Universidad Rey Juan Carlos

Repaso de Introducción a la Informática

Grado de Ingeniería del Software

Operaciones Aritmético-Lógicas

Ángel Serrano Sánchez de León

Índice

INTRODUCCIÓN

- En los computadores, cualquier tipo de cálculo se reduce a un conjunto muy sencillo de operaciones numéricas básicas:
  - **Aritméticas:** sumar, restar, multiplicar, dividir, cambio de signo, etc.
  - **Lógicas:** AND, OR, NOT, XOR, etc.
  - **De desplazamiento de bits:** movimiento de los bits hacia la derecha/izquierda, rotaciones, etc.
- Las operaciones aritméticas y las de desplazamiento de bits dependen del sistema de representación numérica.

INTRODUCCIÓN

- El tamaño de la representación numérica en los computadores está limitado ➔ **Rango representable**.
- Números enteros sin signo (coma fija): Binario puro con n bits (p bits para parte entera + q bits para parte fraccionaria)
- Números enteros con signo (coma fija): Complemento a 2 con n bits (p bits para parte entera + q bits para parte fraccionaria)
- Números reales (coma flotante): IEEE 754 (precisión simple con 4 bytes o doble con 8 bytes)
- Si necesitamos más bits para un cierto número, no sería representable ➔ **Desbordamiento** (*overflow*, V).

Aritmética en Binario Puro

- Sumar
- Restar
- Multiplicar
- Dividir
- Cambio de signo
- Extensión de signo

SUMAR EN BINARIO PURO

- En las sumas, pueden surgir **acarreos** (*carry*), que son bits adicionales que se han de sumar en la columna inmediatamente a la izquierda.
- Las reglas de sumar son más fáciles que en base 10:

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \qquad \begin{array}{rl} 1 & \leftarrow \text{Sumando} \\ +\,1 & \leftarrow \text{Sumando} \\ \hline \underset{\text{acarreo}}{\color{red}{1}}0 & \leftarrow \text{Suma} \end{array}$$

- No confundir la suma aritmética (+) con la suma lógica (OR).

SUMAR EN BINARIO PURO

- **Condición de desbordamiento:** para datos de n bits, el resultado necesita $n + 1$ bits.
- Ejemplo. Supongamos datos de $n = 4$ bits:

$(5 + 7)_{10} = (0101 + 0111)_{BP}$

$$\begin{array}{rrrrr} & 1 & 1 & 1 & \quad \text{acarreo} \\ \hline & 0 & 1 & 0 & 1 \\ + & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Resultado = 12_{10}

$(9 + 11)_{10} = (1001 + 1011)_{BP}$

$$\begin{array}{rrrrr} & & 1 & 1 & \quad \text{acarreo} \\ \hline & 1 & 0 & 0 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 \end{array}$$

Resultado = 4_{10}

¡¡Desbordamiento!!

Restar en Binario Puro

- En las restas, aparecen **acarreos negativos** (*borrows*), que se suman al sustraendo de la columna inmediatamente a la izquierda.
- Las reglas de restar son algo engorrosas cuando aparecen dichos acarreos:

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ -\,0 & -\,1 & -\,0 & -1 \\ \hline 0 & \color{red}{1}1 & 1 & 0 \end{array}$$

acarreo

Minuendo

Sustraendo

Resta

- **Inconveniente:** las reglas de la suma en binario puro son diferentes a las de la resta, lo cual implica el uso de circuitos separados para cada operación.

Restar en Binario Puro

- **Condición de desbordamiento:** Se produce cuando el minuendo es **menor** que el sustraendo, ya que en binario puro no podemos representar los números negativos.
- Ejemplo. Supongamos datos de n = 4 bits:

$(8 - 3)_{10} = (1000 - 0011)_{BP}$

$$
\begin{array}{rl}
1\,0\,0\,0 & \\
-\,0\,0\,1\,1 & \\
1\,1\,1\;\; & \text{acarreo} \\
\hline
0\,1\,0\,1 &
\end{array}
$$

Resultado = 5_{10}

$(6 - 9)_{10} = (0110 - 1001)_{BP}$

$$
\begin{array}{rl}
0\,1\,1\,0 & \\
-\,1\,0\,0\,1 & \\
1\;\;\;\; & \text{acarreo} \\
\hline
1\,1\,1\,0\,1 &
\end{array}
$$

Resultado = 13_{10}

¡¡Desbordamiento!!

MULTIPLICAR EN BINARIO PURO

Multiplicando

Multiplicador

$$
\begin{array}{rr}
 & 0010 \\
\times & 0011 \\
\hline
 & 0010 \\
 & 0010 \\
 & 0000 \\
+ & 0000 \\
\hline
 & 0000110
\end{array}
$$

Productos parciales

Producto

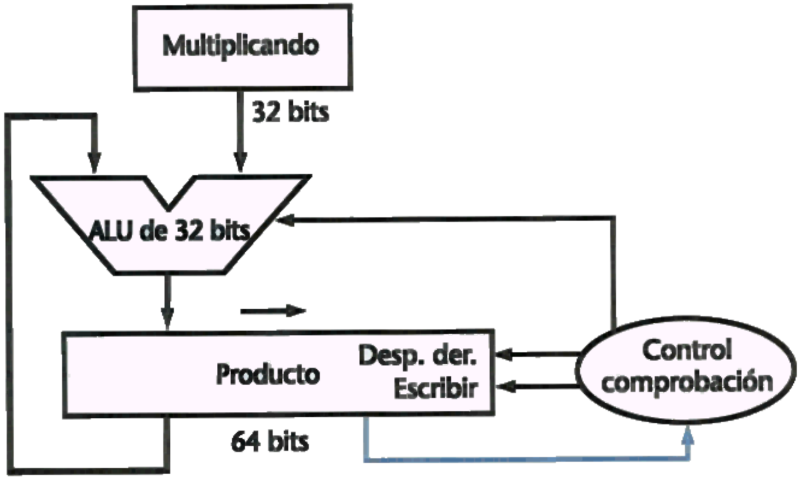
$$
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
\times\, 0 & \times\, 1 & \times\, 0 & \times\, 1 \\
\hline
0 & 0 & 0 & 1
\end{array}
$$

