621.395
A532

Análisis y diseño de circuitos lógicos digitales

Primera edición

UNIVERSIDAD TECNOLOGICA
BIBLIOTECA
DE PEREIRA

Victor P. Nelson
Auburn University

H. Troy Nagle
North Carolina State University

Bill D. Carroll
University of Texas-Arlington

J. David Irwin
Auburn University

Traducción:

M en C, Oscar Alfredo Palmas Velasco
Facultad de Ciencias, UNAM

Revisor técnico

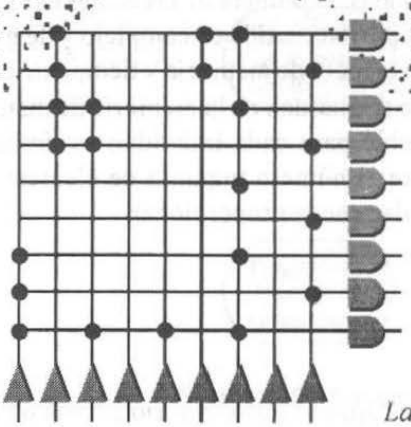
Carlos Henández Pérez
Ingeniero Mecánico Electricista
UNAM

Biblioteca Univ. Tecnologica de Pereira
6 3100 00045511 5

PRENTICE-HALL HISPANOAMERICANA, S.A.

MÉXICO • NUEVA YORK • BOGOTÁ • LONDRES • SYDNEY • PARÍS • MUNICH • TORONTO •
NUEVA DELHI • TOKIO • SINGAPUR • RÍO DE JANEIRO • ZURICH

GRUPO NORIEGA LIDA
$71.655
9 DE SEPTIEMBRE DE 2003



La función principal de las computadoras y otros sistemas digitales es la de procesar información. Por tanto, es necesario contar con métodos y sistemas para representar la información en formas que se puedan manipular y almacenar utilizando equipo electrónico o de otro tipo.

En este capítulo analizaremos los sistemas numéricos y los códigos que se utilizan con frecuencia en las computadoras y los sistemas digitales. Los temas incluyen los sistemas numéricos binario, octal y hexadecimal y su aritmética; las técnicas de conversión de base; los métodos para representar números negativos, como los de magnitud y signo, complemento a dos y complemento a uno; los códigos numéricos para números de punto fijo y flotante; los códigos de caracteres, incluidos el decimal codificado en binario y el ASCII; los códigos Gray y de exceso; y los códigos para detección y corrección de errores. Los capítulos posteriores del libro tratan el análisis y diseño de hardware para procesar información representada en las formas aquí descritas.

Sistemas numéricos y códigos

1.1 Sistemas numéricos

Un sistema numérico consta de un conjunto ordenado de símbolos, llamados *dígitos*, con relaciones definidas para la suma (+), resta (−), multiplicación (×) y división (÷). La *base* (r) del sistema numérico es el número total de dígitos permitidos en dicho sistema. Los sistemas numéricos de uso común en el diseño de sistemas digitales y la programación de computadoras incluyen el *decimal* ($r = 10$), el *binario* ($r = 2$), el *octal* ($r = 8$) y el *hexadecimal* ($r = 16$). Cualquier número en un sistema dado puede tener una parte entera y una parte fraccionaria, que se separan mediante un punto (.). En algunos casos, puede faltar la parte entera o la parte fraccionaria. Ahora, examinaremos las notaciones *posicional* y *polinomial* de un número.

1.1.1 Notación posicional

Suponga que pide a su banco local un préstamo por ciento veintitrés dólares y treinta y cinco centavos. El cheque que le dan indica la cantidad como $123.35. Al escribir este número, se ha utilizado la notación posicional. El cheque puede cobrarse con un billete de cien dólares, dos billetes de diez dólares, tres billetes de un dólar, tres monedas de diez centavos y cinco monedas de un centavo. Por tanto, la posición de cada dígito indica su peso o significado relativo.

En general, un número positivo N se puede escribir en notación posicional como

$$N = (a_{n-1}a_{n-2}\ldots a_1 a_0 \,.\, a_{-1}a_{-2}\ldots a_{-m})_r \tag{1.1}$$

donde

. = punto que separa los dígitos enteros y fraccionarios
r = base del sistema numérico que se está utilizando
n = número de dígitos enteros a la izquierda del punto
m = número de dígitos fraccionarios a la derecha del punto
a_i = dígito entero i cuando $n-1 \geq i \geq 0$
a_i = dígito fraccionario i cuando $-1 \geq i \geq -m$
a_{n-1} = dígito más significativo
a_{-m} = dígito menos significativo

Observe que el intervalo de valores para los dígitos a_i es $r-1 \geq a_i \geq 0$. Con esta notación, la cantidad del préstamo bancario podría escribirse $\$(123.35)_{10}$. Los paréntesis y el subíndice que denota la base pueden eliminarse sin perder información siempre que la base se conozca por el contexto o se especifique de otra forma.

Notación polinomial

Podemos escribir la cantidad del préstamo de $(123.35)_{10}$ dólares en forma polinomial como

$$N = 1 \times 100 + 2 \times 10 + 3 \times 1 + 3 \times 0.1 + 5 \times 0.01$$
$$= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 3 \times 10^{-1} + 5 \times 10^{-2}$$

Observe que cada dígito está en una posición ponderada y que el peso de cada posición es una potencia de la base, 10. En general, cualquier número N con base r se puede escribir como un polinomio de la forma

$$N = \sum_{i=-m}^{n-1} a_i r^i \tag{1.2}$$

donde cada símbolo se define como en la ecuación 1.1. Para el préstamo del banco, $r = 10$, $a_2 = 1$, $a_1 = 2$, $a_0 = 3$, $a_{-1} = 3$, $a_{-2} = 5$ y $a_i = 0$, para $i \geq 3$ y para $i \leq -3$.

1.1.2 Sistemas numéricos de uso común

Los sistemas numéricos decimal, binario, octal y hexadecimal son importantes para el estudio de los sistemas digitales. La tabla 1.1 resume las características fundamentales de cada sistema e ilustra un intervalo limitado de enteros positivos en cada uno. Todos los números de la tabla 1.1 están escritos en notación posicional.

Por lo general, los sistemas digitales se construyen con dispositivos de dos estados (apagado o encendido). Por tanto, el sistema numérico binario es ideal para representar números en sistemas digitales, ya que sólo se necesitan dos dígitos, 0 y 1, llamados *bits*. Un bit puede guardarse en un dispositivo de almacenamiento de dos estados conocido como *latch*. Los números binarios de longitud n se pueden guardar en un dispositivo de n bits de longitud, llamado *registro*, construido con n latches. Un registro de 8 bits que contiene el número binario 10011010 se muestra en la figura 1.1.

1	0	0	1	1	0	1	0

Figura 1.1 Un registro de 8 bits.

1.2 Aritmética

Todos los niños aprenden los rudimentos de la aritmética memorizando las tablas de suma y multiplicación en base 10 que aparecen en las tablas 1.2a y b, respectivamente. La resta se puede realizar utilizando la tabla de la suma al revés. De manera análoga, la división utiliza la multiplicación y la resta mediante prueba y error para obtener el cociente. El fundamento de la aritmética en cualquier base es el conocimiento de tablas de suma y multiplicación en la base dada. Dadas estas tablas, las operaciones aritméticas son similares para todas las bases. En el resto de esta sección presentaremos la aritmética en los sistemas numéricos binario, octal y hexadecimal.

TABLA 1.1 SISTEMAS NUMÉRICOS IMPORTANTES

Nombre	Decimal	Binario	Octal	Hexadecimal
Base	10	2	8	16
Dígitos	0,1,2,3,4, 5,6,7,8,9	0,1	0,1,2,3, 4,5,6,7	0,1,2,3,4,5, 6,7,8,9,*A,B, C,D,E,F*
Primeros	0	0	0	0
diecisiete	1	1	1	1
positivos	2	10	2	2
enteros	3	11	3	3
	4	100	4	4
	5	101	5	5
	6	110	6	6
	7	111	7	7
	8	1000	10	8
	9	1001	11	9
	10	1010	12	A
	11	1011	13	B
	12	1100	14	C
	13	1101	15	D
	14	1110	16	E
	15	1111	17	F
	16	10000	20	10

1.2.1 Aritmética binaria

Suma

Las tablas 1.3a y b muestran las tablas de suma y multiplicación, respectivamente, para el sistema numérico binario. Las tablas son muy pequeñas ya que sólo hay dos dígitos, o *bits*, en el sistema. En consecuencia, la aritmética binaria es muy sencilla. Observe que la suma 1+1 produce un bit de suma de 0 y un bit de acarreo de 1. El acarreo debe sumarse a la siguiente columna de bits para realizar la suma en el patrón normal, de derecha a izquierda. Daremos dos ejemplos de suma binaria a continuación.

EJEMPLO 1.1

Sumar los dos números binarios $(111101)_2$ y $(10111)_2$.

	1	*1*	*1*	*1*	*1*	*1*	*Acarreos*
		1	1	1	0	0	1 Sumando
+			1	0	1	1	1 Sumando
	1	0	1	0	1	0	0 Suma

TABLA 1.2 (a) TABLA DE SUMA DECIMAL; (b) TABLA DE MULTIPLICACIÓN DECIMAL.

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

(a)

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	9	9	18	27	36	45	54	63	72	81

(b)

TABLA 1.3 (a) TABLA DE SUMA BINARIA; (b) TABLA DE MULTIPLICACIÓN BINARIA.

+	0	1
0	0	1
1	1	10

(a)

×	0	1
0	0	0
1	0	1

(b)

En el ejemplo 1.1 hay dos columnas que tienen dos bits 1 y un bit de acarreo 1, lo que debe sumarse. Esta suma de tres unos puede verse más fácilmente como

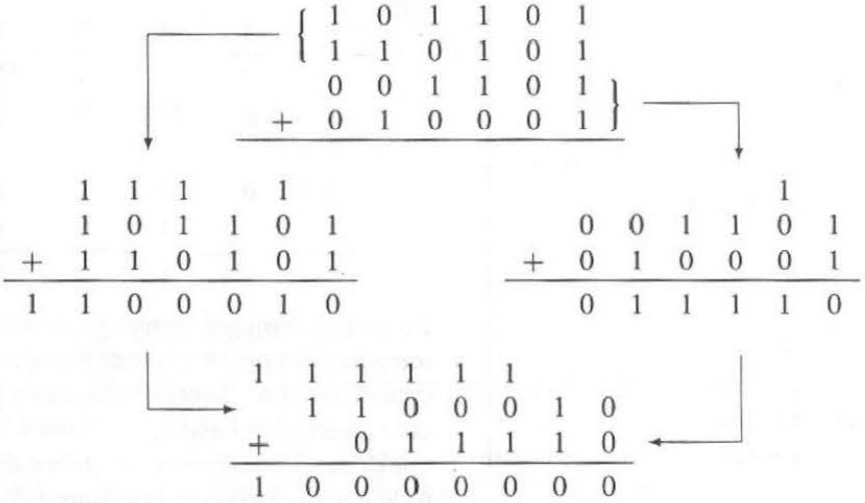
$$\begin{aligned} 1 + 1 + 1 &= (1 + 1) + 1 \\ &= (10)_2 + (01)_2 \\ &= 11 \end{aligned}$$

Así, los bits de suma y de acarreo son ambos 1.

Cuando hay que sumar una gran lista de números binarios, los cálculos se realizan fácilmente si sumamos los números por pares, como se demuestra en el siguiente ejemplo.

EJEMPLO 1.2

Sumar los cuatro números $(101101)_2$, $(110101)_2$, $(001101)_2$ y $(010001)_2$.



Sin embargo, podemos optar por sumar en forma directa para evitar los pasos intermedios necesarios en el método anterior. El método directo se ilustra en el siguiente ejemplo.

EJEMPLO 1.3

Repetir el ejemplo anterior sumando toda una columna a la vez.

	10	*10*	*10*	*10*	*1*	*10*		*Acarreos*
		1	0	1	1	0	1	
		1	1	0	1	0	1	
		0	0	1	1	0	1	
+		0	1	0	0	0	1	
1	0	0	0	0	0	0	0	Suma

Observe que la suma de los dígitos en la primera columna es $1 + 1 + 1 + 1 = (100)_2$. Esto produce un dígito de suma igual a 0 en esa columna y un acarreo de 10 en la siguiente columna a la izquierda.

Resta

La resta se puede visualizar como el inverso de la suma. Las reglas para la resta binaria se derivan directamente de la tabla de suma binaria (tabla 1.3a) y son

$$1 - 0 = 1$$
$$1 - 1 = 0$$

$0 - 0 = 0$

$0 - 1 = 1$ tomando prestado 1, o $10 - 1 = 1$

La última regla muestra que si se resta un bit 1 de un bit 0, hay que tomar prestado un 1 de la siguiente columna más significativa. Los préstamos se propagan hacia la izquierda de columna en columna, como se ilustra a continuación.

EJEMPLO 1.4

Restar $(10111)_2$ a $(1001101)_2$.

	6	5	4	3	2	1	0	Columna
		1			*10*			*Préstamos*
	0	*X Ø*	*10*	*0*	*Ø*	*10*		*Préstamos*
	X	*Ø*	*Ø*	*X*	*X*	*Ø*	1	Minuendo
–			1	0	1	1	1	Sustraendo
		1	1	0	1	1	0	Diferencia

En este ejemplo, el primer préstamo aparece en la columna 1. El préstamo se toma de la columna 2, lo que produce 10 en la columna 1 y 0 en la columna 2. El 0 de la columna 2 necesita ahora un préstamo de la columna 3. No se necesitan otros préstamos hasta la columna 4. En este caso, no hay un 1 en la columna 5 que pueda prestar; por tanto, debemos primero tomar prestado el 1 de la columna 6, lo que produce 0 en la columna 6 y 10 en la columna 5. Ahora, la columna 4 toma prestado 1 de la columna 5, lo que deja 1 en la columna 5 ($10 - 1 = 1$) y 10 en la columna 4. Esta serie de préstamos se muestra arriba de los términos del minuendo.

Multiplicación y división

La multiplicación binaria se realiza en forma similar a la multiplicación decimal, excepto que las operaciones de multiplicación binaria son mucho más sencillas, como se puede ver en la tabla 1.3b. No obstante, se debe tener mucho cuidado al sumar los productos parciales, como se ilustra en el siguiente ejemplo.

EJEMPLO 1.5

Multiplicar $(10111)_2$ por $(1010)_2$.

				1	0	1	1	1	Multiplicando
×					1	0	1	0	Multiplicador
				0	0	0	0	0	
			1	0	1	1	1		
		0	0	0	0	0			
	1	0	1	1	1				
	1	1	1	0	0	1	1	0	Producto

Observe que hay un producto parcial por cada bit del multiplicador. Este procedimiento puede realizarse con mayor eficiencia si sólo recorremos una columna a la izquierda, en vez de anotar un producto parcial con ceros para un bit 0 del multiplicador. Este ejemplo nos sirve para ver lo sencillo de este procedimiento.

La división binaria se realiza utilizando el mismo procedimiento de prueba y error de la división decimal. Sin embargo, la división binaria es más sencilla pues sólo hay que intentar con dos valores. Se restan del dividendo copias de los términos del divisor, de lo cual se obtienen residuos intermedios positivos. El siguiente ejemplo ilustra la división binaria.

EJEMPLO 1.6

Dividir $(1110111)_2$ entre $(1001)_2$.

							1	1	0	1		Cociente
Divisor	1	0	0	1	1	1	1	0	1	1	1	Dividendo
					1	0	0	1				
						1	0	1	1			
						1	0	0	1			
								1	0	1	1	
								1	0	0	1	
										1	0	Residuo

1.2.2 Aritmética octal

Las tablas de la suma y la multiplicación para el sistema de numeración octal aparecen en la tabla 1.4. Con la ayuda de estas tablas, la aritmética octal se puede realizar utilizando el mismo procedimiento que en los sistemas decimal y binario, como se ilustra en los siguientes cuatro ejemplos.

