unit-1

Introduction

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

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

Decimal Numbers

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

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

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

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

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

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

r = base or radix of the system.

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

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

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

Thus for decimal fraction 0.7123

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

where a -1 = 7
a -2 = 1
a -3 = 2
a -4 = 3

Binary Numbers

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

Example : The decimal equivalent of the binary number 101010.

$$N = 101010$$

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

$$= 43$$

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

$$2^r > N$$

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

Octal Numbers

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, octal and hexadecimal, are used for representing large binary numbers. Octal systems use a base or radix of 8. Thus it has digits from 0 to 7 (r-1). As in the decimal and binary systems, the positional valued of each digit in a sequence of numbers is

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

Example : The decimal equivalent of the octal number 15.2.

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

Hexadecimal Numbers

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

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

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

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

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

Examples:

+3 = +3

-3 = 10^2 -3 = 97

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

Example:

119 10 = 01110111 2

Complementing bits will result

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

Properties of Two's Complement Numbers

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

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

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

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

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

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

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

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

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

Two' complement addition/subtraction example

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

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

Number Base Conversions

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

number in base r. If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since each part must be converted differently. The conversion of a decimal integer to a number in base r is done by dividing the number and all successive quotients by r and accumulating the remainders. The conversion of a decimal fraction is done by repeated multiplication by r and the integers are accumulated instead of remainders.

Integer part - repeated divisions by r yield LSD to MSD

Fractional part - repeated multiplications by r yield MSD to LSD

Example: Conversion of decimal 23 to binary is by divide decimal value by 2 (the base) until the value is 0

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

The answer is $23_{10} = 10111_2$

Divide number by 2; keep track of remainder; repeat with dividend equal to quotient until zero; first remainder is binary LSB and last is MSB.

The conversion from decimal integers to any base-r system is similar to this above example, except that division is done by r instead of 2.

Example:

Convert $(0.7854)_{10}$ to binary.

$0.7854 \times 2 = 1.5708$; $a_{-1} = 1$

$0.5708 \times 2 = 1.1416$; $a_{-2} = 1$

$0.1416 \times 2 = 0.2832$; $a_{-3} = 0$

$0.2832 \times 2 = 0.5664$; $a_{-4} = 0$

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

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

The conversion of decimal numbers with both integer and fraction parts is done by converting the integer and the fraction separately and then combining the two answers.

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

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

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

Examples:

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

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

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

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

Examples:

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

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

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

Example:

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

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

Boolean Algebra and Basic Operators

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

Closure

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

Identity

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

Distributive

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

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

Complement

$X + \overline{X} = 1$

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

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

Associative

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

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

Idempotence

$X \cdot X = X$

$X + X = X$

Absorption

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

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

Simplification

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

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

Consensus

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

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

Adjacency

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

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

Demorgans

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

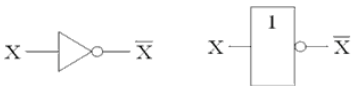
Switching Algebra Operations

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

NOT

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

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



AND

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

X	Y	X·Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

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

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

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

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

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

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

1.1 Introduction

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

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

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

1.2 Decimal Number System

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

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

Binary Number System

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

Octal Number System

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

Hexadecimal Number System

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

Converting any Radix to Decimal

In general, numbers can be represented as

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

where N = Number in decimal

A = Digit

r = Radix or base of a number system

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

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

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

Conversion of Decimal Numbers to any Radix Number

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

Successive Division for Integer Part Conversion

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

Successive Multiplication for Fractional Part Conversion

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

Complement Representation of Negative Numbers

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

1's Complement Representation

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

2's Complement Representation

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

2's complement = 1's complement + 1

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

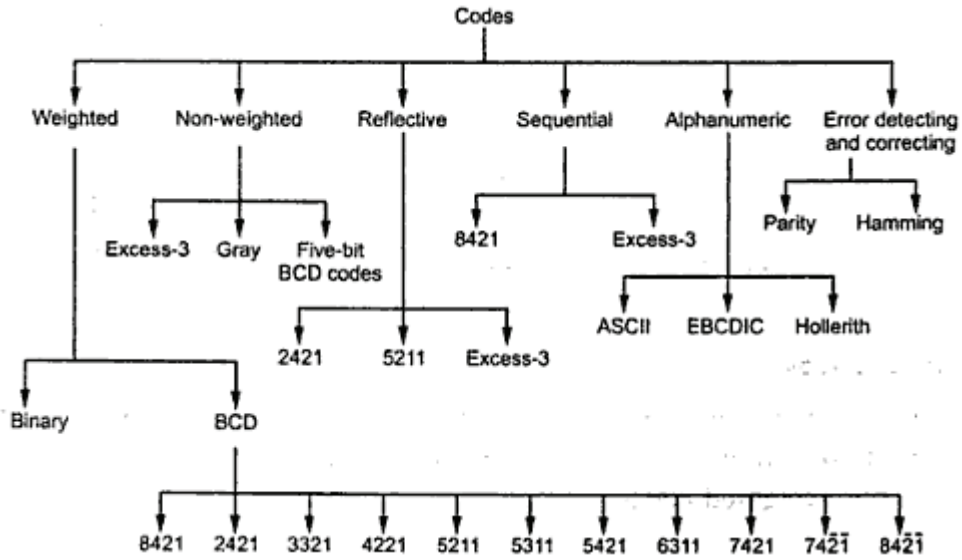
Rules for Binary Addition

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

Rules for Binary subtraction

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

Classification of Binary Codes



Excess-3 Code

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

Table shows excess-3 codes to represent single decimal digit

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

Excess-3 code

Excess-3 Addition

To perform Excess-3 addition we have to

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

Excess-3 Subtraction

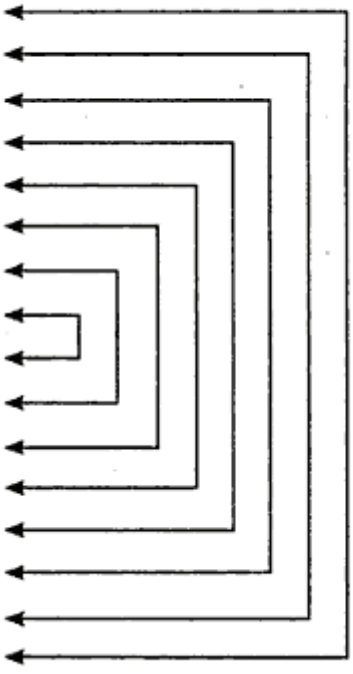
To perform Excess-3 subtraction we have to

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

Gray Code

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

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



Gray-to-Binary Conversion

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

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

Binary to Gray Conversion

Let us represent binary number as

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

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

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

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

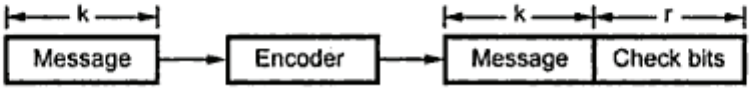
Parity Bit

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

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

Linear Block Codes

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



Generation of an (n, k) block code

Matrix Representation of Linear Block Codes

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

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

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

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

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

where

I_k is the identity matrix of order k and

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

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

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

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

We know that generation matrix is given as

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

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

$$C = DG$$

where C = Block code

D = Message bits

G = Generation matrix

Case 1 : Message '00'

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

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

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

Case 2 : Message '01'

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

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

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

Case 3 : Message '10'

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

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

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

Case 4 : Message '1 1'

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

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

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

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

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

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

Generalized Steps for Construction of Code

1. Construct G matrix as

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

where I_k : Identity matrix of order k

A : Arbitrary matrix

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

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

2. Determine all possible combinations of code using

$$C = D\,G$$

In general for this can be written as

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

Decoding the Received Codewords

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

1. From the matrix H as

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

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

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

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

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

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

where R : Received code

R^T : Transpose of R

Error Correction

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

The procedure is given below :

1. Find $S = RH^T$

   where R : Received code

   H^T : Transpose of T

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

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

Hamming Code

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

Number of Parity Bits

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

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

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

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

and

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

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

Then

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

and

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

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

Locations of the Parity Bits in the Code

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

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

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

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

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

Assigning Values to Parity Bits

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

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

Bit position table for a seven bit error correcting code

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

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

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

Detecting and Correcting an Error

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

UNIT-II

Boolean Postulates and Laws

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

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

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

- **Closure (b) :** Closure with respect to the operator · (dot).

