

# ARITMÉTICA EN COMA FLOTANTE

AMPLIACIÓN DE ESTRUCTURA DE COMPUTADORES

José Luis Risco, Carlos García, Eduardo Huedo  
Facultad de Informática

# Aritmética en coma flotante

1. Representación de números en coma flotante
2. IEEE 754
3. Suma y resta en coma flotante
4. Multiplicación en coma flotante
5. División en coma flotante

# Bibliografía

- "Computer architecture. A quantitative approach". Hennessy & Patterson, Morgan Kaufmann, 1995. Apéndice A (David Goldberg).
- "Computer arithmetic algorithms". I. Koren, Prentice Hall, 2002
- "Computer Arithmetic: Algorithms and Hardware Design", B.Parhami. Oxford UP. 2000
- "What every computer scientist should know about floating-point arithmetic", David Goldberg, ACM Computing Surveys. V.23, n.1, pg.5-18, 1991.

# Representación de n°s. en coma flotante

La representación en coma flotante está basada en la **notación científica**:

- La coma decimal no se halla en una posición fija dentro de la secuencia de bits, sino que su posición se indica como una potencia de la base:

$$\begin{array}{ccc} \text{signo} & & \text{exponente} \\ \underbrace{\quad} & & \underbrace{\quad} \\ + & 6.02 & \cdot 10^{-23} \\ \underbrace{\quad} & \underbrace{\quad} & \\ \text{mantisa} & & \text{base} \end{array} \qquad \begin{array}{ccc} \text{signo} & & \text{exponente} \\ \underbrace{\quad} & & \underbrace{\quad} \\ + & 1.01110 & \cdot 2^{-1101} \\ \underbrace{\quad} & \underbrace{\quad} & \\ \text{mantisa} & & \text{base} \end{array}$$

En todo número en coma flotante se distinguen tres componentes:

- **Signo**: indica el signo del número (0= positivo, 1=negativo)
- **Mantisa**: contiene la magnitud del número (en binario puro)
- **Exponente**: contiene el valor de la potencia de la base (sesgado)
- La **base** queda implícita y es común a todos los números, la más usada es 2.

El **valor** de la secuencia de bits (  $s, e_{p-1}, \dots, e_0, m_{q-1}, \dots, m_0$  ) es:  $(-1)^s \cdot V(m) \cdot 2^{V(e)}$

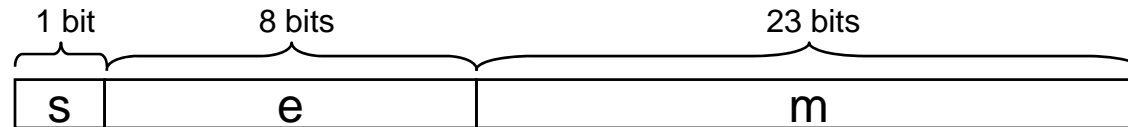
Dado que un mismo número puede tener varias representaciones ( $0.110 \cdot 2^5 = 110 \cdot 2^2 = 0.0110 \cdot 2^6$ )

los números suelen estar normalizados:

- un número está normalizado si tiene la forma  $1.xx... \cdot 2^{xx...}$  (ó  $0.1xx \dots \cdot 2^{xx...}$ )
- dado que los números normalizados en base 2 tienen siempre un 1 a la izquierda, éste suele quedar implícito (pero debe ser tenido en cuenta al calcular el valor de la secuencia)

