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 = 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

UNIT – II

CONCEPT OF BOOLEAN ALGEBRA

2.1 Introduction:

Binary logic deals with variables that assume discrete values and with operators that assume logical meaning. While each logical element or condition must always have a logic value of either "0" or "1", we also need to have ways to combine different logical signals or conditions to provide a logical result.

For example, consider the logical statement: "If I move the switch on the wall up, the light will turn on." At first glance, this seems to be a correct statement. However, if we look at a few other factors, we realize that there's more to it than this. In this example, a more complete statement would be: "If I move the switch on the wall up and the light bulb is good and the power is on, the light will turn on."

If we look at these two statements as logical expressions and use logical terminology, we can reduce the first statement to:

Light = Switch

This means nothing more than that the light will follow the action of the switch, so that when the switch is up/on/true/1 the light will also be on/true/1. Conversely, if the switch is down/off/false/0 the light will also be off/false/0.

Looking at the second version of the statement, we have a slightly more complex expression:

Light = Switch and Bulb and Power

When we deal with logical circuits (as in computers), we not only need to deal with logical functions; we also need some special symbols to denote these functions in a logical diagram. There are three fundamental logical operations, from which all other functions, no matter how complex, can be derived. These functions are named and, or, and not. Each of these has a specific symbol and a clearly-defined behavior.

AND:
The AND operation is represented by a dot (.) or by the absence of an operator. E.g. $x.y=z$ $xy=z$ are all read as x AND y=z. the logical operation AND is interpreted to mean that z=1 if and only if x=1 and y=1 otherwise z=0

OR:
The operation is represented by a + sign for example, $x+y=z$ is interpreted as x OR y=z meaning that z=1 if x=1 or y=1 or if both x=1 and y=1. If both x and y are 0, then z=0

NOT:
This operation is represented by a bar or a prime. For example $x'= x=z$ is interpreted as NOT x =z meaning that z is what x is not.

It should be noted that although the AND and the OR operation have some similarity with the multiplication and addition respectively in binary arithmetic, however one should note that an arithmetic variable may consist of many digits. A binary logic variable is always 0 or 1.

E.g. in binary arithmetic, 1+1=10 while in binary logic 1+1=1

2.2 Basic Gates:

The basic building blocks of a computer are called logical gates or just gates. Gates are basic circuits that have at least one (and usually more) input and exactly one output. Input and output values are the logical values true and false. In computer architecture it is common to use 0 for false and 1 for true. Gates have no memory. The value of the output depends only on the current value of the inputs. A useful way of describing the relationship between the inputs of gates and their output is the truth table. In a truth table, the value of each output is tabulated for every possible combination of the input values.

We usually consider three basic kinds of gates, and-gates, or-gates, and not-gates (or inverters).

2.2.1 The AND Gate:

The AND gate implements the AND function. With the gate shown to the left, both inputs must have logic 1 signals applied to them in order for the output to be logic 1. With either input at logic 0, the output will be held to logic 0.

The truth table for an and-gate with two inputs looks like this:

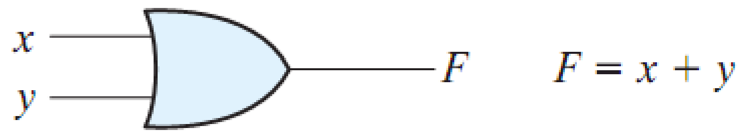
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

There is no limit to the number of inputs that may be applied to an AND function, so there is no functional limit to the number of inputs an AND gate may have. However, for practical reasons, commercial AND gates are most commonly manufactured with 2, 3, or 4 inputs. A standard Integrated Circuit (IC) package contains 14 or 16 pins, for practical size and handling. A standard 14-pin package can contain four 2-input gates, three 3-input gates, or two 4input gates, and still have room for two pins for power supply connections.

2.2.2 The OR Gate:

The OR gate is sort of the reverse of the AND gate. The OR function, like its verbal counterpart, allows the output to be true (logic 1) if any one or more of its inputs are true. Verbally, we might say, "If it is raining OR if I turn on the sprinkler, the lawn will be wet." Note that the lawn will still be wet if the sprinkler is on and it is also raining. This is correctly reflected by the basic OR function.

In symbols, the OR function is designated with a plus sign (+). In logical diagrams, the symbol below designates the OR gate.



The truth table for an or-gate with two inputs looks like this:

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

As with the AND function, the OR function can have any number of inputs. However, practical commercial OR gates are mostly limited to 2, 3, and 4 inputs, as with AND gates.

2.2.3 The NOT Gate, or Inverter:

The inverter is a little different from AND and OR gates in that it always has exactly one input as well as one output. Whatever logical state is applied to the input, the opposite state will appear at the output.

The truth table for an inverter looks like this:

x	F
0	1
1	0

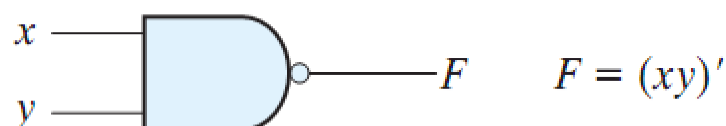
The NOT function, as it is called, is necessary in many applications and highly useful in others. A practical verbal application might be:

The door is NOT locked = you may enter

In the inverter symbol, the triangle actually denotes only an amplifier, which in digital terms means that it "cleans up" the signal but does not change its logical sense. It is the circle at the output which denotes the logical inversion. The circle could have been placed at the input instead, and the logical meaning would still be the same.

2.2.4 The NAND Gate:

The NAND-gate is an and-gate with an inverter on the output. So instead of drawing several gates, we draw a single and-gate with a little ring on the output like this:



The nand-gate, like the and-gate can take an arbitrary number of inputs.

