

Datapath

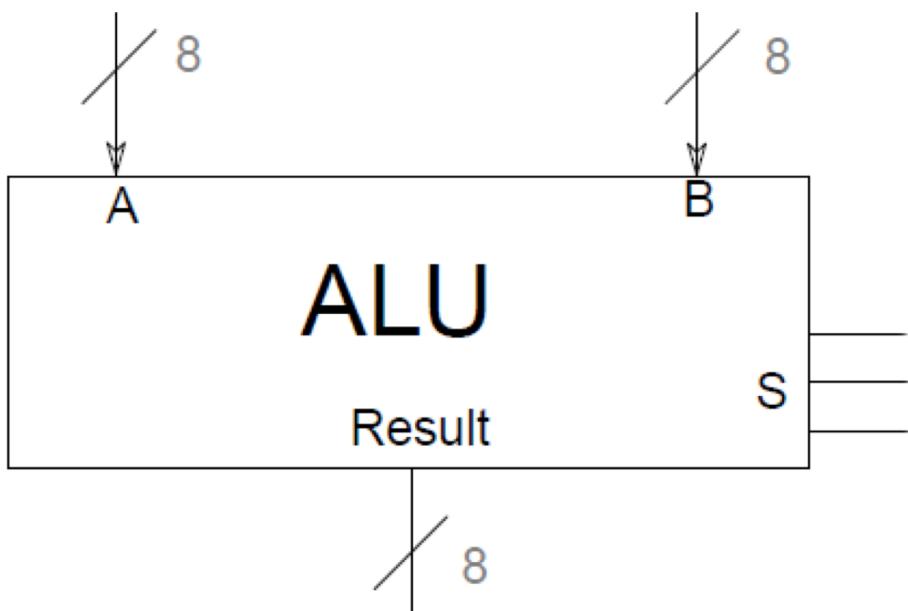
Arquitectura de Computadores – IIC2343

Datapath

El componente del procesador, o CPU, que realiza operaciones aritméticas

La ruta que conecta a los registros con la ALU, y por la cual viajan los datos

Dónde estamos

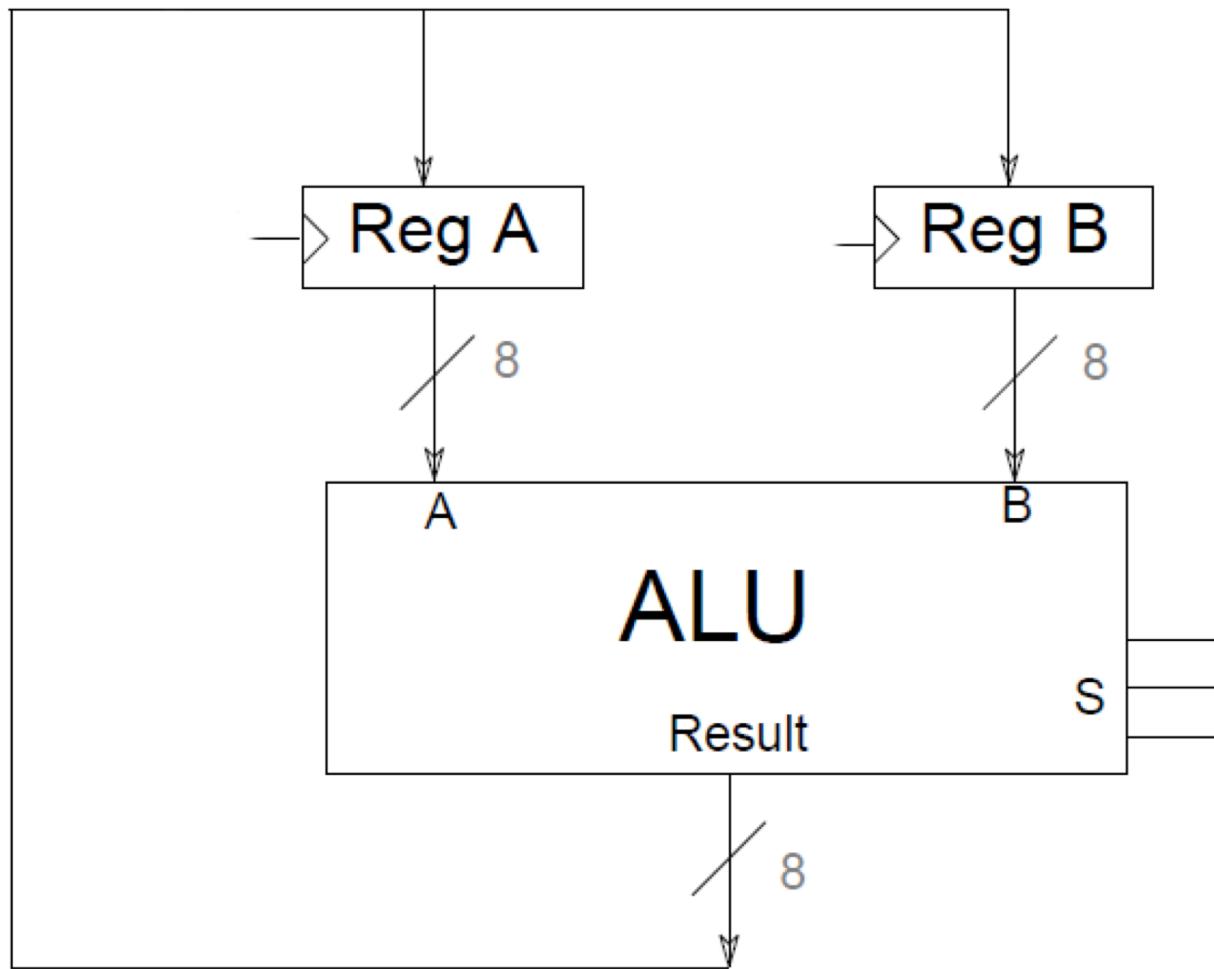


S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

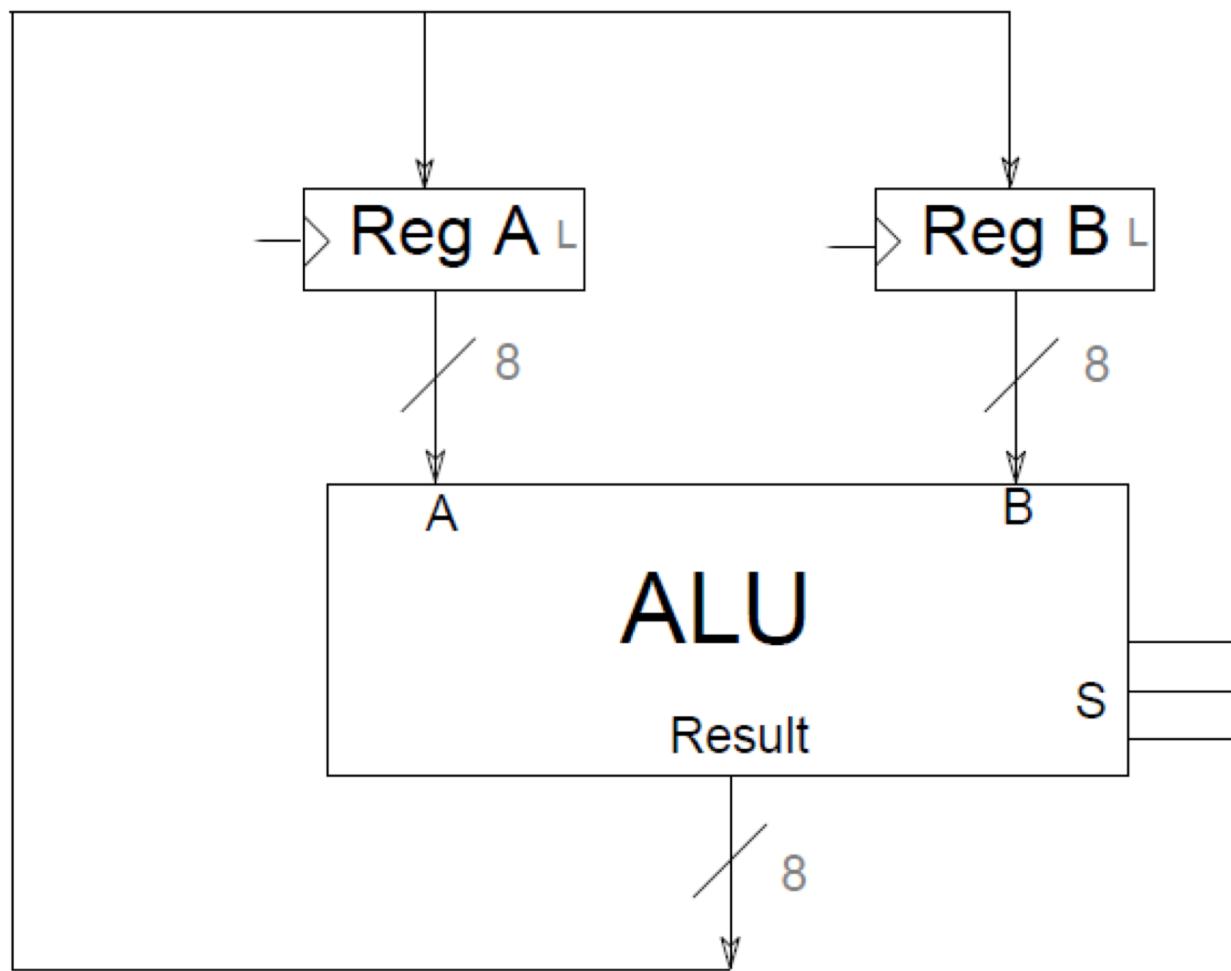
Las entradas *A* y *B* provienen de **registros** de la CPU:

- número limitado de ubicaciones especiales construidas directamente en el hardware —los ladrillos de la construcción de computadores

La salida *Result* va a parar a esos mismos registros: *A* y *B*

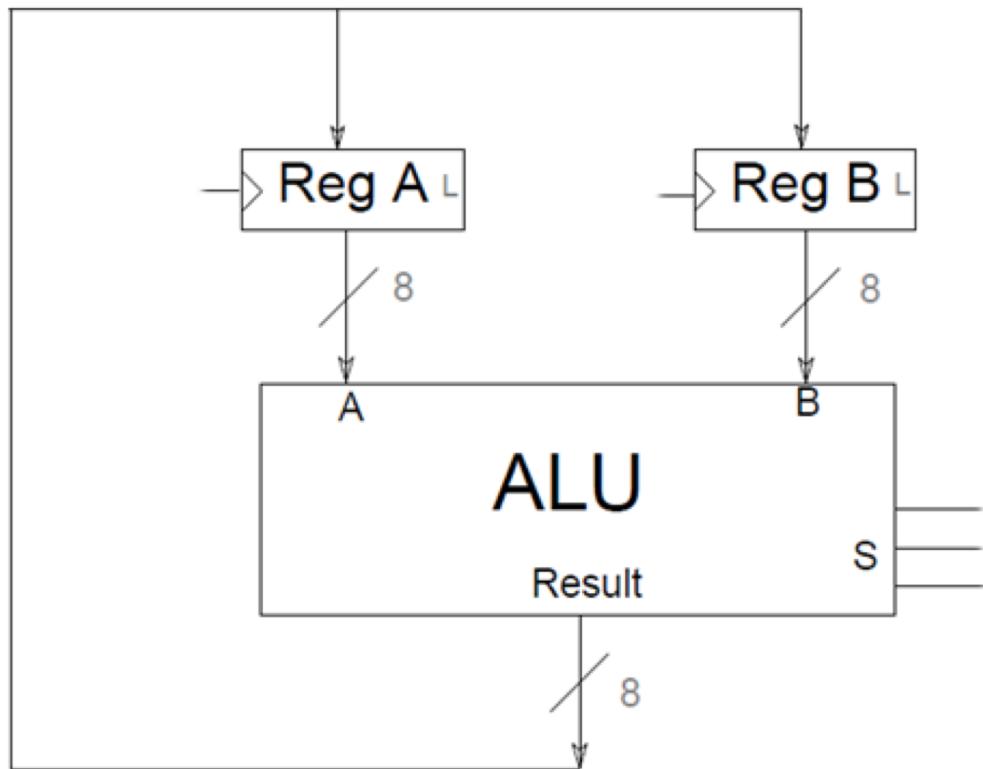


Agregamos las señales de control L_A y L_B para controlar la escritura —o actualización— de los valores— de los registros



Las diferentes combinaciones de valores de las cinco señales de control especifican qué acciones puede ejecutar este circuito:

- qué operación se ejecuta — S_0 , S_1 y S_2
- a dónde va a parar el resultado — L_A y L_B



la	lb	s2	s1	s0	operación
1	0	0	0	0	$A=A+B$
0	1	0	0	0	$B=A+B$
1	0	0	0	1	$A=A-B$
0	1	0	0	1	$B=A-B$
1	0	0	1	0	$A=A \text{ and } B$
0	1	0	1	0	$B=A \text{ and } B$
1	0	0	1	1	$A=A \text{ or } B$
0	1	0	1	1	$B=A \text{ or } B$
1	0	1	0	0	$A=\text{not}A$
0	1	1	0	0	$B=\text{not}A$
1	0	1	0	1	$A=A \text{ xor } B$
0	1	1	0	1	$B=A \text{ xor } B$
1	0	1	1	0	$A=\text{shift left } A$
0	1	1	1	0	$B=\text{shift left } A$
1	0	1	1	1	$A=\text{shift right } A$
0	1	1	1	1	$B=\text{shift right } A$

