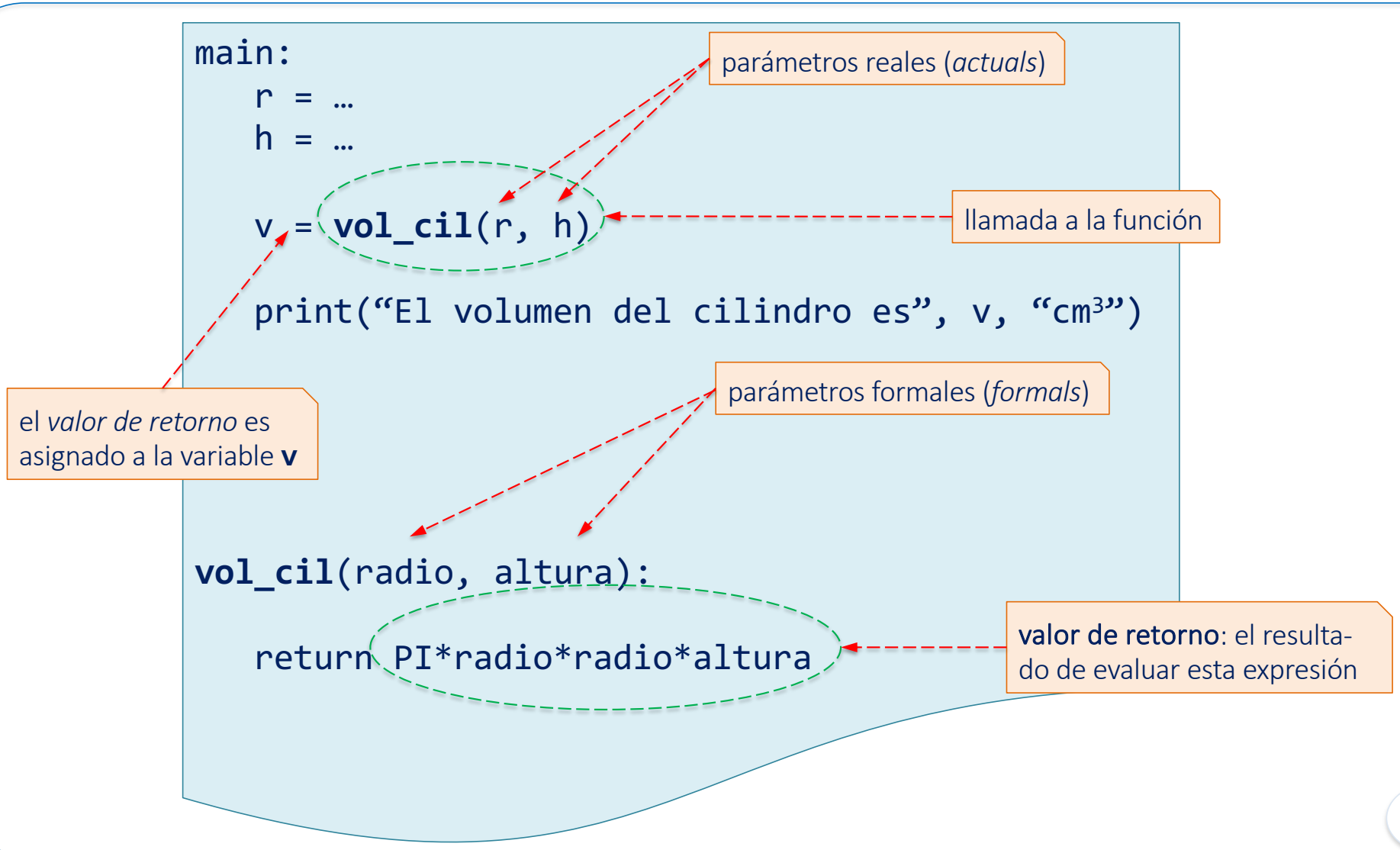
Subrutinas

(también *funciones*, *métodos*, *procedimientos*)

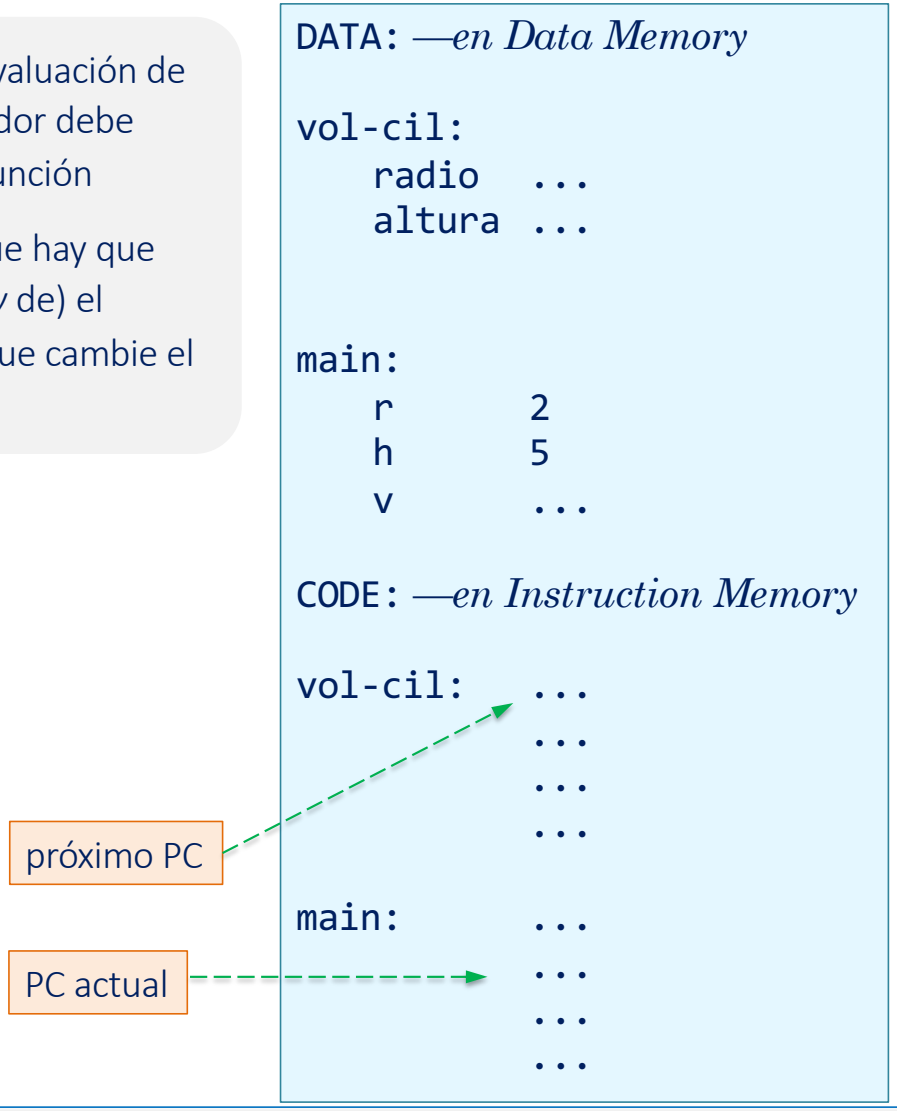
```
main:           —programa principal
    r = …       —se asigna un valor a esta variable
    h = …       —se asigna un valor a esta variable
    v = vol_cil(r, h)
    print("El volumen del cilindro es", v, "cm3")

vol_cil(radio, altura):        —función o subrutina
    return PI*radio*radio*altura
```


