Representación de datos numéricos

Hasta el momento hemos visto los sistemas numéricos, especialmente el **sistema binario**, desde el punto de vista puramente matemático, y hemos considerado únicamente la representación y la aritmética del conjunto de los naturales y el cero, es decir, de **números enteros no negativos**.

Sin embargo, en nuestro dominio, la computación, tarde o temprano nos enfrentaremos con algunas dificultades:

- Existen otros conjuntos numéricos, como los **enteros (positivos, cero y negativos)**, que aparecen frecuentemente en los problemas de la vida real.
- Otros conjuntos muy importantes que aún no podemos manejar son los **racionales**, los **reales** y los **complejos**.
- Además, hemos visto anteriormente que las computadoras disponen de una **cantidad limitada de dígitos** binarios para la representación numérica.

Estudiaremos entonces cómo resolver el problema de representar diferentes conjuntos numéricos con una **cantidad finita** de dígitos binarios.

Datos y tipos de datos

Al procesar los datos, las computadoras los mantienen en contenedores o unidades de almacenamiento de un tamaño determinado. Los tamaños habituales para estas unidades son, por ejemplo, **8, 16, 32 o 64 bits**. Por otro lado, los lenguajes de programación proveen una cantidad limitada de **tipos de datos**, con los que pueden operar los programas escritos en esos lenguajes. Los datos de cada tipo de datos se guardan en unidades de almacenamiento de un tamaño conveniente.

Además de usar unidades de almacenamiento de diferentes tamaños, diferentes tipos de datos tienen determinadas características. Cuando un programador necesita manejar un cierto dato dentro de un programa, elige cuidadosamente un tipo de dato provisto por el lenguaje, que le ofrezca las características que necesita para su problema. Por ejemplo, un tipo de datos **entero** puede ser **con signo** (en cuyo caso puede representar datos positivos o negativos) **o sin signo** (en cuyo caso representará únicamente positivos y el cero). O bien, un tipo de datos **real** puede tener **una determinada precisión**.

Dejemos claro que cuando hablamos de **enteros con o sin signo**, nos referimos a los **tipos de datos**, es decir, a la forma convencional como los representamos, y no a los datos mismos. Un **dato** puede ser un número **positivo** (y no llevar un signo “menos”) pero estar representado con un tipo de dato **con signo**. Como la información de si un número es o no negativo corresponde a una **pregunta binaria**, un tipo de dato entero **con signo** necesita destinar un bit para representar esta información, sea o no negativo el dato almacenado.

Rango de representación

Al representar números enteros, con o sin signo, con una cantidad finita **n** de dígitos, en un sistema cualquiera de base **b**, tenemos una primera limitación. No todos los números pueden ser representados. ¿Cuántos números podremos representar en el sistema decimal, **con n dígitos**? El intervalo de números representables en el sistema se llama su **rango de representación**. El tamaño del rango de representación es una importante característica de un tipo de datos.

- En tipos de datos **sin signo**, el **menor número** que podemos representar es el 0; pero el mayor número **sin signo** representable en un sistema de base **b** resulta ser $\mathbf{b^n-1}$. El intervalo $\mathbf{[0, b^n-1]}$ **es el llamado rango de representación** del sistema. En el sistema binario los números **sin signo** abarcan el rango **de 0 a** $\mathbf{2^n-1}$, siendo **n** la cantidad de dígitos.
- Cuando consideramos números **con signo**, este rango de representación cambia **según la técnica de representación** que usemos.

Representación de enteros con signo

Veremos dos métodos de representación, con signo, de enteros[1]. El primero se llama **signo-magnitud**, y el segundo, **complemento a 2**.

Signo-magnitud

La notación **signo-magnitud** representa el **signo** de los datos con su bit de orden más alto, o **bit más significativo** (el bit más a la izquierda), y el **valor absoluto** del dato con los restantes bits. Por ejemplo, si la unidad de almacenamiento del dato mide 8 bits, **-1** se representará como **10000001**, y **+1** como **00000001** (igual que si se tratara de un entero sin signo). El número **-3** se representará como **10000011** y el número **-64** como **11000000**. El método es fácil de ejecutar a mano, si sabemos representar los enteros sin signo.

