



## Representación de Datos

Sabemos que cuando se ejecuta un programa, tanto el programa, como los datos que éste utiliza, se almacenan en la memoria. También sabemos que las celdas de la memoria sólo pueden almacenar datos numéricos, representados como unos y ceros. Por esto, tanto los programas como sus datos deben representarse como unos y ceros para que puedan ser almacenados en la memoria.

Las instrucciones de un programa se representan mediante una secuencia de bits, de acuerdo a una tabla que asocia una de estas secuencias a cada código de operación del conjunto de instrucciones del procesador.

Los datos deben *codificarse* de alguna forma, para que de esa forma sea posible almacenarlos en la memoria. La forma en que ésto se hace depende del tipo de datos, ya sean éstos números enteros, números reales, caracteres, figuras, imágenes, animaciones, audio o video. Para cada tipo de datos existen múltiples estándares de codificación.

## Representación de enteros positivos

Los computadores son capaces de manipular números, compuestos por unos y ceros. Toda la información y los programas deben convertirse a unos y ceros para que sean tratables. Esta restricción obliga al uso de un sistema de numeración distinto al que estamos acostumbrados, conocido como sistema decimal, o base 10.

En el sistema decimal, existen 10 dígitos, y con esos dígitos representamos cualquier número, de acuerdo a un esquema posicional. Este esquema define el valor de cada dígito, de acuerdo a su posición en el número. El dígito de más a la derecha es el de menor valor, pues corresponde a las unidades, es decir  $10^0$ . En el siguiente dígito, hacia la izquierda, cada unidad vale 10 veces más, pues representa las decenas, es decir  $10^1$ , y así sucesivamente.

### Sistema binario

El sistema binario, o base 2, es el más apropiado para representar números en un computador, pues utiliza sólo 2 dígitos: 0 y 1. El esquema posicional es el mismo que para el caso anterior, con la diferencia de que las posiciones se asocian con potencias de 2, pues 2 es la base del sistema. Así, el dígito de más a la derecha es el menos significativo, pues corresponde a las unidades, es decir,  $2^0$ . Después, moviéndose hacia la izquierda, los dígitos empiezan a tener mayor valor:  $2^1$ ,  $2^2$ ,  $2^3$ ,  $2^4$ , y así sucesivamente.

### Interpretación de números binarios

De acuerdo al esquema posicional descrito, interpretar un número binario se reduce a calcular una sumatoria. Supongamos que el número binario está compuesto por  $n$  dígitos, numerados de 0 a  $n - 1$  empezando desde la derecha:

$$d_{n-1}d_{n-2} \cdots d_2d_1d_0$$

La interpretación de este número, en el sistema decimal, puede obtenerse calculando la sumatoria:

$$\sum_{i=0}^{n-1} d_i \times 2^i$$

Observe que cada dígito es multiplicado por la potencia de 2 correspondiente a su posición. La posición de más a la derecha es la 0, y se van incrementando hacia la izquierda. También note que, dado que los únicos posibles valores para un dígito son 0 y 1, muchos de los términos de la sumatoria serán 0, y el resto serán potencias de 2, multiplicadas por 1.

En resumen, para calcular la equivalencia decimal de un número binario, basta sumar las potencias de 2 que corresponden a las posiciones del número binario en que se encuentra un 1. El resto de las posiciones, es decir, las que tienen un 0, no influyen en el resultado.

### Conversión a binario

La conversión de un número decimal a binario involucra una serie de divisiones sucesivas por 2. Se empieza dividiendo el número original, obteniendo un cociente y un residuo. El cociente se divide una vez más, y así sucesivamente hasta llegar a un cociente igual a 0. Posteriormente, los residuos se colocan en el orden inverso a como fueron obtenidos, y el resultado es el número binario. Observe que como estamos dividiendo por 2, los únicos residuos posibles son 0 y 1.

Este procedimiento se comprende mejor con un ejemplo. Consideremos el número decimal 139. El Cuadro 1 muestra la conversión a binario de este número. Puede verse como el procedimiento se detiene cuando el cociente de la división es 0. Para interpretar el resultado, basta con leer los residuos en el orden inverso al que se produjeron, es decir, 10001011.

