Semester 6 DIGITAL ELECTRONICS- core subject -10 Credit-4

Unit I

Number system, Binary, decimal, octal, hexadecimal-conversion from one another-binary addition, subtraction, multiplication, division-binary subtraction by 1,s and complement and 2's complement. Weighted binary code- BCD or 8421 code- basic laws of Boolean algebra- Boolean addition. Multiplication-properties of Boolean algebra-principle of Duality-De Morgan's theorems-their proof-sum of products (SOP) – expression from a truth table- Karnaugh map- 2 variable-3 variable, - simplification using K-map.

Unit II

Positive and negative logic- logic gates- OR,AND,NAT,NOR,NAND and EX-OR – Universal gates- logic families-Diode resistor logic (DRL)- Or gate, AND gate-RTL Not gate-DLT NOR<TTL NOR-DTL NAND –Totem pole TTL NAND .

Unit III

Half adder-full adder – 4 bit binary adder-half subtractor- full subtractor- 4 bit parallel subtractor – multiplexer (MUX)- 4 to 1 MUX de-multiplexer(DMUX)- 1 to 4DMUX-Encoder-8to 3 encoder- Decimal tyo BCD encoder –Decoder-3 to 8 decoder, BCD to decimal decoder, BCD to decimal decoder-BCD to seven segment decoder.

Unit IV

Timer- IC 555 mono and astable multivibrator –Flip flops -RS flip flops-clocked RS flip flops-JK flip flops-JK Master slave flip flop- D flip flop and T flip flop- Application of flip flops parallel data storage.

Unit V

Register-Shift Register-classification-serial in serial out (SISO) shift register-counters-ring counter, 4 bit binary counter- Decade counter – digital to analog counter (D/A0 – binary ladder type- analog to digital converter (A/D)- successive approximation type – quality study of microprocessor.

Books for study:

1.Digital Principles and applications 6th ediction; Leech,Malvino and Saha.
2. Digital Fundamentals ; V.Vijayandran S. Viswanathan Publication – 2007.
3. Fundamentals of Digital Electronics and microprocessor S.Chand and coy-2005.

Reference books:

1. Introduction o integrated electronics digital and analog V.Vijayandran S. Viswanathan Publication – 2007.
2. . Electrnics devices and circuits Salivaganan, Suresh kumar and vallavaraj.

DIGITAL ELECTRONICS

UNIT-I

NUMBER SYSTEM:

The Decimal Number System

The Decimal Number System uses base 10. It includes the digits from 0 through 9. The weighted values for each position are as follows:

10^4	10^3	10^2	10^1	10^0	10^-1	10^-2	10^-3
10000	1000	100	10	1	.1	.01	.001

You have been using the decimal(base 10) numbering system for so long that you often take it for granted. When you see a number like "123", you don't think about the value123. Instead, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents:

1 * 10^2 + 2 * 10^1 + 3 * 10^0 =

1 * 100 + 2 * 10 + 3 * 1 =

100 + 20 + 3 =

123

Each digit appearing to the left of the decimal point represents a value between zero and nine times power of ten represented by its position in the number. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten. For example, the value 725.194 is represented as follows:

7 * 10^2 + 2 * 10^1 + 5 * 10^0 + 1 * 10^-1 + 9 * 10^-2 + 4 * 10^-3 =

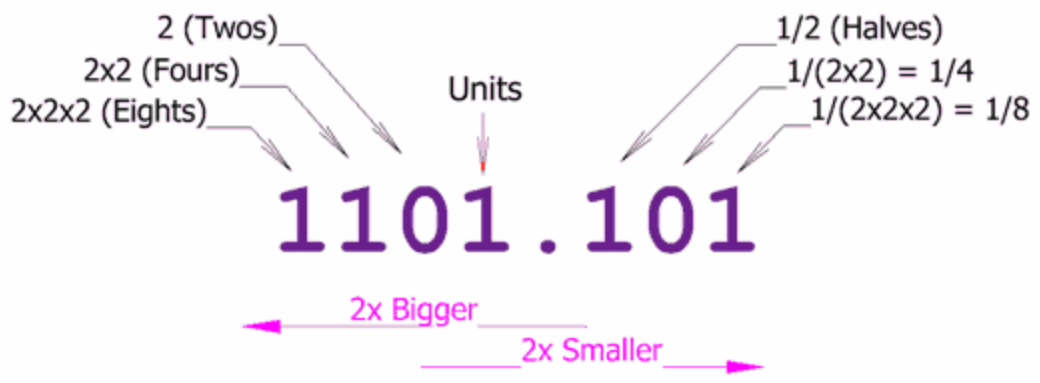
7 * 100 + 2 * 10 + 5 * 1 + 1 * 0.1 + 9 * 0.01 + 4 * 0.001 =

700 + 20 + 5 + 0.1 + 0.09 + 0.004 =

725.194

Binary Number System

The word binary comes from "Bi-" meaning two. We see "bi-" in words such as "bicycle" (two wheels) or "binocular" (two eyes). A Binary Number is made up of only 0's and 1s.



This is 1×8 + 1×4 + 0×2 + 1 + 1×(1/2) + 0×(1/4) + 1×(1/8)
(=13.625 in Decimal).

Similar to the Decimal System, numbers can be placed to the left or right of the point, to indicate values greater than one or less than one.

For Binary Numbers:

The number just to the left of the point is a whole number, we call this place units.

As we move left, every number place gets 2 times bigger.

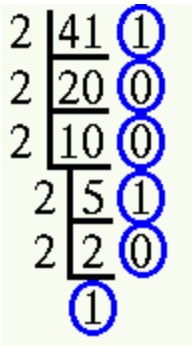
The first digit on the right of the point means halves (1/2).

As we move further right, every number place gets 2 times smaller (one half as big).

Decimal to Binary

We progressively divide the given decimal number by 2, till a quotient 0 is reached. The reminder of each division is written on the right. The reminders, taken from down to up, given the binary number.

For example, to convert the decimal number 41to binary number:



Now the reminders are read upward to get the binary equivalent. Thus,

Decimal number 41= 101001

A binary digit (a one or a zero) is called a bit. A group of bits having a gnificance is a byte or code. A byte is sometimes called a character and a group of one or more characters is a word. When binary number has 4 bits, it is called a nibble. A binary number with 8 bits is known as a byte.

Bit = X

Nibble = XXXX

Byte = XXXXXXXX

Equivalent numbers in decimal and binary notation.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Decimal:	20	25	30	40	50	100	200	500
Binary:	10100	11001	11110	101000	110010	1100100	11001000	111110100

CONVERTING OCTAL TO BINARY

The primary application of octal numbers is representing binary numbers, as it is easier to read large numbers in octal form that in binary form. Because each octal digit can be represented by a three-bit binary number, it is very easy to convert from octal to binary. Simply replace each octal digit with the appropriate three-bit binary number as indicated in the examples below.

Octal Digit 0 1 2 3 4 5 6 7

Binary Digit 000 001 010 011 100 101 110 111

13 8 = (001011) 2 (37.12) 8 = (011111. 001010)

CONVERTING BINARY TO OCTAL

Converting binary to octal is also a simple process. Break the binary digits into groups of three starting from the binary point and convert each group into its appropriate octal digit. For whole numbers, it may be necessary to add a zero as the MSB in order to complete a grouping of three bits. Note that this does not change the value of the binary number similarly, when representing fractions, it may be necessary to add a trailing zero in the LSB in order to form a complete grouping of three.

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

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:

```
  1001101          1001001          1000111
+0010010         +0011001         +0010110
-----------      -----------      -----------
1011111          1100010          101110
```

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.

As we'll see later, there are ways that electronic circuits can be built to perform this very task of addition, by representing each bit of each binary number as a voltage signal (either "high," for a 1; or "low" for a 0). This is the very foundation of all the arithmetic which modern digital computers perform.

BINARY SUBTRACTION

Subtraction is generally simpler than addition since only two numbers are involved and the upper value representation is greater than the lower value representation. The problem of "borrow" is similar in binary subtraction to that in decimal. We can construct a subtraction table that has two parts - the three cases of subtracting without borrow, and the one case of the involvement of a borrow digit, no matter how far to the left is the next available binary digit.

A	B	A-B
One digit cases		
0	0	0
1	0	1
1	1	0
The One case of Borrow		
10...0	1	original = 01...1 and result = 1

BINARY MULTIPLICATION

It is actually much simpler than decimal multiplication. In the case of decimal multiplication, we need to remember 3 x 9 = 27, 7 x 8 = 56, and so on. In binary multiplication, we only need to remember the following,

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Note that since binary operates in base 2, the multiplication rules we need to remember are those that involve 0 and 1 only. As an example of binary multiplication we have 101 times 11,

101
x11

First we multiply 101 by 1, which produces 101. Then we put a 0 as a placeholder as we would in decimal multiplication, and multiply 101 by 1, which produces 101.

101
x11

```
  101
1010  <-- the 0 here is the placeholder
```

The next step, as with decimal multiplication, is to add. The results from our previous step indicate that we must add 101 and 1010, the sum of which is 1111.

```
  101
 x11
 ---
  101
1010
----
1111
```

BINARY DIVISION

Binary division is almost as easy, and involves our knowledge of binary multiplication. Take for example the division of 1011 into 11.

```
     _11_  R=10
11 )1011
    -11
    ---
     101
     -11
     ---
      10  <-- remainder, R
```

To check our answer, we first multiply our divisor 11 by our quotient 11. Then we add its' product to the remainder 10, and compare it to our dividend of 1011.

```
   11
 x 11
 ---
   11
  11
 ----
 1001  <-- product of 11 and 11

 1001
+ 10
 ----
 1011  <-- sum of product and remainder
```

The sum is equal to our initial dividend, therefore our solution is correct.

BINARY SUBTRACTION BY 1'S COMPLEMENT

The same basic steps for subtraction apply in one's complement subtraction that is applied in two's complement subtraction, with very little change in the procedure. To one's complement a number, just turn the ones into zeros and the zeros into ones

Let's consider how we would solve our problem of subtracting 1_{10} from 7_{10} using 1's complement.

First, we need to convert 0001_2 to its negative equivalent in 1's complement.

To do this we change all the 1's to 0's and 0's to 1's. Notice that the most-significant digit is now 1 since the number is negative.

Next, we add the negative value we computed to 0111_2. This gives us a result of 10101_2.

Notice that our addition caused an overflow bit. Whenever we have an overflow bit in 1's complement, we add this bit to our sum to get the correct answer. If there is no overflow bit, then we leave the sum as it is.

This gives us a final answer of 0110_2 (or 6_{10}).

```
  0111    (7)
- 0001  - (1)

0001 -> 1110

  0111    (7)
+ 1110  +(-1)

  10101   (?)
   0101
+     1

  0110    (6)
  0111    (7)
- 0001  - (1)

  0110    (6)
```

Property

Two's complement representation allows the use of binary arithmetic operations on signed integers, yielding the correct 2's complement results.

Positive Numbers

Positive 2's complement numbers are represented as the simple binary.

Negative Numbers

Negative 2's complement numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero.

The most significant (leftmost) bit indicates the sign of the integer; therefore it is sometimes called the sign bit.

If the sign bit is zero,

then the number is greater than or equal to zero, or positive.

If the sign bit is one, then the number is less than zero, or negative.

2's Complement Subtraction

Two's complement subtraction is the **binary addition** of the minuend to the 2's complement of the subtrahend (adding a negative number is the same as subtracting a positive one).

For example,

7 - 12 = (-5)

```
  0000 0111   = +7
+ 1111 0100   = -12
-------------
  1111 1011   = -5
```

The two's complement of a binary number is defined as the value obtained by subtracting the number from a large power of two (specifically, from 2^N for an *N*-bit two's complement). The two's complement of the number then behaves like the negative of the original number in most arithmetic, and it can coexist with positive numbers in a natural way.

A two's-complement system or two's-complement arithmetic is a system in which negative numbers are represented by the two's complement of the absolute value; this system is the most common method of representing signed integers on computers. In such a system, a number is negated (converted from positive to negative or vice versa) by computing its two's complement. An N-bit two's-complement numeral system can represent every integer in the range -2^{N-1} to $+2^{N-1}-1$.

The two's-complement system has the advantage of not requiring that the addition and subtraction circuitry examine the signs of the operands to determine whether to add or subtract. This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic. Also, zero has only a single representation, obviating the subtleties associated with negative zero, which exists in ones'-complement systems.

LAWS OF BOOLEAN ALGEBRA

Boolean algebra

The most obvious way to simplify Boolean expressions is to manipulate them in the same way as normal algebraic expressions are manipulated. With regards to logic relations in digital forms, a set of rules for symbolic manipulation is needed in order to solve for the unknowns.

A set of rules formulated by the English mathematician George Boole describe certain propositions whose outcome would be either true or false. With regard to

digital logic, these rules are used to describe circuits whose state can be either, 1 (true) or 0 (false). In order to fully understand this, the relation between the AND gate, OR gate and NOT gate operations should be appreciated. A number of rules can be derived from these relations as Table 1 demonstrates.

