DIGITAL SYSTEM DESIGN

COURSEFILE

UNIT-I
Number Systems and Boolean Algebra

Introduction Digital and Analog Signals

Signals carry information and are defined as any physical quantity that varies with time, space, or any other independent variable. For example, a sine wave whose amplitude varies with respect to time or the motion of a particle with respect to space can be considered as signals. A system can be defined as a physical device that performs an operation on a signal. For example, an amplifier is used to amplify the input signal amplitude. In this case, the amplifier performs some operation(s) on the signal, which has the effect of increasing the amplitude of the desired information-bearing signal.

Signals can be categorized in various ways; for example discrete and continuous time domains. Discrete-time signals are defined only on a discrete set of times. Continuous-time signals are often referred to as continuous signals even when the signal functions are not continuous; an example is a square-wave signal.

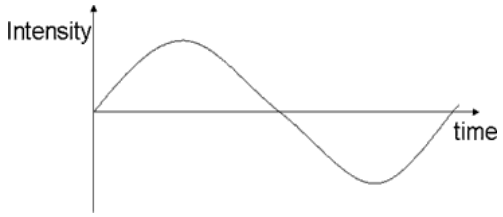


Figure 1a: Analog Signal

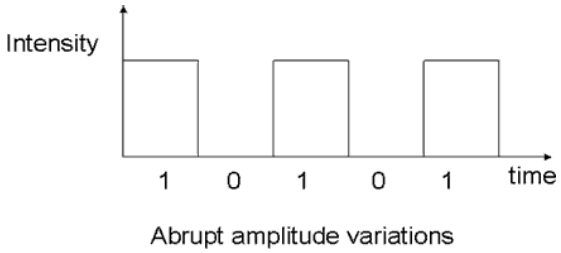


Figure 1b : Digital Signal

Another category of signals is discrete-valued and continuous-valued or otherwise known as digital and analog signals. Digital signals are discrete-valued and analog signals are continuous electrical signals that vary in time as shown in Figure 1 (a) and (b). Analog devices and systems process signals whose voltages or other quantities vary in a continuous manner.

They can take on any value across a continuous range of voltage, current, or other metric. The analog signals can have an infinite number of values. Analog systems can be called wave systems. They have a value that changes steadily over time and can have any one of an infinite set of values in a range. Analog signals represent some physical quantity and they can be a model of the real quantity. Most of the time, the variations corresponds to that of the non-electric (original) signal. For example, the telephone transmitter converts the sounds into an electrical voltage signal. The intensity of the voice causes electric current variations. Therefore, the two are analogous hence the name analog. At the receiving end, the signal is reproduced in the same proportion. Hence the electric current is a model and is an electrical representation of one's voice.

Not all analog signals vary as smoothly as the waveform shown in Fig 1(a). Digital signals are non-continuous, they change in individual steps. They consist of pulses or digits with discrete levels or values. The value of each pulse is constant, but there is an abrupt change from one digit to the next. Digital signals have two amplitude levels. The value of which are specified as one of two possibilities such as 1 or 0, HIGH or LOW , TRUE or FALSE and so on. In reality, the values are anywhere within specific ranges and we define values within a given range.

A digital system is the one that handles only discrete values or signals. Any set that is restricted to a finite number of elements contains discrete information. The word digital describes any system based on discontinuous data or events. Digital is the method of storing, processing and transmitting information through the use of distinct electronic pulses that represent the binary digits 0 and 1. Examples of discrete sets are the 10 decimal digits, the 26 letters of the alphabet etc. A digital system would be to flick the light switch on and off. There's no 'in between' values.

Advantages of digital signals

The usual advantages of digital circuits when compared to analog circuits are:

Noise Margin (resistance to noise/robustness) : Digital circuits are less affected by noise. If the noise is below a certain level (the noise margin), a digital circuit behaves as if there was no noise at all. The stream of bits can be reconstructed into a perfect replica of the original source. However, if the noise exceeds this level, the digital circuit cannot give correct results.

Error Correction and Detection : Digital signals can be regenerated to achieve lossless data transmission, within certain limits. Analog signal transmission and processing, by contrast, always introduces noise.

Easily Programmable : Digital systems interface well with computers and are easy to control with software. It is often possible to add new features to a digital system without changing hardware, and to do this remotely, just by uploading new software. Design errors or bugs can be worked-around with a software upgrade, after the product is in customer hands. A digital system is often preferred because of (re-)programmability and ease of upgrading without requiring hardware changes.

Cheap Electronic Circuits : More digital circuitry can be fabricated per square millimeter of integrated-circuit material. Information storage can be much easier in digital systems than in analog ones. In particular, the great noise-immunity of digital systems makes it possible to store data and retrieve it later without degradation. In an analog system, aging and wear and tear will degrade the information in storage, but in a digital system, as long as the wear and tear is below a certain level, the information can be recovered perfectly. Theoretically, there is no data-loss when copying digital data. This is a great advantage over analog systems, which faithfully reproduce every bit of noise that makes its way into the signal.

Disadvantages The world in which we live is analog, and signals from this world such as light, temperature, sound, electrical conductivity, electric and magnetic fields, and phenomena such as the flow of time, are for most practical purposes continuous and thus analog quantities rather than discrete digital ones. For a digital system to do useful things in the real world, translation from the continuous realm to the discrete digital realm must occur, resulting in quantization errors. This problem can usually be mitigated by designing the system to store enough digital data to represent the signal to the desired degree of fidelity. The Nyquist-Shannon sampling theorem provides an important guideline as to how much digital data is needed to accurately portray a given analog signal.

Digital systems can be fragile, in that if a single piece of digital data is lost or misinterpreted, the meaning of large blocks of related data can completely change. This problem can be diminished by designing the digital system for robustness. For example, a parity bit or other error-detecting or error-correcting code can be inserted into the signal path so that minor data corruptions can be detected and possibly corrected.

Digital circuits use more energy than analog circuits to accomplish the same calculations and signal processing tasks, thus producing more heat as well. In portable or battery-powered systems this can be a major limiting factor.

Digital circuits are made from analog components, and care has to be taken to all noise and timing margins, to parasitic inductances and capacitances, to proper filtering of power and ground connections, to electromagnetic coupling amongst data lines. Inattention to these can cause problems such as "glitches", pulses do not reach valid switching (threshold) voltages, or unexpected ("undecoded") combinations of logic states.

A corollary of the fact that digital circuits are made from analog components is the fact that digital circuits are slower to perform calculations than analog circuits that occupy a similar amount of physical space and consume the same amount of power. However, the digital circuit will perform the calculation with much better repeatability, due to the high noise immunity of digital circuitry.

Number Systems

Introduction

Number systems provide the basis for all operations in information processing systems. In a number system the information is divided into a group of symbols; for example, 26 English letters, 10 decimal digits etc. In conventional arithmetic, a number system based upon ten units (0 to 9) is used. However, arithmetic and logic circuits used in computers and other digital systems operate with only 0's and 1's because it is very difficult to design circuits that require ten distinct states. The number system with the basic symbols 0 and 1 is called binary. ie. A binary system uses just two discrete values. The binary digit (either 0 or 1) is called a bit.

A group of bits which is used to represent the discrete elements of information is a symbol. The mapping of symbols to a binary value is known a binary code. This mapping must be unique. For example, the decimal digits 0 through 9 are represented in a digital system with a code of four bits. Thus a digital system is a system that manipulates discrete elements of information that is represented internally in binary form.

Decimal Numbers

The invention of decimal number system has been the most important factor in the development of science and technology. The decimal number system uses positional number representation, which means that the value of each digit is determined by its position in a number.

The base, also called the radix of a number system is the number of symbols that the system contains. The decimal system has ten symbols: 0,1,2,3,4,5,6,7,8,9. In other words, it has a base of 10. Each position in the decimal system is 10 times more significant than the previous position. The numeric value of a decimal number is determined by multiplying each digit of the number by the value of the position in which the digit appears and then adding the products. Thus the number 2734 is interpreted as

$$2 \times 1000 + 7 \times 100 + 3 \times 10 + 4 \times 1 = 2000 + 700 + 30 + 4$$

Here 4 is the least significant digit (LSD) and 2 is the most significant digit (MSD).

In general in a number system with a base or radix r, the digits used are from 0 to r-1 and the number can be represented as

$$N = a_n r^n + a_{n-1} r^{n-1} + \ldots\ldots\ldots\ldots + a_1 r^1 + a_0 r^0 \quad \text{where, for } n = 0,1,2,3,\ldots.(1)$$

r = base or radix of the system.

a = number of digits having values between 0 and r-1

Equation (1) is for all integers and for the fractions (numbers between 0 and 1), the following equation holds.

$$N = a_{-1} r^{-1} + a_{-2} r^{-2} + \ldots\ldots\ldots + a_{-n+1} r^{-n+1} + a_{-n} r^{-n}$$

Thus for decimal fraction 0.7123

$$N = 0.7000 + 0.0100 + 0.0020 + 0.0003$$

where a -1 = 7

a -2 = 1

a -3 = 2

a -4 = 3

Binary Numbers

The binary number has a radix of 2. As r = 2, only two digits are needed, and these are 0 and 1. Like the decimal system, binary is a positional system, except that each bit position corresponds to a power of 2 instead of a power of 10. In digital systems, the binary number system and other number systems closely related to it are used almost exclusively. Hence, digital systems often provide conversion between decimal and binary numbers. The decimal value of a binary number can be formed by multiplying each power of 2 by either 1 or 0 followed by adding the values together.

Example : The decimal equivalent of the binary number 101010.

$$N = 101010$$

$$= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$= 43$$

In binary r bits can represent $n = 2^r$ symbols. e.g. 3 bits can represent up to 8 symbols, 4 bits for 16 symbols etc. For N symbols to be represented, the minimum number of bits required is the lowest integer 'r" that satisfies the relationship.

$$2^r > N$$

e.g. if N = 26, minimum r is 5 since $2^4 = 16$ and $2^5 = 32$.

Octal Numbers

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, octal and hexadecimal, are used for representing large binary numbers. Octal systems use a base or radix of 8. Thus it has digits

from 0 to 7 (r-1). As in the decimal and binary systems, the positional valued of each digit in a sequence of numbers is

fixed. Each position in an octal number is a power of 8, and each position is 8 times more significant than the previous position.

Example : The decimal equivalent of the octal number 15.2.

$$N = 15.2_8$$
$$= 1 \times 8^1 + 5 \times 8^0 + 2 \times 8^{-1}$$
$$= 13.25$$

Hexadecimal Numbers

The hexadecimal numbering system has a base of 16. There are 16 symbols. The decimal digits 0 to 9 are used as the first ten digits as in the decimal system, followed by the letters A, B, C, D, E and F, which represent the values 10, 11,12,13,14 and 15 respectively. Table 1 shows the relationship between decimal, binary, octal and hexadecimal number systems.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Hexadecimal numbers are often used in describing the data in computer memory. A computer memory stores a large number of words, each of which is a standard size collection of bits. An 8-bit word is known as a **Byte.** A hexadecimal digit may be considered as half of a byte. Two hexadecimal digits constitute one byte, the rightmost 4 bits corresponding to half a byte, and the leftmost 4 bits corresponding to the other half of the byte. Often a half-byte is called nibble.

If "word" size is n bits there are 2n possible bit patterns so only 2n possible distinct numbers can be represented. It implies that all possible numbers cannot be represent and some of these bit patterns (half?) to represent negative numbers. The negative numbers are generally represented with sign magnitude i.e. reserve one bit for the sign and the rest of bits are interpreted directly as the number. For example in a 4 bit system, 0000 to 0111 can be used to positive numbers from +0 to $+2^{n-1}$ and represent 1000 to 1111 can be used for negative numbers from -0 to -2^{n-1}. The two possible zero's redundant and also it can be seen that such representations are arithmetically costly.

Another way to represent negative numbers are by radix and radix-1 complement (also called r's and (r-1)'s). For example -k is represented as R^n -k. In the case of base 10 and corresponding 10's complement with n=2, 0 to 99 are the possible numbers. In such a system, 0 to 49 is reserved for positive numbers and 50 to 99 are for positive numbers.

Examples:

+3 = +3

-3 = 10^2 -3 = 97

2's complement is a special case of complement representation. The negative number -k is equal to 2 n -k. In 4 bits system, positive numbers 0 to 2^{n-1} is represented by 0000 to 0111 and negative numbers -2^{n-1} to -1 is represented by 1000 to 1111. Such a representation has only one zero and arithmetic is easier. To negate a number complement all bits and add 1

Example:

119 10 = 01110111 2

Complementing bits will result

10001000
+1 *add* *1*
10001001
That is 10001001 2 = - 119 $_{10}$

Properties of Two's Complement Numbers

1. X plus the complement of X equals 0.
2. There is one unique 0.
3. Positive numbers have 0 as their leading bit (MSB); while negatives have 1 as their MSB .
4. The range for an n-bit binary number in 2's complement representation is from -2 $^{(n-1)}$ to 2 $^{(n-1)}$ - 1
5. The complement of the complement of a number is the original number.
6. Subtraction is done by addition to the 2's complement of the number.

Value of Two's Complement Numbers
For an n-bit 2's complement number the weights of the bits is the same as for unsigned numbers except of the MSB . For the MSB or sign bit, the weight is -2 $^{n-1}$. The value of the n-bit 2's complement number is given by:

$$A_{\text{2's-complement}} = (a^{n-1}) \times (-2^{n-1}) + (a^{n-2}) \times (2^{n-1}) + \ldots\ldots (a_1) \times (2^1) + a_0$$

For example, the value of the 4-bit 2's complement number 1011 is given by:
= 1 x -2 ^{3} + 0 x 2 ^{2} + 1 x 2 ^{1} + 1
= -8 + 0 + 2 + 1
= -5
An n-bit 2's complement number can converted to an m-bit number where m>n by appending m-n copies of the sign bit to the left of the number. This process is called sign extension. Example: To convert the 4-bit 2's complement number 1011 to an 8-bit representation, the sign bit (here = 1) must be extended by appending four 1's to left of the number:

$$1011_{\text{4-bit 2's-complement}} = 11111011_{\text{8-bit 2's-complement}}$$

To verify that the value of the 8-bit number is still -5; value of 8-bit number
= -27 + 26 + 25 + 24 + 23 +2 +1
= -128 + 64 + 32 + 16 +8 +2+1
= -128 + 123 = -5
Similar to decimal number addition, two binary numbers are added by adding each pair of bits together with carry propagation. An addition example is illustrated below:
X 190
Y 141
X + Y 331

```
1 0 1 1 1 1 0 0 0    Carry
  1 0 1 1 1 1 1 0      X
+ 1 0 0 0 1 1 0 1      Y
1 0 1 0 0 1 0 1 1
```

Similar to addition, two binary numbers are subtracted by subtracting each pair of bits together with borrowing, where needed. For example:
X 229
Y 46
X - Y 183

```
0 0 1 1 1 1 1 0 0    Borrow
1 1 1 0 0 1 0 1        X
0 0 1 0 1 1 1 0        Y
1 0 1 1 0 1 1 1        X - Y
```

Two' complement addition/subtraction example

4	0100	-2	1110
- 7	1001	-6	1010
- 3	1101	-8	1 1000

Overflow occurs if signs (MSBs) of both operands are the same and the sign of the result is different. Overflow can also be detected if the carry in the sign position is different from the carry out of the sign position. Ignore carry out from MSB.

Number Base Conversions

This section describes the conversion of numbers from one number system to another. Radix Divide and Multiply Method is generally used for conversion. There is a general procedure for the operation of converting a decimal number to a

number in base r. If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since each part must be converted differently. The conversion of a decimal integer to a number in base r is done by dividing the number and all successive quotients by r and accumulating the remainders. The conversion of a decimal fraction is done by repeated multiplication by r and the integers are accumulated instead of remainders.
Integer part - repeated divisions by r yield LSD to MSD
Fractional part - repeated multiplications by r yield MSD to LSD
Example: Conversion of decimal 23 to binary is by divide decimal value by 2 (the base) until the value is 0

Integer	remainder	
23		
11	1	→ LSB
5	1	
2	1	
1	0	
0	1	→ MSB

The answer is $23_{10} = 10111_2$
Divide number by 2; keep track of remainder; repeat with dividend equal to quotient until zero; first remainder is binary LSB and last is MSB.
The conversion from decimal integers to any base-r system is similar to this above example, except that division is done by r instead of 2.
Example:
Convert (0.7854) 10 to binary.

$0.7854 \times 2 = 1.5708$; $a_{-1} = 1$
$0.5708 \times 2 = 1.1416$; $a_{-2} = 1$
$0.1416 \times 2 = 0.2832$; $a_{-3} = 0$
$0.2832 \times 2 = 0.5664$; $a_{-4} = 0$

The answer is $(0.7854)_{10} = (0.1100)_2$

Multiply fraction by two; keep track of integer part; repeat with multiplier equal to product fraction; first integer is MSB , last is the LSB; conversion may not be exact; a repeated fraction. The conversion from decimal fraction to any base-r system is similar to this above example, except the multiplication is done by r instead of 2.
The conversion of decimal numbers with both integer and fraction parts is done by converting the integer and the fraction separately and then combining the two answers.

Thus $(23.7854)_{10} = (10111.1100)_2$

For converting a binary number to octal, the following two step procedure can be used.

1. Group the number of bits into 3's starting at least significant symbol. If the number of bits is not evenly divisible by 3, then add 0's at the most significant end.
2. Write the corresponding 1 octal digit for each group

Examples:

```
100 010 111 (binary)
 4   2   7 (octal)

10 101 110 (binary)
 2   5   6 (octal)
```

Similarly for converting a binary number to hex, the following two step procedure can be used.

1. Group the number of bits into 4's starting at least significant symbol. If the number of bits is not evenly divisible by 4, then add 0's at the most significant end.
2. Write the corresponding 1 hex digit for each group

Examples:

```
1001 1110 0111 0000 (binary)
  9    e    7    0  (hex)

1 1111 1010 0011     (binary)
1  f    a    3      (hex)
```

The hex to binary conversion is very simple; just write down the 4 bit binary code for each hexadecimal digit

Example:

```
3    9    c    8        (hex)
0011 1001 1100 1000     (binary)
```

Similarly for octal to binary conversion, write down the 8 bit binary code for each octal digit.
The hex to octal conversion can be carried out in 2 steps; first the hex to binary followed by the binary to octal. Similarly, decimal to hex conversion is completed in 2 steps; first the decimal to binary and from binary to hex as described above.

Boolean Algebra and Basic Operators

Due to historical reasons, digital circuits are called switching circuits, digital circuit functions are called switching functions and the algebra is called switching algebra. The algebraic system known as Boolean algebra named after the mathematician George Boole. George Boole Invented multi-valued discrete algebra (1854) and E. V. Huntington developed its postulates and theorems (1904). Historically, the theory of switching networks (or systems) is credited to Claude Shannon, who applied mathematical logic to describe relay circuits (1938). Relays are controlled electromechanical switches and they have been replaced by electronic controlled switches called logic gates. A special case of Boolean Algebra known as Switching Algebra is a useful mathematical model for describing the combinational circuits. In this section we will briefly discus how the Boolean algebra is applied to the design of digital systems.
Examples of Huntington 's postulates are given below:

Closure

If X and Y are in set (0, 1) then operations $X+Y$ and $X \cdot Y$ are also in set (0, 1)

Identity

$X + 0 = X \qquad X \cdot 1 = X$

Distributive

$X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$

$X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$

Complement

$X + \overline{X} = 1$

$X \cdot \overline{X} = 0$

Note that for each property, one form is the dual of the other; (zeros to ones, ones to zeros, '.' operations to '+' operations, '+' operations to '.' operations).
From the above postulates the following theorems could be derived.

Associative

$X + (Y + Z) = (X + Y) + Z$

$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$

Idempotence

$X \cdot X = X$

$X + X = X$

Absorption

$X + (X \cdot Y) = X$

$X \cdot (X + Y) = X$

Simplification

$X + (\overline{X} \cdot Y) = X + Y$

$X \cdot (\overline{X} + Y) = X \cdot Y$

Consensus

$X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z$

$(X + Y) \cdot (\overline{X} + Z) \cdot (Y + Z) = (X + Y) \cdot (\overline{X} + Z)$

Adjacency

$X \cdot Y + X \cdot \overline{Y} = X$

$(X + Y) \cdot (X + \overline{Y}) = X$

Demorgans

$\overline{X + Y} = \overline{X} \cdot \overline{Y}$
$\overline{X \cdot Y} = \overline{X} + \overline{Y}$
In general form
$\overline{F(\cdot, +, X_1, \ldots X_n)} = G(+, \cdot, \overline{X_1}, \ldots \overline{X_n})$
Very useful for complementing function expressions; for example
$F = X + Y \cdot Z; \quad \overline{F} = \overline{X + Y \cdot Z}$
$\overline{F} = \overline{X} \cdot \overline{Y \cdot Z} \quad F = \overline{X} \cdot (\overline{Y} + \overline{Z})$
$\overline{F} = \overline{X} \cdot \overline{Y} + \overline{X} \cdot \overline{Z}$

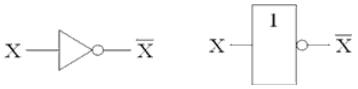
Switching Algebra Operations

A set is a collection of objects (or elements) and for example a set Z {0, 1} means that Z is a set containing two elements distinguished by the symbols 0 and 1. There are three primary operations AND , OR and NOT.

NOT

It is anary complement or inversion operation. Usually shown as over bar ($\overline{X}$), other forms are X' and $\sim X$

X	$\overline{X}$
0	1
1	0



AND

Also known as the conjunction operation; output is true (1) only if all inputs are true. Algebraic operators are '.', '&', ' $\wedge$ '

X	Y	X·Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

Also known as the disjunction operation; output is true (1) if any input is true. Algebraic operators are '+', '|', ' $\vee$ '

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

AND and OR are called binary operations because they are defined on two operands X and Y. Not is called a unary operation because it is defined on a single operand X. All of these operations are closed. That means if one applies the operation to two elements in a set Z {0, 1}, the result will be always an element in the set B and not something else.

Like standard algebra, switching algebra operators have a precedence of evaluation. The following rules are useful in this regard.

1. NOT operations have the highest precedence
2. AND operations are next
3. OR operations are lowest
4. Parentheses explicitly define the order of operator evaluation and it is a good practice to use parentheses especially for situations which can cases doubt.

Note that in Boolean algebra the operators AND and OR are not linear group operations; so one cannot solve equations by "adding to" of "multiplying" on both sides of the equal sign as is done with real, complex numbers in standard algebra.

1.1 Introduction

Number system is a basis for counting various items. On hearing the word 'number', all of us immediately think of the familiar decimal number system with its 10 digits : 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Modern computers communicate and operate with binary numbers which use only the digits 0 and 1. Let us consider decimal number 18. This number is represented in binary as 10010. In the example, if decimal number is considered, we require only two digits to represent the number, whereas if binary number is considered we require five digits. Therefore we can say that, when decimal quantities are represented in the binary form, they take more digits. For large decimal numbers people have to deal with very large binary strings and therefore, they do not like working with binary numbers. This fact gave rise to three new number systems : Octal, Hexadecimal and Binary Coded Decimal (BCD). These number systems represent binary number in a compressed form. Therefore, these number systems are now widely used to compress long strings of binary numbers.

In this chapter, we discuss binary, octal, hexadecimal, and BCD number systems, and we will see how to convert from decimal to binary, octal and hexadecimal, and vice versa. In the later section of this chapter we are going to see binary, hexadecimal, Excess-3 and BCD arithmetic.

1.2 Decimal Number System

Before considering any number system, let us consider familiar decimal number system. In decimal number system we can express any decimal number in units, tens, hundreds, thousands and so on. When we write a decimal number say, 5678.9, we know it can be represented as

$$5000 + 600 + 70 + 8 + 0.9 = 5678.9$$

Binary Number System

We know that decimal system with its ten digits is a base-ten system. Similarly, binary system with its two digits is a base-two system. The two binary digits (bits) are 1 and 0. Like digital system, in binary system each binary digit commonly known as bit, has its own value or weight.

Octal Number System

We know that the base of the decimal number system is 10 because it uses the digits 0 to 9, and the base of binary number system is 2 because it uses digits 0 and 1. The octal number system uses first eight digits of decimal number system : 0, 1, 2, 3, 4, 5, 6, and 7. As it uses 8 digits, its base is 8.

Hexadecimal Number System

The hexadecimal number system has a base of 16 having 16 digits : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. It is another number system that is particularly useful for human communications with a computer. Although it is somewhat more difficult to interpret than the octal number system, it has become the most popular. Since its base is a power of 2 (2^4), it is easy to convert hexadecimal numbers to binary and vice versa.

Converting any Radix to Decimal

In general, numbers can be represented as

$$N = A_{n-1} r^{n-1} + A_{n-2} r^{n-2} + \ldots + A_1 r^1 + A_0 r^0 + A_{-1} r^{-1} + A_{-2} r^{-2} + \ldots C_{-m} r^{-m}$$

where N = Number in decimal

A = Digit

r = Radix or base of a number system

n = The number of digits in the integer portion of number

m = The number of digits in the fractional portion of number

From this general equation we can convert number with any radix into its decimal equivalent.

Conversion of Decimal Numbers to any Radix Number

We have to carry out the conversion of decimal number to any radix number in two steps. In step 1, we have to convert integer part and in step 2 we have to convert fractional part. The conversion of integer part is accomplished by successive division method, and the conversion of fractional part is accomplished by successive multiplication method.

Successive Division for Integer Part Conversion

In this method we repeatedly divide the integer part of the decimal number by r (the new radix) until quotient is zero. The remainder of each division becomes the numeral in the new radix. The remainders are taken in the reverse order to form a new radix number. This means that the first remainder is the least significant digit (LSD) and the last remainder is the most significant digit (MSD) in the new radix number.

Successive Multiplication for Fractional Part Conversion

Conversion of fractional decimal numbers to another radix number is accomplished using a successive multiplication method. In this method, the number to be converted is multiplied by the radix of the new number, producing a product that has an integer part and a fractional part. The integer part (carry) of the product becomes a numeral in the new radix number. The fractional part is again multiplied by the radix and this process is repeated until fractional part reaches 0 or until the new radix number is carried out to sufficient digits. The integer part (carry) of each product is read downward to represent the new radix number.

Complement Representation of Negative Numbers

In digital computers, to simplify the subtraction operation and for logical manipulation complements are used. There are two types of complements for each radix system : **The radix complement** and **diminished radix complement**. The first is referred to as the r's complement and the second as the (r–1)'s complement. For example, in binary system we substitute base value 2 in place of r to refer complements as 2's complement and 1's complement. In decimal number system, we substitute base value 10 in place of r to refer complements as 10's complement and 9's complement.

1's Complement Representation

The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.

2's Complement Representation

The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as

2's complement = 1's complement + 1

The 2's complement form is used to represent negative numbers.

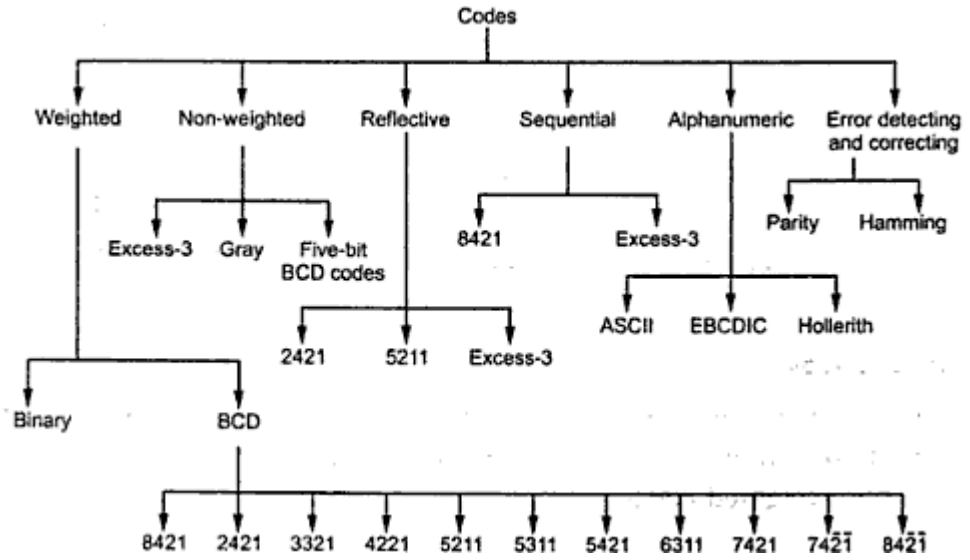
Rules for Binary Addition

A + B	SUM	CARRY
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

Rules for Binary subtraction

A B	Diff.	Borrow
0 – 0	0	0
0 – 1	1	1
1 – 0	1	0
1 – 1	0	0

Classification of Binary Codes



Excess-3 Code

Excess-3 code is a modified form of a BCD number. The Excess-3 code can be derived from the natural BCD code by adding 3 to each coded number. For example, decimal 12 can be represented in BCD as 0001 0010. Now adding 3 to each digit we get Excess-3 code as 0100 0101 (12 in decimal).

Table shows excess-3 codes to represent single decimal digit

Decimal digit	Excess-3 Code			
0	0	0	1	1
1	0	1	0	0
2	0	1	0	1
3	0	1	1	0
4	0	1	1	1
5	1	0	0	0
6	1	0	0	1
7	1	0	1	0
8	1	0	1	1
9	1	1	0	0

Excess-3 code

Excess-3 Addition

To perform Excess-3 addition we have to

- Add two Excess-3 numbers
- If Carry = 1 → add 3 to the sum of two digits
  = 0 → subtract 3

Excess-3 Subtraction

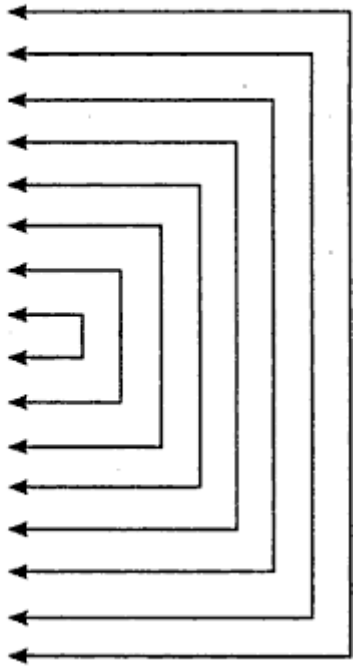
To perform Excess-3 subtraction we have to

- Complement the subtrahend
- Add complemented subtrahend to minuend
- If carry = 1 Result is positive. Add 3 and end around carry
- If carry = 0 Result is negative. Subtract 3.

Gray Code

Gray code is a special case of unit-distance code. In unit-distance code, bit patterns for two consecutive numbers differ in only one bit position. These codes are also called **cyclic codes.**

Decimal Code	Gray code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000



Gray-to-Binary Conversion

The gray to binary code conversion can be achieved using following steps.

1. The most significant bit of the binary number is the same as the most significant bit of the gray code number. So write it down.
2. To obtain the next binary digit, perform an exclusive-OR-operation between the bit just written down and the next gray code bit. Write down the result.
3. Repeat step 2 until all gray code bits have been exclusive-ORed with binary digits. The sequence of bits that have been written down is the binary equivalent of the gray-code number.

Binary to Gray Conversion

Let us represent binary number as

$B_1\ B_2\ B_3\ B_4\ \ldots\ B_n$ and its equivalent gray code as

$G_1\ G_2\ G_3\ G_4\ \ldots\ G_n$.

With this representation gray code bits are obtained from the binary bits follows :

$$G_1 = B_1$$
$$G_2 = B_1 \oplus B_2$$
$$G_3 = B_2 \oplus B_3$$
$$G_4 = B_3 \oplus B_4$$
$$\vdots$$
$$G_n = B_{n-1} \oplus B_n$$

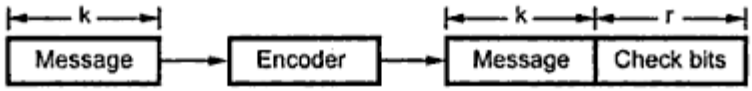
Parity Bit

A parity bit is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1s either odd or even. The message, including the parity bit is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a **parity generator** and the circuit that checks the parity in the receiver is called a **parity checker**.