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

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

  $A + 0 = 0 + A = A$

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

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

- Distributive property of + over · :

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

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

Rules in Boolean Algebra

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

   0 + 0 = 0 1 + 0 = 1

   0 + 1 = 1 1 + 1 = 1

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

Rule 1 :

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

Rule 2 :

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

Rule 3 :

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

Rule 4 :

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

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

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

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

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

Rule 5 :

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

Rule 6 :

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

Rule 7 :

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

Rule 8 :

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

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

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

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

From the previous result following rule is defined in Boolean algebra

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

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

Laws of Boolean Algebra

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

Commutative Laws

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

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

=

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

Truth table for commutative law for OR gates

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

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

=

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

Truth table for commutative law for AND gates

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

Associative Laws

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

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

=

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

Truth tables for associative law for OR gates

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

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

=

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

Truth table for associative law for AND gates

UNIT-III

GATE LEVEL MINIMIZATION

Karnaugh Maps

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

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

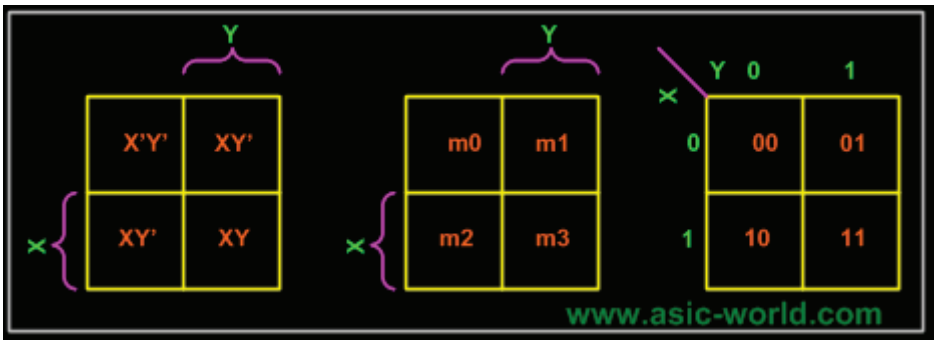
A K-map consists of a grid of squares, each square representing one canonical minterm combination of the variables or their inverse. The map is arranged so that squares representing minterms which differ by only one variable are adjacent both vertically and horizontally. Therefore XY'Z' would be adjacent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

Minimization Technique

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

2-Variable K-Map

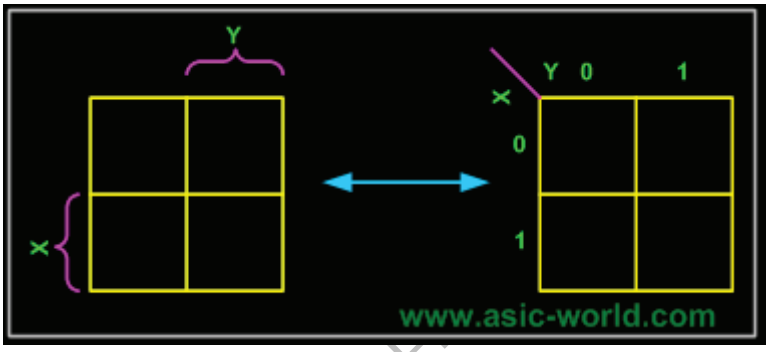
In any K-Map, each square represents a minterm. Adjacent squares always differ by just one literal (So that the unifying theorem may apply: X + X' = 1). For the 2-variable case (e.g.: variables X, Y), the map can be drawn as below. Two variable map is the one which has got only two variables as input.



