



Universidad
EAF

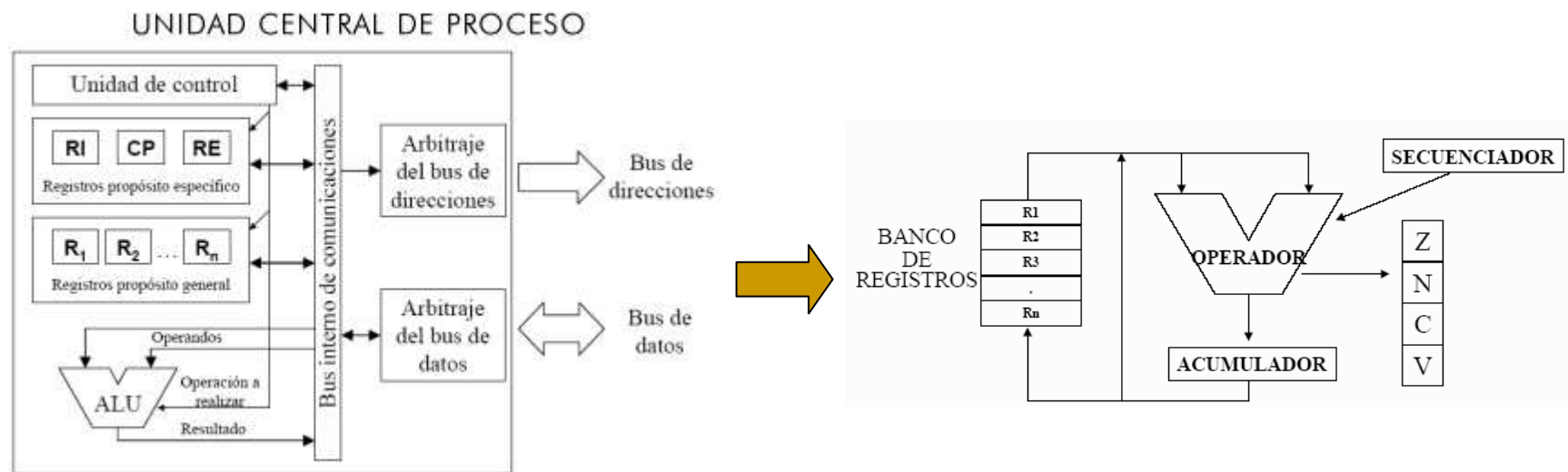
Aritmética del computador

Contenido

- **La unidad aritmético lógica (ALU)**
 - **Representación posicional. Sistemas numéricos**
 - **Representación de números enteros**
 - **Aritmética con enteros**
 - Circuito Sumador
 - Circuito Multiplicador
 - Mutiplicación secuencial
 - Algoritmo de Booth
 - División
 - Con restauración
 - Sin restauración
 - **Representación en coma flotante**
 - Estándar IEEE 754
 - **Aritmética en coma flotante**
 - Suma y resta
 - Multiplicación y división
 - Precisión
-

La unidad aritmético lógica (ALU)

- La ALU es la parte del computador que realiza las operaciones aritméticas y lógicas con los datos.
- La unidad aritmético-lógica se basa en el uso de dispositivos lógicos digitales sencillos que pueden almacenar dígitos binarios y realizar operaciones lógicas booleanas elementales.



Representación posicional. Sistemas numéricos

- Los sistemas de representación mas empleados son los denominados: **sistemas posicionales**.
- **TEOREMA FUNDAMENTAL DE LA NUMERACIÓN**
 - La fórmula general para construir un número ' N ' en un sistema de numeración posicional de base ' b ' es la siguiente:

$$N = d_{n-1} \dots d_1 d_0, d_{-1} \dots d_{-k}$$

$$N = \sum_{i=-k}^{n-1} d_i \cdot b^i$$

Sistema decimal

- El sistema decimal común es un sistema de numeración posicional que emplea 10 símbolos y donde la base es 10:
 - Símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$$1327 = 1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$


Pesos:	1000	100	10	1	
Símbolos:	<div>1</div>	<div>3</div>	<div>2</div>	<div>7</div>	
Valor:	1000	300	20	7	<div>→</div>
					<div>Suma</div> <div>1327</div>

Sistema binario

- Los sistemas digitales pueden representar de forma "natural" números en base 2, usando los símbolos {0,1}

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Pesos:	8	4	2	1	
Símbolos:	<div>1</div>	<div>1</div>	<div>0</div>	<div>1</div>	Suma
Valor:	8	4	0	1	<div>13</div>



Equivalencia entre los primeros 16 números

Decimal	Binario	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversión entre sistemas de numeración

■ Conversión al sistema decimal:

- Dada una cantidad expresada en un sistema con base b su valor en base decimal puede obtenerse por aplicación del Teorema Fundamental de la Numeración:

$$N = \sum_{i=-k}^{n-1} d_i \cdot b^i$$

■ Ejemplos:

$$X = 10101_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 1 = 21_{10}$$

$$X = 1C_{16} = 1 \cdot 16^1 + C \cdot 16^0 = 16 + 12 = 28_{10}$$

$$X = 374_8 = 3 \cdot 8^2 + 7 \cdot 8^1 + 4 \cdot 8^0 = 192 + 56 + 4 = 252_{10}$$

$$X = 101,01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 1 + 0,25 = 5,25_{10}$$

