Universidad Nacional de Ingeniería
Facultad de Ciencias

Introducción a la Ciencia de la Computación

Almacenamiento de Datos

Prof: J. Solano
2011-I

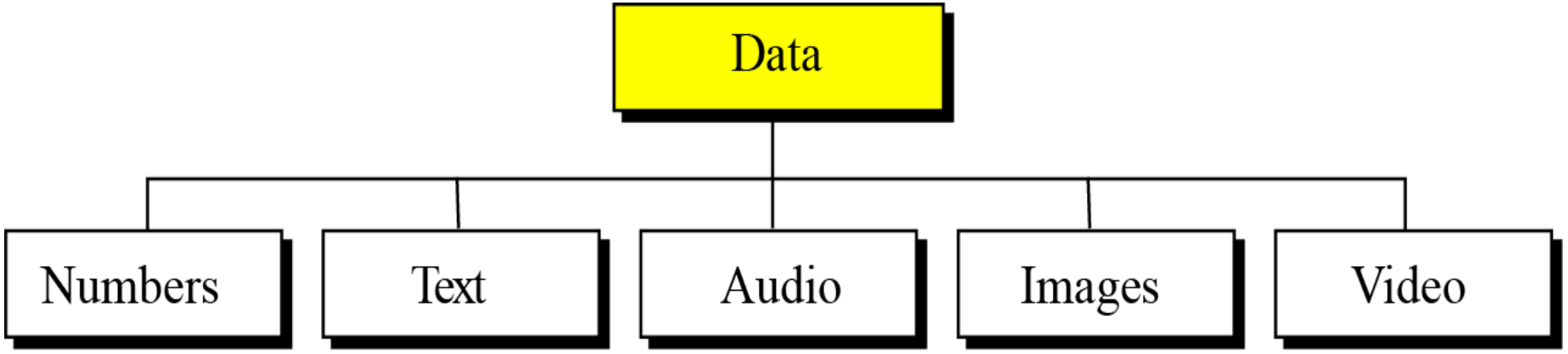
Objetivos

Despues de estudiar este capítulo el estudiante será capaz de:

- **Listar cinco diferentes tipos de datos usados en un computador.**
- **Describir como diferentes datos son almacenados dentro de un computador.**
- **Describir como enteros son almacenados dentro de un computador.**
- **Describir como reales son almacenados en un computador.**
- **Describir como audio es almacenado en un computador usando uno de los varios sistemas de codificación.**
- **Describir como texto es almacenado en un computador usando muestreo, cuantificación y codificación.**
- **Describir como las imágenes son almacenadas en un computador usando regímenes de raster (mapeo de bits) y gráficos vectoriales.**
- **Describir como las imágenes son almacenadas en un computador como una representación de imágenes cambiando en el tiempo.**

INTRODUCCION

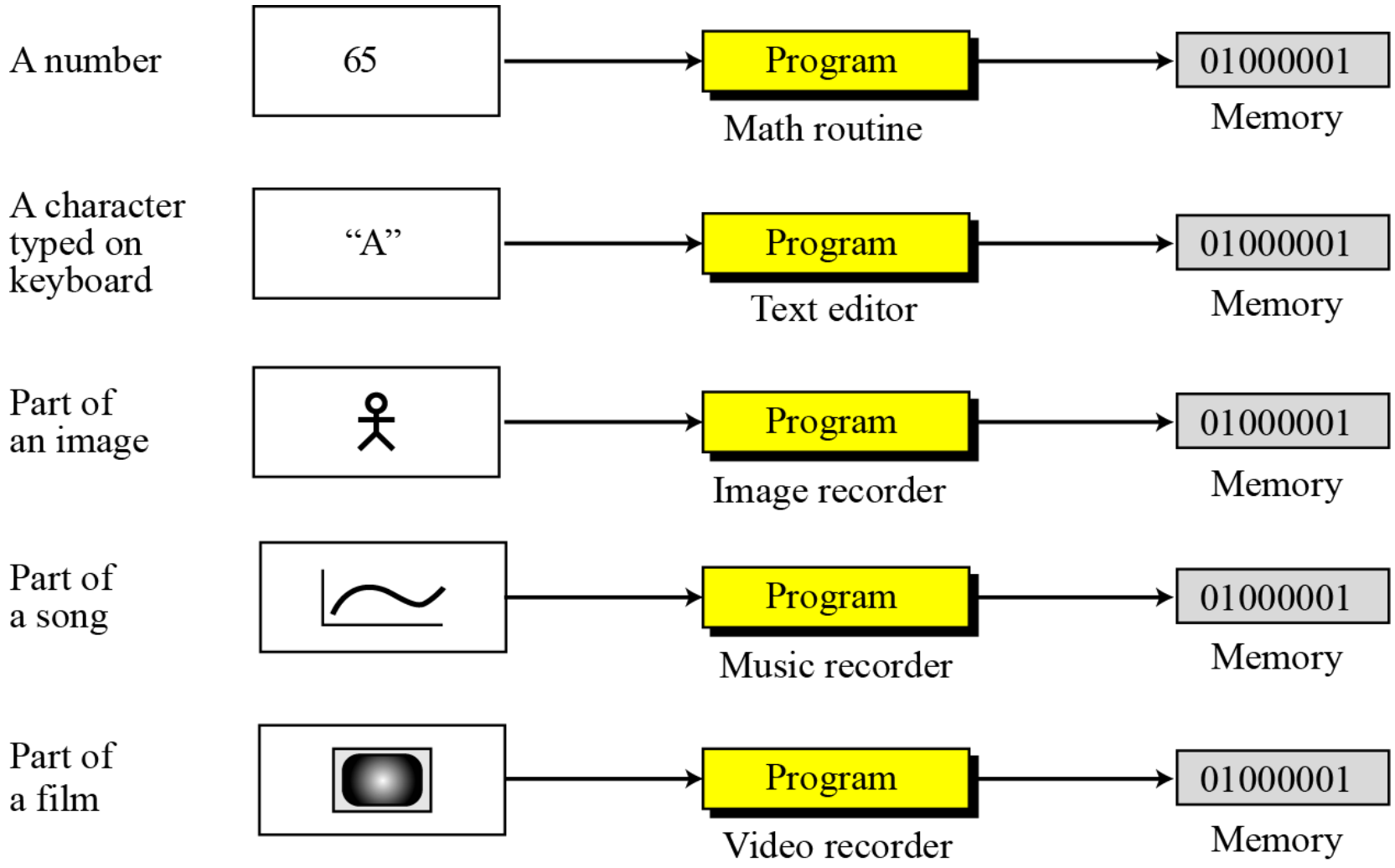
Los datos hoy dia vienen en diferentes formas incluyendo números, texto, audio, imágenes y video.



La industria de computación usa el término "multimedia" para definir información que contiene números, texto, imágenes y video.

Datos dentro del computador

Todos los tipos de datos se transforman en una representación uniforme cuando se almacenan en un computador y se transforma de nuevo a su forma original cuando se recuperan. Esta representación universal se llama un patrón de bits (**bit pattern**).



Almacenamiento de diferentes tipos de datos

Compresión de datos

Para ocupar menos espacio de memoria, los datos son normalmente comprimidos antes de ser almacenados en el ordenador. La compresión de datos es un tema muy amplio y complicado.

Compresión de datos

Deteccion y correccion de errores

Otro tema relacionado a los datos es la detección y corrección de errores durante la transmisión o el almacenamiento.

Detección y corrección de errores

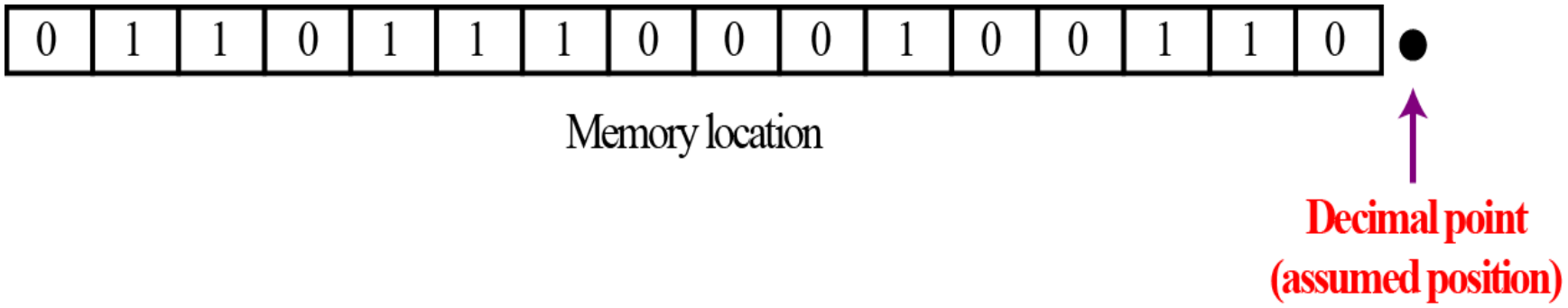
ALMACENANDO NUMEROS

Un **número** se cambia al sistema binario antes de ser almacenados en la memoria del ordenador, como se describe en la clase anterior. Sin embargo, aún hay dos cuestiones que deben manejarse:

1- Como al macenar el signo de un número
2- Como mostrar el punto decimal

Almacenando enteros

Los enteros son números sin parte fraccionaria. Por ejemplo, 134 y -125 son enteros, mientras que 134.23 y -0.235 no lo son. Un entero puede ser considerado como un número en el que se fija la posición del punto decimal: el punto decimal esta a la derecha del bit menos significativo (más a la derecha). Por esta razón, la representación de punto fijo se utiliza para almacenar números enteros. En esta representación el punto decimal se supone pero no se almacena.



Representación de punto fijo para enteros

Un entero es normalmente almacenado en memoria usando representación de punto fijo

Representación sin signo (unsigned)

Un entero sin signo es un entero que nunca puede ser negativo y sólo puede tomar valores de 0 o positivo. Su rango es de 0 a infinito positivo.

Un dispositivo de entrada almacena un entero sin signo mediante los pasos siguientes:

1- El entero se cambia a binario.
2- Si el número de bits es menor que n, 0s se añaden a la izquierda.

Ejemplo 1

Almacenar 7 en una ubicación de memoria de 8 bits usando representación sin signo

Solucion

Primero cambie el número entero a binario, $(111)_2$. Agregue cinco 0s para hacer un total de ocho bits $(00000111)_2$. El entero se almacena en la ubicación de memoria. Tenga en cuenta que el subíndice 2 se utiliza para enfatizar que el entero es binario, pero el subíndice no se almacena en el ordenador.

Change 7 to binary	→						1	1	1
Add five bits at the left	→	0	0	0	0	0	1	1	1

Ejemplo 2

Almacenar 258 en una ubicación de memoria de 16 bits.

