Capítulo 5

FUNDAMENTOS DE LÓGICA DIGITAL CON DISEÑO VHDL

SEGUNDA EDICIÓN

Stephen Brown y Zvonko Vranesic
Departamento de Ingeniería Eléctrica y Computación
University of Toronto

Traducción
Lorena Peralta Rosales
Víctor Campos Olguín
Traductores profesionales

Revisión técnica
Jorge Valeriano Assem
Coordinador de la carrera de Ingeniería en Computación
Universidad Nacional Autónoma de México

Felipe Antonio Trujillo Fernández
Profesor del Departamento de Ingenierías
Universidad Iberoamericana, Campus Santa Fe

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA
MADRID • NUEVA YORK • SAN JUAN • SANTIAGO
AUCKLAND • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

capítulo

5

Representación de números y circuitos aritméticos

Objetivos del capítulo

En este capítulo se estudian los temas siguientes:

- Cómo se representan los números en las computadoras
- Los circuitos utilizados para realizar operaciones aritméticas
- Problemas de rendimiento en los circuitos grandes
- Cómo usar VHDL para especificar circuitos aritméticos

En este capítulo estudiaremos los circuitos lógicos que realizan operaciones aritméticas. Explicaremos cómo pueden sumarse, restarse y multiplicarse números. También mostraremos cómo escribir código de VHDL para describir los circuitos aritméticos, los cuales ofrecen una plataforma estupenda para ilustrar el poder y la versatilidad de ese lenguaje para especificar ensambles complejos de circuitos lógicos. Los conceptos relativos al diseño de los circuitos aritméticos se aplican con facilidad en muchos otros tipos de circuitos.

Antes de abordar el diseño de los circuitos aritméticos es preciso analizar cómo se representan los números en los sistemas digitales. En los capítulos anteriores explicamos las variables lógicas de un modo general, empleándolas para representar el estado de un interruptor o ciertas condiciones generales. Ahora las utilizaremos para representar números. Se necesita más de una variable para indicar un número, y cada variable corresponde a un dígito de éste.

5.1 REPRESENTACIÓN NUMÉRICA POSICIONAL

Cuando se estudian los números y las operaciones aritméticas es útil usar símbolos estándar. Por ende, para representar la suma ocupamos el símbolo más (+) y para la resta el símbolo menos (−). En capítulos anteriores empleamos + para representar la operación lógica OR y − para denotar la eliminación de un elemento de un conjunto. Ahora usaremos los mismos símbolos para dos propósitos diferentes, mas el significado de cada uno será sin duda claro a partir del contexto de la exposición. En los casos donde pueda haber cierta ambigüedad, explicaremos su significado.

5.1.1 ENTEROS SIN SIGNO

Los números más simples son los enteros. Empezaremos por considerar los enteros positivos y luego incluiremos los negativos. Los números positivos se llaman también *sin signo*, y los que pueden ser negativos se denominan *con signo*. La representación de los números que incluyen un punto en la base (números reales) se explica más adelante.

En el sistema decimal un número consta de dígitos que tienen 10 valores posibles, de 0 a 9, y cada dígito representa un múltiplo de una potencia de 10. Por ejemplo, el número 8547 representa $8 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$. No es común escribir las potencias de 10 con el número, ya que están implícitas en las posiciones de los dígitos. En general, un entero decimal se expresa por medio de una *n-tupla* que comprende *n* dígitos decimales

$$D = d_{n-1}d_{n-2}\cdots d_1d_0$$

que representa el valor

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

Esto se conoce como *representación numérica posicional*.

Puesto que los dígitos tienen 10 posibles valores y cada dígito se evalúa como una potencia de 10, se dice que los números decimales son números de *base* 10, o *raíz* 10. Los números decimales son conocidos, convenientes y fáciles de entender. Sin embargo, en los circuitos digitales no resulta práctico usar dígitos que pueden adquirir 10 valores. En los sistemas digitales se usa el

sistema binario, o de *base* 2, en el que los dígitos pueden ser 0 o 1. Cada dígito binario se llama *bit*. En el sistema numérico binario se emplea la misma representación numérica posicional, de modo que

$$B = b_{n-1}b_{n-2}\cdots b_1b_0$$

representa un entero que tiene el valor

$$\begin{aligned} V(B) &= b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{n-1} b_i \times 2^i \end{aligned} \qquad \textbf{[5.1]}$$

Por ejemplo, el número binario 1101 representa el valor

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Puesto que el patrón particular de un dígito tiene diferentes significados para distintas bases numéricas, las bases (o raíces) se indican con un subíndice cuando pueda haber alguna confusión. Por tanto, para indicar que 1101 es un número en base 2 se escribe $(1101)_2$. Al evaluar la expresión anterior para V se obtiene $V = 8 + 4 + 1 = 13$. Por tanto,

$$(1101)_2 = (13)_{10}$$

Nótese que el intervalo de enteros que puede representarse mediante un número binario depende del número de bits utilizados. Por ejemplo, con cuatro bits el número más grande es $(1111)_2 = (15)_{10}$. Un ejemplo de un número más grande es $(10110111)_2 = (183)_{10}$. En general, el uso de n bits permite representar enteros en el intervalo de 0 a $2^n - 1$.

En un número binario el bit del extremo derecho se denomina *bit menos significativo* (LSB, *least-significant bit*). El bit del extremo izquierdo en un entero sin signo, que tiene asociada la potencia más alta de 2, se llama *bit más significativo* (MSB, *most-significant bit*). En los sistemas digitales es útil considerar varios bits juntos como un grupo. Un grupo de cuatro bits se llama *nibble* y uno de ocho bits se denomina *byte*.

5.1.2 CONVERSIÓN ENTRE SISTEMAS DECIMAL Y BINARIO

Un número binario se convierte en un número decimal simplemente aplicando la ecuación 5.1 y evaluándolo con aritmética decimal. La conversión de un número decimal en uno binario no es tan directa. Puede llevarse a cabo mediante la división sucesiva entre 2 del número decimal, del modo siguiente. Supóngase que un número decimal $D = d_{k-1}\cdots d_1d_0$, con un valor V, se convertirá en un número binario $B = b_{n-1}\cdots b_2b_1b_0$. Por tanto,

$$V = b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0$$

Si V se divide entre 2 el resultado es

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

El cociente de esta división entera es $b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2 + b_1$, y el residuo es b_0. Si el residuo es 0, entonces $b_0 = 0$; si es 1, entonces $b_0 = 1$. Obsérvese que el cociente es justo otro número binario, que comprende $n - 1$ bits, en lugar de n bits. Al dividir este número entre 2 queda el residuo b_1. El nuevo cociente es

$$b_{n-1} \times 2^{n-3} + \cdots + b_2$$

Si se continúa el proceso de dividir el nuevo cociente entre 2 y se determina un bit en cada paso se producirán todos los bits del número binario. El proceso continúa hasta que el cociente es 0. En la figura 5.1 se ilustra el proceso de conversión mediante el ejemplo $(857)_{10} = (1101011001)_2$. Nótese que primero se genera el bit menos significativo (LSB) y al final queda el más significativo (MSB).

5.1.3 REPRESENTACIONES OCTAL Y HEXADECIMAL

La representación numérica posicional puede usarse para cualquier base. Si ésta es r, entonces el número

$$K = k_{n-1}k_{n-2} \cdots k_1k_0$$

tiene el valor

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

En el texto, el interés se limita a las bases más prácticas. Usaremos números decimales porque son los que utiliza la gente, y emplearemos números binarios porque son los que ocupan las computadoras. Además, son útiles otras dos bases: 8 y 16. Los números representados con la

Convertir $(857)_{10}$

			Residuo	
$857 \div 2$	=	428	1	LSB
$428 \div 2$	=	214	0	
$214 \div 2$	=	107	0	
$107 \div 2$	=	53	1	
$53 \div 2$	=	26	1	
$26 \div 2$	=	13	0	
$13 \div 2$	=	6	1	
$6 \div 2$	=	3	0	
$3 \div 2$	=	1	1	
$1 \div 2$	=	0	1	MSB

El resultado es $(1101011001)_2$

Figura 5.1 Conversión de decimal a binario.

base 8 se llaman números *octales* y los de base 16 se denominan *hexadecimales*. En la representación octal, los valores de los dígitos van de 0 a 7; en la hexadecimal (que se abrevia *hexa*), cada dígito puede tener uno de 16 valores. Los primeros 10 se denotan igual que en el sistema decimal: de 0 a 9. Los dígitos que corresponden a los valores decimales 10, 11, 12, 13, 14 y 15 se denotan mediante las letras A, B, C, D, E y F. En la tabla 5.1 se presentan los primeros 18 enteros en estos sistemas numéricos.

Tabla 5.1 Números en diferentes sistemas.

Decimal	Binario	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

El sistema numérico dominante en las computadoras es el binario. Los sistemas octal y hexadecimal se usan porque brindan una notación abreviada para los números binarios. Un dígito octal representa tres bits. Por tanto, un número binario se convierte en un número octal tomando grupos de tres bits, desde el menos significativo, y sustituyéndolos con el correspondiente dígito octal. Por ejemplo, 101011010111 se convierte en

$$\underbrace{1\,0\,1}_{5}\qquad\underbrace{0\,1\,1}_{3}\qquad\underbrace{0\,1\,0}_{2}\qquad\underbrace{1\,1\,1}_{7}$$

lo que significa que $(101011010111)_2 = (5327)_8$. Si el número de bits no es un múltiplo de tres, entonces se agrega 0 a la izquierda del bit más significativo. Por ejemplo, $(10111011)_2 = (273)_8$ porque

$$\underbrace{0\,1\,0}_{2}\qquad\underbrace{1\,1\,1}_{7}\qquad\underbrace{0\,1\,1}_{3}$$

La conversión de octal a binario es así de directa; cada dígito octal simplemente se sustituye por tres bits que denotan el mismo valor.

De manera similar, un dígito hexadecimal se representa con cuatro bits. Por ejemplo, un número de 16 bits se representa con cuatro dígitos hexa, como en

$$(1010111100100101)_2 = (AF25)_{16}$$

porque

$$\underbrace{1\,0\,1\,0}_{A} \qquad \underbrace{1\,1\,1\,1}_{F} \qquad \underbrace{0\,0\,1\,0}_{2} \qquad \underbrace{0\,1\,0\,1}_{5}$$

Si el número de bits no es múltiplo de cuatro, se agregan ceros a la izquierda del bit más significativo. Por ejemplo, $(1101101000)_2 = (368)_{16}$, porque

$$\underbrace{0\,0\,1\,1}_{3} \qquad \underbrace{0\,1\,1\,0}_{6} \qquad \underbrace{1\,0\,0\,0}_{8}$$

La conversión de hexadecimal a binario involucra la sustitución directa de cada dígito hexa por cuatro bits que denotan el mismo valor.

Los números binarios usados en las computadoras modernas con frecuencia tienen 32 o 64 bits. Escritos como *n*-tuplas binarias (a veces llamadas vectores bit), tales números son complicados para que la gente los manipule. Es mucho más simple lidiar con ellos en la forma de números de 8 o 16 dígitos. Puesto que las operaciones aritméticas en un sistema digital suelen comprender números binarios, nos centraremos en los circuitos que los usan. En ocasiones emplearemos la representación hexadecimal como una cómoda descripción abreviada.

Ya expusimos los números más simples: los enteros sin signo. Es necesario ser capaz de trabajar con varios tipos de números. Más adelante, en este capítulo, se tratará la representación de los números con signo, los números con punto fijo y con punto flotante. Pero antes examinaremos algunos circuitos simples que operan sobre los números para dar al lector una idea de los circuitos digitales que realizan operaciones aritméticas y alentarlo así para la discusión posterior.

5.2 SUMA DE NÚMEROS SIN SIGNO

La suma binaria se realiza igual que la decimal, excepto que los valores de los dígitos individuales sólo pueden ser 0 o 1. La suma de dos números de un bit conlleva cuatro posibles combinaciones, como se indica en la figura 5.2*a*. Se necesitan dos bits para representar el resultado de la suma. El bit del extremo derecho se llama *suma* (*sum*), *s*. El del extremo izquierdo, que se produce como acarreo cuando los dos bits que se suman son iguales a 1, se denomina *acarreo* (*carry*), *c*. La operación suma se define en forma de una tabla de verdad en el inciso *b*) de la figura. El bit suma *s* es la función XOR, expuesta en la sección 3.9.1. El acarreo *c* es la función AND de las entradas *x* y *y*. En la figura 5.2*c* se muestra una realización del circuito de estas funciones. Este circuito, que implementa la suma sólo de dos bits, se llama medio sumador (HA, *half-adder*).

Un caso más interesante es cuando se involucran números más grandes que tienen varios bits. Aun así es necesario sumar cada par de bits, pero, para cada posición de bit *i*, la operación suma puede incluir un *acarreo* desde la posición de bit $i - 1$.

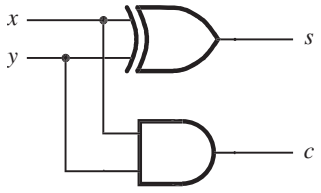
x	0	0	1	1
$+y$	+0	+1	+0	+1
$c\ s$	0 0	0 1	0 1	1 0

Acarreo (*carry*, *c*) — Suma (*sum*, *s*)

a) Los cuatro casos posibles

x	y	Acarreo c	Suma s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

b) Tabla de verdad



c) Circuito

d) Símbolo gráfico

Figura 5.2 Medio sumador.

En la figura 5.3 se presenta un ejemplo de la operación suma. Los dos operandos son $X = (01111)_2 = (15)_{10}$ y $Y = (01010)_2 = (10)_{10}$. Nótese que se usan cinco bits para representar X y Y. Con cinco bits es posible representar enteros que estén en el intervalo de 0 a 31; por ende, la suma $S = X + Y = (25)_{10}$ también puede denotarse como un entero de cinco bits. Obsérvese también la etiqueta de cada bit, tal que $X = x_4x_3x_2x_1x_0$ y $Y = y_4y_3y_2y_1y_0$. En la figura se muestran

$X = x_4x_3x_2x_1x_0$	0 1 1 1 1	$(15)_{10}$
$+Y = y_4y_3y_2y_1y_0$	0 1 0 1 0	$(10)_{10}$
	1 1 1 0 ← Acarreos generados	
$S = s_4s_3s_2s_1s_0$	1 1 0 0 1	$(25)_{10}$

Figura 5.3 Ejemplo de suma.

los acarreos generados durante el proceso de suma. Por ejemplo, un acarreo de 0 se genera cuando se suman x_0 y y_0; un acarreo de 1 se produce cuando se suman x_1 y y_1, etcétera.

En los capítulos 2 y 4 diseñamos circuitos lógicos especificando primero sus comportamientos en la forma de tabla de verdad. Este enfoque no es práctico al diseñar un circuito sumador que puede sumar los números de cinco bits de la figura 5.3. La tabla de verdad requerida tendría 10 variables de entrada, cinco para cada número X y Y. ¡Tendría $2^{10} = 1024$ filas! Un enfoque mejor consiste en considerar la suma de cada par de bits, x_i y y_i por separado.

Para la posición de bit 0 no hay acarreo y, por tanto, la suma es la misma que para la figura 5.2. Para cada otra posición de bit i, la suma comprende los bits x_i y y_i, y un acarreo en c_i. Las funciones suma y acarreo de las variables x_i, y_i y c_i se especifican en la tabla de verdad de la figura 5.4*a*. El bit suma, s_i, es la suma módulo 2 de x_i, y_i y c_i. El acarreo, c_{i+1}, es igual a 1 si la suma de x_i, y_i y c_i es igual a 2 o a 3. Los mapas de Karnaugh para estas funciones se muestran en el inciso *b*) de la figura. Para la función acarreo la realización óptima en suma de productos es

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Para la función s_i, una realización en suma de productos es

$$s_i = \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + \overline{x}_i \overline{y}_i c_i + x_i y_i c_i$$

Una forma más atractiva de implementar esta función es con compuertas XOR, como se explica a continuación.

Uso de compuertas XOR

La función XOR de dos variables se define como $x_1 \oplus x_2 = \overline{x}_1 x_2 + x_1 \overline{x}_2$. La expresión anterior para el bit suma puede manipularse del modo siguiente en una forma que sólo use operaciones XOR

$$\begin{aligned} s_i &= (\overline{x}_i y_i + x_i \overline{y}_i)\overline{c}_i + (\overline{x}_i \overline{y}_i + x_i y_i) c_i \\ &= (x_i \oplus y_i)\overline{c}_i + \overline{(x_i \oplus y_i)} c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

La operación XOR es asociativa; por tanto, puede escribirse

$$s_i = x_i \oplus y_i \oplus c_i$$

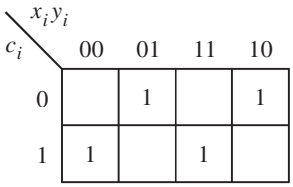
En consecuencia, para realizar s_i es posible utilizar una sola compuerta XOR de tres entradas.

La compuerta XOR genera como salida una suma módulo 2 de sus entradas. La salida es igual a 1 si un número impar de entradas tiene el valor 1; de otro modo es igual a 0. Por ello, la XOR a veces se llama *función impar*. Obsérvese que la XOR no tiene minitérminos que puedan combinarse en un término producto más grande, como es evidente a partir del patrón de marcas de la función s_i en el mapa de la figura 5.4*b*. En la figura 5.4*c* se proporciona el circuito lógico que implementa la tabla de verdad de la figura 5.4*a*. Este circuito se conoce como *sumador completo* (FA, *full-adder*).

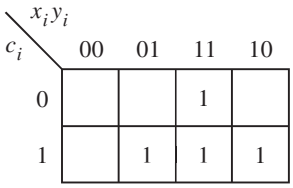
Otra característica interesante de las compuertas XOR es que puede considerarse que una compuerta XOR de dos entradas utiliza una entrada como señal de control que determina si a través de la compuerta pasará como valor de salida el valor verdadero o complementado de la otra entrada. Esto es claro a partir de la definición de XOR, donde $x_i \oplus y_i = \overline{x}y + x\overline{y}$. Sea x la entrada de control. Entonces, si $x = 0$, la salida será igual al valor de y. Pero si $x = 1$, la salida

c_i	x_i	y_i	c_{i+1}	s_i
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

a) Tabla de verdad

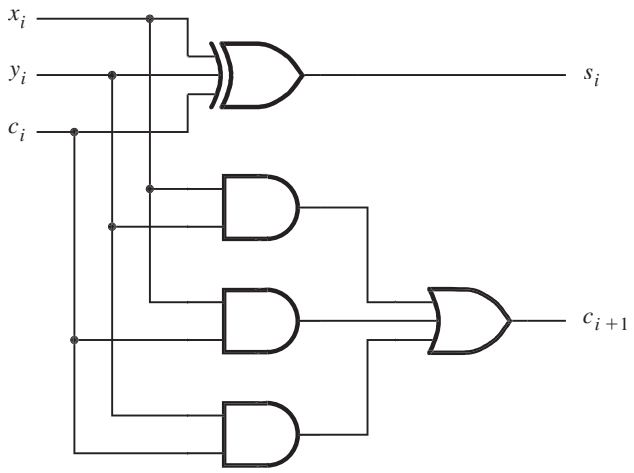


$s_i = x_i \oplus y_i \oplus c_i$



$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

b) Mapas de Karnaugh



c) Circuito

Figura 5.4 Sumador completo.

será igual al complemento de y. En la deducción anterior usamos manipulación algebraica para derivar $s_i = (x_i \oplus y_i) \oplus c_i$. Pudimos haber obtenido la misma expresión de inmediato haciendo la observación siguiente. En la mitad superior de la tabla de verdad de la figura 5.4*a*, c_i es igual a 0, y la función suma s_i es la XOR de x_i y y_i. En la mitad inferior de la tabla, c_i es igual a 1, mientras que s_i es la versión complementada de su mitad superior. Esta observación conduce directamente a nuestra expresión que utiliza dos operaciones XOR de dos entradas. En la

sección 5.3.3 encontraremos un importante ejemplo del uso de las compuertas XOR para pasar señales verdaderas o complementadas bajo el control de otra señal.

En la explicación precedente vimos el complemento de la operación XOR, que se denotó como $\overline{x \oplus y}$. Esta operación se usa de manera tan común que se le ha dado un nombre distintivo: XNOR. Con frecuencia se utiliza el símbolo especial $\odot$, para denotar la operación XNOR:

$$x \odot y = \overline{x \oplus y}$$

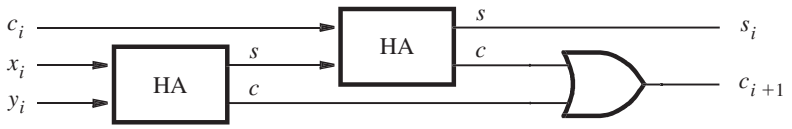
La operación XNOR a veces también se conoce como *operación coincidencia* porque produce la salida de 1 cuando sus entradas coinciden en valor; es decir, ambas son 0 o ambas son 1.

5.2.1 SUMADOR COMPLETO DESCOMPUESTO

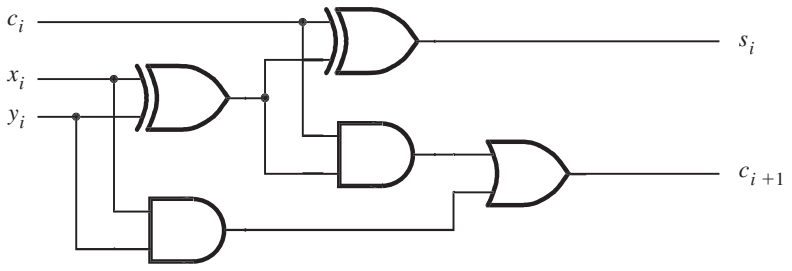
En vista de los nombres utilizados para los circuitos, cabe esperar que un sumador completo se construya con medios sumadores. Esto puede lograrse creando un circuito multinivel del tipo expuesto en la sección 4.6.2. El circuito se presenta en la figura 5.5. Emplea dos medios sumadores para formar un sumador completo. Se deja como ejercicio al lector comprobar su exactitud.

5.2.2 SUMADOR CON ACARREO EN CASCADA

Para realizar la adición a mano, se comienza desde el dígito menos significativo y se agregan pares de dígitos; luego se continúa hacia el dígito más significativo. Si en la posición *i* se produce



a) Diagrama de bloques



b) Diagrama detallado

Figura 5.5 Implementación descompuesta del circuito sumador completo.

un acarreo, éste se suma a los operandos en la posición $i + 1$. El mismo ordenamiento se usa en un circuito lógico que realice sumas. Para cada posición de bit puede utilizarse un circuito sumador completo, conectado como se muestra en la figura 5.6. Nótese que para ser consistentes con la forma habitual de escribir números, la posición del bit menos significativo está a la derecha. Los acarreos producidos por los sumadores completos se propagan a la izquierda.

