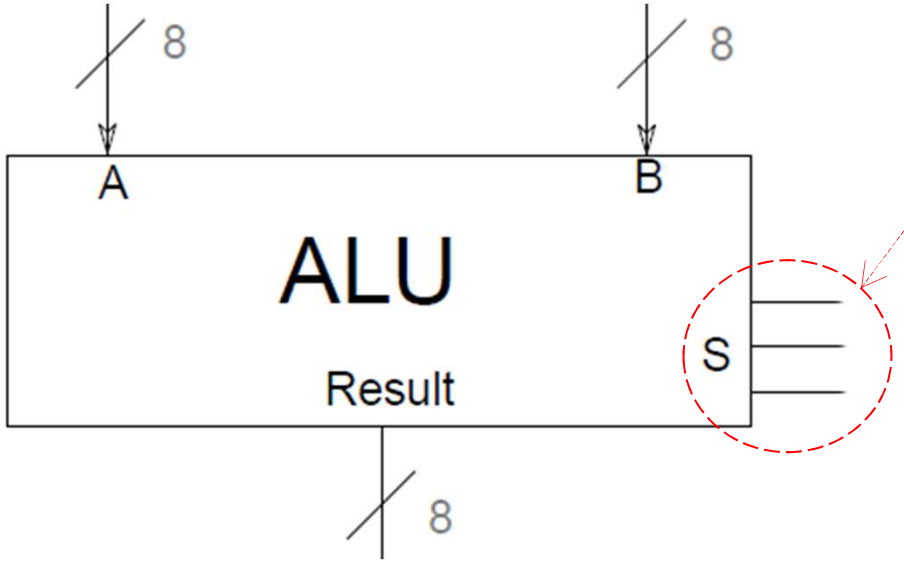
Procesador: componentes, instrucciones, *datapath*

Arquitectura de Computadores – IIC2343

2021-1 Hans Löbel, Yadran Eterovic

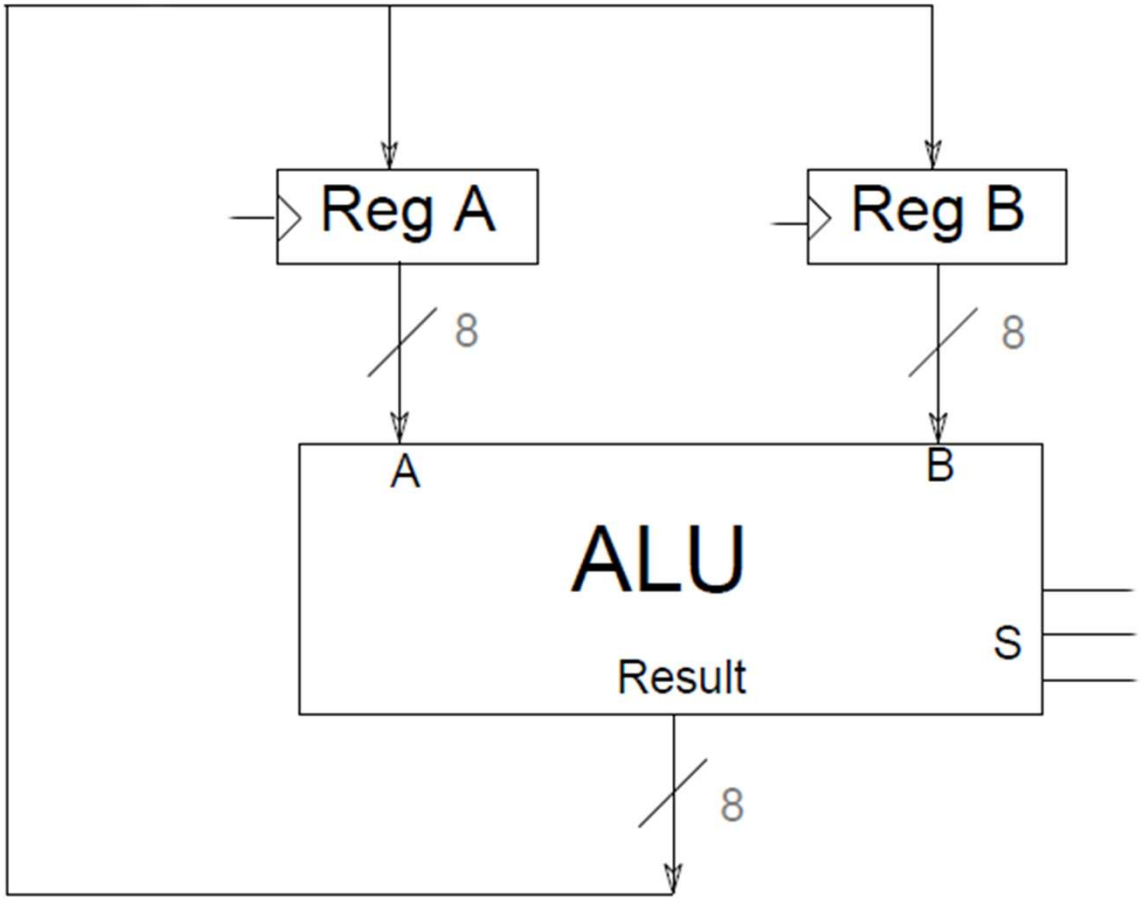
Dónde estamos



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* de la *ALU* provienen de **registros**:

- ubicaciones especiales construidas directamente en el hardware a base de flip-flops
- son los “ladrillos” de la construcción de computadores, y su número es limitado

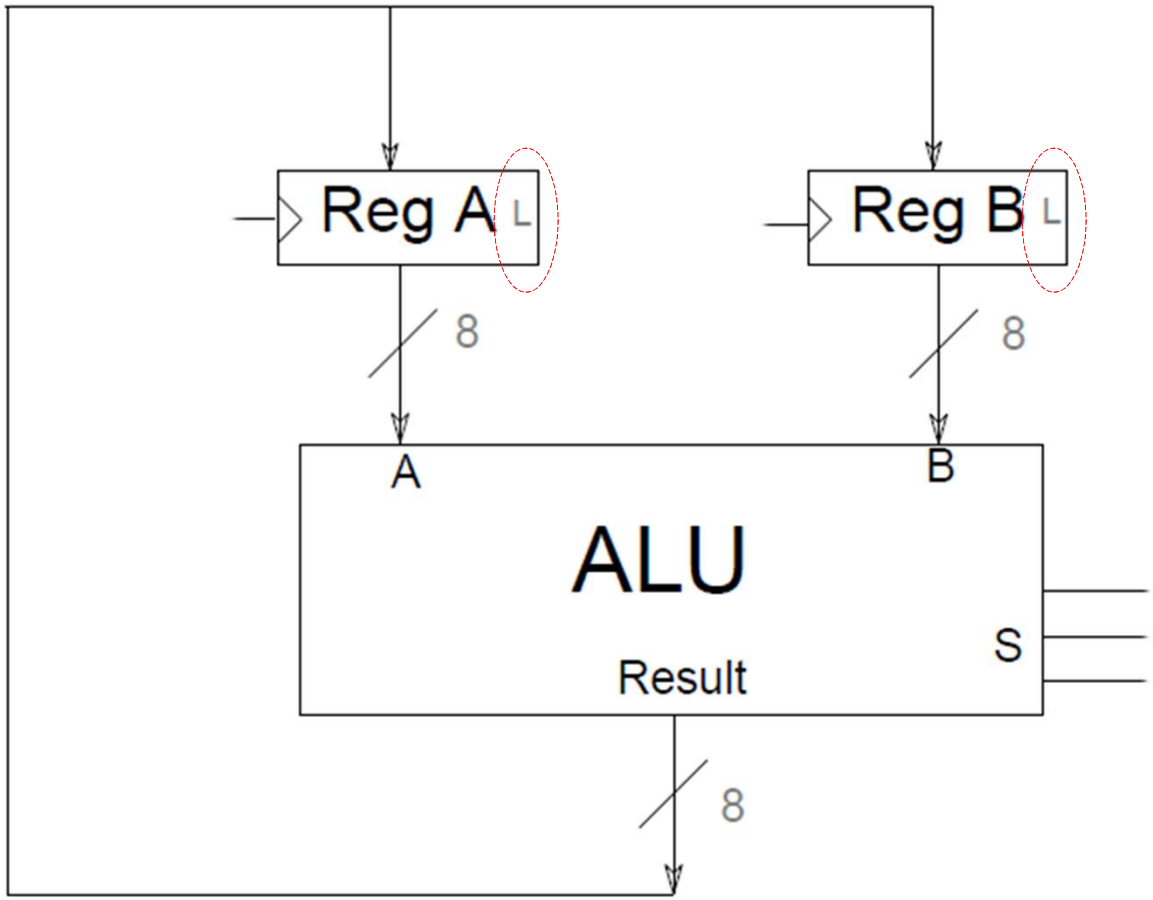


Vamos a suponer registros de 8 bits, es decir, de 8 flip-flops

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

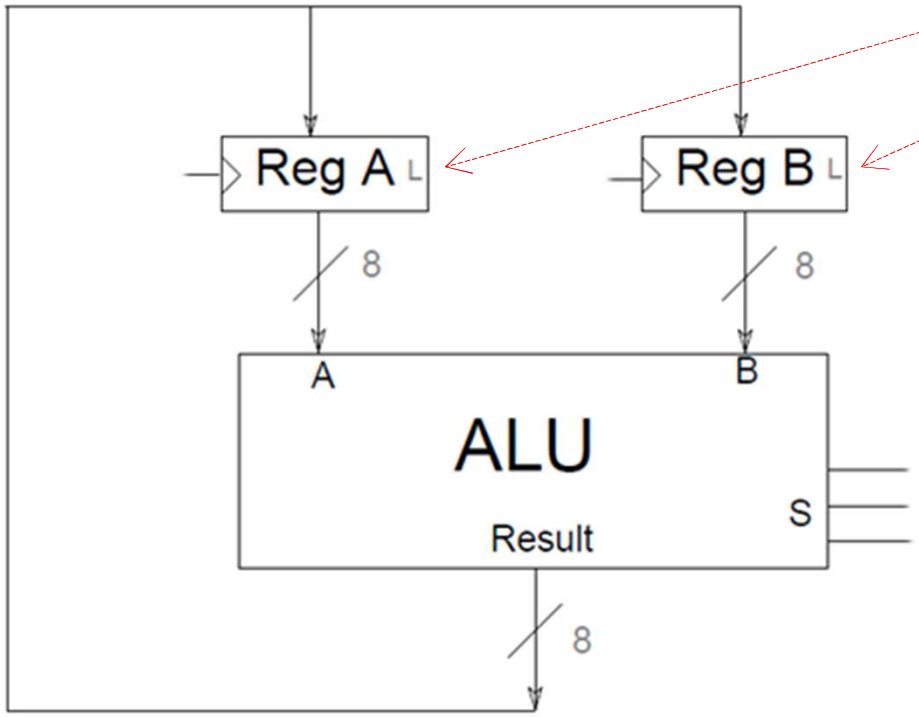
Agregamos las señales de control L_A y L_B para controlar la escritura —o actualización de los valores— de los registros:

- podemos conectar L_A y L_B directamente al input *C* (reloj) de cada registro
- … o usar L_A y L_B como uno de los inputs (*enabler*) de una AND, cuyo otro input es la señal del reloj y cuyo output va al input *C* de cada registro



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 and B
0	1	0	1	0	B=A and B
1	0	0	1	1	A=A or B
0	1	0	1	1	B=A or B
1	0	1	0	0	A=notA
0	1	1	0	0	B=notA
1	0	1	0	1	A=A xor B
0	1	1	0	1	B=A xor B
1	0	1	1	0	A=shift left A
0	1	1	1	0	B=shift left A
1	0	1	1	1	A=shift right A
0	1	1	1	1	B=shift right A

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

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 …**

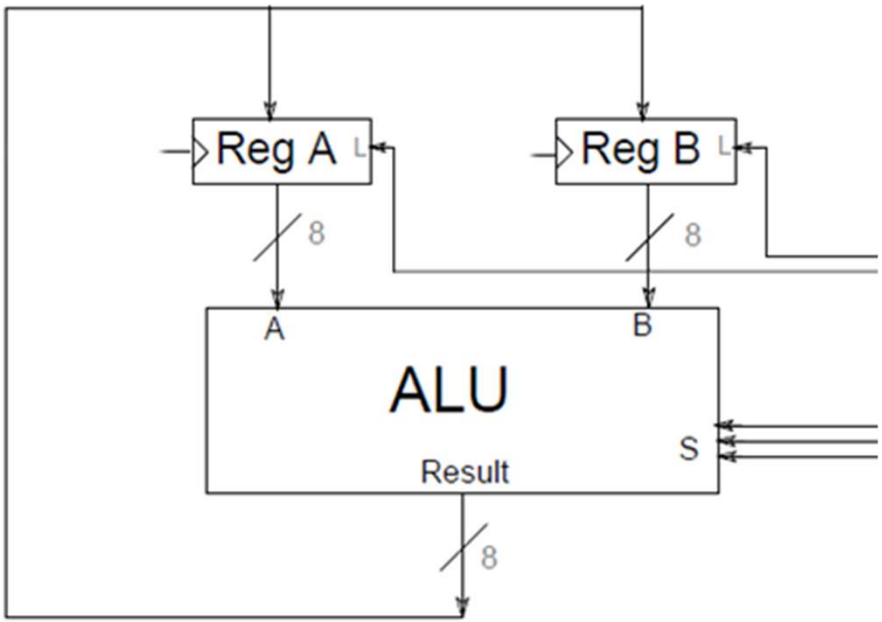
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	**0**	1
1	0	0	0	0	A=A+B	**1**	1
0	1	0	0	0	B=A+B	1	**2**
1	0	0	0	0	A=A+B	**3**	2
0	1	0	0	0	B=A+B	3	**5**
1	0	0	0	0	A=A+B	**8**	5
0	1	0	0	0	B=A+B	8	**13**

