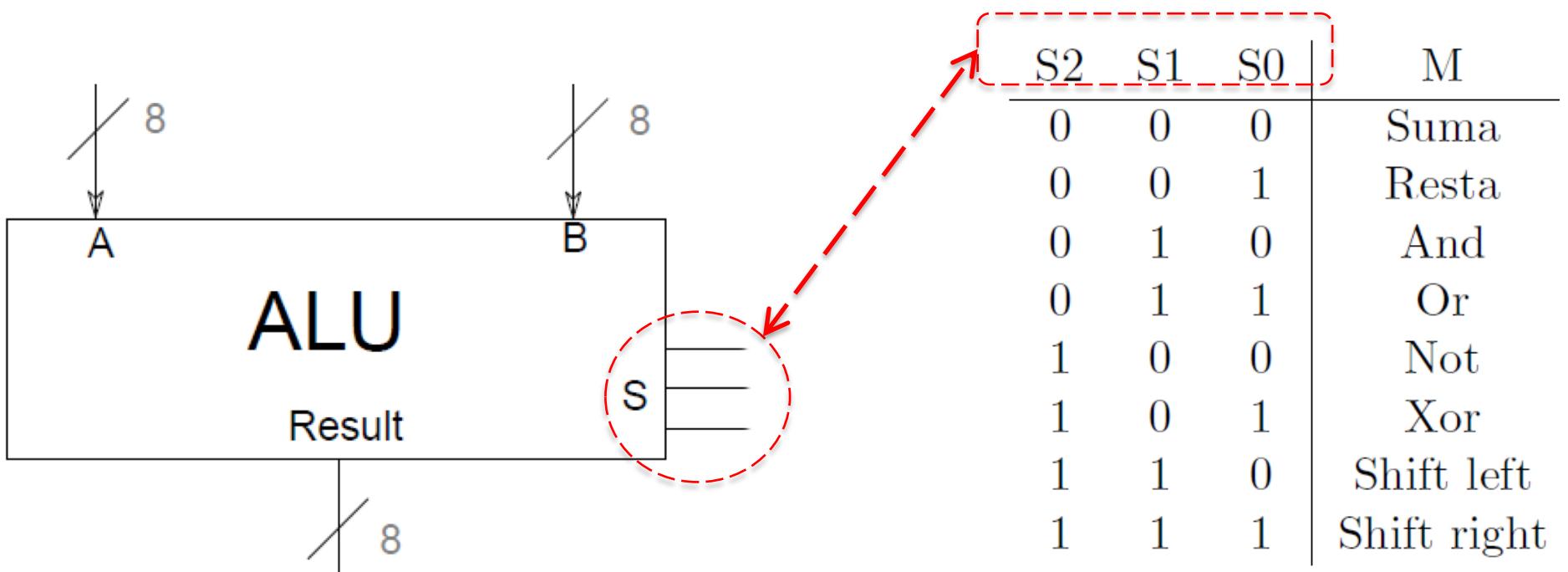


Procesador: componentes, instrucciones, *datapath*

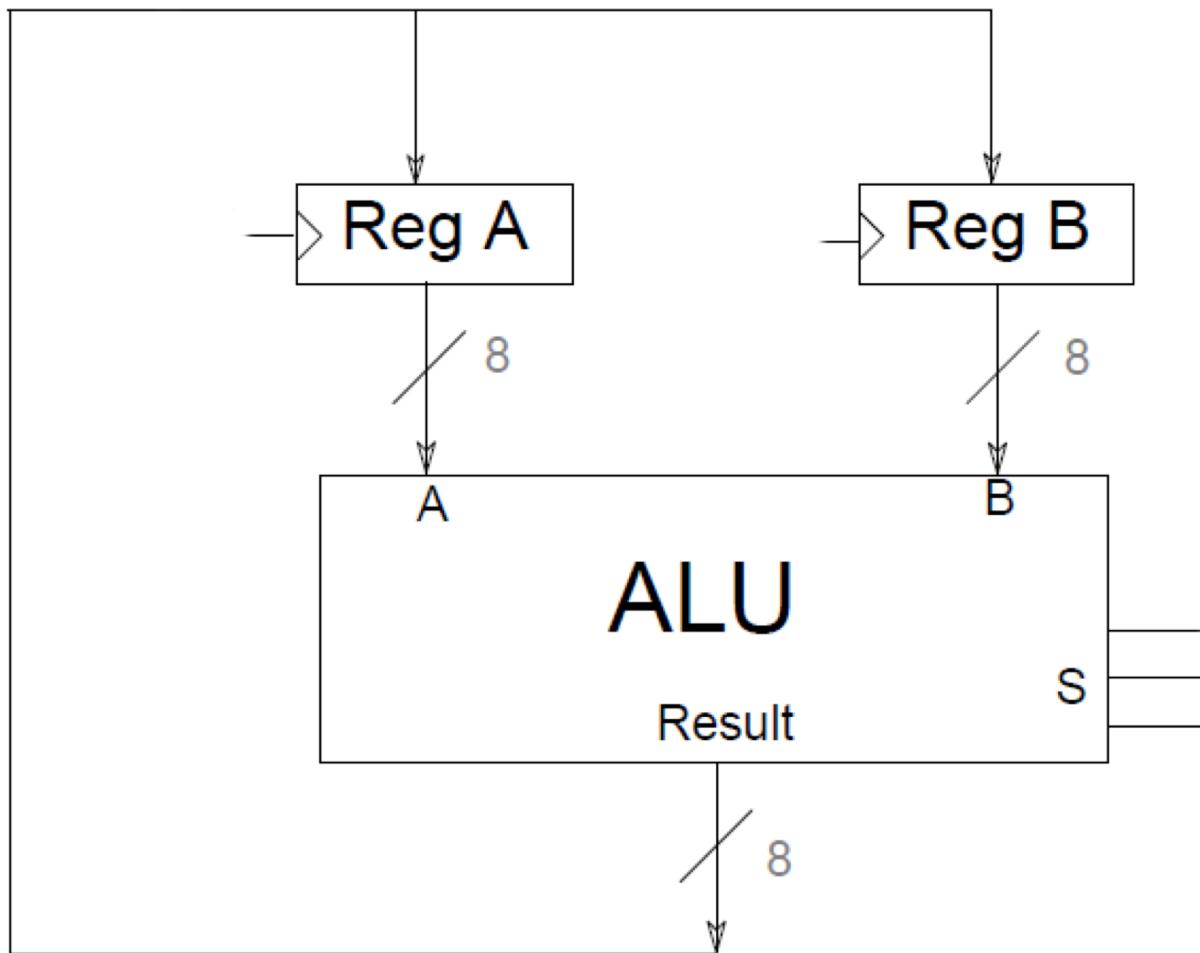
Arquitectura de Computadores – IIC2343

Dónde estamos



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

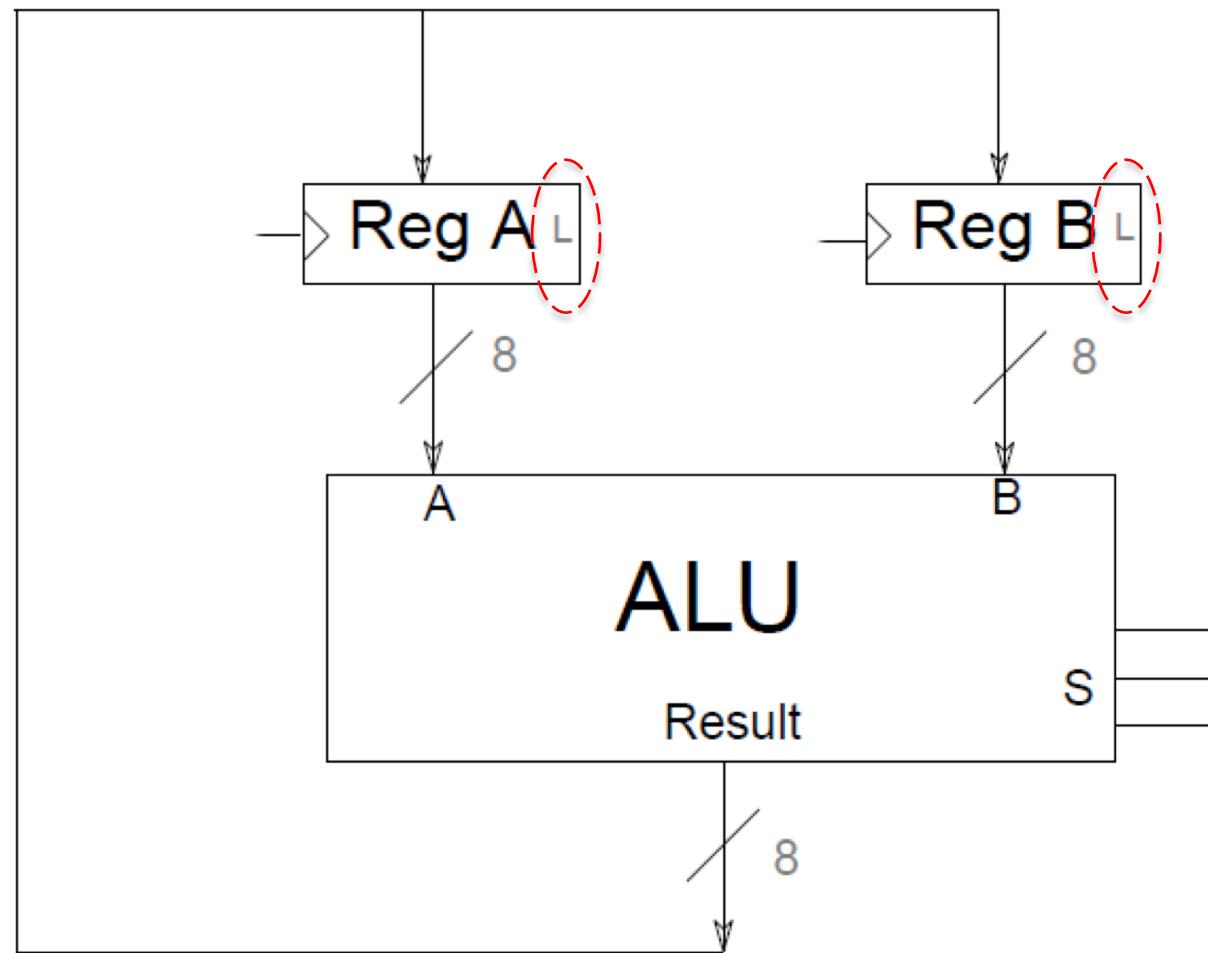


Vamos a suponer
registros de 8 bits, es
decir, de 8 flip-flops

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

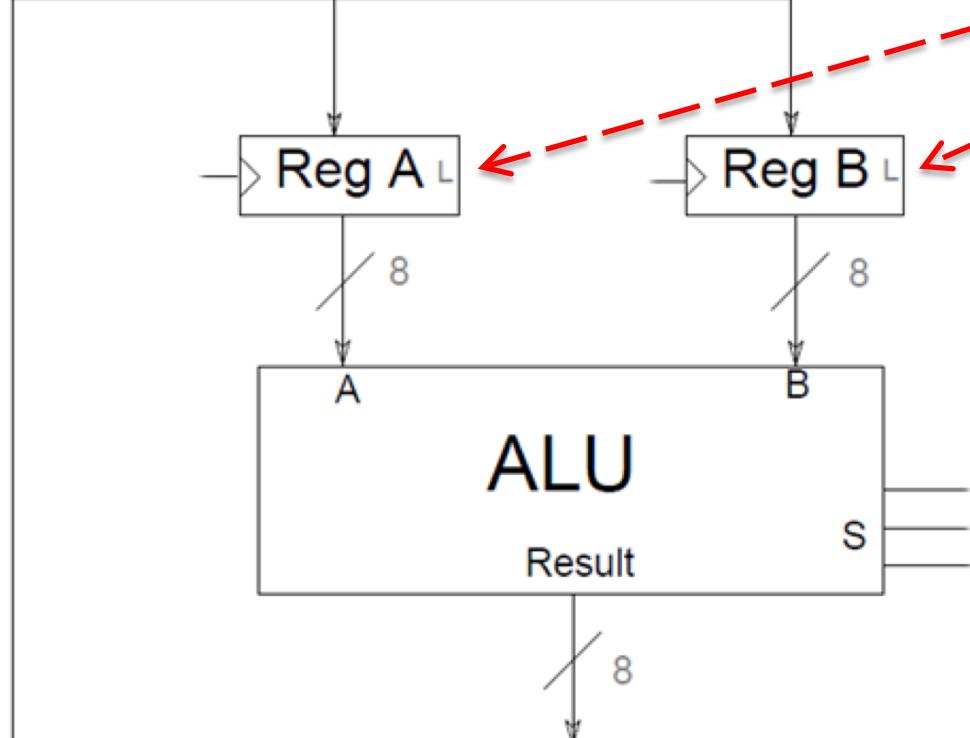
