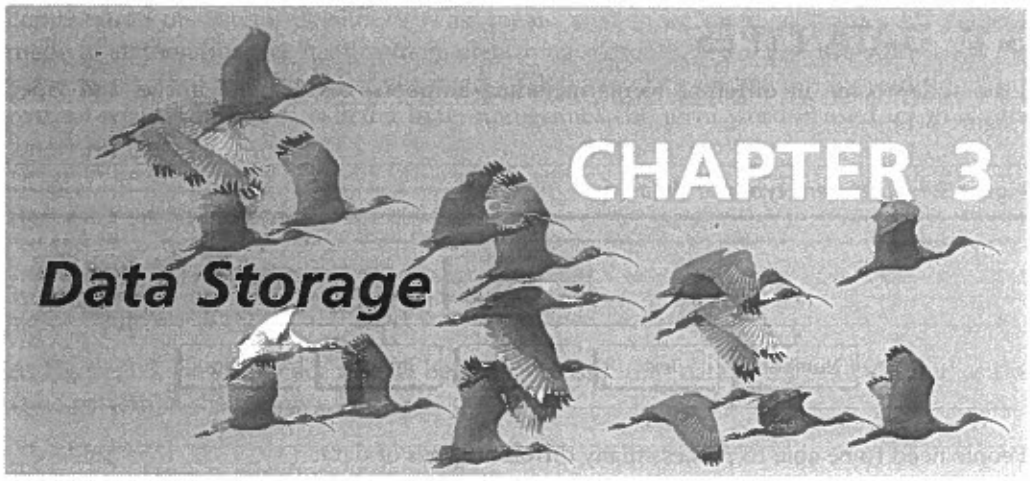


As discussed in Chapter 1, a computer is a programmable data processing machine. Before we can talk about processing data, we need to understand the nature of data. In this chapter we discuss different data types and how they are stored inside a computer. In Chapter 4, we show how data is manipulated inside a computer.

Objectives

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

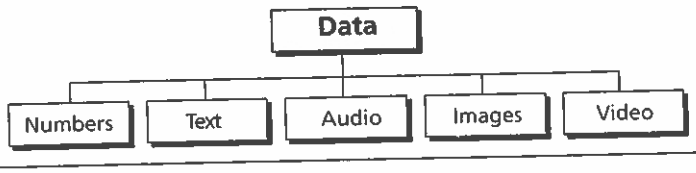
- List five different data types used in a computer.
- Describe how different data is stored inside the computer as bit patterns.
- Describe how integers are stored in a computer using unsigned format.
- Describe how integers are stored in a computer using sign-and-magnitude format.
- Describe how integers are stored in two's complement format.
- Describe how reals are stored in a computer using floating-point format.
- 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.

Open Rubric

3.1 DATA TYPES

Data today come in different forms including numbers, text, audio, image, and video (Figure 3.1).

Figure 3.1 *Different types of data*



People need to be able to process many different types of data:

- An engineering program uses a computer mainly to process numbers: to do arithmetic, to solve algebraic or trigonometric equations, to find the roots of a differential equation, and so on.
- A word processing program, on the other hand, uses a computer mainly to process text: justify, move, delete, and so on.
- A computer also handles audio data. We can play music on a computer and can records sound as data.
- An image processing program uses a computer to manipulate images: create, shrink, expand, rotate, and so on.
- Finally, a computer can be used not only to show movies, but also to create the special effects seen in movies.

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

3.1.1 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*, as discussed shortly.

Bits

A **bit (binary digit)** is the smallest unit of data that can be stored in a computer and has a value of 0 or 1. A bit represents the state of a device that can take one of two states. For example, a *switch* can be on or off. A convention can be established to represent the 'on' state as 1 and the 'off' state as 0, or *vice versa*. In this way, a switch can store one bit of information. Today, computers use various two-state devices to store data.

Bit patterns

To represent different types of data, we use a **bit pattern**, a sequence, or as it is sometimes called, a **string of bits**. Figure 3.2 shows a bit pattern made up of sixteen bits. It is a

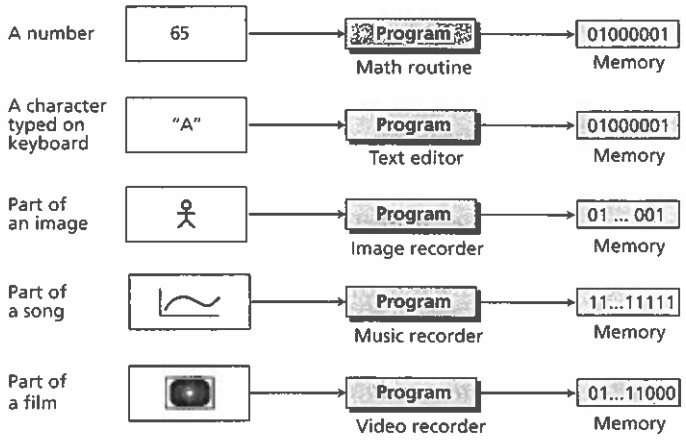
combination of sixteen 0s and 1s. This means that if we need to store a bit pattern made of sixteen bits, we need sixteen electronic switches. If we need to store 1000 bit patterns, each sixteen bits long, we need 16000 switches, and so on. By tradition a bit pattern with eight bits is called a **byte**. Sometimes the term word is used to refer to a longer bit pattern.

Figure 3.2 *A bit pattern*

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

As Figure 3.3 shows, a piece of data belonging to different data types can be stored as the same pattern in the memory.

Figure 3.3 *Storage of different data types*



If we are using a text editor (a word processor), the character A typed on the keyboard can be stored as the 8-bit pattern 01000001. The same 8-bit pattern can represent the number 65 if we are using a mathematical routine. In the same way, the same pattern can represent part of an image, part of a song, or part of a scene in a film. The computer's memory stores all of them without recognizing what type of data they represent.

3.1.2 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 this subject.

Data compression is discussed in Chapter 15.

3.1.3 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.

Error detection and correction is discussed in Appendix H.

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:

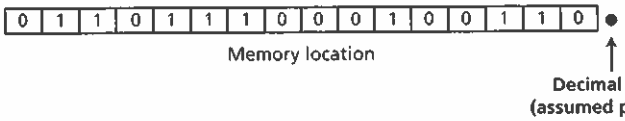
1. How to store the sign of the number.
2. How to show the decimal point.

