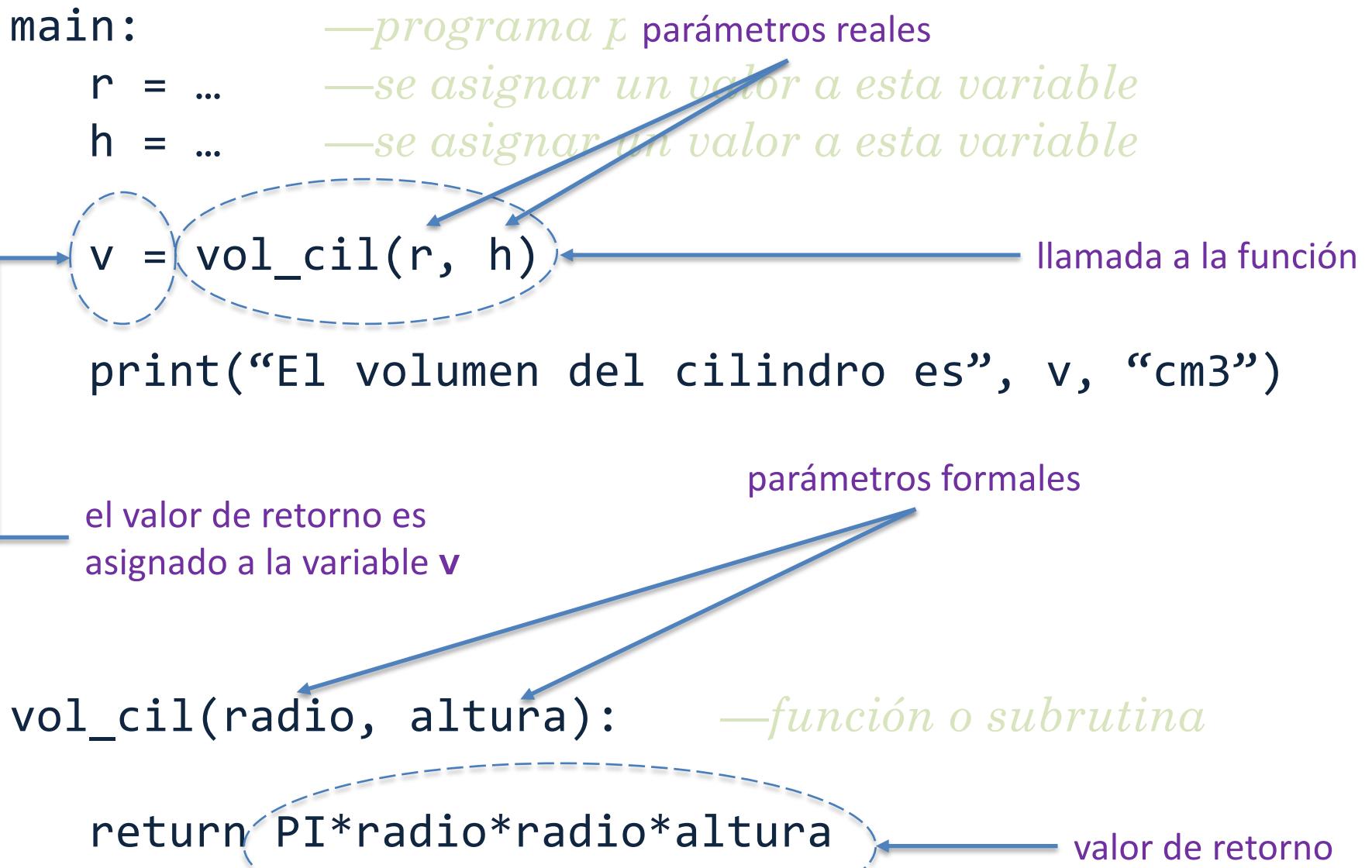


Subrutinas (funciones o métodos)

```
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
```



DATA: —en Data Memory

vol-cil:

radio ...

altura ...

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

- ... mediante una instrucción —que hay que agregar al **main**— equivalente a un salto incondicional, que cambie el valor del registro *PC*

próximo *PC*

main:

r 2

h 5

v ...

CODE: —en Instruction Memory

vol-cil:

...

...

...

PC actual →

main:

...

...

...

