Circuitos Combinatorios

Unidad 2.1
Simplificaciones y circuitos básicos

Versión 1.0.0

Carlos Maidana, Carlos Rodríguez, Edgardo Gho, Martín Ferreyra Biron, Jaír Hnatiuk

Simplificación visual

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C$$

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	1
1	1	1	1

Vemos que en este caso, cuando A=1 y B=1 (remarcado en azul) existen dos valores para C… 0 o 1 (remarcados en violeta). En cualquiera de los dos casos, la función toma el valor **1**. Esto quiere decir que no importa el valor de C en los casos donde A=1 y B=1. Es por esto que podríamos simplificar ambos términos y escribir simplemente A.B

Notemos que NO existen otros casos en donde A=1 y B=1 fuera de los marcados en azul.

Simplificación visual

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C$$

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	1
1	1	1	1

Algo similar ocurre en este caso, donde en azul vemos que los valores de B y C son 1 para ambos valores de A (remarcados en violeta) y en ambos casos la función F toma el valor 1. Esto quiere decir que no importa el valor de A siempre y cuando B=1 y C=1, la función toma 1. Por ende podemos simplificar a B.C.

Esto es más difícil de ver que el caso anterior ya que no se encuentran en renglones consecutivos en la tabla de verdad.

Simplificación algebraica

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C$$

$$F(A,B,C)=\overline{A}.B.C+A.\overline{B}.C+A.B.\overline{C}+A.B.C+A.B.C+A.B.C$$

$$F(A,B,C)=\overline{A}.B.C+A.B.C+A.\overline{B}.C+A.B.C+A.B.\overline{C}+A.B.C$$

$$F(A,B,C)=(\overline{A}.B.C+A.B.C)+(A.\overline{B}.C+A.B.C)+(A.B.\overline{C}+A.B.C)$$

$$F(A,B,C)=B.C+A.C+A.B$$

Podemos simplificar de forma algebraica utilizando los postulados. En este caso obtenemos una Función nueva, en donde seguimos usando A, B y C, pero notamos que de los 4 términos originales ahora pasamos a 3 términos. A su vez cada término tiene solo dos variables. Ambas funciones son matemáticamente equivalentes.

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

Reacomodamos la forma de ver la función utilizando un mapa de karnaugh. Esto es simplemente una forma distinta de acomodar los valores de la función. La forma en la que acomodamos las variables en el mapa de Karnaugh no es importante.

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

Lo que es importante es acomodar las filas y columnas siguiendo código de gray. Vemos que en rojo A=1,B=1, en violeta C=1 y entonces F=1, el cual marcamos en verde.

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

C\AB	00	01	11	10
0			1	
1		1	1	1

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

Acomodamos los unos de la función en su posición correspondiente en el mapa. Esta forma de ubicarlos permite identificar términos candidatos a ser simplificados

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

Resaltamos una de las posibles simplificaciones. Notemos que A=1 y B=1 en ambos casos, mientras que C=0 y C=1 en los unos resaltados. Esto quiere decir que podemos simplificar C dejando solamente A.B.

Método más simple

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	1
1	1	1	1

AB\C	0	**1**
00		
01		1
11	1	1
10		1

Vemos ahora que en este caso en la agrupación B=1 y C=1 tenemos los valores A=0 y A=1. Eso quiere decir que la variable A puede ser simplificada quedando B.C.

Método más simple

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	1
1	1	1	1

AB\C	0	**1**
00		
01		1
11	1	1
10		1

Por último nos queda el grupo A=1 C=1 para B=0,B=1. Por ende simplificamos B quedando A.C.

Método más simple

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	1
1	1	1	1

AB\C	0	1
00		
01		1
11	1	1
10		1

$$F(A,B,C)=B.C+A.C+A.B$$

En este caso todos los unos de la función pudieron agruparse. Podríamos tener casos donde un uno no pueda agruparse. En ese caso, ese término no puede simplificarse.

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

Notemos en esta función el valor X para 101. Este valor representa “don’t care” (no importa). Eso quiere decir que puede tomar cualquier valor (0 o 1) ya que ese caso (101) no nos importa en la función.

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

AB\C	0	1
00		1
01		
11	1	1
10	1	X

En este caso, el valor X puede ser tomado como 1 o 0 dependiendo que tan conveniente sea.

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

AB\C	0	1
00		1
01		
11	1	1
10	1	X

Tenemos dos grupos. El primero en violeta agrupa 4 unos, vemos que agrupa 110, 111, 100, 101. El único que se mantiene constante es el primer 1 correspondiente a A. Por ende se simplifica como A.

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

AB\C	0	1
00		1
01		
11	1	1
10	1	X

El segundo grupo en rojo, vemos que “pega la vuelta” (adyacencia). Esto es válido como grupo. Vemos que agrupa 001 y 101. Vemos que A cambia (0 y 1) mientras que B=0 y C=1, por ende se simplifica como !B.C

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

AB\C	0	1
00		1
01		
11	1	1
10	1	**1**

$$F(A,B,C)=A+\overline{B}.C$$

Vemos que el convertir X a 1 es muy conveniente ya que permite agrupar.

Una función con valores no definidos

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	**X**
1	1	0	1
1	1	1	1

AB\C	0	1
00		1
01		
11	1	1
10	1	**0**

Si hubiésemos tomado 0 como valor para 101, entonces solo podemos hacer dos simplificaciones.

$$F(A,B,C) = A.B + A.\overline{C} + \overline{A}.\overline{B}.C$$

Producto de sumas

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

AB\C	0	1
00	0	
01	0	0
11		
10		X

Podemos usar el mapa de karnaugh para simplificar por los ceros. Marcamos los ceros y agrupamos. Recordemos que los ceros generan productos de sumas, por ende las variables en 0 se complementan.