In even parity the added parity bit will make the total number of 1s an even amount. In odd parity the added parity bit will make the total number of 1s an odd amount.

Linear Block Codes

Block codes are not necessarily linear, but general all block codes used in practice are linear. A linear block code consists of k message bits and r check bits. These r check bits are derived from the original k message bits to form a n-bit block code, as shown in the Fig. The addition of the r check bits introduces redundancy into the code, thus enabling some form of error control. Such a code is designated as an (n, k) code. At the receiving end, the check bits are used to decide the validity of the received message.



Generation of an (n, k) block code

Matrix Representation of Linear Block Codes

In this method, matrices are used to encode the massage. Now before going to see generalized equations for matrix encoding we will see the illustration of matrix encoding with the help of example.

Let us assume that we have to transmit 2-bit binary codes. So we can only have four symbols in our word set. Let our message be :

"a" = 00, "b" = 01, "c" = 10, "d' = 11

Now we have to encode these messages by **coding matrix**. Coding matrix is also called the **generation matrix**. It has the form

$$G = [I_k : A]_{k \times n}$$

where

I_k is the identity matrix of order k and

A is an arbitrary $k \times (n - k)$ sub-matrix.

When the arbitrary sub-matrix A has been specified, the (n, k) block code can be defined completely so that an important step in the design of an (n, k) block code is the structure of A. One of the important criterion in the choice of A is that the resulting code should allow the correction of a codeword received in error.

As an example of the construction of an (n, k) block code, consider the A sub-matrix (2, 2) as

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

We know that generation matrix is given as

$$G = [\, I_k \;\vdots\; A \,] = \begin{bmatrix} 1 & 0 & \vdots & 1 & 1 \\ 0 & 1 & \vdots & 0 & 1 \end{bmatrix}_{(k \times n = 2 \times 4)} \quad \because k = \text{message length} = 2$$

Let us see how to find block code for each message. The block code for each message can be given as,

$$C = DG$$

where C = Block code

D = Message bits

G = Generation matrix

Case 1 : Message '00'

$$C = [0\ 0] \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$= [0 \cdot 1 + 0 \cdot 0 \quad 0 \cdot 0 + 0 \cdot 1 \quad 0 \cdot 1 + 0 \cdot 0 \quad 0 \cdot 1 + 0 \cdot 1]$$

$$= [0\ 0\ 0\ 0]$$

Case 2 : Message '01'

$$C = [0\ 1] \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$= [0 \cdot 1 + 1 \cdot 0 \quad 0 \cdot 0 + 1 \cdot 1 \quad 0 \cdot 1 + 1 \cdot 0 \quad 0 \cdot 1 + 1 \cdot 1]$$

$$= [0\ 1\ 0\ 1]$$

Case 3 : Message '10'

$$C = [1\ 0] \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$= [1 \cdot 1 + 0 \cdot 0 \quad 1 \cdot 0 + 0 \cdot 1 \quad 1 \cdot 1 + 0 \cdot 0 \quad 1 \cdot 1 + 0 \cdot 1]$$

$$= [1\ 0\ 1\ 1]$$

Case 4 : Message '1 1'

$$C = [1\ 1] \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$= [1 \cdot 1 + 1 \cdot 0 \quad 1 \cdot 0 + 1 \cdot 1 \quad 1 \cdot 1 + 1 \cdot 0 \quad 1 \cdot 1 + 1 \cdot 1]$$

$$= [1\ 1\ 1\ 0]$$

The above calculations give the block codes for all messages and are listed in Table

Message		Code words			
		Message		Check bits	
				P_1	P_2
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	1
1	1	1	1	1	0

The (4, 4) code constructed from a specified G matrix

Generalized Steps for Construction of Code

1. Construct G matrix as

$$G = [I_k : A]_{k \times n}$$

where I_k : Identity matrix of order k

A : Arbitrary matrix

$$\underbrace{[C_1, C_2, \ldots C_n]}_{\text{n - code bits}} = \underbrace{[d_1, d_2, \ldots d_k]}_{\text{k - message bits}}$$

$$\left[\begin{array}{cc} 100 \ldots 0 & A_{11}\, A_{12} \ldots A_{r1} \\ 010 \ldots 0 & A_{12}\, A_{22} \ldots A_{r2} \\ 001 \ldots 0 & A_{13}\, A_{23} \ldots A_{r3} \\ \vdots & \vdots \\ \underbrace{000 \ldots 1}_{I_k} & \underbrace{A_{1k}\, A_{2k} \ldots A_{rk}}_{A} \end{array}\right]_{k \times n}$$

2. Determine all possible combinations of code using

$$C = D\,G$$

In general for this can be written as

Note : We have seen that for matrix multiplication we have to use MOD 2 arithmetic, i.e. $1 + 1 = 0$. For multiple additions this can be generalized as $1 \oplus 1 = 0$, or $1 \oplus 1 \oplus 1 = 1$ or $1 \oplus 1 \oplus 1 \oplus 1 = 0$.

Decoding the Received Codewords

At the receiving end the receiver does not know the transmitted word. However, it knows A matrix used for generation of code words. Its function is to check the message bits using check bit along with it. This can be done with the following procedure.

1. From the matrix H as

$$H = [A^T : I_r]$$

where A^T : Transpose (interchanging row and columns) of sub-matrix A

I_r = Identity matrix of the order of r (r = number of check bits)

Matrix H is called **parity-check** matrix.

2. Now if $H\,R^T = 0$ Received word is correct i.e. R = C

$H\,R^T \neq 0$ Error in the received code i.e. $R \neq C$

where R : Received code

R^T : Transpose of R

Error Correction

It is assumed that the coding/decoding system has been designed to correct single error only. In order to correct the codeword we multiply received codeword with transpose of parity-check matrix to get **syndrome**. Then result of RH^T, i.e. syndrome is compared with the row of transpose of parity-check matrix (H^T). Matching row number is the number of bit in error. Error bit is then inverted to get the correct code.

The procedure is given below :

1. Find $S = RH^T$

   where R : Received code

   H^T : Transpose of T

   $S = [S_1, S_2, S_3 \ldots]$ is called **syndrome**.

2. Match the result, i.e. S with row of H^T. The number of row where the match occur gives the number of bit in error. This bit is inverted to correct the error.

Hamming Code

Hamming code not only provides the detection of a bit error, but also identifies which bit is in error so that it can be corrected. Thus Hamming code is called error detecting and correcting code. The code uses a number of parity bits (dependent on the number of information bits) located at certain positions in the code group. Follows sections describe how Hamming code can be constructed for single error correction.

Number of Parity Bits

As mentioned earlier, number of parity bits depend on the number of information bits. If the number of information bits is designed x, then the number of parity bits, P is determined by the following relationship :

$$2^P \geq x + p + 1 \qquad \ldots(1)$$

For example, if we have four information bit, i.e. $x = 4$, then P is found by trial and error using equation 1. Let $p = 2$. Then

$$2^P = 2^2 = 4$$

and

$$x + p + 1 = 4 + 2 + 1 = 7$$

Since 2^P must be equal to or greater than $x + p + 1$, the relationship in equation 1 is not satisfied. Hence we have to try with next value of p. Let $p = 3$.

Then

$$2^P = 2^3 = 8$$

and

$$x + p + 1 = 4 + 3 + 1 = 8$$

This value of p satisfies the relationship given is equation 1, and therefore we can say that three parity bits are required to provide single error correction for four information bits.

Locations of the Parity Bits in the Code

Now we know that how to calculate the number of parity bits required to provide single error correction for given number of information bits. In our example we have four information bits and three parity bits. Therefore, the code is of seven bits. The right-most bit is designated bit 1, the next bit is bit 2, and so on, as shown below :

Bit 7, Bit 6, Bit 5, Bit 4, Bit 3, Bit 2, Bit 1

The parity bits are located in the positions that are numbered corresponding to ascending powers of two (1, 2, 4, 8 ...). Therefore, for 7 - bit code, locations for parity bits and information bit are as shown below :

D_7, D_6, D_5, P_4, D_3, P_2, P_1

where symbol P_n designates a particular parity bit, D_n designates a particular information bit, and n is the location number.

Assigning Values to Parity Bits

Now we know the format of the code. Let us see how to determine 1 or 0 value to each parity bit. In Hamming code, each parity bit provides a check on certain other bits in the total code, therefore, we must know the value of these others in order to assign the parity bit value. To do this, we must write the binary number for each decimal location number as shown in the third row of table.

Bit designation	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	7	6	5	4	3	2	1
Binary location number	111	110	101	100	011	010	001
Information bits (D_n)							
Parity bits (P_n)							

Bit position table for a seven bit error correcting code

Assignment of P_1 : Looking at the Table 3.4 we can see that the binary location number of parity bit P_1 has a 1 for its right-most digit. This parity bit checks all bit locations, including itself, that have **1s in the same location in the binary location numbers.** Therefore, parity bit P_1 checks bit locations 1, 3, 5 and 7, and assigns P_1 according to even or odd parity. For even parity Hamming code, it assigns P_1 such that bit locations 1, 3, 5, and 7 will have even parity.

Assignment of P_2 : Looking at the Table 3.4 we can see that the binary location number of parity bit P_2 has a 1 for its middle bit. This parity bit checks all bit locations, including itself, that have 1s in the middle bit. Therefore, parity bit P_2 checks bit locations 2, 3, 6 and 7 and assigns P_2 according to even or odd parity.

Assignment of P_4 : Looking at the Table 3.4 we can see that the binary location number of parity bit P_4 has a 1 for its left-most digit. This parity bit checks all bit locations, including itself, that have 1s in the left-most bit. Therefore, parity bit P_4 checks bit locations 4, 5, 6 and 7 and assigns P_4 according to even and odd parity.

Detecting and Correcting an Error

In the last section we have seen how to construct Hamming code for given number of information bits. Now we will see how to use it to locate and correct an error. To do this, each parity bit, along with its corresponding group of bits must be checked for proper parity. The correct result of individual parity check is marked by 0 whereas wrong result is marked by 1. After all parity checks, binary word is formed taking resulting bit for P_1 as LSB. This word gives bit location where error has occurred. If word has all bits 0 then there is no error in the Hamming code.

Boolean Postulates and Laws

The postulates of a mathematical system from the basic assumption from which it is possible to deduce the rules, law theorems, and properties of the system. Boolean algebra is formulated by a defined set of elements, together with two binary operators, + and $\cdot$, provided that the following postulates are satisfied.

- **Closure (a) :** Closure with respect to the operator +

  When two binary elements are operated by operator + the result is a unique binary element.

- **Closure (b) :** Closure with respect to the operator $\cdot$ (dot).

  When two binary elements are operated by operator $\cdot$ (dot), the result is a unique binary element.

- An identity element with respect to +, designated by 0 :

  $A + 0 = 0 + A = A$

- An identity element with respect to $\cdot$, designated by 1 : $A \cdot 1 = 1 \cdot A = A$
- Commutative with respect to + : $A + B = B + A$
- Commutative with respect to $\cdot$: $A \cdot B = B \cdot A$
- Distributive property of $\cdot$ over + :

  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

- Distributive property of + over $\cdot$:

  $A + (B \cdot C) = (A + B) \cdot (A + C)$

- For every binary element, there exists complement element. For example, if A is an element, we have $\overline{A}$ is a complement of A. i.e. if $A = 0$, $\overline{A} = 1$ and if $A = 1$, $\overline{A} = 0$.
- There exists at least two elements, say A and B in the set of binary elements such that $A \neq B$.

Rules in Boolean Algebra

1. The symbol which represent an arbitrary elements of an Boolean algebra is known as **variable**. Any single variable or a function of several variables can have either a 1 or 0 value. For example, in expression Y = A + BC, variables A, B and C can have either a 1 or 0 value, and function Y also can have either a 1 or 0 value; however its value depends on the value of Boolean expression.
2. A complement of a variable is represented by a "bar" over the letter. For example, the complement of a variable A will be denoted by $\overline{A}$. So if A = 1, $\overline{A} = 0$ and if A = 0, $\overline{A}$ = 1. Sometimes a prime symbol (') is used to denote the complement. For example, the complement of A can be written as A'.
3. The logical AND operator of two variables is represented either by writing a dot (·) between two variables, such as $A \cdot B$ or by simply writting two variables, such as AB. Similarly, AND operation between three variables can be represented as $A \cdot B \cdot C$ or ABC.
4. The logical OR operator of two variables is represented by writing a '+' sign between the two variables such as A + B. Similarly, OR operation between three variables can be represented as A + B + C.
5. The logical OR operator in the Boolean algebra with variables having value either a 0 or a 1 gives following results.

$0 + 0 = 0$ $\qquad$ $1 + 0 = 1$

$0 + 1 = 1$ $\qquad$ $1 + 1 = 1$

From the above results following rules are defined in the Boolean algebra.

Rule 1 : $\begin{matrix} 0 & + & 0 & = & 0 \\ & & & & \\ 0 & + & 1 & = & 1 \end{matrix} \Rightarrow 0 + A = A \text{ or } A + 0 = A$

Rule 2 :

$$\begin{matrix} 1 + 0 = 1 \\ \\ 1 + 1 = 1 \end{matrix} \Rightarrow 1 + A = 1 \text{ or } A + 1 = 1$$

Rule 3 :

$$\begin{matrix} 0 + 0 = 0 \\ \\ 1 + 1 = 1 \end{matrix} \Rightarrow A + A = A$$

Rule 4 :

$$\begin{matrix} 0 + 1 = 1 \\ \\ 1 + 0 = 1 \end{matrix} \Rightarrow A + \overline{A} = 1 \text{ or } \overline{A} + A = 1$$

6. The logical AND operator in the Boolean algebra with variables having value either a 0 or a 1 gives following results.

$0 \cdot 0 = 0 \quad 1 \cdot 0 = 0$

$0 \cdot 1 = 0 \quad 1 \cdot 1 = 1$

From the above result following rules are defined in the Boolean algebra.

Rule 5 :

$$\begin{matrix} 0 \cdot 0 = 0 \\ \\ 0 \cdot 1 = 0 \end{matrix} \Rightarrow 0 \cdot A = 0 \text{ or } A \cdot 0 = 0$$

Rule 6 :

$$\begin{matrix} 1 \cdot 0 = 0 \\ \\ 1 \cdot 1 = 1 \end{matrix} \Rightarrow 1 \cdot A = A \text{ or } A \cdot 1 = A$$

Rule 7 :

$$\begin{matrix} 0 \cdot 0 = 0 \\ \\ 1 \cdot 1 = 1 \end{matrix} \Rightarrow A \cdot A = A$$

Rule 8 :

$$\begin{matrix} 0 \cdot 1 = 0 \\ \\ 1 \cdot 0 = 0 \end{matrix} \Rightarrow A \cdot \overline{A} = 0 \text{ or } \overline{A} \cdot A = 0$$

7. The NOT operator in the Boolean algebra with variable having value either a 0 or a 1 gives following results.

$$\overline{0} = 1 \quad \overline{\overline{0}} = 0$$

$$\overline{1} = 0 \quad \overline{\overline{1}} = 1$$

From the previous result following rule is defined in Boolean algebra

Rule 9 : $\overline{\overline{0}} = 0$ $\qquad \overline{\overline{A}} = A$

$\overline{\overline{1}} = 1$

Laws of Boolean Algebra

Three of the basic laws of Boolean algebra are the same as in ordinary algebra : the commutative laws, associative laws, and the distributive law.

Commutative Laws

LAW 1 : A + B = B + A : This states that the order in which the variables are ORed makes no difference in the output. The truth tables are identical. Therefore, A OR B is same as B OR A.

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

=

B	A	B + A
0	0	0
0	1	1
1	0	1
1	1	1

Truth table for commutative law for OR gates

LAW 2 : AB = BA : The commutative law of multiplication states that the order in which the variables are ANDed makes no difference in the output. The truth tables are identical. Therefore, A AND B is same as B AND A.

A	B	A B
0	0	0
0	1	0
1	0	0
1	1	1

=

B	A	B A
0	0	0
0	1	0
1	0	0
1	1	1

Truth table for commutative law for AND gates

It is important to note that the commutative laws can be extended to any number of variables. For example, since A + B = B + A, it follows that A + B + C = B+A + C , and since A + C = C + A, it is true that B + A + C = B + C + A. Similarly, ABCD = BACD = BADC = ABDC, and so on.

Associative Laws

LAW 1 : A + (B + C) = (A + B) + C : This law states that in the ORing of several variables, the result is the same regardless of the grouping of the variables. For three variables, A OR B ORed with C is the same as A ORed with B OR C.

A	B	C	A + B	(A + B) + C
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

=

A	B	C	B + C	A + (B + C)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Truth tables for associative law for OR gates

LAW 2 : (AB) C = A (BC) : The associative law of multiplication states that it makes no difference in what order the variables are grouped when ANDing several variables. For three variables, A AND B ANDed with C is the same as A ANDed with B and C.

A	B	C	AB	(AB) C
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

=

A	B	C	BC	A (BC)
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Truth table for associative law for AND gates

UNIT-II

GATE LEVEL MINIMIZATION

Karnaugh Maps

Karnaugh maps provide a systematic method to obtain simplified sum-of-products (SOPs) Boolean expressions. This is a compact way of representing a truth table and is a technique that is used to simplify logic expressions. It is ideally suited for four or less variables, becoming cumbersome for five or more variables. Each square represents either a minterm

or maxterm. A K-map of n variables will have 2 squares. For a Boolean expression, product terms are denoted by 1's, while sum terms are denoted by 0's - but 0's are often left blank.

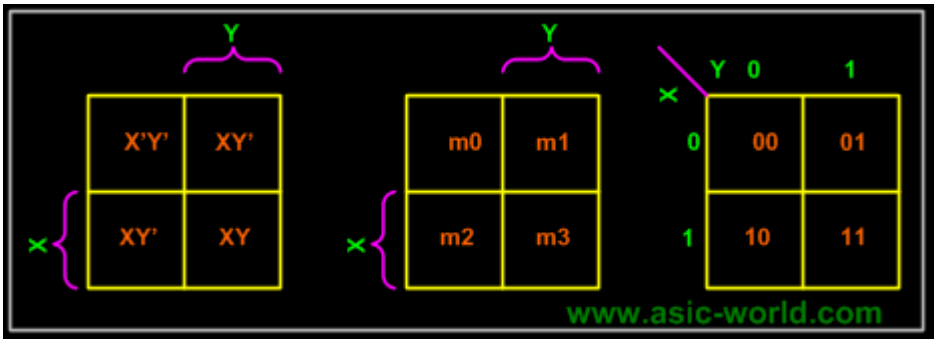
A K-map consists of a grid of squares, each square representing one canonical minterm combination of the variables or their inverse. The map is arranged so that squares representing minterms which differ by only one variable are adjacent both vertically and horizontally. Therefore XY'Z' would be adjacent to X'Y'Z' and would also adjacent to XY'Z and XYZ'.

Minimization Technique

- Based on the Unifying Theorem: X + X' = 1
- The expression to be minimized should generally be in sum-of-product form (If necessary, the conversion process is applied to create the sum-of-product form).
- The function is mapped onto the K-map by marking a 1 in those squares corresponding to the terms in the expression to be simplified (The other squares may be filled with 0's).
- Pairs of 1's on the map which are adjacent are combined using the theorem Y(X+X') = Y where Y is any Boolean expression (If two pairs are also adjacent, then these can also be combined using the same theorem).
- The minimization procedure consists of recognizing those pairs and multiple pairs.
    - These are circled indicating reduced terms.
    - Groups which can be circled are those which have two (2^1) 1's, four (2^2) 1's, eight (2^3) 1's, and so on.
    - Note that because squares on one edge of the map are considered adjacent to those on the opposite edge, group can be formed with these squares.
    - Groups are allowed to overlap.
- The objective is to cover all the 1's on the map in the fewest number of groups and to create the largest groups to do this.
- Once all possible groups have been formed, the corresponding terms are identified.
    - A group of two 1's eliminates one variable from the original minterm.
    - A group of four 1's eliminates two variables from the original minterm.
    - A group of eight 1's eliminates three variables from the original minterm, and so on.
    - The variables eliminated are those which are different in the original minterms of the group.

2-Variable K-Map

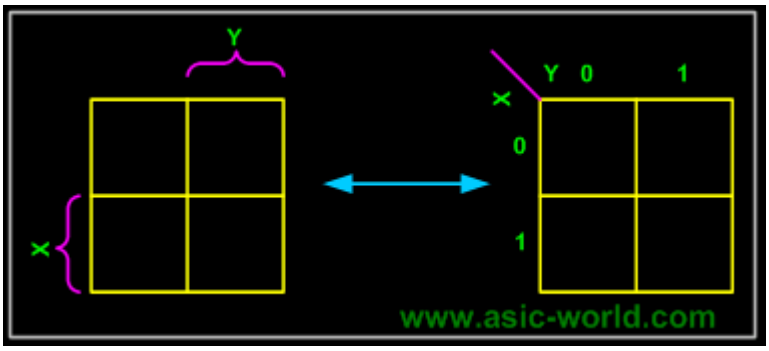
In any K-Map, each square represents a minterm. Adjacent squares always differ by just one literal (So that the unifying theorem may apply: X + X' = 1). For the 2-variable case (e.g.: variables X, Y), the map can be drawn as below. Two variable map is the one which has got only two variables as input.



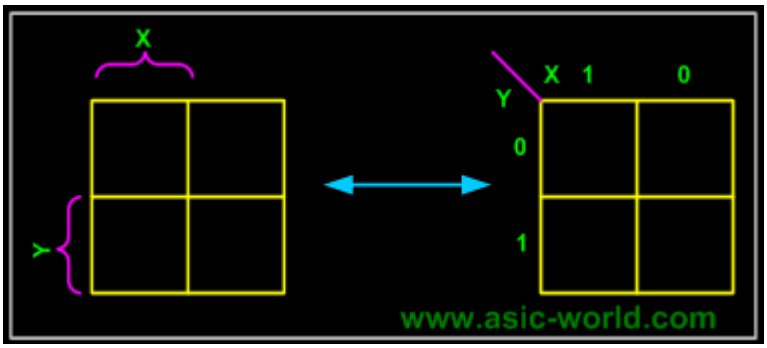
Equivalent labeling

K-map needs not follow the ordering as shown in the figure above. What this means is that we can change the position of m0, m1, m2, m3 of the above figure as shown in the two figures below.

Position assignment is the same as the default k-maps positions. This is the one which we will be using throughout this tutorial.



This figure is with changed position of m0, m1, m2, m3.



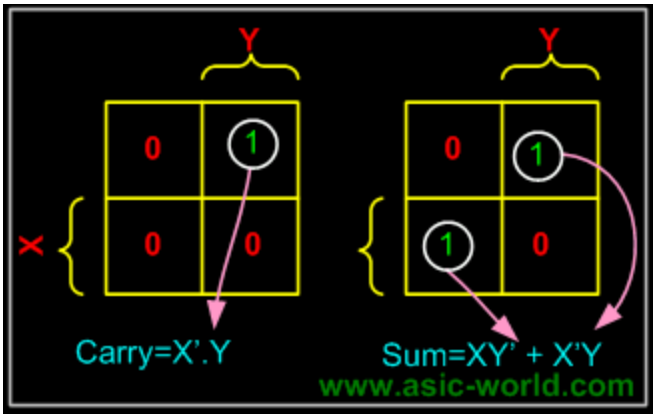
The K-map for a function is specified by putting a '1' in the square corresponding to a minterm, a '0' otherwise.

Example- Carry and Sum of a half adder

In this example we have the truth table as input, and we have two output functions. Generally we may have n output functions for m input variables. Since we have two output

functions, we need to draw two k-maps (i.e. one for each function). Truth table of 1 bit adder is shown below. Draw the k-map for Carry and Sum as shown below.

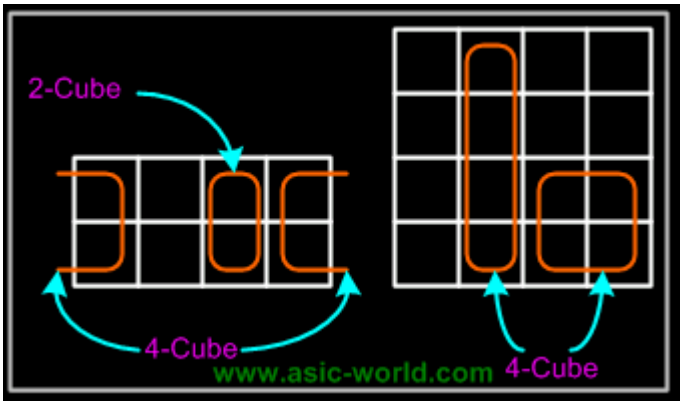
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Grouping/Circling K-maps

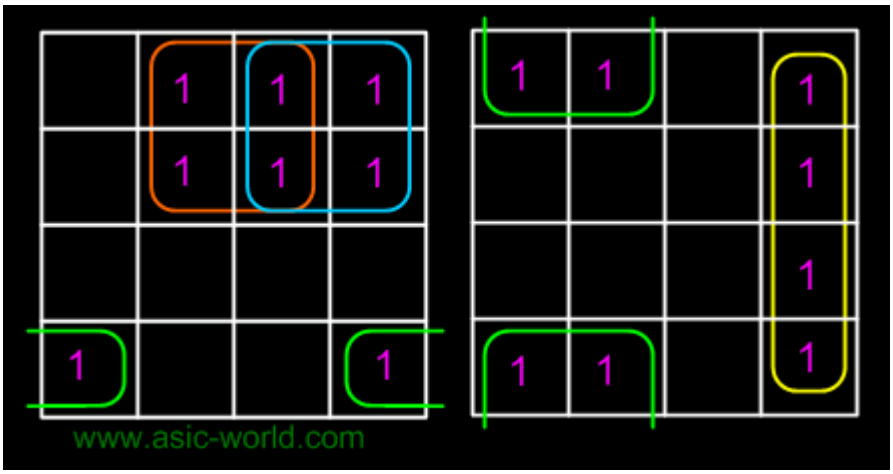
The power of K-maps is in minimizing the terms, K-maps can be minimized with the help of grouping the terms to form single terms. When forming groups of squares, observe/consider the following:

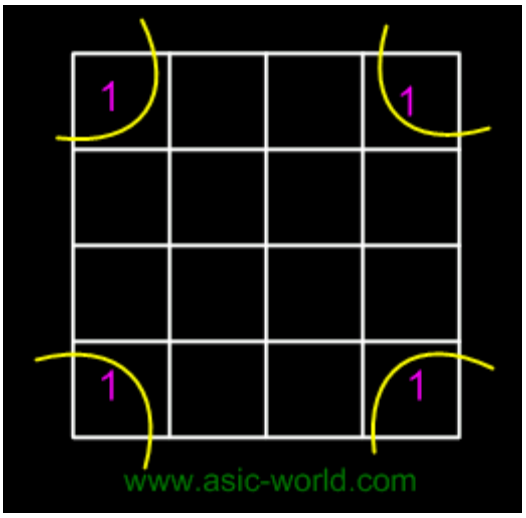
- Every square containing 1 must be considered at least once.
- A square containing 1 can be included in as many groups as desired.
- A group must be as large as possible.



- If a square containing 1 cannot be placed in a group, then leave it out to include in final expression.
- The number of squares in a group must be equal to 2
- , i.e. 2,4,8,.

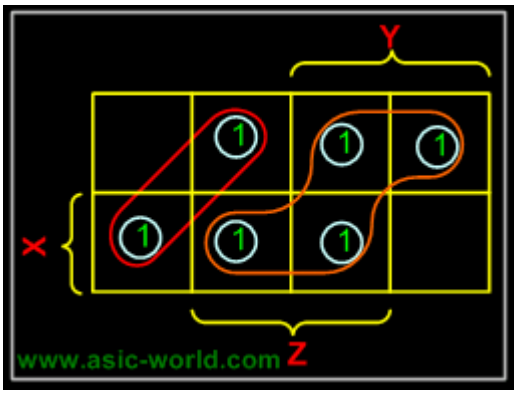
- The map is considered to be folded or spherical, therefore squares at the end of a row or column are treated as adjacent squares.
- The simplified logic expression obtained from a K-map is not always unique. Groupings can be made in different ways.
- Before drawing a K-map the logic expression must be in canonical form.





In the next few pages we will see some examples on grouping.

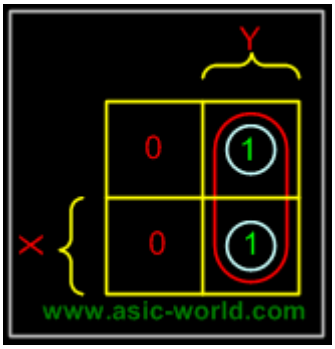
Example of invalid groups



Example - X'Y+XY

In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y and XY position. Now combine two 1's as shown in figure to form the single term. As you can see X and X' get canceled and only Y remains.

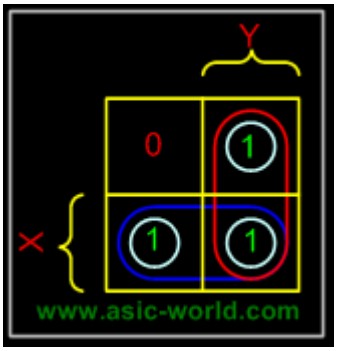
F = Y



Example - X'Y+XY+XY'

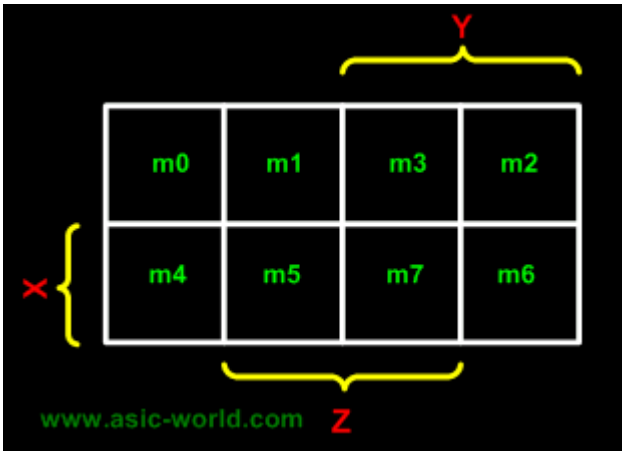
In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y, XY and XY position. Now combine two 1's as shown in figure to form the two single terms.

F = X + Y



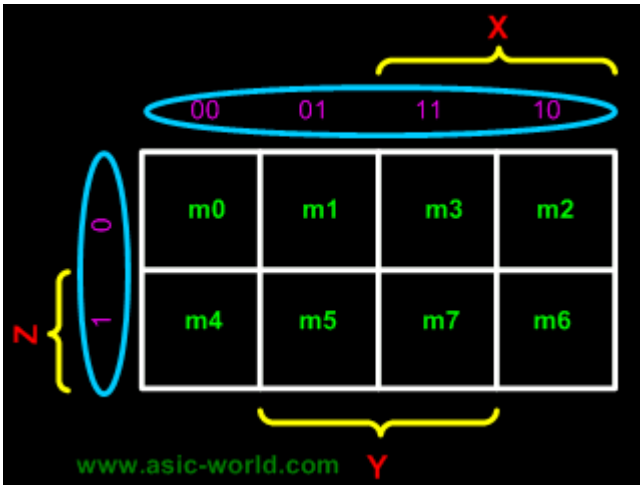
3-Variable K-Map

There are 8 minterms for 3 variables (X, Y, Z). Therefore, there are 8 cells in a 3-variable K-map. One important thing to note is that K-maps follow the gray code sequence, not the binary one.



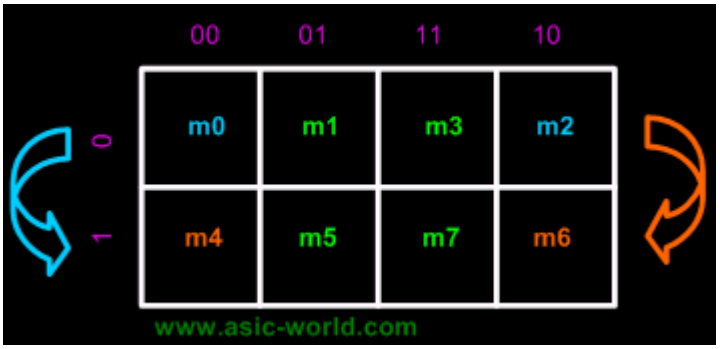
Using gray code arrangement ensures that minterms of adjacent cells differ by only ONE literal. (Other arrangements which satisfy this criterion may also be used.)

