# Lógica digital y diseño de circuitos digitales

José Galaviz Casas

*Facultad de Ciencias, UNAM*

# Índice general

# 1. Lógica digital

## 1.1. Álgebra Booleana

El álgebra subyacente a las redes de conmutación de Shannon es un caso particular de la que desarrolló George Boole en 1854 en su trabajo[1] por dotar a la lógica proposicional de un lenguaje algebraico. Más tarde William Stanley Jevons y Edward Huntington le dieron al *álgebra booleana* la forma que hoy tiene. En particular Huntington[2] construyó el álgebra de Boole sobre un conjunto de axiomas en 1904.

### 1.1.1. Los postulados de Huntington

Un *álgebra booleana* $\mathbb{B}$ es una terna $(B, +, \cdot)$ en la que $B$ es un conjunto con al menos dos elementos distintos, $+$ denota el operador de suma y $\cdot$ el de producto. Si $x$, $y$ y $z$ son tres elementos cualesquiera de $B$, el álgebra booleana $\mathbb{B} = (B, +, \cdot)$ satisface axiomas.

1. Cerradura.

   **P1a.** $B$ es cerrado bajo la operación suma: $x + y \in B$.

   **P1b.** $B$ es cerrado bajo la operación producto: $x \cdot y \in B$.

2. Identidad.

   **P2a.** Existe un elemento *identidad de la suma* en $B$, denotado por 0, tal que: $x + 0 = 0 + x = x$.

   **P2b.** Existe un elemento *identidad del producto* en $B$, denotado por 1, tal que: $x \cdot 1 = 1 \cdot x = x$.

[1] Boole, George, *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*, Macmillan, 1854.

[2] Huntington, Edward V., "Sets of independent postulates for the algebra of logic", Transactions of the AMS, Vol. 5, 1904, pp. 288-309.

3. Conmutatividad.

   **P3a.** La suma es conmutativa: $x + y = y + x$.

   **P3b.** El producto es conmutativo: $x \cdot y = y \cdot x$.

4. Distributividad.

   **P4a.** El producto distribuye a la suma: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

   **P4b.** La suma distribuye al producto: $x + (y \cdot z) = (x + y) \cdot (x + z)$.

5. Complemento. Para cada $x \in B$ existe un único elemento $\overline{x} \in B$, llamado *el complemento* de $x$, tal que

   **P5a.** $x + \overline{x} = 1$

   **P5b.** $x \cdot \overline{x} = 0$

Por conveniencia consideraremos que, para obtener el complemento de un elemento cualquiera $x$, se aplica también una operación unaria (de un sólo argumento) y que bien puede verse denotada por la barra que ponemos sobre el argumento cuyo complemento queremos. Esto resulta conveniente porque tendremos entonces una idea clara de que, para construir un álgebra booleana, de acuerdo con los axiomas de Huntington, se requiere de tres operadores: suma, producto y complemento, aún cuando la definición formal sólo contempla dos. En 1933 Huntington reelaboró los axiomas del álgebra booleana para simplificarlos[3]. Es casi un deseo innato de los matemáticos obtener el mínimo número de axiomas que definen un sistema formal, así que Huntington encontró que con sólo tres axiomas y una operación (la suma) se podía tener un álgebra booleana: la conmutatividad, la asociatividad (que en los postulados de 1904, que aquí usaremos, no está contemplada) y la llamada ecuación de Huntington $(\overline{(\overline{x} + y)} + \overline{(\overline{x} + \overline{y})} = x)$.

Notará el lector que, además de que la suma, el producto y el complemento se pueden interpretar, respectivamente, como la disyunción, la conjunción y la negación de la lógica proposicional, en cuyo caso el 0 puede interpretarse como *falso* y el 1 como *verdadero*; también se pueden ver como la unión y la intersección y el complemento de la teoría de conjuntos elemental y entonces el 0 hace las veces del conjunto vacío y el 1 se puede interpretar como el

[3]Huntington, E.W. "New sets of independent postulates for the algebra of logic, with special reference to Whithead and Russell's Principia Mathematica", Transactions of the AMS, Vol. 35, 1933, pp. 274-304. Con una corrección posterior: "Boolean algebra. A correction.", ibid., Vol. 35, 1933, pp. 557-558.

conjunto universal. Así las cosas resulta que la teoría de conjuntos elemental, esa en la que el lector se ejercitó en el uso de diagramas de Venn en cursos elementales y la lógica proposicional son, ambas, álgebras booleanas. Además, claro, no son las únicas, este hecho le concedió al álgebra booleana un carácter más general e interesante.

El estudio de las álgebras booleanas trascendió el de la lógica proposicional y la teoría de conjuntos básica y se convirtió en un área por sí misma. A nosotros, sin embargo, nos interesa el caso mínimo, cuando el conjunto de elementos es dos, el caso que ocupó a Shannon. Pero antes necesitamos algunas herramientas más. Por simplicidad en adelante omitiremos el operador "$\cdot$" las más de las veces.

**Teorema 1** (Idempotencia.)**.** *[(a)]*

1. $x + x = x$
2. $x \cdot x = x$

*Demostración.*

$$\begin{aligned} x + x &= (x + x) \cdot 1 && P2b \\ &= (x + x)(x + \overline{x}) && P5a \\ &= x + x\overline{x} && P4b \\ &= x + 0 && P5b \\ &= x && P2a \end{aligned}$$

□

Por supuesto el inciso b del teorema se demuestra de manera completamente análoga: reemplazando los 1 por 0, los "+" por "$\cdot$" y justificando con el inciso complementario de cada postulado.

**Teorema 2** (Aniquilación)**.** *[(a)]*

1. $x + 1 = 1$
2. $x \cdot 0 = 0$

*Demostración.*

$$\begin{aligned} x+1 &= 1\cdot(x+1) && P2b \\ &= (x+\overline{x})(x+1) && P5a \\ &= x+\overline{x}\cdot 1 && P4b \\ &= x+\overline{x} && P2b \\ &= 1 && P5a \end{aligned}$$

□

**Teorema 3** (Absorción.)**.** *[(a)]*

1. $x + xy = x$
2. $x(x + y) = x$

*Demostración.*

$$\begin{aligned} x+xy &= x\cdot 1+xy && P2b \\ &= x(1+y) && P4a \\ &= x(y+1) && P3a \\ &= x\cdot 1 && Teo.2(a) \\ &= x && P2b \end{aligned}$$

□

**Teorema 4** (Involución.)**.** $\overline{(\overline{x})} = x$.

*Demostración.* Por el quinto postulado se tiene que las expresiones $x+\overline{x} = 1$ y $x \cdot \overline{x} = 0$ definen al complemento de un elemento cualquiera en $B$. Si reemplazamos $x$ por su complemento tendremos $\overline{x} + \overline{\overline{x}} = 1$ y $\overline{x} \cdot \overline{\overline{x}} = 0$. Combinándolas tenemos que:

$$x + \overline{x} = \overline{x} + \overline{\overline{x}}$$

y

$$x \cdot \overline{x} = \overline{x} \cdot \overline{\overline{x}}$$

De donde, considerando que el complemento de $x$ es único por el postulado 5, se tiene que:

$$x = \overline{\overline{x}}$$

□

**Teorema 5** (Eliminación)**.** *[(a)]*

1. $x + \overline{x}\ y = x + y$
2. $x\ (\overline{x} + y) = xy$

*Demostración.*

$$\begin{aligned} x + \overline{x}\ y &= x + xy + \overline{x}\ y && Teo.3(a) \\ &= x + 0 + xy + \overline{x}\ y && P2a \\ &= x + x\ \overline{x} + xy + \overline{x}\ y && P5b \\ &= xx + x\ \overline{x} + xy + \overline{x}\ y && Teo.1(b) \\ &= xx + xy + x\ \overline{x} + \overline{x}\ y && P3a \\ &= (x + y)(x + \overline{x}) && P4a \\ &= (x + y) \cdot 1 && P5a \\ &= x + y && P2b \end{aligned}$$

□

**Teorema 6** (Asociatividad.)**.** *[(a)]*

1. $x + (y + z) = (x + y) + z$
2. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

*Demostración.* Sean $I = x + (y + z)$ y $D = (x + y) + z$, los respectivos lados izquierdo y derecho de la expresión a comprobar. Nótese que:

$$x \cdot I = x \cdot [x + (y + z)] = x \cdot (x + E)$$

donde $E = y + z$. Esa última expresión es de la forma $x(x + y)$, el lado izquierdo del inciso (b) del teorema 3 (absorción). Así que:

$$\begin{aligned} x \cdot [x + (y + z)] &= x \cdot I \\ &= x \quad Teo.3(b) \end{aligned} \tag{1.1}$$

Por otra parte:

$$\begin{aligned} x \cdot [(x + y) + z] &= x \cdot D \\ &= x \cdot (x + y) + xz && P4a \\ &= x + xz && Teo.3(b) \\ &= x && Teo.3(a) \end{aligned} \tag{1.2}$$

Sintetizando 1.1 y 1.2:

$$x \cdot [x + (y + z)] = xI = x = xD = x \cdot [(x + y) + z] \tag{1.3}$$

Además:

$$\begin{aligned} \overline{x} \cdot [x + (y + z)] &= \overline{x} \cdot I \\ &= \overline{x}x + \overline{x}(y + z) && P4a \\ &= x\overline{x} + \overline{x}(y + z) && P3b \\ &= 0 + \overline{x}(y + z) && P5b \\ &= \overline{x}(y + z) && P2a \end{aligned} \tag{1.4}$$

