

CAPÍTULO 9

Aritmética del computador

9.1. La unidad aritmético-lógica

9.2. Representación de enteros

Representación en signo y magnitud
Representación en complemento a dos
Conversión entre longitudes de bits diferentes
Representación en coma fija

9.3. Aritmética con enteros

Negación
Suma y resta
Multiplicación
División

9.4. Representación en coma flotante

Fundamentos
Estándar del IEEE para la representación binaria en coma flotante

9.5. Aritmética en coma flotante

Suma y resta
Multiplicación y división
Consideraciones sobre precisión
Estándar IEEE para la aritmética binaria en coma flotante

9.6. Lecturas y sitios web recomendados

Sitios web recomendados

9.7. Palabras clave, preguntas de repaso y problemas

Palabras clave
Preguntas de repaso
Problemas

PUNTOS CLAVE

- Los dos aspectos fundamentales de la aritmética del computador son la forma de representar los números (el formato binario) y los algoritmos utilizados para realizar las operaciones aritméticas básicas (suma, resta, multiplicación y división). Estas dos consideraciones se aplican tanto a la aritmética de enteros como a la de coma flotante.
- Las cantidades en coma flotante se expresan como un número (mantisa) multiplicado por una constante (base) elevada a una potencia entera (exponente). Los números en coma flotante pueden utilizarse para representar cantidades muy grandes y muy pequeñas.
- La mayoría de los procesadores implementan la norma o estándar IEEE 754 para la representación de números y aritmética en coma flotante. Esta norma define el formato de 32 bits así como el de 64 bits.

Comenzamos nuestro estudio del procesador con la unidad aritmético-lógica (ALU). Tras una breve introducción a la ALU, el capítulo se centra en el aspecto más complejo de la misma: la aritmética del computador. Las funciones lógicas que forman parte de la ALU se describen en el Capítulo 10, y la implementación de funciones lógicas y aritméticas sencillas mediante lógica digital se describen en el Apéndice B del libro.

La aritmética de un computador se realiza normalmente con dos tipos de números muy diferentes: enteros y en coma flotante. En ambos casos, la representación elegida es un aspecto de diseño crucial que trataremos en primer lugar, seguido de una discusión sobre las operaciones aritméticas.

Este capítulo incluye diversos ejemplos que se resaltan en el texto mediante recuadros sombreados.

9.1. LA UNIDAD ARITMÉTICO-LÓGICA

La ALU es la parte del computador que realiza realmente las operaciones aritméticas y lógicas con los datos. El resto de los elementos del computador (unidad de control, registros, memoria, E/S) están principalmente para suministrar datos a la ALU, a fin de que esta los procese y para recuperar los resultados. Con la ALU llegamos al estudio de lo que puede considerarse el núcleo o esencia del computador.

Una unidad aritmético-lógica, y en realidad todos los componentes electrónicos del computador, se basan en el uso de dispositivos lógicos digitales sencillos que pueden almacenar dígitos binarios y realizar operaciones lógicas booleanas elementales. El Apéndice B explora, para el lector interesado, la implementación de circuitos lógicos digitales.

La Figura 9.1 indica, en términos generales, cómo se interconecta la ALU con el resto del procesador. Los datos se presentan a la ALU en registros, y en registros se almacenan los resultados de las operaciones producidos por la ALU. Estos registros son posiciones de memoria temporal internas al



Figura 9.1. Entradas y salidas de la ALU.

procesador que están conectados a la ALU (véase por ejemplo la Figura 2.3). La ALU puede también activar indicadores (*flags*) como resultado de una operación.

Por ejemplo, un indicador de desbordamiento se pondrá a 1 si el resultado de una operación excede la longitud del registro en donde éste debe almacenarse. Los valores de los indicadores se almacenan también en otro registro dentro del procesador. La unidad de control proporciona las señales que gobiernan el funcionamiento de la ALU y la transferencia de datos dentro y fuera de la ALU.

9.2. REPRESENTACIÓN DE ENTEROS

En el sistema de numeración binaria¹, cualquier número puede representarse tan solo con los dígitos 1 y 0, el signo menos, y la **coma de la base** (que separa la parte entera de la decimal, el punto en los países anglosajones). Por ejemplo:

$$-1101,0101_2 = -13,3125_{10}$$

Sin embargo, para ser almacenados y procesados por un computador, no se tiene la posibilidad de disponer del signo y de la coma. Para representar los números solo pueden utilizarse dígitos 0 y 1. Si utilizáramos solo enteros no negativos, su representación sería inmediata.

Una palabra de ocho bits puede representar números desde 0 hasta 255, entre los que se encuentran:

00000000	=	0
00000001	=	1
00101001	=	41
10000000	=	128
11111111	=	255

¹ Para una revisión de los sistemas de numeración (decimal, binario, hexadecimal), consulte el Apéndice A.

En general, si una secuencia de n dígitos binarios $a_{n-1}a_{n-2}\dots a_1a_0$ es interpretada como un entero sin signo A, su valor es:

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

REPRESENTACIÓN EN SIGNO Y MAGNITUD

Existen varias convenciones alternativas para representar números enteros tanto positivos como negativos. Todas ellas implican tratar el bit más significativo (el más a la izquierda) de la palabra como un bit de signo: si dicho bit es 0 el número es positivo, y si es 1, el número es negativo.

La forma más sencilla de representación que emplea un bit de signo es la denominada representación signo-magnitud. En una palabra de n bits, los $n-1$ bits de la derecha representan la magnitud del entero. Por ejemplo:

$+18 = 00010010$
$-18 = 10010010$
(signo-magnitud)

El caso general puede expresarse como sigue:

Signo y magnitud

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{si } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{si } a_{n-1} = 1 \end{cases} \quad (9.1)$$

La representación signo-magnitud posee varias limitaciones. Una de ellas es que la suma y la resta requieren tener en cuenta tanto los signos de los números como sus magnitudes relativas para llevar a cabo la operación en cuestión. Esto debiera quedar claro con la discusión de la Sección 9.3. Otra limitación es que hay dos representaciones del número 0:

$+0_{10} = 00000000$
$-0_{10} = 10000000$
(signo-magnitud)

Esto es un inconveniente porque es algo más difícil comprobar el valor 0 (una operación frecuentemente en los computadores) que si hubiera una sola representación.

Debido a estas limitaciones, raramente se usa la representación en signo-magnitud para implementar en la ALU las operaciones con enteros. En su lugar, el esquema más común es la representación en complemento a dos.

REPRESENTACIÓN EN COMPLEMENTO A DOS

Al igual que la de signo-magnitud, la representación en complemento a dos utiliza el bit más significativo como bit de signo, facilitando la comprobación de si el entero es positivo o negativo. Difiere

Tabla 9.1. Características de la representación numérica y la aritmética en complemento a dos.

Rango	-2^{n-1} hasta $2^{n-1} - 1$
Número de representaciones del cero	Una
Negación	Realizar el complemento booleano de cada bit del correspondiente número positivo, y entonces sumar 1 al patrón de bits resultante visto como un entero sin signo.
Extensión de la longitud en bits	Añadir posiciones de bits a la izquierda rellenándolas con el valor del bit de signo original.
Regla de desbordamiento	Si se suman dos números con el mismo signo (ambos positivos o ambos negativos), solo se produce desbordamiento cuando el resultado tiene signo opuesto.
Regla de la sustracción	Para restar B de A , efectuar el complemento a dos de B y sumarlo a A .

de la representación signo-magnitud en la forma de interpretar los bits restantes. La Tabla 9.1 destaca las características clave de la representación y la aritmética en complemento a dos que serán elaboradas en esta sección y la siguiente.

La mayoría de los tratados sobre representación en complemento a dos se centran en las reglas para la obtención de los números negativos, sin pruebas formales de que el esquema utilizado «funcione». En su lugar, la presentación que hacemos de los números enteros en complemento a dos, en ésta y en la Sección 9.3, está basada en [DATT93], donde se sugiere que la representación en complemento a dos se entiende mejor definiéndola en términos de una suma ponderada de bits, como hicimos antes para las representaciones sin signo y en signo-magnitud. La ventaja de este tratamiento del tema está en que no queda duda alguna de si las reglas para las operaciones aritméticas con la notación en complemento a dos puedan no funcionar en algunos casos concretos.

Consideremos un entero de n bits, A , representado en complemento a dos. Si A es positivo, el bit de signo, a_{n-1} , es cero. Los restantes bits representan la magnitud del número de la misma forma que en la representación signo-magnitud; es decir:

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{para } A \geq 0$$

El número cero se identifica como positivo y tiene por tanto un bit de signo 0 y una magnitud de todo ceros. Podemos ver que el rango de los enteros positivos que pueden representarse es desde 0 (todos los bits de magnitud son 0) hasta $2^{n-1} - 1$ (todos los bits de magnitud a 1). Cualquier número mayor requeriría más bits.

Ahora, para un número negativo A ($A < 0$), el bit de signo, a_{n-1} , es 1. Los $n - 1$ bits restantes pueden tomar cualquiera de las 2^{n-1} combinaciones. Por lo tanto, el rango de los enteros negativos que pueden representarse es desde -1 hasta -2^{n-1} . Sería deseable asignar los bits de los enteros negativos de tal manera que su manipulación aritmética pueda efectuarse de una forma directa, similar a la de los enteros sin signo. En la representación sin signo, para calcular el valor de un entero a partir de su expresión en bits, el peso del bit más significativo es $+2^{n-1}$. Como veremos en la

Sección 9.3, para una representación con bit de signo resulta que las propiedades aritméticas deseadas se consiguen si el peso del bit más significativo es (-2^{n-1}) . Este es el convenio utilizado para la representación en complemento a dos, obteniéndose la siguiente expresión para los números negativos:

Complemento a dos

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \quad (9.2)$$

Para los enteros positivos $a_{n-1} = 0$, de forma que el término $-2^{n-1} a_{n-1} = 0$. Así pues, la ecuación (9.2) define la representación en complemento a dos, tanto para los números positivos como los negativos.

La Tabla 9.2 compara, para enteros de cuatro bits, las representaciones en signo-magnitud y en complemento a dos. Veremos que la representación en complemento a dos, aunque nos pueda resultar engorrosa, facilita las operaciones aritméticas más importantes, la suma y la resta. Por esta razón, es utilizada casi universalmente como representación de los enteros en los procesadores.

Tabla 9.2. Representaciones alternativas de los enteros de 4 bits.

Representación decimal	Representación signo-magnitud	Representación complemento a dos	Representación sesgada
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	—	—
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	—	1000	—

-128	64	32	16	8	4	2	1

(a) Caja de valores de complemento a dos, de ocho posiciones

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$$-128 - 128 + 2 + 1 = -125$$

(b) Conversión a decimal del número binario 10000011

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$$-120 = -128 + 8$$

(c) Conversión a binario del número decimal -120

Figura 9.2. Utilización de la caja de valores para convertir entre binario en complemento a dos y decimal.

Una ilustración útil de la naturaleza de la representación en complemento a dos es una «caja» de valores, en la que el valor más a la derecha en la caja es $1 (2^0)$, y cada posición consecutiva hacia la izquierda tiene un valor doble a sumar (si el correspondiente bit es 1), hasta la posición más a la izquierda, cuyo valor es a restar. Como se puede ver en la Figura 9.2a, el número en complemento a dos más negativo representable es -2^{n-1} ; si cualquiera de los bits distintos del de signo es 1, este añade una cantidad positiva al número. También, está claro que un número negativo debe tener un 1 en la posición más a la izquierda, y un número positivo tendrá un cero en dicha posición. Por tanto, el número positivo mayor es un 0 seguido de todo unos, que es igual a $2^{n-1} - 1$.

El resto de la Figura 9.2 ilustra el uso de la caja de valores para convertir de complemento a dos a decimal, y de decimal a complemento a dos.

CONVERSIÓN ENTRE LONGITUDES DE BITS DIFERENTES

A veces se desea tomar un entero de n bits y almacenarlo en m bits, siendo $m > n$. Esto se resuelve fácilmente en la notación signo-magnitud: simplemente trasladando el bit de signo hasta la nueva posición más a la izquierda y rellenando con ceros.

+18	=	00010010	(signo-magnitud, 8 bits)
+18	=	0000000000010010	(signo-magnitud, 16 bits)
-18	=	11101110	(signo-magnitud, 8 bits)
-18	=	1000000000010010	(signo-magnitud, 16 bits)

Este procedimiento no funciona con los enteros negativos en complemento a dos. Utilizando el mismo ejemplo:

+18	=	00010010	(complemento a dos, 8 bits)
+18	=	00000000000010010	(complemento a dos, 16 bits)
-18	=	11101110	(complemento a dos, 8 bits)
-32.658	=	1000000001101110	(complemento a dos, 16 bits)

La penúltima línea anterior puede comprobarse fácilmente mediante la caja de valores de la Figura 9.2. La última línea puede verificarse utilizando la Ecuación (9.2) o mediante una caja de valores de 16 bits.

