Números de punto flotante

El estándar IEEE 754

Arquitectura de Computadores—IIC2343

2-2025

Yadran Eterovic (`yadran@uc.cl`)

Números reales

Los lenguajes de programación (C, Python, Java ...) permiten manejar números con fracciones —***números reales***, en matemáticas:

3.14159265…

2.71828…

0.000000001 (lo que dura un nanosegundo —una mil millonésima de segundo— en segundos)

… y también números enteros más grandes que el número entero más grande que puede ser representado en un entero con signo en 32 bits:

3,155,760,000 > 2,147,483,647

La cantidad de segundos que hay en un siglo

El número positivo más grande en 32 bits con signo

Dos comentarios

1) En ambos casos —números reales y números enteros muy grandes— la representación en el computador va a ser sólo aproximada la mayoría de las veces

2) Si bien habitualmente realizamos operaciones aritméticas mezclando número enteros con números reales —números con fracciones, incluso irracionales—

... computacionalmente los números enteros y los números reales *son dos tipos de datos muy diferentes*

... desde la forma en que los representamos

... y hasta cómo ejecutamos las operaciones aritméticas con ellos

Notación científica : Un único dígito a la izquierda del punto decimal

$$0.000000001 \;=\; 0.0001 \times 10^{-5} \;=\; 0.01 \times 10^{-7} \;=\; 1.0 \times 10^{-9}$$

$$3{,}155{,}760{,}000 \;=\; 3.15576 \times 10^{9} \;=\; 0.315576 \times 10^{10} \;=\; 0.0315576 \times 10^{11}$$

Un mismo número puede representarse de varias maneras distintas en notación científica

Notación científica normalizada : Ese único dígito es $\neq 0$

$$0.000000001 = 1.0 \times 10^{-9}$$

$$3{,}155{,}760{,}000 = 3.15576 \times 10^{9}$$

Un número tiene una ***única representación*** en notación científica normalizada

Los números reales también se pueden representar en base 2, generalizando la representación posicional de números enteros:

- los exponentes de la base 2 al lado derecho del punto decimal son negativos

3×10^{0} 6×10^{-1} 6×10^{-2} 4×10^{-3}

$$3\,.\,6\;6\;4\ldots_{10} = 1\;1\,.\,1\;0\;1\;0\;1\;0\;1_{2}$$

1×2^{1} 1×2^{0} 1×2^{-1} 0×2^{-2} 1×2^{-3} 0×2^{-4} 1×2^{-5} 0×2^{-6} 1×2^{-7}

Y también podemos escribir números (reales) binarios en notación científica normalizada

… usando 2 como base del exponente:

$$11.1010101 = 11.1010101 \times 2^0 = \mathbf{1.11010101 \times 2^1}$$

La aritmética computacional correspondiente se llama ***aritmética de punto flotante***:

- maneja números en los que la posición del punto decimal (o punto binario) no está fija, sino que "flota"
- en C, el tipo de datos correspondiente se llama **`float`**

Los números son representados a partir de un único dígito ≠ 0 a la izquierda del punto —necesariamente, un **1**:

$$\mathbf{1}.\text{xxxxxxxx} \times 2^{\text{yyy}}$$

Los números son representados a partir de un único dígito ≠ 0 a la izquierda del punto —necesariamente, un **1**:

$\mathbf{1}.xxxxxxxx \times 2^{yyy}$

Ventajas:

- simplifica el intercambio de datos
- simplifica los algoritmos para la aritmética de punto flotante
- aumenta la exactitud de los números que pueden almacenarse en una palabra —los 0s iniciales innecesarios son reemplazados por dígitos reales “valiosos” a la derecha del punto

Representación de números de punto flotante

$$\pm\ 1.xxxxxxxx\ldots \times 2^{yyy\ldots}$$

El número total de bits para representar un número real queda fijo al momento de diseñar el computador (igual que en el caso de los números enteros)