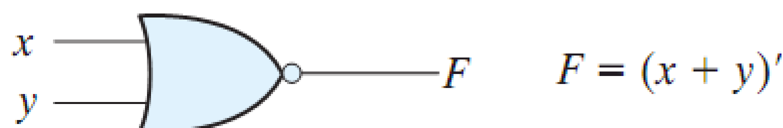
The truth table for the nand-gate is like the one for the and-gate, except that all output values have been inverted:

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

The truth table clearly shows that the NAND operation is the complement of the AND gate.

2.2.5 The NOR-Gate:

The nor-gate is an or-gate with an inverter on the output. So instead of drawing several gates, we draw a single and-gate with a little ring on the output like this:

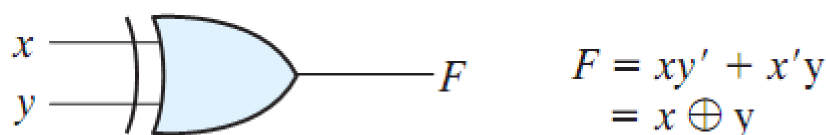


The nor-gate, like the or-gate can take an arbitrary number of inputs. The truth table for the nor-gate is like the one for the or-gate, except that all output values have been inverted:

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

2.2.6 The Exclusive-OR-Gate:

The exclusive-or-gate is similar to an or-gate. It can have an arbitrary number of inputs, and its output value is 1 if and only if exactly one input is 1 (and thus the others 0). Otherwise, the output is 0. We draw an exclusive-or-gate like this:

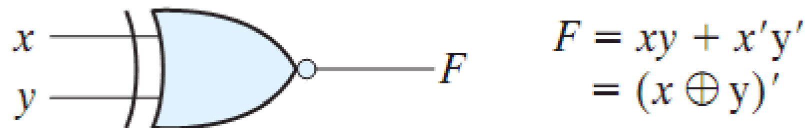


The truth table for an exclusive-or-gate with two inputs looks like this:

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

2.2.7 The Exclusive-NOR-Gate:

The exclusive-Nor-gate is similar to an N or-gate. It can have an arbitrary number of inputs, and its output value is 1 if and only if the two inputs are of the same values (1 and 1 or 0 and 0). Otherwise, the output is 0. We draw an exclusive-Nor-gate like this:



The truth table for an exclusive-nor-gate with two inputs looks like this:

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

2.3 Boolean algebra:

One of the primary requirements when dealing with digital circuits is to find ways to make them as simple as possible. This constantly requires that complex logical expressions be reduced to simpler expressions that nevertheless produce the same results under all possible conditions. The simpler expression can then be implemented with a smaller, simpler circuit, which in turn saves the price of the unnecessary gates, reduces the number of gates needed, and reduces the power and the amount of space required by those gates.

One tool to reduce logical expressions is the mathematics of logical expressions, introduced by George Boole in 1854 and known today as Boolean algebra. The rules of Boolean algebra are simple and straight-forward, and can be applied to any logical expression. The resulting reduced expression can then be readily tested with a Truth Table, to verify that the reduction was valid.

Boolean algebra is an algebraic structure defined on a set of elements B, together with two binary operators (+, .) provided the following postulates are satisfied.

1. Closure with respect to operator '+' and Closure with respect to operator '.'

2(a) An identity element with respect to + designated by 0: X+0= 0+X=X

2(b) An identity element with respect to .designated by 1: X.1= 1.X=X

3(a) Commutative with respect to +: X=Y=Y+X

3(b) Commutative with respect to . : X.Y=Y.X

4(a) .distributive over +: X . (Y+Z)=X.Y+X.Z

4(b) + distributive Over. X+ (Y.Z) =(X+Y). (X+Z)

5. For every element x belonging to B, there exist an element x' or x called the complement of x such that x . x'=0 and x + x'=1

6. There exists at least two elements x, y belonging to B such that x≠y

The two valued Boolean algebra is defined on a set B= {0, 1} with two binary operators '+' and '.'.

X	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

x	x^1
0	1
1	0

Closure: from the tables, the result of each operation is either 0 or 1 and 1, 0 belongs to B

Identity: From the truth table we see that 0 is the identity element for '**+**' and 1 is the identity element for '**.**'.

Commutative law is obvious from the symmetry of binary operators table.

Distributive Law. (y + z) = x . y + x .z

x	y	z	y+z	x .(y+z)	x .y	x .z	x .y+x.z
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Distributive law of '**+**' over '**.**' can be shown as in the truth table above.

From the *complement* table we can see that x+ x'=1 i.e. 1+0=1 and x. x'=0 i.e. 1.0=0

2.3.1 Principle of duality of Boolean algebra:

The principle of duality state that every algebraic expression which can be deduced from the postulates of Boolean algebra remains valid if the operators and the identity elements are interchanged. This mean that the dual of an expression is obtained changing every AND (.) to OR (+), every OR (+) to AND (.) and all 1's to 0's and vice-versa.

2.3.2 Laws of Boolean algebra:

Postulate 2:

(a) 0 + A = A (b) 1. A = A

Postulate 5:

(a) A + A' =1 (b) A . A'=0

Theorem1: Identity Law

(a) A + A = A (b) A . A = A

Theorem2

(a) $1 + A = 1$ (b) $0 . A = 0$

Theorem3: involution

$A'' = A$

Postulate 3: Commutative Law

(a) $A + B = B + A$ (b) $A . B = B . A$

Theorem4: Associate Law

(a) $(A + B) + C = A + (B + C)$ (b) $(A B) C = A (B C)$

Postulate4: Distributive Law

(a) $A (B + C) = A B + A C$ (b) $A + (B C) = (A + B) (A + C)$

Theorem5: De Morgan's Theorem

(a) $(A+B)' = A'B'$ (b) $(AB)' = A' + B'$

Theorem6: Absorption

(a) $A + A B = A$ (b) $A (A + B) = A$

Prove Theorem 1: (a)

$X+X=X$

$x+x=(X+X).1$ by postulate 2b

$=(x+x) (x+x')$ 5a

$=x+xx'$ 4b

$=x+0$ 5b

$=x$ 2a

Prove Theorem 1: (b)