Y también:

$$\begin{aligned} \overline{x} \cdot [(x + y) + z] &= \overline{x} \cdot D \\ &= \overline{x}(x + y) + \overline{x}z && P4a \\ &= (\overline{x}x + \overline{x}y) + \overline{x}z && P4a \\ &= (x\overline{x} + \overline{x}y) + \overline{x}z && P3b \\ &= (0 + \overline{x}y) + \overline{x}z && P5b \\ &= \overline{x}y + \overline{x}z && P2a \\ &= \overline{x}(y + z) && P4a \end{aligned} \tag{1.5}$$

Sintetizando 1.4 y 1.5:

$$\overline{x} \cdot [x + (y + z)] = \overline{x}I = \overline{x}(y + z) = \overline{x}D = \overline{x} \cdot [(x + y) + z] \tag{1.6}$$

Finalmente:

$$\begin{aligned} I &= I \cdot 1 \\ &= I(x + \overline{x}) && P5a \\ &= Ix + I\overline{x} && P4a \\ &= xI + \overline{x}I && P3b \\ &= xD + \overline{x}I && 1.3 \\ &= xD + \overline{x}D && 1.6 \\ &= Dx + D\overline{x} && P3b \\ &= D(x + \overline{x}) && P4a \\ &= D \cdot 1 && P5a \\ &= D && P2b \end{aligned}$$

Con lo que $x + (y + z) = (x + y) + z$. □

**Teorema 7** (Consenso.)**.** *[(a)]*

1. $xy + \overline{x}z + yz = xy + \overline{x}z$
2. $(x + y)(\overline{x} + z)(y + z) = (x + y)(\overline{x} + z)$

*Demostración.*

$$\begin{aligned} xy + \overline{x}z + yz &= xy + \overline{x}z + yz \cdot 1 && P2b \\ &= xy + \overline{x}z + yz \cdot (x + \overline{x}) && P5a \\ &= xy + \overline{x}z + xyz + \overline{x}yz && P4a \\ &= xy(1 + z) + \overline{x}z(1 + y) && P4a \\ &= xy \cdot 1 + \overline{x}z \cdot 1 && Teo.2 \\ &= xy + \overline{x}z && P2b \end{aligned}$$

□

**Teorema 8** (Leyes de De Morgan.)**.** *[(a)]*

1. $\overline{x + y} = \overline{x} \cdot \overline{y}$
2. $\overline{x \cdot y} = \overline{x} + \overline{y}$

*Demostración.* Por el postulado 5 sabemos que, para todo elemento $x$ existe un único elemento $\overline{x}$, tal que: $x + \overline{x} = 1$ y $x \cdot \overline{x} = 0$. Así que para probar que $\overline{x} \cdot \overline{y}$ es el complemento de $x + y$, basta probar que $(x + y) + \overline{x} \cdot \overline{y} = 1$ y que $(x + y) \cdot (\overline{x} \cdot \overline{y}) = 0$.

En efecto:

$$\begin{aligned} (x + y) + \overline{x}\ \overline{y} &= [(x + y) + \overline{x}] \cdot [(x + y) + \overline{y}] && P4b \\ &= [(y + x) + \overline{x}] \cdot [(x + y) + \overline{y}] && P3a \\ &= [y + (x + \overline{x})] \cdot [x + (y + \overline{y})] && Teo.6(a) \\ &= (y + 1) \cdot (x + 1) && P5a \\ &= 1 \cdot 1 && Teo.2(a) \\ &= 1 && P2b \end{aligned}$$

También:

$$\begin{aligned}
(x+y)\cdot(\overline{x}\ \overline{y}) &= (\overline{x}\ \overline{y})\cdot(x+y) && P3b\\
&= (\overline{x}\ \overline{y})x + (\overline{x}\ \overline{y})y && P4a\\
&= (\overline{y}\ \overline{x})x + (\overline{x}\ \overline{y})y && P4a\\
&= \overline{y}(\overline{x}\ x) + \overline{x}(\overline{y}\ y) && Teo.6(b)\\
&= \overline{y}(x\ \overline{x}) + \overline{x}(y\ \overline{y}) && P3b\\
&= \overline{y}\cdot 0 + \overline{x}\cdot 0 && P5b\\
&= 0 + 0 && Teo.2(b)\\
&= 0 && Teo.1(a)
\end{aligned}$$

□

## 1.2. Funciones de conmutación

El conjunto $B$ sobre el que se definen las operaciones del álgebra booleana tiene, sabemos, al menos dos elementos. Cuando tiene exactamente ese número, estos elementos son los que en los axiomas aparecen etiquetados como "0" y "1". Podemos ahora pensar entonces en funciones cuyas variables independientes puedan adquirir valores en ese conjunto mínimo y calculen, a partir de ello, un valor también en ese conjunto. A estas, un caso particular del álgebra booleana, les llamaremos *funciones de conmutación*.

¿Cómo se puede expresar una función? Bueno, estamos acostumbrados a hacerlo mediante lo que llamamos la *regla de correspondencia*: una expresión analítica que proporciona el valor de la variable dependiente, dado el de la o las variables independientes. Pero eso es porque solemos trabajar con los números reales y en ese conjunto, por ser infinito y continuo, es imposible pensar en otra forma de hacerlo. Pero en estricto sentido una función es una relación entre dos conjuntos: cada elemento de uno de ellos, llamado dominio, se asocia con uno y sólo uno del otro conjunto, llamado contradominio. Así las cosas se puede pensar en hacer una tabla que diga explícitamente, para cada elemento del dominio, cuál elemento del contradominio le es asociado por la función. Claro esto sólo es posible si realmente podemos listar todos los elementos del dominio, es decir, si éste es finito. Si el conjunto dominio de nuestra álgebra booleana contiene el mínimo número de elementos, es decir dos, entonces cualquier función de $n$ variables independientes puede tener hasta $2^n$ posibles combinaciones de dichas variables de entrada y entonces podemos hacer una tabla que muestre explícitamente cuál es el valor que

adquiere la función dada cualquiera de las $2^n$ posibles entradas. En el álgebra de las funciones de conmutación tenemos entonces dos maneras de especificar una función: con una expresión algebraica o con una tabla análoga a las tablas de verdad que solíamos usar en el cálculo proposicional.

Por cierto, si pensamos en funciones de conmutación de $n$ variables, entonces, cómo dijimos, las variables pueden adquirir $2^n$ valores diferentes. Por cada una de estas combinaciones la función puede arrojar uno de dos valores posibles, así que en total hay $2^{2^n}$ funciones de conmutación de $n$ variables.

Resulta además bastante sencillo pasar de la representación de una función mediante su tabla de verdad a su equivalente regla de correspondencia.

**Ejemplo 1.2.1.** Observemos la función $F(x_0, x_1, x_2)$ representada en la tabla 1.1. Para obtener la expresión analítica de esta función debemos fijar nuestra atención en el primer renglón de la tabla en el que $F$ vale 1, a saber, el primer renglón de la tabla. Si sabemos que en ese renglón $x_2 = x_1 = x_0 = 0$, ¿qué deberíamos hacerle a las variables para obtener justo ese y sólo ese 1 operándolas con un AND? Cada variable vale 0, y basta con un cero en un AND para que el resultado sea 0, así que necesariamente deberemos negar todas las variables, es decir, la función $m_0 = \overline{x_0}\ \overline{x_1}\ \overline{x_2}$ tiene la propiedad de valer 1 en un único sitio en la tabla, justo cuando $x_2 = x_1 = x_0 = 0$, que es lo que buscamos.

Podemos luego hacernos la misma pregunta para cada renglón de la tabla en el que $F$ valga 1 y obtendremos cada uno de los productos (AND) que lo generan. Bastaría luego con hacer el OR de todos estos productos para obtener una expresión analítica de $F$.

Lo que estamos haciendo es, implícitamente, encontrar cuales deben ser los valores de los coeficientes $c_i$ en la expresión:

$$\begin{aligned} F(x_0, x_1, x_2) &= c_0\overline{x_2}\ \overline{x_1}\ \overline{x_0} + c_1\overline{x_2}\ \overline{x_1}\ x_0 + \\ &\quad c_2\overline{x_2}\ x_1\ \overline{x_0} + c_3\overline{x_2}\ x_1\ x_0 + \\ &\quad c_4 x_2\ \overline{x_1}\ \overline{x_0} + c_5 x_2\ \overline{x_1}\ x_0 + \\ &\quad c_6 x_2\ x_1\ \overline{x_0} + c_7 x_2\ x_1\ x_0 \end{aligned} \tag{1.7}$$

donde $c_i \in \{0, 1\}$. Cada coeficiente indica, realmente, si el $i$-ésimo producto que le corresponde debe ser considerado o no para construir la función.

Con base en lo expuesto, la función $F$ se puede expresar como:

$$\begin{aligned} F(x_0, x_1, x_2) &= \overline{x_2}\ \overline{x_1}\ \overline{x_0} + \overline{x_2}x_1\overline{x_0} + \overline{x_2}\ x_1x_0 + \\ &\quad x_2\overline{x_1}x_0 + x_2x_1\overline{x_0} + x_2x_1x_0 \end{aligned} \tag{1.8}$$

| $\mathbf{x_2}$ | $\mathbf{x_1}$ | $\mathbf{x_0}$ | **F** |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Tabla 1.1: Tabla de verdad de una función de conmutación.