Solucion

Cambie el entero a binario, $(100000010)_2$. Agregue siete 0s para hacer un total de dieciseis bits $(0000000100000010)_2$. El entero se almacena en la ubicación de memoria.

Change 258 to binary	→								1	0	0	0	0	0	1	0
Add seven bits at the left	→	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

Ejemplo 3

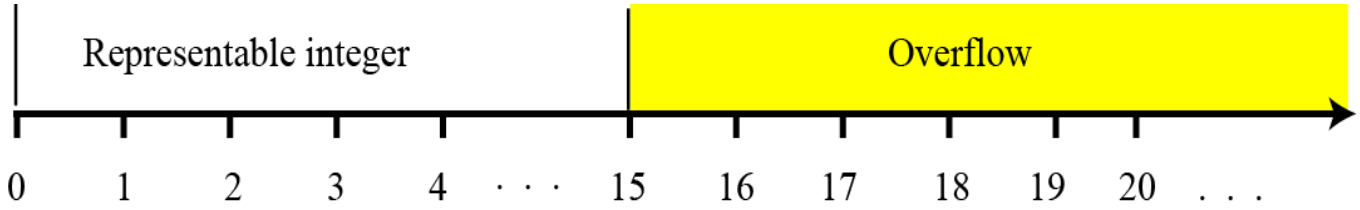
Que es retornado de un dispositivo de salida cuando se recupera un patrón de bits 00101011 almacenado en memoria como un entero sin signo (unsigned)?

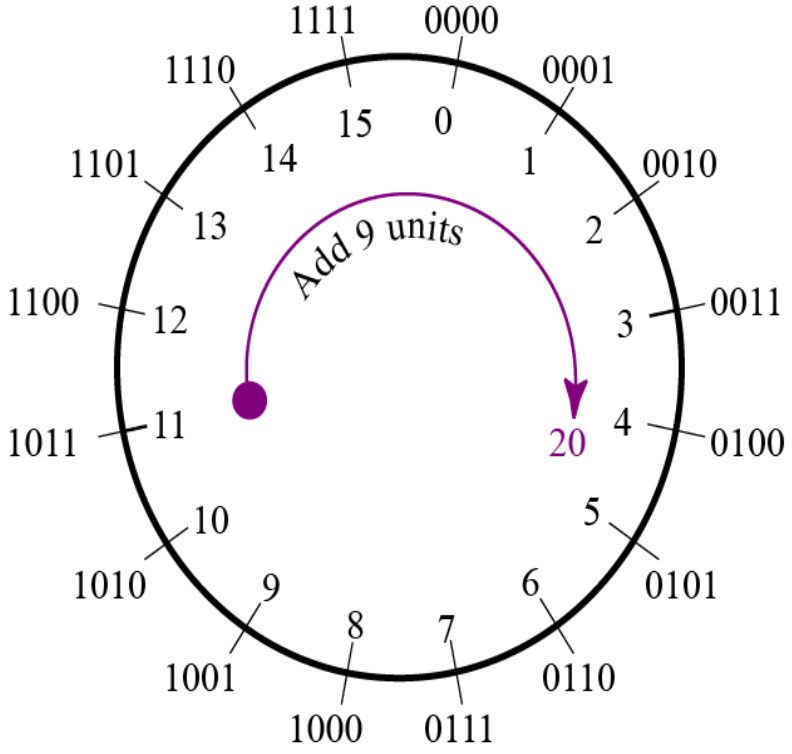
Solución

Usando el procedimiento de la clase anterior, el binario es convertido al entero sin signo 43.

Overflow de un entero sin signo

La figura muestra lo que sucede si se intenta almacenar un número entero que es mayor de $2^4 - 1 = 15$ en una ubicación de memoria que sólo puede contener cuatro bits





Representación magnitud y signo (sign-and-magnitude)

En este método, el rango disponible para los enteros sin signo (0 a 2^n - 1) se divide en dos sub-rangos iguales. La primera mitad representa enteros positivos, la segunda mitad, enteros negativos

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	–0	–1	–2	–3	–4	–5	–6	–7

En representatión de magnitud y signo, el bit mas a la izquierda define el signo del entero. Si es 0, el entero es positivo. Si es 1, el entero es negativo.

Ejemplo 4

Almacenar +28 en una ubicación de memoria de 8 bits usando representación sign-and-magnitude

Solucion

El entero es cambiado a binario de 7 bits. Al bit mas a la izquierda se le asigna 0. El numero de 8 bits es almacenado.

Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	**0**	0	0	1	1	1	0	0

Ejemplo 5

Almacenar -28 en una ubicación de memoria de 8 bits usando representación sign-and-magnitude

Solución

El entero es cambiado a binario de 7 bits. Al bit mas a la izquierda se le asigna 1. El número de 8 bits es almacenado.

Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	1	0	0	1	1	1	0	0

Ejemplo 6

Recuperar el entero que es almacenado como 01001101 en representación sign-and-magnitude

Solución

Desde que el bit mas a la izquierda es 0, el signo es positivo. El resto de los bits (1001101) es cambiado a decimal como 77. Despues de añadir el signo, el entero es +77.

Ejemplo 7

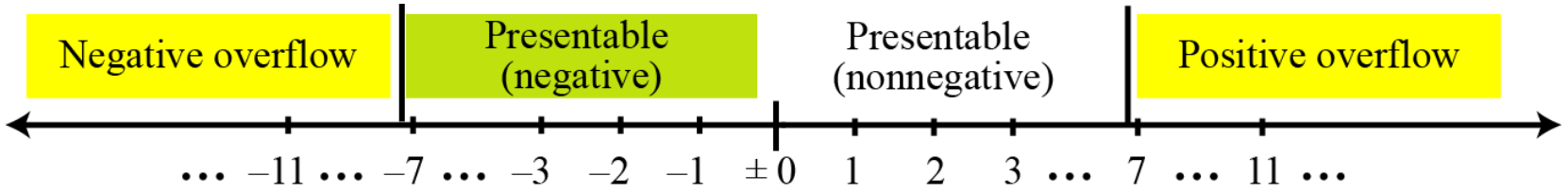
Recuperar el entero que es almacenado como 10100001 en representación sign-and-magnitude

Solución

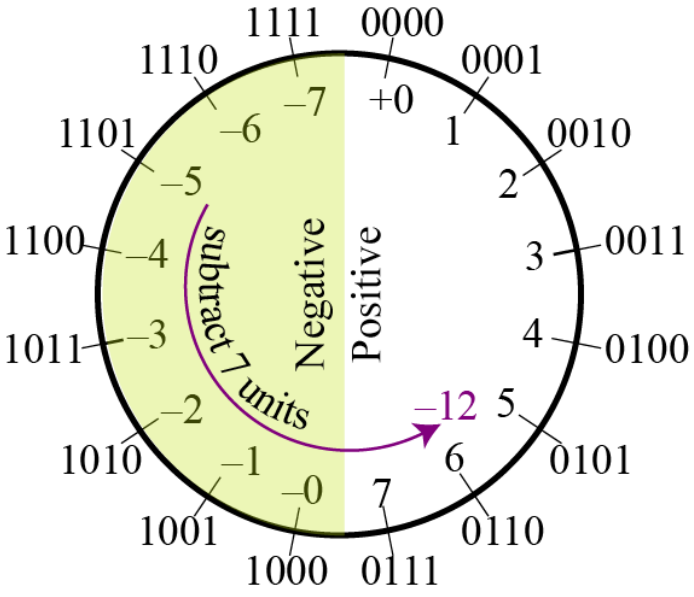
Desde que el bit mas a la izquierda es 1, el signo es negativo. El resto de los bits (0100001) es cambiado a decimal como 33. Después de añadir el signo, el entero es -33.

Overflow en representacion sign-and-magnitude

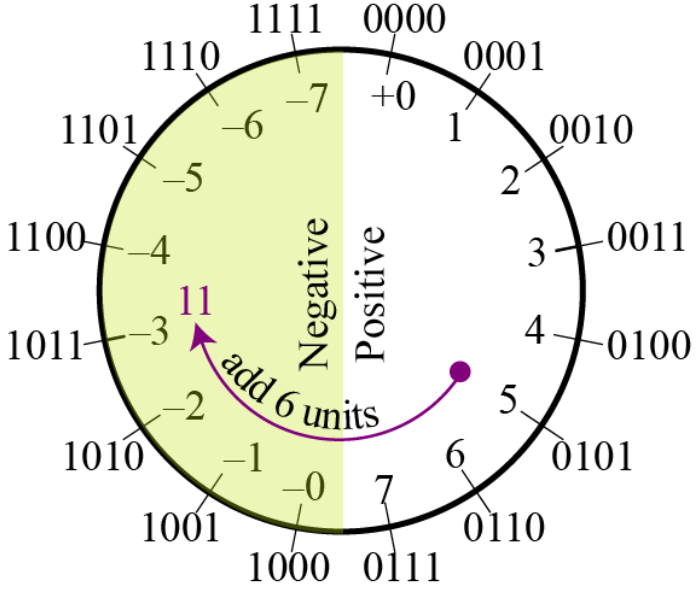
La figura muestra overflow postivo y negativo cuando almacenamos un entero en representacion sign-and-magnitude usando una ubicación de memoria de 4 bits.



a. Linear characteristic of an integer in sign-and-magnitude format



b. Negative overflow

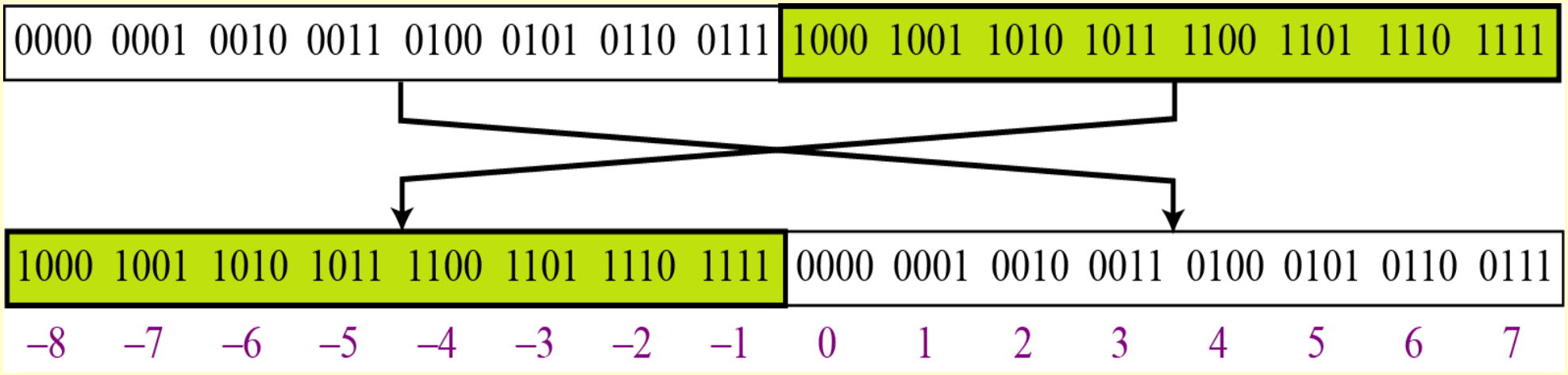


c. Positive overflow

Representación de complemento a dos (two's complement)

Casi todos los computadores usan la representación de complemento a dos para almacenar un entero con signo en una ubicación de memoria de n bits. En este método, el rango disponible para un número entero sin signo de (0 a 2^n - 1) se divide en dos sub-rangos iguales. El primer sub-rango se utiliza para representar números enteros no negativos, la segunda mitad para representar enteros negativos. A los patrones de bits se le asignan enteros negativos y no negativos (cero y positivo).