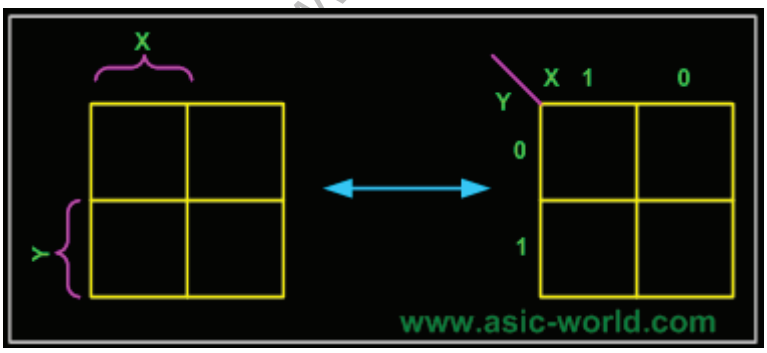
Equivalent labeling

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

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



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



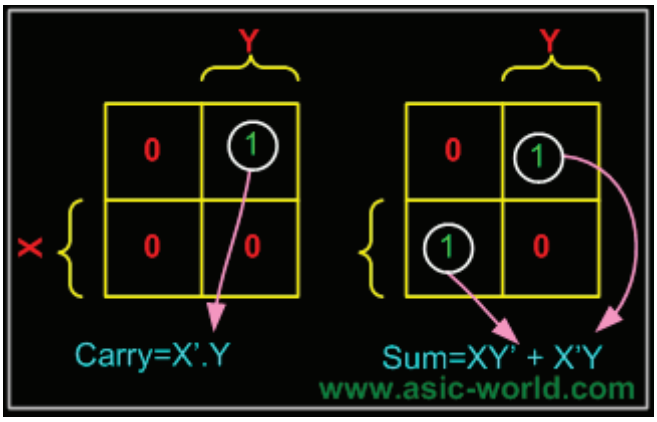
The K-map for a function is specified by putting a '1' in the square corresponding to a minterm, a '0' otherwise.

Example- Carry and Sum of a half adder

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

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

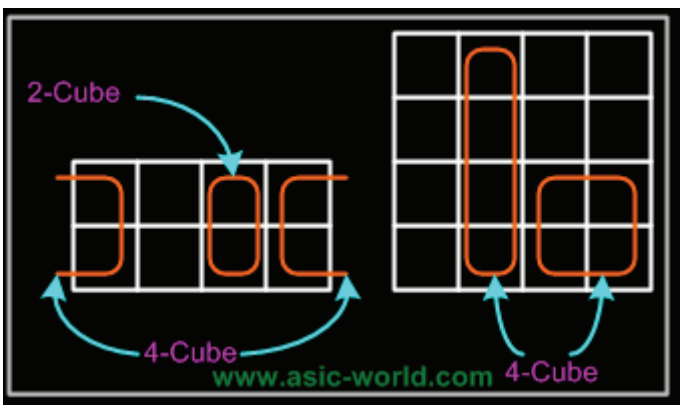
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Grouping/Circling K-maps

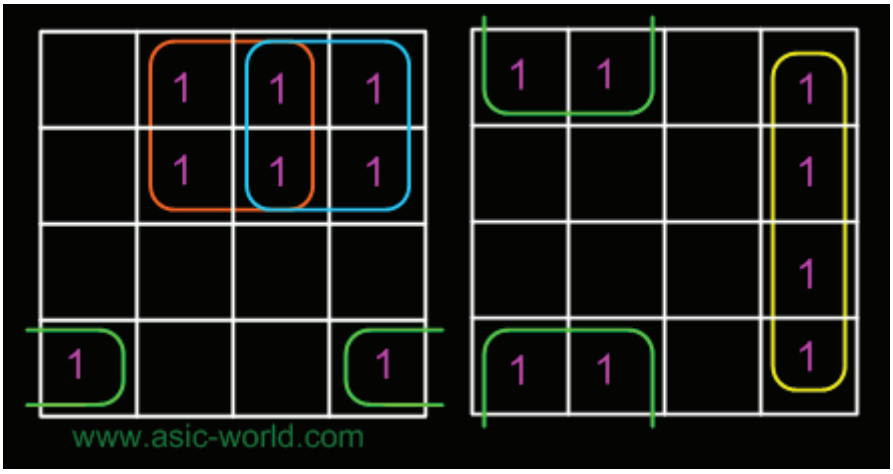
The power of K-maps is in minimizing the terms, K-maps can be minimized with the help of grouping the terms to form single terms. When forming groups of squares, observe/consider the following:

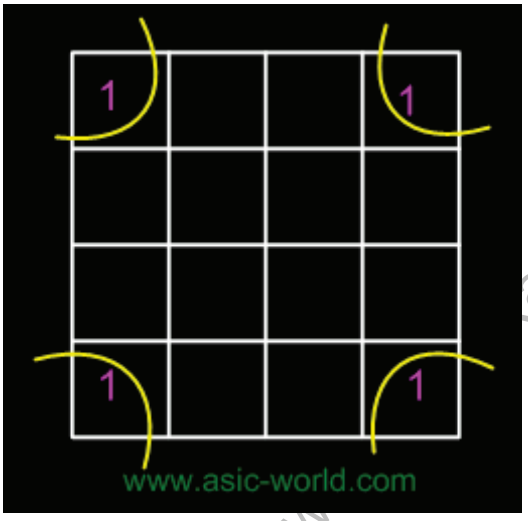
- Every square containing 1 must be considered at least once.
- A square containing 1 can be included in as many groups as desired.
- A group must be as large as possible.