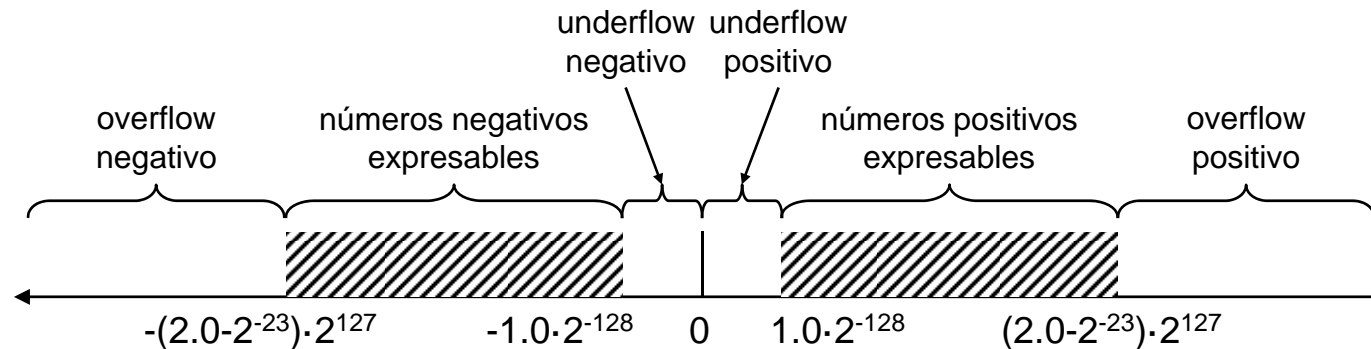
# Representación de n°s. en coma flotante

Sea el siguiente formato de coma flotante de 32 bits (base 2, normalizado)

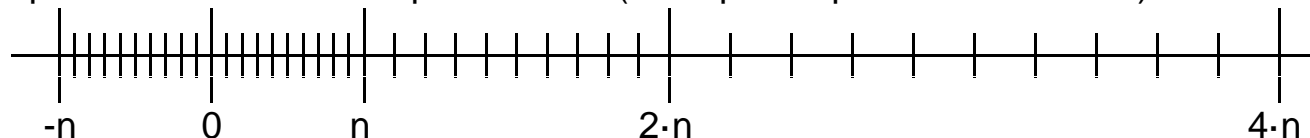


✓ El rango de valores representable por cada uno de los campos es:

- **Exponente** (8 bits con sesgo de 128) : -128 ... +127
- **Mantisa** (23 bits normalizados) : los valores binarios representables oscilan entre 1.00... y 1.11..., es decir entre 1 y  $2 \cdot 2^{-23}$  (2-ulp) (  $1.11...1 = 10.00...0 - 0.0...1$  )



✓ Obsérvese que la cantidad de números representables es  $2^{32}$  (igual que en coma fija). Lo que permite la representación en coma flotante es ampliar el rango representable a costa de aumentar el espacio entre números representable (un espacio que no es uniforme).



# IEEE 754

## IEEE Standard 754 for Binary Floating-Point Arithmetic.



**Prof. Kahan**

**1989  
ACM Turing  
Award Winner!**

<http://www.cs.berkeley.edu/~wkahan/>

# IEEE 754

**2 formatos** con signo explícito, representación sesgada del exponente (sesgo igual a  $(2^{n-1}-1=127)$ ), mantisa normalizada con un 1 implícito (1.M) y base 2.

- **precisión simple** (32 bits): 1 bit de signo, 8 de exponente, 23 de mantisa
  - $1.0 \cdot 2^{-126} \dots (2-2^{-23}) \cdot 2^{127} = 1.2 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$
- **precisión doble** (64 bits): 1 bit de signo, 11 de exponente, 52 de mantisa
  - $1.0 \cdot 2^{-1022} \dots (2-2^{-52}) \cdot 2^{1023} = 2.2 \cdot 10^{-308} \dots 1.8 \cdot 10^{308}$

**2 formatos** ampliados para cálculos intermedios (43 y 79 bits).

**Codificaciones** con significado especial

- **Infinito** ( $e=255, m=0$ ): representan cualquier valor de la región de overflow
- **NaN** (*Not-a-Number*) ( $e=255, m>0$ ): se obtienen como resultado de operaciones inválidas
- **Número denormalizado** ( $e=0, m>0$ ): es un número sin normalizar cuyo bit implícito se supone que es 0. Al ser el exponente 0, permiten representar números en las regiones de underflow. El valor del exponente es el del exponente más pequeño de los números no denormalizados: -126 en precisión simple y -1022 en doble.
- **Cero** ( $e=0, m=0$ ): número no normalizado que representa al cero (en lugar de al 1)

