UNIT – I
DIGITAL SYSTEMS AND BINARY NUMBERS

1.1 Introduction:

A digital computer stores data in terms of digits (numbers) and proceeds in discrete steps from one state to the next. The states of a digital computer typically involve binary digits which may take the form of the presence or absence of magnetic markers in a storage medium, on-off switches or relays. In digital computers, even letters, words and whole texts are represented digitally.

Digital Logic is the basis of electronic systems, such as computers and cell phones. Digital Logic is rooted in binary code, a series of zeroes and ones each having an opposite value. This system facilitates the design of electronic circuits that convey information, including logic gates. Digital Logic gate functions include and, or and not. The value system translates input signals into specific output. Digital Logic facilitates computing, robotics and other electronic applications.

Digital Logic Design is foundational to the fields of electrical engineering and computer engineering. Digital Logic designers build complex electronic components that use both electrical and computational characteristics. These characteristics may involve power, current, logical function, protocol and user input. Digital Logic Design is used to develop hardware, such as circuit boards and microchip processors. This hardware processes user input, system protocol and other data in computers, navigational systems, cell phones or other high-tech systems.

1.2 Data Representation and Number system:

1.2.1 Numeric systems:

The numeric system we use daily is the decimal system, but this system is not convenient for machines since the information is handled codified in the shape of on or off bits; this way of codifying takes us to the necessity of knowing the positional calculation which will allow us to express a number in any base where we need it.

Radix number systems

A base of a number system or radix defines the range of values that a digit may have.

In the binary system or base 2, there can be only two values for each digit of a number, either a "0" or a "1".

In the octal system or base 8, there can be eight choices for each digit of a number:

"0", "1", "2", "3", "4", "5", "6", "7".

In the decimal system or base 10, there are ten different values for each digit of a number:

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

In the hexadecimal system, we allow 16 values for each digit of a number:

"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", and "F".

Where “A” stands for 10, “B” for 11 and so on. Conversion among radices

1.2.2 Convert from Decimal to Any Base:

Let’s think about what you do to obtain each digit. As an example, let's start with a decimal number 1234 and convert it to decimal notation. To extract the last digit, you move the decimal point left by one digit, which means that you divide the given number by its base 10.

$$1234/10 = 123 + 4/10$$

The remainder of 4 is the last digit. To extract the next last digit, you again move the decimal point left by one digit and see what drops out.

$$123/10 = 12 + 3/10$$

The remainder of 3 is the next last digit. You repeat this process until there is nothing left. Then you stop. In summary, you do the following:

```
Quotient  Remainder

     ----------------------------
     1234/10 =   123        4 --------+
      123/10 =    12        3 ------+ |
       12/10 =     1        2 ----+ | |
        1/10 =     0        1 --+ | | |(Stop when the quotient is 0)
                                | | | |
                                1 2 3 4   (Base 10)
```

Now, let's try a nontrivial example. Let's express a decimal number 1341 in binary notation. Note that the desired base is 2, so we repeatedly divide the given decimal number by 2.

```
          Quotient  Remainder
 ----------------------------
 1341/2  =   670        1 ----------------------+
  670/2  =   335        0 --------------------+ |
  335/2  =   167        1 ------------------+ | |
  167/2  =    83        1 ----------------+ | | |
   83/2  =    41        1 --------------+ | | | |
   41/2  =    20        1 ------------+ | | | | |
   20/2  =    10        0 ----------+ | | | | | |
   10/2  =     5        0 --------+ | | | | | | |
    5/2  =     2        1 ------+ | | | | | | | |
    2/2  =     1        0 ----+ | | | | | | | | |
    1/2  =     0        1 --+ | | | | | | | | | |(Stop when the
                            | | | | | | | | | | quotient is 0)
                            1 0 1 0 0 1 1 1 1 0 1  (BIN; Base 2)
```