main:
r = …
h = …
v = vol_cil(r, h)
print("El volumen del cilindro es", v, "cm3")
vol_cil(radio, altura):
return PI*radio*radio*altura
parámetros reales (actuals)
llamada a la función
el valor de retorno es asignado a la variable v
parámetros formales (formals)
valor de retorno: el resultado de evaluar esta expresión

1) Al producirse la llamada a la función —la evaluación de la expresión **`vol_cil(r, h)`**— el procesador debe empezar a ejecutar las instrucciones de la función

⟹ necesitamos una nueva instrucción —que hay que agregar a (la versión en lenguaje *assembly* de) el **`main`**— similar a un salto incondicional, que cambie el valor del registro *PC*



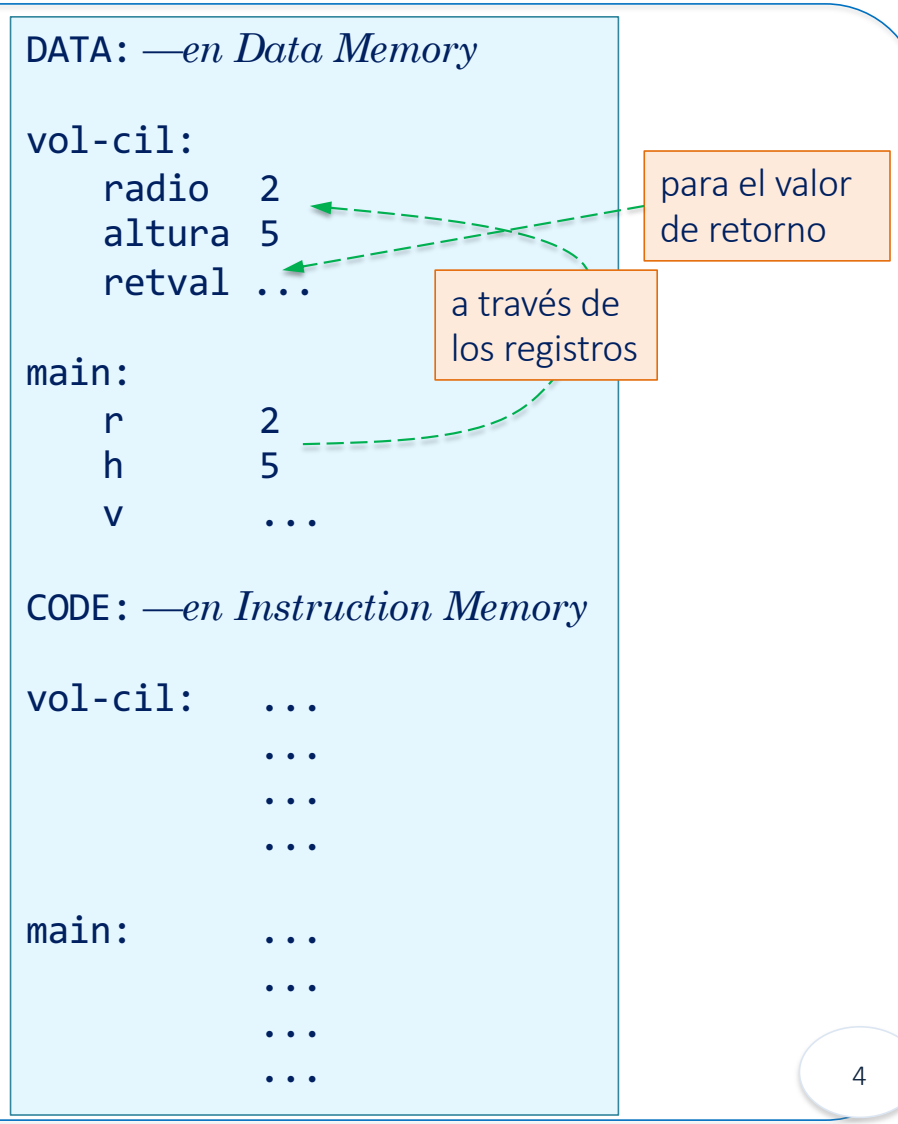
2) Sólo que justo antes, es necesario "pasarle" a la función **vol_cil** los valores que deben tomar los parámetros formales **radio** y **altura**

... es decir, los valores que en ese momento tienen las variables **r** y **h** (los parámetros reales)

⟹ hay que copiar los valores de los parámetros reales en las variables de la *Data Memory* que representan los parámetros formales (variables **radio** y **altura** a las que la función tiene acceso)

