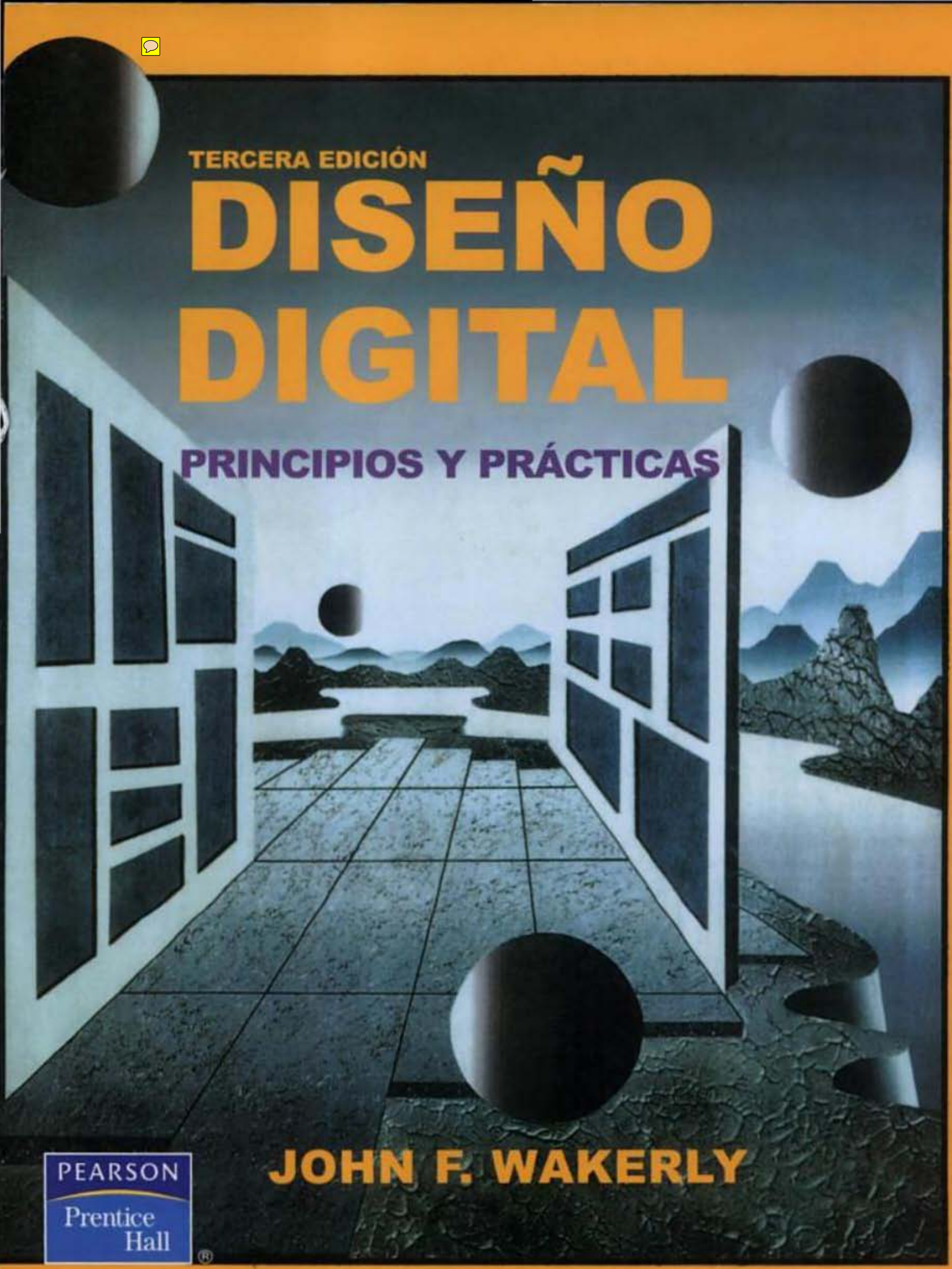
TERCERA EDICIÓN

DISEÑO DIGITAL

PRINCIPIOS Y PRÁCTICAS



JOHN F. WAKERLY

PEARSON
Prentice
Hall
®

capítulo 2

as y códigos numéricos

os sistemas digitales se construyen a partir de circuitos integrados que procesan dígitos binarios (ceros y unos), aunque muy pocos problemas de la vida real están basados en números binarios o cualquier tipo de números en absoluto. Como resultado, un diseñador de sistemas digitales debe establecer cierta correspondencia entre los dígitos binarios que n los circuitos digitales y los números, eventos y condiciones de la vida real. 5sito de este capítulo es demostrarle a usted cómo las cantidades numéricas es pueden representarse y manipularse en un sistema digital, y cómo los ventos y condiciones no numéricos también pueden representarse.

as primeras nueve secciones describen los sistemas numéricos binarios y n cómo se llevan a cabo las operaciones de suma, resta, multiplicación y divi- estos sistemas. Las secciones 2.10 a 2.13 indican cómo se pueden codificar s decimales, caracteres de texto, posiciones mecánicas y condiciones arbi- nediante el uso de cadenas de dígitos binarios.

a sección 2.14 presenta los "cubos *n*", los cuales proporcionan una forma de ar la relación entre diferentes cadenas de bits. Los cubos *n* son especial- tiles en el estudio de los códigos de detección de error en la sección 2.15. mos el capítulo con una introducción a los códigos para transmisión y amiento de datos de un bit a la vez.

2.1 Sistemas numéricos posicionales

sistema numérico posicional

peso o ponderación

El sistema numérico tradicional que aprendimos en la escuela y utilizamos todos los días en los negocios se conoce como un *sistema numérico posicional.* En un sistema de esta clase, un número se representa por medio de una cadena de dígitos, donde cada posición del dígito tiene un *peso* asociado. El valor de un número es una suma ponderada de los dígitos, por ejemplo:

$$1734 = 1 \cdot 1000 + 7 \cdot 100 + 3 \cdot 10 + 4 \cdot 1$$

Cada peso es una potencia de 10 que corresponde a la posición del dígito. Un punto decimal permite que se utilicen tanto potencias negativas como positivas de 10:

$$5185.68 = 5 \cdot 1000 + 1 \cdot 100 + 8 \cdot 10 + 5 \cdot 1 + 6 \cdot 0.1 + 8 \cdot 0.01$$

En general, un número D de la forma $d_1 d_0 . d_{-1} d_{-2}$ tiene el valor

$$D = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

base

raíz

Aquí, el 10 se denomina la *base* (*raíz, origen*) del sistema numérico. En un sistema numérico posicional general, la base puede ser cualquier entero $r \geq 2$, y un dígito en la posición i tiene un peso r^i. La forma de expresión de un número en un sistema de esta clase será

$$d_{p-1} d_{p-2} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$$

punto de la base

donde existen p dígitos a la izquierda del punto y n dígitos a la derecha de éste, llamado el *punto base*. Si este punto se omite, se supone que se encuentra a la derecha del dígito del extremo derecho. El valor del número es la suma de cada dígito multiplicado por el valor correspondiente de la base:

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

dígito de mayor orden

dígito más significativo

dígito de menor orden

dígito menos significativo

Excepto por la posibilidad de tener ceros al principio o al final, la representación de un número en un sistema numérico posicional es única. (Obviamente, 0185.6300 es igual a 185.63, y así por el estilo.) El dígito que está en el extremo izquierdo en un número de este tipo se denomina el *dígito de mayor orden* o *el dígito más significativo*; el que se encuentra en el extremo derecho es el *dígito de menor orden* o *el dígito menos significativo.*

dígito binario

bit

base binaria

Como aprendimos en el capítulo 3, los circuitos digitales tienen señales que se encuentran normalmente en una de dos condiciones: bajo o alto, cargado o descargado, encendido (on) o apagado (off). Las señales en estos circuitos se interpretan para representar *dígitos binarios* (o *bits*) que tienen uno de ambos valores, 0 y 1. De este modo, la *base binaria* se emplea normalmente para representar números en un sistema digital. La forma general de un número binario es

$$b_{p-1} b_{p-2} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$$

y su valor es

$$B = \sum_{i=-n}^{p-1} b_i \cdot 2^i$$

En un número binario, el punto base se denomina el *punto binario*. Cuando se trata con números binarios y otros números no decimales, utilizamos un subíndice para indicar la base de cada número, a menos que la base se sobreentienda en el contexto de trabajo. Ejemplos de números binarios y sus equivalentes decimales se ilustran a continuación.

punto binario

$$10011_2 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 19_{10}$$

$$100010_2 = 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 34_{10}$$

$$101.001_2 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 0 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 = 5.125_{10}$$

El bit que está en el extremo izquierdo de un número binario se conoce como el *bit de mayor orden* o el *bit más significativo* (*MSB*, por las siglas en inglés de *most significant bit*); el bit que está en el extremo derecho es el *bit de menor orden* o *bit menos significativo* (*LSB, least significant bit*).

MSB

LSB

2.2 Números octales y hexadecimales

La base 10 es importante debido a que la utilizamos en las operaciones cotidianas, mientras que la base 2 es importante porque los números binarios se pueden procesar directamente en los circuitos digitales. Los números en otras bases no se procesan directamente pero pueden ser importantes para documentación u otros propósitos. En particular, las bases 8 y 16 proporcionan representaciones convenientes abreviadas para números de múltiples bits en un sistema digital.

El *sistema numérico octal* utiliza la base 8, mientras que el *sistema numérico hexadecimal* emplea la base 16. La tabla 2-1 muestra los enteros binarios desde 0 hasta 1111 y sus equivalentes octal, decimal y hexadecimal. El sistema octal necesita 8 dígitos, de manera que utiliza los dígitos del 0 al 7 del sistema decimal. El sistema hexadecimal necesita 16 dígitos, de modo que combina los dígitos decimales del 0 al 9 con las letras *A* hasta la *F*.

sistema numérico octal

sistema numérico hexadecimal

dígitos hexadecimales A–F

Los sistemas numéricos octal y hexadecimal son útiles para representar números de bits múltiples, debido a que sus bases son potencias de 2. Puesto que una cadena de tres bits puede tomarse en ocho diferentes combinaciones, se sigue que cada cadena de 3 bits puede representarse de manera única mediante un dígito octal, de acuerdo con la tercera y la cuarta columnas de la tabla 2-1. De manera semejante, una cadena de 4 bits puede ser representada por un dígito hexadecimal de acuerdo con las columnas quinta y sexta de la tabla.

De este modo, es muy fácil convertir un número binario en octal. Comenzando desde el punto binario y viajando hacia la izquierda, simplemente separamos los bits en grupos de tres y reemplazamos cada grupo con el correspondiente dígito octal:

conversión de binario a octal

$$1000110011110_2 = 100\ 011\ 001\ 110_2 = 4316_8$$

$$1110110111010101001_2 = 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8$$

El procedimiento para la conversión de binario a hexadecimal es semejante, excepto que utilizamos grupos de cuatro bits:

conversión de binario a hexadecimal

$$1000110011110_2 = 1000\ 1100\ 1110_2 = 8CE_{16}$$

$$1110110111010101001_2 = 0001\ 1101\ 1011\ 1010\ 1001_2 = 1DBA9_{16}$$

En estos ejemplos, hemos agregado libremente ceros a la izquierda para hacer el número total de bits un múltiplo de 3 o 4 como se requiere.

Tabla 2-1
Números binarios, decimales, octales y hexadecimales.

Binario	*Decimal*	*Octal*	*Cadena de 3 bits*	*Hexadecimal*	*Cadena de 4 bits*
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

Si un número binario contiene dígitos a la derecha del punto binario, podemos convertirlo a octal o hexadecimal al comenzar por el punto binario y movernos hacia la derecha. Tanto el lado izquierdo como el derecho pueden rellenarse con ceros para obtener múltiplos de tres o cuatro bits, como se muestra en el ejemplo que se presenta a continuación:

$$10.1011001011_2 = 010\,.\,101\ 100\ 101\ 100_2 = 2.5454_8$$
$$= 0010\,.\,1011\ 0010\ 1100_2 = 2.B2C_{16}$$

conversión octal o hexadecimal a binaria

Realizar la conversión en la dirección inversa, partiendo de octal o hexadecimal hacia binario, es algo muy sencillo. Simplemente reemplazamos cada dígito octal o hexadecimal con la correspondiente cadena de 3 o 4 bits, como se muestra a continuación:

$$1357_8 = 001\ 011\ 101\ 111_2$$
$$2046.17_8 = 010\ 000\ 100\ 110\,.\,001\ 111_2$$
$$BEAD_{16} = 1011\ 1110\ 1010\ 1101_2$$
$$9F.46C_{16} = 1001\ 111\,.\,0100\ 0110\ 1100_2$$

byte (octeto)

El sistema numérico octal fue bastante popular hace 15 años debido a que ciertas minicomputadoras tenían sus luces e interruptores frontales acomodados en grupos de tres. Sin embargo, el sistema numérico octal no se utiliza mucho en la actualidad, a consecuencia de la preponderancia de las máquinas que procesan *bytes* compuestos de 8 bits. Es difícil extraer los valores de byte individual en cantidades de bytes múltiples en

la representación octal; por ejemplo, ¿cuáles son los valores en octal de los cuatro bytes de 8 bits en el número de 32 bits cuya representación octal es 12345670123_8?

En el sistema hexadecimal, dos dígitos representan un byte de 8 bits y $2n$ dígitos representan una palabra de n bytes; cada par de dígitos constituyen exactamente un byte. Por ejemplo, el número hexadecimal de 32 bits $5678ABCD_{16}$ se compone de cuatro bytes con valores 56_{16}, 78_{16}, AB_{16} y CD_{16}. En este contexto, un dígito hexadecimal de 4 bits se denomina a veces un *nibble (medio byte)*; un número de 32 bits (4 bytes) tiene ocho nibbles. Los números hexadecimales se utilizan con cierta frecuencia para describir un espacio de dirección de la memoria de la computadora. Por ejemplo, una computadora con direcciones de 16 bits puede describirse como que tiene memoria de lectura/escritura instalada en las direcciones $0\text{-}EFFF_{16}$ y memoria sólo de lectura en las direcciones $F000\text{-}FFFF_{16}$. Muchos lenguajes de programación para computadoras utilizan el prefijo "0x" para denotar un número hexadecimal, por ejemplo, `0xBFC0000`.

nibble (medio byte o medio octeto)

prefijo `0x`

2.3 Conversiones generales de sistema numérico posicional

En general, la conversión entre dos bases no puede hacerse por simple sustitución; se requieren operaciones aritméticas. En esta sección mostraremos cómo convertir un número en cualquier base a la base 10 y viceversa, haciendo uso de aritmética de base 10.

En la sección 2.1, indicamos que el valor de un número en cualquier base está dado por la fórmula

conversión de base-r a decimal

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

donde r es la base del número y existen p dígitos a la izquierda del punto base y n a la derecha. De esta forma, el valor del número puede encontrarse al convertir cada dígito del número a su equivalente en base 10, y expandir la fórmula utilizando aritmética de base 10. A continuación se proporcionan algunos ejemplos:

$$1CE8_{16} = 1\cdot16^3 + 12\cdot16^2 + 14\cdot16^1 + 8\cdot16^0 = 7400_{10}$$
$$F1A3_{16} = 15\cdot16^3 + 1\cdot16^2 + 10\cdot16^1 + 3\cdot16^0 = 61859_{10}$$
$$436.5_8 = 4\cdot8^2 + 3\cdot8^1 + 6\cdot8^0 + 5\cdot8^{-1} = 286.625_{10}$$
$$132.3_4 = 1\cdot4^2 + 3\cdot4^1 + 2\cdot4^0 + 3\cdot4^{-1} = 30.75_{10}$$

fórmula de expansión anidada

Un atajo para convertir números enteros a base 10 puede obtenerse al volver a escribir la fórmula de expansión de manera anidada:

$$D = ((\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)\cdots r + d_1)\cdot r + d_0$$

Esto es, comenzamos con una suma de 0; iniciando con el dígito que está en el extremo izquierdo, multiplicamos la suma por *r*, y agregamos el siguiente dígito a la suma, repetimos este proceso hasta que todos los dígitos hayan sido procesados. Por ejemplo, podemos escribir

$$\mathrm{F1AC}_{16} = (((15)\cdot 16 + 1)\cdot 16 + 10)\cdot 16 + 12$$

conversión de decimal a base r

Esta fórmula se emplea en algoritmos de conversión programados e iterativos (como los de la tabla 4-38 en la página 279). También es el fundamento de un método muy conveniente para convertir un número decimal *D* a una base *r*. Considere lo que ocurre si dividimos la fórmula entre *r*. Puesto que la parte entre paréntesis de la fórmula es igualmente divisible entre *r*, el cociente será

$$Q = (\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)\cdot r + d_1$$

y el residuo será d_0. De este modo, d_0 puede calcularse como el residuo de la división larga de *D* entre *r*. Adicionalmente, el cociente *Q* tiene la misma forma que la fórmula original. Por lo tanto, divisiones sucesivas entre *r* nos proporcionan dígitos sucesivos de *D* de derecha a izquierda, hasta que todos los dígitos de *D* hayan sido derivados. Acto seguido, se muestran ejemplos que ilustran lo anterior:

179 ÷ 2 = 89 residuo 1 (LSB)
÷2 = 44 residuo 1
÷2 = 22 residuo 0
÷2 = 11 residuo 0
÷2 = 5 residuo 1
÷2 = 2 residuo 1
÷2 = 1 residuo 0
÷2 = 0 residuo 1 (MSB)

$179_{10} = 10110011_2$

467 ÷ 8 = 58 residuo 3 (dígito significativo menor)
÷8 = 7 residuo 2
÷ 8 = 0 residuo 7 (dígito significativo mayor)

$467_{10} = 723_8$

3417 ÷ 16 = 213 residuo 9 (dígito significativo menor)
÷ 16 = 13 residuo 5
÷ 16 = 0 residuo 13 (dígito significativo mayor)

$3417_{10} = \mathrm{D59}_{16}$

La tabla 2-2 resume los métodos para la conversión entre las bases más comunes.

Tabla 2-2 Métodos de conversión para bases comunes.

Conversión	*Método*	*Ejemplo*
Binario a		
Octal	Sustitución	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Sustitución	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Suma	$10111011001_2 = 1 \cdot 1024 + 0 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1497_{10}$
Octal a		
Binario	Sustitución	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Sustitución	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Suma	$1234_8 = 1 \cdot 512 + 2 \cdot 64 + 3 \cdot 8 + 4 \cdot 1 = 668_{10}$
Hexadecimal a		
Binario	Sustitución	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Sustitución	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Suma	$C0DE_{16} = 12 \cdot 4096 + 0 \cdot 256 + 13 \cdot 16 + 14 \cdot 1 = 49374_{10}$
Decimal a		
Binario	División	$108_{10} \div 2 = 54$ residuo 0 (LSB) $\div 2 = 27$ residuo 0 $\div 2 = 13$ residuo 1 $\div 2 = 6$ residuo 1 $\div 2 = 3$ residuo 0 $\div 2 = 1$ residuo 1 $\div 2 = 0$ residuo 1 (MSB) $108_{10} = 1101100_2$
Octal	División	$108_{10} \div 8 = 13$ residuo 4 (dígito menos significativo) $\div 8 = 1$ residuo 5 $\div 8 = 0$ residuo 1 (dígito más significativo) $108_{10} = 154_8$
Hexadecimal	División	$108_{10} \div 16 = 6$ residuo 12 (dígito menos significativo) $\div 16 = 0$ residuo 6 (dígito más significativo) $108_{10} = 6C_{16}$

Tabla 2-3
Suma binaria y tabla de restas.

c_{ent} o b_{ent}	x	y	c_{sal}	s	b_{sal}	d
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

2.4 Suma y resta de números no decimales

La suma y la resta de números no decimales con el método manual utiliza la misma técnica que aprendimos en la escuela primaria para los números decimales; la única trampa es que las tablas de suma y resta son diferentes.

suma binaria

La tabla 2-3 es la tabla de suma y resta para dígitos binarios. Para sumar dos números binarios X y Y, sumamos juntos los bits menos significativos con un acarreo inicial (c_{ent}) de 0, produciendo bits de acarreo (c_{sal}) y de suma (s) de acuerdo con la tabla. Continuamos procesando bits de derecha a izquierda, sumando el acarreo fuera de cada columna a la suma de la siguiente columna.