Conversión entre sistemas de numeración

- **Conversión de decimal a base B:**

- ❑ Parte entera: dividiendo sucesivamente por la base y tomando los restos
- ❑ Parte decimal: multiplicando sucesivamente por la base y tomando la parte entera

$$123_{(10)} \rightarrow 234_{(7)}$$

1 2 3 | 7

5 3 1 7 | 7

4 3 2

$$3,27_{(10)} = 11,010001_{(2)}$$

$$- 3_{(10)} = 11_{(2)}$$

- $0,27 \times 2 = 0,54 \rightarrow "0"$

- $0,54 \times 2 = 1,08 \rightarrow "1"$

- $0,08 \times 2 = 0,16 \rightarrow "0"$

- $0,16 \times 2 = 0,32 \rightarrow "0"$

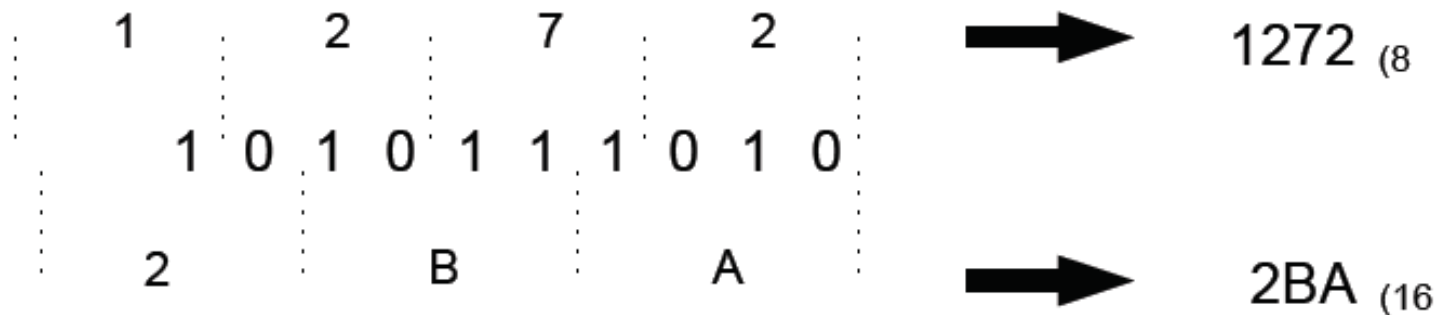
- $0,32 \times 2 = 0,64 \rightarrow "0"$

- $0,64 \times 2 = 1,28 \rightarrow "1"$

— . . .

Cambios entre las bases 2, 8 y 16

- 1 cifra octal -> 3 cifras binarias
- 1 cifra hexadecimal -> 4 cifras binarias



Representación en exceso a Z

- Un número binario representa su valor binario **menos** Z (binario puro):
 - 01010 en exceso 16 representa el número $10 - 16 = -6$
- La representación binaria de un número en exceso Z, se obtiene **sumando** Z al número
 - Representación de -6 en exceso 16 con 5 bits es 10: 01010
 - Representación de -8 en exceso 16 con 5 bits es 8: 01000
 - Representación de 8 en exceso 16 con 5 bits es 24: 11000
- El rango de valores que se pueden representar con n bits en exceso 2^{n-1} es:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

Números con signo. Resumen

x	s-m	Ca1	Ca2	exc. 2^{n-1}
-8	-	-	1000	0000
-7	1111	1000	1001	0001
-6	1110	1001	1010	0010
-5	1101	1010	1011	0011
-4	1100	1011	1100	0100
-3	1011	1100	1101	0101
-2	1010	1101	1110	0110
-1	1001	1110	1111	0111
0	0000/1000	0000/1111	0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

Cuidado con las conversiones



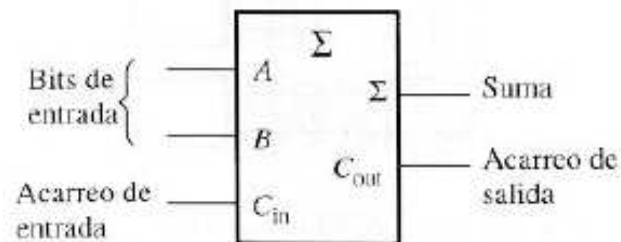
Guayana Francesa, 4 Junio 1996.
Un Cohete Ariane V de la Agencia Espacial Europea estalla a los 40 segundos del despegue. Era el primer lanzamiento del Ariane V, después de 10 años de desarrollo (7 billones de \$). El cohete y su carga estaban valorados en 800 millones de \$.

La explosión fue debida a un fallo software (Sistema de Referencia Inercial):

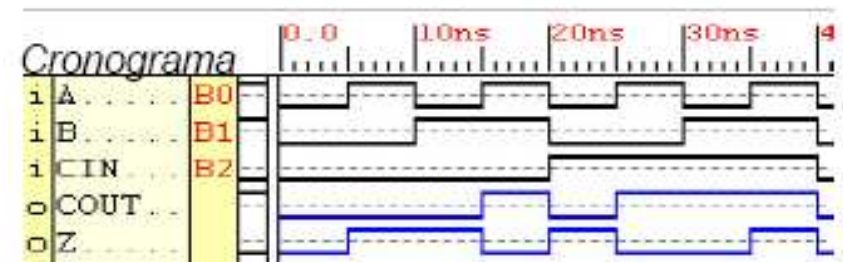
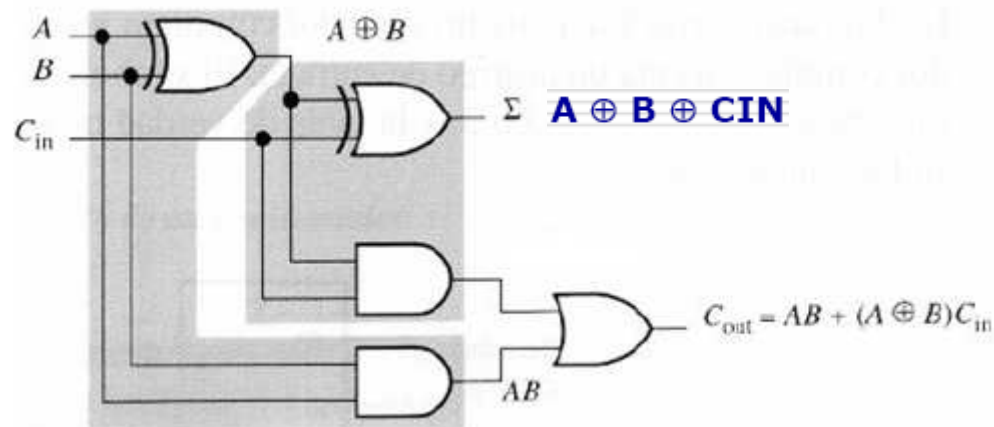
Un numero en punto flotante en doble precisión relacionado con la velocidad horizontal del cohete con respecto a la plataforma se convertía a un entero (con signo) de 16 bits. El número era superior a +32768 (el mayor entero con signo representable con 16 bits) y la conversión fallo.