- p.ej., para 32 bits

  … el signo, el 1, los *xxx*…, el 2 y los *yyy*… deben repartirse los 32 bits

Valor “fijo”

Un bit, sí o sí

$\pm\ 1.\text{xxxxxxxx}\ldots \times 2^{\text{yyy}\ldots}$

Valor “fijo”

en 32 bits

$\Longrightarrow$ compromiso entre el número de bits para la fracción —o mantisa— y el número de bits para el exponente:

hay que quitarle un bit a la fracción (los *xxx*…) para agregárselo al exponente (los *yyy*…), y viceversa

… es decir, compromiso entre ***precisión*** y ***rango***:

- mayor número de bits de la fracción $\Longrightarrow$ mayor precisión de la fracción
- mayor número de bits del exponente $\Longrightarrow$ mayor rango de los números representables

P.ej., en el caso de 32 bits:

un bit para el signo del número (0 → positivo, 1 → negativo)

23 bits para la fracción

8 bits para el exponente (un número entero: positivo, negativo, o 0)

| 1 | 10000001 | 01000000000000000000000 |

Signo: un bit

Exponente: 8 bits

Fracción: 23 bits

Signo: un bit

Exponente: 8 bits

Fracción: 23 bits

| 1 | 10000001 | 01000000000000000000000 |

Esta es una representación de tipo ***signo y magnitud***:

→ el signo es un bit separado del resto del número

El valor de un número así representado es

$$(-1)^{signo} \times fracción \times 2^{exponente}$$

Cinco propiedades de la representación computacional

1) Gran rango de números representables:

aproximadamente, desde $\pm 1.7 \times 10^{-38}$ hasta $\pm 1.7 \times 10^{38}$

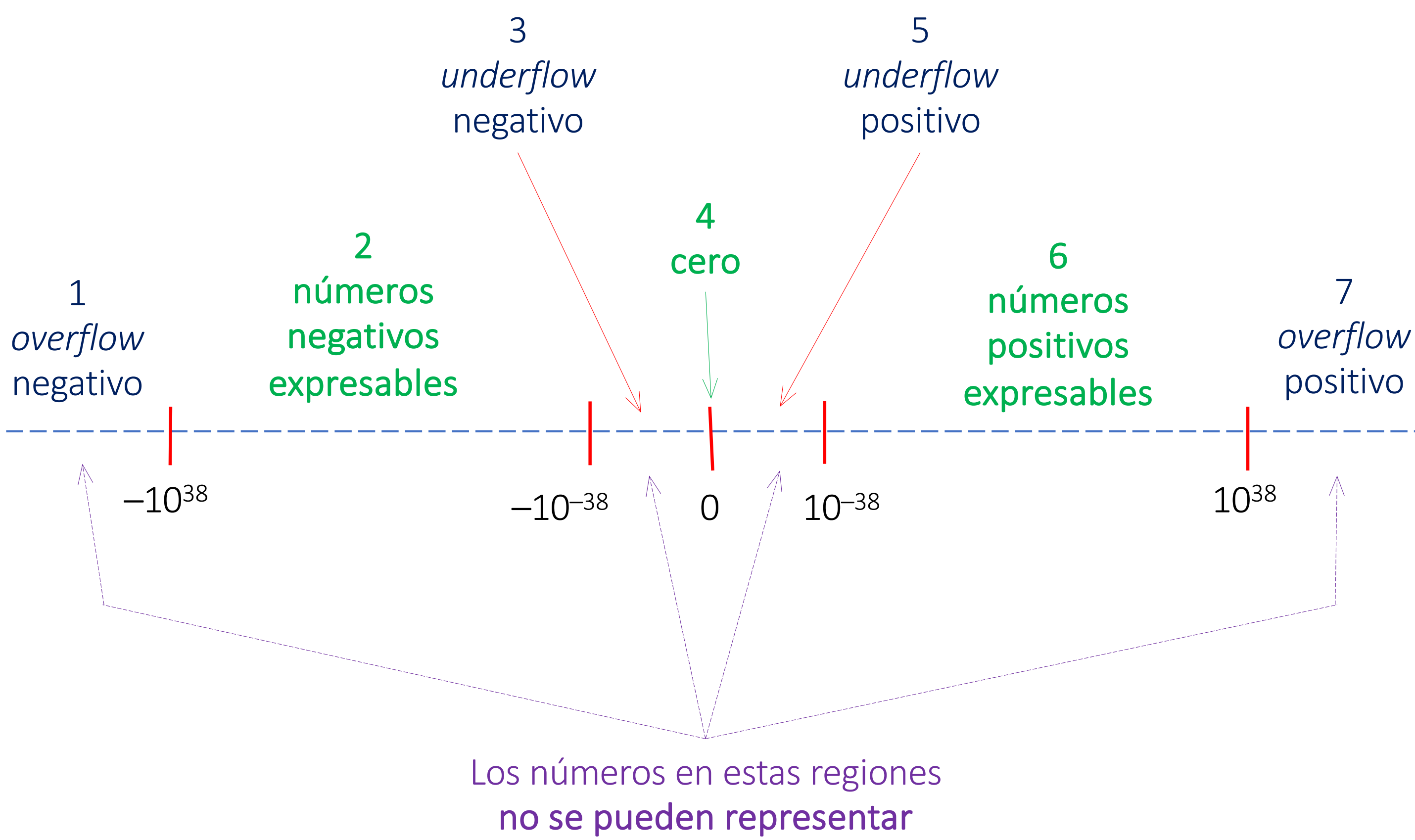
2) Buena precisión: $2^{-22} = 2.384 \times 10^{-7}$ ($\cong$ uno en 4 millones)