Cuando los operandos X y Y se aplican como entradas al sumador, se requiere cierto tiempo antes de que la salida suma (*sum*), S, sea válida. Cada sumador completo introduce cierto retraso antes de que sus salidas s_i y c_{i+1} sean válidas. Denotemos tal retraso como Δt. Por tanto, el acarreo de la primera etapa, c_1, llega a la segunda etapa Δt después de la aplicación de las entradas x_0 y y_0. El acarreo de la segunda etapa, c_2, llega a la tercera con un retraso $2\Delta t$, etc. La señal c_{n-1} es válida después de un retraso de $(n - 1)\Delta t$, lo que significa que la suma completa está disponible después de un retraso de $n\Delta t$. Por la forma en que las señales "se propagan" a través de las etapas del sumador completo, el circuito de la figura 5.6 se denomina *sumador con acarreo en cascada*.

El retraso en que se incurre para producir la suma final y el acarreo en un sumador de acarreo en cascada depende del tamaño de los números. Cuando se usan números de 32 o 64 bits, este retraso puede llegar a ser inaceptablemente alto. Puesto que el circuito en cada sumador completo deja poco espacio para una reducción drástica del retraso, quizá sea necesario buscar diferentes estructuras para implementar sumadores de n bits. En la sección 5.4 expondremos una técnica para construir sumadores de alta velocidad.

Hasta el momento hemos trabajado sólo con enteros sin signo. La suma de tales números no requiere un acarreo para la etapa 0. En la figura 5.6 incluimos c_0 en el diagrama, de modo que el sumador con acarreo en cascada también sirva para restar números, como veremos en la sección 5.3.

5.2.3 EJEMPLO DE DISEÑO

Supóngase que se necesita un circuito que multiplique por 3 un número sin signo de ocho bits. Sea $A = a_7a_6 \ldots a_1a_0$ el número y $P = p_9p_8 \ldots p_1p_0$ el producto $P = 3A$. Nótese que se necesitan 10 bits para representar el producto.

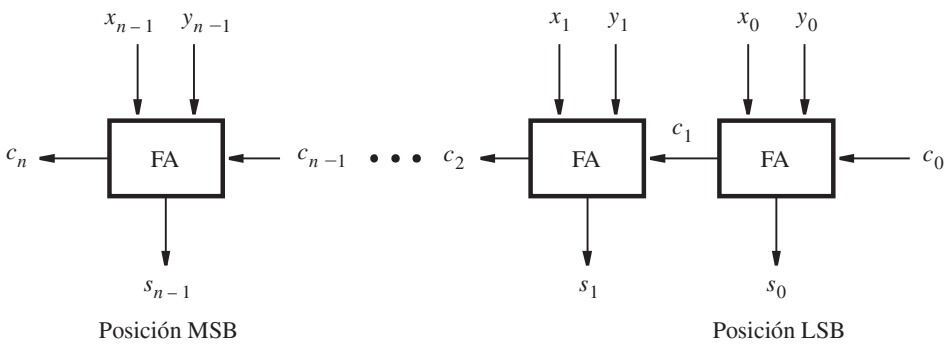


Figura 5.6 Sumador de n-bits con acarreo en cascada.

Un enfoque simple para diseñar ese circuito consiste en usar dos sumadores con acarreo en cascada para sumar tres copias del número A, como se ilustra en la figura 5.7*a*. El símbolo que denota a cada sumador es uno utilizado comúnmente para los sumadores. Las letras x_i, y_i, s_i y c_i indican el significado de las entradas y las salidas de acuerdo con la figura 5.6. El primer sumador produce $A + A = 2A$. Su resultado se representa como ocho bits suma y el acarreo proveniente del bit más significativo. El segundo sumador produce $2A + A = 3A$. Tiene que ser un sumador de nueve bits para poder manejar los nueve bits de $2A$, que el primer sumador genera. Puesto que las entradas y_i deben dirigirse sólo por los ocho bits de A, la novena entrada y_8 se conecta a una constante 0.

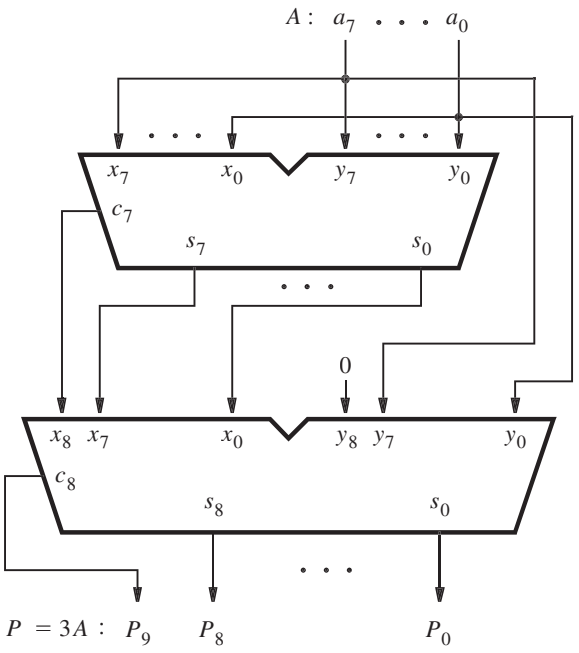
Este método es directo, pero poco eficiente. Puesto que $3A = 2A + A$, se observa que $2A$ puede generarse corriendo los bits de A una posición de un bit a la izquierda, lo que produce el patrón de bits $a_7a_6a_5a_4a_3a_2a_1a_0$0. De acuerdo con la ecuación 5.1, este patrón es igual a $2A$. Entonces basta un solo sumador con acarreo en cascada para implementar $3A$, como se muestra en la figura 5.7*b*. Éste es en esencia el mismo circuito que el segundo sumador del inciso *a*) de la figura. Nótese que la entrada x_0 se conecta a una constante 0. Nótese también que en el segundo sumador del inciso *a*), el valor de x_0 siempre es 0, aun cuando lo dirija el bit menos significativo, s_0, de la suma del primer sumador. Como $x_0 = y_0 = a_0$ en el primer sumador, el bit suma s_0 será 0, ya sea que a_0 sea 0 o 1.

5.3 NÚMEROS CON SIGNO

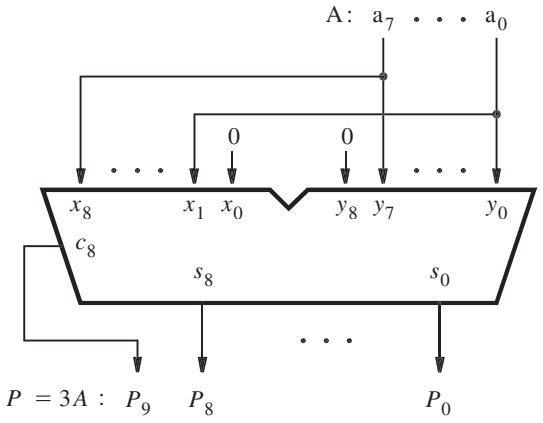
En el sistema decimal el signo de un número se indica mediante los símbolos + o − a la izquierda del dígito más significativo. En el sistema binario el *signo* de un número se denota por el bit del extremo izquierdo. Para un número positivo ese bit es igual a 0, y para un número negativo es igual a 1. Por tanto, en los números con signo el bit del extremo izquierdo representa el signo y los restantes $n - 1$ bits representan la magnitud, como se ilustra en la figura 5.8. Es importante notar la diferencia en la ubicación del bit más significativo (MSB). En los números sin signo todos los bits representan la magnitud de un número; por ende, los n bits son *significativos* al definir la magnitud. En consecuencia, el MSB es el bit más a la izquierda, b_{n-1}. En los números con signo existen $n - 1$ bits significativos, y el MSB se halla en la posición del bit b_{n-2}.

5.3.1 NÚMEROS NEGATIVOS

Los números positivos se indican con la representación numérica posicional, como explicamos en la sección anterior. Los números negativos pueden representarse de tres formas: signo y magnitud, complemento a 1 y complemento a 2.



a) Enfoque poco eficiente



b) Diseño eficiente

Figura 5.7 Circuito que multiplica por 3 un número sin signo de ocho bits.

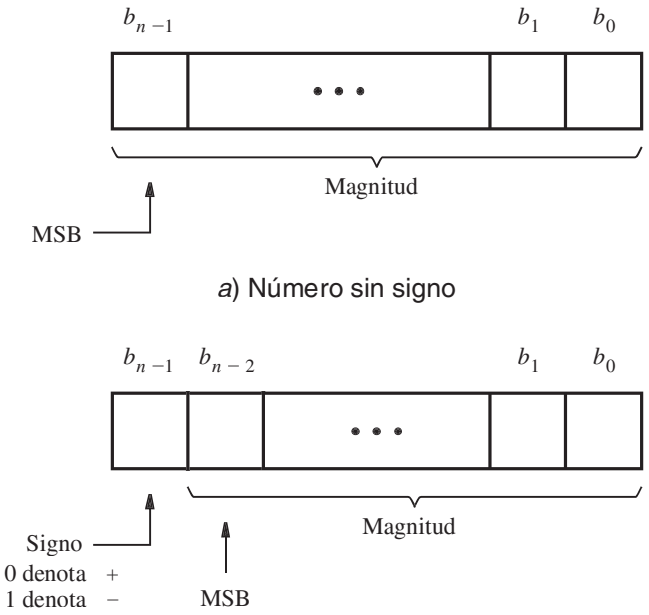


Figura 5.8 Formatos para la representación de enteros.

Representación signo y magnitud

En la representación decimal, la magnitud de los números positivos y negativos se expresa igual. El signo distingue un número como positivo o negativo. Este esquema se llama representación numérica *signo y magnitud*. El mismo esquema puede usarse con números binarios, caso en el que el bit del signo es 0 o 1 para los números positivos o negativos, respectivamente. Por ejemplo, si utilizamos números de cuatro bits, entonces $+5 = 0101$ y $-5 = 1101$. Por su similitud con los números decimales de signo y magnitud, esta representación es fácil de entender. Sin embargo, como muy pronto veremos, no está bien adaptada para usarla en las computadoras. Las representaciones más adecuadas se basan en sistemas de complementación, como explicamos enseguida.

Representación en complemento a 1

En un sistema numérico complementario, los números negativos se definen de acuerdo con una operación de sustracción que implica números positivos. Consideraremos dos esquemas para números binarios: el complemento a 1 y el complemento a 2. En el esquema de *complemento a 1*, un número negativo de n bits, K, se obtiene restando su equivalente número positivo, P, de $2^n - 1$; es decir, $K = (2^n - 1) - P$. Por ejemplo, si $n = 4$, entonces $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$. Si $+5$ se convierte en negativo se obtiene $-5 = 1111 - 0101 = 1010$.

De manera similar, $+3 = 0011$ y $-3 = 1111 - 0011 = 1100$. Es claro que el complemento a 1 puede obtenerse simplemente complementando cada bit del número, incluido el bit del signo. Si bien los números en complemento a 1 son fáciles de derivar, tienen ciertos inconvenientes cuando se emplean en operaciones aritméticas, como veremos en la sección siguiente.

Representación en complemento a 2

En el esquema de complemento a 2, un número negativo, K, se obtiene mediante la sustracción de su equivalente número positivo, P, de 2^n, de modo que $K = 2^n - P$. Si usamos nuestro ejemplo de cuatro bits, $-5 = 10000 - 0101 = 1011$ y $-3 = 10000 - 0011 = 1101$. Encontrar el complemento a 2 de esta forma requiere una sustracción que implica "pedir prestado". Sin embargo, puede observarse que si K_1 es el complemento a 1 de P y K_2 es el complemento a 2 de P, entonces

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

Se sigue que $K_2 = K_1 + 1$. Por tanto, una forma más simple de encontrar el complemento a 2 de un número consiste en sumar 1 a su complemento a 1, porque encontrar un complemento a 1 es sencillo. Es así como se obtienen los números en complemento a 2 en los circuitos lógicos que realizan operaciones aritméticas.

El lector habrá de desarrollar la habilidad para encontrar rápidamente números en complemento a 2. Hay una regla simple que sirve para tal propósito.

Regla para encontrar complementos a 2 Dado un número con signo, $B = b_{n-1}b_{n-2}, \ldots b_1b_0$, su complemento a 2, $K = k_{n-1}k_{n-2}\cdots k_1k_0$, se encuentra examinando los bits de B de derecha a izquierda y tomando la acción siguiente: se copian todos los bits de B que sean 0 y el primer bit que sea 1; luego simplemente se complementa el resto de los bits.

Por ejemplo, si $B = 0110$, entonces se copia $k_0 = b_0 = 0$ y $k_1 = b_1 = 1$ y se complementa el resto de modo que $k_2 = \overline{b}_2 = 0$ y $k_3 = \overline{b}_3 = 1$. Por tanto, $K = 1010$. Como otro ejemplo, si $B = 10110100$, entonces $K = 01001100$. La comprobación de esta regla se deja como ejercicio para el lector.

En la tabla 5.2 se ilustra la interpretación de los 16 patrones de cuatro bits en las tres representaciones de números con signo que hemos considerado. Nótese que tanto para la representación signo y magnitud como para la representación en complemento a 1 existen dos patrones que representan el valor cero. Para el complemento a 2 hay sólo uno de tales patrones. Además, obsérvese que el intervalo de los números que pueden representarse con cuatro bits en forma de complemento a 2 va de -8 a $+7$, mientras que en las otras dos representaciones va de -7 a $+7$.

Si se utiliza la representación en complemento a 2, un número $B = b_{n-1}b_{n-2}\cdots b_1b_0$ de n bits representa el valor

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad \textbf{[5.2]}$$

Por ende, el número negativo más grande, 100 . . . 00, tiene el valor -2^{n-1}. El número positivo más grande, 011 . . . 11, tiene el valor $2^{n-1} - 1$.

Tabla 5.2 Interpretación de enteros con signo de cuatro bits.

$b_3b_2b_1b_0$	Signo y magnitud	Complemento a 1	Complemento a 2
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

5.3.2 SUMA Y RESTA

Para valorar lo adecuado de las diferentes representaciones numéricas es preciso investigar sus usos en las operaciones, en particular la suma y la resta. Es posible ilustrar los aspectos buenos y malos de cada representación si consideramos números muy pequeños. Emplearemos números de cuatro bits, que constan del bit del signo y tres bits significativos. Por tanto, los números han de ser lo suficientemente pequeños para que la magnitud de su suma pueda expresarse en tres bits, lo que significa que la suma no puede superar el valor 7.

La suma de números positivos es igual en las tres representaciones numéricas. De hecho, es igual que la suma de números sin signo expuesta en la sección 5.2. Pero hay diferencias significativas cuando se trata de los números negativos. Las dificultades que surgen se evidencian si se consideran operandos con diferentes combinaciones de signos.

Suma en signo y magnitud

Si ambos operandos tienen el mismo signo, entonces la suma de los números con signo y magnitud es simple. Se suman las magnitudes y se da a la suma resultante el signo de los operandos. Sin embargo, la tarea se vuelve más complicada si los operandos tienen signos opuestos. Entonces es preciso sustraer el número más pequeño del más grande. Esto significa que también se necesitan circuitos lógicos que comparen y resten números. Dentro de poco veremos que es posible realizar restas sin la necesidad de estos circuitos. Por ello, la representación de signo y magnitud no se usa en las computadoras.

Suma en complemento a 1

Una ventaja obvia de la representación en complemento a 1 es que un número negativo se genera mediante la simple complementación de todos los bits del correspondiente número positivo. En la figura 5.9 se muestra lo que ocurre cuando se suman dos números. Hay que considerar cuatro casos en términos de diferentes combinaciones de signos. Como se observa en la mitad superior de la figura, el cálculo de $5 + 2 = 7$ y de $(-5) + 2 = (-3)$ es directo; una simple suma de los operandos produce el resultado correcto. Pero éste no es el caso con las otras dos posibilidades. El cálculo de $5 + (-2) = 3$ produce el vector bit 10010. Puesto que estamos usando números de cuatro bits, hay un acarreo de la posición del bit del signo. Además, los cuatro bits del resultado representan el número 2 en vez de 3, lo que es un resultado erróneo. Curiosamente, si se toma el acarreo de la posición del bit del signo y se suma al resultado en la posición del bit menos significativo, el nuevo resultado es la suma correcta de 3. Esta corrección se indica en gris oscuro en la figura. Una situación similar se presenta cuando se suman $(-5) + (-2) = (-7)$. Después de la suma inicial, el resultado es equivocado porque los cuatro bits de la suma son 0111, que representa $+7$ en vez de -7. Pero, de nuevo, existe un acarreo de la posición del bit del signo, que puede usarse para corregir el resultado sumándolo en la posición LSB, como se muestra en la figura 5.9.

La conclusión a partir de estos ejemplos es que la suma de números en complemento a 1 puede o no ser simple. En ciertos casos se requiere una corrección, lo que significa que debe realizarse una suma adicional. En consecuencia, el tiempo necesario para sumar dos números en complemento a 1 puede ser el doble del requerido para sumar dos números sin signo.

Suma en complemento a 2

Considérense las mismas combinaciones de números usados en el ejemplo de complemento a 1. En la figura 5.10 se indica cómo se realiza la suma con números en complemento a 2. La suma de $5 + 2 = 7$ y de $(-5) + 2 = (-3)$ es directa. El cálculo de $5 + (-2) = 3$ genera los cuatro bits correctos del resultado, 0011. Hay un acarreo de la posición del bit del signo, que simplemente puede ignorarse. El cuarto caso es $(-5) + (-2) = (-7)$. De nuevo, los cuatro bits del resultado, 1001, proporcionan la suma correcta (-7). En este caso también puede ignorarse el acarreo de la posición del bit del signo.

(+5)	0 1 0 1	(−5)	1 0 1 0
+(+2)	+ 0 0 1 0	+(+2)	+ 0 0 1 0
(+7)	0 1 1 1	(−3)	1 1 0 0
(+5)	0 1 0 1	(−5)	1 0 1 0
+ (−2)	+ 1 1 0 1	+ (−2)	+ 1 1 0 1
(+3)	1 0 0 1 0	(−7)	1 0 1 1 1
	→ 1		→ 1
	0 0 1 1		1 0 0 0

Figura 5.9 Ejemplos de suma en complemento a 1.

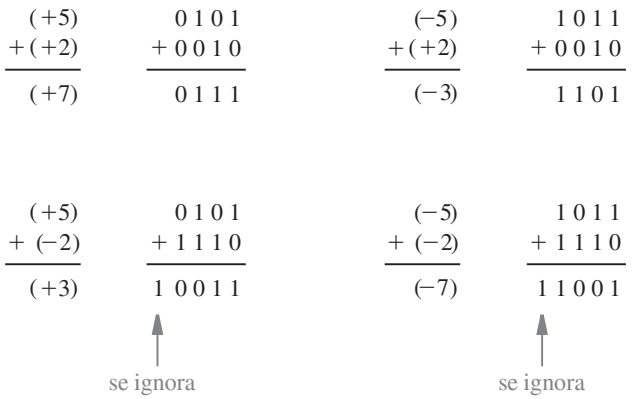


Figura 5.10 Ejemplos de suma en complemento a 2.

Como se ilustra con estos ejemplos, la suma de números en complemento a 2 es muy simple. Cuando los números se suman, el resultado siempre es correcto. Si hay un acarreo de la posición del bit del signo, simplemente se ignora. En consecuencia, el proceso de suma es el mismo, independientemente del signo de los operandos. Se puede realizar mediante un circuito sumador como el presentado en la figura 5.6. Por tanto, la notación en complemento a 2 es muy recomendable para la implementación de operaciones de suma. Ahora consideraremos su uso en operaciones de sustracción.

Resta en complemento a 2

La forma más sencilla de realizar restas es negar el sustraendo y sumarlo al minuendo. Esto se hace encontrando el complemento a 2 del sustraendo y luego realizando la suma. En la figura 5.11 se ilustra el proceso. La operación $5 - (+2) = 3$ supone encontrar el complemento a 2 de $+2$, que es 1110. Cuando se suma este número a 0101 el resultado es $0011 = (+3)$ y ocurre un acarreo de la posición del bit del signo, que se ignora. Una situación similar surge para $(-5) - (+2) = (-7)$. En los dos casos restantes no hay acarreo y el resultado es correcto.

Como un auxiliar gráfico para ver los ejemplos de suma y resta, en las figuras 5.10 y 5.11 colocamos todos los posibles patrones de cuatro bits en un círculo de módulo 16 dado en la figura 5.12. Si esos patrones de bit representasen enteros sin signo, serían los números de 0 a 15. Si representasen enteros en complemento a 2, entonces variarían de -8 a $+7$, como se muestra. La operación suma se realiza avanzando en el sentido de las manecillas del reloj para la magnitud del número que se va a sumar. Por ejemplo, $-5 + 2$ se determina comenzando en 1011 $(= -5)$ y dando dos pasos en el sentido de las manecillas del reloj, lo que produce el resultado 1101 $(= -3)$. La resta se realiza avanzando en contrasentido a las manecillas del reloj. Por ejemplo, $-5 - (+2)$ se determina empezando en 1011 y moviéndose dos pasos contra las manecillas del reloj, lo que produce 1001 $(= -7)$.

La conclusión clave de esta sección es que la operación resta puede realizarse como la operación suma, usando el complemento a 2 del sustraendo, sin importar los signos de los dos

```
 (+5)      0 1 0 1          0 1 0 1
−(+2)    − 0 0 1 0        + 1 1 1 0
-----    ---------        ---------
 (+3)                      1 0 0 1 1
                                ↑
                           se ignora

 (–5)      1 0 1 1          1 0 1 1
−(+2)    − 0 0 1 0        + 1 1 1 0
-----    ---------        ---------
 (−7)                      1 1 0 0 1
                                ↑
                           se ignora

 (+5)      0 1 0 1          0 1 0 1
−(−2)    − 1 1 1 0        + 0 0 1 0
-----    ---------        ---------
 (+7)                        0 1 1 1

 (−5)      1 0 1 1          1 0 1 1
−(−2)    − 1 1 1 0        + 0 0 1 0
-----    ---------        ---------
 (−3)                      1 1 0 1
```

Figura 5.11 Ejemplos de resta en complemento a 2.

