

EENG 394: DIGITAL ELECTRONICS

Course purpose

To impart skills and knowledge on number systems and codes, Boolean algebra, minimization of logic expressions and sequential logic circuits.

Expected learning outcomes

By the end of the course the student should be able to;

- Convert numbers from one system to another
- Simplify logic expressions using Boolean algebra laws and K_map
- Design combinatorial circuit using various gates
- Design and implement sequential circuits

Course content

Number systems. Codes and number representation. Sign magnitude numbers. The (r-1)'s complement, r's complement. Boolean algebra and logic gates. Flipflop types: JK, SR, D, T. Master-slave, Level-driven, edgetriggered. Sequential logic: counters, registers, logic sequences. Laws of Boolean algebra and their conversion into logic diagrams. De-morgan's theorem. Reduction of Boolean expressions, sum-of-products equations and logic circuits. Simplification using Karnaugh maps. Arithmetic circuits. Exclusive –OR (XOR) gates. Halfadders. Full-adders and parallel operation of a full-adder and subtractors. definitions – logic conventions: RTL, DTL, TTL ECL, CMOS logic circuits

Teaching methodology and materials

Lectures, Tutorials, Laboratory experiments, Demonstrations and illustrations; Laboratory experiments, Demonstrations and illustrations

Instructional materials and equipment

Overhead projector, Power point, Flip charts, Hand-outs, Charts and Felt Pens.

References

- Jain R.P (2011). Modern Digital Electronics. 6th. Edition, Tata McGraw Hill Education: New Delhi.
- Maini, A.K. (2007). Digital Electronics: Principles, Devices and Applications: West Sussex, John Wiley & Sons
- Godse, A.P. Godse, D.A. (2009). Digital Electronics and Logic Design: Pune, Technical Publications Pune
- Godse, A.P. Godse, D.A. (2008). Digital Electronics. 3rd Revised Edition: Pune, Technical Publications Pune

Number Systems

The number systems are used quite frequently in the field of digital electronics and computers. However, the type of number system used in computers could be different at different stages of the usage.

For example, when a user key-in some data into the computer, s(he), will do it using decimal number system. But when the information goes inside the computer, it needs to be converted to a form suitable for processing data by the digital circuitry. Similarly, when the data has to be displayed on the monitor for the user, it has to be again in the decimal number system. Hence the conversion from one number system to another one is an important topic to be understood.

There are four systems of arithmetic which are often used in digital circuits. These systems are:

1. **Decimal**-it has a base (or radix) of 10 i.e. it uses 10 different symbols to represent numbers.
2. **Binary**-it has a base of 2 i.e. it uses only two different symbols.
3. **Octal**-it has a base of 8 i.e. it uses eight different symbols.
4. **Hexadecimal**-it has a base of 16 i.e. it uses sixteen different symbols.

All these systems use the same type of positional notation except that

- decimal system uses powers of 10
- binary system uses power of 2
- octal system uses powers of 8
- hexadecimal system uses powers of 16.

Decimal numbers are used to represent quantities which are outside the digital system.

Binary system is extensively used by digital systems like digital computers which operate on binary information.

Octal system has certain advantages in digital work because it requires less circuitry to get information into and out of a digital system. Moreover, it is easier to read, record and print out octal numbers than binary numbers.

Hexadecimal number system is particularly suited for microcomputers.

The Decimal Number System

This system has a base of 10 and is a position-value system (meaning that value of a digit depends on its position).

It has following characteristics:

Base or Radix

It is defined as the number of different digits which can occur in each position in the number System.

The decimal number system has a base of 10 meaning that it contains ten unique symbols (or digits). These are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Any one of these may be used in each position of the number.

As is well-known, it expresses 10 and any number above 10 as a combination of its ten unique symbols.

Position Value

The absolute value of each digit is fixed but **its position value (or place value or weight) is determined by its position in the overall number.**

For example, position value of 3 in 3000 is not the same as in 300. The number 2573 can be broken down as follows:

$$2573 = 2 \times 10^3 + 5 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$$

It will be noted that in this number, 3 is the least significant digit (LSD) whereas 2 is the most significant digit (MSD).

Again, the number 2573.469 can be written as

$$2573.469 = 2 \times 10^3 + 5 \times 10^2 + 7 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 6 \times 10^{-2} + 9 \times 10^{-3}$$

It is seen that position values are found by raising the base of the number system (i.e. 10 in this case) to the power of the position.

Also, powers are numbered to the left of the decimal point starting with 0 and to the right of the decimal point starting with -1.

Binary Number System

Radix

Its base or radix is two because it uses only two digits 0 and 1 (the word 'binary digit' is contracted to bit).

All binary numbers consist of a string of 0s and 1s.

To avoid confusion, a subscript of 10 is added for decimal numbers and 2 for binary numbers as illustrated below.

10_{10} , 101_{10} , 5742_{10} – decimal number and 10_2 , 101_2 , 110001_2 – binary numbers.

It may be noted that binary numbers need more places for counting because their base is small

Position Value

Like the decimal system, binary system is also positionally-weighted.

However, in this case, **the position value of each bit corresponds to some power of 2.** In each binary number, the value increases in powers of 2 starting with 0 to the left of the binary point and decreases to the right of the binary point starting with power of -1.

The decimal equivalent of the binary number may be found as under

$$\begin{aligned} 1101.011_2 &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) \\ &= 8 + 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} = 13.375_{10} \end{aligned}$$

Binary to Decimal Conversion

Following procedure should be adopted for converting a given binary integer (whole number) into its equivalent decimal number:

Step 1. Write the binary number i.e. all its bits in a row.

Step 2. Directly under the bits, write 1, 2, 4, 8, 16, starting from right to left.

Step 3. Cross out the decimal weights which lie under 0 bits.

Step 4. Add the remaining weights to get the decimal equivalent.

Example Convert 11001_2 to its equivalent decimal number.

Solution. The four steps involved in the conversion are as under

Step 1.	1	1	0	0	1
Step 2.	16	8	4	2	1
Step 3.	16	8	4	2	1
Step 4.	$16 + 8 + 1 = 25$			$\therefore$	$11001_2 = 25_{10}$

Binary Fractions

Here, procedure is the same as for binary integers except that the following weights are used for different bit positions.

	2^{-1}	2^{-2}	2^{-2}	2^{-4}	$\rightarrow$
	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\rightarrow$

Binary Point

Example Find the decimal equivalent of the 6-bit binary number 101.101_2 .

Solution.	1	0	1	1		0	1
	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	
	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$= 5 + \frac{1}{2} + \frac{1}{8} = 5.625$
$\therefore$	$101.101_2 = 5.625_{10}$						

Decimal to Binary Conversion

Integers

It is also known as divide-by-two method.

In this method, we progressively divide the given decimal number by 2 and write down the remainders after each division.

These remainders taken in the reverse order (i.e. from bottom-to-top) form the required binary number.

As an example, let us convert 25_{10} into its binary equivalent.

$25 \div 2 = 12 + \text{remainder of } 1$	 TOP
$12 \div 2 = 6 + \text{remainder of } 0$	
$6 \div 2 = 3 + \text{remainder of } 0$	
$3 \div 2 = 1 + \text{remainder of } 1$	
$1 \div 2 = 0 + \text{remainder of } 1$	
$\therefore 25_{10} = 11001_2$	BOTTOM

Fractions

In this case, Multiply-by-two rule is used i.e. we multiply each bit by 2 and record the carry in the integer position. These carries taken in the forward (top-to-bottom) direction gives the required binary fraction.

Let us convert 0.8125_{10} into its binary equivalent.

$0.8125 \times 2 = 1.625$	$= 0.625$	with a carry of 1	
$0.625 \times 2 = 1.25$	$= 0.25$	with a carry of 1	
$0.25 \times 2 = 0.5$	$= 0.5$	with a carry* of 0	
$0.5 \times 2 = 1.0$	$= 0.0$	with a carry of 1	
$\therefore 0.8125_{10} = 0.1101_2$			

Example Convert 25.625_{10} into its binary equivalent.

Solution. We will do the conversion in two steps (i) first for the integer and (ii) then for the fraction.

(a) Integer $25 \div 2 = 12 + 1$ $12 \div 2 = 6 + 0$ $6 \div 2 = 3 + 0$ $3 \div 2 = 1 + 1$ $1 \div 2 = 0 + 1$ $\therefore 25_{10} = 11001_2$ 	(b) fraction $0.625 \times 2 = 1.25 = 0.25 + 1$ $0.25 \times 2 = 0.5 = 0.5 + 0$ $0.5 \times 2 = 1.0 = 0.0 + 1$ $\therefore 0.625_{10} = 0.101_2$ 
---	---

Considering the complete number, we have $25.625_{10} = 11001.101_2$

Binary Operations

We will now consider the following four binary operations:

1. addition
2. subtraction
3. multiplication
4. division

Addition is the most important of these four operations.

In fact, by using ‘complements’, subtraction can be reduced to addition. Most digital computers subtract by complements.

It leads to reduction in hardware because only adding type of circuits are required. Similarly, multiplication is nothing but repeated addition and finally, division is nothing but repeated subtraction.

Binary Addition

Binary addition can be summarized as under:

$$\begin{array}{ll} 0 + 0 = 0 & 0 + 1 = 1 \\ 1 + 0 = 1 & 1 + 1 = 0 \quad \text{with a carry of 1} \end{array} \quad \text{or } = 10_2$$

Example Add 110011_2 to 101101_2

Solution.

$$\begin{array}{r} 110011 \\ 101101 \\ \hline 1100000 \end{array}$$

1. **first column :** $1 + 1 = 0$ with a carry of 1. Hence, we put down zero there and carry 1 to the second column.
2. **second column :** $1 + 0 = 1$. But combined with carry 1 from first column, it gives $1 + 1 = 0$ and a carry of 1. Hence, we put down 0 there and carry 1 further to the third column.
3. **third column :** $0 + 1 = 1$. Again, when it is combined with carry of 1 from the second column, we get $1 + 1 = 0$ with a carry of 1. Hence, again, we put down 0 and carry 1 to the fourth column.
4. **fourth column :** Here, $0 + 1 = 1$. When combined with carry 1 from third column, we get $1 + 1 = 0$ with a carry of 1. Hence, we put down 0 there and carry 1 to the fifth column.
5. **fifth column :** Here, $1 + 0 = 1$. When combined with the carry 1 from fourth column, we get $1 + 1 = 0$ with a carry of 1. Hence, we put down 0 there and carry 1 to the sixth column.
6. **sixth column :** Here, it is a case of $1 + 1 + \text{carry of } 1 = 11_2$ as stated earlier in (b) above.

Binary Subtraction

The four rules for binary subtraction are as under:

1. $0 - 0 = 0$,
2. $1 - 0 = 1$,
3. $1 - 1 = 0$,
4. $0 - 1 = 1$ with a borrow of 1 from the next column of the minuend
or $10 - 1 = 1$

Example Subtract 0111_2 from 1001_2

Solution.

$$\begin{array}{r} 1001 \\ - 0111 \\ \hline 0010 \end{array}$$

- 1st column : $1 - 1 = 0$
- 2nd column : $0 - 1 = 1$ with a borrow of 1
- 3rd column : 1 (after borrow) $- 1 = 0$
- 4th column : 0 (after borrow) $- 0 = 0$

Complement of a Number

In digital work, **two types of complements of a binary number are used for complemental subtraction:**

a) 1's complement

The 1's complement of a binary number is obtained by changing its each 0 into a 1 and each 1 into a 0.

It is also called radix-minus-one complement.

For example, 1's complement of 100_2 is 011_2 and of 1110_2 is 0001_2 .

b) 2's complement

The 2's complement of a binary number is obtained by adding 1 to its 1's complement.

$2's\ complement = 1's\ complement + 1$

It is also known as **true complement**.

Suppose we are asked to find 2's complement of 1011_2 . Its 1's complement is 0100_2 . Next, add 1 to get 0101_2 . Hence, 2's complement of 1011_2 is 0101_2 .

The complement method of subtraction reduces subtraction to an addition process.

This method is popular in digital computers because

1. only adder circuits are needed thus simplifying the circuitry,
2. it is easy with digital circuits to get the complements.

1's Complemental Subtraction

In this method, instead of subtracting a number, we add its 1's complement to the minuend.

The last carry (whether 0 or 1) is then added to get the final answer.

The rules for subtraction by 1's complement are as under:

1. compute the 1's complement of the subtrahend by changing all its 1s to 0s and all its 0s to 1s.
2. add this complement to the minuend
3. perform the end-around carry of the last 1 or 0
4. if there is no end-around carry (i.e. 0 carry), then the answer must be recomplemented and a negative sign attached to it.
5. if the end-around carry is 1, no recomplementing is necessary.

Suppose we want to subtract 101_2 from 111_2 . The procedure is as under:

$$\begin{array}{r} 111 \\ + 010 \quad \leftarrow \text{1's complement of subtrahend } 101 \\ \hline 1001 \\ \quad 1 \quad \leftarrow \text{end-round carry} \\ \hline 010 \end{array}$$

As seen, we have removed from the addition sum the 1 carry in the last position and added it onto the remainder. It is called end-around carry.

Let us now subtract 1101_2 from 1010_2 .

$$\begin{array}{r}
 1010 \\
 + 0010 \quad \leftarrow \text{1's complement of 1101} \\
 \hline
 1100 \\
 \text{NO CARRY}
 \end{array}$$

As seen, there is no end-around carry in this case. Hence, as per Rule 4 given above, answer must be recomplemented to get 0011 and a negative sign attached to it. Therefore, the final answer becomes -0011.

2's Complement Subtraction

In this case, the procedure is as under:

1. find the 2's complement of the subtrahend,
2. add this complement to the minuend,
3. drop the final carry,
4. if the carry is 1, the answer is positive and needs no recomplementing,
5. if there is no carry, subtract 1 then recomplement the answer and attach minus sign.

Example Using 2's complement, subtract 1010_2 from 1101_2 .

Solution. The 1's complement of 1010 is 0101. The 2's complement is $0101 + 1 = 0110$. We will add it to 1101.

$$\begin{array}{r}
 1101 \\
 + 0110 \quad \leftarrow \text{2's complement of } 1010_2 \\
 \hline
 10011 \\
 \text{DROP}
 \end{array}$$

The final answer is 0011_2 .

Example Use 2's complement to subtract 1101_2 from 1010_2 .

Solution. The 1's complement of 1101 is 0010. The 2's complement is 0011.

$$\begin{array}{r}
 1010 \\
 + 0011 \quad \leftarrow \text{2's complement of } 1101_2 \\
 \hline
 1101 \\
 \text{NO CARRY}
 \end{array}$$

In this case, there is no carry. Hence, we have to recomplement the answer. For this purpose, we first subtract 1 from it to get 1100.

Next, we recomplement it to get 0011. After attaching the minus sign, the final answer becomes -0011_2 .

Binary Multiplication

The procedure for this multiplication is the same as for decimal multiplication though it is comparatively much easier.

The four simple rules are as under:

1. $0 \times 0 = 0$, 2. $0 \times 1 = 0$, 3. $1 \times 0 = 0$, 4. $1 \times 1 = 1$.

As in the decimal system, the procedure is

1. copy the multiplicand when multiplier digit is 1 but not when it is 0
2. shift as in decimal multiplication
3. add the resulting binary numbers according to the rules of binary addition.

Example Multiply 111_2 by 101_2 using binary multiplication method.

Solution.

$$\begin{array}{r}
 111 \\
 \times 101 \\
 \hline
 111 \\
 000 \\
 111 \\
 \hline
 \end{array}$$

— shift left, no add
— shift left and add

Ans. 100011

Example Multiply 1101_2 by 1100_2

Solution.

$$\begin{array}{r}
 1101 \\
 \times 1100 \\
 \hline
 0000 \\
 0000 \\
 1101 \\
 1101 \\
 \hline
 10011100
 \end{array}$$

Binary Division

It is similar to the division in the decimal system. As in that system, here also division by 0 is meaningless. Rules are:

1. $0 \div 1 = 0$ or $\frac{0}{1} = 0$, 2. $1 \div 1 = 1$ or $\frac{1}{1} = 1$.

Example Carry out the binary division $11001 \div 101$

Solution.

$$\begin{array}{r} 101 \\ 101 \overline{) 11001} \\ \underline{101} \\ 10 \\ \underline{101} \\ \underline{101} \\ 000 \end{array}$$

After we bring down the next 0 bit, the number 10 so formed is not divisible by 101. Hence, we put a 0 in the quotient.

Therefore, the answer is 101_2 (decimal 5).

Incidentally, it may be noted that the dividend 25_{10} and divisor is 5_{10} so that the result of division, as expected, is 5_{10} .

Example Divide 11011_2 by 100_2

Solution.

$$\begin{array}{r} 110.11 \\ 100 \overline{) 11011} \\ \underline{100} \\ 101 \\ \underline{100} \\ 11 \\ \underline{110} \\ \underline{100} \\ 100 \\ \underline{100} \\ 000 \end{array}$$

$$\begin{array}{r} 6.75 \\ 4 \overline{) 27} \\ \underline{24} \\ 30 \\ \underline{28} \\ 20 \\ \underline{20} \\ 00 \end{array}$$

Octal Number System

Radix or Base

It has a base of 8 which means that it has eight distinct counting digits:

0, 1, 2, 3, 4, 5, 6, and 7

These digits 0 through 7, have exactly the same physical meaning as in decimal system.

For counting beyond 7, 2-digit combinations are formed taking the second digit followed by the first, then the second followed by the second and so on.

Hence, after 7, the next octal number is 10 (second digit followed by first), then 11 (second digit followed by second) and so on. Hence, different octal numbers are:

0,	1,	2,	3,	4,	5,	6,	7,
10,	11,	12,	13,	14,	15,	16,	17,
20,	21,	22,	23,	24,	25,	26,	27,
30,	31,	32,	...	...	...	...	...

Position Value

The position value (or weight) for each digit is given by different powers of 8 as shown below:

$$\leftarrow 8^3 \quad 8^2 \quad 8^1 \quad 8^0 \quad \cdot \quad 8^{-1} \quad 8^{-2} \quad 8^{-3} \rightarrow$$

↑
octal point

For example, decimal equivalent of octal 352 is

$$\begin{array}{ccccccc} 3 & 5 & 2 & \cdot & 0 & & \\ 8^2 & 8^1 & 8^0 & & & & \\ 64 & 8 & 1 & & & & \end{array} = 3 \times 64 + 5 \times 8 + 2 \times 1 = 234_{10}$$

$$352_8 = 3 \times 8^2 + 5 \times 8^1 + 2 \times 8^0 = 192 + 40 + 2 = 234_{10}$$

Similarly, decimal equivalent of octal 127.24 is

$$\begin{aligned} 127.24_8 &= 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 2 \times 8^{-1} + 4 \times 8^{-2} \\ &= 64 + 16 + 7 + \frac{2}{8} + \frac{4}{64} = 87.3125_{10} \end{aligned}$$

Octal to Decimal Conversion

Suppose, we want to convert octal 206.104₈ into its decimal equivalent number. The procedure is as under :

$$\begin{array}{ccccccc} 2 & 0 & 6 & \cdot & 1 & 0 & 4 \\ 8^2 & 8^1 & 8^0 & & 8^{-1} & 8^{-2} & 8^{-3} \end{array}$$

$$\therefore 206.104_8 = 2 \times 8^2 + 0 \times 8^1 + 6 \times 8^0 + \frac{1}{8} + \frac{0}{8^2} + \frac{4}{8^3} = 128 + 6 + \frac{1}{8} + \frac{4}{512} = \left(134 \frac{17}{128} \right)_{10}$$

Decimal to Octal Conversion

Let us see how we can convert 175₁₀ into its octal equivalent.

$175 \div 8 = 21$	with 7 remainder	$\uparrow$
$21 \div 8 = 2$	with 5 remainder	
$2 \div 8 = 0$	with 2 remainder	

Taking the remainders in the **reverse order**, we get 257_8 . $\therefore 175_{10} = 257_8$

Let us now take decimal fraction 0.15. Its octal equivalent can be found as under:

$0.15 \times 8 = 1.20 = 0.20$	with a carry of 1	$\downarrow$
$0.20 \times 8 = 1.60 = 0.60$	with a carry of 1	
$0.60 \times 8 = 4.80 = 0.80$	with a carry of 4	

$\therefore 0.15_{10} \equiv 114_8$

As seen, here carries have been taken in the **forward direction** i.e. from top to bottom.

Binary to Octal Conversion

The simplest procedure is to **use binary-triplet method**.

In this method, the given binary number is arranged into groups of 3 bits starting from the octal point and then each group is converted to its equivalent octal number.

Of course, where necessary, extra 0s can be added in front (i.e. left end) of the binary number to complete groups of three.

Suppose, we want to convert 101011_2 into its octal equivalent. Converting the bits into groups of three, we have

$$101 \quad 011$$

Now, 101_2 is 5 octal and 011 is 3 octal.

$$\therefore \begin{array}{cc} 101 & 011 \\ \downarrow & \downarrow \\ 5 & 3 \end{array}$$

$$\therefore 101011_2 = 53_8$$

Now, take 111110111_2 . We will first split it into groups of three bits (space is left between the groups for easy reading). Then, each group is given its octal number as shown below.

$$\begin{array}{ccc} 111 & 110 & 111 \\ \downarrow & \downarrow & \downarrow \\ 7 & 6 & 7 \end{array}$$

$$\therefore 111110111_2 = 767_8$$

Finally, take the example of a mixed binary number 10101.11_2 . Here, we will have to add one 0 in front of the integral part as well as to the fractional part

$$\begin{array}{ccc} 010 & 101 & 110 \\ \downarrow & \downarrow & \downarrow \\ 2 & 5 & 6 \end{array}$$

$$\therefore 10101.11_2 = 25.6_8$$

Octal to Binary Conversion

Here, each digit of the given octal number is converted into its equivalent binary triplet. For example, to change 75_8 into its binary equivalent, proceed as under:

$$\begin{array}{cc} 7 & 5 \\ \downarrow & \downarrow \\ 111 & 101 \end{array} \quad \therefore \quad 75_8 = 111\ 101_2$$

Similarly, 74.562_8 can be converted into binary equivalent as under:

$$\begin{array}{ccccc} 7 & 4 & . & 5 & 6 & 2 \\ \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow \\ 111 & 100 & & 101 & 110 & 010 \end{array} \quad \therefore \quad 74.562_8 = 111\ 100.101\ 110\ 010_2$$

Hexadecimal Number System

The characteristics of this system are as under:

1. it has a base of 16. Hence, it uses sixteen distinct counting digits 0 through 9 and A through F as detailed below :
2. place value (or weight) for each digit is in ascending powers of 16 for integers and descending powers of 16 for fractions.

The chief use of this system is in connection with byte-organised machines. It is **used for specifying addresses** of different binary numbers stored in computer memory.

Binary to Hexadecimal Conversion

The simple method is to split the given binary number into 4-bit groups (supplying 0s from our own side if necessary) and then give each group its 'hex' value as found from Table below

Binary	Hex.	Binary	Hex.	Binary	Hex.
0000	0	0110	6	1100	C
0001	1	0111	7	1101	D
0010	2	1000	8	1110	E
0011	3	1001	9	1111	F
0100	4	1010	A	10000	10
0101	5	1011	B	10001	11

Let us see how we would convert 10001100_2 into its hexadecimal equivalent number. We will first split the given binary number into 4-bit groups and then give each group its proper value from the table above

$$\begin{array}{cc} 1000 & 1100 \\ \downarrow & \downarrow \\ 8 & C \end{array} \quad \therefore \quad 1000\ 1100_2 = 8C_{16}$$

Let us now consider 1011010111_2 . Following the above procedure, we have

$$\begin{array}{ccc}
 0010 & 1101 & 0111 \\
 \downarrow & \downarrow & \downarrow \\
 2 & D & 7
 \end{array}
 \quad \therefore \quad 1011010111_2 = 2D7_{16}$$

It is seen that two 0s have been added to complete the 4-bit groups.

Hexadecimal to Binary Conversion

Each hexadecimal digit is converted into its equivalent 4-bit binary.

Suppose, we want to convert $23A_{16}$ into its binary equivalent. It can be done as given below:

$$\begin{array}{ccc}
 2 & 3 & A \\
 \downarrow & \downarrow & \downarrow \\
 0010 & 0011 & 1010
 \end{array}
 \quad \therefore \quad 23A_{16} = 0010\ 0011\ 1010_2$$

Decimal to Hexadecimal Conversion

Two methods are available for such a conversion.

One is to go from decimal to binary and then to hexadecimal.

The other method called hex dabble method where we use 16 for successive divisions. As an example let us convert decimal 1983 into hexadecimal

$$\begin{array}{lcl}
 \text{Hence, } 1983_{10} = 7BF_{16} & 1983 \div 16 = 123 + 15 \rightarrow & F \\
 & 123 \div 16 = 7 + 11 \rightarrow & B \\
 & 7 \div 16 = 0 + 7 \rightarrow & 7
 \end{array}
 \quad \uparrow$$

Hexadecimal to Decimal Conversion

Two methods are available for such a conversion.

One is to convert from hexadecimal to binary and then to decimal.

The other direct method is as follows:

Use power of 16 for the weights. Then, sum up the products of hexadecimal digits and their weights to get the decimal equivalent.

As an example, let us convert $F6D9$ to decimal.

$$\begin{aligned}
 F6D9 &= F(16^3) + 6(16^2) + D(16^1) + 9(16^0) = 15 \times 16^3 + 6 \times 16^2 + 13 \times 16^1 + 9 \times 16^0 \\
 &= 61,440 + 1536 + 208 + 9 = 63,193_{10}
 \end{aligned}$$

Binary Codes

Binary Coded Decimal

The **binary coded decimal (BCD)** is a type of binary code used to represent a given decimal number in an equivalent binary form.

BCD-to-decimal and decimal-to-BCD conversions are very easy and straightforward.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four-bit binary equivalent.