### 1.2.1. Suma de productos o producto de sumas

Podríamos, de acuerdo con lo que hicimos en el ejemplo, expresar cualquier función de conmutación de tres variables, como la suma de algunos de los 8 productos que se pueden formar tomando cada variable o su negación. En la literatura a estos productos se les suele llamar *mintérminos*. En la tabla 1.2 se muestran todos los mintérminos ($m_i$) de tres variables.

**Ejemplo 1.2.2.** La función del ejemplo puede expresarse como suma de mintérminos como:

$$F(x_0, x_1, x_2) = m_0 + m_2 + m_3 + m_5 + m_6 + m_7 = \sum(0, 2, 3, 5, 6, 7)$$

Por supuesto no hay una única expresión analítica para la regla de correspondencia que defina una función. De hecho podemos hacer lo que podríamos llamar el *procedimiento dual* del que llevamos a cabo: en vez de ir agregando productos a una suma de tal forma que vayamos generando cada uno de los 1's presentes en la tabla, podemos tratar de generar los 0's.

Cada vez que añadimos un mintérmino a una expresión, forzamos a que, en el lugar de la tabla que corresponde con ese mintérmino la función valdrá 1. Además cada mintérmino aporta uno y sólo un 1. La suma de mintérminos,

| Mintérmino | Expresión |
|---|---|
| $m_0$ | $\overline{x_2}\ \overline{x_1}\ \overline{x_0}$ |
| $m_1$ | $\overline{x_2}\ \overline{x_1}\ x_0$ |
| $m_2$ | $\overline{x_2}\ x_1\ \overline{x_0}$ |
| $m_3$ | $\overline{x_2}\ x_1\ x_0$ |
| $m_4$ | $x_2\ \overline{x_1}\ \overline{x_0}$ |
| $m_5$ | $x_2\ \overline{x_1}\ x_0$ |
| $m_6$ | $x_2\ x_1\ \overline{x_0}$ |
| $m_7$ | $x_2\ x_1\ x_0$ |

Tabla 1.2: Lista de los mintérminos posibles de tres variables.

literalmente, suma las contribuciones de cada uno para generar todos los 1 de la tabla.

En pensamiento dual corresponde a considerar un término hecho por la suma de cada variable o su negación de tal forma que sólo valga 0 en un lugar de la tabla, en todos los demás lugares la suma valdrá 1. Si tenemos todos los términos necesarios para generar cada uno de los ceros de la tabla y hacemos el producto de ellos, entonces la tabla resultante tendrá ceros exactamente en los lugares asociados con cada término y unos en el resto. A estos términos se les denomina *maxtérminos*. Los maxtérminos de 3 variables aparecen listados en la tabla 1.3.

**Ejemplo 1.2.3.** La función que estamos usando como ejemplo puede escribirse también como:

$$\begin{aligned} F(x_0, x_1, x_2) &= (x_2 + x_1 + \overline{x_0})(\overline{x_2} + x_1 + x_0) \\ &= M_1 \cdot M_4 \\ &= \prod(1, 4) \end{aligned}$$

A la expresión de la función como suma de productos (mintérminos) se le denomina *Forma Canónica Normal Disjuntiva* o *CDNF* por sus siglas

| Maxtérmino | Expresión |
|---|---|
| $M_0$ | $x_2 + x_1 + x_0$ |
| $M_1$ | $x_2 + x_1 + \overline{x_0}$ |
| $M_2$ | $x_2 + \overline{x_1} + x_0$ |
| $M_3$ | $x_2 + \overline{x_1} + \overline{x_0}$ |
| $M_4$ | $\overline{x_2} + x_1 + x_0$ |
| $M_5$ | $\overline{x_2} + x_1 + \overline{x_0}$ |
| $M_6$ | $\overline{x_2} + \overline{x_1} + x_0$ |
| $M_7$ | $\overline{x_2} + \overline{x_1} + \overline{x_0}$ |

Tabla 1.3: Lista de los maxtérminos posibles de tres variables.

en inglés y a el equivalente como producto de sumas (maxtérminos) *Forma Canónica Normal Conjuntiva* o *CCNF*. Por supuesto una de ellas puede siempre ser transformada en la otra usando las leyes de De Morgan y, claro está esto puede hacerse algorítmicamente, sólo que en principio requiere una cantidad de pasos que depende exponencialmente del número de variables involucradas. Es por eso que el conocido problema SAT, cuya entrada es una expresión booleana en forma conjuntiva, es intratable, a pesar de que al poner la expresión en forma disyuntiva resulta trivial decidir si es satisfactible o no.

## 1.3. Minimización de funciones de conmutación

### 1.3.1. Manipulación algebraica

Existen entonces varias posibles expresiones para la misma función, pero ciertamente debe haber una que es mínima en el sentido de tener la menor cantidad posible de operaciones. Podemos tratar de obtenerla mediante transformaciones sucesivas usando los axiomas y teoremas del álgebra booleana.

**Ejemplo 1.3.1.** Podemos simplificar la expresión para la función $F$ del ejemplo 1.2.1 haciendo lo siguiente.

$$\begin{aligned}
F(x_0, x_1, x_2) &= \overline{x_2}\ \overline{x_1}\ \overline{x_0} + \overline{x_2} x_1 \overline{x_0} + \overline{x_2}\ x_1 x_0 + \\
&\quad x_2 \overline{x_1} x_0 + x_2 x_1 \overline{x_0} + x_2 x_1 x_0 \\
&= \overline{x_2}\ \overline{x_1}\ \overline{x_0} + \overline{x_2}\ x_1 x_0 + \\
&\quad x_1 \overline{x_0}(x_2 + \overline{x_2}) + x_2 x_0 (x_1 + \overline{x_1}) && P4a \\
&= \overline{x_0}(x_1 + \overline{x_1}\ \overline{x_2}) + x_0(x_2 + \overline{x_2}\ x_1) && P5a \text{ y } P4a \\
&= \overline{x_0}(x_1 + \overline{x_2}) + x_0(x_2 + x_1) && Teo.5(a) \\
&= \overline{x_0}\ x_1 + \overline{x_0}\ \overline{x_2} + x_0 x_2 + x_0 x_1 && P4a \\
&= x_1(x_0 + \overline{x_0}) + \overline{x_0}\ \overline{x_2} + x_0 x_2 && P4a \\
&= x_1 + \overline{x_0}\ \overline{x_2} + x_0 x_2 && P5a
\end{aligned}$$

Llevar a cabo este proceso no es una labor fácil, en general dependemos de nuestra habilidad para notar qué axioma o teorema del álgebra booleana se puede usar, es truculento. En general, obtener la mínima expresión de una formula booleana dada es un problema NP-difícil. Pero podemos facilitarnos la vida un poco si reflexionamos acerca de lo que acabamos de hacer en el ejemplo. Nótese que siempre que podemos eliminar una variable (o su negación) de algunos términos es cuando esos términos tienen una subexpresión común que se puede *factorizar* y la variable que se elimina aparece en algunos de los términos afirmada y en otros negada. La variable se elimina gracias al hecho de que $x + \overline{x} = 1$ o $x\ \overline{x} = 0$. Es decir, siempre que tenemos una subexpresión común asociada con una variable que *cambia* entonces la variable desaparece. Si tenemos una función que contiene a los mintérminos 4 ($x_2\ \overline{x_1}\ \overline{x_0}$) y 6 ($x_2\ x_1\ \overline{x_0}$), por ejemplo, entonces tenemos como subexpresión común factorizable: $x_2\ \overline{x_0}$ y la variable $x_1$ cambia de afirmada a negada entre ambos términos, así que se eliminará dejando sólo la subexpresión común.

Es conveniente notar que, si consideramos la expresión binaria de los índices de los mintérminos, lo que equivale a considerar las variables afirmadas (1 en la expresión binaria del índice) y las negadas (0 en la expresión), entonces necesitamos considerar mintérminos cuyos índices, en binario, difieran en un bit. La subexpresión común factorizable que se preserva corresponde a aquellos bits que son comunes a ambos índices. La variable que se elimina es la que cambia de 0 a 1. En el ejemplo del párrafo anterior, los índices eran $6_{10} = 110_2$ y $4_{10} = 100_2$. Denotando con $*$ el sitio que cambia entre ambos mintérminos: $1 * 0$. Los mintérminos 4 y 6 se simplifican como $x_2\ \overline{x_0}$,

$x_2$ corresponde con el 1 y $\overline{x_0}$ con el cero. La variable $x_1$ quedó en el lugar del $*$, así que se pierde.

Podemos pensar en aprovechar esta observación para tratar de simplificar la expresión de una función tratando de distinguir visualmente las subexpresiones comunes y las variables que cambian.

### 1.3.2. Mapas de Karnaugh

Entre 1952 y 1953 Edward Veitch[4] y Maurice Karnaugh[5] inventaron un método para simplificar funciones de conmutación basado en el principio descrito y aprovechando la habilidad humana para reconocer patrones. El método, cabe decirlo, es una heurística, es decir, en general no tenemos garantías de obtener la expresión mínima para la función, sobre todo si tiene muchas variables, porque entonces los patrones son menos evidentes.

El método consiste en acomodar la tabla de verdad de la función en un mapa cuadriculado, de tal forma que en el mapa queden contiguas las entradas de la tabla cuyo índice difiere en sólo un bit y posteriormente agrupar las celdas de acuerdo con ciertas reglas. Por supuesto en un arreglo bidimensional, conforme el número de variables crece, se vuelve difícil hacer que sean contiguas las celdas que difieren de otras en sólo un bit. Se comienza considerando que las celdas en el extremo derecho son contiguas a las del extremo izquierdo y las de arriba a las de abajo, luego se complica. Describiremos aquí las que se deben usar para obtener sumas de productos, por supuesto si se usan las reglas duales, se obtendrán productos de sumas.

