Los datos. Tipos de datos y su representación.

Desde el principio, los ordenadores se construyeron pensando en manipular datos. La idea era crear máquinas que realizaran cálculos de una forma rápida y eficaz.

Como todos sabemos, desde aquel primer ordenador basado en relés que creó *George Stibitz* en *1937*, todos los que han venido después han representado la información a partir de elementos *biestables*, es decir, elementos que pueden tomar dos valores diferentes.

En realidad esta lógica se aplica a multitud de soportes con características completamente diferentes: si pasa o no corriente eléctrica por un determinado punto, si existe o no un campo magnético en un lugar concreto de la superficie de un *disco duro*, si un punto en particular de la superficie de un *DVD* refleja o no la luz láser, etc. Es decir, dentro de un ordenador, la información se guarda en función de dos posibles estados.

Por este motivo, desde un primer momento, el sistema de numeración que emplearon los ordenadores fue el *binario*.

A diferencia del sistema de numeración *decimal*, que utiliza diez símbolos diferentes para representar cualquier cantidad (0 a 9), el sistema *binario* utiliza sólo dos (el 0 y el 1).

La cantidad mínima de información que puede representar un ordenador es un *dígito binario* (en inglés, ***B**inary dig**it***), aunque para nombrarlo suele utilizarse el acrónimo *bit*.

Por lo tanto, la información almacenada en el interior de un ordenador estará representada por una secuencia de unos y ceros (más o menos larga, en función del dato almacenado), que estará codificada siguiendo unas normas particulares. A esto es a lo que llamamos *representación*.

Lógicamente la representación de la información dependerá del tipo al que ésta pertenezca. A continuación nombraremos los tipos de representación más frecuentes.

Representación de números enteros con signo

Para representar un número negativo en matemáticas, sólo tenemos que hacer algo tan fácil como escribir un guión delante de la cantidad. Sin embargo, este fue el primer problema al que se enfrentaron los primeros diseñadores de ordenadores. Existen varios enfoques a este problema:

- *Módulo-signo*, también conocido por *signo-magnitud*.
- *Complemento a uno*, también llamado *complemento a la base menos uno*.
- *Complemento a dos*, también llamado *complemento a la base*.
- *En exceso a N*.

Módulo y signo (MS)

Este es el modo más sencillo para representar en binario un número entero con signo. La idea es que, si partimos de un número fijo de bits (8, 16, 32, …), se reserva el primero de ellos (el que hay más a la izquierda), para representar el signo. El resto representarán el valor absoluto del número en binario.

Por lo tanto, su rango de valores podríamos representarlo con la siguiente expresión, donde *X* representa el valor y *n* el número total de dígitos que usaremos para representarlo, incluido el signo:

$$\left(-\left(2^{(n-1)}-1\right) \leqslant X \leqslant 2^{(n-1)}-1\right)$$

Así, por ejemplo, si utilizamos 8 dígitos binarios, sólo usamos 7 para el valor absoluto. Dado que el número máximo de combinaciones posibles con 7 bits es de 128 (2^7), los posibles valores que podremos representar se encontrarán entre 0 y 127.

Por lo tanto, si tenemos en cuenta el signo, el rango de valores que podremos representar con 8 bits mediante representación en módulo y signo estará entre -127 y 127.

Veamos algunos ejemplos:

	7	6	5	4	3	2	1	0
37	0	0	1	0	0	1	0	1
-37	1	0	1	0	0	1	0	1
127	0	1	1	1	1	1	1	1
-127	1	1	1	1	1	1	1	1

En la siguiente tabla mostramos los rangos de valores para algunos tamaños de palabra frecuentes:

Nº de bits	Rango inferior	Rango superior
8	-127	127
16	-32767	32767
32	-2147483647	2147483647

Aunque se trata de un modo muy evidente de representación numérica, tiene dos inconvenientes: La primera, y menos importantes es que la representación del valor cero es doble (0 y -0). La segunda es que resulta complicado usarla en operaciones aritméticas.

Complemento a uno (C-1)

Como en el caso anterior, su objetivo es representar números enteros, tanto positivos como negativos. Además, también en este caso se utiliza el primer bit para representar el signo.

Como en módulo y signo, los números positivos se representan poniendo a cero (positivo) el primer bit, usando el resto para representar el valor absoluto del número en binario.

Sin embargo, un valor negativo se representa complementando el valor positivo correspondiente. Es decir, invirtiendo el valor de todos los bits del número positivo, incluido el que representa al signo.