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

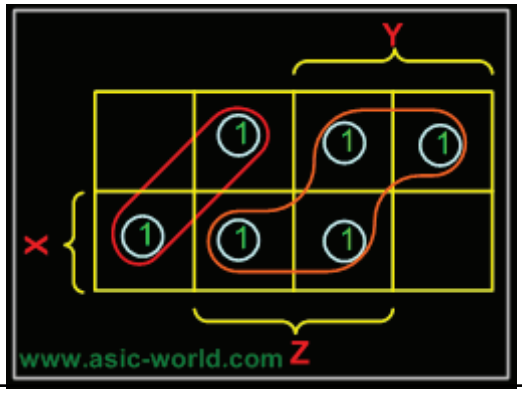
- The map is considered to be folded or spherical, therefore squares at the end of a row or column are treated as adjacent squares.
- The simplified logic expression obtained from a K-map is not always unique. Groupings can be made in different ways.
- Before drawing a K-map the logic expression must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





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

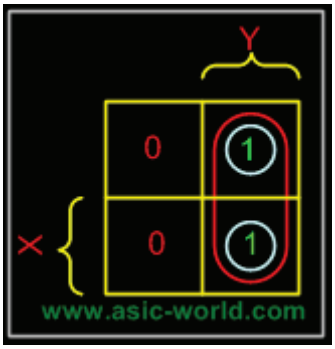
Example of invalid groups



Example - X'Y+XY

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

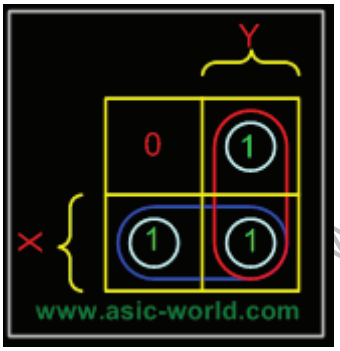
F = Y



Example - X'Y+XY+XY'

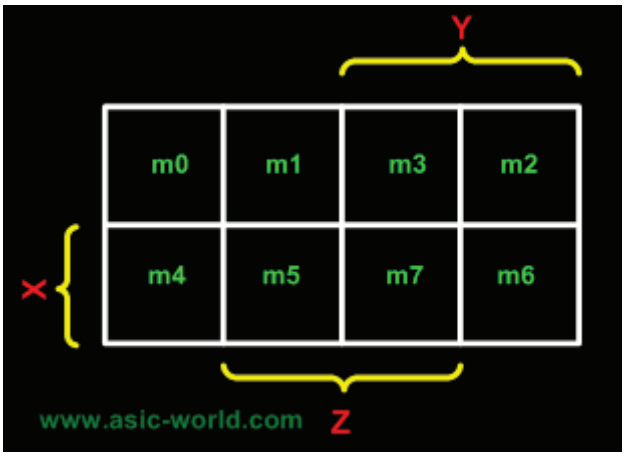
In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y, XY and XY position. Now combine two 1's as shown in figure to form the two single terms.

F = X + Y



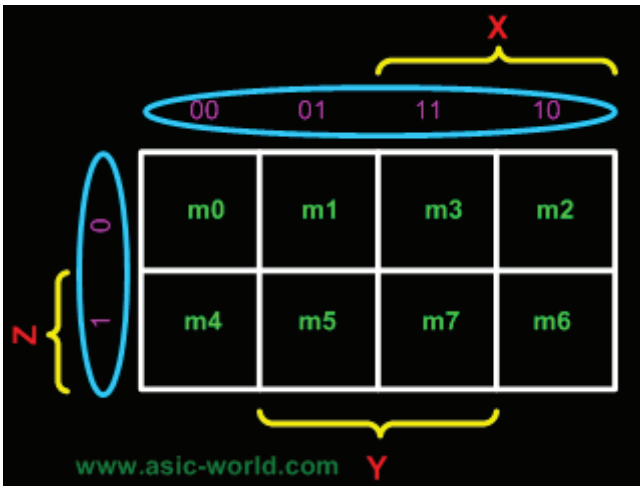
3-Variable K-Map

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



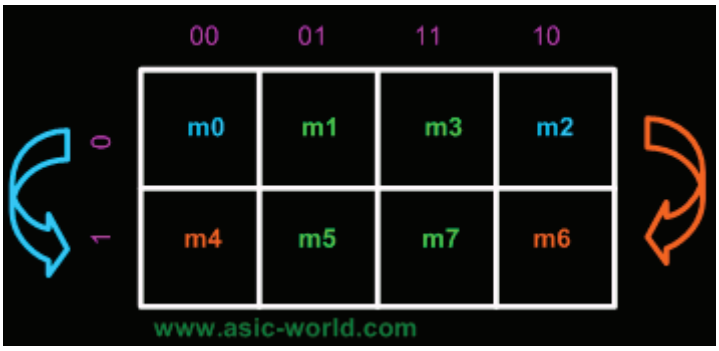
Using gray code arrangement ensures that minterms of adjacent cells differ by only ONE literal. (Other arrangements which satisfy this criterion may also be used.)