- Los grupos no pueden incluir celdas con valor 0.
- Los grupos pueden crecer horizontal o verticalmente, pero no en diagonal.
- Los grupos deben contener un número de celdas de la forma $2^k$.
- Cada grupo debe ser tan grande como sea posible.
- Toda celda con valor 1, debe estar en, al menos un grupo.
- Los grupos pueden tener intersección no vacía.

[4] Veitch, Edward W.,."A Chart Method for Simplifying Truth Functions", *Proceedings of the 1952 ACM Annual Meeting*, 1952, pp. 127–133.

[5] Karnaugh, Maurice, "The Map Method for Synthesis of Combinational Logic Circuits". *Transactions of the American Institute of Electrical Engineers*, Vol. 72, No. 9, noviembre de 1953, pp. 593–599.

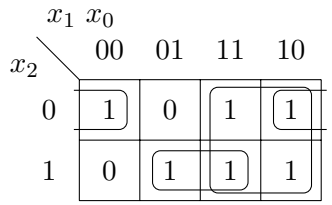


Figura 1.1: Mapa de Karnaugh para la función del ejemplo.

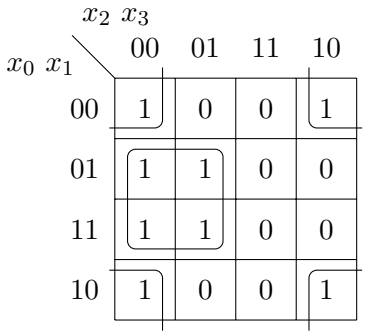


Figura 1.2: Mapa de Karnaugh para la $F_2$.

- Se debe procurar minimizar el número de grupos.

Ciertamente es más fácil comprender esto mediante un ejemplo.

**Ejemplo 1.3.2.** El mapa de Karnaugh para la función $F$ del ejemplo, se muestra en la figura 1.1. Cómo se puede apreciar en el mapa:

$$F(x_0, x_1, x_2) = \overline{x_2}\ \overline{x_0} + x_1 + x_2 x_0$$

que es justamente la expresión obtenida previamente por manipulación algebraica.

**Ejemplo 1.3.3.** Sea $F_2(x_0, x_1, x_2, x_3)$ una función descrita por la tabla de verdad 1.4.

El mapa de Karnaugh para $F_2$ se puede ver en la figura 1.2, de allí se desprende que:

$$F_2(x_0, x_1, x_2, x_3) = \overline{x_1}\ \overline{x_3} + x_1\ \overline{x_2}$$

| $\mathbf{x_0}$ | $\mathbf{x_1}$ | $\mathbf{x_2}$ | $\mathbf{x_3}$ | $\mathbf{F_2}$ | $\mathbf{x_0}$ | $\mathbf{x_1}$ | $\mathbf{x_2}$ | $\mathbf{x_3}$ | $\mathbf{F_2}$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

Tabla 1.4: Tabla de verdad de la función $F_2$.

Cuando el número de variables crece, el mapa de Karnaugh ya no resulta tan práctico. Los patrones ya no son tan evidentes. La noción de celdas contiguas en términos del código binario ya no se mapea en una contigüidad espacial. Hay, de hecho, dos alternativas para acomodar una tabla de 5 variables, Hay quienes prefieren una o la otra.

**Ejemplo 1.3.4.** En la figura 1.3 se muestran las dos alternativas usuales para un mapa de Karnaugh de 5 variables. Como puede observarse los patrones que dan lugar a simplificaciones se ven diferentes en ambos mapas.

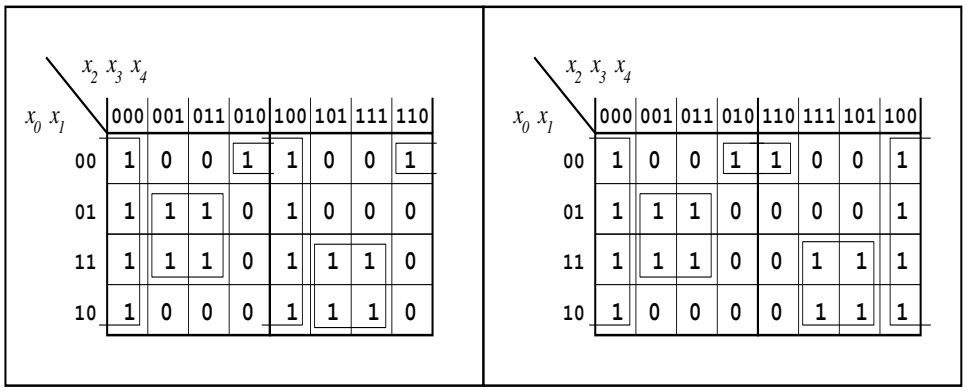


Figura 1.3: Alternativas de mapa de 5 variables..

La función representada es:

$$\begin{aligned} F_3(x_0,x_1,x_2,x_3,x_4) \;=\; & \overline{x_0}\;\overline{x_1}\;\overline{x_2}\;\overline{x_3}\;\overline{x_4} + \overline{x_0}\;\overline{x_1}\;\overline{x_2}\;x_3\;\overline{x_4} + \\ & \overline{x_0}\;\overline{x_1}\;x_2\;x_3\;\overline{x_4} + \overline{x_0}\;\overline{x_1}\;x_2\;\overline{x_3}\;\overline{x_4} + \\ & \overline{x_0}\;x_1\;\overline{x_2}\;\overline{x_3}\;\overline{x_4} + \overline{x_0}\;x_1\;\overline{x_2}\;\overline{x_3}\;x_4 + \\ & \overline{x_0}\;x_1\;\overline{x_2}\;x_3\;x_4 + \overline{x_0}\;x_1\;x_2\;\overline{x_3}\;\overline{x_4} + \\ & x_0\;x_1\;\overline{x_2}\;\overline{x_3}\;\overline{x_4} + x_0\;x_1\;\overline{x_2}\;\overline{x_3}\;x_4 + \\ & x_0\;x_1\;\overline{x_2}\;x_3\;x_4 + x_0\;x_1\;x_2\;x_3\;x_4 + \\ & x_0\;x_1\;x_2\;\overline{x_3}\;x_4 + x_0\;x_1\;x_2\;\overline{x_3}\;\overline{x_4} + \\ & x_0\;\overline{x_1}\;\overline{x_2}\;\overline{x_3}\;\overline{x_4} + x_0\;\overline{x_1}\;x_2\;x_3\;x_4 + \\ & x_0\;\overline{x_1}\;x_2\;\overline{x_3}\;x_4 + x_0\;\overline{x_1}\;x_2\;\overline{x_3}\;\overline{x_4} \end{aligned}$$

De acuerdo con el mapa:

$$F_3(x_0,x_1,x_2,x_3,x_4) = \overline{x_3}\;\overline{x_4} + \overline{x_0}\;\overline{x_1}\;x_3\;\overline{x_4} + x_1\;\overline{x_2}\;x_4 + x_0\;x_2\;x_4$$

### 1.3.3. Método de Quine–McCluskey

Entre 1952 y 1956 Willard Van Orman Quine y Edward McCluskey elaboraron un método algorítmico para la minimización de una función de conmutación. Dado que es un algoritmo, se puede programar y entonces se eliminan los problemas inherentes al método de Karnaugh: la expresión obtenida es ciertamente mínima y correcta y podemos pensar en minimizar funciones de más de 5 variables, cuando la capacidad humana para reconocer patrones se ve seriamente afectada por el orden del mapa de Karnaugh. Podemos olvidarnos pues, de los posibles errores humanos.

El algoritmo encuentra lo que se denominan los *implicantes primos* de la expresión original dada en mintérminos. Se parte de una tabla en la que aparecen los renglones de la tabla de verdad de la función en los que esta vale 1, pero agrupados por su *peso de Hamming*, o sea por el número de unos en su expresión binaria. Se procede luego como sigue:

1. Se empata cada miembro de un grupo con todos y cada uno de los del grupo contiguo siguiente con los que difiera en el valor de una sola variable.

2. Cada pareja genera un nuevo patrón de empate que es registrado en una nueva entrada de la que llamaremos tabla siguiente. En este patrón se elimina aquella variables cuyo valor cambió en los términos que se empataron.

3. Se marcan como usadas aquellas entradas de la tabla actual que se pudieron empatar.

4. Si la tabla siguiente tiene sólo un patrón o patrones imposibles de empatar entonces ya se ha terminado el proceso, si no es así la tabla actual se reemplaza por la tabla nueva y se regresa al primer paso.

Los patrones que quedan al final y aquellos que nunca fueron marcados durante el proceso definen los implicantes primos de la función.