Sumador completo

- El sumador completo (*full adder*) es un circuito que suma dos bits de entrada a_i y b_i más un acarreo de entrada c_{i-1} y devuelve un bit de resultado z_i y un bit de acarreo c_i



A	B	C _{in}	C _{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Multiplicación de enteros sin signo

- La multiplicación se suele realizar utilizando un sumador-restador y un algoritmo adecuado:

- Algoritmo Suma-desplazamiento
- Algoritmo de Booth

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

Multiplicando (11)
Multiplicador (13)
Productos Parciales
Producto (143)

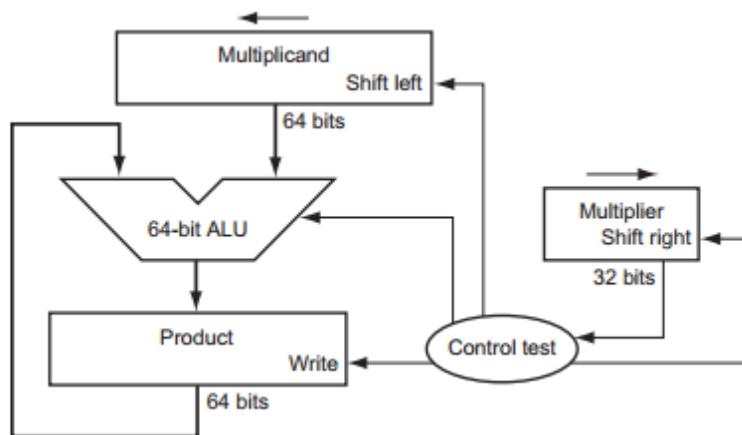
- **Método tradicional de multiplicación:**

- Obtener los productos parciales
- Cada producto parcial debe estar desplazado una posición a la izquierda respecto al producto parcial anterior
- Una vez calculados todos los productos parciales se suman para obtener el producto
- Para un multiplicando de n bits y un multiplicador de m bits el producto ocupa como máximo $n+m$ bits

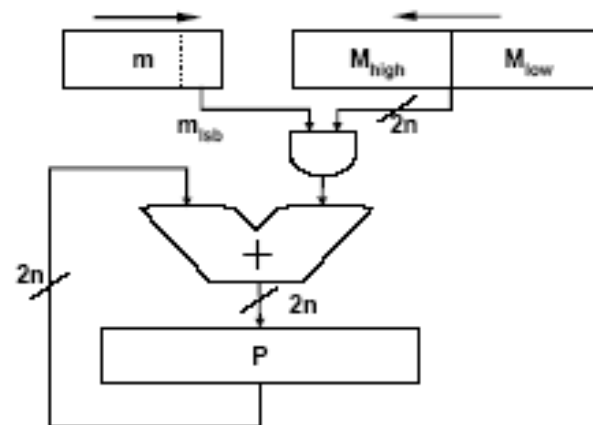
Multiplicación de enteros sin signo

■ Ruta de datos:

- ❑ 3 registros: multiplicando, multiplicador y producto
- ❑ 1 sumador de 2 entradas, en cada iteración sumar el producto parcial obtenido a la suma de los anteriores
- ❑ Para alinear correctamente los productos parciales, en cada iteración desplazar el multiplicando a la izquierda
- ❑ Para leer del mismo lugar cada uno de los bits del multiplicador, en cada iteración desplazarlo a la derecha



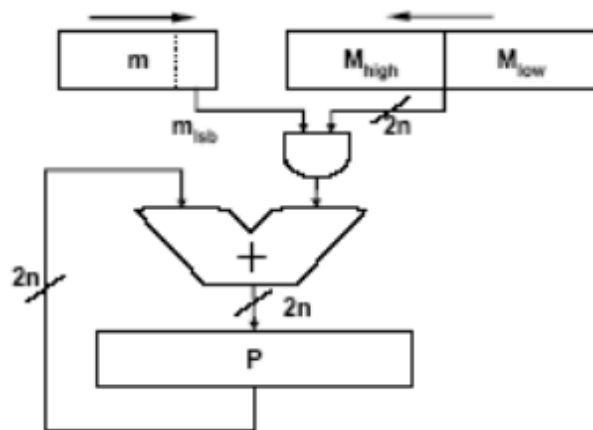
Ruta de datos:



Multiplicación de enteros sin signo

Algoritmo:

S0 : cargar multiplicador en m
 cargar multiplicando en M_{low}
 borrar M_{high}
 borrar P
 S1 : si $m_{lsb} = 1$ entonces $P \leftarrow P + M$
 si $m_{lsb} = 0$ entonces $P \leftarrow P + 0$
 S2 : desplazar M a la izquierda
 desplazar m a la derecha
 si S1-S2 no se han repetido n veces, ir a S1



Ejemplo

tras	m	M	P
S0	<u>1101</u>	00001011	00000000
S1	1101	00001011	00001011
S2	<u>0110</u>	00010110	00001011
S1	0110	00010110	00001011
S2	<u>0011</u>	00101100	00001011
S1	0011	00101100	00110111
S2	<u>0001</u>	01011000	00110111
S1	0001	01011000	10001111
S2	0000	10110000	10001111

Multiplicación de enteros con signo en C2

$$5 * 5 = 25$$

$$\begin{array}{r} 0101 \\ X 0101 \\ \hline 0101 \\ 0000 \\ 0101 \\ 0000 \\ \hline 00011001 \end{array}$$

$$(-5) * 5 = -25$$

$$\begin{array}{r} 1011 \\ X 0101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 0000 \\ \hline 00110111 \neq -25 \end{array}$$

$$(-5) * 5 = -25$$

$$\begin{array}{r} 1011 \\ X 0101 \\ \hline 11111011 \\ 00000000 \\ 111011 \\ 00000 \\ \hline 11100111 = -25 \end{array}$$



