Representación de la información

10 de noviembre de 2017

Representación de la Información

Veremos de qué manera puede ser tratada mediante computadoras la información correspondiente a números, textos, imágenes y otros datos. Necesitaremos conocer las formas de representación de datos, y comenzaremos por los datos numéricos.

Representación de datos numéricos

Hemos visto ejemplos de sistemas de numeración: en base 6, en base 10, o decimal, en base 2, o binario, en base 16, o hexadecimal, y en base 8, u octal; y sabemos convertir la representación de un número en cada una de estas bases, a los sistemas en las demás bases. Sin embargo, aún nos falta considerar la representación numérica de varios casos importantes:

- Hemos utilizado estos sistemas para representar únicamente números **enteros**. Nos falta ver de qué manera representar números racionales, es decir aquellos que tienen una parte fraccionaria (los "decimales").
- Además estos enteros han sido siempre **no negativos**, es decir, sabemos representar únicamente el 0 y los naturales. Nos falta considerar los negativos.
- Por otra parte, no nos hemos planteado el problema de la **cantidad de dígitos**. Idealmente, un sistema de numeración puede usar infinitos dígitos para representar números arbitrariamente grandes. Si bien esto es matemáticamente correcto, las computadoras son objetos físicos que tienen unas ciertas limitaciones, y con ellas no es posible representar números de infinita cantidad de dígitos.

En esta parte de la unidad mostraremos sistemas de representación utilizados en computación que permiten tratar estos problemas.

Clasificación de los números

Es conveniente repasar la clasificación de los diferentes conjuntos de números y conocer las diferencias importantes entre éstos. Los títulos en el cuadro (tomado de Wikipedia) son referencias a los artículos enciclopédicos sobre cada uno de esos conjuntos.

- Números complejos
- Complejos
- Reales
- Racionales
- Enteros
- Naturales
- Uno
- Naturales primos
- Naturales compuestos
- Cero
- Enteros negativos
- Fraccionarios
- Fracción propia
- Fracción impropia
- Irracionales
- Irracionales algebraicos
- Trascendentes
- Imaginarios

Preguntas

- El **cero**, ¿es un natural?
- ¿Existen números naturales negativos? ¿Y racionales negativos?
- ¿Es correcto decir que un racional tiene una parte decimal que es, o bien finita, o bien periódica?
- ¿Puede haber dos expresiones diferentes para el mismo número, en el mismo sistema de numeración decimal?
- El número $0.9999\ldots$ con 9 periódico, y el número 1, ¿son dos números diferentes o el mismo número? Si son diferentes, ¿qué número se encuentra entre ellos dos?
- El número 1 es a la vez natural y entero. ¿Por qué no puede haber un número que sea a la vez racional e irracional?
- ¿Por qué jamás podremos computar la sucesión completa de decimales de π?

Datos enteros

Veremos un sistema de representación de datos no negativos, llamado **sin signo**, y tres sistemas de representación de datos numéricos enteros, llamados **signo-magnitud**, **complemento a 2** y **notación en exceso**.

Datos fraccionarios

Para representar fraccionarios consideraremos los sistemas de **punto fijo** y **punto flotante**.

Rango de representación

Cada sistema de representación de datos numéricos tiene su propio **rango de representación** (que podemos abreviar **RR**), o intervalo de números representables. Ningún número fuera de este rango puede ser representado en dicho sistema. Conocer este intervalo es importante para saber con qué limitaciones puede enfrentarse un programa que utilice alguno de esos sistemas.

El rango de los números representados bajo un sistema está dado por sus **límites inferior y superior**, que definen qué zona de la recta numérica puede ser representada. Como ocurre con todo intervalo numérico cerrado, el rango de representación puede ser escrito como $[a, b]$, donde a y b son sus límites inferior y superior, respectivamente.

Por la forma en que están diseñados, algunos sistemas de representación sólo pueden representar números muy pequeños, o sólo positivos, o tanto negativos como positivos. En general, el RR **será más grande cuantos más dígitos binarios**, o bits, tenga el sistema. Sin embargo, el RR depende también de la forma como el sistema **utilice** esos dígitos binarios, ya que un sistema puede ser más o menos **eficiente** que otro en el uso de esos dígitos, aunque la cantidad de dígitos sea la misma en ambos sistemas.

Por lo tanto, decimos que el rango de representación depende a la vez de la **cantidad de dígitos** y de la **forma de funcionamiento** del sistema de representación.

Representación sin signo SS(k)

Consideremos primero qué ocurre cuando queremos representar números enteros **no negativos** (es decir, **positivos o cero**) sobre una cantidad fija de bits.

En el sistema **sin signo**, simplemente usamos el sistema binario de numeración, tal como lo conocemos, **pero limitándonos a una cantidad fija** de dígitos binarios o bits. Podemos entonces abreviar el nombre de este sistema como **SS(k)**, donde k es la cantidad fija de bits, o ancho, de cada número representado.

Rango de representación de SS(k)

¿Cuál será el rango de representación? El **cero** puede representarse, así que el límite inferior del rango de representación será 0. Pero ¿cuál será el límite superior? Es decir, si la cantidad de dígitos binarios en este sistema es k, ¿cuál es el número más grande que podremos representar?

Podemos estudiarlo de dos maneras.

1. Usando combinatoria

Contemos cuántos números diferentes podemos escribir con k dígitos binarios. Imaginemos un número binario cualquiera con k dígitos. El dígito de más a la derecha tiene únicamente dos posibilidades (0 o 1). Por cada una de éstas hay nuevamente dos posibilidades para el siguiente hacia la izquierda (lo que da las cuatro posibilidades 00, 01, 10, 11). Por cada una de éstas, hay dos posibilidades para el siguiente (dando las ocho posibilidades 000, 001, 010, 011, 100, 101, 110, 111), etc., y así hasta la posición k. No hay más posibilidades. Como hemos multiplicado 2 por sí mismo k veces, la cantidad de números que se pueden escribir es 2^k. Luego, el número más grande posible es $2^k - 1$. (**Pregunta**: ¿Por qué $2^k - 1$ y no 2^k?).

2. Usando álgebra

El número más grande que podemos representar en un sistema sin signo a k dígitos es, seguramente, aquel donde todos los k dígitos valen **1**. La Expresión General que hemos visto nos dice que si un número n está escrito en base 2, **con k dígitos**, entonces

$$n = x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0$$

y, si queremos escribir el más grande de todos, deberán ser todos los x_i iguales a 1. (**Pregunta**: ¿Por qué si el número n tiene k dígitos binarios, el índice del más significativo es $k - 1$ y no k?)