Producto de sumas

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

AB\C	0	1
00	0	
01	0	0
11		
10		X

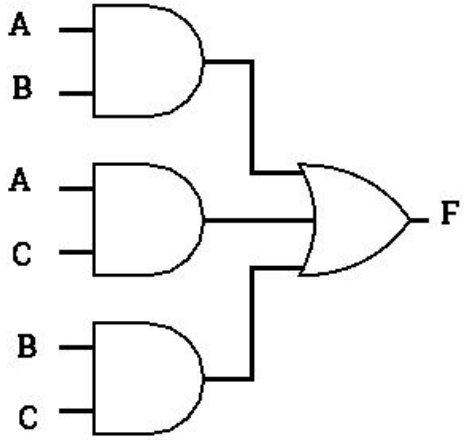
$$F(A,B,C)=(A+C).(A+\overline{B})$$

Vemos que 000 y 010 puede simplificarse B. En este caso los 0s se complementan, por ende queda A+C. Luego en 010 011 podemos simplificar C. Queda A+!B. Vemos que el X conviene tomarlo como 1 e ignorarlo.

Función mayoría con compuertas

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	1
1	1	1	1

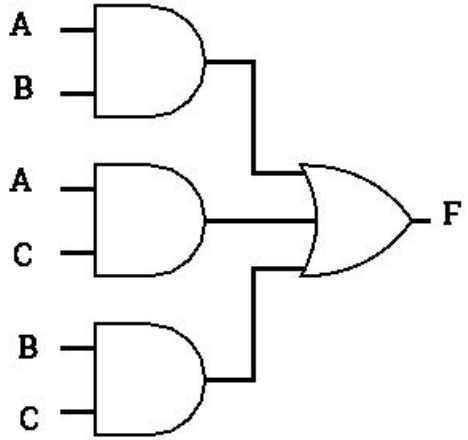
$$F(A,B,C)=B.C+A.C+A.B$$

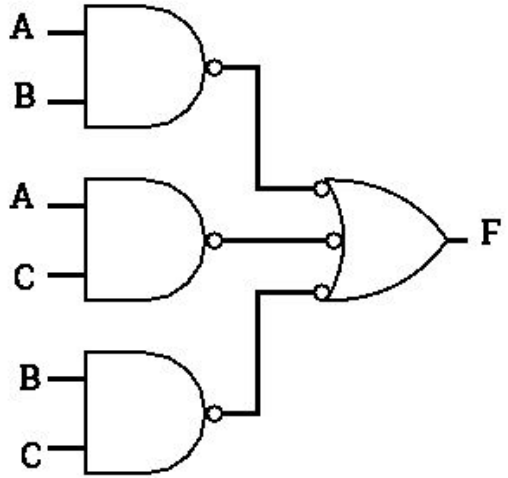


La función simplificada utiliza menos compuertas que la función original. Dado que ambas cumplen la misma función, son equivalentes, y nos conviene simplificar siempre y cuando sea posible.

Implementar con un tipo de compuertas

$$F(A,B,C)=B.C+A.C+A.B$$

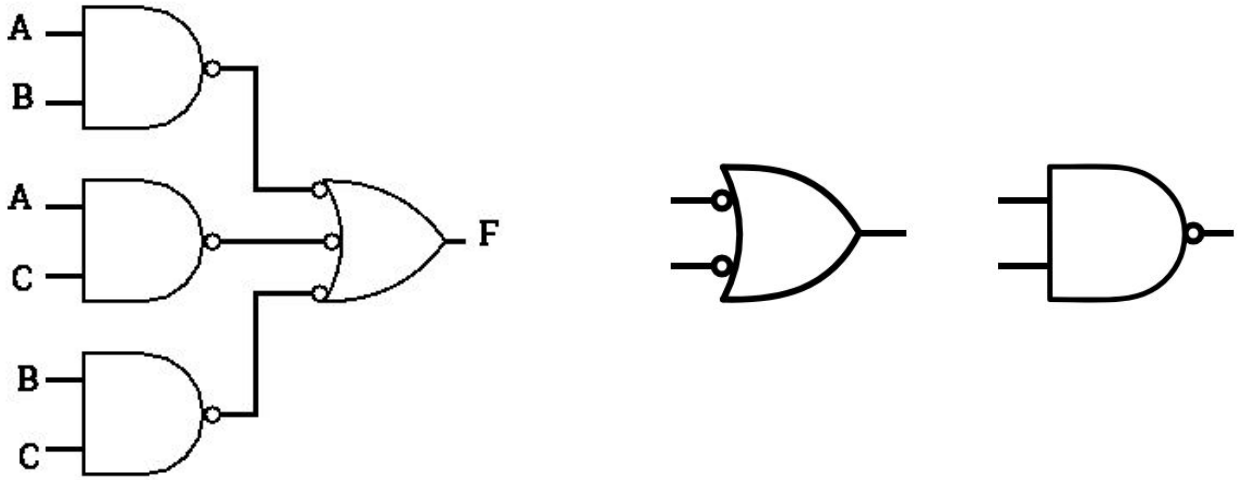




La función de la derecha cumple la misma función, nótese que la salida de cada AND está negada pero la entrada correspondiente en la OR también está negada, y dos negados se cancelan mutuamente.

