Lecture One

Digital Logic Design

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.

Numeric Systems and Conversion

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. 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 Numeric Systems

1. 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)
```

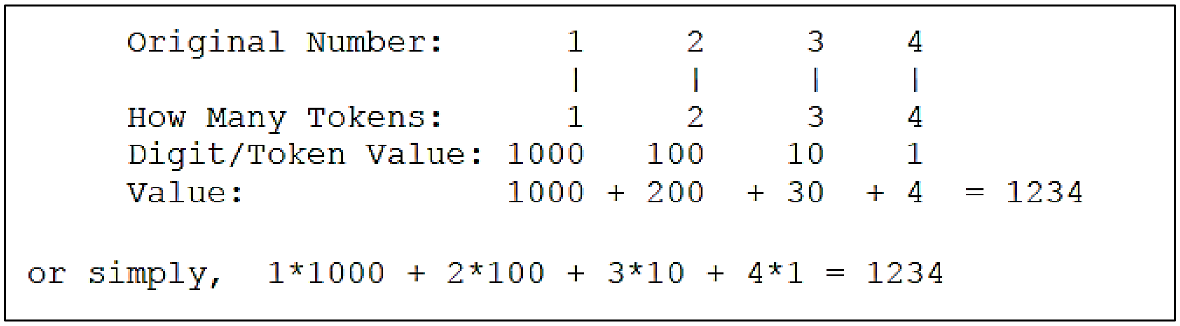
Let's express the same decimal number 1341 in hexadecimal notation.

```
          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)
```

```
                                      (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)
```

2. 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

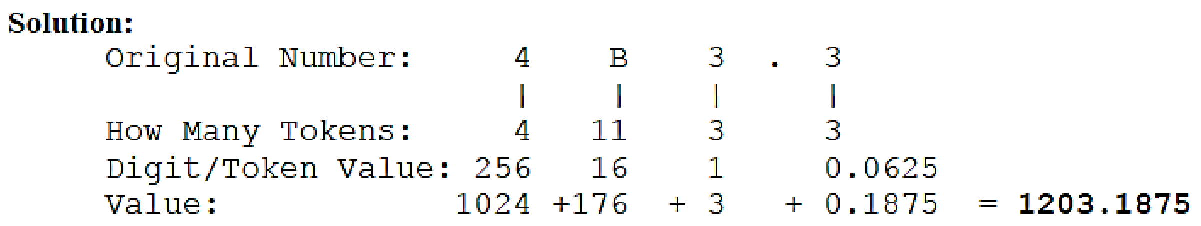
$$x_{10} = \sum_{i=-m}^{n-1} d_i \, b^i$$

Where b is the base, d_i the digit at position i, m the number of digit after the decimal point, n the number of digits of the integer part and x_{10} is the obtained number in decimal. This form the basic of the polynomial method of converting numbers from any base to decimal

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

$2*8^2 + 3*8^1 + 4*8^0+1*8^{-1} + 4*8^{-2} = 2*64 +3*8 +4*1 +1/8 +4/64 =156.1875$

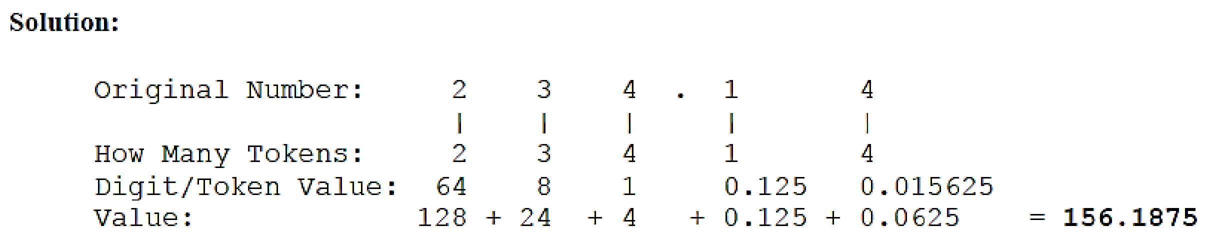
Example. Convert the hexadecimal number 4B3 to decimal notation. What about the decimal equivalent of the hexadecimal number 4B3.3?



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. Convert 234.14 expressed in an octal notation to decimal.

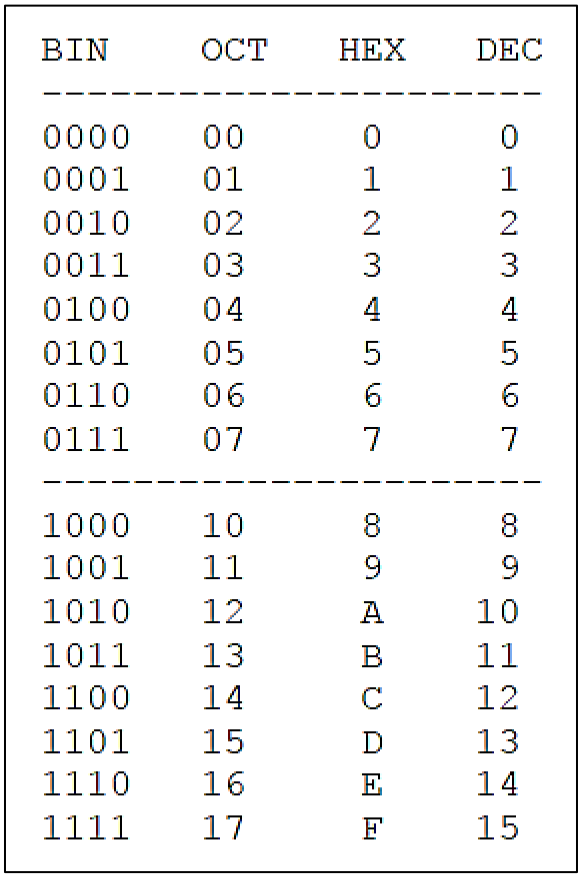


Solution:

```
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
```

<u>Relationship between Binary - Octal and Binary-hexadecimal</u>

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.



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

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:

$10110_2 = 010_2\ 110_2 = 2_8\ 6_8 = 26_8$

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

Now consider converting 101101102 to base 16:

$10110110_2 = 1011_2\ 0110_2 = B_{16}\ 6_{16} = B6_{16}$

(Note that 'B' is a base 16 digit corresponding to 11_{10}. 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 forms 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.

Lecture Two

Binary Arithmetic Operations

Arithmetic operations in digital systems are usually done in binary because design of logic circuits to perform binary arithmetic is much easier than for decimal. Binary arithmetic is carried out in much the same manner as decimal, except the addition and multiplication tables are much simpler.

The addition table for binary numbers is

$$\begin{array}{l} 0 + 0 = 0 \\ 0 + 1 = 1 \\ 1 + 0 = 1 \\ 1 + 1 = 0 \quad \text{and carry 1 to the next column} \end{array}$$

Carrying 1 to a column is equivalent to adding 1 to that column.

Example: Add 13_{10} and 11_{10} in binary.

$$\begin{array}{rrl} & {\scriptstyle 1111} & \leftarrow \text{carries} \\ 13_{10} = & 1101 & \\ 11_{10} = & \underline{1011} & \\ & 11000 & = 24_{10} \end{array}$$

The subtraction table for binary numbers is

$$\begin{array}{l} 0 - 0 = 0 \\ 0 - 1 = 1 \quad \text{and borrow 1 from the next column} \\ 1 - 0 = 1 \\ 1 - 1 = 0 \end{array}$$

Borrowing 1 from a column is equivalent to subtracting 1 from that column.

Example:

$$\text{(a)}\quad \begin{array}{r} {\scriptstyle 1} \\ 11101 \\ \underline{-10011} \\ 1010 \end{array} \qquad\qquad \text{(b)}\quad \begin{array}{r} {\scriptstyle 1111} \\ 10000 \\ \underline{-\quad 11} \\ 1101 \end{array} \qquad\qquad \text{(c)}\quad \begin{array}{r} {\scriptstyle 111} \\ 111001 \\ \underline{-\quad 1011} \\ 101110 \end{array}$$

The multiplication table for binary numbers is:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

The following example illustrates multiplication of 13_{10} by 11_{10} in binary:

```
     1101
     1011
  -------
     1101
    1101
   0000
  1101
  -------
  10001111 = 143₁₀
```

Example:

1111	multiplicand
1101	multiplier
1111	first partial product
0000	second partial product
(01111)	sum of first two partial products
1111	third partial product
(1001011)	sum after adding third partial product
1111	fourth partial product
11000011	final product (sum after adding fourth partial product)

The following example illustrates division of 145_{10} by 11_{10} in binary:

```
          1101
     ---------
1011 |10010001
      1011
      ----
       1110
       1011
       ----
         1101
         1011
         ----
           10
```

The quotient is 1101 with a remainder of 10.

Boolean Algebra

The basic mathematics needed for the study of the logic design of digital systems is Boolean algebra. Boolean algebra has many other applications including set theory and mathematical logic. Because all of the switching devices which we will use are essentially two-state devices (such as a transistor with high or low output voltage), we will study the special case of Boolean algebra in which all of the variables assume only one of two values. This two-valued Boolean algebra is often referred to as switching algebra. George Boole developed Boolean algebra in 1847 and used it to solve problems in mathematical logic. Claude Shannon first applied Boolean algebra to the design of switching circuits in 1939.

We will use a Boolean variable, such as X or Y, to represent the input or output of a switching circuit. We will assume that each of these variables can take on only two different values. The symbols "0" and "1" are used to represent these two different values. Thus, if X is a Boolean (switching) variable, then either X = 0 or X = 1.

The symbols "0" and "1" used in Boolean algebra do not have a numeric value; instead they represent two different states in a logic circuit and are the two values of a switching variable. In a logic gate circuit, 0 (usually) represents a range of low voltages, and 1 represents a range of high voltages. In a switch circuit, 0 (usually) represents an open switch, and 1 represents a closed circuit. In general, 0 and 1 can be used to represent the two states in any binary-valued system.

Basic Operations

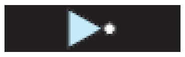
The basic operations of Boolean algebra are AND, OR, and complement (or inverse). The complement of 0 is 1, and the complement of 1 is 0. Symbolically, we write:

$$0' = 1 \text{ and } 1' = 0$$

where the prime (′) denotes complementation. If X is a switching variable,

$$X' = 1 \text{ if } X = 0 \text{ and } X' = 0 \text{ if } X = 1$$

An alternate name for complementation is inversion, and the electronic circuit which forms the inverse of X is referred to as an inverter. Symbolically, we represent an inverter by:

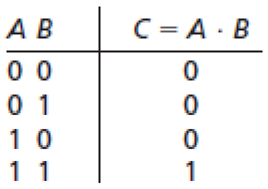


where the circle at the output indicates inversion. If a logic 0 corresponds to a low voltage and a logic 1 corresponds to a high voltage, a low voltage at the inverter input produces a high voltage at the output and vice versa. Complementation is sometimes referred to as the NOT operation because X = 1 if X is not equal to 0.

