Unit 1 Data Representation in Computers

Structure:

1.1 Introduction

The digital computer is a digital system that performs various computational tasks. The word ***digital*** implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, 9, for example, provide 10 discrete values. The first electronic digital computers, developed in the late 1940s, were used primarily for numerical computations. In this case the discrete elements are the digits. From this application the term ***digital computer*** has emerged. In practice, digital computers function more reliably only if two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e. true-or-false, yes-or-no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be ***binary***.

Objectives:

After studying this unit, the learner will be able to

- Explain various units of a digital computer
- Understand different data types
- Explain fixed and floating point number representation
- Discuss various binary and error detection codes

1.2 Digital Computers

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a ***bit***. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet. By the judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations.

In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1. The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French.

A computer system is sometimes subdivided into two functional entities: hardware and software. The ***hardware*** of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer ***software*** consists of the instructions and the data that the computer manipulates to perform various data-processing tasks. A sequence of instructions for the computer is called a ***program***. The data that are manipulated by the program constitute the ***data base***.

A computer system is composed of its hardware and the system software available for its use. The ***system software*** of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the ***operating system***. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an ***application program***, but the compiler that translates the high-level language program to machine language is a ***system program***. The customer who buys a computer system would need, in addition to the hardware, any available software needed for effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware.

The hardware of the computer is usually divided into three major parts as shown in Fig. 1.1.

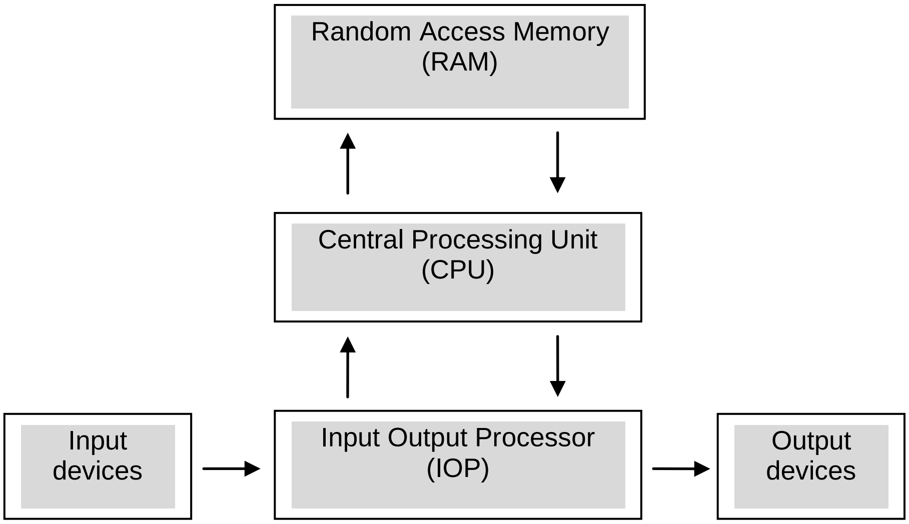


Fig. 1.1: Block diagram of a digital computer

The ***Central Processing Unit (CPU)*** contains an ***Arithmetic and Logic Unit (ALU)*** for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a ***Random Access Memory (RAM)*** because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The ***Input-Output Processor (IOP)*** contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

This book provides the basic knowledge necessary to understand the hardware operations of a computer system. The subject is sometimes considered from three different points of view, depending on the interest of the investigator. When dealing with computer hardware it is customary to distinguish between what is referred to as computer organization, computer design, and computer architecture.

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as ***computer implementation***.

Computer architecture is concerned with the structure and behavior of the computers as seen by the user. It includes the information formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

1.3 Data Types

Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on, to achieve required computational results.

The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form. This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only 1's and 0's. The binary number system is the most natural system to be used in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number Systems

A number system of ***base***, or ***radix, r*** is a system that uses distinct symbols for ***r*** digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of ***r*** and then form the sum of all weighted digits. For example, the ***decimal number system*** in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity

$$7 \text{ x } 10^2 + 2 \text{ x } 10^1 + 4 \text{ x } 10^0 + 5 \text{ x } 10^{-1}$$

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents.

The ***binary number system*** uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \text{ x } 2^5 + 0 \text{ x } 2^4 + 1 \text{ x } 2^3 + 1 \text{ x } 2^2 + 0 \text{ x } 2^1 + 1 \text{ x } 2^0 = 45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$.

