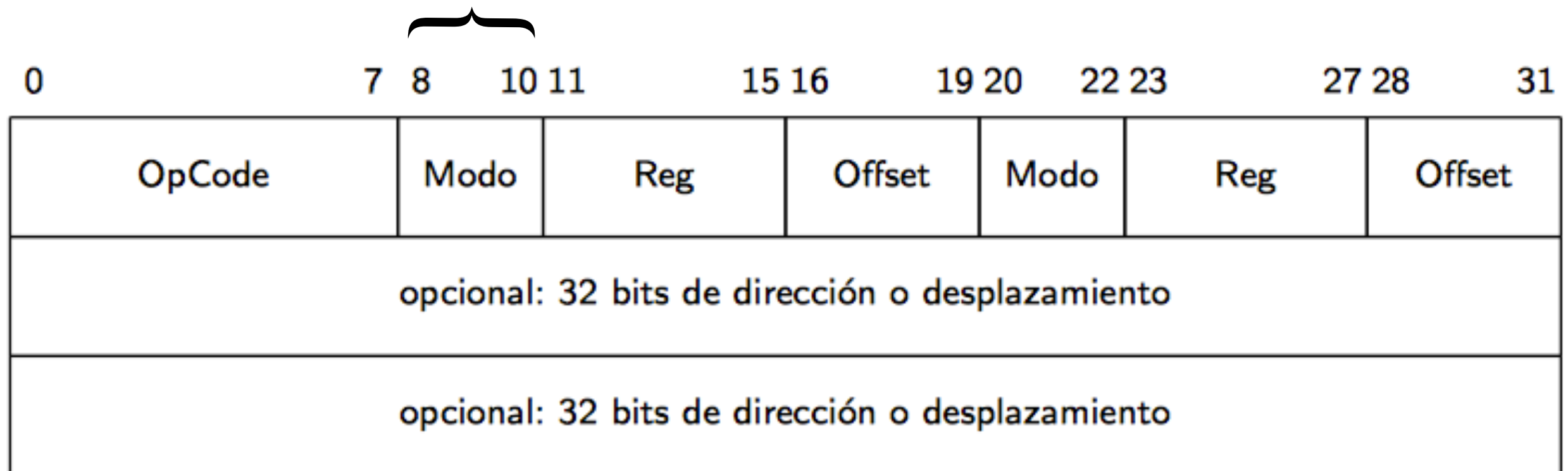
Ejemplo

5 bits para identificar registros
 $=2^5$ registros=32



Ejemplo

3 bits para identificar modos de direccionamiento
 $= 2^3$ modos de direc. = 8



Ejemplo

4 bits para indicar desplazamiento o escala



0	7	8	10	11	15	16	19	20	22	23	27	28	31
OpCode		Modo	Reg		Offset		Modo		Reg		Offset		
opcional: 32 bits de dirección o desplazamiento													
opcional: 32 bits de dirección o desplazamiento													

Ejemplo

0	7	8	10	11	15	16	19	20	22	23	27	28	31
OpCode	Modo	Reg	Offset	Modo	Reg	Offset							
opcional: 32 bits de dirección o desplazamiento													
opcional: 32 bits de dirección o desplazamiento													

32 bits para direcciones/datos/desplazamiento

- Es necesario leer una segunda palabra valores inmediatos/direcciones/etc.