Dos ejemplos de sumas decimales y las correspondientes sumas binarias se muestran en la figura 2-1, los dígitos en negritas indican el acarreo de 1. Los mismos ejemplos se repiten a continuación junto con dos más, con los acarreos mostrados como una cadena de bits C:

C		**101111000**	***C***		**001011000**
X	190	10111110	X	173	**10101101**
Y	+141	+ 10001101	Y	+ 44	+ **00101100**
$X+Y$	331	101001011	$X+Y$	217	11011001
C		**011111110**	***C***		**000000000**
X	127	01111111	X	170	**10101010**
Y	+ 63	+ 00111111	Y	+ 85	+ **01010101**
$X+Y$	190	10111110	$X+Y$	255	11111111

resta binaria

minuendo

sustraendo

La resta binaria se realiza de modo similar, empleando "préstamos" (b_{ent} y b_{sal}) en lugar de acarreos entre pasos, y produciendo un bit de diferencia d. Dos ejemplos de restas decimales y las correspondientes restas binarias se muestran en la figura 2-2. Como en la resta decimal, los valores del minuendo binario en las columnas se modifican cuando se presenta el préstamo, como se ilustra mediante las flechas y los bits que aparecen

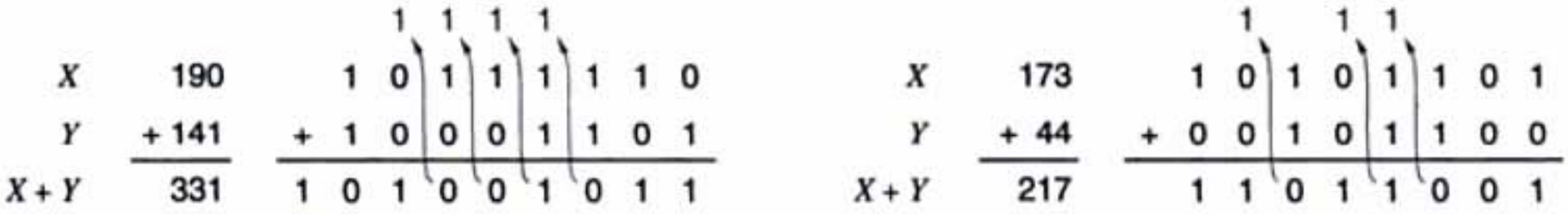


Figura 2-1 Ejemplos de sumas decimales y sus correspondientes.

en negritas. Los ejemplos de la figura se repiten a continuación con dos más, mostrando esta vez los préstamos como una cadena de bits *B*:

B		**001111100**	***B***		**011011010**
X	229	11100101	*X*	210	11010010
Y	− 46	− 00101110	*Y*	−109	− 01101101
X−Y	183	10110111	*X−Y*	101	01100101
B		**010101010**	***B***		**000000000**
X	170	10101010	*X*	221	11011101
Y	− 85	− 01010101	*Y*	− 76	− 01001100
X−Y	85	01010101	*X−Y*	145	10010001

comparación de números

Un uso muy común de la resta en computadoras es la comparación de dos números. Por ejemplo, si la operación $X - Y$ produce un préstamo (o transporte negativo) que procede de la posición del bit más significativo, entonces X es menor que Y; de otro modo, X es mayor o igual que Y. La relación entre acarreos y préstamos en sumadores y restadores se explorará en la sección 5.10.

Las tablas de suma y resta pueden desarrollarse para dígitos octales y hexadecimales, o cualquier otra base deseada. Sin embargo, pocos ingenieros en computación se molestan en memorizar estas tablas. Si usted necesita manipular números no decimales, entonces será fácil en esas ocasiones convertirlos a formato decimal, calcular resultados,

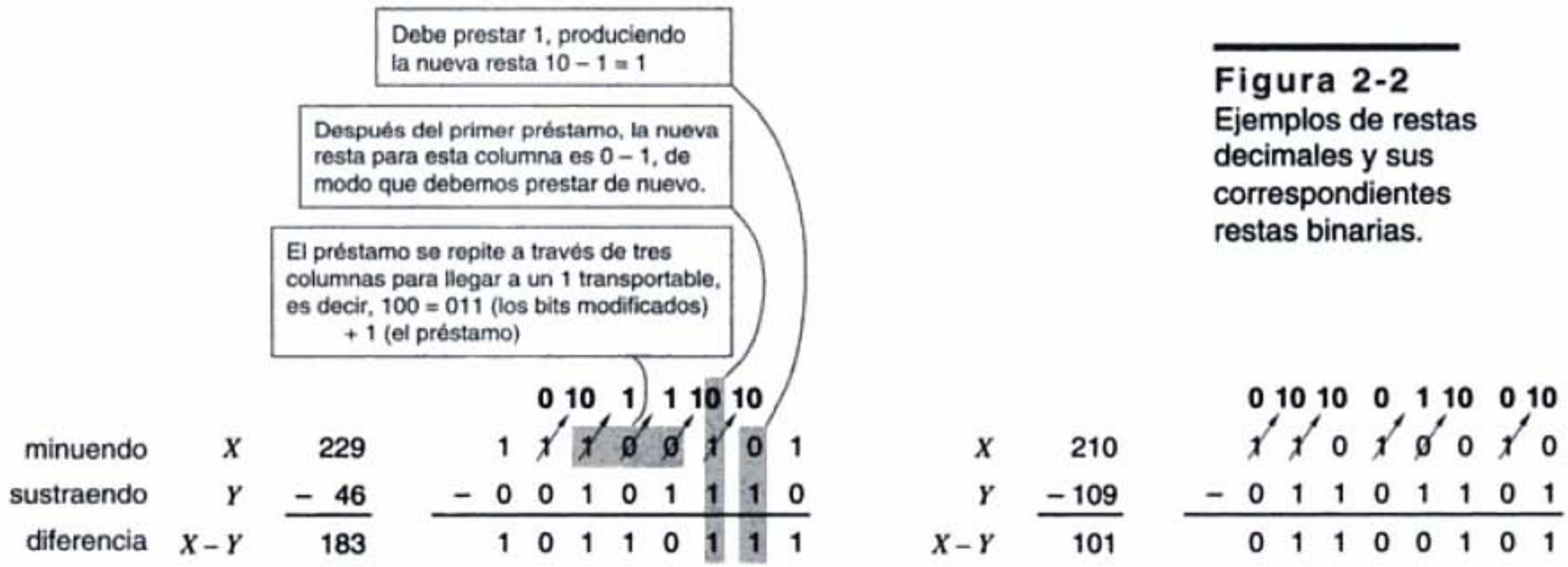


Figura 2-2 Ejemplos de restas decimales y sus correspondientes restas binarias.

y convertirlos de vuelta otra vez. Por otro lado, si debe realizar cálculos en formato binario, octal o hexadecimal con bastante frecuencia, entonces debería pedirle a Santa Claus una calculadora hexadecimal para programadores de Texas Instruments o de Casio.

Si las baterías de la calculadora se agotan, puede utilizar algunos atajos mentales para facilitar la aritmética no decimal. En general, cada suma (o resta) de columna puede hacerse al convertir los dígitos de la columna a formato decimal, sumar en decimal, y posteriormente convertir el resultado a la suma correspondiente y dígitos de acarreo en la base no decimal (un acarreo se produce siempre que la suma de la columna es igual o mayor a la base). Puesto que la suma se hace en formato decimal, confiamos en nuestros conocimientos de la tabla decimal de adición; la única cosa novedosa que necesitamos aprender es la conversión de dígitos decimales a no decimales y viceversa. La secuencia de pasos para sumar mentalmente dos números hexadecimales se muestra a continuación:

suma hexadecimal

C		**1**	**1**	**0**	**0**		**1**	**1**	**0**	**0**
X		1	9	B	9_{16}		1	9	11	9
Y	+	C	7	E	6_{16}	+	12	7	14	6
X+Y		E	1	9	F_{16}		14	17	25	15
							14	16+1	16+9	15
							E	1	9	F

2.5 Representación de números negativos

Hasta ahora, solamente hemos tratado con números positivos, pero existen muchas formas de representar números negativos. En los negocios utilizamos el sistema de magnitud con signo, que se discute más adelante. Sin embargo, la mayor parte de las computadoras emplean alguno de los sistemas numéricos de complemento que presentaremos posteriormente.

2.5.1 Representación de magnitud con signo

sistema de magnitud con signo

En el *sistema de magnitud con signo*, un número se compone de una magnitud y de un símbolo que indica si la magnitud es positiva o negativa. De esta forma, interpretamos los números decimales +98, –57, +123.5 y –13 de la manera habitual, y también suponemos que el signo es "+" si no aparece ningún símbolo escrito. Existen dos posibles representaciones de cero, "+0" y "–0", pero ambas tienen el mismo valor.

bit de signo

El sistema de magnitud con signo se aplica a los números binarios haciendo uso de una posición de bit extra para representar el signo (el *bit de signo*). Tradicionalmente, el bit más significativo (MSB, por sus siglas en inglés) de una cadena de bits es empleado como el bit de signo (0 = signo más, 1 = signo menos), y los bits de menor orden contienen la magnitud. Así, podemos escribir varios enteros de 8 bits con magnitud con signo y sus equivalentes decimales:

$$01010101_2 = +85_{10} \qquad 11010101_2 = -85_{10}$$
$$01111111_2 = +127_{10} \qquad 11111111_2 = -127_{10}$$
$$00000000_2 = +0_{10} \qquad 10000000_2 = -0_{10}$$

El sistema de magnitud con signo tiene un número idéntico de enteros positivos y negativos. Un entero de magnitud con signo de n bits está situado dentro del intervalo que va desde $-(2^{n-1}-1)$ hasta $+(2^{n-1}-1)$ y existen dos representaciones posibles del cero.

sumador de magnitud con signo

Ahora supongamos que deseamos construir un circuito lógico digital que sume números de magnitud con signo. El circuito debe examinar los signos de los sumandos para determinar qué hacer con las magnitudes. Si los signos son los mismos, debe sumar las magnitudes y proporcionar el resultado con el mismo signo. Si los signos son diferentes, debe comparar las magnitudes, restar el más pequeño del mayor y proporcionar al resultado el signo del más grande. Todos estos "si" condicionales, "sumas", "restas" y "comparaciones" se traducen en una gran cantidad de circuitos lógicos complejos. Los sumadores para los sistemas numéricos de complemento son mucho más simples, como demostraremos a continuación. Quizás la única característica que redime a un sistema de magnitud con signo es que, una vez que sabemos cómo construir un sumador de magnitud con signo, la fabricación de un restador de magnitud con signo es algo casi trivial: solamente se debe cambiar el signo del sustraendo y pasarlo junto con el minuendo a un sumador.

restador de magnitud con signo

2.5.2 Sistemas numéricos de complemento

sistema numérico de complemento

Mientras que el sistema de magnitud con signo convierte en negativo un número al cambiar su signo, un *sistema numérico de complemento* convierte en negativo un número tomando su complemento como definido por el sistema. Tomar el complemento es más difícil que cambiar el signo, pero dos números en un sistema numérico de complemento pueden sumarse o restarse directamente sin tener que realizar las verificaciones de magnitud y signo que requiere el sistema de magnitud con signo. Describiremos dos sistemas numéricos de complemento, llamados el "complemento de base" y el "complemento de base reducida".

En cualquier sistema numérico de complemento, normalmente tratamos con un número fijo de dígitos, digamos n. (Sin embargo, podemos aumentar el número de dígitos mediante "extensión de signo" como se muestra en el ejercicio 2.23, y disminuir el número mediante el truncamiento de los dígitos de orden mayor como se muestra en el ejercicio 2.24.) Suponemos adicionalmente que la base es r, y que los números tienen la forma

$$D = d_{n-1}d_{n-2}\cdots d_1 d_0.$$

El punto de base se encuentra a la derecha y por tanto el número es un entero. Si una operación produce un resultado que requiera más de n dígitos, eliminamos el (los) dígito(s) extra de mayor orden. Si un número D se complementa dos veces, el resultado será D.

2.5.3 Representación de complemento de base

sistema de complemento de base

complemento a 10

En un *sistema de complemento de base*, el complemento de un número de n dígitos se obtiene al restarlo de r^n. En el sistema numérico decimal, el complemento de base se denomina *complemento de 10*. Algunos ejemplos utilizando números decimales de 4 dígitos (y resta de 10,000) se muestran en la tabla 2–4.

Por definición, el complemento de base de un número D de n dígitos se obtiene al restarlo de r^n. Si D se encuentra entre 1 y $r^n - 1$, esta resta produce otro número entre 1 y

Tabla 2-4
Ejemplos de complementos de 10 y 9.

Número	*Complemento de 10*	*Complemento de 9*
1849	8151	8150
2067	7933	7932
100	9900	9899
7	9993	9992
8151	1849	1848
0	10000 (= 0)	9999

$r^n - 1$. Si D es 0, el resultado de la resta es r^n, lo cual tiene la forma $100 \cdots 00$, donde hay un total de $n + 1$ dígitos. Descartamos el dígito extra de mayor orden y obtenemos el resultado 0. Por consiguiente, sólo existe una representación de cero en un sistema de complemento de base.

cálculo del complemento de base

Parece de la definición que una operación de resta es necesaria para calcular el complemento de base de D. Sin embargo, esta resta puede evitarse al volver a escribir r^n como $(r^n - 1) + 1$ y $r^n - D$ como $((r^n - 1) - D) + 1$. El número $r^n - 1$ tiene la forma $mm \cdots mm$, donde $m = r - 1$ y hay n cantidad de m's. Por ejemplo, 10,000 es igual a 9,999 + 1. Si definimos el complemento de un dígito d como $r - 1 - d$, entonces $(r^n - 1) - D$ se obtiene mediante la complementación de los dígitos de D. Por consiguiente, el complemento de base de un número D se obtiene al complementar los dígitos individuales de

Tabla 2-5
Complementos de dígito.

	Complemento			
Dígito	*Binario*	*Octal*	*Decimal*	*Hexadecimal*
0	1	7	9	F
1	0	6	8	E
2	–	5	7	D
3	–	4	6	C
4	–	3	5	B
5	–	2	4	A
6	–	1	3	9
7	–	0	2	8
8	–	–	1	7
9	–	–	0	6
A	–	–	–	5
B	–	–	–	4
C	–	–	–	3
D	–	–	–	2
E	–	–	–	1
F	–	–	–	0

resultado es válido solamente si todos los bits descartados son iguales que el bit de signo del resultado (véase el ejercicio 2.24).

La mayoría de las computadoras y otros sistemas digitales utilizan el sistema de complemento a dos para representar números negativos. No obstante, para completar nuestro estudio, también describiremos los sistemas de complemento de base reducida y de complemento a unos.

*2.5.5 Representación de complemento de base reducida

sistema de complemento de base reducida

En un *sistema de complemento de base reducida*, el complemento de un número de n dígitos D se obtiene al restarlo de $r^n - 1$. Esto puede conseguirse al complementar los dígitos individuales de D, *sin* sumar 1, como sucede en el sistema de complemento de base. En representación decimal, esto se denomina *complemento de 9s*; se proporcionan algunos ejemplos en la última columna de la tabla 2-4 de la página 36.

complemento de 9s

*2.5.6 Representación de complemento a unos

complemento a unos

El sistema de complemento de base reducida para números binarios se conoce como el *complemento a unos*. Como en el complemento a dos, el bit más significativo es el signo, 0 si es positivo y 1 si es negativo. De este modo hay dos representaciones del cero, cero positivo ($00\cdots00$) y cero negativo ($11\cdots11$). Las representaciones de los números positivos son las mismas tanto para el complemento a dos como para el complemento a unos. Sin embargo, las representaciones de los números negativos difieren por 1. Un peso de $-(2^{n-1}-1)$, en vez de -2^{n-1}, se otorga al bit más significativo cuando se calcula el equivalente decimal de un número de complemento a unos. El intervalo de números representables va de $-(2^{n-1}-1)$ hasta $+(2^{n-1}-1)$. Algunos números de 8 bits y sus complementos de uno se muestran enseguida:

$$17_{10} = 00010001_2 \Downarrow 11101110_2 = -17_{10} \qquad -99_{10} = 10011100_2 \Downarrow 01100011_2 = 99_{10}$$

$$119_{10} = 01110111_2 \Downarrow 10001000_2 = -119_{10} \qquad -127_{10} = 10000000_2 \Downarrow 01111111_2 = 127_{10}$$

$$0_{10} = 00000000_2 \text{ (cero positivo)} \Downarrow 11111111_2 = 0_{10} \text{ (cero negativo)}$$

Las principales ventajas del sistema de complemento a unos es su simetría y la facilidad de complementación. Sin embargo, el diseño del sumador para números de complemento a unos es algo más delicado que un sumador de complemento a dos (véase el ejercicio 7.72). Además, los circuitos de detección de cero en un sistema de com-

* En todo el libro, las secciones opcionales están marcadas con un asterisco.

plemento a unos deben verificar ambas representaciones de cero, o convertir siempre $11\cdots11$ a $00\cdots00$.

*2.5.7 Representaciones por exceso

Sí, el número de sistemas diferentes para la representación de números negativos *es* enorme, pero existe una más que debemos incluir. En la *representación de exceso B*, una cadena de m bits cuyo valor de entero sin signo es $M(0 \le M < 2^m)$representa el entero con signo $M - B$, donde B se denomina la *tendencia* del sistema numérico.

representación de exceso B

tendencia

Por ejemplo, un *sistema de exceso* 2^{m-1} representa cualquier número X en el intervalo de -2^{m-1} a $+2^{m-1} - 1$ mediante la representación binaria de m bits de $X + 2^{m-1}$(que siempre es no negativa y menor que 2^m). El intervalo de esta representación es exactamente la misma que en el caso de los números de complemento a dos de m bits. De hecho, las representaciones de cualquier número en ambos sistemas son idénticas, excepto por los bits de signo, que siempre son opuestos. (Advierta que esto es verdad solamente cuando la tendencia es 2^{m-1}.)

sistema de exceso de 2^{m-1}

El uso más común de las representaciones de exceso se encuentra en los sistemas numéricos de punto flotante (véase la parte de Referencias).

2.6 Suma y resta de complemento a dos

2.6.1 Reglas de la suma

Una tabla de números decimales y sus equivalentes en diferentes sistemas numéricos, tabla 2-6, revela por qué se prefiere el complemento a dos en las operaciones aritméticas. Si comenzamos con 1000_2 (-8_{10}) y contamos de manera ascendente, vemos que cada número sucesivo de complemento a dos en todo el recorrido hasta 0111_2 $(+7_{10})$ puede obtenerse mediante la suma de 1 al anterior, ignorando cualquier acarreo más allá de la posición del cuarto bit. No puede decirse lo mismo de los números de complemento a unos y de magnitud con signo. Debido a que la suma ordinaria es simplemente una extensión del conteo, los números de complemento a dos pueden sumarse mediante la suma binaria ordinaria, ignorando cualquier acarreo más allá del MSB. El resultado siempre será la suma correcta siempre y cuando no se exceda el intervalo del sistema numérico. Algunos ejemplos de suma decimal y las correspondientes sumas de complemento a dos de 4 bits confirman esto:

suma de complemento a dos

+3	0011	−2	1110
+ +4	+ 0100	+ −6	+ 1010
+7	0111	−8	**1**1000
+6	0110	+4	0100
+ −3	+ 1101	+ −7	+ 1001
+3	**1**0011	−3	1101

Tabla 2-6 Números decimales y de 4 bits.

Decimal	*Complemento a dos*	*Complemento a unos*	*Magnitud con signo*	*Exceso de 2^{m-1}*
–8	1000	—	—	0000
–7	1001	1000	1111	0001
–6	1010	1001	1110	0010
–5	1011	1010	1101	0011
–4	1100	1011	1100	0100
–3	1101	1100	1011	0101
–2	1110	1101	1010	0110
–1	1111	1110	1001	0111
0	0000	1111 o 0000	1000 o 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

2.6.2 Una visión gráfica

Otra manera de ver el sistema de complemento a dos hace uso del "contador" de 4 bits que se ilustra en la figura 2-3. Aquí hemos ilustrado los números en una representación circular o "modular". La operación de este contador imita muy de cerca la de un circuito contador en ambos sentidos en la vida real, el cual estudiaremos en la sección 8.4. Si empezamos con la flecha que apunta a cualquier número, podemos sumar $+n$ a ese número al contar de manera ascendente n veces, es decir, moviendo la flecha n posiciones

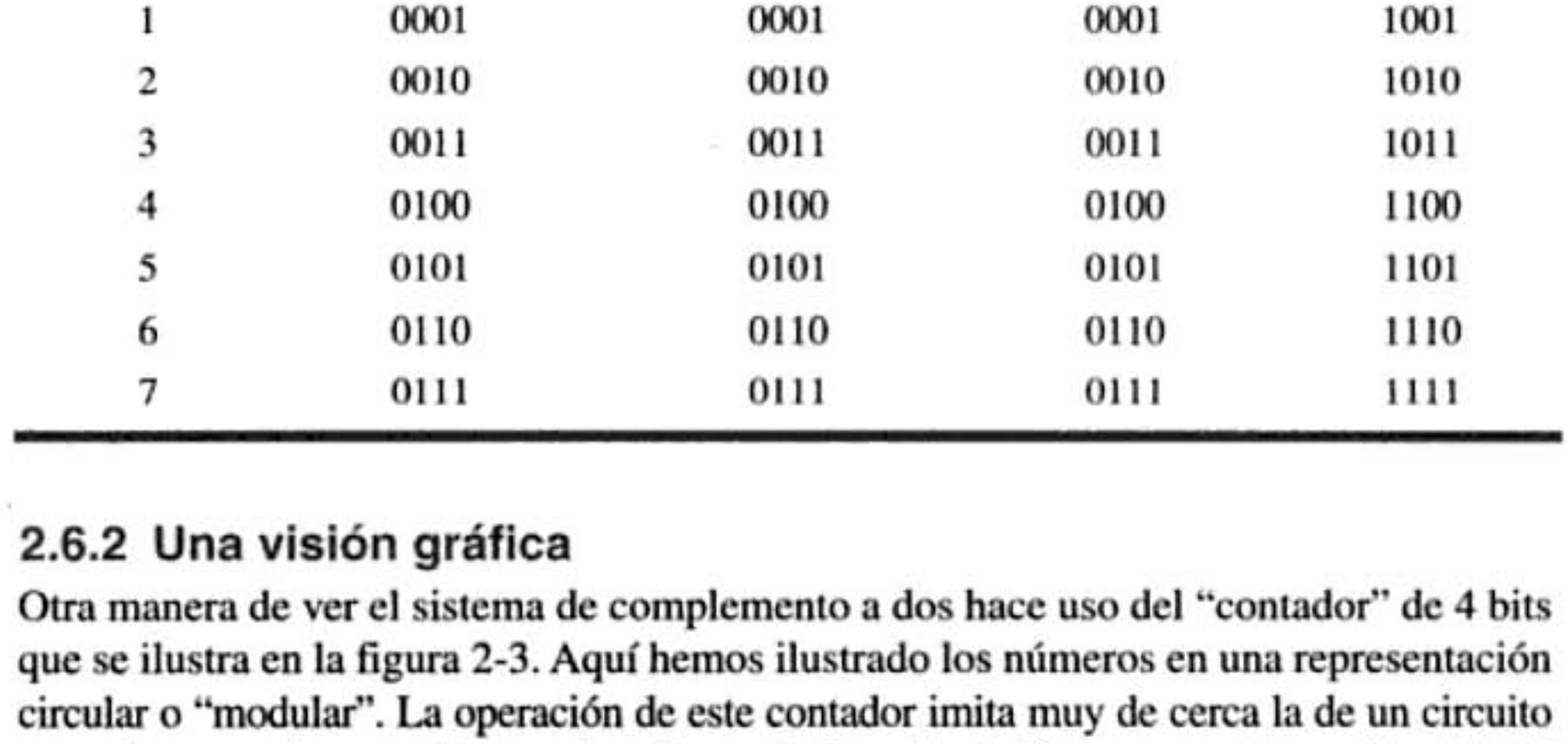


Figura 2-3 Una representación de conteo modular de números de complemento a dos de 4 bits.

en el sentido de giro de las manecillas del reloj. También es evidente que podemos restar n de un número al contar de forma descendente n veces, es decir, al mover la flecha n posiciones en sentido contrario de las manecillas del reloj. Naturalmente, estas operaciones proporcionan resultados correctos solamente si n es lo suficientemente pequeño de modo que no cruce la discontinuidad que existe entre –8 y +7.

