Chapter Four

Data Representation in Computers

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. So data are represented in the form of a code that can have a corresponding electrical signal.

4.4 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 of 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 using a single byte 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-bit, 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 larger the word length a computer has the more powerful and faster it is.

4.4 Concept of Number Systems and Binary Arithmetic

Since the early days of human civilization, people have been using their fingers, sticks, and other things for counting. As daily activities became more complex, numbers became more important in trade, time, distance, and in all spheres of human life. A number system defines a set of values used to represent *quantity*. There are various number systems e.g. decimal, binary, octal, hexadecimal, etc each differs one another by the number of symbols used in the system. Each numbering system used different symbols to represent a given quantity.

For a computer, everything is a number whether it may be numbers, alphabets, punctuation marks, its own instructions, etc. The number systems that are generally used by computers are: decimal, binary, octal, and hexadecimal.

4.4.4 The Decimal Number System

The primary number system used is a base ten number system or *decimal number system*. The Decimal number system is based on the ten different digits or symbols (0,1,2,3,4,5,6,7,8,9).

Starting at the decimal point and moving to the left, each position is represented by the base (radix) value (10 for decimal) raised to power. The power starts at Zero for the position just to the left of the decimal point. The power incremented for each positions that continues to the left. Moving to the right of the decimal point is just like moving to

the left except that we will need to place a minus sign in front of each power.

For example: $(8762)_{10} = (8*10^3) + (7*10^2) + (6*10^1) + (2*10^0)$

$(0.475)_{10} = (4*10^{-1}) + (7*10^{-2}) + (5*10^{-3})$

4.4.4 The Binary number system

Computers do not use the ten digits of the decimal system for counting and arithmetic. Their internal structure (mainly the CPU and memory) are made up of millions of tiny switches that can be either in an ON or OFF states. Two digits, 0 and 1, are used to refer for these two states.

Binary number system is based on the two different digits; 0 and 1. With binary number system, it is very easier for the hardware to represent the data. Binary number system is base two number system.

For example: $(01100)_2$, $(10110.011)_2$, etc

4.4.4 Octal number system

The octal number system with its eight symbols (0, 1, 2, 3, 4, 5, 6, 7) is a base 8 system.

For example: $(322)_8$, $(10.25)_8$, etc

4.4.4 Hexadecimal number system

Hexadecimal number system is another number system that works exactly like the decimal and binary number systems, except that the base is 16. It uses 16 symbols (0-9, and A-F characters to represent 10-15).

For example: $(8F0)_{16}$, $(D.45)_{16}$, etc

4.4.4 Conversion between Number Systems

Computers use binary numbers for internal data representation whereas they use decimal numbers externally. Therefore, there should be some conversion between number systems in order to represent data in a computer that is originally represented in other number systems. Some conversion methods are discussed below.

Decimal to Binary

It is important to note that every decimal number system has its equivalent binary number. For example

Binary	Decimal
0	0

01	1	110	6	
10	2	111	7	
11	3	1000	8	
100	4	1001	9	etc.

Binary	Decimal
101	5

Conversion from binary to its equivalent decimal and from decimal to its equivalent binary is possible. The method, which is used for the conversion of decimal into binary, is often called as the remainder method. This method involves the following steps.

- Begin by dividing the decimal number by 2 (the base of binary number system)
- Note the remainder separately as the rightmost digit of the binary equivalent
- Continually repeat the process of dividing by 2 until quotient is zero and keep writing the remainders after each step of division (these remainders will either be 0 or 1)
- Finally, when no more division can occur, write down the remainders in reverse order (last remainder written first)

2	44
2	22
2	11
2	5
2	2
2	1
	1

Example: Determine the binary equivalent of $(44)_{10}$

Remainder

0
0
1
1
0

MSB (Most Significant Bit)

LSB (List Significant Bit)

Taking the remainder in reverse order we have 101100. Thus the binary equivalent of $(44)_{10}$ is $(101100)_2$

In general to convert a decimal number X to a number in base M, divide X by M, store the remainder, again divide the quotient by M, store the remainder, and continue until the quotient is 0. And concatenate (collect) the remainders starting from the last up to the first.

Example: Convert 78_{10} to base eight (Octal)

$78_{10}=116_8$

Example: Convert 30_{10} to base sixteen (hexadecimal)

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

Binary to Decimal

In the binary to decimal conversion, each digit of the binary number is multiplied by its weighted position, and each of the weighted values is added together to get the decimal number.