Each cell in a 3-variable K-map has 3 adjacent neighbours. In general, each cell in an n-variable K-map has n adjacent neighbours.



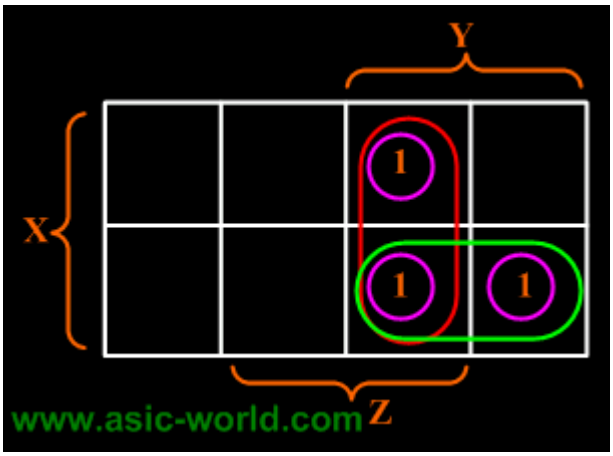
There is wrap-around in the K-map

- X'Y'Z' (m0) is adjacent to X'YZ' (m2)
- XY'Z' (m4) is adjacent to XYZ' (m6)



Example

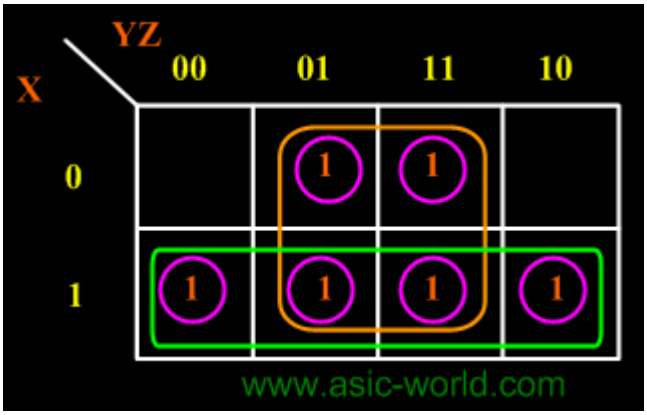
F = XYZ'+XYZ+X'YZ



F = XY + YZ

Example

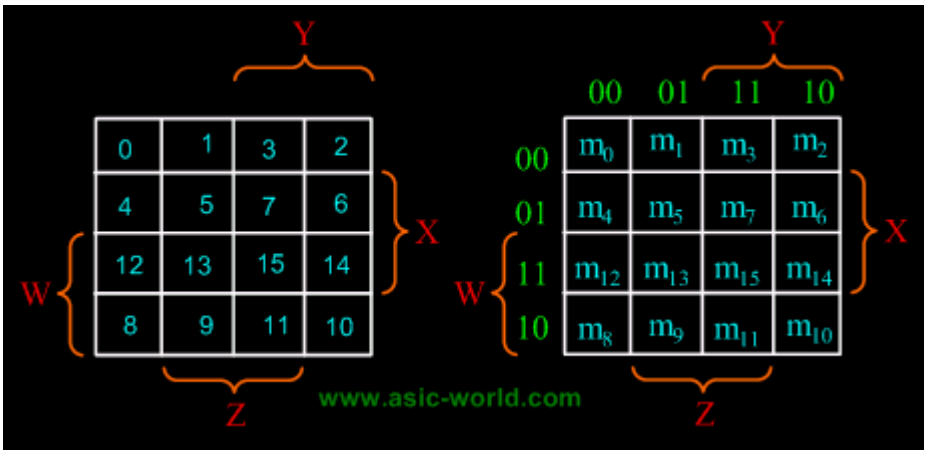
F(X,Y,Z) = Σ(1,3,4,5,6,7)



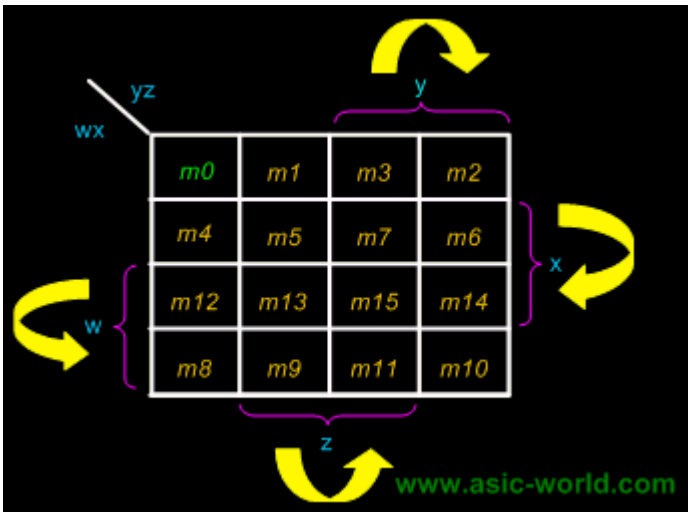
F = X + Z

4-Variable K-Map

There are 16 cells in a 4-variable (W, X, Y, Z); K-map as shown in the figure below.

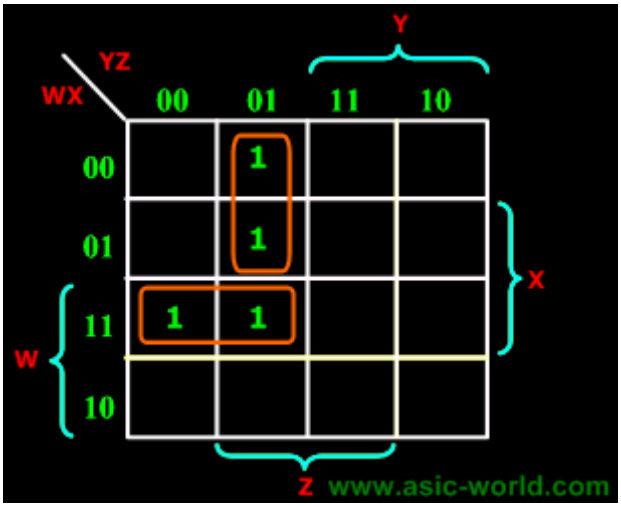


There are 2 wrap-around: a horizontal wrap-around and a vertical wrap-around. Every cell thus has 4 neighbours. For example, the cell corresponding to minterm m0 has neighbours m1, m2, m4 and m8.



Example

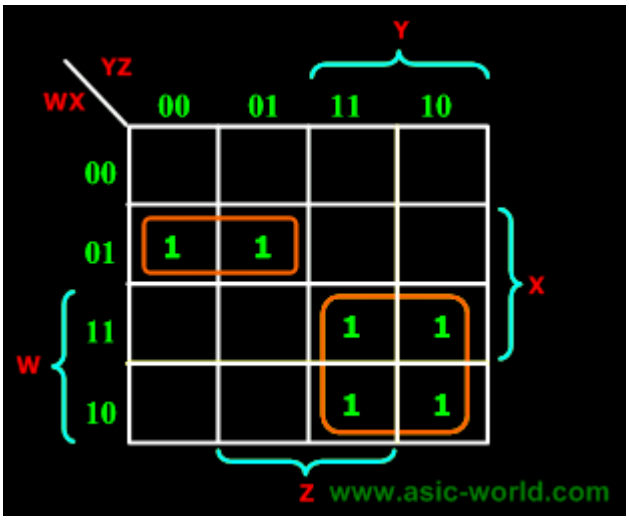
F(W,X,Y,Z) = (1,5,12,13)



F = WY'Z + W'Y'Z

Example

F(W,X,Y,Z) = (4, 5, 10, 11, 14, 15)



F = W'XY' + WY

QUINE-McCLUSKEY MINIMIZATION

Quine-McCluskey minimization method uses the same theorem to produce the solution as the K-map method, namely X(Y+Y')=X

Minimization Technique

- The expression is represented in the canonical SOP form if not already in that form.
- The function is converted into numeric notation.
- The numbers are converted into binary form.
- The minterms are arranged in a column divided into groups.
- Begin with the minimization procedure.
    - Each minterm of one group is compared with each minterm in the group immediately below.
    - Each time a number is found in one group which is the same as a number in the group below except for one digit, the numbers pair is ticked and a new composite is created.
    - This composite number has the same number of digits as the numbers in the pair except the digit different which is replaced by an "x".
- The above procedure is repeated on the second column to generate a third column.
- The next step is to identify the essential prime implicants, which can be done using a prime implicant chart.
    - Where a prime implicant covers a minterm, the intersection of the corresponding row and column is marked with a cross.
    - Those columns with only one cross identify the essential prime implicants. -> These prime implicants must be in the final answer.
    - The single crosses on a column are circled and all the crosses on the same row are also circled, indicating that these crosses are covered by the prime implicants selected.
    - Once one cross on a column is circled, all the crosses on that column can be circled since the minterm is now covered.
    - If any non-essential prime implicant has all its crosses circled, the prime implicant is redundant and need not be considered further.
- Next, a selection must be made from the remaining nonessential prime implicants, by considering how the non-circled crosses can be covered best.
    - One generally would take those prime implicants which cover the greatest number of crosses on their row.
    - If all the crosses in one row also occur on another row which includes further crosses, then the latter is said to dominate the former and can be selected.
    - The dominated prime implicant can then be deleted.

Example

Find the minimal sum of products for the Boolean expression, f= Σ (1,2,3,7,8,9,10,11,14,15), using Quine-McCluskey method.

Firstly these minterms are represented in the binary form as shown in the table below. The above binary representations are grouped into a number of sections in terms of the number of 1's as shown in the table below.

Binary representation of minterms

Minterms	U	V	W	X
1	0	0	0	1
2	0	0	1	0

Group of minterms for different number of 1's

Any two numbers in these groups which differ from each other by only one variable can be chosen and combined, to get 2-cell combination, as shown in the table below.

2-Cell combinations

Combinations	U	V	W	X
(1,3)	0	0	-	1
(1,9)	-	0	0	1
(2,3)	0	0	1	-
(2,10)	-	0	1	0
(8,9)	1	0	0	-
(8,10)	1	0	-	0
(3,7)	0	-	1	1
(3,11)	-	0	1	1
(9,11)	1	0	-	1
(10,11)	1	0	1	-
(10,14)	1	-	1	0

(7,15)	-	1	1	1
(11,15)	1	-	1	1
(14,15)	1	1	1	-

From the 2-cell combinations, one variable and dash in the same position can be combined to form 4-cell combinations as shown in the figure below.

4-Cell combinations

Combinations	U	V	W	X
(1,3,9,11)	-	0	-	1
(2,3,10,11)	-	0	1	-
(8,9,10,11)	1	0	-	-
(3,7,11,15)	-	-	1	1
(10,11,14,15)	1	-	1	-

The cells (1,3) and (9,11) form the same 4-cell combination as the cells (1,9) and (3,11). The order in which the cells are placed in a combination does not have any effect. Thus the (1,3,9,11) combination could be written as (1,9,3,11).

From above 4-cell combination table, the prime implicants table can be plotted as shown in table below.

Prime Implicants Table

Prime Implicants	1	2	3	7	8	9	10	11	14	15
(1,3,9,11)	X	-	X	-	-	X	-	X	-	-
(2,3,10,11)	-	X	X	-	-	-	X	X	-	-
(8,9,10,11)	-	-	-	-	X	X	X	X	-	-
(3,7,11,15)	-	-	-	-	-	-	X	X	X	X
-	X	X	-	X	X	-	-	-	X	-

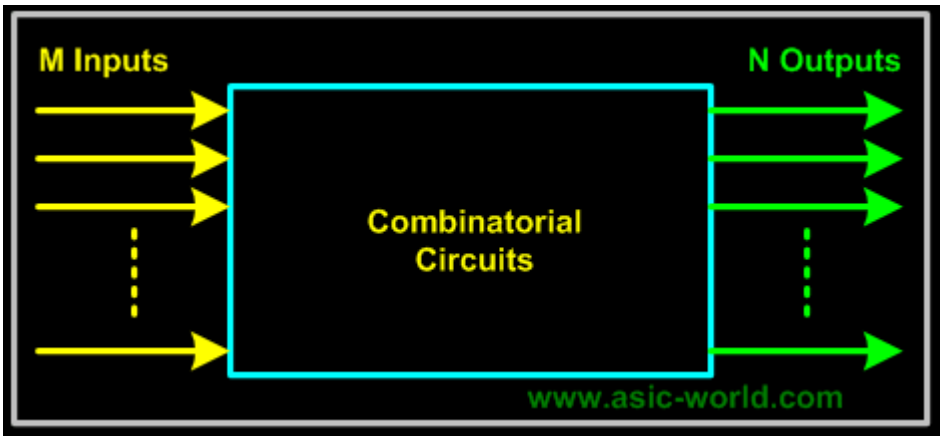
The columns having only one cross mark correspond to essential prime implicants. A yellow cross is used against every essential prime implicant. The prime implicants sum gives the function in its minimal SOP form.

Y = V'X + V'W + UV' + WX + UW

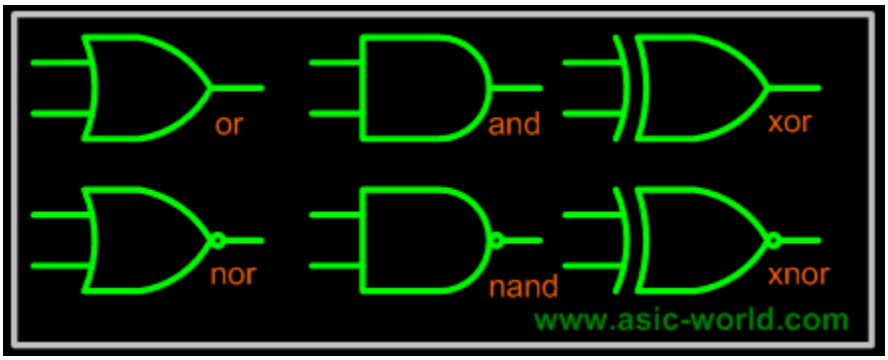
Combinational Logic

Introduction

Combinatorial Circuits are circuits which can be considered to have the following generic structure.



Whenever the same set of inputs is fed in to a combinatorial circuit, the same outputs will be generated. Such circuits are said to be stateless. Some simple combinational logic elements that we have seen in previous sections are "Gates".



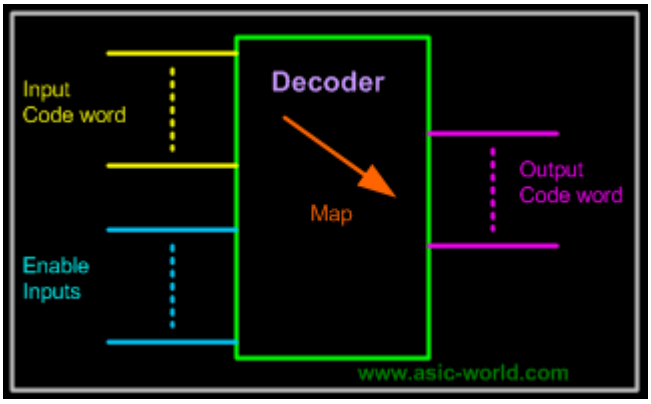
All the gates in the above figure have 2 inputs and one output; combinational elements simplest form are "not" gate and "buffer" as shown in the figure below. They have only one input and one output.

Decoders

A decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different; e.g. n-to-2n, BCD decoders.

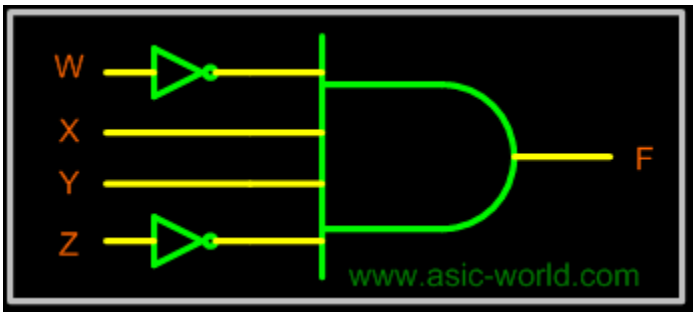
Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output code word.

Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding. Figure below shows the pseudo block of a decoder.



Basic Binary Decoder

And AND gate can be used as the basic decoding element, because its output is HIGH only when all its inputs are HIGH. For example, if the input binary number is 0110, then, to make all the inputs to the AND gate HIGH, the two outer bits must be inverted using two inverters as shown in figure below.



Binary n-to-2^n Decoders

A binary decoder has n inputs and 2^n outputs. Only one output is active at any one time, corresponding to the input value. Figure below shows a representation of Binary n-to-2^n decoder

Example - 2-to-4 Binary Decoder

A 2 to 4 decoder consists of two inputs and four outputs, truth table and symbols of which is shown below.

Truth Table

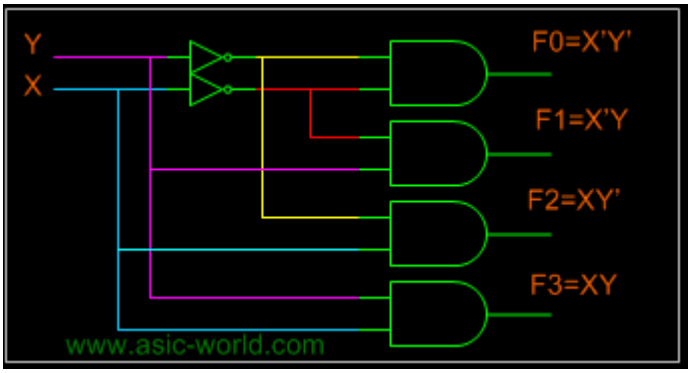
X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Symbol

To minimize the above truth table we may use kmap, but doing that you will realize that it is a waste of time. One can directly write down the function for each of the outputs. Thus we can draw the circuit as shown in figure below.

Note: Each output is a 2-variable minterm (X'Y', X'Y, XY', XY)

Circuit



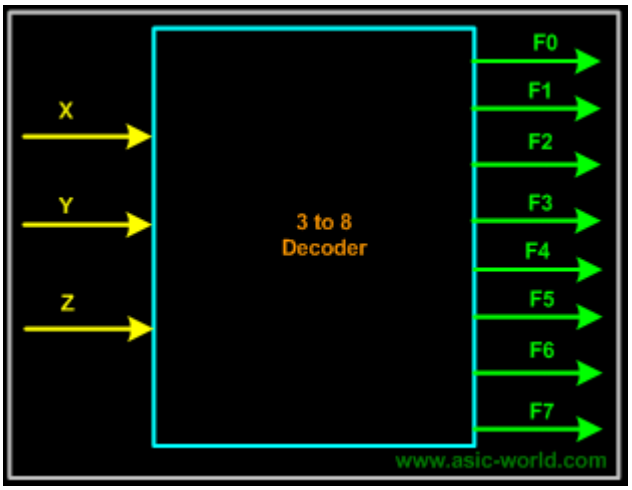
Example - 3-to-8 Binary Decoder

A 3 to 8 decoder consists of three inputs and eight outputs, truth table and symbols of which is shown below.

Truth Table

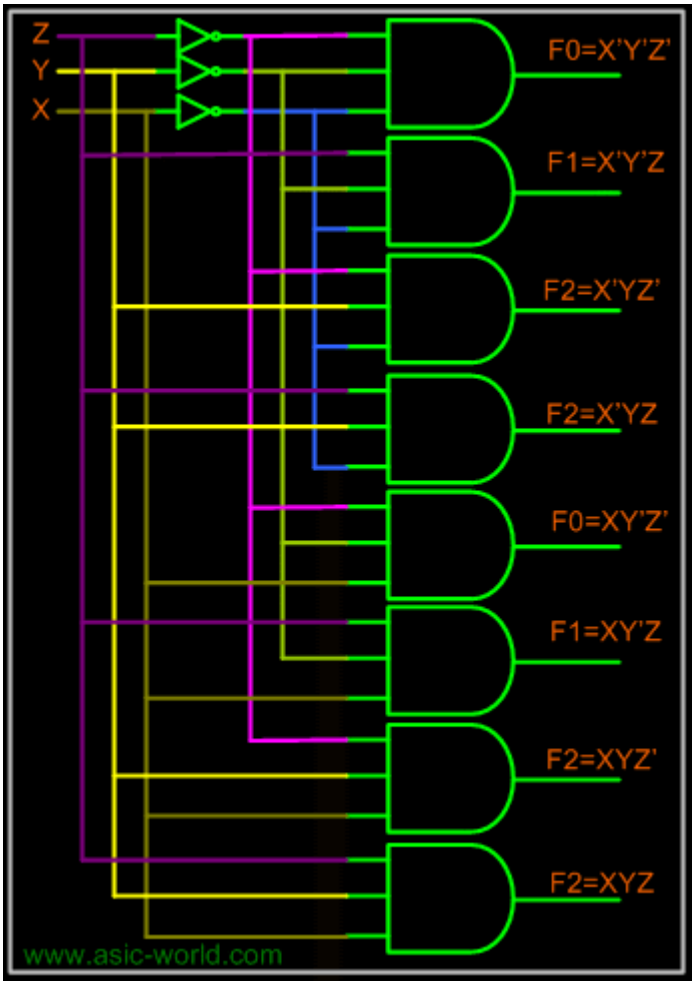
X	Y	Z	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Symbol



From the truth table we can draw the circuit diagram as shown in figure below.

Circuit



Implementing Functions Using Decoders

- Any n-variable logic function, in canonical sum-of-minterms form can be implemented using a single n-to-2^n decoder to generate the minterms, and an OR gate to form the sum.
    - The output lines of the decoder corresponding to the minterms of the function are used as inputs to the or gate.
- Any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n decoder with m OR gates.
- Suitable when a circuit has many outputs, and each output function is expressed with few minterms.

Example - Full adder

Equation

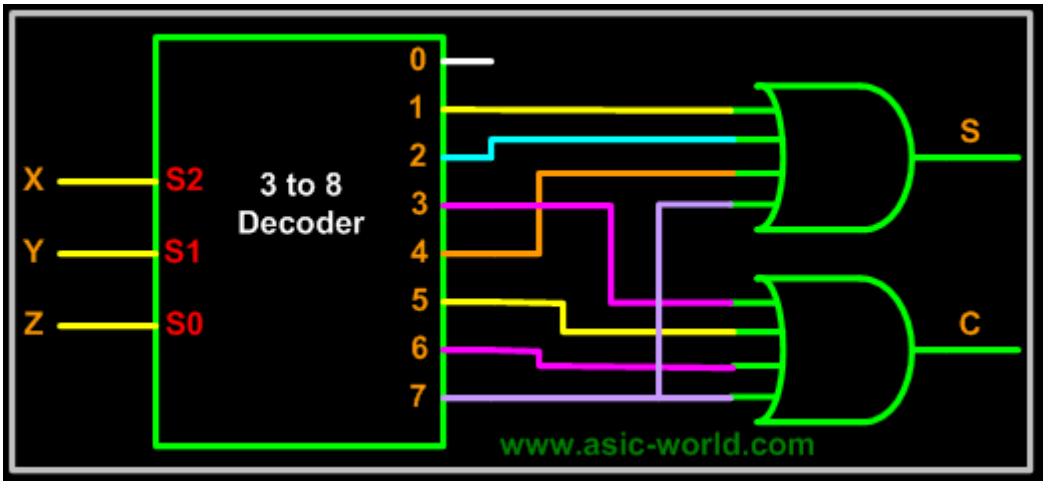
$S(x, y, z) = \Sigma(1,2,4,7)$
$C(x, y, z) = \Sigma(3,5,6,7)$

Truth Table

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

From the truth table we know the values for which the sum (s) is active and also the carry (c) is active. Thus we have the equation as shown above and a circuit can be drawn as shown below from the equation derived.

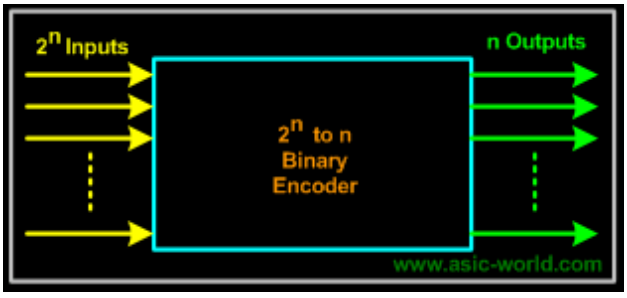
Circuit



Encoders

An encoder is a combinational circuit that performs the inverse operation of a decoder. If a device output code has fewer bits than the input code has, the device is usually called an encoder. e.g. 2^n-to-n, priority encoders.

The simplest encoder is a 2^n-to-n binary encoder, where it has only one of 2^n inputs = 1 and the output is the n-bit binary number corresponding to the active input.



Example - Octal-to-Binary Encoder

Octal-to-Binary take 8 inputs and provides 3 outputs, thus doing the opposite of what the 3-to-8 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of an Octal-to-binary encoder.

Truth Table

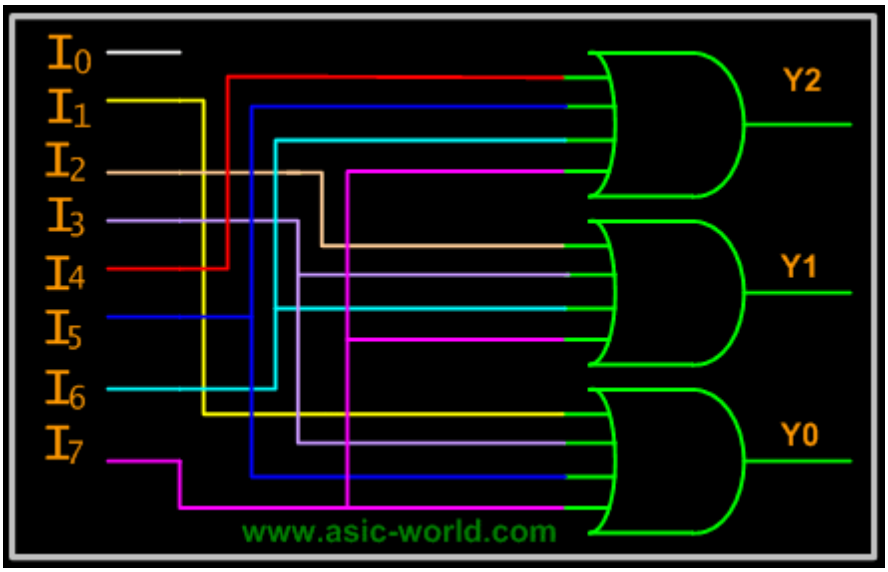
I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

Y0 = I1 + I3 + I5 + I7
Y1= I2 + I3 + I6 + I7
Y2 = I4 + I5 + I6 +I7

Based on the above equations, we can draw the circuit as shown below

Circuit



Example - Decimal-to-Binary Encoder

Decimal-to-Binary take 10 inputs and provides 4 outputs, thus doing the opposite of what the 4-to-10 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of a Decimal-to-binary encoder.

Truth Table

I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	Y3	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

From the above truth table , we can derive the functions Y3, Y2, Y1 and Y0 as given below.

Y3 = I8 + I9
Y2 = I4 + I5 + I6 + I7
Y1 = I2 + I3 + I6 + I7
Y0 = I1 + I3 + I5 + I7 + I9

Priority Encoder

If we look carefully at the Encoder circuits that we got, we see the following limitations. If more then two inputs are active simultaneously, the output is unpredictable or rather it is not what we expect it to be.

This ambiguity is resolved if priority is established so that only one input is encoded, no matter how many inputs are active at a given point of time.

The priority encoder includes a priority function. The operation of the priority encoder is such that if two or more inputs are active at the same time, the input having the highest priority will take precedence.

Example - 4to3 Priority Encoder

The truth table of a 4-input priority encoder is as shown below. The input D3 has the highest priority, D2 has next highest priority, D0 has the lowest priority. This means output Y2 and Y1 are 0 only when none of the inputs D1, D2, D3 are high and only D0 is high.

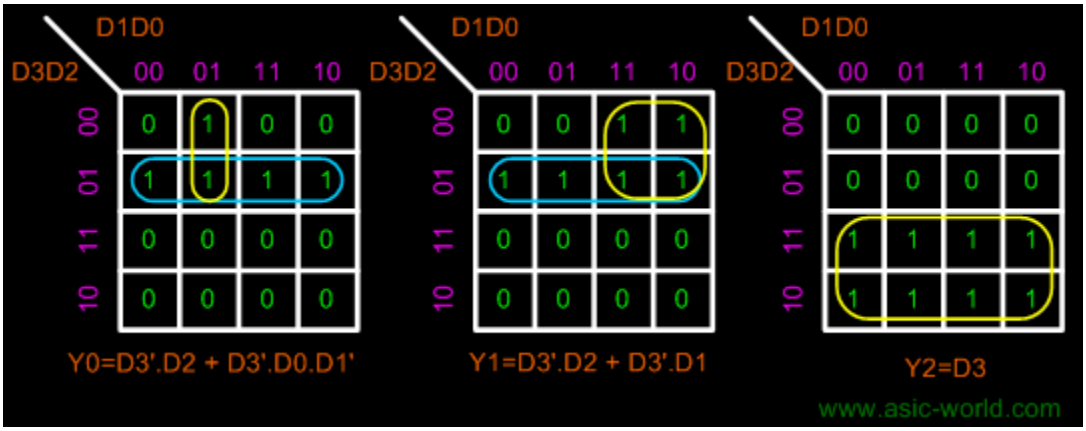
A 4 to 3 encoder consists of four inputs and three outputs, truth table and symbols of which is shown below.

Truth Table

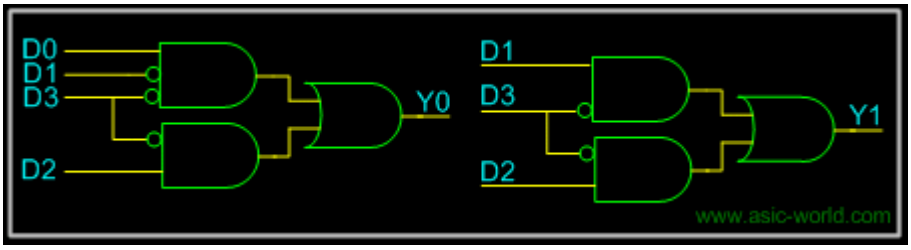
D3	D2	D1	D0	Y2	Y1	Y0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	0
0	1	x	x	0	1	1
1	x	x	x	1	0	0

Now that we have the truth table, we can draw the Kmaps as shown below.

Kmaps



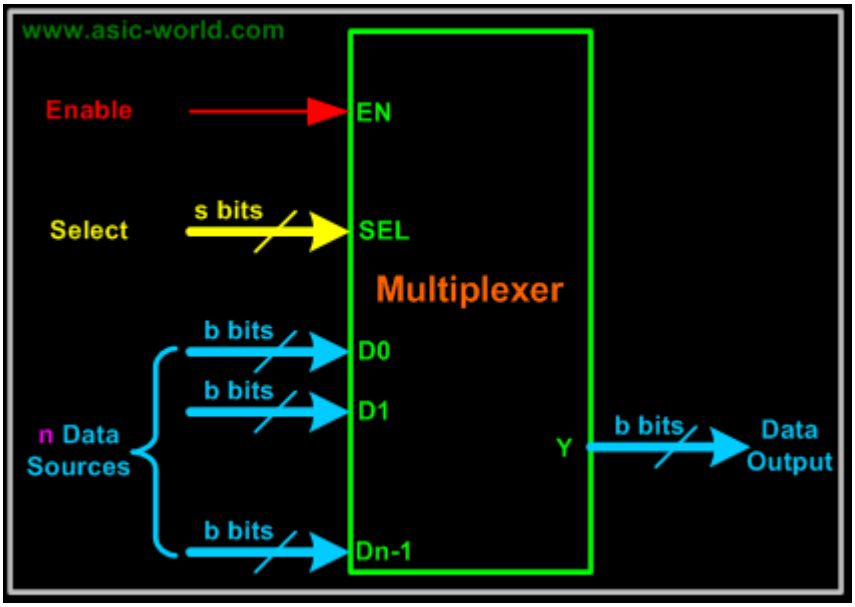
From the Kmap we can draw the circuit as shown below. For Y2, we connect directly to D3.