P.ej., si a partir de los valores 0 y 1 almacenados en los registros A y B, respectivamente, ejecutamos las seis acciones que se muestran en la columna de la izquierda, entonces los registros quedan con los valores que se muestran en las columnas de la derecha

la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

Cada combinación de valores de las señales de control es una **instrucción**

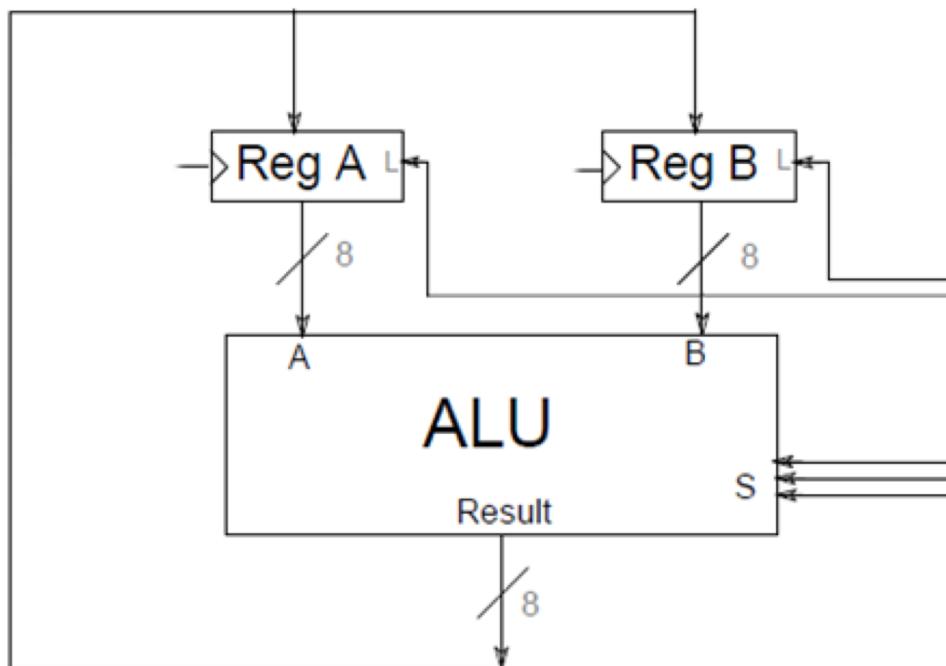
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A=A+B$	1	1
0	1	0	0	0	$B=A+B$	1	2
1	0	0	0	0	$A=A+B$	3	2
0	1	0	0	0	$B=A+B$	3	5
1	0	0	0	0	$A=A+B$	8	5
0	1	0	0	0	$B=A+B$	8	13

Cada combinación de valores de las señales de control es una **instrucción**

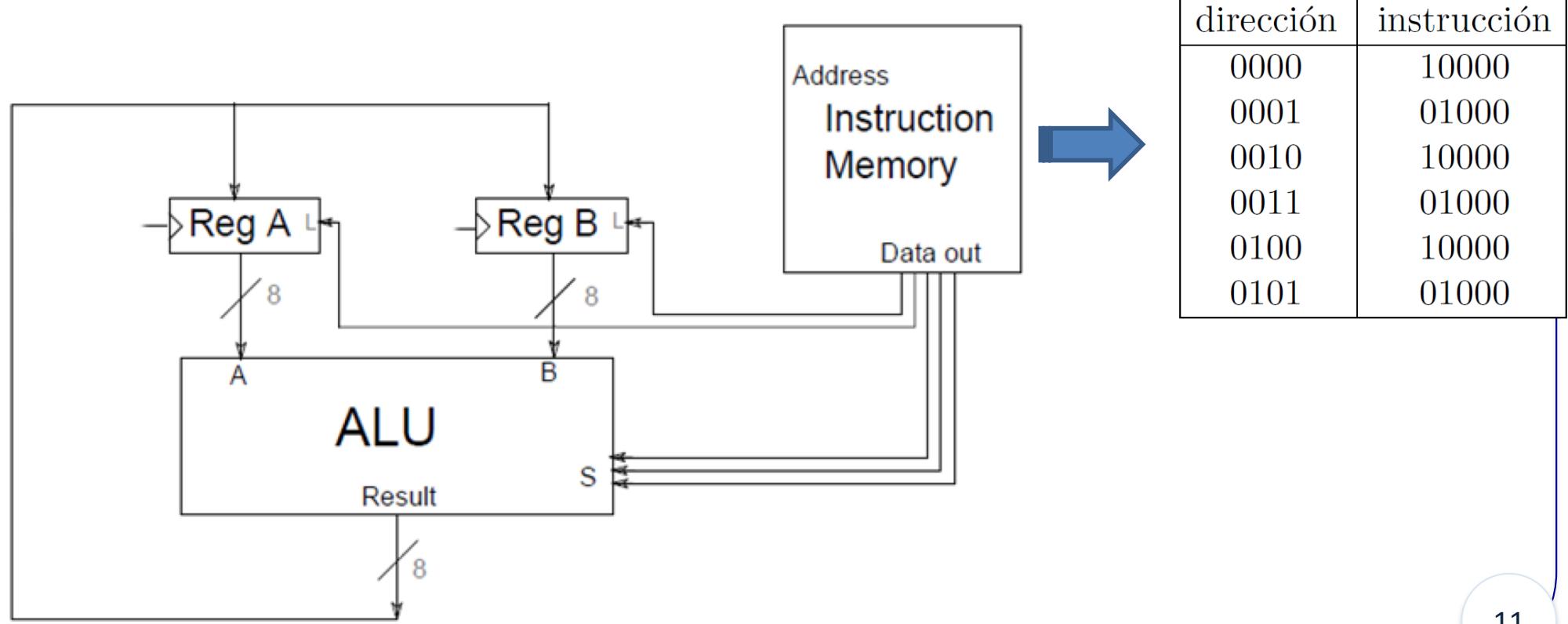
... y una secuencia de instrucciones es un **programa**

la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

Los computadores (que siguen el modelo de arquitectura llamado) von Neumann se caracterizan porque el programa —la secuencia de instrucciones— está almacenado en el mismo computador ...

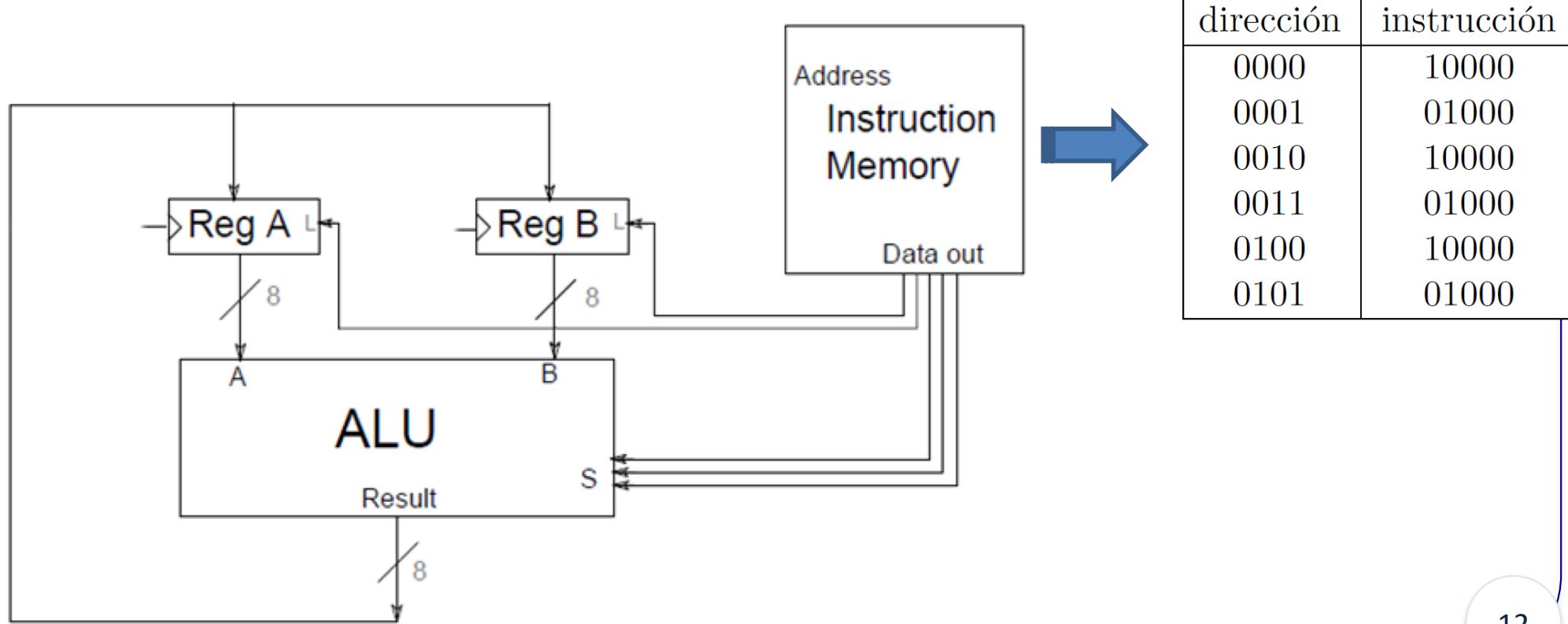


... en una memoria, p.ej., una memoria de tipo ROM (*read-only memory*)



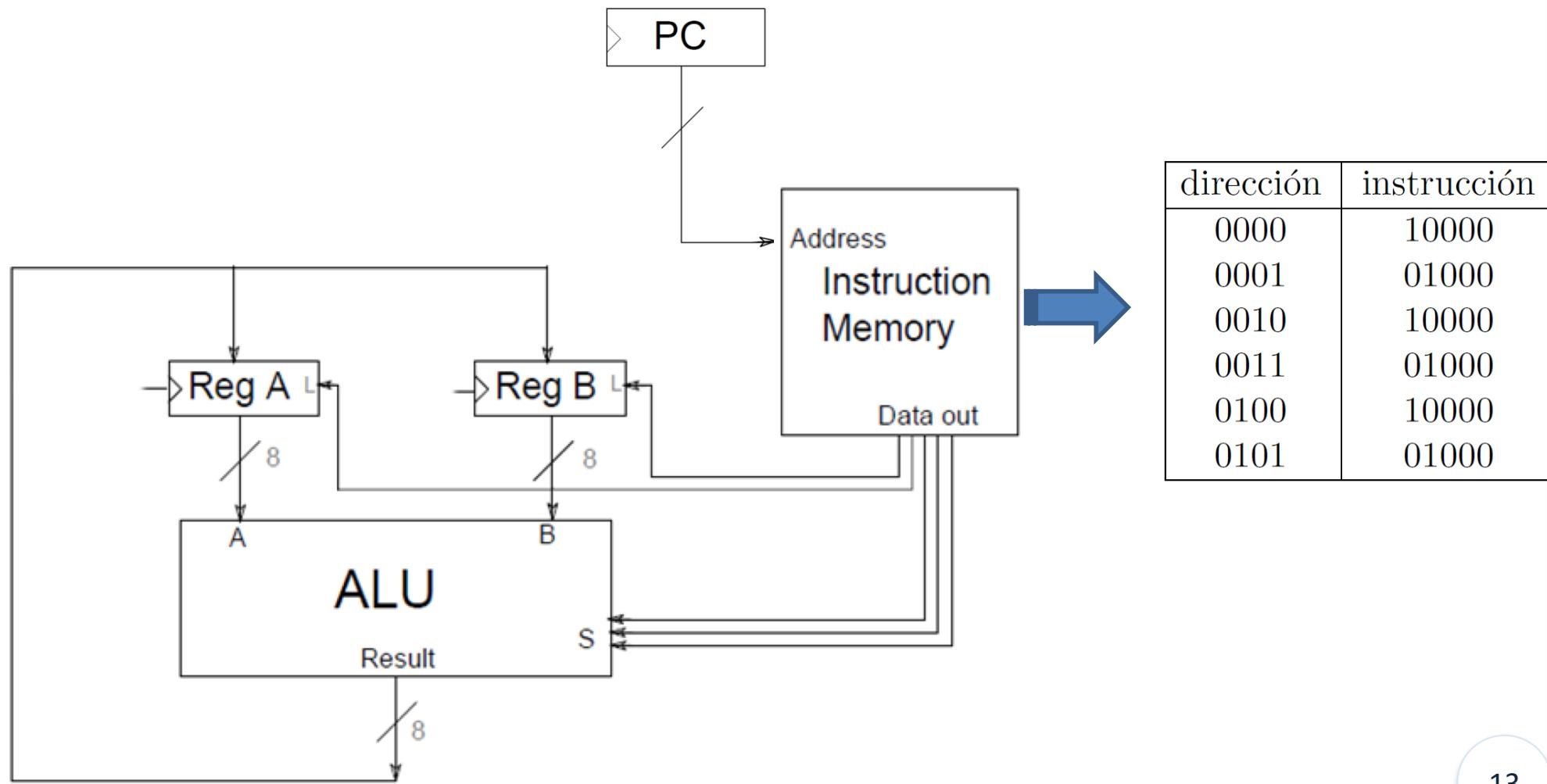
Necesitamos que la lectura/ejecución de las instrucciones sea secuencial

... es decir, que por la salida *Data out* vayan saliendo las instrucciones en el mismo orden en que están escritas en el programa