**Excepciones:**

- **Operación inválida:**  $\infty \pm \infty$ ,  $0 \times \infty$ ,  $0 \div 0$ ,  $\infty \div \infty$ ,  $x \bmod 0$ ,  $\sqrt{x}$  cuando  $x < 0$ ,  $x = \infty$
- **Inexacto:** el resultado redondeado no coincide con el real
- **Overflow y underflow**
- **División por cero**

## Example: Converting Binary FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

Sign: 0 => positive

Exponent:

$$-0110\ 1000_{\text{two}} = 104_{\text{ten}}$$

$$- \text{Bias adjustment: } 104 - 127 = -23$$

Significand:

$$\begin{aligned} &1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots \\ &= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} \\ &\quad + 2^{-22} = 1.0 + 0.666115 \end{aligned}$$

- Represents:  $1.666115_{\text{ten}} \times 2^{-23} \sim 1.986 \times 10^{-7}$   
(about 2/10,000,000)



## Example: Converting Decimal to FP

**$-2.340625 \times 10^1$**

1. Denormalize: **-23.40625**
2. Convert integer part:  
 **$23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$**
3. Convert fractional part:  
 **$.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$**
4. Put parts together and normalize:  
 **$10111.01101 = 1.011101101 \times 2^4$**
5. Convert exponent:  **$127 + 4 = 10000011_2$**

<b>1</b>	<b>1000 0011</b>	<b>011 1011 0100 0000 0000 0000</b>
----------	------------------	-------------------------------------

# Understanding the Significand (1/2)

## Method 1 (Fractions):

- In decimal:  $0.340_{10} \Rightarrow 340_{10}/1000_{10}$   
 $\Rightarrow 34_{10}/100_{10}$
- In binary:  $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$   
 $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$
- Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

# Understanding the Significand (2/2)

## Method 2 (Place Values):

- Convert from scientific notation
- In decimal:
  - $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
- In binary:
  - $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
- Interpretation of value in each position extends beyond the decimal/binary point
- Advantage: good for quickly calculating significand value; use this method for translating FP numbers

## Peer Instruction

1	1000 0001	111 0000 0000 0000 0000 0000
---	-----------	------------------------------

**What is the decimal equivalent of the floating pt # above?**

1:	-1.75
2:	-3.5
3:	-3.75
4:	-7
5:	-7.5
6:	-15
7:	-7 * 2 <sup>129</sup>
8:	-129 * 2 <sup>7</sup>

## Peer Instruction Answer

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^1 \times (1 + .111) \times 2^{(129-127)}$$

$$-1 \times (1.111) \times 2^{(2)}$$

$$-111.1$$

$$-7.5$$

1: -1.75

2: -3.5

3: -3.75

4: -7

5: -7.5

6: -15

7: -7 \* 2<sup>129</sup>

8: -129 \* 2<sup>7</sup>

# IEEE 754

El estándar exige que el resultado de las operaciones sea el mismo que se obtendría si se realizasen con precisión absoluta y después se redondease.

Hacer la operación con precisión absoluta no tiene sentido pues se podrían necesitar operandos de mucha anchura.

**Existen 4 modos de redondeo:**

- Redondeo al **más cercano** (al par en caso de empate)
- Redondeo a **más infinito** (por exceso)
- Redondeo a **menos infinito** (por defecto)
- Redondeo a **cero** (truncamiento)