EJEMPLO 1.7

Calcular $(4163)_8 + (7520)_8$.

	1		1			Acarreos
		4	1	6	3	Sumando
+		7	5	2	0	Sumando
	1	3	7	0	3	Suma

EJEMPLO 1.8

Calcular $(6204)_8 - (5173)_8$.

		1	10			Prestado
	6	~~2~~	~~0~~	4		Minuendo
−	5	1	7	3		Sustraendo
	1	0	1	1		Diferencia

EJEMPLO 1.9

Calcular $(4167)_8 \times (2503)_8$.

			4	1	6	7		Multiplicando
		×	2	5	0	3		Multiplicador
		1	4	5	4	5		Productos parciales
	2	5	1	2	3	0		
1	0	3	5	6				
1	3	1	0	5	0	4	5	Producto

TABLA 1.4 (a) TABLA DE SUMA OCTAL; (b) TABLA DE MULTIPLICACIÓN OCTAL

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

(a)

X	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

(b)

EJEMPLO 1.10

Calcular $(4163)_8 \div (25)_8$.

				1	4	7	Cociente
Divisor	2	5	4	1	6	3	Dividendo
			2	5			
			1	4	6		
			1	2	4		
				2	2	3	
				2	2	3	
						0	Residuo

1.2.3 Aritmética hexadecimal

Las tablas de la suma y la multiplicación hexadecimales son más complejas que las de los sistemas numéricos estudiados anteriormente, y aparecen en la tabla 1.5. De cualquier manera, al igual que en los otros sistemas numéricos, el

TABLA 1.5 TABLAS DE SUMA Y MULTIPLICACIÓN HEXADECIMAL (a) TABLA DE SUMA HEXADECIMAL (b) TABLA DE MULTIPLICACIÓN HEXADECIMAL

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

(a)

×	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

(b)

conocimiento de estas tablas permite realizar la aritmética hexadecimal con procedimientos bien conocidos. Los siguientes cuatro ejemplos ilustran la aritmética hexadecimal.

EJEMPLO 1.11

Calcular $(2A58)_{16} + (71D0)_{16}$.

		1			Acarreos
	2	*A*	5	8	Sumando
+	7	1	*D*	0	Sumando
	9	*C*	2	8	Suma

EJEMPLO 1.12

Calcular $(9F1B)_{16} - (4A36)_{16}$.

		E	11		Préstamos
	9	*F*	*X*	*B*	Minuendo
−	4	*A*	3	6	Sustraendo
	5	4	*E*	5	Diferencia

EJEMPLO 1.13

Calcular $(5C2A)_{16} \times (71D0)_{16}$.

			5	*C*	2	*A*		Multiplicando
		×	7	1	*D*	0		Multiplicador
		4	*A*	*E*	2	2	0	Productos parciales
		5	*C*	2	*A*			
2	8	5	2	6				
2	8	*F*	9	6	*C*	2	0	Producto

EJEMPLO 1.14

Calcular $(27FCA)_{16} \div (3E)_{16}$.

			A	5	1	Cociente
Divisor 3	*E*	2	7	*F*	*C*	*A* Dividendo
		2	6	*C*		
			1	3	*C*	
			1	3	6	
					6	*A*
					3	*E*
					2	*C* Residuo

1.3 Conversiones de base

Con frecuencia, los usuarios y diseñadores de computadoras y otros sistemas digitales tienen necesidad de convertir un número dado en base *A* a su equivalente en base *B*. En esta sección presentamos e ilustramos los algoritmos para realizar conversiones de base.

1.3.1 Métodos de conversión

Sustitución de una serie

La representación polinomial de un número dado por la ecuación 1.2 es la base del método de conversión por *sustitución de una serie*. La ecuación puede expandirse como sigue:

$$N = a_{n-1}r^{n-1} + \ldots + a_0 r^0 + a_{-1}r^{-1} + \ldots + a_{-m}r^{-m} \qquad (1.3)$$

Un número en base A puede convertirse en un número en base B en dos pasos.

1. Se forma la representación en serie del número en base A en el formato de la ecuación 1.3.
2. Se evalúa la serie utilizando la aritmética de base B.

Los siguientes cuatro ejemplos ilustran este procedimiento.

EJEMPLO 1.15

Convertir $(10100)_2$ a base 10.

Realizamos esta conversión sustituyendo cada dígito de acuerdo con su peso. Si contamos de derecha a izquierda en $(10100)_2$, vemos que el dígito del extremo derecho, 0, tiene un peso de 2^0, el siguiente dígito, 2^1, y así sucesivamente. Al sustituir estos valores en la ecuación 1.3 y evaluar la serie con aritmética de base 10 obtenemos

$$\begin{aligned} N &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= (16)_{10} + 0 + (4)_{10} + 0 + 0 \\ &= (20)_{10} \end{aligned}$$

EJEMPLO 1.16

Convertir $(274)_8$ a base 10.

$$\begin{aligned} N &= 2 \times 8^2 + 7 \times 8^1 + 4 \times 8^0 \\ &= (128)_{10} + (56)_{10} + (4)_{10} \\ &= (188)_{10} \end{aligned}$$

EJEMPLO 1.17

Convertir $(1101.011)_2$ a base 8.

La parte entera del número se convierte como en los ejemplos anteriores. Con los dígitos que están a la derecha del punto binario, contamos de izquierda a derecha. El primer dígito a la derecha del punto binario, 0, tiene un peso de 2^{-1}, el siguiente dígito, 1, tiene un peso de 2^{-2} y el tercer dígito, 1, tiene un peso de 2^{-3}. Al sustituir en la ecuación 1.3 obtenemos

$$\begin{aligned} N &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= (10)_8 + (4)_8 + 0 + (1)_8 + 0 + (.2)_8 + (.1)_8 \\ &= (15.3)_8 \end{aligned}$$

EJEMPLO 1.18

Convertir $(AF3.15)_{16}$ a base 10.

$$\begin{aligned} N &= A \times 16^2 + F \times 16^1 + 3 \times 16^0 + 1 \times 16^{-1} + 5 \times 16^{-2} \\ &= 10_{10} \times 256_{10} + 15_{10} \times 16_{10} + 3_{10} \times 1_{10} \\ &\quad + 1_{10} \times 0.0625_{10} + 5_{10} \times 0.00390625_{10} \\ &= 2560_{10} + 240_{10} + 3_{10} + 0.0625_{10} + 0.01953125_{10} \\ &= (2803.08203125)_{10} \end{aligned}$$

En los ejemplos anteriores, observe que los cálculos fueron más sencillos para las conversiones de base A a base B cuando $A < B$. Ahora describiremos algunos métodos de conversión en donde es cierto lo contrario.

Método de división entre la base

El método de conversión de *división entre la base* se puede utilizar para convertir un entero en base A al entero equivalente en base B. Para comprender el método, consideremos la siguiente representación de un entero N_I.

$$(N_I)_A = b_{n-1}B^{n-1} + \ldots + b_0B^0 \qquad (1.4)$$

En la ecuación 1.4, los b_i representan los dígitos de $(N_I)_B$ en base A. Podemos determinar el dígito menos significativo, $(b_0)_A$, al dividir $(N_I)_A$ entre $(B)_A$ como sigue:

$$\begin{aligned} N_I/B &= (b_{n-1}B^{n-1} + \ldots + b_1B^1 + b_0B^0)/B \\ &= \underbrace{b_{n-1}B^{n-2} + \ldots + b_1B^0}_{\text{Cociente, } Q_1} + \underbrace{b_0}_{\text{Residuo, } R_0} \end{aligned}$$

En otras palabras, $(b_0)_A$ es el residuo obtenido al dividir $(N_I)_A$ entre $(B)_A$. En general, $(b_i)_A$ es el residuo R_i obtenido al dividir el cociente Q_i entre $(B)_A$. La conversión termina al convertir cada $(b_i)_A$ a base B. Sin embargo, este último paso es trivial si $B < A$. El procedimiento para la conversión mediante la división entre la base se resume como sigue.

1. Dividimos $(N_I)A$ entre la base deseada $(B)A$, obteniendo el cociente Q_1 y el residuo R_0. R_0 es el dígito menos significativo, d_0, del resultado.
2. Calculamos cada uno de los dígitos restantes, d_i, para $i = 1 \ldots n\text{-}1$, dividiendo el cociente Q_i entre $(B)_A$ para obtener el cociente Q_{i+1} y el residuo R_i, que representa a d_i.
3. Nos detenemos cuando el cociente $Q_{i+1} = 0$.

El método de división entre la base se ilustra en los siguientes dos ejemplos.

EJEMPLO 1.19

Convertir $(234)_{10}$ a base 8.

Resolvemos este problema dividiendo repetidamente el entero $(234)_{10}$ (es decir, $(N)_A$, entre 8 (es decir, $(B)_A$, hasta que el cociente sea 0.

$$\begin{array}{r|l} 8 & 234 \\ \hline & 16 \\ \hline & 74 \\ & 72 \\ \hline & 2 = b_0 \end{array} \qquad \begin{array}{r|l} 8 & 29 \\ \hline & 24 \\ \hline & 5 = b_1 \end{array} \qquad \begin{array}{r|l} 8 & 3 \\ \hline & 0 \\ \hline & 3 = b_2 \end{array}$$

Por tanto, $(234)_{10} = (352)_8$. Estos cálculos se pueden resumir en el siguiente formato breve:

$$\begin{array}{r|l|l} 8 & 234 & 2 \\ 8 & 29 & 5 \\ 8 & 3 & 3 \\ & 0 & \end{array} \quad \begin{array}{l} \uparrow \\ \\ \\ \\ \end{array} \quad \begin{array}{l} \text{LSB} \\ \\ \text{MSB} \\ \\ \end{array}$$

EJEMPLO 1.20

Convertir $(234)_{10}$ a base 16.

$$\begin{array}{r|l} 16 & 234 \\ \hline & 16 \\ \hline & 74 \\ & 64 \\ \hline & 10 = (A)_{16} = b_0 \end{array} \qquad \begin{array}{r|l} 16 & 14 \\ \hline & 0 \\ \hline & 14 = (E)_{16} = b_1 \end{array}$$

Por tanto, $(234)_{10} = (EA)_{16}$. En notación breve:

$$\begin{array}{r|l|l} 16 & 234 & 10 = (A)_{16} \\ 16 & 14 & 14 = (E)_{16} \\ & 0 & \end{array} \quad \uparrow$$

Método de multiplicación por la base

Las conversiones de base para fracciones pueden realizarse mediante el método de *multiplicación por la base*. Sea *NF* una fracción en base *A*. La fracción se puede escribir como una serie:

$$(N_F)_A = b_{-1}B^{-1} + b_{-2}B^{-2} + \ldots + b_{-m}B^{-m} \qquad (1.5)$$

Los b_i de la ecuación 1.5 representan los dígitos de $(N_F)_B$ en base A. El dígito más significativo $(b_{-1})_A$ se puede obtener al multiplicar $(N_F)_A$ por $(B)_A$ como sigue:

$$\begin{aligned} B \times N_F &= B \times (b_{-1}B^{-1} + b_{-2}B^{-2} + \ldots + b_{-m}B^{-m}) \\ &= \underbrace{b_{-1}}_{\text{Entero, } I_{-1}} + \underbrace{b_{-2}B^{-1} + \ldots + b_{-m}B^{-(m-1)}}_{\text{Fracción, } F_{-2}} \end{aligned}$$

Así, $(b_{-1})_A$ es la parte entera del producto que resulta de multiplicar $(N_F)_A$ por $(B)_A$. En general, $(b_{-i})_A$ es la parte entera I_{-i} del producto que resulta de multiplicar la fracción $F_{-(i+1)}$ por $(B)_A$. Por tanto, el procedimiento de multiplicación por la base se resume como sigue:

1. Sea $F_{-1} = (N_F)_A$.
2. Calculamos los dígitos $(b_{-i})_A$, para $i = 1...m$, multiplicando F_i por $(B)_A$ para obtener el entero I_{-i} que representa al dígito $(b_{-i})_A$ y la fracción $F_{-(i+1)}$.
3. Convertimos cada dígito $(b_{-i})_A$ a la base B.

Los siguientes dos ejemplos ilustran este método.

EJEMPLO 1.21

Convertir $(0.1285)_{10}$ a base 8.

0.1285	0.0280	0.2240	0.7920
× 8	× 8	× 8	× 8
1.0280	0.2240	1.7920	6.3360
↑	↑	↑	↑
b_{-1}	b_{-2}	b_{-3}	b_{-4}

0.3360	0.6880	0.5040	0.0320
× 8	× 8	× 8	× 8
2.6880	5.5040	4.0320	0.2560
↑	↑	↑	↑
b_{-5}	b_{-6}	b_{-7}	b_{-8}

Así,

$$0.1285_{10} = (0.10162540...)_8$$

EJEMPLO 1.22

Convertir $(0.828125)_{10}$ a base 2.

Utilizaremos una notación abreviada en este ejemplo al aplicar el método de multiplicación por la base. En cada línea, la fracción se multiplica por 2 para obtener la línea siguiente:

$$
\begin{array}{rl}
\text{MSD} & 1.656250 \leftarrow 0.828125 \times 2 \\
 & 1.312500 \leftarrow 0.656250 \times 2 \\
 & 0.625000 \leftarrow 0.312500 \times 2 \\
 & 1.250000 \leftarrow 0.625000 \times 2 \\
 & 0.500000 \leftarrow 0.250000 \times 2 \\
\text{LSD} & 1.000000 \leftarrow 0.500000 \times 2
\end{array}
$$

$$0.828125_{10} = (0.110101)_2$$

1.3.2 Algoritmos generales de conversión

Los ejemplos presentados hasta ahora ilustran los principios de la conversión de bases. Con frecuencia, es útil definir procedimientos generales para resolver diversos problemas de modo que se puedan aplicar los pasos básicos en el orden adecuado. Ahora formularemos los métodos de conversión de base que utilizamos como dos algoritmos de conversión generalizados.

Algoritmo 1.1

Para convertir un número N de base A a base B, utilizamos

(a) el método de sustitución de una serie con aritmética de base B, o

(b) el método de división entre la base o multiplicación por la base con aritmética de base A.

Podemos utilizar el algoritmo 1.1 para la conversión entre dos bases cualesquiera. Sin embargo, al hacer esto, tal vez sea necesario utilizar la aritmética en una base poco familiar. El siguiente algoritmo supera esta dificultad a costa de un procedimiento más largo.

Algoritmo 1.2

Para convertir un número N de base A a base B, utilizamos

(a) el método de sustitución de una serie con aritmética de base 10 para convertir N de base A a base 10, y

(b) el método de división entre la base o multiplicación por la base con aritmética decimal, para convertir N de base 10 a base B.

En general, el algoritmo 1.2 requiere más pasos que el algoritmo 1.1, Pero, este último es más sencillo, rápido, y menos propenso a errores, ya que toda la aritmética se realiza en forma decimal.

EJEMPLO 1.23

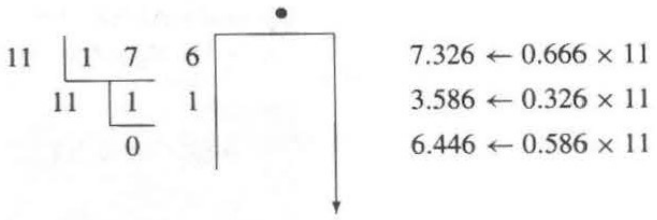
Convertir $(18.6)_9 = (?)_{11}$

$$NA = (18.6)9$$

a. Al convertir a base 10 por sustitución de una serie obtenemos

$$\begin{aligned} N_{10} &= 1 \times 9^1 + 8 \times 9^0 + 6 \times 9^{-1} \\ &= 9 + 8 + 0.666... \\ &= (17.666...)_{10} \end{aligned}$$

b. Al convertir de base 10 a base 11 por división entre la base obtenemos



Al unir las partes entera y fraccionaria,

$$N_{11} = (16.736...)_{11}.$$

1.3.3 Conversión entre la base *A* y la base *B* cuando $B = A^k$

Podemos utilizar procedimientos de conversión más sencillos cuando una base es una potencia de la otra, o sea, $B = A^k$. Los siguientes procedimientos son muy útiles.