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



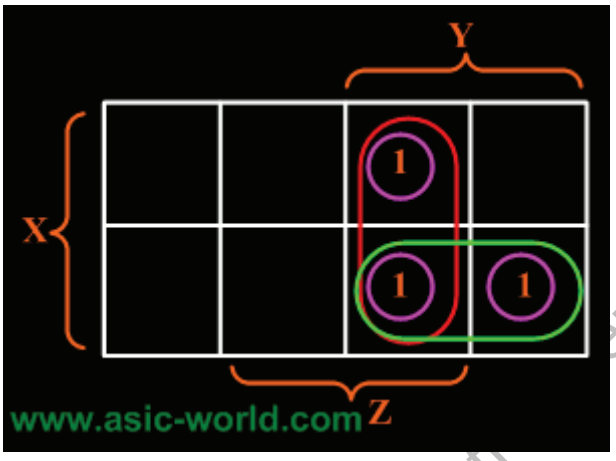
There is wrap-around in the K-map

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



Example

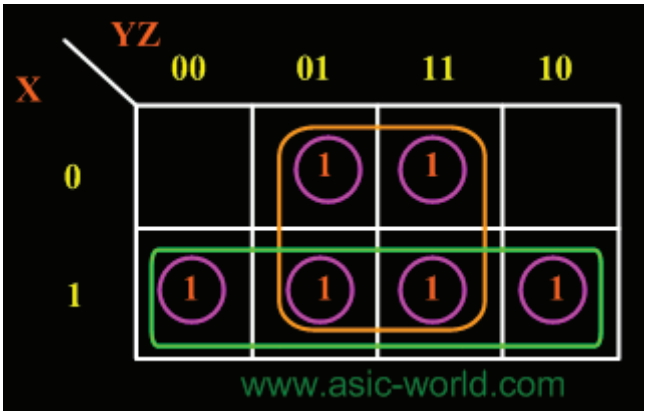
F = XYZ'+XYZ+X'YZ



F = XY + YZ

Example

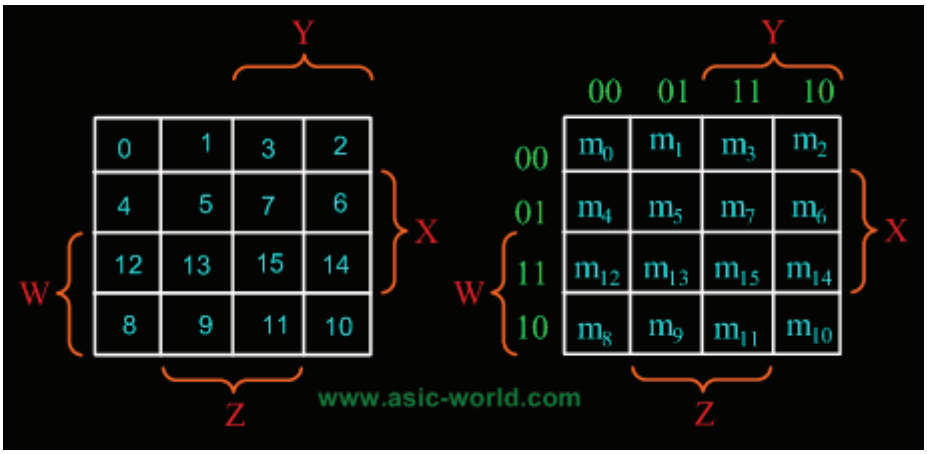
F(X,Y,Z) = Σ(1,3,4,5,6,7)