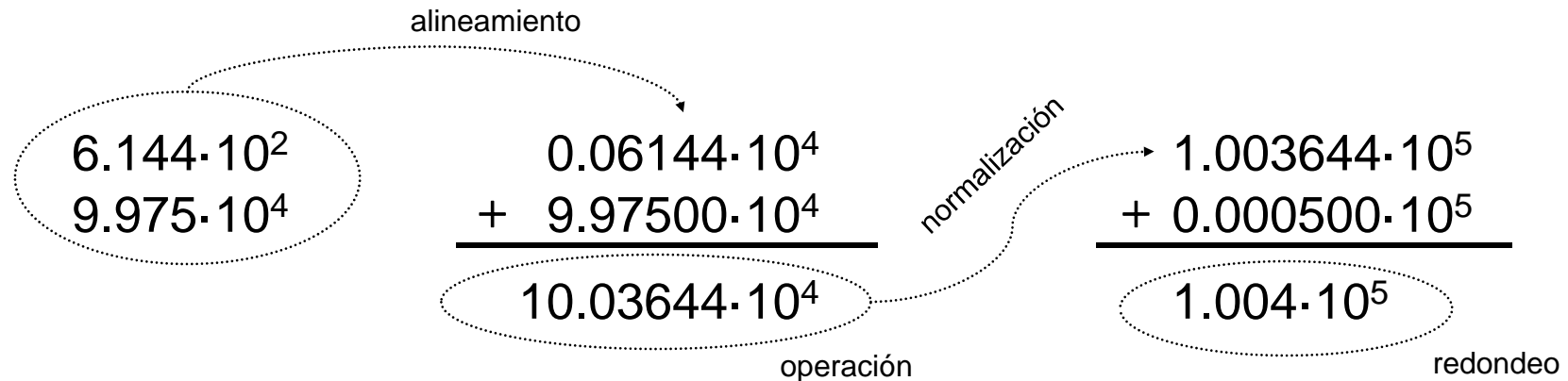
**Al realizar una operación ¿cuántos bits adicionales se necesitan para tener la precisión requerida?**

- Un bit **r** para el redondeo
- Un bit **s** (sticky) para determinar, cuando  $r=1$ , si el número está por encima de 0,5

**Tabla de operaciones de redondeo**

Tipo de redondeo	Signo del resultado $\geq 0$	Signo del resultado $< 0$
$-\infty$		+1 si (r or s)
$+\infty$	+1 si (r or s)	
0		
Más próximo	+1 si (r and $p_0$ ) or (r and s)	+1 si (r and $p_0$ ) or (r and s)

# Suma/resta en coma flotante



## Método de suma/resta :

- Extraer signos, exponentes y magnitudes.
- Tratar operandos especiales (por ejemplo, alguno de ellos a cero)
- Desplazar la mantisa del número con exponente más pequeño a la derecha  $|e_1 - e_2|$  bits
- Fijar el exponente del resultado al máximo de los exponentes
- Si la operación es suma y los signos son iguales, o si la operación es resta y los signos son diferentes, sumar las mantisas. En otro caso restarlas
- Detectar overflow de la mantisa
- Normalizar la mantisa, desplazándola a la derecha o a la izquierda hasta que el dígito más significativo esté delante de la coma decimal.
- Redondear el resultado y renormalizar la mantisa si es necesario.
- Corregir el exponente en función de los desplazamientos realizados sobre la mantisa.
- Detectar overflow o underflow del exponente

# Suma/resta en coma flotante

## Redondeo

El estándar exige que el resultado de las operaciones sea el mismo que se obtendría si se realizasen con precisión absoluta y después se redondease.

Hacer la operación con precisión absoluta no tiene sentido pues se podrían necesitar operandos de mucha anchura.

**Al realizar una operación ¿cuántos bits adicionales se necesitan para tener la precisión requerida?**

### Suma

- $1 \leq s_1 < 2$

- $ulp \leq s_2 < 2$

Por tanto  $s = s_1 + s_2$  cumplirá:

- $1 < s_3 < 4$

Si  $s_3 > 2$  se deberá normalizar desplazando a la derecha una posición, y ajustando el exponente.

Redondeo:

Caso 1:  $e_1 = e_2$  y  $s_3 > 2$

1.001000\*2<sup>3</sup>  
1.110001\*2<sup>3</sup>  
10.111001\*2<sup>3</sup>  
1.0111001\*2<sup>4</sup>  
Bit de redondeo

