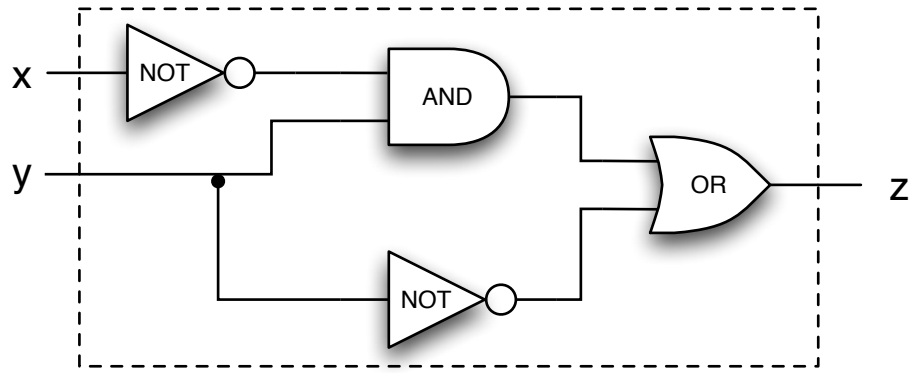
Métodos de diseño de circuitos digitales

Sistemas combinacionales

En un circuito combinacional los valores de las salidas dependen únicamente de los valores que tienen las entradas en el presente. Se construyen a partir de compuertas lógicas: AND, OR, NOT u otras.



Ejemplo: Circuito C1

Para especificar la función de un circuito se usan:
1. **Tablas de verdad**: Para cada combinación de entradas se especifican las salidas.
2. **Formulas Algebraicas**: Se usa notación algebraica que se deriva directamente del circuito.

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

Tabla de verdad de C1

$$z = \neg x \cdot y + \neg y$$

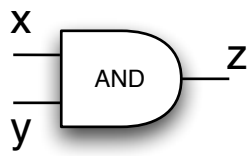
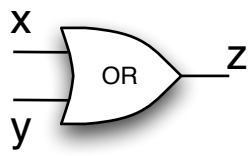
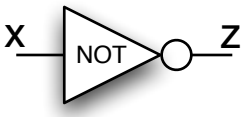
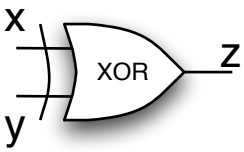
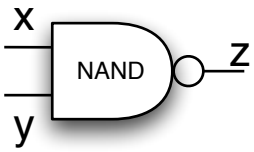
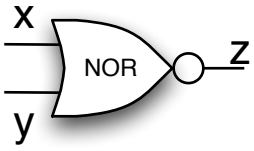
Formula de C1

Observe que la tabla de verdad de C1 es idéntica a la tabla de NAND. Entonces C1 es equivalente a **z = ¬(x.y)** Dado que un NAND produce las mismas salidas que C1 y que un NAND necesita menos transistores que C1, entonces un NAND resulta más conveniente.

Se necesita una metodología que permita transformar circuitos costosos en circuitos más simples.

La teoría del Algebra de Boole nos permitirá comprender esta metodología.

Tabla de Compuertas Lógicas

Símbolo	Significado	Tabla de Verdad	Formula Algebraica
	y-lógico	x y | z 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	z = xy
	o-lógico	x y | z 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	z = x + y
	negación	x | z 0 | 1 1 | 0	z = ~x
	o-exclusivo	x y | z 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	z = x ⊕ y
	y-negado	x y | z 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	z = ~(xy)
	o-negado	x y | z 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	z = ~(x+y)

Algebra de Boole

Un álgebra de Boole es una estructura algebraica denotada por **< A , + , . , 0 , 1 >**, en donde:

- **A** es un conjunto
- **+** y **.** son operadores binarios
- **{0, 1}** ⊂ **A**

La estructura verifica los siguientes axiomas:

A1	asociatividad de **+**	∀ **a, b, c : (a+b) + c = a + (b+c) = a+b+c**
A1'	asociatividad de **.**	∀ **a, b, c : (a.b) . c = a . (b.c) = a.b.c = abc**
A2	conmutatividad de **+**	∀ **a, b: a+b = b+a**
A2'	conmutatividad de **.**	∀ **a, b: a.b = b.a**
A3	neutro para **+**	**a+0 = a**
A3'	neutro para **.**	**a.1 = a**
A4	distrib de **.** c/r a **+**	**a . (b+c) = ab + ac**
A4'	distrib de **+** c/r a **.**	**a + (bc) = (a+b) . (a+c)**
A5	existencia de complemento	∀**a** ∃**!¬a tq a + ¬a = 1 y a . ¬a = 0**