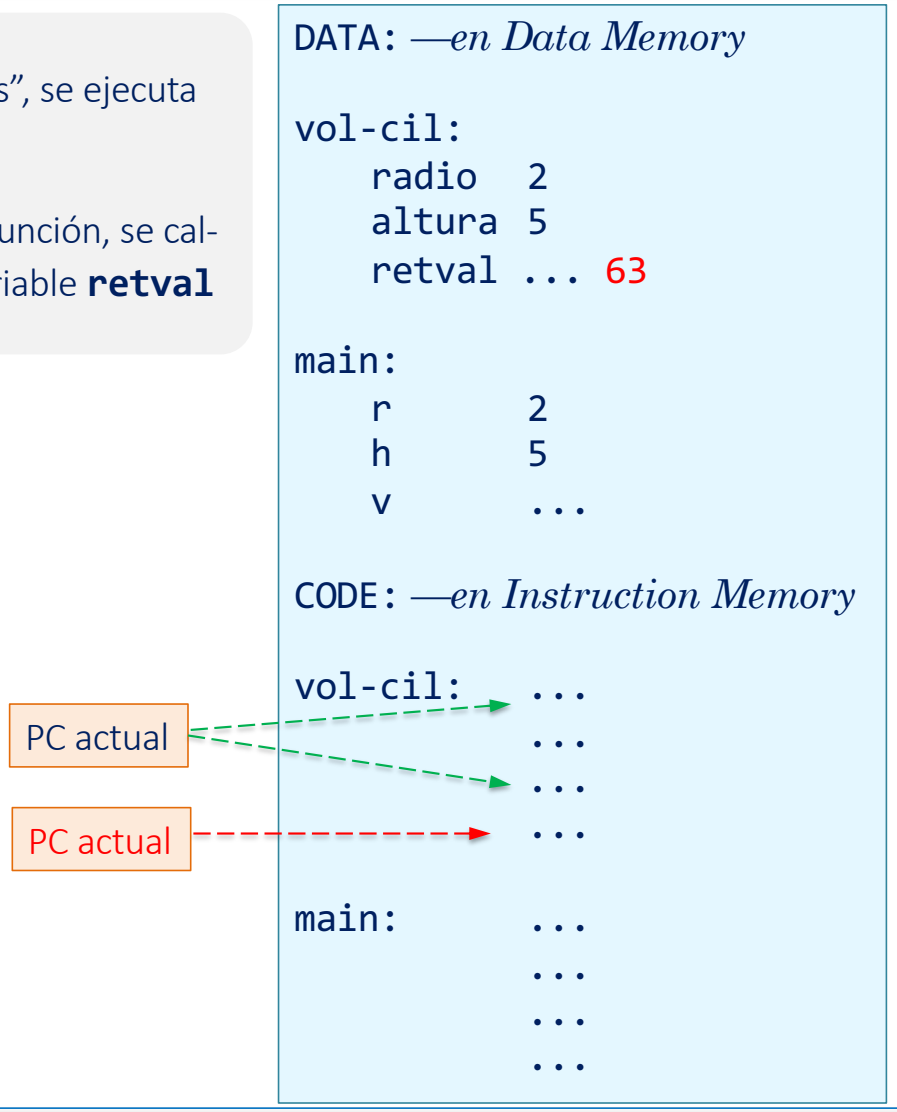
... mediante instrucciones adicionales en el **main**

Además, como la función "retorna" un valor, hay que reservar espacio en la *Data Memory* para este "valor de retorno": **retval**



3) Una vez que los parámetros han sido “pasados”, se ejecuta el código correspondiente a la función

… y justo antes de terminar la ejecución de la función, se calcula el valor de retorno y se almacena en la variable **retval**



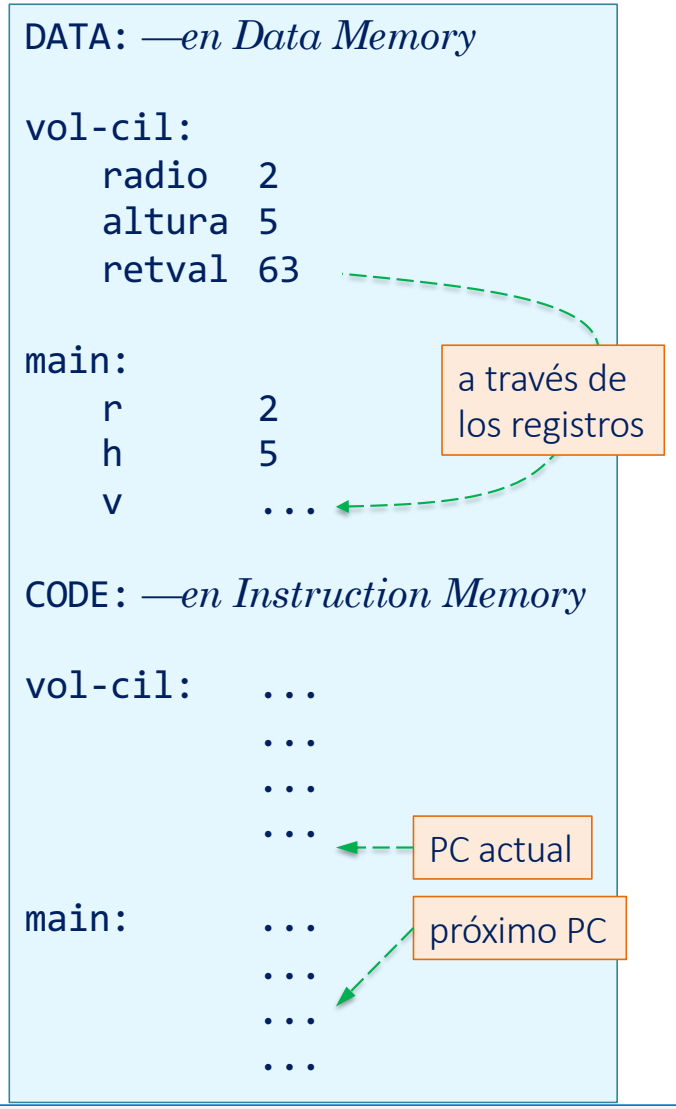
4) Finalmente, al terminar la ejecución de la función, hay que "pasar de vuelta", o "retornar", el valor calculado por la función:

- usando nuevamente la *Data Memory*

... y reanudar la ejecución del programa **`main`** en el punto en que fue suspendida:

- retomando el valor original del registro *PC* más 1
- $\Longrightarrow$ este valor debió haber quedado guardado en alguna parte antes de que se empezara a ejecutar la función

$\Longrightarrow$ **necesitamos una nueva instrucción** —que hay que agregar a (la versión en lenguaje *assembly* de) la función



En las siguientes diaps., vamos a construir de a poco una solución (a nivel del lenguaje *assembly*) al problema de llamar funciones con parámetros, cuando hay llamadas anidadas:

- veremos primero el caso de una única llamada a una función
- luego, el caso de una llamada a una función que a su vez llama a otra función
- finalmente, cómo manejar los registros *A* y *B* cuando sus valores antes de la llamada a una función deben seguir disponibles al volver de la llamada

Ahora vamos a construir la solución en el caso de nuestro computador básico … más adelante, vamos a ver cómo se resuelve en RISC-V

Más funcionalidad en el computador básico

$\Longrightarrow$ componentes adicionales:

- tanto de hardware (nuevas componentes físicas y conexiones)

  ... como de software (nuevas instrucciones)

Para poder tener variables:

$\Longrightarrow$ memoria de datos (*Data Memory*) + nuevas conexiones + nuevas instrucciones

diaps. 15 a 28 de "clase-6"