Implementar con un tipo de compuertas

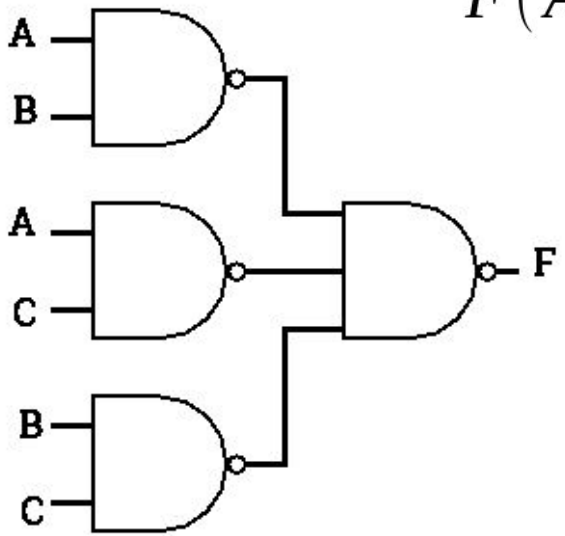
$$F(A,B,C)=B.C+A.C+A.B$$



Utilizando DeMorgan sabemos que una compuerta OR con sus entradas negadas equivale a una compuerta NAND.

Implementar con un tipo de compuertas

$$F(A,B,C)=\overline{\overline{A.B}.\overline{A.C}.\overline{B.C}}$$



Veremos en la unidad de tecnología que las compuertas NAND tienen ventajas sobre las AND/OR en ciertas tecnologías, por ende preferimos la implementación con NAND.

Implementar con un tipo de compuertas

$$F(A,B,C)=(A+B+C).(A+B+\overline{C}).(A+\overline{B}+C).(\overline{A}+B+C)$$

Si simplificamos la función en su segunda forma canónica obtendremos un producto de sumas. Aplicando la misma estrategia de negar la salida y la entrada obtendremos compuertas NOR en vez de NAND. Dejamos esto como ejercicio.

Circuitos combinatorios básicos

Sumador de 2 números de 1 bit

Ahora que conocemos como implementar funciones (simplificadas, con lógica de 2 niveles) podemos estudiar circuitos combinacionales comúnmente encontrados en diseños de circuitos. Uno de los más comunes es un sumador que permite sumar dos números de 1 bit.

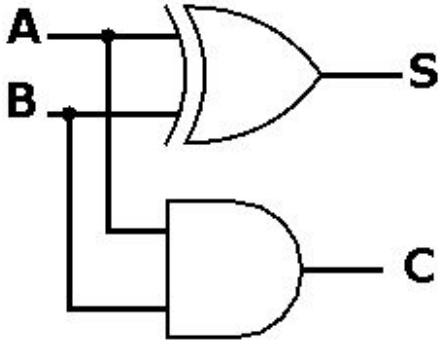
entrada		salida	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

A este circuito lo conocemos como Half-Adder (semisumador) y posee como entrada dos números de 1 bit (A y B) y como salida la suma (S) y el carry (C).

Sumador de 2 números de 1 bit

Vemos la implementación circuital. Cada salida es una función. Vemos que S cumple con la tabla de verdad de la XOR y que C cumple con la AND.

entrada		salida	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

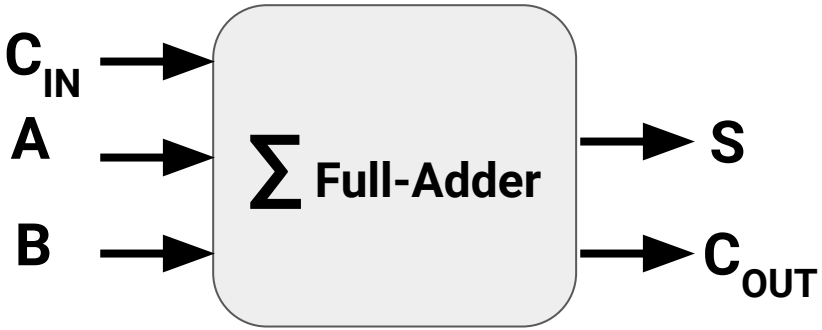


Sumador de 3 números de 1 bit

Si bien podemos juntar dos Half-Adder y armar un sumador que considere un carry entrante, también podemos directamente plantearlo desde la tabla de verdad.

Pero si miramos C_{OUT} tiene “pinta” de función mayoría. El caso de S es una XOR de 3 entradas. Podemos verlo planteando C XOR (A XOR B). Resolver primero A XOR B y al resultado aplicar XOR C.

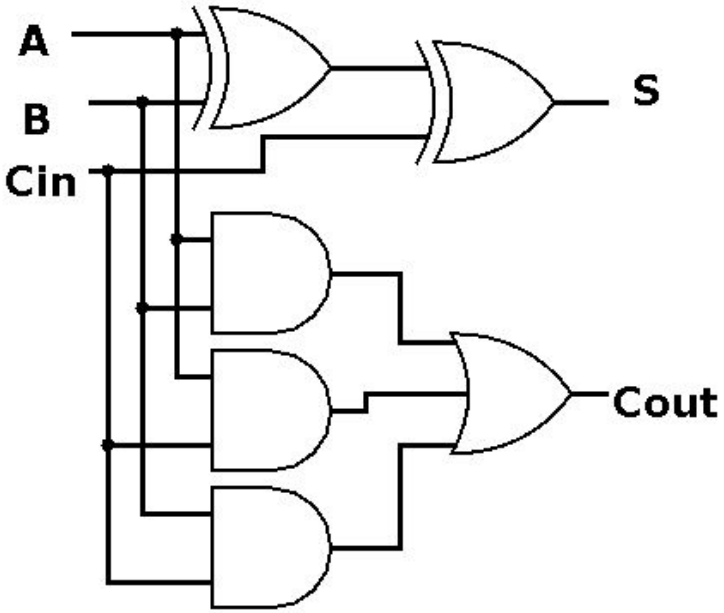
entrada			salida	
C_{IN}	A	B	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Sumador de 3 números de 1 bit

