Lógica digital

Arquitectura de Computadores – IIC2343

2021-1 Yadran Eterovic

Primero, lo más básico acerca de la representación de datos en un computador

… en particular, la **representación de números** (enteros)

Cuando se trata de números, las personas pensamos en base 10 —*números decimales*:

- diez símbolos diferentes —0, 1, 2, 3, 4, 5, 6, 7, 8, 9— llamados dígitos decimales, o simplemente, dígitos

… y los representamos en lo que llamamos la *notación posicional*:

- el verdadero valor de un dígito en un número de varios dígitos depende no sólo del dígito mismo

  … sino también de su posición dentro del número

P.ej., cuando escribimos 421, en realidad estamos hablando del número (o valor numérico) que es el resultado de la siguiente operación:

$$4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Pero los números pueden ser representados en cualquier base, usando la misma notación posicional

… en particular, en **base 2** (*números binarios*):

- sólo dos símbolos diferentes —0, 1— llamados *dígitos binarios* o ***bits***

P.ej., el número 421 en base 2 se representa así:

$$1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1$$

… ya que (ver diap. #6):

$$1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4$$
$$+ 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 421$$

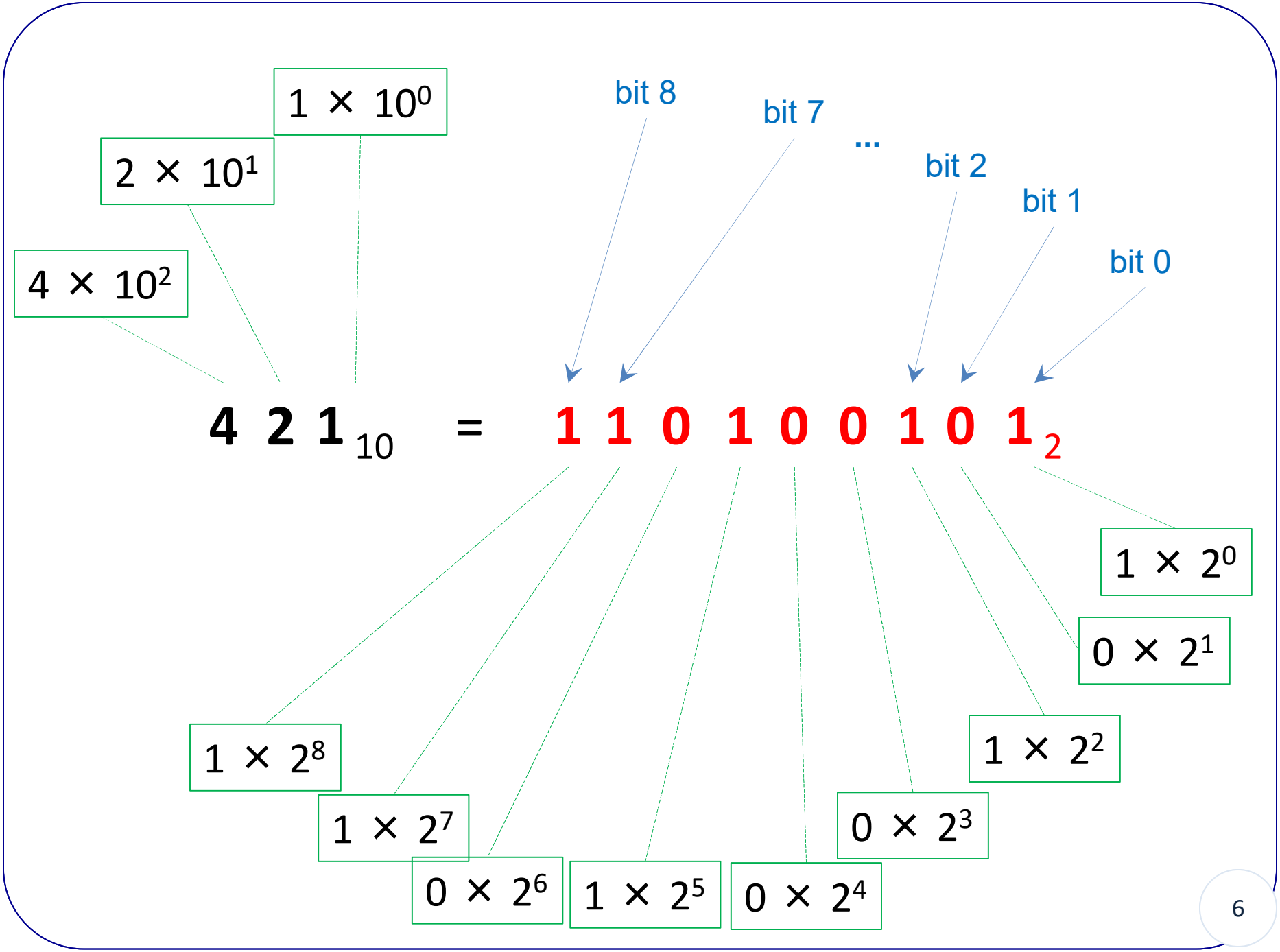
Los bits son los “átomos” de la computación:

- toda información en formato “computacional” se compone sólo de bits

Para poder referirnos a ellos de una manera precisa, numeramos los bits de un número binario —el bit 0, el bit 1, el bit 2, el bit 3, …— de derecha a izquierda:

- es decir, desde el bit menos significativo —el que va multiplicado por 2^0—

  … al bit más significativo —el que va multiplicado por 2^{n-1}, si el número de bits es n


1 × 10^0
2 × 10^1
4 × 10^2
bit 8
bit 7
...
bit 2
bit 1
bit 0
4 2 1 10 = 1 1 0 1 0 0 1 0 1 2
1 × 2^0
0 × 2^1
1 × 2^2
0 × 2^3
1 × 2^5
0 × 2^4
0 × 2^6
1 × 2^7
1 × 2^8

Tanto en base 10 como en base 2, un número (un valor numérico) tiene una única representación

… → si cambiamos cualquier dígito de 421, el nuevo número va a tener un valor numérico distinto de 421

… y lo mismo con 1 1 0 1 0 0 1 0 1: si cambiamos cualquiera de los 1's por 0's y/o cualquiera de los 0's por 1's, el valor numérico del número binario resultante va a ser distinto de 421

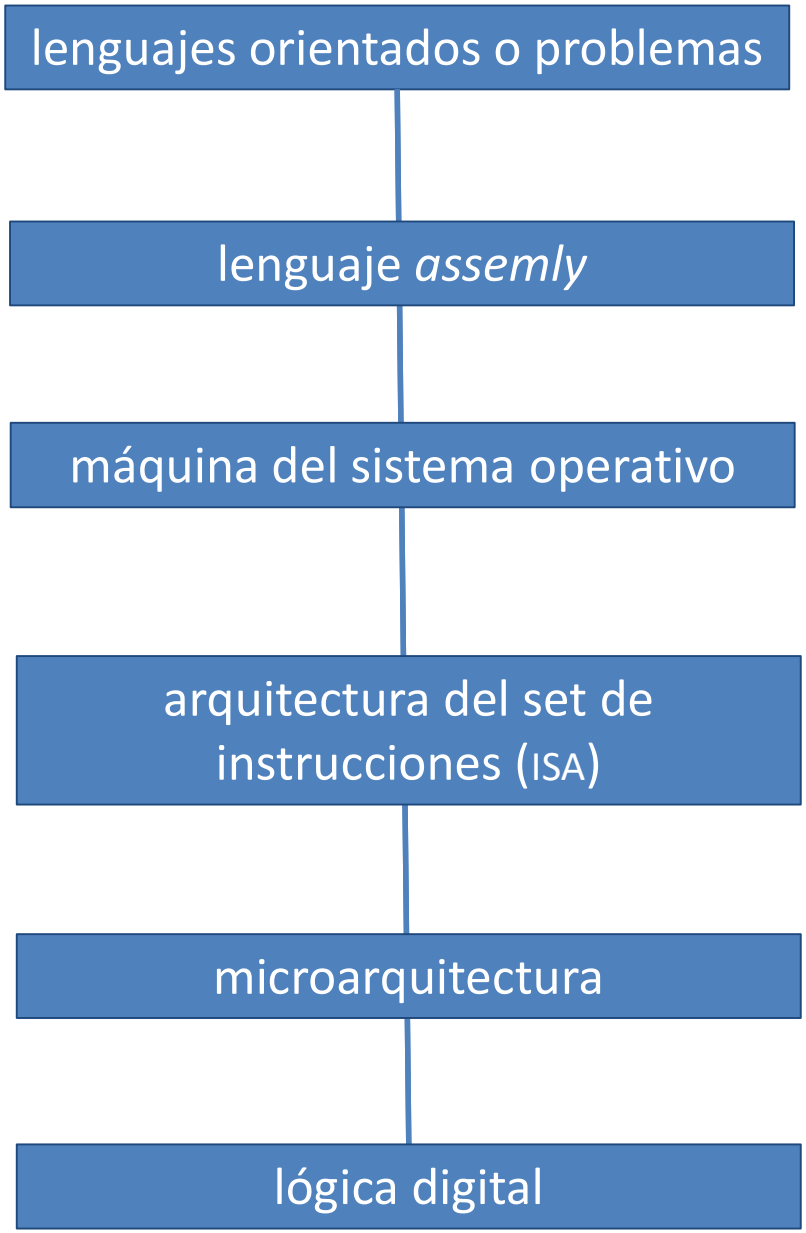
Lo que acabamos de ver es lo que podríamos llamar la representación conceptual

Pero, ¿cómo se hace en términos físicos?

… lo que llamamos el *hardware* del computador

La ***lógica digital*** es el verdadero hardware del computador

Conceptualmente, está en la frontera entre la ciencia de la computación y la ingeniería electrónica



5) lenguajes de alto nivel, traducidos por compiladores, o interpretados

4) forma simbólica de los lenguajes de más abajo, traducida por el assembler

3) instrucciones adicionales, otra organización de memoria, capacidad de ejecución concurrente → interpretado por un programa llamado el *sistema operativo*

2) instrucciones ejecutadas por el micro-programa o circuitos del hardware

1) registros + ALU → *datapath*, controlada por hardware o microprogramada

0) compuertas (*gates*), álgebra de Boole, registros, el motor de computación

Todos los (circuitos digitales de los) computadores modernos se construyen físicamente a partir de unos pocos elementos básicos muy simples, combinándolos de innumerables formas

En un circuito digital hay solo dos valores lógicos —**lógica binaria:**

- una señal eléctrica de entre 0 y 0.5 volts representa el valor binario 0
- una señal eléctrica de entre 1 y 1.5 volts representa el valor binario 1

Las **compuertas** —dispositivos electrónicos pequeños, hechos de *transistores*— pueden calcular varias funciones a partir de estas señales

… y son la base de hardware sobre la cual se construyen todos los computadores digitales

Un **transistor** es un dispositivo semiconductor que puede usarse como amplificador y como interruptor de señales electrónicas (y de potencia eléctrica):

- inventado en la práctica en 1947 (el concepto es de unos veinte años antes)
- **W. Shockley, J. Bardeen y W. Brattain ganaron el Premio Nobel de Física en 1956**

Toda la lógica digital moderna se basa en que un transistor puede funcionar como un interruptor binario muy rápido

[Aquí, y durante unas pocas diapositivas, vamos a describir breve y parcialmente el nivel que está bajo el nivel 0: los transistores]

(El transistor MOSFET —lejos, el más usado (99.9%)— fue inventado en 1959 por M. Atalla y D. Kahng:

- *metal-oxide-semiconductor field-effect transistor*
- compacto, podía miniaturizarse y ser producido en masa
- consume poquísima electricidad
- se calcula que hasta la fecha se han fabricado más de 13 *sextillions* (1.3×10^{22}) MOSFETs → el dispositivo más fabricado en toda la historia

)

Un transistor tiene tres terminales o conectores hacia el resto del circuito:

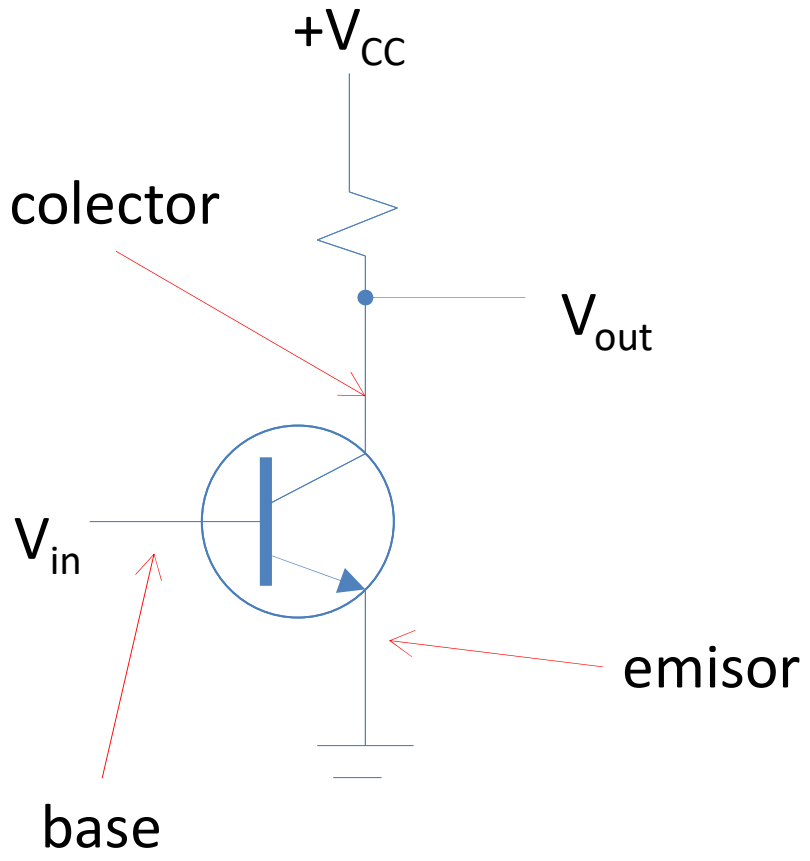
- *colector*, *base* y *emisor*

Cuando el voltaje de entrada, V_{in} , es menor que un cierto valor crítico, el transistor se apaga y actúa como una resistencia infinita:

