Organización de computadoras - Teóricos

Resumen 1

Representación de datos

Las computadoras almacenan datos e instrucciones en memoria, siempre de la misma manera: utilizando el sistema binario, o sea, con ceros y unos. En esencia, en el nivel más básico, todo lo que la computadora puede distinguir es si un bit vale 0 o si vale 1, que es una manera abstracta de decir que alguna cosa del mundo real (por ejemplo, una corriente eléctrica), está presente o no está presente. Y es justamente por esta conexión con el mundo real que se usa el sistema binario: porque es más fácil distinguir entre dos estados distintos, que entre muchos estados diferentes con distintos grados de intensidad, donde los límites pueden tornarse difusos entre una u otra opción.

Para ilustrar esta idea, pensamos por ejemplo en la diferencia que hay entre un interruptor común y corriente (que enciende y apaga una lamparita), y un dimmer (que puede graduar la intensidad de la lamparita). En el primer caso, sólo tenemos dos opciones, pero súper claras y definidas: la lámpara está encendida o apagada. En el segundo caso podríamos establecer alguna convención y diferenciar entre diez tipos de intensidades desde la luz apagada hasta muy brillante (lo cual sería ideal, porque permitiría trabajar directamente con el sistema decimal), pero sería complicado de operar y propenso a errores, porque no es fácil distinguir entre dos intensidades que están cerca.

No es importante saber de electrónica, pero sí está bueno tener una mínima idea de dónde viene todo lo que hacemos en informática, cuáles son los fundamentos y qué es, en definitiva, lo que está pasando adentro de una computadora. Esto es lo que empezaremos a comprender con la primera parte de la materia.

Tipos de datos

Las computadoras manejan 4 tipos básicos de datos binarios:

- Números enteros sin/con signo
- Números reales con signo
- Números decimales codificados en binario (BCD)
- Caracteres (por ejemplo, con la tabla ASCII)

Un solo recordatorio: con n bits puedo representar 2^n cadenas binarias distintas. (Lo demás puede leerse en el resumen de los prácticos).

Lógica digital

Un circuito digital es un circuito electrónico que maneja información en forma binaria, utilizando dos estados lógicos, generalmente "1" y "0", que representan niveles de voltaje alto y bajo, respectivamente.

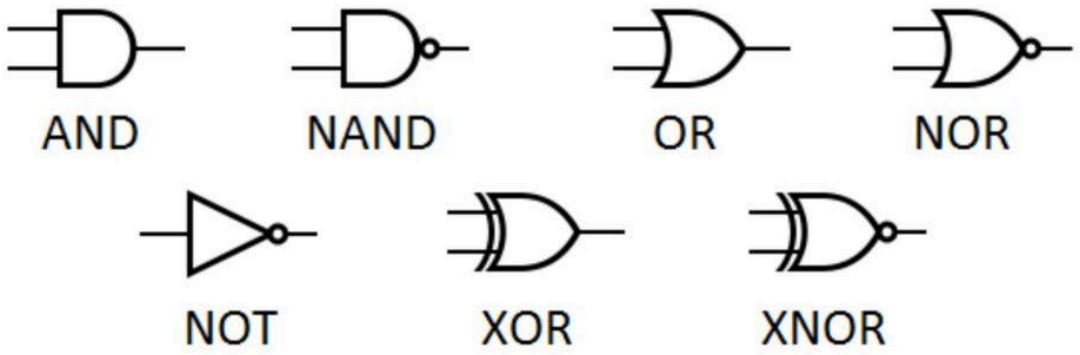
Las compuertas son dispositivos electrónicos que pueden realizar distintas funciones con estos dos valores lógicos.

Las compuertas básicas son: AND, OR, NOT, NAND, NOR y XOR. Y también está XNOR.

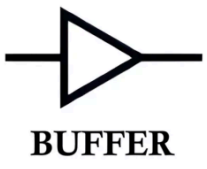
“And” viene del inglés y significa “y” (es decir, la conjunción lógica), “or” significa “o” (la disyunción inclusiva), y “not” es “no” (la negación).

La N que precede a algunas compuertas, está ahí porque se construyó con una negación. O sea: NAND, por ejemplo, viene de NOT AND. Y la X viene de *exclusive*. Por eso XOR = “exclusive or” = “disyunción exclusiva”. Combinando estos saberes, llegamos a que XNOR = NOT XOR.

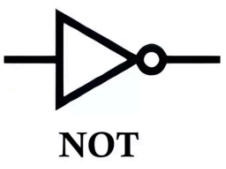
Y así como los nombres están relacionados, sus gráficos también lo están. Los pares AND y NAND, OR y NOR, XOR y XNOR difieren solamente en el punto del final. Y, según se dice, la raya que se le agrega del OR al XOR trata de representar parte de la X de *exclusive*.



Algo importante para entender y recordar mejor estos dibujos, es que la negación es únicamente el punto o círculo con relleno blanco. El triángulo que se ve en la compuerta NOT en realidad no hace nada, es simplemente un buffer (que entre otras cosas puede servir para amplificar la señal, pero eso no nos interesa). Lo importante es que la negación es el círculo con relleno blanco.



Deja pasar la señal.



Niega la señal.

¿Pero qué hacen las compuertas? Las compuertas son básicamente funciones, con inputs y un output, para señales o cargas eléctricas que representan los estados lógicos de 1 y 0. La compuerta NOT es la única que tiene un solo input, mientras que las demás tienen dos o más inputs. Todas tienen un solo output. Cada compuerta es una función distinta que convierte sus inputs en un determinado output. Estas resultan útiles para realizar diversas operaciones lógicas sobre los bits. (En el mundo real, las compuertas suelen implementarse con transistores. Es por eso que los procesadores, al ser básicamente un conjunto de compuertas lógicas interconectadas, están hechos con de millones de transistores).

Podemos definirlas o describirlas mediante tablas de verdad, de la siguiente manera:

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

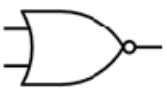
XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

NAND

INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

NOR

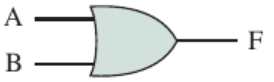
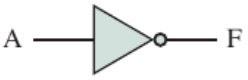
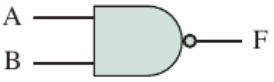
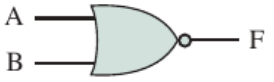
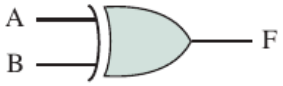


INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

Y ya que estamos, no viene mal repasar la notación algebraica:

Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

La lógica digital —tal como la conocemos hoy en día— fue inventada por George Boole a mediados del siglo XIX, quien pasó a la historia como el inventor del álgebra booleana, una rama del álgebra bastante particular. Se diferencia del álgebra elemental en dos aspectos. Primero, los valores de las variables son los valores de verdad verdadero y falso, normalmente denotados 1 y 0, mientras que en álgebra elemental los valores de las variables son números. En segundo lugar, el álgebra booleana utiliza operadores lógicos como la conjunción (denotada como ∧), la disyunción (denotada como ∨) y la negación (denotada como ¬). El álgebra elemental, por otro lado, utiliza operadores aritméticos como la suma, la multiplicación, la resta y la división. Por tanto, el álgebra de Boole es una modalidad formal de describir operaciones lógicas, de la misma manera que el álgebra elemental describe operaciones numéricas.

Esta nueva álgebra, entre otras cosas, sirve para describir los circuitos que pueden construirse combinando compuertas, donde las variables y funciones sólo pueden adoptar dos valores: 0 o 1.

Puesto que una función booleana de n variables tiene 2^n combinaciones de los valores de entrada, la función puede describirse totalmente con una tabla de 2^n renglones, donde cada uno indica un valor de la función (0 o 1) para cada combinación distinta de las entradas. En otras palabras, puede describirse mediante una tabla de verdad.

Ejemplo: la función F = (A OR B) AND C. Esta función tiene 3 valores de entrada, y por eso tendrá $2^3 = 8$ renglones. Puede describirse así:

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Aquí hay algunas identidades del álgebra booleana que serán útiles tanto para trabajar en términos abstractos como para construir circuitos.

Identidad	$1.A=A$	$0+A=A$
Nula	$0.A=0$	$1+A=1$
Idempotencia	$A.A=A$	$A+A=A$
Inversa	$A.\overline{A}=0$	$A+\overline{A}=1$
Conmutativa	$A.B=B.A$	$A+B=B+A$
Asociativa	$(AB).C=A(BC)$	$(A+B)+C=A+(B+C)$
Distributiva	$A+B.C=(A+B).(A+C)$	$A.(B+C)=AB+AC$
Absorción	$A.(A+B)=A$	$A+A.B=A$
De Morgan	$\overline{A.B}=\overline{A}+\overline{B}$	$\overline{A+B}=\overline{A}.\overline{B}$

Faltan algunas, como la doble negación, pero son conocidas: son las mismas que vemos en matemáticas (ver Capítulo 4, Algebras de Boole).

Pasemos ahora a la aplicación práctica. En el mundo real, a la hora de construir circuitos, normalmente no se usan todas las compuertas mezcladas, sino que —por cuestiones de practicidad de la fabricación del hardware— se suele usar un solo tipo de compuerta. O sea que, por lo general, los circuitos, sean cuales sean, suelen traducirse a una versión equivalente hecha con un solo tipo de compuerta lógica —siendo la compuerta NAND una de las más usadas. Por ello, es importante saber cómo expresar diversas compuertas con un solo tipo de compuerta.

Empecemos con un caso muy sencillo: implementar el NOT con la compuerta NAND. Primero que nada, repasemos sus tablas de verdad.

NOT

INPUT	OUTPUT
A	
0	1
1	0

NAND

INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

Lo que queremos lograr es que de un solo input x, salga un output que sea $\neg x$. (Donde x representa un bit que puede ser 0 o 1).

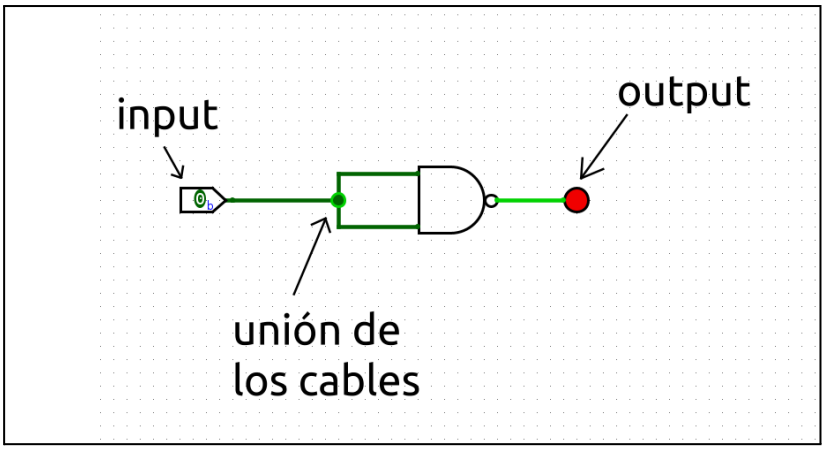
Primer problema: la compuerta NAND tiene dos inputs. Solución: al mismo input le conectamos otro cable (literalmente), y tenemos dos inputs iguales.

¿Con eso estamos? En este caso, que es muy sencillo, sí. Porque al ver la tabla de NAND, vemos que cuando hay dos inputs iguales (ver los renglones resaltados con verde), tenemos el output inverso.

NAND

INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

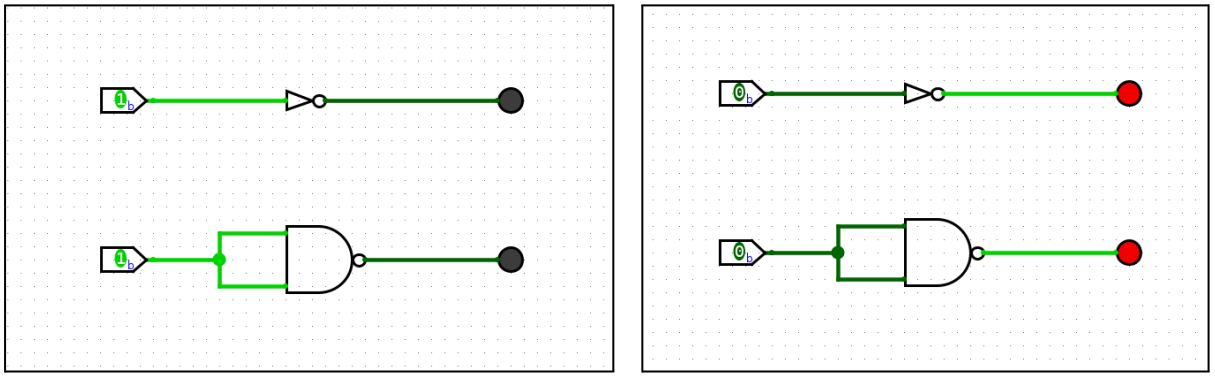
¿Pero cómo era eso de andar conectando cables? La representación gráfica puede variar un poco según el programa utilizado (o el dibujante), pero básicamente nos queda una cosa así:



Eso es el NOT implementado con una compuerta NAND. En la imagen anterior, el input es un bit (que puede ser 0 o 1, y en ese caso está en 0), y el output está representado por un led rojo, que si está prendido es 1 y si está apagado es 0.