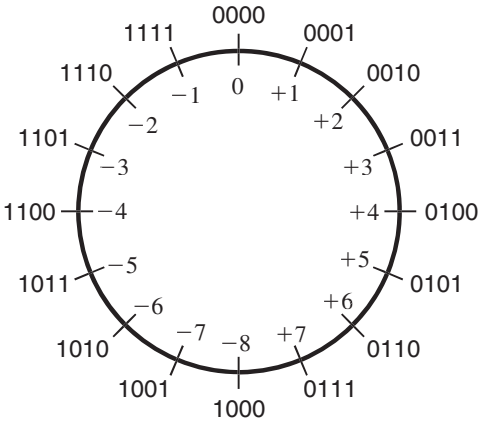


Figura 5.12 Interpretación gráfica de números de cuatro bits en complemento a 2.

operandos. Por tanto, debe ser posible utilizar el mismo circuito sumador para efectuar tanto suma como resta.

5.3.3 UNIDAD SUMADORA Y RESTADORA

La única diferencia entre realizar sumas y restas es que para la resta es necesario usar el complemento a 2 de un operando. Sean X y Y los dos operandos, tal que Y funciona como el sustraendo en la resta. En la sección 5.3.1 aprendimos que el complemento a 2 se obtiene sumando 1 al complemento a 1 de Y. Sumar 1 en la posición del bit menos significativo puede lograrse simplemente poniendo el bit de acarreo c_0 en 1. El complemento a 1 de un número se obtiene complementando cada uno de sus bits. Esto podría hacerse con compuertas NOT, pero necesitamos un circuito más flexible donde pueda emplearse el valor verdadero de Y para la suma y su complemento para la resta.

En la sección 5.2 explicamos que es posible usar compuertas XOR de dos entradas para elegir entre versiones verdadera y complementada de un valor de entrada, bajo el control de la otra entrada. Esta idea puede aplicarse en el diseño de la unidad sumadora/restadora como sigue. Supóngase que existe una señal de control que elige si ha de realizarse suma o resta. Llámese esta señal $\overline{\text{Add}}$/Sub. Además, sea 0 su valor para la suma y 1 para la resta. Para indicar este hecho se coloca una barra sobre Add. Ésta es una convención usada de manera común, donde una barra sobre un nombre significa que la acción especificada por el nombre se toma si la señal de control tiene el valor 0. Ahora, conéctese cada bit de Y a una entrada de una compuerta XOR, con la otra entrada conectada a $\overline{\text{Add}}$/Sub. Las salidas de las compuertas XOR representan Y si $\overline{\text{Add}}/\text{Sub} = 0$, y representan el complemento a 1 de Y si $\overline{\text{Add}}/\text{Sub} = 1$. Esto conduce al circuito de la figura 5.13, cuya parte principal es un sumador de n bits, el cual puede implementarse con la estructura de acarreo en cascada de la figura 5.6. Nótese que la señal de control $\overline{\text{Add}}$/Sub también se conecta al

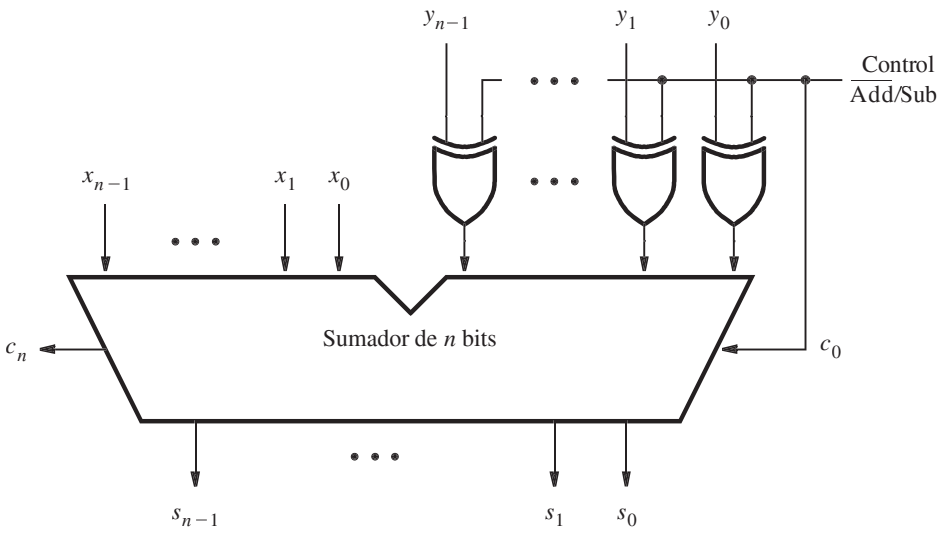


Figura 5.13 Unidad sumadora/restadora.

acarreo c_0. Esto hace $c_0 = 1$ cuando se realiza la resta; por tanto, suma el 1 que se necesita para formar el complemento a 2 de *Y*. Cuando se haga la operación suma se tendrá $c_i = 0$.

La unidad combinada sumador/restador es un buen ejemplo de un concepto importante en el diseño de circuitos lógicos. Es útil diseñar circuitos para que sean lo más flexibles posible y para explotar las partes comunes de los circuitos para cuantas tareas sea factible. Este enfoque minimiza el número de compuertas necesarias para implementar tales circuitos, al tiempo que reduce sustancialmente la complejidad del cableado.

5.3.4 ESQUEMA DE COMPLEMENTO A LA BASE (RAÍZ)

La idea de realizar una resta mediante la suma de un complemento del sustraendo no se restringe a los números binarios. El esquema del complemento a 2 se entiende mejor si consideramos su contraparte en el sistema numérico decimal. Considérese la resta de números decimales de dos dígitos. Calcular un resultado como $74 - 33 = 41$ es simple porque cada dígito del sustraendo es más pequeño que el dígito correspondiente del minuendo; por tanto, no se necesita "pedir prestado" al restar el dígito menos significativo. Pero restar $74 - 36 = 38$ no es tan sencillo porque se requiere "pedir prestado" una vez al restar el dígito menos significativo. Si se "pide prestado", el cálculo se vuelve más complicado.

Supóngase que el cálculo requerido se reestructura del modo siguiente

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Ahora se necesitan dos restas. Restar 36 de 100 aún supone "pedir prestado" una vez. Pero si observamos que $100 = 99 + 1$, ello puede evitarse si escribimos

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

La resta entre paréntesis no requiere "pedir prestado"; se realiza restando de 9 cada dígito del sustraendo. Es posible observar una correlación directa entre esta expresión y la utilizada para el complemento a 2, como se refleja en el circuito de la figura 5.13. La operación (99 – 36) es análoga a complementar el sustraendo *Y* para encontrar su complemento a 1, que es lo mismo que restar cada bit de 1. Si utilizamos números decimales se encuentra el *complemento* a 9 del sustraendo restando cada dígito de 9. En la figura 5.13 se suma el acarreo de 1 para formar el complemento a 2 de *Y*. En nuestro ejemplo decimal se realiza $(99 - 36) + 1 = 64$. Aquí 64 es el complemento a 10 de 36. Para un número decimal de *n* dígitos, *N*, su *complemento a 10,* K_{10}, se define como $K_{10} = 10^n - N$, mientras que su complemento a 9, K_9, es $K_9 = (10^n - 1) - N$.

Por ende, la resta requerida $(74 - 36)$ puede realizarse al sumar el complemento a 10 del sustraendo, como en

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

La resta 138 − 100 es trivial porque significa que el primer dígito de 138 simplemente se borra. Esto es análogo a ignorar el acarreo del circuito en la figura 5.13, como explicamos para los ejemplos de resta de la figura 5.11.

Ejemplo 5.1 Suponga que A y B son números decimales de n dígitos. Si se usa el anterior enfoque de complemento a 10, B puede restarse de A del modo siguiente:

$$A - B = A + (10^n - B) - 10^n$$

Si $A \geq B$, entonces la operación $A + (10^n - \text{B})$ produce un acarreo de 1, el cual es equivalente a 10^n; por tanto, simplemente se puede ignorar.

Pero si A < B, entonces la operación A + $(10^n - \text{B})$ produce un acarreo de 0. Sea M el resultado obtenido, de modo que

$$A - B = M - 10^n$$

Esto puede volverse a escribir como

$$10^n - (B - A) = M$$

El miembro izquierdo de esta ecuación es el complemento a 10 de $(B - A)$. El complemento a 10 de un número positivo representa un número negativo que tiene la misma magnitud. Por ende, M representa correctamente el valor negativo obtenido del cálculo $A - B$ cuando $A < B$. Este concepto se ilustra en los ejemplos siguientes.

Ejemplo 5.2 Cuando se usan números binarios con signo se emplea 0 en el bit que está más a la izquierda para denotar un número positivo y 1 para indicar un número negativo. Si se quiere construir hardware que opere con números decimales con signo, podría usarse un enfoque similar. Sea 0 en la posición del dígito más a la izquierda un número positivo y 9 uno negativo. Note que 9 es el complemento a 9 de 0 en el sistema decimal, así como 1 es el complemento a 1 de 0 en el sistema binario.

Por tanto, al usar números de tres dígitos con signo, $A = 045$ y $B = 027$ son números positivos con magnitudes 45 y 27, respectivamente. El número B puede restarse de A del modo siguiente

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999 - 027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

Esto proporciona la respuesta correcta: +18.

Considere a continuación el caso donde el minuendo tiene un valor menor que el sustraendo, lo cual se ilustra mediante el cálculo

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

A partir de esta expresión parece que tendremos que realizar la resta 982 − 1000. Pero como vimos en el ejemplo 5.1, esto puede volverse a escribir como

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

En consecuencia, 982 es el número negativo que resulta cuando se forma el complemento a 10 de $(A - B)$. Con base en el cálculo previo se sabe que $(A - B) = 018$, que indica +18. Por tanto, el número con signo 982 es la representación en complemento a 10 de −18, que es el resultado requerido.

Ejemplo 5.3

Sean $C = 955$ y $D = 973$; por tanto, los valores de C y D son −45 y −27, respectivamente. El número D puede restarse de C del modo siguiente

$$\begin{aligned} C - D &= 955 - 973 \\ &= 955 + 1000 - 1000 - 973 \\ &= 955 + (999 - 973) + 1 - 1000 \\ &= 955 + 026 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

El número 982 es la representación en complemento a 10 de −18, que es el resultado correcto.

Considere ahora el caso $D - A$, donde $D = 973$ y $A = 045$:

$$\begin{aligned} D - A &= 973 - 045 \\ &= 973 + 1000 - 1000 - 045 \\ &= 973 + (999 - 045) + 1 - 1000 \\ &= 973 + 954 + 1 - 1000 \\ &= 1928 - 1000 \\ &= 928 \end{aligned}$$

El resultado 928 es la representación en complemento a 10 de −72.

Estos ejemplos ilustran que los números con signo pueden restarse sin una operación de resta que implique "pedir prestado". La única resta que se necesita es para formar el complemento a 9 del sustraendo, caso en el que cada dígito simplemente se resta de 9. Por ende, un circuito

que forme el complemento a 9, combinado con un sumador normal, será suficiente tanto para la suma como para la resta de números decimales con signo. Un punto clave es que el hardware necesita trabajar sólo con n dígitos si se usan números de n dígitos. Cualquier acarreo que pueda generarse desde la posición del dígito más a la izquierda simplemente se ignora.

El concepto de restar un número mediante la suma de su complemento a la base es general. Si la base es r, entonces el complemento a r, K_r, de un número de n dígitos, N, se determina como $K_r = r^n - N$. El complemento a $(r - 1)$, K_{r-1}, se define como $K_{r-1} = (r^n - 1) - N$; se calcula simplemente restando cada dígito de N del valor $(r - 1)$. El complemento a $(r - 1)$ se denomina complemento a la base disminuida. Los circuitos para formar los complementos a $(r - 1)$ son más simples que los de resta general que implican "pedir prestado". Los circuitos son particularmente simples en el caso binario, donde el complemento a 1 requiere sólo la inversión de cada bit.

Ejemplo 5.4 En la figura 5.11 ilustramos la operación resta sobre números binarios dados en representación de complemento a 2. Considere el cálculo $(+5) - (+2) = (+3)$ con el enfoque expuesto antes. Cada número está representado por un patrón de cuatro bits. El valor 2^4 se representa como 10000. Luego

$$\begin{aligned} 0101 - 0010 &= 0101 + (10000 - 0010) - 10000 \\ &= 0101 + (1111 - 0010) + 1 - 10000 \\ &= 0101 + 1101 + 1 - 10000 \\ &= 10011 - 10000 \\ &= 0011 \end{aligned}$$

Puesto que $5 > 2$, hay un acarreo de la posición del cuarto bit. Representa el valor 2^4, que se denota mediante el patrón 10000.

Ejemplo 5.5 Considere ahora el cálculo $(+2) - (+5) = (-3)$, que produce

$$\begin{aligned} 0010 - 0101 &= 0010 + (10000 - 0101) - 10000 \\ &= 0010 + (1111 - 0101) + 1 - 10000 \\ &= 0010 + 1010 + 1 - 10000 \\ &= 1101 - 10000 \end{aligned}$$

Puesto que $2 < 5$, no hay acarreo de la posición del cuarto bit. La respuesta, 1101, es la representación en complemento a 2 de -3. Note que

$$\begin{aligned} 1101 &= 10000 + 0010 - 0101 \\ &= 10000 - (0101 - 0010) \\ &= 10000 - 0011 \end{aligned}$$

que indica que 1101 es el complemento a 2 de 0011 $(+3)$.

Ejemplo 5.6

Por último, considere el caso donde el sustraendo es un número negativo. El cálculo $(+5) - (-2) = (+7)$ se realiza del modo siguiente:

$$\begin{aligned} 0101 - 1110 &= 0101 + (10000 - 1110) - 10000 \\ &= 0101 + (1111 - 1110) + 1 - 10000 \\ &= 0101 + 0001 + 1 - 10000 \\ &= 0111 - 10000 \end{aligned}$$

Aunque $5 > (-2)$, el patrón 1110 es mayor que el patrón 0101 cuando los patrones se tratan como números sin signo. Por tanto, no hay acarreo de la posición del cuarto bit. La repuesta 0111 es la representación en complemento a 2 de $+7$. Note que

$$\begin{aligned} 0111 &= 10000 + 0101 - 1110 \\ &= 10000 - (1110 - 0101) \\ &= 10000 - 1001 \end{aligned}$$

y 1001 representa -7.

5.3.5 DESBORDAMIENTO ARITMÉTICO

Se supone que el resultado de la suma o la resta encaja dentro de los bits significativos utilizados para representar los números. Si se usan n bits para representar números con signo, entonces el resultado debe estar en el intervalo que va de -2^{n-1} a $2^{n-1} - 1$. Si el resultado no se halla en ese intervalo, entonces se dice que ocurrió un *desbordamiento aritmético*. Para garantizar la operación correcta de un circuito aritmético es importante detectar cuándo ocurre el desbordamiento.

En la figura 5.14 se presentan los cuatro casos donde se suman números en complemento a 2 con magnitudes de 7 y 2. Puesto que estamos empleando números de cuatro bits, hay tres bits significativos, b_{2-0}. Cuando los números tienen signos opuestos, no hay desbordamiento. Pero si

(+7)	0 1 1 1	(−7)	1 0 0 1
+(+2)	+ 0 0 1 0	+(+2)	+ 0 0 1 0
(+9)	1 0 0 1	(−5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(−7)	1 0 0 1
+ (−2)	+ 1 1 1 0	+ (−2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(−9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figura 5.14 Ejemplos de determinación de desbordamiento.

ambos números tienen el mismo signo, la magnitud del resultado es 9, que no puede representarse sólo con tres bits significativos; por tanto, hay desbordamiento. La clave para determinar si éste se presenta es el acarreo proveniente de la posición del MSB, llamado c_3 en la figura, y el proveniente de la posición del bit del signo, llamado c_4. En la figura se indica que hay desbordamiento cuando esos acarreos tienen diferentes valores, y se produce una suma correcta cuando tienen el mismo valor. De hecho, esto es cierto en general tanto para la suma como para la resta de números en complemento a 2. Como comprobación rápida de esta afirmación, considérense los ejemplos de la figura 5.10, donde los números son lo suficientemente pequeños como para que no haya desbordamiento en ningún caso. En los dos ejemplos de la parte superior de la figura, existe un acarreo de 0 proveniente de las posiciones del signo y del MSB. En los dos ejemplos inferiores, hay un acarreo de 1 proveniente de ambas posiciones. En consecuencia, para los ejemplos de las figuras 5.10 y 5.14 el desbordamiento se detecta mediante

$$\begin{aligned}\text{Desbordamiento} &= c_3\overline{c}_4 + \overline{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

Para números de n bits se tiene

$$\text{Desbordamiento} = c_{n-1} \oplus c_n$$

Por ende, el circuito de la figura 5.13 puede modificarse para incluir comprobación de desbordamiento con la adición de una compuerta XOR.

5.3.6 PROBLEMAS DE RENDIMIENTO

Cuando se compra un sistema digital, como una computadora, el comprador presta atención especial al rendimiento que espera le proporcione el sistema y al costo de adquirirlo. El rendimiento superior casi siempre implica mayor costo. Sin embargo, un gran aumento en el rendimiento puede lograrse con un modesto incremento en el costo. Un indicador del valor de un sistema usado comúnmente es su *razón precio/rendimiento*.

La suma y la resta de números son operaciones fundamentales que se realizan con frecuencia en el curso de un cálculo. La velocidad con la que se evalúan tiene un efecto enorme en el rendimiento total de una computadora. A la luz de esto, echaremos un vistazo cercano a la velocidad de la unidad sumadora/restadora de la figura 5.13. Nos interesa el retraso más largo desde el momento en que los operandos X y Y se presentan como entradas, hasta el instante en que todos los bits de la suma S y el acarreo final, c_n, son válidos. La mayor parte de este retraso lo provoca el circuito sumador de n bits. Supóngase que el sumador se implementa con la estructura de acarreo en cascada de la figura 5.6 y que cada etapa del sumador completo es el circuito de la figura 5.4*c*. El retraso para la señal de acarreo en este circuito, Δt, es igual a dos retrasos de compuerta. En la sección 5.2.2 indicamos que el resultado final de la suma será válido luego de un retraso de $n\Delta t$, que es igual a $2n$ retrasos de compuerta. Además del retraso en la trayectoria del acarreo en cascada, también hay un retraso en las compuertas XOR que alimentan el valor verdadero o el complementado de Y a las entradas del sumador. Si este retraso es igual a un retraso de compuerta, entonces el retraso total del circuito en la figura 5.13 es $2n + 1$ retrasos de compuerta. Para un n grande, digamos $n = 32$ o $n = 64$, el retraso conduciría a un rendimiento inaceptablemente malo. Por tanto, es importante encontrar los circuitos más veloces para realizar la suma.

La velocidad de cualquier circuito está limitada por el retraso mayor a lo largo de las trayectorias a través del circuito. En el caso del circuito de la figura 5.13, el retraso más grande está a

lo largo de la trayectoria que va de la entrada y_i, pasa por la compuerta XOR y por el circuito de acarreo de cada etapa del sumador. El retraso más largo se conoce como *retraso de trayectoria crítica*, y la trayectoria que lo ocasiona se llama *trayectoria crítica*.

5.4 SUMADORES VELOCES

El rendimiento de un gran sistema digital depende de la velocidad de los circuitos que forman sus diversas unidades funcionales. Es obvio que el mejor rendimiento se logra con circuitos más rápidos, los cuales se consiguen empleando tecnología superior (casi siempre más nueva) en la que los retrasos en las compuertas básicas se reducen. Pero también puede lograrse modificando la estructura global de una unidad funcional, lo cual puede conducir incluso a un rendimiento más impresionante. En esta sección explicaremos una posibilidad para implementar un sumador de n bits que reduce sustancialmente el tiempo necesario para sumar números.

5.4.1 SUMADOR CON ACARREO DE ADELANTO

Para reducir el retraso producido por el efecto de propagación de acarreo a través del sumador con acarreo en cascada puede evaluarse rápidamente si en cada etapa el acarreo proveniente de la etapa previa tendrá valor 0 o 1. Si es posible hacer una evaluación correcta en un tiempo hasta cierto punto breve, entonces se mejorará el rendimiento de todo el sumador.

A partir de la figura 5.4*b*, la función acarreo para la etapa i puede realizarse como

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Si esta expresión se factoriza como

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

entonces se puede escribir como

$$c_{i+1} = g_i + p_i c_i \qquad \textbf{[5.3]}$$

donde

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

La función g_i es igual a 1 cuando ambas entradas, x_i y y_i, son iguales a 1, independientemente del valor del acarreo entrante a esta etapa, c_i. Como en este caso está garantizado que la etapa i generará un acarreo, a g se le llama función *generada*. La función p_i es igual a 1 cuando al menos una de las entradas, x_i y y_i, es igual a 1. En este caso se produce un acarreo si $c_i = 1$. El efecto es que el acarreo de 1 se propaga a lo largo de la etapa i; por tanto, p_i se denomina función *propagada*.

Al expandir la expresión 5.3 en términos de la etapa $i - 1$ se produce

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

La misma expansión para las otras etapas, finalizando con la etapa 0, produce

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \qquad \textbf{[5.4]}$$

Esta expresión representa un circuito AND-OR de dos niveles en el que c_{i+1} se evalúa muy rápidamente. Un sumador basado en esta expresión se denomina *sumador con acarreo de adelanto*.

