

Números y aritmética

Arquitectura de Computadores – IIC2343

Cuando se trata de números, las personas pensamos en base 10 —números decimales:

- diez símbolos diferentes —0, 1, 2, 3, 4, 5, 6, 7, 8, 9— llamados dígitos decimales, o simplemente, dígitos

... y los representamos en lo que llamamos la *notación posicional*:

- el verdadero valor de un dígito en un número de varios dígitos depende no sólo del dígito mismo

... sino también de su posición dentro del número

P.ej., cuando escribimos 421, en realidad estamos hablando del número (o valor numérico) que es el resultado de la siguiente operación:

$$4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Los números pueden ser representados en cualquier base, usando la misma notación posicional

... en particular, en **base 2** (*números binarios*):

- sólo dos símbolos diferentes —**0**, **1**— llamados *dígitos binarios* o **bits**

P.ej., el número 421 en base 2 se representa así:

1 1 0 1 0 0 1 0 1

... ya que (ver diap. 5):

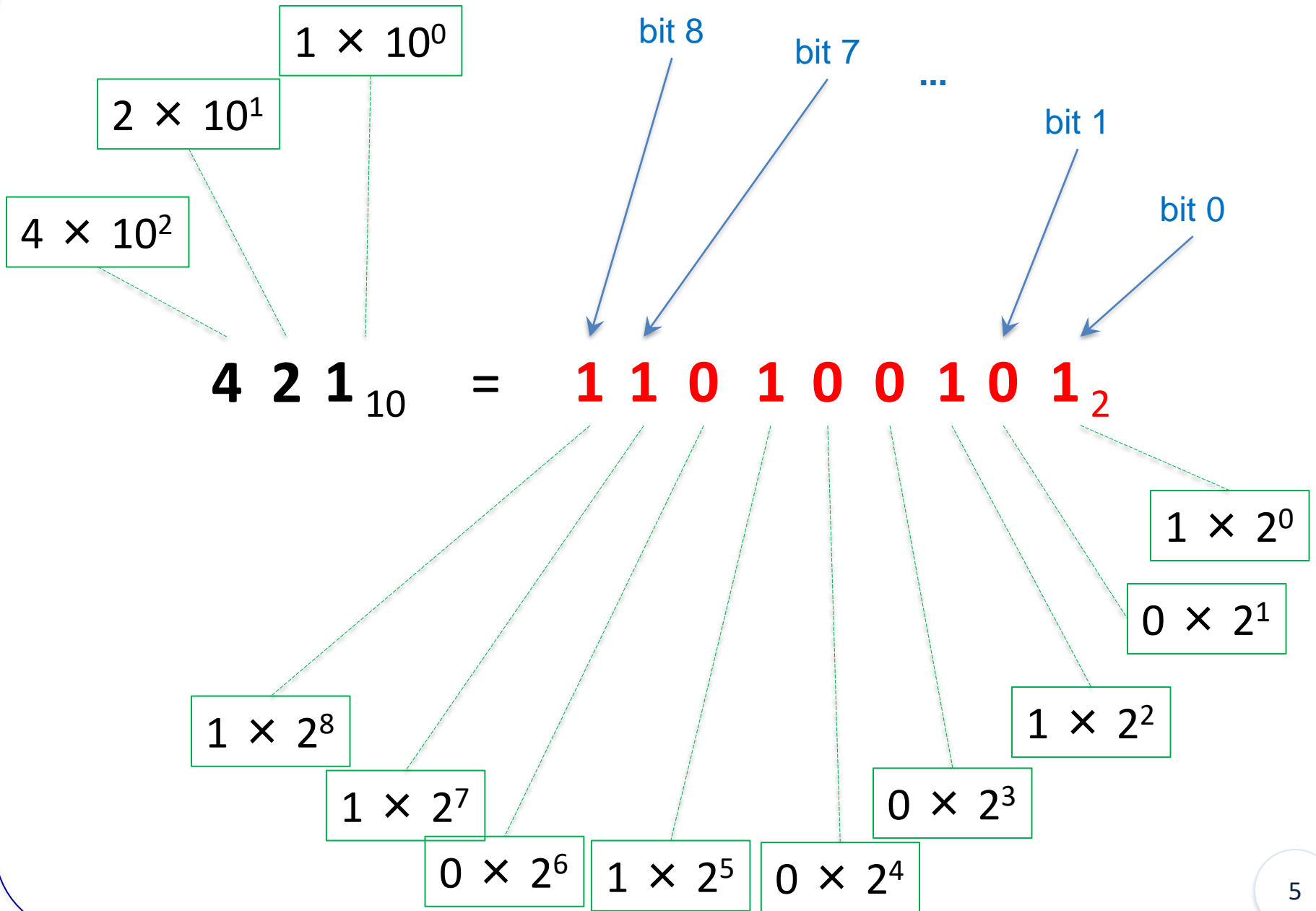
$$\begin{aligned} 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 \\ + 0 \times 2^1 + 1 \times 2^0 = 421 \end{aligned}$$

