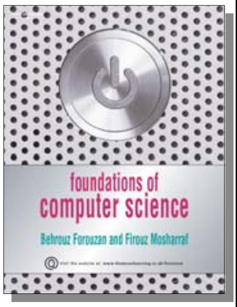
3
Data Storage



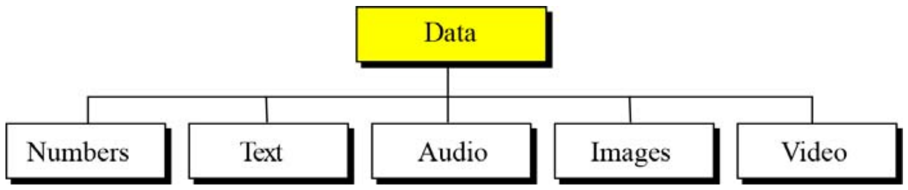
Objectives

After studying this chapter, students should be able to:

- ❑ List five different data types used in a computer.
- ❑ Describe how integers are stored in a computer.
- ❑ Describe how reals are stored in a computer.
- ❑ Describe how text is stored in a computer using one of the various encoding systems.
- ❑ Describe how audio is stored in a computer using sampling, quantization and encoding.
- ❑ Describe how images are stored in a computer using raster and vector graphics schemes.
- ❑ Describe how video is stored in a computer as a representation of images changing in time.

3-1 Introduction

Data today comes in different forms including numbers, text, audio, image and video.



The computer industry uses the term "multimedia" to define information that contains numbers, text, images, audio and video.

3.3

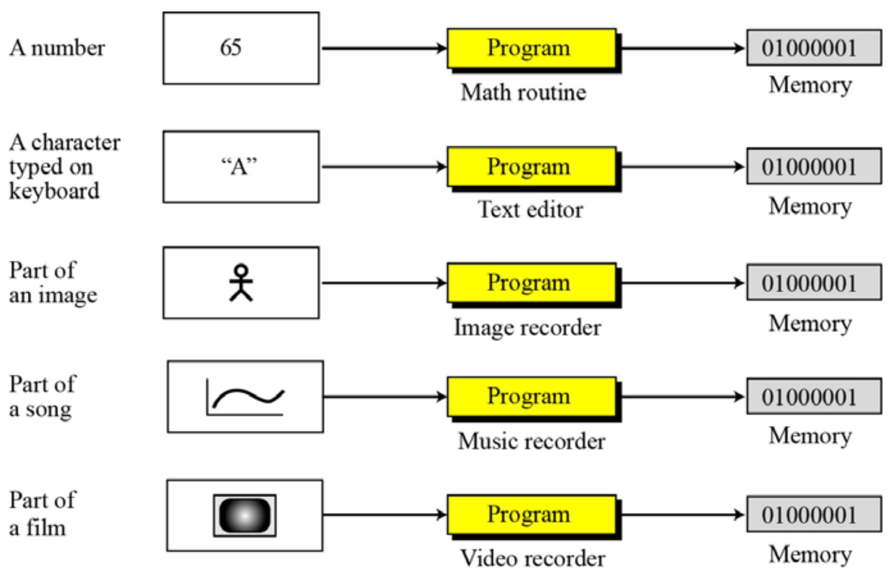
Data Inside the Computer

All data types are transformed into a uniform representation when they are stored in a computer and transformed back to their original form when retrieved. **This universal representation is called a bit pattern or a sequence of 0s and 1s**.

1 0 0 0 1 0 1 0 1 1 1 1 1 1

3.4

Storage of Different Data Types



3.5

Data Compression

To occupy less memory space, data is normally compressed before being stored in the computer. Data compression is a very broad and involved subject, so we have dedicated the whole of Chapter 15 to the subject.

Error Detection and Correction

Another issue related to **data is the detection and correction of errors during transmission or storage**. We discuss this issue briefly in Appendix H.

3.6

3-2 Storing Numbers

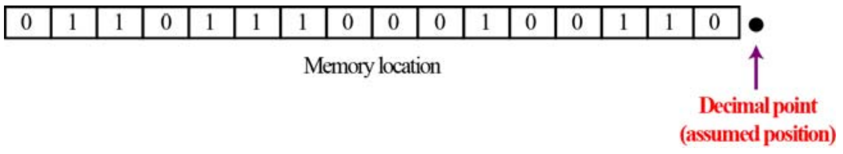
A **number** is changed to the binary system before being stored in the computer memory, as described in Chapter 2. However, there are still two issues that need to be handled:

1. How to store the sign of the number.
2. How to show the decimal point.

3.7

Storing Integers

Integers are whole numbers (numbers without a fractional part). For example, 134 and −125 are integers, whereas 134.23 and −0.235 are not. An integer can be thought of as a number in which the position of the decimal point is fixed: the decimal point is to the right of the least significant (rightmost) bit. For this reason, **fixed-point representation is used to store an integer**, as shown below.