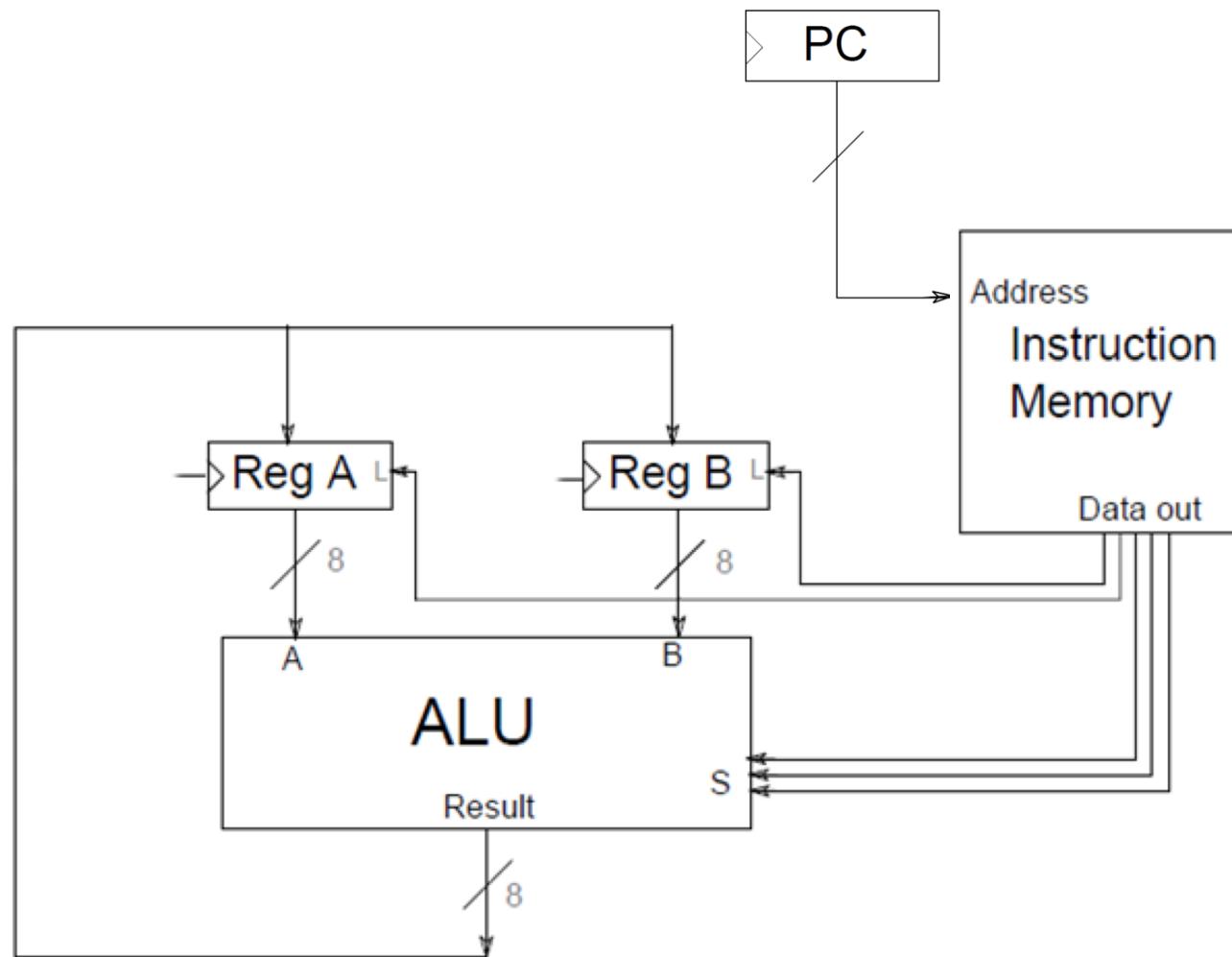
El registro **PC** —o *program counter* (o *instruction pointer*)— almacena una dirección de memoria,

... tal que al conectarse a la entrada *Address* de la memoria, la instrucción que está en esa dirección es seleccionada y puesta en *Data out*

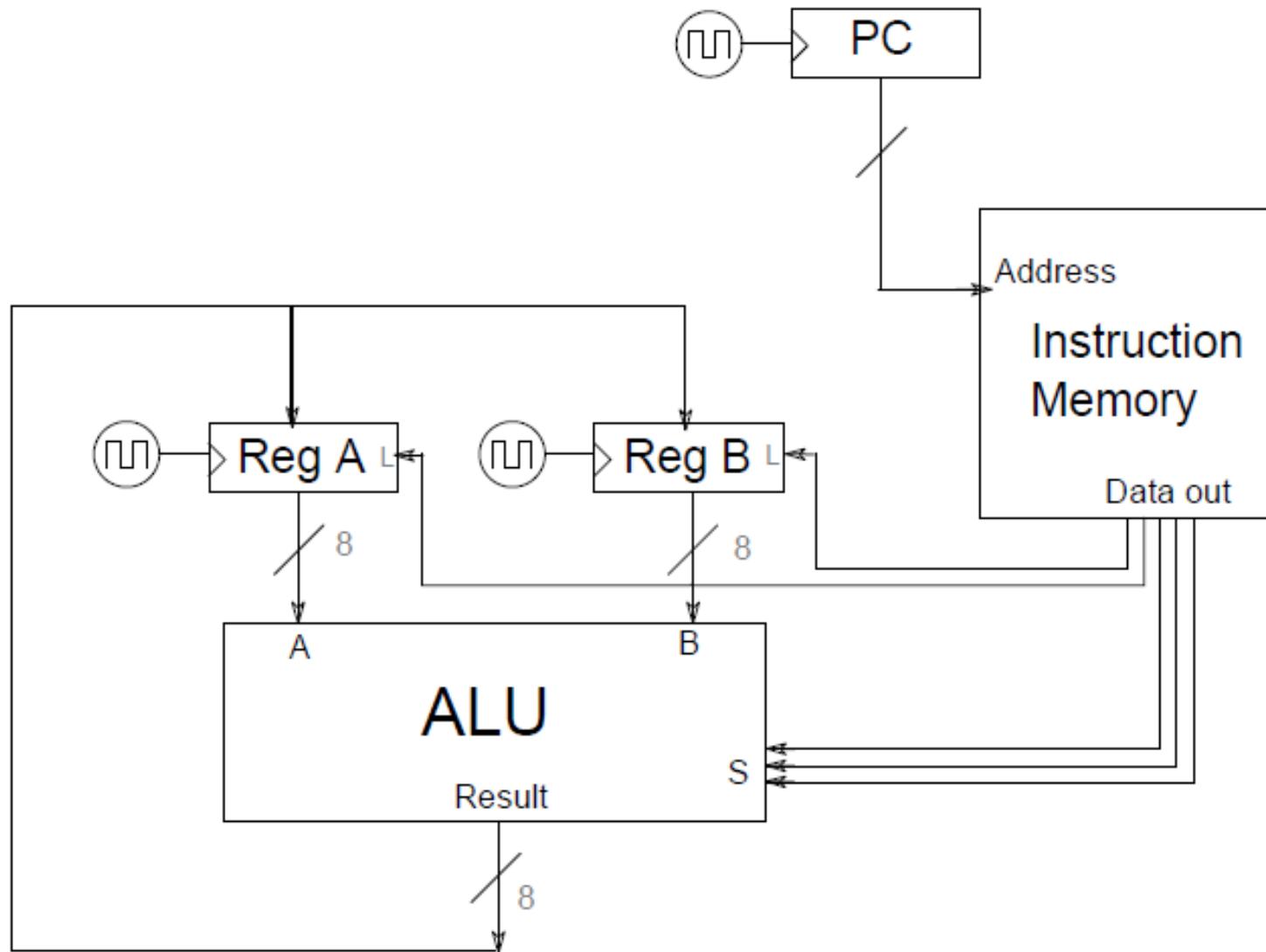


Finalmente, necesitamos que el registro *PC* vaya incrementando automáticamente su contenido,

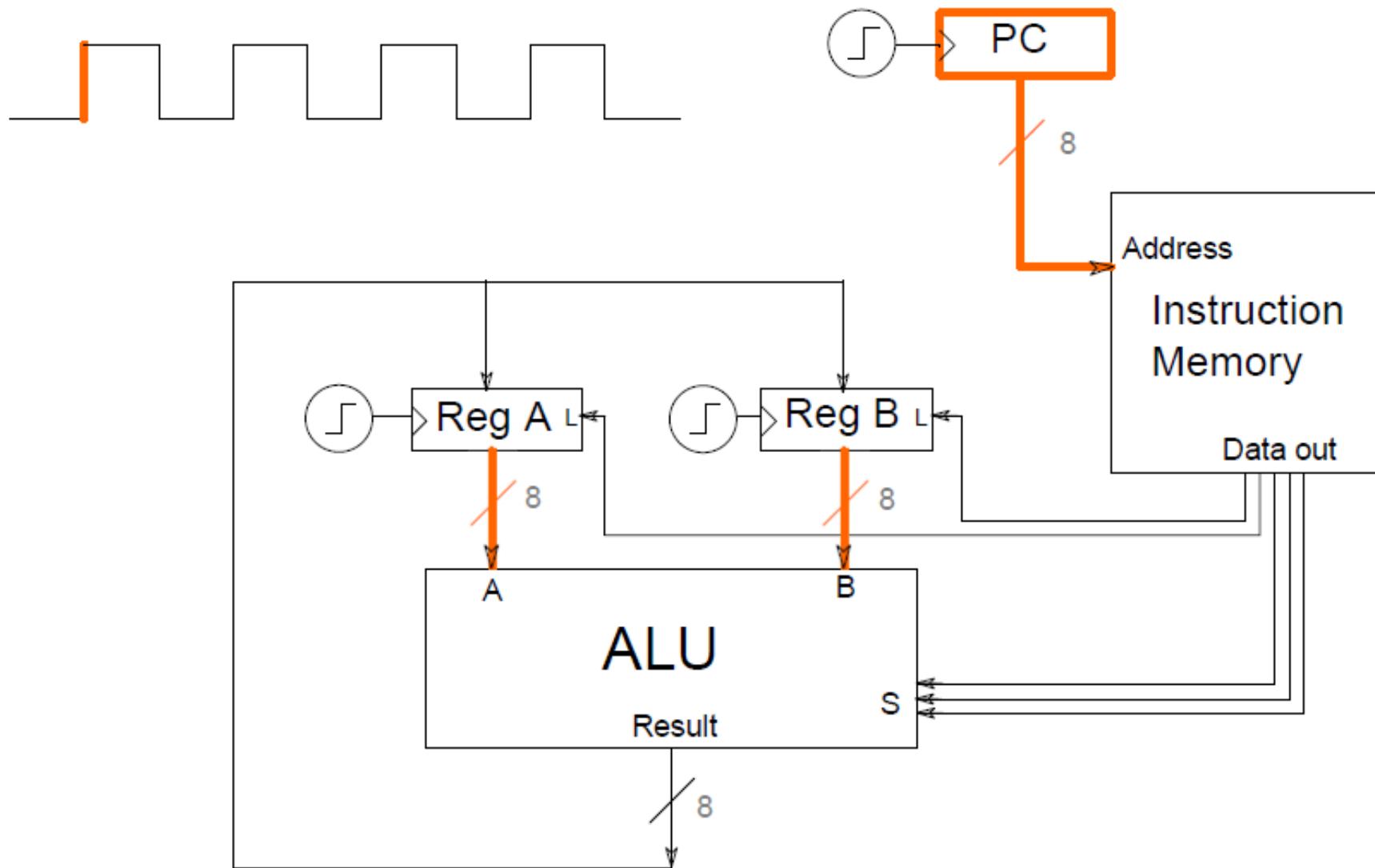
... para que el programa se ejecute por completo sin más intervención nuestra que a la partida

