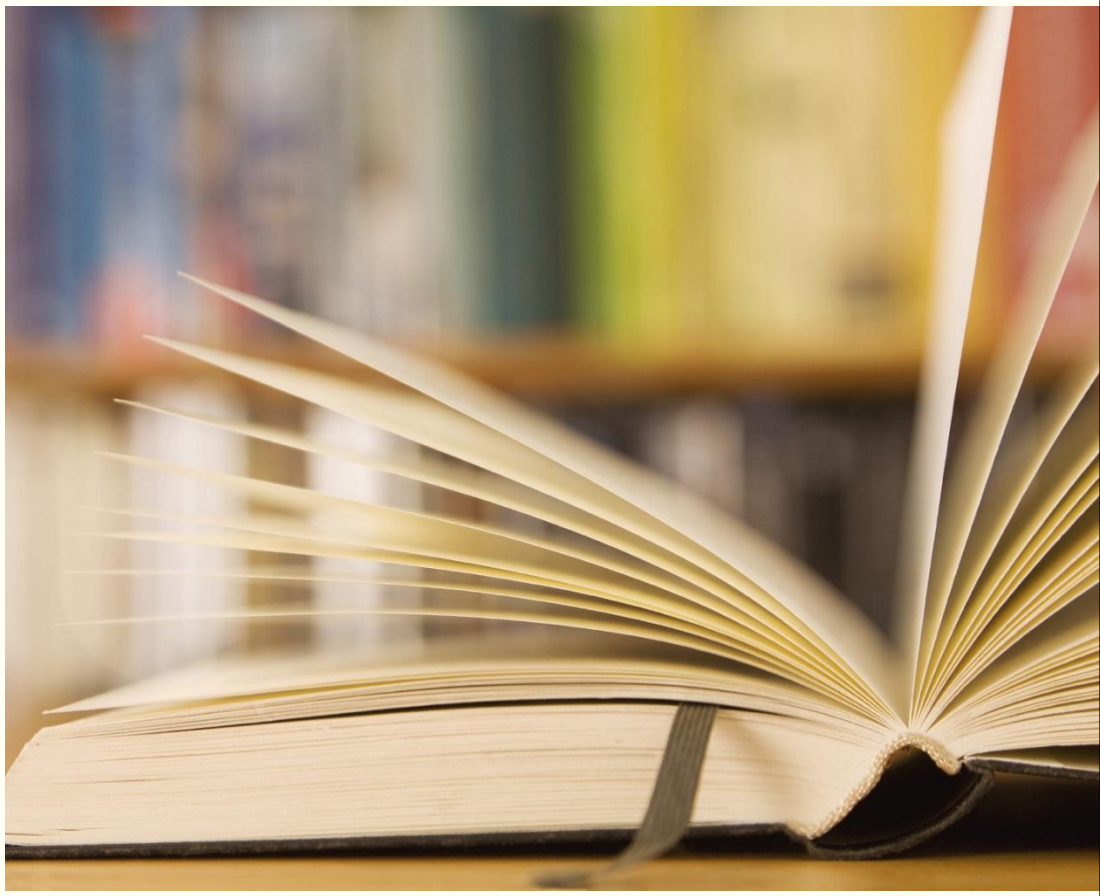
# 3.1 DATA STORAGE



# Content

- 3.1 Data types
- 3.2 Storing Numbers
- 3.3 Storing Text, Media, Image, Video

# Objectives

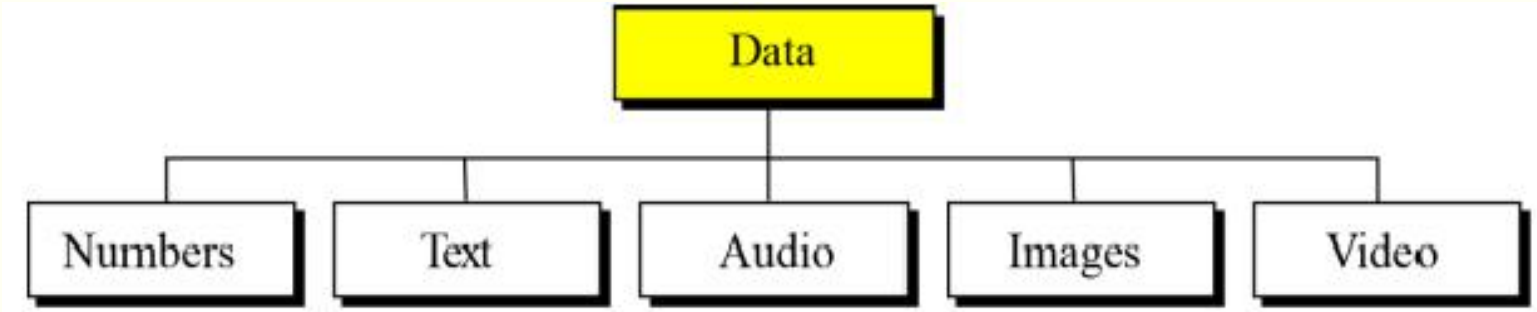
**After studying this chapter, the student should be able to:**

- 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.

# 1-DATA TYPES

# 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.**

Figure -3.1 Types of data types

# 2. 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**.

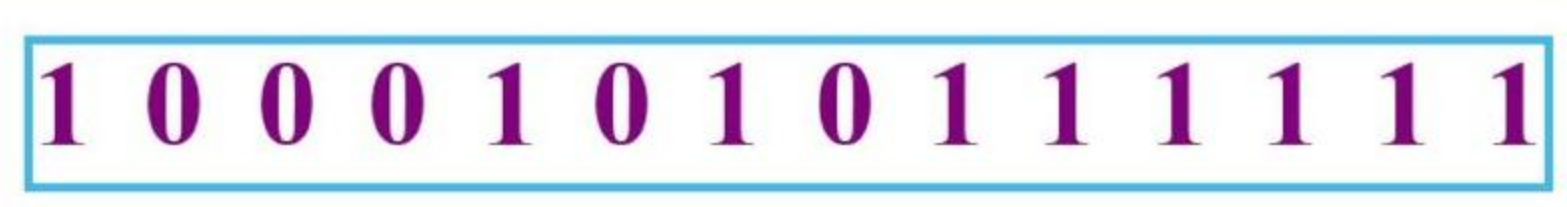
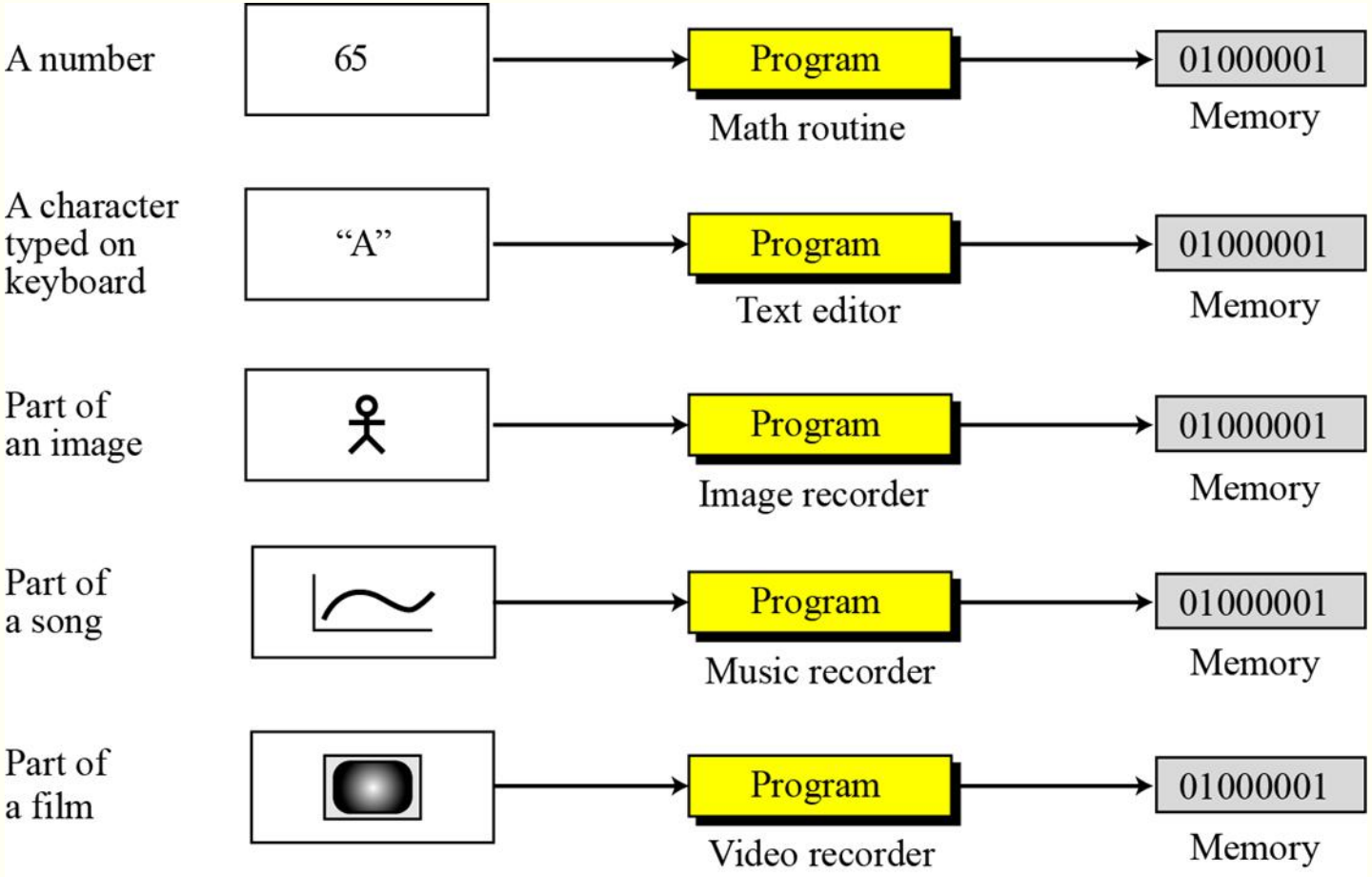