Para apreciar el significado físico de la expresión 5.4 es necesario considerar su efecto en la construcción de un sumador veloz en comparación con los detalles del sumador con acarreo en cascada. Así lo haremos al examinar la estructura detallada de las dos etapas que suman los bits menos significativos: las etapas 0 y 1. En la figura 5.15 se muestran las primeras dos etapas de un sumador con acarreo en cascada en el que las funciones de acarreo se implementan como se indica en la expresión 5.3. Cada etapa es en esencia el circuito de la figura 5.4*c*, excepto que

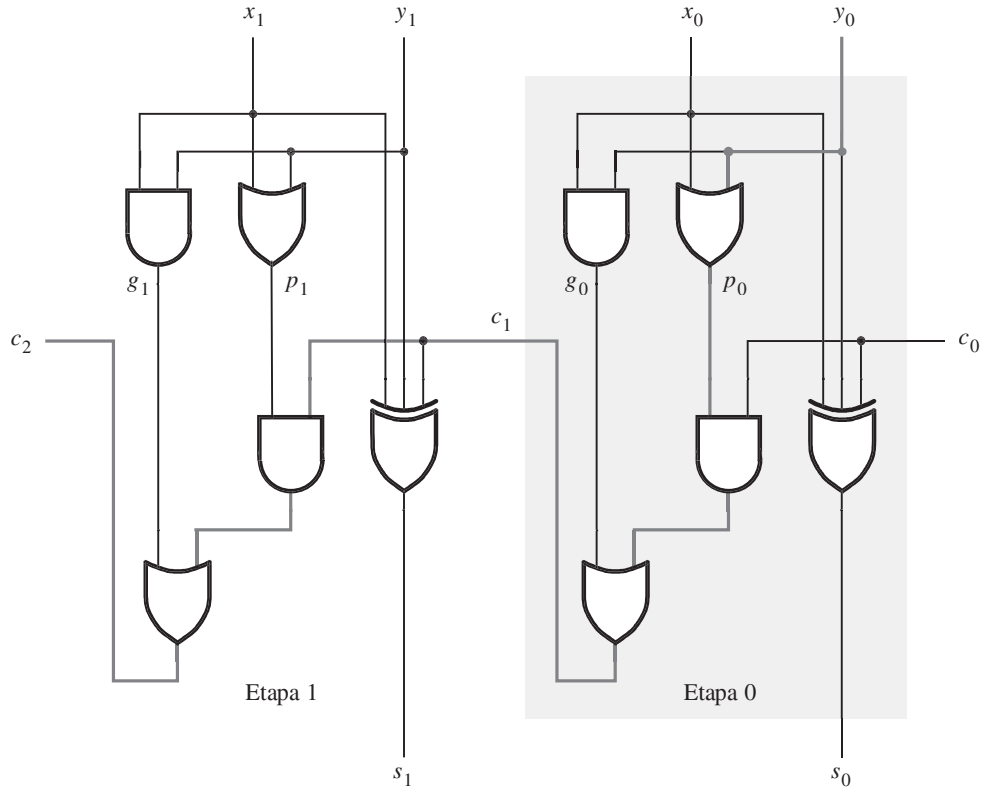


Figura 5.15 Sumador con acarreo en cascada basado en la expresión 5.3.

se usa una compuerta OR adicional (que produce la señal p_i), en lugar de una compuerta AND porque se factorizó la expresión de suma de productos para c_{i+1}.

La lentitud del sumador con acarreo en cascada es producto de la larga trayectoria por la que una señal de acarreo debe propagarse. En la figura 5.15 la trayectoria crítica va de las entradas x_0 y y_0 a la salida c_2. Pasa por cinco compuertas, resaltadas en gris oscuro. La trayectoria en las otras etapas de un sumador de n bits es la misma que en la etapa 1. Por tanto, el retraso total a lo largo de la trayectoria crítica es $2n + 1$.

En la figura 5.16 se presentan las primeras dos etapas del sumador con acarreo de adelanto; se usa la expresión 5.4 para implementar las funciones de acarreo. En consecuencia

$$c_1 = g_0 + p_0c_0$$
$$c_2 = g_1 + p_1g_0 + p_1p_0c_0$$

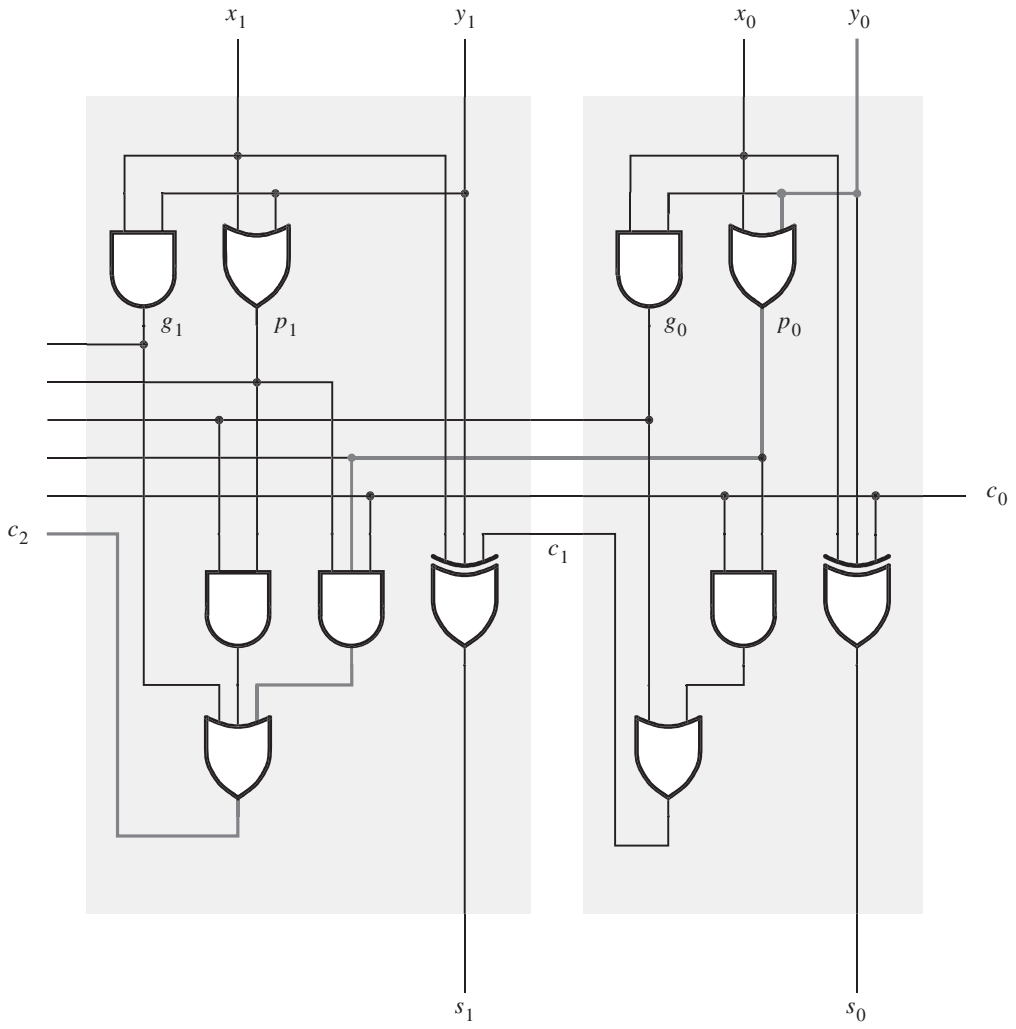


Figura 5.16 Primeras dos etapas de un sumador con acarreo de adelanto.

La trayectoria crítica para producir la señal c_2 se resalta en gris oscuro. En este circuito, c_2 se produce tan rápido como c_1, después de un total de tres retrasos de compuerta. Si extendemos el circuito a *n* bits, la señal de acarreo final c_n también se producirá después de sólo tres retrasos de compuerta, ya que la expresión 5.4 es un gran circuito de dos niveles (AND-OR).

El retraso total en el sumador de *n* bits con acarreo de adelanto es de cuatro retrasos de compuerta. Los valores de todas las señales g_i y p_i se determinan después de un retraso de compuerta. Se precisan dos retrasos de compuerta más para evaluar todas las señales de acarreo. Finalmente, se requiere un retraso de compuerta más (XOR) para generar todos los bits suma. La clave para el buen rendimiento del sumador es la evaluación rápida de las señales de acarreo.

La complejidad de un sumador de *n* bits con acarreo de adelanto aumenta rápidamente conforme *n* se vuelve más grande. Para reducir la complejidad es posible aplicar un enfoque *jerárquico* al diseñar sumadores grandes. Supóngase que queremos diseñar un sumador de 32 bits. Este sumador puede dividirse en cuatro bloques de ocho bits, de tal modo que los bits $b_{7\text{-}0}$ sean el bloque 0, los bits $b_{15\text{-}8}$ sean el bloque 1, los bits $b_{23\text{-}16}$ sean el bloque 2 y los bits $b_{31\text{-}24}$ sean el bloque 3. Luego podemos implementar cada bloque como un sumador de ocho bits con acarreo de adelanto. Las señales de acarreo provenientes de los cuatro bloques son c_8, c_{16}, c_{24} y c_{32}. Ahora tenemos dos posibilidades. Se pueden conectar los cuatro bloques como cuatro etapas en un sumador con acarreo en cascada. Por ende, mientras el acarreo de adelanto se use dentro de cada bloque, el acarreo cae en cascada entre los bloques. Este circuito se ilustra en la figura 5.17.

En vez de usar un enfoque de acarreo en cascada entre bloques es posible diseñar un circuito más veloz en el que se realice un acarreo de adelanto de segundo nivel para producir rápidamente las señales de acarreo entre bloques. La estructura de este "sumador jerárquico con acarreo de adelanto" se muestra en la figura 5.18. Cada bloque de la fila superior incluye un sumador de ocho bits con acarreo de adelanto, basado en las señales generadas, g_i, y las señales propagadas, p_i, para cada etapa en el bloque, como explicamos líneas arriba. Sin embargo, en vez de producir una señal de acarreo proveniente del bit más significativo del bloque, cada bloque produce señales generada y propagada para todo el bloque. Sean G_j y P_j dichas señales para cada bloque *j*. Ahora G_j y P_j se usan como entradas a un circuito con acarreo de adelanto de segundo nivel —parte inferior de la figura 5.18— que evalúe todos los acarreos entre bloques. Es posible derivar las señales de bloque generada y propagada para el bloque 0 examinando la expresión para c_8

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

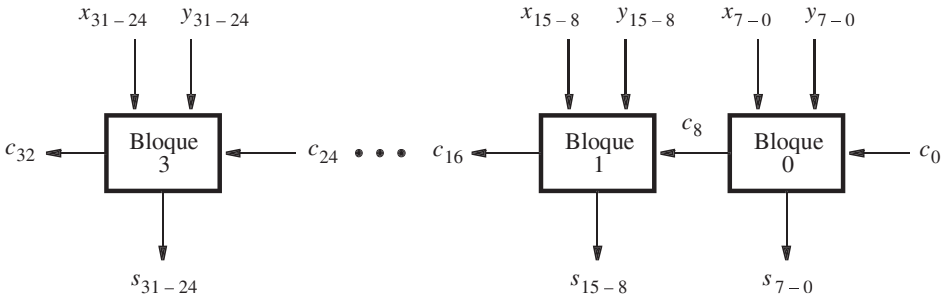


Figura 5.17 Sumador jerárquico con acarreo de adelanto con acarreo en cascada entre bloques.

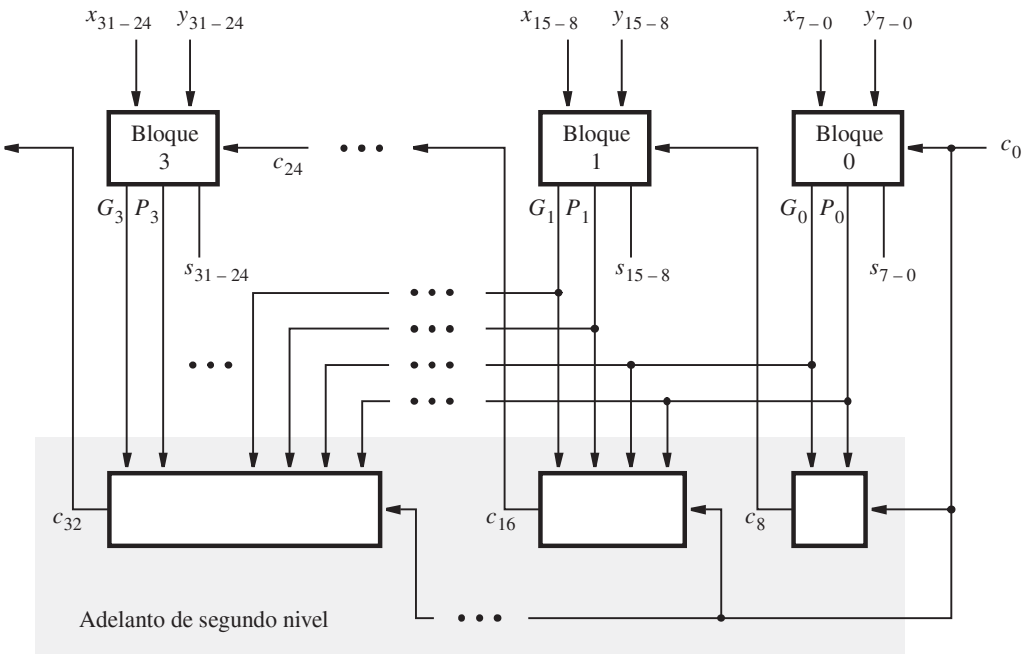


Figura 5.18 Sumador jerárquico con acarreo de adelanto.

El último término de esta expresión indica que si las ocho funciones propagadas son 1, entonces el acarreo c_0 se propaga por todo el bloque. En consecuencia

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

El resto de los términos de la expresión para c_8 representan todos los demás casos cuando el bloque produce un acarreo. Por tanto

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

La expresión para c_8 en el sumador jerárquico está dada por

$$c_8 = G_0 + P_0c_0$$

Para el bloque 1, las expresiones para G_1 y P_1 tienen la misma forma que para G_0 y P_0, excepto que cada subíndice i se sustituye con $i + 8$. Las expresiones para G_2, P_2, G_3 y P_3 se deducen de la misma forma. La expresión para el acarreo del bloque 1, c_{16}, es

$$\begin{aligned} c_{16} &= G_1 + P_1c_8 \\ &= G_1 + P_1G_0 + P_1P_0c_0 \end{aligned}$$

De manera similar, las expresiones para c_{24} y c_{32} son

$$\begin{aligned} c_{24} &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0 \\ c_{32} &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0 \end{aligned}$$

Si se utiliza este esquema se precisan dos retrasos de compuerta más para producir las señales de acarreo c_8, c_{16} y c_{24} que el tiempo necesario para generar las funciones G_j y P_j. Por consiguiente, como G_j y P_j requieren tres retrasos de compuerta, c_8, c_{16} y c_{24} quedan disponibles después de cinco retrasos de compuerta. El tiempo necesario para sumar dos números de 32 bits supone estos cinco retrasos de compuerta más dos adicionales para producir los acarreos internos en los bloques 1, 2 y 3, más un retraso de compuerta adicional (XOR) para generar todos los bits suma. Esto da un total de ocho retrasos de compuerta.

En la sección 5.3.5 establecimos que se requieren $2n + 1$ retrasos de compuerta para sumar dos números con el sumador con acarreo en cascada. Para números de 32 bits esto implica 65 retrasos de compuerta. Es evidente que el sumador con acarreo de adelanto ofrece una mejora mayor en el rendimiento. A cambio, el circuito requerido tiene una complejidad mucho mayor.

Consideraciones tecnológicas

El anterior análisis de retraso supone que es posible utilizar compuertas con cualquier número de entradas. En los capítulos 3 y 4 explicamos que la tecnología empleada para implementar las compuertas limita la carga de entrada a un número más bien pequeño de entradas. Por tanto, hay que tomar en cuenta la realidad de las restricciones en la carga de entrada. Para ilustrar este problema, considérense las expresiones para los primeros ocho acarreos:

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\
&\vdots \\
c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\
&\quad + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0
\end{aligned}$$

Supóngase que la máxima entrada de carga de las compuertas es de cuatro entradas. Entonces es imposible implementar todas estas expresiones con un circuito AND-OR de dos niveles. El mayor problema es c_8, donde una de las compuertas AND requiere nueve entradas; más aún, la compuerta OR también precisa nueve entradas. Para satisfacer la restricción de la carga de entrada la expresión para c_8 puede volverse a escribir como

$$\begin{aligned}
c_8 = {} & (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\
& + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0
\end{aligned}$$

Para implementar esta expresión se necesitan 11 compuertas AND y tres OR. El retraso de propagación para generar c_8 consta de un retraso de compuerta para desarrollar todas las g_i y p_i, dos retrasos de compuerta para producir los términos suma de productos entre paréntesis, un retraso de compuerta para formar el término producto entre corchetes y un retraso para la operación OR final de los términos. En consecuencia, c_8 es válida después de cinco retrasos de compuerta, en lugar de tres retrasos de compuerta que se necesitarían sin la restricción de carga de entrada.

Puesto que las limitaciones de carga de entrada reducen la velocidad del sumador con acarreo de adelanto, algunos dispositivos que se caracterizan por baja entrada de carga incluyen circuitos dedicados para la implementación de sumadores veloces. Ejemplos de tales dispositivos incluyen los FPGA, cuyos bloques lógicos se basan en tablas de consulta.

Antes de dejar el tema de los sumadores con acarreo de adelanto hay que considerar otra implementación de la estructura de la figura 5.16. Es posible obtener la misma funcionalidad con el circuito de la figura 5.19. En este caso, la etapa 0 se implementa con el circuito de la

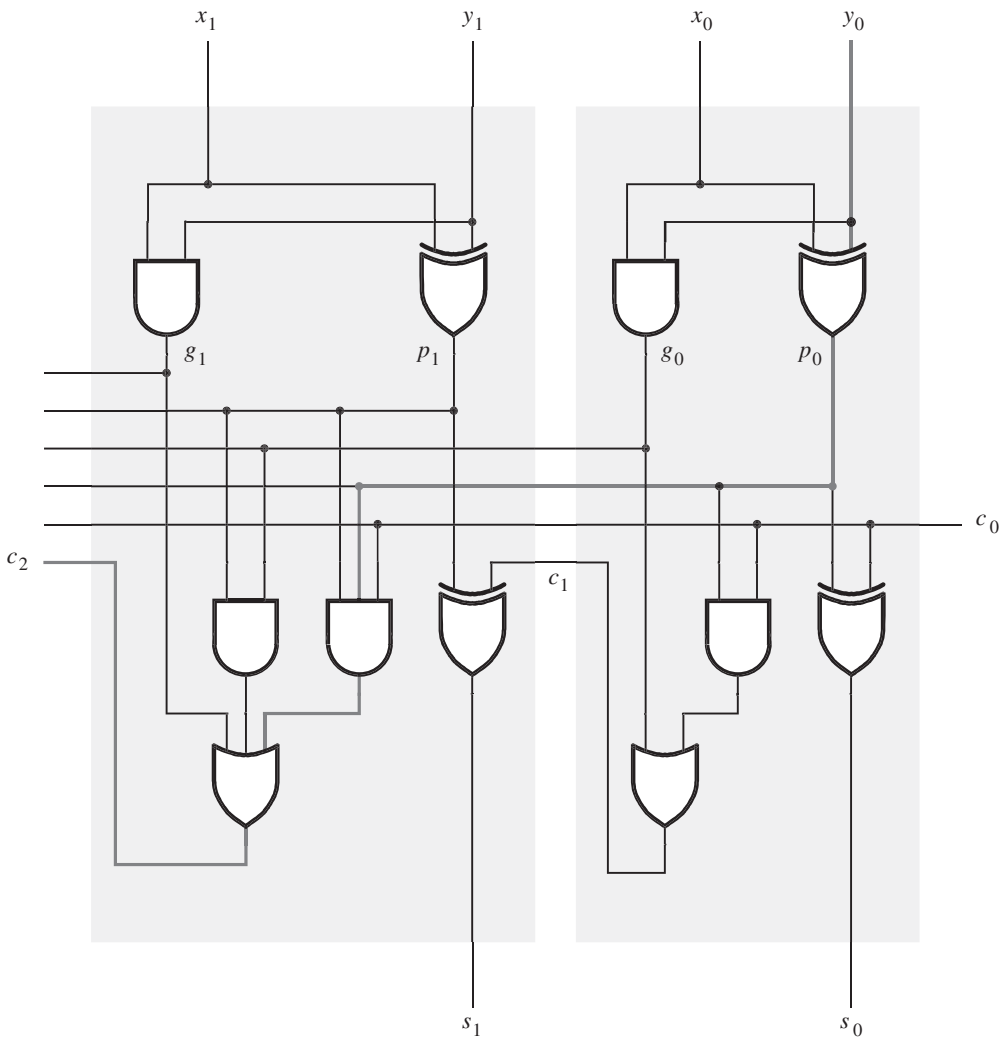


Figura 5.19 Otro diseño para un sumador con acarreo de adelanto.

figura 5.5, en el que se usan dos compuertas XOR de dos entradas para generar el bit suma, en vez de tener 1 compuerta XOR de tres entradas. La salida de la primera compuerta XOR también puede servir como la señal propagada p_0. Por tanto, no se necesita la correspondiente compuerta OR de la figura 5.16. La etapa 1 se construye siguiendo el mismo enfoque.

Los circuitos de las figuras 5.16 y 5.19 requieren el mismo número de compuertas. ¿Pero alguno de ellos es mejor en algún sentido? La respuesta debe buscarse al considerar los aspectos específicos de la tecnología usada para implementar los circuitos. Si se emplearon un CPLD o un FPGA, como los de las figuras 3.33 y 3.39, entonces no importa cuál circuito se elija. Con una macrocelda puede realizarse una función XOR de tres entradas en el CPLD, utilizando la

expresión en suma de productos

$$s_i = x_i\overline{y}_i\overline{c}_i + \overline{x}_i y_i\overline{c}_i + \overline{x}_i\overline{y}_i c_i + x_i y_i c_i$$

porque la macrocelda permite la implementación de cuatro términos producto.

En el FPGA, cualquier función de tres entradas puede implementarse en una sola celda lógica; por tanto, es fácil realizar una XOR de tres entradas. Sin embargo, supóngase que deseamos construir un sumador con acarreo de adelanto en un chip a la medida. Si la compuerta XOR se construye con el enfoque expuesto en la sección 3.9.1, entonces en realidad se implementaría una XOR de tres entradas usando dos compuertas XOR de dos entradas, como se hizo para los bits suma de la figura 5.19. Por ende, si la primera compuerta XOR realiza la función $x_i \oplus y_i$, que también es la función propagada p_i, entonces es obvio que la alternativa presentada en la figura 5.19 es más atractiva. El punto importante de esta explicación es que la optimización de los circuitos lógicos puede depender de la tecnología destino. Las herramientas CAD toman en cuenta este hecho.

El sumador con acarreo de adelanto es un concepto bien conocido. Existen chips estándar que implementan una parte del circuito de acarreo de adelanto. Se llaman *generadores de acarreo de adelanto*. Las herramientas CAD suelen incluir subcircuitos prediseñados para sumadores, que los diseñadores pueden usar para diseñar unidades más grandes.

5.5 DISEÑO DE CIRCUITOS ARITMÉTICOS CON EL USO DE HERRAMIENTAS CAD

En esta sección mostraremos cómo diseñar circuitos aritméticos con herramientas CAD. Analizaremos dos métodos de diseño: uso de captura esquemática y uso de código de VHDL.