An integer is normally stored in memory using fixed-point representation.

3.8

Unsigned Representation

An **unsigned integer** is an integer that can never be negative and can take only 0 or positive values. Its range is between 0 and positive infinity.

An input device stores an unsigned integer using the following steps:

1. The integer is changed to binary.
2. If the number of bits is less than *n*, 0s are added to the left.

Example 3.1

Store 7 in an 8-bit memory location using unsigned representation.

Solution

First change the integer to binary, $(111)_2$. Add five 0s to make a total of eight bits, $(00000111)_2$. The integer is stored in the memory location. Note that the subscript 2 is used to emphasize that the integer is binary, but the subscript is not stored in the computer.

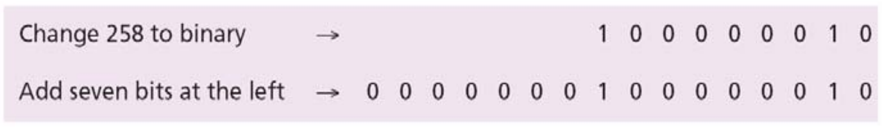
Change 7 to binary	→							1	1	1
Add five bits at the left	→		0	0	0	0	0	1	1	1

Example 3.2

Store 258 in a 16-bit memory location.

Solution

First change the integer to binary $(100000010)_2$. Add seven 0s to make a total of sixteen bits, $(0000000100000010)_2$. The integer is stored in the memory location.



Change 258 to binary	→								1	0	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	0	1	0

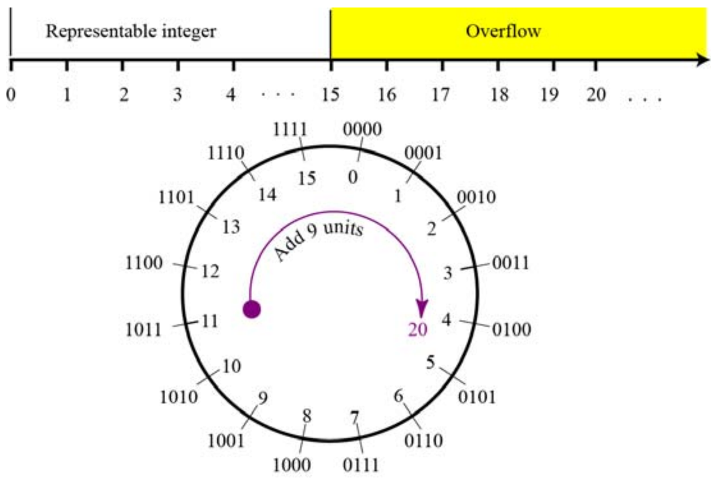
Example 3.3

What is returned from an output device when it retrieves the bit string 00101011 stored in memory **as an unsigned integer**?

Solution

Using the procedure shown in Chapter 2, the binary integer is converted to the unsigned integer 43.

What happens if we try to store an integer that is larger than $2^4 - 1 = 15$ in a memory location that can only hold four bits.



Overflow in unsigned integers

3.13

Sign-and-Magnitude Representation

In this method, the available range for unsigned integers (0 to 2^n-1) is divided into two equal sub-ranges. **The first half represents positive integers**, **the second half, negative integers**.

Noted that there are **two different representations for zero**.

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

In sign-and-magnitude representation, the leftmost bit defines the sign of the integer. If it is 0, the integer is positive. If it is 1, the integer is negative.

3.14

Example 3.4

Store +28 in an 8-bit memory location using sign-and-magnitude representation.

Solution

The integer is changed to 7-bit binary. The leftmost bit is set to 0. The 8-bit number is stored.

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

Example 3.5

Store –28 in an 8-bit memory location using sign-and-magnitude representation.

Solution

The integer is changed to 7-bit binary. The leftmost bit is set to 1. The 8-bit number is stored.

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

Example 3.6

Retrieve the integer that is stored as 01001101 in sign-and-magnitude representation.

Solution

Since the leftmost bit is 0, the sign is positive. The rest of the bits (1001101) are changed to decimal as 77. After adding the sign, the integer is +77.