Figure -3.2 A bit pattern

- ***Bits*** (binary digit) : The symbol 0 or 1 is called a **bit** (binary digit), the smallest unit of data that can be stored in a computer.
- ***Bits patterns*** a sequence, or a string of bits. A bit patterns with 8 bits is called a **byte**

# 3. Storage of Different Data Types



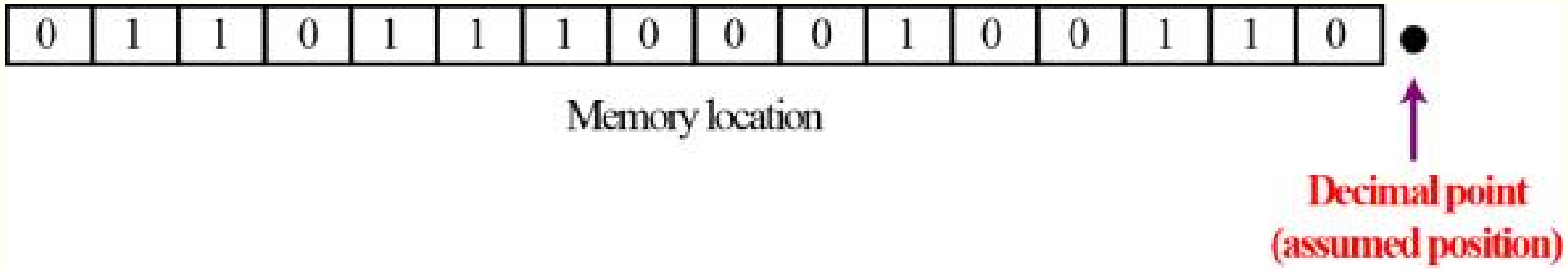
# 2 - STORING NUMBERS

# 1. Introduction

- 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.
- There are several ways to handle the sign issue, discussed later in this chapter. For the decimal point, computers use two different representations: fixed-point and floating point.
- The first is used to store a number as an integer—without a fractional part, the second is used to store a number as a real—with a fractional part.

# 2. 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.



# 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.

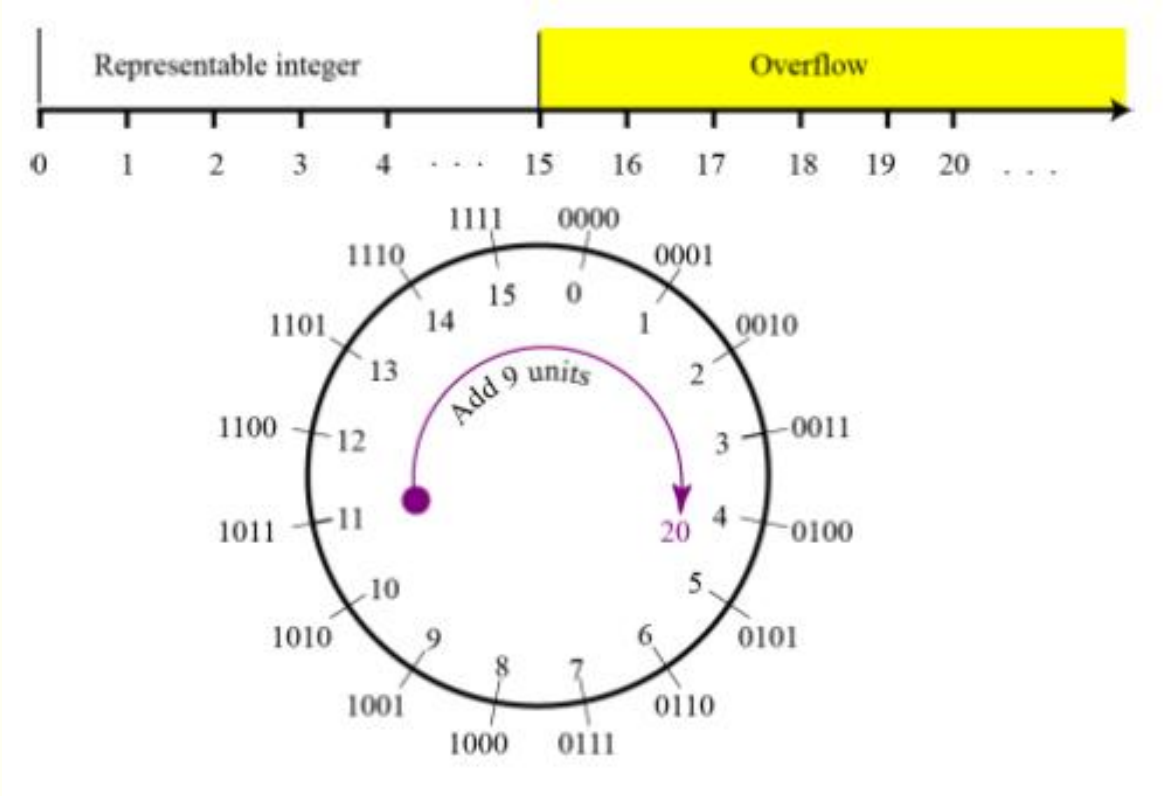
**Solution**

- First change the integer to binary, (111)2.
- Add five 0s to make a total of eight bits, (00000111)2. The integer is stored i

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

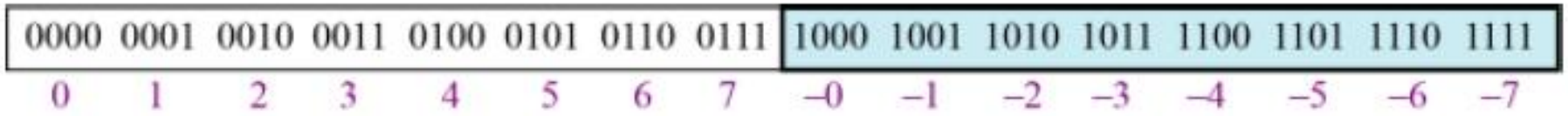
# Overflow in unsigned integers

- What happens if we try to store an integer that is larger than 24 – 1 = 15 in a memory location that can only hold four bits.