3) Ahora, además de *overflow*, puede ocurrir *underflow*:

- ***overflow*** : cuando el exponente es demasiado grande para ser representado en 8 bits

- ***underflow*** : cuando el exponente negativo es demasiado grande → el valor de la fracción es muy pequeño (por lo que 0 es una buena aproximación)

4) La recta de números reales queda dividida en 7 regiones :



5) No todos los números reales en las regiones 2 y 6 pueden ser representados:

- p.ej., entre $1.0000000000000000000000_2$ y $1.1111111111111111111111_2$

  … hay sólo 2^{22} números distintos,

  … pero en la recta de números reales hay infinitos números entre 1.0 y 1.999999…

- → para representar los números que no se pueden representar de manera exacta, es necesario emplear ***redondeo***

En la práctica, desde 1985 se usa el estándar *IEEE 754*

Fracción:

23 bits —los 23 bits menos significativos de la palabra (del bit 0 al bit 22)

el 1 a la izquierda del punto decimal es ***implícito***

$\Longrightarrow$ el ***significante*** tiene 24 bits

Signo:

el bit más significativo (bit 31)

$\Longrightarrow$ facilita las comparaciones < 0, $= 0$, > 0

Exponente:

8 bits : bits 23 al 30

inmediatamente a continuación del signo

… facilita la ordenación usando instrucciones de comparación de números enteros

… siempre que todos los exponentes tengan el mismo signo:

… ⟹ se los representa como números sin signo desplazados (*biased*) en 127

(en lugar de representarlos como números enteros en complemento de 2)

El valor de un número en punto flotante en IEEE 754 es

$$(-1)^{signo} \times (1+fracción) \times 2^{exponente-127}$$

11000000101000000000000000000000

Signo: bit 31

Exponente: bits 23 al 30

Fracción: bits 0 al 22

| 1 | 10000001 | 01000000000000000000000 |

P.ej., ¿cuál es el número decimal representado por el siguiente float?