Let's express the same decimal number 1341 in octal notation.

```
          Quotient  Remainder
 ----------------------------
 1341/8  =   167        5 --------+
  167/8  =    20        7 ------+ |
   20/8  =     2        4 ----+ | |
    2/8  =     0        2 --+ | | |  (Stop when the quotient is 0)
                            | | | |
                            2 4 7 5  (OCT; Base 8)
```

Let's express the same decimal number 1341 in hexadecimal notation.

```
          Quotient  Remainder
 ----------------------------
 1341/16 =    83       13 ------+
   83/16 =     5        3 ----+ |
    5/16 =     0        5 --+ | |  (Stop when the quotient is 0)
                            | | |
                            5 3 D  (HEX; Base 16)
```

In conclusion, the easiest way to convert fixed point numbers to any base is to convert each part separately. We begin by separating the number into its integer and fractional part. The integer part is converted using the remainder method, by using a successive division of the number by the base until a zero is obtained. At each division, the reminder is kept and then the new number in the base r is obtained by reading the remainder from the last remainder upwards.

The conversion of the fractional part can be obtained by successively multiplying the fraction with the base. If we iterate this process on the remaining fraction, then we will obtain successive significant digit. This methods form the basis of the multiplication methods of converting fractions between bases.

Example 1.1: Convert the decimal number 3315 to hexadecimal notation. What about the hexadecimal equivalent of the decimal number 3315.3?

Solution:

```
           Quotient  Remainder
-----------------------------
3315/16 =   207        3 ------+
 207/16 =    12       15 ----+ |
  12/16 =     0       12 --+ | |  (Stop when the quotient is 0)
                           | | |
                          C F 3  (HEX; Base 16)


                                    (HEX; Base 16)
            Product  Integer Part    0.4 C C C ...
--------------------------------      | | | |
0.3*16   =   4.8         4          ----+ | | | |
0.8*16   =  12.8        12          ------+ | | |
0.8*16   =  12.8        12          --------+ | |
0.8*16   =  12.8        12          ----------+ |
          :                         ------------------------+
          :
Thus, 3315.3 (DEC) --> CF3.4CCC... (HEX)
```

1.2.3 Convert From Any Base to Decimal:

Let's think more carefully what a decimal number means. For example, 1234 means that there are four boxes (digits); and there are 4 one's in the right-most box (least significant digit), 3 ten's in the next box, 2 hundred's in the next box, and finally 1 thousand's in the left-most box (most significant digit). The total is 1234:

```
Original Number:       1     2     3    4
                       |     |     |    |
How Many Tokens:       1     2     3    4
Digit/Token Value:  1000   100    10    1
Value:              1000 + 200  + 30  + 4  = 1234

or simply,  1*1000 + 2*100 + 3*10 + 4*1 = 1234
```

Thus, each digit has a value: 10^0=1 for the least significant digit, increasing to 10^1=10, 10^2=100, 10^3=1000, and so forth.

Likewise, the least significant digit in a hexadecimal number has a value of 16^0=1 for the least significant digit, increasing to 16^1=16 for the next digit, 16^2=256 for the next, 16^3=4096 for the next, and so forth. Thus, 1234 means that there are four boxes (digits); and there are 4 one's in the

right-most box (least significant digit), 3 sixteen's in the next box, 2 256's in the next, and 1 4096's in the left-most box (most significant digit). The total is:

$$1*4096 + 2*256 + 3*16 + 4*1 = 4660$$

In summary, the conversion from any base to base 10 can be obtained from the formulae

$$x_{10} = \sum_{i=-m}^{n-1} d_i b^i$$

Where b is the base, d_i is the digit at position i, m the number of digit after the decimal point, n the number of digits of the integer part and X10 is the obtained number in decimal. This form the basic of the polynomial method of converting numbers from any base to decimal

Example 1.2: Convert 234.14 expressed in an octal notation to decimal.

$2*82 + 3*81 + 4*80+1*8-1 + 4*8-2 \quad = 2*64 +3*8 +4*1 +1/8 +4/64 =156.1875$

Example 1.3: Convert the hexadecimal number 4B3 to decimal notation. What about the decimal equivalent of the hexadecimal number 4B3.3?