Representación de two's complement



En representación two's complement, el bit mas la izquierda define el signo del entero. Si es 0, el entero es positivo. Si es 1, el entero es negativo.

Complemento de uno (one's complementing)

Antes de discutir más esta representación, tenemos que introducir dos operaciones. El primero se llama ***one's complementing*** o tomando el one's complement (complemento a uno) de un número entero. La operación se puede aplicar a cualquier número entero, positivo o negativo. Esta operación simplemente invierte (flips) cada bit. Un bit 0 se cambia a un bit 1, un bit 1 se cambia a un bit 0.

Ejemplo 8

Lo siguiente muestra como tomamos el one's complement de un entero 00110110.

Original pattern	0	0	1	1	0	1	1	0
After applying one's complement operation	1	1	0	0	1	0	0	1

Ejemplo 9

Aqui se muestra como se obtiene el entero original si aplicamos la operación one's complement dos veces.

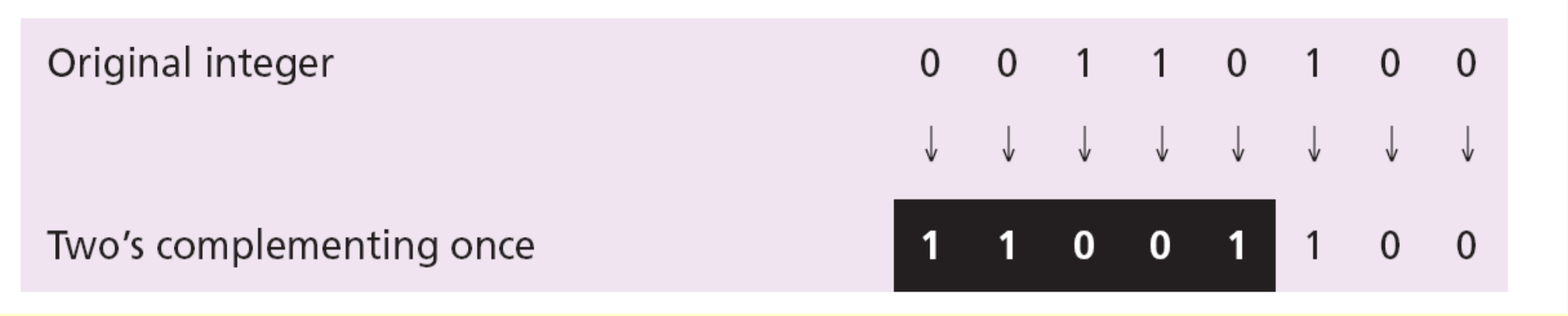
Original pattern	0	0	1	1	0	1	1	0
One's complementing once	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**
One's complementing twice	0	0	1	1	0	1	1	0

Two's Complementing

La segunda operación se llama ***two's complementing*** o tomar el two's complement de un entero en binario. Esta operacion es hecha en dos pasos. Primero copiamos bits de la derecha hasta que un 1 es copiado; entonces, cambiamos (flip) el resto de los bits.

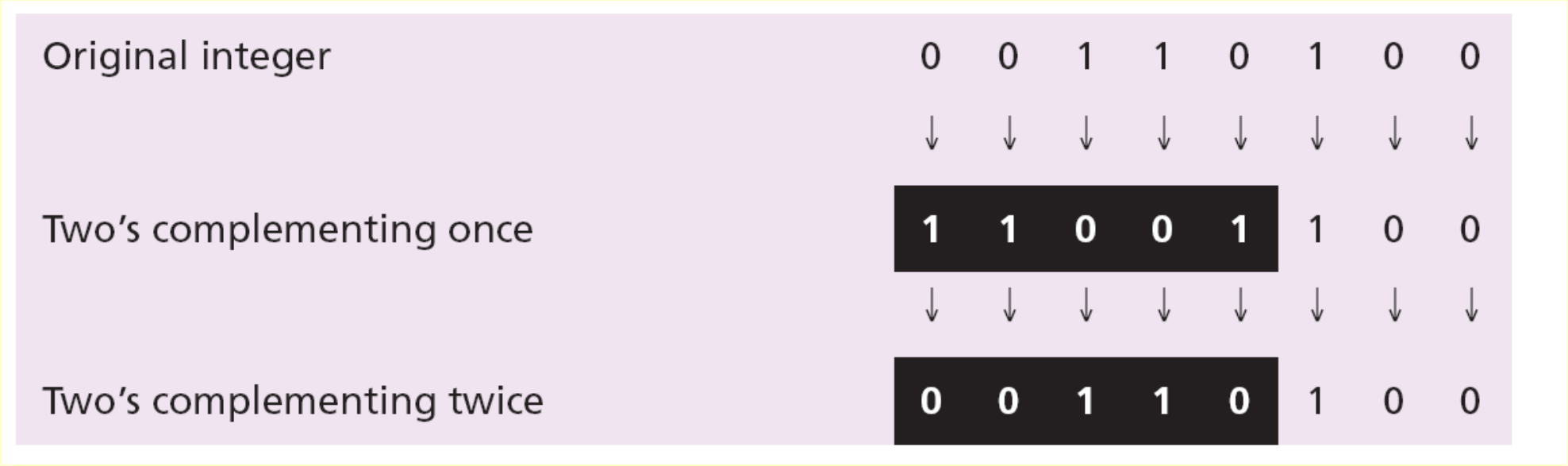
Ejemplo 10

Lo siguiente muestra como tomamos el two's complement de un entero 00110100.



Ejemplo 11

Aqui se muestra que siempre se obtiene el entero original si aplicamos la operación two's complement dos veces.



Un camino alternativo para tomar el two's complement de un entero es tomar primero el one's complement y despues añadir 1 al resultado.

Ejemplo 12

Almacenar el entero 28 en una ubicación de memoria de 8 bits usando representación two's complement.

Solución

El entero es positivo (sin signo significa positivo), entonces después de transformarlo de decimal a binario ninguna otra acción es necesaria. Notar que cinco 0s adicionales son añadidos a la izquierda para completar ocho bits.

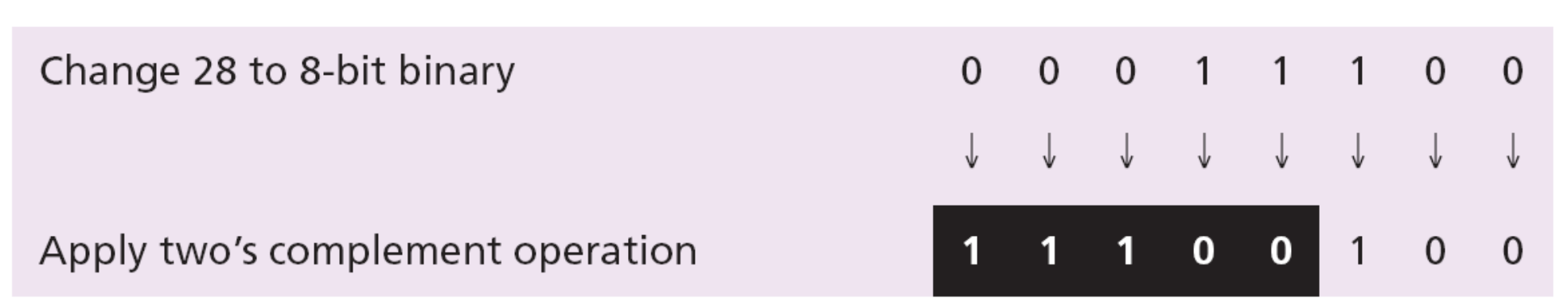
Change 28 to 8-bit binary	0	0	0	1	1	1	0	0

Ejemplo 13

Almacenar el entero -28 en una ubicación de memoria de 8 bits usando representación two's complement.

Solución

El entero es negativo, entonces despues de transformarlo a binario, el computador aplica la operación two's complement al entero.



Ejemplo 14

Recuperar el entero que es almacenado como 00001101 en memoria, en formato two's complement.

Solución

El bit mas a la izquierda es 0, entonces el signo es positivo. El entero es cambiado a decimal y el signo es añadido.

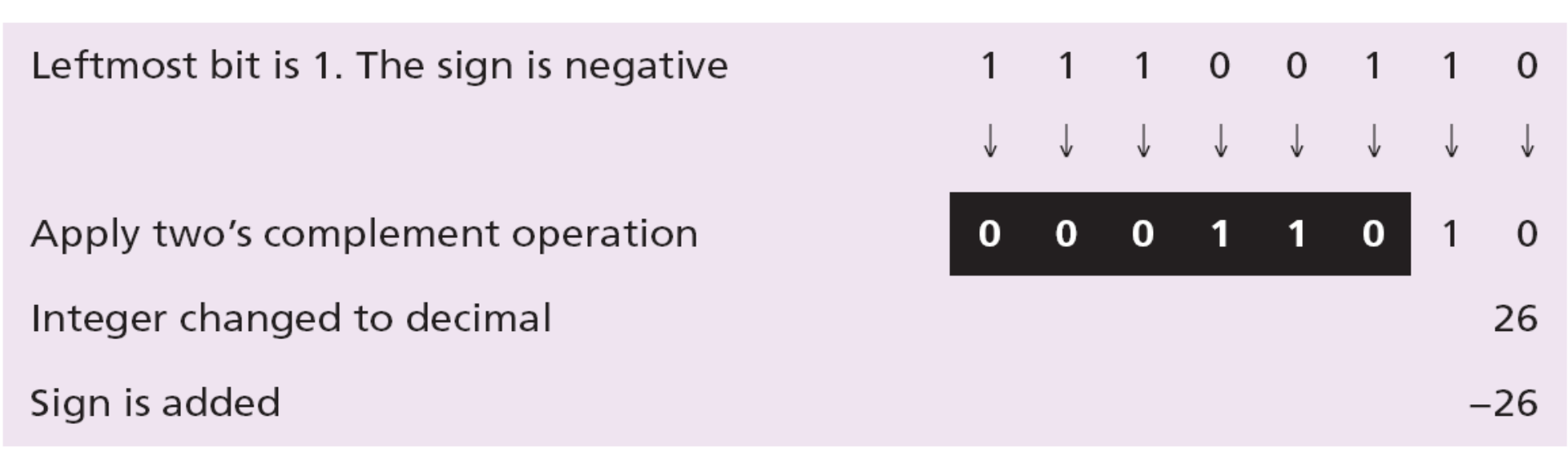
Leftmost bit is 0. The sign is positive	0	0	0	0	1	1	0	1
Integer changed to decimal								13
Sign is added (optional)								+13

Ejemplo 15

Recuperar el entero que es almacenado como 11100110 en memoria, en formato two's complement.