Agregamos las señales de control L_A y L_B para controlar la escritura —o actualización de los valores— de los registros:

- 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	$A=A \text{ and } B$
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 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** ...

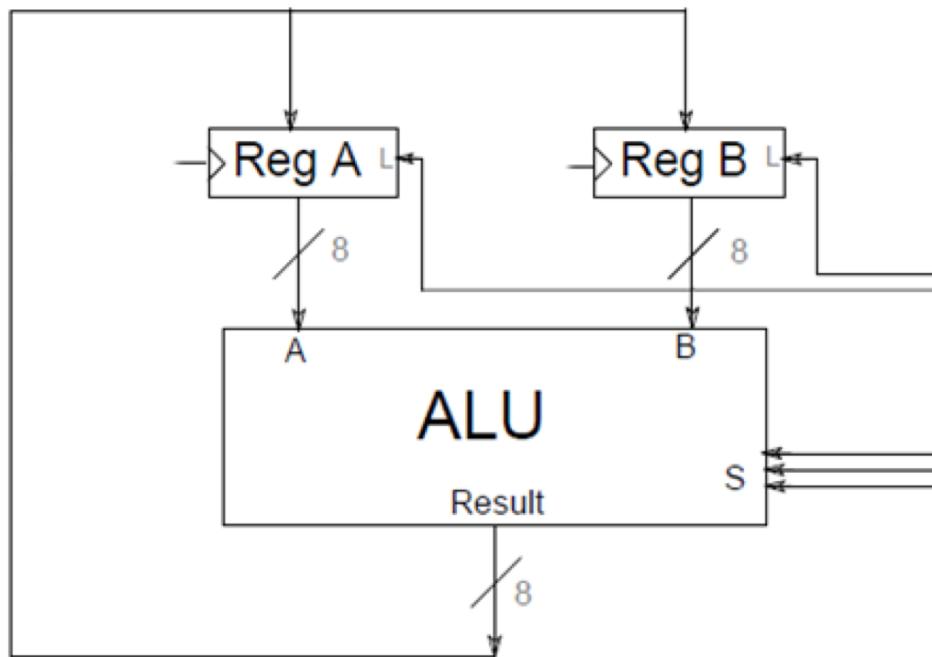
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