Veamos algunos ejemplos usando una representación de 8 bits:

	7	6	5	4	3	2	1	0
37	0	0	1	0	0	1	0	1
-37	1	1	0	1	1	0	1	0
127	0	1	1	1	1	1	1	1
-127	1	0	0	0	0	0	0	0

Como habrás deducido, el rango de valores posibles para un determinado número de bits es idéntico al obtenido en módulo y signo. Además, también aquí la representación de valor cero es doble. Por ejemplo, utilizando 8 bits, podríamos representarlo como 00000000 (0) y 11111111 (-0).

Tampoco suele emplearse para realizar operaciones aritméticas.

Complemento a dos (C-2)

La representación en Complemento a dos es idéntica a la que hemos aplicado en complemento a uno, pero sumando después el valor 1 a los números negativos. Si en esta suma se produce acarreo, deberemos despreciarlo.

Veamos algunos ejemplos representando valores en complemento a uno y en complemento a dos usando una representación de 8 bits:

	7	6	5	4	3	2	1	0
127 (C-1 y C-2)	0	1	1	1	1	1	1	1
-127 (C-1)	1	0	0	0	0	0	0	0
-127 (C-2)	1	0	0	0	0	0	0	1
32 (C-1 y C-2)	0	0	1	0	0	0	0	0
-32 (C-1)	1	1	0	1	1	1	1	1
-32 (C-2)	1	1	1	0	0	0	0	0

En esta situación, cabría preguntarse qué ocurre cuando representamos 0 partiendo de la segunda representación de complemento a 1 (11111111), es decir, partiendo de la representación negativa. Como puedes ver en la siguiente tabla, al sumar 1, todos los bits quedarían a cero, obteniendo al final un acarreo de 1 que se despreciaría.

	7	6	5	4	3	2	1	0
-0 (C-1)	1	1	1	1	1	1	1	1
-0 (C-2)	0	0	0	0	0	0	0	0

En definitiva, llegamos a la conclusión de que sólo existe una representación del valor 0, lo que implica también disponer de un valor negativo más en el rango y, por consiguiente, un rango asimétrico:

	7	6	5	4	3	2	1	0
-127 (C-2)	1	0	0	0	0	0	0	1
-128 (C-2)	1	0	0	0	0	0	0	0

Por lo tanto, su rango de valores podríamos representarlo son la siguiente expresión, donde X representa el valor y n el número total de dígitos que usaremos para representarlo, incluido el signo:

$$\left(-2^{(n-1)} \leqslant X \leqslant 2^{(n-1)} - 1\right)$$

En la siguiente tabla mostramos los rangos de valores para algunos tamaños de palabra frecuentes:

Nº de bits	Rango inferior	Rango superior
8	-128	127
16	-32768	32767
32	-2147483648	2147483647

A diferencia de los casos anteriores, la representación en complemento a dos si es muy utilizada para operaciones aritméticas con números enteros. Y para ilustrarlo, pondremos algunos ejemplos.

Suma en complemento a dos

La suma se realiza de la forma habitual, sin tener en cuenta si los números son positivos o negativos.

El único inconveniente puede producirse cuando sumamos valores con el mismo signo (dos valores positivos o dos valores negativos. En estos casos, puede ocurrir que el resultado obtenido sobrepase los límites del rango que ofrezca el número de bits que estemos empleando.

Por ejemplo, esto ocurriría si, utilizando una representación de 8 bits, sumáramos los valores 96 y 37. En principio, podremos representar, de forma correcta, ambos valores en complemento a dos.

Sin embargo, al sumarlos, el resultado arrojaría el valor 133, que excede el valor máximo representable (que es 127).

Si ocurre esto cuando un ordenador trata de realizar una operación, se obtiene un error de **desbordamiento** (en inglés, **overflow**).

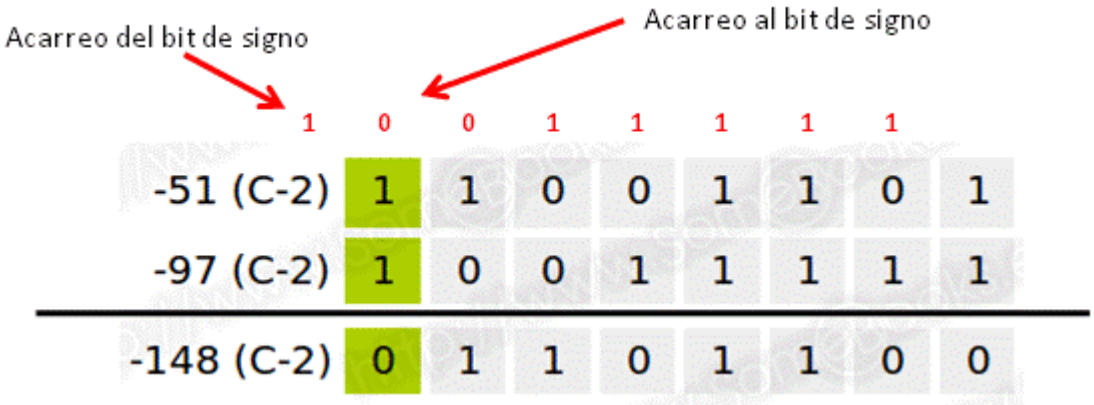
Decimos que hay overflow o desbordamiento **en una operación de suma o resta cuando "el acarreo al bit de signo es distinto del acarreo del bit de signo".**