The AND operation can be defined as follows:

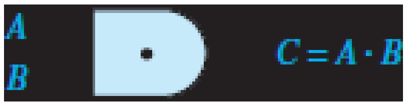
$$0 .0= 0 \qquad 0 .1= 0 \qquad 1.0 =0 \qquad 1.1=1$$

where "**.** " denotes AND. (Although this looks like binary multiplication, it is not, because 0 and 1 here are Boolean constants rather than binary numbers.) If we write the Boolean expression C = A .B, then given the values of A and B, we can determine C from the following table:



A B	$C = A \cdot B$
0 0	0
0 1	0
1 0	0
1 1	1

Note that C =1 iff (if and only if) A and B are both 1, hence, the name AND operation. A logic gate which performs the AND operation is represented by:

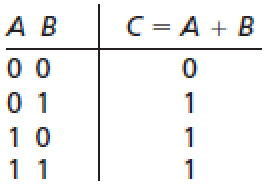


The dot symbol (.) is frequently omitted in a Boolean expression, and we will usually write AB instead of A .B. The AND operation is also referred to as logical (or Boolean) multiplication.

The OR operation can be defined as follows:

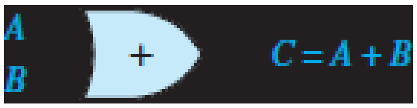
$$0 + 0=0 \quad 0+ 1 =1 \quad 1+ 0=1 \quad 1+ 1 = 1$$

where " + " denotes OR. If we write C = A + B, then given the values of A and B, we can determine C from the following table:



A B	C = A + B
0 0	0
0 1	1
1 0	1
1 1	1

Note that C = 1 iff A or B (or both) is 1, hence, the name OR operation. This type of OR operation is sometimes referred to as inclusive-OR. A logic gate which performs the OR operation is represented by



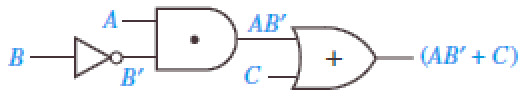
Boolean Expressions and Truth Tables

Boolean expressions are formed by application of the basic operations to one or more variables or constants. The simplest expressions consist of a single constant or variable, such as 0, X, or Y′. More complicated expressions are formed by combining two or more other expressions using AND or OR, or by complementing another expression. Examples of expressions are

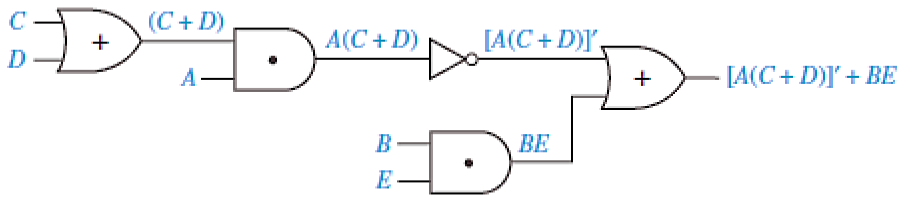
$$AB' + C \qquad (1)$$
$$[A(C + D)]' + BE \qquad (2)$$

Parentheses are added as needed to specify the order in which the operations are performed. When parentheses are omitted, complementation is performed first followed by AND then OR. Thus in Expression (2-1), B′ is formed first, then AB′, and finally AB′ + C.

Each expression corresponds directly to a circuit of logic gates. Figure below gives the circuits for Expressions (1) and (2).



(a)



(b)

An expression is evaluated by substituting a value of 0 or 1 for each variable. If A = B = C = 1 and D= E = 0, the value of Expression (2) is

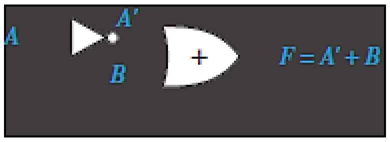
[A(C + D)]′ + BE = [1(1 + 0)]′ + 1 . 0 = [1(1)]′ + 0 = 0 + 0 = 0

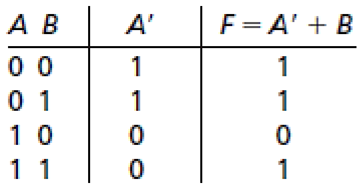
Each appearance of a variable or its complement in an expression will be referred to as a literal. Thus, the following expression, which has three variables, has 10 literals:

$$ab'c + a'b + a'bc' + b'c'$$

When an expression is realized using logic gates, each literal in the expression corresponds to a gate input.

A *truth table* (also called a table of combinations) specifies the values of a Boolean expression for every possible combination of values of the variables in the expression. The name truth table comes from a similar table which is used in symbolic logic to list the truth or falsity of a statement under all possible conditions. We can use a truth table to specify the output values for a circuit of logic gates in terms of the values of the input variables. The output of the circuit in Figure (1)a is F = A′ + B. Figure 1(b) shows a truth table which specifies the output of the circuit for all possible combinations of values of the inputs A and B. The first two columns list the four combinations of values of A and B, and the next column gives the corresponding values of A′. The last column, which gives the values of A′ + B, is formed by ORing together corresponding values of A′ and B in each row.





A B	A′	F = A′ + B
0 0	1	1
0 1	1	1
1 0	0	0
1 1	0	1

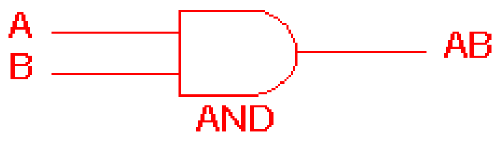
(b)

Lecture Three

Logic Gates

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

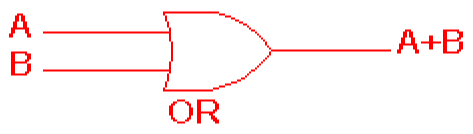
AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a high output (1) only if all its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

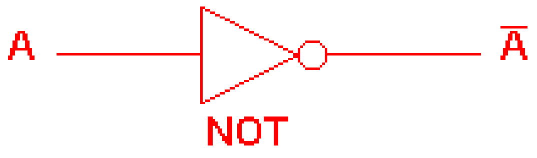
OR gate



2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

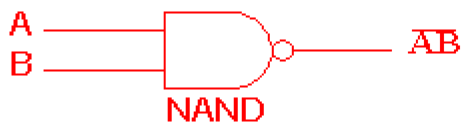
NOT gate



NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.

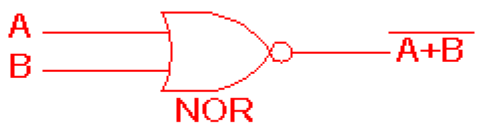
NAND gate



2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if any of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

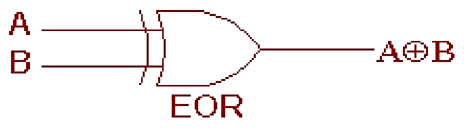
NOR gate



2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if any of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

EXOR gate

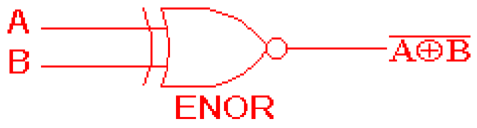


2 Input EXOR gate		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

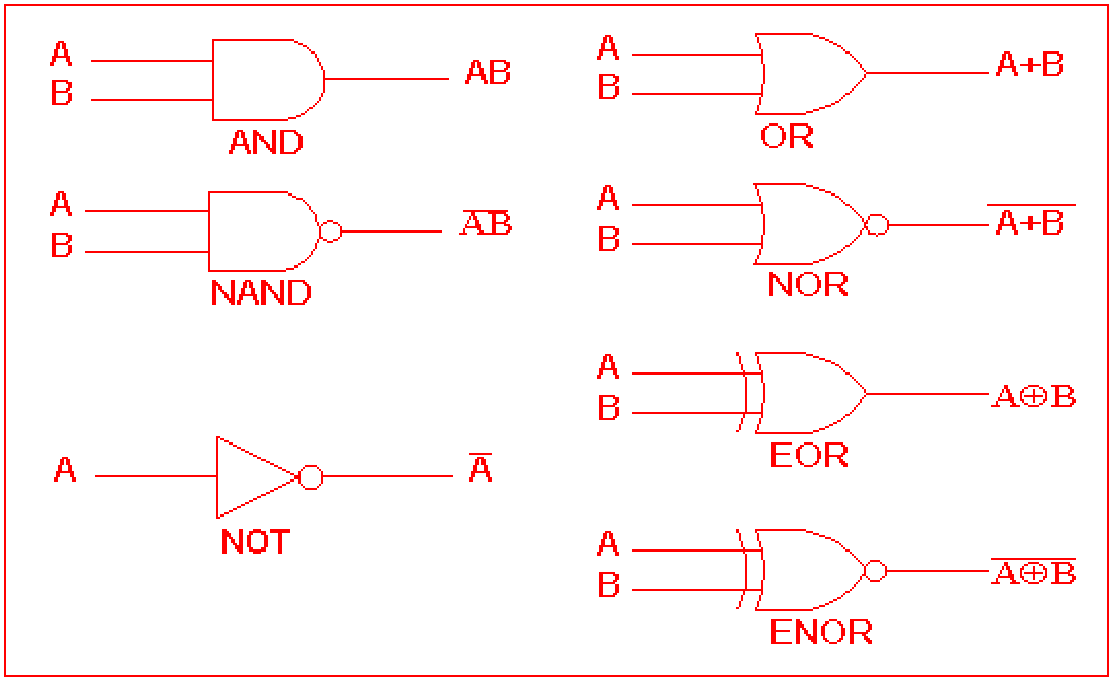
The 'Exclusive-OR' gate is a circuit which will give a high output if either, but not both, of its two inputs are high. An encircled plus sign () is used to show the EOR operation.

The 'Exclusive-NOR' gate circuit does the opposite to the EOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1



Laws and Theorems of Boolean Algebra

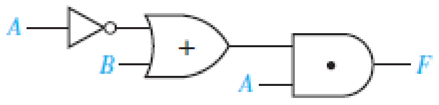
Operations with 0 and 1:	
1. $X + 0 = X$	1D. $X \cdot 1 = X$
2. $X + 1 = 1$	2D. $X \cdot 0 = 0$
Idempotent laws:	
3. $X + X = X$	3D. $X \cdot X = X$
Involution law:	
4. $(X')' = X$	
Laws of complementarity:	
5. $X + X' = 1$	5D. $X \cdot X' = 0$
Commutative laws:	
6. $X + Y = Y + X$	6D. $XY = YX$
Associative laws:	
7. $(X + Y) + Z = X + (Y + Z) = X + Y + Z$	7D. $(XY)Z = X(YZ) = XYZ$
Distributive laws:	
8. $X(Y + Z) = XY + XZ$	8D. $X + YZ = (X + Y)(X + Z)$
Simplification theorems:	
9. $XY + XY' = X$	9D. $(X + Y)(X + Y') = X$
10. $X + XY = X$	10D. $X(X + Y) = X$
11. $(X + Y')Y = XY$	11D. $XY' + Y = X + Y$

Example : simplify the expression $F = A(A' + B)$ using algebra theorem.