Lo que es más interesante es que también podemos restar n (o sumar $-n$) al mover la flecha $16 - n$ posiciones en el sentido de giro de las manecillas del reloj. Nótese que la cantidad $16 - n$ es lo que definimos como el complemento a dos de 4 bits de n, es decir, la representación de complemento a dos de $-n$. Esto apoya gráficamente nuestra afirmación anterior, en la que un número negativo en la representación de complemento a dos puede ser sumado a otro número simplemente sumando las representaciones de 4 bits utilizando la suma binaria ordinaria. Sumar un número en la figura 2–3 es equivalente a mover la flecha un número correspondiente de posiciones en sentido contrario al de las manecillas del reloj.

2.6.3 Desbordamiento

desbordamiento

Si una operación de suma produce un resultado que excede el intervalo del sistema numérico, se dice que ocurre un *desbordamiento*. En la representación de conteo modular de la figura 2-3, el desbordamiento se presenta durante la suma de números positivos cuando contamos pasando +7. La suma de dos números con signos diferentes nunca puede producir desbordamiento, pero la suma de dos números de signo semejante puede hacerlo, como se muestra en los ejemplos siguientes:

–3	1101		+5	0101	
+ –6	+ 1010		+ +6	+ 0110	
–9	**1**0111	= +7	+11	1011	= –5
–8	1000		+7	0111	
+ –8	+ 1000		+ +7	+ 0111	
–16	**1**0000	= +0	+14	1110	= –2

reglas del desbordamiento

Afortunadamente, existe una regla simple para detectar el desbordamiento en la suma: una suma provoca desbordamiento si los signos de los sumandos son los mismos y el signo de la suma es diferente del signo de los sumandos. La regla del desbordamiento se establece en ocasiones en términos de acarreos que se generan durante la operación de suma: una suma provoca desbordamiento si los bits de acarreo de entrada c_{ent} y de salida c_{sal} de la posición del signo son diferentes. Un examen detallado de la tabla 2-3 en la página 32 muestra que ambas reglas son equivalentes: solamente existen dos casos donde $c_{ent} \neq c_{sal}$ y éstos son los dos únicos casos donde $x = y$, y el bit de suma es diferente.

2.6.4 Reglas de la resta

resta de complemento a dos

Los números de complemento a dos pueden restarse como si fueran números binarios ordinarios sin signo, y pueden formularse reglas apropiadas para detectar un desbordamiento. Sin embargo, la mayor parte de los circuitos de resta para números de complemento a dos no realizan la resta en forma directa. En vez de ello, hacen negativo el sustraendo tomando su complemento a dos y posteriormente lo suman al minuendo aplicando las reglas normales para la suma.

La negación del sustraendo y su adición al minuendo pueden llevarse a cabo con solamente una operación de suma como se explica a continuación: realice un complemento bit a bit del sustraendo y sume el sustraendo complementado al minuendo con un acarreo de entrada (c_{ent}) de 1 en lugar de 0. A continuación presentamos algunos ejemplos:

		$\mathbf{1} - c_{ent}$			$\mathbf{1} - c_{ent}$
+4	0100	0100	+3	0011	0011
− +3	− 0011	+ 1100	− +4	− 0100	+ 1011
+1		**1**0001	−1		1111

		$\mathbf{1} - c_{ent}$			$\mathbf{1} - c_{ent}$
+3	0011	0011	−3	1101	1101
− −4	− 1100	+ 0011	− −4	− 1100	+ 0011
+7		0111	+1		**1**0001

El desbordamiento en la resta puede detectarse mediante el examen de los signos del minuendo y del sustraendo *complementado*, utilizando la misma regla que en la suma. O, empleando la técnica de los ejemplos anteriores, se pueden observar los acarreos de entrada y de salida de la posición del signo y detectar el desbordamiento, independientemente de los signos de entrada y salida, usando de nueva cuenta la misma regla de la suma.

Un intento de hacer negativo el número negativo "extra" da como resultado un desbordamiento de acuerdo con las reglas anteriores, cuando agregamos 1 en el proceso de complementación:

$$-(-8) = -1000 = \begin{array}{r} 0111 \\ +\ 0001 \\ \hline 1000 \end{array} = -8$$

No obstante, este número todavía puede ser utilizado en sumas y restas mientras que el resultado final no exceda el intervalo del número:

				$\mathbf{1} - c_{ent}$
+4	0100	−3	1101	1101
+ −8	+ 1000	− −8	− 1000	+ 0111
−4	1100	+5		**1**0101

2.6.5 Números binarios sin signo y complemento a dos

Puesto que los números de complemento a dos se suman y restan siguiendo los mismos algoritmos de suma y resta binaria básicas que los números sin signo de la misma longitud, una computadora u otro sistema digital puede usar el mismo circuito sumador para manejar números de ambos tipos. Sin embargo, los resultados deben ser

interpretados de manera diferente, dependiendo de si el sistema está tratando con números con signo (por ejemplo, de –8 hasta +7) o con números sin signo (por ejemplo, de 0 hasta 15).

números con signo vs. números sin signo

Presentamos una representación gráfica del sistema de complemento a dos de 4 bits en la figura 2-3. Podemos volver a etiquetar esa figura como se ilustra en la figura 2-4 para obtener una representación de los números sin signo de 4 bits. Las combinaciones binarias ocupan las mismas posiciones sobre la rueda, y un número se suma moviendo la flecha un número correspondiente de posiciones en el sentido de giro de las manecillas del reloj, y se resta desplazando la flecha en sentido contrario al giro de las manecillas del reloj.

Puede verse cómo una operación de suma excede el intervalo del sistema numérico sin signo de 4 bits en la figura 2-4, cuando la flecha se mueve en el sentido de giro de las manecillas del reloj a través de la discontinuidad entre 0 y 15. En este caso se dice que ocurre un *acarreo* hacia afuera de la posición del bit más significativo.

acarreo

Asimismo una operación de resta excede el intervalo del sistema numérico si la flecha se mueve en sentido contrario al giro de las manecillas del reloj, a través de la discontinuidad. En este caso se dice que se presenta un *traslado o préstamo* fuera de la posición del bit más significativo.

traslado o préstamo

De la figura 2-4 también es evidente que podemos restar un número sin signo n al contar *en el sentido de las manecillas del reloj* $16 - n$ posiciones. Esto equivale a *sumar* el complemento a dos de 4 bits de n. La resta produce un préstamo si la suma correspondiente del complemento a dos *no* produce un acarreo.

En resumen, en la suma sin signo, el acarreo o préstamo en la posición del bit más significativo indica un resultado fuera de intervalo. En el caso con signo que considera la suma de complemento a dos, la condición de desbordamiento que se definió anteriormente indica un resultado fuera de intervalo. El acarreo desde la posición del bit más significativo es irrelevante en la suma con signo, ya que el desbordamiento puede o no puede ocurrir independientemente de que se presente o no un acarreo.

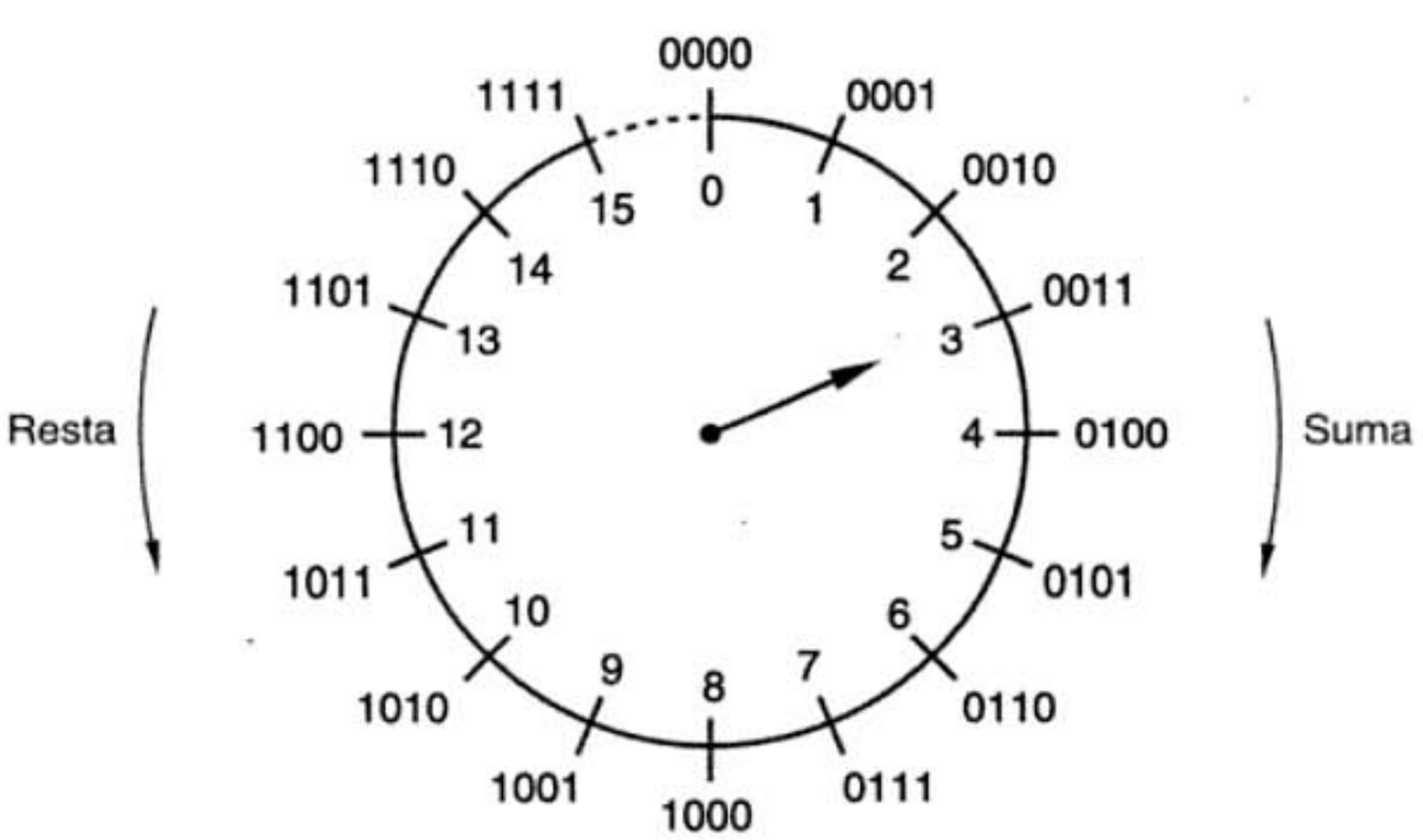


Figura 2-4
Una representación de conteo modular de números sin signo de 4 bits.

*2.7 Suma y resta de complemento a uno

Si analiza la tabla 2-6 encontrará la explicación de la regla que indica cómo sumar números de complemento a uno. Si comenzamos en 1000_2 (-7_{10}) y contamos en forma ascendente, obtenemos un número sucesivo de complemento a unos al sumar 1 al anterior, *excepto* en la transición de 1111_2 (0 negativo) a 0001_2 ($+1_{10}$). Para mantener el conteo apropiado, debemos agregar 2 en vez de 1 dondequiera que nuestro conteo rebase 1111_2. Esto sugiere una técnica para sumar números de complemento a uno: efectuar una suma binaria estándar, pero agregar un 1 extra cada vez que rebasemos 1111_2. El conteo que rebasa 1111_2 durante una suma puede detectarse al observar el acarreo de salida de la posición de signo. Por lo tanto, la regla para sumar números de complemento a uno puede establecerse de manera muy simple:

suma de complemento a uno

- Realice una suma binaria estándar; si hay un acarreo de salida de la posición de signo, agregue 1 al resultado.

acarreo de redondeo final

Esta regla se conoce a menudo como *acarreo de redondeo final* (*end-around carry*). A continuación se presentan varios ejemplos de suma de complemento a uno; los últimos tres incluyen un acarreo de redondeo final:

+3	0011	+4	0100	+5	0101
+ +4	+ 0100	+ −7	+ 1000	+ −5	+ 1010
+7	0111	−3	1100	−0	1111
−2	1101	+6	0110	−0	1111
+ −5	+ 1010	+ −3	+ 1100	+ −0	+ 1111
−7	10111	+3	10010	−0	11110
	+ 1		+ 1		+ 1
	1000		0011		1111

Siguiendo la regla de suma de dos pasos, la adición de un número y su complemento a uno produce un 0 negativo. De hecho, una operación de suma que utiliza esta regla nunca producirá un 0 positivo, a menos que ambos sumandos sean 0 positivos.

resta de complemento a unos

Como sucede en el complemento a dos, la manera más fácil de hacer la resta de complemento a uno es complementar el sustraendo y sumar. Las reglas de desbordamiento para la suma y resta del complemento a uno son las mismas que para el complemento a dos.

La tabla 2-7 resume las reglas que presentamos en ésta y las secciones anteriores para la negación, suma y resta en sistemas numéricos binarios.

Tabla 2-7 Resumen de las reglas de suma y resta para números binarios.

Sistema numérico	*Reglas de la suma*	*Reglas de la negación*	*Reglas de la resta*
Sin signo	Sumar los números. El resultado se encuentra fuera de intervalo si se presenta un acarreo fuera del MSB.	No se aplican.	Reste el sustraendo del minuendo. El resultado se encuentra fuera de intervalo si se presenta un préstamo fuera del MSB.
Magnitud con signo	(mismo signo) Sume las magnitudes; ocurre un desbordamiento si se presenta un acarreo fuera del MSB; el resultado tiene el mismo signo. (signos opuestos) Reste la magnitud más pequeña de la más grande; el desbordamiento es imposible; el resultado tiene el signo de la más grande.	Cambie el bit de signo del número.	Cambie el bit de signo del sustraendo y proceda como en la suma.
Complemento a dos	Sumar, ignorando cualquier acarreo de salida del MSB. Ocurre un desbordamiento si los acarreos de entrada y de salida del MSB son diferentes.	Complemente todos los bits del número; agregue 1 al resultado.	Complemente todos los bits del sustraendo y sume al minuendo con un acarreo inicial de 1.
Complemento a unos	Sume; si hay un acarreo fuera del MSB, agregue 1 al resultado. Se presenta un desbordamiento si los acarreos de entrada y de salida del MSB son diferentes.	Complemente todos los bits del número.	Complemente todos los bits del sustraendo y proceda como en el caso de la suma.

*2.8 Multiplicación binaria

multiplicación con corrimiento y suma

multiplicación binaria sin signo

En la escuela primaria aprendimos a multiplicar mediante la suma de una lista de multiplicandos trasladados, que se calculaban de acuerdo con los dígitos del multiplicador. Se puede utilizar el mismo método para obtener el producto de dos números binarios sin signo. La formación de los multiplicandos trasladados es trivial en la multiplicación binaria, puesto que los únicos valores posibles de los dígitos del multiplicador son 0 y 1. A continuación presentamos un ejemplo:

```
   11          1011     multiplicando
 × 13        × 1101     multiplicador
 ----        ------
   33          1011
   11          0000     multiplicandos desplazados
 ----        ------
  143         1011
             1011
             --------
             10001111   producto
```

producto parcial

En vez de listar todos los multiplicandos trasladados y posteriormente sumar, en un sistema digital es más conveniente sumar cada multiplicando trasladado como si fuera creado para un *producto parcial*. Aplicando esta técnica al ejemplo anterior, se utilizan cuatro sumas y productos parciales para multiplicar números de 4 bits:

11	1011	multiplicando
× 13	× 1101	multiplicador
	0000	producto parcial
	1011	multiplicando desplazado
	01011	producto parcial
	0000↓	multiplicando desplazado
	001011	producto parcial
	1011↓↓	multiplicando desplazado
	0110111	producto parcial
	1011↓↓↓	multiplicando desplazado
	10001111	producto

En general, cuando multiplicamos un número de n bits por un número de m bits, para expresar el producto resultante se requieren como máximo $n + m$ bits. El algoritmo de desplazamiento y suma requiere m productos y sumas parciales para obtener el resultado, pero la primera suma es trivial, puesto que el primer producto parcial es cero. Aunque el primer producto parcial tiene solamente n bits significativos, después de cada paso de suma, el producto parcial gana un bit significativo más, ya que cada suma puede producir un acarreo. Al mismo tiempo, cada paso produce un bit de producto parcial adicional, comenzando con el que está más a la derecha y continuando hacia la izquierda, que no cambia. El algoritmo de desplazamiento y suma puede realizarse mediante un circuito digital que incluirá un registro de corrimiento, un sumador y la lógica de control, como se verá en la sección 8.7.2.

multiplicación con signo

La multiplicación de números con signo puede realizarse aplicando la multiplicación sin signo y las reglas de escuela habituales: realice una multiplicación sin signo de las magnitudes y haga el producto positivo si los operandos tienen el mismo signo, negativo si tienen signos diferentes. Esto es muy conveniente en sistemas de magnitud con signo, puesto que el signo y la magnitud están separados.

multiplicación de complemento a dos

En el sistema de complemento a dos, la obtención de la magnitud de un número negativo y la conversión a negativo del producto sin signo son operaciones poco triviales. Esto nos conduce a buscar una manera más eficiente de realizar la multiplicación de complemento a dos, la cual se describe a continuación.

Conceptualmente, la multiplicación sin signo se realiza por medio de una secuencia de sumas sin signo de los multiplicandos trasladados; en cada paso, el desplazamiento del multiplicando corresponde a la ponderación del bit multiplicador. Los bits en un número de complemento a dos tienen las mismas ponderaciones que en un número sin signo, excepto para el MSB, que tiene una ponderación negativa (véase la sección 2.5.4). De este modo, podemos realizar la multiplicación de complemento a dos mediante una secuencia de sumas de complemento a dos (de multiplicandos desplazados), excepto para el último paso, donde el multiplicando trasladado (que corresponde al MSB

del multiplicador) debe convertirse a negativo antes de sumarlo al producto parcial. Nuestro ejemplo anterior se repite a continuación, en este caso el multiplicador y el multiplicando aparecen como números de complemento a dos:

−5	1011	multiplicando
× −3	× 1101	multiplicador
	00000	producto parcial
	11011	multiplicando desplazado
	111011	producto parcial
	00000↓	multiplicando desplazado
	1111011	producto parcial
	11011↓↓	multiplicando desplazado
	11100111	producto parcial
	00101↓↓↓	multiplicando convertido a negativo y desplazado
	00001111	producto

El manejo de los MSB es algo delicado porque ganamos un bit significativo en cada paso y trabajamos con números que tienen signo. Por consiguiente, antes de sumar cada multiplicando trasladado y producto parcial de k-bits, los cambiamos a $k + 1$ bits significativos por extensión de signo, como se muestra en negritas en el desarrollo de la operación. Cada suma resultante tiene $k + 1$ bits y se ignora cualquier acarreo fuera del MSB de la suma del $k + 1$ bit.

*2.9 División binaria

división de desplazamiento y resta

El algoritmo más simple de división binaria se basa en el método de desplazamiento y resta que aprendimos cuando estudiamos en la primaria. La tabla 2-8 proporciona ejemplos de este método para números binarios y decimales sin signo. En ambos casos, comparamos mentalmente el dividendo reducido con múltiplos del divisor para determinar cuál múltiplo del divisor desplazado se debe restar. En el caso decimal, primero elegimos 11 como el mayor múltiplo de 11 menor que 21 y luego elegimos 99 como el

división sin signo

	19		10011	cociente
11	)217	1011	)11011001	dividendo
	11		1011	divisor desplazado
	107		0101	dividendo reducido
	99		0000	divisor desplazado
	8		1010	dividendo reducido
			0000	divisor desplazado
			10100	dividendo reducido
			1011	divisor desplazado
			10011	dividendo reducido
			1011	divisor desplazado
			1000	residuo

Tabla 2-8
Ejemplo de división larga.

mayor múltiplo menor que 107. En el caso binario, la elección es más sencilla, puesto que las únicas opciones son cero y el mismo divisor.