Algoritmo 1.3

(a) Para convertir un número N de base A a base B cuando $B = A^k$ y k es un entero positivo, agrupamos los dígitos de N en grupos de k dígitos en ambas direcciones a partir del punto y después remplazamos cada grupo por el dígito equivalente en base B.

(b) Para convertir un número N de base B a base A cuando $B = A^k$ y k es un entero positivo, remplazamos cada dígito en base B de N por los k dígitos equivalentes en base A.

Los siguientes ejemplos ilustran la potencia y velocidad de este algoritmo para el caso $A = 2$.

EJEMPLO 1.24

Convertir $(1011011.1010111)_2$ a base 8.

Podemos aplicar el algoritmo 1.3a, con $B = 8 = 2^3 = A^k$. Por tanto, agrupamos tres dígitos binarios por cada dígito octal.

$$\underbrace{001}_{1}\ \underbrace{011}_{3}\ \underbrace{011}_{3}\ .\ \underbrace{101}_{5}\ \underbrace{011}_{3}\ \underbrace{100}_{4}$$

$$1011011.1010111_2 = (133.534)_8$$

EJEMPLO 1.25

Convertir $(AF.16C)_{16}$ a base 8.

Como 16 y 8 son potencias de 2, podemos aplicar el algoritmo 1.3 dos veces, como sigue.

- Usamos el algoritmo 1.3b para convertir $(AF.16C)16$ a base 2, ya que $16 = 2^4$. Remplazamos cada dígito hexadecimal por cuatro dígitos binarios.

$$\underbrace{1010}_{A}\ \underbrace{1111}_{F}\ .\ \underbrace{0001}_{1}\ \underbrace{0110}_{6}\ \underbrace{1100}_{C}$$

$$(AF.16C)_{16} = (10101111.0001011011)_2$$

- Usamos el algoritmo 1.3a para convertir el número binario a base 8.

$$\underbrace{010}_{2}\ \underbrace{101}_{5}\ \underbrace{111}_{7}\ .\ \underbrace{000}_{0}\ \underbrace{101}_{5}\ \underbrace{101}_{5}\ \underbrace{100}_{4}$$

Por tanto,

$$(AF.16C)_{16} = (257.0554)_8$$

1.4 Representación de números con signo

El signo de los números almacenados en los sistemas digitales se especifica mediante un dígito llamado *dígito de signo*, que por lo general se coloca en la posición de extrema izquierda de los dígitos del número, como se ilustra en la figura 1.2. Los números positivos se especifican con un dígito de signo igual a cero, y los negativos, con un dígito de signo distinto de cero. La magnitud de un número positivo se representa simplemente mediante sus dígitos posicionales, pero existen varios métodos para representar la magnitud de los números negativos. La tabla 1.6 ilustra los métodos de magnitud y signo, de complemento a la base y de complemento disminuido a la base. A continuación analizaremos cada método con más detalle.

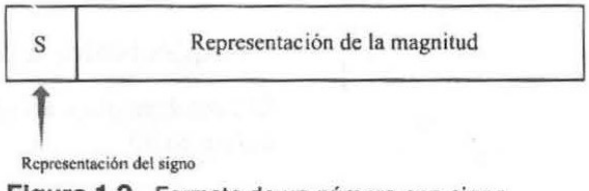


Figura 1.2 Formato de un número con signo

1.4.1 Números con magnitud y signo

El método más sencillo para representar los números con signo es el de *magnitud y signo*. Sin embargo, el empleo de este método requiere circuitos aritméticos y algoritmos con un costo mayor en términos de componentes y de tiempo de cálculo, en relación con otros métodos; por tanto, no suele utilizarse en la práctica para representar números enteros.

Podemos escribir un número con signo $N = \pm(a_{n-1}...a_0.a_{-1}...a_{-m})_r$ en el formato de magnitud y signo como sigue.

$$N = (sa_{n-1}...a_0.a_{-1}...a_{-m})_{rsm} \tag{1.6}$$

donde $s = 0$ si N es positivo y $s = r - 1$ si N es negativo.

EJEMPLO 1.26

Determinar el código de magnitud y signo de $N = -(13)_{10}$ en binario ($r = 2$) y decimal ($r = 10$).

En binario:

$$\begin{aligned} N &= -(13)_{10} \\ &= -(1101)_2 \\ &= (1,1101)_{2sm} \end{aligned}$$

En decimal:

$$\begin{aligned} N &= -(13)_{10} \\ &= (9, 13)_{10sm} \end{aligned}$$

donde 9 representa el signo negativo para $r = 10$.

En la tabla 1.6 se dan más ejemplos de números binarios con magnitud y signo. Para mayor claridad, utilizamos las comas para delimitar los dígitos de signo.

1.4.2 Sistemas numéricos complementarios

Los números complementarios son la base de la aritmética complementaria, un método de gran utilidad que se emplea con frecuencia en los sistemas digitales para realizar operaciones matemáticas con números con signo. En estos sistemas, los números positivos se representan de la misma manera que en un sistema con magnitud y signo, mientras que los números negativos se representan como el complemento del número positivo correspondiente. El complemento a una base y el complemento disminuido a una base son sistemas numéricos importantes que analizaremos a continuación. La aritmética complementaria se ilustra también con ejemplos en este capítulo.

Complementos a una base

El *complemento a una base* $[N]_r$ de un número $(N)_r$ dado por la ecuación 1.2 se define como

$$[N]_r = r^n - (N)_r \tag{1.7}$$

TABLA 1.6 EJEMPLOS DE REPRESENTACIÓN DE NÚMEROS CON SIGNO*

Signo decimal	Binario en magnitud y signo	Sistema de complemento a dos	Sistema de complemento a uno
+15	0,1111	0,1111	0,1111
+14	0,1110	0,1110	0,1110
+13	0,1101	0,1101	0,1101
+12	0,1100	0,1100	0,1100
+11	0,1011	0,1011	0,1011
+10	0,1010	0,1010	0,1010
+9	0,1001	0,1001	0,1001
+8	0,1000	0,1000	0,1000
+7	0,0111	0,0111	0,0111
+6	0,0110	0,0110	0,0110
+5	0,0101	0,0101	0,0101
+4	0,0100	0,0100	0,0100
+3	0,0011	0,0011	0,0011
+2	0,0010	0,0010	0,0010
+1	0,0001	0,0001	0,0001
0	0,0000	0,0000	0,0000
	(1,0000)		(1,1111)
-1	1,0001	1,1111	1,1110
-2	1,0010	1,1110	1,1101
-3	1,0011	1,1101	1,1100
-4	1,0100	1,1100	1,1011
-5	1,0101	1,1011	1,1010
-6	1,0110	1,1010	1,1001
-7	1,0111	1,1001	1,1000
-8	1,1000	1,1000	1,0111
-9	1,1001	1,0111	1,0110
-10	1,1010	1,0110	1,0101
-11	1,1011	1,0101	1,0100
-12	1,1100	1,0100	1,0011
-13	1,1101	1,0011	1,0010
-14	1,1110	1,0010	1,0001
-15	1,1111	1,0001	1,0000
-16	—	1,0000	—

* Note que los bits de signo están delimitados por comas.

donde n es el número de dígitos de $(N)_r$. El número positivo más grande (llamado *escala total positiva*) que puede representarse es $r^{n-1} - 1$, mientras que el número negativo más pequeño (llamado *escala total negativa*) es $-r^{n-1}$.

El *complemento a dos* es un caso especial del complemento a una base para números binarios ($r = 2$) y está dado por

$$[N]_2 = 2^n - (N)_2 \tag{1.8}$$

donde n es el número de bits de $(N)_2$. El complemento a dos es el formato de uso más común para los números con signo en los sistemas digitales y, por tanto, será el centro de la mayor parte de los ejemplos de este texto.

Los siguientes ejemplos ilustran la forma de determinar el complemento a dos de un número binario utilizando la ecuación 1.8.

Ejemplo 1.27

Determinar el complemento a dos de $(N)_2 = (01100101)_2$.

De la ecuación 1.8,

$$\begin{aligned}[N]_2 &= [01100101]_2 \\ &= 2^8 - (01100101)_2 \\ &= (100000000)_2 - (01100101)_2 \\ &= (10011011)_2.\end{aligned}$$

Ejemplo 1.28

Determinar el complemento a dos de $(N)_2 = (11010100)_2$ y verificar que puede servir para representar $-(N)_2$, demostrando que $(N)_2 + [N]_2 = 0$.

Primero, determinamos el complemento a dos mediante la ecuación 1.8:

$$\begin{aligned}[N]_2 &= [11010100]_2 \\ &= 2^8 - (11010100)_2 \\ &= (100000000)_2 - (11010100)_2 \\ &= (00101100)_2.\end{aligned}$$

Para verificar que $[N]_2$ puede servir para representar $-(N)_2$, calculemos $(N)_2 + [N]_2$:

$$\begin{array}{cccccccccc} & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

↑
acarreo

Si descartamos el bit de acarreo, obtenemos $(N)_2 + [N]_2 = (00000000)_2$; es decir, la suma de un número binario y su complemento a dos es 0. Por tanto, podemos utilizar $[N]_2$ para representar $-(N)_2$.

Ejemplo 1.29

Determinar el complemento a dos de $[N]_2 = (00101100)_2$, calculado en el ejemplo 1.28.

$$[[N]_2]_2 = [00101100]_2$$

$$= 2^8 - (00101100)_2$$
$$= (100000000)_2 - (00101100)_2$$
$$= (11010100)_2$$

Observe que el resultado es el valor original de $(N)_2$ dado en el ejemplo 1.28.

Del último ejemplo, vemos que si aplicamos dos veces la operación de complemento a dos a un número, obtenemos su valor original. Podemos verificar esto con facilidad para el caso general si sustituimos $-(N)_2$ por $[N]_2$, como sigue:

$$[[N]_2]_2 = [-(N)_2]_2$$
$$= -(-(N)_2)_2$$
$$= (N)_2$$

EJEMPLO 1.30

Determinar el complemento a dos de $(N)_2 = (10110)_2$ para $n = 8$.

De la ecuación 1.8,

$$[N]_2 = [10110]_2$$
$$= 2^8 - (10110)_2$$
$$= (100000000)_2 - (10110)_2$$
$$= (11101010)_2$$

Observe que mantenemos 8 bits en el resultado. Aconsejamos al lector verificar el hecho de que este valor de $[N]_2$ puede servir para representar $-(N)_2$ y que $[[N]_2]_2 = (N)_2$.

El siguiente ejemplo muestra que el procedimiento básico para determinar el complemento a una base de un número es el mismo para cualquier base.

EJEMPLO 1.31

Determinar el complemento a 10 de $(N)_{10} = (40960)_{10}$.

De la ecuación 1.7,

$$[N]_{10} = [40960]_{10}$$
$$= 10^5 - (40960)_{10}$$
$$= (100000)_{10} - (40960)_{10}$$
$$= (59040)_{10}.$$

Observe que mantenemos 5 dígitos en el resultado. El lector debe verificar que $[N]_{10}$ puede servir para representar $-(N)_{10}$ y que $[[N]_{10}]_{10} = (N)_{10}$.

Aunque siempre podemos determinar el complemento a una base de un número mediante la definición de la ecuación 1.7, hay métodos más sencillos. Presentamos los siguientes dos algoritmos para calcular $[N]_r$ dado $(N)_r$ sin demostración.

Algoritmo 1.4 Determinar $[N]_r$ dado $(N)_r$.

Copiamos los dígitos de N, a partir del menos significativo y procediendo hacia el más significativo, hasta llegar al primer dígito distinto de cero. Remplazamos este dígito, a_i, con $r - a_i$. Después, en caso necesario, continuamos remplazando cada dígito restante aj de N por $(r - 1) - a_j$ hasta haber remplazado el dígito más significativo.

Para el caso especial de los números binarios ($r = 2$), el primer dígito distinto de cero, a_i, es siempre 1. Por tanto, remplazamos a_i por $r - a_i = 2 - 1 = 1$, de modo que ai permanece sin cambio. Cada bit restante a_j se remplaza por $(r - 1) - a_j = 1 - a_j = \bar{a}_j$. Por tanto, la aplicación del algoritmo 1.4 a los números binarios sólo copia todos los bits hasta el primer bit igual a 1 inclusive, y complementa después los bits restantes.

Ejemplo 1.32

Determinar el complemento a dos de $N = (01100101)_2$.

$$\begin{array}{rcccccccc} N = & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ & & & & & & & & \updownarrow \\ [N]_2 = (1 & 0 & 0 & 1 & 1 & 0 & 1 & 1)_2 \end{array}$$

primer dígito distinto de cero

Ejemplo 1.33

Determinar el complemento a dos de $N = (11010100)_2$.

$$\begin{array}{rcccccccc} N = & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ & & & & & & \updownarrow & & \\ [N]_2 = (0 & 0 & 1 & 0 & 1 & 1 & 0 & 0)_2 \end{array}$$

primer dígito distinto de cero

Ejemplo 1.34

Determinar el complemento a dos de $N = (10110)_2$ para $n = 8$.

En primer lugar, como $n = 8$, debemos concatenar tres ceros en las posiciones de los bits más significativos para formar un número de 8 bits. Después aplicamos el algoritmo 1.4.

$$\begin{array}{rcccccccc} N = & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ & & & & & & & \updownarrow & \\ [N]_2 = (1 & 1 & 1 & 0 & 1 & 0 & 1 & 0)_2 \end{array}$$

primer dígito distinto de cero

Ejemplo 1.35

Determinar el complemento a diez de $(40960)_{10}$.

$$\begin{array}{rccccc} N = & 4 & 0 & 9 & 6 & 0 \\ & & & & \updownarrow & \\ [N]_{10} = (5 & 9 & 0 & 4 & 0)_{10} \end{array}$$

primer dígito distinto de cero

Algoritmo 1.5 Determinar $[N]_r$ dado $(N)_r$.

Primero remplazamos cada dígito, a_k, de $(N)_r$ por $(r - 1) - a_k$ y después sumamos 1 al resultado.

En el caso particular de los números binarios ($r = 2$), remplazamos cada bit a_k, por $(r - 1) - a_k = 1 - a_k = \overline{a}_k$. Por tanto, aplicamos el algoritmo 1.5 complementando cada bit y sumando 1 al resultado.

EJEMPLO 1.36

Determinar el complemento a dos de $N = (01100101)_2$.

$$\begin{aligned} N = \ & 01100101 \\ & 10011010 \quad \text{Complementar los bits} \\ & +1 \quad \text{Sumar 1} \\ [N]_2 = \ & (10011011)_2 \end{aligned}$$

EJEMPLO 1.37

Determinar el complemento a dos de $N = (11010100)_2$.

$$\begin{aligned} N = \ & 11010100 \\ & 00101011 \quad \text{Complementar los bits} \\ & +1 \quad \text{Sumar 1} \\ [N]_2 = \ & (00101100)_2 \end{aligned}$$

EJEMPLO 1.38

Determinar el complemento a diez de $(40960)_{10}$.

$$\begin{aligned} N = \ & 40960 \\ & 59039 \quad \text{Complementar los dígitos} \\ & +1 \quad \text{Sumar 1} \\ [N]_{10} = \ & (59040)_{10} \end{aligned}$$

Observe que el algoritmo 1.4 es adecuado para los cálculos a mano, mientras que el 1.5 es más útil para implantación en máquina, pues no implica toma de decisiones.