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

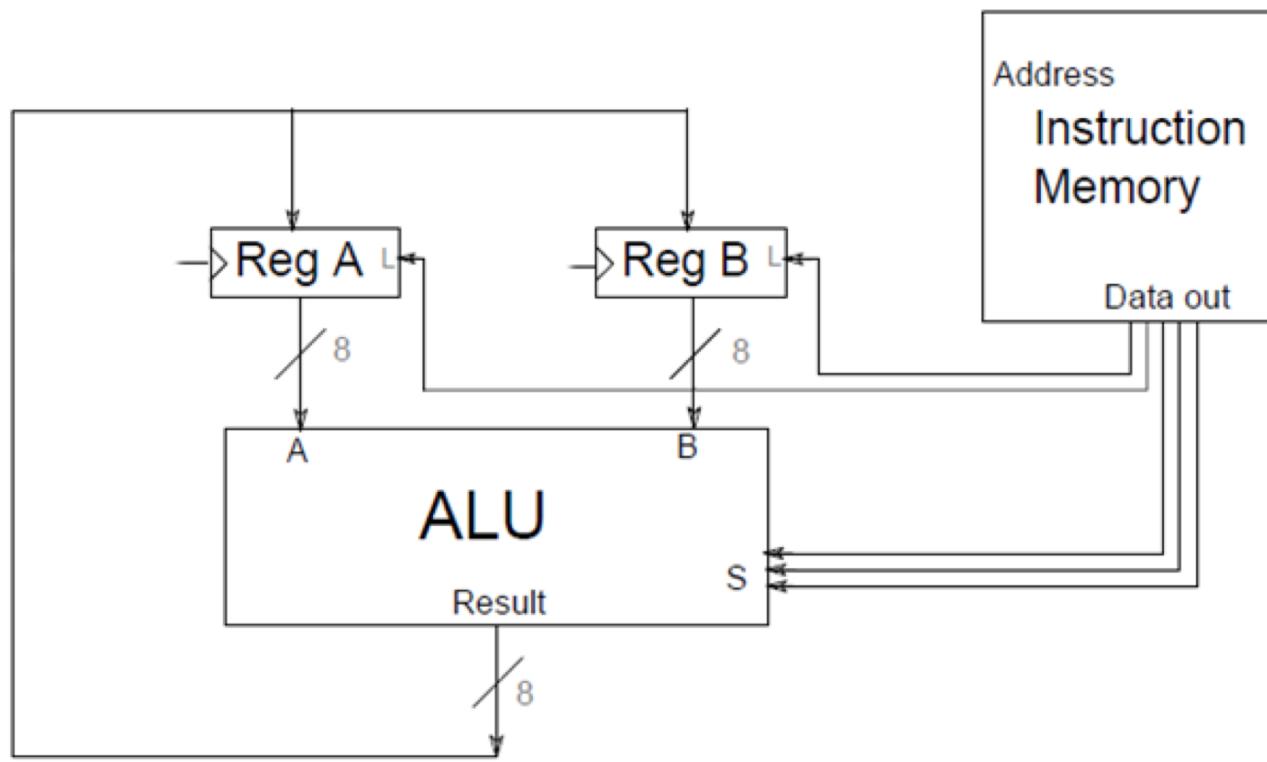
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	0	1
1	0	0	0	0	$A = A + B$	1	1
0	1	0	0	0	$B = A + B$	1	2
1	0	0	0	0	$A = A + B$	3	2
0	1	0	0	0	$B = A + B$	3	5
1	0	0	0	0	$A = A + B$	8	5
0	1	0	0	0	$B = A + B$	8	13

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 ...

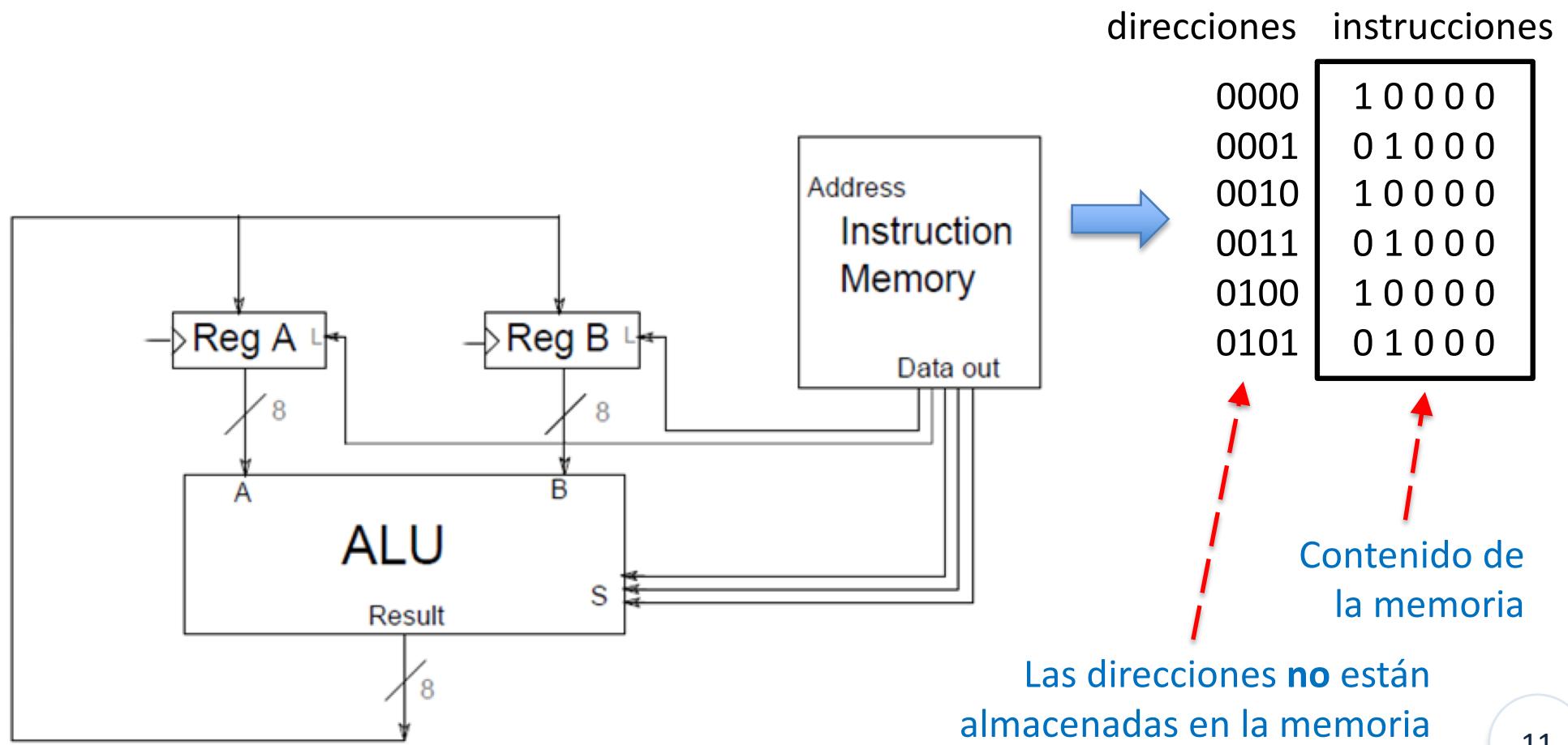


... en una memoria —que llamamos **Instruction Memory**: tiene un input, *Address* (que ya vamos a ver), y un output, *Data out*, que es por donde “sale” la instrucción (la combinación de señales de control) que hay que ejecutar a continuación