Decimos que hay desbordamiento u overflow **en una operación de suma cuando "los sumandos tiene el mismo signo y el resultado tiene un signo diferente".**

Cuando los sumandos tienen el mismo signo, detectaremos que se ha producido un desbordamiento cuando el resultado tiene un signo diferente.

Por ejemplo, si realizamos la suma anterior, el resultado sería este:

Y si lo intentamos con números negativos…



En este caso, hemos despreciado el último acarreo.

Resta en Complemento a dos

Para restar dos números en complemento a dos, basta con volver a calcular el complemento a 2 del sustraendo (lo que equivale a cambiarlo de signo) y realizar la suma con el método explicado más arriba.

Para comprobarlo, vamos a restar dos números con los que hemos trabajado antes: 96 y 37.

	7	6	5	4	3	2	1	0
96 (C-2)	0	1	1	0	0	0	0	0
37 (C-2)	0	0	1	0	0	1	0	1

Lo primero será cambiar el signo de 37 en complemento a dos:

	7	6	5	4	3	2	1	0
-37 (C-2)	1	1	0	1	1	0	1	1

A continuación, realizamos la suma:

	7	6	5	4	3	2	1	0
96 (C-2)	0	1	1	0	0	0	0	0
-37 (C-2)	1	1	0	1	1	0	1	1
59 (C-2)	0	0	1	1	1	0	1	1

La importancia de este funcionamiento radica en que los ordenadores que representen los valores enteros en complemento a dos sólo necesitarán un circuito sumador para realizar tanto sumas como restas. Por otro lado, los circuitos que complementan valores binarios son muy sencillos de implementar.

Extensión del número de bits

- Un mismo número se puede representar con diferente número de bits:

$+25_{10} = 011001_{SM} = 0000011001_{SM} = 000000000011001_{SM}$

- Para extender el signo:
  - En SM:
    - Añadir ceros justo después del signo
  - En Ca1 y Ca2:
    - Si es positivo añadir ceros a la izquierda
    - Si es negativo añadir unos a la izquierda

$+25_{10} = 011001_{Ca2} = 0000011001_{Ca2} = 000000000011001_{Ca2}$

$-25_{10} = 100111_{Ca2} = 1111100111_{Ca2} = 1111111111100111_{Ca2}$

Exceso a 2^{n-1}.

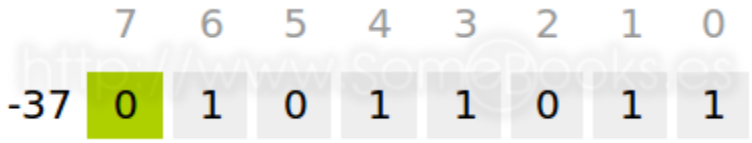
Es otra de las formas habituales de representar números enteros.

En este caso, antes de representar cada valor, se le suma el resultado de calcular 2^{n-1}, donde *n* representa el número de dígitos utilizados para la representación. Es decir, si vamos a representar valores usando 8 bits, a cada valor a representar debemos sumarle 128 (2^7). A continuación, hacemos la conversión a binario sin tener en cuenta el signo.

