Capítulo 4

Representación de datos numéricos

Los datos de tipo numérico pueden codificarse en binario siguiendo distintos métodos. Los más usuales están basados en uno de dos principios: o bien codificar cada dígito decimal mediante un código BCD (de modo que un número de n dígitos decimales se codifique con n cuartetos, apartado 3.1), o bien representar el número en el sistema de numeración de base 2. En cualquier caso, hay que añadir convenios para representar números negativos y números no enteros.

La codificación en BCD actualmente sólo se usa en dispositivos especializados (relojes, calculadoras, etc.).

Por otra parte, se pueden representar números con *precisión arbitraria*, utilizando cuantos bits sean necesarios (con la limitación de la capacidad de memoria o del caudal del canal de transmisión disponibles, por eso no es «precisión infinita»). Hay lenguajes y bibliotecas de programas que permiten representar de este modo y realizar operaciones aritméticas *(«bignum arithmetic»)* para aplicaciones especiales, pero tampoco nos ocuparemos de este tipo de representación.

Nos vamos a centrar en los convenios normalmente adoptados en los diseños de los procesadores de uso general. Se basan en reservar un número fijo de bits (normalmente, una palabra, o un número reducido de palabras) para representar números enteros y racionales con una precisión y un rango (números máximo y mínimo) determinados por el número de bits. La figura 4.1 generaliza a n bits otra que ya habíamos visto (figura 1.1). Como ya dijimos, el bit 0 (o «bit de peso cero») es el *bit menos significativo, «bms»* (o «LSB»: Least Significant Bit) y el de peso $n-1$ es el *bit más significativo, «bMs»* (o «MSB»: Most Significant Bit).

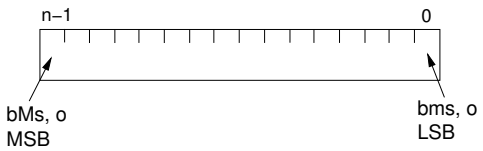

Figura 4.1: «Contenedor» de n bits.

4.1. Números enteros sin signo

Algunos datos son de tipo entero no negativo, como las direcciones de la memoria, o el número de segundos transcurridos a partir de un instante dado. Su representación es, simplemente, su expresión en binario. Con n bits el máximo número representable es $0b111\ldots1 = 2^n - 1$ («0b» indica que lo que sigue está en el sistema de numeración de base 2). El *rango* de la representación es $[0, 2^n - 1]$.

Por ejemplo, para expresar las direcciones de una memoria de 1 MiB = 2^{20} bytes necesitamos $n = 20$ bits. La primera dirección será 0x00000 y la más alta 0xFFFFF = $2^{20} - 1$. Otro ejemplo: las direcciones IPv4 (Internet Protocol versión 4) son enteros sin signo de 32 bits. La más pequeña es 0x00000000 (0.0.0.0) y la más grande 0xFFFFFFFF (255.255.255.255). En total, 2^{32} = 4.294.967.296 direcciones.

4.2. Formatos de coma fija

En el contexto de la representación de números, un **formato** es la definición del significado de cada uno de los n bits dedicados a la representación. Se distinguen en él partes o **campos**, grupos de bits con significado propio. Los formatos más utilizados para la representación de datos numéricos son los de **coma fija** y **coma flotante** (o «punto fijo» y «punto flotante»).

Los formatos de coma fija sólo tienen dos campos. Uno es el bit más significativo o **bit de signo**, bMs = S: si S = 0, el número representado es positivo, y si S = 1, es negativo. El campo formado por los $n - 1$ bits restantes representa la magnitud del número. Se siguen distintos convenios sobre cómo se codifica esta magnitud.

4.3. Números enteros

Para un número entero no tiene sentido hablar de «coma». En todo caso, esta «coma» estaría situada inmediatamente a la derecha del bms.

Como el primer bit es el de signo, el máximo número representable (positivo) es $0b0111\ldots11 = 2^{n-1} - 1$. El mínimo depende del convenio para los números negativos.

Convenios para números negativos

Hay tres convenios:

- **signo y magnitud** (o **signo y módulo**): después del signo (S = 1), el valor absoluto;
- **complemento a 1**: se cambian los «ceros» por «unos» y los «unos» por «ceros» en la representación binaria del correspondiente número positivo;
- **complemento a 2**: se suma una unidad al complemento a 1.