5.5.1 DISEÑO DE CIRCUITOS ARITMÉTICOS CON EL USO DE CAPTURA ESQUEMÁTICA

Una forma obvia de diseñar un circuito aritmético mediante captura esquemática es trazar un esquema que contenga las compuertas lógicas necesarias. Por ejemplo, para crear un sumador de *n* bits, primero podríamos dibujar un esquema que represente un sumador completo. Luego podría crearse un sumador de *n* bits con acarreo en cascada trazando un esquema de mayor nivel que conecte las *n instancias* del sumador completo. Un esquema jerárquico creado de esta forma se parecería al circuito de la figura 5.6. También podríamos usar este método para crear un circuito sumador/restador, como el bosquejado en la figura 5.13.

El problema principal de este enfoque es que resulta engorroso, sobre todo cuando el número de bits es grande. Ello se torna aún más evidente si consideramos la creación de un esquema para un sumador con acarreo de adelanto. Como expusimos en la sección 5.4.1, los circuitos de acarreo en cada etapa del sumador con acarreo de adelanto se vuelven cada vez más complejos. Por ende, es preciso trazar un esquema separado por cada una de sus etapas. Un mejor enfoque para crear circuitos aritméticos mediante captura esquemática es usar subcircuitos predefinidos.

En la sección 2.9.1 dijimos que las herramientas de captura esquemática ofrecen una biblioteca de símbolos gráficos que representan compuertas lógicas básicas. Tales compuertas sirven para crear esquemas de circuitos relativamente simples. Además de las compuertas básicas, la

mayor parte de las herramientas de captura esquemática también ofrece una biblioteca de circuitos de uso común, como los sumadores. Cada circuito se ofrece como un módulo que puede importarse en un esquema y utilizarse como parte de un circuito más grande. En algunos sistemas CAD, los módulos se llaman *macrofunciones* o *megafunciones*.

Hay dos tipos principales de macrofunciones: dependientes de la tecnología e independientes de ella. Una *macrofunción dependiente de la tecnología* está diseñada para adaptarse a tipos específicos de chip. Por citar un caso, en la sección 5.4.1 describimos una expresión para un sumador con acarreo de adelanto diseñado para satisfacer una restricción en la carga de entrada de compuertas de cuatro entradas. Una macrofunción que implemente esta expresión sería de tecnología específica. Una *macrofunción independiente de la tecnología* puede implementarse en cualquier tipo de chip. La macrofunción para un sumador que represente diferentes circuitos para distintos tipos de chips es independiente de la tecnología.

Un buen ejemplo de biblioteca de macrofunciones es la *Library of Parameterized Modules* (*LPM*, biblioteca de módulos parametrizados) incluida como parte del sistema CAD Quartus II. Cada módulo de la biblioteca es independiente de la tecnología. Además, está *parametrizado*, lo que significa que puede usarse de varias formas. Por ejemplo, la biblioteca LPM incluye un módulo sumador de *n* bits, llamado *lpm_add_sub*.

En la figura 5.20 se presenta una ilustración esquemática de la capacidad del módulo *lpm_add_sub*, el cual tiene varios parámetros asociados que se configuran mediante las herramientas CAD. Los dos parámetros más importantes para los propósitos de nuestra explicación se llaman LPM_WIDTH y LPM_REPRESENTATION. El parámetro LPM_WIDTH especifica el número de bits, *n*, en el sumador. El parámetro LPM_REPRESENTATION especifica si se usan enteros con o sin signo. Esto sólo afecta la parte del módulo que determina cuándo ocurre desbordamiento aritmético. Para el esquema mostrado, LPM_WIDTH = 16, y se usan números con signo.

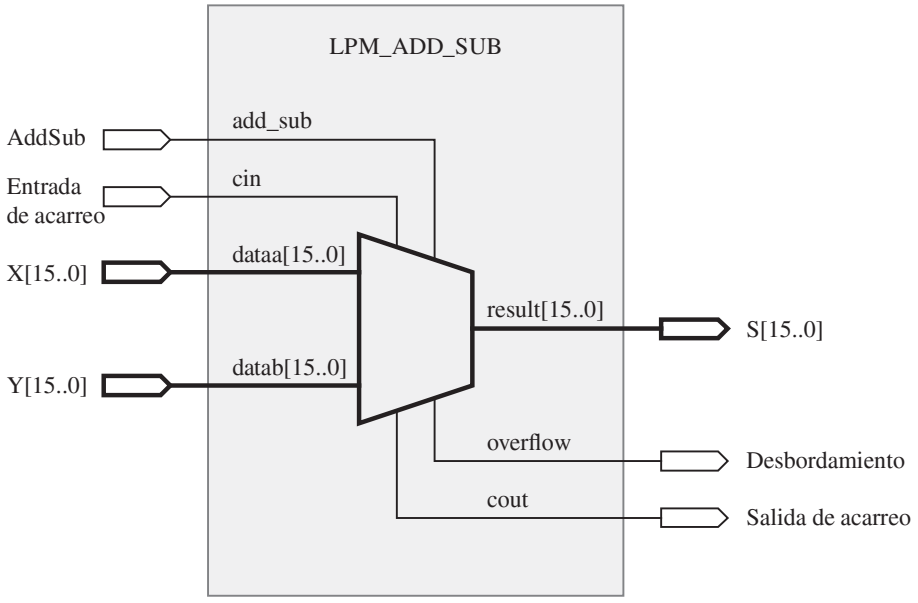


Figura 5.20 Esquema que usa un módulo LPM sumador/restador.

El módulo puede realizar sumas o restas, determinadas por la entrada *add_sub*. Por ende, el módulo representa un circuito sumador/restador, como el mostrado en la figura 5.13.

Los números que el módulo *lpm_add_sub* sumará se conectan a las terminales llamadas *dataa*[15..0] y *datab*[15..0]. Los corchetes en estos nombres significan que representan números multibit. En el esquema, *dataa* y *datab* se conectan a las señales de entrada de 16 bits *X*[15..0] y *Y*[15..0]. El significado de la sintaxis *X*[15..0] es que la señal *X* representa 16 bits, llamados *X*[15], *X*[14], . . . , *X*[0]. El módulo *lpm_add_sub* produce la suma en la terminal llamada *result*[15..0], que se conecta a la salida *S*[15..0]. En la figura 5.20 también se muestra que la LPM soporta una entrada de acarreo, así como salidas de acarreo y desbordamiento.

Para valorar la eficacia de la LPM, el módulo *lpm_add_sub* se configura para realizar un sumador de 16 bits que calcule las salidas suma, acarreo y desbordamiento; esto significa que las señales *add_sub* y *cin* no se necesitan. Usamos herramientas CAD para implementar este circuito en un chip FPGA, y simulamos su rendimiento. El diagrama de tiempo resultante se muestra en la figura 5.21, que es una pantalla del simulador de tiempo. Los valores de las señales de 16 bits, *X*, *Y* y *S* se muestran en la salida de simulación como números hexadecimales. Al comienzo de la simulación, *X* y *Y* se establecen en 0000. Después de 50 ns, *Y* cambia a 0001, lo que ocasiona que *S* cambie a 0001. El siguiente cambio en las entradas ocurre a 150 ns, cuando *X* cambia a 3FFF. Para producir la nueva suma, que es 4000, el sumador debe esperar a que sus señales de acarreo caigan en cascada de la primera a la última etapas. Esto se advierte en la salida de la simulación como una secuencia de cambios rápidos en el valor de *S*, y posteriormente se estabiliza en la suma correcta. Nótese que la línea de referencia del simulador, la línea vertical gruesa en la figura, muestra que la suma correcta se produce 160.93 ns desde el principio de la simulación. Puesto que el cambio en las entradas ocurrió a 150 ns, el sumador demora 160.93 – 150 = 10.93 ns para calcular la suma. A 250 ns, *X* cambia a 7FFF, lo que ocasiona que la suma sea 8000. Esta suma es demasiado grande para un número con signo de 16 bits; por tanto, *Overflow* se hace 1 para indicar el desbordamiento aritmético.

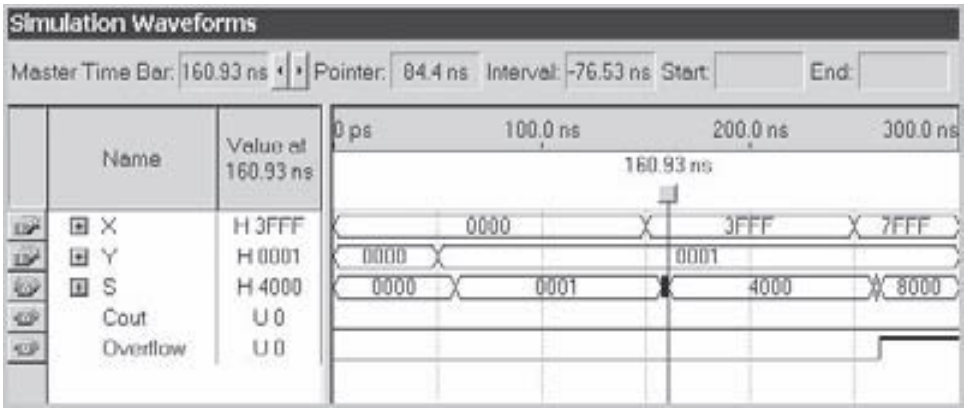


Figura 5.21 Resultados de simulación para el sumador de la LPM.

5.5.2 DISEÑO DE CIRCUITOS ARITMÉTICOS CON VHDL

En la sección 5.5.1 dijimos que una forma obvia de crear un sumador de *n* bits es trazar un esquema jerárquico que contenga *n* sumadores completos. Este enfoque también puede seguirse con VHDL: primero se crea una entidad de VHDL para un sumador completo y luego una entidad de nivel superior que use cuatro *instancias* del sumador completo. Como primer intento en el diseño de circuitos aritméticos mediante VHDL mostraremos cómo escribir el código jerárquico para un sumador con acarreo en cascada.

En la figura 5.22 se presenta el código completo para una entidad de sumador completo. Tiene las entradas *Cin*, *x* y *y*, y produce las salidas *s* y *Cout*. La suma, *s*, y el acarreo de salida, *Cout*, se describen mediante ecuaciones lógicas.

Ahora debemos crear una entidad de VHDL separada para el sumador con acarreo en cascada, que use la entidad *fulladd* como un subcircuito. En la figura 5.23 se muestra un método para hacerlo. En ella se proporciona el código para una entidad sumador con acarreo en cascada de cuatro bits, llamada *adder4*. Uno de los números de cuatro bits por sumar se representa mediante las cuatro señales x_3, x_2, x_1, x_0, y el otro número con y_3, y_2, y_1, y_0. La suma se representa s_3, s_2, s_1, s_0.

Obsérvese que el cuerpo arquitectónico tiene el nombre *Structure*, el cual se eligió porque el estilo del código en el que se describe un circuito de forma jerárquica, mediante la conexión en conjunto de los subcircuitos, usualmente se llama *estructural*. En ejemplos previos de código de VHDL, todas las señales usadas se declararon como puertos en la declaración de entidad. Como se muestra en la figura 5.23, las señales también pueden declarar antes de la palabra clave BEGIN en el cuerpo arquitectónico. Las tres señales declaradas, denominadas c_1, c_2 y c_3, se utilizan como señales de acarreo provenientes de las tres primeras etapas del sumador. La instrucción siguiente se llama de *declaración de componentes*. Utiliza una sintaxis similar a la de una declaración de entidad, y permite que la entidad *fulladd* se use como componente (subcircuito) en el cuerpo arquitectónico.

El sumador de cuatro bits de la figura 5.23 se describe empleando cuatro instrucciones de *instanciación*. Cada una de ellas comienza con un *nombre de instancia*, que puede ser cualquier nombre legal en VHDL, seguido por el signo de dos puntos. Los nombres deben ser únicos. La etapa menos significativa del sumador se llama *stage0*, y la más significativa *stage3*. Después

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y   : IN     STD_LOGIC ;
           s, Cout     : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

Figura 5.22 Código de VHDL para el sumador completo.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin            : IN    STD_LOGIC ;
           x3, x2, x1, x0 : IN    STD_LOGIC ;
           y3, y2, y1, y0 : IN    STD_LOGIC ;
           s3, s2, s1, s0 : OUT   STD_LOGIC ;
           Cout           : OUT   STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN    STD_LOGIC ;
               s, Cout   : OUT   STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figura 5.23 Código de VHDL para un sumador de cuatro bits.

de los dos puntos sigue el nombre del componente, *fulladd*, y luego la palabra clave PORT MAP (mapa de puertos). Entonces se hace una lista con los nombres de las señales en la entidad *adder4* que se conectarán a cada puerto de entrada y salida en el componente *fulladd*. Obsérvese que en las primeras tres instrucciones de instanciación, las señales se enumeran en el mismo orden que en la instrucción de declaración COMPONENT de *fulladd*: *Cin*, *x*, *y*, *s*, *Cout*. Pueden enumerarse en otro orden especificando explícitamente cuál de ellas se conectará a qué puerto en el componente. Un ejemplo de este estilo se muestra para la instancia *stage3*. Este estilo de instanciación de componente se conoce como *asociación por nombre* en la terminología de VHDL, mientras que el estilo usado en las otras tres instancias se denomina *asociación por posición*. Nótese que para la instancia *stage3* el nombre de señal *Cout* se usa tanto para el nombre del puerto componente como para el de la señal en la entidad *adder4*. Esto no causa problema al compilador de VHDL, pues el nombre del puerto componente siempre es el del lado izquierdo de los caracteres =>.

Los nombres de señal asociados con cada instancia del componente *fulladd* especifican de manera implícita cómo se conectan los sumadores completos. Por ejemplo, la salida de acarreo de la instancia *stage0* se conecta a la entrada de acarreo de la instancia *stage1*. Cuando el código de la figura 5.23 se analiza mediante el compilador de VHDL, automáticamente busca el código que

debe emplear para el componente *fulladd* presentado en la figura 5.22. El circuito sintetizado tiene la misma estructura que el de la figura 5.6.

Estilo alternativo de código

En la figura 5.23, una instrucción de declaración de componente para la entidad *fulladd* se incluye en la arquitectura *adder4*. Un enfoque alternativo consiste en colocar la instrucción de declaración de componente en un *paquete* de VHDL. En general, un paquete permite que los constructores de VHDL se definan en un archivo de código fuente y luego se usen en otros archivos semejantes. Dos ejemplos de constructores que con frecuencia se colocan en un paquete son las declaraciones de tipo datos y las de componentes.

Ya vimos un ejemplo de cómo utilizar un paquete para un tipo de datos. En el capítulo 4 presentamos el paquete llamado *std_logic_1164*, que define el tipo de señal STD_LOGIC. Recuérdese que para acceder a este paquete el código de VHDL debe incluir las instrucciones

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

Las cuales aparecen en las figuras 5.22 y 5.23 porque el tipo STD_LOGIC se usa en el código. La primera proporciona acceso a la biblioteca llamada *ieee*. Como explicamos en la sección 4.12, la biblioteca representa la ubicación, o *directorio*, en el sistema de archivos de la computadora donde se almacena el paquete *std_logic_1164*.

El código de la figura 5.24 define el paquete llamado *fulladd_package*. Este código puede almacenarse en un archivo de código fuente de VHDL por separado o incluirse en el mismo archivo de código fuente utilizado para almacenar el código de la entidad *fulladd* mostrado en la figura 5.22. La sintaxis de VHDL exige que la declaración de paquete tenga sus propias cláusulas LIBRARY y USE; por tanto, se incluyen en el código. Dentro del paquete, la entidad *fulladd* se declara como un COMPONENT. Cuando este código se compila se crea el paquete *fulladd_package* y se almacena en el directorio de trabajo donde se guarda el código.

Entonces toda entidad de VHDL puede usar el componente *fulladd* como un subcircuito para utilizar el paquete *fulladd_package*. Al paquete se accede empleando las dos instrucciones siguientes

```
LIBRARY work;
USE work.fulladd_package.all ;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figura 5.24 Declaración de un paquete.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin            : IN   STD_LOGIC ;
           x3, x2, x1, x0 : IN   STD_LOGIC ;
           y3, y2, y1, y0 : IN   STD_LOGIC ;
           s3, s2, s1, s0 : OUT  STD_LOGIC ;
           Cout           : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

Figura 5.25 Una forma diferente de especificar un sumador de cuatro bits.

La biblioteca llamada *work* representa el directorio de trabajo donde se almacena el código de VHDL que define el paquete. En realidad, esta instrucción no es necesaria, ya que el compilador de VHDL siempre tiene acceso al directorio de trabajo.

En la figura 5.25 se muestra cómo volver a escribir el código de la figura 5.23 para usar el *fulladd_package*. El código es el mismo que el de la figura 5.23 con dos excepciones: se agrega la cláusula adicional USE y en la arquitectura se borra la instrucción de declaración de componente. Los circuitos sintetizados de las dos versiones del código son idénticos.

En las figuras 5.23 y 5.25, cada una de las entradas de cuatro bits y la salida de cuatro bits del sumador se representan con señales de un solo bit. Un estilo más de código más práctico es usar señales multibit para representar los números.

5.5.3 REPRESENTACIÓN DE NÚMEROS EN CÓDIGO DE VHDL

Igual que un número se representa en un circuito lógico como señales sobre varios cables, un número se indica en código de VHDL como un objeto de datos SIGNAL multibit. El siguiente es un ejemplo de una señal multibit

```
SIGNAL C : STD_LOGIC_VECTOR (1 TO 3) ;
```

El tipo de datos STD_LOGIC_VECTOR representa un arreglo lineal de objetos de datos STD_LOGIC. En VHDL se dice que STD_LOGIC_VECTOR es un subtipo de STD_LOGIC. Hay un subtipo similar, llamado BIT_VECTOR, que corresponde al tipo BIT usado en la sec-

ción 2.10.2. La precedente declaración SIGNAL define C como una señal STD_LOGIC de tres bits. Puede usarse en código de VHDL como una cantidad de tres bits simplemente empleando el nombre *C*, o de otro modo cada bit individual se puede referir por separado mediante los nombres *C*(1), *C*(2) y *C*(3). La sintaxis 1 TO 3 en la instrucción de declaración especifica que el bit más significativo en *C* se llama *C*(1) y el menos significativo *C*(3). A *C* puede asignársele un valor de señal de tres bits del modo siguiente:

```
C <= "100" ;
```

El valor de tres bits se denota con comillas, en lugar de los apóstrofos que se utilizan para valores de un bit, como en '1' o '0'. La instrucción de asignación resulta en $C(1) = 1$, $C(2) = 0$ y $C(3) = 0$. La numeración de los bits en la señal *C*, con el índice más alto usado para el menos significativo, es una forma natural de representar señales que sólo están agrupadas por conveniencia pero que no representan un número. Por ejemplo, este esquema de numeración sería una forma adecuada de declarar las tres señales de acarreo llamadas c_1, c_2 y c_3 en la figura 5.25. Sin embargo, cuando se emplea una señal multibit para representar un número binario es más lógico numerar los bits de forma opuesta, con el índice más alto para el bit más significativo. Para este propósito, VHDL ofrece una segunda forma de declarar una señal multibit:

```
SIGNAL X : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

Esta instrucción define *X* como una señal STD_LOGIC_VECTOR de cuatro bits. La sintaxis 3 DOWNTO 0 especifica que el bit más significativo en *X* se llama *X*(3) y el menos significativo es *X*(0). Este esquema es una forma más natural de numerar los bits si se usará *X* en el código de VHDL para representar un número binario porque el índice de cada bit corresponde a su posición en el número. La instrucción de asignación

```
X <= "1100" ;
```

resulta en $X(3) = 1, X(2) = 1, X(1) = 0$ y $X(0) = 0$.

En la figura 5.26 se muestra cómo escribir el código de la figura 5.25 para usar señales multibit. Las entradas de datos son las señales de cuatro bits *X* y *Y*, y la salida suma es la señal de cuatro bits *S*. Las señales intermedias de acarreo se declaran en la arquitectura como la señal de tres bits *C*.

El uso de código de VHDL jerárquico para definir grandes circuitos aritméticos puede ser engorroso. Por ello, los circuitos aritméticos se implementan de otra forma en VHDL, con instrucciones de asignación aritmética y señales multibit.

5.5.4 INSTRUCCIONES DE ASIGNACIÓN ARITMÉTICA

Si se definen las señales siguientes

```
SIGNAL X, Y, S : STD_LOGIC_VECTOR (15 DOWNTO 0) ;
```

entonces la instrucción de asignación aritmética

```
S <= X + Y;
```

representa un sumador de 16 bits.

Además del operador +, usado para la suma, VHDL ofrece otros operadores aritméticos. En la tabla A.1 del Apéndice A hay una lista de ellos. En la figura 5.27 se ofrece todo el código

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figura 5.26 Sumador de cuatro bits definido mediante señales multibit.

de VHDL que incluye la instrucción anterior. El paquete *std_logic_1164* no especifica que las señales STD_LOGIC puedan utilizarse con operadores aritméticos. El segundo paquete incluido en el código, *std_logic_signed*, permite que las señales se usen de esta forma. Cuando el compilador de VHDL traduce el código de la figura genera un circuito sumador para implementar el operador +. Cuando se emplea el sistema CAD Quartus II, el sumador que ocupa el compilador es en realidad el módulo *lpm_add_sub* mostrado en la figura 5.20. El compilador automáticamente establece los parámetros para el módulo, de modo que representa un sumador de 16 bits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( X, Y  : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figura 5.27 Código de VHDL para un sumador de 16 bits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S                : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0'& X) + Y + Cin ;
    S<= Sum(15 DOWNTO 0) ;
    Cout<= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figura 5.28 Sumador de 16 bits de la figura 5.27 con señales de acarreo y desbordamiento.

El código de la figura 5.27 no incluye señales de salida o entrada de acarreo. Además, no ofrece la señal de desbordamiento aritmético. En la figura 5.28 se presenta un modo en que es posible agregar estas señales. La señal de 17 bits llamada *Sum* se define en la arquitectura. El bit adicional, *Sum*(16), se usa para la salida de acarreo de la posición de bit 15 en el sumador. La instrucción empleada para asignar la suma de *X*, *Y* y la entrada de acarreo, *Cin*, a la señal *Sum* recurre a una sintaxis inusual. El significado del término entre paréntesis, ('0' & X), es que un 0 se concatena a la señal de 16 bits *X* para crear una señal de 17 bits. En VHDL, & se llama operador de *concatenación*. El lector no debe confundir este significado de & con el más tradicional de otros lenguajes de descripción de hardware donde es el operador lógico AND. La razón por la que el operador de concatenación se necesita en la figura 5.28 es que VHDL requiere que al menos uno de los operandos de una expresión aritmética tenga el mismo número de bits que el resultado. Puesto que *Sum* es un operando de 17 bits, entonces por lo menos *X* o *Y* deben modificarse para convertirse en un número de 17 bits.