2) Sólo que 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 almacenar los valores de los parámetros reales en algún lugar de la *Data Memory* al que la función tenga acceso

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

DATA: —en *Data Memory*

vol-cil:

radio 2
altura 5

a través de
los registros

main:

r 2
h 5
v ...

CODE: —en *Instruction Memory*

vol-cil:

...
...
...
...

main:

...
...
...
...

DATA: —en Data Memory

3) Finalmente, al terminar la ejecución de la función, es necesario “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
- → este valor debió haber quedado guardado en alguna parte antes de que se empezara a ejecutar la función

vol-cil:

radio	2
altura	5
retval	63

main:

r	2
h	5
v	...

a través de los registros

CODE: —en Instruction Memory

vol-cil:

...
...
...
...

main:

...
...
...
...

PC actual

próximo *PC*

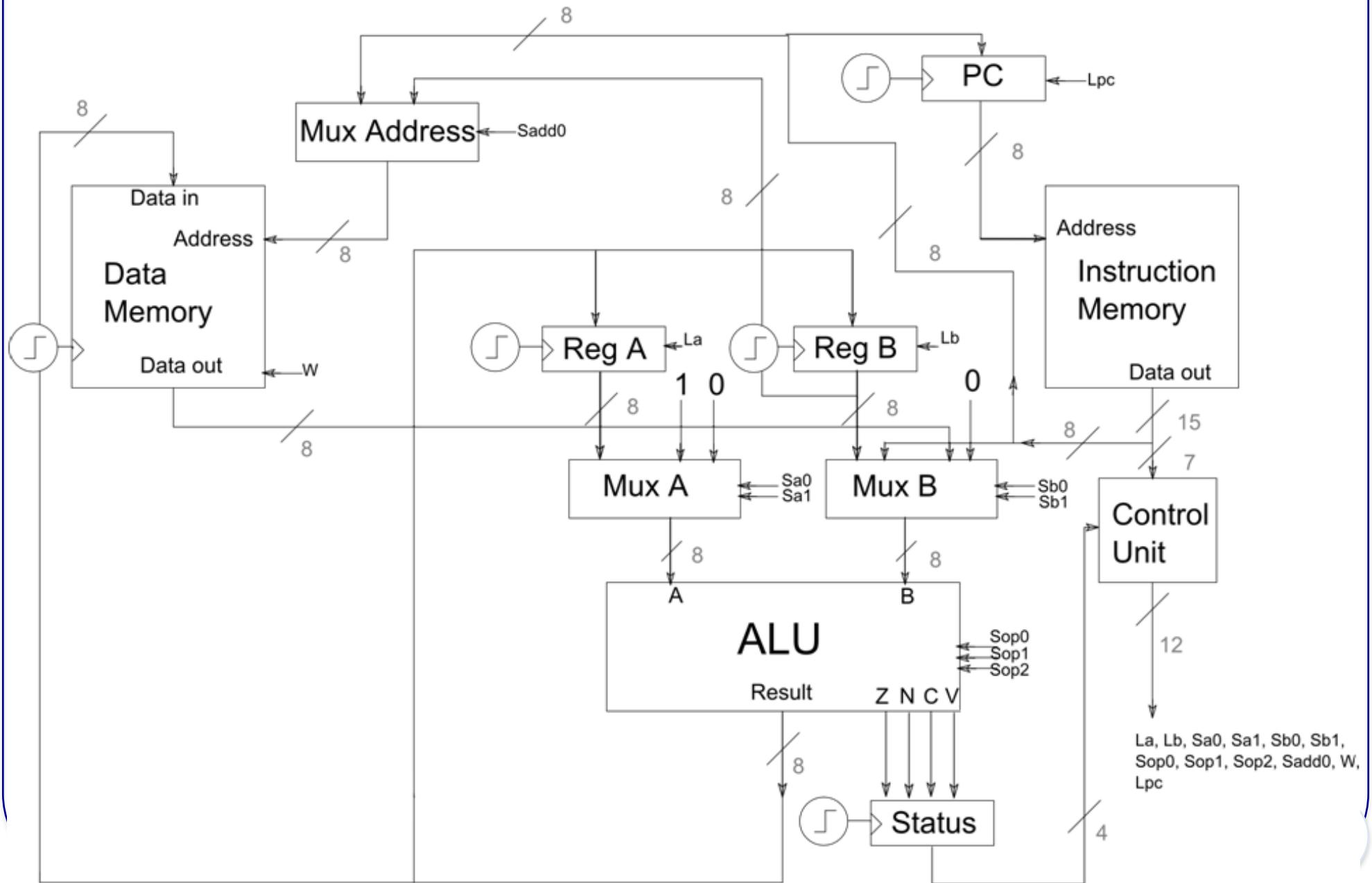
En las siguientes diapositivas, vamos a construir de a poco una solución al problema de llamar funciones con parámetros, cuando hay llamadas anidadas