Teoremas básicos de un álgebra de Boole:

T1	**a + a = a**	**T1'**	**a . a = a**
T2	**a + 1 = 1**	**T2'**	**a . 0 = 0**
T3	**a + ab = a**	**T3'**	**a (a+b) = a**
T4	**¬¬a = a**		
T5	**¬(ab) = ¬a + ¬b**	**T5'**	**¬(a+b) = ¬a ¬b**
T6	**a + ¬ab = a + b**	**T6'**	**a (¬a+b) = ab**

Otras maneras de escribir

OR:	**+**	∪	∨	**I**	**II**
AND:	**.**	∩	∧	**&**	**&&**
NOT:	**ā**	**’**	**¬**	**~**	**!**

Propuesto: La estructura **< {0, 1}, OR, AND, 0, 1>** correspondiente a las compuertas lógicas es un álgebra de Boole, con **¬0 = 1 y ¬1 = 0**

Demostración: usando tablas de verdad para verificar que se cumple cada uno de los axiomas.

Ejemplo: (completar esta tabla)

a	b	c	ab	(ab)c	bc	a(bc)
0	0	0				
0	0	1	0	0	0	0
0	1	0				
0	1	1	0	0	1	0
1	0	0				
1	0	1				
1	1	0				
1	1	1	1	1	1	1

Nota: 3 variables implican 2^3 filas. Columnas 3 y 5 resultan ser iguales.

Propuesto: Verifique la distributividad de **+** c/r **.** : **a+(bc) = (a+b)(a+c)**

Demostración: (ejercicio ...)

Demostración de teoremas del álgebra de Boole:

T1: a + a = (a + a) . 1 = (a + a) . (a + ~a) = a + (a . ~a) = a + 0 = a [A3', A5, A4', A5, A3]

T1': Basta reemplazar en la demostración anterior los **+** por **.** , **0** por **1** , **Ax** por **Ax'** , y viceversa.

T4: Sea **b = ~a** , entonces:

b + a = 1 ya que **b + a = ~a + a = a + ~a = 1 [A2, A5]**
b . a = 0 ya que **b . a = ~a . a = a . ~a = 0 [A2', A5]**

luego **a = ~b** por la unicidad de **A5** ⇒ **a = ~~a**

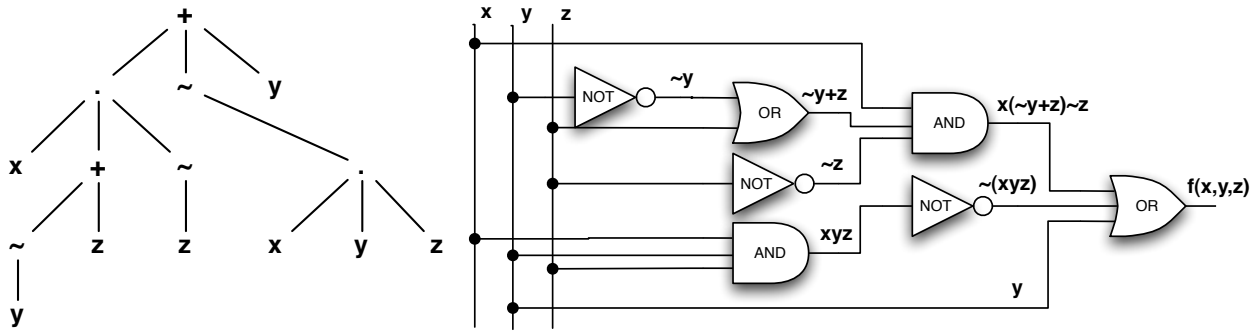
Propuesto: Mostrar que **~((~(a.~b)).~b) = a+b** usando algebra de Boole

Demostración: (ejercicio ...)

Metodología de diseño de circuitos combinacionales a partir de una tabla de verdad.

Por etapas:
1. Dada una tabla de verdad, obtener una formula algebraica.
2. Dada una formula algebraica obtener el diagrama circuital.

Etapa 2 es la mas sencilla: Se construye el árbol de operaciones de la expresión y se reemplazan los operadores por las respectivas compuertas.