En su lugar, la regla para los enteros en complemento a dos es trasladar el bit de signo a la nueva posición más a la izquierda y completar con copias del bit de signo. Para números positivos, llenar con ceros, y para negativos con unos. Así pues, se tiene:

-18	=	11101110	(complemento a dos, 8 bits)
-18	=	111111111101110	(complemento a dos, 16 bits)

Para ver que esta regla funciona, consideremos de nuevo una secuencia de n dígitos binarios $a_{n-1} a_{n-2} \dots a_1 a_0$ interpretada como entero A en complemento a dos, tal que su valor es:

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Si A es positivo, la regla claramente funciona. Ahora, supongamos que A es negativo y que queremos construir una representación de m bits, con $m > n$. Entonces

$$A = -2^{m-1}a_{n-1} + \sum_{i=0}^{m-2} 2^i a_i$$

Los dos valores deben ser iguales:

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow a_{m-2} = \dots = a_{n-2} = a_{n-1} = 1$$

Al pasar de la primera a la segunda ecuación, se requiere que los $n - 1$ bits menos significativos no cambien entre las dos representaciones. Entonces, llegamos a la ecuación final, que solo es cierta si todos los bits desde la posición $n - 1$ hasta la $m - 2$ son 1. En consecuencia la regla funciona.

REPRESENTACIÓN EN COMA FIJA

Finalmente, mencionamos que la representación tratada en esta sección se denomina a veces de coma fija. Esto es porque la coma de la base (coma binaria) está fija y se supone que a la derecha del bit menos significativo. El programador puede utilizar la misma representación para fracciones binarias escalando los números de manera que la coma binaria esté implícitamente en alguna otra posición.

9.3. ARITMÉTICA CON ENTEROS

Esta sección examina funciones aritméticas comunes con números enteros representados en complemento a dos.

NEGACIÓN

En la representación signo-magnitud, la regla para obtener el opuesto de un entero es sencilla: invertir el bit de signo. En la notación de complemento a dos, la negación de un entero puede realizarse siguiendo las reglas:

1. Obtener el complemento booleano de cada bit del entero (incluyendo el bit de signo). Es decir, cambiar cada 1 por 0, y cada 0 por 1.
2. Tratando el resultado como un entero binario sin signo, sumarle 1.

Este proceso en dos etapas se denomina **transformación a complemento a dos**, u obtención del complemento a dos de un entero. Por ejemplo:

$+18$	$=$	00010010	(complemento a dos)
complemento bit a bit	$=$	11101101	
		$\begin{array}{r} + \\ \hline \end{array}$	1
		11101110 = -18	

Como es de esperar, el opuesto del opuesto es el propio número:

-18	=	11101110	(complemento a dos)
complemento bit a bit		=	00010001
		+	1
		<hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/>	
		00010010	= +18

Podemos demostrar la validez de la operación que acabamos de describir utilizando la definición de representación en complemento a dos dada en la Ecuación (9.2). De nuevo interpretamos una secuencia de n dígitos binarios $a_{n-1} a_{n-2} \dots a_1 a_0$ como un entero A en complemento a dos, tal que su valor es:

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Ahora se construye el complemento bit a bit, $\overline{a_{n-1}} \overline{a_{n-2}} \dots \overline{a_0}$, y tratándolo como un entero sin signo, se le suma 1. Finalmente, se interpreta la secuencia resultante de n bits como un entero B en complemento a dos, de manera que su valor es

$$B = -2^{n-1} \overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Ahora queremos que $A = -B$, lo que significa que $A + B = 0$. Esto se comprueba fácilmente:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}}) 2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i}) \right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i \right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

El desarrollo anterior supone que podemos primero tratar el complemento de A bit a bit como entero sin signo al objeto de sumarle 1, y entonces tratar el resultado como un entero en complemento a dos. Hay dos casos especiales a tener en cuenta. En primer lugar, consideremos que $A = 0$. En este caso, para una representación con ocho bits:

0	=	00000000	(complemento a dos)
complemento bit a bit		=	11111111
		+	1
		<hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/>	
		100000000	= 0

Hay un *acarreo* de la posición de bit más significativa, que es ignorado. El resultado es que la negación u opuesto del 0 es 0, como debe ser.

El segundo caso especial es más problemático. Si generamos el opuesto de la combinación de bits consistente en un 1 seguido de $n - 1$ ceros, se obtiene de nuevo el mismo número. Por ejemplo, para palabras de ocho bits,

$-128 = 10000000$	(complemento a dos)
complemento bit a bit	$= 01111111$
	$+ \underline{\hspace{2em}} \quad 1$
	$\hline 10000000 = -128$

Esta anomalía debe evitarse. El número de combinaciones diferentes en una palabra de ocho bits es 2^n , un número par. Con ellas queremos representar enteros positivos, negativos y el 0. Cuando se representa el mismo número de enteros positivos que de negativos (en signo-magnitud) resultan dos representaciones distintas del 0. Si hay solo una representación del 0 (en complemento a dos), entonces debe haber un número desigual de números positivos que de negativos representados. En el caso del complemento a dos, hay una representación de n bits para el -2^{n-1} , pero no para el $+2^{n-1}$.

SUMA Y RESTA

La suma en complemento a dos se ilustra en la Figura 9.3. La suma se efectúa igual que si los números fuesen enteros sin signo. Los cuatro primeros ejemplos muestran operaciones correctas. Si el resultado de la operación es positivo, se obtiene un número positivo en forma de complemento a dos, que tiene la misma forma como entero sin signo. Si el resultado de la operación es negativo, conseguimos un número negativo en forma de complemento a dos. Obsérvese que, en algunos casos, hay un bit acarreo más allá del final de la palabra (sombreado en la figura). Este bit se ignora.

En cualquier suma, el resultado puede que sea mayor que el permitido por la longitud de palabra que está utilizando. Esta condición se denomina **desbordamiento** (*overflow*). Cuando ocurre un

$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$ <p>(a) $(-7) + (+5)$</p>	$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 1000 = 0 \end{array}$ <p>(b) $(-4) + (+4)$</p>
$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$ <p>(c) $(+3) + (+4)$</p>	$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$ <p>(d) $(-4) + (-1)$</p>
$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Desbordamiento} \end{array}$ <p>(e) $(+5) + (+4)$</p>	$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Desbordamiento} \end{array}$ <p>(f) $(-7) + (-6)$</p>

Figura 9.3. Suma de números representados en complemento a dos.

desbordamiento, la ALU debe indicarlo para que no se intente utilizar el resultado obtenido. Para detectar el desbordamiento se debe observar la siguiente regla:

REGLA DE DESBORDAMIENTO: al sumar dos números, y ambos son o bien positivos o negativos, se produce desbordamiento si y solo si el resultado tiene signo opuesto.

Las Figuras 9.3e y f muestran ejemplos de desbordamiento. Obsérvese que el desbordamiento puede ocurrir habiéndose producido o no acarreo.

La resta se trata también fácilmente con la siguiente regla:

REGLA DE LA RESTA: para substraer un número (el substraendo) de otro (minuendo), se obtiene el complemento a dos del substraendo y se le suma al minuendo.

Así pues, la resta se consigue usando la suma, como se muestra en la Figura 9.4. Los dos últimos ejemplos demuestran que también es aplicable la regla de desbordamiento anterior.

Una ilustración gráfica como la mostrada en la Figura 9.5 [BENH92] proporciona una visión más palpable de la suma y la resta en complemento a dos. Los círculos (mitades superiores de la figura) se obtienen a partir de los correspondientes segmentos lineales de números (mitades inferiores), juntando los extremos. Observe que cuando los números se trazan en el círculo, el complemento a dos de cualquier número es el horizontalmente opuesto del mismo (indicado mediante líneas horizontales discontinuas). Comenzando en cualquier número del círculo, al sumarle un positivo k (o restarle un

$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$ (a) $M = 2 = 0010$ $S = 7 = 0111$ $-S = 1001$	$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$ (b) $M = 5 = 0101$ $S = 2 = 0010$ $-S = 1110$
$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$ (c) $M = -5 = 1011$ $S = 2 = 0010$ $-S = 1110$	$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$ (d) $M = 5 = 0101$ $S = -2 = 1110$ $-S = 0010$
$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Desbordamiento} \end{array}$ (e) $M = 7 = 0111$ $S = -7 = 1001$ $-S = 0111$	$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Desbordamiento} \end{array}$ (f) $M = -6 = 1010$ $S = 4 = 0100$ $-S = 1100$

Figura 9.4. Substracción de números en la notación de complemento a dos ($M - S$).

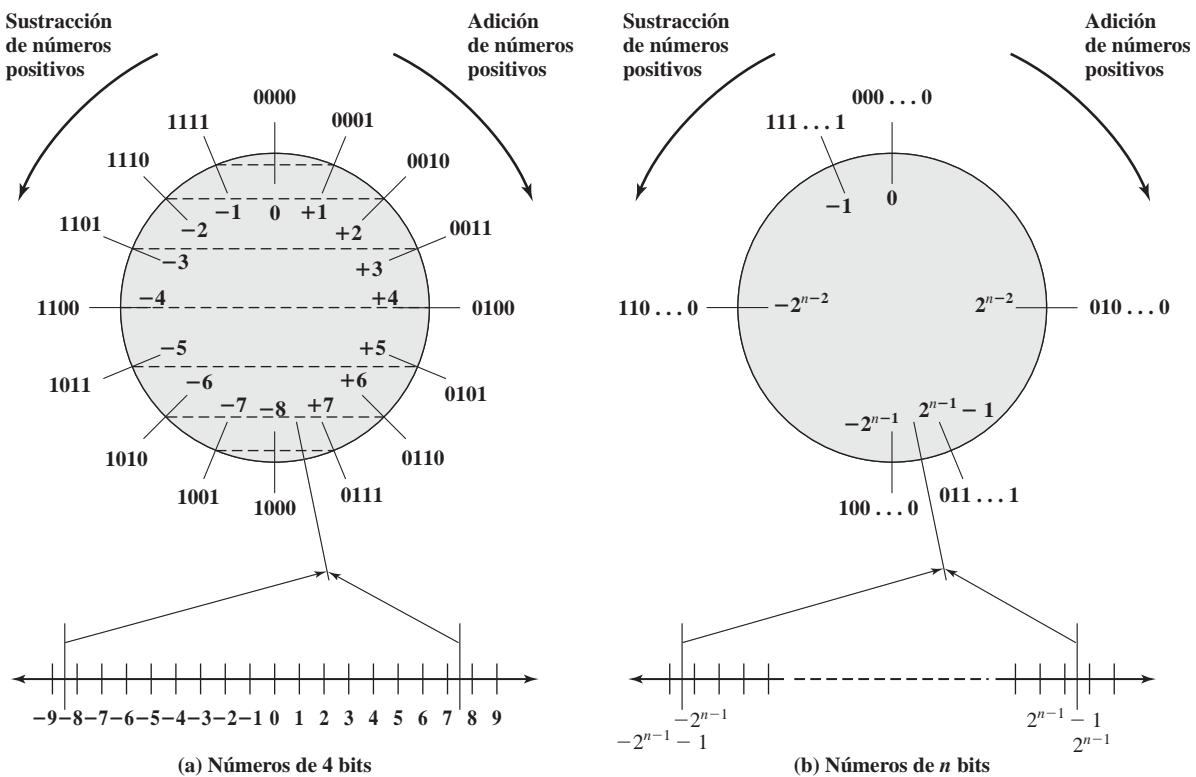
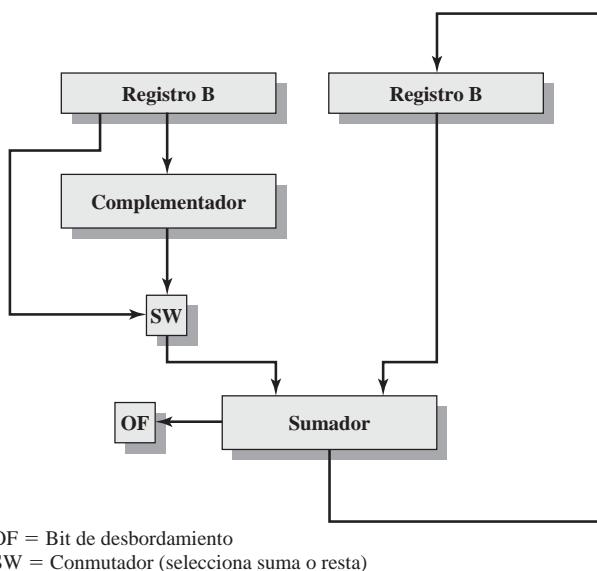


Figura 9.5. Descripción geométrica de los enteros en complemento a dos.

negativo k) nos desplazamos k posiciones en el sentido de las agujas del reloj. Restarle un positivo k (o sumarle un negativo k) equivale a desplazarse k posiciones en sentido contrario a las agujas del reloj. Si la operación realizada hace que se sobrepase el punto en que se juntaron los extremos del segmento, el resultado es incorrecto (desbordamiento).

