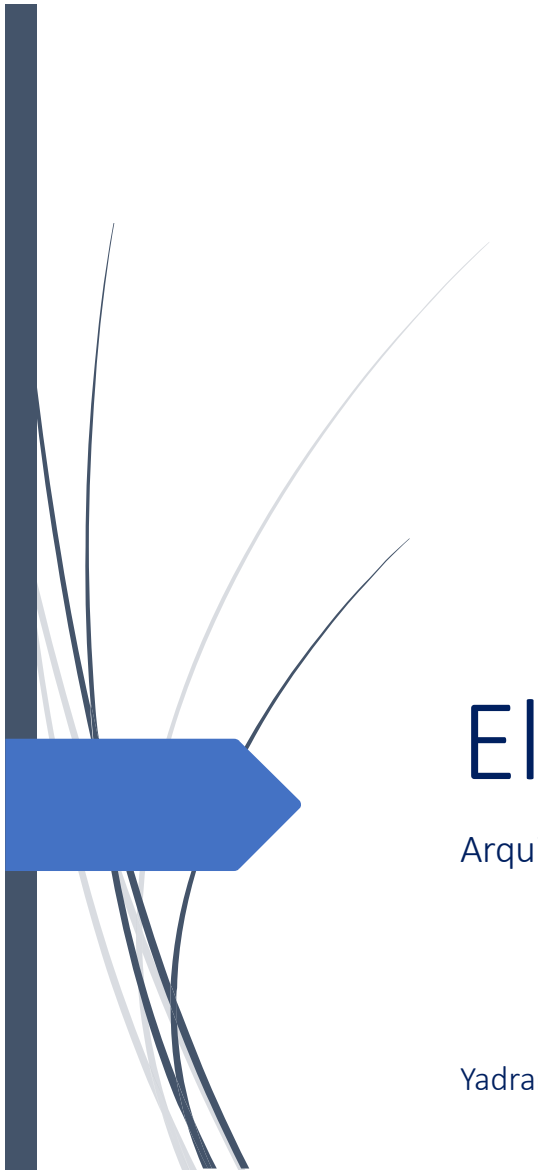


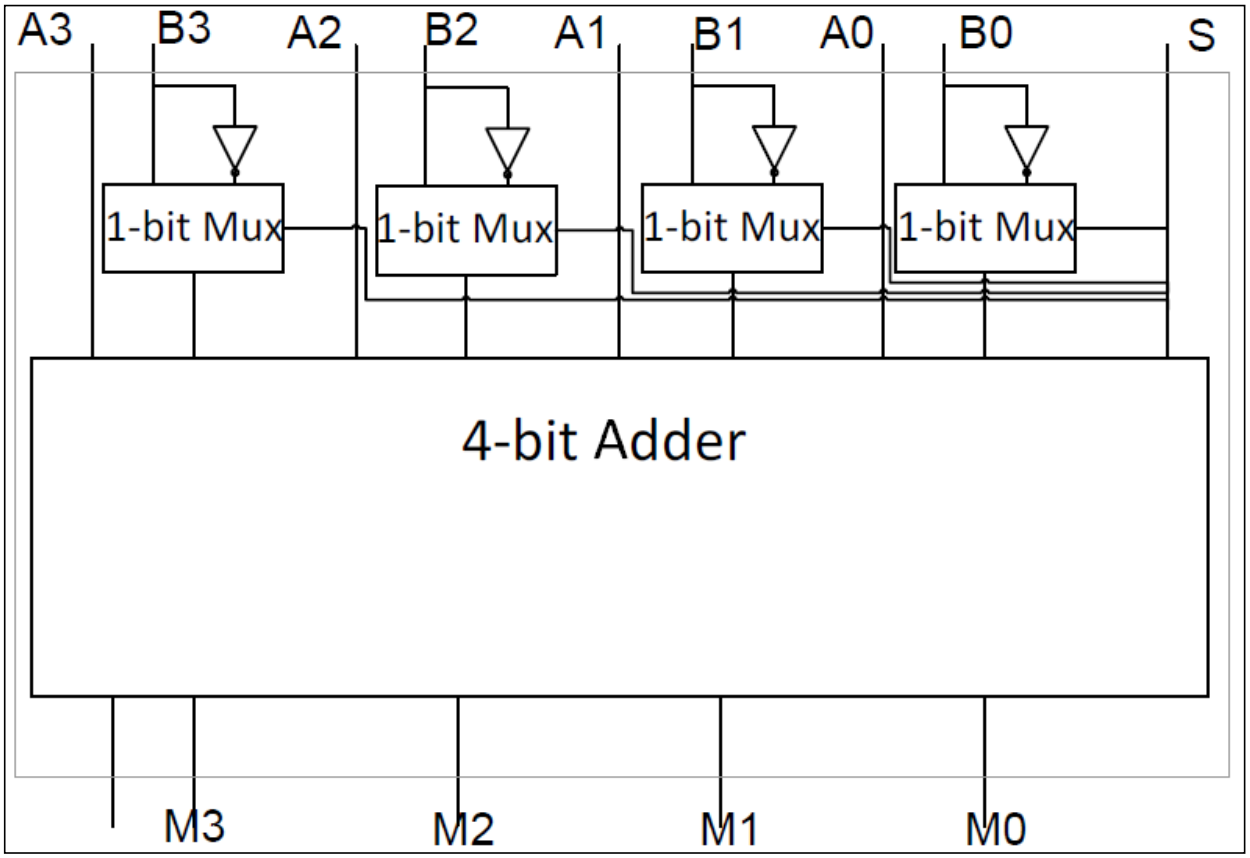
El procesador: Introducción

Arquitectura de Computadores – IIC2343

Yadran Eterovic S. (yadran@uc.cl)

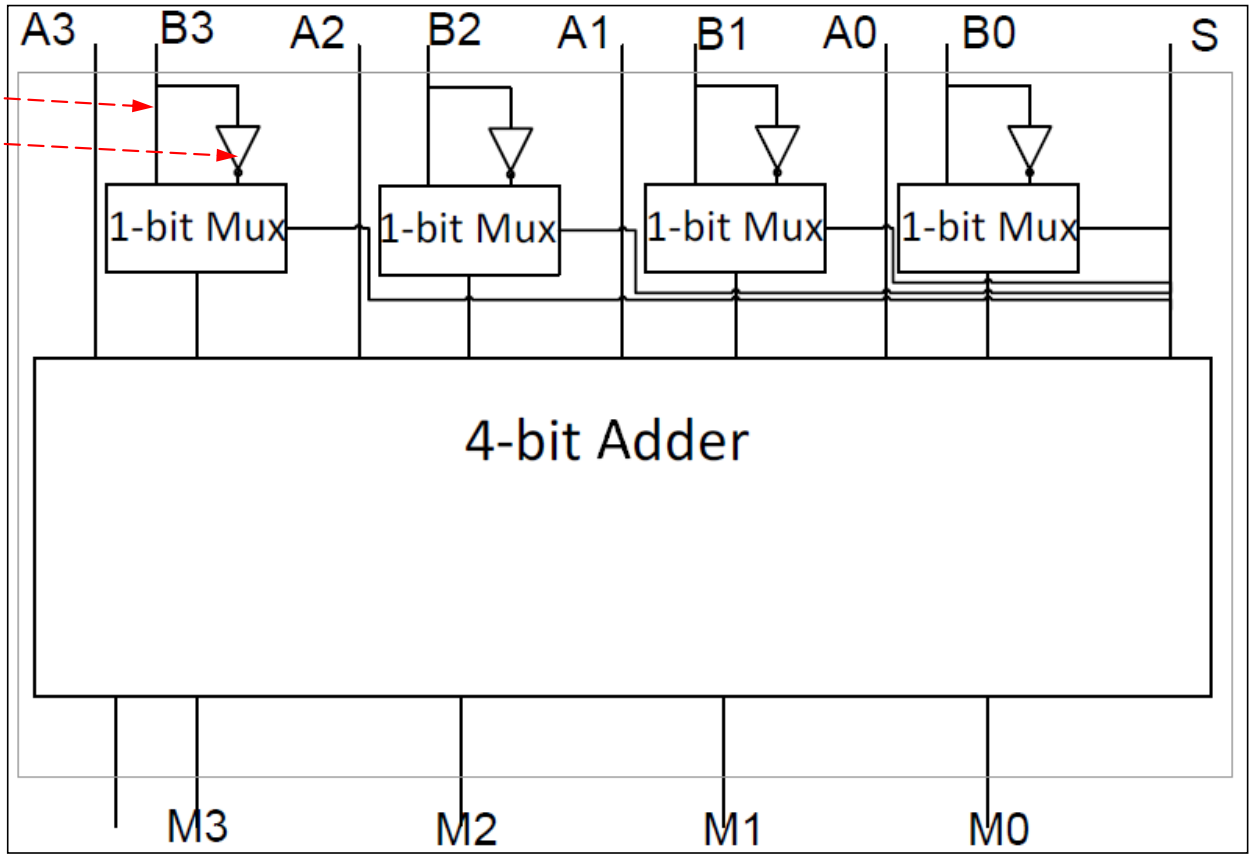
2025-2

Sumador/restador
de 4 bits



Sumador/restador de 4 bits

El input *B* puede entrar
- "directo" –en el caso de una suma
- invertido –en el caso de una resta

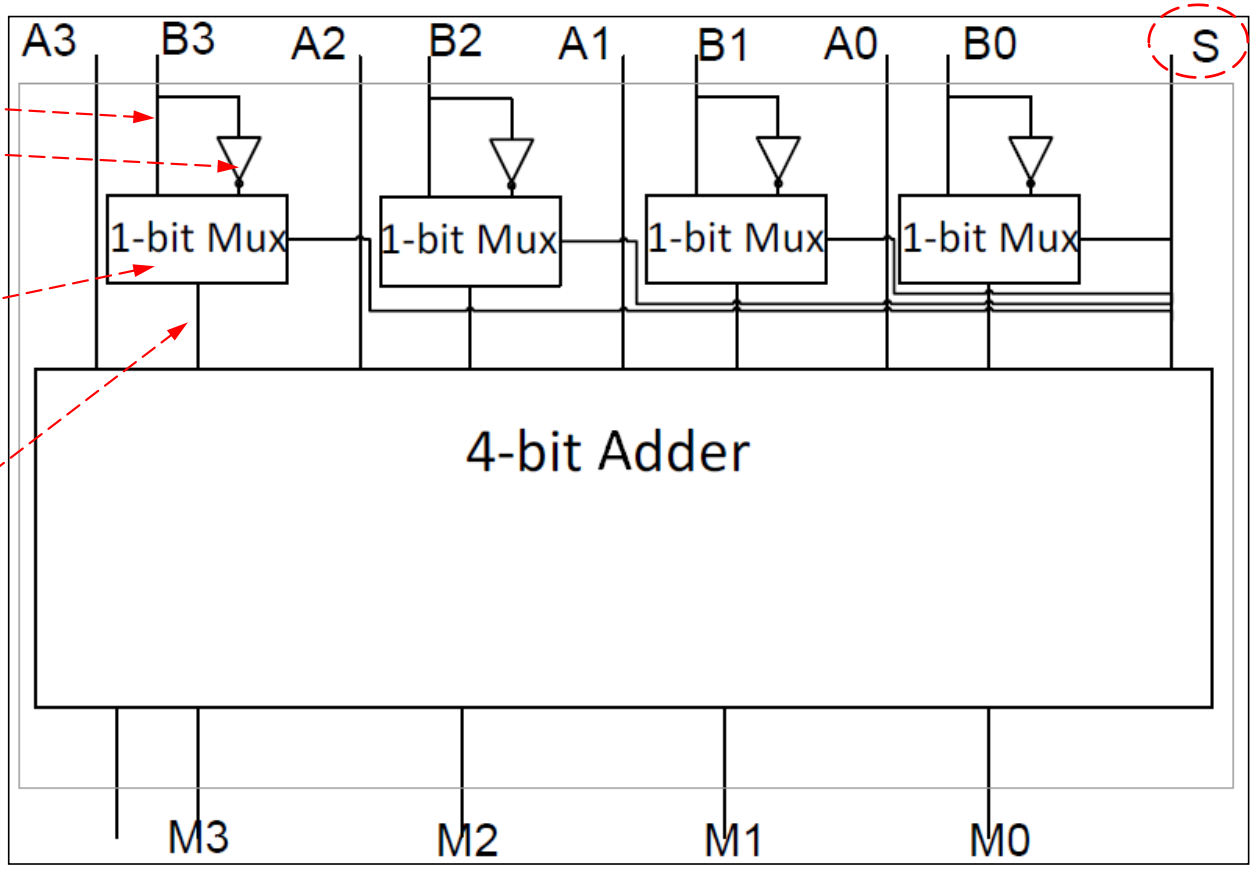


Sumador/restador de 4 bits

El input *B* puede entrar
- "directo" –en el caso de una suma
- invertido –en el caso de una resta

… por lo que ambas versiones pasan por un multiplexor controlado por el input de control *S* antes de entrar al circuito sumador:

- *S* = 0 ⟹ el *mux* selecciona la versión "directa" de *B*
- *S* = 1 ⟹ el *mux* selecciona la versión invertida de *B*

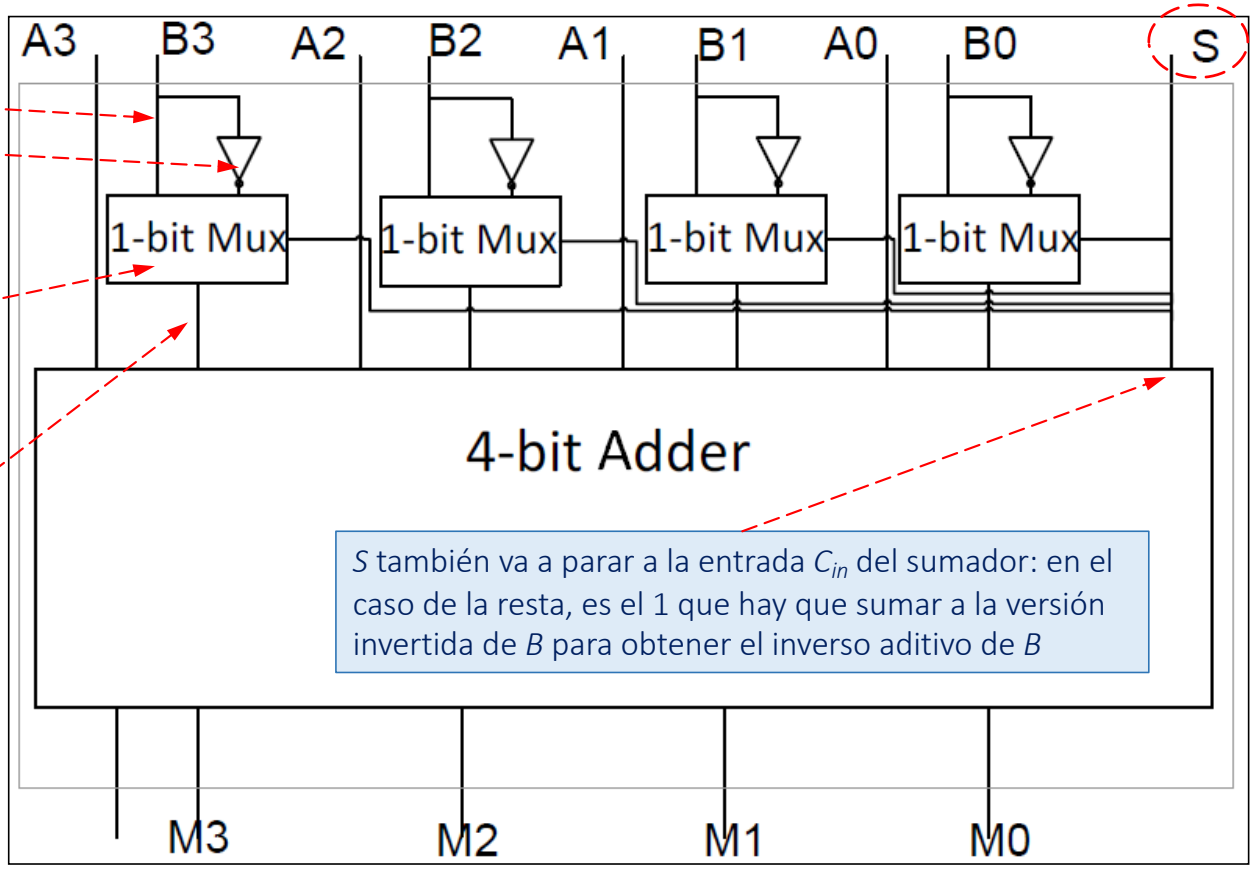


Sumador/restador de 4 bits

El input *B* puede entrar
- "directo" –en el caso de una suma
- invertido –en el caso de una resta