Dividendo	Cociente ( $\div 2$ )	Resto
139	69	1
69	34	1
34	17	0
17	8	1
8	4	0
4	2	0
2	1	0
1	0	1

Cuadro 1: Conversión a binario del número decimal 139

Cuando ya se tiene suficiente experiencia, la conversión a binario se vuelve más simple. En forma mental es posible descomponer un número decimal en las potencias de 2 que lo conforman, es decir, aquellas potencias de 2 que, al ser sumadas, producen el número que se quiere convertir. Esto es particularmente simple para números pequeños.

Por ejemplo, consideremos el número decimal 72. Es posible darse cuenta que la potencia más grande que no sobrepasa al número en cuestión es  $64 = 2^6$ . Luego, se trabaja con el resto, es decir, 8, el cual resulta ser una potencia:  $8 = 2^3$ . Por lo tanto, en el número 72 participan las potencias 6 y 3, por lo que basta con ubicar unos en esas posiciones del número binario. El resto de los bits serán ceros. Así, 72 en decimal equivale a 01001000 en binario, utilizando 8 bits.

## Otros sistemas numéricos

Un problema que existe con la notación binaria es que las representaciones de los números son muy grandes, pues se cuenta sólo con 2 dígitos. Entre mayor es la cantidad de dígitos con que cuenta un sistema numérico, más pequeñas serán las representaciones.

Por este motivo, es común utilizar sistemas con una base mayor, para manipular los números. La base 10 no es una buena elección, pues la conversión con la base 2 no es del todo expedita. Una mejor alternativa la constituye la base 16, también conocida como hexadecimal, o la base 8, también conocida como octal. La ventaja que tienen estas bases es que la conversión con el sistema binario es muy simple.

En el sistema hexadecimal se cuenta con 16 dígitos, compuestos por los dígitos del sistema decimal: 0, 1, ..., 9 junto con las letras  $A, B, C, D, E, F$ , en ese orden de importancia. Por ejemplo, el dígito hexadecimal  $E$  equivale al número 14 en el sistema decimal. Observe que, en este ejemplo, un número que en hexadecimal se representa con un único dígito, requirió de dos dígitos en decimal.

La conversión entre el sistema binario y el sistema hexadecimal se hace agrupando los bits en grupos de cuatro, empezando desde la derecha. Cada grupo de cuatro se convierte a un dígito del sistema hexadecimal. Por ejemplo, si un grupo de bits es 0101, se convertirá al dígito hexadecimal 5. Por otra parte, el grupo de bits 1111 se convertirá al dígito hexadecimal  $F$ .

Consideremos el siguiente número binario, compuesto por 16 bits:

010111110100011

Para convertir este número a hexadecimal bastaría con formar 4 grupos de 4 bits, y convertir cada uno de ellos al dígito hexadecimal correspondiente. Así, tendríamos:

0101 = 5      1111 =  $F$       1010 =  $A$       0011 = 3

Por lo tanto, el equivalente hexadecimal sería  $5FA3$ .

Con el objeto de diferenciar números escritos en diferentes sistemas numéricos, es común agregar un subíndice con la base. Así, podríamos escribir:

$010111110100011_2 = 5FA3_{16} = 24483_{10}$

Para el caso de los números hexadecimales, se acostumbra agregar al final una  $x$  en minúscula. Por ejemplo:

$5FA3_{16} = 5FA3x$

La conversión de hexadecimal a binario es muy simple también. Por ejemplo, consideremos el número  $01F0_{16}$ . Para hacer la conversión basta con convertir cada dígito en su equivalente de cuatro bits:

0 = 0000      1 = 0001       $F$  = 1111      0 = 0000

Luego, los bits se agrupan en el mismo orden en que se encontraban los dígitos correspondientes en el número original. El resultado sería:

$01F0_{16} = 0000000111110000_2$

## Operaciones con números binarios

Las operaciones con números expresados en sistemas numéricos distintos al decimal, se llevan a cabo siguiendo las mismas reglas a las que estamos acostumbrados.

Consideremos en primera instancia algunas operaciones elementales, que conocemos muy bien del sistema decimal. Analicemos la multiplicación por 10 de un número decimal. En este caso, 10 es la base del sistema numérico, por lo que el resultado se obtiene fácilmente, agregando un dígito más, un 0, a la derecha del número. Así, por ejemplo,  $45 \times 10 = 450$ .