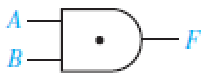
$F = A(A' + B)$

$= AA' + AB \qquad (8. \mathit{distributive\ Law})$

$F = AB \qquad (5D. \quad AA' = 0)$



(a)



(b)

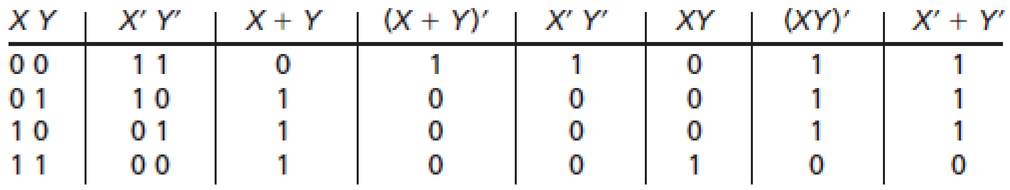
DeMorgan’s Laws

The inverse or complement of any Boolean expression can easily be found by successively applying the following theorems, which are frequently referred to as DeMorgan’s laws:

$$(X + Y)' = X'Y'$$

$$(XY)' = X' + Y'$$

We will verify these laws using a truth table:



X Y	X′ Y′	X + Y	(X + Y)′	X′ Y′	XY	(XY)′	X′ + Y′
0 0	1 1	0	1	1	0	1	1
0 1	1 0	1	0	0	0	1	1
1 0	0 1	1	0	0	0	1	1
1 1	0 0	1	0	0	1	0	0

DeMorgan’s laws are easily generalized to n variables:

$$(X_1 + X_2 + X_3 + \cdots + X_n)' = X_1' \, X'_2 \, X'_3 \dots X_n'$$

$$(X_1 \ X_2 \ X_3 \ \dots \ X_n)' = X_1' + X_2' + X_3 \dots + X_n'$$

For example, for n=3,

$$(X_1 + X_2 + X_3)' = (X_1 + X_2)' \, X_3' = X_1' \, X_2' \, X_3'$$

Referring to the OR operation as the logical sum and the AND operation as logical product, DeMorgan’s laws can be stated as:

The complement of the product is the sum of the complements.

The complement of the sum is the product of the complements.

Lecture Four

Simplification and Boolean Functions

When a function is realized using AND and OR gates, the cost of realizing the function is directly related to the number of gates and gate inputs used. The Karnaugh map techniques developed in this unit lead directly to minimum cost two-level circuits composed of AND and OR gates. An expression consisting of a sum of product terms corresponds directly to a two-level circuit composed of a group of AND gates feeding a single OR gate. Similarly, a product-of sums expression corresponds to a two-level circuit composed of OR gates feeding a single AND gate. Therefore, to find minimum cost two-level AND-OR gate circuits, we must find minimum expressions in sum-of-products or product-of-sums form. A minimum sum-of-products expression for a function is defined as a sum of product terms which has a minimum number of terms and of all those expressions which have the same minimum number of terms, has a minimum number of literals. The minimum sum of products corresponds directly to a minimum two-level gate circuit which has a minimum number of gates and a minimum number of gate inputs.

Minimization of Boolean expressions

The minimization will result in reduction of the number of gates (resulting from less number of terms) and the number of inputs per gate (resulting from less number of variables per term). The minimization will reduce cost, efficiency and power consumption.

- y(x+x`)=y.1=y
- y+xx`=y+0=y
- (x`y+xy`)=x⊕y
- (x`y`+xy)=(x⊕y)`

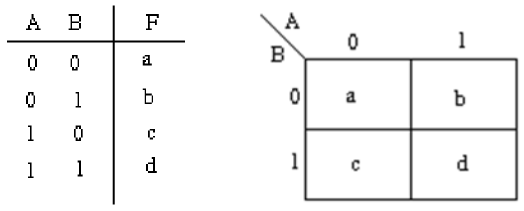
Karnaugh Maps - Rules of Simplification

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 Karnaugh map provides a simple and straight-forward method of minimizing e Boolean expressions. With the Karnaugh map Boolean expressions having up to four and even six variables can be simplified.

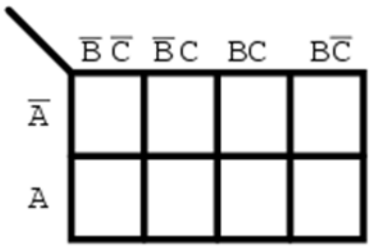
A Karnaugh map provides a pictorial method of grouping together expressions with common factors and therefore eliminating unwanted variables. The Karnaugh map can also be described as a special arrangement of a truth table.

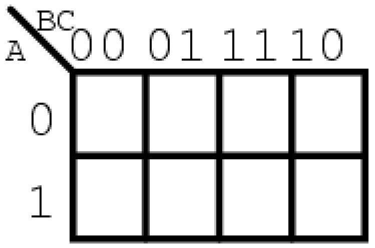
The diagram below illustrates the correspondence between the Karnaugh map and the truth table for the general case of a two variable problem.



A	B	F
0	0	a
0	1	b
1	0	c
1	1	d

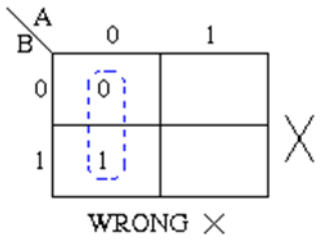
B \ A	0	1
0	a	b
1	c	d

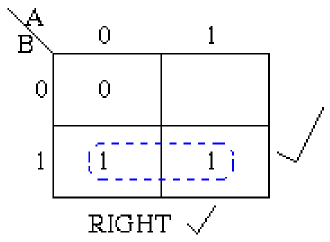




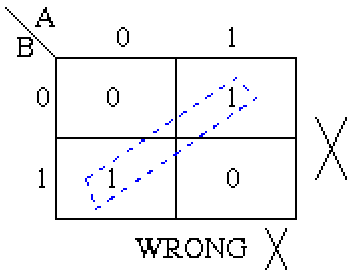
The Karnaugh map uses the following rules for the simplification of expressions by grouping together adjacent cells containing ones.

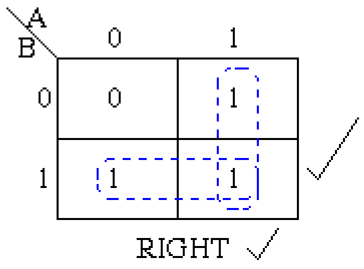
- Groups may not include any cell containing a zero



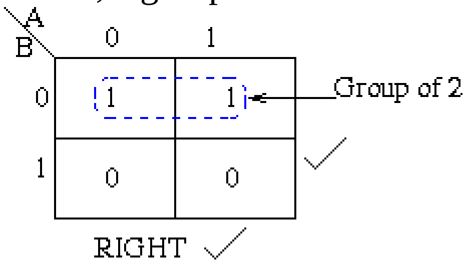


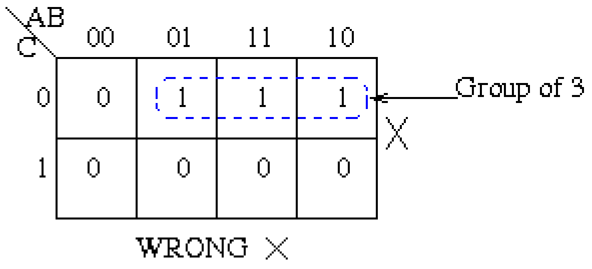
- Groups may be horizontal or vertical, but not diagonal.

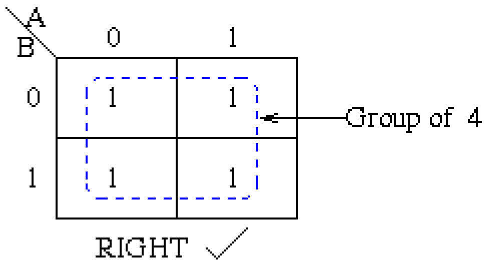


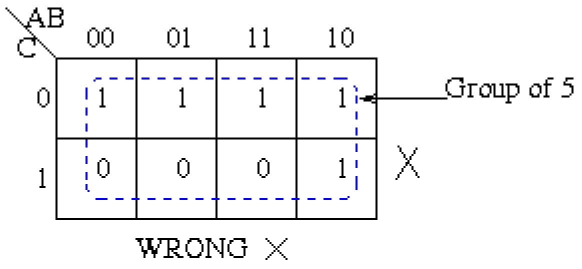


- Groups must contain 1, 2, 4, 8, or in general 2^n cells.
  That is if n = 1, a group will contain two 1's since $2^1 = 2$.
  If n = 2, a group will contain four 1's since $2^2 = 4$.

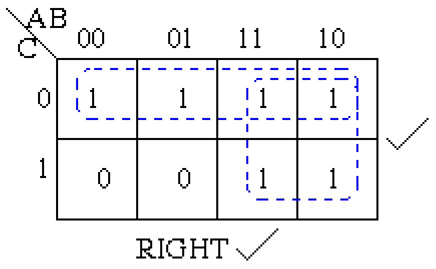


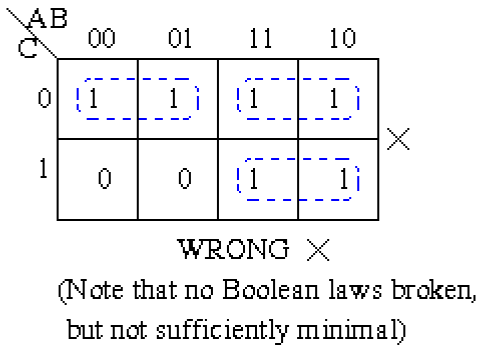




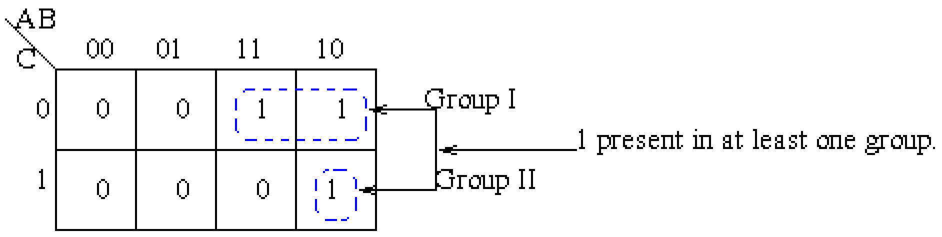


• Each group should be as large as possible.