¿Cómo podemos corroborar que este nuevo circuito es equivalente al NOT? Lo sabemos porque aplicamos correctamente el álgebra de boole. Pero también podemos saberlo de una manera menos ortodoxa: comparando el funcionamiento de los circuitos. Obviamente, si fueran circuitos complejos, es muy trabajoso, pero en este caso es sencillo.

La imagen de la izquierda muestra (en la parte de arriba) una compuerta NOT con input 1 y output 0 (representado con un led apagado); en la parte de abajo, vemos una compuerta NAND con dos inputs 1, y output 0. La imagen de la derecha lo mismo, pero el input es 0. Como podemos observar, cuando los inputs son iguales, los outputs también. Eso significa que los circuitos son equivalentes.

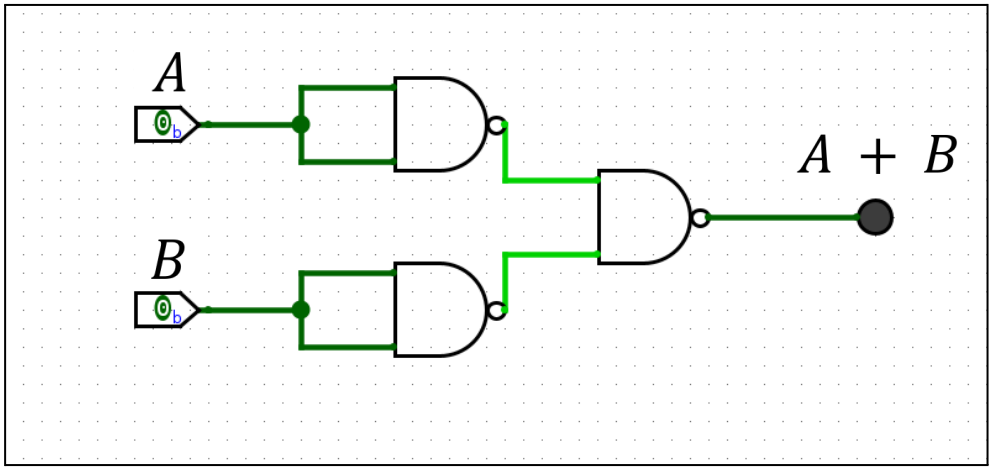


Pasemos a un caso más interesante: un OR a partir de NAND. Para esto conviene usar las identidades del álgebra booleana. La idea es pasar de $A + B = F$ a algo equivalente, pero todo escrito con NAND. ¿Y qué es NAND? Es un AND negado. Esto nos da la pista: la cosa va por la doble negación, y De Morgan.

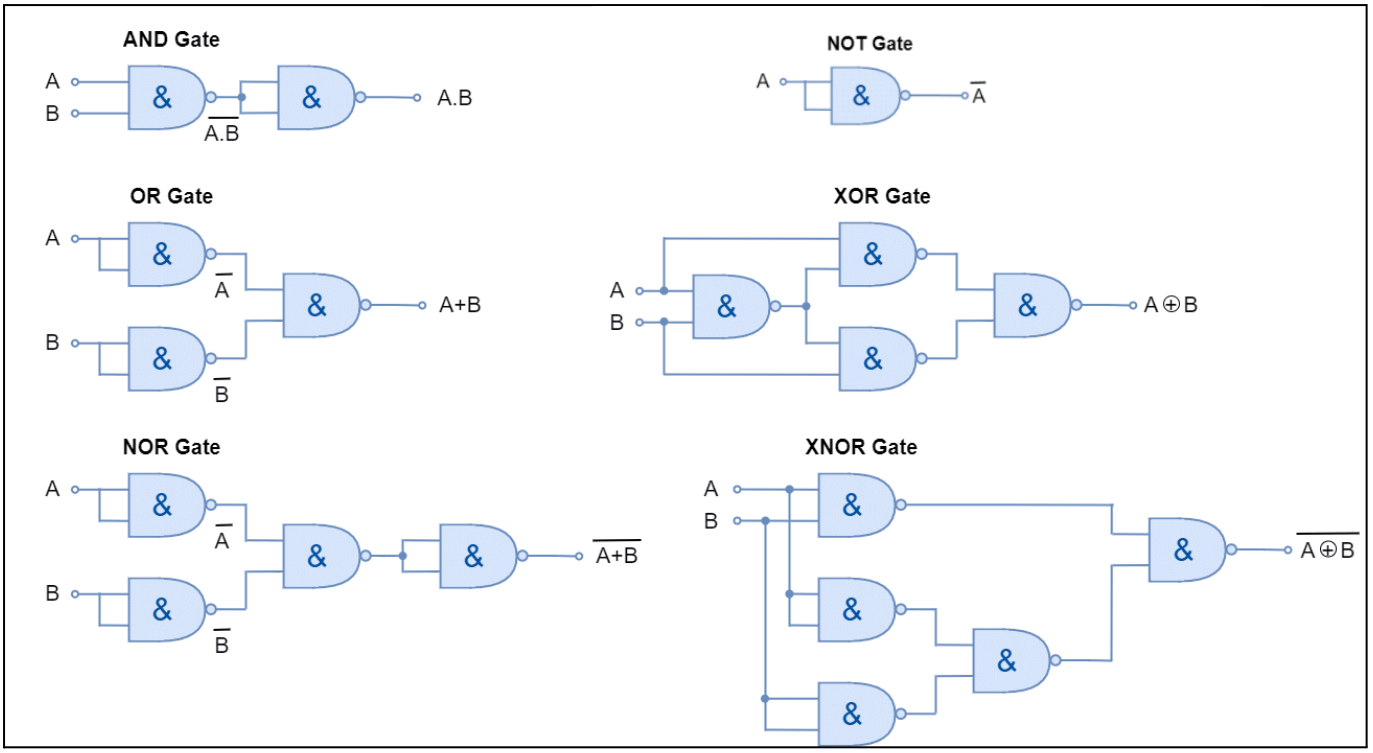
Veamos a dónde llegamos: $A + B = (\overline{\overline{A + B}})$ por la doble negación.

$(\overline{\overline{A + B}}) = (\overline{\overline{A} \cdot \overline{B}})$ por el viejo y peludo De Morgan.

Y listo: llegamos a una expresión hecha con dos NOT (estos son NOT A y NOT B), y un NAND, a saber: (NOT A) NAND (NOT B). Y el NOT, como vimos recién, lo podemos expresar con NAND. Así que pasando todo a NAND queda así: (A NAND A) NAND (B NAND B). Gráficamente sería:



Siguiendo esta lógica, todo puede escribirse con NAND. Y así llegamos a esto:



Ahora bien, si queremos construir una función booleana, ¿por dónde empezamos? Lo mejor sería definir bien cuántos inputs hay, y cómo queremos que sea el output. Una vez definido esto, lo plasmamos en una tabla de verdad. ¿Y después? ¿Cómo pasamos de eso a su expresión algebraica? ¿Y a su gráfica? Para eso, hay un par de técnicas. Empecemos por la más fácil, llamada **suma de productos**.

Una vez que tenemos la tabla de verdad, los pasos son los siguientes:

1) escribir el AND (con *todos* sus inputs) para cada renglón donde el output es 1,
2) unir todas esas ANDs a una OR final.

Veamos esto con un ejemplo concreto. Supongamos que queremos implementar el circuito de una función booleana M, de tres entradas A, B y C, tal que M = 1 cuando la cantidad de unos en A, B y C es mayor o igual a dos, y M = 0 en cualquier otro caso. La tabla nos queda así:

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	**1**	**1**	**1**

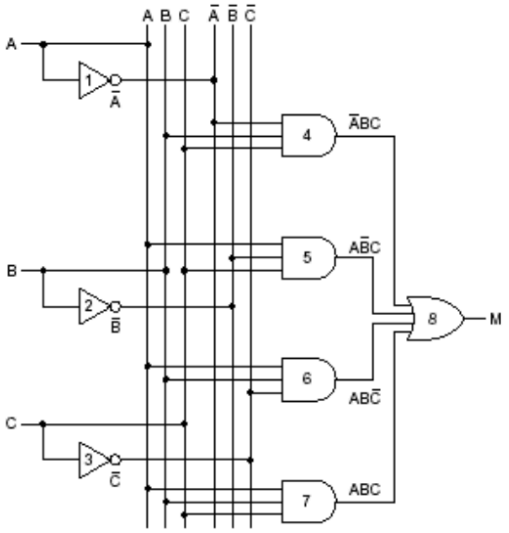
1	0	0	0
1	**0**	**1**	**1**
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Hecha la tabla, aplicamos el procedimiento de la suma de productos.

Primer paso: como hay cuatro unos, tendremos cuatro ANDs: $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ y ABC.

Segundo paso: juntar todo eso en una OR, así: $\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = M$.

Y listo. Esa es la expresión algebraica final, la cual puede graficarse de la siguiente manera:



Ahora bien, ¿podemos simplificar esa expresión, o esa es la única manera de escribirla? Muchas veces, este tipo de expresiones pueden simplificarse, y este es uno de esos casos. Podemos hacerlo utilizando las identidades y/o leyes y/o axiomas del álgebra booleana.

Antes de ir a la formula, veamos un truquito clásico. (A veces se lo encuentra con el nombre de "Ley de Distribución" o "Ley de Absorción", pero es confuso porque hay otras identidades con ese nombre, así que lo voy a llamar así: truquito).

Cuanto tenemos algo tipo $AB + A\overline{B}$ podemos aplicar la distributiva, para llegar a $A(B + \overline{B})$, pero por la ley del complemento (también llamada "inversa") sabemos que $B + \overline{B} = 1$. Entonces $A(B + \overline{B}) = A(1)$. Y como el 1 es el elemento neutro de la operación $\cdot$, tenemos que $A(1) = A$.

Es decir que $AB + A\overline{B} = A$.

Si lo pensamos dos segundos, tiene toda la lógica del mundo.

Volviendo a nuestra expresión, tenemos $\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$. Si miramos con atención, podemos aplicar el truco repetidas veces entre los tres primeros miembros y el último, de esta manera:

$\overline{A}BC + ABC = BC$

$A\overline{B}C + ABC = AC$

$AB\overline{C} + ABC = AB$

Para eso, si queremos ser rigurosos y prolijos, tenemos que "hacer copias" del último miembro y moverlas a donde corresponde:

$\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

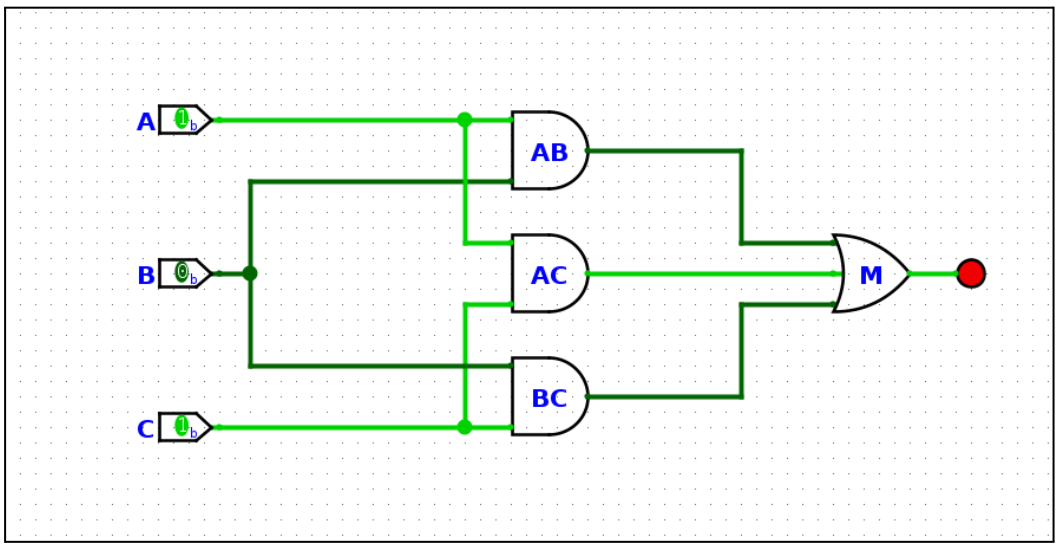
$= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC + ABC + ABC$ →por la ley de idempotencia

$= \overline{A}BC + ABC + A\overline{B}C + ABC + AB\overline{C} + ABC$ →por la conmutatividad de la operación +

$= BC + AC + AB$ →por el truquito

$= AB + AC + BC$ →por la conmutatividad de la operación +

O sea que $\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = AB + AC + BC = M$. Con lo cual, podemos graficarlo de la siguiente manera:



¿Puede simplificarse aún más? ¿Cómo sabemos cuándo termina la simplificación? Eso depende un poco de lo que consideremos "expresión simplificada". La expresión de recién puede considerarse simplificada, aunque podría simplificarse aún más. Lo importante es que usando bien el álgebra, deberíamos quedarnos tranquilos: con eso ya es suficiente. Aunque,

por supuesto, existen algunas técnicas específicas para esto. Los curiosos podrán encontrarlas en las págs. 375-377 de *Matemáticas discretas* de Lipschutz y Lipson (3ra edición), así como en *Computer Organization and Architecture* de Stallings (10th edition), págs. 380-387.