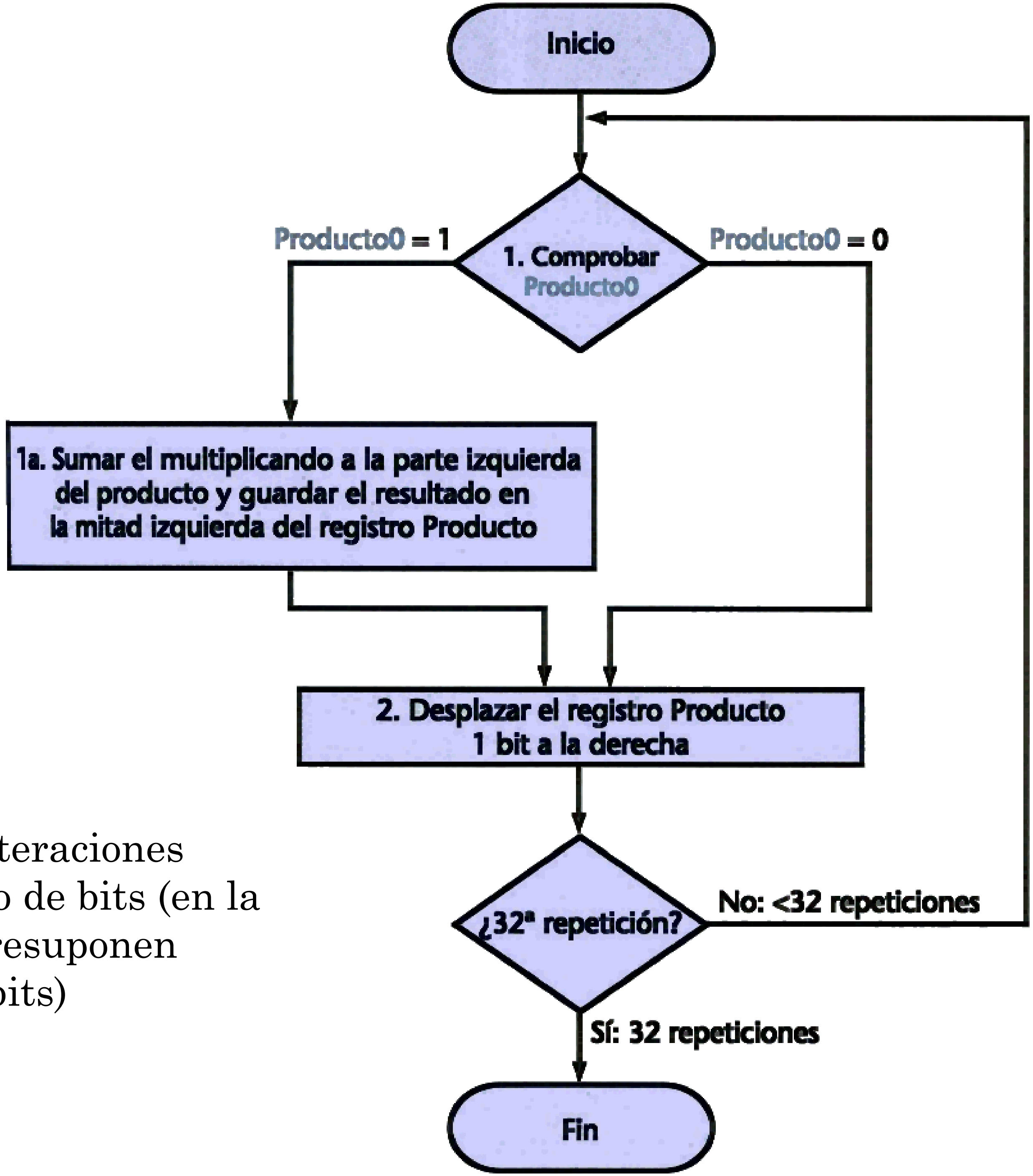
$2 \times 3 = 6$

Para datos en binario puro de n bits, el producto puede llegar a tener hasta $2n$ bits.

Multiplicar en Binario Puro

- Algoritmo de suma-desplazamiento para realizar multiplicaciones.
- Hardware necesario (suponiendo datos de 32 bits):
  - 1 registro de 32 bits (multiplicando).
  - 1 registro desplazador hacia la derecha de 64 bits (producto). Inicialmente contiene el multiplicador.
  - ALU para datos de 32 bits (permite sumar).
  - Circuitería de control.





Hay tantas iteraciones como número de bits (en la imagen se presuponen datos de 32 bits)

$0010 \times 0011 = ?$

Multiplicador

Iteración	Operación	Multiplicando	Producto
0	$\text{Producto}_{\leftarrow} = 0$ $\text{Producto}_{\rightarrow} = \text{Multiplicador}$	0010	0000 001<u>1</u>
1	1a: $\text{Producto}_0 = 1 \Rightarrow \text{Producto}_{\leftarrow} = \text{Producto}_{\leftarrow} + \text{Multiplicando}$	0010	**0010** 0011
	2: Desplazar Producto 1 bit a la derecha	0010	**0001 000<u>1</u>**
2	1a: $\text{Producto}_0 = 1 \Rightarrow \text{Producto}_{\leftarrow} = \text{Producto}_{\leftarrow} + \text{Multiplicando}$	0010	**0011** 0001
	2: Desplazar Producto 1 bit a la derecha	0010	**0001 100<u>0</u>**
3	1: $\text{Producto}_0 = 0 \Rightarrow$ Ninguna operación	0010	0001 1000
	2: Desplazar Producto 1 bit a la derecha	0010	**0000 110<u>0</u>**
4	1: $\text{Producto}_0 = 0 \Rightarrow$ Ninguna operación	0010	0000 1100
	2: Desplazar Producto 1 bit a la derecha	0010	**0000 0110**