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

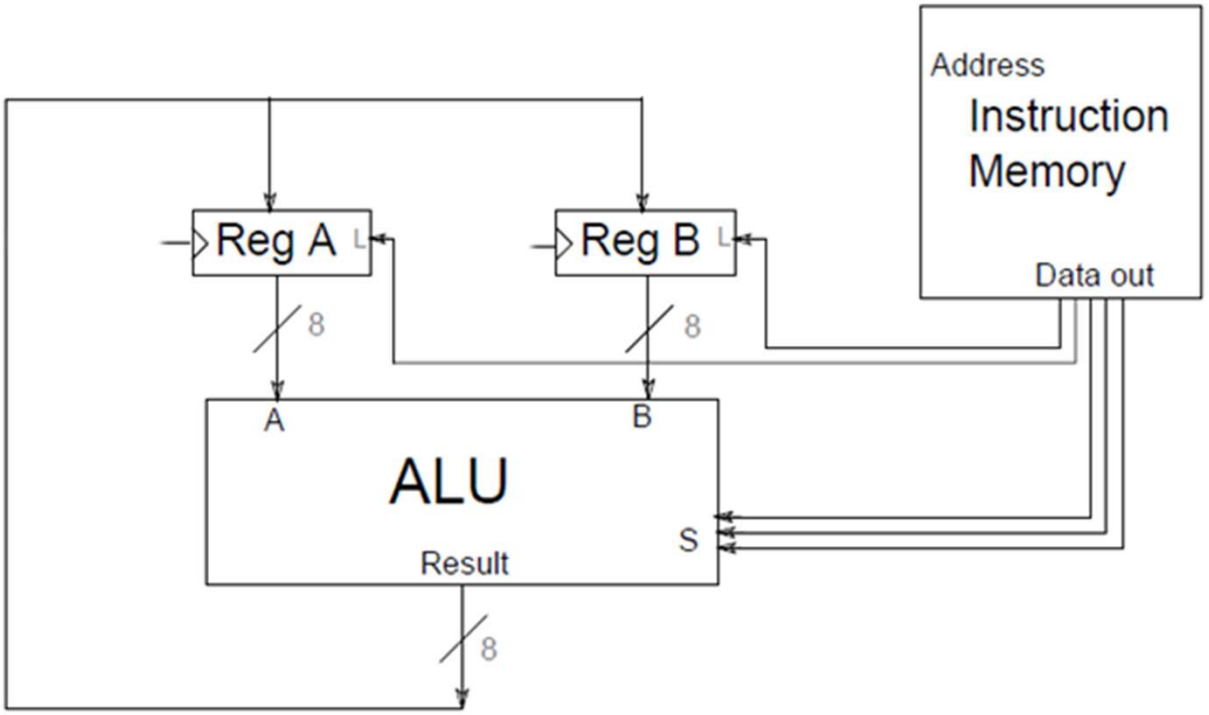
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	**0**	1
1	0	0	0	0	A=A+B	**1**	1
0	1	0	0	0	B=A+B	1	**2**
1	0	0	0	0	A=A+B	**3**	2
0	1	0	0	0	B=A+B	3	**5**
1	0	0	0	0	A=A+B	**8**	5
0	1	0	0	0	B=A+B	8	**13**

Pero, ¿de dónde viene el programa?

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 **…**

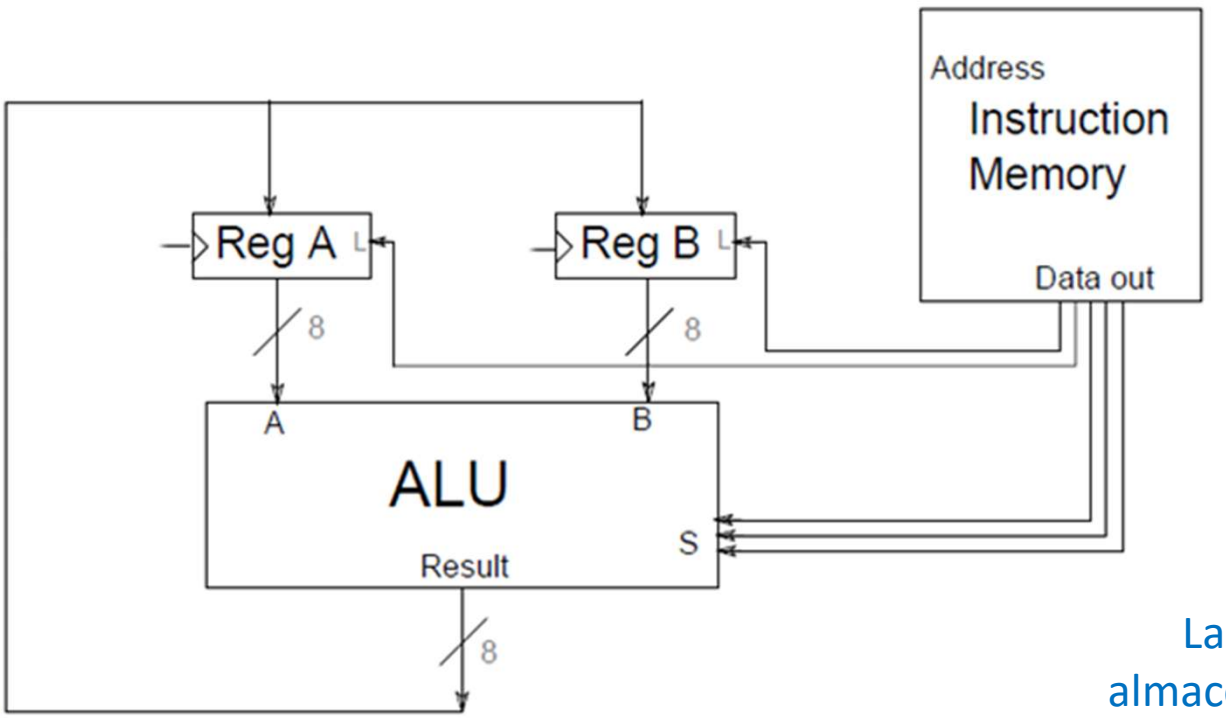


… en una **memoria** —que llamamos **Instruction Memory**: tiene un input, *Address* (que ya vamos a ver), y un output, *Data out*, que es por donde "sale" la instruccón (la combinación de señales de control) que hay que ejecutar a continuación



Internamente, la memoria está organizada como un (gran) arreglo de registros; cada registro —o *palabra de memoria*— almacena una instrucción y se identifica por la posición que ocupa en el arreglo

Estas posiciones correlativas, similares a los índices de un arreglo, se llaman **direcciones** (de memoria) y se especifican en binario (también, en hexadecimal)



direcciones	instrucciones
0000	1 0 0 0 0
0001	0 1 0 0 0
0010	1 0 0 0 0
0011	0 1 0 0 0
0100	1 0 0 0 0
0101	0 1 0 0 0

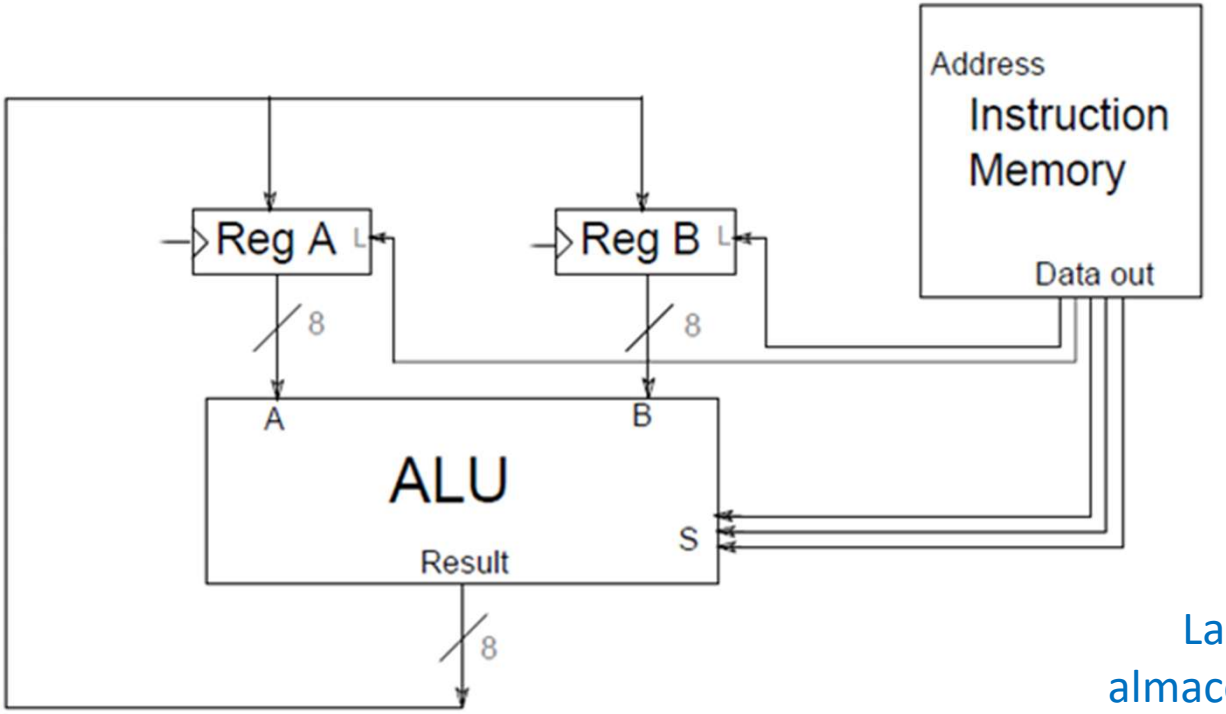
Contenido de la memoria

Las direcciones **no** están almacenadas en la memoria

Necesitamos que la ejecución de las instrucciones sea **secuencial**

… es decir, que por *Data out* las instrucciones vayan apareciendo una por una en el orden en que tienen que ser ejecutadas

→ hay que poder controlar cuál es la próxima instrucción que debe salir por *Data out*



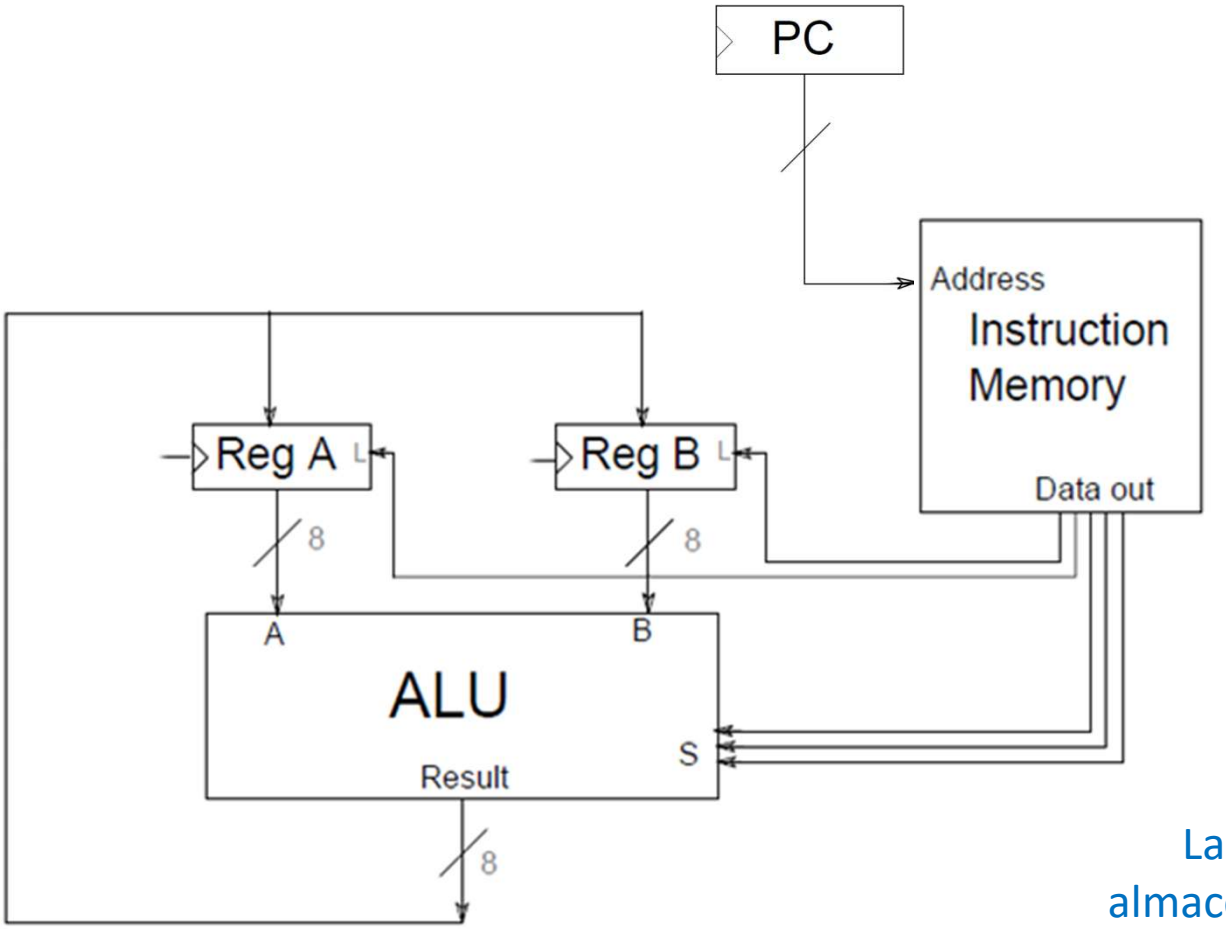
direcciones	instrucciones
0000	1 0 0 0 0
0001	0 1 0 0 0
0010	1 0 0 0 0
0011	0 1 0 0 0
0100	1 0 0 0 0
0101	0 1 0 0 0