Por ejemplo, el número –58 = –0x3A = –0b111010, con $n = 16$ bits, y según el convenio que se adopte, da lugar a las siguientes representaciones:

- con signo y módulo: 1000000000111010 (0x803A)
- con complemento a 1: 1111111111000101 (0xFFC5)
- con complemento a 2: 1111111111000110 (0xFFC6)

Es fácil comprobar que si X es la representación de un número con n bits, incluido el de signo, y si $X1$ y $X2$ son las representaciones en complemento a 1 y complemento a 2, respectivamente, del mismo número cambiado de signo, entonces $X + X1 = 2^n - 1$, y $X + X2 = 2^n$. (Por tanto, en rigor, lo que habitualmente llamamos «complemento a 2» es «complemento a 2^n», y «complemento a 1» es «complemento a $2^n - 1$»).

La tabla 4.1 muestra una comparación de los tres convenios para n bits. Como puede usted observar, en signo y magnitud y en complemento a 1 existen dos representaciones para el número 0, mientras que en complemento a 2 la representación es única (y hay un número negativo más, cuyo valor absoluto no puede representarse con n bits). El *rango*, o *extensión* de la representación (diferencia entre el número máximo y el mínimo representables) es una función exponencial de n.

En lo sucesivo, y salvo en alguna nota marginal, *supondremos siempre que el convenio es el de complemento a 2*, que es el más común.

	configuraciones binarias	interpretaciones decimales		
		signo y magnitud	complemento a 1	complemento a 2
números positivos	0000…0000	0	0	0
	0000…0001	1	1	1
	…	…	…	…
	0111…1110	$2^{n-1}-2$	$2^{n-1}-2$	$2^{n-1}-2$
	0111…1111	$2^{n-1}-1$	$2^{n-1}-1$	$2^{n-1}-1$
números negativos	1000…0000	0	$-(2^{n-1}-1)$	$-(2^{n-1})$
	1000…0001	-1	$-(2^{n-1}-2)$	$-(2^{n-1}-1)$
	…	…	…	…
	1111…1110	$-(2^{n-1}-2)$	-1	-2
	1111…1111	$-(2^{n-1}-1)$	0	-1

Tabla 4.1: Comparación de convenios para formatos de coma fija.

4.4. Operaciones básicas de procesamiento

En este capítulo estamos estudiando los convenios de representación de varios tipos de datos. Es posible que, a veces, alguno de esos convenios le parezca a usted arbitrario. Por ejemplo, ¿por qué representar en complemento los números negativos, cuando parece más «natural» el hacerlo con signo y módulo?

Pero tenga en cuenta que los convenios no se adoptan de forma caprichosa, sino guiada por el tipo de operaciones que se realizan sobre los datos y por los algoritmos o por los circuitos electrónicos que implementan estas operaciones en software o en hardware. Aunque nos desviemos algo del contenido estricto del tema («representación»), éste es un buen momento para detenernos en algunas operaciones básicas de *procesamiento*.

Para no perdernos con largas cadenas de bits pondremos ejemplos muy simplificados, en el sentido de un número de bits muy reducido: $n = 6$. Esto daría un rango de representación para números enteros sin signo de $[0, 63]$ y un rango para enteros de $[-32, 31]$, lo que sería insuficiente para la mayoría de las aplicaciones prácticas, pero es más que suficiente para ilustrar las ideas.

Suma y resta de enteros sin signo

La suma binaria se realiza igual que la decimal: primero se suman los bits de peso 0, luego los de peso 1 más el eventual acarreo (que se habrá producido en el caso de 1+1=10), etc. Por ejemplo:

```
  13    001101   (0x0D)
+39   +100111  (+0x27)
  52    110100   (0x34)
```

Observe que se puede producir acarreo en todos los bits salvo en el más significativo, porque eso indica que el resultado no puede representarse en el rango, y se dice que hay un **desbordamiento**. Por ejemplo:

```
  60     111100   (0x3C)
+15    +001111  (+0x0F)
  75  1) 001011   (0x4B, pero el máximo es 0x3F: desbordamiento)
```

Para la resta también se podría aplicar un algoritmo similar al que conocemos para la base 10,

pero para el diseño de circuitos es más fácil trabajar con complementos. Como estamos tratando de enteros *sin signo* con seis bits, no tiene sentido el complemento a 2^6, pero si sumamos al minuendo el complemento a 2^7 del sustraendo obtenemos el resultado correcto. Por ejemplo:

$$\begin{array}{rrl} 60 & 111100 & (0x3C) \\ \underline{-15} & \underline{+1)110001} & (\underline{+0x71}) \text{ (complemento a } 2^7\text{de 15)} \\ 45 & 10)\ 101101 & (0x2D, \text{ olvidando el acarreo)} \end{array}$$

Fíjese que ahora *siempre hay un acarreo* de los bits más significativos que hay que despreciar, incluso si el resultado es 0 (puede comprobarlo fácilmente).