Example: Determine the decimal equivalent of $(100100)_2$

$1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 32+4 = 36$

Therefore, the decimal equivalent of $(100100)_2$ is 36

In general To convert a number X consists of digits $X_1\ X_2\ X_3 \cdots X_n$ in base m to decimal; simply expand the number with base m. That is

$$(X_1X_2X_3\cdots X_n)_m = X_1*m^{n-1}+X_2*m^{n-2}+X_3*m^{n-3}+\ldots+X_{i*}m^{n-i}+\cdots X_{n-1}m^1+X_n*m^0$$

$$=Y_{10}$$

Example: convert $(234)_8$ to decimal

$=2*8^2 + 3*8^1 + 4*8^0 = 128+24+4 = 156$

Example: convert $(A1B)_{16}$ to decimal

$=A*16^2 + 1*16^1 + B*16^0 = 2587$

Binary (base2) to Octal (base 8) or hexadecimal (base16) and vice versa

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 to the front end (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.

Example: Convert 111100100 to hexadecimal

111100100 =0001 1110 0100

=1 14 4

=1 E 4

$= (1E4)_{16}$

Convert 111001111 to Hexadecimal

111001111 =0001 1100 1111

=1 12 15

=1 B F

$=(1BF)_{16}$

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

Example: Convert $(675)_{eight}$ to binary

675_{eight} =110 111 101

$=(110111101)_{two}$

Convert 231_{eight} to binary

231_{eight} = 010 011 001

$=(10011001)_{two}$

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

Example: Convert 234_{16} to binary

234_{16} =0010 0011 0100

$= 1000110100_2$

Convert 2AC to binary

$2AC_{16}$ =0010 1010 1100

$=1010101100_2$

Octal to hexadecimal and Vise versa

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.

Example: Convert 235_8 to hexadecimal

238_8=010 011 101

=0000 1001 1101

= 0 9 13

$=9D_{16}$

Convert $(1A)_{16}$ to Octal

1A=0001 1010

=000 011 010

= 0 3 2

$=32_8$

Summary of conversion from One base to another base

From base	To base	Method

2	10	Expand binary number in powers of 2
10	2	Factor the decimal number by 2
2	8	Group 3 binary digits together
8	2	Each Octal digit is converted to 3 binary digits
2	16	Group 4 binary digits together
16	2	Each hexadecimal digit is converted to 4 binary digits
8	10	Expand the number in powers of 8
10	8	Factor the decimal number by 8
16	10	Expand the number in powers of 16
10	16	Factor the decimal number by 16
8	16	Go from 8 …….2……..16
16	8	Go from 16 …….2……..8

Converting Decimal Number with Fractions to Binary

- 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.

Example. A) Convert 12.25_{10} to binary 1100.01

B) Convert3.1875 to binary 11.0011

Converting Binary with Fraction to Decimal

To convert a binary number $Y_1Y_2Y_3Y_4Y_n.d_1d_2d_3\cdots d_m$ to decimal

- first convert the integer part to decimal by using

$y_1\, y_2\, y_3\, y_4\cdots y_n = y_1*2^{n-1}+y_2*2^{n-2}+\cdots.y_j*2^{n-j}+\cdots.+y_{n-1}*2^1+y_n*2^0=Q$ and

- Convert the fractional part to decimal by using

$d1d2d3\cdots dm=d_1*2^{-1}+d_2*2^{-2}+d_3*2^{-3}+\cdots+d_j*2^{-j}+..+d_m*2^{-m}=R$

- Then decimal equivalence of $y_1\ y_2\, y_3\, y_4\cdots..y_n.d_1d_2\cdots d_m$ will be Q+R where Q is the integer part and R is the fractional part.

Ex1 : Convert 11001-0101 to decimal =>11001.0101 = 25.3125.

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

$0101 = 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4}$

$= 0 + ¼ + 0 + 1/16 = 0.3125 = R$

Ex 2: Convert 1000.1 to decimal

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

$1= 1x2^{-1} = ½ = 0.5$

$1000.1 = 8.5_{10}$

Conversion from Binary with Fraction to Octal/Hexadecimal

- Group three/four digits together starting from the last digit of the integer part, and if there is less number of digits add some zeros in the beginning.
- Group three/ four digits together starting from the first digit of the fractional part, and if there is less number of digits add some zeros to the end.
- Covert each group of the integer and the fractional part to their equivalent Octal/hexadecimal and collect the results by adding point (.) to separate the integer part from the fractional part.