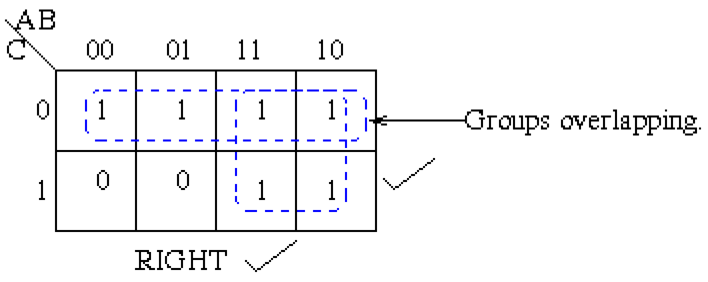


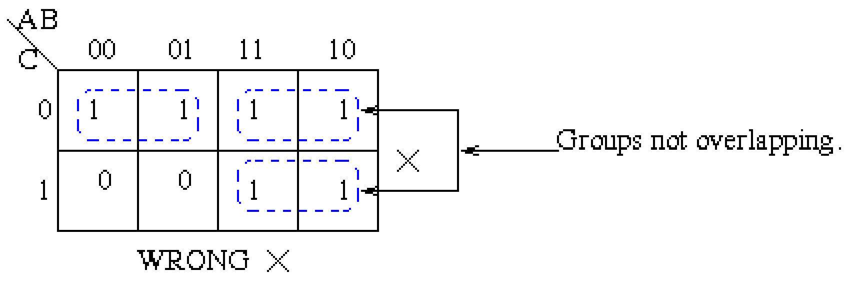

AB
C
00 01 11 10
WRONG ✗
(Note that no Boolean laws broken, but not sufficiently minimal)

• Each cell containing a one must be in at least one group.

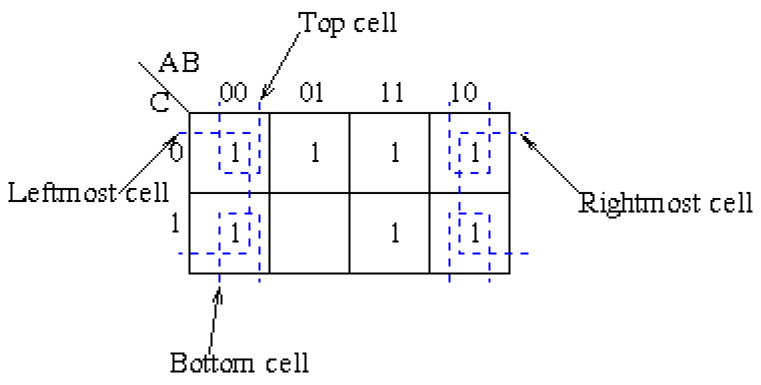


• Groups may overlap.

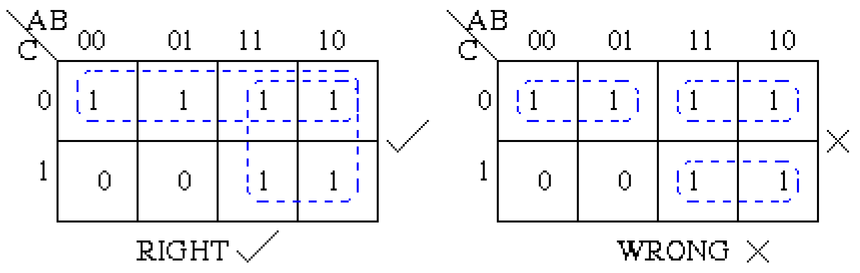




- Groups may wrap around the table. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.



- There should be as few groups as possible, as long as this does not contradict any of the previous rules.



<u>Summary</u>

1. No zeros allowed.

2. No diagonals.

3. Only power of 2 numbers of cells in each group.

4. Groups should be as large as possible.

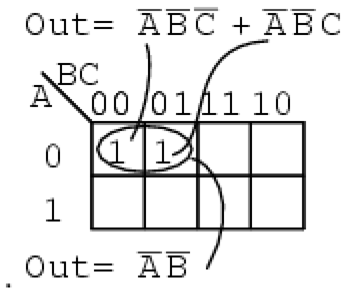
5. Everyone must be in at least one group.

6. Overlapping allowed.

7. Wrap around allowed.

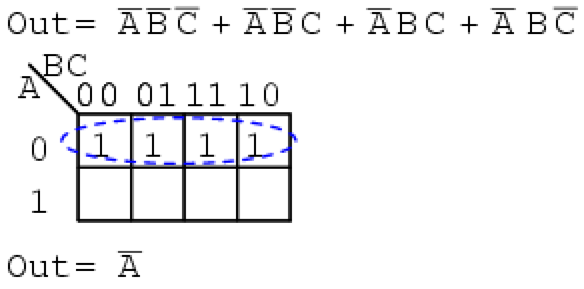
8. Fewest number of groups possible

Example 1:


Out= $\overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C$
BC
A
00 01 11 10
0
1
1 1
Out= $\overline{A}\overline{B}$

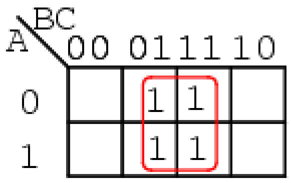
$$A'B'C' + A'B'C = A'B'(C' + C)$$
$$= A'B'$$

Example 2:


Out= $\overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}BC + \overline{A}B\overline{C}$
BC
A
00 01 11 10
0
1
1 1 1 1
Out= $\overline{A}$

Example 3

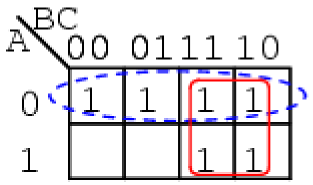
$$Out = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$$



$$Out = C$$

Example 4:

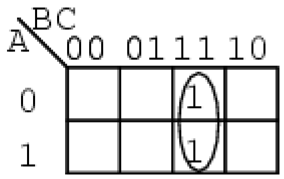
$$Out = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + \bar{A}B\bar{C} + ABC + AB\bar{C}$$



$$Out = \bar{A} + B$$

Example 5:

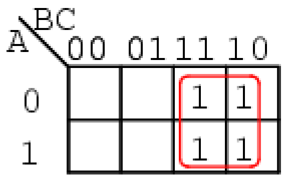
$$Out = \bar{A}BC + ABC$$



$$Out = BC$$

Example 6:

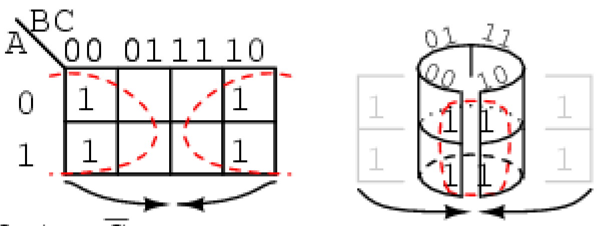
$$Out = \bar{A}BC + \bar{A}B\bar{C} + ABC + AB\bar{C}$$



$$Out = B$$

Example 7:

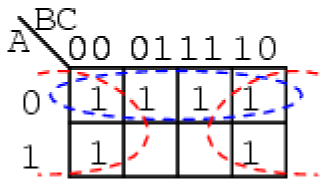
$$Out = \overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + \overline{A}\,B\,\overline{C} + A\,B\,\overline{C}$$



$$Out = \overline{C}$$

Example 8:

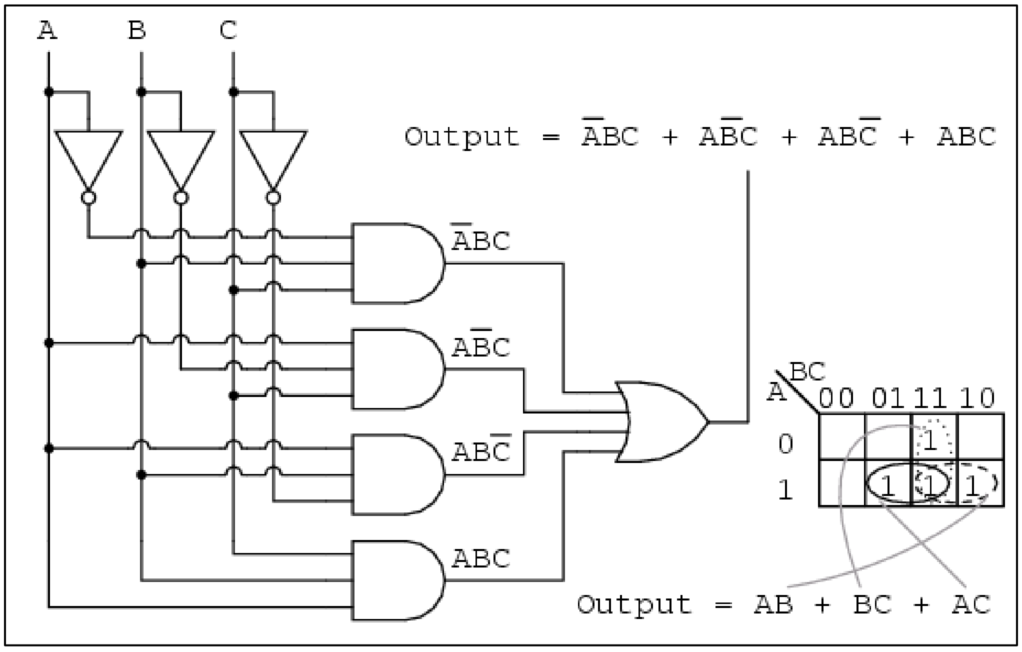
$$Out = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,\overline{C}$$

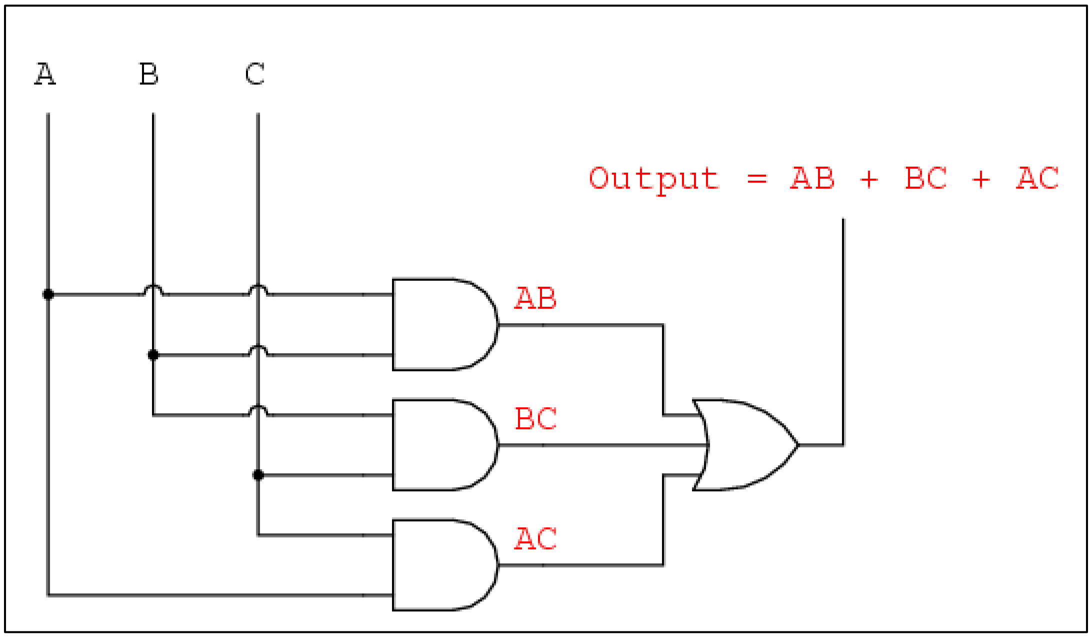


$$Out = \overline{A} + \overline{C}$$

<u>Simplifying Boolean Equations with Karnaugh Maps</u>

Example1: simplify the logic using a Karnaugh map.



