Sistemas Numéricos y Representación de Números en una Computadora

José Galaviz Casas

Departamento de Matemáticas
Facultad de Ciencias
Universidad Nacional Autónoma de México

Contenido

1

Sistemas numéricos posicionales en bases 2, 8 y 16

A lo largo de la historia la humanidad ha utilizado muy diversos métodos para escribir números, los romanos, egipcios y babilonios utilizaban sistemas de escritura que suelen llamarse *aditivos*, en estos sistemas el valor de un número es la suma de los valores de cada uno de los dígitos que lo componen, por ejemplo XXVI en romano es un 26, es decir dos veces 10 (X), más 5 (V), más uno (I). Un símbolo "V" en cualquier parte de un número romano siempre vale 5 unidades.

A diferencia de los sistemas numéricos aditivos, en los sistemas posicionales el valor de cada dígito depende de su posición dentro del número donde aparece. El sistema numérico posicional más conocido es, por supuesto, nuestro usual sistema indo-arábigo. Como nos lo dijeron en la enseñanza elemental:
3486.03 = 3 millares + 4 centenas + 8 decenas + 6 unidades + 3 centésimos

es decir:

$$3486.03 = 3 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 6 \times 10^0 + 0 \times 10^{-1} + 3 \times 10^{-2}$$

el primer "3" significa "tres millares"mientras que el último significa "tres centésimos", el símbolo "3" tiene un valor diferente dependiendo de su posición dentro del número.

Hay que notar además que cada posición dentro del número está asociada a una potencia de 10. Algunos sistemas numéricos posicionales tienen asociada una *base*[1]. Así, cada dígito tiene un "valor intrínseco" (como el "3" del ejemplo anterior cuyo valor intrínseco es justamente 3) y ese valor se multiplica por una potencia de la base del sistema (10 en el ejemplo), el valor del exponente al que se eleva la base crece hacia la izquierda y decrece hacia la derecha.

El valor intrínseco de los dígitos de un sistema posicional en base b está en el conjunto $\{0, \ldots, b-1\}$. En nuestro sistema indo-arábigo decimal (es decir, en base 10) los dígitos posibles son los símbolos "0", "1", ..., "9"; cuyos valores son, respectivamente, 0 unidades, una unidad, etc.

Así pues, un número x en un sistema posicional en base b se escribe como una secuencia de dígitos:

$$x = x_n x_{n-1} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$$

donde cada dígito x_i posee un valor intrínseco $|x_i|$ en el conjunto $\{0, \ldots, b-1\}$ ($i \in \{-m, \ldots, n\}$). El valor del número x, denotado $|x|$, es:

$$|x| = \sum_{i=-m}^{n} |x_i| \, b^i \tag{1.0.1}$$

Mientras mayor sea la potencia de la base asociada a un dígito en el número se dice que el dígito es *más significativo*. Así el dígito de la extrema derecha de un número es el *menos significativo* y el de la extrema izquierda es el *más significativo*. A partir de este momento denotaremos que el número x está escrito en base b como x_b y nos avocaremos a tratar con números enteros.

En el ámbito de la computación electrónica son particularmente interesantes los sistemas numéricos posicionales en las bases 2 (binario), 8 (octal) y 16 (hexadecimal). Esto es porque, como probablemente ha escuchado el lector en repetidas ocasiones, las computadoras electrónicas modernas operan en binario, en ceros y unos (los únicos dos dígitos en un

[1] No en todos los sistemas numéricos posicionales hay una base asociada. Los mayas usaban un sistema numérico posicional que "casi" tenía base 20 para representar números relacionados con el calendario. La tercera posición de los números no se multiplicaba por 20 sino por 18.

sistema posicional base 2). Los circuitos digitales de nuestras computadoras sólo distinguen cuando hay corriente en una línea y cuando no la hay. Las bases 8 y 16 son interesantes porque es muy fácil traducir la expresión de un número entre cualesquiera de estas bases y la base 2 y viceversa.

