

Sistemas Informáticos

Unidad 02.

Representación de la información



Autores: Sergi García, Alfredo Oltra

Actualizado Septiembre 2025



Licencia



Reconocimiento - No comercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se ha de hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Importante**

 **Atención**

 **Interesante**

ÍNDICE

1. Introducción	4
1.1. Dato, pieza de información e información	4
1.2. Representación interna de los datos	4
1.3. El verdadero “conocimiento” de un ordenador	5
2. Sistemas de numeración	5
2.1. Descomposición de números en un sistema de numeración	5
2.2. Generalización a cualquier sistema de base B	5
2.3. Código binario	6
2.4. Máximo valor representable con n bits	9
2.5. Operaciones con números binarios	9
2.6. Representación de números negativos en binario	11
2.6.4. Resumen práctico	14
2.7. Números reales en los ordenadores	15
3. Álgebra booleana	17
3.1. Operación NOT (negación)	17
3.2. Operación AND (conjunción lógica)	17
3.3. Operación OR (disyunción lógica)	18
3.4. Operación XOR (disyunción exclusiva)	18
3.5. Propiedades del álgebra booleana	19
4. Sistemas octal y hexadecimal	19
4.1. Sistema octal	19

4.2. Conversión de binario a octal	20
4.3. Conversión de octal a binario	20
4.4. Sistema hexadecimal	20
4.5. Conversión de binario a hexadecimal	21
5. Representación alfanumérica	22
5.1. Datos numéricos y alfanuméricos	22
5.2. Representación interna	23
6. Sistema de unidades	23
7. Bibliografía	24

UNIDAD 02. REPRESENTACIÓN DE LA INFORMACIÓN

1. INTRODUCCIÓN

1.1. Dato, pieza de información e información

Los ordenadores (o, de manera más precisa, los sistemas de información) son máquinas diseñadas para **procesar información**, es decir, obtener resultados a partir de la aplicación de operaciones sobre un conjunto de datos.

Pero antes de continuar, conviene hacerse unas preguntas clave:

- ¿Qué entendemos por **información**?
- ¿Qué es exactamente una **pieza de información**?
- ¿Y qué significa **operación** en este contexto?

Veámoslo con un ejemplo sencillo:

- **Dato o pieza de información:** una representación formal y objetiva de un concepto. Por ejemplo: “30”.
- **Información:** el resultado de interpretar ese dato en un contexto. Ejemplo: “Hace calor”.
- **Operación:** la regla o el criterio que transforma el dato en información. Ejemplo: “Si la temperatura es superior a 23 °C, entonces hace calor”.

👉 Conclusión: **el dato por sí solo carece de sentido**. Solo cuando lo interpretamos aplicando una regla, se convierte en información útil.

1.2 Representación interna de los datos

Si queremos que un ordenador trabaje con información, necesita almacenar y manipular **datos** y también las **operaciones** que permiten interpretarlos. Para lograrlo, utiliza un sistema universal: **el código binario**.

⚠ **Atención:** todo tipo de datos, ya sean números, letras, imágenes o sonidos, se representan internamente con este sistema.

El sistema binario se basa en el uso exclusivo de dos dígitos: **0 y 1**, a diferencia del sistema decimal que emplea diez (0–9).

Esto se debe a que los ordenadores no “piensan” en números como nosotros, sino que trabajan con señales eléctricas que solo pueden asumir **dos estados posibles**:

- **0** → cuando la tensión o potencial eléctrico está próxima a 0 voltios.
- **1** → cuando la tensión supera un umbral, normalmente alrededor de **3 o 5 voltios**.

💬 **Interesante:** en términos eléctricos, el “potencial” puede entenderse como la fuerza con la que circula la corriente a través de un conductor.

💬 **Interesante:** aunque los valores concretos varían según la tecnología, la idea es siempre la misma: un estado de “bajo” (0) y un estado de “alto” (1).

1.3. El verdadero “conocimiento” de un ordenador

Todos los componentes de un ordenador, desde la memoria hasta el procesador, utilizan este sistema binario para representar y procesar la información.

Por eso, puede afirmarse que en realidad **los ordenadores no “saben” nada**:

- Solo “entienden” 0 y 1.
- Solo saben aplicar operaciones básicas con ellos (sumar, restar, comparar...).
- La diferencia está en que lo hacen a una **velocidad extremadamente alta**, lo que les permite resolver problemas complejos y simular procesos del mundo real.

2. SISTEMAS DE NUMERACIÓN

Un **sistema de numeración** es un conjunto ordenado de símbolos que se utilizan para representar cantidades. El número de símbolos que lo forman recibe el nombre de **base del sistema**.