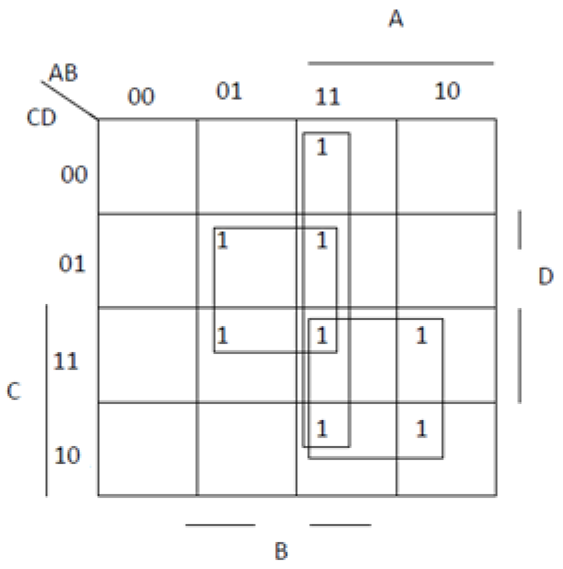

A
B
C
Output = AB + BC + AC
AB
BC
AC

Lecture Five

The following corresponds to the Boolean expression

Example 1:

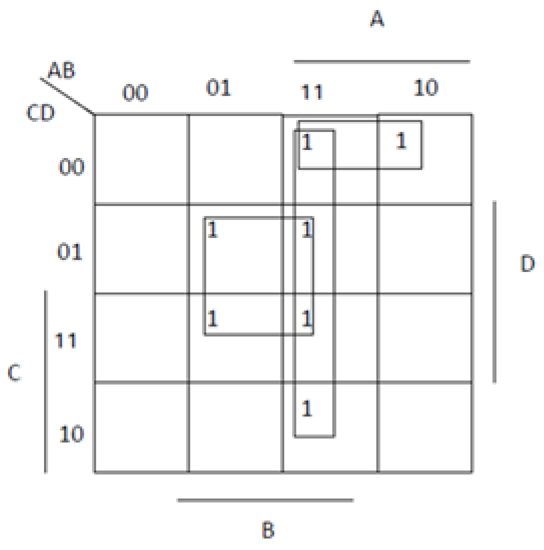
$$Q = A'BC'D + A'BCD + ABC'D' + ABC'D + ABCD + ABCD' + AB'CD + AB'CD'$$



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 2:

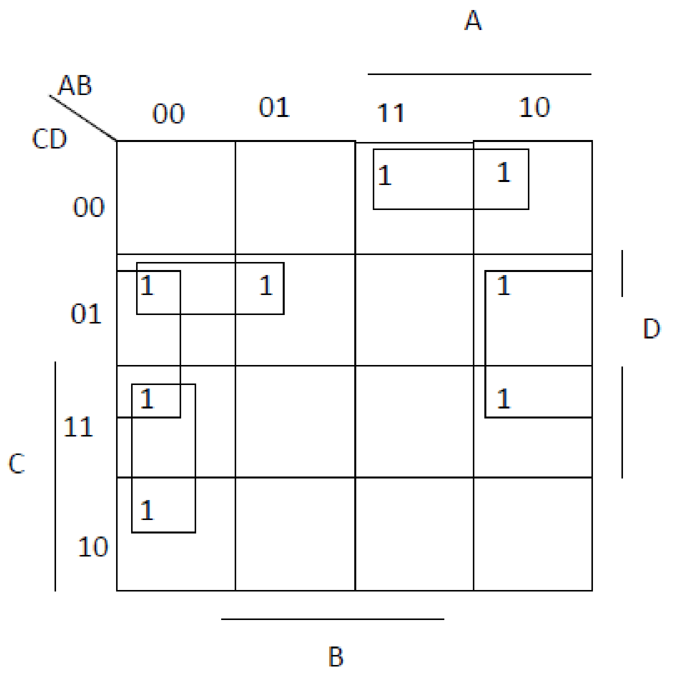
$$F = AB + A'BC'D + A'BCD + AB'C'D'$$



$$= BD + AB + AC'D'$$

Example 3:

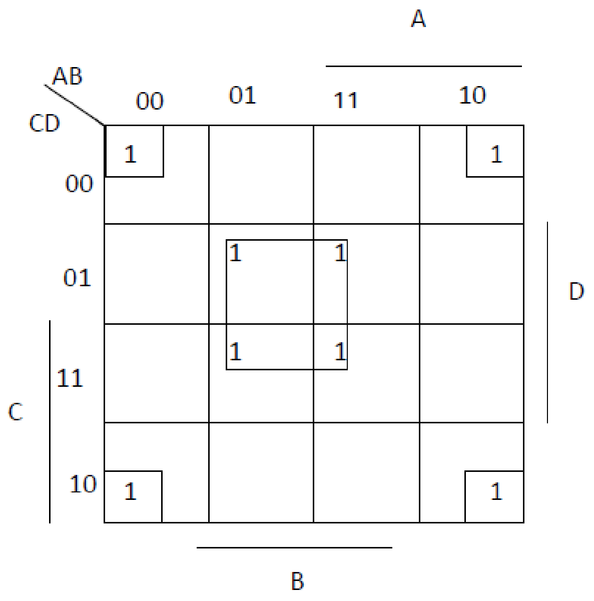
$$F = AC'D' + A'B'C + A'C'D + AB'D$$



=B'D+AC'D'+A'C'D+A'B'C

Example 4:

$$F = A'B'C'D' + AB'C'D' + A'BC'D + ABC'D + A'BCD + ABCD$$



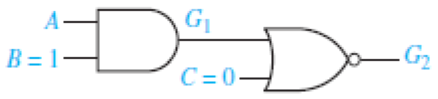
=BD+D'B'

Gate Delays and Timing Diagrams

When the input to a logic gate is changed, the output will not change instantaneously. The transistors or other switching elements within the gate take a finite time to react to a change in input, so that the change in the gate output is delayed with respect to the input change. If the change in output is delayed by time,ϵ , with respect to the input, we say that this gate has a propagation delay of ϵ. In practice, the propagation delay for a 0 to 1 output change may be different than the delay for a 1 to 0 change. Propagation delays for integrated circuit gates may be as short as a few nanoseconds (1 nanosecond = 10^{-9} second), and in many cases these delays can be neglected. However, in the analysis of some types of sequential circuits, even short delays may be important.

Timing diagrams are frequently used in the analysis of sequential circuits. These diagrams show various signals in the circuit as a function of time. Several variables are usually plotted with the same time scale so that the times at which these variables change with respect to each other can easily be observed.

Figure below shows the timing diagram for a circuit with two gates. We will assume that each gate has a propagation delay of 20 ns (nanoseconds).This timing diagram indicates what happens when gate inputs B and C are held at constant values 1 and 0, respectively, and input A is changed to 1 at t = 40 ns and then changed back to 0 at t =100 ns. The output of gate $G1$ changes 20 ns after A changes, and the output of gate $G2$ changes 20 ns after $G1$ changes.



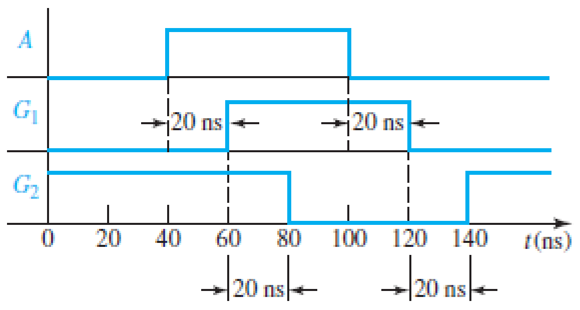
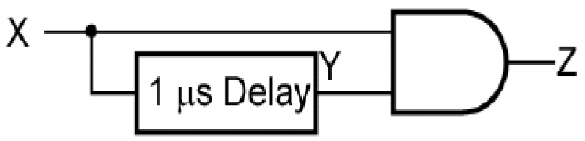
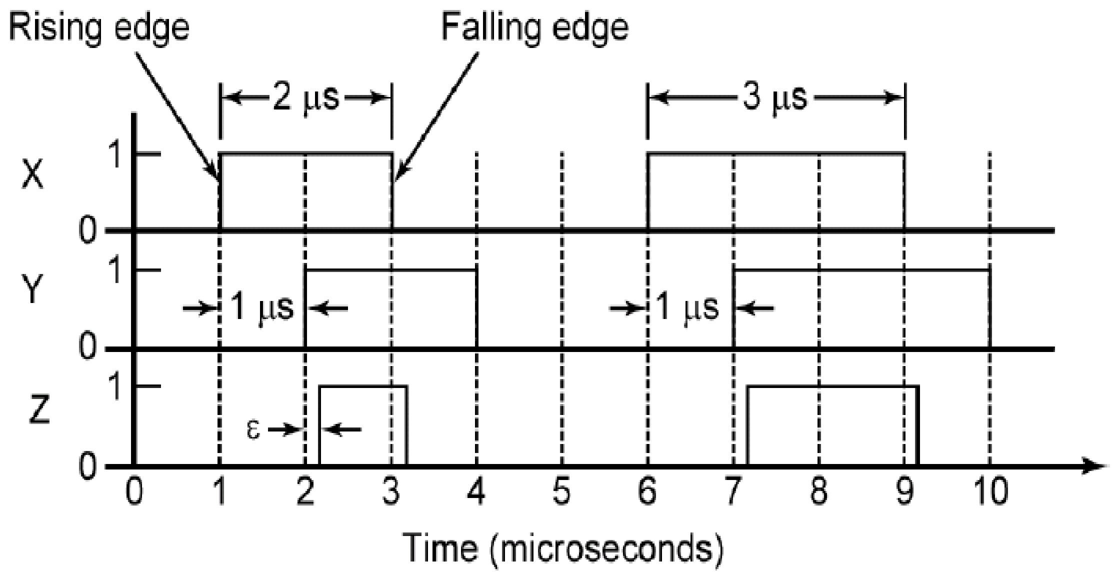


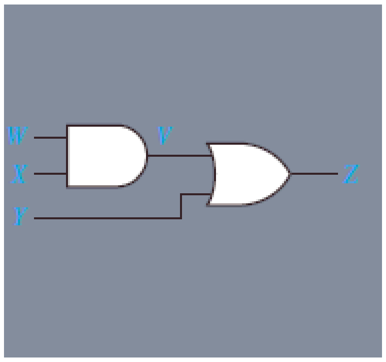
Figure below shows a timing diagram for a circuit with an added delay element. The input X consists of two pulses, the first of which is 2 microseconds (2 $\times 10^6$ second) wide and the second is 3 microseconds wide. The delay element has an output Y which is the same as the input except that it is delayed by 1 microsecond. That is Y changes to a 1 value 1 microsecond after the rising edge of the X pulse and returns to 0 1 microsecond after the falling edge of the X pulse. The output (Z) of the AND gate should be 1 during the time interval in which both X and Y are 1. If we assume a small propagation delay in the AND gate (ε), then Z will be as shown in Figure below.