We can apply the same logic to get higher order priority encoders.

Multiplexer

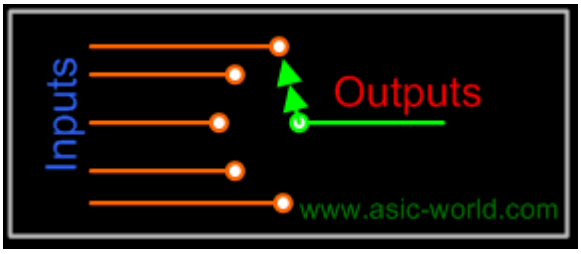
A multiplexer (MUX) is a digital switch which connects data from one of n sources to the output. A number of select inputs determine which data source is connected to the output. The block diagram of MUX with n data sources of b bits wide and s bits wide select line is shown in below figure.



MUX acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the input source that will be switched on to the output as shown in the figure below. At any given point of time only one input gets selected and is connected to output, based on the select input signal.

Mechanical Equivalent of a Multiplexer

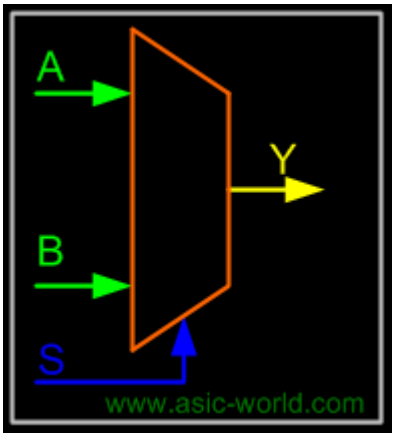
The operation of a multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the inputs, which is connected to the output. As you can see at any given point of time only one input gets transferred to output.



Example - 2x1 MUX

A 2 to 1 line multiplexer is shown in figure below, each 2 input lines A to B is applied to one input of an AND gate. Selection lines S are decoded to select a particular AND gate. The truth table for the 2:1 mux is given in the table below.

Symbol



Truth Table

S	Y
0	A
1	B

Design of a 2:1 Mux

To derive the gate level implementation of 2:1 mux we need to have truth table as shown in figure. And once we have the truth table, we can draw the K-map as shown in figure for all the cases when Y is equal to '1'.

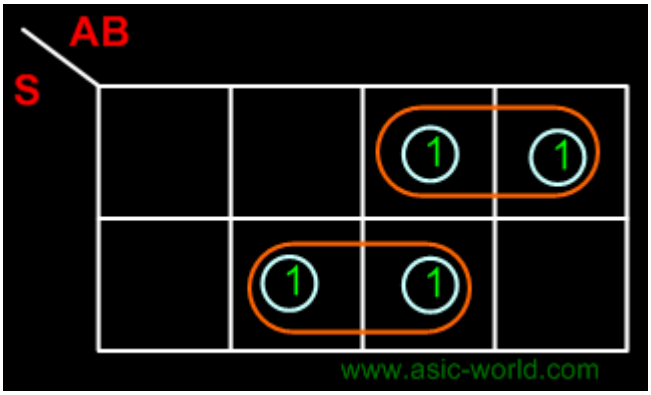
Combining the two 1' as shown in figure, we can drive the output y as shown below

Y = A.S' + B.S

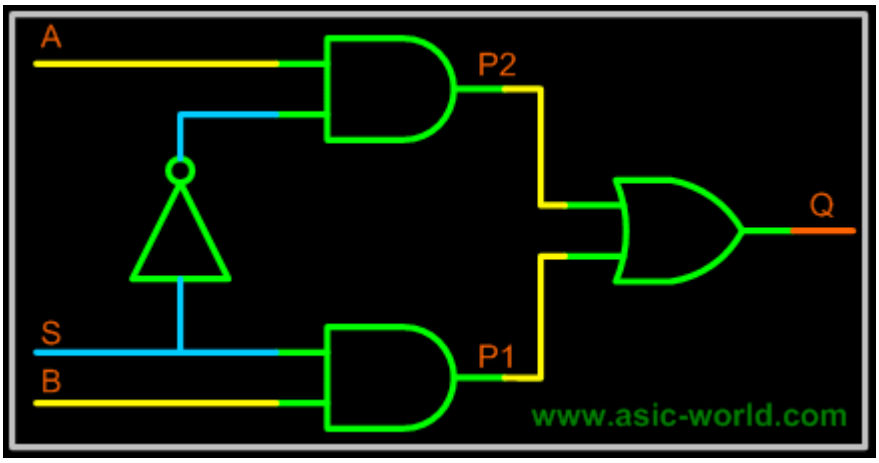
Truth Table

B	A	S	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Kmap



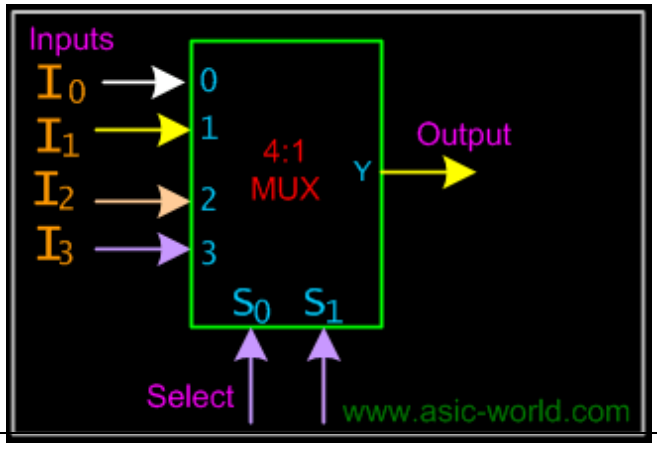
Circuit



Example : 4:1 MUX

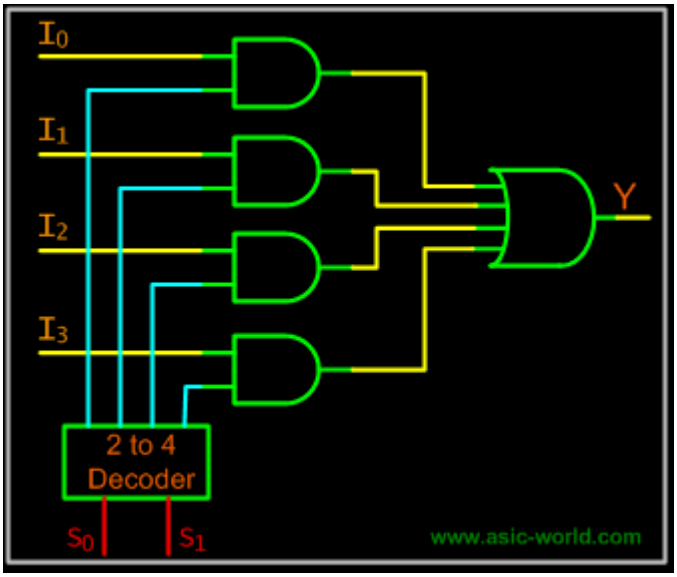
A 4 to 1 line multiplexer is shown in figure below, each of 4 input lines I0 to I3 is applied to one input of an AND gate. Selection lines S0 and S1 are decoded to select a particular AND gate. The truth table for the 4:1 mux is given in the table below.

Symbol



Truth Table

Circuit



Larger Multiplexers

Larger multiplexers can be constructed from smaller ones. An 8-to-1 multiplexer can be constructed from smaller multiplexers as shown below.

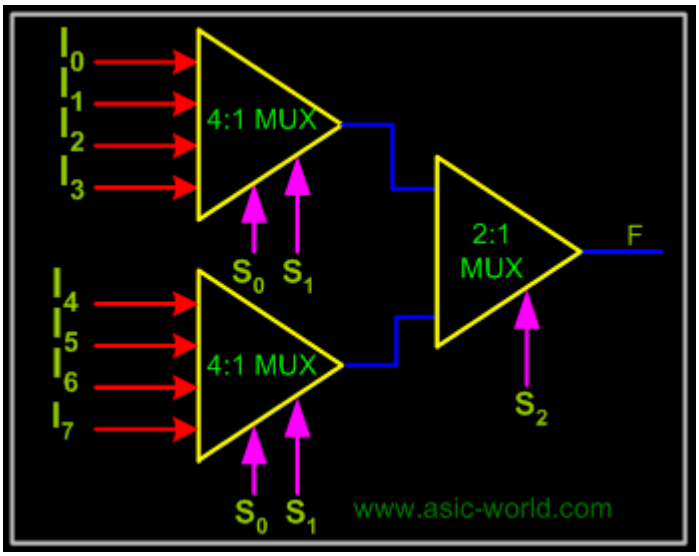
Example - 8-to-1 multiplexer from Smaller MUX

Truth Table

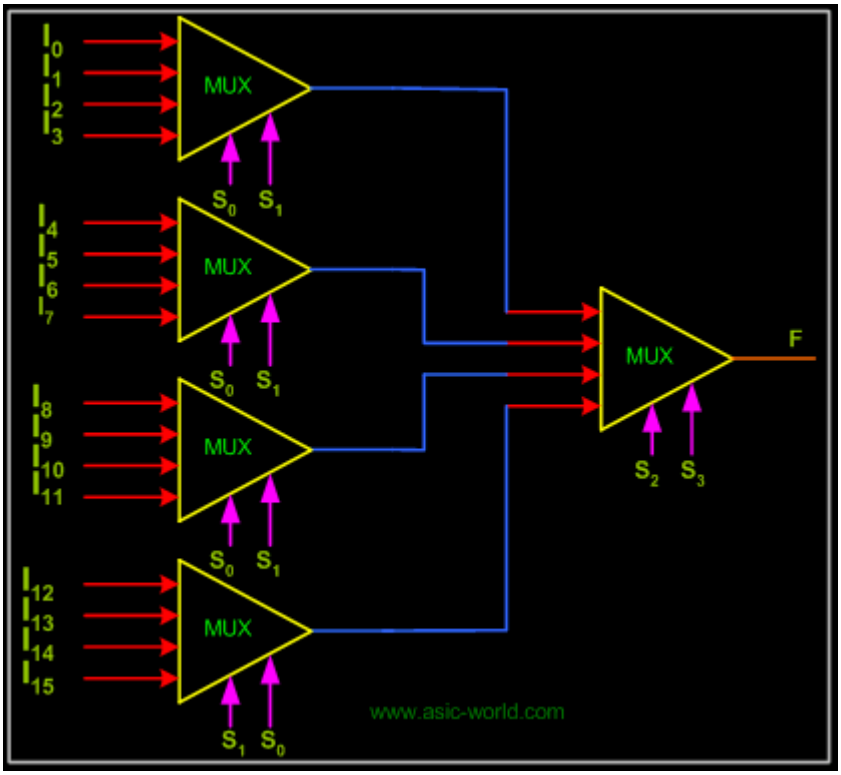
S2	S1	S0	F
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6

1	1	1	I7

Circuit



Example - 16-to-1 multiplexer from 4:1 mux

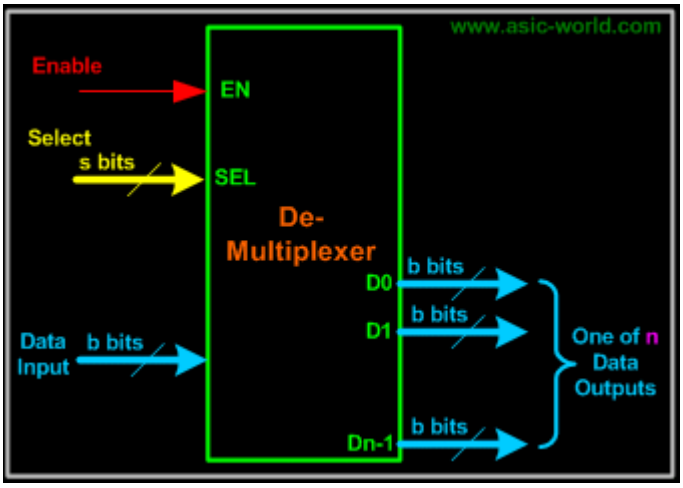


De-multiplexers

They are digital switches which connect data from one input source to one of n outputs.

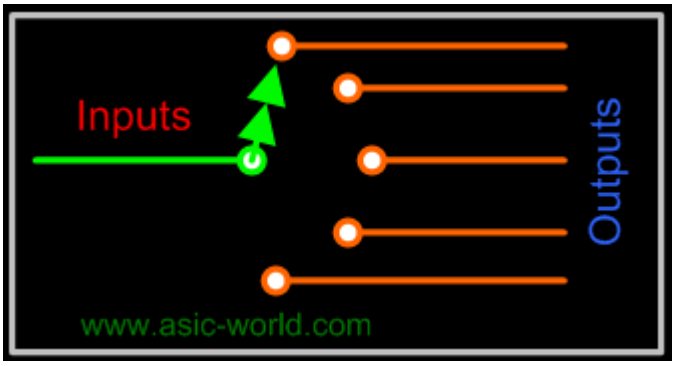
Usually implemented by using n-to-2^n binary decoders where the decoder enable line is used for data input of the de-multiplexer.

The figure below shows a de-multiplexer block diagram which has got s-bits-wide select input, one b-bits-wide data input and n b-bits-wide outputs.

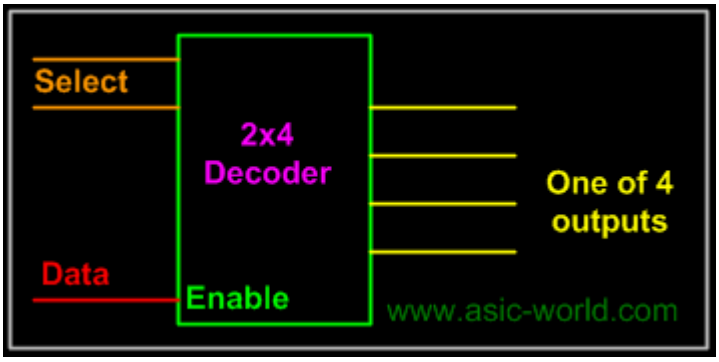


Mechanical Equivalent of a De-Multiplexer

The operation of a de-multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the outputs, which is connected to the input. As you can see at any given point of time only one output gets connected to input.

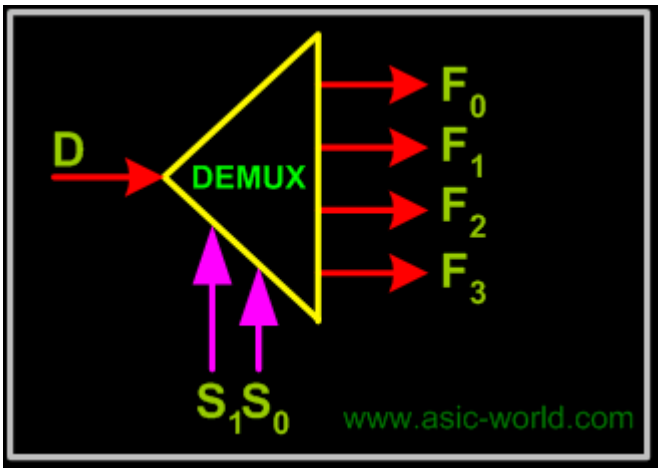


1-bit 4-output de-multiplexer using a 2x4 binary decoder.



Example: 1-to-4 De-multiplexer

Symbol



Truth Table

S1	S0	F0	F1	F2	F3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Boolean Function Implementation

Earlier we had seen that it is possible to implement Boolean functions using decoders. In the same way it is also possible to implement Boolean functions using muxers and de-muxers.

Implementing Functions Multiplexers

Any n-variable logic function can be implemented using a smaller 2^{n-1}-to-1 multiplexer and a single inverter (e.g 4-to-1 mux to implement 3 variable functions) as follows.

Express function in canonical sum-of-minterms form. Choose n-1 variables as inputs to mux select lines. Construct the truth table for the function, but grouping inputs by selection line values (i.e select lines as most significant inputs).

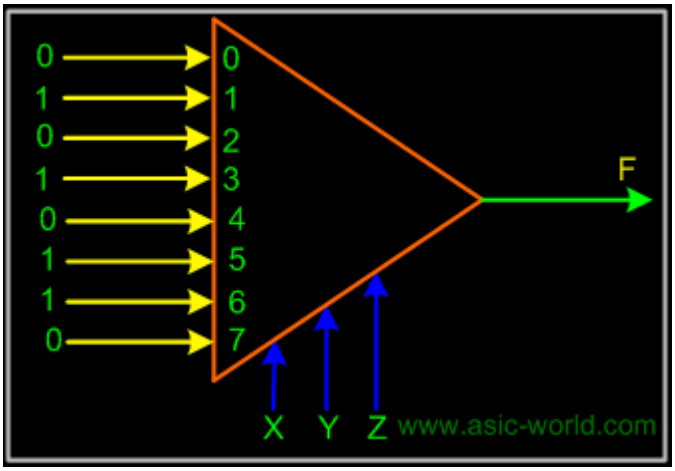
Determine multiplexer input line i values by comparing the remaining input variable and the function F for the corresponding selection lines value i.

We have four possible mux input line i values:

- Connect to 0 if the function is 0 for both values of remaining variable.
- Connect to 1 if the function is 1 for both values of remaining variable.
- Connect to remaining variable if function is equal to the remaining variable.
- Connect to the inverted remaining variable if the function is equal to the remaining variable inverted.

Example: 3-variable Function Using 8-to-1 mux

Implement the function F(X,Y,Z) = S(1,3,5,6) using an 8-to-1 mux. Connect the input variables X, Y, Z to mux select lines. Mux data input lines 1, 3, 5, 6 that correspond to the function minterms are connected to 1. The remaining mux data input lines 0, 2, 4, 7 are connected to 0.



Example: 3-variable Function Using 4-to-1 mux

Implement the function F(X,Y,Z) = S(0,1,3,6) using a single 4-to-1 mux and an inverter. We choose the two most significant inputs X, Y as mux select lines.

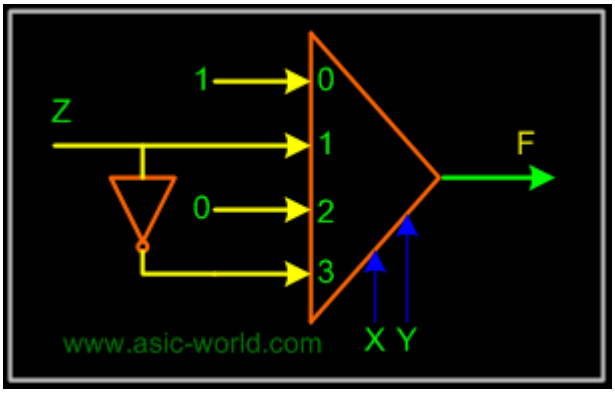
Construct truth table.

Truth Table

Select i	X	Y	Z	F	Mux Input i

0	0	0	0	1	1
0	0	0	1	1	1
1	0	1	0	0	Z
1	0	1	1	1	Z
2	1	0	0	0	0
2	1	0	1	0	0
3	1	1	0	1	Z'
3	1	1	1	0	Z'

Circuit



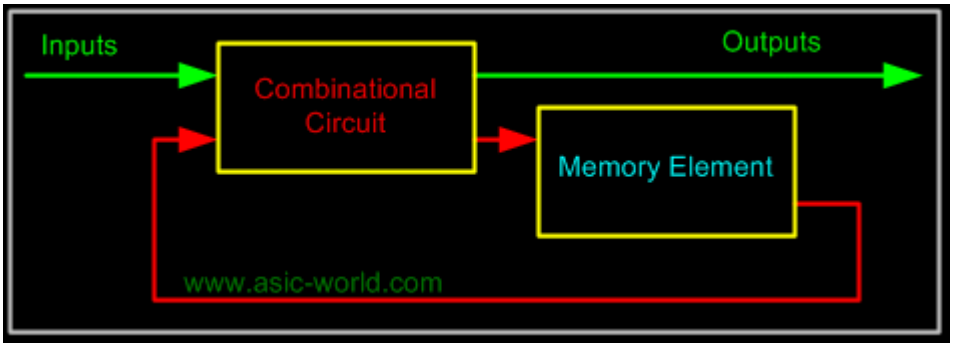
We determine multiplexer input line i values by comparing the remaining input variable Z and the function F for the corresponding selection lines value i

- when XY=00 the function F is 1 (for both Z=0, Z=1) thus mux input0 = 1
- when XY=01 the function F is Z thus mux input1 = Z
- when XY=10 the function F is 0 (for both Z=0, Z=1) thus mux input2 = 0
- when XY=11 the function F is Z' thus mux input3 = Z'

UNIT-III

Introduction

Digital electronics is classified into combinational logic and sequential logic. Combinational logic output depends on the inputs levels, whereas sequential logic output depends on stored levels and also the input levels.



The memory elements are devices capable of storing binary info. The binary info stored in the memory elements at any given time defines the state of the sequential circuit. The input and the present state of the memory element determines the output. Memory elements next state is also a function of external inputs and present state. A sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

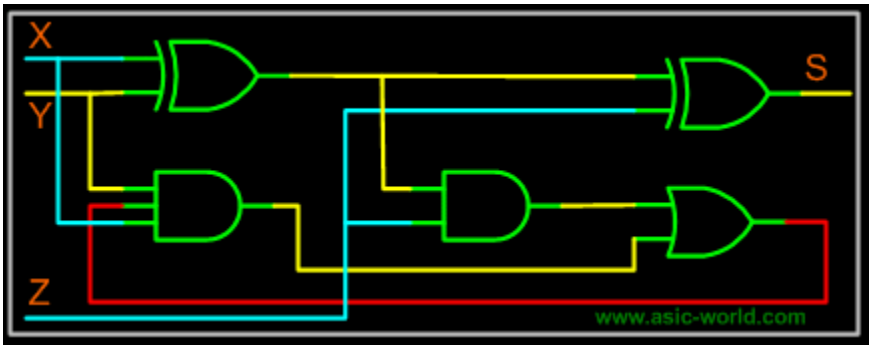
There are two types of sequential circuits. Their classification depends on the timing of their signals:

- Synchronous sequential circuits
- Asynchronous sequential circuits

Asynchronous sequential circuit

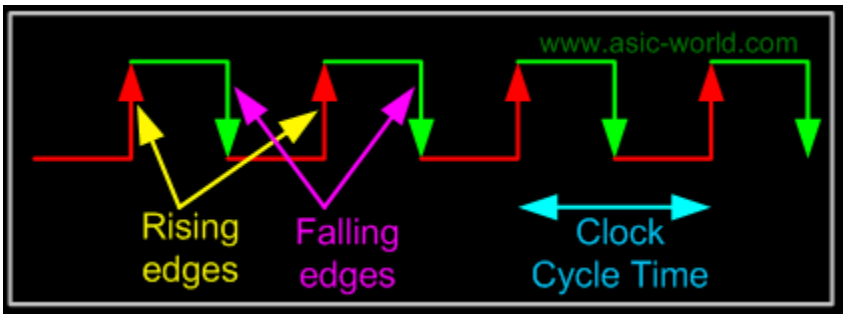
This is a system whose outputs depend upon the order in which its input variables change and can be affected at any instant of time.

Gate-type asynchronous systems are basically combinational circuits with feedback paths. Because of the feedback among logic gates, the system may, at times, become unstable. Consequently they are not often used.



Synchronous sequential circuits

This type of system uses storage elements called flip-flops that are employed to change their binary value only at discrete instants of time. Synchronous sequential circuits use logic gates and flip-flop storage devices. Sequential circuits have a clock signal as one of their inputs. All state transitions in such circuits occur only when the clock value is either 0 or 1 or happen at the rising or falling edges of the clock depending on the type of memory elements used in the circuit. Synchronization is achieved by a timing device called a clock pulse generator. Clock pulses are distributed throughout the system in such a way that the flip-flops are affected only with the arrival of the synchronization pulse. Synchronous sequential circuits that use clock pulses in the inputs are called clocked-sequential circuits. They are stable and their timing can easily be broken down into independent discrete steps, each of which is considered separately.

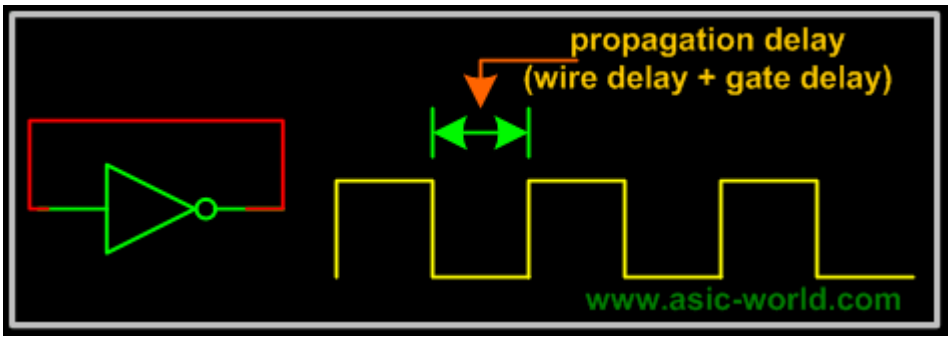


A clock signal is a periodic square wave that indefinitely switches from 0 to 1 and from 1 to 0 at fixed intervals. Clock cycle time or clock period: the time interval between two consecutive rising or falling edges of the clock.

Clock Frequency = 1 / clock cycle time (measured in cycles per second or Hz)

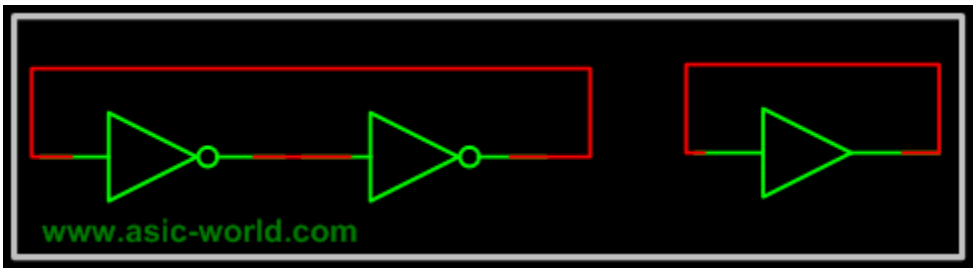
Concept of Sequential Logic

A sequential circuit as seen in the last page, is combinational logic with some feedback to maintain its current value, like a memory cell. To understand the basics let's consider the basic feedback logic circuit below, which is a simple NOT gate whose output is connected to its input. The effect is that output oscillates between HIGH and LOW (i.e. 1 and 0). Oscillation frequency depends on gate delay and wire delay. Assuming a wire delay of 0 and a gate delay of 10ns, then oscillation frequency would be (on time + off time = 20ns) 50Mhz.



The basic idea of having the feedback is to store the value or hold the value, but in the above circuit, output keeps toggling. We can overcome this problem with the circuit below,

which is basically cascading two inverters, so that the feedback is in-phase, thus avoids toggling. The equivalent circuit is the same as having a buffer with its output connected to its input.



But there is a problem here too: each gate output value is stable, but what will it be? Or in other words buffer output can not be known. There is no way to tell. If we could know or set the value we would have a simple 1-bit storage/memory element.

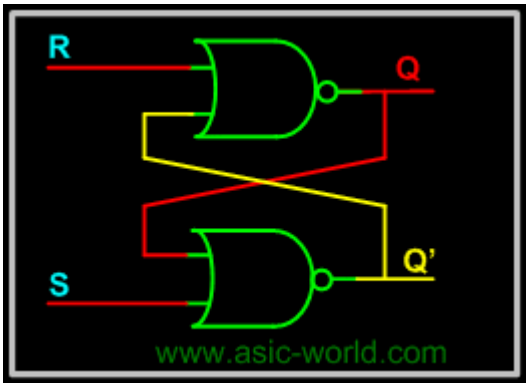
Latches and Flip-Flops

There are two types types of sequential circuits.

- Asynchronous Circuits.
- Synchronous Circuits.
- As seen in last section, Latches and Flip-flops are one and the same with a slight variation: Latches have level sensitive control signal input and Flip-flops have edge sensitive control signal input. Flip-flops and latches which use this control signals are called synchronous circuits. So if they don't use clock inputs, then they are called asynchronous circuits.

RS Latch

RS latch have two inputs, S and R. S is called set and R is called reset. The S input is used to produce HIGH on Q (i.e. store binary 1 in flip-flop). The R input is used to produce LOW on Q (i.e. store binary 0 in flip-flop). Q' is Q complementary output, so it always holds the opposite value of Q. The output of the S-R latch depends on current as well as previous inputs or state, and its state (value stored) can change as soon as its inputs change. The circuit and the truth table of RS latch is shown below. (This circuit is as we saw in the last page, but arranged to look beautiful :-)).

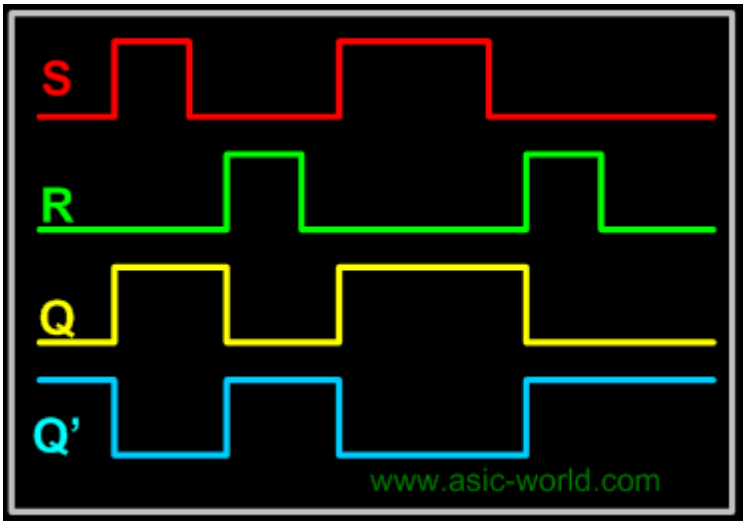


S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	X	0
1	0	X	1
1	1	X	0

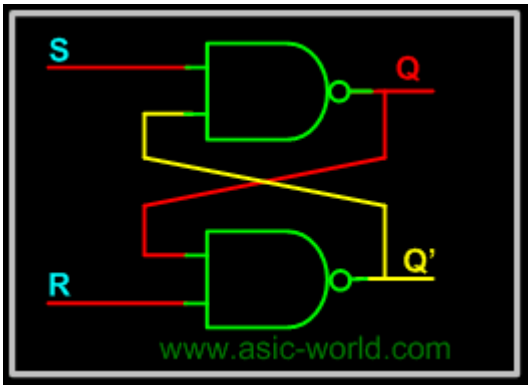
The operation has to be analyzed with the 4 inputs combinations together with the 2 possible previous states.

- **When S = 0 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So it is clear that when both S and R inputs are LOW, the output is retained as before the application of inputs. (i.e. there is no state change).
- **When S = 1 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. So in simple words when S is HIGH and R is LOW, output Q is HIGH.
- **When S = 0 and R = 1:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So in simple words when S is LOW and R is HIGH, output Q is LOW.
- **When S = 1 and R =1 :** No matter what state Q and Q' are in, application of 1 at input of NOR gate always results in 0 at output of NOR gate, which results in both Q and Q' set to LOW (i.e. Q = Q'). LOW in both the outputs basically is wrong, so this case is invalid.

The waveform below shows the operation of NOR gates based RS Latch.



It is possible to construct the RS latch using NAND gates (of course as seen in Logic gates section). The only difference is that NAND is NOR gate dual form (Did I say that in Logic gates section?). So in this case the R = 0 and S = 0 case becomes the invalid case. The circuit and Truth table of RS latch using NAND is shown below.

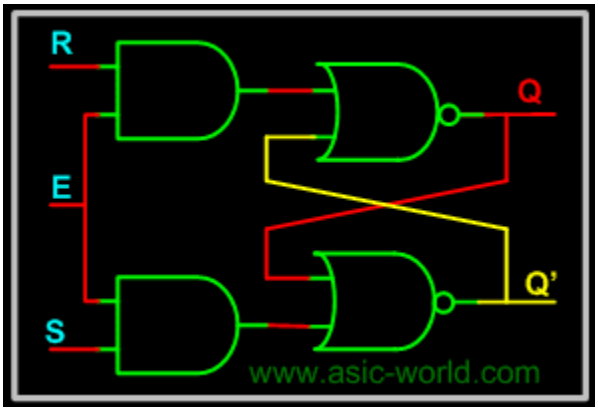


S	R	Q	Q+
1	1	0	0
1	1	1	1
0	1	X	0
1	0	X	1
0	0	X	1

If you look closely, there is no control signal (i.e. no clock and no enable), so this kind of latches or flip-flops are called asynchronous logic elements. Since all the sequential circuits are built around the RS latch, we will concentrate on synchronous circuits and not on asynchronous circuits.