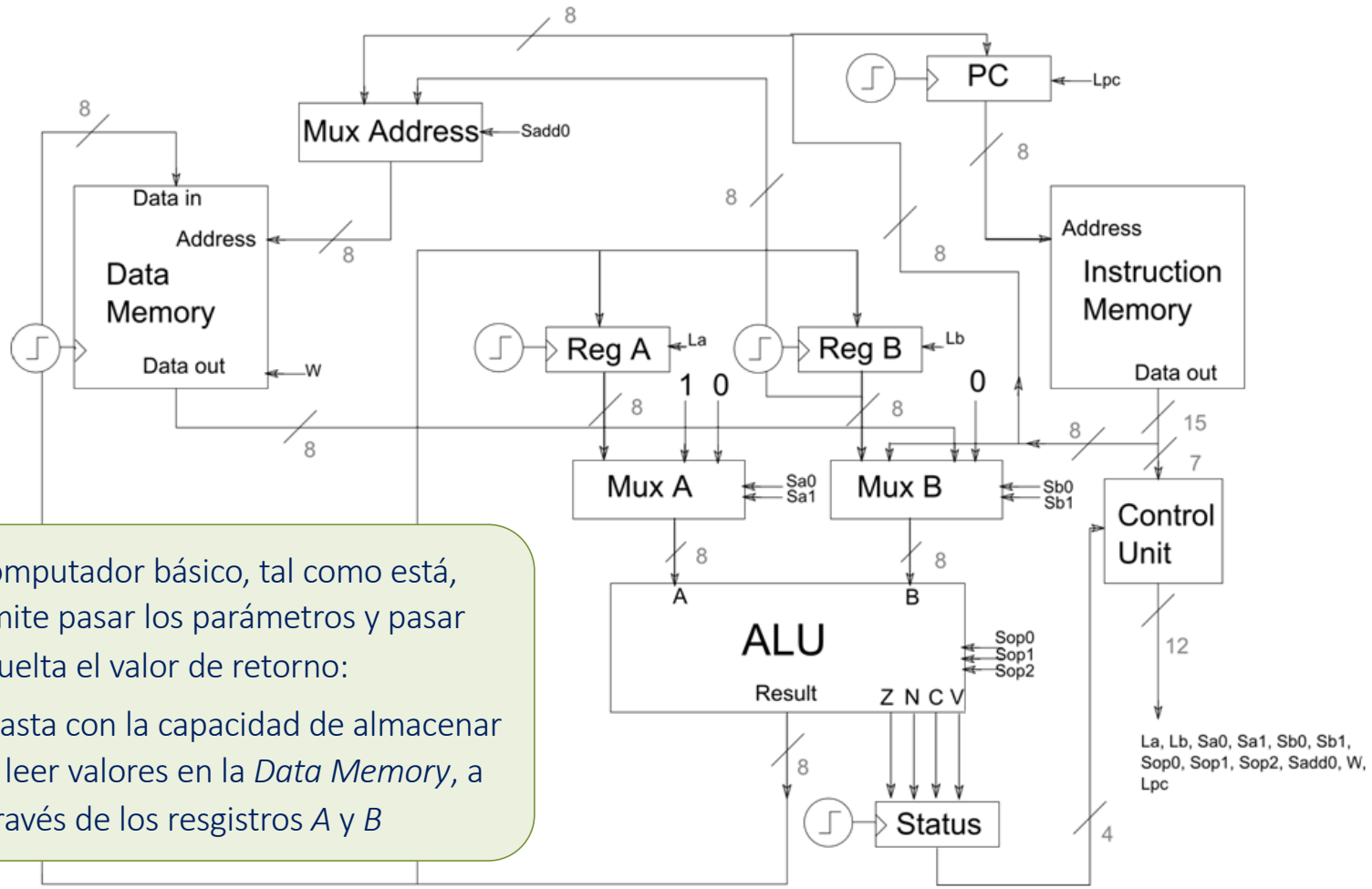
Para poder tener sentencias `if` y `while`:

$\Longrightarrow$ registro *Status* + nuevas conexiones + nuevas instrucciones

diaps. 29 a 41 de "clase-6"

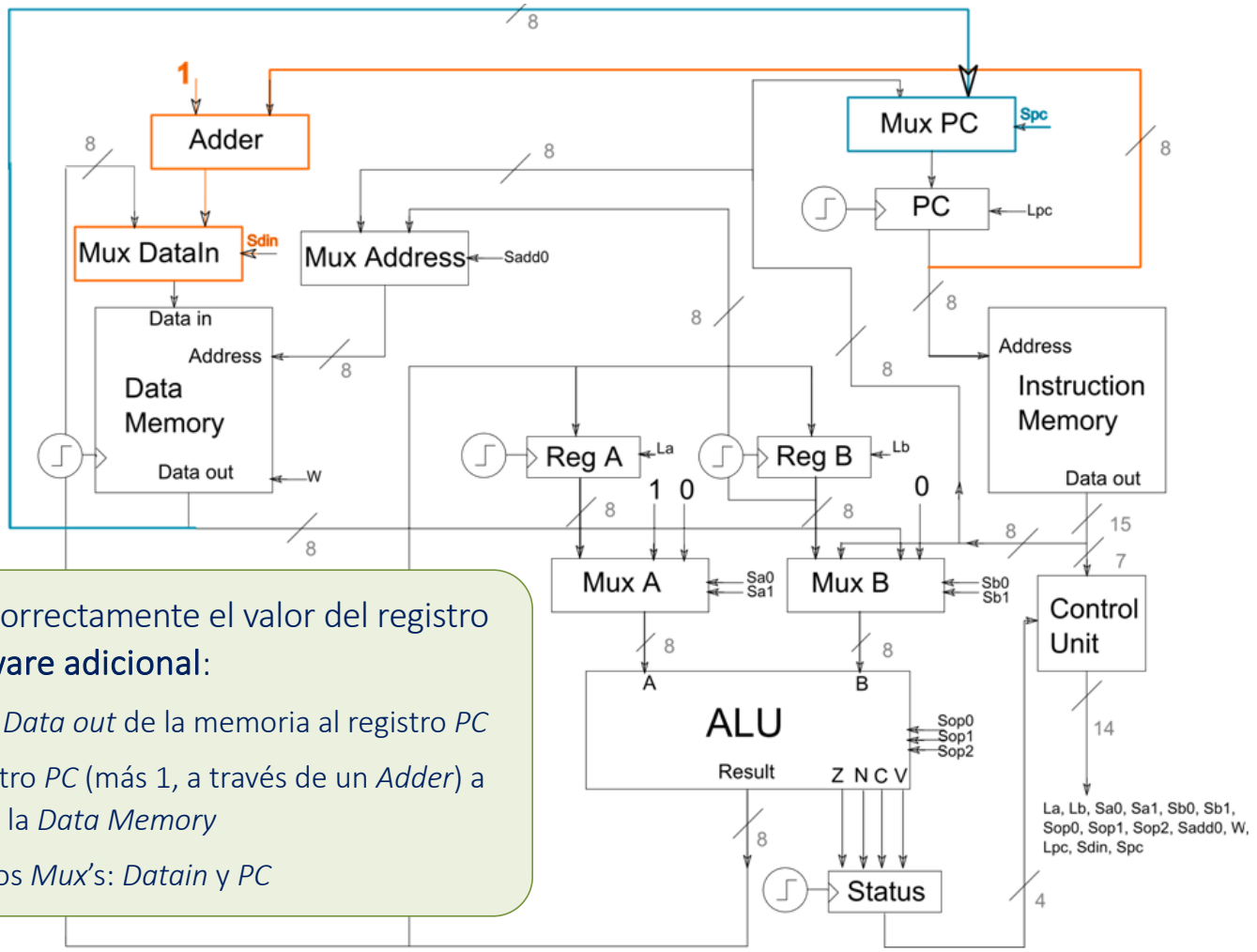
Para poder tener **subrutinas**:

$\Longrightarrow$ ... + nuevas conexiones + nuevas instrucciones



El computador básico, tal como está, permite pasar los parámetros y pasar de vuelta el valor de retorno:

- basta con la capacidad de almacenar y leer valores en la *Data Memory*, a través de los resgistros *A* y *B*

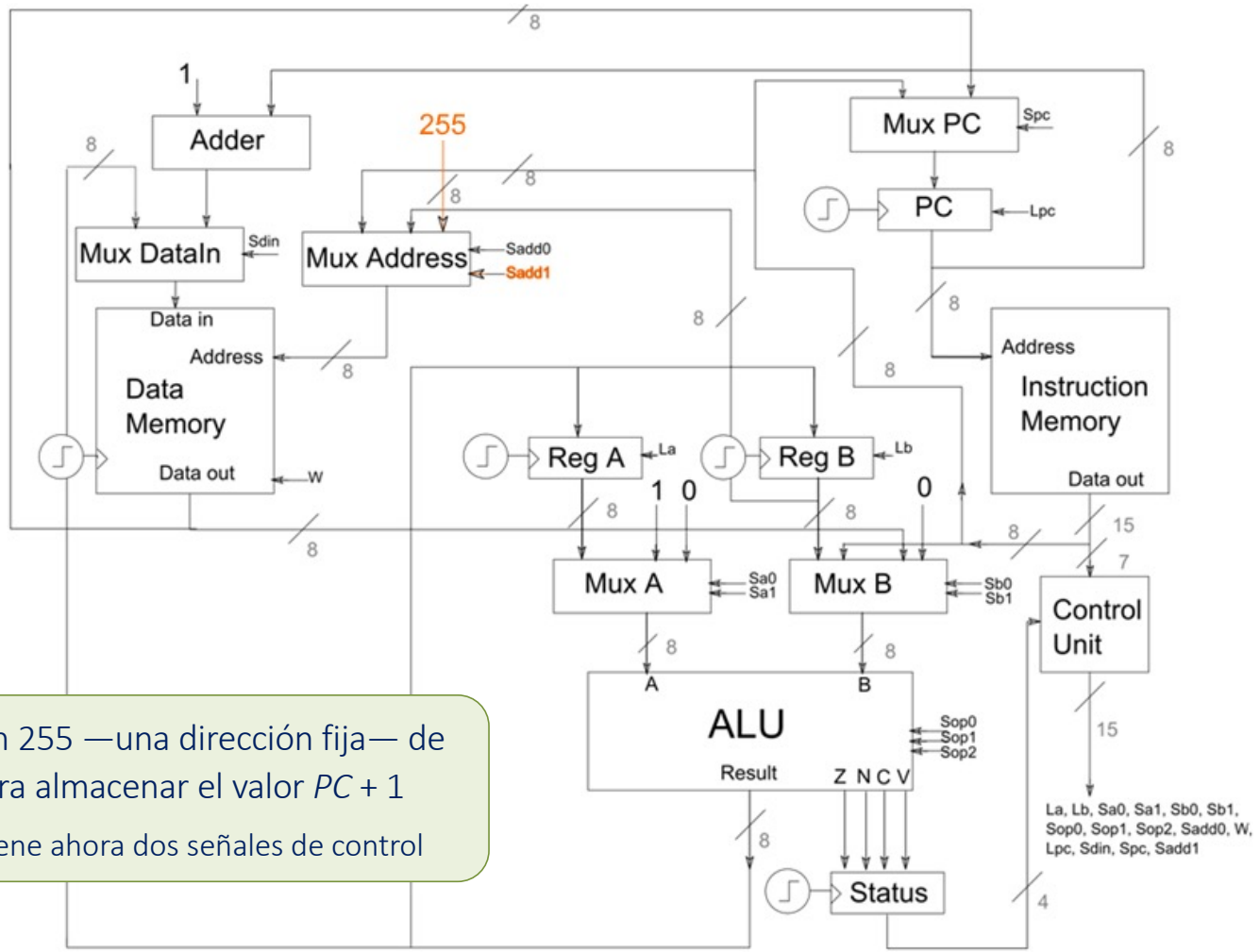


… pero para manejar correctamente el valor del registro *PC* necesitamos **hardware adicional**:

- conectamos la salida *Data out* de la memoria al registro *PC*

  … y la salida del registro *PC* (más 1, a través de un *Adder*) a la entrada *Data in* de la *Data Memory*

- agregamos dos nuevos *Mux*'s: *Datain* y *PC*



Usamos la dirección 255 —una dirección fija— de la *Data Memory* para almacenar el valor *PC* + 1

- el *Mux Address* tiene ahora dos señales de control

Por lo tanto, agregamos dos nuevas instrucciones a nuestro *assembly*:

`CALL dir` : almacena ***PC*** + 1 en la dirección 255 de la *Data Memory*

primer efecto de **`CALL dir`**

$$\mathbf{Mem}[255] \leftarrow \mathbf{PC}+1$$

… y al mismo tiempo salta a la dirección ***dir*** de la *Instruction Memory*

segundo efecto de **`CALL dir`**

$$\mathbf{PC} \leftarrow \mathbf{dir}$$

la instrucción **`CALL dir`** tiene dos efectos

`RET` : guarda en ***PC*** el valor de **Mem**[255]

efecto de **`RET`**

$$\mathbf{PC} \leftarrow \mathbf{Mem}[255]$$

`RET` es siempre la última instrucción de la función (en lenguaje *assembly*)

… es decir, se reanuda la ejecución de la instrucción (del `main`) inmediatamente siguiente al llamado a la función

P.ej., **main** inicia su ejecución colocando los valores 5 y 2 en los registros *A* y *B*

… entonces tiene que llamar a **func1**, que tiene dos parámetros: **var1** y **var2**

Así, antes de hacer la llamada, **main** asigna los valores que están en *A* y *B* a **var1** y **var2** ⟶ "pasa" los parámetros

func1 suma los valores de **var1** y **var2** entre sí, para lo cual primero los coloca en los registros