Los métodos de división para números binarios son un tanto complementarios a los métodos de multiplicación binaria. Un típico algoritmo de división toma un dividendo de $(n + m)$ bits y un divisor de n bits, para producir un cociente de m bits y un residuo de n bits. El *desbordamiento* de la división se presentará cuando el divisor sea cero o el cociente necesita más de m bits para expresarse. En la mayoría de los circuitos de división de las computadoras, $n = m$.

desbordamiento de división

división con signo

La división de números con signo puede llevarse a cabo aplicando la división sin signo y las reglas habituales: efectuar una división sin signo de las magnitudes y hacer el cociente positivo si los operandos tienen el mismo signo, y negativo si tienen signos diferentes. Se debería dar al residuo el mismo signo que al dividendo. Como en el caso de la multiplicación, existen técnicas especiales para realizar directamente la división sobre números de complemento a dos; con cierta frecuencia se implementan estas técnicas en circuitos de división para computadoras (véanse las referencias).

2.10 Códigos binarios para números decimales

Aun cuando los números binarios son los más apropiados para los cálculos internos de un sistema digital, la mayoría de la gente prefiere trabajar con números decimales. Como resultado, las interfaces externas de un sistema digital pueden leer o presentar números decimales, y en realidad algunos dispositivos digitales procesan directamente números decimales.

La necesidad humana de representar los números decimales no cambia la naturaleza básica de los circuitos electrónicos digitales; éstos procesan las señales que pueden tener uno de dos estados que llamamos 0 y 1. Por tanto, un número decimal se representa en un sistema digital mediante una cadena de bits, donde las diferentes combinaciones de los valores de bits en la cadena representan diferentes números decimales. Por ejemplo, si empleamos una cadena de 4 bits para representar un número decimal, podemos asignar la combinación de bits 0000 al dígito decimal 0, 0001 al 1, 0010 al 2, y así sucesivamente.

Un conjunto de cadenas de n bits en el cual diferentes cadenas de bits representan diferentes números u otras cosas se denomina *código*. Una combinación particular de valores de n bits se conoce como *palabra de código*. Como veremos en los ejemplos de códigos decimales en esta sección, puede o no existir una relación aritmética entre los valores de bits en una palabra de código y lo que representa. Por eso, un código que utiliza cadenas de n bits no necesita contener palabras de código válidas de 2^n.

código

palabra de código

Como mínimo se necesitan cuatro bits para representar los diez dígitos decimales. Existen miles y miles de millones de maneras diferentes para elegir 10 palabras de código de 4 bits; sin embargo, la tabla 2-9 muestra los códigos decimales más comunes.

Quizás el código decimal más "natural" sea el *decimal codificado en binario* (*BCD*), el cual codifica los dígitos del 0 al 9 mediante sus representaciones binarias sin signo de 4 bits, desde 0000 hasta 1001. Las palabras de código restantes, de 1010 a 1111, no se utilizan. Las conversiones entre las representaciones BCD y decimal son triviales, implican la sustitución directa de cuatro bits por cada dígito decimal. Algunos programas de computadora colocan dos dígitos BCD en un byte de 8 bits en la *representación BCD*

decimal codificado en binario (BCD)

representación BCD empaquetada

Tabla 2-9 Códigos decimales.

Dígito decimal	*BCD (8421)*	*2421*	*Exceso-3*	*Biquinario*	*1 de 10*
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
Palabras de código sin utilizar					
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	...	...

empaquetada; así, un byte puede representar los valores desde 0 hasta 99 en oposición al intervalo de 0 a 255 para un número binario normal de 8 bits sin signo. Se pueden obtener los números BCD con cualquier cantidad de dígitos mediante el uso de un byte para cada par de dígitos.

Como sucede con los números binarios, existen muchas representaciones posibles de los números BCD negativos. Los números BCD con signo tienen una posición de dígito extra para el signo. Tanto la representación de magnitud con signo como las

representaciones del complemento de 10 son populares. En BCD de magnitud con signo, la codificación de la cadena de bit de signo es arbitraria; en el complemento de 10, el dato 0000 indica el signo más y 1001 indica el signo menos.

suma BCD

La suma de los dígitos BCD es semejante la adición de números binarios sin signo de 4 bits, excepto que debe hacerse una corrección si un resultado excede 1001. El resultado se corrige agregando 6; como se muestra en los siguientes ejemplos:

5	0101		4	0100	
+ 9	+ 1001		+ 5	+ 0101	
14	1110		9	1001	
	+ 0110	— **corrección**			
10 + 4	**1** 0100				
8	1000		9	1001	
+ 8	+ 1000		+ 9	+ 1001	
−16	**1** 0000		18	**1** 0010	
	+ 0110	— **corrección**		+ 0110	— **corrección**
10 + 6	**1** 0110		10 + 8	**1** 1000	

Nótese que la suma de dos dígitos BCD produce un acarreo en la siguiente posición de dígito, si la suma binaria inicial o la suma de factor de corrección produce un acarreo. Muchas computadoras realizan aritmética de BCD empaquetado aplicando instrucciones especiales que manejan la corrección de acarreo en forma automática.

código ponderado

El decimal codificado en binario es un *código ponderado,* puesto que cada dígito decimal puede obtenerse a partir de su palabra de código asignando un peso fijo a cada bit de palabra de código. Los pesos para los bits BCD son 8, 4, 2 y 1, y por esta razón el código se denomina en ocasiones *código 8421*. Otro conjunto de pesos da como resultado el *código 2421* que se indica en la tabla 2-9. Este código tiene la ventaja de que es *autocomplementado,* es decir, se puede obtener la palabra de código para el complemento a nueve de cualquier dígito al complementar los bits individuales de la palabra de código del dígito.

código 8421
código 2421
código autocomplementado

código de exceso 3

La tabla 2-9 muestra otro código autocomplementado, el *código de exceso 3*. Aunque este código no está ponderado, tiene una relación aritmética con el código BCD (la palabra de código para cada dígito decimal es la correspondiente palabra de código BCD más 0011_2). Como las palabras del código siguen una secuencia de conteo binaria estándar, pueden hacerse fácilmente contadores binarios estándar para contar en el código de exceso 3, como mostraremos en la figura 8-37 (pág. 700).

código biquinario

Los códigos decimales pueden tener más de cuatro bits; por ejemplo, el *código biquinario* de la tabla 2-9 utiliza siete. Los primeros dos bits en una palabra de código indican si el número se encuentra en el intervalo 0-4 o 5-9, mientras que los cinco últimos indican cuál de los cinco números en el intervalo seleccionado está representado.

Una ventaja potencial que se obtiene al utilizar más del número mínimo de bits en un código es la propiedad de detección de errores. En el código biquinario, si cualquier bit en una palabra de código se cambia accidentalmente al valor opuesto, la palabra de

código resultante no representará un dígito decimal y por consiguiente se puede señalar como un error. Fuera de 128 posibles palabras de código de 7 bits, solamente 10 son válidas y reconocidas como dígitos decimales; el resto se puede señalar como error, si es que aparecen.

Un *código 1 de 10*, tal como el que se muestra en la última columna de la tabla 2-9, es la codificación más escasa para dígitos decimales, utilizando 10 de 1024 posibles palabras de código de 10 bits. *código 1 de 10*

2.11 Código Gray

En las aplicaciones electromecánicas de los sistemas digitales (tales como herramientas mecánicas, sistemas de frenado para automóviles y fotocopiadoras) a veces es necesario que un sensor de entrada produzca un valor digital que indique una posición mecánica. Por ejemplo, la figura 2-5 es un esquema conceptual de un disco de codificación y un conjunto de contactos que producen uno de ocho valores codificados en binario de 3 bits, dependiendo de la posición rotacional del disco. Las áreas oscuras del disco se conectan a una fuente de señal que corresponde al 1 lógico y las áreas claras no se conectan, lo que los contactos interpretan como un 0 lógico.

El codificador de la figura 2-5 tiene un problema cuando el disco se coloca en ciertas fronteras entre las regiones. Por ejemplo, considere la frontera entre las regiones 001 y 010 del disco; dos de los bits codificados cambian aquí. ¿Qué valor producirá el codificador si el disco se posiciona justo sobre la frontera teórica? Puesto que nos encontramos sobre el borde, tanto 001 como 010 son aceptables. Sin embargo, ya que el ensamble mecánico no es perfecto, los dos contactos de la derecha pueden tocar una región "1", dando una lectura incorrecta de 011. De igual forma, se puede obtener una lectura de 000. En general, esta clase de problema puede presentarse en cualquier frontera donde cambia más de un bit. Los peores problemas ocurren cuando cambian los tres bits, como en las fronteras 000-111 y 011-100.

El problema del disco codificado puede resolverse mediante la creación de un código digital en el cual solamente cambie un bit entre cada par de palabras de código sucesivas. Un código de esta clase se denomina *código Gray*; la tabla 2-10 muestra un código Gray de 3 bits. Hemos rediseñado el disco codificado usando este código como *código Gray*

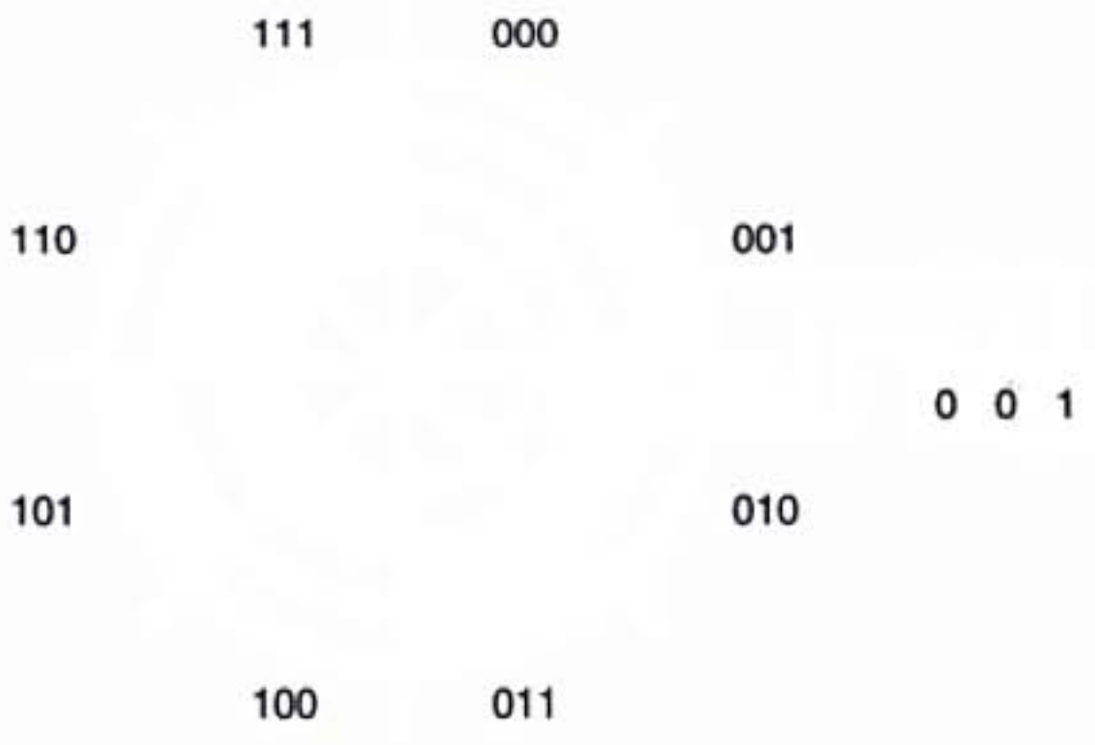


Figura 2-5
Disco mecánico de codificación que utiliza un código binario de 3 bits.

Tabla 2-10
Una comparación de código binario de 3 bits y código Gray.

Número decimal	*Código binario*	*Código Gray*
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

se muestra en la figura 2-6. Solamente cambia un bit del nuevo disco en cada borde, de modo que las lecturas de las líneas fronterizas nos dan un valor que está en cualquiera de los lados del borde.

Existen dos maneras convenientes de construir un código Gray con cualquier número deseado de bits. El primer método está basado en el hecho de que el código Gray es un *código reflejado*; puede ser definido (y construido) de manera recursiva utilizando las reglas siguientes:

código reflejado

1. Un código Gray de 1 bit tiene dos palabras de código, 0 y 1.
2. Las primeras 2^n palabras de código de un código Gray de $(n + 1)$ bits son iguales a las palabras de código de un código Gray de n bits, escritas en orden con un 0 principal agregado.
3. Las últimas 2^n palabras de código de un código Gray de $(n + 1)$ bits son iguales a las palabras de código de un código Gray de n bits, pero escritas en orden inverso con un 1 principal agregado.

Si trazamos una línea entre los renglones 3 y 4 de la tabla 2-10, podemos ver que las reglas 2 y 3 son verdaderas para el código Gray de 3 bits. Por supuesto, para construir un código Gray de n bits para un valor arbitrario de n con este método, también debemos construir un código Gray para cada longitud más pequeña que n.

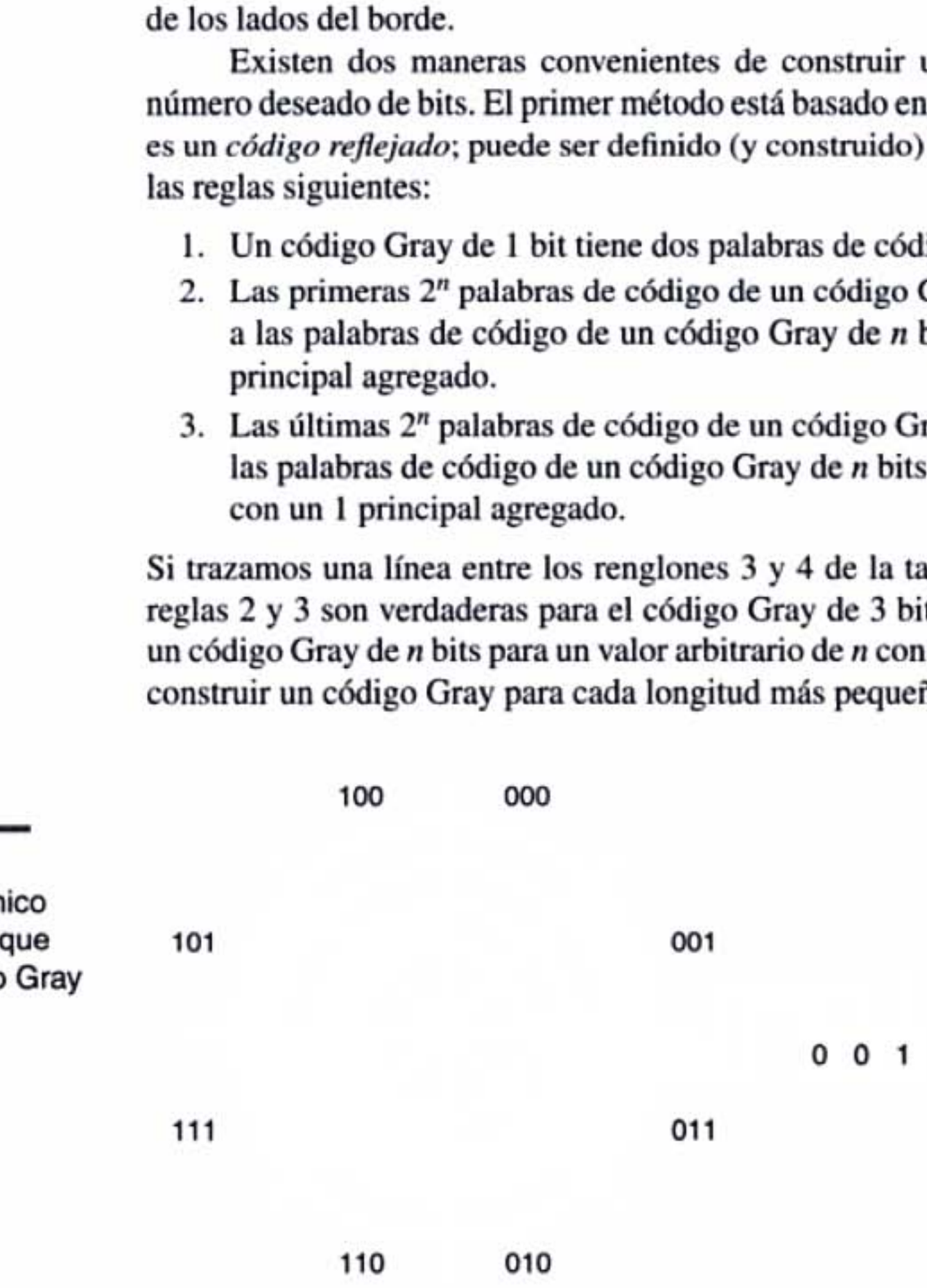


Figura 2-6
Un disco mecánico de codificación que utiliza un código Gray de 3 bits.

El segundo método nos permite derivar una palabra de código (del código Gray) de n bits directamente de la correspondiente palabra de código binaria de n bits:

1. Los bits de una palabra de código binaria de n bits o código Gray se numeran de derecha a izquierda, desde 0 hasta $n - 1$.
2. El bit i de una palabra de código del código Gray es 0 si los bits i e $i + 1$ de la correspondiente palabra de código binario son los mismos, si no el bit i es 1. (Cuando $i + 1 = n$, el bit n de la palabra de código binario se considera que es 0.)

De nueva cuenta, el examen de la tabla 2-10 muestra que esto es verdadero para el código Gray de 3 bits.

*2.12 Códigos de carácter

Como mostramos en la sección precedente, una cadena de bits no necesita representar un número y en realidad la mayor parte de la información que procesan las computadoras no es numérica. El tipo más común de datos no numéricos es el *texto*, cadenas de caracteres de algún conjunto de caracteres. Cada carácter es representado en la computadora por una cadena de bits de acuerdo a una convención establecida.

texto

El código de caracteres más comúnmente utilizado es el *ASCII* (pronunciado *ASSKI, American Standard Code for Information Interchange*), el Código Estandar Americano para Intercambio de Información, por sus siglas en inglés. El ASCII representa cada carácter con una cadena de 7 bits, y produce un total de 128 caracteres diferentes, los cuales se indican en la tabla 2-11. El código contiene el alfabeto en mayúsculas y minúsculas, números, signos de puntuación y diversos caracteres de control no imprimibles. De este modo, la cadena de texto "Yeccch!" está representada por una lista de aspecto bastante inocuo de siete números de 7 bits:

ASCII

1011001 1100101 1100011 1100011 1100011 1101000 0100001

2.13 Códigos para acciones, condiciones y estados

Los códigos que hemos descrito hasta ahora se emplean generalmente para representar cosas que usted probablemente consideraría como "datos" (cosas como números, posiciones y caracteres). Los programadores saben que docenas de diferentes tipos de datos pueden ser utilizados en un solo programa de computadora.

En el diseño de sistemas digitales, con frecuencia hallamos aplicaciones sin datos donde una cadena de bits debe emplearse para controlar una acción, para señalar una condición, o para representar el estado actual del hardware. Probablemente el tipo de código más comúnmente empleado para una aplicación de esta clase es un simple código binario. Si hay n diferentes acciones, condiciones o estados, podemos representarlos con un código binario de b bits con $b = \lceil \log_2 n \rceil$ bits. (Los símbolos $\lceil\ \rceil$ indican la *función techo*, el entero más pequeño mayor o igual que la cantidad encerrada entre ellos. De este modo, b es el entero más pequeño tal que $2^b \geq n$.)

$\lceil\ \rceil$
función techo

Por ejemplo, considere un simple controlador de semáforos. Las señales en la intersección de una calle en dirección norte-sur (N-S) y una de este a oeste (E-W) pueden estar en cualquiera de los seis estados que se indican en la tabla 2-12. Estos estados pueden estar codificados en tres bits, como se muestra en la última columna de la tabla. Solamente seis de las ocho posibles palabras de código de 3 bits se utilizan, y la asignación de las seis palabras de

Tabla 2-11 Código Estándar Americano para Intercambio de Información (ASCII), estándar No. X3.4-1968 del American National Standards Institute.

		$b_6b_5b_4$ (columna)							
$b_3b_2b_1b_0$	Renglón (hex)	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000	0	NUL	DLE	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS		=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Códigos de control

NUL	Nulo (Null)	DLE	Escape de enlace de datos (Data link escape)
SOH	Comienzo de encabezado (Start of heading)	DC1	Control de dispositivo 1 (Device control 1)
STX	Comienzo de texto (Start of text)	DC2	Control de dispositivo 2 (Device control 2)
ETX	Fin de texto (End of text)	DC3	Control de dispositivo 3 (Device control 3)
EOT	Fin de transmisión (End of transmission)	DC4	Control de dispositivo 4 (Device control 4)
ENQ	Pregunta (Enquiry)	NAK	Admisión negativa (Negative acknowledge)
ACK	Admisión (Acknowledge)	SYN	Sincronía (Synchronize)
BEL	Campana (Bell)	ETB	Fin de bloque transmitido (End transmitted block)
BS	Retroceso (Backspace)	CAN	Cancelación (Cancel)
HT	Tabulador horizontal (Horizontal tab)	EM	Fin del medio (End of medium)
LF	Avance de línea (Line feed)	SUB	Sustitución (Substitute)
VT	Tabulador vertical (Vertical tab)	ESC	Escape (Escape)
FF	Avance de página (Form feed)	FS	Separador de archivo (File separator)
CR	Retorno de carro (Carriage return)	GS	Separador de grupo (Group separator)
SO	Corrimiento hacia fuera (Shift out)	RS	Separador de registro (Record separator)
SI	Corrimiento hacia dentro (Shift in)	US	Separador de unidad (Unit separator)
SP	Espacio (Space)	DEL	Eliminación o borrado (Delete or rubout)

Tabla 2-12 Estados en un controlador de semáforos.