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.

3.2.1 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 in Figure 3.4. In this representation the decimal point is assumed but not stored.

Figure 3.4 *Fixed point representation of integers*



However, a user (or a program) may store an integer as a real with the fractional part set to zero. This may happen, for example, if an integer is too large to be stored in the size defined for an integer. To use computer memory more efficiently, unsigned and signed integers are stored inside the computer differently.

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

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. However, since no computer can

possibly represent all the integers in this range, most computers define a constant called the *maximum unsigned integer*, which has the value of $(2^n - 1)$ where n is the number of bits allocated to represent an unsigned integer.

Storing unsigned integers

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 of the binary integer so that there is a total of n bits. If the number of bits is greater than n, the integer cannot be stored. A condition referred to as *overflow* will occur, which we discuss later.

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

Retrieving unsigned integers

An output device retrieves a bit string from memory as a bit pattern and converts it to an unsigned decimal integer.

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

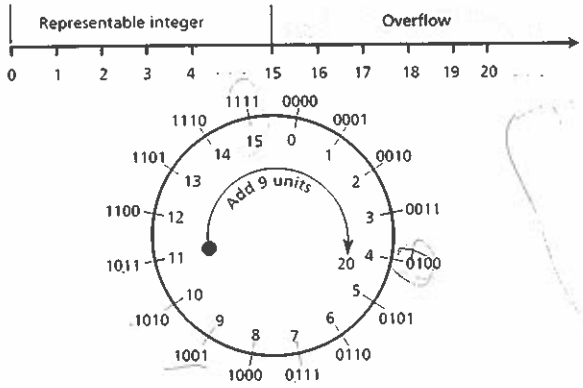
Using the procedure shown in Chapter 2, the binary integer is converted to the unsigned integer 43.

Overflow

Due to size limitations—the allocated number of bits—the range of integers that can be represented is limited. In an n-bit memory location we can only store an unsigned integer

between 0 and $2^n - 1$. Figure 3.5 shows 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. This situation, called **overflow**, happens when, for example, we have stored the integer 11 in a memory location and then try to add 9 to the integer. The minimum number of bits we need to represent the decimal 20 is five bits. In other words, $20 = (10100)_2$, so the computer drops the leftmost bit and keeps the rightmost four bits $(0100)_2$. People are surprised when they see that the new integer is printed as 4 instead of 20. Figure 3.5 shows why this happens.

Figure 3.5 *Overflow in unsigned integers*



Applications of unsigned integers

Unsigned integer representation can improve the efficiency of storage because we do not need to store the sign of the integer. This means that the entire bit allocation can be used for storing the number. Unsigned integer representation can be used whenever we do not need negative integers. The following lists some cases:

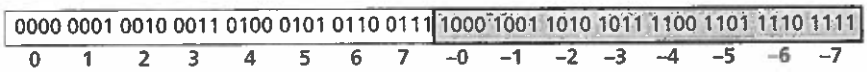
- **Counting**. When we count, we do not need negative numbers. We start counting from 1 (sometimes 0) and go up.
- **Addressing**. Some computer programs store the address of a memory location inside another memory location. Addresses are positive integers starting from 0 (the first memory location) and going up to an integer representing the total memory capacity. Here again, we do not need negative integers—unsigned integers can easily do the job.
- **Storing other data types**. Other data types (text, images, audio, and video), as we will discuss shortly, are stored as bit patterns, which can be interpreted as unsigned integers.

Sign-and-magnitude representation

Although the **sign-and-magnitude representation** format is not commonly used to store integers, this format is used to store part of a real number in a computer, as described in the next section. For this reason we briefly discuss this format here. In this method, the available range for unsigned integers (0 to $2^n - 1$) is divided into two equal subranges. The first half represents positive integers, the second half, negative integers. For example, if n is 4, the range is 0000 to 1111. This range is divided into two halves: 0000 to 0111 and 1000 to 1111 (Figure 3.6). The bit patterns are then assigned to negative and positive integers.

Note that the negative numbers appear to the right of the positive numbers, which is contrary to conventional thinking about positive and negative numbers. Also note that we have two 0s: positive zero (0000) and negative zero (1000).

Figure 3.6 *Sign-and-magnitude representation*



Storing an integer in sign-and-magnitude format requires 1 bit to represent the sign (0 for positive, 1 for negative). This means that in an 8-bit allocation, we can only use seven bits to represent the absolute value of the number (number without the sign). Therefore, the maximum positive value is one half the unsigned value. The range of numbers that can be stored in an n-bit location is $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. In an n-bit allocation, the leftmost bit is dedicated to store the sign (0 for positive, 1 for negative).

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.

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

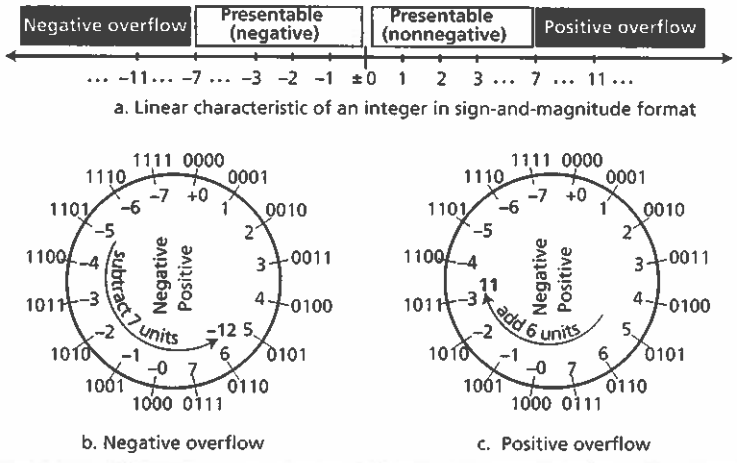
Since the leftmost bit is 1, the sign is negative. The rest of the bits (0100001) are changed to decimal as 33. After adding the sign, the integer is –33.

