NUMBER SYSTEMS

Introduction: Modern computers do not work with decimal numbers. Instead of, they process binary numbers, groups of Os and 1s. Because electronic devices are most reliable when designed for two – states (Binary) operation either on or off. Entering binary numbers into computer becomes tedious. Therefore octal and hexadecimal numbers are widely used to compress long strings of binary numbers.

1.1 Number System: In general, in any number system there is an ordered set of symbols known as digits with rules defined for performing arithmetic operations like addition, subtraction, multiplication and division. A collection of these digits makes a number which in general has two parts – integer and fractional, set a part by a radix point (.), that is

$$(N)_b = \underbrace{d_{n-1}, d_{n-2} \text{ ---- } d_2, \ldots\ldots d_1, \ldots\ldots\ldots d_0}_{\text{Integer portion}} \underset{\text{radix Point}}{.} \underbrace{d_{-1}, d_{-2} \ldots\ldots d_{-f} \ldots\ldots d_{-m}}_{\text{fractional portion}}$$

The digits in a number are placed side by side and each position in the number is assigned a weight or index. Table 1.1 gives the details of commonly used number systems.

Number system	Base or radix (b)	symbol used (di or d_{-f})	weight assigned to position		Example
Binary	2	0,1	2^{-i}	2^{-f}	1011.11
Octal	8	0,1,2,3,4	8^{-i}	8^{-f}	3567.25
Decimal	10	0,1,2,3,4,5	10^{-i}	10^{-f}	3974.57
Hexadecimal	16	0,1,2,3,4,5,6,7,8 A,B,C,D,E,F	16^{-i}	16^{-f}	3FA9.56

1.2. Binary number system: The number system with base (or radix) two is known as the binary number system. Only two symbols are used to represent numbers in this system and these are 0 and 1. these are known as bits. It is a positional system that is every position is assigned a specific weight.

Table 1.2 illustrates counting in binary number system. The corresponding decimal numbers are given in the right – hand column. Similar to decimal number system the left – most bit is known as **most significant bit (MSB)** and the right – most bit is known as the **least significant bit(LSB).** Any number of Os can be added to the left of the number without changing the value of the number. A group of four bits is known as nibble and a group of eight bits is known as a byte.

Binary number				**Decimal Number**	
B_3	B_2	B_1	B_0	D_1	D_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	2
0	0	1	1	0	3
0	1	0	0	0	4
0	1	0	1	0	5
0	1	1	0	0	6
0	1	1	1	0	7
1	0	0	0	0	8
1	0	0	1	0	9
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	2
1	1	0	1	1	3
1	1	1	0	1	4
1	1	1	1	1	5

Table 1.2 4- bit binary numbers and their corresponding decimal numbers.

Binary – to – decimal – conversation:

Any binary number can be converted into its equivalent decimal number using the weights assigned to each bit position. Since only two digits are used, the weights are powers of 2. these weights are2^0 (Units), 2^1 (two s), 2^2 (fours) 2^3 (eights) and 2^4 (sixteen). If longer binary number involved, the weights continue in ascending powers of 2

The decimal equivalent of a binary number equals the sum of all binary number equal the sum of all binary digits multiplied by their weights.

Example 1.1 : Find the decimal equivalent of binary number 11111_2.

Solution: The equivalent decimal number is,

$$= 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 16 + 8 + 4 + 2 + 1$$
$$= (31)_{10}$$

The steps involved in fast and easy conversion.

1. Write the binary number
2. Write the weights 1,2,4, 8 ………… under binary digits.
3. Cross out any weight under a 0 .
4. Add the remaining weights.

Example 1.2 : Convert binary number $1\ 1\ 0\ 1_2$ into decimal number.

1)	1	1	0	1	write binary number
2)	2^3	2^2	2^1	2^0	write weights
3)	8	4	2	1	cross weight under 0
4)	1 x 8 +1 x 4 + 2 x 0 + 1 x 1 = 13				add weights

Example 1.3 : Convert $1\ 1\ 1\ 0\ 1\ 0\ 1_2$ into decimal number

1	1	1	0	1	0	1	
64	32	16	$\cancel{8}$	4	$\cancel{2}$	1	117

Example 1.4: Determine the decimal numbers represented by the following binary numbers.

a) 101101. 10101 n) 1001 . 0101

Solution: $(101101.10101)_2 =$

$$1x\,2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1}$$
$$+ 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5}$$
$$= 32 + 0 + 8 + 4 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} + 0 + \frac{1}{32}$$
$$= (45.65625)_{10}$$

$b)$ $\quad (1001.0101)_2 \quad = 8 + 0 + 0 + 1 + 0 = 0.25 + 0 + 0.625$
$$= (9.3125)_{10}$$

Decimal – to binary conversation: Any decimal number can be converted into its equivalent binary number. For integers, the conversion is obtained by continuous division by 2 and keeping track of the remainders, while for fractional parts, the conversion is effected by continuous multiplications by 2 and keeping track of the integers generated. The conversion process is illustrated by the following examples.

How to convert decimal 13 to its binary equivalent.

$$\begin{array}{r|lcr}
2 & 13 & & \\ \hline
2 & 6 & - & 1 \\ \hline
2 & 3 & - & 0 \\ \hline
2 & 1 & - & 1 \\ \hline
2 & 0 & - & 1 \\ \hline
\end{array}$$

$13_{10} = 1\,1\,0\,1_2$ ——— Read down to up of remainders which is equalent to binary number.

In this final division, 2 does not divide into therefore, the quotient is 0 with a remainder of 1.

Whenever you arrive at a quotient of 0 with a remainder of 1, the conversion is finished the reminders when read downward give the binary equivalent. In this example, binary 1 1 0 1 is equivalent to decimal 13.

Example 1.5: convert decimal 23 to binary.

2	23	
2	11 –	1
2	5 –	1
2	2 –	1
2	1 –	0
2	0 –	1

Read binary number 23_{10} = $\longrightarrow$ 10111_2
Read down to up and becomes binary equialent number
This says that binary 1 0 1 1 1 is equivalent to decimal 23.

Example 1.5: Convert (0.65625) $_{10}$ to an equivalent base –2 number.

Solution:

0.65625	0.31250	0.62500	0.25000
X 2	X 2	X 2	X 2
1.31250	0.62500	1.25000	0.50000
1	**0**	**1**	**0**

0.50000
X 2

1.00000

Thus, (0.62500) $_{10}$ = (0.10101) $_2$

Example 1.6: Express the following decimal numbers in the binary form.

a)25.5　　　　b) 10.625

Solution:

a) Integer part:

Thus, $(25)_{10} = (11001)_2$

$$\begin{array}{r|lcr}
2 & \underline{25} & & \\
2 & \underline{12} & - & 1 \\
2 & \underline{6} & - & 0 \\
2 & \underline{3} & - & 0 \\
2 & \underline{1} & - & 1 \\
2 & \underline{0} & - & 1
\end{array}$$

Read down to up

Fraction part

$$\begin{array}{r}
0.\ 5 \\
\text{X}\ \ 2 \\
\hline
1\ .\ 0 \\
\hline
\end{array}$$

1

i.e $0\ .\ 5_{10} = 0.1_2$

Therefore $25.5_{10} = 11001.1_2$

b) Integer part $10_{10} = 1010_2$

Fractional part.

0.625	0.250	0.500
X 2	X 2	X 2
1.250	0.500	1.000
1	0	1

i.e $10.625_{10} = 1010.101_2$

Therefore , $10.625_{10} = 1010.101_2$

1.3. Binary Arithmetic: We all are familiar with the arithmetic operations such as additions, subtraction, multiplication and division of decimal numbers. Similar operations can be performed on binary numbers. Binary arithmetic is much simpler than decimal arithmetic because here only two digits, 0 and 1 are involved

Binary addition: The rules of binary addition are given in the following table.

Augend	0	0	1	1	1	carry in from
Addend	+0	+1	+0	+1	1	previous table
Sum	0	1	1	(1)0	(1)1	

carryout to next most place

Example 1.7: Add the following binary numbers.
i) 1011 and 1100 ii) 0101 and 1111

```
Solution:i)  1   0   1   1    ii)     0   1   0   1
        (+)  1   1   0   0    ( + )   1   1   1   1

        ->1  0   1   1   1  Carry 1   0   1   0   0
```

Binary subtraction: The rules of binary subtraction are given in the table.

minus end	0	1	1	0
subtrats end	-0	-1	-0	-1
difference	0	0	1	1

The first three rules are the same as in decimal subtraction. The last rule requires a borrow from the next most significant place. The minuend is then binary 10 and the subtrahend is 1 with a difference of 1.

Example 1.8: perform the following subtraction.

```
i)   1011                 ii) 01010101
__ 0110                   __00111001
```

```
Solution: i) Subtraction        ii)              10
  0  10                                    10    0    10
  1  0     1     1          0    1    0    1    0     1    0    1
  0  1     1     0     __   0    0    1    1    1     0    0    1
_____________________________________________________________________
     1     0     1          0    0    1    1    1     1    0
_____________________________________________________________________
```

Binary multiplication:

The rules of binary multiplication are given in the following table.

Multiplied	0	1	0	1
Multiplier	X 0	X 0	X 1	X 1
Product	0	0	0	1

When the multiplier is 1 in binary multiplication, the multiplicand is copied as the product. When the multiplier is 0, the product is always 0.

Example 1.9 perform the following multiplication

```
      i)    1101              ii)    1001
          X  101                   X 1101
```

Sol: i) 1 1 0 1 X 1 0 1 ii) 1 0 0 1 X 1 1 0 1

```
         1 1 0 1                    1 0 0 1
       00 0 0                    0  0  0 0
     1 10 1                    1 0 0 1
   1 1 1                    1 0 0 1
  ______________           _______________
   1 0 1 1 0 0 0 1            1 1 1 0 1 0 1
  ______________           _______________
```

Observe: That binary multiplication is similar to decimal multiplication.

Binary division: Binary division is obtained using the same procedure as decimal division. An example of binary division is given below.

Example. 1.10 divide 1 1 1 0 1 01 by 1 0 0 1

Solution: 1 1 0 1<------------------ quotient

Divisior: 1001) 1 1 1 0 1 0 1 <-- Dividend

```
          1 0 0 1
          _______
            1 0 1 1
            1 0 0 1
          _______
              1 0 0 1
              1 0 0 1
          _______
                  0
```

1.4. 1s complement of binary number: In a binary number, if each 1 is represented by 0 and each 0 by 1, the resulting number is known as the one's complement of the first number. In fact, both the numbers are complement of each other.

Example 1.11: find the one's complement of the following.

Binary numbers. a) 1011000110 b) 00100101

Solution : a) 1 0 1 1 0 0 0 1 1 0 1^s is 0 1 0 0 1 1 1 0 0 1

b) 00100101 1^s complement number is 11011010

Two's complement: If 1 is added to is complement of a binary number, the resulting number is known as the two's complement of the binary number. It is also observed that the 2's complement of the 2's complement of a number is the number is the number itself.

Example 1.12: Find the two's complement of the following.

	a)	b)
Solution: a) Number	0 1 0 0 1 1 1 0	0 0 1 1 0 1 0 1
is complement	1 0 1 1 0 0 0 1	1 1 0 0 1 0 1 0
add 1	1	1
2's complen	1 0 1 1 0 0 1 0	1 1 0 0 1 0 1 1

Octal number system : The number system with base (or redixy) eight is known as the octal number system. In this system eight symbols, 0, 1 , 2 ,3,4,5,6, and 7 are used to represent the number. Similar to decimal and binary number systems, it is also a positional system and has, in general, two parts : Integer and fractional , set a part by a radix point. For example (6327. 4051)$_8$ is an octal number. Using the weights it can be written as.

$$\begin{aligned}(6327.4057)_8 &= 6x8^3 + 3x8^2 + 2x8^1 + 7x8^0 + \\ &\quad 4x8^{-1} + 0x8^{-2} + 5x8^{-3} + 1x8^{-4} \\ &= 3072 + 192 + 16 + 7 + \frac{4}{8} + 0 + \frac{5}{512} + \frac{1}{4096} \\ &= (3287.5100098)_{10}\end{aligned}$$

Using the above procedure, an octal number can be converted into an equivalent decimal number. The conversion from decimal to octal is similar to the conversion procedure from decimal to binary. The only difference is that number 8 is used tin place of 2 for division in the case of integers and for multiplication in the case of fractional numbers.

Example 1.13

a) Convert 247_{10} into octal

b) convert0.6875_{10} into octal

c) convert 3287.5100098_{10} into octal

Solution

	Quotient	Remainder
8 | 247		
8 | 30 − 7	30	7
8 | 3 − 6	3	6
8 | 0 − 3	0	3

Read the number from down to up reminder part 3 6 7

b) 0.6 8 7 5   0 . 5 0 0 0

X 8   X 8

5.5 0 0 0   4. 0 0 0 0

5   **4**   Thus $(0.6875)_{10} = (0.54)_{10}$

c) Integer Part:

	Quotient	Remainder
8 | 3287		
8 | 410 − 7	410	7
8 | 51 − 2	51	2
8 | 6 − 3	6	3
8 | 0 − 6	0	6

Now read the number from down to up as $6327_8 = 3287_{10}$

$$\begin{array}{r} 0.5100098 \\ \text{X } 8 \\ \hline 4.0800784 \\ \big| \\ 4 \end{array} \qquad \begin{array}{r} 0.0800784 \\ \text{X } 8 \\ \hline 0.6406272 \\ \big| \\ 0 \end{array} \qquad \begin{array}{r} 0.6406272 \\ \text{X } 8 \\ \hline 5.1250176 \\ \big| \\ 5 \end{array}$$

$$\begin{array}{r} 0.1250176 \\ \text{X } 8 \\ \hline 1.0001408 \end{array}$$

Therefore $3287.5100098_{10} = 6327.4051_8$

Octal to binary: Octal numbers can be converted into equivalent binary numbers by replacing each octal digit by its 3 – bit binary equivalent. The following table 1.3 gives octal

Numbers and their binary equivalents for decimal numbers 0 to 15.

octal	Decimal	Binary
0	0	0 0 0
1	1	0 0 1
2	2	0 1 0
3	3	0 1 1
4	4	1 0 0
5	5	1 0 1
6	6	1 1 0
7	7	1 1 1
10	8	0 0 1 0 0 0
11	9	0 0 1 0 0 1
12	10	0 0 1 0 1 0
13	11	0 0 1 0 1 1
14	12	0 0 1 1 0 0
15	13	0 0 1 1 0 1
16	14	0 0 1 1 1 0
17	15	0 0 1 1 1 1

Example 1.14: convert 736_8 into an equivalent binary number.

Solution: From the above table, the binary equivalents of 7, 3 and 6 are 111, 011 & 110 respectively. Therefore $736_8 = 1\,1\,1\,0\,1\,1\,1\,1\,0_2$

Binary to octal: Binary numbers can be converted into equivalent octal numbers by making groups of three bits starting from LSB and moving towards MSB for integer part of the number and then replacing each group of three bits by its octal representation. For fractional part the groupings of three bits are made starting from the binary point.

Example 1. 15: a) convert 100110_2 to its octal equivalent.
Solution :

$$\begin{aligned} 100110_2 &= 001 \quad 001 \quad 110_2 \\ &= 1 \quad 1 \quad 6_8 \\ &= 116_8 \end{aligned}$$

Octal number:

$$\begin{aligned} 0.10100110_2 &= 0.101 \quad 001\ 100_2 \\ &= 0.5 \quad 1 \quad 4_8 \\ &= 0.514_8 \end{aligned}$$

Application of octal number system:

It is highly in convenient to handle long strings of binary numbers while entering into the digital systems. It may cause errors also. Therefore, octal numbers are used for entering binary data and displaying certain information.

Hexadecimal Number system: Hexadecimal number system is very popular in computer uses. The base for hexadecimal number system is 16 which requires 16 distinct symbols to represent the number. These are numerals 0 through 9 and alphabets A through F. this is an alphanumeric number system because its uses both alphabets and numerical to represent a hexadecimal number. Table 1.4 gives hexadecimal number with their binary equivalents for decimal numbers 0 through 15.

Table 1.4 Binary and decimal equivalents of hexadecimal numbers.

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

Hexadecimal – to decimal conversion: Hexadecimal numbers can be converted to their equivalent decimal numbers.

Example 1.16: Obtain decimal equivalent of hexadecimal number 3A . 2F $_{16}$ solution.

$$3A.\,2F_{16} = 3x16^{1} + 10x16^{0} + 2x16^{-1} + 15\,x16^{-2}$$

$$= 48 + 10 + \frac{2}{16} + \frac{15}{16^{2}}$$

$$= 58.1836_{10}$$

Decimal – to Hexadecimal conversion: For conversion from decimal to hexadecimal the procedure used in binary as well as octal system is applicable, using 16 as the dividing (n for integer part) and multiplying (for fractional part) factor.

Example 1. 17: convert the following decimal number into hexadecimal number.

Solution:

Integer Part	**Quotient**	**Remainder**
16 | 95	5	15
16 | 5 - 15		
16 | 0 - 5	0	5

5 F

Thus $95_{10} = 5F_{16}$

Fraction part:

0.5
x16
8.0

8 thus $0.5_{10} = 0.8_{16}$

Hexadecimal – to – binary conversion:
Hexadecimal numbers can be converted into equivalent binary numbers by replacing each hex digit by its equivalent 4 – bit binary numbers.

Example 1.18: convert 2 F 9 A $_{16}$ to equivalent binary number using table 1.4 find the binary equivalent of each hex digit.

$$2F.9\,A_{16} = 0010\ 1111\ 1001\ 1010_2$$

$$= 0010111110\ 011010_2$$

Binary – to Hexadecimal conversion: Binary numbers can be converted into the equivalent hexadecimal numbers by making groups of four bits starting from LSB and moving towards MSB for integers part and then replacing each group of four bits by its hexadecimal representation.

For the fractional part, the above procedure is repeated starting from the bit next to the binary point and moving towards the right.

Example 1.19. Convert the following binary numbers to their equivalent hex numbers.

a) 10100110101111
b) 0.00011110101101.