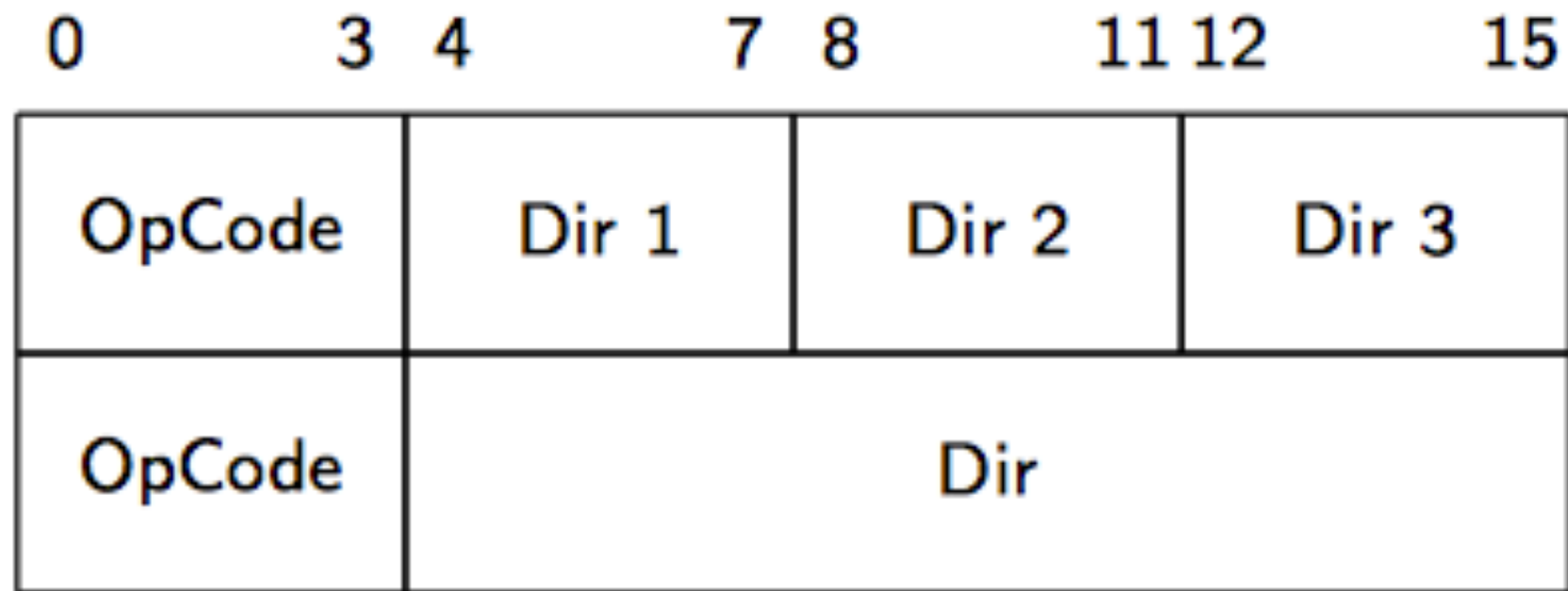
Formato de Instrucción

- El tamaño de las instrucciones depende fuertemente del número de operandos que soporta la Arquitectura
- No todas las instrucciones requieren el mismo número de operandos
 - Por ejemplo, la operación RET ni siquiera necesita operando, qué podemos hacer con el espacio que sobra
 - **Rta**: podríamos utilizar códigos de operación variables

Ejemplo

- Supongamos una arquitectura con 16 registros, 4K direcciones de memoria.
- Los registros se codifican con 4 bits ($2^4=16$)
- Las direcciones se codifican con 12 bits ($2^{12}=4K$)
- Si la longitud de instrucción es fija de 16 bits, ¿cuántas instrucciones puede tener la Arquitectura?

OpCode Fijo



- $2^4 = 16$ instrucciones:
- Algunas con 1 dirección de memoria de operando
- Otros con hasta 3 registros como operandos

OpCode Fijo

- Supongamos que se necesita únicamente 2 instrucciones con acceso a memoria

0000 0001	DIR			Tipo 1: 2 instrucciones de 1 dirección de memoria
0010 ... 1111	R1	R2	R3	Tipo 2: 14 instrucciones de 3 registros

- Si el OpCode es fijo de 4 bits, quedan únicamente 14 instrucciones de a lo sumo 3 registros.
- ¿Qué pasa si el OpCode es **variable**?

OpCode Variable

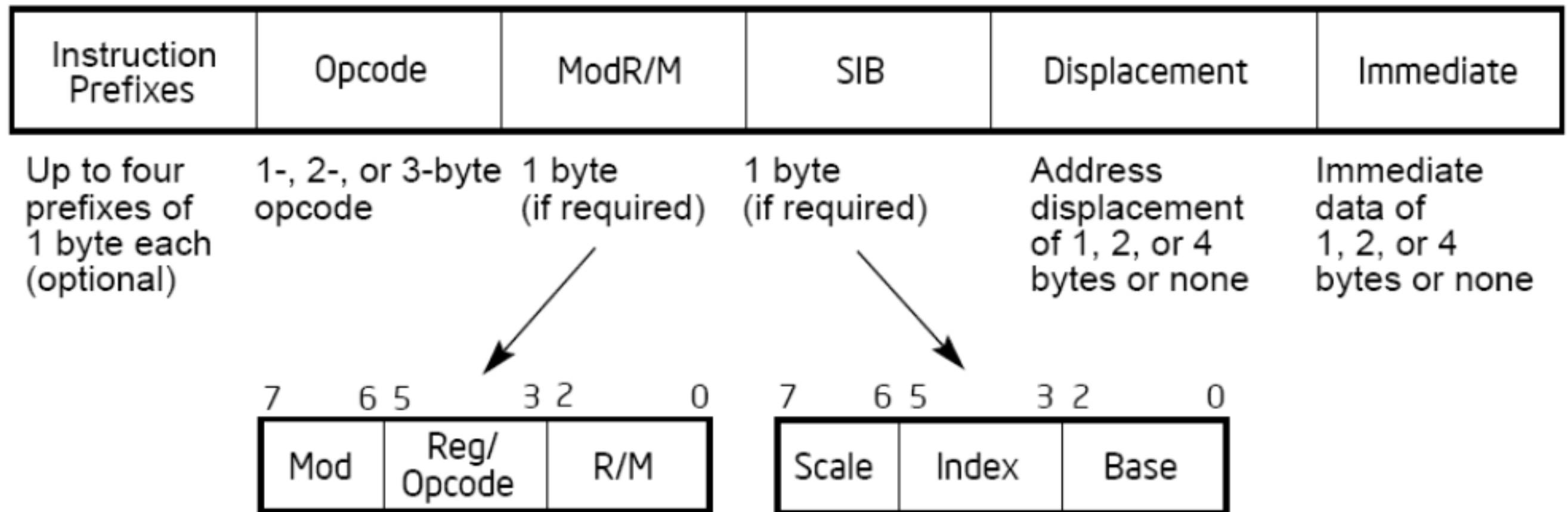
0000 0001	DIR			Tipo 1: 2 instrucciones de 1 dirección de memoria
0010 ... 1110	R1	R2	R3	Tipo 2: 13 instrucciones de 3 registros
1111	0000 1110	R1	R2	Tipo 3: 15 instrucciones de 2 registros
1111	1111	0000 ... 1110	R1	Tipo 4: 15 instrucciones de 1 registro
1111	1111	1111	0000 ... 1111	Tipo 5: 16 instrucciones de 0 operandos