$X.X=X$

$xx= (X.X) +0$ by postulate 2a

$=x.x+x.x'$ 5b

$=x(x+x')$ 4a

$=x.1$ 5a

$=x$ 2b

Prove Theorem 2: (a)

$X+1=X$

$x+1=1.(X+1)$ by postulate 2b

$(x+x')=(x+1)$ 5a

$=x+x'.1$ 4b

$=x+x'$ 2b

$=1$ 5a

Prove Theorem 2: (b)

$X.0=0$

$x.0=0+(X.0)$ by postulate 2a

$(x.x')=(x.0)$ 5b

$=x.x'+0$ 4a

$=x.x'$ 2a

$=0$ 5b

Prove Theorem 6: (a)

$X+xy=X$

$x+xy=x.1+xy$ by postulate 2b

$=x(1+y)$ 4b

$=x(y+1)$ 3a

$=x.1$ 2b

$=x$ 2b

Prove Theorem 6: (b)

$X(x+y)=X$

$X(x+y)=(x+0).(x+y)$ by postulate 2a

$=x+0.y$ 4a

$=x+0$ 2a

$=x$ 2a

Using the laws given above, complicated expressions can be simplified.

2.4 Boolean functions:

Operations of binary variables can be described by mean of appropriate mathematical function called Boolean function. A Boolean function defines a mapping from a set of binary input values into a set of output values. A Boolean function is formed with binary variables, the binary operators AND and OR and the unary operator NOT.

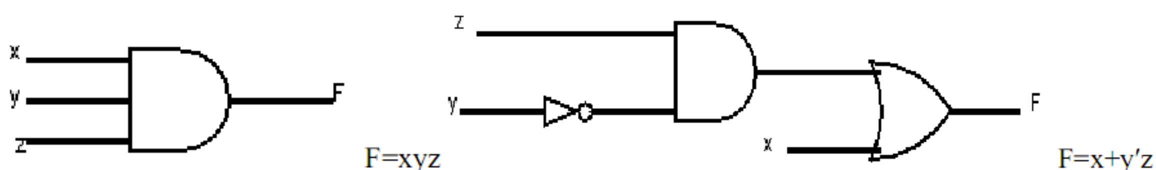
For example, a Boolean function $f(x_1,x_2,x_3,\ldots,x_n)$ =y defines a mapping from an arbitrary combination of binary input values $(x_1,x_2,x_3,\ldots,x_n)$ into a binary value y. a binary function with n input variable can operate on 2n distinct values. Any such function can be described by using a truth table consisting of 2n rows and n columns. The content of this table are the values produced by that function when applied to all the possible combination of the n input variable.

x	*y*	*F*
0	0	0
0	1	0
1	0	0
1	1	1

The function f, representing x.y, that is f(x, y) =xy. Which mean that f=1 if x=1 and y=1 and f=0 otherwise.

For each rows of the table, there is a value of the function equal to 1 or 0. The function f is equal to the sum of all rows that gives a value of 1.

A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gate. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate and each term is implemented with a logic gate. e.g.



2.5 Complement of a function:

The complement of a function F is F' and is obtained from an interchange of 0's to 1's and 1's to 0's in the value of F. The complement of a function may be derived algebraically trough De Morgan's theorem

(A+B+C+….)'= A'B'C'….

(ABC….)'= A'+ B'+C'……

The generalized form of de Morgan's theorem state that the complement of function is obtained by interchanging AND and OR and complementing each literal.

F = X'YZ'+X'Y'Z'

F' = (X'YZ'+X'Y'Z')'

= (X'YZ')'. (X'Y'Z')'

= (X''+Y'+Z'') (X''+Y''+Z'')

=(X+Y'+Z) (X+Y+Z)

2.6 Canonical form (Minterms and Maxterms):

A binary variable may appear either in its normal form or in its complement form. Consider two binary variables x and y combined with AND operation. Since each variable may appears in either form there are four possible combinations: x'y', x'y, xy', xy. Each of the term represent one distinct area in the Venn diagram and is called minterm or a standard product. With n variable, 2n minterms can be formed.

In a similar fashion, n variables forming an OR term provide 2n possible combinations called maxterms or standard sum. Each maxterm is obtained from an OR term of the n variables, with each variable being primed if the corresponding bit is 1 and un-primed if the corresponding bit is 0. Note that each maxterm is the complement of its corresponding minterm and vice versa.

X	Y	Z	MIN TERM	MAX TERM
0	0	0	$X^1Y^1Z^1$	X+Y+Z
0	0	1	X^1Y^1Z	$X+Y+Z^1$
0	1	0	X^1YZ^1	$X+Y^1+Z$
0	1	1	X^1YZ	$X+Y^1+Z^1$
1	0	0	XY^1Z^1	X^1+Y+Z
1	0	1	XY^1Z	X^1+Y+Z^1
1	1	0	XYZ^1	X^1+Y^1+Z
1	1	1	XYZ	$X^1+Y^1+Z^1$

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of variable that produce a 1 and taken the OR of those terms.

Similarly, the same function can be obtained by forming the maxterm for each combination of variable that produces 0 and then taken the AND of those term.

It is sometime convenient to express the boolean function when it is in sum of minterms, in the following notation:

F(X, Y, Z) =∑ (1, 4, 5, 6, 7)

The summation symbol ∑ stands for the ORing of the terms; the numbers following it are the minterms of the function. The letters in the parenthesis following F form list of the variables in the order taken when the minterm is converted to an AND term.

So, F(X,Y,Z) =∑ (1,4,5,6,7) = X'Y'Z+XY'Z'+XY'Z+XYZ'+XYZ

Sometime it is convenient to express a Boolean function in its sum of minterm. If it is not in that case, the expression is expanded into the sum of AND term and if there is any missing variable, it is ANDed with an expression such as x+x' where x is one of the missing variable.