**Ejemplo 1.3.5.** Recordemos la función $F$ de nuestro ejemplo:

$$F(x_2, x_1, x_0) = \sum(0, 2, 3, 5, 6, 7)$$

Las tablas del proceso de Quine–McCluskey son las siguientes, se han marcado con un "*" los patrones usados.

| **m** | $x_2$ | $x_1$ | $x_0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | * |
| 2 | 0 | 1 | 0 | * |
| 3 | 0 | 1 | 1 | * |
| 5 | 1 | 0 | 1 | * |
| 6 | 1 | 1 | 0 | * |
| 7 | 1 | 1 | 1 | * |

| **m** | $x_2$ | $x_1$ | $x_0$ | |
|---|---|---|---|---|
| 0,2 | 0 | – | 0 | |
| 2,3 | 0 | 1 | – | * |
| 2,6 | – | 1 | 0 | * |
| 3,7 | – | 1 | 1 | * |
| 5,7 | 1 | – | 1 | |
| 6,7 | 1 | 1 | – | * |

| **m** | $x_2$ | $x_1$ | $x_0$ |
|---|---|---|---|
| 2,3,6,7 | – | 1 | – |
| 2,6,3,7 | – | 1 | – |

Como se puede observar los únicos patrones no usados son el primero y el quinto de la segunda tabla y el único de la última. Estos patrones corresponden, respectivamente a:

$$F(x_2, x_1, x_0) = \overline{x_2}\ \overline{x_0} + x_2\ x_0 + x_1$$

lo que coincide con nuestras expresines previas para esa función.

Por ser algorítmico, el método de Quine–McCluskey fue utilizado en el software usado para diseñar hardware. Pero dado que el problema de minimizar una expresión booleana es, en general intratable, desde hace tiemp se prefiere una heurística bastante efectiva comunmente llamada *Espresso*, desarrollada en los ochenta por Robert Brayton, de IBM y más tarde por Richard Rudell y que resulta mucho más eficiente en términos de consumo de recursos y, aunque no garantiza el mínimo global, ofrece muy buenas aproximaciones.

# 2. Circuitos digitales

## 2.1. Representación gráfica de circuitos

Ahora estamos en condiciones de representar gráficamente los circuitos digitales cuyas expresiones o tablas de verdad poseemos y simplificamos. Necesitaremos mínimamente para ello, símbolos que representen a los diferentes operadores del álgebra de las funciones de conmutación. A saber: el NOT, el AND y el OR.

En la figura 2.1 están representadas cada una de los símbolos que usaremos. A el dispositivo que implementa cada uno de estos operadores le llamaremos *compuerta*. Por cierto se ha incluido en la figura la representación de una compuerta asociada al XOR, una operación que no es de las que podríamos llamar *básicas*, ya que el XOR de $A$ y $B$, denotado con el operador $\oplus$ se define como:

$$A \oplus B = A\ \overline{B} + \overline{A}\ B = (A + B)\ (\overline{A} + \overline{B})$$

La representación del XOR se ha incluido por ser muy usual, la tabla de verdad es la 2.1.

Estos símbolos gráficos básicos admiten modificadores, una pequeña *bolita* en un cable, se entrada o salida de una compuerta indica que la se;al que viaja por ese cable se ha negado. De hecho el símbolo del negador era originalmente el de un rectificador y al añadirle la bolita se convierte en inversor. Suelen añadirse también líneas de entrada a las compuertas, así podemos tener OR o AND de tres o cuatro entradas.

**Ejemplo 2.1.1.** El circuito para calcular la función de la tabla 1.1 se muestra en la figura 2.2.

**Ejemplo 2.1.2.** El circuito para calcular la función de la tabla 1.4 se muestra en la figura 2.3.

| A | B | A⊕B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Tabla 2.1: Tabla de verdad del OR exclusivo, disyunción exclusiva o XOR.

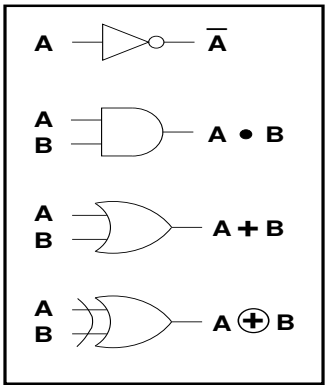


Figura 2.1: Representación gráfica de las principales compuertas. De arriba hacía abajo: NOT, AND, OR y XOR. Esta última no es indispensable, pero sí muy conveniente.

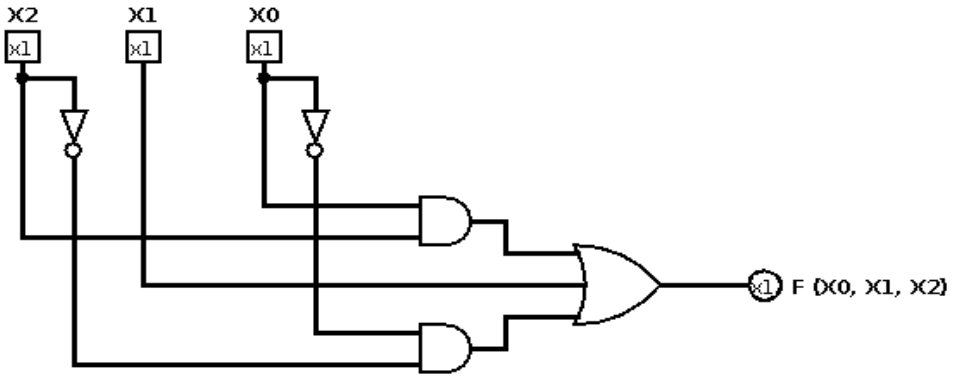


Figura 2.2: Circuito para calcular la función de la tabla 1.1.

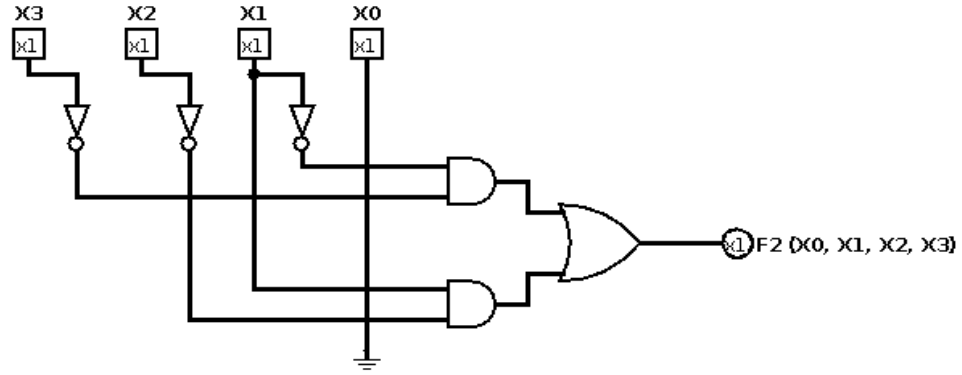


Figura 2.3: Circuito para calcular la función de la tabla 1.4.

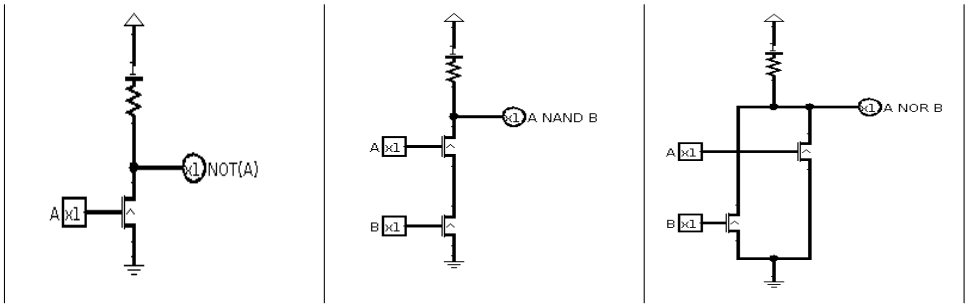


Figura 2.4: Compuertas NOT, NAND y NOR elaboradas con transistores NMOS, tipo NPN.

## 2.2. Conjunto de operadores completo

Como cualquier función de conmutación puede escribirse con base en los operadores AND, OR y NOT, se dice que éste es un *conjunto de operadores completo*. Pero no es el único.

Es posible, por ejemplo, escribir cualquier función de conmutación usando solamente NAND o NOR, cuyas tablas de verdad se muestran en la tabla 2.2. Por supuesto ambos operadores se representan con la respectiva compuerta pero negando su salida con una bolita. Para demostrarlo basta observar la figura 2.5. Formalmente bastaría aplicar las leyes de DeMorgan. La razón por la que se eligen algunos conjuntos completos distintos de la terna AND, OR, NOT es porque la compuerta elemental usando transisitores en serie puede no ser un AND, sino un NAND. En la figura 2.4 se muestran las compuertas más elementales usando transistores NMOS tipo NPN.

| $A$ | $B$ | $\overline{AB}$ | $\overline{A+B}$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

Tabla 2.2: Tabla de verdad del NAND y el NOR.

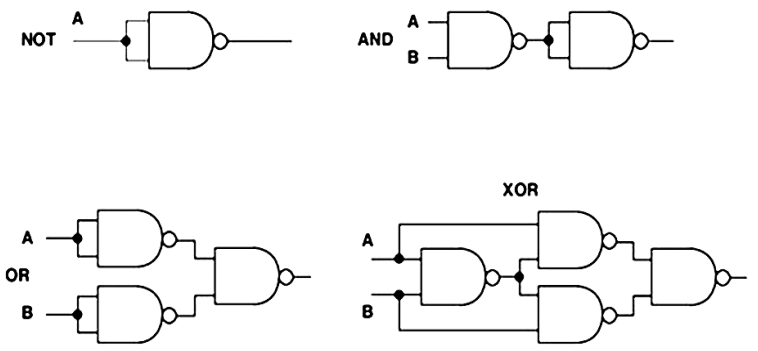


Figura 2.5: Las compuertas NOT, AND, OR y XOR expresadas en función de la NAND.

**Ejemplo 2.3.1.**

| A | B | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Tabla 2.3: Tabla de verdad de un semisumador.