RS Latch with Clock

We have seen this circuit earlier with two possible input configurations: one with level sensitive input and one with edge sensitive input. The circuit below shows the level sensitive RS latch. Control signal "Enable" E is used to gate the input S and R to the RS Latch. When Enable E is HIGH, both the AND gates act as buffers and thus R and S appears at the RS latch input and it functions like a normal RS latch. When Enable E is LOW, it drives LOW to both inputs of RS latch. As we saw in previous page, when both inputs of a NOR latch are low, values are retained (i.e. the output does not change).

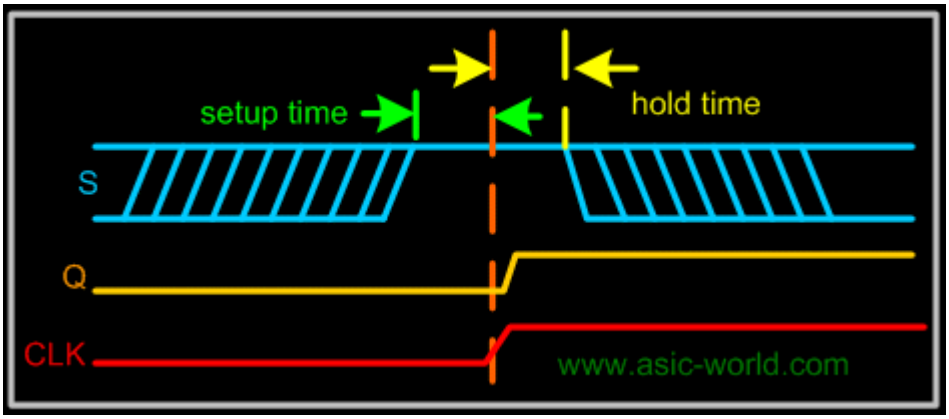


Setup and Hold Time

For synchronous flip-flops, we have special requirements for the inputs with respect to clock signal input. They are

- **Setup Time:** Minimum time period during which data must be stable before the clock makes a valid transition. For example, for a posedge triggered flip-flop, with a setup time of 2 ns, Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 2 ns before clock makes transition from 0 to 1.
- **Hold Time:** Minimum time period during which data must be stable after the clock has made a valid transition. For example, for a posedge triggered flip-flop, with a hold time of 1 ns. Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 1 ns after clock has made transition from 0 to 1.
- If data makes transition within this setup window and before the hold window, then the flip-flop output is not predictable, and flip-flop enters what is known as **meta stable state**. In this state flip-flop output oscillates between 0 and 1. It takes some time for the flip-flop to settle down. The whole process is called **metastability**. You could refer to tidbits section to know more information on this topic.

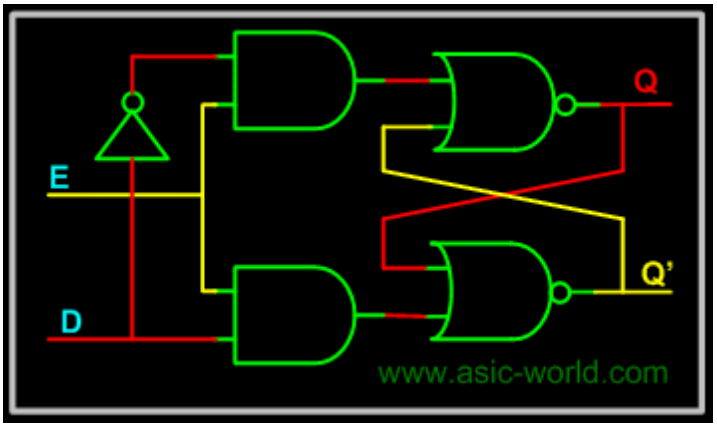
The waveform below shows input S (R is not shown), and CLK and output Q (Q' is not shown) for a SR posedge flip-flop.



D Latch

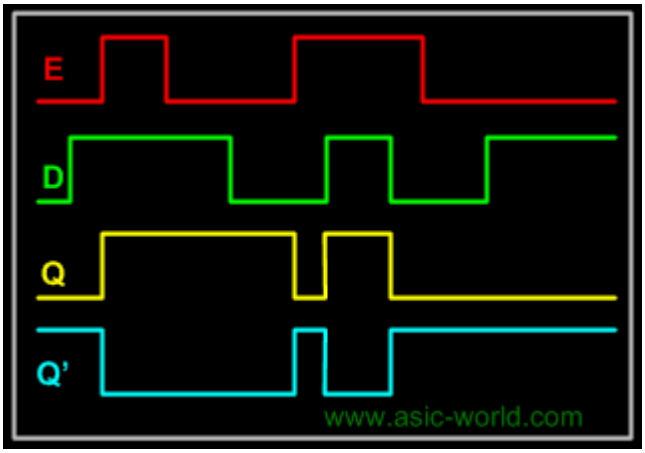
The RS latch seen earlier contains ambiguous state; to eliminate this condition we can ensure that S and R are never equal. This is done by connecting S and R together with an inverter. Thus we have D Latch: the same as the RS latch, with the only difference that there is only one input, instead of two (R and S). This input is called D or Data input. D latch is called D transparent latch for the reasons explained earlier. Delay flip-flop or delay latch is another name used. Below is the truth table and circuit of D latch.

In real world designs (ASIC/FPGA Designs) only D latches/Flip-Flops are used.



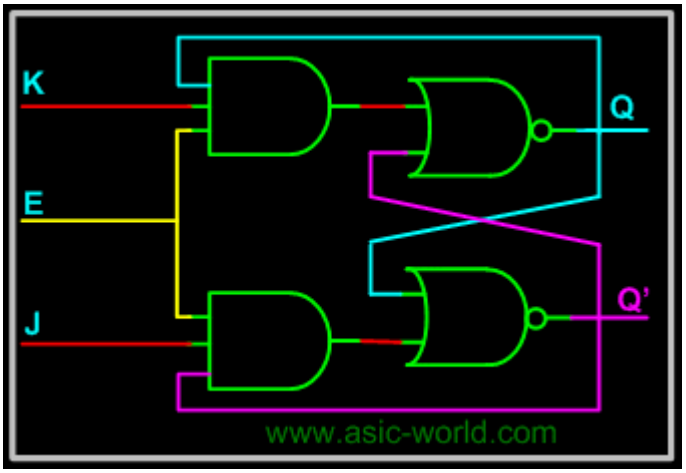
D	Q	Q+
1	X	1
0	X	0

Below is the D latch waveform, which is similar to the RS latch one, but with R removed.



JK Latch

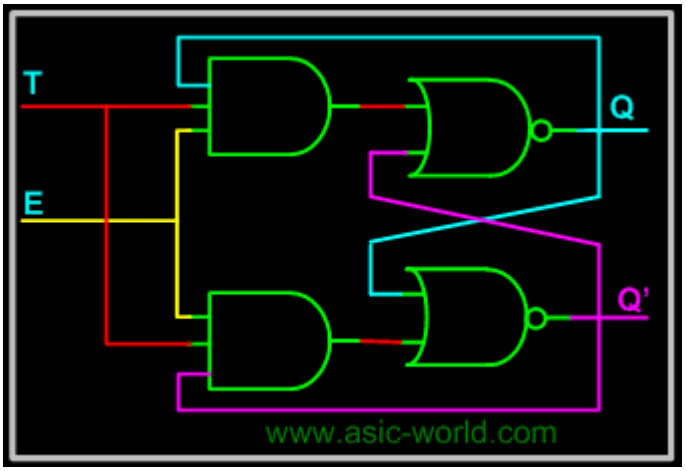
The ambiguous state output in the RS latch was eliminated in the D latch by joining the inputs with an inverter. But the D latch has a single input. JK latch is similar to RS latch in that it has 2 inputs J and K as shown figure below. The ambiguous state has been eliminated here: when both inputs are high, output toggles. The only difference we see here is output feedback to inputs, which is not there in the RS latch.



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

T Latch

When the two inputs of JK latch are shorted, a T Latch is formed. It is called T latch as, when input is held HIGH, output toggles.

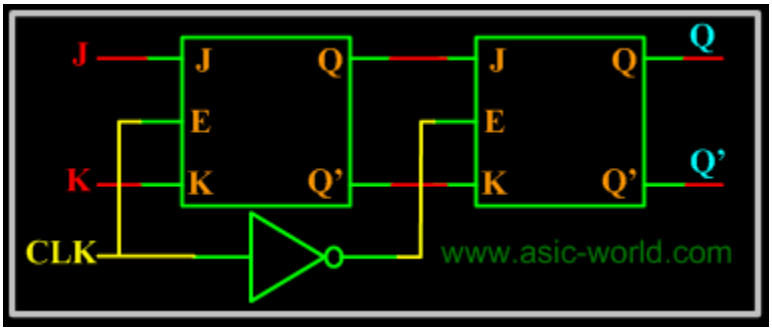


T	Q	Q+
1	0	1
1	1	0
0	1	1
0	0	0

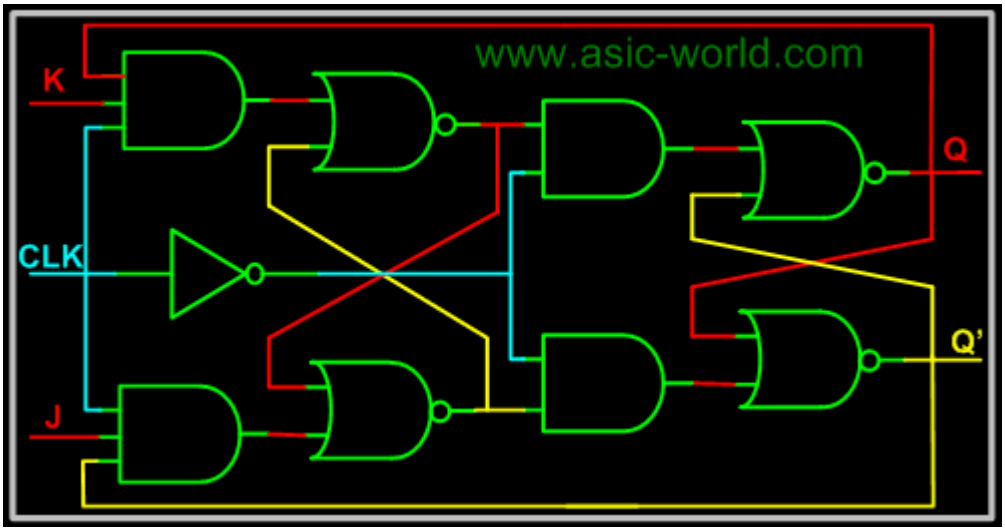
JK Master Slave Flip-Flop

All sequential circuits that we have seen in the last few pages have a problem (All level sensitive sequential circuits have this problem). Before the enable input changes state from HIGH to LOW (assuming HIGH is ON and LOW is OFF state), if inputs changes, then another state transition occurs for the same enable pulse. This sort of multiple transition problem is called racing.

If we make the sequential element sensitive to edges, instead of levels, we can overcome this problem, as input is evaluated only during enable/clock edges.



In the figure above there are two latches, the first latch on the left is called master latch and the one on the right is called slave latch. Master latch is positively clocked and slave latch is negatively clocked.



UNIT - IV

Sequential Circuits Design

We saw in the combinational circuits section how to design a combinational circuit from the given problem. We convert the problem into a truth table, then draw K-map for the truth table, and then finally draw the gate level circuit for the problem. Similarly we have a flow for the sequential circuit design. The steps are given below.

- Draw state diagram.
- Draw the state table (excitation table) for each output.
- Draw the K-map for each output.
- Draw the circuit.

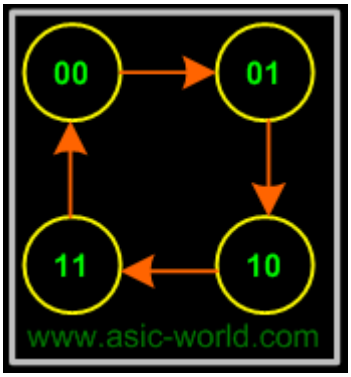
Looks like sequential circuit design flow is very much the same as for combinational circuit.

State Diagram

The state diagram is constructed using all the states of the sequential circuit in question. It builds up the relationship between various states and also shows how inputs affect the states.

To ease the following of the tutorial, let's consider designing the 2 bit up counter (Binary counter is one which counts a binary sequence) using the T flip-flop.

Below is the state diagram of the 2-bit binary counter.



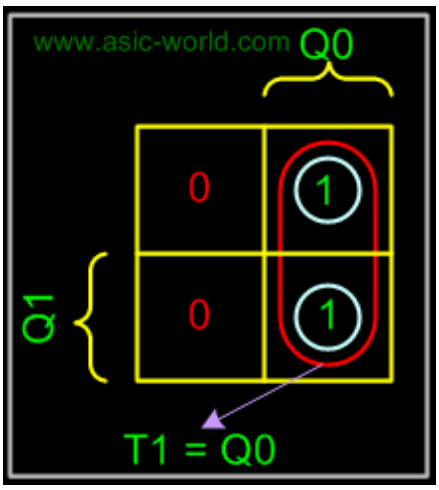
State Table

The state table is the same as the excitation table of a flip-flop, i.e. what inputs need to be applied to get the required output. In other words this table gives the inputs required to produce the specific outputs.

Q1	Q0	Q1+	Q0+	T1	T0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

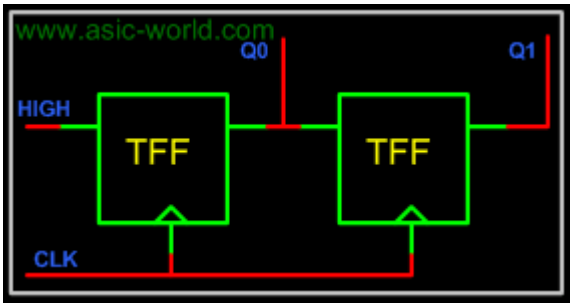
K-map

The K-map is the same as the combinational circuits K-map. Only difference: we draw K-map for the inputs i.e. T1 and T0 in the above table. From the table we deduct that we don't need to draw K-map for T0, as it is high for all the state combinations. But for T1 we need to draw the K-map as shown below, using SOP.



Circuit

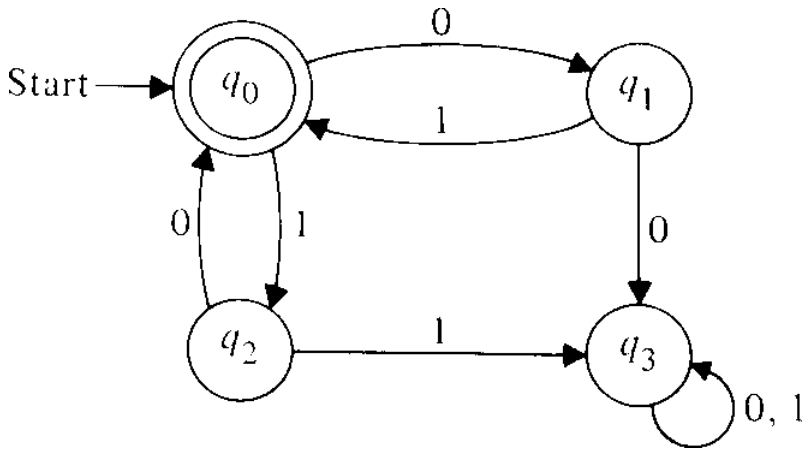
There is nothing special in drawing the circuit, it is the same as any circuit drawing from K-map output. Below is the circuit of 2-bit up counter using the T flip-flop.



UNIT-V

Finite State Machine

A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machine), one or more states designated as accepting states (recognizer), etc.



Definition 1 *A finite state machine is a 5-tuple,* (S, A,R, _, s0) *where* S *is a finite set of* states*,* A *is a finite* alphabet*,* R *is a finite alphabet of* responses *and* _ *is a* transition function *such that for any state,*s 2 S *and symbol* a 2 A*,* _(s, a) = (s0, r0) *indicates the* next state*,* s0 *and the* output symbol*,* r0 2 R*.*
s0 *is the initial state.*

Definition 2 *A* recogniser *is a special kind of finite state machine in which the output alphabet contains two special symbols:* accept *and* reject*. The machine responds to any finite sequence of input*
symbols, terminated with a special end of input *symbol (_), with either* accept *or* reject*.*

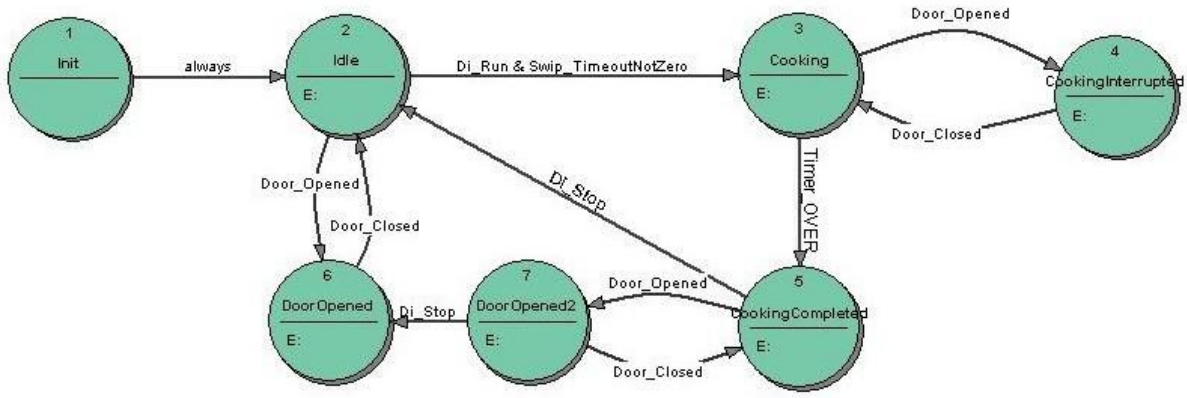
Limitations of Finite State Machines:

- Number of states in the composed FSM grows dramatically (state explosion problem)
- Composing FSMs of n subsystems, with k1 , k2 , k3, ... kn, states respectively, results in a system whose FSM has k1 x k2 x …. x kn states - This growth is exponential with the number of subsystems , not linear (i.e., k1 + k2 +… + kn).
- Since at any time, a global state of the system must be defined and a single transition must occur, FSM model is not suitable for describing asynchronous concurrent activities in the system.

Mealy And Moore Models

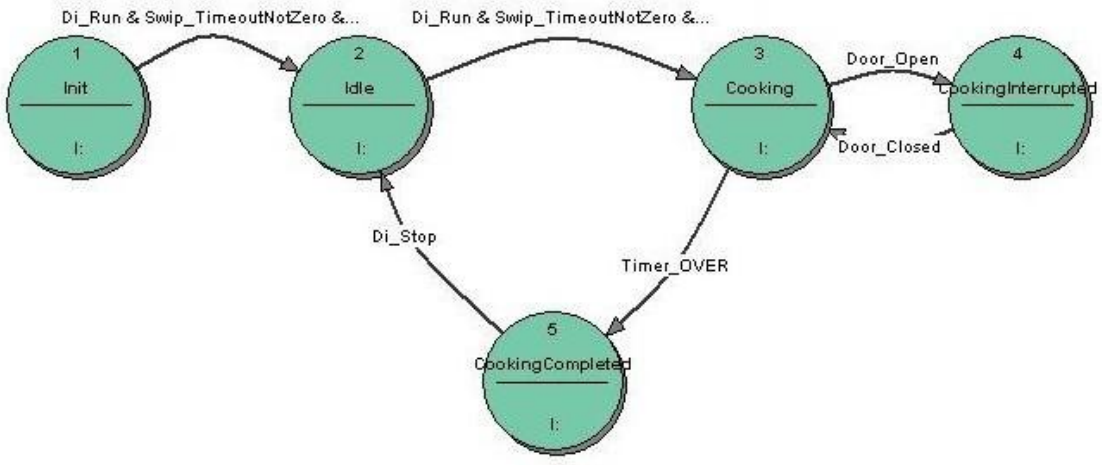
Mealy and Moore models are the basic models of state machines. A state machine which uses only Entry Actions, so that its output depends on the state, is called a Moore model. A state machine which uses only Input Actions, so that the output depends on the state and also on inputs, is called a Mealy model. The models selected will influence a design but there are no general indications as to which model is better. Choice of a model depends on the application, execution means (for instance, hardware systems are usually best realized as Moore models) and personal preferences of a designer or programmer. In practice, mixed models are often used with several action types. On an example we will show the consequences of using a specific model. As the example we have taken a Microwave Oven control. The oven has a momentary-action push button Run to start (apply the power) and a Timer that determines the cooking length. Cooking can be interrupted at any time by opening the oven door. After closing the door the cooking is continued. Cooking is terminated when the Timer elapses. When the door is open a lamp inside the oven is switched on, when the door is closed the lamp is off. During cooking the lamp is also switched on. The cooking period (timeout value) is set by a potentiometer which supplies a voltage to the control system: the voltage is represented by a numeric value 0..4095 which is scaled by the Ni object to 1799. This arrangement allows the maximum cooking time to equal 1799 seconds, i.e. 30 minutes. The solution should also take into account the possibility that the push button Run could get blocked continuously in the active Position (which is easy to demonstrate if testing the system in SWLab, which has only the two-positions buttons): in such a case cooking must not start again until it is deactivated when the cooking is terminated (otherwise our meal which we wanted to heat for instance for 5 minutes could be burned until we discover that the button has got stuck in the active position). In other words, each cooking requires intentional activation of the Run button.
The control system has the following inputs: Run momentary-action push button - when activated starts cooking, Timer - while this runs keep on cooking, Door sensor - can be true (door closed) or false (door open). And the following outputs: Power - can be true (power on) or false (power off), Lamp - can be true (lamp on) or false (lamp off).
Moore model : Using Moore model we get a state machine whose state transition diagram is shown in Figure 1.
This solution requires 7 states. Figure 2, Figure 3 and Figure 4 show state transition tables for three of those states: Init, Cooking and CookingInterrupted. The state machine uses only Entry actions. Other states can be studied in the provided file MWaveOven_Moore.fsm.
While specifying that state machine the states dominate. We think in the following manner: if the input condition changes the state machine changes its state (if a specific transition condition is valid). Entering the new state, the state machine does some actions and waits for the reaction of the

controlled system. In a Moore model the entry actions define effectively the state. For instance, we would think about the state Cooking: it is a state where the Timer runs and the state machines waits for the Timer OVER signal.



Mealy model

The Mealy model is shown in Figure 5. It requires only 5 states. The states: Idle, Cooking and Cooking Interrupted for that model (see Figure 6, Figure 7 and Figure 8) illustrate its features. Other states can be studied in the provided file MWaveOven_Mealy.fsm. All activities are done as Input actions, which means that actions essential for a state must be performed in all states which have a transition to that state. The Timer must be now started in both states: Idle and Cooking Interrupted.
This may be considered as a disadvantage: the functioning becomes a bit confusing.



Minimization of FSM:

INPUT **OUTPUT**

Algorithmic State Machine

The **Algorithmic State Machine** (ASM) method is a method for designing finite state machines. It is used to represent diagrams of digital integrated circuits. The ASM diagram is like a state diagram but less formal and thus easier to understand. An ASM chart is a method of describing the sequential operations of a digital system.

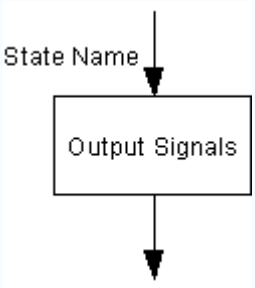
ASM Method

The ASM method is composed of the following steps:

1. Create an algorithm, using *pseudocode*, to describe the desired operation of the device.
2. Convert the pseudocode into an *ASM chart*.
3. Design the *datapath* based on the ASM chart.
4. Create a *detailed ASM chart* based on the datapath.
5. Design the *control logic* based on the detailed ASM chart.

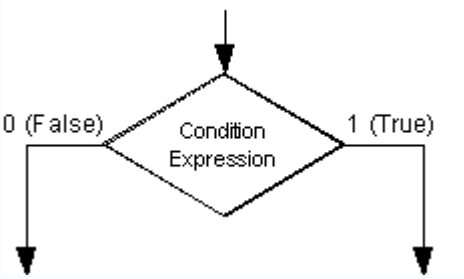
ASM Chart

An ASM chart consists of an interconnection of three types of basic elements: states, condition checks, and conditional outputs. An ASM state, represented as a rectangle, corresponds to one state of a regular state diagram or finite state machine. The name of the state is indicated outside the box in the top left corner. The Moore type outputs are listed inside the box.



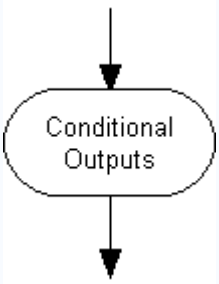
State box

An ASM condition check, indicated by a diamond with one input and two outputs (for true and false), is used to conditionally transfer between two states or between a state and a conditional output. The decision box contains the stated condition expression to be tested, the expression contains one or more inputs of the FSM.



Decision box

Decision box: A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly. The condition expression contains one or more inputs to the FSM.



Conditional output box

Conditional output box: An oval denotes the output signals that are of Mealy type. These outputs depend not only on the state but also the inputs to the FSM.

Datapath

Once the desired operation of a circuit has been described using RTL operations, the datapath components may be derived. Every unique variable that is assigned a value in the RTL program can be implemented as a register. Depending on the functional operation performed when assigning a value to a variable, the register for that variable may be implemented as a straightforward register, a shift register, a counter, or a register preceded by a combinational logic block. The combinational logic block associated with a register may implement an adder, subtracter, multiplexer, or some other type of combinational logic function.

Binary multiplier:

A **binary multiplier** is a electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders.

Theory

Using long multiplication, a product of two N-bit numbers can be expressed as the sum of N N-bit partial products, which are then added to produce a 2N-bit product.

$$
\begin{array}{rcccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & & \\
 & \times & & b_3 & b_2 & b_1 & b_0 & & \\
\hline
 & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & & & \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & & & \\
a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

The partial products can be trivially computed from the fact that $a_i \times b_j = a_i$ **AND** b_j. The complexity of the multiplier is in adding the partial products.

Implementation

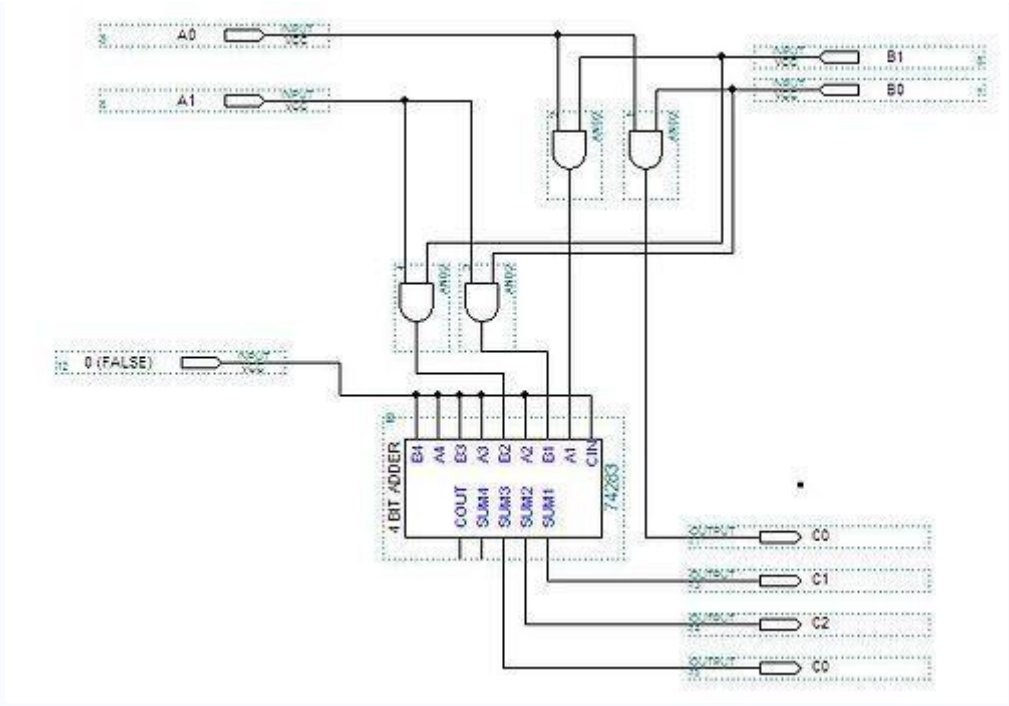
There are several ways to implement a binary multiplier.

Multiple adders

Partial products are added in pairs using binary adders until the entire product is computed similar to multiplying large numbers by hand. This requires N-1 adders.

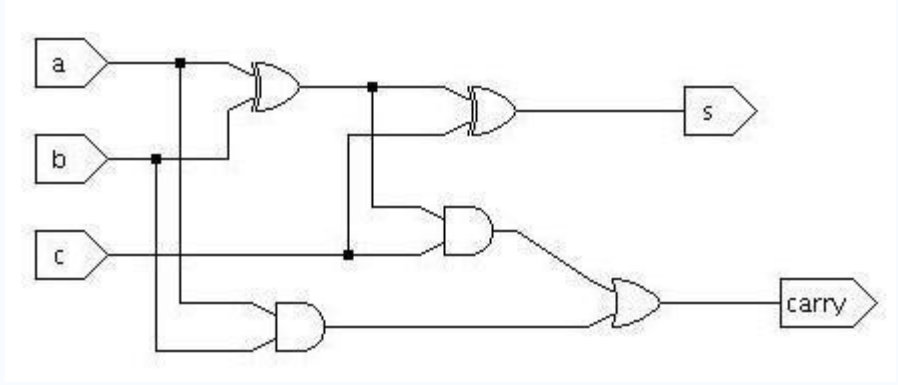
Typically those adders are arranged as an adder compressor tree.

Example:

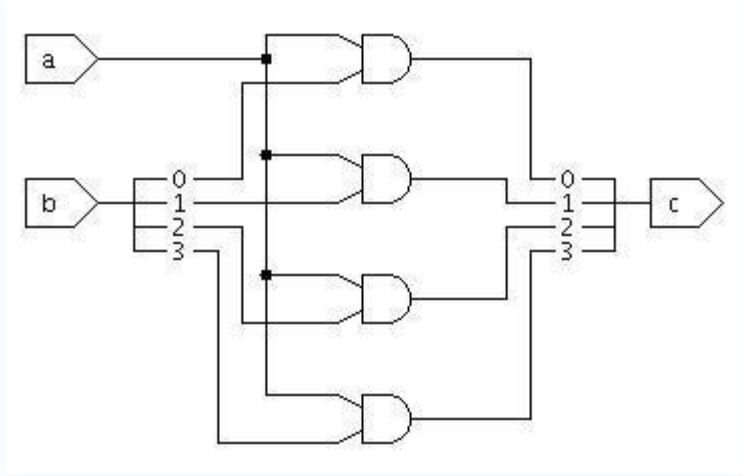


2 Bit by 2 Bit Binary Multiplier
Using a 4 Bit + 4 Bit Adder

(The following are to clarify each level of abstraction)

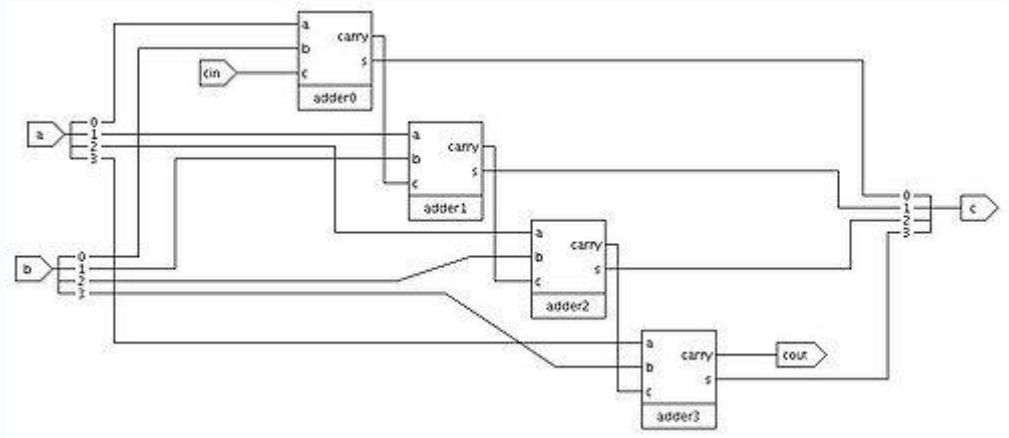


A simple adder
1-bit adder



"Ander"

A 1x4 bit Ander



4-bit Adder

Using 4 1-bit adders

19. Tutorial Problems:

TUTORIAL-I

Converting from binary, hexadecimal and octal to decimal

binary$_2$ -> decimal

1101_2 -> decimal

First write 2 to the power of the numbers

0,1,2,3,... 2^3 2^2 2^1 2^0

Now change them to their real values.