Parece que no tiene mucho sentido preguntarse por el caso de que el sustraendo sea mayor que el minuendo, ya que estamos tratando de números positivos, pero a veces lo que interesa es *comparar* dos números: saber si X es mayor, igual o menor que Y, y esto se puede averiguar sumando a X el complemento de Y. Por ejemplo:

$$\begin{array}{rrl} 15 & 001111 & (0x0F) \\ \underline{-60} & \underline{+1)000100} & (+0x44) \text{ (complemento a } 2^7\text{de 60)} \\ & 010011 & \end{array}$$

Lo importante ahora no es el resultado, obviamente incorrecto, sino el hecho de que *no hay acarreo de los bits más significativos*.

Conclusión: para comparar X e Y, representados sin signo en n bits, sumar a X el complemento a 2^{n+1} de Y. Si se produce acarreo de los bits más significativos, $X \geq Y$; en caso contrario, $X < Y$.

Suma y resta de enteros

La representación de los números negativos por su complemento conduce a algoritmos más fáciles para la suma y resta que en el caso de signo y módulo (y, consecuentemente, a circuitos electrónicos más sencillos para su implementación).

Como hemos visto antes, si X es un entero positivo y $X2$ es su complemento a 2, $X + X2 = 2^n$.

Por ejemplo:
$$\begin{array}{rrl} 7 & 000111 & \\ \underline{+(-7)} & \underline{+111001} & \\ 0 & 1000000 & (2^6) \end{array}$$

Veamos cómo se aplica esta propiedad a la resta. Sean X e Y dos números positivos. La diferencia $D = X - Y$ se puede escribir así:
$D = X - Y = X - (2^n - Y2) = X + Y2 - 2^n$
donde $Y2$ es el complemento a 2 de Y.

Analicemos dos posibilidades: o bien $X + Y2 \geq 2^n$ ($D \geq 0$), o bien $X + Y2 < 2^n$ ($D < 0$).

- Si $D \geq 0$, restar 2^n de $X + Y2$ es lo mismo que quitar el bit de peso n (acarreo de los bits de signo).
- Si $D < 0$ la representación de D en complemento a 2 será $D2 = 2^n - (-D) = X + Y2$.

O sea, la suma binaria $X+Y2$, olvidando el posible acarreo de los bits de signo, nos va a dar siempre el resultado correcto de $X - Y$.

¿Y si el minuendo es negativo? La operación $L = -X - Y$ (siendo X e Y positivos) se escribe así en función de los complementos:

$L = -X - Y = -(2^n - X2) - (2^n - Y2) = X2 + Y2 - 2^{n+1}$

El resultado ha de ser siempre negativo (a menos, como veremos luego, que haya desbordamiento), es decir, $X2 + Y2 < 2^{n+1}$. La representación en complemento a 2 de este resultado es:

$L2 = 2^n - (-L) = X2 + Y2 - 2^n$.

En conclusión, *para restar dos números basta sumar al minuendo el complemento a 2 del sustraendo y no tener en cuenta el eventual acarreo de los bits de signo*[1]. Ejemplos:

- $\begin{array}{rrl} 25 & 011001 & \\ \underline{-7} & \underline{+111001} & \\ 18 & 1010010 & \rightsquigarrow \quad 010010 \text{ (representación de 18)} \end{array}$
- $\begin{array}{rrl} 14 & 001110 & \\ \underline{-30} & \underline{+100010} & \\ -16 & 110000 & \quad (-16 \text{ en complemento a 2}) \end{array}$
- $\begin{array}{rrl} -10 & 110110 & \\ \underline{-18} & \underline{+101110} & \\ -28 & 1100100 & \rightsquigarrow \quad 100100 \ (-28 \text{ en complemento a 2}) \end{array}$

Desbordamiento

En todos los ejemplos anteriores (salvo en uno) hemos supuesto que tanto los operandos como el resultado estaban dentro del rango de números representables, que en el formato de los ejemplos es $-32 \leq X \leq 31$. Si no es así, las representaciones resultantes son erróneas. Por ejemplo:

- $\begin{array}{rrl} 15 & 001111 & \\ \underline{+20} & \underline{+010100} & \\ 35 & 100011 & \quad (-29 \text{ en complemento a 2}) \end{array}$
- $\begin{array}{rrl} -13 & 110011 & \\ \underline{-27} & \underline{+100101} & \\ -40 & 1011000 & \rightsquigarrow \quad 011000 \ (+24) \end{array}$

Este **desbordamiento** sólo se produce cuando los dos operandos son del mismo signo (se supone, por supuesto, que ambos están dentro del rango representable). Y su detección es muy fácil: cuando lo hay, el resultado es de signo distinto al de los operandos[2].