## 2.3. Lógica combinacional

Los circuitos que hemos construido hasta ahora poseen la característica de que su salida está completamente determinada por los valores de las líneas de entrada. A este tipo de circuitos se les denomina *combinacionales* y su diseño consiste, esencialmente, en los pasos que hemos ido llevando a cabo:

1. Para cada combinación de valores de las variables de entrada, determinar la salida. Esto es, construir la tabla de verdad del circuito.

2. Minimizar la expresión de la función booleana calculada por el circuito.

3. Diseñar el circuito que la calcula o, alternativamente, especificar el circuito en algún lenguaje de descripción de hardware como VHDL o Verilog.

En la tabla 2.3 se muestran un par de funciones de conmutación. Podríamos considerar que $S$ es la suma y $C$ el acarreo que se produce luego de sumar los bits $A$ y $B$. No estamos considerando, por ahora, un posible acarreo de entrada, por lo que el circuito que calcula las funciones se denomina *semisumador* de un bit.

Es bastante evidente que $S = A \oplus B$ y $C = A\,B$.

**Ejemplo 2.3.2.** La tabla 2.4 muestra un par de funciones de conmutación, $S$ y $C_{i_1}$ cuyo valor depende de tres variables: $C_i$, $A$ y $B$. Si pensamos la pareja $C_{i+1}$, $S$ como el acarreo de salida y la suma de los bits de entrada, respectivamente, lo que tenemos es la especificación de un sumador completo. Para ser más específicos, la tabla 2.4 muestra las funciones suma ($S$) y acarreo de salida ($C_{i+1}$) de los operandos $A$ y $B$ con acarreo de entrada $C_i$.

| $\mathbf{C_i}$ | **A** | **B** | $\mathbf{C_{i+1}}$ | **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 |

Tabla 2.4: Tabla de verdad de un par de funciones de conmutación.

Podríamos hacer los circuitos que corresponden de acuerdo con los mintérminos, $S$ no puede simplificarse. $C_{i+1}$ queda con tres en vez de cuatro términos al simplificar. Pero podemos ahorrarnos trabajo si pensamos en que el resultado debería ser la suma de $A$ y $B$, sumada con $C_i$ y el acarreo de salida debería ser el que resulte de sumar $A$ y $B$ o bien el que resulte de sumar esa suma con el acarreo de entrada. Podemos entonces usar dos semisumadores: encadenar las sumas para obtener $S$ y hacer un OR de los acarreos para obtener $C_{i+1}$.

En la figura 2.6 se muestra el circuito correspondiente, se le denomina *sumador completo* o *full adder* en inglés.

Con el resultado del ejemplo previo podríamos pensar en hacer un sumador de un número arbitrario de bits. En la figura 2.7 se muestra un sumador de 8 bits.

Así es como se procede normalmente, construyendo módulos que realizan alguna operación básica que luego puede usarse para llevar a cabo una más completa.

Este sumador de 8 bits funcionaría, ciertamente, pero no resultaría muy práctico. ¿Cuál sería el retardo necesario para que acarreo se propague hasta la salida? habría que pasar por dos niveles de compuerta, al menos, por cada

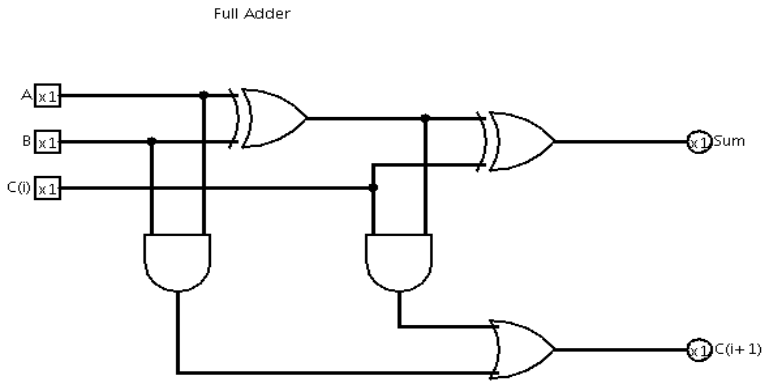


Figura 2.6: Sumador completo de un bit.

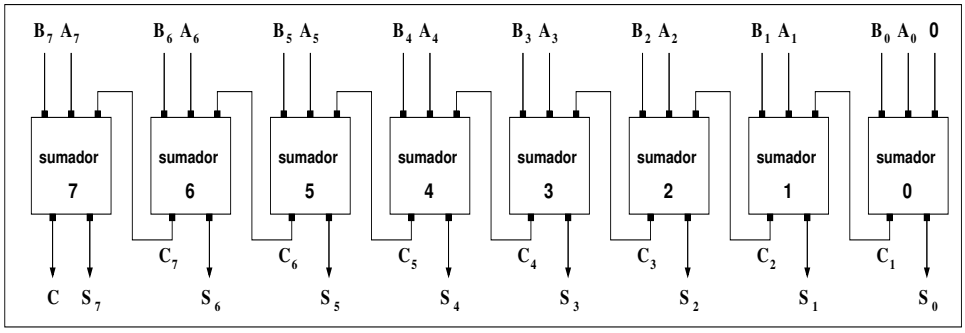


Figura 2.7: Sumador de 8 bits construido con base en sumadores de un bit.

sumador, así que se deben pasar por un total de $2n$ compuertas, inaceptable.

Pero podemos hacer otra cosa. Definamos:

$$P_i = A_i \oplus B_i$$

y

$$G_i = A_i \, B_i$$

La suma y el acarreo se expresarían entonces:

$$S_i = P_i \oplus C_i$$

y

$$C_{i+1} = G_i + P_i \, C_i$$

Es decir: el acarreo de salida vale uno cuando los dos sumandos valen uno o bien cuando la suma de ellos vale uno y hay un acarreo de entrada. Podemos usar esta última expresión para escribir el valor de los acarreos:

$$\begin{aligned}
C_1 &= G_0 + P_0 \, C_0 \\
C_2 &= G_1 + P_1 \, C_1 = G_1 + P_1 \, G_0 + P_0 \, P_1 \, C_0 \\
C_3 &= G_2 + P_2 \, C_2 = G_2 + P_2 \, G_1 + P_1 \, P_2 \, G_0 + P_0 \, P_1 \, P_2 \, C_0 \\
C_4 &= G_3 + G_2 \, P_3 + G_1 \, P_2 \, P_3 + G_0 \, P_1 \, P_2 \, P_3 + C_0 \, P_0 P_1 P_2 P_3 \\
\vdots &= \vdots \\
C_{n+1} &= G_n + \sum_{k=1}^{n} \left[ G_{k-1} \prod_{i=k}^{n} P_i \right] + C_0 \prod_{i=0}^{n} P_i
\end{aligned}$$

En la figura 2.8 se muestra el resultado.

## 2.4. Decodificadores y multiplexores

Otros circuitos combinacionales de singular importancia y que son, por cierto, similares, son los decodificadores y los multiplexores.

Un decodificador recibe una entrada codificada, un número binario de $k$ bits y entrega un conjunto de $2^k$ lineas de salida, pero sólo una de ellas con valor 1 y el resto en 0. El código de entrada es el índice de la línea que debe salir en 1. En la figura 2.9 se muestra un decodificador 3 a 8.

Un multiplexor es un circuito que, cómo el decodificador, recibe un código de $k$ bits de entrada que determina, la salida. Pero además recibe también como entrada $2^k$ líneas diferentes, cada una de ellas con valor 0 o 1. El multiplexor tiene sólo una línea de salida y por ella es transportado el valor

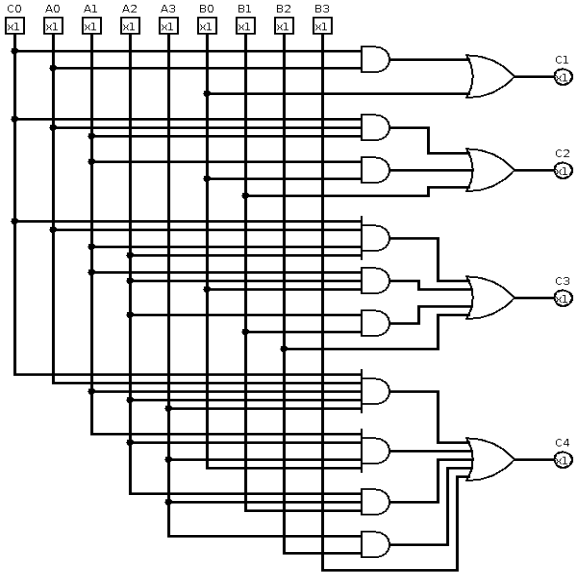


Figura 2.8: Acarreo anticipado.

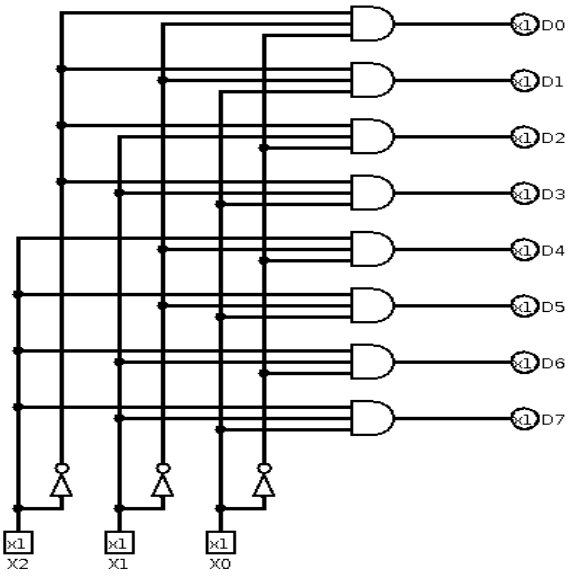


Figura 2.9: Decodificador 3 a 8.

