

## Chapter 4: Data representation in Computers

Objectives:

- To understand how really data represented, flow, stored and processed inside the computer.
- To know the different coding systems & unit of data representation.
- To know different number systems
- To know how to convert from one base to another
- To know how to apply arithmetic operators on binary number system

We enter data into a computer or review (see) output data from a computer using the letter of alphabet, various special symbols, and the numerals in the decimal number system. But since computer is an electronic device which understands electrical flow (signal) there is no letter, symbol or number inside the computer. Computer works with binary numbers. As a semiconductor is conducting or isn't conducting; a switch is closed or opened. It is not surprising, therefore, that almost all of the present day computers are based on binary system of representation that uses only two symbols – 1 and 0.

Any information stored in the computer system is first converted into binary number and then stored in the computer memory or hard disk in the form of 1s and 0s.

For example, to convert 7 into binary number, we divide it by 2.

7/2

|   | Number | Remainder |
|---|--------|-----------|
| 2 | 7      |           |
| 2 | 3      | 1         |
|   | 1      | 1         |

It gives a result of (111)<sub>2</sub>

### UNITS OF DATA REPRESENTATION

When data is stored, processed or communicated within the computer system, it is packed in units;

Arranged from the smallest to the largest, the units are called bit, byte and word;  
These units are based on the binary number system;

## BIT

Bits are the smallest units and can convey only two possible states 0 or 1; bit stands for binary digits;

A bit is a single element in the computer, on a disk that stands for either “ON” indicating 1 or “OFF” indicating 0;

In the computer “ON” is represented by the existence of current and “OFF” is represented by the non existence of current

On a magnetic disk, the same information is stored by changing the polarity or magnetized particles on the disk’s surface;

## BYTE

Bits can be organized into large units to make them represent more and meaningful information;

This large unit is called a byte and is the basic “unit of data representation” in a computer system;

The commonly used byte contains 8 bits;

Since each bit has two states and there are 8 bits in a byte, the total amount of data that can be represented is  $2^8$  or 256 possible combinations;

Each byte can represent a character(a character is either a letter, a number or a special symbol such as +,-,?,\*, \$, etc

A byte is then used as a unit of measurement in the computer memory, processing unit, external storage and during communication;

If the computer memory is 524288 byte, this is expressed in short by saying 512KB, where KB stands for kilobyte.

1 Kilobyte(1KB) is  $2^{10}$  or 1024 bytes

1 Megabyte(MB) is  $2^{20}$  bytes or  $2^{10}$  kilobytes

1 Gigabyte(GB) is  $2^{30}$  bytes or  $2^{20}$  kilobytes or  $2^{10}$  megabytes

## WORD

Word refers the number of bits that a computer process at a time or a transmission media transmits at a time

Although bytes can store or transmit information, the process can even be faster if more than one byte is processed at a once;

A combination of bytes, then form a “word”

A word can contain one, two, three or four bytes based on the capacity of the computer;

Word length is usually given in bits

We say that a computer is an 8-bits, a 16 bit, a 32 bit or a 64 bit computer to indicate that the amount of data it can process at a time;

The large the word length a computer has the more powerful and faster it is.

## **CODING METHODS**

It is possible to represent any of the character in our language in a way as a series of electrical switches in arranged manner;

These switch arrangements can therefore be coded as a series of an equivalent arrangements of bits

There are different coding systems that convert one or more character sets into computer codes. Some are: BCD, ASCII, EBCDIC and Unicode is three of the most popular text coding systems invented

### **BCD – Binary Coded Decimal**

The BCD used a group of four bits to represent information in the computer system. It had a maximum of 16 different alternative characters or numbers. It could represent only numbers and some special symbols.

With BCD, each digit of a number is converted into its binary equivalent rather than converting the entire decimal number to its binary form.

Example: The BCD value of the decimal number 5319 is 0101 0011 0001 1001

### **BCD (6-bits)**

It uses 6-bits to code a Character

Coding Examples

| Character | BCD  |        |
|-----------|------|--------|
|           | Zone | digits |
| A         | 11   | 0001   |
| Q         | 10   | 1000   |
| 8         | 00   | 1000   |
| 9         | 00   | 1001   |

## ASCII-7

ANSI's solution to representing symbols with bits of data was the ASCII character set. ASCII (pronounced AS-key) stands for the American Standard Code for Information Interchange.

Today, the ASCII character set is by far the most commonly used in computers of all types.