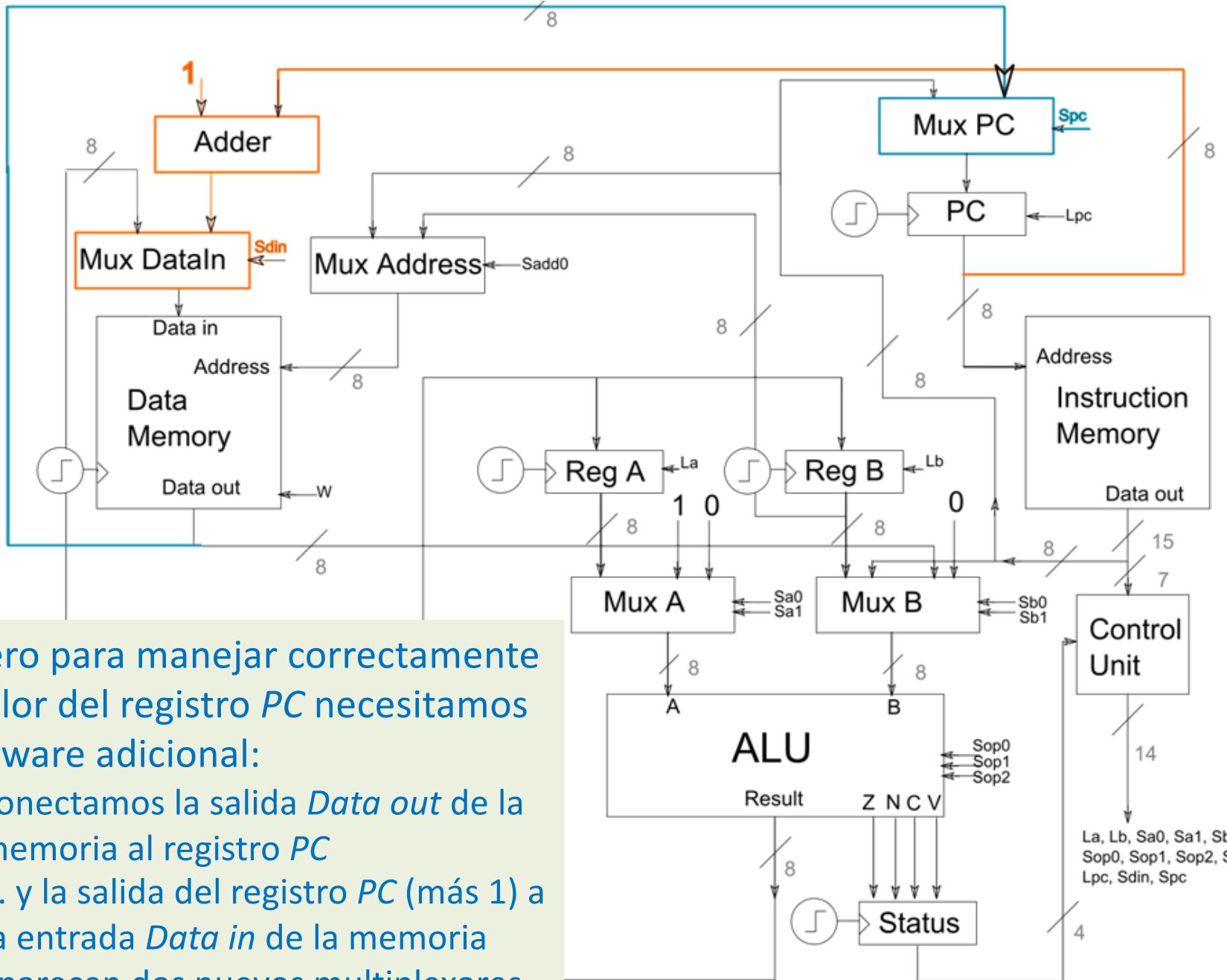
- veremos primero el caso de una única llamada a una función
- luego, el caso de una llamada a una función que a su vez llama a otra función
- finalmente, cómo manejar los registros *A* y *B* cuando sus valores antes de la llamada a una función deben seguir disponibles al volver de la llamada