Los productos parciales deben representarse en C2

$$5 * (-5) = -25$$

$$\begin{array}{r} 0101 \\ X 1011 \\ \hline 0101 \\ 0101 \\ 0000 \\ 0101 \\ \hline 00110111 \neq -25 \end{array}$$

$$5 * (-5) = -25$$

$$\begin{array}{r} 0101 \\ X 1011 \\ \hline 00000101 \\ 0000101 \\ 000000 \\ \mathbf{11011} \\ \hline 11100111 = -25 \end{array}$$

En la última iteración hay que restar el multiplicando

Algoritmo de Booth

- Permite multiplicar directamente enteros representados en C2.
- Evita ejecutar sumas consecutivas cuando el multiplicador presenta cadenas de 0s o de 1s.
- **Idea:** Convertir el **multiplicador** en un número recodificado sobre un sistema binario no canónico bajo la forma de dígitos con signo:
 - Sistema binario canónico $D=\{0,1\} \Rightarrow$ Sistema binario no canónico $D=\{-1,0,1\}$.
- A la hora de realizar el algoritmo basta almacenar el valor binario del multiplicador y fijarse en el bit anterior para averiguar si se trata de un +1, un -1 o un 0 en la codificación de Booth.
- Las combinaciones:

a_i	a_{i-1}	operación
0	0	No hacer nada
0	1	Sumar b
1	0	Restar b
1	1	No hacer nada

“00” y “11” corresponden a “0”
“01” corresponde a un “+1”
“10” corresponde a un “-1”

Algoritmo de Booth

Bits del multiplicador	Dígito recodificado	Operación a realizar
$Y_i Y_{i+1}$	Z_i	
0 0	0	0 * multiplicando
0 1	1	1 * multiplicando
1 0	-1	-1 * multiplicando
1 1	0	0 * multiplicando