Used widely before the introduction of ASCII-8 (the Extended ASCII)

Uses 7 bits to represent a character;

With the seven bits,  $2^7$  ( or 128) different characters can be coded (0000000-1111111)

It has a zone and digit bits positions

The characters from 0 to 31 and 127 are control characters;

from 32 to 64, special characters and numbers;

from 65 (A)to 96, uppercase letters and a few symbols;

From 97(a) to 126, lowercase letters plus a handful of common symbols.

### **Coding examples:**

| ASCII-7 |           |      |       |
|---------|-----------|------|-------|
| Value   | Character | Zone | digit |
| 36      | \$        | 010  | 0100  |
| 37      | %         | 010  | 0101  |
| 65      | A         | 100  | 0001  |
| 66      | B         | 100  | 0010  |
| .       |           |      |       |
| .       |           |      |       |
| .       |           |      |       |
| 90      | Z         |      |       |
| 97      | a         | 110  | 0001  |
| 98      | b         | 110  | 0010  |
| .       |           |      |       |
| .       |           |      |       |
| .       |           |      |       |
| 122     | z         |      |       |

### **The ASCII System**

Also referred as ASCII-8 or Extended ASCII

It is the most widely used type of coding scheme for Micro Computer system

ASCII uses 8-bits to represent alphanumeric characters(letters, digits and special symbols).

With the 8-bits, ASCII can represent  $2^8$  or 256 different characters(00000000-11111111).

### **Coding Examples**

| Character | Binary representation in ASCII |
|-----------|--------------------------------|
| a         | 01100001                       |
| b         | 01100010                       |
| A         | 01000001                       |
| B         | 01000010                       |
| ?         | 00111111                       |
| +         | 00101011                       |
| 1         | 00110001                       |
|           | 00110010                       |
|           | 00110011                       |

**EBCDIC** pronounced as “Eb-see-dick” and stands for Extended Binary Coded Decimal Interchange Code.

It is an 8-bit coding scheme; (00000000 – 11111111)

It accommodates to code  $2^8$  or 256 different characters

It is a standard coding scheme for the large computers.

### **Coding Examples**

| EBCDIC    |      |       |
|-----------|------|-------|
| Character | Zone | Digit |
| a         | 1000 | 0001  |
| b         | 1000 | 0010  |
| A         | 1100 | 0001  |
| B         | 1100 | 0010  |
| 0         | 1111 | 0000  |
| 9         | 1111 | 1001  |

### **Unicode**

An evolving standard for data representation, called the Unicode Worldwide Character Standard, provides 2 bytes (=16 bits) to represent each letter, number, or symbol.

With 2 bytes, enough Unicode codes can be created to represent more than 65,536 different characters or symbols.

This total is enough for every unique character and symbol in the world, including the vast Chinese, Korean, and Japanese character sets and those found in known classical and historical texts.

One major advantage that Unicode has over other text code systems is its compatibility with ASCII codes.

The first 256 codes in Unicode are identical to the 256 codes used by Extended ASCII systems.

Unicode then extends far beyond the standard ASCII character set.

The Unicode standard was developed in 1991 by a joint engineering team from Apple Computer Corporation and Xerox Corporation. It has been updated continually since its inception.

In fall 1999, version three of the worldwide Unicode standard was released by the Unicode Consortium, a group of developers, corporations, researchers, and other groups working in conjunction with the ISO. In version three, the Unicode standard includes a total of 57,709 16-bit code values.

Many software publishers, including IBM, Microsoft, Netscape, and others, encourage their developers to use Unicode in their programs.

Unicode is supported by some operating systems (such as Windows 2000 and OS/2) and applications. Even so, it has not yet been universally adopted by software developers.

Still, the goal is a worthwhile one. If a single character set were available to cover all languages, computer programs and data would be interchangeable.

### **Number systems**

There are different number systems. Some of are:

1. Decimal number systems
2. Binary number systems
3. Octal number systems
4. Hexadecimal number systems

### **Decimal number systems (base 10)**

It is the common and widely used number system

It uses 10 symbols to represent or count things

Probably it was invented because we have 10 fingers with which we count and add.

The Decimal number system is based on the ten different digits (or symbols)

0,1,2,3,4,5,6,7,8,9.

For numbers greater than 9 it uses the symbols 0-9 to represent the number example 10 which is a combination of 1 and 0 and so on.

We say it is a base ten number system

Though it is widely used, it is inconvenient for computer to represent data. So we need another number system.

