Breves Notas sobre
Diseño de Computadoras y Sistemas Digitales

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Octubre 2003

Índice general

Prefacio

Las *Breves Notas sobre Diseño de Computadoras y Sistemas Digitales* introducen en forma simple y sencilla a algunos de los temas relevantes en las áreas de Diseño de Sistemas Digitales y Arquitectura de Computadoras. No tiene la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que por su relevancia se consideran esenciales para el conocimiento básico de estas áreas, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Sistemas de Lógica, Lógica Booleana, Mapas de Karnaugh, Codificadores y Multiplexores, Circuitos Secuenciales, la SCRAM, y Computadoras VLSI. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe conocer en las asigna-turas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Octubre 2003

Capítulo 1

Sistemas de Lógica
Bases Booleanas

En esta era de computadoras y automatización, casi cualquier dispositivo electrónico que uno pudiera nombrar incorpora al menos una función booleana. Por ejemplo, muchos de los modelos actuales de automóviles emiten una señal, sonido, alarma, u otro ruido molesto hasta que el conductor se coloca el cinturón de seguridad. Tal ruido se produce por un dispositivo que realiza una función booleana de dos variables A ($A = 1$ si la llave de ignición se enciende) y B ($B = 1$ si el cinturón de seguridad ha sido ajustado). Ambas variables toman valores de 0 o 1, usualmente interpretadas como falso (*false*) o verdadero (*true*) respectivamente, y la función misma toma los mismo valores. Supóngase que la función se representa por W, de tal modo que cuando $W = 1$ la señal sonora se enciende, mientras que si $W = 0$, se apaga. La operación del dispositivo puede resumirse en términos de W, donde:

$$W = \begin{cases} 1 & \text{si } A \text{ es 1 y } B \text{ es 0} \\ 0 & \text{de otra forma} \end{cases}$$

De hecho, es posible representar esto de una mejor manera mediante definir dos operadores lógicos **AND** y **NOT**, y usándolos como sigue:

$$W = A\,\mathbf{AND}(\mathbf{NOT}B)$$

donde para dos variables booleanas X y Y definimos $X\mathbf{AND}\,Y$ a ser 1 en el caso de que tanto X como Y valgan 1; y $\mathbf{NOT}X$ es 1 cuando X valga 0, y es 0 cuando X valga 1.

Acabamos entonces de definir una función booleana llamada W en términos de otras dos: **AND** y **NOT**. Generalmente éstas dos últimas se escriben con un símbolo de producto $(\cdot)$ y un símbolo de complemento $(')$, respectivamente. Por lo tanto, podemos escribir la función W en una forma aún más compacta, como $W = A \cdot B'$.

Cualquier función booleana puede definirse mediante una *tabla de verdad*, la cual es meramente una forma conveniente de listar los valores de la función para cada combinación posible de valores de sus variables. Por ejemplo, W puede definirse por la siguiente tabla.

A B	W
0 0	0
0 1	0
1 0	1
1 1	0

Podría parecer que, si se da una función booleana arbitraria de dos variables, sería por mera casualidad el poder encontrar una expresión usando **AND** y **NOT** para describirla. Es interesante, pero este no es el caso: *cualquier* función booleana de dos variables puede ser escrita de esta manera. La forma más sencilla de entender esto es considerar el siguiente ejemplo, que involucra una función F cuya tabla de verdad es de la forma:

X Y	F
0 0	0
0 1	1
1 0	0
1 1	1

La idea esencial es escribir F como un producto de expresiones, una por cada renglón de la tabla, las cuales individualmente producen un valor de 0 para F. Por lo tanto, cuando cualquiera de las expresiones individuales es igual a 0, también F es 0, por lo que se fija la atención en los renglones de la tabla donde $F = 0$. Un ejemplo es el primer renglón de la tabla. Debido a que la expresión puede solo utilizar los operadores $\cdot$ y $'$, es necesario preguntarse ¿qué combinación de las variables producirá un valor de 0 cuando X y Y sean ambas 0?

La respuesta es $(X' \cdot Y')'$. Para la combinación $X = 0$ y $Y = 0$, esta expresión tiene el valor de 0:

$$\begin{aligned}(0' \cdot 0') &= (1 \cdot 1)' \\ &= 1' \\ &= 0\end{aligned}$$

En forma similar, el tercer renglón de la tabla para F también corresponde a un valor de 0 para F. Su expresión es $(X \cdot Y')'$. Cuando (y sólo cuando) $X = 1$ y $Y = 0$, esta expresión es igual a 0.

La expresión de la función F puede escribirse entonces como:

$$F = (X' \cdot Y')' \cdot (X \cdot Y')'$$

Ya que cualquier función booleana de dos variables puede ser expresada en términos de $(\cdot)$ y $(')$, se dice que estas dos operaciones forman una *base completa* para las funciones de dos variables. Más aún, $\{\cdot,'\}$ forma una *base completa mínima* porque ninguna de estas operaciones por sí sola sería suficiente para expresar todas las funciones booleanas de dos variables. Esto es cierto también para funciones con tres, cuatro,.., n variables. El conjunto $\{\cdot,'\}$ es una base completa para una función booleana de cualquier número de variables. Si una función F tiene n variables, sólo es necesario observar los renglones de su tabla de verdad que tengan el valor de 0, y escribir el "producto" de las variables (apropiadamente complementadas) que corresponde a cada renglón. Si el renglón es 010110, se escribe ${x_1}' \cdot x_2 \cdot {x_3}' \cdot x_4 \cdot x_5 \cdot {x_6}'$. En seguida, se encierra cada una de las expresiones por cada renglón entre paréntesis, se aplica el operador complemento a cada una de ellas, y se concatena todas las expresiones resultantes juntas en un gran producto. Este producto tiene valor de 0 si y solo si F tiene valor de 0, por las mismas razones que se han indicado anteriormente.

Surge entonces la pregunta ¿cuáles otros conjuntos de operadores forman bases completas? Suponiendo que por "operadores" se refiera a funciones booleanas de dos variables, esta pregunta se puede responder sólo tras examinar todas las posibles funciones de dos variables que se muestran en la siguiente tabla.

Nombre de la función	Símbolo	En términos de $\cdot$, + y $'$
OR	$+$	$x + y$
AND	$\cdot$	$x \cdot y$
NOT(x)	$'$	x'
NOT(y)	$'$	y'
OR Exclusiva	$\oplus$	$x' \cdot y + x \cdot y'$
Equivalencia	$\equiv$	$x \cdot y + x' \cdot y'$
Implicación	$\rightarrow$	$x' + y$
No implicación	$\nrightarrow$	$x \cdot y'$
Implicación reversa	$\leftarrow$	$x + y'$
No implicación reversa	$\nleftarrow$	$x' \cdot y$
Proyección(x)	x	x
Proyección(y)	y	y
NAND	$\mid$	$(x \cdot y)'$
NOR		$(x + y)'$
0 Contstante	0	0
1 Contstante	1	1

No es de sorprenderse que haya exactamente 16 funciones booleanas de dos variables (x,y); éstas corresponden a todas las posibles formas de llenar la tabla de verdad de una función F desconocida:

x y	F
0 0	?
0 1	?
1 0	?
1 1	?

Ciertas combinaciones de estas 16 funciones booleanas forman bases completas en el sentido siguiente: cuando las funciones se consideran operaciones binarias, cualquier función booleana de cualquier número de variables puede ser expresada en términos de esas funciones. Por ejemplo, el conjunto $\{+,'\}$ y el conjunto $\{+,\equiv,\oplus\}$ son ambos bases completas. De hecho, son bases completas mínimas. Ningún subconjunto propio de ellos es una base completa.

La figura 1.1 resume todas las bases completas. Cada punto representa una de las funciones booleanas de dos variables listadas anteriormente, y las bases completas mínimas se representan por puntos, líneas, y triángulos sólidos. Un punto no contenido en una línea y una línea no contenida en un

triángulo sólido representan bases completas con uno y dos elementos respectivamente. Los triángulos sólidos representan bases completas mínimas de tres elementos.

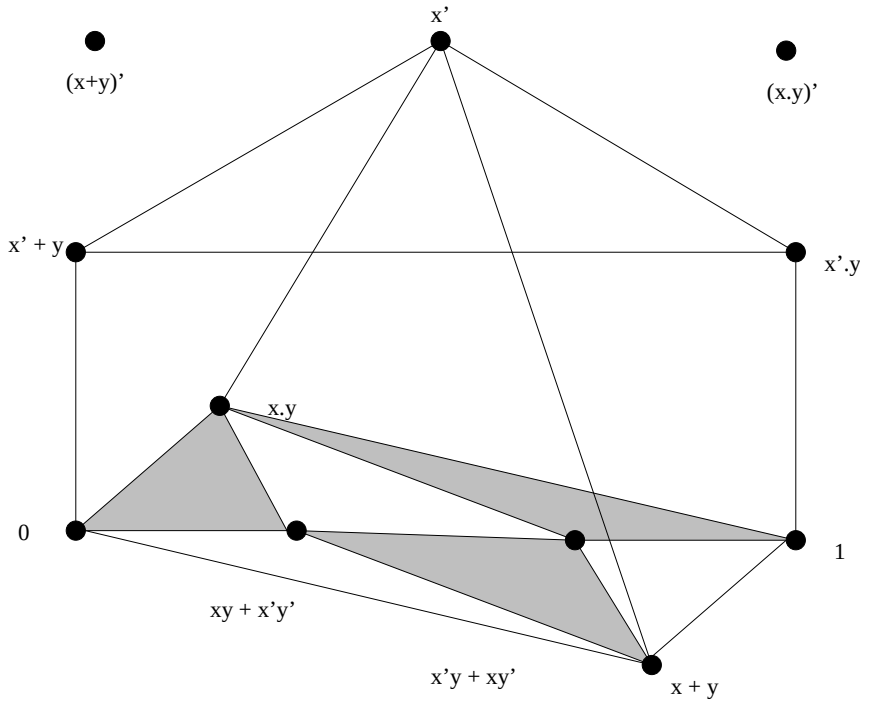


Figura 1.1: Estructura de bases completas

Ya que solo 11 operaciones son representadas en la figura 1.1, faltan claramente 5 de las 16 operaciones presentadas anteriormente. Cuando se les considera como *operaciones*, sin embargo, tres de las funciones de dos variables de la lista resultan redundantes. Por ejemplo, cuando es considerada como operación, x' es la misma que y'. Más aún, $x' + y$ es la misma que $x + y'$, y $x'y$ es la misma que xy'. Estas operaciones se realizan en pares de funciones booleanas, y $x' + y$ significa tan solo "complementa una de las funciones y súmese a la otra"; la primera función puede ser x, y la otra puede ser y, o viceversa.

En el contexto de la lógica de computadoras, las bases completas adquieren un significado que va más allá del interés puramente matemático del esquema anterior. Para que la unidad de control, la unidad lógico-aritmética, y otros componentes lógicos de una computadora funcionen apropiadamente, la lógica debe involucrar un conjunto de operaciones que formen una base completa.

En la mayoría de los textos sobre diseño lógico de computadoras, las varias operaciones lógicas se representan por *compuertas*. Estas son simplemente operaciones booleanas de alguno de los tipos especificados anteriormente, aplicadas a conjuntos específicos de líneas eléctricas en un circuito. Por ejemplo, un diagrama de compuertas para el dispositivo del cinturón de seguridad que se menciona al inicio de este capítulo se vería como se muestra en la figura 1.2.

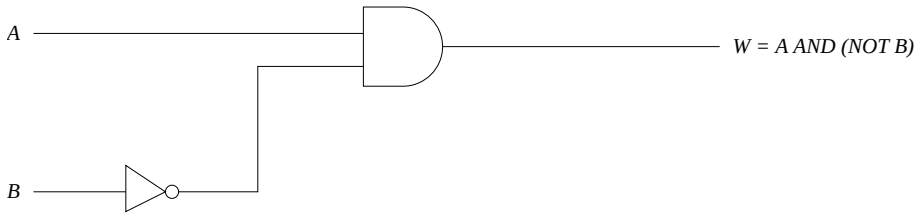


Figura 1.2: Lógica de la señal para colocarse el cinturón de seguridad

La compuerta semicircular representa una operación **AND** y la compuerta triangular (a veces llamado "inversor") representa una operación **NOT**. La función **OR** se representa como una figura en forma de escudo como se muestra en la figura 1.3, la cual muestra un circuito algo más sofisticado llamado *multiplexor*.

Los multiplexores se encuentran en todas las computadoras. Permiten que varias funciones diferentes (las líneas de entrada etiquetadas como 0, 1, 2 y 3) usen una sola línea de salida (etiquetada como *output*). Esto se hace algunas veces por economía y otras veces porque las funciones que se multiplexan son usadas por un solo dispositivo al otro lado de la línea de salida. El multiplexor que se muestra aquí opera de la siguiente manera: dependiendo del valor que se asigne a las señales de selección, S_1 y S_0, se selecciona la entrada a ser transmitida por la salida a partir de la siguiente tabla.

S_0 S_1	Línea seleccionada
0 0	0
0 1	1
1 0	2
1 1	3

Sólo una línea de las cuatro compuertas **AND** transmite la señal que llega por su entrada de la izquierda. Por ejemplo, si $S_0 = 0$ y $S_1 = 1$, entonces las compuertas **AND** 0, 2 y 3 reciben al menos un 0 de las líneas de selección, y por lo tanto, no pueden contribuir mas que con un 0 a la compuerta **OR** operando la línea de salida. Sin embargo, la compuerta **AND** 1 recibe dos entradas 1 de las líneas de selección, y su salida depende del valor de la entrada 1, ya sea ésta 0 o 1. Nótese que los bits (0 y 1) que aparecen en las líneas de selección simplemente dan la expansión del número de línea seleccionado para la salida.

Dada esta breve introducción a los diagramas lógicos que involucran compuertas **AND**, **OR** y **NOT**, es posible ahora examinar algunas de las bases completas mínimas vistas anteriormente en términos de la lógica de computadoras.