Acarreo

A veces el bit de signo no es un bit de signo, ni el desbordamiento indica desbordamiento. Veamos cuáles son esas «veces».

Supongamos que tenemos que sumar 1075 (0x433) y 933 (0x3A5). El resultado debe ser 2008 (0x7D8). Para hacerlo en binario necesitamos una extensión de al menos doce bits:

$$\begin{array}{rcl} 0x433 & \rightsquigarrow & 010000110011 \\ 0x3A5 & \rightsquigarrow & \underline{001110100101} \\ & & 011111011000 \quad \rightsquigarrow \quad 0x7D8 \end{array}$$

Pero supongamos también que nuestro procesador solamente permite sumar con una extensión de seis bits. Podemos resolver el problema en tres pasos:

[1] Si se entretiene usted en hacer un análisis similar para el caso de complemento a 1, descubrirá que se hace necesario un paso más: el eventual acarreo de los bits de signo no se debe despreciar, sino sumar a los bits de peso 0 para que el resultado sea correcto.

[2] Una condición equivalente (y más fácil de comprobar en los circuitos, aunque su enunciado sea más largo) es que el acarreo de la suma de los bits de pesos inmediatamente inferior al de signo es diferente al acarreo de los bits de signo (el que se descarta).

1. Sumar los seis bits menos significativos como si fuesen números sin signo:
   $$\begin{array}{r} 110011 \\ +100101 \\ \hline 1\,011000 \end{array}$$ (los seis bits menos significativos del resultado)
   Obtenemos un resultado en seis bits y un acarreo de los bits más significativos. Observe que esta suma es justamente la que provocaba desbordamiento en el ejemplo anterior de $-13-27$, pero ahora no estamos interpretando esos seis bits como la representación de un número entero con signo: los bits más significativos no son «bits de signo».
2. Sumar los seis bits más significativos:
   $$\begin{array}{r} 010000 \\ +001110 \\ \hline 011110 \end{array}$$
3. Al último resultado, sumarle el acarreo de los seis menos significativos:
   $$\begin{array}{r} 011110 \\ +1 \\ \hline 011111 \end{array}$$ (los seis bits más significativos del resultado)

El ejemplo ha tratado de la suma de dos números positivos, pero el principio funciona exactamente igual con números negativos representados en complemento.

Veremos en el capítulo 9 que los procesadores suelen tener dos instrucciones de suma: suma «normal» (que suma operandos de, por ejemplo, 32 bits) y suma con acarreo, que suma también dos operandos, pero añadiendo el eventual acarreo de la suma anterior.

Indicadores

Los circuitos de los procesadores (unidades aritmética y lógica y de desplazamiento) incluyen indicadores de un bit (materializados como biestables, apartado 1.3) que toman el valor 0 o 1 dependiendo del resultado de la operación. Normalmente son cuatro:

C: acarreo. Toma el valor 1 si, como acabamos de ver, se produce acarreo en la suma sin signo. Y, como vimos antes, en la resta sin signo $X - Y$ también se pone a 1 si $X \geq Y$.

V: desbordamiento. Toma el valor 1 si se produce un desbordamiento en la suma o en la resta con signo.

N: negativo. Coincide con el bit más significativo del resultado (S). Si no se ha producido desbordamiento, N = 1 indica que el resultado es negativo.

Z: cero. Toma el valor 1 si todos los bits del resultado son 0.

N y V, conjuntamente, permiten detectar el mayor de dos enteros *con signo* al hacer la resta $X - Y$:

- $X \geq Y$ si N = V, o sea, si N = 0 y V = 0 o si N = 1 y V = 1
- $X < Y$ si N $\neq$ V, o sea, si N = 0 y V = 1 o si N = 1 y V = 0

Operaciones lógicas

Las principales operaciones lógicas son la *complementación* o *negación* (**NOT**), el *producto lógico* (**AND**), la *suma lógica* (**OR**), la *suma módulo 2* u *or excluyente* (**EOR**), el *producto negado* (**NAND**) y la *suma negada* (**NOR**). Se trata de las mismas operaciones del álgebra de Boole, aplicadas en paralelo a un conjunto de bits. Aunque usted ya las debe conocer por la asignatura «Sistemas electrónicos», las resumimos brevemente.