# Sign-and-Magnitude Representation

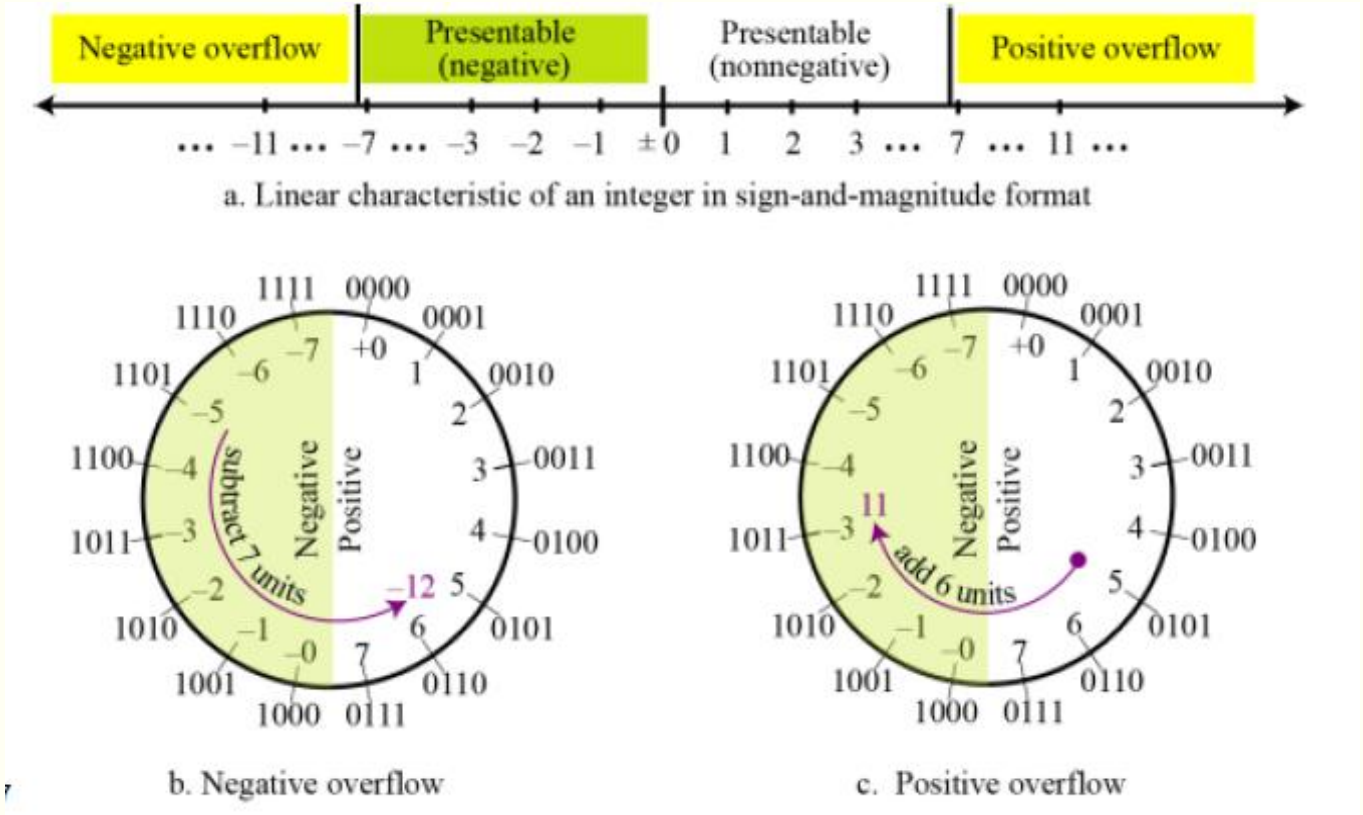
- In this method, the available range for **unsigned integers (0 to $2n-1$)** is divided into two equal sub-ranges. **The first half represents positive integers**, **the second half, negative integers**.
- Noted that there are **two different representations for zero**.




**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.**

# Overflow in sign-and-magnitude representation

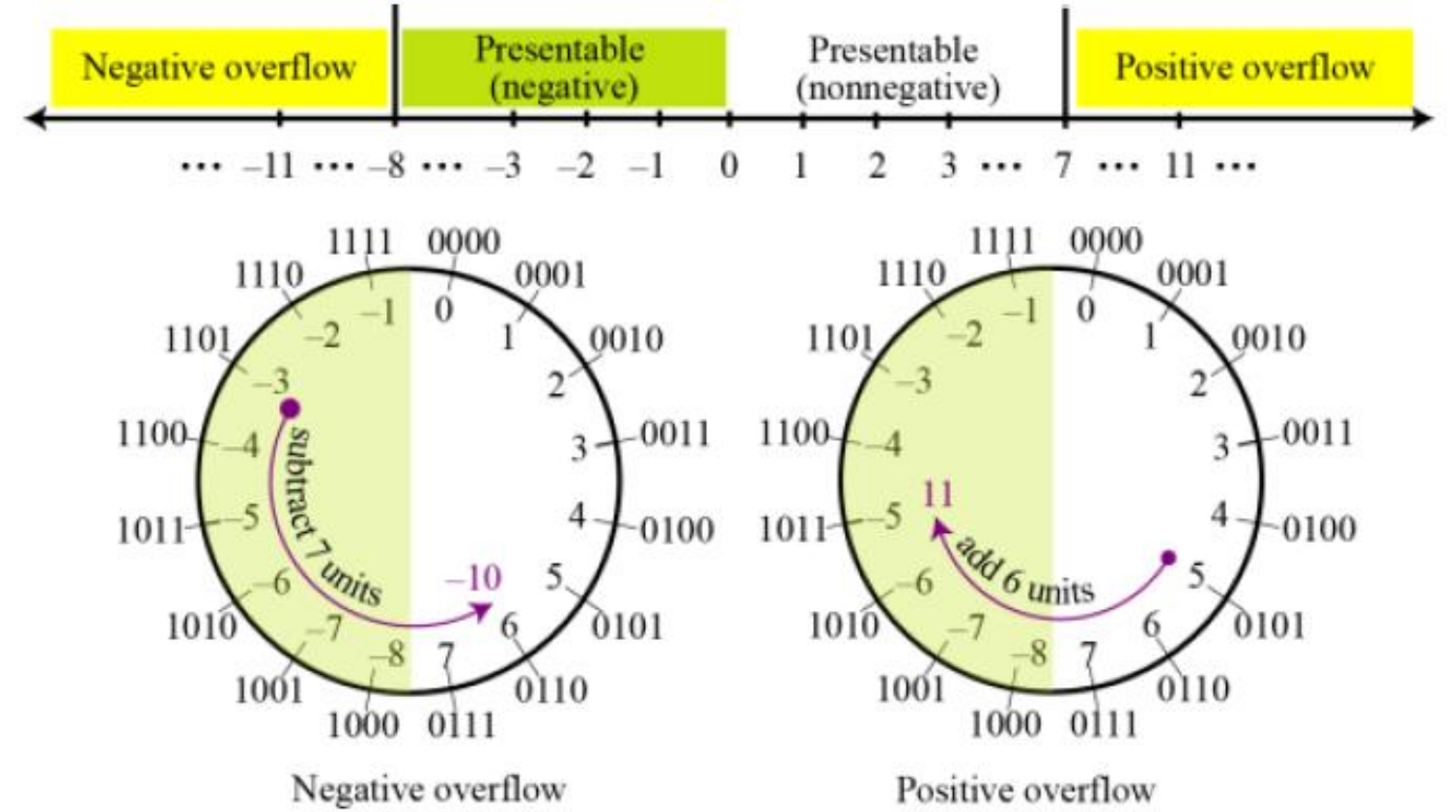
- Both **positive and negative overflow** when storing an integer in sign-and-magnitude representation using a 4-bit memory location.