Los dígitos en el sistema posicional binario, normalmente llamados *bits* (*Binary digITS*), son "0" y "1" cuyos valores intrínsecos son, respectivamente 0 y 1. Los dígitos en el sistema octal son: "0", "1", ..., "7", y sus valores son: 0, 1, ..., 7, respectivamente. En hexadecimal los dígitos son: "0", "1", ..., "9", "A", "B", "C", "D", "E", "F" y sus valores son: 0, 1, ..., 9, 10, 11, 12, 13, 14 y 15, respectivamente.

1.1 Conversiones binario-decimal y decimal a cualquier base

Receta: Para traducir un número escrito en base 2 a su expresión equivalente en base 10 no hay más que obtener su valor de acuerdo a la expresión 1.0.1

Por ejemplo:

$$01010_2 = 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 2 = 10_{10}$$

Hacer la operación inversa es un poco más complicado.

Receta: Si se tiene un número x_{10} su expresión en base 2 se obtiene dividiendo x_{10} por 2 tantas veces como sea posible hasta obtener cero como cociente y colocar los residuos en el orden inverso de como fueron obtenidos.

Por ejemplo, para convertir 41_{10} a binario se sigue el proceso mostrado en la figura 1.1.

Ahora sabemos cómo hacer la conversión decimal-binario, pero no sabemos por qué se hace así. De hecho el procedimiento funciona para traducir la expresión decimal de un número a su expresión en cualquier base arbitraria b. Para explicar el por qué funciona el método recordemos el algoritmo de la división:

Teorema 1 *Sean $m, n \in \mathbb{Z}$ con $n > 0$. Existen dos enteros únicos v y r tales que:*

$$m = v\,n + r \tag{1.1.2}$$

donde $0 \leq r < n$. A v se le llama el cociente y a r el residuo.

Con esto en mente, sea R_{10} el número en decimal que se desea traducir a una base b. Es decir, se desean obtener los dígitos d_i del número:

$$(d_n d_{n-1} \ldots d_1 d_0)_b$$

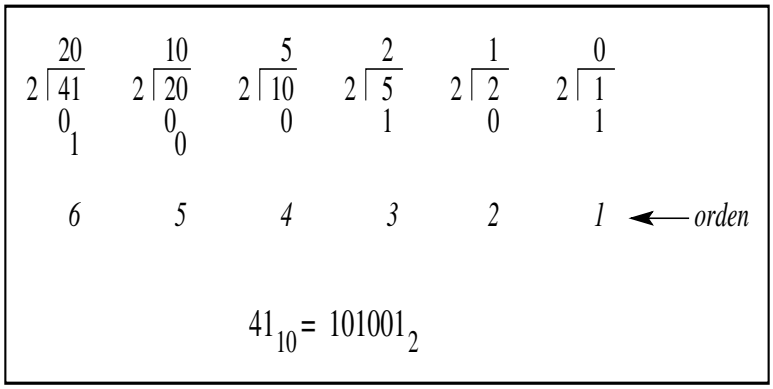

Figura 1.1: Conversión de 41_{10} a binario. Nótese que los residuos se escriben en el orden inverso de como fueron obtenidos, el último residuo es el bit más significativo.

tal que:

$$R_{10} = d_n b^n + \cdots + d_1 b + d_0 = b(d_n b^{n-1} + \cdots + d_1) + d_0 \tag{1.1.3}$$

Homologando las expresiones 1.1.2 del algoritmo de la división y 1.1.3, sabemos que el residuo de dividir R_{10} por b es d_0 (el dígito menos significativo del número en base b). Si repetimos el proceso de división por b, usando ahora como dividendo el cociente $(d_n b^{n-1} + \cdots + d_1)$ de la expresión 1.1.3, obtendremos de residuo d_1. Iterando el proceso vamos obteniendo los dígitos d_2, d_3, etc., en orden de menos a más significativo, de allí que se tenga que invertir el orden al momento de escribir el resultado de la conversión.