Solution:

$$a)\quad 10100110101111_2 = \underbrace{0010}_{2}\ \underbrace{1001}_{9}\ \underbrace{1011}_{A}\ \underbrace{1111}_{F}$$

$$\therefore 10100110101111_2 = 29AF_{16}$$

$$b)\quad 0.00011110101101_2 = 0.\underbrace{0001}_{1}\ \underbrace{1110}_{E}\ \underbrace{1011}_{B}\ \underbrace{0100}_{4}$$

$$\therefore 0.00011110101101_2 = 0.1EB4_{16}$$

1.5. The ASCII code: To get information into and out of a computer, we need to use numbers, letters, and other symbols. This implies some kind of alphanumeric code for the I/O unit of a computer. At one time, every manufacturer had a different code, which led to all kinds of confusion. Eventually, industry settles on a input – output code known as the American Standard code for Information Interchange (abbreviated ASCII). This code allows manufacturers, to standardize I/O hardware such as keyboards, Printers, video displays, and so on.

The ASCII (Pronounced ask' – ee) code is a 7 – bit code whose format (arrangement) is

Where each x is a 0 or a 1. for instant , the letter kA is coded as

Table The ASCII code.

$X_2\ X_2 X_1\ \ X_0$ **$X_6\ X_5\ \ X_4$**

	010	011	100	101	110	111
0000	SP	0	@	P		P
0001	!	1	A	Q	A	q
0010	,,	2	B	R	b	r
0011	#	3	C	S	C	S
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	,	7	G	W	g	w
1000	‘	8	H	X	h	x
1010	*	9	I	Y	i	y
1011	+	:	J	Z	j	z
1100	,	;	K			
1101	-	=	M			
1110	.	>	N		n	
1111	1	?	0		0	

Table shows the ASCII code. Read the table the same as a graph. For instance the letter A has an xxx of 100 and an $X_3\ X_2\ X_1\ X_0$ of 0001. Its ASCII code is

100 0001 (A)

Table includes the ASCII code for lowercase letters. The letter is coded as.

110 0001 (a)

1.6.Logic gates: Circuits used to process digital signals are called logic gates. Logic symbols are used to identify these circuits. They have one or more input signals but only one output signal. Gates are often called logic circuits because they can be analyzed with Boolean algebra

Logic Gate Truth Tables

As well as a standard Boolean Expression, the input and output information of any **Logic Gate** or circuit can be plotted into a table to give a visual representation of the switching function of the system and this is commonly called a **Truth Table**. A logic gate truth table shows each possible input combination to the gate or circuit with the resultant output depending upon the combination of these input(s).

For example, consider a single **2-input** logic circuit with input variables labeled as A and B. There are "four" possible input combinations or 2^2 of "OFF" and "ON" for the two inputs. However, when dealing with Boolean expressions and especially logic gate truth tables, we do not general use "ON" or "OFF" but instead give them bit values which represent a logic level "1" or a logic level "0" respectively.

Then the four possible combinations of A and B for a 2-input logic gate is given as:

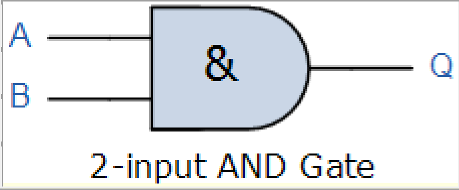
- Input Combination 1. - "OFF" - "OFF" or (0, 0)
-
- Input Combination 2. - "OFF" - "ON" or (0, 1)
-
- Input Combination 3. - "ON" - "OFF" or (1, 0)
-
- Input Combination 4. - "ON" - "ON" or (1, 1)

Therefore, a 3-input logic circuit would have 8 possible input combinations or 2^3 and a 4-input logic circuit would have 16 or 2^4, and so on as the number of inputs increases. Then a logic circuit with "n" number of inputs would have 2^n possible input combinations of both "OFF" and "ON". In order to keep things simple to understand, we will only deal with simple **2-input** logic gates, but the principals are still the same for gates with more inputs.

The Truth tables for a 2-input AND Gate, a 2-input OR Gate and a NOT Gate are given as:

2-input AND Gate

For a 2-input AND gate, the output Q is true if BOTH input A "AND" input B are both true, giving the Boolean Expression of: (Q = A and B).

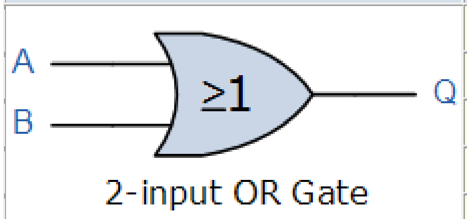
Symbol	Truth Table		
	A	B	Q
	0	0	0
	0	1	0
	1	0	0

	1	1	1
Boolean Expression Q = A.B	Read as A **AND** B gives Q		

Note that the Boolean Expression for a two input AND gate can be written as: A.B or just simply AB without the decimal point.

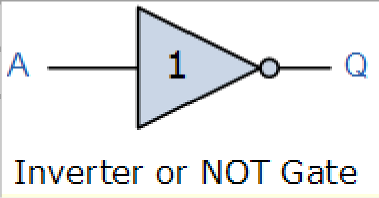
2-input OR (Inclusive OR) Gate

For a 2-input OR gate, the output Q is true if EITHER input A "OR" input B is true, giving the Boolean Expression of: (Q = A or B).

Symbol	Truth Table		
	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Boolean Expression Q = A+B	Read as A **OR** B gives Q		

NOT Gate

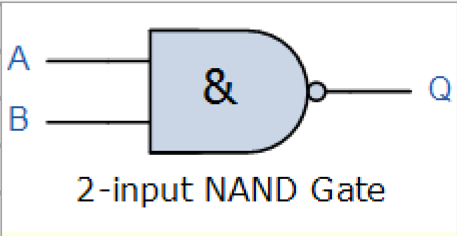
For a single input NOT gate, the output Q is ONLY true when the input is "NOT" true, the output is the inverse or complement of the input giving the Boolean Expression of: (Q = NOT A).

Symbol	Truth Table	
	A	Q
	0	1
	1	0
Boolean Expression Q = NOT A or A	Read as inverse of A gives Q	

The NAND and the NOR Gates are a combination of the AND and OR Gates with that of a NOT Gate or inverter.

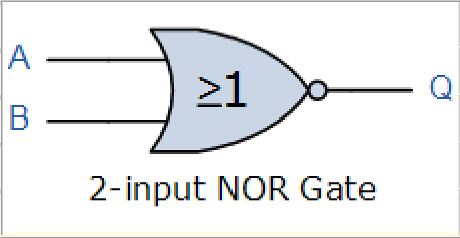
2-input NAND (Not AND) Gate

For a 2-input NAND gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not(A and B)).

Symbol	Truth Table		
	A	B	Q
	0	0	1
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A .B	Read as A **AND** B gives NOT-Q		

2-input NOR (Not OR) Gate

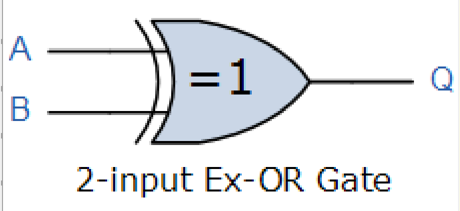
For a 2-input NOR gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not(A or B)).

Symbol	Truth Table		
	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	0
Boolean Expression Q = A+B	Read as A **OR** B gives NOT-Q		

As well as the standard logic gates there are also two special types of logic gate function called an Exclusive-OR Gate and an Exclusive-NOR Gate. The actions of both of these types of gates can be made using the above standard gates however, as they are widely used functions, they are now available in standard IC form and have been included here as reference.

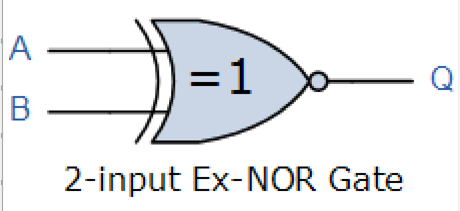
2-input EX-OR (Exclusive OR) Gate

For a 2-input Ex-OR gate, the output Q is true if EITHER input A or if input B is true, but NOT both giving the Boolean Expression of: (Q = (A and NOT B) or (NOT A and B)).

Symbol	Truth Table		
	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A⊕B			

2-input EX-NOR (Exclusive NOR) Gate

For a 2-input Ex-NOR gate, the output Q is true if BOTH input A and input B are the same, either true or false, giving the Boolean Expression of: (Q = (A and B) or (NOT A and NOT B)).

Symbol	Truth Table		
	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression Q = A ⊕ B			

Summary of all the 2-input Gates described above.

The following Truth Table compares the logical functions of the 2-input logic gates above.

Inputs		Truth Table Outputs for each Gate					
A	B	AND	NAND	OR	NOR	EX-OR	EX-NOR
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

The following table gives a list of the common logic functions and their equivalent Boolean notation.

Logic Function	Boolean Notation
AND	A.B
OR	A+B
NOT	A
NAND	A .B
NOR	A+B
EX-OR	(A.B) + (A.B) or A$\oplus$B
EX-NOR	(A.B) + or A $\oplus$ B

2-input logic gate truth tables are given here as examples of the operation of each logic function, but there are many more logic gates with 3, 4 even 8 individual inputs. The multiple input gates are no different to the simple 2-input gates above, So a 4-input AND gate would still require ALL 4-inputs to be present to produce the required output at Q and its larger truth table would reflect that.

1.10. Full adder: A logic circuit that cans odd 3 bits at a time is to referred to as a full adder. The third bit is the carry from, a lower column. Fig 1.11 Shows how to build a full adder. The output of the EX – OR gate is called the sum. While the output of the or gate is the carry.

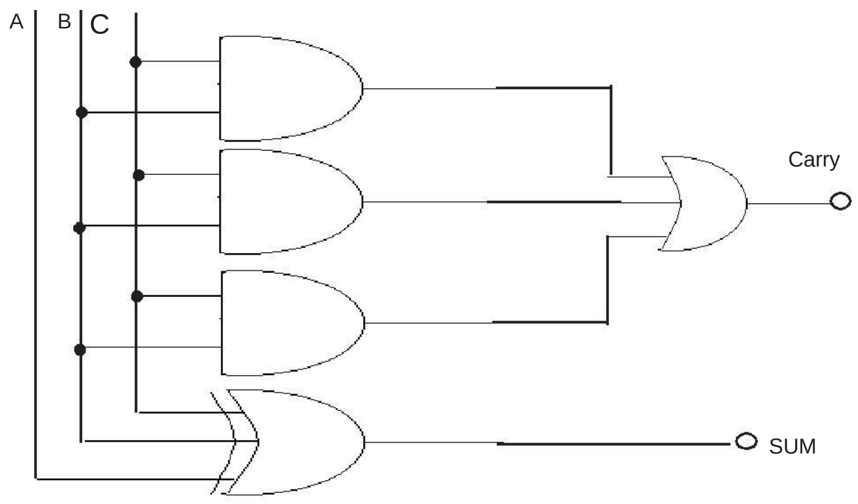


Fig. No. 1.11 Full -addeer

Fig. 1.9. A full adder

Table 1.13. full adder truth table

A	B	C	CARRY	SUM
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

Table 1.13 shows the truth table of a full adder. When you examine each entry in table you can see that a full adder performs. Binary addition on 3 bits.

Half – Subtractor: A logic circuit for the subtraction of B (subtrahend) from A (minuend) where A and B are 1 bit numbers is referred to as a half – subtractor. Here, A and B are the two inputs and difference and borrow are the two outputs.

Fig. 1.10 Realization of a half – subtractor.

Table 1.14 Truth table of a half - subtractor

A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

2- FLIP – FLOPS , REGISTERS AND COUNTERS

2.0.Introduction :

Gates are decision making element. They can perform binary addition and subtractions. But these elements are not enough. A computer also needs memory elements, devices that can store a binary digit. A flip – flop is one such circuit, and characteristics of the most common types of flip – flops are used in digital systems are considered in this chapter. They are used in the construction of Registers and counters, and in numerous other applications.

A register is another important digital building block. It is simply a group of flip – flops that can be used to store binary information. The other kind of Registers will modify the stored word by shifting its bits left or right. A counter is a special kind of register, designed to count the number of clock pulses arriving at its input. This chapter discusses some basic registers and counters used in micro computer. A brief introduction of Multiplier / Demultiplexer and encoder / decoder is presented.

2.2. RS LATCHES:

A flip flop is a device with two stable states; it remains in one of these states until triggered into the other. The RS latch/flip-flop , discussed in this section, is one of the simplest flip-flop.

Truth Table:

Table 2.1 summarizes the operation of the transistor latch. With both control inputs low no change can occur in the output and the circuit remains latched in the last state.

Table 7.1 Transistor Latch

R	S	Q	COMMENTS
0	0	NC	No Changes
0	1	1	Set
1	0	0	Reset
1	1	*	Race

When R is low and S is high, the circuit sets the Q output to a high. One the other hand, if R is high and S is low the Q output resets to a low.

RACE CONDITION

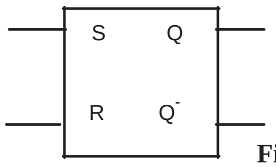


Fig. No.2.3

Look at the last entry in table 7.1 R and S are simultaneously. This is called a race condition; it is never used because it leads to unpredictable operation. If both control inputs are high both transistors saturated when the R and S inputs return to low, both transistor try to come out of saturation. It is a race between the transistors to see which one de saturates first. The faster transistor (the one with the shorter saturation delay time) will win the race and latch the circuit. If the faster transistor is on the left side of fig C, the Q output will be 10 W. If the faster transistor is on the right side, the Q output will go high.

NOR LATCHES:

A discrete circuit like Fig. 2.1 C is rarely used because we are in the age of integrated circuits. Now a days you build Rs. Latches with nor gates or NAND gates.

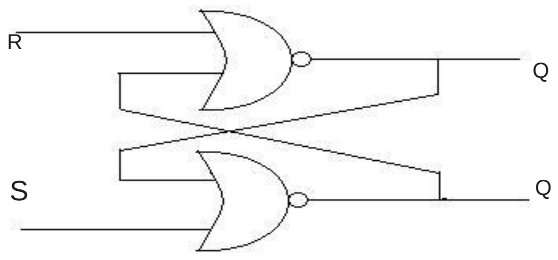


Fig. No.2.4

Fig. 2.2 shows how it's done with NOR gates.

It is the most basic and useful sequential logic circuit. It is called the R – S flip – flop because it has two inputs. The S(set) and R (reset) inputs. The two outputs denoted by $\overline{Q}$ and Q are complementary to each other. Whenever, Q is at the state 1 , the $\overline{Q}$ is at the state 0 and vice versa.

Table 2.2 NOR latch.

R	S	Q	COMMENTS
0	0	NC	No Changes
0	1	1	Set
1	0	0	Reset
1	1	*	Race

As shown in table a low R and a low S give us the inactive state; that is remain circuit remains in the same state. A low R and a high S represent the set state, while a high R and a low S give the reset state. Finally, a high R and a high S produce a race conditions therefore, we must avoid R = S = 1 when using a **NOR** latch.

NAND LATCHES:

A slightly different latch can be constructed by using NAND gates as shown in figure 2.3 to understand how this circuit functions, recall that a low on any input to a nand gate will force its output high. Thus a low R and high S set a Q to low and Q = 0 A high R and low S reset Q to low.

Because of NAND gate inversion, the in active and race conditions are reversed.

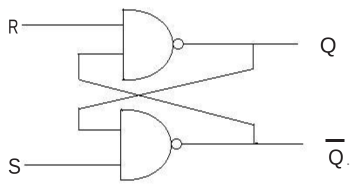


Fig. No.2.5

R	S	Q	Comments
0	0	*	Race
0	1	1	Set
1	0	0	Reset
1	1	NC	No Changes

2.3 NAND gate latch and truth table.

Clocked RS flip – flop

Two different methods for constructing an RS flip – flop were discussed in the previous section with NOR gate and NAND gate realization. Both of these RS flip – flops or latches, are said to be “transparent “; that is any change in input at R or S is transmitted immediately to the output at Q and Q thus they acts as short term memory.

It is possible to store or clock the flip – flop in order to store information (set it or reset it) at any time, and then hold the stored information for any desired period of time. This flip – flop is called clocked RS flip – flop.

The circuit of a clocked RS flip – flop is shown in figure 2.6 with its symbol and truth table.

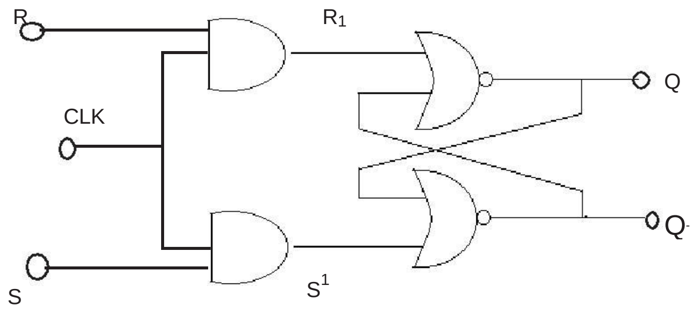


Fig. No. 2.6

CLK	R	S	Q
0	0	0	NC
0	0	1	NC
0	1	0	NC
0	1	1	NC
1	0	0	NC
1	0	1	1
1	1	0	0
1	1	1	* (Race)

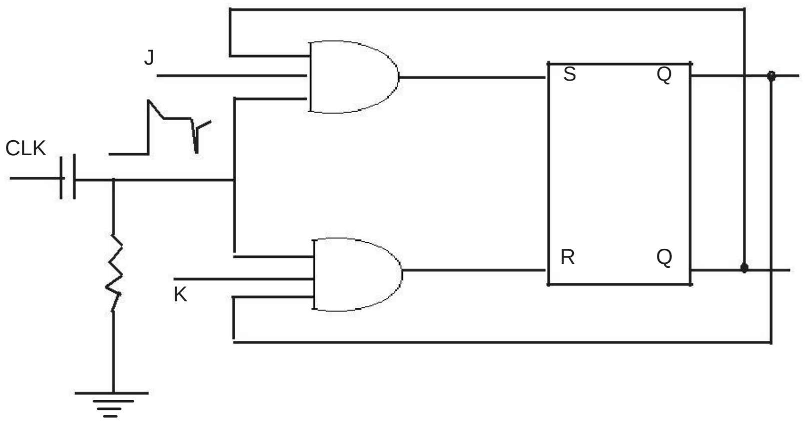
Fig. 2.4 clocked RS flip – flop x symbol and its truth table

It consists of two additional and gates added at the input of R S flip – flop. In addition to control inputs R and S, there is a clock input CK.