Los bits son los “átomos” de la computación:

- toda información en formato “computacional” se compone sólo de bits

Para poder referirnos a ellos de manera precisa, numeramos los bits de un número binario —el bit 0, el bit 1, el bit 2, el bit 3, ...— de derecha a izquierda:

- es decir, desde el **bit menos significativo** —el que va multiplicado por 2^0
... al **bit más significativo** —el que va multiplicado por 2^{n-1} , si el número de bits es n



Tanto en base 10 como en base 2, un número (un valor numérico) tiene una única representación

... → si cambiamos cualquier dígito de 421, el nuevo número va a tener un valor numérico distinto de 421

... y lo mismo con 1 1 0 1 0 0 1 0 1: si cambiamos cualquiera de los 1's por 0's o cualquiera de los 0's por 1's, el valor numérico del número binario resultante va a ser distinto de 421

La cantidad de bits disponible para representar un número queda fija al momento de diseñar el computador:

números de precisión finita

Y esto tiene consecuencias

(Esta no es una limitación de la base 2, sino de que la memoria del computador —independientemente de la tecnología que se use para implementarla— es finita)

P.ej., consideremos el conjunto de números enteros positivos representables mediante tres dígitos decimales, sin punto decimal ni signo:

000, 001, 002, ..., 999

En este caso, es imposible representar ciertos números:

- mayores que 999, negativos, fracciones, irracionales, complejos

Además, el conjunto no es cerrado con respecto a las operaciones aritméticas básicas:

- $600 + 600 = 1200 \rightarrow$ muy grande
- $003 - 005 = -2 \rightarrow$ negativo
- $050 \times 050 = 2500 \rightarrow$ muy grande
- $007 / 002 = 3.5 \rightarrow$ no es un entero

Finalmente, el álgebra de los números de precisión finita es diferente del álgebra “normal”:

- p.ej., la ley de asociatividad $a + (b - c) = (a + b) - c$ no se cumple si $a = 700$, $b = 400$ y $c = 300$, porque al calcular $a + b$ en el lado derecho produce *overflow*

Estas mismas limitaciones o problemas se dan al representar números enteros positivos (en base 2) mediante, p.ej., 32 bits —típico en un computador moderno ... bueno, casi moderno

Obviamente, no es que los computadores sean inadecuados para hacer aritmética, sino que

... es importante entender cómo funcionan (lo que vamos a ver a continuación)

(En las próximas diapos, vamos a emplear el subíndice 10 cuando escribamos números en base 10, y el subíndice 2 cuando escribamos números en base 2; p.ej., 421_{10} y 110100101_2)

En computadores en que las palabras tienen 32 bits, podemos representar 2^{32} patrones diferentes de bits

... los números desde el 0 hasta el $2^{32} - 1 = 4,294,967,295_{10}$

Así, con 32 bits, el valor del número representado como

$$x_{31}x_{30}\dots x_1x_0$$

... es

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Estos números positivos, incluyendo el 0, se llaman **números sin signo** (*unsigned numbers*) o **enteros sin signo** (*unsigned integers*)

Pero también tenemos que representar números negativos

P.ej., podríamos agregar un signo, representado en un bit:

$$0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1_2 = 421_{10}$$

$$1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1_2 = -421_{10}$$

... esta representación se llama ***signo y magnitud***

Problemas:

- el bit de signo, ¿es el de más a la derecha o el de más a la izquierda?
- al sumar, se necesita un paso adicional para saber el valor de este bit
- hay dos ceros —uno positivo y otro negativo ($00...00_2$ y $10...00_2$)

Hay otras opciones, todas con sus pros y sus contras

No habiendo una mejor opción obvia

... los arquitectos computacionales eligieron finalmente la representación llamada **complemento de 2** que hacía más simple el hardware:

- 0s a la izquierda significa positivo
- 1s a la izquierda significa negativo

Veámoslo para 32 bits (próx. diap.)