Resultado

$\text{Producto}_0 \equiv$ Bit menos significativo del producto
$\text{Producto}_{\rightarrow} \equiv$ Mitad derecha (inferior) del producto
$\text{Producto}_{\leftarrow} \equiv$ Mitad izquierda (superior) del producto

Dividir en Binario Puro

Dividendo

Divisor

```
 111 | 10
-10  |-----
 011   11
 -10
  01
```

Cociente

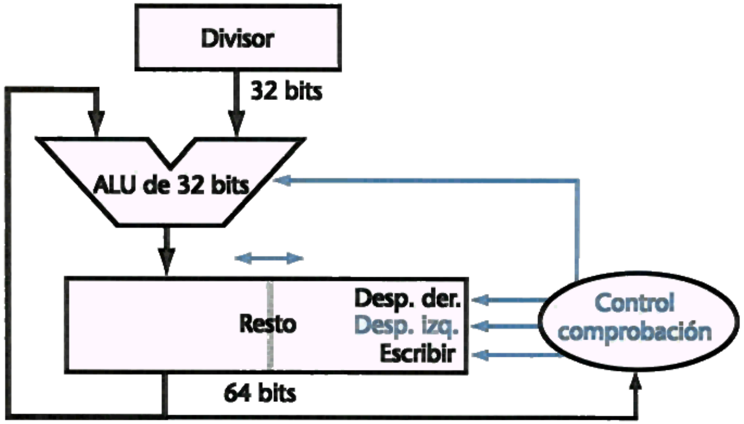
Resto

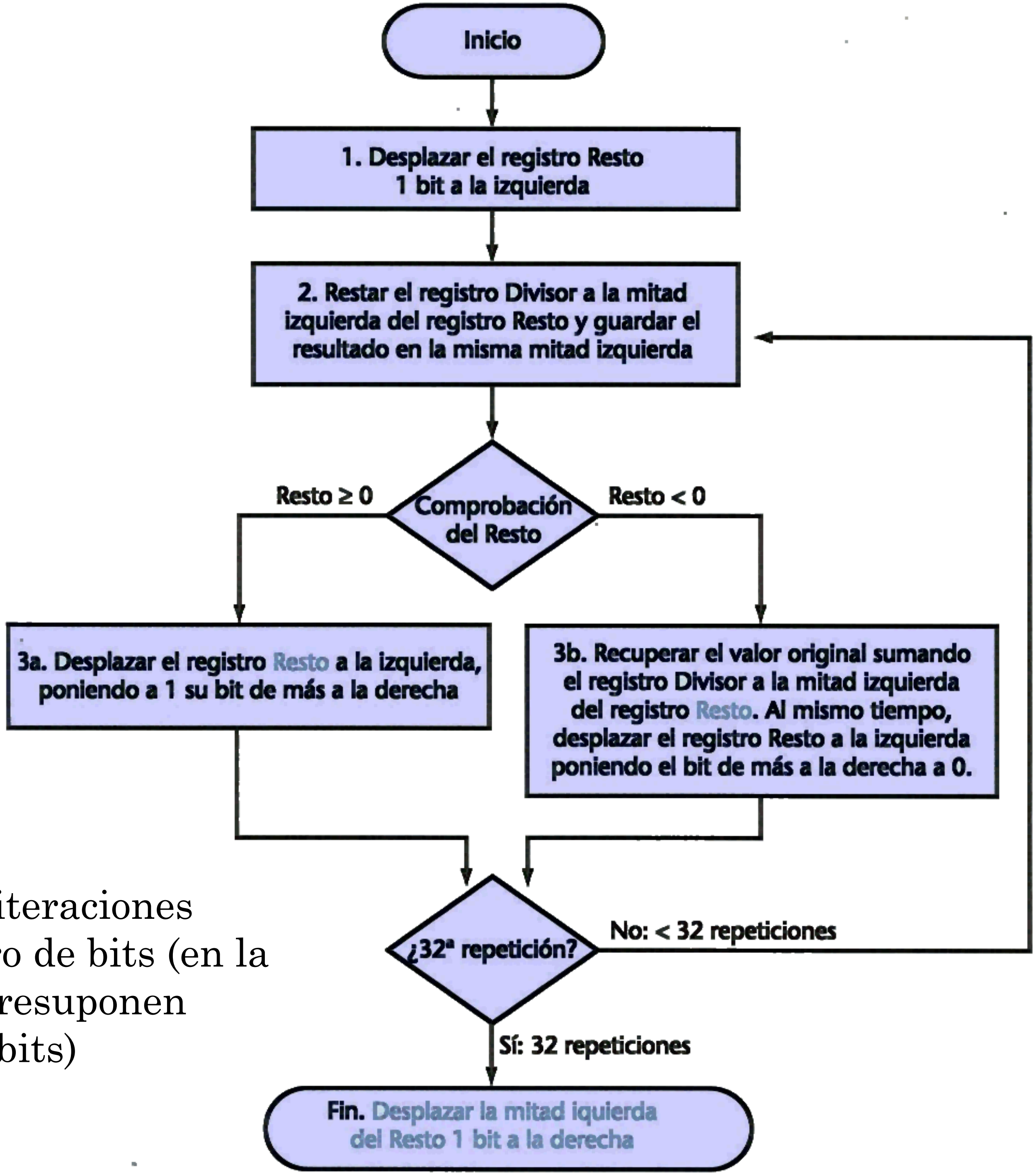
$$\begin{array}{r} 0 \\ \div\ 1 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ \div\ 1 \\ \hline 1 \end{array}$$

- No existe un bloque combinacional “divisor”.
- Hacer un circuito que resuelva la operación rápido es más complicado que en la multiplicación.
- Secuencia de sumas, restas, comparaciones y desplazamientos.
- Cuidado: la división “ofrece” la posibilidad de **dividir por 0**, lo cual produce **desbordamiento**.
- Vamos a estudiar un algoritmo de división entera (no calcularemos parte fraccionaria).