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

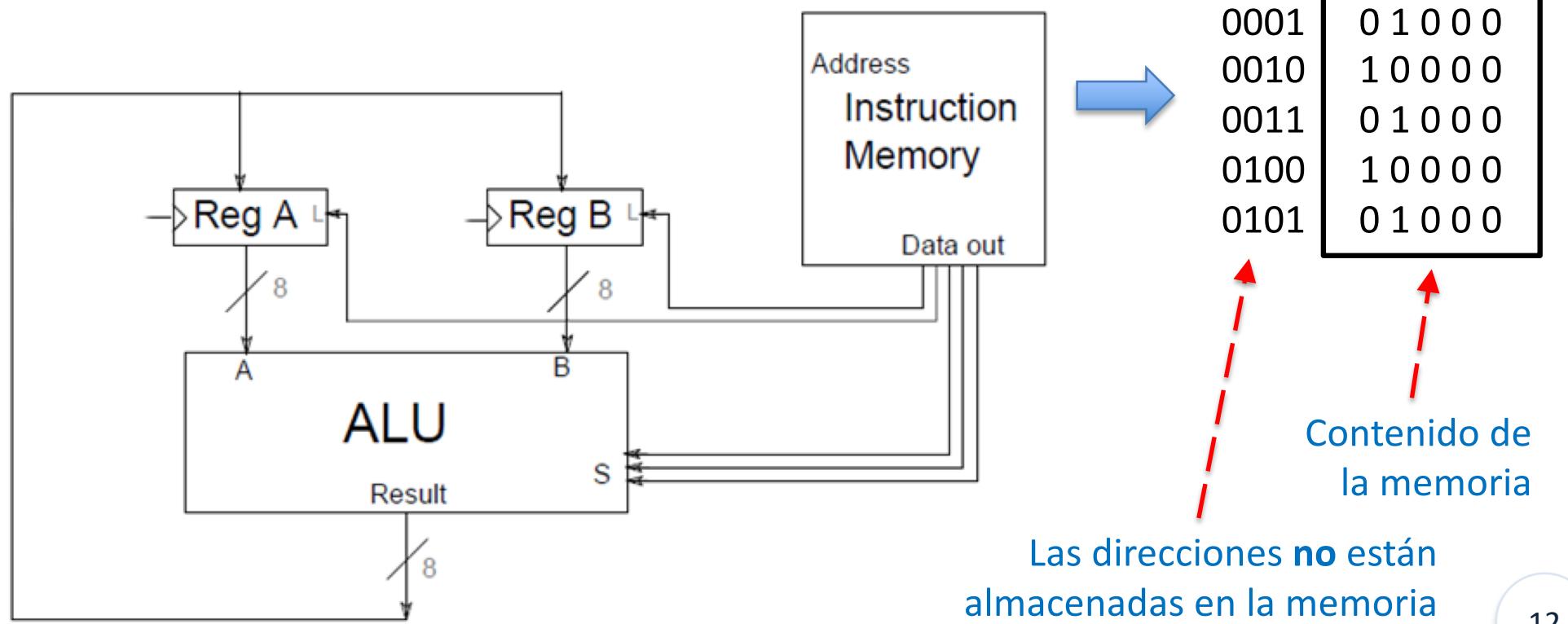
Estas posiciones correlativas, similares a los índices de un arreglo, se llaman **direcciones** (de memoria)



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

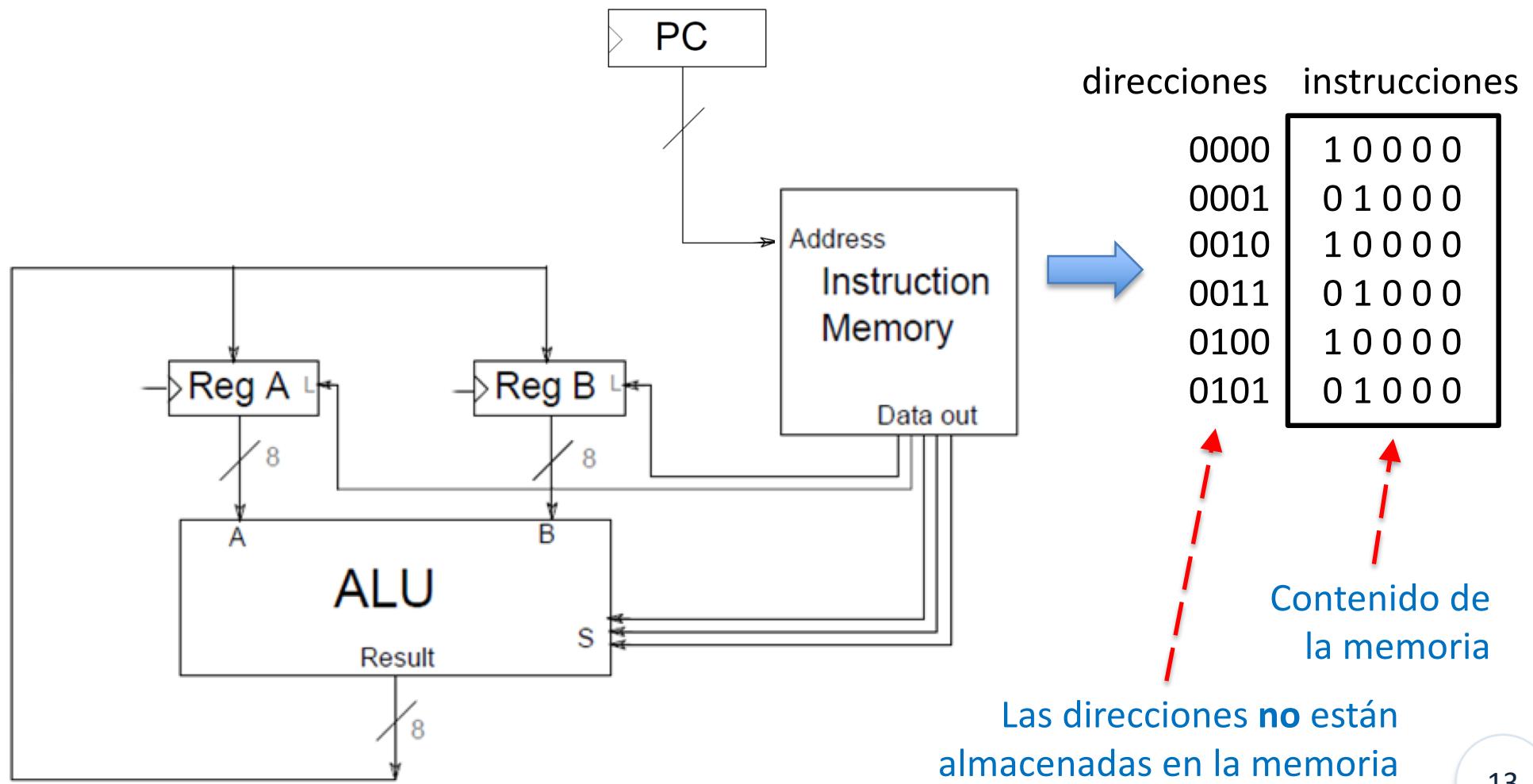
... es decir, que por *Data out* las instrucciones vayan apareciendo una por una en el orden en que tienen que ser ejecutadas