8 4 2 1

Now put the binary number underneath the other numbers and multiply the top number by the number beneath it and put the answer underneath the other 2 with + between each number and add the bottom row together to get your final answer.

8 4 2 1
1 1 0 1
8+4+0+1 = 13

1101_2 -> 13

hexadecimal$_{16}$ -> decimal

Once again you use 16 instead of 2.

$1F_{16}$ -> decimal

$16^1\ 16^0$

16 1
1 F
16+F = 16 + 15 = 31

$1F_{16}$ -> 31

octal$_8$ -> decimal

25_8 -> decimal

$8^1\ 8^0$

8 1
2 5
16+5 = 21

15_8 -> 21

TUTORIAL - II

Subtracting the numbers using two's complement

The most common way of subtracting binary numbers is done by first taking the second value (the number to be subtracted) and apply what is known as **two's complement**, this is done in two steps:

1. complement each digit in turn (change 1 for 0 and 0 for 1).
2. add 1 (one) to the result.

note: the first step by itself is known as **one's complement**.

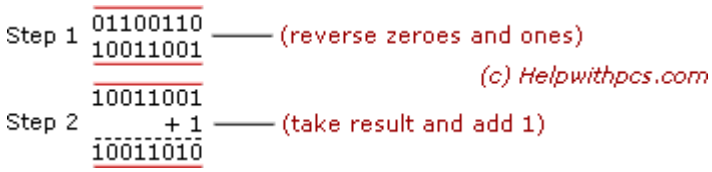
By applying these steps you are effectively turning the value into a negative number, and as when dealing with decimal numbers, if you add a negative number to a positive number then you are effectively subtracting to the same value.

In other words 25 + (-8) = 17, which is the same as writing 25 - 8 = 17.

An example, let's do the following subtraction **11101011 - 01100110** (235_{10} - 102_{10}

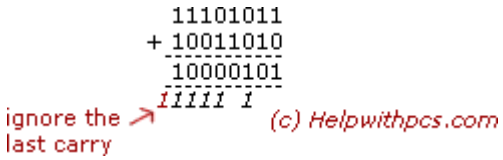
NOTE: When subtracting binary values it is important to maintain the same amount of digits for each number, even if it means placing zeroes to the left of the value to make up the digits, for instance, in our example we have added a zero to the left of the value **1100110** to make the amount of numerals up to 8 (one byte) **01100110**.

First we apply two's complement to 01100110



which gives us **10011010**.

Now we need to add **11101011 + 10011010**, however when you do the addition you always disregard the last carry, so our example would be:



which gives us **10000101**, now we can convert this value into decimal, which gives **133_{10}**

So the full calculation in decimal is 235_{10} - 102_{10} = 133_{10} (correct !!)

Binary multiplication

Example 1

Base 2 Place	2^3	2^2	2^1	2^0	
			1	1	3
	x		1	1	x 3
Carry	1	1	1	1	9
		1	1		
	1	0	0	1	

Example 2

Base 2 Place	2^4	2^3	2^2	2^1	2^0	
		1	0	0	1	9
		x		1	0	x 2
Carry		0	0	0	0	18
	1	0	0	1		
	1	0	0	1	0	

Example 3

Base 2 Place	2^5	2^4	2^3	2^2	2^1	2^0	
			1	0	1	1	11
			x		1	1	x 3
Carry	1	1	1	1	0	1	33
		1	0	1	1		
	1	0	0	0	0	1	

TUTORIAL - III
Subtracting the numbers using 9's complement & 10's complement

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.

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

873 (x)

+ 781 (nines' complement of y)

=====

1654

-1000 (y + nines' complement of y + 1) or (y + tens' complement of y)

=====

654

The first "1" digit is then dropped, in an effort to keep the same digits as the original, giving 654. This is not yet correct. We have essentially added 999 to the equation in the first step. Then we remove 1000 when we drop the first 1 in the answer (above) this will make the answer we get one less than the correct answer. To fix this, we must add 1 to the answer.

654

+1

====

655

Adding a 1 gives 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 complement is taken. For example:

48032 (x)

- 391 (y)

becomes the sum:

48032 (x)

+ 99608 (nines' complement of y)

=======

147640

Dropping the "1" yields 47640, and adding the dropped "1" to 47640 gives the answer: 47641.

Binary addition

Adding binary numbers is a very simple task, and very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

0 + 0 = 0

1 + 0 = 1

0 + 1 = 1

1 + 1 = 10

1 + 1 + 1 = 11

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples:

. 11 1 <--- Carry bits -----> 11

. 1001101 1001001 1000111

. + 0010010 + 0011001 + 0010110

. ---------- --- --------- ----------

. 1011111 1100010 1011101

The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

TUTORIAL-IV

Digital logic gates

Digital logic gates, which are also known as combinational logic gates or simply 'logic gates', are digital IC's whose output at any time is determined by the states of its inputs at that time. Since logic gates are digital IC's, their input and output signals can only be in one of two possible digital states, i.e., logic '0' or logic '1'. Thus, the logic state in which the output of a logic gate will be put in depends on the logic states of each of its individual inputs.
The primary application of logic gates is to implement 'logic' in the flow of digital signals in a digital circuit. Logic in its ordinary sense is defined as a branch of philosophy that deals with what is true and false, based on what other things are true and false. This essentially is the function of logic gates in digital circuits - to determine which outputs will be true or false, given a set of inputs that can either be true (logic '1') or false (logic '0').

The response output (usually denoted by Q) of a logic gate to any combination of inputs may be tabulated into what is known as a truth table. A truth table shows each possible combination of inputs to a logic gate and the combination's corresponding output. Table 1, which describes the various types of logic gates, provides a truth table for each of them as well.
Interestingly, the operation of logic gates in relation to one another may be represented and analyzed using a branch of mathematics called Boolean Algebra which, like the common algebra, deals with manipulation of expressions to solve or simplify equations. Expressions used in Boolean Algebra are called, well, Boolean expressions.

Table 1. Logic Gates and their Properties

<table>
<tr><th>Gate</th><th>Description</th><th>Truth Table</th></tr>
<tr><td>AND Gate</td><td>The AND gate is a logic gate that gives an output of '1' only when all of its inputs are '1'. Thus, its output is '0' whenever at least one of its inputs is '0'. Mathematically, Q = A · B.</td><td>
<table>
<tr><th>A</th><th>B</th><th>Output Q</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>OR Gate</td><td>The OR gate is a logic gate that gives an output of '0' only when all of its inputs are '0'. Thus, its output is '1' whenever at least one of its inputs is '1'. Mathematically, Q = A + B.</td><td>
<table>
<tr><th>A</th><th>B</th><th>Output Q</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>NOT Gate</td><td>The NOT gate is a logic gate that gives an output that is opposite the state of its input. Mathematically, Q = A.</td><td>
<table>
<tr><th>A</th><th>Output Q</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>NAND Gate</td><td>The NAND gate is an AND gate with a NOT gate at its end. Thus, for the same combination of inputs, the output of a NAND gate will be opposite that of an AND gate. Mathematically, Q = A · B.</td><td>
<table>
<tr><th>A</th><th>B</th><th>Output Q</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>NOR Gate</td><td>The NOR gate is an OR gate with a NOT gate at its end. Thus, for the same combination of inputs, the output of a NOR gate</td><td>
<table>
<tr><th>A</th><th>B</th><th>Output Q</th></tr>
</table>
</td></tr>
</table>

	will be opposite that of an OR gate. Mathematically, Q = A + B.	0	0	1
		0	1	0
		1	0	0
		1	1	0
EXOR Gate	The EXOR gate (for 'EXclusive OR' gate) is a logic gate that gives an output of '1' when only one of its inputs is '1'.	**A**	**B**	**Output Q**
		0	0	0
		0	1	1
		1	0	1
		1	1	0

There are several kinds of logic gates, each one of which performs a specific function. These are the: 1) AND gate; 2) OR gate; 3) NOT gate; 4) NAND gate; 5) NOR gate; and 6) EXOR gate. Table 1 above presents these and their characteristics.

TUTORIAL - V

Minimization of Boolean expressions

1. $Z = f(A,B,C) = \bar{A}\,\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC$

   $= (B + \quad) + A(C + B \quad)$

   $= (B + \quad) + A(C + B)$ from T10b

   $= \bar{A}B + \bar{A}\bar{C} + AC + AB$

   $= B(\bar{A} + A) + \bar{A}\bar{C} + AC$ from T9a and T8b

   $= B + \bar{A}\bar{C} + AC$

   $Z = f(A,B,C,D) = \bar{A}B + B\bar{C} + BC + A\bar{B}\bar{C}$

$= \bar{A}B + B(C + \bar{C}) + \bar{C}(B + A\bar{B})$ using $B\bar{C}$ (twice) T4a

$= \bar{A}B + B + \bar{C}(B + A)$ from T9a, T8b and T10b

$= B(1 + \bar{A}) + B\bar{C} + A\bar{C}$ from T8a

$= B + B\bar{C} + A\bar{C}$ from T8a

$= B(1 + \bar{C} + A\bar{C}$

$= B + A\bar{C}$ from T8a

2. $Z = f(A,B,C,D) = AB\bar{C}\bar{D} + AB\bar{C}D + A\bar{B}\bar{C}D + ABCD + A\bar{B}CD + ABC\bar{D} + A\bar{B}C\bar{D}$

$= AB\bar{C} + ABC + A\bar{B}C + A\bar{B}D$ from T9a and T8b

$= AB + AC + A\bar{B}D$ from T9a and T8b

$= A(B + \bar{B}D) + AC$

$= A(B + D) + AC$ from T10a

$= AB + AD + AC$

Expressing the functions in SOP terms

Two step approach to represent three variables in term of Minterms:-

Step1: Represent the Minterm needs to be considered for the function by '1'

Step2: Take the OR of all Minterms to represent the function.

The Function of Minterms F = x'y'z + x'yz' + xy'z + xyz' + xyz

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

= xyz + xyz’ + xy’z + xy’z’ + xyz + x’yz + xyz + xyz’

= xyz + xyz’ + xy’z + xy’z’ + x’yz

Expressing the functions in POS terms

Representation of Boolean Function in Product of Maxterms or Canonical Forms

Product of Maxterms can be simply obtained by taking the complement of sum of Minterms from the Truth Table.

F1 = (x + y + z)(x + y’ + z’)(x’ + y + z)

Example: Represent F = x + yz + xy in Product of Sum terms

F = (x + yz + x)(x + yz + y)

= (x + yz)(x + y +yz)

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

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

= (x +y + zz’)(x + z + yy’)(x +y +z)

= (x + y + z)(x +y + z’)(x + z + y)(x + z + y’)(x + y + z)

= (x + y + z)(x + y + z’)(x + y’ +z)

TUTORIAL - VI

Duality principle

Duality Principle states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

Postulates *a* and *b*

Postulate 2	$x + 0 = x$	$x \cdot 1 = x$

Postulate 3, Commutative	$x + y = y + x$	$xy = yx$
Postulate 4, Distributive	$x\,(y + z) = xy$	$x + yz = (x + y)(x$
Postulate 5	$x + x' = 1$	$x \bullet x' = 0$

Theorems *a* and *b*

Theore	*a*	*b*
Theorem 1	$x + x = x$	$x \bullet x = x$
Theorem 2	$x + 1 = 1$	$x \bullet 0 = 0$
Theorem 3,	$(x')' = x$	
Theorem 4,	$x + (y + z) = (x +$	$x \bullet (y \bullet z) = (x \bullet y)$
Theorem 5,	$(x + y)' = x'y'$	$(x \bullet y)' = x' + y'$
Theorem 6,	$x + xy = x$	$x\,(x + y) = x$

Prove: (A*B)*(A + B) = 0

(A*B)*(A + B) = (A*B)*A + (A*B)*B) by distributive postulate

= (A*A)*B + A*(B*B) by Associativity postulate

= 0*B + A*0 by Complement postulate

= 0 + 0 by Nullity theorem

= 0 by identity theorem

(A*B)*(A + B) = 0

Prove: (A*B) + (A + B) =1

(A*B) + (A + B) =(A + A + B))*(B + A + B) by distributivity B*C + A = (B + A)*(C + A) (A*B) + (A + B) =(A + A + B))*(B + B + A) by associativity postulate

=(1 + B)*(1 + A) by complement postulate

=1*1 by nullity theorem

=1 by identity theorem

(A*B) + (A + B) =1

Since (A*B)*(A + B) = 0, and (A*B) + (A + B) =1,

A*B is the complement of A + B, meaning that A*B=(A + B)';

(note that ' = complement or NOT - double bars don't show in HTML) Thus A*B= (A + B)".

The involution theorem states that A" = A. Thus by the involution theorem, (A + B)" = A + B. This proves DeMorgan's Theorem (b).

DeMorgan's Theorem (a) may be proven using a similar approach.

Proof of Theorem 6(a)

x + xy = x

x + xy = x•1 + xy = x(y+1) =x •1 = x

Proof of Theorem 6(b)

x(x+y) = x By duality

TUTORIAL-VIII

Obtaining simplified expression in SOP terms

K-Maps for Sum of Products (SOP)

Consider the Canonical SOP expression F(X,Y,Z) = X'•Y•Z + X•Y'•Z + X•Y•Z' + X•Y•Z.

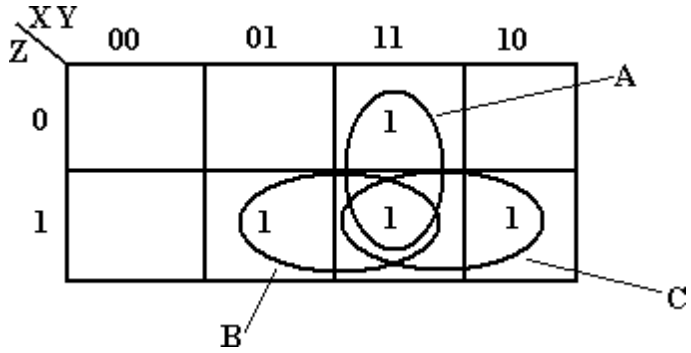
The first step in using K-Maps to simplify this expression is to use the SOP numbering to represent these as 0's and 1's. The negated variable is written as a 0, the plain as a 1. Thus, this function is represented as 011, 101, 110, and 111.

Z \ XY	00	01	11	10
0			1	
1		1	1	1

Place a 1 in each of the squares with the "coordinates" given in the list above. In the K-Map at left, the entry in the top

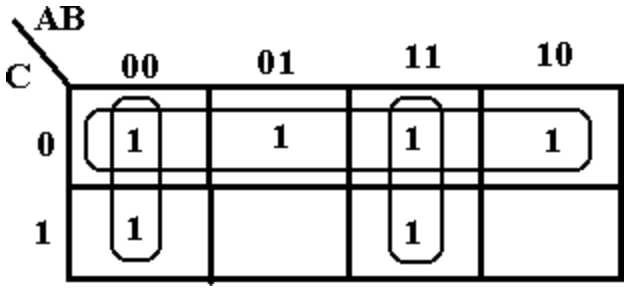
row corresponds to 110 and the entries in the bottom row correspond to 011, 111, and 101 respectively. Remember that we do not write the 0's when we are simplifying expressions in SOP form.

The next step is to notice the physical adjacencies. We group adjacent 1's into "rectangular" groupings of 2, 4, or 8 boxes. Here there are no groupings of 4 boxes in the form or a rectangle, so we group by two's. There are three such groupings, labeled A, B, and C.



The grouping labeled A represents the product term XY. The B group represents the product term YZ and the C group represents the product term XZ. Examine the B grouping: it has 011 and 111. In this we have Y and Z staying the same and X having both values; thus the product term YZ. This function is X•Y + X•Z + Y•Z.

the next example is to simplify F(A, B, C) = **Π**(3, 5). We shall consider use of K-Maps to simplify POS expressions, but for now the solution is to convert the expression to the SOP form F(A, B, C) = **Σ**(0, 1, 2, 4, 6, 7). We could write each of the six product terms, but the easiest solution is to write the numbers as binary: 000, 001, 010, 100, 110, and 111.



The top row of the K-Map corresponds to the entries 000, 010, 100, and 110, arranged in the order 000, 010, 110, and 100 to preserver logical adjacency. The bottom row corresponds to the entries 001 and 111. The top row simplifies to C'.

The first column simplifies to A'B' and the third column to AB. Thus we have F(A, B, C) = A'•B' + A•B + C'.

We next consider a somewhat offbeat example not in a canonical form.

F(W, X, Y, Z) = W'•X'•Y'•Z' + W'•X'•Y'•Z + W•X'•Y'.

The trouble with K-Maps is that the technique is designed to be used only with expressions in canonical form. In order to use the K-Map method we need to convert the term W•X'•Y' to its equivalent W•X'•Y'•Z' + W•X'•Y'•Z, thus obtaining a four-term canonical SOP.

Before actually doing the K-Map, we first apply simple algebraic simplification to F.

F(W, X, Y, Z)= W'•X'•Y'•Z' + W'•X'•Y'•Z + W•X'•Y'

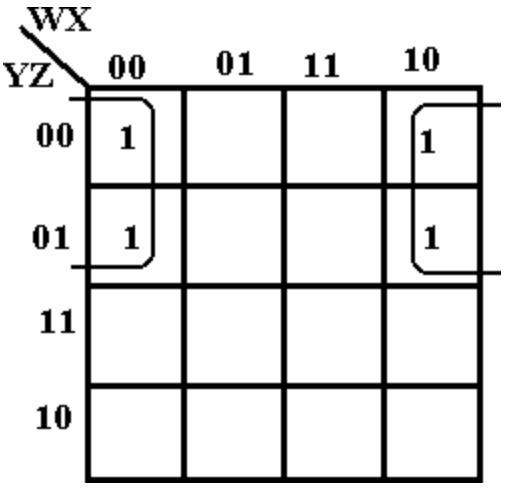
= W'•X'•Y'•(Z' + Z) + W•X'•Y'

= W'•X'•Y' + W•X'•Y'

= (W' + W)•X'•Y' = X'•Y'

Now that we see where we need to go with the tool, we draw the four-variable K-Map.

F(W, X, Y, Z) = W'•X'•Y'•Z' + W'•X'•Y'•Z + W•X'•Y'•Z' + W•X'•Y'•Z. Using the SOP encoding method, these are terms 0000, 0001, 1000, and 1001. The K-Map is



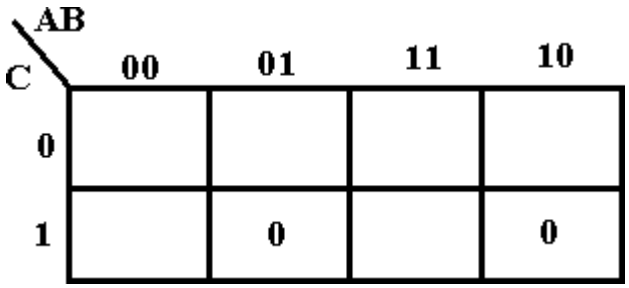
The first row in the K-Map represents the entries 0000 and 1000. The second row in the K-Map represents the entries 0001 and 1001. The trick here is to see that the last column is adjacent to the first column The four cells in the K-Map are thus adjacent and can be grouped into a square. We simplify by noting the values that are constant in the square: X = 0 and Y = 0.

Obtaining simplified expression in POS terms

K-Maps for POS

K-Maps for Product of Sums simplification are constructed similarly to those for Sum of Products simplification, except that the POS copy rule must be enforced: 1 for a negated variable and 0 for a non-negated (plain) variable.

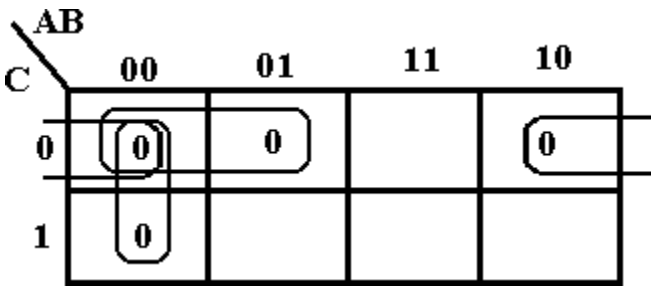
As our first example we consider F(A, B, C) = **Π**(3, 5) = (A + B' + C')•(A' + B + C'). Recall that the term (A + B' + C') corresponds to 011 and that (A' + B + C') to 101.



This is really somewhat of a trick question used only to illustrate placing of the terms for POS. Place a 0 at each location, rather than the 1 placed for SOP. Note that the two 0's placed are not adjacent, so we cannot simplify the expression.

For the next example consider F2 = (A + B + C)•(A + B + C')•(A + B' + C)•(A' + B + C). Using the POS copy rule, we translate this to 000, 001, 010, and 100.

Before we attempt to simplify F2, we note that it is a very good candidate for simplification. Compare the first term 000 to each of the following three terms. The term 000 differs from the term 001 in exactly one position. The same applies for comparison to the other two terms. Any two terms that differ in exactly one position can be combined in a simplification.



We begin the K-Map for POS simplification by placing a 0 in each of the four positions 000, 001, 010, 100. Noting that 000 is adjacent to 001, just below it, we combine to get 00– or (A + B). The term 000 is adjacent to 010 to its right to get 0–0 or (A + C). The term 000 is adjacent to 100 to its "left" to get –00 or (B + C). As a result, we get the simplified form. F2 = (A + B)•(A + C)•(B + C)

Just for fun, we simplify this expression algebraically, using the derived Boolean identity

X•X•X = X for any Boolean expression X.

F2 = (A + B + C)•(A + B + C')•(A + B' + C)•(A' + B + C)

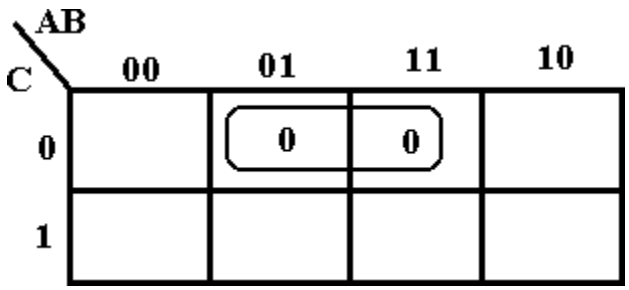
= (A + B + C)•(A + B + C')•(A + B + C)•(A + B' + C)•(A + B + C)•(A' + B + C)

= (A + B)•(A + C)•(B + C)

It is encouraging that we get the same answer.

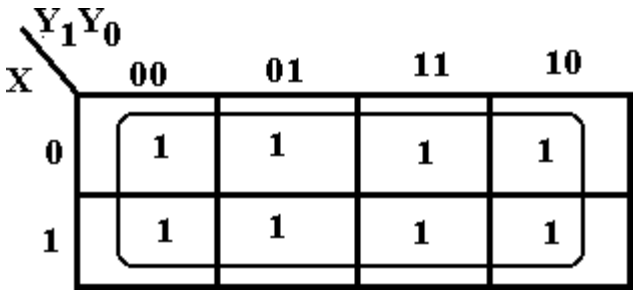
We now consider simplification of a POS function specified by a truth table.

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



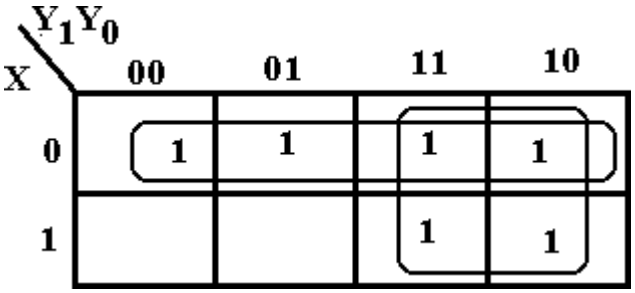
We plot two 0's for the POS representation of the function – one at 010 and one at 110. The two are combined to get –10, which translates to (B' + C).

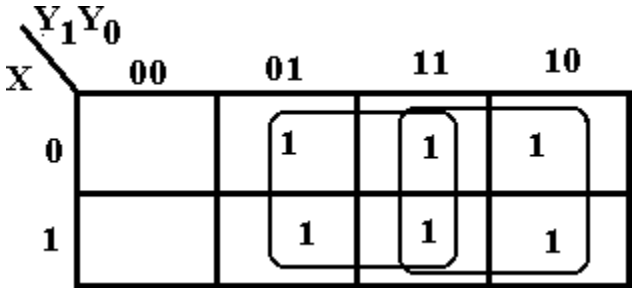
More Examples of K-Maps



The sample at left, based on an earlier design shows a particularly simple problem. We find that all the entries in the K-Map are covered with a single grouping, thus removing all three variables. Since the entire K-Map is covered, the simplification is F = 1.

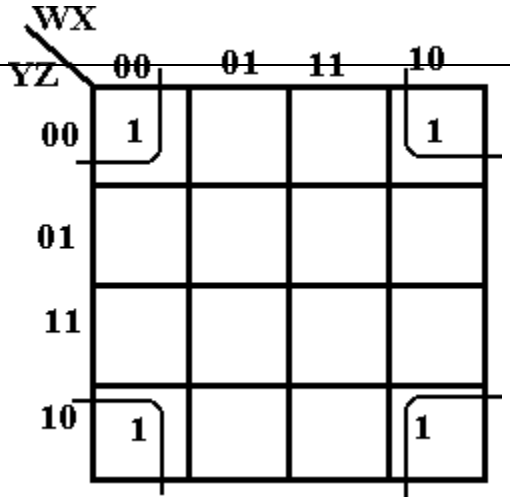
The K-Map at right shows an example with overlap of two groupings of 1's. All 1's in the map must be covered and some should be covered twice. The top row corresponds to X'. We then form the 2-by-2 grouping at the right to obtain the term Y_1. Thus F = X' + Y_1.





There is another simplification that should be considered. This corresponds to two 2-by-2 groupings. The 2-by-2 grouping at the right still corresponds to Y_1. The new 2-by-2 grouping in the middle gives rise to Y_0, so we get another simplification F = Y_0 + Y_1.

We close the discussion of SOP K-Maps with the example at right, which shows that the four corners of the square are adjacent and can be grouped into a 2 by 2 square. This K-Map represents the terms 0000, 0010, 1000, 1010 or W'•X'•Y'•Z' + W'•X'•Y•Z' +
W•X'•Y'•Z' + W•X'•Y•Z'. The values in the square that are constant are X = 0 and Z = 0, thus the expression simplifies to X'•Z'.



TUTORIAL - IX

Finding prime implicants of Boolean functions and determining the essentials

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 function 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*.

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.

Using the example above, one can easily see that while *xy* (and others) is a prime implicant, *xyz* and *xyzw* are not. From the latter, multiple literals can be removed to make it prime:

- *x*, *y* and *z* can be removed, yielding *w*.
- Alternatively, *z* and *w* can be removed, yielding *xy*.
- Finally, *x* and *w* can be removed, yielding *yz*.

The process of removing literals from a Boolean term is called **expanding** the term. Expanding by one literal doubles the number of input combinations for which the term is true (in binary Boolean algebra). Using the example function above, we may expand *xyz* to *xy* or to *yz* without changing the cover of *f*. [1]

The sum of all prime implicants of a Boolean function is called the **complete sum** of that function.

Simplification using Karnaugh maps

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

Truth Table.

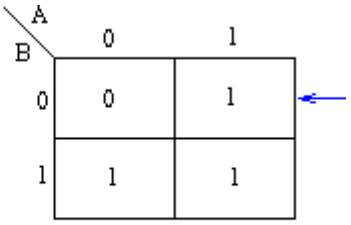
B \ A	0	1
0	a	b
1	c	d

F.

The values inside the squares are copied from the output column of the truth table, therefore there is one square in the map for every row in the truth table. Around the edge of the Karnaugh map are the values of the two input variable. A is along the top and B is down the left hand side. The diagram below explains this:

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table.

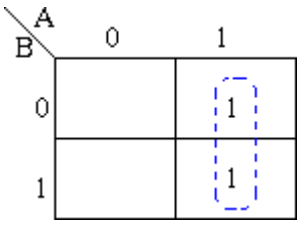


F.

The values around the edge of the map can be thought of as coordinates. So as an example, the square on the top right hand corner of the map in the above diagram has coordinates A=1 and B=0. This square corresponds to the row in the truth table where A=1 and B=0 and F=1. Note that the value in the F column represents a particular function to which the Karnaugh map corresponds.

Example 1:

Consider the following map. The function plotted is: $Z = f(A,B) = A\,\overline{B} + AB$



- Note that values of the input variables form the rows and columns. That is the logic values of the variables A and B (with one denoting true form and zero denoting false form) form the head of the rows and columns respectively.
- Bear in mind that the above map is a one dimensional type which can be used to simplify an expression in two variables.
- There is a two-dimensional map that can be used for up to four variables, and a three-dimensional map for up to six variables.

Using algebraic simplification,

$$Z = A\,\overline{B} + AB$$

$$Z = A(\overline{B} + B)$$

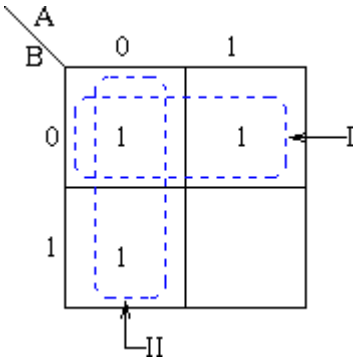
$$Z = A$$

Variable B becomes redundant due to Boolean Theorem T9a.

Referring to the map above, the two adjacent 1's are grouped together. Through inspection it can be seen that variable B has its true and false form within the group. This eliminates variable B leaving only variable A which only has its true form. The minimised answer therefore is Z = A.