Overflow in sign-and-magnitude representation

Like unsigned integers, signed integers are also subjected to overflow. However, in this case, we may have both positive and negative overflow. Figure 3.7 shows both positive and negative overflow when storing an integer in sign-and-magnitude representation using a 4-bit memory location. Positive overflow occurs when we try to store a positive integer larger than 7. For example, assume that we have stored integer 5 in a memory location and we then try to add 6 to the integer. We expect the result to be 11, but the computer's response is –3. The reason is that if we start from 5 on a circular representation and go six units in the clockwise direction, we end up at –3. A positive overflow wraps the integer back to the range.

A negative overflow can happen when we try to store a integer that is less than –7, for example if we have stored the integer –5 in a memory and try to subtract 7 from it. We expect the result to be –12, but the computer's response is +6. The reason is that if we start from –5 on a circular representation and go seven units in the counterclockwise direction, we end up at +6.

Figure 3.7 *Overflow in sign-and-magnitude representation*



There are two 0s in sign-and-magnitude representation: +0 and –0.

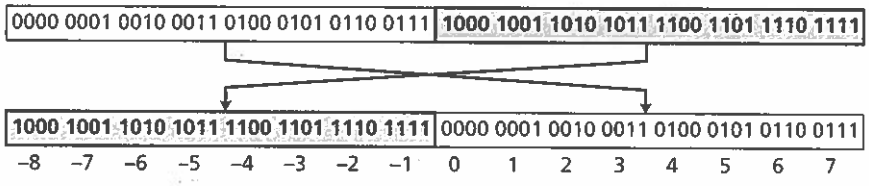
Applications of sign-and-magnitude representation

Sign-and-magnitude representation is not used to store integers. However, it is used to store part of real numbers, as we will see shortly. In addition, sign-and-magnitude representation is often used when we quantize an analog signal, such as audio.

Two's complement representation

Almost all computers use **two's complement representation** to store a signed integer in an n-bit memory location. In this method, the available range for an unsigned integer of (0 to $2^n - 1$) is divided into two equal subranges. The first subrange is used to represent nonnegative integers, the second half to represent negative integers. For example, if n is 4, the range is 0000 to 1111. This range is divided into two halves: 0000 to 0111 and 1000 to 1111. The two halves are swapped to be in agreement with the common convention of showing negative integers to the left of positive integers. The bit patterns are then assigned to negative and nonnegative (zero and positive) integers as shown in Figure 3.8.

Figure 3.8 *Two's complement representation*



Although the sign of the integer affects every bit in the binary integer stored, the first (leftmost) bit determines the sign. If the leftmost bit is 0, the integer is nonnegative: if the leftmost bit is 1, the integer is negative.

> 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.

Two operations

Before we discuss this representation further, we need to introduce two operations. The first is called *one's completing* 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

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

The second operation is called *two's completing* 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

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 (see Chapter 4 for binary addition).

Storing an integer in two's complement format

To store an integer in two's complement representation, the computer follows the steps below:

- The integer is changed to an n-bit binary.
- If the integer is positive or zero, it is stored as it is: if it is negative, the computer takes the two's complement of the integer and then stores it.

Retrieving an integer in two's complement format

To retrieve an integer in two's complement representation, the computer follows the steps below:

- If the leftmost bit is 1, the computer applies the two's complement operation to the integer. If the leftmost bit is 0, no operation is applied.
- The computer changes the integer to decimal.

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 three 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

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.								+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.								62
Sign is added.								–62

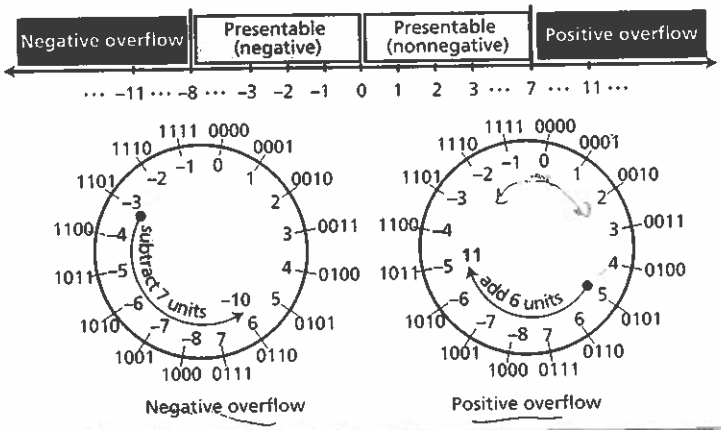
A very interesting point about two's complement is that there is only one zero in this representation. In sign-and-magnitude representation, there are two zeros (+0 and –0).

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

Overflow in two's complement notation

Like other representations, integers stored in two's complement format are also subject to overflow. Figure 3.9 shows both positive and negative overflow when storing a signed integer in a 4-bit memory location. Positive overflow occurs when we try to store a positive integer larger than 7. For example, assume that we have stored an integer value 5 in a memory location and we then try to add 6 to the integer. We expect the result to be 11, but the computer's response is –5. The reason is if we start from 5 on the circular representation and move six units in the clockwise direction, we end up at –5. The positive overflow wraps the integer back to the range.

Figure 3.9 *Overflow in two's complement representation*



A negative overflow can happen when we try to store a integer that is less than –8, for example if we have stored –3 and try to subtract 7 from it. We expect the result to be –10, but the computer's response is +6. The reason is that if we start from –3 on a circular representation and go seven units in the counterclockwise direction, we end up at +6.

Applications of two's complement notation

Two's complement representation is the standard representation for storing integers in computers today. In the next chapter we will see why this is the case when we see the simplicity of operations using two's complement.

3.2.2 Comparison of the three systems

Table 3.1 shows a comparison between unsigned, two's complement, and sign-and-magnitude integers. A 4-bit memory location can store an unsigned integer between 0 and 15, and the same location can store two's complement signed integers between –8 and +7. It is very important that we store and retrieve an integer in the same format. For example, if the integer 13 is stored in signed format, it needs to be retrieved in signed format: the same integer is retrieved as –3 in two's complement format.

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

3.2.3 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 23 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.

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.

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

Floating-point representation

The solution for maintaining accuracy or precision is to use **floating-point representation**. This representation allows the decimal point to float: we can have different numbers of digits to the left or right of the decimal point. The range of real numbers that can be stored using this method increases tremendously: numbers with large integral parts or small fractional parts can be stored in memory. In floating-point representation, either decimal or binary, a number is made up of three sections, as shown in Figure 3.10.