P1: X = 0 or X = 1
P2: 0. 0 = 0
P3: 1 + 1 = 1
P4: 0 + 0 = 0
P5: 1 . 1 = 1
P6: 1 . 0 = 0 . 1 = 0
P7: 1 + 0 = 0 + 1 = 1
Laws of Boolean algebra

the basic Boolean laws. Note that every law has two expressions, (a) and (b). This is known as duality. These are obtained by changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa.
It has become conventional to drop the . (AND symbol) i.e. A.B is written as AB.

T1: Commutative Law

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

T2: Associate Law

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

T3: Distributive Law

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

T4: Identity Law

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

T5:

(a) $A\bar{B} + AB = A$
(b) $(A+\bar{B})(A+B) = A$

T6: Redundance Law

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

T7:

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

T8:

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

T9:

(a) $A + \overline{A}\ B = A + B$
(b) $A\,(\overline{A} + B) = A\,B$

T10:

(a) $\overline{A} + A = 1$
(b) $\overline{A}\ A = 0$

T11: De Morgan's Theorem

(a) $(\overline{A + B}) = \overline{A}\ \overline{B}$
(b) $(\overline{A\,B}) = \overline{A} + \overline{B}$

Examples

Prove T10: (a) $A + \overline{A}\ B = A + B$

(**1) Algebraically:**

$$
\begin{aligned}
A + \overline{A}\ B &= A\,1 + \overline{A}\ B && \text{T7(a)} \\
&= A\,(1 + B) + \overline{A}\ B && \text{T7(c)} \\
&= A + A\,B + \overline{A}\ B && \text{T3(a)} \\
&= A + B\,(A + \overline{A}) && \text{T3(a)} \\
&= A + B && \text{T(8)}
\end{aligned}
$$

(**2) Using the truth table:**

A	B	$A+B$	$\overline{A}\,B$	$A+\overline{A}\,B$
0	0	**0**	0	**0**
0	1	**1**	1	**1**
1	0	**1**	0	**1**
1	1	**1**	0	**1**

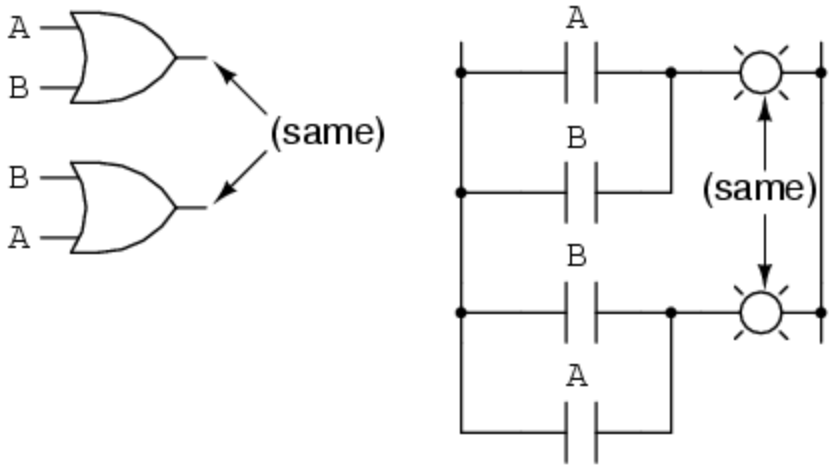
Using the laws given above, complicated expressions can be simplified.

BOOLEAN ALGEBRAIC PROPERTIES

Another type of mathematical identity, called a "property" or a "law," describes how differing variables relate to each other in a system of numbers. One of these properties is known as the *commutative property*, and it applies equally to addition and multiplication. In essence, the commutative property tells us we can reverse the order of variables that are either added together or multiplied together without changing the truth of the expression:

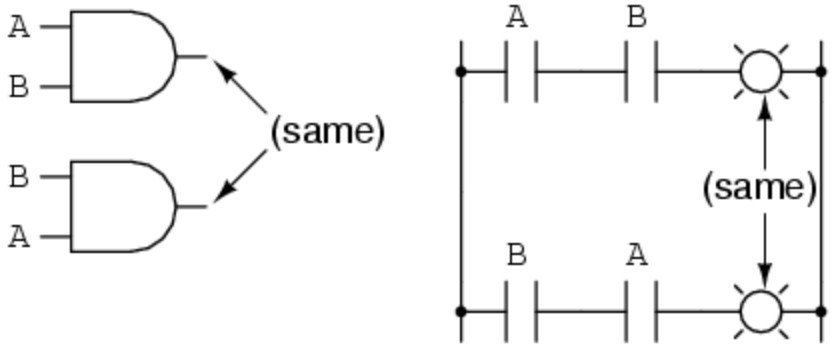
Commutative property of addition

A + B = B + A



Commutative property of multiplication

AB = BA

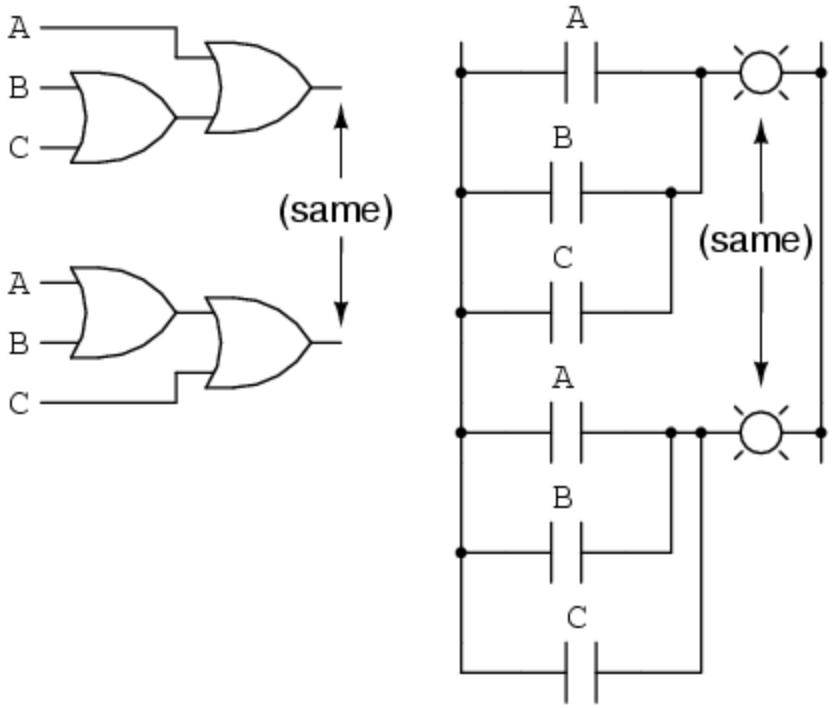


Along with the commutative

properties or addition and multiplication, we have the *associative property*, again applying equally well to addition and multiplication. This property tells us we can associate groups of added or multiplied variables together with parentheses without altering the truth of the equations.

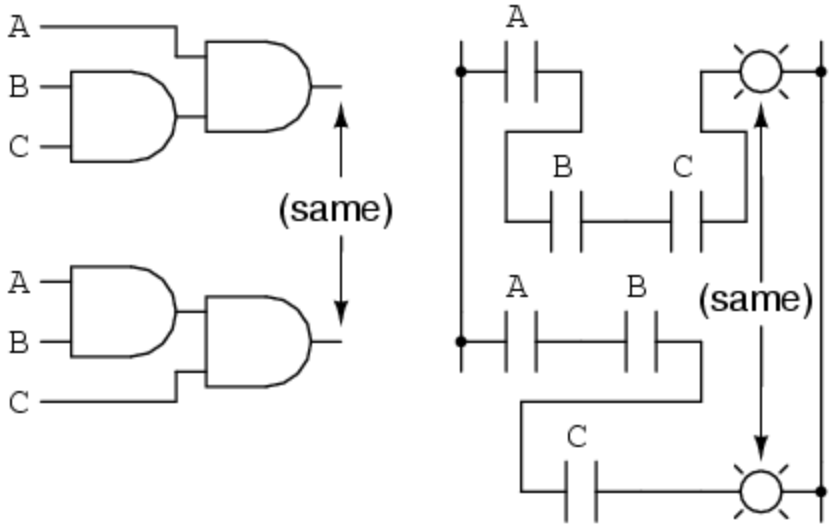
Associative property of addition

A + (B + C) = (A + B) + C



Associative property of multiplication

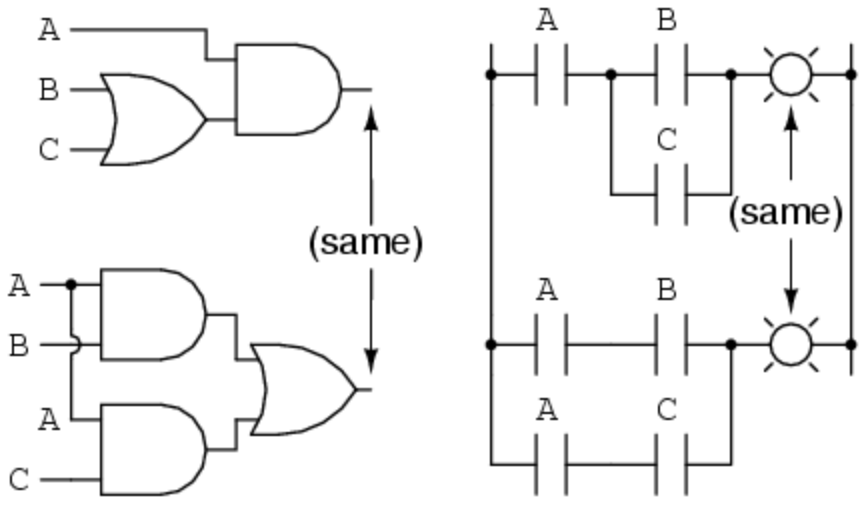
A(BC) = (AB)C



Lastly, we have the *distributive property*, illustrating how to expand a Boolean expression formed by the product of a sum, and in reverse shows us how terms may be factored out of Boolean sums-of-products:

Distributive property

A(B + C) = AB + AC



To summarize, here are the three basic properties: commutative, associative, and distributive.

Basic Boolean algebraic properties

Additive	Multiplicative
$A + B = B + A$	$AB = BA$
$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
$A(B + C) = AB + AC$	

FORMAL PROOF OF DEMORGAN'S THEOREMS

DeMorgan's Theorems:

a. (A + B) = A* B
b. A*B = A + B
Note: * = AND operation

Proof of DeMorgan's Theorem (b):
For any theorem X=Y, if we can show that X Y = 0, and that X + Y = 1, then
by the complement postulates, A A = 0 and A + A = 1,
X = Y. By the uniqueness of the complement, X = Y.
Thus the proof consists of showing that (A*B)*(A + B) = 0; and also that (A*B) + (
A + B) = 1.

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		Q.E.D.

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	Q.E.D.

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.

KARNAUGH MAP

The Karnaugh map provides a simple and straight-forward method of minimizing Boolean expressions. With the Karnaugh map Boolean expressions having up to four and even six variables can be simplified.
A Karnaugh map provides a pictorial method of grouping together expressions with common factors and therefore eliminating unwanted variables. The Karnaugh map can also be described as a special arrangement of a truth table.

TWO VARIABLE MAP

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.

A / B	0	1
0	a	b
1	c	d

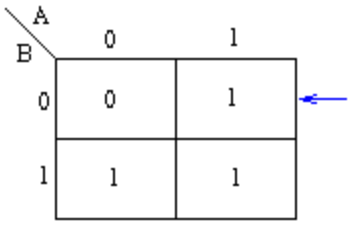
F.

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.

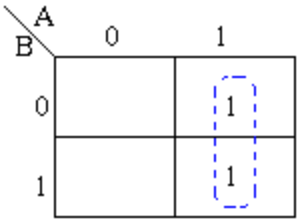


F.

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:

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

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 minimized answer therefore is Z = A.

UNIT 2

Positive Logic and Negative Logic system:

A digital bit of information (0 or 1) can be represented by a voltage leveling a d.c (or level logic) system. A binary information can have any one of the two possible states.

Logic level 1 or logic level 0

If more positive voltage is taken as the 1 level and the other as the 0 level, the logic system is known as the positive logic system.

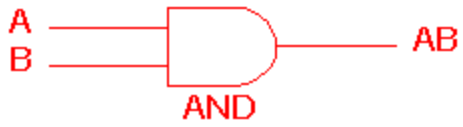
If more negative voltage is taken to represent the 1 level and the other for the 0 level, the logic system is known as negative logic system.

V (0) = 5 volt
V (1) = 0 volt

Logic gates

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

AND gate



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

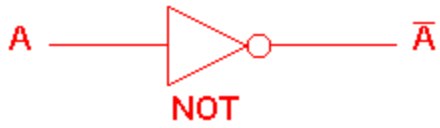
The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

OR gate

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

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

NOT gate



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

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs.