11000000101000000000000000000000

Identifiquemos sus partes:

| 1 | 10000001 | 01000000000000000000000 |

signo → 1

exponente → 129

fracción → $1 \times 2^{-2} = 1/4 = 0.25$

Por lo tanto, el número es (ver fórmula en diap. anterior):

$$(-1)^1 \times (1 + 0.25) \times 2^{129-127} = -1 \times 1.25 \times 2^2$$

$$= -1.25 \times 4$$

$$= \mathbf{-5.0}$$

P.ej., ¿cuál es la representación en IEEE 754 de -0.75_{10} ?

$$-0.75_{10} = -3/4_{10} = -3/2^2{}_{10} = -11_2/2^2{}_{10} = -0.11_2$$

$$= -0.11_2 \times 2^0 \qquad \text{(en notación científica)}$$

$$= -1.1_2 \times 2^{-1} \qquad \text{(en notación científica normalizada)}$$

Por lo tanto, *signo* = 1

significante = 1.1 → *fracción* = .1

exponente = –1 + 127 = 126

… y la representación es

1 01111110 10000000000000000000000

Combinaciones especiales de *exponente* y *fracción*:

0 : *exponente* = 0 y *fracción* = 0

número no normalizado : *exponente* = 0 y *fracción* ≠ 0

infinito : *exponente* = 255 (o 2047) y *fracción* = 0

$\Longrightarrow$ los exponentes "válidos" son desde el 1 al 254, que representan los valores -126 a $+127$

NaN (*not a number*): *exponente* = 255 (o 2047) y *fracción* ≠ 0

El estándar IEEE 754 tiene también un formato de 64 bits

→ ***precisión doble***:

exponente de 11 bits, con desplazamiento de 1023

fracción de 52 bits $\Longrightarrow$ *significante* de 53 bits

Rango de números representables:

10^{-38} ($\approx 2^{-126}$) a 10^{38} ($\approx 2^{128}$), en precisión simple

10^{-308} ($\approx 2^{-1022}$) a 10^{308} ($\approx 2^{1024}$), en precisión doble

Ejemplo de suma de números de punto flotante

(para que sea más fácil de entender, el ej. está en base 10)

$$9.999 \times 10^{1} + 1.610 \times 10^{-1} = \ldots$$

Suponemos significantes de 4 dígitos, y exponentes de dos dígitos

Primero, hay que alinear el punto decimal del número con el menor exponente:

$$1.610 \times 10^{-1} = 0.1610 \times 10^{0} = 0.01610 \times 10^{1} \rightarrow 0.016 \times 10^{1}$$

es decir, hicimos shift a la derecha del significante, aumentando el exponente en 1 cada vez, hasta quedar con el exponente correcto

… y recordamos que sólo podemos representar 4 dígito

(sigue)

Luego, sumamos los significantes

… y normalizamos y redondeamos el resultado:

$9.999 + 0.016 = 10.015$ —sumamos significantes

$\rightarrow\ 10.015 \times 10^1 = 1.0015 \times 10^2$ —normalizamos *

$\rightarrow\ 1.002 \times 10^2$ —redondeamos **

* Al normalizar, aumentando o disminuyendo el exponente, hay que revisar si se produce overflow o underflow

** Al redondear, hay que revisar si el resultado se mantiene normalizado, o si es necesario normalizarlo de nuevo

Algoritmo de suma de punto flotante

1. Comparar los exponentes; “shift” el número más pequeño a la derecha hasta que su exponente sea igual al exponente más grande

2. Sumar los significantes

3. Normalizar la suma, ya sea “shifting” a la derecha e incrementado el exponente, o “shifting” a la izquierda y decrementando el exponente

   ¿“Overflow” o “underflow”? → Excepción

4. Redondear el significante al número apropiado de bits

   ¿Está normalizado? → terminar

   Volver al paso 3

Algoritmo de multiplicación de punto flotante

1. Sumar los exponentes desplazados; restar el deplaza-miento a la suma para obtener el nuevo exponente desplazado

2. Multiplicar los significantes

3. Normalizar el producto si es necesario, “shifting” a la derecha e incrementado el exponente

   ¿“Overflow” o “underflow”? → Excepción

4. Redondear el significante al número apropiado de bits

   ¿Está normalizado? → Ir al paso 5

   Volver al paso 3

5. Poner el signo del producto en positivo si los signos de los operandos son iguales; en negativo, en caso contrario

   Terminar

Multiplicación de números enteros

(algoritmo del colegio)