La primera se aplica a un solo operando y consiste, simplemente, en cambiar todos sus «ceros» por «unos» y viceversa. Si el operando se interpreta como la representación de un número entero, el resultado es su complemento a 1. Por ejemplo: NOT(01101011) = 10010100

Las otras cinco se aplican, bit a bit, sobre dos operandos, y se definen como indica la tabla 4.2, donde en las dos primeras columnas se encuentran los cuatro valores posibles de dos operandos de un bit, y en las restantes se indican los resultados correspondientes a cada operación.

op1	op2	AND	OR	EOR	NAND	NOR
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

Tabla 4.2: Operaciones lógicas sobre dos operandos.

El **enmascaramiento** consiste en realizar la operación *AND* entre un dato binario y un conjunto de bits prefijado (**máscara**) para poner a cero determinados bits (y dejar los demás con el valor que tengan). Así, para extraer los cuatro bits menos significativos de un byte utilizaremos la máscara 0x0F = 00001111, y si la información original es «abcdefgh» (a, b,... ,h $\in \{0,1\}$) el resultado será «0000efgh».

Operaciones de desplazamiento

Los desplazamientos, lo mismo que las operaciones lógicas, se realizan sobre un conjunto de bits, normalmente, una palabra. Hay dos tipos básicos: *desplazamientos a la derecha*, en los que el bit *i* del operando pasa a ocupar la posición del bit $i-1$, éste la del $i-2$, etc., y *desplazamientos a la izquierda*, en los que se procede al revés. Y hay varios subtipos (figura 4.2), dependiendo de lo que se haga con el bit que está más a la derecha (bms) y con el que está más a la izquierda (bMs):

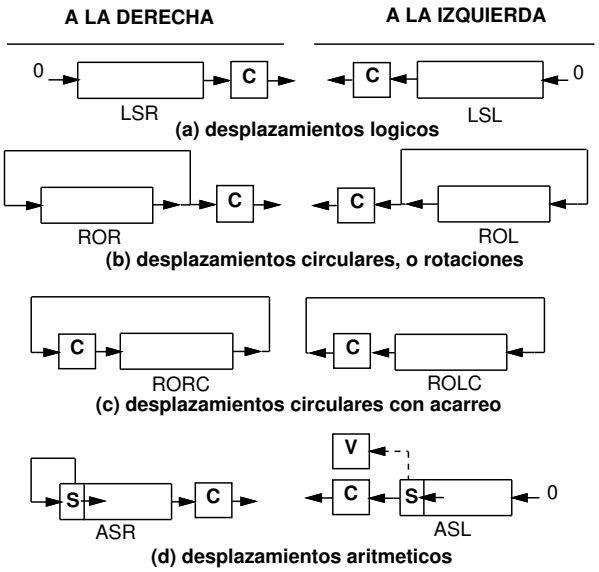

Figura 4.2: Operaciones de desplazamiento.

Desplazamientos lógicos

En un desplazamiento lógico a la derecha el bms se introduce en el indicador C (perdiéndose el valor anterior que éste tuviese), y por la izquierda se introduce un «0» en el bMs. Justamente lo contrario se hace en el desplazamiento lógico a la izquierda: el bMs se lleva a C y se introduce un «0» por la derecha, como ilustra la figura 4.2(a).

Las siglas que aparecen en la figura son nombres nemónicos en inglés: `LSR` es «Logical Shift Right» y `LSL` es «Logical Shift Left».

Desplazamientos circulares

En los desplazamientos circulares, también llamados **rotaciones**, el bit que sale por un extremo se introduce por el otro, como indica la figura 4.2(b). Los nemónicos son `ROR` («Rotate Right») y `ROL` («Rotate Left»).

Las rotaciones a través del indicador C funcionan como muestra la figura 4.2(c).