Otro detalle que ha de observarse en la figura es la instrucción

```
S  <=  Sum(15 DOWNTO 0) ;
```

Que asigna los 16 bits menos significativos de *Sum* a la suma de salida *S*. La instrucción siguiente asigna la salida de acarreo de la suma, *Sum*(16), a la señal de salida de acarreo, *Cout*. La expresión para el desbordamiento aritmético se definió en la sección 5.3.5 como $c_{n-1} \oplus c_n$. En este caso, c_n corresponde a *Sum*(16), pero no hay forma directa de acceder a c_{n-1}, que es la salida de acarreo de la posición de bit 14. El lector debe comprobar que la expresión $X(15) \oplus Y(15) \oplus Sum(15)$ corresponde a c_{n-1}.

Dijimos que el compilador de VHDL puede generar un circuito sumador para implementar el operador +, y que el sistema Quartus II en realidad usa el módulo *lpm_add_sub* para ello.

Para redondear el tema también debemos mencionar que el módulo *lpm_add_sub* puede instanciarse directamente en código de VHDL, de forma similar a como se instanció el componente *fulladd* en la figura 5.23. En la sección A.6 del Apéndice A se da un ejemplo.

El código de la figura 5.28 usa el paquete *std_logic_signed* para permitir que las señales STD_LOGIC se empleen con operadores aritméticos. El paquete *std_logic_signed* en realidad utiliza otro paquete, *std_logic_arith*, que define dos tipos de datos: SIGNED y UNSIGNED, para usarlos en circuitos aritméticos que emplean números con y sin signo. Estos tipos de datos son los mismos que el tipo STD_LOGIC_VECTOR; cada uno es un arreglo de señales STD_LOGIC. El código de la figura 5.28 puede escribirse para usar directamente el paquete *std_logic_arith* como se muestra en la figura 5.29. Las señales multibit *X*, *Y*, *S* y *Sum* tienen el tipo SIGNED. De otro modo, el código es idéntico al de la figura 5.28 y resulta en el mismo circuito.

Es una elección arbitraria si se usa el paquete *std_logic_signed* y las señales STD_LOGIC_VECTOR, como en la figura 5.28, o el paquete *std_logic_arith* y las señales SIGNED, como en la 5.29. Para usar números sin signo también hay dos opciones. Se puede utilizar el paquete *std_logic_unsigned* con señales STD_LOGIC_VECTOR o el paquete *std_logic_arith* con señales UNSIGNED. Para nuestro código de ejemplo de las figuras 5.28 y 5.29, se generaría el mismo circuito si se suponen números con o sin signo. Pero para números sin signo no debemos producir una salida *Overflow* por separado, ya que la salida de acarreo representa el desbordamiento aritmético para los números sin signo.

Antes de dejar la explicación de las instrucciones aritméticas en VHDL debemos mencionar otra señal de tipo de datos que sirve para la aritmética. La instrucción siguiente define la señal *X*

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
    PORT ( Cin              : IN   STD_LOGIC ;
           X, Y             : IN   SIGNED(15 DOWNTO 0) ;
           S                : OUT  SIGNED(15 DOWNTO 0) ;
           Cout, Overflow   : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16);
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Figura 5.29 Uso del paquete aritmético.

```
ENTITY adder16 IS
    PORT ( X, Y  : IN   INTEGER RANGE -32768 TO 32767 ;
           S     : OUT  INTEGER RANGE -32768 TO 32767 ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

Figura 5.30 El sumador de 16 bits de la figura 5.27 con señales INTEGER.

como un entero (INTEGER)

```
SIGNAL X : INTEGER RANGE −32768 TO 32767
```

Para un objeto de datos INTEGER, el número de bits no se especifica explícitamente. En vez de ello, se especifica el *intervalo* de números que se representarán. Para un entero con signo de 16 bits, el intervalo de números representables es −32768 a 32767. Un ejemplo del uso del tipo de datos INTEGER en el código correspondiente a la figura 5.27 se muestra en la figura 5.30. Ahí no aparecen cláusulas LIBRARY o USE, porque el tipo INTEGER está predefinido en el VHDL estándar. Aunque el código de la figura es directo, es más difícil modificarlo para incluir las señales de acarreo y la salida de desbordamiento que se muestran en las figuras 5.28 y 5.29. El método que usamos, en el que los bits provenientes de la señal *Sum* sirvieron para definir las señales de salida de acarreo y de desbordamiento aritmético, no puede emplearse para objetos INTEGER.

5.6 MULTIPLICACIÓN

Antes de abordar el tema general de la multiplicación, cabe señalar que un número binario, B, puede multiplicarse por 2 simplemente agregando un cero a la derecha de su bit menos significativo. Esto efectivamente mueve todos los bits de B a la izquierda, y se dice que B se corre a la izquierda una posición de bit. Por ende, si $B = b_{n-1}b_{n-2}\cdots b_1b_0$, entonces $2 \times B = b_{n-1}b_{n-2}\cdots b_1b_00$. (En la sección 5.2.3 ya empleamos este hecho.) De manera similar, un número se multiplica por 2^k corriéndolo a la izquierda k posiciones de bit. Esto es cierto tanto para números con signo como para los que no lo llevan.

También habremos de considerar lo que ocurre si un número binario se corre a la derecha k posiciones de bit. De acuerdo con la representación numérica posicional, esta acción divide el número entre 2^k. Para números sin signo, el corrimiento explica la adición de k ceros a la izquierda del bit más significativo. Por ejemplo, si B es un número sin signo, entonces $B \div 2 = 0b_{n-1}b_{n-2}\cdots b_2b_1$. Nótese que el bit b_0 se pierde cuando se corre a la derecha. En números con signo es preciso conservar éste, lo cual se hace corriendo los bits a la derecha y llenando desde la izquierda con el valor del bit del signo. Por tanto, si B es un número con signo, entonces $B \div 2 = b_{n-1}b_{n-1}b_{n-2}\cdots b_2b_1$. Por ejemplo, si $B = 011000 = (24)_{10}$ entonces; $B \div 2 = 001100 = (12)_{10}$ y $B \div 4 = 000110 = (6)_{10}$. De manera similar, si $B = 101000 = -(24)_{10}$,

entonces $B \div 2 = 110100 = -(12)_{10}$ y $B \div 4 = 111010 = -(6)_{10}$. El lector también debe observar que cuanto más pequeño sea el número positivo, más 0 quedarán a la izquierda del primer 1, mientras que para los números negativos habrá más 1 a la izquierda del primer 0.

Ahora podemos centrar la atención en la tarea general de la multiplicación. Dos números binarios pueden multiplicarse mediante el mismo método empleado para los números decimales. Limitaremos la explicación a la multiplicación de números sin signo. En la figura 5.31*a* se muestra cómo se realiza la multiplicación manualmente, con números de cuatro bits. Cada bit multiplicador se examina de derecha a izquierda. Si un bit es igual a 1, se agrega una versión apropiadamente corrida del multiplicando para formar un *producto parcial*. Si el bit multiplicador es igual a 0, entonces nada se suma. La suma de todas las versiones corridas del multiplicando es el producto deseado. Nótese que éste ocupa ocho bits.

El mismo esquema sirve para diseñar un circuito multiplicador. En este ejemplo conservaremos los números de cuatro bits para mantener simple el análisis. Sean $M = m_3m_2m_1m_0$, $Q = q_3q_2q_1q_0$ y $P = p_7p_6p_5p_4p_3p_2p_1p_0$ el multiplicando, el multiplicador y el producto, respectivamente. Una forma simple de implementar el esquema de la multiplicación es usar un enfoque secuencial,

Multiplicando M	(14)	1 1 1 0
Multiplicador Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Producto P	(154)	1 0 0 1 1 0 1 0

a) Multiplicación a mano

Multiplicando M	(11)	1 1 1 0
Multiplicador Q	(14)	× 1 0 1 1
Producto parcial 0		1 1 1 0
		+ 1 1 1 0
Producto parcial 1		1 0 1 0 1
		+ 0 0 0 0
Producto parcial 2		0 1 0 1 0
		+ 1 1 1 0
Producto P	(154)	1 0 0 1 1 0 1 0

b) Multiplicación para implementación en hardware

Figura 5.31 Multiplicación de números sin signo.

donde un sumador de ocho bits se utilice para calcular productos parciales. Como primer paso, se examina el bit q_0. Si $q_0 = 1$, entonces M se suma al producto parcial inicial, que se inicializa en 0. Si $q_0 = 0$, entonces se suma 0 al producto parcial. A continuación se examina q_1. Si $q_1 = 1$, entonces se suma el valor $2 \times M$ al producto parcial. El valor $2 \times M$ se crea con sólo correr M una posición de bit a la izquierda. De manera similar, $4 \times M$ se suma al producto parcial si $q_2 = 1$, y $8 \times M$ se suma si $q_3 = 1$. En el capítulo 10 demostraremos cómo implementar tal circuito.

Este enfoque secuencial conduce a un circuito relativamente lento, sobre todo porque usa un solo sumador de ocho bits para realizar las sumas necesarias para generar los productos parciales y final. Es posible obtener un circuito mucho más rápido si se emplean varios sumadores para calcular los productos parciales.

5.6.1 ARREGLO MULTIPLICADOR PARA NÚMEROS SIN SIGNO

En la figura 5.31*b* se indica cómo llevar a cabo una multiplicación usando varios sumadores. En cada paso se utiliza un sumador de cuatro bits para calcular el nuevo producto parcial. Nótese que a medida que el cálculo avanza, los bits menos significativos no se afectan por las sumas subsecuentes; por tanto, se pueden pasar directamente al producto final, como se indica mediante las flechas gris oscuro. Desde luego, dichos bits también son parte de los productos parciales.

Es posible diseñar un circuito multiplicador más rápido con un arreglo estructural de organización semejante al de la figura 5.31*b*. Considérese un ejemplo de 4×4, donde el multiplicando y el multiplicador son $M = m_3m_2m_1m_0$ y $Q = q_3q_2q_1q_0$, respectivamente. El producto parcial 0, $PP0 = pp0_3\ pp0_2\ pp0_1\ pp0_0$, puede generarse usando la AND de q_0 con cada bit de M. Por ende

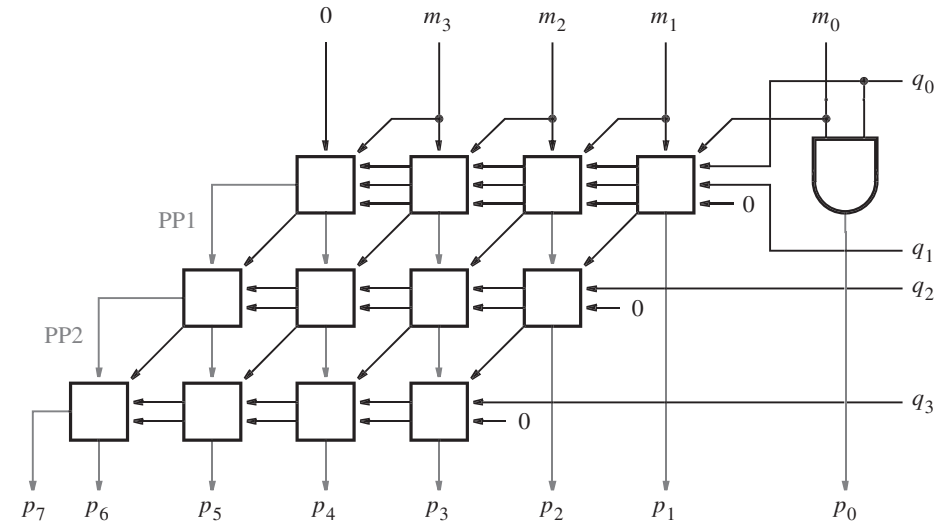
$$PP0 = m_3q_0\ \ m_2q_0\ \ m_1q_0\ \ m_0q_0$$

El producto parcial 1, $PP1$, se genera usando la AND de q_1 con M y sumándola a $PP0$ como sigue

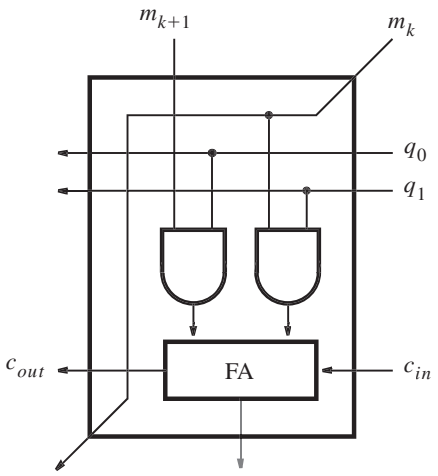
$PP0$:		0	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$
	+	m_3q_1	m_2q_1	m_1q_1	m_0q_1	0
$PP1$:		$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$

De manera similar, el producto parcial 2, $PP2$, se genera empleando la AND de q_2 con M y sumándola a $PP1$, etcétera.

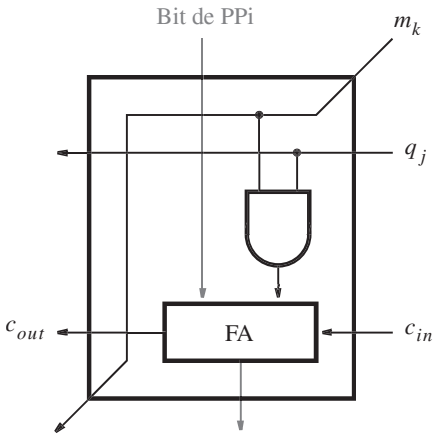
Un circuito que implemente las operaciones anteriores se organiza en un arreglo, como se muestra en la figura 5.32*a*. Hay dos tipos de bloques en el arreglo. En el inciso (*b*) de la figura se presentan los detalles de los bloques de la fila superior, y en el inciso (*c*) se muestra el bloque utilizado en la segunda y tercera filas. Obsérvese que las versiones corridas del multiplicando se ofrecen al enrutar las señales m_k diagonalmente de un bloque a otro. El sumador completo incluido en cada bloque implementa un sumador con acarreo en cascada para generar cada producto parcial. Es posible diseñar multiplicadores incluso más veloces utilizando otros tipos de sumadores [1].



a) Estructura del circuito



b) Bloque de la fila superior



c) Bloque de las dos filas inferiores

Figura 5.32 Circuito multiplicador de 4 × 4.

5.6.2 MULTIPLICACIÓN DE NÚMEROS CON SIGNO

La multiplicación de números sin signo ilustra los aspectos centrales relativos al diseño de circuitos multiplicadores. La multiplicación de números con signo es un tanto más compleja.

Si el operando multiplicador es positivo, es posible usar esencialmente el mismo esquema que para los números sin signo. Por cada bit del operando multiplicador que sea igual a 1 hay

que sumar al producto parcial una versión adecuadamente corrida del multiplicando. El multiplicando puede ser o positivo o negativo.

Puesto que las versiones corridas del multiplicando se suman a los productos parciales es importante garantizar que los números implicados estén representados de manera correcta. Por ejemplo, si los dos bits más a la derecha del multiplicador son iguales a 1, entonces la primera suma debe producir el producto parcial $PP1 = M + 2M$, donde M es el multiplicando. Si $M = m_{n-1}m_{n-2}\cdots m_1m_0$, entonces $PP1 = m_{n-1}m_{n-2}\cdots m_1m_0 + m_{n-1}m_{n-2}\cdots m_1m_00$. El sumador que realiza esta suma comprende circuitos que suman dos operandos de igual longitud. Puesto que el corrimiento del multiplicando a la izquierda para generar $2M$ resulta en que uno de los operandos tenga $n + 1$ bits, la suma requerida debe llevarse a cabo usando el segundo operando, M, representado también como un número de $(n + 1)$ bits. Un número con signo de n bits se representa como un número de $(n + 1)$ bits replicando el bit del signo como el nuevo bit más a la izquierda. Por tanto, $M = m_{n-1}m_{n-2}\cdots m_1m_0$ se representa empleando $(n + 1)$ bits como $M = m_{n-1}m_{n-1}m_{n-2}\cdots m_1m_0$. El valor de un número positivo no cambia si se añaden 0 como los bits más significativos; el valor de un número negativo no cambia si se añaden 1 como los bits más significativos. Tal replicación del bit del signo se llama *extensión de signo*.

Cuando a un producto parcial se suma una versión corrida del multiplicando hay que evitar el desbordamiento. Por consiguiente, el nuevo producto parcial debe ser más grande en un bit. En la figura 5.33*a* se ilustra el proceso de multiplicar dos números positivos. Los bits de signo extendido se muestran en gris oscuro. En el inciso (*b*) de la figura se supone un multiplicando negativo. Nótese que el producto resultante tiene $2n$ bits en ambos casos.

Para un operando multiplicador negativo es posible convertir tanto el multiplicador como el multiplicando en sus complementos a 2 porque ello no cambiará el valor del resultado. Entonces puede usarse el esquema para un multiplicador positivo.

Hemos presentado un esquema hasta cierto punto simple para multiplicar números con signo. Hay otras técnicas más eficientes, pero también más complejas. No las seguiremos, pero el lector interesado puede consultar la referencia [1].

Ya expusimos los circuitos que realizan suma, resta y multiplicación. Otra operación aritmética necesaria en los sistemas de cómputo es la división. Los circuitos que efectúan divisiones son más complejos; en el capítulo 10 presentaremos un ejemplo. En los libros que tratan acerca de la organización de las computadoras se analizan varias técnicas para realizar divisiones, y pueden encontrarse en las referencias [1, 2].

5.7 OTRAS REPRESENTACIONES NUMÉRICAS

En las secciones previas empleamos enteros binarios denotados en representación numérica posicional. En los sistemas digitales también se usan otros tipos de números. En esta sección expondremos brevemente otros tres tipos: punto fijo, punto flotante y números decimales codificados en binario.

5.7.1 NÚMEROS CON PUNTO FIJO

Un número con *punto fijo* consta de partes entera y fraccionaria. Puede escribirse en la representación numérica posicional como

Multiplicando M	(+14)	0 1 1 1 0
Multiplicador Q	(+11)	× 0 1 0 1 1
Producto parcial 0		0 0 0 1 1 1 0
		+ 0 0 1 1 1 0
Producto parcial 1		0 0 1 0 1 0 1
		+ 0 0 0 0 0 0
Producto parcial 2		0 0 0 1 0 1 0
		+ 0 0 1 1 1 0
Producto parcial 3		0 0 1 0 0 1 1
		+ 0 0 0 0 0 0
Producto P	(+154)	0 0 1 0 0 1 1 0 1 0

a) Multiplicando positivo

Multiplicando M	(−14)	1 0 0 1 0
Multiplicador Q	(+11)	× 0 1 0 1 1
Producto parcial 0		1 1 1 0 0 1 0
		+ 1 1 0 0 1 0
Producto parcial 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0
Producto parcial 2		1 1 1 0 1 0 1
		+ 1 1 0 0 1 0
Producto parcial 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0
Producto P	(−154)	1 1 0 1 1 0 0 1 1 0

b) Multiplicando negativo

Figura 5.33 Multiplicación de números con signo.

$$B = b_{n-1}b_{n-2}\cdots b_1b_0.b_{-1}b_{-2}\cdots b_{-k}$$

El valor del número es

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

La posición del punto de la base (raíz) se supone fija; de ahí el nombre de *número con punto fijo*. Si el punto de la base no se muestra, entonces se asume que está a la derecha del dígito menos significativo, lo que indica que el número es un entero.

Los circuitos lógicos que tratan con números con punto fijo son, en esencia, los mismos que los utilizados para los enteros. No los explicaremos por separado.

5.7.2 NÚMEROS CON PUNTO FLOTANTE

Los números con punto fijo tienen un intervalo señalado por los dígitos significativos usados para representar el número. Por ejemplo, si se utilizan ocho dígitos y un signo para representar enteros decimales, entonces el intervalo de valores que pueden representarse va de 0 a ± 99999999. Si se emplean ocho dígitos para representar una fracción, entonces el intervalo representable va de 0.00000001 a ± 0.99999999. En aplicaciones científicas suele ser necesario lidiar con números muy grandes o muy pequeños. En vez de usar la representación con punto fijo, que requeriría muchos dígitos significativos, es mejor utilizar la representación con punto flotante en la que los números se representan mediante una *mantisa* que comprende los dígitos significativos y un *exponente* de la base *R*. El formato es

$$Mantisa \times R^{Exponente}$$

Con frecuencia, los números se *normalizan*, de modo que el punto de la base se coloca a la derecha del primer dígito distinto de cero, como en 5.234×10^{43} o 6.31×10^{-28}.

La representación binaria con punto flotante la norma el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, por sus siglas en inglés) [3]. En la norma se especifican dos tamaños de formatos: un formato de 32 bits de *precisión sencilla* y un formato de 64 bits de *doble precisión*. Ambos formatos se ilustran en la figura 5.34.

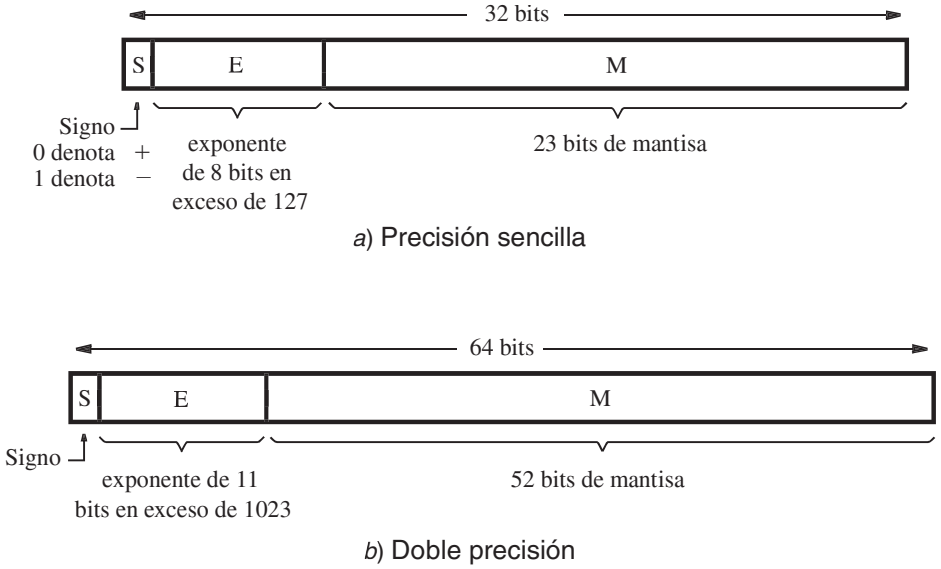


Figura 5.34 Normas del IEEE para formatos con punto flotante.

Formato con punto flotante de precisión sencilla

En la figura 5.34*a* se describe el formato de precisión sencilla. El bit más a la izquierda es el del signo: 0 para números positivos y 1 para negativos. Existe un campo exponente, *E*, de 8 bits, y un campo mantisa, *M*, de 23 bits. El exponente es respecto a la base 2. Puesto que es necesario tener capacidad para representar números muy grandes y muy pequeños, el exponente puede ser positivo o negativo. En vez de simplemente usar un número con signo de 8 bits como exponente, lo que permitiría valores de exponente en el intervalo de –128 a 127, la norma del IEEE especifica el exponente en formato de *exceso de 127*. En este formato, el valor 127 se suma al valor del exponente real de modo que

$$Exponente = E - 127$$

De esta forma, *E* se convierte en un entero positivo. Este formato es práctico para sumar y restar números con punto flotante, ya que el primer paso en estas operaciones supone la comparación de los exponentes para determinar si la mantisa debe correrse adecuadamente para sumar/restar los bits significativos. El intervalo de *E* va de 0 a 255. Los valores extremo de $E = 0$ y $E = 255$ se toman para denotar el cero exacto y el infinito, respectivamente. Por tanto, el intervalo normal del exponente es -126 a 127, que se representa mediante los valores de *E* de 1 a 254.

La mantisa se representa mediante 23 bits. La norma del IEEE pide una mantisa normalizada, lo que implica que el bit más significativo siempre es igual a 1. Por ende, no es necesario incluir este bit explícitamente en el campo mantisa. En consecuencia, si *M* es el vector bit en el campo mantisa, el valor real de la mantisa es 1.*M*, lo que produce una mantisa de 24 bits. Consecuentemente, el formato de punto flotante de la figura 5.34*a* representa el número

$$\text{Valor} = \pm 1.M \times 2^{E-127}$$

El tamaño del campo mantisa permite la representación de números que tienen la precisión de más o menos siete dígitos decimales. El intervalo del campo exponente, de 2^{-126} a 2^{127}, corresponde aproximadamente a $10^{\pm 38}$.

Formato con punto flotante de doble precisión

En la figura 5.34*b* se muestra el formato de doble precisión, que usa 64 bits. Los campos exponente y mantisa son más grandes. Este formato permite un mayor intervalo y precisión de números. El campo exponente tiene 11 bits y especifica el exponente en formato de *exceso de 1023*, donde

$$Exponente = E - 1023$$

El intervalo de *E* va de 0 a 2047, pero de nuevo los valores $E = 0$ y $E = 2047$ se usan para indicar el 0 exacto y el infinito, respectivamente. Por tanto, el intervalo normal del exponente va de -1022 a 1023, lo que se representa con los valores de *E* de 1 a 2046.

El campo mantisa tiene 52 bits. Como se supone que la mantisa está normalizada, su valor real de nuevo es 1.*M*. Por ello el valor de un número con punto flotante es

$$\text{Valor} = \pm 1.M \times 2^{E-1023}$$

Este formato permite representar números que tienen la precisión de alrededor de 16 dígitos decimales y el intervalo de aproximadamente $10^{\pm 308}$.

Las operaciones aritméticas que usan operandos con punto flotante son mucho más complejas que las operaciones con enteros con signo. Puesto que éste es un terreno más bien especializado, no se ahondará en torno al diseño de circuitos lógicos que puedan realizar tales operaciones. Para una explicación más completa de las operaciones con punto flotante, el lector puede consultar las referencias [1, 2].

5.7.3 REPRESENTACIÓN DECIMAL CODIFICADO EN BINARIO

En los sistemas digitales es posible representar números decimales simplemente codificando cada dígito en forma binaria. Esto se denomina representación *decimal codificado en binario* (*BCD, binary-coded-decimal*). Puesto que existen 10 dígitos para codificar, es preciso usar cuatro bits por cada uno de ellos. Cada dígito se codifica mediante el patrón binario que representa su valor sin signo, como se muestra en la tabla 5.3. Nótese que en BCD sólo se usan 10 de los 16 patrones disponibles, lo que significa que los restantes seis no deben ocurrir en circuitos lógicos que funcionen con operandos BCD; tales patrones usualmente se tratan como condiciones "no importa" en el proceso de diseño. La representación BCD se empleó en algunas de las primeras computadoras, así como en muchas calculadoras de mano. Su principal virtud radica en que ofrece un formato práctico cuando hay que mostrar información numérica en una pantalla orientada a dígitos. Sus inconvenientes son la complejidad de los circuitos que realizan las operaciones aritméticas y el hecho de que se desperdician seis de los posibles patrones de código.

Aun cuando la importancia de la representación BCD ha disminuido, todavía se le encuentra. Para brindar al lector una pista de la complejidad de los circuitos requeridos estudiaremos la suma BCD con cierto detalle.

Suma BCD

La suma de dos dígitos BCD se complica por el hecho de que la suma puede exceder 9, en cuyo caso se tendrá que hacer una corrección. Sean $X = x_3x_2x_1x_0$ y $Y = y_3y_2y_1y_0$ las representaciones de dos dígitos BCD, y sea $S = s_3s_2s_1s_0$ el dígito suma deseado, $S = X + Y$. Obviamente, si $X + Y \leq 9$, entonces la suma es la misma que la suma de dos números binarios sin signo

Tabla 5.3 Dígitos decimales codificados en binario.

Dígito decimal	Código BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

de cuatro bits. Pero si $X + Y > 9$, entonces el resultado requiere dos dígitos BCD. Más aún, la suma de cuatro bits obtenida del sumador de cuatro bits puede ser incorrecta.

En algunos casos es preciso hacer alguna corrección: cuando la suma es mayor que 9 pero no se genera salida de acarreo usando cuatro bits, y cuando la suma es mayor que 15 de modo que se genera salida de acarreo empleando cuatro bits. En la figura 5.35 se ilustran estos casos. En el primero, la suma de cuatro bits produce $7 + 5 = 12 = Z$. Para obtener un resultado BCD correcto hay que generar $S = 2$ y un acarreo de 1. La corrección necesaria es evidente a partir del hecho de que la suma de cuatro bits es un esquema de módulo 16, mientras que la suma decimal es un esquema de módulo 10. Por tanto, puede generarse un dígito decimal correcto sumando 6 al resultado de la suma de cuatro bits siempre que este resultado supere 9. En consecuencia, podemos arreglar el cálculo como sigue

$$Z = X + Y$$

Si $Z \leq 9$, entonces $S = Z$ y salida de acarreo $= 0$

Si $Z > 9$, entonces $X = Z + 6$ y salida de acarreo $= 1$

El segundo ejemplo de la figura 5.35 muestra lo que ocurre cuando $X + Y > 15$. En este caso, los cuatro bits menos significativos de Z representan el dígito 1, lo cual es erróneo. Pero se genera un acarreo, que corresponde al valor 16, que debe tomarse en cuenta. De nuevo, si se suma 6 a la suma intermedia Z se produce la corrección necesaria.

En la figura 5.36 se presenta un diagrama de bloques de un sumador BCD de un dígito basado en este esquema. El bloque que detecta si $Z > 9$ produce una señal de salida, *Adjust* (ajuste), que controla al multiplexor que proporciona la corrección cuando es necesario. Un segundo sumador de cuatro bits genera los bits suma corregidos. Si *Adjust* $= 0$, entonces $S = Z + 0$; si *Adjust* $= 1$, entonces $S = Z + 6$ y salida de acarreo $= 1$.