Último punto. Habíamos dicho que había un par de métodos para construir una función booleana. Uno era, como vimos, la suma de productos, y el otro es un pariente suyo: el **producto de sumas**. Aunque no está muy bueno, capaz es más conveniente que el otro cuando hay muy pocas salidas en 0.

La suma de productos se basa en que el output es 1 si cualquiera de las combinaciones de inputs que produce 1 es verdadera; es decir, si *al menos una* de esas combinaciones es verdadera. Por eso la compuerta final es un OR. El producto de sumas, en cambio, saca provecho de que el output es 1 si ninguna de las combinaciones de inputs que producen 0 es verdadera; es decir, si *todas* las combinaciones de inputs que producen 0 son falsas. Por eso la compuerta final es un AND.

Tomemos el mismo ejemplo de antes: queremos pasar de la siguiente tabla a una función booleana.

A	B	C	M
0	**0**	**0**	**0**
0	**0**	**1**	**0**
0	**1**	**0**	**0**
0	1	1	1
1	**0**	**0**	**0**
1	0	1	1
1	1	0	1
1	1	1	1

¿Qué hacemos? En vez de mirar los renglones con salida 1, miramos los que tienen salida 0. Y escribimos: $\overline{(\overline{A} \cdot \overline{B} \cdot \overline{C})} \cdot \overline{(\overline{A} \cdot \overline{B} \cdot C)} \cdot \overline{(\overline{A} \cdot B \cdot \overline{C})} \cdot \overline{(A \cdot \overline{B} \cdot \overline{C})}$. Y por De Morgan eso es igual a $(\overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}}) \cdot (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{\overline{A}} + \overline{B} + \overline{\overline{C}}) \cdot (\overline{A} + \overline{\overline{B}} + \overline{\overline{C}})$, y sacando las dobles negaciones llegamos a $(A + B + C) \cdot (A + B + \overline{C}) \cdot (A + \overline{B} + C) \cdot (\overline{A} + B + \overline{C})$.

Obviamente también podemos llegar directamente desde la tabla siguiendo estos pasos:

- seleccionar renglones con output 0,
- tomar los inputs en ceros como los inputs normales y los unos como los negados y expresarlos en una suma,
- hacer un producto de todas las sumas.

Pero nada, está bueno hacer alguna vez el proceso anterior usando De Morgan para entender de dónde viene la idea.

Pequeña digresión: las máscaras

La ALU, como su nombre lo indica, realiza operaciones aritméticas (como sumar y restar) y lógicas (como AND, OR y NOT). Aunque todavía no le encontremos mucho sentido, al aplicar operaciones lógicas sobre ciertas cadenas de bits con otras cadenas de bits especiales llamadas máscaras, nos permite transformar las cadenas originales, y lograr así resultados que pueden sernos de alguna utilidad.

Tratemos primero de encontrarle un sentido a todo esto. Supongamos que estamos trabajando al nivel de la ALU, con números en 8 bits y en BSS. Supongamos, también, que en determinado momento queremos saber si el número que tenemos ahí es par o impar. ¿Cómo hacemos? Acá no estamos en Pascal, no podemos hacer *if ((num MOD 2) = 0) then* ni nada por el estilo. Tenemos un número, que en realidad es una cadena binaria como 00100101, y queremos saber si es par o impar. ¿Qué recursos nos da la ALU? Los flags. Ellos son los indicadores que se activan después de cualquier operación, sea aritmética o lógica. ¿Y cómo nos pueden ayudar en este caso? Nos pueden dar la información que queremos, si aplicamos correctamente la máscara que necesitamos. O sea, tendremos que hacer una operación lógica entre dos cadenas de bits, y tratar de que algún flag nos diga si ese número es par o impar. ¿Cómo llegamos a eso?

Primer problema: ¿qué me indica si un número es par o impar (en BSS)? El bit menos significativo: si es 0 es par, si no, es impar.

Segundo: ¿cómo puedo conocer solamente ese último bit? Pasando a cero todos lo demás, y dejando ese bit menos significativo como estaba.

Tercero: ¿cómo sabe la ALU si era 1 o 0? Mirando el flag zero: si Z quedó encendido, entonces el último bit era cero, y por lo tanto el número era par; si Z quedó apagado, el bit era uno, y por lo tanto era impar.

En concreto, si queremos hacer esto con el 00100101 tenemos que hacer

00100101 → el bit rojo es el único que nos interesa conocer
AND 00000001 → AND, pues los 0s fuerzan a 0 el resultado, y el 1 deja pasar lo que había
00000001 → Z = 0 (ya que el resultado no son todos ceros)

Lo único que queda por hacer es preguntar si Z = 0 o no, pero eso ya excede el presente tema (lo veremos cuando estudiemos el assembly language). La idea de esto era ilustrar un poco la verdadera utilidad de las máscaras, siendo esta una de las muchas aplicaciones que existen.

Ahora, ¿cómo sé cuándo usar cada máscara? ¿Qué hace cada una? Eso proviene de la tabla de verdad. Si miramos con atención la tabla de verdad de cada una de las operaciones que existen (como AND, OR, NOT, etc.), notaremos para qué sirve cada una. Acá van nomás algunos recordatorios clave:

- Con AND, podemos forzar a 0 con 0, y dejar pasar con 1
- Con OR, podemos forzar a 1 con 1, y dejar pasar con 0

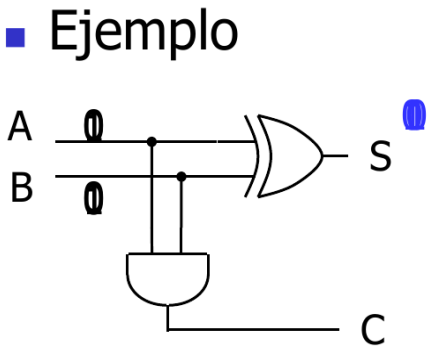
- Con XOR, podemos invertir el bit con 1, y dejar pasar con 0
- Con XNOR, invertimos el bit con 0, y dejamos pasar con 1
- Con NAND, podemos forzar a 1 con 0, e invertir con 1
- Con NOR, podemos forzar a 0 con 1, e invertir con 0

Circuitos combinatorios o combinacionales

La introducción anterior a la lógica digital es clave para entender bien los circuitos combinatorios, o sea, circuitos que responden a los valores lógicos en las entradas, es decir, donde la salida está determinada exclusivamente por los valores de las entradas en ese instante. Aquí no hay todavía un análisis temporal, es solamente un análisis lógico. Otra historia será el asunto de los circuitos secuenciales, pero eso viene después. En los circuitos combinatorios, si cambia la entrada, cambia la salida, y los valores pasados de las entradas no influyen en los valores de las salidas.

En verdad, ya vimos algunos ejemplos de circuitos combinatorios, pero ninguno con una utilidad práctica. Veamos ahora uno muy sencillo, pero que sirve para algo: el semi-sumador, o **half adder**, que sirve para sumar dos bits. Se trata de un circuito con dos inputs y dos outputs: los dos inputs (A y B) son los bits que se van a sumar, un output es el resultado de la suma (S), y el otro output es el carry (C).

La gráfica y la tabla serían así:

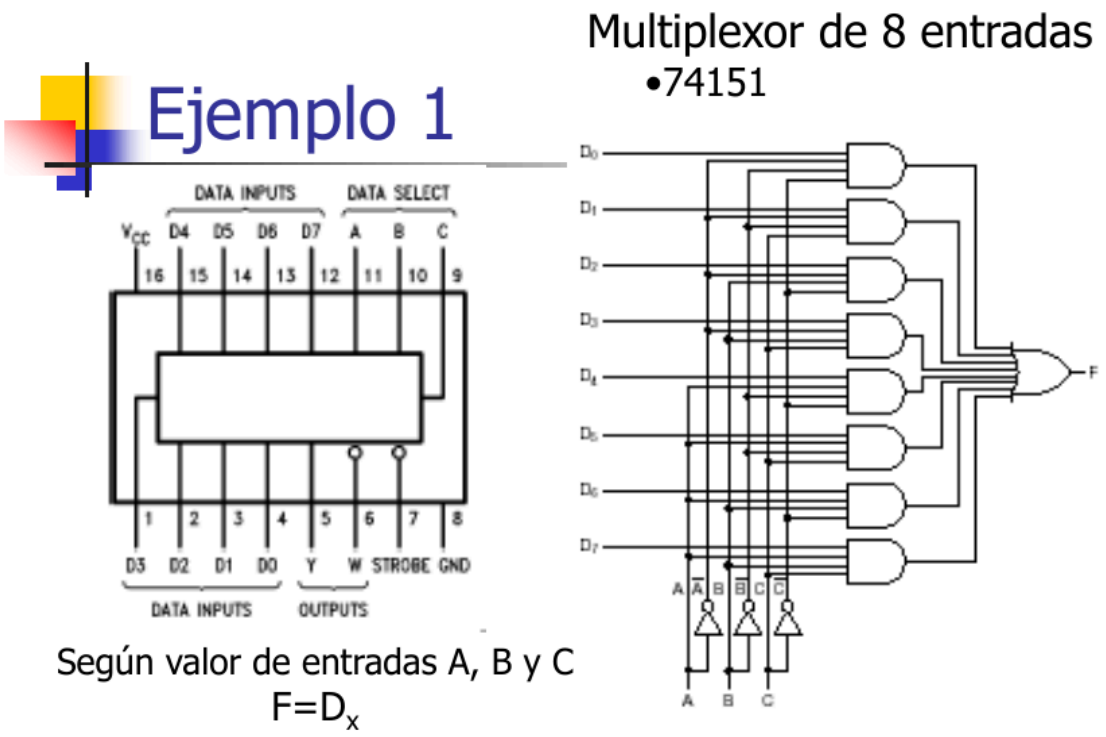


A B	S	C
0 0	0	0
0 1	1	0
1 0	1	0
1 1	0	1

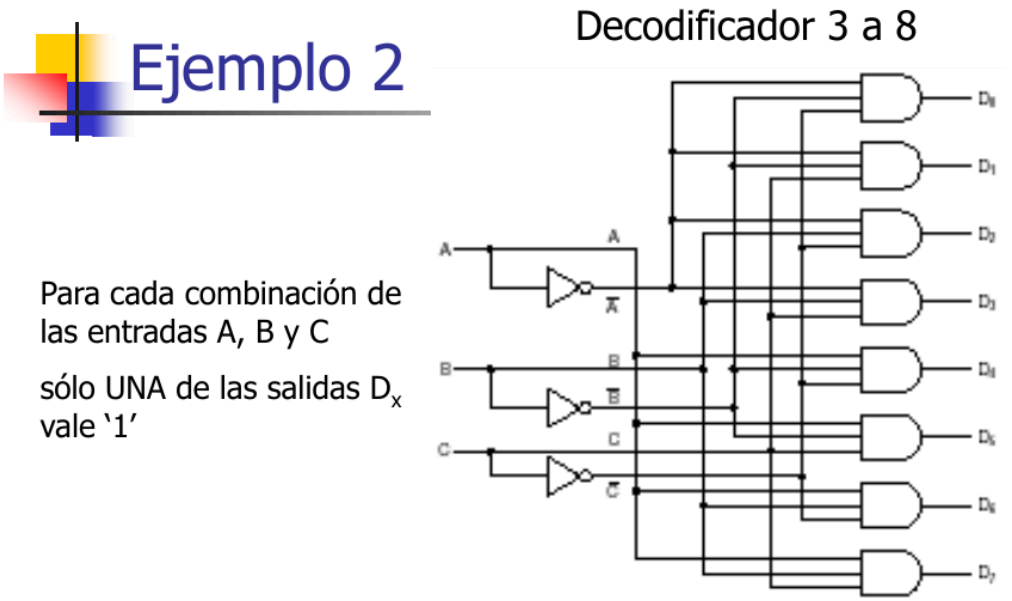
S representa la suma aritmética de 2 bits y **C** es el acarreo

Otro ejemplo clásico: el **multiplexor** (o mux). Este circuito es básicamente un selector de canales, por eso tiene una buena analogía con el televisor. Pensemos: al televisor le llegan por el aire o por el cable muchos canales, pero solo muestra de a uno por vez; eso significa que tiene que elegir cuál de todos los canales que recibe va a mostrar, o, en otras palabras, cuál canal de datos va a dejar pasar. Acá pasa lo mismo: al multiplexor le llegan varios canales de datos, pero deja pasar uno solo, dependiendo de la salida que seleccionemos.