Si dividimos el 41_{10}, usado de ejemplo en la figura 1.1, por 16 obtenemos 2 de cociente y 9 de residuo, si luego tratamos de dividir el cociente (2) entre 16 nuevamente obtenemos 0 de cociente y (evidentemente) 2 de residuo, así que:

$$41_{10} = 29_{16}$$

1.2 Conversiones binario-octal, binario-hexadecimal y sus inversas

Los sistemas numéricos octal y hexadecimal son importantes en el ámbito de la computación electrónica porque, como mencionamos, es muy fácil pasar un número de base 2 a base 8 ó 16. Es decir, los computologos usan las bases 8 y 16 como medios para escribir, abreviadamente, números en base 2.

Octal	binario	Octal	binario
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Tabla 1.1: Los dígitos octales y su valor en binario.

Receta: Para convertir la expresión binaria de un número a su correspondiente octal se agrupan, de derecha a izquierda y de tres en tres, los bits del número y se escribe el valor de cada terna.

En la tabla 1.1 se muestra cada dígito octal y la expresión de su valor en binario.

Por ejemplo, usando el mismo número de la figura 1.1:

$$101\ 001_2 = 51_8$$

los tres bits menos significativos son 001_2 esto es un $1 \times 2^0 = 1$, el dígito octal menos significativo y los siguientes tres son 101_2 lo que equivale a $2^4 + 2^0 = 5$, el dígito octal más significativo. Si el número de bits en la expresión binaria no es exactamente múltiplo de tres entonces no se alcanza a formar la terna más significativa, para arreglar esto simplemente se agregan tantos ceros a la izquierda como sean necesarios. Por ejemplo:

$$10\ 111\ 011_2 = 010\ 111\ 011_2 = 273_8$$

Para convertir la expresión binaria de un número a su equivalente en hexadecimal hay que hacer algo similar, sólo que ahora se deben agrupar de cuatro en cuatro los bits.

Receta: Para convertir la expresión binaria de un número a su correspondiente hexadecimal se agrupan, de derecha a izquierda y de cuatro en cuatro, los bits del número y se escribe el valor de cada cuarteta.

En la tabla 1.2 se muestra cada dígito octal y la expresión de su valor en binario.

Regresando a nuestro ejemplo:

$$10\ 1001_2 = 0010\ 1001_2 = 29_{16}$$

Nuevamente sabemos cómo hacer estas conversiones pero aún no sabemos por qué. Analicemos con cuidado qué hacemos para pasar de base 2 a 8.

Hexadecimal	**binario**	**Hexadecimal**	**binario**
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tabla 1.2: Los dígitos hexadecimales y su valor en binario.

Sea $(a_n a_{n-1} \dots a_2 a_1 a_0)_2$ un número escrito en binario, su valor está dado por:

$$\begin{aligned} a_n \times 2^n + \cdots + a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2 + a_0 \quad &= \\ 8(a_n \times 2^{n-3} + \cdots + a_3) + a_2 \times 2^2 + a_1 \times 2 + a_0 \end{aligned} \tag{1.2.4}$$

Por otra parte necesitamos encontrar los dígitos octales b_k, b_{k-1}, ..., b_1, b_0, que constituyen la expresión octal del número. Usando esta expresión el valor es:

$$\begin{aligned} b_k \times 8^k + \cdots + b_1 \times 8 + b_0 \quad &= \\ 8(b_k \times 8^{k-1} + \cdots + b_1) + b_0 \end{aligned} \tag{1.2.5}$$

Así que 1.2.4 y 1.2.5 deben coincidir:

$$\begin{aligned} 8(a_n \times 2^n + \cdots + a_3) + a_2 \times 2^2 + a_1 \times 2 + a_0 \quad &= \\ 8(b_k \times 8^{k-1} + \cdots + b_1) + b_0 \end{aligned} \tag{1.2.6}$$

Por el algoritmo de la división sabemos que son únicos el cociente y el residuo que resultan de dividir por 8 el valor del número, así que de la expresión 1.2.6 tenemos:

$$a_n \times 2^n + \cdots + a_3 = b_k \times 8^{k-1} + \cdots + b_1$$

y lo que es más importante:

$$b_0 = a_2 \times 2^2 + a_1 \times 2 + a_0 \tag{1.2.7}$$

es decir, el valor del dígito octal menos significativo es justamente el de los tres bits menos significativos de la expresión binaria. Si repetimos el proceso dividiendo por 8 los cocientes

obtendremos que el valor de cada dígito octal es el de la terna que le corresponde en la expresión binaria.