Ejemplo: Circuito C2: **f(x, y, z) = x(~y+z)~z+~(xyz)+y**

Etapa 1 es mas complejo. Por ejemplo considerando la tabla de verdad a la derecha:

x	y	f(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

Por "simple inspección": **f(x,y) = ~xy + x~y** por que
f(x,y) vale **1** cuando **x** es **0** e **y** es **1**
f(x,y) vale **0** cuando **x** es **1** e **y** es **0**

Esta forma de determinar intuitivamente una formula está respaldada teóricamente por el siguiente teorema.

Teorema de Shannon:

Sea $\mathbf{f:\{0,1\}^n \rightarrow \{0,1\}}$

$$\Rightarrow f(x_1 \ldots x_i \ldots x_n) = x_i \,.\, f(x_1 \ldots 1 \ldots x_n) + \sim x_i \,.\, f(x_1 \ldots 0 \ldots x_n)$$

Demostración:
Basta probar la igualdad para $x_i = 0$ y $x_i = 1$

Caso $x_i = 0$: $f(x_1 \ldots 0 \ldots x_n) = 0 \,.\, f(x_1 \ldots 1 \ldots x_n) + \sim 0\, f(x_1 \ldots 0 \ldots x_n)$
$= 0 + f(x_1 \ldots 0 \ldots x_n)$
$= f(x_1 \ldots 0 \ldots x_n)$

Caso $x_i = 1$: $f(x_1 \ldots 1 \ldots x_n) = 1 \,.\, f(x_1 \ldots 1 \ldots x_n) + \sim 1\, f(x_1 \ldots 1 \ldots x_n)$
$= f(x_1 \ldots 1 \ldots x_n) + 0$
$= f(x_1 \ldots 1 \ldots x_n)$

Aplicando recursivamente el teorema:

$$
\begin{aligned}
f(x_1 \ldots x_n) = {} & \sim x_1 . \ldots . \sim x_{n-1} . \sim x_n . \quad f(0 \ldots 00) + \\
& \sim x_1 . \ldots . \sim x_{n-1} . x_n . f(0 \ldots 01) + \\
& \sim x_1 . \ldots . x_{n-1} . \sim x_n . f(0 \ldots 10) + \\
& \sim x_1 . \ldots . x_{n-1} . x_n . \quad f(0 \ldots 11) + \\
& \ldots \\
& x_1 . \ldots . x_{n-1} . x_n . \quad f(1 \ldots 11)
\end{aligned}
$$

En el ejemplo:

x	y	f(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

$$
\begin{aligned}
f(x,y) = {} & \sim x \sim y . f(0,0) + \\
& \sim x y . f(0,1) + \\
& x \sim y . f(1,0) + \\
& x y . f(1,1) \\
= {} & \sim x \sim y . 0 + \\
& \sim x y . 1 + \\
& x \sim y . 1 + \\
& x y . 0 \\
= {} & \sim x y + x \sim y
\end{aligned}
$$

Observe que el teorema de Shannon es sólo válido para la lógica con {0,1}, no para cualquier álgebra de Boole.

Metodo general

1. Dada la tabla de verdad para **f**, para cada fila se crea un sumando del tipo:
   $[\sim]x_1 [\sim]x_2 \ldots [\sim]x_n$ f(fila), en donde x_i aparece negado si en esa fila $x_i = 0$
2. Se eliminan todos los sumandos que se multiplican por **0** (porque **x.0 = 0** y **x . 1 = x**)

Ejemplo:

x	y	g(x,y)	
0	0	0	
0	1	1	~xy +
1	0	1	x~y+
1	1	1	xy

g(x,y) = ~xy + x~y +xy

Pero claramente (!) **g(x,y) = x + y**

Demostración:
1. Tabla de **+** igual a tabla de **g**
2. Usando álgebra de Boole:
(xy = xy + xy)

~xy + xy + x~y = ~xy + xy + xy + x~y
= (~x + x) y + x (y + ~y) = y + x = x+ y

Con las siguientes dos reglas se puede llegar a simplificar cualquier fórmula del tipo suma de productos:
1. **$xy\sim z + x\sim y\sim z = x\sim z$** (no importa el número de variables)
2. **$x\sim yz + xyz + xy\sim z = xz + xy$** (porque **$= x\sim yz + xyz + xyz + xy\sim z$**)

Mapas de Karnaugh

Para visualizar mejor los productos que se pueden reducir se usan tablas de verdad re-ordenadas, llamadas mapas de Karnaugh.