- $011101 \Leftrightarrow 100-11-1$

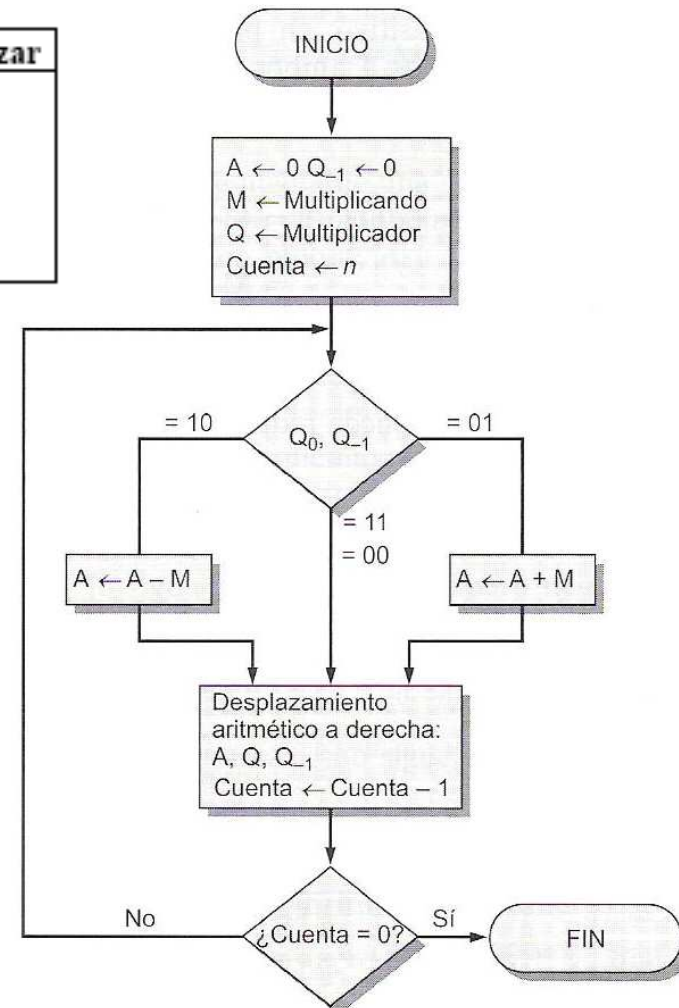
$$V(a) = 2^4 + 2^3 + 2^2 + 2^0 = 29$$

$$V(a') = 2^5 - 2^2 + 2^1 - 2^0 = 32 - 4 + 2 - 1 = 29$$

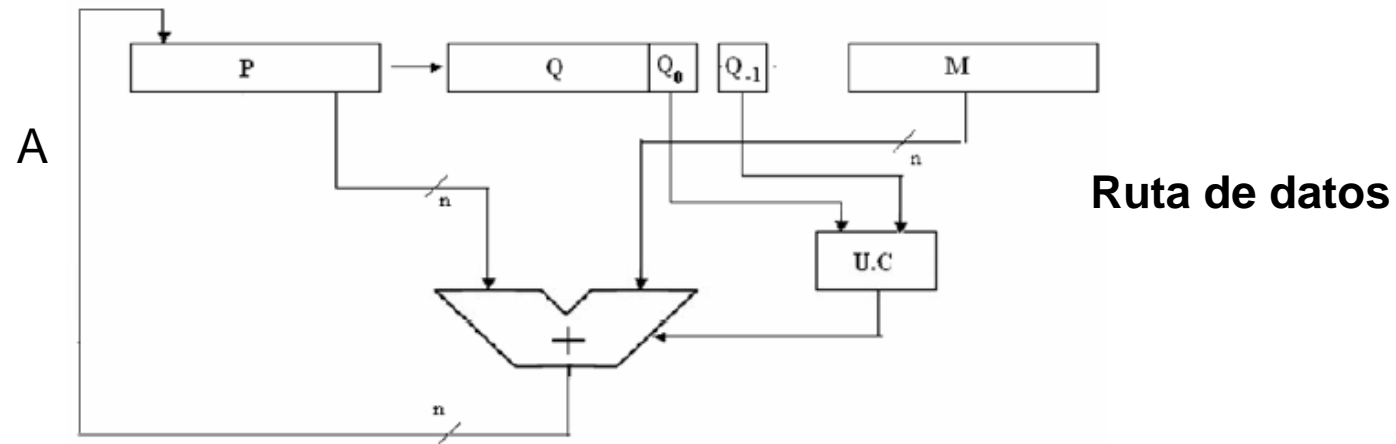
- $110010 \Leftrightarrow 0-101-10$

$$V(b) = -V(001110) = -(2^3 + 2^2 + 2^1) = -14$$

$$V(b') = -2^4 + 2^2 - 2^1 = -14$$



Algoritmo de Booth



■ Algoritmo de multiplicación

1. M multiplicando Q multiplicador $P = Q_{-1} \leftarrow 0$
2. Si $Q_0 Q_{-1} = 00$ ó $Q_0 Q_{-1} = 11 \Rightarrow Dder(P, Q, Q_{-1})$
Si $Q_0 Q_{-1} = 10 \Rightarrow P \leftarrow P - M \mid Dder(P, Q, Q_{-1})$
Si $Q_0 Q_{-1} = 01 \Rightarrow P \leftarrow P + M \mid Dder(P, Q, Q_{-1})$

- El paso 2 se realiza n veces, siendo n el número de bits del multiplicador. Una vez finalizado, el resultado se hallará en los registros P (parte más significativa) y Q (parte menos significativa).

Algoritmo de Booth

3/6/2019

■ Ejemplo:

A	Q	Q ₋₁	M		
0000	0011	0	0111	Valores iniciales	
1001	0011	0	0111	$A \leftarrow A - M$	} Primer ciclo
1100	1001	1	0111	Desplazamiento	
1110	0100	1	0111	Desplazamiento	} Segundo ciclo
0101	0100	1	0111	$A \leftarrow A + M$	
0010	1010	0	0111	Desplazamiento	} Tercer ciclo
0001	0101	0	0111	Desplazamiento	
					} Cuarto ciclo

División de enteros sin signo

- Al igual que con la multiplicación, para realizar esta operación se utiliza un sumador-restador y un algoritmo adecuado.
- **Método tradicional de división:**
 - Examen de los bits del dividendo de izquierda derecha, hasta que el divisor sea capaz de dividir al dividendo.
 - si el resto parcial es mayor que el divisor, añadir un 1 al cociente; el nuevo resto parcial será la resta del resto parcial y del divisor
 - si el resto parcial es menor que el divisor, añadir un 0 al cociente y ampliar el resto parcial con un bit más del dividendo.

Suposiciones:
Dividendo= $D \Rightarrow 2n$ bits
Divisor= $d \Rightarrow n$ bits
Cociente: $q \Rightarrow n$ bits
Resto: $r \Rightarrow n$ bits

Restricciones:
 $0 \leq r < d$
 $0 < d \leq D < 2^n \cdot d \Rightarrow 0 < q < 2^n$

Impide:
• División por cero
• Cociente cero
• Rebose del cociente

Dividendo: D
10010011 | 1011 Divisor: d
- 1011
001110
- 1011
001111
- 1011
100 Resto: r
1101 Cociente: q

restos parciales

$$D = d * q + r$$

147 / 11 = 13, resto=4

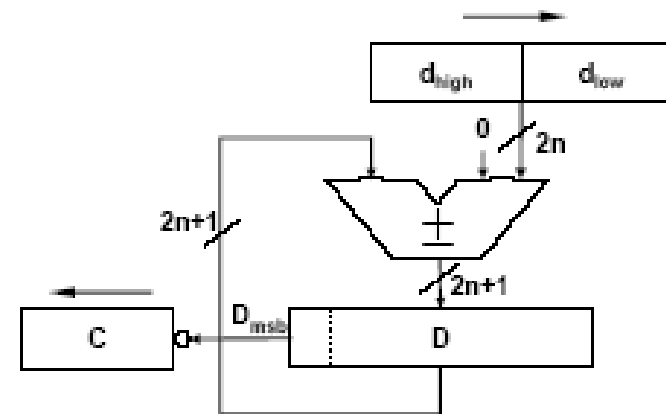
División con restauración

- Usar 3 registros: resto/dividendo, divisor y cociente
- Para alinear correctamente los restos parciales y el divisor, en cada iteración desplazar el divisor a la derecha
- Para escribir en el mismo lugar cada uno de los bits del cociente, en cada iteración desplazarlo a la izquierda
- Para evitar tener un comparador y un restador, usar éste último para comparar: el signo de la resta determinará si el resto parcial “cabe” entre el divisor

```

S0  : cargar (0,dividendo) en D
      cargar divisor en  $d_{high}$ 
       $d_{low} = 0$ 
       $C = 0$ 
S1  :  $D \leftarrow D - (0, d)$ 
S2  : si  $D_{msb} = 0$  entonces
      desplazar C a la izquierda insertando un 1
      si  $D_{msb} = 1$  entonces
      desplazar C a la izquierda insertando un 0
       $D \leftarrow D + (0, d)$ 
      desplazar d a la derecha
      si S1-S2 no se han repetido  $n+1$  veces ir a S1

```



División con restauración

■ Ejemplo:

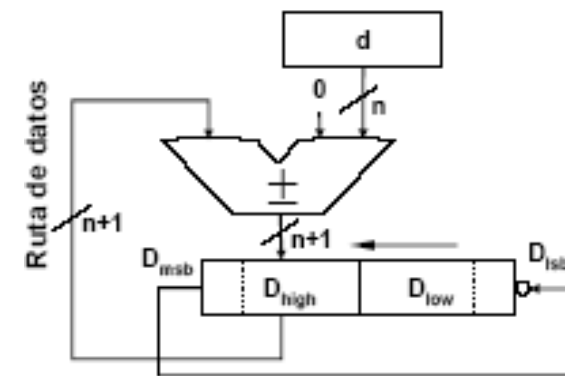
<i>tras</i>	<i>D</i>	<i>d</i>	<i>C</i>
S0	010010011	10110000	0000
S1	111100011	10110000	0000
S2	010010011	01011000	0000
S1	000111011	01011000	0000
S2	000111011	00101100	0001
S1	000001111	00101100	0001
S2	000001111	00010110	0011
S1	111111001	00010110	0011
S2	000001111	00001011	0110
S1	000000100	00001011	0110
S2	000000100	00000101	1101