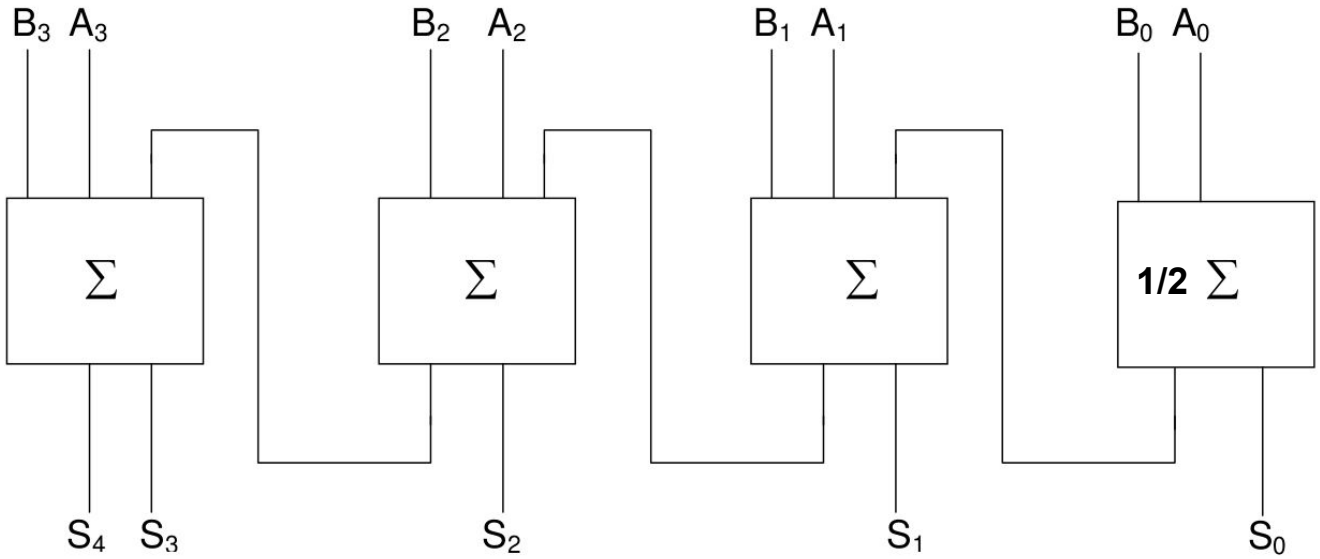
En esta implementación nos aseguramos que existan dos niveles de compuertas tanto para S como para C_{out}. Cuando veamos retardos veremos que es importante mantener los mismos retardos en todas las salidas.

entrada			salida	
C_{IN}	A	B	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



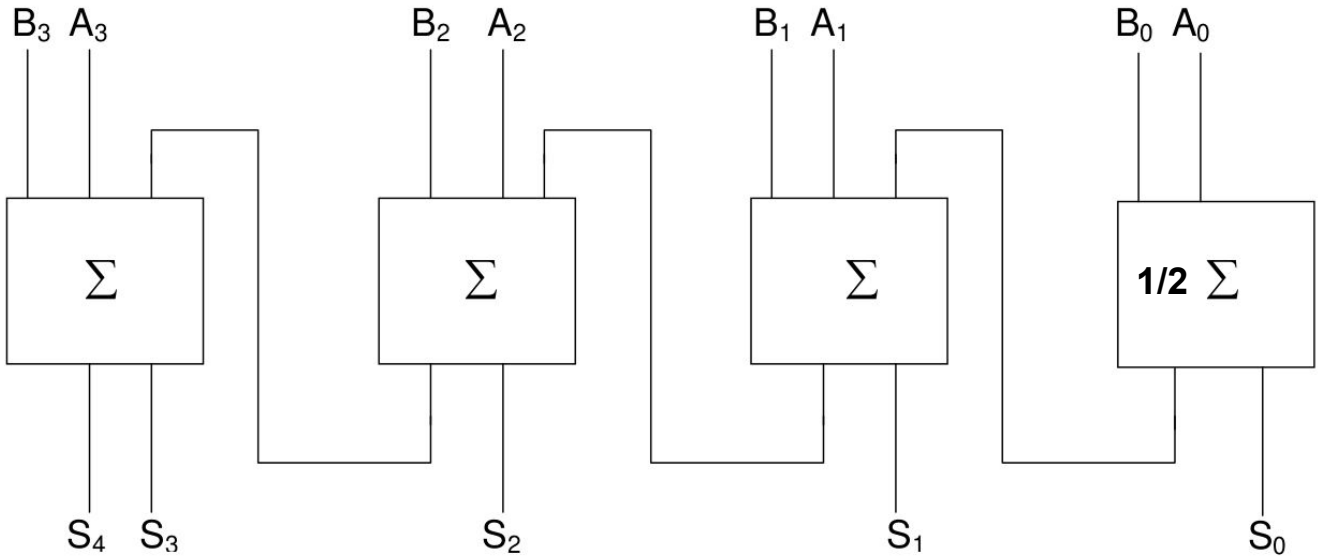
Encadenando sumadores

Si deseamos sumar 2 números de 4 bits, podemos tomar la salida de carry del primer sumador y conectarlo a la entrada de carry del siguiente, y así con el resto formando una cadena. S4 en este caso es el carry de salida final.