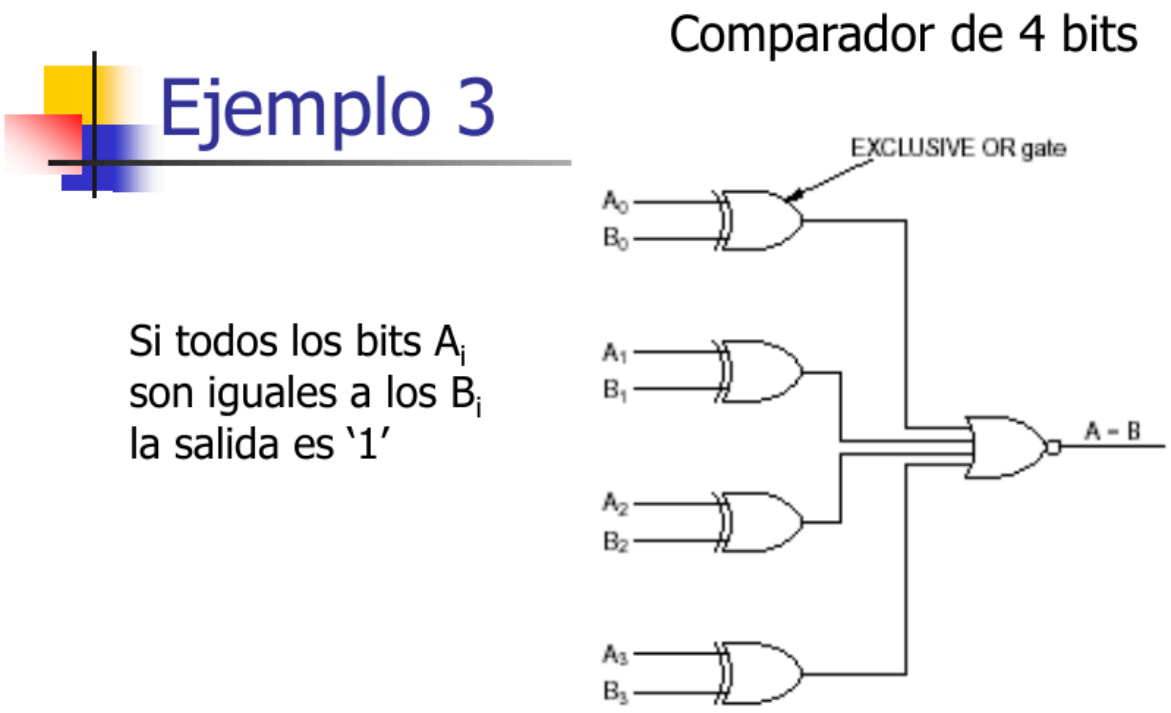
Veámoslo con un caso concreto: un multiplexor de 8 entradas (de datos). Este multiplexor tiene 8 entradas de datos (desde D_0 a D_7), y tres entradas para seleccionar el canal que nos interesa (A, B y C). En total, pues, tenemos 11 entradas, y una salida. ¿Pero por qué 3 para seleccionar? Porque con 3 bits de entrada, tenemos 8 combinaciones posibles, o sea, 2^3, y dependiendo de estas tres entradas, se activará una, y solo una, de las 8 compuertas que permiten el paso del canal de datos seleccionado. La gráfica sería así:



Otro clásico: el **binary decoder**. Es como tener solamente la parte que selecciona canales del anterior. En este circuito hay n entradas, y 2^n salidas, donde todas serán 0, salvo una, que será 1. Acá podemos ver un decodificador que, al tener tiene 3 entradas, tiene 8 salidas:



Otro caso: el **comparador**. Este sirve para saber si dos cadenas de bits son idénticas o no Si tenemos, por ejemplo, dos cadenas de 4 bits (representadas genéricamente como $A_3A_2A_1A_0$ y $B_3B_2B_1B_0$) el comparador tendrá que ser de 8 entradas (una para cada bit), y tendrá, como siempre, una salida, la cual indicará si se cumple que todos los pares de bits comparados coinciden o no (o sea, se comparará posición por posición: A_0 con B_0, A_1 con B_1, etc.). Una manera de implementar esto es con varias compuertas XOR y una NOR. A cada XOR le entra el par de bits a comparar. XOR será 1 si son distintos, y 0 si son iguales. Si las dos cadenas son iguales, entonces todos los pares de bits coinciden; y si todos los pares de bits coinciden, entonces a la compuerta final NOR llegan todos ceros, y NOR es verdadera únicamente en ese caso particular; por lo tanto, la salida del comparador será 1 cuando todos los bits de las dos cadenas sean iguales, y 0 en cualquier otro caso.



Otro caso: el **desplazador** (o shifter). Esto sirve para correr los bits para un lado o para el otro en una cadena de bits; algo parecido a lo que hacíamos con las mantisas para poder sumar dos números en punto flotante con distinto exponente.

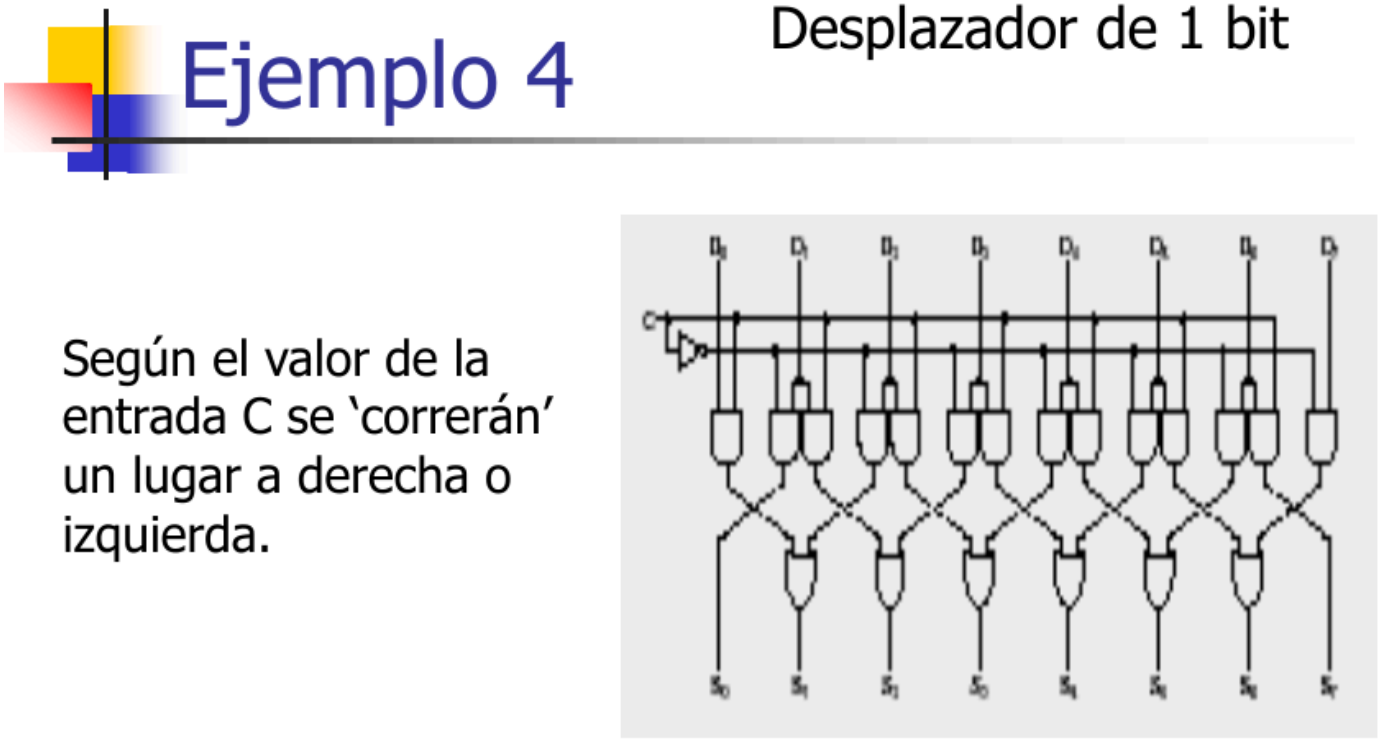
Veamos el caso concreto que propone la cátedra: el desplazador de 1 bit. Este circuito recibe una cadena de 8 bits y tiene como salida esos 8 bits, pero corridos o bien un bit a la derecha, o bien un bit a la izquierda. En ambos casos, un bit se pierde, y se agrega un bit en 0. Si desplazamos la cadena a la derecha, el bit más significativo quedará en 0, y se perderá el bit menos significativo; si lo desplazamos a la izquierda, perderemos el bit más significativo, y el bit menos significativo quedará en 0.

Ejemplo: dada la cadena 10011001, si la desplazamos a la derecha nos queda

010011001 →el bit en negrita es el nuevo; en rojo el que se pierde

01001100 → resultado final

Hay 8 entradas para los bits, y una para el selector de derecha o izquierda (representado con la letra C), o sea que tenemos 9 entradas en total. En este caso, C tiene que estar en 1 para desplazar la cadena a la derecha, y en 0 para la izquierda.



Último ejemplo: **un bit de ALU**. Describir todo este circuito es un quilombo, pero si uno presta atención, es como una combinación de circuitos anteriores. Lo importante es comprender la visión general: el circuito siempre trabaja al 100%, pero el output principal será uno solo, y estará definido por el *decoder*, que tiene 4 opciones, y solamente una de ellas será la que esté activada en cada momento dado. Así que, el output principal tiene 4 alternativas:

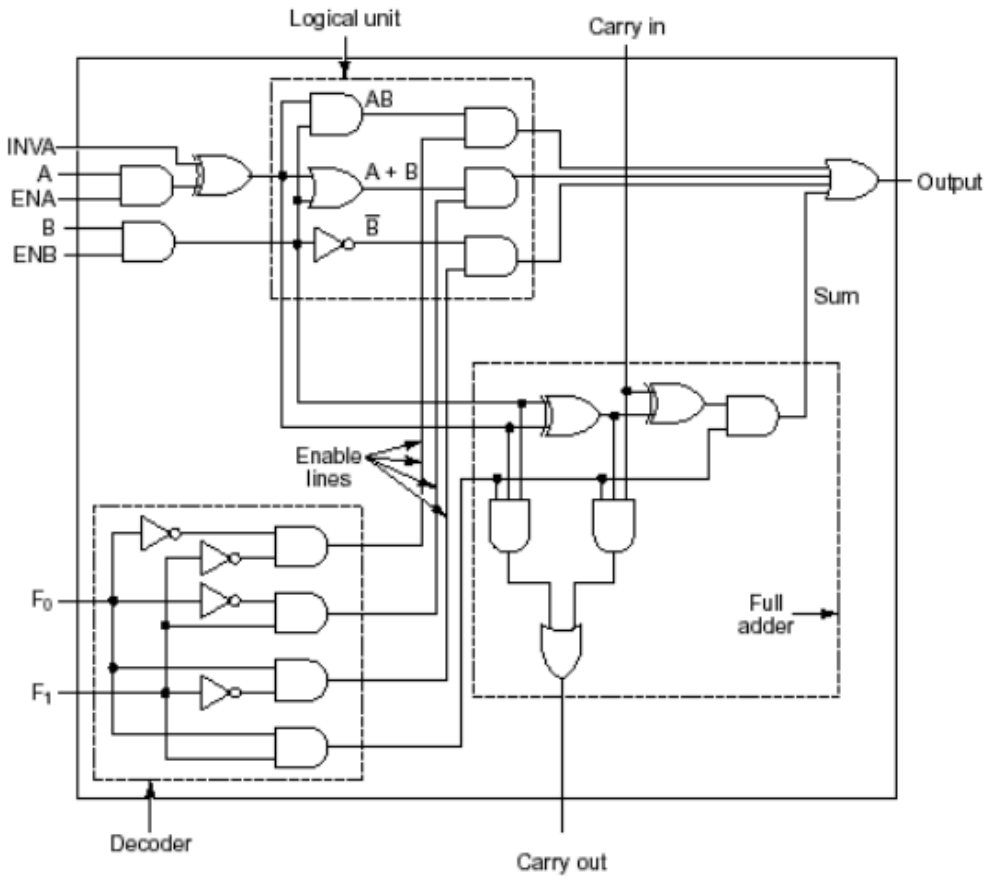
- o bien la salida es AB
- o bien $A + B$
- o bien $\overline{B}$
- o bien la suma del *full adder*.

Solamente una de esas opciones puede estar seleccionada para ser el output principal.

Digo el output *principal*, porque también hay otro output, que podríamos denominar "output secundario", que siempre sale: el *carry out*. El *carry out* de un bit de la ALU, es el *carry in* del siguiente bit de la ALU, excepto en los extremos: en el primer bit, el *carry in* está forzado a 0, y en el último, el *carry out* va al flag de carry.

Ejemplo 5

1 bit de ALU



Según F_1F_0 será la función que se realizará sobre A y B.

Circuitos secuenciales

Hasta ahora todo era felicidad y alegría. Las diferentes entradas a un circuito en cualquier momento dado determinaban una salida para ese circuito, y ya. No había mucho misterio; y no estaba permitida la realimentación. Pero siendo más realistas, la situación es más complicada. La electricidad no viaja a una velocidad instantánea por los cables; y, encima, las compuertas hacen que se retarde aún más. Esto significa que, si queremos ser bien conscientes de lo que sucede en los circuitos eléctricos, debemos incluir la dimensión temporal. Y eso es lo que hacen los circuitos secuenciales.

En este contexto, *las salidas dependen tanto de las entradas como del estado interno del circuito.* Y una ventaja que tenemos ahora, es que algunos de estos circuitos son capaces de "almacenar" valores lógicos internamente.