Todos los ejemplos de las Figuras 9.3 y 9.4 pueden trazarse fácilmente en el círculo de la Figura 9.5.

La Figura 9.6 sugiere los caminos de datos y elementos hardware necesarios para realizar sumas y restas. El elemento central es un sumador binario, al que se presentan los números a sumar y produce una suma y un indicador de desbordamiento. El sumador binario trata los dos números como enteros sin signo (una implementación lógica de un sumador se da en el Apéndice B de este libro). Para sumar, los números se presentan al sumador desde dos registros, designados en este caso registros A y B. El resultado es normalmente almacenado en uno de estos registros en lugar de un tercero. La indicación de desbordamiento se almacena en un indicador o biestable de desbordamiento (OF: *Overflow Flag*) de 1 bit (0 = no desbordamiento; 1 = desbordamiento). Para la resta, el substraendo (registro B) se pasa a través de un complementador, de manera que el valor que se presenta al sumador sea el complemento a dos de B.



OF = Bit de desbordamiento
SW = Conmutador (selecciona suma o resta)

Figura 9.6. Diagrama de bloques del hardware para la suma y la resta.

MULTIPLICACIÓN

Comparada con la suma y la resta, la multiplicación es una operación compleja, ya se realice en hardware o en software. En distintos computadores se han utilizado diversos algoritmos. El propósito de esta subsección es dar al lector una idea del tipo de aproximación normalmente utilizada. Comenzaremos con el caso más sencillo de multiplicar dos enteros sin signo (no negativos), y después veremos una de las técnicas más comunes para el producto de números representados en complemento a dos.

Enteros sin signo. La Figura 9.7 ilustra la multiplicación de enteros binarios sin signo, que se realiza igual que cuando utilizamos papel y lápiz. Se pueden hacer varias observaciones:

1. La multiplicación implica la generación de productos parciales, uno para cada dígito del multiplicador. Estos productos parciales se suman después para producir el producto final.
2. Los productos parciales se definen fácilmente. Cuando el bit del multiplicador es 0, el producto parcial es 0. Cuando el multiplicador es 1, el producto parcial es el multiplicando.

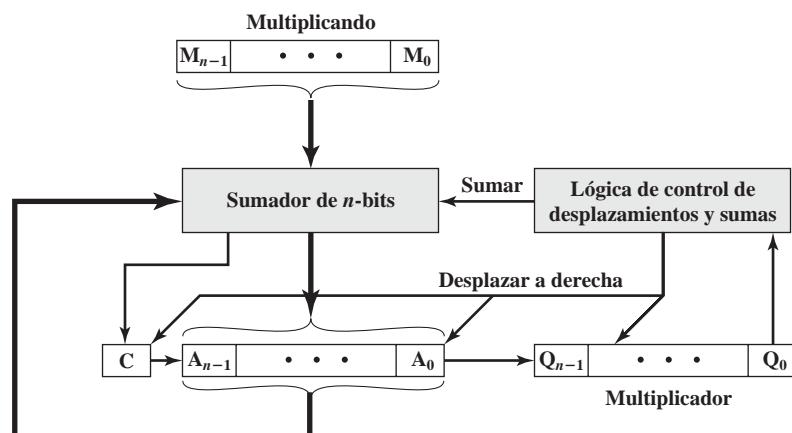
$ \begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array} $	Multiplicando (11) Multiplicador (13) Productos parciales Producto (143)
--	---

Figura 9.7. Multiplicación de enteros binarios sin signo.

3. El producto total se obtiene sumando los productos parciales. Para esta operación, cada producto parcial sucesivo se desplaza en una posición hacia la izquierda con respecto al producto parcial precedente.
4. El producto de dos enteros binarios sin signo de n bits da como resultado un producto de hasta $2n$ bits de longitud (por ejemplo, $11 \times 11 = 1001$).

En comparación con la aproximación de «papel y lápiz», hay varias modificaciones que se pueden hacer para efectuar la operación más eficientemente. En primer lugar, podemos realizar una suma progresiva de los productos parciales en lugar de esperar hasta el final. Esto evita la necesidad de almacenar todos los productos parciales, necesitándose menos registros. En segundo lugar, podemos ahorrar algún tiempo en la generación de los productos parciales. Para cada 1 del multiplicador se requiere un desplazamiento y una suma; pero por cada 0, solo se necesita el desplazamiento.

La Figura 9.8a muestra una posible implementación que hace uso de las ideas anteriores. El multiplicador y el multiplicando están ubicados en dos registros (Q y M). Un tercer registro, el registro A , es también necesario y es inicialmente puesto a 0. Hay también un registro C de un bit, inicializado a 0, que retiene los posibles bits de acarreo resultantes de las sumas.



(a) Diagrama de bloques

C	A	Q	M		Valores iniciales
0	0000	1101	1011		
0	1011	1101	1011	Suma	Primer ciclo
	0101	1110	1011	Desplaz.	
0	0010	1111	1011	Desplaz.	Segundo ciclo
	1101	1111	1011	Suma	
0	0110	1111	1011	Desplaz.	Tercer ciclo
	0001	1111	1011	Suma	
0	1000	1111	1011	Desplaz.	Cuarto ciclo

(b) Ejemplo de la Figura 9.7 (producto en A, Q)

Figura 9.8. Implementación hardware de la multiplicación de binarios sin signo.

La multiplicación se efectúa de la siguiente manera. La lógica de control lee uno por uno los bits del multiplicador. Si Q_0 es 1, se suma el multiplicando al registro A y el resultado es almacenado en A, utilizando el bit C para el acarreo. Entonces se desplazan todos los bits de los registros C, A, y Q, una posición a la derecha, de manera que el bit de C pasa a A_{n-1} , A_0 pasa a Q_{n-1} , y Q_0 se pierde. Si Q_0 era 0, no se realiza la suma, solo el desplazamiento. Este proceso se repite para cada bit del multiplicador original. El producto de $2n$ bits resultante queda en los registros A y Q. La Figura 9.9 muestra un diagrama de flujo de la operación, y en la Figura 9.8b se da un ejemplo. Obsérvese que en el ciclo segundo, cuando el bit del multiplicador es 0, no hay operación de suma.

Multiplicación en complemento a dos. Hemos visto que la suma y la resta pueden realizarse con números en notación de complemento a dos, tratándolos como enteros sin signo. Consideremos:

$$\begin{array}{r} 1001 \\ +0011 \\ \hline 1100 \end{array}$$

Si estos números se interpretan como enteros sin signo, estamos sumando 9 (1001) más 3 (0011) para obtener 12 (1100). Como enteros en complemento a dos, estamos sumando -7 (1001) a 3 (0011) para obtener -4 (1100).

Desafortunadamente, este sencillo esquema no es correcto para la multiplicación. Para verlo, consideremos de nuevo la Figura 9.7. Multiplicamos 11 (1011) por 13 (1101) para obtener 143 (10001111). Si interpretamos estos como números en complemento a dos, tendríamos -5 (1011)

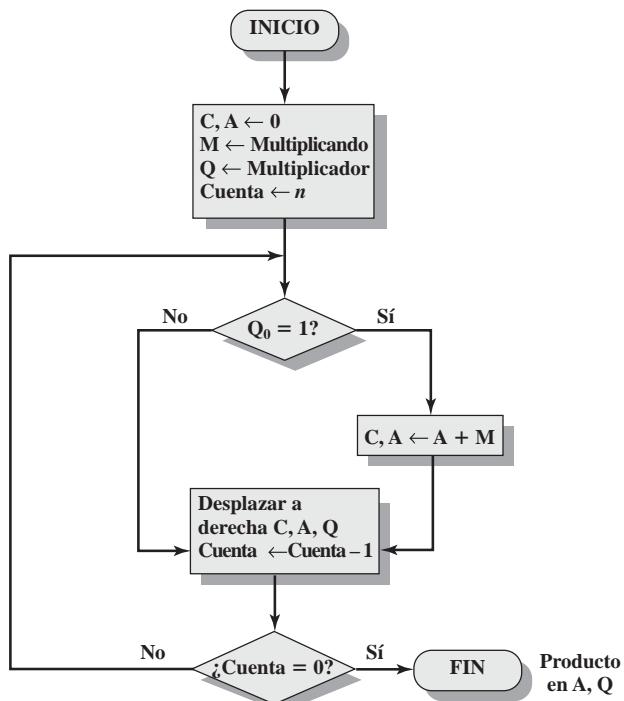


Figura 9.9. Diagrama de flujo para la multiplicación de binarios sin signo.

por -3 (1101) igual a -113 (10001111). Este ejemplo demuestra que la multiplicación directa no es adecuada si tanto el multiplicando como el multiplicador son negativos. De hecho, tampoco lo es si alguno de los dos es negativo. Para explicar este comportamiento necesitamos volver sobre la Figura 9.7 y explicar lo que se está haciendo en términos de operaciones con potencias de 2 . Recuérdese que cualquier número binario sin signo puede expresarse como suma de potencias de 2 . Por tanto,

$$\begin{aligned} 1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 2^3 + 2^2 + 2^0 \end{aligned}$$

Además, el producto de un número binario por 2^n se obtiene desplazando dicho número n bits hacia la izquierda. Teniendo esto en mente, la Figura 9.10 reestructura la Figura 9.7 para hacer la generación de productos parciales mediante multiplicación explícita. La única diferencia en la Figura 9.10 es que reconoce que los productos parciales debieran verse como números de $2n$ bits generados a partir del multiplicando de n bits.

Así pues, el multiplicando de cuatro bits 1011 , como entero sin signo, es almacenado en una palabra de ocho bits como 00001011 . Cada producto parcial (distinto del correspondiente a 2^0) consiste en dicho número desplazado a la izquierda, con las posiciones de la derecha llenadas con ceros (por ejemplo, un desplazamiento a la izquierda en dos posiciones produce 00101100).

Ahora podemos demostrar cómo la multiplicación directa no es correcta si el multiplicando es negativo. El problema es que cada contribución del multiplicando negativo como producto parcial tiene que ser un número negativo en un campo de $2n$ bits; los bits de signo de los productos parciales deben estar alineados. Esto se demuestra en la Figura 9.11, que muestra el producto de 1001 por 0011 . Si estos se tratan como enteros sin signo se realiza el producto $9 \times 3 = 27$. Sin embargo, si 1001 se interpreta en complemento a dos como -7 , cada producto parcial debe ser un número negativo en complemento a dos de $2n$ (es decir, ocho) bits, como muestra la Figura 9.11b. Obsérvese que eso podría hacerse rellenando la parte izquierda de cada producto parcial con unos.

$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 00001011 \end{array}$	$1011 \times 1 \times 2^0$
	$00000000 \quad 1011 \times 0 \times 2^1$
	$00101100 \quad 1011 \times 1 \times 2^2$
	$01011000 \quad 1011 \times 1 \times 2^3$
	$\hline 10001111$

Figura 9.10. Multiplicación de dos enteros sin signo de cuatro bits para producir un resultado de ocho bits.

$\begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ 00011011 \quad (27) \end{array}$	$\begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ 11101011 \quad (-21) \end{array}$
--	---

(a) Enteros sin signo

(b) Enteros en complemento a dos

Figura 9.11. Comparación del producto de enteros sin signo y en complemento a dos.

Si el multiplicador es negativo, la multiplicación directa tampoco es correcta. La razón es que los bits del multiplicador ya no se corresponden con los desplazamientos o productos que deben producirse. Por ejemplo, el número decimal -3 se representa con cuatro bits en complemento a dos como 1101 . Si simplemente tomamos los productos parciales basándonos en cada posición de bit, tendríamos la siguiente correspondencia:

$$1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

De hecho, lo que se quiere es $-(2^1 + 2^0)$. Por tanto este multiplicador no puede utilizarse directamente en la forma anteriormente descrita.

Hay varias maneras de salir de este dilema. Una sería convertir tanto el multiplicando como el multiplicador en números positivos, realizar el producto y obtener después el complemento a dos del resultado si y solo si el signo de los dos números iniciales difiere. Los diseñadores han preferido utilizar técnicas que no requieren esta etapa de transformación final. Una de las técnicas más comunes es el algoritmo de Booth. Este algoritmo tiene la ventaja adicional de acelerar el proceso de multiplicación con respecto a una aproximación más directa.