Acarreo	1	1	1	0	0
Número 1	0	1	1	1	1
Número 2	0	0	1	1	0
Resultado	1	0	1	0	1

Cuadro 2: Suma de dos números binarios

En binario el comportamiento es el mismo, cuando multiplicamos por la base del sistema numérico, es decir, por 2, o mejor escrito, por  $10_2$ . Por ejemplo,  $101_2 \times 10_2 = 1010_2$ . Cuando analizamos esta operación en notación decimal, podemos ver que el equivalente sería:  $5_{10} \times 2_{10} = 10_{10}$ .

Por otra parte, cuando dividimos por 10 en el sistema decimal, el resultado se obtiene fácilmente al truncar el dígito de la derecha del número que se está dividiendo. En el sistema binario el mecanismo es el mismo, cuando dividimos por la base, es decir, por  $10_2$ . Por ejemplo,  $1011_2 \div 10_2 = 101_2$ , con un residuo de 1. Si analizamos esto en decimal, la operación es:  $11_{10} \div 2_{10} = 5_{10}$ , con un residuo de 1.

Operaciones más complejas pueden llevarse a cabo utilizando las mismas reglas que se aplican a los números decimales. Consideremos el caso de la suma. Cuando sumamos dos números en decimal, empezamos sumando los dígitos desde la derecha hacia la izquierda. Cuando la suma de dos dígitos arroja un resultado con más de un dígito, el dígito de la izquierda se pasa para ser sumado con la pareja que sigue a continuación, es decir, la de la izquierda. A este dígito se le conoce como un *acarreo* o un *carry*. Con los números binarios el procedimiento es el mismo. Se suman las parejas de dígitos, una a una. Cuando el resultado es 10, ó 11, el 1 de la izquierda se convierte en un acarreo que se acumula a la siguiente suma a la izquierda. Cuando el resultado es 0, ó 1, no hay acarreo. Observe que la única forma para que la suma de los dígitos sea 11 es que se esté sumando un acarreo anterior.

Analicemos el procedimiento de suma en binario con un ejemplo. Supongamos que se quiere sumar los números  $1111_2$  y  $0110_2$ , es decir,  $15_{10}$  y  $6_{10}$ . El Cuadro 2 muestra los acarreos involucrados en esta operación. Observe que se agregó un 0 a la izquierda a cada operando, para preveer que la cantidad de dígitos del resultado sea mayor a la de los operandos, como ocurrió en este caso. Sin embargo, un 0 a la izquierda de un número no es significativo.

### Cantidad de bits y cantidad de combinaciones

La cantidad de bits que se utiliza para representar un número, determina el valor máximo que el número puede tener. Cuando estamos tratando con números positivos, la relación se obtiene de la cantidad posible de combinaciones que se pueden hacer con los bits.

Si se tiene  $n$  bits, es posible representar  $2^n$  combinaciones distintas con esos bits. Es decir, se pueden formar  $2^n$  números distintos. Por lo tanto, el rango de valores numéricos positivos que se puede representar con  $n$  bits va desde 0 hasta  $2^n - 1$ .

## Representación de enteros negativos

En el sistema decimal, los números negativos se representan de la misma forma que los positivos, agregando un guión al principio del número. De esta forma, 5 representa el número *cinco*, mientras que  $-5$  representa el *menos cinco*. En base dos podría hacerse lo mismo. Así, dado que el número *cinco* se representa como 101, entonces el *menos cinco* podría ser  $-101$ . Sin embargo, recordemos que el computador sólo puede almacenar ceros y unos en su memoria, por lo que no tenemos posibilidad de guardar el guión. Por lo tanto, es necesario utilizar otros esquemas de representación para los números negativos.

## Signo y magnitud

El esquema simple recién descrito — basado en un guión inicial — se puede implementar con unos y ceros. Para eso, es necesario conocer de antemano la cantidad de bits que se utilizarán para representar un número entero, y esa cantidad debe ser constante. El primero de esos bits, es decir, el de más a la izquierda, se utilizará para representar el signo — el guión. Puede utilizarse un uno en esa posición para denotar los números negativos, mientras que los positivos siempre van a tener un cero. El esquema inverso también es válido, pero debe ser constante.