… por lo que ambas versiones pasan por un multiplexor controlado por el input de control *S* antes de entrar al circuito sumador:

- $S = 0 \Longrightarrow$ el *mus* selecciona la versión "directa" de *B*
- $S = 1 \Longrightarrow$ el *mux* selecciona la versión invertida de *B*

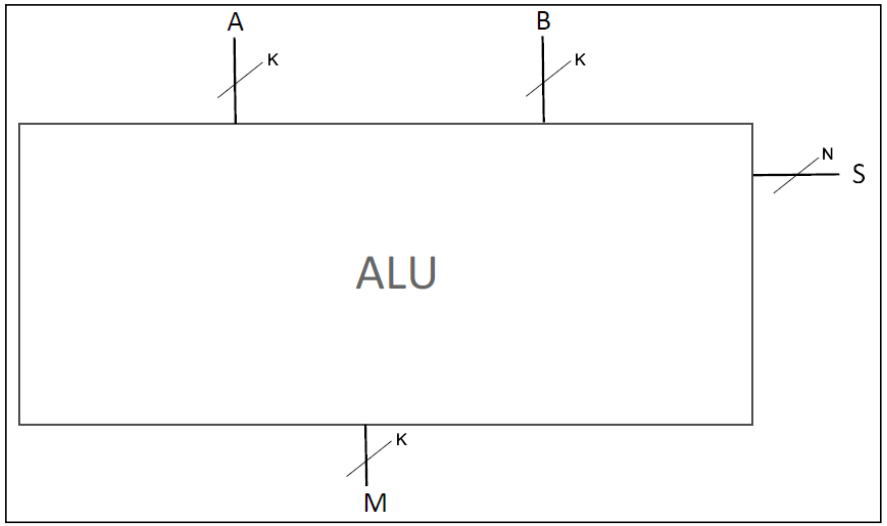


S también va a parar a la entrada C_{in} del sumador: en el caso de la resta, es el 1 que hay que sumar a la versión invertida de *B* para obtener el inverso aditivo de *B*

Todos los computadores tienen un circuito para realizar operaciones lógicas (p.ej., AND, OR) y aritméticas (p.ej., suma, resta) sobre dos operandos

→ la **unidad lógica aritmética** o ***ALU*** :

- operandos *A* y *B*, de *K* bits c/u (en la realidad, *K* = 32, 64, 128, … bits)
- resultado *M*, de *K* bits
- input de control *S*, de *N* bits, para seleccionar entre 2^N operaciones distintas



P.ej., el circuito sumador / restador de la diap. anterior correspondería a una *ALU* muy simple que sólo suma o resta operandos de *K* = 4 bits

⟹ sólo dos operaciones, por lo que *S* tiene sólo un bit (*N* = 1)

La ALU también realiza las operaciones lógicas básicas sobre sus inputs:

NOT AND OR XOR

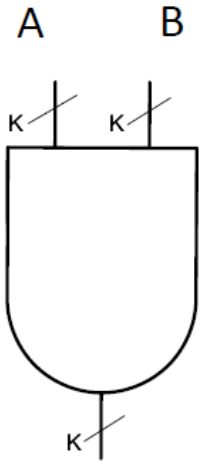
Son muy útiles por sí mismas y formando parte de otras operaciones más complejas

Se realizan bit a bit (*bitwise*)

K = 8 bits

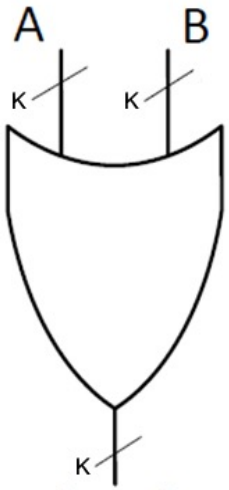
A = 0 1 1 0 0 1 0 1

B = 1 1 0 1 0 0 0 1



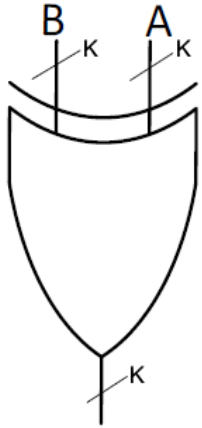
A and B

0 1 0 0 0 0 0 1



A or B

1 1 1 1 0 1 0 1



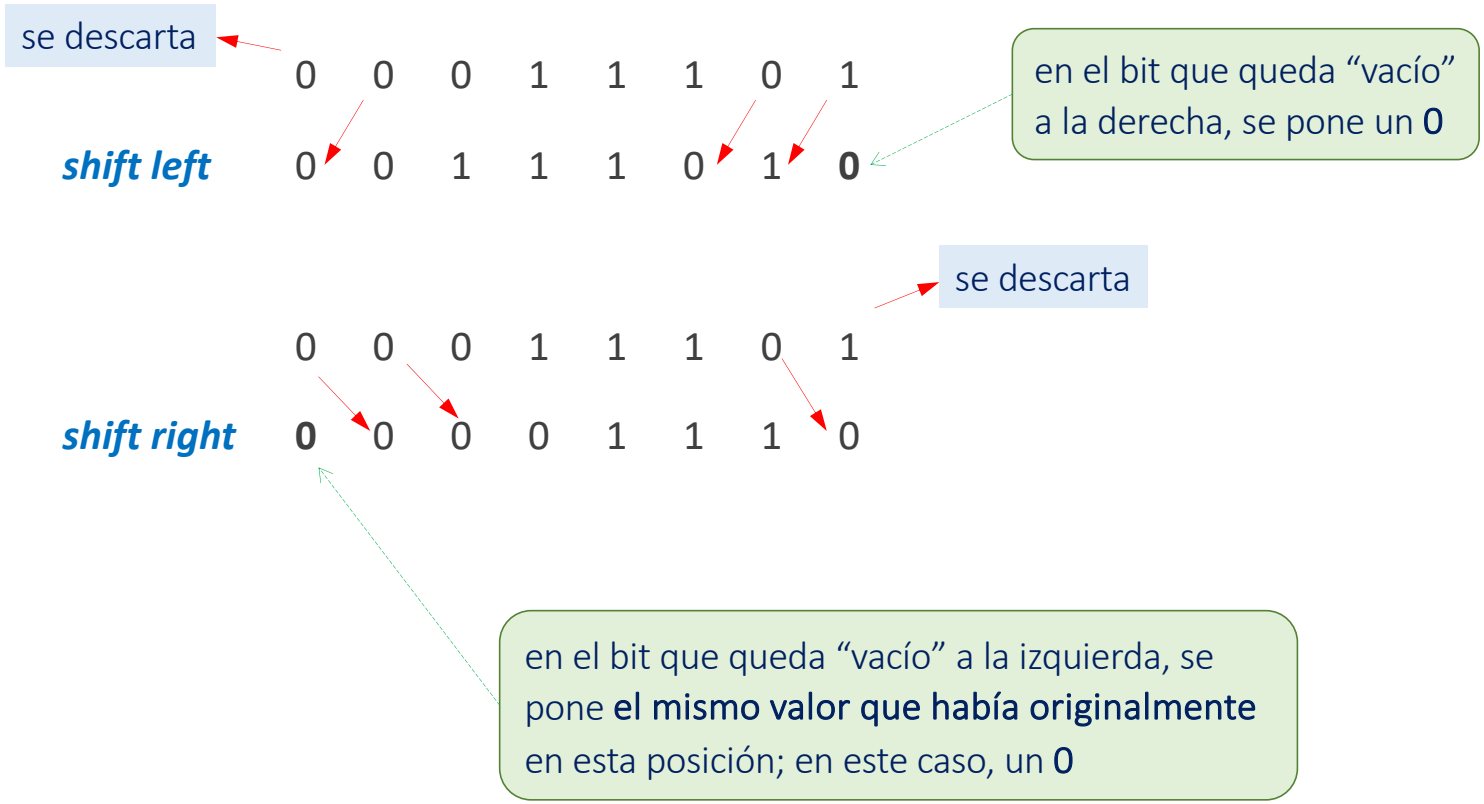
A xor B

1 0 1 1 0 1 0 0

not(A)

1 0 0 1 1 0 1 0

Las dos últimas operaciones que vamos a implementar en nuestra *ALU* básica son *shift left* y *shift right*



Las operaciones *shift* pueden parecer arbitrarias, pero son muy útiles

P.ej., podemos verlas como la **multiplicación por 2** (*shift left*)

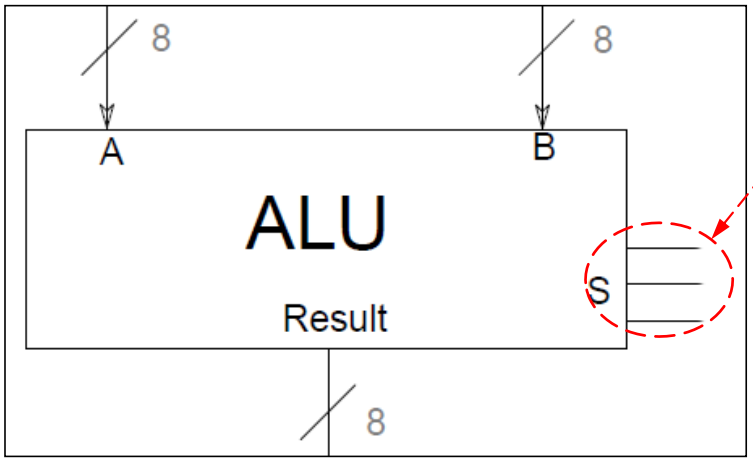
… y la **división** (entera) **por 2** (*shift right*)

… análogas, en base 10, a una multiplicación por 10 (se agrega un cero a la derecha de la secuencia de dígitos) y una división entera por 10 (se descarta el dígito de más a la derecha):

$47293 \times 10 = 472930$ $47293 / 10 = 4729$

(En *shift right*, la razón de "llenar" el bit de más a la izquierda con el mismo valor que había allí originalmente, es mantener el signo —positivo o negativo— del número representado por el patrón de bits)

ALU básica de 8 operaciones, dibujada esquemáticamente para dos inputs de datos de 8 bits (***A*** y ***B***), un input de control de 3 bits (***S***), y un output de 8 bits (***Result***)



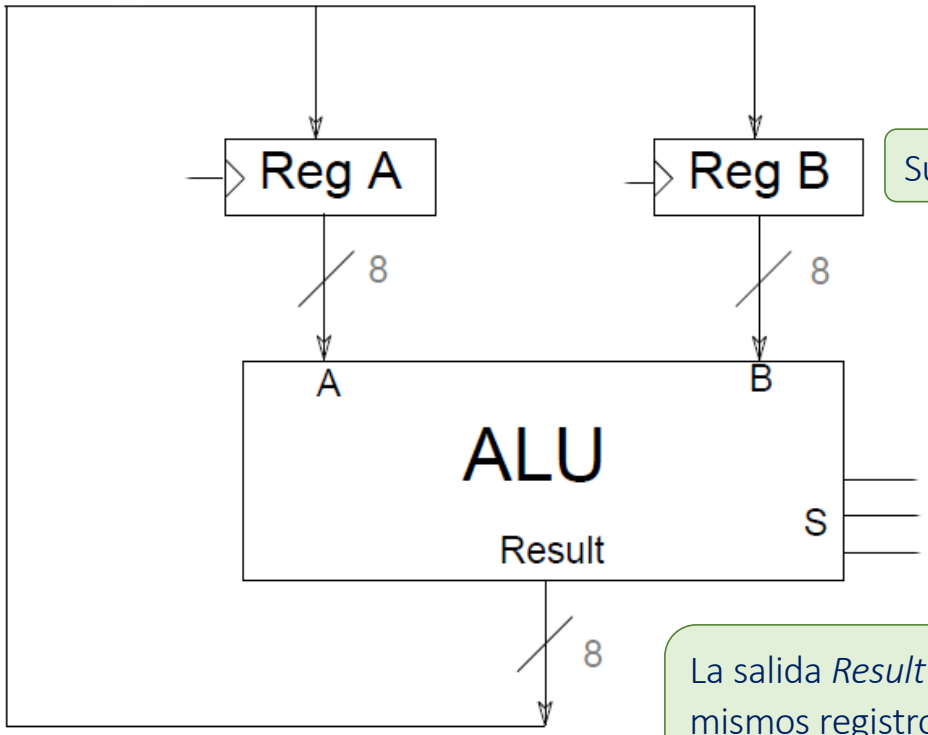
S2	S1	S0	Result
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

El output ***Result*** va a contener el resultado de la ejecución de una de las 8 operaciones, según la combinación de valores en los tres bits del input de control, $S_2\ S_1\ S_0$; p.ej.:

0 0 1 $\Longrightarrow$ *A* – *B* 0 1 1 $\Longrightarrow$ *A* OR *B* 1 1 1 $\Longrightarrow$ *shift right*(*A*)

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

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



Suponemos registros de 8 bits

La salida *Result* va a parar a esos mismos registros, *RegA* y *RegB* (a partir de ahora, registros *A* y *B*)

Una componente esencial de todo computador es su **memoria**

Sin memoria, no habría computadores tal como los conocemos hoy día

La memoria se usa para almacenar tanto las **instrucciones** a ser ejecutadas

… como los **datos** que se usan en la ejecución de las instrucciones

… y los que se producen como resultados de esa ejecución

Para tener una memoria (de un bit), necesitamos un circuito que recuerde valores de *input* previos —es decir, que **almacene su estado**

El *output* del circuito depende ahora tanto de los datos de *input*

… **como del valor almacenado dentro del circuito**

Esto se puede conseguir gracias a que cada compuerta tiene un pequeño *retardo de propagación*:

- hay un retardo entre el instante en que el *input* cambia y el instante en que el *output* cambia consecuentemente

P.ej., efecto del retardo de propagación en un circuito muy simple:

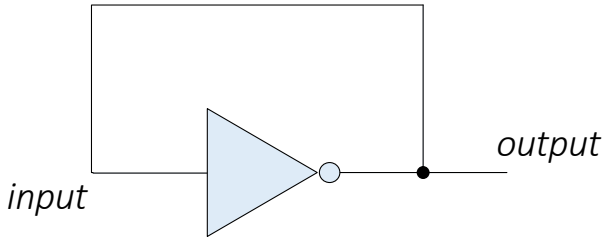
- si el *output* del circuito es 0, entonces el *input* del circuito es 0

  … esto parece una contradicción, ya que la (única) compuerta del circuito es un inversor (NOT)

  … excepto que sólo una vez que ha transcurrido el retardo de propagación (menos de un nanosegundo), el *output* cambia a 1

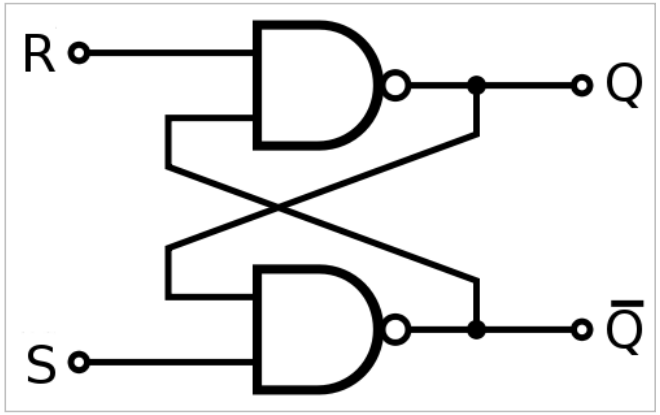
  … y sólo una vez que ha transcurrido otro retardo de propagación, el *output* vuelve a 0 nuevamente