Example 2:

Consider the expression Z = f(A,B) = $\bar{A}\bar{B}$+ A $\bar{B}$+ $\bar{A}$B plotted on the Karnaugh map:



Pairs of 1's are *grouped* as shown above, and the simplified answer is obtained by using the following steps:

Note that two groups can be formed for the example given above, bearing in mind that the largest rectangular clusters that can be made consist of two 1s. Notice that a 1 can belong to more than one group.
The first group labelled I, consists of two 1s which correspond to A = 0, B = 0 and A = 1, B = 0. Put in another way, all squares in this example that correspond to the area of the map where B = 0 contains 1s, independent of the value of A. So when B = 0 the output is 1. The expression of the output will contain the term $\bar{B}$

For group labelled II corresponds to the area of the map where A = 0. The group can therefore be defined as $\bar{A}$. This implies that when A = 0 the output is 1. The output is therefore 1 whenever B = 0 and A = 0
Hence the simplified answer is Z = $\bar{A}$+ $\bar{B}$

TUTORIAL-X

1

1
2

COMPARATORS

A>B

A \ B	0	1
0	0	0
1	1	0

Equation is $A>B = A.\overline{B}$

A<B

A \ B	0	1
0	0	1
1	0	0

Equation is $A<B = \overline{A}.B$

(A=B)

A \ B	0	1
0	1	0
1	0	1

The equation is $f(A=B) = \overline{A}.\overline{B} + A.B$

$= A \text{ XNOR } B$

or we can write the equation for $f(A=B)$ as $\overline{A.\overline{B} + \overline{A}.B} = \overline{f(A>B)+f(A<B)}$

4-Bit magnitude comparator

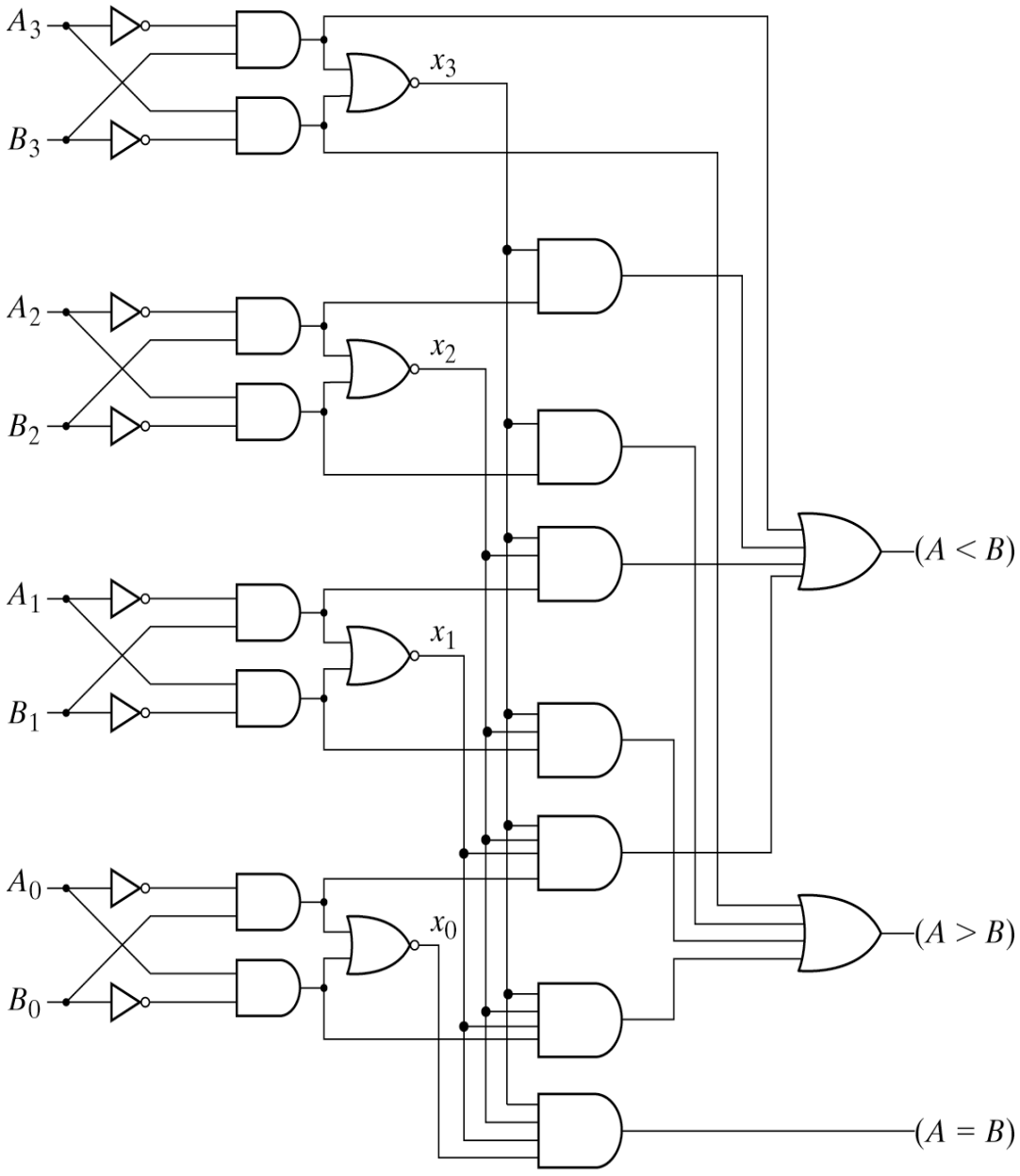
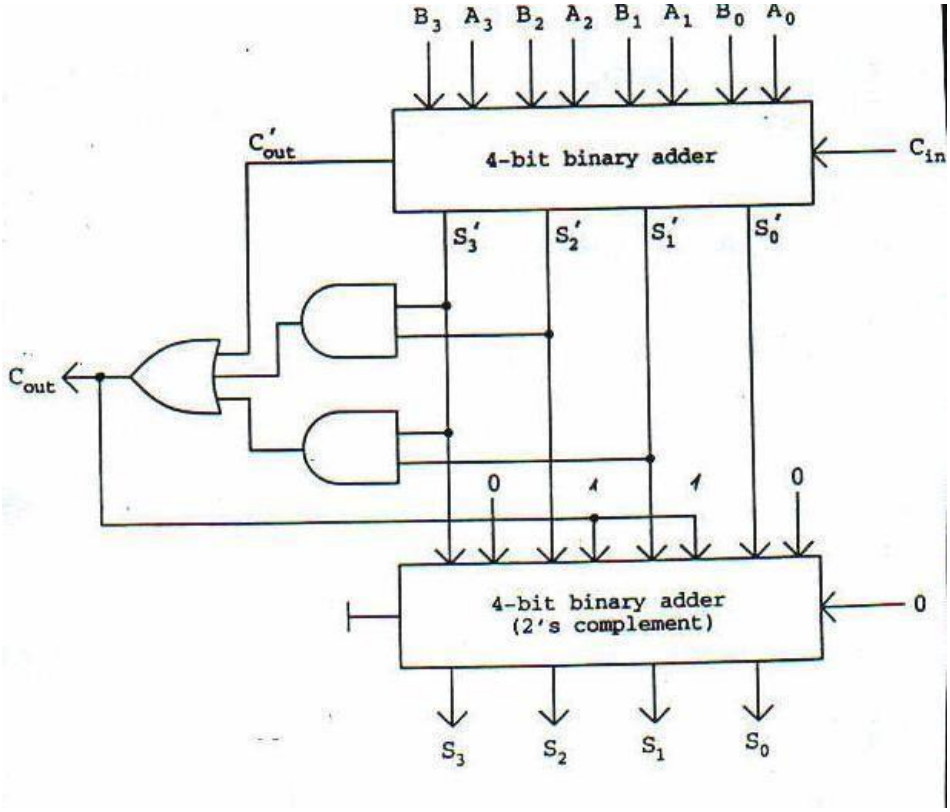
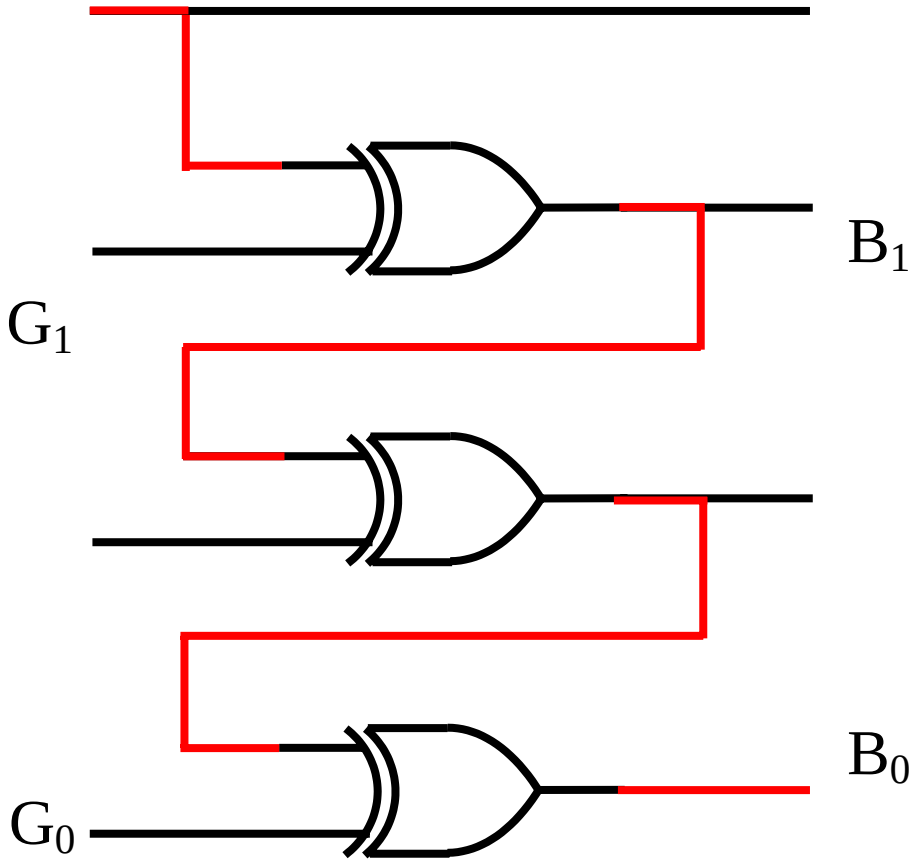


Fig. 4-17 4-Bit Magnitude Comparator

BCD adder



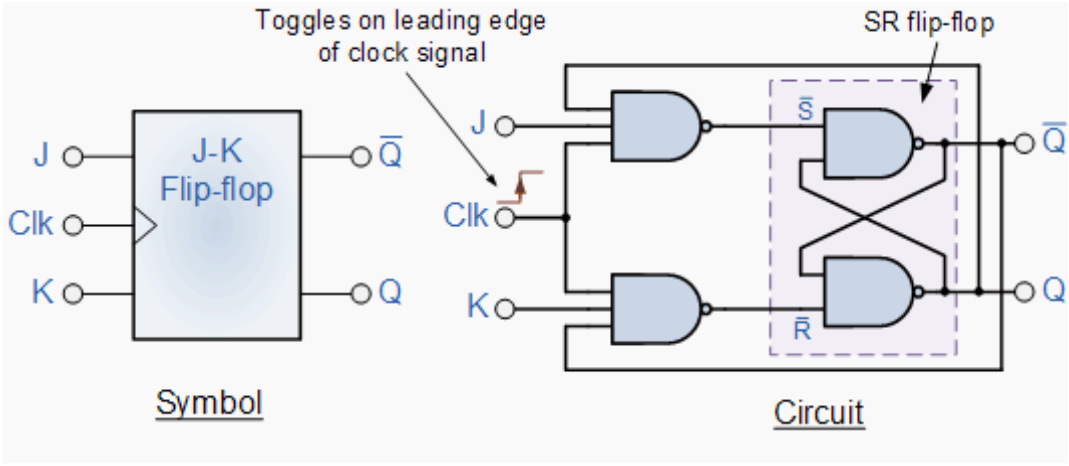
Binary to Gray code converter


B1
G1
B0
G0

TUTORIAL-XI

Design of clocked JK-FF

JK Flip-flop



Both the S and the R inputs of the previous SR bistable have now been replaced by two inputs called the J and K inputs, respectively after its inventor Jack Kilby. Then this equates to: J = S and K = R.

The two 2-input AND gates of the gated SR bistable have now been replaced by two 3-input NAND gates with the third input of each gate connected to the outputs at Q and Q. This cross coupling of the SR flip-flop allows the previously invalid condition of S = "1" and R = "1" state to be used to produce a "toggle action" as the two inputs are now interlocked. If the circuit is "SET" the J input is inhibited by the "0" status of Q through the lower NAND gate. If the circuit is "RESET" the K input is inhibited by the "0" status of Q through the upper NAND gate. As Q and Q are always different we can use them to control the input. When both inputs J and K are equal to logic "1", the JK flip-flop toggles as shown in the following truth table.

The Truth Table for the JK Function

	J	K	Q	Q	Description
same as for the SR Latch	0	0	0	0	Memory no change
	0	0	0	1	
	0	1	1	0	Reset Q » 0
	0	1	0	1	

	1	0	0	1	Set Q » 1
	1	0	1	0	
toggle action	1	1	0	1	Toggle
	1	1	1	0	

Then the JK flip-flop is basically an SR flip-flop with feedback which enables only one of its two input terminals, either SET or RESET to be active at any one time thereby eliminating the invalid condition seen previously in the SR flip-flop circuit. Also when both the J and the K inputs are at logic level "1" at the same time, and the clock input is pulsed either "HIGH", the circuit will "toggle" from its SET state to a RESET state, or visa-versa. This results in the JK flip-flop acting more like a T-type toggle flip-flop when both terminals are HIGH.

Although this circuit is an improvement on the clocked SR flip-flop it still suffers from timing problems called "race" if the output Q changes state before the timing pulse of the clock input has time to go "OFF". To avoid this the timing pulse period (T) must be kept as short as possible (high frequency). As this is sometimes not possible with modern TTL IC's the much improved **Master-Slave JK Flip-flop** was developed. This eliminates all the timing problems by using two SR flip-flops connected together in series, one for the "Master" circuit, which triggers on the leading edge of the clock pulse and the other, the "Slave" circuit, which triggers on the falling edge of the clock pulse. This results in the two sections, the master section and the slave section being enabled during opposite half-cycles of the clock signal.

The 74LS73 is a Dual JK flip-flop IC, which contains two individual JK type bistable's within a single chip enabling single or master-slave toggle flip-flops to be made. Other JK flip-flop IC's include the 74LS107 Dual JK flip-flop with clear, the 74LS109 Dual positive-edge triggered JK flip-flop and the 74LS112 Dual negative-edge triggered flip-flop with both preset and clear inputs.

Define between latch and flip-flop, Mention the similarities and differences between them

Latches and flip-flops

In the same way that gates are the building blocks of combinatorial circuits, *latches* and *flip-flops* are the building blocks of sequential circuits.

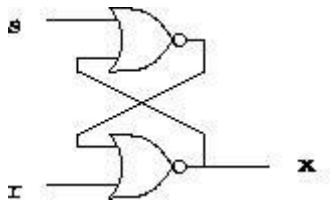
While gates had to be built directly from transistors, latches can be built from gates, and flip-flops can be built from latches. This fact will make it somewhat easier to understand latches and flip-flops.

Both latches and flip-flops are circuit elements whose output depends not only on the current inputs, but also on previous inputs and outputs. The difference between a latch and a flip-flop is that a latch does not have a *clock signal*, whereas a flip-flop always does.

Latches

How can we make a circuit out of gates that is not combinatorial? The answer is *feed-back*, which means that we create *loops* in the circuit diagrams so that output values depend, indirectly, on themselves. If such feed-back is *positive* then the circuit tends to have stable states, and if it is *negative* the circuit will tend to oscillate.

A latch has positive feedback. Here is an example of a simple latch:



This latch is called *SR-latch*, which stands for *set* and *reset*.

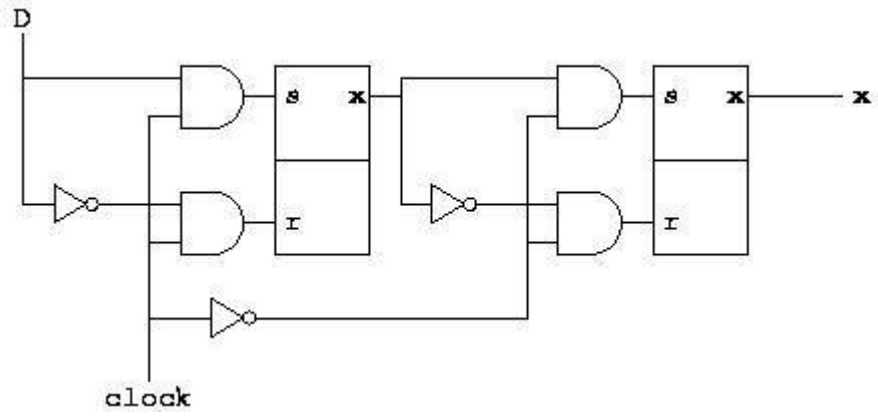
Flip-flops

Latches are *asynchronous*, which means that the output changes very soon after the input changes. Most computers today, on the other hand, are

synchronous, which means that the outputs of all the sequential circuits change simultaneously to the rhythm of a global *clock signal*.

A *flip-flop* is a synchronous version of the latch. To complicate the situation even more, there are several fundamental types of flip-flops. Here, we shall only consider a type called *master-slave* flip-flop.

In addition to the fundamental types of flip-flops, there are minor variations depending on the number of inputs and how they control the state of the flip-flop. Here, we shall only consider a very simple type of flip-flop called a *D-flip-flop*. A master-slave D-flip-flop is built from two SR-latches and some gates. Here is the circuit diagram:

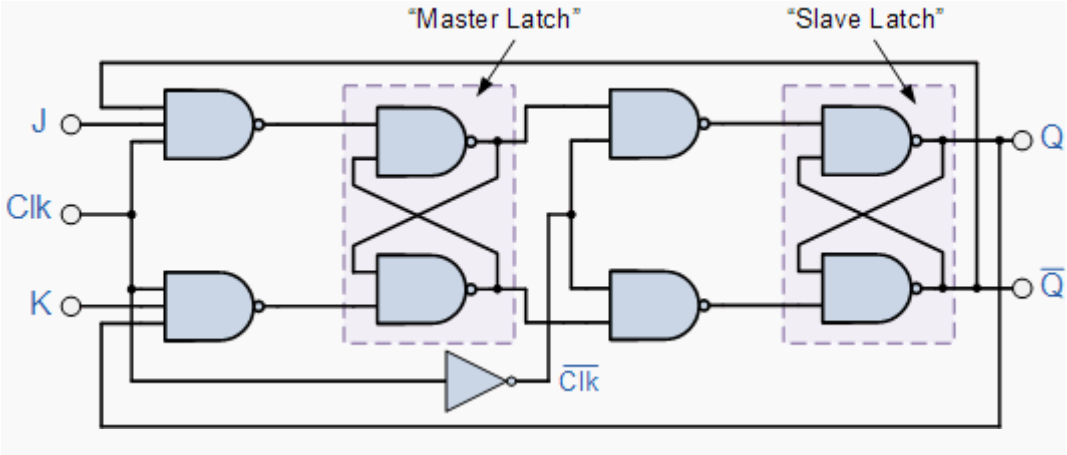


The leftmost SR-latch is called the *master* and the rightmost is called the *slave*.

What is master slave flip flop design a clocked master slave JK flip-flop

The **Master-Slave Flip-Flop** is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse. The outputs from Q and Q from the "Slave" flip-flop are fed back to the inputs of the "Master" with the outputs of the "Master" flip-flop being connected to the two inputs of the "Slave" flip-flop. This feedback configuration from the slave's output to the master's input gives the characteristic toggle of the JK flip-flop as shown below.

The Master-Slave JK Flip-Flop



The input signals J and K are connected to the gated "master" SR flip-flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip-flop is the inverse (complement) of the "master" clock input, the "slave" SR flip-flop does not toggle. The outputs from the "master" flip-flop are only "seen" by the gated "slave" flip-flop when the clock input goes "LOW" to logic level "0". When the clock is "LOW", the outputs from the "master" flip-flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip-flop now responds to the state of its inputs passed over by the "master" section. Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flip-flop are fed through to the gated inputs of the "slave" flip-flop and on the "High-to-Low" transition the same inputs are reflected on the output of the "slave" making this type of flip-flop edge or pulse-triggered.

Then, the circuit accepts input data when the clock signal is "HIGH", and passes the data to the output on the falling-edge of the clock signal. In other words, the **Master-Slave JK Flip-flop** is a "Synchronous" device as it only passes data with the timing of the clock signal..

TUTORIAL-XII

Morse and Mealy machines and there comparison

Objectives

There are two basic ways to design clocked sequential circuits.
These are using:

1. Mealy Machine, which we have seen so far.

2. Moore Machine.

The objectives of this lesson are:

1. Study Mealy and Moore machines
2. Comparison of the two machine types
3. Timing diagram and state machines

Mealy Machine

In a Mealy machine, the outputs are a function of the present state and the value of the inputs as shown in Figure 1.

Accordingly, the outputs may change asynchronously in response to any change in the inputs.

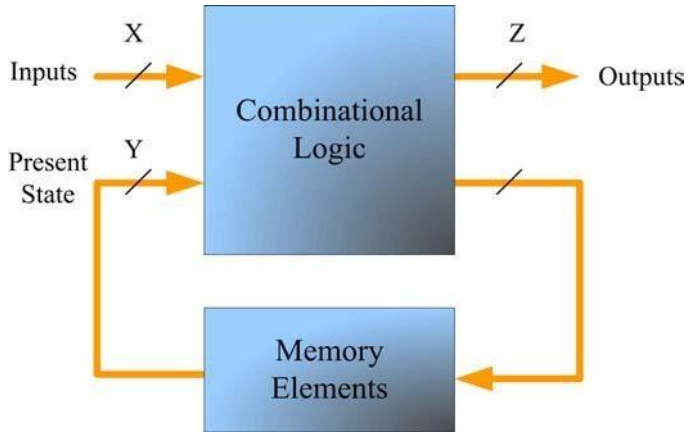


Figure 1: Mealy Type Machine

Mealy Machine

In a Moore machine the outputs depend only on the present state as shown in

Figure 2.

A combinational logic block maps the inputs and the current

state into the necessary flip-flop inputs to store the appropriate next state just like Mealy machine.

However, the outputs are computed by a combinational logic block whose inputs are only the flip-flops state outputs.

The outputs change synchronously with the state transition triggered by the

active clock edge.

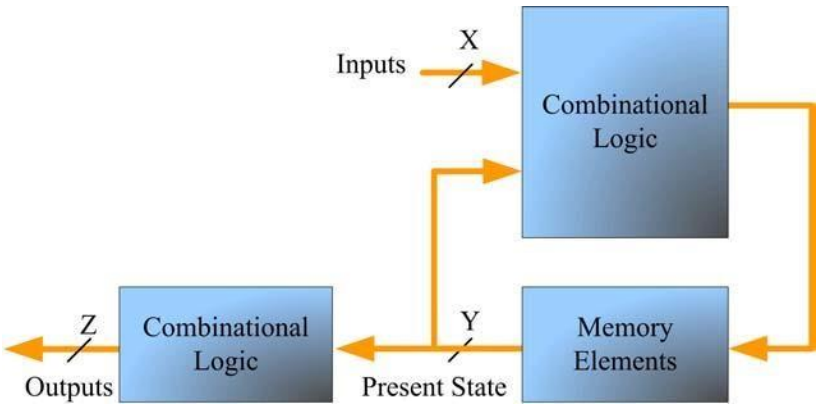


Figure 2: Moore Type Machine

Comparison of the Two Machine Types

Consider a finite state machine that checks for a pattern of '10' and asserts

logic high when it is detected.

The state diagram representations for the Mealy and Moore machines are shown in Figure 3.

The state diagram of the Mealy machine lists the inputs with their associated outputs on state transitions arcs.

The value stated on the arrows for Mealy machine is of the form Zi/Xi where

Zi represents input value and Xi represents output value.

A Moore machine produces a unique output for every state irrespective of

Inputs.

Accordingly the state diagram of the Moore machine associates the output with the state in the form state-notation/output-value.

The state transition arrows of Moore machine are labeled with the input value that triggers such transition.

Since a Mealy machine associates outputs with transitions, an output sequence can be generated in fewer states using Mealy machine as compared to Moore machine. This was illustrated in the previous example.

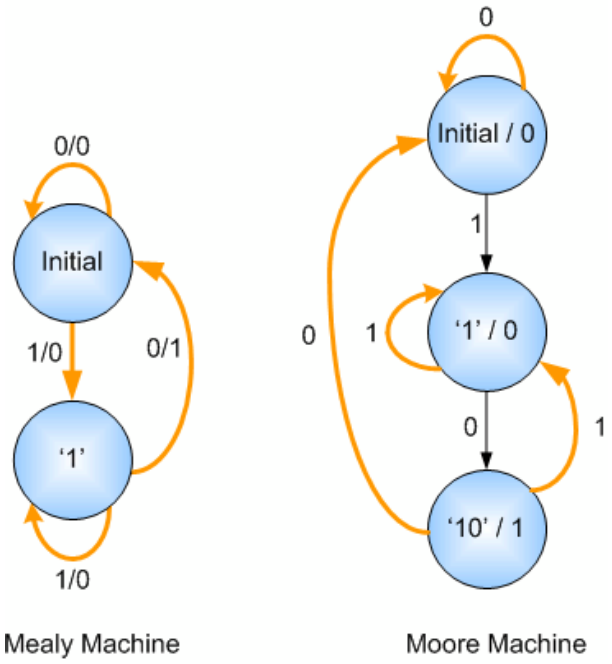


Figure 3: Mealy and Moore State Diagrams for '10' Sequence Detector

Procedure of state minimization using merger graph and merger table

Example problem

Design a gated latch circuit with two inputs, *G* (gate) and *D* (data), and one output *Q*. The gated latch is a memory element that accepts the value of *D* when *G* = 1 and retains this value after *G* goes to 0. Once *G* = 0, a change in *D* does not change the value of the output *Q*.

Solution

State table

State	Inputs		Output
	D	G	Q
a	0	1	0
b	1	1	1
c	0	0	0
d	1	0	0
e	1	0	1
f	0	0	1

Primitive Flow table

	DG = 00	DG = 01	DG = 11	DG = 10
a	*c* , –	(*a*), 0	*b* , –	– , –
b	– , –	*a* , –	(*b*), 1	*e* , –
c	(*c*), 0	*a* , –	– , –	*d* , –
d	*c* , –	– , –	*b* , –	(*d*), 0
e	*f* , –	– , –	*b* , –	(*e*), 1
f	(*f*), 1	*a* , –	– , –	*e* , –

Informal Merging

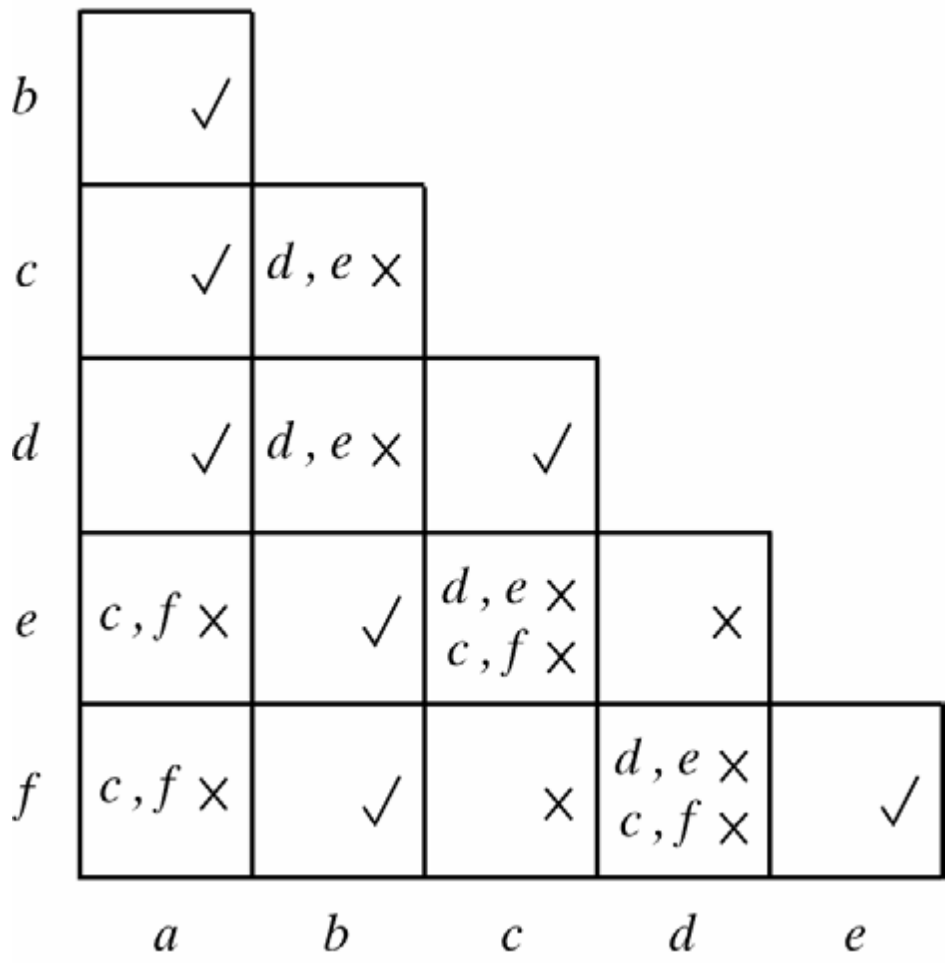
	DG 00	01	11	10
a, c, d	(*c*), 0	(*a*), 0	*b* , –	(*d*), 0
b, e, f	(*f*), 1	*a* , –	(*b*), 1	(*e*), 1

	DG 00	01	11	10
a	(*a*), 0	(*a*), 0	*b* , –	(*a*), 0
b	(*b*), 1	*a* , –	(*b*), 1	(*b*), 1

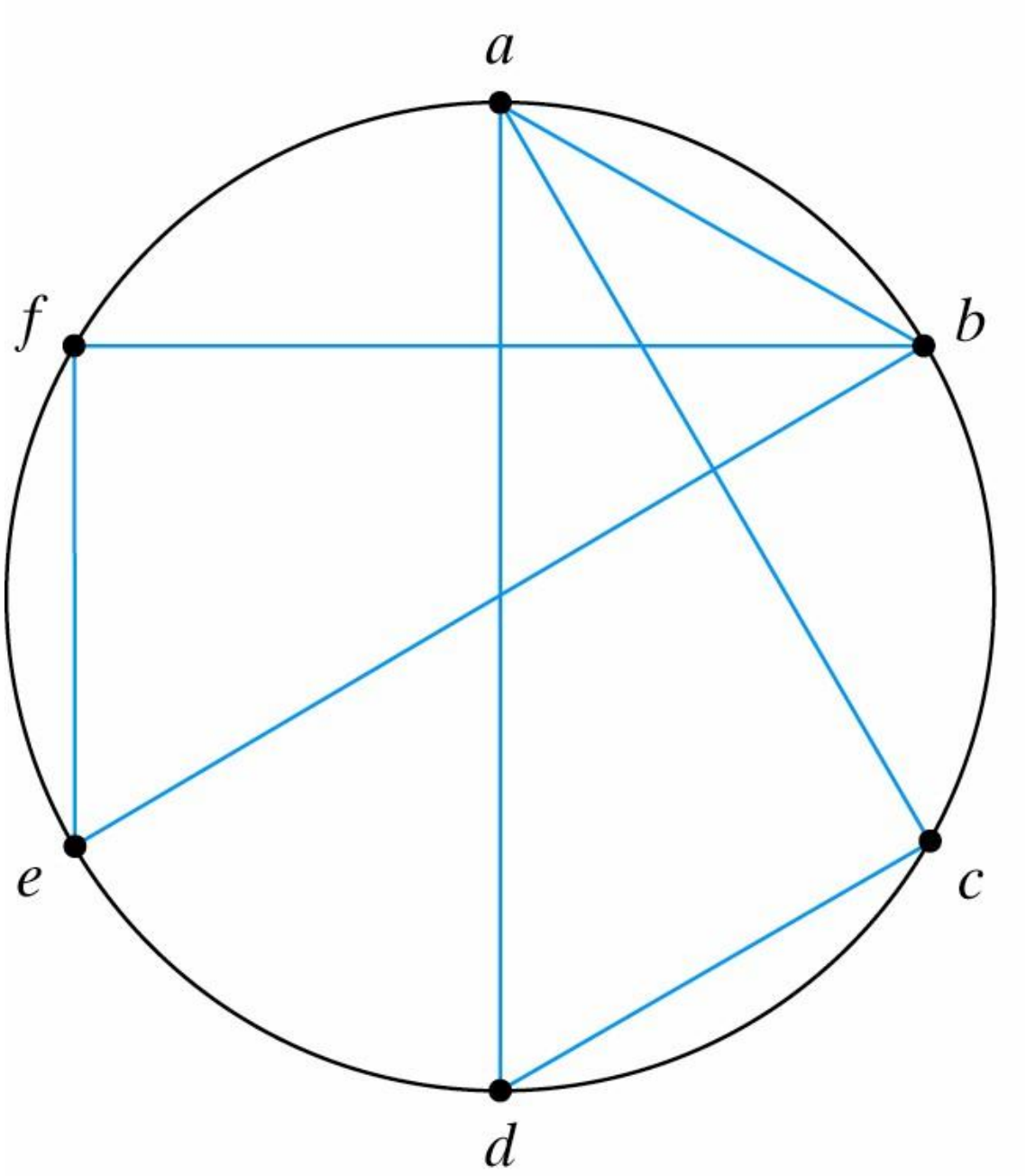
(b) Reduced table (two alternatives)

Formal Merging

Compatible Pairs



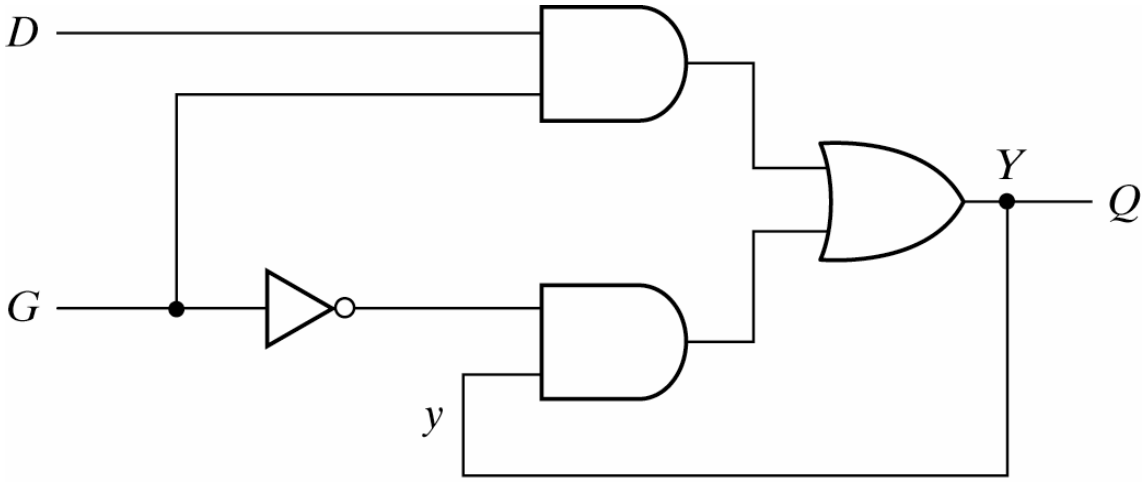
Maximal Compatibles



Reduced Table

	DG = 00	DG = 01	DG = 11	DG = 10
a	a, 0	a, 0	b , –	a, 0
b	b, 1	a , –	b, 1	b, 1

Logic Diagram



Draw the diagram of mealy type FSM for serial adder

A serial adder is a digital circuit that can add any two arbitrarily large numbers using a single full adder. Just as humans, the serial adder operates on one pair of bits/digits at a time. When you add the two 4–digit numbers 7852 and 1974, for example, you typically start by adding 2 plus 4 equal 6, then 5 plus 7 equal 12 (place 2 and carry the 1), and so on. Similarly, given the two 4–bit numbers 1011 and 0110, the serial adder starts by adding 1 plus 0 equal to 1, and then 1 plus 1 equal to 10 (place 0 and carry the 1), and so on.