Solution:

```
Original Number:      4     B     3   .   3
                      |     |     |       |
How Many Tokens:      4    11     3       3
Digit/Token Value:  256    16     1       0.0625
Value:             1024  +176   + 3    + 0.1875   = 1203.1875
```

Example 1.4: Convert 234.14 expressed in an octal notation to decimal.

Solution:

```
Original Number:      2     3     4   .   1        4
                      |     |     |       |        |
How Many Tokens:      2     3     4       1        4
Digit/Token Value:   64     8     1       0.125    0.015625
Value:              128 + 24   + 4    + 0.125 + 0.0625     = 156.1875
```

1.2.4 Relationship between Binary - Octal and Binary-hexadecimal:

As demonstrated by the table below, there is a direct correspondence between the binary system and the octal system, with three binary digits corresponding to one octal digit. Likewise, four binary digits translate directly into one hexadecimal digit.

With such relationship, In order to convert a binary number to octal, we partition the base 2 number into groups of three starting from the radix point, and pad the outermost groups with 0's as needed to form triples. Then, we convert each triple to the octal equivalent.

For conversion from base 2 to base 16, we use groups of four.

Consider converting 101102 to base 8:

101102 = 0102 1102 = 28 68 = 268

Notice that the leftmost two bits are padded with a 0 on the left in order to create a full triplet.

BIN	OCT	HEX	DEC
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Now consider converting 101101102 to base 16:

101101102 = 10112 01102 = B16 616 = B616

(Note that 'B' is a base 16 digit corresponding to 1110. B is not a variable.)

The conversion methods can be used to convert a number from any base to any other base, but it may not be very intuitive to convert something like 513.03 to base 7. As an aid in performing an unnatural conversion, we can convert to the more familiar base 10 form as an intermediate step, and then continue the conversion from base 10 to the target base. As a general rule, we use the polynomial method when converting into base 10, and we use the remainder and multiplication methods when converting out of base 10.

1.3 Numeric complements:

The radix complement of an n digit number y in radix b is, by definition, $b^n - y$. Adding this to x results in the value $x + b^n - y$ or $x - y + b^n$. Assuming $y \leq x$, the result will always be greater than b^n and dropping the initial '1' is the same as subtracting b^n, making the result $x - y + b^n - b^n$ or just $x - y$, the desired result.

The radix complement is most easily obtained by adding 1 to the diminished radix complement, which is $(b^n - 1) - y$. Since $(b^n - 1)$ is the digit $b - 1$ repeated n times (because $b^n - 1 = b^n - 1n = (b - 1)(b^n - 1 + b^n - 2 + ... + b + 1) = (b - 1)b^n - 1 + ... + (b - 1)$, see also binomial numbers), the diminished radix complement of a number is found by complementing each digit with respect to $b - 1$ (that is, subtracting each digit in y from $b - 1$). Adding 1 to obtain the radix complement can be done separately, but is most often combined with the addition of x and the complement of y.

In the decimal numbering system, the radix complement is called the ten's complement and the diminished radix complement the nines' complement.

In binary, the radix complement is called the two's complement and the diminished radix complement the ones' complement. The naming of complements in other bases is similar.

1.3.1 Decimal example:

To subtract a decimal number y from another number x using the method of complements, the ten's complement of y (nines' complement plus 1) is added to x. Typically, the nines' complement

of y is first obtained by determining the complement of each digit. The complement of a decimal digit in the nines' complement system is the number that must be added to it to produce 9. The complement of 3 is 6; the complement of 7 is 2, and so on. Given a subtraction problem:

873 (x)
- 218 (y)

The nines' complement of y (218) is 781. In this case, because y is three digits long, this is the same as subtracting y from 999. (The number of 9's is equal to the number of digits of y.)

Next, the sum of x, the nines' complement of y, and 1 is taken:

873 (x)
+ 781 (complement of y)
+ 1 (to get the ten's complement of y)
=====
1655

The first "1" digit is then dropped, giving 655, the correct answer.