Análogamente podríamos justificar nuestra receta para pasar de base 2 a hexadecimal.

Para traducir expresiones octales o hexadecimales a su equivalente en binario se aplica el mismo principio pero a la inversa. Para pasar de octal a binario simplemente se reemplaza cada dígito octal por la terna de bits equivalente, se deben usar estrictamente tres bits, sin importar que el valor pueda ser escrito en menos. Para pasar de hexadecimal a binario se reemplaza cada dígito hexadecimal por la cuarteta de bits, estricta, del mismo valor.

Receta: Para convertir la expresión octal de un número a su correspondiente binaria se reemplaza cada dígito octal por la terna de bits que representa su valor en binario. Para convertir una expresión hexadecimal a binario se reemplaza cada dígito hexadecimal por la cuarteta binaria que representa su valor.

Por ejemplo:

$$217_8 = 010\ 001\ 111_2$$

$$F30_{16} = 1111\ 0011\ 0000_2$$

2

Representación de números negativos

2.1 Signo y magnitud

Normalmente utilizamos el símbolo "-" precediendo a un número para indicar que éste es menor que cero. Esta es una notación muy práctica en la vida cotidiana pero no puede ser utilizada en la representación que se hace de los números en una computadora, recordemos que sólo se pueden utilizar los dígitos binarios para representar cualquier cosa en ellas y el "-" no es ningún bit.

Pero podemos utilizar la misma idea, preceder el número de un bit que indique su signo, después de todo sólo hay dos posibles signos, a saber: "+"y "-". Todo lo que tenemos que hacer es asignar arbitrariamente un bit a cada signo. Convencionalmente se hace: "+" = 0, "-" = 1.

A este método de representación de números negativos se le denomina *signo y magnitud* porque, análogamente a lo que solemos hacer, se coloca un símbolo que precede al número y

que indica su signo y luego se pone la magnitud del número. En esta notación por ejemplo:

$$1\ 01010_2 = -10_{10}$$

De esta manera es muy fácil distinguir los números positivos de los negativos, si utilizamos un número fijo de bits para representarlos y decidimos que siempre el primer bit es para el signo del número, entonces basta con observar el primer bit de la izquierda (al que en este caso no podemos decirle formalmente "el más significativo", dado que no tiene asociada ninguna potencia de 2) para determinar si se trata de un número negativo o positivo.

En este caso la "multiplicación por -1" de un número equivale a *negar* o *invertir* el bit de la extrema izquierda, esto es, convertirlo en cero si vale 1 y viceversa. De hecho la idea fundamental detrás de la representación de signo y magnitud es que ocurra: $-(-a) = a$. Lo que nos parece evidente por estar acostumbrados a nuestra representación de números negativos convencional.

Un inconveniente del sistema de signo y magnitud es que existen dos representaciones distintas para el cero, es decir, el neutro aditivo no es único formalmente hablando, tanto el $10\ldots0$ como el $00\ldots0$ son cero, el primero con signo "-"y el segundo con signo "+", lo que tampoco es correcto desde el punto de vista matemático, dado que el cero no es ni positivo ni negativo.

Esta dualidad del cero tiene implicaciones importantes en una computadora digital. Los procesadores tienen generalmente instrucciones para cambiar al flujo de los programas llamadas saltos, hay saltos incondicionales (siempre que el procesador ejecuta la instrucción de salto la siguiente instrucción es aquella indicada por el salto) y hay saltos condicionales (la instrucción siguiente es a veces la que está bajo la del salto y a veces la indicada por el salto dependiendo de alguna condición). Y generalmente la condición de salto es establecida comparando algún dato con cero. Si hay dos representaciones del cero hay que hacer dos comparaciones y eso lleva más tiempo que hacer sólo una.

2.2 Complemento a 1

Otra manera de representar números negativos es la conocida como *complemento a 1*. Para hablar de ella primero trataremos con una generalización.

Definición 1 El complemento a $b-1$ de un número r, representado en k dígitos en base b se define como:

$$C_{b-1}(r_b) = (b-1_k \ldots b-1_1) - r_b$$

donde $b-1$ es el valor máximo de un dígito en base b.