Tomamos los dígitos del multiplicador uno a uno de derecha a izquierda —es decir, desde el dígito menos significativo al dígito más significativo

… para cada dígito del multiplicador, multiplicamos el multipli-cando por ese dígito (→ producto intermedio)

… y escribimos este producto intermedio, desplazado un dígito a la izquierda con respecto al producto intermedio anterior

Finalmente, sumamos los productos intermedios, respetando el desplazamiento de cada uno

Multiplicando

Multiplicador

		1	2	3	× 4	5
		6	1	5	= 123 × 5	
+	4	9	2	0	= 123 × 40	
=	5	5	3	5		

Producto

Si restringimos los dígitos a 0 y 1 (como es siempre el caso con los números binarios),

... cada paso de la multiplicación es simple:

				1	0	1	0	×	1 0 0 1
				1	0	1	0		= 1010 × 1
			0	0	0	0			= 1010 × 0
		0	0	0	0				= 1010 × 0
+	1	0	1	0					= 1010 × 1
=	1	0	1	1	0	1	0		suma y producto final

Algoritmo de multiplicación binaria

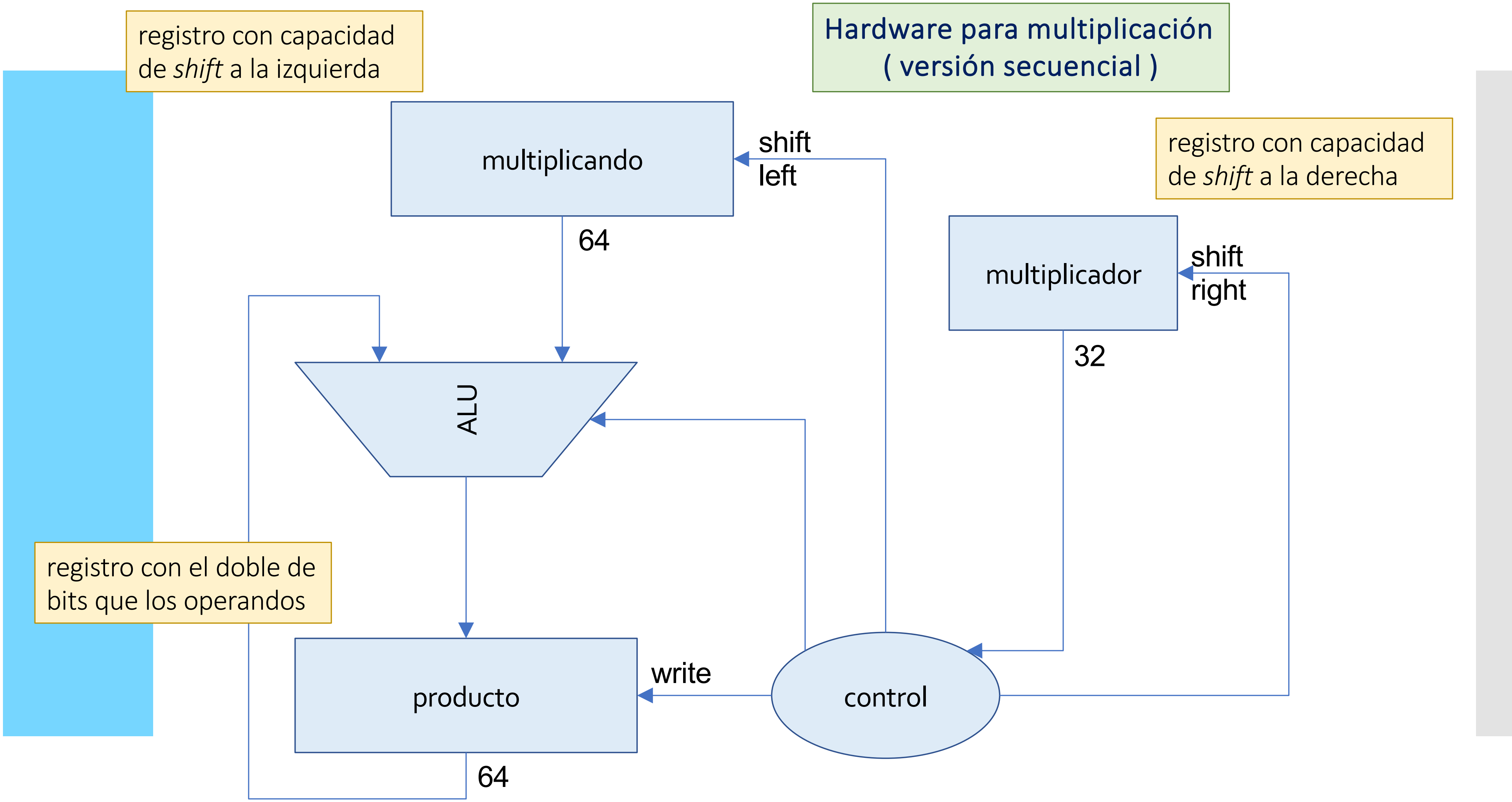
Colocar una copia del multiplicando en el lugar correcto*, si el dígito del multiplicador es 1 (= 1 × multiplicando)