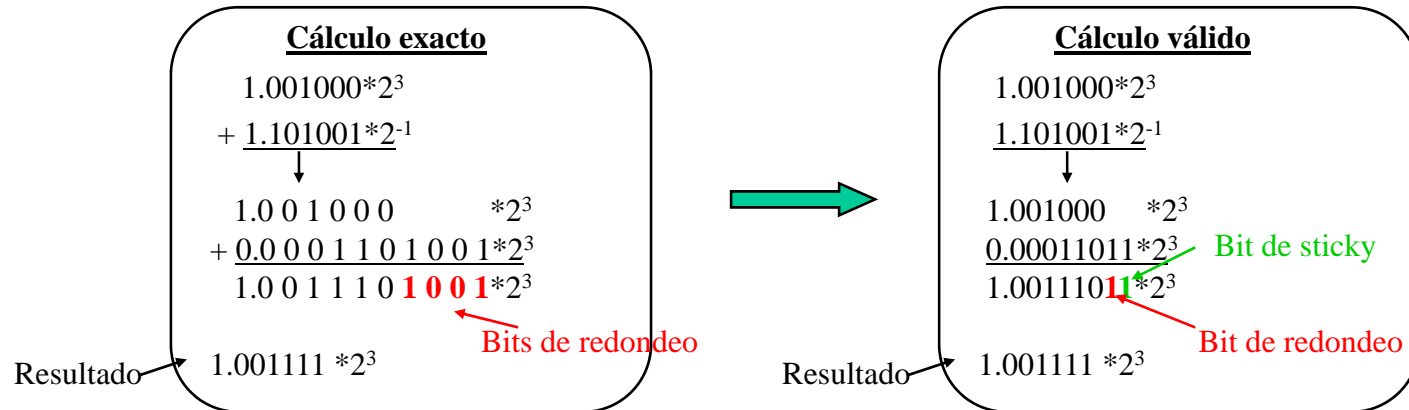


# Suma/resta en coma flotante

Suma

Redondeo:

Caso 2:  $e_1 - e_2 > 0$



# Suma/resta en coma flotante

## •Resta:

### •Caso 1: $e_1 = e_2$

$1 \leq s_1 < 2 \Rightarrow s \in [0, 1) \Rightarrow$  Normalización  
No se necesitan bits adicionales.

$$\begin{array}{r} 1.001111 * 2^3 \\ - 1.001001 * 2^3 \\ \hline 0.000110 * 2^3 \\ 1.100000 * 2^{-1} \end{array}$$

Normalización

### •Caso 2: $e_1 - e_2 = 1$ y $s > 0,5$

$$\left. \begin{array}{l} 1 \leq s_1 < 2 \\ 0.5 \leq s_2 < 1 \end{array} \right\} \text{ulp} \leq s < 1.5$$

#### Cálculo exacto

$$\begin{array}{r} 1.001111 * 2^3 \\ - 1.001001 * 2^2 \\ \hline 1.001111 * 2^3 \\ - 0.1001001 * 2^3 \\ \hline 0.1010101 * 2^3 \end{array}$$

Resultado

1.010101 \* 2<sup>2</sup> Bit de guarda

#### Cálculo exacto

$$\begin{array}{r} 1.111000 * 2^3 \\ - 1.100001 * 2^2 \\ \hline 1.001111 * 2^3 \\ - 0.1100001 * 2^3 \\ \hline 1.0001111 * 2^3 \end{array}$$

Resultado

1.001000 \* 2<sup>3</sup> Bit de redondeo

# Suma/resta en coma flotante

## •Resta:

Caso3:  $e_1 - e_2 > 1$

$$\left. \begin{array}{l} 1 \leq s_1 < 2 \\ \text{ulp} \leq s_2 < 0.5 \end{array} \right\} 0.5 < s \leq 2$$

**Cálculo exacto**

$$\begin{array}{r} 1.000000 * 2^3 \\ - 1.110001 * 2^{-1} \\ \hline 1.000000 \quad * 2^3 \\ - 0.0001110001 * 2^3 \\ \hline 0.111000 \textcolor{red}{1111} * 2^3 \end{array}$$

Resultado →  $1.110010 * 2^2$  Bits de redondeo



**Cálculo válido**

$$\begin{array}{r} 1.000000 * 2^3 \\ - 1.110001 * 2^{-1} \\ \hline 1.000000 \quad * 2^3 \\ - 0.0001110001 * 2^3 \\ \hline 0.111000 \textcolor{orange}{1} \textcolor{red}{11} * 2^3 \end{array}$$

Resultado →  $1.110010 * 2^2$  Bit de sticky Bit de redondeo Bit de guarda

# Suma/resta en coma flotante

## Algoritmo:

Sean  $e_1$  y  $e_2$  los exponentes de los dos operandos y  $m_1$ ,  $m_2$  las mantisas con el bit oculto.