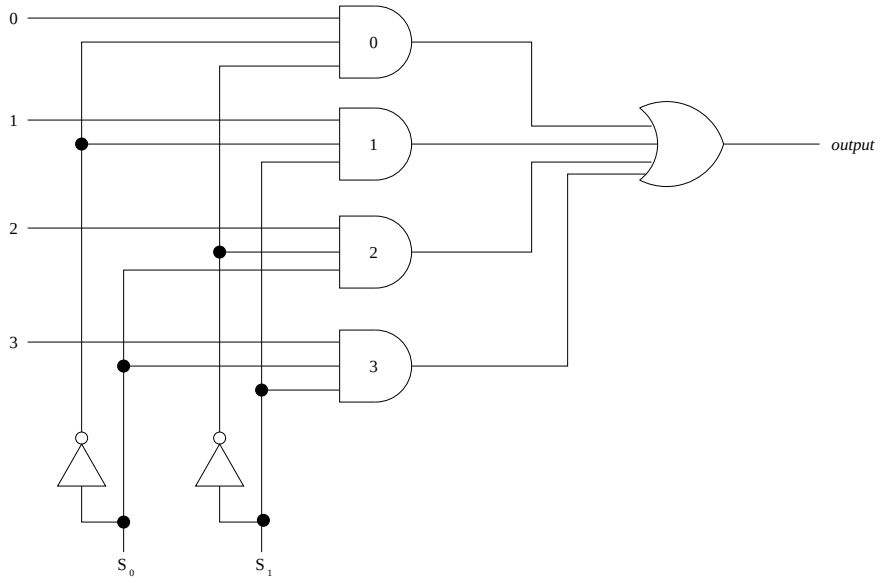


Figura 1.3: Un multiplexor

Una de las porciones más interesantes de la estructura de bases completas de la figura 1.1 es el punto aislado etiquetado como $(x \cdot y)'$. Esta operación, llamada **NAND**, forma una base completa por sí misma. La forma más sencilla de observar esto es escribir las operaciones de una base completa, por ejemplo $\{\cdot,'\}$, en términos de la operación **NAND**. La figura 1.4 muestra un ejemplo en términos de compuertas.

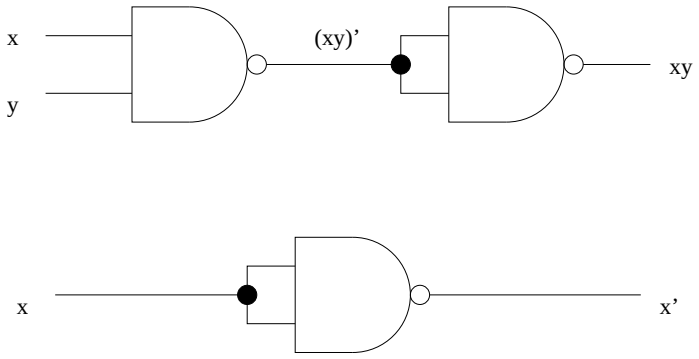


Figura 1.4: Convirtiendo de base $\{\cdot,'\}$ a $\{|\}$

Se utilizan compuertas **NAND** (un semicírculo con un pequeño circulo a su salida) para realizar las operaciones **AND** y **NOT**. Una compuerta **NAND** de dos entradas, cuyas entradas se encuentran conectada entre sí, opera exactamente igual a un inversor, e invirtiendo la salida de una compuerta **NAND** reproduce la operación **AND**.

Por fortuna, las compuertas **NAND** son muy sencillas de construir a partir de transistores, ya sea una tecnología de compuertas lógicas o por la más sofisticada tecnología de circuitos de silicio. La mayoría de los diagramas que se utilizan para representar qué sucede realmente dentro de las computadoras de hoy contienen un gran número de compuertas **NAND**.

Capítulo 2

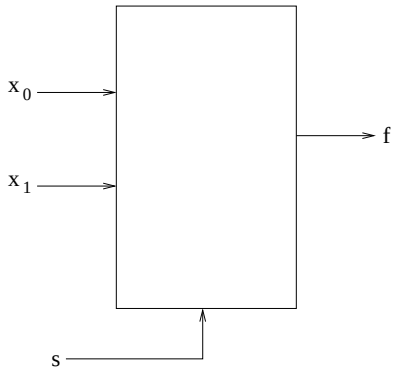
Lógica Booleana
Expresiones y Circuitos

En el núcleo mismo de cualquier computadora existe una unidad de control, la cual dirige la transferencia y manipulación de datos entre y dentro de miles de localidades, donde la información es almacenada. Esencialmente, esta unidad es un circuito electrónico con docenas de líneas de entrada y de salida, que pasa la información de entrada a través de una red de compuertas lógicas. La configuración particular de estas compuertas determina precisamente la función que la unidad de control realiza. Para ponerlo de otro modo, primero los diseñadores de computadoras deciden qué función o funciones desean que la unidad realice, y después diseñan el circuito para hacerlo. Claramente, los circuitos lógicos son utilizados en un gran número de otros dispositivos, además de las computadoras.

La progresión desde la especificación funcional de un circuito lógico hasta la realización del circuito mismo generalmente involucra un paso intermedio en el que la función es expresada como una fórmula. El desarrollo de este proceso de tres pasos ha comprendido un periodo de tiempo de cien años, a partir de los trabajos de George Boole a mediados del siglo XIX, y hasta finales de la primera mitad del siglo XX. Fue Boole quien desarrolló mucho de la labor intelectual de tales fórmulas y su manipulación.

Una *variable booleana* puede tener uno de solo dos valores posibles, llamados 0 ó 1. Una *función booleana* tiene un número de variables de entrada, todas booleanas, y por cada combinación posible de valores de entrada, tal función tiene una salida con valor booleano, ya sea 0 ó 1. La figura 2.1 muestra un dispositivo y una tabla que especifica la función que representa. Por

ahora, el interior del dispositivo se muestra vacío. Más adelante, se muestra cómo llenarlo.



s	x_0	x_1	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

Figura 2.1: La función multiplexor

La figura 2.1 especifica la forma más simple posible de un "multiplexor", cuyo propósito es combinar la información de las dos líneas x_0 y x_1, hacia un sola línea f. Específicamente, permite que la información de una de las líneas pase a través de sus circuitos y hacia la salida f.¿Cuál línea tendrá el "privilegio"? En un momento dado esto se decide por una tercera línea de entrada, etiquetada s por "selección".

Cuando la entrada s tiene el valor de 0, el valor que se encuentra en la entrada x_0 se transmite a f; y cuando s es igual a 1, se transmite el valor de x_1. Esta descripción funcional se resume completamente en la tabla de verdad, en la cual por cada una de las ocho combinaciones posibles de los valores booleanos de las entradas, se lista un valor correspondiente de la salida (etiquetada f). En este caso en particular, es notorio que en todo renglón en que $s = 0$, f tiene el mismo valor que x_0, y donde $s = 1$, $f = x_1$.

La función f que se ha descrito hasta aquí tiene tres variables, y su tabla de verdad (llamada así por que 0 y 1 son considerados respectivamente como falso y verdadero) tiene ocho renglones. El patrón particular de ceros y unos en la columna de f sirve para especificar la función f completamente. Un patrón diferente significa que se trata de una función diferente. Hay en total $2^8 = 256$ patrones, es decir, existen un total de 256 funciones booleanas de tres variables que pueden ser especificadas. Ahora bien, si se tratara de una tabla de verdad para una función booleana de cuatro variables, entonces

tendría 16 renglones, y por lo tanto, hay $2^{16} = 65,536$ posibles funciones de cuatro variables booleanas. Claramente, el número de funciones booleanas crece rápidamente al incrementar su número de variables. De hecho, el número de las funciones booleanas de n variables es 2^{2^n}.

Así como sucede con otros tipos de variables, las variables booleanas pueden ser "sumadas" y "multiplicadas" en su propia manera, y hasta "negadas". Mediante el uso de tales operadores (denotados por $+$, $\cdot$ y $'$, respectivamente) es posible construir expresiones de gran complejidad. Dadas dos variables booleanas x y y, las tres operaciones se resumen utilizando tablas de verdad, tal y como una operación aritmética puede resumirse utilizando tablas aritméticas.

x y	$+$
0 0	0
0 1	1
1 0	1
1 1	1

x y	$\cdot$
0 0	0
0 1	0
1 0	0
1 1	1

x	$'$
0	1
1	0

Otra manera de considerar estas operaciones involucra la idea más tradicional de ver al 0 como falso y al 1 como verdadero. En este escenario, se piensa en $+$ como un **OR**, en $\cdot$ como un **AND**, y en $'$ como un **NOT**. De este modo, $x + y$ significa en efecto "x **OR** y", y tal enunciado es verdadero si al menos x o y es verdadero. Por lo tanto, $x + y = 1$ si al menos x **OR** y es igual a 1.

Es notorio que $+$ y $\cdot$ son sólo dos de las posibles $2^{2^2} = 16$ funciones booleanas de dos variables. ¿Podrían otras de tales funciones servir igualmente como operaciones booleanas? Ciertamente que sí; de hecho, algunas corresponden más cercanamente a las compuertas reales que se encuentran en los circuitos lógicos de una computadora, como se menciona en el capítulo anterior.

En lo que concierne a la construcción de *expresiones booleanas* complejas, el proceso es realmente muy sencillo. Supóngase que se tiene un conjunto de símbolos $\sum$ el cual se utiliza para denotar variables booleanas.

- Si $x \in \sum$, entonces x es una expresión booleana.
- Si A y B son expresiones booleanas, entonces $(A) + (B)$, $(A) \cdot (B)$ y $(A)'$ también son expresiones booleanas.

- Ninguna otra cosa es una expresión booleana.

Estas tres reglas nos permiten comenzar con variables como x, y y z, y combinarlas en una expresión como:

$$(((x + (y'))') \cdot (z \cdot (y \cdot (x'))))$$

Nótese que debido al uso de tantos paréntesis, la lectura de expresiones como ésta resulta complicada. Sin embargo, en la práctica, la mayoría de los paréntesis pueden eliminarse, ya sea por la precedencia de operaciones o por simplificaciones algebraicas. Por ejemplo, si se da precedencia a $\cdot$ sobre $+$, de tal modo que una expresión como $x + y \cdot z$ significa $x + (y \cdot z)$, y no $(x+y)\cdot z$; y además si se da precedencia a $'$ sobre $+$ y $\cdot$, de tal modo que $x+y'$ significa $x + (y')$ y no $(x + y)'$. Así, aplicando estas reglas de precedencia, la expresión anterior se simplifica a:

$$(x + y')' \cdot (z \cdot (y \cdot x'))$$

El hecho de que $z \cdot (y \cdot (x'))$ pueda escribirse simplemente como $z \cdot y \cdot x'$ sigue de la asociatividad de la operación $\cdot$, que a su vez es parte de los axiomas del Algebra Booleana, formuladas por primera vez por Boole en 1854.

Esta álgebra en su forma moderna se define como sigue: Sea B un conjunto, sean $+$ y $\cdot$ operaciones binarias, es decir, de dos argumentos que son elementos del conjunto B, y sea $'$ un operador unario o de un argumento que también es un elemento del conjunto B. Recordando que $\forall$ significa "para todo" y que $\exists$ significa "existe", se enuncian los siguientes axiomas del Algebra de Boole:

1. $x + y \in B$ **y** $x \cdot y \in B$**,** $\forall x, y \in B$
2. $\exists 0, 1 \in B$ **tal que** $x + 0 = x$ **y** $x \cdot 1 = x$**,** $\forall x \in B$
3. $x + y = y + x$ **y** $x \cdot y = y \cdot x$**,** $\forall x, y \in B$
4. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ **y** $x + (y \cdot z) = (x + y) \cdot (x + z)$**,** $\forall x, y, z \in B$
5. $\forall x \in B\, \exists x' \in B$ **tal que** $x + x' = 1$ **y** $x \cdot x' = 0$

Estos axiomas resumen todo lo que Boole quiso expresar por conectivas lógicas como **AND**, **OR** y **NOT**. En tal contexto, no es difícil notar que todos los axiomas son ciertos bajo la interpretación de la lógica ordinaria.

Sin embargo, estos axiomas tienen la intención de ser un punto de inicio para deducciones. Como tales, comprenden la base de un sistema, llamado Algebra Booleana.

Por ejemplo, a partir de los axiomas anteriores, es posible deducir la ley asociativa para la multiplicación:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Esto permite finalmente la simplificación de la expresión:

$$(x + y')' \cdot (z \cdot (y \cdot x'))$$

a la expresión:

$$(x + y')' \cdot (z \cdot y \cdot x')$$

y a la expresión:

$$(x + y')' \cdot z \cdot y \cdot x'$$

En general, es posible utilizar herramientas algebraicas deducidas a partir de los axiomas anteriores para manipular expresiones booleanas, especialmente con el objetivo de simplificarlas. Más adelante se muestra un ejemplo de esa simplificación.

Mientras, es importante hacer notar algo sumamente significativo respecto a las expresiones booleanas como se han definido: cada expresión booleana define una función booleana debido a que tal expresión puede ser evaluada para cada combinación de los valores de sus variables. Por ejemplo, cuando $x = 0$, $y = 1$, y $z = 1$ en la expresión previa se tiene que:

$$\begin{aligned}
(x + y')' \cdot (z \cdot (y \cdot x')) &= (0 + 1')' \cdot (1 \cdot (1 \cdot 0')) \\
&= (0 + 0)' \cdot (1 \cdot 1) \\
&= 0' \cdot 1 \\
&= 1 \cdot 1 \\
&= 1
\end{aligned}$$

Aún más significativo es que podemos hacer esta afirmación a la inversa: cada función booleana se define por alguna expresión (de hecho, una cantidad infinita de expresiones). La forma más sencilla de ver esto es examinar la tabla de verdad de una función de la cual se requiere una expresión booleana:

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

Para cada renglón de la tabla en que $f = 1$, examínese los valores correspondientes de x, y y z. Escríbase un producto en que la variable aparece negada si su valor es 0, o aparece sin negarse si su valor es 1. Por ejemplo, el renglón 011 de la tabla da como resultado el producto $x' \cdot y \cdot z$. Ahora bien, todos los productos obtenidos se suman. La suma del ejemplo es:

$$x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y \cdot z' + x \cdot y \cdot z$$

Nótese lo siguiente:

- La única forma en que la función f pueda tener valor de 1 es cuando al menos uno de los productos tiene valor de 1.
- La única manera de que uno de los productos tenga valor de 1 es cuando las variables que lo constituyen tiene los valores en el renglón de la tabla de verdad correspondiente a ese producto.

De esta forma, se sabe que la expresión anterior obtenida realiza la función f. Este tipo de expresiones reciben el nombre de *forma normal disyuntiva*. No es de sorprenderse que toda función también pueda ser escrita como un producto de sumas, en la *forma normal conjuntiva*.