Veamos algunos ejemplos usando una representación de 8 bits:

- Ejemplo 1: el valor 37.
  - Como hemos dicho, lo primero será sumarle 128. El resultado es 165.
  - A continuación, convertimos 165 en binario puro. El resultado es:

- Ejemplo 2: el valor -37.
  - Como antes, lo primero será sumarle 128. El resultado es 91.
  - Si ahora convertimos 91 a binario puro, obtenemos:



- Ejemplo 3: el valor 0.
  - De nuevo, comenzamos sumando 128, aunque, en este caso, el resultado es 128.
  - A continuación, convertimos 128 en binario puro. El resultado es:

Si seguimos haciendo pruebas, comprobaremos que el bit 7 de todos los números positivos acabará siendo 1. Y el de los números negativos será 0. Además, comprobamos que el valor cero se representa en el intervalo de números positivos, de lo que se desprende que el rango de representación es asimétrico (como nos ocurría en *complemento a dos*), pudiendo representarlo con la siguiente expresión, donde X representa el valor y n el número total de dígitos que usaremos para representarlo:

$$\left(-2^{(n-1)} \leqslant X \leqslant 2^{(n-1)} - 1\right)$$

En la siguiente tabla mostramos los rangos de representación para algunos tamaños de palabra frecuentes:

Nº de bits	Rango inferior	Rango superior
8	-128	127
16	-32768	32767
32	-2147483648	2147483647

Representación de números con decimales

Cuando se trata de representar números con decimales, los tipos de representación más frecuentes son estos:

- *Decimal codificado en binario* o *BCD* (del inglés, *Binary-Coded Decimal*).
- *Decimal desempaquetado*.
- *Decimal empaquetado*.
- *Coma flotante*.

Representación usando BCD

La representación en *BCD* asigna 4 bits para representar en binario cada uno de los dígitos de un número decimal. Así, por ejemplo, el número 1 podría representarse como 0001. Siguiendo esta idea general, suelen aplicarse distintos modos de representación. En la siguiente tabla mostramos los más frecuentes:

Representaciones BCD

Decimal	Natural	Aiken	5 4 2 1	Exceso 3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0010	0010	0101
3	0011	0011	0011	0110
4	0100	0100	0100	0111
5	0101	1011	1000	1000
6	0110	1100	1001	1001
7	0111	1101	1010	1010
8	1000	1110	1011	1011
9	1001	1111	1100	1100

Aunque el método es indiferente de la representación concreta que utilicemos, para esta documentación, utilizaremos la representación *Natural*.

Observa que, a pesar de que, con cuatro bits, podríamos representar hasta quince valores, en *BCD* siempre se desprecian seis de ellos.

Cuando el número decimal que queremos representar con *BCD* tiene más de un dígito, se usan tantos grupos de cuatro dígitos binarios como dígitos decimales tenga el número de origen. Como dicho así puede resultar algo lioso, veamos el siguiente ejemplo:

La ventaja que ofrece *BCD* es que evita la pérdida de exactitud producida por las conversiones de decimal a binario y viceversa.

Por contra, el inconveniente es que los cálculos con números representados en *BCD* son más complejos y, por consiguiente, emplean más tiempo de proceso.

Decimal desempaquetado

La representación en *Decimal desempaquetado* trata de suplir una de las carencias de *BCD*: su imposibilidad de representar números negativos.

Para solventarlo, *Decimal desempaquetado* utiliza un byte para representar cada dígito decimal. El *nibble* de la derecha se corresponde con la representación en *BCD*, mientras que el *nibble* de la izquierda puede tener tres valores posibles:

- El valor $1111_{(2}$ en todos los dígitos excepto en el primero.
- El valor $1100_{(2}$ en el primer dígito, cuando el número es positivo.
- El valor $1101_{(2}$ en el primer dígito, cuando el número es negativo.

A modo de ejemplo, veamos la representación del número 425 que ya mostrábamos en *BCD*:

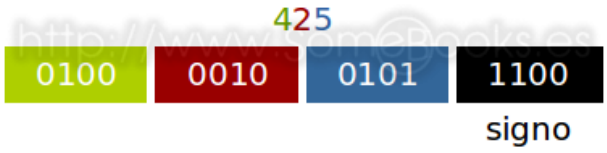
Y si el número fuese -425, quedaría así:

Este método facilita mucho la representación de los valores, ya que utiliza un byte binario para cada dígito decimal. Sin embargo, conlleva el uso de un gran número de bits innecesarios. Piensa que, con cada byte podemos representar 256 valores diferentes, de los que sólo se utilizan 10 para cada dígito decimal (20 en el caso del primero).

Decimal empaquetado

Resuelve los inconvenientes del sistema decimal desempaquetado ya que elimina el *nibble* de la izquierda para cada dígito, salvo el primero. De esta forma, se pueden representar dos dígitos decimales en cada byte (por ese motivo se le llama *empaquetado*).

El signo siempre se representa en el *nibble* de la derecha del byte más a la derecha.



Punto fijo y punto flotante.

Los números con decimales Se pueden representar con punto fijo o con punto flotante.

El punto, o coma decimal, es quien marca el modelo.