Figure 3.10 *The three parts of a real number in floating-point representation*

Floating-point representation

The first section is the sign, either positive or negative. The second section shows how many places the decimal point should be shifted to the right or left to form the actual number. The third section is a fixed-point representation in which the position of the decimal is fixed.

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

Floating-point representation is used in science to represent very small or very large decimal numbers. In this representation, which is called **scientific notation**, the fixed-point section has only one digit to the left of the decimal point and the shifter is the power of 10.

Example 3.18

The following shows the decimal number 7,425,000,000,000,000,000,000.00 in scientific notation (floating-point representation).

Solution

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. We can easily see the advantage of this. Even if we just want to write the number on a piece of paper, the scientific notation is shorter and takes less space. The notation uses the concept of floating-point because the position of the decimal point, which is near the right-hand end in the example, has moved 21 digits to the left to make the fixed-point part of the number. Some programming languages and calculators shows the number as +7.425E21 because the base 10 is understood and does not need to be mentioned.

Example 3.19

Show the number –0.0000000000000232 in scientific notation.

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}

Note that the exponent is negative here because the decimal point in 2.32 needs to move to the left (fourteen positions) to form the original number. Again, we can say that the number in this notation is made of three parts: sign (–), the real number (2.32), and the negative integer (–14). Some programming languages and calculators show this as –2.32E–14.

Similar approaches have been used to represent very large or very small numbers (both integers and reals) in binary, to be stored in computers.

Example 3.20

Show the number $(10100100000000000000000000000000.00)_2$ in floating-point format.

Solution

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

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

Note that we don't have to worry about all those 0s at the right of the rightmost 1, because they are not significant when we use the real $(1.01001)_2$. The exponent is shown as 32, but it is actually stored in the computer in binary, as we will see shortly. We have also shown the sign as positive, but it would be stored as one bit.

Example 3.21

Show the number $-(0.00000000000000000000000101)_2$ in floating-point format.

Solution

We use the same idea, keeping only one non-zero digit on the left-hand side of the decimal point.

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

Note that exponent is stored as a negative binary in the computer.

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**

Sign, exponent, and mantissa

After a binary number is normalized, only three pieces of information about the number are stored: sign, exponent, and mantissa (the bits to the right of the decimal point). For example, +1000111.0101 becomes:

Sign	*Exponent*		*Mantissa*
+	2^6	×	1.0001110101
+	6		0001110101

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

Sign

The sign of the number can be stored using 1 bit (0 or 1).

Exponent

The exponent (power of 2) defines the shifting of the decimal point. Note that the power can be negative or positive. The **Excess representation** (discussed later) is the method used to store the exponent.

Mantissa

The **mantissa** is the binary integer to the right of the decimal point. It defines the precision of the number. The mantissa is stored in fixed-point notation. If we think of the mantissa and the sign together, we can say this combination is stored as an integer in sign-and-magnitude format. However, we need to remember that it is not an integer—it is a fractional part that is stored like an integer. We emphasize this point because in a mantissa, if we insert extra 0s

to the *right* of the number, the value will not change, whereas in a real integer if we insert extra 0s to the *left* of the number, the value will not change.

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

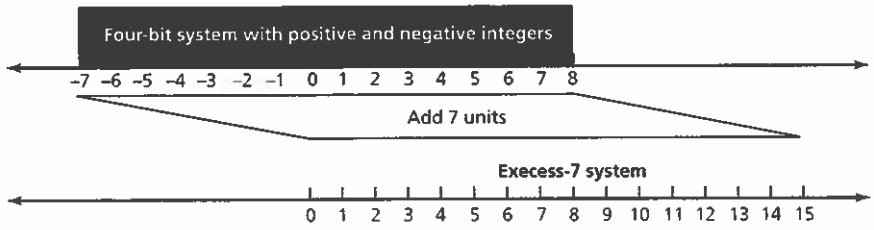
The Excess system

The mantissa can be stored as an unsigned integer. 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.

Example 3.22

We can express sixteen integers in a number system with 4-bit allocation. Using one location for 0 and splitting the other fifteen (not quite equally) we can express integers in the range of −7 to 8, as shown in Figure 3.11. 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*



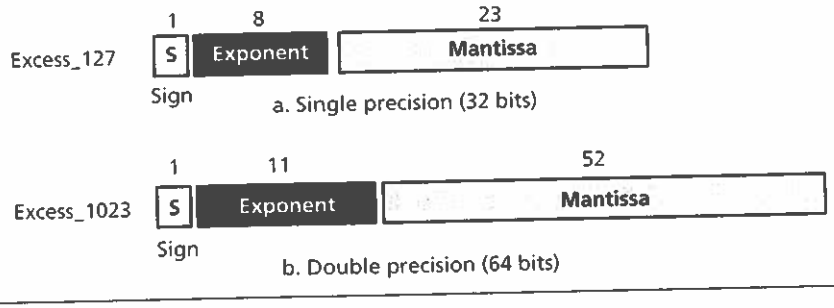
The advantage of this new representation compared to that before the translation is that all integers in the Excess system are positive, so we don't need to be concerned about the sign when we are comparing or doing operations on the integers. For 4-bit allocation, the bias is $2^{4-1} - 1 = 7$, as we expected.

IEEE standards

The Institute of Electrical and Electronics Engineers (IEEE) has defined several standards for storing floating-point numbers. We discuss the two most common ones here, single

precision and double precision. These formats are shown in Figure 3.12. The numbers above the boxes are the number of bits for each field.

Figure 3.12 *IEEE standards for floating-point representation*



Single precision format uses a total of 32 bits to store a real number in floating-point representation. The sign occupies one bit (0 for positive and 1 for negative), the exponent occupies eight bits (using a bias of 127), the mantissa uses 23 bits (unsigned number). This standard is sometimes referred to as **Excess_127** because the bias is 127.

Double precision format uses a total of 64 bits to store a real number in floating-point representation. The sign occupies one bit, the exponent occupies eleven bits (using a bias of 1023), and the mantissa uses 52 bits. The standard is sometimes referred to as **Excess_1023** because the bias is 1023. Table 3.2 summarizes the specification of the two standards.