NAND gate

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

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

NOR gate

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

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

EXOR gate

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

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

Table 1: Logic gate symbols

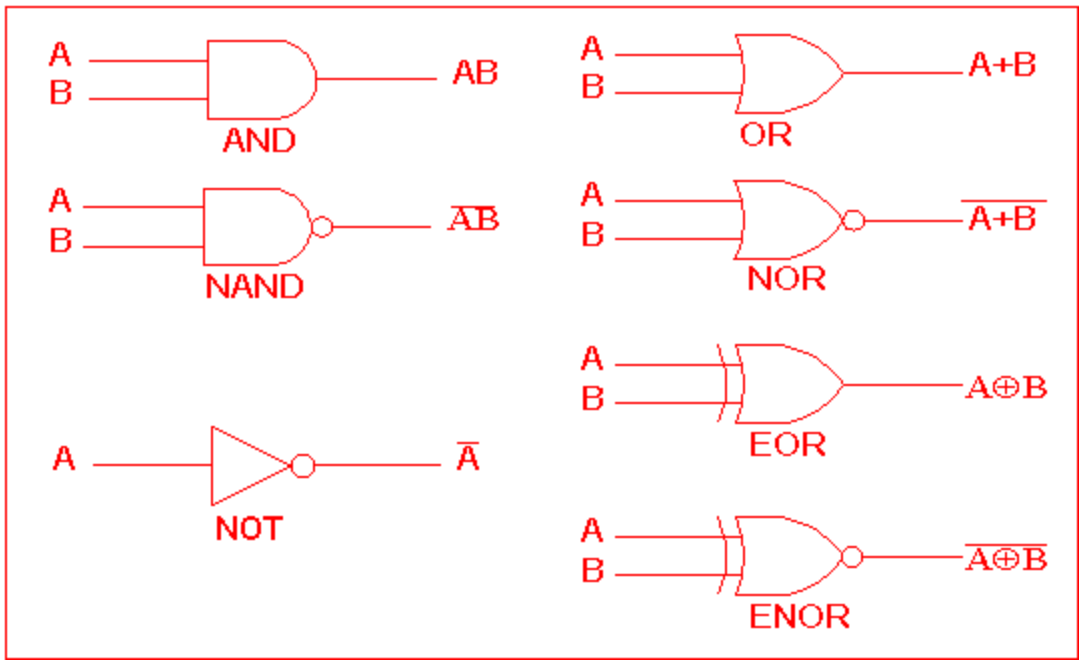


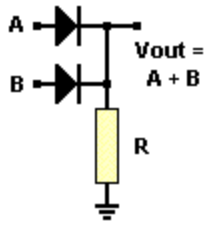
Table 2: Logic gates representation using the Truth table

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

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Diode Register Logic OR gate

Looking at the picture:



If one or both inputs are at logic "1" (5 volts), the current will flow through one or both diodes.

This current passes through the resistor and causes the appearance of a voltage across its terminals, thereby obtaining logic "1" on the output.

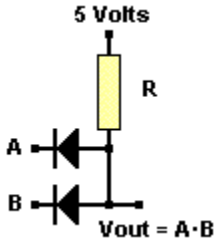
We only get logic "0" (0 volts) on the output when both inputs are in logic "0".

In this case, the diodes do not conduct, there is no current through the resistor R and there is no voltage across its terminals.

As a result the voltage at V out is the same as ground (0 volts)

Diode Register Logic AND gate

Looking at the picture:



When both inputs are at logic "1", the two diodes are reverse biased and there is no current flowing to ground.

Therefore the output is logic "1" because there is no voltage drop across the resistor R.

If one of the inputs is logic "0", the current will flow through the corresponding diode and through the resistor.

Thus the diode anode (the output) will be logic "0".

This method works fine when the circuits are simple, but there are problems when you have to make interconnections with such gates.

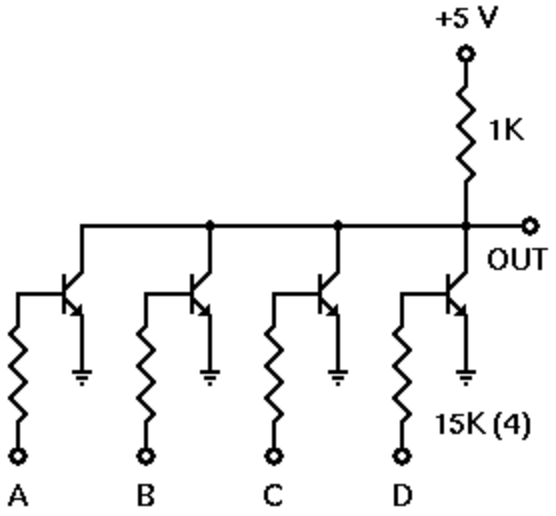
4-Input RTL NOR Gate

Introduction

In the previous experiment, we looked at the basic RTL inverter circuit. We found that it performed its job properly and well, and that the output voltage is not seriously degraded by a connection to a similar circuit, which then acts as a load on the first output.

Now it's time to use RTL technology to combine multiple logic signals into a single output signal. In this experiment, we will combine multiple RTL inverters into a single 4-input NOR gate. However, the method we will demonstrate here can be used for any number of inputs.

Schematic Diagram



It is possible to create an RTL NOR gate by connecting multiple input resistors to a single transistor. However, because the base of the transistor does not operate at ground potential, there will be some interaction between input signals and a limit on the number of input signals that can be applied to one transistor. To avoid that problem, the circuit to the right is preferred. Here, each input signal is applied to its own transistor, just as in an inverter circuit. However, the transistor collectors are connected together with just a single collector load resistor. This way, the signals all combine only after inversion has occurred, so the input signals cannot interfere or interact with each other under any circumstances.

Any number of transistors can be used in this configuration to produce an RTL NOR gate of that many inputs.

Two-Input DTL NOR Gate

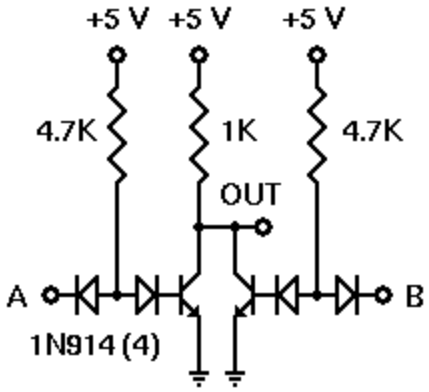
Introduction

In the previous experiment, we found that we could combine a Diode Logic AND gate with a DTL inverter, and thereby create a DTL NAND gate. This leads us to the question, "Can we combine a DL OR gate with a DTL inverter to get a NOR gate?"

We could do that, but we might run into a bit of trouble. Both the inverter and the NAND gate use a pull-up resistor in the base circuit of the transistor, and the preceding gate must actively pull the input down to logic 0 in order to override that behavior. However, the DL OR gate would require the active drive current for the transistor to come from the previous circuit. This led to trouble with Diode Logic, since it changed all the references around. We would prefer to keep the input circuit for DTL gates operating in the same way for all gates.

We can accomplish this by using the same technique that we used with RTL. That is, we can combine the outputs of multiple inverters to provide a NOR function. We'll explore this idea in this experiment.

Schematic Diagram



The DTL NOR gate combines multiple DTL inverters with a common output as shown in the schematic diagram to the right. This is exactly the same as the method used to combine RTL inverters to form a NOR gate. Any number of inverters may be combined in this fashion to allow the required number of inputs to the NOR gate.

It would have been possible to construct the NOR gate by using a Diode Logic OR gate followed by a transistor inverter, just as we used a DL AND gate as the input to the DTL NAND gate. However, if we did that, the input circuitry for the NOR gate would have very different characteristics than the input circuitry for the NAND gate, and this might well lead to some serious complications in the design of any circuit involving both types of gates. By keeping the input circuitry and behavior of all gates

the same, we ensure that the input characteristics of all gates will be consistent and easy to work with.

Three-Input DTL NAND Gate

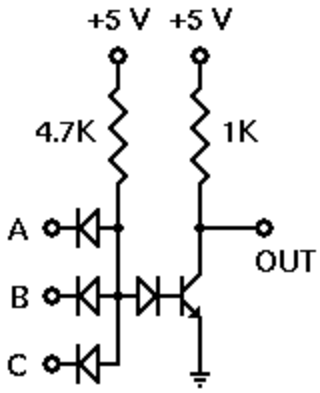
Introduction

One of the limitations of RTL circuits was that while NOR gates are easy to construct, it is difficult to obtain a NAND function in a single gate. Theoretically we could stack multiple transistors in series, but this leads to a range of problems.

With DTL, however, we can simply add more input diodes to the inverter circuit. This effectively combines a Diode Logic AND gate with an inverter circuit. The use of the inverter re-amplifies the signal and thus overcomes the limitations we saw in the plain DL AND gate.

In this experiment, we will extend our DTL inverter according to this idea, and see how well it works.

Schematic Diagram



The DTL NAND gate combines the DTL inverter with a simple Diode Logic (DL) AND gate. Thus, any number of inputs can be added simply by adding input diodes to the circuit. The problem of signal degradation caused by Diode Logic is overcome by the transistor, which amplifies the signal while inverting it. This means DTL gates can be cascaded to any required extent, without losing the digital signal.

In addition, the use of diodes in this configuration permits the construction of NAND gates; something that was not practical with RTL because there was no practical way to allow any single logic 0 input to override multiple logic 1 inputs. Therefore, DTL offers more possible configurations as well as better performance than RTL.

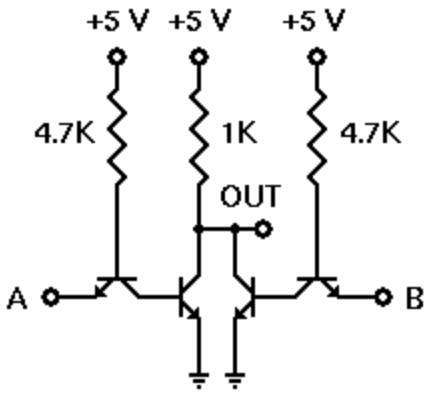
Two-Input TTL NOR Gate

Introduction

TTL integrated circuits provide multiple inputs to NAND gates by designing transistors with multiple emitters on the chip. Unfortunately, we can't very well simulate that on a breadboard socket. What we can do, however, is use the same technique that we tried in both RTL and DTL, to form a NOR gate. This requires separate inverter transistors just as it did with the other logic families.

In this experiment, we'll construct a TTL NOR circuit and test its operation.

Schematic Diagram



The figure to the right shows the schematic diagram of a TTL NOR gate. As you may recall from earlier experiments, it is substantially the same as the RTL and DTL NOR gates we have already explored. The only difference is that this time we are using the TTL input circuitry.

UNIT 3

Binary Addition

Binary Addition follows the same basic rules as for the denary addition above except in binary there are only two digits and the largest digit is "1", so any "SUM" greater than 1 will result in a "CARRY". This carry 1 is passed over to the next column for addition and so on. Consider the single bit addition below.

0	0	1	1
+ 0	+ 1	+ 0	+ 1
0	1	1	10

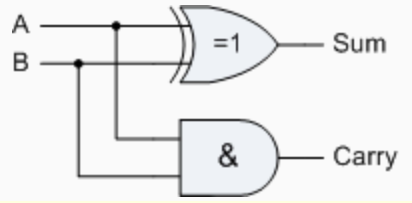
The single bits are added together and "0 + 0", "0 + 1", or "1 + 0" results in a sum of "0" or "1" until you get to "1 + 1" then the sum is equal to "2". For a simple 1-bit addition problem like this, the resulting carry bit could be ignored which would result in an output truth table resembling that of an **Ex-OR Gate** as seen in the Logic Gates section and whose result is the sum of the two bits but without the carry. An Ex-OR gate only produces an output "1" when either input is at logic "1", but not both. However, all microprocessors and electronic calculators require the carry bit to correctly calculate the equations so we need to rewrite them to include 2 bits of output data as shown below.

00	00	01	01
+ 00	+ 01	+ 00	+ 01
00	01	01	10

From the above equations we know that an **Ex-OR** gate will only produce an output "1" when "EITHER" input is at logic "1", so we need an additional output to produce a carry output, "1" when "BOTH" inputs "A" and "B" are at logic "1" and a standard **AND Gate** fits the bill nicely. By combining the Ex-OR gate with the AND gate results in a simple digital binary adder circuit known commonly as the "**Half Adder**" circuit.

The Half Adder Circuit

1-bit Adder with Carry-Out

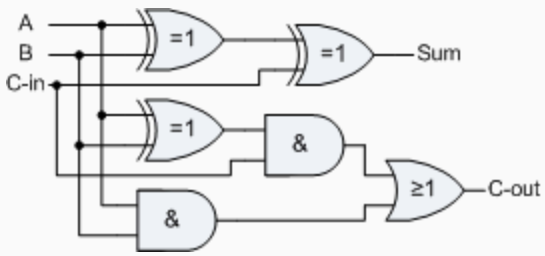
Symbol	Truth Table			
	A	B	SUM	CARRY
	0	0	0	0
	0	1	1	0
	1	0	1	0
	1	1	0	1
Boolean Expression: Sum = A ⊕B Carry = A. B				

From the truth table we can see that the SUM (S) output is the result of the Ex-OR gate and the Carry-out (Cout) is the result of the AND gate. One major disadvantage of the Half Adder circuit when used as a binary adder, is that there is no provision for a "Carry-in" from the previous circuit when adding together multiple data bits. For example, suppose we want to add together two 8-bit bytes of data, any resulting carry bit would need to be able to "ripple" or move across the bit patterns starting from the least significant bit (LSB). The most complicated operation the half adder can do is "1 + 1" but as the half adder has no carry input the resultant added value would be incorrect. One simple way to overcome this problem is to use a **Full Adder** type binary adder circuit.