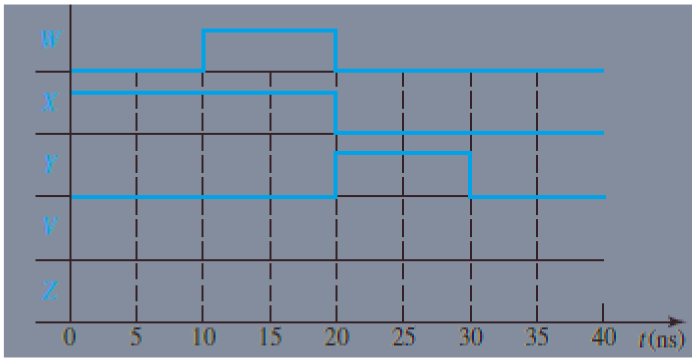




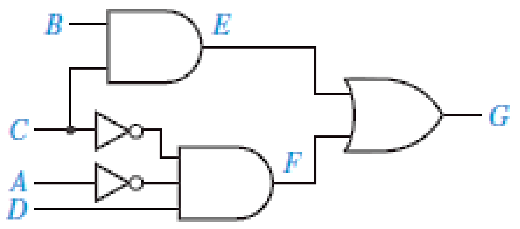
Example 1:

Complete the timing diagram for the given circuit. Assume that both gates have a propagation delay of 5ns.



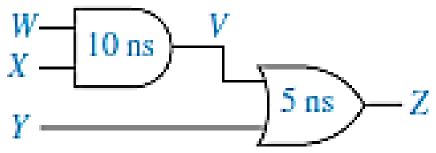


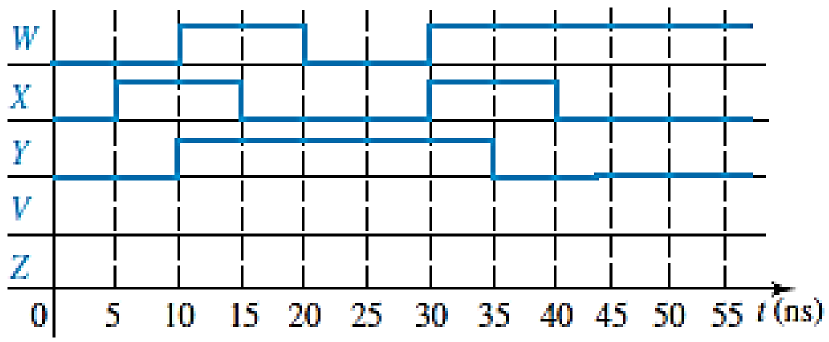
Example 2: for the following circuit:



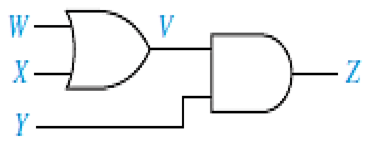
Assume that the inverters have a delay of 1 ns and the other gates have a delay of 2 ns. Initially A = 0 and B = C = D = 1, and C changes to 0 at time = 2 ns. Draw a timing diagram.

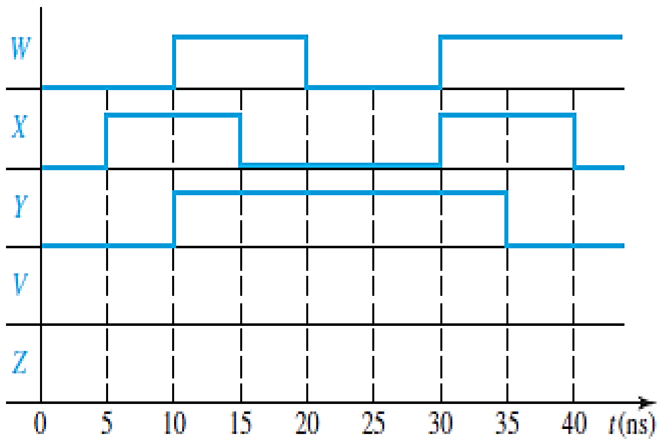
Example 3: Draw the timing diagram for V and Z for the circuit. Assume that the AND gate has a delay of 10 ns and the OR gate has a delay of 5 ns.





Example 4: Complete the timing diagram for the given circuit. Assume that both gates have a propagation delay of 5 ns.





Lecture 6

Sequential Logic circuits

Digital circuits can be classified into two types:

- Combinational Logic Circuits
- Sequential Logic Circuits

1. Combination Logic Circuits:

Are made up from basic gates (AND, OR, NOT) or universal gates (NAND, NOR) gates that are "combined" or connected together to produce more complicated switching circuits. These logic gates are the building blocks of combinational logic circuits. An example of a combinational circuit is a decoder, which converts the binary code data present at its input into a number of different output lines, one at a time producing an equivalent decimal code at its output.

- In these circuits “the outputs at any instant of time depends on the inputs present at that instant only.”

- For the design of Combinational digital circuits Basic gates (AND, OR, NOT) or universal gates (NAND, NOR) are used. Examples for combinational digital circuits are Half adder, Full adder, Half subtractor, Full subtractor, Code converter, Decoder, Multiplexer, Demultiplexer, Encoder, ROM, etc.

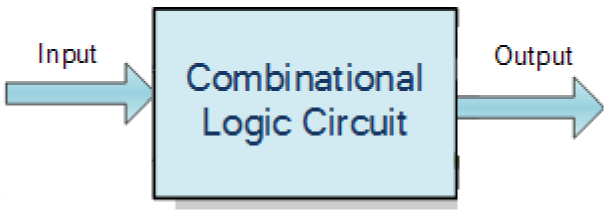


Figure 1.1: combinational Logic Circuit

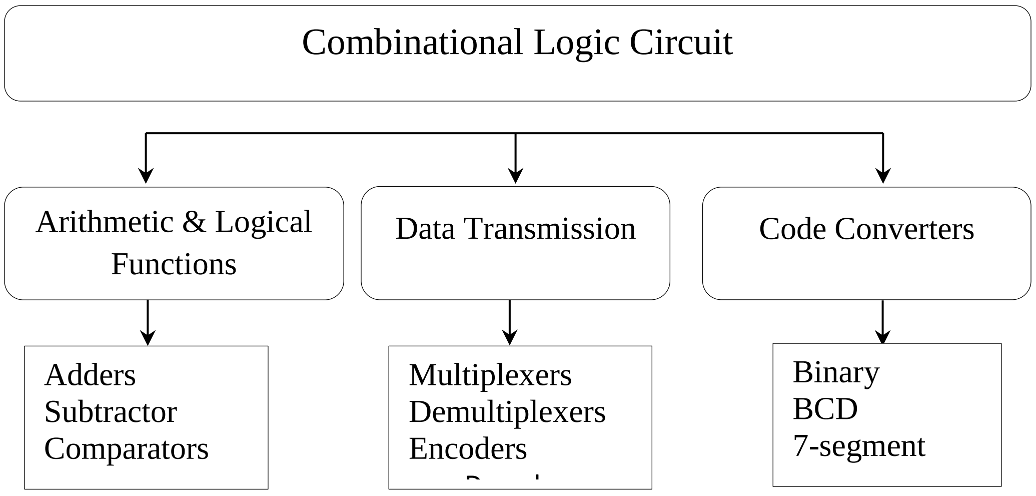


Figure 1.2: Classification of Combinational Logic Circuit

2. <u>**Sequential Logic Circuits**</u>

- Sequential logic differs from combinational logic in that the output of the logic device is dependent not only on the present inputs to the device, but also on past inputs; *i.e.*, the output of a sequential logic device depends on its present internal state and the present inputs. This implies that a sequential logic device has some kind of *memory* of at least part of it's ``history'' (*i.e.*, its previous inputs).

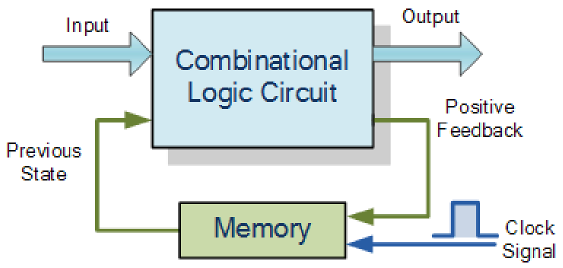


Figure 1.3: sequential logic circuit

- A simple memory device can be constructed from combinational devices with which we are already familiar. By a memory device, we mean a device which can remember if a signal of logic level 0 or 1 has been connected to one of its inputs, and can make this fact available at an output. A very simple, but still useful, memory device can be constructed from a simple OR gate , as shown in Figure below:

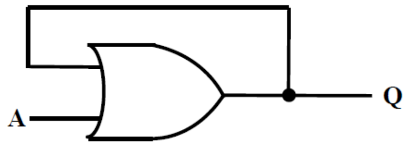


Figure 1.4 Sequential Logic Circuit

- In this memory device, if A and Q are initially at logic 0, then Q remains at logic 0. However if the single input A ever becomes a logic 1, then the output Q will be logic 1 ever after, regardless of any further changes in the input at A. In this simple memory, the output is a function of the state of the memory element only; after the memory is ``written" then it cannot be changed back. However, it can be ``read." Such a device could be used as a simple read only memory, which could be ``programmed" only once. Often a *state table* or *timing diagram* is used to describe the behavior of a sequential device.

- Note that the output of the memory is used as one of the inputs; this is called feedback and is characteristic of programmable memory devices. (Without feedback, a ``permanent" electronic memory device would not be possible.) The use of feedback in a device can introduce problems which are not found in strictly combinational circuits.

- The word “Sequential” means that things happen in a “sequence”, one after another and in Sequential Logic circuits, the actual clock signal determines when things will happen next. Simple sequential logic circuits can be constructed from standard Bistable circuits such as: Flip-flops, Latches and Counters and which themselves can be made by simply connecting together universal NAND Gates and/or NOR Gates in a particular combinational way to produce the required sequential circuit.