### **Binary number system (base 2)**

Binary number system is based on the two different digits; 0 and 1;

It is important to note that every decimal number system has its equivalent binary number;

Conversion from binary to its equivalent decimal and from decimal to its equivalent binary is possible;

Operation on binary number system is also possible

The most elementary form to organize data within a computer (an electronic device) is in the form of a code which utilizes the “ON” and “OFF” states of electric switches or there is “current” and “no current” condition of the electronic components;

We see that the nature of the electronic devices has similarity with the binary number system in that both represent only two elementary states;

It is therefore convenient to use binary number system to represent data in a computer;

An “ON” corresponds to a 1;

An “OFF” corresponds to a 0;

In the computer “ON” is represented by the existence of a current and “OFF” is represented by non existence of current

On a magnetic disk, the same information is stored by changing the polarity of magnetized particles on the disk’s surface

### **Octal number System (base 8) (Oct)**

Again what would happened if we had only 8 fingers off course we would have used the 8 symbols (0-7) and for numbers greater than we would represented them in terms of the first 8 symbols example for 8,10 and so on.

It uses 8 symbols 0-7 to represent numbers

When we compare the octal with the decimal 0-7 in octal is the same as 0-7 in decimal but 10 in octal is not the same as 10 in decimal because 10 in octal holds the position of 8 in decimal, off course 10 in octal is the same as 8 in decimal.

## **Hexadecimal number system (16) (hex)**

Suppose we had 16 fingers instead of 10 in case of 10 fingers they invented new symbols until they exhausted all their fingers. Now they would have to invent new symbols because there would have been some more fingers left unrepresented. These extra symbols are usually represented by the letters at the beginning of the English alphabet.

It uses 16 symbols to represent numbers. But for the numbers greater than 15 they represented in terms of the 16 symbols example 16 as 10 and so on.

When we compare the hexadecimal with decimal 0-9 in hexadecimal is the same as 0-9 in decimal but 10 in hexadecimal is not the same as 10 in decimal, of course 10 in hexadecimal is equal to 16 in decimal because it holds the position of 16 in decimal.

## **Conversion from one Base to Another**

The values that members have within given number systems are largely determined by their positional notation. Positional notation means that the position of one symbol relative to other symbols in a given number system determines the value of that symbol. For example, these symbols 1 & 7 can represent either 17 or 71 depending upon their relative position to one another.

The decimal number 135 may be expanded as:

$$(135)_{10} = 1*10^2 + 3*10^1 + 5*10^0$$

The subscript 10 is used to indicate that 135 is a base 10 number.

The number 10 in binary is not the same as 10 in decimal because the value of 1 in the binary is not the same as the value of the 1 in the decimal.

The binary number 1101 may be expanded as

$$\begin{aligned} 1101_2 &= 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \\ &= (1*8) + (1*4) + (0*2) + (1*1) \\ &= 8 + 4 + 0 + 1 \\ &= 13_{10} \end{aligned}$$

A number  $X_1 X_2 X_3 \dots X_n$  in base M can be expanded as

$$(X_0 X_1 X_2 X_3 \dots X_n)_M = X_1 * M^{n-1} + X_2 * M^{n-2} + X_3 * M^{n-3} + \dots + X_1 * M^{n-1} + X_n M^0$$

### Conversion from decimal (base 10) to other base (base M)

(Base 10 —————> Base M)

To convert a decimal number X to a number in base m, divide X by m, find the remainder, again divide the result by M, find the remainder, continue until the result is 0 . And concatenate (collect) the remainders starting from the last up to the first.

Ex.1 Convert  $56_{10}$  to base (binary)  $X=56$   $M=2$

$$56_{10} = 111000_2$$

EX.2 Convert  $78_{10}$  to base 8 (Octal)

$$78_{10} = 116_8$$

EX. 3. Convert  $30_{10}$  to base 16 (hexadecimal)

$$30_{10} = 1E_{16}$$

EX. 4. Convert  $16_{10}$  to base 16 (hexadecimal)

$$16_{10} = A0_{16}$$

### Conversion from base M to base 10 (decimal)

To convert a number X consists of digits  $X_1 X_2 X_3 \dots X_n$  in base m to decimal, simply expand the number with base M. That is

$$(X_1 X_2 X_3 \dots X_n)_m = X_1 * m^{n-1} + X_2 * m^{n-2} + X_3 * m^{n-3} + \dots + X_i * m^{n-i} + \dots X_{n-1} m^1 + X_n * m^0 = Y_{10}$$

Examples:

1. Convert  $(1001001)_2$  to decimal =73

2. convert  $(234)_8$  to decimal =156

3. convert  $(101)_8$  to decimal =65

4. convert  $(A1B)_{16}$  to decimal = 2587

5. covert  $(101)_{16}$  to decimal =257

## Conversion from binary (base2) to Octal (base 8) or

### hexadecimal (base16) and viceversal

To convert a number in binary to octal ,group three binary digits together starting from the last digit (right and if there are no enough digits add zeros in front , (left) and find the corresponding Octal of each group.