As an example, the BCD equivalent of $(23.15)_{10}$ is written as $(0010\ 0011.0001\ 0101)_{\text{BCD}}$. The BCD code described above is more precisely known as the 8421 BCD code, with 8, 4, 2 and 1 representing the weights of different bits in the four-bit groups, starting from MSB and proceeding towards LSB. This feature makes it a weighted code, which means that each bit in the four-bit group representing a given decimal digit has an assigned weight. Other weighted BCD codes include the 4221 BCD and 5421 BCD codes. Again, 4, 2, 2 and 1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code represent weights of the relevant bits.

Since the 8421 code is the most popular of all the BCD codes, it is simply referred to as the BCD code.

BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent. The first step is straightforward, and the second step will be explained.

As an example, we will find the binary equivalent of the BCD number 0010 1001.0111 0101:

- BCD number: 0010 1001.0111 0101.
- Corresponding decimal number: 29.75.
- The binary equivalent of 29.75 can be determined to be 11101 for the integer part and .11 for the fractional part.

Integer part=29

29/2 14 rem 1

14/2 7 rem 0

7/2 3 rem 1

3/2 1 rem 1

$\frac{1}{2}$ 0 rem 1

$29_{10} = 11101_2$

Fraction part= 0.75

0.75x2 0.5 carry 1

0.5x2 0 carry 1

$0.75_{10} = .11_2$

Therefore, $(0010\ 1001.0111\ 0101)_{\text{BCD}} = (11101.11)_2$.

Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order. A given binary number can be converted into an equivalent BCD number by first determining its decimal equivalent and then writing the corresponding BCD equivalent. As an example, we will find the BCD equivalent of the binary number 10101011.101:

Integer part=10101011

$$1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = 1 + 2 + 0 + 8 + 0 + 32 + 0 + 128 = 171$$

$$\text{Fraction part} = 101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0 + 0.125 = 0.625$$

- The decimal equivalent of this binary number can be determined to be 171.625.
- The BCD equivalent can then be written as 0001 0111 0001.0110 0010 0101.

Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953.

It is an unweighted binary code in which two successive values differ only by 1 bit. Owing to this feature, the maximum error that can creep into a system using the binary Gray code to encode data is much less than the worst-case error encountered in the case of straight binary encoding.

Table below lists the binary and Gray code equivalents of decimal numbers 0–15. An examination of the four-bit Gray code numbers, as listed in Table below, shows that the last entry rolls over to the first entry. That is, the last and the first entry also differ by only 1 bit. This is known as the *cyclic property* of the Gray code. Although there can be more than one Gray code for a given word length, the term was first applied to a specific binary code for non-negative integers and called the *binary-reflected Gray code* or simply the Gray code.

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

Binary–Gray Code Conversion

A given binary number can be converted into its Gray code equivalent by going through the following steps:

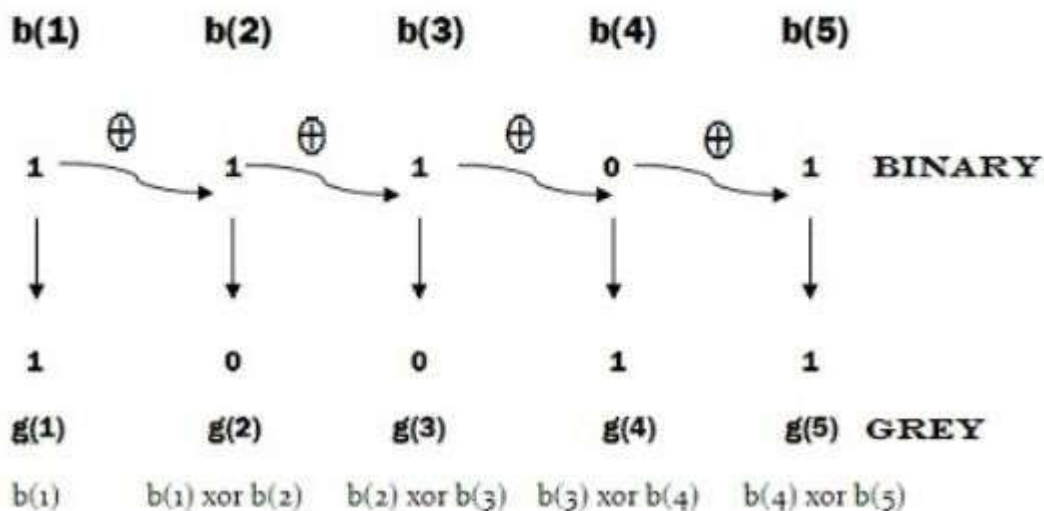
1. Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.

2. The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both '1', then the corresponding Gray code bit would be a '0'.
3. The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.
4. The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of $(1011)_2$ into its Gray code equivalent:

Binary	1011
Gray code	1- - -
Binary	1011
Gray code	11- -
Binary	1011
Gray code	111-
Binary	1011
Gray code	1110

Convert the binary 11101_2 to its equivalent Grey code



Gray Code–Binary Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

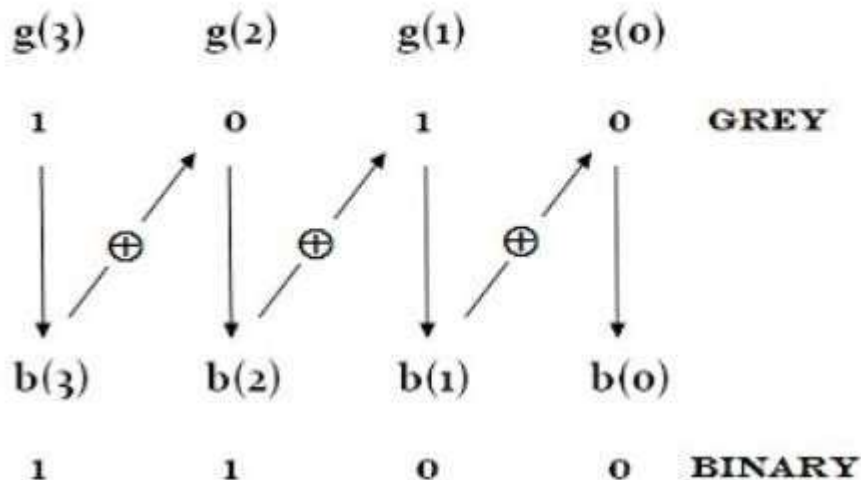
1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.
2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.

3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.
4. The process continues until we obtain the LSB of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code	1110
Binary	1 - - -
Gray code	1110
Binary	10 - -
Gray code	1110
Binary	101
Gray code	1110
Binary	1011

Convert the Grey code 1010 to its equivalent Binary



Applications

1. The Gray code is used in the transmission of digital signals as it minimizes the occurrence of errors.
2. The Gray code is preferred over the straight binary code in angle-measuring devices. Use of the Gray code almost eliminates the possibility of an angle misread, which is likely if the angle is represented in straight binary. The cyclic property of the Gray code is a plus in this application.
3. The Gray code is used for labelling the axes of Karnaugh maps, a graphical technique used for minimization of Boolean expressions.

4. The use of Gray codes to address program memory in computers minimizes power consumption. This is due to fewer address lines changing state with advances in the program counter.
5. Gray codes are also very useful in genetic algorithms since mutations in the code allow for mostly incremental changes. However, occasionally a one-bit change can result in a big leap, thus leading to new properties.

Find (a) the Gray code equivalent of decimal 13 and (b) the binary equivalent of Gray code number 1111.

Solution

- (a) The binary equivalent of decimal 13 is 1101.

Binary–Gray conversion

Binary 1101

Gray 1 - - -

Binary 1101

Gray 10 - -

Binary 1101

Gray 101 -

Binary 1101

Gray 1011

- (b) *Gray–binary conversion*

Gray 1111

Binary 1 - - -

Gray 1111

Binary 10 - -

Gray 1111

Binary 101 -

Gray 1111

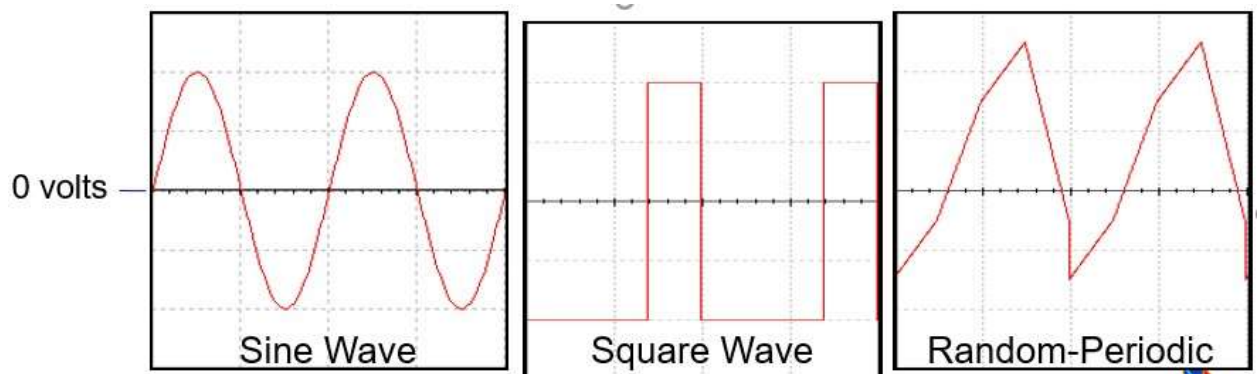
Binary 1010

Signals Analog and Digital Signals

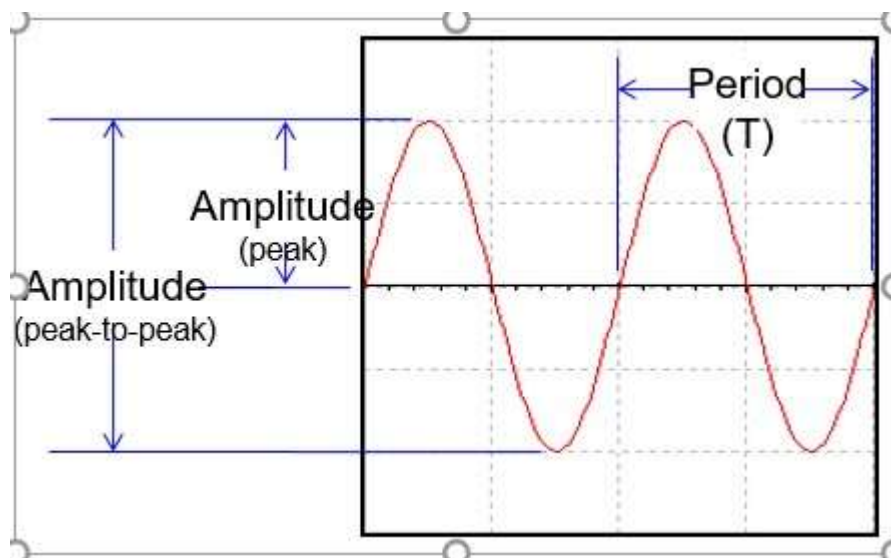
An analog signal is continuous, meaning that there are no breaks or interruptions.

Example of Analog Signals

- An analog signal can be any time-varying signal.
- Minimum and maximum values can be either positive or negative.
- They can be periodic (repeating) or non-periodic.
- Sine waves and square waves are two common analog signals.
- Note that this square wave is not a digital signal because its minimum value is negative.



Parts of an Analog Signal



The largest value reached in a half cycle is called the **peak value** or the **maximum value** or the **amplitude** of the waveform. **Such values are represented by V_m , I_m , E_m , etc.**

A **peak-to-peak** value is the difference between the maximum and minimum values in a cycle.

The time taken to complete one cycle is called the **period** or the **periodic time, T** (seconds), of the waveform.

The number of cycles completed in one second is called the **frequency, f** (Hertz, Hz).

Logic Levels

Before examining digital signals, we must define logic levels.

A **logic level** is a voltage level that represents a defined digital state.

Logic HIGH: The higher of two voltages, typically 5 volts

Logic LOW: The lower of two voltages, typically 0 volts

Logic Level	Voltage	True/False	On/Off	0/1
-------------	---------	------------	--------	-----

HIGH	5 volts	True	On	1
LOW	0 volts	False	Off	0

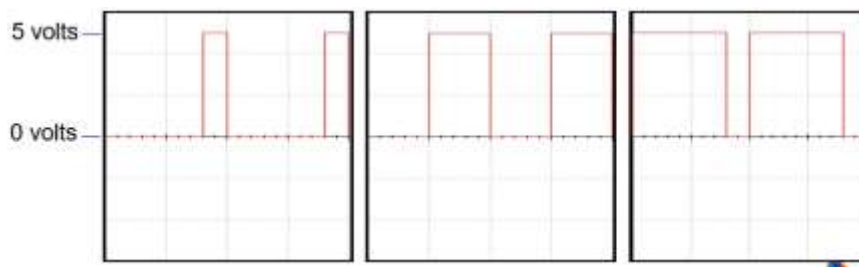
A digital signal is a signal that represent data as a sequence of discrete values; at any given time it can only take on one of a finite number of values.

Digital signals are not continuous.

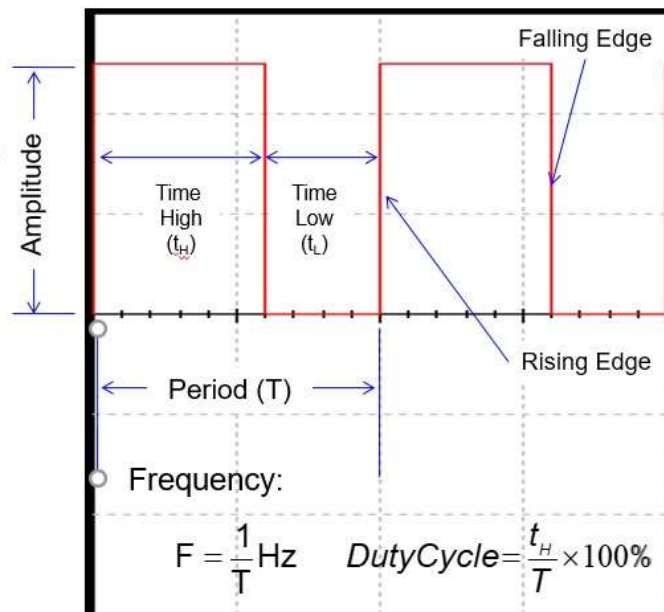
They use specific values to represent information.

Example of Digital Signals

- Digital signal are commonly referred to as clock signals.
- Their minimum value must be 0 volts, and their maximum value must be 5 volts.
- They can be periodic (repeating) or non-periodic.
- The time the signal is high (t_H) can vary anywhere from 1% of the period to 99% of the period.



Parts of a Digital Signal



Amplitude:

For digital signals, this will ALWAYS be 5 volts.

Time High (t_H):

The time the signal is at 5 v.

Time Low (t_L):

The time the signal is at 0 v.

Duty Cycle:

The ratio of t_H to the total period (T).

Rising Edge:

A 0-to-1 transition of the signal.

Falling Edge:

A 1-to-0 transition of the signal.

Logic Gates

A logic gate is an electronic circuit which makes logic decisions.

It has one output and one or more inputs.

The output signal appears only for certain combinations of input signals.

Logic gates are **the basic building blocks** from which most of the digital systems are built up. They implement the hardware logic function based on the logical algebra developed by George Boole which is called Boolean algebra in his honour.

A unique characteristic of the Boolean algebra is that variables used in it can assume only one of the two values i.e. either 0 or 1. Hence, every variable is either a 0 or a 1.

These gates are available today in the form of various IC families. The most popular families are: transistor-transistor logic (TTL), emitter-coupled logic (ECL), metal-oxide-semiconductor (MOS) and complementary metal-oxide-semiconductor (CMOS).

Here, we will consider the OR, AND, NOT, NOR, NAND, exclusive OR (XOR) and exclusive NOR (XNOR) gates along with their truth tables.

Positive and Negative Logic

In computing systems, the number symbols 0 and 1 represent two possible states of a circuit or device.

It makes no difference if these two states are referred to as ON and OFF, CLOSED and OPEN, HIGH and LOW, PLUS and MINUS or TRUE and FALSE depending on the circumstances.

Main point is that they must be symbolized by two opposite conditions.

In positive logic, a 1 represents

1. an ON circuit
2. a CLOSED switch
3. a HIGH voltage
4. a PLUS sign
5. a TRUE statement

Consequently, a 0 represents

1. an OFF circuit
2. an OPEN switch
3. a LOW voltage
4. a MINUS sign
5. a FALSE statement.

In negative logic, just opposite conditions prevail.

Suppose, a digital system has two voltage levels of 0V and 5. If we say that symbol 1 stands for 5V and symbol 0 for 0V, then we have positive logic system. If, on other hand, we decide that a 1 should represent 0 V and 0 should represent 5V, then we will get negative logic system.

Truth Table

A truth table is a table with a list all possible combinations of input binary variables and the corresponding outputs of a logic system.

The logic system output can be found from the logic expression, often referred to as the Boolean expression that relates the output with the inputs of that very logic system.

When the number of input binary variables is only one, then there are only two possible inputs, i.e. '0' and '1'. If the number of inputs is two, there can be four possible input combinations, i.e. 00, 01, 10 and 11.

OR Gate

An OR gate performs an ORing operation on two or more than two logic variables.

The OR operation on two independent logic variables A and B is written as $Y = A+B$ and reads as Y equals A OR B and not as A plus B.

The output of an OR gate is LOW only when all of its inputs are LOW.

For all other possible input combinations, the output is HIGH. This statement when interpreted for a positive logic system means the following.

The output of an OR gate is a logic '0' only when all of its inputs are at logic '0'. For all other possible input combinations, the output is a logic '1'. Figure 1 shows the circuit symbol and the truth table of a two-input OR gate. The operation of a two-input OR gate is explained by the logic expression

$$Y = A + B$$

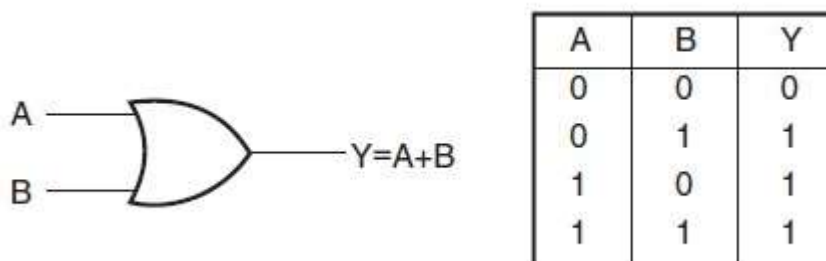
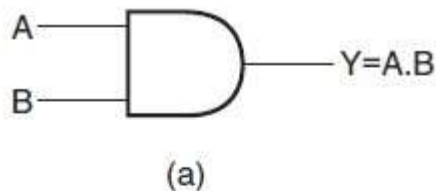


Figure 1: Two-input OR gate.

AND Gate

The output of an AND gate is HIGH only when all of its inputs are in the HIGH state. In all other cases, the output is LOW.

When interpreted for a positive logic system, this means that the output of the AND gate is a logic '1' only when all of its inputs are in logic '1' state. In all other cases, the output is logic '0'. The logic symbol and truth table of a two-input AND gate are shown in Figs 2 (a) and (b) respectively.



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

(b)

Figure 2: Two-input AND gate.

The AND operation on two independent logic variables A and B is written as $Y = A.B$ and reads as Y equals A AND B and not as A multiplied by B. Here, A and B are input logic variables and Y is the output. An AND gate performs an ANDing operation:

NOT Gate

A NOT gate is a one-input, one-output logic circuit whose output is always the complement of the input.

That is, a LOW input produces a HIGH output, and vice versa.

When interpreted for a positive logic system, a logic '0' at the input produces a logic '1' at the output, and vice versa.

It is also known as a 'complementing circuit' or an 'inverting circuit'. Figure 3 shows the circuit symbol and the truth table.

The NOT operation on a logic variable X is denoted as $\overline{X}$ or X' . That is, if X is the input to a NOT circuit, then its output Y is given by $Y = \overline{X}$ or X' and reads as Y equals NOT X. Thus, if $X = 0$, $Y = 1$ and if $X = 1$, $Y = 0$.

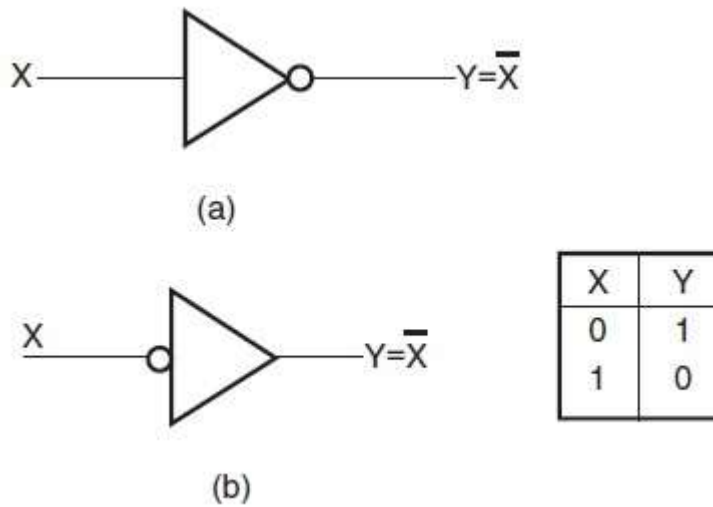


Figure 3 (a) Circuit symbol of a NOT circuit and (b) the truth table of a NOT circuit.

EXCLUSIVE-OR Gate

The EXCLUSIVE-OR gate, commonly written as EX-OR gate.

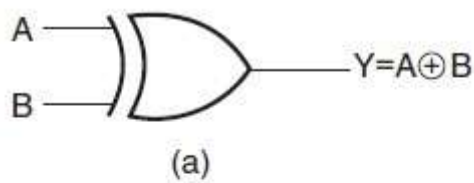
Figures 4(a) and (b) respectively show the logic symbol and truth table of a two-input EX-OR gate.

As can be seen from the truth table, the output of an EX-OR gate is a logic '1' when the inputs are unlike and a logic '0' when the inputs are like.

Although EX-OR gates are available in integrated circuit form only as two-input gates, unlike other gates which are available in multiple inputs also, multiple-input EX-OR logic functions can be implemented using more than one two-input gates.

The truth table of a multiple-input EX-OR function can be expressed as follows. The output of a multiple-input EX-OR logic function is a logic '1' when the number of 1s in the input sequence is odd and a logic '0' when the number of 1s in the input sequence is even, including zero. That is, an all 0s input sequence also produces a logic '0' at the output. The output of a two-input EX-OR gate is expressed by

$$Y = (A \oplus B) = \overline{A}B + A\overline{B}$$



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

(b)

Figure 4 (a) Circuit symbol of a two-input EXCLUSIVE-OR gate, (b) the truth table of a two-input EXCLUSIVE-OR gate

NAND Gate

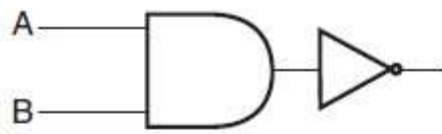
NAND stands for NOT AND.

An AND gate followed by a NOT circuit makes it a NAND gate

[Fig.5 (a)]. Figure 5 (b) shows the circuit symbol of a two-input NAND gate. The truth table of a NAND gate is obtained from the truth table of an AND gate by complementing the output entries [Fig.5 (c)].

The output of a NAND gate is a logic '0' when all its inputs are a logic '1'. For all other input combinations, the output is a logic '1'. NAND gate operation is logically expressed as

$$Y = \overline{A \cdot B}$$



(a)



(b)

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

(c)

Figure 5 (a) Two-input NAND implementation using an AND gate and a NOT circuit, (b) the circuit symbol of a two-input NAND gate and (c) the truth table of a two-input NAND gate.

NOR Gate

NOR stands for NOT OR.

An OR gate followed by a NOT circuit makes it a NOR gate [Fig. 6(a)].

The truth table of a NOR gate is obtained from the truth table of an OR gate by complementing the output entries.

The output of a NOR gate is a logic '1' when all its inputs are logic '0'. For all other input combinations, the output is a logic '0'. The output of a two-input NOR gate is logically expressed as

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

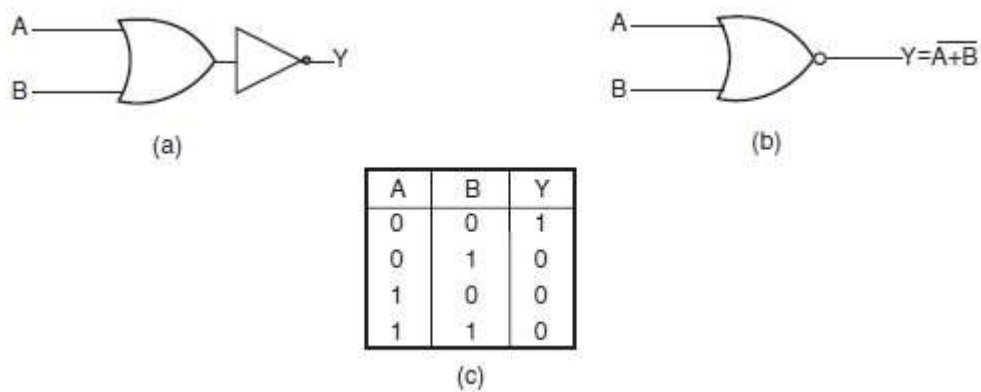


Figure 6 (a) Two-input NOR implementation using an OR gate and a NOT circuit, (b) the circuit symbol of a two-input NOR gate and (c) the truth table of a two-input NOR gate.

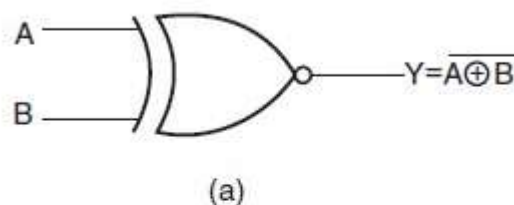
EXCLUSIVE-NOR Gate

EXCLUSIVE-NOR (commonly written as EX-NOR/ EX-NOR) means NOT of EX-OR, i.e. the logic gate that we get by complementing the output of an EX-OR gate.