Por ejemplo, supongamos que contamos con 8 bits para representar los números enteros. Sabemos que con 8 bits es posible representar  $2^8 = 256$  números distintos. Si esos números fueran todos positivos, podríamos representar valores que van desde 0 hasta 255. Si queremos utilizar una representación con signo y magnitud, para poder manipular números negativos, entonces el rango de valores varía desde  $-127$  hasta  $127$ . El bit de más a la izquierda es un uno en los números negativos, y es un cero en los positivos. Los otros siete bits se utilizan para representar la magnitud del número, utilizando el sistema binario estudiado anteriormente. Así, el número  $00000101_2$  representa el  $5_{10}$ , mientras que el número  $10000101_2$  representa el  $-5_{10}$ . Observe que la única diferencia se da en el bit de signo — el primero — mientras que la magnitud — representada en los restantes siete bits — es la misma.

Con signo y magnitud es posible encontrar dos representaciones para el cero, una positiva y una negativa, lo que constituye una ambigüedad a la que hay que prestarle una especial atención.

Este esquema es muy simple de comprender para nosotros, pero la implementación en el *hardware* de las operaciones asociadas, es muy ineficiente. Por esta razón, la representación utilizando signo y magnitud muy rara vez se encuentra en un computador.

## Complemento a 1 y a 2

El esquema de representación que más se utiliza para los números negativos es el de *complemento a 2*, principalmente porque las operaciones aritméticas en el *hardware* se pueden implementar muy fácilmente. Sin embargo, es más conveniente estudiar primero un esquema similar, conocido como *complemento a 1*.

En el esquema de complemento a 1, los números positivos se representan utilizando el sistema binario, como hemos estudiado hasta ahora. Sin embargo, para obtener el inverso de un número positivo, es decir, el negativo que le corresponde, es necesario llevar a cabo una operación simple con los bits. Cada bit debe ser invertido, de modo que los unos se conviertan en ceros y viceversa. Por ejemplo, el inverso de  $00000101_2 = 5_{10}$  es  $11111010_2 = -5_{10}$ .

Todos los números negativos van a tener un 1 en el bit de más a la izquierda, también conocido como el *bit más significativo*, o simplemente *MSB*. Sin embargo, no se debe confundir con el bit de signo en una representación con signo y magnitud. Además, con el complemento a 1 también se presenta una ambigüedad, pues existen dos representaciones para el cero:  $00000000_2$  y  $11111111_2$ .

La representación con complemento a 2 es una variación de la que utiliza complemento a 1, con la ventaja de que existe una única representación para el cero. Para obtener el complemento a 2 de un número, basta con encontrar primero el complemento a 1 y después sumar un 1. Al igual que en el caso anterior, los números negativos siempre van a tener un 1 en el *MSB*.

Consideremos el número  $00000101_2 = 5_{10}$ . Para encontrar el complemento a 2 primero debemos obtener el complemento a 1:  $11111010_2$ . Posteriormente, sumamos 1, obteniendo:  $11111011_2 = -5_{10}$ .

La operación que suma 1 puede hacerse siguiendo unas reglas simples:

1. Empiece desde la derecha hacia la izquierda
2. Mientras se encuentre un uno, cámbielo por un cero

3. Cuando se encuentre un cero, cámbielo por un uno y deténgase
4. El resto de los bits hacia la izquierda no sufrirán cambios

Para convertir un número representado con complemento a 2, a su equivalente en el sistema decimal, primero es necesario saber si se trata de un número positivo o negativo. Para esto basta con mirar el bit de más a la izquierda. Cuando se trata de un número positivo, bastará con interpretar sus bits como un número interpretado en binario. Por otra parte, cuando se trata de un número negativo, primero debe encontrarse el complemento a 2, para luego interpretar el número resultante en binario. En este caso, al número resultante se le debe asignar un signo de negativo.

Por ejemplo, supongamos que queremos interpretar el número  $00000101_2$ , el cual sabemos que está representado en complemento a 2. Como el bit de más a la izquierda es un 0, sabemos que se trata de un número positivo, por lo que basta con interpretarlo como cualquier número binario, obteniendo  $5_{10}$ .

Supongamos ahora que queremos interpretar el número  $11111011_2$ , el cual sabemos que está representado en complemento a 2. En este caso, el bit de más a la izquierda es un 1, por lo que podemos deducir que se trata de un número negativo. El primer paso consiste en encontrar el complemento a 2 del número, que sería:  $00000101_2$ . Puesto que esta es la representación binaria de  $5_{10}$ , podemos concluir que el número  $11111011_2$  se debe interpretar como  $-5_{10}$ .

### Observación importante

Ahora que conocemos distintas formas para representar números enteros, debemos detenernos y analizar un aspecto importante.