# 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

# Overflow in two's complement representation




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

# 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 exa

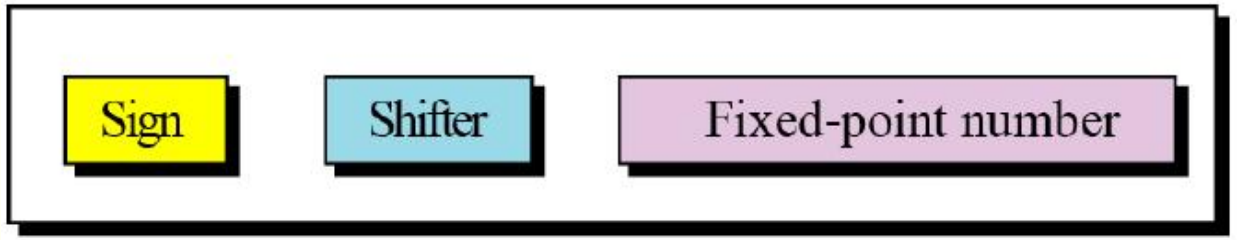
**Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.**

*Example 3.1*

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**.

# Floating-Point Representation for a Real number

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




**Example 3.2** 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 | → | + | $7.425 \times 10^{21}$ |

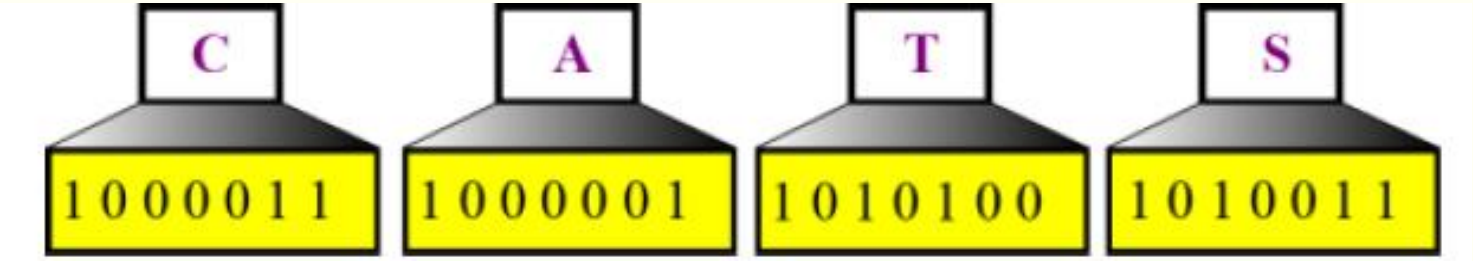
# 3 - STORING TEXT, MEDIA, IMAGE, VIDEO

# 3.1 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.

**Representing symbols using bit patterns**

- 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.



# Codes

- **ASCII** (American Standard Code of Information Interchange):
- **ASCII**: Using 7 bits for each symbol
- **Extended ASCII**: Using 8 bits for each symbol
- Using 32 bits to represent up to 232 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

**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 |

# 3.2 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.

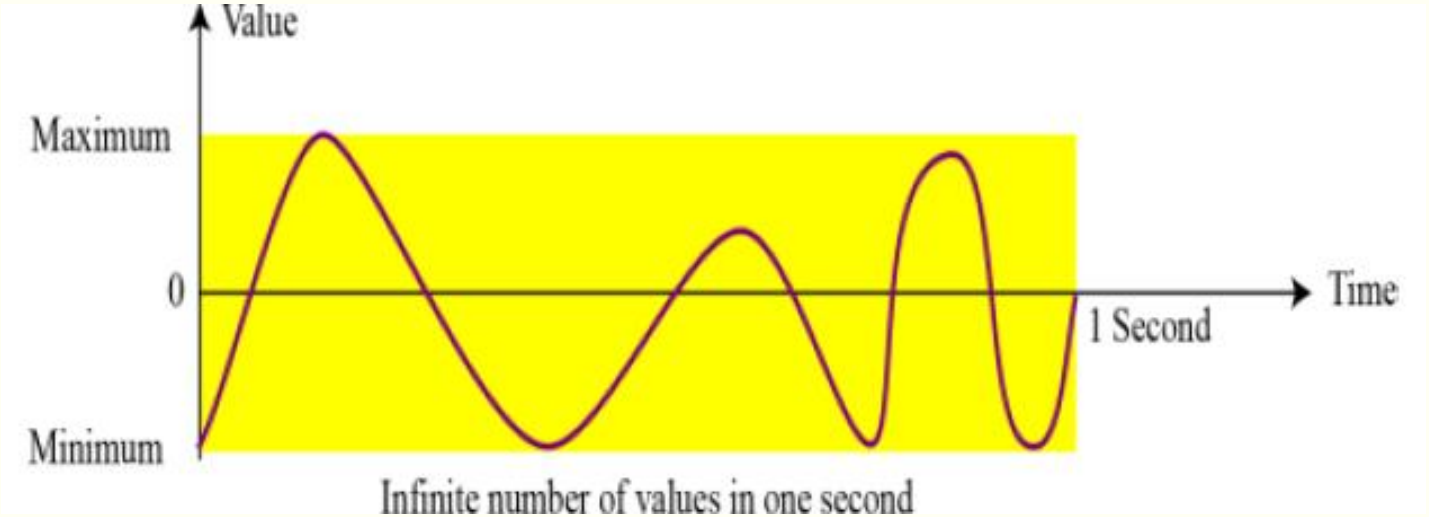



Figure -3.3 an example of analog data

# 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**.

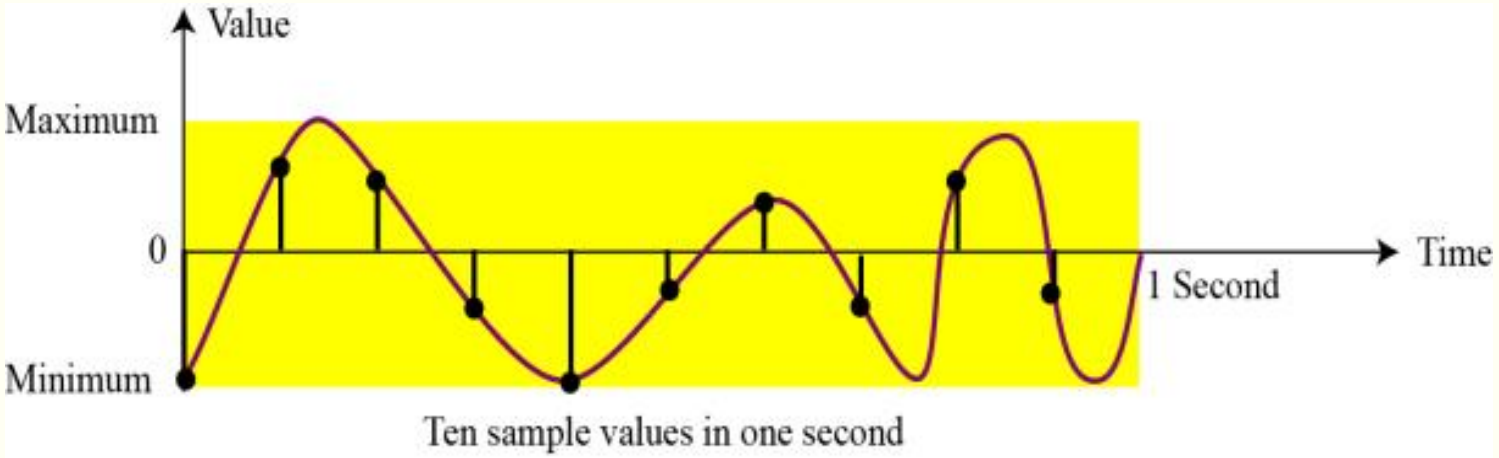



Figure -3.4 an example of sampling

# 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.

# 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**

$$\text{Bit rate} = 40{,}000 \times 16 = 640{,}000 \text{ bits per second}$$

# 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.

# 3.3 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**).

- **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**.