[muchos computadores hoy día usan 64 bits y en el pasado se usaban 16 bits —la representación en complemento de 2 es independiente del número de bits: básicamente, lo único que se ve afectado es el rango de números representados]

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$$

⋮

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2,147,483,645_{10}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2,147,483,646_{10}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2,147,483,647_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2,147,483,648_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2,147,483,647_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2,147,483,646_{10}$$

⋮

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$$

En 32 bits (diap. anterior):

- la mitad positiva, de 0 a $2,147,483,647_{10}$ ($= 2^{31} - 1$) usa la misma representación que antes
- el patrón de bits que sigue ($1000...0000_2$) representa el número más negativo, $-2,147,483,648_{10}$ ($= -2^{31}$)
... el cual (en esta representación y para 32 bits) no tiene un número positivo correspondiente
- y luego viene una secuencia de números negativos de magnitud decreciente, desde $-2,147,483,647_{10}$ ($= 1000...001_2$) hasta -1_{10} ($= 1111...111_2$)

Todo computador hoy en día, y desde 1965, usa complemento de 2 para representar **números con signo** (*signed numbers*)

string	unsigned	sign & magnitude	1's complement	2's complement
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

En complemento de 2, todos los números negativos —y sólo los números negativos— tienen un 1 en el bit más significativo (el de más a la izquierda):

- *bit de signo*, aunque no es un bit que pueda realmente separarse de la magnitud del número (como se explica en la próx. diap.)

p.ej., en 4 bits (diap. anterior), $1001_2 = -7_{10}$, pero $001_2 \neq 7_{10}$

- basta examinar este bit para saber si un número es positivo o negativo (0 se considera positivo)

Así, con 32 bits en complemento de 2, el valor del número representado como

$$x_{31}x_{30}\dots x_1x_0$$

... es

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Notar la diferencia en la interpretación del valor del bit 31 con respecto al caso de los números sin signo (diap. 10)

Complemento de 2

8 bits

$$00000000_2 = 0_{10}$$

$$00000001_2 = 1_{10}$$

$$00000010_2 = 2_{10}$$

$$00000011_2 = 3_{10}$$

⋮

$$01111100_2 = 124_{10}$$

$$01111101_2 = 125_{10}$$

$$01111110_2 = 126_{10}$$

$$01111111_2 = 127_{10}$$

$$10000000_2 = -128_{10}$$

$$10000001_2 = -127_{10}$$

$$10000010_2 = -126_{10}$$

$$10000011_2 = -125_{10}$$

⋮

$$11111100_2 = -4_{10}$$

$$11111101_2 = -3_{10}$$

$$11111110_2 = -2_{10}$$

$$11111111_2 = -1_{10}$$

$$\begin{array}{r} 00000011_2 \quad (= 3_{10}) \\ + 01111100_2 \quad (= 124_{10}) \\ \hline \end{array}$$

$$01111111_2 \quad (= 127_{10})$$

$$\begin{array}{r} 10000011_2 \quad (= -125_{10}) \\ + 11111111_2 \quad (= -1_{10}) \\ \hline \end{array}$$

$$1100000010_2 \quad (= -126_{10})$$

$$\begin{array}{r} 00000010_2 \quad (= 2_{10}) \\ + 01111111_2 \quad (= 127_{10}) \\ \hline \end{array}$$

$$100000001_2 \quad (= -127_{10})$$

$$\begin{array}{r} 10000011_2 \quad (= -125_{10}) \\ + 11111100_2 \quad (= -4_{10}) \\ \hline \end{array}$$

$$101111111_2 \quad (= 127_{10})$$

En complemento de 2, ocurre *overflow* (como en las dos últimas sumas de la diap. anterior) cuando el resultado de la operación produce un bit de signo incorrecto (bits rojos):

- un 0 en el bit de más a la izquierda (bit $n-1$ ésimo) cuando el número es negativo
- un 1 en el bit de más a la izquierda (bit $n-1$ ésimo) cuando el número es positivo

¿ Cómo determinamos el inverso aditivo de un número binario Y de n bits en complemento de 2 ?