En la vida cotidiana estamos acostumbrados al **sistema decimal** (base 10), cuyos símbolos son: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.

2.1. Descomposición de números en un sistema de numeración

Cualquier número, en cualquier sistema, puede descomponerse en **dígitos**. Por ejemplo:

- El número **128** puede descomponerse en los dígitos: 1, 2 y 8.
- El número **34,76** en los dígitos: 3, 4, 7 y 6.

A partir de estos dígitos, su **posición** y la **base** del sistema, es posible reconstruir el número original:

$$128 = 1 * 10^2 + 2 * 10^1 + 8 * 10^0$$

$$34,76 = 3 * 10^1 + 4 * 10^0 + 7 * 10^{-1} + 6 * 10^{-2}$$

Un número decimal puede representarse como una suma de **potencias de base 10**.


2.2. Generalización a cualquier sistema de base B

De forma general, un número **N** expresado en un sistema de base **B** puede escribirse como:

$$N = a_{n-1} a_{n-2} a_{n-3} \dots a_1 a_0 , a_{-1} a_{-2} \dots a_{-p+1} a_{-p}$$


Donde:

- **N**: número que se quiere representar
- **a**: cada uno de los dígitos del sistema, que siempre son enteros comprendidos entre 0 y (B - 1).
- Los dígitos **antes de la coma** representan la **parte entera**.
- Los dígitos **después de la coma** representan la **parte fraccionaria**.

 **Interesante:** en la cultura anglosajona, el separador entre parte entera y fraccionaria no es la coma (,) sino el **punto decimal** (.).


2.3. Código binario

El **código binario** es un sistema de numeración cuya base es **2**. Sus símbolos son únicamente: **0 y 1**.

 **Importante:** cada dígito de un número binario recibe el nombre de **bit** (*binary digit*).

El bit es la **unidad mínima de información**: lo más pequeño que puede representarse en un ordenador.

- $101_{(10)}$ significa 101 en base decimal.
- $101_{(2)}$ significa 101 en base binaria (que en decimal equivale a 5).

 **Interesante:** para evitar confusiones entre sistemas de numeración, es habitual indicar la base con un subíndice a la derecha:

2.3.1. Conversión de decimal a otra base

En general, para convertir un número decimal a otro sistema de numeración de base **B**, se sigue este procedimiento:

1. Parte entera:

- Se realizan **divisiones sucesivas** del número entre la base.
- En cada paso se guarda el **resto** de la división.
- El proceso termina cuando el cociente es cero.
- El número convertido se obtiene tomando los restos en **orden inverso** (del último al primero).

2. Parte fraccionaria:

- Se multiplica la fracción sucesivamente por la base.
- En cada paso se toma la **parte entera** del resultado.
- El proceso se repite hasta obtener el nivel de precisión deseado o hasta que el resultado sea exacto.
- En este caso, los dígitos se leen en el **orden directo** (del primero al último obtenido).

Ejemplo 1: conversión de número entero ($45_{10} \rightarrow$ binario)

1. $45 \div 2 = 22$, resto 1
2. $22 \div 2 = 11$, resto 0
3. $11 \div 2 = 5$, resto 1
4. $5 \div 2 = 2$, resto 1
5. $2 \div 2 = 1$, resto 0
6. $1 \div 2 = 0$, resto 1

Leyendo los restos en orden inverso: $45_{10} = 101101_2$.

Ejemplo 2: conversión de número fraccionario ($6,25_{10} \rightarrow$ binario)

Parte entera (6):

1. $6 \div 2 = 3$, resto 0
2. $3 \div 2 = 1$, resto 1

3. $1 \div 2 = 0$, resto 1

Parte entera en binario: **110**.

Parte fraccionaria (0,25):

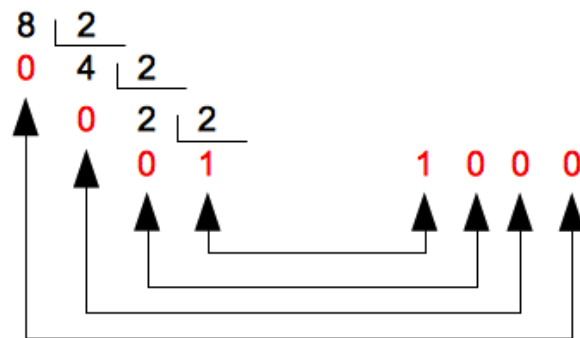
1. $0,25 \times 2 = 0,50 \rightarrow$ entero 0
2. $0,50 \times 2 = 1,00 \rightarrow$ entero 1

Parte fraccionaria en binario: **01**.