Curiosamente, si substituimos x por x_0, y por x_1, y z por s en la tabla de verdad anterior, se reconoce que esta representa a la función del multiplexor definida al inicio de este capítulo. De este modo, se ha encontrado una expresión booleana que describe precisamente al multiplexor:

$$f = {x_0}' \cdot x_1 \cdot s + x_0 \cdot {x_1}' \cdot s' + x_0 \cdot x_1 \cdot s' + x_0 \cdot x_1 \cdot s$$

El siguiente paso del proceso descrito en este capítulo consiste en pasar de una expresión booleana a un circuito lógico. Sin embargo, el circuito lógico que corresponde a la expresión anterior es bastante complicado, y

en general se reconoce que mientras más corta sea la expresión, el circuito será más sencillo. En la empresa del hardware digital se requiere entonces que todos los circuitos (y por tanto, las expresiones) sean tan sencillas como sea posible.

Una técnica para simplificar expresiones booleanas se basa en utilizar los axiomas y teoremas del Algebra Booleana:

$$\begin{aligned} f &= x_0{}' \cdot x_1 \cdot s + x_0 \cdot x_1{}' \cdot s' + x_0 \cdot x_1 \cdot s' + x_0 \cdot x_1 \cdot s \text{ (por el axioma 1)} \\ &= x_1 \cdot s \cdot (x_0{}' + x_0) + x_0 \cdot s' \cdot (x_1{}' + x_1) \text{ (por los axiomas 3 y 4)} \\ &= x_1 \cdot s \cdot 1 + x_0 \cdot s' \cdot 1 \text{ (por el axioma 5)} \\ &= x_1 \cdot s + x_0 \cdot s' \text{ (por el axioma 2)} \end{aligned}$$

Otra técnica (que se explica en el siguiente capítulo) involucra el uso de tablas o mapas especiales.

Cada expresión booleana corresponde a un circuito lógico único, y la forma más sencilla de explicar esta correspondencia formalmente es imitar la definición de una expresión booleana. Así como se construye la expresión, se construye el circuito:

- Una sola línea etiquetada x es un circuito lógico. Un extremo se considera como la entrada, y el otro extremo como la salida.
- Si A y B son circuitos lógicos con salidas a y b respectivamente, entonces también lo son los circuitos que se muestran en la figura 2.2.
- Nada más es un circuito lógico.

En la figura 2.2, la compuerta **OR** (que tiene forma de escudo) representa la operación $A + B$: su salida es 1 si y solo si A ó B entregan una entrada 1 a la compuerta. La compuerta **AND** (que tiene forma de semicírculo) representa la operación $A \cdot B$: su salida es 1 si y solo si A y B entregan un 1 a la compuerta. El inversor representa la operación A': cambia el valor entregado de 1 a 0, o viceversa.

Las convenciones para simplificar la representación de circuitos lógicos es muy similar a las utilizadas para eliminar los paréntesis de una expresión booleana. Esto permite tener compuertas, por ejemplo, con más de una entrada, lo que resulta en un circuito como el que se muestra en la figura 2.3, que corresponde a la expresión del multiplexor en la forma normal disyuntiva.

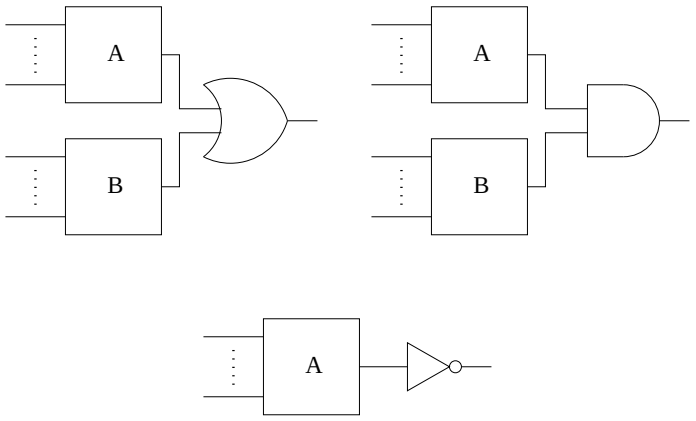


Figura 2.2: Circuitos lógicos formales

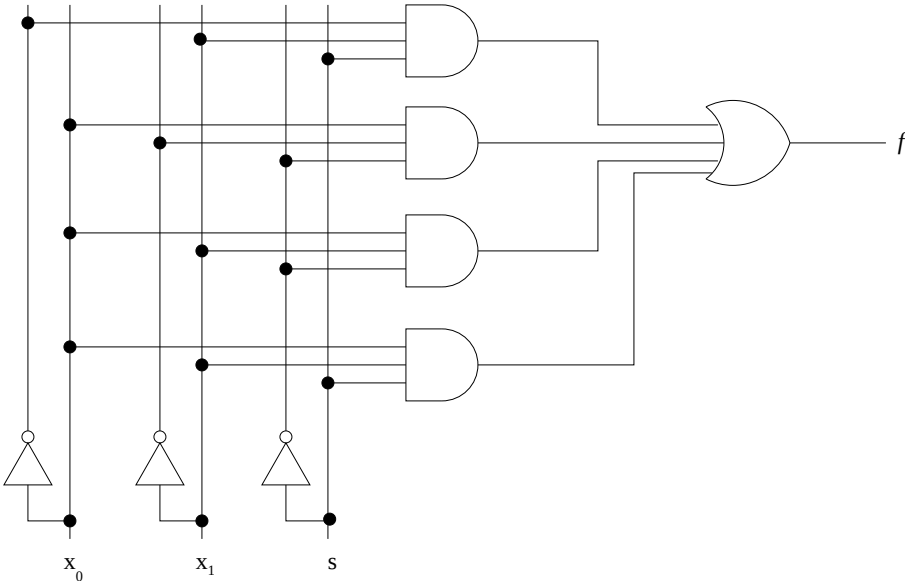


Figura 2.3: Un circuito multiplexor complicado

La complejidad de este circuito es obvia a la vista. Objetivamente, este circuito tiene 8 compuertas. Compárese este circuito con el que se muestra en la figura 2.4, que corresponde a la versión simplificada de la expresión booleana para el multiplexor.

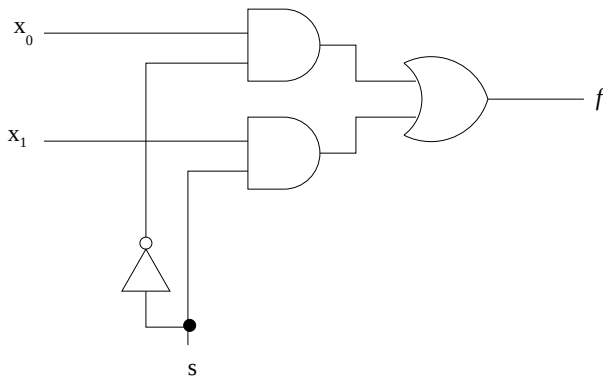


Figura 2.4: Un circuito multiplexor sencillo

Este circuito tiene 4 compuertas, y es claramente más sencillo aun cuando realiza la misma función. En verdad, es más factible consi-derar este circuito para la implementación de un multiplexor que la versión anterior.

No sería completamente cierto decir que las computadoras usan compuertas **AND**, **OR** y **NOT** directamente. De hecho, los transistores de efecto de campo que se utilizan en la tecnología de silicio (véase el capítulo 7) corresponde a tipos de compuertas diferentes a las consideradas aquí, como puede ser la compuerta **NAND** (véase el capítulo 1), la cual realiza la función $(x \cdot y)'$.

Finalmente, cabe mencionar que el multiplexor que ha sido presentado en este capítulo se utiliza más adelante para el diseño de una computadora sencilla (véase el capítulo 6).

Capítulo 3

Mapas de Karnaugh
Minimización de Circuitos

Además de las computadoras, nuestra sociedad tecnológica se encuentra repleta de dispositivos que requieren un control lógico: máquinas vendedoras, sistemas de encendido y carburación de automóviles, elevadores, y otros. Todos estos dispositivos reciben y emiten señales que realizan una labor. Por ejemplo, el dispositivo de control de un elevador recibe peticiones de los diferentes pisos como entradas, y genera señales de control al motor como salida (figura 3.1). ¿Qué tan complicada es la función esencial de control? Sorprendentemente, ésta resulta ser relativamente simple.

El problema de minimizar un circuito lógico se conoce como *minimización booleana*, y una de las mejores técnicas para resolver pequeños ejemplos de problemas es la técnica de *Mapas de Karnaugh*. Tales mapas proveen de visualizaciones bidimensionales de funciones booleanas, que son relativamente fáciles de inspeccionar, guiando directamente a una expresión sencilla para la función; mientras más sencilla sea la expresión, más simple es el circuito (véase el capítulo anterior). Los mapas de Karnaugh resultan útiles para funciones con dos, tres y cuatro variables lógicas. Sin embargo, su complejidad aumenta considerablemente para funciones con cinco o más variables lógicas.

Específicamente, un mapa de Karnaugh es un arreglo bidimensional de celdas, cada una de las cuales representa un producto específico de las variables y sus complementos. Por ejemplo, un mapa de dos variables lógicas tiene cuatro celdas para los productos de sus dos variables.

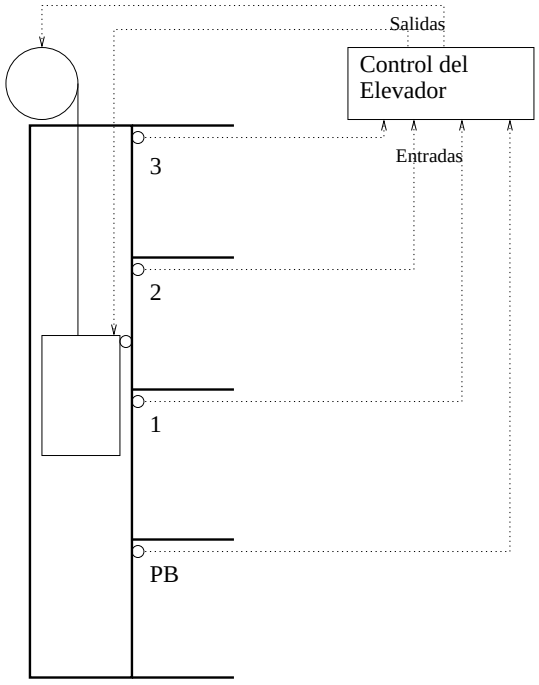
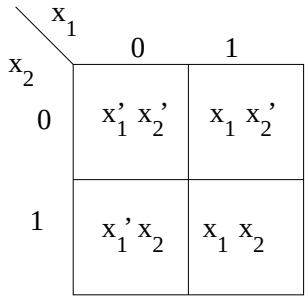


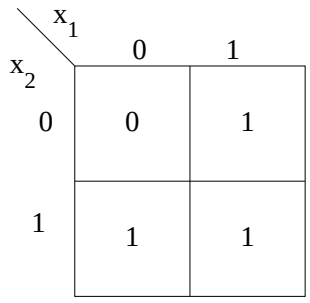
Figura 3.1: Un sistema elevador

Considérese, por ejemplo, dos variables x_1 y x_2. Los productos del mapa serían $x_1{}'x_2{}'$, $x_1{}'x_2$, $x_1x_2{}'$, y x_1x_2. El mapa se representa como sigue:



Cada renglón y columna del mapa corresponden a un valor de cada variable lógica. Estos valores se asignan de tal modo que produzcan un 1 cuando se substituyen en el producto que se encuentra en la intersección de su renglón y columna respectivos. Por ejemplo, el renglón $x_2 = 1$ intersecta la columna $x_1 = 0$ en $x_1{}'x_2$, y para estos valores $x_1{}'x_2 = 1$. La razón de este arreglo se hace notoria cuando se muestre cómo se usan los mapas para la minimización.

Considere la función $f(x_1, x_2)$ escrita en forma normal disyuntiva como $x_1x_2{}' + x_1{}'x_2 + x_1x_2$. Para cada producto que parece en esta expresión, se coloca un 1 en la celda correspondiente del mapa de Karnaugh de dos variables.



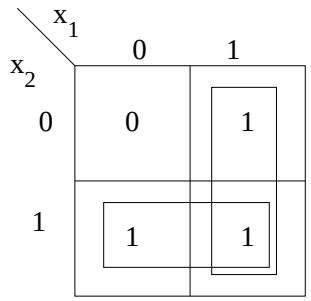
$f(x_1,x_2) = x_1\ x_2{}' + x_1{}'\ x_2 + x_1\ x_2$

Nótese que celdas adyacentes corresponden a productos con factores comunes. Por ejemplo, las dos celdas en la parte inferior corresponde a $x_1{}'x_2$ y x_1x_2. Estos productos tienen como factor común a x_2, por lo que podemos

escribir:

$$\begin{aligned} x_1x_2 + x_1{}'x_2 &= x_2(x_1 + x_1{}') \\ &= x_2 \end{aligned}$$

obteniendo una simplificación en parte de la expresión para f. De acuerdo con esto, podemos "ligar" las dos celdas con un rectángulo. De manera similar, es posible ligar los unos de la segunda columna.



En esencia, lo que se hace con la expresión para f en forma gráfica es lo siguiente:

$$\begin{aligned} f(x_1, x_2) &= x_1x_2{}' + x_1{}'x_2 + x_1x_2 \\ &= x_1x_2{}' + x_1x_2 + x_1{}'x_2 + x_1x_2 \\ &= x_1(x_2{}' + x_2) + (x_1{}' + x_1)x_2 \\ f(x_1, x_2) &= x_1 + x_2 \end{aligned}$$

Para evitar toda esta manipulación algebraica, pero llegar a la misma conclusión, la regla usada en mapas de Karnaugh es simplemente substituir *cada rectángulo* que dibujamos sobre el mapa por el factor constante de los productos correspondientes: el segundo renglón del mapa contiene a los productos $x_1{}'x_2$ y x_1x_2, de los cuales x_2 se considera la parte constante. Respecto al rectángulo vertical de la segunda columna, éste contiene a $x_1x_2{}'$ y a x_1x_2, de donde la parte constante es x_1. De esta forma, directamente del mapa es posible deducir la minimización de la función como:

$$f(x_1, x_2) = x_1 + x_2$$

Los mapas de Karnaugh de tres variables son doblemente complicados que los mapas para dos variables. Aquí, las celdas corresponden a todos los productos posibles de las tres variables (por ejemplo, x_1, x_2 y x_3) y sus complementos.

x_1 \ x_2 x_3	00	01	11	10
0	$x_1' x_2' x_3'$	$x_1' x_2' x_3$	$x_1' x_2 x_3$	$x_1' x_2 x_3'$
1	$x_1 x_2' x_3'$	$x_1 x_2' x_3$	$x_1 x_2 x_3$	$x_1 x_2 x_3'$

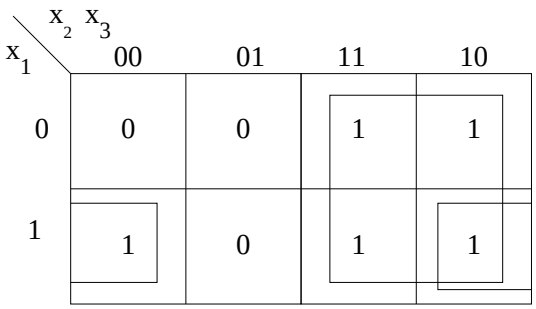
En este mapa, los renglones corresponden a la variable x_1, pero las columnas corresponden a pares de valores de las variables x_2x_3. De nuevo, estos valores se asignan de tal manera que el producto de la intersección de un renglón dado y una columna toma el valor de 1 cuando las substituciones correspondientes se hacen en el producto. Al mismo tiempo, los pares de valores asignados a las columnas tienen la interesante propiedad de que sólo un bit cambia cuando se va de un par al siguiente. Esto es cierto aun cuando se vaya del último par de regreso al primero. Tal secuencia es un ejemplo sencillo del código Gray reflejado. Si uno examina cualquier par de celdas adyacentes en el mapa, es notorio que el producto correspondiente tiene dos variables en su parte común. Es también cierto que los productos que residen dentro de una configuración cuadrada de cuatro celdas adyacentes tienen solo una variable como parte común.