Esta suma vale entonces

$$\begin{aligned} x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0 = \\ = 1 \times 2^{k-1} + ... + 1 \times 2^1 + 1 \times 2^0 = \\ = 2^{k-1} + ... + 2^1 + 2^0 = \\ = 2^k - 1 \end{aligned}$$

Usando ambos argumentos hemos llegado a que el número más grande que podemos representar con k dígitos binarios es $2^k - 1$. Por lo tanto, **el rango de representación de un sistema sin signo a k dígitos, o SS(k), es** $[0, 2^k - 1]$. Todos los números representables en esta clase de sistemas son **positivos o cero**.

Ejemplos

- Para un sistema de representación sin signo a 8 bits: $[0, 2^8 - 1] = [0, 255]$
- Con 16 bits: $[0, 2^{16} - 1] = [0, 65{,}535]$
- Con 32 bits: $[0, 2^{32} - 1] = [0, 4{,}294{,}967{,}295]$

Representación con signo

En la vida diaria manejamos continuamente números negativos, y los distinguimos de los positivos simplemente agregando un signo "menos". Representar esos datos en la memoria de la computadora no es tan directo, porque, como hemos visto, la memoria **solamente puede alojar ceros y unos**. Es decir, ¡no podemos simplemente guardar un signo "menos"! Lo único que podemos hacer es almacenar secuencias de ceros y unos.

Esto no era un problema cuando los números eran no negativos. Para poder representar, ahora, tanto números **positivos como negativos**, necesitamos cambiar la forma de representación. Esto quiere decir que una secuencia particular de dígitos binarios, que en un sistema sin signo tiene un cierto significado, ahora tendrá un significado diferente. Algunas secuencias, que antes representaban números positivos, ahora representarán negativos.

Veremos los **sistemas de representación con signo** llamados **Signo-magnitud (SM)**, **Complemento a 2 (C2)** y **Notación en exceso**.

Es importante tener en cuenta que **solamente se puede operar entre datos representados con el mismo sistema de representación**, y que el **resultado** de toda operación **vuelve a estar representado en el mismo sistema**.

Preguntas

- ¿Cuáles son los límites del rango de representación de un sistema de representación numérica?
- Un número escrito en un sistema de representación **con signo**, ¿es siempre negativo?
- ¿Para qué querríamos escribir un número positivo en un sistema de representación con signo?

Sistema de Signo-magnitud SM(k)

El sistema de **Signo-magnitud** no es el más utilizado en la práctica, pero es el más sencillo de comprender. Se trata simplemente de utilizar un bit (el de más a la izquierda) para representar el **signo**. Si este bit tiene valor 0, el número representado es positivo; si es 1, es negativo. Los demás bits se utilizan para representar la **magnitud**, es decir, el **valor absoluto** del número en cuestión.

Ejemplos

- $7_{(10} = 00000111_{(2}$
- $-7_{(10} = 10000111_{(2}$

Como estamos reservando un bit para expresar el signo, ese bit ya no se puede usar para representar magnitud; y como el sistema tiene una cantidad de bits fija, el RR ya no podrá representar el número máximo que era posible con el sistema **sin signo**.

Rango de representación de SM(k)

- En todo número escrito en el sistema de signo-magnitud a k bits, ya sea positivo o negativo, hay un bit reservado para el signo, lo que implica que quedan $k-1$ bits para representar su valor absoluto.
- Siendo un valor absoluto, estos $k-1$ bits representan un número **no negativo**. Además este número está representado con el sistema **sin signo** sobre $k-1$ bits, es decir, SS(k-1).
- Este número no negativo en SS(k-1) tendrá un valor máximo representable que coincide con el **límite superior** del rango de representación **de SS(k-1)**.
- Sabemos que el rango de representación de SS(k) es $[0, 2^k - 1]$. Por lo tanto, el rango de SS(k-1), reemplazando, será $[0, 2^{k-1} - 1]$.
- Esto quiere decir que el número representable en SM(k) cuyo valor absoluto es máximo, es $2^{k-1} - 1$. Por lo tanto éste es el límite superior del rango de representación de SM(k).
- Pero en SM(k) también se puede representar su opuesto negativo, simplemente cambiando el bit más alto por 1. El opuesto del máximo positivo representable es a su vez el número más pequeño, negativo, representable: $-(2^{k-1} - 1)$.

Con lo cual hemos calculado tanto el límite inferior como el superior del rango de representación de SM(k), que, finalmente, es $[-(2^{k-1} - 1), 2^{k-1} - 1]$.

Limitaciones de Signo-Magnitud

Si bien **SM(k)** es simple, no es tan efectivo, por varias razones:

- Existen dos representaciones del 0 ("positiva" y "negativa"), lo cual desperdicia un representante.
- Esto acorta el rango de representación.
- La aritmética en SM no es fácil, ya que cada operación debe comenzar por averiguar si los operandos son positivos o negativos, operar con los valores absolutos y ajustar el resultado de acuerdo al signo reconocido anteriormente.
- El problema aritmético se agrava con la existencia de las dos representaciones del cero: cada vez que un programa quisiera comparar un valor resultado de un cómputo con 0, debería hacer **dos** comparaciones.

Por estos motivos, el sistema de SM dejó de usarse y se diseñó un sistema que eliminó estos problemas, el sistema de **complemento a 2**.

Sistema de Complemento a 2

Para comprender el sistema de complemento a 2 es necesario primero conocer la **operación** de complementar a 2.

Operación de Complemento a 2

La **operación** de complementar a 2 consiste aritméticamente en obtener el **opuesto** de un número (el que tiene el mismo valor absoluto pero signo opuesto).

Para obtener el complemento a 2 de un número escrito en base 2, **se invierte cada uno de los bits (reemplazando 0 por 1 y viceversa) y al resultado se le suma 1**.

Otra forma

Otro modo de calcular el complemento a 2 de un número en base 2 es **copiar los bits, desde la derecha, hasta el primer 1 inclusive; e invertir todos los demás a la izquierda**.

Propiedad fundamental

El resultado de esta operación, C2(a), es el opuesto del número original a, y por lo tanto tiene la propiedad de que a y C2(a) suman 0:

$$C2(a) + a = 0$$

Comprobación

Podemos comprobar si la complementación fue bien hecha aplicando la **propiedad fundamental** del complemento. Si, al sumar nuestro resultado con el número original, no obtenemos 0, corresponde revisar la operación.

Ejemplos

- Busquemos el complemento a 2 de 111010. Invirtiendo todos los bits, obtenemos 000101. Sumando 1, queda 000110.
- Busquemos el complemento a 2 de 0011. Invirtiendo todos los bits, obtenemos 1100. Sumando 1, queda 1101.
- Comprobemos que el resultado obtenido en el último caso, 1101, es efectivamente el opuesto de 0011: $0011 + 1101 = 0$.