Notemos que al usar un bit para signo, automáticamente perdemos ese bit para la representación de la magnitud o valor absoluto, disminuyendo así el rango de representación, el cual se divide por 2. Con la misma unidad de almacenamiento de 8 bits, el entero **sin signo** más grande representable era $\mathbf{2^8-1 = 255}$ (que se escribe **11111111**). Ahora, con signo-magnitud, el entero con signo más grande representable será $\mathbf{2^7-1 = 127}$ (que se escribe **01111111**).

El rango de representación en **signo-magnitud** con **n dígitos** es $\mathbf{[-2^{n-1} + 1, 2^{n-1} - 1]}$.

Para los humanos, el método de signo-magnitud resulta sumamente cómodo, pero presenta algunos inconvenientes para su uso dentro de la computadora.

- En primer lugar, el 0 tiene dos representaciones, lo que complica las cosas. Cuando la computadora necesita saber si un número es 0, debe hacer dos comparaciones distintas, una por cada valor posible del cero.
- En segundo lugar, la aritmética en signo-magnitud es bastante dificultosa, tanto para los humanos como para la computadora. Para ver la dificultad de operar en este sistema, basta con tratar de hacer a mano algunas operaciones sencillas de suma o resta en aritmética binaria en signo-magnitud. Primero hay que decidir si los signos son iguales; en caso contrario, identificar el de mayor magnitud para determinar el signo del resultado; luego hacer la operación con los valores absolutos, restituir el signo, etc. La operación se vuelve bastante compleja.

Otros métodos de representación (los sistemas de complemento) resuelven el problema con mayor limpieza.

Complemento a 2

El segundo método es el de **complemento a 2**, que resuelve el problema de tener dos representaciones para el 0 y simplifica la aritmética binaria, tanto para los humanos como para las

1 http://es.wikipedia.org/wiki/Representación_de_números_con_signo

computadoras. El método de complemento a 2 **también emplea el bit de orden más alto de acuerdo al signo**, pero la forma de representación es diferente:

- Los positivos se representan igual que si se tratara de enteros sin signo (igual que en el caso de signo-magnitud).
- Para los negativos, se usa el procedimiento de **complementar a 2,** que puede resumirse en tres pasos:
  - expresamos en binario el **valor absoluto** del número, en la cantidad de bits del sistema;
  - **invertimos los bits**
  - y **sumamos 1**.

Ejemplo

Expresar los números **-23, -9 y 8** en sistema binario en complemento a 2 con 8 bits.

- El valor absoluto de -23 es **23**. Expresado en binario en 8 bits es **00010111**. Invertido bit a bit es **11101000**. Sumando 1, queda **11101001**, luego $\mathbf{-23_{(10} = 11101001_{(2}}$.
- El valor absoluto de -9 es **9**. Expresado en binario en 8 bits, es **00001001**. Invertido bit a bit es **11110110**. Sumando 1, queda **11110111**, luego $\mathbf{-9_{(10} = 11110111_{(2}}$.
- El número **8** es positivo. Su expresión en el sistema de complemento a 2 es la misma que si fuera un número sin signo. Queda $\mathbf{8_{(10} = 00001000_{(2}}$.

El rango de representación para el sistema de **complemento a 2 con n dígitos** es $\mathbf{[-2^{n-1}, 2^{n-1} - 1]}$.

Es decir, el rango de representación de complemento a 2 **contiene un valor más** que **signo-magnitud**.

Otra propiedad interesante del método es que, si sabemos que un número binario es **negativo** y está escrito en complemento a 2, podemos recuperar su valor en decimal usando casi exactamente el mismo procedimiento: **invertimos los bits** y **sumamos 1**. Convertimos a decimal como si fuera un entero sin signo y luego agregamos el signo menos habitual de los decimales.

Ejemplo

Convertir a decimal los números, **en complemento a 2**, $11101001_{(2}$, $11110111_{(2}$ y $00000111_{(2}$.

- **11101001** es negativo, ya que su bit de orden más alto es **1**; invertido bit a bit es 00010110; sumando 1 queda 00010111 que es 23. Agregando el signo menos decimal queda $\mathbf{11101001_{(2} = -23_{(10}}$.
- **11110111** es negativo; invertido bit a bit es 00001000; sumando 1 queda 00001001 que es 9. Agregando el signo decimal queda $\mathbf{11110111_{(2} = -9_{(10}}$.
- **00000111** es positivo, porque su bit de orden más alto es 0. Lo interpretamos igual que si fuera un entero sin signo y entonces $\mathbf{00000111_{(2} = 7_{(10}}$.