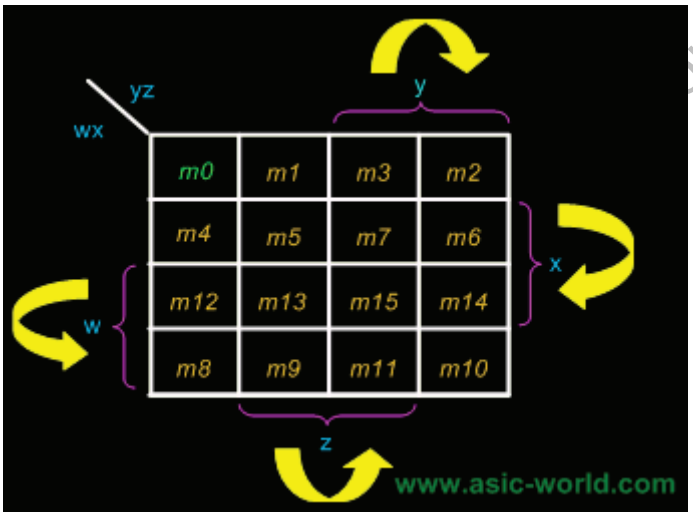
F = X + Z

4-Variable K-Map

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

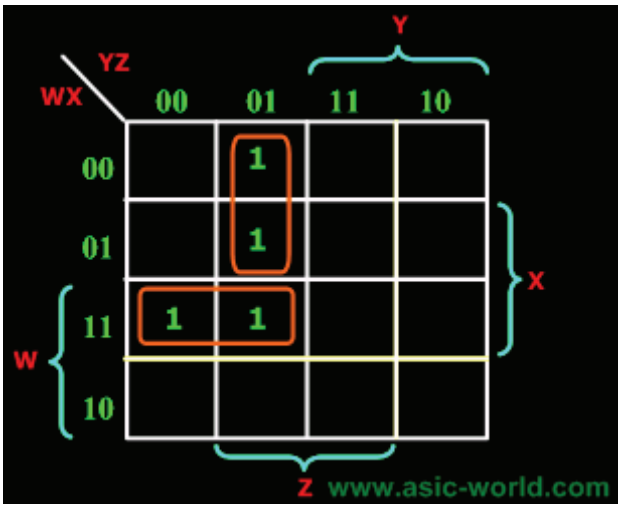


There are 2 wrap-around: a horizontal wrap-around and a vertical wrap-around. Every cell thus has 4 neighbours. For example, the cell corresponding to minterm m0 has neighbours m1, m2, m4 and m8.



Example

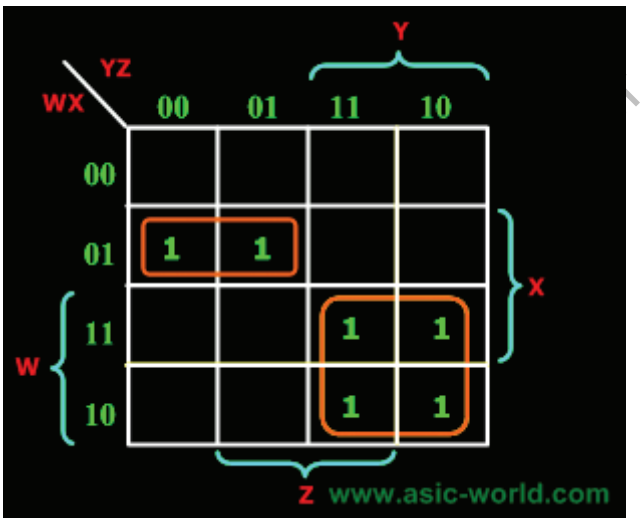
F(W,X,Y,Z) = (1,5,12,13)



F = WY'Z + W'Y'Z

Example

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



F = W'XY' + WY

QUINE-McCLUSKEY MINIMIZATION

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

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

Example

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

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

Binary representation of minterms

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

3	0	0	1	1
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
14	1	1	1	0
15	1	1	1	1

Group of minterms for different number of 1's

No of 1's	Minterms	U	V	W	X
1	1	0	0	0	1
1	2	0	0	1	0
1	8	1	0	0	0
2	3	0	0	1	1
2	9	1	0	0	1
2	10	1	0	1	0
3	7	0	1	1	1
3	11	1	0	1	1
3	14	1	1	1	0
4	15	1	1	1	1

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

2-Cell combinations

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

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

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

4-Cell combinations

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

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

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

Prime Implicants Table

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

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

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

UNIT-IV

Combinational Logic

UNIT-6

REGISTERS AND COUNTERS

Flip-flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capac bits, we have to use a group of flip-flop. Such a group of flip-flop is known as a **Register**. The **n-** of **n** number of flip-flop and it is capable of storing an **n-bit** word.

