Representación de números por ordenador

Álvaro Fernández Barrero

10/10/2025

Desde tiempos ancestrales el ser humano ha tenido diversas formas para representar números. Una de ellas y las más míticas eran los números romanos, los cuales tenían la forma sustractiva que todo conocemos o la aditiva, en la cual siempre ibas sumando dando igual la posición de los números y pudiendo poner hasta cuatro mismos números seguidos, aunque no era tan usado por tener números demasiados extensos de escribir.

En la antigua mesopotamia, los babilonios contaban con una forma de representar números que te permitían escribir hasta 60 símbolos, contando así con una base 60 (sexagesimal) y de la que viene el motivo por el que un minuto son 60 segundos y una hora son 60 minutos. A día de hoy contamos con un sistema decimal (de base 10) que cuenta con 9 números (0-9).

La forma de contar que tenemos, parecida a la que tenían los babilonios, se basa en tener unos símbolos que vamos cambiando y, cuando nos quedamos sin más símbolos, iniciamos de nuevo con el primero y a la izquierda ponemos el primero. A la cantidad de símbolos que se cuentan se le llama "base".

Viendo todo esto, los matemáticos dieron con el *teorema fundamental de la numeración*, una forma de representar números de cualquier sistema con cualquier base, teniendo la siguiente forma:

$$x = \sum_{\mu=-\xi}^{\zeta} x_\mu \beta^\mu, \quad \xi, \zeta \in \mathbb{N}_0$$

En dicha fórmula, x es el número en cuestión, x_μ es el dígito μ-*ésimo* del número, ξ representa la cantidad de decimales que tiene x, ζ la cantidad de dígitos enteros contando desde 0 y β la base en cuestión (la cantidad de símbolos con la que cuenta nuestro sistema).

Gracias a dicha fórmula, podemos tener sistemas de numeración bastante extraños como un sistema numérico de base φ, γ o incluso π.

Sin embargo, cuando hablamos de circuitos como puede ser los que tiene un ordenador, solo contamos con dos posibilidades, pasa o no pasa corriente, por lo que es justo pensar que a cada uno le podemos asignar un símbolo, el 0 y el 1, contando así con un sistema binario, dándole en la fórmula anterior el valor de $\beta = 2$.

Con todo esto, ya tenemos una forma increíble para representar cualquier número en cualquier dispositivo, ya que mediante pasar o no pasar corriente por sitios en concreto (0 y 1), podemos convertirlo con el *teorema fundamental de la numeración* a un sistema decimal que nosotros podamos comprender.

Cuando hablamos de ingeniería, informática o, en general, circuitos eléctricos, no podemos contar con infinitas cifras como contamos en matemáticas, tienes un máximo ya que no se puede pasar o no corriente por infinitos puntos. En estos casos, cada dígito binario tiene el nombre de *bit (b)* y, para hablar de números más grandes de forma más sencilla, podemos decir que un *byte (B)* son 8 bits, un *kilobyte (KB)* son 1024 bytes, un *Megabyte (MB)* son 1024 kilobytes, un *gigabyte (GB)* son 1024 megabytes, un *terabyte (TB)* son 1024 gigabytes...

Desde aquí, se puede entender el *álgebra de Boole*, un tipo de álgebra destinada a trabajar con bits y que es de vital importancia ya que te ofrece operaciones matemáticas fundamentales cuando vamos a operar con éstos.

Siguiendo con esto, nos vamos a encontrar un problema y es que, cuando estamos hablando en términos matemáticos, todo desde aquí se hace trivial, pero no tenemos una forma de representar decimales o números negativos con circuitos.

Para el caso de los números negativos es muy sencillo, ya que el primer bit de todos, el más significativo, el que más a la izquierda se encuentra se utiliza para el signo; quedando que si el primer bit es 1, estamos hablando de un número negativo, pero en caso de ser 0 será positivo. Sin embargo, al ser el más significativo, perdemos un gran rango numérico para representar con esa cantidad de bits, por lo que cuando estamos programando con todo esto, los lenguajes de programación de más bajo nivel nos suelen dotar con la posibilidad de decidir si queremos destinar ese bit para el signo o no añadiendo la palabra *unsigned* si lo queremos guardar en C/C++.

Esto permite una cosa que, matemáticamente, no debería existir, y es que con esto podemos crear ceros positivos y negativos, conceptos que realmente no existen porque solo está el cero, no tiene signo. Además, este método es poco eficiente, ya que a la hora de operar con estos números tenemos que tener en cuenta el signo de forma independiente.

Por ejemplo, supongamos que tenemos 1 byte para representar el número $5_{(10}$, tenemos el número $00000101_{(2}$, y si queremos representar el número $-5_{(10}$, tendremos $10000101_{(2}$. En el caso del 0 que se explicaba antes, $0_{(10} = 00000000_{(2}$, pero también se puede poner como $10000000_{(2}$.

Poco a poco se fue revisando y refinando esta interpretación de números negativos. Se mejora la eficiencia dejando el primer bit para el signo y sacando el *complemento a 1* a dicho número (invertir todos los bits: aquellos que eran 1 pasan a ser un 0 y viceversa). Al sacar el complemento a 1 automáticamente también se cambia el primer bit, por lo que me aporta el signo correcto.