Representación en Complemento a 2

Ahora que contamos con la **operación de complementar a 2**, podemos ver cómo se construye el **sistema de representación en Complemento a 2**.

Para representar un número a en complemento a 2 a k bits, comenzamos por considerar su signo:

- Si a es positivo o cero, lo representamos como en SM(k), es decir, lo escribimos en base 2 a k bits.
- Si a es negativo, tomamos su valor absoluto y lo complementamos a 2.

Ejemplos

- Representemos el número 17 en complemento a 2 con 8 bits. Como es positivo, lo escribimos en base 2, obteniendo 00010001, que es 17 en notación complemento a 2 con 8 bits.
- Representemos el número -17 en complemento a 2 con 8 bits. Como es negativo, escribimos su valor absoluto en base 2, que es 00010001, y lo complementamos a 2. El resultado final es 11101111 que es -17 en notación complemento a 2 con 8 bits.

Conversión de C2 a base 10

Para convertir un número n, escrito en el sistema de complemento a 2, a decimal, lo primero es determinar el signo. Si el bit más alto es 1, n es negativo. En otro caso, n es positivo. Utilizaremos esta información enseguida.

- Si n es positivo, se interpreta el número como en el sistema sin signo, es decir, se utiliza la Expresión General para hacer la conversión de base como normalmente.
- Si n es negativo, se lo complementa a 2, obteniendo el opuesto de n. Este número, que ahora es positivo, se convierte a base 10 como en el caso anterior; y finalmente se le agrega el signo "-" para reflejar el hecho de que es negativo.

Ejemplos

- Convertir a decimal $n = 00010001$. Es positivo, luego, aplicamos la Expresión General dando $17_{(10}$.
- Convertir a decimal $n = 11101111$. Es negativo; luego, lo complementamos a 2 obteniendo 00010001. Aplicamos la Expresión General obteniendo $17_{(10}$. Como n era negativo, agregamos el signo menos y obtenemos el resultado final $-17_{(10}$.

RR de C2 con k bits

La forma de utilizar los bits en el sistema de complemento a 2 permite recuperar un representante que estaba desperdiciado en SM.

El rango de representación del sistema complemento a 2 sobre k bits es $[-(2^{k-1}), 2^{k-1} - 1]$. El límite superior del RR de C2 es el mismo que el de SM, pero el **límite inferior** es menor; luego el RR de C2 es mayor que el de SM.

El sistema de complemento a 2 tiene otras ventajas sobre SM:

- El cero tiene una única representación, lo que facilita las comparaciones.

- Las cuentas se hacen bit a bit, en lugar de requerir comprobaciones de signo.
- El mecanismo de cálculo es eficiente y fácil de implementar en hardware.
- Solamente se requiere diseñar un algoritmo para **sumar**, no uno para sumar y otro para restar.

Comparando rangos de representación

Diferentes sistemas, entonces, tienen diferentes rangos de representación. Si construimos un cuadro donde podamos comparar los rangos de representación **sin signo, signo-magnitud y complemento a 2** para una misma cantidad de bits, veremos que todas las combinaciones de bits están utilizadas, sólo que de diferente forma.

El cuadro comparativo para cuatro bits mostrará que las combinaciones 0000...1111 representan los primeros 16 números no negativos para el sistema sin signo, mientras que esas mismas combinaciones tienen otro significado en los sistemas con signo. En éstos últimos, una misma combinación con el bit más significativo en 1 siempre es negativa, pero el orden en que aparecen esas combinaciones es diferente entre SM y C2.

Por otro lado, los números positivos quedan representados por combinaciones idénticas en los tres sistemas, hasta donde lo permite el rango de representación de cada uno.

Si descartamos el bit de signo y consideramos sólo las magnitudes, los números negativos en SM aparecen con sus magnitudes crecientes alejándose del 0, mientras que en C2 esas magnitudes comienzan en cero al representar el negativo más pequeño posible y crecen a medida que se acercan al cero.

Aritmética en C2

Una gran ventaja que aporta el sistema en Complemento a 2 es que los diseñadores de hardware no necesitan implementar algoritmos de resta además de los de la suma. Cuando se necesita efectuar una resta, **se complementa el sustraendo** y luego se lo **suma** al minuendo. Las computadoras no restan: siempre suman.

Por ejemplo, la operación $9-8$ se realiza como $9+(-8)$, donde (-8) es el complemento a 2 de 8.

Preguntas

- Un número en complemento a 2, ¿tiene siempre su bit más a la izquierda en 1?
- El complemento a 2 de un número, es decir, **C2(x)**, ¿es siempre un número negativo?
- ¿Quién es **C2(0)**?

- ¿Cuánto vale **C2(C2(x))**? Es decir, ¿qué pasa si complemento a 2 el complemento a 2 de x?
- ¿Cuánto vale **x + C2(x)**? Es decir, ¿qué pasa si sumo a x su propio complemento a 2?
- ¿Cómo puedo verificar si calculé correctamente un complemento a 2?

Overflow o desbordamiento en C2

En todo sistema de ancho fijo, la suma de **dos números positivos, o de dos números negativos** puede dar un resultado que sea imposible de representar debido a las limitaciones del rango de representación. Este problema se conoce como desbordamiento, u *overflow*. Cuando ocurre una situación de overflow, el resultado de la operación **no es válido** y debe ser descartado.

Si conocemos los valores en decimal de dos números que queremos sumar, usando nuestro conocimiento del rango de representación del sistema podemos saber si el resultado quedará dentro de ese rango, y así sabemos, de antemano, si ese resultado será válido. Pero las computadoras no tienen forma de conocer a priori esta condición, ya que todo lo que tienen es la representación en C2 de ambos números. Por eso necesitan alguna forma de detectar las situaciones de overflow, y el modo más fácil para ellas es comprobar los dos últimos bits de la fila de bits de acarreo o *carry*.

El último bit de la fila de carry, el que se posiciona en la última de las k columnas de la representación, se llama *carry-in*. El siguiente bit de carry, el que ya no puede acarrearse sobre ningún dígito válido porque se han rebasado los k dígitos de la representación, se llama el *carry-out*.

- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **iguales**, entonces la computadora detecta que el resultado no ha desbordado y que **la suma es válida**. La operación de suma se ha efectuado exitosamente.
- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **diferentes**, entonces la computadora detecta que el resultado ha desbordado y que **la suma no es válida**. La operación de suma no se ha llevado a cabo exitosamente, y el resultado debe ser descartado.

Suma sin overflow

Siguiendo atentamente la secuencia de bits de carry podemos detectar, igual que lo hace la computadora, si se producirá un desbordamiento. En el caso de la operación $23 + (-9)$, el resultado (que es 14) cae dentro del rango de representación, y esto se refleja en los bits de *carry-in* y de *carry-out*, cuyos valores son iguales.