147	11*16=178	
-29		0 (0)
147	11*8 = 88	
59		1 (1)
59	11*4=44	
15		1 (3)
15	11*2=22	
-7		0 (6)
15	11	
4		1 (13)
4	11	13

División sin restauración

- División sin restauración
 - considerar la secuencia de operaciones que se realiza tras la resta en S2:
 - si $D_{msb} = 0$ ("cabe") se desplaza D a la izquierda y se resta d. Queda: $2 \cdot D - d$
 - si $D_{msb} = 1$ ("no cabe") se suma d, se desplaza el resultado y se resta d. Queda: $2(D+d) - d = 2 \cdot D + d$
 - entonces, en lugar de restaurar:
 - sumar o restar d en función de D_{msb}
 - en la última iteración restaurar el resto (sumándole d) si es necesario

```
S0: cargar (0,dividendo) en D
    cargar divisor en d
S1: desplazar D a la izquierda
S2:  $D_{high} \leftarrow D_{high} - (0,d)$ 
S3: si  $D_{msb} = 0$  entonces  $D_{lsb} \leftarrow 1$ 
    si  $D_{msb} = 1$  entonces  $D_{lsb} \leftarrow 0$ 
S4: si  $D_{msb} = 0$  entonces: a) desplazar D a la izquierda
                           b)  $D_{high} \leftarrow D_{high} - (0,d)$ 
    si  $D_{msb} = 1$  entonces: a) desplazar D a la izquierda
                           b)  $D_{high} \leftarrow D_{high} + (0,d)$ 
    si S3-S4 no se han repetido n-1 veces ir a S3
S5: si  $D_{msb} = 0$  entonces  $D_{lsb} \leftarrow 1$ 
    si  $D_{msb} = 1$  entonces  $D_{high} \leftarrow D_{high} + (0,d)$ ,  $D_{lsb} \leftarrow 0$ 
```



División sin restauración

■ Ejemplo

<i>tras</i>	<i>D</i>	<i>d</i>	
S0	01001 0011	1011	(9, 3)
S1	<u>10010</u> 011 0	1011	(18, 6) <
S2	<u>00111</u> 011 0	1011	(7, 6) R
S3	00111 011 <u>1</u>	1011	(7, 6) 1
S4	<u>01110</u> 11 10	1011	(14, 12) <
S4	<u>00011</u> 11 10	1011	(3, 12) R
S3	00011 11 <u>11</u>	1011	(3, 12) 1
S4	<u>00111</u> 1 110	1011	(7, 8) <
S4	<u>11100</u> 1 110	1011	(-4, 8) R
S3	11100 1 <u>110</u>	1011	(-4, 8) 0
S4	<u>11001</u> 1100	1011	(-7, 0) <
S4	<u>00100</u> 1100	1011	(4, 0) S
S5	00100 <u>1101</u>	1011	(4, 0) 1

Algoritmo división con signo

- Inicializar registros:
 - $M \Rightarrow$ divisor expresado en n bits siendo n el número de bits del dividendo
 - $A, Q \Rightarrow$ dividendo expresado en C2 de $2n$ bits
- Desplazar A y Q una posición de bit a la izquierda
- Si M y A tienen el mismo signo:
 - $A = A - M$
- En caso contrario:
 - $A = A + M$
- La operación anterior tiene éxito si el signo de A es el mismo antes y después de la operación:
 - Si la operación tiene éxito ó $(A=0 \text{ AND } Q=0)$ $Q_0 = 1$
 - Si la operación no tiene éxito y $(A \neq 0 \text{ OR } Q \neq 0)$ $Q_0 = 0$ y restablecer el valor anterior de A
- Repetir los pasos 2 a 4 tantas veces como número de bits tenga Q
- El resto está en A . Si los signos del divisor y del dividendo eran iguales, el cociente está en Q ; en caso contrario, el cociente correcto es el C2 de Q .

Algoritmo división con signo: ejemplo

Ejemplo

$(-7) : (-3)$

	A	Q	M	-M
S1:	1111	1001	1101	0011
S2:	1111	001?	Desplazamiento	
	<u>0011</u>		Restar	
S3:	0010			
S4:	1111	0010	Restablecer y $Q_0=0$	
S2:	1110	010?	Desplazamiento	
	<u>0011</u>		Restar	
S3:	0001			
S4:	1110	0100	Restablecer y $Q_0=0$	
S2:	1100	100?	Desplazamiento	
	<u>0011</u>		Restar	
S3:	1111			
S4:	1111	1001	$Q_0=1$	
S2:	1111	001?	Desplazamiento	
	<u>0011</u>		Restar	
S3:	0010			
S4:	1111	0010	Restablecer y $Q_0=0$	
Resto: 1111 = $-(0001)_2 = -1$				
Cociente: $(0010)_2 = -2$				

Representación en coma flotante (I)

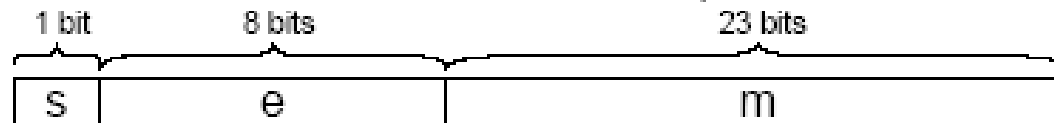
- La representación en coma flotante está basada en la **notación científica**:
 - La coma decimal no se halla en una posición fija dentro de la secuencia de bits, sino que su posición se indica como una potencia de la base:

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

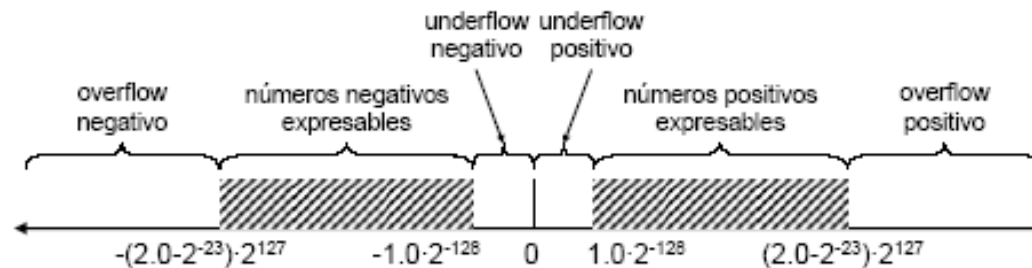
- En todo número en coma flotante se distinguen tres componentes:
 - **Signo**: indica el signo del número (0= positivo, 1=negativo)
 - **Mantisa**: contiene la magnitud del número (en binario puro)
 - **Exponente**: contiene el valor de la potencia de la base (sesgado)
 - La **base** queda implícita 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 $1,xx \dots \cdot 2^{xx \dots}$ (ó $0.1xx \dots \cdot 2^{xx \dots}$)
 - 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 en coma flotante (II)

- 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 $2 \cdot 2^{-23}$



Estándar IEEE 754

- **2 formatos** con signo explícito, representación sesgada del exponente (sesgo igual a $(2^{n-1}-1=127)$), mantisa normalizada con 1 bit implícito (**1.Mantisa**) 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
 $1.0 \cdot 2^{-1022} \dots (2-2^{-52}) \cdot 2^{1023} = 2.2 \cdot 10^{-38} \dots 1.8 \cdot 10^{38}$
- **2 formatos** ampliados para cálculos intermedios (43 y 79 bits).
- **Codificaciones** con significado especial

Exponente	Mantisa	Objeto representado
todos ceros	todos ceros	CERO
todos ceros	distinta de cero	número no normalizado
todos unos	todos ceros	∞
todos unos	distinta de cero	NaN (<i>Not a number</i>)

- **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**

Precisión

- 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.
- El IEEE 754 ofrece cuatro modos de redondeo:
 - Redondeo **al mas cercano** (al par en caso de empate)
 - Redondeo **a mas 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 lógica de los bits que se desprecian)

Ejemplo. Redondear a 5 decimales.

Tabla de operaciones de redondeo

Tipo de redondeo	Signo del resultado ≥ 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)

	Redondeo al más cercano	Redondeo a $+\infty$	Redondeo a $-\infty$	Redondeo a cero
+1,01100 11	+1,01101	+1,01101	+1,01100	+1,01100
+1,10011 10	+1,10100	+1,10100	+1,10011	+1,10011
-1,10011 01	- 1,10011	- 1,10011	-1,10100	- 1,10011

Aritmética en coma flotante

- La siguiente tabla resume las operaciones básicas de la aritmética en coma flotante:
 - En sumas y restas es necesario igualar los exponentes
 - La multiplicación y división son más directas.

Números en punto flotante	Operaciones aritméticas
$X = X_s \times B^{X_E}$ $Y = Y_s \times B^{Y_E}$	$\left. \begin{aligned} X + Y &= (X_s \times B^{X_E - Y_E} + Y_s) \times B^{Y_E} \\ X - Y &= (X_s \times B^{X_E - Y_E} - Y_s) \times B^{Y_E} \end{aligned} \right\} X_E \leq Y_E$ $X \times Y = (X_s \times Y_s) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_s}{Y_s} \right) \times B^{X_E - Y_E}$

Ejemplos:

$$X = 0,3 \times 10^2 = 30$$

$$Y = 0,2 \times 10^3 = 200$$

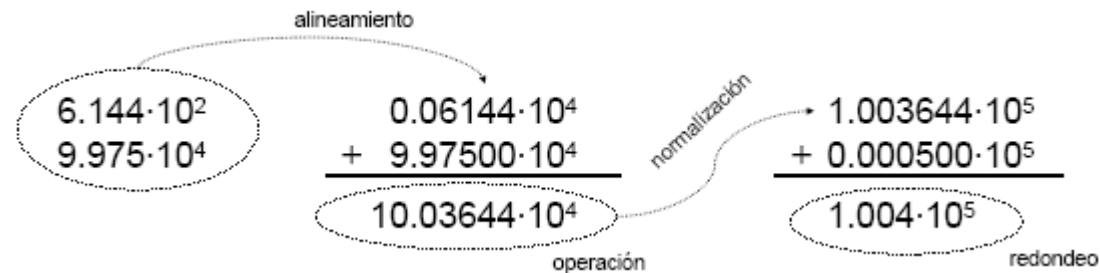
$$X + Y = (0,3 \times 10^{2-3} + 0,2) \times 10^3 = 0,23 \times 10^3 = 230$$

$$X - Y = (0,3 \times 10^{2-3} - 0,2) \times 10^3 = (-0,17) \times 10^3 = -170$$

$$X \times Y = (0,3 \times 0,2) \times 10^{2+3} = 0,06 \times 10^5 = 6.000$$

$$X \div Y = (0,3 \div 0,2) \times 10^{2-3} = 1,5 \times 10^{-1} = 0,15$$

Suma/resta en coma flotante (I)



■ 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 (II)

■ 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.