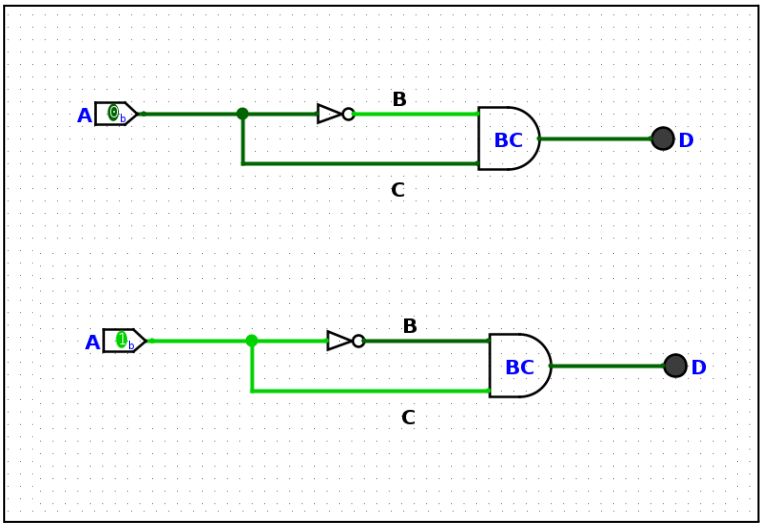
Pero primero, antes de meternos con la memoria, volvamos al asunto del tiempo y la electricidad un momento, y revisemos un "circuito" que quizás creíamos conocer bien: la contradicción (o al menos una de las implementaciones de la contradicción): el clásico $p \wedge \neg p$.

Si hacemos su tabla de verdad, da siempre 0.

p	$\neg p$	$p \wedge \neg p$
1	0	0
0	1	0

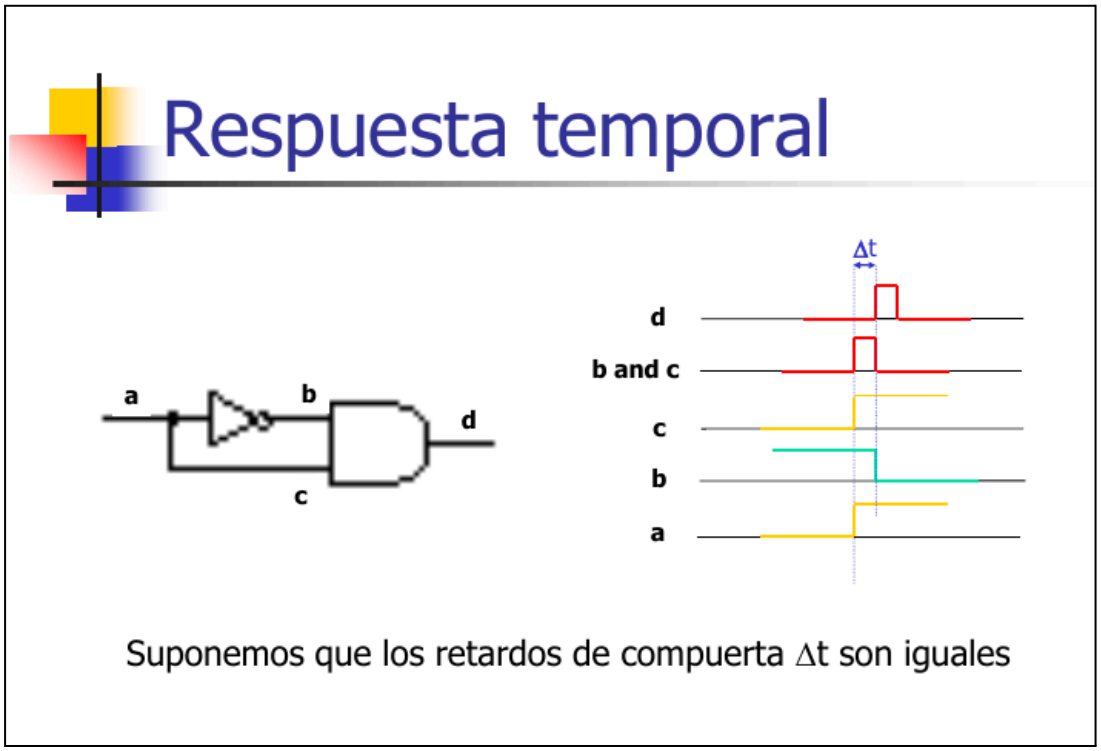
Esto, en términos lógicos, es completamente cierto y correcto. Ahora bien, una vez que construimos un circuito como ese, pasamos a la vida real. Y en la vida real pasan cosas.

Uno podría mirar el simulador y decir: mirá aunque lo ponga en 0 o en en 1 al input (A), la salida (D) siempre es 0.



Y la mayor parte del tiempo, esto es cierto. Pero no *todo* el tiempo. Si prestamos atención, una de las entradas del AND (la entrada B) pasa por un NOT, mientras que la otra (la C) viene directo por el cable. La compuerta NOT hace que, cuando A cambie de estado, el "mensaje" va a llegar primero por C que por B, ya que va directamente por el cable. Es decir que, aunque sea por unos nanosegundos (o incluso menos), cuando A cambie de estado, el AND va a estar recibiendo la señal vieja de A desde B y y la nueva de A desde C.

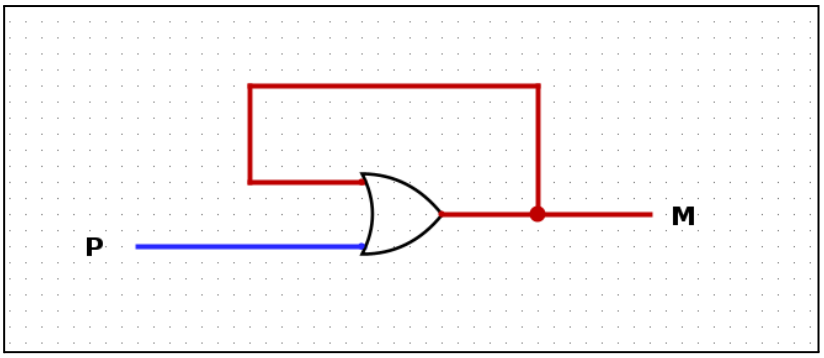
Cuando A pase de 1 a 0, no tendrá ningún efecto, ya que el retardo de B quedó en 0, y por lo tanto el output seguirá siendo 0. Pero cuando A pase de 0 a 1, es decir, cuando el retardo haga que B quede un momento en 1, ¡tendremos un AND con las dos entradas en 1! Esto es lo que muestra el siguiente gráfico:



Empero, no nos adentraremos tanto en el mundo de los electrones, los nanosegundos y toda esa wea. Con lo que sí vamos a trabajar en profundidad, es con cómo almacenar valores lógicos, y, más precisamente, con los flip-flops.

Entonces, ¿cómo se almacena un valor lógico? Para almacenar un bit (en 0 o en 1) por el tiempo que nos interese, podríamos intentarlo con una compuerta OR. Pero de una forma bastante particular: haciendo que una de las entradas sea, a su vez, la salida. Esto es inédito: en ningún circuito combinatorio una salida transportaba información hacia la entrada. (*Remember with me*: ¡en los circuitos combinatorios no existe la realimentación!).

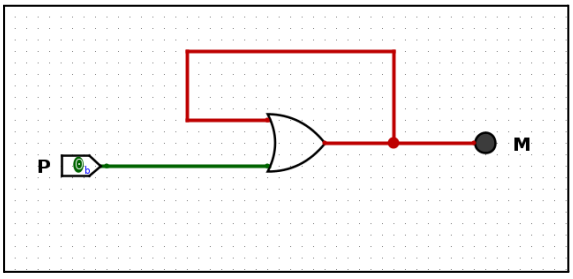
La ecuación lógica o la expresión algebraica sería así: $M = M + P$. Y la gráfica así:



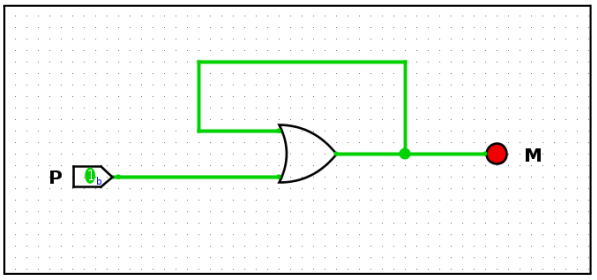
Asumiendo que la historia empieza con P en 0, la historia empieza bien, porque si P = 0, M también, ya que el resto del circuito está apagado, y por lo tanto, la otra entrada también será 0 (ver CUADRO 1). Y si queremos que M sea 1, pasamos P a 1, y listo M = 1 (ver CUADRO 2).

El tema es que una vez que entramos a M = 1, no podemos salir, porque aunque pongamos P en 0, M quedó con una de las entradas en 1 del estado anterior, y como estamos en una OR, con que una de las entradas sea 1 ya está, la salida será 1 (ver CUADRO 3).

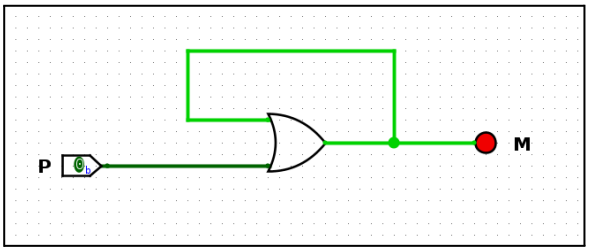
(CUADRO 1)



(CUADRO 2)



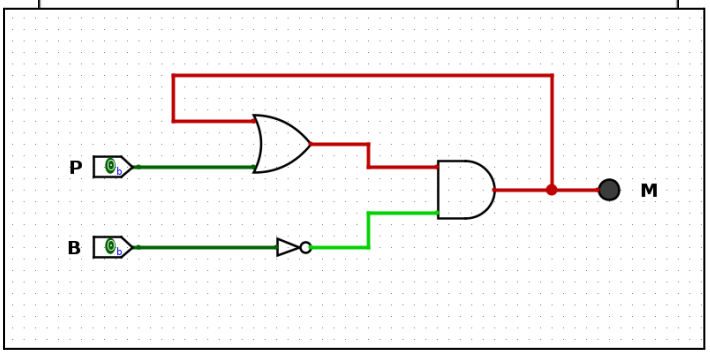
(CUADRO 3)



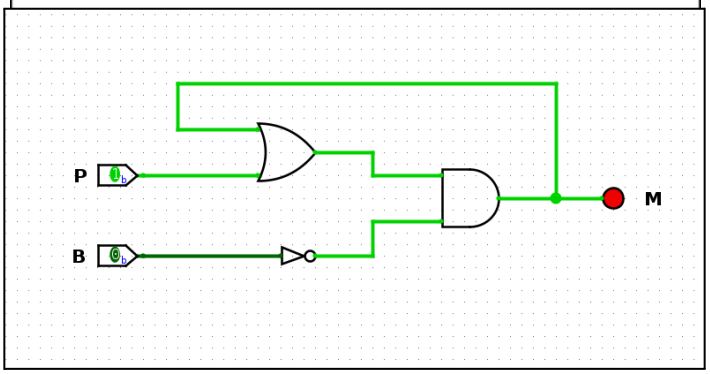
O sea que para volver a 0 habría que resetear todo. Desenchufar los cables y empezar de nuevo. O también, claro, podríamos mejorar el circuito. Acá va una segunda versión: $M = (M + P) \cdot \overline{B}$.

La principal diferencia es que ahora el OR no se retroalimenta a sí mismo directamente, sino que antes pasa por un AND. Y se agrega, además, otra entrada para poder controlar bien el AND. La gráfica y la secuencia de estados, empezando todo en 0, sería así:

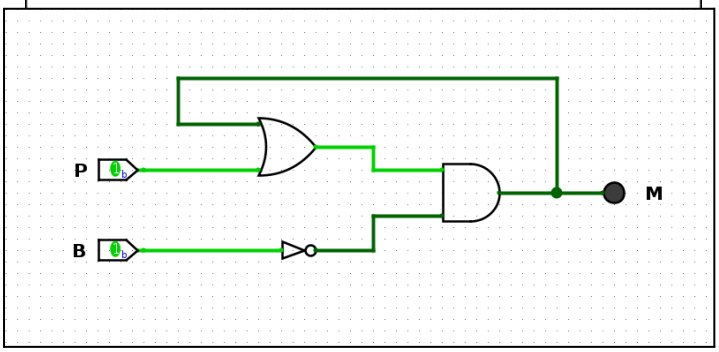
[1] Todo en 0, porque recién lo prendimos.
Más adelante, *no siempre* sucederá lo mismo



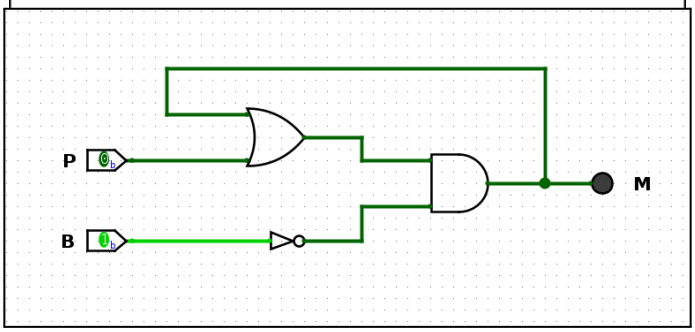
[2] Primer cambio: P = 1.
Con esto, prendemos el AND



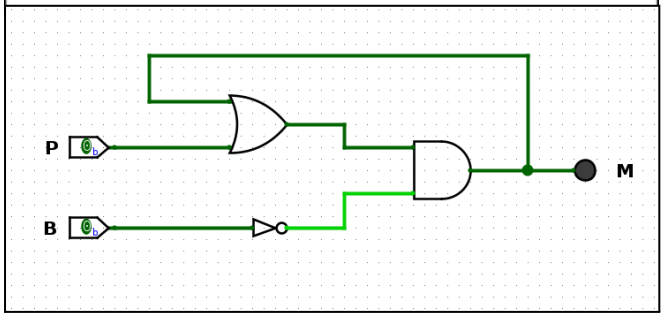
[3] P igual que antes. Ahora B = 1, y apagamos el AND