Ejercicios

- Utilizando sólo dos bits, representar todos los números permitidos por el rango de

representación en complemento a 2 e indicar su valor decimal.

- Idem con tres bits.
- ¿Cuál es el rango de representación en complemento a 2 para **n = 8**?

Nota

Es interesante ver cómo los diferentes sistemas de representación asignan diferentes valores a cada combinación de símbolos. Veámoslo para unos pocos bits.

Decimal	Sin signo	Signo-magnitud	Complemento a 2
-8			1000
-7		1111	1001
-6		1110	1010
-5		1101	1011
-4		1100	1100
-3		1011	1101
-2		1010	1110
-1		1001	1111
0	0000	0000 y 1000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111
8	1000		
9	1001		
10	1010		
11	1011		
12	1100		
13	1101		
14	1110		
15	1111		

La tabla muestra que la representación en complemento a 2 es más conveniente para las operaciones aritméticas:

- La operación de complementar a 2 da un número negativo que permite resolver las restas. Las computadoras **no restan: suman complementos**.
- El cero tiene una única representación.

- En complemento a 2 se ha recuperado la representación del -8, que estaba desperdiciada en signo-magnitud.
- El rango de representación tiene un valor más en complemento a 2 que en signo-magnitud.
- Los números en complemento a 2 se disponen como en una rueda. Si tomamos cada número en la columna de complemento a 2 (salvo el último) y le sumamos 1, obtenemos en forma directa el número aritméticamente correcto. Similarmente si restamos 1 (salvo al primero). Al sumar 1 al **7**, obtenemos el menor número negativo posible. Al restar 1 a **-8**, obtenemos el mayor positivo ("entramos a la tabla por el extremo opuesto"). Esta propiedad, que no se cumple para signo-magnitud, hace más fácil la implementación de las operaciones aritméticas en los circuitos de la computadora.

Por estos motivos, la representación en complemento a 2 es más natural y consistente. Se la elige normalmente para las computadoras, en lugar de otros métodos de representación.

Aritmética en complemento a 2

Una suma en complemento a 2 se ejecuta sencillamente bit a bit, sin las consideraciones necesarias para signo-magnitud. Una resta **A – B** se interpreta como la suma de **A** y el complemento de **B**.

- La suma se realiza bit a bit desde la posición menos significativa, con **acarreo** (nos "llevamos un bit" cuando la suma de dos bits da $10_{(2}$).
- Cuando uno de los sumandos es negativo, lo complementamos y entonces hacemos la suma.

Ejemplos

- Restar $\mathbf{2_{(10}}$ **de** $\mathbf{23_{(10}}$. Complementamos 2 para convertirlo en negativo: $\mathbf{-2_{(10}} = \mathbf{11111110_{(2}}$, y lo sumamos a $\mathbf{23_{(10}}$. La cuenta es $\mathbf{23_{(10}} + \mathbf{(-2_{(10)}} = \mathbf{00010111_{(2}} + \mathbf{11111110_{(2}} = \mathbf{00010101_{(2}}$, que es positivo y vale **21**.
- Sumar $\mathbf{23_{(10}}$ **y** $\mathbf{-9_{(10}}$ con 8 bits. Se tiene que $\mathbf{23_{(10}} = \mathbf{00010111_{(2}}$ y $\mathbf{-9_{(10}} = \mathbf{11110111_{(2}}$. La suma da $\mathbf{00001110_{(2}}$, que es positivo y equivale a $\mathbf{14_{(10}}$.
- Sumar $\mathbf{-23_{(10}}$ **y** $\mathbf{-9_{(10}}$ con 8 bits. Se tiene que $\mathbf{23_{(10}} = \mathbf{00010111_{(2}}$ y por lo tanto $\mathbf{-23_{(10}} = \mathbf{11101000} + \mathbf{1} = \mathbf{11101001_{(2}}$. Por otro lado $\mathbf{-9_{(10}} = \mathbf{11110111_{(2}}$ como calculamos antes. Al sumarlos bit a bit resulta $\mathbf{11101001_{(2}} + \mathbf{11110111_{(2}} = \mathbf{11100000_{(2}}$. Descartamos el acarreo que se "cae" de los 8 bits. El resultado es negativo porque su bit más significativo es 1, y para calcular su valor decimal repetimos el procedimiento de complemento a 2: invertimos y sumamos 1. El número $\mathbf{11100000_{(2}}$ invertido es $\mathbf{00011111_{(2}}$. Sumando 1 obtenemos $\mathbf{00100000_{(2}}$ que es $\mathbf{32_{(10}}$. Como sabíamos que este número era negativo, agregamos el signo **menos** y obtenemos $\mathbf{-32_{(10}}$.