Primero, vamos a construir la solución en el caso de nuestro computador básico

... y luego, vamos a ver cómo se resuelve en RISC-V

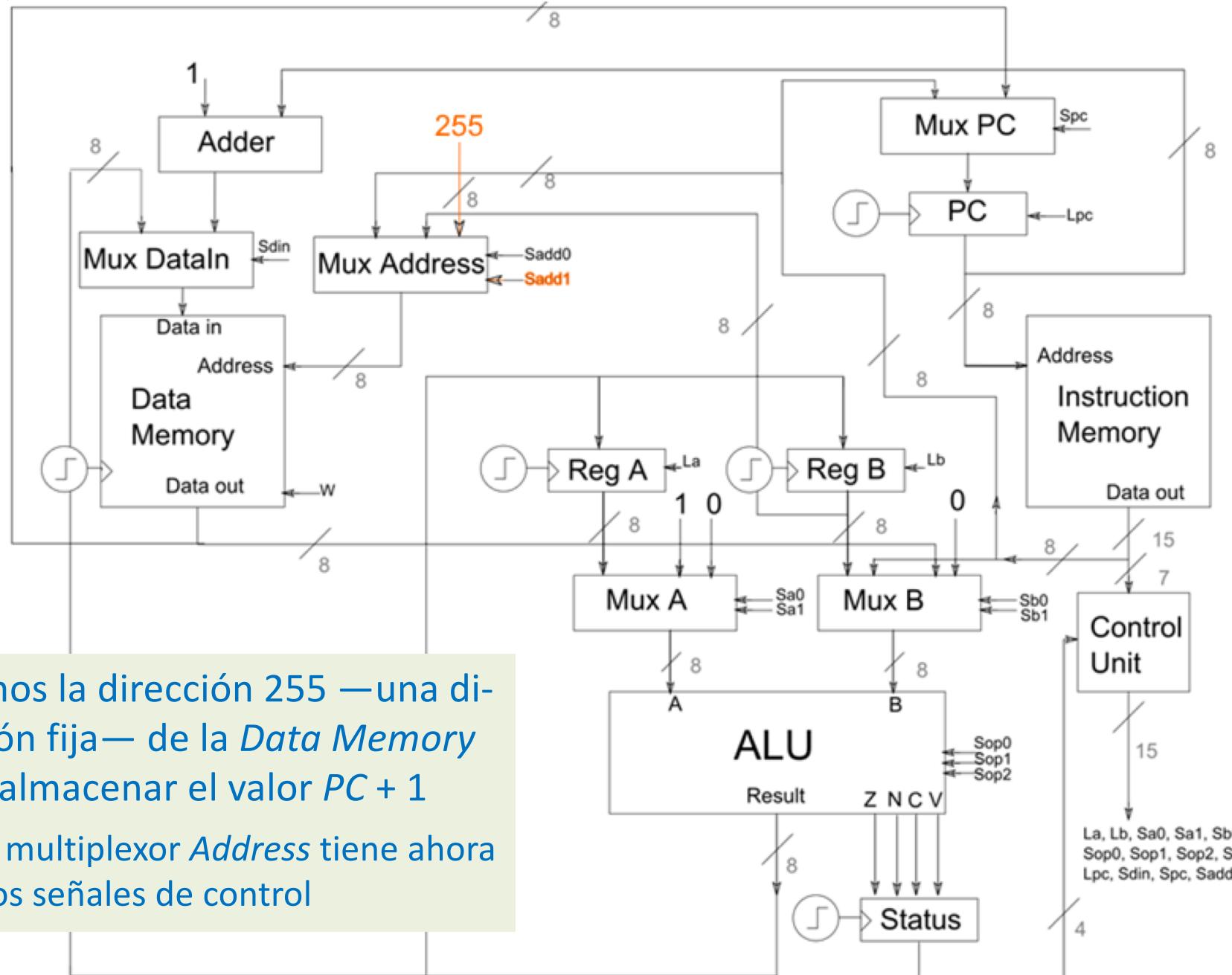
El computador básico permite pasar los parámetros y pasar de vuelta el valor de retorno: es (sólo) almacenar y leer valores en la *Data Memory*