- el voltaje de salida, V_{out} , es igual a V_{cc} , un voltaje regulado externamente (p.ej., 1.5 volts)

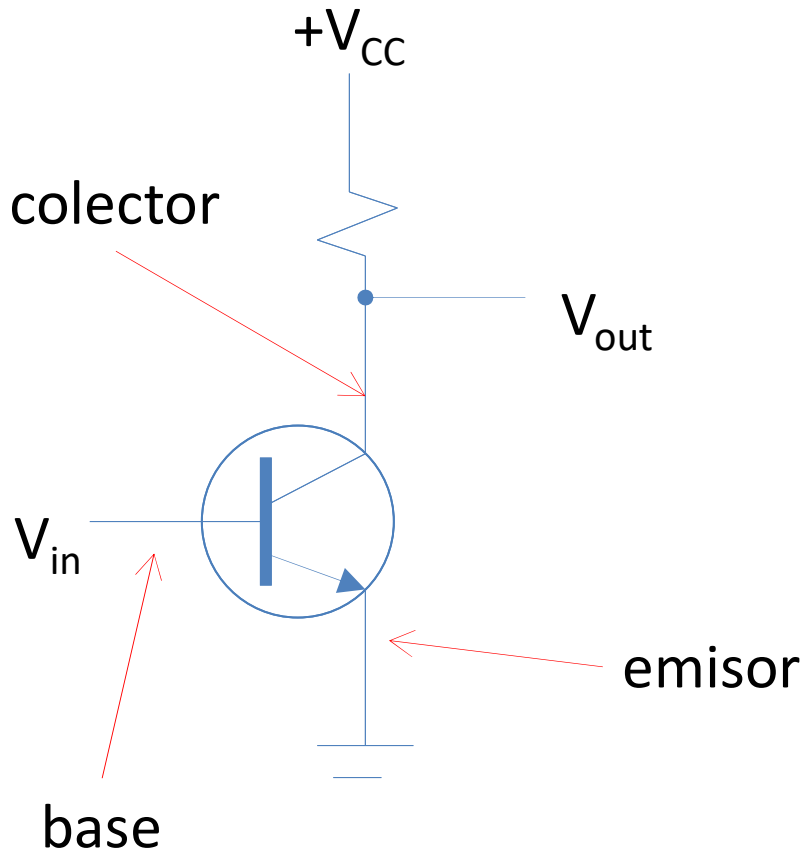
Cuando V_{in} excede el valor crítico, el transistor se enciende y actúa como un alambre:

- V_{out} se hace 0



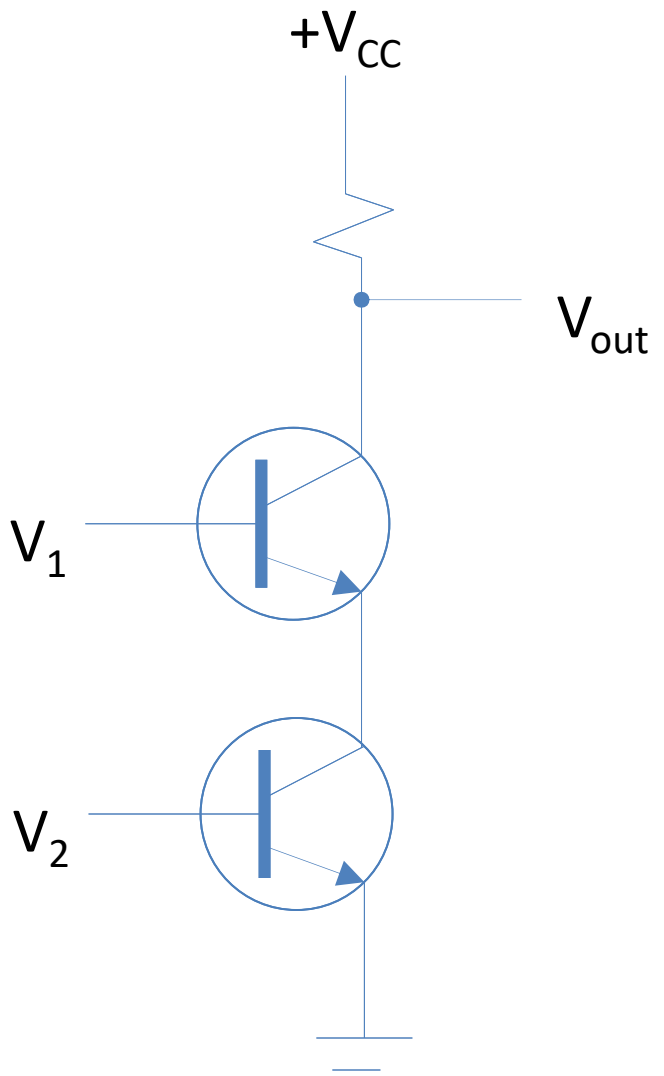
Así, un transistor por sí solo actúa como un ***inversor***:

- si V_{in} es 0, entonces V_{out} es 1, y viceversa
- no es instantáneo, pero el tiempo que toma cambiar de un estado al otro es un nanosegundo o menos
- (corresponde, como ya vamos a ver, a la compuerta lógica **NOT**)



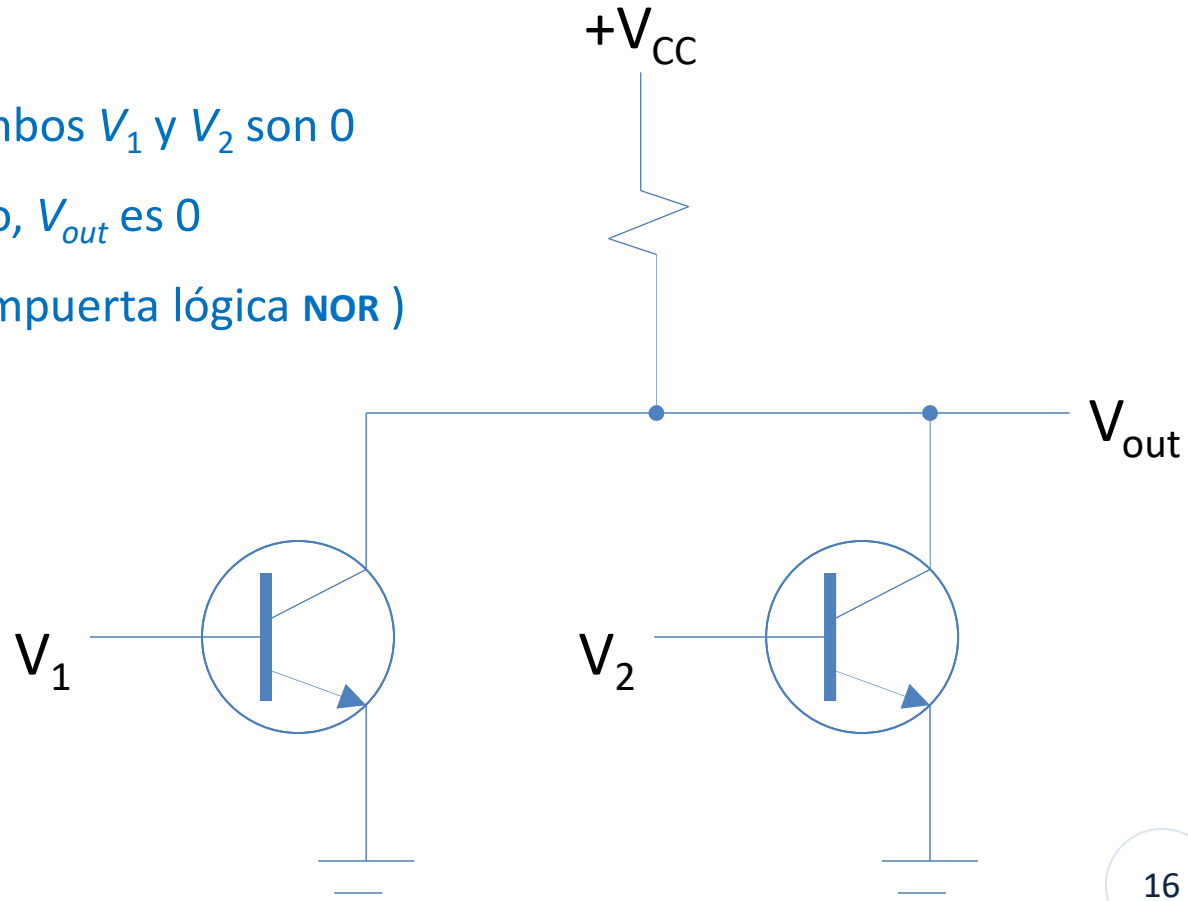
Dos transistores conectados *en serie* (el emisor del primero es conectado al colector del segundo) forman un circuito con la siguiente propiedad:

- basta que uno de los transistores actúe como una resistencia infinita para que el par de transistores sea una resis-tencia infinita
- V_{out} es 0 si y sólo si ambos voltajes de entrada V_1 y V_2 son 1
- en cualquier otro caso, V_{out} es 1
- (corresponde a la compuerta lógica **NAND**)



Dos transistores conectados en paralelo (los colectores están conectados entre ellos) forman un circuito con la siguiente propiedad:

- basta que uno de los transistores actúe como un alambre para que el par de transistores sea un alambre
- V_{out} es 1 si y sólo si ambos V_1 y V_2 son 0
- en cualquier otro caso, V_{out} es 0
- (corresponde a la compuerta lógica **NOR**)



Los tres circuitos anteriores (diapos. # 13-14, 15 y 16)

… que son los más simples de de implementar fisicamente, ya que sólo requieren de uno o dos transistores

… constituyen tres de las ***compuertas lógicas*** fundamentales: **NOT**, **NAND** y **NOR**

Si conectamos un inversor (una compuerta NOT) a la salida de una compuerta NAND, el nuevo circuito se llama una compuerta **AND**

Similarmente, si conectamos una compuerta NOT a la salida de una compuerta NOR, obtenemos una compuerta **OR**

Si bien las compuertas NAND y NOR son las más simples de implementar —con dos transistores—

… conceptualmente es más fácil trabajar con las compuertas AND y OR

[Aquí retomamos el nivel 0]

Las próximas diapositivas muestran los símbolos que se usan para representar las compuertas NOT (o inversor), AND y OR

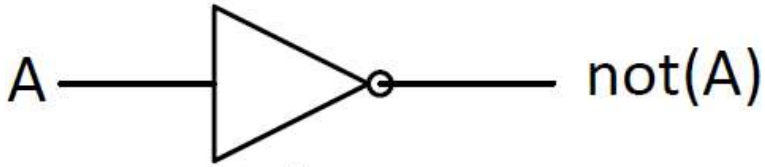
… y la función que cada una calcula:

- la tabla debajo del símbolo (y que ya vamos a explicar)

Las compuertas NOT, AND, OR, NAND y NOR son los cinco bloques básicos principales de la lógica digital:

- a veces se agrega la compuerta **XOR** (*exclusive* OR)

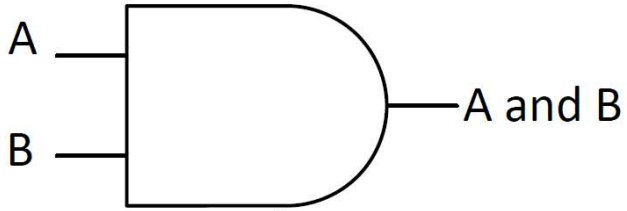
Compuerta NOT, de un input:
output = $\tilde{A}$ = ¬A = not(A)



A	not(A)
0	1
1	0

Compuerta AND, de dos inputs:

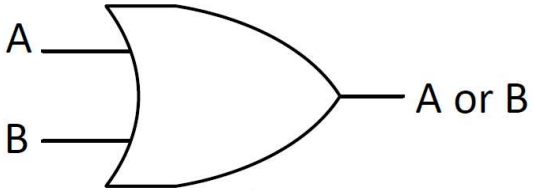
output = $A \bullet B = A \wedge B$ = *A* and *B*



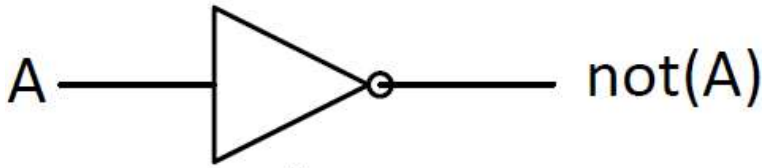
A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Compuerta OR, de dos inputs:

output = $A + B = A \vee B$ = *A* or *B*

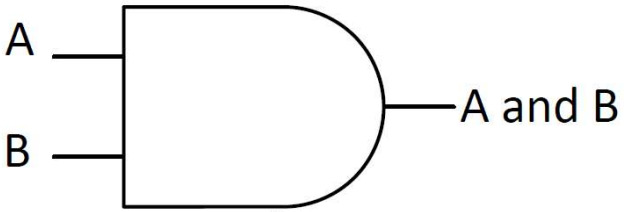


A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

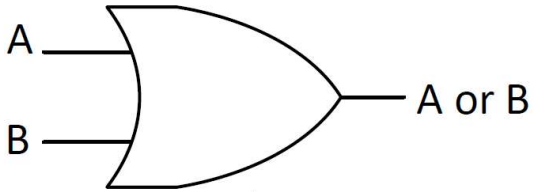


A	not(A)
0	1
1	0

Compuerta NOT: output = $\tilde{A}$ = ¬*A*



A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1



A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Compuerta AND:
output = $A \bullet B = A \wedge B$