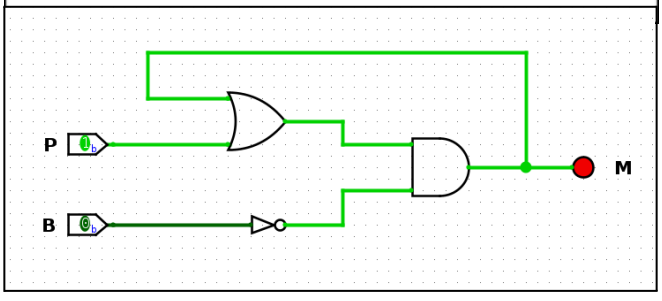
[4] Con B = 1, no importa cómo esté P: AND quedará en 0



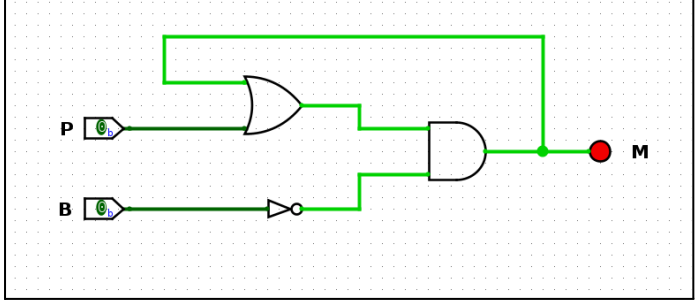
[5] Si venimos del estado anterior, apague o prenda B, el AND queda apagado



[6] PERO, si volvemos a prender el AND …



[7] y venimos del estado anterior, apague o prenda P, el AND queda en 1



Esto es algo que distingue a los circuitos secuenciales: el hecho de que tener el mismo input no asegura obtener el mismo output. Compare [5] y [7].

¿Por qué tantas vueltas con este circuito? Porque si lo pasamos todo a NOR, obtenemos lo siguiente:

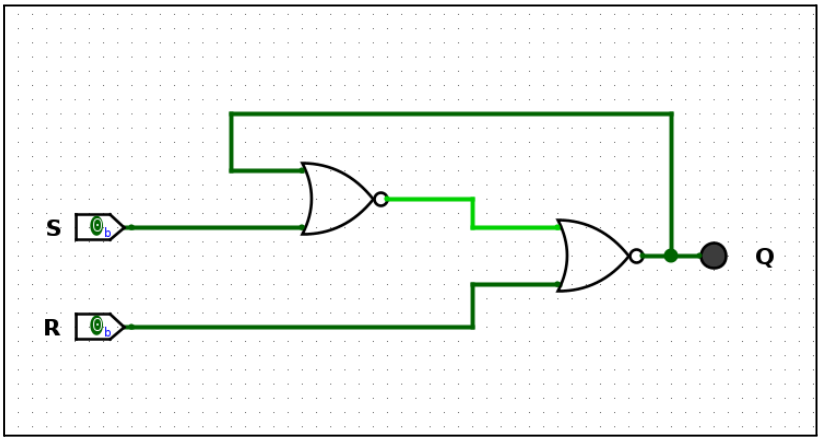
$M = (M + P) \cdot \overline{B}$

$= \overline{\overline{(M + P) \cdot \overline{B}}}$ → agregamos una doble negación a toda la expresión

$= \overline{\overline{(M + P)} + \overline{\overline{B}}}$ → aplicamos De Morgan

$= \overline{\overline{\overline{(M + P)} + B}}$ → simplificamos la doble negación de B

Y esto $\overline{\overline{(M + P)} + B}$, si cambiamos los nombres de las variables a $\overline{\overline{(Q + S)} + R}$, no es nada más, ni nada menos, ni nada más que… el flip-flop; o al menos una especie de borrador del flip-flop SR, ya que falta la salida $\overline{Q}$.



Como veremos en breve, el hecho de agregar esta nueva salida $\overline{Q}$, hace que una de las combinaciones (la combinación S = 1 y R = 1) quede prohibida. Ya veremos por qué.

Flip-flops hay muchos, pero empezaremos por analizar la versión más básica: el **flip-flop SR**; o FF SR, para abreviar. Este tiene dos entradas (llamadas S por set (poner a 1), y R por reset (poner a 0)), y dos salidas: Q y su complemento $\overline{Q}$.

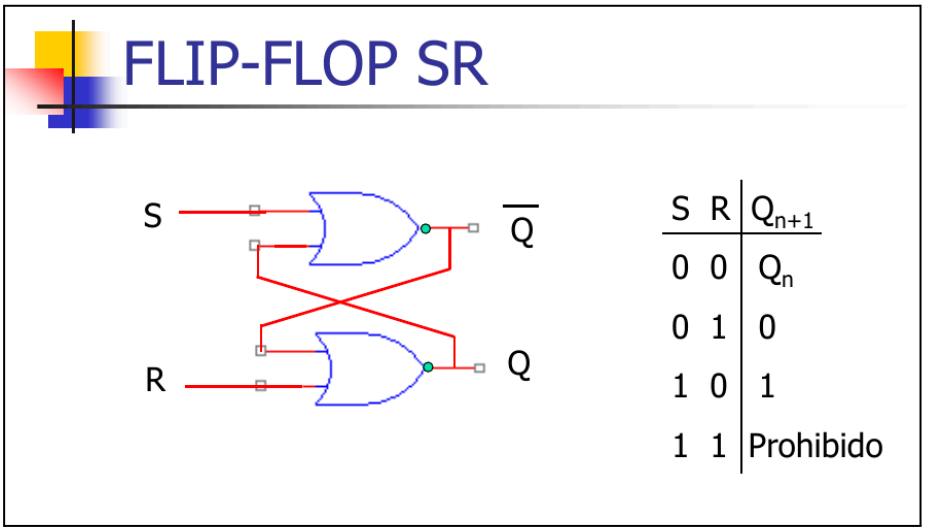
Si hiciéramos una tabla de verdad, podríamos pensar que, tal como sucedía antes con los circuitos combinacionales, la tabla debería tener 2^2 renglones, porque tiene dos entradas. Pero si hacemos una tabla de verdad que contemple también los estados anteriores, la tabla nos quedará de 2^3 renglones, ya que se agrega otra columna con los valores de la salida anterior, los cuales pueden alterar el resultado final. Para poder representar dos estados temporales distintos, se agrega una nueva notación: el subíndice para Q. Así, Q_n viene a ser la salida anterior, y Q_{n+1} la siguiente.

La tabla completa del FF SR es la siguiente:

S	R	Q_n	Q_{n+1}	
0	0	0	0	} Aquí la salida anterior es igual a la siguiente
0	0	1	1	
0	1	0	0	} No importa la salida anterior, la siguiente será 0
0	1	1	0	
1	0	0	1	} No importa la salida anterior, la siguiente será 1
1	0	1	1	
1	1	⊥	⊥	} Inputs prohibidos
1	1	⊥	⊥	

(Stallings (10th edition), págs. 396-398 lo explica todo esto mucho muy bien, ya que agrega un diagrama temporal, y le da mayor detalle aún a la explicación).

La tabla compacta, junto con su gráfico, queda así:



Ahora, ¿por qué nos queda una combinación prohibida, si antes cuando estábamos armando el circuito con M, P y B no había problema? Porque antes había una sola salida, en cambio ahora hay dos, que están definidas de una forma bastante particular; recordemos que se llaman Q y $\overline{Q}$. Y el problema es que si le ingresa S = 1 y R = 1, los dos outputs Q y $\overline{Q}$ serán iguales (quedando ambos en 0) y no estarán, por lo tanto, cumpliendo con la definición de que uno es el complemento del otro. Estaríamos llegando, por así decir, a un "circuito absurdo".

Entonces, ¿para qué me sirve un flip-flop? Básicamente, para almacenar un bit. Vale decir, pues, que *un flip-flop es una memoria de un bit*.

Cabe aclarar, de paso, que el flip-flop también se lo conoce como "biestable" porque el circuito posee sólo 2 estados posibles de funcionamiento, y se queda siempre en alguno de esos dos, salvo que las entradas provoquen un cambio.

Pero el flip-flop SR es simplemente el más básico. En realidad hay muchos flip-flops. Y **según la manera en que las salidas respondan a las señales lógicas presentes en la entrada, los biestables se clasifican en: SR, J-K, D y T.**

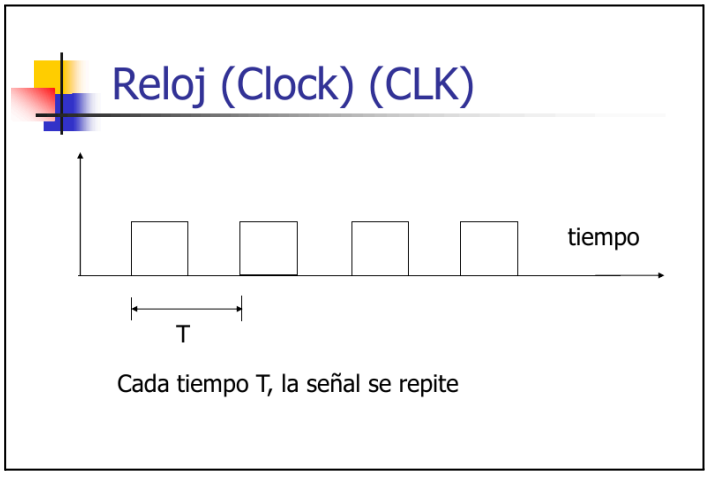
Respecto del instante en que pueden cambiar dichas salidas, los flip-flops pueden ser:

- Asincrónicos: cuando en la entrada se establece una combinación, las salidas cambiarán.
- Sincrónicos: la presencia de una entrada especial, determina "cuándo" cambian las salidas acorde a las entradas.

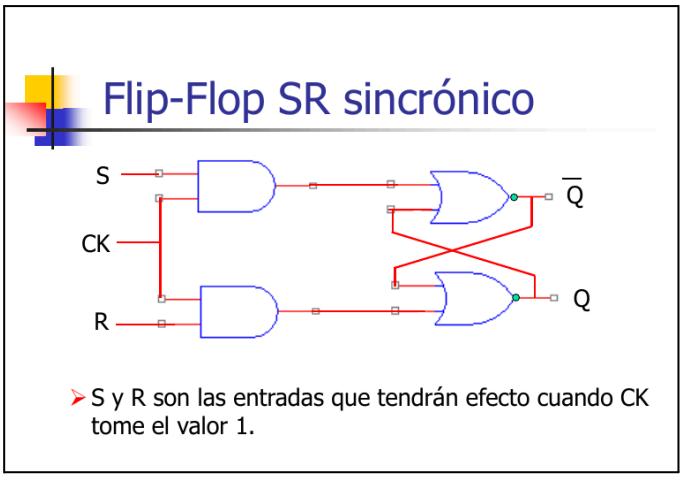
Si bien cada uno de los FFs puede implementarse de ambas formas, por convención, solamente solemos distinguir entre los SR asincrónicos y los sincrónicos, mientras que los demás (J-K, D y T) los veremos siempre en su forma sincrónica.

Volviendo a la definición anterior, aunque esa "entrada especial" puede ser varias cosas, normalmente se trata de un reloj. O sea que, por lo general, cuando decimos que es asincrónico es porque no tiene reloj, y sincrónico es porque tiene reloj.

¿Para qué queremos un reloj? Porque el orden en que ocurren los sucesos es importante. A veces los sucesos deben ocurrir simultaneamente, y para ello es fundamental contar con algo que coordine entre los distintos elementos de un sistema. ¿Pero cómo está hecho? ¿Vemos algo tipo "son las 15:05" en algún lado? No. Este reloj se parece más a algo que hace tic-tac todo el tiempo, pero nunca dice —ni sabe— la hora. ("Metrónomo" le sentaría mejor). El reloj es una señal de tiempo precisa que determina cuándo se producen los eventos. Básicamente, es un pulso electromagnético que oscila de forma uniforme, o sea, que genera flancos de subida y de bajada con intervalos iguales a lo largo del tiempo. Entre cada flanco de subida, o entre cada flanco de bajada (eso depende de cómo querramos interpretarlo), tenemos un ciclo del reloj. En la figura de abajo, viene a ser lo que está marcado con una T.



Armados con este nuevo elemento, podríamos construir un nuevo flip-flop que solo acepte las señales de entrada cuando el reloj lo permita. Así llegamos al



La tabla, obviamente, es igual que antes cuando el reloj permite el paso a las entradas (lo cual está representado con CK en 1), y todo se mantiene sin cambios cuando el reloj no permite que lleguen nuevas señales (cuando CK está en 0), ya que, naturalmente, si no cambian las entradas, no cambia la salida.

CK	S	R	Q_{n+1}
1	0	0	Q_n
1	0	1	0
1	1	0	1
1	1	1	Prohibido
0	x	x	Q_n

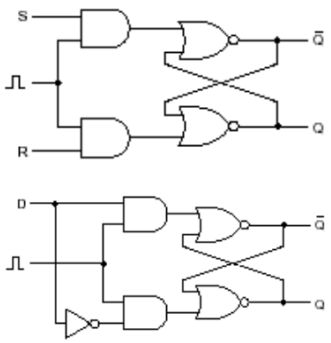
Esta versión ya es un poco mejor, pero todavía está el problema de los inputs prohibidos.