… finalmente, deja el resultado en **var1** para que lo pueda usar el **main**

Veamos ahora cómo ocurre esto paso a paso

```
        DATA:
…
128     var1
129     var2
…
255

        CODE:
…
20      main:  MOV A,5
21          MOV B,2
22          MOV (var1),A
23          MOV (var2),B
24          CALL func1
25          ...
…
55      func1: MOV A,(var1)
56          MOV B,(var2)
57          ADD A,B
58          MOV (var1),A
59          RET
```

main inicia su ejecución

PC = 20
PC = 21

... colocando los valores 5 y 2 en los registros *A* y *B*

```
        DATA:
...
128     var1
129     var2
...
255

        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             RET
```

... entonces tiene que llamar a **func1**, que tiene dos parámetros: **var1** y **var2**

Así, antes de hacer la llamada, **main** asigna los valores en *A* y *B* a **var1** y **var2** ⟶ "pasa" los parámetros

PC = 22

PC = 23

```
        DATA:
...
128     var1    5
129     var2    2
...
255

        CODE:
...
20      main:  MOV A,5
21          MOV B,2
22          MOV (var1),A
23          MOV (var2),B
24          CALL func1
25          ...
...
55      func1: MOV A,(var1)
56          MOV B,(var2)
57          ADD A,B
58          MOV (var1),A
59          RET
```

Ahora **`main`** hace la llamada **`CALL func1`**

PC = 24

... que produce los dos efectos descritos en la diap. 12

Mem[255] ← ***PC***+1 (= 25)

PC ← ***dir*** (= 55)

```
        DATA:
...
128     var1    5
129     var2    2
...
255             25

        CODE:
...
20      main:  MOV A,5
21          MOV B,2
22          MOV (var1),A
23          MOV (var2),B
24          CALL func1
25          ...
...
55      func1: MOV A,(var1)
56          MOV B,(var2)
57          ADD A,B
58          MOV (var1),A
59          RET
```

func1 suma los valores de **var1** y **var2** entre sí, para lo cual primero los coloca en los registros

PC = 55
PC = 56
PC = *57*

```
        DATA:
...
128     var1    5
129     var2    2
...
255             25

        CODE:
...
20      main:  MOV A,5
21          MOV B,2
22          MOV (var1),A
23          MOV (var2),B
24          CALL func1
25          ...
...
55      func1: MOV A,(var1)
56          MOV B,(var2)
57          ADD A,B
58          MOV (var1),A
59          RET
```

func1 finalmente deja el resultado en **var1** para que lo pueda usar el **main**

PC = 58

... y ejecuta **RET**

PC = 59

... que produce el efecto descrito en la diap. 12

PC ← **Mem**[255] (= 25)

(... de modo que se reanudará la ejecución del **main** a partir de la instrucción inmediatamente siguiente al **CALL**)

```
        DATA:
...
128     var1    7
129     var2    2
...
255             25

        CODE:
...
20      main:  MOV A,5
21          MOV B,2
22          MOV (var1),A
23          MOV (var2),B
24          CALL func1
25          ...
...
55      func1: MOV A,(var1)
56          MOV B,(var2)
57          ADD A,B
58          MOV (var1),A
59          RET
```

Pero, ¿qué pasa en esta caso?

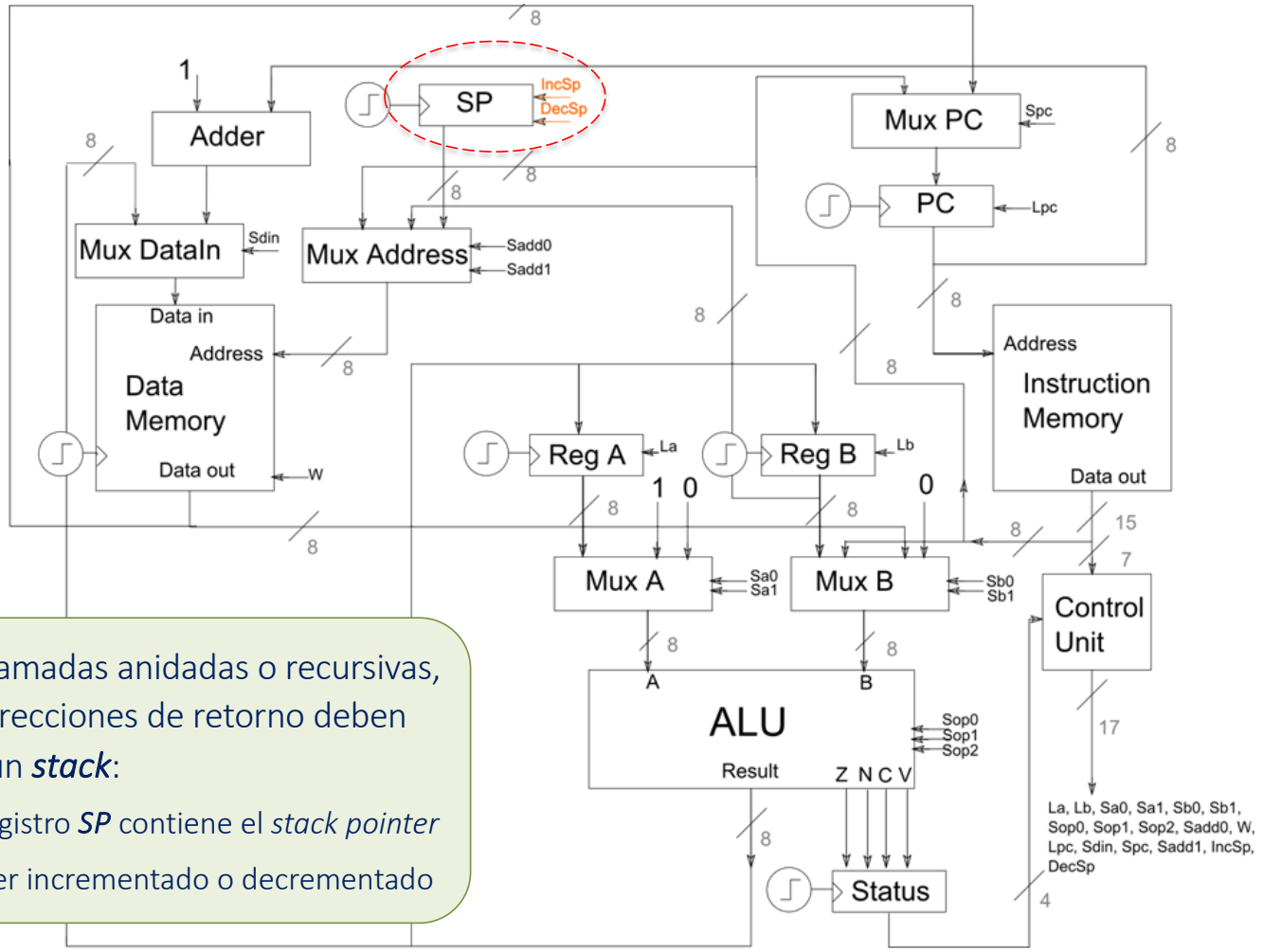
main llama a **func1** ⟶ en **Mem**[255] queda almacenada la dirección de retorno, 25