	Luces						
Estado	*Verde N-S*	*Amarillo N-S*	*Rojo N-S*	*Verde E-W*	*Amarillo E-W*	*Rojo E-W*	*Palabra de código*
N-S siga	ENCEN-DIDO	apagado	apagado	apagado	apagado	ENCEN-DIDO	000
N-S espere	apagado	ENCEN-DIDO	apagado	apagado	apagado	ENCEN-DIDO	001
N-S alto	apagado	apagado	ENCEN-DIDO	apagado	apagado	ENCEN-DIDO	010
E-W siga	apagado	apagado	ENCEN-DIDO	ENCEN-DIDO	apagado	apagado	100
E-W espere	apagado	apagado	ENCEN-DIDO	apagado	ENCEN-DIDO	apagado	101
E-W alto	apagado	apagado	ENCEN-DIDO	apagado	apagado	ENCEN-DIDO	110

código elegidas a los estados es arbitraria, de modo que muchas otras codificaciones son posibles. Un diseñador digital experimentado elige una codificación particular para minimizar el costo del circuito o para optimizar algún otro parámetro (como el tiempo de diseño: no hay necesidad de intentar miles y miles de millones de posibles codificaciones).

Otra aplicación del código binario se ilustra en la figura 2-7(a). Aquí, tenemos un sistema con n dispositivos, cada uno de los cuales puede realizar una acción en particular. Las características de los dispositivos son tales que pueden ser habilitados para funcionar solamente uno a la vez. La unidad de control produce una palabra de "selección de dispositivo" codificada en binario con $\lceil \log_2 n \rceil$ bits para indicar cuál dispositivo se encuentra habilitado en cualquier tiempo. La palabra de código "selectora de dispositivo" se aplica a cada dispositivo, a su vez ésta se compara con la "ID propia del dispositivo" para determinar si se encuentra habilitado. Aunque sus palabras de código tienen el mínimo número de bits, un código binario no siempre es la mejor elección para codificar acciones, condiciones o estados. La figura 2-7(b) muestra cómo controlar n dispositivos con un *código 1 de n*, un código de n bits en el cual las palabras de código válidas tienen un bit igual a 1 y el resto de los bits son iguales a 0. Cada bit de la palabra del código 1 de n se conecta directamente a la entrada de habilitación del dispositivo correspondiente. Esto simplifica el diseño de los dispositivos, puesto que ellos ya no tienen ID de dispositivo; solamente necesitan un bit de entrada de "habilitación".

código 1 de n

Las palabras de código de un código 1 de 10 se indican en la tabla 2-9. En ocasiones una palabra "llena" de ceros puede incluirse en un código 1 de n, para indicar que ningún dispositivo se encuentra seleccionado. Otro código común es un *código 1 de n invertido*, en el cual las palabras de código válidas tienen un bit 0 y el resto de los bits son iguales a 1.

código 1 de n invertido

En sistemas complejos, puede emplearse una combinación de las técnicas de codificación. Por ejemplo, considere un sistema similar al que se muestra en la figura 2-7(b), en el que cada uno de los n dispositivos contiene hasta s subdispositivos. La unidad de control produciría una palabra de código de selección de dispositivo con un campo codi-

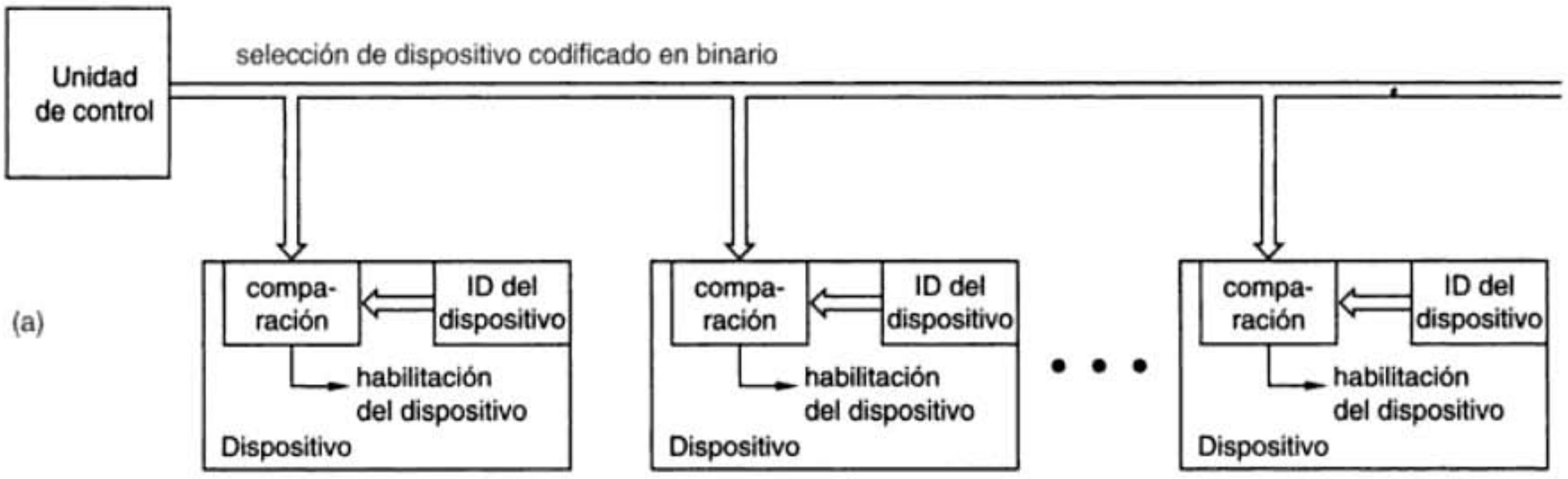


Figura 2-7 Estructura de control para un sistema digital con *n* dispositivos: a) que utiliza un código binario; b) que utiliza un código 1 de *n*.

ficado de 1 de *n* para seleccionar un dispositivo, y un campo codificado en binario de $\lceil \log_2 s \rceil$ bits para seleccionar uno de los *s* subdispositivos del dispositivo seleccionado.

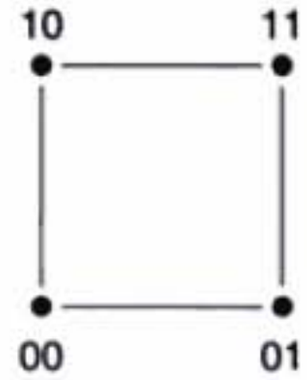
código m de n

Un *código m de n* es una generalización del código 1 de *n*, en el cual las palabras de código válidas tienen *m* bits iguales a 1 y el resto de los bits son iguales a 0. Una palabra de código *m* de *n* puede ser detectada con una compuerta AND de entrada *m*, que produce una salida 1 si todas sus entradas son 1. Esto es simple y barato de hacer, además, para la mayor parte de los valores de *m*, un código *m* de *n* típicamente tiene palabras de código mucho más válidas que un código 1 de *n*. El número total de palabras de código está dado por el coeficiente binomial $\binom{n}{m}$, el cual tiene el valor $\frac{n!}{m! \cdot (n-m)!}$. Así, un código 2 de 4 tiene 6 palabras de código válidas y un código 3 de 10 tiene 120.

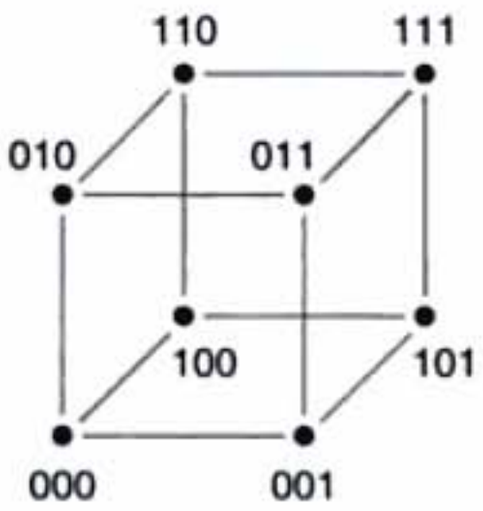
código 8B10B

Una variación importante en el código *m* de *n* es el *código 8B10B* que se utiliza en el estándar Ethernet Gigabit 802.3z. Este código emplea 10 bits para representar 256 palabras de código válidas, o valor de datos de 8 bits. La gran mayoría de las palabras de código utilizan una codificación de 5 de 10. Sin embargo, puesto que $\binom{5}{10}$ es sólo 252, algunas palabras de 4 de 10 y 6 de 10 también se emplean para completar el código de una forma muy interesante; se verá más sobre esto en la sección 2.16.2.

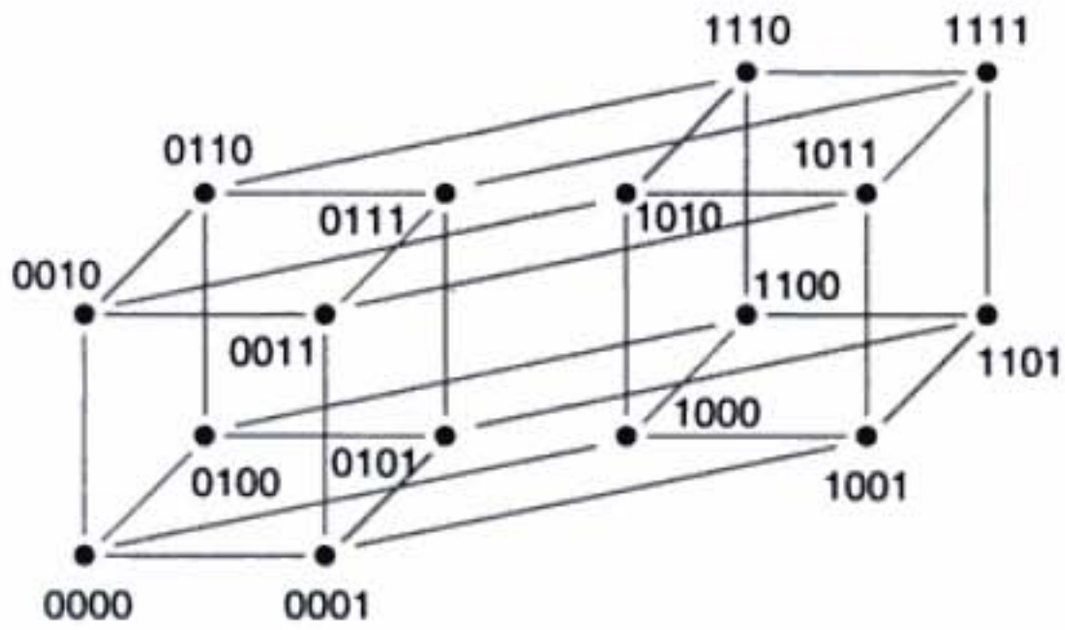
Cubo 1



Cubo 2



Cubo 3



Cubo 4

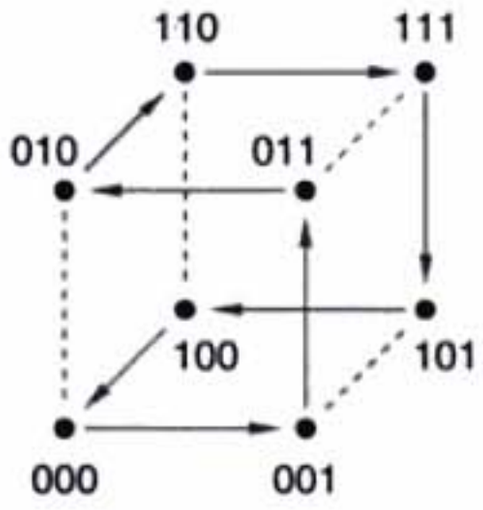
Figura 2-8
Cubos *n* para *n* = 1, 2, 3 y 4.

*2.14 Cubos *n* y distancia

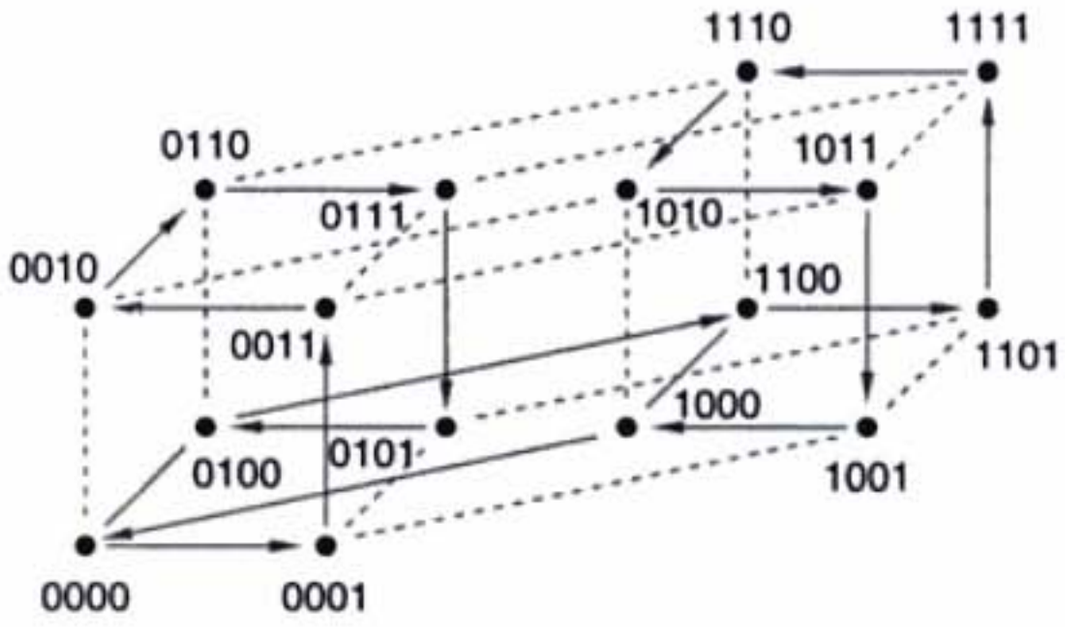
Una cadena de *n* bits puede visualizarse geométricamente, como un vértice de un objeto llamado un *cubo n*. La figura 2-8 muestra cubos *n* para *n* = 1, 2, 3, 4. Un cubo *n* tiene 2^n vértices, cada uno de los cuales se encuentra etiquetado con una cadena de *n* bits. Los bordes se dibujan de manera que cada vértice sea adyacente a otros *n* vértices cuyas etiquetas difieren del vértice dado solamente en un bit. Más allá de *n* = 4, los cubos *n* son realmente difíciles de dibujar.

cubo n

Para valores razonables de *n*, los cubos *n* hacen fácil la visualización de ciertos problemas de codificación y minimización de lógica. Por ejemplo, el problema del diseño de un código Gray de *n* bits es equivalente a encontrar una trayectoria a lo largo de los bordes de un cubo *n*, una trayectoria que visita cada vértice exactamente una vez. Las trayectorias para códigos Gray de 3 y de 4 bits se muestran en la figura 2-9.



(a)



(b)

Figura 2-9
Atravesando cubos *n* en orden de código Gray: a) cubo 3; b) cubo 4.

distancia
distancia de Hamming

Los cubos también proporcionan una interpretación geométrica para el concepto de *distancia*, también llamado *distancia de Hamming*. La distancia entre dos cadenas de n bits es el número de posiciones de bit en las cuales difieren. En términos de un cubo n, la distancia es la mínima longitud de una trayectoria entre los dos vértices correspondientes. Dos vértices adyacentes tienen distancia 1; los vértices 001 y 100 en el cubo 3 tienen distancia 2. El concepto de distancia es crucial en el diseño y comprensión de códigos de detección de errores, discutidos en la siguiente sección.

subcubo m

Un *subcubo m* de un cubo n es un conjunto de 2^m vértices en el cual $n - m$ de los bits tienen el mismo valor en cada vértice y los m bits restantes se hacen cargo de todas las combinaciones restantes 2^m. Por ejemplo, los vértices (000, 010, 100, 110) forman un subcubo 2 del cubo 3. Este subcubo también puede indicarse mediante una cadena simple, xx0, donde "x" indica que un bit en particular es del tipo "sin *importancia*"; cualquier vértice cuyos bits coincidan en las posiciones no x pertenece a este subcubo. El concepto de subcubos es particularmente útil al visualizar algoritmos que minimizan el costo de las funciones de lógica combinacional, como mostraremos en la sección 4.4.

"sin importancia"

*2.15 Códigos para detectar y corregir errores

error
falla
falla temporal
falla permanente

Un *error* en un sistema digital es la alteración de datos a partir de su valor correcto a algún otro valor. Un error es causado por una *falla* física. Las fallas pueden ser temporales o permanentes. Por ejemplo, un rayo cósmico o una partícula alfa pueden ocasionar una falla temporal de un circuito de memoria, cambiando el valor de un bit que está almacenado en ella. Dejar que un circuito se sobrecaliente o que sufra una descarga de electricidad estática puede provocar una falla permanente, de tal manera que nunca volverá a funcionar en forma correcta.

modelo de error
modelo de error independiente
error simple
error múltiple

Los efectos que tienen las fallas en los datos se predicen mediante *modelos de error*. El modelo de error más sencillo que consideraremos aquí, se conoce como el *modelo de error independiente*. Este modelo supone que solamente una falla física afecta a un bit de datos; se dice que los datos corrompidos solamente contienen *un error*. Varias fallas pueden ocasionar *errores* múltiples (dos o más bits con error) pero normalmente se supone que la aparición de errores múltiples es menos probable que la aparición de errores simples.

2.15.1 Códigos de detección de error

Recordemos de nuestras definiciones en la sección 2.10 que un código que utiliza cadenas de n bits no necesita contener 2^n palabras de código válidas; ciertamente éste es el caso para los códigos que ahora consideramos. Un *código de detección de error* tiene la propiedad de que la corrupción o confusión de una palabra de código probablemente producirá una cadena de bits que no sea una palabra de código (una *palabra de no código*).

código de detección de error
palabra de no código

Un sistema que utiliza un código de detección de error genera, transmite y almacena solamente palabras de código. De este modo, los errores en una cadena de bits pueden detectarse mediante una regla simple: si la cadena de bits es una palabra de código, se supone que es correcta; si es una palabra de no código, entonces contiene un error.

Un código de n bits y sus propiedades de detección de error bajo el modelo de error independiente se explican fácilmente en términos de un cubo n. Un código es simplemente un subconjunto de los vértices del cubo n. A fin de que el código detecte todos

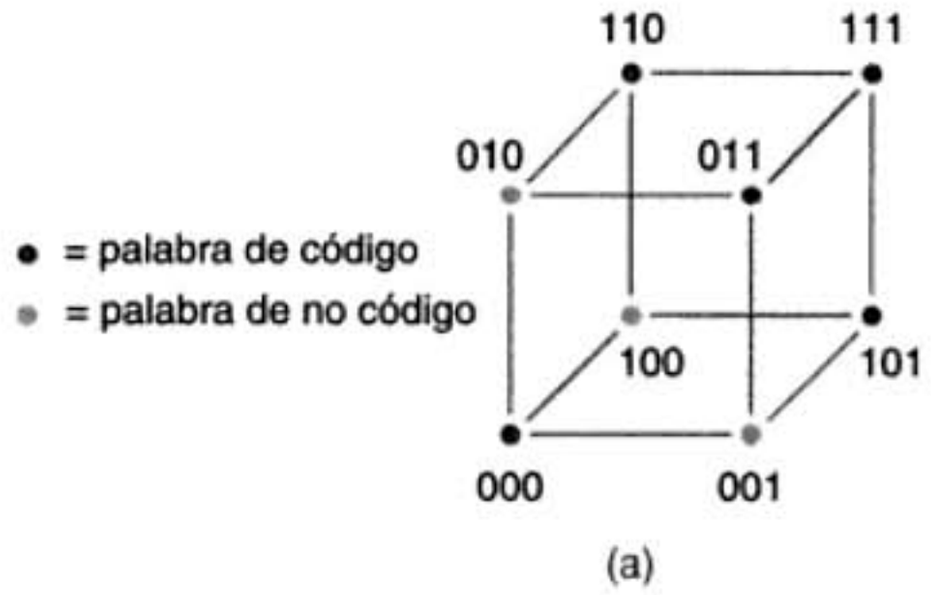


Figura 2-10
Palabras de código en dos diferentes códigos de 3 bits: a) distancia mínima = 1, no detecta todos los errores simples; b) distancia mínima = 2, detecta todos los errores simples.

los errores simples, ningún vértice de palabra de código puede estar adyacente a otro vértice de palabra de código.