Besides the decimal and binary number systems, the ***octal*** (radix 8) and ***hexadecimal*** (radix 16) are important in digital computer work. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.

A number in radix ***r*** can be converted into the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

$$\begin{aligned}(736.4)_8 &= 7 \text{ x } 8^2 + 3 \text{ x } 8^1 + 6 \text{ x } 8^0 + 4 \text{ x } 8^{-1} \\ &= 7 \text{ x } 64 + 3 \text{ x } 8 + 6 \text{ x } 1 + 4 / 8 = (478.5)_{10}\end{aligned}$$

The equivalent decimal number of hexadecimal F3 is obtained from the following calculation:

$$(F3)_{16} = F \text{ x } 16 + 3 = 15 \text{ x } 16 + 3 = (243)_{10}$$

Conversion from decimal to its equivalent representation in the radix ***r*** system is carried out by separating the number into its integer and fraction parts and converting each part separately. The conversion of a decimal integer into a base ***r*** representation is done by successive divisions by ***r*** and accumulation of the remainders. The conversion of a decimal fraction to radix ***r*** representation is accomplished by successive multiplications by ***r*** and accumulation of the integer digits so obtained. Fig.1.2 demonstrates these procedures.

Integer = 41

41	
20	**1**
10	**0**
5	**0**
2	**1**
1	**0**
0	**1**

Fraction = 0.6875

$$\begin{array}{r} 0.6875 \\ \times\ 2 \\ \hline 1.3750 \\ \times\ 2 \\ \hline 0.7500 \\ \times\ 2 \\ \hline 1.5000 \\ \mathrm{X}\ 2 \\ \hline 1.0000 \end{array}$$

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

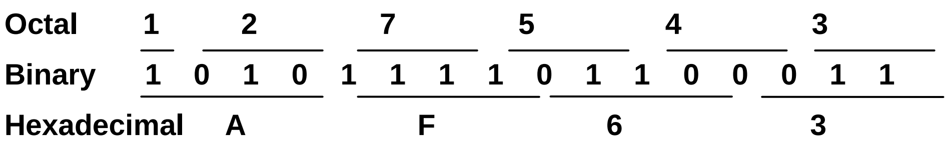
Fig. 1.2: Conversion of decimal 41.6875 into binary

The conversion of decimal 41.6875 into binary is done by first separating the number into its integer part 41 and fraction part 0.6875. The integer part is converted by dividing 41 by ***r*** = 2 to give an integer quotient 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.

The fraction part is converted by multiplying it by ***r*** = 2 to give an integer and a fraction. The new fraction (without the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy. The coefficients of the binary fraction are obtained from the integer digits with the first integer computed being the digit to be placed next to the binary point. Finally, the two parts are combined to give the total required conversion.

Octal and Hexadecimal Numbers

The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers. Since 2^3 = 8 and 2^4 = 16, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each starting from the least significant bit (LSB) position. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register. Physically, one may think of the register as composed of 16 binary storage cells, with each cell capable of holding either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown in Fig.1.3. Since a binary number consists of a string of 1's and 0's, the 16-bit register can be used to store any binary number from 0 to 2^{16} – 1. For the particular example shown, the binary number stored in the register is the equivalent of decimal 44899. Starting from the low-order bit, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself). Each group of three bits is assigned its octal equivalent and placed on the top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.



```
Octal         1    2        7        5        4        3
Binary        1 0 1 0  1  1 1 1  0  1 1  0 0  0  1 1
Hexadecimal      A              F             6             3
```

Fig. 1.3: Binary, octal and hexadecimal conversion.

Octal number	***Binary-coded octal***	***Decimal equivalent***
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7
10	001 000	8
11	001 001	9
12	001 010	10
24	010 100	20
62	110 010	50
143	001 100 011	99
370	011 111 000	248

Code for one octal digit

Table 1.1: Binary-Coded Octal Numbers

Hexadecimal number	***Binary-coded hexadecimal***	***Decimal equivalent***
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15
14	0001 0100	20
32	0011 0010	50
63	0110 0011	99
F8	1111 1000	248

Code for one hexadecimal digit

Table 1.2: Binary-Coded Hexadecimal Numbers

Conversion from binary to hexadecimal is similar except that the bits are divided into groups of four. The corresponding hexadecimal digit for each group of four bits is written as shown below the register of Fig.1.3. The string of hexadecimal digits so obtained represents the hexadecimal equivalent of the binary number. The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in Table 1.1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 1.2.