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This can be done by using distributive law x+xz=(x+y) (x+z). Then if there is any missing variable, say x in each OR term is ORded with xx'.

Example 2.1: Represent F=xy+x'z as a product of maxterm

F = (xy +x') (xy+z)

= (x+x') (y+x') (x+z) (y+z)

= (y+x') (x+z) (y+z)

Adding missing variable in each term

(y+x')= x'+y+zz' = (x'+y+z) (x'+y+z')

(x+z)= x+z+yy' =(x+y+z) (x+y'+z)

(y+z)= y+z+xx' =(x+y+z) (x'+y+z)

F= (x+y+z) (x+y'+z) (x'+y+z) (x'+y+z')

A convenient way to express this function is as follows:

$F(x, y, z) = \prod (0, 2, 4, 5)$

2.7 Standard form:

Another way to express a Boolean function is in standard form. Here the term that form the function may contains one, two or nay number of literals. There are two types of standard form. The sum of product and the product of sum.

The sum of product (SOP) is a Boolean expression containing AND terms called product term of one or more literals each. The sum denotes the ORing of these terms

E.g. F=x+xy'+x'yz

The product of sum (POS) is a Boolean expression containing OR terms called SUM terms. Each term may have any number of literals. The product denotes the ANDing of these terms

E.g. F= x(x+y') (x'+y+z)

A Boolean function may also be expressed in a non-standard form. In that case, distributive law can be used to remove the parenthesis

F = (xy+zw) (x'y'+z'w')

= xy (x'y'+z'w') +zw (x'y'+z'w')

= xyx'y +xyz'w' +zwx'y' +zwz'w'

= xyz'w'+zwx'y'

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only method is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar.

2.7.1 Describing existing circuits using Logic expressions:

To define what a combinatorial circuit does, we can use a logic expression or an expression for short. Such an expression uses the two constants 0 and 1, variables such as x, y, and z (sometimes with suffixes) as names of inputs and outputs, and the operators +, .and a horizontal bar or a prime

(which stands for not). As usual, multiplication is considered to have higher priority than addition. Parentheses are used to modify the priority.

If Boolean functions in either Sum of Product or Product of Sum forms can be implemented using 2-Level implementations.

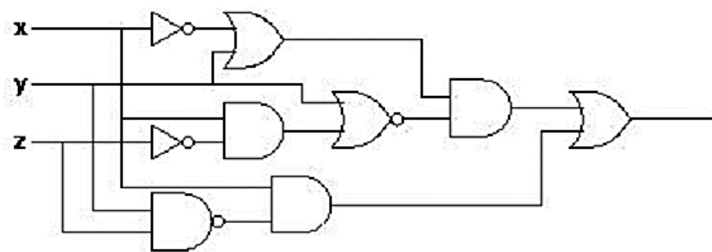
For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

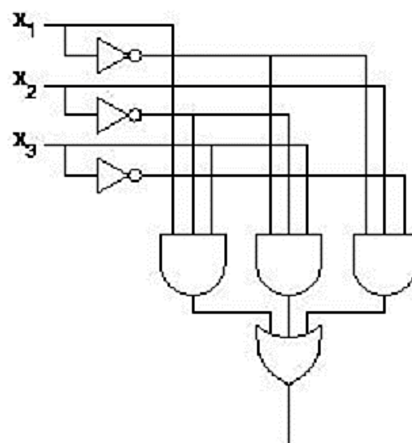
Examples2.3: $F = (X'+Y)\ (Y+XZ')'+X\ (YZ)'$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



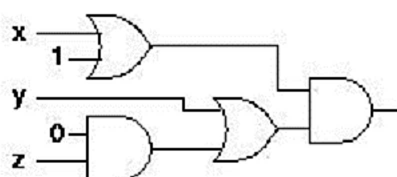
Example 2.4: $F = X_1X_2'X_3+X_1'X_2'X_2+X_1'X_2X'_3$

The equation is in sum of product. The implementation is in 2-Levels. AND gates form the first level and a single OR gate the second level.



Example 2.5: $F = (X+1)\ (Y+0Z)$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



UNIT- III

GATE LEVEL MINIMIZATION

3.1 Introduction:

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, it can appear in many different forms when expressed algebraically.

3.2 Simplification through algebraic manipulation:

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation as stated above. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only methods is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar

E.g.3.1: Simplify $x+x'y$

$x+x'y = (x+x')\ (x+y)$

$= x+y$

E.g.3.2: Simplify $x'y'z+x'yz+xy'$

$x'y'z+x'yz+xy' = x'z(y+y') +xy'$

$= x'z+xy'$

E.g.3.3: Simplify $xy +x'z+yz$

$xy +x'z+yz= xy +x'z+yz(x+x')$

$= xy +x'z+yzx+yzx'$

$= xy\ (1+z) +x'z\ (1+y)$

$=xy+x'z$

3.3 Karnaugh map:

The Karnaugh map also known as Veitch diagram or simply as K map is a two dimensional form of the truth table, drawn in such a way that the simplification of Boolean expression can be immediately be seen from the location of 1's in the map. The map is a diagram made up of squares, each square represent one minterm. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.

3.3.1 Two variable K-Map:

A two variable Boolean function can be represented as follows:

m_0	m_1
m_2	m_3

(a)

$x \backslash y$	0	1
0	m_0 $x'y'$	m_1 $x'y$
1	m_2 xy'	m_3 xy

(b)

3.3.2 Three variable K-Map:

A three variable function can be represented as follows:

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

$x \backslash yz$	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

3.3.3 Four variable K-Map:

A four variable Boolean function can be represented in the map below.

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

$wx \backslash yz$	00	01	11	10
00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$
01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$
11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$
10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$

To simplify a Boolean function using Karnaugh map, the first step is to plot all ones in the function truth table on the map. The next step is to combine adjacent 1's into a group of one, two, four, eight, and sixteen. The group of minterm should be as large as possible. A single group of four minterm yields a simpler expression than two groups of two minterms.