Una posible solución, es la implementada por el **flip-flop D**, donde en vez de tener dos entradas independientes (S y R), hay una sola entrada (D) que tiene su complemento ($\overline{D}$). O sea que los inputs siempre serán distintos. Con lo cual, no tenemos más la opción prohibida 1-1, pero tampoco la que dejaba todo como estaba (la 0-0).

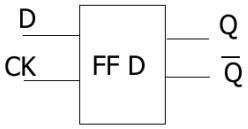
En resumen, con D = 0, la salida será 0, y con D = 1, será 1.

Flip-Flop D

- En el FF SR hay que aplicar 2 entradas diferentes para cambiar de estado.
- El FF D permite aplicar una sola entrada para cambiar la salida.



Flip-Flop D

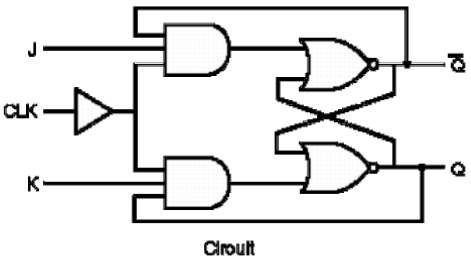


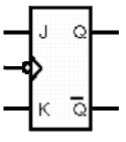
D	Q_{n+1}
0	0
1	1

con CK=1

Puede que el flip-flop D sirva para algo, pero se extraña la vieja combinación 0-0. Así que para poder tener una versión mejorada del SR original, llegó el **flip-flop J-K**. Es igual al SR (así que volvemos a tener la posibilidad de ingresar 0-0), con la excepción de que se añaden dos cables que van desde las salidas, a las ANDs de las entradas.

Flip Flop J-K

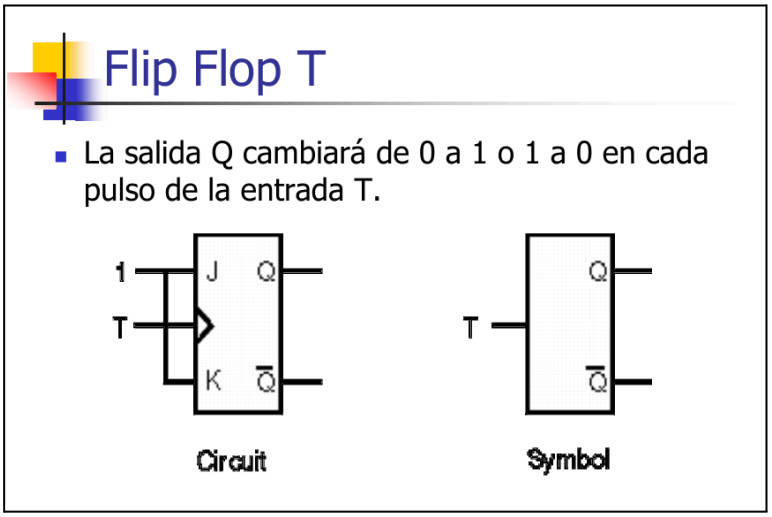




J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

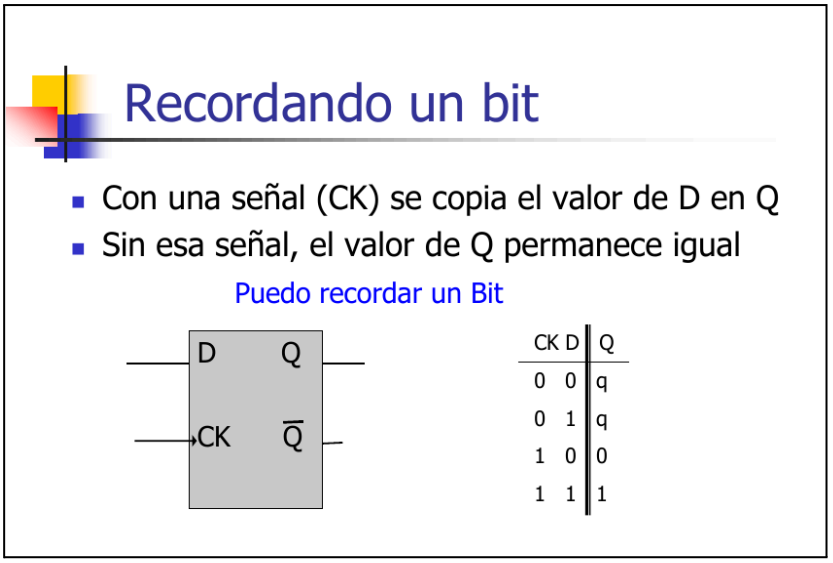
Esto posibilita manejar el problemático input 1-1. Aunque, en verdad, lo que se hizo fue eliminarlo o transformarlo de forma elegante, ya que las ANDs de la entrada (gracias a sus nuevos cables que vienen desde Q y $\overline{Q}$) ofician como agentes de seguridad que no permiten que ingresen dos unos al mismo tiempo. ¿Cómo lo logran? Pensemos: aunque desde afuera (desde J y K) lleguen las señales 1-1, gracias a los cables extras de las ANDs solamente una de las dos ANDs tendrá todo en 1 y emitirá un 1 (ya que *o bien* Q *o bien* $\overline{Q}$ estará en 1). O sea que, en verdad, el input que le llegará a las NOR será *o bien* 1-0, *o bien* 0-1. Es decir que, necesariamente, se producirá o bien un set o bien un reset. ¿Y cómo sabemos cuál va a suceder? Gracias a la conexión de los cables, sabemos que la nueva salida será el complemento del estado anterior. Así que si estamos en $Q = 0$ y nos llega un 1-1, tendremos un set y la cosa pasará a $Q = 1$. Y viceversa: si estamos en $Q = 1$ y nos llega un 1-1, tendremos un reset y la cosa pasará a $Q = 0$. Eso es lo que se muestra en la nueva tabla de la imagen anterior.

Por último, tenemos el **flip-flop T** (cuya T viene de *toggle*, una palabra con varios significados, aunque en este contexto lo mejor sería traducirlo como switch o alternador). Este flip-flop no para de *flip-flopear* a cada rato. Básicamente sirve para alternar entre 0 y 1 sin parar. Podemos pensarlo como un J-K cuyas entradas siempre son 1-1. Por eso cuando se activa con el reloj (o, mejor dicho, cuando llega un pulso del reloj), su salida pasa de 0 a 1, o de 1 a 0. Su salida se invierte a cada pulso del reloj.



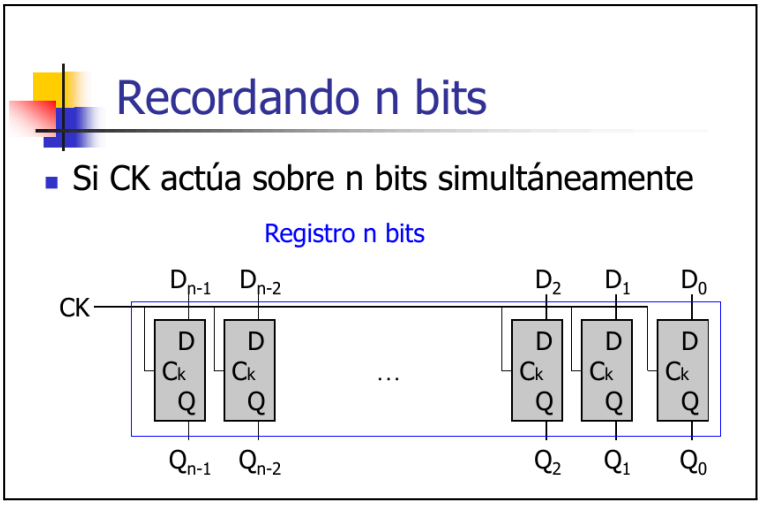
Volvamos ahora al asunto de la memoria. ¿Cómo podemos recordar un bit? Hay varias maneras de hacerlo, pero aquí veremos una sencilla: usando el flip-flop D; uno que parecía ser de lo más inútil, al final resultó alto capo, como el Neville.

Tal como vimos, en este flip-flop el input es igual al output, si el reloj está activado, si no, se queda como está. Sacando provecho de esta lógica, tenemos una memoria de un bit:



Podríamos decir que, en la imagen anterior, la operación de *write* (= escribir en memoria = entrada/almacenamiento de información) se activa o habilita únicamente cuando el reloj lo permite, y la de *read* (= leer de la memoria = salida/lectura de información) está activada o habilitada todo el tiempo.

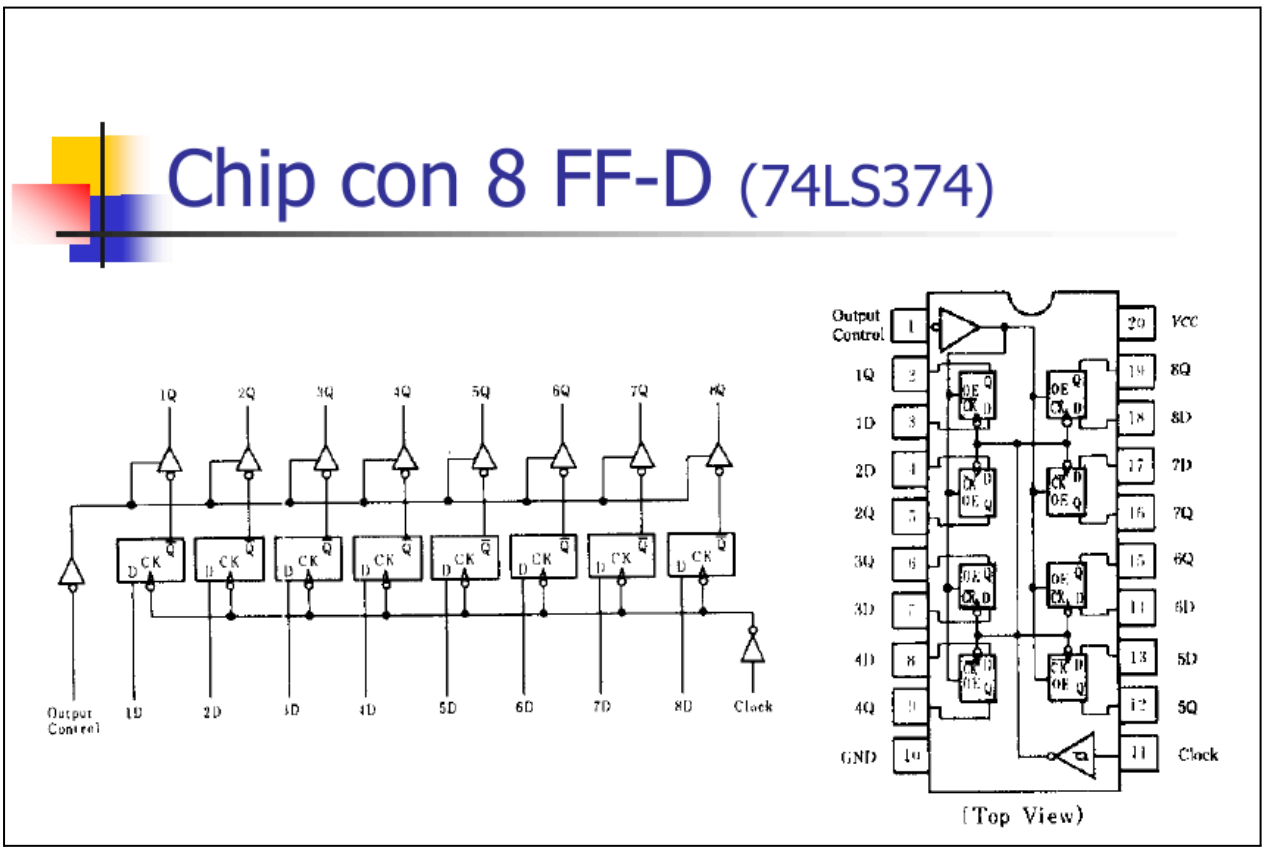
Ahora, ¿para qué nos sirve un bit? Ni idea, pero si encadenamos muchos de estos, podemos almacenar más y más, hasta llegar, por ejemplo, a n bits. Y esto ya tiene más pinta. Así nos vamos acercando a lo que son los registros.