Example 3.7

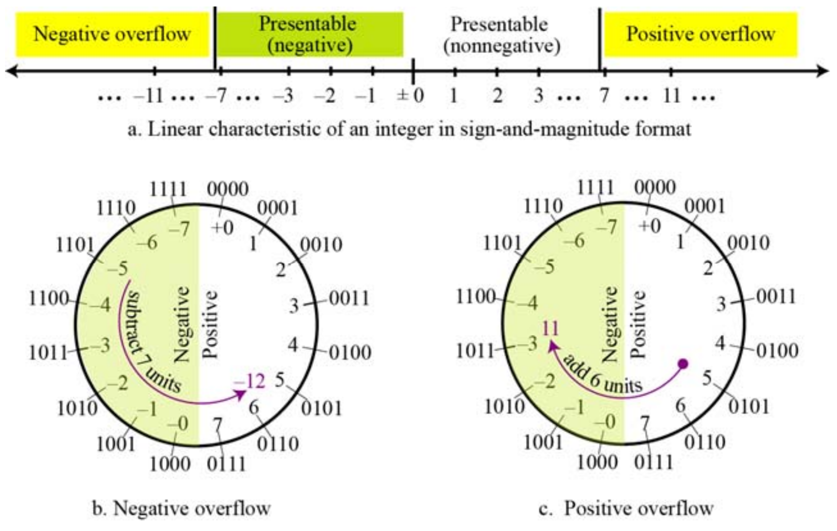
Retrieve the integer that is stored as 10100001 in sign-and-magnitude representation.

Solution

Since the leftmost bit is 1, the sign is negative. The rest of the bits (0100001) are changed to decimal as 33. After adding the sign, the integer is −33.

3.17

Both **positive and negative overflow** when storing an integer in sign-and-magnitude representation using a 4-bit memory location.

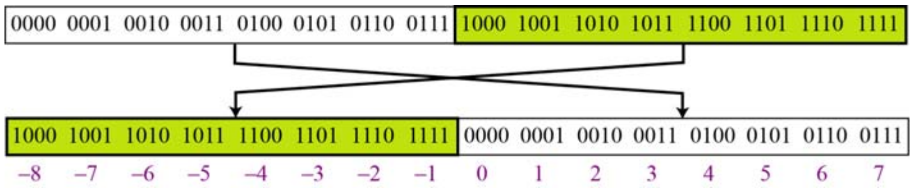


Overflow in sign-and-magnitude representation

3.18

Two’s Complement Representation

Almost all computers use two’s complement representation to store a signed integer in an *n*-bit memory location. An unsigned integer of (0 to 2^n-1) is divided into two equal sub-ranges. The first sub-range is used to represent nonnegative integers, the second half to represent negative integers. The bit patterns are then assigned to negative and nonnegative integers.



In two’s complement representation, the leftmost bit defines the sign of the integer. If it is 0, the integer is positive. If it is 1, the integer is negative.

3.19

One’s Complementing

Before we discuss this representation further, we need to introduce two operations. The first is called ***one’s complementing*** or taking the one’s complement of an integer. The operation can be applied to any integer, positive or negative. This operation simply reverses (flips) each bit. A 0-bit is changed to a 1-bit, a 1-bit is changed to a 0-bit.

Example 3.8

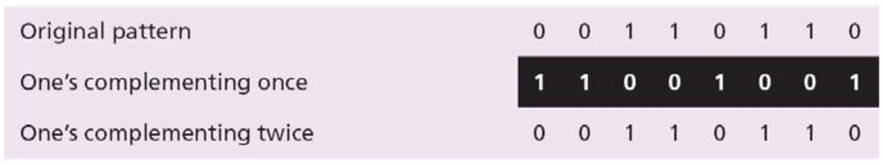
The following shows how we take the one’s complement of the integer 00110110.

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

3.20

Example 3.9

The following shows that we get the original integer if we apply the one's complement operations twice.



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

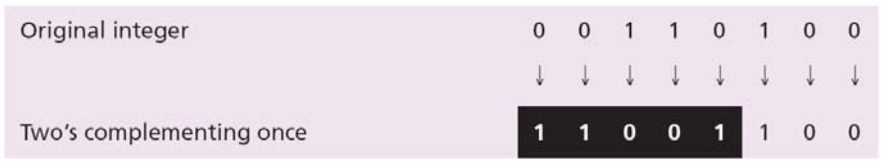
3.21

Two's Complementing

The second operation is called ***two's complementing*** or taking the two's complement of an integer in binary. This operation is done in two steps. **First, we copy bits from the right until a 1 is copied; then, we flip the rest of the bits.**

Example 3.10

The following shows how we take the two's complement of the integer 00110100.



Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing once	1	1	0	0	1	1	0	0

3.22

Example 3.11

The following shows that we always get the original integer if we apply the two's complement operation twice.

Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing once	1	1	0	0	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing twice	0	0	1	1	0	1	0	0

An alternative way to take the two's complement of an integer is to first take the one's complement and then add 1 to the result.