- **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

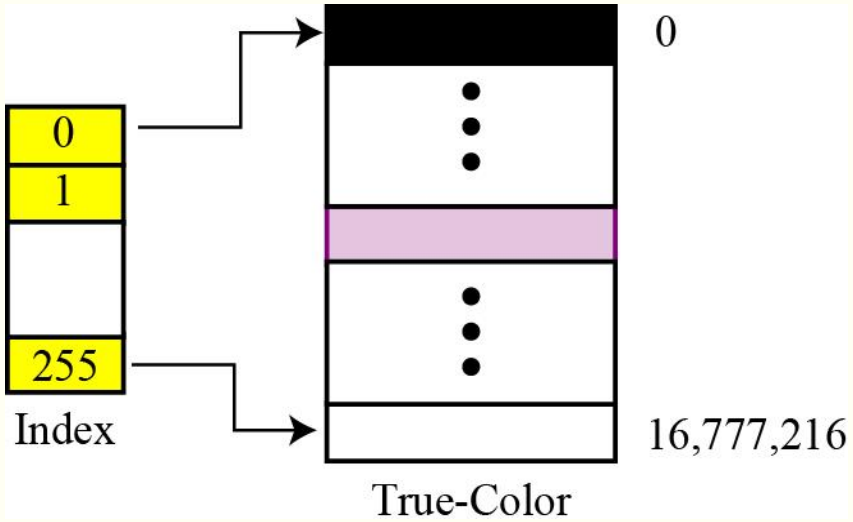
# Color

- ***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 |

- ***Indexed Color*** The **indexed color**—or **palette color—**scheme uses only a portion of these colors



# 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**.
- **Softwares**: **Photoshop, PhotoImpact, Corel Painter**

# Vector Graphics

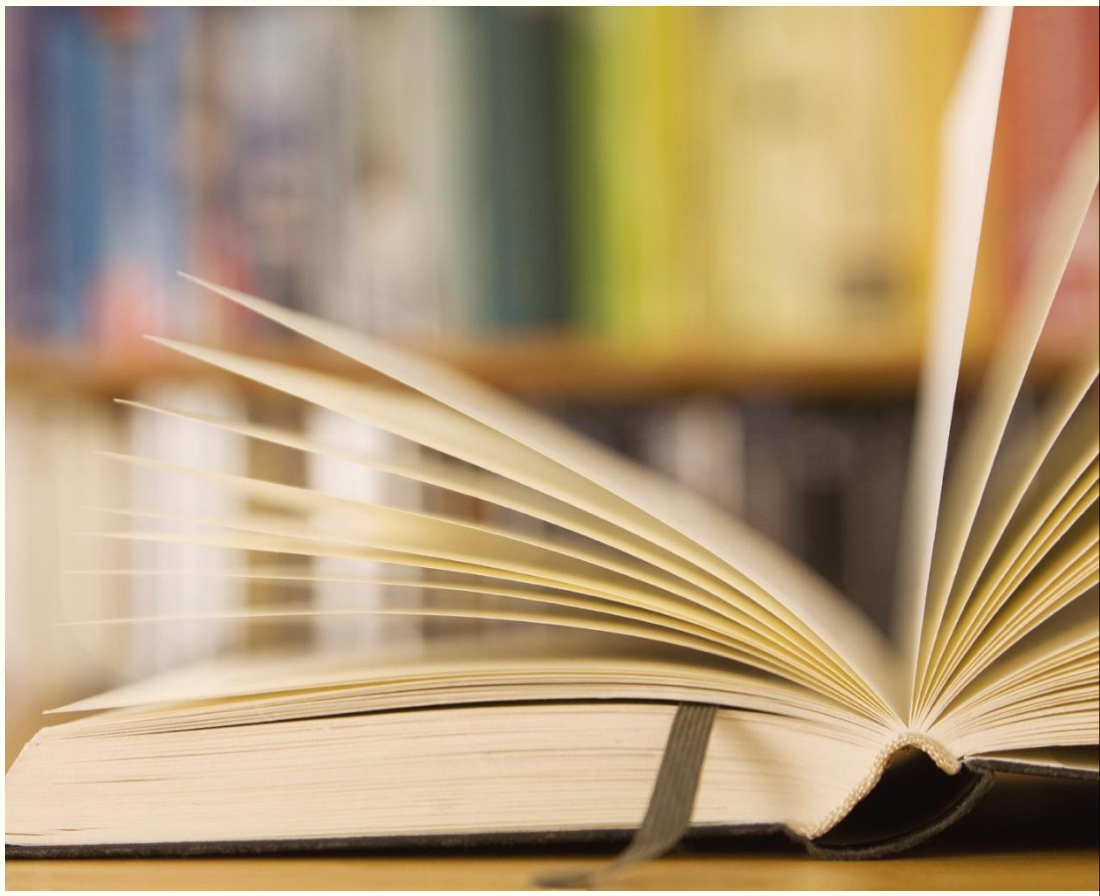
- **Raster graphics** has two disadvantages: **the file size is big and rescaling is troublesome**. To enlarge a raster graphics image means enlarging the pixels, so the **image looks ragged when it is enlarged**. The **vector graphic** image encoding method, however, does not store the bit patterns 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.**
- **Standards for Vector Graphis** EPS, WMF, AI, CDR
- **Softwares**: **Illustrator, CorelDRAW, Flash**

# 3.4 Storing Video

- **Video is a representation of images (called frames) over time**.
- **A movie** consists of a series of frames shown one after another to create the illusion of motion.
- In other words, **video is the representation of information that changes in space (single image) and in time (a series of images).**

# 3.2 OPERATIONS ON DATA

Subtitle



# Content

- 4.1 Logic
- 4.2 Shift
- 4.3 Arithmetic operations

# Objectives

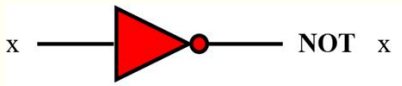
**After studying this chapter, the student should be able to:**

- List the three categories of operations performed on data.
- Perform unary and binary logic operations on bit patterns.
- Distinguish between logic and arithmetic shift operations.
- Perform addition and subtraction on integers when they are stored in two's complement format.
- Perform addition and subtraction on integers when stored in sign-and-magnitude format.
- Perform addition and subtraction operations on reals stored in floating-point format.

# 1-LOGIC OPERATIONS

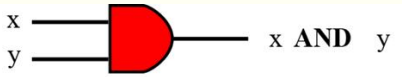
# 1. Introduction

- **Logic operations** refer to those operations that apply the same basic operation on **individual bits of a pattern**, or on **two corresponding bits in two patterns**.
- A logic operation at the **pattern level** is *n* logic operations, of the same type, at the **bit level** where *n* is the number of bits in the pattern.
- A bit can take one of the two values: 0 or 1. If we interpret **0 as the value *false*** and **1 as the value *true***,



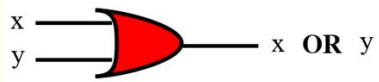

NOT

| x | NOT x |
|---|---|
| 0 | 1 |
| 1 | 0 |



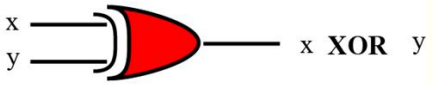

AND

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




OR

| x | y | x OR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |




XOR

| x | y | x XOR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# 2. The NOT operator

- **The NOT operator** is a **unary operator**: it takes only one input. The output bit is the complement of the input.