- en principio, este ciclo continúa para siempre, por lo que el circuito oscila: el *output* cambia una y otra vez entre 0 y 1



El funcionamiento de un ***latch S-R*** se basa en el retardo de propagación:

- construido a partir de dos compuertas NAND (o dos compuertas NOR)
- dos inputs, *S* (*set*) y *R* (*reset*)
- dos *outputs*, *Q* y ¬*Q*, complementarios
- los *outputs* no están determinados únicamente por los inputs vigentes —no es un circuito *combinacional* … sino ***secuencial***
- si $R = S = 0$ ($\Rightarrow Q = \neg Q = 1$), el circuito se vuelve no determinista cuando *R* y *S* vuelvan a 1

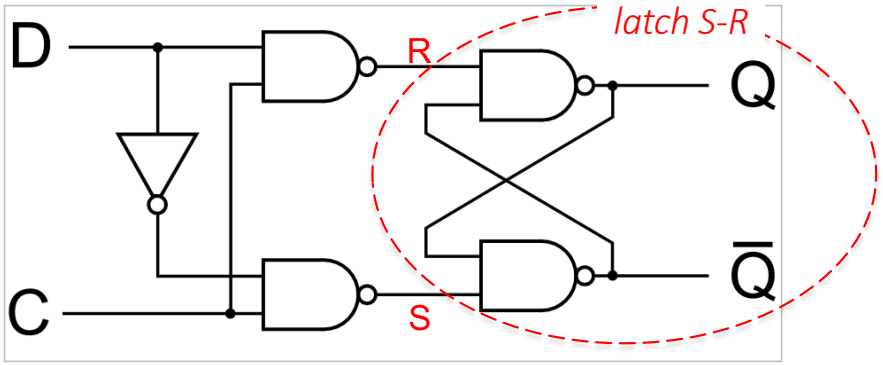


R	S	Q^{t+1}
0	0	-
0	1	1
1	0	0
1	1	Q^t

Latch D (*controlado por C*) —para permitir que el *latch* cambie de estado sólo en ciertos instantes específicos:

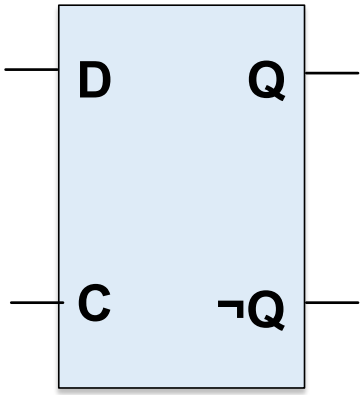
- evita el caso no determinista del *latch S-R* previniendo su ocurrencia
- el input *D* y su complemento ¬*D* entran a sendas compuertas NAND justo antes del *latch S-R*

  $\Longrightarrow$ los inputs *R* y *S* del *latch S-R* no pueden ser ambos 0 simultáneamente
- cuando *C* está en 1, el *latch* "se carga con" el valor de *D*
- el valor almacenado en el *latch* está siempre disponible en el output *Q*



C	D	Q^{t+1}
0	0/1	Q^t
1	0	0
1	1	1

Latch D (controlado por *C*) —más esquemáticamente



C	D	Q^{t+1}
0	0/1	Q^{t}
1	0	0
1	1	1

D: dato (*input*)
C: control (p.ej., un reloj, como ya veremos)
Q: estado (*output*)

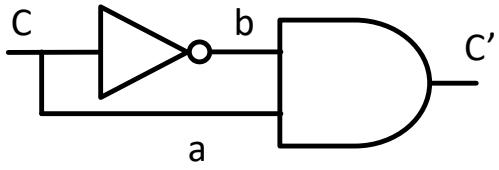
En el diseño y construcción de computadores digitales, necesitamos circuitos que puedan leer el valor de una señal en un instante particular en el tiempo y almacenarlo:

- esto elimina los problemas que podrían ocurrir si varias señales fueran leídas en momentos en el tiempo levemente diferentes

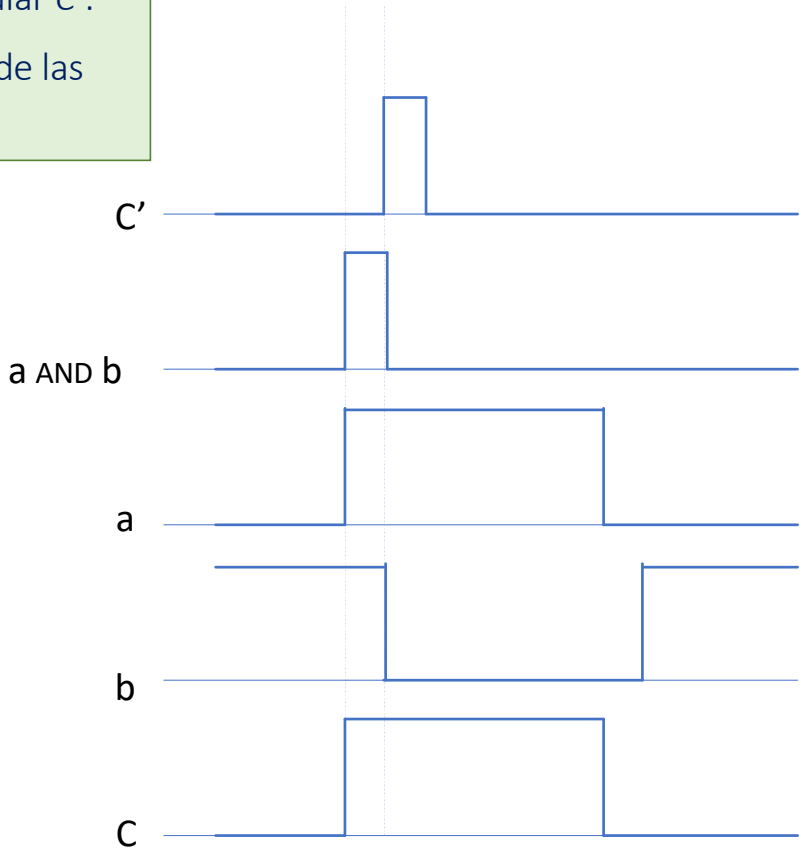
  … esto podría ocurrir en el caso de los *latchs* como el anterior, en que la señal *D* puede ser leída en cualquier instante mientras *C* está en su valor alto, o 1 (o, equivalentemente, en su valor bajo)

Una posibilidad es “estrechar” al máximo el intervalo de tiempo durante el cual *C* está en su valor alto, p.ej., usando un generador de pulsos *C’* a partir de un oscilador rectangular *C* :

el ancho del pulso *C’* es el retardo de propagación de las compuertas



generador de pulsos



Estos circuitos secuenciales se conocen como ***flip-flops*** y pueden ser construidos de diferentes maneras

Lo importante es que la transición de estado —es decir, el almacenamiento en el *latch* del valor que hay en el input *D*

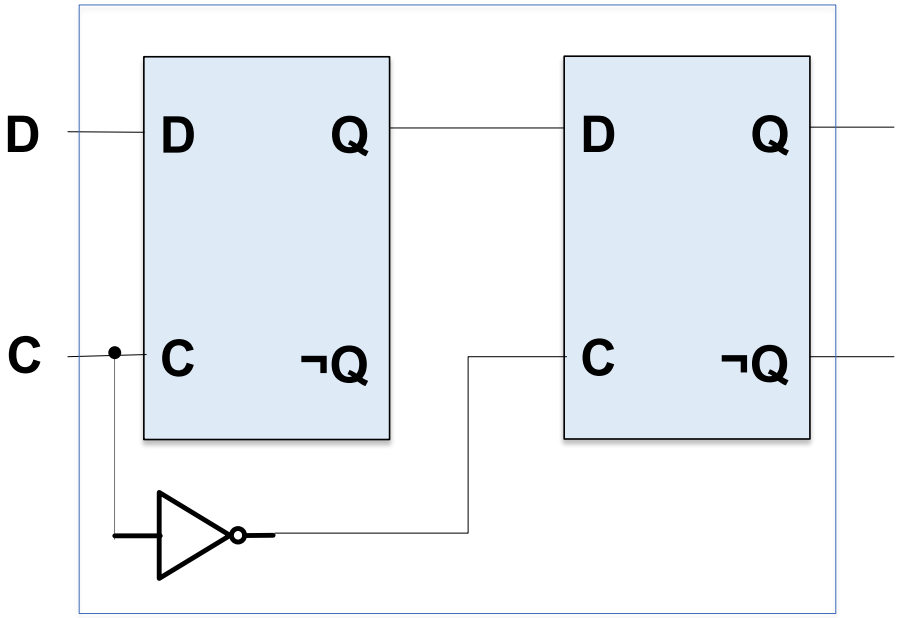
… ocurre, en la práctica, durante la transición de *C* de 0 a 1, o *flanco de subida*

… y no cuando *C* está en 1

(la transición de estado también podría ocurrir en el *flanco de bajada*)

Flip-flop D:

- construido a partir de dos *latches D* y un inversor (compuerta NOT)
- cuando *C* cambia de 1 a 0 (o, en otros circuitos equivalentes, de 0 a 1), el *output Q* almacena el valor del *input D*



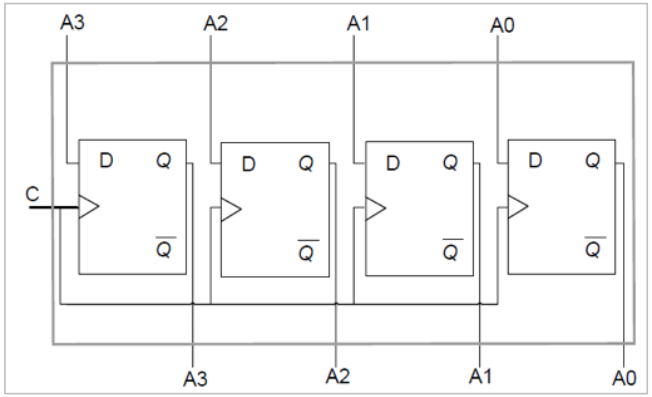
C	D	Q^{t+1}
0/1/↑	0/1	Q^t
↓	0	0
↓	1	1

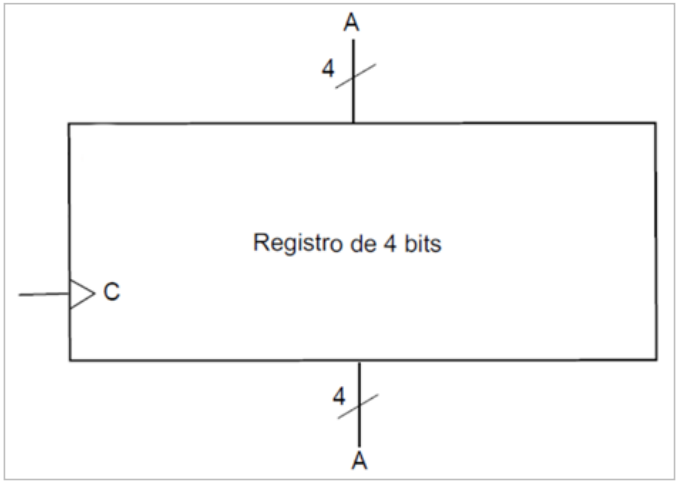
Ir a diap. 34

Usamos un arreglo de flip-flops D para construir un **registro** …

… que puede almacenar un dato de varios bits, tal como un *byte* (8 bits) o una palabra (32 o 64 bits en los computadores modernos):

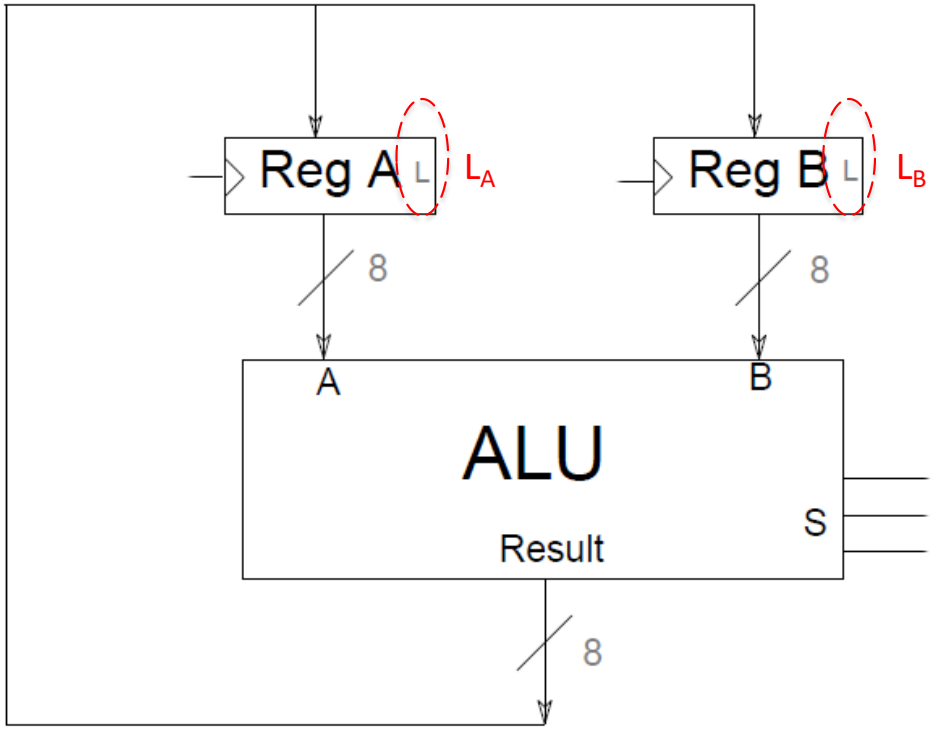
- p.ej., un registro de 4 bits:





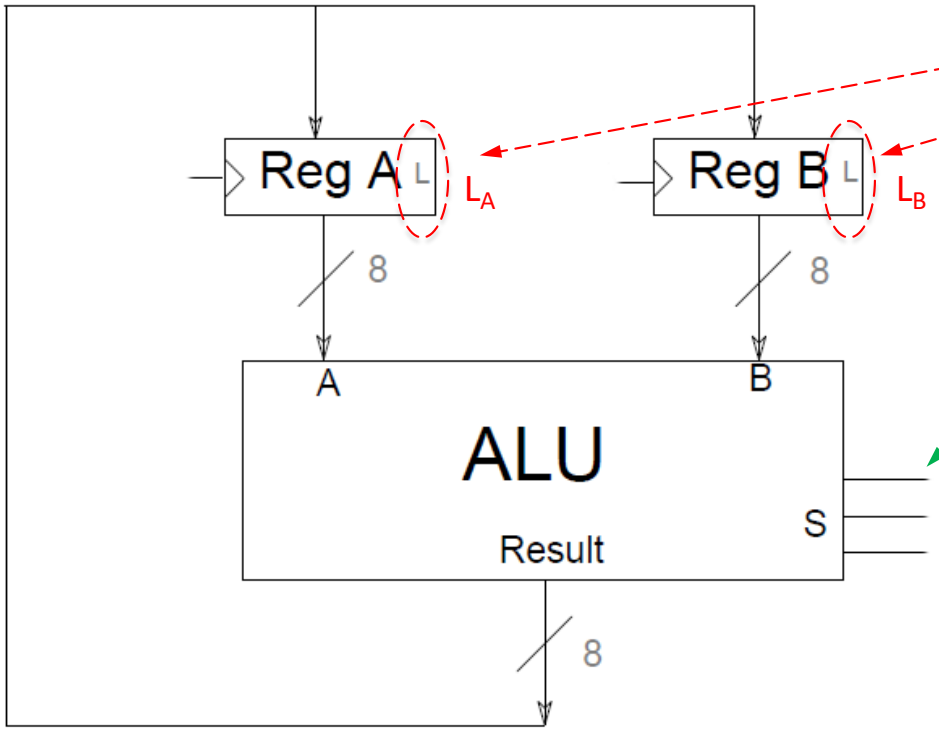
Agregamos las señales de control L_A y L_B para controlar la escritura —actualización de los valores— de los registros (*explicar*):