The output of the two and circuits (S^1 and R^1) will be 0 as long as CK = 0. then the state of the flip – flop will remain unchanged and if S = 0 , R = 1 then S = 0 R = 1 and the flip – flop is reset to 0. On the other hand if S = 1 , R = 0 then S =1 R = 1 and the flip flop is set to 1. the presence of R^1 =1 and S^1 =1 , will however , results in an undetermined state.

JK FLIP – FLOP

Fig. 2.5 shows one way to build a JK flip – flop the variables J and K are called control inputs because they determine what the flip – flop does on the arrival of a positive clock edge.



CLK	K	L	Q
0	X	X	NC
1	X	X	NC
↓	X	X	NC
X	0	0	NC
↑	0	1	0
↑	1	0	1
↑	1	1	Toggle

Fig2.7 JK flip flop symbol and its truth table.

Inactiver: When J and K are both low, both AND gates are disabled and the circuit is inactive at all times including the rising edge of the clock.

Reset : When J is low and K is high, the upper gate is disabled; so there is no way to set the flip - flop. The only possibility is reset. When Q is high, the lower gate passes a reset trigger as soon as the next positive clock edge arrives. This forces Q to become low. Therefore, J=0 and K=1 means that the next positive clock edge resets the flip-flop.

Set: When J is high and K is low, the lower gate is disabled; So it is impossible to reset the flip-flop. But you can set the flip-flop as follows. When Q is low, Q is high; therefore, the upper gate passes a set trigger on the positive clock edge. This drives Q into the high state. That is , J=1 and K =0 means that the next positive clock edge sets the flip - flop.

Toggle: When J and K both are high (notice that this is the forbidden state with an RS flip - flop), it is possible to set or reset the flip - flop. If Q is high, the lower gate passes a reset trigger on the next positive clock edge. When Q is low, the upper gate passes a set trigger on the next positive clock edge. Either way Q changes to the compliment of the last state. Therefore, J =1 and K=1 means that the flip - flop will toggle on the next positive clock edge.
(“toggle” means switch to opposite state)

JK MASTER - SLAVE FLIP-FLOP:

a master - slave flip-flop is a combination of two clocked latches. The first one is called master and second one is the slave. Here are master is positively clocked while slave is negatively clocked.
This means:
1. While the clock is high, the master is active and slave is inactive.
2. While the clock is low, the master is inactive and slave is active.

Fig. 2.8. shows the logic diagram of JK master - slave flip - flop; symbol and its truth table.

CLK	**J**	**K**	**Q**
X	0	0	NC
⎍	0	1	0
⎍	1	0	1
⎍	1	1	Toggle

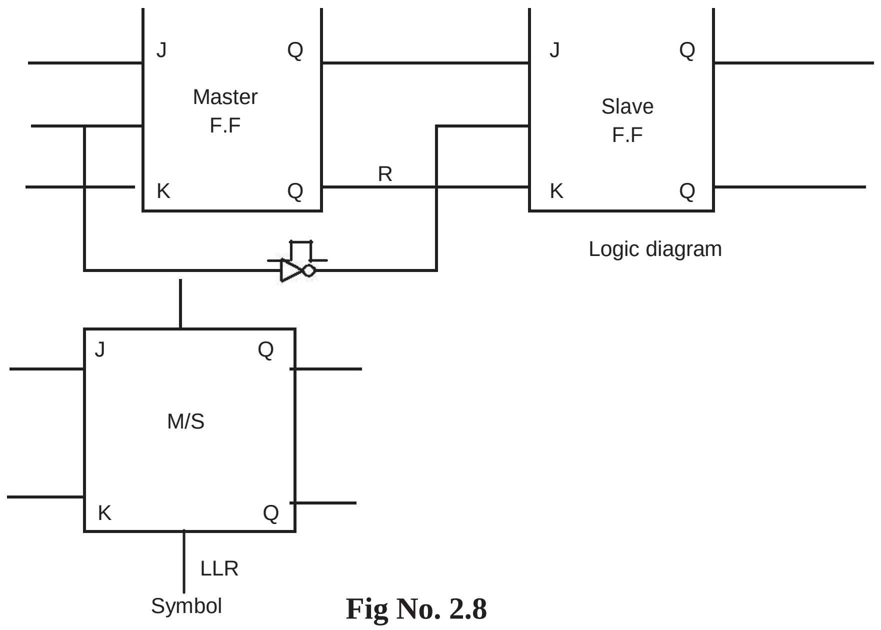


Fig No. 2.8

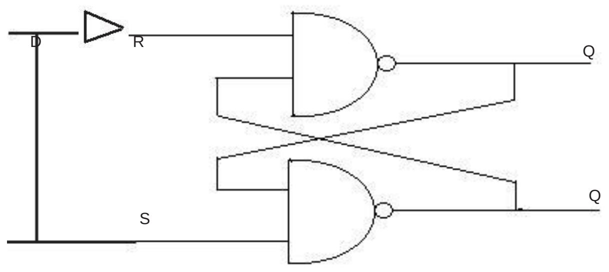
Set: To start the analysis, let’s assume low Q and high $\overline{Q}$. For an input condition of high J, law K and high CLK, the master goes into the set state, producing high S and Low R. Nothing happens to the Q and Q outputs because the slave is inactive while the clock is high. When the clock goes low, the high S and low R forces the slave into the set state, producing a high Q and a low Q.

Reset: When the slave is set, Q is high and Q is low. for the input condition of low J, high K and high CLK, the master will reset, forling S to go low and R to go high. Again, no changes can occur in Q and Q because tghe slave is in active while the clock is high. When the clock returns to low state, the low S and high R forces the slave to reset.

Toggle: If the J and K inputs are both high, the master toggles once while the clock is high and then slave toggles once when the clock goes low.

The basic idea behind the master slave flip - flop is that, every action the master slave flip - flop is that, every action of the master with high CLK is copied by the slave when CLK goes low. It is used to avoid racing condition.

D Flip - flop: We will modify the design of Rs flop flop to eliminate the possibility of a raced condition. the result is a new kind of flip - flop known as D latch.



D	Q
0	0
1	1

Fig. No. 2.9

Fig. 2.9. shows one way to build a D latch (unlocked) because of the inverter, data bit D drives the S input of a NAND latch and compliment D drives the R input. Therefore, a high D sets the latch, and low D reset it. Observe, there is no race condition in this truth table. The inverter guarantees that S and R will always be in opposite states. Therefore it is impossible to set up a race condition in the D latch.

Applications of Flip - Flops: Some of the common uses of flip - flops are:

1) latch
2) Registers
3) Counters
4) Memory (RAM)

2.3. Need for a Register: As discussed in previous section a flip – flop can store
1 – bit of digital information (1 or 0) it is also referred to as a 1- bit register.

But in digital system a situation will a raise to store more than 1 – bit of information. For example, the binary number 0101 is called as 4 bit binary number. A 4 – bit register is needed to store this 4 bit binary number. There are 4 flip flop’s in an 4 bit register 8 flip flops in an 8 bit register, and so on.

The 8 bit and 16 bit registers are extensively used as a part of CPU of a 8/ 16 – bit microprocessors. They are involved in all mathematical / logical operations that can be performed by CPU and in storing the information temporarily.

Register: . A register is simply a group of flip – flops (memory elements) that can be used to store binary information. There must be one flip – flop for each bit in the binary number. For example a register used to store an 8 – bit binary number must have eight flip – flops.

Right – Shift Register: Fig. 2.12 is a shift right register. As shown each Q output sets up the D input of the preceding flip flop. When the positive clock pulse arrives, the stored bits move one position to the right.

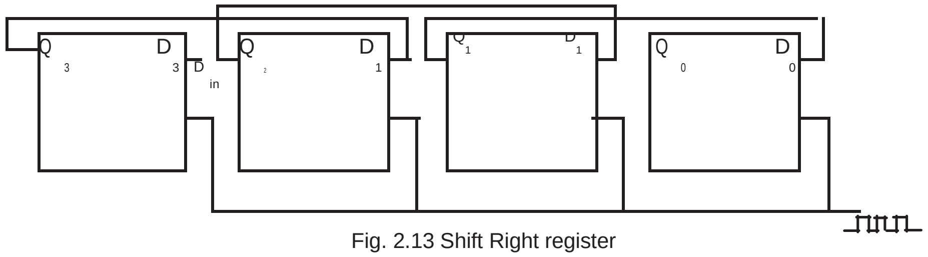


Fig. 2.13 Shift Right register

As an example , consider Q = 0000 and Dim = 1 (That is D == 1) All data inputs except the one on the left are O s. The arrival of first clock pulse sets the left flip – flop and the stored word becomes, Q = 1000.

This new word means D as well as D_1 now equals 1 . when the next clock pulse hits, the Q_2 flip – flop sets and the stored word becomes Q = 1100

The third clock pulse give Q = 1110 and the forth clock pulse give Q = 1111.

2.70. Counters: A counter is a special kind of register, designed to count the number of clock pulses arriving at its input. It is one of the most useful subsystem in a digital systems. The input to this counter is a rectangular waveform called clock. Each time the clock signal changes state from low to high, the counter will add one (1) to the number stored in its slip flop. This means the counter will count the number of clock transitions from low to high.

A clock having a small circle (bubble) in the input side would count clock transitions form high to low. Since clock pulses occur at know intervals, the counter can be used as an instrument for measuring time and therefore frequency. There are basically two different types of counters synchronous and Asynchronous.

A ripple counter can be constructed by use of clocked JK flip flops as shown in the Fig. 2.14. here negative edge triggered, JK flip – flops are connected in cascade. The clock pulse (square wave) drivers flip – flop A. the output of A drives B, and the output of B drives flip flop C. All the J and K inputs are tied to + Vcc. This means that each flip – flop will change state(

toggles) with negative transition at its input.

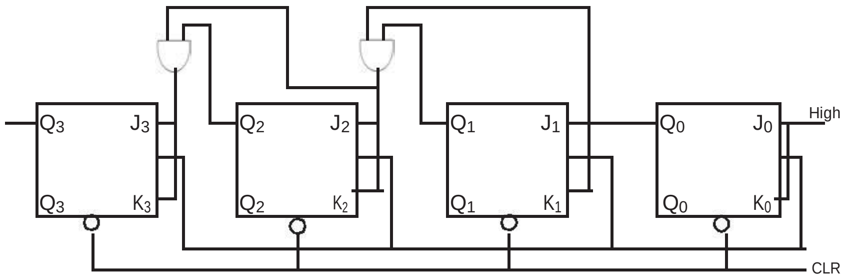


Fig. 2-16 Synchronous counter

The above waveforms shows the action of counter. Let's assume that all the flip – flops are reset to produce 0 outputs. If we consider A output as least – significant Bit (L.S.B) and C the most – significant Bit (MSB) then the contents of the counter is CBA = 000

For every clock transition flip flop A will change state. This is indicated by arrows ( ) on the time line. Thus at point a on the time line A goes high, at

point B it goes back low, at C it goes back high and so on.

Since A acts as clock for B, for each negative transition in A , flip – flop B will toggle. Thus at point b on the time line B goes high; it then goes low at d and toggles back high again at pint F.

Similarly B acts as clock for C , for each negative transition in B, flip – flop C, will toggle. Thus C goes high at point d and goes back at point h.

If we observe the output condition of all flip flop it is the binary number equivalent to the number of negative clock transitions that are occurring. Prior to point A, the output is CBA = 000 at point a it changes to CBA = 001, at point b it changes to 010, and so on. That is counter content advances on count with each negative clock transition. It is summarized in the truth table.

It is observed that a counter having n flip – flop will have 2^n output conditions. For example. The three flip – flop counter just discussed has 2^3 = 8 output conditions (000 through 111). The largest number that can be represented by n flip – flop has decimal equivalent of (2^n- 1). In this example, it is (2^3 – 1) 7.

3- SEMICONDUCTOR MEMORIES

3.0.Introduction:

The ability to store information in an important requirement in a digital system circuits or systems designed for this purpose are called as memory. It is a part of computer where the program and data area stored. A flip flop and Register are two kinds of simple memory elements which can store a one bit and a one word of binary information.

In a large system, such as a microcomputer, a huge amount of registers each storing a binary with thousands of registers each storing a binary word. The latest generation of computer uses semiconductor memories because they are less expensive and easy to work. A typical microcomputer has a semiconductor memory of 655360 memory locations, each capable of storing 1 byte (8 – bit size) of information

3.1. Semiconductor memories:

Recent developments in semiconductor technology have provided a number or reliable and economical MSI and LSI memory circuits. The typical semiconductor memory consists of a rectangular array of memory cells. The basic memory cell is typically a transistor flip flop or a circuit capable of storing charge and used to store 1 bit of information. The two general categories of semiconductor memories are RAM and ROM. They can be further divided into various types as illustrated in fig. 1

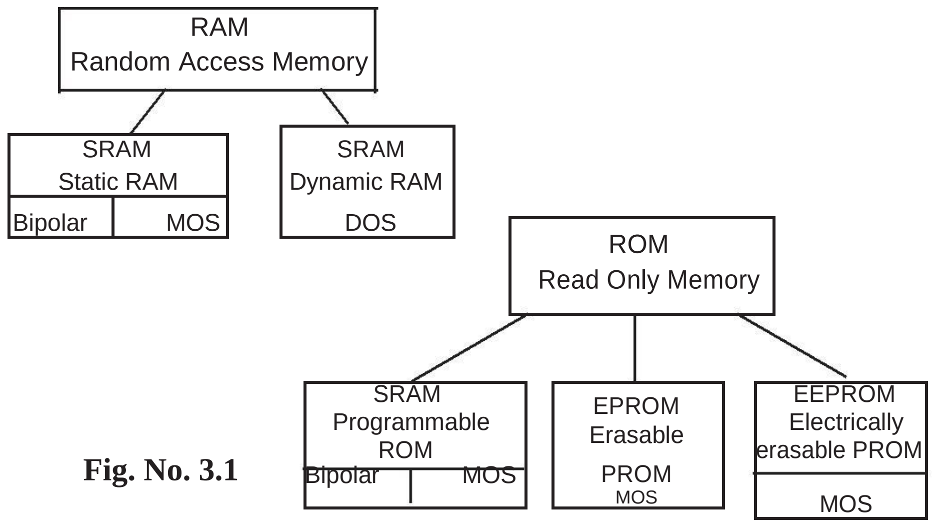


Fig. No. 3.1

3.3. Memory Terminology:

Read only memories come in four versions. The standard ROM is programmed by the manufacturer. The PROM can be programmed permanently by the user or manufacturer using special equipment. It can be programmed only once. The EPROM (Erasable Programmable Read Only Memory) can be programmed and erased by the user. Stored data in EPROM can be erased by passing high – intensity ultraviolet light through a special transparent window in the top of the IC. Another erasable PROM is the EEPROM (electrically Erasable Programmable Read Only Memory) . The EEPROM can be erased and programmed by the user with special equipment. It is erased electrically rather than with ultraviolet light. All the types of ROM s are non volatile light. All the types of ROM s area non volatile, which means they will not lose their data when power to the IC is turned off:

PROMS AND EPROMS: The term ROM is generally reserved for memory chips that are programmed by the manufacturer. You have to send a list of data to be stored in the different memory locations to the manufacturer, who then produce a mask. Now the user can be able to read the stored. Data from the ROM.

Programmable: A programmable ROM(PROM) is different. It allows the user to store the data. An instrument called a PROM programmer is used in storing the data(also called "burning in"). once this has been done, the programming is permanent, that is the stored bits cannot be erased.

Erasable: The erasable PROM(EPROM) used MOSFETs. Data is stored with a PROM programmer. Later , data can be erased with ultraviolet light passing through a window present on the top of memory chip. This releases stored charges and the effect is to wipe out the stored contents. The user can erase and store until the program and data are perfected.

EEPROM: Another type of reprogrammable ROM device is the EEPROM (Electrically Erasable Read Only Memory). It is a non volatile like EPROM but does not require ultraviolet light to erase the data. It can be erased by using electrical pulses. It can be used for slower application.

Semi conductor RAM's: Semiconductor RAMs may be static or dynamic. The static RAM uses bipolar or MOS flip – flops . A dynamic RAM uses MOSFET s and capacitors to store the data. The stored data must be refreshed (Recharged) for every few milliseconds. Because the capacitor charge leaks off.

Static RAM: The figure shows one of the flip flops used in a static MOSRAM. Q_1 and Q_2 acts like switches. Q_3 and Q_4 are active leads which means they behave like resistors. The Circuit action is similar to the transistor latch discussed in chapter2. Either Q_1 is cut off or vice versa. A static RAM will contain thousands of flip – flop like this. As long as power is applied, the flip flop
remains latched and can store the bit indefinitely.

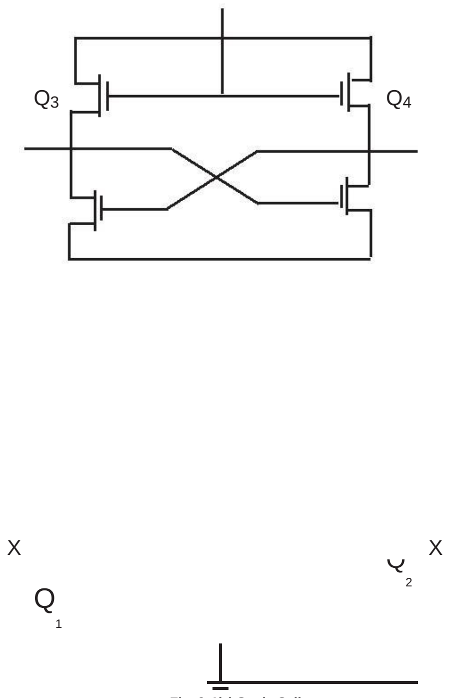


Fig. 9-4(a) Static Cell

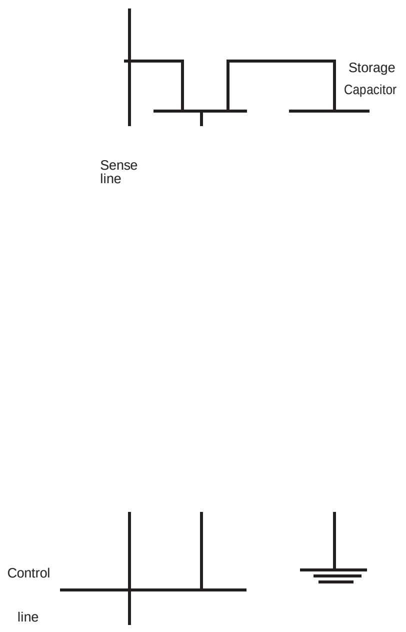


Fig. 9-4(b)

Dynamic RAM: Figure shown one of the memory elements(called calls) in a dynamic RAM. When the sense and control lines go high, the MOSFET conducts and charges the capacitor. When the sense and control – lines go low, the MOSFET opens and the capacitor retains its charge. In this way, it can store 1 bit. A dynamic RAM may contain thousand of memory cells like this.