Example. Convert 1001001 to octal

$$1001001=001,001,001$$

$$= 111_8$$

convert 101101001 to octal

$$101101001=101,101,001$$

$$=551_8$$

To convert binary to hexadecimal group four binary digits together starting from right and if there are no enough digits add zeros at the left.

EX. Convert 111100100 to hexadecimal

$$111100100=0001,1110,0100$$

$$= 1 \quad 14 \quad 4$$

$$= 1E4_{16}$$

To convert from Octal to binary , convert each octal digit to its equivalent 3 bit binary starting from right.

EX. Convert  $(675)_8$  to binary

$$675_8=110 \ 111 \ 101$$

$$=110111101_2$$

Convert  $231_8$  to binary

$$231_8=010 \ 011 \ 001$$

$$=10011001_2$$

To convert from Hexadecimal to binary convert each hex. Digit to its equivalent 4-bit binary, starting from right.

Ex. Convert  $234_{16}$  to binary

$$234_{16} = 0010 \ 0011 \ 0100$$

$$= 1000110100_2$$

Convert  $2AC$  to binary

$$2AC_{16} = 0010 \ 1010 \ 1100$$

$$= 1010101100_2$$

### **Conversion from Octal to hexadecimal and Viceversa**

To convert from Octal to hexadecimal, first we have to convert to binary and the binary to hexadecimal. To convert from hexadecimal to Octal, first we have to convert to binary and then the binary to Octal.

EX.1. Convert  $235_8$  to hexadecimal

$$238_8 = 010 \ 011 \ 101$$

$$= 0000 \ 1001 \ 1101$$

$$= 0 \ 9 \ 13$$

$$= 9D_{16}$$

Convert  $1A$  to Octal

$$1A = 0001 \ 1010$$

$$= 000 \ 011 \ 010$$

$$= 0 \ 3 \ 2$$

$$= 32_8$$

### **Converting decimal number to BCD equivalence**

To convert a decimal number to its equivalent BCD simply convert each decimal digit to its 40bit binary and combine the groups together

EX. Convert  $432$  to BCD

$$432 = 0100 \ 0011 \ 0010$$

$$=10000110010_{\text{BCD}}$$

To convert BCD to decimal group into four & find its corresponding decimal.

**Converting decimal number which contains fractions to Binary and Binary with fraction to decimal.**

First change the integer part to its equivalent binary. Multiply the fractional part by 2 and take out the integer value, and again multiply the fractional part of the result by 2 and take out the integer part, continue this until the product is 0. Collect the integer values from top to bottom & concatenate with the integer part.

Ex.

B. Convert  $12.25_{10}$  to binary

A) Convert  $3.1875$  to binary

– Convert 3 to binary  $\gg (11)_2$

– Convert  $0.1875$  to binary

Hence  $3.1875 = (11.0011)_2$

B) Convert  $12.25$  to binary  $(1100.01)_2$

C) Convert  $25.34$  to binary.

To convert a binary number  $Y_1Y_2Y_3Y_4Y_n.d_1d_2d_3\dots d_m$  to decimal, first convert the integer part to decimal by using  $y_1y_2y_3y_4\dots y_n = y_1*2^{n-1} + y_2*2^{n-2} + \dots + y_j*2^{n-j} + \dots + y_{n-1}*2^1 + y_n*2^0 = Q$  and convert the fractional part to decimal by using  $d_1d_2d_3\dots d_m = d_1*2^{-1} + d_2*2^{-2} + d_3*2^{-3} + \dots + d_j*2^{-j} + \dots + d_m*2^{-m} = R$  then decimal equivalence of  $y_1y_2y_3y_4\dots y_n.d_1d_2\dots d_m$  will be  $Q+R$  where  $Q$  is the integer part and  $R$  is the fractional part.