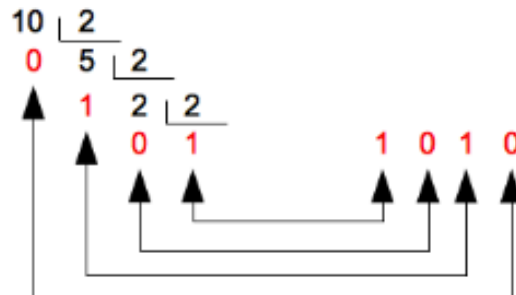
Resultado final: $6,25_{10} = 110,01_2$.

Vamos a verlo de forma gráfica

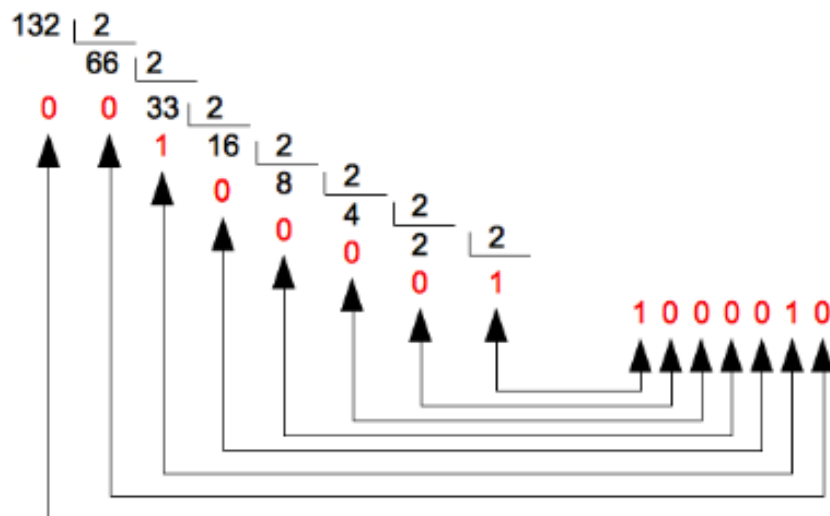
$8_{(10)} \Rightarrow ?_{(2)}$



$10_{(10)} \Rightarrow ?_{(2)}$



$132_{(10)} \Rightarrow ?_{(2)}$



Importante:

- El **bit más a la izquierda** se denomina **bit más significativo (MSB, Most Significant Bit)**.
- El **bit más a la derecha** se denomina **bit menos significativo (LSB, Least Significant Bit)**.

2.3.2. Conversión de un número binario a decimal

El proceso de conversión de binario a decimal es muy sencillo.

Como ya vimos, un número en cualquier sistema puede representarse como una **suma de potencias de la base**.

En general, para un número expresado en una base **B**, la fórmula es:

$$N = a_{n-1}B^{n-1} + a_{n-2}B^{n-2} + \dots + a_1B^1 + a_0B^0 + a_{-1}B^{-1} + \dots + a_{-p}B^{-p} = \sum a_i B^i$$

donde:

- **N** es el número representado.
- **B** es la base del sistema de numeración.
- **a_i** son los dígitos del número (de 0 hasta B – 1).

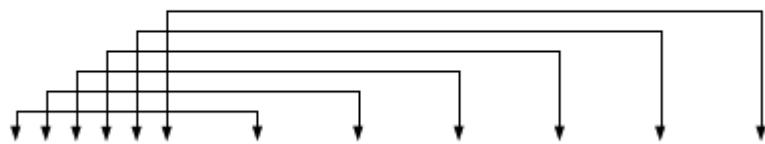
2.3.3. Pasos para convertir binario a decimal

1. Escribir las cifras del número binario.
2. Multiplicar cada cifra por **2 elevado al exponente correspondiente**.
 - El exponente empieza en 0 para el último dígito de la parte entera (a la derecha).
 - Hacia la izquierda aumenta de 1 en 1.
 - Hacia la derecha (parte fraccionaria) disminuye de 1 en 1.
3. Colocar un **signo +** entre cada producto.
4. Realizar la suma total.

Ejemplo 1: número entero

Convertir **101001₂** a decimal:

101001₂ => ?₍₁₀₎



$$101001 \Rightarrow 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 41$$

Por tanto: **101001₂ = 41₁₀**.

Ejemplo 2: número fraccionario

Convertir **10,01₂** a decimal:

$10,01_{(2)} \Rightarrow ?_{(10)}$

$$10,01 \Rightarrow 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2,25$$

Por tanto: $10,01_2 = 2,25_{10}$.