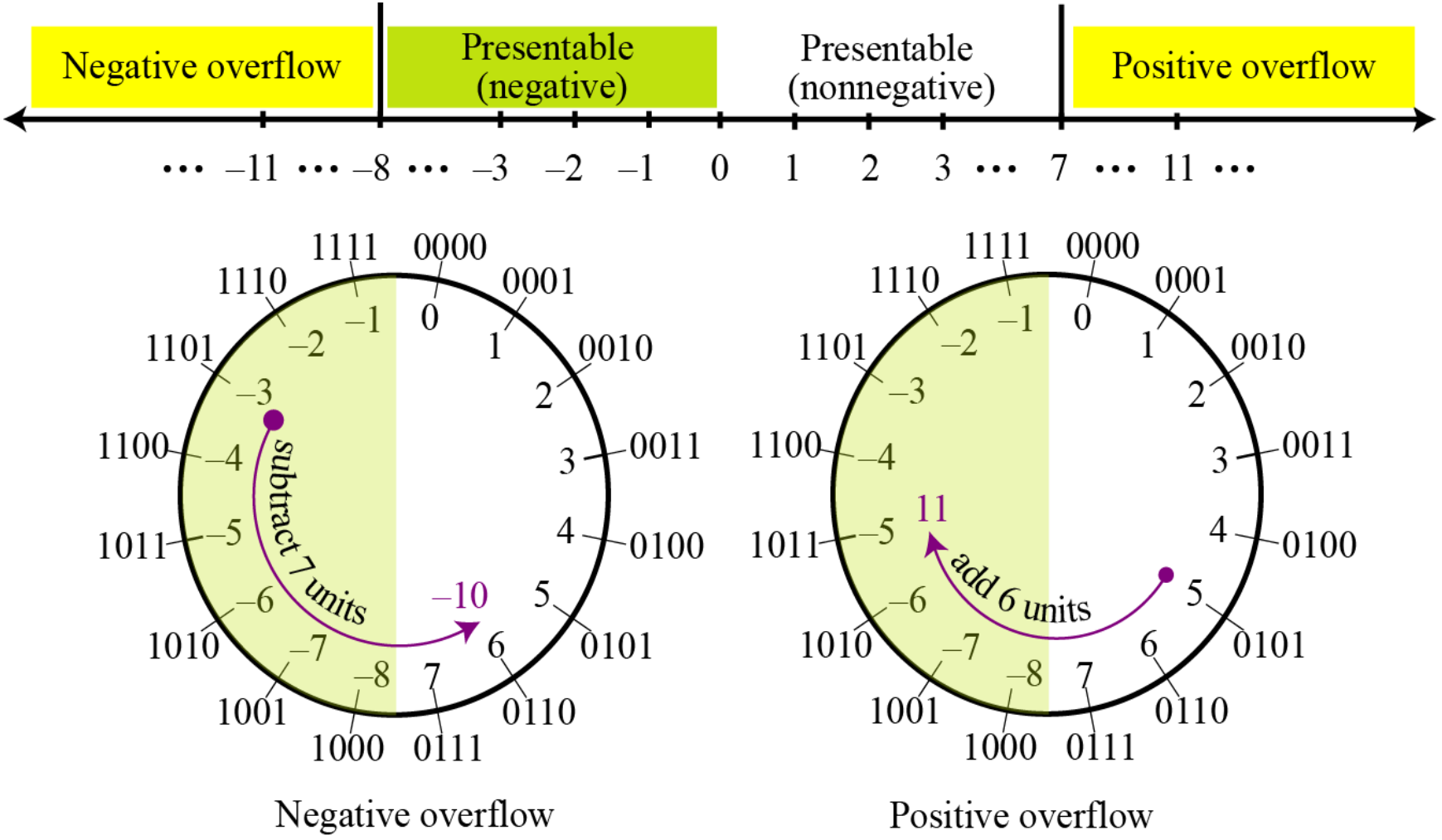
Solucion

El bit mas a la izquierda es 1, entonces el signo es negativo. El entero necesita la operación two's complement antes de ser cambiado a decimal.



Overflow en representación two's complement

Hay solo un cero en notación two’s complement



Comparación

Table 3.1 Summary of integer representations

Contents of memory	*Unsigned*	*Sign-and-magnitude*	*Two's complement*
0000	0	0	+0
0001	1	1	+1
0010	2	2	+2
0011	3	3	+3
0100	4	4	+4
0101	5	5	+5
0110	6	6	+6
0111	7	7	+7
1000	8	–0	–8
1001	9	–1	–7
1010	10	–2	–6
1011	11	–3	–5
1100	12	–4	–4
1101	13	–5	–3
1110	14	–6	–2
1111	15	–7	–1

Almacenando reales

Un real es un número con una parte integral y una parte fraccional. Por ejemplo, el 23,7 es un número real - la parte entera es 23 y la parte fraccionaria es 0.7. Aunque una representación de punto fijo puede ser usada para representar un número real, el resultado puede no ser preciso o puede no tener la precisión requerida. Los siguientes dos ejemplos explican por qué

Números reales con partes enteras grandes o partes fraccionales muy pequeñas no deberían ser almacenadas en representación de punto fijo.

Ejemplo 16

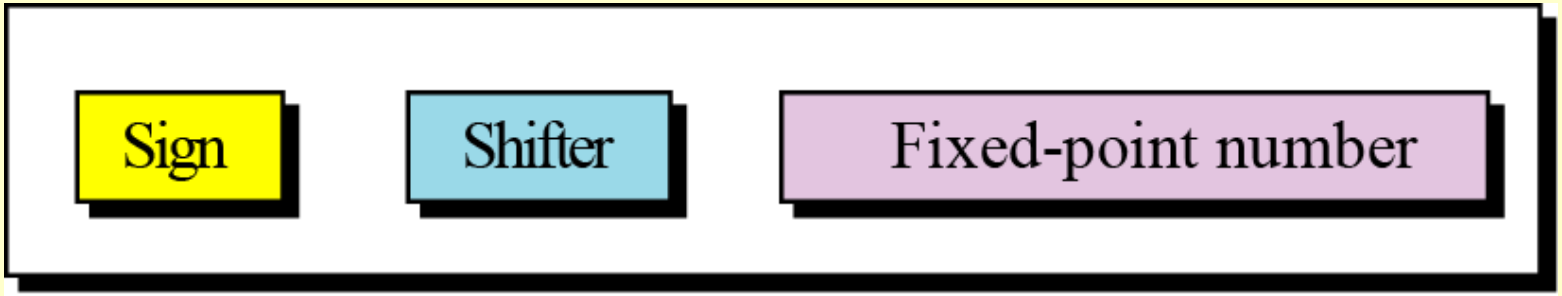
En el sistema decimal, supongamos que usamos una representación de punto fijo con dos dígitos a la derecha del punto decimal y catorce dígitos a la izquierda del punto decimal, para un total de dieciseis dígitos. La precisión de un número real en este sistema se pierde si tratamos de representar un número decimal como 1.00234: el sistema almacena el número como 1,00

Ejemplo 17

En el sistema decimal, asumamos que usamos una representación de punto fijo con seis dígitos a la derecha del punto decimal y diez dígitos de la izquierda del punto decimal, para un total de dieciséis dígitos. La precisión de un número real en este sistema se pierde si tratamos de representar un número decimal como 236,154,302,345.00. El sistema almacena el número como 6,154,302,345.00: la parte entera es mucho menor de lo que debería ser

Representación de punto flotante

La solución para mantener precisión es usar **representación de punto flotante.**



Floating-point pepresentation

Las tres partes de un número real en representación de punto flotante

Una representación de punto flotante de un número es hecha de tres partes: un signo, un shifter (cambiador) y un número de punto fijo.

Ejemplo 18

Aqui se muestra el número decimal

7,452,000,000,000,000,000,000.00

en notación científica (representación de punto flotante).

Actual number	→	+	7,425,000,000,000,000,000,000.00
Scientific notation	→	+	7.425×10^{21}

Las tres secciones son el **signo** (+), el **shifter** (21) y la **parte de punto fijo** (7.425). Notar que el shifter es el exponente.

Ejemplo 19

Mostrar el número

$$-0.0000000000000232$$

en notación científica (representación de punto flotante)

Solución

Utilizamos el mismo criterio que en el ejemplo anterior, movemos el punto decimal tras el dígito 2, como se muestra a continuación:

Actual number	→	–	0.0000000000000232
Scientific notation	→	–	2.32×10^{-14}

Las tres seccions son el **signo** (-), el **shifter** (-14) y la **parte de punto fijo** (2.32). Notar que el shifter es el exponente.

Ejemplo 20

Mostrar el número

$$(10100100000000000000000000000000.00)_2$$

en representación de punto flotante.

Solución

Utilizamos la misma idea, manteniendo solo un dígito a la izquierda del punto decimal.

Actual number	→	+	$(10100100000000000000000000000000.00)_2$
Scientific notation	→	+	1.01001×2^{32}

Ejemplo 21

Mostrar el número

$$-(0.00000000000000000000000101)_2$$

en representación de punto flotante.