Sin embargo, tengo aún el problema de contar con un 0 negativo y otro positivo y, aunque ya no es tan complicado operar con estos números negativos, sigue siendo una tarea algo complicada.

Aquí, el $-5_{(10}$ se representa $11111010_{(2}$ que, como se aprecia, está cada bit cambiado.

Posteriormente, a estos números negativos ya no solamente se le sacaba el complemento a 1, sino que además le sumaban uno al mismo, o lo que se conoce como *complemento a 2*. Esto por fin quita el problema del doble 0 y hace que las operaciones con números negativos sea más sencilla ya que se puede interpretar como suma que es, por lo que sabemos, la definición de resta:

$$a - b := a + (-b)$$

En este caso, ese $-5_{(10}$ es $11111011_{(2}$.

Para representar decimales es un poco más complicado que simplemente definir si un número es positivo o negativo. Para éstos, viene bien conocer lo que es la *notación científica*.

Cuando trabajamos con la ciencia, vemos números muy extensos, como lo podría ser la constante de gravitación universal G (0,0000000000667430), la unidad de un mol (6022140760 00000000000000) o la permitividad del vacío ε_0 (0,00000000000885418781 88). Estos números son horriblemente complejos de maniobrar o incluso de escribir, pero por suerte contamos con la notación científica que, aprovechando el *teorema fundamental de la numeración*, obtiene solo las cifras significativas y las multiplica por diez elevado a la cantidad de espacios que se debe desplazar la coma, quedando como resultado:

$$G = 6.67430 \cdot 10^{-11} \quad \varepsilon_0 = 8.8541878188 \cdot 10^{-12} \quad 1\ mol = 6,02214076 \cdot 10^{23}$$

Visualmente ya se ve más atractivo, pero además es más fácil de operar. Imagina que tenemos dos números como 3500000 y 0,00000065 y queremos multiplicarlos. Puede resultar una cosa muy tediosa, pero si lo pasamos a notación científica tenemos números como $3,5 \cdot 10^6$ y $6,5 \cdot 10^{-8}$ simplifica mucho la operación:

$$3,5 \cdot 10^6 \cdot 6,5 \cdot 10^{-8} = 3,5 \cdot 6,5 \cdot 10^6 \cdot 10^{-8} = 3,5 \cdot 6,5 \cdot 10^6 \cdot 10^{-8} = 3,5 \cdot 6,5 \cdot 10^{6-8}$$

$$= 3,5 \cdot 6,5 \cdot 10^{-2} = 22,75 \cdot 10^{-2} = 0,2275$$

Como se puede ver, la notación científica facilita muchísimo grandes cálculos. En programación, se puede usar la notación *ne+m* y *ne-m*, lo que corresponde que:

$$ne + m = n \cdot 10^m \quad ne - m = n \cdot 10^{-m}$$

Todo este concepto se puede generalizar para cualquier sistema de numeración con cualquier base de la siguiente manera:

$$m \cdot \beta^\eta$$

Aquí, m es la primera parte llamada *"mantissa"*, β es la base en cuestión y η es la cantidad de desplazamientos que debe hacer la coma de la mantissa. La mantissa además debe cumplir que solo debe haber un número distinto de cero delante de la coma.

Con los ejemplos dados, nos podemos dar cuenta que realmente representar un número en notación científica nos puede ahorrar muchos quebraderos de cabeza y vamos a poder manejar números más sencillos. Esta simplificación es la que también podemos usar a la hora de representar unos números con decimales en electrónica: Si tenemos n bits para

representar un número, el primero lo reservamos para el signo, aproximadamente $\frac{1}{3}$ para el exponente y el resto para la mantissa. Como hemos fijado por regla general que solo debe haber un número delante de la coma y, como estamos en un sistema binario, solo puede ser un 1, por lo que se puede obviar y eliminar para ganar un bit más para la mantissa.

Esto genera un problema, porque unos números tan comunes para nosotros como lo podría ser el 0,1 son fáciles de calcular ya que ese 1 en los decimales es $1 \cdot 10^{-1}$ o $1 \cdot \frac{1}{10}$, pero en el caso binario, no es tan fácil, ya que no tienen un 10 como el sistema decimal, sino que para este contexto sería $1 \cdot 2^{-1}$, lo que ya no representa el 0,1, sino que representa $\frac{1}{2}$. Para poder representar el 0,1 en binario tenemos $0{,}0\overline{0011}_{(2}$, lo cual matemáticamente no es un problema el contar con dígitos infinitos, pero computacionalmente no es demasiado exacto. Es por esto que si en un lenguaje de programación hacer la operación 0,1+0,2, el resultado no te va a dar 0,3 exacto, sino 0.30000000000000004.

No obstante, para la gran mayoría de casos, el no tener esa exactitud no es un problema ya que el error es despreciable.

Este problema no es como el de los números negativos ya que, para solucionar esto, requerimos de infinitos bits, cosa que no es realista en lo absoluto.

Personalmente se me hace especialmente curioso cómo los ordenadores de ahora son capaces de hacer simulaciones tanto gráficas como físicas impresionantes y ultra realistas pero, sin embargo, no son capaces de operar bien con decimales sin que haya un error.