Ex1 : Convert  $11001.0101$  to decimal

$$11001 = 1x2^4 + 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 16 + 8 + 1 = 25 = Q$$

$$0101 = 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4} = 0 + \frac{1}{4} + 0 + \frac{1}{16} = 0.3125 = R$$

$$11001.0101 = 25.3125.$$

Ex 2: Convert  $1000.1$  to decimal

$$1000 = 1 + 2_3 + 0 + 0 + 0 = 8$$

$$1 = 1x2^{-1} = \frac{1}{2} = 0.5$$

$$1000 \frac{1}{2} = 8.5_{10}$$

### **Conversion from Binary with fraction to Octal or hexadecimal**

Group three/ four digits together starting from the last of the integer part, and if there are not enough digits add some zeros in the front.

Group three/ four digits together starting from the first of the fractional part, and if there are not enough digits add some zeros to the last.

Convert each group of the integer and the fractional part to their equivalent Octal/hexa and collect the results by adding point (.) to separate the integer part from the fractional part.

Ex 1:- Convert  $1010.0111_2$  to octal

Ex2:- Convert  $1110101.10111_2$  to hexa

### **Conversion from Octal or hexadecimal with fraction to binary.**

Convert each Octal/hexa digit to its equivalent 3-bit/ 4-bit binary.

Collect the binary sequences by separating the integer part binaries from the fractional part binaries with point (.)

Conversion from Octal with fraction to hexa and hexa with fraction to octal and octal/hexa with fraction to decimal.

To convert from Octal to hexa, first convert the Octal to binary and then the binary to hexa

To convert from hexa to Octal, first convert the hexa to binary and then the binary to Octal.

To convert from Octal/hexa to decimal, first convert to binary and –then the binary to decimal.

Remark:-

BCD numbers are useful whenever decimal information is transferred into or out of a digital system. Examples of BCD systems are electronic ousters, digital voltmeter, and digital clocks; their circuits can work with BCD numbers.

BCD's are easy for conversion but slower for processing than binary. And they have limited numbers because with BCD we can represent only numbers 0000 for 0 and 1001 for 9 and ,1010,1011,1100,1101,1110, 1111 can't be used because 1010 represent 10 in decimal at 10 in decimal is 1010 0000 in BCD.

Hexadecimals are used for computation and memory addresses.

## **Binary Arithmetic**

Computer understands only the language of binary numbers. Therefore, the machine performs what is called binary arithmetic (binary computation).

### **Binary addition**

Binary addition operates by the same rule as decimal addition, except that it is simpler. A carry to the next higher order ( or more significant) position occurs when the sum is decimal 2, that is, binary 10. Therefore, the binary addition rules may be written as follows:

$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1=0 \text{ plus a carry of 1 into the next position}$$

$$1+1+1=1 \text{ plus a carry of 1 into the next position.}$$

The last case occurs when the two binary digits in a certain position are 1s and there is a carry from the previous position.

Example;

| Decimal | = | Binary     | Ex3 |                |
|---------|---|------------|-----|----------------|
| 6       | + | 110        |     | 10011          |
|         |   |            |     | + 11111        |
|         |   |            |     | 1010           |
| 7       |   | <u>111</u> |     |                |
| 13      |   | 1101       |     | <u>1001</u>    |
|         |   |            |     | <u>1000101</u> |

Ex2

$$\begin{array}{r} 100101 \\ \underline{11111} \\ 1000100 \end{array}$$

Ex4

$$\begin{array}{r} 1101.1101 \\ + \quad \underline{10.0100} \\ \underline{10000.0001} \end{array}$$

### **Binary Subtraction**

It operates by the same rule as decimal subtraction. The rule is as follows;

$$0-0=0$$

$$1-0=1$$

$$1-1=0$$

$$10-1=1$$

Example:

|                |               |                  |
|----------------|---------------|------------------|
| 11100          | 101101        | 11001.011        |
| - <u>11010</u> | - <u>111</u>  | - <u>111.110</u> |
| <u>00010</u>   | <u>101100</u> | <u>10001.101</u> |

### Binary Multiplication:

- It is a very simple process that operates by the following obvious rules:

Multiplying any number by 1 will not change the multiplicand

$$0 \times 1 = 0$$

$$1 \times 1 = 1$$

Multiplying any number by 0 produces 0

$$0 \times 0 = 0$$

$$1 \times 0 = 0$$

Ex1: 1001 x 1110

$$\begin{array}{r}
 1001 \\
 \times 1110 \\
 \hline
 0000 \\
 1001 \\
 1001 \\
 1001 \\
 \hline
 1111110
 \end{array}$$