Para poder interpretar correctamente un dato que se encuentra almacenado en la memoria del computador, debemos saber con toda certeza cuál esquema de representación se está utilizando. Si no se conoce esta información, es imposible poder interpretarlo correctamente.

Consideremos por ejemplo  $11111011$ . Si esto se tratara de un número entero positivo, sería  $251_{10}$ . Por el contrario, si se tratara de un número entero, representado con signo y magnitud, entonces sería  $-123_{10}$ . Si fuera un entero representado con complemento a 1, sería  $-4_{10}$ . También podría ser un entero representado con complemento a 2, en cuyo caso sería  $-5_{10}$ .

Por lo tanto, es de vital importancia conocer el esquema de representación que se está utilizando, para poder interpretar correctamente los datos almacenados.

### Operaciones con números negativos

Un aspecto importante de la representación de números negativos mediante complemento a 2 es que las operaciones aritméticas se llevan a cabo de la manera tradicional, sin tomar en cuenta si los operandos son positivos o negativos.

Consideremos la suma de un número con su complemento a 2. Por ejemplo,  $00000101_2$  con  $11111011_2$ . Si mantenemos constante la cantidad de bits — en este caso 8 — la suma de estos dos números resulta ser cero, desechando el último acarreo que se genera al sumar los bits de más a la izquierda. El desarrollo de esta operación puede verse en el Cuadro 3.

El resto de las operaciones tiene un comportamiento similar. Por ejemplo, podemos multiplicar  $11111011_2 \times 00000010_2$ , es decir,  $-5_{10} \times 2_{10}$ . El Cuadro 4 muestra el desarrollo de la multiplicación.

Una vez más, para mantener el número de bits desechamos el bit de más a la izquierda, por lo que el resultado de la multiplicación sería:  $11110110_2$ , que equivale a  $-10_{10}$ .

$5_{10}$	0	0	0	0	0	1	0	1
$+ -5_{10}$	+	1	1	1	1	1	0	1
	1	0	0	0	0	0	0	0
$0_{10}$	0	0	0	0	0	0	0	0

Cuadro 3: Suma de un número con su complemento a 2, es decir, con su inverso

$-5_{10}$	×	1	1	1	1	1	0	1	1
$\times 2_{10}$		0	0	0	0	0	0	1	0
		0	0	0	0	0	0	0	0
		1	1	1	1	1	0	1	1
$-10_{10}$		1	1	1	1	1	0	1	0

Cuadro 4: Multiplicación por 2 de un número negativo representado en complemento a 2

## Representación de números reales

Los computadores representan los números reales, es decir, aquellos que tienen decimales, de una forma muy similar a como lo hacen las calculadoras científicas. La forma más usual de lograr esto, es seguir un estándar establecido por el *Institute of Electrical and Electronics Engineers*, o simplemente *IEEE*. Sin embargo, por su complejidad, no será tratado en detalle, sino que discutiremos sólo las generalidades de la representación en punto flotante.

Un número en punto flotante tiene 3 componentes:

- Signo
- Mantisa (número positivo)
- Exponente (con signo)

La Fig. 1 muestra los tres componentes que forman un número de punto flotante. El signo determina si el número es positivo o negativo, y corresponde a un único bit al inicio de la representación. Tanto la mantisa como el exponente pueden codificarse de distintas maneras. Sin embargo, la cantidad de bits que se utilizan para cada uno determina el rango de números reales que se podrá representar.

S	Exponente	Mantisa
---	-----------	---------

Figura 1: Componentes de un número real representado con punto flotante

Cuando se usa 2 como la base de la representación — que es lo más común —, el número representado se puede interpretar como:

$$(-1)^S \times Mantisa \times 2^{Exponente}$$

Por ejemplo, si el bit de signo es un uno, el exponente es  $-2$  y la mantisa es 1, el número representado sería:

$$-1 \times 1 \times 2^{-2} = -0,25$$

Los estándares actuales definen dos tipos de números de punto flotante: Los de *precisión simple* y los de *doble precisión*. Según el estándar de la *IEEE*, los números de precisión simple utilizan 24 bits para la mantisa y 8 bits para el exponente, por lo que este último puede variar desde  $-126$  hasta  $127$ . Por otra parte, los de doble precisión utilizan 53 bits para la mantisa y 11 para el exponente, por lo que este último puede variar en el rango de  $-1022$  a  $1023$ .