Figure below shows its circuit symbol along with its truth table.

The truth table of an EX-NOR gate is obtained from the truth table of an EX-OR gate by complementing the output entries.

$$Y = (\overline{A \oplus B}) = (A.B + \overline{A}.\overline{B})$$



(b)

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

(a) Circuit symbol of a two-input EXCLUSIVE-NOR gate and (b) the truth table of a two-input EXCLUSIVE-NOR gate.

The output of a two-input EX-NOR gate is a logic '1' when the inputs are like and a logic '0' when they are unlike.

In general, the output of a multiple-input EX-NOR logic function is a logic '0' when the number of 1s in the input sequence is odd and a logic '1' when the number of 1s in the input sequence is even including zero.

That is, an all 0s input sequence also produces a logic '1' at the output.

Universal Gates

OR, AND and NOT gates are the three basic logic gates as they together can be used to construct the logic circuit for any given Boolean expression.

NOR and NAND gates have the property that they individually can be used to hardware implement a logic circuit corresponding to any given Boolean expression.

That is, it is possible to use either only NAND gates or only NOR gates to implement any Boolean expression.

This is so because a combination of NAND gates or a combination of NOR gates can be used to perform functions of any of the basic logic gates. It is for this reason that NAND and NOR gates are universal gates.

As an illustration, Fig. 7 a, b, c shows how two-input NAND gates can be used to construct a NOT circuit

[Fig. 7, (a)], a two-input AND gate [Fig.(b)] and a two-input OR gate [Fig.(c)]. Fig 8 shows the same using NOR gates. Understanding the conversion of NAND to OR and NOR to AND requires the use of DeMorgan's theorem, which is discussed on Boolean algebra

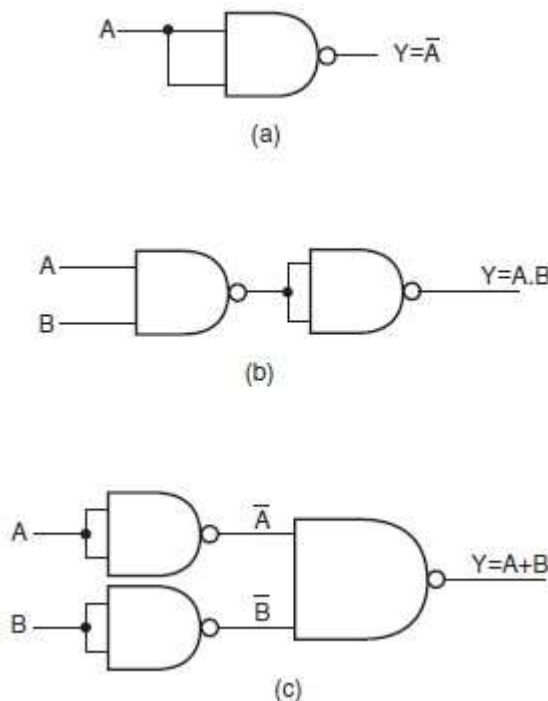


Fig 7: Implementation of basic logic gates using only NAND gates.

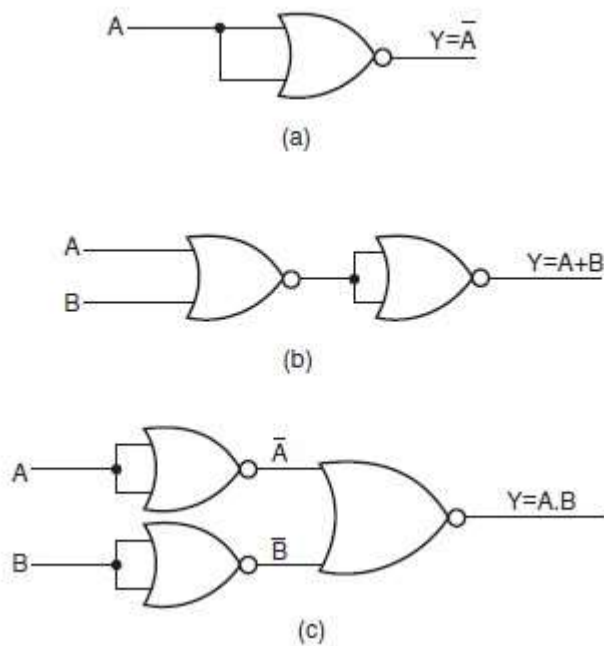


Fig 8: Implementation of basic logic gates using only NOR gates.

Unique Feature of Boolean Algebra

Boolean algebra have a unique property i.e. they can assume only one of the two possible values of 0 and 1.

Each of the variable used in a logical or Boolean equation can assume only the value 0 or 1. For example, in the logical equation $A + B = C$, each of the three variables A , B and C can have only the values of either 0 or 1. This point must be clearly taken note of by the reader for easy understanding of the laws of Boolean algebra.

Laws of Boolean Algebra

Boolean algebra is a system of Mathematics based on logic. It has its own set of fundamental laws which are necessary for manipulating different Boolean expressions.

1. OR Laws

Law 1. $A + 0 = A$

Law 2. $A + 1 = 1$

Law 3. $A + A = A$

Law 4. $A + \bar{A} = 1$

2. AND Laws

Law 5. $A \cdot 0 = 0$

Law 6. $A \cdot 1 = A$

Law 7. $A \cdot A = A$

Law 8. $A \cdot \bar{A} = 0$

3. Laws of Complementation

Law 9. $\bar{0} = 1$

Law 10. $\bar{1} = 0$

Law 11. if $A = 0$, then $\bar{A} = 1$

Law 12. if $A = 1$, then $\bar{A} = 0$

Law 13. $A = \bar{\bar{A}}$

4. Commutative Laws

These laws allow change in the position of variables in OR and AND expressions.

Law 14. $A + B = B + A$

Law 15. $A \cdot B = B \cdot A$

These two laws express the fact that the order in which a combination of terms is performed does not affect the final result of the combination.

5. Associative Laws

These laws allow removal of brackets from logical expression and regrouping of variables.

Law 16. $A + (B + C) = (A + B) + C$

Law 17. $(A + B) + (C + D) = A + B + C + D$

Law 18. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

6. Distributive Laws

These laws permit factoring or multiplying out of an expression.

Law 19. $A (B + C) = A B + A C$

Law 20. $A + B C = (A + B) (A + C)$

Law 21. $A + \bar{A} \cdot B = A + B$

7. Absorptive Laws

These enable us to reduce a complicated logic expression to a simpler form by absorbing some of the terms into existing terms.

Law 22. $A + A B = A$

Law 23. $A \cdot (A + B) = A$

Law 24. $A \cdot (\bar{A} + B) = A B$

The above laws can be used to prove any given Boolean identity and also for simplifying complicated expressions.

Example: Prove the following Boolean identity: $AC + ABC = AC$

Solution. Taking the left hand side expression as y , we get

$$\begin{aligned}
 y &= AC + ABC = AC(1+B) && \text{---Law 2} \\
 \text{Now } 1+B &= 1 && \text{---Law 2} \\
 \therefore y &= AC \cdot 1 = AC && \text{---Law 6} \\
 \therefore AC + ABC &= AC && = AC
 \end{aligned}$$

Example: Determine the logic expression for the output Y , from the truth table shown. Simplify and sketch the logic circuit for the simplified expression.

Inputs			Output
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

$\bar{A} \bar{B} C$ (for row 2)
 $A \bar{B} C$ (for row 6)

Fig. 71.6

The resulting expression for the output,

$$\begin{aligned}
 Y &= \bar{A} \bar{B} C + A \bar{B} C \\
 &= (\bar{A} + A) \bar{B} C && \dots(\text{Law 19}) \\
 &= 1 \cdot \bar{B} C && \dots(\text{Law 4}) \\
 &= \bar{B} C && \dots(\text{Law 6})
 \end{aligned}$$

Example: Prove the following Boolean identity: $(A + B)(A + C) = A + BC$ Solution: Putting the left hand side expression equal to y , we get

$$\begin{aligned}
 &= (A + B)(A + C) \\
 &= AA + AC + AB + BC \\
 &= A + AC + AB + BC \\
 &= A + AB + AC + BC \\
 &= A(1 + B) + AC + BC \\
 &= A + AC + BC \\
 &= A(1 + C) + BC \\
 &= A + BC \\
 (A + B)(A + C) &= A + BC
 \end{aligned}$$

DE Morgan's Theorem

These two theorems (or rules) are a great help in simplifying complicated logical expressions. The theorems can be stated as under:

Law 1.

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Law 2.

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

The first statement says that the complement of a sum equals the product of complements. The second statement says that the complement of a product equals the sum of the complements. In fact, it allows transformation from a sum-of-products form to a product-of-sum form.

As seen from the above two laws, **the procedure required for taking out an expression from under a NOT sign is as follows:**

1. complement the given expression i.e., remove the overall NOT sign
2. change all the ANDs to ORs and all the ORs to ANDs.
3. complement or negate all individual variables.

As an illustration, take the following example

$$\begin{aligned}\overline{A + BC} &= A + BC \\ &= A (B + C) \\ &= \overline{\overline{A} (\overline{B} + \overline{C})}\end{aligned}$$

Next, consider this example,

$$\begin{aligned}\overline{(\overline{A} + B + \overline{C})(\overline{A} + B + C)} &= (\overline{\overline{A} + B + \overline{C}})(\overline{\overline{A} + B + C}) \\ &= \overline{\overline{A}} \overline{B} \overline{\overline{C}} + \overline{\overline{A}} \overline{B} \overline{C} \\ &= \overline{\overline{A}} \overline{B} \overline{\overline{C}} + \overline{\overline{A}} \overline{B} \overline{C} \\ &= A \overline{B} \overline{C} + A \overline{B} \overline{C}\end{aligned}$$

Standard Forms of Boolean Expressions

All Boolean expressions, regardless of their form, can be converted into either of the two following **standard forms**

1. Sum-of-products (SOP) form and
2. Product-of-sums (POS) form.

The standardization of Boolean expressions makes their evaluation, simplification and implementation much more systematic and easier.

Now we shall discuss these two standard forms in more detail.

The Sum-of-Products (SOP) Form

A product term is a term consisting of the product (or Boolean multiplication) of literals (i.e. **variables or their complements**).

When we add two or more product terms, the resulting expression is called, sum-of-products (SOP) expression. Some examples of sum-of-products expressions are

$$\overline{A}\overline{B} + \overline{A}BC, \overline{A}BC + ACD + A\overline{B}CD, \overline{A}B + A\overline{C} + A\overline{B}\overline{C}$$

Sometimes, it is convenient to define the set of variables contained in the expression (in either complemented or uncomplemented form) as a domain.

For example, domain of the expression $\overline{A}\overline{B} + A\overline{B}$ is the set of variables A and B. Similarly, the domain of expression $\overline{A}BC + ABC + A\overline{B}CD$ is the set of variables A, B, C and D.

Any logic expression can be changed into SOP form by applying Boolean algebra laws and theorems. For example, the expression $A(BC + D)$ can be converted to SOP form by applying the distributive law:

$$A(BC + D) = ABC + AD$$

The Standard SOP Form

So far, we have seen SOP (sum-of-products) expressions in which some of the product terms do not contain all the variables in the domain of the expression.

For example, the expression, $\overline{A}BC + A\overline{C}D + A\overline{B}CD$ has a domain made up of the variables A, B, C and D. But notice that the first two terms contains only three variables, i.e. $\overline{D}$ or D is missing from the first term and $\overline{B}$ or B is missing from the second term.

A standard SOP expression is defined as an expression in which all the variables in the domain appear in each product term.

For example $\overline{A}BCD + AB\overline{C}D + A\overline{B}\overline{C}D$ is a standard SOP expression.

Standard SOP expressions are important in constructing truth tables and in karnaugh map simplification method.

It is very straightforward to convert non-standard product term to a standard SOP using Boolean algebra. Each product term in the SOP expression that does not contain all the variables in the domain has to be expanded to standard form to include all the variables in the domain and their complements.

As stated below, a nonstandard SOP expression is converted into standard form using

Boolean algebra **law 4** : $A + \overline{A} = 1$ i.e. a variable added to its complement equals 1.

Step 1. Multiply each nonstandard product term by a term made up of the sum of a missing variable and its complement. This results in two product terms. It is possible because we know that we can multiply anything by 1 without changing its value.

Step 2. Repeat step 1 until all resulting product terms contain all variables in the domain in either complemented or uncomplemented form. Note that in converting a product term to a standard form, the number of product terms is doubled for each missing variable.

For example, suppose we want to convert the Boolean expression $\overline{A}BC + A\overline{C}D + A\overline{B}CD$ to a standard SOP form. Then following the above procedure we proceed as below:

$$\begin{aligned}\overline{A}BC(D + \overline{D}) + A(B + \overline{B})\overline{C}D + A\overline{B}CD \\ = \overline{A}BCD + \overline{A}BC\overline{D} + A\overline{B}\overline{C}D + A\overline{B}C\overline{D} + A\overline{B}CD\end{aligned}$$

The expression given above is a standard SOP expression.

The Product-of-sums (POS) Form

The sum term is a term consisting of the sum (or Boolean addition) of literals (i.e. variables or their complements).

When we multiply two or more sum terms, the resulting expression is called product-of-sums (POS). Some examples of POS form are

$$(A + \overline{B})(A + B + C), (A + B + \overline{C})(\overline{A} + C + \overline{D})(A + \overline{B} + C + D) \text{ and } (\overline{A} + B)(A + C)(\overline{A} + \overline{B} + C).$$

It may be carefully noted that a POS expression, can contain a single-variable term $\overline{A}$ as in $\overline{A}(\overline{A} + B + C)(B + \overline{C} + \overline{D})$.

In POS expression, a single overbar cannot extend over more than one variable, although more than one variable in a term can have an overbar.

The Standard POS Form

So far, we have seen POS (product-of-sums) expressions in which some of the sum terms do not contain all the variables in the domain of the expression.

For example, the expression $(\overline{A} + B + C)(A + \overline{C} + D)(A + \overline{B} + C + D)$ has a domain made up of variables, A, B, C and D. Notice that the complete set of variables in the domain is not represented in the first two terms of the expression, i.e. $\overline{D}$ or D is missing in the first term and $\overline{B}$ or B is missing in the second term.

A standard POS expression is defined as an expression in which all the variables in the domain appear in each sum term.

For example $(\overline{A} + B + C + D)(A + B + \overline{C} + D)(A + \overline{B} + C + D)$ is a standard POS form.

Any nonstandard POS expression can be converted to a standard form using Boolean algebra.

Each sum term in an POS expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements.

As stated below:

a nonstandard POS expression is converted into a standard form using **Boolean algebra Law 8** : $\bar{A} \cdot A = 0$, i.e. a variable multiplied by its complemented equals 0.

Step 1. Add to each nonstandard product term a term made up of the product of a missing variable and its complement. This results in two sum terms. This is possible because we know that we can add 0 to anything without changing its value.

Step 2. Apply law 20, i.e. $A + BC = (A + B)(A + C)$

Step 3. Repeat Step 1 until all resulting sum terms contain all variables in the domain in either complemented or uncomplemented form.

For example we want to convert the Boolean expression $(\bar{A} + B + C)(A + \bar{C} + D)(A + \bar{B} + C + D)$ into a standard POS form. Then following the above procedure, we proceed as below:
example

$$\begin{aligned} & (\bar{A} + B + C + D \cdot \bar{D})(A + B + \bar{C} + D)(A + \bar{B} + C + D) \\ &= (\bar{A} + B + C + D)(\bar{A} + B + C + \bar{D})(A + B + \bar{C} + D)(A + \bar{B} + \bar{C} + D)(A + \bar{B} + C + D) \end{aligned}$$

The expression given above is a standard POS expression

An expanded form of Boolean expression, where each term contains all Boolean variables in their true or complemented form, is also known as the **canonical form** of the expression.

CANONICAL FORMS OF BOOLEAN EXPRESSIONS

With one variable x & $\bar{x}$.

With two variables $\bar{x}\bar{y}$, $x\bar{y}$, $\bar{x}y$ and xy .

With three variables $\bar{x}\bar{y}\bar{z}$, $\bar{x}\bar{y}z$, $\bar{x}y\bar{z}$, $\bar{x}yz$, $x\bar{y}\bar{z}$, $x\bar{y}z$, $xy\bar{z}$ & xyz .

These eight AND terms are called minterms.

n variables can be combined to form 2^n minterms.

x	y	z	minterm	designation	maxterm	designation
0	0	0	$\bar{x}\bar{y}\bar{z}$	m_0	$x+y+z$	M_0
0	0	1	$\bar{x}\bar{y}z$	m_1	$x+y+\bar{z}$	M_1
0	1	0	$\bar{x}y\bar{z}$	m_2	$x+\bar{y}+z$	M_2
0	1	1	$\bar{x}yz$	m_3	$x+\bar{y}+\bar{z}$	M_3
1	0	0	$x\bar{y}\bar{z}$	m_4	$\bar{x}+y+z$	M_4
1	0	1	$x\bar{y}z$	m_5	$\bar{x}+y+\bar{z}$	M_5
1	1	0	$xy\bar{z}$	m_6	$\bar{x}+\bar{y}+z$	M_6
1	1	1	xyz	m_7	$\bar{x}+\bar{y}+\bar{z}$	M_7
(AND terms)				(OR terms)		

Note that each maxterm is the complement of its corresponding minterm and vice versa.

For example the function F

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

$$F = \overline{x} \overline{y} z + x \overline{y} \overline{z} + x y z$$

$$F = m_1 + m_4 + m_7$$

The complement of a function expressed as the product of maxterms equal to the product of maxterms missing from the original function.

To convert from one canonical form to another, interchange the symbols $\sum$, $\prod$ and list those numbers missing from the original form.

$$F = M_4 M_5 M_0 M_2 = m_1 + m_3 + m_6 + m_7$$

$$F(x, y, z) = \prod(0, 2, 4, 5) = \sum(1, 3, 6, 7)$$

The Karnaugh Map

The Karnaugh map (or simply a K-map) is similar to a truth table because it presents all the possible values of input variables and the resulting output for each value.

However, instead of being organised into columns and rows like a truth table, the **Karnaugh map is an array of squares (or cells) in which each square represents a binary value of the input variables.**

The squares are arranged in a way so that simplification of given expression is simply a matter of grouping the squares.

Karnaugh maps can be used for expression with two, three and four variable Karnaugh maps.

The number of squares in a Karnaugh map is equal to the total number of possible input variable combinations (as is the number of rows in a truth table).

For two variables, the number of square is $2^2 = 4$, for three variables, the number of squares is $2^3 = 8$ and for four variables, the number of squares is $2^4 = 16$.

The Two-variable Karnaugh Map

Fig. 1(a) shows a two-variable Karnaugh map. As seen, it is an array of four squares.

In this case, A and B are used for two variables although any other two letters could be used. The binary values of A (i.e. 0 and 1) are indicated along the left side as $\bar{A}$ and A (notice the sequence) and the binary values of B are indicated across the top as $\bar{B}$ and B .

The value of a given square is the value of A at the left in the same row combined with the value of B at the top in the same column.

For example, a square in the upper left corner has a value of $\bar{A}\bar{B}$ and a square in the lower right corner has a value of AB . Fig. 1 (b) shows the standard product terms represented by each square in the Karnaugh map.

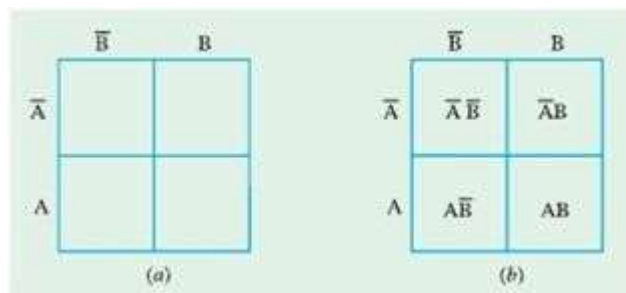


Fig. 1

The Three-variable Karnaugh Map

Fig. 2 (a) shows a three-variable Karnaugh map. As seen it is an array of eight squares.

In this case, A, B and C are used for the variables although any other three letters could be used. The value of A and B are along the left side (notice the sequence carefully) and the values of C are across the top.

The value of a given square is the values of A and B at the left in the same row combined with the value of C at the top in the same column.

For example, a square in the upper left corner has a value of $\bar{A}\bar{B}\bar{C}$ and a square in the bottom right corner has a value of ABC . Fig. 2 (b) shows the product terms that are represented by each square in the Karnaugh map.

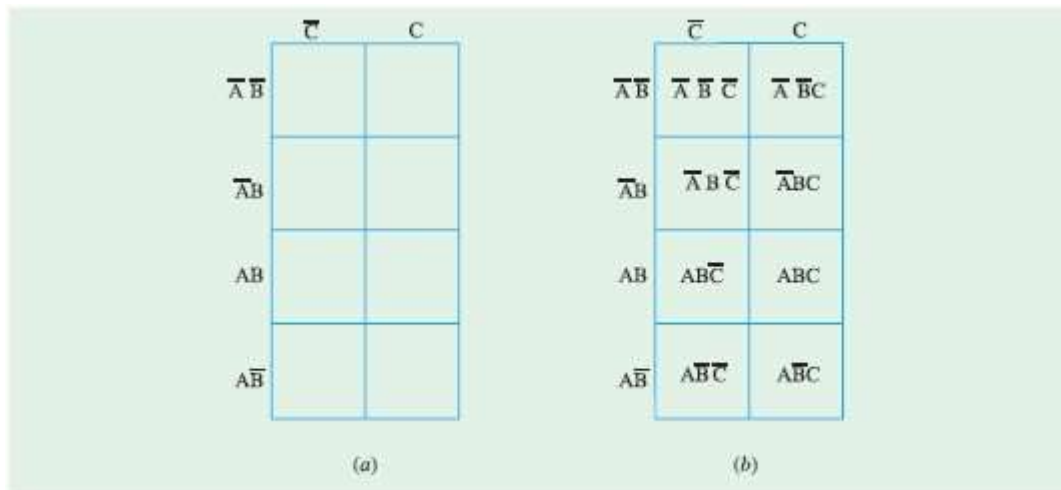


Fig. 2

The Four-variable Karnaugh Map

Fig. 3 (a) shows a four-variable Karnaugh map. As seen, it is an array of sixteen squares. In this case A, B, C and D are used for the variables. The values of A and B are along the left side, and the values of C and D are across the top. The sequence of the variable values may be noted carefully.

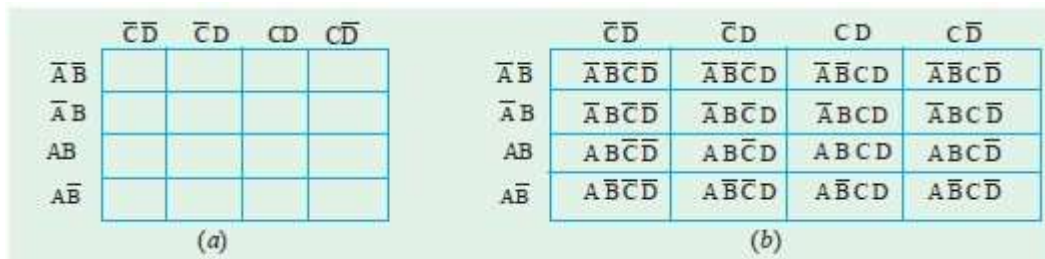


Fig. 3

The value of a given square is the values of A and B at the left in the same row combined with the values of C and D at the top in the same column.

For example, a square in the upper right corner has a value $\overline{A} \overline{B} \overline{C} \overline{D}$ and a square in the lower right corner has a value $A \overline{B} C \overline{D}$. Fig. 3 (b) shows the standard product terms that are represented by each square in the four-variable Karnaugh map.

Square Adjacency in Karnaugh Map

It will be interesting to know that the squares in a Karnaugh map are arranged in such a way that there is only a single-variable change between adjacent squares.

Adjacency is defined as a single-variable change. It means the squares that differ by only one variable are adjacent.

For example, in a three-variable Karnaugh map shown in Fig. 2 (b), the $\overline{A} \overline{B} \overline{C}$ square is adjacent to $\overline{A} \overline{B} C$ square, the $\overline{A} B \overline{C}$ square and the $A \overline{B} \overline{C}$ square.

It may be carefully noted that square with values that differ by more than one variable are not adjacent. For example, the $\overline{A}\overline{B}\overline{C}$ square is not adjacent to the $\overline{A}\overline{B}C$ square, the ABC square, the $A\overline{B}\overline{C}$ square or the $A\overline{B}C$ square. In other words, each square is adjacent to the squares that are immediately next to it on any of its four sides.

However, a square is not adjacent to the squares that diagonally touch any of its corners.

It may also be noted that squares in the top row are adjacent to the corresponding squares in the bottom row and squares in the outer left column are adjacent to the corresponding squares in the outer right column.

This is called “wrap around” adjacency because we can think of the map as wrapping around from top to bottom to form a cylinder or from left to right to form a cylinder.

Fig. 4 (a) and (b) shows the square adjacencies with a three-variable and a four-variable Karnaugh maps respectively. Notice the square adjacencies in a four variable Karnaugh map:

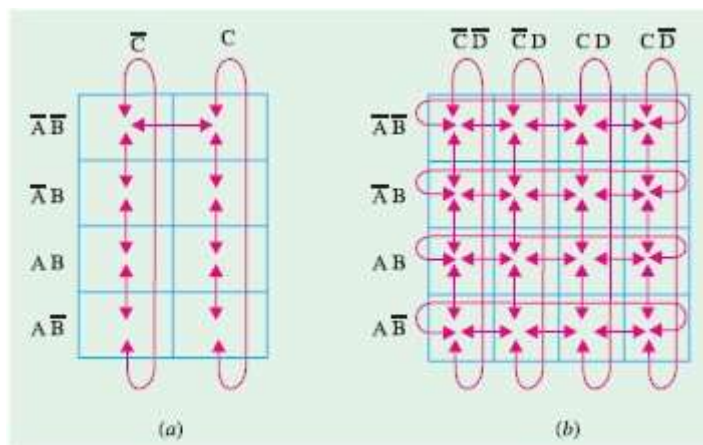


Fig. 4

Here for example, the square $\overline{A}\overline{B}\overline{C}\overline{D}$ is adjacent to $\overline{A}\overline{B}\overline{C}D$ square, $\overline{A}\overline{B}C\overline{D}$ square, $A\overline{B}\overline{C}\overline{D}$ square and $A\overline{B}C\overline{D}$ square. Similarly $\overline{A}B\overline{C}\overline{D}$ square is adjacent to $\overline{A}B\overline{C}D$ square, $\overline{A}B C\overline{D}$ square, $\overline{A}BCD$ square and $AB\overline{C}\overline{D}$ square.