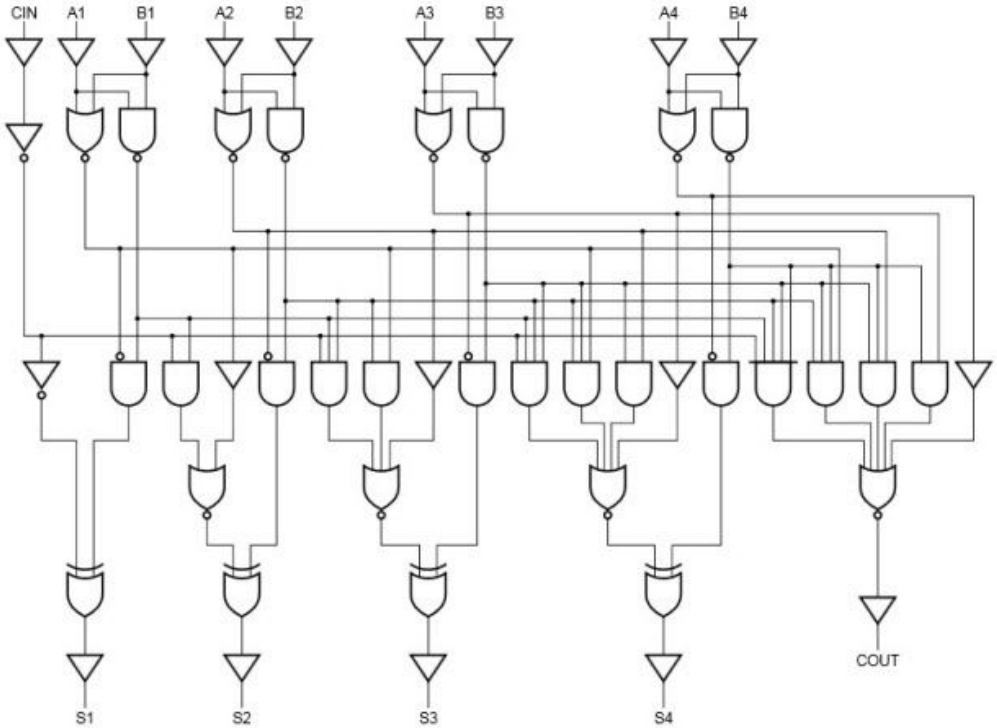
Encadenando sumadores

En la unidad de tecnología veremos que este encadenamiento implica tiempo. Debe resolverse la suma de los bits menos significativos antes de poder resolver los bits más significativos.



Encadenando sumadores

Existen sumadores con “carry look-ahead” los cuales anticipan el valor del carry en cada sumador.



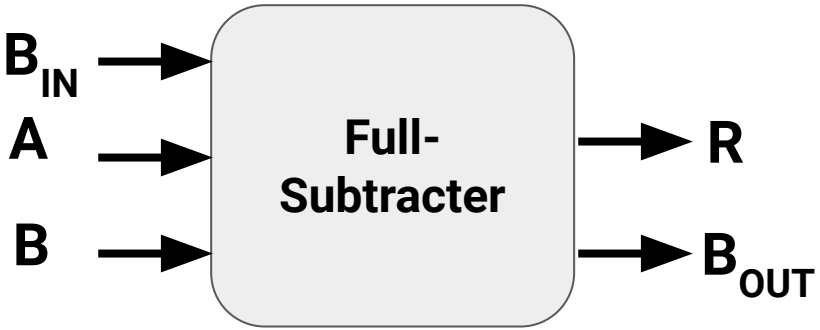
https://laptrinhx.com/carry-lookahead-adder-1766329680/

Restador de 3 números de 1 bit

Así como podemos armar un circuito que sume y considere un carry de entrada y uno de salida, podemos hacer el equivalente con la resta (A-B).

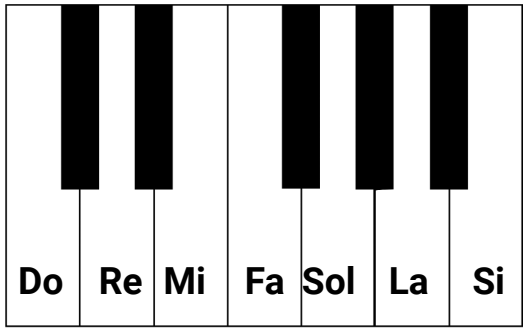
entrada			salida	
A	B	B_{IN}	B_{OUT}	R
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

En este caso la señal de B_{IN} representa un préstamo (borrow) pedido por la etapa anterior, y B_{OUT} representa un pedido de préstamo a la etapa siguiente. Vemos que R es parecido a algo del Full-Adder.