⚠ Atención: como veremos más adelante, B puede ser cualquier sistema de numeración.

2.4. Máximo valor representable con n bits

Un número binario de **n bits** puede representar exactamente **2^n valores diferentes**.

Ejemplo:

- Con **4 bits** $\rightarrow 2^4 = 16$ valores.
- El rango es de **0** (0000_2) a **15** (1111_2).

2.5. Operaciones con números binarios

2.5.1. Suma

Reglas básicas:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $0 + 1 = 1$
- $1 + 1 = 0$ (con **acarreo** 1 a la siguiente posición)

Ejemplo:

$$\begin{array}{r} \textcolor{red}{11} \\ 10011010 \\ + 01001100 \\ \hline 11100110 \end{array}$$

$$\begin{array}{r} \textcolor{red}{111111} \\ 1011 \\ + 111101 \\ \hline 1001000 \end{array}$$

⚠ Atención: si el resultado excede el número de bits disponibles, se produce un **overflow (desbordamiento)**. Por ejemplo, en un sistema de 8 bits (máximo 255_{10}):

Por ejemplo, si queremos sumar $10000000_{(2)}$ ($128_{(10)}$) más $10000000_{(2)}$ ($128_{(10)}$), tenemos un problema porque el resultado es $100000000_{(2)}$ ($256_{(10)}$) que es mayor que 255, ocurriendo desbordamiento al no caber en 8 bits.


2.5.2. Resta

Reglas básicas:

- $0 - 0 = 0$
- $1 - 0 = 1$
- $0 - 1 = 1$ (con préstamo de 1 a la siguiente columna)
- $1 - 1 = 0$

Ejemplo:

$$\begin{array}{r} 101101 \\ \textcolor{red}{1} \\ - 10101 \\ \hline 011000 \end{array} \qquad \begin{array}{r} 11101 \\ \textcolor{red}{11} \\ - 00111 \\ \hline 10110 \end{array}$$

 **Atención:** el préstamo no se añade al minuendo, sino al **sustraendo**.

2.5.3. Multiplicación

Reglas básicas:

- $0 \times 0 = 0$
- $1 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 1 = 1$


Ejemplo:

$$1012 \times 112101_2 * 11_2 1012 \times 112$$

Se resuelve igual que en decimal, pero aplicando las reglas binarias:

$$\begin{array}{r} 11101 \\ * 1001 \\ \hline 11101 \\ + 00000 \\ + 00000 \\ + 11101 \\ \hline \end{array}$$

Resultado: $11101 \times 1001 = 100000101_2 = 261_{10}$.

 **Atención:** si en una columna aparecen más de dos unos, se van sumando en grupos de dos y generando acarreo.


2.5.4. División

Reglas básicas:

- $0 \div 1 = 0$
- $1 \div 1 = 1$
- $1 \div 0 = \text{no definido (error)}$.
- $0 \div 0 = \text{indeterminado}$.

En binario:

$$\begin{array}{r}
 101010 \quad | \quad 110 \\
 -110 \\
 \hline
 1001 \\
 1 \\
 -1110 \\
 \hline
 00110 \\
 110 \\
 \hline
 000
 \end{array}$$

 **Atención:** en la división binaria, se comienza comparando divisor y dividendo con el mismo número de cifras, y se va desplazando hasta poder “contener” el divisor dentro del dividendo.

2.6. Representación de números negativos en binario

Cuando necesitamos representar números negativos en binario, existen varias formas de hacerlo. Las más importantes son **cuatro**:

1. **Signo y magnitud.**
2. **Complemento a uno.**
3. **Complemento a dos.**
4. **Exceso-K.**

 La elección del método es una **convención** entre quién genera el número y quién lo interpreta. Si no existe acuerdo, el valor real podría malinterpretarse.

2.6.1. Signo y magnitud

Es el método más sencillo de comprender.

- El **bit más significativo (MSB)** indica el signo:
 - 0 → positivo
 - 1 → negativo
- Los demás bits representan el valor absoluto del número.

Ejemplo con 4 bits:

Decimal	Binario puro	Positivo (signo-magnitud)	Negativo (signo-magnitud)
+5	101	0101	1101 (-5)

👉 Con 4 bits:

- En binario normal, el rango es [0, 15].
- En signo-magnitud, se dedica un bit al signo, por lo que el rango es de -7 a +7.

⚠ **Atención:** existen dos formas de representar el 0 → 0000 (+0) y 1000 (-0). Esto complica operaciones aritméticas.

2.6.2. Complemento a uno

Este método también usa el **primer bit como signo**, pero los negativos se obtienen **invirtiendo todos los bits** del número positivo (cambiando 0 → 1 y 1 → 0).