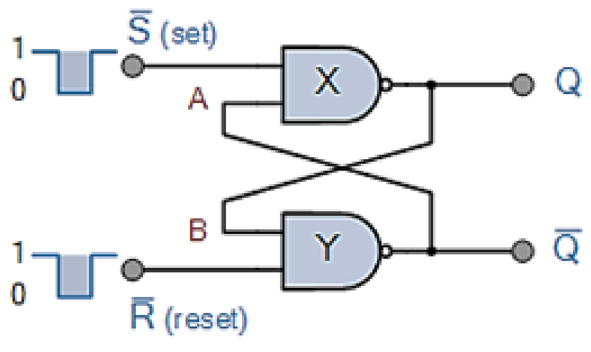
Flip- Flops

A flip flop is an electronic circuit with two stable states that can be used to store binary data, it can store one bit of information. The stored data can be changed by applying varying inputs. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Flip-flops and latches are used as data storage elements. It is the basic storage element in sequential logic.

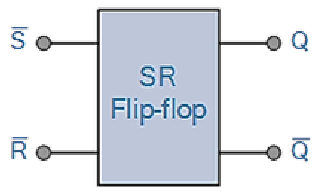
SR Flip-Flop

The SR flip-flop, also known as a SR Latch (figure 1.5), can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will “SET” the device (meaning the output = “1”), and is labeled S and one which will “RESET” the device (meaning the output = “0”), labeled R.

Then the SR description stands for “Set-Reset”. The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.



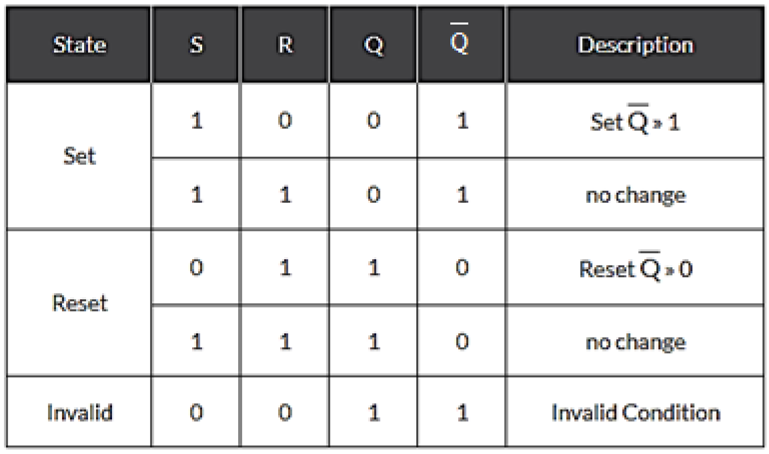
(a) Circuit



(b) Symbol

Figure 1.5 (a) Circuit, (b) Symbol

Table 1.1 Truth Table for this Set-Reset Function



State	S	R	Q	$\overline{Q}$	Description
Set	1	0	0	1	Set $\overline{Q}$ » 1
	1	1	0	1	no change
Reset	0	1	1	0	Reset $\overline{Q}$ » 0
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

The Set State

Consider the circuit shown above. If the input R is at logic level “0” (R = 0) and input S is at logic level “1” (S = 1), the NAND gate Y has at least one of its inputs at logic “0” therefore, its output Q must be at a logic level “1” (NAND Gate principles). Output Q is also fed back to input “A” and so both inputs to NAND gate X are at logic level “1”, and therefore its output Q must be at logic level “0”. Again NAND gate principals. If the reset input R changes state, and goes HIGH to logic “1” with S remaining HIGH also at logic level “1”, NAND gate Y inputs are now R = “1” and B = “0”. Since one of its inputs is still at logic level “0” the output at Q still remains HIGH at logic level “1” and there is no change of state. Therefore, the flip-flop circuit is said to be “Latched” or “Set” with Q = “1” and Q = “0”.

Reset State

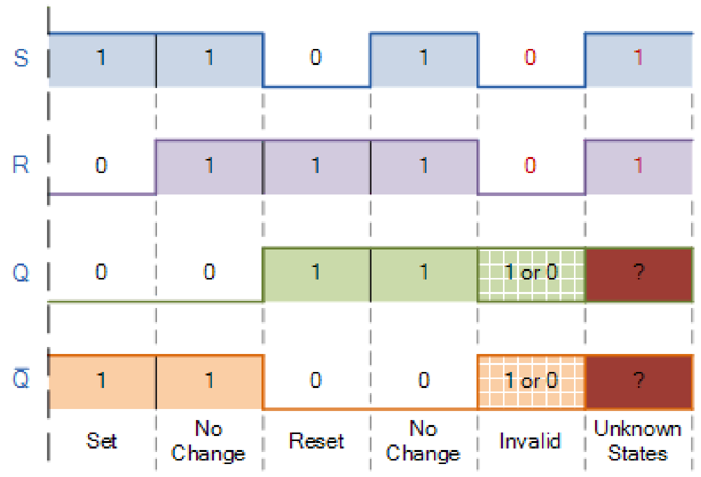
In this second stable state, Q is at logic level “0”, (not Q = “0”) its inverse output at Q is at logic level “1”, (Q = “1”), and is given by R = “1” and S = “0”. As gate X has one of its inputs at logic “0” its output Q must equal logic level “1” (again NAND gate principles). Output Q is fed back to input “B”, so both inputs to NAND gate Y are at logic “1”, therefore, Q = “0”.

If the set input, S now changes state to logic “1” with input R remaining at logic “1”, output Q still remains LOW at logic level “0” and there is no change of state. Therefore, the flip-flop circuits “Reset” state has also been latched and we can define this “set/reset” action in the following truth table.

It can be seen that when both inputs S = “1” and R = “1” the outputs Q and Q can be at either logic level “1” or “0”, depending upon the state of the inputs S or R before this input condition existed. Therefore the condition of S = R = “1” does not change the state of the outputs Q and Q.

However, the input state of S = “0” and R = “0” is an undesirable or invalid condition and must be avoided. The condition of S = R = “0” causes both outputs Q and Q to be HIGH together at logic level “1” when we would normally want Q to be the inverse of Q. The result is that the flip-flop loses control of Q and Q, and if the two inputs are now switched “HIGH” again after this condition to logic “1”, the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance as shown in the following switching diagram.

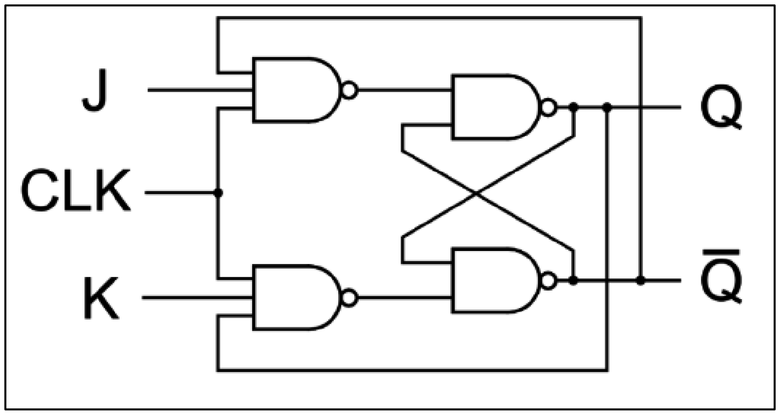
S-R Flip-flop Switching Diagram



Lecture Seven

JK Flip-flop

Due to the undefined state in the SR flip flop, another is required in electronics. The JK flip flop is an improvement on the SR flip flop where S=R=1 is not a problem.

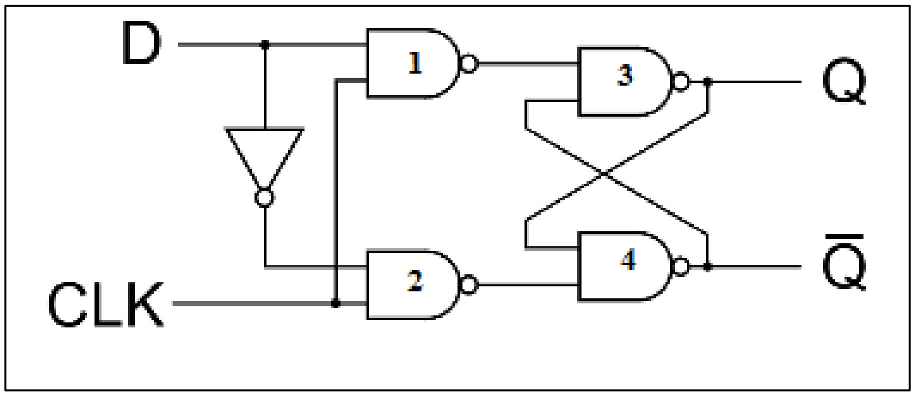


The input condition of J=K=1, gives an output inverting the output state. However, the outputs are same when one tests the circuit practically.

J	K	Q	Q'
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0

D Flip Flop

D flip flop is a better alternative that is very popular with digital electronics. They are commonly used for counters and shift-registers and input synchronization.



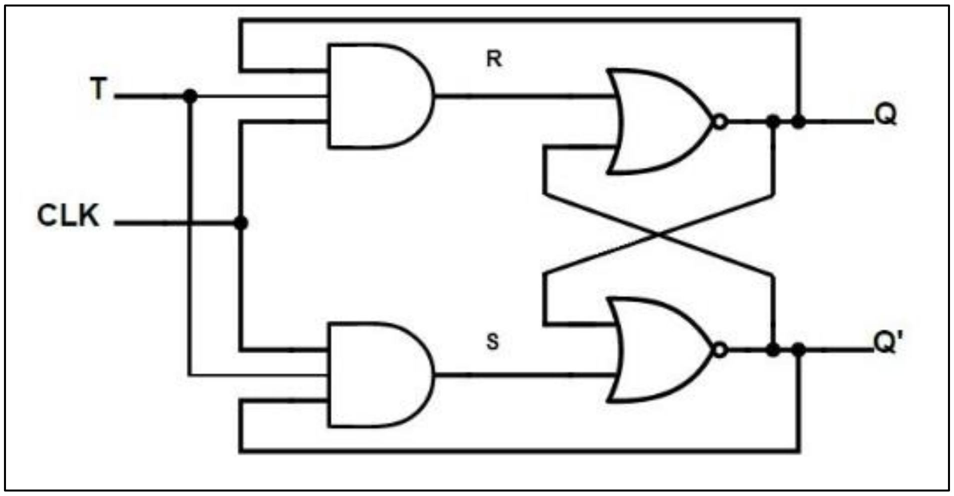
In a D flip flop, the output can be only changed at the clock edge, and if the input changes at other times, the output will be unaffected.

Clock	D	Q	Q'
↓ » 0	0	0	1
↑ » 1	0	0	1
↓ » 0	1	0	1
↑ » 1	1	1	0

The change of state of the output is dependent on the rising edge of the clock. The output (Q) is same as the input and can only change at the rising edge of the clock.

T Flip Flop

A T flip flop is like JK flip-flop. These are basically single input version of JK flip flop. This modified form of JK flip-flop is obtained by connecting both inputs J and K together. This flip-flop has only one input along with the clock input.



The “T” in “T flip-flop” stands for “toggle.” When you toggle a light switch, you are changing from one state (on or off) to the other state (off or on). This is equivalent to what happens when you provide a logic-high input to a T flip-flop: if the output is currently logic high, it changes to logic low; if it’s currently logic low, it changes to logic high. A logic-low input causes the T flip-flop to maintain its current output state.