Contenido de la memoria

Las direcciones **no** están almacenadas en la memoria

El registro especializado ***PC*** (*program counter* o *instruction pointer*) almacena una dirección de memoria (la dirección de una instrucción)

… 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*



direcciones	instrucciones
0000	1 0 0 0 0
0001	0 1 0 0 0
0010	1 0 0 0 0
0011	0 1 0 0 0
0100	1 0 0 0 0
0101	0 1 0 0 0

Contenido de la memoria

Las direcciones **no** están almacenadas en la memoria

La entrada *Address* debe tener tantas líneas como sea necesario para poder especificar cada una de las direcciones posibles de la memoria:

- en el ej., las direcciones son de 4 bits → la memoria puede almacenar 16 (= 2^4) registros, o *palabras de memoria*, en este caso de 5 bits c/u → cada palabra de memoria está formada por 5 flip-flops D

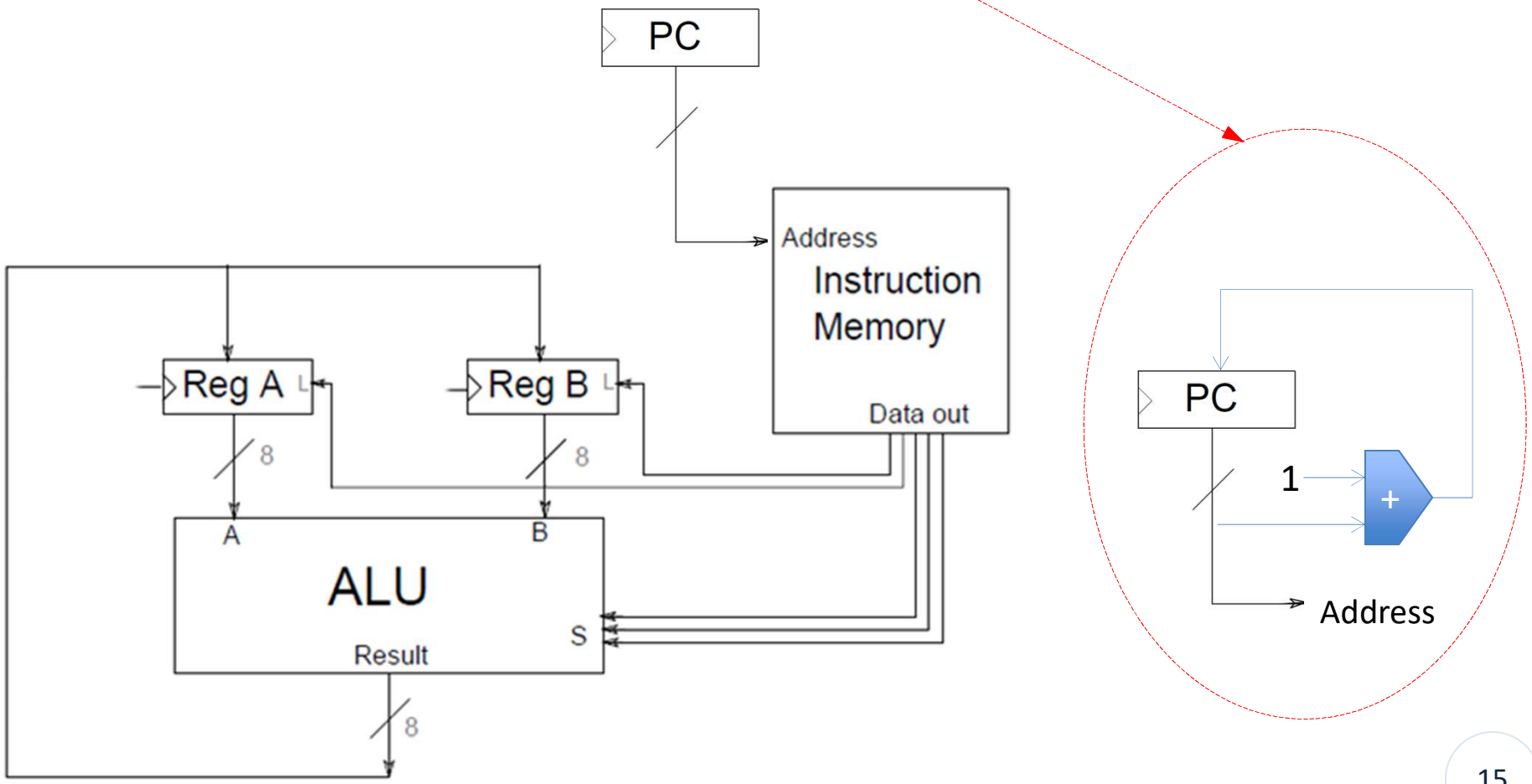
Las 4 líneas de *Address* entran a un **circuito decodificador** —como el de la diap. #47 de "logicaDigital", pero con 4 líneas de input en vez de 2 (N_i) y 16 líneas de output en vez de 4 (D_j)

C/u de las 16 líneas de output D_j del decodificador controla la *lectura* de una de las 16 palabras de memoria: D_j es uno de los dos inputs de 5 compuertas AND; el otro input de cada AND es el output *Q* de uno de los 5 flip-flops que forman la palabra de memoria

Finalmente, los outputs de las 16 ANDs que están en una misma posición (columna) en las 16 palabras de memoria, son inputs de una misma compuerta OR cuyo output va a *Data out*

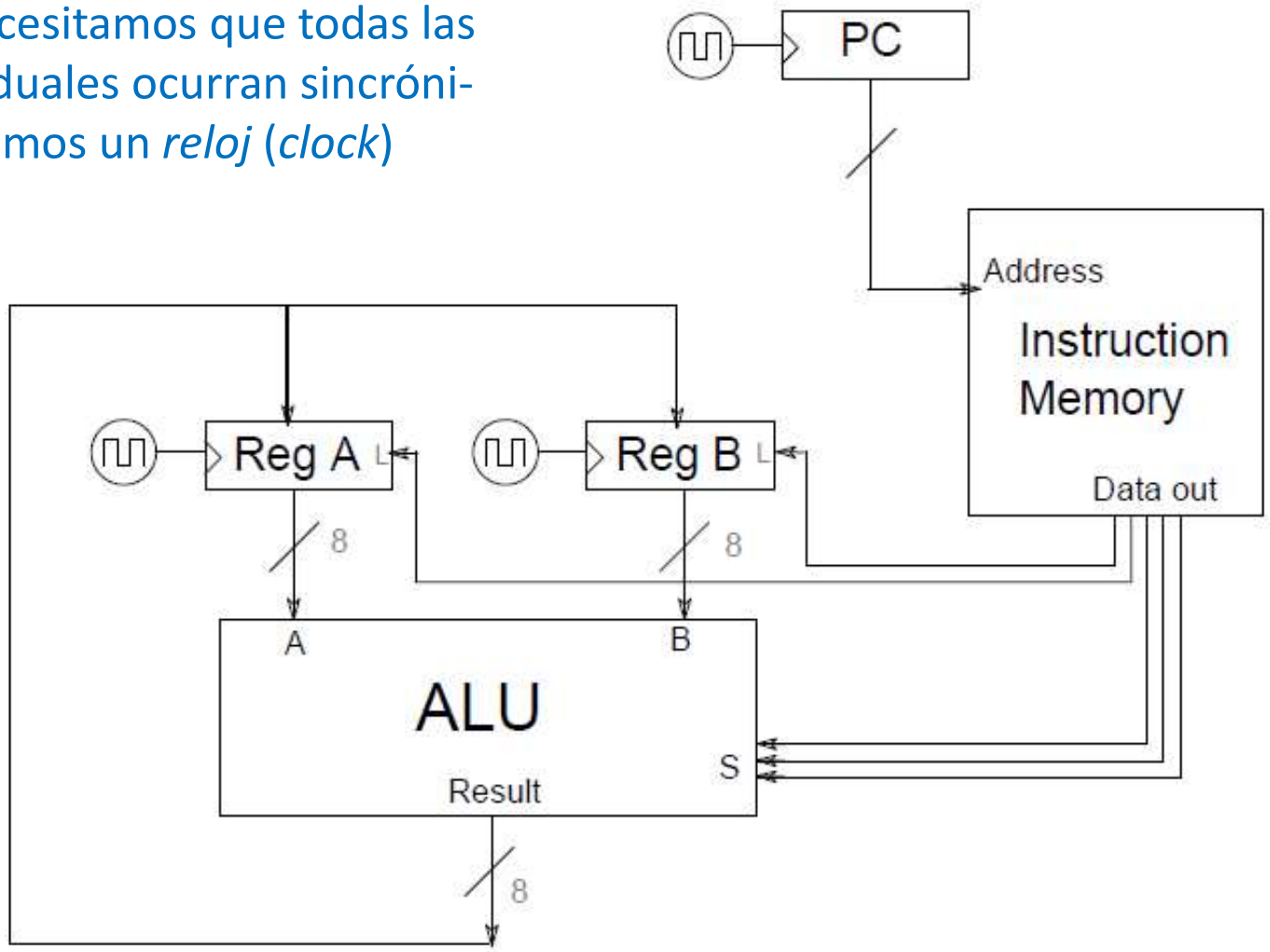
El registro *PC* tiene que ir incrementando automáticamente su contenido para que el programa se ejecute por completo sin más intervención nuestra que a la partida:

- requiere un circuito sumador adicional que no se muestra en las demás figuras

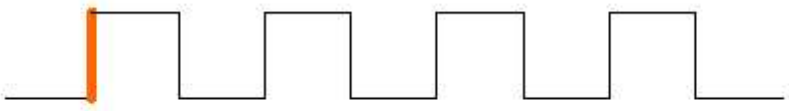


En lo que sigue, vamos a suponer una *Instruction Memory* con 2^8 palabras (o registros) → direcciones de 8 bits

Finalmente, necesitamos que todas las acciones individuales ocurran sincróni-camente: incluimos un *reloj* (*clock*)

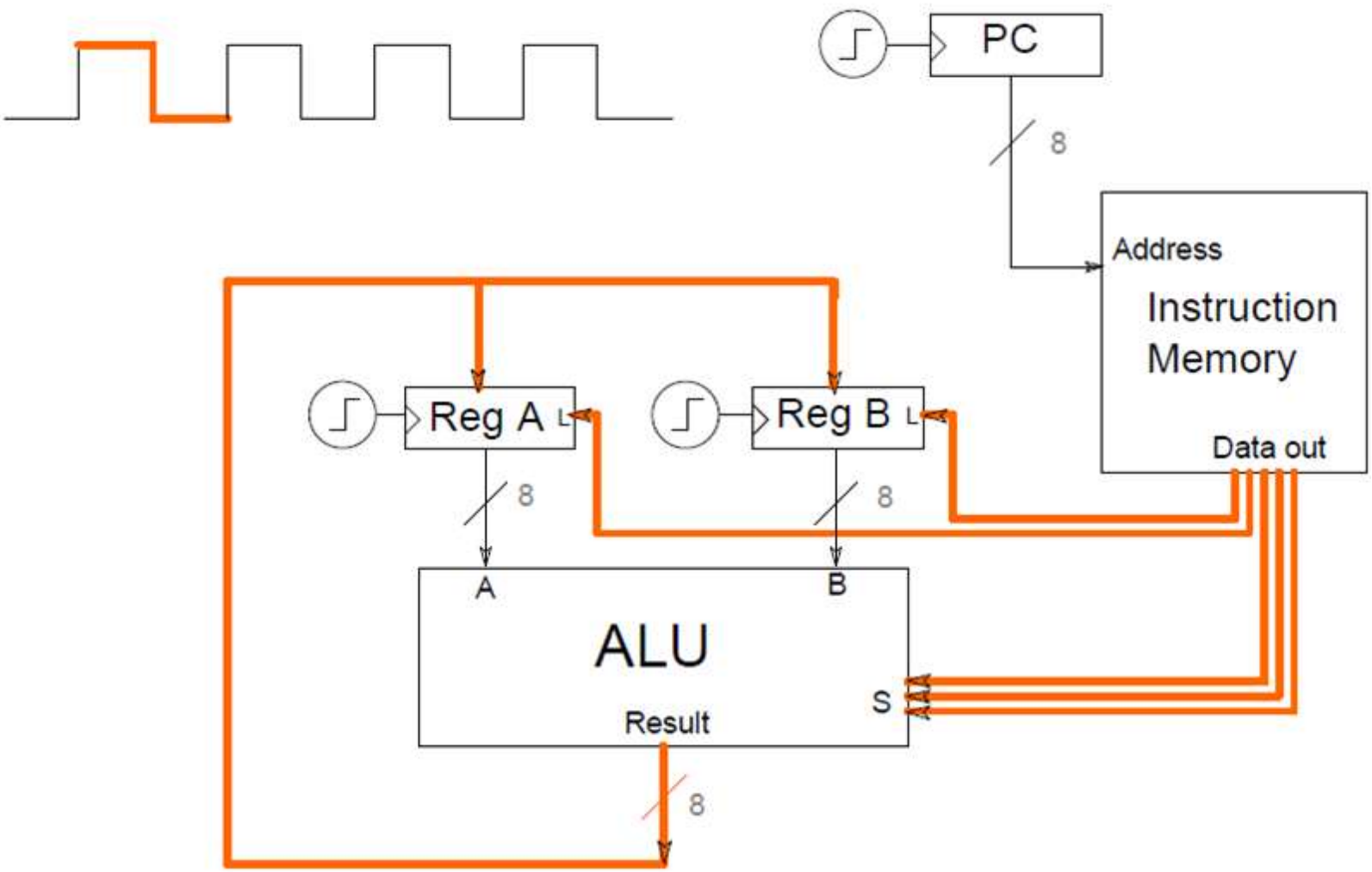


Reloj: Circuito que emite una serie de pulsos con un ancho preciso y un intervalo preciso entre pulsos consecutivos, y cuya frecuencia es controlada por un oscilador de cristal