... 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 la entrada *Data in* de la memoria
- aparecen dos nuevos multiplexores



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

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

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

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

$$\text{Mem}[255] \leftarrow PC+1 \quad \begin{matrix} \text{primer efecto} \\ \text{de CALL } dir \end{matrix}$$

... e inmediatamente salta a la dirección *dir* de la *Instruction Memory*

$$PC \leftarrow dir \quad \begin{matrix} \text{segundo efecto} \\ \text{de CALL } dir \end{matrix}$$

RET : guarda en PC el valor de $\text{Mem}[255]$

$$PC \leftarrow \text{Mem}[255] \quad \begin{matrix} \text{efecto de RET} \end{matrix}$$

... es decir, se reanuda la ejecución de la instrucción inmediatamente siguiente al llamado a la función:

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

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 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**

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

DATA:

```
...  
128    var1  
129    var2  
...  
255
```

P.ej.: **main** inicia su ejecución

PC = 20

PC = 21

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

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. #71

Mem[255] $\leftarrow PC+1 (= 25)$

PC $\leftarrow dir (= 55)$

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

DATA:

...		
128	var1	5
129	var2	2
...		
255		25

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

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. #71

PC \leftarrow 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 este caso?

main llama a **func1**, con lo que en **Mem[255]** queda almacenada, como ya vimos, la dirección de retorno 25

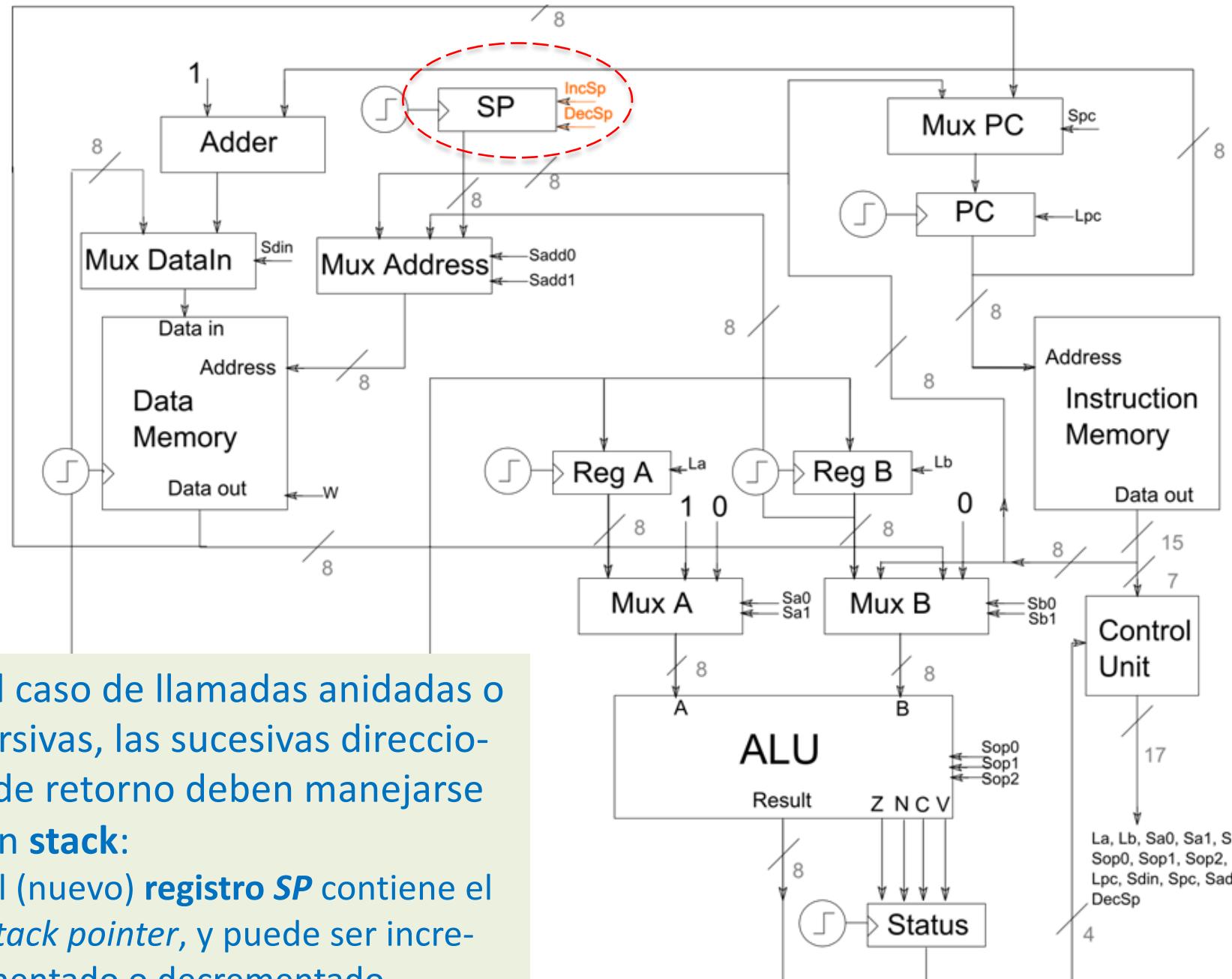
... pero antes de que **func1** termine (ejecute **RET**), la propia **func1** llama a su vez a la función **func2**