In a four variable Karnaugh map,

1 variable product term is obtained if 8 adjacent squares are covered.

2 variable product term is obtained if 4 adjacent squares are covered.

3 variable product term is obtained if 2 adjacent squares are covered.

1 variable product term is obtained if 1 square is covered.

A square having a 1 may belong to more than one term in the sum of product expression.

The final stage is reached when each of the group of minterms are ORed together to form the simplified sum of product expression.

The Karnaugh map is not a square or rectangle as it may appear in the diagram. The top edge is adjacent to the bottom edge and the left hand edge adjacent to the right hand edge. Consequent, two squares in Karnaugh map are said to be adjacent if they differ by only one variable.

3.4 Implicants:

In Boolean logic, an implicant is a "covering" (sum term or product term) of one or more minterms in a sum of products (or maxterms in a product of sums) of a boolean function. Formally, a product term P in a sum of products is an implicant of the Boolean function F if P implies F. More precisely:

P implies F (and thus is an implicant of F) if F also takes the value 1 whenever P equals 1.

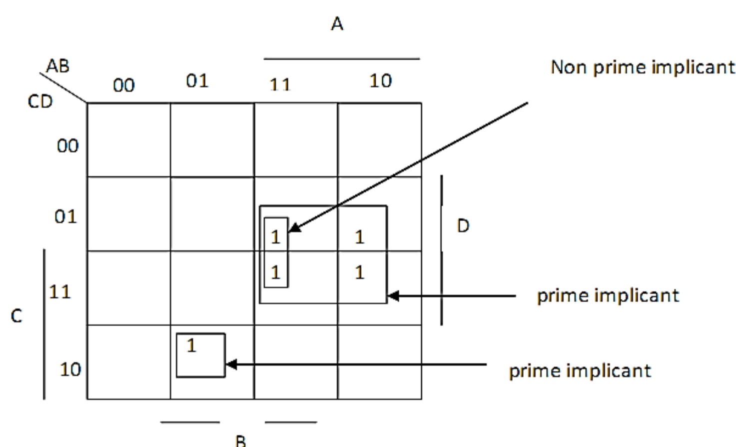
Where

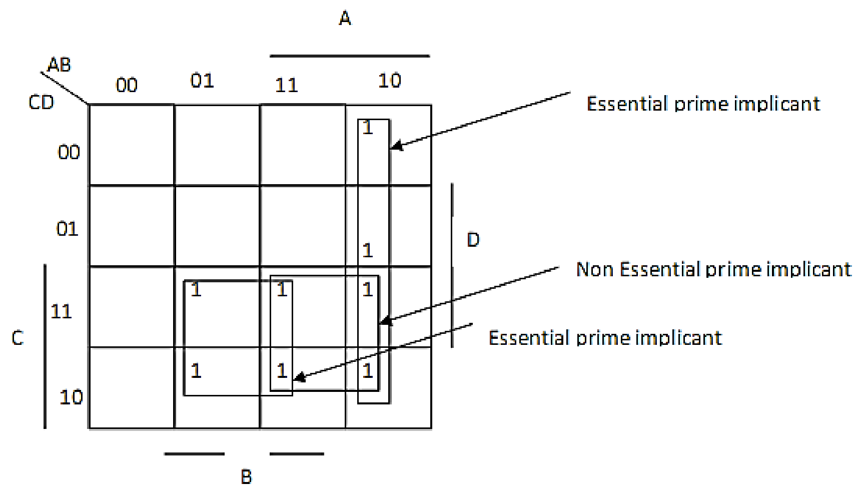
- F is a Boolean of n variables.
- P is a product term

This means that P <= F with respect to the natural ordering of the Boolean space. For instance, the function $f(x,y,z,w) = xy + yz + w$ is implied by xy, by xyz, by xyzw, by w and many others; these are the implicants of f.

3.4.1 Prime and Essential prime implicants:

A prime implicant of a function is an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant. W.V. Quine defined a prime implicant of F to be an implicant that is minimal - that is, if the removal of any literal from P results in a non-implicant for F. Essential prime implicants are prime implicants that cover an output of the function that no combination of other prime implicants is able to cover.





In simplifying a Boolean function using Karnaugh map, non-essential prime implicant are not needed.

3.5 Minimization of Boolean expressions using Karnaugh maps:

Example3.4: Given the following truth table for the majority function.

a	b	c	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The Boolean algebraic expression is

M = a'bc + ab'c + abc' + abc.

The minimization using algebraic manipulation can be done as follows.

M = a'bc + abc + ab'c + abc + abc' + abc

= (a' + a) bc + a (b' + b) c + ab (c' + c)

= bc + ac + ab

The abc term was replicated and combined with the other terms.

To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 101.

c \ ab	00	01	11	10
0			1	
1		1	1	1

The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 110 (a = 1, b = 1, c = 0).

The minimization is done by drawing circles around sets of adjacent 1s. Adjacency is horizontal, vertical, or both. The circles must always contain 2^n 1s where n is an integer.

c \ ab	00	01	11	10
0			1	
1		1	1	1

We have circled two 1s. The fact that the circle spans the two possible values of a (0 and 1) means that the a term is eliminated from the Boolean expression corresponding to this circle.

Now we have drawn circles around all the 1s. Thus the expression reduces to bc + ac + ab as we saw before.

What is happening? What does adjacency and grouping the 1s together have to do with minimization? Notice that the 1 at position 111 was used by all 3 circles. This 1 corresponds to the abc term that was replicated in the original algebraic minimization. Adjacency of 2 1s means that the terms corresponding to those 1s differ in one variable only. In one case that variable is negated and in the other it is not.