Ejemplo con 4 bits:

Decimal	Binario puro	Positivo (comp. a 1)	Negativo (comp. a 1)
+5	101	0101	1010 (-5)

Ejemplo con 8 bits:

- +5 → 00000101
- -5 → invertir bits → 11111010

⚠ **Atención:** también existen dos ceros posibles (0000 y 1111).

2.6.3. Complemento a dos

Es el sistema más utilizado en la práctica, porque **simplifica al máximo las operaciones aritméticas**.

El proceso es:

1. Tomar el número positivo en binario.
2. Aplicar el complemento a uno (invertir bits).
3. Sumar 1 al resultado.

Ejemplo (-5 en 8 bits):

1. +5 → 00000101

2. Complemento a 1 \rightarrow 11111010

3. +1 \rightarrow 11111011

Por tanto:

- +5 = 00000101
- -5 = 11111011

Cómo recuperar el valor decimal en complemento a dos

Ejemplo: 11111011_2

1. Detectamos que es negativo (MSB = 1).
2. Aplicamos complemento a 1 \rightarrow 00000100.
3. Sumamos 1 \rightarrow 00000101.
4. Resultado = -5.

Ventaja clave

En complemento a dos, la **resta** puede realizarse como una **suma con el negativo**.

Ejemplo:

Por ejemplo, restar $45_{10} - 21_{10}$, se suma 45_{10} al número - 21_{10} y da 24_{10}

En binario (6 bits):

- $45 = 101101$
- $21 = 010101$

Obtenemos -21:

- $010101 \rightarrow$ complemento a 1 = $101010 \rightarrow +1 = 101011$.


Ahora sumamos:

$$101101_2 + 101011_2 = 011000 = 24_{10}$$

2.6.4. Exceso-K

Este método reparte el rango entre negativos y positivos desplazando el **cero al centro del rango**.

- Se define un valor de **exceso K**: $K = 2^{n-1}$.
- El rango resultante será: $[-K, K-1]$.

 **Interesante:** hay otra versión de este método con $K = (2^{n-1} - 1)$.

Ejemplo con 3 bits ($n = 3$):

- Se pueden representar $2^3 = 8$ valores.
- $K = 2^2 = 4$.
- El rango será $[-4, 3]$.

Decimal	Binario
-4	000
-3	001
-2	010
-1	011
0	100
1	101
2	110
3	111

Ejemplo con 8 bits ($n = 8$):

- $K = 2^7 = 128$.
- Rango = $[-128, 127]$.

Conversión:

- $11001100_2 = 204_{10} \rightarrow 204 - 128 = 76$
- $00111100_2 = 60_{10} \rightarrow 60 - 128 = -68$

2.6.4. Resumen práctico

- **Signo-magnitud:** simple, pero con dos ceros y operaciones complicadas.
- **Complemento a uno:** mejora, pero sigue teniendo dos ceros.
- **Complemento a dos:** el estándar actual (suma y resta unificadas).
- **Exceso-K:** muy usado en hardware (representación de exponentes en punto flotante, como en IEEE 754).

2.7. Números reales en los ordenadores

Cuando escribimos un número real en papel, utilizamos una **coma decimal** (o punto decimal, según la cultura) para separar la **parte entera** de la **parte fraccionaria**. En un ordenador, el espacio disponible para representarlo se divide en **dos campos**: uno para la parte entera y otro para la parte fraccionaria. Existen dos métodos principales para fijar el tamaño de estos campos y, por tanto, la posición de la coma:

- **Coma fijo.**
- **Coma flotante.**

2.7.1. Representación en coma fijo

En esta notación, se reserva un número **fijo de bits** para la parte entera y otro número fijo para la parte fraccionaria. Es decir, la posición de la coma no varía.

✓ **Ventaja:** Las operaciones básicas (suma, resta, multiplicación) se realizan igual que con números enteros.

✗ **Inconveniente:** No aprovecha bien la capacidad de representación. El rango de valores es muy limitado.