Suma con overflow

En el caso de la operación $123 + 9$ en C2 a 8 bits, el resultado (que es 132) cae fuera del rango de representación. Esto se refleja en los bits de *carry-in* y de *carry-out*, que son diferentes. El resultado no es válido y debe ser descartado.

Preguntas

- ¿Qué condición sobre los bits de carry permite asegurar que **no habrá** overflow?
- ¿Para qué sistemas de representación numérica usamos la condición de detección de overflow?
- ¿Puede existir overflow al sumar dos números de diferente signo?
- ¿Qué condición sobre los bits **de signo** de los operandos permite asegurar que **no habrá** overflow?
- ¿Puede haber casos de overflow al sumar dos números negativos?
- ¿Puede haber casos de overflow al restar dos números?

Extensión de signo en C2

Para poder efectuar una suma de dos números, ambos operandos deben estar representados en el mismo sistema de representación.

- Una suma de dos operandos donde uno esté, por ejemplo, en SM y el otro en C2, no tiene sentido aritmético.
- Además, la cantidad de bits de representación debe ser la misma.

En una suma en C2, si uno de los operandos estuviera expresado en un sistema con menos bits que el otro, será necesario convertirlo al sistema del otro (**extenderlo**) y operar con ambos en ese sistema de mayor ancho.

Si el operando en el sistema de menor ancho es positivo, la extensión se realiza simplemente **completando con ceros a la izquierda** hasta obtener la cantidad de dígitos del otro sistema. Si el operando del menor ancho es negativo, la extensión de signo se hace **agregando unos**.

Ejemplos

- $A + B = 00101011_{(2} + 00101_{(2}$
  - A está en C_2^8 y B en $C_2^5 \rightarrow$ llevar ambos a C_2^8
  - Se completa B (positivo) como $00000101_{(2}$
- $A + B = 1010_{(2} + 0110100_{(2}$
  - A está en C_2^4 y B en $C_2^7 \rightarrow$ llevar ambos a C_2^7
  - Se completa A (negativo) como $1111010_{(2}$

Notación en exceso o *bias*

En un sistema de notación en exceso, se elige un intervalo $[a, b]$ de enteros a representar, y todos los valores dentro del intervalo se representan con una secuencia de bits de la misma longitud.

La cantidad de bits deberá ser la necesaria para representar todos los enteros del intervalo, inclusive los límites, y por lo tanto estará en función de la longitud del intervalo. Un intervalo $[a, b]$ de enteros, con sus límites incluidos, comprende exactamente $n = b - a + 1$ valores. Esta longitud del intervalo debe ser cubierta con una cantidad k de bits suficiente, lo cual obliga a que $2^k \geq n$. Supongamos que n sea una potencia de 2 para facilitar las ideas, de forma que $2^k = n$.

Las 2^k secuencias de k bits, ordenadas como de costumbre según su valor aritmético, se aplican a los enteros en $[a, b]$, uno por uno. Es decir, si usamos 3 bits, las secuencias serán 000, 001, 010, ... hasta 111; y los valores representados serán respectivamente:

- 000 = a
- 001 = $a + 1$
- 010 = $a + 2$
- ...
- 111 = b

Notemos que tanto a como b pueden ser **negativos**. Así podemos representar intervalos de enteros arbitrarios con secuencias de k bits, lo que nos vuelve a dar un sistema de representación con signo.

Con este método no es necesario que el bit de orden más alto represente el signo. Tampoco que el intervalo contenga la misma cantidad de números negativos que positivos o cero, aunque para la mayoría de las aplicaciones es lo más razonable.

El sistema en exceso se utiliza como componente de otro sistema de representación más complejo, la representación en punto flotante.

Conversión entre exceso y decimal

Una vez establecido un sistema en exceso que representa el intervalo $[a, b]$ en k bits:

- Para calcular la secuencia binaria que corresponde a un valor decimal d, a d **le restamos** a y luego convertimos el resultado (que será **no negativo**) a **SS(k)**, es decir, a binario sin signo sobre k bits.
- Para calcular el valor decimal d representado por una secuencia binaria, convertimos la secuencia a decimal como en **SS(k)**, y al resultado (que será **no negativo**) le **sumamos** el valor de a.

Ejemplos

Representemos en sistema en exceso el intervalo $[10, 25]$ (que contiene $25-10+1 = 16$ enteros). Como necesitamos 16 secuencias binarias, usaremos 4 bits que producirán las secuencias 0000, 0001, ..., 1111.

- Para calcular la secuencia que corresponde al número 20, hacemos $20-10=10$ y el resultado será la secuencia **1010**.
- Para calcular el valor decimal que está representando la secuencia **1011**, convertimos 1011 a decimal, que es 11, y le sumamos 10; el resultado es 21.

Representemos en sistema en exceso el intervalo $[-3,4]$ (que contiene $4-(-3)+1=8$ enteros). Como necesitamos 8 secuencias binarias, usaremos 3 bits que producirán las secuencias 000, 001, ..., 111.

- Para calcular la secuencia que corresponde al número 2, hacemos $2-(-3)=5$ y el resultado será la secuencia **101**.
- Para calcular el valor decimal que está representando la secuencia **011**, convertimos 011 a decimal, que es 3, y le sumamos -3; el resultado es 0.

Preguntas sobre Notación en Exceso

- Dado un valor decimal a representar, ¿cómo calculamos el binario?
- Dado un binario, ¿cómo calculamos el valor decimal representado?
- El sistema en exceso ¿destina un bit para representar el signo?
- ¿Se puede representar un intervalo que no contenga el cero?
- ¿Cómo se comparan dos números en exceso para saber cuál es el mayor?

Representación de fraccionarios

Racionales

Los números fraccionarios son aquellos **racionales** que no son enteros. Se escriben como una razón, fracción o cociente de dos enteros. Por ejemplo, $3/4$ y $-12/5$ son números fraccionarios. El signo de división que usamos para escribir las fracciones tiene precisamente ese significado aritmético: si hacemos la operación de división correspondiente entre numerador y divisor de la fracción, obtenemos la forma decimal del mismo número, con **una parte entera y una parte decimal**. Así, por ejemplo, $3/4$ también puede escribirse como $0{,}75$, y $-12/5$ como $-2{,}4$. Estas dos formas son equivalentes. En los racionales, la parte decimal es **finita** o **periódica**.