Punto fijo o coma fija:

La coma decimal se considera fija en un punto.

Ejemplo: Datos de 32 bits, utilizar 20 bits para la parte entera y 12 bits para los decimales

Fácil realizar las operaciones de sumas, restas, multiplicaciones y divisiones

Notación: Qm,n

m: número de bits de la parte entera (opcional)

n: número de bits para la parte decimal

Se utiliza un bit adicional para el signo (en total hacen falta m+n+1 bits).

Ejemplos: Q16,16, Q.32, etc.

En Informática no se suele utilizar la representación con punto fijo.

Representación de números en coma flotante

Cuando se trata de representar números muy grandes, o muy pequeños, se utiliza la representación en *coma flotante*, que está basada en la *notación científica* utilizada habitualmente en matemáticas.

La notación científica consiste en representar las cantidades en función de una potencia de 10. Así, el número 6.400.000 se representaría como $6,4 \times 10^6$, mientras que el número 64.000.000 se representaría como $6,4 \times 10^7$.

En informática, normalmente se utilizan 32 bits (simple precisión) o 64 bits (doble precisión) para representar cada número en *coma flotante*.

La representación de números enteros no es adecuada para un gran número de situaciones. En algunos casos, podríamos utilizar parte de los dígitos para representar valores decimales, pero estaríamos reduciendo el rango de valores representables.

Para entender la idea, podemos pensar en la representación en *complemento a 2*, utilizando 32 bits. En condiciones normales, el rango de representación iría desde -2.147.483.648 hasta 2.147.483.647. Sin embargo, si reserváramos los dos últimos dígitos para representar valores decimales, el rango de representación se reduciría desde -21.474.836,48 hasta 21.474.836,47. Imagina lo que ocurrirá cuando necesitemos una mayor precisión decimal.

Podríamos pensar que la solución está en aumentar el número de bits que utilizamos en la representación (pasar de 32 a 64, o a 128). Sin embargo incluso en ese caso, podemos encontrarnos, sobre todo en el ámbito científico, con cálculos que necesiten manejar valores extraordinariamente grandes o extraordinariamente pequeños.

Para resolver esta situación, en el campo de las matemáticas se utiliza notación científica, que consiste en representar las cantidades en función de una

potencia de 10. Así, por ejemplo, para representar la distancia más grande que podemos observar en el universo, que es de 740.000.000.000.000.000.000.000.000 metros, podemos utilizar la expresión $7{,}4 \times 10^{26}$ que, como puedes ver, resulta mucho más manejable.

Recuerda: 1 gúgol, que es el origen del nombre Google, equivale a 10.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000 unidades, pero podemos representarlo, cómodamente, como 10^{100}.

Lo mismo ocurre con valores muy pequeños. Por ejemplo, la masa de un electrón es de 0,00000000000000000000000000000091 Kg, pero podemos representarlo como $9{,}1 \times 10^{-31}$.

A esta forma de notación se le llama *representación en coma flotante*. En ella, podemos diferenciar tres elementos:

- La mantisa: Es el primer número de la expresión.
- La base: Representa el sistema de numeración en el que estamos trabajando. En matemáticas siempre es 10.
- La característica: Es el exponente al que debe elevarse la base

Cómo se utiliza representación en coma flotante en un ordenador.

Lo primero que debemos contemplar cuando pretendemos representar un número en *coma flotante* dentro de un ordenador es que, en lugar de utilizar base 10, emplearemos base 2. Además, deben seguirse las siguientes pautas del *IEEE*:

- El signo debe representarse con un solo bit (0 = positivo y 1 = negativo).
- La característica se representará mediante exceso a 2^{N-1}-1. Es decir, en el caso de que el número de bits a utilizar sea 8, al exponente le sumaremos 127 (2^{8-1}-1 = 127) para almacenarlo. El motivo es que, al manejar todos los valores como positivos, se simplifican algunas operaciones. Además, sólo

se utiliza el rango 1 a 254, porque se le otorga un significado especial a 0 ($00000000_{(2}$) y a 255 ($11111111_{(2}$).

- La mantisa se normaliza para situar la coma detrás del primer dígito que sea igual a 1 por la izquierda (por ejemplo: 1,100101). Además, ese primer dígito puede darse por sobreentendido, lo que permite no representarlo y aumentar la precisión en un bit.
- La base, que siempre es 2, también se da por sobreentendida y no se representa.