Table 1.1 lists a few octal numbers and their representation in registers in binary-coded form. The binary code is obtained by the procedure explained above. Each octal digit is assigned a 3-bit code as specified by the entries of the first eight digits in the table. Similarly, Table 1.2 lists a few hexadecimal numbers and their representation in registers in binary-coded form. Here the binary code is obtained by assigning to each hexadecimal digit the 4-bit code listed in the first 16 entries of the table.

Comparing the binary-coded octal and hexadecimal numbers with their binary number equivalent we find that the bit combination in all three representations is exactly the same. For example, decimal 99, when converted to binary, becomes 1100011. The binary-coded octal equivalent of decimal 99 is 143 (001 100 011) and the binary-coded hexadecimal of decimal 99 is 63 (0110 0011). If we neglect the leading zeros in these two binary representations, we find that their bit combination is identical. This should be so because of the straight forward conversion that exists between binary numbers and octal or hexadecimal. The point of all this is that a string of 1s and 0s stored in a register could represent a binary number, but this same string of bits may be interpreted as holding an octal number in binary-coded form (if we divide the bits into groups of three) or as holding a hexadecimal number in binary-coded form (if we divide the bits into groups of four).

The registers in a digital computer contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits. It is more convenient to specify content of registers by their octal or hexadecimal equivalent. The number of digits is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits. It can be expressed in octal as 7777 (four digits) or in hexadecimal as FFF (three digits). Computer manuals invariably choose either the octal or the hexadecimal designation for specifying contents of registers.

Decimal Representation

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand. However, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form. Decimal numbers enter the computer usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are converted into a binary code with four bits per digit.

A binary code is a group of n bits that assume up to 2^n distinct combination of 1s and 0s with each combination representing one element of the set that is being coded. For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations; 00, 01, 10, or 11. A set of eight elements requires a 3-bit code; a set of 16 elements requires a 4-bit code, and so on. A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits from 0 to 9 form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned. Numerous different codes can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table 1.3. This particular code is called ***Binary Coded Decimal*** **(BCD)**. Other decimal codes are sometimes used and a few of them are given in the section 1.7.

It is very important to understand the difference between the conversion of decimal numbers into binary and the binary coding of decimal numbers. For example, when converted to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001. The only difference between a decimal number represented by the familiar digit symbols 0, 1, 2, …, 9 and the BCD symbols 0001, 0010, …, 1001 is in the symbols used to represent the digits – the number itself is exactly the same. A few decimal numbers and their representation in BCD are listed in Table 1.3.

Decimal number	***Binary-coded decimal (BCD) number***
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000
20	0010 0000
50	0101 0000
99	1001 1001
248	0010 0100 1000

Code for one decimal digit

Table 1.3: Binary Coded Decimal (BCD) Numbers

Alphanumeric Representation

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters. An ***alphanumeric character set*** is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, +, and =. Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included). In the first case, the binary code will require six bits and in the second case, seven bits. The standard alphanumeric binary code is the ***American Standard Code for Information Interchange (ASCII),*** which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table 1.4. Note that the decimal digits in ASCII can be converted into BCD by removing the three high-order bits, 011.

Binary codes play an important part in digital computer operations. The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent. The operations specified for digital computers must take into consideration the meaning of the bits stored in registers so that operations are performed on operands of the same type. In inspecting the bits of a computer register at random, one is likely to find that it represents some type of coded information rather than a binary number.

Binary codes can be formulated for any set of discrete elements such as the musical notes and chess pieces and their positions on the chessboard. Binary codes are also used to formulate instructions that specify control information for the computer.

Character	***Binary code***	***Character***	***Binary code***
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	Space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 0100
S	101 0011	)	010 1001
T	101 0100	-	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Table 1.4: American Standard Code for Information Interchange (ASCII)

1.4 Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base ***r*** system: the ***r's*** complement and the ***(r-1)'s*** complement. When the value of the base ***r*** is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