Por ejemplo, la figura 2-10(a) muestra un código de 3 bits con cinco palabras de código. La palabra de código 111 se encuentra adyacente a las palabras de código 110, 011 y 101. Puesto que una falla simple podría cambiar 111 a 110, 011 o 101, este código no detecta todos los errores simples. Si hacemos 111 una palabra de no código, obtenemos un código que tiene la propiedad de detección de errores simples, como se ilustra en (b). Ningún error simple puede cambiar una palabra de código en otra.

La capacidad de un código para detectar errores simples puede establecerse en términos del concepto de distancia que se presentó en la sección anterior:

- Un código detecta todos los errores simples si la *distancia mínima* entre todos los pares posibles de palabras de código es 2.

distancia mínima

En general, necesitamos $n + 1$ bits para construir un código de detección de errores simples con 2^n palabras de código. Los primeros n bits de una palabra de código, llamados *bits de información*, pueden ser cualquiera de las 2^n cadenas de n bits. Para obtener un código de mínima distancia 2, agregamos un bit adicional, llamado *bit de paridad*, que se establece a 0 si hay un número par de unos entre los bits de información, y 1 de otro modo. Esto se ilustra en las primeras dos columnas de la tabla 2-13 para un código con tres bits de información. Una palabra de código válida de $(n + 1)$ bits tiene un número par de unos, y este código se llama *código de paridad par*. También podemos construir

bit de información
bit de paridad
código de paridad par

Bits de información	*Código de paridad par*	*Código de paridad impar*
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

Tabla 2-13
Códigos de distancia 2 con tres bits de información.

transmitimos palabras de código y asumamos que las fallas afectan como máximo un bit de cada palabra de código recibida. Entonces una palabra de no código recibida con un error de 1 bit estará más cerca de la palabra de código (que se transmitió originalmente) que de cualquier otra palabra de código. Por consiguiente, cuando recibimos una palabra de no código, podemos *corregir* el error cambiando la palabra de no código a la palabra de código más cercana, como indican las flechas de la figura. La decisión que implica saber cuál palabra de código se transmitió originalmente para producir una palabra que se recibió se conoce como*decodificación*, y el hardware que hace posible esto es un *decodificador* de corrección de errores.

corrección de error

decodificación
decodificador

Un código que se utiliza para corregir errores se denomina *código de corrección de errores*. En general, si un código tiene una distancia mínima de $2c + 1$, puede utilizarse para corregir errores que afecten hasta c bits ($c = 1$ en el ejemplo anterior). Si una distancia mínima de código es $2c + d + 1$, puede emplearse para corregir errores en hasta c bits y detectar errores hasta en d bits adicionales.

código de corrección de errores

Por ejemplo, la figura 2-12(a) muestra un fragmento del cubo n para un código con distancia mínima 4 ($c = 1$, $d = 1$). Los errores de un solo bit que producen palabras de no código 00101010 y 11010011 pueden corregirse. No obstante, un error que produce 10100011 no puede corregirse, porque ningún error de un solo bit puede producir esta palabra de no código, y cualquiera de los dos errores de 2 bits podría producir esta palabra. De modo que el código puede detectar un error de 2 bits, pero no puede corregirlo.

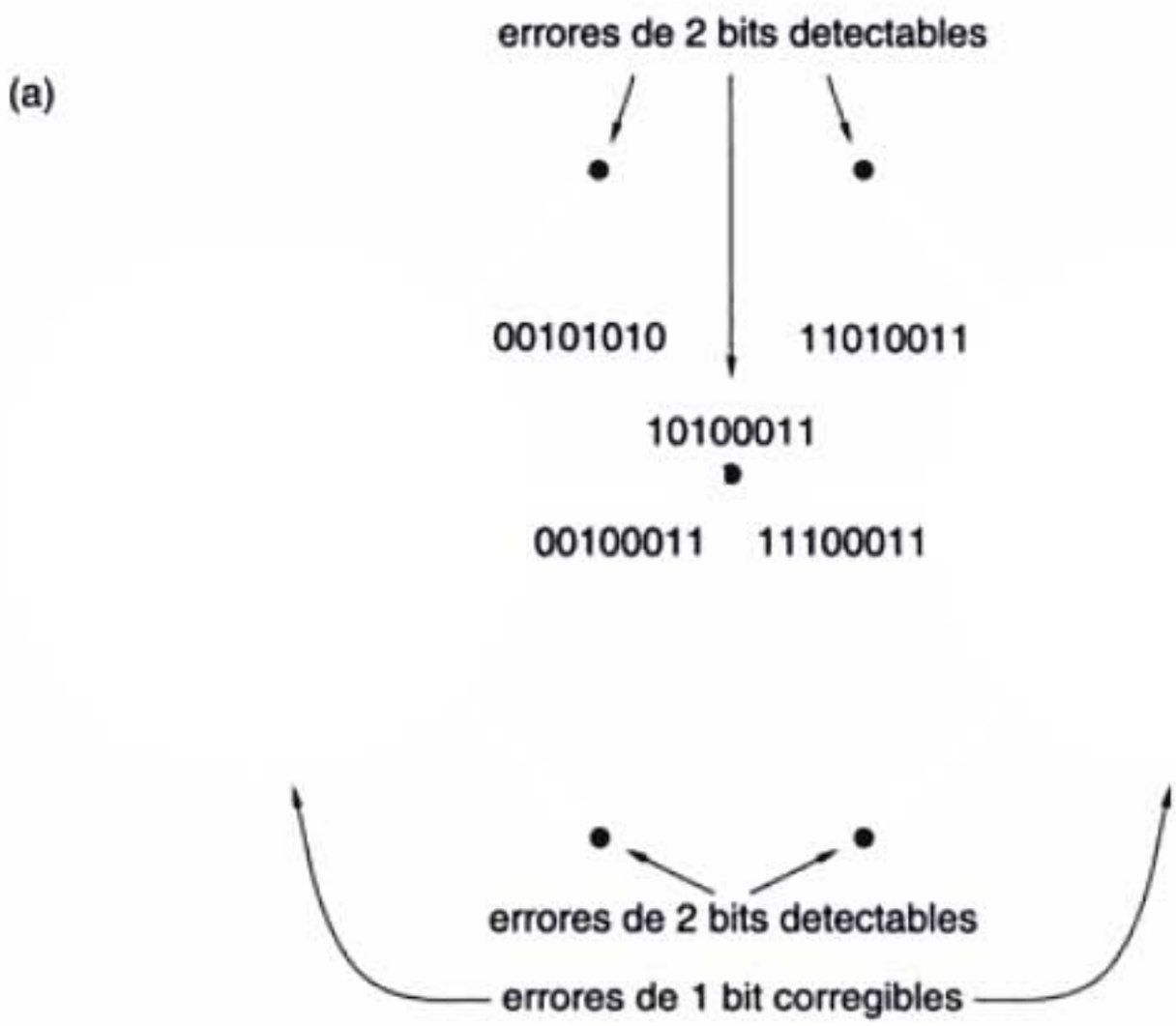
Cuando se recibe una palabra de no código, no sabemos qué palabra de código se transmitió originalmente; solamente sabemos cuál palabra de código se encuentra más cercana a lo que hemos recibido. Así, como se muestra en la figura 2-12(b), un error de 3 bits puede "corregirse". La posibilidad de hacer esta clase de equivocación puede ser aceptable si los errores de 3 bits tienen muy poca probabilidad de ocurrir. Por otro lado, si estamos interesados en los errores de 3 bits, podemos cambiar la política de decodificación para el código. En lugar de intentar corregir errores, únicamente señalamos todas las palabras de no código como errores incorregibles. De este modo, como se muestra en (c), podemos utilizar el mismo código de distancia 4 para detectar errores hasta de 3 bits, pero corregiremos los no errores (c = 0, d = 3).

2.15.3 Códigos de Hamming

En 1950, R. W. Hamming describió un método general para construir códigos con una distancia mínima de 3, que ahora se conocen como *códigos de Hamming*. Para cualquier valor de i, su método produciría un código de $(2^i - 1)$ bits con i bits de verificación y $2^i - 1 - i$ bits de información. Los códigos de distancia 3 con un número más pequeño de bits de información se obtienen al eliminar bits de información de un código de Hamming con un número más grande de bits.

código de Hamming

Las posiciones de bits en una palabra de código de Hamming pueden numerarse desde 1 hasta $2^i - 1$. En este caso, cualquier posición cuyo número sea una potencia de 2 será un bit de verificación, y las posiciones restantes serán bits de información, como se



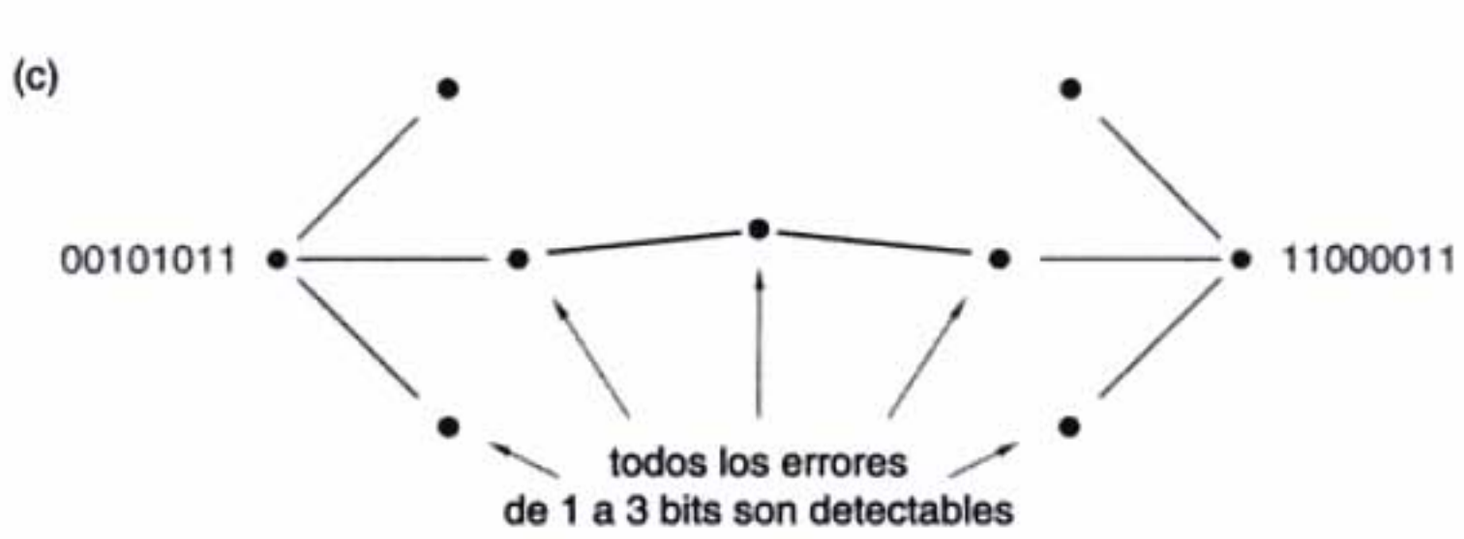


Figura 2-12
Algunas palabras de código y palabras de no código en un código de distancia 4, de 8 bits: a) corrección de errores de 1 bit y detección de errores de 2 bits; b) "corrección" incorrecta de un error de 3 bits; ausencia de corrección de errores pero detección de errores de hasta 3 bits.

matriz de verificación de paridad

especifican mediante una *matriz de verificación de paridad.* Como se muestra en la figura 2-13(a), cada bit de verificación se agrupa con las posiciones de información cuyos números tienen un 1 en el mismo bit cuando se expresan en binario. Por ejemplo, el bit de verificación 2 (010) se agrupa con los bits de información 3 (011), 6 (110) y 7 (111). Para una combinación determinada de bits de información, cada bit de verificación se elige para producir paridad par, es decir, de modo que el número total de unos en su grupo sea par.

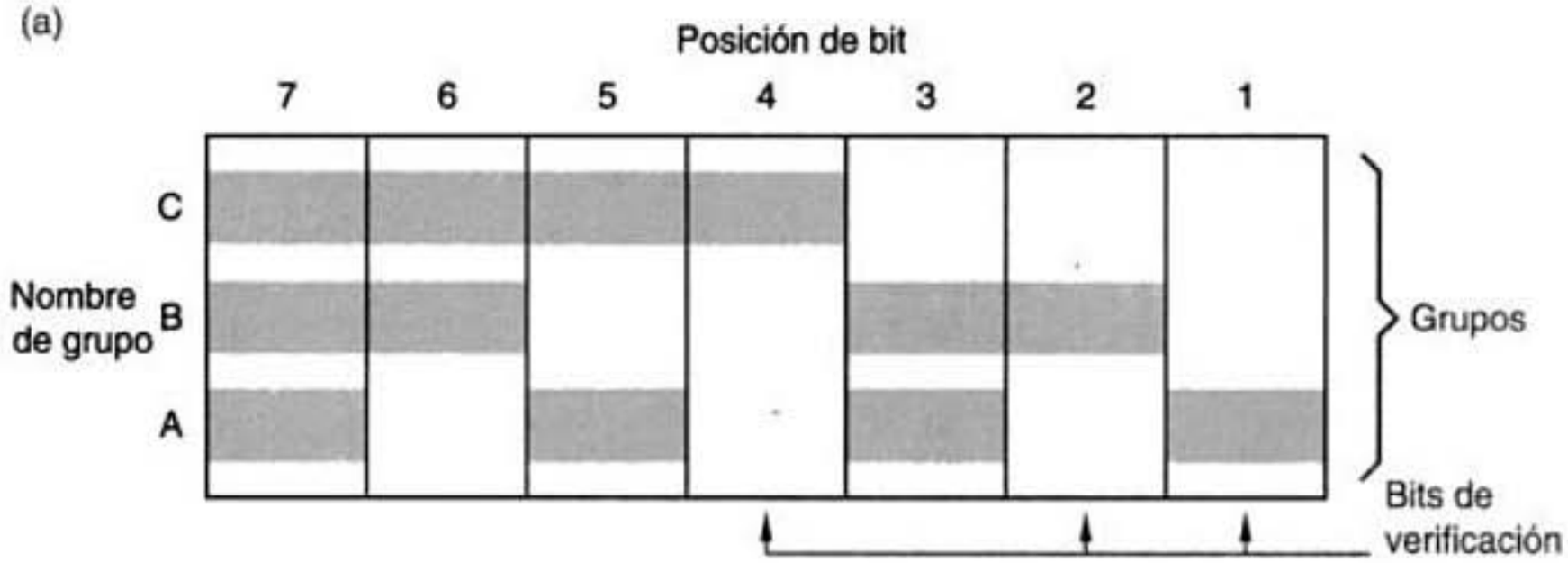


Figura 2-13
Matrices de verificación de paridad para códigos de Hamming de 7 bits: a) con posiciones de bit en orden numérico; b) con bits de verificación y bits de información por separado.

Tradicionalmente, las posiciones de los bits en una matriz de verificación de paridad y las palabras de código resultantes se reacomodan de modo que todos los bits de verificación se encuentren a la derecha, como en la figura 2-13(b). Las primeras dos columnas de la tabla 2-14 muestran las palabras de código resultantes.

Podemos probar que la distancia mínima de un código de Hamming es 3 al verificar que se debe hacer un cambio de 3 bits (como mínimo) a una palabra de código para obtener otra palabra de código. Es decir, probaremos que un cambio de 1 bit o de 2 bits en una palabra de código producirá una palabra de no código.

Si cambiamos un bit de una palabra de código, en la posición j, entonces cambiamos la paridad de cada grupo que contenga la posición j. Puesto que todo bit de información está contenido en por lo menos un grupo, al menos un grupo tiene paridad incorrecta, y el resultado es una palabra de no código.

¿Qué ocurre si cambiamos dos bits, en las posiciones j y k? Los grupos de paridad que contengan ambas posiciones j y k todavía tendrán paridad correcta, puesto que la paridad no se ve afectada cuando se modifican un número par de bits. Sin embargo, puesto que j y k son diferentes, sus representaciones binarias difieren en por lo menos un bit, correspondiendo a uno de los grupos de paridad. Este grupo tiene solamente un bit cambiado, lo que resulta en una paridad incorrecta y una palabra de no código.

Si usted comprende esta prueba, también podrá entender que las reglas de numeración de posición para construir un código de Hamming son una simple consecuencia de la prueba. Para la primera parte de la prueba (errores de 1 bit), requerimos que los números de posición sean distintos de cero. Y para la segunda parte (errores de 2 bits), necesitamos que ningún par de posiciones tengan el mismo número. De esta forma, con un número de

Tabla 2-14 Palabras de código en códigos de Hamming de distancia 3 y distancia 4 con cuatro bits de información.

Código 3 de distancia mínima		*Código 4 de distancia mínima*	
Bits de información	***Bits de paridad***	***Bits de información***	***Bits de paridad***
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

posición de bit i, se puede construir un código de Hamming hasta con $2^i - 1$ posiciones de bit.

decodificador de corrección de errores

La prueba sugiere también cómo podemos diseñar un *decodificador de corrección de errores* para aplicarlo a una palabra de código de Hamming que se recibe. Primero, verificamos todos los grupos de paridad; si todos tienen paridad par, entonces se asume que la palabra recibida es correcta. Si uno o más grupos tienen paridad impar, entonces se supone que ha ocurrido un error simple. El patrón de los grupos que tienen paridad impar (llamado el *síndrome*) debe coincidir con una de las columnas de la matriz de verificación de paridad; se supone que la posición correspondiente del bit contiene el valor equivocado y está complementado. Por ejemplo, usando el código definido por la figura 2-13(b), supongamos que recibimos la palabra 0101011. Los grupos B y C tienen paridad impar, correspondiente a la posición 6 de la matriz de verificación de paridad (el síndrome es 110, o 6). Al complementar el bit en la posición 6 de la palabra recibida, determinamos que la palabra correcta es 0001011.

síndrome

Un código de Hamming de distancia 3 se puede modificar fácilmente para incrementar su distancia mínima a 4. Simplemente agregamos un bit de verificación más, el cual se selecciona de modo tal que la paridad de todos los bits, incluyendo el nuevo, sea par. Como sucede en el código de paridad par de 1 bit, este bit asegura la detección de todos los errores que afectan un número impar de bits. En particular, cualquier error de 3 bits puede detectarse. Ya demostramos que los otros bits de paridad pueden detectar los errores de 1 y 2 bits, así que la distancia mínima del código modificado debe ser 4.

Los códigos de Hamming de distancia 3 y distancia 4 se utilizan comúnmente para detectar y corregir errores en los sistemas de memoria de las computadoras, especialmente en grandes computadoras "mainframes" donde los circuitos de memoria tienen que ver con la mayoría de las fallas del sistema. Estos códigos son especialmente atractivos para palabras de memoria sumamente amplias, puesto que el número requerido de bits de paridad aumenta lentamente con la amplitud de la palabra de memoria, tal como se muestra en la tabla 2-15.

Tabla 2-15 Tamaños de palabra de códigos de Hamming de distancia 3 y de distancia 4.

	Códigos 3 de distancia mínima		***Códigos 4 de distancia mínima***	
Bits de información	***Bits de paridad***	***Bits totales***	***Bits de paridad***	***Bits totales***
1	2	3	3	4
≤ 4	3	≤ 7	4	≤ 8
≤ 11	4	≤ 15	5	≤ 16
≤ 26	5	≤ 31	6	≤ 32
≤ 57	6	≤ 63	7	≤ 64
≤ 120	7	≤ 127	8	≤ 128

2.15.4 Códigos CRC

Más allá de los códigos de Hamming, se han desarrollado otros códigos de detección y corrección de errores. Los más importantes que casualmente incluyen los códigos de Hamming son los *códigos de verificación de redundancia cíclica (CRC, cyclic-redundancy-check)*. Se ha desarrollado una gran cantidad de teoría para estos códigos, que abarcan tanto sus propiedades de detección y corrección de errores como el diseño de sus codificadores y decodificadores de bajo costo (véanse Referencias).

código de verificación de redundancia cíclica (CRC)

Dos aplicaciones importantes de los códigos CRC se encuentran en las unidades controladoras de disco y en las redes de datos. En una unidad de disco, cada bloque de datos (por lo regular de 512 bytes) está protegido mediante un código CRC, de modo que los errores que están en el interior de un bloque pueden detectarse y, en algunas unidades, corregirse. En una red de datos, cada paquete de datos termina con bits de verificación en un código CRC. Los códigos CRC para ambas aplicaciones fueron seleccionados debido a sus propiedades de detección de errores ráfaga. Además de los errores de bit simple,

pueden detectar errores de múltiples bits que estén agrupados en forma contigua, en el interior del bloque (o paquete) del disco. Los errores de múltiples bits tienen una mayor probabilidad de incidencia que los errores de bits que tienen una distribución aleatoria, debido a las probables causas físicas que producen los errores en ambas aplicaciones (defectos de superficie en las unidades de disco y ráfagas de ruido en los enlaces de comunicación).

2.15.5 Códigos bidimensionales

código bidimensional

Otra manera de obtener un código que maximice la distancia mínima es construir un *código bidimensional*, como se ilustra en la figura 2-14(a). Los bits de información se acomodan conceptualmente en un arreglo bidimensional y se proporcionan bits de paridad para verificar tanto los renglones como las columnas. Un código $C_{\text{renglón}}$ con distancia mínima $d_{\text{renglón}}$ se utiliza para los renglones y un posiblemente diferente código C_{columna} con distancia mínima d_{columna} se utiliza para las columnas. Es decir, los bits de paridad de renglón se seleccionan de manera que cada renglón es una palabra de código en $C_{\text{renglón}}$ y los bits de paridad de columna se eligen de manera que es una palabra de código en C_{columna}. (Los bits de paridad de "esquina" pueden seleccionarse de acuerdo con cualquier código.) La distancia mínima del código bidimensional es el producto de $d_{\text{renglón}}$ y d_{columna}; de hecho, los códigos bidimensionales se denominan en ocasiones *códigos de producto*.

código de producto

Figura 2-14
Códigos bidimensionales: a) estructura general; b) uso de la paridad par tanto para los códigos de renglón como de columna para obtener distancia mínima 4; c) patrón típico de un error indetectable.