- podemos conectar L_A y L_B directamente al input *C* de cada registro
- … o usar L_A y L_B como uno de los inputs (enabler) de una compuerta 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 ejecuta: S_0, S_1 y S_2
- a dónde va a parar el resultado: L_A y L_B



… p.ej., las siguientes 16 operaciones

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** …

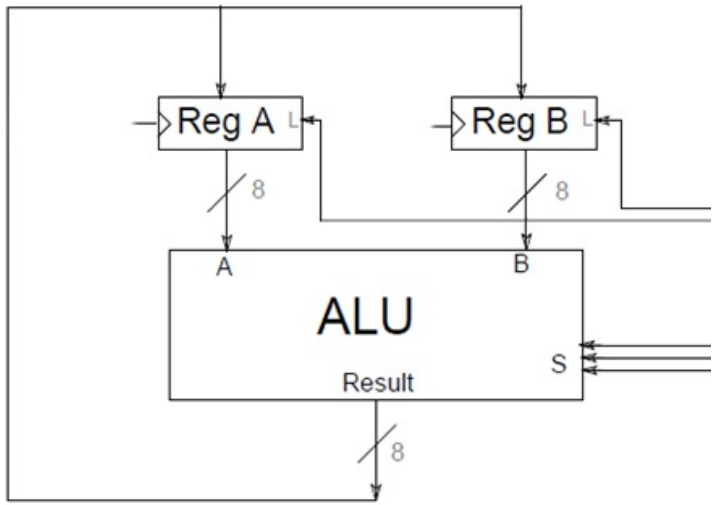
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	**0**	1
1	0	0	0	0	A=A+B	**1**	1
0	1	0	0	0	B=A+B	1	**2**
1	0	0	0	0	A=A+B	**3**	2
0	1	0	0	0	B=A+B	3	**5**
1	0	0	0	0	A=A+B	**8**	5
0	1	0	0	0	B=A+B	8	**13**

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

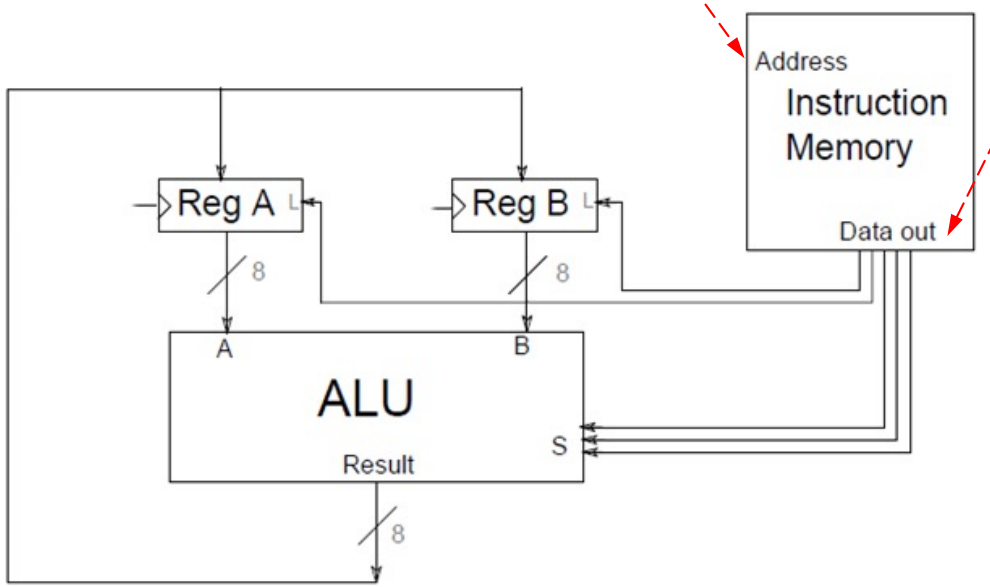
la	lb	s2	s1	s0	operación	A	B
0	0	-	-	-	-	**0**	1
1	0	0	0	0	A=A+B	**1**	1
0	1	0	0	0	B=A+B	1	**2**
1	0	0	0	0	A=A+B	**3**	2
0	1	0	0	0	B=A+B	3	**5**
1	0	0	0	0	A=A+B	**8**	5
0	1	0	0	0	B=A+B	8	**13**

¿De dónde viene el programa?

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

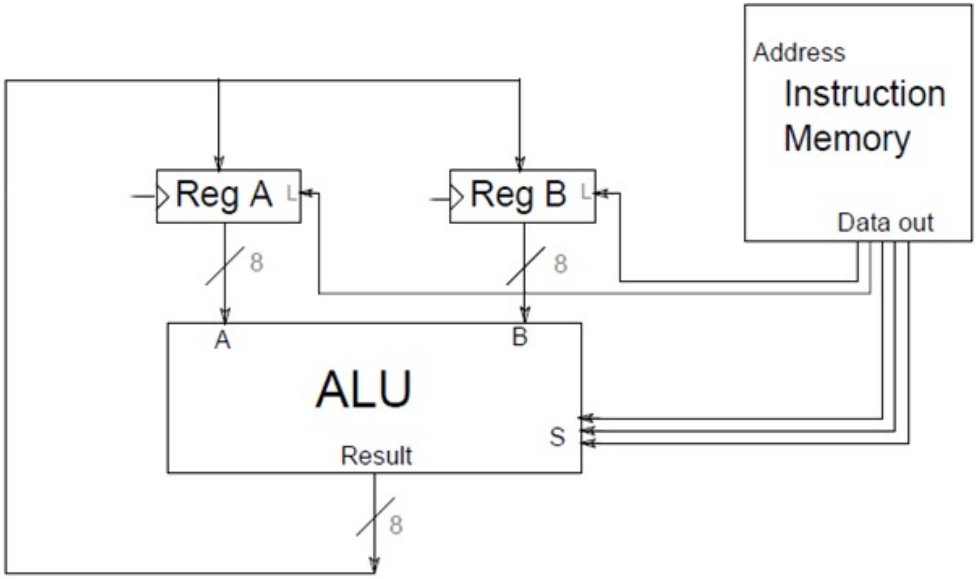


… en una **memoria** —que llamamos ***Instruction Memory***: tiene un input, ***Address*** (que ya vamos a ver), y un output, ***Data out***, que es por donde "sale" la instruccón —la combinación de señales de control— que hay que ejecutar a continuación



Internamente, la memoria está organizada como un (gran) arreglo de registros

- cada registro —o **palabra de memoria**— almacena una instrucción



instrucciones

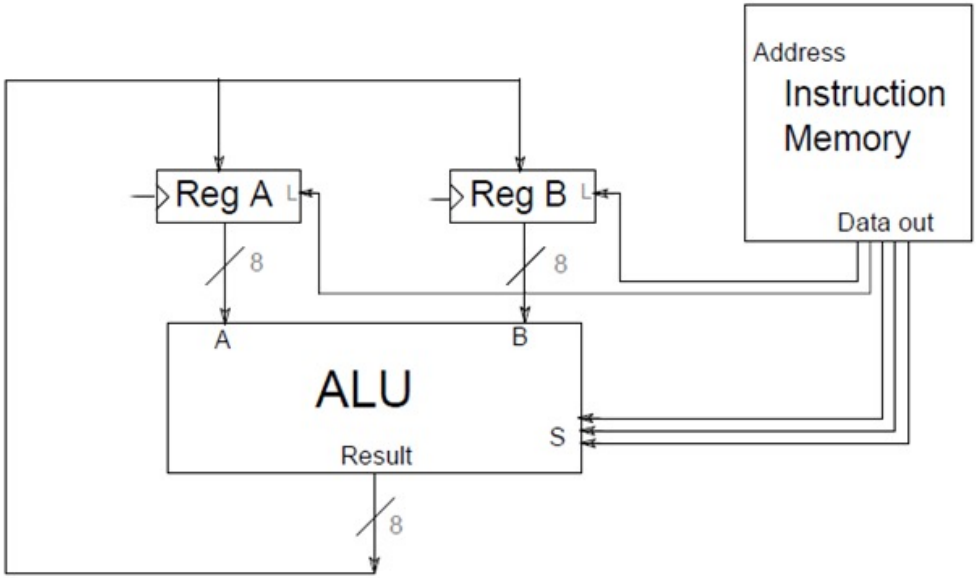
1 0 0 0 0
0 1 0 0 0
1 0 0 0 0
0 1 0 0 0
1 0 0 0 0
0 1 0 0 0

Contenido de la memoria

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 relativa que ocupa en el arreglo



instrucciones

0	10000
1	01000
2	10000
3	01000
4	10000
5	01000

Contenido de la memoria

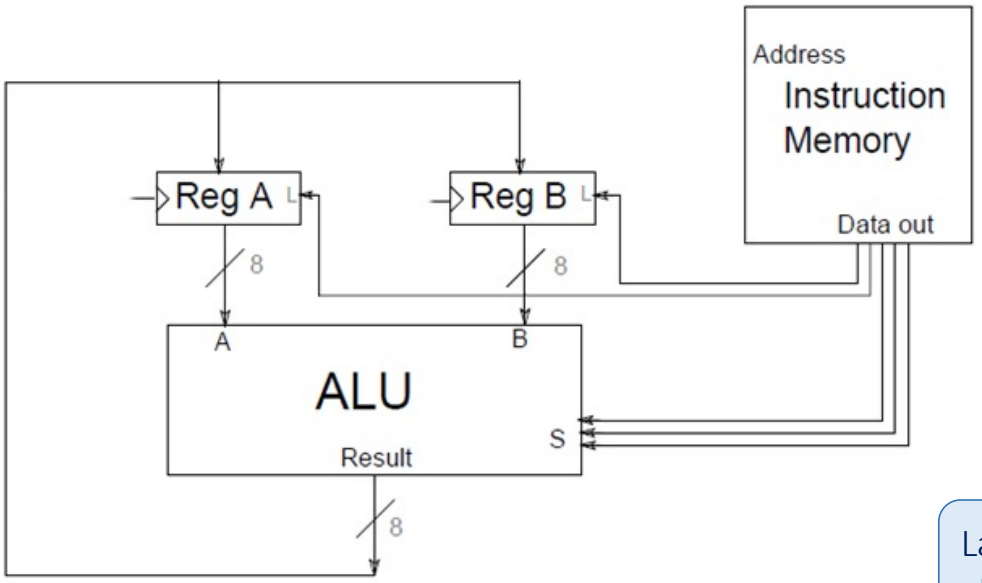
Internamente, la memoria está organizada como un (gran) arreglo de registros

- cada registro —o **palabra de memoria**— almacena una instrucción

  … y se identifica por la posición relativa que ocupa en el arreglo

  … estas posiciones relativas se llaman **direcciones** (de memoria)

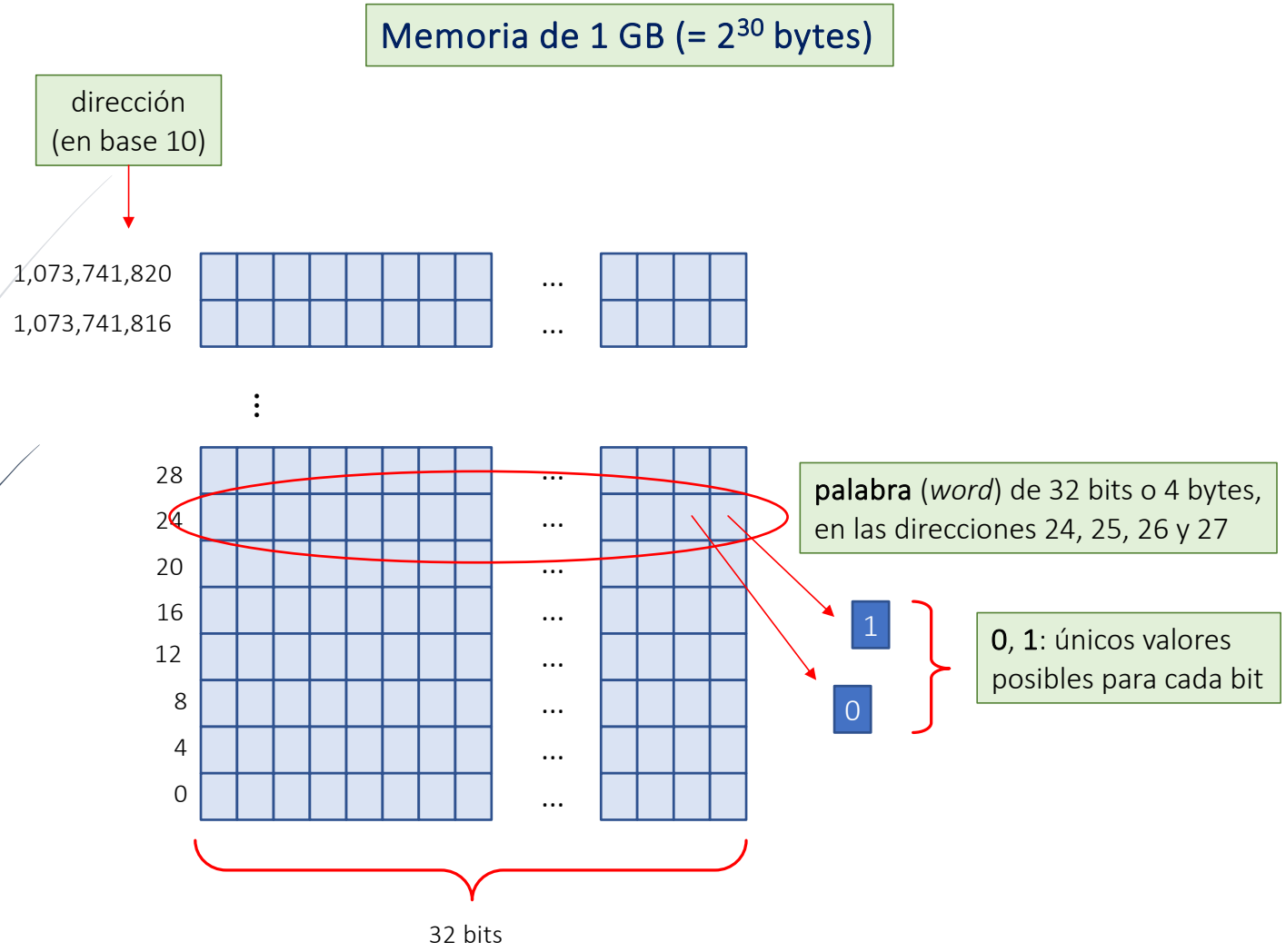
  … y se especifican en base 2 (también, en base 16 o *hexadecimal*)



direcciones	instrucciones
0000	10000
0001	01000
0010	10000
0011	01000
0100	10000
0101	01000