NOT

| x | NOT x |
|---|---|
| 0 | 1 |
| 1 | 0 |


AND

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Example 3.3** Use the **NOT** operator on the bit pattern 10011000.
**Solution** The solution is shown below. Note that the NOT operator changes every 0 to 1 and every 1 to 0.

| NOT | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Input |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Output |

# 3. The AND operator

- **The AND operator** is a **binary operator**: it takes two inputs. The output bit is 1 if both inputs are 1s and the output is 0 in the other three cases.

$$x \text{ AND } 0 \rightarrow 0$$

**Example 3.4** Use the **AND** operator on the bit patterns 10011000 and 00101010.
**Solution** The solution is shown below. Note that only one bit in the output is 1, where both corresponding inputs are 1s.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Input 1 |
| AND | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Input 2 |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Output |

# 4. The OR operator

- **The OR operator** is a binary operator: it takes two inputs. The output bit is 0 if both inputs are 0s and the output is 1 in other three cases.

$$x \text{ OR } 1 \rightarrow 1$$

**Example 3.5** Use the **OR** operator on the bit patterns 10011001 and 00101110.

**Solution** The solution is shown below. Note that only one bit in the output is 0, where both corresponding inputs are 0s.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | Input 1 |
| OR | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Input 2 |
| | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | Output |

# 5. The XOR operator

- **The XOR operator** is a binary operator like the OR operator, with only one difference: the output is 0 if both inputs are 1s.

$$x \text{ XOR } 1 \rightarrow \text{NOT } x$$

**Example 3.6** Use the **XOR** operator on the bit patterns 10011001 and 00101110.

**Solution**

The solution is shown below. Compare the output in this example with the one in Example 4.5. The only difference is that when the two inputs are 1s, the result is 0 (the effect of exclusion).

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | Input 1 |
| XOR | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | Input 2 |
| | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | Output |

# 2 - SHIFT OPERATIONS

# 1. Introduction

- **Shift operations** move the bits in a pattern, changing the positions of the bits. They can move bits to the left or to the right.
- We can divide shift operations into two categories: **logical shift operations** and **arithmetic shift operations.**
- **Logical Shift Operations** A logical shift operation is applied to a pattern **that does not represent a signed number**.
- The reason is that these shift operations may change the sign of the number that is defined by the leftmost bit in the pattern. We distinguish two types of logical shift operations, as described below:
  - ❑**Logical shift**
  - ❑**Logical circular shift (Rotate)**

# 2. Logical Right/Left Shift Operations

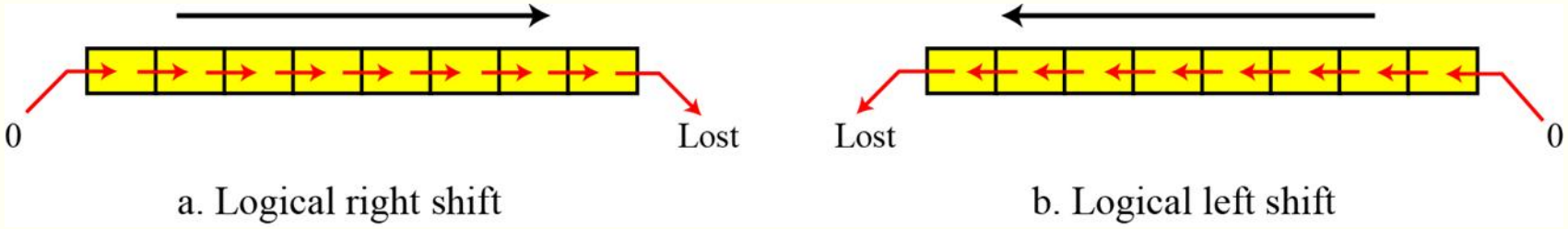


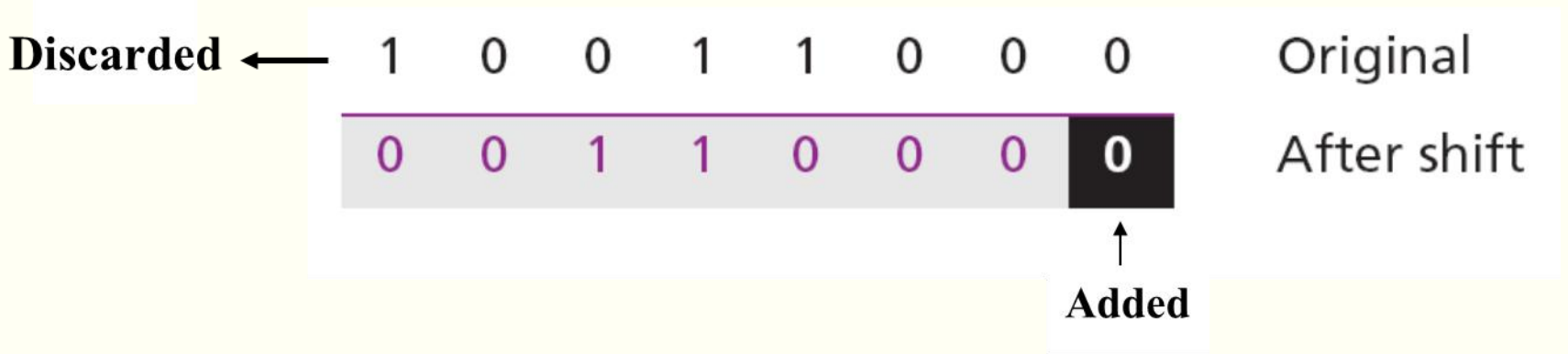

**Figure 3.5** Logical shift operations

*Example 3.7*

Use a logical left shift operation on the bit pattern 10011000.

*Solution*

The leftmost bit is lost and a 0 is inserted as the rightmost bit.



# 3. Logical Right/Left Circular Shift Operations

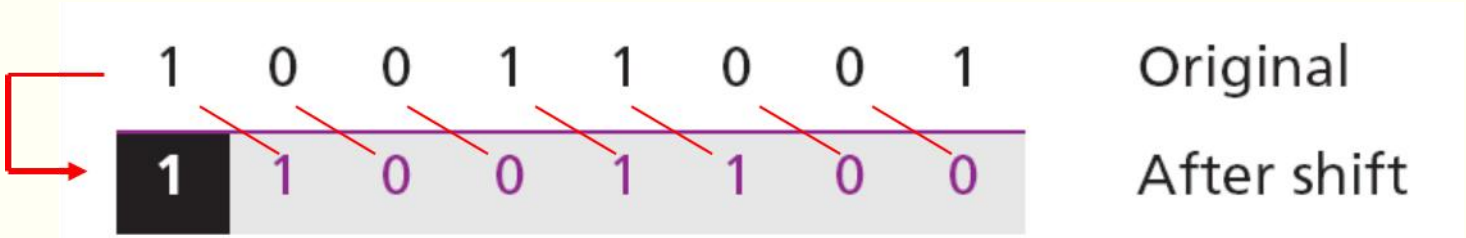

**Figure 3.6** Logical shift operations

**Example 3.8**

Use a circular left shift operation on the bit pattern 10011000.

**Solution**

The solution is shown below. The leftmost bit is circulated and becomes the rightmost bit.