The Full Adder Circuit

The main difference between the **Full Adder** and the previous seen **Half Adder** is that a full adder has three inputs, the same two single bit binary inputs A and B as before plus an additional *Carry-In* (C-in) input as shown below.

Full Adder with Carry-In

Symbol	Truth Table				
	A	B	C-in	Sum	C-out
	0	0	0	0	0
	0	1	0	1	0
	1	0	0	1	0
	1	1	0	0	1
	0	0	1	1	0
	0	1	1	0	1
	1	0	1	0	1
	1	1	1	1	1

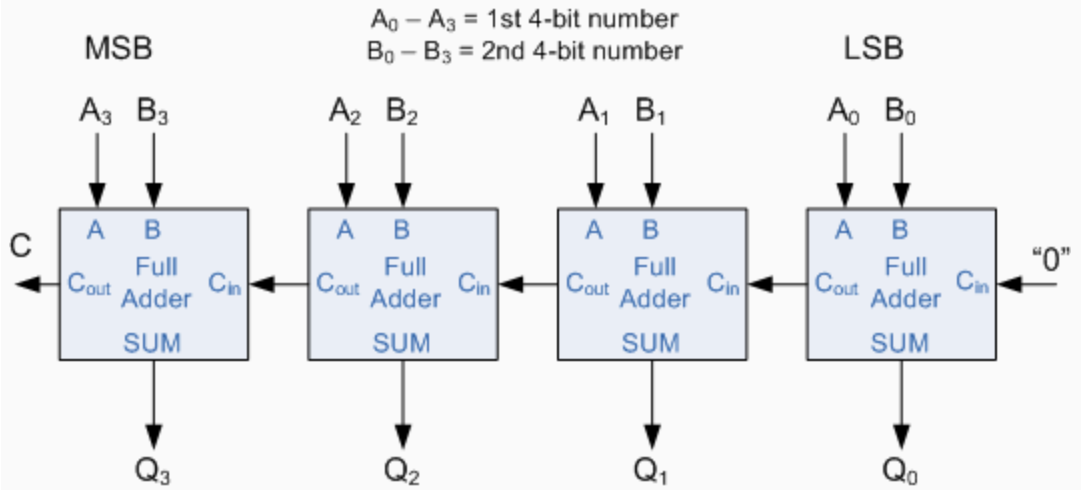
Boolean Expression: Sum = A ⊕B ⊕C-in

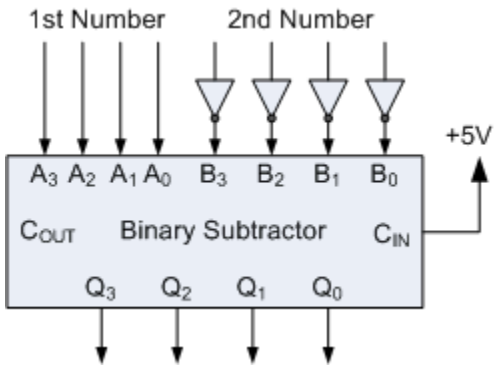
The 1-bit **Full Adder** circuit above is basically two half adders connected together and consists of three Ex-OR gates, two AND gates and an OR gate, six logic gates in total. The truth table for the full adder includes an additional column to take into account the Carry-in input as well as the summed output and carry-output. 4-bit full adder circuits are available as standard IC packages in the form of the TTL 74LS83 or the 74LS283 which can add together two 4-bit binary numbers and generate a SUM and a CARRY output. But what if we wanted to add together two n-bit numbers, then n 1-bit full adders need to be connected together to produce what is known as the **Ripple Carry Adder**.

The 4-bit Binary Adder

The **Ripple Carry Binary Adder** is simply n, full adders cascaded together with each full adder represents a single weighted column in the long addition with the carry signals producing a "ripple" effect through the binary adder from right to left. For example, suppose we want to "add" together two 4-bit numbers, the two outputs of the first full adder will provide the first place digit sum of the addition plus a carry-out bit that acts as the carry-in digit of the next binary adder. The second binary adder in the chain also produces a summed output (the 2nd bit) plus another carry-out bit and we can keep adding more full adders to the combination to add larger numbers, linking the carry bit output from the first full binary adder to the next full adder, and so forth. An example of a 4-bit adder is given below.

A 4-bit Binary Adder





One main disadvantage of "cascading" together 1-bit **binary adders** to add large binary numbers is that if inputs A and B change, the sum at its output will not be valid until any carry-input has "rippled" through every full adder in the chain. Consequently, there will be a finite delay before the output of a adder responds to a change in its inputs resulting in the accumulated delay especially in large multi-bit binary adders becoming prohibitively large. This delay is called **Propagation delay**. Also "overflow" occurs when an n-bit adder adds two numbers together whose sum is greater than or equal to 2^n

One solution is to generate the carry-input signals directly from the A and B inputs rather than using the ripple arrangement above. This then produces another type of binary adder circuit called a **Carry Look Ahead Binary Adder** were the speed of the parallel adder can be greatly improved using carry-look ahead logic.

The 4-bit Binary Subtractor

Now that we know how to "ADD" together two 4-bit binary numbers how would we subtract two 4-bit binary numbers, for example, A - B using the circuit above. The answer is to use 2's-complement notation on all the bits in B must be complemented (inverted) and an extra one added using the carry-input. This can be achieved by inverting each B input bit using an inverter or NOT-gate.

Also, in the above circuit for the 4-bit binary adder, the first carry-in input is held LOW at logic "0", for the circuit to perform subtraction this input needs to be held HIGH at "1". With this in mind a ripple carry adder can with a small modification be used to perform half subtraction, full subtraction and/or comparison.

There are a number of 4-bit full-adder ICs available such as the 74LS283 and CD4008. which will add two 4-bit binary number and provide an additional input carry bit, as well as an output carry bit, so you can cascade them together to produce 8-bit, 12-bit, 16-bit, etc. adders.

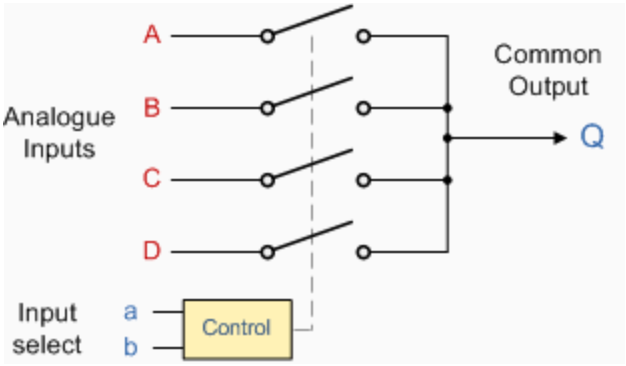
The Multiplexer

A data selector, more commonly called a **Multiplexer**, shortened to "Mux" or "MPX", are combinational logic switching devices that operate like a very fast acting multiple position rotary switch. They connect or control, multiple input lines called "channels" consisting of either 2, 4, 8 or 16 individual inputs, one at a time to an output. Then the job of a multiplexer is to allow multiple signals to *share* a single common output.

For example, a single 8-channel multiplexer would connect one of its eight inputs to the single data output. Multiplexers are used as one method of reducing the number of logic gates required in a circuit or when a single data line is required to carry two or more different digital signals.

Digital **Multiplexers** are constructed from individual **analogue switches** encased in a single IC package as opposed to the "mechanical" type selectors such as normal conventional switches and relays. Generally, multiplexers have an even number of data inputs, usually an even power of two, n^2 , a number of "control" inputs that correspond with the number of data inputs and according to the binary condition of these control inputs, the appropriate data input is connected directly to the output. An example of a **Multiplexer** configuration is shown below.

4-to-1 Channel Multiplexer



Addressing		Input Selected
b	a	
0	0	A
0	1	B
1	0	C
1	1	D

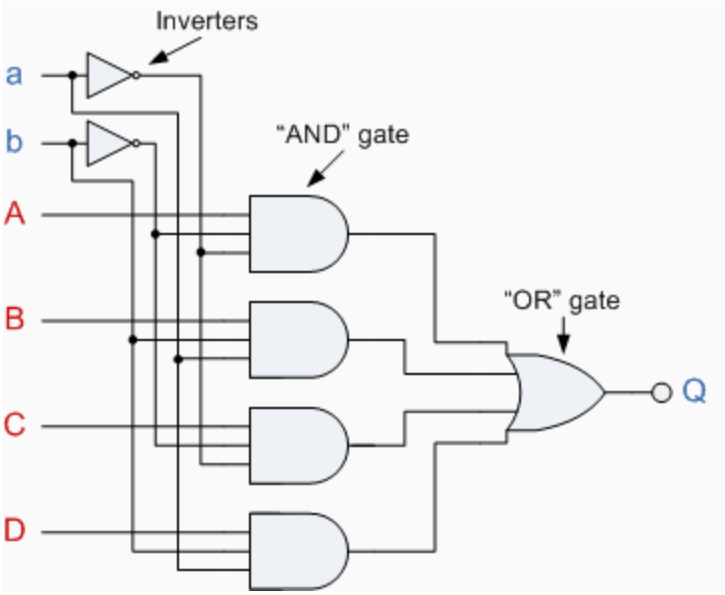
The Boolean expression for this 4-to-1 **Multiplexer** above with inputs A to D and data select lines a, b is given as:

Q = abA + abB + abC + abD

In this example at any one instant in time only ONE of the four analogue switches is closed, connecting only one of the input lines A to D to the single output at Q. As to which switch is closed depends upon the addressing input code on lines "a" and "b", so for this example to select input B to the output at Q, the binary input address would need to be "a" = logic "1" and "b" = logic "0". Adding more control address lines will allow the multiplexer to control more inputs but each control line configuration will connect only ONE input to the output.

Then the implementation of this Boolean expression above using individual logic gates would require the use of seven individual gates consisting of AND, OR and NOT gates as shown.

4 Channel Multiplexer using Logic Gates

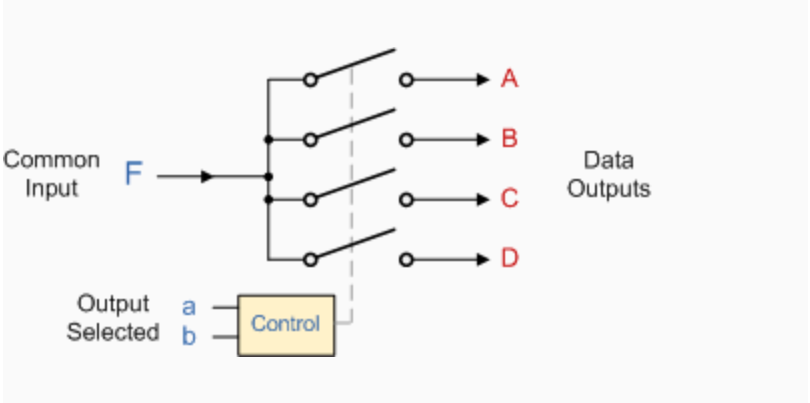


The symbol used in logic diagrams to identify a multiplexer is as follows.

The Demultiplexer

The data distributor, known more commonly as a **Demultiplexer** or "Demux", is the exact opposite of the **Multiplexer** we saw in the previous tutorial. The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The **demultiplexer** converts a serial data signal at the input to a parallel data at its output lines as shown below.

1-to-4 Channel De-multiplexer



Addressing		Output Selected
b	a	
0	0	A
0	1	B
1	0	C
1	1	D

The Boolean expression for this 1-to-4 **Demultiplexer** above with outputs A to D and data select lines a, b is given as:

F = ab A + abB + abC + abD

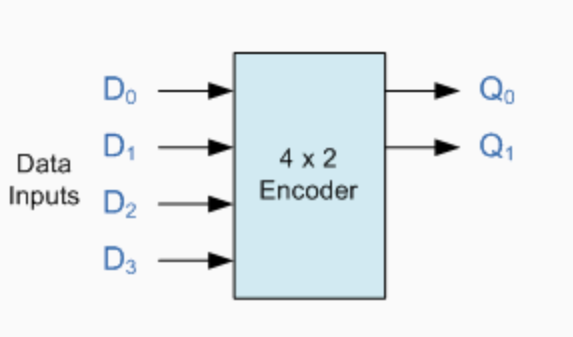
The function of the **Demultiplexer** is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individual solid state switches are selected by the binary input address code on the output select pins "a" and "b" and by adding more address line inputs it is possible to switch more outputs giving a 1-to-2^n data line outputs. Some standard demultiplexer IC´s also have an "enable output" input pin which disables or prevents the input from being passed to the selected output. Also some have latches built into their outputs to maintain the output logic level after the address inputs have been changed. However, in standard decoder type circuits the address input will determine which single data output will have the same value as the data input with all other data outputs having the value of logic "0".

The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

The Digital Encoder

Unlike a multiplexer that selects one individual data input line and then sends that data to a single output line or switch, a **Digital Encoder** more commonly called a **Binary Encoder** takes *ALL* its data inputs one at a time and then converts them into a single encoded output. So we can say that a binary encoder, is a multi-input combinational logic circuit that converts the logic level "1" data at its inputs into an equivalent binary code at its output. Generally, digital encoders produce outputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines. An "n-bit" binary encoder has 2^n input lines and n-bit output lines with common types that include 4-to-2, 8-to-3 and 16-to-4 line configurations. The output lines of a digital encoder generate the binary equivalent of the input line whose value is equal to "1" and are available to encode either a decimal or hexadecimal input pattern to typically a binary or B.C.D. output code.