El algoritmo de Booth se ilustra en la Figura 9.12 y puede describirse como sigue. Como antes, el multiplicador y el multiplicando se ubican en los registros Q y M respectivamente. Hay también un registro de un bit con una ubicación lógica a la derecha del bit menos significativo (Q_0) del registro Q, y que denominamos Q_{-1} ; explicamos brevemente su uso. El producto resultante aparecerá en los

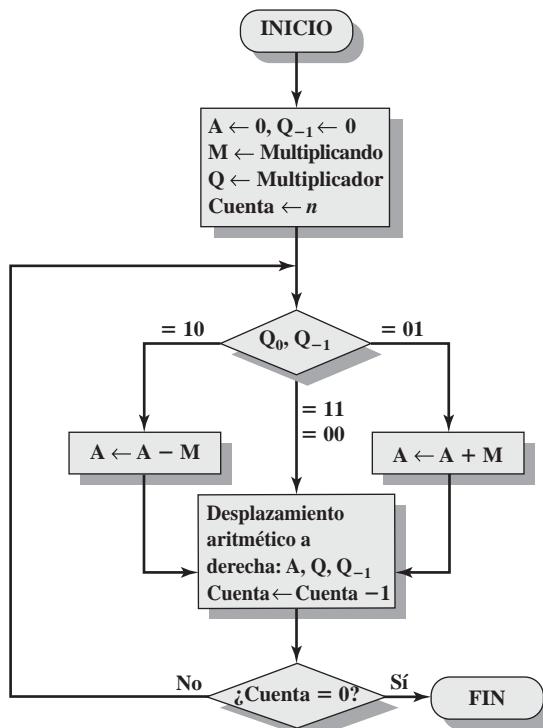


Figura 9.12. Algoritmo de Booth para la multiplicación en complemento a dos.

registros A y Q. A y Q_{-1} se fijan inicialmente a 0. Como antes, la lógica de control recorre los bits del multiplicador uno por uno. Ahora, al examinar cada bit, también se comprueba el bit a su derecha; si los dos son iguales (1-1 ó 0-0), todos los bits de los registros A, Q, y Q_{-1} se desplazan un bit a la derecha. Si dichos bits difieren, el multiplicando se suma o se resta al registro A, según que los dos bits sean 0-1 ó 1-0. A continuación de la suma o resta se realiza un desplazamiento a la derecha. En cualquier caso, el desplazamiento a la derecha es tal que el bit más a la izquierda de A, es decir A_{n-1} , no solo se desplaza a A_{n-2} , sino que también queda en A_{n-1} . Esto es necesario para preservar el signo del número contenido en la pareja de registros A y Q. Este desplazamiento se denomina **desplazamiento aritmético**, ya que preserva el bit de signo.

La Figura 9.13 muestra la secuencia de eventos para multiplicar siete por tres con el algoritmo de Booth. La misma operación se describe de manera más compacta en la Figura 9.14a.

El resto de la Figura 9.14 da otros ejemplos del algoritmo. Como puede verse actúa correctamente con cualquier combinación de números positivos y negativos. Obsérvese también la eficiencia del algoritmo. Los bloques de unos o de ceros se saltan, con un promedio de solo una suma o resta por bloque.

A	Q	Q_{-1}	M		Valores iniciales
0000	0011	0	0111		
1001	0011	0	0111	A $\leftarrow A - M$	Primer
1100	1001	1	0111	Desplazam.	ciclo
1110	0100	1	0111	Desplazam.	Segundo
0101	0100	1	0111	A $\leftarrow A + M$	Tercer
0010	1010	0	0111	Desplazam.	ciclo
0001	0101	0	0111	Desplazam.	Cuarto
					ciclo

Figura 9.13. Ejemplo de aplicación del algoritmo de Booth (7×3).

$ \begin{array}{r} 0111 \\ \times 0011 \\ \hline 11111001 \\ 0000000 \\ \hline 000111 \\ \hline 00010101 \end{array} $ (0)	$ \begin{array}{r} 0111 \\ \times 1101 \\ \hline 11111001 \\ 0000111 \\ \hline 111001 \\ \hline 11101011 \end{array} $ (0)
--	--

(a) $(7) \times (3) = (21)$

(b) $(7) \times (-3) = (-21)$

$ \begin{array}{r} 1001 \\ \times 0011 \\ \hline 00000111 \\ 0000000 \\ \hline 111001 \end{array} $ (0)	$ \begin{array}{r} 1001 \\ \times 1101 \\ \hline 00000111 \\ 1111001 \\ \hline 000111 \end{array} $ (0)
---	---

(c) $(-7) \times (3) = (-21)$

(d) $(-7) \times (-3) = (21)$

Figura 9.14. Ejemplos de uso del algoritmo de Booth.

¿Por qué se comporta correctamente el algoritmo de Booth? Considere primero el caso en que el multiplicador sea positivo. En particular, un multiplicador positivo consistente en un bloque de unos con ceros a ambos lados (por ejemplo 00011110). Como sabemos, la multiplicación puede obtenerse sumando adecuadamente copias desplazadas del multiplicando:

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

El número de tales operaciones puede reducirse a dos si observamos que

$$2^n + 2^{n-1} + \dots + 2^{n-K} = 2^{n+1} - 2^{n-K} \quad (9.3)$$

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

Así pues, el producto puede generarse mediante una suma y una resta del multiplicando. Este esquema se extiende a cualquier número de bloques de unos del multiplicador, incluyendo el caso en que un solo 1 es tratado como bloque.

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

El algoritmo de Booth obedece a este esquema, realizando una resta cuando se encuentra el primer 1 del bloque (1-0), y una suma cuando se encuentra el final (0-1) del bloque.

Para comprobar que el mismo esquema es adecuado en el caso de un multiplicador negativo, necesitamos observar lo siguiente. Sea X un número negativo en notación de complemento a dos:

Representación de $X = \{1x_{n-2}x_{n-2} \dots x_1x_0\}$

Entonces el valor de X puede expresarse como sigue:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0) \quad (9.4)$$

El lector puede verificarlo aplicando el algoritmo a los números de la Tabla 9.2.

El bit más a la izquierda de X es 1, ya que X es negativo. Suponga que el bit 0 más a la izquierda está en la posición k -ésima. Entonces X será en la forma:

$$\text{Representación de } X = \{111 \dots 10x_{k-1}x_{k-2} \dots x_1x_0\} \quad (9.5)$$

Entonces el valor de X es:

$$X = -2^{n-1} + 2^{n-2} + \dots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0) \quad (9.6)$$

Utilizando la ecuación (9.3) podemos decir que:

$$2^{n-2} + 2^{n-3} + \dots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Reagrupando se tiene:

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^{k+1} = -2^{k+1} \quad (9.7)$$

Sustituyendo la ecuación (9.7) en la (9.6) tenemos:

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0) \quad (9.8)$$

Al fin podemos volver sobre el algoritmo de Booth. Recordando la representación de X [(Ecuación (9.5))], está claro que todos los bits desde x_0 hasta el 0 más a la izquierda son manipulados de forma apropiada, ya que producen todos los términos de la ecuación (9.8) excepto el (-2^{k+1}) . Cuando el algoritmo recorre hasta el 0 más a la izquierda y encuentra el 1 siguiente (2^{k+1}), ocurre una transición 1-0 y tiene lugar una resta (-2^{k+1}) . Este es el término restante de la ecuación (9.8).

Como ejemplo, considere la multiplicación de algún multiplicando por (-6) . En la representación de complemento a dos, utilizando una palabra de ocho bits, (-6) se expresa como 11111010. Por la ecuación (9.4) sabemos que:

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

como puede verificar fácilmente el lector. Por tanto,

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Usando la ecuación (9.7),

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

que, como el lector puede verificar, es también $M \times (-6)$. Finalmente, siguiendo nuestra línea de razonamiento anterior:

$$M \times (11111010) = M \times (-2^3 + 2^2 - 2^1)$$

Podemos ver que el algoritmo de Booth se ajusta a este esquema. Realiza una resta cuando se encuentra el primer 1, (1-0), una suma cuando se encuentra (0-1), y finalmente resta otra vez cuando encuentra el primer 1 del siguiente bloque de unos. Por consiguiente, el algoritmo de Booth realiza menos sumas y restas que un algoritmo directo.

DIVISIÓN

La división es algo más compleja que la multiplicación pero está basada en los mismos principios generales. Como antes, la base para el algoritmo es la aproximación de «papel y lápiz», y la operación conlleva repetidos desplazamientos y sumas o restas.

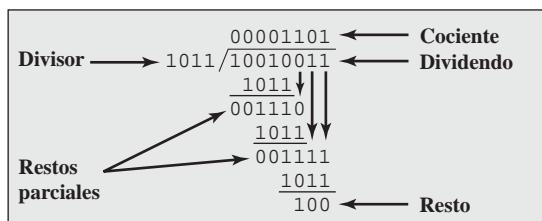


Figura 9.15. División de enteros binarios sin signo.

La Figura 9.15 muestra un ejemplo de división larga de enteros binarios sin signo. Es instructivo describir en detalle el proceso. Primero se examinan los bits del dividendo de izquierda a derecha hasta que el conjunto de bits examinados represente un número mayor o igual que el divisor; o, en otras palabras, hasta que el divisor sea capaz de dividir al número. Hasta que eso ocurre, se van colocando ceros en el cociente de izquierda a derecha. Cuando dicho evento ocurre, se coloca un 1 en el cociente, y se substrae el divisor del dividendo parcial. Al resultado se le denomina *resto parcial*. Desde este punto en adelante, la división sigue un patrón cíclico. En cada ciclo, se añaden bits adicionales del dividendo al resto parcial hasta que el resultado sea mayor o igual que el divisor. Como antes, de este número se resta el divisor para producir un nuevo resto parcial. El proceso continúa hasta que se acaban los bits del dividendo.

La Figura 9.16 muestra un algoritmo máquina que corresponde al proceso de división larga descrito. El divisor se ubica en el registro M, y el dividendo en el registro Q. En cada paso, los registros A y Q son desplazados conjuntamente un bit a la izquierda. M es restado de A para determinar si A divide el resto parcial². Si esto se cumple, entonces Q_0 se hace 1. Si no, Q_0 se hace 0, y M debe sumarse de nuevo a A para restablecer el valor anterior. La cuenta entonces se decremente, y el proceso continúa hasta n pasos. Al final, el cociente queda en el registro Q y el resto en el A.

Este proceso puede, con cierta dificultad, aplicarse también a números negativos. Aquí damos una posible aproximación para números en complemento a dos. En la Figura 9.17 se muestran varios ejemplos de esta aproximación. El algoritmo puede resumirse como sigue:

1. Cargar el divisor en el registro M y el dividendo en los registros A y Q. El dividendo debe estar expresado como número en complemento a dos de $2n$ bits. Por ejemplo, el número de 4 bits 0111 pasa a ser 00000111, y el 1001 pasa a 11111001.
2. Desplazar A y Q una posición de bit a la izquierda.
3. Si M y A tienen el mismo signo, ejecutar $A \leftarrow A - M$; si no, $A \leftarrow A + M$.
4. La operación anterior tiene éxito si el signo de A es el mismo antes y después de la operación.
 - a) Si la operación tiene éxito o $A = 0$, entonces hacer $Q_0 \leftarrow 1$.
 - b) Si la operación no tiene éxito y $A \neq 0$, entonces $Q_0 \leftarrow 0$, y restablecer el valor anterior de A.

² Es una resta de enteros sin signo. Si se produce un acarreo negativo (adeudo) a partir del bit más significativo, el resultado es negativo.

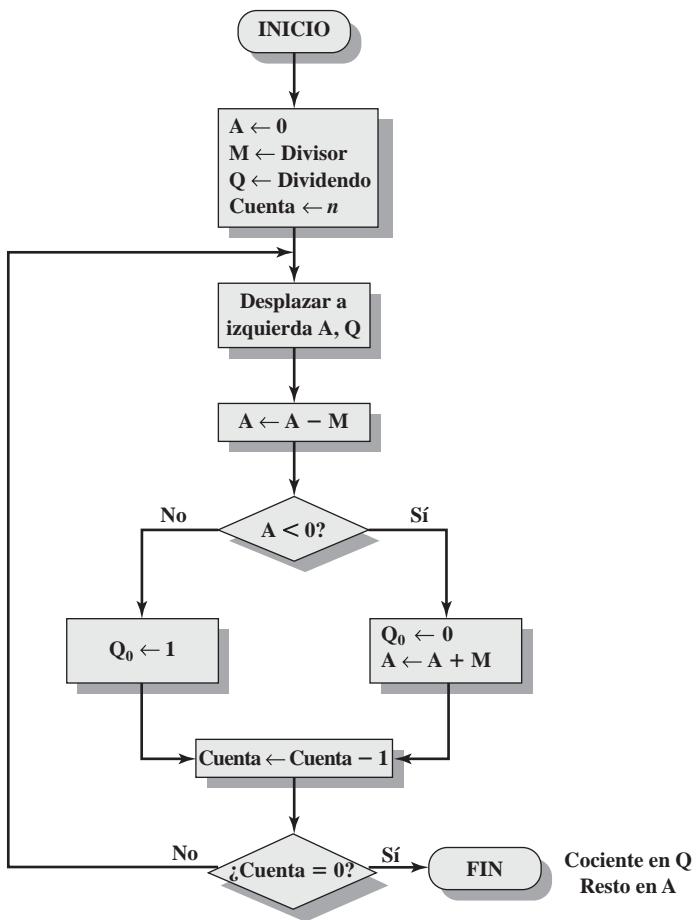


Figura 9.16. Diagrama de flujo para la división de binarios sin signo.