Contenido de la memoria

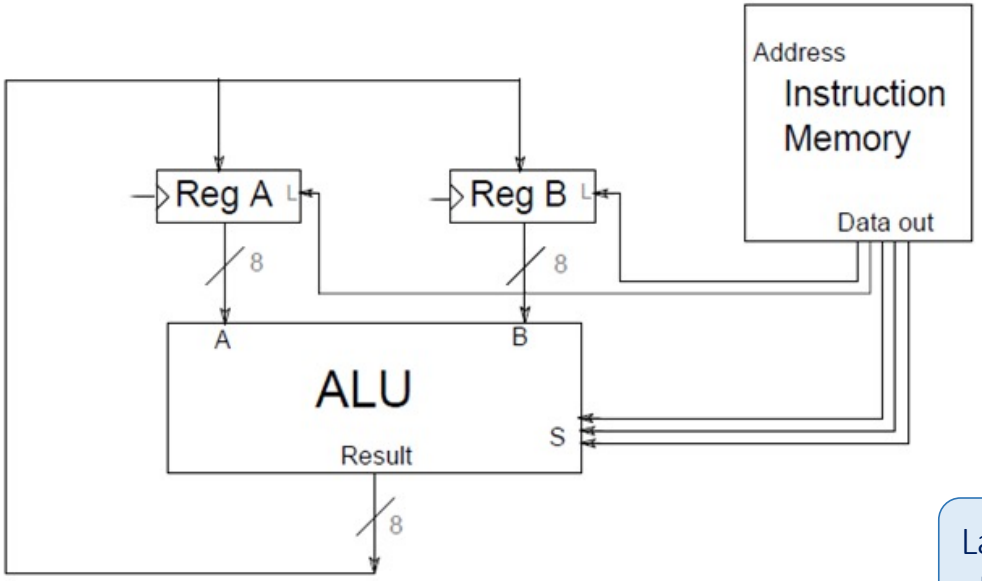
Las direcciones no están almacenadas en la memoria


Memoria de 1 GB (= 2^30 bytes)
dirección
(en base 10)
1,073,741,820
1,073,741,816
28
24
20
16
12
8
4
0
palabra (word) de 32 bits o 4 bytes,
en las direcciones 24, 25, 26 y 27
1
0
0, 1: únicos valores
posibles para cada bit
32 bits

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

… es decir, que por *Data out* las instrucciones vayan apareciendo una a 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



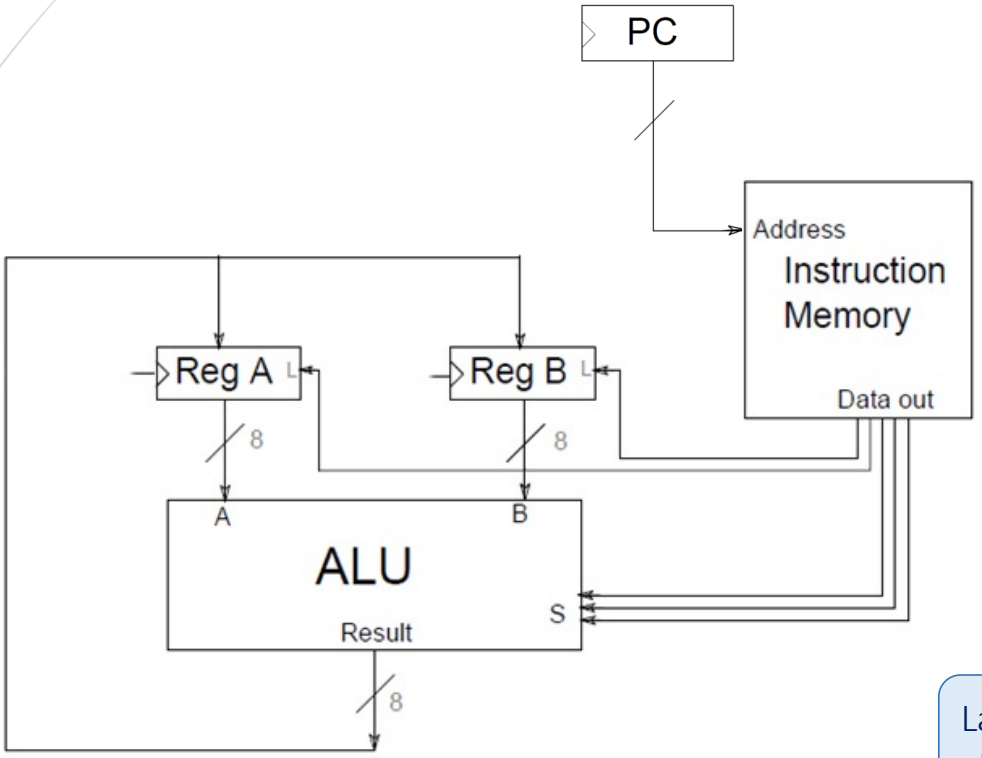
direcciones	instrucciones
0000	1 0 0 0 0
0001	0 1 0 0 0
0010	1 0 0 0 0
0011	0 1 0 0 0
0100	1 0 0 0 0
0101	0 1 0 0 0

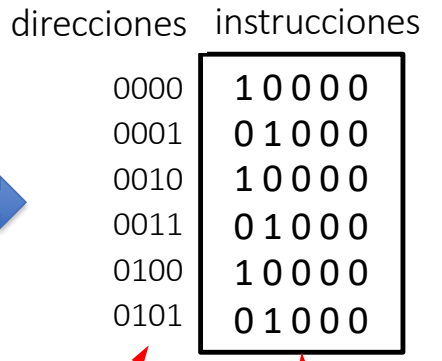
Contenido de la memoria

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*



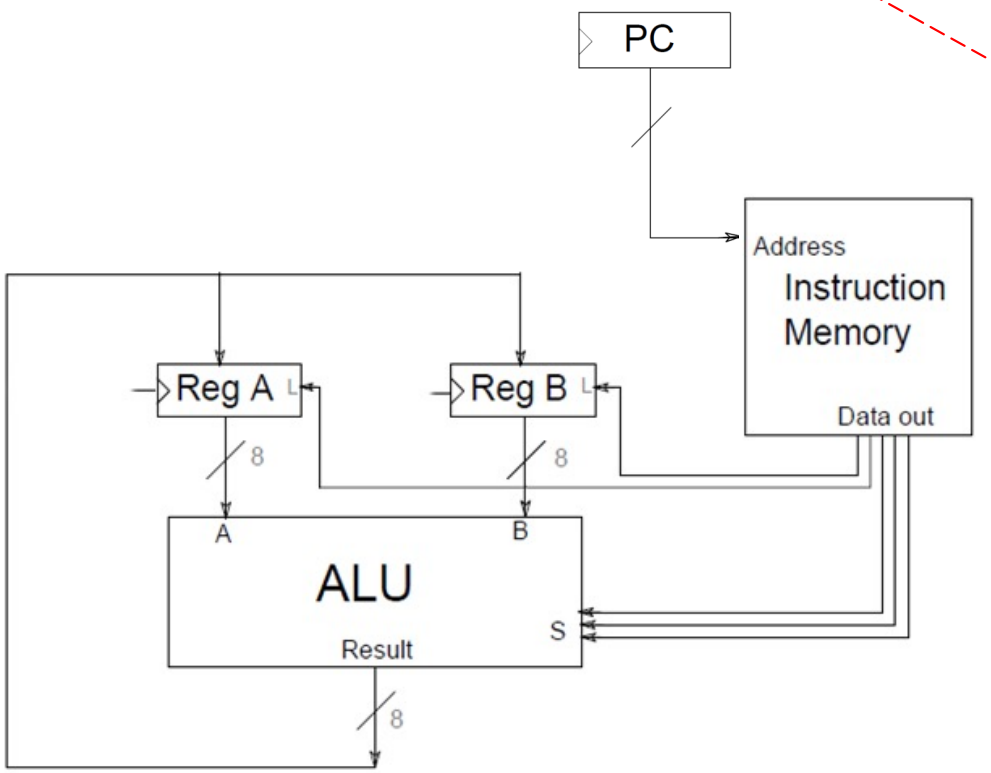


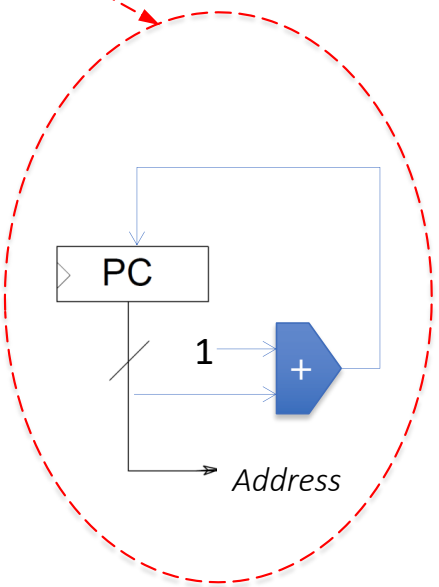
Contenido de la memoria

Las direcciones no están almacenadas en la memoria

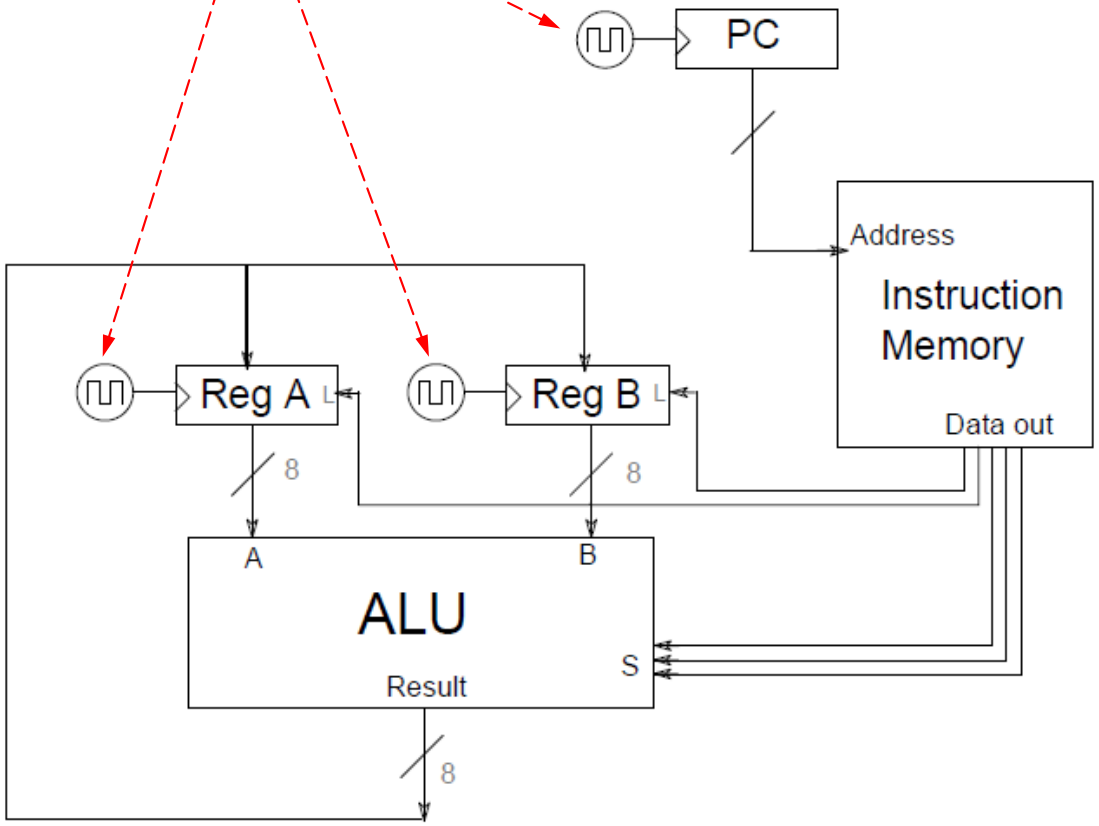
El registro ***PC*** tiene que ir incrementando automáticamente su contenido para que el programa se ejecute por completo sin más intervención nuestra que a la partida:

- requiere un circuito sumador adicional (que no lo mostramos en las demás figuras)





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



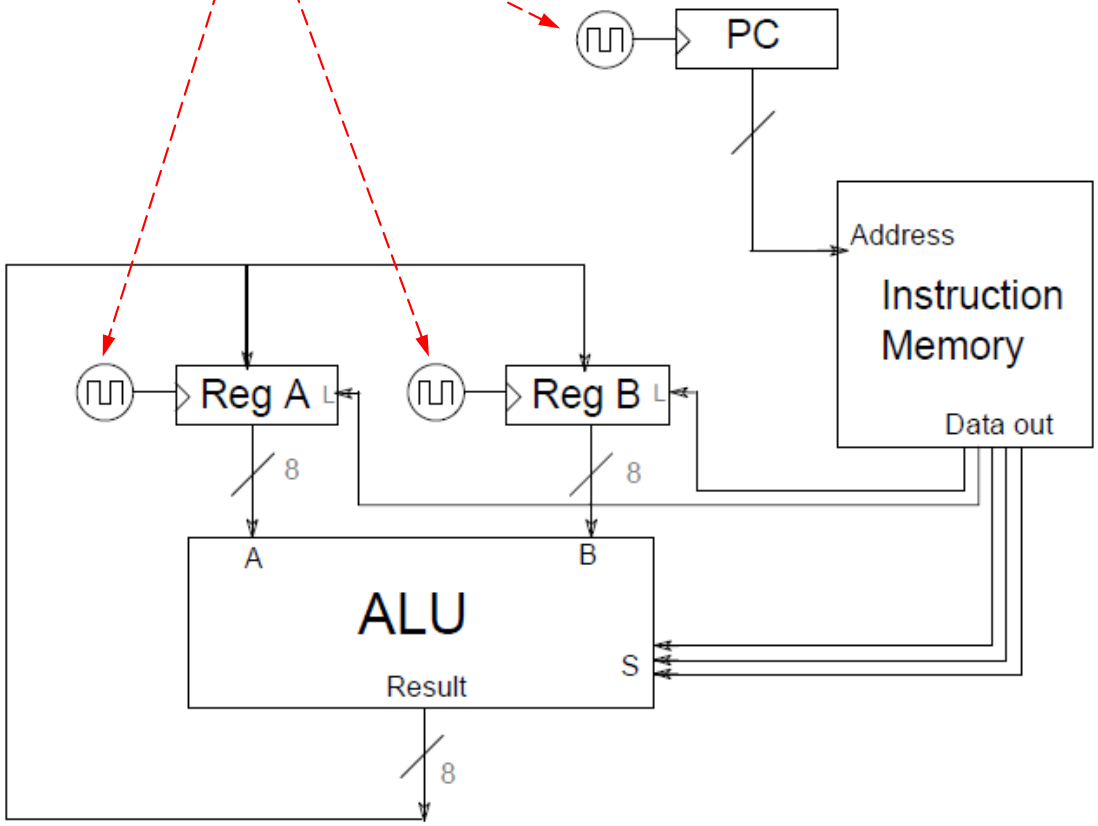
Finalmente, necesitamos que todas las acciones individuales ocurran *sincrónicamente*: incluimos un **reloj** (*clock*)

En muchos circuitos digitales, el orden en el cual pasan las cosas es crítico

Los circuitos digitales usan **relojes** para proporcionar *sincronización*

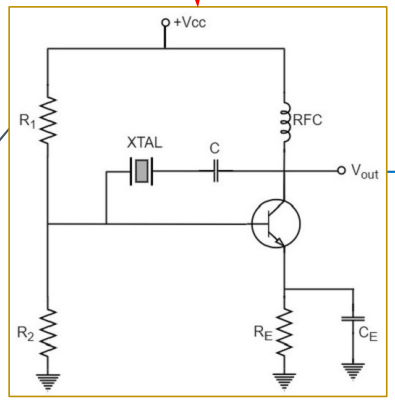
… permitir que el hardware ejecute una acción sin que sea necesario un cambio en los inputs

… y actualizar componentes que contienen *estado*

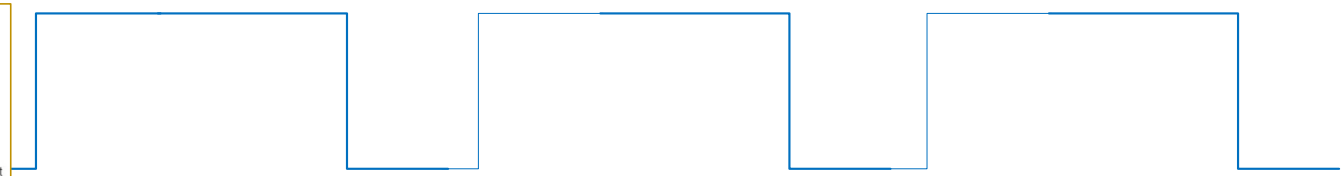


Un **reloj** —en un circuito digital— es un circuito electrónico que oscila a una tasa regular, emitiendo una serie de pulsos con dos propiedades:

- el ancho de pulso es preciso
- el intervalo entre pulsos consecutivos es preciso

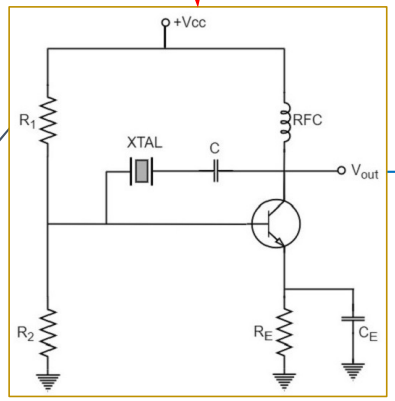


https://www.tutorialspoint.com/sinusoidal_oscillators/sinusoidal_crystal_oscillators.htm



Un **reloj** —en un circuito digital— es un circuito electrónico que oscila a una tasa regular, emitiendo una serie de pulsos con dos propiedades:

- el ancho de pulso es preciso
- el intervalo entre pulsos consecutivos es preciso



https://www.tutorialspoint.com/sinusoidal_oscillators/sinusoidal_crystal_oscillators.htm

Estado, o valor, *alto*

Flanco de bajada

Tiempo de ciclo o *período*
El inverso es la *frecuencia*, p.ej., desde 100 MHz hasta 4 GHz

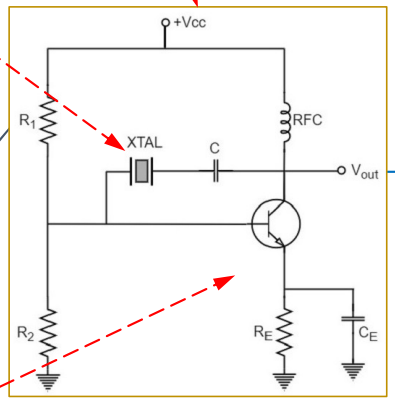
Flanco de subida

Estado, o valor, *bajo*

Un **reloj** —en un circuito digital— es un circuito electrónico que oscila a una tasa regular, emitiendo una serie de pulsos con dos propiedades:

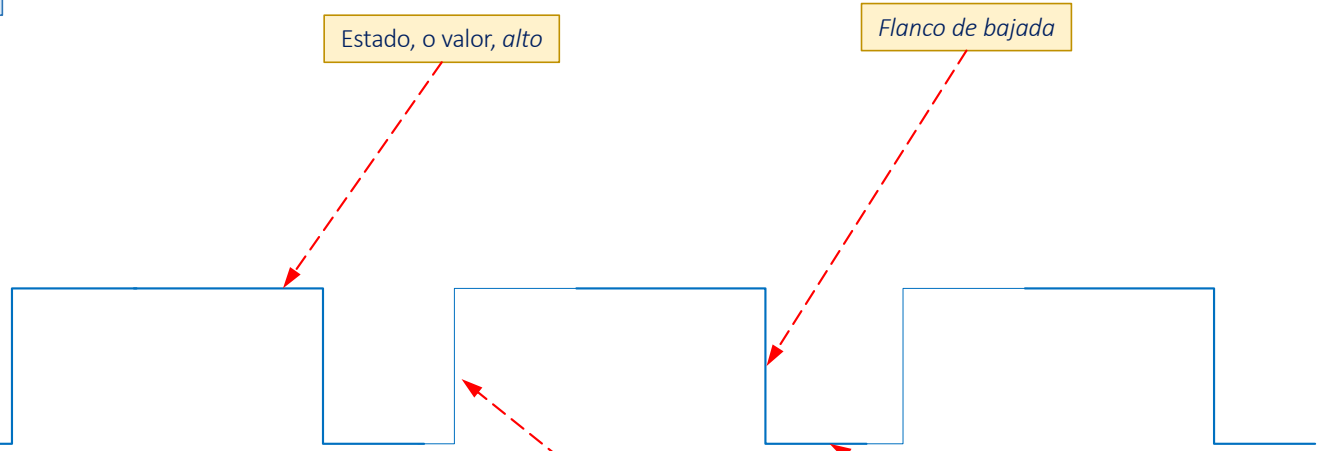
- el ancho de pulso es preciso
- el intervalo entre pulsos consecutivos es preciso

Cristal de cuarzo, que oscila naturalmente a una frecuencia precisa



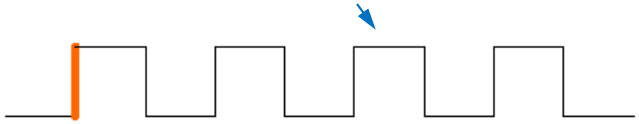
El circuito amplifica la señal y la cambia de una onda sinusoidal a una onda cuadrada

https://www.tutorialspoint.com/sinusoidal_oscillators/sinusoidal_crystal_oscillators.htm



Los relojes son simétricos: el tiempo en el estado alto es igual al tiempo en el estado bajo. Para generar un tren de pulsos asimétricos, desplazamos el reloj básico usando un circuito de retardo y hacemos un AND entre esta señal y la original

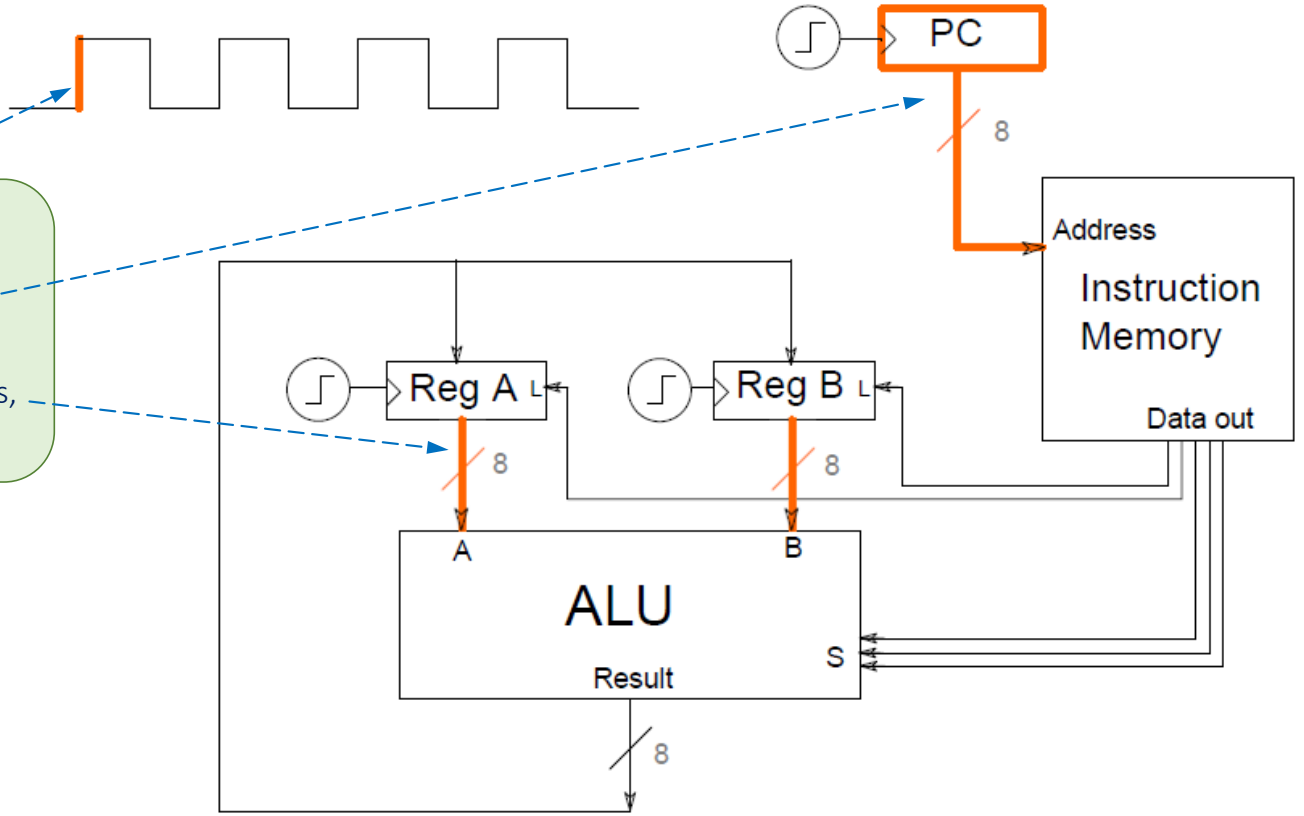
Reloj: Circuito que emite una ***serie de pulsos*** con un ancho preciso y un intervalo preciso entre pulsos consecutivos, y cuya frecuencia es controlada por un oscilador de cristal



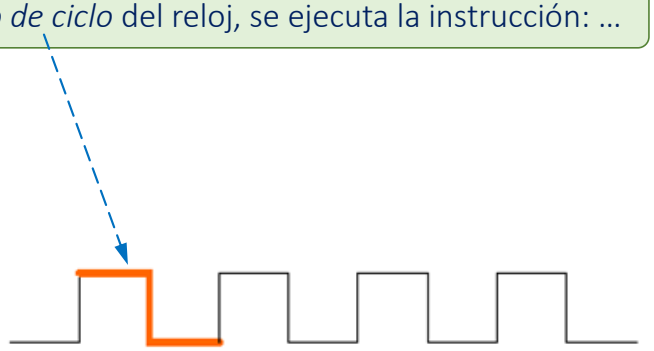
Cuando el pulso sube (*flanco de subida*)

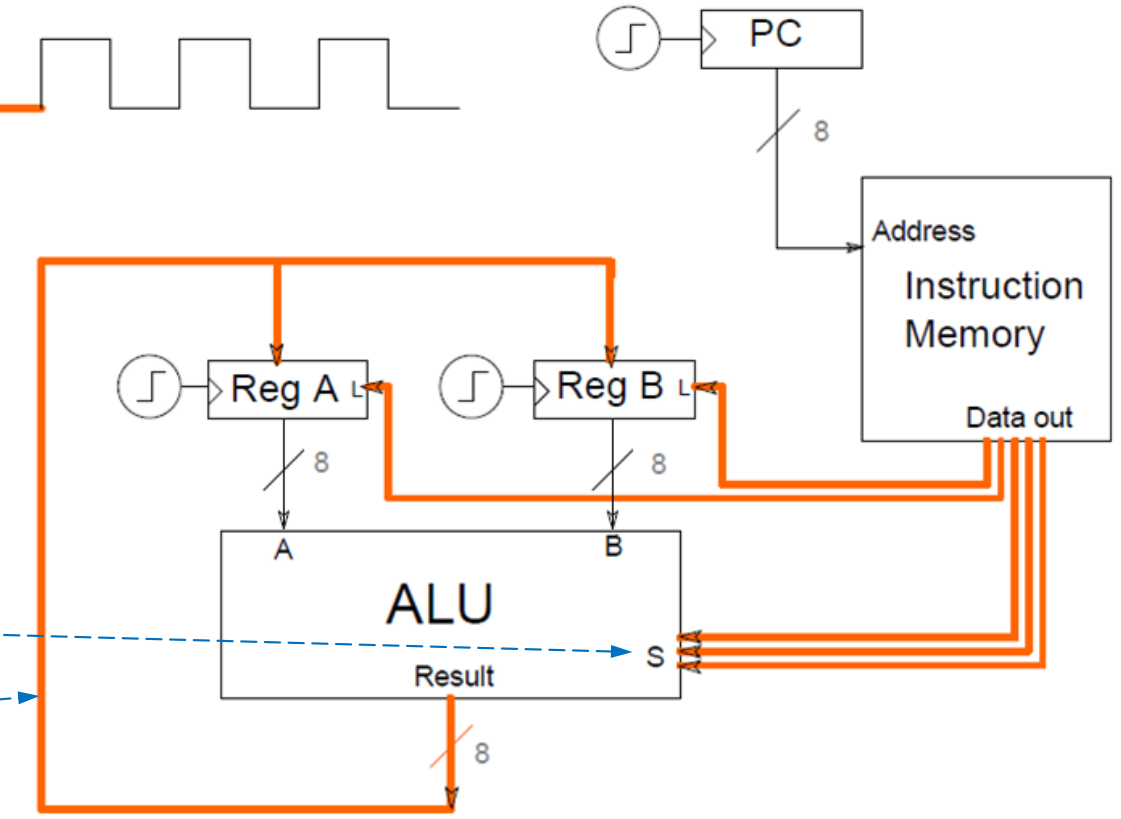
… el ***PC*** actualiza su valor, que va a la entrada *Address* de la memoria

… y los registros *A* y *B* actualizan sus valores, que van a las entradas *A* y *B* de la ALU