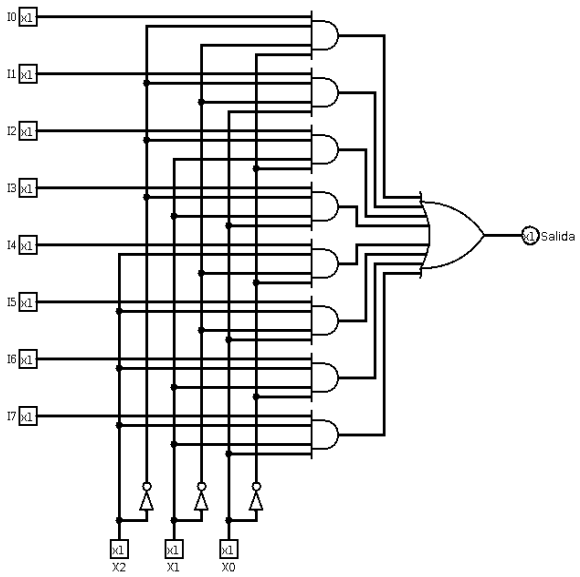


Figura 2.10: Multiplexor 8 a 1. A es el bit más significativo y C el menos significativo.

de una y sólo una de las líneas de entrada, a saber, aquella cuyo índice es el código dado al multiplexor. En la figura 2.10 se muestra un multiplexor 8 a 1.

## 2.5. Lógica secuencial

Un circuito secuencial es aquel cuya salida depende tanto de sus entradas como del estado en el que se encontraba el circuito al recibir dicha entrada. Esto significa que el circuito debe "saber" en que estado está, debe recordarlo; es una entidad con memoria.

Podemos almacenar un bit usando compuertas con un circuito llamado *biestable* (figura 2.11). Una vez que este circuito adquiere una configuración, en teoría, la mantiene *ad infinitum*. Lo malo es que no admite entrada alguna. Necesitamos un circuito que admita una entrada y la mantenga encerrada, esto es lo que se conoce como *cerrojo* o *latch*. De un latch necesitamos que se pueda poner en 1 (*set*) y este valor sea conservado hasta que otra cosa pase, por ejemplo restablecer el latch (dar *reset*). Con la letra $Q$ identificaremos el estado del latch.

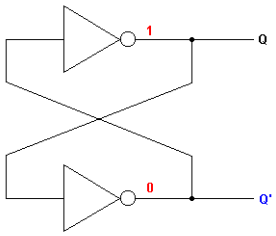


Figura 2.11: Circuito biestable.

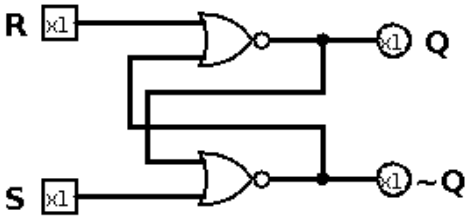


Figura 2.12: Latch SR implementado con compuertas NOR.

### 2.5.1. Latch SR

El latch SR hace justamente lo que hemos descrito. Tiene dos líneas de entrada que, en principio, no pueden estar simultáneamente en 1: *set* y *reset*, para establecer (poner en 1) o borrar (poner en cero) el estado del latch respectivamente. Si ambas líneas son 0 significa que el circuito debe permanecer en el estado que posea actualmente y que, por supuesto, es 0 o 1. El circuito posee dos salidas, una con el estado $Q$ y otra con el complemento de este $\overline{Q}$. En la tabla 2.5 se muestra la función del latch y en la figura 2.12 el circuito implementado con compuertas NOR, se puede también implementar con NAND. Hay que señalar que poner ambas entradas en 1 lleva el latch a un estado inconsistente en el que ambas salidas son 0, lo que no debería ocurrir.

Podemos considerar también la tabla de verdad que determina el estado del latch en función de su estado previo y sus entradas (tabla 2.6).

Es posible simplificar la función (véase el mapa de Karnaugh en la figura 2.13), con lo que se obtiene:

$$\begin{aligned} Q_{t+1} &= S + \overline{R}\, Q_t \\ S \cdot R &= 0 \end{aligned} \tag{2.1}$$

| **S** | **R** | **Q** | $\overline{\mathbf{Q}}$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

Tabla 2.5: Tabla de función del latch SR. Hay dos renglones con $S = R = 0$, el primero suponemos que ocurre luego de hacer set, el segundo luego de reset, es decir, cuando ambas entradas están en cero el circuito mantiene el estado que hayamos establecido previamente.

| $\mathbf{Q_t}$ | **S** | **R** | $\mathbf{Q_{t+1}}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | X |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | X |

Tabla 2.6: Tabla de transición de estados del latch SR.

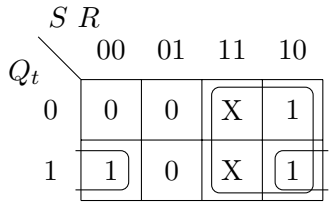


Figura 2.13: Mapa de Karnaugh para las transiciones del latch SR.

| **S** | **R** | $Q_{t+1}$ |
|---|---|---|
| 0 | 0 | $Q_t$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | ¿? |

Tabla 2.7: Tabla característica del latch SR.

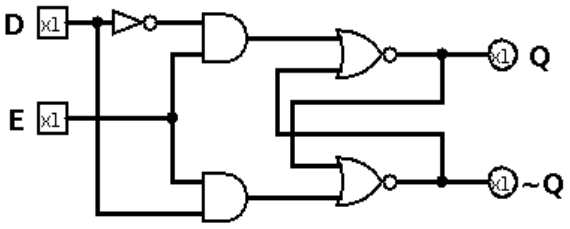


Figura 2.14: Latch D. La línea D es la entrada, la línea E es "enable" y tiene la función de controlar la entrada al latch.

| $Q_t$ | **D** | $Q_{t+1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Tabla 2.8: Tabla de transición del latch D.

A la que se le suele llamar *ecuación característica* y expresa lo mismo que la llamada *tabla característica* 2.7.

### 2.5.2. Latch D

Una solución rápida y simple para evitar la indeseable condición indeterminada en el latch SR es ponerle una sola línea de entrada y obligar a que S y R sean siempre complementarios. Podríamos de hecho hacer algo más, poner una línea de control que permita la entrada al latch o no. En la figura 2.14 se muestra el resultado.

La tabla 2.8 muestra las transiciones del latch D. Su ecuación es muy simple:

$$Q_{t+1} = D \tag{2.2}$$

La introducción de nuestra línea de control es una buena idea, así el biestable no cambiará cuando cambie su entrada sino cuando decidamos que así debe ser o cuando estemos seguros de que lo que va a entrar no

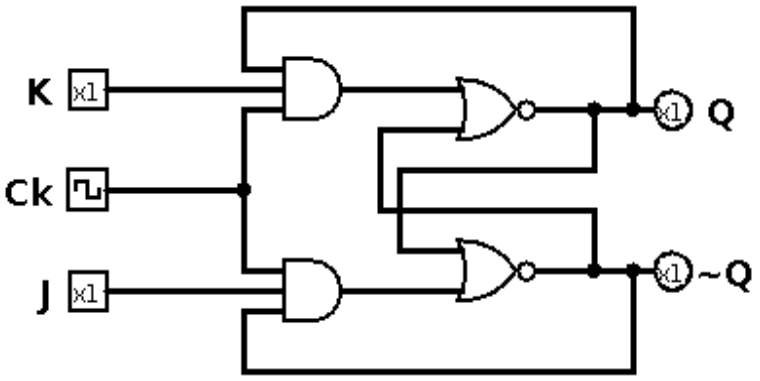


Figura 2.15: Flip-flop JK.

es una se;al espuria. Normalmente estas líneas de control se conectan a un oscilador, un reloj que, con cierta frecuencia, habilita y deshabilita la entrada del latch. En estas condiciones el latch es síncrono y suele llamarse entonces *flip-flop*.

### 2.5.3. Flip-flop JK

Otra posibilidad de corregir el latch SR es colocándole una líneas de retroalimentación para que en caso de entrar 1 por ambas entradas el resultado sea el estado complementario del actual. El resultado es el flip-flop (por añadirse el reloj) JK mostrado en la figura 2.15.

La tabla de transición es la 2.9, la simplificación mostrada en el mapa de la figura 2.16 da lugar a:

$$Q_{t+1} = J\ \overline{Q_t} + \overline{K}\ Q_t \tag{2.3}$$

Esto se puede simplificar como se observa en el mapa de Karnaugh de la figura 2.16

Finalmente la tabla característica del flip-flop JK es la 2.10.

## 2.6. Análisis y diseño de circuitos secuenciales

Lo visto en la sección anterior se puede resumir en la tabla 2.11.

Usando flip-flops es posible construir, en hardware, máquinas de estados finitos. Esos modelos de cómputo restringidos de acuerdo con el tamaño y las características de acceso a su memoria. Los fip-flops son el medio que

| $\mathbf{Q_t}$ | **J** | **K** | $\mathbf{Q_{t+1}}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Tabla 2.9: Tabla de transición de estados del flip-flop JK.