Si miramos la diap. 13 (o la última columna de la diap. 15), vemos que $Y + \bar{Y}$ (invertimos cada bit de Y) $= 111...111_2 = -1$

... es decir, $Y + \bar{Y} = -1 \Rightarrow -Y = \bar{Y} + 1$

Por lo tanto, el inverso aditivo $-Y$ se obtiene así:

primero, invertimos cada bit de Y ($0 \rightarrow 1, 1 \rightarrow 0$), lo que nos da \bar{Y}

... y luego, sumamos 1 al resultado

(por otra parte, si no tomamos en cuenta el signo, $Y + \bar{Y} = 111...111_2 = 2^n - 1 \Rightarrow Y + \bar{Y} + 1 = 2^n \Rightarrow Y + (-Y) = 2^n$; de aquí el nombre “complemento de 2”)

¿Qué sabemos?

Números enteros con signo:

Los representamos en *complemento de 2*

Suma:

Los números son sumados bit a bit de derecha a izquierda —lo mismo que en el caso de base 10

... y la reserva (*carry*) se va pasando al próximo par de bits a la izquierda

Resta:

Hace uso de la suma,

... sólo que el sustraendo es convertido a su inverso aditivo antes de ser sumado al minuendo

Overflow es cuando el resultado de una operación no puede ser representado con el número de bits disponible:

- p.ej., si en 32 bits con complemento de 2, el resultado de una operación fuera un número positivo mayor que $2^{31}-1$ (éste es el positivo más grande)
... o un número negativo de magnitud mayor que 2^{31} (-2^{31} es el negativo más grande)

...

...

No puede ocurrir *overflow* cuando sumamos operandos con diferentes signos —uno positivo y el otro negativo:

- ya que la suma no puede ser más grande que uno de los operandos

... ni cuando restamos operandos con el mismo signo —ambos positivos o ambos negativos:

- ya que primero convertimos el sustraendo a su inverso aditivo (es decir, le cambiamos el signo) y luego sumamos

...

...

Cuando ocurre *overflow*, es porque falta un bit para representar el resultado

... entonces el bit de signo —más precisamente, el bit más significativo o más a la izquierda— recibe el valor del resultado

... en lugar de recibir el signo correspondiente al resultado:

- si sumamos dos números positivos y el resultado es negativo, es porque ocurrió *overflow*

... o si sumamos dos números negativos y el resultado es positivo, es porque ocurrió *overflow*

Es responsabilidad del lenguaje de programación, sistema operativo y/o programa determinar qué hacer en caso de *overflow*:

- el hardware no tiene cómo saber qué conviene hacer en este caso

Números reales

Los lenguajes de programación permiten manejar números con fracciones —*números reales*, en matemáticas:

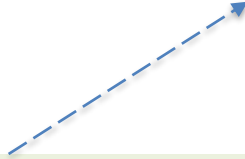
3.14159265...

2.71828...

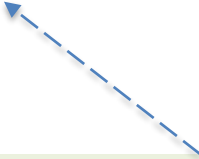
0.000000001 (lo que dura un *nanosegundo* —una mil millonésima de segundo— en segundos)

... y también números enteros más grandes que el número entero más grande que puede ser representado en un entero con signo en 32 bits (o 64):

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



la cantidad de segundos
que hay en un siglo



el número positivo más grande
en 32 bits con signo

Dos comentarios

- 1) Por supuesto, en ambos casos —números reales y números enteros muy grandes— la representación es sólo aproximada la mayoría de las veces, como vamos a ver
- 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*, como vamos a ver,
 - ... desde la forma en que los representamos (p.ej., en 32 bits)
 - ... 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