3.23

Example 3.12

Store the integer 28 in an 8-bit memory location using two's complement representation.

Solution

The integer is positive (no sign means positive), so after decimal to binary transformation no more action is needed. Note that five extra 0s are added to the left of the integer to make it eight bits.

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

3.24

Example 3.13

Store −28 in an 8-bit memory location using two's complement representation.

Solution

The integer is negative, so after changing to binary, the computer applies the two's complement operation on the integer.

Change 28 to 8-bit binary	0	0	0	1	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two's complement operation	1	1	1	0	0	1	0	0

Example 3.14

Retrieve the integer that is stored as 00001101 in memory in two's complement format.

Solution

The leftmost bit is 0, so the sign is positive. The integer is changed to decimal and the sign is added.

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

Example 3.15

Retrieve the integer that is stored as 11100110 in memory using two's complement format.

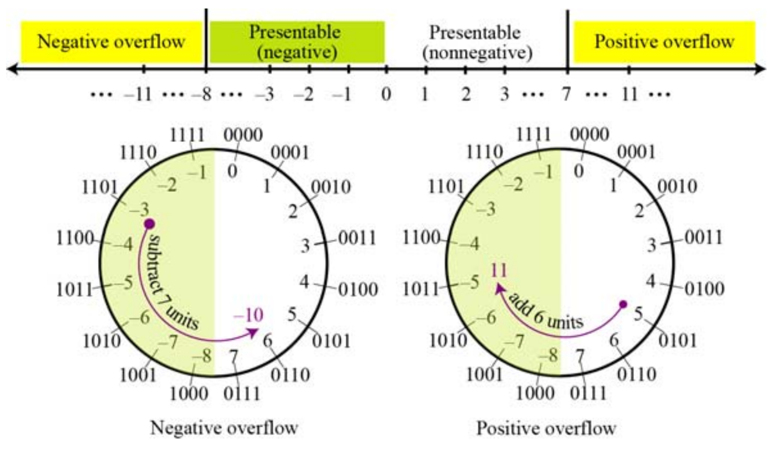
Solution

The leftmost bit is 1, so the integer is negative. The integer needs to be two's complemented before changing to decimal.

Leftmost bit is 1. The sign is negative	1	1	1	0	0	1	1	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two's complement operation	0	0	0	1	1	0	1	0
Integer changed to decimal								26
Sign is added								−26

3.27

There is only one zero in two's complement notation.



Overflow in two's complement representation

3.28

Comparison

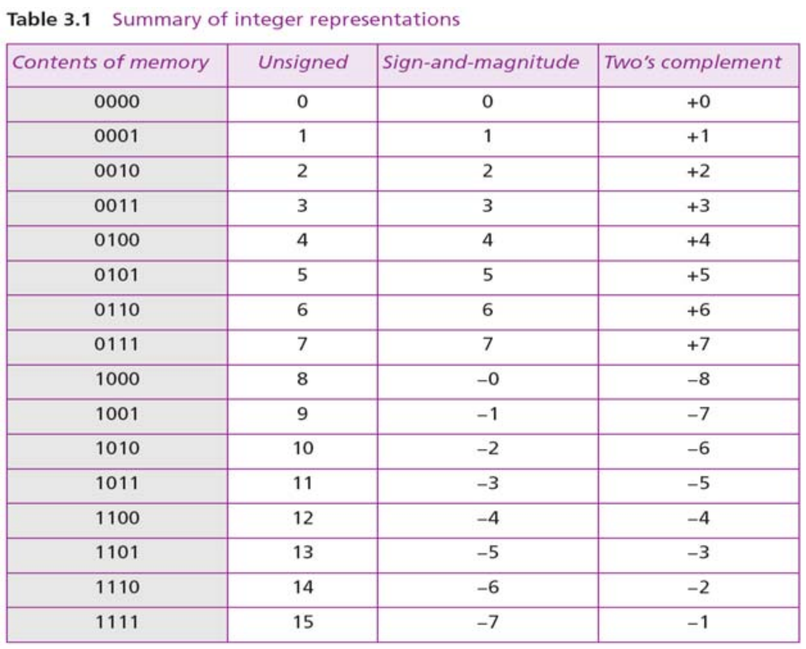


Table 3.1 Summary of integer representations

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

3.29

Storing Reals

A real is a number with an integral part and a fractional part. For example, 23.7 is a real number—the integral part is 27 and the fractional part is 7/10. Although a **fixed-point representation can be used to represent a real number**, the result **may not be accurate** or it **may not have the required precision**. The next two examples explain why.

Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.

3.30

Example 3.16

In the decimal system, assume that we use a **fixed-point representation** with two digits at the right of the decimal point and fourteen digits at the left of the decimal point, for a total of sixteen digits. The precision of a real number in this system is lost if we try to represent a decimal number such as **1.00234**: the system stores the number as **1.00**.