Solución

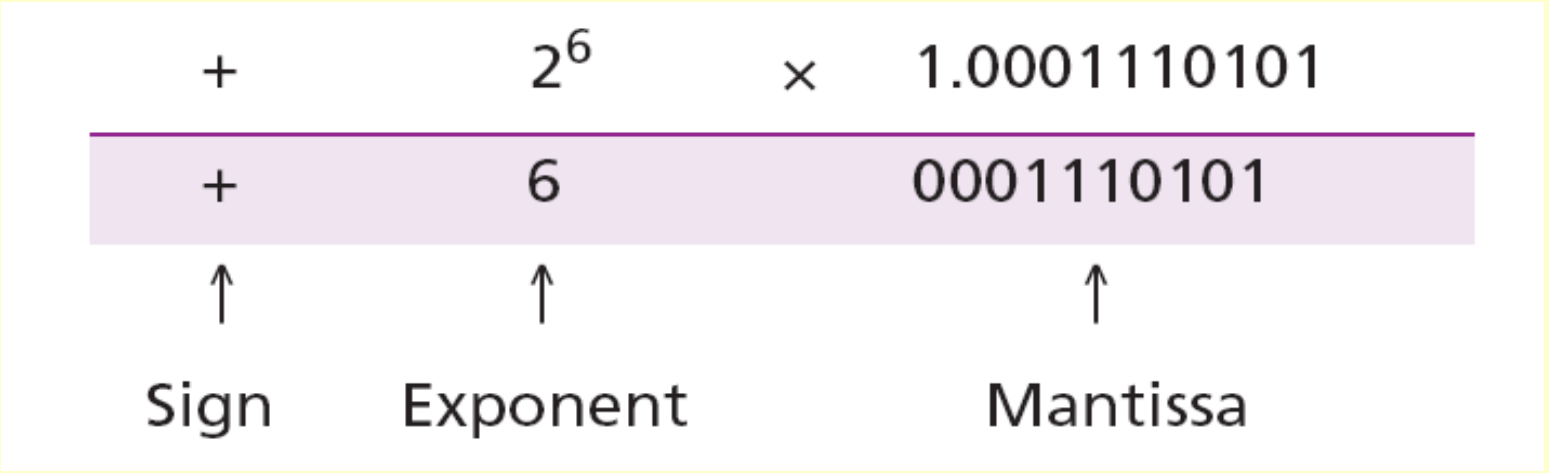
Utilizamos la misma idea, manteniendo solo un digito a la izquierda del punto decimal.

Actual number	→	–	$(0.00000000000000000000000101)_2$
Scientific notation	→	–	1.01×2^{-24}

Normalización

Para hacer la parte fija de la representación uniforme, tanto el método científico (para el sistema decimal) como el método de punto flotante (para el sistema binario) utilizan sólo un dígito diferente de cero a la izquierda del punto decimal. Esta es la llamada normalización. En el sistema decimal este dígito puede ser de 1 a 9, mientras que en el sistema binario sólo puede ser 1. De aqui en adelante, d es un dígito diferente de cero, x es un dígito, e y es 0 o 1.

Decimal	→	±	d.xxxxxxxxxxxxxx	Note: *d* is 1 to 9 and each *x* is 0 to 9
Binary	→	±	1.yyyyyyyyyyyyyy	Note: each *y* is 0 or 1



Notar que el punto y el bit 1 a la izquierda de la sección del punto fijo no son almacenados – estan implícitos.

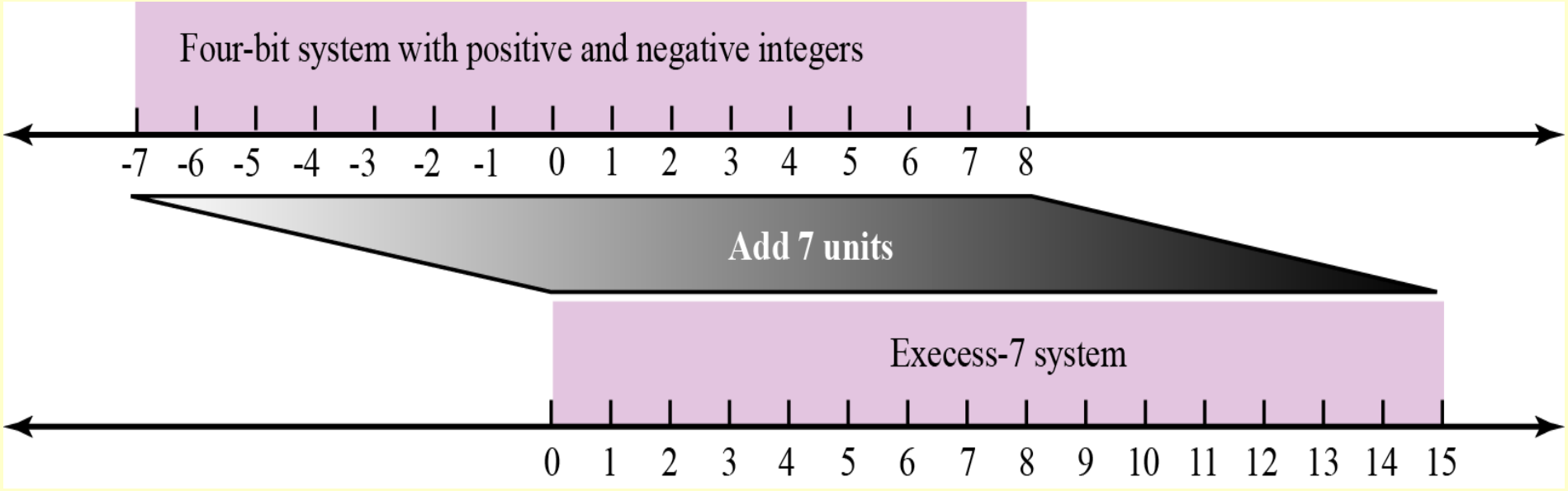
La mantisa es una parte fraccional que, junto con el signo, es tratada como un entero almacenado en representación sign-and-magnitude.

Sistema de exceso

El exponente, la potencia que muestra cuantos bits del punto decimal debe moverse hacia la izquierda o derecha, es un número con signo. Aunque esto podría haber sido almacenados usando representación two's complement, una nueva representación, llamada el Sistema de Exceso (Excess System), se utiliza en su lugar. En el sistema de exceso, enteros positivos y negativos se almacenan como enteros sin signo. Para representar un entero positivo o negativo, un entero positivo (llamado sesgo (bias)) se añade a cada número para cambiar (shift) de manera uniforme hacia el lado no negativo. El valor de este sesgo es $2^{m-1} - 1$, donde m es el tamaño de la ubicación de memoria para almacenar el exponente.

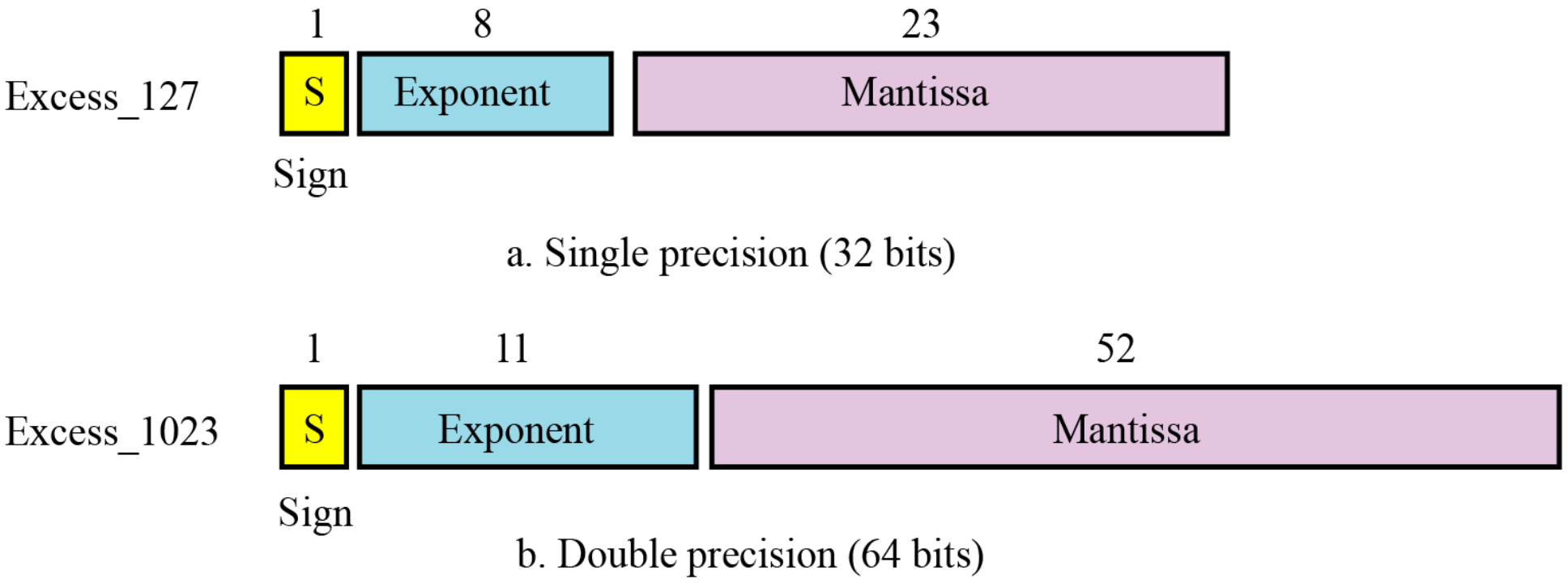
Ejemplo 22

Podemos expresar dieciseis enteros en un sistema numérico con asignación de 4 bits. Mediante la adición de siete unidades a cada número entero en este rango, se puede trasladar de manera uniforme todos los enteros a la derecha y hacer todos ellos positivos, sin cambiar la posición relativa de los enteros con respecto uno a otro, como muestra la figura. El nuevo sistema se conoce como exceso-7, o representación sesgada con un valor de 7 de sesgo.



Shifting en representación en Exceso-7

Estándar IEEE



Estándares IEEE para representación de punto flotante

Especificaciones IEEE

Table 3.2 Specifications of the two IEEE floating-point standards

Parameter	*Single Precision*	*Double Precision*
Memory location size (number of bits)	32	64
Sign size (number of bits)	1	1
Exponent size (number of bits)	8	11
Mantissa size (number of bits)	23	52
Bias (integer)	127	1023

Ejemplo 23

Mostrar la representación Excess_127 (single precision) del numero decimal 5.75.

Solucion

a. El signo es positivo, entonces S = 0.
b. Transformación decimal a binario: $5.75 = (110.11)_2$.
c. Normalización: $(110.11)_2 = (1.1011)_2 \times 2^2$.
d. $E = 2 + 127 = 129 = (10000001)_2$, M = 1011. Necesitamos anñadir diecinueve ceros a la derecha de M para hacer 23 bits.
e. La representación se muestra abajo:

0	10000001	10110000000000000000000
S	E	M

El número es almacenado en el computador como

01000000110110000000000000000000

Ejemplo 24

Mostrar la representacion Excess_127 (single precision) del numero decimal -161.875.