(r-1)'s Complement

Given a number ***N*** in base ***r*** having ***n*** digits, the ***(r-1)'s*** complement of ***N*** is defined as $(r^n - 1) - N$. For decimal numbers $r = 10$ and $r - 1 = 9$, so the ***9's*** complement of ***N*** is $(10^n - 1) - N$. Now, 10^n represents a number that consists of a single ***1*** followed by ***n 0s***. $10^n - 1$ is a number represented by ***n 9s***. For example, with $n = 4$ we have $10^4 = 10000$ and $10^4 - 1 = 9999$. It follows that the ***9's*** complement of a decimal number is obtained by subtracting each digit from ***9***. For example, the ***9's*** complement of ***546700*** is $999999 - 546700 = 453299$ and the ***9's*** complement of ***12389*** is $99999 - 12389 = 87610.$

For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of ***N*** is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a ***1*** followed by ***n 0s***. $2^n - 1$ is a binary number represented by ***n 1s***. For example, with $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the ***1's*** complement of a binary number is obtained by subtracting each digit from ***1***. However, the subtraction of a binary digit from ***1*** causes the bit to change from ***0 to 1*** or from ***1 to 0***. Therefore, the ***1's*** complement of a binary number is formed by changing ***1s into 0s*** and ***0s into 1s***. For example, the ***1's*** complement of ***1011001*** is ***0100110*** and the ***1's*** complement of ***0001111*** is ***1110000.***

The ***(r – 1)'s*** complement of octal or hexadecimal numbers are obtained by subtracting each digit from ***7 or F*** (decimal 15) respectively.

(r’s) Complement

The ***r’s*** complement of an ***n*** digit number ***N*** in base ***r*** is defined as $r^n - N$ for $N \neq 0$ and ***0*** for $N = 0$. Comparing with the ***(r – 1)’s*** complement, we note that the ***r’s*** complement is obtained by adding ***1*** to the ***(r – 1)’s*** complement since $r^n - N = [(r^n - 1) - N] + 1$. Thus the ***10’s*** complement of the decimal ***2389*** is ***7610 + 1 = 7611*** and is obtained by adding ***1*** to the ***9’s*** complement value. The ***2’s*** complement of binary ***101100*** is ***010011 + 1 = 010100*** and is obtained by adding ***1*** to the ***1’s*** complement value.

Since 10^n is a number represented by a ***1*** followed by ***n 0s***, then $10^n - N$, which is the ***10’s*** complement of ***N***, can be formed also by leaving all least significant ***0s*** unchanged, subtracting the first non-zero least significant digit from ***10***, and then subtracting all higher significant digits from ***9***. The ***10’s*** complement of ***246700*** is ***753300*** and is obtained by ***leaving the two zeros unchanged***, subtracting ***7 from 10***, and subtracting the ***other three digits from 9***. Similarly, the ***2’s*** complement can be formed by leaving all least significant ***0’s*** and the first ***1*** unchanged, and then replacing ***1s by 0s*** and ***0s by 1s*** in all other higher significant bits. The ***2’s*** complement of ***1101100*** is ***0010100*** and is obtained by ***leaving the two low-order 0s and the first 1 unchanged,*** and then replacing ***1s by 0s*** and ***0s by 1s*** in the other four most significant bits.

In the definitions above it was assumed that the numbers do not have a radix point. If the original number ***N*** contains a radix point, it should be removed temporarily to form the ***r’s*** or ***(r-1)’s*** complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The ***r’s*** complement of ***N*** is $r^n - N$. The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

The subtraction of two ***n*** digit unsigned numbers ***M – N (N ≠ 0)*** in base ***r*** can be done as follows:

1. ***Add the minuend M to the r's complement of the subtrahend N. This performs M + (r^n – N) = M – N + r^n.***
2. ***If M ≥ N, the sum will produce an end carry r^n which is discarded, and what is left is the result M – N.***
3. ***If M < N, the sum does not produce an end carry and is equal to r^n – (N – M), which is the r's complement of (N – M). To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.***

Consider, for example, the subtraction 72532 – 13250 = 59282. The 10's complement of 13250 is 86750. Therefore:

M	=	72532
10's complement of N	=	+86750
Sum	=	159282
Discard end carry 10^5	=	-100000
Answer	=	59282

Now consider an example with M < N. The subtraction 13250 – 72532 produces negative 59282. Using the procedure with complements, we have

M	=	13250
10’s complement of N	=	+27468
Sum	=	40718

There is no end carry. Answer is negative 59282 = 10’s complement of 40718.

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction X – Y and Y – X using 2’s complements:

X	=	1010100
2’s complement of Y	=	+0111101
Sum	=	10010001
Discard end carry 2^7	=	-10000000
Answer: X – Y	=	0010001

Y	=	1000011
2’s complement of X	=	+0101100
Sum	=	1101111

There is no end carry. Answer is negative 0010001 = 2’s complement of 1101111.