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



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*



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

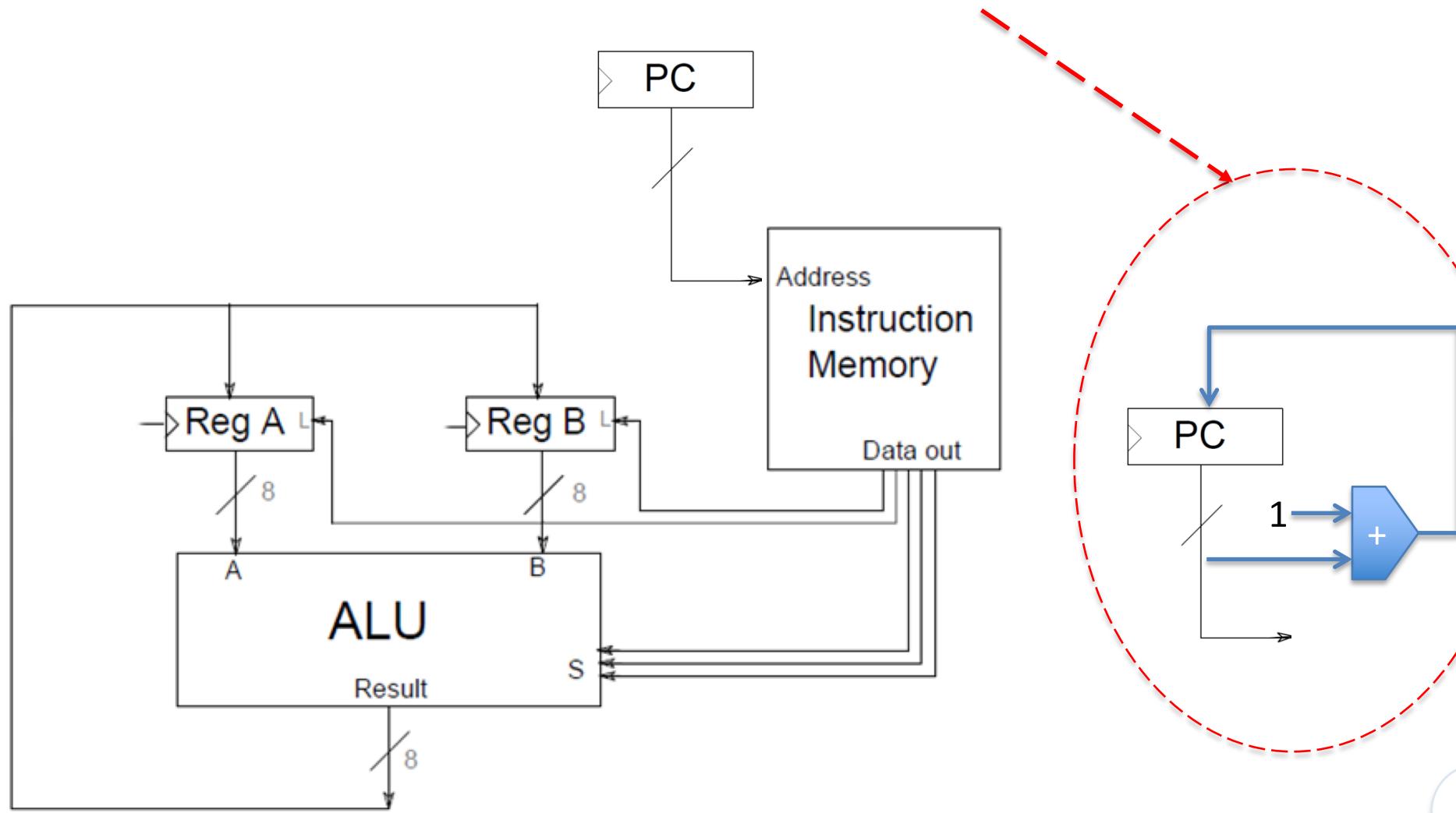
Las 4 líneas de *Address* entran a un **circuito decodificador** —como el de la diap. #47 de “logicaDigital”, pero con 4 líneas de input en vez de 2 (N_i) y 16 líneas de output en vez de 4 (D_j)

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*

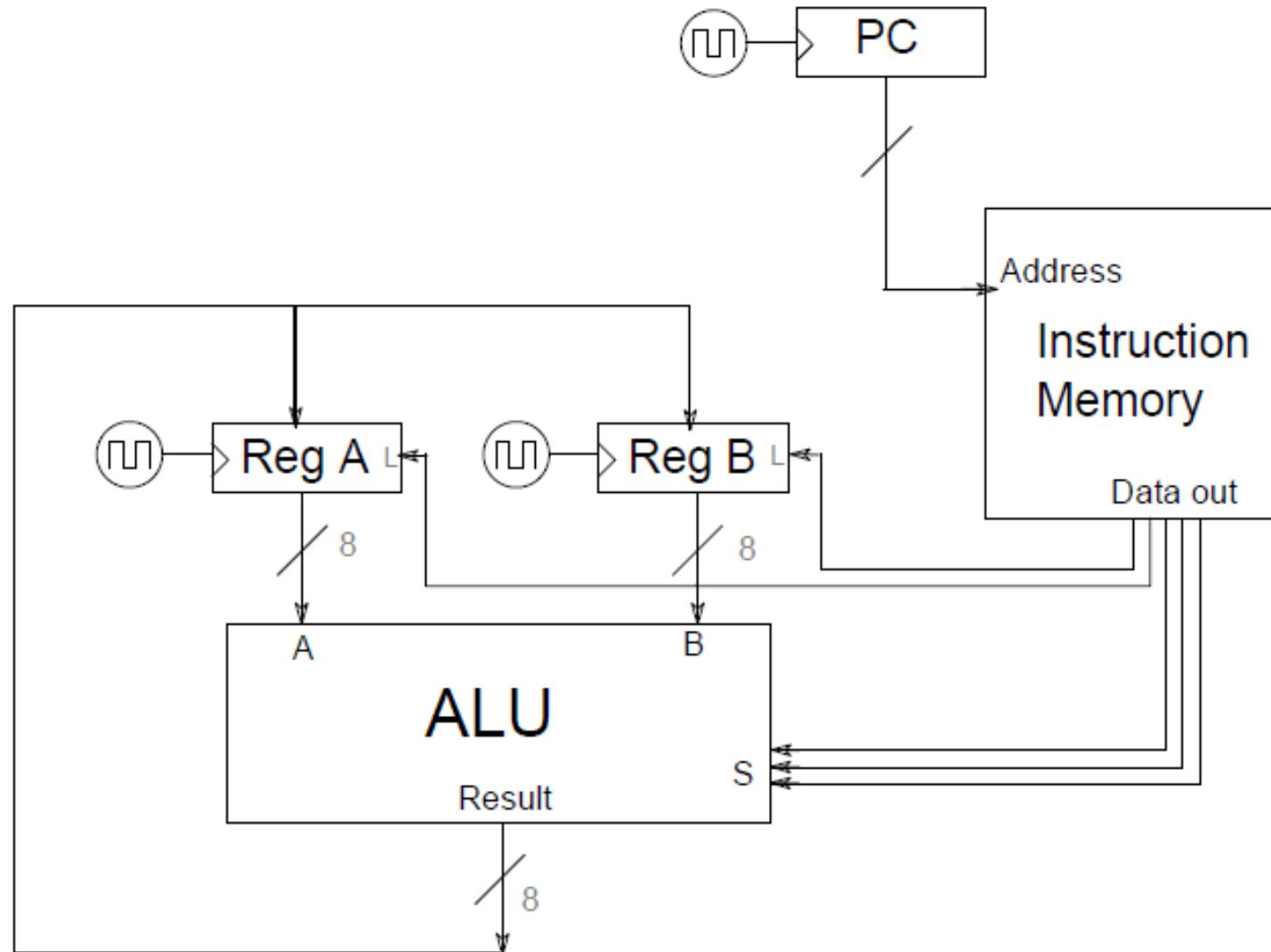
El registro *PC* tiene que ir incrementando automáticamente su contenido para que el programa se ejecute por completo sin más intervención nuestra que a la partida:

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

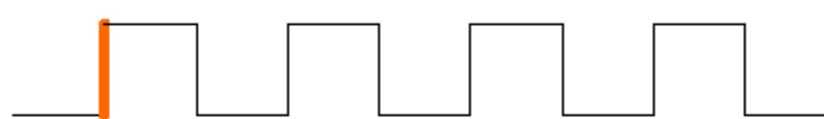


Finalmente, necesitamos que todas las acciones individuales ocurran sincrónicamente:

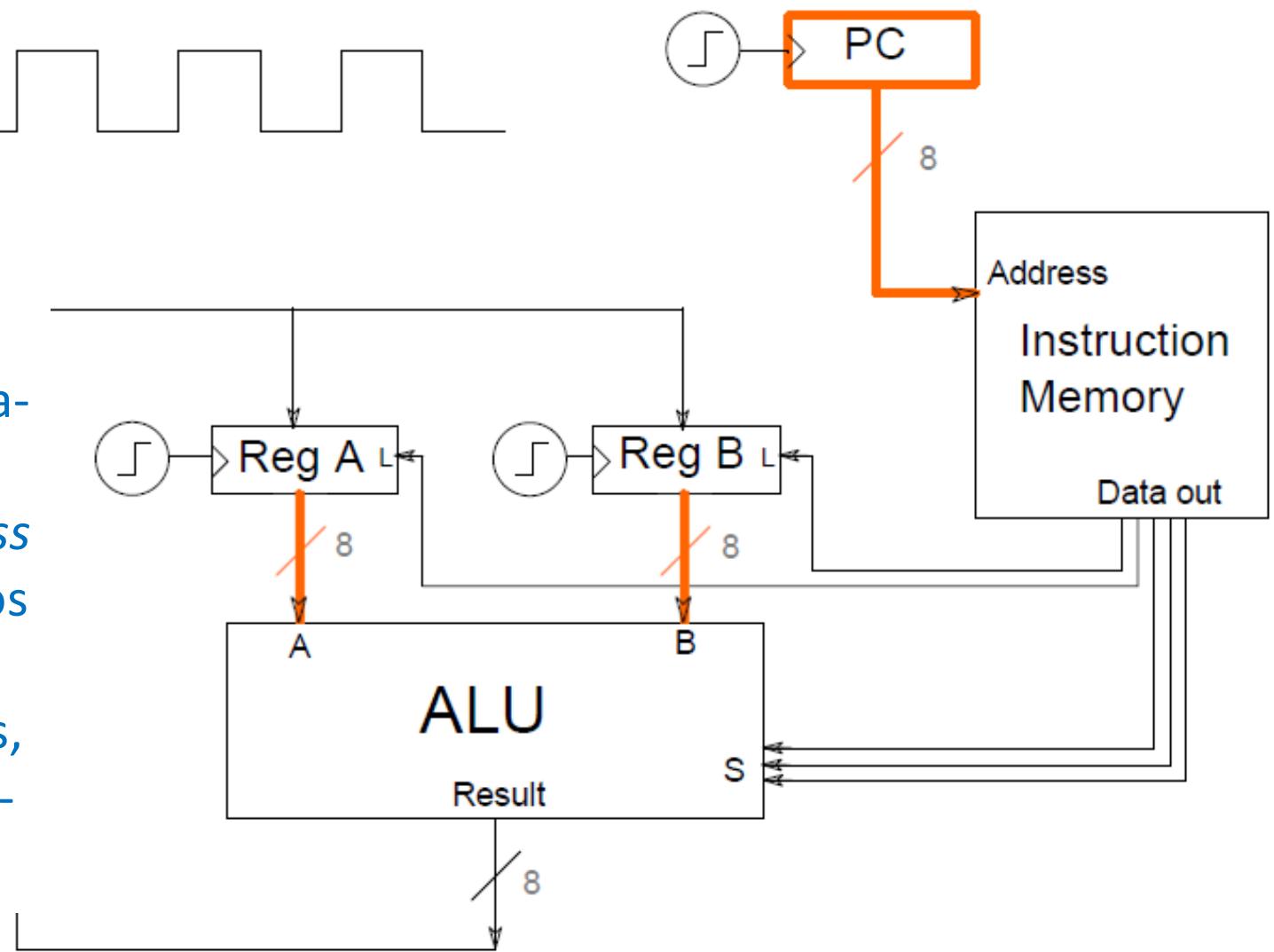
incluimos un *reloj* (*clock*)



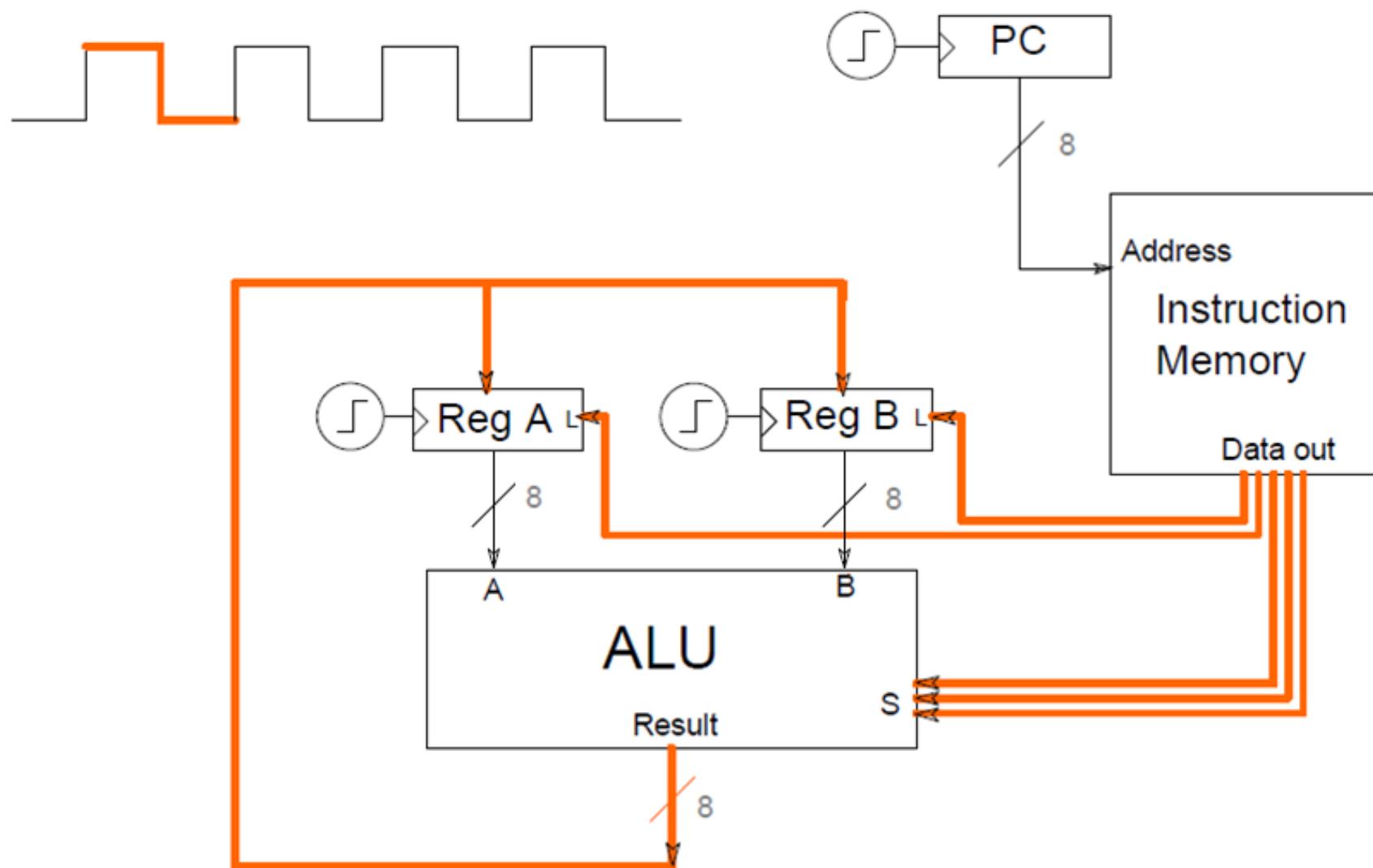
Reloj: Circuito que emite una serie de pulsos con un ancho preciso y un intervalo preciso entre pulsos consecutivos, y cuya frecuencia es controlada por un oscilador de cristal