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

Storage of IEEE standard floating point numbers

A real number can be stored in one of the IEEE standard floating-point formats using the following procedure, with reference to Figure 3.12:

- Store the sign in S (0 or 1).
- Change the number to binary.
- Normalize.
- Find the values of E and M.
- Concatenate S, E, and M.

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: $(101.11)_2 = (1.0111)_2 \times 2^2$.

d. $E = 2 + 127 = 129 = (10000001)_2$, M = 0111. We need to add nineteen zeros at the right of M to make it 23 bits.

e. The presentation is shown below:

S	E	M
0	10000001	01110000000000000000000

The number is stored in the computer as 01000000110110000000000000000000.

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:

S	E	M
1	10000110	01000011110000000000000

The number is stored in the computer as 11000011010000111100000000000000.

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:

S	E	M
1	01111001	10000000000000000000000

The number is stored in the computer as 10111100110000000000000000000000.

Retrieving numbers stored in IEEE standard floating point format

A number stored in one of the IEEE floating-point formats can be retrieved using the following method:

- Find the value of S, E, and M.
- If S = 0, set the sign to positive, otherwise, set the sign to negative.
- Find the shifter (E –127).
- Denormalize the mantissa.
- Change the denormalized number to binary to find the absolute value.
- Add the sign.

Example 3.26

The bit pattern $(11001010000000000111000100001111)_2$ is stored in memory in Excess_127 format. Show what the value of the number is in decimal notation.

Solution

a. The first bit represents S, the next eight bits, Em 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. Denormalization gives us $(1.00000000111000100001111)_2 \times 2^{21}$.

e. The binary number is $(10000000011100010000011.11)_2$.

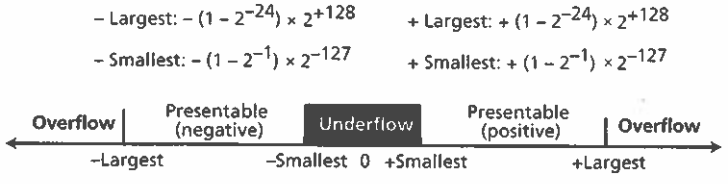
f. The absolute value is 2 104 378.75.

g. The number is –2 104 378.75.

Overflow and underflow

In the case of floating point numbers, we can have both an overflow and **underflow**. Figure 3.13 shows the ranges of floating-point representations using 32-bit memory locations (Excess_127). This representation cannot store numbers with very small or very large absolute values. An attempt to store numbers with very small absolute values results in an underflow condition, while an attempt to store numbers with very large absolute values results to an overflow condition. We leave the calculation of boundary values (+largest, –largest, +smallest, and –smallest) as problems.

Figure 3.13 *Overflow and underflow in floating-point representation of reals*



Storing zero

You may have noticed that 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.

Truncation errors

When a real number is stored using floating-point representation, the value of the numbered stored may not be exactly as we expect it to be. For example, assume we need to store the number:

$$(1111111111111111.1111111111)_2$$

in memory using Excess_127 representation. After normalization, we have:

$$(1.1111111111111111111111111)_2$$

This means that the mantissa has 26 1's. This mantissa needs to be truncated to 23 1's. In other words, what is stored in the computer is:

$$(1.11111111111111111111111)_2$$

which means the original number is changed to:

$$(1111111111111111.11111111)_2$$

with the three 1s at the right of the fractional part truncated. The difference between the original number and what is retrieved is called the **truncation error**. This type of error is very important in areas in which very small or very large number are being used, such as calculations in the space industry. In such cases we need to use larger memory locations and other presentations. The IEEE defines other standards with larger mantissas for these purposes.

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, ten symbols (0, 1, 2, ..., 9) to represent numeric characters (not actual numbers—numbers are treated separately, as we explained in the previous section), 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 3.14 *Representing symbols using bit patterns*

Now the question is: how many bits are needed in a bit pattern to represent a symbol in a language? It depends on how many symbols are in the set used for the language. For example, if we create an imaginary language that uses only English uppercase letters, we need only 26 symbols. A bit pattern in this language needs to represent at least 26 symbols.

For another language, such as Chinese, we may need many more symbols. The length of the bit pattern that represents a symbol in a language depends on the number of symbols used in that language. More symbols mean a longer bit pattern.

Although the length of the bit pattern depends on the number of symbols, the relationship is not linear: it is logarithmic. If we need two symbols, the length is one bit ($\log_2 2$ is 1). If we need four symbols, the length is two bits ($\log_2 4$ is 2). Table 3.3 shows the relationship. A bit pattern of two bits can take four different forms: 00, 01, 10, and 11. Each of these forms can represent a symbol. In the same way, a bit pattern of three bits can take eight different forms: 000, 001, 010, 011, 100, 101, 110, and 111.

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

3.3.1 Codes

Different sets of bit patterns have been designed to represent text symbols. Each set is called a **code**, and the process of representing symbols is called coding. In this section, we explain the common codes.

ASCII

The **American National Standards Institute (ANSI)** developed a code called **American Standard Code for Information Interchange (ASCII)**. This code uses seven bits for each symbol. This means that $2^7 = 128$ different symbols can be defined in this code. The full bit patterns for ASCII code are included in Appendix A. Today ASCII is part of Unicode, which is discussed next.

Unicode

A coalition of hardware and software manufacturers have designed a code called **Unicode** that uses 32 bits and can therefore represent up to $2^{32} = 4\,294\,967\,296$ symbols. Different

sections of the code are allocated to symbols from different languages in the world. Some parts of the code are used for graphical and special symbols. A brief set of Unicode symbols is listed in Appendix A. ASCII is part of Unicode today.

Other codes

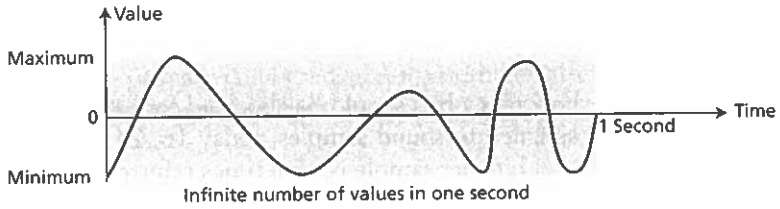
Other codes have been developed during the last few decades. Most of these codes have been made less common with the advent of Unicode. We leave the exploration of these codes as a problem.