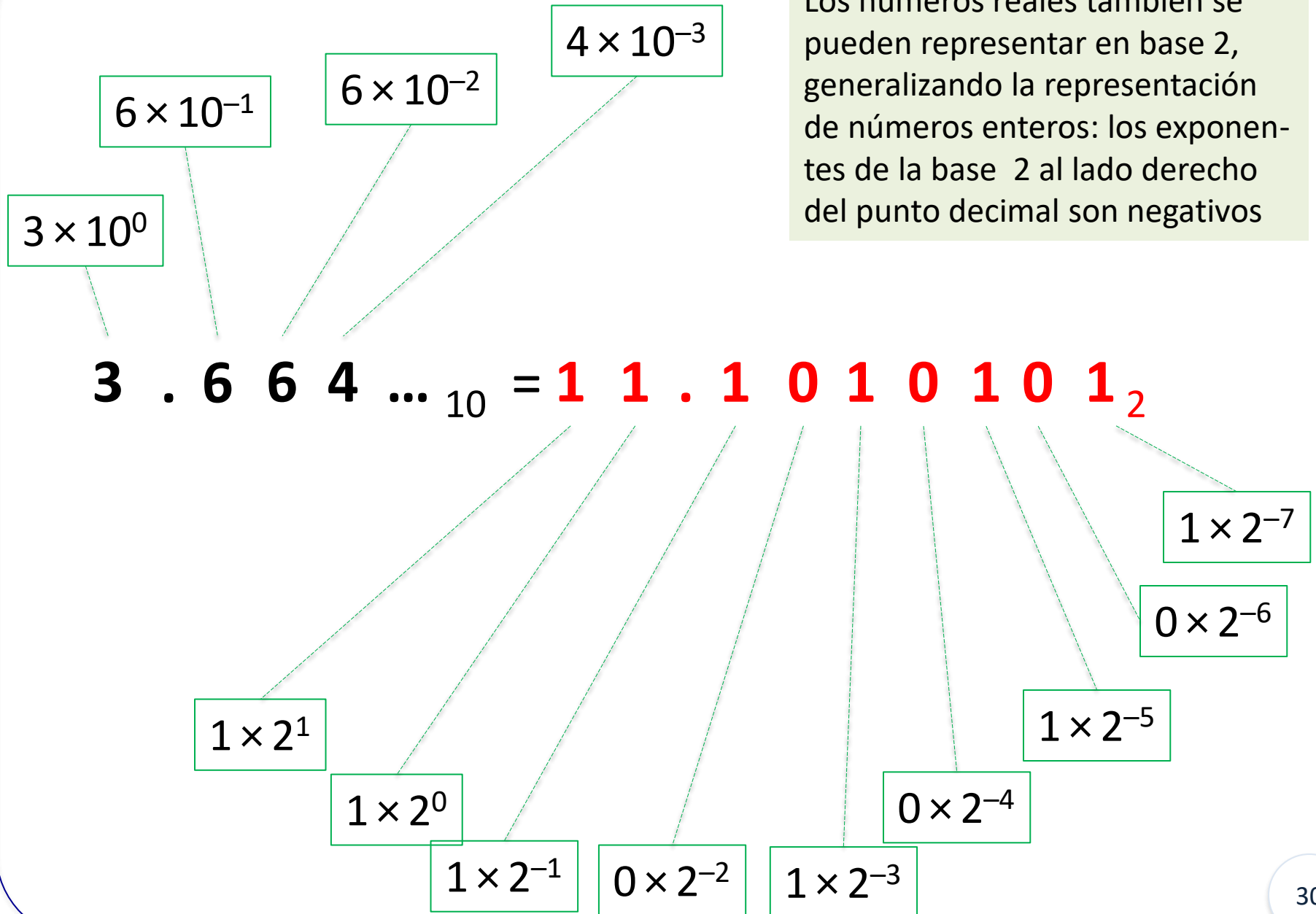
Notación científica **normalizada** : ese único dígito es $\neq 0$

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

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

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

Los números reales también se pueden representar en base 2, generalizando la representación de números enteros: los exponentes de la base 2 al lado derecho del punto decimal son negativos



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 ***de punto flotante***:

maneja números en los que la posición del punto decimal (o *punto binario*) no está fija

en C, el tipo de datos correspondiente se llama **float**

Los números son representados a partir de un único dígito $\neq 0$ a la izquierda del punto:

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

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 a la derecha del punto

Representación de números de punto flotante

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

Al igual que en el caso de los números enteros, el número total de bits para representar un número real queda fijo al momento de diseñar el computador; p.ej., 32 bits:

- el signo, el 1, los xxx..., el 2 y los yyy... deben repartirse los 32 bits

un bit, sí o sí

valor “fijo”

valor “fijo”

$\pm 1.\text{xxxxxxxxx}\dots \times 2^{\text{yyy}\dots}$ en 32 bits

→ 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*:

a mayor tamaño de la fracción, mayor precisión de la fracción

a mayor tamaño del exponente, 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 | 010000000000000000000000 |

signo
un bit

exponente
8 bits

fracción
23 bits

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

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

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 \rightarrow el valor de la fracción es muy pequeño (por lo que 0 es una buena aproximación)

...