Overflow

Un riesgo de disponer de finitos dígitos binarios es que las operaciones aritméticas pueden exigir más bits de los que tenemos disponibles, originando originar errores de cómputo que deben ser detectados. Cuando ocurren estos errores, la operación debe ser declarada no válida.

Notemos que en los casos de sumar 23 y -9, o -23 y -9, el acarreo (o "**carry**") puede llegar hasta el bit de signo (**carry-in**), y también pasar más allá, siendo descartado (**carry-out**). El acarreo a veces denuncia un problema con las magnitudes permitidas por el rango de representación del sistema, como se ve en el siguiente ejemplo.

Ejemplo

Sumar **$123_{(10}$ y $9_{(10}$** con 8 bits. Se tiene que **$123_{(10}$ = 01111011 $_{(2}$** y **$9_{(10}$ = $00001001_{(2}$**. Notemos que la suma provoca el acarreo de un **1** que llega al bit de signo ("**carry-in**"). El resultado da **$10000100_{(2}$**. Claramente tenemos un problema, porque la suma de 123 + 9 debería dar positiva, pero el resultado que obtuvimos tiene el bit de orden más alto en **1**; por lo cual es negativo. Este es un caso patológico de la suma que se llama **overflow**.

Los números de nuestro ejemplo, **$123_{(10}$** y **$9_{(10}$**, pueden expresarse individualmente como enteros con signo en 8 bits, pero suman **$132_{(10}$**, número positivo que excede el rango de representación. El resultado simplemente **no se puede** expresar en 8 bits con signo en complemento a 2. Es una consecuencia de tener una cantidad limitada de bits en nuestro tipo de datos.

Cuando pasa esto se dice que ha ocurrido una condición de **overflow** o **desbordamiento aritmético**[2]. Ocurre **overflow** cuando se suman dos números positivos y se obtiene un resultado en bits que sería negativo en el rango de representación del sistema, o cuando se suman dos negativos y el resultado es positivo.

Sin embargo, en complemento a 2, nunca puede ocurrir **overflow** cuando se suman un positivo y un negativo.

Las computadoras detectan la condición de overflow porque **el bit de acarreo al llegar a la posición del signo** (**carry-in**) y el que se descarta (**carry-out**) son de diferente valor. Al sumar **23 más -9**, el carry-in y el carry-out eran **iguales**, por lo cual la suma no sufrió overflow. Al sumar **123 más 9**, el carry-in es 1 pero el carry-out es 0, lo que nos advierte que ha ocurrido overflow.

	1	**1**	1	1	1	1	1	
23	0	0	0	1	0	1	1	1
+								
-9	1	1	1	1	0	1	1	1
	0	0	0	0	1	1	1	0

No hay overflow

	0	**1**	1	1	1	1	1	
123	0	1	1	1	1	0	1	1
+								
9	0	0	0	0	1	0	0	1
	1	0	0	0	0	1	0	0

Hay overflow

Representación de números reales

Notación de punto fijo

Al representar enteros, con o sin signo, y en cualquier base, hemos supuesto que al final de los dígitos estaba el **punto fraccionario (o coma fraccionaria)** que separa la **parte entera** de la **parte fraccionaria**. No se trata de otra cosa que lo que llamamos, en el sistema numérico decimal habitual, **punto o coma decimal**. En el sistema decimal, cuando trabajamos con parte fraccionaria, el desarrollo en suma de potencias se mantiene como hemos visto pero además se extiende a los exponentes negativos:

$$3.1416 = 3 * 10^{0} + 1 * 10^{-1} + 4 * 10^{-2} + 1 * 10^{-3} + 6 * 10^{-4}$$

2 http://es.wikipedia.org/wiki/Desbordamiento_aritmético

Los sistemas numéricos que hemos usado hasta el momento, sin dígitos fraccionarios, son sistemas **de punto fijo con cero dígitos fraccionarios**. Si hacemos la convención de que utilizamos un sistema numérico **de punto fijo con n dígitos fraccionarios**, sigue siendo innecesario escribir el punto (porque sobreentendemos que el punto está **n lugares contando desde la derecha** del número escrito), pero ahora podemos representar números con parte fraccionaria. Por ejemplo, si declaramos que usamos el **sistema binario de punto fijo a 8 dígitos con 3 dígitos fraccionarios**, entonces el número **11001101** debe ser interpretado como:

$$\mathbf{11001.101 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3}}$$

Sin embargo, esta representación no es la más adecuada para todas las cantidades. En ciencias e ingenierías es común tener que manejar números muy grandes o muy pequeños, que resultan de mediciones físicas con una determinada precisión (es decir, con una cierta cantidad de decimales). Por ejemplo, la masa de la Tierra, la velocidad de la luz, o las distancias entre los planetas, son números “grandes”; la masa del electrón, o el diámetro de una célula, son números sumamente “pequeños”. En el caso de las cantidades “grandes”, lo más importante es el **orden de magnitud** o cantidad de cifras enteras, y no tiene mayor sentido consignar los decimales, ya que raramente se conocen con precisión. Por el contrario, en el caso de las cantidades “pequeñas”, normalmente la parte de mayor interés es la **precisión**, dada por los decimales.

En notación de punto fijo, una vez elegida la cantidad de dígitos enteros y fraccionarios, nos encontraremos con frecuencia que, o bien los dígitos que reservamos para la parte entera son insuficientes para una clase de números, o los dígitos fraccionarios son escasos para la otra clase. Como veremos, hay soluciones mejores que la representación de punto fijo.

Notación científica

Las matemáticas aplicadas permiten manejar estos números con una notación especial, la llamada **notación científica**[3], que expresa de una manera uniforme estos números tan diferentes. Si tuviéramos que hacer intervenir, en un mismo cómputo, cantidades muy grandes y muy pequeñas, la notación científica sería la forma más cómoda para operar.

En notación científica, los números se expresan como el producto de un coeficiente (mayor o igual que 1 y menor que 10) y una potencia entera de 10. Por ejemplo, la velocidad de la luz, 300000000 m/s, se expresa como $\mathbf{3x10^8}$ m/s, y la masa de un electrón es de alrededor de 0.000000000000000000000000000091 kg, o sea $\mathbf{9.1x10^{-31}}$ kg.

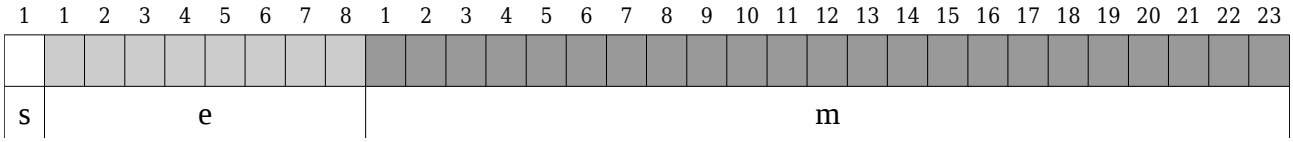
Notación en punto flotante

Al momento de almacenar y manejar estas diferentes clases de números con computadoras, tiene sentido tomar la idea de la notación científica, porque así podemos representar un gran rango de números en una cantidad fija de bits. La forma de representación de números reales más utilizada en las computadoras es la de **punto flotante**[4], que se inspira en la notación científica (pero **de base 2**), y está normalizada por la organización de ingeniería IEEE[5].

Un número en punto flotante tiene una representación bastante compleja, con tres componentes: un

3 http://es.wikipedia.org/wiki/Notación_científica
4 http://es.wikipedia.org/wiki/Coma_flotante
5 http://es.wikipedia.org/wiki/IEEE_coma_flotante

bit de signo (**s**); 8 bits para el exponente (**e**), y 23 bits para los dígitos fraccionarios, llamados **la mantisa** (**m**). De esta manera, un número en punto flotante se almacenará en **32 bits** (cuatro bytes). Se han normalizado otras cantidades de bits, pero esta forma es la más simple.



Estos tres componentes **s, e y m** tienen las siguientes particularidades.

- El signo, almacenado en el campo **s**, puede ser 0 (positivo) o 1 (negativo).
- El exponente de la notación científica puede ser negativo o positivo, pero se almacena en el campo **e** como un número sin signo (por lo tanto, entre 0 y 255), sumándole el valor 127 antes de almacenarlo.
- El coeficiente representado siempre comienza por un dígito 1. Como este dígito se conoce, no se almacena, para ahorrar espacio.
- Finalmente, un número **n** expresado en la notación en punto flotante se escribirá como
  n = $\pm 2^{(e-127)}$ * m.