Ex 1:- Covert 1010.0111_2 to octal

Ex2:- Covert 1110101.10111_2 to hexadecimal

Conversion from Octal or Hexadecimal with Fraction to Binary

- Convert each Octal/hexadecimal digit to its equivalent 3/4-bit binary digit.
- Collect the binary sequences by separating the integer part binaries from the fractional part binaries with point (.)

Conversion from Octal with Fraction to Hexadecimal

- To convert from Octal to hexadecimal, first convert the Octal to binary and then the binary to hexadecimal

Conversion from Hexadecimal with Fraction to Octal

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

Conversion from Octal/Hexadecimal with Fraction to Decimal.

¨ To convert from Octal/hexadecimal to decimal, first convert to binary and -then the binary to decimal.

4.4.4 Binary Arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals. The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10 (carry:1)

Adding two "1" values produces the value "10" (spoken as "one-zero"), equivalent to the decimal value 2. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result equals or exceeds the value of the radix (10), the digit to the left is incremented: e. g 5 + 5 = 10 , 7 + 9 = 16 .This is known as *carrying* in most numeral systems. When the result of an addition exceeds the value of the radix, the procedure is to "carry the one" to the left, adding it to the next positional value. Carrying works the same way in binary:

```
  1 1 1 1 1  (carried digits)
  0 1 1 0 1
+ 1 0 1 1 1
-----------
= 1 0 0 1 0 0
```

In this example, two numerals are being added together: 01101_2 (13 decimal) and 10111_2 (23 decimal). The top row shows the carry bits used. Starting in the rightmost column, 1 + 1 = 10_2. The 1 is carried to the left, and the 0 is written at the bottom of the rightmost column. The second column from the right is added: 1 + 0 + 1 = 10_2 again; the 1 is carried, and 0 is written at the bottom. The third column: 1 + 1 + 1 = 11_2. This time, a 1 is carried, and a 1 is written in the bottom row. Proceeding like this gives the final answer 100100_2 (36 decimal).

Subtraction works in much the same way:

0 −0 = 0

0 −1 = 1 (with borrow)

1 −0 = 1

1 −1 = 0

One binary numeral can be subtracted from another as follows:

```
  * * *   (starred columns are borrowed from)
    1 0 0 1 0 1
  −   1 0 1 1
  ---------------
  = 0 1 1 0 1 0
```

The following steps are involved:

- First, for the least significant bit(the right most bit) , 1-1 is 0
- For the next bit, 0-1 cannot be computed since the subtrahend is smaller than the minuend. Borrow 1 from the third bit to form the binary number 10 (decimal 2) and do the subtraction. The operation is 10-1=1 which in decimal number system is 2-1=1
- For the third bit, since we borrowed 1 for the second bit, we have 0-0 that is 0
- For the forth bit again, we cannot perform the subtraction. However the fifth bit in the minuend is zero, so we must borrow from the sixth bit. This makes the fifth bit 10 (decimal 2). Borrowing from the fifth bit makes it 1 and the fourth bit become 10 (decimal 2). Now the subtraction in binary is 10-1=1 which is the result of the fourth bit.
- For the fifth bit, we now have 1-0=1
- Since we borrowed 1 from the sixth bit for the fourth bit, so for the sixth bit, the subtraction is 0-0=0

Multiplication in binary is similar to its decimal counterpart. Two numbers *A* and *B* can be multiplied by partial products: for each digit in *B*, the product of that digit in *A* is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in *B* that was used. The sum of all these partial products gives the final result.

Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

For example, the binary numbers 1011 and 1010 are multiplied as follows:

```
    1 0 1 1   (A)
          × 1 0 1 0   (B)
    ---------
        0 0 0 0   ← Corresponds to a zero in B
  +     1 0 1 1     ← Corresponds to a one in B
  +   0 0 0 0
  + 1 0 1 1
  ---------------
= 1 1 0 1 1 1 0
```

Binary *Division* is again similar to its decimal counterpart:

```
     _________
1 0 1 | 1 1 0 1 1
```

Here, the divisor is 101_2, or 5 decimal, while the dividend is 11011_2, or 27 decimal. The procedure is the same as that of decimal long division; here, the divisor 101_2 goes into the first three digits 110_2 of the dividend one time, so a "1" is written on the top line. This result is multiplied by the divisor, and subtracted from the first three digits of the dividend; the next digit (a "1") is included to obtain a new three-digit sequence:

```
         1
     _________
1 0 1 | 1 1 0 1 1
      −1 0 1
       -----
         0 1 1
```

The procedure is then repeated with the new sequence, continuing until the digits in the dividend have been exhausted:

```
          1 0 1
        ---------
1 0 1 | 1 1 0 1 1
      −1 0 1
       -----
         0 1 1
        −0 0 0
         -----
           1 1 1
          −1 0 1
           -----
             1 0
```

Thus, the quotient of 11011_2 divided by 101_2 is 101_2, as shown on the top line, while the remainder, shown on the bottom line, is 10_2. In decimal, 27 divided by 5 is 5, with a remainder of 2.