Aproximación racional a los irracionales

Por otro lado, existen números reales que no son racionales, en el sentido de que no existe una razón, fracción o cociente que les sea igual, pero también pueden escribirse como decimales con una parte entera y una parte decimal. Estos son los **irracionales**. Los irracionales pueden expresarse sintéticamente como el resultado de alguna operación (como cuando escribimos $\sqrt{2}$) o en su forma decimal. Sin embargo,

tienen la característica de que su desarrollo decimal **es infinito** no periódico, por lo cual siempre que escribimos un irracional por su desarrollo decimal, en realidad estamos **truncando** ese desarrollo a alguna porción inicial. Jamás podremos escribir la sucesión completa de decimales.

De manera que, al escribir irracionales en su forma decimal, en realidad siempre tratamos con **aproximaciones racionales** a esos irracionales. Por ejemplo, $3{,}14$ y $3{,}1459$ son aproximaciones racionales al verdadero valor irracional de π, cuya parte decimal tiene infinitos dígitos.

Coma o punto decimal

Al escribir un número con cifras decimales en nuestro sistema numérico habitual de base 10, usamos una marca especial para separar la parte entera de la decimal: es la **coma o punto decimal**. En el desarrollo decimal, la coma o punto decimal señala el lugar donde los exponentes de la base en el desarrollo de potencias de 10 **se hacen negativos**. Cuando queremos representar números fraccionarios con computadoras, nos vemos en el problema de representar este signo especial.

Podemos trasladar la idea de coma o punto decimal al sistema binario. Si extendemos la Expresión General con exponentes negativos, podemos escribir números fraccionarios en base 2.

Fraccionario en base 2 a decimal

Si encontramos una expresión como $11{,}101_{(2}$, la Expresión General extendida nos dice cómo obtener su valor en base 10:

$$11{,}101_{(2} =$$

$$1 \times 2^1 + 1 \times 2^0 +$$

$$1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} =$$

$$2 + 1 + 0{,}5 + 0 + 0{,}125 =$$

$$3{,}625$$

Otra manera

Otra manera de obtener el valor decimal de un número fraccionario n en base 2 consiste en utilizar el hecho de que cada vez que desplazamos el punto fraccionario un lugar hacia la derecha, estamos multiplicando n por 2, y viceversa, si desplazamos el punto hacia la izquierda, lo dividimos por 2.

El método consiste en:

- Identificar cuántas posiciones fraccionarias tiene n (llamémoslas k)

- Multiplicar n por 2^k obteniendo un **entero** en base 2
- Convertir el entero resultante a base 10, lo cual ya sabemos hacer
- Dividir el resultado por 2^k, obteniendo n en base 10

Con este método esencialmente estamos calculando el valor decimal de n **sin considerar el signo de coma fraccionaria** (es decir, imaginando que n fuera un entero); convirtiendo ese valor a decimal, y luego dividiendo el resultado en base 10 por 2^k para recuperar el valor original de n, sólo que ahora en base 10.

Ejemplo

El número $n = 11{,}101_{(2}$ tiene tres cifras decimales ($k = 3$). Lo convertimos en entero dejando $11101_{(2}$; averiguamos que este número en base 10 es 29; y finalmente dividimos 29 por 2^3. Concluimos que $n = 11{,}101_{(2} = 29/8 = 3{,}625$.

Decimal fraccionario a base 2

Para convertir un decimal con parte fraccionaria a base 2:

1. Se separan la parte entera (PE) y la parte fraccionaria (PF).
2. Se convierte la PE a base 2 separadamente.
3. La PF se multiplica por 2 y se toma la PE del resultado. Este dígito binario se agrega al resultado.
4. Se repite el paso anterior hasta llegar a 0, o hasta lograr la precisión deseada.

Ejemplo

Convirtamos el número $n = 3{,}625$ a base 2. Primero separamos parte entera (3) y parte fraccionaria (0.625).

Parte entera

La parte entera de n se convierte a base 2 como entero sin signo (dando $11_{(2}$).

Parte fraccionaria

Para calcular la parte fraccionaria binaria de n seguimos un procedimiento iterativo (es decir, que consta de pasos que se repiten).

La parte fraccionaria decimal de n se multiplica por 2: $0{,}625 \times 2 = 1{,}25$. Separamos este resultado a su vez en parte entera y parte fraccionaria. Guardamos la parte entera del resultado (que es 1) **y repetimos**, es decir, volvemos a multiplicar por 2 la parte fraccionaria recién obtenida (que es 0.25), separamos la parte entera, etc.

La sucesión de dígitos aparecidos como partes enteras durante este procedimiento servirán para **construir la parte fraccionaria** del resultado. Notemos que estos dígitos que aparecen solamente pueden ser ceros y unos, porque son la parte entera de $2 \times x$ con $x < 1$.

El procedimiento de separar, guardar, multiplicar, se repite hasta que la parte entera obtenida sea 0 (ya no tiene sentido seguir el procedimiento porque el resultado será siempre 0) o hasta que tengamos suficientes dígitos computados para nuestra aplicación.

El resultado final es la suma, en base 2, de la parte entera de n calculada anteriormente, más una parte fraccionaria construida con los dígitos que fueron apareciendo durante el procedimiento de duplicar la parte fraccionaria.

La conversión a base 2 del número $n = 3{,}625$ que buscábamos será $11_{(2} + 0{,}101_{(2} = 11{,}101_{(2}$.

Representación de punto fijo

¿Cómo aplicamos el método de conversión visto, de fraccionarios decimales a binarios y viceversa, en las computadoras? El problema es parecido al de almacenar el signo "menos": no podemos guardar en la memoria otra cosa que bits, de forma que habrá que establecer alguna convención para indicar dónde está el punto o coma fraccionaria.

A veces las computadoras utilizan sistemas **de punto fijo** para representar números con parte fraccionaria. Los sistemas de punto fijo establecen una cantidad de bits o **ancho** total (que llamaremos n) y una cantidad fija de bits para la parte fraccionaria (que llamaremos k). Todos los datos manipulados por la computadora tienen la misma cantidad $n - k$ de bits de parte entera y la misma cantidad k de bits de parte fraccionaria. Por ejemplo, la notación $PF(8, 3)$ denota un **sistema de punto fijo con 8 bits en total, de los cuales 3 son para la parte fraccionaria**.

Esta convención contiene toda la información necesaria. Al ser fijos los anchos de parte entera y fraccionaria, la computadora **puede tratar aritméticamente a todos los números como si fueran enteros**, sin preocuparse por partes enteras ni fraccionarias. Solamente habrá que utilizar la convención al momento de imprimir o comunicar un resultado. La impresora, o la pantalla, deberán mostrar un resultado con coma fraccionaria en el lugar correcto.

Sin embargo, todas las operaciones intermedias, entre datos expresados en punto fijo, habrán podido llevarse a cabo sin tener en cuenta el lugar de la coma. Dos números en punto fijo se sumarán como si los representados fueran dos enteros.