5. Repetir los pasos 2 a 4 tantas veces como número de bits tenga Q.
6. El resto está en A. Si los signos del divisor y el dividendo eran iguales, el cociente está en Q; si no, el cociente correcto es el complemento a dos de Q.

El lector notará en la Figura 9.17 que $(-7)/(3)$ y $(7)/(-3)$ producen restos diferentes. Esto es debido a que el resto se define como:

$$D = Q \times V + R$$

en donde:

D = dividendo

Q = cociente

V = divisor

R = resto

Los resultados de la Figura 9.17 son consistentes con dicha fórmula.

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Valor inicial	0000	0111	Valor inicial
0000	1110	Desplazamiento	0000	1110	Desplazamiento
1101		Restar	1101		Sumar
0000	1110	Restablecer	0000	1110	Restablecer
0001	1100	Desplazamiento	0001	1100	Desplazamiento
1110		Restar	1110		Sumar
0001	1100	Restablecer	0001	1100	Restablecer
0011	1000	Desplazamiento	0011	1000	Desplazamiento
0000		Restar	0000		Sumar
0000	1001	Poner Q ₀ = 1	0000	1001	Poner Q ₀ = 1
0001	0010	Desplazamiento	0001	0010	Desplazamiento
1110		Restar	1110		Sumar
0001	0010	Restablecer	0001	0010	Restablecer

(a) (7)/(3)

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Valor inicial	1111	1001	Valor inicial
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer
1110	0100	Desplazamiento	1110	0100	Desplazamiento
0001		Sumar	0001		Restar
1110	0100	Restablecer	1110	0100	Restablecer
1100	1000	Desplazamiento	1100	1000	Desplazamiento
1111		Sumar	1111		Restar
1111	1001	Poner Q ₀ = 1	1111	1001	Poner Q ₀ = 1
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer

(b) (7)/(-3)

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Valor inicial	1111	1001	Valor inicial
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer
1110	0100	Desplazamiento	1110	0100	Desplazamiento
0001		Sumar	0001		Restar
1110	0100	Restablecer	1110	0100	Restablecer
1100	1000	Desplazamiento	1100	1000	Desplazamiento
1111		Sumar	1111		Restar
1111	1001	Poner Q ₀ = 1	1111	1001	Poner Q ₀ = 1
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer

(c) (-7)/(3)

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Valor inicial	1111	1001	Valor inicial
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer
1110	0100	Desplazamiento	1110	0100	Desplazamiento
0001		Sumar	0001		Restar
1110	0100	Restablecer	1110	0100	Restablecer
1100	1000	Desplazamiento	1100	1000	Desplazamiento
1111		Sumar	1111		Restar
1111	1001	Poner Q ₀ = 1	1111	1001	Poner Q ₀ = 1
1111	0010	Desplazamiento	1111	0010	Desplazamiento
0010		Sumar	0010		Restar
1111	0010	Restablecer	1111	0010	Restablecer

(d) (-7)/(-3)

Figura 9.17. Ejemplos de división en complemento a dos.

9.4. REPRESENTACIÓN EN COMA FLOTANTE

FUNDAMENTOS

Con una notación de coma fija (por ejemplo, la representación en complemento a dos) es posible representar un rango de enteros positivos y negativos centrado en el cero. Asumiendo una coma binaria fija, dicho formato permite también representar números con parte fraccionaria.

La aproximación anterior tiene limitaciones. Los números muy grandes no pueden representarse, ni tampoco las fracciones muy pequeñas. Además, en la división de dos números grandes puede perderse la parte fraccionaria del cociente.

Para números decimales, esta limitación se supera utilizando la notación científica. Así, 976.000.000.000.000 puede representarse como 9.76×10^{14} , y 0,0000000000000976 puede expresarse como 9.76×10^{-14} . Lo que se ha hecho es mover dinámicamente la coma decimal a una posición

conveniente y utilizar el exponente del diez para mantener registrada la posición de la coma. Esto permite representar un rango de números muy grandes y muy pequeños con solo unos cuantos dígitos.

Esa misma técnica puede aplicarse a los números binarios. Podemos representar un número en la forma

$$\pm S \times B^{\pm E}$$

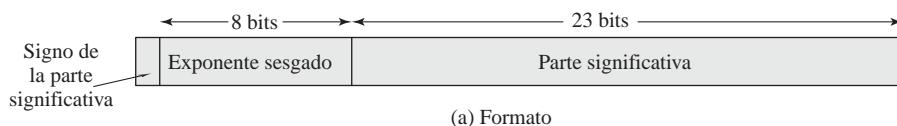
Este número puede almacenarse en una palabra binaria con tres campos:

- Signo: más o menos
- Parte significativa o mantisa S (del término inglés *significand*)
- Exponente E

La **base** B está implícita y no necesita memorizarse ya que es la misma para todos los números. Normalmente se supone que la coma de la base está a la derecha del bit más a la izquierda, o más significativo, de la mantisa. Es decir, se asume que hay un bit a la izquierda de la coma de la base.

Las reglas utilizadas en la representación de números binarios en coma flotante se explican mejor con un ejemplo. La Figura 9.18a muestra un formato típico de coma flotante de 32 bits. El bit más a la izquierda contiene el **signo** del número (0 = positivo, 1 = negativo). El valor del **exponente** se almacena en los ocho bits siguientes. La representación utilizada se conoce con el término de **representación sesgada**. Un valor fijo, llamado sesgo, se resta de este campo para conseguir el valor del exponente verdadero. Normalmente, el sesgo tiene un valor $(2^{k-1} - 1)$, donde k es el número de bits en el exponente binario. En este caso, un campo de ocho bits comprende números entre 0 y 255. Con un sesgo de 127, $(2^7 - 1)$, los valores verdaderos de exponente varían en el rango -127 a +128. En este ejemplo, se supone la base 2.

La Tabla 9.2 mostraba la representación sesgada para enteros de cuatro bits. Observe que cuando los bits de una representación sesgada se tratan como enteros sin signo, las magnitudes relativas de los números no cambian. Por ejemplo, tanto en la representación sin signo como en la sesgada, el número más grande es el 1111 y el más pequeño el 0000. Esto no se cumple para las representaciones en signo-magnitud y en complemento a dos. Una ventaja de la representación sesgada es que los números en coma flotante no negativos pueden ser tratados de la misma forma que los enteros al efectuar comparaciones.



(a) Formato

$$\begin{array}{rcl}
 1,1010001 \times 2^{10100} & = & 0\ 10010011\ 10100010000000000000000000000000 = 1,6328125 \times 2^{20} \\
 -1,1010001 \times 2^{10100} & = & 1\ 10010011\ 10100010000000000000000000000000 = -1,6328125 \times 2^{20} \\
 1,1010001 \times 2^{-10100} & = & 0\ 01101011\ 10100010000000000000000000000000 = 1,6328125 \times 2^{-20} \\
 -1,1010001 \times 2^{-10100} & = & 1\ 01101011\ 10100010000000000000000000000000 = -1,6328125 \times 2^{-20}
 \end{array}$$

(b) Ejemplos

Figura 9.18. Formato típico de 32 bits en coma flotante.

La parte final de la palabra (23 bits en el ejemplo en este caso) es la **parte significativa**³.

Cualquier número en coma flotante puede expresarse de distintas formas.

Las siguientes representaciones, con la parte significativa expresada en forma binaria, son equivalentes:

$$\begin{aligned} &0,110 \times 2^5 \\ &\quad 110 \times 2^2 \\ &0,0110 \times 2^6 \end{aligned}$$

Para simplificar los cálculos con números en coma flotante se requiere usualmente que estén normalizados. En un número normalizado, el dígito más significativo de la parte significativa es distinto de cero. Por lo tanto, para la representación en base 2, el bit más significativo de la parte significativa de un número normalizado debe ser uno. Como habíamos mencionado, es una convención usual que se tenga un bit a la izquierda de la coma de la base.

Por lo tanto, un número normalizado en base dos tiene la forma:

$$\pm 0,1bbb\ldots b \times 2^{\pm E}$$

en donde cada b es un dígito binario (1 ó 0). Ya que el bit más significativo es siempre un 1, es innecesario almacenar este bit, está implícito. Así, el campo de 23 bits se emplea para albergar una parte significativa de 24 bits con un valor en el intervalo abierto [1, 2). Un número que no esté normalizado puede normalizarse desplazando la coma de la base hasta que quede a la derecha del bit más a la izquierda, y ajustando convenientemente el exponente.

La Figura 9.18b da algunos ejemplos de números almacenados en este formato. Observe las siguientes características:

- El signo se almacena en el primer bit de la palabra.
- El primer bit de la parte significativa original siempre es 1 y no necesita almacenarse en el campo asignado a la parte significativa.
- Se suma 127 al exponente original para almacenarlo en el campo de exponente.
- La base es 2.

La Figura 9.19 compara los rangos de números que pueden representarse en una palabra de 32 bits. Usando la notación entera en complemento a dos, pueden representarse todos los enteros desde -2^{31} hasta $2^{31} - 1$, con un total de 2^{32} números diferentes. Con el ejemplo de formato en coma flotante de la Figura 9.18 son posibles los siguientes rangos de números:

³ El autor considera que el término *mantisa*, que suele usarse en lugar de parte *significativa*, está obsoleto. La mantisa representa también la parte fraccionaria de un logaritmo. Por eso, salvo ocasiones en que pueda haber confusión en el texto, evitaremos usar el término.

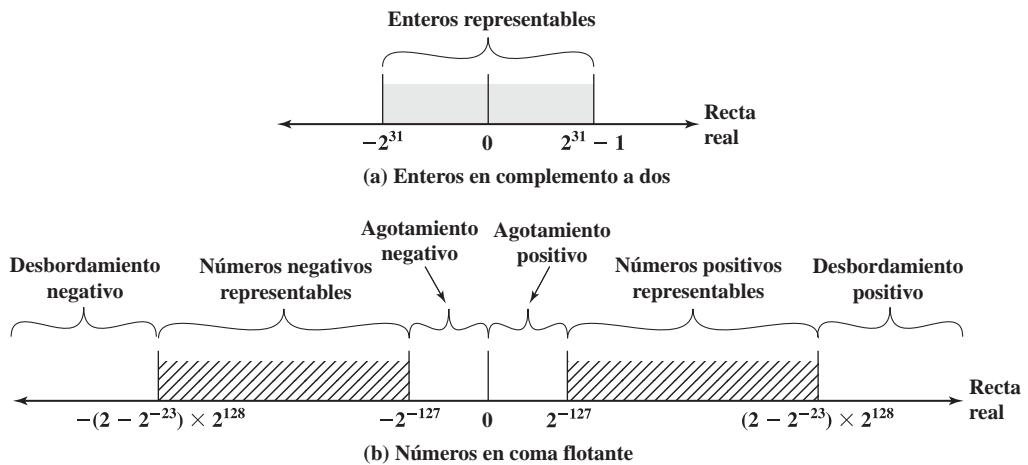


Figura 9.19. Números representables en formatos típicos de 32 bits.

- Números negativos entre $-(2 - 2^{-23}) \times 2^{128}$ y -2^{-127}
- Números positivos entre 2^{-127} y $(2 - 2^{-23}) \times 2^{128}$

En la recta de los números reales hay cinco regiones excluidas de dichos rangos:

- Los números negativos menores que $-(2 - 2^{-23}) \times 2^{128}$, región denominada **desbordamiento negativo**.
- Los números negativos mayores que 2^{-127} , denominada **agotamiento negativo**.
- El cero.
- Los números positivos menores que 2^{-127} , región denominada **agotamiento positivo**.
- Los números positivos mayores que $(2 - 2^{-23}) \times 2^{128}$, denominada **desbordamiento positivo**.

La representación indicada no contempla un valor para el 0. Sin embargo, como veremos, en la práctica se incluye una combinación de bits especial para designar el cero. Un desbordamiento ocurre cuando una operación aritmética da lugar a un número cuyo exponente es mayor que 128 (por ejemplo, $2^{120} \times 2^{100} = 2^{220}$ produciría desbordamiento). Un agotamiento ocurre cuando una magnitud fraccionaria es demasiado pequeña (por ejemplo, $2^{-120} \times 2^{-100} = 2^{-220}$). Un agotamiento es un problema menos serio porque el resultado puede generalmente aproximarse satisfactoriamente por 0.