Since only a single MOSFET and capacitor are needed, the dynamic RAM contains more memory cells than a static RAM of the same physical size. The disadvantage of dynamic RAM is the need to refresh the capacitor charge every few milliseconds.

BASIC MEMORY TERMINOLOGY:

Word Length: The number of bits stored in a memory location is called its word length. It will depend upon the number of flip – flops in a row of the register. Generally 4, 8 and 16 size of memory locations are available in

semiconductor memories.

CAPACITY: The capacity or size of a memory is defined as the number of memory locations available in the memory multiplied by the number of bits in each location. Mathematically it is given by.

$$C = M \times N$$

Where, C is the no. of memory locations and
N is the no of bits in each memory location.

The commonly used values of the number of memory location are 64, 256, 512, 1024, 2048, 4096 and the number of bits in each memory location(i.e. word length) are 1,4 and 8.

Address: The different memory locations in a memory chip are identified by its unique address. This unique address is identified by placing appropriate 0 s and 1s on the address pins. For accessing any one of M memory locations,
P address lines are required , where $2^p = m$. This set of P lines is called address bus. For example , a ROM with 256 memory locations can be accessed with 8 address lines. ($2^8 = 256$)

Write operation: Moving 1 storing the data into the selected memory location is called write operation. For writing a word into a particular memory location, the following sequence of operations is to be performed.

1) The chip select signal is applied to the CS terminal.
2) Place the DATA to be stored on the data – input terminals.
3) Place the ADDRESS of the desired memory location on the address – input terminal.
4) Finally apply a write command signal to the write control input terminal.

In response to the above operations, the information memory location is cleared off and the information present on the DATA input terminal is stored.

Read Operation: Copying/ detecting data from memory, without destroying the contents, is called read operation. For reading a data from a particular memory location, the following sequence of operation is to be performed.

1) The chip select signal is to be applied to the CS terminal.
2) Place the ADDRESS of desired memory location on the address – input terminal.
3) Apply a READ command signal to the read control input terminal.

In response to the above operations, the data from the addressed memory location appears on the DATA – OUTPUT terminal.

Write cycle Time (T_{WC}) This is the minimum amount of time for which the valid address must be present for writing a word in the memory. That is, it is the minimum time required between two successive write operations.

Read cycle time (T_{RC}) : This is the maximum time amount of time for which the valid address must be present for reading a word from the memory. That is, it is the minimum time required between two successive read operations.

Sequential access memory: It is a memory in which the words (data's) are stored – in and read out in sequence. For example if the 10th location is being accessed at a particular time, then 15th location is not accessible, unless all other intermediate locations (i.e. 11,12,13 and 14th) have been accessed one – by – one. This means that the access time is not same for all the locations.

Advantages of semiconductor memories: The following are the major advantage of semiconductor memories.

1) They are available in small size (min the form of Ics)
2) Low cost and high speed of operations
3) High Reliability
4) Easy to expand memory size(capacity)

Applications: The following are some of basic applications of semiconductor memories.

1) RAMS are used for temporary storage of user programs and data.
2) SRAMS are used for cache memory.
3) ROMs are most often used of storing permanent instruction necessary for strat up and operation of a computer.
4) In micro controllers for specialized applications.

A/ D AND D/ A CONVERTERS

4.0. Introduction: Digital systems are widely used in many applications such as communication , control , computers, instrumentations…… etc. In many of these applications , the signals are available only in analog form , but not in digital form. Hence by using digital hardware. These analog signals have to be converted into digital form. The process of converting analog signals into digital form is known as analog to digital conversion (A/D). the system which is used for this process is known as 'A/D converter (ADC)

Conversely, it is often necessary to convert the digital output of the system into analog voltage or current. This process is known as 'Digital to analog conversion' (D/A) and the system used for this purpose is known as digital to analog D/A converter (DAC).

4.1 Need for A/D and D/A conversion: In many of the applications digital systems must be interfaced with analog equipment. In such cases, the digital system is needed for A/D and D/A converters ; for the communication or the data transfer with analog equipment.

1) In the Fig. 2.1 shows a typical situation in which the digital system has analog inputs and outputs.

The input section is a variable voltage source, giving a continous voltage ranging from 0 to 10 V. then the A/D converter translates the analog input digital data. After the computer of processing the data in the digital system, the system gives the digital 1 data output. This output will be translated into an analog voltage by the D/A converter. This output analog voltage is indicated by the voltmeter.

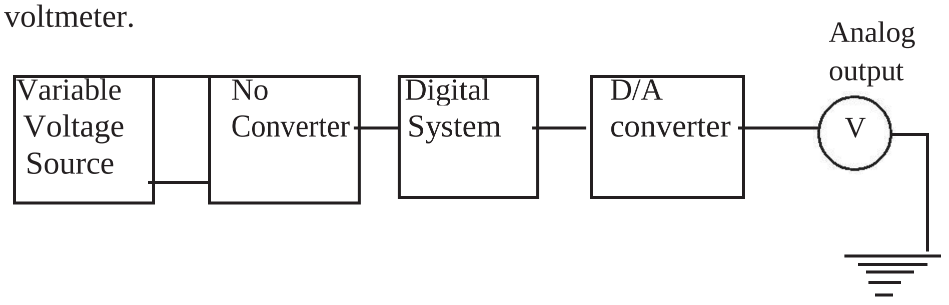


Fig. 4.1 Digital System with A/D and D/A converter

Thus the above situation necessitate A/D and D/A converters along with the digital system.

2) Similarly , a digital communication system is used to transmit the signals, which are in the form of electric signals. This necessitate or requires an D/A converter at the transmitting end and a A/D converter at the receiving end.

4.2 Methods of A/D and D/A conversions:

Digital to Analog (D/A) Conversion: The process of converting a digital signal into analog is known as 'Digital – to – analog ' (D/A) conversion. There is two types of D/A converters 1) Weighted – Register D/A converter 2) R – 2 R D/A converter.

Basic Principle of D/A conversion :

The basic principle of operation of D/A conversion is shown in the block diagram of a D/A converter in the below figure.

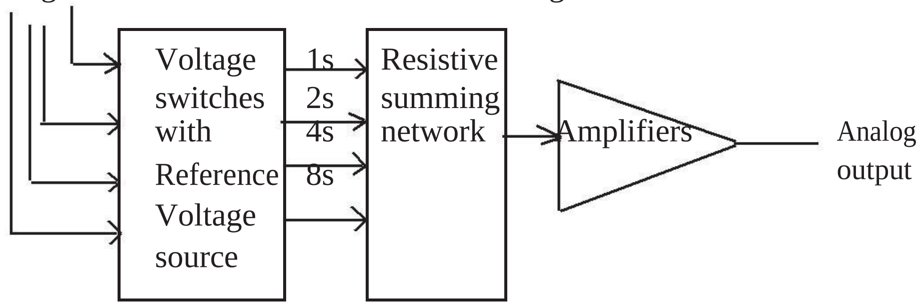

Voltage switches with Reference Voltage source
1s
2s
4s
8s
Resistive summing network
Amplifiers
Analog output

Fig. No. 4.2

Block Diagrame of D/ A converters

Let the input digital data is applied to the voltage switches. The switches are with reference voltage source (Vref); in which each source feeds a precisely regulated voltage to the switches. When the binary signals of digital data are applied on voltage switches, then the switches provide one of two possible outputs; i.e.OV or the precision source voltage. The switches feed resistive summing network which converts each bit into its weighted current value and sums them for a total current. This total value is then fed to the amplifier and gives the proper analog voltage value for the equivalent digital data. In most of the converters, operational amplifier (OP-AMP) is mostly used. Note that this amplifier performs two functions. These are current to voltage conversion and scaling. So that the output voltage of the D/A converter will be the proper value.

4.3 Counter type A/D converter: The block diagram of a counter type 4 – bit A/D converter is shown in the fig below. It employs a voltage comparator, an and gate a BCD counter and a D/A converter.

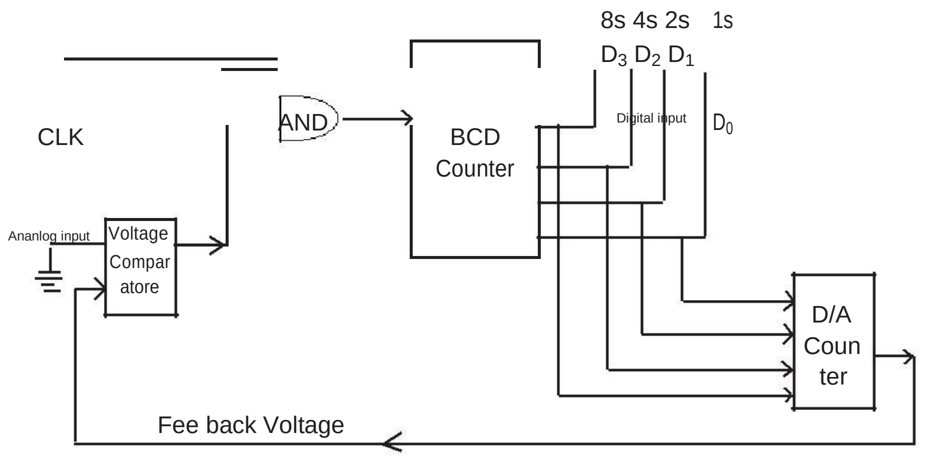


Fig. No. 4.3

Black diagram of A counter type 4 - bit aid converter

Operation : Let the counter begin RESET and the output of the D/A converter is zero. Apply the analog voltage at the input. If the analog input voltage at A is grater than the voltage at the input B of the comparator, then the output of the comparator switches to a high state and enables the AND gate. The clock pulse is allowed to increase or advance the count of the BCD counter through its binary states. The counter continues to advance from one binary state to the next producing successively higher steps and the count will be displayed at the digital or binary output as D D D D. the count on the counter increases until the feed back voltage from the D/A converter increases becomes greater than the analog input voltage. Whenever the voltage at B is greater than at A, the comparator output will go low and disables the AND gate. It results the cutting off the clock pulses to stop the counter. The state of the counter at this point equals the number of steps in the reference voltage or feed back voltage at which the comparison occurs voltage. Note that for each of sample of analog

voltage, the counter must count from zero up to point at which the feed back voltage reaches the analog input voltage.

Thus A/D converter produces the digital output of an analog input voltage.

Disadvantages : 1) Slow speed of operation is the main drawback of this method. In the worst case of maximum input, the counter must sequence through its maximum number of states before the conversion occurs. For a 4 –0 bit, 8 –

bit and 16 – bit conversions, this means a maximum of 16, 256 and 4096 counter states are required respectively.

2) For each sample of analog voltage, the counter must count from zero up to the point at which the feed back voltage reaches the analog input voltage.

3) The conversion time varies , depending upon the analog voltage.

STORAGE DEVICES USED IN COMPUTERS:

The purpose of external storage is to retain data and programs for future use. The information stored on these devices is permanent and not erased when the equipment is turned off. The popular external storage media used with computers are:

1. Floppy disk.
2. Hard Disk
3. Magnetic tapes

Floppy Disks: The most common storage medium used on small computers in s floppy disk. It is a flexible plastic disk coated with magnetic material and looks like a phonograph record. Information can be recorded or read by inserting it into a disk drive connected to the computer. The disks are permanently erased in stiff paper jackets for protection and easy handling. An opening is provided in the jacket to facilitate reading and writing of information.

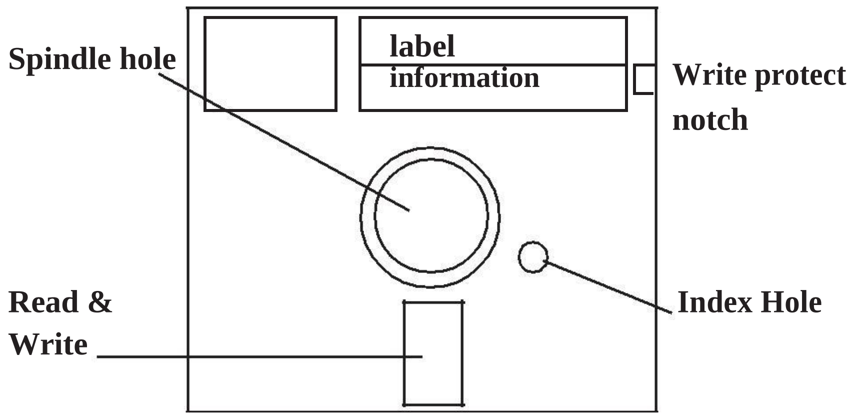


Fig. No. 5.4

Floppy disks are available in three standard sizes. 8 – inches, 5 ½ - Inches, 3 ½ Inches.

Hard Disk: Another magnetic media suitable for storing large volumes of information is the hard disk. A hard disk pack consist of two or more magnetic plates fixed to a spindle, one below the other with a set read/ write heads as shown in fig. 5. the disk pack is permanently sealed inside a casing to protect it from dust and other contaminations.
Hard disks possess a number of advantages compared to floppy disks.

1. They can hold much larger volume of information.
2. They are very fast in reading and writing.
3. The not susceptible to dust and static electricity.
4. Storage capacity ranges 10 MB to 80 MB.

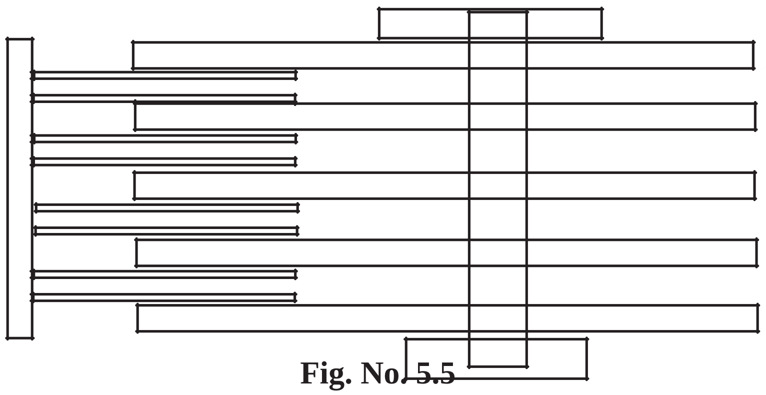


Fig. No. 5.5

Magnetic Tapes: Relatively inexpensive storage media known as magnetic tapes and are used as back up media. A standard 2,400 feet tape can store about 40 million characters and can be read at a speed of 1, 60,000 characters per second. It is like a music cassette, that is a sequential device and therefore one has to read all the previous records to reach a particular one.

5.10. STORAGE DEVICES USED IN COMPUTERS:

The purpose of external storage is to retain data and programs for future use. The information stored on these devices is permanent and not erased when the equipment is turned off. The popular external storage media used with computers are:

1. Floppy disk.
2. Hard Disk

Floppy Disks: The most common storage medium used on small computers in s floppy disk. It is a flexible plastic disk coated with magnetic material and looks like a phonograph record. Information can be recorded or read by inserting it into a disk drive connected to the computer. The disks are permanently erased in stiff paper jackets for protection and easy handling. An opening is provided in the jacket to facilitate reading and writing of information.

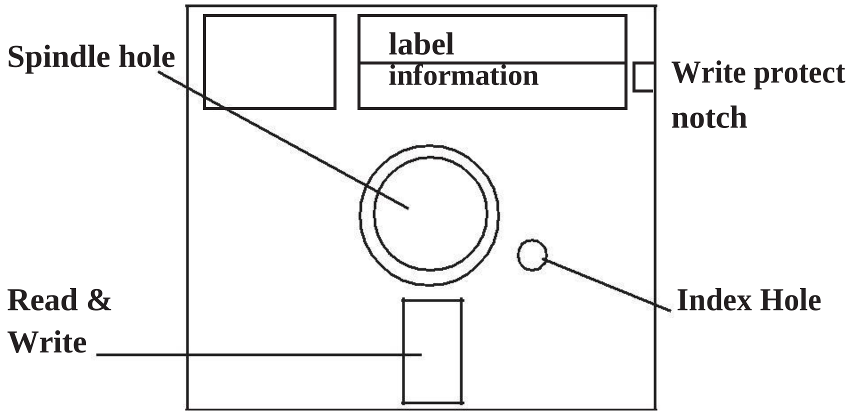


Fig. No. 5.4

Floppy disks are available in three standard sizes. 8 – inches, 5 ½ - Inches, 3 ½ Inches.

Hard Disk: Another magnetic media suitable for storing large volumes of information is the hard disk. A hard disk pack consist of two or more magnetic plates fixed to a spindle, one below the other with a set read/ write heads as shown in fig. 5. the disk pack is permanently sealed inside a casing to protect it from dust and other contaminations.

Hard disks possess a number of advantages compared to floppy disks.

1. They can hold much larger volume of information.
2. They are very fast in reading and writing.
3. The not susceptible to dust and static electricity.
4. Storage capacity ranges 10 MB to 80 MB.

Magnetic Tapes: Relatively inexpensive storage media known as magnetic tapes and are used as back up media. A standard 2,400 feet tape can store about 40 million characters and can be read at a speed of 1, 60,000 characters per second. It is like a music cassette, that is a sequential device and therefore one has to read all the previous records to reach a particular one.

The Laws of Boolean

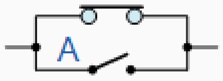
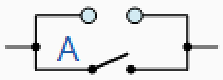
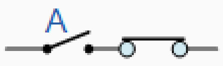
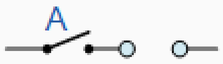
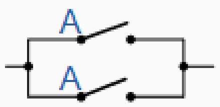
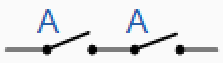
As well as the logic symbols "0" and "1" being used to represent a digital input or output, we can also use them as constants for a permanently "Open" or "Closed" circuit or contact respectively. Laws or rules for Boolean Algebra expressions have been invented to help reduce the number of logic gates needed to perform a particular logic operation resulting in a list of functions or theorems known commonly as the **Laws of Boolean**.