## Representación de caracteres

Los caracteres son los símbolos que manipula el computador para representar letras, dígitos y otros elementos de puntuación o con significados específicos. Básicamente, todos los símbolos que aparecen en el teclado del computador son caracteres, pero también hay otros caracteres que no aparecen en el teclado.

Originalmente existían dos estándares de codificación para representar los caracteres en un computador. Estos códigos asignan un número entero con cada caracter, el cual después puede representarse en notación binaria. El primero de ellos, el *EBCDIC*, toma su nombre de las siglas de *Extended Binary Coded Decimal Interchange Code*, y era el esquema de codificación utilizado en los primeros *mainframes* que fabricaba IBM. El *EBCDIC* utiliza 8 bits para codificar los caracteres. Su uso es muy limitado en la actualidad.

El segundo estándar, que todavía se utiliza en la actualidad es el *ASCII*, que toma su nombre de las siglas de *American Standard Code for Information Interchange*. Originalmente, el código utilizaba sólo 7 bits, lo que le permitía representar 128 caracteres distintos. En la actualidad, el código *ASCII* utiliza 8 bits, lo que le permite representar 256 caracteres distintos, incluyendo símbolos de lenguajes distintos al Inglés <sup>1</sup>.

Algunos de los caracteres *ASCII* no son visibles, y se utilizan para controlar dispositivos de entrada y salida, como *modems*, *terminales* e *impresoras*. Las letras minúsculas y mayúsculas — sin considerar las vocales con acento — así como los dígitos decimales, se encuentran consecutivos en el código, por lo que el código de una letra es muy similar al de su sucesora. El Cuadro 5 muestra un subconjunto del código *ASCII*.

Más recientemente, la *International Standard Organization*, o simplemente *ISO*, estableció un estándar para la representación universal de caracteres, llamado *Unicode*. La idea es que el nuevo código permita representar símbolos provenientes de cualquier lenguaje, así como su correcta comunicación a través de Internet, por ejemplo, en mensajes de correo electrónico. El *Unicode* utiliza 16 bits para representar hasta 65536 caracteres distintos. Sus códigos se expresan normalmente como 4 dígitos hexadecimales.

Los primeros 128 códigos del *Unicode* coinciden con los 128 códigos que asigna el *ASCII* de 7 bits. A este subconjunto se le conoce como el *Basic Latin*, y abarca los códigos que van desde el  $0000_{16}$  hasta el  $007F_{16}$ . Posteriormente existen el *Latin-1* ( $0080_{16}$  al  $00FF_{16}$ ) y el *Latin Extended* ( $0100_{16}$  al  $024F_{16}$ ). Para denotar que el código de un caracter se está expresando según el *Unicode*, se agregan los símbolos U+ antes del código hexadecimal. Por ejemplo, U + 0041, para representar la letra A.

El Cuadro 6 compara los códigos decimales asignados a diversos caracteres, utilizando distintos esquemas de codificación: *ASCII* de 8 bits, *EBCDIC* y *Unicode*.

En el sitio web <http://www.unicode.org/charts> puede encontrarse los distintos códigos asociados con el *Unicode*.

---

<sup>1</sup>El código *ASCII* original, de 7 bits, no podía representar, por ejemplo, las vocales con acento o la letra ñ.