4-to-2 Bit Binary Encoder



Inputs				Outputs	
D_3	D_2	D_1	D_0	Q_1	Q_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	0	x	x

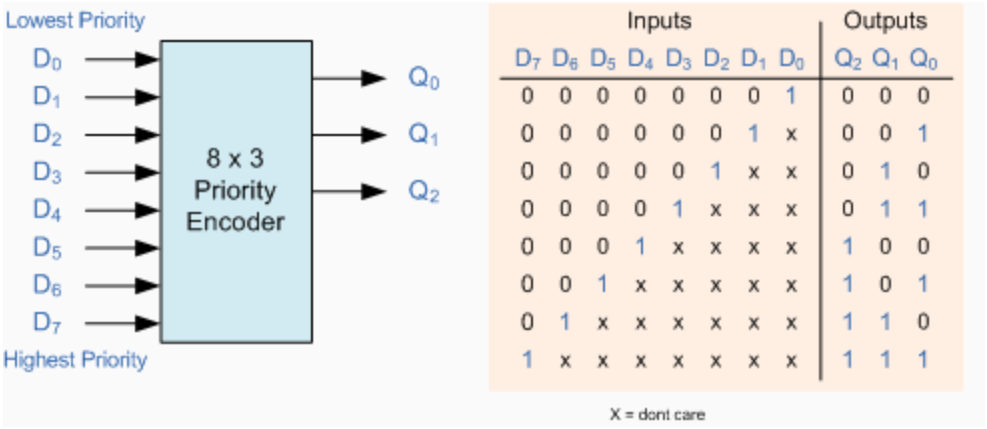
One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level "1". For example, if we make inputs D_1 and D_2 HIGH at logic "1" at the same time, the resulting output is neither at "01" or at "10" but will be at "11" which is an output binary number that is different to the actual input present. Also, an output code of all logic "0"s can be generated when all of its inputs are at "0" OR when input D_0 is equal to one.

One simple way to overcome this problem is to "Prioritise" the level of each input pin and if there was more than one input at logic level "1" the actual output code would only correspond to the input with the highest designated priority. Then this type of digital encoder is known commonly as a **Priority Encoder** or **P-encoder** for short.

Priority Encoders

Priority Encoders solve the problem mentioned above by allocating a priority level to each input. The encoder output corresponds to the currently active input with the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. Priority encoders come in many forms with an example of an 8-input priority encoder along with its truth table shown below.

8-to-3 Bit Priority Encoder



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

Priority encoders are available in standard IC form and the TTL 74LS148 is an 8 to 3 bit priority encoder which has eight active LOW (logic "0") inputs and provides a 3-bit code of the highest ranked input at its output. Priority encoders output the highest order input first for example, if input lines "D2", "D3" and "D5" are applied simultaneously the output code would be for input "D5" ("101") as this has the highest order out of the 3 inputs. Once input "D5" had been removed the next highest output code would be for input "D3" ("011"), and so on.

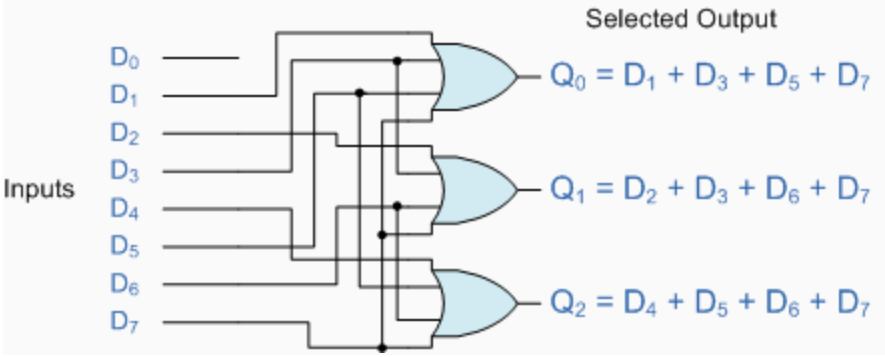
The Boolean expression for this 8-to-3 encoder above with inputs D_0 to D_7 and outputs Q_0, Q_1, Q_2 is given as:

$Q_0 = D_1 + D_3 + D_5 + D_7$

$Q_1 = D_2 + D_3 + D_6 + D_7$

$Q_2 = D_4 + D_5 + D_6 + D_7$

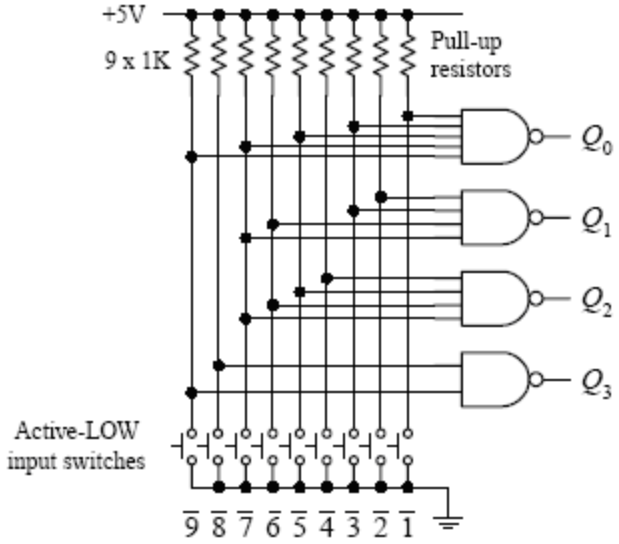
Then the implementation of these Boolean expression outputs above using individual OR gates is as follows. **Digital Encoder using Logic Gates**



DECIMAL to BCD ENCODER

Encoders are the opposite of decoders. They are used to generate a coded output from a single active numeric input. To illustrate this in a simple manner, let's take a look at the simple decimal-to-BCD encoder circuit shown below. In this circuit, normally all lines are held high by the pull-up resistors connected to +5 V. To generate a BCD output that is equivalent to a single selected decimal input, the switch corresponding to that decimal is closed. (The switch acts as an active-low input.) The truth table in Fig. explains the rest. Figure 12.40 shows a 74LS147 decimal-to-BCD (10-line-to-4-line) priority encoder IC. The 74LS147 provides the same basic function as the circuit shown in Fig. 12.39, but it has active-low outputs. This means that instead of getting an LLHH output when "3" is selected, as in the previous encoder, you get HHLL. The two outputs represent the same thing ("3"); one is expressed in positive true logic, and the other (the 74LS147) is expressed in negative true logic. If you do not like negative true logic, you can slap inverters on the outputs of the 74LS147 to get positive true logic. The choice to use positive or negative true logic really depends on what you are planning to drive. For example, negative true logic is useful when the device that you wish to drive uses active-low inputs.

Simple decimal-to-BCD encoder



Truth table

$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$	$\overline{8}$	$\overline{9}$	Q_0	Q_1	Q_2	Q_3	BCD (pos. logic)
H	H	H	H	H	H	H	H	H	L	L	L	L	0000 (0_{10})
L	H	H	H	H	H	H	H	H	L	L	L	H	0001 (1_{10})
H	L	H	H	H	H	H	H	H	L	L	H	L	0010 (2_{10})
H	H	L	H	H	H	H	H	H	L	L	H	H	0011 (3_{10})
H	H	H	L	H	H	H	H	H	L	H	L	L	0100 (4_{10})
H	H	H	H	L	H	H	H	H	L	H	L	H	0101 (5_{10})
H	H	H	H	H	L	H	H	H	L	H	H	L	0110 (6_{10})
H	H	H	H	H	H	L	H	H	L	H	H	H	0111 (7_{10})
H	H	H	H	H	H	H	L	H	H	L	L	L	1000 (8_{10})
H	H	H	H	H	H	H	H	L	H	L	L	H	1001 (9_{10})

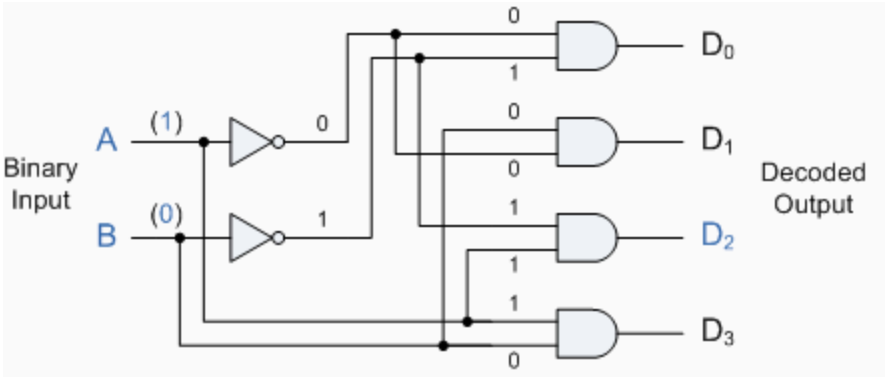
H = High voltage level, L = Low voltage level

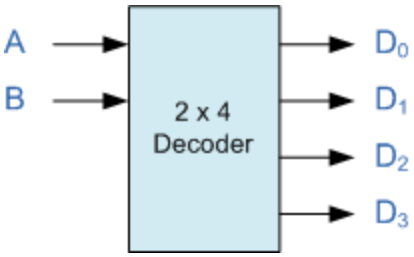
Binary Decoder

A **Decoder** is the exact opposite to that of an "Encoder" we looked at in the last tutorial. It is basically, a combinational type logic circuit that converts the binary code data at its input into one of a number of different output lines, one at a time producing an equivalent decimal code at its output. **Binary Decoders** have inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, and a n-bit decoder has 2^n output lines. Therefore, if it receives n inputs (usually grouped as a binary or Boolean number) it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. A decoders output code normally has more bits than its input code and practical binary decoder circuits include, 2-to-4, 3-to-8 and 4-to-16 line configurations.

A binary decoder converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to "decode" either a Binary or BCD (8421 code) input pattern to typically a Decimal output code. Commonly available BCD-to-Decimal decoders include the TTL 7442 or the CMOS 4028. An example of a 2-to-4 line decoder along with its truth table is given below. It consists of an array of four NAND gates, one of which is selected for each combination of the input signals A and B.

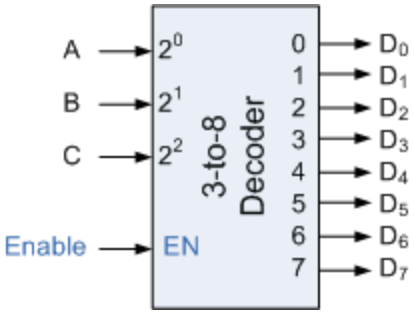
A 2-to-4 Binary Decoders.





A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

In this simple example of a 2-to-4 line binary decoder, the binary inputs A and B determine which output line from D0 to D3 is "HIGH" at logic level "1" while the remaining outputs are held "LOW" at logic "0" so only one output can be active (HIGH) at any one time. Therefore, whichever output line is "HIGH" identifies the binary code present at the input, in other words it "de-codes" the binary input and these types of binary decoders are commonly used as **Address Decoders** in microprocessor memory applications.



74LS138 Binary Decoder

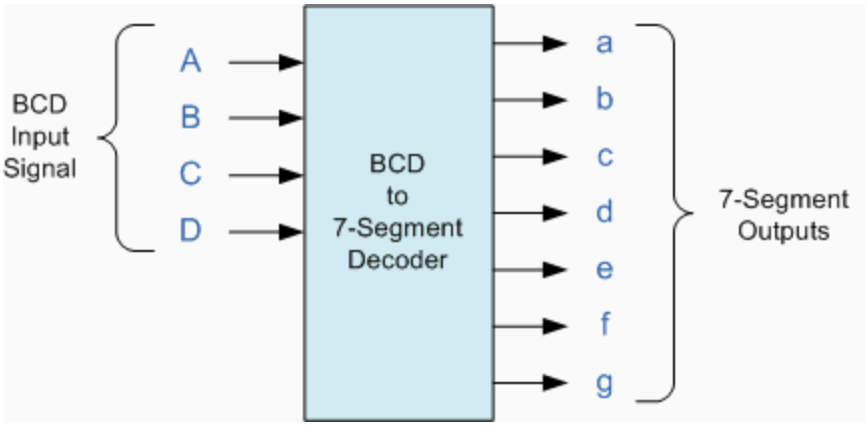
Some binary decoders have an additional input labelled "Enable" that controls the outputs from the device. This allows the decoders outputs to be turned "ON" or "OFF" and we can see that the logic diagram of the basic decoder is identical to that of the basic demultiplexer. Therefore, we say that a demultiplexer is a decoder with an additional data line that is used to enable the decoder. An alternative way of looking at the decoder circuit is to regard inputs A, B and C as address signals. Each combination of A, B or C defines a unique address which can access a location having that address.

Sometimes it is required to have a **Binary Decoder** with a number of outputs greater than is available, or if we only have small devices available, we can combine multiple decoders together to form larger decoder networks as shown. Here a much larger 4-to-16 line binary decoder has been implemented using two smaller 3-to-8 decoders.