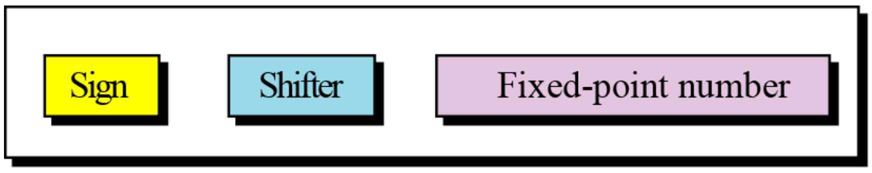
Example 3.17

In the decimal system, assume that we use a fixed-point representation with six digits to the right of the decimal point and ten digits for the left of the decimal point, for a total of sixteen digits. The accuracy of a real number in this system is lost if we try to represent a decimal number such as **236154302345.00**. The system stores the number as **6154302345.00**: the integral part is much smaller than it should be.

3.31

Floating-Point Representation

The solution for maintaining accuracy or precision is to use **floating-point representation**.



Floating-point pepresentation

Figure 3.9 **The three parts of a real number in floating-point representation**

A floating point representation of a number is made up of three parts: a sign, a shifter and a fixed-point number.

3.32

Example 3.18

The following shows the decimal number

7,425,000,000,000,000,000,000.00

in scientific notation (floating-point representation).

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

The three sections are the **sign** (+), the **shifter** (21) and the **fixed-point part** (7.425). Note that the shifter is the exponent.

3.33

Example 3.19

Show the number

−0.0000000000000232

in scientific notation (floating-point representation).

Solution

We use the same approach as in the previous example—we move the decimal point after the digit 2, as shown below:

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

The three sections are the **sign** (−), the **shifter** (−14) and the **fixed-point part** (2.32). Note that the shifter is the exponent.

3.34

Example 3.20

Show the number

$$(101001000000000000000000000000000.00)_2$$

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the decimal point.

Actual number	→	+	$(101001000000000000000000000000000.00)_2$
Scientific notation	→	+	1.01001×2^{32}

3.35

Example 3.21

Show the number

$$-(0.00000000000000000000000101)_2$$

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the decimal point.

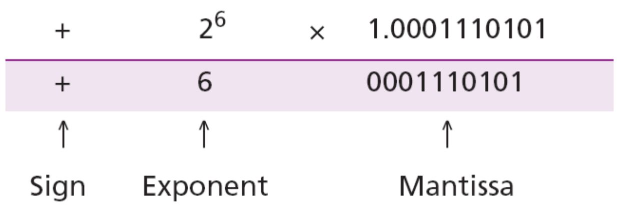
Actual number	→	–	$(0.00000000000000000000000101)_2$
Scientific notation	→	–	1.01×2^{-24}

3.36

Normalization

To make the fixed part of the representation uniform, both the scientific method (for the decimal system) and the floating-point method (for the binary system) use only **one non-zero digit on the left of the decimal point**. This is called **normalization**. In the decimal system this digit can be 1 to 9, while in the binary system it can only be 1. In the following, *d* is a non-zero digit, *x* is a digit, and *y* is either 0 or 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



Note that the point and the bit 1 to the left of the fixed-point section are not stored —they are implicit.

The mantissa is a fractional part that, together with the sign, is treated like an integer stored in sign-and-magnitude representation.

Excess System

The exponent, the power that shows how many bits the decimal point should be moved to the left or right, is a signed number. Although this could have been stored using two's complement representation, a new representation, called the **Excess system**, is used instead. In the Excess system, both positive and negative integers are stored as unsigned integers. To represent a positive or negative integer, a positive integer (called a bias) is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where *m* is the size of the memory location to store the exponent.

3.39

Example 3.22

We can express sixteen integers in a number system with 4-bit allocation. By adding seven units to each integer in this range, we can uniformly translate all integers to the right and make all of them positive without changing the relative position of the integers with respect to each other, as shown in the figure. The new system is referred to as **Excess-7**, or biased representation with biasing value of 7.