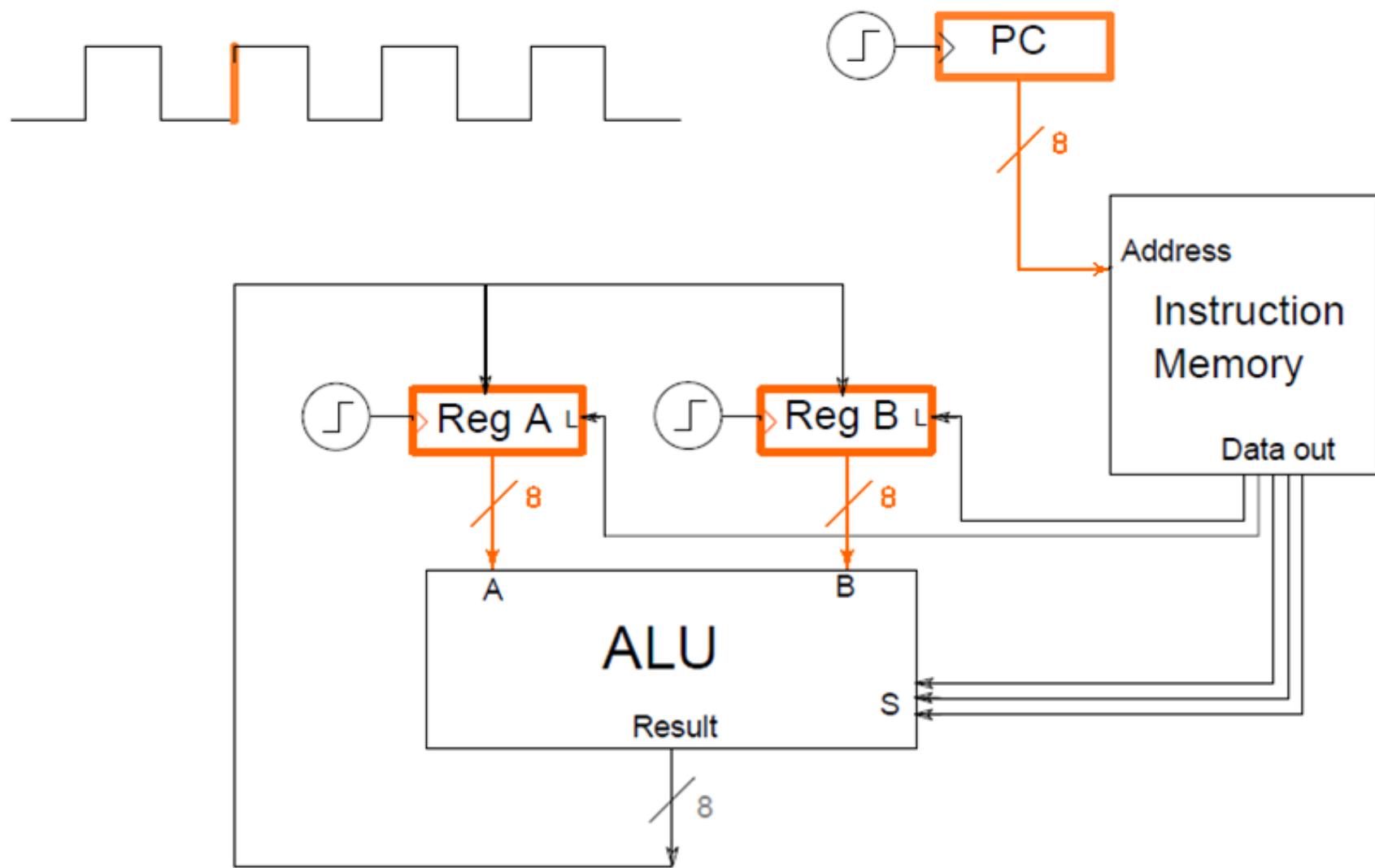
Cuando el pulso *sube* (*flanco de subida*), el *PC* actualiza su valor que va a la entrada *Address* de la memoria, y los registros *A* y *B* actualizan sus valores, que van a las entradas de la *ALU*



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

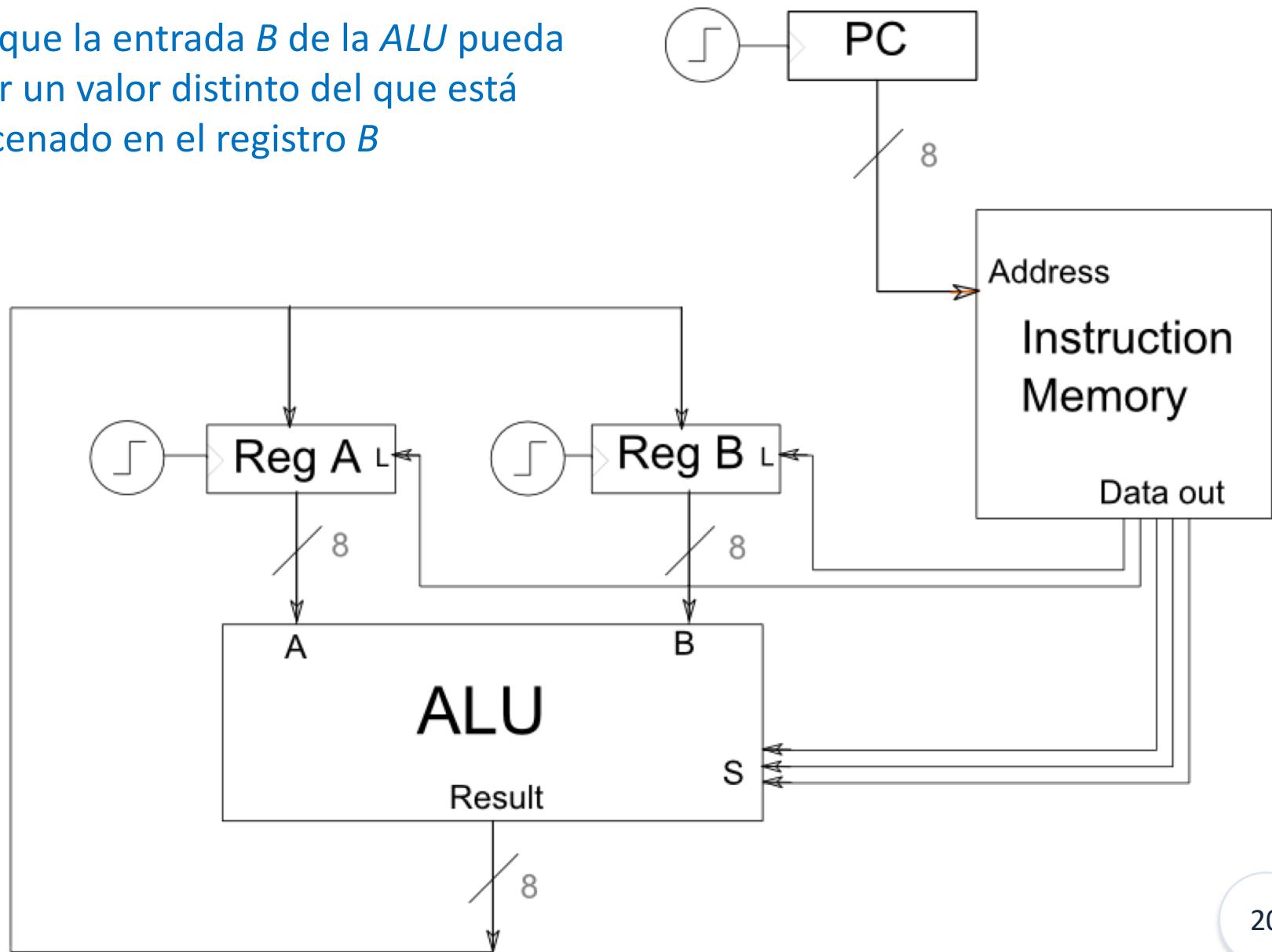


En el siguiente flanco de subida, PC pone una nueva dirección (la anterior más 1) a la entrada de la memoria, y los registros A y B ponen (posiblemente) nuevos valores en las entradas de la ALU