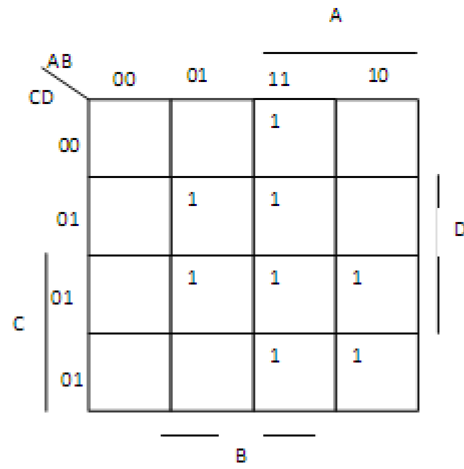
The map is easier than algebraic minimization because we just have to recognize patterns of 1s in the map instead of using the algebraic manipulations. Adjacency also applies to the edges of the map.

For 4 Boolean variables, the Karnaugh map is drawn as shown below.

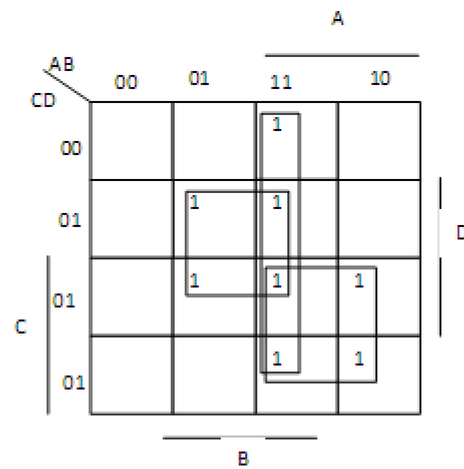
Example 3.5: The following corresponds to the Boolean expression

Q = A'BC'D + A'BCD + ABC'D' + ABC'D + ABCD + ABCD' + AB'CD + AB'CD'

RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s.



Grouping the 1s together results in the following.

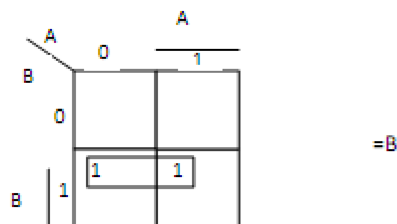


The expression for the groupings above is

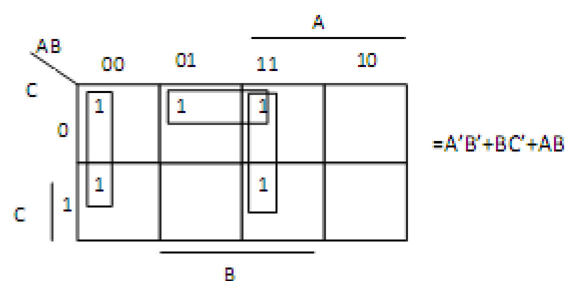
Q = BD + AC + AB

This expression requires 3 2-input AND gates and 1 3-input OR gate.

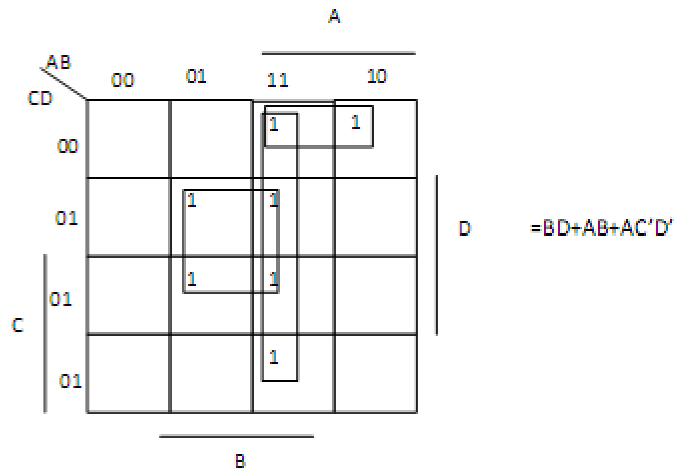
Example 3.6: F=A'B+AB



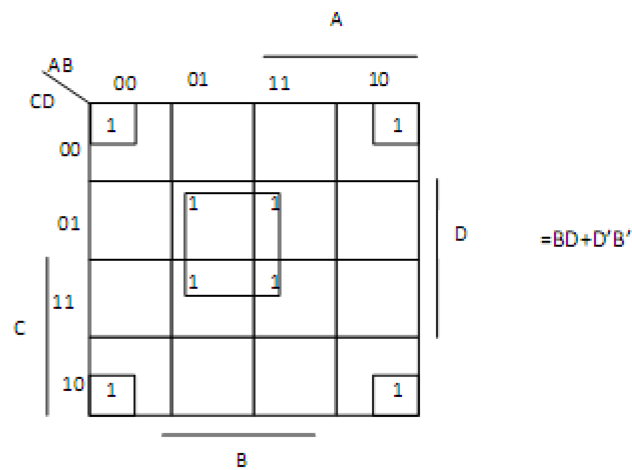
Example 3.7: F=A'B'C'+A'B'C+A'BC'+ABC'+ABC



Example 3.8: F=AB+A'BC'D+A'BCD+AB'C'D'



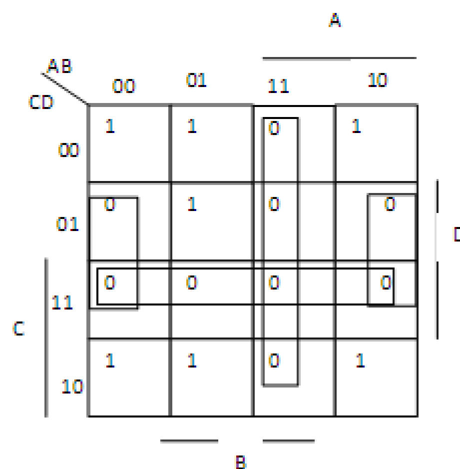
Example 3.9: F=A'B'C'D'+AB'C'D'+A'BC'D+ABC'D+A'BCD+ABCD



3.6 Obtaining a Simplified product of sum using Karnaugh map:

The simplification of the product of sum follows the same rule as the product of sum. However, adjacent cells to be combined are the cells containing 0. In this approach, the obtained simplified function is F'. Since F is represented by the square marked with 1, the function F can be obtained in product of sum by applying de Morgan's rule on F'.