Ejemplo

Representar en punto flotante el número **3.14**. El procedimiento manual lleva varios pasos:

1. Para pasar el número a sistema binario lo separamos en parte entera y en parte fraccionaria.
   - ¡La parte entera es fácil!: $\mathbf{3_{(10} = 11_{(2}}$.
   - Para escribir la parte fraccionaria en sistema binario, multiplicamos sucesivamente por 2 la parte fraccionaria decimal y tomamos la parte entera del resultado como dígito binario, repitiendo hasta llegar a 0, o hasta lograr la precisión que buscamos[6].

Fracción	Fracción * 2	Dígito binario
0.14	0.28	0
0.28	0.56	0
0.56	1.12	1
0.12	0.24	0
0.24	0.48	0
0.48	0.96 ≈ **1.00**	**1 (¡aproximando!)**
0.00	**0.00**	**0**
0.00	**0.00**	**0**

Como **0.96** está suficientemente cercano a **1.0**, podemos aproximarlo a 1 y dar por terminado el desarrollo (ya que todos los dígitos siguientes serán 0). Los dígitos binarios del resultado se leen de arriba hacia abajo. Finalmente obtuvimos:

$\mathbf{3.14_{(10} = 11.001001_{(2}}$.

2. Para normalizar esta expresión hay que correr el punto fraccionario de **11.001001** un lugar a

6 http://es.wikipedia.org/wiki/Sistema_de_numeración_binario#Decimal_.28con_decimales.29_a_binario

la izquierda, dejando un único dígito en la parte entera. Al correr el punto a la izquierda hemos dividido por 2. Compensamos la división por 2 que hicimos, multiplicando todo el número por 2, para que no cambie su valor:

$11.001001 = 1.1001001 * 2^1$.

3. Hemos calculado el exponente de la notación científica, que es 1. Para almacenarlo en los 8 bits de punto flotante le sumamos 127: $\mathbf{1}_{(2} + \mathbf{01111111}_{(2} = \mathbf{10000000}_{(2}$. El exponente **e** del punto flotante será $\mathbf{10000000}_{(2}$.
4. La mantisa **m** es la parte fraccionaria que hemos calculado: $\mathbf{1001001}_{(2}$, con tantos ceros agregados a la derecha como haga falta para llenar los 23 bits.
5. El signo **s** es **0** porque 3.14 es positivo.

Finalmente, tenemos que **3.14** se representará en punto flotante como la siguiente sucesión de bits en la memoria:

1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	e								m																						

Si quisiéramos volver a obtener el número decimal original, a partir de esta representación en punto flotante binario, podríamos encontrarnos con diferencias con el valor de partida, ya que hemos truncado el desarrollo fraccionario, aproximando al séptimo dígito. Reconstruyamos el número decimal que hemos representado en punto flotante, recordando que la mantisa tiene un 1 implícito:

$$\mathbf{n = +2^{(e-127)} * m = 2^{(128-127)} * 1.1001001}_{(2} = \mathbf{2^1 * 1.1001001}_{(2} =$$

$$= \mathbf{11.001001}_{(2} = \mathbf{2^1 + 2^0 + 2^{-3} + 2^{-6} = 3 + 1/8 + 1/64 = 3 + 0.125 + 0.015625 = 3.140625}$$

Notemos que al ser finita la cantidad de bits de **e** y de **m**, no podremos representar números de cualquier tamaño, ni almacenar todos los dígitos fraccionarios de cualquier número, sino que en la mayoría de los casos utilizaremos **aproximaciones.** De hecho, el número que hemos representado no es **3.14** sino **3.140625**. De cualquier manera, el desarrollo fraccionario no podría haber continuado más allá del dígito 23, porque sólo hay esa cantidad de dígitos en la representación en punto flotante.

Al definirse un formato de punto flotante como el que hemos descripto, la cantidad de bits asignada al exponente determina el **rango** de los números que se podrán manejar en esa representación. Con más bits en el exponente, hubiéramos podido representar números **más grandes y más pequeños**. Con más bits en la mantisa, hubiéramos podido almacenar más dígitos fraccionarios, es decir, **con mayor precisión**. Las cantidades de bits para e y para m se han escogido de modo de satisfacer la mayoría de los problemas de cómputo manteniendo un costo aceptable en trabajo de máquina y en espacio de almacenamiento.