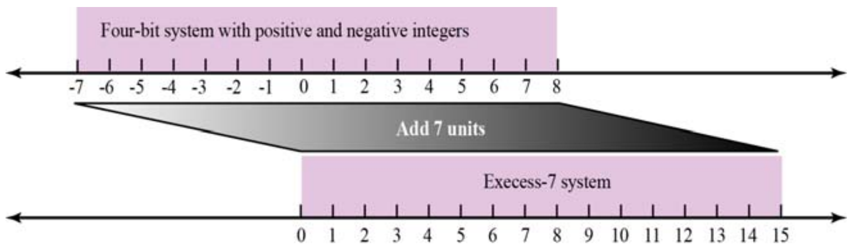


Figure 3.11 **Shifting in Excess representation**

3.40

IEEE Standard

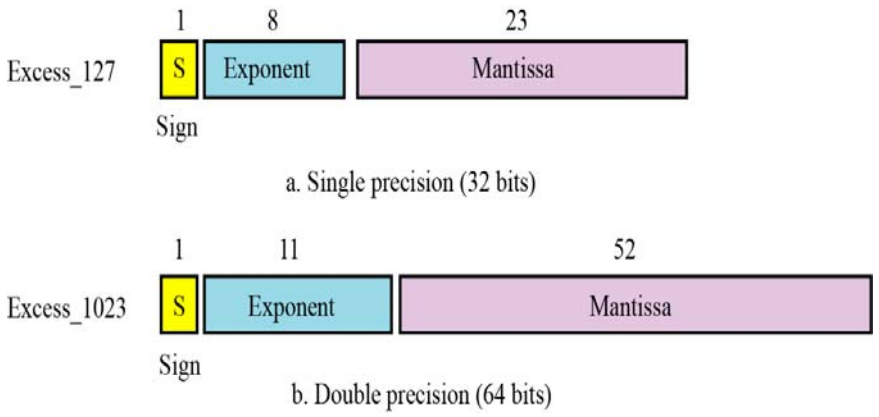


Figure 3.12 **IEEE standards for floating-point representation**

Note that the point and the bit 1 to the left of the fixed-point section are not stored —they are implicit.

3.41

IEEE Specifications

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

3.42

Example 3.23

Show the Excess_127 (single precision) representation of the decimal number 5.75.

Solution

a. The sign is positive, so S = 0.
b. Decimal to binary transformation: $5.75 = (101.11)_2$.
c. Normalization: $(101.11)_2 = (1.1011)_2 \times 2^2$.
d. $E = 2 + 127 = 129 = (10000001)_2$, M = 1011. We need to add nineteen zeros at the right of M to make it 23 bits.
e. The presentation is shown below:

0	10000001	10110000000000000000000
S	E	M

The number is stored in the computer as

$\boxed{01000000110110000000000000000000}$

3.43

Example 3.24

Show the Excess_127 (single precision) representation of the decimal number –161.875.

Solution

a. The sign is negative, so S = 1.
b. Decimal to binary transformation: $161.875 = (10100001.111)_2$.
c. Normalization: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.
d. $E = 7 + 127 = 134 = (10000110)_2$ and $M = (0100001111)_2$.
e. Representation:

1	10000110	01000011110000000000000
S	E	M

The number is stored in the computer as

$\boxed{11000011010000111100000000000000}$

3.44

Example 3.25

Show the Excess_127 (single precision) representation of the decimal number –0.0234375.

Solution

a. S = 1 (the number is negative).
b. Decimal to binary transformation: 0.0234375 = $(0.0000011)_2$.
c. Normalization: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.
d. E = –6 + 127 = 121 = $(01111001)_2$ and M = $(1)_2$.
e. Representation:

1	01111001	10000000000000000000000
S	E	M

The number is stored in the computer as

$\boxed{10111100110000000000000000000000}$

3.45

Example 3.26

The bit pattern $(11001010000000000111000100001111)_2$ is stored in Excess_127 format. Show the value in decimal.

Solution

a. The first bit represents S, the next eight bits, E and the remaining 23 bits, M.

S	E	M
1	10010100	00000000111000100001111

b. The sign is negative.
c. The shifter = E − 127 = 148 − 127 = 21.
d. This gives us $(1.00000000111000100001111)_2 \times 2^{21}$.
e. The binary number is $(1000000001110001000011.11)_2$.
f. The absolute value is 2,104,378.75.
g. The number is **−2,104,378.75**.

3.46

Overflow and Underflow

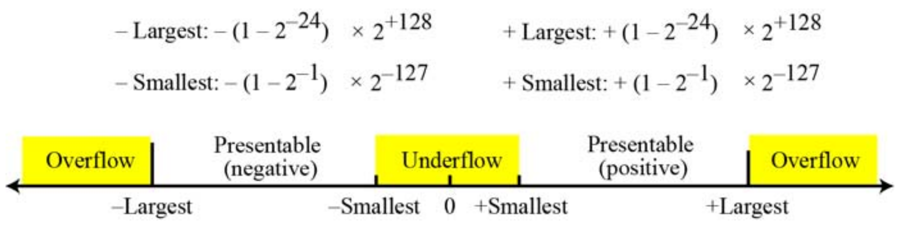

– Largest: – $(1 - 2^{-24}) \times 2^{+128}$
– Smallest: – $(1 - 2^{-1}) \times 2^{-127}$
+ Largest: + $(1 - 2^{-24}) \times 2^{+128}$
+ Smallest: + $(1 - 2^{-1}) \times 2^{-127}$
Overflow
Presentable (negative)
Underflow
Presentable (positive)
Overflow
–Largest
–Smallest
0
+Smallest
+Largest

Figure 3.12 **Overflow and underflow in floating-point representation of reals**

Storing Zero

A real number with an integral part and the fractional part set to zero, that is, 0.0, cannot be stored using the steps discussed above. To handle this special case, it is agreed that in this case the sign, exponent and the mantissa are set to 0s.