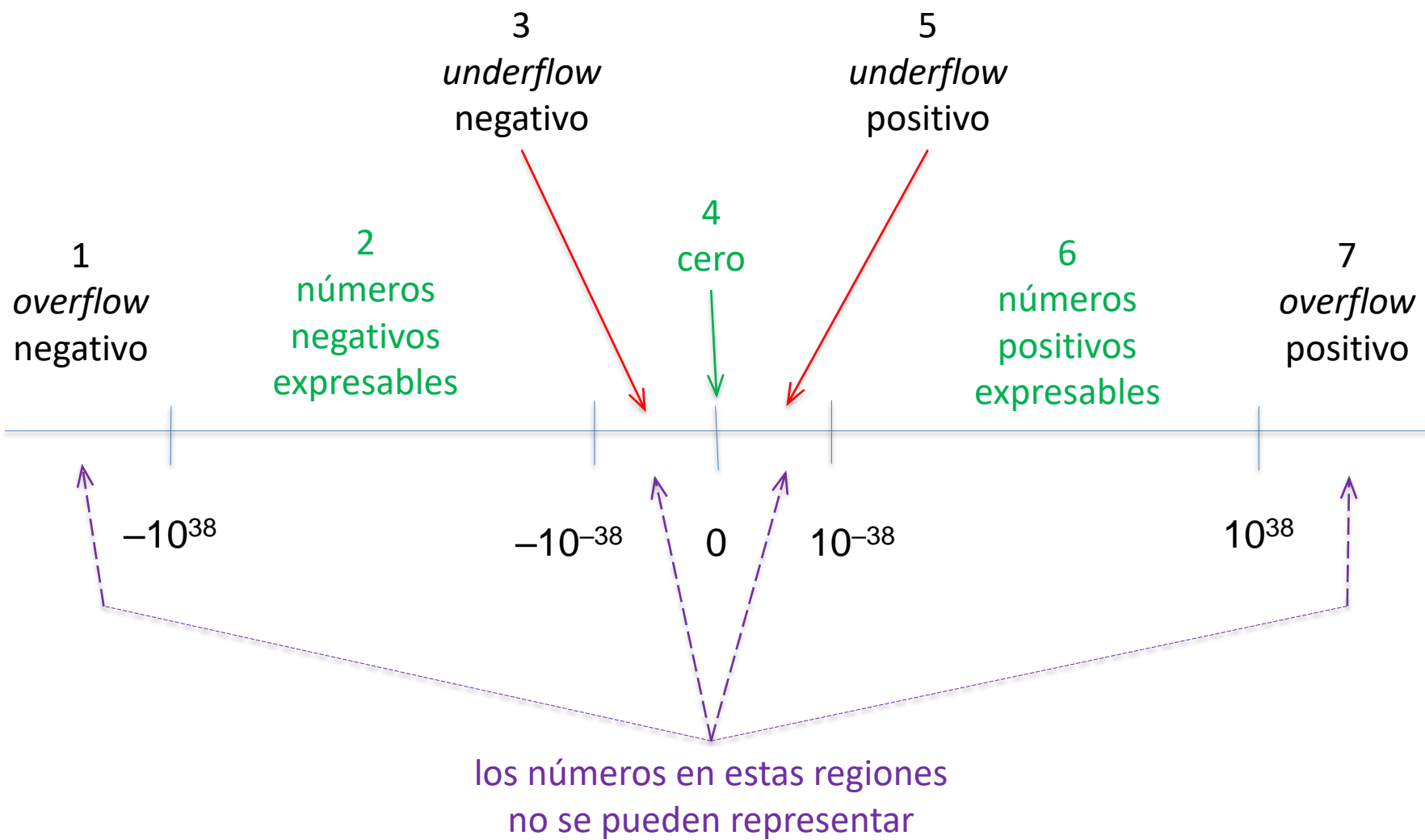
...