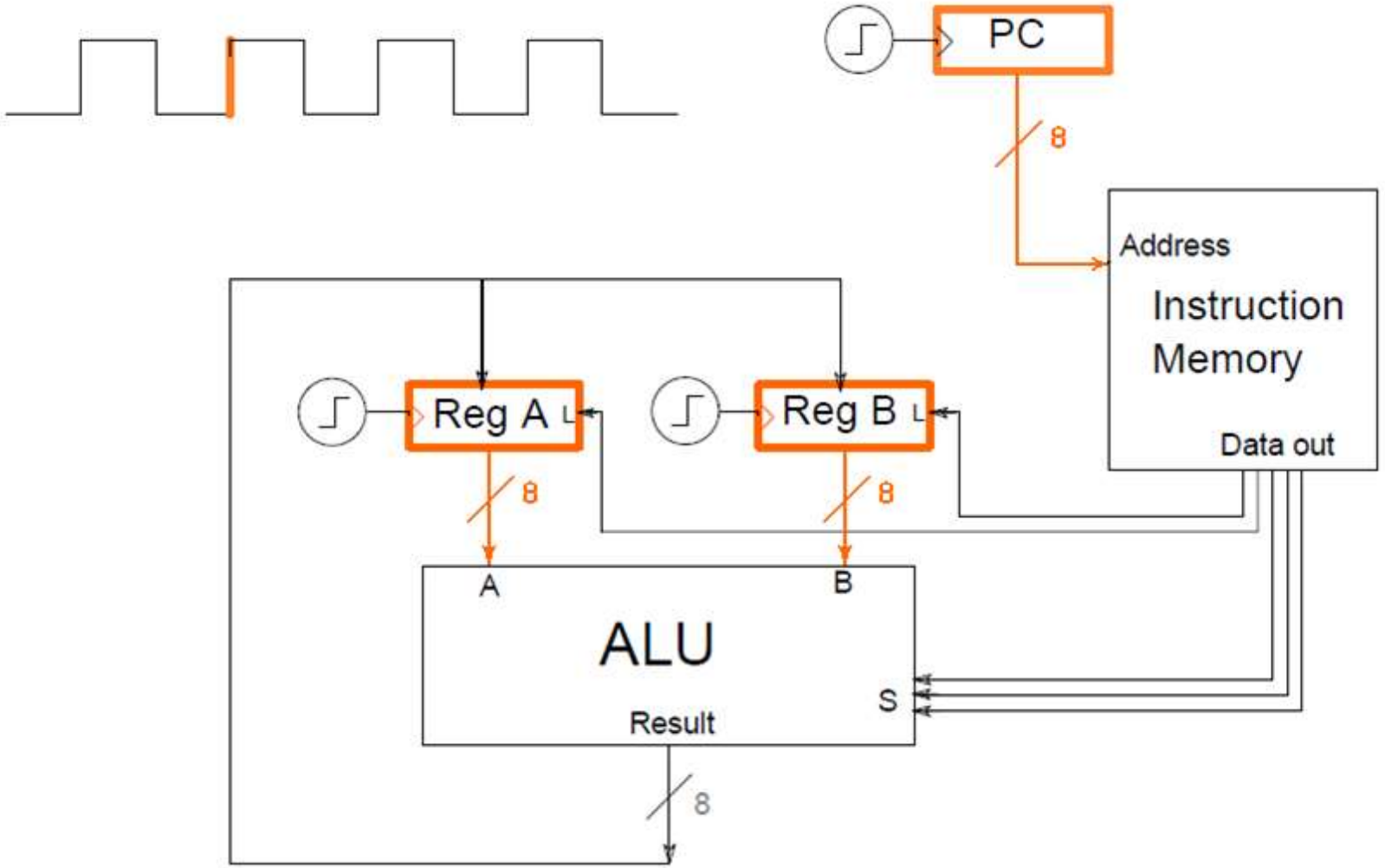


Cuando el pulso *sube* (*flanco de subida*), el *PC* actualiza su valor que va a la entrada *Address* de la memoria, y los registros *A* y *B* actualizan sus valores, que van a las entradas de la *ALU*

Durante el tiempo de ciclo del reloj, se ejecuta la instrucción: la *ALU* realiza la operación especificada por sus tres señales de control, y el resultado se pone a la entrada de los registros *A* y *B*

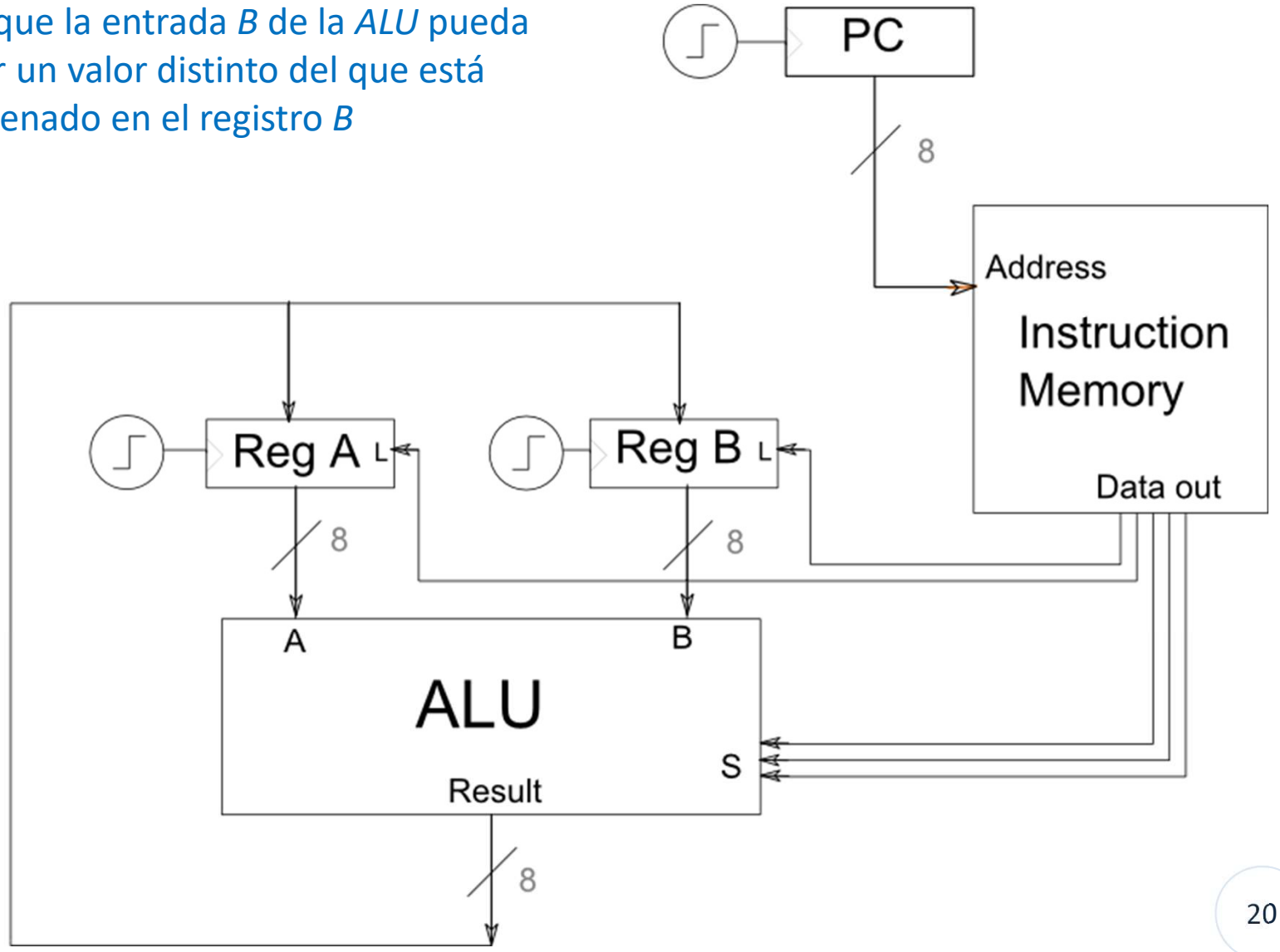


En el siguiente flanco de subida, *PC* pone una nueva dirección (la anterior más 1) a la entrada de la memoria, y los registros *A* y *B* ponen (posiblemente) nuevos valores en las entradas de la *ALU*



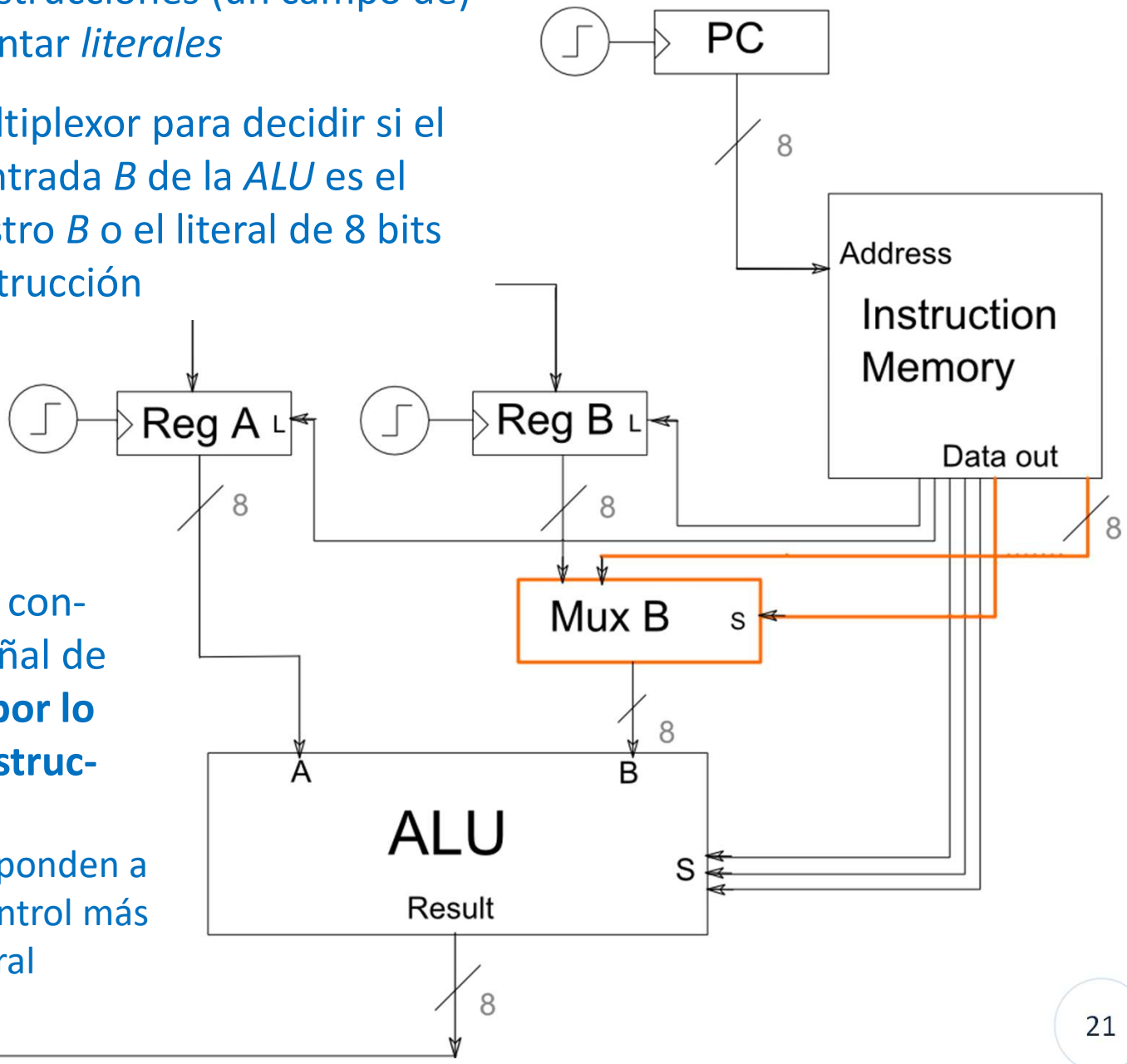
¿Cómo podemos independizarnos de los valores iniciales en los registros?