For a general demonstration, both a human person and a serial adder follow the same sequential method. Given two 4–figure numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, we add two figures at a time starting with the least significant pair, and so on. First, we do $A_0 + B_0 = S_0$. Second, we do $A_1 + B_1 + \text{carry} = S_1$, and so on; where the S figures represent the sum: A + B = S.

Notice that in the operation $A_1 + B_1 + \text{carry} = S1$, carry is not one of the inputs being added; the inputs being added are A_1 and B_1. Furthermore, the value of carry does not depend on the inputs A_1 and B_1. Carry is simply a given condition, the consequence of something that happened in the past; namely, $A_0 + B_0$.

Therefore, if we were tasked to "build a circuit that can add any two binary numbers using the sequential method that humans use," we would treat the carry variable as a state variable. (In computer engineering talk, any circuit with one or more state variables is referred to as a finite state machine.)

Since the carry variable can either be 1 or 0, we say that our circuit will be a two states machine. When the circuit is in the state where carry = 0, the relationship between the inputs A and B and the output S is such that: if AB = 00 then S = 0; if AB = 01 then S = 1; if AB = 10 then S = 1; and if AB = 11 then S = 0. When the circuit is in the state where carry = 1, it also follows that: if AB = 00 then S = 1; if AB = 01 then S = 0; if AB = 10 then S = 0; and if AB = 11 then S = 1. We illustrate these relationships in the state diagram in Figure 1.

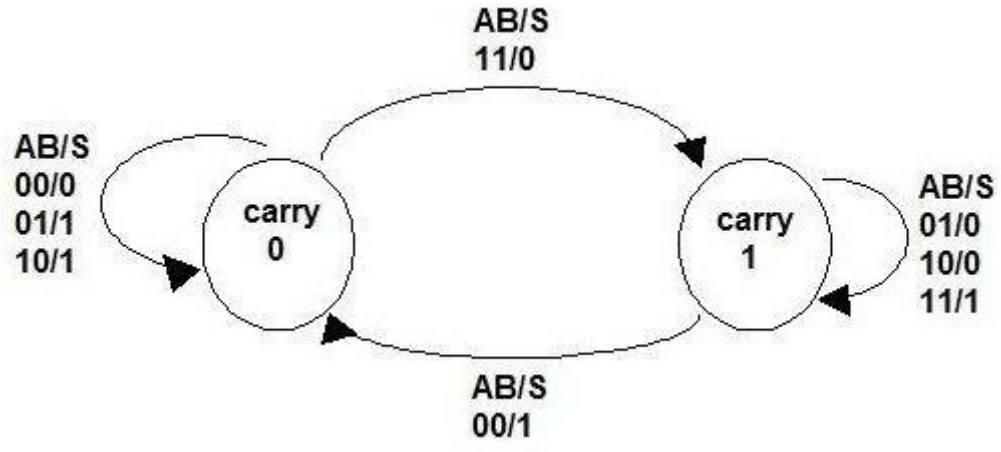


Figure 1: *State transition diagram for serial adder FSM*

To present the information in the state diagram in table form, we re-label the carry variable Z (Z for carry–out and z for carry–in) for convenience. We show the tabulated information in Table 1 below.
From a finite state machine analysis perspective, we say z is the present state of the machine because z is presently available as one of the inputs to the full adder; Z on the other hand is the next state because it is one of the variables we are solving for — given the inputs A, B and the present state (or the carry–in) z.

Given state of doo	AB = 0	AB = 0	AB = 1	AB = 1	AB = 0	AB = 0	AB = 1	AB = 1
	Next state				Output			
z	Z				S			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Table 1: State transition table for serial adder FSM

At this point we can formulate the Boolean expressions for S and Z, where S is the sum output bit and Z is the carry output bit. Just in case you can't see the Boolean functions in the Table1, we recast the transition table as two K–maps in Table 2 for your convenience.

K-map For Z					K-map For S				
z/AB	00	01	10	11	z/AB	00	01	10	11
0	0	0	0	1	0	0	1	1	0
1	0	1	1	1	1	1	0	0	1

Table 2: K-maps for the next state variable Z and the output variable S

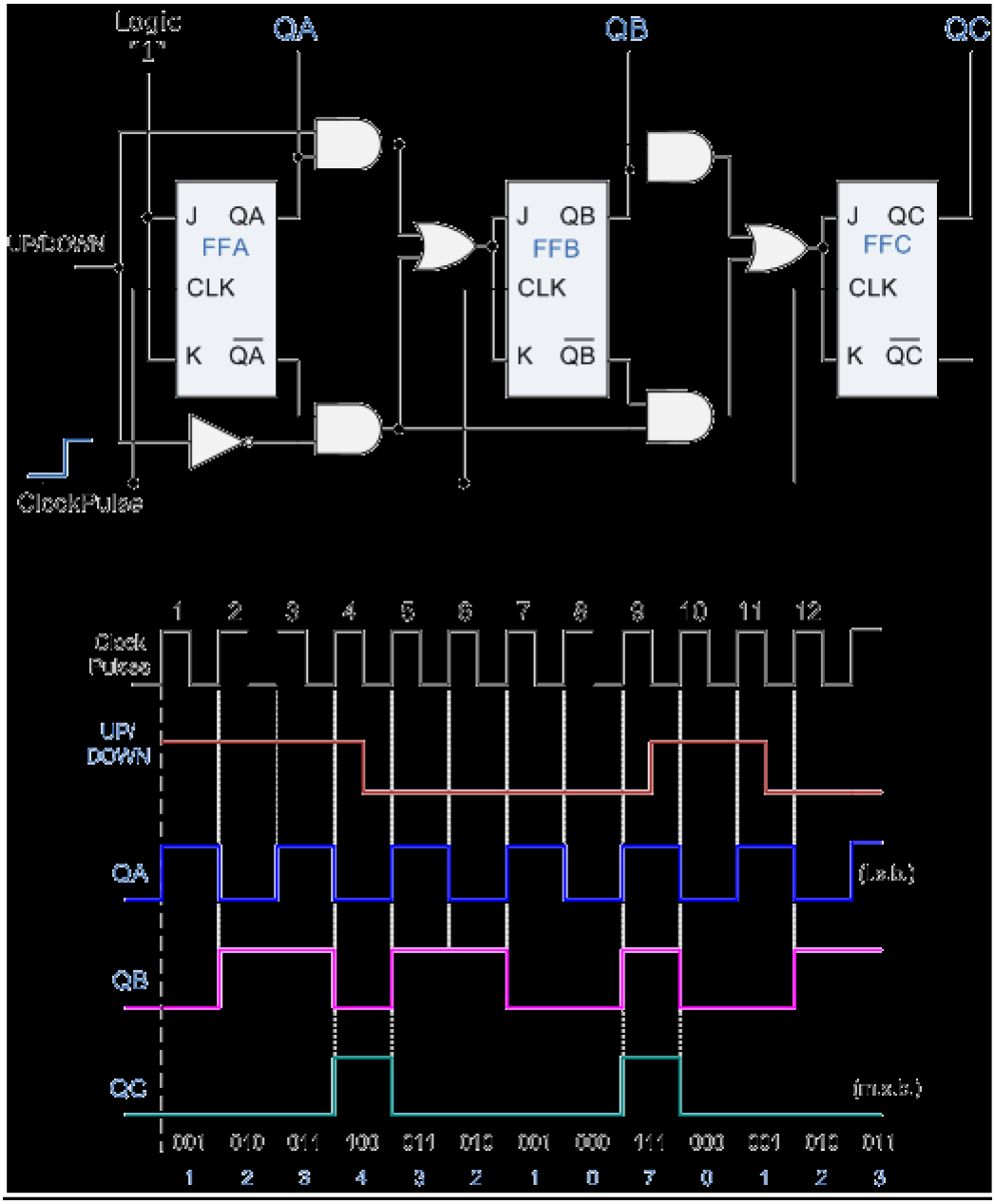
$S = A \oplus B \oplus z$

$Z = A \bullet B + A \bullet z + B \bullet z$

The reason these Boolean expressions look similar to the full adder equation is because they are the full adder expression. Here z is the carry–in signal and Z is the carry–out signal. Since the carry–out of the full adder becomes the carry–in to the full adder on the next operation, we us a D flipflop to save the carry signal. We use a D flipflop because we need the data in Z to pass to z intact for the next operation. Any other flipflop will return some z that may or may not be equal to Z.

TUTORIAL-XIII

Draw ASM chart for 3-bit updown counter



The circuit above is of a simple 3-bit Up/Down synchronous counter using JK flip-flops configured to operate as toggle or T-type flip-flops giving a maximum count of zero (000) to seven (111) and back to zero again. Then the 3-Bit counter advances upward in sequence (0,1,2,3,4,5,6,7) or downwards in reverse sequence (7,6,5,4,3,2,1,0) but generally, bidirectional counters can be made to change their count direction at any point in the counting sequence. An additional

input determines the direction of the count, either Up or Down and the timing diagram gives an example of the counters operation as this Up/Down input changes state.

Nowadays, both up and down counters are incorporated into single IC that is fully programmable to count in both an "Up" and a "Down" direction from any preset value producing a complete **Bidirectional Counter** chip. Common chips available are the 74HC190 4-bit BCD decade Up/Down counter, the 74F569 is a fully synchronous Up/Down binary counter and the CMOS 4029 4-bit Synchronous Up/Down counter.

UNIT – V

SAMPLING GATES AND LOGIC GATES

IC families:

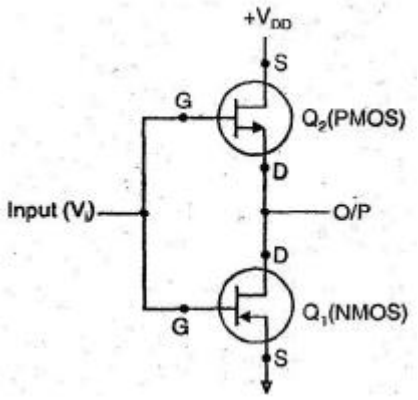
comparison of the important characteristics of various IC logic families.

	Parameter	RTL	I^2L	DTL	HTL	TTL	ECL	MOS	CMOS
1.	Basic Gate	NOR	NOR	NAND	NAND	NAND	OR-NOR	NAND	NOR or NAND
2.	Fan-out	5	Depends on injector current	8	10	10 to 20	25	20	20 to 50
3.	Power dissip-ation in mW	12	6mm to 70 μM	8-12	55	10	40-55	0.2-10	0.0025
4.	Noise immunity	Nominal	Poor	Good	Excellent	Very Good	Poor	Good	Very Good
5.	Propagation delay (in sec.)	12	25-250	30	90	10	0.75	300	70.0
6.	Clock rate (MHZ)	8	–	72	4	35	>60	2	10
7.	Available functions	High	LSI only	Fairly high	Nominal	Very high	High	low	High

(i) CMOS inverter

(ii) Tristate logic

(i) CMOS Inverter: It is complementary MOSFET obtained by using P-channel MOSPET and n-channel MOSFET simultaneously. The P and N channel are connected in series, their drains are connected together, output is taken from common drain point. Input is applied at common gate terminal. CMOS is very fast and consumes less power.



Case 1. When input Vi = 0. The V_{GS} (Gate source) voltage of Q1 will be 0 volt, it will be off. But Q2 will be ON; Hence output will be equal to +VDD or logic 1.

Case 2. When input Vi = 1, The V_{GS} (Gate source) voltage of Q2 will be 0 volt, it will be OFF, But Q1 will be ON. Hence output will be connected to•
ground or logic 0.

In this way, CMOS function as an inverter.

(ii) Tri-state logic: When there are three states i.e. state 0, state 1 and high impendence i.e. called Tri-state logic. High impedance is considered as state when no current pass through circuit. Although in state 0 and state 1 circuit functions and current flows through it.

- Propagation delay is the average transition delay time for a pulse to propagate from input to output of a switching circuit.
- Fan-in is the number of inputs to the gate which it can handle.
- Fan-out is the number of loads the output of a gate can drive without effecting its operation.
- Power dissipation is the supply voltage required by the gate to operate with 50% duty cycle at a given frequency
- RTL, DTL, DTL are the logic families which are now obsolete.
- TTL is the most widely used logic family.
- TTL gates may be:
  (a) Totem pole
  (b) Open collector
  (c) Tri-state .
- TTL is used in SSI and MSI Integrated circuits and is the fastest of all standard logic families.
- Totem pole TTL has the advantage of high speed and low power dissipation but its disadvantage is that it cannot be wired ANDed because of current spikes generation.
- Tri-state has three states : .
  (a) High

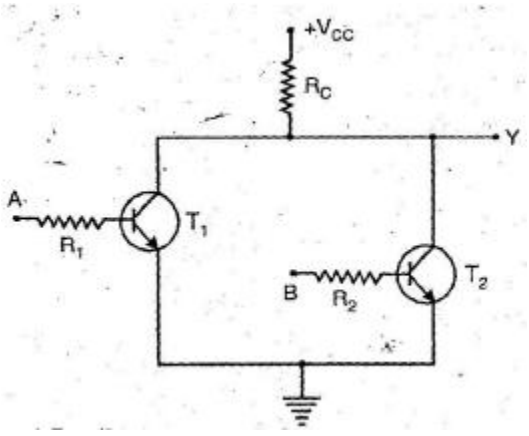
(b) Low

(c) High Impedance

- ECL is the fastest of all logic families because its propagation delay is very small i.e. of about 2 nsec.
- ECL can be wired ORed.
- MOS logic is the simplest to fabricate.
- MOS transistor can be connected as a resistor.
- MOSFET circuitry are normally constructed from NMOS devices because they are 3 times faster than PMOS devices.
- CMOS uses both P-MOS and N-MOS.
- CMOS needs less power as compared to ECL as they need maximum power.
- Both NMOS and PMOS are more economical than CMOS because of their greater packing densities.
- Speed of CMOS gates increases with increase in VDD.
- CMOS has large fan-out because of its low output resistance.

Schematic of RTL NOR gate and explain its operation.

RTL was the first to introduced. RTL NOR gate is as shown in fig.



Working:

Case I: When A = B = 0.

Both T1 and T2 transistors are in cut off state because the voltage is insufficient to drive the transistors i.e. VBE < 0.6 V: Thus, output Y will be high, approximately equal to supply voltage Vcc. As no current flows through Rc and drop across Rc is also zero.

Thus, Y = 1, when A = B = 0.

Case II : When A = 0 and B = 1 or A = 1 and B = 0.

The transistor whose input is high goes into saturation where as other will goes to off cut state. This positive input to transistor increases the voltage drop across the collector resistor and decreasing the positive output voltage.

Thus, Y = 0,when A= 0 and B = 1 or A = 1 and B = 0.

Case III : When A = B = 1**.** Both the transistors T1 and T2 goes into saturation and output voltage is equal to saturation voltage.

Thus, Y = 0,when A = B = 1

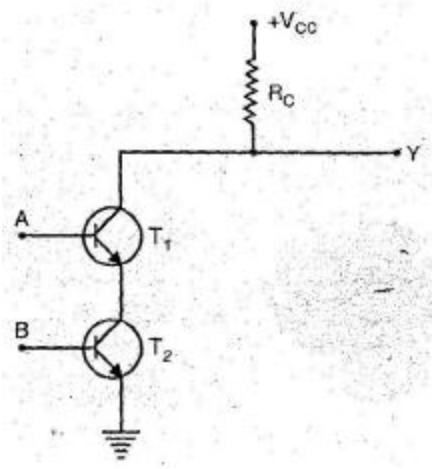
Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Which is the output of NOR gate.

DCTL NAND gate with the help of suitable circuit diagram.

DCTL NAND gate circuit diagram is as shown:



Working

Case I: When A = B = 0. Both transistors T1 and T2 goes to cut off state. As the voltage is not sufficient to drive the transistor into saturation. Thus, the output voltage equal to Vcc.

When A = B = 0, output Y = 1

Case II: When A = 0 and B = 1 or A = 1 and, B = 0. The corresponding transistor goes to cut off state and the output voltage equals to Vcc.

Thus, When A = 0 and B = 1 or A = 1 and B = 0, Output Y = 1.

Case III: When A = B = 1. Both transistors T1 and T2 goes into saturation state and output voltage is insufficient to consider as '1'

Thus when A B = 1, output Y = 0.

Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Which is the output of NAND gate.

Compare standard TTL, Low power TTL and high speed TTL logic families.

Name of the Logic Family	Propagation Delay (ns)	Power Dissipation (mW)	Fan out	Max. Clock Rate (MHz)
1. Standard TTL	9	10	10	35
2. Low power TTL	33	1	20	3
3. High sped TTL	6	23	10	50

characteristics and specification of CMOS.

1 Power supply (VDD) = 3 — 15 Volts

2. Power dissipation (Pd) = 10 nW

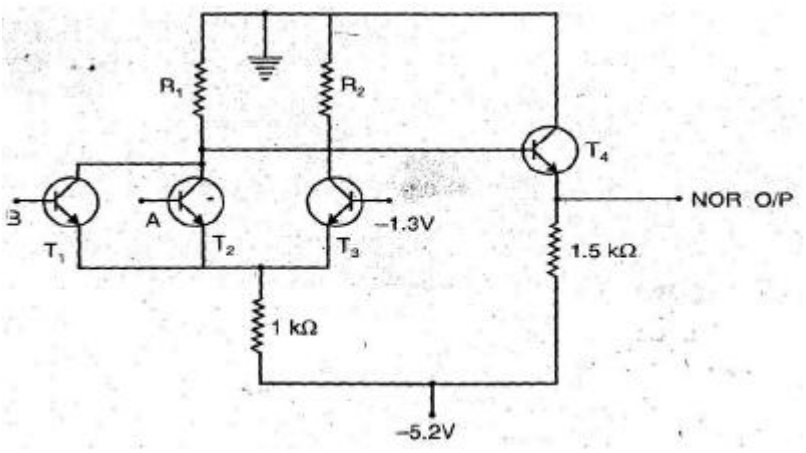
3. Propagation delay (td) = 25 ns

4. Noise margine (NM) = 45% of

VDD 5, Fan out (FO) = >50

Two input ECL NOR gate

The circuit diagram of two input ECL NOR gate is as shown:



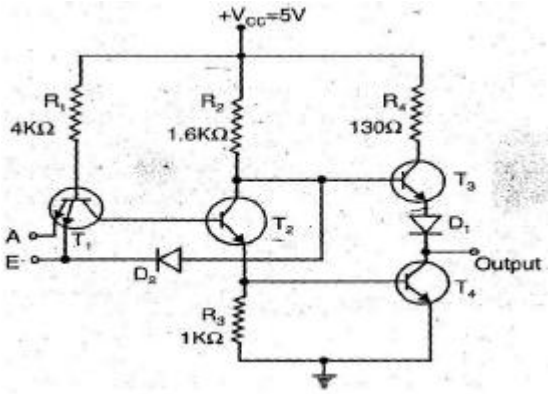
Working

Case I : When A = B = 0, the reference voltage of T3 is more forward biased then T1 and T2. Thus, T3 is ON and T1, T2 remains OFF. The value of R1 is such-that the output of NOR gate is high .i.e. '1'.

Case II: When A = 1 or B = 1 or A = B = 1, the corresponding transistors are ON, as they are more forward biased that T3 and thus T3 is OFF. Which makes the NOR output to be low i.e. '0'.

This shows that the circuit works as a NOR gate.

TTL inverter.



Tristate TTL inverter utilizer the high-speed operation of totem-pole arrangement while permitting outputs to be wired ANDed (connected together). It is called tristate TTL because it allows three possible output stages. HIGH, LOW and High-Impedance. We know that transistor T3 is ON when output is HIGH and T4 is ON when output is LOW. In the high impedance state both transistors, transistor T3 and T4 in the totem pole arrangement are med OFF. As a result the output is open or floating, it is neither LOW nor HIGH.

The above fig. shows the simplified tristate inverter. It has two inputs A and E. A is the normal logic input whereas E is an ENABLE input. When ENABLE input is HIGH, the circuit works as a normal inverter. Because when E is HIGH, the state-of the transistor T1 (either ON or OFF) depends on the logic input A and the additional component diode is open circuited as cathode is at logic HIGH. When ENABLE input is LOW, regardless of the state of logic input the base-emitter junction of T is forward biased and as a result it turns ON. This shunts the current through R1 away from T2 making it OFF. As T2 is OFF, there is no sufficient drive for T4 conduct and hence T4 turns OFF. The LOW at ENABLE input also forward biases diode D2, which shunt the current away from the base of T3, making it OFF. In this way, when ENABLE output is LOW, both transistors are OFF and output is at high impedance state.

ECL OR gate

ECL or gate : Emitter-coupled logic (ECL) is the fastest of all logic families and thus it is used in applications where very high speed is essential. High speeds have become possible in ECL because the transistors are used in difference amplifier configuration, in which they are never driven into saturation and thereby the storage time is eliminated. Here, rather than switching the transistors from ON to OFF and vice-versa, they are switched between cut-off and active regions. Propagation delays of less than 1 ns per gate have become possible in ECL.

Basically, ECL is realized using difference amplifier in which the emitters of the two transistors are connected and hence it referred to as emitter-coupled logic. A 3-input ECL gate is shown in Fig. (A) which has three parts. The middle part is the difference amplifier which performs the logic operation.

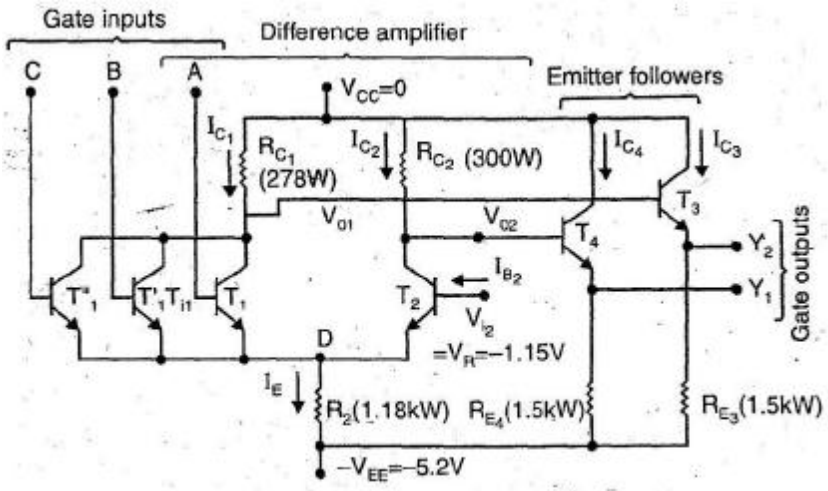


Fig. (A) A 3-input ECL OR/NOR Gate

Emitter followers are used for d.c. level shifting of the outputs, so that V (0) and V (1) are same for the inputs and the outputs. Note that two output Y1 and Y2 are available in this circuit which are complementary. Y1. corresponds to OR logic and Y2 to NOR logic and hence it is named as an OR/NOR gate.

Additional transistors are used in parallel to T1 to get the required fan-in. There is a fundamental difference between all other logic families (including MOS logic) and ECL as far as the supply voltage is concerned. In ECL, the positive end of the supply is connected to ground in contrast to other logic families in which negative end of the supply is grounded. This is done to minimize the effect of noise induced in the power supply and protection of the gate from an accidental short circuit developing between the output of a gate and ground. The voltage corresponding to V (0) and V (1) are both negative due to positive end of the supply being connected to ground. The symbol of an ECL OR/NOR gate is shown in Fig. (B)

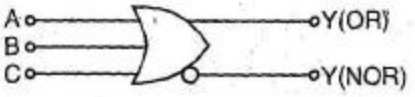
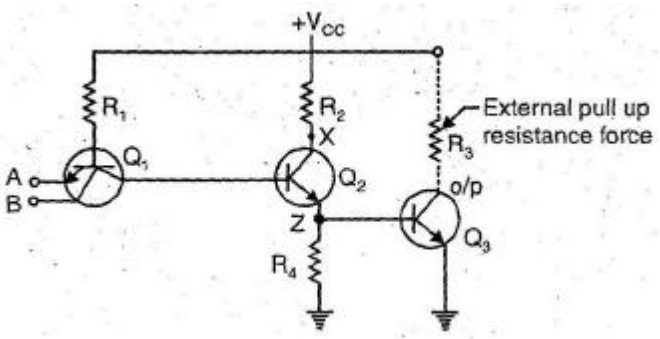


Fig. (B) The Symbol for a 3-input OR/NOR Gate

Open collector TTL NAND gate and explain its operation

The circuit diagram of 2-input NAND gate open-collector TTL gate is as shown:



Working:

Case.1 : When A = 0,B = 0

When both inputs A and B are low, both functions of Q1 are forward biased and Q2 remains off. So no current flows through R4 and Q3 is also off and its collector voltage is equal to Vcc i.e. Y =1

Case2 : When A = 0, B = 1 and

Case 3: When A = 1, B = 0

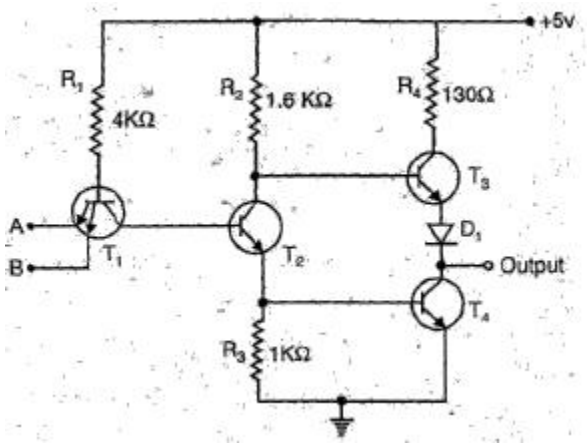
When one input is high and. other is low, then one junction is forward biased so Q2 is off and Q3 is also off. So collector voltage is equal to Vcc i.e. Y = 1

Case 4: When A = 1, B = 1

When both inputs are high, Q1 is turned off and Q2 turned 'ON' Q3 goes into saturation and hence Y = 0. The open-collector output has main advantage that wired ANDing is possible in it.

TTL NAND gate

Two input TTL NAND gate-is given in fig. (1). In this transistor T3 and T4 form a totem pole. Such type of configuration is called-as totem-pole output or active pull up output.



So, when A = 0 and B = 1 or (+5V). T1 conducts and T2 switch off. Since T2 is like an open switch, no current flows through it. But the current flows through the resistor R2 and into the base of transistor T3 to turn it ON. T4 remains OFF because there is no path through which it can receive base current. The output current flows through resistor R4 and diode D1. Thus, we get high' output.

When both inputs are high i.e. A = B = 1 or (+ 5V), T2 is ON and it drives T4 turning it ON. It is noted that the voltage at the base of T3 equals the sum of the base to emitter drop of

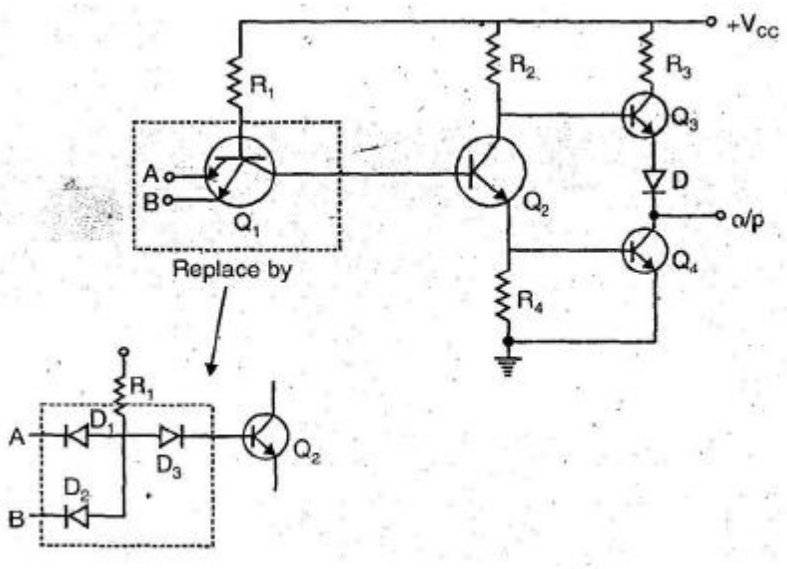
T4 and $V_{CE\ (Sat)}$ of T2.

The diode D1 does not allow base-emitter junction of T3 to be forward-biased and hence, T3 remains OFF when T4 is ON. Thus, we get low output.

It works as TTL NAND gate.

Totem pole NAND gate

In TTL Totem pole NAND gate, multiple emitter transistor as input is used. The no. of inputs may be from 2 to 8 emitters. The circuit diagram is as shown



Case 1:

When A = 0, B = 0

Now D1 and D2 both conduct, hence D3 will be off and make Q2 off. So its collector voltage rises and make Q3 ‘ON’ and Q4 off; Hence output at Y = 1 (High)

Case 2 and Case 3:

If A = 0, B = 1 and A = 1, B=0

In both cases, the diode corresponding to low input will conduct and hence diode P3 will be OFF making Q2 OFF. In a similar way its collector voltage rises Q3 ‘ON’ and Q4 ‘OFF’. Hence output voltage Y = 1 (High).

Case 4: A = 1, B = 1

Both diodes D1 and D2 will be off. D3 will be ‘ON’ and Q2 will ‘ON’ making Q4 also ‘ON’.

But Q3 will be ‘OFF’. So output voltage Y = 0.

All the four cases shows that circuit operates as a NAND gate.

Totem pole can’t be Wired ANDed due to current spike problem. The transistors used in circuits may get damaged over a period of time though not immediately. Sometimes voltage level rises high than the allowable.