### Binary division

That is, the process for dividing one binary number (the dividend) by another (the divisor) is based on the rules for binary subtraction and multiplication and

Similar to decimal division

$$\begin{array}{r} \text{Ex } 1111101 \mid 11001 \\ \underline{11001} \quad \underline{101} \\ 11001 \\ \underline{11001} \\ 00000 \\ 1111101 \div 11001 = \underline{101} \end{array}$$

### Representation of Negative numbers

- There are three ways of representing negative numbers in a computer
  - Using Sign Bit
  - One's Complement
  - Two's Complement

#### 1.Sign- magnitude representation.

In signed binary representation, the left-most bit is used to indicate the sign of the number. Traditionally, 0 is used to denote a positive number and 1 is used to denote a negative number. But the magnitude part will be the same for the negative and positive values.

For example **11111111** represents **-127** while **01111111** represents **+ 127**.

We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127.

In a 5- bit representation, we use the first bit for sign and the remaining 4- bits for the magnitude. So using this 5 bit representation the range of numbers that can be represented is from -15 (11111) to 15(11111)

Ex1: Represent-12 using 5-bit **sign magnitude** representation.

**Solution:** First, we convert 12 to binary i. e 1100

Now -12 = 11100

**Ex2:** Represent -24 using 8-bits

$$24 = 00011000$$

$$-24 = 1001100$$

In general, for n-bit sign –magnitude representation, the range of values that can be represented are  $-(2^{n-1}-1)$  to  $(2^{n-1}-1)$ .

**i.e.**  $2^{n-1} + 1$  to  $2^{n-1} - 1$

**Note:** In sign magnitude representation zero can be represented as 0 or -0

This representation has two problems one is it reduces the maximum magnitude, and the second one is speed efficiency.

To see the second problem let us perform addition in the signed binary representation. We want to add +7 and -5

+7 represented by 00000111

-5 represented by 10000101

The binary sum is = 10001100, or -12

This is not the correct result. The correct result is +2. We have neglected the fact that in order to use this representation, special actions must be taken, depending on the sign. This results in increased complexity and reduced performance. In other words, the binary addition of signed numbers does not “Work Correctly”.

The solution to this problem is called **the two’s complement representation**.

## **2. One’s complement.**

In one’s complement representation, all positive integers are represented in their correct binary format. For example +3 is represented as usual by 00000011. However, its complement, -3, is obtained by complementing every bit in the original representation. Each 0 is transformed into a1 and each 1 into a0. In our example, the one’s complement representation of -3 is 11111111.

Ex: +2 is 00000010

-2 is 11111101

Note that in this representation positive numbers start with a 0 on the left, and negative numbers start with a 1 on the left.

Ex1. Add -4 and +6

- 4 is 11111011

+ 6 is 00000110

the sum is (1) 00000001

Where 1 indicates a carry. The correct result should be 2 or 00000010.

Ex2:

-3+-2

-3 is 11111100

-2 is 11111101

11111001 or -6 plus a carry. The correct result is -5. The representation of -5 is 11111010. This representation does represent positive and negative numbers: however, the result of an ordinary addition does not always come out correctly. We will now use another representation. It is evolved from the one's complement and is called the two's complement representation.

## **2. Two's Complement Representation**

In two's complement representation, positive numbers are represented, as usual, in signed binary, just like in one's complement. The difference lies in the representation of negative numbers. A negative number represented in two's complement is obtained by first computing the one's complement and then adding one.

Ex: +3 is represented in signed binary as 00000011

Its one's complement representation is 11111100.

The two's complement is obtained by adding one.

It is 11111101.

Ex let's try addition.

$$\begin{array}{r} (3) \quad 00000011 \\ + (5) + 00000101 \\ \hline (8) \quad 0001000 \end{array}$$

The result is correct

Ex. Let's try subtraction

$$\begin{array}{r}
 (3) \quad 00000011 \\
 \underline{(-5) + 11111011} \\
 11111110
 \end{array}$$

Ex. add +4 and -3(the subtraction is performed by adding the two's complement).

+4 is 00000100

-3 is 11111101

The result is [1] 00000001

If we ignore the carry the result is 00000001 ( i. e 1 In decimal). This is the correct result. In two's complement, it is possible to add or subtract signed numbers, regardless of the sign. Using the usual rules of binary addition, the result comes out correct, including the sign. The carry is ignored. One's complement may be used, but if one's complement is used, special circuitry is required to "correct the result".