BCD to 7-Segment Display Decoders

A binary coded decimal (BCD) to 7-segment display decoder such as the TTL 74LS47 or 74LS48, have 4 BCD inputs and 7 output lines, one for each LED segment. This allows a smaller 4-bit binary number (half a byte) to be used to display all the denary numbers from 0 to 9 and by adding two displays together, a full range of numbers from 00 to 99 can be displayed with just a single byte of 8 data bits.

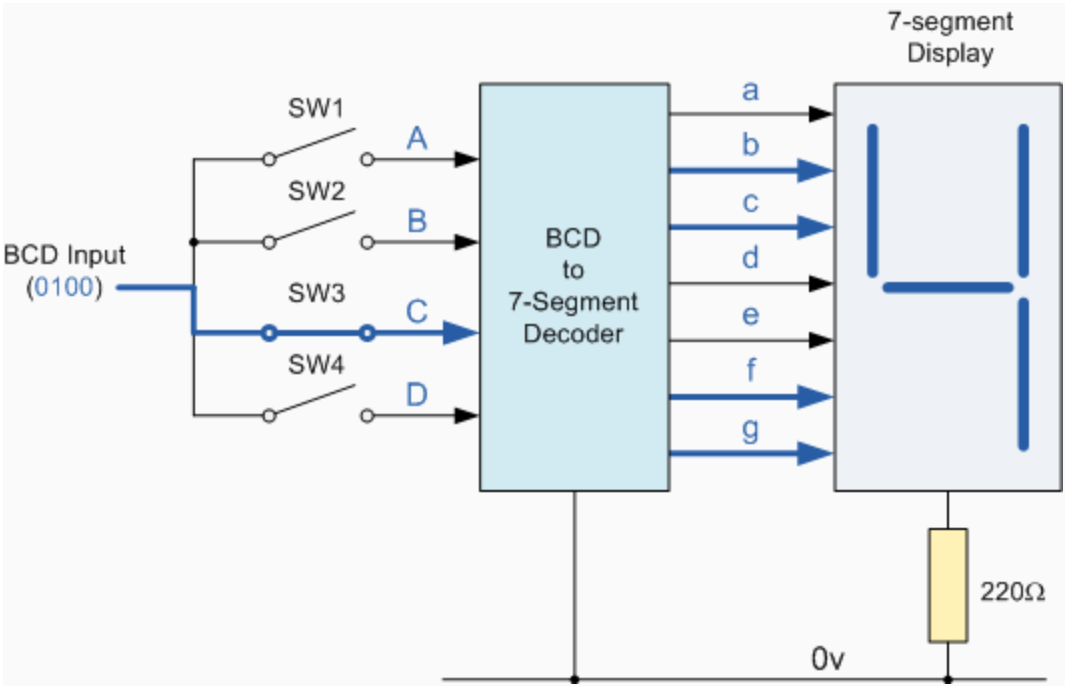
BCD to 7-Segment Decoder



The use of **packed** BCD allows two BCD digits to be stored within a single byte (8-bits) of data, allowing a single data byte to hold a BCD number in the range of 00 to 99.

An example of the 4-bit BCD input (0100) representing the number 4 is given below.

Example No1



In practice current limiting resistors of about 150Ω to 220Ω would be connected in series between the decoder/driver chip and each LED display segment to limit the maximum current flow. Different display decoders or drivers are available for the different types of display available, e.g. 74LS48 for common-cathode LED types, 74LS47 for common-anode LED types, or the CMOS CD4543 for liquid crystal display (LCD) types.Liquid crystal displays (LCD´s) have one major advantage over similar

LED types in that they consume much less power and nowadays, both LCD and LED displays are combined together to form larger Dot-Matrix Alphanumeric type displays which can show letters and characters as well as numbers in standard Red or Tri-colour outputs.

UNIT 4

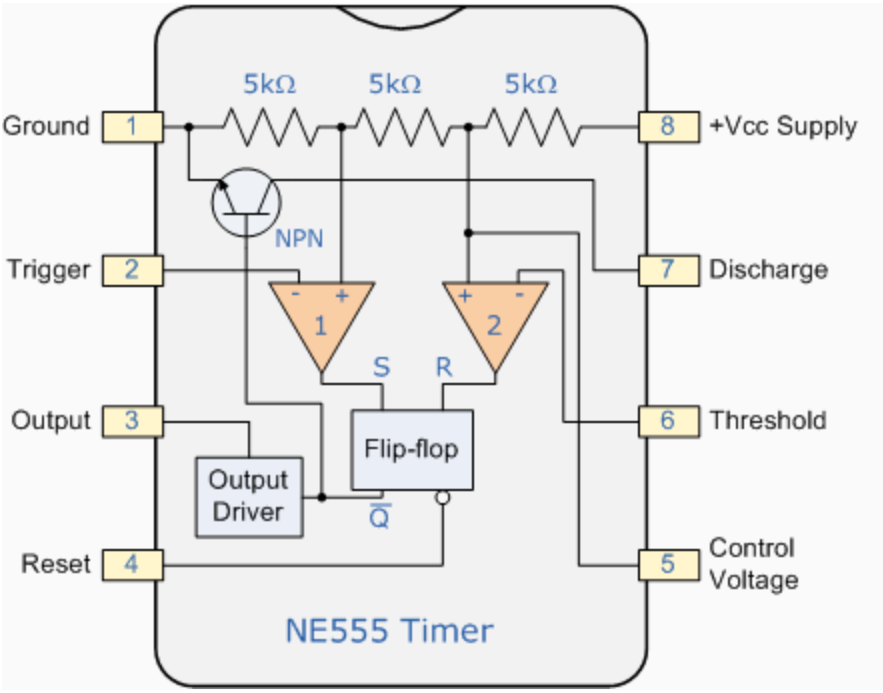
555 Timer

The **555 Timer** is a very cheap, popular and useful precision timing device that can act as either a simple timer to generate single pulses or long time delays, or as a relaxation oscillator producing stabilized waveforms of varying duty cycles from 50 to 100%. The 555 timer chip is extremely robust and stable 8-pin device that can be operated either as a very accurate **Monostable**, **Bistable** or **Astable** Multivibrator to produce a variety of applications such as one-shot or delay timers, pulse generation, LED and lamp flashers, alarms and tone generation, logic clocks, frequency division, power supplies and converters etc, in fact any circuit that requires some form of time control as the list is endless.

The single 555 Timer chip in its basic form is a Bipolar 8-pin mini Dual-in-line Package (DIP) device consisting of some 25 transistors, 2 diodes and about 16 resistors arranged to form two comparators, a flip-flop and a high current output stage as shown below. As well as the 555 Timer there is also available the NE556 Timer Oscillator which combines TWO individual 555's within a single 14-pin DIP package and low power CMOS versions of the single 555 timer such as the 7555 and LMC555 which use MOSFET transistors instead.

A simplified "block diagram" representing the internal circuitry of the **555 timer** is given below with a brief explanation of each of its connecting pins to help provide a clearer understanding of how it works.

555 Timer Block Diagram



Multivibrators

Individual **Sequential Logic** circuits can be used to build more complex circuits such as Counters, Shift Registers, Latches or Memories etc, but for these types of circuits to operate in a "Sequential" way, they require the addition of a clock pulse or timing signal to cause them to change their state. **Clock pulses** are generally square shaped waves that are produced by a single pulse generator circuit such as a **Multivibrator** which oscillates between a "HIGH" and a "LOW" state and generally has an even 50% duty cycle, that is it has a 50% "ON" time and a 50% "OFF" time. Sequential logic circuits that use the clock signal for synchronization may also change their state on either the rising or falling edge, or both of the actual clock signals. There are basically three types of pulse generation circuits,

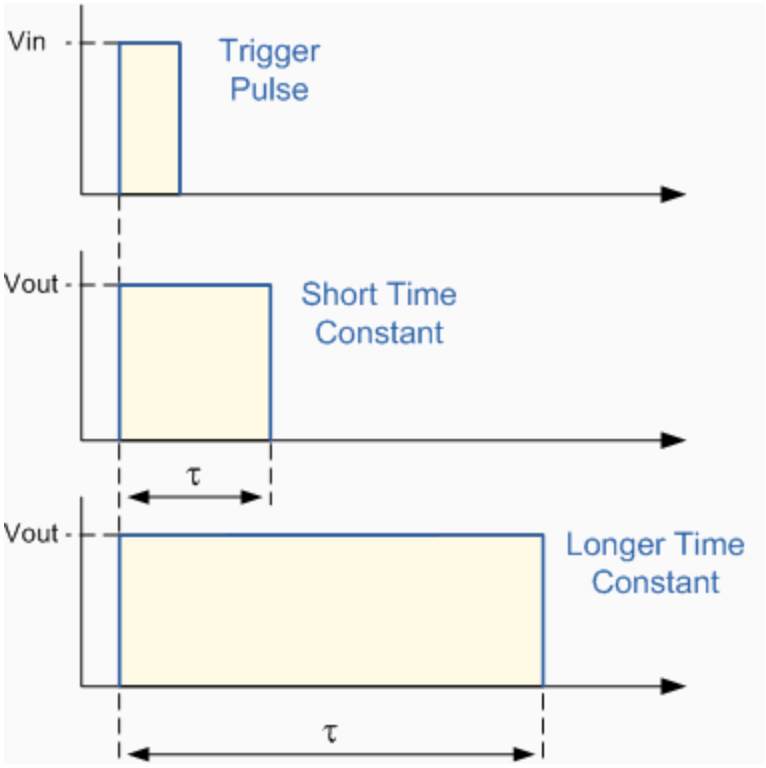
- Astable - has **NO** stable states but switches continuously between two states this action produces a train of square wave pulses at a fixed frequency.
- Monostable - has only **ONE** stable state and is triggered externally with it returning back to its first stable state.
- Bistable - has **TWO** stable states that produces a single pulse either positive or negative in value.

One way of producing a very simple clock signal is by the interconnection of logic gates. As a NAND gate contains amplification, it can also be used to provide a clock signal or pulse with the aid of a single **Capacitor, C** and **Resistor, R** which provides the feedback network. This simple type of RC Oscillator network is sometimes called a "Relaxation Oscillator".

Monostable Circuits.

Monostable Multivibrators or "One-Shot" pulse generators are used to generate a single output pulse, either "High" or "Low", when a suitable external trigger signal or pulse T is applied. This trigger signal initiates a timing cycle which causes the output of the monostable to change state at the start of the timing cycle and remain in this second state, which is determined by the time constant of the Capacitor, C and the Resistor, R until it automatically resets or returns itself back to its original (stable) state. Then, a monostable circuit has only one stable state. A more common name for this type of circuit is the "Flip-Flop".

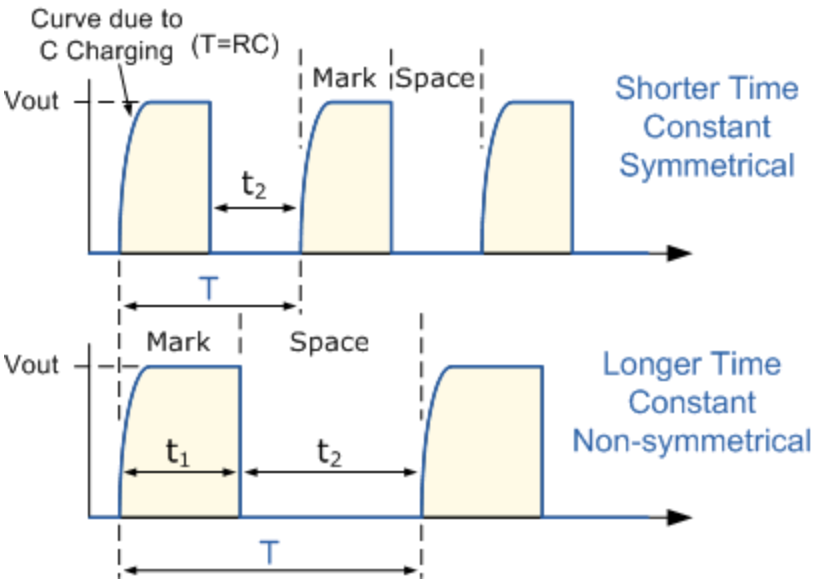
Monostable Multivibrator Waveforms



Astable Circuits.

Astable Multivibrators are a type of "free running oscillator" that have no permanent "Meta" or "Steady" state but are continually changing there output from one state ("LOW") to the other state ("HIGH") and then back again to its original state. This continual switching action from "HIGH" to "LOW" and "LOW" to "HIGH" produces a continuous square wave output whose timing cycle is determined by the time constant of the Resistor-Capacitor, (**RC Network**) connected to it.

[Astable Multivibrator Waveforms]



SR Flip-Flop

An **SR Flip-Flop** can be considered as a basic one-bit memory device that has two inputs, one which will "SET" the device and another which will "RESET" the device back to its original state and an output Q that will be either at a logic level "1" or logic "0" depending upon this Set/Reset condition. A basic NAND Gate SR flip flop circuit provides feedback from its outputs to its inputs and is commonly used in memory circuits to store data bits. The term "Flip-flop" relates to the actual operation of the device, as it can be "Flipped" into one logic state or "Flopped" back into another.