Ejemplo

- Supongamos que queremos computar $3{,}625 + 1{,}25$ en un sistema $PF(8, 3)$.
- Las conversiones de estos sumandos a fraccionarios binarios son, respectivamente, $11{,}101$ y $1{,}01$.
- Pero en la memoria se almacenarán como **00011101** y **00001010**. Nótese que al ser todas las partes fraccionarias del mismo ancho, quedan automáticamente "encolumnados" los invisibles puntos fraccionarios.

- La suma se efectuará bit a bit como si se tratara de enteros y será **00100111**.
- Si pedimos a la computadora que imprima este valor, aplicará la convención $PF(8,3)$ e imprimirá **00100.111**, o su interpretación en decimal, $4{,}875$, que es efectivamente $3{,}625 + 1{,}25$.

Decimal a PF(n,k)

Para representar un decimal fraccionario a, positivo o negativo, en notación de punto fijo en n lugares con k fraccionarios ($PF(n,k)$), necesitamos obtener su parte entera y su parte fraccionaria, y expresar cada una de ellas en la cantidad de bits adecuada a la notación. Para esto completaremos la parte entera con ceros a la izquierda hasta obtener $n-k$ dígitos, y completaremos la parte fraccionaria con ceros por la derecha, hasta obtener k dígitos. Una vez expresado así, lo tratamos como si en realidad fuera $a \times 2^k$, y por lo tanto, un entero.

- Si es positivo, calculamos la secuencia de dígitos binarios que expresan su parte entera y su parte fraccionaria, y escribimos ambas sobre la cantidad de bits adecuada.
- Si es negativo, consideramos su valor absoluto y procedemos como en el punto anterior. Luego complementamos a 2 como si se tratara de un entero.

Truncamiento

Al escribir la parte fraccionaria de un número a en k bits (porque ésta es la capacidad del sistema de representación de punto fijo con k dígitos fraccionarios), en el caso general estaremos **truncando** el desarrollo fraccionario. El número a podría tener otros dígitos diferentes de cero más allá de la posición k. Sin embargo, el sistema no permite representarlos, y esa información se perderá.

La consecuencia del truncamiento es la aparición de un **error de truncamiento** o pérdida de precisión. El número almacenado en el sistema PF(n,k) será una aproximación con k dígitos fraccionarios al número original a, y no estará representándolo con todos sus dígitos fraccionarios.

¿Cuál es el valor de este error de truncamiento, es decir, cuál es, cuantitativamente, la diferencia entre a y la representación en PF(n,k)? Si los primeros k dígitos del desarrollo fraccionario real de a se han conservado, entonces la diferencia es menor que 2^{-k}.

Ejemplo

Representemos 3.1459 en notación PF(8,3). Parte entera: 00011. Parte fraccionaria: 001. Representación obtenida: 00011001. Reconvirtiendo 00011001 a decimal, obtenemos parte entera 3 y parte fraccionaria 0.125; de modo que el número representado en PF(8,3) como 00011001 es en realidad **3.125** y no 3.1459.

El error de truncamiento es $3{,}1459 - 3{,}1250 = 0{,}0209$, que es menor que $2^{-3} = 0{,}125$.

PF(n,k) a decimal

Para convertir un binario en notación de punto fijo en n lugares con k fraccionarios (PF(n,k)) a decimal:

- Si es positivo, aplicamos la Expresión General extendida, utilizando los exponentes negativos para la parte fraccionaria.
- O bien, lo consideramos como un entero, convertimos a decimal y finalmente lo dividimos por 2^k.
- Si es negativo, lo complementamos a 2 y terminamos operando como en el caso positivo.
- Finalmente agregamos el signo $-$ para expresar que se trata de un número negativo.

Preguntas

- ¿A qué número decimal corresponde...
  - $0011{,}0000$?
  - $0001{,}1000$?
  - $0000{,}1100$?
- ¿Cómo se representan en $PF(8,4)$...
  - $0{,}5$?
  - $-7{,}5$?
- ¿Cuál es el RR de $PF(8,3)$? ¿Y de $PF(8,k)$?

La representación de punto fijo es adecuada para cierta clase de problemas donde los datos que se manejan son de magnitudes y precisiones comparables.

- En la situación contraria, cuando las magnitudes de los datos son muy variadas, habrá datos de valor absoluto muy grande, lo que hará que sea necesario elegir una representación de una gran cantidad de bits de ancho. Pero esta cantidad de bits quedará desperdiciada al representar los datos de magnitud pequeña.

- Otro tanto ocurre con los bits destinados a la parte fraccionaria. Si los requerimientos de precisión de los diferentes datos son muy altos, será necesario reservar una gran cantidad de bits para la parte fraccionaria. Esto permitirá almacenar los datos con mayor cantidad de dígitos fraccionarios, pero esos bits quedarán desperdiciados al almacenar otros datos.

Las ventajas de la representación en punto fijo provienen, sobre todo, de que permite reutilizar completamente la lógica ya implementada para tratar enteros en complemento a 2, sin introducir nuevos problemas ni necesidad de nuevos recursos. Como la lógica para C2 es sencilla y rápida, la representación de punto fijo es adecuada para sistemas que deben ofrecer una determinada *performance*:

- Los sistemas que deben ofrecer un tiempo de respuesta corto, especialmente aquellos interactivos, como los juegos.
- Los de tiempo real, donde la respuesta a un cómputo debe estar disponible en un tiempo menor a un plazo límite, generalmente muy corto.
- Los sistemas empotrados o embebidos, que suelen enfrentar restricciones de espacio de memoria y de potencia de procesamiento.

Por el contrario, algunas clases de programas suelen manipular datos de otra naturaleza. No es raro que aparezcan en el mismo programa, e incluso en la misma instrucción de programa, datos o variables de magnitud o precisión extremadamente diferentes.

Por ejemplo, si un programa de cómputo científico necesita calcular el **tiempo en que la luz recorre una millonésima de milímetro**, la fórmula a aplicar relacionará la velocidad de la luz en metros por segundo (unos $300{,}000{,}000 m/s$) con el tamaño en metros de un nanómetro ($0{,}000000001 m$).

Estos dos datos son extremadamente diferentes en magnitud y cantidad de dígitos fraccionarios. La velocidad de la luz es un número astronómicamente grande en comparación a la cantidad de metros en un nanómetro; y la precisión con que necesitamos representar al nanómetro no es para nada necesaria al representar la velocidad de la luz.