4.4 CODING METHODS

In today's technology, the binary number system is used by the computer system to represent the data in the computer in understandable format. There are a lot of ways to represent, numeric, alphabetic, and special characters in computer's internal storage area. 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 equivalent arrangements of bits. In this way, every character can be represented by a combination of bits that is different from any other combination.

There are different coding systems that convert one or more character sets into computer codes. Some are: EBCDIC, BCD, ASCII-7 & ASCII-8, Unicode, etc.

In these encodings, binary coding schemes separate the characters, known as character set, in to zones. Zone groups characters together so as to make the coding scheme to decipher and the data easier to process. With in each zone, the individual characters are identified by digit code.

EBCDIC: Pronounced as "Eb-see-dick" and stands for Extended Binary Coded Decimal Interchange Code.

It is an 8-bit coding scheme: (00000000 - 11111111), i.e. it uses 8 bits to represent each character. It accommodates to code 2^8 or 256 different characters. This provides a unique

code for each decimal value 0 to 9 , each upper and lower case English letter (for total of 52), and for a variety of special characters. Since it is an 8-bit code, each group of the eight bits makes up one alphabetic, numeric, or special character. It is a standard coding scheme for the large computers.

Coding Examples

EBCDIC

Character	zone (4 Bit)	digit (4 Bit)
0-9	15	0-9
a-i	8	1-9
j-r	9	1-9
s-z	10	2-9
A-I	12	1-9
9	1111	1001

J-R	13	1-9
S-Z	14	2-9

Character	Zone	Digit
a	1000	0001
b	1000	0010
A	1100	0001
B	1100	0010
0	1111	0000

BCD (Binary Coded Decimal)

There were two types of BCD coding techniques used before. The 4 bit BCD, which represent any digit of decimal number by four bits of binary numbers.

If you want to represent 219 using 4 bit BCD you have to say 0010 0001 1001

- 4 bits 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 .

BCD (6-bits)

It uses 6-bits to code a Character (2 for zone bit and 4 for digit bit) it can represent 2^6 = 64 characters (10 digits, 26 capital characters and some other special characters).

Some Coding Examples

Character	zone (2 Bit)	digit(4 Bit)
0-9	0	0-9
A-I	3	1-9

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

ASCII-7

ASCII stands for American Standard Code for Information Interchange. ASCII-7 used widely before the introduction of ASCII-8 (the Extended ASCII). It uses 7 bits to represent a character. With the seven bits, 2^7(or 128) different characters can be coded (0000000-1111111). It has 3 zone and 4 digit bits positions

A-O	4	1-15
P-Z	5	1-10

Coding examples:

Charter	**zone (3 bit)**	**digit(4 bit)**
0-9	3	0-9
A	100	0001
a	110	0001
b	110	0010

$	010	0100
%	010	0101

The ASCII System

Also referred as ASCII-8 or Extended ASCII. It is commonly used in the transmission of data through data communication and is used almost exclusively to represent data internally in microcomputers. 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). It assigns 4 bits for the zone and the rest for the digit.

Coding Examples:

Character	zone (3 BIT)	digit (4 BIT)	b	0110	0010
0-9	3	0-9	A	0100	0001
A-O	4	1-15	B	0100	0010
P-Z	5	0-10	?	0011	1111
a-o	6	1-15	+	0010	1011
p-z	7	0-10	1	0011	0001

a	0110	0001

Unicode

Unicode has started to replace ASCII and other coding methods at all levels. It enables users to handle not only practically any script and language used on this planet; it also supports a comprehensive set of mathematical and technical symbols to simplify scientific information exchange. Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. Unicode was originally designed to be a 16-bit code, but it was extended so that currently code positions are expressed as integers in the hexadecimal range 0..10FFFF (decimal 0..1 114 111).

4.4 Representation of Negative Numbers and Arithmetic

There are different ways of representing negative numbers in a computer.