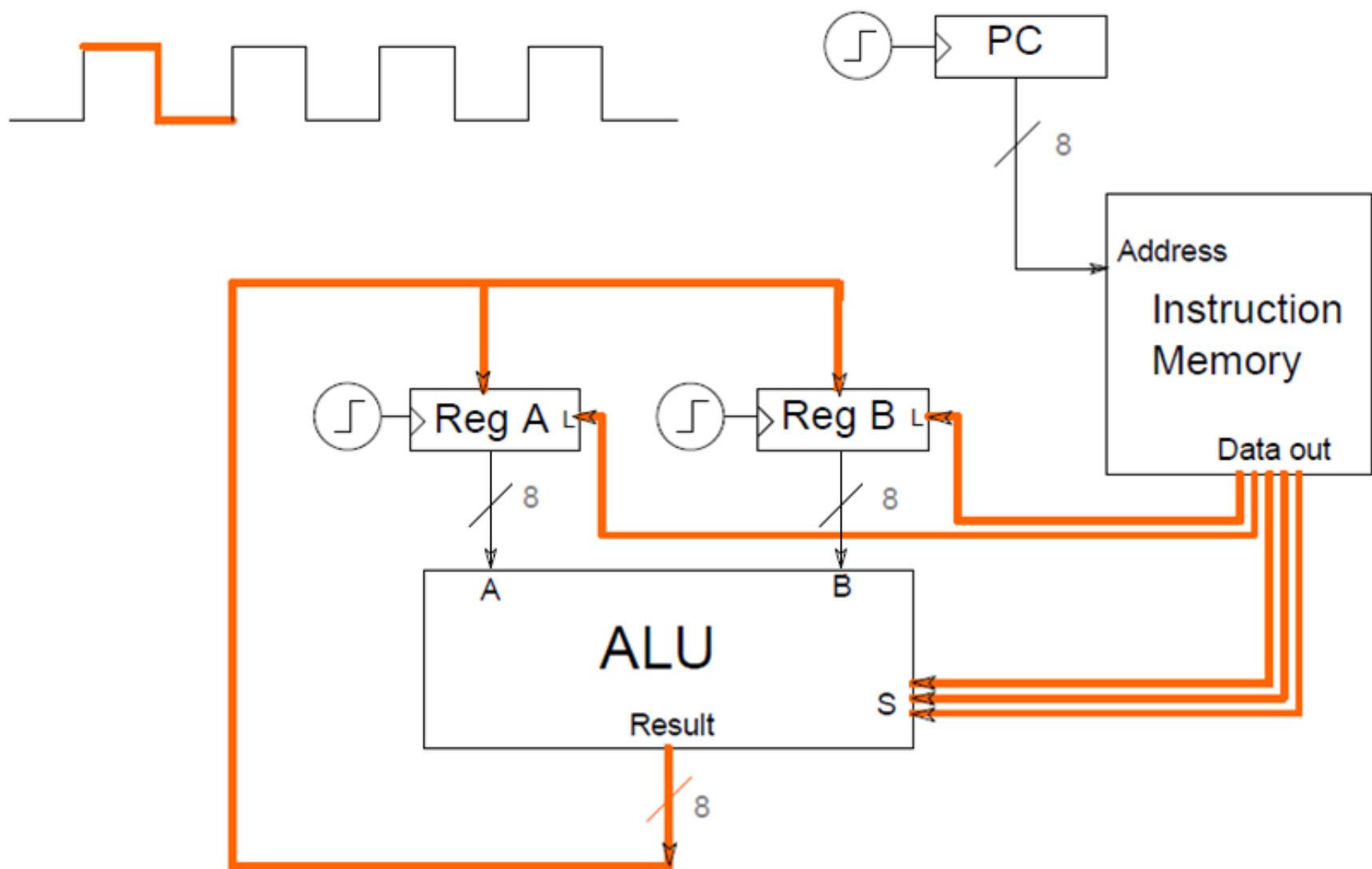


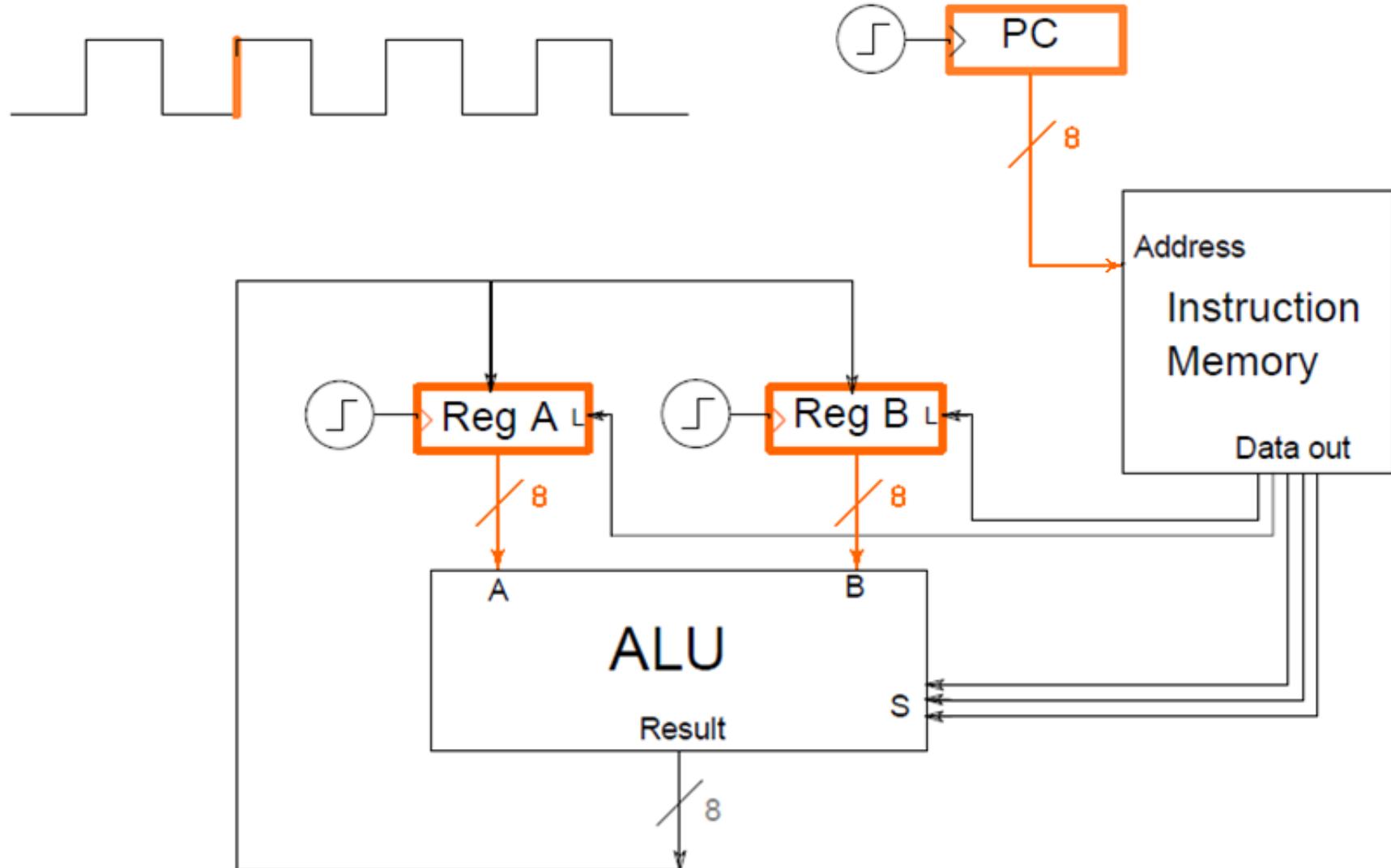
... y que todas las acciones individuales ocurran sincrónicamente:
Incluimos un *reloj* (*clock*)



Reloj: Circuito que emite una serie de pulsos con un ancho preciso y un intervalo preciso entre pulsos consecutivos, y cuya frecuencia es controlada por un oscilador de cristal

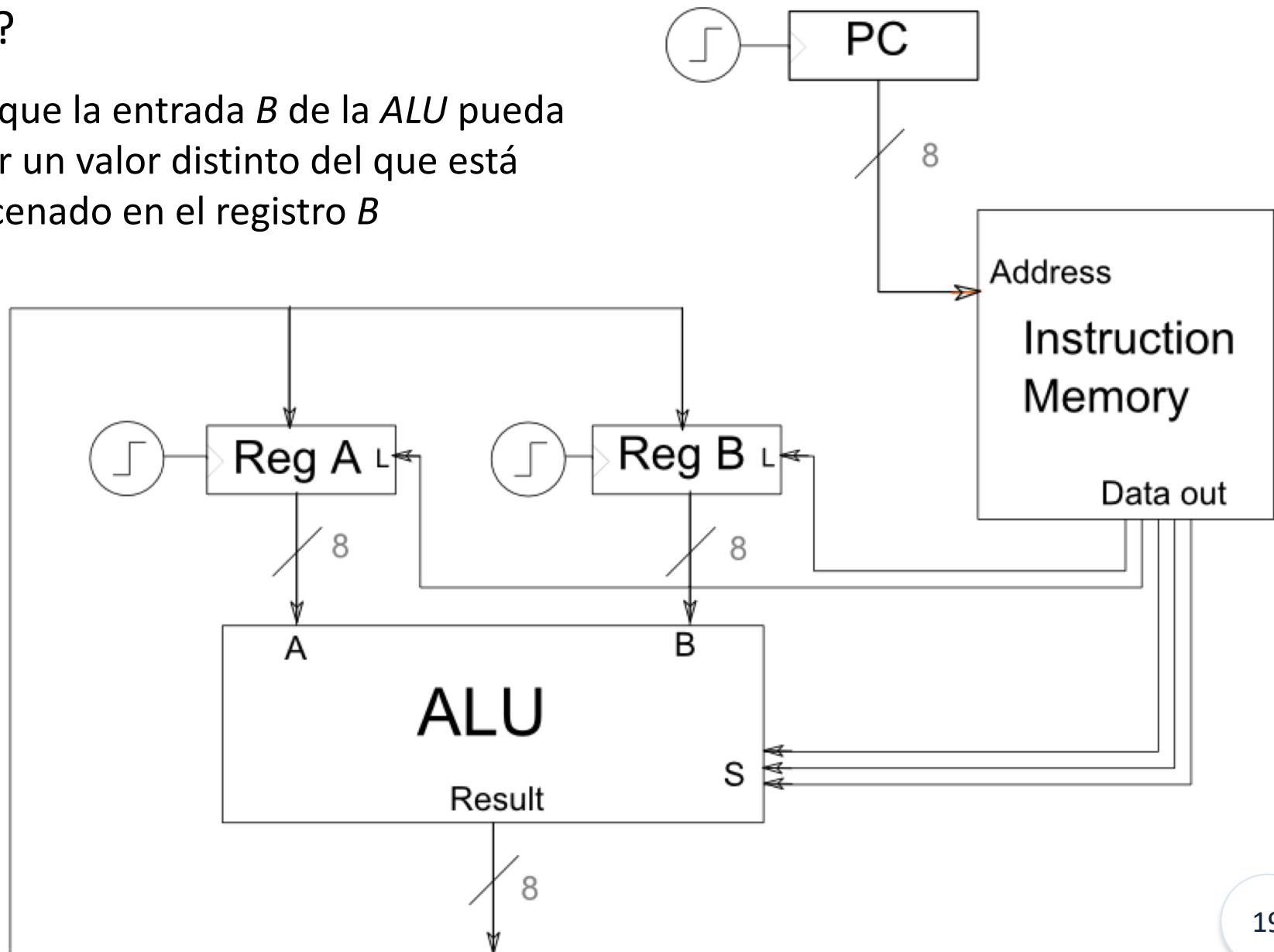






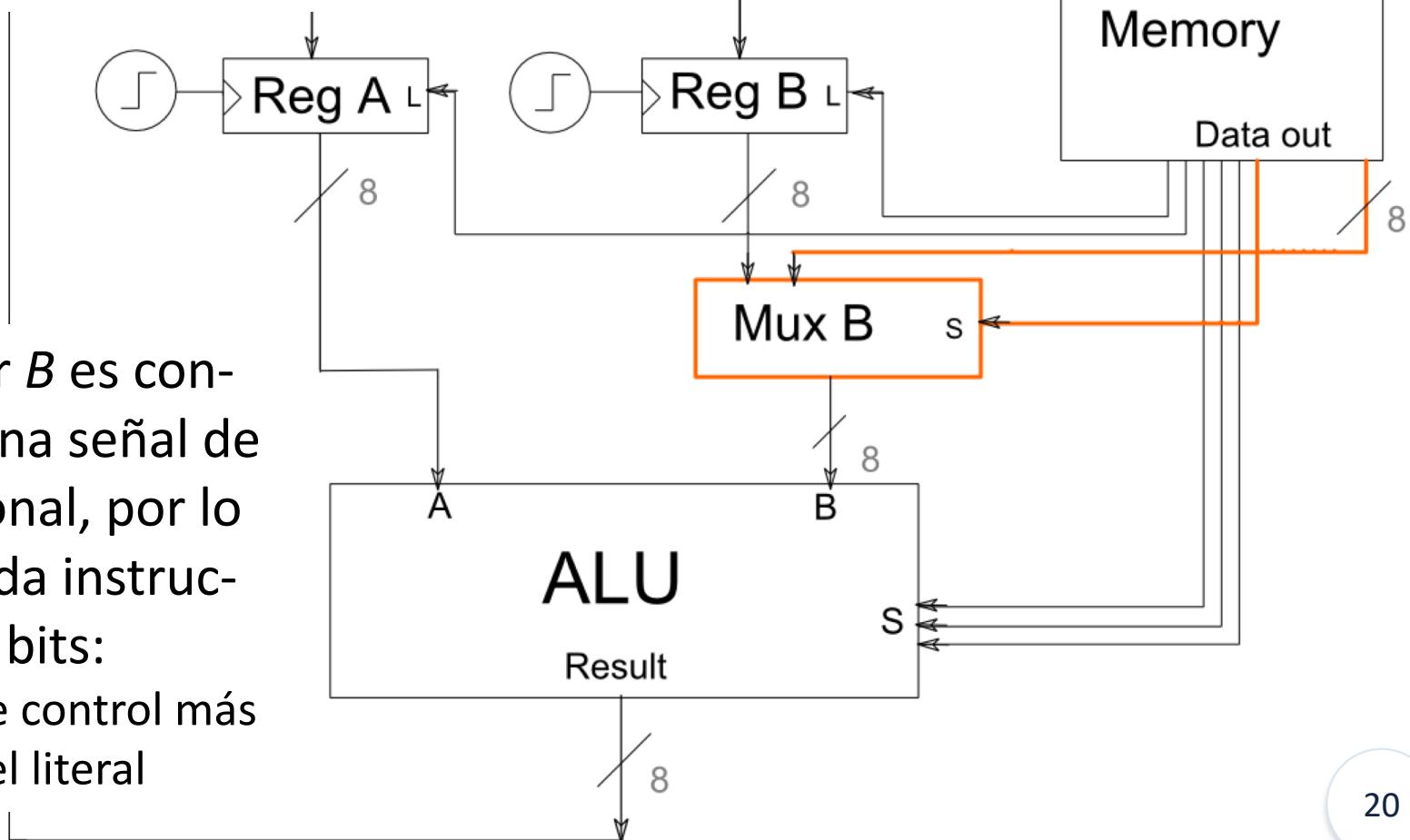
¿Cómo podemos hacer para independizarnos de los valores iniciales en los registros?