### Carry and overflow

$$\begin{array}{r}
 \text{Ex} \quad (128) \quad 10000000 \\
 \underline{+ (129) = 10000001} \\
 [257] \quad = (1) 00000001
 \end{array}$$

Where 1 indicates a carry. The result requires a ninth bit (bit 8, since the right- most bit is 0). It is the carry bit.

If we assume that the carry is the ninth bit of 100000001= 257. However, the carry must be recognized and handled with care. Inside the microprocessor, the registers used to hold information are generally only eight bits wide. When storing the result, only bits 0 to 7 will be preserved.

### **Overflow**

$$\begin{array}{r}
 \text{Ex1} \quad (64) \quad 01000000 \\
 \underline{(65) \quad 01000001} \\
 10000001
 \end{array}$$

Internal carry has been generated from bit 6 into bit 7. This called an overflow. The result is now negative by “accident”. This situation must be detected, so that it can be corrected.

Ex2: (-1) 11111111

$$\begin{array}{r} 11111111 \\ \hline (-2) (1) 11111110 \end{array}$$

Carry.

The rules of two’s complement arithmetic specify that this carry should be ignored. The result is then correct. This is because the carry from bit 6 to bit 7 didn’t change the sign bit.

The carry from bit 6 into bit 7 is not an overflow condition.

**Overflow** will occur in four situations, including:-

1. The addition of large positive numbers.
2. The addition of large negative numbers.

$$\begin{array}{r} \text{Ex } (-64) \quad 11000000 \\ \hline (-65) \quad 10111111 \\ \hline (+127) (1) 01111111 \end{array}$$

Carry ↙

It is carry but the result is incorrect because bit 7 has been changed.

3. The subtraction of a large positive number from a large negative numbers.
4. The subtraction of a large negative number from a large positive number.

Overflow indicates that the result of an addition or subtraction requires more bits than are available in the standard 8-bit register used to contain the result.

Fixed format representation: We now know how to represent signed integers: however, we have not yet resolved the problem of magnitude. If we want to represent large integers, we will need several bytes. In order to perform arithmetic operations efficiently, it is necessary to use a fixed number of bytes, rather than a variable number. Therefore, once the number of bytes is chosen, the maximum magnitude of the number that can be represented is fixed.

### Subtraction by use of complements

Complements are mainly used for representing negative numbers and subtraction.

In performing binary subtraction or addition of negative number by use of binary complements only one procedure, addition, is needed as one can subtract by adding its complements.

To subtract any number, positive or negative, substitute the required complement for the numbers to be subtracted and then add.

If the result is (n+1)-bit number, and the arithmetic is in ones complement the (n+1)<sup>th</sup> bit, a carry, is added to the right most bit of the result. This process is called an end-around carry.

Two's complement discard the (n+1)<sup>th</sup> bit.

An n-bit number and the arithmetic is in ones complement, to read the binary value, calculate the ones complement of the magnitude and place a minus sign in front of it.

Two's complement, to read the binary value, calculate the two's complement of the magnitude and place a minus sign in front of it.

Example:

Perform the following in ones and two's complements in 5-bits.

a. 12-6

b. 6-12

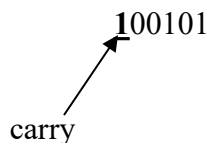
c. -12-6

a) In ones complement:

ones complement of 6=11001

12        01100

-6        + 11001

100101  
  
 carry

Add the carry at the LSB.

The result is 00101

      +1  
 (6)    00110

In two's complement:

two's complement of 6 = 11001 + 1 = 11010

12        01100

-6        + 11010

100110

Carry to be discarded. The result is 00110 which is correct result.

b) In ones complement:

1's complement of 12 is 10011

|            |                        |
|------------|------------------------|
| 6          | 00110                  |
| <u>-12</u> | <u>10011</u>           |
| -6         | 11001 which is correct |

In 2's complement:

2's complement of 12 is: 10011 + 1 = 10100

|            |                         |
|------------|-------------------------|
| - 6        | 00110                   |
| <u>-12</u> | <u>10100</u>            |
| -6         | 11010 ,which is correct |

c) In 1's complement:

|           |              |
|-----------|--------------|
| -12       | 10011        |
| <u>-6</u> | <u>11001</u> |
| -18       | 101100       |

Carry to be added

i.e. The result is 01100 + 1 = 01101 which is wrong

In 2's complement:

2's complement of 12 is: 10011 + 1 = 10100

2's complement of 6 is 11001

|           |                |
|-----------|----------------|
| -12       | 10100          |
| <u>-6</u> | <u>+ 11010</u> |
| -18       | 101110         |

Carry to be discarded. The result is 01110 which is wrong.