Solucion

a. El signo es negativo, entonces S = 1.
b. Transformacion decimal a binario: $161.875 = (10100001.111)_2$.
c. Normalizacion: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.
d. $E = 7 + 127 = 134 = (10000110)_2$, y $M = (0100001111)_2$.
e. Representacion:

1	10000110	01000011110000000000000
S	E	M

El numero es almacenado en el computador como

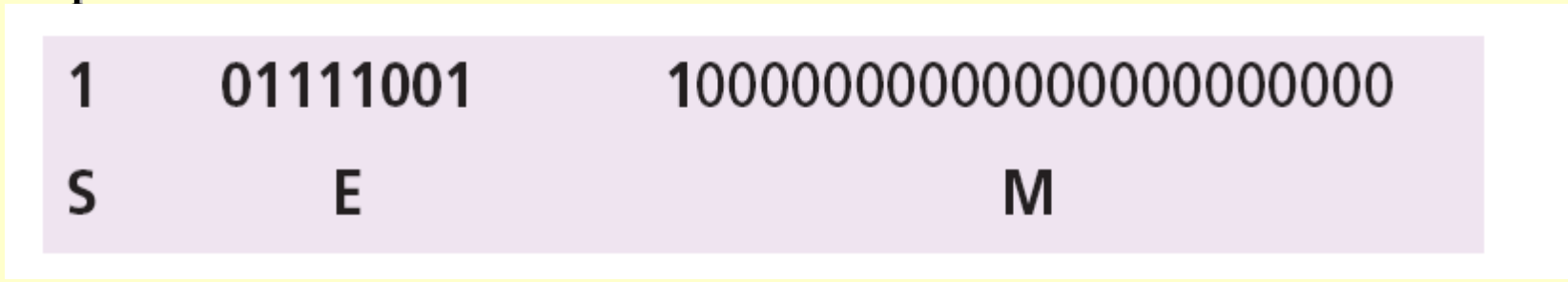
11000011010000111100000000000000

Ejemplo 25

Mostrar la representación Excess_127 (single precision) del número decimal -0.0234375.

Solución

a. El signo es negativo, entonces S = 1.
b. Transformación decimal a binario: $0.0234375 = (0.0000011)_2$.
c. Normalización: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.
d. $E = -6 + 127 = 121 = (01111001)_2$, y $M = (1)_2$.
e. Representación:



El número es almacenado en el computador como

10111100110000000000000000000000

Ejemplo 26

El patrón de bits $(11001010000000000111000100001111)_2$ esta almacenado en formato Excess_127. Mostrar el valor en decimal.

Solución

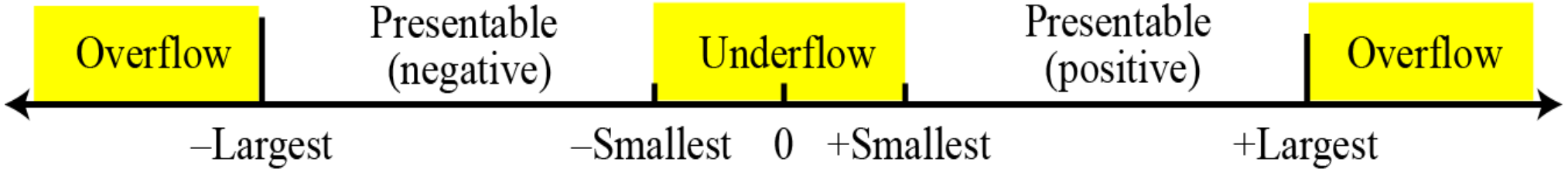
S	E	M
1	10010100	00000000111000100001111

a. El primer bit representa S, los siguientes ocho bits, E, y los restantes 23 bits, M.
b. El signo es negativo.
c. El shifter = E – 127 = 148 - 127 = 21.
d. Esto dá $(1.00000000111000100001111)_2 \times 2^{21}$.
e. El número binario es $(100000000111000100011.11)_2$.
f. El valor absoluto es 2,104,378.75.
g. El número es **–2,104,378.75**.

Overflow y Underflow

$$-\text{Largest}: -(1-2^{-24}) \times 2^{+128} \qquad +\text{Largest}: +(1-2^{-24}) \times 2^{+128}$$

$$-\text{Smallest}: -(1-2^{-1}) \times 2^{-127} \qquad +\text{Smallest}: +(1-2^{-1}) \times 2^{-127}$$



Overflow y underflow en representacion de punto flotante de reales

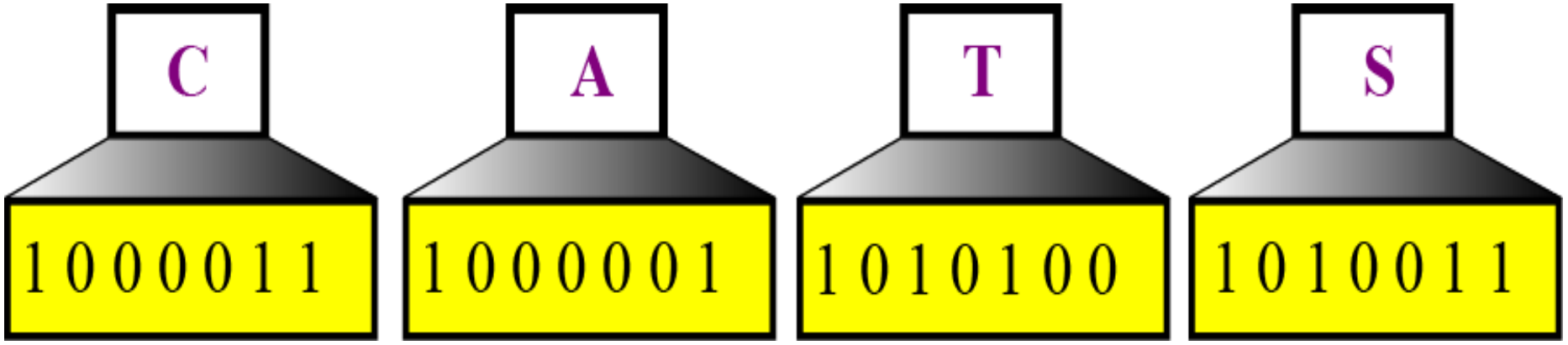
Almacenando Cero

Un número real con la parte integral y la parte fraccional siendo cero, esto es, 0.0, no puede ser almacenado usando los pasos discutidos antes. Para manejar este caso especial, se ha convenido que en este caso el signo, exponente y la mantisa sean 0s.

ALMACENANDO TEXTO

Una sección de texto en cualquier idioma es una secuencia de símbolos utilizados para representar una idea en ese idioma. Por ejemplo, el idioma Inglés utiliza 26 símbolos (A, B, C, ..., Z) para representar las letras mayúsculas, 26 de símbolos (a, b, c, ..., z) para representar letras minúsculas, nueve símbolos (0, 1, 2, ..., 9) para representar caracteres numéricos y símbolos (., ?, :, ; , …, !) para representar puntuación!. Otros símbolos como espacio, nueva línea, en blanco, y la ficha (tab) se utiliza para la alineación del texto y la legibilidad.

Podemos representar cada símbolo con un patrón de bits. En otras palabras, texto tal como “CATS”, que esta hecho de cuatro símbolos, puede ser representado por un patrón de *n*-bit, cada patrón definiendo un único símbolo.



Representando símbolos usando patrones de bits

Table 3.3 Number of symbols and bit pattern length

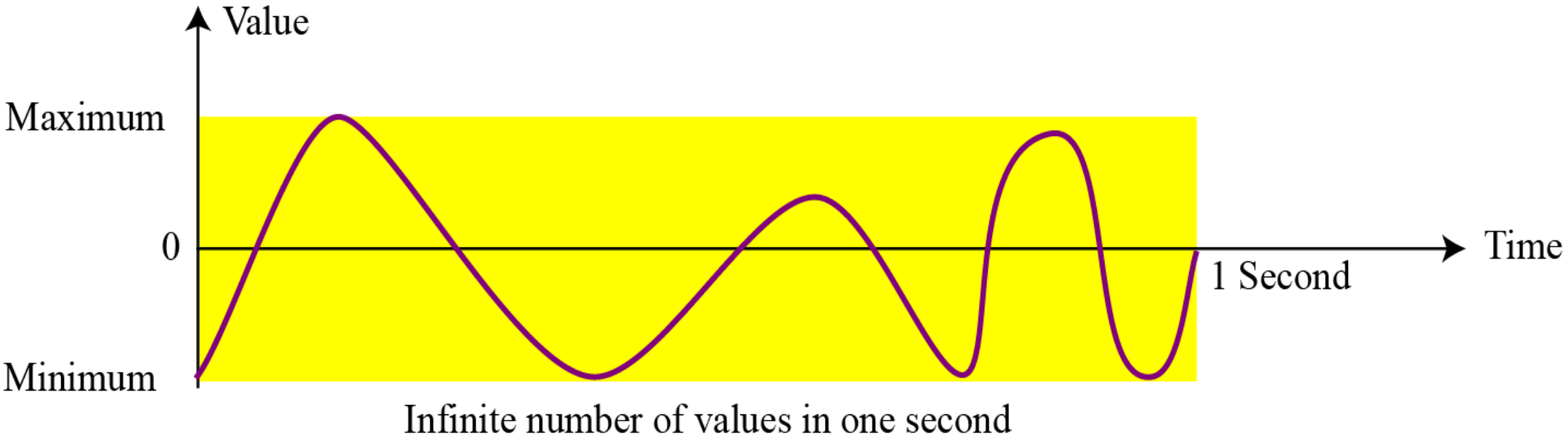
Number of symbols	*Bit pattern length*	*Number of symbols*	*Bit pattern length*
2	1	128	7
4	2	256	8
8	3	65,536	16
16	4	4,294,967,296	32