- p.ej., que la entrada *B* de la *ALU* pueda recibir un valor distinto del que está almacenado en el registro *B*



Agregamos a las instrucciones un campo de 8 bits para representar *literales*

... y usamos un multiplexor para decidir si el valor que va a la entrada *B* de la *ALU* es el contenido del registro *B* o el literal que viene en la instrucción

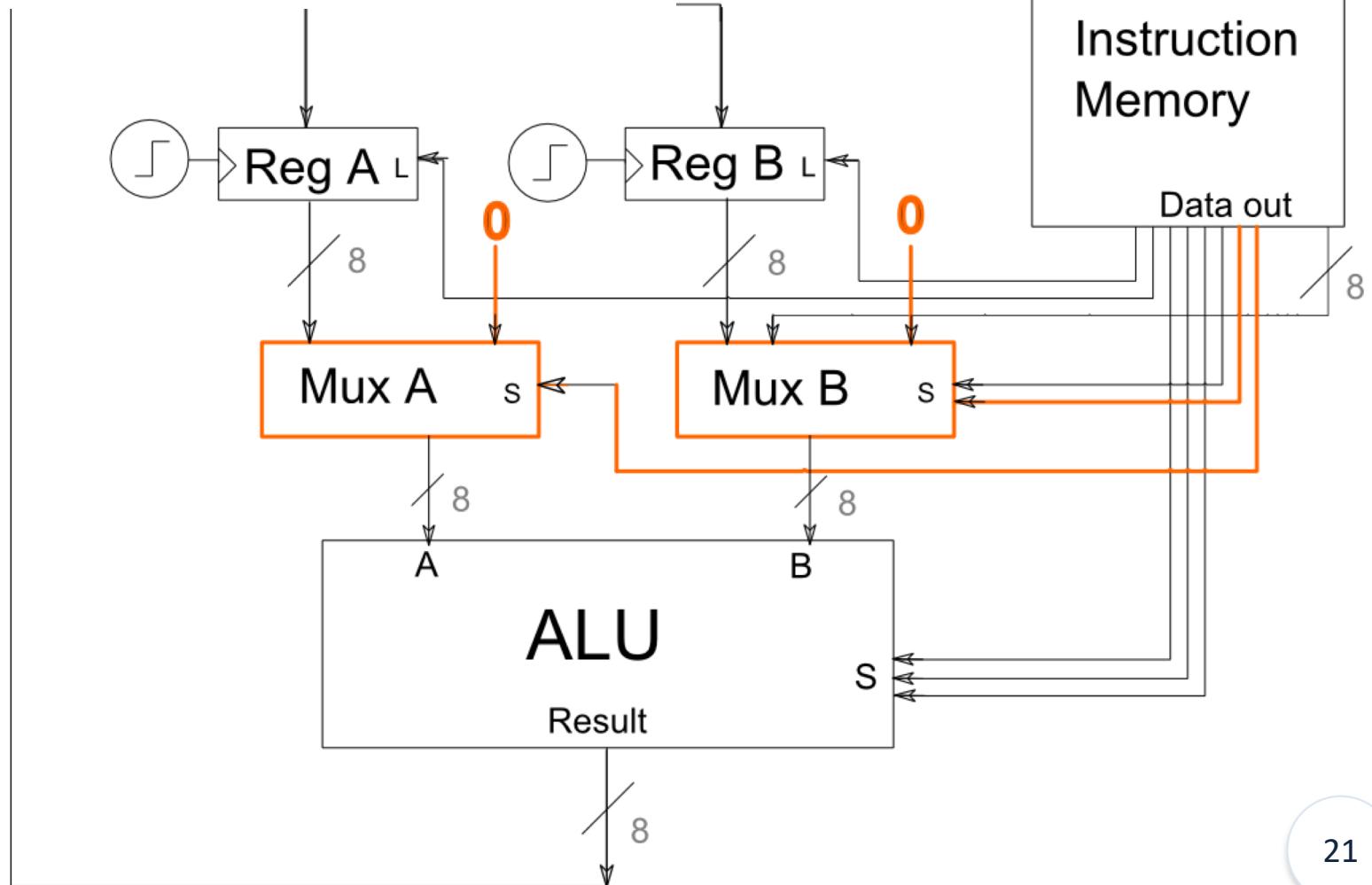


El multiplexor *B* es controlado por una señal de control adicional, por lo que ahora cada instrucción tiene 14 bits:

- 6 señales de control más 8 bits para el literal

Extendemos el uso de los multiplexores para permitir poner el valor 0 (muy común) en las entradas A o B

Estos dos casos, y el de la diapositiva anterior, exigen señales de control adicionales



Hay 8 señales de control,
permitiendo $2^8 = 256$
instrucciones posibles

... pero solo tenemos 28
instrucciones

¿Cómo podemos ahorrar
espacio en la memoria
de instrucciones?

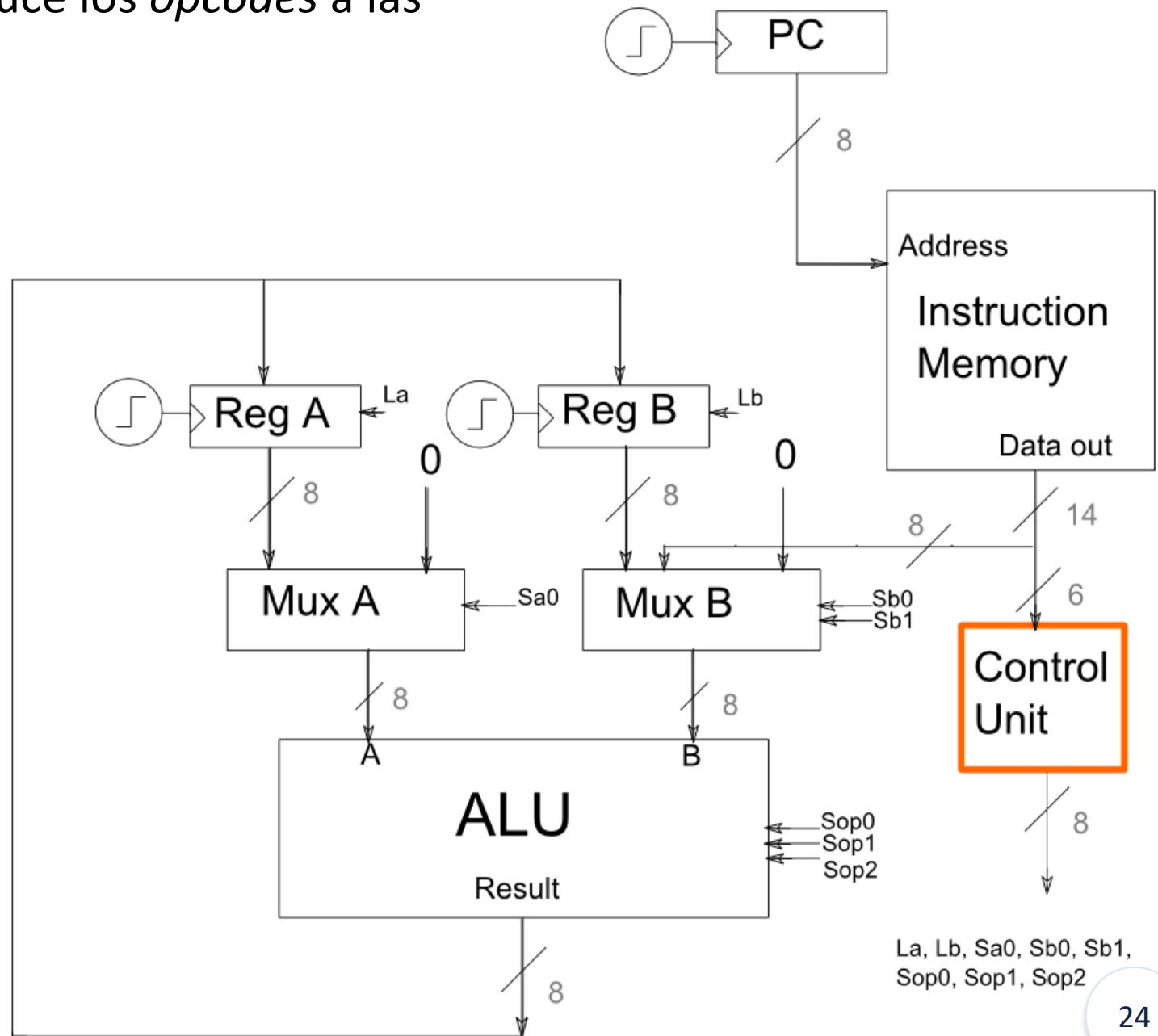
La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
1	0	1	0	0	0	0	0	A=B
0	1	0	1	1	0	0	0	B=A
1	0	0	0	1	0	0	0	A=Lit
0	1	0	0	1	0	0	0	B=Lit
1	0	0	0	0	0	0	0	A=A+B
0	1	0	0	0	0	0	0	B=A+B
1	0	0	0	1	0	0	0	A=A+Lit
1	0	0	0	0	0	0	1	A=A-B
0	1	0	0	0	0	0	1	B=A-B
1	0	0	0	1	0	0	1	A=A-Lit
1	0	0	0	0	0	1	0	A=A and B
0	1	0	0	0	0	1	0	B=A and B
1	0	0	0	1	0	1	0	A=A and Lit
1	0	0	0	0	0	1	1	A=A or B
0	1	0	0	0	0	1	1	B=A or B
1	0	0	0	1	0	1	1	A=A or Lit
1	0	0	0	0	1	0	0	A=notA
0	1	0	0	0	1	0	0	B=notA
1	0	0	0	1	1	0	0	A=notLit
1	0	0	0	0	1	0	1	A=A xor B
0	1	0	0	0	1	0	1	B=A xor B
1	0	0	0	1	1	0	1	A=A xor Lit
1	0	0	0	0	1	1	0	A=shift left A
0	1	0	0	0	1	1	0	B=shift left A
1	0	0	0	1	1	1	0	A=shift left Lit
1	0	0	0	0	1	1	1	A=shift right A
0	1	0	0	0	1	1	1	B=shift right A
1	0	0	0	1	1	1	1	A=shift right Lit

Usamos **opcodes**, cada uno asociado a una instrucción:

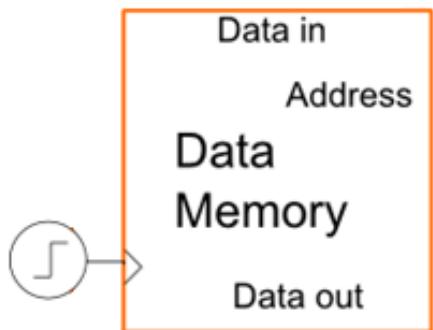
- es decir, numeramos (en binario) las instrucciones correlativamente y usamos estos números como identificadores de las instrucciones

Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	0	0	1	0	0	0	A=Lit
000011	0	1	0	0	1	0	0	0	B=Lit
000100	1	0	0	0	0	0	0	0	A=A+B
000101	0	1	0	0	0	0	0	0	B=A+B
000110	1	0	0	0	1	0	0	0	A=A+Lit
000111	1	0	0	0	0	0	0	1	A=A-B
001000	0	1	0	0	0	0	0	1	B=A-B
001001	1	0	0	0	1	0	0	1	A=A-Lit
001010	1	0	0	0	0	0	1	0	A=A and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=notA
010001	0	1	0	0	0	1	0	0	B=notA
010010	1	0	0	0	1	1	0	0	A=notLit
010011	1	0	0	0	0	1	0	1	A=A xor B
010100	0	1	0	0	0	1	0	1	B=A xor B
010101	1	0	0	0	1	1	0	1	A=A xor Lit
010110	1	0	0	0	0	1	1	0	A=shift left A
010111	0	1	0	0	0	1	1	0	B=shift left A
011000	1	0	0	0	1	1	1	0	A=shift left Lit
011001	1	0	0	0	0	1	1	1	A=shift right A
011010	0	1	0	0	0	1	1	1	B=shift right A
011011	1	0	0	0	1	1	1	1	A=shift right Lit