Ejemplo:

x	y	z	f(x,y,z)	
0	0	0	1	~x~y~z +
0	0	1	1	~x~yz +
0	1	0	0	
0	1	1	1	~xyz +
1	0	0	1	x~y~z
1	0	1	0	
1	1	0	0	
1	1	1	0	

Mapa de Karnaugh:

! Fijense en el orden de yz !

x \ yz	0 0	0 1	1 1	1 0
0	1 ~x~y~z	1 ~x~yz	1 ~xyz	0
1	1 x~y~z	0	0	0

f(x,y,z) = ~y~z + ~xz

Principio fundamental: en una mapa de Karnaugh los términos que se originan de dos celdas adyacentes siempre difieren en una sola variable (Gray code).

Los mapas son circulares:

f(x,y,z) = x~z

x \ yz	0 0	0 1	1 1	1 0
0	0	0	0	0
1	1 x~y~z	0	0	1 xy~z

Un 1 se puede usar más de una vez en una simplificación, aprovechando que
~x~y~z = ~x~y~z + ~x~y~z

f(x,y,z) = ~y~z + ~x~z

x \ yz	0 0	0 1	1 1	1 0
0	1 ~x~y~z	0	0	1 ~xy~z
1	1 x~y~z	0	0	0

Se pueden agrupar bloques adyacentes: Agrupando 2^n **1** se eliminan n variables. Cada 1 debe ser cubierto al menos una vez.

f(x,y,z) = ~yz + yz = z

x \ yz	0 0	0 1	1 1	1 0
0	0	1 ~xy~z	1 ~xyz	0
1	0	1 xy~z	1 xyz	0

Por 4 variables:

! Fijense en el orden de xy !

f(x,y,z,w) = ~x~y~z +
~xyzw+
~y~w

xy \ zw	0 0	0 1	1 1	1 0
0 0	1	1	0	1
0 1	0	0	1	0
1 1	0	0	0	0
1 0	1	0	0	1

Observe que la función sin simplificar require 18 AND y 5 OR, mientras que al simplificarla queda en 6 AND y 2 OR.

Mapas de Karnaugh de cualquier tamaño

El problema se reduce a encontrar secuencias de los valores binarios en que los elementos adyacentes difieren en un solo bit (n-bit Gray codes).

S_1 = 0, 1
S_2 = 00, 01, 11, 10,
S_3 = 000, 001, 011, 010, 110, 111, 101, 100
...
informalmente: **S_{n+1} = 0.S_n + 1.invertir(S_n)**

Problema: Construir un circuito que calcule la suma de 3 números binarios de 1 bit **x,y,z**, dando como resultado un número de 2 bits **c, r** (acarreo y resultado) .

Tabla de verdad y mapas de Karnaugh:

x	y	z	c	r
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

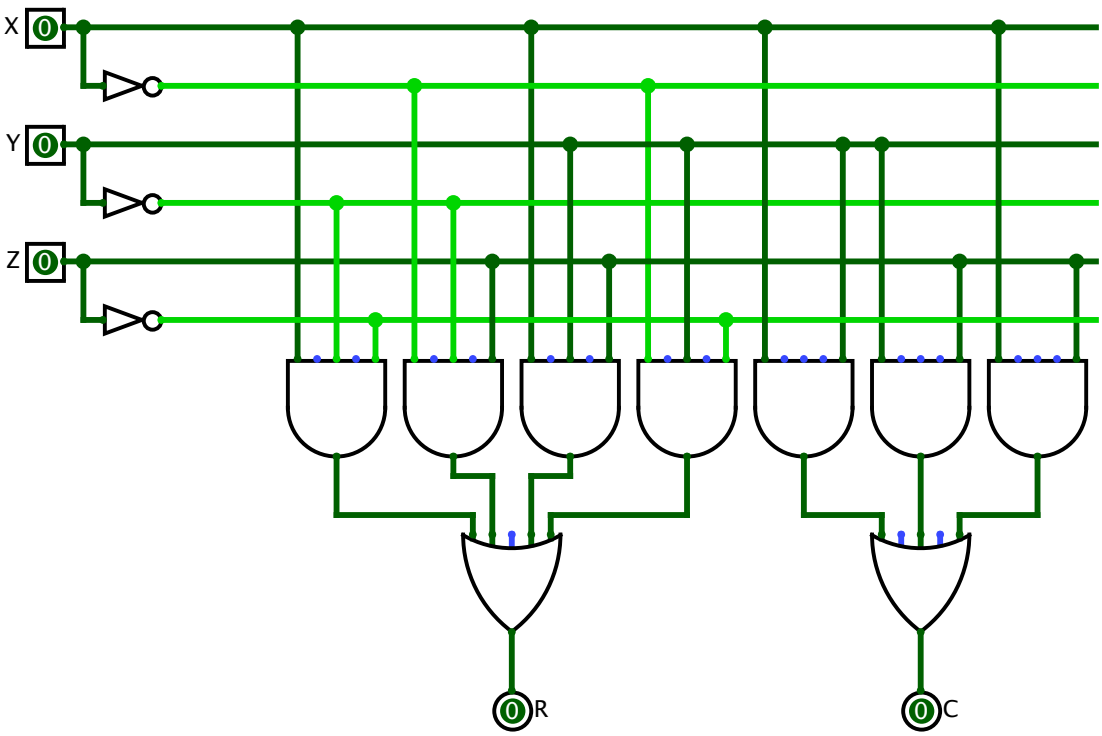
x \ yz	0 0	0 1	1 1	1 0
0	0	1	0	1
1	1	0	1	0