**1.-** Si  $e_1 < e_2$  intercambiar los operandos. Esto garantiza que la diferencia de exponentes  $d = e_1 - e_2$  es siempre positiva.

Poner como exponente tentativo del resultado  $e = e_1$ .

**2.-** Si los signos de los operandos difieren, reemplazar  $m_2$  por  $C2(m_2)$ .

**3.-** Colocar  $m_2$  en un registro de  $p$  bits y desplazarlo a la derecha  $d$  posiciones (introduciendo unos si se ha complementado  $m_2$ ) para igualar los exponentes.

De los bits desplazados fuera de  $p$ , poner el más significativo en un biestable  $g$ , el siguiente más significativo en un biestable  $r$ , y la “o-lógica” del resto de bits en un biestable  $s$  (sticky).

**4.-** Calcular  $m$  como la suma de  $m_1$  y el contenido de  $p$  ( $m_2$  modificado).

Si (signo de  $a_1 \neq$  signo de  $a_2$ ) and (bit\_mas\_significativo( $m$ )=1) and (no carry-out) entonces  $m$  es negativo.

Reemplazar  $m$  con  $C2(m)$ .

Esto sólo puede ocurrir si  $d=0$ .

# Suma/resta en coma flotante

## 5.- Normalización y ajuste de r y s :

Caso 1. Si (signo de  $a_1$  = signo de  $a_2$ ) and (carry-out) desplazar m una posición a la derecha, introduciendo un 1.

•  $s = (g \text{ or } r \text{ or } s)$ .  $r = \text{bit que sale de m al desplazar}$ .

Caso 2. Si no estamos en el caso 1, desplazar a la izquierda hasta que el número esté normalizado.

(En el primer desplazamiento a la izquierda se introduce el contenido de g, en los restantes se introduce 0).

• Si no se ha necesitado desplazar  $\Rightarrow s = r \text{ or } s \quad r = g$

• Si se ha desplazado una posición  $\Rightarrow r = r \quad s = s$

• Si se ha desplazado más de una posición  $\Rightarrow r = s = 0$  (esto sólo ocurre si  $a_1$  y  $a_2$  tienen signos opuestos y el mismo exponente, por lo que la suma ha sido exacta).

Significa que los exponentes son iguales y  $a_2 > a_1$

mente. Si se produce carry-out, desplazar a la derecha y

## 7.- Signo del resultado:

Si (signo de  $a_1$  = signo de  $a_2$ ) este es el signo del resultado.

Si (signo de  $a_1 \neq$  signo de  $a_2$ ) el signo depende de cuál de los dos operandos fuera negativo, de si en el paso 1 se intercambiaron y de si en el paso 4 se reemplazó por su  $c_2$ , como se muestra en la tabla.

<i>Intercambio</i>	<i>C2</i>	<i>signo(a1)</i>	<i>signo(a2)</i>	<i>signo(resultado)</i>
si		+	-	-
si		-	+	+
no	no	+	-	+
no	no	-	+	-
no	si	+	-	-
no	si	-	+	+

# Multiplicación en coma flotante

Sean  $x$  e  $y$  dos números representados en coma flotante con valor:

$$\bullet X = (-1)^{s1} * 1.\text{mant1} * 2^{e1}$$

$$\bullet Y = (-1)^{s2} * 1.\text{mant2} * 2^{e2}$$

El producto de estos dos números será otro número  $z$  con valor:

$$\bullet Z = (-1)^{(s1 \oplus s2)} * (1.\text{mant1} * 1.\text{mant2}) * 2^{(e1 + e2 - \text{sesgo})}$$

El proceso de multiplicación tiene varios pasos:

**1.-** El signo del resultado es igual a la o-exclusiva de los signos de los operandos.

**2.-** La mantisa del resultado es igual al producto de las mantisas.

Este producto es sin signo.