3.4 STORING AUDIO

Audio is a representation of sound or music. Audio, by nature, is different than the numbers or text we have discussed so far. Text is composed of countable entities (characters): we can count the number of characters in text. Text is an example of **digital** data. In contrast, audio is not countable. Audio is an entity that changes with time—we can only measure the intensity of the sound at each moment. When we discuss storing audio in computer memory, we mean storing the intensity of an audio signal, such as the signal from a microphone, over a period of time: one second, one hour.

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 infinite number of memory locations. Figure 3.15 shows the nature of an analog signal, such as audio, that varies with time.

Figure 3.15 *An audio signal*



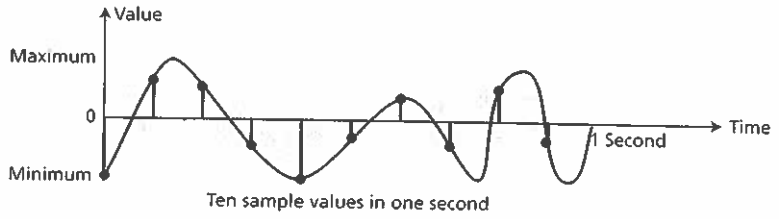
3.4.1 Sampling

If we cannot record all the values of a 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 3.16 shows that selection of ten samples from the signal: we can then record these value to represent the analog signal.

Sampling rate

The next logical question is, how many samples do we need in each second to be able to retrieve a replica of the original signal? The number of samples depends on the maximum number of changes in the analog signal. If the signal is smooth, we need less samples: if the signal is changing rapidly, we need more samples. It has been shown that a **sampling rate** of 40000 samples per second is good enough to reproduce an audio signal.

Figure 3.16 *Sampling an audio signal*



3.4.2 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 number (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.

3.4.3 Encoding

The next task is 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. In other words, some systems use an unsigned integer to represent a sample, while others use signed integers to do so. However, the signed integers don't have to be in two's complement, they can be sign-and-magnitude values. The leftmost bit is used to represent the sign (0 for positive values and 1 for negative values), and the rest of the bits are used to represent the absolute values.

Bit per sample

The system needs to decide how many bits should be allocated for each sample. Although in the past only 8 bits were assigned to sound samples, today 16, 24, or even 32 bits per sample is normal. The number of bits per sample is sometimes referred to as the **bit depth**.

Bit rate

If we call the bit depth or number of bits per sample B, the number of samples per second, S, we need to store $S \times B$ bits for each second of audio. This product is sometimes refer to as **bit rate**, R. For example, if we use 40 000 samples per second and 16 bits per sample, the bit rate is $R = 40\,000 \times 16 = 640\,000$ bits per second = 640 kilobits per second.

3.4.4 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 44 100 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 discard information that cannot be detected by the human ear. This is called lossy compression, as opposed to lossless compression: see Chapter 15.

3.5 STORING IMAGES

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

3.5.1 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, similarly 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** (which stands for **picture elements**). In other words, the whole image is divided into small pixels where each pixel is assumed to have a single intensity value.

Resolution

Just like audio sampling, in image scanning we need to decide how many pixels we need to 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 a 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.

True-Color

One of the techniques used to encode a pixel is called **True-Color**, which uses 24 bits to encode a pixel. In this technique, each of the three primary colors (RGB) are represented by eight bits. Since an 8-bit pattern can represent a number between 0 to 255 in this technique, each color is represented by three decimal numbers between 0 to 255. Table 3.4 shows the three values for some of the colors in this technique.

Table 3.4 Some colors defined in True-Color

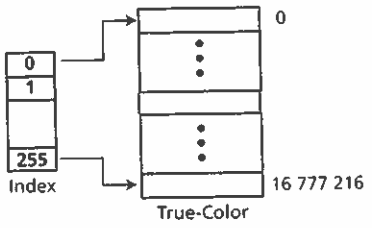
Color	*Red*	*Green*	*Blue*	*Color*	*Red*	*Green*	*Blue*
Black	0	0	0	Yellow	255	255	0
Red	255	0	0	Cyan	0	255	255
Green	0	255	0	Magenta	255	0	255
Blue	0	0	255	White	255	255	255

Note that the True-Color scheme can encode 2^{24} or 16 776 216 colors. In other words, the color intensity of each pixel is one of these values.

Indexed color

The True-Color scheme uses more than 16 million colors. Many applications do not need such a large range of colors. The **indexed color**—or **palette color**—scheme uses only a portion of these colors. In this scheme each application selects a few (normally 256) colors from the large set of colors and indexes them, assigning a number between 0 and 255 to each selected color. This is similar the way in which an artist might have a great many colors in their studio, but at each moment use only a few on their palette. Figure 3.17 illustrates the idea of indexed color.

Figure 3.17 *Relationship of the indexed color scheme to the True-Color scheme*



The use of indexing reduces the number of bits required to store a pixel. For example, in the True-Color scheme 24 bits are needed to store a single pixel. The index color scheme normally uses 256 indexes, which needs only eight bits to store the same pixel. For example, a high-quality digital camera uses almost three million pixels for a 3 × 5 inch photo. The following shows the number of bits that need to be stored using each scheme:

True-Color:	**3000000**	×	**24**	=	**72000000**
Indexed-Color:	**3000000**	×	**8**	=	**24000000**

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.

3.5.2 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. Each geometrical shape is represented by a mathematical formula. For example, a line may be described by the coordinates of its endpoints, and a circle may be described by the coordinates of its center and the length of its radius. A

vector graphic image is made up from a series of commands that defines how these shape should be drawn.

When the image is to be displayed or printed, the size of the image is given to the system as an input. The system rescales the image to the new size and uses the same formulae to draw the image. In this case, each time an image is drawn, the formulae are reevaluated. For this reason, vector graphics are also called geometric modeling or object-oriented graphics.

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.

When the size of the circle is changed, the program changes the value of the radius and recalculates the information to draw the circle again. Rescaling does not change the quality of the drawing.