These flip-flops are called T flip-flops because of their ability to complement its state (i.e.) Toggle, hence the name Toggle flip-flop.

T	**Q**	**Q (t+1)**
0	0	0
1	1	1
0	1	1
1	1	0

Lecture Eight

Shift Registers

A Flip flops can be used to store a single bit of binary data (1or 0). However in order to store multiple bits of data we need multiple flip flops. N flip flops are to be connected in an order to store n bits of data. A Register is a device which is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data.

The information stored within these registers can be transferred with the help of shift registers. Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.

The registers which will shift the bits to left are called “Shift left registers”.

The registers which will shift the bits to right are called “Shift right registers”.

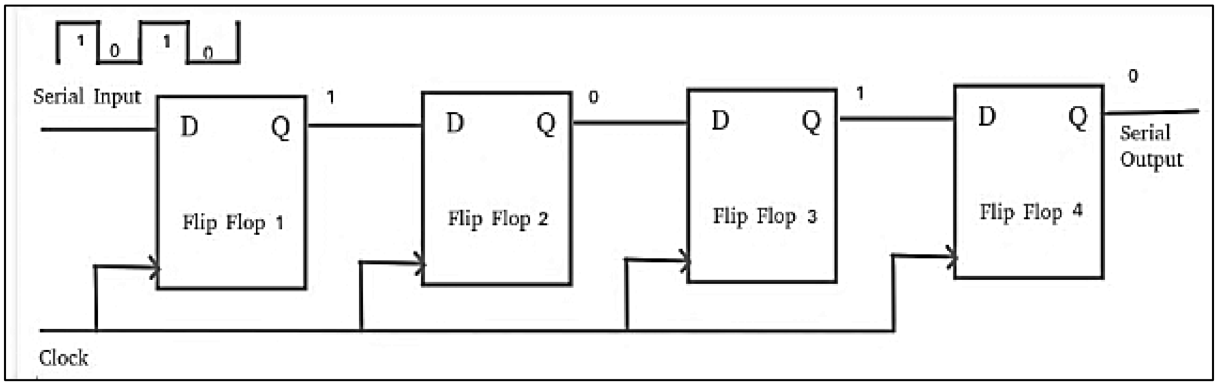
Shift registers are basically of 4 types. These are:

1. Serial In serial out shift register
2. Serial In parallel out shift register
3. Parallel In serial out shift register
4. Parallel In parallel out shift register

Serial-In Serial-Out Shift Register (SISO)

The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output, is known as Serial-In Serial-Out shift register. Since there is only one output, the data leaves the shift register one bit at a time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.

The logic circuit given below shows a serial-in serial-out shift register. The circuit consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

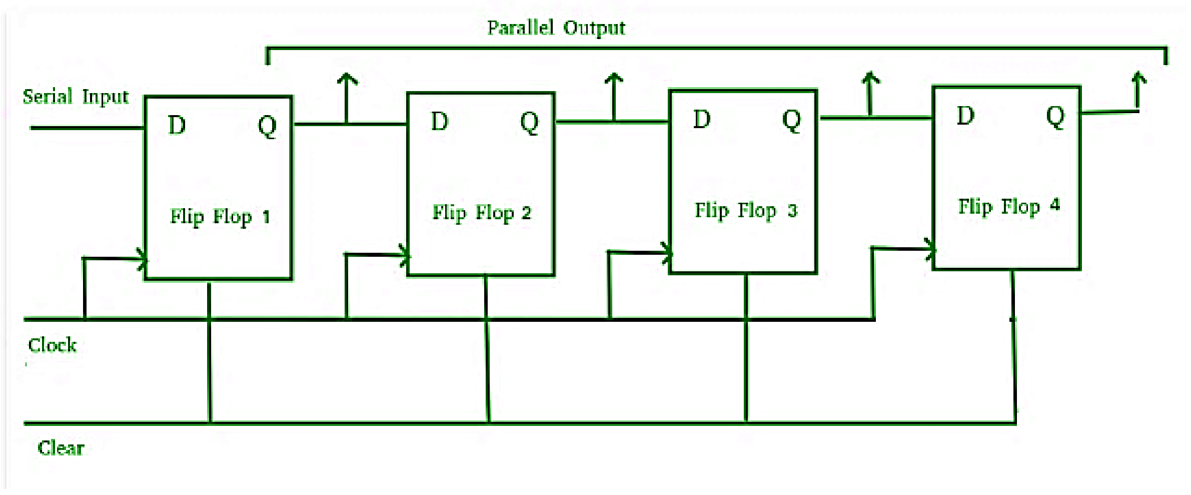


The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop. The main use of a SISO is to act as a delay element.

Serial-In Parallel-Out shift Register (SIPO)

The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output, is known as Serial-In Parallel-Out shift register.

The logic circuit given below shows a serial-in parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal is connected in addition to clock signal to all the 4 flip flops in order to RESET them. The output of the first flip flop is connected to the input of the next flip flop and so on. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.



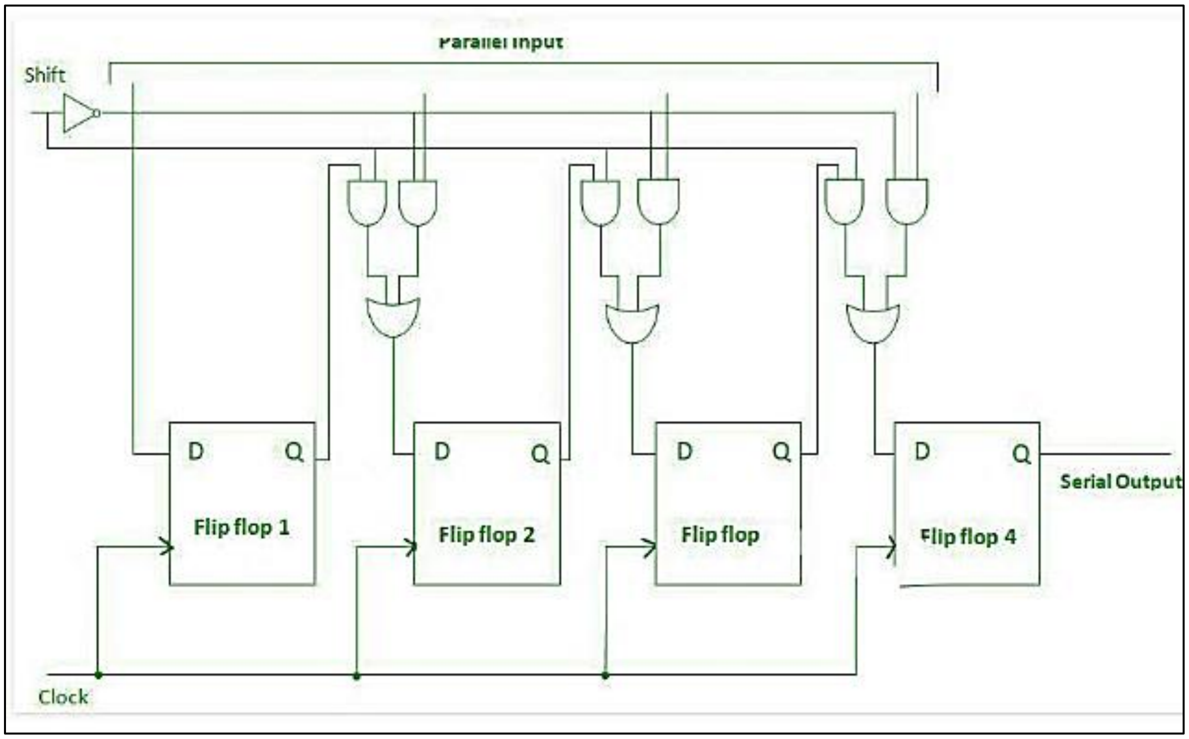
The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop and producing a parallel output. They are used in communication lines where multiplexing of a data line into several parallel line is required because the main use of SIPO register is to convert serial data into parallel data.

Lecture Nine

Parallel-In Serial-Out Shift Register (PISO)

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as Parallel-In Serial-Out shift register.

The logic circuit given below shows a parallel-in serial-out shift register. The circuit consists of four D flip-flops which are connected. The clock input is directly connected to all the flip flops but the input data is connected individually to each flip flop through a multiplexer at input of every flip flop. The output of the previous flip flop and parallel data input are connected to the input of the MUX and the output of MUX is connected to the next flip flop. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

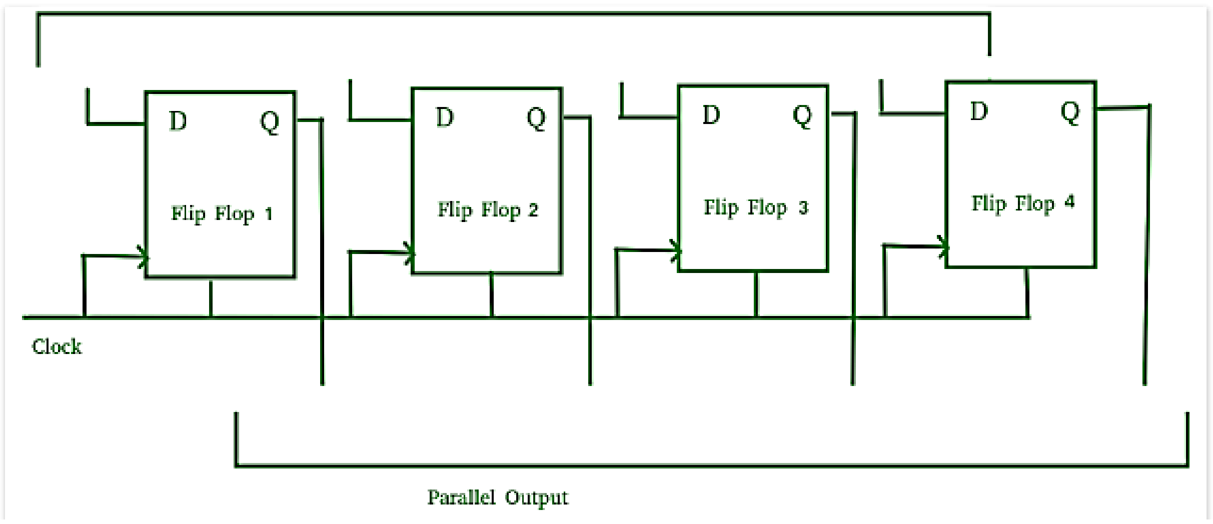


A Parallel in Serial out (PISO) shift register us used to convert parallel data to serial data.

Parallel-In Parallel-Out Shift Register (PIPO)

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register.

The logic circuit given below shows a parallel-in parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal and clock signals are connected to all the 4 flip flops. In this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required. Data is given as input separately for each flip flop and in the same way, output also collected individually from each flip flop.


D
Q
Flip Flop 1
D
Q
Flip Flop 2
D
Q
Flip Flop 3
D
Q
Flip Flop 4
Clock
Parallel Output