```
  X      0 1 1 1      7
+ Y    + 0 1 0 1    + 5
-----  -----------  -----
  Z      1 1 0 0     12
       + 0 1 1 0
       -----------
acarreo 1 0 0 1 0
        \_______/
          S = 2

  X      1 0 0 0      8
+ Y    + 1 0 0 1    + 9
-----  -----------  -----
  Z      1 0 0 0 1   17
       + 0 1 1 0
       -----------
acarreo 1 0 1 1 1
        \_______/
          S = 7
```

Figura 5.35 Suma de dígitos BCD.

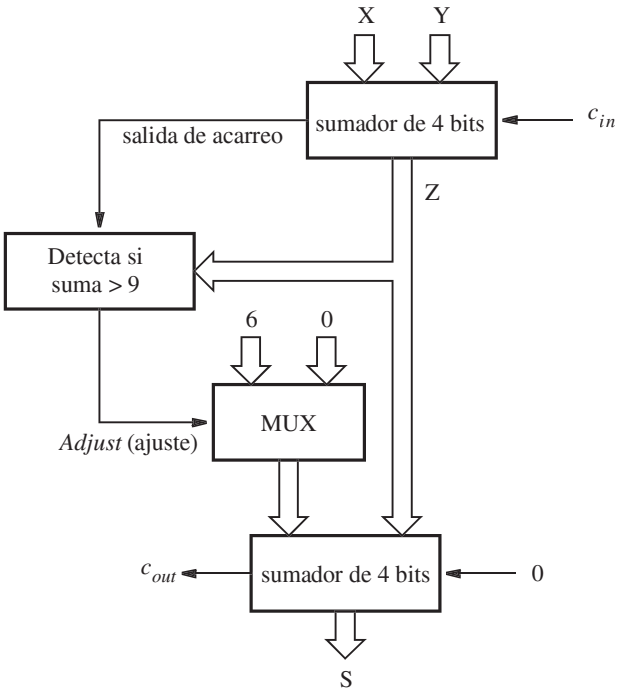


Figura 5.36 Diagrama de bloques para un sumador BCD de un dígito.

En la figura 5.37 se muestra una implementación de este diagrama de bloques usando código de VHDL. Las entradas *X* y *Y* se definen como números de cuatro bits. La salida suma, *S*, se define como un número de cinco bits, lo que permite que la salida de acarreo aparezca en el bit S_4, en tanto que la suma se produce en los bits S_{3-0}. La suma intermedia *Z* también se define como un número de cinco bits. Recuérdese de lo expuesto en la sección 5.5.4 que VHDL requiere que al menos uno de los operandos de una operación aritmética tenga el mismo número de bits que el resultado. Este requisito explica por qué es preciso concatenar un 0 a la entrada X en la expresión Z <= ('0' & X) + Y.

La instrucción

```
<= '1' WHEN Z > 9 ELSE '0'
```

usa un tipo de instrucción de asignación de señal de VHDL que no hemos visto. Se llama *asignación de señal seleccionada* y se usa para asignar uno de múltiples valores a una señal, con base en cierto criterio. En este caso el criterio es la condición $Z > 9$. Si esta condición se satisface, la instrucción asigna 1 a *Adjust*; de otro modo, le asigna 0. Otros ejemplos de la asignación de señal seleccionada se brindan en el capítulo 6.

También cabe notar que incluimos la señal *Adjust* en el código de VHDL sólo para que fuera consistente con la figura 5.36. Pudimos haberla eliminado fácilmente y escribir la expresión como

```
S <= Z WHEN Z < 10 ELSE Z + 6
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCD IS
    PORT ( X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(4 DOWNTO 0) ) ;
END BCD ;

ARCHITECTURE Behavior OF BCD IS
    SIGNAL Z : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL Adjust : STD_LOGIC ;
BEGIN
    Z <= ('0' & X) + Y ;
    Adjust <= '1' WHEN Z > 9 ELSE '0' ;
    S <= Z WHEN (Adjust = '0') ELSE Z + 6 ;
END Behavior
```

Figura 5.37 Código de VHDL para un sumador BCD de un dígito.

Para comprobar la exactitud funcional del código se realiza una simulación funcional. En la figura 5.38 se ofrece un ejemplo de los resultados obtenidos.

Si se desea derivar un circuito para implementar el diagrama de bloques de la figura 5.36 a mano, en vez de usar VHDL, entonces podemos emplear el enfoque siguiente. Para definir la función *Adjust*, obsérvese que la suma intermedia superará 9 si la salida de acarreo proveniente del sumador de cuatro bits es igual a 1, o si $z_3 = 1$ y z_2 o z_1 (o ambos) son iguales a 1. Por tanto, la expresión lógica para esta función es

$$Adjust = \text{salida de carreo} + z_3(z_2 + z_1)$$

En lugar de implementar otro sumador completo de cuatro bits para realizar la corrección, podemos usar un circuito más simple porque la suma de una constante 6 no requiere la capacidad completa de un sumador de cuatro bits. Nótese que el bit menos significativo de la suma, s_0, no se afecta en absoluto; por tanto, $s_0 = z_0$. Se puede usar un sumador de dos bits para desarrollar los bits s_2 y s_1. El bit s_3 es el mismo que z_3 si la salida de acarreo proveniente del sumador de dos bits es 0, y es igual a $\overline{z}_3$ si esta salida de acarreo es igual a 1. En la figura 5.39 se muestra

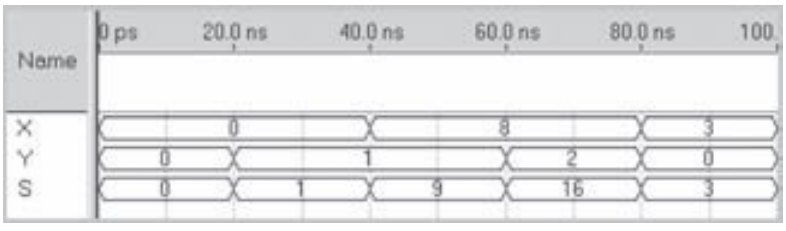


Figura 5.38 Simulación funcional del código de VHDL de la figura 5.37.

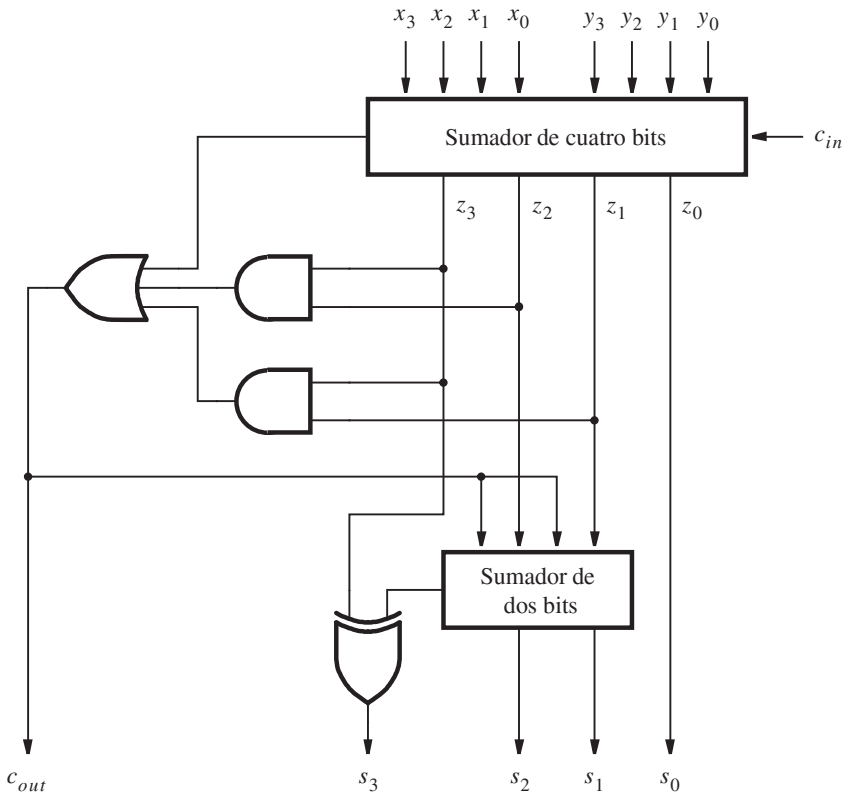


Figura 5.39 Circuito para un sumador BCD de un dígito.

un circuito completo que implementa este esquema. Al usar el sumador BCD de un dígito como bloque básico es posible construir sumadores BCD más grandes de la misma forma en que se utiliza un sumador completo binario para construir sumadores binarios con acarreo en cascada más grandes.

La resta de números BCD puede manejarse con el enfoque de complemento a la base. Igual que empleamos la representación en complemento a 2 para manejar los números binarios negativos, podemos usar la representación en complemento a 10 para manejar los decimales. El desarrollo de tal esquema se deja como ejercicio al lector (véase el problema 5.19).

5.8 CÓDIGO DE CARACTERES ASCII

El código más popular para representar información en sistemas digitales se usa para letras y para números, así como para algunos caracteres de control. Se conoce como *código ASCII*, que significa American Standard Code for Information Interchange (Código Americano Estándar para el Intercambio de Información). En la tabla 5.4 se presenta el código especificado por este estándar.

Tabla 5.4 Código ASCII de siete bits.

Posiciones de bit	Posiciones de bit 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	ESPACIO	0	@	P	′	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	°	DEL

NUL	Nulo/Espera	SI	Salir corrimiento
SOH	Comienzo de encabezado	DLE	Ingresar corrimiento
STX	Comienzo de texto	DC1-DC4	Escape de liga de datos
ETX	Fin de texto	NAK	Control de dispositivo
EOT	Fin de transmisión	SYN	Confirmación negativa
ENQ	Petición	ETB	Fin de bloque transmitido Espera síncrona
ACQ	Confirmación	CAN	Cancelar (error en datos)
BEL	Señal audible	EM	Fin de medio
BS	Retroceso	SUB	Secuencia especial
HT	Horizontal tab	ESC	Escape
LF	Tabulador horizontal	FS	Separador de archivo
VT	Salto de línea	GS	Separador de grupo
FF	Tabulador vertical	RS	Separador de registro
CR	Avance de página	US	Separador de unidad
SO	Retorno de carro	DEL	Borrar/Espera

Posiciones de bit de formato de código = 6 5 4 3 2 1 0

El código ASCII usa patrones de siete bits para denotar 128 caracteres. Diez de ellos son dígitos decimales de 0 a 9. Nótese que los bits de orden superior tienen el mismo patrón, $b_6b_5b_4 = 011$, para los 10 dígitos. Cada dígito se identifica mediante los cuatro bits de orden inferior, b_{3-0}, que usan los patrones binarios para esos dígitos. Las letras mayúsculas y minúsculas se codifican de una manera que facilita el ordenamiento textual de la información. Los códigos para la A a la Z están en secuencia numérica ascendente, lo que significa que la tarea de ordenar letras (o palabras) se logra mediante una simple comparación aritmética de los códigos que las representan.

Los caracteres que son letras del alfabeto o números se conocen como caracteres *alfanuméricos*. Además de ellos, en el código ASCII se incluyen signos de puntuación como ! y ?; signos que se usan comúnmente, como & y %; y un juego de caracteres de control. Los caracteres de control son los que se necesitan en los sistemas de cómputo para manejar y transferir datos entre dispositivos. Por ejemplo, el carácter de retorno de carro, que se abrevia CR en la tabla, indica que el carro, o posición del cursor, de un dispositivo de salida, digamos una impresora o un monitor, debe regresar a la columna del extremo izquierdo.

El código ASCII se usa para codificar información que se maneja como texto. No es práctico para la representación de números que se emplean como operandos en operaciones aritméticas. Para este propósito es mejor convertir los números codificados en ASCII en una representación binaria, como explicamos antes.

El estándar ASCII utiliza siete bits para codificar un carácter. En los sistemas de cómputo, un tamaño más natural es de ocho bits, o un byte. Hay dos formas comunes de encajar un carácter codificado en ASCII en un byte. Una es establecer el octavo bit, b_7, en 0. Otra consiste en usar ese bit para indicar la paridad de los otros siete bits, lo que significa mostrar si el número de 1 en el código de siete bits es par o impar.

Paridad

El concepto de *paridad* se usa mucho en los sistemas digitales con propósitos de comprobación de errores. Cuando la información digital se transmite de un punto a otro, quizá a través de largos cables, es posible que algunos bits se corrompan durante la transmisión. Por ejemplo, el remitente puede transmitir un bit cuyo valor es igual a 1, pero el receptor observa un bit cuyo valor es 0. Supóngase que un elemento de datos consta de n bits. Es posible implementar un simple mecanismo de comprobación de errores incluyendo un bit adicional, p, que indica la paridad del elemento de n bits. Se pueden usar dos tipos de paridad. Para *paridad par*, se da al bit p el valor tal que el número total de 1 en los $n + 1$ bits transmitidos (que comprenden los datos de n bits y el bit de paridad p) sea par. Para *paridad impar*, se da al bit p el valor que hace que el número total de 1 sea impar. El remitente genera el bit p con base en el elemento de datos de n bits que se transmitirá. El receptor comprueba si la paridad del elemento recibido es correcta.

Los circuitos que generan y comprueban la paridad pueden realizarse con compuertas XOR. Por ejemplo, para un elemento de datos de cuatro bits que consta de los bits $x_3x_2x_1x_0$, el bit de paridad par puede generarse como

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

En el extremo receptor, la comprobación se realiza con

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

Si $c = 0$, entonces el elemento recibido muestra la paridad correcta. Si $c = 1$, entonces ocurrió un error. Nótese que observar $c = 0$ no es garantía de que el elemento recibido sea correcto. Si

dos o cualesquier número par de bits invierten sus valores durante la transmisión, la paridad del elemento de datos no cambiará; por tanto, el error no será detectado. Pero si se corrompe un número impar de bits, entonces sí se detectará.

El atractivo de la comprobación de paridad radica en su simpleza. Existen otros esquemas más refinados que proporcionan mecanismos de comprobación de errores más confiables [4]. En la sección 9.3 veremos de nuevo los circuitos de paridad.

5.9 EJEMPLOS DE PROBLEMAS RESUELTOS

En esta sección se presentan algunos problemas usuales que el lector puede encontrar, y se muestra cómo resolverlos.

Ejemplo 5.7 **Problema:** Convierta el número decimal 14959 en número hexadecimal.

Solución: Un entero se convierte en representación hexadecimal mediante divisiones sucesivas entre 16, de modo que en cada paso el residuo sea un dígito hexa. Para ver por qué es cierto esto, considere un número de cuatro dígitos $H = h_3h_2h_1h_0$. Su valor es

$$V = h_3 \times 16^3 + h_2 \times 16^2 + h_1 \times 16 + h_0$$

Si dividimos esto entre 16 se obtiene

$$\frac{V}{16} = h_3 \times 16^2 + h_2 \times 16 + h_1 + \frac{h_0}{16}$$

Por tanto, el residuo da h_0. En la figura 5.40 se muestran los pasos necesarios para realizar la conversión $(14959)_{10} = (3A6F)_{16}$.

Convertir $(14959)_{10}$

			Residuo	Dígito hexa	
$14959 \div 16$	=	934	15	F	LSB
$934 \div 16$	=	58	6	6	
$58 \div 16$	=	3	10	A	
$3 \div 16$	=	0	3	3	MSB

El resultado es $(3A6F)_{16}$

Figura 5.40 Conversión de decimal en hexadecimal.

Ejemplo 5.8

Problema: Convierta la fracción decimal 0.8254 en representación binaria.
Solución: Como se indicó en la sección 5.7.1, una fracción binaria se representa como el patrón de bit $B = 0.b_{-1}b_{-2}\cdots b_{-m}$ y su valor es

$$V = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

Al multiplicar esta expresión por 2 se produce

$$b_{-1} + b_{-2} \times 2^{-1} + \cdots + b_{-m} \times 2^{-(m-1)}$$

Aquí, el término del extremo izquierdo es el primer bit a la derecha del punto de la base. Los términos restantes constituyen otra fracción binaria que puede manipularse de la misma forma. Por tanto, para convertir una fracción decimal en binaria se multiplica el número decimal por 2 y el bit calculado se establece en 0 si el producto es menor que 1, y se pone en 1 si el producto es mayor o igual a 1. Este cálculo se repite hasta obtener un número suficiente de bits para satisfacer la precisión deseada. Note que puede no ser posible representar una fracción decimal con una fracción binaria que tenga exactamente el mismo valor. En la figura 5.41 se muestra el cálculo requerido que produce $(0.8254)_{10} = (0.11010011\ldots)_2$.

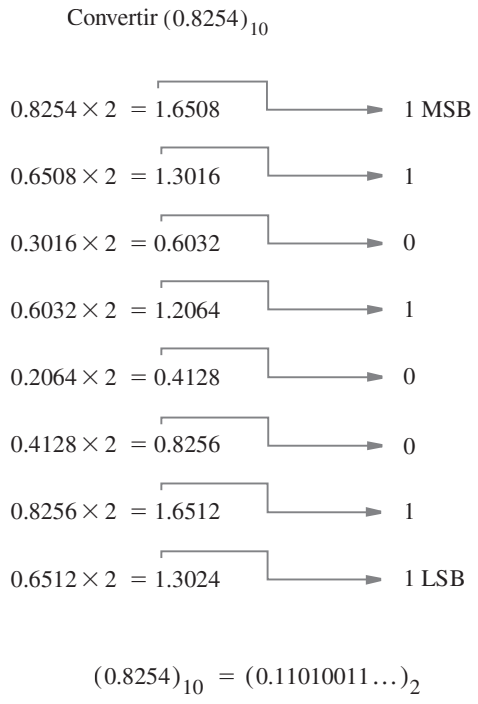


Figura 5.41 Conversión de fracciones de decimal a binario.

Ejemplo 5.9 **Problema:** Convierta el número decimal con punto fijo 214.45 en un número binario con punto fijo.

Solución: Para la parte entera se realizan divisiones sucesivas entre 2, como se ilustra en la figura 5.1. Para la parte fraccionaria se efectúan multiplicaciones sucesivas por 2, como se describe en el ejemplo 5.8. El cálculo completo se presenta en la figura 5.42, lo que produce $(214.45)_{10} = (11010110.0111001\ldots)_2$.

Ejemplo 5.10 **Problema:** En los cálculos en computadora con frecuencia es necesario comparar números. Dos números con signo de cuatro bits, $X = x_3x_2x_1x_0$ y $Y = y_3y_2y_1y_0$, pueden compararse mediante el circuito restador de la figura 5.43, que realiza la operación $X - Y$. Las tres salidas denotan lo siguiente:

- $Z = 1$ si el resultado es 0; de otro modo $Z = 0$
- $N = 1$ si el resultado es negativo; de otro modo $N = 0$
- $V = 1$ si ocurre desbordamiento aritmético; de otro modo $V = 0$

Muestre cómo pueden usarse Z, N y V para determinar los casos $X = Y$, $X < Y$, $X \leq Y$, $X > Y$, y $X \geq Y$.

Solución: Considere primero el caso $X <$ Y, donde pueden surgir las posibilidades siguientes:

- Si X y Y tienen el mismo signo no habrá desbordamiento; por tanto, $V = 0$. Entonces, para X y Y positivos y negativos la diferencia será negativa ($N = 1$).
- Si X es negativa y Y positiva, la diferencia será negativa ($N = 1$) si no hay desbordamiento ($V = 0$); pero el resultado será positivo ($N = 0$) si hay desbordamiento ($V = 1$).

En consecuencia, si $X < Y$, entonces $N \oplus V = 1$.

El caso $X = Y$ se detecta con $Z = 1$. Entonces, $X \leq Y$ se detecta mediante $Z + (N \oplus V) = 1$. Los dos últimos casos sólo son simples inversos: $X > Y$ si $Z + (N \oplus V) = 1$ y $X \geq Y$ si $\overline{N \oplus V} = 1$.

Ejemplo 5.11 **Problema:** Escriba el código de VHDL para especificar el circuito de la figura 5.43.

Solución: Es posible especificar el circuito mediante el enfoque estructural presentado en la figura 5.26, como se indica en la figura 5.44. Los cuatro sumadores completos se definen en un paquete en la figura 5.24.

Este enfoque se complica cuando se incluyen grandes circuitos, como sería el caso si el comparador tuviese operandos de 32 bits. Una posibilidad consiste en usar una especificación por comportamiento, como se muestra en la figura 5.45, que se basa en el esquema de la figura 5.28. Note que hemos establecido directamente que Y debe restarse de X, de modo que no tenemos que complementar Y de manera explícita. Como el compilador de VHDL implementará el circuito usando un módulo de biblioteca, es preciso especificar la señal desbordamiento, V, sólo en términos de los bits S, ya que las señales de acarreo entre etapas no son accesibles como se explicó para la figura 5.28.

Convertir $(214.45)_{10}$

$$\frac{214}{2} = 107 + \frac{0}{2} \rightarrow 0 \text{ LSB}$$

$$\frac{107}{2} = 53 + \frac{1}{2} \rightarrow 1$$

$$\frac{53}{2} = 26 + \frac{1}{2} \rightarrow 1$$

$$\frac{26}{2} = 13 + \frac{0}{2} \rightarrow 0$$

$$\frac{13}{2} = 6 + \frac{1}{2} \rightarrow 1$$

$$\frac{6}{2} = 3 + \frac{0}{2} \rightarrow 0$$

$$\frac{3}{2} = 1 + \frac{1}{2} \rightarrow 1$$

$$\frac{1}{2} = 0 + \frac{1}{2} \rightarrow 1 \text{ MSB}$$

$$0.45 \times 2 = 0.90 \rightarrow 0 \text{ MSB}$$

$$0.90 \times 2 = 1.80 \rightarrow 1$$

$$0.80 \times 2 = 1.60 \rightarrow 1$$

$$0.60 \times 2 = 1.20 \rightarrow 1$$

$$0.20 \times 2 = 0.40 \rightarrow 0$$

$$0.40 \times 2 = 0.80 \rightarrow 0$$

$$0.80 \times 2 = 1.60 \rightarrow 1 \text{ LSB}$$

$$(214.45)_{10} = (11010110.0111001\ldots)_2$$

Figura 5.42 Conversión de números de punto fijo de decimal a binario.

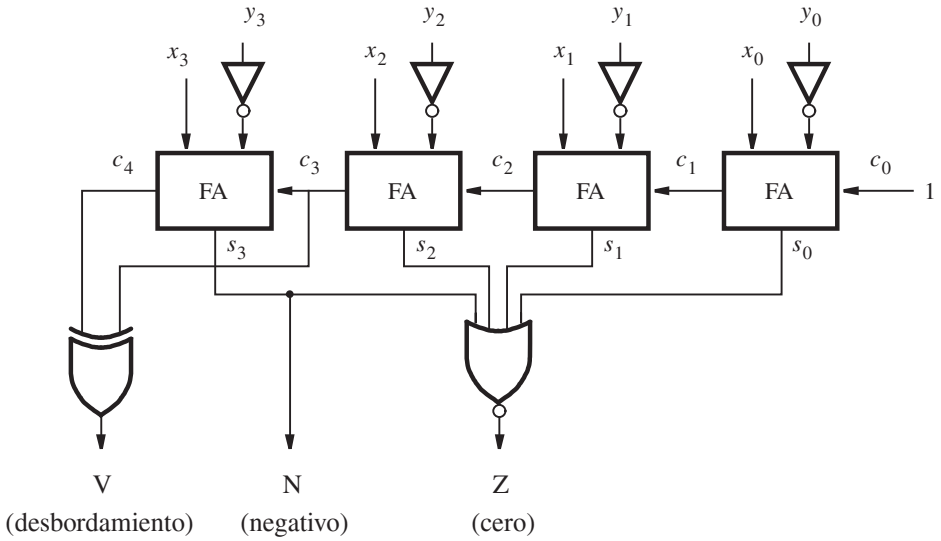


Figura 5.43 Circuito comparador.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY comparator IS
    PORT ( X, Y     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z  : OUT   STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Structure OF comparator IS
    SIGNAL S : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 4) ;
BEGIN
    stage0: fulladd PORT MAP ( '1', X(0), NOT Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), NOT Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), NOT Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), NOT Y(3), S(3), C(4) ) ;
    V <= C(4) XOR C(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = "0000" ELSE '0';
END Structure ;
```