¿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*

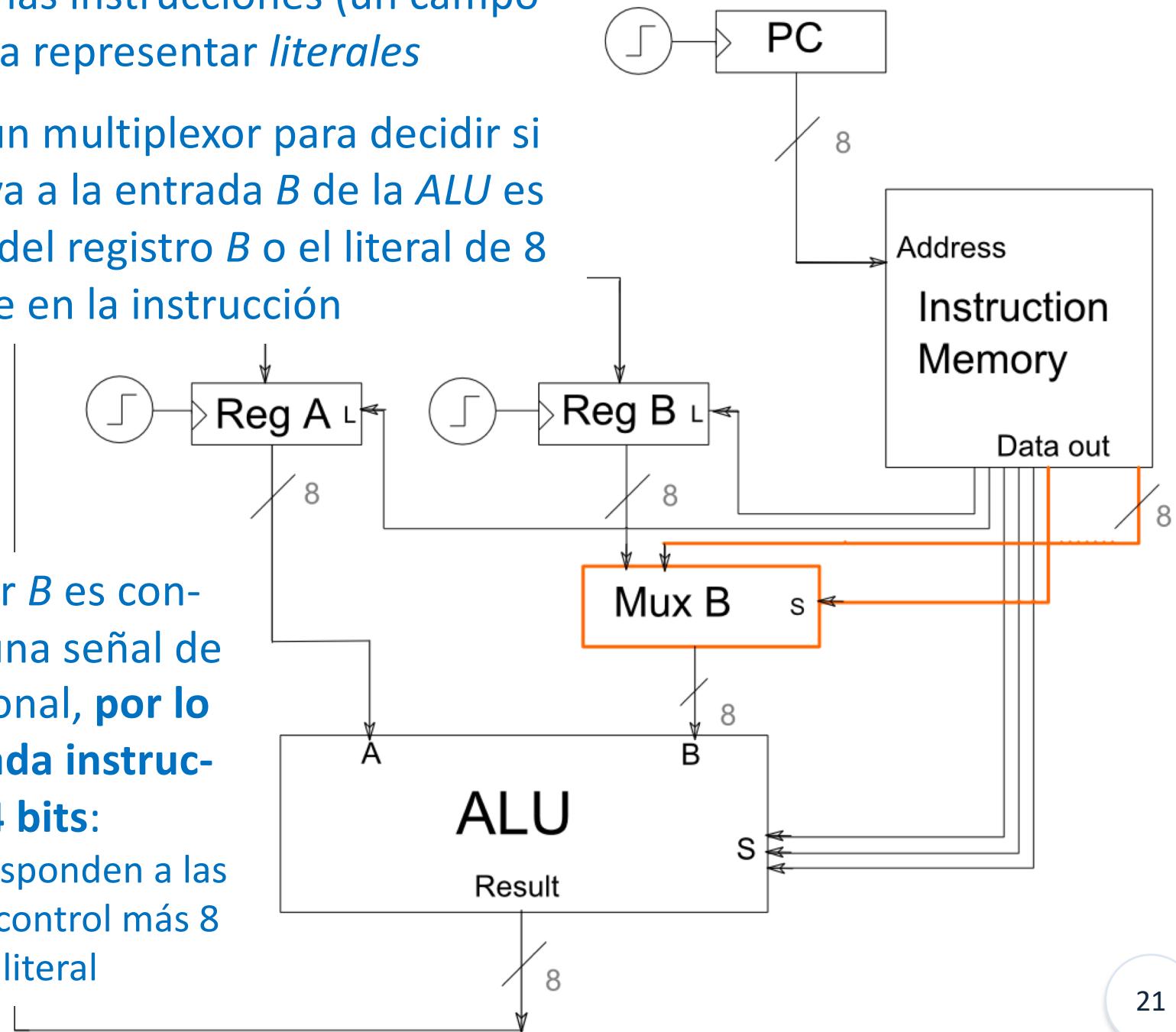


Agregamos a las instrucciones (un campo de) 8 bits para representar *literales*

... 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

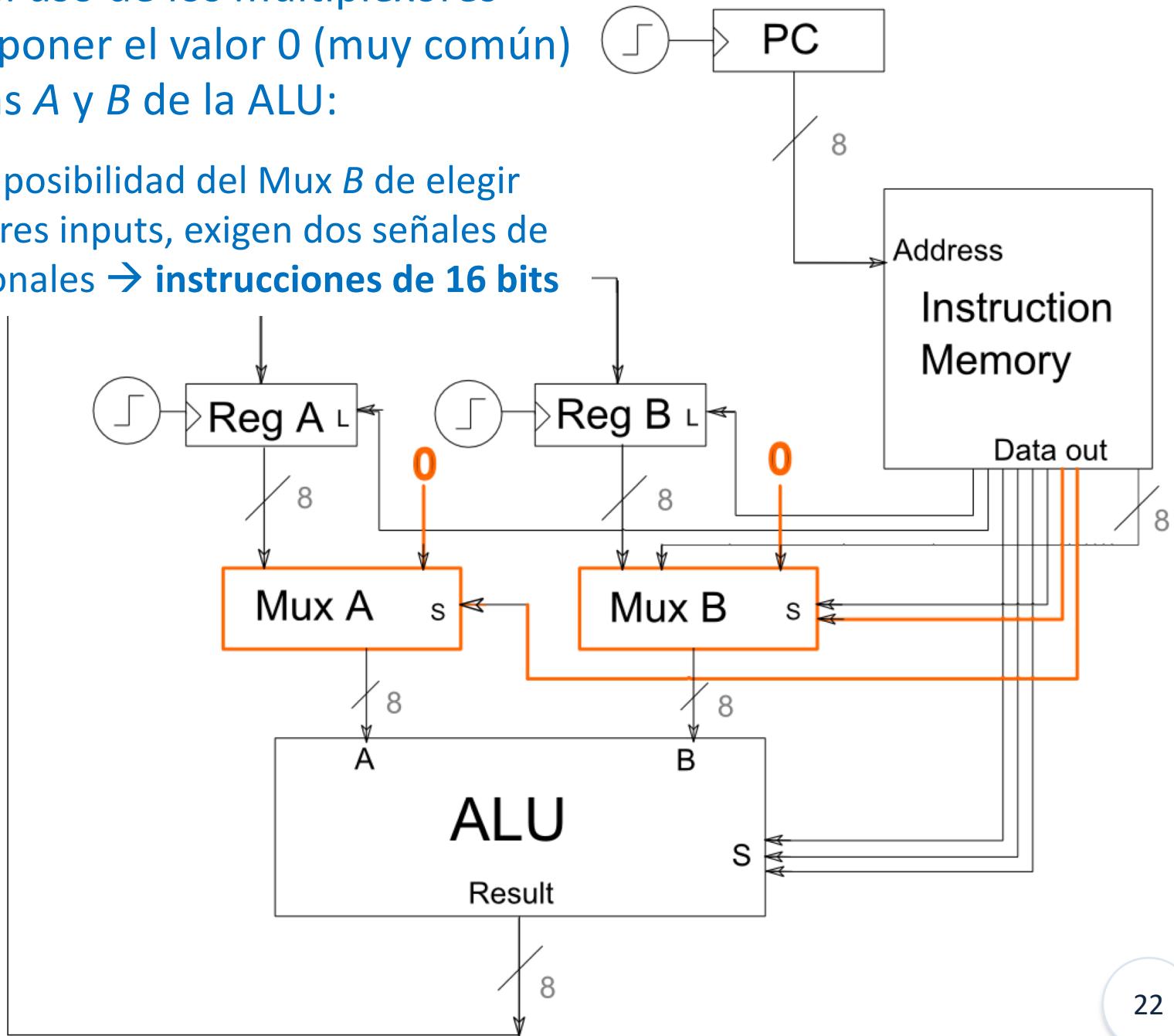
El multiplexor *B* es controlado por una señal de control adicional, **por lo que ahora cada instrucción tiene 14 bits**:

- 6 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**



Instrucciones de 16 bits de largo implica que la *Instruction Memory* debe estar formada por registros, o palabras de memoria, 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