Desplazamientos aritméticos

En el desplazamiento aritmético a la izquierda, el bMs se introduce en el indicador de acarreo (C), y en el desplazamiento aritmético a la derecha se propaga el bit de signo. La figura 4.2(d) ilustra este tipo de desplazamiento, muy útil para los algoritmos de multiplicación. Es fácil comprobar que un desplazamiento aritmético de un bit a la derecha de un entero con signo conduce a la representación de la división (entera) por 2 de ese número. Y que el desplazamiento a la izquierda hace una multiplicación por 2.

Los nemónicos son `ASR` («Arithmetic Shift Right») y `ASL` («Arithmetic Shift Left»).

Aparentemente, el desplazamiento aritmético a la izquierda tiene el mismo efecto que el desplazamiento lógico a la izquierda, pero hay una diferencia sutil: el primero pone un «1» en el indicador de desbordamiento (V) si como consecuencia del desplazamiento cambia el bit de signo (S).

4.5. Números racionales

Volvamos al asunto principal de este capítulo, que es la representación de datos numéricos.

Para la representación de números racionales con una cantidad finita de bits hemos de tener en cuenta que no sólo habrá una **extensión** finita (un número máximo y un número mínimo), sino también una **resolución** (diferencia entre dos representaciones consecutivas) finita, es decir, hay una *discretización* del espacio continuo de los números reales.

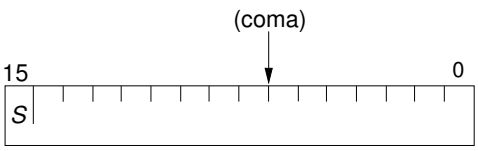

Figura 4.3: Formato de 16 bits para representación en coma fija de números racionales.

En principio, con un formato de coma fija podemos representar también números racionales: basta con añadir el convenio sobre «dónde está la coma». Si, por ejemplo, tenemos una longitud de dieciséis bits, podríamos convenir que los ocho que siguen al bit de signo representan la parte entera y los siete restantes la parte fraccionaria (figura 4.3).

En este ejemplo, el máximo número representable es 0b0 1111 1111,1111 111 = +0xFF,FE = +255,9921875 y el mínimo, suponiendo complemento a 2, es el representado por 1000000000000000, que corresponde a –0b1 0000 0000,0000 000 = –0x100 = –256. La resolución es: 0b0,0000 001 = 0x0,02 = 0,0078125.

En general, si tenemos f bits para la parte fraccionaria, estamos aplicando un factor de escala $1/2^f$ con respecto al número entero que resultaría de considerar la coma a la derecha. El máximo

representable es $(2^{n-1} - 1)/2^f$, y el mínimo, $-2^{n-1}/2^f$. La diferencia máx − mín $= 2^{-f}$ es igual a la resolución.

Pero la rigidez que resulta es evidente: para conseguir la máxima extensión, en cada caso tendríamos que decidir dónde se encuentra la coma, en función de que estemos trabajando con números muy grandes o muy pequeños. Aun así, la extensión que se consigue puede ser insuficiente: con 32 bits, el número máximo representable (todos «unos» y la coma a la derecha del todo) sería $2^{31} - 1 \approx 2 \times 10^9$, y el mínimo positivo (coma a la izquierda y todos «ceros» salvo el bms) sería $2^{-31} \approx 2 \times 10^{-10}$

Por esto, lo más común es utilizar un formato de coma flotante, mucho más flexible y con mayor extensión.

Formatos de coma flotante

El principio de la representación en coma flotante de un número X consiste en codificar dos cantidades, A y E, de modo que el número representado sea:

$$X = \pm A \times b^E$$

Normalmente, $b = 2$. Sólo en algunos ordenadores se ha seguido el convenio de que sea $b = 16$.

En un formato de coma flotante hay tres campos (figura 4.4):

- S: *signo* del número, 1 bit.
- M: *mantisa*, m bits.
- C: *característica*, e bits.

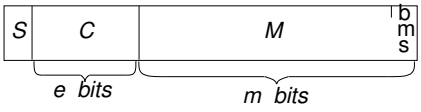

Figura 4.4: Formato de coma flotante.

Se pueden seguir varios convenios para determinar cómo se obtienen los valores de A y E en función de los contenidos binarios de M y C

Mantisa

Los m bits de la mantisa representan un número en coma fija, A. Hace falta un convenio sobre la parte entera y la fraccionaria (o sea «dónde está la coma»). Normalmente la representación está «normalizada». Esto quiere decir que el exponente se ajusta de modo que la coma quede a la derecha del bit menos significativo de M (normalización entera) o a la izquierda del más significativo (normalización fraccionaria). Es decir:

- Normalización entera: $X = \pm M \times 2^E$
- Normalización fraccionaria: $X = \pm 0, M \times 2^E$, o bien:
  $X = \pm 1, M \times 2^E$

Normalizar es trivial: por cada bit que se desplaza la coma a la izquierda o a la derecha se le suma o se le resta una unidad, respectivamente, a E.