Sistemas numéricos de complemento a una base

Hemos definido el complemento a una base, y presentado e ilustrado varios métodos para determinar el complemento a una base de un número dado. También sugerimos, por ejemplo, que podemos utilizar el complemento a una base de un número dado para representar el negativo de ese número. Ahora describiremos con más precisión un sistema numérico que utiliza el complemento a dos para representar los números negativos. Podemos definir de manera análoga sistemas para otras bases.

En el *sistema numérico de complemento a dos*, los valores positivos se representan de la misma forma que en el sistema de magnitud y signo; utilizamos un bit de 0 a la izquierda para representar el signo. Los números negativos se representan con el complemento a dos del número positivo correspondiente. Utilizaremos la notación $(N)_{2cns}$ para denotar un número representado en el sistema numérico de complemento a dos. Así, $N = +(a_{n-2},\ldots,a_0)_2 = (0,a_{n-2},\ldots,a_0)_{2cns}$, donde $0 \le N \le 2^{n-1} - 1$. Si $N = (a_{n-1},a_{n-2},\ldots,a_0)_2$, entonces $-N$ se representa en el sistema numérico de complemento a dos como $[a_{n-1},\ldots,a_0]_2$, donde $-1 \ge -N \ge -2^{n-1}$. Todos los números negativos en el sistema numérico de complemento a dos tienen un bit de signo igual a 1. La tabla 1.6 muestra los códigos del sistema numérico de complemento a dos para $n = 5$.

Los siguientes ejemplos ilustran la codificación de los números positivos y negativos en el sistema numérico de complemento a dos. Se aconseja al lector verificar los datos de complemento a dos de la tabla 1.6 después de estudiar los ejemplos.

EJEMPLO 1.39

Dado $(N)_2 = (1100101)_2$, determinar las representaciones de $\pm(N)_2$ en el sistema numérico de complemento a dos, para $n = 8$.

Por inspección,

$$+(N)_2 = (0, 1100101)_{2cns}$$

De la ecuación 1.8,

$$\begin{aligned} -(N)_2 &= [+(N)_2]_2 \\ &= [0, 1100101]_2 \\ &= 2^8 - (0, 1100101)_2 \\ &= (100000000)_2 - (0, 1100101)_2 \\ &= (1, 0011011)_{2cns}. \end{aligned}$$

Por sus bits de signo, vemos que $(0,1100101)_{2cns}$ representa un valor positivo y $(1,0011011)_{2cns}$ es su negativo. En este ejemplo y los posteriores, utilizaremos una coma para facilitar la identificación del bit de signo.

EJEMPLO 1.40

Determinar las representaciones en el sistema numérico de complemento a dos de $\pm(110101)_2$ para $n = 8$.

Por inspección,

$$+(110101)_2 = (0, 0110101)_{2cns}$$

De la ecuación 1.8,

$$\begin{aligned} -(110101)_2 &= [110101]_2 \\ &= 2^8 - (110101)_2 \\ &= (100000000)_2 - (110101)_2 \\ &= (1, 1001011)_{2cns} \end{aligned}$$

EJEMPLO 1.41

Determinar la codificación en el sistema numérico de complemento a dos de $-(13)_{10}$ para $n = 8$.

Comenzamos con la conversión de $(13)_{10}$ de decimal a binario.

$$+(13)_{10} = +(1101)_2 = (0,0001101)_{2cns}$$

Después calculamos el complemento a dos de $(0,0001101)_{2cns}$ para representar $-(13)_{10}$:

$$\begin{aligned} -(13)_{10} &= -(0, 0001101)_{2cns} \\ &= [0, 0001101]_2 \\ &= 2^8 - (0, 0001101)_2 \\ &= (1, 1110011)_{2cns} \end{aligned}$$

EJEMPLO 1.42

Determinar el número decimal representado por $N = (1,1111010)_{2cns}$.

Por bit de signo, vemos que N es un número negativo. Así, determinamos la magnitud de N (el valor positivo correspondiente) calculando su complemento a dos.

$$\begin{aligned} N &= (1, 1111010)_{2cns} \\ &= -[1, 1111010]_2 \\ &= -(2^8 - (1, 1111010)_2) \\ &= -(0, 0000110)_{2cns} \\ &= -(6)_{10} \end{aligned}$$

donde $(0,0000110)_{2cns} = +(6)_{10}$. Por tanto, $(1,1111010)_{2cns}$ representa a $-(6)_{10}$.

Consideremos ahora algunos ejemplos de la aritmética con números de complemento a una base.

Aritmética de complemento a una base

Muchas computadoras digitales utilizan un sistema numérico de complemento a una base a fin de minimizar la cantidad de circuitos necesarios para realizar la aritmética de enteros. Por ejemplo, podemos realizar la operación $A - B$ calculando $A + (-B)$, donde $(-B)$ está representado por el complemento a dos de B. Por tanto, la computadora sólo necesita un sumador binario y algunos circuitos complementarios para la suma y la resta. Este punto de vista es conveniente para el análisis de la aritmética de complemento a una base y por ello lo utilizaremos en los párrafos siguientes. Como en una computadora la aritmética se realiza principalmente en binario, centraremos nuestro análisis en la aritmética de complemento a dos.

Antes de iniciar un análisis profundo, consideremos una limitación fundamental de la representación de números en la máquina. Las máquinas del tipo de las computadoras digitales operan con sistemas de números finitos impuestos por el número de bits que se pueden utilizar en la representación de cantidades numéricas. El número de bits disponibles en la unidad aritmética de la computadora limita el intervalo numérico que se puede representar en la máquina. El sistema no puede manejar números fuera de este intervalo. Las máquinas que utilizan el sistema numérico de complemento a dos (2cns) pueden representar enteros en el intervalo

$$-2^{n-1} \leq N \leq 2^{n-1} - 1 \qquad (1.9)$$

donde n es el número de bits disponibles para representar N. Observe que $2^{n-1} - 1 = (0,11...1)_{2cns}$ y que $-2^{n-1} = (1,00...0)_{2cns}$ (el bit del extremo izquierdo representa el signo y los $n - 1$ bits restantes representan la magnitud).

Si una operación produce un resultado fuera del intervalo disponible definido en la ecuación 1.9; es decir, si $N > 2^{n-1} - 1$ o $N < -2^{n-1}$, decimos que se presenta una *condición de desbordamiento*. En estos casos, el número de n bits originado por la operación no será una representación válida del resultado. Las computadoras digitales vigilan sus operaciones aritméticas al realizar aritmética

de complemento a dos y generan una señal de advertencia cuando se presenta un desbordamiento, a fin de que los números no válidos no sean considerados como resultados correctos.

Ahora consideraremos tres casos para ilustrar la aritmética en el sistema numérico de complemento a dos: $A = B + C$, $A = B - C$ y $A = -B - C$. Describiremos cada caso en general y después lo aclararemos con ejemplos apropiados. Para todos los casos, suponga que $B \geq 0$ y $C \geq 0$. Los resultados se pueden generalizar fácilmente para incluir valores negativos de B y C.

Caso 1. Calcular $A = B + C$. Como B y C son no negativos, A también será no negativo, y esto simplemente se convierte en

$$(A)_2 = (B)_2 + (C)_2$$

Como los tres números son positivos, no tenemos que utilizar el complemento a dos.

La única dificultad que puede surgir en este caso es cuando $A > 2^{n-1} - 1$, es decir, cuando se presenta un desbordamiento. Es fácil detectar tal condición, ya que el bit de signo de A será incorrecto. Para mostrar esto, consideremos la suma de los dos máximos números positivos de n bits representables.

$$0 \leq A \leq (2^{n-1} - 1) + (2^{n-1} - 1) = 2^n - 2$$

Como el máximo valor positivo de n bits representable es $2^{n-1} - 1$, se presenta una condición de desbordamiento para cualquier suma en el intervalo

$$A \geq 2^{n-1}$$

El $n^{\text{ésimo}}$ bit de un número binario en este intervalo adoptará el valor 1. Por desgracia, éste es el bit que representa el signo en un número de n bits de complemento a dos. Por tanto, el resultado parece ser negativo, lo que indica la condición de desbordamiento.

Debe observarse que si $A < 2^n$, nunca habrá un acarreo más allá del $n^{\text{ésimo}}$ bit del sumador binario.

Los siguientes ejemplos utilizarán el sistema numérico de complemento a dos de 5 bits, cuyos valores se dan en la tabla 1.6.

EJEMPLO 1.43

Calcular $(9)_{10} + (5)_{10}$ con aritmética de complemento a dos de 5 bits.

Comenzamos escribiendo $(9)_{10}$ y $(5)_{10}$ como números de complemento a dos de 5 bits. Puesto que ambos números son positivos, utilizamos un bit de signo 0 para cada uno. De la tabla 1.6,

$$+(9)_{10} = +(1001)_2 = (0, 1001)_{2cns}$$
$$+(5)_{10} = +(0101)_2 = (0, 0101)_{2cns}$$

Al sumar estos dos códigos de 5 bits obtenemos

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 1 & 0 \end{array}$$

Como el resultado también tiene un bit de signo 0, representa correctamente la suma positiva deseada, que se interpreta como

$$(0, 1110)_{2cns} = +(1110)_2 = +(14)_{10}$$

EJEMPLO 1.44

Calcular $(12)_{10} + (7)_{10}$.

De la tabla 1.6,

$$+(12)_{10} = +(1100)_2 = (0,1100)_{2cns}$$
$$+(7)_{10} = +(0111)_2 = (0,0111)_{2cns}$$

Sumando los dos códigos de 5 bits resulta

	0	1	1	0	0
+	0	0	1	1	1
	1	0	0	1	1

El resultado es $(1,0011)_{2cns}$, que se interpreta, según la tabla 1.6, como

$$(1,0011)_{2cns} = -(1101)_2 = -(13)_{10}$$

Si examinamos más de cerca este cálculo vemos que ¡la suma de dos números positivos parece haber dado un resultado negativo! Sin embargo, esto no puede ser correcto, por lo que debe haber una explicación. La respuesta es que la suma de los dos números dados requiere más espacio que los 5 bits asignados para representarla. La suma correcta es $+(19)_{10}$, que está fuera del intervalo de números de complemento a dos de 5 bits, pues la escala total positiva es $(0,1111)_{2cns} = +(15)_{10}$. El bit de signo incorrecto obtenido en los cálculos indica un resultado incorrecto. Por tanto, hay una condición de desbordamiento.

Caso 2. Calcular $A = B - C$. Consideramos este cálculo como $A = B + (C)$ de la siguiente manera. Queremos calcular

$$A = (B)_2 + (-(C)_2)$$

Suponga que representamos esta operación codificando los números en complemento a dos. El número positivo $(B)_2$ no cambia, pero $-(C)_2$ se convierte en $[C]_2$:

$$\begin{aligned} A &= (B)_2 + [C]_2 \\ &= (B)_2 + 2^n - (C)_2 \\ &= 2^n + (B - C)_2 \end{aligned}$$

Por tanto, el cálculo equivale a $2^n + (B - C)$. Ésta es la respuesta deseada, excepto que aparece un término 2^n adicional. ¿Podemos ignorarlo? Si $B \geq C$, entonces $B - C \geq 0$, lo que hace $A \geq 2^n$. El término 2^n representa un bit de acarreo y se puede descartar, conservando $(B - C)_2$ (un sumador binario de n bits generará un acarreo para cualquier suma $A \geq 2^n$). Por tanto,

$$(A)_2 = (B)_2 + [C]_2 \text{ | acarreo descartado}$$

Si $B < C$, entonces $B - C < 0$, lo que da $A = 2^n - (C - B)_2 = [C - B]_2$, o $A = -(C - B)_2$ que es la respuesta deseada. Observe que en este caso no hay acarreo. Más adelante resumiremos todas las condiciones posibles en forma tabular.

Si B y C son ambos números positivos, la magnitud de $B - C$ siempre será menor que cualquiera de los dos números. Esto significa que no se puede presentar un desbordamiento al calcular $B - C$.

Ejemplo 1.45

Calcular $(12)_{10} - (5)_{10}$.

Realizamos este cálculo como $(12)_{10} + (-(5)_{10})$

$$(12)_{10} = +(1100)_2 = (0, 1100)_{2cns}$$
$$-(5)_{10} = -(0101)_2 = (1, 1011)_{2cns}$$

Sumamos los dos códigos de 5 bits para obtener

	0	1	1	0	0
+	1	1	0	1	1
1	0	0	1	1	1

↑
Acarreo

Al descartar el acarreo, el bit de signo es 0 y, por tanto, interpretamos el resultado como

$$(0, 0111)_{2cns} = +(0111)_2 = +(7)_{10}$$

Ejemplo 1.46

Invertir el orden de los operandos del ejemplo anterior, y calcular $(5)_{10} - (12)_{10}$.

Realizamos el cálculo como $(5)_{10} + (-(12)_{10})$.

$$(5)_{10} = +(0101)_2 = (0, 0101)_{2cns}$$
$$-(12)_{10} = -(1100)_2 = (1, 0100)_{2cns}$$

Sumamos los dos códigos de 5 bits para obtener

	0	0	1	0	1
+	1	0	1	0	0
	1	1	0	0	1

En este caso no hay acarreo y el bit de signo es 1, lo que indica el siguiente resultado negativo:

$$(1, 1001)_{2cns} = -(0111)_2 = -(7)_{10}$$

Ejemplo 1.47

Calcular $(0, 0111)_{2cns} - (1, 1010)_{2cns}$.

Realizamos los cálculos como $(0, 0111)_{2cns} + (-(1, 1010)_{2cns})$. El operando del lado izquierdo ya tiene el formato del sistema numérico de complemento a dos. Como su bit de signo es 1, el operando del lado derecho representa un número negativo. Para negarlo, calculamos el complemento a dos de este número negativo para obtener el valor positivo correspondiente. Observe que según la definición de complemento a dos es

$$\begin{aligned} -[A]_2 &= [[A]_2]_2 \\ &= 2^n - [A]_2 \\ &= 2^n - (2^n - A) \\ &= A \end{aligned}$$

Por tanto,

$$-(1, 1010)_{2cns} = (0, 0110)_{2cns}$$

Al sumar los dos códigos de 5 bits obtenemos

$$\begin{array}{cccccc} & 0 & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 & 0 \\ \hline & 0 & 1 & 1 & 0 & 1 \end{array}$$

El resultado es positivo, como lo indica el bit de signo 0, y lo interpretamos como

$$(0, 1101)_{2cns} = +(1101)_2 = +(13)_{10}$$

El lector debe verificar que este cálculo equivale a calcular $(7)_{10} - (-(6)_{10}) = (13)_{10}$.

Caso 3. Calcular $A = -B - C$. El resultado deseado es $A = -(B + C) = [B + C]_2$. Representamos $-B$ y $-C$ mediante el complemento a dos de sus magnitudes, y calculamos $A = (-B) + (-C)$. Por tanto,

$$\begin{aligned} A &= [B]_2 + [C]_2 \\ &= 2^n - (B)_2 + 2^n - (C)_2 \\ &= 2^n + 2^n - (B + C)_2 \\ &= 2^n + [B + C]_2 \end{aligned}$$

Si se descarta el bit de acarreo (2^n), el cálculo produce el resultado correcto, la representación en complemento a dos de $-(B + C)_2$.

EJEMPLO 1.48

Calcular $-(9)_{10} - (5)_{10}$.

Realizamos el cálculo como $(-(9)_{10}) + (-(5)_{10})$.

$$-(9)_{10} = -(1001)_2 = (1, 0111)_{2cns}$$
$$-(5)_{10} = -(0101)_2 = (1, 1011)_{2cns}$$

Al sumar los dos códigos de 5 bits obtenemos

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

↑
Acarreo

Al descartar el acarreo tenemos un bit de signo de 1. Por tanto, el resultado es correcto y se interpreta como

$$(1, 0010)_{2cns} = -(1110)_2 = -(14)_{10}$$

Como cuando se suman dos valores positivos, puede ocurrir un desbordamiento al sumar dos números negativos, para dar un resultado en el intervalo

$$A < -2^{n-1}$$

lo que se indica mediante un resultado que tiene su bit de signo incorrecto (es decir, un resultado que parece ser positivo). Ilustraremos esto en el siguiente ejemplo.