Dividir en Binario Puro

- Algoritmo de la división por restauración (calcula cociente y resto).
- Hardware necesario (suponiendo datos de 32 bits):
  - 1 registro de 32 bits (divisor).
  - 1 registro desplazador derecha-izquierda de 64 bits (resto). Inicialmente contiene el dividendo.
  - ALU para datos de 32 bits.
  - Circuitería de control.





Hay tantas iteraciones como número de bits (en la imagen se presuponen datos de 32 bits)

Universidad Rey Juan Carlos

$0111 \div 0010 = ?$

Dividendo

Iteración	Operación	Divisor	Resto
0	$\text{Resto}_{\leftarrow} = 0$ $\text{Resto}_{\rightarrow} = \text{Dividendo}$	0010	0000 0111
	Desplazar Resto 1 bit a la izquierda	0010	**0000 1110**
1	2: $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} - \text{Divisor}$	0010	**1110** 1110
	3b: $\text{Resto}_{\leftarrow} < 0$ ➔ $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} + \text{Divisor}$, desplazar Resto 1 bit a la izquierda y $\text{Resto}_0 = 0$	0010	**0001 1100**
2	2: $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} - \text{Divisor}$	0010	**1111** 1100
	3b: $\text{Resto}_{\leftarrow} < 0$ ➔ $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} + \text{Divisor}$, desplazar Resto 1 bit a la izquierda y $\text{Resto}_0 = 0$	0010	**0011 1000**
3	2: $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} - \text{Divisor}$	0010	**0001** 1000
	3a: $\text{Resto}_{\leftarrow} \geq 0$ ➔ Desplazar Resto 1 bit a la izquierda y $\text{Resto}_0 = 1$	0010	**0011 0001**
4	2: $\text{Resto}_{\leftarrow} = \text{Resto}_{\leftarrow} - \text{Divisor}$	0010	**0001** 0001
	3a: $\text{Resto}_{\leftarrow} \geq 0$ ➔ Desplazar Resto 1 bit a la izquierda y $\text{Resto}_0 = 1$	0010	**0010 0011**
5	Desplazar $\text{Resto}_{\leftarrow}$ 1 bit a la derecha	0010	**0001** 0011

Resto

Cociente

Resto_0 ≡ Bit menos significativo del resto
$\text{Resto}_{\rightarrow}$ ≡ Mitad derecha (inferior) del resto
$\text{Resto}_{\leftarrow}$ ≡ Mitad izquierda (superior) del resto

MULTIPLICACIÓN Y DIVISIÓN RÁPIDAS

- En el caso de que haya que multiplicar (dividir) un número en binario puro por una **potencia de 2**, se desplaza la coma hacia la derecha (izquierda) tantas posiciones como indique el exponente.
- Ejemplos:

$$1101001{,}111_{BP} \times 2^3 = 1101001111{,}0_{BP}$$

$$1101001{,}111_{BP} \div 2^4 = 110{,}1001111_{BP}$$

Cambio de signo en Binario Puro

- Esta operación no es posible para datos en binario puro, ya que siempre son positivos (o cero).

Extensión de signo en Binario Puro

- Consiste en expresar el mismo valor, pero con un mayor número de bits.
- Los bits adicionales, que se añaden por la izquierda, se ponen a 0.
- Ejemplo: 6 con 4 bits y con 8 bits.

$$6_{10} = 0110_{BP} = 00000110_{BP}$$

Aritmética en Complemento a 2

- Sumar y restar
- Multiplicar y dividir
- Cambio de signo
- Extensión de signo

SUMAR Y RESTAR EN COMPLEMENTO A 2

- En complemento a 2, **restar significa sumar el opuesto** (el complemento a 2 del sustraendo). Por tanto, sumar y restar utilizan las mismas reglas y se ahorra circuitería.
- Sin embargo, la gestión de los acarreos y de los desbordamientos es diferente a binario puro.
- Para datos de n bits, si se produce un bit $n + 1$, este3 bit se desprecia y **no** forma parte del resultado.
- **Detección de desbordamiento:** cuando el signo del resultado no es coherente con el de los operandos.
    - Ejemplo: si al sumar dos números positivos, da negativo o viceversa.

SUMAR Y RESTAR EN COMPLEMENTO A 2

- Supongamos datos de $n = 4$ bits:

$(6 + (-5))_{10} = (0110 + 1011)_{C2}$

$$
\begin{array}{r l}
1\,1 & \text{acarreo} \\
\hline
0\,1\,1\,0 & \\
+\,1\,0\,1\,1 & \\
\hline
\cancel{1}\,0\,0\,0\,1 &
\end{array}
$$

Resultado = $0001_{C2} = 1_{10}$

$(5 + 7)_{10} = (0101 + 0111)_{C2}$

$$
\begin{array}{r l}
1\,1\,1 & \text{acarreo} \\
\hline
0\,1\,0\,1 & \\
+\,0\,1\,1\,1 & \\
\hline
1\,1\,0\,0 &
\end{array}
$$

Resultado = $1100_{C2} = -4_{10}$

¡¡Desbordamiento!!