Figura 5.44 Código estructural de VHDL para el circuito comparador.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY comparator IS
    PORT ( X, Y      : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           V, N, Z   : OUT  STD_LOGIC ) ;
END comparator ;

ARCHITECTURE Behavior OF comparator IS
    SIGNAL S : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    S <= ('0' & X) + Y ;
    V <= S(4) XOR X(3) XOR Y(3) XOR S(3) ;
    N <= S(3) ;
    Z <= '1' WHEN S(3 DOWNTO 0) = 0 ELSE '0';
END Behavior ;
```

Figura 5.45 Código VHDL por comportamiento para el circuito comparador.

Ejemplo 5.12

Problema: En la figura 5.32 se muestra un circuito multiplicador de cuatro bits. Cada fila consta de cuatro bloques sumadores completos (FA) conectados en una configuración de acarreo en cascada. El retraso ocasionado por las señales de acarreo que caen en cascada a través de las filas tiene un efecto significativo en el tiempo necesario para generar el producto salida. Con la intención de agilizar el circuito, es posible usar el ordenamiento de la figura 5.46. Aquí, los acarreos en una fila se "guardan" e incluyen en la fila siguiente en la posición de bit correcta. Luego, en la primera fila los sumadores completos pueden usarse para sumar tres bits corridos adecuadamente del multiplicando según los seleccionen los bits multiplicadores. Por ejemplo, en la posición de bit 2 las tres entradas son m_2q_0, m_1q_1 y m_0q_2. En la última fila aún es necesario usar el sumador con acarreo en cascada. Un circuito que consta de un arreglo de sumadores completos conectados de esta forma se llama *arreglo sumador de acarreo guardado*.

¿Cuál es el retraso total del circuito en la figura 5.46, comparado con el del circuito de la figura 5.32?

Solución: En el circuito de la figura 5.32a la trayectoria más larga es a través de los dos sumadores completos del extremo derecho de la fila superior, seguida por los dos FA más a la derecha en la segunda fila, y luego por los cuatro FA de la fila inferior. Por tanto, este retraso es ocho veces el retraso a través de un bloque sumador completo. Además, hay un retraso de compuerta AND necesario para formar las entradas al primer FA en la fila superior. Estos retrasos combinados son el retraso crítico, que determina la velocidad del circuito multiplicador.

En el circuito de la figura 5.46, la trayectoria más larga es a través de los FA más a la derecha de la primera y segunda filas, seguida por los cuatro FA de la fila inferior. Por tanto, el retraso crítico es seis veces el retraso a través de un bloque sumador completo más el retraso de la compuerta AND necesario para formar las entradas al primer FA de la fila superior.

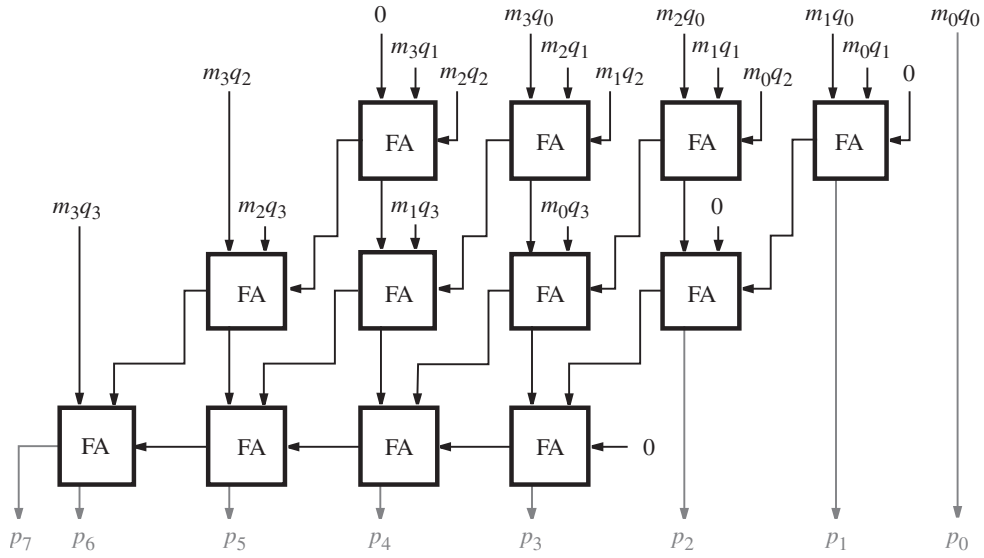


Figura 5.46 Arreglo multiplicador de acarreo guardado.

PROBLEMAS

Al final del libro se proporcionan las respuestas a los problemas marcados con asterisco.

***5.1** Determine los valores decimales de los siguientes números sin signo:

a) $(0111011110)_2$
b) $(1011100111)_2$
c) $(3751)_8$
d) $(A25F)_{16}$
e) $(F0F0)_{16}$

***5.2** Determine los valores decimales de los siguientes números en complemento a 1:

a) 0111011110
b) 1011100111
c) 1111111110

***5.3** Determine los valores decimales de los siguientes números en complemento a 2:

a) 0111011110
b) 1011100111
c) 1111111110

***5.4** Convierta los números decimales 73, 1906, −95 y −1630 en números con signo de 12 bits en las representaciones siguientes:
a) Signo y magnitud
b) Complemento a 1
c) Complemento a 2

5.5 Realice las operaciones siguientes que implican números en complemento a 2 de ocho bits e indique si ocurre desbordamiento aritmético. Compruebe sus respuestas convirtiendo en representación decimal signo y magnitud.

00110110	01110101	11011111
+ 01000101	+ 11011110	+ 10111000
00110110	01110101	11010011
− 00101011	− 11010110	− 11101100

5.6 Pruebe que la operación XOR es asociativa, lo que significa que $x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$.

5.7 Demuestre que el circuito de la figura 5.5 implementa el sumador completo especificado en la figura 5.4*a*.

5.8 Pruebe la validez de la simple regla para encontrar el complemento a 2 de un número presentada en la sección 5.3. Recuerde que la regla establece que al revisar un número de derecha a izquierda, todos los 0 y el primer 1 se copian; luego todos los bits restantes se complementan.

5.9 Pruebe la validez de la expresión Desbordamiento (*overflow*) $= c_n \oplus c_{n-1}$ para la suma de números con signo de n bits.

5.10 En la sección 5.5.4 establecimos que una señal de acarreo, c_k, a partir de la posición de bit $k - 1$ de un circuito sumador puede generarse como $c_k = x_k \oplus y_k \oplus s_k$, donde x_k y y_k son entradas y s_k es el bit suma. Compruebe la exactitud de esta afirmación.

***5.11** Considere el circuito de la figura P5.1. ¿Puede usarse este circuito como una etapa en un sumador con acarreo en cascada? Discuta las ventajas y las desventajas.

***5.12** Determine el número de compuertas necesarias para implementar un sumador con acarreo de adelanto de n bits, si no se suponen restricciones de carga de entrada. Use compuertas AND, OR y XOR con cualquier número de entradas.

***5.13** Determine el número de compuertas necesarias para implementar un sumador con acarreo de adelanto de ocho bits, si se supone que la máxima entrada de carga para las compuertas es cuatro.

5.14 En al figura 5.18 se presentó la estructura de un sumador jerárquico con acarreo de adelanto. Muestre el circuito completo para una versión de cuatro bits de este sumador, construido con dos bloques de dos bits.

5.15 ¿Cuál es la trayectoria de retraso crítico en el multiplicador de la figura 5.32? ¿Cuál es el retraso a lo largo de esta trayectoria en términos del número de compuertas?

5.16 *a*) Escriba una entidad de VHDL para describir el bloque de circuito de la figura 5.32*b*. Use las herramientas CAD para sintetizar un circuito a partir del código y compruebe su exactitud funcional.

b) Escriba una entidad de VHDL para describir el bloque de circuito de la figura 5.32*c*. Use las herramientas CAD para sintetizar un circuito a partir del código y compruebe su exactitud funcional.

c) Escriba una entidad de VHDL para describir el multiplicador de 4×4 de la figura 5.32*a*. Su código debe ser jerárquico y usar los subíndices diseñados en los incisos *a*) y *b*). Sintetice un circuito a partir del código y compruebe su exactitud funcional.

***5.17** Considere el código de VHDL de la figura P5.2. Dada la relación entre las señales IN y OUT, ¿cuál es la funcionalidad del circuito descrito por el código? Comente si este código constituye o no un buen estilo para la funcionalidad que representa.

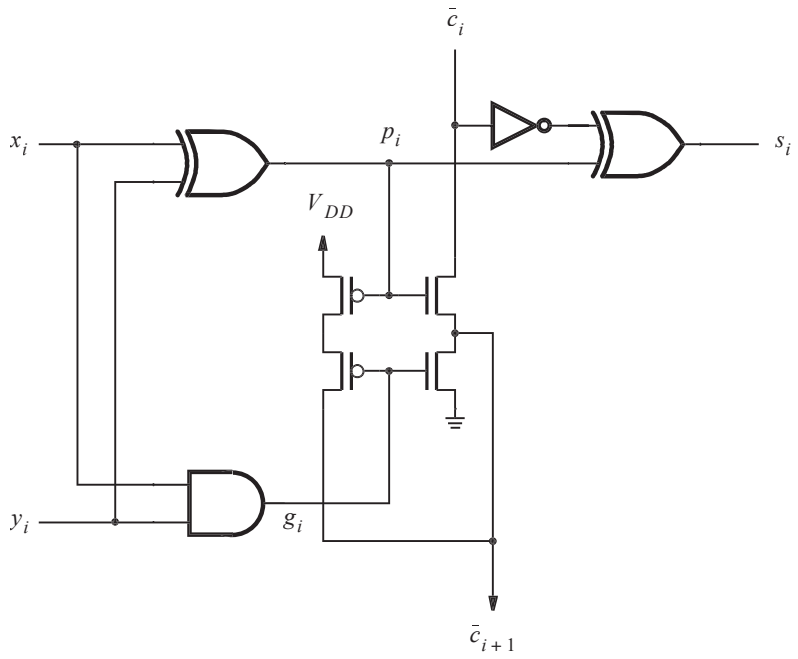


Figura P5.1 Circuito para el problema 5.11.

5.18 Diseñe un circuito que genere el complemento a 9 de un dígito BCD. Note que el complemento a 9 de d es $9 - d$.

5.19 Derive un esquema para realizar la resta usando operandos BCD. Muestre un diagrama de bloques para el circuito restador.

Sugerencia: La resta puede realizarse fácilmente si los operandos están en representación de complemento a 10 (complemento a la base). En esta representación, el dígito signo es 0 para un número positivo y 9 para un número negativo.

5.20 Escriba código completo de VHDL para el circuito que derivó en el problema 5.19.

***5.21** Suponga que se quiere determinar cuántos bits en un número sin signo de tres bits son iguales a 1. Diseñe el circuito más simple que pueda realizar esa tarea.

5.22 Repita el problema 5.21 para un número sin signo de seis bits.

5.23 Repita el problema 5.21 para un número sin signo de ocho bits.

5.24 Muestre una interpretación gráfica de números decimales de tres dígitos, similar a la figura 5.12. El dígito más a la izquierda es 0 para números positivos y 9 para negativos. Compruebe la validez de la respuesta probando algunos ejemplos de suma y resta.

5.25 El sistema numérico ternario tiene tres dígitos: 0, 1, y 2. En la figura P5.3 se define un medio sumador ternario. Diseñe un circuito que implemente este medio sumador usando señales codificadas en binario, tal que dos bits se usen para cada dígito ternario. Sean $A = a_1a_0$, $B = b_1b_0$ y $Sum = s_1s_0$; note que *Carry* es sólo una señal binaria. Use la codificación siguiente: $00 = (0)_3$, $01 = (1)_3$ y $10 = (2)_3$. Minimice el costo del circuito.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY problem IS
    PORT ( Input  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Output : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END problem ;

ARCHITECTURE LogicFunc OF problem IS
BEGIN
    WITH Input SELECT
        Output <= "0001" WHEN "0101",
                  "0010" WHEN "0110",
                  "0011" WHEN "0111",
                  "0010" WHEN "1001",
                  "0100" WHEN "1010",
                  "0110" WHEN "1011",
                  "0011" WHEN "1101",
                  "0110" WHEN "1110",
                  "1001" WHEN "1111",
                  "0000" WHEN OTHERS ;
END LogicFunc ;
```

Figura P5.2 Código para el problema 5.17.

A B	*Carry*	*Sum*
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figura P5.3 Medio sumador ternario.

5.26 Diseñe un circuito sumador completo ternario aplicando el enfoque descrito en el problema 5.25.

5.27 Considere las restas 26 – 27 = 99 y 18 – 34 = 84. Con los conceptos presentados en la sección 5.3.4, explique cómo pueden interpretarse estas respuestas (99 y 84) como los resultados con signos correctos de tales restas.

BIBLIOGRAFÍA

1. V. C. Hamacher, Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 5a. ed. (McGraw-Hill: Nueva York, 2002).
2. D. A. Patterson y J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2a. ed. (Morgan Kaufmann: San Francisco, CA, 1998).
3. Intitute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic", *Computer* 14, núm. 3 (marzo de 1981), pp. 51-62.
4. W. W. Peterson y E. J. Weldon Jr., *Error-Correcting Codes*, 2a. ed. (MIT Press: Boston, MA, 1972).