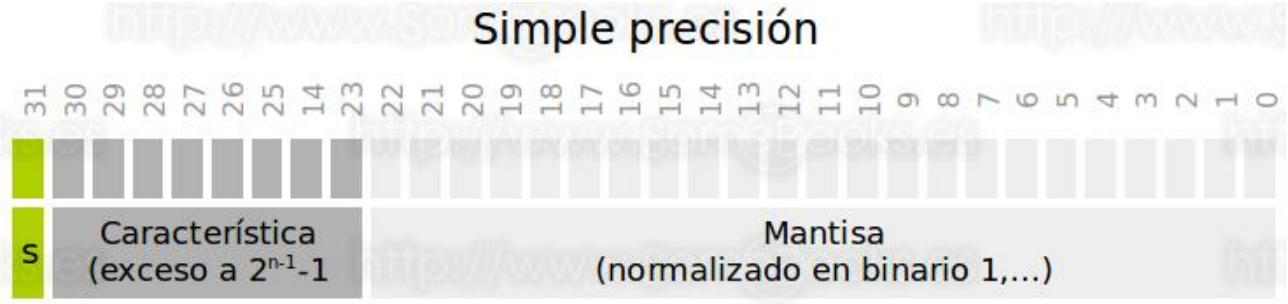
A esta norma se le plantea una excepción evidente: el valor cero debe poder representarse, pero la mantisa nunca tendrá un 1 tras el que situar la coma. Para resolverlo, se estableció que el valor cero se representaría con los bits de característica y mantisa a cero. Además de ésta, también se contemplan otras excepciones que se describen en la siguiente tabla:

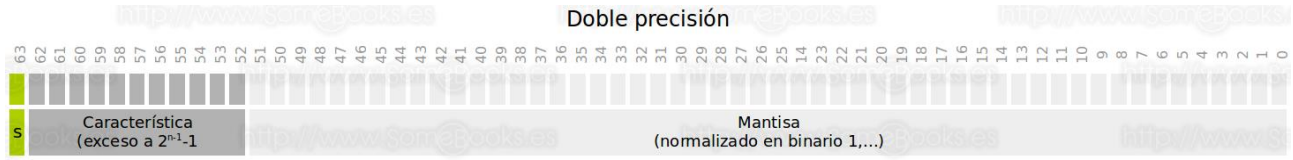
Excepción	Característica	Mantisa
Representación del cero	0	0
Representación de infinito	255	0
Valores desnormalizados	0	Distinto de cero
NaN (no numérico)	255	Distinto de cero

El IEEE (*Institute of Electrical and Electronic Engineering*) es un organismo de ámbito mundial encargado de desarrollar estándares relacionados con la tecnología.

La norma *IEEE754* establece dos formas diferentes de representación:

- Utilizando 32 bits, que recibe el nombre de *simple precisión*, donde se utiliza 1 bit para el signo, 8 bits para la característica y 23 bits para la mantisa.
- Utilizando 64 bits, que recibe el nombre de *doble precisión*, en la que se utiliza 1 bit para el signo, 11 bits para la característica y 52 bits para la mantisa.



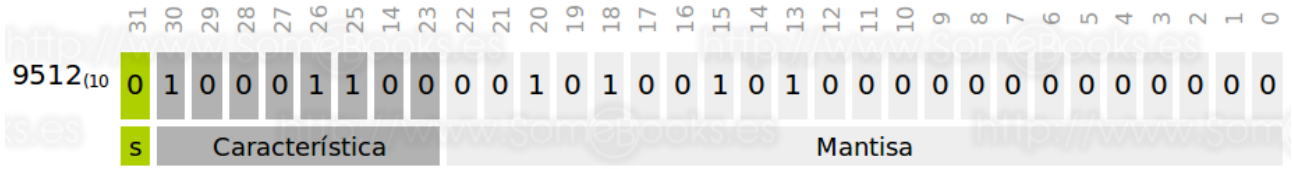


Aún así, hay fabricantes que añadan sus propias representaciones, como la *precisión ampliada* de *IBM*, que utiliza 128 bits.

A modo de ejemplo, representaremos el número $9512_{(10}$ en coma flotante. Para lograrlo, seguiremos el siguiente proceso:

1. Primero lo convertimos en binario: El resultado es $10010100101000_{(2}$.
2. A continuación, movemos la coma 13 posiciones a la izquierda y, para dejar el valor constante, multiplicamos por la potencia correspondiente: $1,0010100101000_{(2} \times 2^{13}{}_{(10}$.
3. Después, comenzamos la representación:
    a. Como el número es positivo, el primer bit será 0.
    b. La característica la representamos como 13 + 127 (140) en binario: 10001100
    c. La mantisa estará formada por los dígitos que aparecen después de la coma, completando por la derecha con tantos ceros como sean necesarios para completar los 23 dígitos que debe ocupar la mantisa. Es decir: 00101001010000000000000