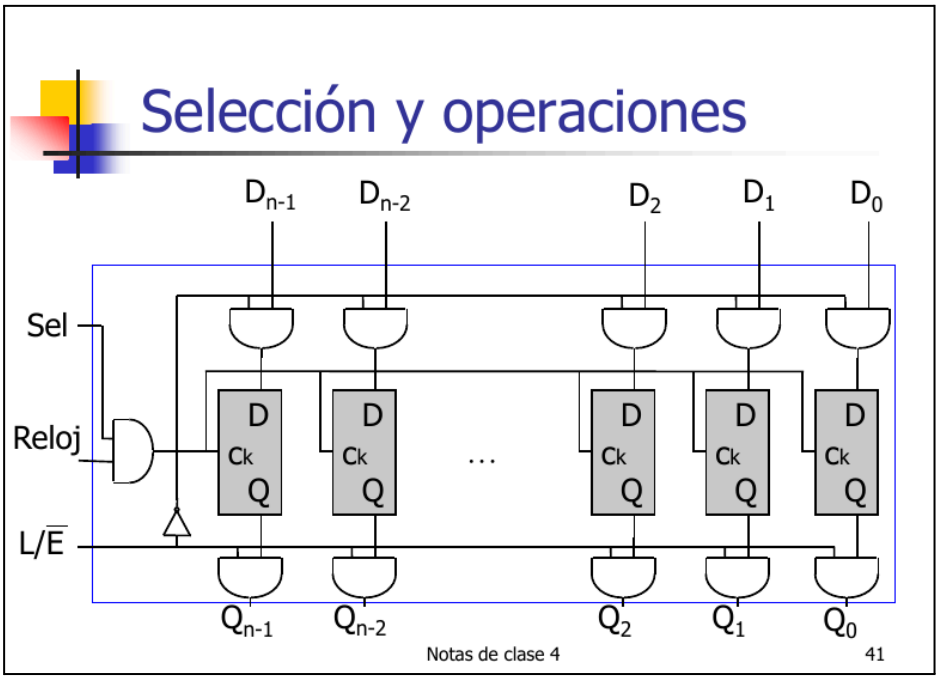
Veamos un ejemplo concreto: un chip con 8 FF D para almacenar un byte. Aquí sí ya estamos ante un registro (de un byte), porque ya cuenta con compuertas para manejar el flujo de datos.

Los 8 bits ingresan desde 1D a 8D. La entrada del reloj tiene un símbolo de burbuja (círculo) que indica que es sensible al flanco positivo, no que esté negada. Y el *output control* está para habilitar o deshabilitar la operación de lectura. (Por lo visto, estos diagramas más realistas

pueden diferir un poco a lo que estamos acostumbrados a ver en la materia, por eso para estudiarlo bien conviene buscar información desde otras fuentes).



Aplicando esta idea, pueden construirse muchos dispositivos similares. Como el siguiente, donde si $L/\overline{E} = 1$, se habilita la lectura, y si es 0, la escritura. El selector habilita o deshabilita que pase la señal del reloj, y, en consecuencia, la posibilidad que ingresen datos. O sea: habilita o deshabilita la operación de escritura.



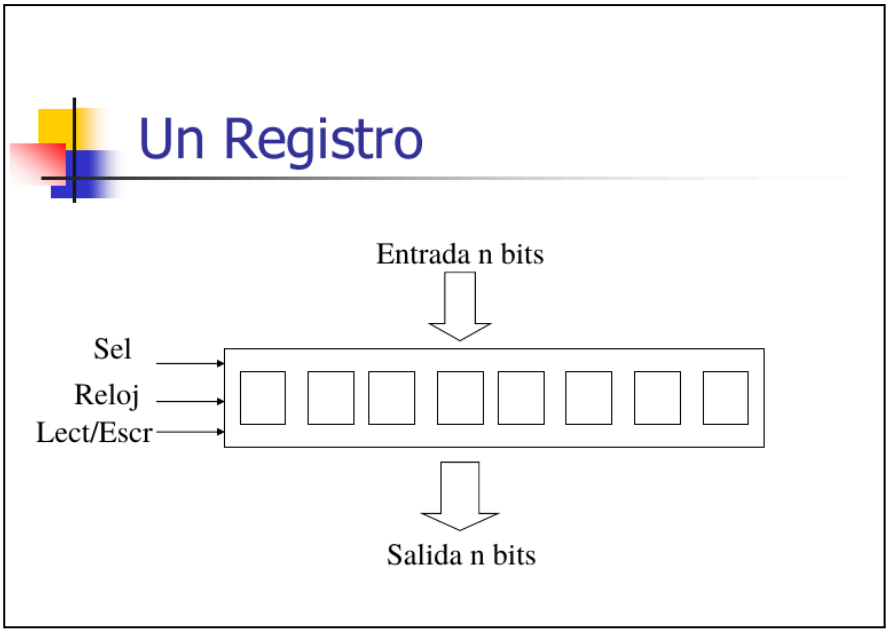
Si miramos esto desde un punto de vista más abstracto (sin prestar atención a la implementación de los circuitos que lo componen), podríamos decir que un registro está compuesto de la siguiente manera:

Los n flip-flops que lo componen, donde se almacenan los n bits.

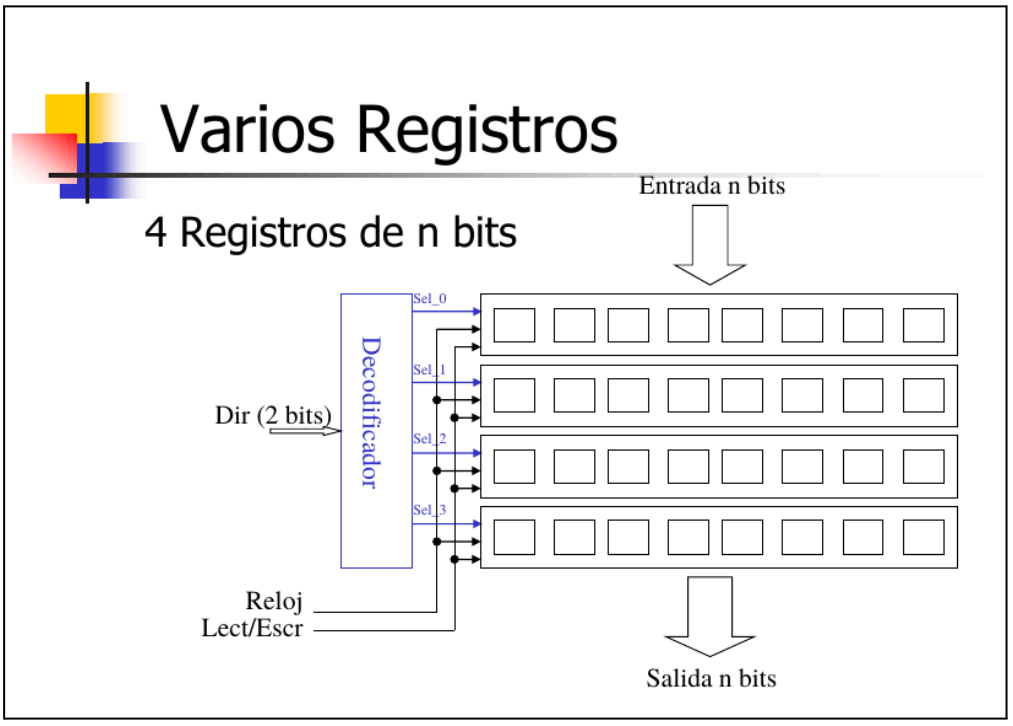
Un selector, que habilita o deshabilita la operación de escritura.

Un reloj, para sincronizar las operaciones.

Un bit para seleccionar la operación que vamos a realizar (lectura o escritura).



¿Y para qué queremos el selector? Entre otras cosas, es fundamental para cuando tenemos muchos registros, y queremos elegir a un registro en particular y no a otro. Esto puede implementarse con un decodificador, tal como se ve en la siguiente imagen. (Aunque a decir verdad, aquí el selector parece servir también para controlar la salida. No queda del todo claro).



Algo un poco más loco que podemos hacer con esto es un contador. Pensemos en algo fácil, como en un contador de 3 bits. Los 8 números que cuenta, de 0 a 7, son estos:

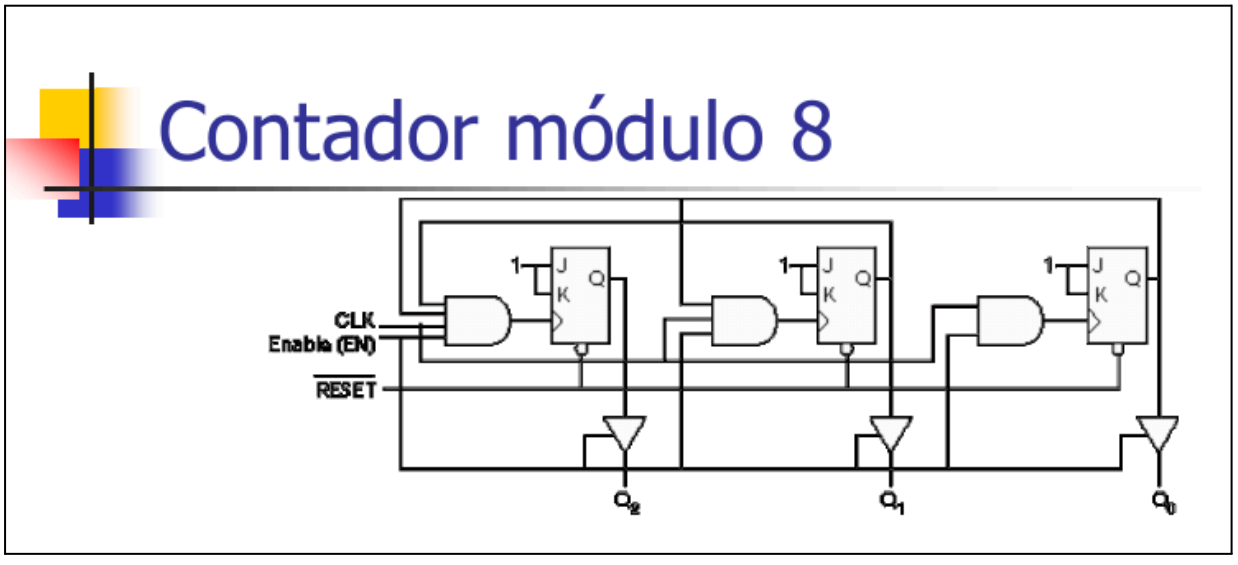
000
001
010
011
100
101
110
111

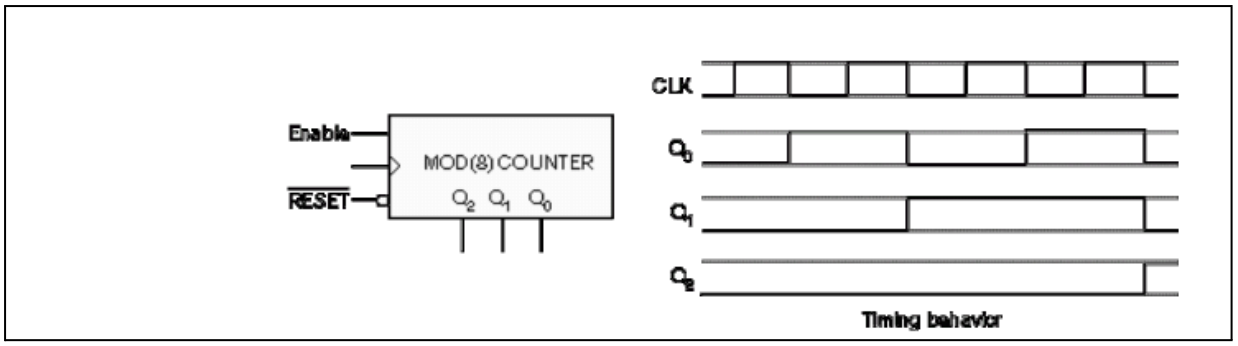
Si miramos detenidamente cada columna, y la conexión entre una columna y la siguiente, podremos observar dos patrones que se repiten. Primero el más fácil: que la primera columna es 01010101…, la segunda es 00110011…, etc. Es decir, que dentro de la propia columna, la cantidad de ceros y unos se repite sin parar y sin variar. En la primera de a uno, en la segunda de a dos, después de a cuatro, y así siguiendo, siendo cada una el doble de la anterior.

El segundo patrón es más sutil. Si miramos los números de recién, podemos pensar que una columna cambia de estado dependiendo de si en la columna que está a su derecha hubo una "bajada" de 1 a 0. Visto de esa manera, tenemos una "fuente", que es la primera columna, que va haciendo avanzar el contador. Empieza en cero, así que no cambia nada (primer renglón). Después pasa a 1 (segundo renglón), y de ahí a 0 (tercer renglón). Cuando eso suceda, la segunda columna cambiará de estado; por eso en el tercer renglón pasa de 0 a 1. Y solo cuando la segunda columna baje de 1 a 0, la tercera cambiará de estado; y esto pasa en el quinto renglón.

El contador no hace esto, pero la idea es análoga y sirve para empezar a entenderlo. Existe una fuente inicial (el reloj), que hace avanzar el contador marcando el pulso, y el resto de los elementos del sistema (los flip-flops D), están encadenados de una manera similar a las columnas que vimos recién, haciendo que el tiempo se duplique a medida que nos alejamos de la fuente original.

No digo que sea fácil, pero con toda esa información debería ser comprensible el siguiente circuito (cuyo comportamiento en el tiempo —retratado por un osciloscopio— está en la siguiente página):





Obviamente, por cada FF tenemos un bit más, y así es como sabemos hasta dónde puede contar. O sea que el rango de un contador con n FFs es igual al rango del BSS con n bits.

En *Computer Organization and Architecture* de Stallings (10th edition), págs. 402-403 todo esto está un poco mejor presentado.

ANEXO: datos sueltos sobre FFs

FF SR y JK pueden usarse en diversas aplicaciones, no limitadas a contadores o registros

Ambos pueden ser sincrónicos o asincrónicos

Ambos pueden activarse por nivel o por flanco dependiendo de su diseño