Vector graphics is not suitable for storing the subtleties of photographic images. JPEG or GIF raster graphics provide much better and more vivid pictures. Vector graphics *is* suitable for applications that use mainly geometric primitives to create images. It is used in applications such as FLASH, and to create TrueType (Microsoft, Apple) and PostScript (Adobe) fonts. Computer-aided design (CAD) also uses vector graphics for engineering drawings.

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 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). 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. Today video is normally compressed. In Chapter 15 we discuss MPEG, a common video compression technique.

3.7 END-CHAPTER MATERIALS

3.7.1 Recommended reading

For more details about the subjects discussed in this chapter, the following books are recommended:

- Halsall, F. *Multimedia Communication*, Boston, MA: Addison Wesley, 2001
- Koren, I. *Computer Arithmetic Algorithms*, Natick, MA: A K Peters, 2001

- Long, B. *Complete Digital Photography*, Hignham, MA: Charles River Media, 200
- Mano, M. *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993
- Miano, J. *Compressed Image File Formats*, Boston, MA: Addison Wesley, 1999

3.7.2 Key terms

American National Standards Institute (ANSI) 62

American Standard Code for Information Interchange (ASCII) 62

analog 63

audio 63

binary digit 42

bit 42

bit depth 64

bit pattern 42

bit rate 64

bitmap graphic 65

byte 43

code 62

color depth 65

digital 63

Excess_1023 58

Excess representation 56

fixed-point representation 44

Excess_127 58

frames 67

floating-point representation 54

Graphic Interchange Format (GIF) 66

indexed color 66

Joint Photographer Expert Group (JPEG) 66

mantissa 56

MP3 64

MPEG 64

normalization 56

one's complement 49

overflow 46

palette color 66

picture element 65

quantization 64

pixel 65

real 53

raster graphic 65

RGB 65

resolution 65

sampling rate 63

sampling 63

sign-and-magnitude representation 46

scanning 65

True-Color 65

text 61

two's complement 50

truncation error 61

unicode 62

two's complement representation 49

vector graphic 66

underflow 60

unsigned integer 44

video 67

3.7.3 Summary

- Data comes in different forms, including numbers, text, audio, image, and video. All data types are transformed into a uniform representation called a bit pattern.
- A number is changed to the binary system before being stored in computer memory. There are several ways to handle the sign. There are two ways to handle the decimal point: fixed-point and floating-point. An integer can be thought of as a number in which the position of the decimal point is fixed: the decimal point is at the right of the least significant bit. An unsigned integer is an integer that can never be negative. One of the methods used to store a signed integer is the sign-and-magnitude format. In this format, the leftmost bit is used to show the sign and the rest of the bits define the magnitude. Sign and magnitude are separated from each other. Almost all computers use the two's complement representation to store a signed integer in an n-bit memory location. In this method, the available range for unsigned integers is divided into two equal subranges. The first half is used to represent non-negative integers, the second half is used to represent negative integers. In two's complement representation, the leftmost bit defines the sign of the integer, but sign and magnitude are not separated from each other. A real is a number with an integral part and a fractional part. Real numbers are stored in the computer using floating-point representation. In floating-point representation a number is made up of three sections: a sign, a shifter, and a fixed-point number.
- A piece of text in any language is a sequence of symbols. We can represent each symbol with a bit pattern. Different sets of bit patterns (codes) have been designed to represent text symbols. A coalition of hardware and software manufacturers have designed a code called Unicode that uses 32 bits to represent a symbol.
- Audio is a representation of sound or music. Audio is analog data. We cannot record an infinite number of values in an interval, we can only record some samples. The number of samples depends on the maximum number of changes in the analog signal. The values measured for each sample is a real number. Quantization refers to a process that rounds up the sample values to integers.
- Storage of images is done using two different techniques: raster graphics and vector graphics. Raster graphics are used when we need to store an analog image such as a photograph. The image is scanned (sampled) and pixels are stored. In the vector graphic method, an image is decomposed into a combination of geometrical shapes such as lines, squares, or circles. Each geometrical shape is represented by a mathematical formula.
- Video is a representation of images (called frames) in time. A movie is a series of frames shown one after another to create the illusion of continuous motion. In other words, video is the representation of information that changes in space (single image) and in time (a series of images).

3.8 PRACTICE SET

3.8.1 Quizzes

A set of interactive quizzes for this chapter can be found on the book website. It is strongly recommended that the student take the quizzes to check his/her understanding of the materials before continuing with the practice set.

3.8.2 Review questions

Q3-1. Name five types of data that a computer can process.

Q3-2. How is bit pattern length related to the number of symbols the bit pattern can represent?

Q3-3. How does the bitmap graphic method represent an image as a bit pattern?

Q3-4. What is the advantage of the vector graphic method over the bitmap graphic method? What is the disadvantage?

Q3-5. What steps are needed to convert audio data to bit patterns?

Q3-6. Compare and contrast the representation of positive integers in unsigned, sign-and-magnitude format, and two's complement format.

Q3-7. Compare and contrast the representation of negative integers in sign-and-magnitude and two's complement format.

Q3-8. Compare and contrast the representation of zero in sign-and-magnitude, two's complement, and Excess formats.

Q3-9. Discuss the role of the leftmost bit in sign-and-magnitude, and two's complement formats.

Q3-10. Answer the following questions about floating-point representations of real numbers:

a. Why is normalization necessary?

b. What is the mantissa?

c. After a number is normalized, what kind of information does a computer store in memory?

3.8.3 Problems

P3-1. How many distinct 5-bit patterns can we have?

P3-2. In some countries vehicle license plates have two decimal digits (0 to 9). How many distinct plates can we have? If the digit 0 is not allowed on the license plate, how many distinct plates can we have?

P3-3. Redo Problem P3-2 for a license plate that has two digits followed by three uppercase letters (A to Z).

P3-4. A machine has eight different cycles. How many bits are needed to represent each cycle?

P3-5. A student's grade in a course can be A, B, C, D, F, W (withdraw), or I (incomplete). How many bits are needed to represent the grade?