Es importante observar que con la notación en coma flotante no estamos representando más valores individuales. El máximo número de valores diferentes que pueden representarse con 32 bits es 2^{32} . Lo que hemos hecho es repartir dichos números a lo largo de dos intervalos, uno positivo y otro negativo.

Obsérvese también que, al contrario de lo que ocurre con los números en coma fija, los números representados en la notación de coma flotante no están espaciados por igual a lo largo de la recta real.

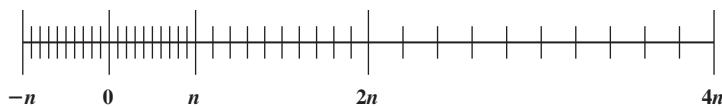


Figura 9.20. Densidad de los números en coma flotante.

Los posibles valores están más próximos cerca del origen y más separados a medida que nos alejamos de él, según se muestra en la Figura 9.20. Este es uno de los inconvenientes del cálculo en coma flotante: muchos cálculos producen resultados que no son exactos y tienen que redondearse al valor más próximo representable con la notación.

En el formato indicado en la Figura 9.18 existe un compromiso entre rango y precisión. El ejemplo muestra ocho bits dedicados al exponente y 23 bits a la parte significativa. Si incrementamos el número de bits del exponente expandimos el rango de números representables. Pero ya que solo puede expresarse un número dado de valores diferentes, habríamos reducido la densidad de dichos números y en consecuencia la precisión. La única forma de incrementar tanto el rango como la precisión es utilizar más bits. Así, la mayoría de los computadores ofrecen la posibilidad de utilizar, al menos, números de precisión simple y números en doble precisión. Por ejemplo, un formato de precisión simple podría ser de 32 bits, y uno de precisión doble de 64 bits.

Existe pues un compromiso entre el número de bits del exponente y el de la parte significativa. En realidad se trata de algo más complicado, ya que la base del exponente no tiene por qué ser dos. Por ejemplo, la arquitectura IBM S/390 utilizaba la base 16 [ANDE67b]. El formato consiste en un exponente de 7 bits y una parte significativa de 24 bits.

Con el formato en base 16 de IBM,

$$0,11010001 \times 2^{10100} = 0,11010001 \times 16^{101}$$

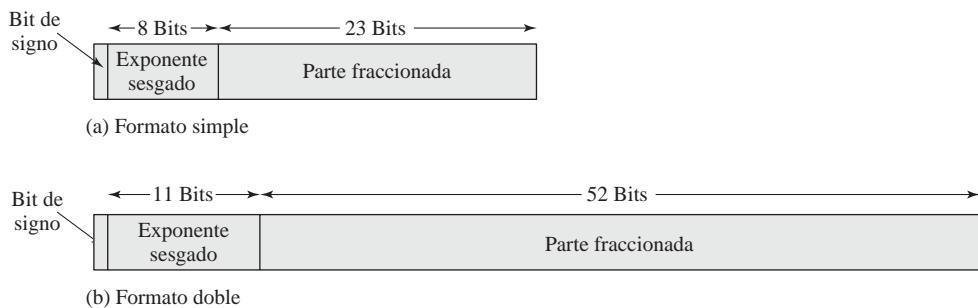
y el exponente que se almacena representa el 5 en lugar del 20.

La ventaja de utilizar una base mayor es que, para el mismo número de bits de exponente, se puede conseguir un rango de números mayor. Pero recuerde que con ello no incrementamos el número de valores diferentes que pueden ser representados. Así, para un formato dado, una base del exponente mayor proporciona un rango mayor a costa de menor precisión.

ESTÁNDAR DEL IEEE PARA LA REPRESENTACIÓN BINARIA EN COMA FLOTANTE

La representación en coma flotante más importante es la definida en la norma o estándar 754 del IEEE, y adoptada en 1985. Este estándar se desarrolló para facilitar la portabilidad o transferencia de los programas de un procesador a otro y para alentar el desarrollo de programas numéricos sofisticados. Este estándar ha sido ampliamente adoptado y se utiliza prácticamente en todos los procesadores y los coprocesadores aritméticos actuales.

El estándar del IEEE define tanto el formato simple de 32 bits como el doble de 64 bits (Figura 9.21), con exponentes de ocho y once bits respectivamente. La base implícita es dos. Además defi-

**Figura 9.21.** Formatos IEEE 754.

ne dos formatos ampliados, simple y doble, cuya forma exacta depende de la implementación (puede variar de unos procesadores a otros). Estos formatos ampliados incluyen bits adicionales en el exponente (rango ampliado o extendido) y en la parte significativa (precisión ampliada). Los formatos ampliados se utilizan para cálculos intermedios. Con su mayor precisión, dichos formatos reducen la posibilidad de que el resultado final se vea deteriorado por un error excesivo de redondeo; con su mayor intervalo de variación, también reducen la posibilidad de un desbordamiento intermedio que aborte un cálculo cuyo resultado habría sido representable en un formato básico. Una motivación adicional para el formato ampliado simple es que proporciona algunas de las ventajas del formato doble sin incurrir en la penalización de tiempo normalmente asociada con una precisión más elevada. La Tabla 9.3 resume las características de los cuatro formatos indicados.

En los formatos del IEEE no todos los patrones de bits se interpretan de la manera habitual. Algunas combinaciones se emplean para representar valores especiales. La Tabla 9.4 indica los valores asignados a ciertos patrones de bits. Los valores extremos de exponente consistentes en todo ceros

Tabla 9.3. Parámetros del formato IEEE 754.

Parámetro	Formato			
	Simple	Simple ampliado	Doble	Doble ampliado
Longitud de palabra (bits)	32	≥ 43	64	≥ 79
Longitud de exponente (bits)	8	≥ 11	11	≥ 15
Sesgo del exponente	127	sin especificar	1023	sin especificar
Exponente máximo	127	≥ 1023	1023	≥ 16.383
Exponente mínimo	-126	≤ -1022	-1022	≤ -16.382
Rango de números (base 10)	$10^{-38}, 10^{+38}$	sin especificar	$10^{-308}, 10^{+308}$	sin especificar
Longitud de mantisa (bits)*	23	≥ 31	52	≥ 63
Número de exponentes	254	sin especificar	2046	sin especificar
Número de fracciones	2^{23}	sin especificar	2^{52}	sin especificar
Número de valores	1.98×2^{31}	sin especificar	1.99×2^{53}	sin especificar

* Excluyendo el bit implícito.

Tabla 9.4. Interpretación de los números en la notación de coma flotante IEEE 754.

	Precisión simple (32 bits)				Doble precisión (64bits)			
	Signo	Exponente sesgado	Parte fraccionaria	Valor	Signo	Exponente sesgado	Parte fraccionaria	Valor
Cero positivo	0	0	0	0	0	0	0	0
Cero negativo	1	0	0	-0	1	0	0	0
Más infinito	0	255 (todo unos)	0	∞	0	2047 (todo unos)	0	∞
Menos infinito	1	255 (todo unos)	0	$-\infty$	1	2047 (todo unos)	0	$-\infty$
NaN silencioso	0 ó 1	255 (todo unos)	$\neq 0$	NaN	0 ó 1	2047 (todo unos)	$\neq 0$	NaN
NaN indicador	0 ó 1	255 (todo unos)	$\neq 0$	NaN	0 ó 1	1047 (todo unos)	$\neq 0$	NaN
Positivo normalizado \neq cero	0	$0 < e < 255$	f	$2^{e-127}(1,f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1,f)$
Negativo normalizado \neq cero	1	$0 < e < 255$	f	$-2^{e-127}(1,f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1,f)$
Positivo denormalizado	0	0	$f \neq 0$	$2^{e-126}(0,f)$	0	0	$f \neq 0$	$2^{e-1022}(0,f)$
Negativo denormalizado	1	0	$f \neq 0$	$-2^{e-126}(0,f)$	1	0	$f \neq 0$	$-2^{e-1022}(0,f)$

(0) y todo unos (255 en el formato simple, 2047 en el doble) definen valores especiales. Se representan las siguientes clases de números:

- Para valores de exponente desde 1 hasta 254 en el formato simple y desde 1 hasta 2046 en el formato doble, se representan números en coma flotante normalizados distintos de cero. El exponente está sesgado, siendo el rango de exponentes desde -126 hasta +127 en el formato simple y de (-1022) a +1023 en el doble. Un número normalizado debe contener un bit 1 a la izquierda de la coma binaria; este bit está implícito, dando una parte significativa efectiva de 24 bits o de 53 bits (denominada *fracción* o *parte fraccionaria* en el estándar).
- Un exponente cero junto con una fracción cero representa el cero positivo o el negativo, dependiendo del bit de signo. Como se mencionó, es útil tener una representación del valor 0 exacto.
- Un exponente todo unos junto con una parte fraccionaria cero representa, dependiendo del bit de signo, el infinito positivo o el negativo. Es también útil tener una representación de infinito. Esto deja al usuario decidir si trata el desbordamiento como error o si prosigue con él en el programa que se esté ejecutando.
- Un exponente cero junto con una parte fraccionaria distinta de cero representa un número denormalizado. En este caso, el bit a la izquierda de la coma binaria es cero y el exponente original es -126 ó -1022. El número es positivo o negativo dependiendo del bit de signo.
- A un exponente de todo unos junto con una fracción distinta de cero se le da el nombre de NaN, que viene de “*not a number*” (*no representa un número*), y se emplea para señalar varias condiciones de excepción.

El significado de los números denormalizados y de los NaN se discute en la Sección 9.5.

9.5. ARITMÉTICA EN COMA FLOTANTE

La Tabla 9.5 resume las operaciones básicas de la aritmética en coma flotante. En sumas y restas es necesario asegurar que ambos operandos tengan el mismo exponente. Esto puede requerir desplazar la coma de la base en uno de los operandos para conseguir alinearlos. La multiplicación y la división son más directas.

En una operación en coma flotante se puede producir alguna de estas condiciones:

- **Desbordamiento del exponente:** un exponente positivo que excede el valor de exponente máximo posible. En algunos sistemas, este desbordamiento puede designarse como $+\infty$ o $-\infty$.
- **Agotamiento del exponente:** un exponente negativo menor que el mínimo valor posible (ejemplo: -200 es menor que -127). Esto significa que el número es demasiado pequeño para ser representado, y que puede considerarse como 0 .
- **Agotamiento de la parte significativa:** en el proceso de alineación o ajuste pueden perderse dígitos por la derecha de la parte significativa. Como comentaremos, se requiere efectuar algún tipo de redondeo.
- **Desbordamiento de la parte significativa:** la suma de dos mantisas del mismo signo puede producir un acarreo procedente del bit más significativo. Esto, como se explicará, puede arreglarse con un reajuste.

SUMA Y RESTA

En la aritmética de coma flotante, la suma y la resta son más complejas que la multiplicación y la división. Esto es debido a la necesidad de ajustar las partes significativas. Hay cuatro etapas básicas del algoritmo para sumar o restar:

Tabla 9.5. Números y operaciones aritméticas en coma flotante.

Números en punto flotante	Operaciones aritméticas
$X = X_s \times B^{X_e}$ $Y = Y_s \times B^{Y_e}$	$\left. \begin{array}{l} X + Y = (X_s \times B^{X_e - Y_e} + Y_s) \times B^{Y_e} \\ X - Y = (X_s - B^{X_e - Y_e} - Y_s) \times B^{Y_e} \\ X \times y = (X_s \times Y_s) \times B^{X_e + Y_e} \\ \frac{X}{Y} = \left(\frac{X_s}{Y_s} \right) \times B^{X_e - Y_e} \end{array} \right\} X_e \leq Y_e$

Ejemplos:

$$X = 0,3 \times 10^2 = 30$$

$$Y = 0,2 \times 10^3 = 200$$

$$X + Y = (0,3 \times 10^{2-3} + 0,2) \times 10^3 = 0,23 \times 10^3 = 230$$

$$X - Y = (0,3 \times 10^{2-3} - 0,2) \times 10^3 = (-0,17) \times 10^3 = -170$$

$$X \times Y = (0,3 \times 0,2) \times 10^{2+3} = 0,06 \times 10^5 = 6.000$$

$$X \div Y = (0,3 \div 0,2) \times 10^{2-3} = 1,5 \times 10^{-1} = 0,15$$

1. Comprobar valores cero.
2. Ajuste de partes significativas.
3. Sumar o restar las partes significativas.
4. Normalizar el resultado.

Un diagrama de flujo típico se muestra en la Figura 9.22. En ella, una descripción paso a paso resalta las funciones principales requeridas para la suma y la resta en coma flotante. Se asume un formato similar al de la Figura 9.21. Para la operación de suma o de resta, los dos operandos deben transferirse a los registros que serán utilizados por la ALU. Si el formato en coma flotante incluye un bit implícito en la parte significativa, dicho bit debe hacerse explícito para la operación.