Compuerta OR: output = $A + B = A \vee B$

Empleamos el **álgebra** (de *switching*) **de Boole** para describir mediante ecuaciones lógicas las funciones lógicas correspondientes a los circuitos que pueden ser construidos combinando compuertas

[G. Boole fue un matemático inglés del siglo XIX]

En el álgebra de Boole, las variables y funciones pueden tomar sólo los valores 0 y 1

… y típicamente hay tres operadores:

OR ("o" lógico —*disjunción*):

$A + B$ es 1 si cualquiera de las variables es 1

AND ("y" lógico —*conjunción*):

$A \bullet B$ es 1 sólo si ambos inputs son 1

NOT (*negación*):

$\neg A$ es 1 sólo si el input es 0

Leyes del álgebra de Boole

de identidad: $A + 0 = A$ y $A \bullet 1 = A$

del uno y del cero: $A + 1 = 1$ y $A \bullet 0 = 0$

Inversas: $A + \neg A = 1$ y $A \bullet \neg A = 0$

Conmutativas: $A + B = B + A$ y $A \bullet B = B \bullet A$

Asociativas: $A + (B + C) = (A + B) + C$ y $A \bullet (B \bullet C) = (A \bullet B) \bullet C$

Distributivas:

$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$ y $A + (B \bullet C) = (A + B) \bullet (A + C)$

de De Morgan: $\neg(A \bullet B) = \neg A + \neg B$ y $\neg(A + B) = \neg A \bullet \neg B$

Una función de Boole tiene una o más variables de entrada (*inputs*)

… y produce un resultado (*output*) que depende sólo de los valores de las variables de entrada —**lógica combinacional**

P.ej., la diapositiva #23 muestra tres funciones de Boole:

- las funciones correspondientes a las compuertas NOT (una variable de entrada: *A*), AND y OR (dos variables de entrada c/u: *A* y *B*)
- … especificadas mediante sendas **tablas de verdad**

Las otras tres compuertas (y funciones de Boole) comunes, expresadas en función de las compuertas explicadas anteriormente (ver tablas en la próxima diapositiva):

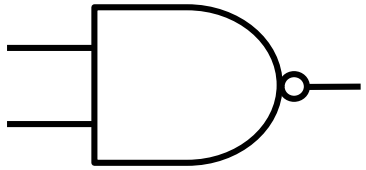
NAND —el inverso de la compuerta AND— es una compuerta **completa** → *cualquier función de Boole* puede ser calculada usando sólo compuertas NAND: $\neg(A \bullet B)$

NOR —el inverso de la compuerta OR— también es una compuerta **completa** → cualquier función de Boole puede ser calculada usando sólo compuertas NOR: $\neg(A+B)$

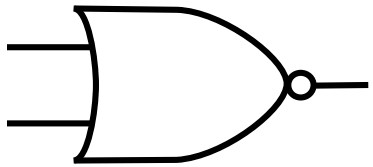
XOR —se puede construir a base de compuertas AND, OR y NOT:
$A \oplus B = (\neg A \bullet B) + (A \bullet \neg B)$

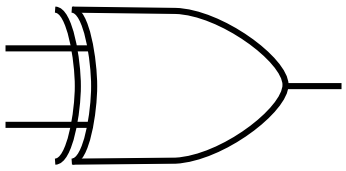
… o bien sólo a base de compuertas NAND, o bien sólo a base de compuertas NOR

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0



A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0





A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Para ver la aplicabilidad de estas compuertas y funciones, diseñemos un circuito que sume números (expresados como números binarios)

Las reglas para sumar dos sumandos de un bit c/u son las siguientes:

- 0 + 0 = 0, sin reserva (*carry*) (en rigor, la reserva es 0)
- 0 + 1 = 1 + 0 = 1, sin reserva (en rigor, la reserva es 0)
- 1 + 1 = 10, es decir, el dígito correspondiente a la suma es 0 y hay una reserva (*carry*) de 1 que pasa a la próxima posición a la izquierda

Por lo tanto, cuando se suman dos dígitos, *A* y *B*, se producen dos resultados → un sumador de (sumandos de) un bit tiene dos outputs:

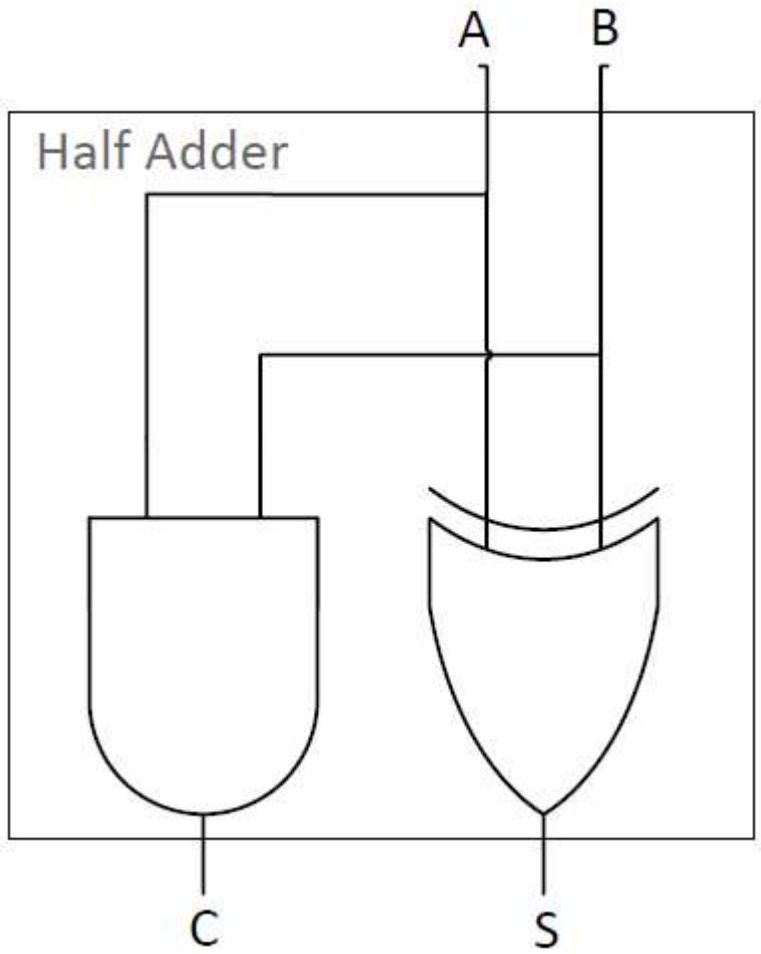
- un dígito (binario), *S*, correspondiente a la suma
- un dígito (binario), *C*, correspondiente a la reserva

Tabla de verdad y circuito digital —llamado ***half adder***— de un sumador de un bit, con dos inputs, *A* y *B*, y dos outputs, *S* y *C*

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$A \bullet B$

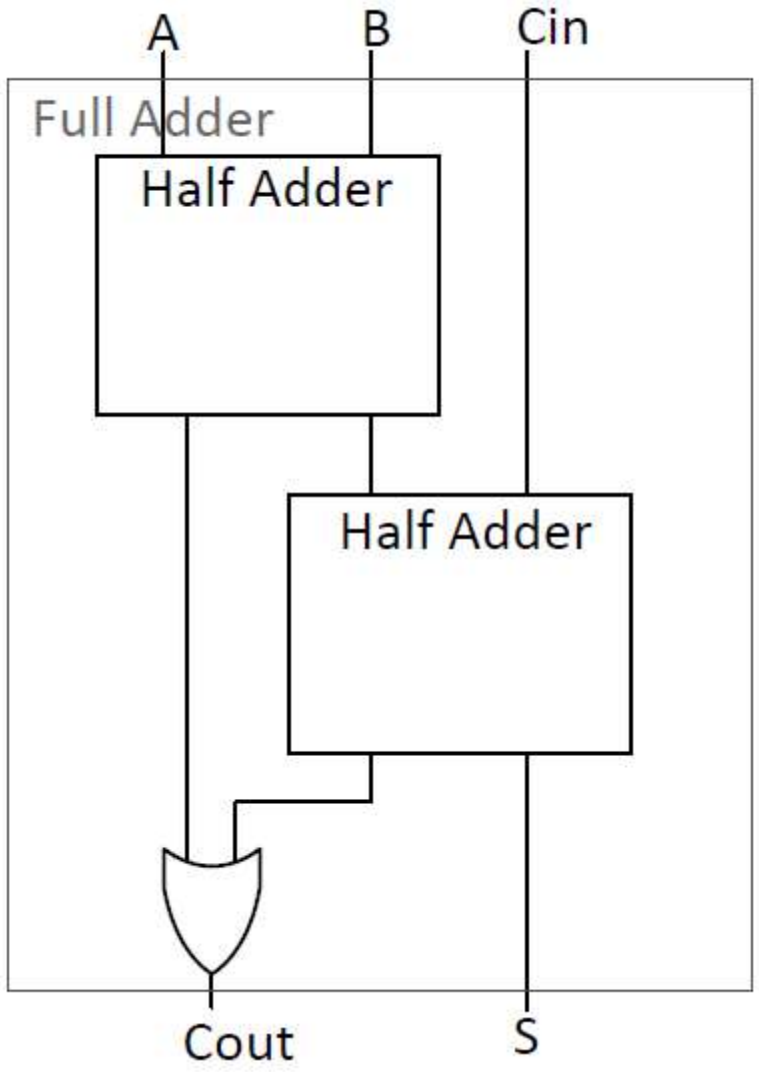
$A \oplus B$



Un ***full adder*** —para sumandos de un bit— es construido a partir de dos *half adders*

Suma tres inputs: *A*, *B* y la reserva, C_{in}, que viene desde la derecha

Y produce dos outputs: la suma, *S*, y la reserva hacia la izquierda, C_{out}



P.ej., si quiero sumar dos números binarios de cuatro dígitos cada uno, 1 0 1 1 + 0 1 1 0, escribo los números uno debajo del otro (alineados, tal como lo haría si estuviera sumando números decimales):

$$\begin{array}{rr} & 1\ 0\ 1\ 1 \\ + & 0\ 0\ 1\ 0 \\ \hline = & \mathbf{1\ 1\ 0\ 1} \end{array}$$

… y voy sumando pares de dígitos en una misma columna (de un mismo color), a partir de la columna de más a la derecha:

1 + 0 = **1** y no produce reserva

1 + 1 = **0** pero produce una reserva de 1 que pasa a la columna de la izquierda

0 + 0 = 0 pero hay que sumarle la reserva de 1 que viene desde la columna de la derecha → 0 + 1 = **1** y no produce reserva

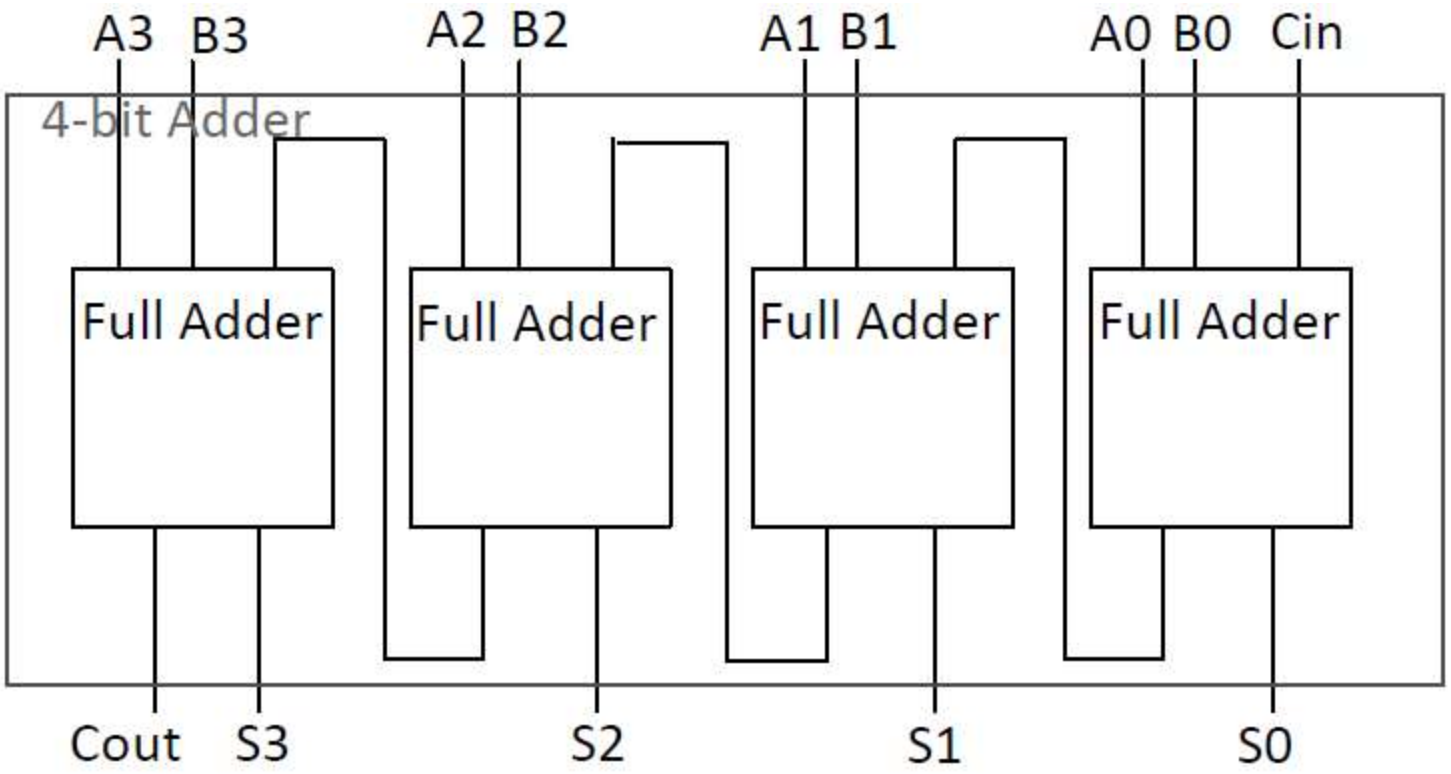
1 + 0 = **1** y no produce reserva

Por lo tanto, un (circuito) sumador de sumandos de 4 bits

$$A_3A_2A_1A_0 + B_3B_2B_1B_0$$

… necesita 4 full adders, conectados como se muestra a continuación:

el C_{out} del full adder para el bit i-ésimo se conecta al C_{in} del full adder para el bit (i+1)-ésimo



¿ Y qué pasa si queremos restar números de 4 bits ?