Por ejemplo el complemento a 9 de el número 13579_{10} es:

$$C_9(13579_{10}) = 99999 - 13579 = 86420_{10}$$

nótese que el minuendo que se ha usado tiene tantos nueves como dígitos tiene el número 13579.

En el caso de nuestro sistema binario hablaremos del complemento a 1 del número n en k bits como el resultado de restar n al número constituido por k unos. Por ejemplo:

$$C_1(01101_2) = 11111 - 01101 = 10010_2$$

nótese que cada bit del resultado es el negado del número original. De hecho esta es la receta práctica para obtener el complemento a 1 de cualquier número binario rápidamente.

Receta: El complemento a uno de un número binario n_2 se obtiene invirtiendo cada bit de n_2.

Por ejemplo:

$$C_1(10010110111_2) = 01101001000_2$$

Una alternativa de representación de números negativos en la computadora es utilizando el complemento a 1, es decir, el negativo de un número es su complemento a 1. Por ejemplo:

$$10_{10} = 01010_2$$

$$-10_{10} = 10101_2$$

Al igual que en el caso de signo y magnitud se adopta la convención de que todos los números cuyo bit del extremo izquierdo sea cero son positivos y por ende, todos aquellos cuyo bit del extremo izquierdo es 1 son negativos.

Nuevamente, como en signo y magnitud, la idea es que $-(-a) = a$. También tenemos el problema de que hay dos distintas representaciones para el cero: $0\ldots0$ y $1\ldots1$. El primero es un cero con signo "+"y el segundo un cero con signo "-".

También hay que notar que tenemos tantos números positivos como negativos, tanto en signo y magnitud como en complemento a 1. Supongamos que se utilizan k bits para representar nuestros números enteros en una computadora. ¿Cuántos números representables tenemos? bueno, si tengo k lugares en los que puedo poner 0 ó 1 y cada vez que elijo el lugar i tengo esas dos posibilidades para el lugar $i+1$ entonces tengo en total 2^k combinaciones,

es decir, números representables, ahora bien, ¿cuántos de estos 2^k números son negativos (o mejor dicho tienen signo "-")? tanto en complemento a 1 como en signo y magnitud los que tienen signo "-" son aquellos que empiezan con 1 que son justamente la mitad de todos nuestros números, es decir 2^{k-1}, lo mismo ocurre con los que tienen signo "+", también son 2^{k-1}.

2.3 Complemento a 2

Ya mencionamos el inconveniente que haya dos representaciones diferentes del cero en el contexto de nuestras computadoras digitales. Para evitar esto (que sin embargo se puede sobrellevar), se inventó otro mecanismo para representar números negativos, se denomina *complemento a 2*, eso nos lleva a considerar, en general, el complemento a la base.

Definición 2 El complemento a b de un número r, representado en k dígitos en base b se define como:

$$C_b(r_b) = (1\,0_k \ldots 0_1) - r_b$$

nótese que el número que se utiliza ahora como minuendo tiene un dígito más que los usados en la representación, es decir tiene $k+1$ dígitos, un 1 seguido de k ceros a la derecha. Por ejemplo, el complemento a 10 de 13579_{10} es:

$$C_{10}(13579_{10}) = 100000 - 13579 = 86421_{10}$$

el resultado es, evidentemente, el mismo que se obtuvo en el complemento a 9 incrementado en uno, es decir: $C_b(x_b) = C_{b-1}(x_b) + 1$.

En el caso particular de base 2, el complemento a 2 de un número x_2 es el resultado de sumar 1 al complemento a 1 de x_2 que, como vimos, no es otro que el número negado bit a bit. Por ejemplo[1]:

$$C_2(01101_2) = 10010 + 00001 = 10011_2$$

También existe una receta rápida para obtener el complemento a 2 de un número binario.

Receta: El complemento a 2 del número x_2 se obtiene copiando, de derecha a izquierda, todos los bits de x_2 hasta encontrar el primer 1 inclusive e invertir todos los bits restantes hacia la izquierda.