Veamos con un ejemplo por qué conviene que la representación esté normalizada. Para no perdernos con cadenas de bits, supongamos por esta vez, para este ejemplo, que la base es $b = 16$. Supongamos también que el formato reserva $m = 24$ bits para la mantisa (24/4 = 6 dígitos hexadecimales), y que se trata de representar el número π :

$X = \pi = 3{,}1415926\ldots = 0\text{x}3{,}243F69A\ldots$

Consideremos primero el caso de normalización entera; tendríamos las siguientes posibilidades:

M	E	Número representado
0x000003	0	$0x3\times16^{0} = 0x3 = 3$
0x000032	-1	$0x32\times16^{-1} = 0x3{,}2 = 3{,}125$
0x000324	-2	$0x324\times16^{-2} = 0x3{,}24 = 3{,}140625$
0x003243	-3	$0x3243\times16^{-3} = 0x3{,}243 = 3{,}1413574$
0x03243F	-4	$0x3243F\times16^{-4} = 0x3{,}243F = 3{,}1415863$
0x3243F6	-5	$0x3243F6\times16^{-5} = 0x3{,}243F6 = 3{,}141592$

La última es la representación normalizada; es la que tiene más dígitos significativos, y, por tanto, mayor precisión.

Con normalización fraccionaria y el convenio $X = \pm 0, M \times 16^{E}$:

M	E	Número representado
0x000003	6	$0x0{,}000003\times16^{6} = 0x3 = 3$
0x000032	5	$0x0{,}000032\times16^{5} = 0x3{,}2 = 3{,}125$
...	...	...
0x3243F6	1	$0x0{,}3243F6\times16^{1} = 0x3{,}243F6 = 3{,}141592$

y, como antes, la última es la representación normalizada.

El número «0» no se puede normalizar. Se conviene en representarlo por todos los bits del formato nulos.

Para los números negativos se pueden seguir también los convenios de signo y magnitud, complemento a 1 o complemento a 2 de la mantisa.

Exponente

En los e bits reservados para el exponente podría seguirse también el convenio de reservar uno para el signo y los $e-1$ restantes para el módulo, o el complemento a 1 o a 2 si $E < 0$. Pero el convenio más utilizado no es éste, sino el de la representación *en exceso de* 2^{e-1}: el número *sin signo* contenido en esos e bits, llamado **característica**, es:

$$C = E + 2^{e-1}$$

Dado que $0 \leq C \leq 2^{e} - 1$, los valores representables de E estarán comprendidos entre -2^{e-1} (para $C = 0$) y $2^{e-1} - 1$ (para $C = 2^{e} - 1$).

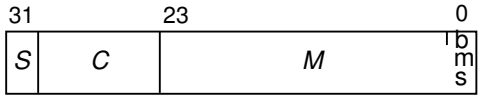

Figura 4.5: Un formato de coma flotante con 32 bits.

Como ejemplo, veamos la representación del número π en el formato de la figura 4.5, suponiendo que el exponente está en exceso de 64 con base 16 y que la mantisa tiene normalización fraccionaria. De acuerdo con lo que hemos explicado antes, la mantisa será:
0x3243F6 = 0b0011001001000011111110110.
Para la característica tendremos: $C = 1 + 64 = 65 = 0x41 = 0b1000001$, y la representación resulta ser:
0 1000001 0011 0010 0100 0011 1111 0110

Observe que, al ser la base 16, la mantisa está normalizada en hexadecimal, pero no en binario, ni lo puede estar en este caso, porque cada incremento o decremento de E corresponde a un desplazamiento de cuatro bits en M.

Si cambiamos uno de los elementos del convenio, la base, y suponemos ahora que es 2, al normalizar la mantisa resulta:
0x3,243F6A... =
0b11,001001000011111101101010... =
$0b0{,}1100100100001111111011010101\ldots\times2^{2}$

La característica es ahora $C = 2 + 64$. Truncando a 24 los bits de la mantisa obtenemos:
0 1000010 1100 1001 0000 1111 1101 1010
Vemos que al normalizar en binario ganamos dos dígitos significativos en la mantisa.