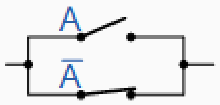
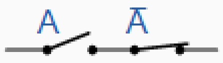
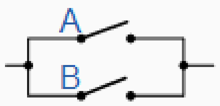
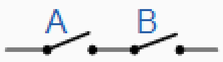
Boolean Algebra is the mathematics we use to analyse digital gates and circuits. We can use these "Laws of Boolean" to both reduce and simplify a complex Boolean expression in an attempt to reduce the number of logic gates required. *Boolean Algebra* is therefore a system of mathematics based on logic that has its own set of rules or laws which are used to define and reduce Boolean expressions.

The variables used in Boolean Algebra only have one of two possible values, a logic "0" and a logic "1" but an expression can have an infinite number of variables all labelled individually to represent inputs to the expression, For example, variables A, B, C etc, giving us a logical expression of A + B = C, but each variable can ONLY be a 0 or a 1.

Examples of these individual laws of Boolean, rules and theorems for Boolean Algebra are given in the following table.

Truth Tables for the Laws of Boolean

Boolean Expression	Description	Equivalent Switching Circuit	Boolean Algebra Law or Rule
A + 1 = 1	A in parallel with closed = "CLOSED"		Annulment
A + 0 = A	A in parallel with open = "A"		Identity
A . 1 = A	A in series with closed = "A"		Identity
A . 0 = 0	A in series with open = "OPEN"		Annulment
A + A = A	A in parallel with A = "A"		Indempotent
A . A = A	A in series with A = "A"		Indempotent
NOT A = A	NOT NOT A		Double Negation

	(double negative) = "A"		
A + A = 1	A in parallel with not A = "CLOSED"		Complement
A . A = 0	A in series with not A = "OPEN"		Complement
A+B = B+A	A in parallel with B = B in parallel with A		Commutative
A.B = B.A	A in series with B = B in series with A		Commutative
A+B = A.B	invert and replace OR with AND		de Morgan's Theorem
A.B = A+B	invert and replace AND with OR		de Morgan's Theorem

The basic **Laws of Boolean Algebra** that relate to the **Commutative Law** allowing a change in position for addition and multiplication, the **Associative Law** allowing the removal of brackets for addition and multiplication, as well as the **distributive Law** allowing the factoring of an expression, are the same as in ordinary algebra.

Each of the Boolean laws above are given with just a single or two variables, but the number of variables defined by a single law is not limited to this as there can be an infinite number of variables as inputs too the expression. These Boolean laws detailed above can be used to prove any given Boolean expression as well as for simplifying complicated digital circuits.

A brief description of the various **Laws of Boolean** are given below.

Description of the Laws and Theorems

Annulment Law - A term AND´ed with a "0" equals 0 or OR´ed with a "1" will equal 1.

1. A . 0 = 0,   A variable AND'ed with 0 is always equal to 0.
2. A + 1 = 1,   A variable OR'ed with 1 is always equal to 1.

Identity Law - A term OR´ed with a "0" or AND´ed with a "1" will always equal that term.

3. A + 0 = A,   A variable OR'ed with 0 is always equal to the variable.
4. A . 1 = A,   A variable AND'ed with 1 is always equal to the variable.

Indempotent Law - An input AND´ed with itself or OR´ed with itself is equal to that input.

5. $A + A = A$, A variable OR'ed with itself is always equal to the variable.
6. $A . A = A$, A variable AND'ed with itself is always equal to the variable.

Complement Law - A term AND´ed with its complement equals "0" and a term OR´ed with its complement equals "1".

7. $A . A = 0$, A variable AND'ed with its complement is always equal to 0.
8. $A + A = 1$, A variable OR'ed with its complement is always equal to 1.

Commutative Law - The order of application of two separate terms is not important.

9. $A . B = B . A$, The order in which two variables are AND'ed makes no difference.
10. $A + B = B + A$, The order in which two variables are OR'ed makes no difference.

Double Negation Law - A term that is inverted twice is equal to the original term.

11. $A = A$, A double complement of a variable is always equal to the variable.

de Morgan´s Theorem - There are two "de Morgan´s" rules or theorems,

(**1**) Two separate terms NOR´ed together is the same as the two terms inverted (Complement) and AND´ed for example, $A+B = A. B$.

(**2**) Two separate terms NAND´ed together is the same as the two terms inverted (Complement) and OR´ed for example, A.B = A +B.

Other algebraic laws not detailed above include:

Distributive Law - This law permits the multiplying or factoring out of an expression.

Absorptive Law - This law enables a reduction in a complicated expression to a simpler one by absorbing like terms.

Associative Law - This law allows the removal of brackets from an expression and regrouping of the variables.

Boolean Algebra Functions

Using the information above, simple 2-input AND, OR and NOT Gates can be represented by 16 possible functions as shown in the following table.

Function	Description	Expression
1.	NULL	0
2.	IDENTITY	1
3.	Input A	A
4.	Input B	B
5.	NOT A	A
6.	NOT B	B
7.	A AND B (AND)	A . B
8.	A AND NOT B	A . B
9.	NOT A AND B	A . B
10.	NOT A AND NOT B (NAND)	A . B
11.	A OR B (OR)	A + B
12.	A OR NOT B	A + B
13.	NOT A OR B	A + B
14.	NOT OR (NOR)	A + B
15.	Exclusive-OR	A.B + A.B
16.	Exclusive-NOR	A.B + A.B

Example No1

Using the above laws, simplify the following expression: (A + B)(A + C)

Q =	(A + B)(A + C)	
	AA + AC + AB + BC	- Distributive law
	A + AC + AB + BC	- Identity AND law (A.A = A)
	A(1 + C) + AB + BC	- Distributive law
	A.1 + AB + BC	- Identity OR law (1 + C = 1)
	A(1 + B) + BC	- Distributive law
	A.1 + BC	- Identity OR law (1 + B = 1)
Q =	A + BC	- Identity AND law (A.1 = A)

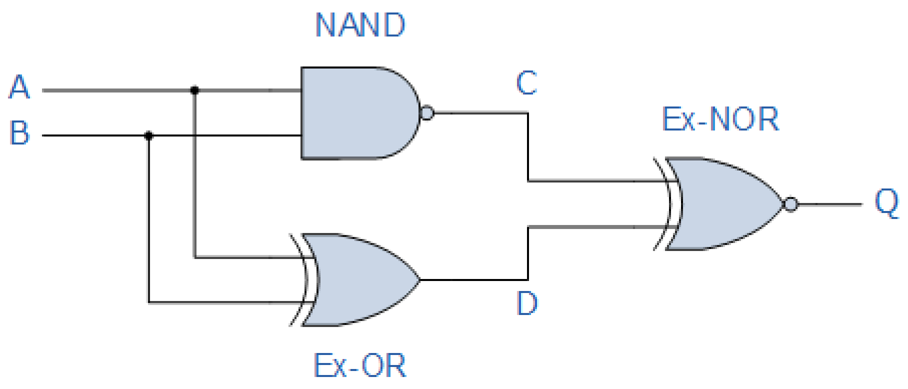
Then the expression: (A + B)(A + C) can be simplified to A + BC

Boolean Algebra Examples

Here are a few examples of how to use **Boolean Algebra** to simplify larger logic circuits.

Boolean Example No1

Construct a Truth Table for the logical functions at points C, D and Q in the following circuit and identify a single logic gate that can be used to replace the whole circuit.



First observations tell us that the circuit consists of a 2-input NAND gate, a 2-input EX-OR gate and finally a 2-input EX-NOR gate at the output. As there are only 2 inputs to the circuit labelled A and B, there can only be 4 possible combinations of the input (2^2) and these are: 0-0, 0-1, 1-0 and finally 1-1. Plotting the logical functions from each gate in tabular form will give us the following truth table for the whole of the logic circuit below.

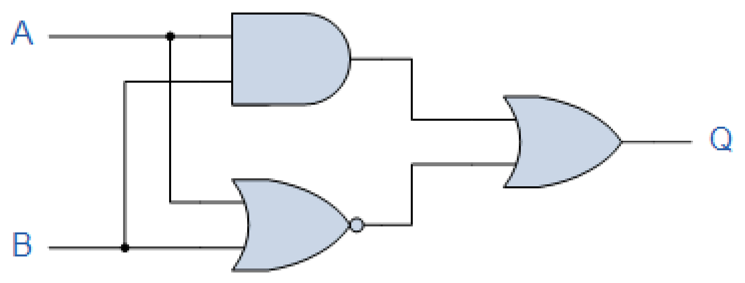
Inputs		Output at		
A	B	C	D	Q
0	0	1	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	0	1

From the truth table above, column C represents the output function generated by the NAND gate, while column D represents the output function from the Ex-OR gate. Both of these two output expressions then become the input condition for the Ex-NOR gate at the output.

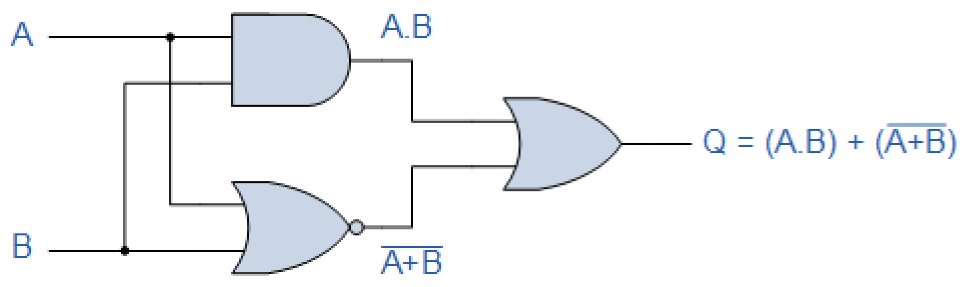
It can be seen from the truth table that an output at Q is present when any of the two inputs A or B are at logic 1. The only truth table that satisfies this condition is that of an OR Gate. Therefore, the whole of the above circuit can be replaced by just one single **2-input** OR Gate.

Boolean Example No2

Find the Boolean algebra expression for the following system.



The system consists of an AND Gate, a NOR Gate and finally an OR Gate. The expression for the AND gate is A.B, and the expression for the NOR gate is A+B. Both these expressions are also separate inputs to the OR gate which is defined as A+B. Thus the final output expression is given as:



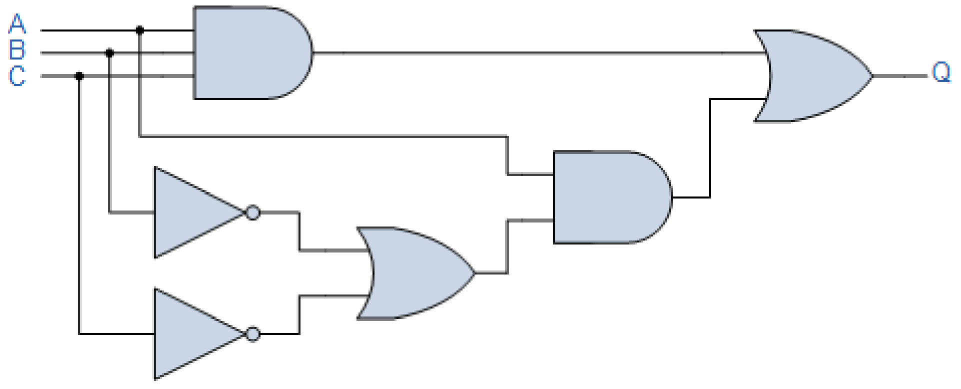
The output of the system is given as Q = (A.B) + (A+B), but the notation A+B is the same as the De Morgan´s notation A.B, Then substituting A.B into the output expression gives us a final output notation of Q = (A.B)+(A.B), which is the Boolean notation for an Exclusive-NOR Gate as seen in the previous section.

Inputs		Intermediates		Output
B	A	A.B	A + B	Q
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

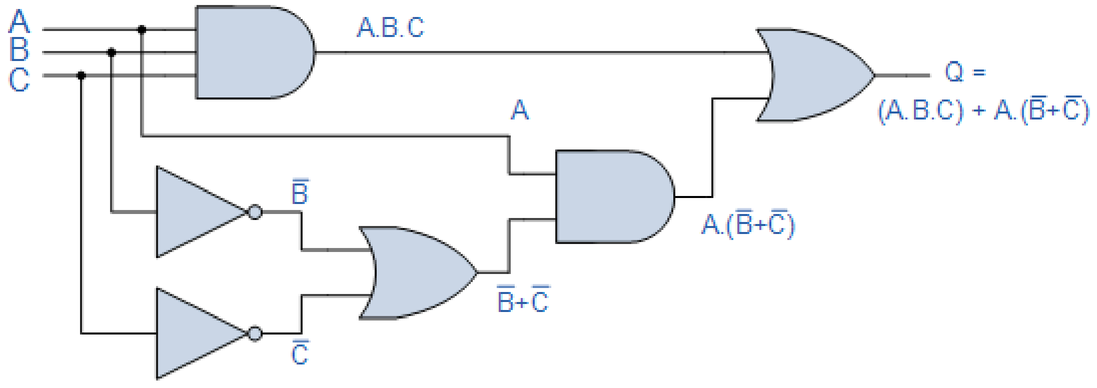
Then, the whole circuit above can be replaced by just one single Exclusive-NOR Gate and indeed an Exclusive-NOR Gate is made up of these individual gates.

Boolean Example No3

Find the Boolean algebra expression for the following system.



This system may look more complicated than the others to analyse but it again, it just consists of simple AND, OR and NOT gates connected together. As with the previous Boolean example, we can write the Boolean notation for each logic function inturn to give us a final expression for the output at Q.



The output from the 3-input AND gate is only a "1" when **ALL** the inputs are at logic level "1" (A.B.C). The output from the lower OR gate is only a "1" when one or both inputs B or C are at logic level "0". The output from the 2-input AND gate is a "1" when input A is a "1" and inputs B or C are at "0".

Then the output at Q is only a "1" when inputs A.B.C equal "1" or A is equal to "1" and both inputs B or C equal "0", A.(B+C). By using "**de Morgan's theorem**" inputs B and input C cancel out as to produce an output at Q they can be either at logic "1" or at logic "0". Then this just leaves input A as the only input needed to give an output at Q as shown in the table below.

Inputs			Intermediates					Output
C	B	A	A.B.C	B	C	B+C	A.(B+C)	Q
0	0	0	0	1	1	1	0	0
0	0	1	0	1	1	1	1	1
0	1	0	0	0	1	1	0	0
0	1	1	0	0	1	1	1	1
1	0	0	0	1	0	1	0	0
1	0	1	0	1	0	1	1	1
1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	1

Then we can see that the entire logic circuit above can be replaced by just one single input labelled A thereby reducing a circuit of six individual logic gates to just one single piece of wire, (or *Buffer*). This type of circuit analysis using **Boolean Algebra** can quickly identify any unnecessary logic gates within a digital logic design thereby reducing the number of gates required, the power consumption of the circuit and of course the cost.

- **Simplify: C + BC:**

Expression	Rule(s) Used
C + BC	Original Expression
C + (B + C)	DeMorgan's Law.
(C + C) + B	Commutative, Associative Laws.
T + B	Complement Law.
T	Identity Law.

•**Simplify: AB(A + B)(B + B):**

Expression	Rule(s) Used
AB(A + B)(B + B)	Original Expression
AB(A + B)	Complement law, Identity law.

(A + B)(A + B)	DeMorgan's Law
A + BB	Distributive law. This step uses the fact that or distributes over and. It can look a bit strange since addition does not distribute over multiplication.
A	Complement, Identity.

- **Simplify: (A + C)(AD + AD) + AC + C:**

Expression	Rule(s) Used
(A + C)(AD + AD) + AC + C	Original Expression
(A + C)A(D + D) + AC + C	Distributive.
(A + C)A + AC + C	Complement, Identity.
A((A + C) + C) + C	Commutative, Distributive.
A(A + C) + C	Associative, Idempotent.
AA + AC + C	Distributive.
A + (A + *T*)C	Idempotent, Identity, Distributive.
A + C	Identity, twice.

You can also use distribution of or over and starting from A(A+C)+C to reach the same result by another route.

- **Simplify: A(A + B) + (B + AA)(A + B):**

Expression	Rule(s) Used
A(A + B) + (B + AA)(A + B)	Original Expression
AA + AB + (B + A)A + (B + A)B	Idempotent (AA to A), then Distributive, used twice.
AB + (B + A)A + (B + A)B	Complement, then Identity. (Strictly speaking, we also used the Commutative Law for each of these applications.)
AB + BA + AA + BB + AB	Distributive, two places.
AB + BA + A + AB	Idempotent (for the A's), then Complement and Identity to remove BB.
AB + AB + A*T* + AB	Commutative, Identity; setting up for the next step.
AB + A(B + *T* + B)	Distributive.
AB + A	Identity, twice (depending how you count it).
A + AB	Commutative.
(A + A)(A + B)	Distributive.
A + B	Complement, Identity.

K-maps (Karnaugh Maps)

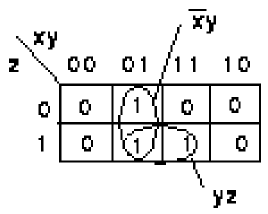
- **A Tool to generate minimal size circuits**
- **Graphical means of performing equivalent of algebraic implications**
- **Places candidate terms for simplification together**

Example: Simplify the function
f(xyz)=Sum(2,3,7).