EJEMPLO 1.49

Calcular $-(12)_{10} - (5)_{10}$.

Realizamos el cálculo como $(-(12)_{10}) + (-(5)_{10})$

$$-(12)_{10} = -(1100)_2 = (1,0100)_{2cns}$$
$$-(5)_{10} = -(0101)_2 = (1,1011)_{2cns}$$

Al sumar los dos códigos de 5 bits obtenemos

	1	0	1	0	0
+	1	1	0	1	1
1	0	1	1	1	1

↑

Acarreo

Al descartar el acarreo, interpretamos el resultado como

$$(0,1111)_{2cns} = +(1111)_2 = +(15)_{10}$$

Observe que el bit de signo es incorrecto, lo que indica un desbordamiento. Este resultado es "demasiado negativo"; excede en 1 el intervalo numérico en la dirección negativa, pues el resultado deseado era $-(17)_{10}$. En consecuencia, debido al desbordamiento, el resultado se interpreta de manera incorrecta como $+(15)_{10}$.

El siguiente ejemplo muestra la utilidad de la aritmética de complemento a dos en las computadoras digitales.

EJEMPLO 1.50

A y B son variables enteras de un programa para computadora, con $A = (25)_{10}$ y $B = -(46)_{10}$. Si la computadora utiliza aritmética de complemento a dos con 8 bits, mostrar la forma como calcularía $A + B$, $A - B$, $B - A$ y $-A - B$.

Las variables A y B se guardarían en la memoria de la computadora con el formato de sistema numérico de 8 bits de complemento a dos:

$$A = +(25)_{10} = (0, 0011001)_{2cns}$$
$$B = -(46)_{10} = -(0, 0101110)_{2cns} = (1, 1010010)_{2cns}$$

En primer lugar, calcularemos los complementos a dos de A y B para representar $-A$ y $-B$, respectivamente:

$$-A = -(25)_{10} = -(0, 0011001)_{2cns} = (1, 1100111)_{2cns}$$
$$-B = -(-(46)_{10}) = -(1, 1010010)_{2cns} = (0, 0101110)_{2cns}$$

Al realizar los cálculos,

		0	0	0	1	1	0	0	1
$A + B$:	+	1	1	0	1	0	0	1	0
		1	1	1	0	1	0	1	1

El resultado es $(1, 1101011)_{2cns} = -(0, 0010101)_{2cns} = -(21)_{10}$.

		0	0	0	1	1	0	0	1
$A - B = A + (-B)$:	+	0	0	1	0	1	1	1	0
		0	1	0	0	0	1	1	1

El resultado es $(0, 1000111)_{2cns} = +(71)_{10}$.

		1	1	0	1	0	0	1	0
$B - A = B + (-A)$:	+	1	1	1	0	0	1	1	1
	1	1	0	1	1	1	0	0	1

El resultado es $(1, 0111001)_{2cns} = -(0, 1000111)_{2cns} = -(71)_{10}$.

		1	1	1	0	0	1	1	1
$-A - B = (-A) + (-B)$:	+	0	0	1	0	1	1	1	0
	1	0	0	0	1	0	1	0	1

El resultado es $(0, 0010101)_{2cns} = +(21)_{10}$. Observe que en los dos últimos casos se descarta el bit de acarreo.

En la tabla 1.7 damos un resumen de la suma y resta de complemento a dos.

TABLA 1.7 RESUMEN DE LA SUMA Y RESTA DE COMPLEMENTO A DOS

Caso*	Acarreo	Bit de signo	Condición	¿Desbordamiento?
$B + C$	0	0	$B + C \leq 2^{n-1} - 1$	No
	0	1	$B + C > 2^{n-1} - 1$	Sí
$B - C$	1	0	$B \leq C$	No
	0	1	$B > C$	No
$-B - C$	1	1	$-(B + C) > -2^{n-1}$	No
	1	0	$-(B + C) < -2^{n-1}$	Sí

* B y C son números positivos.

La aritmética de complemento a una base se puede utilizar con cualquier base no sólo con los números binarios. Para ilustrar este hecho, los siguientes dos ejemplos usarán la aritmética de complemento a 10 con números de tres dígitos.

EJEMPLO 1.51

Sumar $+(75)_{10}$ y $-(21)_{10}$ utilizando aritmética de complemento a 10 con tres dígitos.

En primer lugar, determinamos los códigos de complemento a 10 de los dos números mediante la ecuación 1.7:

$$(75)_{10} = (0, 75)_{10cns}$$
$$-(21)_{10} = (9, 79)_{10cns}$$

Después realizamos el cálculo como $(75)_{10} + (-(21)_{10})$. Al sumar los dos código de 3 dígitos, obtenemos

$$\begin{array}{rcccc} & & 0 & 7 & 5 \\ & + & 9 & 7 & 9 \\ \hline & 1 & 0 & 5 & 4 \\ & \uparrow & & & \\ & \text{Acarreo} & & & \end{array}$$

Al descartar el dígito de acarreo, el resultado es $(0,54)_{10cns} = (54)_{10}$, que es el resultado correcto.

EJEMPLO 1.52

Sumar $+(21)_{10}$ y $-(75)_{10}$.

De nuevo, comenzamos por determinar los códigos de complemento a diez para los dos números, por medio de la ecuación 1.7:

$$(21)_{10} = (0, 21)_{10cns}$$
$$-(75)_{10} = (9, 25)_{10cns}$$

Al sumar los dos códigos de tres dígitos obtenemos

$$\begin{array}{rccc} & 0 & 2 & 1 \\ + & 9 & 2 & 5 \\ \hline & 9 & 4 & 6 \end{array}$$

El resultado es $(9,46)_{10cns}$, donde el 9 indica que este número representa un valor negativo. El lector debe verificar que $(9,46)_{10cns}$ es la representación correcta del resultado deseado, $-(54)_{10}$, en un sistema numérico de complemento a diez.

Sistemas numéricos con complemento disminuido a una base

El *complemento disminuido a una base* $[N]_{r-1}$ de un número $(N)_r$ se define como

$$[N]_{r-1} = r^n - (N)_r - 1 \qquad (1.10)$$

donde n es el número de dígitos de $(N)_r$.

El *complemento a uno* es un caso particular del complemento disminuido a una base para los números binarios ($r = 2$) y está dado por

$$[N]_{2-1} = 2^n - (N)_2 - 1 \qquad (1.11)$$

donde n es el número de bits de $(N)_2$.

Podemos determinar el complemento a uno de un número binario dado en forma directa a partir de la ecuación 1.11, como se muestra en los siguientes ejemplos. El lector debe verificar los datos del complemento a uno de la tabla 1.6 después de estudiar estos ejemplos.

EJEMPLO 1.53

Determinar el complemento a uno de $(01100101)_2$.

De la ecuación 1.11,

$$\begin{aligned}[N]_{2-1} &= 2^8 - (01100101)_2 - 1 \\ &= (100000000)_2 - (01100101)_2 - (00000001)_2 \\ &= (10011011)_2 - (00000001)_2 \\ &= (10011010)_2.\end{aligned}$$

EJEMPLO 1.54

Determinar el complemento a uno de $(11010100)_2$.

De la ecuación 1.11,

$$\begin{aligned}[N]_{2-1} &= 2^8 - (11010100)_2 - (00000001)_2 \\ &= (100000000)_2 - (11010100)_2 - (00000001)_2 \\ &= (00101100)_2 - (00000001)_2 \\ &= (00101011)_2.\end{aligned}$$

EJEMPLO 1.55

Determinar el complemento a nueve de $(40960)_{10}$.

De la ecuación 1.10,

$$\begin{aligned}[N]_{10-1} &= 10^5 - (40960)_{10} - (00001)_{10} \\ &= (100000)_{10} - (40960)_{10} - (00001)_{10} \\ &= (59040)_{10} - (00001)_{10} \\ &= (59039)_{10}.\end{aligned}$$

Aunque siempre podemos determinar el complemento a uno de un número mediante la definición dada en la ecuación 1.11, hay métodos más sencillos. Los ejemplos anteriores sugieren el siguiente algoritmo para el cálculo de $[N]_{r-1}$ dado $(N)_r$, y lo presentamos sin demostración.

Algoritmo 1.6 Determinar $[N]_{r-1}$ dado $(N)_r$.

Remplazamos cada dígito a_i de $(N)_r$ por $r - 1 - a_i$. Observe que si $r = 2$, basta con complementar cada bit individual de $(N)_r$.

Al comparar las ecuaciones 1.7 y 1.10, vemos que la relación entre el complemento a una base y el complemento disminuido a una base de un número $(N)r$ es la siguiente.

$$[N]_r = [N]_{r-1} + 1 \tag{1.12}$$

Ahora debe quedar claro que el algoritmo 1.5 para calcular el complemento a una base es consecuencia del algoritmo 1.6.

Podemos formular sistemas numéricos que utilicen el complemento disminuido a una base para representar los números negativos de una manera similar a la que usamos con el complemento a una base. Sin embargo, no haremos esto aquí; nos limitaremos a ilustrar la aritmética.

Aritmética de complemento disminuido a una base

En los siguientes ejemplos ilustramos las principales características de la aritmética de complemento disminuido a una base. Los tres primeros ejemplos se centran en la suma de complemento a uno con diversas combinaciones de operandos positivos y negativos. Los números utilizados en estos ejemplos se tomaron de la tabla 1.6.

EJEMPLO 1.56

Sumar $+(1001)_2$ y $-(0100)_2$.

Representamos el número positivo como 01001 y el negativo con el complemento a uno de 00100, que es 11011. Por tanto, 00100 + 11011 = 100100. Observe que éste *no* es el resultado correcto. Sin embargo, obtenemos el resultado correcto si el acarreo de salida del bit más significativo se suma a la posición de bit menos significativa; es decir, 00100 + 1 = 00101. Este procedimiento se conoce como *acarreo final circular* y es un paso de corrección necesario en la aritmética de complemento disminuido.

EJEMPLO 1.57

Sumar $+(1001)_2$ y $-(1111)_2$.

Representamos el número positivo como 01001 y el negativo como 10000. Esto produce 01001 + 10000 = 11001. Observe que en este caso el acarreo final circular es 0 y, por tanto, no afecta el resultado.

EJEMPLO 1.58

Sumar $-(1001)_2$ y $-(0011)_2$.

Si representamos ambos números con su complemento a uno, obtenemos 10110 + 11100 = 110010. El paso de acarreo final circular produce el resultado correcto, es decir, 10010 + 1 = 10011.

Los siguientes dos ejemplos ilustran la aritmética de complemento a nueve.

EJEMPLO 1.59

Sumar $+(75)_{10}$ y $-(21)_{10}$.

El complemento a nueve de 021 es 978. Por tanto, la operación es 075 + 978 = 1053, que es el resultado correcto después del procedimiento de acarreo final circular: 053 + 1 = 054.

Ejemplo 1.60

Sumar $+(21)_{10}$ y $-(75)_{10}$.

El cálculo es 021 + 924 = 945, que es el resultado correcto, pues el acarreo final circular es 0.

1.5 Códigos de computadora

Un *código* es un uso sistemático y de preferencia estandarizado de un conjunto dado de símbolos para representar información. En la vida cotidiana aparecen varias formas sencillas de códigos. Por ejemplo, al acercarnos a un semáforo, se sobreentiende que la luz roja significa alto, que la luz verde significa siga y que la señal ámbar significa precaución. En otras palabras, el código es

Luz roja:	Alto
Luz ámbar:	Precaución
Luz verde:	Siga

Otro código conocido es el que se usa en el beisbol. Cuando un *umpire* levanta sus brazos con dos dedos en la mano derecha y tres dedos en la mano izquierda, se sobreentiende que la cuenta del bateador es dos *strikes* y tres bolas. Estos dos sencillos ejemplos ilustran la idea de los códigos y sin duda que el lector podrá imaginar más ejemplos.

Las computadoras y otros sistemas digitales utilizan códigos más complejos para el procesamiento, almacenamiento e intercambio de información de diversos tipos. Tres tipos importantes de códigos para computadora son el numérico, el de caracteres, y el de detección y corrección de errores. A continuación analizaremos brevemente algunos códigos importantes de cada una de estas categorías.

1.5.1 Códigos numéricos

Los códigos numéricos sirven para representar números con fines de procesamiento, y/o de almacenamiento. Los números de punto fijo y de punto flotante son ejemplos de estos códigos.

Números de punto fijo

Los *números de punto fijo* se utilizan para representar ya sea enteros con signo o bien fracciones con signo. En ambos casos se usan los sistemas de magnitud y signo, de complemento a dos o de complemento a uno para representar los valores con signo. Los enteros de punto fijo tienen un punto binario implícito a la derecha del bit menos significativo, como se muestra en la figura 1.3a, y las fracciones de punto fijo tienen el punto binario implícito entre el bit de signo y el bit de magnitud más significativo, como se muestra en la figura 1.3b.

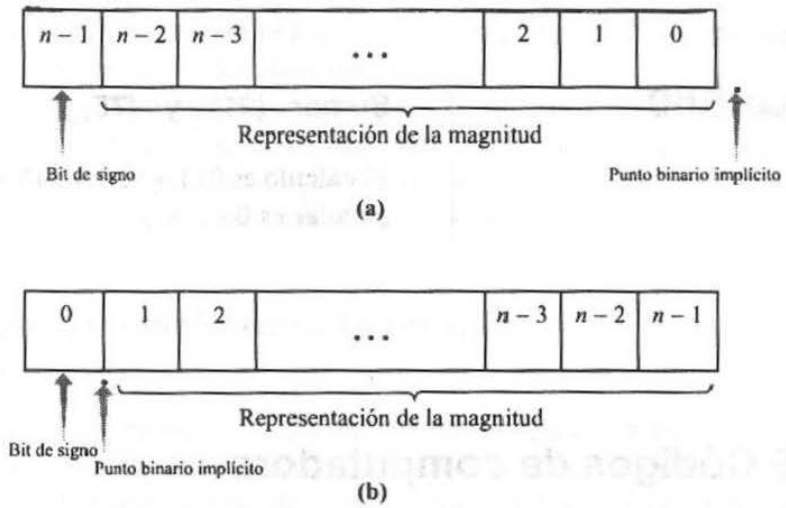


Figura 1.3 Representaciones de números de punto fijo. **(a)** Entero de punto fijo. **(b)** Fracción de punto fijo.

EJEMPLO 1.61

Dar dos posibles interpretaciones del número de punto fijo de 8 bits 01101010, usando el sistema de complemento a dos.

Como el bit de signo es 0, el número representa el entero positivo 1101010. si el punto binario se coloca como en la figura 1.3a, o bien la fracción positiva 0.1101010 si el punto binario se coloca como en la figura 1.3b.

EJEMPLO 1.62

Dar dos posibles interpretaciones del número de punto fijo de 8 bits 11101010, usando el sistema de complemento a dos.

El bit de signo es 1; por tanto, el número representa –0010110. o bien –0.0010110, según sea la convención utilizada para colocar el punto binario.

Representaciones con exceso o sesgadas

Una *representación con exceso –K* de un código C se forma sumando el valor K a cada palabra de código de C. Las representaciones con exceso se utilizan a menudo para representar los exponentes de los números de punto flotante, de modo que el valor mínimo del exponente se represente mediante ceros. Observe que los números con exceso -2^n no son sino los números de complemento a dos, ¡con el bit de signo invertido!

La representación con exceso 8 de la tabla 1.8 se obtiene al sumar $(1000)_2$ al código de 4 bits de complemento a dos. Observe que el resultado es que el número mínimo (–8) se representa con 0000, y el máximo (+7), con 1111.