📌 Ejemplo:

En un ordenador de **8 bits**, podemos usar:

- 5 bits para la parte entera
- 3 bits para la parte fraccionaria

Formato: b7 b6 b5 b4 b3 , b2 b1 b0

- Máximo representable: $01111,111_2 = 15,875_{10}$
- Mínimo positivo: $00000,001_2 = 0,125_{10}$

Si en lugar de fijar la coma, esta fuera “flotante”, podríamos representar:

- Desde $011111111_2 = 255$
- Hasta $0,0000001_2 \approx 0,0078$

2.7.2. Representación en coma flotante

El rango de valores en **punto fijo** es insuficiente para aplicaciones científicas, donde aparecen números **muy grandes** o **muy pequeños**. Por eso se usa el **punto flotante**, inspirado en la **notación científica**:

📌 Ejemplo en decimal:

$$0,00000025 = 2,5 \cdot 10^{-7}$$

En general, cualquier número real puede expresarse como:

$$N = M * BE \text{ or } N = (M;B;E)$$

donde:

- **M:** mantisa (los dígitos significativos del número)
- **B:** base (por lo general 2 en informática)
- **E:** exponente

📌 Ejemplo en binario:

Por ejemplo en decimal (B=10) $259,75_{(10)} = 0,25975 \cdot 10^3$ o (0,25975;10;3) o, en código binario

$$259,75_{(10)} \rightarrow 100000011,11_{(2)} \rightarrow 0,10000001111 \cdot 2^9_{(2)} \rightarrow 0,10000001111 \cdot 2^{1001_{(2)}} \rightarrow (0,10000001111;1001)$$

2.7.3. Normalización

Un mismo número puede expresarse de infinitas formas en notación científica.

Ejemplo:

2,5 representado en una forma normalizada es $0,25 \cdot 10^1$

$$(0,000011101; 2; 0111) \rightarrow (0,11101; 0011) \rightarrow \text{Exponente en exceso-}k \text{ } (0,11101; 1011)$$

Para **evitar ambigüedad**, se elige la **forma normalizada**, que asegura **máxima precisión**.

👉 En binario, esto significa que el **punto binario** se coloca inmediatamente a la izquierda del primer dígito significativo (el primer 1).

Más ejemplos:

$$(100,11110; 2; 0010) \rightarrow (0,10011110; 2; 0101) \xrightarrow{\text{Exponente exceso-}k} (0,10011110; 1101)$$

$$(101,001; 2; 0100) \rightarrow (0,1010010; 2; 0111) \xrightarrow{\text{Exponente exceso-}k} (0,10011110; 1111)$$

2.7.4. Estándar IEEE 754

El formato más usado en informática para representar números en **punto flotante binario** es el estándar **IEEE 754**, creado por el **Institute of Electrical and Electronics Engineers**.

Además de números normales, este formato define:

- $+\infty$ y $-\infty$
- NaN (Not a Number) para resultados indefinidos

📌 Tres formatos principales:

1. Half precision (precisión reducida) – 16 bits

- 1 bit: signo
- 5 bits: exponente
- 10 bits: mantisa



2. Single precision (precisión simple) – 32 bits

- 1 bit: signo
- 8 bits: exponente
- 23 bits: mantisa




3. Double precision (precisión doble) – 64 bits

- 1 bit: signo
- 11 bits: exponente
- 52 bits: mantisa




Detalles importantes:

- En **mantisa normalizada**, el primer dígito significativo siempre es **1** en binario.
 -  Este **1 no se almacena**, solo los bits a la derecha de la coma. Esto ahorra espacio y da **un bit extra de precisión**.
- El **exponente** se guarda en **exceso-K**, donde $K = (2^{n-1} - 1)$ y donde n = número de bits del exponente.
Ejemplo:
 - Con 8 bits → $K=127$.
 - Con 11 bits → $K=1023$.


3. ÁLGEBRA BOOLEANA

Además de las operaciones matemáticas habituales (+, −, ×, ÷) que se pueden aplicar a números binarios, también existen operaciones **booleanas o lógicas**: **AND**, **OR**, **XOR**, **NOT**, entre otras.

 **Importante:** para comprender mejor estas operaciones, es útil pensar en el **1 como “verdadero” (true)** y el **0 como “falso” (false)**.

3.1. Operación NOT (negación)

- Se representa de varias formas: **NOT**, \neg , $!$.
- Su función es invertir el valor lógico:
 - NOT 0 = 1
 - NOT 1 = 0

 Es decir, cambia un valor verdadero en falso y viceversa.

3.2. Operación AND (conjunción lógica)

- Se representa como AND, Y, \wedge , \cdot .
- Solo devuelve **1 (verdadero)** si los dos operandos son **1**.
- Tabla de verdad:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

👉 Equivale a la **multiplicación** en aritmética ($1 \times 1 = 1$, todo lo demás da 0).

3.3. Operación OR (disyunción lógica)

- Se representa como OR, O, v, +.
- Devuelve **1 (verdadero)** si **al menos uno de los operandos es 1**.
- Tabla de verdad:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

👉 Es suficiente con que uno de los valores sea verdadero para que el resultado sea verdadero.