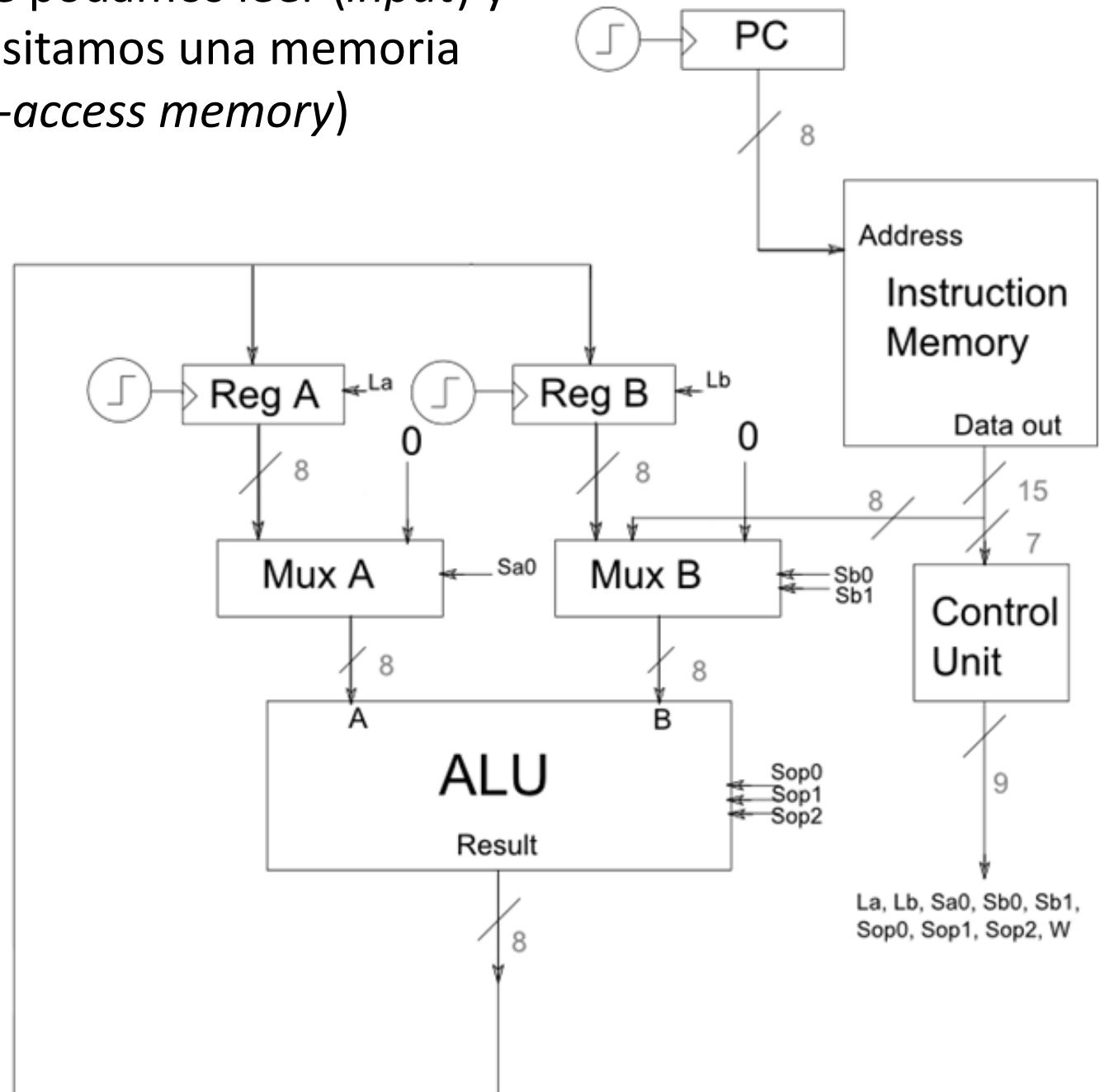
Una **unidad de control** (*CU*), implementada en una ROM, traduce los *opcodes* a las señales de control



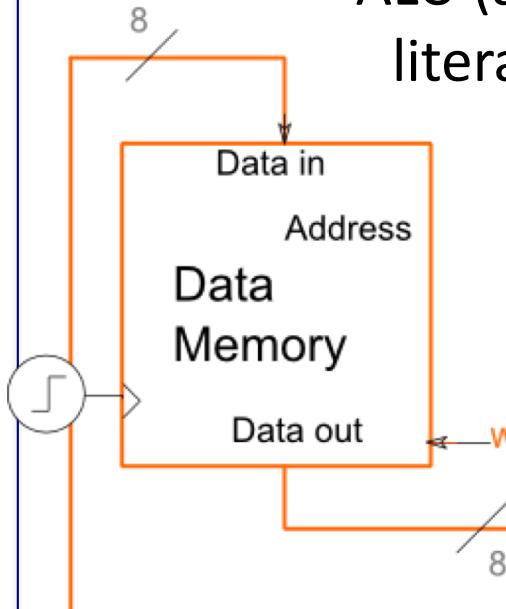
Para poder manejar una (mucho) mayor cantidad de datos, que podamos leer (*input*) y escribir (*output*), necesitamos una memoria de tipo **RAM** (*random-access memory*)



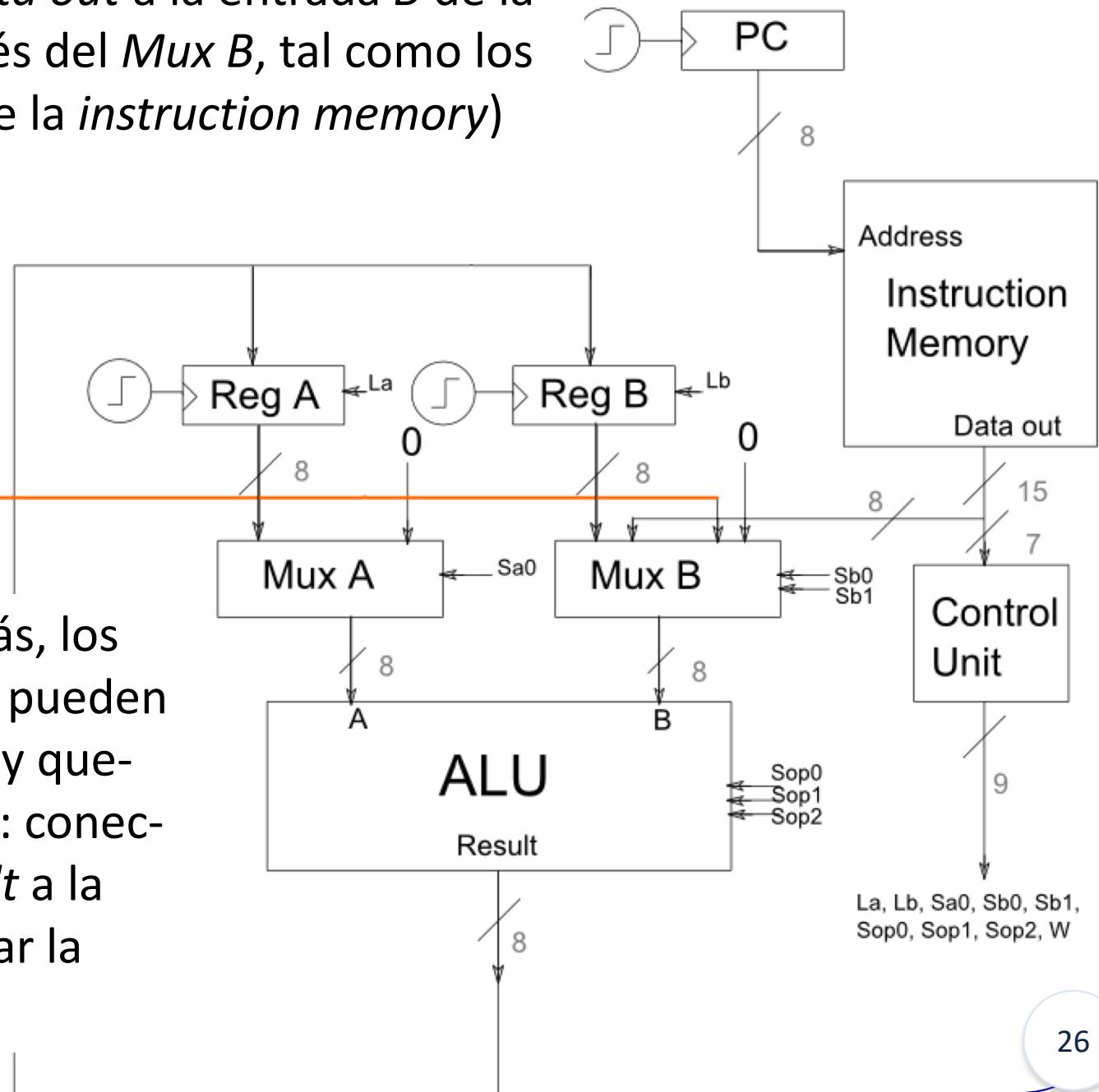
Esta *data memory* es similar a la *instruction memory*, pero además de la entrada *Address* tiene una entrada *Data in*



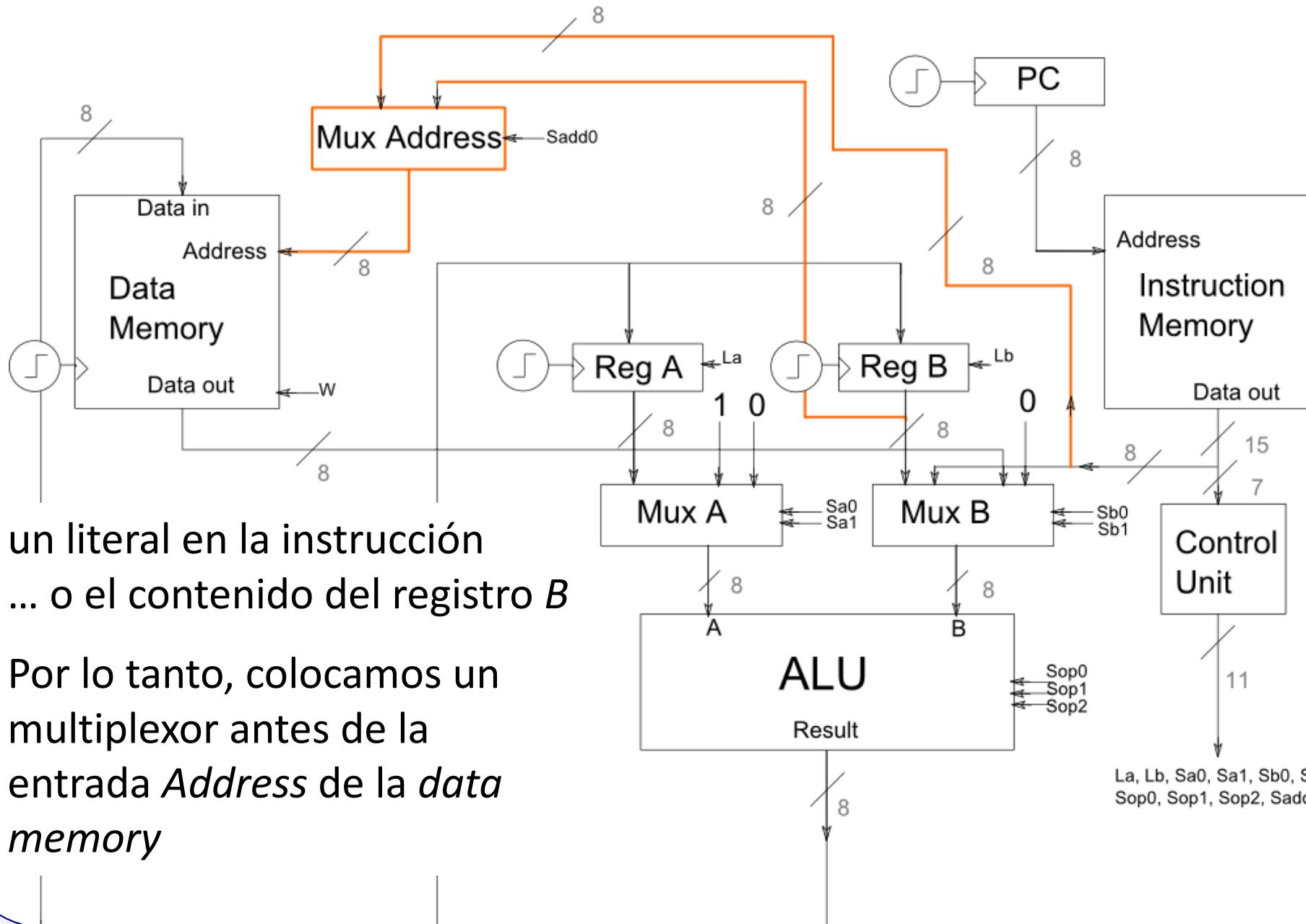
Los datos de la *data memory* van desde la salida *Data out* a la entrada *B* de la ALU (a través del *Mux B*, tal como los literales de la *instruction memory*)



... pero ahora, además, los resultados de la ALU pueden ir a la *data memory*, y quedar almacenados allí: conectamos la salida *Result* a la entrada *Data in*; notar la señal de control *w*



La dirección de la palabra de la *data memory* que queremos leer ($w = 0$) o escribir ($w = 1$) se puede especificar de dos maneras:



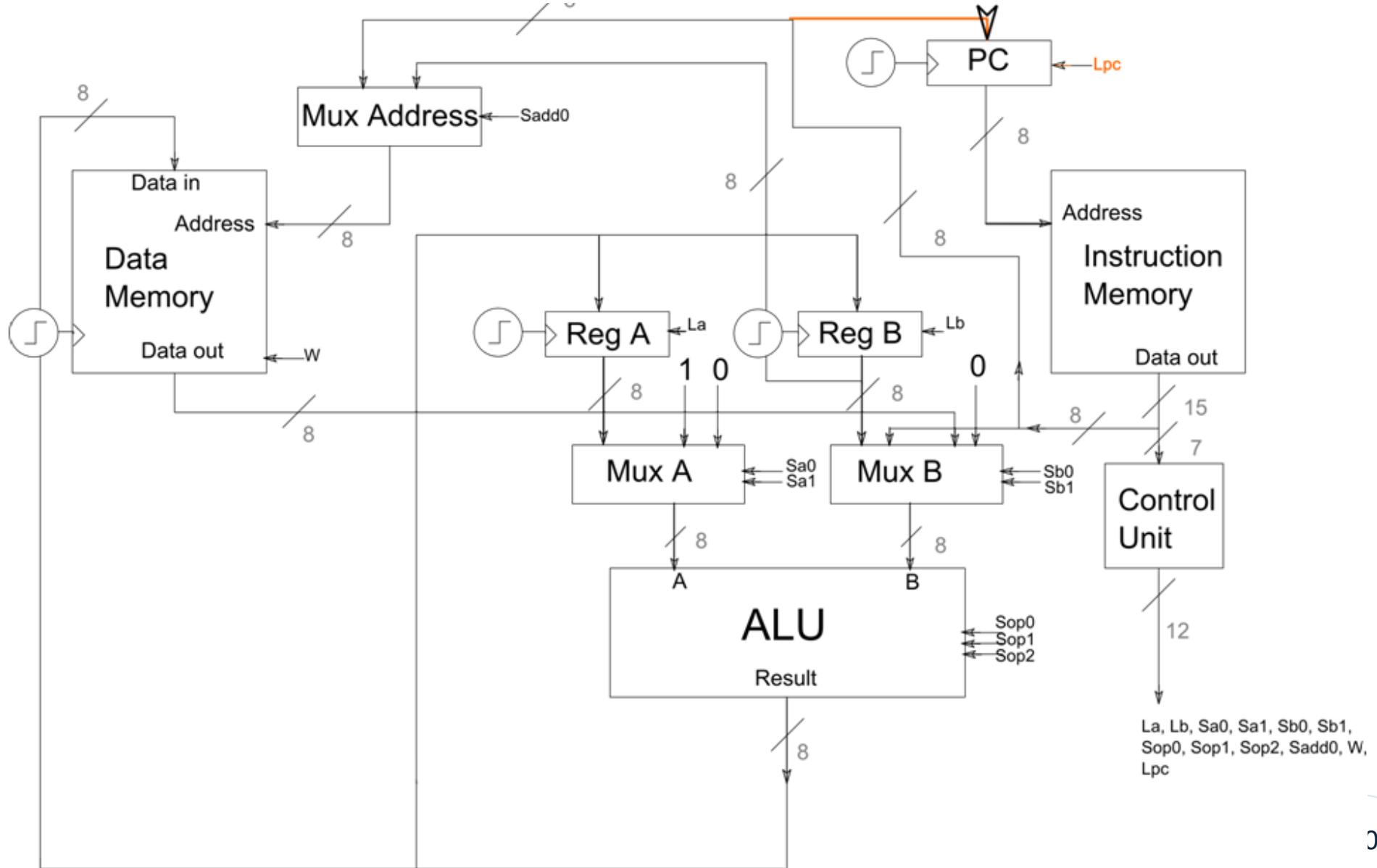
Así va nuestro lenguaje de máquina para el computador básico

Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
000000	1	0	1	0	0	0	0	0	A=B
000001	0	1	0	1	1	0	0	0	B=A
000010	1	0	1	0	1	0	0	0	A=Lit
000011	0	1	1	0	1	0	0	0	B=Lit
000100	1	0	0	0	0	0	0	0	A=A+B
000101	0	1	0	0	0	0	0	0	B=A+B
000110	1	0	0	0	1	0	0	0	A=A+Lit
000111	1	0	0	0	0	0	0	1	A=A-B
001000	0	1	0	0	0	0	0	1	B=A-B
001001	1	0	0	0	1	0	0	1	A=A-Lit
001010	1	0	0	0	0	0	1	0	A=A and B
001011	0	1	0	0	0	0	1	0	B=A and B
001100	1	0	0	0	1	0	1	0	A=A and Lit
001101	1	0	0	0	0	0	1	1	A=A or B
001110	0	1	0	0	0	0	1	1	B=A or B
001111	1	0	0	0	1	0	1	1	A=A or Lit
010000	1	0	0	0	0	1	0	0	A=notA
010001	0	1	0	0	0	1	0	0	B=notA
010010	1	0	0	0	1	1	0	0	A=notLit
010011	1	0	0	0	0	1	0	1	A=A xor B
010100	0	1	0	0	0	1	0	1	B=A xor B
010101	1	0	0	0	1	1	0	1	A=A xor Lit
010110	1	0	0	0	0	1	1	0	A=shift left A
010111	0	1	0	0	0	1	1	0	B=shift left A
011000	1	0	0	0	1	1	1	0	A=shift left Lit
011001	1	0	0	0	0	1	1	1	A=shift right A
011010	0	1	0	0	0	1	1	1	B=shift right A
011011	1	0	0	0	1	1	1	1	A=shift right Lit

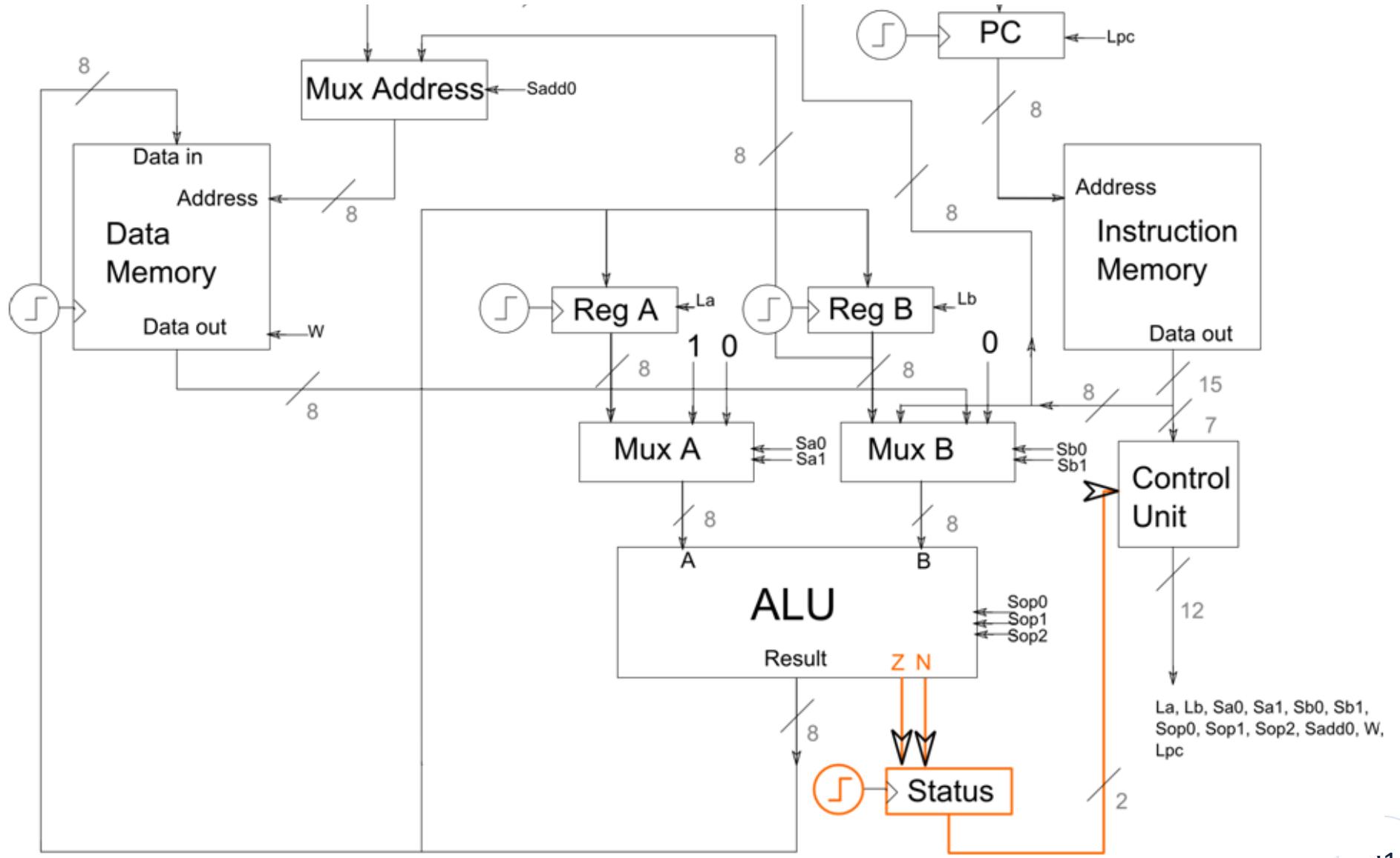
... y su conversión a un lenguaje assembly básico

Instrucción	Operandos	Opcode	La	Lb	Sa0	Sb0	Sb1	Sop2	Sop1	Sop0	Operación
MOV	A,B	000000	1	0	1	0	0	0	0	0	A=B
	B,A	000001	0	1	0	1	1	0	0	0	B=A
	A,Lit	000010	1	0	0	0	1	0	0	0	A=Lit
	B,Lit	000011	0	1	0	0	1	0	0	0	B=Lit
ADD	A,B	000100	1	0	0	0	0	0	0	0	A=A+B
	B,A	000101	0	1	0	0	0	0	0	0	B=A+B
	A,Lit	000110	1	0	0	0	1	0	0	0	A=A+Lit
SUB	A,B	000111	1	0	0	0	0	0	0	1	A=A-B
	B,A	001000	0	1	0	0	0	0	0	1	B=A-B
	A,Lit	001001	1	0	0	0	1	0	0	1	A=A-Lit
AND	A,B	001010	1	0	0	0	0	0	1	0	A=A and B
	B,A	001011	0	1	0	0	0	0	1	0	B=A and B
	A,Lit	001100	1	0	0	0	1	0	1	0	A=A and Lit
OR	A,B	001101	1	0	0	0	0	0	1	1	A=A or B
	B,A	001110	0	1	0	0	0	0	1	1	B=A or B
	A,Lit	001111	1	0	0	0	1	0	1	1	A=A or Lit
NOT	A,A	010000	1	0	0	0	0	1	0	0	A=notA
	B,A	010001	0	1	0	0	0	1	0	0	B=notA
	A,Lit	010010	1	0	0	0	1	1	0	0	A=notLit
XOR	A,A	010011	1	0	0	0	0	1	0	1	A=A xor B
	B,A	010100	0	1	0	0	0	1	0	1	B=A xor B
	A,Lit	010101	1	0	0	0	1	1	0	1	A=A xor Lit
SHL	A,A	010110	1	0	0	0	0	1	1	0	A=shift left A
	B,A	010111	0	1	0	0	0	1	1	0	B=shift left A
	A,Lit	011000	1	0	0	0	1	1	1	0	A=shift left Lit
SHR	A,A	011001	1	0	0	0	0	1	1	1	A=shift right A
	B,A	011010	0	1	0	0	0	1	1	1	B=shift right A
	A,Lit	011011	1	0	0	0	1	1	1	1	A=shift right Lit

Para permitir **saltos incondicionales (jump)** es necesario poder poner en el registro **PC** la dirección de la instrucción a la que queremos ir (saltar); notemos la (nueva) señal de control L_{PC}

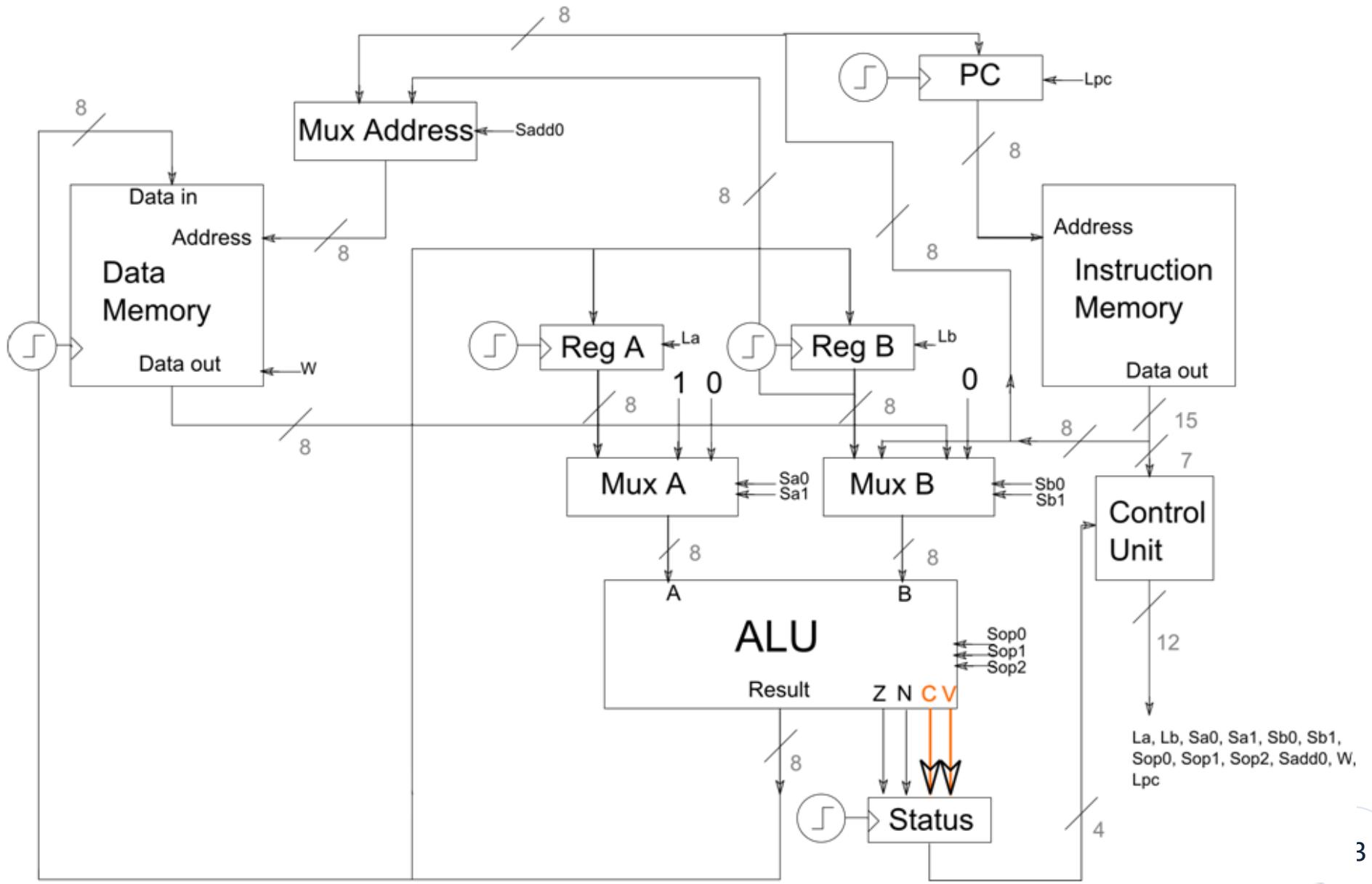


Para **saltos condicionales** (*beq, bne*) es necesario que la *CU* examine el resultado de la *ALU* —p.ej., si es cero o negativo: agregamos el registro ***Status*** entre la *ALU* y la *CU* con esta información



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

El registro *Status* puede también tener bits que indican la presencia de un *carry* o de un *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

Subrutinas, funciones, procedimientos, ..., métodos

main:

r = ... —asignar un valor a esta variable

h = ... —asignar un valor a esta variable

v = vol_cil(r, h)

print("El volumen del cilindro es", v, "cm³)

vol_cil(radio, altura):

*return PI*radio*radio*altura*

¿Qué es qué?