Mapping a Standard SOP Expression on the Karnaugh Map

Consider a SOP (sum-of-products) expression $\overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}C + A\overline{B}\overline{C}$

In order to map this expression on the Karnaugh map, we need a three variable Karnaugh-map because the given expression has three variables A, B, and C. Then select the first product term and enter 1 in the corresponding square (i.e. the first row and the first column) as shown in Fig. 5

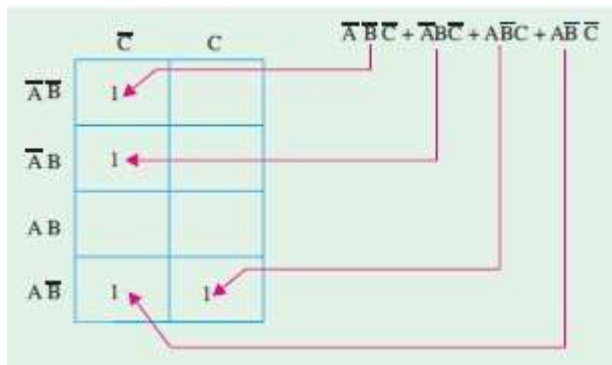


Fig 5

Similarly, for the second product term place a 1 in the second row and first column. Repeat this process for the other two product terms. The squares that do not have 1 are the squares for which the expression is 0. Usually when working with sum-of-products expressions, the 0s are left off the map.

Example 71.22. Map the following standard sum-of-products (SOP) expression on a Karnaugh map:

$$\overline{A} B C \overline{D} + A B C \overline{D} + A B \overline{C} \overline{D} + \overline{A} \overline{B} \overline{C} \overline{D} + A B \overline{C} D + \overline{A} \overline{B} C D + A \overline{B} C \overline{D} + A B C D$$

Solution. Sketch a four variable Karnaugh map. Select the first product term $\overline{A} B C \overline{D}$ from the given SOP expression and enter 1 in the corresponding square as shown in Fig. 7. Similarly enter 1 for the other product terms in the given SOP expression. Check that number of 1s in the Karnaugh Map is equal to the number of product terms in the given SOP expression.

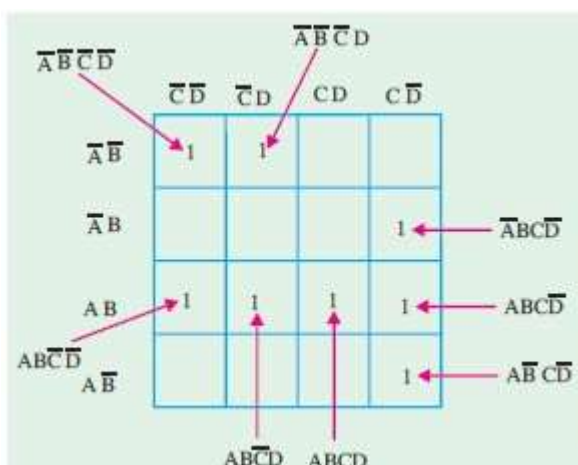


Fig .7

Mapping a Nonstandard SOP Expression on the Karnaugh Map

A nonstandard sum-of-products (SOP) expression is a one that has product terms with one or more missing variables.

In such a case, Boolean expression must first be converted to a standard form. Let us consider an example to illustrate the procedure for mapping a nonstandard SOP expression on the Karnaugh map. Suppose we have the SOP expression:

$$\bar{A} + A\bar{B} + ABC$$

As seen, this expression is obviously not in standard form because each product term does not have three variables. The first term is missing two variables, the second term is missing one variable and the third term is standard. In order to convert the given nonstandard SOP expression to a standard form, we multiply the first product term by $B + \bar{B}$ and $C + \bar{C}$, and the second term by $C + \bar{C}$.

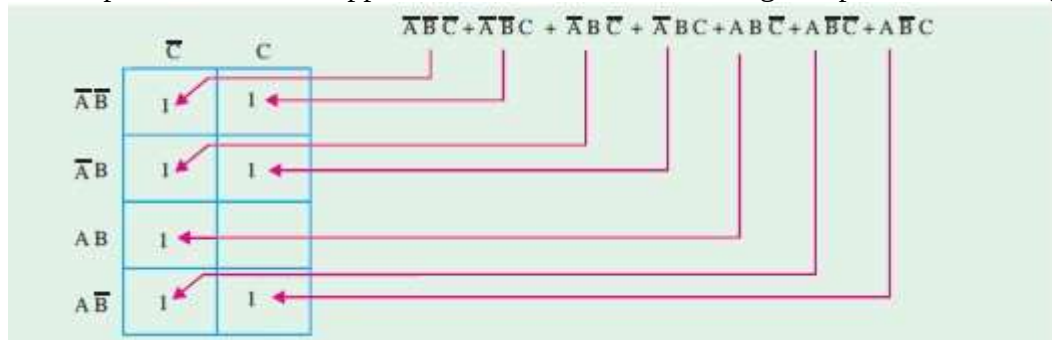
Expanding the resulting expression, we get,

$$\begin{aligned} & \bar{A}(B + \bar{B})(C + \bar{C}) + A\bar{B}(C + \bar{C}) + ABC \\ &= \bar{A}BC + \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}\bar{B}\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + ABC \end{aligned}$$

Rearranging the expression for our convenience, we get

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

This expression can be mapped on a three-variable Karnaugh map as shown in Fig. 8.



Simplification of Boolean Expression Using Karnaugh Map

The process that results in an expression containing the minimum number of possible terms with the minimum number of variables is called simplification or minimization.

After the SOP (or the Boolean) expression has been mapped on the Karnaugh map, there are three steps in the process of obtaining a minimum SOP expression.

The three steps are:

- a. grouping the 1s,
- b. determining the product term for each group and
- c. summing the resulting product terms.

(a) Grouping the 1s : We can group the 1s on the Karnaugh map according to the following rules by enclosing those adjacent squares containing 1s.

The objective is to maximize the size of the groups and to minimize the number of groups.

1. A group must contain either 1, 2, 4, 8 or 16 squares.
In the case of two-variable Karnaugh map, 4 squares is the maximum group, for three-variable map, 8 squares are the maximum group and so on.
2. Each square in the group must be adjacent to one or more squares in that same group but all squares in the same group do not have to be adjacent to each other.
3. Always include the largest possible number of 1s in a group in accordance with rule 1.
4. Each 1 on the Karnaugh map must be included in at least one group. The 1s already in a group can be included in another group as long as the overlapping groups include non-common 1s.

(b) Determining the Product Term for each group: Following are the rules that are applied to find the minimum product terms and the minimum sum-of-products expression :

1. Group the squares that have 1s : Each group of squares containing 1s creates one product term composed of all variables that occur in only one form (either uncomplemented or complemented) within the group. Variables that occur both uncomplemented and complemented within the group are eliminated. These are known as contradictory variables.
2. In order to determine the minimum product term for each group, we need to look at the standard methodology for three-variable and four-variable Karnaugh map respectively.
 - (i) For a three-variable K-map : (1) for 1-square, group we get a three-variable product term, (2) for a 2-square group, we get a two-variable product term, (3) for a 4-square product term, we get a one-variable product term.
 - (ii) For a four-variable K-map : (1) For a 1-square group, we get a four-variable product term, (2) for a 2-square group, we get a three-variable product term, (3) for a 4-square group, we get a two-variable product term and (4) for a 8-square group, we get a onevariable product term.

(c) Summing the resulting product terms : When all the minimum product terms are derived from the Karnaugh map, these are summed to form the minimum sum-of-products expression.

Note : In some cases, there may be more than one way to group the 1s to form the product terms. Whatever be the way, the minimal expression must have the same number of product terms and each product term, the same number of Boolean variables.

The example given below will help you to understand and apply the simplification of the SOP expression using Karnaugh map.

Example. Simplify the following Boolean expression using the Karnaugh mapping technique :

$$X = \overline{A} B + \overline{A} \overline{B} \overline{C} + AB \overline{C} + A \overline{B} \overline{C}$$

Solution. The first step is to map the given Boolean expression on the Karnaugh map. Notice that there are three variables A, B and C in the Boolean expression, therefore we need a three-variable Karnaugh map.

The Boolean expression to be mapped is,

$$X = \overline{A} B + \overline{A} \overline{B} \overline{C} + AB \overline{C} + A \overline{B} \overline{C}$$

Note that the given Boolean expression is a nonstandard SOP expression because the first product term $A B$ has the variable C missing in it. This can be converted into a standard SOP form by modifying the expression as below.

$$X = \overline{A} B . 1 + \overline{A} \overline{B} \overline{C} + AB \overline{C} + A \overline{B} \overline{C}$$

$$X = \overline{A} B (C + \overline{C}) + \overline{A} \overline{B} \overline{C} + AB \overline{C} + A \overline{B} \overline{C}$$

$$X = \overline{A} B C + \overline{A} B \overline{C} + \overline{A} \overline{B} \overline{C} + AB \overline{C} + A \overline{B} \overline{C}$$

Equation above can be mapped on the Karnaugh map as shown in Fig. 9.

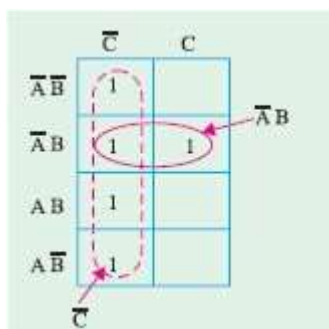


Fig. 9

$$X = \overline{A} B + \overline{C}$$

Example. Fig. 10 shows a Karnaugh map of a sum-of-products (SOP) function. Determine the simplified SOP function.

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	1		1
$\overline{A}B$		1		
AB				
$A\overline{B}$	1	1		1

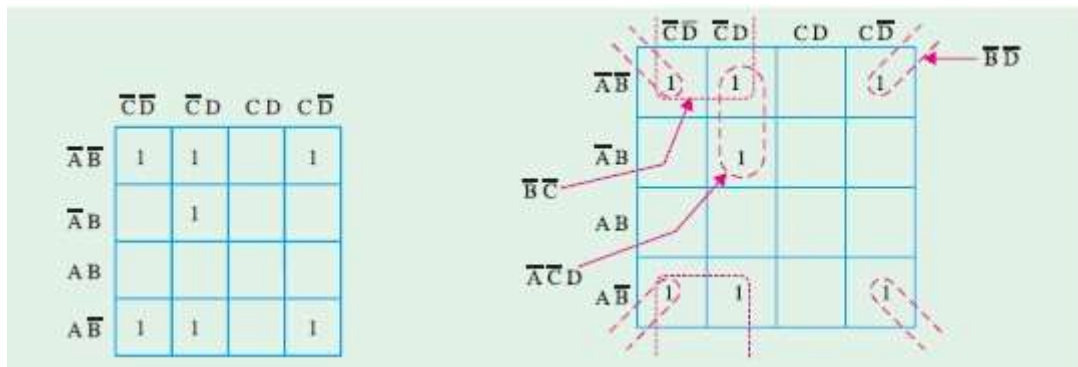


Fig 10

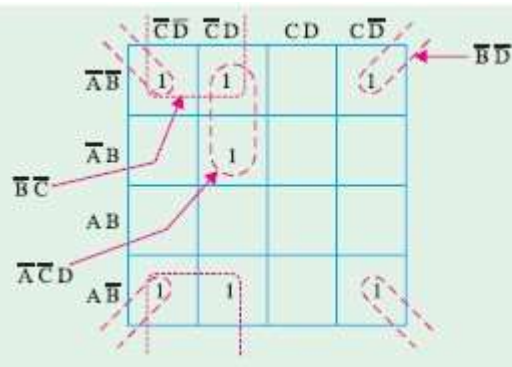


Fig 11

Solution. The grouping of 1s is as shown in the Fig. 11. Notice the “wrap around” four-square group that includes the 1s on four corners of the Karnaugh map. This group produces a product term $\overline{B} \overline{D}$. This is determined by observing that the group contains both A and $\overline{A}$ and C and $\overline{C}$, so these variables are eliminated.

Another group of four with “wrap-around” adjacency is formed from the top and the bottom rows of the Karnaugh map. This group overlaps with the previous group and produces a product term $\overline{B} \overline{C}$. This is determined by observing that this group contains both A and $\overline{A}$ and D and $\overline{D}$, so these variables are eliminated.

The remaining 1 is absorbed in a overlapping group of two squares. This group produces three-variable term $\overline{A} \overline{C} D$. This is determined by observing that this group contains both B and $\overline{B}$, so this variable is eliminated. This resulting simplified SOP function is the sum of the product terms $\overline{B} \overline{D} + \overline{B} \overline{C} + \overline{A} \overline{C} D$, i.e.

$$X = \overline{B} \overline{D} + \overline{B} \overline{C} + \overline{A} \overline{C} D$$

Combinational Logic Circuit

It is a circuit built from various logic gate combinations.

The circuit possesses a set of inputs, a memoryless logic network to operate on the inputs and a set of outputs as shown in Fig. 1 (a).

The output from a combinational logic circuit depends solely on the present input values and not on the previous ones. Moreover, output combinational networks are used to make logical decisions and control the operation of different circuits in digital electronic systems. For a given set of input conditions, the output of such a circuit is the same. Consequently, a truth table can fully describe the operation of such a circuit. Examples of such a circuit are: decoders, adders, multiplexer and de-multiplexers etc.

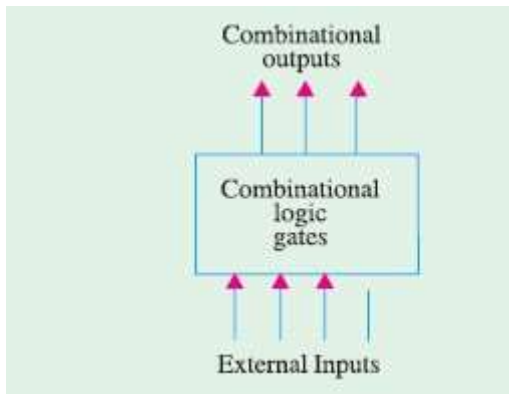


Fig. 1

Adders

The logical gates discussed so far can be used for performing arithmetical functions like addition, subtraction, multiplication and division in electronic calculators and digital instruments. In the central processing unit (CPU) of a computer, these arithmetic functions are carried out by the arithmetic and logic unit (ALU). The logic functions used generally are XOR, OR and AND. We will consider the following:

1. **Half Adder**—It carries out binary addition with the help of XOR and AND gates. It has two inputs and two outputs.
2. **Full Adder**—It has three inputs and can add three bits at a time. It is made up of two half adders and one OR gate.
It can add 2 binary digits at a time and produce a 2-bit data i.e. sum and carry according to the binary addition rules.

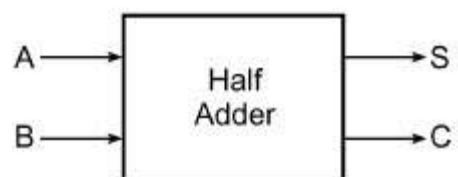
Half-Adder

A half-adder is an arithmetic circuit block that can be used to add two bits.

Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.

Figure below shows the truth table of a half-adder, showing all possible input combinations and the corresponding outputs.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



The Boolean expressions for the SUM and CARRY outputs are given by the equations

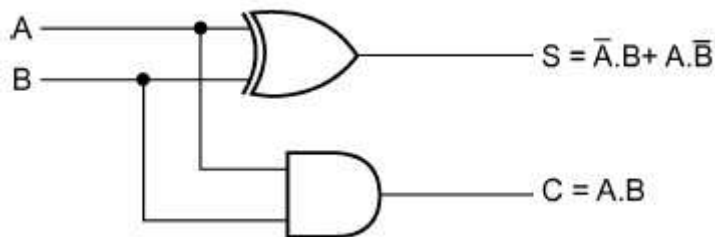
$$\text{SUM } S = A.\bar{B} + \bar{A}.B$$

$$\text{CARRY } C = A.B$$

An examination of the two expressions tells that there is no scope for further simplification. While the first one representing the SUM output is that of an EX-OR gate, the second one representing the CARRY output is that of an AND gate.

However, these two expressions can certainly be represented in different forms using various laws and theorems of Boolean algebra to illustrate the flexibility that the designer has in hardware implementing as simple a combinational function as that of a half-adder.

Although the simplest way to hardware-implement a half-adder would be to use a two-input EX-OR gate for the SUM output and a two-input AND gate for the CARRY output, as shown below

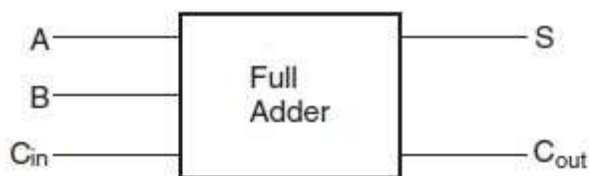


Full Adder

A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output.

Such a building block becomes a necessity when it comes to adding binary numbers with a large number of bits.

The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only.



A	B	C _{in}	SUM (S)	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure above shows the truth table of a full adder circuit showing all possible input combinations and corresponding outputs.

In order to arrive at the logic circuit for hardware implementation of a full adder, we will firstly write the Boolean expressions for the two output variables, that is, the SUM and CARRY outputs, in terms of input variables.

These expressions are then simplified by using any of the simplification techniques described previously.

The Boolean expressions for the two output variables are given in below for the SUM output (S) and for the CARRY output (C_{out}):

$$S = \overline{A}.\overline{B}.C_{in} + \overline{A}.B.\overline{C}_{in} + A.\overline{B}.\overline{C}_{in} + A.B.C_{in}$$

$$C_{out} = \overline{A}.B.C_{in} + A.\overline{B}.C_{in} + A.B.\overline{C}_{in} + A.B.C_{in}$$

The next step is to simplify the two expressions. We will do so with the help of the Karnaugh mapping technique.

AB \ C _{in}		$\overline{C}_{in}$	C_{in}
$\overline{A}\overline{B}$			1
$\overline{A}B$	1		
AB			1
$A\overline{B}$	1		

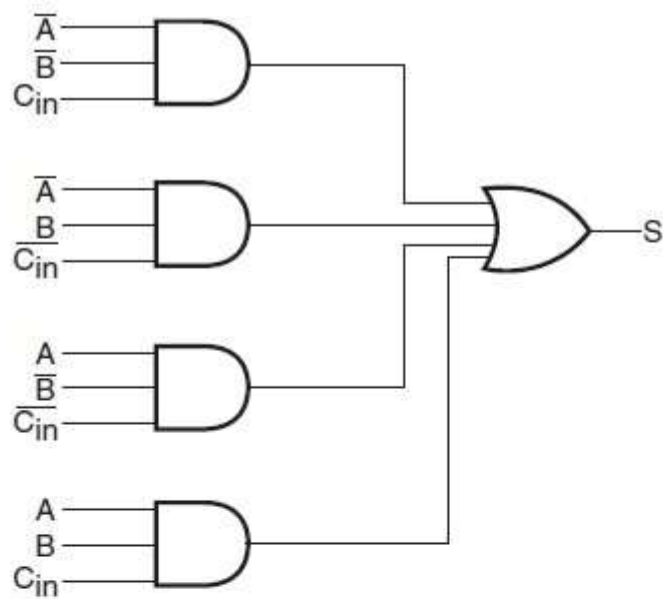
(a)

AB \ C _{in}		$\overline{C}_{in}$	C_{in}
$\overline{A}\overline{B}$			
$\overline{A}B$			1
AB	1		1
$A\overline{B}$			1

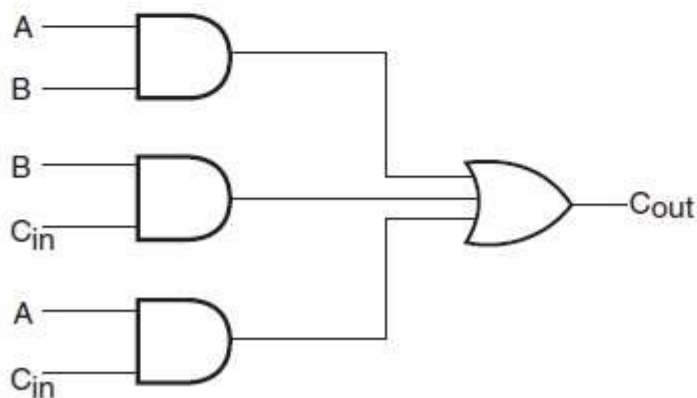
(b)

As is clear from the two maps, the expression for the SUM (S)output cannot be simplified any further, whereas the simplified Boolean expression for C_{out} is given by the equation

$$C_{out} = B.C_{in} + A.B + A.C_{in}$$



(a)



(b)

Figure above shows the logic circuit diagram of the full adder.

A full adder can also be seen to comprise two half-adders and an OR gate. The expressions for SUM and CARRY outputs can be rewritten as follows:

$$S = \overline{C}_{in} \cdot (\overline{A} \cdot B + A \cdot \overline{B}) + C_{in} \cdot (A \cdot B + \overline{A} \cdot \overline{B})$$

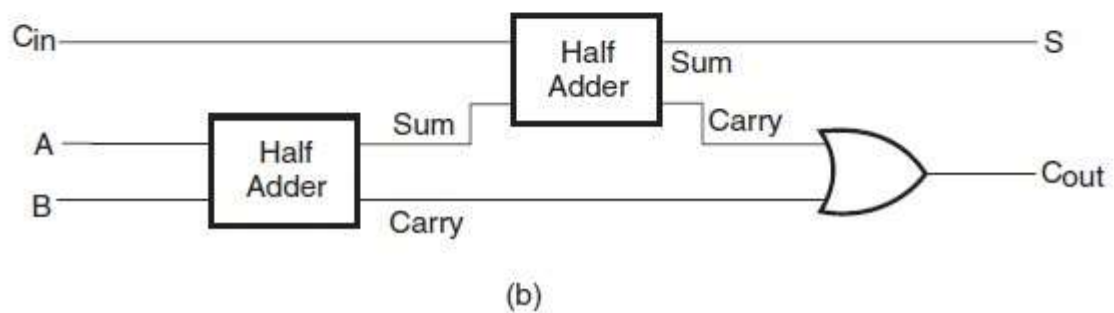
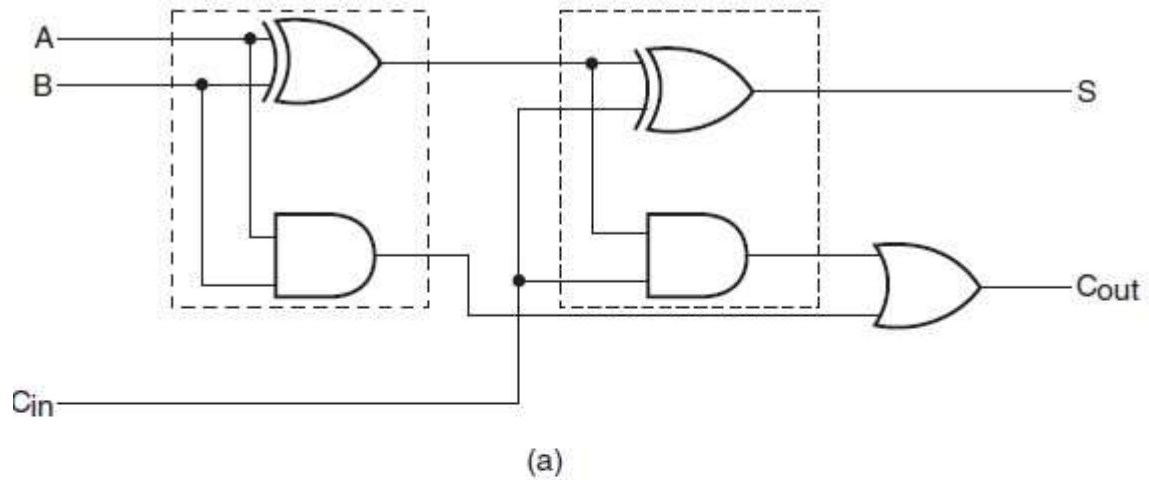
$$S = \overline{C}_{in} \cdot (\overline{A} \cdot B + A \cdot \overline{B}) + C_{in} \cdot (\overline{\overline{A} \cdot B + A \cdot \overline{B}})$$

Similarly, the expression for CARRY output can be rewritten as follows:

$$\begin{aligned} C_{out} &= B \cdot C_{in} \cdot (A + \overline{A}) + A \cdot B + A \cdot C_{in} \cdot (B + \overline{B}) \\ &= A \cdot B + A \cdot B \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot C_{in} = A \cdot B + A \cdot B \cdot C_{in} + \overline{A} \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot C_{in} \\ &= A \cdot B \cdot (1 + C_{in}) + C_{in} \cdot (\overline{A} \cdot B + A \cdot \overline{B}) \end{aligned}$$

$$C_{out} = A.B + C_{in}.(\bar{A}.B + A.\bar{B})$$

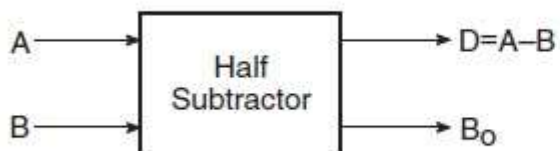
5



Logic implementation of a full adder with half-adders.

Half-Subtractor

A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The truth table of a half-subtractor, as shown in Figure below.