Entonces, la dirección de retorno que hay en **Mem[255]** en ese instante —la dirección 25, del código de **main**— 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**

... 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*

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

PC = 22, SP = 255

PC = 23, SP = 255

DATA:

...

128 var1 5

129 var2 2

...

253

254

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

Ahora **main** hace la llamada
CALL func1

PC = 24, SP = 255

... que produce los tres efectos descritos en la diap. #80:

Mem[SP] \leftarrow PC+1 (= 25)

SP \leftarrow SP-1 (= 254)

PC \leftarrow 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. #80:

Mem[SP] $\leftarrow PC+1 (= 60)$

SP $\leftarrow SP-1 (= 253)$

PC $\leftarrow dir (= 77)$

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

CODE:

Un poco después **func2** ejecuta RET

PC = 80, SP = 253

... que produce los dos efectos descritos en la diap. #80:

SP ← SP+1 (= 254)

PC ← Mem[SP] (= 60)

DATA:

...

128 var1 7

129 var2 2

...

253

254 60

255 25

...	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

CODE:

Y luego **func1** ejecuta RET

PC = 60, SP = 254

... que produce los dos efectos descritos en la diap. #80:

SP ← SP+1 (= 255)

PC ← Mem[SP] (= 25)

DATA:

...

128 var1 7

129 var2 2

...

253

254 60

255 25

...

20

21

22

23

24

25

...

55

56

57

58

59

60

...

77

78

79

80

main: MOV A,5

MOV B,2

MOV (var1),A

MOV (var2),B

CALL func1

...

func1: MOV A,(var1)

MOV B,(var2)

ADD A,B

MOV (var1),A

CALL func2

RET

func2: MOV A,(var1)

MOV B,(var2)

ADD A,B

RET

Finalmente

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

PC = 21, A = 5, B = ...

PC = 22, A = 5, B = 3

PC = 23 ⇒ PC = 55, A = 5, B = 3

PC = 55, A = 5, B = 4

PC = 56, A = 9, B = 4

PC = 57 ⇒ PC = 24, A = 9, B = 4

PC = 24, A = 13, B = 4

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

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

Así como en el stack de la *Data Memory* guardamos las direcciones de retorno (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 vuelve