Como se ilustra en la figura 2-14(b), el código bidimensional más simple utiliza códigos de paridad par de 1 bit para los renglones y columnas, y tiene una distancia mínima de 2 • 2, o 4. Se puede demostrar fácilmente que la distancia mínima es 4 si usted comprende que cualquier patrón de uno, dos o tres bits en error causa paridad incorrecta de un renglón, una columna o ambos. A fin de obtener un error no detectable, al menos cuatro bits deben ser cambiados en un patrón rectangular como en (c).

Los procedimientos de detección y corrección de errores para este código son directos. Supongamos que estamos leyendo información un renglón a la vez. A medida que leemos cada renglón, verificamos su código de renglón. Si se detecta un error en un renglón, no podemos decir cuál bit es erróneo de la verificación de renglón únicamente. Sin embargo, suponiendo que solamente un renglón se encuentre mal, podemos reconstruirlo formando el OR Exclusivo bit por bit de las columnas, omitiendo el renglón malo, pero incluyendo el renglón de verificación de columna.

Para obtener una distancia mínima aún más grande, puede emplearse un código de Hamming de distancia 3 o 4 para el código del renglón, de la columna o de ambos. También es posible construir un código en tres o más dimensiones, con distancia mínima igual al producto de las distancias mínimas en cada dimensión.

Una aplicación importante de los códigos bidimensionales se encuentra en sistemas de almacenamiento RAID. *RAID* son las siglas en inglés de "arreglo redundante de discos económicos". En este esquema se utilizan $n + 1$ unidades de disco idénticas para almacenar valores de datos de n discos. Por ejemplo, ocho unidades de 8 gigabytes podrían emplearse para almacenar 64 gigabytes de datos no redundantes y un noveno disco de 8 gigabytes se utilizaría para almacenar información de verificación.

RAID

La figura 2-15 muestra el esquema general de un código bidimensional para un sistema RAID; cada unidad de disco es considerada como un renglón en el código. Cada unidad almacena m bloques de datos, donde un bloque contiene por lo regular 512 bytes. Por ejemplo, una unidad de 8 gigabytes almacenaría alrededor de 16 millones de bloques. Como se muestra en la figura, cada bloque incluye sus propios bits de verificación en un código CRC, para detectar errores dentro de ese bloque. Las primeras n unidades almacenan los datos no redundantes. Cada bloque en la unidad $n + 1$ almacena bits de

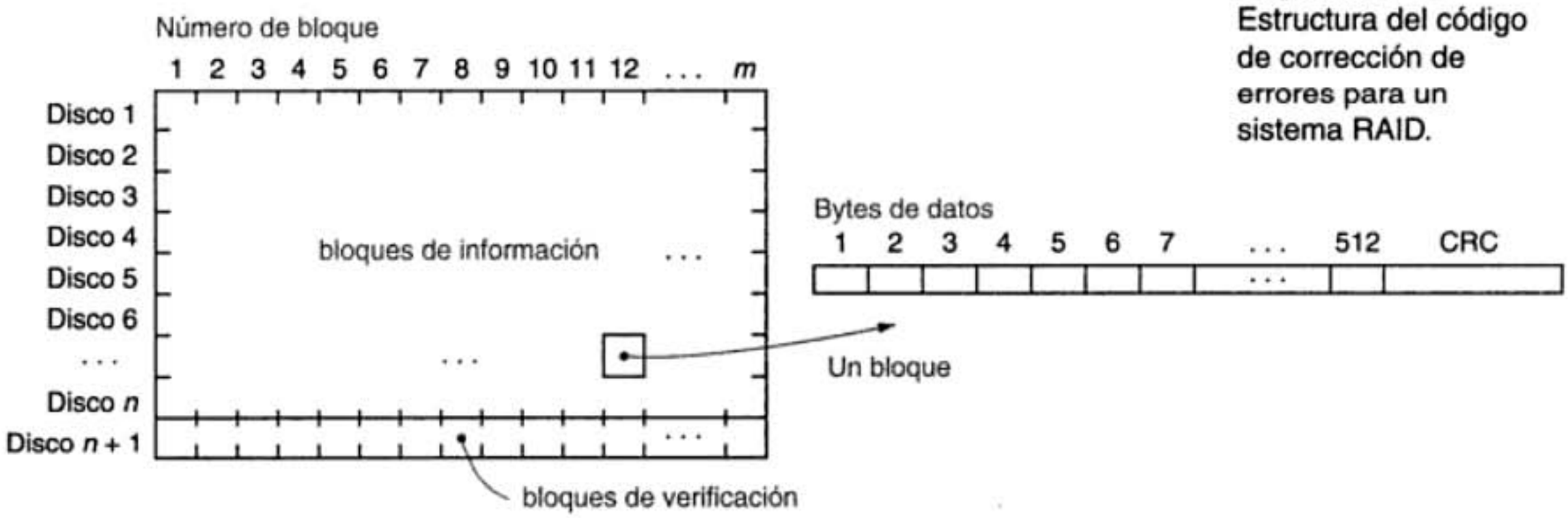


Figura 2-15
Estructura del código de corrección de errores para un sistema RAID.

paridad para los bloques correspondientes en las primeras n unidades. Es decir, cada bit i en la unidad $n + 1$, bloque b, se elige de modo que existe una cantidad par de unos en el bloque b, posición de bit i, en todas las unidades.

Una vez en operación, los errores en los bloques de información son detectados por el código CRC. Cuando se detecta un error en el bloque de una unidad de disco, se puede reconstruir el contenido correcto de ese bloque calculando la paridad de los bloques correspondientes en las demás unidades, incluyendo la unidad $n + 1$. ¡Aunque esto requiere de n operaciones adicionales de lectura en el disco, es preferible a la pérdida de sus datos! Las operaciones de escritura requieren también de accesos extras al disco, para actualizar el bloque de verificación correspondiente cuando se escribe un bloque de información (véase el ejercicio 2.46). Puesto que las escrituras en disco son mucho menos frecuentes que las lecturas en aplicaciones típicas, este gasto extra por lo regular no es un problema.

2.15.6 Códigos de suma de verificación

La operación de verificación de paridad que hemos empleado en las subsecciones anteriores es esencialmente una suma de bits módulo 2 (la suma módulo 2 de un grupo de bits es 0, si la cantidad de unos en el grupo es par, y 1, si es impar). La técnica de suma modular puede aplicarse a otras bases, aparte de la base 2, para formar dígitos de verificación.

Por ejemplo, una computadora almacena información como un conjunto de bytes de 8 bits. Se considera que cada byte puede tener un valor decimal desde 0 hasta 255. Por tanto, podemos utilizar una suma módulo 256 para verificar los bytes. Formamos un byte de verificación simple, llamado una *suma de verificación*, que es la suma módulo 256 de todos los bytes de información. El *código de suma de verificación* resultante puede detectar cualquier error de *byte* simple, puesto que un error de este tipo generará una suma recalculada de bytes cuyo resultado no coincidirá con la suma de verificación.

suma de verificación

código de suma de verificación

Los códigos de suma de verificación también pueden utilizar diferentes módulos de suma. En particular, los códigos de suma de verificación que usan la suma de complemento a unos, módulo 255, son importantes debido a sus propiedades computacionales especiales y de detección de errores; cabe indicar que se utilizan para verificar encabezados de paquetes en el ubicuo Internet Protocol (vea la sección Referencias).

código de suma de verificación de complemento a unos

2.15.7 Códigos *m* de *n*

Los códigos 1 de n y m de n que presentamos en la sección 2.13 tienen una distancia mínima de 2, puesto que cambiando solamente un bit se modifica el número total de unos en una palabra de código, y por tanto produce una palabra de no código.

Estos códigos tienen otra propiedad que se aplica en la detección de errores: pueden localizar errores múltiples unidireccionales. En un *error unidireccional*, todos los bits erróneos cambian en la misma dirección (los ceros cambian a unos, o viceversa). Esta propiedad es muy útil en sistemas donde el mecanismo del error predominante tiende a cambiar todos los bits en la misma dirección.

error unidireccional

2.16 Códigos para el almacenamiento y la transmisión de datos en serie

2.16.1 Datos en paralelo y en serie

La mayor parte de las computadoras y otros sistemas digitales transmiten y almacenan datos en un formato *paralelo*. En la transmisión de datos en paralelo, se proporciona una línea de señal independiente para cada bit de una palabra de datos. En el almacenamiento de datos en paralelo, todos los bits de una palabra de datos se pueden escribir o leer en forma simultánea.

datos en paralelo

El costo de los formatos en paralelo es elevado para algunas aplicaciones. Por ejemplo, la transmisión en paralelo de bytes de datos en la red telefónica requiere ocho líneas telefónicas y el almacenamiento paralelo de bytes de datos en un disco magnético requeriría una unidad de disco con ocho cabezas independientes de lectura/escritura. Los formatos *en serie* permiten la transmisión o el almacenamiento de datos a razón de un bit a la vez, reduciendo así el costo del sistema en muchas aplicaciones.

datos en serie

La figura 2-16 ilustra algunas de las ideas básicas en la transmisión de datos en serie. Una señal de reloj repetitiva, CLOCK en la figura, define la velocidad a la cual se transmiten los datos, un bit por cada ciclo de reloj. De este modo, la *velocidad de los bits*, en bits por segundo (bps), equivale numéricamente a la frecuencia del reloj en ciclos por segundo (hertz, o Hz).

velocidad de transmisión de bits, bps

El recíproco de la velocidad de los bits se conoce como el *tiempo de bit* y numéricamente es igual al periodo del reloj en segundos (s). Esta cantidad de tiempo está reservada en la línea de datos en serie (marcada como SERDATA en la figura) para cada bit que se transmite. El tiempo que ocupa cada bit se conoce como *celda de bit*. El formato de la señal real que aparece en la línea durante cada celda de bit depende del *código de línea*. En el código de línea más simple, denominado *No retorno a cero (NRZ, Non-Return-to-Zero)*, se transmite un 1 al colocar un 1 en la línea para toda la celda de bit, y un 0 se transmite como un 0. Los códigos de línea más complejos tienen otras reglas, como lo discutiremos en la subsección siguiente.

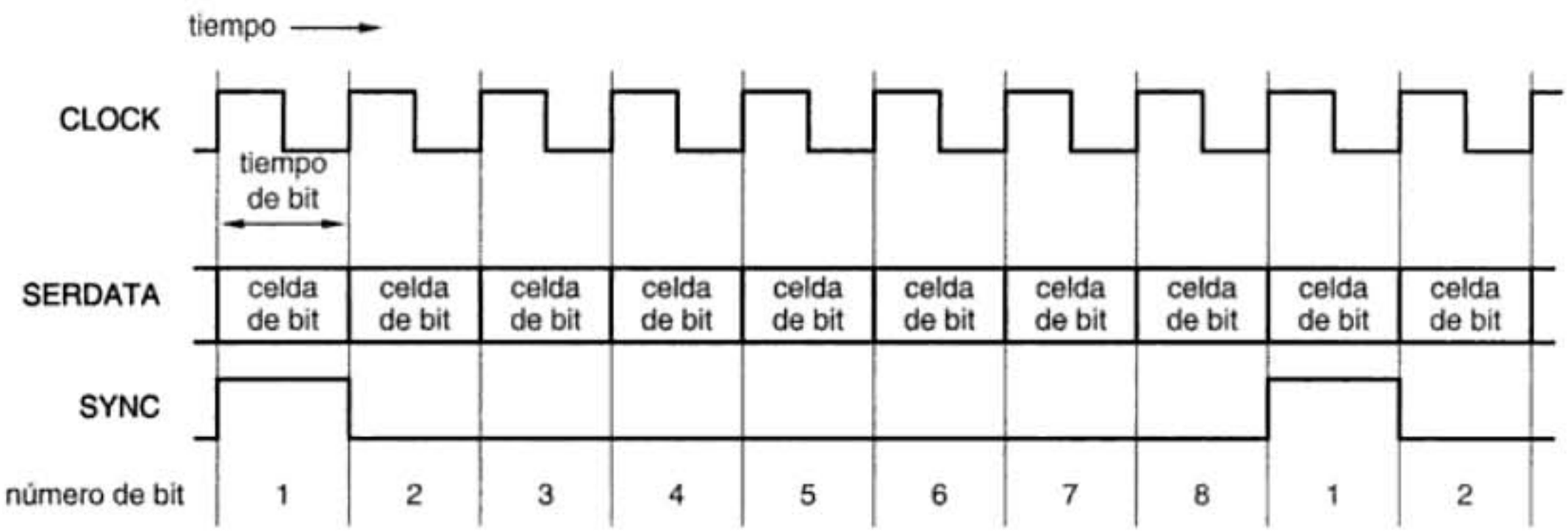
tiempo de bit

celda de bit

código de línea

No retorno a cero (NRZ)

Figura 2-16 Conceptos básicos para la transmisión de datos en serie.



señal de sincronización

Sin importar el código de línea, un sistema de almacenamiento o transmisión de datos en serie necesita algún mecanismo para identificar la importancia de cada bit en el flujo de datos en serie. Por ejemplo, supongamos que se transmiten bytes de 8 bits en serie. ¿Cómo podemos decir cuál es el primer bit de cada byte? Una *señal de sincronización*, identificada como SYNC en la figura 2-16, proporciona la información necesaria; la cual es 1 para el primer bit de cada byte.

Evidentemente, necesitamos un mínimo de tres señales para recuperar un flujo de datos en serie: un reloj para definir las celdas de bits, una señal de sincronización para definir las fronteras de la palabra y los datos en serie. En algunas aplicaciones, como en el caso de la interconexión de módulos (o tarjetas) en una computadora o sistema de telecomunicación, se utiliza un conductor independiente para cada una de estas señales, ya que reducir el número de alambres por conexión de n a tres representa un ahorro considerable. La sección 8.5.4 muestra un ejemplo de un sistema de datos en serie con tres alambres (conductores).

En muchas aplicaciones, el costo de tener tres señales independientes es bastante elevado (por ejemplo, tres líneas telefónicas, tres cabezas de lectura/escritura). Es común que estos sistemas combinen las tres señales en un solo flujo de datos en serie y por tanto, utilizan circuitos analógicos y digitales complejos para recuperar la información de reloj y sincronización del flujo de datos.

*2.16.2 Códigos de línea en serie

Los códigos de línea que se utilizan con mayor frecuencia para la transmisión de datos en serie se ilustran en la figura 2-17. En el código NRZ, cada valor de bit se envía en la línea para toda la celda de bit. Éste constituye el esquema de codificación más simple y confiable para la transmisión a poca distancia. Sin embargo, la señal de reloj debe enviarse con los datos para definir las celdas de bit. De otro modo, el receptor no podría determinar cuántos ceros o unos están representados por un nivel continuo 0 o 1. Por ejemplo, sin un reloj para definir las celdas de bit, la forma de onda NRZ en la figura 2-17 podría interpretarse erróneamente como 01010.

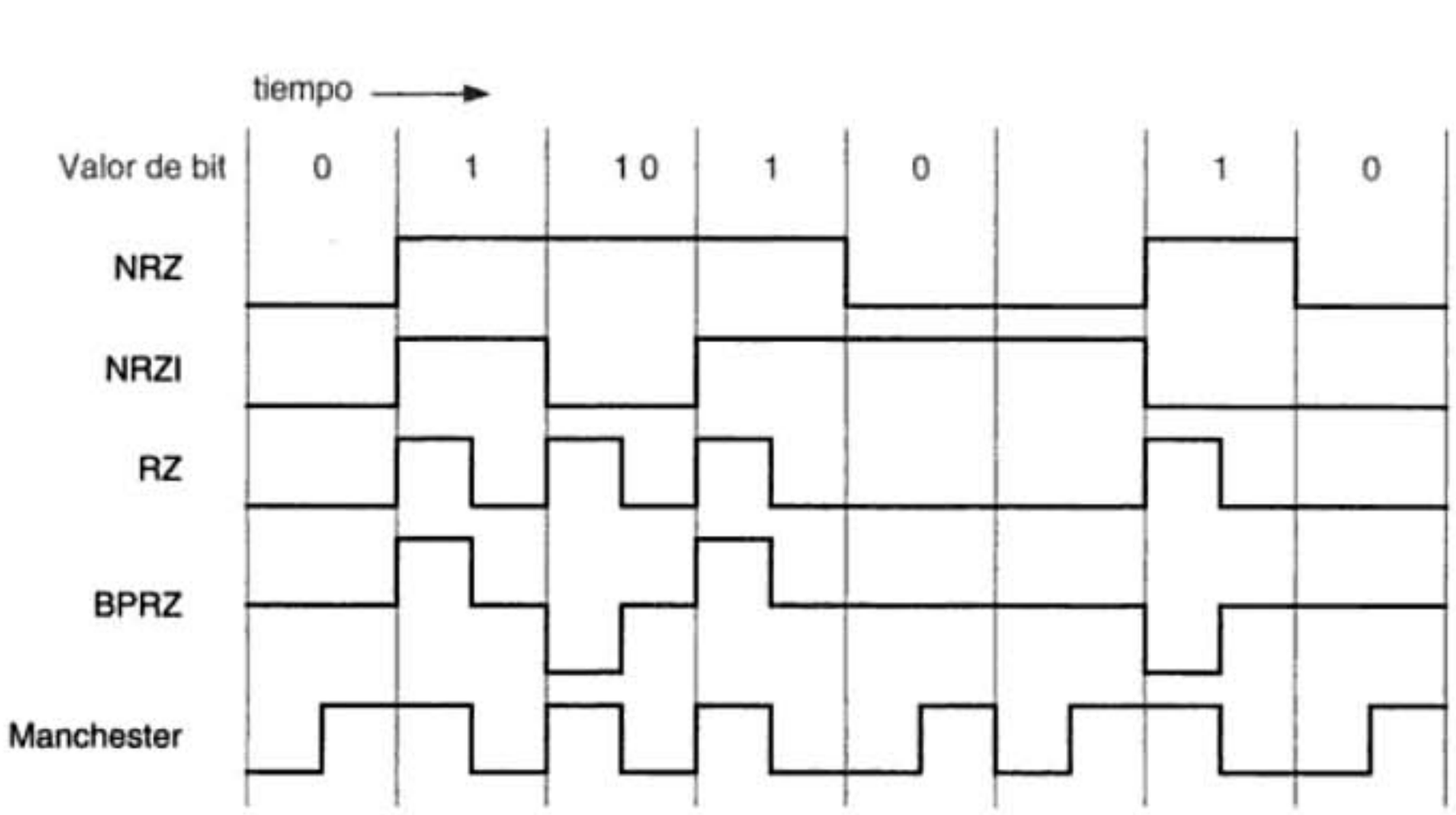


Figura 2-17
Códigos de línea usados comúnmente para datos en serie.

circuito de sincronización de fase digital (DPLL)

Un *circuito de sincronización de fase digital (DPLL, digital phase-locked loop)* es un circuito analógico/digital que se puede utilizar para recuperar una señal de reloj de un flujo de datos en serie. El DPLL funciona solamente si el flujo de datos en serie contiene suficientes transiciones de 0 a 1 y de 1 a 0 que "indican" al DPLL en qué momento se originan las transiciones originales del reloj. Con los datos codificados en NRZ, el DPLL trabaja solamente si los datos no contienen algún flujo continuo y extenso de unos o ceros.

dispositivos sensibles a la transición

Algunos dispositivos de almacenamiento y transmisión en serie son *sensibles a la transición*; no pueden transmitir o almacenar niveles absolutos 0 o 1, solamente transiciones entre dos niveles discretos. Por ejemplo, una cinta o disco magnéticos almacenarán información mediante el cambio de polaridad en la magnetización del elemento, en regiones que corresponden a los bits almacenados. Cuando se recupera la información, no es factible determinar la polaridad de magnetización absoluta de una región, solamente se detecta que la polaridad cambia entre una región y la siguiente.

Los datos que se almacenan en el formato NRZ en dispositivos sensibles a la transición no se recuperan sin cierta ambigüedad; los datos en la figura 2-17 pueden interpretarse como 01110010 o 10001101. El código de *No retorno a cero invertido en unos (NRZI, Non-Return-to-Zero Invert-on-1s)* supera esta limitación al enviar un 1 como el opuesto al nivel que se envió durante la celda de bit anterior, y un 0 como el mismo nivel. Un DPLL puede recuperar la señal de reloj de los datos codificados en NRZI siempre y cuando los datos no contengan algún flujo extenso y continuo de ceros.

No retorno a cero invertido en unos (NRZI)

Retorno a cero (RZ)

El código de Retorno a Cero (RZ) es semejante al NRZ excepto que, para un bit 1, el nivel 1 se transmite solamente por una fracción del tiempo de bit, generalmente 1/2. Con este código, los patrones de datos que contengan muchos unos generan muchas transiciones para que un DPLL pueda utilizarlas a fin de recuperar el reloj. No obstante, como sucede en otros códigos de línea, una cadena de ceros no tiene transiciones, y una larga cadena de ceros hace imposible la recuperación del reloj.

equilibrio de CD

Otro requerimiento de algunos dispositivos de transmisión, tales como enlaces de fibra óptica de alta velocidad, es que el flujo de datos en serie debe estar *equilibrado en CD*. En otras palabras, la cantidad de unos debe ser igual a la de ceros. Si en el flujo de datos existe alguna componente de CD de larga duración (debido a una mayor cantidad de unos que de ceros, o viceversa), aparecerá una polarización en el receptor que reducirá su capacidad para distinguir de manera confiable los unos y los ceros.