Sumar y restar en Complemento a 2

- Supongamos datos de $n = 4$ bits:

$(5 - 7)_{10} = (0101 - 0111)_{C2}$
$= (0101 + 1001)_{C2}$

$$\begin{array}{r} 1 \quad \text{acarreo} \\ 0\,1\,0\,1 \\ +\,1\,0\,0\,1 \\ \hline 1\,1\,1\,0 \end{array}$$

Resultado $= 1110_{C2} = -2_{10}$

$(6 - (-5))_{10} = (0110 - 1011)_{C2}$
$= (0110 + 0101)_{C2}$

$$\begin{array}{r} 1 \quad \text{acarreo} \\ 0\,1\,1\,0 \\ +\,0\,1\,0\,1 \\ \hline 1\,0\,1\,1 \end{array}$$

Resultado $= 1011_{C2} = -5_{10}$

¡¡Desbordamiento!!

SUMAR Y RESTAR EN COMPLEMENTO A 2

- **Detección de desbordamiento:** sumamos los números a y b, ambos con n bits. Los acarreos son los bits c.

$$V = c_{n-1} \oplus c_{n-2} = \left(\overline{a_{n-1} \oplus b_{n-1}}\right) \cdot \left(a_{n-1} \oplus s_{n-1}\right)$$

$$\begin{array}{cccccc}
 & c_{n\text{-}2} & c_{n\text{-}3} & \ldots & c_0 & \\
\hline
 & a_{n\text{-}1} & a_{n\text{-}2} & \ldots & a_1 & a_0 \\
+ & b_{n\text{-}1} & b_{n\text{-}2} & \ldots & b_1 & b_0 \\
\hline
c_{n\text{-}1} & s_{n\text{-}1} & s_{n\text{-}2} & \ldots & s_1 & s_0
\end{array}$$

$$\begin{array}{ccccc}
 & 1 & & & \\
\hline
 & 0 & 1 & 1 & 0 \\
+ & 0 & 1 & 0 & 1 \\
\hline
0 & 1 & 0 & 1 & 1
\end{array}$$

$$V = 0 \oplus 1 =$$
$$= \left(\overline{0 \oplus 0}\right) \cdot (0 \oplus 1) = 1$$

Ya hemos visto que el bit $c_{n\text{-}1}$ se desprecia y no forma parte del resultado. Que dicho bit sea 1 no implica necesariamente desbordamiento.

Multiplicar y dividir en Complemento a 2

- En el caso de querer realizar multiplicaciones y divisiones de números enteros con signo (datos en complemento a 2), la opción más sencilla es pasar los datos a positivo, realizar la operación y después recuperar el signo correcto.
- Existen algoritmos específicos para datos con signo (no los vamos a ver). Por ejemplo, el llamado **algoritmo de Booth**.

CAMBIO DE SIGNO EN COMPLEMENTO A 2

- El bit más significativo es el bit de signo ($0 \equiv +$, $1 \equiv -$).
- Se consigue cambiar de signo aplicando la operación complemento a 2 (ver tema anterior).
- Ejemplo para datos con 8 bits:

$$+9_{10} = 00001001_{C2}$$

$$-9_{10} = C2(00001001) = 11110111_{C2}$$

$$C1(00001001) + 1 = 11110110 + 1$$

- Recordemos que una manera sencilla de calcular el complemento a 2 es calcular primero el complemento a 1 (invertir todos los bits) y al resultado sumarle 1.

Extensión de signo en Complemento a 2

- Para expresar el mismo valor, pero con un mayor número de bits, se repite el bit de signo.
- Ejemplos de pasar datos de 4 a 8 bits:

$$6_{10} = 0110_{C2} = 00000110_{C2}$$

$$-4_{10} = 1100_{C2} = 11111100_{C2}$$

Aritmética en IEEE 754

- Sumar y restar
- Multiplicar y dividir
- Cambio de signo
- Cambio de precisión

Sumar y restar en IEEE 754

Pasos para realizar una suma o una resta en coma flotante:

1. Según el signo de los números, se decide qué operación se realiza para trabajar con las mantisas.
2. Comprobación de 0:
   - Si alguno de los dos operandos es 0 (en la suma), se devuelve como resultado el otro.
   - Si el sustraendo es 0 en una resta, se devuelve el minuendo.
   - Si es minuendo es 0 en una resta, se devuelve el sustraendo con el bit de signo invertido.
3. Alineación de mantisas: se desplazan las mantisas de ambos operandos hasta que los exponentes sean iguales.
4. Realización de la operación binaria correspondiente con las mantisas.
5. Normalizar y redondear.

EJEMPLO (EN BASE 10)

$9{,}999 \times 10^1 + 1{,}610 \times 10^{-1} = ?$

- 4 cifras decimales para la mantisa.
- 2 para el exponente.