- 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 de 8 bits que viene en la instrucción

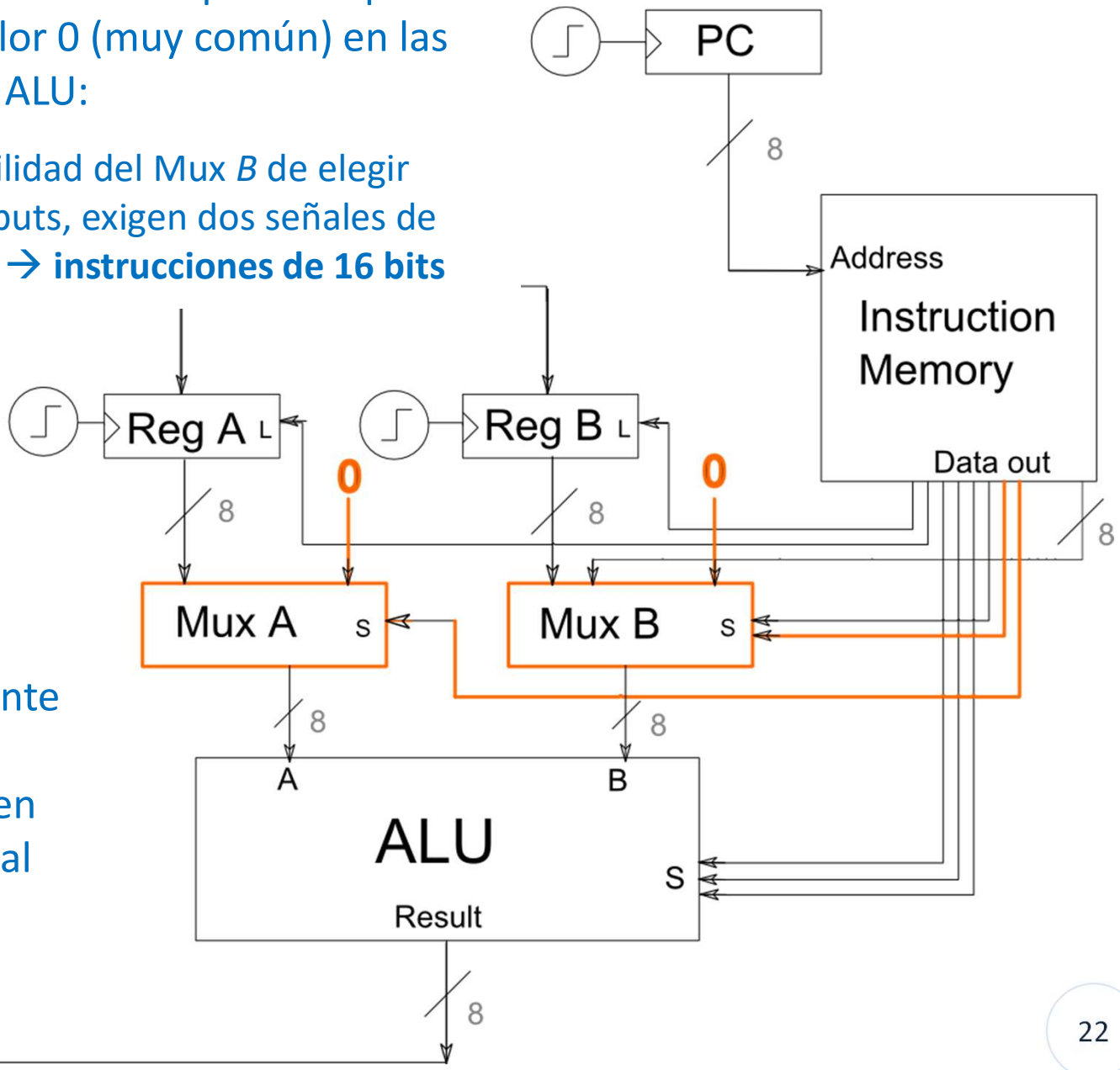


El multiplexor *B* es controlado por una señal de control adicional, **por lo que ahora cada instrucción tiene 14 bits**:

- 6 bits que corresponden a las 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* y *B* de la ALU:

- el Mux *A* y la posibilidad del Mux *B* de elegir ahora entre tres inputs, exigen dos señales de control adicionales → **instrucciones de 16 bits**



El valor 0 normalmente está almacenado de forma permanente en un registro especial al qu eno se le puede cambiar el valor

Instrucciones de 16 bits de largo implica que la *Instruction Memory* debe estar formada por registros, o palabras, de 16 bits de largo → cada palabra de memoria se compone de 16 flip-flops

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

… pero solo tenemos 28 instrucciones distintas

¿Cómo podemos ahorrar flip-flops 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:

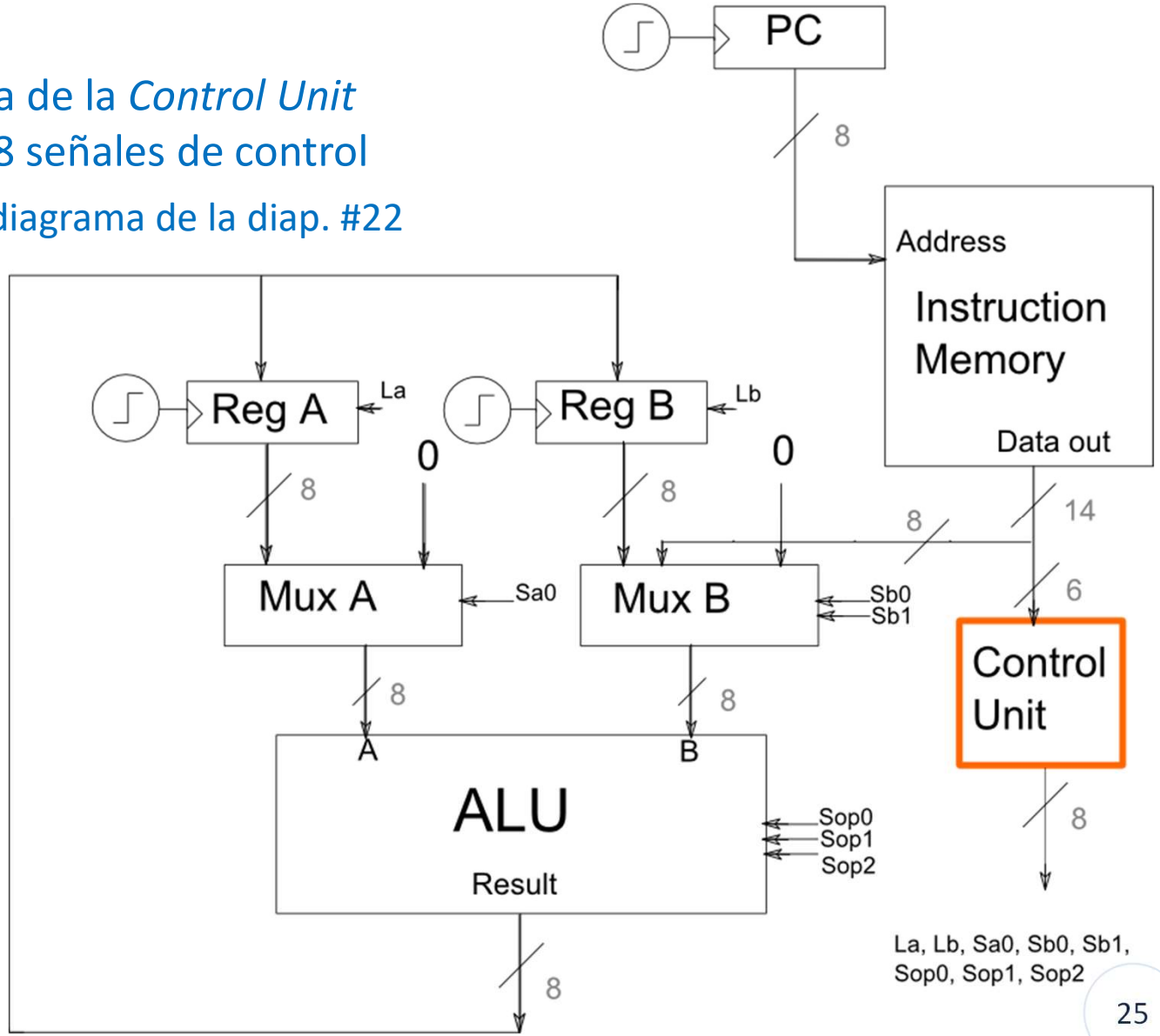
- numeramos (en binario) las instrucciones correlativamente y usamos estos números como identificadores de las instrucciones
- por ahora, vamos a usar opcodes de 6 bits, desde 000000 hasta 011011
- … de modo que las instrucciones van a tener 14 bits: opcode + literal

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 **Control Unit** —un circuito digital—
traduce los *opcodes* a las señales de control

Las 8 líneas de salida de la *Control Unit*
corresponden a las 8 señales de control

… similarmente al diagrama de la diap. #22



Resumamos

Nuestro computador básico tiene instrucciones de 14 bits de largo:

- 6 bits para el *opcode*, o identificador, de la instrucción

  … hay 28 opcodes diferentes, desde 000000 hasta 011011 → 28 instrucciones de máquina diferentes

- 8 bits para un literal

  … hay 10 instrucciones que usan el valor de este literal, en vez de usar el valor del registro *B*

  … p.ej., el opcode 001001 corresponde a la instrucción *A* = *A* – *literal*

  … en el caso de las instrucciones que no usan el literal (p.ej., el opcode 000111), estos 8 bits simplemente no se toman en cuenta

Los 8 bits del literal tienen dos interpretaciones posibles:

- un número entero entre –128 y +127, en las instrucciones aritméticas
- simplemente un patrón de 8 bits, en las instrucciones lógicas y *shifts*

…

Las instrucciones controlan el funcionamiento de los componentes del computador:

- ALU: qué operación aritmética/lógica/*shift* tiene que ejecutar
- registros *A* y *B*: actualizan o no su valor con el valor del resultado de la última operación ejecutada por la ALU (disponible en la salida *Result*)
- multiplexores *A* y *B*: qué valores van a parar a las entradas *A* y *B* de la ALU

…

El mecanismo a través del cual las instrucciones controlan el funcionamiento de los componentes del computador son las señales de control

- → el propósito de las instrucciones (identificadas por sus respectivos opcodes) es poner en 1 o en 0 las señales de control

Nuestro computador básico tiene (por ahora) 8 señales de control:

- *La* (controla el registro *A*), *Lb* (registro *B*), *Sa*0 (Mux *A*), *Sb*0 y *Sb*1 (Mux *B*), *Sop*2, *Sop*1 y *Sop*0 (ALU)

…

La *Control Unit* es un circuito digital que implementa la relación entre opcodes y señales de control —traduce opcodes a señales de control:

- por ahora, es un circuito que tiene 6 inputs (los bits del opcode) y 8 outputs (las señales de control)
- la relación entre un opcode y la combinación correspondiente de 0s y 1's de las señales de control es la que se muestra en la tabla de la próxima diap.

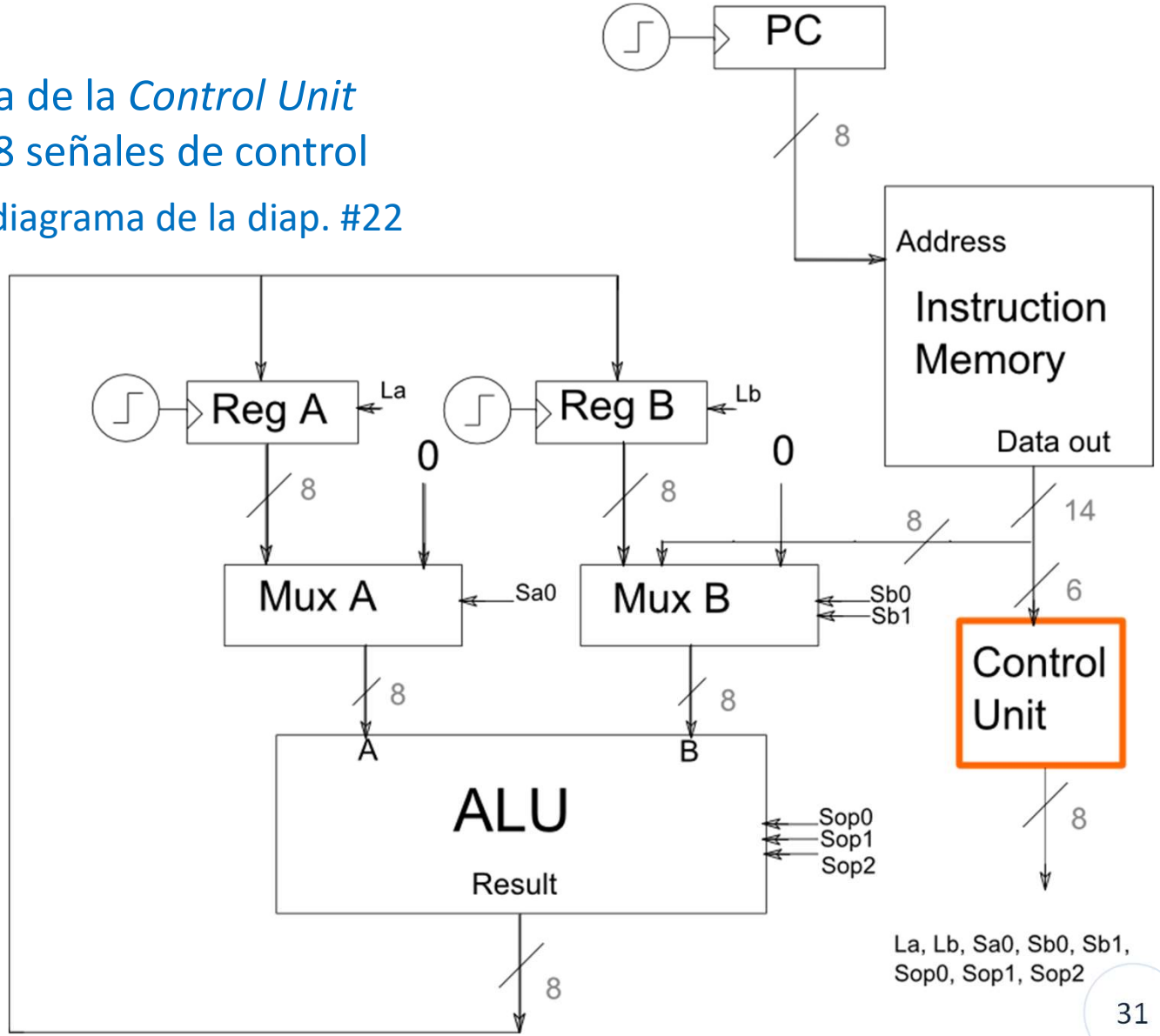
Así va el lenguaje de máquina de nuestro computador básico: las primeras 28 instrucciones (no se muestran los 8 bits del literal)