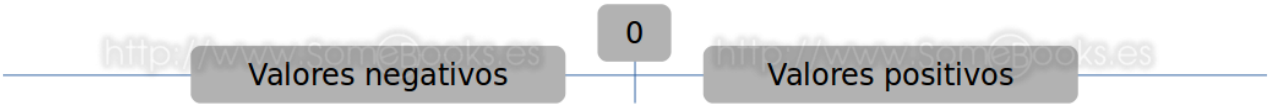
Por lo tanto, la representación quedaría así:



Aunque es muy común utilizar la notación en hexadecimal para que resulte más manejable. Para lograrlo, sólo tenemos que agrupar los dígitos de cuatro en cuatro y buscar su equivalencia. Para el ejemplo anterior, el resultado sería este:

$$4614A000_{(16}$$

Como estamos utilizando un bit para el signo, tendremos dos rangos de representación: uno para los números positivos y otro para los números negativos.



Para averiguar cuáles son esos rangos de valores, debemos pensar en cuál es el valor mínimo y el valor máximo de la característica y de la mantisa. Pondremos como ejemplo la simple *precisión*:

- Para la característica:
  - El valor mínimo es 1 ($00000001_{(2}$), ya que el 0 es un valor excepcional. Como está representado en exceso a 127, el resultado será 1 – 127 = -126.
  - El valor máximo es 254 ($11111110_{(2}$), ya que el 255 es un valor excepcional. De nuevo, como se representa en exceso a 127, el resultado será 254 – 127 = 127.
- Para la mantisa:
  - El valor mínimo será 1 seguido de una coma y 23 ceros ($1,000…0_{(2}$). En definitiva, 1
  - El valor máximo será 1 seguido de una coma y 23 unos ($1,111…1_{(2}$), lo que es, prácticamente, $2 – 2^{-23}$, un valor bastante próximo a 2.

Así pues, los números negativos se encontrarán entre

$(-2+2^{-23}) \times 2^{127}$ y -1×2^{-126}

mientras que los positivos estarán entre 1×2^{-126} y $(2 – 2^{-23}) \times 2^{127}$.

Para la doble precisión podríamos hacer los mismos cálculos y obtendríamos que los números negativos se encontrarán entre

$(-2+2^{-52}) \times 2^{1023}$ y -1×2^{-1022}

mientras que los positivos estarán entre 1×2^{-1022} y $(2 – 2^{-52}) \times 2^{1023}$.

Representación de caracteres

Como podemos deducir de todo lo que hemos dicho hasta el momento en este apartado, dentro de un ordenador todo son números. Por lo tanto, si queremos representar textos, necesitaremos establecer una correlación entre un valor entero y un carácter asociado a ese valor.

De esta forma, obtendremos una tabla donde cada carácter que queremos representar tiene asociado un valor numérico en *binario*. A esta tabla la llamaremos *juego de caracteres*.

Los juegos de caracteres de los primeros ordenadores, dada su escasa capacidad, trataban de utilizar el menor número posible de bits para representar cada carácter. Por ejemplo, *FIELDATA* usaba sólo 6 bits, lo que hacía que sólo

pudieran representarse 64 caracteres diferentes (2^6=64), que incluían únicamente las letras del idioma inglés, en mayúsculas, los números decimales y unos cuantos símbolos de puntuación y matemáticos.

En 1963, el Comité Estadounidense de Estándares creó un juego de caracteres llamado *ASCII* (del inglés, *American Standard Code for Information Interchange*). Utilizaba 7 bits (2^7=128) e incluía letras minúsculas.

Una característica curiosa es que la representación de una letra en mayúsculas y en minúsculas sólo se diferencia en un bit, lo que hace que sea muy sencilla la conversión entre unas y otras. Veamos, a modo de ejemplo, la representación de la letra A en mayúscula y en minúscula:

Carácter	Binario	Decimal
A	01000001	65
a	01100001	97

La variante de 8 bits suele llamarse *ASCII extendido*, o *ASCII ampliado* y, además de los caracteres propios de algunos idiomas occidentales (como nuestra *ñ* o la *ç* francesa) incluía caracteres *semigráficos* que, combinados, podían dar cierta ilusión de gráficos rudimentarios en monitores que sólo permitían representar texto.