Códigos

- **ASCII**
- **Unicode**
- **Other Codes**

ALMACENANDO AUDIO

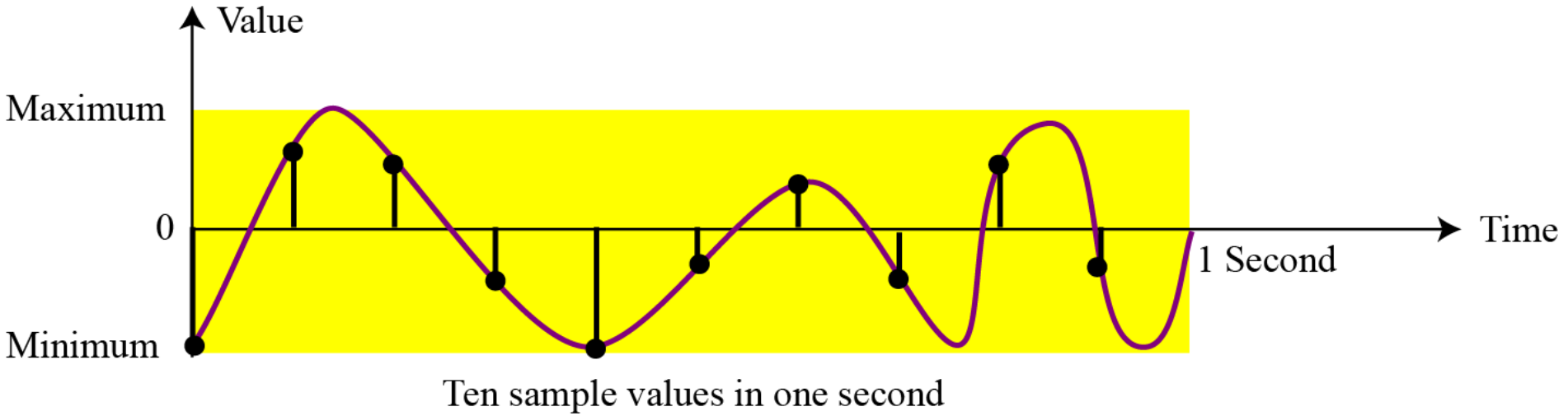
El **audio** es una representación del sonido o la música. Audio, por naturaleza, es diferente a los números o texto que hemos discutido hasta ahora. El texto está compuesto por **entidades contables** (caracteres): podemos contar el número de caracteres de texto. El texto es un ejemplo de **datos digitales**. Por el contrario, el audio no es contable. Audio es un ejemplo de **datos analógicos**. Incluso si somos capaces de medir todos sus valores en un período de tiempo, no podemos almacenarlas en la memoria del ordenador, ya que necesitaríamos un número infinito de posiciones de memoria.



Una señal de audio

Muestreo

Si no puede grabar todos los valores de una señal de audio en un intervalo, podemos grabar algunos de ellos. Muestreo significa seleccionar sólo un número finito de puntos en la señal analógica, medir sus valores y grabarlos



Muestreo de una señal de audio

Cuantización

Los valores medidos para cada muestra es un número real. Esto significa que podemos almacenar 40,000 valores reales para cada muestra de un segundo. Sin embargo, es más fácil de usar un entero sin signo (un patrón de bits) para cada muestra. Cuantización se refiere a un proceso que redondea el valor de una muestra al valor entero más cercano. Por ejemplo, si el valor real es de 17.2, puede ser redondeado a 17: si el valor es de 17,7, se puede redondear a 18.

Codificación

Los valores de la muestra cuantificada deben ser codificados como patrones de bits. Algunos sistemas asignan valores positivos y negativos a las muestras, algunos sólo desplazan la curva a la parte positiva y asignan solo valores positivos.

Si llamamos a la profundidad de bits (**bit depth**) o número de bits por muestra B, el número de muestras por segundo, S, es necesario almacenar S × B bits por cada segundo de audio. Este producto es referido a veces como tasa de bits (**bit rate**), R. Por ejemplo, si usamos 40,000 muestras por segundo y 16 bits por cada muestra, la tasa de bits es

R = 40,000 × 16 = 640,000 bits por segundo

Estandares para codificación de sonido

Hoy en día el estándar dominante para almacenamiento de audio es **MP3** (abreviatura de **MPEG Layer 3**). Esta norma es una modificación del método de compresión **MPEG** (**Motion Picture Experts Group**) utilizado para vídeo. Utiliza 44,100 muestras por segundo y 16 bits por muestra. El resultado es una señal con una velocidad de 705,600 bits por segundo, que es comprimido utilizando un método de compresión que descarta la información que no puede ser detectada por el oído humano. Esto se llama compresión con pérdida, como opuesto a la compresión sin pérdidas.

ALMACENANDO IMAGENES

Imagenes son almacenadas en computadores usando dos tecnicas diferentes: **raster graphics** y **vector graphics**.

Raster graphics

Gráficos de mapa de bits (**Raster graphics** (or **bitmap graphics**)) se utiliza cuando es necesario almacenar una imagen analógica, tal como una fotografía. Una fotografía consiste de datos analógicos, similar a la información de audio. La diferencia es que la intensidad (color) de los datos varía en el espacio en lugar del tiempo. Esto significa que los datos deben muestrearse. Sin embargo, el muestreo en este caso, normalmente se llama digitalización (**scanning**). Las muestras se denominan píxeles (**pixels -** elementos de imagen).

Resolución

Al igual que el muestreo de audio, en la digitalización de la imagen (scanning) necesitamos decidir el número de píxeles que debe registrar por cada pulgada cuadrada o lineal. La tasa de scanning en el procesamiento de la imagen se llama **resolución**. Si la resolución es suficientemente alta, el ojo humano no puede reconocer la discontinuidad en las imágenes reproducidas

Profundidad de color

El número de bits utilizados para representar un píxel, su profundidad de color (**color depth**), depende de cómo el color del píxel es manipulado mediante diferentes técnicas de codificación. La percepción del color es como nuestros ojos responden a un haz de luz. Nuestros ojos tienen diferentes tipos de células fotorreceptoras: algunas responden a los tres colores primarios rojo, verde y azul (a menudo llamado **RGB**), mientras que otros se limitan a responder a la intensidad de la luz.

True-Color

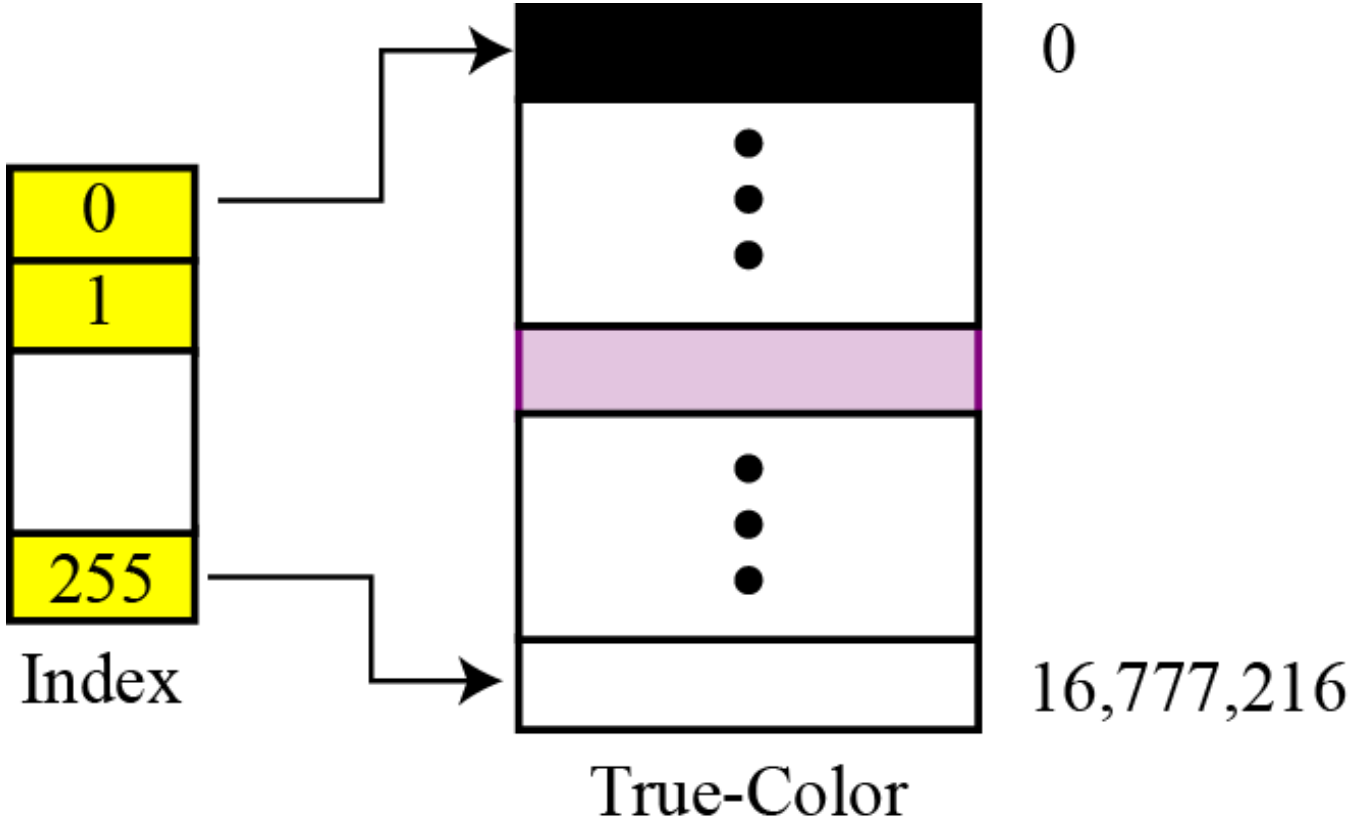
Una de las técnicas usadas para codificar un pixel es llamada True-Color, usa **24** bits para la codificación del pixel.

Table 3.4 Some colors defined in True-Color

Color	*Red*	*Green*	*Blue*	*Color*	*Red*	*Green*	*Blue*
Black	0	0	0	Yellow	255	255	0
Red	255	0	0	Cyan	0	255	255
Green	0	255	0	Magenta	255	0	255
Blue	0	0	255	White	255	255	255

Indexed-Color

El esquema **indexed color**—o **palette color**—usa solo una porción de esos colores.



Relación entre el Indexed-Color y el True-Color

Por ejemplo, una cámara digital de alta calidad utiliza casi tres millones de píxeles de una foto de 3 x 5 pulgadas. A continuación se muestra el número de bits que necesitan ser almacenados con cada régimen:

True-Color:	3,000,000	×	24	=	72,000,000
Indexed-Color:	3,000,000	×	8	=	24,000,000

Estandares para codificacion de imagenes

Varios estándares de facto para la codificación de la imagen están en uso. JPEG (Joint Photographic Experts Group) utiliza el esquema de True-Color, pero comprime la imagen para reducir el número de bits. GIF (Graphic Interchange Format), por el contrario, utiliza el esquema de Indexed-Color.