3.4. Operación XOR (disyunción exclusiva)

- Se representa como XOR, \oplus .
- Devuelve **1 (verdadero)** únicamente cuando **los dos operandos son distintos**.
- Tabla de verdad:

A	B	A XOR B
0	0	0

0	1	1
1	0	1
1	1	0

👉 Es la operación del “o bien... o bien... pero no ambos”.

3.5. Propiedades del álgebra booleana

Estas operaciones cumplen reglas similares a la aritmética, pero adaptadas al sistema binario. Algunas propiedades importantes:

- **Conmutatividad:**
 $A \text{ AND } B = B \text{ AND } A$
 $A \text{ OR } B = B \text{ OR } A$
- **Asociatividad:**
 $(A \text{ AND } B) \text{ AND } C = A \text{ AND } (B \text{ AND } C)$
 $(A \text{ OR } B) \text{ OR } C = A \text{ OR } (B \text{ OR } C)$
- **Distributividad:**
 $A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$
- **Identidad:**
 $A \text{ AND } 1 = A$
 $A \text{ OR } 0 = A$
- **Complemento:**
 $A \text{ AND NOT } A = 0$
 $A \text{ OR NOT } A = 1$

4. SISTEMAS OCTAL Y HEXADECIMAL

Además del sistema binario, existen otros dos sistemas numéricos muy utilizados en informática: **el octal** y **el hexadecimal**. La razón es que permiten realizar conversiones muy fáciles hacia y desde el binario, ya que sus bases son potencias exactas de 2.

4.1. Sistema octal

El **sistema octal** tiene base **8**, lo que significa que utiliza los símbolos: **0, 1, 2, 3, 4, 5, 6, 7**. Su base es una potencia del sistema binario: es decir, con **3 bits** (3 dígitos binarios) se puede representar exactamente un dígito octal.

Decimal	Binario	Octal
0	000	0

1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

4.2. Conversión de binario a octal

Se agrupan los dígitos binarios en bloques de **3 bits**, comenzando por la derecha, y se sustituyen por el valor octal correspondiente.

Ejemplo:

$$1101011_{(2)} \Rightarrow \text{Se hacen grupos de tres elementos } \underline{1} \underline{101} \underline{011} \Rightarrow 153_{(8)}$$

4.3. Conversión de octal a binario

El proceso inverso consiste en convertir **cada dígito octal en su equivalente binario de 3 bits**.

Ejemplo:

$$7402_{(8)} \Rightarrow \text{Se hacen grupos de tres elementos } \underline{111} \underline{100} \underline{000} \underline{010} \Rightarrow 111100000010_{(2)}$$

4.4. Sistema hexadecimal

El **sistema hexadecimal** tiene base **16**. Como se necesitan más de diez símbolos, se utilizan las letras de la A a la F para representar los valores del 10 al 15.

El conjunto de símbolos es: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.

Decimal	Binario	Hexadecimal
0	0000	0

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

4.5. Conversión de binario a hexadecimal

Se agrupan los dígitos binarios en bloques de **4 bits**, comenzando por la derecha.

Ejemplo:

$$1101011_{(2)} \Rightarrow \text{Se hacen grupos de 4 elementos } \underline{110} \underline{1011} \Rightarrow 6B_{(16)}$$

Conversión de hexadecimal a binario

Cada dígito hexadecimal se convierte en su valor binario de **4 bits**.

Ejemplo:

$$7F0A_{(16)} \Rightarrow \text{Se hacen grupos de 4 elementos } \underline{0111} \underline{1111} \underline{0000} \underline{1010} = 111111100001010_{(2)}$$

Conversión de hexadecimal a octal

Se suele pasar primero por binario:

Ejemplo:

$$\begin{aligned} 6B_{(16)} &\Rightarrow \text{Se hacen grupos de 4 elementos } \underline{110} \underline{1011} \Rightarrow 1101011_{(2)} \Rightarrow \\ &\Rightarrow \text{Se hacen grupos de 3 elementos } \underline{1} \underline{101} \underline{011} \Rightarrow 153_{(8)} \end{aligned}$$

Nota importante

Las conversiones entre **octal/hexadecimal y decimal** (o viceversa) también se pueden hacer usando los métodos binario-decimal habituales, pero adaptados:

- Multiplicando cada dígito por una potencia de **8** (en octal) o **16** (en hexadecimal).
- O bien dividiendo sucesivamente entre 8 o 16, tomando los restos.

En el caso del hexadecimal, si el resto es mayor que 9, se sustituye por las letras **A–F**.

👉 Sin embargo, en la práctica suele ser **más rápido y sencillo convertir primero a binario** y desde ahí pasar al sistema deseado.