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

The Boolean expressions for the two outputs are given by the equations

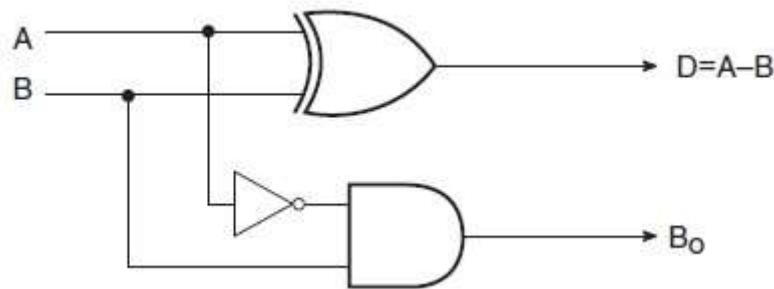
$$D = \bar{A}.B + A.\bar{B}$$

$$B_0 = \bar{A}.B$$

It is obvious that there is no further scope for any simplification of the Boolean expressions given by Equations.

While the expression for the DIFFERENCE (D) output is that of an EX-OR gate, the expression for the BORROW output (B_o) is that of an AND gate with input A complemented before it is fed to the gate.

Figure below shows the logic implementation of a half-subtractor.



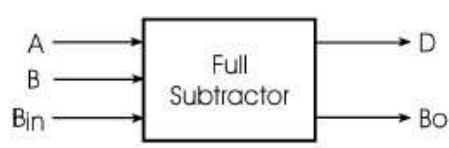
Full Subtractor

A **full subtractor** performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a '1' has already been borrowed by the previous adjacent lower minuend bit or not.

As a result, there are three bits to be handled at the input of a full-subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in} .

There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o . The BORROW output bit tells whether the minuend bit needs to borrow a '1' from the next possible higher minuend bit.

Figure below shows the truth table of a full subtractor.



Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_o)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The Boolean expressions for the two output variables are given by the equations

$$D = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + A.\overline{B}.\overline{B}_{in} + A.B.B_{in}$$

$$B_o = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + \overline{A}.B.B_{in} + A.B.B_{in}$$

The Karnaugh maps for the two expressions are given in Figure below for DIFFERENCE output D and for BORROW output B_o .

AB \ B_{in}	$\overline{B_{in}}$	B_{in}
$\overline{A}\overline{B}$		1
$\overline{A}B$	1	
AB		1
$A\overline{B}$	1	

(a)

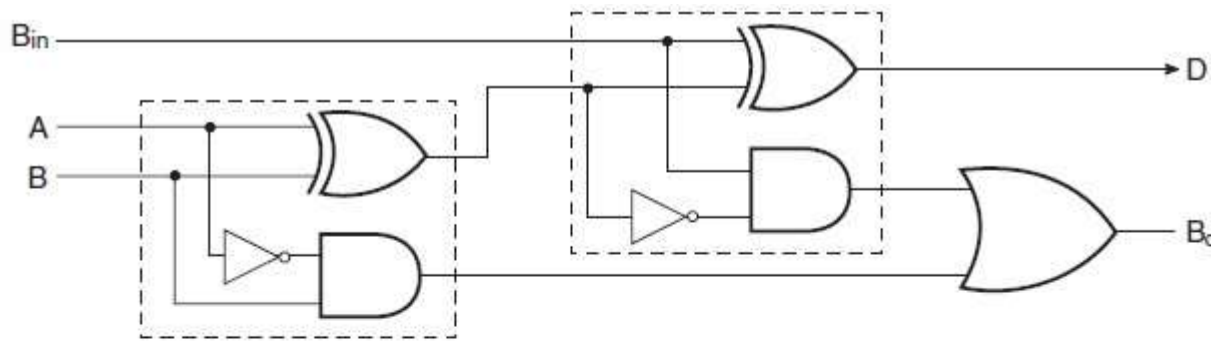
AB \ B_{in}	$\overline{B_{in}}$	B_{in}
$\overline{A}\overline{B}$		1
$\overline{A}B$	1	1
AB		1
$A\overline{B}$		

(b)

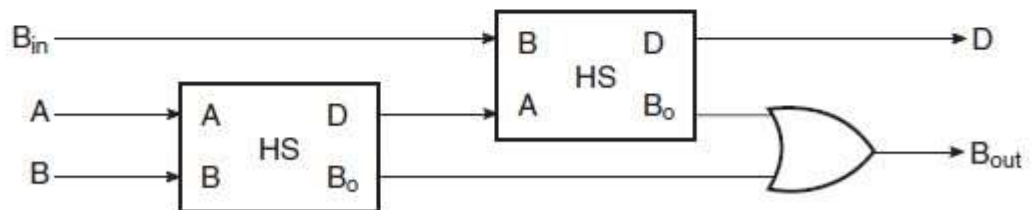
As is clear from the two Karnaugh maps, no simplification is possible for the difference output D. The simplified expression for B_0 is given by the equation

$$B_0 = \overline{A}.B + \overline{A}.B_{in} + B.B_{in}$$

It can be shown that a full subtractor can be implemented with half-subtractors in the same way as a full adder was constructed using half-adders.



(a)



(b)

A Multiplexer (MUX)

A MUX is a digital switch that has multiple inputs (sources) and a single output (destination). The select lines determine which input is connected to the output.

MUX Types

- 2-to-1 (1 select line)
- 4-to-1 (2 select lines)
- 8-to-1 (3 select lines)
- 16-to-1 (4 select lines)

Consider an integer 'm', which is constrained by the following relation:

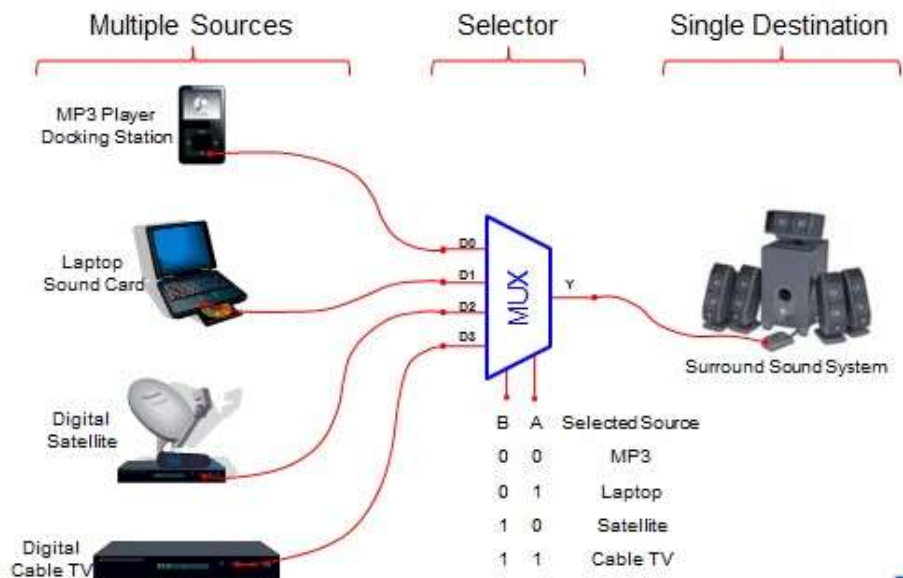
$$m = 2^n, \text{ where } m \text{ and } n \text{ are both integers.}$$

A m-to-1 Multiplexer has

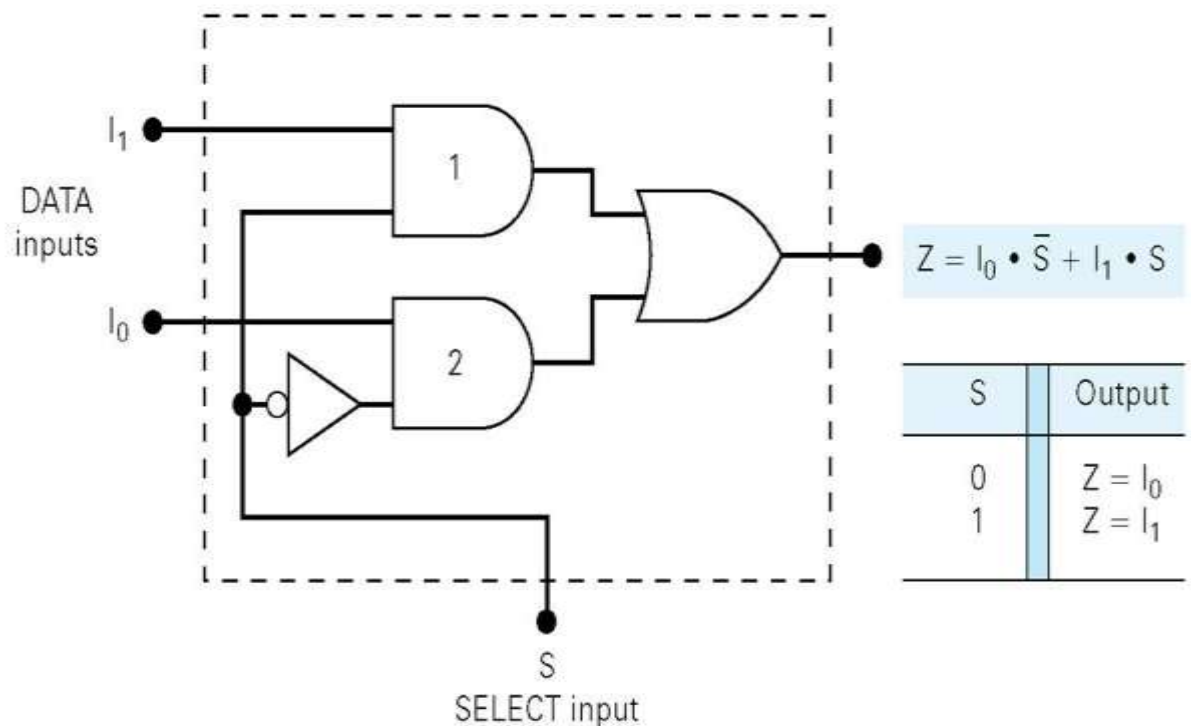
- m Inputs: $I_0, I_1, I_2, \dots, I_{(m-1)}$
- one Output: Z
- n Control inputs: $S_0, S_1, S_2, \dots, S_{(n-1)}$
- One (or more) Enable input(s)

such that Z may be equal to one of the inputs, depending upon the control inputs.

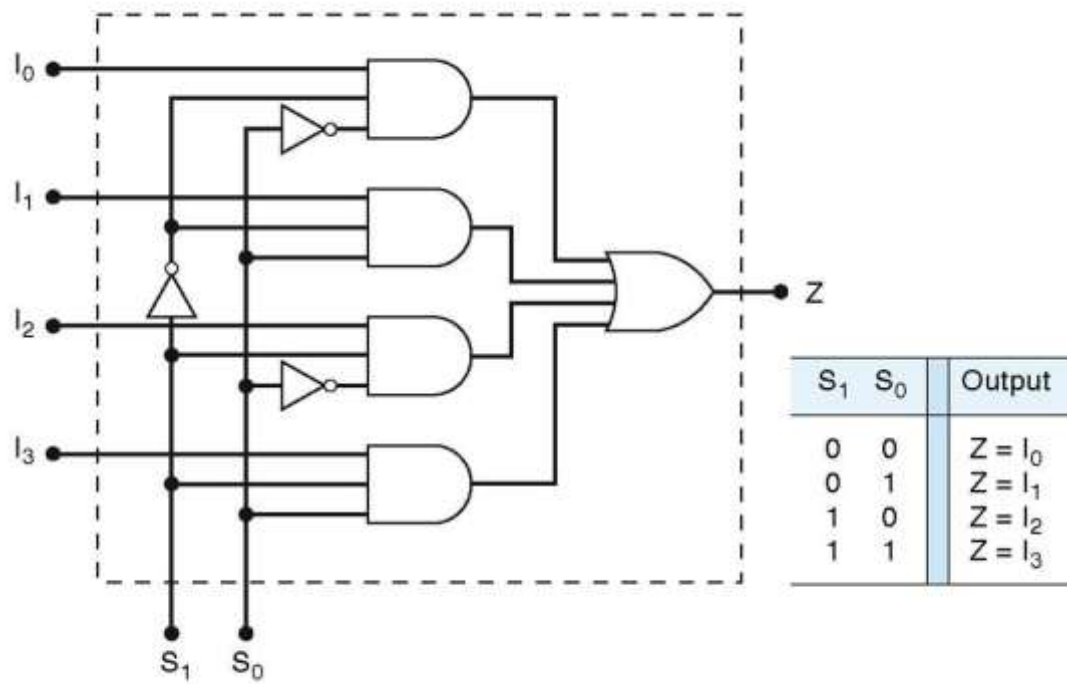
Typical Application of a MUX



Two-input multiplexer



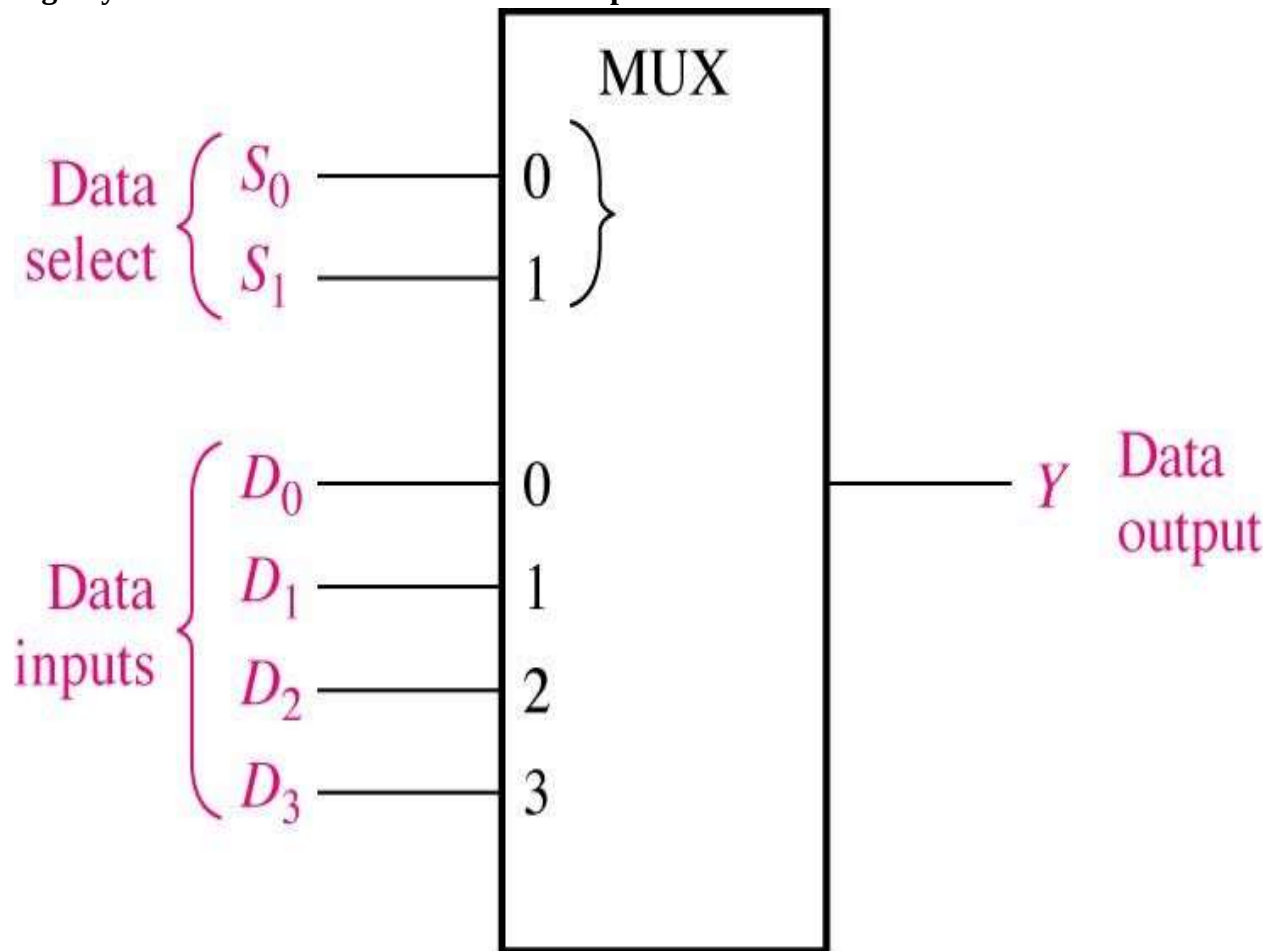
Four-input multiplexer



an 4-to-1 multiplexer can be represented by the Boolean function

$$Z = I_0 \cdot \overline{S_1} \cdot \overline{S_0} + I_1 \cdot \overline{S_1} \cdot S_0 + I_2 \cdot S_1 \cdot \overline{S_0} + I_3 \cdot S_1 \cdot S_0$$

Logic symbol for a 1-of-4 data selector/multiplexer



Implementing Boolean Functions with Multiplexers

One of the most common applications of a multiplexer is its use for implementation of combinational logic Boolean functions. **The simplest technique for doing so is to employ a 2^n -to-1 MUX to implement an n -variable Boolean function.** The input lines corresponding to each of the minterms present in the Boolean function are made equal to logic '1' state. The remaining minterms that are absent in the Boolean function are disabled by making their corresponding input lines equal to logic '0'.

Example implement the Boolean function given by the equation

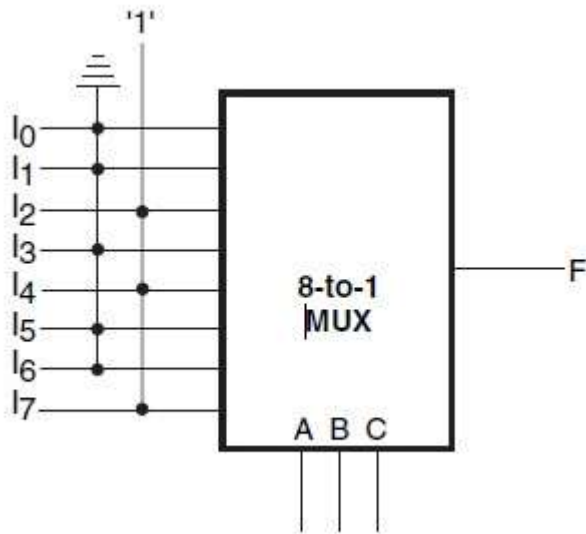
$$f(A, B, C) = \sum 2, 4, 7$$

In terms of variables A, B and C, equation above can be written as follows:

$$f(A, B, C) = \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + A.B.C$$

As shown in Fig. below, the input lines corresponding to the three minterms present in the given

Boolean function are tied to logic '1'. The remaining five possible minterms absent in the Boolean function are tied to logic '0'.



It is a three-variable Boolean function. Conventionally, we will need to use an 8-to-1 multiplexer to implement this function.

The truth table of the given Boolean function is shown in Table below

Minterm	A	B	C	$f(A,B,C)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

d) Demultiplexer:

The demultiplexer is the inverse of the multiplexer, in that it takes a data input and n address inputs.

It has 2^n outputs. The address input determine which data output is going to have the same value as the data input.

The other data outputs will have the value 0.

A demultiplexer has

- N control inputs

- 1 data input
- 2^N outputs

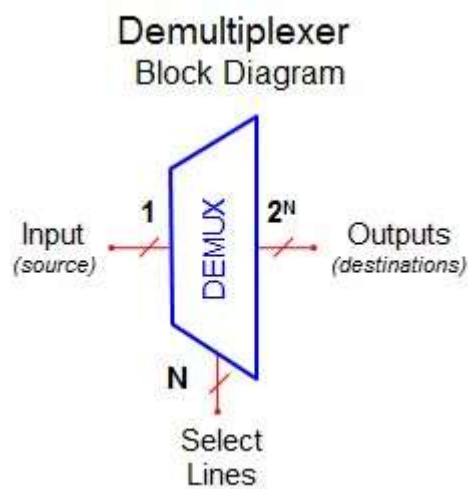
A demultiplexer routes (or connects) the data input to the selected output.

The value of the control inputs determines the output that is selected.

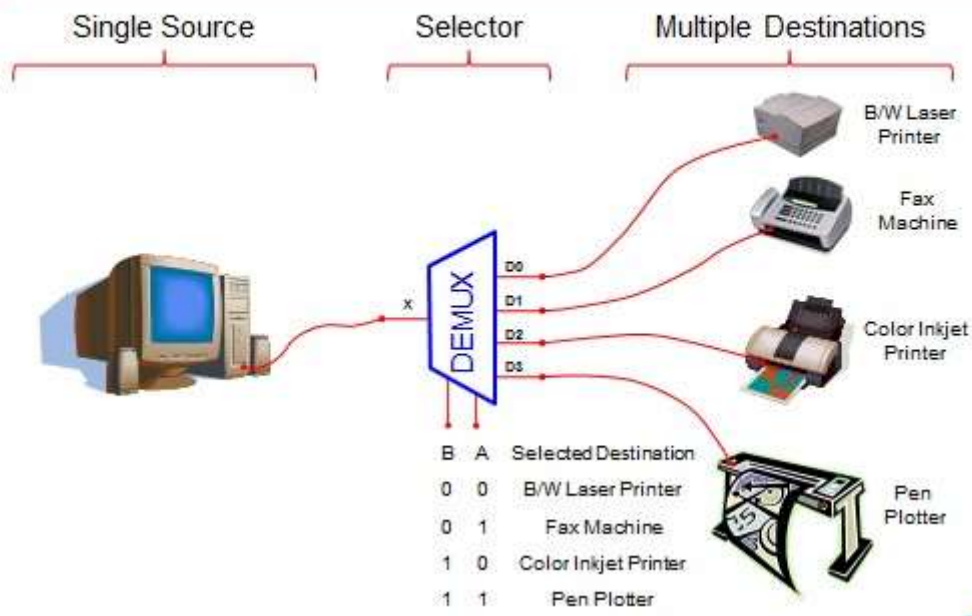
A demultiplexer performs the opposite function of a multiplexer.

DEMUX Types

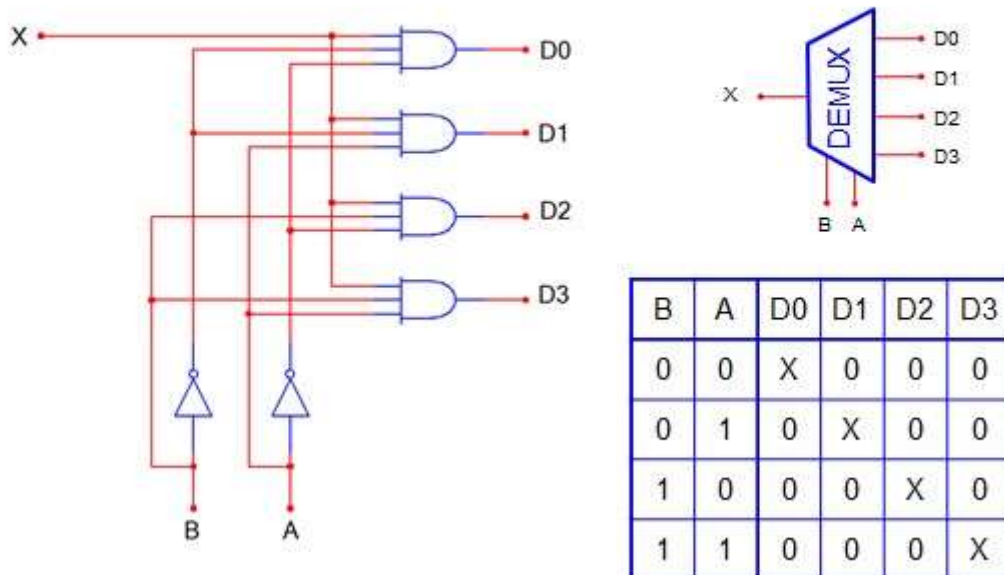
- 1-to-2 (1 select line)
- 1-to-4 (2 select lines)
- 1-to-8 (3 select lines)
- 1-to-16 (4 select lines)



Typical Application of a DEMUX



1-to-4 De-Multiplexer (DEMUX)



Note that i/p X is held High (1)

Decoder

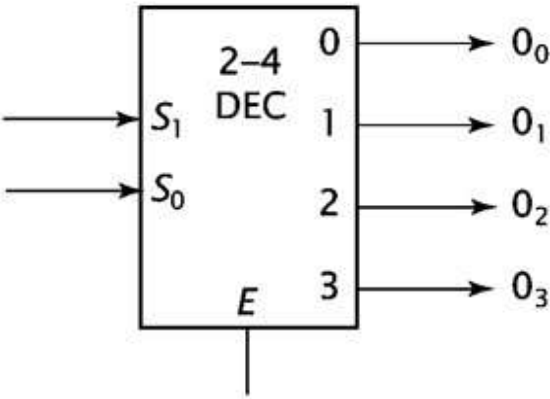
The basic function of a decoder is to detect the presence of a specified combination of bits on its inputs and to indicate that presence by a specified output level.