1.5 Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1s and 0s, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

In addition to the sign, a number may have a ***binary (or decimal) point***. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. ***Signed-magnitude representation***
2. ***Signed-1’s complement representation***
3. ***Signed-2’s complement representation***

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1’s or 2’s complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110. Note that each of the eight bits of the register must have a value and therefore 0’s must be inserted in the most significant positions following the sign bit. ***Although there is only one way to represent +14, there are three different ways to represent -14 with eight bits.***

In signed-magnitude representation	***1***	***0001110***
In signed-1’s complement representation	***1***	***1110001***
In signed-2’s complement representation	***1***	***1110010***

The signed-magnitude representation of -14 is obtained from +14 by complementing only the sign bit. The signed-1’s complement representation of -14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2’s complement representation is obtained by taking the 2’s complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The 1’s complement imposes difficulties because it has two representations of 0 (+0 and -0). It is seldom used for arithmetic operations except in some older computers. The 1’s complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, (+25) + (-37) = -(37 – 25) = -12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction.

By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: ***Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position.*** Numerical examples for addition are shown below. ***Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.***

+6	00000110	-6	11111010
+13	00001101	+13	00001101
-----	-------------	-----	-------------
+19	00010011	+7	00000111
+6	00000110	-6	11111010
-13	11110011	-13	11110011
----	-------------	-----	-------------
-7	11111001	-19	11101101

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows: ***Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.***

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

(±A) – (+B) = (±A) + (-B)

(±A) – (-B) = (±A) + (+B)

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of (-6) – (-13) = +7. In binary with eight bits this is written as 11111010 – 11110011. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give (+13). In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+7).

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow has occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains n+1 bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of singed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

```
carries: 0  1                     carries: 1  0
+70        0  1000110            -70        1  0111010
+80        0  1010000            -80         1  0110000
-------   ----------------       -------    ----------------
+150       1  0010110            -150        0  1101010
```

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

Decimal Fixed-Point Representation

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. Also, the circuits required to perform decimal arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion into binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform to the 4-bit code of the decimal digits. It is customary to designate a plus with four 0’s and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9’s or the 10’s complement, but the 10’s complement is the one most often used. To obtain the 10’s complement of a BCD number, we first take the 9’s complement and then add one to the least significant digit. The 9’s complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2’s complement system apply also to the signed-10’s complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10’s complement form. Consider the addition (+375) + (-240) = +135 done in the signed-10’s complement system.

0 375 $(0000\ 0011\ 0111\ 0101)_{BCD}$
+9 760 $(1001\ 0111\ 0110\ 0000)_{BCD}$
-------- ----------------------------------
0 135 $(0000\ 0001\ 0011\ 0101)_{BCD}$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10’s complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders.

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

1.6 Floating-Point Representation

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the ***mantissa***. The second part designates the position of the decimal (or binary) point and is called the ***exponent***. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	***Exponent***
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation +0.6132789 x 10^{+4}.

Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	***Exponent***
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^{e} = +(.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be ***normalized*** if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0s. The number can be normalized by shifting it three positions to the left and discarding the leading 0s to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0s in the mantissa and exponent.

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers.

1.7 Other Binary Codes

In previous sections we introduced the most common types of binary-coded data found in digital computers. Other binary codes for decimal numbers and alphanumeric characters are sometimes used. Digital computers also employ other binary codes for special applications. A few additional binary codes encountered in digital computers are presented in this section.

Gray Code

Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter. The reflected binary or ***Gray code***, shown in Table 1.5, is sometimes used for the converted digital data. The advantage of the Gray code over straight binary numbers is that the Gray code changes by only one bit as it sequences from one number to the next. In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 or from 1 to 0. A typical application of the Gray code occurs when the analog data are represented by the continuous change of a shaft position. The shaft is partitioned into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Binary code	***Decimal equivalent***	***Binary code***	***Decimal equivalent***
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

Table 1.5: 4-Bit Gray Code

Gray codes counters are sometimes used to provide the timing sequence that control the operations in a digital system. A Gray code counter is a counter whose flip-flops go through a sequence of states as specified in Table 1.5. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

Other Decimal Codes

Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be formulated by arranging four or more bits in 10 distinct possible combinations. A few possibilities are shown in Table 1.6.