# 4. Arithmetic Shift Operations

- Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format. **Arithmetic right shift is used to divide an integer by two**, while **arithmetic left shift is used to multiply an integer by two.**

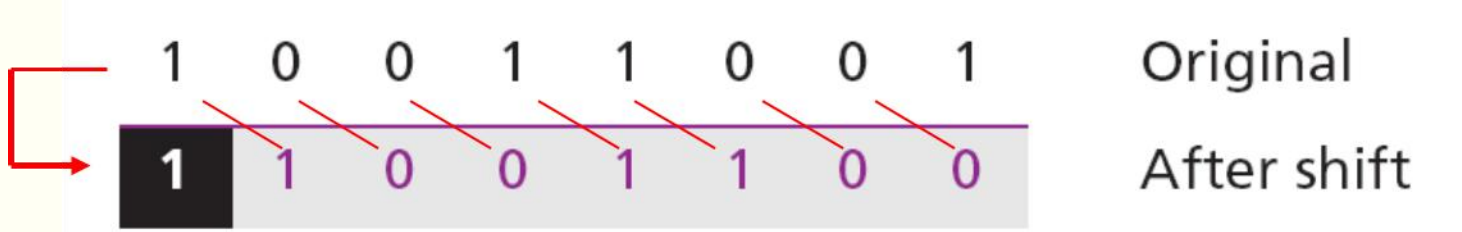

**Figure 3.7** **Arithmetic shift operations**

**Example 3.9**

Use an arithmetic right shift operation on the bit pattern 10011001. The pattern is an integer in two's complement format.

**Solution**

The solution is shown below. The leftmost bit is retained and also copied to its right neighbor bit.



# 3 - ARITHMETIC OPERATIONS

# Introduction

Arithmetic operations involve **adding, subtracting, multiplying and dividing**. We can apply these operations to **integers** and **floating-point numbers**.

## Arithmetic Operations on Integers

- All arithmetic operations such as **addition, subtraction, multiplication and division** can be applied to integers. Although multiplication (division) of integers can be implemented using repeated addition (subtraction), the procedure is not efficient.

## Two's Complement Integers

- **When the subtraction operation is encountered, the computer simply changes it to an addition operation** but makes two's complement of the second number. In ot

$$A - B \leftrightarrow A + (\overline{B} + 1)$$

Where $\overline{B}$ is the one's complement of B and

$(\overline{B} + 1)$ means the **two's complement of B**

# Addition and subtraction of integers in two's complement

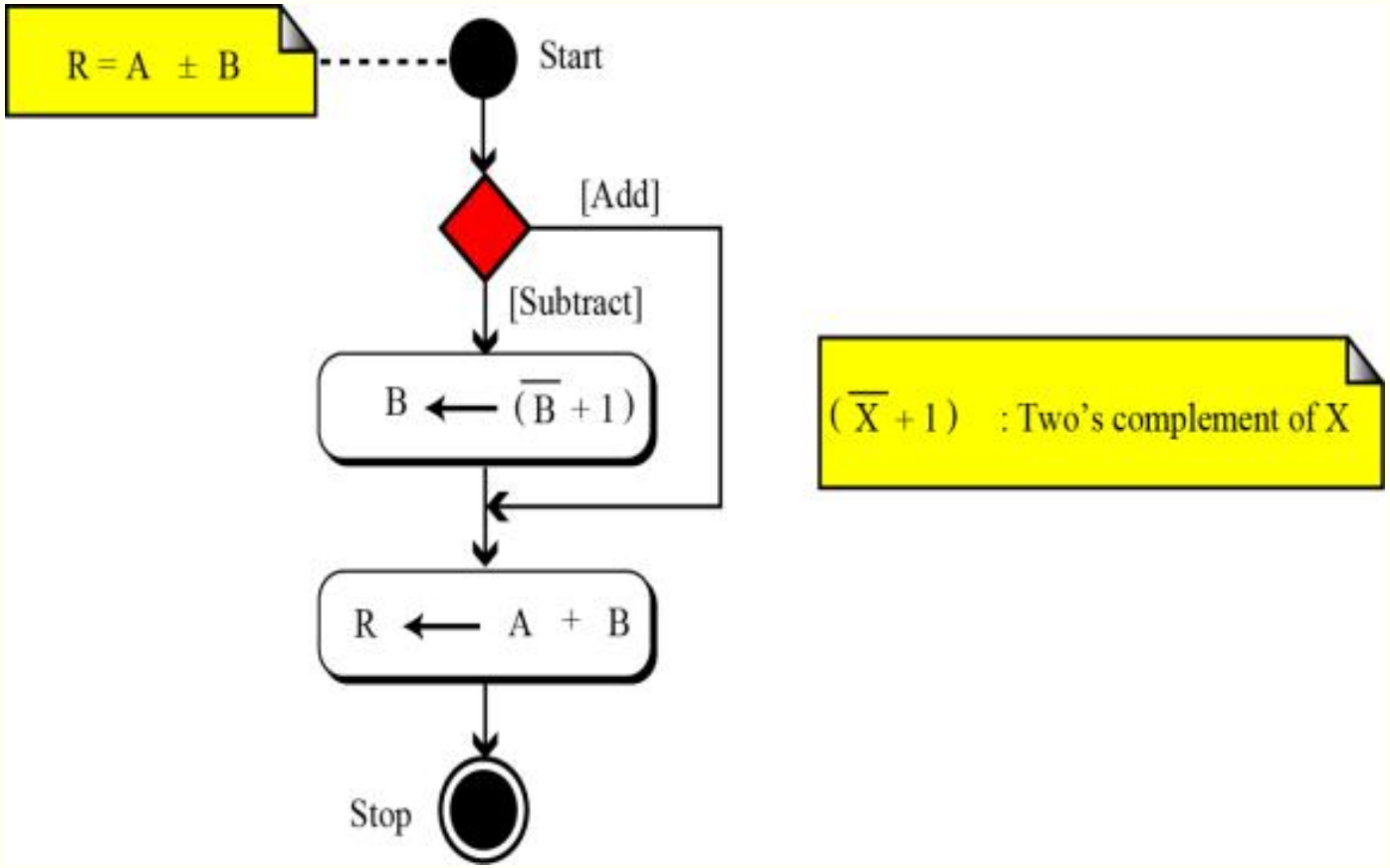



**Figure 3.8** Addition and subtraction of integers

# Addition and subtraction of integers in two's complement (cont)

## Example 3.10

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$\mathbf{A = (00010001)_2} \qquad \mathbf{B = (00010110)_2}$$

## Solution

The operation is adding. A is added to B and the result is stored in R. (+17) + (+22) = (+39).

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | | | | | | Carry |
| | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | A |
| + | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | B |
| | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | R |