Considérese la siguiente función:

$$f(x_1, x_2, x_3) = x_1' x_2 x_3 + x_1' x_2 x_3' + x_1 x_2 x_3 + x_1 x_2 x_3' + x_1 x_2' x_3'$$

Cuando se llena el mapa de Karnaugh para tres variables, escribiendo un 1 donde la función tiene un producto, es notorio que todos los 1s se pueden cubrir con dos rectángulos.

Uno de los rectángulos "envuelve" al mapa, cubriendo tanto la primera columna como la cuarta, agrupando los dos 1s que las celdas contienen. Tal y como sucede en el mapa de dos variables, se extraen las partes comunes de todos los productos contenidos en un rectángulo. El cuadrado resulta



simplemente en la expresión x_2, mientras que el rectángulo horizontal resulta en la expresión $x_1x_3{}'$. Por lo tanto, la función minimizada resulta:

$$f(x_1, x_2, x_3) = x_1x_3{}' + x_2$$

La reducción resultante de la configuración cuadrada de celdas puede ser fácilmente confirmada mediante la siguiente manipulación algebraica:

$$\begin{aligned} x_1{}'x_2x_3 + x_1{}'x_2x_3{}' + & \\ \quad x_1x_2x_3 + x_1x_2x_3{}' &= x_1{}'x_2(x_3 + x_3{}') + x_1x_2(x_3 + x_3{}') \\ &= x_1{}'x_2 + x_1x_2 \\ &= (x_1{}' + x_1)x_2 \\ &= x_2 \end{aligned}$$

Los mapas de Karnaugh de cuatro variables son el doble de complicados que los mapas de tres variables. Los productos son tan largos, que resulta más conveniente numerarlos de acuerdo a la combinación de valores por el que se les indexa. Por ejemplo, en lugar de escribir $x_1x_2{}'x_3{}'x_4$, escribimos el equivalente decimal a 1001, que es 9.

Usando la misma secuencia de código Gray reflejado tanto en renglones como columnas, las propiedades deseables de reducción de celdas adyacentes se preserva: es posible tener 2, 4, 8, o hasta 16 celdas adyacentes en varias configuraciones rectangulares. Aquí, es importante recordar en todo momento que se busca encontrar el número más pequeño de rectángulos que cubran el mayor número de 1s para una función dada. Entonces, la expresión reducida correspondiente se escribe en forma de suma. Esto implica la forma minimizada de la función.

$x_1\ x_2$ \ $x_3\ x_4$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	10	11

Regresando ahora al ejemplo del elevador, se analizan las operaciones lógicas requeridas y se usa la función de cuatro variables resultante para llenar el mapa de Karnaugh correspondiente.

Las llamadas al elevador requiriendo servicio se codifica para todos los cuatro pisos considerados mediante un número binario de 2 bits, como sigue:

x_1	x_2	**Piso**
0	0	Planta Baja
0	1	Primer piso
1	0	Segundo piso
1	1	Tercer piso

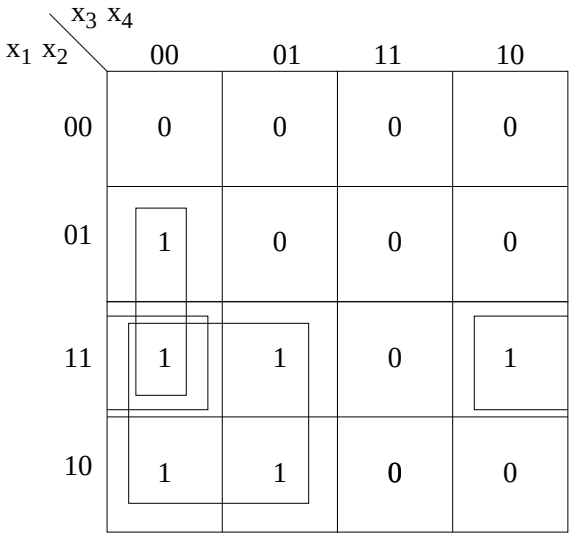
De manera similar se codifica la posición del elevador, pero utilizando otras dos variables:

x_3	x_4	**Piso**
0	0	Planta Baja
0	1	Primer piso
1	0	Segundo piso
1	1	Tercer piso

Las salidas del circuito de control deben considerar señales para hacer que el elevador vaya hacia arriba (*up*), hacia abajo (*down*), y para detenerse

(*stop*). En el presente ejemplo sólo se desarrolla el circuito para hacer que el elevador vaya hacia arriba.

El mapa de Karnaugh de cuatro variables se llena por inspección. Cada celda se examina, y si la posición actual se encuentra debajo del piso requerido, se coloca un 1 en tal celda; de otra forma, se coloca un 0.



Tres rectángulos son suficientes para cubrir todos los 1s del mapa, y la expresión resultante para *up* es:

$$up(x_1, x_2, x_3, x_4) = x_1{x_3}' + x_2{x_3}'{x_4}' + x_1x_2{x_4}'$$

Aún cuando la expresión resultante es la mínima forma disyuntiva, todavía es posible hacer alguna simplificación algebraica:

$$x_1{x_3}' + x_2{x_3}'{x_4}' + x_1x_2{x_4}' = x_1{x_3}' + x_2{x_4}'({x_3}' + x_1)$$

El circuito correspondiente tiene tan solo siete compuertas lógicas (figura 3.2).

Como puede observarse, el tamaño de un mapa de Karnaugh se dobla con cada variable que se añade. Debido a esto, y por la creciente complejidad de configuraciones disponibles, el método no se utiliza mucho cuando se involucran más de seis variables. Debe mencionarse, más aún, que los mapas de Karnaugh tienen la intención de usarse exclusivamente por diseñadores

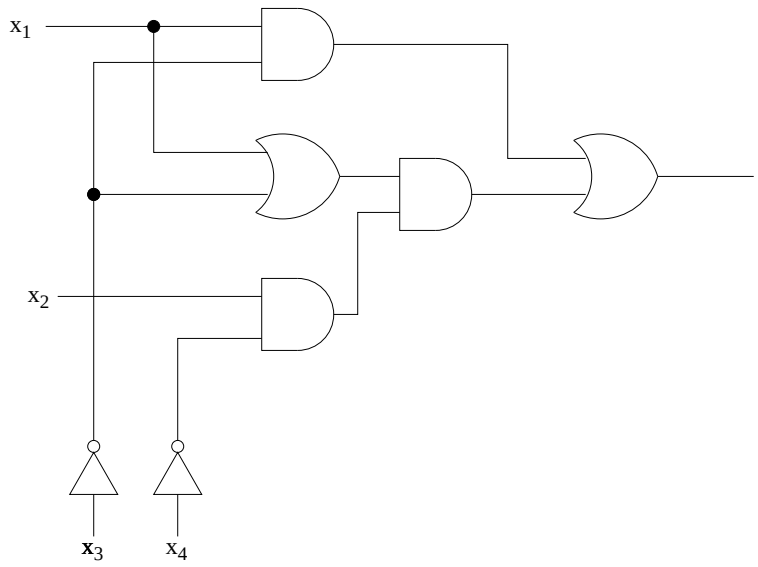


Figura 3.2: Un circuito de control para subir un elevador

humanos. Debido a la habilidad humana de tomar grandes cantidades de información visualmente, el método funciona más rápido que un proceso de revisar líneas de expresiones algebraicas buscando factores comunes. Al mismo tiempo, se han desarrollado en la actualidad elaborados algoritmos para la reducción de funciones booleanas.

Capítulo 4

Codificadores y Multiplexores

Manipulando la Memoria

Hay una clase de circuitos lógicos que se ocupan principalmente de manipular el flujo de información dentro de las computadoras. Esta clase de circuitos comprende a los codificadores, multiplexores, y otros dispositivos relacionados con éstos. Antes de introducirlos, se provee a continuación un ejemplo de su utilidad.

Puede pensarse que la memoria de una computadora consiste en miles de registros, y que cada registro (o "palabra") consiste en un número determinado de bits. Por ejemplo, una memoria muy simplificada se muestran en la figura 4.1. Esta memoria contiene 8 palabras (cada una de 4 bits), un registro de dirección de memoria (*Memory Address Register*, o MAR) y un registro separador de memoria (*Memory Buffer Register*, o MBR).

Cuando un palabra específica de datos debe ser almacenada en la memoria de una computadora, debe colocarse en el MBR antes de ser almacenada. Mientras tanto, el MAR se carga con la dirección en la memoria donde la palabra debe ser colocada.

El MBR se conecta mediante líneas de entrada y salida a cada palabra de la memoria; lo único que se requiere para almacenar el contenido de la MBR (por ejemplo, 0111) es que se envíe una señal al registro en la dirección para que cargue la información que el MBR le hace disponible. La señal para cargar los contenidos del MBR en memoria proviene del MAR, a

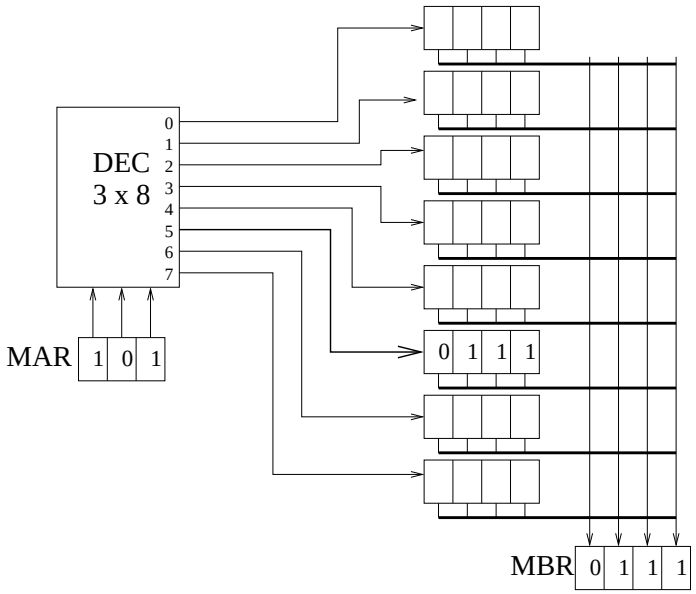


Figura 4.1: Una memoria muy simplificada

través de uno de los dispositivos que se mencionan en este capítulo, llamado "decodificador" (o simplemente, DEC). En el ejemplo, este dispositivo toma tres entradas del MAR, que representan a la dirección en memoria, y las decodifica. Esto significa que una y solo una de las ocho líneas de salida, la que corresponde al patrón de bits particular del MAR, se activa. En el ejemplo, si el MAR contiene el patrón 101, la salida del DEC etiquetada como 5 se activa con un 1, manteniendo las demás salidas del DEC con un valor de 0. La señal de activación llega a la memoria en su palabra 5, causando que el registro cargue los contenidos del MBR. Por lo tanto, al final de la operación, el registro de memoria 5 contiene la palabra 0111.

Los codificadores y decodificadores son opuestos, en cierto sentido. Un codificador convierte información que llega en 2^n señales de entrada a n señales de salida, mientras que el decodificador hace lo opuesto. La figura 4.2 muestra los circuitos de un codificador y decodificador, utilizando compuertas de lógica estándar.

De la figura, en el caso del codificador 4×2 (cuatro entradas, dos salidas), se supone que cada una de las cuatro entradas toma el valor de 1, y el resto son 0. Las líneas de salida llevan la misma información, es decir, ¿cuál de

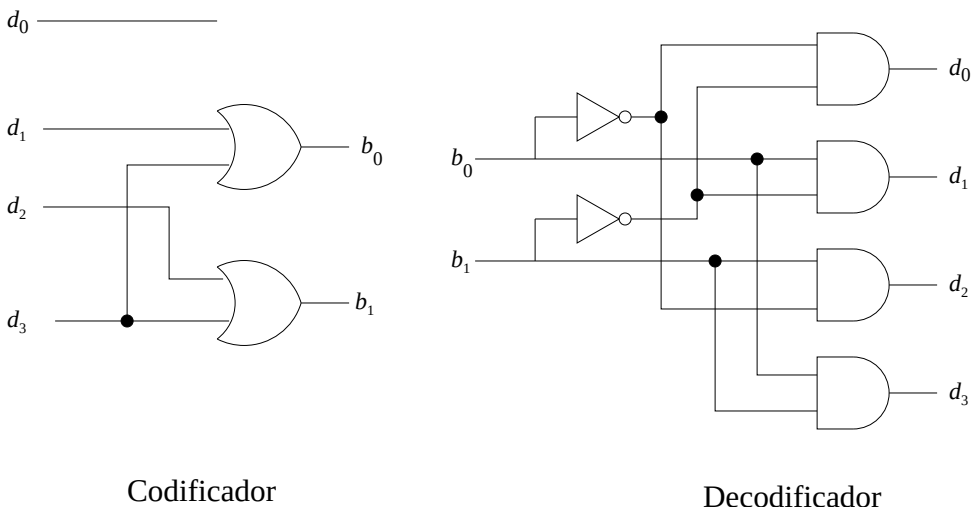


Figura 4.2: Un condificador y un decodificador

d_0, d_1, d_2 o d_3 lleva el 1? Esto se expresa en forma codificada: si $d_0 = 1$, entonces $b_0 = b_1 = 0$; si $d_1 = 1$, entonces $b_0 = 1$ y $b_1 = 0$; si $d_2 = 1$, entonces $b_0 = 0$ y $b_1 = 1$; finalmente, si $d_3 = 1$, entonces $b_0 = b_1 = 1$. En todos los casos, los números binarios $b_1b_0 = 00, 01, 10, 11$ codifican a las líneas d_0, d_1, d_2 y d_3, respectivamente.