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$$
\begin{aligned}
f &= \bar{x}y\bar{z} + \bar{x}yz + xyz \\
&= \bar{x}y\bar{z} + \bar{x}yz + \bar{x}yz + xyz \\
&= \bar{x}y + yz
\end{aligned}
$$

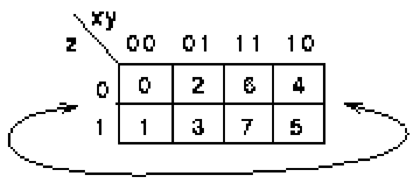
K-map:



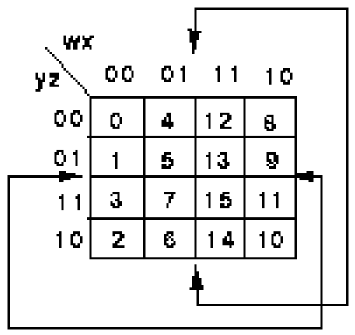
Drawing K-maps

- **Terms which differ in only one variable are placed adjacent to each other.**
- **Edges connected: toroidal topology.**

3-Input K-map



4-Input K-map

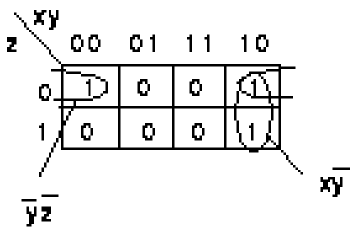


K-maps for Sum-of-Product Design

- **A minimal sum-of-product design can be created as follows:**
- **Create the K-map, entering a 1 or 0 in each square according to the desired logic function**
- **Beginning with large rectangles and going to small rectangles, add rectangles until all minterms are 'covered' (all '1's are included).**
- **Generate the algebraic terms corresponding to the rectangles**

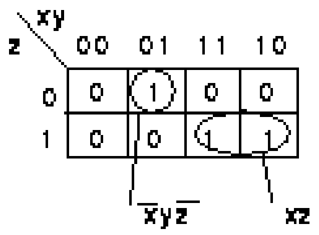
Examples:

f(xyz) = Sum(0,4,5)



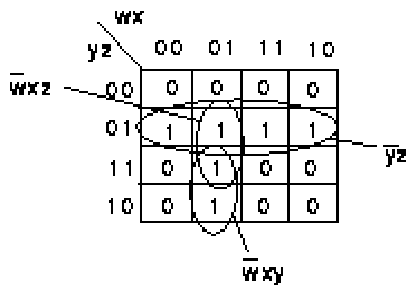
$f = \bar{y}\bar{z} + x\bar{y}$

f(xyz) = Sum(2,5,7)



$f = \bar{x}y\bar{z} + xz$

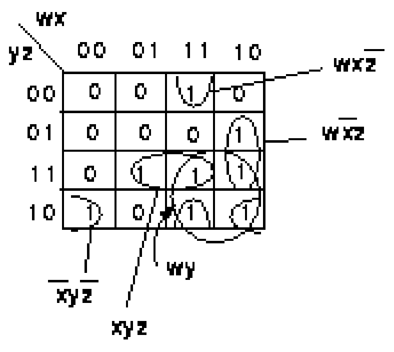
f(wxyz) = Sum(1,5,6,7,9,13)



$f = \bar{w}xy + \bar{y}z + \bar{w}xz$

optional term

f(wxyz) = Sum(2,7,9,10,11,12,14,15)



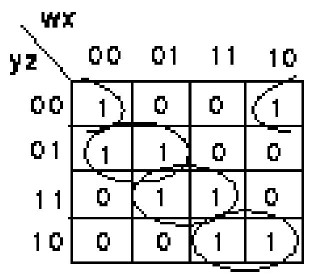
$f = w\bar{x}z + wx\bar{z} + \bar{x}y\bar{z} + xyz + wy$

optional term

f(wxyz) = Sum(0,1,5,7,8,10,14,15)

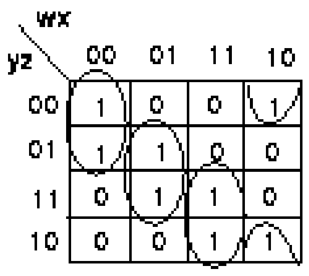
yz \ wx	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	0	1	1	0
10	0	0	1	1

Solution 1



$f = \bar{x}\bar{y}\bar{z} + \bar{w}\bar{y}z + xyz + wy\bar{z}$

Solution 2

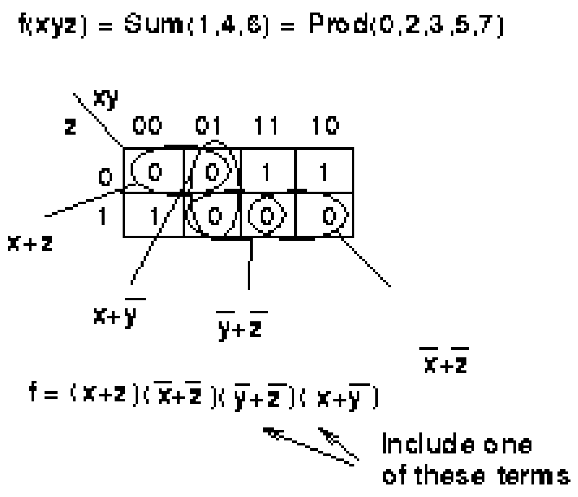


$f = \bar{w}\bar{x}\bar{y} + \bar{w}xz + wxy + w\bar{x}\bar{z}$

K-maps for Product-of-Sum Design

Product-of-sums design uses the same principles, but applied to the zeros of the function.

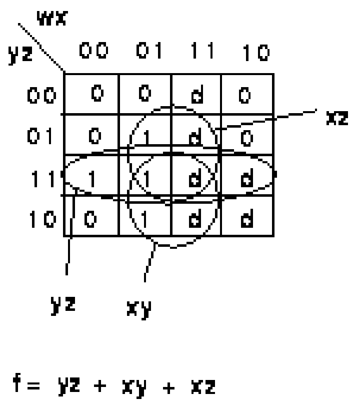
Example:


f(xyz) = Sum(1,4,6) = Prod(0,2,3,5,7)
xy
z
00 01 11 10
0
1
x+z
x+y̅
y̅+z̅
x̅+z̅
f = (x+z)(x̅+z̅)(y̅+z̅)(x+y̅)
Include one of these terms

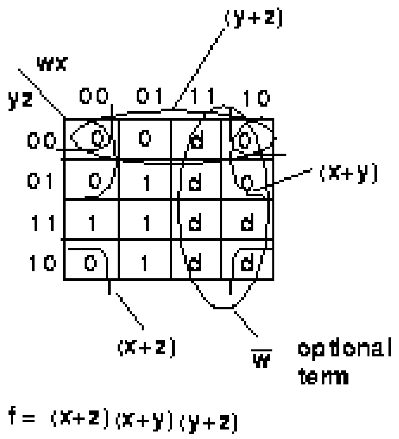
Designing with Don't-Care Values

In some situations, we don't care about the value of a logic function. For example, if we use wxyz to represent a number from 0 to 9, we need not worry about the function value produced for wxyz = 10...15. For these situations, the function can be assigned an output in order to make the resulting circuit as simple as possible.
Suppose we wish to implement the function
f(wxyz)=Sum(3,5,6,7)
and we have the don't-care condition of
d=Sum(10,11,12,13,14,15).

The sum-of-products implementation:



The product-of-sums implementation:



Logic Families

The digital IC technology has advanced considerably and rapidly over the years.Starting from small scale integration (SSI) with less than 12 gates per chip ,advancing to medium scale integration (MSI)with 12 to 100 gates per chip and then to large scale integration (LSI) with up to 10,000 gates per chip and on to Very large scale integration (VLSI) with up to 100,000 gates per chip,the digital IC technology has come a long way.Recently it has advanced to ultra large scale integration(ULSI) with more than 100,000 gates per chip. A group of compatible ICs with similar logic levels and supply voltages fabricated using a specific circuitary is referred as a logic family.

Based on the technology used ,we can have seven basic logic families.They are:

1.RTL- Resistor transistor logic family

2.DTL- Diode transistor logic family

3.IIL- Integrated injection logic family

4.TTL- Transistor transistor logic family

5.ECL- Emitter coupled logic family

6.MOS -Metal oxide semiconductor logic family

7.CMOS- Complementary metal oxide semiconductor logic family.

Among these families we consider here RTL,DTL,ECL,TTL and CMOS logic families only.

1.4.1 Characteristics: The basic characteristics of a logic family are :

(i). Speed of operation,

(ii) Fan-in and Fan-out,

(iii) Power dissipation,

(iv) Propagation Delay,

(v) Operating temperature range,

(vi) Voltage and Current parameters and

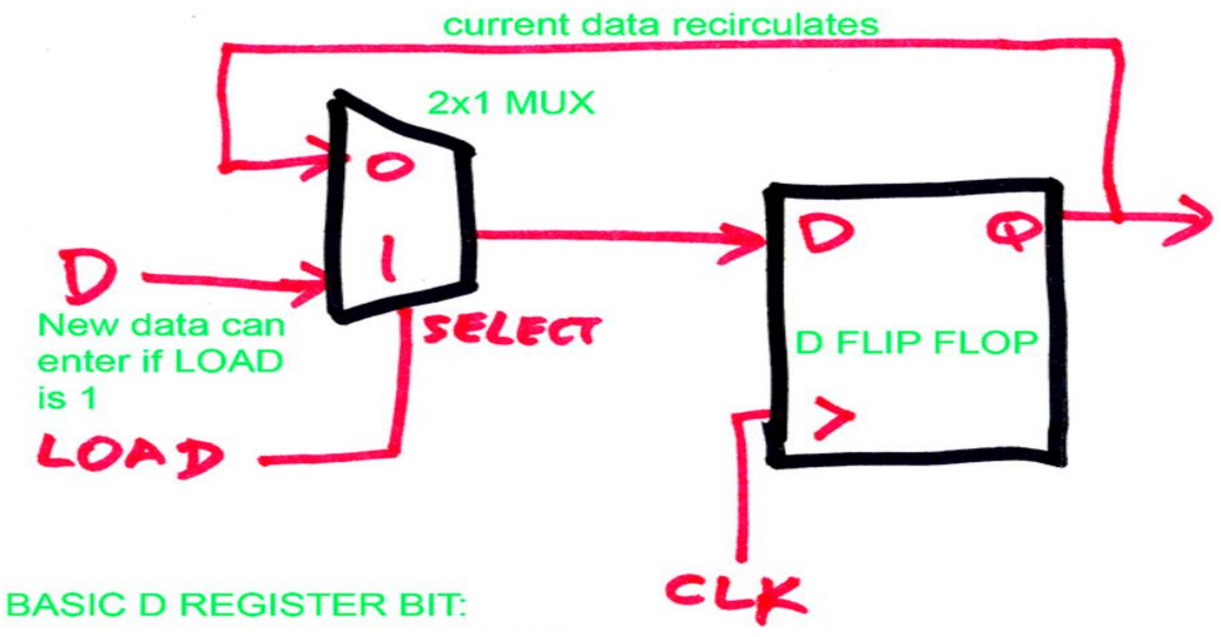
(vii) Noise margin(Noise immunity)

Register transfer logic

Register transfer and data flow

In register transfer logic , we look at the use of simple registers to control the flow of data in a computer system. One can, in fact, divide a computer CPU into two sections: the dataflow section, consisting of registers, busses and arithmetic/logic components, and the control section, containing steering logic, counters for state sequencing, etc. The control section is sometimes called the "glue" logic, as it connects to all the control points (CLEARs, LOADs, etc) of all the registers, octopus-like.

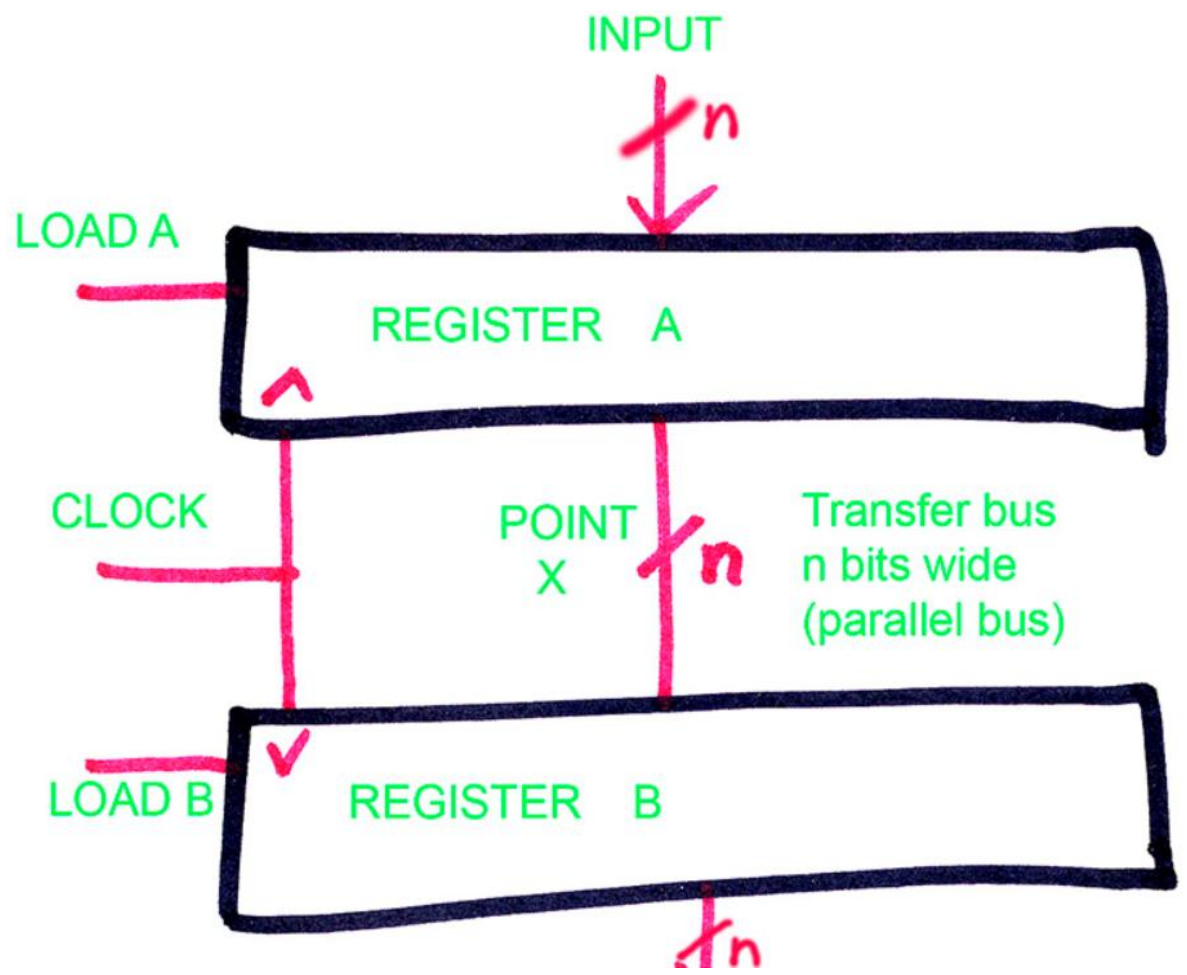
Just to review, the basic register bit is shown below:



The clock runs continuously. If LOAD is 0, the flip flops just keeps re-storing its old data and so doesn't change. If LOAD is 1, then new data will be stored. This diagram is a one-bit slice of a data register. The number of bits in the register is called the width of the register. This is commonly 8, 16, 32, or 64 bits. The registers and other arithmetic logic such as adders are interconnected with parallel busses. In parallel data transfer, the bits all run along similar wires simultaneously, and are stored in the register at the end of the bus at the same time.

(Q)This is really answered on a previous page. Why do we need this rather complex register bit design? Why can't we simply use a latch, saving about 70% of the transistors, and store the data by pulsing the ENABLE on the latch with the LOAD pulse? [As noted previously, this CAN be done, but....]

The diagram below shows the basic register dataflow connection:



Here, data will be loaded from the external data input bus into register A if LOAD A is high when the clock edge arrives. Each register, and their busses, are n bits wide. In the x86 (IA32), for instance, the registers are 32 bits wide, so n is 32 in that case. The 32 bit register consists of the D register itself (see previous diagram) copied 32 times. Data moves in parallel-all 32 bits are transferred together, one into each register bit.

If LOAD B is high when the clock edge arrives, whatever is at POINT X will be transferred into register B. Normally, when the data is loaded into a register, it is visible at the output pins of the register after a short propagation delay through the latches in the register's flip flops. [It is possible to create "tri-state" registers in which an additional "tri-state buffer" is added to the output pins. This tri-state buffer is like a switch or CMOS transmission gate. This is used to allow multiple registers to be connected to the same output bus without electrical contention and without the need for multiplexers to select the desired register for connection to the bus]

If LOAD A and LOAD B are both 1 when the clock edge arrives, external data goes into register A and the previous data in register A (visible at POINT X) will be loaded into register B. This situation is referred to as a pipeline.

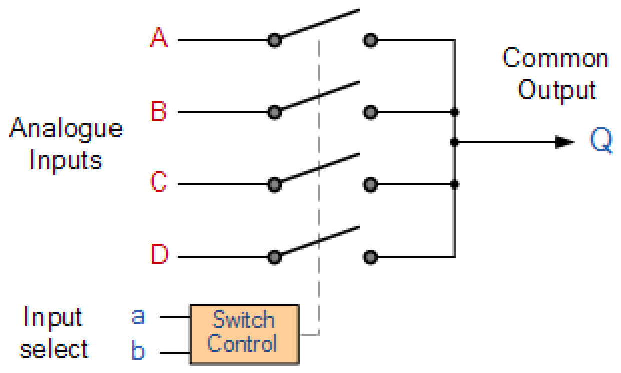
The Multiplexer

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

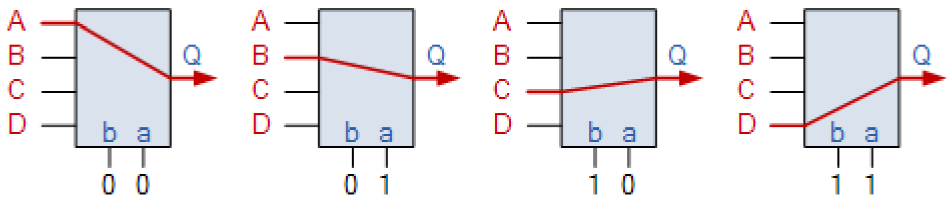
1	1	D

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".

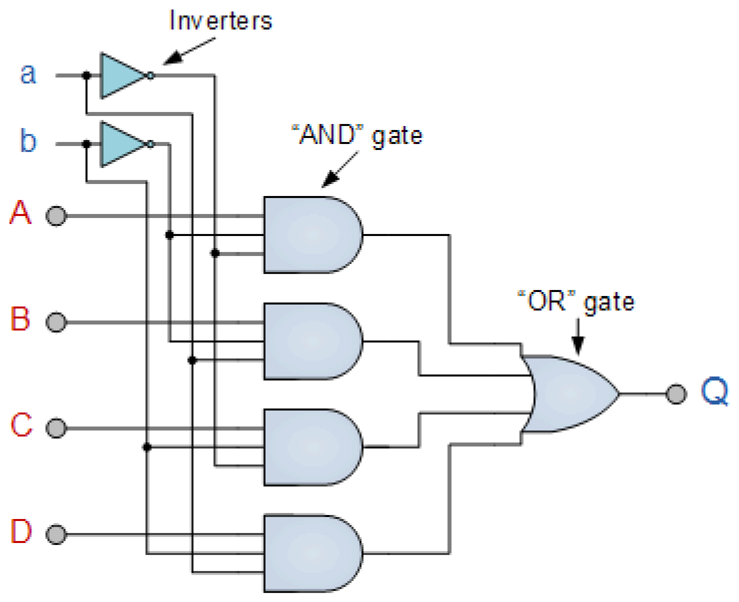
Then we can show the selection of the data through the multiplexer as a function of the data select bits as shown.