F=A'B'C'D'+A'BC'D'+AB'C'D'+A'BC'D+A'B'CD'+A'BCD'+AB'CD'



The obtained simplified F'=AB+CD+BD'. Since F''=F, By applying de Morgan's rule to F', we obtain F'' = (AB+CD+BD') '

= (A'+B') (C'+D') (B'+D) which is he simplified F in product of sum.

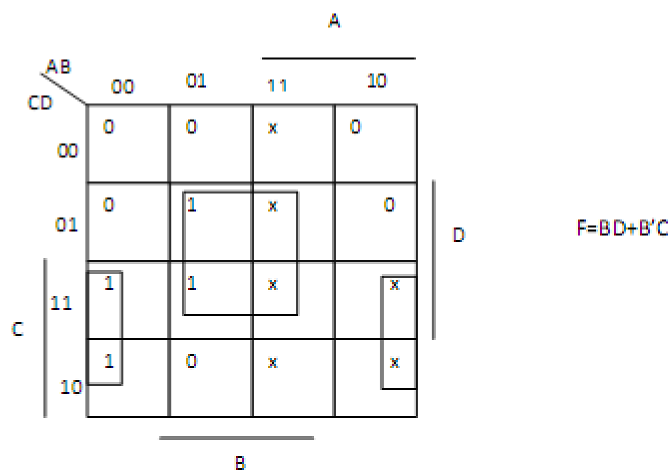
3.7 Don't Care conditions:

Sometimes we do not care whether a 1 or 0 occurs for a certain set of inputs. It may be that those inputs will never occur so it makes no difference what the output is. For example, we might have a BCD (binary coded decimal) code which consists of 4 bits to encode the digits 0 (0000) through 9 (1001). The remaining codes (1010 through 1111) are not used. If we had a truth table for the prime numbers 0 through 9, it would be

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

F=A'B'CD'+A'B'CD+A'BC'D+A'BCD

The 'X' in the above stand for "don't care", we don't care whether a 1 or 0 is the value for that combination of inputs because (in this case) the inputs will never occur.



3.8 Implementing logical circuit using NAND and NOR gates only:

In addition to AND, OR, and NOT gates, other logic gates like NAND and NOR are also used in the design of digital circuits. The NAND gate represents the complement of the AND operation. Its name is an abbreviation of NOT AND. The graphic symbol for the NAND gate

consists of an AND symbol with a bubble on the output, denoting that a complement operation is performed on the output of the AND gate as shown earlier.

The NOR gate represents the complement of the OR operation. Its name is an abbreviation of NOT OR. The graphic symbol for the NOR gate consists of an OR symbol with a bubble on the output, denoting that a complement operation is performed on the output of the OR gate as shown earlier.

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families. In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around.

Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around.

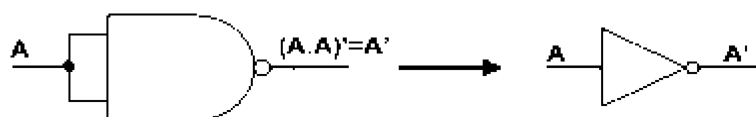
3.8.1 NAND Gate is a Universal Gate:

To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates. A universal gate is a gate which can implement any Boolean function without need to use any other gate type.

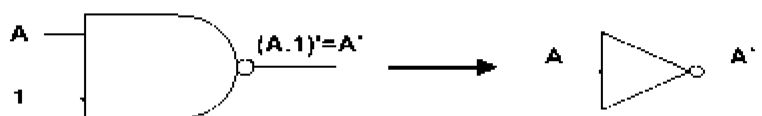
Implementing an Inverter Using only NAND Gate:

The figure shows two ways in which a NAND gate can be used as an inverter (NOT gate).

1. All NAND input pins connect to the input signal A gives an output A'.

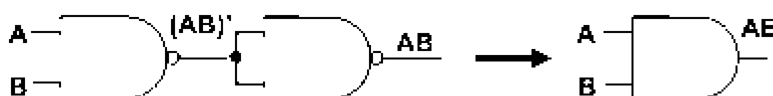


2. One NAND input pin is connected to the input signal A while all other input pins are connected to logic 1. The output will be A'.



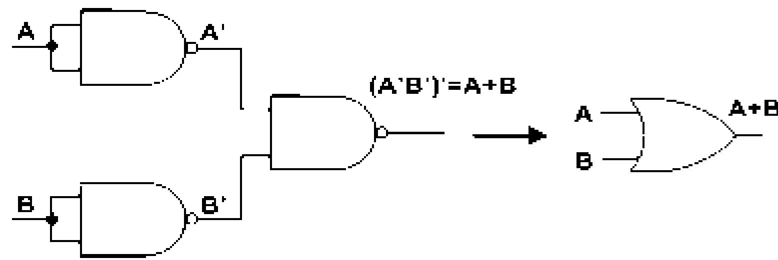
Implementing AND Using only NAND Gates:

An AND gate can be replaced by NAND gates as shown in the figure (The AND is replaced by a NAND gate with its output complemented by a NAND gate inverter).



Implementing OR Using only NAND Gates:

An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).



Thus, the NAND gate is a universal gate since it can implement the AND, OR and NOT functions.

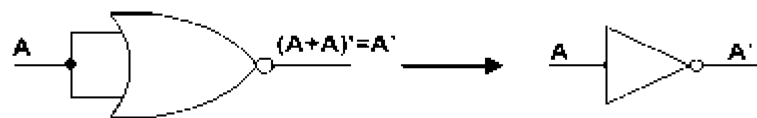
3.8.2 NOR Gate is a Universal Gate:

To prove that any Boolean function can be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

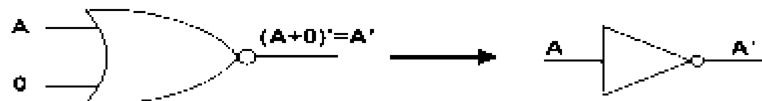
Implementing an Inverter Using only NOR Gate:

The figure shows two ways in which a NOR gate can be used as an inverter (NOT gate).