Durante el *tiempo de ciclo* del reloj, se ejecuta la instrucción: …

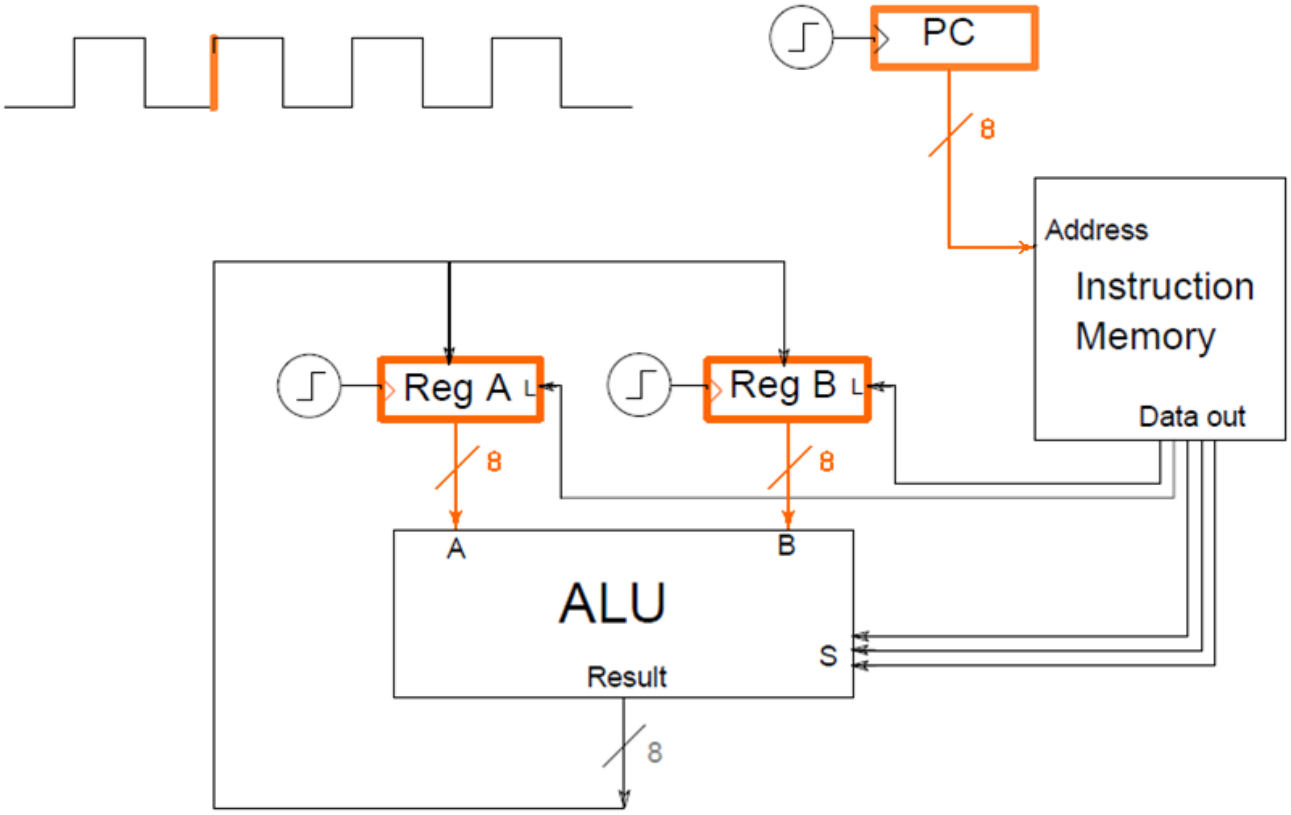




… la ALU realiza la operación especificada por sus tres señales de control

… y el resultado se pone a la entrada de los registros *A* y *B*

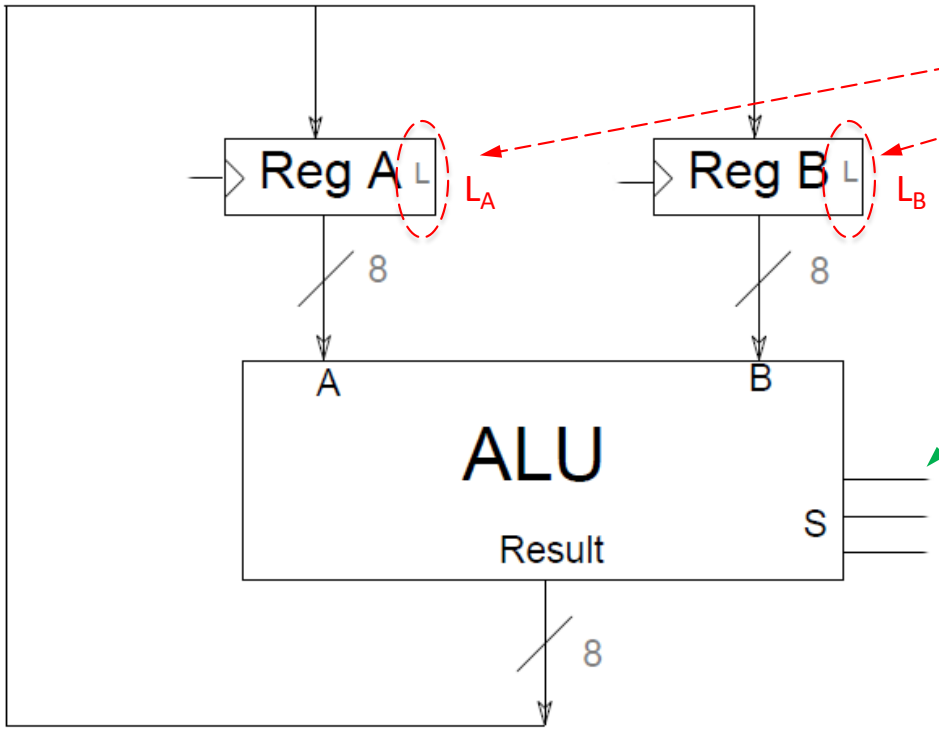
En el siguiente flanco de subida del reloj, ***PC*** pone una nueva dirección (la anterior más 1) en la entrada Address de la memoria, y los registros *A* y *B* ponen (posiblemente) nuevos valores en las entradas de la ALU



Nuestra primera versión de un computador *muy* básico:

- una ALU con 8 operaciones
- dos registros

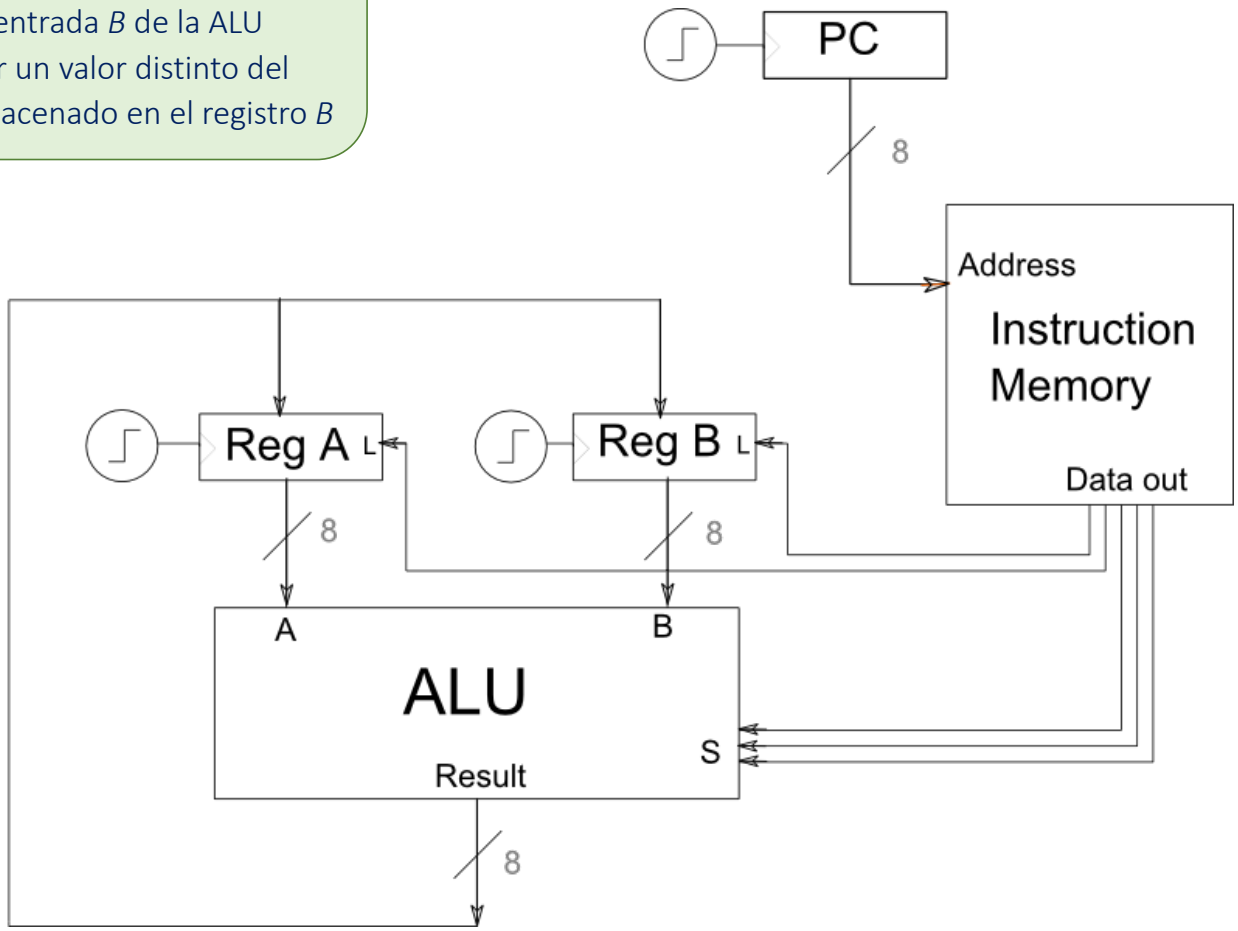
... y su lenguaje de máquina de 16 instrucciones



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

¿Cómo podemos independizarnos de los valores 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*

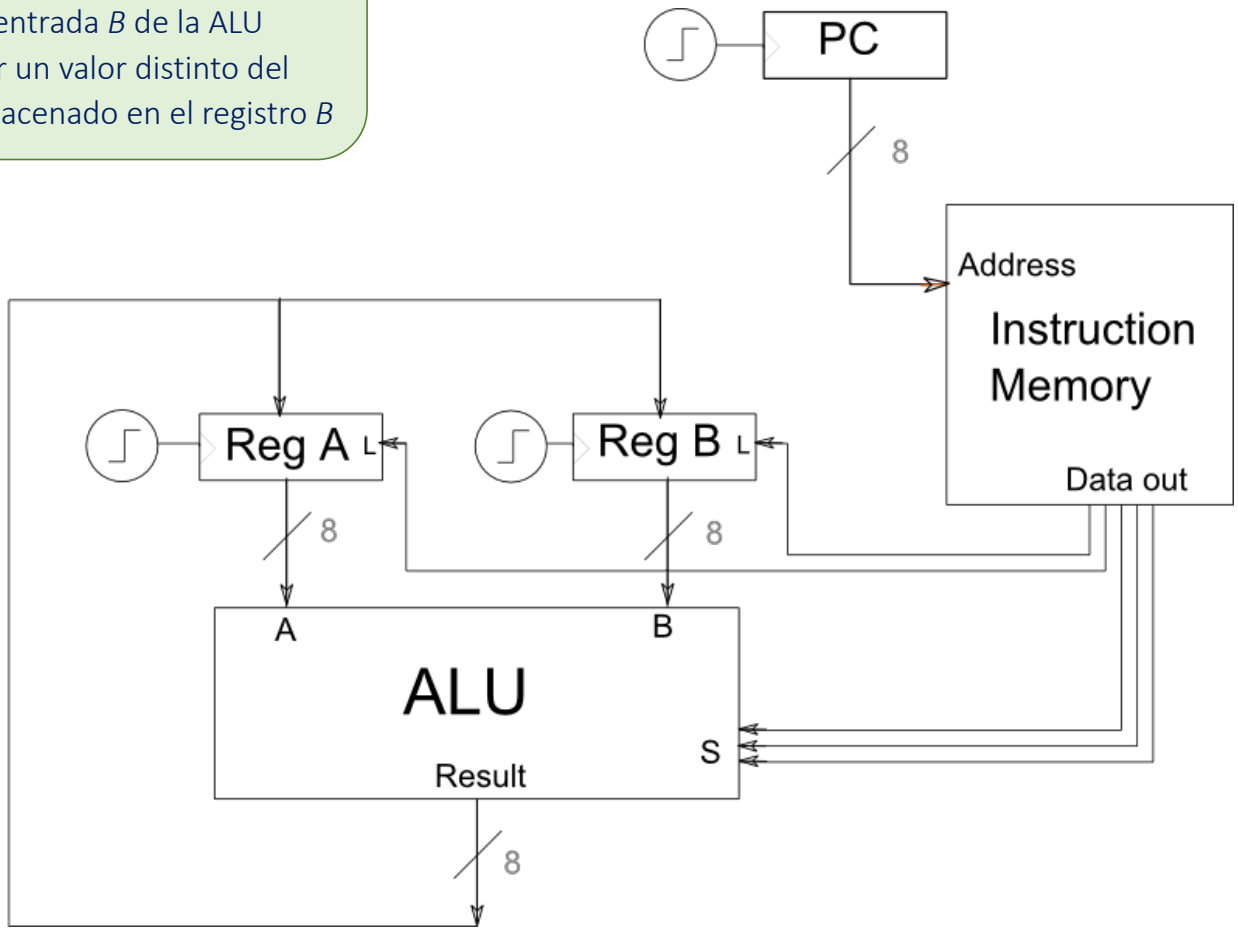


¿Cómo podemos independizarnos de los valores en los registros?

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

Hasta ahora, para ejecutar **A=A−B** escribimos la instrucción de 5 bits **1 0 0 0 1** (diap. anterior)

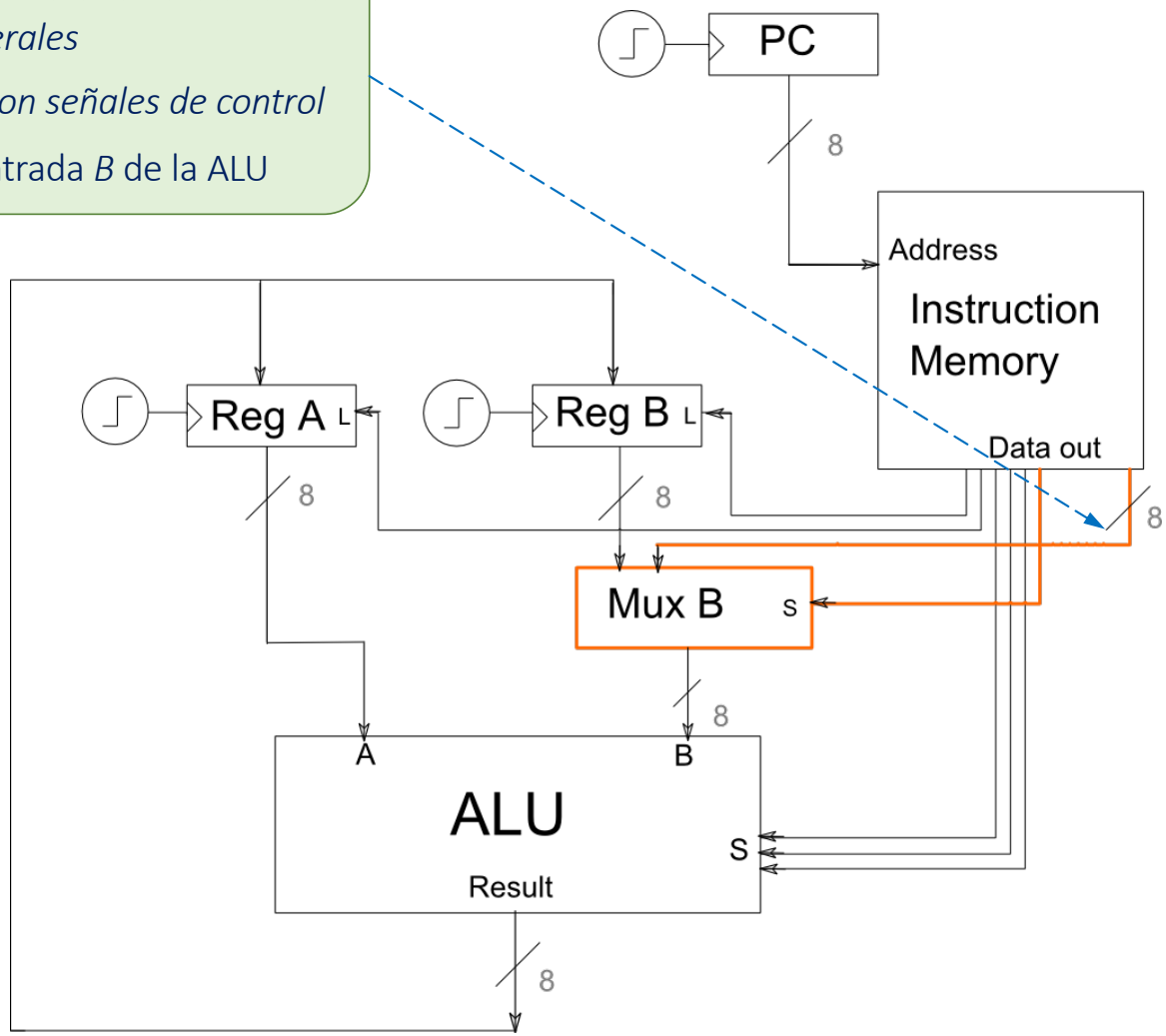
... pero, p.ej., no tenemos cómo ejecutar **A=A+6**



Agregamos a las instrucciones **8 bits adicionales** para representar valores *literales*

Estos 8 bits adicionales *no son señales de control*

… **son datos** que van a la entrada *B* de la ALU



Agregamos a las instrucciones **8 bits adicionales** para representar valores *literales*