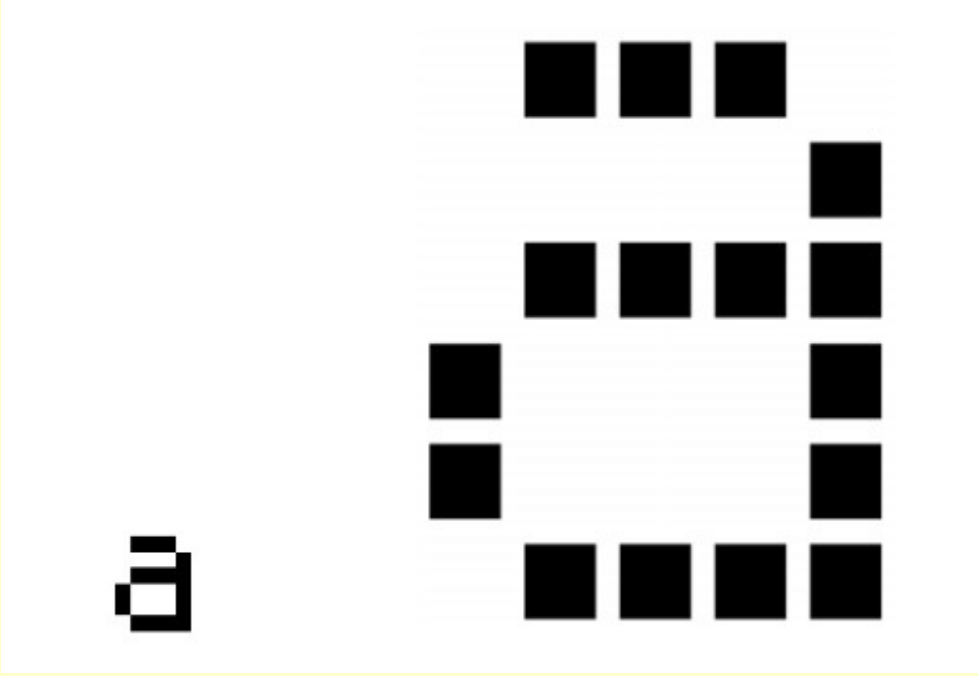
Vector graphics

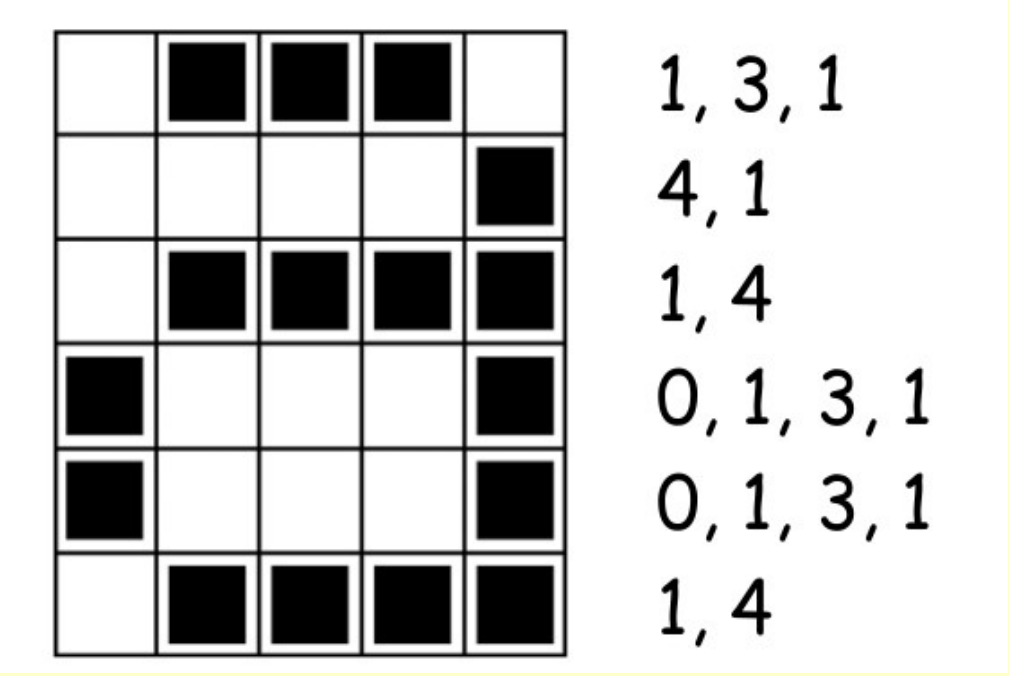
Raster graphics tienen dos desventajas: el tamaño del archivo es grande y reajuste es problematico. Ampliar un raster graphics (imagen de mapa de bits gráficos) significa ampliación de los píxeles, por lo que la imagen se ve desigual cuando se agranda. El método de codificación de imagen de gráfico vectorial (**vector graphic**), sin embargo, no almacena los patrones de bits para cada píxel. Una imagen es descompuesta en una combinación de formas geométricas como líneas, cuadrados o círculos.

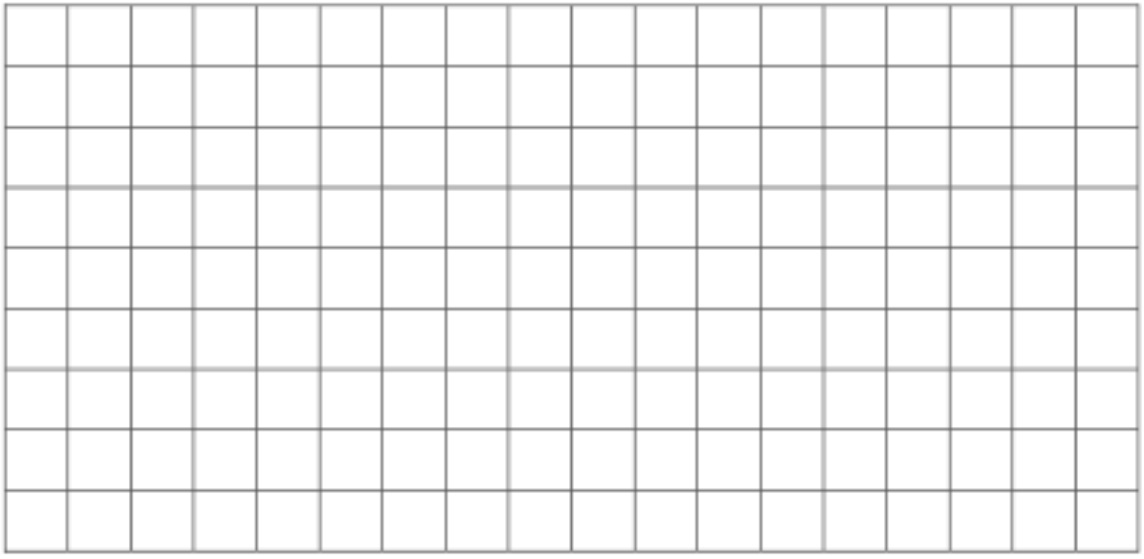
Por ejemplo, considere un círculo de radio r. Las piezas principales de información que un programa necesita para dibujar este círculo son los siguientes:

1. **El radio *r* y la ecuación de un círculo.**
2. **La localización del punto central del círculo.**
3. **El estilo de línea y color del trazo.**
4. **El estilo y color del relleno.**

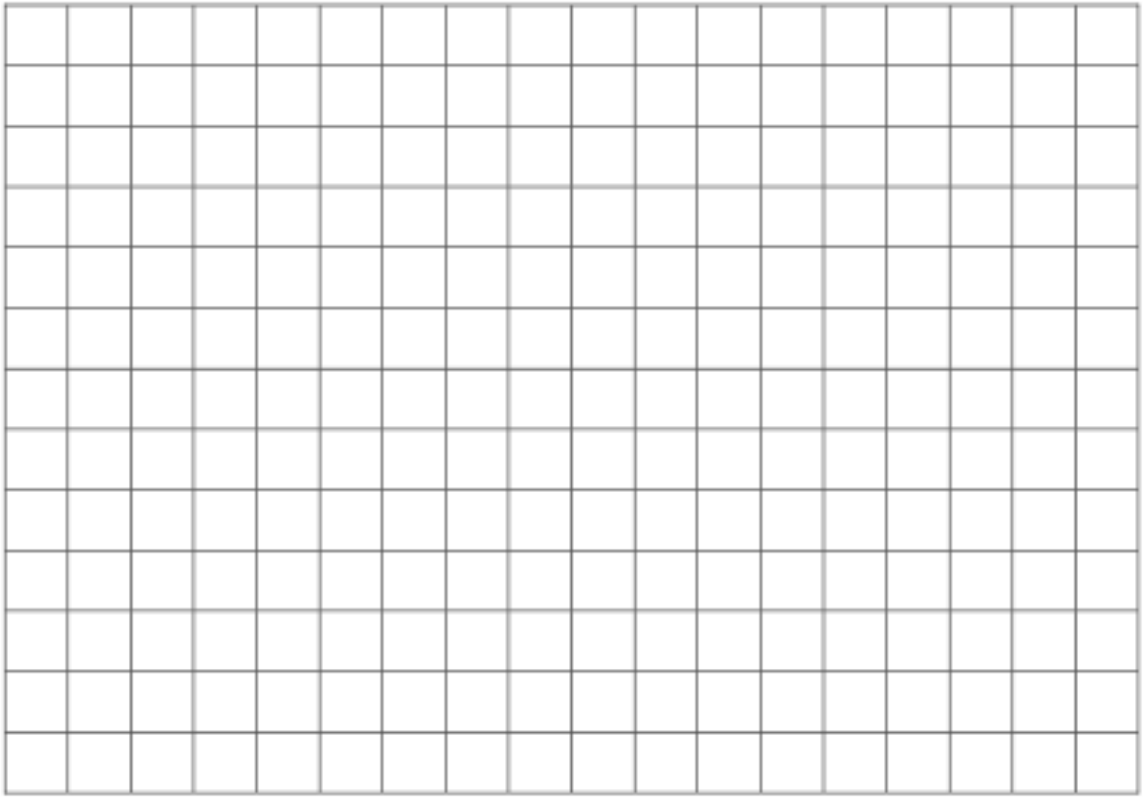
Codificación de imágenes







4, 11
4, 9, 2, 1
4, 9, 2, 1
4, 11
4, 9
4, 9
5, 7
0, 17
1, 15



6, 5, 2, 3
4, 2, 5, 2, 3, 1
3, 1, 9, 1, 2, 1
3, 1, 9, 1, 1, 1
2, 1, 11, 1
2, 1, 10, 2
2, 1, 9, 1, 1, 1
2, 1, 8, 1, 2, 1
2, 1, 7, 1, 3, 1
1, 1, 1, 1, 4, 2, 3, 1
0, 1, 2, 1, 2, 2, 5, 1
0, 1, 3, 2, 5, 2
1, 3, 2, 5

ALMACENANDO VIDEO

Video es una representación de imágenes (llamada marcos - **frames**) con el tiempo. Una película consiste de una serie de frames mostrados uno tras otro. En otras palabras, el video es la representación de la información que cambia en el espacio y en el tiempo. Por lo tanto, si sabemos cómo almacenar una imagen dentro de un computador, también sabemos cómo almacenar vídeo: cada imagen o marco (frame) se transforma en un conjunto de patrones de bits y se almacena. La combinación de las imágenes entonces representa el vídeo