- ¿ podemos aprovechar el sumador ?

Sabemos que $A - B = A + (-B)$

… es decir, en lugar de restar el *sustraendo* (*B*) al *minuendo* (*A*)

… podemos, equivalentemente, sumar al minuendo

… el *inverso aditivo* del sustraendo ($-B$)

¿ Por qué esto ayuda ?

- recordemos que en complemento de 2 el inverso aditivo $-B$ se obtiene primero invirtiendo cada 0 de *B* a 1 y cada 1 de *B* a 0, y luego sumando 1 al patrón de bits resultante (simplemente poniendo un 1 en el *carry-in*)

P.ej., si queremos restar:

	0	1	0	1	1	0	—minuendo *A*
–	0	0	1	1	0	1	—sustraendo *B*

… primero calculamos el inverso aditivo del sustraendo *B* (en complemento de 2), en dos pasos:

1)	1	1	0	0	1	0	—invertimos cada dígito
2)	1	1	0	0	1	1	—… y sumamos 1 al resultado

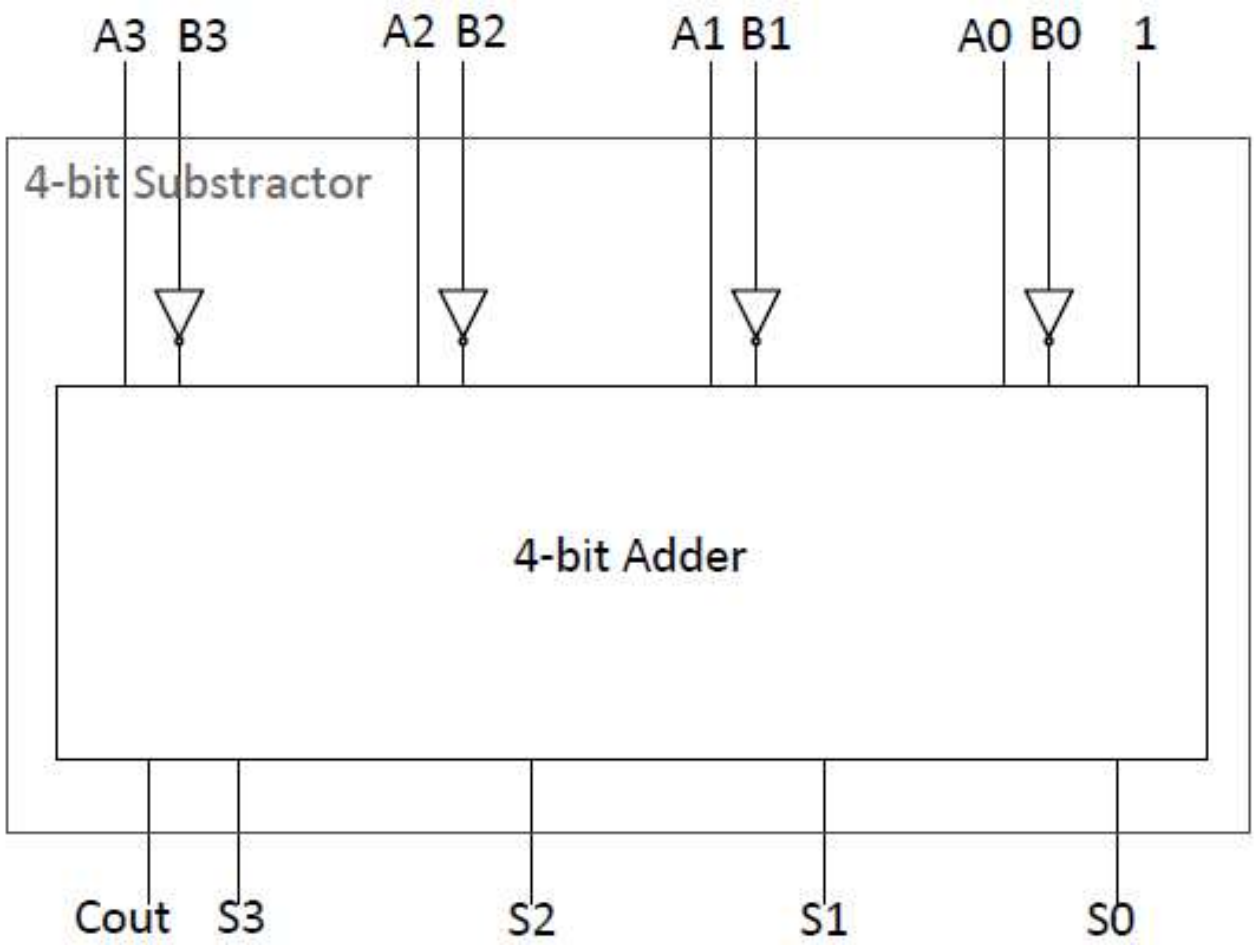
… y así convertimos la resta anterior en una suma:

	0	1	0	1	1	0	— *A*
+	1	1	0	0	1	1	— –*B*

Por lo tanto, un (circuito) restador de operandos de 4 bits

$$A_3A_2A_1A_0 - B_3B_2B_1B_0$$

… es como se muestra a continuación:



¿ Cómo construimos ahora un circuito sumador/restador de operandos de 4 bits ?

- ¿ podemos hacerlo usando sólo los elementos vistos hasta ahora ?

El circuito va a tener una entrada *A* de 4 bits, otra entrada *B* de 4 bits, y una salida *S* de 4 bits

… pero de alguna manera tenemos que decirle si queremos que sume, $S = A + B$, o bien que reste, $S = A - B$

… → tenemos que controlar la operación del circuito

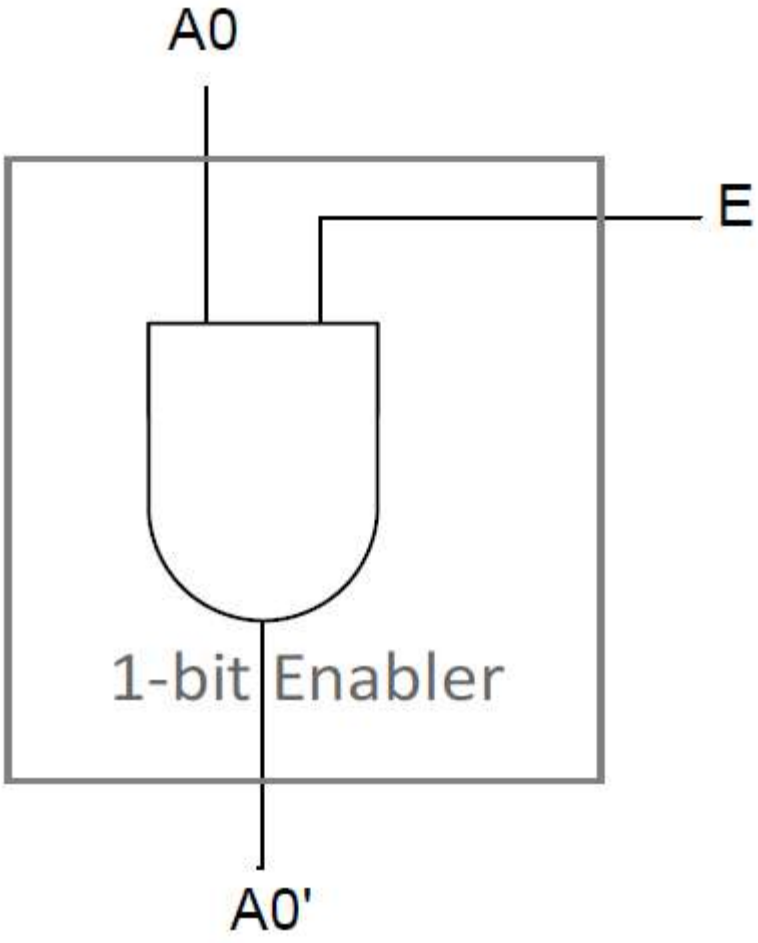
… → tenemos que poder **programar** el circuito:

- además de inputs de datos, el circuito tiene que tener ***inputs de control***

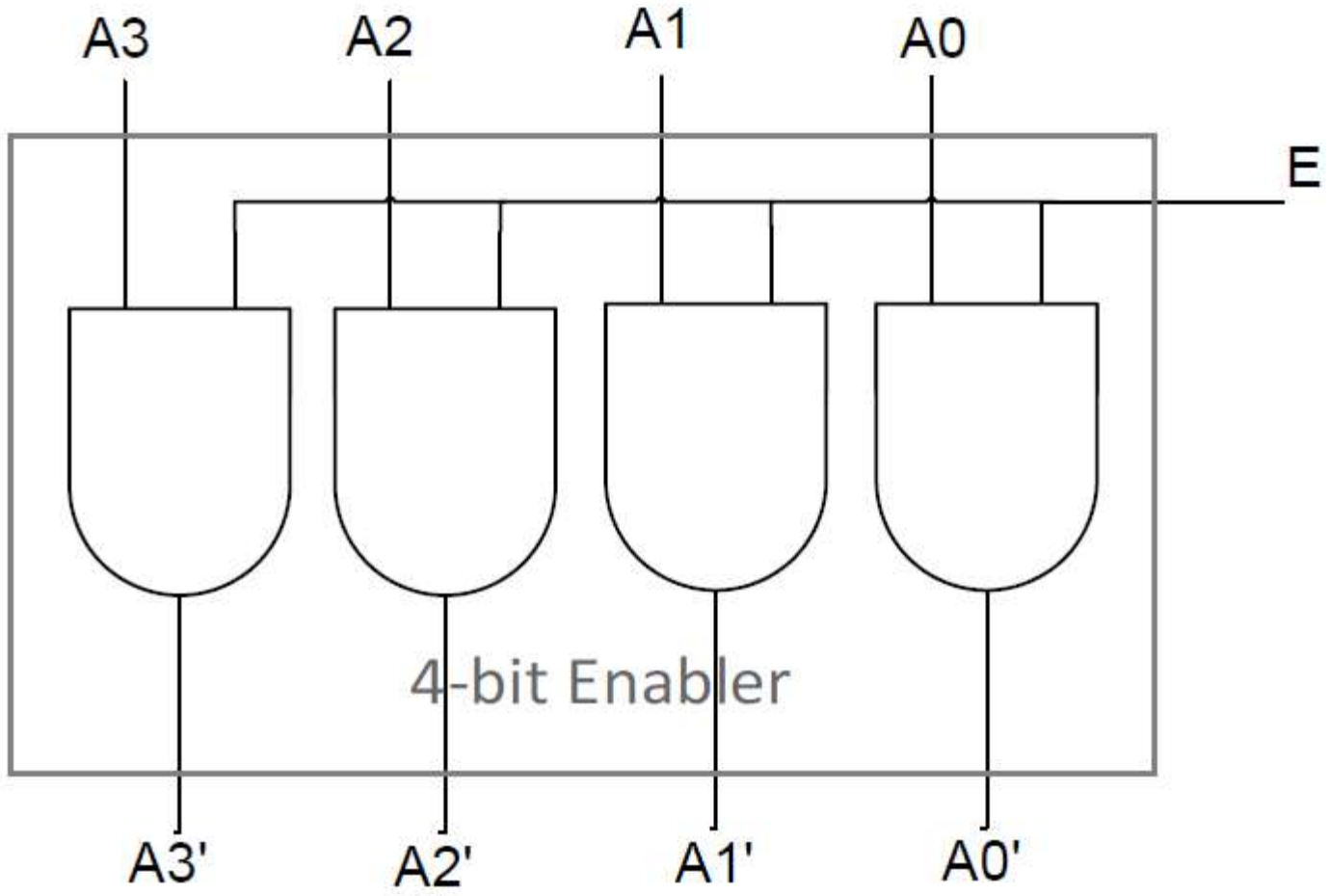
P.ej., este es un circuito muy simple con un input de control, *E*, además de un input de datos, A_0 :

dependiendo del valor de *E*, el output A_0' será 0 (si *E* = 0), o exactamente igual a A_0 (si *E* = 1)