r = x~y~z + ~x~yz +xyz +~xy~z

x \ yz	0 0	0 1	1 1	1 0
0	0	0	1	0
1	0	1	1	1

c = xy + yz +xz

Diagrama circuital:



Propuesto: usando álgebra de Boole, muestre que si se remplazan tanto AND como OR por NAND en este circuito, el circuito sigue calculando la misma función.

Problema: Construir un sumador de 2 números binarios de 4 bits cada uno.

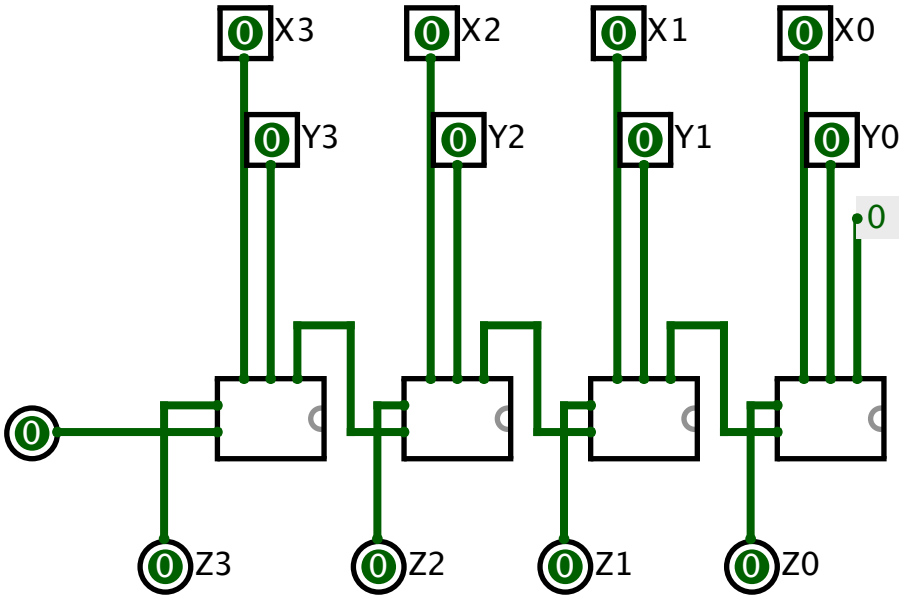
Solución 1: usando la metodología recién vista. ! Con 8 entradas, la tabla de verdad tiene 256 filas ! Demasiado largo.

c	0	0	1	0
x	1	1	0	1
y	1	0	0	1
1	0	1	1	0

Solución 2: usando el circuito sumador de 3 bits.

Se suma cada columna con un sumador de 3 bits.

Diagrama circuital:



<u>Funciones incompletamente especificadas</u>

Es frecuente que el valor de una función ante ciertas entradas pueda ser **0** o **1**. Eso se anota con **X** en la posición correspondiente de la mapa de Karnaugh:

xy \ zw	**0 0**	**0 1**	**1 1**	**1 0**
0 0	**1**	**0**	**1**	**1**
0 1	**0**	**1**	**1**	**1**
1 1	**X**	**X**	**X**	**X**
1 0	**1**	**1**	**X**	**X**

f(x,y,z,w) = x +
z +
yw +
~y~w

Las **X** se hacen valer **1** si esto es conveniente para obtener una fórmula más pequeña.