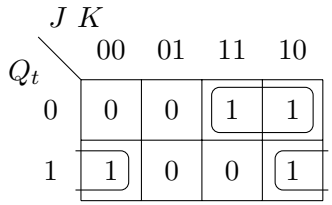


Figura 2.16: Mapa de Karnaugh para las transiciones del flip-flop JK.

| **J** | **K** | $Q_{t+1}$ |
|---|---|---|
| 0 | 0 | $Q_t$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q_t}$ |

Tabla 2.10: Tabla característica del flip-flop JK.

| $Q_t$ | $Q_{t+1}$ | S | R | J | K | D |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | X | 0 | X | 0 |
| 0 | 1 | 1 | 0 | 1 | X | 1 |
| 1 | 0 | 0 | 1 | X | 1 | 0 |
| 1 | 1 | X | 0 | X | 0 | 1 |

Tabla 2.11: Síntesis de las características de los flip-flops.

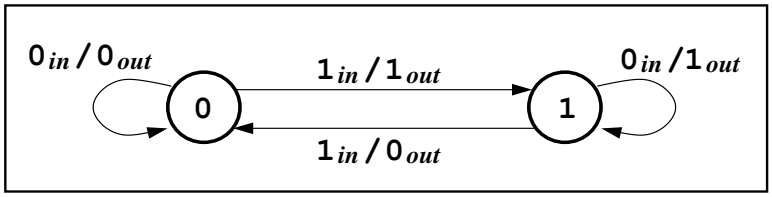


Figura 2.17: Un ejemplo de máquina de estados finitos para paridad par.

| Edo. actual | Edo. Sig. | | Salida | |
|---|---|---|---|---|
| | In=0 | In=1 | In=0 | In=1 |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

Tabla 2.12: Tabla de transiciones para la máquina de estados finitos de paridad par.

necesitamos para recordar el estado en que nos encontramos. Usaremos las funciones características para llevar a cabo las transiciones del autómata.

**Ejemplo 2.6.1.** En la figura 2.17 se muestra un autómata para determinar la paridad de una secuencia de bits recibidos como entrada. La idea es que el autómata complete un número par de unos. Diseñaremos el dispositivo de control con base en flip-flops tipo D. En la tabla 2.12 se especifica el comportamiento del autómata. Llamaremos $A$ al estado del autómata.

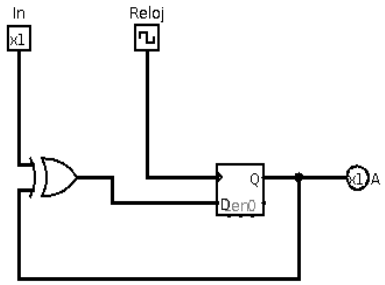


Figura 2.18: Circuito secuencia para el autómata de paridad par.

| $A$ | $In$ | $A_s$ | $Out$ | $D_A$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

Tabla 2.13: Tabla de transición de flip-flop para la máquina de paridad par.

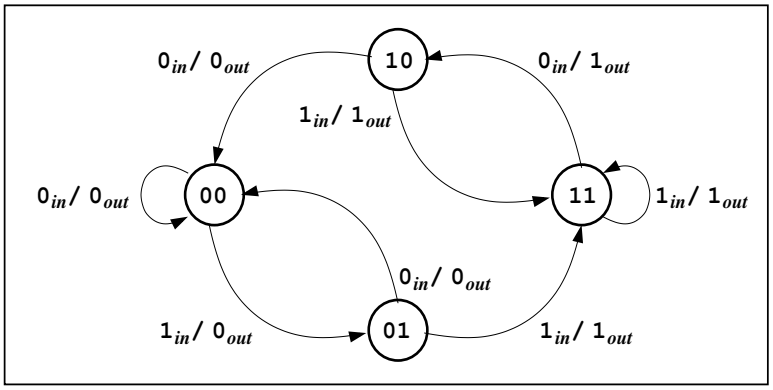


Figura 2.19: Un ejemplo de otra máquina de estados finitos.

**Ejemplo 2.6.2.** En la figura 2.19 se muestra una máquina de estados finitos. Más adelante develaremos su utilidad, por ahora sólo queremos diseñar su dispositivo de control con base en flip-flops. Usaremos los del tipo JK.

Podemos sintetizar el comportamiento de la máquina mediante la tabla 2.14.

Las etiquetas de los estados deben medir 2 bits, dado que hay cuatro estados. Como cada flip-flop puede almacenar un bit, debemos usar entonces dos flip-flops para representar el estado, llamemos $A$, al más significativo, y $B$ al menos significativo. Si denotamos con $A_s$ y $B_s$ al contenido *siguiente* de los flip-flops $A$ y $B$ respectivamente, luego de que el circuito reciba una entrada dada $In$, podemos representar en las cinco primeras columnas de la izquierda de la tabla 2.15 lo que el ocurre al autómata dado su estado actual y la entrada que recibe. En las columnas etiquetadas con $J_A$ y $K_A$ se

| Edo. actual | Edo. Sig. | | Salida | |
|---|---|---|---|---|
| | In=0 | In=1 | In=0 | In=1 |
| 00 | 00 | 01 | 0 | 0 |
| 01 | 00 | 11 | 0 | 1 |
| 10 | 00 | 11 | 0 | 1 |
| 11 | 10 | 11 | 1 | 1 |

Tabla 2.14: Tabla de transiciones para la máquina de estados finitos de la figura 2.19.

ha puesto lo que debe recibir por sus entradas J y K el flip-flop $A$ y en las etiquetadas $J_B$ y $K_B$ lo correspondiente al flip-flop $B$. La última columna ($Out$) denota lo que se supone debe producir de salida el autómata.

Tenemos ahora completamente especificadas las cinco funciones que necesitamos, a saber: $J_A$, $K_A$, $J_B$, $K_B$ y $Out$. Luego de simplificarlas como se muestra en los mapas de Karnaugh:

$J_A = B \cdot In.$

$K_A = \overline{B} \cdot \overline{In}.$

$J_B = In.$

$K_B = \overline{In}.$

$Out = B \cdot In + A \cdot In + A \cdot B.$

El resultado final se muestra en la figura 2.25.

| $A$ | $B$ | $In$ | $A_s$ | $B_s$ | $J_A$ | $K_A$ | $J_B$ | $K_B$ | $Out$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | X | 0 | X | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | X | 1 | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | X | X | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | X | X | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | X | 1 | 0 | X | 0 |
| 1 | 0 | 1 | 1 | 1 | X | 0 | 1 | X | 1 |
| 1 | 1 | 0 | 1 | 0 | X | 0 | X | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | X | 0 | X | 0 | 1 |

Tabla 2.15: Tabla de transiciones en el estado de los flip-flops.

| $A$ \ $B\ In$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | X | X | X | X |

Figura 2.20: $J_A$.

| $A$ \ $B\ In$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | X | X | X | X |
| 1 | 1 | 0 | 0 | 0 |

Figura 2.21: $K_A$.

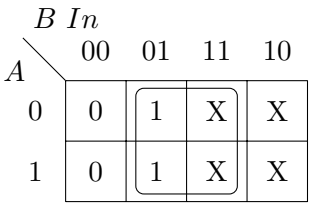


Figura 2.22: $J_B$.

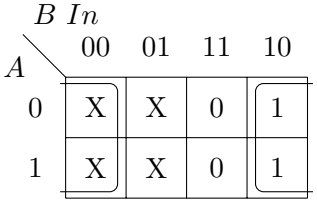


Figura 2.23: $K_B$.

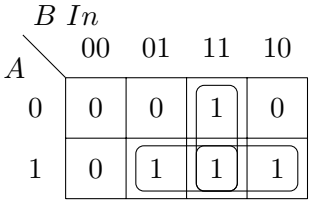


Figura 2.24: $Out$.

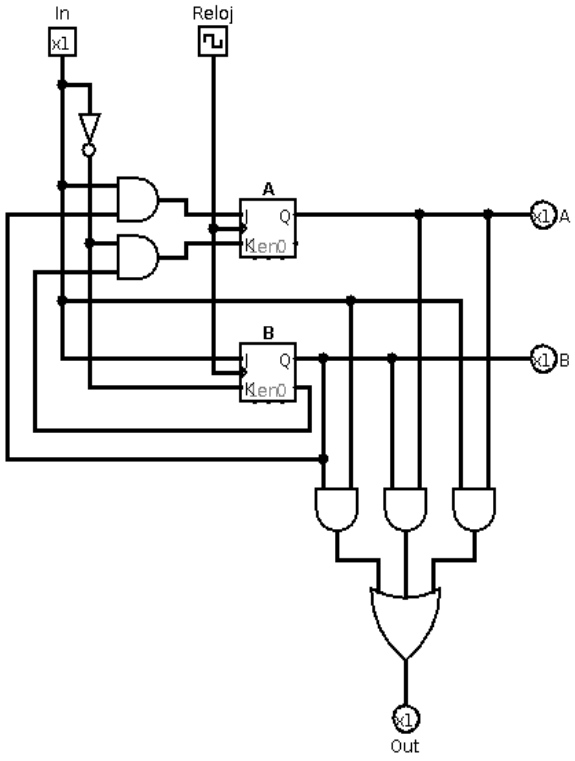


Figura 2.25: Circuito secuencia para el autómata de la figura 2.19.

# Bibliografía

[1] Hennessy, J. y D. Patterson, *Computer Architecture: A Quantitative Approach*, 6a Ed. Morgan Kaufmann, 2017.

[2] Mano, Morris, *Logic And Computer Design Fundamentals*, 4a Ed., Pearson, 2013.