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

PUSH Reg almacena en **Mem[SP]** el valor almacenado en el 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
```

¡Ojo!: los **POPs** deben ejecutarse
en el orden inverso a los **PUSHs**

CODE:

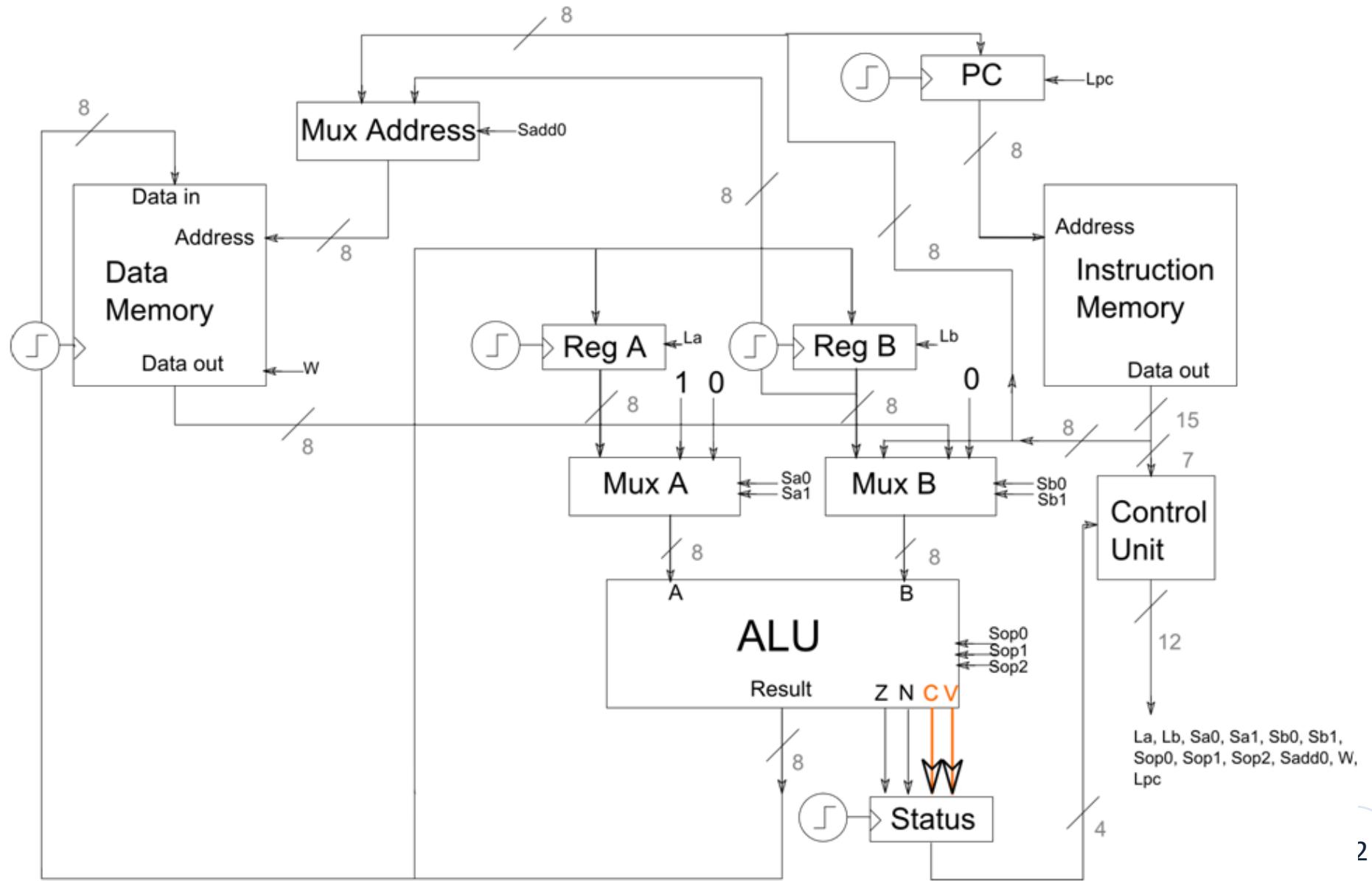
```
...  
20    main: ...  
21    MOV A, 5  
22    MOV B, 3  
23    PUSH A  
24    PUSH B  
25    CALL func  
26    POP B  
27    POP A  
28    ADD A, B  
29    ...  
...  
55    func: INC B  
56    ADD A, B  
57    RET
```

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

Instrucción	Operando	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP- -, PC = Dir SP++		CALL func
RET		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]		-

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
		1001100		0	0	1	-	-	-	-	-	-	0	1	0
POP	B	1001101		0	0	1	ZERO	DOUT	ADD	SP	ALU	-	0	0	0

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,Lit	A-B A-Lit		CMP A,0 JMP end
JMP	Dir	PC = Dir		
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