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 **Control Unit** —un circuito digital—
traduce los *opcodes* a las señales de control

Las 8 líneas de salida de la *Control Unit*
corresponden a las 8 señales de control

… similarmente al diagrama de la diap. #22



(Independientemente del uso de *opcodes*) el lenguaje de máquina es difícil de usar para programar el computador, o de leer al hacer *debugging* de un programa:

- p.ej., sería más fácil poder escribir *A* = *A* – *literal* —similarmente a lo que haríamos en un lenguaje de programación moderno— que tener que recordar 001001

El **lenguaje *assembly*** nos ayuda:

- es una versión simbólica del lenguaje de máquina
- cada instrucción tiene un nombre y dos operandos
- p.ej., la instrucción anterior es una resta (**SUB**), cuyos operandos son el (contenido del) registro *A* (**A**) y un literal (**Lit**), y cuyo resultado se almacena en el mismo registro *A*

  … y por lo tanto podría escribirse simbólicamente como **SUB A,Lit**
- así, a cada instrucción del lenguaje de máquina corresponde una única instrucción en el lenguaje *assembly* —una única combinación de nombre de operación y especificación de dos operandos

El lenguaje *assembly* de nuestro computador básico se muestra en las dos primeras columnas de la próx. diap.; p.ej.:

- `MOV A,B`
- `ADD A,Lit`

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

Este lenguaje (de máquina o *assembly*) y el computador básico asociado sólo nos permiten escribir y ejecutar programas muy simples:

- p.ej., estrictamente secuenciales y sin variables

En las próximas diapositivas, vamos a agregar nuevas funcionalidades a nuestro computador básico:

→ vamos a agregar nuevas componentes y nuevas señales de control

→ el circuito de la *Control Unit* se va a volver un poco más complejo (aunque no lo vamos a describir aquí)

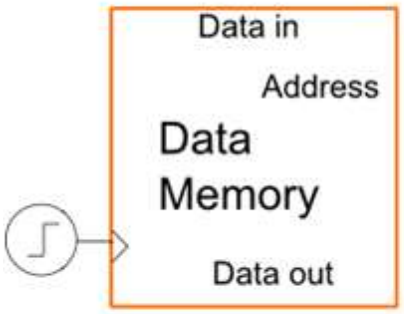
…

… pero esto nos va a permitir agregar instrucciones adicionales a nuestro lenguaje *assembly* (y lenguaje de máquina)

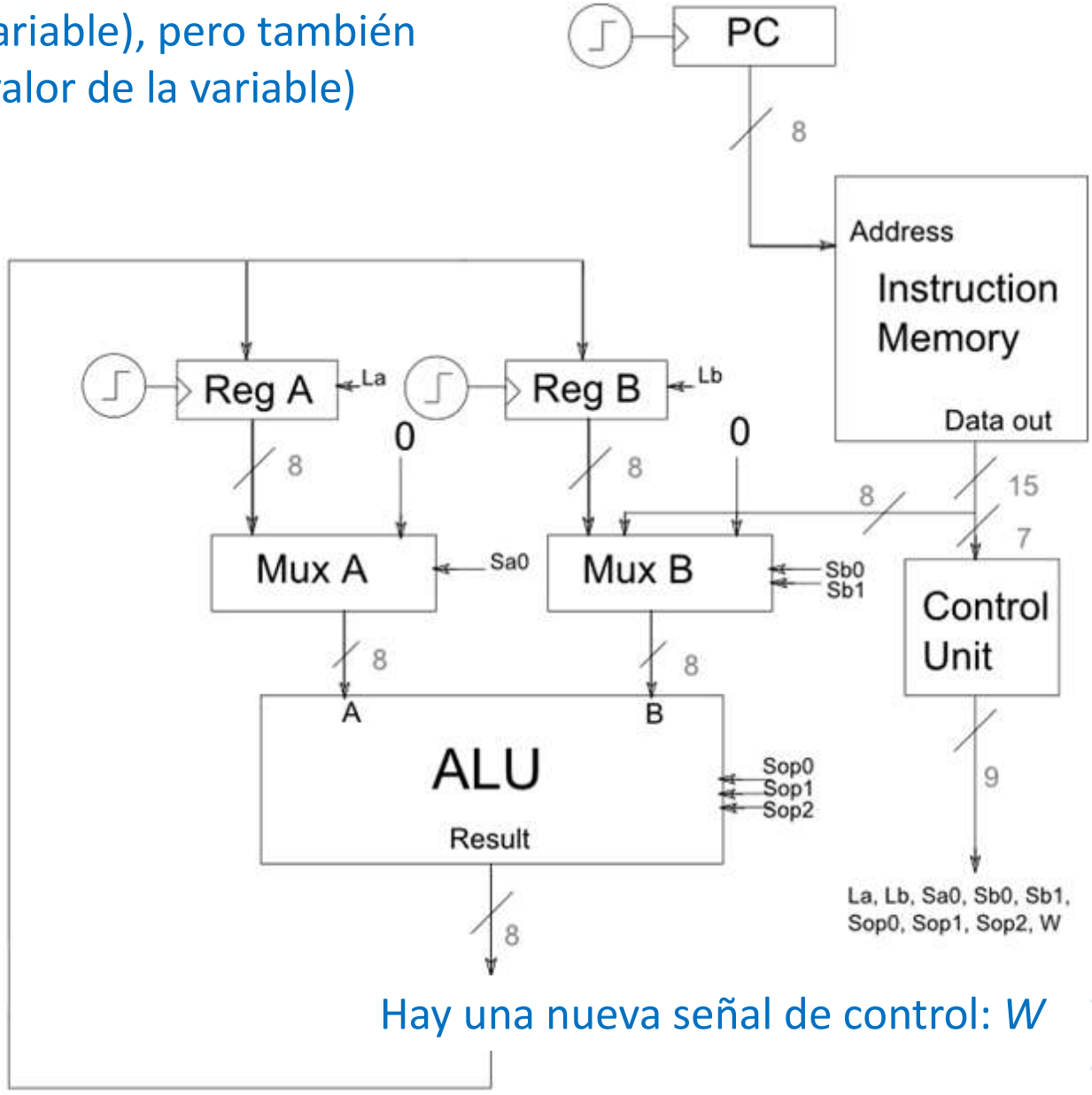
… fundamentales para que el computador pueda ejecutar programas más “reales”, como los que escribimos en lenguajes de programación modernos:

- lectura y escritura de datos —las *variables*— en una memoria de datos (*Data Memory*), adicional a la memoria de instrucciones
- ejecución de instrucciones en un orden distinto al estrictamente secuencial —sentencias de tipo `if` y `while`— mediante ”saltos” condicionales e incondicionales
- llamado a y retorno desde funciones (subrutinas, métodos) con parámetros

Para poder manejar variables en un programa, necesitamos una memoria que podamos leer (usar el valor de la variable), pero también *escribir* (cambiar el valor de la variable)



Esta ***Data Memory*** es similar a la *Instruction Memory*, pero además de la entrada *Address* tiene una entrada *Data in*



Hay una nueva señal de control: *W*

Las instrucciones del programa (el *CODE*) siguen estando en la *Instruction Memory*

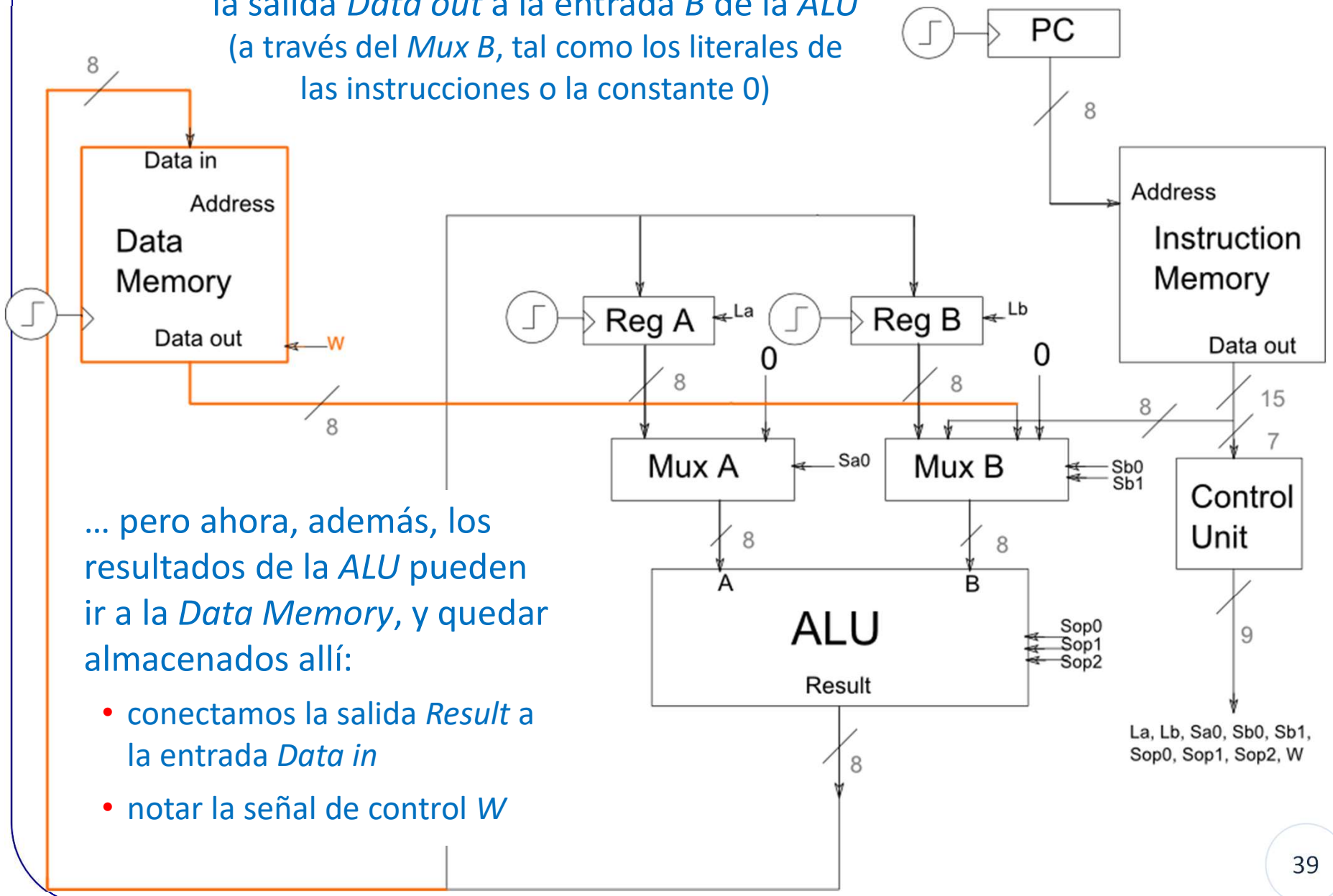
… pero ahora además tenemos las variables del programa (la *DATA*) en la *Data Memory*

Cada variable (su valor, p.ej., *Dato 1*) ocupa una palabra de memoria

… y la identificamos por un *label* (el "nombre" de la variable, p.ej., *var1*), en lugar de su dirección

Dirección	Label	Instrucción/Dato
	DATA:	
0x00	var0	Dato 0
0x01	var1	Dato 1
0x02	var2	Dato 2
0x03		Dato 3
0x04		Dato 4
	CODE:	
0x00		Instrucción 0
0x01		Instrucción 1
0x02		Instrucción 2
0x03		Instrucción 3
0x04		Instrucción 4

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 las instrucciones o la constante 0)



… 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*

Nuestro lenguaje *assembly* va a tener ahora instrucciones adicionales, similares a las que ya conocemos

… pero que operan sobre las variables almacenadas en la *Data Memory*

Son instrucciones adicionales

→ tienen opcodes distintos a los opcodes de las instrucciones que hemos visto hasta ahora (que sólo hacen referencia a los registros):

- para poder acomodar estas nuevas instrucciones, y otras que vamos a ver luego, los opcodes tienen ahora 7 bits (a partir de la diap. #37)

Algunas instrucciones hacen referencia explícitamente a la variable —*direccionamiento directo*:

- el operando se especifica como **`(`***label***`)`** —la instrucción de máquina usa los 8 bits del literal para *label**
- p.ej., **`MOV A,(var1)`** significa guardar en el registro *A* el contenido de la dirección de memoria (es decir, el valor) de la variable **`var1`**

*Así, a partir de ahora, el literal tiene tres interpretaciones posibles:

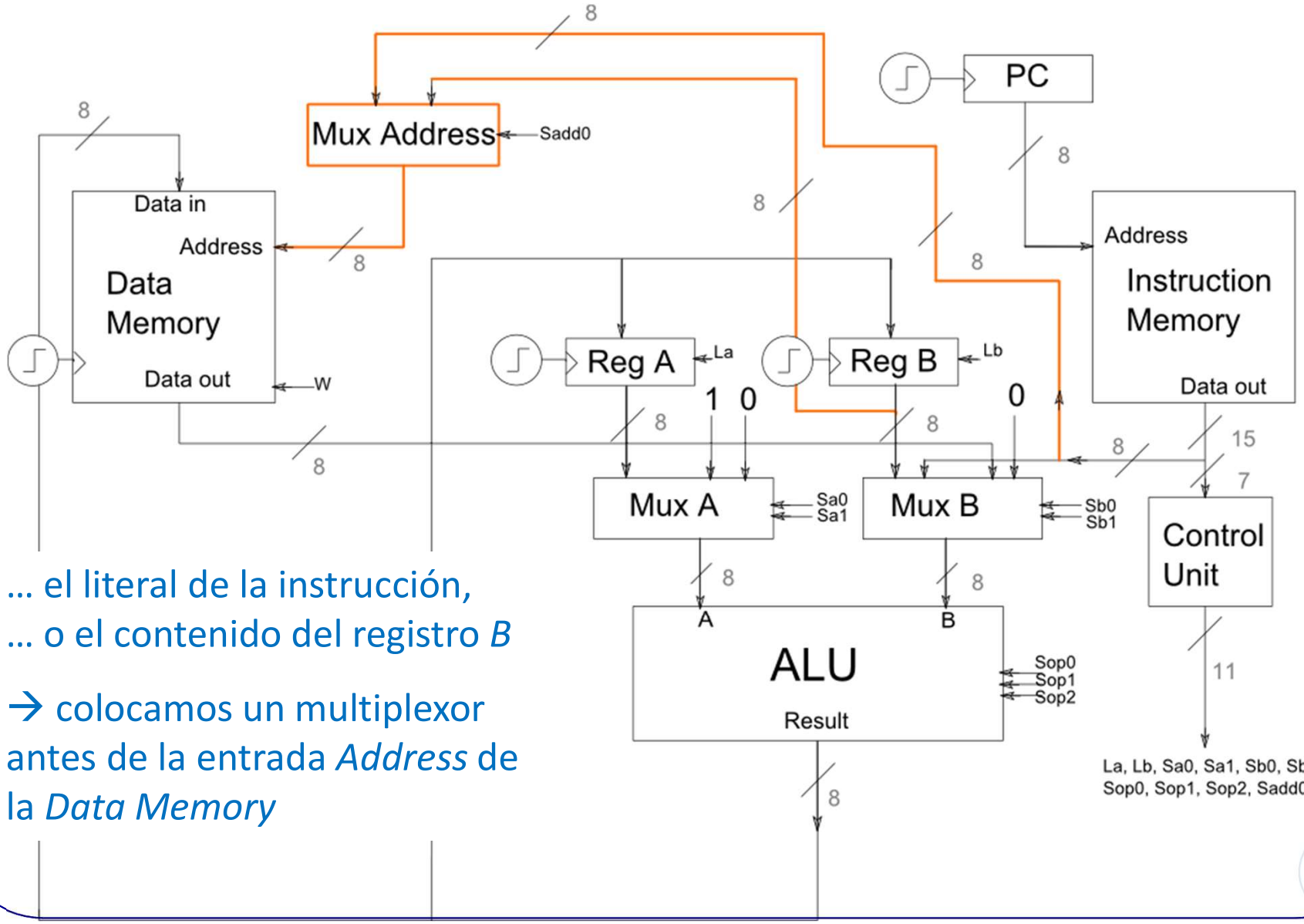
- un número entero entre –128 y +127, en las instrucciones aritméticas
- simplemente un patrón de 8 bits, en las instrucciones lógicas y *shifts*
- un número entero sin signo entre 0 y 255, correspondiente a una dirección de memoria (de la *Data Memory*), en las instrucciones que hacen referencia a una variable del programa

En cambio, en *direccionamiento indirecto*, la variable es especificada a través del registro *B*

… es decir, en el registro *B* está almacenada la dirección de memoria de la variable que nos interesa:

- el operando se especifica como **`(B)`**
- p.ej., **`MOV A, (B)`** significa guardar en el registro *A* el contenido de la celda de memoria (de la *Data Memory*) cuya dirección está almacenada en el registro *B*

La dirección de la palabra (la *variable*) de la *Data Memory* que queremos leer (*W* = 0) o escribir (*W* = 1) se puede especificar de dos maneras:



… el literal de la instrucción,
… o el contenido del registro *B*

→ colocamos un multiplexor antes de la entrada *Address* de la *Data Memory*

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,(Dir)	A=A-Mem[Dir]		SUB A,var1
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		-
OR	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,(Dir)	A=notMem[Dir]		NOT A,(var1)
	A,(B)	A=notMem[B]		-
	(Dir)	Mem[Dir]=not A		NOT (var1)
XOR	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,(Dir)	A=shift left Mem[Dir]		SHL A,(var1)
	A,(B)	A=shiflt left Mem[B]		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,(Dir)	A=shift right Mem[Dir]		SHR A,(var1)
	A,(B)	A=shiflt right Mem[B]		-
	(Dir)	Mem[Dir]=shift right A		SHR(var1)
INC	B	B=B+1		-

La tabla anterior muestra las nuevas instrucciones (*Instrucción* + *Operandos*, en la primera columna) de nuestro lenguaje *assembly*, que nos permiten manejar variables en los programas:

- son 32 instrucciones adicionales a las 28 que ya teníamos

Si bien en las instrucciones con direccionamiento directo uno de los operandos es el nombre de la variable entre **(** y **)** —ver columna *Ejemplo de uso*— la instrucción correspondiente en lenguaje de máquina usa la dirección de la variable en la *Data Memory*:

- el arreglo **`Mem`**, que aparece en la columna *Operación*, representa el arreglo de celdas de memoria disponibles en la *Data Memory*

P.ej., la próx. diap. muestra dos versiones de un programa que simplemente suma los valores de dos variables, **a** y **b**

En la versión 1:

- los sumandos **a** y **b** son *cargados** en los registros *A* y *B*, desde la *Data Memory*
- luego se ejecuta la instrucción **ADD A,B**, que suma ambos registros y deja el resultado en *A*
- finalmente, el contenido de *A* es *almacenado* en la variable **suma** en la *Data Memory*

En la versión 2:

- una vez que los sumandos están en los registros, se ejecuta la instrucción **ADD (suma)**, que suma ambos registros y almacena directamente el resultado en la variable **suma**

* Cuando un valor se copia de la memoria de datos a un registro, se llama **cargar** (*load*) el registro; y cuando un valor se copia de un registro a la memoria, se llama **almacenar** (*store*) en la variable (dirección de memoria)

```
DATA:
a       17
b       54
suma    —no inicializado

CODE-1: —versión 1
        MOV A,(a)
        MOV B,(b)
        ADD A,B
        MOV (suma),A

CODE-2: —versión 2
        MOV A,(a)
        MOV B,(b)
        ADD (suma)
```

La “magia” de convertir los nombres de las variables (o *labels*) a las direcciones físicas que esas variables ocupan en la *Data Memory* es responsabilidad de los programas llamados *assembler*, *linker* y *loader*:

- en la fig. de la diap. #38, las variables *var1*, *var2* y *var3* corresponden a las direeciones 0x00, 0x01 y 0x02, respectivamente

Aún no podemos escribir programas con instrucciones **`if`** o **`while`**:

- en ambos casos, el orden de ejecución de las instrucciones no sigue el orden en que están escritas

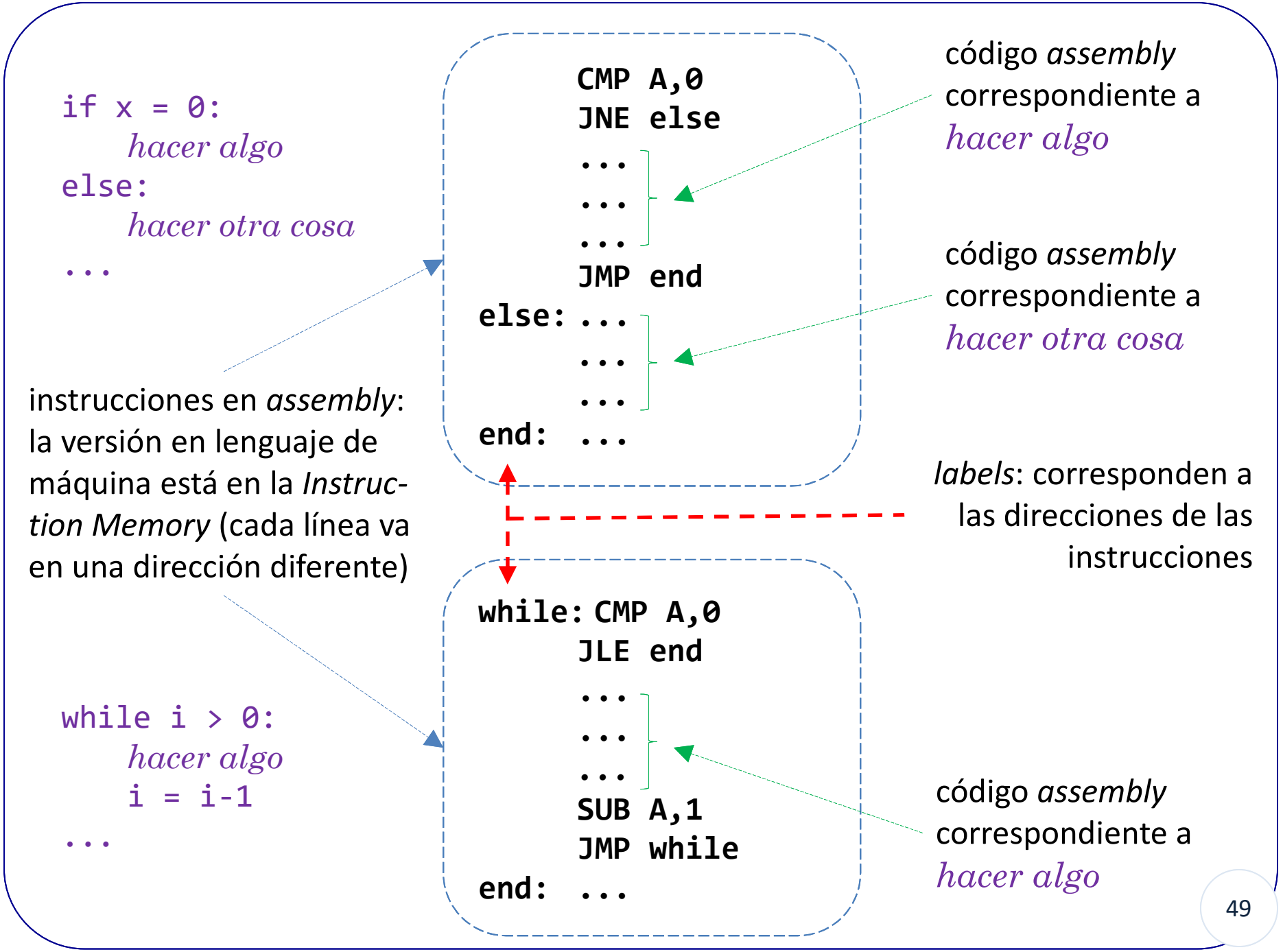
  … sino que depende del resultado de la evaluación de una condición

Para ello, necesitamos instrucciones que comparen dos valores entre sí, y de alguna manera codifiquen el resultado de la comparación:

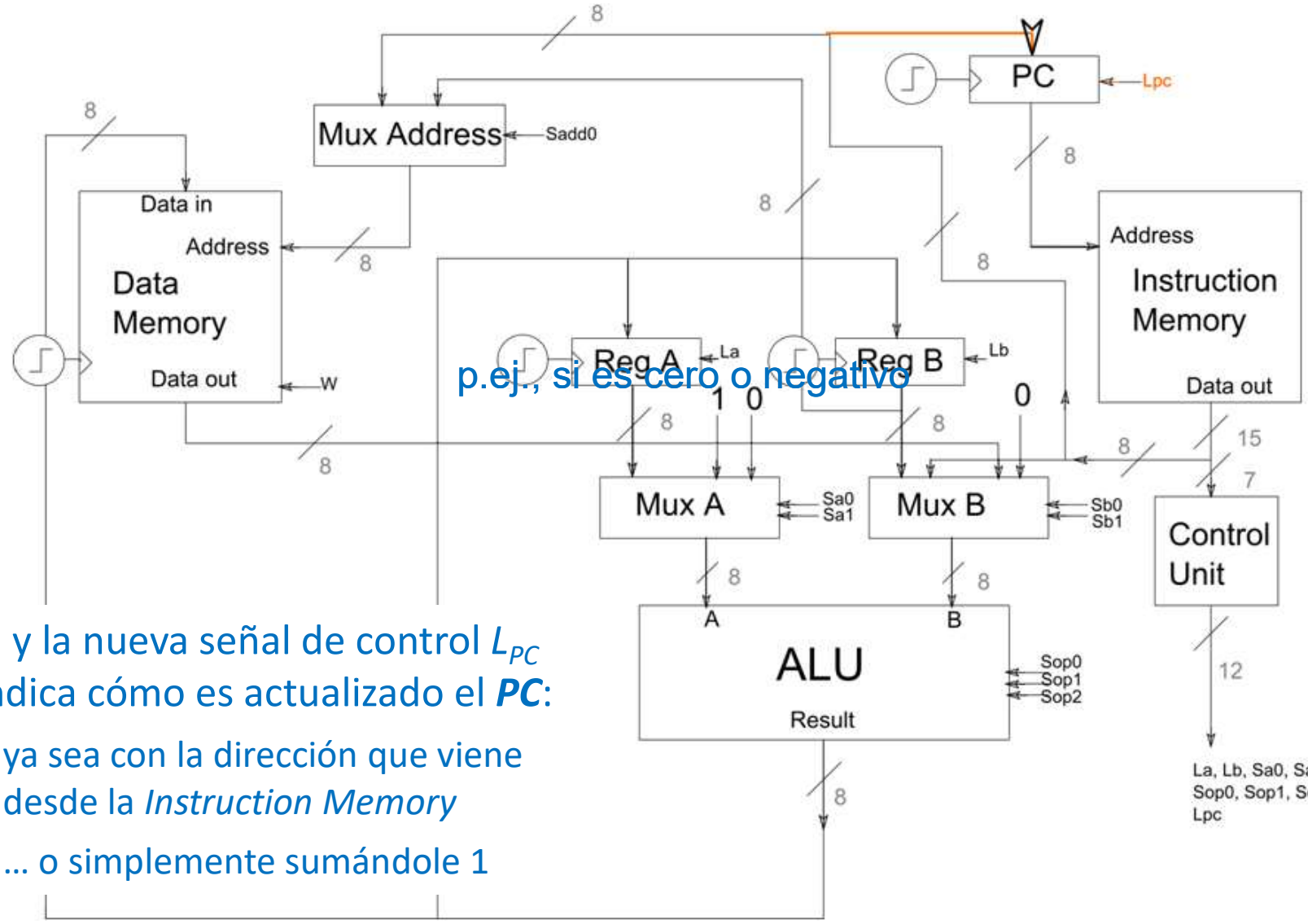
- p.ej., **`CMP A,B`** compara los contenidos de los registros *A* y *B*