Notación Científica

En Matemática, la respuesta al problema del cálculo con variables tan diferentes existe desde hace mucho tiempo, y es la llamada **Notación Científica**. En Notación Científica, los números se expresan en una forma estandarizada que consiste de un **coeficiente, significando o mantisa** multiplicado por **una potencia de 10**. Es decir, la forma general de la notación es $m \times 10^e$, donde m, el coeficiente, **es un número positivo o negativo**, y e, el **exponente**, es un entero positivo o negativo.

La notación científica puede representar entonces números muy pequeños y muy grandes, todos en el mismo formato, con economía de signos y permitiendo operar entre ellos con facilidad. Al operar con cantidades en esta notación podemos aprovechar las reglas del Álgebra para calcular m y e separadamente, y evitar cuentas con muchos dígitos.

Ejemplo

Los números mencionados hace instantes, la velocidad de la luz en metros por segundo, y la longitud en metros de un nanómetro, se representarán en notación científica como 3×10^8 y 1×10^{-9}, respectivamente.

El tiempo en que la luz recorre una millonésima de milímetro se computará con la fórmula $t = e/v$, con los datos expresados en notación científica, como:

$$e = 1 \times 10^{-9} m$$
$$v = 3 \times 10^8 m/s$$
$$t = e/v = (1 \times 10^{-9}\ m)/(3 \times 10^8\ m/s) =$$
$$t = 1/3 \times 10^{-9-8}\ s =$$
$$t = 0{,}333 \times 10^{-17}\ s$$

Normalización

El resultado que hemos obtenido en el ejemplo anterior debe quedar **normalizado** llevando el coeficiente m a un valor **mayor o igual que 1 y menor que 10**. Si modificamos el coeficiente al normalizar, para no cambiar el resultado debemos ajustar el exponente.

Ejemplo

El resultado que obtuvimos anteriormente al computar $t = 1/3 \times 10^{-9-8}\ s$ fue $0{,}333 \times 10^{-17}\ s$. Este coeficiente $0{,}333$ no cumple la regla de normalización porque no es **mayor o igual que 1**.

- Para normalizarlo, lo multiplicamos por 10, convirtiéndolo en $3{,}33$.
- Para no cambiar el resultado, dividimos todo por 10 afectando el exponente, que de -17 pasa a ser -18.
- El resultado queda normalizado como $0{,}333 \times 10^{-18}$.

Normalización en base 2

Es perfectamente posible definir una notación científica en otras bases. En base 2, podemos escribir números con parte fraccionaria en notación científica normalizada desplazando la coma o punto fraccionario hasta dejar una parte entera **igual a 1** (ya que es el único valor binario que cumple la condición de normalización) y ajustando el exponente de base 2, de manera de no modificar el resultado.

Ejemplos

- $100{,}111_{(2} = 1{,}00111_{(2} \times 2^2$
- $0{,}0001101_{(2} = 1{,}101_{(2} \times 2^{-4}$

Representación en Punto Flotante

La herramienta matemática de la Notación Científica ha sido adaptada al dominio de la computación definiendo métodos de **representación en punto flotante**. Estos métodos resuelven los problemas de los sistemas de punto fijo, abandonando la idea de una cantidad fija de bits para parte entera y parte fraccionaria. En su lugar, inspirándose en la notación científica, los formatos de punto flotante permiten escribir números de un gran rango de magnitudes y precisiones en un campo de tamaño fijo.

Actualmente se utilizan los estándares de cómputo en punto flotante definidos por la organización de estándares **IEEE** (Instituto de Ingeniería Eléctrica y Electrónica, o "I triple E").

Estos estándares son dos, llamados **IEEE 754 en precisión simple y en precisión doble**.

- IEEE 754 precisión simple
  - Se define sobre un campo de 32 bits
  - Cuenta con **1 bit de signo**
  - Reserva **8 bits para el exponente**
  - Reserva **23 bits para la mantisa**
- IEEE 754 precisión doble
  - Se define sobre un campo de 64 bits
  - Cuenta con **1 bit de signo** igual que en precisión simple
  - Reserva **11 bits para el exponente**
  - Reserva **52 bits para la mantisa**

La definición de los formatos está acompañada por la especificación de mecanismos de cálculo para usarlos, manejo de errores y otra información importante.

En el curso utilizaremos siempre el formato de precisión simple.

Conversión de decimal a punto flotante

Para convertir manualmente un número decimal n a punto flotante necesitamos calcular los tres elementos del formato de punto flotante: **signo** (que llamaremos s), **exponente** (que llamaremos e) y **mantisa** (que llamaremos m), en la cantidad de bits correcta según el formato de precisión simple o doble que utilicemos.

Una vez conocidos s, e y m, sólo resta escribirlos como secuencias de bits de la longitud que especifica el formato.

1. Separar el **signo** y escribir el valor absoluto de n en base 2.

- Si n es positivo (respectivamente, negativo), s será 0 (respectivamente, 1). Separado el signo, consideramos únicamente el **valor absoluto** de n y lo representamos en base 2 como se vio al convertir un decimal fraccionario a base 2.

2. Escribir el valor binario de n en notación científica **en base 2 normalizada**.
   - Para convertir n a notación científica lo multiplicamos por una potencia de 2 de modo que la parte entera sea 1 (condición para la normalización). El resto de la expresión binaria se convierte en parte fraccionaria. Para no cambiar el valor de n, lo multiplicamos por una potencia de 2 inversa a aquella que utilizamos.
3. El exponente, positivo o negativo, que aplicamos en el paso anterior debe ser expresado en notación en exceso a 127.
   - Al exponente se le suma 127 para representar valores en el intervalo $[-127, 128]$ con 8 bits. Esta representación se elige para poder hacer comparables directamente dos números expresados en punto flotante.
4. El coeficiente calculado se guarda **sin su parte entera** en la parte de mantisa.
   - Como la normalización obliga a que la parte entera de la mantisa sea 1, no tiene mayor sentido utilizar un bit para guardarlo en el formato de punto flotante: guardarlo no aportaría ninguna información. Por eso basta con almacenar la parte fraccionaria de la mantisa, hasta los 23 bits disponibles (o completando con ceros).

Ejemplo de Punto Flotante

Recorramos los pasos para la conversión manual a punto flotante precisión simple, partiendo del decimal $n = -5{,}5$. Recordemos que necesitamos averiguar s, e y m.

- n es negativo, luego $s = 1$.
- $|n| = 5{,}5$. Convirtiendo el valor absoluto a binario obtenemos $101{,}1_{(2}$.
- Normalizando, queda $101{,}1_{(2} = 1{,}011_{(2} \times 2^2$.
- Del paso anterior, el exponente 2 se representa en exceso a 127 como $e = 2 + 127 = 129$. En base 2, $129 = 10000001_{(2}$.
- Del mismo paso anterior extraemos la mantisa quitando la parte entera: $1{,}011 - 1 = 0{,}011$. Los bits de m son $011000000...$ con ceros hasta la posición 23.
- Finalmente, $s, e, m = 1, 10000001, 011000000000....$

Lo que significa que la representación en punto flotante de $-5{,}5$ es igual a $11000000110110000000...$ (con ceros hasta completar los 32 bits de ancho total).