Paso 1: Alinear mantisas

$1{,}610 \times 10^{-1} = 0{,}01610 \times 10^1 \rightarrow 0{,}016 \times 10^1$

Paso 2: Sumar mantisas

$9{,}999 + 0{,}016 = 10{,}015 \rightarrow$

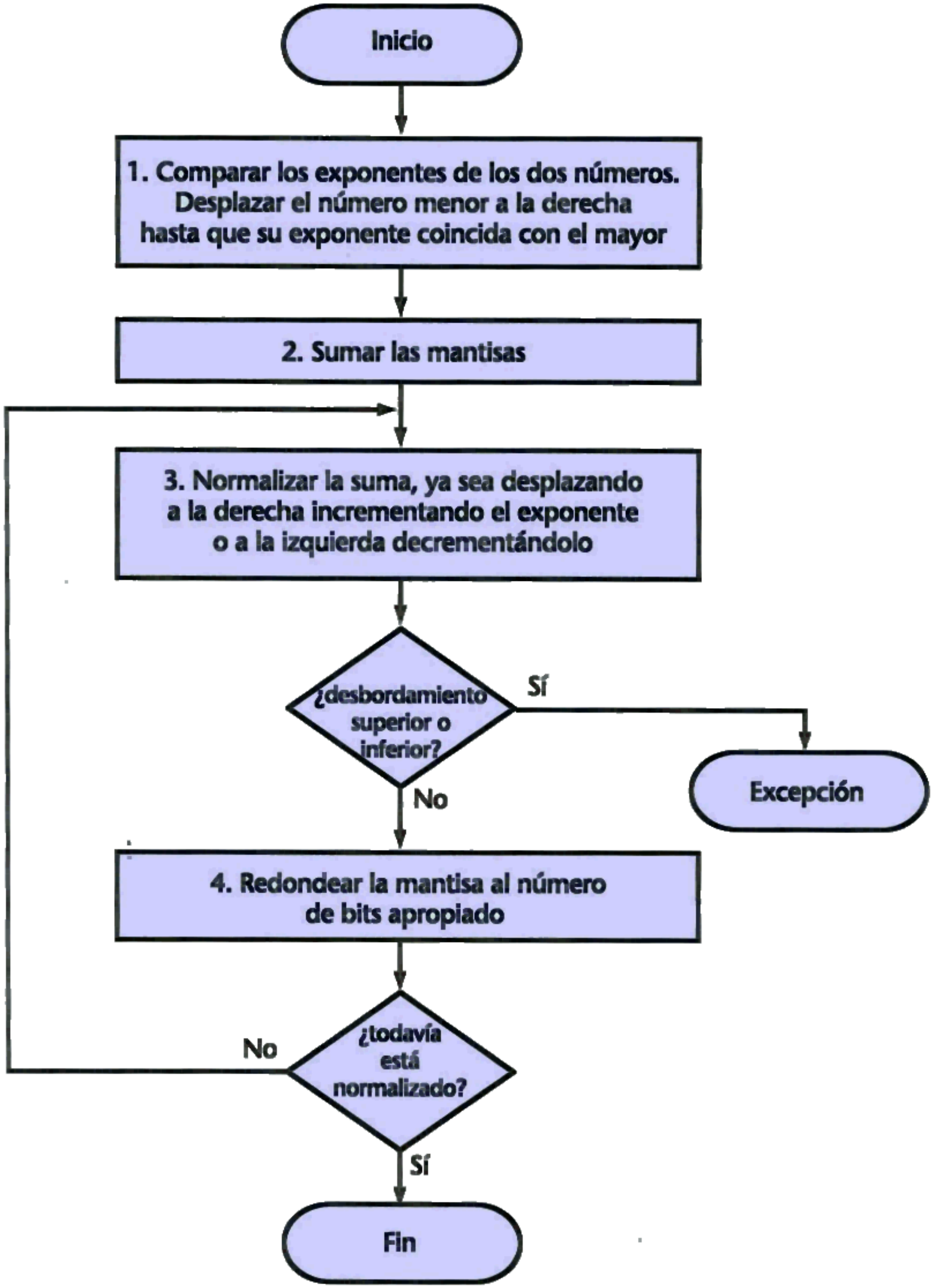
Suma $= 10{,}015 \times 10^1$

Paso 3: Normalizar resultado

$10{,}015 \times 10^1 = 1{,}0015 \times 10^2$

Paso 4: Redondear resultado

Resultado final $= 1{,}0015 \times 10^2 \approx 1{,}002 \times 10^2$



MULTIPLICAR Y DIVIDIR EN IEEE 754

- Pasos para realizar un producto en coma flotante:

1. Comprobación de 0: devolvemos 0 directamente.
2. Sumar los exponentes (restar el exceso).
3. Comprobar (sub)desbordamiento de exponente.
4. Multiplicar mantisas entre sí.
5. Normalizar y redondear.

Pasos para realizar un cociente en coma flotante:

1. Comprobar dividendo 0: se devuelve 0 directamente.
2. Comprobar si divisor es 0: se devuelve $\pm\infty$.
3. Restar los exponentes (se suma el exceso).
4. Comprobar (sub)desbordamiento de exponente.
5. Dividir mantisas entre sí.
6. Normalizar y redondear.

EJEMPLO (EN BASE 10)

$1{,}110 \times 10^{10} \times 9{,}200 \times 10^{-5} = ?$

- 4 cifras decimales para la mantisa.
- 2 para el exponente.

Paso 1: Sumar exponentes

Exponente = 10 + (–5) = 5

Paso 2: Multiplicar mantisas

$1{,}110 \times 9{,}200 = 10{,}212$

Paso 3: Normalizar resultado

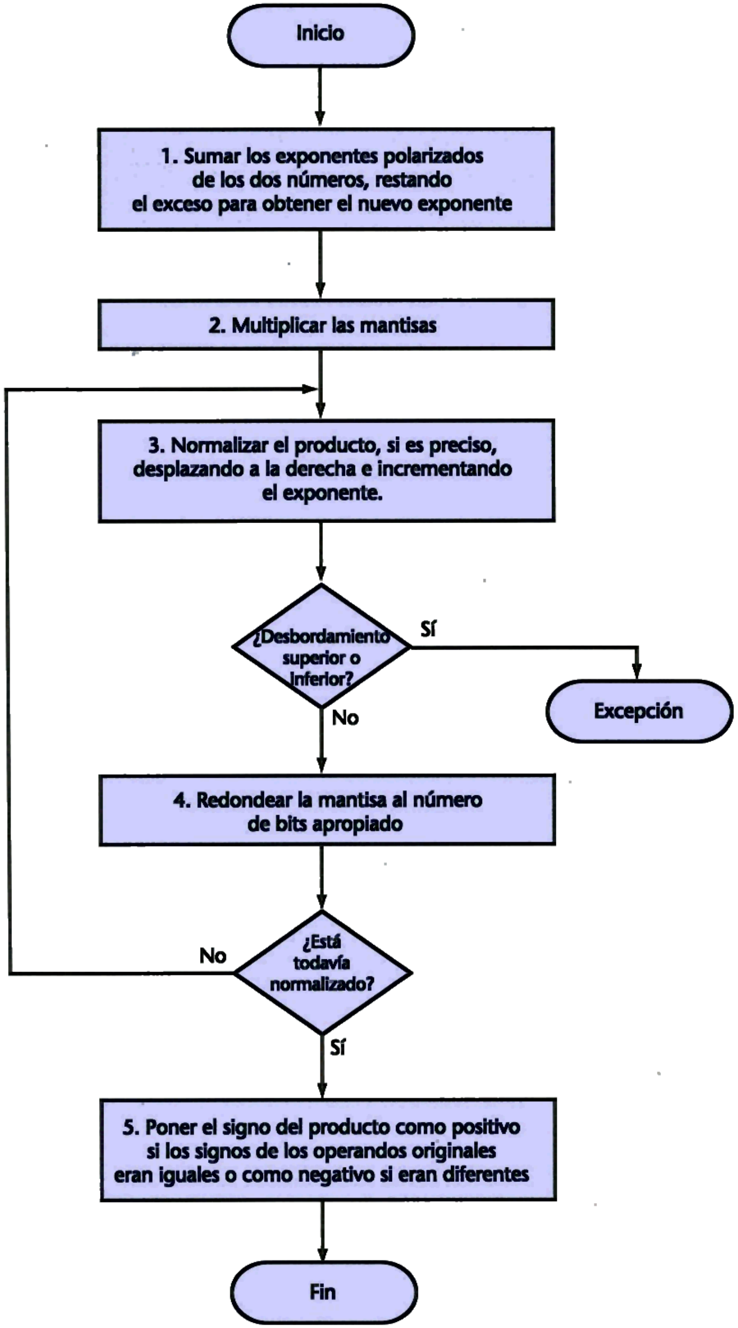
$10{,}212 \times 10^5 = 1{,}0212 \times 10^6$