E	A0'
0	0
1	A0



P.ej., esta es la versión del circuito anterior para un input de datos de cuatro bits, $A_3A_2A_1A_0$:



Un **multiplexor** es un circuito con 2^n inputs de datos (de un cierto número de bits),

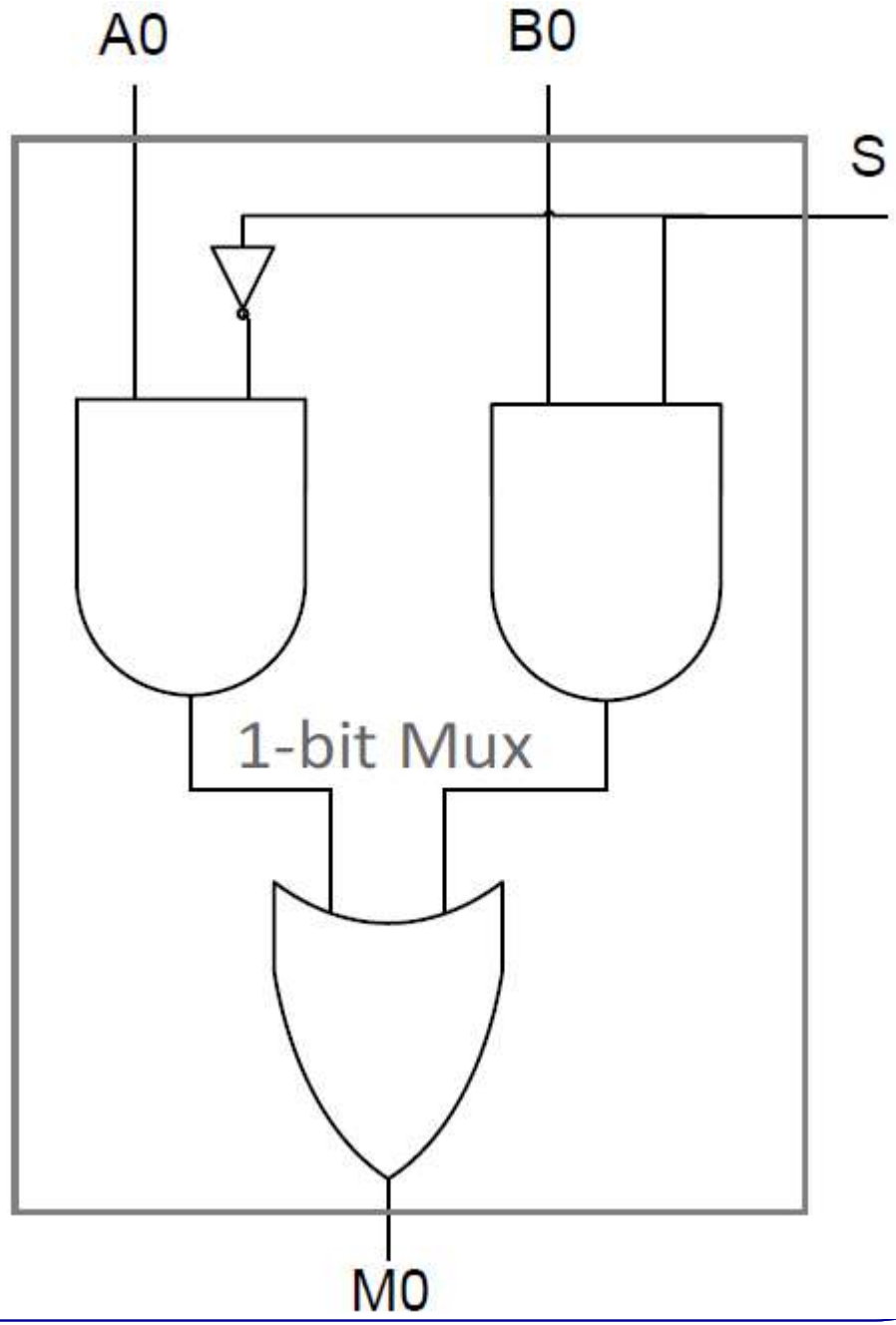
… un output (del mismo número de bits que los inputs),

… y *n* inputs de control (de un bit c/u) que seleccionan uno de los inputs de datos y lo ponen en el output

P.ej., para n = 1 e inputs de un bit, hay dos inputs de datos, A_0 y B_0, un input de control, S, y un output, M_0:

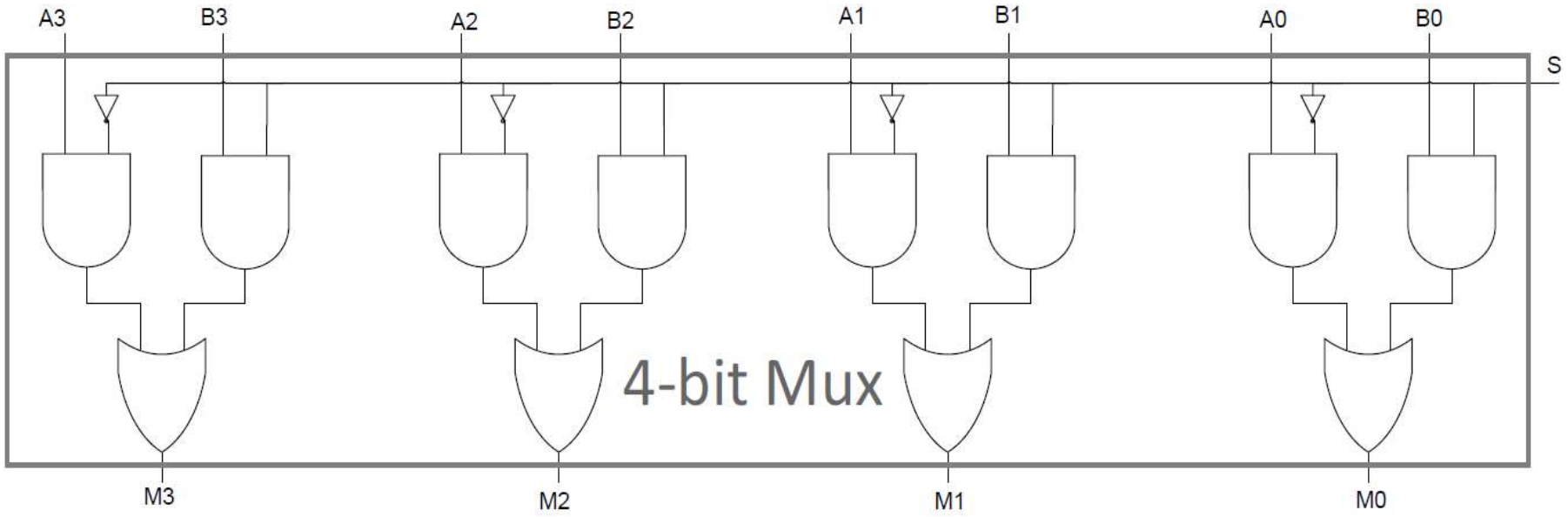
dependiendo del valor de S, M_0 será exactamente igual a A_0 (si S = 0), o a B_0 (si S = 1)

S	M0
0	A0
1	B0



P.ej., para n = 1 e inputs de 4 bits, hay dos inputs de datos, $A_3A_2A_1A_0$ y $B_3B_2B_1B_0$, un input de control, S, y un output, $M_3M_2M_1M_0$:

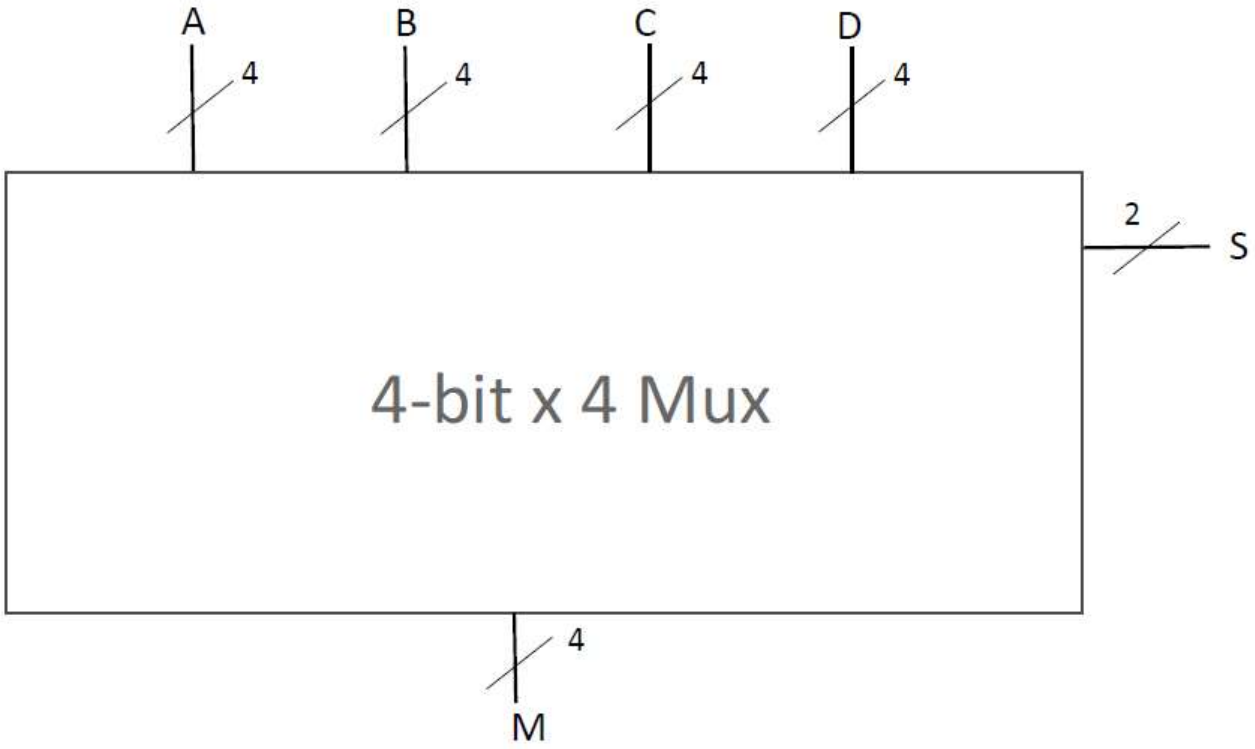
dependiendo del valor de S, $M_3M_2M_1M_0$ será exactamente $A_3A_2A_1A_0$ (si S = 0) o $B_3B_2B_1B_0$ (si S = 1)



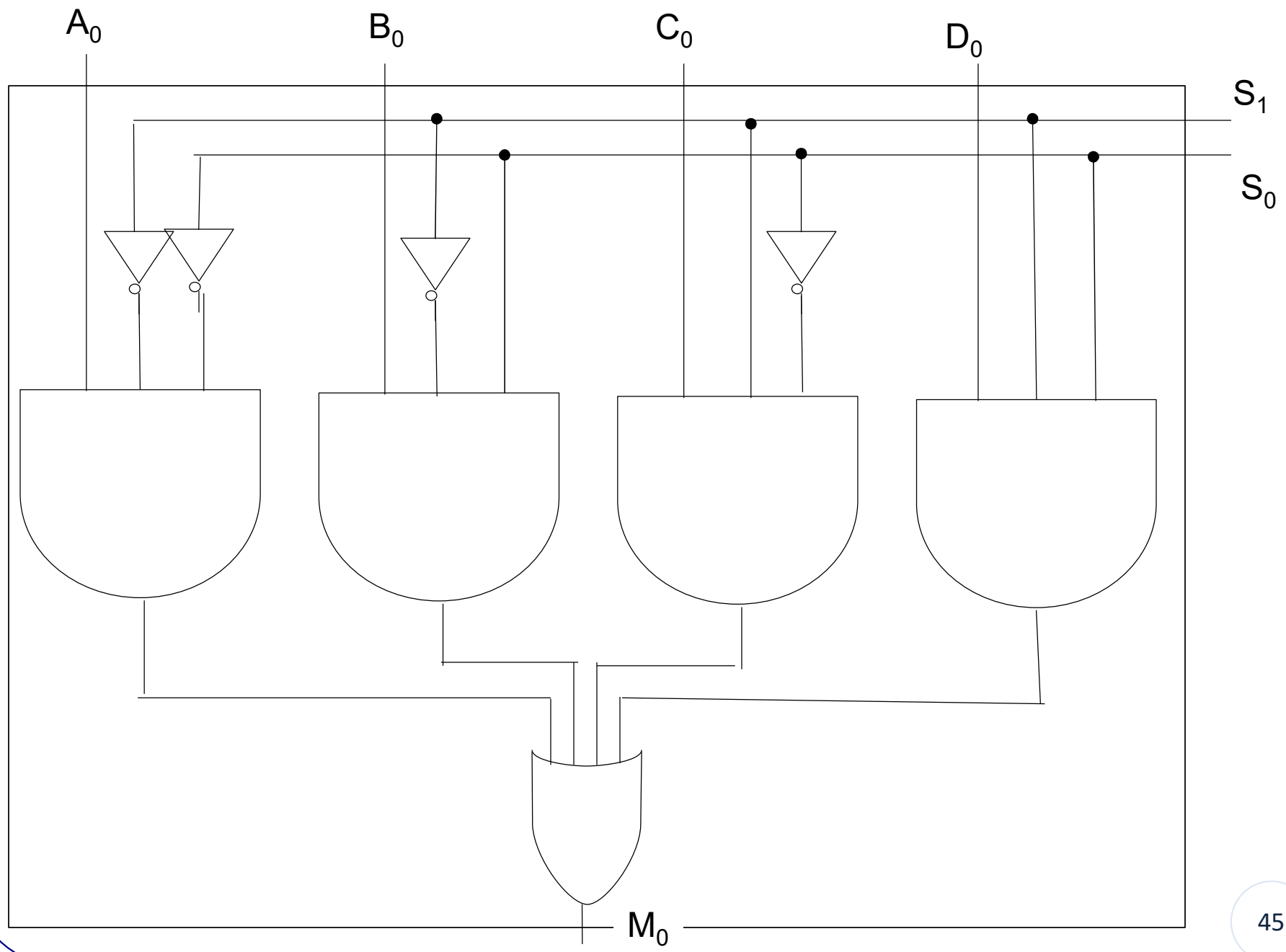
P.ej., para n = 2 e inputs de 4 bits, hay cuatro inputs de datos, A, B, C y D, dos inputs de control, S de dos bits (= S_1S_0), y un output, M:

dependiendo del valor de S, M será exactamente igual a A (si S = 00) o B (si S = 01) o C (si S = 10) o D (si S = 11)

la próx. diapo. muestra el circuito para un bit



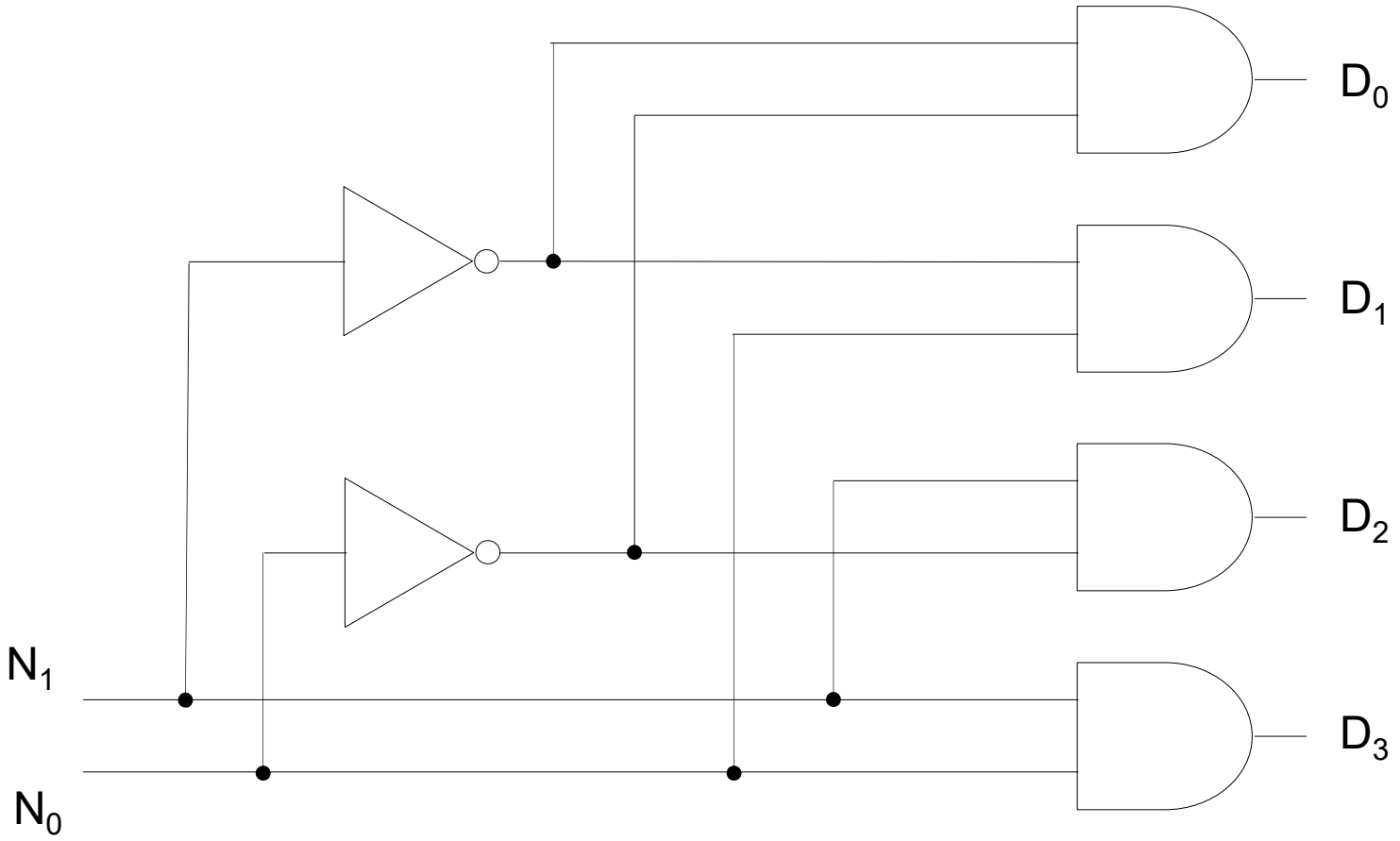
S_1	S_0	M
0	0	A
0	1	B
1	0	C
1	1	D


A_0
B_0
C_0
D_0
S_1
S_0
M_0

Un **decodificador** es un circuito que recibe como input un número de *n* bits,

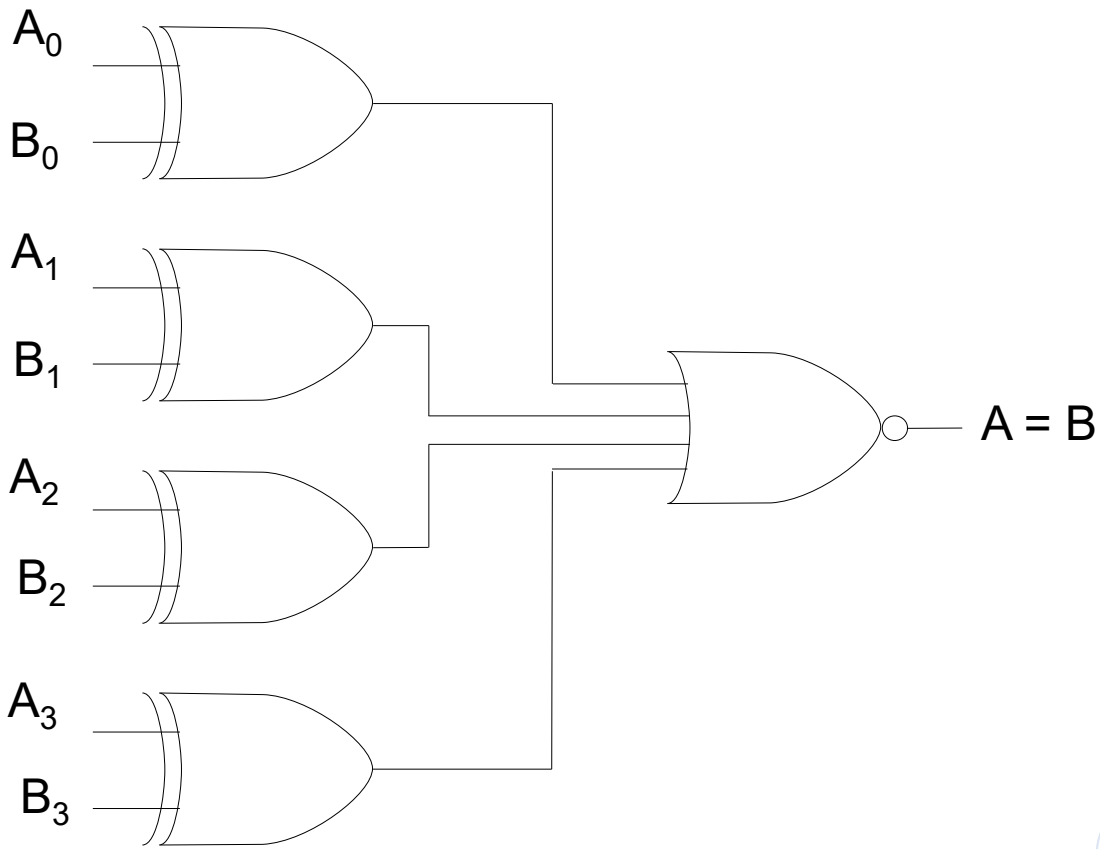
… y lo usa para poner en 1 exactamente una de 2^n líneas de salida:

- p.ej., para $n = 2$



Un **comparador** es un circuito que compara dos patrones de bits del mismo largo (los inputs) y produce un 1 (el output) si los patrones son exactamente iguales (bit a bit); de lo contrario, un 0:

p.ej., para el caso de patrones de 4 bits cada uno



Nuestra primera versión del circuito sumador/restador de 4 bits (próx. diap.) incluye

los inputs de datos, *A* y *B*, de 4 bits c/u

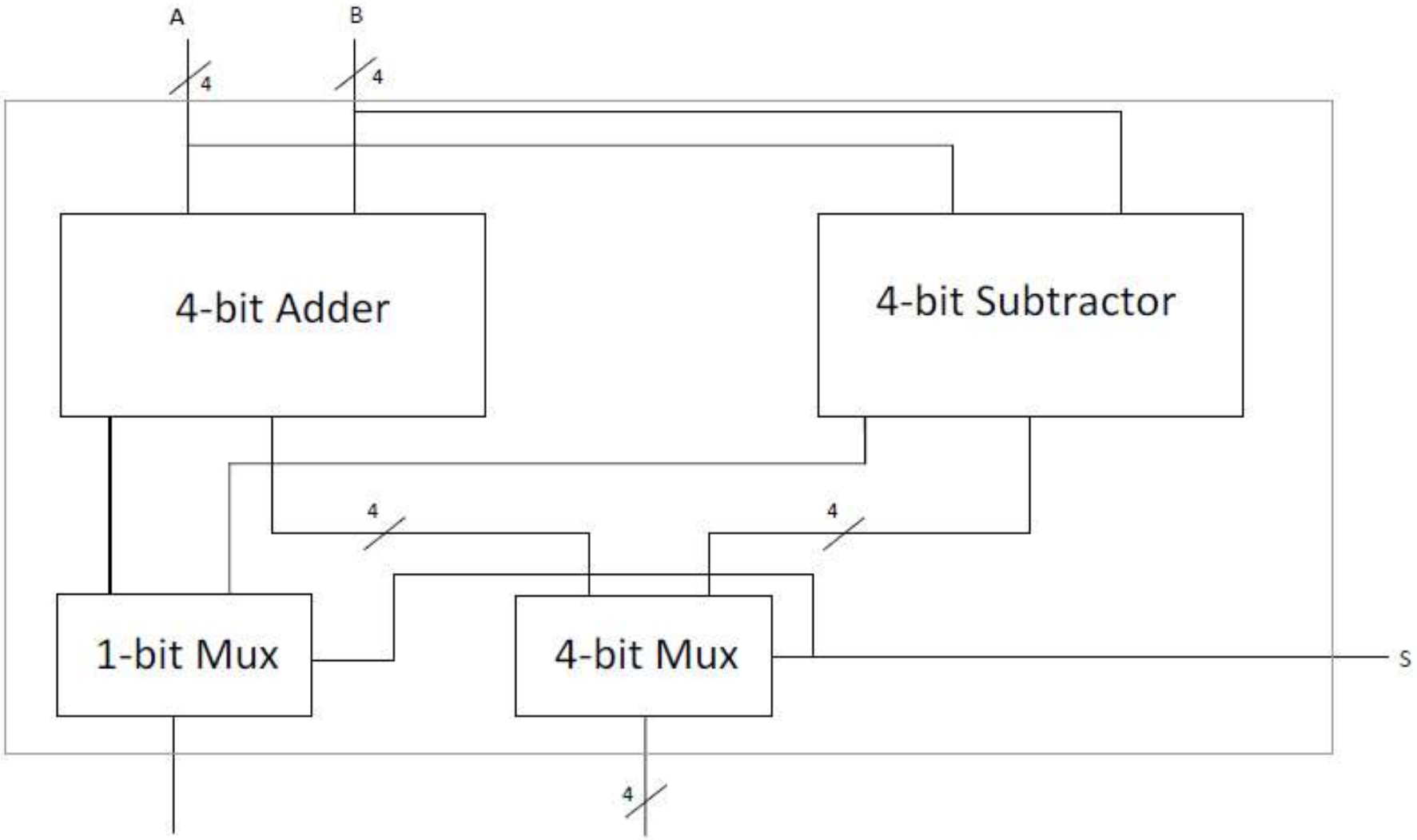
el sumador de 4 bit (diap. #30)

el restador de 4 bits (diap. #33)

un multiplexor de 4 bits para seleccionar entre el output del sumador y el output del restador (diap. #39)

un multiplexor de un bit para seleccionar entre el C_{out} del sumador y el C_{out} del restador (diap. #38)