El mismo número pero negativo ($-\pi$), con convenio de complemento a 2, se representaría en el primer caso ($b = 16$) como:
1 1000001 1100 1101 1011 1100 0000 1010
y en el segundo ($b = 2$):
1 1000010 0011 0110 1111 0000 0010 0110

El inconveniente de la base 16 es que la mantisa no siempre se puede normalizar en binario, y esto tiene dos consecuencias: por una parte, como hemos visto, pueden perderse algunos bits (1, 2 o 3) con respecto a la representación con base 2; por otra, hace algo más compleja la realización de operaciones aritméticas. La ventaja es que, con el mismo número de bits reservados para el exponente, la escala de números representables es mayor. Concretamente, para $e = 7$ el factor de escala, con $b = 2$, va de 2^{-64} a 2^{63} (o, aproximadamente, de 5×10^{-20} a 10^{19}), mientras que con $b = 16$ va de 16^{-64} a 16^{63} (aproximadamente, de 10^{-77} a 10^{76}).

Norma IEEE 754

El estándar IEEE 754 ha sido adoptado por la mayoría de los fabricantes de hardware y de software. Define los formatos, las operaciones aritméticas y el tratamiento de las «excepciones» (desbordamiento, división por cero, etc.). Aquí sólo nos ocuparemos de los formatos, que son los esquematizados en la figura 4.6.[3]

Los convenios son:

- Números negativos: signo y magnitud.
- Mantisa: normalización fraccionaria.
- Base implícita: 2.
- Exponente: en exceso de $2^{e-1} - 1$

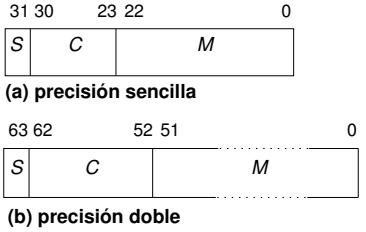

Figura 4.6: Formatos de la norma IEEE 754.

Como la mantisa tiene normalización fraccionaria y los números negativos se representan con signo y magnitud, el primer bit de la mantisa siempre sería 1, lo que permite prescindir de él en la representación y ganar así un bit. La coma se supone inmediatamente a la derecha de ese «1 implícito», por lo que, dada una mantisa M y un exponente E, el número representado es $\pm(1, M) \times 2^E$.

El exponente, E, se representa en exceso de $2^{e-1} - 1$ (no en exceso de 2^{e-1}). La extensión de E en el formato de 32 bits, que tiene $e = 8$, resulta ser: $-127 \leq E \leq 128$, y en el de 64 bits ($e = 11$): $-1.023 \leq E \leq 1.024$. Pero los valores extremos de C (todos ceros o todos unos) se reservan para representar casos especiales, como ahora veremos. Esto nos permite disponer, en definitiva, de un factor de escala que está comprendido entre 2^{-126} y 2^{127} en el formato de 32 bits y entre $2^{-1.022}$ y $2^{1.023}$ en el de 64 bits.

Resumiendo, dados unos valores de M y C, el número representado, X, si se trata del formato de precisión sencilla (figura 4.6(a)), es el siguiente:

1. Si $0 < C < 255, X = \pm(1, M) \times 2^{C-127}$

[3] La norma contempla otros dos formatos, que son versiones extendidas de éstos: la mantisa ocupa una palabra completa (32 o 64 bits) y la característica tiene 10 o 14 bits. Pero los de la figura 4.6 son los más utilizados.

2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-126}$
   (números más pequeños que el mínimo representable en forma normalizada)
4. Si $C = 255$ y $M = 0, X = \pm\infty$
   (el formato prevé una representación específica para «infinito»)
5. Si $C = 255$ y $M \neq 0, X = \text{NaN}$
   («NaN» significa «no es un número»; aquí se representan resultados de operaciones como $0/0, 0\times\infty$, etc.)

Y en el formato de precisión doble (figura 4.6(b)):

1. Si $0 < C < 2.047, X = \pm(1, M) \times 2^{C-1.023}$
2. Si $C = 0$ y $M = 0, X = 0$
3. Si $C = 0$ y $M \neq 0, X = \pm 0, M \times 2^{-1.022}$
4. Si $C = 2.047$ y $M = 0, X = \pm\infty$
5. Si $C = 2.047$ y $M \neq 0, X = \text{NaN}$

Una «calculadora» para pasar de un número a su representación y a la inversa se encuentra en `http://babbage.cs.qc.edu/IEEE-754/`