P3-6. A company has decided to assign a unique bit pattern to each employee. If the company has 900 employees, what is the minimum number of bits needed to create this system of representation? How many patterns are unassigned? If the

company hires another 300 employees, should it increase the number of bits? Explain your answer.

P3-7. If we use a 4-bit pattern to represent the digits 0 to 9, how many bit patterns are wasted?

P3-8. An audio signal is sampled 8000 times per second. Each sample is represented by 256 different levels. How many bits per second are needed to represent this signal?

P3-9. Change the following decimal numbers to 8-bit unsigned integers.

a. 23 c. 34

b. 121 d. 342

P3-10. Change the following decimal numbers to 16-bit unsigned integers.

a. 41 c. 1234

b. 411 d. 342

P3-11. Change the following decimal numbers to 8-bit two's complement integers.

a. −12 c. 56

b. −145 d. 142

P3-12. Change the following decimal numbers to 16-bit two's complement integers.

a. 102 c. 534

b. −179 d. 62,056

P3-13. Change the following 8-bit unsigned numbers to decimal.

a. 01101011 c. 00000110

b. 10010100 d. 01010000

P3-14. Change the following 8-bit two's complement numbers to decimal.

a. 01110111 c. 01110100

b. 11111100 d. 11001110

P3-15. The following are two's complement binary numbers. Show how to change the sign of the number.

a. 01110111 c. 01110111

b. 11111100 d. 11001110

P3-16. If we apply the two's complement operation to a number twice, we should get the original number. Apply the two's complement operation to each of the following numbers and see if we can get the original number.

a. 01110111 c. 01110100

b. 11111100 d. 11001110

P3-17. Normalize the following binary floating point numbers. Explicitly show the value of the exponent after normalization.

a. 1.10001 c. $2^{-2} \times 101.110011$

b. $2^{3} \times 111.1111$ d. $2^{-5} \times 101101.00000110011000$

P3-18. Convert the following numbers in 32-bit IEEE format.

a. $-2^{0} \times 1.10001$ c. $+2^{-4} \times 1.01110011$

b. $+2^{3} \times 1.111111$ d. $-2^{-5} \times 1.01101000$

P3-19. Convert the following numbers in 64-bit IEEE format.

a. $-2^{0} \times 1.10001$
b. $+2^{3} \times 1.111111$
c. $+2^{-1} \times 1.01110011$
d. $-2^{-5} \times 1.01101000$

P3-20. Convert the following numbers in 32-bit IEEE format.

a. 7.1875
b. −12.640625
c. 11.40625
d. −0.375

P3-21. The following are sign-and-magnitude binary numbers in a 8-bit allocation. Convert them to decimal.

a. 01110111
b. 11111100
c. 01110100
d. 11001110

P3-22. Convert the following decimal integers to sign-and-magnitude with 8-bit allocation.

a. 53
b. −107
c. −5
d. 154

P3-23. One method of representing signed numbers in a computer is one's complement representation. In this representation, to represent a positive number, we store the binary number. To represent a negative number, we apply the one's complement operation on the number. Store the following decimal integers to one's complement with 8-bit allocation.

a. 53
b. −107
c. −5
d. 154

P3-24. The following are one's complement binary numbers in a 8-bit allocation. Convert them to decimal.

a. 01110111
b. 11111100
c. 01110100
d. 11001110

P3-25. If we apply the one's complement operation to a number twice, we should get the original number. Apply the one's complement operation twice to each of the following numbers and see if you can get the original number.

a. 01110111
b. 11111100
c. 01110100
d. 11001110

P3-26. An alternative method to find the two's complement of a number is to first take the one's complement of the number and then add 1 to the result. (Adding binary integers is explained in Chapter 4). Try both method using the following numbers. Compare and contrast the results.

a. 01110111
b. 11111100
c. 01110100
d. 11001110

P3-27. The equivalent of one's complement in the binary system is nine's complement in decimal system (1 = 2 − 1 and 9 = 10 − 1). With n-digit allocation, we can represent nine's complement numbers in the range of: $-[(10^n/2) - 1]$ to $+[(10^n/2 - 1)]$. The nine's complement of a number with

n digit allocation is obtained as follows. If the number is positive, the nine's complement of the number is itself. If the number is negative, we subtract each digit from 9. Answer the following questions for three-digit allocation:

a. What is the range of the numbers we can represent using nine's complement?

b. In this system, how can we determine the sign of a number?

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

P3-28. Assuming three-digit allocation, find the nine's complement of the following decimal numbers.

a. +234 c. −125

b. +560 d. −111

P3-29. The equivalent of two's complement in the binary system is ten's complement in the decimal system (in the binary system, 2 is the base, in the decimal system, 10 is the base). Using n-digit allocation, we can represent numbers in the range of:

$$-(10^n/2) \quad \text{to} \quad +(10^n/2-1)$$

in ten's complement format. The ten's complement of a number with n-digit allocation is obtained by first finding the nine's complement of the number and then adding 1 to the result. Answer the following questions for three-digit allocation.

a. What is the range of the numbers we can represent using ten's complement?

b. In this system, how can we determine the sign of a number?

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

P3-30. Assuming three-digit allocation, find the ten's complement of the following decimal numbers.

a. +234 c. −125

b. +560 d. −111

P3-31. The equivalent of one's complement in the binary system is fifteen's complement in the hexadecimal system ($1 = 2 - 1$ and $15 = 16 - 1$).

a. What range of numbers can we represent with three-digit allocation in fifteen's complement?

b. Explain how the fifteen's complement of a number is obtained in the hexadecimal system.

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

P3-32. Assuming three-digit allocation, find the fifteen's complement of the following hexadecimal numbers.

a. +B14 c. −1A

b. +FE1 d. −1E2

P3-33. The equivalent of two's complement in the binary system is sixteen's complement in the hexadecimal system.

a. What range of numbers can we represent with three-digit allocation in sixteen's complement?

b. Explain how a sixteen's complement of a number is obtained in the hexadecimal system.

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

P3-34. Assuming three-digit allocation, find the sixteen's complement of the following hexadecimal numbers.

a. +B14

b. +FE1

c. −1A

d. −1E2

3.8.4 Applets

We have created some Java applets to simulate some concepts discussed in each chapter. It is strongly recommended that the students active these applets to improve their understanding of the materials discussed in each chapter.