If the subtrahend has fewer digits than the minuend, leading zeros must be added which will become leading nines when the nines' complement is taken. For example:

```
  48032   (x)
-   391   (y)
```
becomes the sum:
```
  48032   (x)
+ 99608   (nines' complement of y)
+     1   (to get the ten's complement)
=======
 147641
```
Dropping the "1" gives the answer: 47641

1.3.2 Binary example:

The method of complements is especially useful in binary (radix 2) since the ones' complement is very easily obtained by inverting each bit (changing '0' to '1' and vice versa). And adding 1 to get the two's complement can be done by simulating a carry into the least significant bit. For example:

```
  01100100   (x, equals decimal 100)
- 00010110   (y, equals decimal 22)
```

becomes the sum:
```
  01100100   (x)
+ 11101001   (ones' complement of y)
+        1   (to get the two's complement)
==========
 101001110
```

Dropping the initial "1" gives the answer: 01001110 (equals decimal 78)

1.4 Signed fixed point numbers:

Up to this point we have considered only the representation of unsigned fixed point numbers. The situation is quite different in representing signed fixed point numbers. There are four different ways of representing signed numbers that are commonly used: sign-magnitude, one's complement,

two's complement, and excess notation. We will cover each in turn, using integers for our examples. The Table below shows for a 3-bit number how the various representations appear.

Decimal	Unsigned	Sign–Mag.	1's Comp.	2's Comp.	Excess 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	–	100	111	000	100
-1	–	101	110	111	011
-2	–	110	101	110	010
-3	–	111	100	101	001
-4	–	–	–	100	000

Table1. 3 bit number representation

1.4.1 Signed Magnitude Representation:

The signed magnitude (also referred to as sign and magnitude) representation is most familiar to us as the base 10 number system. A plus or minus sign to the left of a number indicates whether the number is positive or negative as in $+12_{10}$ or -12_{10}. In the binary signed magnitude representation, the leftmost bit is used for the sign, which takes on a value of 0 or 1 for '+' or '−', respectively. The remaining bits contain the absolute magnitude.

Consider representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format:

$(+12)_{10} = (00001100)_2$

$(-12)_{10} = (10001100)_2$

The negative number is formed by simply changing the sign bit in the positive number from 0 to 1. Notice that there are both positive and negative representations for zero: +0= 00000000 and -0= 10000000.

1.4.2 One's Complement Representation:

The one's complement operation is trivial to perform: convert all of the 1's in the number to 0's, and all of the 0's to 1's. See the fourth column in Table1 for examples. We can observe from the table that in the one's complement representation the leftmost bit is 0 for positive numbers and 1 for negative numbers, as it is for the signed magnitude representation. This negation, changing 1's to 0's and changing 0's to 1's is known as complementing the bits. Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, now using the one's complement representation:

$(+12)_{10} = (00001100)_2$

$(-12)_{10} = (11110011)_2$

Note again that there are representations for both +0 and −0, which are 00000000 and 11111111, respectively. As a result, there are only 28 − 1 = 255 different numbers that can be represented even though there are 28 different bit patterns.

The one's complement representation is not commonly used. This is at least partly due to the difficulty in making comparisons when there are two representations for 0. There is also additional complexity involved in adding numbers.

1.4.3 Two's Complement Representation:

The two's complement is formed in a way similar to forming the one's complement: complement all of the bits in the number, but then add 1, and if that addition results in a carry-out from the most significant bit of the number, discard the carry-out.

Examination of the fifth column of Table above shows that in the two's complement representation, the leftmost bit is again 0 for positive numbers and is 1 for negative numbers. However, this number format does not have the unfortunate characteristic of signed-magnitude and one's complement representations: it has only one representation for zero. To see that this is true, consider forming the negative of $(+0)_{10}$, which has the bit pattern: $(+0)_{10} = (00000000)_2$

Forming the one's complement of $(00000000)_2$ produces $(11111111)_2$ and adding 1 to it yields $(00000000)_2$, thus $(-0)_{10} = (00000000)_2$. The carry out of the leftmost position is discarded in two's complement addition (except when detecting an overflow condition). Since there is only one representation for 0, and since all bit patterns are valid, there are 28 = 256 different numbers that can be represented.

Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, this time using the two's complement representation. Starting with $(+12)_{10}= (00001100)_2$, complement, or negate the number, producing $(11110011)_2$.

Now add 1, producing $(11110100)_2$, and thus $(-12)_{10} = (11110100)_2$:

$(+12)_{10} = (00001100)_2$

$(-12)_{10} = (11110100)_2$

There is an equal number of positive and negative numbers provided zero is considered to be a positive number, which is reasonable because its sign bit is 0. The positive numbers start at 0, but the negative numbers start at −1, and so the magnitude of the most negative number is one greater than the magnitude of the most positive number. The positive number with the largest magnitude is +127, and the negative number with the largest magnitude is −128. There is thus no positive number that can be represented that corresponds to the negative of −128. If we try to form the two's complement negative of −128, then we will arrive at a negative number, as shown below:

$(-128)_{10} = (10000000)_2$
$(-128)_{10} = (01111111)$
$(-128)_{10} + (+0000001)_2$
$(-128)10$ ——————$)_2$
$(-128)_{10} = (10000000)_2$

The two's complement representation is the representation most commonly used in conventional computers.

1.5 Binary code:

Internally, digital computers operate on binary numbers. When interfacing to humans, digital processors, e.g. pocket calculators, communication is decimal-based. Input is done in decimal then

converted to binary for internal processing. For output, the result has to be converted from its internal binary representation to a decimal form. Digital system represents and manipulates not only binary number but also many other discrete elements of information.

1.5.1 Binary coded Decimal:

In computing and electronic systems, binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding. It occupies more space than a pure binary representation. In BCD, a digit is usually represented by four bits which, in general, represent the values/digits/characters 0-9

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

Decimal:	0	1	2	3	4	5	6	7	8	9
BCD:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Thus, the BCD encoding for the number 127 would be:

0001 0010 0111

The position weights of the BCD code are 8, 4, 2, 1. Other codes (shown in the table) use position weights of 8, 4, -2, -1 and 2, 4, 2, 1.

An example of a non-weighted code is the excess-3 code where digit codes are obtained from their binary equivalent after adding 3. Thus the code of a decimal 0 is 0011, that of 6 is 1001, etc.

It is very important to understand the difference between the conversion of a decimal number to binary and the binary coding of a decimal number.

Decimal Digit	**8 4 2 1 Code**	**8 4 -2 -1 code**	**2 4 2 1 code**	**Excess-3 code**
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 1 1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 1 0	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 0 1	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 0 0	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 1 1	1 0 1 1	1 0 0 0
6	0 1 1 0	1 0 1 0	1 1 0 0	1 0 0 1
7	0 1 1 1	1 0 0 1	1 1 0 1	1 0 1 0
8	1 0 0 0	1 0 0 0	1 1 1 0	1 0 1 1
9	1 0 0 1	1 1 1 1	1 1 1 1	1 1 0 0

In each case, the final result is a series of bits. The bits obtained from conversion are binary digit. Bits obtained from coding are a combination of 1's and 0's arranged according to the rule of the code used. E.g. the binary conversion of 13 is 1101; the BCD coding of 13 is 00010011.

1.6 Gray code:

The Gray code consists of 16 4-bit code words to represent the decimal Numbers 0 to 15. For Gray code, successive code words differ by only one bit from one to the next

Gray Code	Decimal Equivalent
0 0 0 0	0
0 0 0 1	1
0 0 1 1	2
0 0 1 0	3
0 1 1 0	4
0 1 1 1	5
0 1 0 1	6
0 1 0 0	7
1 1 0 0	8
1 1 0 1	9
1 1 1 1	10
1 1 1 0	11
1 0 1 0	12
1 0 1 1	13
1 0 0 1	14
1 0 0 0	15