A continuación, se incluye el mapa completo de caracteres *ASCII*

Caracteres de control ASCII

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

Caracteres ASCII imprimibles

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

elCodigoASCII.com.ar

ASCII extendido

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Como puede comprenderse fácilmente, *ASCII* estaba pensado para satisfacer las necesidades de los idiomas occidentales, pero dejaba fuera cualquier otro que estuviese basado en caracteres diferentes (griego, árabe, ruso, chino, …). Tampoco es adecuado para la representación de textos en lenguas clásicas ni para la utilización de símbolos científicos.

Las primeras soluciones consistieron en crear variantes locales de los juegos de caracteres donde un mismo valor podía representar diferentes caracteres según la variante que usáramos. Esto podía ocasionar problemas, sobre todo en sistemas servidores o cuando varios ordenadores compartían información entre ellos.

Para resolver este tipo de situaciones, varios ingenieros de *Apple* y *Xerox* propusieron en 1988 un nuevo estándar de codificación de caracteres al que se llamó *Unicode* (el primer borrador recibió el nombre de *Unicode88* y utilizaba 16 bits para representar caracteres). Sin embargo, el Consorcio *Unicode* no se constituyó hasta 1991.

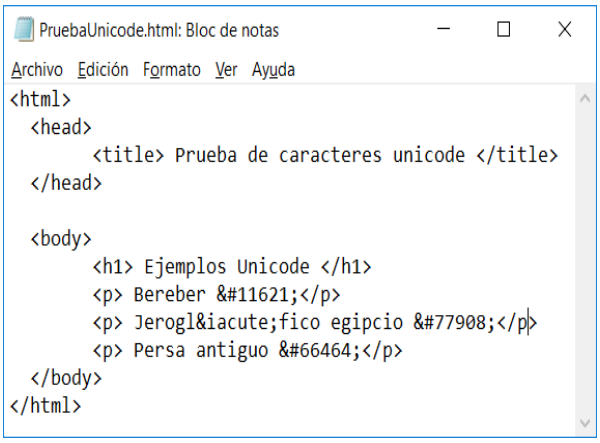
En la actualidad, se trata del esquema de representación de caracteres más completo y permite que los ordenadores se adapten al modo en el que se representa el texto en prácticamente todos los idiomas del planeta.

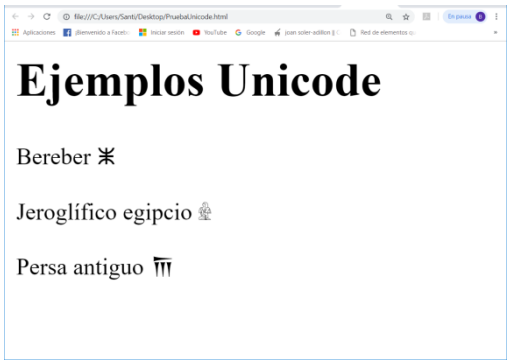
Una de las mayores ventajas de *Unicode* es que ofrece un mismo valor numérico para cada uno de los caracteres, de forma independiente del *hardware*, del *software* y del idioma que se estén utilizando. Esto permite que programas y sitios web se adapten fácilmente a diferentes contextos de uso de forma sencilla y que los datos puedan compartirse entre sistemas diferentes sin verse alterados.

Hoy en día, *Unicode* ha sido asumida por la mayoría de las empresas líderes de la industria informática (como *Microsoft*, *Apple*, *Oracle*, *IBM*, etc.) y por casi todos los estándares actuales (como *LDAP*, *XML*, *Java*, etc).

Si quieres consultar todos los esquemas disponibles, puedes recurrir a la página **https://unicode-table.com/es**

Se puede probar la escritura de todos los caracteres desde una pequeña página web como la del ejemplo:





También se puede copiar el símbolo desde la web anterior y pegar en el bloc de notas. En este caso al guardar deberemos cambiar de la codificación ANSI (por defecto) a la codificación Unicode.

Otros tipos de representaciones

Hasta ahora, hemos hablado del modo en el que un ordenador puede representar los tipos de datos básicos. Sin embargo, existe una gran diversidad de información que se codifica y se manipula dentro de un ordenador. Veamos algunos ejemplos de los formatos de codificación más comunes:

- Para audio: *mp3*, *wav*, *ogg*, *aiff*, …
- Para vídeo: *mpeg*
- Para imágenes: *jpeg, png*, *gif*, *tiff*, …

Hasta ahora hemos hablado de datos e información sin hacer distinciones entre ambos conceptos. Sin embargo, no se trata de dos términos equivalentes.

El concepto de *dato* se puede definir como la representación simbólica de una propiedad relativa a una entidad. El concepto de *información* se define como un conjunto organizado de datos relativos a una entidad.