Codificador Simple

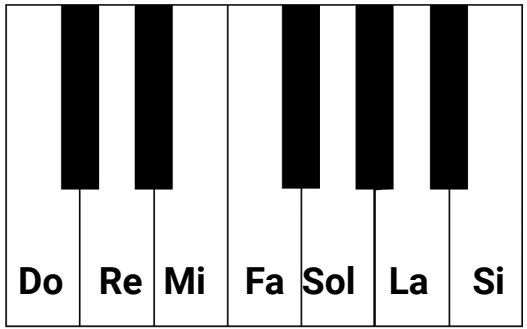
Un codificador es un circuito que toma una cantidad de entradas (por ejemplo las 7 teclas blancas del piano) y genera un número binario equivalente a la posición de la tecla apretada. No se permite presionar más de una tecla a la vez. Esto generaría una salida invalida.



entrada							salida		
Do	Re	Mi	Fa	Sol	La	Si	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	1
0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	1	1	1	1

Codificador Simple

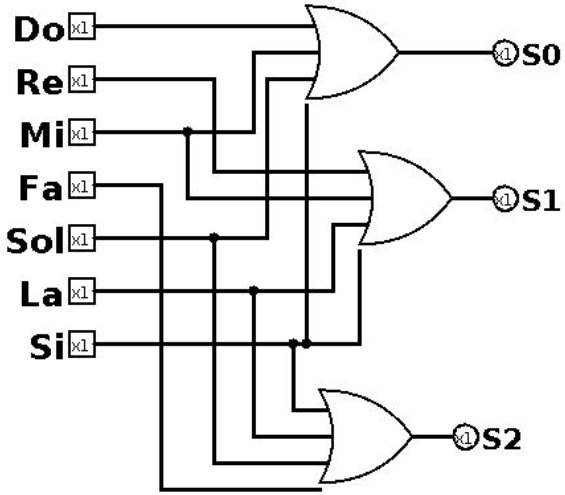
Vemos que cuando ninguna tecla se encuentra presionada (lo que generaría un 1 en la entrada) la salida es 000. Luego si Do se encuentra apretada la salida es 001, Re=010, Mi=011 …. Si=111. Vemos que este circuito posee 7 entradas, por ende tiene 2^7 = 128 combinaciones en la tabla de verdad. Sin embargo solo algunas nos interesan.



entrada							salida		
Do	Re	Mi	Fa	Sol	La	Si	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	1
0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	1	1	1	1

Codificador Simple

La implementación de este circuito consiste en compuertas OR que identifiquen los unos en las entradas que hacen que la salida sea 1. Ej: S_2 vale 1 siempre y cuando Fa=1 o Sol=1 o La=1 o Si=1. El problema ocurre cuando dos entradas se encuentran en uno a la vez.

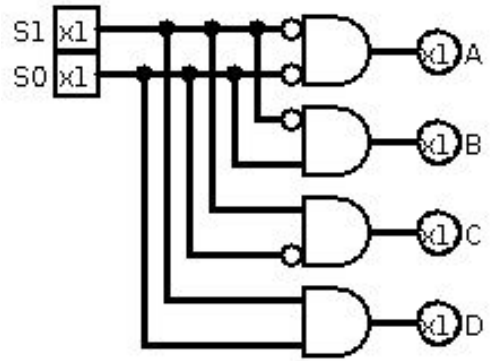


entrada							salida		
Do	Re	Mi	Fa	Sol	La	Si	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	1	1
0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	1	1	1	1

Decodificador

El decodificador toma como entrada un número binario y genera una combinación de salida “one-hot”, o sea una combinación de salida donde solo una de las salidas tienen un uno. El decodificador es un circuito muy útil para transformar una dirección representada como un número binario de N bits en 2^N salidas donde solo una se encuentra en uno.

entrada		salida			
S_1	S_0	A	B	C	D
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



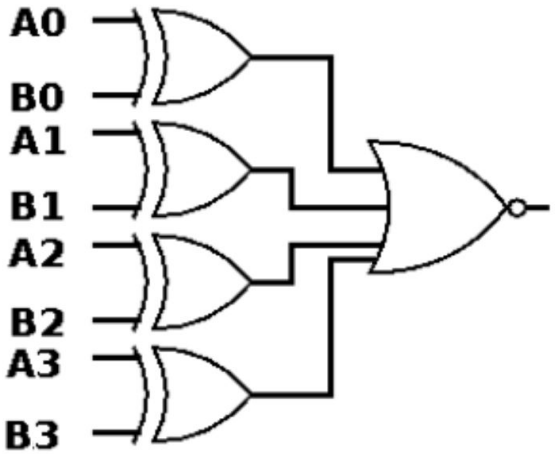
Generador de Paridad

Vemos que la compuerta XOR calcula la paridad par en los unos para sus dos entradas. Cuando A=B entonces la paridad es cero. Cuando A!=B entonces la paridad es 1. Nótese que si tenemos 3 entradas (o N) podemos ir encadenando XOR y tomando el resultado con el bit siguiente. Podemos transformar este generador de paridad en un validador de paridad tomando la salida P como entrada en un generador. En ese caso el resultado de P’ del segundo generador deberá ser 0 si la paridad es válida.

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

Comparador por Igual

Si disponemos de dos números (A y B) de 4 bits cada uno, de forma que $A=A_3A_2A_1A_0$ y $B=B_3B_2B_1B_0$, podemos restarlos y si el resultado es 0 saber si A=B. Pero viendo la tabla de verdad de la XOR, si A=B entonces A XOR B es 0. Luego con una NOR verificamos si alguna de las comparaciones bit a bit resultó en 1.