1. All NOR input pins connect to the input signal A gives an output A'.

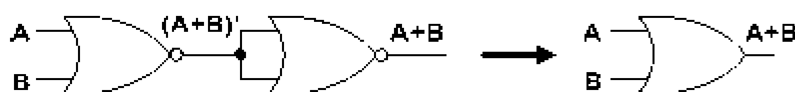


2. One NOR input pin is connected to the input signal A while all other input pins are connected to logic 0. The output will be A'.



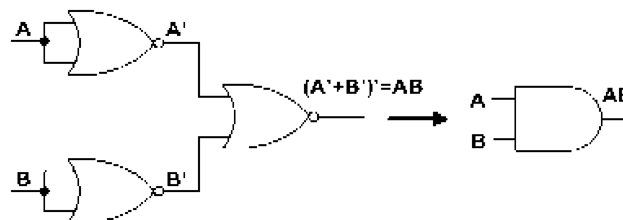
Implementing OR Using only NOR Gates:

An OR gate can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)



Implementing AND Using only NOR Gates:

An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters).

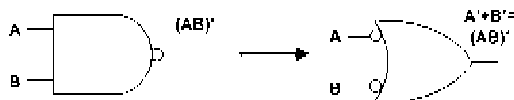


Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.

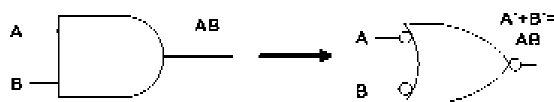
3.8.3 Equivalent Gates:

The shown figure summarizes important cases of gate equivalence. Note that bubbles indicate a complement operation (inverter).

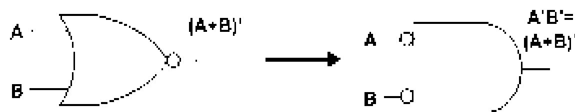
A NAND gate is equivalent to an inverted-input OR gate.



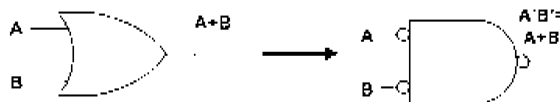
An AND gate is equivalent to an inverted-input NOR gate.



A NOR gate is equivalent to an inverted-input AND gate.



An OR gate is equivalent to an inverted-input NAND gate.



Two NOT gates in series are same as a buffer because they cancel each other as A''=A.

3.9 Two-Level Implementations:

We have seen before that Boolean functions in either SOP or POS forms can be implemented using 2-Level implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

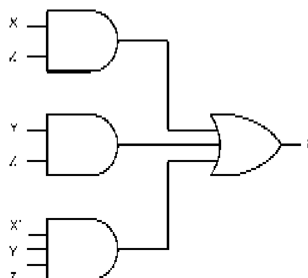
Note that using inverters to complement input variables is not counted as a level.

To implement a function using NAND gates only, it must first be simplified to a sum of product and to implement a function using NOR gates only, it must first be simplified to a product of sum. We will show that SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only NOR gates through examples.

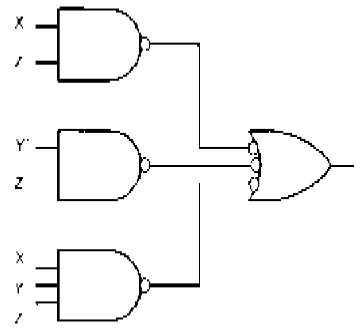
Example 1: Implement the following SOP function using NAND gate only

F = XZ + Y'Z + X'YZ

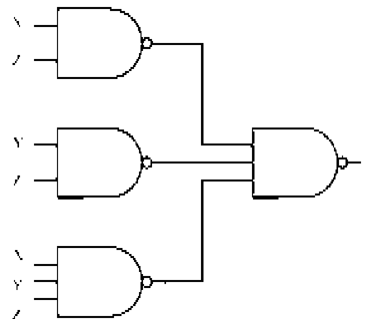
Being an SOP expression, it is implemented in 2-levels as shown in the figure.



Introducing two successive inverters at the inputs of the OR gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.



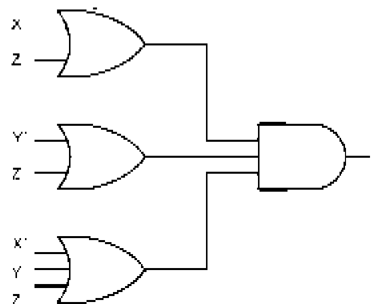
By associating one of the inverters with the output of the first level AND gate and the other with the input of the OR gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NAND gates as shown in Figure.



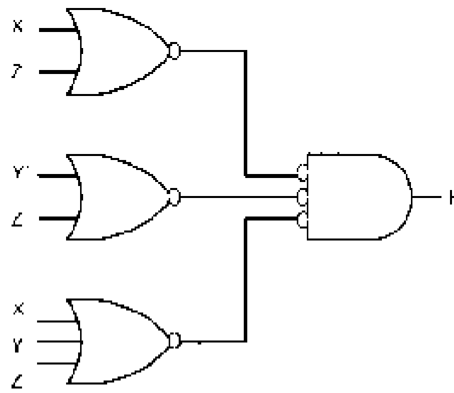
Example 2: Implement the following POS function using NOR gates only

F = (X+Z) (Y'+Z) (X'+Y+Z)

Being a POS expression, it is implemented in 2-levels as shown in the figure.



Introducing two successive inverters at the inputs of the AND gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.



By associating one of the inverters with the output of the first level OR gates and the other with the input of the AND gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NOR gates as shown in Figure.