[1] En este ejemplo no ocurre, pero pudiera ser que al sumar dos dígitos binarios ambos fueran 1, el resultado en este caso sería $2_{10} = 10_2$, por lo que se colocaría, a la manera de una suma convencional en base 10, el dígito menos significativo y el otro se lleva como acarreo.

Por ejemplo:

$$C_2(00110011\,100_2) = 11001100\,100_2$$

Entonces es posible representar el negativo de un número binario como su complemento a 2. La idea detrás de esta representación es que: $a + (-a) = 0$. Un número más su negativo, que es de hecho su inverso aditivo, nos da cero, *un único cero*. A diferencia de signo y magnitud y de complemento a 1, en la representación en complemento a 2 de números negativos tenemos una sola representación de cero, a saber: $0\ldots0$. Esta vez el negativo, es decir el complemento a 2, de $0\ldots0$ es justamente $0\ldots0$.

Además conservamos la ventajosa propiedad exhibida por signo y magnitud y complemento a 1 de poder determinar fácilmente si un número es negativo o positivo observando el bit del extremo izquierdo.

Sin embargo no todo es perfecto, tenemos una desventaja. Concluimos que hay una sola representación de cero en complemento a 2 en k bits, eso está bien, ahora ¿cuántos números negativos se pueden representar? todos los números de k bits que empiezan con 1, es decir 2^{k-1}, ¿cuantos números positivos se pueden representar? pues también hay 2^{k-1} que empiezan con cero, pero uno de ellos es el cero $(0\ldots0)$, hay entonces exactamente $2^{k-1} - 1$ números positivos. ¡Ajá! hay un número negativo, el más grande en magnitud, $10\ldots0$, que no tiene su inverso aditivo en el conjunto de 2^k posibles números. Por ejemplo, en cuatro bits:

$$C_2(0100_2) = 1100_2$$

donde: $0100_2 = 4_{10}$ y $1100_2 = -4_{10}$, porque $0100 + 1100 = 0000_2$. En cambio:

$$C_2(1000_2) = 10000_2$$

lo que es un error.

Por ejemplo, en una computadora que utilice, como es común, 16 bits para representar ciertos enteros con signo[2], el número más grande positivo representable es 32767 y el más grande negativo es -32768 que no posee su inverso aditivo en el conjunto $\{-32768, \ldots, 32767\}$.

2.4 Exceso

Otro mecanismo para representar números negativos es el conocido como *exceso a x*. Si el número de bits usados para representar enteros es k entonces generalmente $x = 2^{k-1}$ o $x = 2^{k-1} - 1$. Por ejemplo, si se utilizan 8 bits para representar enteros entonces el

[2]Como el tipo `short` de Java

sistema utilizado podría ser *exceso a 128* o bien *exceso a 127*. La idea del sistema es que, para representar el número n en k bits se le suma a n, el valor del exceso (esto es 2^{k-1} o $2^{k-1}-1$), obteniéndose $n+e$ entonces n se escribe como $n+e$ en binario (ya sin consideraciones de signo por supuesto). Por ejemplo, para escribir en 8 bits el número -100_{10} en exceso a $2^{8-1}=2^7=128$ hacemos: $-100+128=28_{10}$, esto en binario se escribe: 00011100_2 (16+8+4), por lo que, en exceso a 128 en 8 bits $-100_{10}=00011100_2$. En cambio, usando las mismas condiciones (8 bits, exceso a 128): $100+128=228_{10}=11100100_2$, es decir: $100_{10}=11100100_2$. Nuevamente hay un solo cero (en exceso a 128 en 8 bits sería 10000000_2), lo que significa, dado que la cantidad de números representables en k bits es par, que hay un negativo o un positivo "de más", en nuestro ejemplo es el -128 (porque su inverso aditivo sería 128 y $128+128=256$, que no se puede escribir en 8 bits), el único cero es 10000000_2 y el rango de representatividad es: $\{-128,\ldots,127\}$, cabe señalar que los números en exceso a 2^{k-1} y en complemento a 2 se escriben igual salvo el bit más significativo.