Por otro lado, el decodificador 2×4 (dos entradas, cuatro salidas) toma el mismo tipo de señal binaria como entrada, y la decodifica en cuatro líneas de salida. Cada una de las compuertas **AND** responde a una combinación específica de valores de entrada: $d_0 = 1$ si $b_1b_0 = 00$; $d_1 = 1$ si $b_1b_0 = 01$; $d_2 = 1$ si $b_1b_0 = 10$; y $d_3 = 1$ si $b_1b_0 = 11$.

Los multiplexores y demultiplexores son esencialmente interruptores que determinan, para cada estado de un circuito lógico o una computadora, cuál ruta siguen los datos entre los varios componentes digitales. Un multiplexor tiene 2^n líneas de entrada (para un valor "modesto" de n), n líneas de selección, y una línea de salida. La combinación binaria sobre las líneas de selección determina cuál de las 2^n entradas se rutea a través de la única salida de este dispositivo. La figura 4.3 es un multiplexor con cuatro líneas de entrada y dos líneas de control. Esta últimas son necesarias para seleccionar cuál de las cuatro entradas se conecta a la salida del circuito.

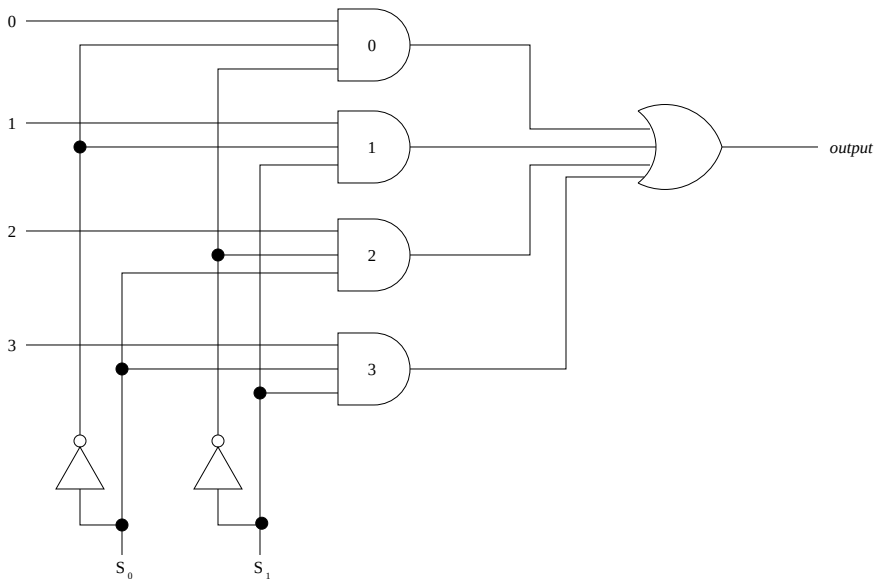


Figura 4.3: Un multiplexor de 4×1

Para cada combinación de los valores binarios de las dos líneas de control, exactamente una de las cuatro compuertas **AND** recibe dos 1s a su entrada. El tercer valor binario se recibe de cada una de las entradas del circuito, ya sea 0 o 1, el cual se transmite a través de la compuerta **OR** hacia la salida. De hecho, cuando las entradas de selección s_1 y s_0 llevan el equivalente binario a i, entonces la i-ésima entrada se transmite a través del dispositivo. Tal función de interrupción y transmisión es muy útil para las computadoras. Por ejemplo, dentro de una computadora hay varias posibles fuentes de datos a ser cargados en el MBR. Cada una de estas fuentes envía una línea de entrada a un multiplexor, que a la vez conecta su única salida al MBR. La lógica de control de la computadora decide en un momento dado cuál fuente de datos usar como entrada para el MBR, de tal modo que envía las señales apropiadas a las líneas de control del multiplexor para conectar a una fuente particular con el MBR.

Un demultiplexor, a la inversa del multiplexor, tiene una sola línea de entrada que, bajo la dirección de sus líneas de control, selecciona una de varias salidas para la transmisión de una señal binaria.

Con codificadores, decodificadores, multiplexores y demultiplexores, se maneja y controla virtualmente todo flujo y toda comunicación de datos de una parte de la computadora a otra.

Capítulo 5

Circuitos Secuenciales
La Memoria de la Computadora

En el campo del diseño lógico existe una distinción fundamental entre circuitos combinacionales y circuitos secuenciales (figura 5.1). El primero puede ser representado por una "caja negra" con líneas de entrada de un lado y líneas de salida por el otro. Dentro de la caja negra hay un circuito lógico (véase el capítulo 2) compuesto de compuertas lógicas; cada trazo lógico en el circuito lleva de una entrada a una salida, y no hay trazos lógicos circulares. El segundo tipo de circuitos contienen elementos de memoria de dos estados, los cuales pueden almacenar o "recordar" el estado durante un periodo de tiempo.

En un circuito combinacional, un patrón de bits en la entrada genera siempre la misma salida. Esta propiedad no se cumple para un circuito secuencial: el contenido actual de la memoria forma una entrada adicional al circuito lógico, y la salida puede cambiar de una ocasión a la siguiente aún con la misma entrada. La habilidad de modificar su operación con el tiempo es una característica distinguible en las computadoras. El propio tiempo es dividido en pasos ("discretizado") mediante pulsos que emanan de un circuito de reloj. Estos pulsos coordinan la actividad dentro de una computadora, y dentro de los (en cierto modo) sencillos circuitos secuenciales que se describen aquí.

A la unidad básica de memoria se le puede referir como una *celda*. Hay varias formas de diseñar celdas a partir de compuertas lógicas simples. Sin embargo, estas descripciones de celdas se usan mayormente para propósitos ilustrativos, y no corresponden propiamente a las verdaderas celdas de

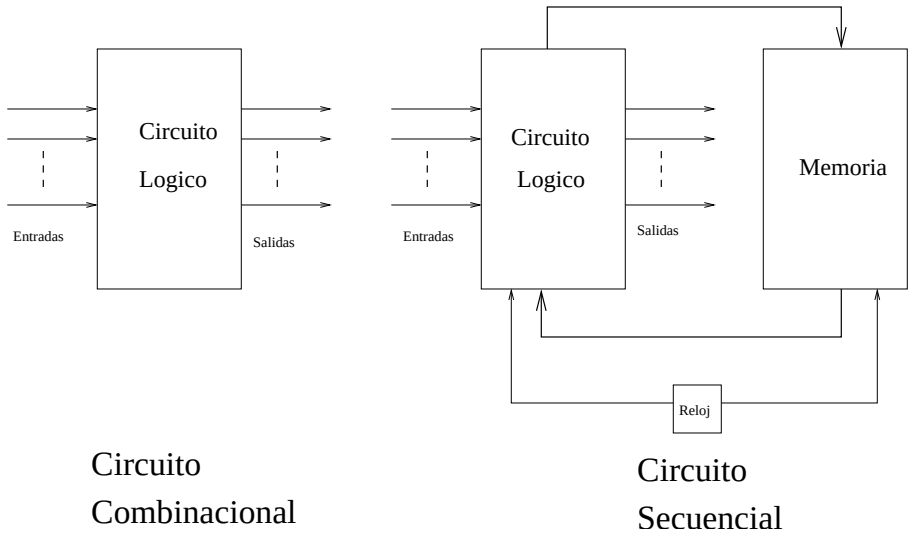


Figura 5.1: Circuitos Combinacional y Secuencial

memoria en computadoras reales, excepto en su función. Una celda real se fabrica generalmente utilizando tecnología sobre silicio a partir de una colección de transistores y otros componentes electrónicos (véase el capítulo 7).

Siguiendo la aproximación inicial, se mantiene la ficción de que todos los componentes de los dispositivos considerados se realizan a partir de compuertas básicas como **AND**, **OR** y **NOT**, o tal vez por compuertas **NAND** e inversores. De hecho, esto no es propiamente erróneo, ya que al nivel del diseño de hardware, es fácil de traducir una especificación de compuertas a una descripción de transistores. En verdad, en ciertos casos, algunos tipos de transistores resultan compuertas.

Una celda de memoria muy simple, llamada *flip-flop RS*, puede hacerse a partir de dos compuertas **NAND**, como se muestra en la figura 5.2. Este sencillo circuito no es combinacional por la definición dada anteriormente, ya que efectivamente contiene un trazo circular. La salida de cada compuerta **NAND** se conecta como la entrada a la otra compuerta. Esto hace el análisis de este circuito algo difícil. Por eso, en lugar de comenzar por el lado de la entrada, la explicación de su funcionamiento comienza del lado de la salida. ¿Pueden ambas salida Q y Q' ser 0 simultáneamente? Evidentemente, no,

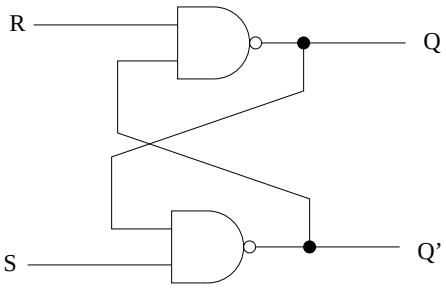


Figura 5.2: Un flip-flop RS

ya que una compuerta **NAND** debe dar a su salida un 1 si cualquiera de sus entradas es 0. Se debe entonces tener cuidado para asegurar que las entradas R y S nunca sean simultáneamente 0. Suponiendo, entonces, que sólo dos estados son posibles: $Q = 1$, $Q' = 0$ (estado 1) y $Q = 0$, $Q' = 1$ (estado 0), entonces es posible expresar el funcionamiento del flip-flop RS en términos del autómata finito de la figura 5.3.

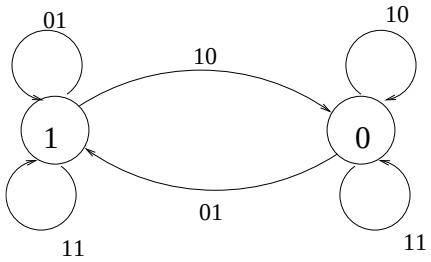


Figura 5.3: Un flip-flop RS como autómata finito

Las etiquetas de los arcos codifican las entradas al flip-flop como un número binario de dos bits SR. Cuando el flip-flop está en el estado 0, si al bit S se le introduce un 1 y al bit R se le da un 0 ($SR = 10$) en forma simultánea, entonces el dispositivo "salta" al estado 1. De ahí, un 1 en R y un 0 en S ($SR = 01$) hace "saltar" al circuito de regreso al estado 0. Por esto, el curioso nombre de este circuito: "flip" levanta la salida Q del estado 0 al estado 1, mientras que "flop" hace caer la salida Q del estado 1 al 0.

Como se menciona anteriormente, los circuitos secuenciales normalmente incorporan una señal de reloj. Por lo tanto, es necesario considerar una

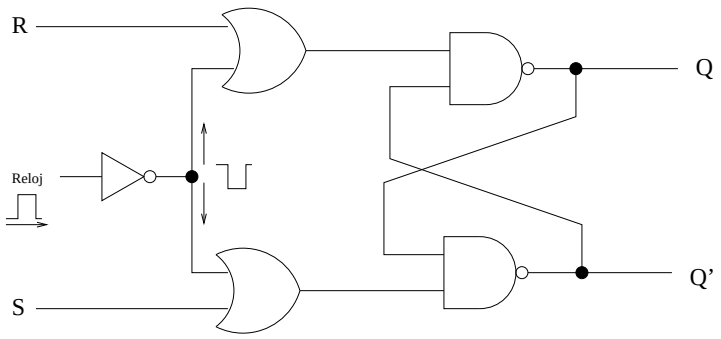


Figura 5.4: Un flip-flop RS con entrada de reloj

entrada de reloj para el flip-flop RS, lo que hace que el circuito se dispare sólo durante un pulso de reloj. El circuito de la figura 5.4 añade una entrada de reloj al flip-flop RS.

El pulso de reloj (que se supone llega como un 1) se dice que "habilita" a la señal de S o la señal de R, por que sólo cuando las dos compuertas **OR** de la figura 5.4 reciben un 0 (invertido del 1 del reloj) pueden ambas señales pasar por la propia compuerta. La señal S lleva entonces al circuito al estado 1, si éste no se encuentra ya en tal estado. Esto "instruye" al flip-flop, en efecto, a "recordar 1". Similarmente, la señal R le "instruye" al circuito a "recordar 0".

Si el circuito del flip-flop RS se encierra dentro de una caja negra con las etiquetas apropiadas para entradas y salidas, es posible diseñar varias formas de memoria además de la celda mencionada anteriormente. Por ejemplo, toda computadora contiene un número de "registros de trabajo", los cuales mantenien o cambian los datos en promedio más frecuentemente que una celda específica de memoria. La figura 5.5 muestra los primeros dos bits de un registro de n bits.

Este registro está organizado verticalmente. Cada caja representa un flip-flop, el cual cambia de estado sólo cuando llega un pulso de reloj (CLK). Los bits individuales se almacenan en cada flip-flop del registro, y en conjunto, representan un valor numérico binario. El contenido puede también cambiarse sólo cuando la línea de carga (LOAD) se activa con un 1. Esto causa que las salidas de las dos compuertas **NAND** que alimentan a un solo flip-flop, ambas normalmente con salida 1, transmitan una forma inversa de

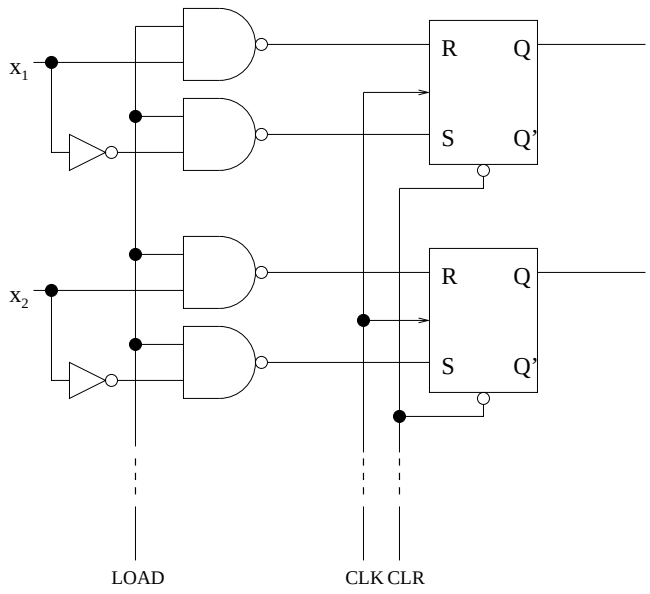


Figura 5.5: Los primeros dos bits de un registro de n bits

la entrada que llega por las líneas x_i, así como su negación. Por lo tanto, si $x_1 = 1$, y la señal de carga (LOAD) es 1, la entrada en R se hace 0 mientras que la entrada en S es 1. Esto hace que la salida Q del flip-flop tome el valor de 1, aun cuando éste no sea su valor original.