Binario	Decimal	Caracter	Binario	Decimal	Caracter	Binario	Decimal	Caracter
00100001	033	!	01000010	066	B	01100110	102	f
00100010	034	”	01000011	067	C	01100111	103	g
00100011	035	#	01000100	068	D	01101000	104	h
00100100	036	\$	01000101	069	E	01101001	105	i
00100101	037	%	01000110	070	F	01101010	106	j
00100110	038	&	01000111	071	G	01101011	107	k
00100111	039	,	01001000	072	H	01101100	108	l
00101000	040	(	01001001	073	I	01101101	109	m
00101001	041	)	01001010	074	J	01101110	110	n
00101010	042	*	01001011	075	K	01101111	111	o
00101011	043	+	01001100	076	L	01110000	112	p
00101100	044	,	01001101	077	M	01110001	113	q
00101101	045	-	01001110	078	N	01110010	114	r
00101110	046	.	01001111	079	O	01110011	115	s
00101111	047	/	01010000	080	P	01110100	116	t
00110000	048	0	01010001	081	Q	01110101	117	u
00110001	049	1	01010010	082	R	01110110	118	v
00110010	050	2	01010011	083	S	01110111	119	w
00110011	051	3	01010100	084	T	01111000	120	x
00110100	052	4	01010101	085	U	01111001	121	y
00110101	053	5	01010110	086	V	01111010	122	z
00110110	054	6	01010111	087	W	01111011	123	{
00110111	055	7	01011000	088	X	01111101	125	}
00111000	056	8	01011001	089	Y	01111110	126	~
00111001	057	9	01011010	090	Z	10000010	130	é
00111010	058	:	01011011	091	[	10010000	144	É
00111011	059	;	01011101	093	]	10100000	160	á
00111100	060	<	01011111	095	-	10100001	161	í
00111101	061	=	01100001	097	a	10100010	162	ó
00111110	062	>	01100010	098	b	10100011	163	ú
00111111	063	?	01100011	099	c	10100100	164	ñ
01000000	064	@	01100100	100	d	10100101	165	Ñ
01000001	065	A	01100101	101	e	10101000	168	¿

Cuadro 5: Fragmento de la tabla ASCII de 8 bits

Caracter	ASCII (8 bits)	EBCDIC	Unicode
0	48	240	48
A	65	193	65
a	97	129	97
\$	36	91	36
á	160		225
ñ	164		241

Cuadro 6: Representación de caracteres según distintos estándares de codificación

## Representación de otros tipos de datos

Existe una enorme diversidad de tipos de datos que pueden ser manipulados en un computador. Para cada uno de ellos, debe existir un estándar o código que permita su representación mediante unos y ceros. En esta sección mencionaremos algunos estándares para tipos de datos de uso común, como las imágenes, el audio y el video.

### Representación de imágenes

La calidad de una imagen representada en un computador está determinada por la *resolución* con la que fue digitalizada. La digitalización es el proceso mediante el cual una imagen es transformada en información numérica. Básicamente, la imagen se representa en una matriz de puntos, llamados *pixels*, manipulables en forma independiente. En una imagen a color, los pixels tienen información de color. En una imagen en blanco y negro, los pixels pueden estar únicamente encendidos o apagados.

Una misma imagen puede digitalizarse en distintas resoluciones. Por ejemplo, una fotografía de  $10 \times 5$  cms. podría digitalizarse en una matriz de  $512 \times 512$  pixels, o en una de  $1024 \times 1024$  pixels. En el segundo caso, existen más pixels por cada centímetro cuadrado de la fotografía, por lo que la calidad de la imagen digitalizada será mayor.

La forma más simple para representar una imagen es mediante un *mapa de bits*, también conocido como *bitmap*. En un bitmap, se asigna una cantidad constante de bits a cada pixel de la imagen, y en esos bits se almacena la información de color, o de tonalidades de gris, que representan al pixel. En el caso de imágenes en blanco y negro, basta con asociar un bit con cada pixel. Así, una imagen de  $1024 \times 1024$  pixels, en blanco y negro, puede representarse utilizando  $2^{20}$  bits, lo que equivale a 128 Kbytes.

Con las imágenes a color el tamaño de los bitmaps aumenta, dependiendo de la cantidad de colores distintos que se quiera manipular, lo que se conoce como la *paleta de colores*. Por ejemplo, si se utilizan 256 colores distintos, cada pixel requiere de 8 bits para representar su color. Así, una imagen de  $1024 \times 1024$  pixels, con una paleta de 256 colores, requiere de  $2^{23}$  bits, es decir, 1 Mbyte. La paleta de colores preferida es la de 24 bits, que puede formar alrededor de 16 millones de colores diferentes. A este modo de representación se le conoce como *color verdadero*, o *true color*. En esta paleta se utilizan 8 bits para representar una tonalidad de cada uno de los tres colores *básicos*, con los que se forma el color en un computador: Rojo, verde y azul <sup>2</sup>. En este caso, la imagen de  $1024 \times 1024$  pixels requiere de  $3 \times 2^{23}$  bits, es decir, 3 Mbytes.

La enorme cantidad de espacio que requieren las imágenes representadas como bitmaps ha llevado al desarrollo de técnicas de *compresión*. Los algoritmos más populares de compresión de datos, como por ejemplo el conocido como *zip*, pertenecen a una categoría llamada *compresión sin pérdida de información*. Esto quiere decir que la información comprimida, una vez recuperada, es exactamente la original. Estos algoritmos son muy útiles para compresión de caracteres, documentos, etc., pero no logra buenas tasas de compresión con las imágenes.