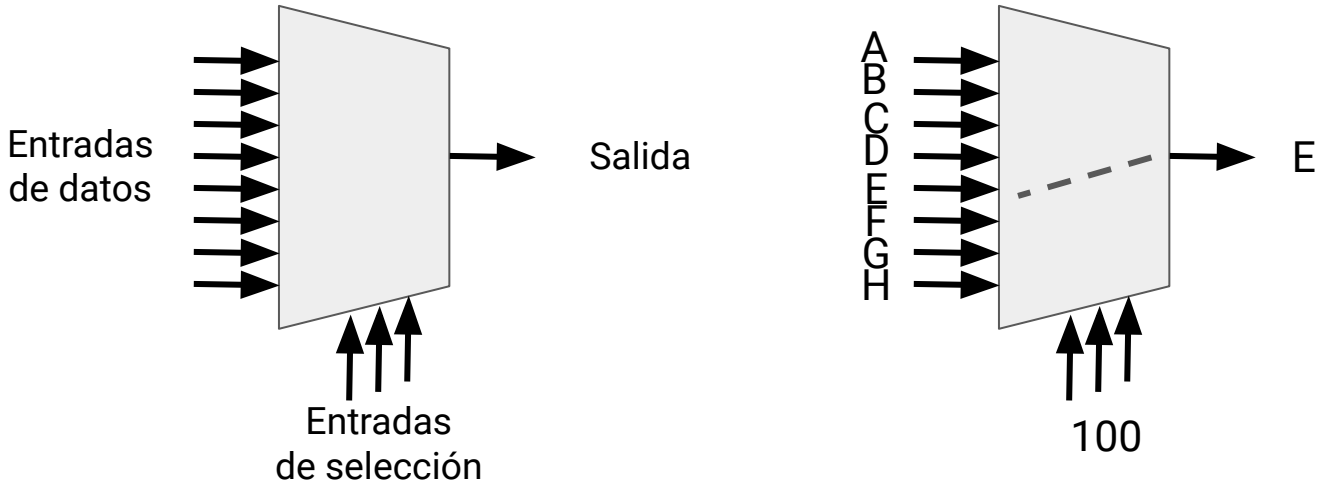


A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

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

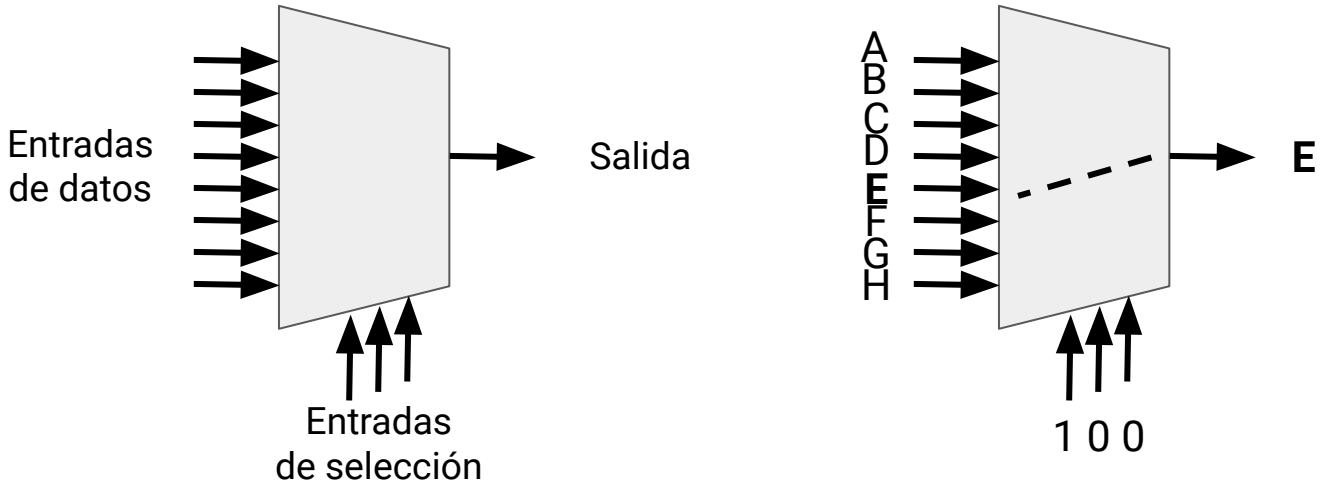
Multiplexores

El multiplexor es un circuito muy particular, ya que posee una salida pero dos tipos de entradas, las de selección y las de datos. Se lo representa de la siguiente forma, donde las entradas de datos se encuentran a su izquierda, la salida a la derecha y las entradas de selección por debajo.



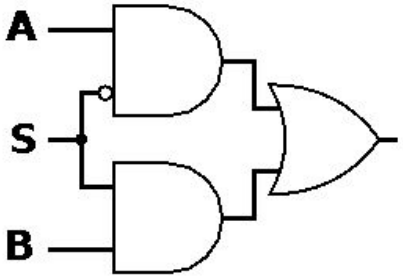
Multiplexores

El valor de la entrada de selección indica cuál de las entradas de datos está conectada a la salida. En este caso de MUX8 (8 entradas de datos) necesitamos $Log_2 8$ entradas de selección. Vemos en el ejemplo que si selección es 100 (4) se selecciona la entrada 4 (empezando de 0) y se la conecta a la salida.