3-3 Storing Text

A section of text in any language is a sequence of symbols used to represent an idea in that language. For example, the English language uses 26 symbols (A, B, C,…, Z) to represent uppercase letters, 26 symbols (a, b, c, …, z) to represent lowercase letters, nine symbols (0, 1, 2, …, 9) to represent numeric characters and symbols (., ?, :, ; , …, !) to represent punctuation. Other symbols such as blank, newline, and tab are used for text alignment and readability.

We can represent each symbol with a bit pattern. In other words, text such as “CATS”, which is made up from four symbols, can be represented as four *n*-bit patterns, each pattern defining a single symbol (Figure 3.14).

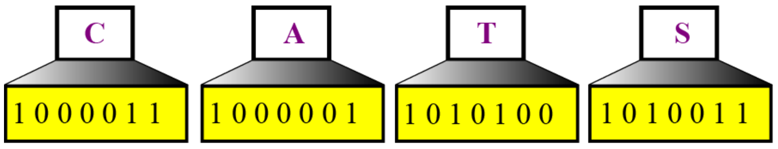


Figure 3.13 **Representing symbols using bit patterns**

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

3.49

Codes

- ❑ **ASCII (American Standard Code of Information Interchange):**
  ASCII: Using 7 bits for each symbol
  Extended ASCII: Using 8 bits for each symbol
- ❑ **Unicode**

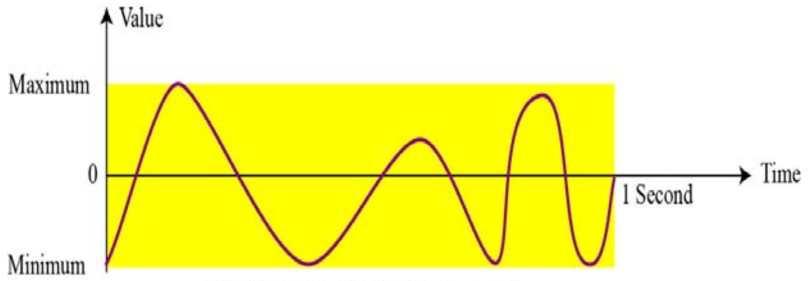
Using 32 bits to represent up to 2^{32} symbols (including graphical and special symbols). Unicode is suitable for the communication in multiple languages. ASCII and extended ASCII are parts of Unicode.

- ❑ **Other Codes**

3.50

3-4 Storing Audio

Audio is a representation of sound or music. Audio is not countable. **Audio is an example of analog data**. Even if we are able to measure all its values in a period of time, we cannot store these in the computer's memory, as we would need an infinite number of memory locations. Audio varies with time.

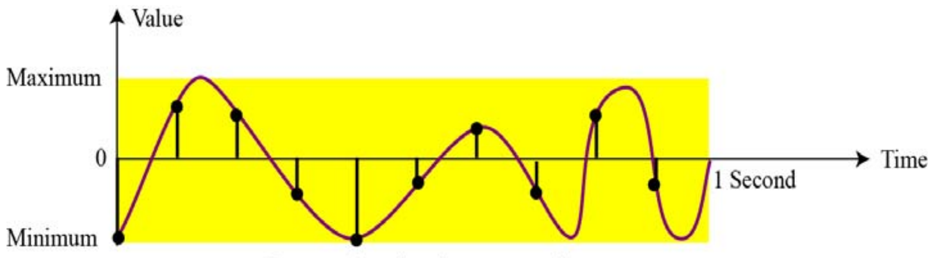


Infinite number of values in one second

3.51

Sampling

If we cannot record all the values of an audio signal over an interval, we can record some of them. **Sampling means that we select only a finite number of points on the analog signal, measure their values, and record them**.



Ten sample values in one second

3.52

Quantization

The value measured for each sample is a real number. This means that we can **store 40,000 real values for each one second sample**. However, it is simpler to use an unsigned integer (a bit pattern) for each sample. **Quantization refers to a process that rounds the value of a sample to the closest integer value**. For example, if the real value is 17.2, it can be rounded down to 17: if the value is 17.7, it can be rounded up to 18.