Fase 1: comprobación de cero. Dado que la suma y la resta son idénticas excepto por el cambio de signo, el proceso comienza cambiando el signo del substraendo cuando se trata de una resta. A continuación, si alguno de los operandos es cero, se da el otro como resultado.

Fase 2: ajuste de las partes significativas. La etapa siguiente manipula los números para que los dos exponentes sean iguales.

Para ver la necesidad de la fase de ajuste, considere la siguiente suma en decimal:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Claramente, no podemos sumar directamente sus partes significativas. Los dígitos deben ponerse primero en posiciones equivalentes, es decir, el 4 del segundo número debe alinearse con el 3 del primero. Bajo estas condiciones, los exponentes serían iguales, que es la condición matemática para que dichos números se puedan sumar. Así:

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4,56 \times 10^0) = 127,56 \times 10^0$$

La alineación o ajuste se consigue o bien desplazando a la derecha el número más pequeño (incrementando su exponente) o desplazando a la izquierda el más grande. Ya que cualquiera de dichas operaciones puede hacer que se pierdan dígitos, es el menor el que se desplaza; con lo que los dígitos que se pierdan tienen una importancia relativa pequeña. El ajuste se consigue repitiendo desplazamientos en un dígito a la derecha de la parte de magnitud de la mantisa e incrementando el exponente hasta igualarlo con el otro (observe que si la base implícita es 16, un desplazamiento de un dígito equivale a desplazar 4 bits.) Si este proceso lleva a que parte significativa se hace 0, se da como resultado el otro número. Así, cuando dos números tienen exponentes muy diferentes, se pierde el menor de los números.

Fase 3: suma. A continuación se suman las dos partes significativas, teniendo en cuenta sus signos. Ya que los signos pueden diferir, el resultado puede ser 0. Existe también la posibilidad de desbordamiento de la mantisa en un dígito. Si es así, se desplaza a la derecha la parte significativa del resultado, y se incrementa el exponente. Como resultado podría producirse un desbordamiento en el exponente; esto se debe informar y la operación se detendría.

Fase 4: normalización. La etapa final normaliza el resultado. La normalización consiste en desplazar a la izquierda los dígitos de la mantisa hasta que el más significativo (1 bit, o 4 bits para exponentes de base 16) sea distinto de cero. Cada desplazamiento causa un decremento del

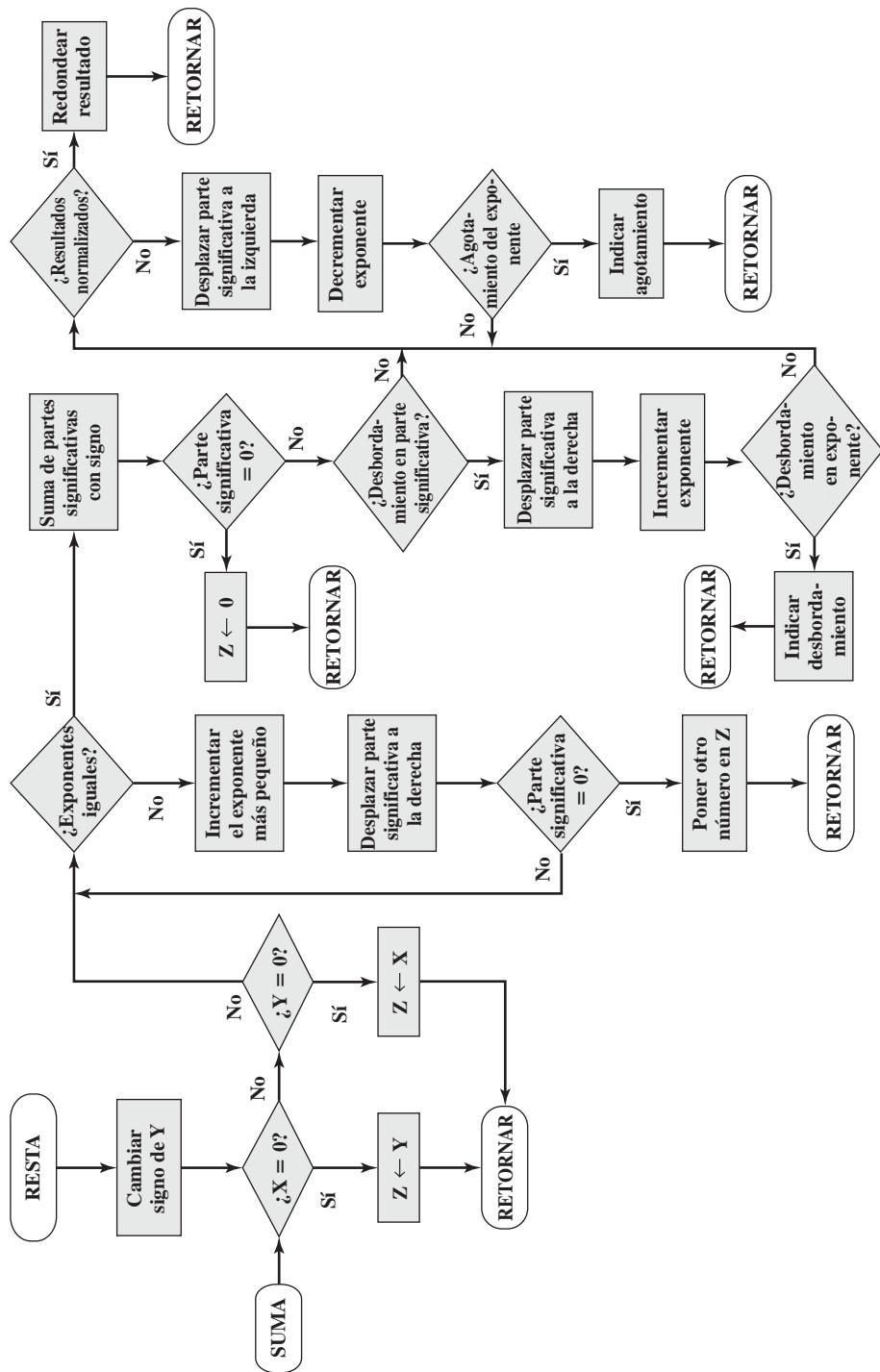


Figura 9.22. Suma y resta en coma flotante ($Z \leftarrow X \pm Y$).

exponente, lo que podría producir un desbordamiento a cero del mismo. Finalmente, el resultado debe redondearse y proporcionarse. Posponemos el tema del redondeo para después del estudio de la multiplicación y la división.

MULTIPLICACIÓN Y DIVISIÓN

La multiplicación y la división en coma flotante son procesos mucho más sencillos que la suma y la resta, como se indica en la discusión que sigue.

Consideremos primero la multiplicación, que se ilustra en la Figura 9.23. En primer lugar, si cualquiera de los operandos es 0, se indica como resultado 0. El siguiente paso es sumar los exponentes. Si los exponentes están almacenados en forma sesgada, su suma tendría un sesgo doble. Por ello debe restarse de la suma el valor de sesgo. El resultado podría dar lugar a un desbordamiento o a un desbordamiento a cero del exponente, lo que debe indicarse, y concluir el algoritmo.

Si el exponente del producto está dentro del rango apropiado, el paso siguiente es multiplicar las partes significativas teniendo en cuenta sus signos. Dicha multiplicación se realiza como en el caso de los enteros. En este caso estamos tratando con una representación en signo-magnitud, pero los

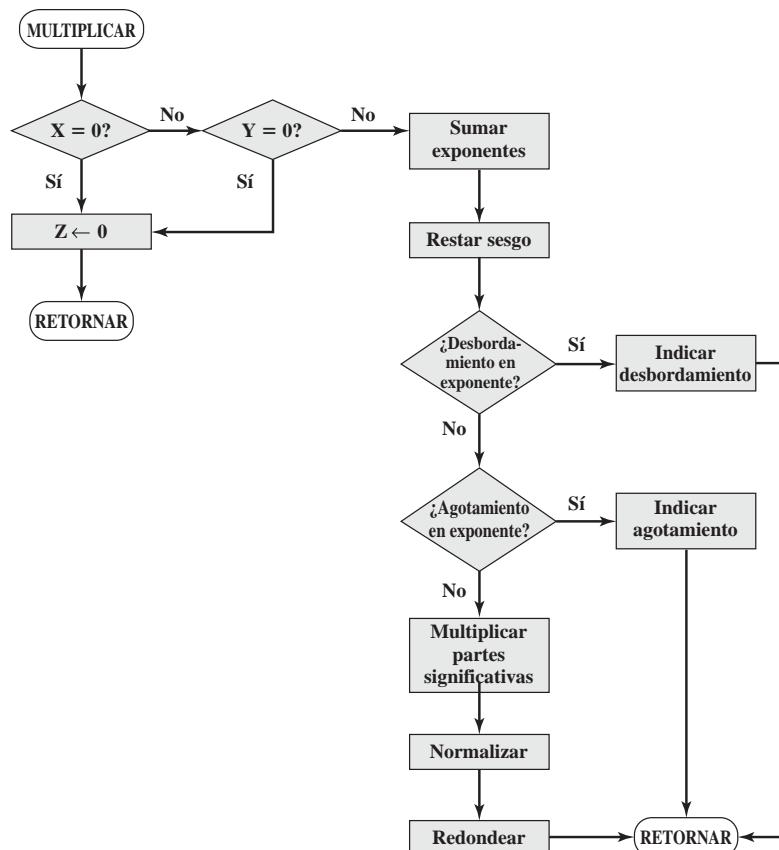


Figura 9.23. Multiplicación en coma flotante ($Z \leftarrow X \times Y$).

detalles son similares a los de representación en complemento a dos. El producto doblará la longitud del multiplicando y multiplicador. Los bits extra se perderán durante el redondeo.

Tras calcular el producto se normaliza y redondea el resultado, como se hizo para la suma y la resta. Observe que la normalización podría producir un desbordamiento a cero del exponente.

Consideremos finalmente el diagrama de flujo para la división mostrado en la Figura 9.24. De nuevo, el primer paso es comprobar ceros. Si el divisor es cero se da una indicación de error, o se pone el resultado a infinito, dependiendo de la implementación en cuestión. Un dividendo 0 da como resultado 0. A continuación el exponente del divisor se resta al del dividendo. Esto elimina el sesgo, que debe añadirse de nuevo. Se comprueban entonces posibles desbordamientos (a cero o no) del exponente.

El siguiente paso es dividir las partes significativas. A este paso sigue la normalización y redondeo usuales.

CONSIDERACIONES SOBRE PRECISIÓN

Bits de guarda. Hemos mencionado que, previo a una operación en coma flotante, se cargan el exponente y la parte significativa en registros de la ALU. En el caso de la parte significativa, el

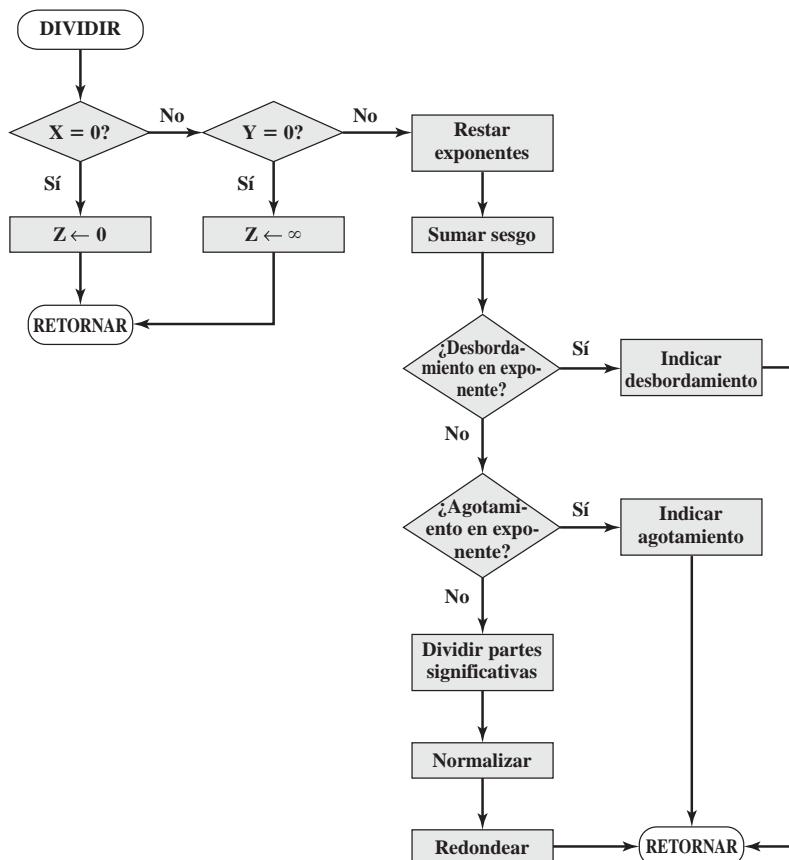


Figura 9.24. División en coma flotante ($Z \leftarrow X/Y$).

tamaño del registro es casi siempre mayor que la longitud de la parte significativa más el bit. El registro contiene bits adicionales, llamados bits de guarda o de respaldo, que se añaden a la derecha de la parte significativa en forma de ceros.