Expresión de punto flotante en hexadecimal

Para facilitar la escritura y comprobación de los resultados, es conveniente leer los 32 bits de la representación en punto flotante precisión simple como si se tratara de 8 dígitos hexadecimales. Se aplica la regla, que ya conocemos, de sustituir directamente cada grupo de 4 bits por un dígito hexadecimal.

Así, en el ejemplo anterior, la conversión del decimal $-5{,}5$ resultó en la secuencia de bits $11000000101100000...$ (con más ceros).

Es fácil equivocarse al transcribir este resultado. Pero sustituyendo los bits, de a grupos de 4, por dígitos hexadecimales, obtenemos la secuencia equivalente $C0B00000$, que es más simple de leer y de comunicar.

Conversión de punto flotante a decimal

Teniendo un número expresado en punto flotante precisión simple, queremos saber a qué número decimal equivale. Separamos la representación en sus componentes s, e y m, que tienen **1, 8 y 23 bits** respectivamente, y "deshacemos" la transformación que llevó a esos datos a ocupar esos lugares. De cada componente obtendremos un factor de la fórmula final.

- Signo
  - El valor de s nos dice si el decimal es positivo o negativo.
  - La fórmula $(-1)^s$ da -1 si $s = 1$, y 1 si $s = 0$.
- Exponente
  - El exponente está almacenado en la representación IEEE 754 como ocho bits en exceso a 127. Corresponde **restar 127** para volver a obtener el exponente de 2 que afectaba al número originalmente en notación científica normalizada.
  - La fórmula $2^{(e-127)}$ dice cuál es la potencia de 2 que debemos usar para ajustar la mantisa.
- Mantisa
  - La mantisa está almacenada sin su parte entera, que en la notación científica normalizada en base 2 **siempre es 1**. Para recuperar el coeficiente o mantisa original hay que restituir esa parte entera igual a 1.
  - La fórmula $1 + m$ nos da la mantisa binaria original.

Reuniendo las fórmulas aplicadas a los tres elementos de la representación, hacemos el cálculo multiplicando los tres factores:

$$n = (-1)^s \times 2^{(e-127)} \times (1 + m)$$

obteniendo finalmente el valor decimal representado.

Ejemplo

Para el valor de punto flotante IEEE 754 precisión simple representado por la secuencia hexadecimal $C0B00000$, encontramos que $s = 1$, $e = 129$, $m = 011000....$

- Signo
  - $(-1)^s = (-1)^1 = -1$
- Exponente
  - $e = 129 \rightarrow 2^{(e-127)} = 2^2$
- Mantisa
  - $m = 0110000... \rightarrow (1 + m) = 1{,}011000....$

Ajustando la mantisa $1{,}011000...$ por el factor 2^2 obtenemos $101{,}1$. Convirtiendo a decimal obtenemos $5{,}5$. Aplicando el signo recuperamos finalmente el valor $-5{,}5$, que es lo que está representando la secuencia $C0B00000$.

Error de truncamiento

Aunque los 23 bits de mantisa del formato de punto flotante en precisión simple son suficientes para la mayoría de las aplicaciones, existen números que no pueden ser representados, ni aun en doble precisión. El caso más evidente es el de aquellos números que por su magnitud caen fuera del rango de representación del sistema. Sin embargo, el formato IEEE 754 también encuentra limitaciones al tratar con números aparentemente tan pequeños como 0.1 o 0.2. ¿Cuál es el problema en este caso?

Si hacemos manualmente el cálculo de la parte fraccionaria binaria de 0.1 (o de 0.2) encontraremos que esta parte fraccionaria es **periódica**. Esto ocurre porque $0{,}1 = 1/10$, y el denominador 10 contiene factores que no dividen a la base (es decir, el 5, que no divide a 2). Lo mismo ocurre en base 10 cuando computamos $1/3$, que tiene infinitos decimales periódicos porque el denominador 3 no divide a 10, la base.

Cuando un lenguaje de computación reconoce una cadena de caracteres como "0.1", introducida por el programador o el usuario, advierte que se está haciendo referencia a un número con decimales, e intenta representarlo en la memoria como un número en punto flotante. La parte fraccionaria debe ser forzosamente **truncada**, ya sea a los 23 bits, porque se utiliza precisión simple, o a los 52 bits, cuando se utiliza precisión doble. En ambos casos, el número representado es una aproximación al 0.1 original, y esta aproximación será mejor cuantos más bits se utilicen; pero en cualquier caso, esta parte fraccionaria almacenada en la representación en punto flotante es **finita**, de manera que nunca refleja el verdadero valor que le atribuimos al número original.

A partir del momento en que ese número queda representado en forma aproximada, todos los cómputos realizados con esa representación adolecen de un **error de truncamiento**, que va agravándose a medida que se opera con ese número representado.

En precisión simple, se considera que tan sólo **los primeros siete decimales** de un número en base 10 son representados en forma correcta. En precisión doble, sólo los primeros quince decimales son correctos.

Casos especiales en punto flotante

En el estándar IEEE 754, no todas las combinaciones de s, e y m dan representaciones con sentido, o con el sentido esperable.

Por ejemplo, con las fórmulas presentadas, no es posible representar el **cero**, ya que toda mantisa normalizada lleva una parte entera igual a 1, y los demás factores nunca pueden ser iguales a 0. Entonces, para representar el 0 en IEEE 754 se recurre a una **convención**, que se ha definido como la combinación de **exponente 0 y mantisa 0**, cualquiera sea el signo.

Los números **normalizados** en IEEE 754 son aquellos que provienen de una expresión en notación científica normalizada con exponente diferente de -127, y son la gran mayoría de los representables. Sin embargo, el estándar permite la representación de una clase de números muy pequeños, con parte entera 0 en la notación científica, que son los llamados **desnormalizados**.

Otros números especiales son aquellos donde el exponente consiste en ocho **unos** binarios con mantisa 0. Estos casos están reservados para representar los valores **infinito** positivo y negativo (que aparecen cuando una operación arroja un resultado de **overflow** del formato de punto flotante).

Similarmente, cuando el exponente vale ocho unos, y la mantisa es diferente de 0, se está representando un caso de **NaN** (**Not a Number**, "no es un número"). Estos casos patológicos sólo ocurren cuando un proceso de cálculo lleva a una condición de error (por intentar realizar una operación sin sentido en el campo real, como obtener una raíz cuadrada de un real negativo).