3.53

Encoding

The quantized sample values need to be encoded as bit patterns. Some systems assign positive and negative values to samples, some just shift the curve to the positive part and assign only positive values.

If we call the **bit depth** or **number of bits per sample B, the number of samples per second, S, we need to store S × B bits for each second of audio**. This product is sometimes referred to as **bit rate**, R. For example, if we use 40,000 samples per second and 16 bits per each sample, the bit rate is

$$\mathbf{R = 40{,}000 \times 16 = 640{,}000 \text{ bits per second}}$$

3.54

Standards for Sound Encoding

Today the dominant standard for storing audio is **MP3** (short for **MPEG Layer 3**). This standard is a modification of the **MPEG** (**Motion Picture Experts Group**) compression method used for video. It uses **44100 samples per second** and **16 bits per sample**. The result is a signal with a bit rate of **705,600 bits per second**, which is compressed using a compression method that discards information that cannot be detected by the human ear. This is called **lossy compression**, as opposed to **lossless compression**: see Chapter 15.

3.55

3-5 Storing Images

Images are stored in computers using two different techniques: **raster graphics** and **vector graphics**.

Raster graphics

Raster graphics (or **bitmap graphics**) is used when we need to store an analog image such as a photograph. A photograph consists of analog data, similar to audio information. The difference is that the intensity (color) of data varies in space instead of in time. This means that data must be sampled. However, sampling in this case is normally called **scanning**. The samples are called **pixels** (picture elements).

3.56

Resolution

Just like audio sampling, in image scanning we need to decide how many pixels we should record for each square or linear inch. The scanning rate in image processing is called **resolution**. If the resolution is sufficiently high, the human eye cannot recognize the discontinuity in reproduced images.

Color Depth

The number of bits used to represent a pixel, its **color depth**, depends on how the pixel's color is handled by different encoding techniques. The perception of color is how our eyes respond to a beam of light. Our eyes have different types of photoreceptor cells: some respond to the three primary colors red, green and blue (often called **RGB**), while others merely respond to the intensity of light.

3.57

True-Color

One of the techniques used to encode a pixel is called **True-Color, which uses 24 bits to encode a pixel**.

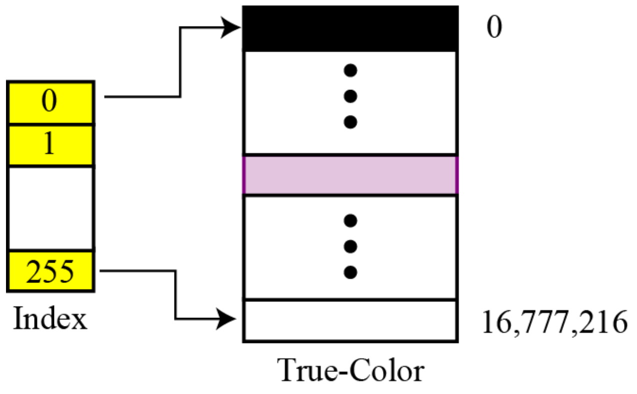
Table 3.4 Some colors defined in True-Color

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

3.58

Indexed Color

The **indexed color**—or **palette color**—scheme uses only a portion of these colors.



3.59

For example, **<u>a high-quality digital camera uses almost three million pixels for a 3 × 5 inch photo</u>**. The following shows the number of bits that need to be stored using each scheme:

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

3.60

Standards for Image Encoding

Several de facto standards for image encoding are in use.

JPEG (**Joint Photographic Experts Group**) uses the True-Color scheme, but compresses the image to reduce the number of bits (see Chapter 15).

GIF (**Graphic Interchange Format**), on the other hand, uses the indexed color scheme.

3.61

Vector Graphics

Raster graphics has two disadvantages: the file size is big and rescaling is troublesome. To enlarge a raster graphics image means enlarging the pixels, so the image looks ragged when it is enlarged. The **vector graphic** image encoding method, however, does not store the bit patterns for each pixel. An image is decomposed into a combination of geometrical shapes such as lines, squares or circles.
For example, consider a circle of radius r. The main pieces of information a program needs to draw this circle are:

1. The radius *r* and equation of a circle.
2. The location of the center point of the circle.
3. The stroke line style and color.
4. The fill style and color.

3.62

3-6 Storing Video

Video is a representation of images (called **frames**) over time. A movie consists of a series of frames shown one after another. In other words, video is the representation of information that changes in space and in time. So, if we know how to store an image inside a computer, we also know how to store video: each image or frame is transformed into a set of bit patterns and stored. The combination of the images then represents the video.

3.63