un input de control, *S*, que controla ambos multiplexores


A
B
4
4
4-bit Adder
4-bit Subtractor
4
4
1-bit Mux
4-bit Mux
S
4

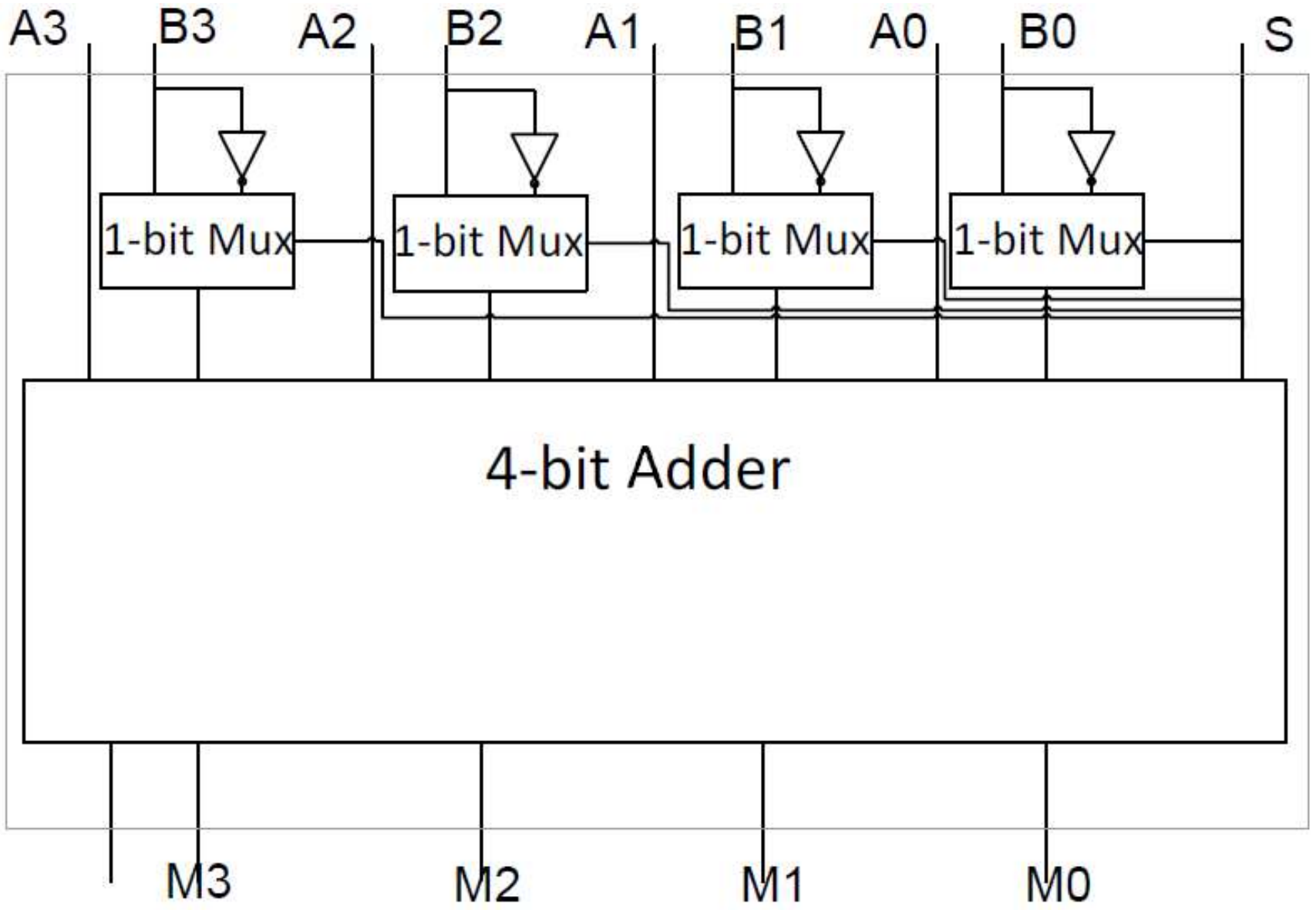
Nuestra segunda versión del circuito sumador/restador de 4 bits (próx. diap.) aprovecha el hecho de que el circuito restador es esencialmente el mismo que el circuito sumador,

… sólo que recibe como uno de sus inputs el inverso aditivo del sustraendo *B* (en vez *B* propiamente tal):

Así, el input *B* puede entrar directo (en el caso de una suma propiamente tal) o invertido (en el caso de una resta)

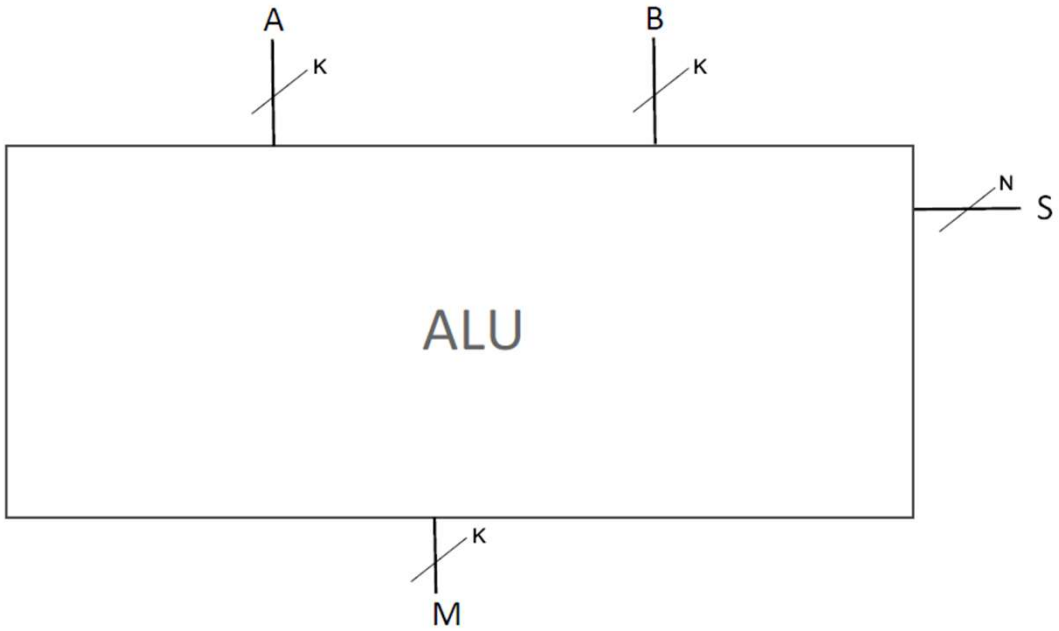
… por lo que ambas versiones pasan por un multiplexor controlado por el input de control *S* antes de entrar al circuito sumador:

- si $S = 0$, entonces el mutiplexor selecciona la versión directa de *B*
- si $S = 1$, entonces el multiplexor selecciona la versión invertida de *B*
- en ambos casos, *S* también va a parar a la entrada C_{in} del sumador, de modo que en el caso de la resta constituye el 1 que hay que sumar a la versión invertida de *B* para obtener el inverso aditivo de *B* (diaps. #31 y 32)


A3
B3
A2
B2
A1
B1
A0
B0
S
1-bit Mux
1-bit Mux
1-bit Mux
1-bit Mux
4-bit Adder
M3
M2
M1
M0

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
- resultado *M*, de *k* bits
- input de control *S*, de *n* bits, para seleccionar entre 2^n operaciones distintas



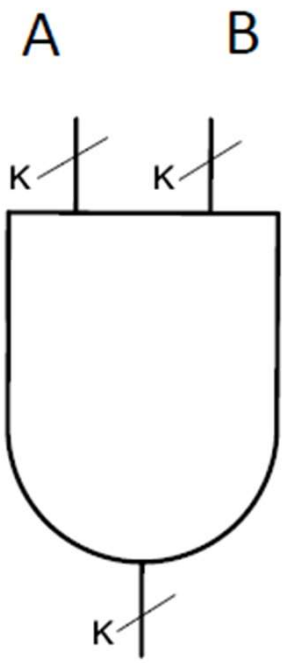
(P.ej., el circuito sumador/ restador de la diap. anterior corresponde a una ALU muy simple que sólo suma o resta operandos de *k* = 4 bits → sólo dos operaciones, por lo que *S* tiene sólo un bit (*n* = 1) **)**

Así, además de la suma y resta que ya vimos, las ALUs también realizan las operaciones lógicas básicas (NOT, AND, OR, XOR) sobre sus inputs,

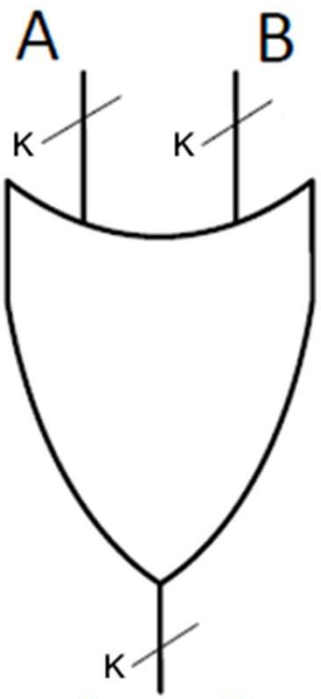
… ya que son muy útiles, tanto por sí mismas, como formando parte de otras operaciones más complejas (p.ej., el NOT para hacer la resta)

Estas operaciones —muy simples de implementar— se realizan bit a bit (*bitwise*) y se representan como las figuras de la próx. diap.:

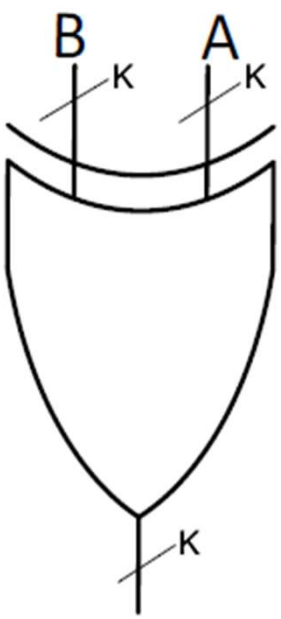
P.ej.,								
	A :	0	1	1	0	0	1	0	1
	B :	1	1	0	1	0	0	0	1
	NOT A :	1	0	0	1	1	0	1	0
	A AND B :	0	1	0	0	0	0	0	1
	A OR B :	1	1	1	1	0	1	0	1
	A XOR B :	1	0	1	1	0	1	0	0



A and B



A or B



A xor B

not(A)

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

… es decir, el desplazamiento del patrón de bits de *A* en un bit, a la izquierda o a la derecha, respectivamente, como se muestra en los ejemplos de la próx. diap.

Cuando el patrón de bits se desplaza en un bit, p.ej., a la izquierda (*shift left*), ocurre lo siguiente:

- el bit en la posición 0 (el de más a la derecha) pasa a la posición 1, el de la posición 1 a la posición 2, …, y el de la posición n–2 a la posición n–1
- el bit en la posición n–1 (el de más a la izquierda) no pasa a otra posición, sino que simplemente se descarta
- y el bit en la posición 0, que quedó “vacío”, recibe un 0

Simétricamente para el caso del desplazamiento a la derecha

en el bit “vacío” que queda a la izquierda, se pone **el mismo valor que había originalmente** en esta posición; en este caso, un **0**

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

P.ej., podemos verlas como la multiplicación por dos (*shift left*) y la división entera por dos (*shift right*)

… análogamente a lo que son 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 de la secuencia de dígitos) en nuestro sistema decimal habitual:

$$47293 \times 10 = 472930$$

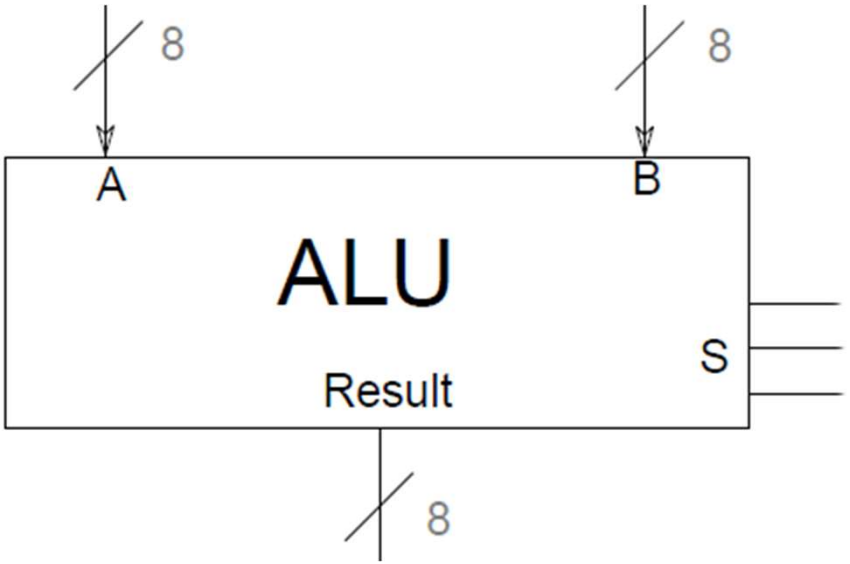
$$47293 / 10 = 4729$$

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

Así, tenemos nuestra 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*)

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

- 0 0 1 → *A* – *B* 0 1 1 → *A* or *B* 1 1 1 → shift right(*A*)



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

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

- a veces, una acción tiene que preceder a otra
- a veces, dos acciones deben ocurrir simultáneamente