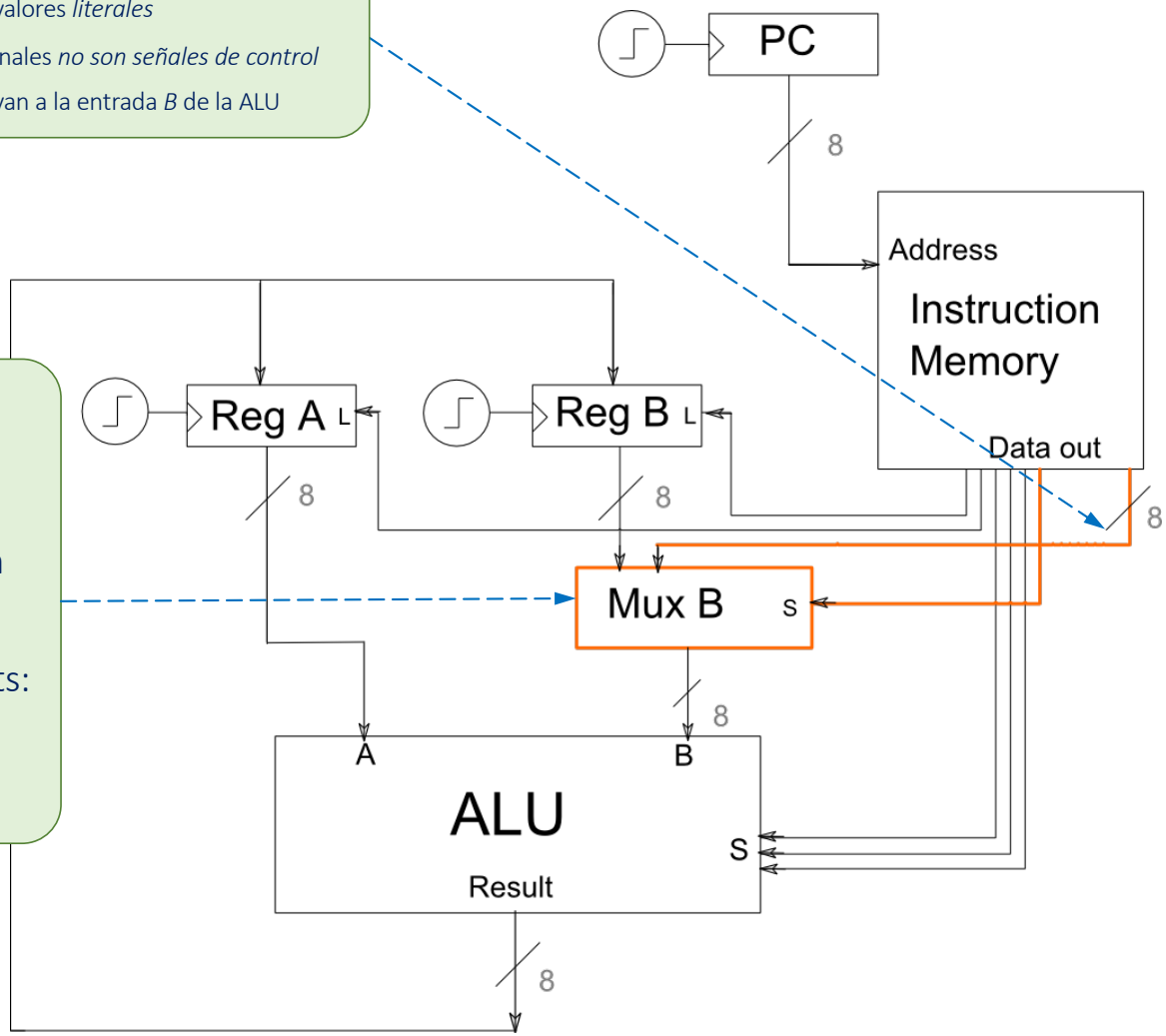
Estos 8 bits adicionales *no son señales de control*

… **son datos** que van a la entrada *B* de la ALU

Agregamos un **multiplexor *B*** 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

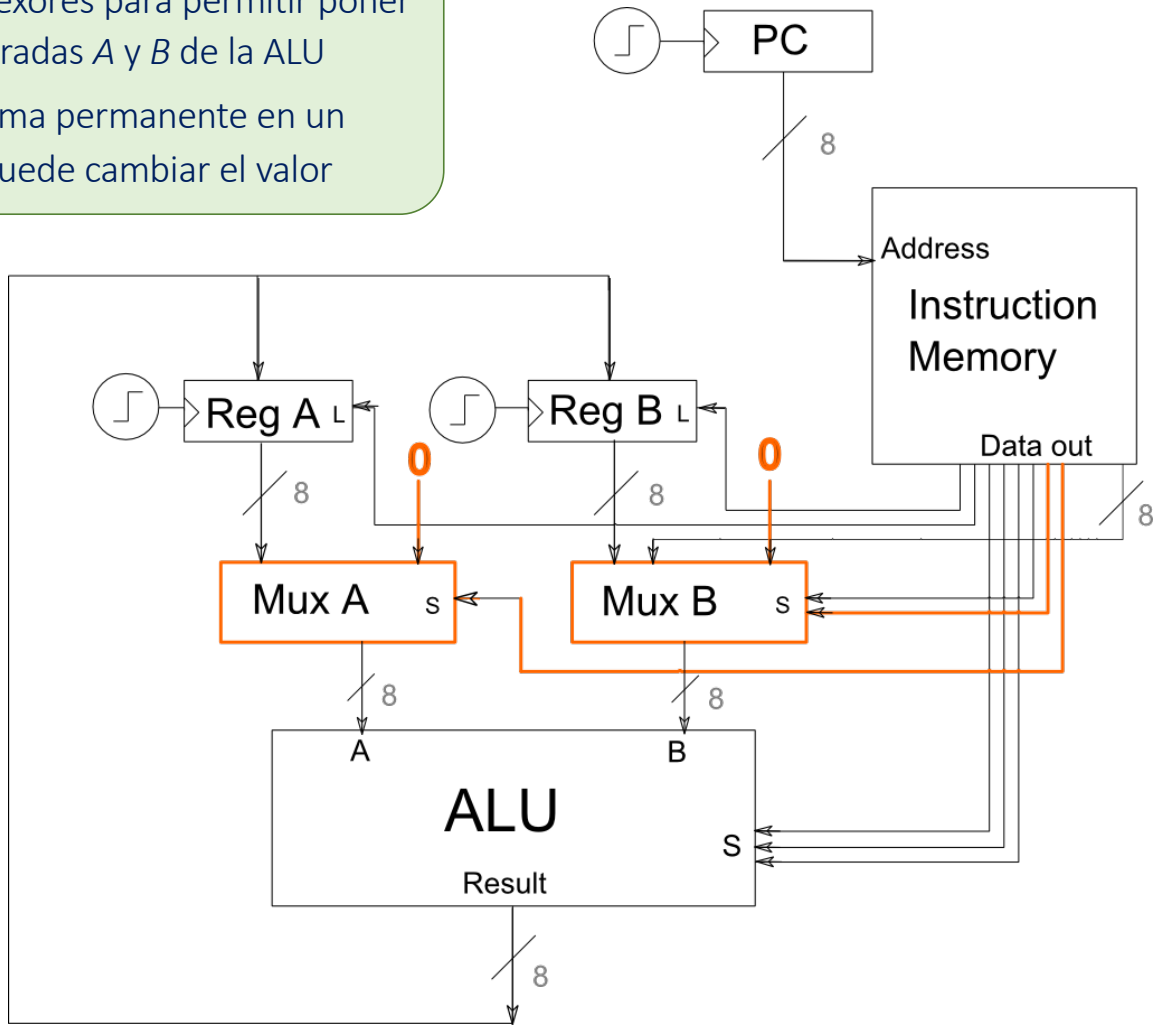
Mux B debe ser controlado por una nueva *señal de control* ⟹ ahora cada instrucción tiene 14 bits:

- 6 bits que corresponden a las señales de control
  ⋯ + 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 valor **0** está almacenado de forma permanente en un registro especial al que no se le puede cambiar el valor

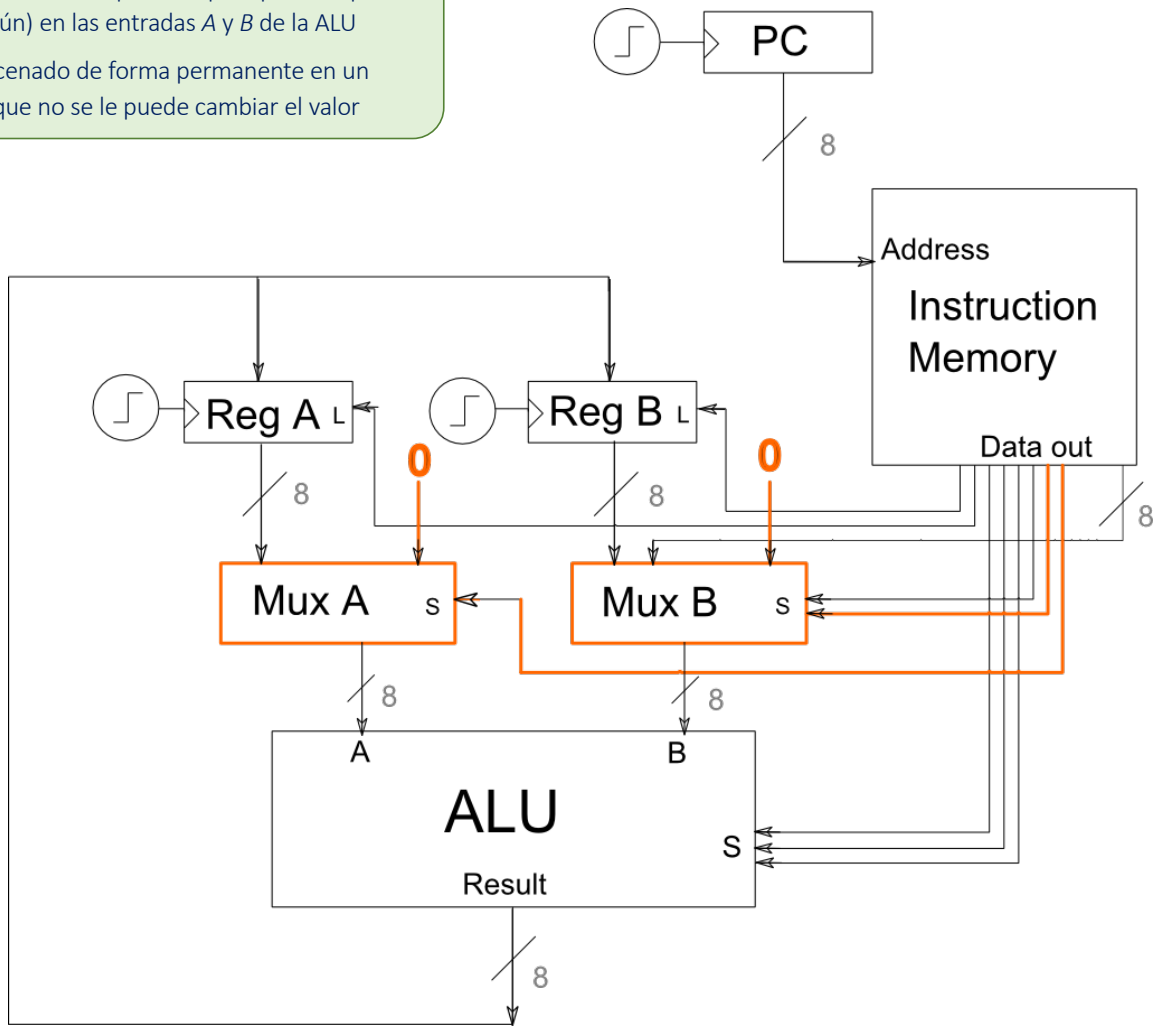


Extendemos el uso de los multiplexores para permitir poner el valor **0** (muy común) en las entradas *A* y *B* de la ALU

El valor **0** está almacenado de forma permanente en un registro especial al que no se le puede cambiar el valor

El nuevo *Mux A* y la posibilidad del *Mux B* de elegir ahora entre tres inputs, exigen dos señales de control adicionales

⟹ **instrucciones de 16 bits**



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

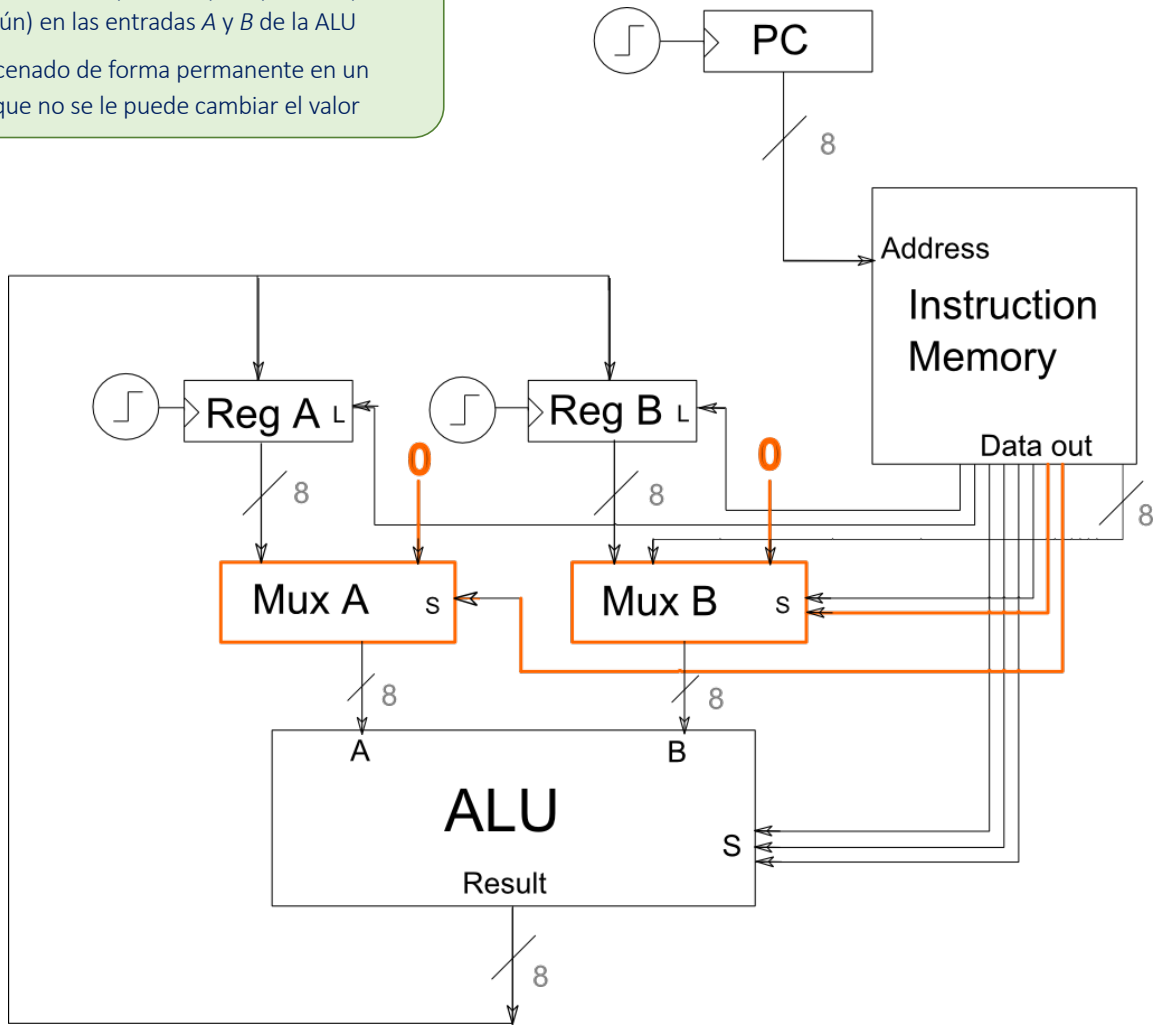
El valor **0** está almacenado de forma permanente en un registro especial al que no se le puede cambiar el valor

El nuevo *Mux A* y la posibilidad del *Mux B* de elegir ahora entre tres inputs, exigen dos señales de control adicionales
⟹ **instrucciones de 16 bits**

Ahora sí podemos ejecutar **A=A+6** ⟹ escribimos la instrucción de 16 bits

1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 (próx. diap.):

- los 8 bits de la izquierda corresponden a las señales de control
- los 8 bits de la derecha corresponden al literal, en este caso el número 6 (en binario)



Instrucciones de 16 bits de largo

⟹ 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*

Señal de control del *Mux A*

Señales de control del *Mux B*

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