En general, los datos NRZ, NRZI o RZ no ofrecen ninguna garantía de equilibrio de CD; nada puede evitar que un flujo de datos tenga una larga cadena de palabras con más unos que ceros o viceversa. Sin embargo, el equilibrio de CD puede conseguirse mediante el uso de unos cuantos bits extra para codificar los datos del usuario en un *código equilibrado*. En este código cada palabra de código tiene una cantidad idéntica de unos y ceros, luego se procede a enviar las palabras de código en formato NRZ.

código equilibrado

Por ejemplo, en la sección 2.13 presentamos el código 8B10B, que codifica 8 bits de datos del usuario en 10 bits en un código en su mayor parte de 5 de 10. Cabe recordar que solamente existen 252 palabras de código 5 de 10, pero que hay otras $\binom{10}{4} = 210$ palabras de código 4 de 10 y un número igual de palabras de código 6 de 10. Naturalmente, estas palabras de código no están suficientemente equilibradas en CD. El código 8B10B resuelve este problema al asociar a cada valor de 8 bits que será codificado un *par* de palabras de código no equilibradas, una de 4 de 10 ("ligera") y otra de 6 de 10 ("pesada"). El codificador también sigue la pista de la *disparidad de tramos*, un bit de información simple

disparidad de tramos

que indica si la última palabra de código no equilibrada que se transmitió era pesada o ligera. Cuando llega el momento de transmitir otra palabra de código no equilibrada, el codificador selecciona aquélla del par con el peso opuesto. Este sencillo truco hace disponibles 252 + 210 = 462 palabras de código para que el 8B10B codifique 8 bits de datos del usuario. Algunas de las palabras de código "extra" se utilizan para codificar de manera conveniente condiciones de ausencia de datos en la línea en serie, tales como IDLE, SYNC y ERROR. No se utilizan todas las palabras de código no equilibradas. Lo mismo sucede con algunas palabras de código balanceadas (como 0000011111), esto favorece a los pares no equilibrados que contienen más transiciones.

Retorno a Cero Bipolar (BPRZ)

Todos los códigos anteriores transmiten o almacenan solamente dos niveles de señal. El código de *Retorno a Cero Bipolar (BPRZ, Bipolar Return–to–Zero)* transmite con tres niveles de señal: +1, 0 y –1. El código es como el RZ excepto que los unos se transmiten alternativamente como +1 y –1; por esta razón, el código también se conoce como de *Inversión de Marca Alternada (AMI, Alternate Mark Inversion).*

Inversión de Marca Alternada (AMI)

La gran ventaja del BPRZ sobre el RZ es que está balanceado en CD. Esto permite enviar flujos de BPRZ a través de los dispositivos de transmisión que toleran una componente de CD, como sucede en las líneas telefónicas acopladas por transformador. De hecho, el código BPRZ se ha utilizado durante décadas en los enlaces telefónicos digitales T1, donde las señales analógicas de voz se transportan como flujos de 8000 muestras digitales de 8 bits por segundo que se transmiten en formato BPRZ a través de canales en serie de 64 Kbps.

Como sucede con el RZ, es posible recuperar una señal de reloj de flujo BPRZ siempre y cuando no existan demasiados ceros en un renglón. Aunque la TPC (Compañía Telefónica, por sus siglas en inglés) no tiene control sobre lo que usted dice (al menos, no en este momento), aún tiene una manera simple de limitar tramos de ceros. Si uno de los bytes de 8 bits (que se generan al muestrear su patrón analógico de voz) está integrado por ceros, simplemente cambian el segundo bit menos significativo a 1! Esto se conoce como *supresión del código cero* y apostaría que usted nunca lo había notado. Esto se debe a que en la mayor parte de las aplicaciones de datos de enlaces T1, usted obtiene solamente 56 Kbps de datos útiles de un canal de 64 Kbps; el LSB de cada byte siempre se fija a 1 para impedir que la supresión de código cero cambie los otros bits.

supresión del código cero

Manchester
difase

El último código en la figura 2-17 se denomina código *Manchester* o *difase*. La mayor ventaja de este código es que, sin tener en cuenta el patrón de datos que se transmiten, proporciona por lo menos una transición por celda de bit, esto facilita la recuperación de la señal del reloj. Como se muestra en la figura, un 0 se codifica como

una transición de 0 a 1 en la parte media de la celda de bit, y un 1 se codifica como una transición de 1 a 0. Pero la mayor ventaja del código Manchester también es su mayor debilidad. Puesto que tiene más transiciones por celda de bit que otros códigos, necesita un mayor ancho de banda en el dispositivo para transmitir bits a una velocidad determinada. No obstante, el ancho de banda no es un problema en los cables coaxiales que se utilizaban en las redes originales de área local (Ethernet) para transportar datos en serie codificados en código Manchester a la velocidad de 10 Mbps (megabits por segundo).

Referencias

La presentación en las primeras nueve secciones de este capítulo está basada en el capítulo 4 de la obra *Microcomputer Architecture and Programming*, de John F. Wakerly (Wiley, 1981). El lector encontrará discusiones precisas, minuciosas y entretenidas de estos temas también en *Seminumerical Algorithms*, tercera edición, de Donald E. Knuth (Addison-Wesley, 1997). A los estudiantes inclinados hacia las matemáticas les parecerán excelentes los análisis de las propiedades de la aritmética y los sistemas numéricos de Knuth, y todos los lectores disfrutarán las ideas y la historia que presenta el texto.

Las descripciones de los circuitos de lógica digital para operaciones aritméticas, así como también una introducción a las propiedades de diversos sistemas numéricos, aparecen en la obra *Computer Arithmetic,* de Kai Hwang (Wiley, 1979). El libro *Decimal Computation* de Hermann Schmid (Wiley, 1974) contiene una profunda descripción de las técnicas para la aritmética BCD.

La obra *Microcomputer Architecture and Programming: The 68000 Family*, de John F. Wakerly (Wiley, 1989) presenta una introducción a los algoritmos para la multiplicación y división binarias y a la aritmética de punto flotante. Una discusión más profunda de las técnicas aritméticas y sistemas numéricos de punto flotante puede hallarse en *Introduction to Arithmetic for Digital Systems Designers* de Shlomo Waser y Michael J. Flynn (Holt, Rinehart and Winston, 1982).

campos finitos

Los códigos CRC se basan en la teoría de los *campos finitos* que desarrolló el matemático francés Évariste Galois (1811-1832), poco antes de morir en un duelo a manos de un oponente político. La obra clásica que aborda los códigos de detección y corrección de errores es *Error-Correcting Codes* de W. W. Peterson y E. J. Weldon, Jr. (MIT Press, 1972, segunda edición); sin embargo, solamente recomiendo este libro a los lectores que dominan las matemáticas. Una introducción más accesible a la codificación se puede encontrar en la obre *Error Control Coding: Fundamentals and Applications* de S. Lin y D. J. Costello, Jr. (Prentice Hall, 1983). Otra introducción orientada a la comunicación para la teoría de la codificación se encuentra en *Error-Control Techniques for*

Digital Communication de A. M. Michelson y A. H. Levesque (Wiley-Interscience, 1985). Las aplicaciones de hardware de códigos en sistemas de cómputo se analizan en la obra *Error-Detecting Codes, Self-Checking Circuits and Applications de* John F. Wakerly (Elsevier/North-Holland, 1978).

Como se muestra en la referencia anterior del autor, los códigos de suma de verificación de complemento a uno tienen la capacidad de detectar largas ráfagas de errores unidireccionales; esto es muy útil en los canales de comunicación donde todos los errores tienden a estar en la misma dirección. Las propiedades especiales de cómputo de estos códigos permiten su aplicación en el cálculo de sumas de verificación mediante programas de software, lo anterior tiene aplicaciones importantes en el Protocolo de Internet; véase RFC-1071 y RFC-1141. Las solicitudes para comentarios (RFC, Requests for Comments) se archivan en muchos lugares de la red; solamente busque "RFC".

La obra *Introduction to Communications Engineering* de R. M. Gagliardi (Wiley-Interscience, 1988, segunda edición) presenta una introducción a las técnicas de codificación para la transmisión de datos en serie, e incluye el análisis matemático del rendimiento y los requerimientos de ancho de banda de diversos códigos. La obra *Computer Storage Systems and Technology* de Richard Matick (Wiley-Interscience, 1977) presenta una atractiva introducción a los códigos en serie que se utilizan en cintas y discos magnéticos.

La estructura del código 8B10B y la lógica que lo soporta se explica de manera agradable en la patente original de IBM de Peter Franaszek y Albert Widmer, U.S. patent number 4,486,739 (1984). Ésta y casi todas las patentes de Estados Unidos expedidas después de 1971 se encuentran en la Web, en la dirección `www.patents.ibm.com`.

Problemas propuestos

2.1 Realice las siguientes conversiones de sistemas numéricos:

(a) $1101011_2 = ?_{16}$ (b) $174003_8 = ?_2$

(c) $10110111_2 = ?_{16}$ (d) $67.24_8 = ?_2$

(e) $10100.1101_2 = ?_{16}$ (f) $F3A5_{16} = ?_2$

(g) $11011001_2 = ?_8$ (h) $AB3D_{16} = ?_2$

(i) $101111.0111_2 = ?_8$ (j) $15C.38_{16} = ?_2$

2.2 Convierta los siguientes números octales en binarios y hexadecimales:

(a) $1023_8 = ?_2 = ?_{16}$ (b) $761302_8 = ?_2 = ?_{16}$

(c) $163417_8 = ?_2 = ?_{16}$ (d) $552273_8 = ?_2 = ?_{16}$

(e) $5436.15_8 = ?_2 = ?_{16}$ (f) $13705.207_8 = ?_2 = ?_{16}$

2.3 Convierta los siguientes números hexadecimales en binarios y octales:

(a) $1023_{16} = ?_2 = ?_8$ (b) $7E6A_{16} = ?_2 = ?_8$

(c) $ABCD_{16} = ?_2 = ?_8$ (d) $C350_{16} = ?_2 = ?_8$

(e) $9E36.7A_{16} = ?_2 = ?_8$ (f) $DEAD.BEEF_{16} = ?_2 = ?_8$

2.4 ¿Cuáles son los valores octales de los cuatro bytes de 8 bits en el número de 32 bits que tiene la representación octal 12345670123_8?

2.5 Convierta los siguientes números en decimales:

(a) $1101011_2 = ?_{10}$ (b) $174003_8 = ?_{10}$

(c) $10110111_2 = ?_{10}$ (d) $67.24_8 = ?_{10}$

(e) $10100.1101_2 = ?_{10}$ (f) $F3A5_{16} = ?_{10}$

(g) $12010_3 = ?_{10}$ (h) $AB3D_{16} = ?_{10}$

(i) $7156_8 = ?_{10}$ (j) $15C.38_{16} = ?_{10}$

2.6 Realice las siguientes conversiones de sistemas numéricos:

(a) $125_{10} = ?_2$ (b) $3489_{10} = ?_8$

(c) $209_{10} = ?_2$ (d) $9714_{10} = ?_8$

(e) $132_{10} = ?_2$ (f) $23851_{10} = ?_{16}$

(g) $727_{10} = ?_5$ (h) $57190_{10} = ?_{16}$

(i) $1435_{10} = ?_8$ (j) $65113_{10} = ?_{16}$

2.7 Sume los siguientes pares de números binarios, mostrando todos los acarreos:

(a) 110101 + 11001 (b) 101110 + 100101 (c) 11011101 + 1100011 (d) 1110010 + 1101101

2.8 Repita el problema 2.7 usando la resta en vez de la suma, y mostrando los préstamos en lugar de los acarreos.

2.9 Sume los siguientes pares de números octales:

(a) 1372 + 4631 (b) 47135 + 5125 (c) 175214 + 152405 (d) 110321 + 56573

2.10 Sume los pares siguientes de números hexadecimales:

(a) 1372 + 4631 (b) 4F1A5 + B8D5 (c) F35B + 27E6 (d) 1B90F + C44E

2.11 Escriba las representaciones de complemento a unos y complemento a dos, de magnitud con signo de 8 bits, para cada uno de estos números decimales: +18, +115, +79, –49, –3, –100.

2.12 Indique si ocurre o no desbordamiento cuando se suman los siguientes números de complemento a dos de 8 bits:

(a) 11010100 + 10101011 (b) 10111001 + 11010110 (c) 01011101 + 00100001 (d) 00100110 + 01011010

2.13 ¿Cuántos errores pueden detectarse por un código con distancia mínima d?

2.14 ¿Cuál es el número mínimo de bits de paridad que se requieren para obtener un código bidimensional, de distancia 4 con n bits de información?

Ejercicios

2.15 Aquí tenemos un problema para abrir su apetito: ¿Cuál es el equivalente hexadecimal de 61453_{10}?

2.16 Cada una de las siguientes operaciones aritméticas es correcta en por lo menos un sistema numérico. Determine las posibles bases de los números en cada operación.

(a) 1234 + 5432 = 6666 (b) 41 / 3 = 13

(c) 33/3 = 11 (d) 23+44+14+32 = 223

(e) 302/20 = 12.1 (f) 14 = 5

2.17 La primera expedición a Marte encontró sólo las ruinas de una civilización. De los artefactos e imágenes, los exploradores dedujeron que las criaturas que produjeron esta civilización eran seres de cuatro piernas con un tentáculo que se ramificaba al final en un número de "dedos" prensiles. Después de mucho estudio, los exploradores fueron capaces de traducir las matemáticas marcianas. Encontraron que la siguiente ecuación:

$$5x^2 - 50x + 125 = 0$$

con las soluciones indicadas $x = 5$ y $x = 8$. El valor $x = 5$ parecía bastante legítimo, pero $x = 8$ requería alguna explicación. Entonces los exploradores reflexionaron en la manera en que se desarrolló el sistema numérico de la Tierra, y hallaron evidencia de que el sistema marciano tenía una historia semejante. ¿Cuántos dedos diría usted que tenían los marcianos? (De *The Bent of Tau Beta Pi*, febrero, 1956.)

2.18 Supongamos que un número B de $4n$ bits está representado por un número hexadecimal H de n dígitos. Demuestre que el complemento a dos de B está representado por el complemento a 16 de H. Establezca y demuestre una proposición similar para la representación octal.

2.19 Repita el ejercicio 2.18 usando el complemento a uno de B y el complemento a 15 de H.

2.20 Dado un entero x en el intervalo $-2^{n-1} \leq x \leq 2^{n-1} - 1$, definimos $[x]$ como la representación de complemento a dos de x, expresada como un número positivo: $[x] = x$ si $x \geq 0$ y $[x] = 2^n - |x|$ si $x < 0$, donde $|x|$ es el valor absoluto de x. Demuestre que las reglas de la suma del complemento a dos dadas en la sección 2.6 son correctas, probando que la siguiente ecuación es siempre verdadera:

$$[x + y] = ([x] + [y]) \text{ módulo } 2^n$$

(*Sugerencias:* considere cuatro casos basados en los signos de x y de y. Sin pérdida de generalidad, se puede suponer que $|x| \geq |y|$.)

2.21 Repita el ejercicio 2.20 utilizando reglas y expresiones apropiadas para la suma del complemento a unos.

2.22 Establezca una regla de desbordamiento para la suma de dos números en complemento a dos en términos de operaciones de conteo en la representación modular de la figura 2-3.

2.23 Demuestre que un número en complemento a dos puede ser convertido a una representación con más bits mediante la *extensión de signo*. Es decir, dado un número X en complemento a dos de n bits, muestre que la representación en complemento a dos de m bits de X, donde $m > n$, puede ser obtenida agregando $m - n$ copias del bit de signo de X a la izquierda de la representación de n bits de X.

2.24 Demuestre que un número de complemento a dos puede convertirse a una representación con menos bits eliminando los bits de mayor orden. Es decir, dado un número X en complemento a dos de n bits, demuestre que el número Y en complemento a dos de m bits obtenido al descartar los d bits más a la izquierda de X representa el mismo número que X, si y sólo si los todos bits descartados igualan el bit de signo de Y.

2.25 ¿Por qué es inconsistente la puntuación de "complemento a dos" y "complemento a uno"? (Véanse las primeras dos citas en la sección de Referencias.)

2.26 Un sumador binario de n bits puede ser utilizado para efectuar una operación de resta sin signo de n bits $X - Y$, realizando la operación $X + y + 1$, donde X y Y son números sin signo de n bits y la Y representa el complemento bit a bit de Y. Demuestre este hecho como sigue. Primero, pruebe que $(X - Y) = (X + Y + 1) - 2^n$. Segundo, demuestre que el acarreo de salida del sumador de n bits es lo opuesto al préstamo de la resta de n bits. Es decir, muestre que la operación $X - Y$ produce un préstamo de salida de la posición MSB si y sólo si la operación $X + Y + 1$ *no* produce un acarreo de salida de la posición MSB.

2.27 En la mayoría de los casos, el producto de dos números de complemento a dos de n bits requiere menos de $2n$ bits para representarlo. De hecho, existe solamente un caso en el cual $2n$ bits son necesarios. Encuentre cuál es este caso.

2.28 Demuestre que un número de complemento a dos puede ser multiplicado por 2 al desplazarlo una posición hacia la izquierda, con un acarreo de 0 en la posición del bit menos significativo y despreciando cualquier acarreo fuera de la posición del bit más significativo, suponiendo que no hay desbordamiento. Establezca la regla para detectar el desbordamiento.

2.29 Establezca y pruebe la exactitud de una técnica semejante a la que se describe en el ejercicio 2.28, para multiplicar un número de complemento a uno por 2.

2.30 Demuestre cómo restar números BCD, estableciendo las reglas para generar préstamos y aplicando un factor de corrección. Demuestre cómo se aplican sus reglas a cada una de las restas siguientes: 9 – 3, 5 – 7, 4 – 9, 1 – 8.

2.31 ¿Cuántas codificaciones diferentes de estado binario de 3 bits son posibles para el controlador de semáforos de la tabla 2-12?

2.32 Enumere todas las fronteras "malas" en el disco de codificación mecánica de la figura 2-5, donde una posición incorrecta puede ser detectada.

2.33 Como una función de n, ¿cuántas fronteras "malas" existen en un disco de codificación mecánica que utiliza un código binario de n bits?

2.34 Los transpondedores (emisores-receptores automáticos de identificación) de altitud a bordo en las aeronaves comerciales y privadas utilizan código Gray para codificar las lecturas de altitud que se transmiten a los controladores de tráfico aéreo. ¿Por qué?

2.35 Un foco incandescente se tensiona cada vez que se enciende, de modo que en algunas aplicaciones el tiempo de vida del foco está limitado por el número de ciclos de encendido/apagado en lugar del tiempo total que ilumina. Utilice sus conocimientos de códigos para sugerir una manera de duplicar el tiempo de vida de focos de 3 intensidades en tales aplicaciones.

2.36 Como una función de n, ¿cuántos subcubos distintos se tienen de un cubo n?

2.37 Encuentre una manera de dibujar un cubo 3 sobre una hoja de papel (u otros objetos bidimensionales) de modo que ninguna de las líneas se crucen, o demuestre que eso sea imposible.

2.38 Repita el ejercicio 2.37 para un cubo 4.

2.39 Escriba una fórmula que nos dé el número de subcubos *m* de un cubo *n* para un valor específico de *m*. (Su respuesta debería ser una función de *n* y *m*.)

2.40 Defina grupos de paridad para un código Hamming de distancia 3 con 11 bits de información.

2.41 Escriba las palabras de código de un código de Hamming con un bit de información.

2.42 Exponga el patrón para un error de 3 bits que no es detectado si los bits de paridad de "esquina" no se incluyen en los códigos bidimensionales de la figura 2-14.

2.43 El *índice de un código* es la razón del número de bits de información con respecto al número total de bits en una palabra de código. Los índices altos, cercanos a 1, son deseables para una eficaz transmisión de la información. Construya una gráfica que compare los índices de códigos de paridad de distancia 2 y códigos Hamming de distancia 3 y 4 hasta de 100 bits de información.

2.44 ¿Qué tipo de código de distancia 4 tiene un índice mayor: un código bidimensional o un código de Hamming? Apoye su respuesta con una tabla del estilo de la tabla 2-15, incluyendo el índice así como también el número de bits de paridad y de información de cada código hasta llegar a 100 bits de información.

2.45 Muestre cómo construir un código de distancia 6 con cuatro bits de información. Escriba una lista de sus palabras de código.

2.46 Describa las operaciones que deben realizarse en un sistema RAID para escribir nuevos datos en el bloque de información *b* en la unidad *d* de manera que los datos puedan ser recuperados en el caso de un error en el bloque *b* en cualquier unidad. Minimice el número de accesos a disco requeridos.

2.47 Del mismo modo que en la figura 2-17, dibuje las formas de onda para el patrón de bits 10101110 cuando se envía en serie utilizando los códigos NRZ, NRZI, RZ, BPRZ y Manchester, suponiendo que los bits sean transmitidos en orden de izquierda a derecha.