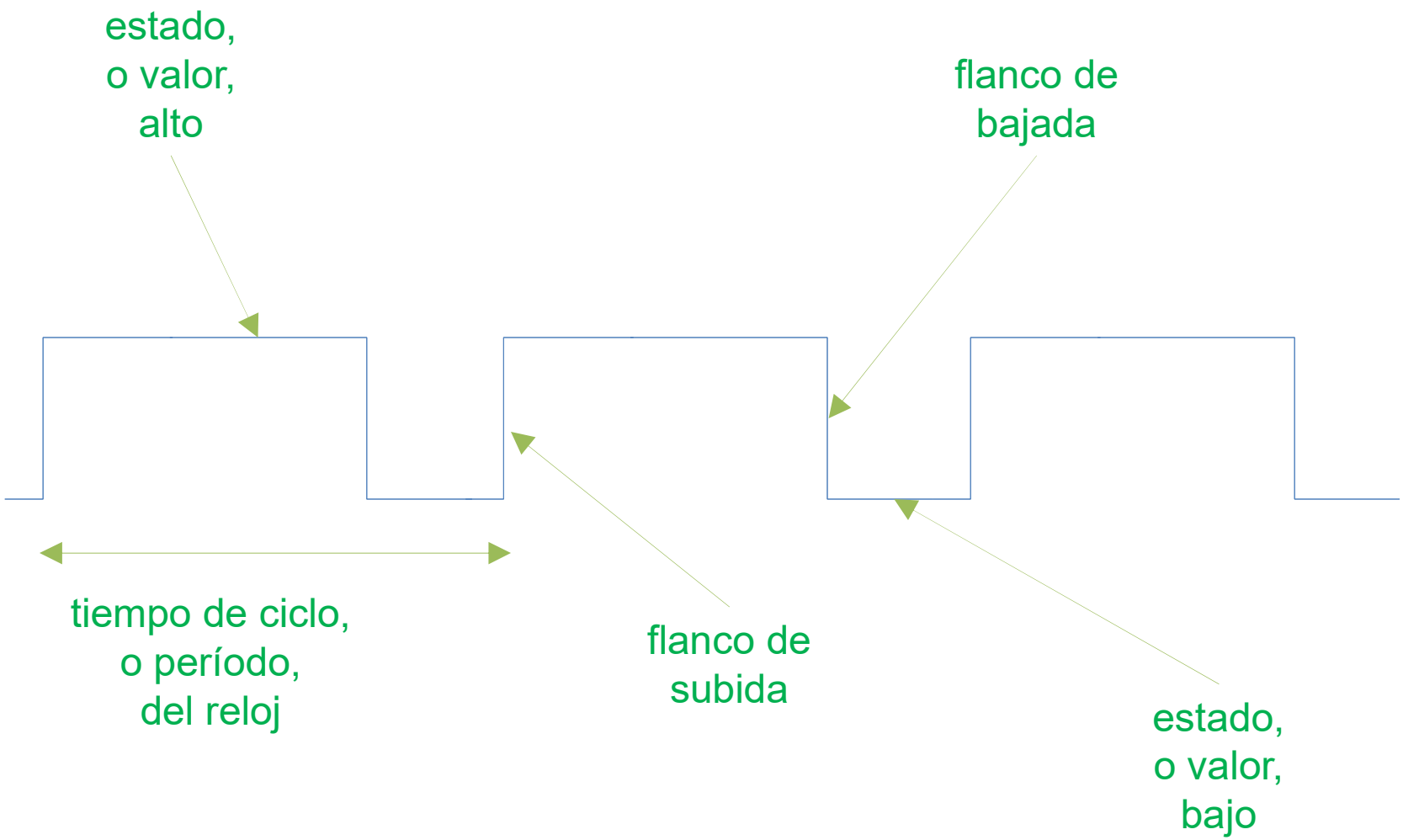
Los circuitos digitales usan *relojes* para proporcionar sincronización

Un **reloj** —en un circuito digital— es simplemente un circuito que emite una serie de pulsos, con dos propiedades:

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

El **tiempo de ciclo** (o *período*) del reloj es el intervalo de tiempo entre los flancos correspondientes —los *flancos de subida* o los *flancos de bajada*— de dos pulsos consecutivos

La **frecuencia** del reloj —p.ej., desde 100 MHz y hasta 4 GHz— es el inverso del tiempo de ciclo


estado,
o valor,
alto
flanco de
bajada
tiempo de ciclo,
o período,
del reloj
flanco de
subida
estado,
o valor,
bajo

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 instrucciones a ser ejecutadas

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

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

- el output del circuito depende tanto de los datos de input como del valor almacenado dentro del circuito

El funcionamiento de un *latch* SR se basa en 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 (ver próx. diap.)
- lo construimos a partir de dos compuertas NAND (o dos compuertas NOR)

Efecto del retardo de propagación en un circuito muy simple:

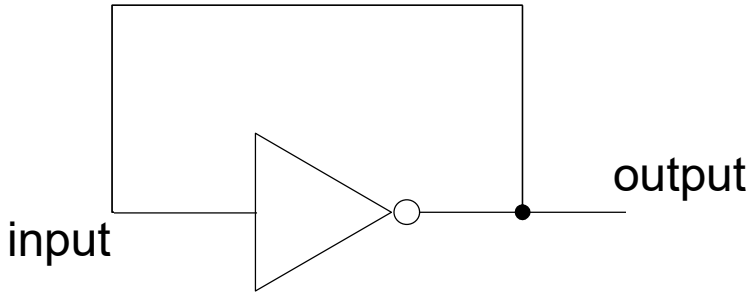
- si el output del circuito es 0, entonces el input del circuito es 0

  … 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

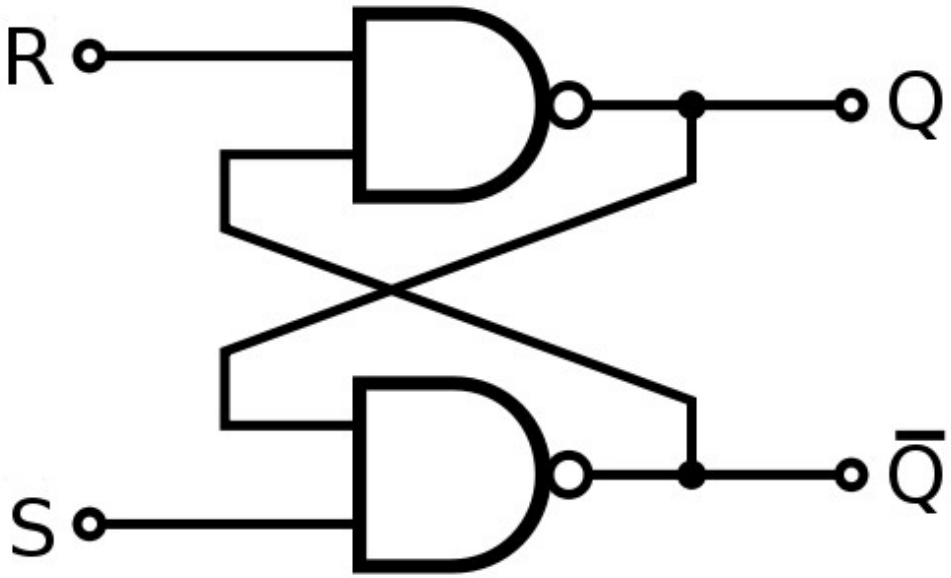
  … y sólo una vez que ha transcurrido otro retardo de propagación, el output vuelve a 0 nuevamente

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



Latch S-R:

- 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 (o, más propiamente, *combinatorio*)
- cuando *R* = *S* = 0, el circuito se vuelve no determinista

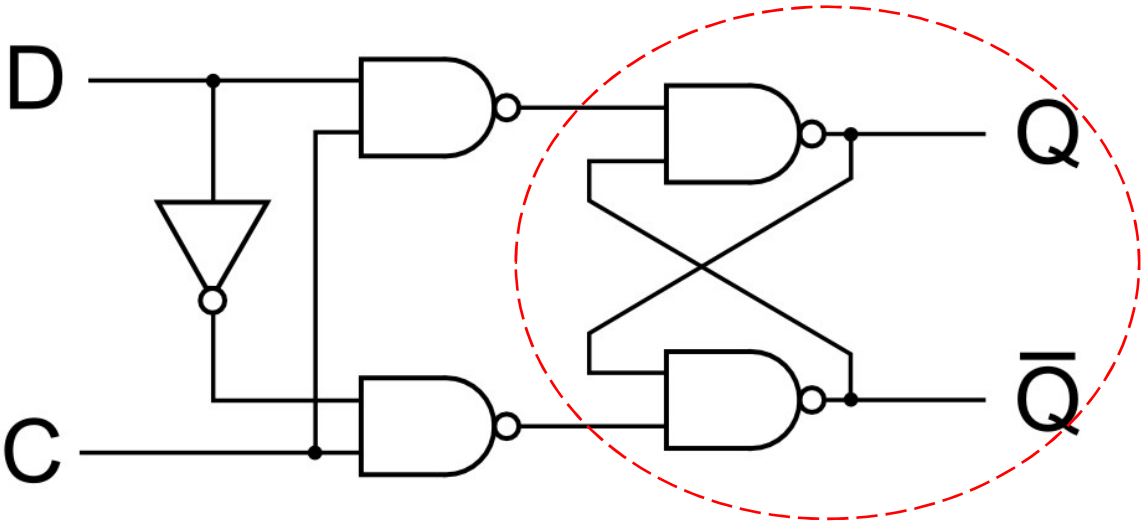


R	S	Q^{t+1}
0	0	-
0	1	1
1	0	0
1	1	Q^t

***Latch* D** (controlado por reloj) —a menudo conviene permitir que el latch cambie de estado sólo en ciertos instantes específicos:

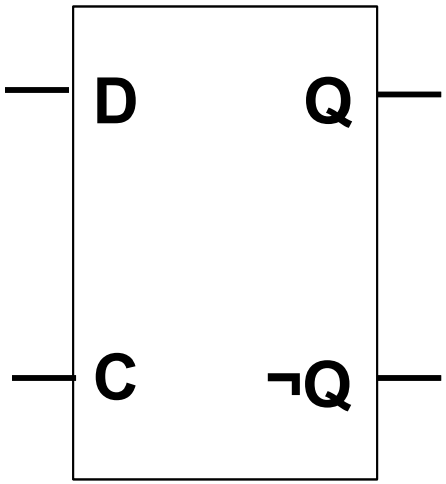
- evita el caso no determinista del *latch* SR previniendo su ocurrencia
- el input *D* y su complemento, ¬*D*, entran a sendas compuertas NANDs justo antes del *latch*

  … → los inputs *R* y *S* del *latch* no pueden ser ambos 0 simultáneamente
- cuando el reloj, *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 reloj) —más esquemáticamente:



D: dato (input)
C: control (reloj)
***Q*: estado** (output)

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

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

… que es lo que podría ocurrir en el caso de los *latches* como el anterior, en que la señal *D* puede ser leída en cualquier instante mientras el reloj está en su valor alto, o 1

Esto se puede conseguir con un circuito conocido como **flip-flop**

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

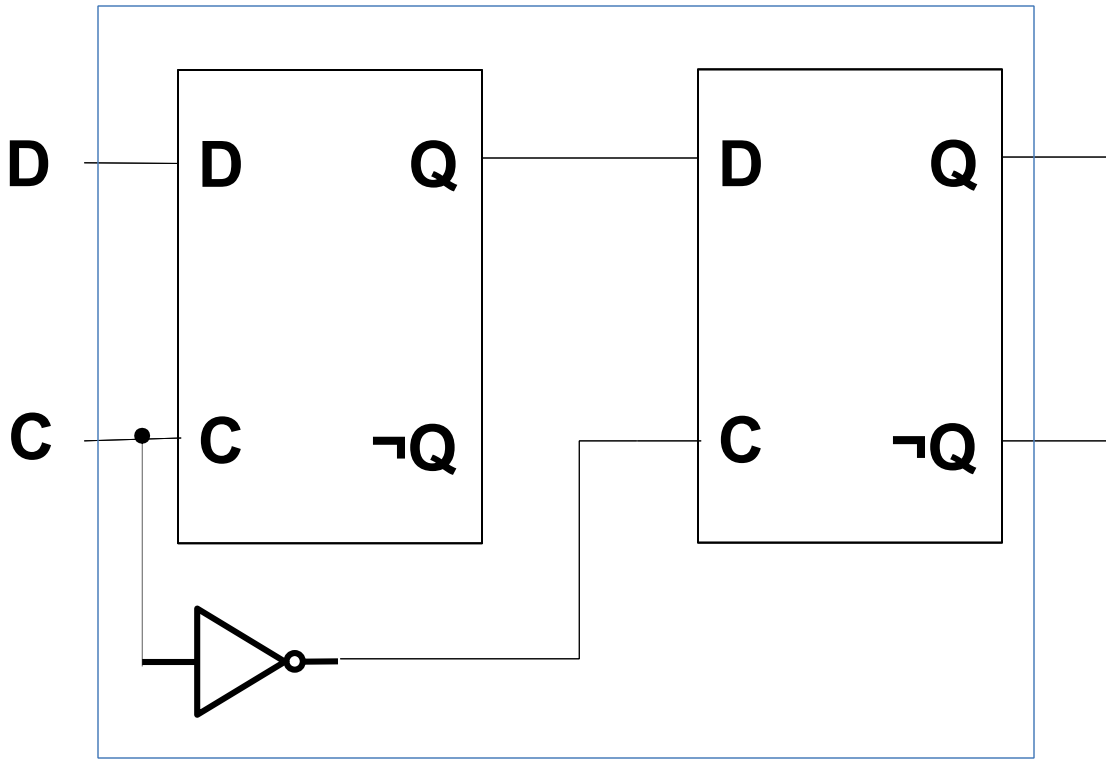
… ocurre **durante la transición** del reloj de 0 a 1 (*flanco de subida*) o de 1 a 0 (*flanco de bajada*)

(… y no cuando el reloj está en 1 **)**

La próxima diapositiva muestra un **flip-flop D** construido a partir de dos latches D y un inversor (o compuerta NOT)

Flip-flop D:

- cuando la señal del reloj *C* cambia de 0 a 1 (o, en otros circuitos equivalentes, de 1 a 0), el output *Q* almacena el valor del input *D*
- usamos un arreglo de flip-flops D para construir un **registro** que puede almacenar un dato de varios bits, tal como un byte o una palanbra



C	D	Q^{t+1}
0/1/↓	0/1	Q^t
↑	0	0
↑	1	1

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 bits en los computadores modernos):

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