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.

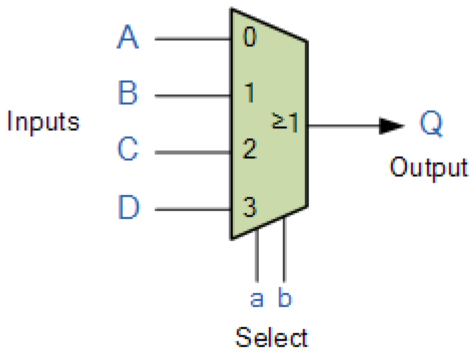
Then the implementation of this Boolean expression above using individual logic gates would require the use of seven individual gates consisting of AND, OR and NOT gates as shown.

4 Channel Multiplexer using Logic Gates



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

Multiplexer Symbol

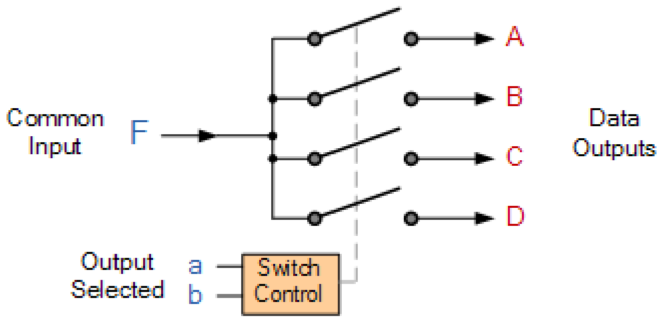


Multiplexers are not limited to just switching a number of different input lines or channels to one common single output. There are also types that can switch their inputs to multiple outputs and have arrangements or 4 to 2, 8 to 3 or even 16 to 4 etc configurations and an example of a simple Dual channel 4 input multiplexer (4 to 2) is given below:

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

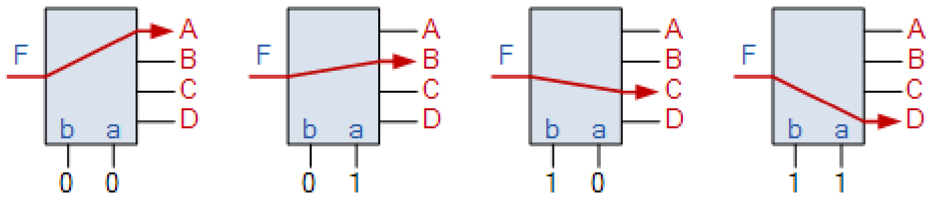


Addressing		Input Selected
b	a	
0	0	A
0	1	B
1	0	C
1	1	D

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" as shown.

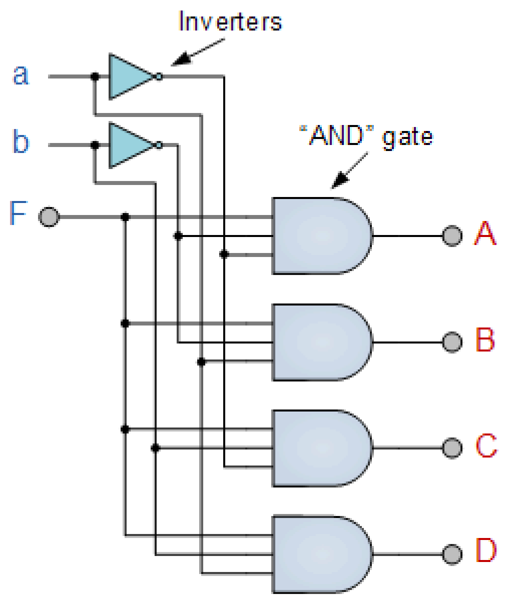


As with the previous *multiplexer circuit*, adding more address line inputs it is possible to switch more outputs giving a 1-to-2^n data line outputs.

Some standard demultiplexer IC´s also have an additional "enable output" 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".

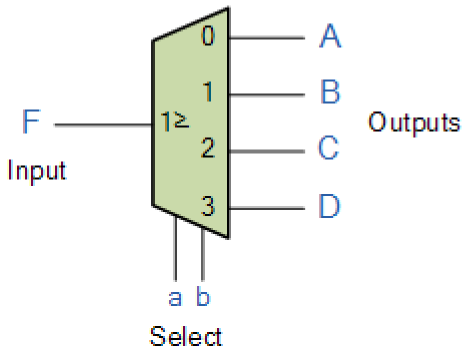
The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

4 Channel Demultiplexer using Logic Gates



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

The Demultiplexer Symbol

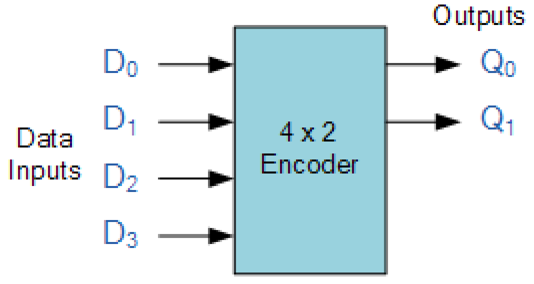


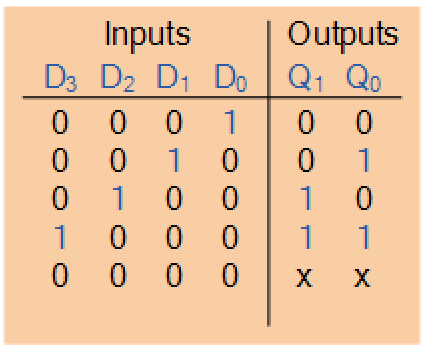
The Digital Encoder

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

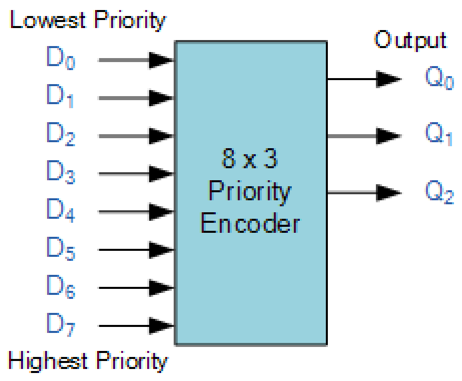
One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level "1". For example, if we make inputs D_1 and D_2 HIGH at logic "1" both at the same time, the resulting output is neither at "01" or at "10" 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.

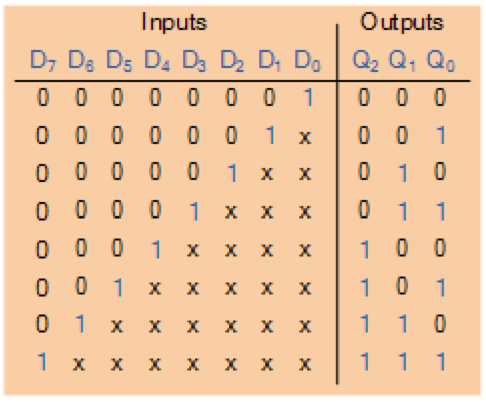
One simple way to overcome this problem is to "Prioritise" the level of each input pin and if there was more than one input at logic level "1" the actual output code would only correspond to the input with the highest designated priority. Then this type of digital encoder is known commonly as a **Priority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

Priority Encoder

The **Priority Encoder** solves the problems mentioned above by allocating a priority level to each input. The *priority encoders* output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. The priority encoder comes in many different 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 truth table for a 8-to-3 bit priority encoder is given as:

Digital Inputs								Binary Output		
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

From this truth table, the Boolean expression for the encoder above with inputs D_0 to D_7 and outputs Q_0, Q_1, Q_2 is given as:

Output Q_0

$$Q_0 = \sum\left(1, 3, 5, 7\right)$$
$$Q_0 = \sum\left(\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2D_1 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_3 + \overline{D}_7\overline{D}_6D_5 + D_7\right)$$
$$Q_0 = \sum\left(\overline{D}_6\overline{D}_4\overline{D}_2D_1 + \overline{D}_6\overline{D}_4D_3 + \overline{D}_6D_5 + D_7\right)$$
$$Q_0 = \sum\left(\overline{D}_6\left(\overline{D}_4\overline{D}_2D_1 + \overline{D}_4D_3 + D_5\right) + D_7\right)$$

Output Q_1

$$Q_1 = \sum\left(2, 3, 6, 7\right)$$
$$Q_1 = \sum\left(\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3D_2 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_3 + \overline{D}_7D_6 + D_7\right)$$
$$Q_1 = \sum\left(\overline{D}_5\overline{D}_4D_2 + \overline{D}_5\overline{D}_4D_3 + D_6 + D_7\right)$$
$$Q_1 = \sum\left(\overline{D}_5\overline{D}_4\left(D_2 + D_3\right) + D_6 + D_7\right)$$

Output Q_2

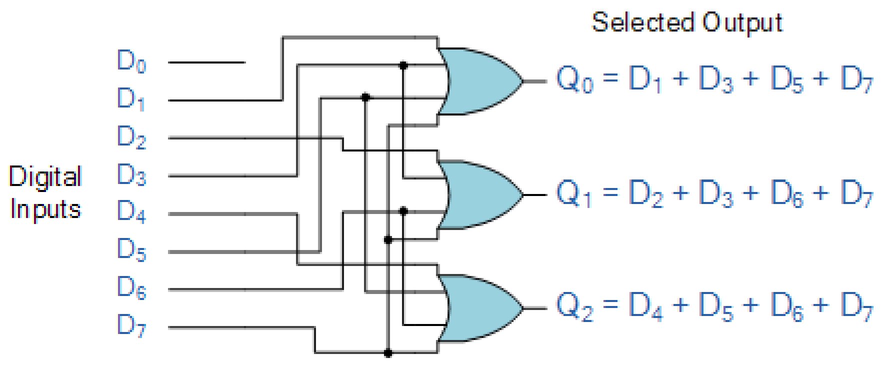
$$Q_2 = \sum\left(4, 5, 6, 7\right)$$
$$Q_2 = \sum\left(\overline{D}_7\overline{D}_6\overline{D}_5D_4 + \overline{D}_7\overline{D}_6D_5 + \overline{D}_7D_6 + D_7\right)$$
$$Q_2 = \sum\left(D_4 + D_5 + D_6 + D_7\right)$$

Then the final Boolean expression for the priority encoder including the zero inputs is defined as:

$$Q_0 = \sum\left(\overline{D}_6\left(\overline{D}_4\overline{D}_2D_1 + \overline{D}_4D_3 + D_5\right) + D_7\right)$$
$$Q_1 = \sum\left(\overline{D}_5\overline{D}_4\left(D_2 + D_3\right) + D_6 + D_7\right)$$
$$Q_2 = \sum\left(D_4 + D_5 + D_6 + D_7\right)$$

In practice these zero inputs would be ignored allowing the implementation of the final Boolean expression for the outputs of the 8-to-3 **priority encoder** above to be constructed using individual OR gates as follows.

Digital Encoder using Logic Gates


Selected Output
Digital Inputs
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
$Q_0 = D_1 + D_3 + D_5 + D_7$
$Q_1 = D_2 + D_3 + D_6 + D_7$
$Q_2 = D_4 + D_5 + D_6 + D_7$

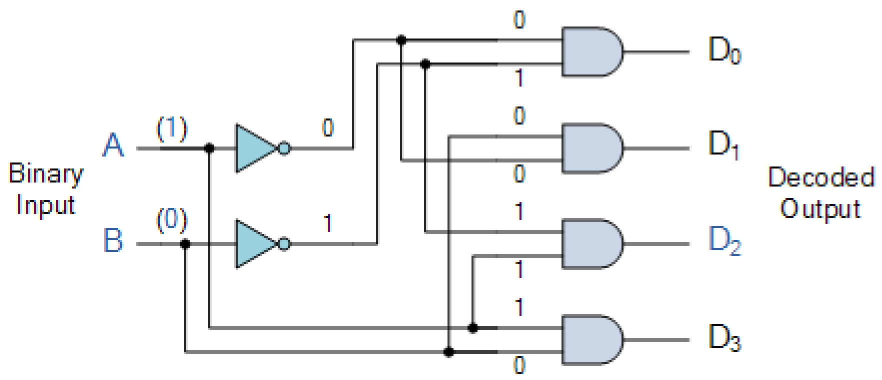
Binary Decoder

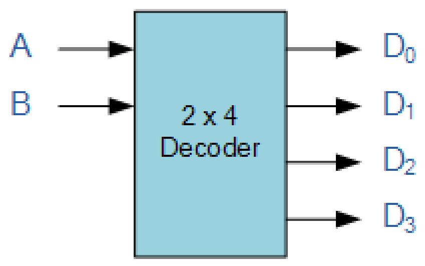
A **Decoder** is the exact opposite to that of an "Encoder" we looked at in the last tutorial. It is basically, a combinational type logic circuit that converts the binary code data at its input into one of a number of different output lines, one at a time producing an equivalent decimal code at its 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.

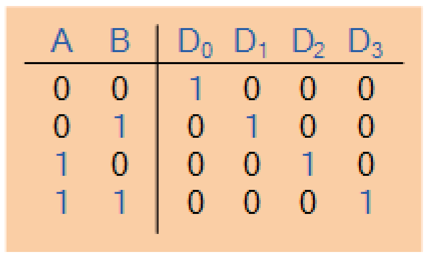
Therefore, if a binary decoder receives n inputs (usually grouped as a binary or Boolean number) it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. A decoders output code normally has more bits than its input code and practical "binary decoder" circuits 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.

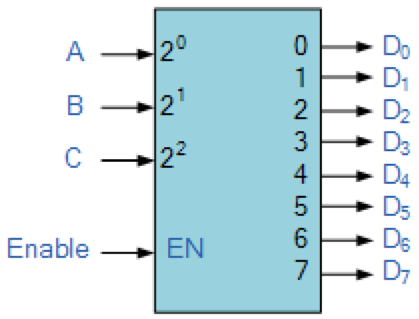






A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

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.

Then, we can say that a binary decoder is a demultiplexer 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.

The Binary Adder

Another common and very useful combinational logic circuit which can be constructed using just a few basic logic gates and adds together binary numbers is the **Binary Adder** circuit. The Binary Adder is made up from standard AND and Ex-OR gates and allow us to "add" together single bit binary numbers, a and b to produce two outputs called the SUM of the addition and a CARRY called the Carry-out, (C_{out}) bit. One of the main uses for the **Binary Adder** is in arithmetic and counting circuits.

Consider the addition of two denary (base 10) numbers below.

123	A	(Augend)
+ 789	B	(Addend)
912	SUM	

Each column is added together starting from the right hand side and each digit has a weighted value depending upon its position in the columns. As each column is added together a carry is generated if the result is greater or equal to ten, the base number. This carry is then added to the result of the addition of the next column to the left and so on, simple school math's addition. The adding of binary numbers is basically the same as that of adding decimal numbers but this time a carry is only generated when the result in any column is greater or equal to "2", the base number of binary.

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.

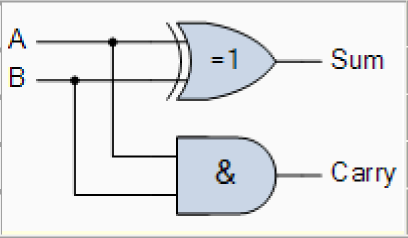
An Ex-OR gate only produces an output "1" when either input is at logic "1", but not both. However, all microprocessors and electronic calculators require the carry bit to correctly calculate the equations so we need to rewrite them to include 2 bits of output data as shown below.

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 = $A \oplus B$ Carry = $A \cdot B$				

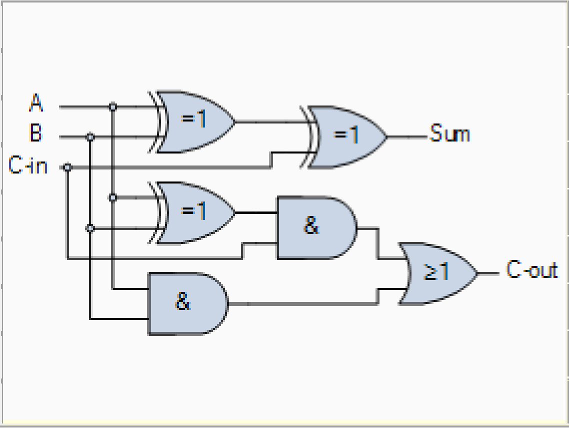
From the truth table we can see that the SUM (S) output is the result of the Ex-OR gate and the Carry-out (Cout) is the result of the AND gate. One major disadvantage of the Half Adder circuit when used as a binary adder, is that there is no provision for a "Carry-in" from the previous circuit when adding 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
 A B C-in =1 =1 Sum =1 & ≥1 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

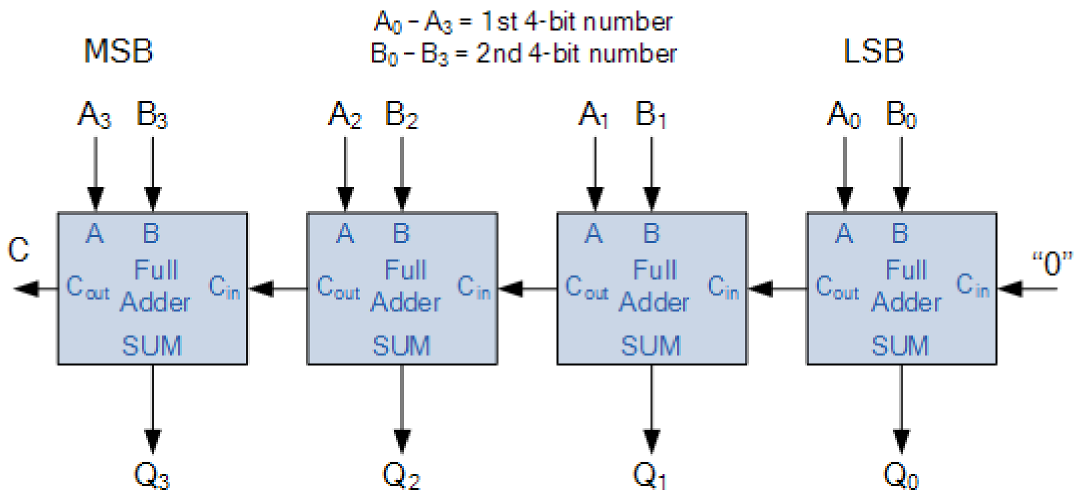
The 1-bit **Full Adder** circuit above is basically two half adders connected together and consists of three Ex-OR gates, two AND gates and an OR gate, six logic gates in total. The truth table for the full adder includes an additional column to take into account the Carry-in input as well as the summed 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

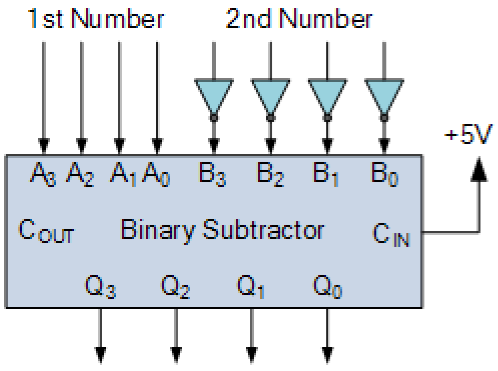

$A_0 - A_3$ = 1st 4-bit number
$B_0 - B_3$ = 2nd 4-bit number
MSB
LSB
A_3 B_3
A_2 B_2
A_1 B_1
A_0 B_0
A B
C_{out} Full Adder C_{in}
SUM
C
"0"
Q_3
Q_2
Q_1
Q_0

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 Digital Comparator

Another common and very useful combinational logic circuit is that of the **Digital Comparator** circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals present at their input terminals and produce an output depending upon the condition of those inputs.