TABLA 1.8 CÓDIGO CON EXCESO 8

Decimal	Complemento a dos	Exceso 8
+7	0111	1111
+6	0110	1110
+5	0101	1101
+4	0100	1100
+3	0011	1011
+2	0010	1010
+1	0001	1001
0	0000	1000
-1	1111	0111
-2	1110	0110
-3	1101	0101
-4	1100	0100
-5	1011	0011
-6	1010	0010
-7	1001	0001
-8	1000	0000

Números de punto flotante

Los números de punto flotante tienen una forma similar a los números escritos en notación científica. En general, la forma de punto flotante de un número N se escribe como

$$N = M \times r^E \tag{1.13}$$

donde M, la *mantisa* o *significando*, es un número de punto fijo que contiene los dígitos significativos de N y E, el *exponente* o *característica,* es un entero de punto fijo. En el caso general, dado un número de punto fijo N, donde

$$N = \pm(a_{n-1} \ldots a_0.a_{-1} \ldots a_{-m})_r$$

entonces, en forma de punto flotante,

$$N = \pm(.a_{n-1} \ldots a_{-m})_r \times r^n$$

Al deducir la representación de un número de punto flotante, la mantisa y la característica se codifican por separado. La base es implícita y no se incluye en la representación.

La mantisa M muchas veces se codifica con magnitud y signo, por lo general como una fracción, y se puede escribir como

$$M = (S_M.a_{n-1} \ldots a_{-m})_{rsm} \tag{1.14}$$

donde $(.a_{n-1} \ldots a_{-m})_r$ representa la magnitud de M y S_M indica el signo del número. S_M se elige por lo general de modo que

$$M = (-1)^{S_M} \times (.a_{n-1} \ldots a_{-m})_r \tag{1.15}$$

y entonces $S_M = 0$ indica un número positivo, mientras que $S_M = 1$ indica un número negativo.

Con frecuencia, el exponente E se codifica en *complemento a dos con exceso* $-K$. El complemento a dos con exceso $-K$ de un exponente se forma sumando un *sesgo* de K al valor entero en complemento a dos del exponente. En el caso de los números binarios de punto flotante (para los que la base es $r = 2$), K es, por lo general, 2^{e-1}, donde e es el número de bits del exponente. Por tanto,

$$\begin{aligned} -2^{e-1} \leq \quad E \quad &< 2^{e-1} \\ 0 \leq E + 2^{e-1} &< 2^e \end{aligned}$$

lo que indica que el valor sesgado de E es un número que varía entre 0 y $2^e - 1$ cuando E va de su valor más negativo a su valor más positivo. Podemos escribir la forma con exceso $-K$ de E como

$$E = (b_{e-1}, b_{e-2} \ldots b_0)\text{exceso}\text{–}K \tag{1.16}$$

donde b_{e-1} indica el signo de E.

Combinamos M y E, codificados mediante las ecuaciones 1.14 y 1.16, para producir el formato numérico de punto flotante:

$$N = (S_M b_{e-1} b_{e-2} \ldots b_0 a_{n-1} \ldots a_{-m})_r \tag{1.17}$$

lo que representa al número

$$N = (-1)^{S_M} \times (.a_{n-1} \ldots a_{-m})_r \times r^{(b_{e-1} b_{e-2} \ldots b_0)_2 - 2^{e-1}} \tag{1.18}$$

Una excepción al formato de la ecuación 1.17 es el número 0, que se considera como un caso particular y, por lo general, se representa mediante una palabra con ceros.

Las representaciones de punto flotante de un número dado no son únicas. Dado un número N, definido como en la ecuación 1.13, podemos ver que

$$N = M \times r^E \tag{1.19}$$

$$= (M \div r) \times r^{E+1} \tag{1.20}$$

$$= (M \times r) \times r^{E-1} \tag{1.21}$$

donde la división $(M \div r)$ se realiza recorriendo los dígitos de M una posición a la derecha, y $(M \times r)$ se realiza recorriendo los dígitos de M una posición a la izquierda. Por tanto, varias combinaciones de mantisa y exponente representan el mismo número. Por ejemplo, sea $M = +(1101.0101)_2$. La representación de M como una fracción con magnitud y signo en el formato de la ecuación 1.14 y la aplicación repetida de la ecuación 1.20 da como resultado

$$M = +(1101.0101)_2$$

$$= (0.11010101)_2 \times 2^4 \tag{1.22}$$

$$= (0.011010101)_2 \times 2^5 \tag{1.23}$$

$$= (0.0011010101)_2 \times 2^6 \tag{1.24}$$

$$\vdots$$

Al realizar cálculos en una computadora, por lo general es más conveniente tener una representación única para cada número. Se utiliza la *normalización* para conferir unicidad a los números de punto flotante. Un número de punto flotante está *normalizado* si el exponente se ajusta de modo que la mantisa tenga un valor distinto de cero en la posición de su dígito más significativo. Por tanto, la ecuación 1.22 es la representación normalizada de N, mientras que los números de las ecuaciones 1.23 y 1.24 no están normalizados.

Observe que el bit más significativo de un número binario normalizado siempre es 1. Por tanto, si representamos M en la forma de magnitud y signo como una fracción normalizada,

$$0.5 \leq |M| < 1.$$

Los formatos de punto flotante que utilizan en los sistemas de cómputo de los diversos fabricantes difieren con frecuencia en la cantidad de bits que se usan para representar la mantisa y el exponente, así como en el método de codificación de cada uno. Casi todos los sistemas utilizan el formato general ilustrado en la figura 1.4, con el signo almacenado en el bit extremo izquierdo, seguido por el exponente y después la mantisa. El formato de una palabra de la figura 1.4a se utiliza por lo general en las computadoras con longitud de palabra de 32 bits o más. El formato de dos palabras de la figura 1.4b se usa en computadoras con longitudes de palabra "cortas" para los números de punto flotante con precisión sencilla o en computadoras con longitudes de palabra largas para la representación con precisión extendida (también llamada doble precisión).

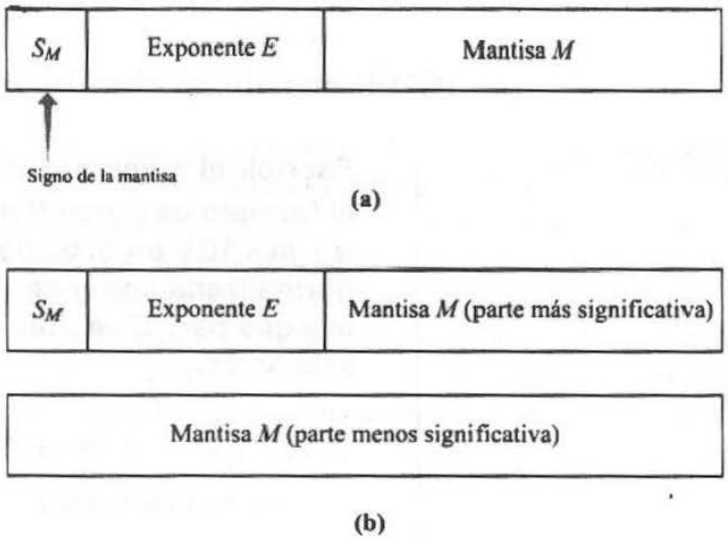


Figura 1.4 Formatos de número de punto flotante. **(a)** Formato típico de precisión simple. **(b)** Formato típico de precisión extendida.

La tabla 1.9 presenta un resumen de los formatos de precisión sencilla y doble utilizados en diversos sistemas de cómputo, incluidos los formatos definidos en el estándar 754-1985 [8] del IEEE. Observe que todos estos formatos emplean un exponente sesgado, con un número variable de bits. Los formatos de la DEC VAX y el estándar de IEEE suprimen el almacenamiento del bit más significativo de la mantisa. Como todos los números son valores binarios en forma normalizada, sabemos que el bit más significativo es 1. Por tanto, no es necesario guardar este bit y se gana un bit adicional para la precisión (denotado por +1 en la tercera columna de la tabla 1.9).

TABLA 1.9 FORMATOS COMUNES PARA NÚMEROS DE PUNTO FLOTANTE*

Sistema/Formato	Total de bits	Bits del significando	Bits del exponente	Sesgo del exponente	Codificación de la mantisa
Estándar 754-1985 del IEEE:					Magnitud/signo (base 2):
Precisión sencilla	32	23(+1)	8	127	$1 \le \lvert M \rvert < 2$
Doble precisión	64	52(+1)	11	1023	$1 \le \lvert M \rvert < 2$
IBM System/360:					Magnitud/signo (base 16):
Precisión sencilla	32	24	7	64	$1/16 \le \lvert M \rvert < 1$
Doble precisión	64	56	7	64	$1/16 \le \lvert M \rvert < 1$
DEC VAX 11/780:					Magnitud/signo (base 2):
Formato F	32	23(+1)	8	128	$1/2 \le \lvert M \rvert < 1$
Formato D	64	55(+1)	8	128	$1/2 \le \lvert M \rvert < 1$
Formato G	64	52(+1)	11	1024	$1/2 \le \lvert M \rvert < 1$
CDC Cyber 70:	60	48	11	1024	Complemento a uno (base 2) $1 \le \lvert M \rvert < 248$

* (+1)⇒ indica que se ha suprimido el bit más significativo.

EJEMPLO 1.63

Escribir el número binario $N = (101101.101)_2$ en el formato de punto flotante de la ecuación 1.17, donde $n + m = 10$ y $e = 5$. Suponga que se utiliza una fracción normalizada con magnitud y signo para representar M y que para E se utiliza el complemento a dos con exceso 16.

$$N = (101101.101)_2 = (0.101101101)_2 \times 2^6$$

Escribimos la mantisa en el formato de la ecuación 1.14:

$$\begin{aligned} M &= +(0.1011011010)_2 \\ &= (0.1011011010)_{2sm} \end{aligned}$$

Codificamos el exponente determinando su forma en complemento a dos y sumando después un sesgo de 16. (Observe que el número de bits del exponente es $e = 5$ y que el valor del sesgo es $2^{e-1} = 2^4 = 16$.) Por tanto,

$$\begin{aligned} E &= +(6)_{10} \\ &= +(0110)_2 \\ &= (00110)_{2cns} \end{aligned}$$

Al sumar el valor de sesgo $16 = (10000)_2$ al complemento a dos de E obtenemos

$$\begin{array}{rr} & 00110 \\ + & 10000 \\ \hline & 10110 \end{array}$$

de modo que

$$E = (1{,}0110)_{\text{exceso-16}}$$

Observe que el signo del exponente, b_{e-1}, es 1, lo cual indica un valor positivo del exponente.

Al unir M y E obtenemos

$$N = (0,\ 1,\ 0110,\ 1011011010)_{fp}$$

Las operaciones aritméticas con números de punto flotante requieren algoritmos especiales para la manipulación de exponentes y mantisas, mismas que rebasan el objetivo de este texto. El lector puede consultar [9] si desea más información acerca de los algoritmos para la aritmética de punto flotante.

1.5.2 Códigos de caracteres y otros códigos

Con frecuencia, es necesario o deseable representar la información como cadenas de caracteres alfabéticos o numéricos. Se han desarrollado muchos códigos de caracteres para esto; ahora analizaremos los más importantes.

Decimal codificado en binario (BCD)

El código *decimal codificado en binario (BCD)* sirve para representar los dígitos decimales del 0 al 9 y es un ejemplo de código *ponderado*, es decir, cada posición de bit en el código tiene un valor o peso numérico fijo asociado a ella. El dígito representado mediante una palabra de código dada se determina sumando los bits ponderados. El código BCD utiliza 4 bits, y los pesos son los mismos que en un entero binario de 4 bits. Por tanto, el código BCD para un dígito decimal dado es igual al equivalente binario del número, con ceros de relleno. Los códigos BCD se conocen también como códigos 8–4–2–1, por los pesos utilizados. El código BCD completo aparece en la tabla 1.10.

TABLA 1.10 CÓDIGO DECIMAL CODIFICADO EN BINARIO (BCD)

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

Los códigos BCD se utilizan para codificar números que se envían a pantallas numéricas y para representar números en procesadores que realizan aritmética decimal. Esto último se realiza en las computadoras *mainframe* en un extremo del espectro, y en las calculadores de bolsillo en el otro.

EJEMPLO 1.64

Codificar el número decimal $N = (9750)_{10}$ en BCD.

En primer lugar, codificamos los dígitos individuales de la tabla 1.10.

$$9 \rightarrow 1001,\ 7 \rightarrow 0111,\ 5 \rightarrow 0101 \text{ y } 0 \rightarrow 0000$$

Después se concatenan los códigos individuales para obtener

$$N = (1001011101010000)_{\text{BCD}}$$

Se han inventado extensiones del código BCD para abarcar no sólo los dígitos decimales sino también los caracteres alfabéticos y otros caracteres que se pueden imprimir, así como caracteres de control que no se imprimen. Estos códigos tienen por lo general una longitud de 6 a 8 bits, y se utilizan para representar datos durante la entrada o salida y para la representación interna de datos no numéricos, como el texto. Uno de estos códigos, utilizado en muchos modelos de computadora *mainframe* de IBM, es el *código de intercambio extendido de decimales codificados en binario* (*EBCDIC*).

ASCII

El código de caracteres más utilizado en las aplicaciones de cómputo es el código *ASCII* (siglas en inglés de Código Estándar Americano para Intercambio de Información), que se pronuncia "aski". El código ASCII de 7 bits está dado en la tabla 1.11. Con frecuencia se utiliza un octavo bit para disponer de la detección de errores. Analizaremos esta técnica, codificación con paridad, en una sección posterior de este capítulo.

EJEMPLO 1.65

Codificar la palabra *Digital* en código ASCII, representando cada caracter con dos dígitos hexadecimales.

Caracter	Código binario	Código hexadecimal
D	1000100	44
i	1101001	69
g	1100111	67
i	1101001	69
t	1110100	74
a	1100001	61
l	1101100	6*C*

Observe que la forma hexadecimal es más compacta y legible que la binaria. Por esta razón, la primera se utiliza con frecuencia para representar información codificada en ASCII.

Códigos Gray

Un *código cíclico* se puede definir como cualquier código en el que, para cualquier palabra de código, un corrimiento circular produce otra palabra del

TABLA 1.11 CÓDIGO DE CARACTERES ASCII

$c_3c_2c_1c_0$	$c_6c_5c_4$ 000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	;	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	‘	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	*S0*	RS	.	>	N	^	n	~
1111	*S1*	US	/	?	O	_	o	DEL

código. El *código Gray* es uno de los tipos más comunes de códigos cíclicos y tiene la característica de que las palabras de código para dos números consecutivos difieren sólo en un bit. Es decir, la distancia entre las dos palabras de código es 1. En general, la *distancia* entre dos palabras de código binario es igual al número de bits en que difieren las dos palabras.

EJEMPLO 1.66

Definir un código Gray para codificar los números decimales de 0 a 15.

Se necesitan cuatro bits para representar todos los números, y podemos construir el código necesario asignando al bit i de la palabra de código el valor 0 si los bits i e $i + 1$ del número binario correspondiente son iguales, y 1 en caso contrario. El bit más significativo del número siempre se debe comparar con 0 al utilizar esta técnica. El código resultante aparece en la tabla 1.12.