De esta forma, los pulsos que llegan simultáneamente a las líneas de entrada del registro x_1, x_2, ... x_n pueden cargarse en paralelo en el registro. Las líneas de salida reflejan el contenido actual del registro, y pueden leerse en cualquier momento. El registro además cuenta con una línea de aclarado (CLEAR), la cual tiene la función de poner todos los bits del registro en 0.

A continuación, se considera el diseño de una celda de memoria. Para que una celda de memoria resulte útil, es necesario que se pueda tanto escribir como leer en ella. El diseño de una celda de memoria en términos de compuertas y flip-flops se muestra en la figura 5.6.

Este diseño es muy similar al del registro basado en flip-flops, y utiliza la misma técnica de habilitación basado en compuertas **NAND**. Su operación es como sigue: primero, la celda debe ser seleccionada mediante colocar un 1 en la línea de selección. Esto habilita que la señal de entrada llegue al flip-

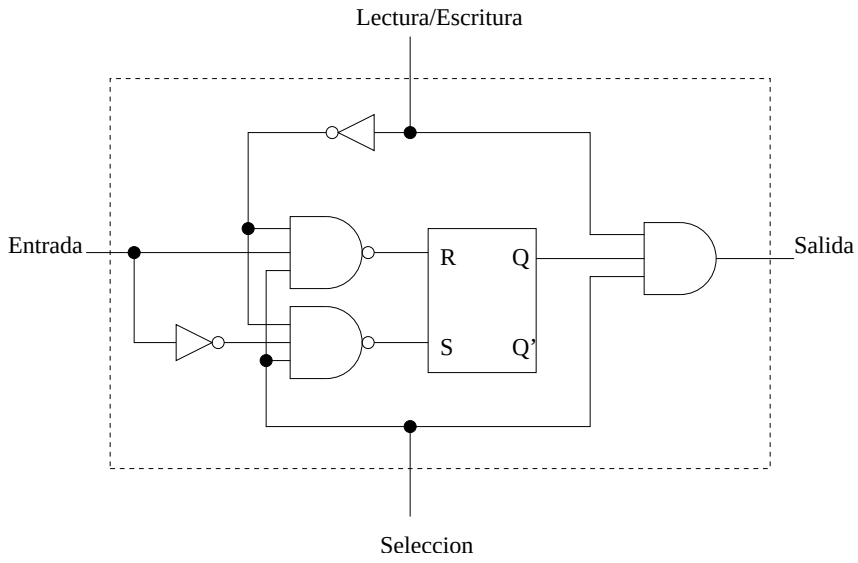


Figura 5.6: Una celda de memoria

flop, ya que habilita ambas compuertas **NAND**. Más aún, un 1 en la línea de selección permite transmitir el contenido actual del flip-flop a la salida de la celda. En seguida, se puede realizar entonces una operación de lectura o escritura. Cuando se intenta escribir un bit en la celda de memoria, se debe colocar un 0 en la entrada de lectura/escritura. Esta señal se invierte antes de llegar a las compuertas **NAND**, así que las habilita conjuntamente con el 1 de la línea de selección. Esto permite que el valor a la entrada de la celda pase a las entradas R y S del flip-flop, y modifique su estado. Por otro lado, un 1 en la entrada de lectura/escritura deshabilita las entradas al flip-flop, y habilita la compuerta **AND** a su salida. Por lo tanto, un 1 en lectura/escritura significa una operación de lectura de la celda.

Al representar a tal celda de memoria como una caja negra, es posible ahora diseñar una pequeña memoria de ocho palabras de 4 bits. como se muestra en la figura 5.7. Sólo las dos primeras y la última de las palabras se muestran, como renglones de cuatro celdas cada una. El circuito tiene cuatro líneas de entrada, una línea de lectura/escritura, y tres entradas de dirección a un decodificador 3×8. Las salidas del decodificador encienden en forma horizontal a las celdas de una palabra utilizando la línea de selección de cada celda. Para localizar una palabra, sólo tres bits son necesarios como

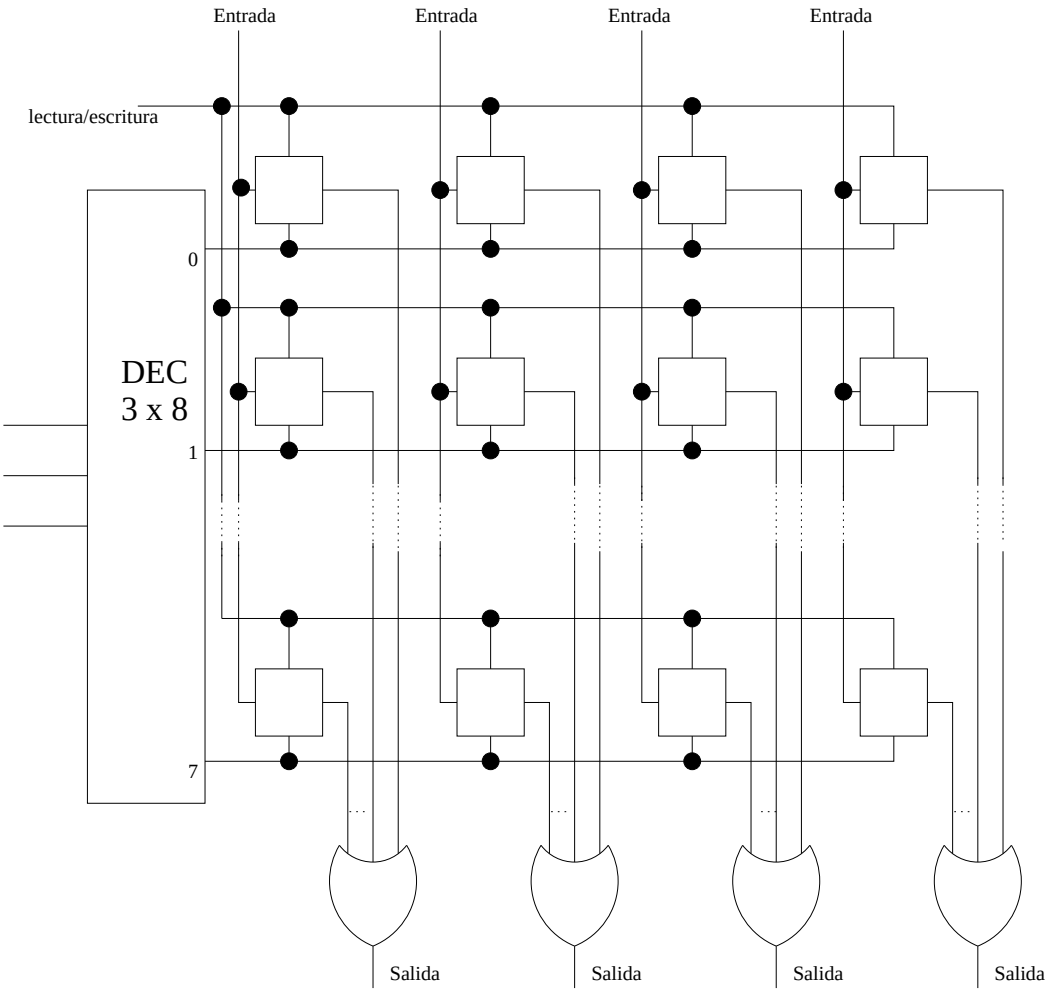


Figura 5.7: Una memoria pequeña pero completa

entrada para el decodificador. Tan pronto como una dirección (numerada del 0 al 7) se da a la entrada del decodificador, la palabra cuya dirección se especifica se selecciona por el mecanismo ya descrito: la línea de selección de cada celda se pone en 1.

Dado que una palabra en particular se ha seleccionado, la presencia de un 0 o un 1 en la línea de lectura/escritura determina si se escribe un valor de 4 bits en la dirección seleccionada a partir de las líneas de entrada, o se lee el valor ya contenido ahí por las líneas de salida. Las salidas de todas las celdas de memoria por columna se dirigen a una sola compuerta **OR**, y éstas se conectan a un registro especial, como se describe en el siguiente capítulo.

Capítulo 6

La SCRAM
Una Computadora Simple

Los principios que gobiernan el diseño y operación de una computadora real pueden ilustrarse mediante la descripción completa de una computadora "de juguete". Nadie sugeriría seriamente que la computadora que se muestra en la figura 6.1 fuera construida, ya que su pequeña cantidad de memoria la hace inútil para la mayoría de los programas prácticos. Sin embargo, esta sencilla computadora tiene todas las principales características de máquinas más grandes y sofisticadas; solo se ha omitido el hardware de entrada y salida. En resumen, es una máquina simple de acceso aleatorio completa (*simple complete random access machine*, o simplemente SCRAM).

Antes de adentrarse en los detalles de la SCRAM, es necesario hacer una descripción previa de sus componentes principales. Todos aquellos componentes que exhiben el acrónimo MUX son multiplexores (véase el capítulo 4). Este tipo de dispositivo sirve para "dirigir" datos de varias fuentes a un solo destino. La selección de la fuente depende de la señalización generada por la Unidad de Control.

Además de multiplexores, siete registros aparecen en el diagrama: el contador de programa (*program counter*, o PC), el registro de instrucción (*instruction register*, o IR), el registro de dirección de memoria (*memory address register*, o MAR), el registro separador de memoria (*memory buffer register*, o MBR), el acumulador (*accumulator*, o AC) y el registro sumador (*adder*. o AD). Este último se incorpora dentro de la Unidad Lógico-Aritmética (*Arithmetic Logic Unit*, o ALU). Un contador T genera los pulsos que son decodificados en líneas de entrada separadas para varios destinos en la Uni-

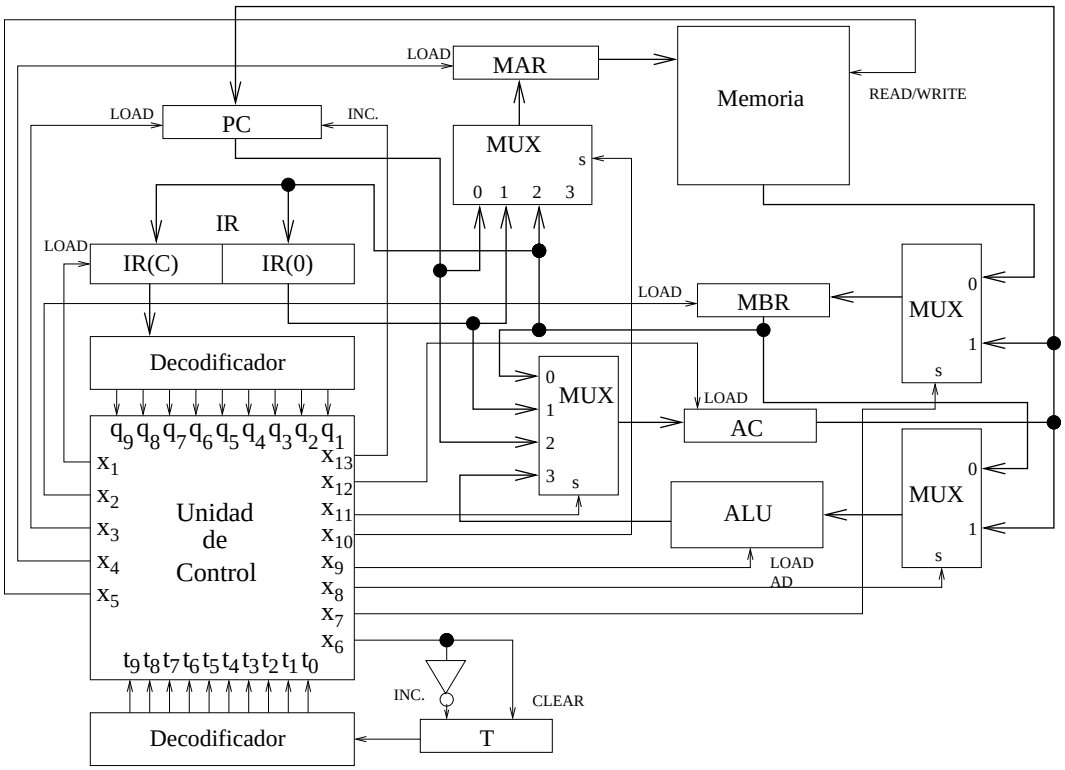


Figura 6.1: Estructura de la SCRAM

dad de Control (*Control Logic Unit*, o CLU). Otro decodificador se encarga de traducir las instrucciones de software en el IR para su uso en la CLU.

El diagrama es enteramente esquemático; no hay casi relación entre su geometría y la implementación con circuitos integrados de silicio sobre una tarjeta. Ciertamente, la memoria sería mucho más grande en un circuito integrado real. Como una regla general, los rectágulos alargados como reglas representan registros, y todo lo demás representa lógica de control o memoria.

La SCRAM tiene palabras de 8 bits, y su memoria contiene hasta 16 palabras. Por tanto, 4 bits son suficientes para especificar la dirección de cualquiera de las palabras en memoria. Como regla general, cada palabra de memoria representa un dato (como valor numérico) o una instrucción. El dato puede tener hasta 8 bits de longitud. Por otro lado, se ha tomado la

decisión de diseño que una instrucción debe repartir sus 8 bits en un *código de operación* (4 bits) y un *operando* (4 bits). En máquinas normales de 8 bits, las operaciones y operandos se almacenan en localidades alternativas de memoria, pero los circuitos para soportar esto deben ser más complejos. La SCRAM es demasiado simple para tal sofisticación, y el resultado es que sólo números de 4 bits pueden ser utilizados como operandos de cualquier instrucción.

A continuación se analiza un ciclo completo de operación para cada instrucción de un lenguaje de muy bajo nivel, cuyas instrucciones, sus códigos, operandos y descripciones se listan en la siguiente tabla:

Operación	**Código**	**Operando**	**Descripción**
LDA	0001	X	Carga el contenido de la dirección de memoria X al AC.
LDI	0010	X	Carga indirectamente el contenido de la dirección X al AC.
STA	0011	X	Almacena el contenido de AC en la dirección de memoria X.
STI	0100	X	Almacena indirectamente el contenido de AC en la dirección X.
ADD	0101	X	Suma el contenido de la dirección X con AC.
SUB	0110	X	Resta el contenido de la dirección de AC.
JMP	0111	X	Salta a la instrucción con dirección X.
JMZ	1000	X	Salta a la instrucción con dirección X si AC contiene 0.