Decimal digit	***BCD 8421***	***2421***	***Excess-3***	***Excess-3 gray***
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111

8	1000	1110	1011	1110
9	1001	1111	1100	1010
Unused bit combinations	1010	0101	0000	0000
	1011	0110	0001	0001
	1100	0111	0010	0011
	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

Table 1.6: Four Different Binary Codes for the Decimal Digit

The BCD (binary-coded decimal) has been introduced before. It uses a straight assignment of the binary equivalent of the digit. The six unused bit combinations listed have no meaning when BCD is used, just as the letter H has no meaning when decimal digit symbols are written down. For example, saying that 1001 110 is a decimal number in BCD is like saying that 9H is a decimal number in the conventional symbol designation. Both cases contain an invalid symbol and therefore designate a meaningless number.

One disadvantage of using BCD is the difficulty encountered when the 9's complement of the number is to be computed. On the other hand, the 9's complement is easily obtained with the 2421 and the excess-3 codes listed in Table 1.6. These two codes have a ***self-complementing*** property which means that the 9's complement of a decimal number, when represented in one of these codes, is easily obtained by changing 1's to 0's and 0's to 1's. The property is useful when arithmetic operations are done in signed-complement representation.

The 2421 is an example of a ***weighted code***. In a weighted code, the bits are multiplied by the weights indicated and the sum of the weighted bits gives the decimal digit. For example, the bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of 2 x 1 + 4 x 1 + 2 x 0 + 1 + 1 = 7. The BCD code can be assigned the weights 8421 and for this reason it is sometimes called the 8421 code.

The ***excess-3 code*** is a decimal code that has been used in older computers. This is an unweighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3 (0011).

From Table 1.5 we note that the Gray code is not suited for a decimal code if we were to choose the first 10 entries in the table. This is because the transition from 9 back to 0 involves a change of three bits (from 1101 to 0000). To overcome this difficulty, we choose the 10 numbers starting from the third entry 0010 up to the twelfth entry 1010. Now the transition from 1010 to 0010 involves a change of only one bit. Since the code has been shifted up three numbers, it is called the excess-3 Gray. This code is listed with the other decimal codes in Table 1.6.

Other Alphanumeric Codes

The ASCII code (Table 1.4) is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity. The code consists of 128 characters. Ninety-five characters represent *graphic symbols* that include upper- and lowercase letters, numerals zero to nine, punctuation marks, and special symbols. Twenty-three characters represent format effectors, which are functional characters for controlling the layout of printing or display devices such as carriage return, line feed, horizontal tabulation, and back space. The other 10 characters are used to direct the data communication flow and report its status.

Another alphanumeric (sometimes called alphameric) code used in IBM equipment is the ***EBCDIC (Extended BCD Interchange Code)***. It uses eight bits for each character (and a ninth bit for parity). EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.

When alphanumeric characters are used internally in a computer for data processing (not for transmission purpose) it is more convenient to use a 6-bit code to represent 64-characters. A 6-bit code can specify the 26 uppercase letters of the alphabet, numerals zero to nine, and up to 28 special characters. This set of characters is usually sufficient for data-processing purposes. Using fewer bits to code characters has the advantage of reducing the memory space needed to store large quantities of alphanumeric data.

1.8 Summary

This unit explains clearly various units of a digital computer and different data formats used in computers. Also we have learnt the number representation in digital computers and different types of codes used in computers and communication.

Self Assessment Questions

1. The word digital implies that the information in the computer is represented by variables that take a limited number of ____________ values.
2. The _________ of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.
3. The ______________ of a computer consists of a collection of programs whose purpose is to make more effective use of the computer.

4. The ____________ contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world.
5. ASCII stands for _________________________.

1.9 Terminal Questions

1. Convert the following binary numbers to decimal: 101110; 1110101; and 110110100.
2. Convert the following decimal numbers to binary: 1231; 673; and 1998.
3. Convert the hexadecimal number F3A7C2 to binary and octal.
4. Explain different types of binary codes.

1.10 Answers

Self Assessment Questions:

1. discrete
2. hardware
3. system software
4. input-output processor (IOP)
5. American Standard Code for Information Interchange

Terminal Questions:

1. Refer Section 1.3
2. Refer Section 1.3
3. Refer Section 1.3
4. Refer Section 1.7