- It consists of:
- Inputs (n)
- Outputs (2^n , numbered from 0 to $2^n - 1$)
- Selectors / Enable (active high or active low)

The truth table of 2-to-4 Decoder

S_1	S_0	E	O_0	O_1	O_2	O_3
X	X	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

2-to-4 Decoder



S_1	S_0	E	O_0	O_1	O_2	O_3
X	X	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

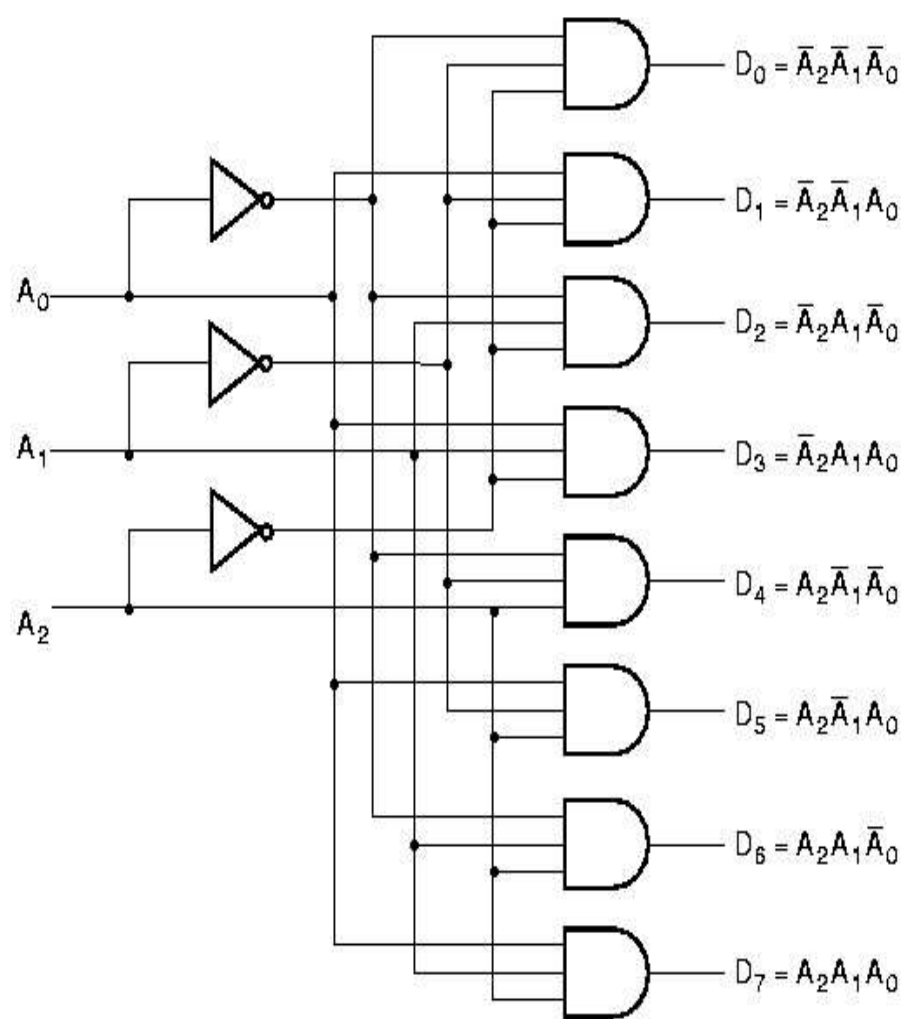
(b)



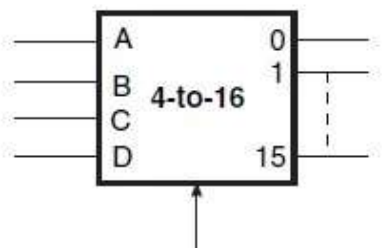
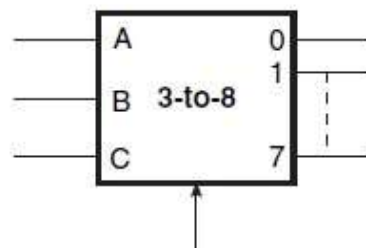
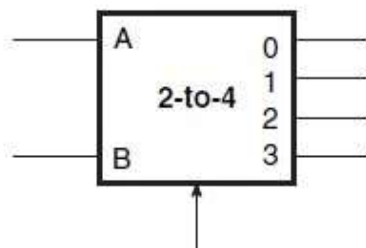
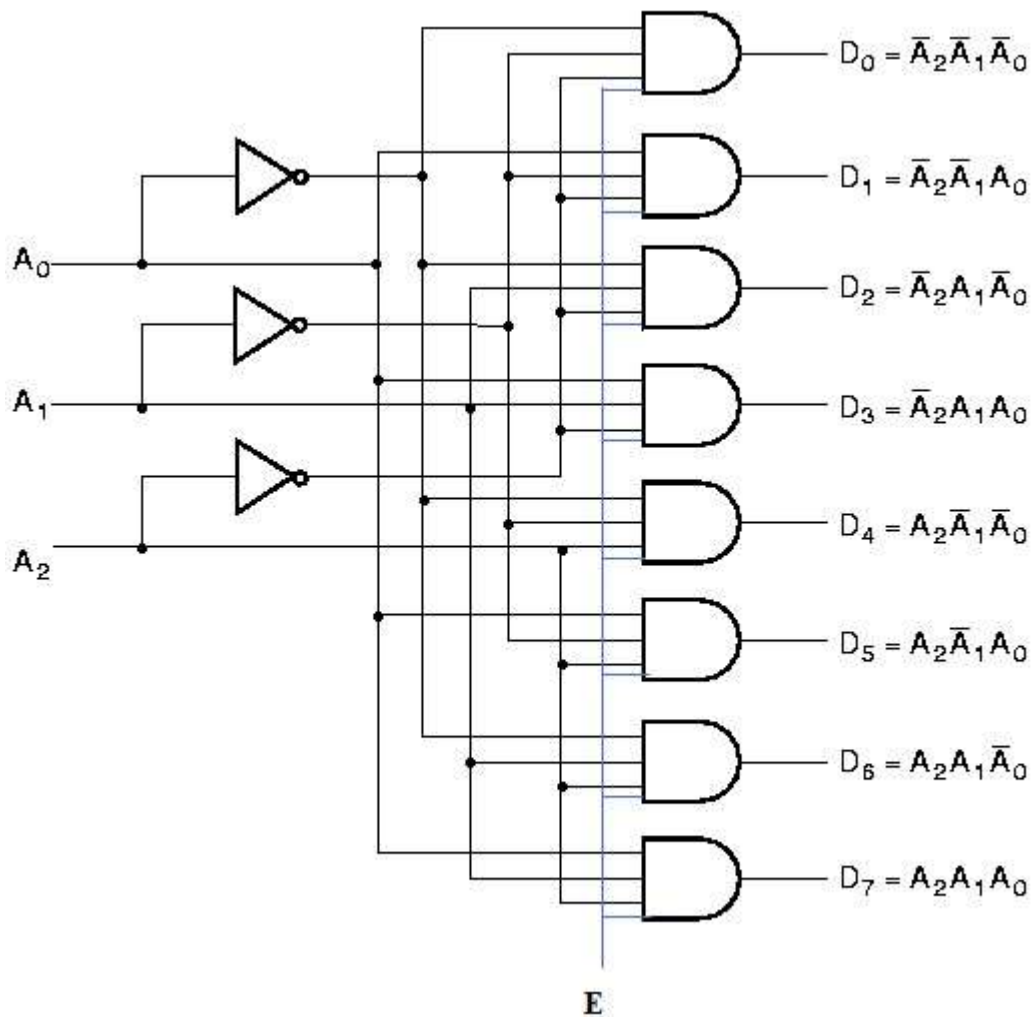
The truth table of 3-to-8 Decoder

A	A	A	D	D	D	D	D	D	D	D
0	0	0	1							

0	0	1		1						
0	1	0			1					
0	1	1				1				
1	0	0					1			
1	0	1						1		
1	1	0							1	
1	1	1								1



3-to-8 Decoder with Enable



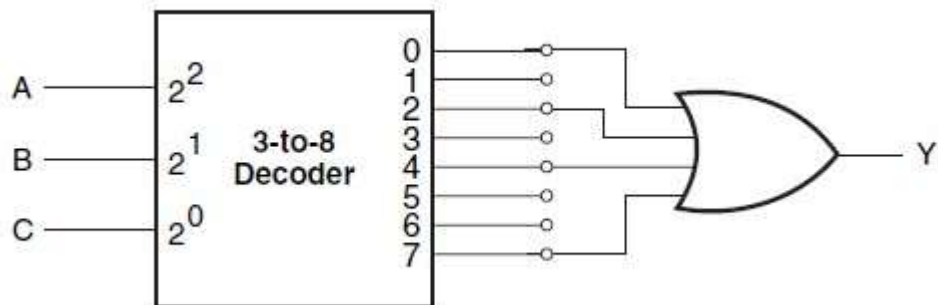
If there are some unused or 'don't care' combinations in the n -bit code, then there will be fewer than 2^n output lines. As an illustration, if there are three input lines, it can have a maximum of eight unique output lines. If, in the three-bit input code, the only used three-bit combinations are 000, 001, 010, 100, 110 and 111 (011 and 101 being either unused or don't care combinations), then this decoder will have only six output lines. In general, if n and m are respectively the numbers of input and output lines, then $m \leq 2^n$.

A decoder can generate a maximum of 2^n possible minterms with an n -bit binary code.

Implementing Boolean Functions with Decoders

Given Boolean function implement it using a Decoder

$$Y = A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.B.C + \overline{A}.\overline{B}.C$$



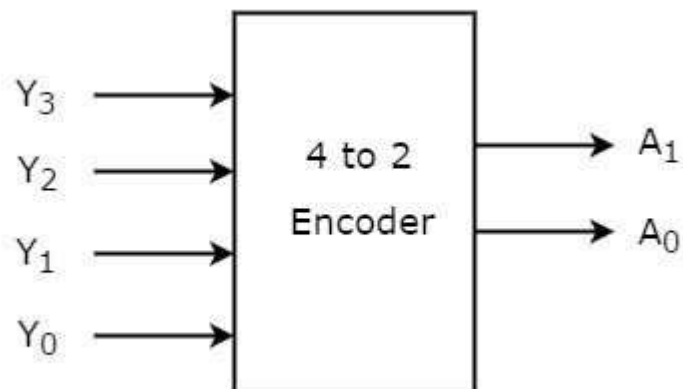
Encoder

An encoder is the combinational circuit which performs a reverse function of decoder. Encode inputs are decimal digits and/or alphabetic characters and outputs are coded representation of these inputs.

- Perform the inverse operation of a decoder
- Inputs (2ⁿ) or less
- Outputs (n output lines)

4 to 2 Encoder

Let 4 to 2 Encoder has four inputs Y₃, Y₂, Y₁ & Y₀ and two outputs A₁ & A₀. The **block diagram** of 4 to 2 Encoder is shown in the following figure.



At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The **Truth table** of 4 to 2 encoder is shown below.

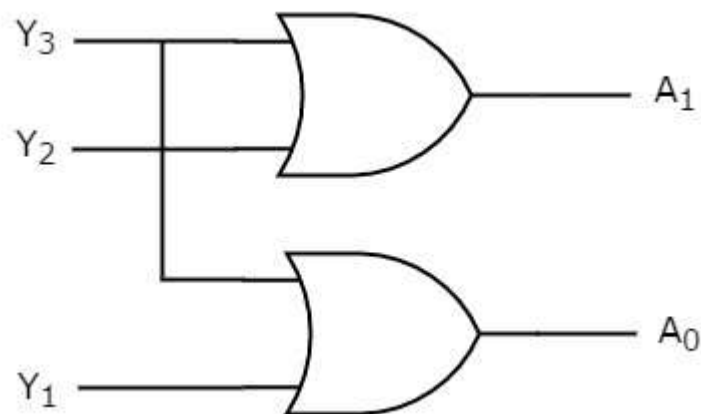
Inputs				Outputs	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

From Truth table, we can write the **Boolean functions** for each output as

$$A_1 = Y_3 + Y_2$$

$$A_0 = Y_3 + Y_1$$

We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



The above circuit diagram contains two OR gates. These OR gates encode the four inputs with two bits

SEQUENTIAL LOGIC CIRCUITS

Introduction

So far, we have dealt with combinational logic circuits, whose outputs are functions of the current inputs only.

Here we will deal with sequential logic circuits.

A **sequential logic circuit** is a circuit whose present outputs are functions of the present inputs, as well as previous inputs.

It has in it a unit called the memory which stores the effect of the previous sequence of inputs. The stored effects of the previous inputs are called the STATE of the circuit. A sequential circuit can be represented by the block diagram on Figure 1.

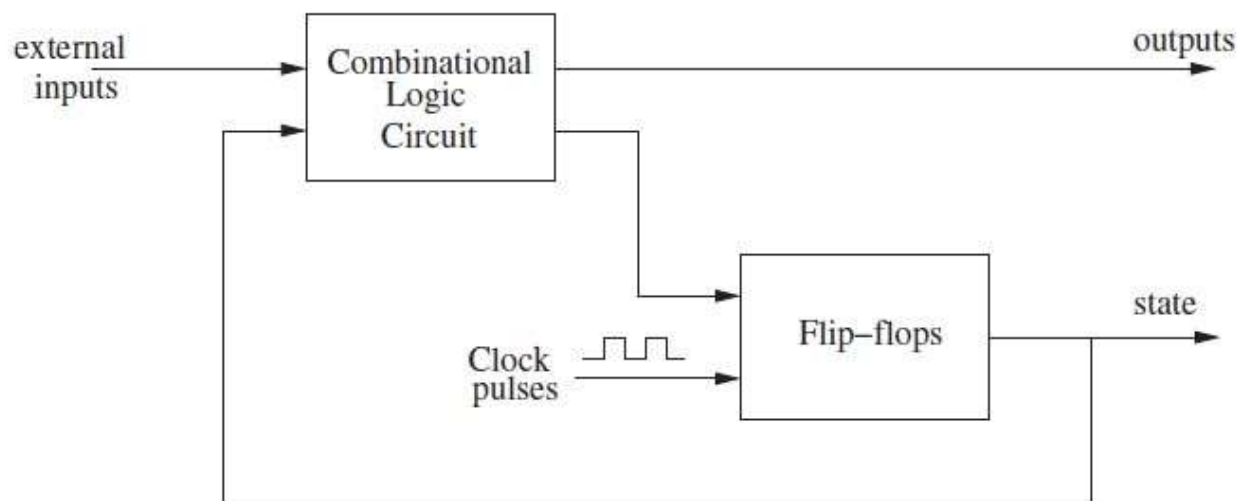


Figure 1: Block diagram of a sequential logic circuit

Flip-flops are the basic building blocks for sequential logic circuits.

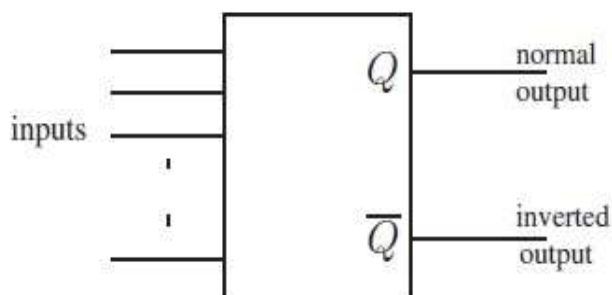
Flip-Flops

A **flip-flop** is a logic circuit that is capable of storing one bit of information (0 or 1).

It stores the one bit of information as long as power is supplied to the circuit. It is the simplest memory element.

It is also referred to as a **bistable multivibrator**.

The block diagram of a flip-flop is shown on Figure 2.



A Flip-flop can have one or more inputs, but it has only two outputs, the normal output Q and the inverted (or complemented) output $\bar{Q}$.

Under normal operating conditions Q and $\bar{Q}$ are always complements of each other, i.e. either $Q = 0$ and $\bar{Q} = 1$, or $Q = 1$ and $\bar{Q} = 0$.

When we refer to the STATE of a flip-flop, we are referring to the state of its normal (Q) output i.e. if we say that the state of a flip-flop is 1, we mean $Q = 1$.

The SET-RESET flip-flop

This is illustrated on Figure below.

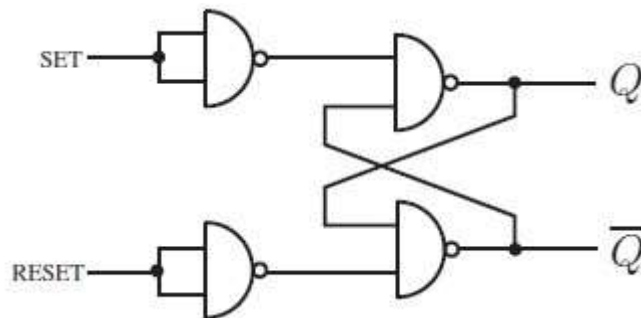


Figure: SET-RESET (SR) flip-flop

The truth-table for this flip-flop is shown below:

SET	RESET	Q_{n+1}
0	0	Q_n - 'remembers' previous state (Memory State)
0	1	0 - Flip-Flop RESET
1	0	1 - Flip-Flop SET
1	1	Disallowed as it causes $Q = \bar{Q} = 1$

When we have $SET = 0$ and $RESET = 1$, the flip-flop is RESET ($Q = 0$), and when $SET = 1$ and $RESET = 0$, the flip-flop is SET ($Q = 1$). If we have $SET = 1$ and $RESET = 1$, it is similar to setting and resetting the flip-flop at the same time, and this mode is disallowed as it causes $Q = \bar{Q} = 1$.

The D Flip-Flop

The D-type flip-flop is illustrated in Figure below.

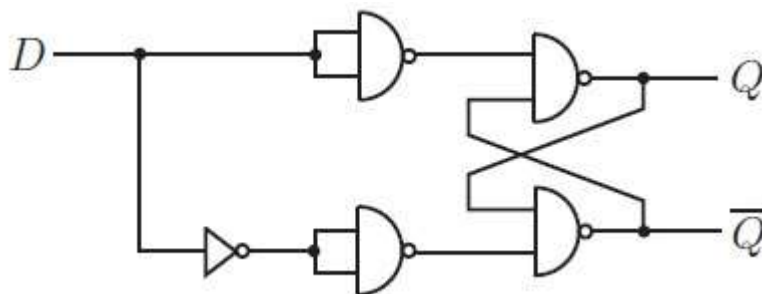


Figure : D flip-flop

It is a modified version of the SET-RESET flip-flop where $\overline{SET} = RESET$.

It has only one input, D.

The truth-table for this flip-flop is shown below:

D	Q_{n+1}
0	0
1	1

From the truth-table, we can see that Q transparently follows the input D, and for this reason, the D flip-flop is sometimes referred to as a transparent latch.

The T flip-flop

A T-flip-flop (also known as a toggle flip-flop) is illustrated on Figure below.

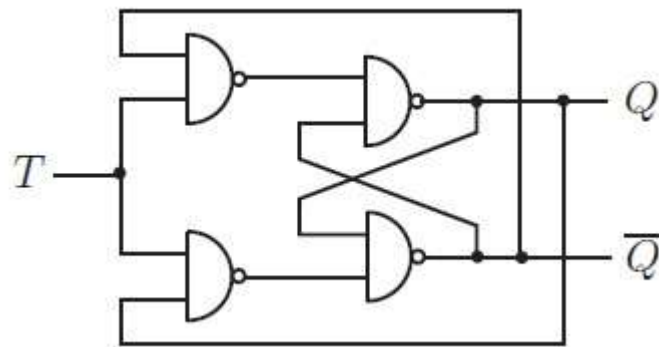


Figure: T flip-flop

The corresponding truth-table is shown below:

T	Q_{n+1}
0	Q_n
1	$\bar{Q}_n$

The JK flip-flop

The JK flip-flop is shown on Figure below.

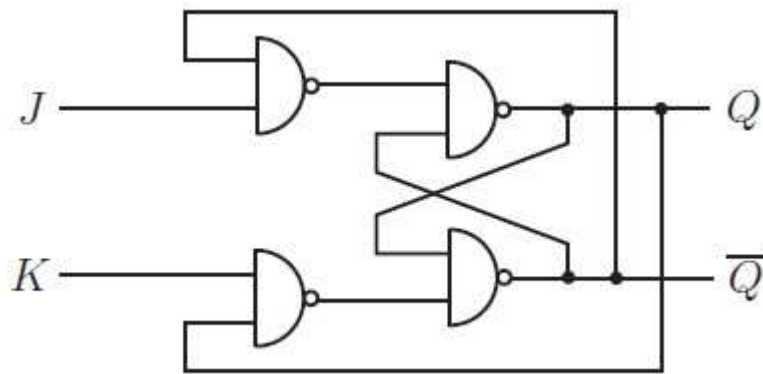


Figure: JK flip-flop

Its truth-table is shown below:

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

This truth-table can be summarized as shown below:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\bar{Q}_n$

From the truth-table above, we can see that the JK flip-flop does not have invalid inputs and it can complement. These properties make the JK flip-flop very versatile, and it is used in many sequential logic circuits.

Types of clocked flip-flops

There are three types of clocked flip-flops:

- Level Driven flip-flops
- Master-Slave flip-flops
- Edge-triggered flip-flops

The first two types are no longer used in modern digital systems, but are mentioned here for completeness.

Edge triggered flip-flops

Edge triggered flip-flops are those that only change state during the clock transitions (HIGH to LOW or LOW to HIGH).

The flip-flops that change state during the HIGH to LOW transitions of the clock (negative going transitions) are known as negative edge triggered flip-flops, while those that change state at the LOW to HIGH transitions of the clock are known as positive edge triggered flip-flops.

Note that there are no flip-flops that trigger on both the positive and the negative going transitions of the clock.

Figure below shows the logic symbol for a positive edge triggered JK flip-flop.

The corresponding truth-table is shown below:

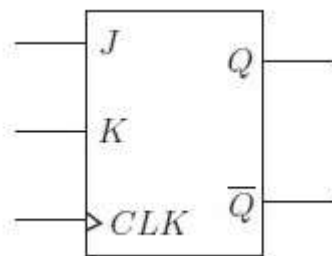


Figure: Symbol of a positive-edge-triggered JK flip-flop

J	K	CLK	Q_{n+1}
0	0	↑	Q_n
0	1	↑	0
1	0	↑	1
1	1	↑	$\bar{Q}_n$

The interpretation of the truth-table is as follows:

If the inputs of the flip-flop are $J = 0$, $K = 0$ and a Positive Going Transition (PGT) of the clock occurs, the flip-flop will remain in its present state (Q_n), while if the inputs are $J = 0$, $K = 1$, then a PGT of the clock will make the output to be LOW (0) e.t.c.

Figure below shows the logic symbol for a negative edge triggered JK flip-flop.

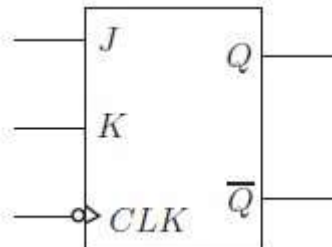


Figure: Symbol of a negative-edge-triggered JK flip-flop

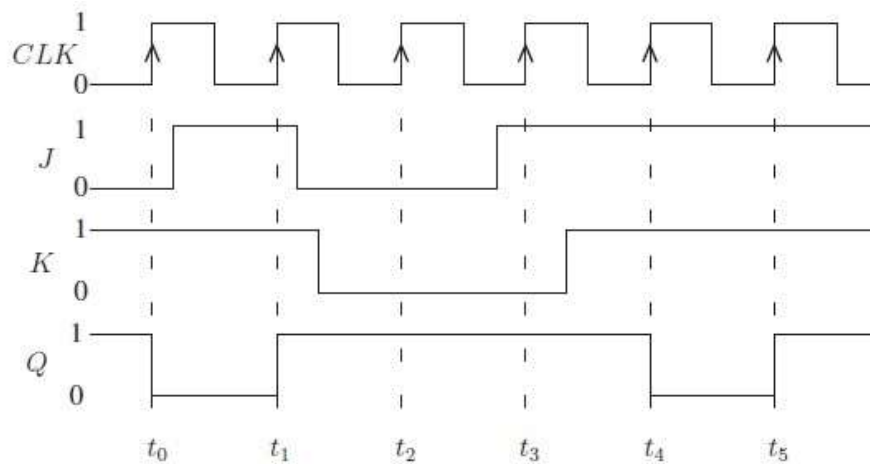
The corresponding truth-table is shown below:

J	K	CLK	Q_{n+1}
0	0	↓	Q_n
0	1	↓	0
1	0	↓	1
1	1	↓	$\bar{Q}_n$

The interpretation of the truth-table is similar to that of the positive edge triggered flip-flop, the only difference is that changes in the state occur only on the negative going transitions of the clock.

Exercise

Figure below shows a clock waveform and inputs J and K applied to a positive-edge triggered JK flip-flop. Using the truth-table for a positive edge triggered JK flip-flop, explain what happens at t_0 , t_1 , t_2 , t_3 , t_4 and t_5 to justify the output waveform shown for Q. (It is assumed that the initial of Q is HIGH)



Flip-flop excitation tables

SET-RESET flip-flop

Q_n	Q_{n+1}	SET	RESET
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

JK flip-flop

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D flip-flop

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

T flip-flop

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Counters

A **counter** is a circuit made up of a series of flip-flops connected in a suitable manner to record sequences of pulses presented to it in digital form. **Counters are used in digital systems to:**

1. **Count events** - convert a given number of event cycles into a prescribed binary code.
2. **To time events** - The duration of an event can be determined from the count and the clock frequency.
3. **To generate special code sequences** - In this application, the counter is used as the heart of a logic sequencer. Such a circuit generates the next-state information as well as the control information.
4. **Frequency division** – They produce an output whose frequency is the input clock frequency divided by the modulus of the counter (for example, each stage in a binary counter halves the input frequency).

Classification of counters

1. Mode of operation

- **Single mode counter** - Counters without external control inputs.

■ **Multi-mode counter** - Counters with external control inputs that can be used to alter the counting sequence, the beginning of a sequence or the end of a sequence.

2. Modulus of the counter

The *modulus* (MOD number) of a counter is the number of different logic states it goes through before it comes back to the initial state to repeat the count sequence.

An n-bit counter that counts through all its natural states and does not skip any of the states has a modulus of 2^n . We can see that such counters have a modulus that is an integral power of 2, that is, 2, 4, 8, 16 and so on.

These can be modified with the help of additional combinational logic to get a modulus of less than 2^n .

To determine the number of flip-flops required to build a counter having a given modulus, identify the smallest integer m that is either equal to or greater than the desired modulus and is also equal to an integral power of 2. For instance, if the desired modulus is 10, which is the case in a decade counter, the smallest integer greater than or equal to 10 and which is also an integral power of 2 is 16. The number of flip-flops in this case would be 4, as $16 = 2^4$. On the same lines, the number of flip-flops required to construct counters with MOD numbers of 3, 6, 14, 28 and 63 would be 2, 3, 4, 5 and 6 respectively. In general, the arrangement of a minimum number of n flip-flops can be used to construct any counter with a modulus is given by the equation

$$2^{n-1} + 1 \leq \text{modulus} \leq 2^n$$

A modulus N counter is also referred to as a MOD N counter or a DIVIDE by N counter.