… o bien colocar 0 en el lugar correcto*, si el dígito del multiplicador es 0 (= 0 × multiplicando)

* en cada nueva iteración, se escribe desplazándolo un dígito a la izquierda

				1	0	1	0	×	1	0	0	1		
				1	0	1	0		= 1010 × 1					
			0	0	0	0			= 1010 × 0					
		0	0	0	0				= 1010 × 0					
+	1	0	1	0					= 1010 × 1					
=	1	0	1	1	0	1	0		suma y producto final					

Hardware para multiplicación (versión secuencial)



Algoritmo de multiplicación de números enteros de 32 bits sin signo, usando el hardware anterior

Partir

1. Examinar el dígito de más a la derecha del multiplicador

1a. Si es 1, sumar el multiplicando al producto y poner el resultado en el registro del producto

2. Desplazar (shift) el registro del multiplicando a la izquierda un bit

3. Desplazar (shift) el registro del multiplicador a la derecha un bit

4. Verificar si los pasos 1, 2 y 3 se han repetido 32 veces

4a. Si aún faltan repeticiones, ir a 1

Terminar

Ejemplo de multiplicación de números de punto flotante

$$1.000 \times 2^{-1} \times -1.110 \times 2^{-2} = \ldots$$

Primero, sumamos los exponentes:

sin desfasarlos (sus verdaderos valores) → $-1 + (-2) = -3$

o desfasándolos (como son representados en el computador)

→ $(-1+127) + (-2+127) - 127 = -3 + 127 = 124$

Luego, multiplicamos los significantes y dejamos el producto en 4 bits:

$1.000 \times 1.110 = 1.110000 \rightarrow 1.110000 \times 2^{-3} \rightarrow 1.110 \times 2^{-3}$

(sigue)

Revisamos que el producto esté normalizado → lo está

… y que el exponente no haya producido *overflow* o *underflow*:

$127 \geq -3 \geq -126 \rightarrow$ no se produjo ninguno (con desfase: $254 \geq 124 \geq 1$)

Redondeamos el producto, sin efecto en este caso

… y finalmente hacemos que el signo del producto sea negativo:

$$-1.110 \times 2^{-3}$$

Sobre redondeos

El hardware necesita bits adicionales para hacer bien los redondeos

El estándar IEEE 754 mantiene dos bits adicionales (a la derecha del bit menos significativo —el de más a la derecha)

… y ofrece cuatro modos de redondeo:

- siempre redondear hacia arriba
- siempre redondear hacia abajo
- truncar
- redondear al par más cercano —el más comúnmente usado (p.ej., el que usa Java)