Para saber entonces que número está siendo representado por una cadena de bits debemos saber el valor del exceso. Si nos topamos con un 01101101_2 y se nos dice que está representando a un número en exceso a 128 entonces sabemos que al valor del número sín consideraciones de signo (109_{10}) se le debe restar un 128 para determinar su verdadero valor, es decir nuestra cadena 01101101_2 está representando al número $109_{10}-128_{10}=-19_{10}$.

En exceso a $2^{k-1}-1$ se hace lo mismo, sólo que el número a sumar es, por supuesto $n=2^{k-1}-1$. Si se utilizan 8 bits en la representación, el sistema sería *exceso a 127*, este caso particular nos resultará útil cuando consideremos la representación de números en punto flotante.

En exceso a 127 en ocho bits un número n es representado como la cadena de bits que le correspondería al número n+127 en binario. Por ejemplo nuestro -19_{10} anterior se escribiría como $-19+127=108_{10}=01101100_2$, el cero sería 01111111_2 el $-127_{10}=00000000_2$ el $127_{10}=11111110_2$ y el $128_{10}=11111111_2$, este último es el que no posee su inverso (-128) en el rango de representatividad del sistema que resulta ser $\{-127,\ldots,128\}$. Nótese que el número negativo más grande es representado como una cadena de ceros.

3

Representación de números reales en punto flotante

Frecuentemente utilizamos en nuestros programas números "reales". Lo decimos así, entre comillas, porque sabemos que el conjunto de números reales es infinito y de hecho con la potencia del continuo y nuestras computadoras sólo pueden manejar números finitos. Es decir nuestras computadoras sólo pueden manejar un subconjunto finito de los números naturales, mientras que los reales, además de ser un conjunto infinito, son no numerables. Así que realmente necesitamos trucos para aparentar que manipulamos números reales, aun cuando realmente sólo podemos manejar un conjunto bastante pequeño de ellos.

Una posibilidad es fijar, dada la longitud de nuestra representación n, un cierto número de bits f para expresar la parte fraccionaria de un número y el resto $n-f$ para representar la parte entera. A este esquema se le denomina *representación de punto fijo*. Es un esquema válido y útil en ciertos contextos, hace que las operaciones sean rápidas y precisas en cierto rango. Pero es muy poco general, hay que cambiar f para representar subconjuntos

diferentes de números.

Por ello se inventó el sistema de representación de punto flotante, análogo a lo que conocemos como notación científica, donde un número se escribe de la forma $\pm a \times b^k$ donde b es la base, a es llamada la *mantisa* y k es el *exponente*. Por razones obvias en la representación en punto flotante en la computadora b es 2.

Al principio cada fabricante de computadoras que adoptaba el sistema de representación de punto flotante decidía su propio formato, es decir, el orden y tamaño que tenía la mantisa, su signo y el exponente. Luego el IEEE estableció un estándar que utilizan todas las computadoras y compiladores de hoy en día.

Hay formatos de IEEE para representar números en punto flotante con diferentes precisiones: simple, doble, cuádruple. Revisaremos el formato de precisión simple.

En el formato de precisión simple de IEEE los números reales se representan en 4 bytes (32 bits). El primer bit, el más significativo de los 32, es el signo de la mantisa en el esquema de signo y magnitud, es decir "1" significa negativo, "0" significa no negativo. Los siguientes 8 bits sirven para decir la magnitud de la mantisa *normalizada*.

Que la mantisa esté normalizada significa esencialmente lo siguiente: supongamos que tenemos el número 0.004×10^3 este es equivalente a 0.04×10^2 y a 0.4×10 y también a 4×10^0. Hay muchas maneras de representar un número en notación científica. Para que la representación sea única fijamos una restricción. Hacemos que el número que queramos representar tenga su primer dígito más significativo distinto de cero inmediatamente a la izquierda del punto. Esto es normalizar la mantisa. En nuestro ejemplo 4×10^0 es la representación en mantisa normalizada de 0.004×10^3 y de los demás equivalentes.