La razón de utilizar estos bits se ilustra en la Figura 9.25. Considere números en el formato IEEE, que tiene una parte significativa de 24 bits, incluyendo un 1 implícito a la izquierda de la coma binaria. Dos números de valor muy próximo son $X = 1,00\dots00 \times 2^1$ e $Y = 1,11\dots11 \times 2^0$. Para restar del mayor el menor de ellos, este debe desplazarse un bit a la derecha para igualar los exponentes. Esto se muestra en la Figura 9.25a. En este proceso Y pierde 1 bit significativo; el resultado que se obtiene es 2^{-22} . En la parte b de la figura se repite la misma operación pero con bits de guarda añadidos. Ahora el bit menos significativo no se pierde al alinear, y el resultado es 2^{-23} , diferente en un factor de 2 respecto del anterior resultado. Cuando la base es 16, la pérdida de precisión puede ser mayor. Como muestran las Figuras 9.25c y 9.25d, la diferencia puede ser de un factor 16.

Redondeo. Otro detalle que afecta a la precisión del resultado es la política de redondeo empleada. El resultado de cualquier operación sobre las partes significativas se almacena generalmente en un registro más grande. Cuando el resultado se pone de nuevo en el formato de coma flotante, hay que deshacerse de los bits extra.

Se han explorado diversas técnicas para realizar el redondeo. De hecho, el estándar del IEEE enumera cuatro aproximaciones alternativas:

- **Redondeo al más próximo:** el resultado se redondea al número representable más próximo.
- **Redondeo hacia $+\infty$:** el resultado se redondea por exceso hacia más infinito.
- **Redondeo hacia $-\infty$:** el resultado se redondea por defecto hacia menos infinito.
- **Redondeo hacia 0:** el resultado se redondea hacia cero.

Consideremos cada uno de los casos anteriores. El **redondeo al más próximo** es el modo implícito contemplado en el estándar y se define como sigue: debe tomarse el valor representable más próximo al resultado exacto.

$\begin{aligned} x &= 1,000\dots00 \times 2^1 \\ -y &= 0,111\dots11 \times 2^1 \\ z &= 0,000\dots01 \times 2^1 \\ &= 1,000\dots00 \times 2^{-22} \end{aligned}$	$\begin{aligned} x &= ,100000 \times 16^1 \\ -y &= ,0FFFFF \times 16^1 \\ z &= ,000001 \times 16^1 \\ &= ,100000 \times 16^{-4} \end{aligned}$
(a) Ejemplo binario, sin bits de guarda	(c) Ejemplo hexadecimal, sin bits de guarda

$\begin{aligned} x &= 1,000\dots00\ 0000 \times 2^1 \\ -y &= 0,111\dots11\ 1000 \times 2^1 \\ z &= 0,000\dots00\ 1000 \times 2^1 \\ &= 1,000\dots00\ 0000 \times 2^{-23} \end{aligned}$	$\begin{aligned} x &= ,100000\ 00 \times 16^1 \\ -y &= ,0FFFFF\ F0 \times 16^1 \\ z &= ,000000\ 10 \times 16^1 \\ &= ,100000\ 00 \times 16^{-5} \end{aligned}$
(b) Ejemplo binario, con bits de guarda	(d) Ejemplo hexadecimal, con bits de guarda

Figura 9.25. Utilización de bits de guarda.

Si los bits extra, además de los 23 bits que pueden almacenarse, son 10010, entonces la cantidad indicada por los bits extra supera la mitad del valor correspondiente a la última posición de bit representable. En este caso, la respuesta correcta es sumar 1 al último bit representable, redondeando por exceso al número siguiente representable. Consideré ahora que los bits extra valen 01111. En este caso representan una cantidad menor que la mitad de la última posición de bit representable. La respuesta correcta es simplemente ignorar los bits extra (truncar), lo que tiene como efecto un redondeo por defecto al número representable precedente.

El estándar también contempla el caso especial de bits extra en la forma 10000..., en que el resultado está exactamente en la mitad de los dos valores representables posibles. Una posible solución sería truncar siempre, que es la operación más sencilla. Sin embargo, la dificultad de esta aproximación simple es que introduce un sesgo pequeño, pero acumulativo, en una secuencia de cálculos. Se necesita un método de redondeo que no introduzca esta tendencia de los resultados. Una posibilidad sería redondear por exceso o por defecto basándose en un número aleatorio, de manera que en promedio el resultado no sería sesgado. Un argumento en contra de esta aproximación es que no produciría resultados predecibles deterministas. La aproximación tomada en la norma del IEEE es forzar que el resultado sea par: si el resultado de un cálculo está exactamente en la mitad de dos números representables, el valor se redondea por exceso cuando el último bit representable actual es 1, y se deja como está si es 0.

Las dos siguientes opciones, **redondeo a más o a menos infinito**, son útiles en la implementación de una técnica conocida como aritmética de intervalos. La aritmética de intervalos proporciona un método eficiente para monitorizar y controlar errores en los cálculos en coma flotante, produciendo dos valores por cada resultado. Los dos valores se corresponden con los límites inferior y superior de un intervalo que contiene el resultado exacto. La longitud del intervalo, que es la diferencia entre los límites superior e inferior, indica la precisión del resultado. Si los extremos del intervalo no son representables, se redondean por defecto y por exceso, respectivamente. Aunque la anchura del intervalo puede variar de unas implementaciones a otras, se han diseñado diversos algoritmos al objeto de conseguir intervalos estrechos. Si el rango de variación entre los límites superior e inferior es suficientemente estrecho se obtiene un resultado bastante preciso. Si no, al menos lo sabemos y podemos realizar análisis adicionales.

La última de las técnicas especificadas en el estándar es el **redondeo hacia cero**. Consiste de hecho en un simple truncamiento: se ignoran los bits extra. Esta es ciertamente la técnica más sencilla. Sin embargo, el resultado es que la magnitud del valor truncado es siempre menor o igual que el valor original más exacto, introduciéndose un sesgo hacia cero en la operación. Este sesgo o tendencia es serio, ya que afecta a todas las operaciones para las que haya algún bit extra distinto de cero.

ESTÁNDAR IEEE PARA LA ARITMÉTICA BINARIA EN COMA FLOTANTE

El IEEE 754 va más allá de la simple definición de un formato, detallando cuestiones prácticas específicas y procedimientos para que la aritmética en coma flotante produzca resultados uniformes y predecibles, independientemente de la plataforma hardware. Uno de estos aspectos ha sido ya discutido, el redondeo. En esta subsección se ven otros tres aspectos: el infinito, los NaN, y los números sin normalizar o «denormalizados».

Infinito. Las operaciones aritméticas con infinito son tratadas como casos límite de la aritmética real, dándose la siguiente interpretación a los valores de infinito:

$$-\infty < (\text{todo número finito}) < +\infty$$

Exceptuando los casos especiales discutidos posteriormente, cualquier operación aritmética que involucre al infinito produce el resultado obvio conocido.

Por ejemplo:

$5 + (+\infty) = +\infty$	$5 \div (+\infty) = +0$
$5 - (+\infty) = -\infty$	$(+\infty) + (+\infty) = +\infty$
$5 + (-\infty) = -\infty$	$(-\infty) + (-\infty) = -\infty$
$5 - (-\infty) = +\infty$	$(-\infty) - (+\infty) = -\infty$
$5 \times (+\infty) = +\infty$	$(+\infty) - (-\infty) = +\infty$

Nan indicadores y silenciosos. Un NaN es una entidad simbólica codificada en formato de coma flotante, del que hay dos tipos: indicador y silencioso. Un NaN indicador señala una condición de operación no válida siempre que aparece como operando. Los NaN indicadores permiten representar valores de variables no inicializadas y tratamientos de tipo aritmético que no están contemplados en el estándar. Un NaN silencioso se propaga en la mayoría de las operaciones sin señalar excepción alguna. La Tabla 9.6 indica operaciones que producirían un NaN silencioso.

Obsérvese que ambos tipos de NaN tienen el mismo formato general (Tabla 9.4): un exponente con todo unos y una parte fraccionaria distinta de cero. El patrón de bits real de dicha fracción depende de cada implementación concreta; sus valores pueden utilizarse para distinguir entre NaN indicadores y silenciosos, y para especificar condiciones de excepción particulares.

Tabla 9.6. Operaciones que producen un NaN silencioso.

Operación	NaN silencioso producido por
Cualquiera	Cualquier operación con un NaN indicador
Suma o resta	Restas con infinito $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiplicación	$0 \times \infty$
División	$\frac{0}{0}$ ó $\frac{\infty}{\infty}$
Resto	$x \text{ RE } 0$ ó $\infty \text{ RE } y$
Raíz cuadrada	\sqrt{x} donde $x < 0$

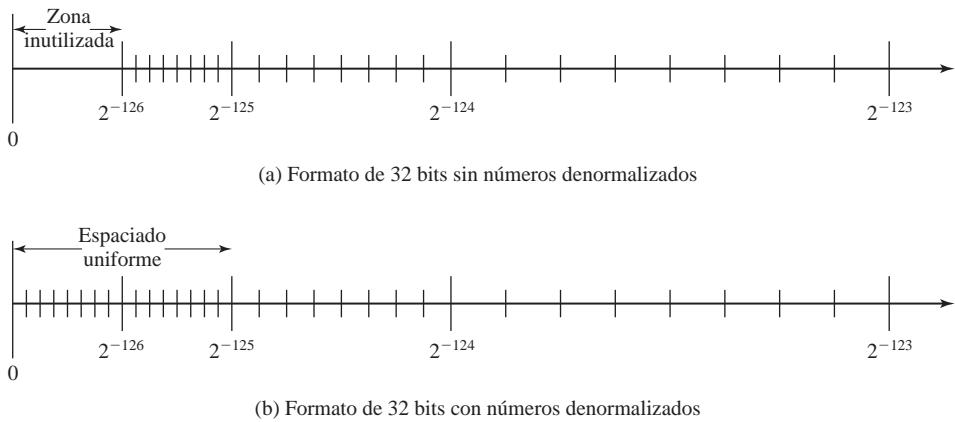


Figura 9.26. Efecto de los números denormalizados del IEEE 754.

Números denormalizados. Los números denormalizados se incluyen en el IEEE 754 para reducir las situaciones de desbordamiento hacia cero de exponentes. Cuando el exponente del resultado es demasiado pequeño (un exponente negativo con magnitud muy grande), el resultado se denormaliza desplazando a la derecha la parte fraccionaria e incrementando el exponente, a cada desplazamiento, hasta que dicho exponente esté dentro de un rango representable.

La Figura 9.26 ilustra el efecto que tiene añadir los números denormalizados. Los números que son representables pueden agruparse en intervalos de la forma $[2^n, 2^{n+1}]$. Dentro de cada intervalo, la parte de exponente del número es la misma y varía la parte fraccionaria, produciéndose un espaciado uniforme de los números representables en el intervalo. A medida que nos aproximamos a cero, cada intervalo sucesivo es la mitad de ancho que el precedente pero contiene la misma cantidad de números representables. Por tanto, la densidad de números representables aumenta a medida que nos aproximamos a cero. Sin embargo, si solo se emplean números normalizados, hay una zona inutilizada entre cero y el menor de los números normalizados. En el caso del IEEE 754 hay 2^{23} números representables en cada intervalo, y el número positivo más pequeño representable es 2^{-126} . Al tener en cuenta también los números renormalizados, se añaden 2^{23} números uniformemente distribuidos entre 0 y 2^{-126} .

El uso de números denormalizados se denomina *desbordamiento hacia cero gradual* [COON81]. Sin números denormalizados, la zona entre cero y el número distinto de cero más pequeño representable es mucho más grande que la zona entre dicho número y el que le sigue. El desbordamiento a cero gradual cubre esta zona y reduce el efecto de desbordamiento a cero del exponente hasta un nivel comparable con el redondeo de números normalizados.

9.6. LECTURAS Y SITIOS WEB RECOMENDADOS

[ERCE04] y [PARH00] son excelentes tratados sobre la aritmética de los computadores, cubriendo con detalle todos los aspectos discutidos en este capítulo. [FLYN01] proporciona una discusión útil centrada en aspectos prácticos de diseño e implementación. Para un estudio serio de la aritmética del computador, una referencia muy útil es el trabajo de dos volúmenes [SWAR90]. El Volumen I fue inicialmente publicado en 1980 y contiene artículos clave (algunos de ellos difíciles de encontrar por otras vías) sobre fundamentos de aritmética del