Ex. Subtract 110.1001 from 1010.100 using 8-bits of two's complement.

## Floating Point Numbers

### Floating-Point Representation

In this representation decimal numbers are represented with a fixed length format. In order not to waste bits, the representation will normalize all the numbers. For example, 0.000123 wastes three zeroes on the left before non-zero digits. These zeroes have no meaning except to indicate the position of the Decimal point. Normalizing this number results in  $.123 \times 10^{-3}$ . 123 is the normalized mantissa; -3 is the exponent. We have normalized this by eliminating all the meaningless zeroes to the left of the first non-zero digit and by adjusting the exponent.

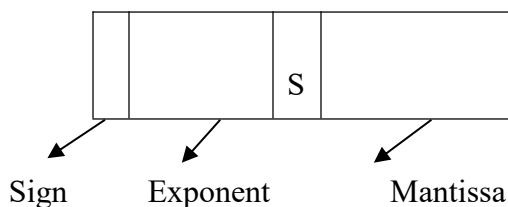
Ex1: 22.1 is normalized as  $.221 \times 10^2$ .

The general form of floating point representation is  $M \times 10^E$  where M is the mantissa, and E is the exponent. It can be seen that a normalized number is characterized by a **mantissa less than 1** and greater than or equal to 0.1 in all cases when the number is not zero.

Ex2 111.01 is normalized as  $.11101 \times 2^3$ .

The mantissa is 11101. The exponent is 3.

The general structure of floating point is



In representing a number in floating point, we use 1 bit for sign, some bits for exponent and the remaining bits for mantissa.

In floating point representation the exponent is represented by a biased exponent (Characteristic).

**Biased exponent** = true exponent + excess  $2^{n-1}$ , where n is the number of bits representing the exponent.

Ex1. Represent -234.375 in floating point using 7 bit for exponent and 16 bit for mantissa.

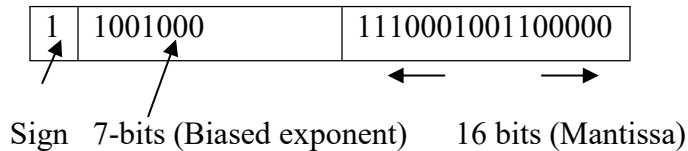
First we have to change to normalized binary

i. e.  $234 = 11100010$  and  $0.375 = 0.011$

$$234.375 = 11100010.011 = 0.11100010011 \times 2^8$$

- true exponent = 8
- excess  $2^{n-1} = 2^{7-1} = 2^6 = 64$
- Biased exponent =  $8 + 2^6 = 8 + 64 = 72 = 1001000_2$

Therefore  $-234.375$  is represented as:



Ex2. Represent 34.25 in floating point using 7 bit for exponent and 24 bits for mantissa.

$$34.25 = 100010.01_2$$

- The normalized form in binary of  $34.25 = .10001001 \times 2^6$
- True exponent = 6
- Excess  $2^{n-1} = 2^{7-1} = 64$
- Biased exponent =  $6 + 64 = 70 = 1000110_2$

Therefore, 34.25 is represented as:

|   |         |                    |
|---|---------|--------------------|
| 0 | 1000110 | 100010010000.....0 |
|---|---------|--------------------|

To represent a number in floating point:

- Represent the number in normalized binary form.
- Find the biased exponent
- Change the biased exponent to binary
- Write the sign, the exponent in the exponent part and the mantissa in the mantissa form
- If there are fewer digits in the exponent add zeros to the left and for mantissa add zeros to the right.

### Floating-point Arithmetic

To perform floating-point arithmetic:

- First correct the numbers to normalized binary of the same exponent
- Apply the operator
- If necessary normalize the result

Ex1. Find  $23.375 + 41.25$  using 7-bit for exponent and 10 bit for mantissa.

$$23.375 = 10111.011_2 = 0.1011101 \times 2^5 = 0.010111011 \times 2^6$$

$$41.25 = 111001.01_2 = 0.11100101 \times 2^6$$

$$\begin{aligned} 23.37 + 41.25 &= 0.01011101 \times 2^6 + 0.1110010 \times 2^6 \\ &= (0.010111011 + 0.11100101) \times 2^6 \\ &= 0.1010000101 \times 2^6 \end{aligned}$$