… e instrucciones que especifiquen la dirección de memoria de la próxima instrucción que debe ser ejecutada (recordemos que las instrucciones están en la *Instruction Memory*):

- p.ej., **`JMP end`** "salta" directamente a ejecutar la instrucción identificada con el *label* (etiqueta) "end" —*saltos incondicionales*
- p.ej., **`JLE end`** "salta" a ejecutar la instrucción etiquetada "end" sólo si el resultado de la comparación ejecutada justo antes fue "menor o igual" (*less than or equal to*) —*saltos condicionales*


if x = 0:
hacer algo
else:
hacer otra cosa
...
CMP A,0
JNE else
...
...
...
JMP end
else: ...
...
...
end: ...
código assembly correspondiente a hacer algo
código assembly correspondiente a hacer otra cosa
instrucciones en assembly: la versión en lenguaje de máquina está en la Instruc-tion Memory (cada línea va en una dirección diferente)
labels: corresponden a las direcciones de las instrucciones
while i > 0:
hacer algo
i = i-1
...
while: CMP A,0
JLE end
...
...
...
SUB A,1
JMP while
end: ...
código assembly correspondiente a hacer algo

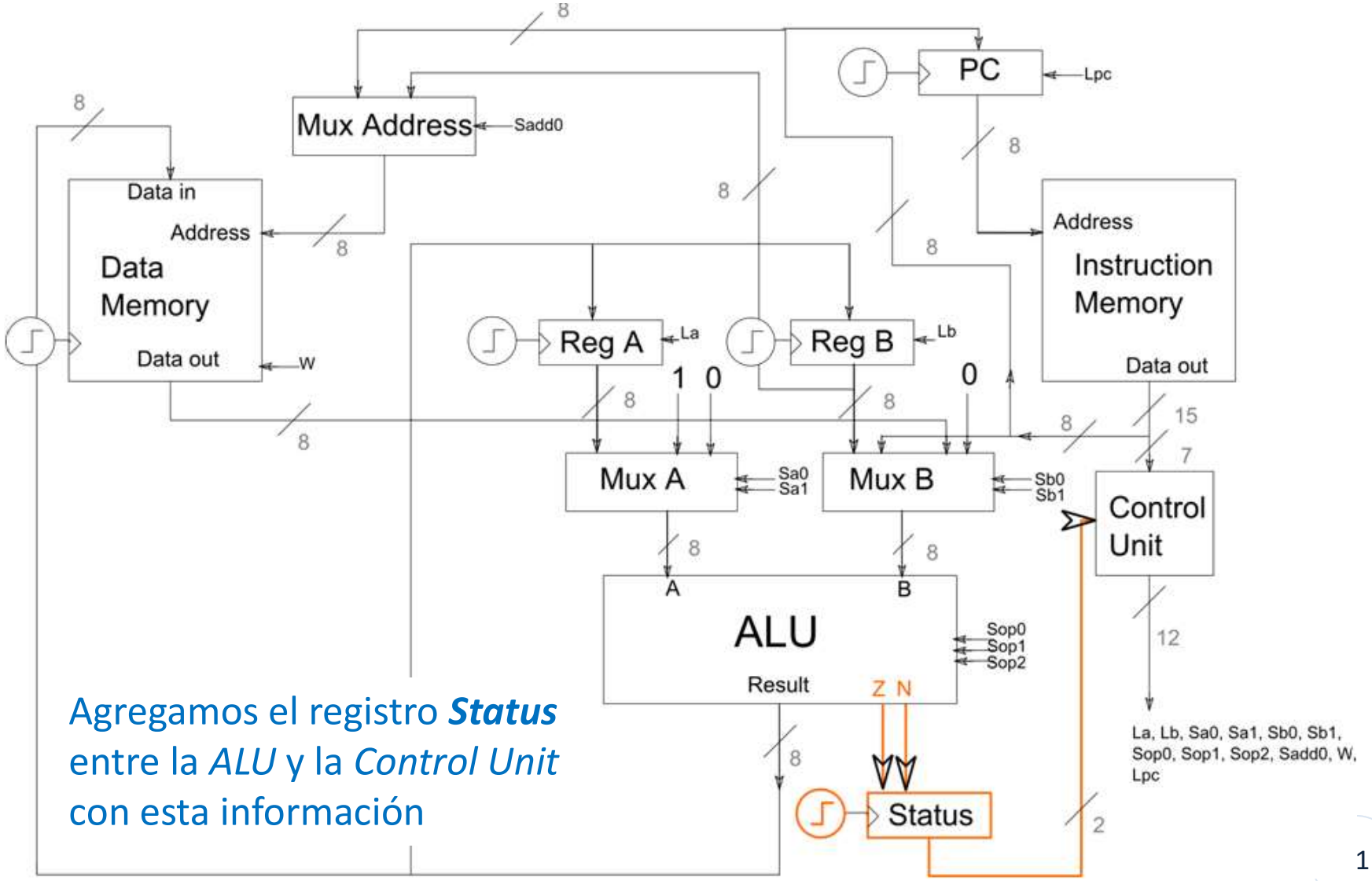
Para permitir **saltos incondicionales** (**JMP**) hay que poder poner en el ***PC*** la dirección de la instrucción que queremos ejecutar a continuación; conectamos la salida *Data out* de la *Instruction Memory* al ***PC*** …



… y la nueva señal de control L_{PC} indica cómo es actualizado el ***PC***:

- ya sea con la dirección que viene desde la *Instruction Memory*
- … o simplemente sumándole 1

Para **saltos condicionales** (**JNE**, **JLE**), la *Control Unit* tiene que examinar el resultado de la *ALU* justo después de hecha la comparación; p.ej., si es 0 o negativo



Agregamos el registro ***Status*** entre la *ALU* y la *Control Unit* con esta información

La tabla muestra 8 nuevas instrucciones adicionales de nuestro lenguaje *assembly* **(que ya totaliza 69 instrucciones ⇒ opcodes de 7 bits)**:

- **CMP** toma dos operandos y los compara por la vía de restarlos

  … el resultado queda codificado en los bits ***N*** (¿es negativo?) y ***Z*** (¿es cero?) del registro ***Status***

- las instrucciones **J*xx*** tienen un solo operando: la dirección de una instrucción en la *Instruction Memory* (especificada por una etiqueta, o *label*)

  … esa dirección (*Dir*) es escrita en el registro *PC* si y sólo si se cumple la condición codificada en términos de los bits *N* y/o *Z*

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
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

Con las 69 instrucciones vistas, nuestro assembly nos permite escribir programas como los siguientes:

… multiplicar y dividir números enteros (con y sin signo)

… procesar todos los elementos de un arreglo de números enteros:

- p.ej., aplicarle la misma operación a cada número

  … o sumar todos los números entre sí

Cada uno de estos programas se puede escribir de varias maneras:

- tal como ocurre al escribir programas en un lenguaje de programación de alto nivel

El próximo programa multiplica dos números no negativos por la vía de sumar repetidamente el multiplicando, tantas veces como lo indica el multiplicador

Primero, se inicializa en 0 la variable **`prod`**, que va a almacenar el resultado de la multiplicación:

- no hay una instrucción para asignar directamente el literal 0 a una variable
- → primero el registro *A* es cargado con 0 y luego el contenido de *A* es almacenado en **`prod`**

Luego, el registro *B* es cargado con 0 —representa, a lo largo de todo el programa, el número de repeticiones realizadas

… y el registro *A*, con el valor del multiplicador —el número de repeticiones que hay que realizar:

- el registro *A* toma diferentes roles a lo largo del programa, a diferencia del *B*

```
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
        JMP loop      —repetimos el ciclo
end:
```

…

Ahora comienza el ciclo que se repite (*label* **`loop`**)

Primero, se compara el número de repeticiones qua hay que realizar (registro *A*) con el número de repeticiones realizadas (registro *B*)

… si son iguales, entonces el programa termina (**`JEQ end`**)

… de lo contrario, se ejecuta una repetición —se suma una vez el multiplicando a la variable **`prod`**:

- se carga el multiplicando en el registro *A*, se le suma lo que ya está acumulado en la variable **`prod`**, y con el resultado se actualiza la variable **`prod`**
- se incrementa el contador de repeticiones —el registro *B*
- se carga el multiplicador en el registro *A*
- … y se inicia una nueva repetición

Subrutinas (funciones o métodos)

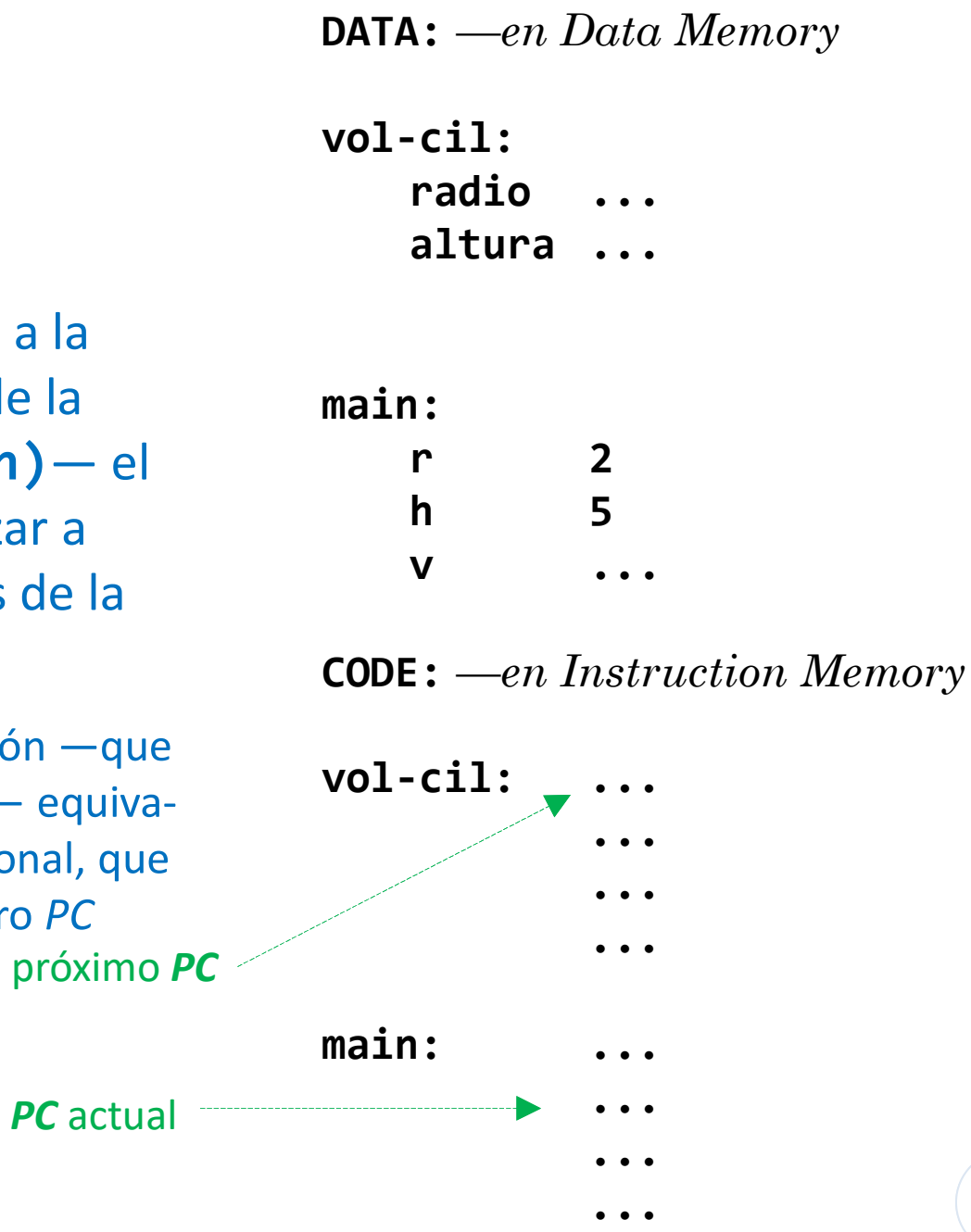
```
main: —programa principal
    r = … —asignar un valor a esta variable
    h = … —asignar un valor a esta variable
    v = vol_cil(r, h)
    print(“El volumen del cilindro es”, v, “cm3”)

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

¿Qué es qué?

- la expresión **`vol_cil(r, h)`** —a la derecha del signo “=“ en el **`main`**— 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 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*



2) Pero 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

main:
    r       2
    h       5
    v       ...
```

a través de los registros

`CODE:` —*en Instruction Memory*

```
vol-cil:    ...
            ...
            ...
            ...

main:       ...
            ...
            ...
            ...
```

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