El ciclo completo se compone de dos sub-ciclos, llamadas ciclo de búsqueda (*fetch*) y ciclo de ejecución (*execute*). El ciclo de búsqueda obtiene la siguiente instrucción ejecutable de la secuencia que se encuentra en proceso, y lo carga al IR. El ciclo mismo se escribe como una secuencia de operaciones de máquina elementales llamado *microprograma*. Cada línea de un microprograma se llama *microoperación*, y se escribe en términos de una notación o lenguaje especial llamado *Lógica de Transferencia de Registros* (*Register Transference Logic* o RTL).

El ciclo de búsqueda, en términos de RTL, se presenta de la siguiente forma:

$$\begin{aligned} t_0 &: MAR \leftarrow PC \\ t_1 &: MBR \leftarrow M, PC \leftarrow PC + 1 \\ t_2 &: IR \leftarrow MBR \end{aligned}$$

Todas las microoperaciones de la SCRAM se ponen en ejecución mediante el contador T, que alimenta de señales de tiempo t_i (a través de un decodificador) a la CLU (véase la figura 6.1). Considérese la primera línea del ciclo de búsqueda: cuando T contiene 0, la salida del decodificador conectado a T contiene un 1 en su salida 0. Esta salida pasa el 1 a la entrada t_0 de la CLU, lo que hace que los contenidos del PC se transfieran al MAR (que se simboliza por la línea $MAR \leftarrow PC$). Cuando t_0 vuelve a ser 0, el contador hace que el decodificador coloque un 1 en la entrada t_1, lo que inicia la siguiente micro-operación: los contenidos de la memoria en la dirección contenida en el MAR se transfieren al MBR. Al mismo tiempo, el PC se incrementa. Esto asegura que la siguiente instrucción a ejecutarse se encuentre almacenada en la siguiente dirección de memoria respecto a la instrucción actual (a menos que la instrucción actual sea un salto). Como se ha descrito anteriormente, se sabe que la instrucción actual consiste de un código de operación y un operando. Ambos son transferidos al IR desde el MBR cuando la entrada t_2 se activa. Los cuatro bits más significativos de IR comprenden un "subregistro" llamado IR(C). Los 4 bits menos significativos de IR son otro "subregistro" llamado IR(0) (véase de nuevo la figura 6.1). De este modo, al final del ciclo de búsqueda, IR(C) contiene el código de operación, e IR(0) contiene el operando.

El subregistro de código IR(C) se conecta mediante cuatro líneas paralelas (que se muestran en la figura 6.1 como una sola línea más gruesa) al decodificador de instrucciones, el cual produce un 1 en una de sus nueve salidas, conectadas a las entradas q_i de la CLU. Cada entrada representa a una de las nueve posibles instrucciones, cada una con su propio patrón binario característico.

Dentro del ciclo básico, después del ciclo de búsqueda sigue el ciclo de ejecución. En este momento, la entrada t_3 de la CLU recibe un 1 proveniente del decodificador. A partir de este paso, y en subsecuentes pasos del ciclo de ejecución, se realizan varias microoperaciones, dependiendo del tipo de instrucción que se ejecuta. Esto ya es posible conocerlo, dado que el objetivo

del ciclo de búsqueda es precisamente colocar en IR(C) el código de operación de la instrucción actual, y éste ya ha sido decodificado.

Cada instrucción descrita anteriormente se representa mendiante un microprograma. Por ejemplo, LDA tiene el siguiente microprograma:

$$\begin{aligned} q_1 t_3 &: MAR \leftarrow IR(0) \\ q_1 t_4 &: MBR \leftarrow M \\ q_1 t_5 &: AC \leftarrow MBR, T \leftarrow 0 \end{aligned}$$

Durante el ciclo de búsqueda, todo lo que se requería para realizar las microoperaciones en sus tres pasos era la presencia de un 1 secuencialmente en las líneas conectadas a t_0, t_1 y t_2. Para el ciclo de ejecución, se hace necesario una mayor cantidad de señales: si el IR(C) contiene el código de la instrucción LDA, esto significa que IR(C) está cargado con el valor binario 0001. El decodificador convierte este valor binario en un 1 sobre la línea q_1, con todas las otras líneas q_i conteniendo 0. Por lo tanto, la expresión lógica $q_1 t_3$ significa en efecto "si q_1 y t_3 entonces...". Más adelante se muestra cómo se implementa la generación de las señales pertinentes para hacer la transferencia.

Por tanto, cuando q_1 y t_3 son ambos 1, la SCRAM carga la porción del operando IR(0) en el MAR. En el siguiente paso, el contenido de tal dirección de memoria se carga en el MBR. En seguida, en el paso siguiente, MBR coloca su contenido en el registro acumulador AC, mientras que al mismo tiempo se hace que el contador recomience su cuenta desde cero ($T \leftarrow 0$).

El ciclo completo de la SCRAM puede requerir de hasta 10 periodos de tiempo consecutivos (que se simbolizan por las entradas t_0, t_1,..., t_9 de la CLU). Los periodos son cada uno, por ejemplo, de un microsegundo, y los genera el contador de tiempos T. La CLU incrementa T (basado en su propio reloj) entre las micro-operaciones, y lo limpia (realizando $T \leftarrow 0$) cuando un nuevo ciclo completo está por comenzar.

El siguiente ejemplo de microprograma es un poco más complicado. La instrucción LDI requiere de cinco microoperaciones:

$$\begin{aligned} q_2 t_3 &: MAR \leftarrow IR(0) \\ q_2 t_4 &: MBR \leftarrow M \\ q_2 t_5 &: MAR \leftarrow MBR \\ q_2 t_6 &: MBR \leftarrow M \\ q_2 t_7 &: AC \leftarrow MBR, T \leftarrow 0 \end{aligned}$$

La única diferencia entre las instrucciones de carga directa e indirecta (LDA y LDI, respectivamente) es que la segunda requiere un conjunto adicional de microoperaciones de lectura en memoria. Esto se debe a que la carga indirecta requiere de dos lecturas de memoria: la primera para obtener la dirección del dato, y la segunda para propiamente leer el dato.

En seguida se muestran los microprogramas para otras dos instrucciones. Después de los siguientes ejemplos, no resulta difícil construir microprogramas para el resto de las instrucciones. Primero, comencemos con la instrucción ADD:

$$\begin{aligned} q_5 t_3 &: MAR \leftarrow IR(0) \\ q_5 t_4 &: MBR \leftarrow M \\ q_5 t_5 &: AD \leftarrow MBR \\ q_5 t_6 &: AC \leftarrow AD + AC, T \leftarrow 0 \end{aligned}$$

Nótese que la instrucción ADD primero recoge el número almacenado en la dirección de memoria X, y entonces lo carga en el registro aritmético especial AD (que no se muestra propiamente en la figura 6.1, ya que está contenido en la ALU). Después de que el contenido del acumulador AC se suma al contenido de AD, el resultado se pone de nuevo en AC.

El siguiente microprograma representa a la instrucción JMP:

$$q_7 t_3 \quad : \quad PC \leftarrow IR(0), T \leftarrow 0$$

Este microprograma simplemente actualiza el contenido de PC con el operando de la instrucción. Por definición, PC contiene la dirección de la siguiente instrucción a ejecutarse. Si la instrucción actual no es un salto, el CLU simplemente incrementa el PC en algún punto de su ciclo de operación.

Ya para este punto, surge naturalmente la pregunta: ¿cómo es que los nueve diferentes microporgramas se implementan en circuitos lógicos reales?

Aun cuando la implementación de este tipo de circuitos se realiza, por su complejidad, mediante transistores en silicio, de nuevo aquí se presenta una aproximación que emplea compuertas lógicas estándar. En cualquier caso, es un ejercicio relativamente sencillo convertir estos circuitos en un conjunto lógico completo de compuertas (véase el capítulo 1).

Los circuitos lógicos individuales para realizar el ciclo de búsqueda para las nueve instrucciones del ciclo de ejecución son relativamente fáciles de implementar, como se muestra en la figura 6.2.

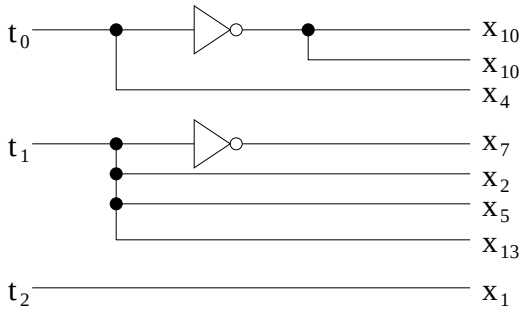


Figura 6.2: Circuitos lógicos para el ciclo de búsqueda

La línea x_{10} se considera como doble salida, ya que debe operar el multiplexor que tiene como destino al MAR con dos entradas. Cuando t_0 es 1, x_{10} toma el valor de 0, que se traduce en un valor 00 en las dos entradas de selección del multiplexor, causando que se seleccione la entrada 0 (proveniente del PC) para su transferencia al MAR. La línea x_4 causa que el MAR cargue de acuerdo con el diseño de registro del capítulo 5. Cuando t_1 es 1, x_7 es 0, y x_2, x_5 y x_{13} toman todos el valor de 1. Esto significa que el multiplexor que sirve como selección de entrada al MBR selecciona la entrada desde la memoria, se emite la señal de carga (LOAD) para el MBR y la línea de lectura/escritura (READ/WRITE) de la memoria tiene un valor para la lectura (es decir, 0), y a la vez, el PC se incrementa. Cuando t_2 es 1, el IR debe cargar el contenido actual del MBR.

Ahora bien, un circuito que implementa las señales de control para la ejecución de la instrucción LDA se muestra en la figura 6.3.

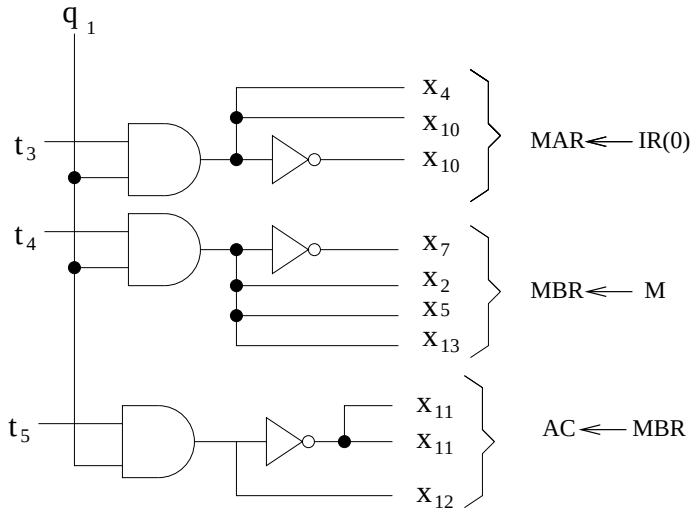


Figura 6.3: Circuitos lógicos para la carga del acumulador

Cuando se consideran en aislamiento, la operación de estos circuitos es clara. Primero, el par de líneas x_{10} controlan el multiplexor que selecciona la entrada al MAR. Cuando t_3 y q_1 son simultáneamente 1, las dos líneas de x_{10} son 01, y la línea x_4 hace cargar al MAR. Cuando t_4 y q_1 son 1, la línea x_7 manda una señal de control con valor de 0 para indicar al multiplexor que seleccione la entrada al MBR, y x_2, x_5 y x_{13} toman todos el valor

de 1, lo que implica una carga en MBR desde la memoria igual que en el ciclo de búsqueda. Finalmente, cuando t_5 y q_1 son 1, se envía x_{12} como la señal de carga a AC, y ambas líneas de control para el multiplexor x_{11}, que seleccionan la entrada a AC, son 0. En este caso, 00 en la entrada de selección del multiplexor significa que se selecciona al MBR.

En este punto, es notorio que algunas de las líneas de salida aparecen en los dos circuitos descritos. Ya que las salidas de dos circuitos no pueden ser conectadas directamente, deben usarse más compuertas para este propósito.

Capítulo 7

Computadoras VLSI
Circuitos en Silicio

En los últimos 20 años, el hardware de computadora ha ido a través de una verdadera revolución. Las computadoras básicas han migrado de las tarjetas de circuito impreso repletas de transistores, capacitores y resistencias, a una nueva tecnología de alambres ultrapequeños y componentes grabados en la superficie de circuitos de silicio. Esta última tecnología se le conoce como "de muy alta escala de integración" (*very large scale of integration*) o simplemente VLSI.

Los otros capítulos de estas notas se han dedicado a la lógica de la computadora, sin mucha atención al hardware específico de su implementación física. Las razones de esta aproximación recaen en la universalidad de la Lógica Booleana: no importa cómo se implemente la lógica, las funciones básicas como **AND** y **NAND** siguen teniendo vigencia debido a los principios de la Lógica Booleana (véase el capítulo 1).

Los circuitos VLSI se fabrican mediante sustratos de tres capas (figura 7.1). La capa superior se fabrica de un metal, como el aluminio, y la capa media está hecha de polisilicio. La capa más baja, llamada capa de difusión, es una pieza de cristal de silicio casi puro que es contaminado ("*doped*") con ciertas impurezas para producir las propiedades eléctricas deseables.

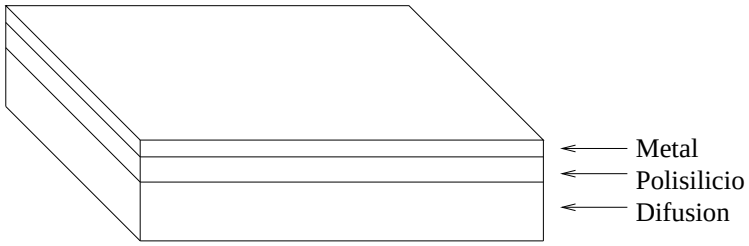


Figura 7.1: El sustrato físico

Entre cada par de capas adyacentes hay una delgada capa aislante de dióxido de silicio. Las capas se producen una a la vez, en patrones que reflejan el circuito a ser construido. Típicamente, los patrones se forman de delgadas tiras de material, llamadas trayectorias (*paths*).