… pero antes de que **func1** termine (ejecute **RET**), la propia **func1** llama a su vez a la función **func2**

Entonces, la dirección de retorno que hay en **Mem**[255] en ese instante —la dirección 25— es sustituida por la dirección 60, del código de **func1**

… y la ejecución nunca vuelve al **main**

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```



En el caso de llamadas anidadas o recursivas, las sucesivas direcciones de retorno deben manejarse en un ***stack***:

- el (nuevo) registro ***SP*** contiene el *stack pointer*

  ... y puede ser incrementado o decrementado

Para estar seguros:

- el registro ***PC*** almacena direcciones de la *Instruction Memory*
- el registro ***SP*** almacena direcciones de la *Data Memory*

Así, al llamar a una función, debemos (en *un ciclo del reloj*):

- guardar ***PC***+1 en la *Data Memory* en la dirección contenida en ***SP***
- decrementar en 1 el valor de ***SP***
- guardar la dirección de la primera instrucción de la función en ***PC***

estas tres acciones se ejecutan en paralelo en **un mismo ciclo**

la instrucción **`CALL dir`** tiene **tres** efectos

… y, luego, al retornar desde la función (en *dos ciclos del reloj*):

- incrementar en 1 el valor de ***SP***
- guardar en ***PC*** la dirección almacenada en la dirección ***SP*** (ya incrementado) de la *Data Memory*

estas dos acciones se ejecutan secuencial-mente **en dos ciclos**

la instrucción **`RET`** tiene **dos** efectos

Tomemos la ejecución justo antes de que **`main`** ejecute **`CALL func1`**

PC = 22, *SP* = 255

PC = 23, *SP* = 255

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

```
        DATA:
...
128     var1    5
129     var2    2
...
253
254
255
```

direcciones de memoria reservadas para manejar el stack

Ahora **main** hace la llamada **CALL func1**

PC = 24, *SP* = 255

... que produce los tres efectos descritos en la diap. 21:

Mem[*SP*] ← ***PC***+1 (= 25)
SP ← *SP*–1 (= 254)
PC ← ***dir*** (= 55)

```
        DATA:
...
128     var1    5
129     var2    2
...
253
254
255             25
```

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Un poco después **func1** hace la llamada **CALL func2**

$PC = 59$, $SP = 254$

... que produce los tres efectos descritos en la diap. 21:

Mem[*SP*] ← ***PC***+1 (= 60)
SP ← *SP*–1 (= 253)
PC ← ***dir*** (= 77)

```
        DATA:
...
128     var1    7
129     var2    2
...
253
254             60
255             25
```

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Un poco después **func2** ejecuta **RET**

PC = 80, *SP* = 253

... que produce los dos efectos descritos en la diap. 21:

SP ← ***SP***+1 (= 254)
PC ← **Mem**[*SP*] (= 60)

```
        DATA:
...
128     var1    7
129     var2    2
...
253
254             60
255             25
```

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Y luego **func1** ejecuta **RET**

PC = 60, *SP* = 254

... que produce los dos efectos descritos en la diap. 21:

SP ← ***SP***+1 (= 255)
PC ← **Mem**[*SP*] (= 25)

```
        DATA:
...
128     var1    7
129     var2    2
...
253
254             60
255             25
```

```
        CODE:
...
20      main:  MOV A,5
21             MOV B,2
22             MOV (var1),A
23             MOV (var2),B
24             CALL func1
25             ...
...
55      func1: MOV A,(var1)
56             MOV B,(var2)
57             ADD A,B
58             MOV (var1),A
59             CALL func2
60             RET
...
77      func2: MOV A,(var1)
78             MOV B,(var2)
79             ADD A,B
80             RET
```

Finalmente

... ¿qué pasa en este caso con los contenidos de los registros *A y* B?

$PC = 21, A = 5, B = \ldots$

$PC = 22, A = 5, B = 3$

$PC = 23 \Rightarrow PC = 55, A = 5, B = 3$

$PC = 55, A = 5, B = 4$

$PC = 56, A = 9, B = 4$

$PC = 57 \Rightarrow PC = 24, A = 9, B = 4$

$PC = 24, A = 13, B = 4$

Pero, ¿es este el resultado esperado por **main**?

```
        CODE:
...
20      main: ...
21          MOV A,5
22          MOV B,3
23          CALL func
24          ADD A,B
25          ...
...
55      func: INC B
56          ADD A,B
57          RET
```

En el stack de la *Data Memory* guardamos las direcciones de retorno —direcciones de las instrucciones cuya ejecución quedó pendiente debido al llamado a una función— para poder recuperarlas al volver de las llamadas correspondientes

También tenemos que guardar, en el mismo stack, los valores que los registros *A* y *B* tienen al momento de llamar a una función

… para que la función llamada pueda usar libremente estos registros

… y el **`main`** (o la función que hizo la llamada) pueda recuperar esos valores una vez que la llamada termina

Agregamos las instrucciones **PUSH** y **POP**:

PUSH Reg almacena en **Mem**[*SP*] el valor del registro ***Reg***, y luego decrementa *SP*

POP Reg primero incrementa *SP*, y luego escribe en ***Reg*** el valor almacenado en **Mem**[*SP*]

Entonces, en lugar del código de la izquierda, escribimos el código de la derecha

```
        CODE:
...
20      main: ...
21          MOV A,5
22          MOV B,3
23          CALL func
24          ADD A,B
25          ...
...
55      func: INC B
56          ADD A,B
57          RET
```

```
        CODE:
...
20      main: ...
21          MOV A,5
22          MOV B,3
23          PUSH A
24          PUSH B
25          CALL func
26          POP B
27          POP A
28          ADD A,B
29          ...
...
55      func: INC B
56          ADD A,B
57          RET
```

¡Ojo!: los **POP**s deben ejecutarse en el orden inverso a los **PUSH**s

En definitiva, para permitir el uso de funciones, hemos agregado 6 nuevas instrucciones a nuestro *assembly*:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP- -, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP- -		-
PUSH	B	Mem[SP] = B, SP- -		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
		B = Mem[SP]		-

SP ← SP–1

SP ← SP+1

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
XOR	A,B	0101000		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0101001		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101010		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101011		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101100		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101101		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101110		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101111		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0110011		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0110100		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110101		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0111001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0111010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
CMP	A,B	0111011		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0111100		0	0	0	A	LIT	SUB	-	-	-	0	0	0
JMP	Dir	0111101		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111110	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111111	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	1000000	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	1000001	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	1000010	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	1000011	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000100	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000101	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000101		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000110		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1001000		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
PUSH	B	1001001		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1001010		0	1	0	-	-	-	-	-	-	0	1	0
		1001011		0	1	0	ZERO	DOUT	ADD	SP	ALU	-	0	0	0
POP	B	1001100		0	0	1	-	-	-	-	-	-	0	1	0
		1001101		0	0	1	ZERO	DOUT	ADD	SP	ALU	-	0	0	0

El ej. de la diap. 27 es artificial (¿por qué la surbutina **func** va a incrementar el registro *B* sin primero averiguar cuánto vale?)