The simplest way to make any basic one-bit Set/Reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND Gates to form a Set-Reset Bistable or a SR NAND Gate Latch, so that there is feedback from each output to one of the other NAND Gate inputs. This device consists of two inputs, one called the Reset, R and the other called the Set, S with two corresponding outputs Q and its inverse or complement Q as shown below.

The SR NAND Gate Latch

The Set State

Consider the circuit shown above. If the input R is at logic level "0" (R = 0) and input S is at logic level "1" (S = 1), the NAND Gate **Y** has at least one of its inputs at logic "0" therefore, its output Q must be at a logic level "1" (NAND Gate principles). Output Q is also fed back to input A and so both inputs to the NAND Gate **X** are at logic level "1", and therefore its output Q must be at logic level "0". Again NAND gate principals. If the Reset input R changes state, and now becomes logic "1" with S remaining HIGH at logic level "1", NAND Gate **Y** inputs are now R = "1" and B = "0" and since one of its inputs is still at logic level "0" the output at Q remains at logic level "1" and the circuit is said to be "Latched" or "Set" with Q = "1" and Q = "0".

Reset State

In this second stable state, Q is at logic level "0", Q = "0" its inverse output Q is at logic level "1", not Q = "1", and is given by R = "1" and S = "0". As gate **X** has one of its inputs at logic "0" its output Q must equal logic level "1" (again NAND gate principles). Output Q is fed back to input B, so both inputs to NAND gate **Y** are at logic "1", therefore, Q = "0". If the set input, S now changes state to logic "1" with R remaining at logic "1", output Q still remains LOW at logic level "0" and the circuit's "Reset" state has been latched.

Truth Table for this Set-Reset Function

State	S	R	Q	Q
Set	1	0	1	0
	1	1	1	0
Reset	0	1	0	1
	1	1	0	1
Invalid	0	0	1	1

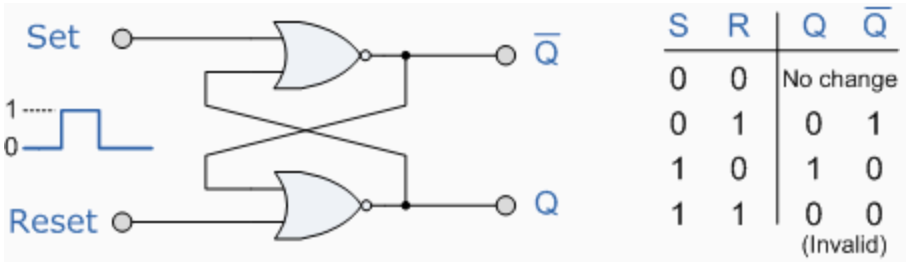
It can be seen that when both inputs S = "1" and R = "1" the outputs Q and Q can be at either logic level "1" or "0", depending upon the state of inputs S or R BEFORE this input condition existed. However, input state R = "0" and S = "0" is an undesirable or invalid condition and must be avoided because this will give both outputs Q and Q to be at logic level "1" at the same time and we would normally want Q to be the inverse of Q. However, if the two inputs are now switched HIGH again after this condition to logic "1", both the outputs will go LOW resulting in the flip-flop becoming unstable and switch to an unknown data state based upon the unbalance. This unbalance can cause one of the outputs to switch faster than the other resulting in the flip-flop switching to one state or the other which may not be the required state and data corruption will exist. This unstable condition is known as its **Meta-stable** state.

Then, a bistable latch is activated or Set by a logic "1" applied to its S input and deactivated or Reset by a logic "1" applied to its R. The SR Latch is said to be in an "invalid" condition (Meta-stable) if both the Set and Reset inputs are activated simultaneously.

As well as using NAND Gates, it is also possible to construct simple 1-bit **SR Flip-flops** using two NOR Gates connected the same configuration. The circuit will work in

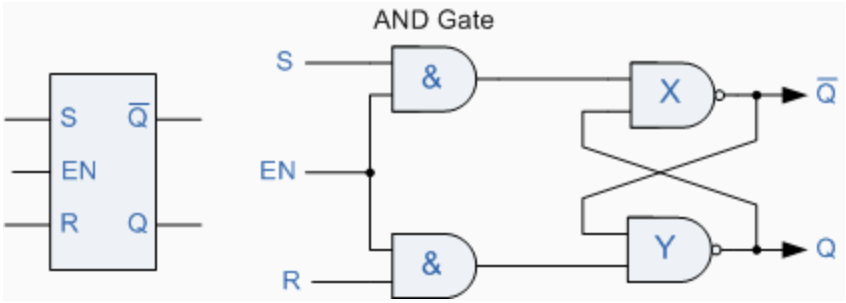
a similar way to the NAND gate circuit above, except that the invalid condition exists when both its inputs are at logic level "1" and this is shown below.

The NOR Gate SR Flip-flop



Gated or Clocked SR Flip-Flop

It is sometimes desirable in sequential logic circuits to have a bistable SR flip-flop that only change state when certain conditions are met regardless of the condition of either the Set or the Reset inputs. By connecting a 2-input NAND gate in series with each input terminal of the SR Flip-flop a Gated SR Flip-flop can be created. This extra conditional input is called an "Enable" input and is given the prefix of "EN" as shown below.



When the Enable input "EN" is at logic level "0", the outputs of the two AND gates are also at logic level "0", (AND Gate principles) regardless of the condition of the two inputs S and R, latching the two outputs Q and Q into their last known state. When the enable input "EN" changes to logic level "1" the circuit responds as a normal SR bistable flip-flop with the two AND gates becoming transparent to the Set and Reset signals. This enable input can also be connected to a clock timing signal adding clock synchronisation to the flip-flop creating what is sometimes called a "Clocked SR Flip-flop".

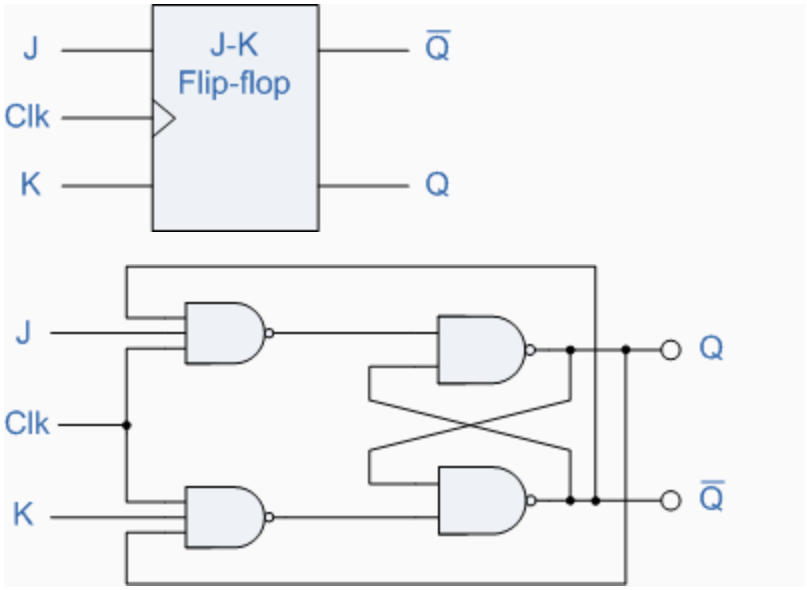
So a **Gated Bistable SR Flip-flop** operates as a standard Bistable Latch but the outputs are only activated when a logic "1" is applied to its EN input and deactivated by a logic "0".

The JK Flip-Flop

The **JK Flip-Flop** is basically a Gated SR Flip-Flop with the addition of clock input circuitry that prevents the illegal or invalid output that can occur when both input S equals logic level "1" and input R equals logic level "1". The symbol for a JK Flip-flop

is similar to that of an **SR Bistable** as seen in the previous tutorial except for the addition of a clock input.

The 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. The two 2-input NAND gates of the gated SR bistable have now been replaced by two 3-input AND gates with the third input of each gate connected to the outputs 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 usefully used to turn it into 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 the Q through the lower AND gate. If the circuit is "**Reset**" the K input is inhibited by the "0" status of Q through the upper AND gate. When both inputs J and K are equal to logic "1", the JK flip-flop changes state and the truth table for this is given below.

The Truth Table for the JK Function

J	K	Q	Q	
0	0	0	0	same as for the SR Latch
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	1	toggle action
1	1	1	0	

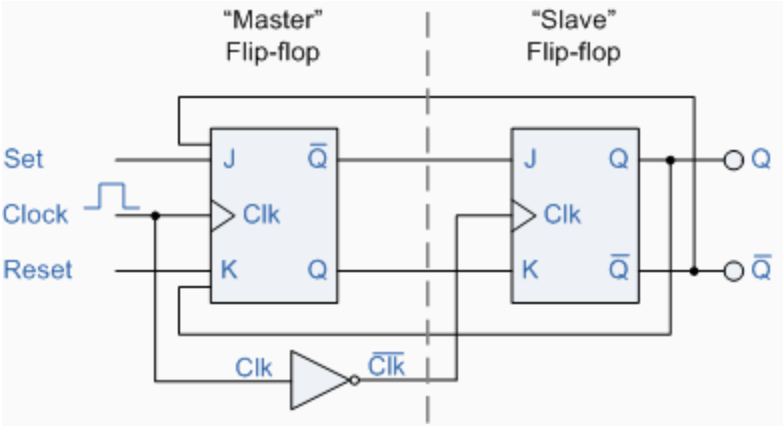
Then the JK Flip-flop is basically an SR Flip-flop with feedback and which enables only one of its two input terminals, either Set or Reset at any one time thereby eliminating the invalid condition seen previously in the SR Flip-flop circuit. Also when both the J and the K inputs are at logic level "1"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

Master-Slave JK Flip-flop

The **Master-Slave Flip-Flop** is basically two JK bistable flip-flops connected together in a series configuration with the outputs from Q and Q from the "Slave" flip-flop being 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 as shown below.

Master-Slave JK Flip-Flops

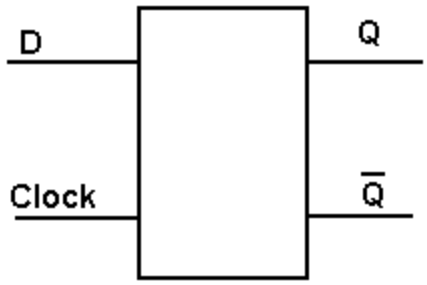


The input signals J and K are connected to the "Master" flip-flop which "locks" the input while the clock (Clk) input is high at logic level "1". As the clock input of the "Slave" flip-flop is the inverse (complement) of the "Master" clock input, the outputs from the "Master" flip-flop are only "seen" by the "Slave" flip-flop when the clock input goes "LOW" to logic level "0". Therefore on the "High-to-Low" transition of the clock pulse the locked outputs of the "Master" flip-flop are fed through to the JK inputs of the "Slave" flip-flop 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.

Clocked D Type Flip-Flop



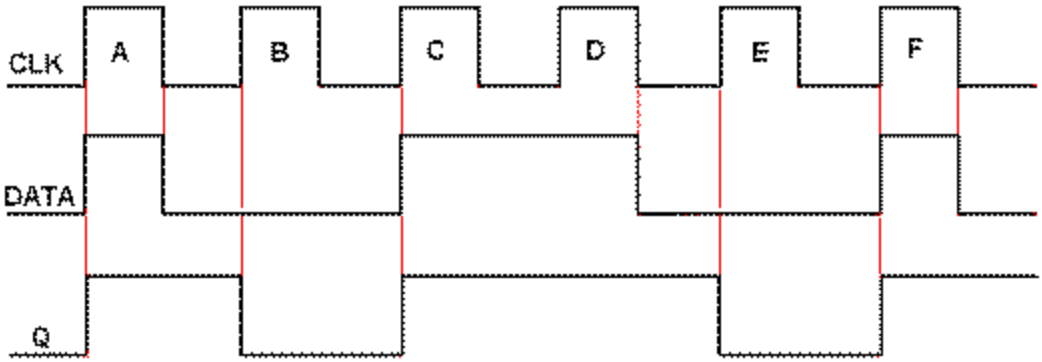
The D type flip-flop has only one input (D for Data) apart from the clock.

The INDETERMINATE state is avoided with this flip-flop.

When the clock goes high, D (a 0 or a 1) is transferred to Q.

When the clock goes low, Q remains unchanged.

Q stores the data until the clock goes high again, when new data may be available.

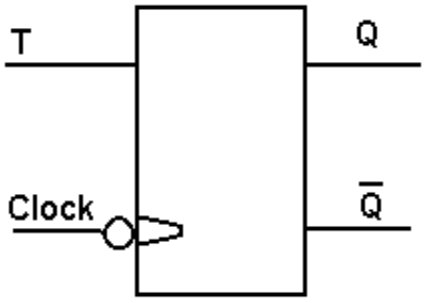


At A, clock and data are high.
Q goes high and stays high until B.

At B, clock is high and data is low.
Q goes low and stays low until C.

At C, clock and data are both high.
Q goes high and stays high until E.
Q does not change during clock pulse D, because clock and data are still both high.

T Type Flip-Flop



This flip-flop toggle (Q changes state) on the negative going edge of the clock pulse.

T acts as an ENABLE / INHIBIT control.