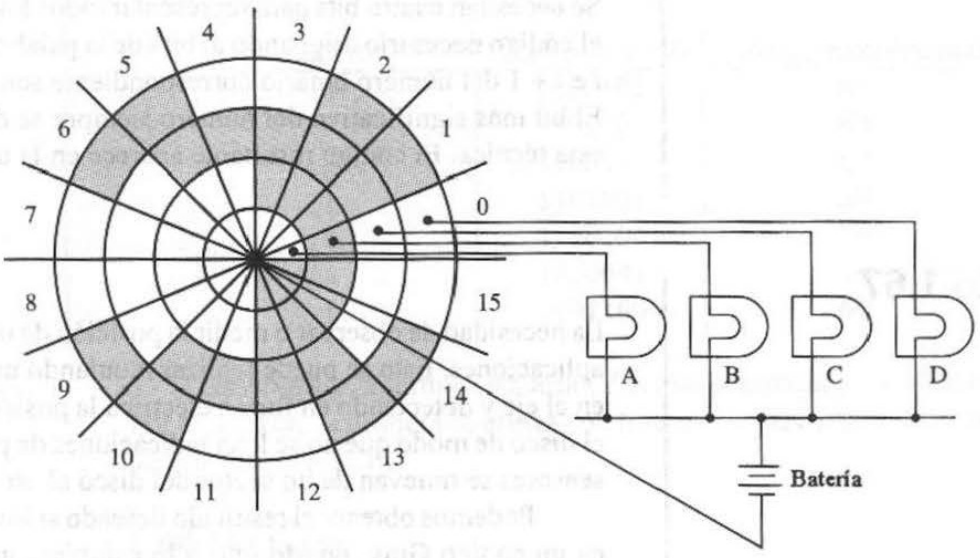
EJEMPLO 1.67

La necesidad de observar o medir la posición de un eje circular ocurre en muchas aplicaciones. Esto se puede realizar montando un círculo conductor codificado en el eje y detectando en forma eléctrica la posición del disco. ¿Cómo codificar el disco de modo que no se lean indicaciones de posición incorrectas cuando los sensores se muevan de un sector del disco al otro?

Podemos obtener el resultado deseado si los sectores del disco se codifican *en un código Gray, puesto que sólo cambiará un bit de posición en el código si los sensores se mueven de un sector al siguiente. La figura 1.5 ilustra la* solución.

TABLA 1.12 CÓDIGO GRAY PARA LOS NÚMEROS DECIMALES DE 0 A 15

Decimal	Binario	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000



1.5.3 Códigos para detección y corrección de errores

Un *error* en un dato binario se define como un valor incorrecto en uno o más bits. Un *error simple* es un valor incorrecto en un solo bit, mientras que un *error múltiple* se refiere a la existencia de uno o más bits incorrectos. Los errores pueden deberse a fallas del hardware, interferencia externa (ruido) u otros eventos no deseados. La información se puede codificar mediante códigos especiales que permitan la detección e incluso la corrección de ciertas clases de errores. A continuación ilustramos algunos códigos sencillos de detección y corrección de errores.

Será útil establecer algunas definiciones y notaciones antes de presentar códigos específicos. Sean I y J palabras de información binaria de n bits. El *peso* de I, $w(I)$, se define como el número de bits de I iguales a 1. La *distancia* entre I y J, $d(I, J)$, es igual al número de posiciones de bit en que difieren I y J.

EJEMPLO 1.68

Determinar los pesos de I y J y la distancia entre ellos si I = (01101100) y J = (11000100).

Si contamos los bits 1 en cada número, tenemos que

$$w(I) = 4 \text{ y } w(J) = 3$$

A continuación, comparamos los dos números bit por bit, observando que difieren de la manera siguiente:

```
0 1 1 0 1 1 0 0
1 1 0 0 0 1 0 0
  ↑   ↑   ↑
```

Los números difieren en posiciones de tres bits. Por tanto,

$$d(I, J) = 3$$

Propiedades generales de los códigos de detección y corrección de errores

Si la distancia entre dos palabras de código de un código C es mayor o igual que d_{min}, se dice que el código tiene *distancia mínima* d_{min}. Las propiedades de detección y corrección de errores de un código quedan determinadas en parte por su distancia mínima. Esto se ilustra en la figura 1.6, donde los puntos encerrados en un círculo representan palabras de código válidas y los no encerrados representan palabras que tienen errores. Unimos dos puntos si las palabras correspondientes difieren en exactamente una posición. Para un d_{min} dado, al menos se necesitan d_{min} errores para transformar una palabra de código válida en otra. Si hay menos de d_{min} errores, entonces se obtiene una palabra que no es del código y se puede detectar. Si la palabra que no es del código está más "cerca" de un código válido que de otro, se puede deducir la palabra de código original, y así corregirse el error.

En general, un código permite *corregir t errores* y detectar s errores adicionales si y sólo si se cumple la siguiente desigualdad.

$$2t + s + 1 \leq d_{min} \tag{1.25}$$

Un análisis de la ecuación 1.25 muestra que un código con detección de errores simples ($s = 1, t = 0$) requiere una distancia mínima de 2; un código con corrección de errores simples ($s = 0, t = 1$) requiere una distancia mínima de 3, y un código con corrección de errores simples y detección de errores dobles ($s = t = 1$) requiere una distancia mínima de 4. La figura 1.6 ilustra estas y otras combinaciones.

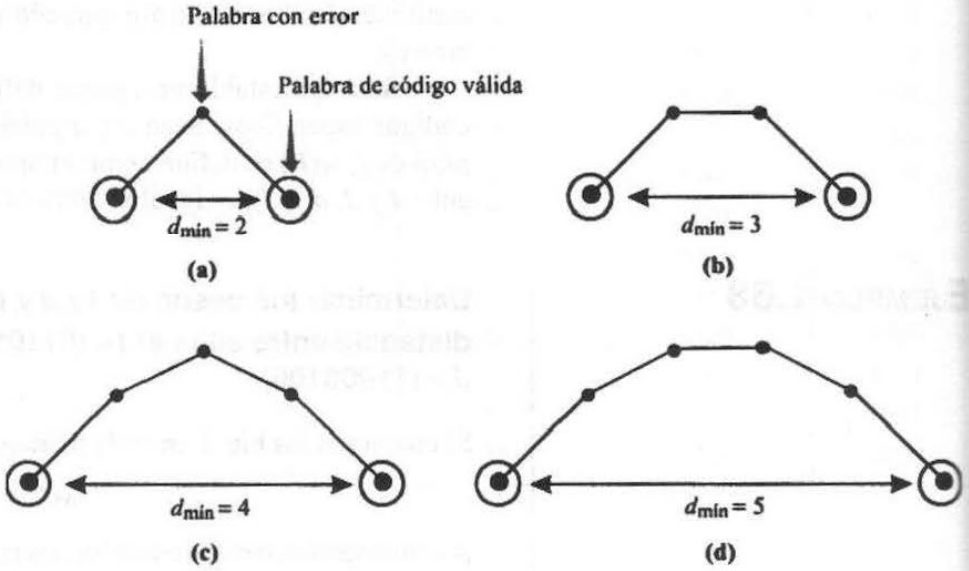


Figura 1.6 Relación entre la distancia mínima entre palabras de código y la capacidad para detectar y corregir errores de bit. (Las palabras unidas difieren exactamente en una posición de bit.) **(a)** Detección de errores simples (SED). **(b)** Corrección de errores simples (SEC) o corrección de errores dobles (DED). **(c)** (SEC y DED) o TED. **(d)** DEC (SEC y 3ED), o 4ED.

Códigos de paridad sencillos

Los códigos de paridad se forman a partir de un código C, concatenando (|) un *bit de paridad, P,* con cada palabra de código de C. La figura 1.7 ilustra el concepto. En un *código de paridad impar*, el bit de paridad se especifica como 0 o 1 de modo que $w(P \mid C)$ sea impar. El bit de paridad de un *código de paridad par* se selecciona de modo que $w(P \mid C)$ sea par. La figura 1.8 muestra la forma en que se utiliza la codificación con paridad en una cinta magnética de nueve pistas.

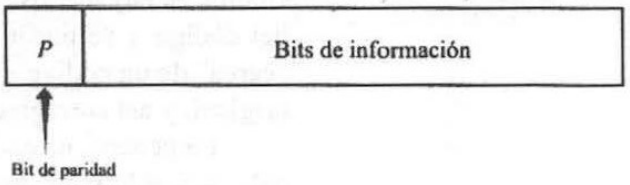


Figura 1.7 Información codificada con paridad.

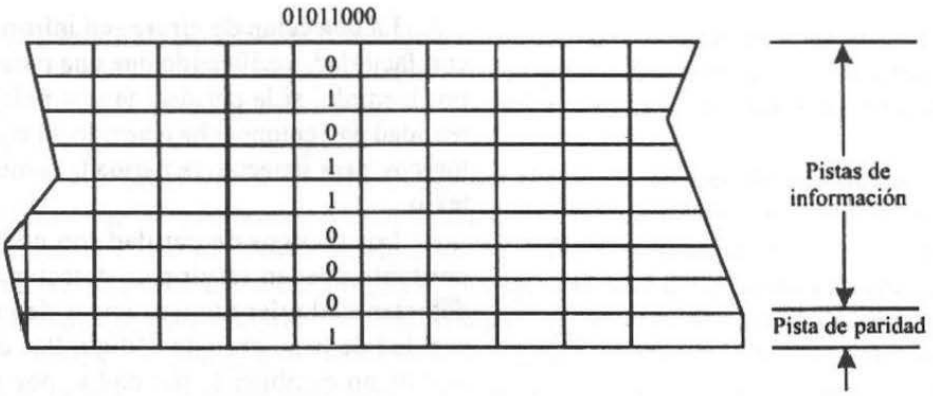


Figura 1.8 Codificación con paridad en cinta mágnetica.

EJEMPLO 1.69

Concatenar un bit de paridad con el código ASCII de los caracteres 0, *X*, = y BEL para obtener un código con paridad impar.

Carácter	Código ASCII	Código con paridad impar
0	0110000	10110000
X	1011000	01011000
=	0111100	10111100
BEL	0000111	00000111

EJEMPLO 1.70

Codificar el mensaje *CATCH 22* en código ASCII con paridad par y agrupar la palabra codificada en segmentos de 16 bits.

Segmento 1: $(\underbrace{11000011}_{C}\underbrace{01000001}_{A})_{\text{ASCII}}$

Segmento 2: $(\underbrace{11010100}_{T}\underbrace{11000011}_{C})_{\text{ASCII}}$

Segmento 3: $(\underbrace{01001000}_{H}\underbrace{10100000}_{\textit{Blanco}})_{\text{ASCII}}$

Segmento 4: $(\underbrace{10110010}_{2}\underbrace{10110010}_{2})_{\text{ASCII}}$

Observe que este mensaje se puede guardar en cuatro palabras de memoria de una computadora de 16 bits como

Palabra X	1100001101000001
Palabra $X + 1$:	1101010011000011
Palabra $X + 2$:	0100100010100000
Palabra $X + 3$:	1011001010110010

La detección de errores en información codificada con paridad se realiza con facilidad, verificando que una palabra de código tenga la paridad correcta. Por ejemplo, si la paridad de una palabra de código con paridad impar es en realidad par, entonces ha ocurrido un error detectable. Es fácil construir circuitos lógicos para detectar la paridad, como veremos en secciones posteriores del texto.

Los códigos de paridad son códigos con distancia mínima igual a 2 y por tanto pueden servir para detectar errores simples. De hecho, sirven para detectar cualquier número impar de errores, pues tales errores cambiarán la paridad de la palabra de código. Por otro lado, los errores en un número par de bits no cambian la paridad y, por tanto, no se pueden detectar mediante un código de paridad.

Código dos de cinco

El *código dos de cinco* es un código para detección de errores que tiene exactamente 2 bits iguales a 1 y 3 bits iguales a 0 en cada palabra de código, y es representativo de los códigos *m* de *n*. La detección de errores se realiza contando el número de unos de una palabra de código. Se nota un error cuando el número de unos no es exactamente igual a 2. Esto implica que los códigos dos de cinco permiten la detección de errores simples y también de errores múltiples en bits adyacentes. La tabla 1.13 presenta un código dos de cinco para los dígitos decimales.

Códigos de Hamming

En 1950, Richard Hamming publicó la descripción de una clase de códigos para corrección de errores que han tenido amplio uso. Los *códigos de Hamming* se pueden ver como una extensión de los códigos de paridad simple, en el sentido de que se utilizan varios *bits* de paridad o *bits de verificación*. Cada bit de verificación se define sobre un subconjunto de los bits de información de una palabra. Los subconjuntos se traslapan de modo que cada bit de información está en al menos dos subconjuntos. Los códigos para *corrección de errores*

TABLA 1.13 CÓDIGOS DOS DE CINCO PARA LOS DÍGITOS DECIMALES

Dígito	Código dos de cinco
0	00011
1	00101
2	01001
3	10001
4	00110
5	01010
6	10010
7	01100
8	10100
9	11000

simples (SEC) permiten la detección y corrección de cualquier error en un único bit. Los códigos para *corrección de errores simples y detección de errores dobles (SEC/DED)* permiten detectar pero no corregir errores dobles, además de detectar y corregir los errores simples.

Las propiedades de detección y corrección de errores de un código de Hamming están determinadas por el número de bits de verificación utilizados y la forma como los bits de verificación se definen en relación con los bits de información. La distancia mínima d_{min} es igual al peso de la palabra de código no nula con peso mínimo. En otras palabras, d_{min} es igual al número de unos de la palabra de código con menos unos. No es un objetivo de este libro analizar con detalle el diseño de los códigos de Hamming. Sin embargo, utilizaremos los dos códigos de Hamming que aparecen en la tabla 1.14 para ilustrar las propiedades de estos códigos. Además, presentaremos un método para el diseño de códigos de Hamming SEC sencillos.

Código de Hamming 1. El código permite corregir errores simples pero no detectar errores dobles, pues su distancia mínima es 3. Esto se puede ver con claridad en el siguiente análisis. Un error simple en el bit extremo izquierdo de la palabra de código 0100110 produce la palabra errónea 1100110. La tabla 1.15 muestra la diferencia y la distancia entre cada palabra de código válida y la palabra errónea.

Observe que sólo la palabra de código donde ha ocurrido el error tiene una distancia 1 con respecto a la palabra errónea. Esto significa que ningún error simple en cualquier otra palabra de código podría haber producido la palabra

TABLA 1.14 DOS CÓDIGOS DE HAMMING PARA PALABRAS DE INFORMACIÓN DE CUATRO BITS

Palabras de información $(i_3 i_2 i_1 i_0)$	Código de Hamming 1 $(i_3 i_2 i_1 i_0 c_2 c_1 c_0)$	Código de Hamming 2 $(i_3 i_2 i_1 i_0 c_3 c_2 c_1 c_0)$
0000	0000000	00000000
0001	0001011	00011011
0010	0010101	00101101
0011	0011110	00110110
0100	0100110	01001110
0101	0101101	01010101
0110	0110011	01100011
0111	0111000	01111000
1000	1000111	10000111
1001	1001100	10011100
1010	1010010	10101010
1011	1011001	10110001
1100	1100001	11001001
1101	1101010	11010010
1110	1110100	11100100
1111	1111111	11111111

errónea. Por tanto, la detección de la palabra errónea 1100110 equivale a corregir el error, pues el único error simple posible que puede producir el patrón es un error en el bit extremo izquierdo de la palabra de código 0100110.

El análisis anterior también sugiere un procedimiento de detección y corrección de errores. Es decir, podríamos determinar la diferencia entre una palabra de datos y cada palabra de código válida posible. Una distancia de 0 indicaría una concordancia válida, una distancia de 1 indicaría un error simple en la palabra de código correspondiente, en la posición que corresponde al bit 1 de la diferencia, y una distancia mayor o igual que 2 respecto a todas las palabras de código indicaría un error múltiple. Aunque este procedimiento funciona en teoría, no sería práctico para códigos con un gran número de palabras de código. Más adelante analizaremos algunos métodos prácticos.

Nuestro análisis también revela que varias palabras de código tienen una distancia 2 con respecto a la palabra errónea. Por tanto, un error doble en cualquiera de estas palabras produciría la misma palabra errónea que el error simple (examine la figura 1.6). Esto implica que, en general, los errores dobles no se pueden detectar con este código. La corrección de errores simples aunada a la detección de errores dobles requiere un código con una distancia mínima de 4.