# Addition and subtraction of integers in sign-and-magnitude

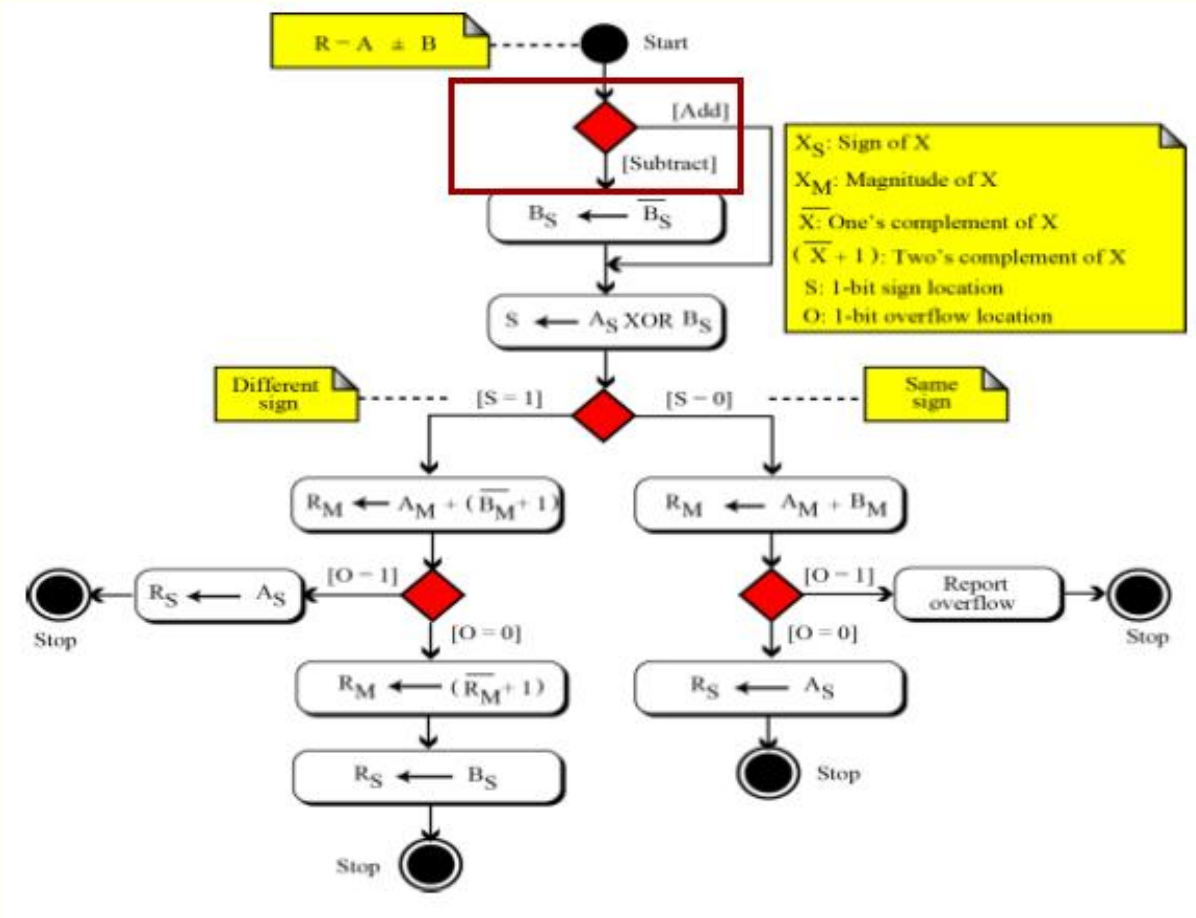



**Figure 3.8** Addition and subtraction of integers in sign-and-magnitude

# Addition and subtraction of integers in sign-and-magnitude (cont)

## Example 3.11

Two integers A and B are stored in sign-and-magnitude format. Show how B is subtracted from A.

$$\mathbf{A = (1\ 1010001)_2} \qquad \mathbf{B = (1\ 0010110)_2}$$

## Solution

The operation is subtracting: $S_B = S_B$. $S = A_S$ XOR $B_S = 1$, $R_M = A_M + (B_M + 1)$. Since there is an overflow, the value of $R_M$ is final. The sign of R is the sign of A. (−81) − (−22) = (−59).

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Overflow → | 1 | | | | | | | Carry |
| $A_S$ | 1 | | 1 | 0 | 1 | 0 | 0 | 0 | 1 | $A_M$ |
| $B_S$ | 1 | + | 1 | 1 | 0 | 1 | 0 | 1 | 0 | $(\overline{B}_M + 1)$ |
| $R_S$ | 1 | | 0 | 1 | 1 | 1 | 0 | 1 | 1 | $R_M$ |

# Arithmetic Operations on Reals

- All arithmetic operations such as addition, subtraction, multiplication and division can be applied to reals stored in floating-point format.
- Multiplication of two reals involves multiplication of two integers in sign-and-magnitude representation.
- Division of two reals involves division of two integers in sign-and-magnitude representations

**Addition and Subtraction of Reals**

- Addition and subtraction of real numbers stored in floating-point numbers is reduced to addition and subtraction of two integers stored in sign-and-magnitude (combination of sign and mantissa) after the alignment of decimal points. Figure 4.8 shows a simplified version of the procedure (there are some special cases that we have ignored).

# Addition and subtraction of reals in floating-point format

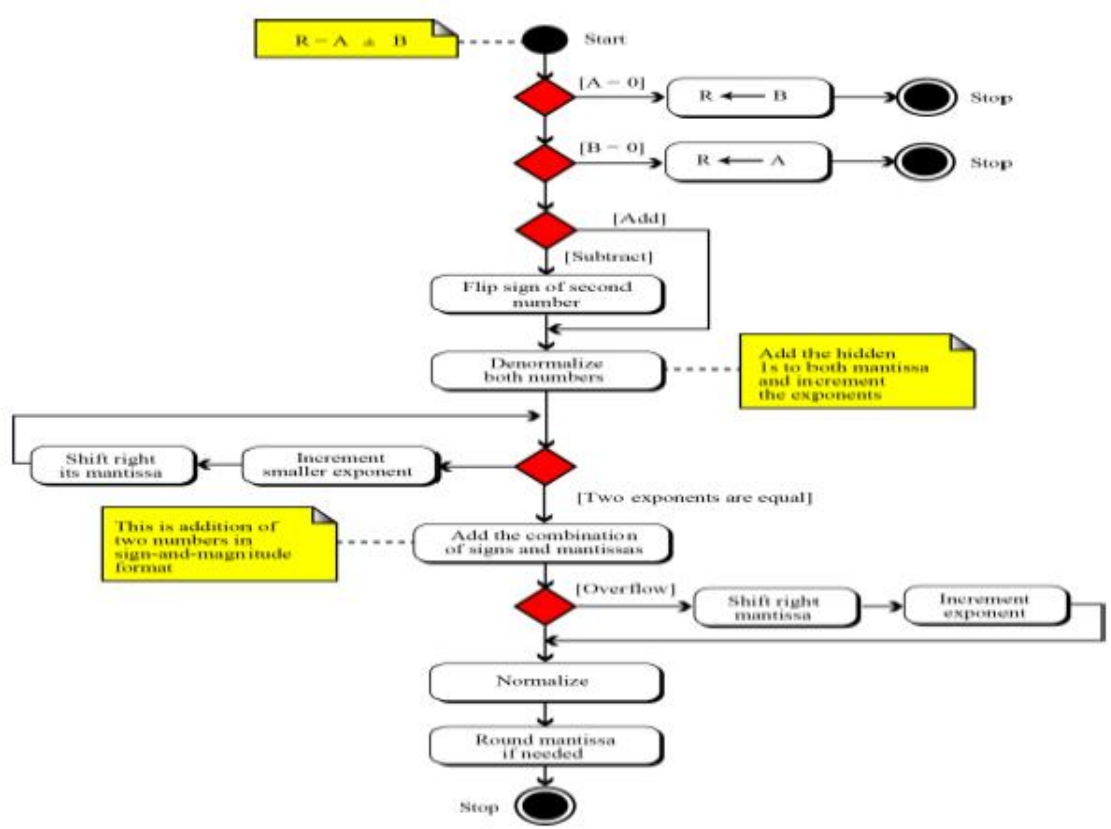



**Figure 3.9** Addition and subtraction of integers in floating-point format

# Addition and subtraction of reals in floating-point format (cont)

**Example 3.11**

Show how the computer finds the result of (+5.75) + (−7.0234375) = − 1.2734375.

**Solution**

These two numbers can be stored in floating-point format, as shown below:

| | S | E | M |
|---|---|---|---|
| A | 0 | 10000001 | 01110000000000000000000 |
| B | 1 | 10000001 | 11000001100000000000000 |

De-normalization results in:

| | S | E | Denormalized M |
|---|---|---|---|
| A | 0 | 10000010 | 10111000000000000000000 |
| B | 1 | 10000010 | 11100000110000000000000 |