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

1: *overflow* negativo, 2: números negativos expresables, 3: *underflow* negativo, 4: *cero*, 5: *underflow* positivo, 6: números positivos expresables, y 7: *overflow* positivo

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

- p.ej., entre $1.000000000000000000000000_2$ y $1.111111111111111111111111_2$
... 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 exactamente, 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
→ el *significante* tiene 24 bits

signo:

- en el bit más significativo (bit 31)
→ 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:

en lugar de representarlos como números enteros en complemento de 2

... **se los representa como números sin signo desfasados (*biased*) en 127**

La forma de un número en punto flotante en IEEE 754 es

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

11000000101000000000000000000000

signo:

bit 31



exponente:

bits 23 al 30



fracción:

bits 0 al 22



| 1 | 10000001 | 010000000000000000000000 |

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

11000000101000000000000000000000

Identifiquemos sus partes:

| 1 | 10000001 | 010000000000000000000000 |

signo \rightarrow 1

exponente \rightarrow 129

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

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

$$\begin{aligned} (-1)^1 \times (1 + 0.25) \times 2^{129-127} &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= \mathbf{-5.0} \end{aligned}$$

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

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

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

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

Por lo tanto, $signo = 1$

$significante = 1.1 \rightarrow fracción = .1$

$exponente = -1 + 127 = 126$

... y la representación es

1 01111110 100000000000000000000000

Combinaciones especiales de exponente y fracción:

0 : *exponente* = 0 y *fracción* = 0

número no normalizado : *exponente* = 0 y *fracción* \neq 0

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

→ 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* \neq 0

El estándar IEEE 754 tiene también un formato de 64 bits

→ *precisión doble*:

exponente de 11 bits, con desfase de 1023

fracción de 52 bits → *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)

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

Suponemos significantes de 4 dígitos, y exponentes de dos dígitos

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

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

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 dígitos

...

...

Luego, sumamos los significantes, y normalizamos y redondeamos el resultado:

$$9.999 + 0.016 = 10.015 \quad \text{—sumamos significantes}$$

$$\rightarrow 10.015 \times 10^1 = 1.0015 \times 10^2 \quad \text{—normalizamos } *$$

$$\rightarrow \mathbf{1.002 \times 10^2} \quad \text{—redondeamos } **$$

* 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 incrementando 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

Multiplicación

Los operandos se llaman *multiplicando* y *multiplicador*

... y el resultado, *producto*

P.ej., multipliquemos 123 (multiplicando) por 45 (multiplicador):

	1	2	3	×	4	5	
	6	1	5				—producto intermedio de 123×5
+	4	9	2				—producto intermedio de 123×4
=	5	5	3	5			—suma de los productos intermedios

Algoritmo del colegio (versión informal, ver ej. en diap. anterior):

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 multiplicando 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

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

... cada paso de la multiplicación es simple:

$$\begin{array}{rcccccccc} & & & 1 & 0 & 1 & 0 & \times & 1 & 0 & 0 & 1 \\ & & & 1 & 0 & 1 & 0 & & = & 1010 & \times & 1 \\ & & 0 & 0 & 0 & 0 & & & = & 1010 & \times & 0 \\ & 0 & 0 & 0 & 0 & & & & = & 1010 & \times & 0 \\ + & 1 & 0 & 1 & 0 & & & & = & 1010 & \times & 1 \\ = & 1 & 0 & 1 & 1 & 0 & 1 & 0 & & \text{suma y producto final} \end{array}$$

Algoritmo de multiplicación binaria (versión informal, ver ej. en diap. anterior):

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

... o bien colocar 0 en el lugar correcto*, si el dígito del multiplicador es 0 (= 0 \times multiplicando)

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

Hardware necesario para multiplicación

El número de dígitos en el producto es mayor (¿cuánto mayor?) que el número de dígitos en cualquiera de los operandos → posibilidad de overflow