Para la compresión de imágenes se acostumbra utilizar *algoritmos de compresión con pérdida de información*. Estos algoritmos desechan información que no es útil, con el objetivo de reducir el tamaño de las imágenes. A pesar de la pérdida de información, por lo general los cambios no son perceptibles por los sentidos humanos. La idea es explotar el hecho de que el ojo humano no percibe con exactitud ligeros cambios en el color, pero sí lo hace ante cambios en el brillo.

Uno de los formatos de representación de imágenes, con compresión con pérdida de información, lo constituye el *jpeg*, que toma su nombre del grupo de expertos que lo diseñó: *Joint Photographic Experts Group*. Este formato funciona muy bien con imágenes a color o en escala de grises, principalmente en escenas del mundo

---

<sup>2</sup>Los dispositivos que utilizan estos colores básicos para formar color reciben el nombre de *RGB*, por las siglas en Inglés de los tres colores: Red, Green, Blue.

real. Sin embargo, no se recomienda su uso en dibujos generados a base de líneas. El ratio de compresión puede alcanzar 20 : 1, pudiendo comprimir una imagen de 2 MBytes en 100 KBytes. Su funcionamiento se basa en el registro de las variaciones existentes entre pixels cercanos.

Otro formato que es muy popular es el *gif*, el cual es superior al *jpeg* cuando se tratan imágenes que no son del mundo real, por ejemplo, figuras o gráficos generados por computador, así como imágenes en blanco y negro. También puede encontrarse formatos como el *tiff* y el *eps*.

## Representación de sonido

La digitalización de sonidos, es decir, la traducción de ondas a números, se lleva a cabo utilizando *transformadas de Fourier*. Dada la gran cantidad de información que resulta es necesario, en primera instancia, eliminar aquellos sonidos que están fuera del rango de frecuencias que el oído humano puede percibir. Aún así, la cantidad de datos es muy grande, incluso para unos pocos segundos de sonido.

Al igual que para el caso de las imágenes, existen estándares de compresión que se basan en la eliminación de información redundante. Este es el caso del formato *mpeg*, que corresponde a la versión para audio del *jpeg*. Su nombre proviene de las siglas de *Moving Picture Experts Group* y no se limita al almacenamiento de audio. En realidad, el *mpeg* comprende una diversidad de estándares para la representación de audio y video. En la actualidad se utiliza el *mpeg* de nivel 4, conocido como *MPEG-4*, a partir del cual han surgido formatos muy populares como el *DivX*.

En la Internet, los formatos más populares para la transmisión de audio son *RealAudio*, un estándar propietario de una empresa comercial, y *WMA*, el estándar de Microsoft.

## Representación de video

El video, así como el cine y la televisión, se basa en una serie de cuadros que se pasan rápidamente frente a la vista del espectador. El video es uno de los tipos de datos que requieren mayor cantidad de espacio para su representación. Por esta razón, la compresión es un factor determinante. Incluso los formatos comerciales, como el *DVD*, existen gracias a complicados esquemas de compresión. La compresión de video puede llevarse a cabo dentro de cada cuadro — registrando diferencias dentro del cuadro — o entre cuadros — registrando diferencias entre un cuadro y el siguiente.

Dentro del ámbito de los computadores, el formato *MOV* se originó en el mundo de los computadores *Macintosh* de Apple, aunque actualmente se utiliza en todo tipo de computadores. Está basado en *MPEG-4* y es utilizado por conocidos programas como el *Quicktime*. En esta gama de formatos se encuentra también el formato *RealVideo*, que es bastante popular para transferir video por la Internet, junto con el *Windows Media* de Microsoft.

El formato *AVI* fue creado por *Microsoft*, para manipular video dentro de su sistema operativo *Windows*, e incluye niveles muy bajos de compresión de datos. Su nombre proviene de las siglas de *Audio Video Interleave*. Este formato permite representar video compuesto de cuadros de hasta  $160 \times 120$  pixels, con una tasa de refrescamiento de 15 cuadros por segundo <sup>3</sup>.

---

<sup>3</sup>La tasa de refrescamiento más propicia para el ojo humano se encuentra alrededor de los 30 cuadros por segundo