Donde existe un cruce entre una trayectoria en la capa de polisilicio y una trayectoria en la capa de difusión, el voltaje de la primera controla el voltaje en la segunda. Esto da como resultado un transistor. La figura 7.2 muestra una porción muy ampliada de dos de esas trayectorias, separadas por la capa de dióxido de silicio. Las capas de la tecnología de circuito de silicio pueden construirse en patrones ya sea por adición o remoción selectiva de material. En el nivel más bajo de la porción aumentada de la figura 7.2, la capa de difusión no ha sido adicionada ni removida. En lugar de eso, ha sido contaminada mediante la difusión de un elemento especial como el fósforo o boro. En el primer caso, el silicio puro que es contaminado con fósforo termina con una cantidad extra de electrones libres que actúan como portadores de cargas eléctricas negativas dentro de la trayectoria en la capa de difusión. En el segundo caso, contaminar con boro remueve electrones del silicio, creando "hoyos" o portadores de cargas eléctricas positivas.

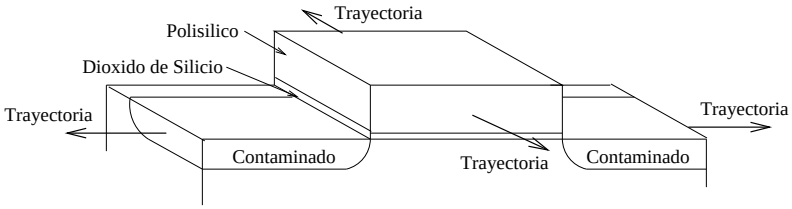


Figura 7.2: Un transistor formado por el cruce de dos trayectorias

Los procesos de fabricación como el enmascarado (*masking*) o grabado (*etching*) dejan una franja de dióxido de silicio bajo una franja de polisilicio. La habilidad de que el voltaje en la trayectoria de la capa de polisilicio controle el voltaje en la trayectoria de la capa de difusión genera un transistor básico, que se muestra esquemáticamente en la figura 7.3.

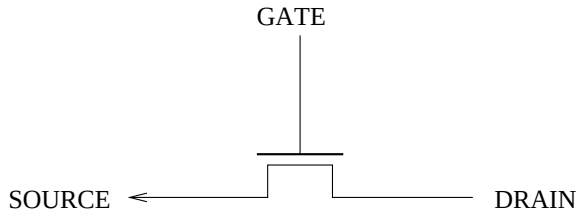


Figura 7.3: Esquema de un transistor

La cantidad de carga o voltaje en la terminal GATE (la trayectoria en el polisilicio) controla el voltaje en la trayectoria en la difusión como sigue: Si la carga en la terminal GATE excede un cierto nivel de umbral, los electrones se acumulan en la región no contaminada de la capa de difusión justo debajo la terminal GATE. Este exceso de electrones forma un paso conductor entre las terminales DRAIN y SOURCE. Consecuentemente, con una carga por debajo del umbral en GATE, virtualmente no hay un flujo de electrones de DRAIN a SOURCE, y por lo tanto, hay un voltaje alto. Cuando la carga en GATE aumenta sobre el umbral, hay un flujo de electrones, y el voltaje cae. El resultado es un interruptor básico, a partir del cual varios componentes lógicos pueden ser construidos.

Por ejemplo, considérese el caso de un inversor (véase el capítulo 2). Un inversor convierte una clase de señal en su opuesto. Considérese además que *alto* significa un voltaje por arriba del umbral, y *bajo* significa un voltaje por debajo del umbral. Vistas desde arriba, las varias trayectorias que forman un inversor pueden verse como se muestra en la figura 7.4. Las regiones de la capa de polisilicio se muestran en blanco, y la capa de difusión en regiones gris obscuro. Bajo de las regiones de polisilicio no hay contaminación. De esta forma, dos transistores forman el circuito inversor básico. La región de polisilicio a la izquierda se conecta a una fuente de voltaje constante (Vcc) diferente de la fuente de la trayectoria en la capa de difusión debajo de ella. El otro transistor opera como el descrito previamente. El voltaje que se conecta a su GATE constituye la entrada X. Cuando este voltaje se

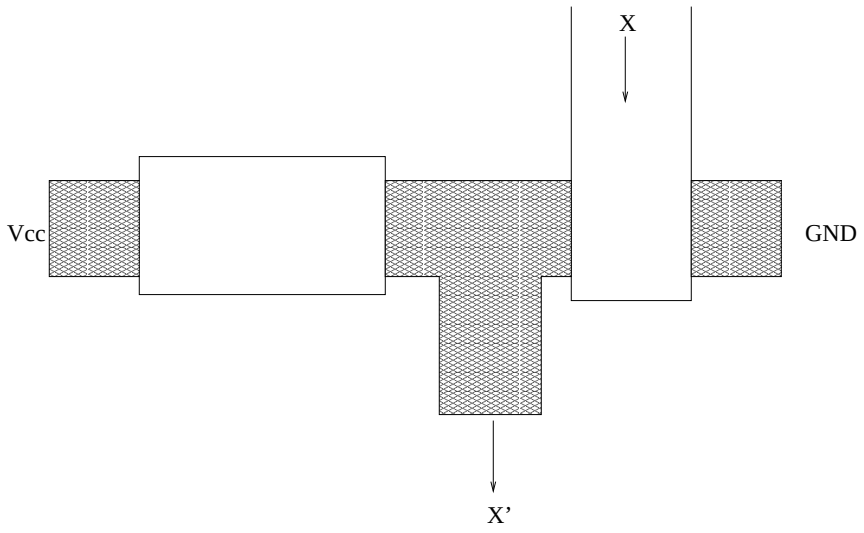


Figura 7.4: Un inversor en silicio

encuentra en alto, el voltaje en la trayectoria X' baja al nivel de GND (cero volts), volviéndose un bajo. Pero cuando el voltaje en la entrada X es bajo, no hay conexión entre la trayectoria X' y GND. En este punto, el transistor a la izquierda entra en acción. Ya que su GATE está siempre encendida (o en alto), hay siempre una corriente fluyendo de la fuente bajo GATE y hacia la trayectoria X'; por consiguiente el voltaje en la salida X' es un alto.

La geometría de transistores descrita hasta ahora puede ser extendida para dos o más transistores de la derecha en la figura 7.4. El patrón VLSI resultante es apenas un poco más complicado que el inversor (figura 7.5). En este circuito, la única forma de que la trayectoria $(XY)'$ puede estar en bajo es cuando ambas terminales GATE X *y* Y tienen un alto. De este modo, la trayectoria de la fuente se conecta directamente a GND a través de las tres terminales GATE. Consecuentemente, el voltaje en la trayectoria $(XY)'$, conectado también a GND, es bajo. Esto se simboliza lógicamente:

$$(XY)' = 0 \iff X = 1 \,\text{y}\, Y = 1$$

Esto no es más que una definición de la función lógica $(XY)'$; la trayectoria fue correctamente etiquetada de antemano, y el circuito opera como se ha especificado, es decir, como una compuerta **NAND**.

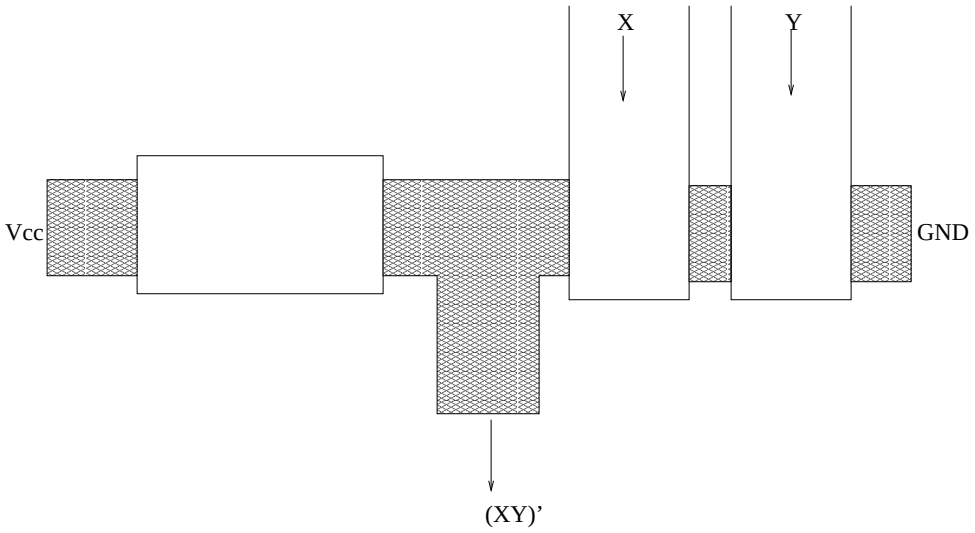


Figura 7.5: Una compuerta NAND en silicio

En el capítulo 1 se muestra que cualquier función lógica puede realizarse por una combinación de inversores y compuertas **NAND**. Así, sólo con las configuraciones de transistores que se han introducido hasta ahora, es posible fabricar todos los elementos de una computadora digital. Sin embargo, ya que compuertas **NOR** y otros componentes lógicos también se encuentran disponibles, una computadora en un circuito integrado de silicio puede hacer uso de estos otros elementos en cualquier momento en que el diseñador del circuito lo considere conveniente.

Hasta ahora, solo las capas conductoras inferiores (difusión y polisilicio) han sido discutidas. La capa conductora superior, la metálica, es la encargada de hacer las conexiones entre las trayectorias de salida de un componente lógico y las trayectorias de entrada de otros. Las conexiones se hacen a través de las capas aislantes mediante hoyos creados durante el proceso de fabricación. En términos esquemáticos, la sección lateral de tal conexión revela que la capa metálica no se encuentra estrictamente confinada a un nivel del circuito. Ciertamente, esto también es verdad para la capa de polisilicio y las capas de aislamiento que intervienen. En la figura 7.6, la salida de un dispositivo lógico se representa por la capa de difusión etiquetado A. Esta se conecta a través del contacto metálico a la capa de polisilicio etiquetada B, que representa el canal de entrada de otro dispositivo lógico.

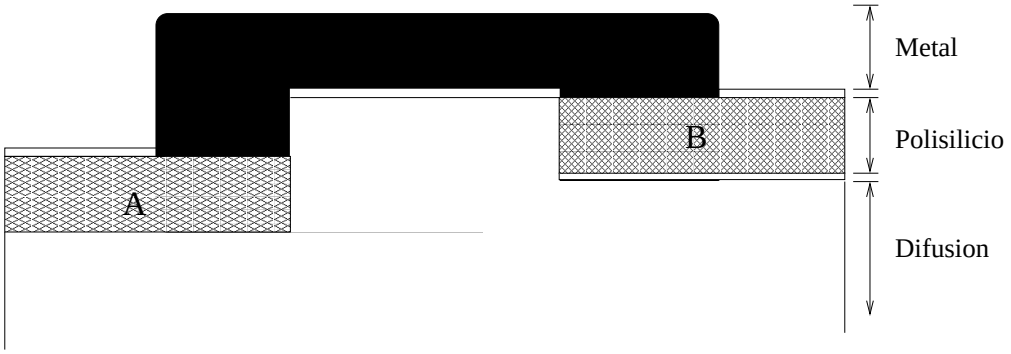


Figura 7.6: Sección lateral de una conexión

Las técnicas de fabricación de la lógica interna de un circuito VLSI tienen más en común con la impresión y litografía que con la electrónica tradicional. El primer paso consiste en que a partir de una solo cristal de silicio, creado bajo las condiciones de pureza más estrictas posibles, se cortan un número de "obleas". Cada oblea tiene varios centrímetros de ancho y largo, y un grosor de algunos milímetros. Las técnicas para VLSI pueden generar varios cientos de copias de un circuito de computadora sobre la superficie de una oblea. Cada circuito puede ser de apenas algunos milímetros de ancho, aun cuando contiene varios miles de componentes lógicos.

El segundo paso importante en la fabricación de circuitos es crear una capa de dióxido de silicio en la superficie de la oblea. Esto se sigue inmediatamente por la remoción selectiva de áreas de material aislante, mediante una pequeña máscara fotográfica impregnada en la oblea. La imagen fotográfica ha sido reducida de tamaño a partir de la escala en la que el diseñador del circuito trabaja, hasta el tamaño del microcircuito mismo. Una intensa radiación proyectada sobre el dióxido de silicio a través de la máscara rompe su estructura molecular en los puntos en que la máscara no es opaca. Tras esto, la oblea se lava con solventes poderosos que eliminan las regiones degradadas para dejar el patrón preciso de silicio expuesto.

El tercer paso involucra cubrir la oblea entera con una delgada cubierta de polisilicio o silicio policristalino. Otra máscara se usa para selectivamente remover ciertas porciones de esta cubierta.

Seguidamente de este paso, la superficie expuesta de silicio (la más baja) se contamina con varias impurezas para crear fuentes y sumideros de todos

los transistores, así como para crear trayectorias adicionales conductoras entre los componentes. Una segunda capa aislante de dióxido de silicio se aplica sobre la superficie, en un patrón que asegura la disconexión entre la capa metálica con la capa de polisilicio o de difusión. Evidentemente, si la capa metálica debe hacer contacto con alguna superficie, la capa de dióxido de silicio se encuentra ausente.

El último paso de la fabricación consiste en cubrir toda la oblea con una delgada cubierta de metal, como aluminio. Una máscara final se utiliza para remover todo el metal excepto en áreas donde se desea que haya trayectorias conductoras. Cuando la oblea VLSI es terminada, se corta en los circuitos individuales. Cada circuito es entonces pegado a un empaquetado plástico. Pequeños alambres conectan ciertas trayectorias metálicas de la orilla del circuito a las patas que sobresalen del encapsulado plástico. El producto final es un objeto familiar a muchas personas como la pequeña caja negra que reside en el corazón mismo de la computadora.

Bibliografía

[1] P.M.Chirlian. *Understanding Computers.* dilithium Press, 1978.

[2] G.W. Gorsline. *Computer Organization: Hardware/Software.* Prentice-Hall, 1986.

[3] V.C. Hamacher, Z.G. Vranesic y S.G. Zaky. *Organización de Computadoras.* McGraw-Hill, 1988.

[4] J.P. Hayes. *Diseño de Sistemas Digitales y Microprocesadores.* McGraw-Hill, 1988.

[5] R.R. Korfhage. *Discrete Computational Structures.* Academic, 1974.

[6] M.M. Mano. *Arquitectura de Computadoras.* Prentice-Hall, 1983.

[7] M.M. Mano. *Lógica Digital y Diseño de Computadoras.* Prentice-Hall, 1982.

[8] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[9] C.H. Roth Jr. *Fundamentals of Logic Design.* St. Paul eds., 1979.

[10] R.S. Sandige. *Data Concepts Using Standard Integrated Circuits.* McGraw-Hill, 1978.

[11] H.S. Stone. *Discrete Mathematical Structures and their Applications.* Science Research Associates, 1973.

[12] R.J. Tocci and L.P. Laskowski. *Microporcessors and Microcomputers.* Prentice-Hall, 1982.

[13] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.