Suma

$$\bullet 1 \leq s_1 < 2$$

$$\bullet \text{ulp} \leq s_2 < 2$$

Por tanto $s = s_1 + s_2$ cumplirá:

$$\bullet 1 < s_3 < 4$$

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 $s_3 > 2$

$$\begin{array}{r} 1.001000 * 2^3 \\ 1.110001 * 2^3 \\ 10.111001 * 2^3 \\ 1.0111001 * 2^4 \end{array}$$

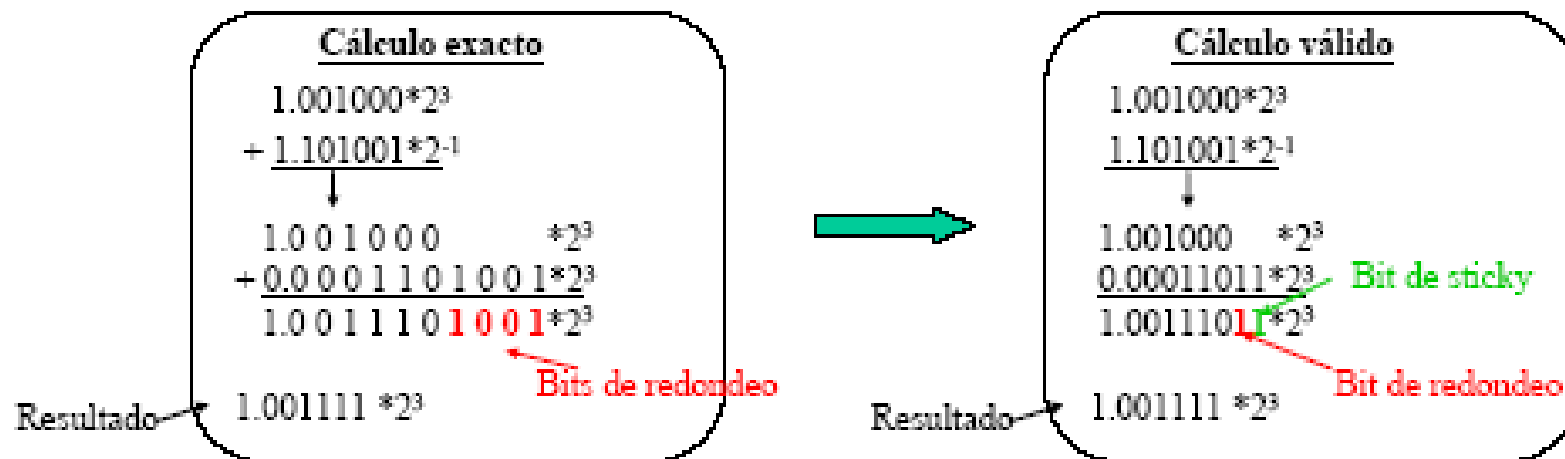
Bit de redondeo

Suma/resta en coma flotante (III)

Suma

Redondeo:

Caso 2: $e_1 - e_2 > 0$



Suma/resta en coma flotante (IV)

•Resta:

•Caso 1: $e_1 = e_2$

$1 \leq s_i < 2 \Rightarrow s \in [0,1) \Rightarrow$ 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 $s > 0,5$

$1 \leq s_1 < 2$
 $0.5 \leq s_2 < 1$ } $ulp \leq s < 1.5$

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 $\rightarrow 1.010101 * 2^2$ Bit de guarda

Cálculo exacto

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

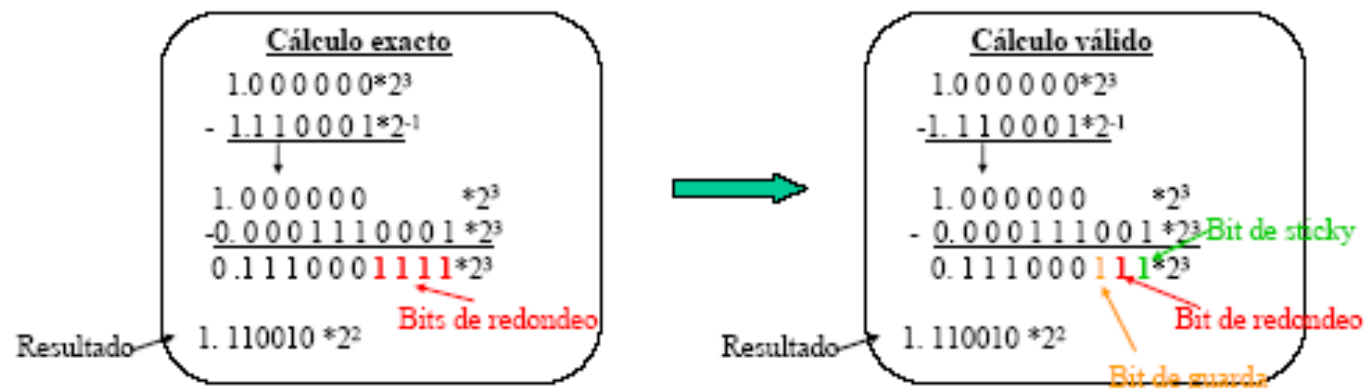
Resultado $\rightarrow 1.001000 * 2^3$ Bit de redondeo

Suma/resta en coma flotante (V)

•Resta:

Caso3: $e_1 - e_2 > 1$

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



Multiplicación coma flotante

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

- $X = (-1)^{s1} * 1, mant1 * 2^{e1}$

- $Y = (-1)^{s2} * 1, mant2 * 2^{e2}$

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

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

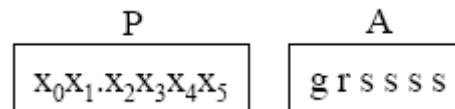
Algoritmo de multiplicación

- El proceso de multiplicación tiene varios pasos:
 - El signo del resultado es igual a la o-exclusiva de los signos de los operandos.
 - 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.
 - 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.
 - 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.
-

Algoritmo de multiplicación

■ 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:

Tabla de operaciones de redondeo

Tipo de redondeo	Signo del resultado >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)

División coma flotante

- Sean x e y dos números representados en coma flotante con valor:
 - $X = (-1)^{s1} \cdot 1, \text{mant1} \cdot 2^{e1}$
 - $Y = (-1)^{s2} \cdot 1, \text{mant2} \cdot 2^{e2}$
- La división de estos dos números será otro número z con valor:
 - $Z = (-1)^{(s1 \oplus s2)} \cdot (1, \text{mant1}/1, \text{mant2}) \cdot 2^{(e1-e2+\text{sesgo})}$

Algoritmo de división

- 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**.
-