Paso 4: Redondear resultado

$1{,}0212 \times 10^6 \approx 1{,}021 \times 10^6$

Paso 5: Colocar signo

Resultado final = $+1{,}021 \times 10^6$



EL ERROR "FDIV" DE PENTIUM

- El **Pentium** es un procesador de Intel que salió al mercado entre 1993 y 1994. Es el sucesor del Intel 486.
- Utiliza algoritmo estándar de división en coma flotante, que genera múltiples bits del cociente paso por paso.
- Utiliza los bits más significativos del divisor y el dividendo para estimar los siguientes 2 bits del cociente a través de una tabla de predicción.
- Si una estimación previa lleva a un resto demasiado grande, su valor se reajusta en el siguiente paso.
- En el 80486 la tabla de predicción tenía 5 elementos que Intel pensó que no se utilizarían en el Pentium. Se devolvería 0 en lugar de 2 en esas posiciones.
- ERROR: los 11 primeros bits eran siempre correctos, pero no del 12 al 52.
- Un ejemplo del fallo era el siguiente:

4195835,0 / 3145727,0 = 1,333 820 449 136 241 002 5 (correcto)

4195835,0 / 3145727,0 = 1,333 739 068 902 037 589 4 (Pentium defectuoso), subrayados los dígitos erróneos

El error "FDIV" de Pentium

- **Julio 1994:** Intel descubre el error. Coste real de **cientos de miles de dólares**. Estimación de fabricación de 3 a 5 millones de procesadores con el error.
- **Septiembre 1994:** Thomas Nicely del "Lynchburg College de Virginia" descubre el error. No hay reacción oficial de Intel. Introduce su descubrimiento en Internet.
- **7 noviembre 1994:** Portadas en los principales periódicos financieros.
- **22 noviembre 1994:** "Pequeño error" que afecta a sólo un grupo de usuarios tipo "matemáticos teóricos" (según Intel).
- **5 diciembre 1994:** El error se produce una vez cada 27000 años para usuarios habituales de hojas de cálculo (según Intel).
- **12 diciembre 1994:** IBM demuestra una probabilidad de error mucho mayor y detiene la producción de nuevos PCs basados en Pentium.
- **21 diciembre 1994:** Intel pide disculpas. Los analistas sitúan el coste de esta operación en **475 millones de dólares**.

Cambio de signo en IEEE 754

- El cambio de signo es tan fácil como invertir el bit más significativo, tanto en precisión simple como en doble.
- La mantisa y el exponente no cambian.
- Ejemplo:

$+2{,}5_{10} = 40\ 20\ 00\ 00_{16,IEEE,s} = 0\mathbf{100\ 0000\ 0}010\ 0000\ 0000\ 0000\ 0000\ 0000_{2,IEEE,s}$

$-2{,}5_{10} = C0\ 20\ 00\ 00_{16,IEEE,s} = 1\mathbf{100\ 0000\ 0}010\ 0000\ 0000\ 0000\ 0000\ 0000_{2,IEEE,s}$

Cambio de precisión en IEEE 754

- El paso de precisión simple a doble no es evidente.
- El bit de signo se conserva.
- La mantisa pasa de tener 23 bits a 52, más el bit implícito, lo cual implicaría calcular los bits adicionales.
- El exponente pasa de estar en exceso a 127 con 8 bits a exceso a 1023 con 11 bits, con lo que debe recalcularse por completo.
- Si un número puede representarse con precisión simple, también se puede representar con precisión doble (a la inversa no siempre es posible).

OPERACIONES LÓGICAS

- NOT
- AND / NAND
- OR / NOR
- XOR / XNOR

Operaciones lógicas

A	A'
0	1
1	0

NOT

Las operaciones lógicas se realizan bit a bit.