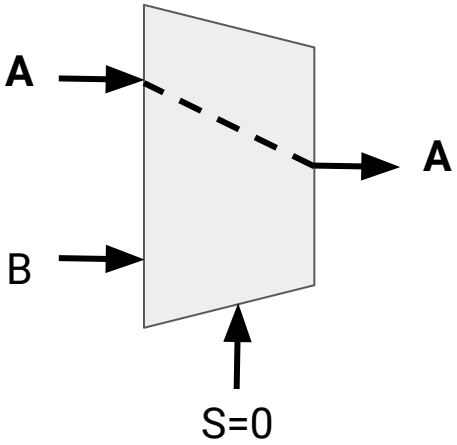


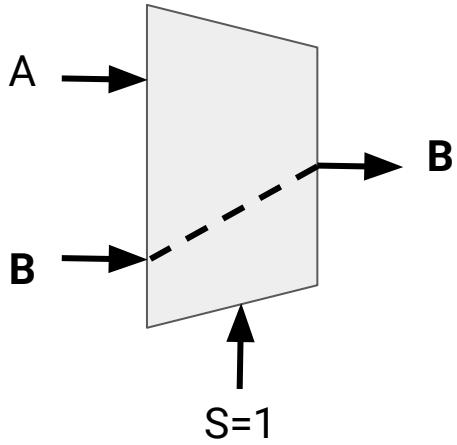
Multiplexores

Vemos que el multiplexor más simple es aquel con dos entradas de datos y una de selección. Vemos que si S=0 entonces la salida queda conectada a la entrada A. Si S=1 entonces la salida queda conectada a la entrada B.



S	F
0	A
1	B



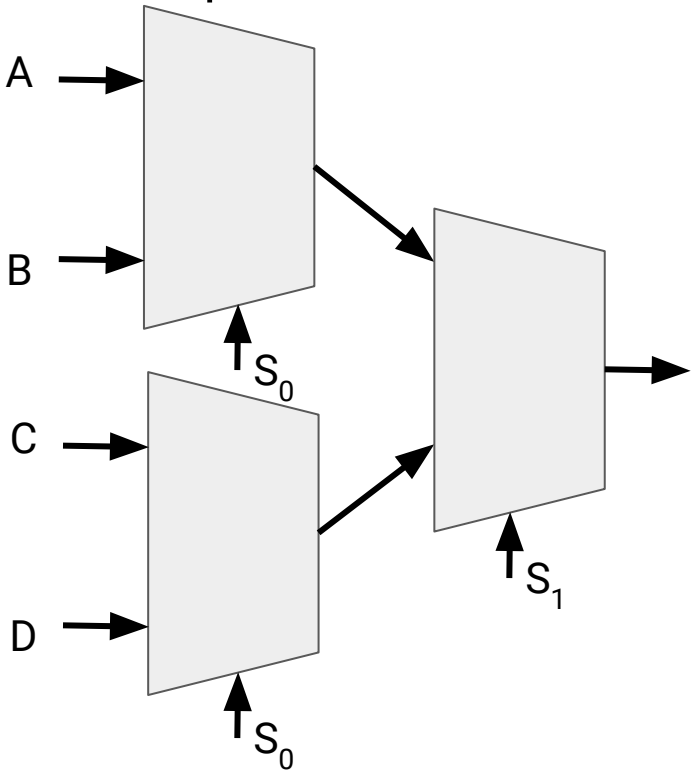


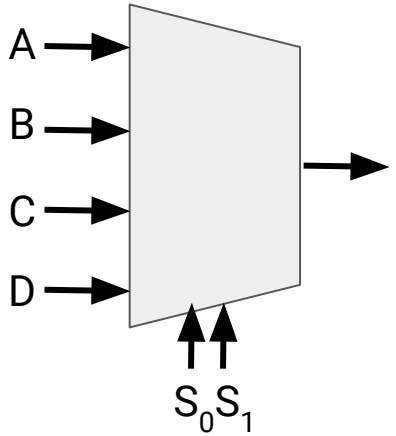
S	A	B	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Multiplexores de órdenes mayores

Podemos construir multiplexores de órdenes mayores utilizando multiplexores más simples.

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

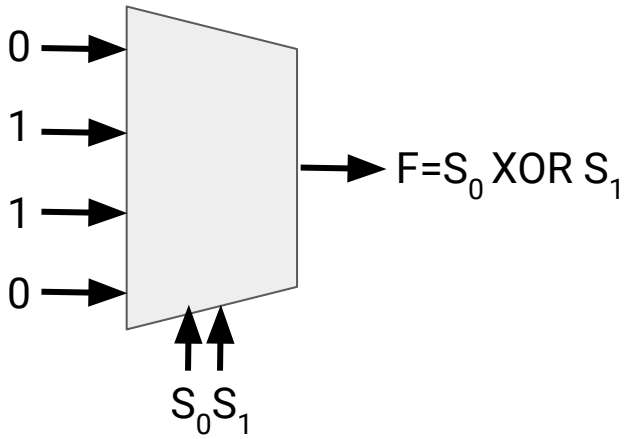




Funciones booleanas con multiplexores

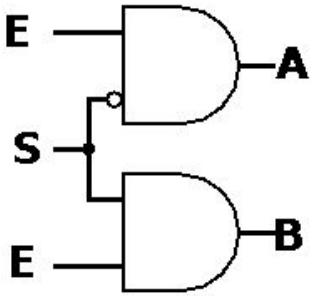
Vemos que las entradas de datos pueden conectarse a variables, pero también pueden llevar valores constantes. Asignando valores fijos a A,B,C y D, podemos implementar cualquier función booleana de dos variables. Vemos por ejemplo cómo generar una compuerta XOR utilizando el MUX y entradas 0110.

S_1	S_0	F
0	0	0
0	1	1
1	0	1
1	1	0



Demultiplexor

Como contraparte del MUX, tenemos el demultiplexor. En este circuito tenemos una sola entrada de datos, N salidas y Log_2 N entradas de selección. La idea es conectar la única entrada de datos a distintas salidas dependiendo de las líneas de selección. También puede implementarse en órdenes superiores.



S	E	A	B
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

S	A	B
0	E	0
1	0	E