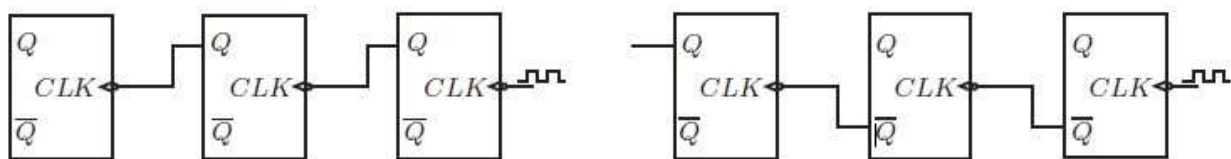
Generally speaking, a counter that can sequence through N states is known as a MODULO N counter or a DIVIDE by N counter.

If N is a power of 2 (2, 4, 8, 16 e.t.c.), the counter is called a **binary counter**.

If N is a power of 10 (10, 100, 1000 e.t.c.), the **counter is known as a decimal or a decade counter**.

3. Clocking method - There are two types of clocking for counters:

- i. **Asynchronous/ripple/serial counter** - Type of counter where each flip-flop serves as the clock input signal for the next flip-flop in the chain. Examples are shown on Figure below.



Asynchronous/ripple/serial counters

- ii. **Synchronous/parallel counter**, - Counter in which all the flip-flops are clocked simultaneously. This is illustrated on Figure below.

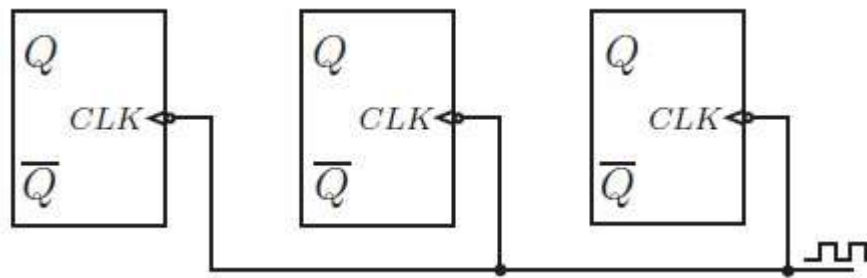


Figure: Synchronous counter

4. **Code sequence generated** - A counter may be a straight forward up or down counter (a counter generating codes $00 \rightarrow 01 \rightarrow 10 \rightarrow 11$ is an up counter, while one generating codes $11 \rightarrow 10 \rightarrow 01 \rightarrow 00$ is a down counter).

Assignment

5. Other counters include Gray Code counters and the shift register counters (ring counter and Johnson counters).

Counters designed using a combination of flip-flops and gates

Asynchronous Counters

An asynchronous counter comprises a chain of flip-flops in which one flip-flop is under the command of an external clock input.

The other flip-flops in the chain are indirectly controlled by the external clock.

The flip-flops are connected such that each flip-flop generates the clock input to the next flip-flop in the chain.

An asynchronous counter is also referred to as a **ripple counter**. This is because the effect of the external clock ripples through the chain of flip-flops. **Example**

Consider the circuit shown on Figure below.

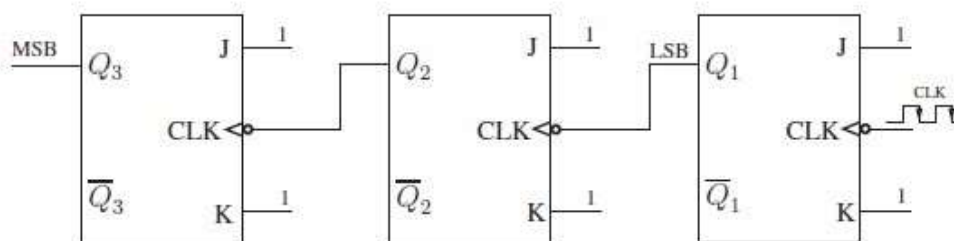
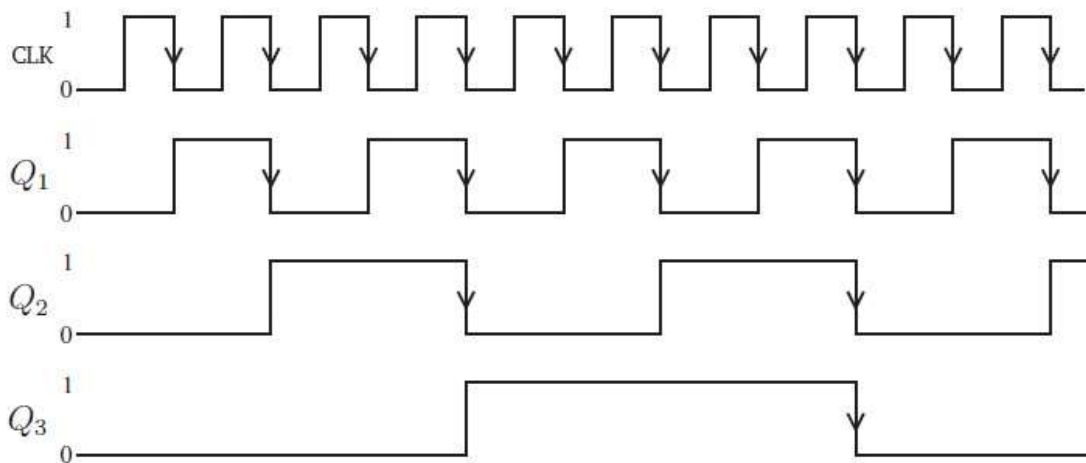


Figure: A ripple counter

Assuming that we start with $Q_3 = Q_2 = Q_1 = 0$, we can draw the waveforms shown on Figure below.



The truth-table corresponding to these waveforms is shown below:

Q_3	Q_2	Q_1	clock pulses
0	0	0	before applying clock pulses
0	0	1	after clock pulse #1
0	1	0	after clock pulse #2
0	1	1	after clock pulse #3
1	0	0	after clock pulse #4
1	0	1	after clock pulse #5
1	1	0	after clock pulse #6
1	1	1	after clock pulse #7
0	0	0	recycles to 000 after clock pulse #8
0	0	1	after clock pulse #9

From the table above, we can see that the counter has eight distinct states and it is an up counter. The counter is therefore a modulo-8 up asynchronous (ripple) counter.

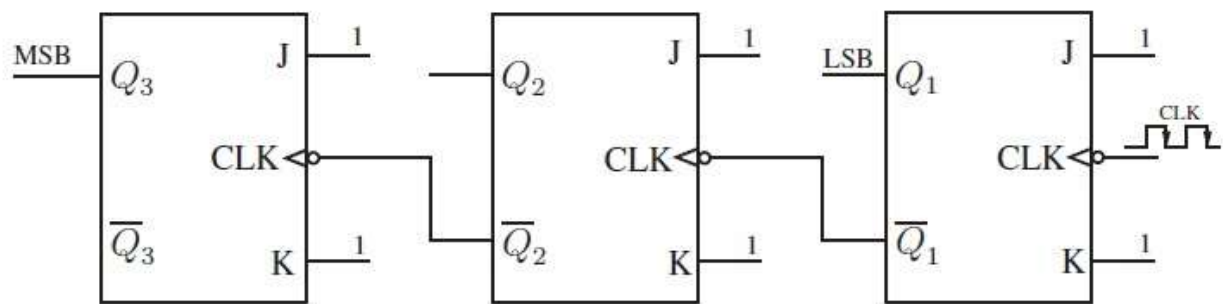
Note that the frequency of the waveform for Q3 is 1/8 of the clock frequency hence this can also be referred to as a divide by 8 counter.

Generally, for N negative-edge triggered JK flip-flops connected in the form shown on Figure above of a ripple counter, the counter formed is a Modulo- 2^N ripple up counter also known as a Divide by 2^N ripple up counter.

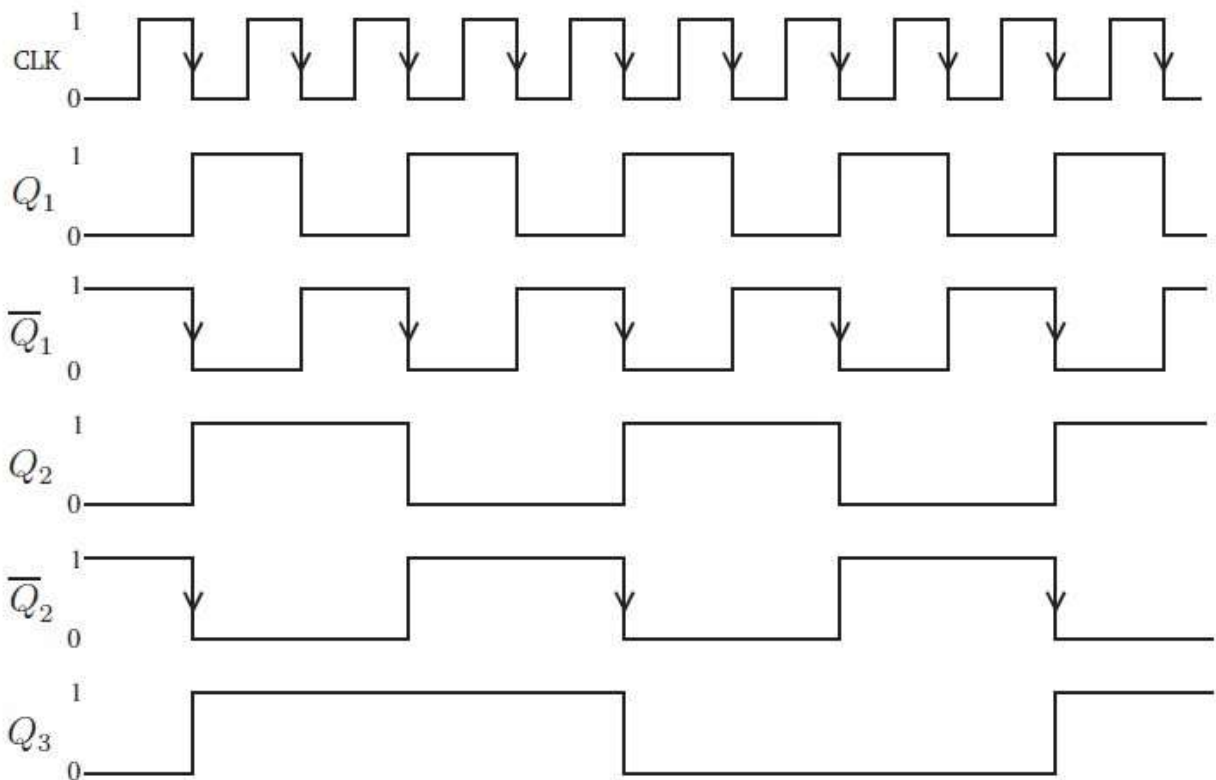
For any ripple counter, the output from the last flip-flop in the chain divides the input clock frequency by the MOD number of the counter.

Example

Consider the circuit shown on Figure below.



Assuming that we start with $Q_3 = Q_2 = Q_1 = 0$, we can draw the waveforms shown on Figure below.



We then get the truth-table for Q_1 , Q_2 and Q_3 , which is shown below:

Q_3	Q_2	Q_1	clock pulses
0	0	0	before applying clock pulses
1	1	1	after clock pulse #1
1	1	0	after clock pulse #2
1	0	1	after clock pulse #3
1	0	0	after clock pulse #4
0	1	1	after clock pulse #5
0	1	0	after clock pulse #6
0	0	1	after clock pulse #7
0	0	0	recycles to 000 after clock pulse #8
1	1	1	after clock pulse #9

From the table above, we can see that the counter has eight distinct states and it is a down counter. The counter is therefore a modulo-8 down asynchronous (ripple) counter or a divide by 8 down asynchronous counter.

Asynchronous Counters with MOD numbers $< 2^N$

The general design procedure is as follows:

1. For a modulo-k counter, find the smallest number of flip-flops N such that $2^{N-1} < k < 2^N$, and connect them as a ripple counter.
2. Connect a NAND gate output to the asynchronous CLEAR inputs of all the flip flops.
3. Determine which flip-flops will be in the HIGH state at a count k and then connect the normal outputs of these flip-flops to the NAND gate inputs. **Example**

Design a JK flip-flop-based modulo-6 asynchronous up counter.

Solution

The MOD number $k = 6$, and this is not a power of 2. Using the equation above:

$$2^{N-1} < 6 < 2^N$$

we find that the value of N which satisfies this is $N = 3$, hence we use three flip-flops which we connect as a ripple counter. $k = 6 = 110_2$, hence the two most-significant digits of the counter are the ones that should be connected to the input of the NAND gate, to yield the circuit shown on Figure below.

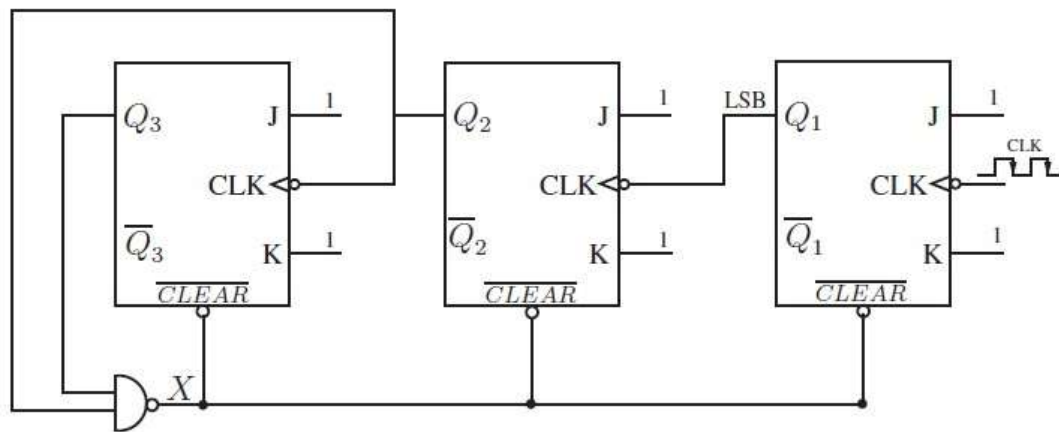


Figure : A modulo 6 up ripple counter

Assuming that we start with $Q_3 = Q_2 = Q_1 = 0$, we can draw the waveforms shown on Figure below.

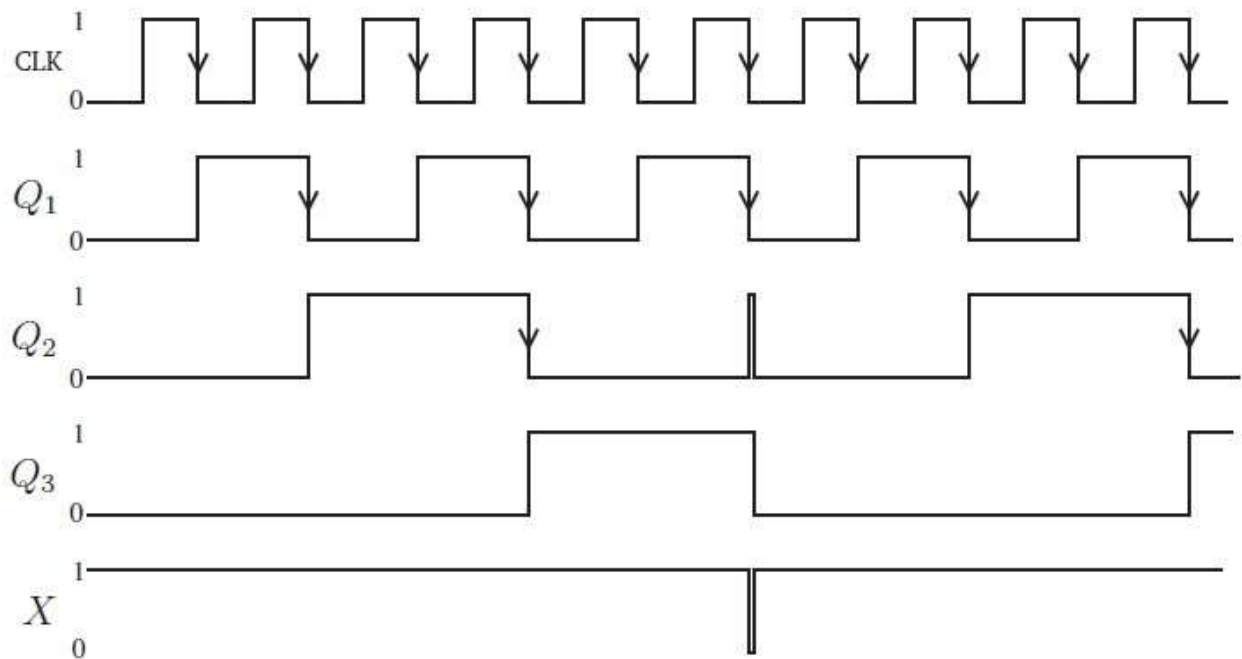


Figure : Waveforms for the counter in Figure above

From Figure of Waveforms for the counter above, we can see that when Q_3 and Q_2 go HIGH, the output of the NAND gate (X) goes LOW, and this clears all the flip-flops (Q_3 , Q_2 and Q_1 are taken to the LOW state). When Q_3 and Q_2 go LOW, the NAND gate output goes HIGH and the counter resumes normal operation.

The duration of the resetting pulse is very short, typically $< 100\text{ns}$, so the counter state 110 a temporary state and is not considered as a valid state of the counter. Assuming that we start with $Q_3 = Q_2 = Q_1 = 0$, we can draw the truth-table for this counter as shown below:

Q_3	Q_2	Q_1	clock pulses
0	0	0	before applying clock pulses
0	0	1	after clock pulse #1
0	1	0	after clock pulse #2
0	1	1	after clock pulse #3
1	0	0	after clock pulse #4
1	0	1	after clock pulse #5
1	1	0	temporary state, cannot be observed
0	0	0	recycles to 000 after clock pulse #6
0	0	1	after clock pulse #7
0	1	0	after clock pulse #8

The counter has 6 stable states (000, 001, 010, 011, 100 and 101) hence it is a modulo-6 up asynchronous counter.

Synchronous Counters

These are counters in which all the flip-flops are clocked simultaneously, and as a result, all flipflop output changes occur simultaneously.

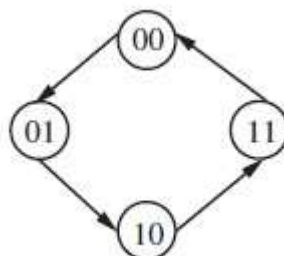
Propagation delays do not add together hence synchronous counters can work at much higher frequencies than ripple counters.

The design procedure is as follows:

- (i) Write the state-transition-table for the count.
- (ii) Derive a minimal logic expression for each flip-flop input.
- (iii) Implement the logic circuit using flip-flops and a suitable set of logic gates.

Example

Design a JK flip-flop based circuit whose state diagram is shown on Figure below.

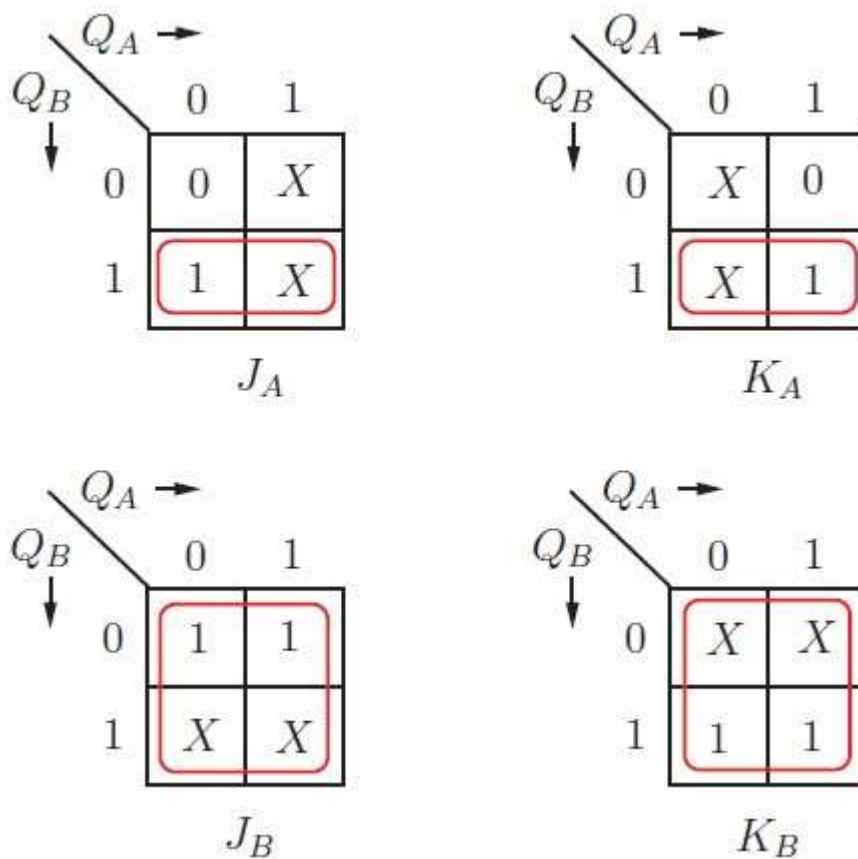


Solution

Since we are to use JK flip-flops to implement this function, we shall use the JK flip flop excitation table. We then construct the table shown below:

PRESENT STATE		NEXT STATE		FLIP-FLOP INPUTS			
Q_A	Q_B	Q_{A+1}	Q_{B+1}	J_A	K_A	J_B	K_B
0	0	0	1	0	X	1	X
0	1	1	0	1	X	X	1
1	0	1	1	X	0	1	X
1	1	0	0	X	1	X	1

The K- maps corresponding to the four flip-flop inputs are shown on Figure below.



From the figure, we can see that:

$$J_A = K_A = Q_B \quad \text{and} \quad J_B = K_B = 1$$

The circuit is shown on Figure below.

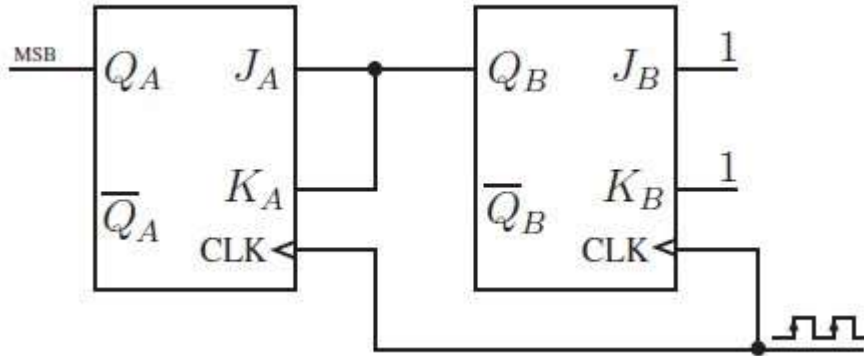
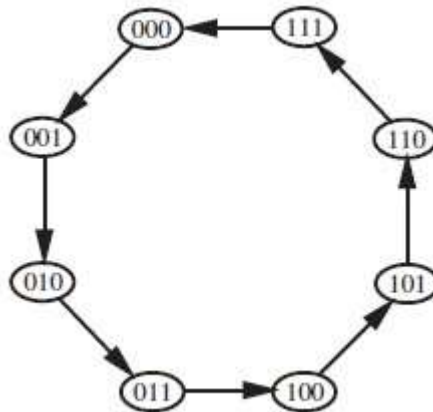


Figure: Circuit for a MOD 4 synchronous counter

Note that the flip-flops may be either positive-edge-triggered or negative-edge-triggered. For observation of the states, Q_A and Q_B may be connected to LEDs (Q_A is the MSB while Q_B is the LSB).

Example

Design a JK flip-flop based circuit whose state diagram is shown on Figure below.



Using the JK flip-flop excitation table, we construct the table shown below:

PRESENT STATE			NEXT STATE			FLIP-FLOP INPUTS					
Q_A	Q_B	Q_C	Q_{A+1}	Q_{B+1}	Q_{C+1}	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	0	0	0	X	1	X	1	X	1

The K- maps corresponding to the four flip-flop inputs are shown on Figure below. From Figure below, we can see that:

$$J_A = K_A = Q_B Q_C, \quad J_B = K_B = Q_C \quad \text{and} \quad J_C = K_C = 1$$

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	0	0	X	X
1	0	1	X	X

J_A

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	X	X	0	0
1	X	X	1	0

K_A

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	0	X	X	0
1	1	X	X	1

J_B

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	X	0	0	X
1	X	1	1	X

K_B

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	1	1	1	1
1	X	X	X	X

J_C

$Q_A Q_B \rightarrow$

Q_C	00	01	11	10
0	X	X	X	X
1	1	1	1	1

K_C

The circuit is shown on Figure below. In this case, QA is the MSB while QC is the LSB.

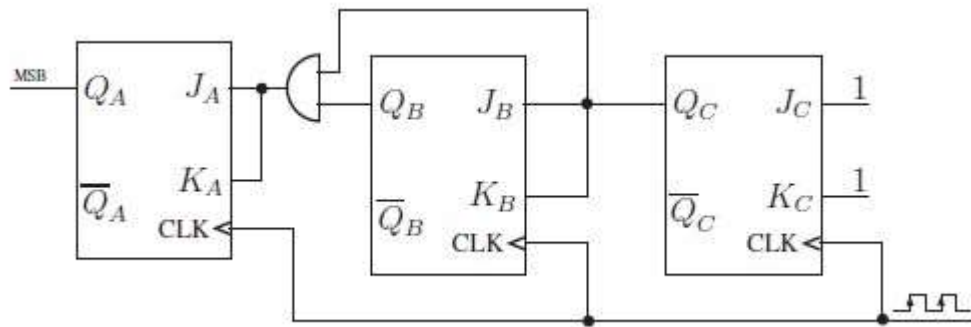


Figure: Circuit for a MOD 8 synchronous counter

Logic families

We have different logic families used to hardware-implement different logic functions in the form of digital integrated circuits. Logic families to be discussed include Resistance-transistor logic (RTL), Diode-transistor logic (DTL), transistor-transistor logic (TTL), metal oxide semiconductor (MOS) logic, emitter coupled logic (ECL), Complementary metal-oxide semiconductor (CMOS) logic and integrated injection logic (I²L).