A	B	$A \cdot B$	$A + B$	$A \oplus B$	$\overline{A \cdot B}$	$\overline{A + B}$	$\overline{A \oplus B}$
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	1	0	0	0	1
		AND	OR	XOR	NAND	NOR	XNOR

OPERACIONES LÓGICAS

- Operación lógica **NOT** (monaria): se invierten todos los bits del operando.
  - Ejemplo: n = 4 bits, A = 0110.

$$\begin{array}{r} A = 0\ \ 1\ \ 1\ \ 0 \\ \hline \text{NOT } A = 1\ \ 0\ \ 0\ \ 1 \end{array}$$

- ➲ Operaciones binarias: se realizan bit a bit con dos operandos.
  - Ejemplo: n = 4 bits, A = 0110, B = 1100.

$$\begin{array}{r} A = 0\ \ 1\ \ 1\ \ 0 \\ B = 1\ \ 1\ \ 0\ \ 0 \\ \hline A \text{ OR } B = 1\ \ 1\ \ 1\ \ 0 \end{array} \qquad \begin{array}{r} A = 0\ \ 1\ \ 1\ \ 0 \\ B = 1\ \ 1\ \ 0\ \ 0 \\ \hline A \text{ AND } B = 0\ \ 1\ \ 0\ \ 0 \end{array} \qquad \begin{array}{r} A = 0\ \ 1\ \ 1\ \ 0 \\ B = 1\ \ 1\ \ 0\ \ 0 \\ \hline A \text{ XOR } B = 1\ \ 0\ \ 1\ \ 0 \end{array}$$

OPERACIONES LÓGICAS

- **AND:** Sirve como máscara para **anular** determinados bits y dejar los demás inalterados (0 = se anula, 1 = se mantiene).
- **OR:** Sirve como máscara para **activar** determinados bits y dejar los demás inalterados (0 = se mantiene, 1 = se activa).
- Ejemplo: dato = 11010010, máscara = 00011111

$$\begin{array}{r} 11010010 \\ \cdot\, 00011111 \\ \hline 00010010 \end{array} \qquad \begin{array}{r} 11010010 \\ +\, 00011111 \\ \hline 11011111 \end{array}$$

OPERACIONES LÓGICAS

- **NOR:** Aplicados a todos los bits de un dato, sirve para detectar si se anulan todos a la vez (detector de cero).
- Ejemplo: dato = 11000010

$$\overline{1+1+0+0+0+0+1+0} = 0$$

- Ejemplo: dato = 00000000

$$\overline{0+0+0+0+0+0+0+0} = 1$$

OPERACIONES LÓGICAS

- **NOT:** Invierte todos los bits.
- **XOR:** Detecta la desigualdad de dos bits.
  - También sirve para invertir los bits en función de una máscara M (0 = se mantiene, 1 = se invierte)

M	A	$M \oplus A$
0	0	$0 = A$
0	1	$1 = A$
1	0	$1 = A'$
1	1	$0 = A'$

- Ejemplo: dato = 11010010, máscara = 00011111

$$\frac{(11010010)'}{00101101}$$

$$\begin{array}{r} 11010010 \\ \oplus\ 00011111 \\ \hline 110\textcolor{red}{01101} \end{array}$$

Operaciones de desplazamiento de bits

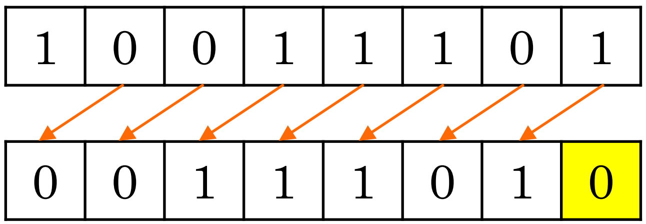
- Desplazamiento hacia la derecha / izquierda
- Rotaciones

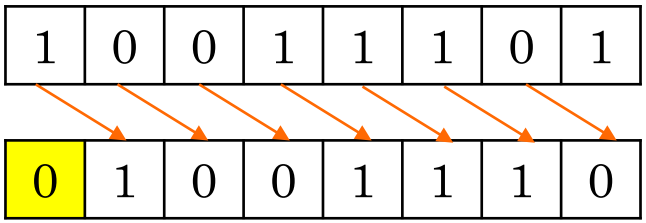
OPERACIONES DE DESPLAZAMIENTO DE BITS

- En estas operaciones, los bits de un dato se cambian de posición según un valor entero de desplazamiento *s* (≥ 1), lo cual produce **desplazamientos hacia la izquierda o hacia la derecha**.
- Como el dato tiene tamaño fijo, se pierden *s* bits por el extremo hacia el que se produce el desplazamiento.
- En el hueco surgido por el extremo contrario, entran *s* nuevos bits (pueden ser 0 o 1, según el tipo de desplazamiento).
- Equivalente a multiplicar o dividir por potencias de 2 (operación más rápida que con los algoritmos de suma-desplazamiento o restauración).

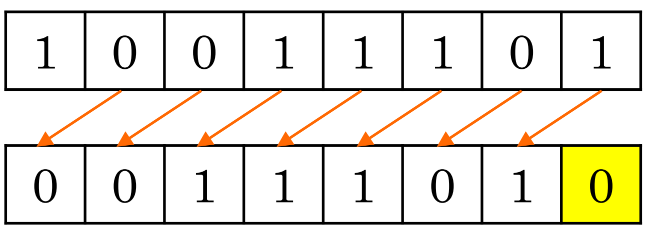
OPERACIONES DE DESPLAZAMIENTO DE BITS

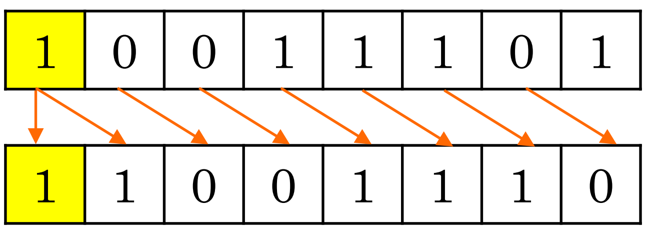
- **Lógico:** Los bits entrantes siempre son 0 (datos en binario puro).





- **Aritmético:** Se repite el bit de signo al desplazar a la derecha (datos en complemento a 2).





- **Rotación:** los bits que salen por un extremo entran por el otro.