Dado que los dos operandos están comprendidos entre 1 y 2 el resultado  $r$  será:

$$1 \leq r < 4$$

Este paso puede requerir una normalización, mediante desplazamiento a la derecha y ajuste del exponente resultado.

**3.-** Si la mantisa resultado es del mismo tamaño que la de los operandos habrá que redondear. El redondeo puede implicar la necesidad de normalización posterior.

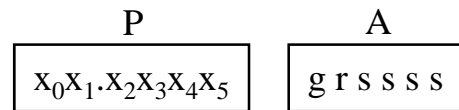
**4.-** El exponente del resultado es igual a la suma de los exponentes de los operandos.

Considerando que usamos una representación sesgada del exponente, al hacer esta suma estamos sumando dos veces el sesgo, y por tanto habrá que restar este sesgo una vez para obtener el resultado correcto.

# Multiplicación en coma flotante

## Pasos 2 y 3: Producto y redondeo

Supongamos que usamos un multiplicador secuencial que es capaz de multiplicar dos números almacenados en dos registros de  $p$  bits A y B, y almacenar el resultado en otro registro P de  $p$  bits (parte más significativa del producto) y en A (parte menos significativa).



Como el resultado final sólo se va a almacenar sobre el registro P, los bits contenidos en A nos servirán para redondear el resultado.

Como el número está comprendido entre 1 y 3,  $x_0$  puede ser tanto 1 como 0.

En cada uno de estos casos el redondeo se realiza de modo distinto:

- $x_0=0 \Rightarrow$  Desplazar P una posición a la izquierda, introduciendo el bit g de A como bit menos significativo de P. Los bit r y s (“o-lógica” de todos los s de A) nos sirven para redondear.
- $x_0=1 \Rightarrow$  La coma decimal se desplaza una posición a la izquierda, y se ajusta el exponente sumándole 1. Poner  $s = (r \text{ or } s)$  y  $r = g$ .

El redondeo siempre se hace de acuerdo a la tabla vista anteriormente.

# Multiplicación en coma flotante

## Overflow y underflow

El resultado redondeado es demasiado grande o pequeño para ser representado.

## Números desnormales:

El uso de este tipo de números complica el proceso de multiplicación.

Si el resultado del cálculo del exponente en una multiplicación es menor que el exponente mínimo representable, puede ocurrir que el resultado sea un número desnormalizado.

En general si el exponente es menor que  $EXP_{min}$ , la mantisa se desplazará a la derecha hasta que el exponente alcance  $EXP_{min}$ . Si toda la mantisa se ha desplazado fuera, se ha producido un underflow.

Al usar como factores de una multiplicación números desnormales, se opera como si fuesen números normales, considerando que el exponente no se calcula de igual modo.



# División en coma flotante

Sean  $x$  e  $y$  dos números representados en coma flotante con valor:

$$\bullet X = (-1)^{s1} * 1.mant1 * 2^{e1}$$

$$\bullet Y = (-1)^{s2} * 1.mant2 * 2^{e2}$$

La división de estos dos números será otro número  $z$  con valor:

$$\bullet Z = (-1)^{(s1 \oplus s2)} * (1.mant1 / 1.mant2) * 2^{(e1 - e2 + sesgo)}$$

- El proceso de división es similar al de multiplicación.
- Al hacer la resta de los exponentes hay que considerar que los sesgos se anularán y por tanto al resultado hay que sumarle el sesgo.
- Al operar con números normalizados, la mantisa del resultado será:  
 $0,5 < r < 2$   
Lo que implicará que la única normalización posible será mediante un desplazamiento a la izquierda, para lo que se necesitará un bit de guarda, **g**.

Junto con el bit de guarda necesitaremos al hacer la división añadir un bit más de redondeo, **r**, a la derecha del bit de guarda, y con el resto del resultado se hará la “o-lógica” en el bit **s**.