Los bits de verificación del código se definen de modo que den una paridad par con un subconjunto de los bits de información, como sigue:

$$c_2: \quad i_3, \quad i_2, \quad i_1$$
$$c_1: \quad i_3, \quad i_2, \quad i_0$$
$$c_0: \quad i_3, \quad i_1, \quad i_0$$

TABLA 1.15 EFECTOS DE LOS ERRORES SOBRE LAS PALABRAS DE CÓDIGO

Palabras de código	Palabra errónea	Diferencia	Distancia
0000000	1100110	1100110	4
0001011	1100110	1101101	5
0010101	1100110	1110011	5
0011110	1100110	1111000	4
0100110	1100110	1000000	1
0101101	1100110	1001011	4
0110011	1100110	1010101	4
0111000	1100110	1011110	5
1000111	1100110	0100001	2
1001100	1100110	0101010	3
1010010	1100110	0110100	3
1011001	1100110	0111111	6
1100001	1100110	0000111	3
1101010	1100110	0001100	2
1110100	1100110	0010010	2
1111111	1100110	0011001	3

Esta relación se puede especificar de manera conveniente mediante una matriz conocida como *matriz generatriz*, o matriz G, como se muestra a continuación. Cada columna de la matriz G corresponde a un bit de la palabra de código, según se indica.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & p_{11} & p_{12} & p_{13} \\ 0 & 1 & 0 & 0 & p_{21} & p_{22} & p_{23} \\ 0 & 0 & 1 & 0 & p_{31} & p_{32} & p_{33} \\ 0 & 0 & 0 & 1 & p_{41} & p_{42} & p_{43} \end{bmatrix} \quad (1.26)$$

$$i_3 \; i_2 \; i_1 \; i_0 \; c_2 \; c_1 \; c_0$$

La codificación de una palabra de información i para obtener una palabra de código c se puede expresar en términos de la matriz generatriz G como sigue.

$$c = iG \quad (1.27)$$

La decodificación de una palabra de código se puede expresar de manera conveniente en términos de una matriz H conocida como *matriz de verificación de paridad*. La matriz H se puede deducir de la matriz G como sigue, para el código anterior.

$$H = \begin{bmatrix} p_{11} & p_{21} & p_{31} & p_{41} & 1 & 0 & 0 \\ p_{12} & p_{22} & p_{32} & p_{42} & 0 & 1 & 0 \\ p_{13} & p_{23} & p_{33} & p_{43} & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \quad (1.28)$$

Una n–tupla c es una palabra de código generada por G si y sólo si

$$Hc^T = 0 \quad (1.29)$$

Sea d una palabra de datos correspondiente a una palabra de código c, que ha sido corrompida por un patrón de error e. Entonces

$$d = c + e \quad (1.30)$$

La decodificación comienza con el cálculo del *síndrome* s de d para determinar si existe un error. Si no hay error, la decodificación concluye al eliminarse los bits de verificación, dejando sólo los bits de información originales. Si se encuentra un error corregible, se corrige antes de eliminar los bits de verificación. Si se encuentra un error no corregible, el proceso termina con una señal de error que indica la situación.

El síndrome de d se calcula como sigue, utilizando H:

$$\begin{aligned} s &= Hd^T && (1.31) \\ &= H(c+e)^T \\ &= Hc^T + He^T \\ &= 0 + He^T \\ &= He^T && (1.32) \end{aligned}$$

Los síndromes para la matriz H dada en la ecuación 1.28 aparecen en la tabla 1.16. Observe que el patrón de cada síndrome es igual al patrón de la columna de la matriz H que corresponde al bit erróneo.

Código de Hamming 2. La distancia mínima es 4, ya que ninguna palabra de código no nula tiene un peso menor que 4. Por tanto, el código tiene las

TABLA 1.16 SÍNDROMES Y PATRONES DE ERROR

Patrón de error	Síndrome	Significado
0 0 0 0 0 0 0	0 0 0	Sin error
0 0 0 0 0 0 1	0 0 1	Error en c_0
0 0 0 0 0 1 0	0 1 0	Error en c_1
0 0 0 0 1 0 0	1 0 0	Error en c_2
0 0 0 1 0 0 0	0 1 1	Error en i_0
0 0 1 0 0 0 0	1 0 1	Error en i_1
0 1 0 0 0 0 0	1 1 0	Error en i_2
1 0 0 0 0 0 0	1 1 1	Error en i_3

propiedades de corrección de errores simples y detección de errores dobles. Las matrices generatriz y de verificación de paridad son las siguientes:

$$G = \begin{bmatrix} 10000111 \\ 01001110 \\ 00101101 \\ 00011011 \end{bmatrix} \tag{1.33}$$

$$H = \begin{bmatrix} 01111000 \\ 11100100 \\ 11010010 \\ 10110001 \end{bmatrix} \tag{1.34}$$

Observe que cada columna de la matriz H de la ecuación 1.34 tiene un número impar de unos. Tales códigos de Hamming se llaman códigos con *columnas de peso impar* y tienen varias propiedades deseables, incluidas la corrección de errores simples, la detección de errores dobles y la detección de otros errores múltiples. Además, permiten el uso de circuitos de codificación y decodificación relativamente económicos y rápidos. Es por esto que, los códigos con columnas de peso impar se utilizan con frecuencia en la práctica.

La forma más fácil de diseñar un código de Hamming es especificando la matriz H. Para cualquier entero positivo $m \geq 3$, existe un código SEC (m,k) con las siguientes propiedades:

- Longitud de código: $n = 2^m - 1$
- Número de bits de información: $k = 2^m - m - 1$
- Número de bits de verificación: $n - k = m$
- Distancia mínima: $d_{min} = 3$

La matriz H para tal código es una matriz $n \times m$ formada por todas las m–tuplas binarias distintas de cero como columnas. La matriz de la ecuación 1.28 es un ejemplo de tales matrices, con $m = 3$. Observe que podemos determinar las demás matrices H para $m = 3$ reordenando las columnas.

Obtenemos un código de Hamming (15,11) cuando $m = 4$. Una posible matriz H para tal código es la siguiente:

$$H = \begin{bmatrix} 111101110001000 \\ 111011001100100 \\ 110110100110010 \\ 101110011010001 \end{bmatrix} \tag{1.35}$$

Podemos eliminar l columnas arbitrarias de una matriz H de un código de Hamming para obtener otro código de Hamming con las siguientes propiedades:

- Longitud de código: $n = 2^m - l - 1$
- Número de bits de información: $k = 2^m - m - l - 1$
- Número de bits de verificación: $n - k = m$
- Distancia mínima: $d_{min} \geq 3$

Estas propiedades conducen a la posibilidad de diseñar códigos con mejores propiedades de detección y corrección de errores y longitudes de código más útiles.

EJEMPLO 1.71

Diseñar un código de Hamming para codificar cinco bits de información ($k = 5$).

Se necesitan cuatro bits de verificación ($m = 4$), ya que para $m = 3$, $k = 2^3 - 3 - 1 = 4 < 5$. Sin embargo, para $m = 4$, $k = 2^4 - 4 - 1 = 11 > 5$. Pero podemos obtener un código (9,5) eliminando seis columnas de la matriz H de un código (15,11). Al eliminar seis columnas de la ecuación 1.35 obtenemos

$$H = \begin{bmatrix} 111101000 \\ 111010100 \\ 110110010 \\ 101110001 \end{bmatrix} \tag{1.36}$$

La matriz generatriz correspondiente es

$$G = \begin{bmatrix} 100001111 \\ 010001110 \\ 001001101 \\ 000101011 \\ 000010111 \end{bmatrix} \tag{1.37}$$

Esto concluye nuestro estudio de los códigos de detección y corrección de errores. Los lectores que deseen aprender más acerca de estos códigos pueden consultar la referencia bibliográfica [4].

1.6 Resumen

Hemos completado nuestra introducción a los sistemas numéricos y los códigos de computadoras. El lector debe estar ahora familiarizado con los sistemas numéricos decimal, binario, octal y hexadecimal y podrá convertir los números de cualquiera de estas bases a cualquier otra. Además, el lector deberá ya entender las operaciones aritméticas en todas las bases y la forma en que pueden

representarse los números negativos en las computadoras. También, deberá haber adquirido cierta familiaridad con los números de punto fijo y de punto flotante, y comprenderá en general los códigos de caracteres decimal codificado en binario (BCD) y ASCII. Así mismo, presentamos los códigos Gray y de exceso o sesgados. Por último, el lector deberá haber adquirido un conocimiento general de los códigos de detección y corrección de errores simples. Puede lograr una comprensión más detallada de estos temas consultando la bibliografía.

REFERENCIAS BIBLIOGRÁFICAS

1. M. Y. Hsiao, "A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes, "*IBM Journal of Research and Development*, Vol. 14, No. 4, págs. 395-401, julio 1970.
2. K. Hwang, *Computer Arithmetic*. Nueva York: Wiley, 1979.
3. D.E.Knuth, *Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1969.
4. S. Lin y D.J. Costello, *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1983.
5. W.W. Peterson y E.J. Weldon, Jr., *Error-correcting Codes*, 2a. edición Cambridge, MA: MIT Press, 1972.
6. J.F. Wakerly, *Microcomputer Architecture and Programming*. Nueva York: Wiley, 1981.
7. S. Waser y M.J. Flynn, *Introduction to Arithmetic for Digital Systems*. Nueva York: Holt, Rinehart, and Winston, 1982.
8. *IEEE Standard for Binary Floating-point Arithmetic*, ANSI/IEEE Std. 754-1985. Institute of Electrical and Electronic Engineers, 345 East 47th St., Nueva York, NY, agosto 1985.
9. Israel, Koren, *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice Hall 1993.

PROBLEMAS

1.1 Calcule $A + B$, $A - B$, $A \times B$ y $A \div B$ para las siguientes parejas de números binarios.

(a) 10101, 1011
(b) 1011010, 101111
(c) 101, 1011
(d) 10110110, 01011011
(e) 1101011, 1010
(f) 1010101, 101010
(g) 10000, 1001
(h) 1011.0101, 110.11

1.2 Calcule $A + B$, $A - B$, $A \times B$ y $A \div B$ para las siguientes parejas de números octales.

(a) 372, 156
(b) 704, 230
(c) 1000, 777
(d) 423, 651

1.3 Calcule $A + B$, $A - B$, $A \times B$ y $A \div B$ para las siguientes parejas de números hexadecimales.

(a) 2CF3, 2B
(b) FFFF, 1000
(c) 9A5, D17
(d) 372, 156

1.4 Convierta los siguientes números decimales a números binarios, octales y hexadecimales.

(a) 27
(b) 915
(c) 0.375
(d) 0.65
(e) 174.25
(f) 250.8

1.5 Convierta los siguientes números binarios a números octales, hexadecimales y decimales utilizando el método de conversión más adecuado.

(a) 1101
(b) 101110
(c) 0.101
(d) 0.01101
(e) 10101.11
(f) 10110110.001

1.6 Convierta los siguientes números octales a números binarios, hexadecimales y decimales utilizando el método de conversión más adecuado.

(a) 65
(b) 371
(c) 240.51
(d) 2000
(e) 111111
(f) 177777

1.7 Convierta los siguientes números hexadecimales a números binarios, octales y decimales utilizando el método de conversión más adecuado.

(a) 4F
(b) ABC
(c) F8.A7
(d) 2000
(e) 201.4
(f) 3D65E

1.8 Determine el complemento a dos de los siguientes números binarios, suponiendo $n = 8$.

(a) 101010
(b) 1101011
(c) 0
(d) 11111111
(e) 10000000
(f) 11000

1.9 Determine el complemento a uno de los siguientes números binarios, suponiendo $n = 8$.

(a) 110101
(b) 1010011
(c) 0
(d) 10000000
(e) 100001
(f) 01111111

1.10 Calcule $A + B$, $A - B$, $-A + B$ y $-A - B$ para las siguientes parejas de números, suponiendo un sistema numérico de complemento a dos y $n = 8$. Verifique sus resultados mediante aritmética decimal. Explique los resultados extraños.

(a) 1010101, 1010
(b) 1101011, 0101010
(c) 11101010, 101111
(d) 10000000, 01111111

1.11 Repita el problema 1.10 para los siguientes números, utilizando un sistema numérico de complemento a uno.

(a) 10101,1, 1101
(b) 10111010, 11010
(c) 1010101, 0101010
(d) 10000000, 01111111

1.12 Muestre la forma en que una computadora de 16 bits con un sistema numérico de complemento a dos realizaría los siguientes cálculos.

(a) $(16850)_{10} + (2925)_{10} = (?)_{10}$

(b) $(16850)_{10} - (2925)_{10} = (?)_{10}$

(c) $(2925)_{10} - (16850)_{10} = (?)_{10}$

(d) $-(2925)_{10} - (16850)_{10} = (?)_{10}$

1.13 Codifique los siguientes números en los códigos BCD y exceso 3.

(a) 39

(b) 1950

(c) 94704

(d) 625

1.14 Codifique las siguientes cadenas en código ASCII. Represente las cadenas codificadas mediante números hexadecimales.

(a) 1980

(b) A = b + C

(c) COMPUTER ENGINEERING

(d) The End.

1.15 Defina un código de cuatro bits para la representación de dígitos decimales, con la propiedad de que las palabras de código para cualesquiera dos dígitos cuya diferencia sea uno difieran sólo en una posición de bit, y que esto también se cumpla para los dígitos 0 y 9.

1.16 ¿Cuántos errores de bit se pueden detectar en un código dos de cinco? ¿Cuántos errores, si acaso, se pueden corregir en un código dos de cinco? Demuestre sus respuestas en forma matemática.

1.17 Examine el disco con código Gray de la figura 1.5. Suponga que los indicadores señalan lo siguiente: *A* está desactivado, *B* está activado, *C* está activado y *D* está parpadeando. Localice la posición del disco según los números de sector.

1.18 Se grabarán los siguientes mensajes de ocho bits en la cinta magnética de nueve pistas de la figura 1.7. Determine el bit de paridad para que cada mensaje tenga paridad impar.

(a) P10111010

(b) P00111000

(c) P10011001

(d) P01011010

1.19 Sea 10111001 una palabra errónea obtenida mediante el código de Hamming 2. Determine la palabra de código correcta, calculando la diferencia y la distancia entre la palabra errónea y cada palabra de código válida.

1.20 Desarrolle una tabla de síndromes para el código de Hamming 2 que considere el caso libre de errores, exclusivamente errores simples y exclusivamente errores dobles. ¿Existe una caracterización sencilla de los síndromes de errores dobles? ¿Existen patrones de error con tres o más errores detectables por el código?

1.21 Utilice la tabla de síndromes desarrollada en el problema 1.20 para decodificar las siguientes palabras.

(a) 10010111

(b) 10011011

(c) 00111110

(d) 00000111

(e) 11101110

(f) 01011000

(g) 11100001

(h) 01101000

1.22 Desarrolle las matrices generatriz y de verificación de paridad para el código SEC de Hamming, codificando palabras de información con una longitud de 6 bits.

1.23 Codifique todas las palabras de información para un código definido por la siguiente matriz de verificación de paridad. Observe que los códigos con una matriz de verificación de paridad en la forma de la ecuación 1.28 se llaman códigos separables, puesto que los bits de información se pueden separar como bloque de los bits de verificación. El código que resulta de la siguiente matriz no es separable, pues los bits de información y los de verificación están mezclados.

$$H = \begin{bmatrix} 1111000 \\ 1100110 \\ 1010101 \end{bmatrix}$$

1.24 ¿Qué propiedades de detección y corrección de errores tiene el código definido en el problema 1.23? Elabore una tabla de síndromes para este código. Describa las características interesantes de los síndromes.

1.25 Describa las ventajas y desventajas de los códigos separables que tienen la forma representada por la matriz de la ecuación 1.28, en comparación con los códigos no separables de la forma representada por la matriz del problema 1.23.