- la sentencia **v = vol_cil(r, h)** es la *llamada a la función*
- **r** y **h** son los *parámetros reales*; **radio** y **altura**, los *parámetros formales*
- el valor de **PI*radio*radio*altura** es el *valor de retorno*, que en el **main** va a ser asignado a la variable **v**

1) Al producirse la llamada a la función —la ejecución de la sentencia **v = vol_cil(r, h)**— el computador debe empezar a ejecutar las instrucciones de máquina correspondientes a la función

- ... mediante una instrucción de máquina equivalente a un salto incondicional, cambiando el valor del registro *PC*

2) Sin embargo, primero es necesario “pasarle” a la función **vol_cil** los valores que deben tomar los parámetros formales **radio** y **altura**, es decir, los valores de las variables **r** y **h** (los parámetros reales)

- ... hay que almacenar los valores de los parámetros reales en algún lugar de la memoria (de datos) al que la función tenga acceso

...

...

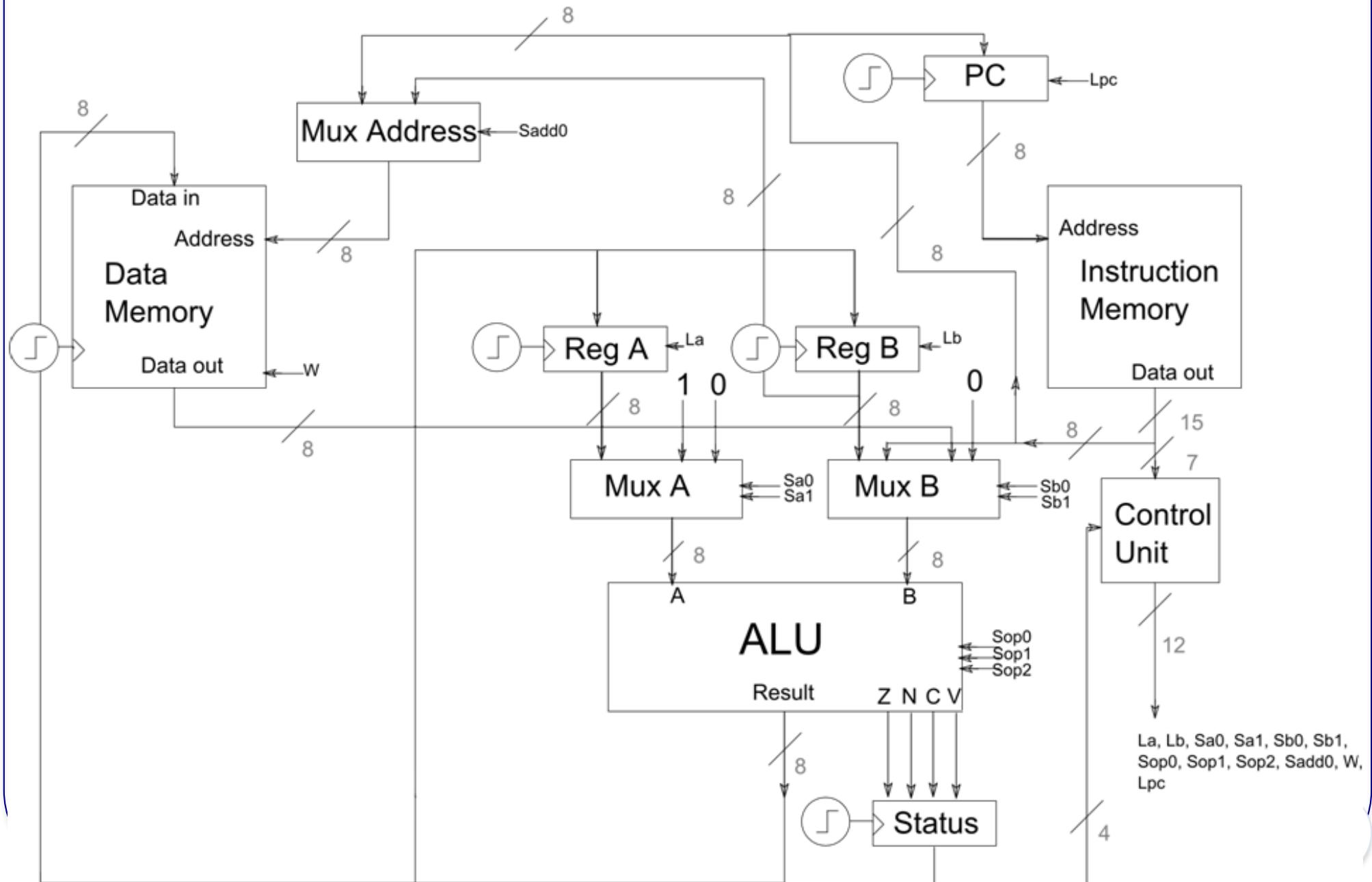
3) Finalmente, al terminar la ejecución de la función, es necesario “pasar de vuelta”, o “retornar”, el valor calculado por la función:

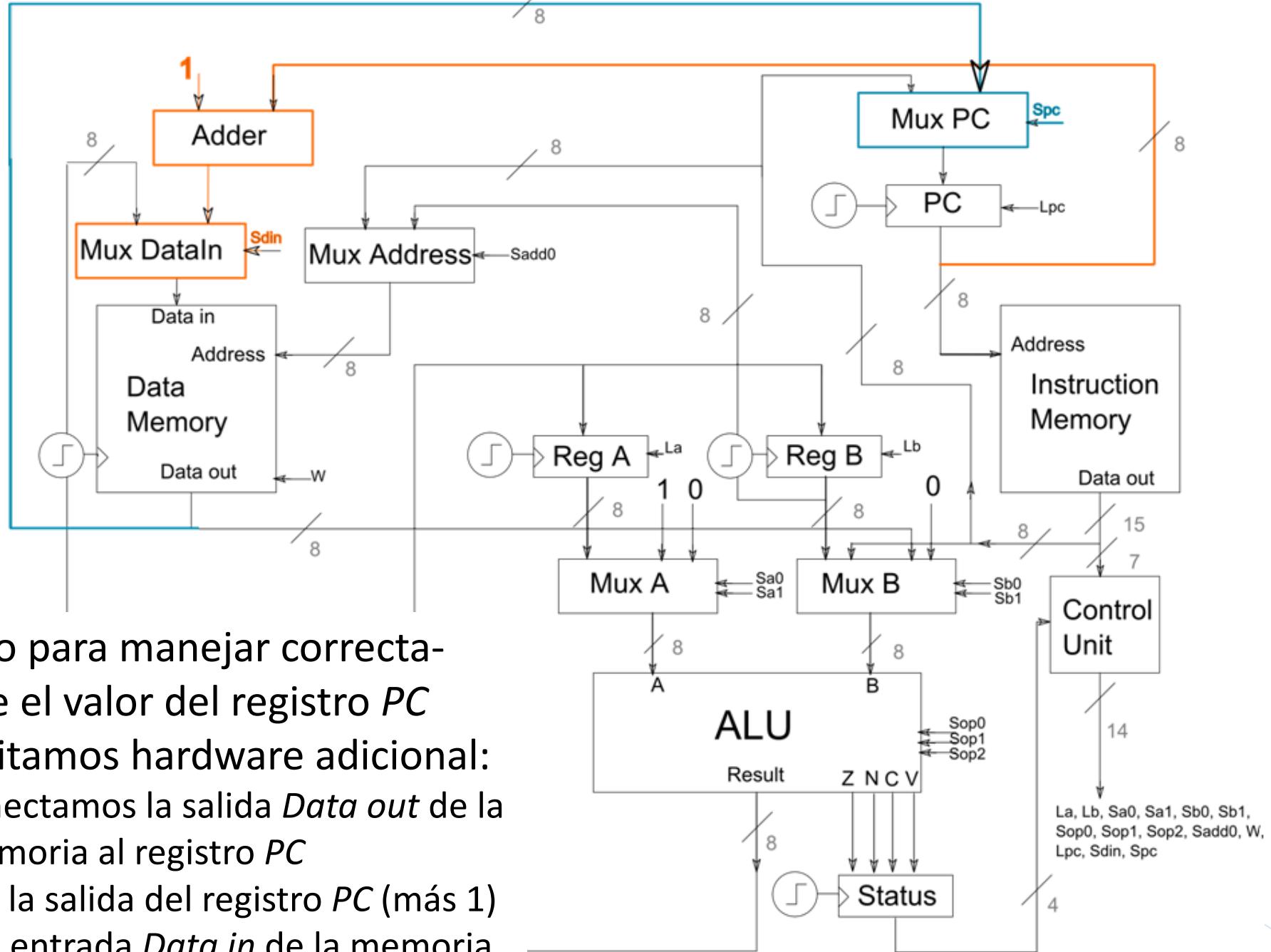
- ... usando nuevamente la memoria (de datos)

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

- retomando el valor original del registro *PC* más 1

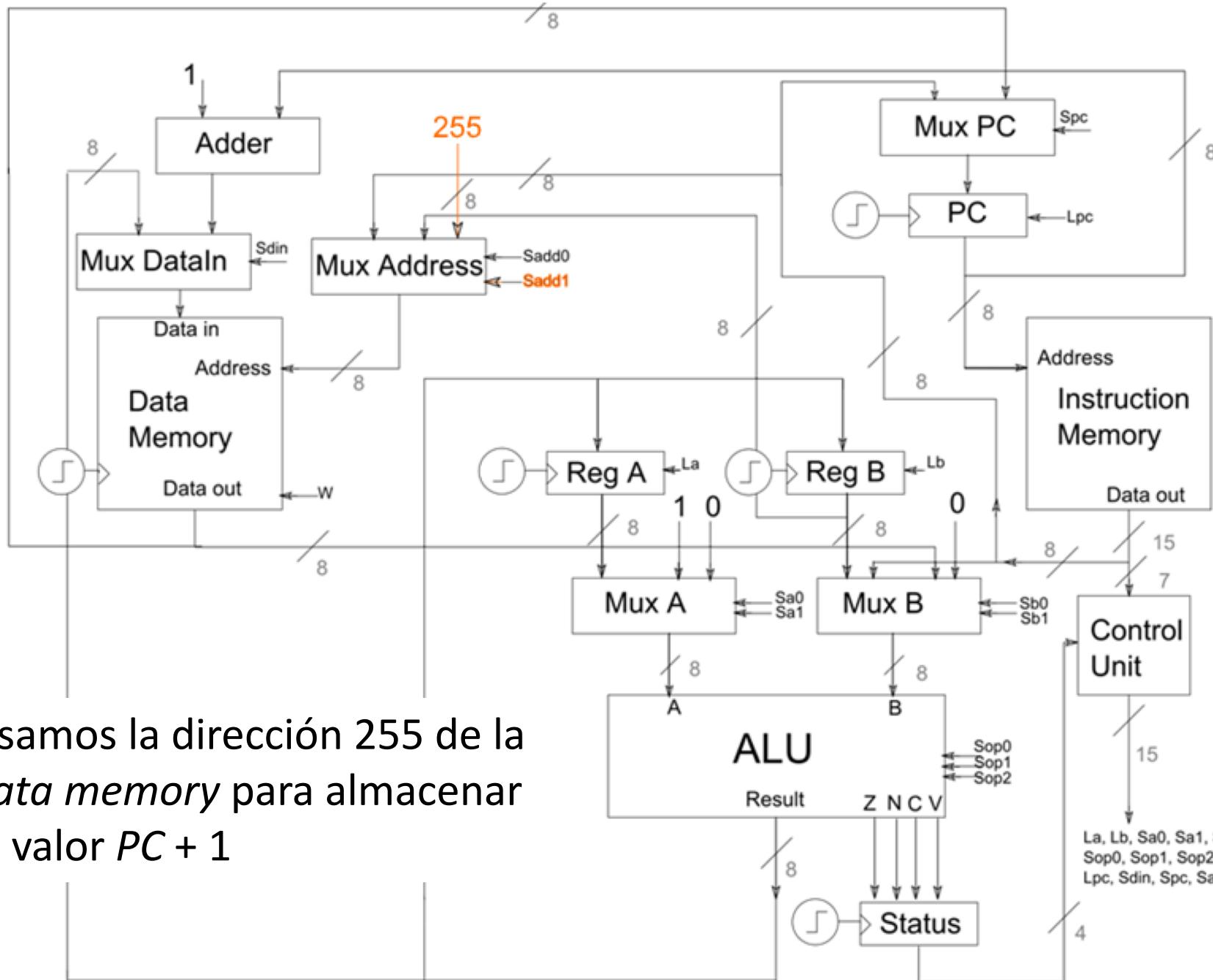
El computador básico permite pasar los parámetros y pasar de vuelta el valor de retorno: es (solo) almacenar y leer valores en la *data memory*





... 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



Usamos la dirección 255 de la *data memory* para almacenar el valor $PC + 1$