For example, along with being able to add and subtract binary numbers we need to be able to compare them and determine whether the value of input A is greater than, smaller than or equal to the value at input B etc. The digital comparator accomplishes this using several logic gates that operate on the principles of Boolean algebra. There are two main types of **Digital Comparator** available and these are.

- 1. Identity Comparator - an *Identity Comparator* is a digital comparator that has only one output terminal for when A = B either "HIGH" A = B = 1 or "LOW" A = B = 0
-
- 2. Magnitude Comparator - a *Magnitude Comparator* is a type of digital comparator that has three output terminals, one each for equality, A = B greater than, A > B and less than A < B

The purpose of a **Digital Comparator** is to compare a set of variables or unknown numbers, for example A (A1, A2, A3, ... An, etc) against that of a constant or unknown value such as B (B1, B2, B3, ... Bn, etc) and produce an output condition or flag depending upon the result of the comparison.

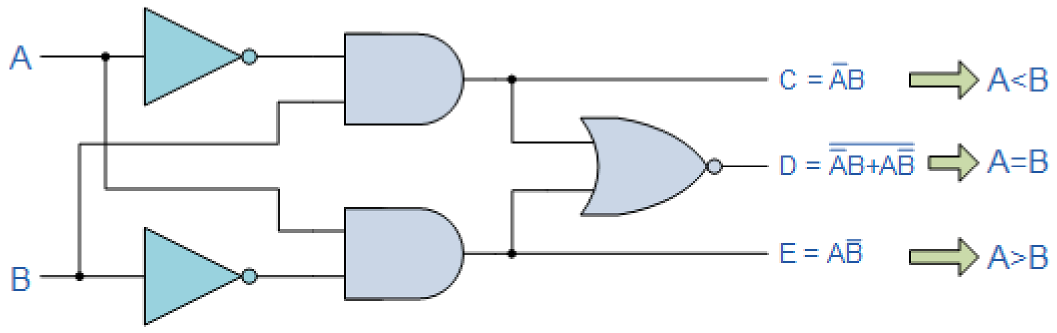
For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

$$A > B, \quad A = B, \quad A < B$$

Which means: A is greater than B, A is equal to B, and A is less than B

This is useful if we want to compare two variables and want to produce an output when any of the above three conditions are achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

1-bit Digital Comparator



Then the operation of a 1-bit digital comparator is given in the following Truth Table.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

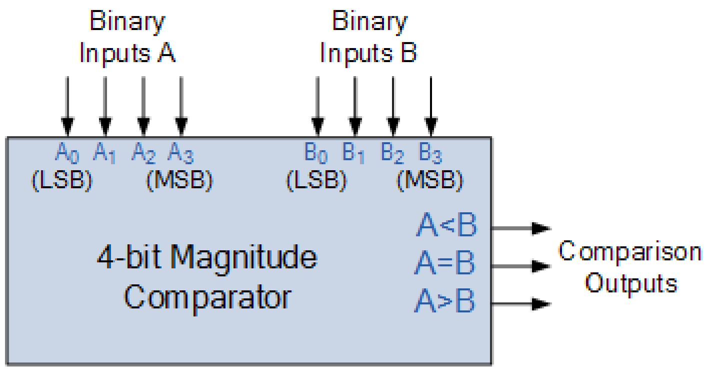
You may notice two distinct features about the comparator from the above truth table. Firstly, the circuit does not distinguish between either two "0" or two "1"'s as an output A = B is produced when they are both equal, either A = B = "0" or A = B = "1". Secondly, the output condition for A = B resembles that of a commonly available logic gate, the Exclusive-NOR or Ex-NOR function (equivalence) on each of the n-bits giving: $Q = A \oplus B$

Digital comparators actually use Exclusive-NOR gates within their design for comparing their respective pairs of bits. When we are comparing two binary or BCD values or variables against each other, we are comparing the "magnitude" of these values, a logic "0" against a logic "1" which is where the term **Magnitude Comparator** comes from.

As well as comparing individual bits, we can design larger bit comparators by cascading together n of these and produce a n-bit comparator just as we did for the n-bit adder in the previous tutorial. Multi-bit comparators can be constructed to compare whole binary or BCD words to produce an output if one word is larger, equal to or less than the other.

A very good example of this is the 4-bit **Magnitude Comparator**. Here, two 4-bit words ("nibbles") are compared to each other to produce the relevant output with one word connected to inputs A and the other to be compared against connected to input B as shown below.

4-bit Magnitude Comparator



A SIMPLE PROCESSOR DESIGN

The primary function of a CPU is to execute programs expressed in the processor's own machine language. During their execution programs and their accompanying data are stored wholly or in part in a main memory M which lies outside the CPU. To actually execute the program the CPU must perform the following actions:

= Determine the address in M of the first (or next as is appropriate) machine language instruction I.

= Fetch I from M by performing one or more memory read operations.

= Decode I to determine the operation(s) to be performed.

= If necessary fetch any operands required by I that are stored in M; again this will require one or more memory read operations.

= Perform the operations specified by I.

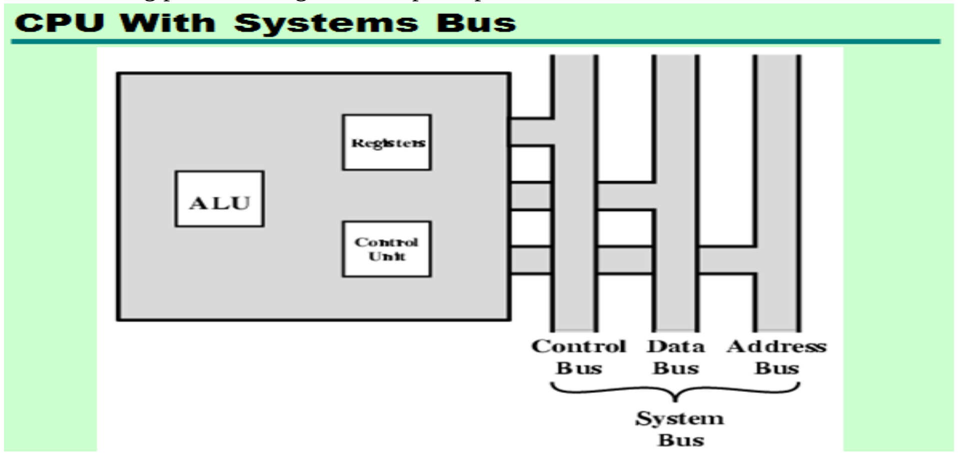
= If required by I, store any results in M. This will require one or more memory write operations.

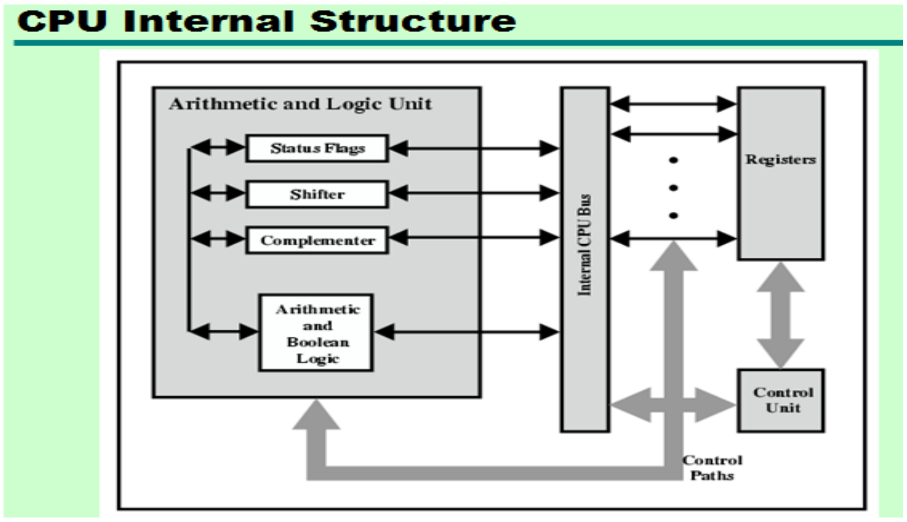
These steps comprise what is known as the *instruction cycle* or *fetch-execute cycle*. Steps 4, 5, and 6 together constitute the execute phase of the instruction cycle.

During normal execution of a program the CPU repeatedly goes through the instruction cycle. The circuitry within the CPU to implement this process consists of:

5. An appropriate sized alu.
6. A variety of registers for the temporary storage of addresses, instructions and data.
7. Control circuitry to properly sequence the transfers of data among the CPU's alu and internal registers that are needed to implement the steps of the instruction cycle for each machine language instruction.

The organization used to connect the registers and CPU is often referred to as the *data path* of the CPU. On the following page we show the data paths for some simple, hypothetical, CPU's. In section 2 we shall describe the data path for our own hypothetical CPU (which we shall call the *Relatively Simple CPU*, or simply *RSCPU*) and use this CPU as a vehicle for introducing processor organization principles.





Data Path for a Hypothetical CPU

Shown below is a variation of the diagram on page 246 of your textbook that represents the data path for the *Relatively Simple CPU* that we shall design for this course. For simplicity here, we have omitted the control signals for transferring data between the bus and the registers and for controlling the alu. We will describe these in detail later, however.

The data path also includes the following components:

c) *a 16-bit address register* (AR) to address words in main memory. The outputs of the address register connect it to the address lines of the system bus connecting the CPU and memory. We also assume that AR has a control line that, when activated (high) increments its current value by 1. We denote this operation by AR++.

d) a 16-bit *program counter* (PC) that contains the address of the **next** instruction to be executed (not the current instruction), or the address of the next required operand of the current instruction. We also assume that PC has a control line that, when activated (high) increments the current value of the PC by 1. We denote this operation by PC++.

e) an 8-bit *data register* (DR) that serves as a data interface between the CPU and memory. It has separate data lines to connect it to those of memory and separate control lines for interacting with memory. Note, in our data path the output of DR is connected to the low order lines of the bus and is also connected directly to the input lines of the IR and TR registers described below. Finally, we include circuitry (not shown, but essentially tri-state buffers) to allow the output of DR to also be placed on the high-order lines of the bus.

c) *An* 8*-bit ALU* capable of performing various operations as shown in the table below. Here we assume that X and Y are the operand inputs to the ALU and W represents the output. The operation of the ALU will be governed the following seven control signals (ALU1,…,ALU7) according to the following table (here D -= "don't care"):

2 an 8-bit accumulator register (AC) that receives the results of any arithmetic or logical operation and provides the X operand for appropriate binary arithmetic or logical operations of the ALU. It is also the source (destination) of any programmer-initiated data transfers to (from) memory. Note that while the output of AC is routed to the data bus and the X input of the alu, it only receives its input from the alu.

3 an 8-bit general-purpose register (R) that supplies the Y operand for appropriate binary arithmetic or logical operations that the ALU performs. It can also be used by a programmer to store data. It is capable of bidirectional data transfers with the (low order lines of the) data bus

4 an *8-bit instruction register* (IR) which contains a copy of the op-code of the current instruction. Note, IR is not connected to the data bus. It receives its input from the output lines of DR, and as we shall see later, its output is directed elsewhere.

5 an 8-bit temporary register (TR) which temporarily stores data during instruction execution. Note, the output lines of TR go to the data path's bus, but TR receives its input from the output lines of DR.

6 a 1-bit flag register Z that is set to 1 if the last arithmetic or logical operation produced a result equal to 0.

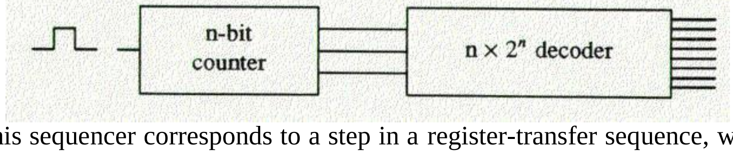
CONTROL LOGIC DESIGN

Control Units

The implementation of the instruction cycle of a CPU reduces to a sequence of register transfers, which are in turn governed by the activation of the proper control signals. In order for any phase of the instruction cycle to be properly implemented however, these control signals must be activated in the correct sequence. It is the responsibility of the *control unit* of the CPU to see that the proper control signals are generated in the correct sequence.

A. ***Hardwired Control Units***

Here the correct control signals for each register transfer step of each phase of the instruction cycle are generated at the proper time by a clock-driven counter operating with a decoder as shown below.



Each output line of this sequencer corresponds to a step in a register-transfer sequence, where 2n is equal to or greater than the maximum number of steps that will ever be required. The required control signals will then be generated by additional combinational circuitry which uses the following input signals:

a. the output of the step sequencer;

b. the content of the instruction register;

c. the content of any condition codes or status registers which may be incorporated into the cpu's design (in our cpu these would be signals which tell us that the result of an alu operation was zero).

B. ***Microprogrammed Control Units***

Here control signals are arranged into a unit known as a *control vector*, where each control vector specifies one or more register transfers. Control vectors for a desired sequence of register transfers are them stored in a special high-speed memory known as the *control memory* and a simpler, hardwired control unit known as the *microcontrol unit* sees that the operations specified by these control vectors are performed in the proper sequence. Each phase of the instruction cycle can be reduced to sequences of register transfers encoded in control vectors (or *microinstructions*) and executed by the microcontrol unit in a way that parallels the way the control unit is supposed to execute machine language instructions.

For the rest of this section we shall give examples to indicate how hardwired control signals can be generated for our hypothetical CPU, and leave it to the reader to pursue the topic further. In the next section shall explore in more detail the structure of a microprogrammed control unit for our CPU.

Hardwired Control Units

In our examples in the previous section we derived the sequences of control signals needed to implement the fetch phase of the instruction cycle as well as each machine language instruction. We note that at most 8 steps are needed to carry out any instruction (3 for fetching it and at most 5 steps to implement it). If we let T0 ,...,T7 denote the (first) eight output lines of the counter-decoder circuit given on the previous page, representing lines which are active (i.e. = 1) on the successive clock ticks, then we can specify the values that the control signals must have to correctly implement the instruction cycle for our machine language instruction set by the control equations such as the following:

```
acload =
T3(MOVR+ADD+SUB+INAC+CLAC+AND+OR+XOR+NOT)+T7•LD
                                      _
AC trbus = T5(LDAC+STAC+JUMP+JMPZ•Z+JPNZ•Z)
read =
                                        _                                    _
T1+T3(LDAC+STAC+JUMP+JMPZ•Z+JPNZ•Z)+T4(LDAC+STAC+JUMP+JMPZ•Z+JPNZ•Z)+T7•S
TAC write = T7•STAC
ALU1 = T5(ADD+SUB+INAC)
```

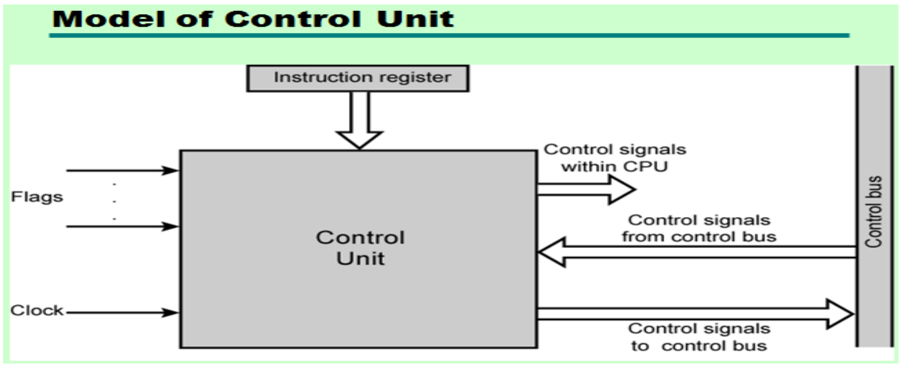
Functions of Control Unit:

1. Sequencing: Causing the CPU to step through a series of micro-operations
2. Execution: Causing the performance of each micro-op

3. This is done using Control Signals

Control Signals:

1. Clock: One micro-instruction (or set of parallel micro-instructions) per clock cycle
2. Instruction register: Op-code for current instruction, Determines which micro-instructions are performed
3. Flags: State of CPU, Results of previous operations
4. From control bus: Interrupts, Acknowledgements


Model of Control Unit
Instruction register
Control signals within CPU
Flags
Control Unit
Control signals from control bus
Control bus
Clock
Control signals to control bus

Control Signals – output

- Within CPU
  - Cause data movement
  - Activate specific functions
- Via control bus
  - To memory
  - To I/O modules

Example Control Signal Sequence – Fetch

- MAR <- (PC)
  - Control unit activates signal to open gates between PC and MAR
- MBR <- (memory)
  - Open gates between MAR and address bus
  - Memory read control signal

Open gates between data bus and MBR