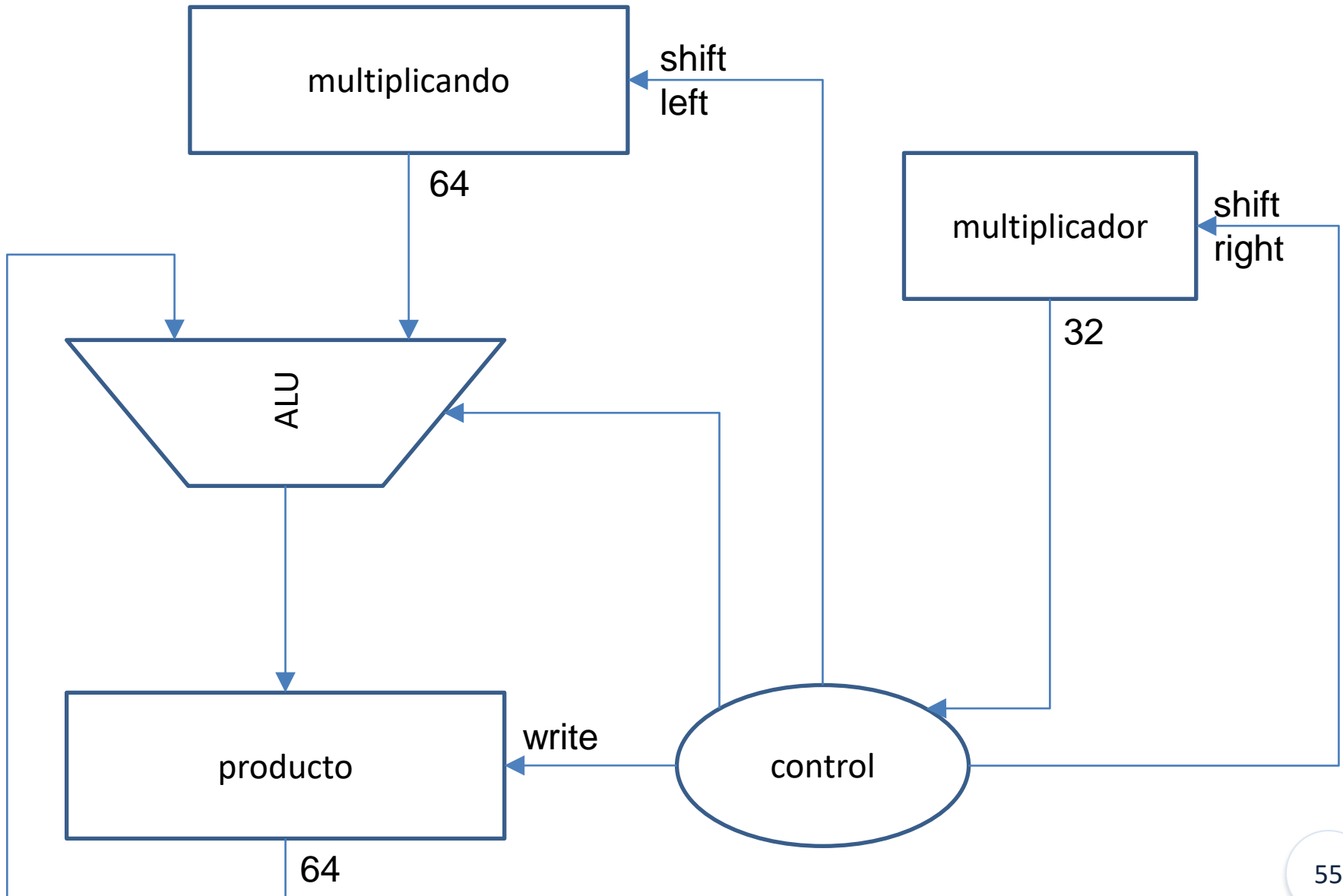
Necesitamos registros para

... el multiplicando (con capacidad de *shift* a la izquierda, ¿con cuántos bits?)

... el multiplicador (con capacidad de *shift* a la derecha, ¿por qué?) y

... el producto (con el doble de bits que los operandos)

Hardware para multiplicación (versión secuencial)



Algoritmo de multiplicación de números 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

Algoritmo de multiplicación de punto flotante

1. sumar los exponentes desfasados y restar el desfase a la suma para obtener el nuevo exponente desfasado
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

Ejemplo de multiplicación de números de punto flotante

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

Primero, sumamos los exponentes:

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

o desfasándolos (como quedan representados en el computador)

$$\rightarrow (-1+127) + (-2+127) - 127 = -3 + 127 = 124$$

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

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

...

...

Revisamos que el producto esté normalizado \rightarrow lo está

... y que el exponente no haya producido overflow o underflow:

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

Redondeamos el producto, sin efecto en este caso

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

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

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)