Si normalizamos la mantisa de un número binario, lo que estamos haciendo es forzar a que el primer bit más significativo de la mantisa distinto de cero (sólo puede ser 1 entonces) quede a la izquierda inmediatamente antes del punto. Como ya sabemos que este dígito sólo puede ser 1, lo omitimos. Así por ejemplo la parte de la mantisa de 1.0×2^k que realmente aparece en la representación de mantisa normalizada del formato de IEEE de punto flotante, es cero, porque el uno a la izquierda del punto no se pone, ya sabemos que esta allí.

El exponente se representa en exceso a 127 usando los 8 bits siguientes después del primero, que, como hemos dicho, sirve para escribir el signo de la mantisa. Los restantes 23 bits son para escribir el valor de la mantisa normalizada.

En este esquema por ejemplo: $3F800000_{16}$ es el número representado por:

signo de la mantisa: El primer bit del 3 inicial, $3_{16} = 0011_2$ así que el signo de la mantisa es 0, la mantisa es no negativa.

valor del exponente: los tres últimos bits del 3 y los siguientes seis bits, los cuatro de la F y el primero del 8. Es decir todos 1's salvo el primero, por lo que el valor que aparece es: $01111111_2 = 127_{10}$ como el exponente está en exceso a 127 hay que restarle a su valor nominal el valor del exceso para obtener su valor real, $127 - 127 = 0$. Así que el exponente es cero.

valor de la mantisa: la mantisa es puros ceros, así que su valor real es $1_2 = 1_{10}$

considerando todo lo anterior el valor del número es: 1.0×2^0 que es 1.0.

Análogamente el 2.0 se escribe 40000000_{16} porque el primer bit es 0 (signo de mantisa), los siguientes 8 bits forman el número 128 y $128 - 127 = 1$ y la magnitud de la mantisa normalizada sin el bit más significativo es cero por lo que su valor real es 1. Así que $40000000_{16} = 1.0 \times 2^1 = 2.0_{10}$ en el formato de IEEE.

Hay un pequeño problema y es que no es posible representar realmente el cero. Dado que siempre suponemos que la mantisa está normalizada y que hay un uno a la izquierda del punto que no se escribe, el cero no se puede representar. Así que IEEE establece convenciones para poder hacerlo y, de hecho, fija valores especiales:

- El cero se representa como puros ceros. El valor real de este número sería 1.0×2^{-127} que es pequeño, pero no cero. Sin embargo si un procesador o un compilador se apega al estándar debe hacer la consideración especial de que la cadena de ceros es el cero.
- Si un número tiene como exponente puros unos y de parte fraccionaria de la mantisa (lo que se escribe de la mantisa) puros ceros representa $+\infty$ o $-\infty$ dependiendo del signo del exponente.
- Si un número tiene puros unos en el exponente y la parte fraccionaria de la mantisa algo distinto de puros ceros entonces se dice que *No es un Número* lo que se denota como NaN.
- Si un número tiene como exponente puros ceros y de parte fraccionaria de mantisa algo distinto de puros ceros entonces se dice que es un número no-normalizado (*de-normalized*) y entonces no se asume la regla de que hay un 1 precediendo la parte fraccionaria de la mantisa que está explícita en el número. Con esta convención el cero puede considerarse como un caso especial de número no-normalizado.

Entonces para representar el infinito se usa ($7F800000_{16}$) el cero es puros ceros. el positivo más pequeño es: 00000001_{16} (nótese que es no-normalizado) que es $(2^{-23}) \times 2^{-127} \approx 1.401298 \times 10^{-45}$ y el más grande $7F7FFFFF_{16}$ que sería $(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{-23}}) \times 2^{127} \approx 3.402823 \times 10^{38}$.

El formato de IEEE de precisión doble usa 64 bits para representar un número: uno para el signo de la mantisa, 11 para el exponente que se representa en exceso a 1023 y los restantes 52 bits para la mantisa normalizada omitiendo el uno más significativo a la izquierda del punto. Se establecen convenciones similares a las de IEEE de precisión simple para representar números especiales, el infinito y los no-normalizados.

El formato de precisión simple corresponde al tipo `float` de C y el de precisión doble al tipo `double`, evidentemente. Hay una explicación detallada del formato de IEEE en la página de Steve Hollasch[1]

[1] `http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html`