OpCode Fijo vs. OpCode Variable

0	3	4	7	8	11	12	15
OpCode	Dir 1		Dir 2		Dir 3		
OpCode	Dir						

- El OpCode Variable nos permite un mayor aprovechamiento del espacio de codificaciones
- OpCode Fijo = 16 instrucciones
- OpCode Variable = 61 instrucciones

Formato de Instrucción Pentium



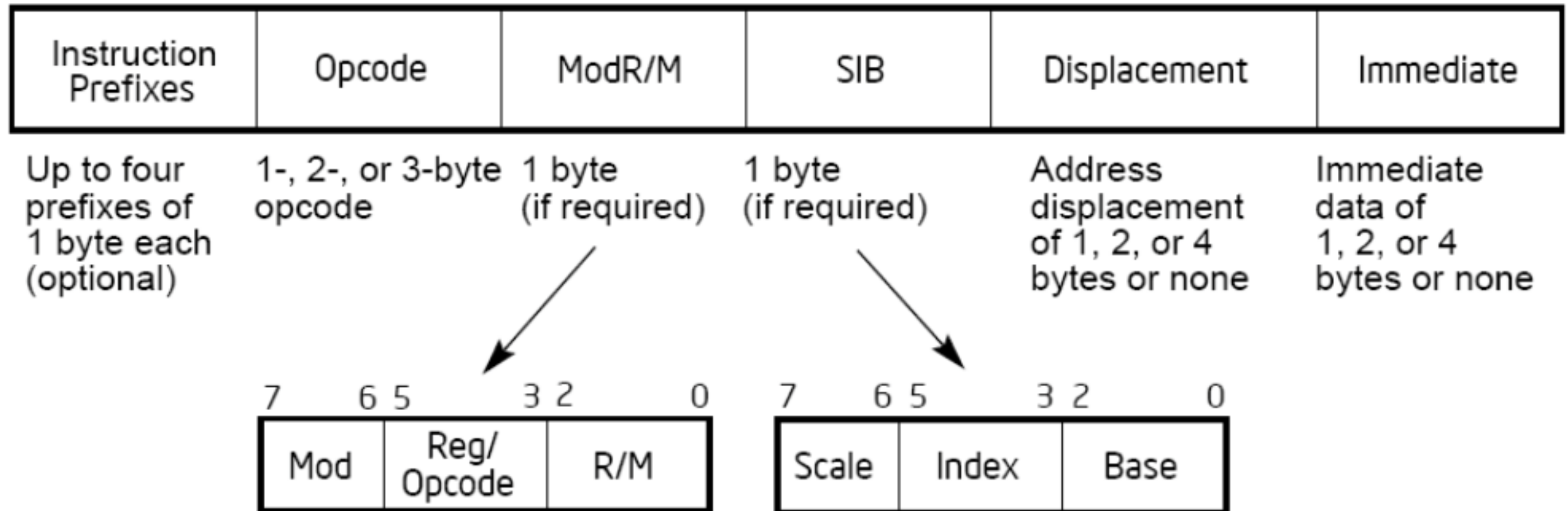
Modos de direccionamiento Pentium: **24!**

Formato de Instrucción Pentium

REP MOVSB # Copia "CX" bytes de DS:SI hacia ES:DI

0xF3

0xA4



Resumen

- Máquina de Stack
- Subrutinas
- Diseño de ISA:
 - Ortogonalidad, Formato de Instrucción
 - OpCode Fijo vs. OpCode Variable

Bibliografía

- Tanenbaum - Capítulo 5
- Stalling - Capítulo 11
- Null - Capítulo 5