Saturated and Non-Saturated Logic Circuits

Those logic circuits in which transistors are driven into saturation are called saturated logic circuits or simply saturated logic.

Those circuits which avoid saturation of their transistors are designed non-saturated logic.

The disadvantage of saturated logic is the delay that occurs when the transistors are brought out of saturation. **When a transistor is saturated, its base is flooded with carriers.** Even when base voltage is switched off, the base remains flooded for some time till all carriers leave it.

The time required by the carriers to leave the base is called saturation delay time (t_s). Obviously, saturated logic circuits have low switching speeds whereas non-saturated types are much faster.

TTL is the example of a saturated logic whereas **ECL represents a non-saturated logic.**

Basic Operating Characteristics and Parameters of Logic Families

When we work with digital ICs from different logic families, we should be familiar with, not only their logical operation but also with the basic operational properties. Following are the important basic operational properties important from the subject point of view.

- | | | |
|-------------------------|-----------------------|-----------------------|
| 1. DC supply voltage. | 2. Fan in/out. | 3. Noise immunity |
| 4. Noise margin. | 5. Power dissipation. | 6. Propagation delay. |
| 7. Speed-power product. | 8. Loading | |

Now we will describe all the above operational characteristics one by one.

DC Supply Voltage

Is the voltage required for an IC to operate.

It is supplied by one or more power-supply voltages and connected at V_{CC} (TTL) or V_{DD} (MOS devices).

For many ICs, current drawn from the supply varies depending on logic states of the circuits on the chip.

Fan in: The number of inputs that the gate can handle properly without disturbing the output level.

Fan out: The number of inputs that can be driven simultaneously by the output without disturbing the output level.

Noise immunity: **is the ability of the logic circuit to tolerate the noise voltage.**

Noise Margin: is the quantitative measure of noise immunity.

Propagation Delay: The propagation delay of gate **is the average transition delay time for the signal to propagate from input to output.**

It is measured in nanoseconds

Threshold Voltage: The voltage at which the circuit changes from one state to another state

Operating Speed: The speed of operation of the logic gate is **the time that elapses between giving input and getting output.**

Power Dissipation: is power needed by the logic circuit.

RTL Circuit

It is a saturated logic. It uses only transistors and resistors as circuit elements and also resistances in the input to each base. This family is based on the NOR circuit shown in Fig. 1. All other members of the family are made up of NOR cells or variations on them.

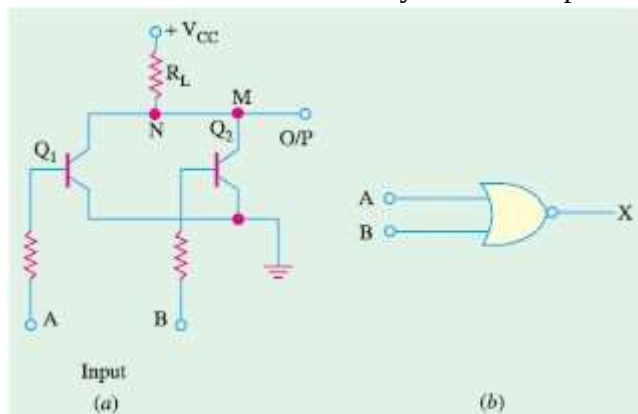


Fig 1

Circuit Operation

We will assume ideal transistors. When both inputs A and B are 0 V (or logic 0) both transistors are turned OFF, hence point M goes to + V CC so that output is logic 1.

If either or both input terminals are at $+V_{CC}$ i.e. are high (or logic 1), one or both transistors would be fully turned ON (i.e. saturate) thereby reducing the voltage of point N to almost 0 V. Hence, output would be at logic 0.

It is seen that the output is at logic 1 only when both inputs are at logic 0 — the NOR logic functions as shown in Fig. 1 (b).

The RTL family has the following characteristics:

1. relatively slow speed,
2. low fan-out of 6 and a fan-in of 4,
3. poor noise immunity,
4. expensive since resistors are required to be fabricated,
5. cannot operate at speed above 4 MHz.

DTL Circuit

It is a saturated logic because transistors operate between cut-off and saturation. It was the next family to be introduced after RTL. It consists of diodes, resistors and transistors. The basic gate of this family performs NAND function. As shown in Fig. 2 the circuit basically consists of a diode AND gate followed by a transistor inverter which leads to a NAND gate.

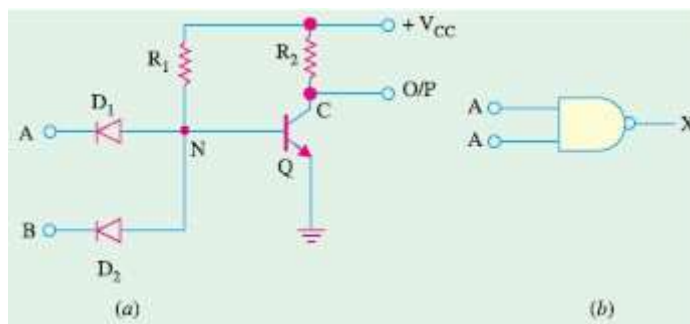


Fig. 2

Circuit Operation

1. When both D_1 and D_2 have positive voltage applied to them (logic 1), neither conducts and Q is turned ON by the current provided by V_{CC} through R_1 . Since Q becomes saturated, point C is brought to 0V (logic 0). Hence output goes logic 0.
2. If either or both inputs are at 0V (logic 0), the associated diode will conduct driving point N to ground i.e. 0V. Since there is no base voltage for Q, it will be cut OFF thereby driving point C and hence output to V_{CC} i.e. logic 1.

It is seen that output is low (a logic 0) only when all inputs are high—the condition for a NAND gate.

The DTL family is characterised by

1. relatively lower speed,
2. Comparatively better noise immunity,

3. propagation delay of 30 ns,

4. a fan-out of 5.

TTL Circuit

It is a saturated logic. It is the most widely used circuit line since early 1970s because of its speed, good fan-out and easy interface with other digital circuitry. The unique feature of this circuit is that it used multiple-emitter transistor at input which replaces the input diodes of the DTL. The number of emitters is equal to the number of inputs of the logic circuit (limited to 8). Since a multi-emitter transistor is small in area than the diodes it replaces, the yield from a wafer is increased. Moreover, smaller area results in lower capacitance to the substrate, thereby wide selection of circuit modules ranging from simple gates and flip-flops in SSI circuit series through various registers in computers in MSI circuit series to micro-processor bit-slice chips in the LSI series.

Basic Circuit

The basic circuit of the TTL family is the NAND gate cell shown in Fig. 3. However, at present NOR, OR and AND gate configurations have also been added to the series.

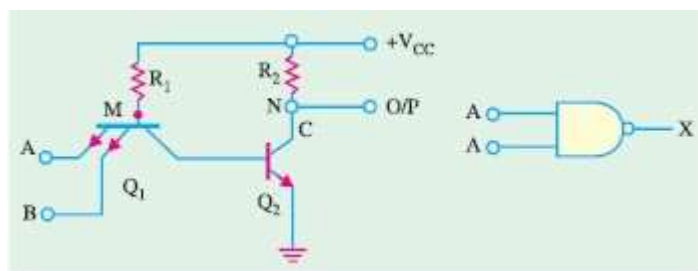


Fig 3

Circuit Operation

1. If both inputs A and B are high (logic 1), E/B junction of Q_1 is reverse-biased so that it has no emitter current. Hence Q_1 is OFF. However, its C/B junction is forward-biased supplying base current to Q_2 from V_{CC} via R_1 . As a result transistor Q_2 is turned fully ON (i.e. it becomes saturated) driving point N to 0 V. Hence, output is a logic 0.
2. When either or both inputs are at 0V (logic 0), the associated E/B junction becomes forward-biased. The value of R_1 is so selected as to ensure that Q_1 is turned fully ON. The voltage at point M falls to 0V with the result the base current for Q_2 is reduced to zero. Hence, Q_2 is cut OFF driving point N and the output to logic 1.

Totem Pole Output

The basic circuit of Fig. 3 is never used in practice. Its modified version with an added output stage is in common use (Fig. 4).

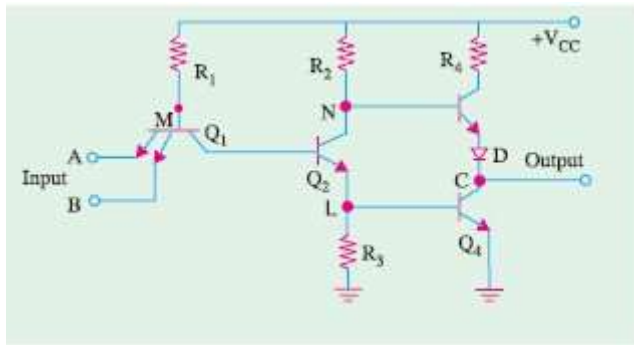


Fig .4

This extra output stage is often known as totem-pole stage because the three components Q3, Q4 and D are stacked one on top of the other in the manner of a totem-pole. The circuit action is as follows:

1. When Input is High

In this case, the two input terminals have positive voltage (logic 1). the E/B junction is reversebiased because of which there is no emitter current. Hence, Q1 is OFF.

Since C/B junction of Q1 is forward-biased, base current of Q2 flows from V_{CC} through R_1 . Hence, Q2 is turned ON. As a result, potential of point N falls so much that Q2 is turned OFF. At the same time Q4 is turned ON by the voltage drop across R_3 . Now, when Q4 is ON, its collector potential (i.e. potential of point C) is nearly that of its emitter. Hence, output is low i.e. at logic 0. In short, when inputs are at logic 1, Q1 is OFF, Q2 is ON, Q3 is OFF and Q4 is ON because of which output is logic 0.

2. When Input is Low

If any of the two inputs or both are low (logic 0), Q1 turns ON and potential of its collector (point M) falls. Hence, Q2 is turned OFF, grounding its emitter and the base of Q4 so that Q4 is also turned OFF.

Since N is at V_{CC} , it turns Q3 ON. The potential of point C is V_{CC} minus drop in R_4 , Q3 and D. Since these drops do not amount to much, output is at logic 1.

It may be noted that when even-numbered transistors are ON, the odd-numbered ones are OFF and vice-versa.

The function of diode D in Fig. 4 is to prevent both Q3 and Q4 from being turned ON simultaneously. If both were to be On at the same time, they would offer low impedance to the supply which will draw excessive current and produce large noise 'spikes' in the output. **It may also be noted that the addition of a pair of totem pole transistor increases the operating speed and output current capability of this circuit.** The standard TTL-family has

1. greater speed than DTL,
2. less noise immunity (0.4 V),
3. average propagation delay per gate of 9 ns,
4. average power dissipation of 10 mW,
5. a fan-out of 10 meaning one output can drive 10 other TTL inputs,

ECL Circuit (OR/NOR circuit)

The ECL also called current-mode logic (CML), has the highest speed of any of the currently available logic circuits. It is primarily due to the fact that transistors never operate fully saturated or cut-off. That is why ECL is known as non-saturated logic. The latest ECL series has propagation delay time varying from 0.1ns to 0.8 ns. However, power dissipation is increased since one transistor is always in the active region.

Another feature of ECL is that it provides two outputs which are always complement of each other (Fig. 5). It is so because the circuit operation is based on a differential amplifier.

This family is particularly suited to monolithic fabrication techniques because logic levels are function of resistor ratios.

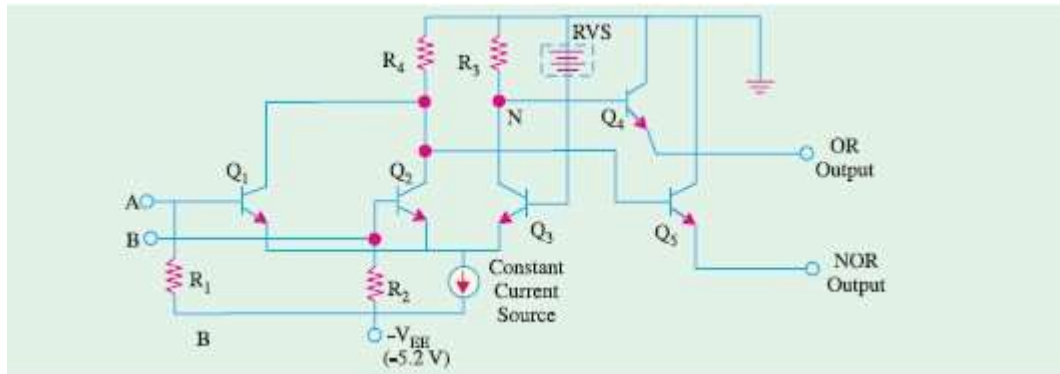


Fig 5

Circuit Operation

The basic circuit shown in Fig. above is combined **OR/NOR circuit and is operated from a $V_{EE} = -5.2 \text{ V}$ supply**. A built-in constant-current source provides current to the emitters. Strictly speaking, logic 1 is represented by -0.9 V (less negative) and logic 0 by -1.75 V (more negative). Please note that it is a positive logic. In negative logic, the functions would be AND/NAND. A reference voltage of -1.29 V is applied to the base of Q_3 from a built-in temperature-compensated reference voltage source (RVS).

1. When both inputs are logical 0 i.e. -1.75 V

In this case, base potential of Q_3 is less negative (or more positive) than the base potential of either Q_1 or Q_2 . Hence, Q_3 conducts whilst Q_1 and Q_2 do not. Only enough base current is drawn by Q_3 from RVS so as to remain out of saturation. The collector current of Q_3 develops a voltage of -1.0 V across R_3 which makes Q_4 to conduct. Transistor Q_4 gives an output voltage at the emitter of about $-1 - 0.7 = -1.7 \text{ V}$ which represents logic 0. Since collector potentials of Q_1 and Q_2 are nearly zero (because they are cut off), the output voltage at the emitter of Q_5 is $0 - 0.7 = -0.7 \text{ V}$ which is a logic 1. Obviously, the two outputs are complements of each other.

2. When either input A or B is at logical 1 i.e. -0.9 V

In that case, the associated transistor (either Q_1 or Q_2) is turned ON while Q_3 is turned OFF. The collector potentials of Q_1/Q_2 and are opposite of the previous case. Hence, now Q_5 output is at logical 0 and Q_4 output is at logical 1.

Typical characteristics of an ECL family are:

1. propagation delay time per gate of 0.3 ns (meaning extremely fast speed),
2. power dissipation of 25 mW ,

3. fan-out of 25 to 50,
4. noise margin from about 0.2 to 0.25 V.

It will be interesting to know that ECL family of ICs does not include a wide range of general purpose logic devices as do the TTL and CMOS logic families. ECL does include complex, special purpose ICs used in applications such as high-speed data transmission, high-speed memories and high-speed arithmetic units. The relatively low noise margins and high power drain of ECL are disadvantages compared with TTL and CMOS. Another drawback is its negative power supply voltage and logic levels which are not compatible with those of other logic families. This makes it difficult to use ECL devices in conjunction with TTL and/or CMOS ICs. Special level-shifting (also called interface) circuits must be connected between ECL devices and the TTL (or CMOS) devices on both input and output.

I²L Circuit

It is the latest entry into the bipolar saturated logic field. It uses no biasing and loading resistors at all! Resistors require lot of power and space on an IC chip. Hence, their elimination results in higher density circuits operating at much reduce power. Because of its high speed and less power dissipation, it is used in large computers. Such circuits are also used where high packing density is of prime consideration as in digital wrist watches. I²L chips are capable of microwatt power dissipation yet can provide high currents when necessary to drive LED displays. Another feature of I²L (integrated injection logic) is that it is easy to fabricate.

Circuit Operation (NOR logic function)

I²L NOR logic circuit is shown in Fig. 6. Here, transistors Q1 and Q3 act as current sources to the bases of Q2 and Q4 respectively.

If A input goes low, the current to the base of Q2 will be shorted to ground which will result in Q2 being turned OFF. The input B controls Q4 in a similar way.

If A is high, base current flowing to Q2 will turn in ON, making output C low. Same would be the case when B is high.

It is obvious that output would be high only when both inputs A and B are low.

The output would be low when either A or B or both are high i.e. NOR logic function.

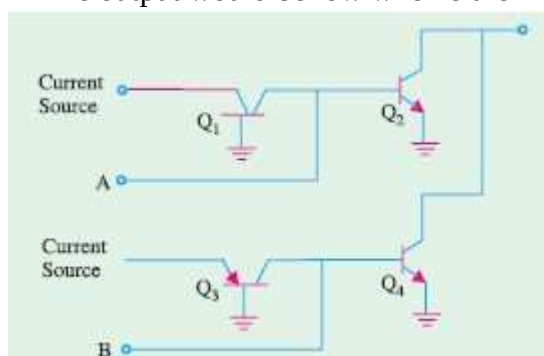


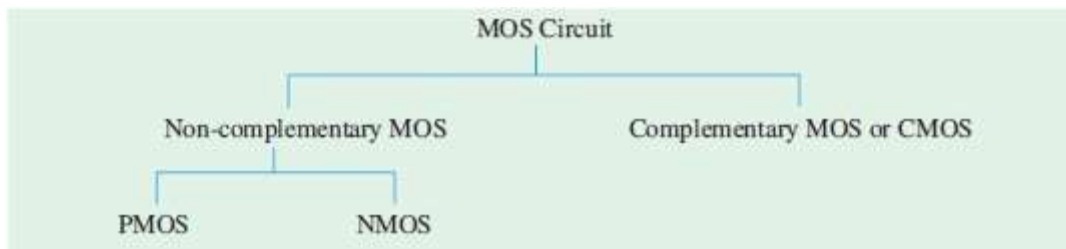
Fig 6

MOS Family

It does not use bipolar transistors but just Enhancement-only MOSFETs. There are two kinds of digital MOS circuits.

- (a) one which uses MOSFETs of one polarity either all of N-type called NMOS or all of P-type called PMOS but not both on the same chip,
- (b) the other which employs both N-type and P-type MOSFETs on the same chip. It is called complementary MOS (CMOS).

The MOS family may be subdivided as under:



Since a FET requires small area, it is possible to fabricate a large number of MOS circuits on a single small chip. Gating arrays with thousands of gates and flip-flops are manufactured in standard containers and are often used in IC memories and microprocessors.

Followings are some of the advantages of MOS ICs over the bipolar ICs (i.e. TTL, ECL etc.)

1. The MOS IC is relatively simple and inexpensive to fabricate.
2. The MOS device size is small and it consumes less power. Because of the small size, the MOS ICs can accommodate a much larger number of circuit elements on a single chip than bipolar IC in the area of large scale integration. This makes them especially well-suited for complex ICs such as microprocessor, memory chips etc.
3. MOS digital ICs normally do not use the IC resistor elements that take up so much of the chip area or bipolar ICs.

The continuous improvement in MOS IC technology has led to the device that are faster than TTL devices. Consequently, MOS devices (especially CMOS) have become dominant in the SSI (small-scale integration) and MSI (medium-scale integration) market. The major disadvantage of MOS devices is that they are susceptible to static-electricity damage. Although we can minimize it by adopting proper handling procedures, yet TTL is still more durable for laboratory experimentation. As a result, we are likely to see TTL devices used in education as long as they are available.

PMOS Circuit (NAND function)

A PMOS NAND gate is shown in Fig. 7 (a). As seen, there are no resistors in the circuit. The gate consists of two E only MOSFETs Q1 and Q2 as logic elements and the third one Q3 as load resistors. When $-V_{DD}$ (say -12V) is applied, MOSFETs will be turned ON and when 0 V is applied they would be turned OFF. Hence, with positive logic, 0 V would be 1 and -12 V would be 0 since 1 is assigned to the most positive voltage.

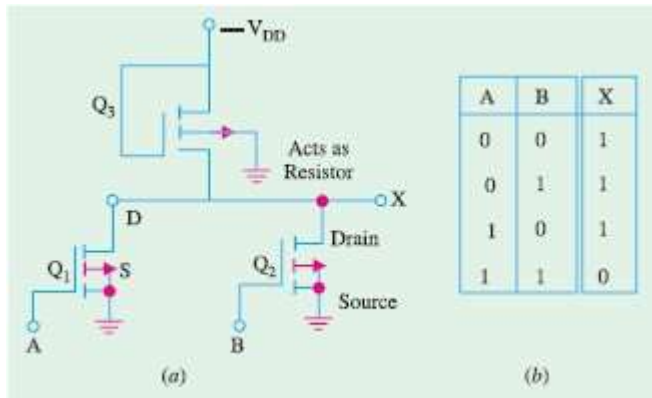


Fig 7

Circuit Operation

If any of the two inputs A or B is at logic 0 (i.e. -12 V), the concerned MOSFET would turn ON thereby offering a low resistance from drain to source thus causing the output to be nearly 0V i.e. a logic 1.

2. The output can be at -12 V i.e. logic 0 only when both inputs A and B are at 0 V i.e. logic 1.

This is the NAND function as shown by the truth table of Fig. 7 (b). Incidentally, the positive logic NAND gate of Fig. 7 (a) would be the negative logic NOR gate since the two are identical.

NMOS Circuit (NOR function)

In Fig. 8 (a) is shown a two-input NOR gate circuit consisting of two MOSFETs Q_1 and Q_2 acting as logic elements and Q_3 as a load resistor. For positive logic, 0V will be logic 0 and positive voltage ($+V_{DD}$) will be logic 1.

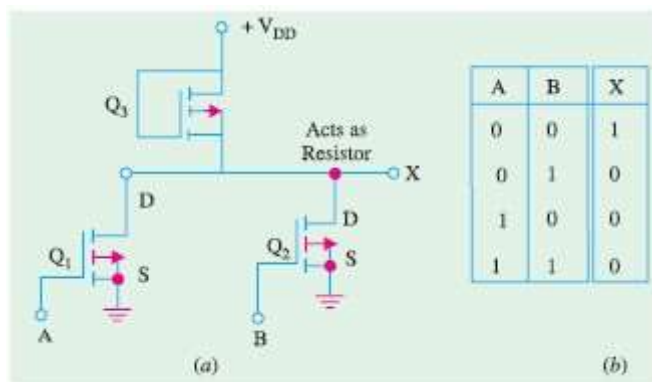


Fig 8

Circuit Operation

1. If any of the inputs A or B is at logic 1, the corresponding MOSFET will conduct causing the output to go low i.e. logic 0.

2. If both inputs A and B are at logic 0, then both MOSFETs will be OFF driving the output to logic 1.

This is NOR function as shown by the truth table of Fig. 8 (b).

CMOS circuit

CMOS logic circuits use both PMOS and NMOS devices in the same circuit. It gives the advantage of drastic decrease in power dissipation (12 nW per gate) and increase in speed of operation. In fact, it has the lowest power dissipation amongst different logic families. It has very high packing density i.e. larger number of circuits can be placed on a single chip. As a result, it is extensively used in VLSI circuits such as on-chip computers and memory systems. The recent silicon-on-sapphire MOS (SOSMOS) is 2 to 4 times faster than the standard CMOS. Hence, they are being widely used for everything from electronic watches and calculators to microprocessors. Fig. 9 (a) shows a CMOS NOR circuit which has two N-channel MOSFETs Q₁ and Q₂ and two P-channel MOSFETs Q₃ and Q₄. The two inputs A and B switch between + V_{DD} (logic 1) and ground (logic 0).

Circuit Operation (NOR function)

The circuit is a function of input signal frequency. Higher the frequency, greater the power dissipation.

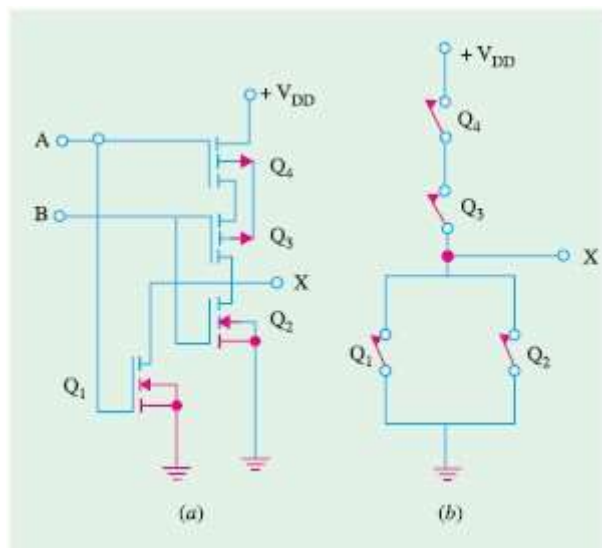


Fig. 9

1. Let A = B = logic 1 i.e. have positive voltage. In that case, Q₁ and Q₂ are ON (closed switches) but Q₃ and Q₄ are OFF and act as open switches as shown in Fig. 9 (b). Hence, output C is logic 0.

2. If either A or B is at logic 1, then the associated N-channel MOSFET (Q1 or Q2) is turned ON but the associated P channel MOSFET (Q3 or Q4) is turned OFF. Since either Q3 or Q4 would be OFF [Fig. 9 (b)], output C would be at logic 0.
 3. When both A and B are at logic 0, Q3 and Q4 would be ON but Q1 and Q2 would be OFF—just the opposite of that in Fig. 9
- (b). Hence output C would be at logic 1 (remember, voltage across an open equals the supply voltage).

The above logic represents a NOR function as shown in the truth table of Fig. 10. It would be observed from Fig. 9 that in each combination of A and B, there is at least one open switch between + V_{DD} and ground. Hence, the gate draws only leakage current from supply for any static state. However, when the gate switches from one level to another, some power is consumed because two MOSFETs are partly ON at the same time. Because of this reason, power dissipated by CMOS

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

Fig 10