I. ***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 number that can be represented is from -15 (11111) to 15 (01111)

Example 1: Represent-12 using 5-bits sign magnitude representation

- first we convert 12 to binary i. e 1100
- Now -12 = 11100

Example 2: Represent -24 using 8-bits sign magnitude representation

24=00011000

-24 = 10011000

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

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

This representation has two problems one is it reduces the maximum size of magnitude,

and the second one is speed efficiency to perform arithmetic and other operations. For sign magnitude representation, correct addition and subtraction are relatively complex, involving the comparison of signs and relative magnitude of the two numbers. The solution to this problem is called the complement representation.

II. 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 a 1 and each 1 into a 0. In our example, the one's complement representation of -3 is 11111100.

Example: +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 most bit.

Example 1: add -3 and 3 with word size 4

3 = 0011

-3=1100

sum =1111 (=0)

Ex2. 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.

In one's complement addition and subtraction, if there is an external carry it should be added to get the correct result. This indicates it requires additional circuitry for implementing this operation.

III. *Two's Complement Representation*

In two's complement representation, positive numbers are represented, as usual, in singed 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 add one.

Example: +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.

Example: let's try addition.

(3) 00000011

\+ (5) +00000101

(8) 0001000

The result is correct

Example: Let's try subtraction

(3) 00000011

(-5) + 111111011

11111110

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

+4 is 00000100

-3 is 111111101

The result is [1] 000000001

If we ignore the external 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

Ex (128) 10000000

+(129) 10000001

[257]=(1) 00000001

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.

The two's complement representation has one anomaly not found with sign magnitude or one's complement. The bit pattern 1 followed by N-1 zeros is its own 2's complement. To

maintain sign bit consistency, this bit pattern is assigned the value -2^N for example, for 8 -bit word,

-128 = 10000000

its 1's complement =01111111

+1

=100000000 = -128

Overflow will occur in four situations, including: -

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

- An (n+1)-bit number, and the arithmetic is in Ones complement the $(n+1)^{th}$ bit, a carry, is added to the right most bit of the result. This process is called an end-around carry. If it is in Two's complement discard the $(n+1)^{th}$ bit.
- An n-bit number and the arithmetic is in Ones complement, to read the binary value calculate the ones complement of the magnitude bits 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 bits 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 = 12 B = 6, A = 01100 B = 00110

Ones complement of -A=10011 & -B=11001

Two's complement of - A= 10100 & -B= 11010

Example c: Is wrong this is because the occurrence of overflow. Arithmetic overflow is that part of the result of an operation which is lost because of the resulting value exceeds the capacity of the intended storage location.

- Arithmetic overflow occurs when the sign bits of A and B are the same but the sign bit of the result is different.

4.4 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 result in $.123x10^{-3}$ .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.

Example: 22.1 is normalized as $.221x10^2$.

The general form of floating point representation is $\pm Mx10^{\pm E}$ 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.1 all cases when the number is not zero.

To represent floating numbers in the computer system it should be normalized after converting to binary number representation system.

Example: 111.01 is normalized as $.11101x2^3$.

The mantissa is 11101. The exponent is 3.

The general structure of floating point is

Sign Exponent Mantissa (significand)

In representing a number in floating point we use 1 bit for sign, some bits for exponent and the remaining bit for mantissa. In floating point representation the exponent is represented by a biased exponent (Characteristics).

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

Example: 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

0.375= 0.011

$234.375 = 11100010.011 = 0.11100010011x2^8$

true exponent = 8

excess $2^{n-1} = 2^{7-1} = 2^6 = 64$

Biased exponent = $8+2^6 = 8+64 = 72 = (100\ 1000)_2$

Therefore -234.375 is represented as

1	1001000	1110001001100000

Sign 7-bits 16 bits

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

$34.25 = 100010.01_2$

The normalized form of $34.25 = .10001001 \times 2^6$

True exponent = 6

Excess $2^{n-1} = 2^{7-1} = 6+2^6$

Biased exponent =6+64=70

$70 = 1000110_2$

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 part
- 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 binary with the same exponent (the highest)
- Apply the operator on the mantissa and take one of the exponent
- Normalize the result

Example: Find 23.375 + 41.25 using 7-bit for exponent and 10 bit for mantissa.

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

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

$23.37+41.25 = 0.01011101 \times 2^6 + 0.1110010 \times 2^6$

$= (0.010111011+0.11100101) \times 2^6$

$= 0.1010000101 \times 2^6$