5. REPRESENTACIÓN ALFANUMÉRICA

5.1. Datos numéricos y alfanuméricos

Un dato se considera **numérico** cuando es posible realizar operaciones matemáticas con él. Por el contrario, un dato es **alfanumérico** cuando **no se pueden realizar operaciones matemáticas**:

- Numérico: ¿Cuántos años tienes? → 45
- Alfánnumérico: ¿Cómo te llamas? → "Roberto"

Para diferenciar claramente los datos alfanuméricos de los numéricos, es habitual representarlos entre **comillas simples o dobles**.

Un error común es pensar que los datos numéricos son solo números y que los alfanuméricos son únicamente letras. Esto no es correcto. Veamos ejemplos:

- ¿Cuál es tu dirección? → "Avenida de las Palmeras, 34"
- ¿Cuál es tu número de móvil? → "555341273"

En el primer caso, la información contiene **letras y números**. En el segundo caso, aunque solo hay

cifras, **no puede considerarse un número** porque no tiene sentido sumar o multiplicar números de teléfono. Por tanto, ambos ejemplos son **cadenas de caracteres** (strings).

5.2. Representación interna

Los ordenadores necesitan una forma estándar de representar caracteres alfanuméricos. Para ello utilizan **tablas de codificación**, donde cada número entero corresponde a un carácter.


Uno de los estándares más antiguos y conocidos es la **tabla ASCII (American Standard Code for Information Interchange)**.

- Utiliza 7 bits, por lo que puede representar $2^7 = 128$ caracteres.
- Entre ellos se incluyen letras mayúsculas y minúsculas del alfabeto inglés, números, signos de puntuación y algunos caracteres de control (no visibles), como el tabulador (código 9) o el retorno de carro (código 13).
- Incluso el espacio " " es un carácter (código 32).

Ejemplos:

- 73 (10) → "I"
- 105 (10) → "i"
- 50 (10) → "2"

El problema del **ASCII original** es que no incluye caracteres propios de otros idiomas, como la ñ, la ç o las vocales acentuadas. Para solventar esto, se creó el **ASCII extendido** de **8 bits** (256 caracteres), que incorpora símbolos gráficos y variantes del alfabeto latino.

 **Importante:** Hoy en día, tanto ASCII como ASCII extendido son insuficientes. Con la globalización e Internet, se requieren sistemas que soporten **todos los alfabetos del mundo** (chino, árabe, ruso, hebreo, etc.). Por ello se utiliza el estándar **Unicode**, siendo **UTF-8** la codificación más extendida, capaz de representar millones de caracteres distintos de forma eficiente.

6. SISTEMA DE UNIDADES

Como ya se explicó, el **bit** es la unidad mínima de información. Sin embargo, en la práctica se trabaja con **agrupaciones de bits**:

- 8 bits = 1 byte

Y a partir de ahí se definen múltiplos:


- Kilobyte (kB)
- Megabyte (MB)
- Gigabyte (GB)
- Terabyte (TB)
- ... y así sucesivamente.

Aquí surge una **doble notación**:

- En el **Sistema Internacional (SI)**, los múltiplos son potencias de 10.
 - 1 kB = 1000 bytes
- En **informática tradicional**, se han usado potencias de 2.
 - 1 KiB = 1024 bytes

Para evitar confusiones, se acordó el uso de prefijos especiales en binario (IEC 1998):

- **kB (kilobyte, decimal)**: 1000 bytes
- **KiB (kibibyte, binario)**: 1024 bytes
- **MB (megabyte)**: 1 000 000 bytes
- **MiB (mebibyte)**: 1 048 576 bytes
- ... y así sucesivamente (GiB, TiB, PiB, EiB, ZiB).

 **Importante:** Aunque en la práctica se suele usar el **Sistema Internacional (kB, MB, GB...)**, los valores **no son exactamente los mismos** que en el sistema binario (KiB, MiB, GiB...).


Por ejemplo:

- 1 MB = 1 000 000 bytes
- 1 MiB = 1 048 576 bytes



Atención: No confundir:

- **kB** = kilobyte
- **kb** = kilobit (8 veces más pequeño)

 **Interesante:** Todos los múltiplos (Mega, Giga, Tera, etc.) se escriben con inicial **mayúscula**, excepto el prefijo **kilo**, que por convención se escribe con minúscula (kB).

7. BIBLIOGRAFÍA

[1] Wikipedia. Signed Number Representations

https://en.wikipedia.org/wiki/Signed_number_representations

[2] Wikipedia. Signed Number Representations

https://en.wikipedia.org/wiki/Signed_number_representations