Veamos un ej. más realista:

Tomemos el programa de la clase 6 que multiplica las variables **a** × **b** (repetido aquí en la fig. a la derecha) y convirtámoslo en una subrutina **mult**

... que mutliplica sus parámetros formales **mnd** × **mdr**

... y luego usemos esta subrutina en el programa **main**

```
DATA:
a    15         —multiplicando
b    7          —multiplicador
prod       —producto, no inicializado

CODE:           —multiplicación, sin signo
     MOV A,0
     MOV (prod),A  —inicializamos prod = 0
     MOV B,0       —B va a ser el contador de repeticiones
     MOV A,(b)     —A = número de repeticiones (multiplicador)
loop:    CMP A,B          —comparamos
     JEQ end       —si son iguales, terminamos (saltamos a end)
     MOV A,(a)     —tomamos el multiplicando
     ADD A,(prod)  —se lo sumamos una vez más a prod
     MOV (prod),A  —y actualizamos el valor de prod
     INC B         —incrementamos el contador de repeticiones
     MOV A,(b)     —A = número de repeticiones (multiplicador)
     JMP loop      —repetimos el ciclo
end:
```

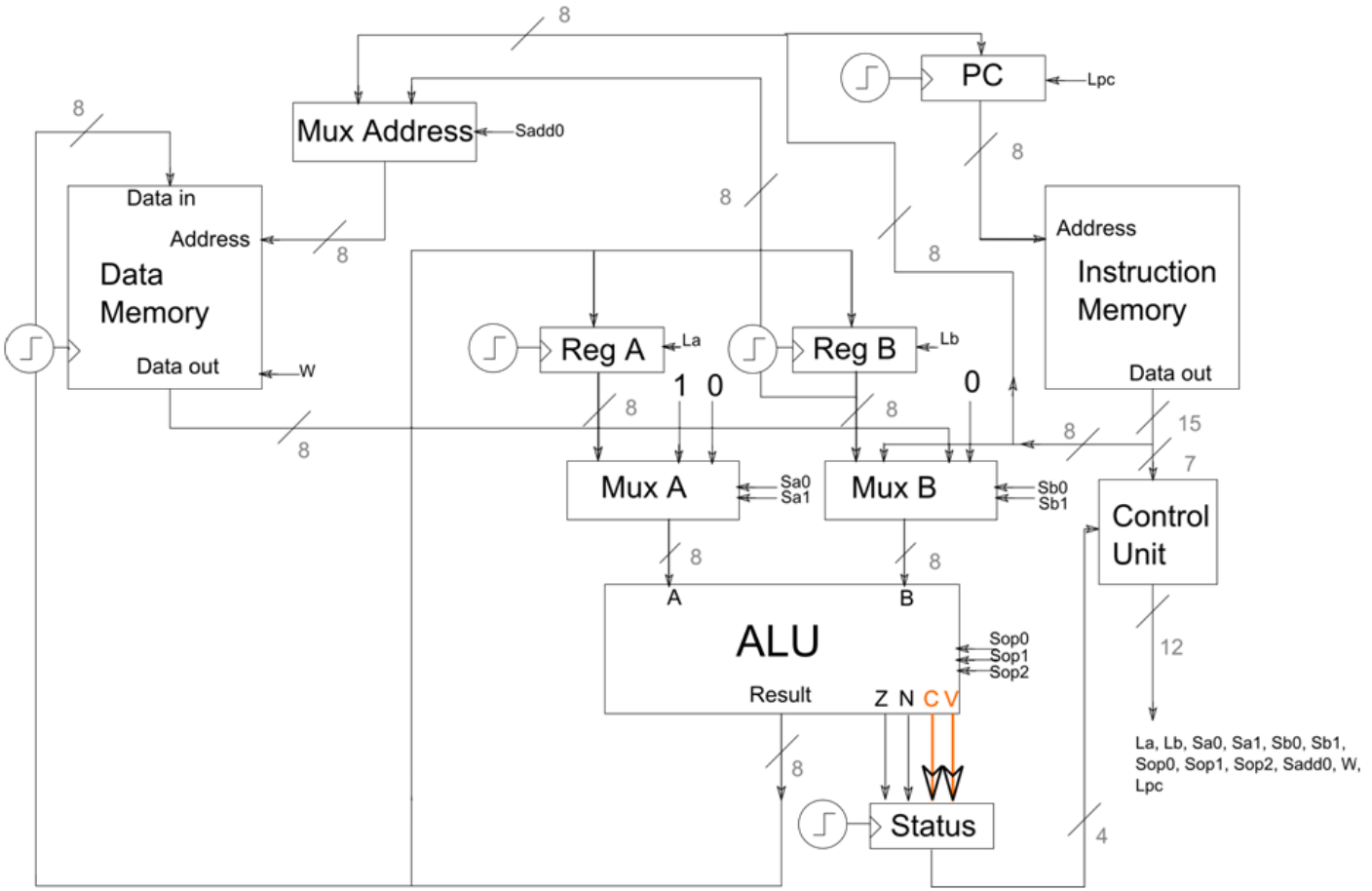
```
DATA:
a    15
b    7
prod

CODE:
     MOV A,0
     MOV (prod),A
     MOV B,0
     MOV A,(b)
loop:    CMP A,B
     JEQ end
     MOV A,(a)
     ADD A,(prod)
     MOV (prod),A
     INC B
     MOV A,(b)
     JMP loop
end:
```

```
DATA:
mnd
mdr
prod

mult:
     MOV A,0
     MOV (prod),A
     MOV B,0
     MOV A,(mdr)
loop:    CMP A,B
     JEQ end
     MOV A,(mnd)
     ADD A,(prod)
     MOV (prod),A
     INC B
     MOV A,(mdr)
     JMP loop
end:     RET
```

El registro ***Status*** puede también tener bits que indican la presencia de un carry o de overflow en el resultado producido por la ALU



Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

```
main:
    w = 3
    l = 4
    h = 5
    v = volumen(w, l, h)
    print(“El volumen es:”, v)

volumen(ancho, largo, alto):
    a = area(ancho, largo)
    return a*alto

area(lado1, lado2):
    return lado1*lado2
```