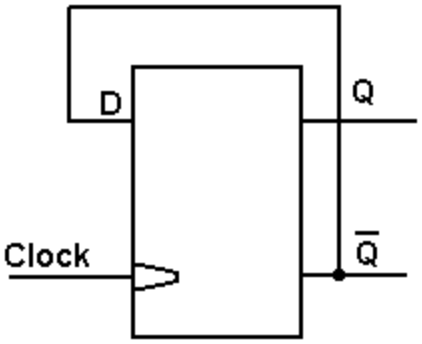
Q will only toggle on the negative edge of the clock pulse, when T is high.

Below is shown a D type flip-flop connected as a toggle type.

On each clock pulse positive going edge, Q will go to the state bar Q was before the clock pulse arrived.

Remember that bar Q is the opposite level to Q.

Therefore Q will toggle.



UNIT 5

Shift Registers

Shift Registers consists of a number of single bit "D-Type Data Latches" connected together in a chain arrangement so that the output from one data latch becomes the input of the next latch and so on, thereby moving the stored data serially from either the left or the right direction. The number of individual Data Latches used to make up **Shift Registers** are determined by the number of bits to be stored with the most common being 8-bits wide. Shift Registers are mainly used to store data and to convert data from either a serial to parallel or parallel to serial format with all the latches being driven by a common clock (Clk) signal making them Synchronous devices. They are generally provided with a Clear or Reset connection so that they can be "SET" or "RESET" as required.

Generally, Shift Registers operate in one of four different modes:

Serial-in to Parallel-out (SIPO)

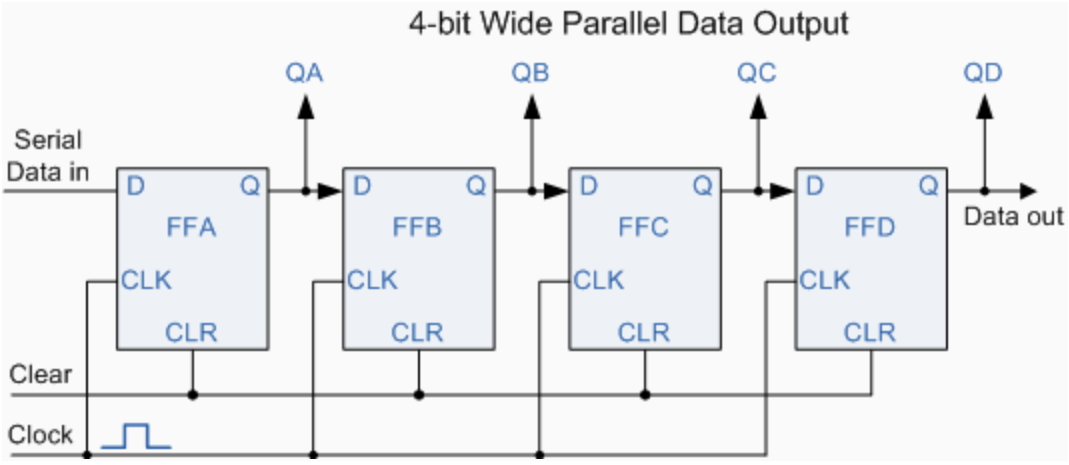
Serial-in to Serial-out (SISO)

Parallel-in to Parallel-out (PIPO)

Parallel-in to Serial-out (PISO)

Serial-in to Parallel-out.

4-bit Serial-in to Parallel-out (SIPO) Shift Register



Lets assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs QA to QD are at logic level "0" ie, no parallel data output. If a logic "1" is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and the resulting QA will be set HIGH to logic "1" with all the other outputs remaining LOW at logic "0". Assume now that the DATA input pin of FFA has

returned LOW to logic "0". The next clock pulse will change the output of FFA to logic "0" and the output of FFB and QB HIGH to logic "1". The logic "1" has now moved or been "Shifted" one place along the register to the right. When the third clock pulse arrives this logic "1" value moves to the output of FFC (QC) and so on until the arrival of the fifth clock pulse which sets all the outputs QA to QD back again to logic level "0" because the input has remained at a constant logic level "0".

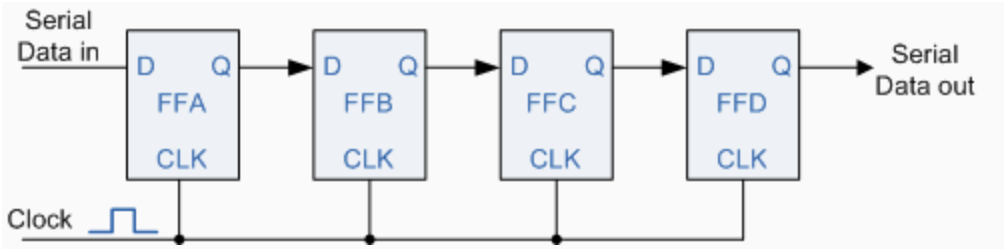
The effect of each clock pulse is to shift the DATA contents of each stage one place to the right, and this is shown in the following table until the complete DATA is stored, which can now be read directly from the outputs of QA to QD. Then the DATA has been converted from a Serial Data signal to a Parallel Data word.

Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0

Serial-in to Serial-out

This Shift Register is very similar to the one above except where as the data was read directly in a parallel form from the outputs QA to QD, this time the DATA is allowed to flow straight through the register. Since there is only one output the DATA leaves the shift register one bit at a time in a serial pattern and hence the name **Serial-in to Serial-Out Shift Register**.

4-bit Serial-in to Serial-out (SISO) Shift Register

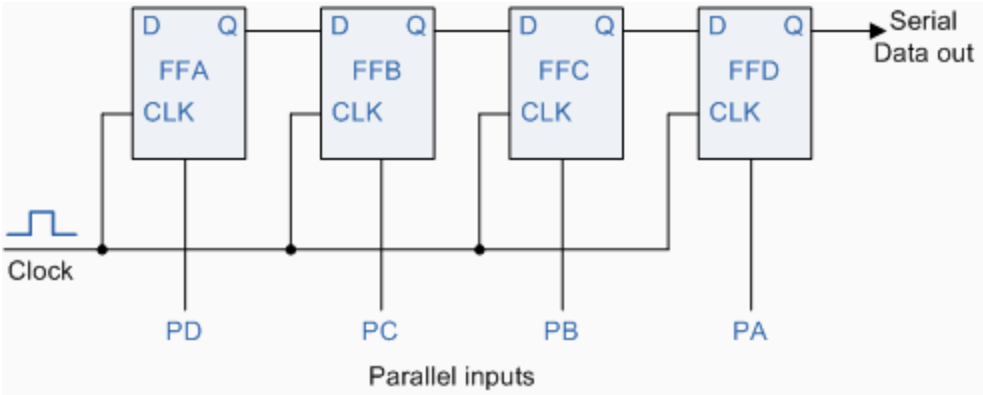


This type of **Shift Register** also acts as a temporary storage device or as a time delay device, with the amount of time delay being controlled by the number of stages in the register, 4, 8, 16 etc or by varying the application of the clock pulses. Commonly available IC's include the 74HC595 8-bit Serial-in/Serial-out Shift Register with 3-state outputs.

Parallel-in to Serial-out

Parallel-in to Serial-out Shift Registers act in the opposite way to the Serial-in to Parallel-out one above. The DATA is applied in parallel form to the parallel input pins PA to PD of the register and is then read out sequentially from the register one bit at a time from PA to PD on each clock cycle in a serial format.

4-bit Parallel-in to Serial-out (PISO) Shift Register

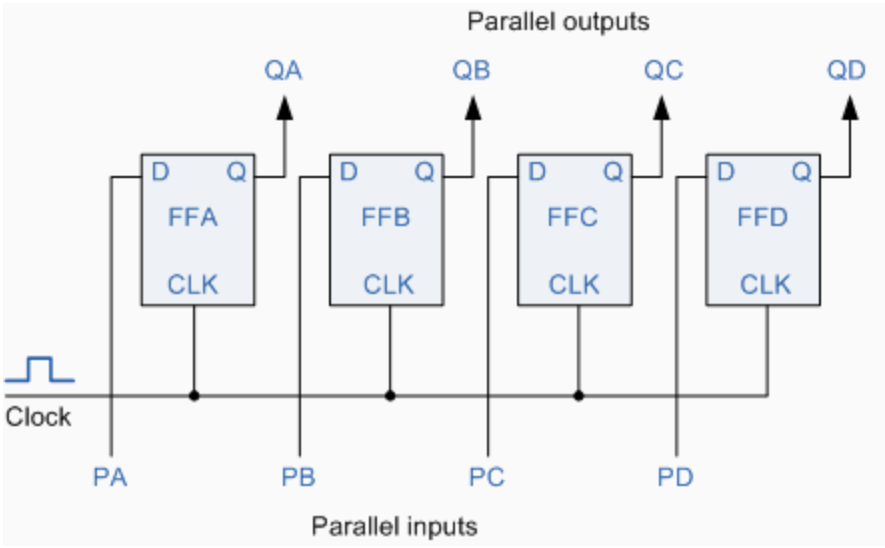


As this type of Shift Register converts parallel data, such as an 8-bit data word into serial data it can be used to multiplex many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line. Commonly available IC's include the 74HC165 8-bit Parallel-in/Serial-out Shift Registers.

Parallel-in to Parallel-out

Parallel-in to Parallel-out Shift Registers also act as a temporary storage device or as a time delay device. The DATA is presented in a parallel format to the parallel input pins PA to PD and then shifts it to the corresponding output pins QA to QD when the registers are clocked.

4-bit Parallel-in/Parallel-out (PIPO) Shift Register



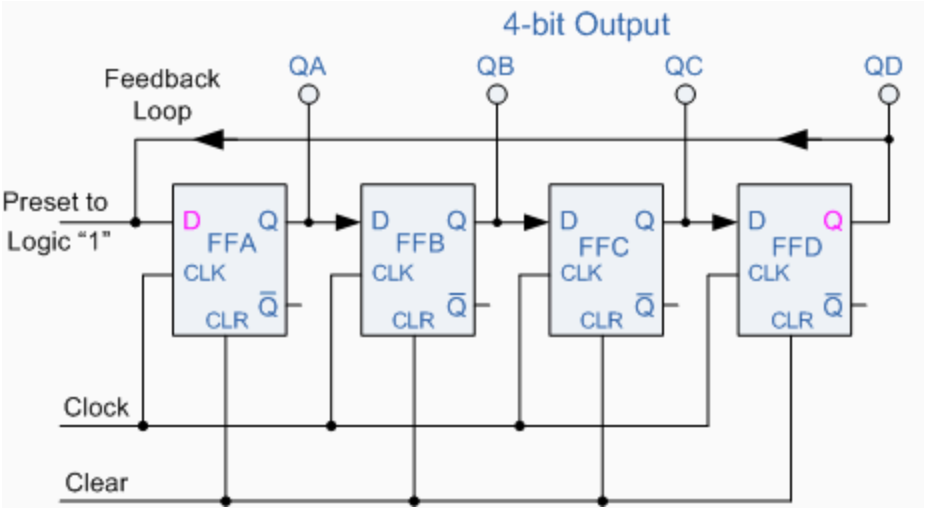
As with the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses.

Today, high speed bi-directional universal type **Shift Registers** such as the TTL 74LS194, 74LS195 or the CMOS 4035 are available as a 4-bit multi-function devices that can be used in serial-serial, shift left, shift right, serial-parallel, parallel-serial, and as a parallel-parallel Data Registers, hence the name "**Universal**".

Ring Counters

In the previous **Shift Register** tutorial we saw that if we apply a serial data signal to the input of a Serial-in to Serial-out Shift Register, the same sequence of data will exit from the last flip-flip in the register chain after a preset number of clock cycles thereby acting as a time delay to the original signal. But what if we were to connect the output of the Shift Register back to its input, we then have a closed loop circuit that "Recirculates" the DATA around a loop, and this is the principal operation of **Ring Counters** or **Walking Ring Counter**. Consider the circuit below.

4-bit Ring Counter



The synchronous **Ring Counter** example above, will re-circulate the same DATA pattern between the 4 Flip-flops over and over again every 4th clock cycle, as long as the clock pulses are applied to it. But in order to cycle the DATA we must first "Load" the counter with a suitable DATA pattern for it to work correctly as all logic "0"'s or all logic "1"'s outputted at each clock cycle would make the ring counter invalid.

For **Ring Counters** to operate correctly they must start with the first flip-flop (FFA) in the logic "1" state and all the others at logic "0". To achieve this, a "CLEAR" signal is firstly applied to all the Flip-flops in order to "RESET" their outputs to a logic "0" level and then a "PRESET" pulse is applied to the input of the first Flip-flop (FFA) before the clock pulses are applied. This then places a single logic "1" value into the circuit of the Ring Counters.

The ring counter example shown above is also known as a "**MODULO-4**" or "MOD-4" counter since it has 4 distinct stages and each Flip-flop output has a frequency equal to one-fourth or a quarter (1/4) that of the main clock frequency. The "MODULO" or

"MODULUS" of a counter is the number of states the counter counts or sequences through before repeating itself and a ring counter can be made to output any MODULO number and a "MOD-N" Ring Counter will require "N" number of Flip-flops connected together. For example, a MOD-8 Ring Counter requires 8 Flip-flops and a MOD-16 Ring Counter would require 16 Flip-flops.