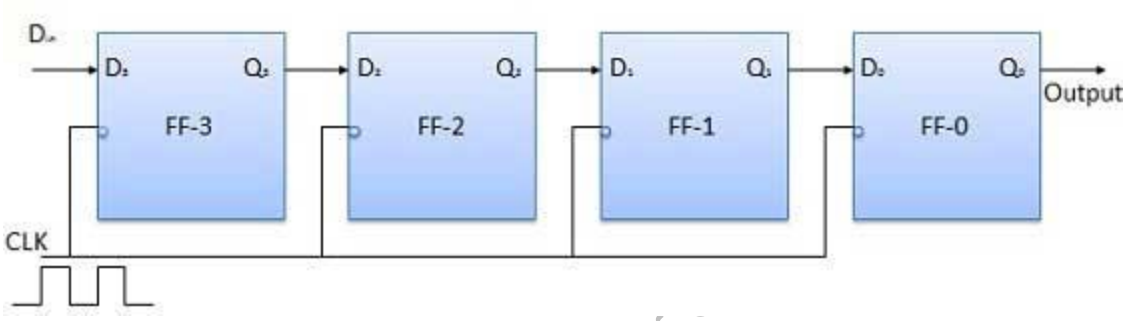
The binary data in a register can be moved within the register from one flip-flop to another. The regist transfers are called as **shift registers**. There are four mode of operations of a shift register.

- Serial Input Serial Output
- Serial Input Parallel Output
- Parallel Input Serial Output
- Parallel Input Parallel Output

Serial Input Serial Output

Let all the flip-flop be initially in the reset condition i.e. $Q_3 = Q_2 = Q_1 = Q_0 = 0$. If an entry of a four bit made into the register, this number should be applied to $\mathbf{D_{in}}$ bit with the LSB bit applied first. The D connected to serial data input $\mathbf{D_{in}}$. Output of FF-3 i.e. Q_3 is connected to the input of the next flip-flop i.e. D_2

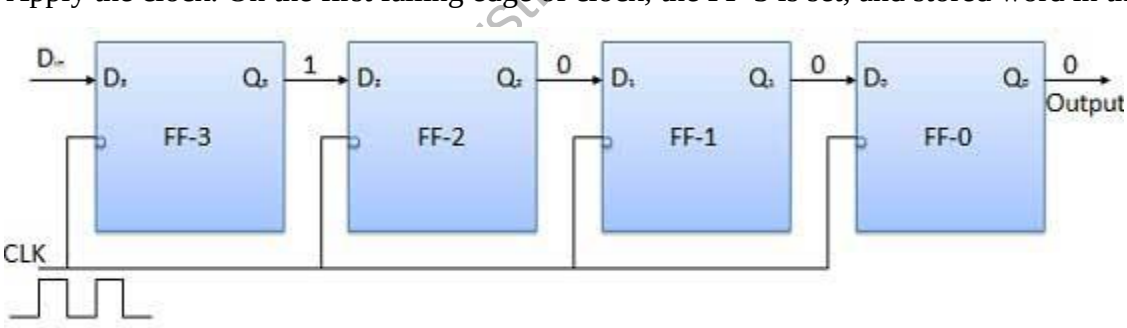
Block Diagram



Operation

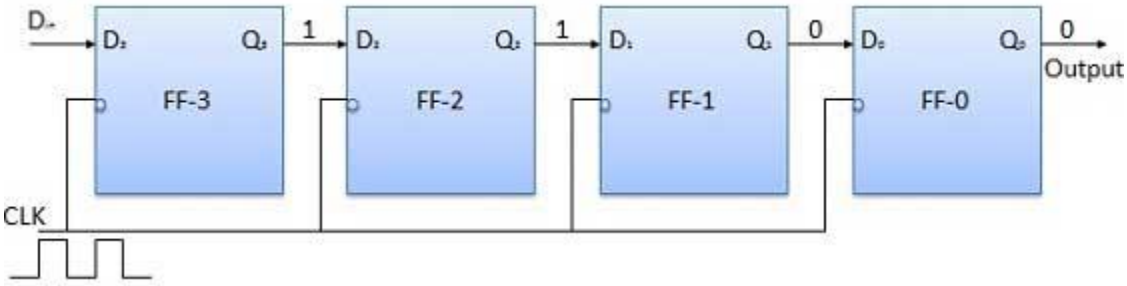
Before application of clock signal, let $Q_3\ Q_2\ Q_1\ Q_0 = 0000$ and apply LSB bit of the number to be entered

Apply the clock. On the first falling edge of clock, the FF-3 is set, and stored word in the register is $Q_3Q_2\ Q_1$

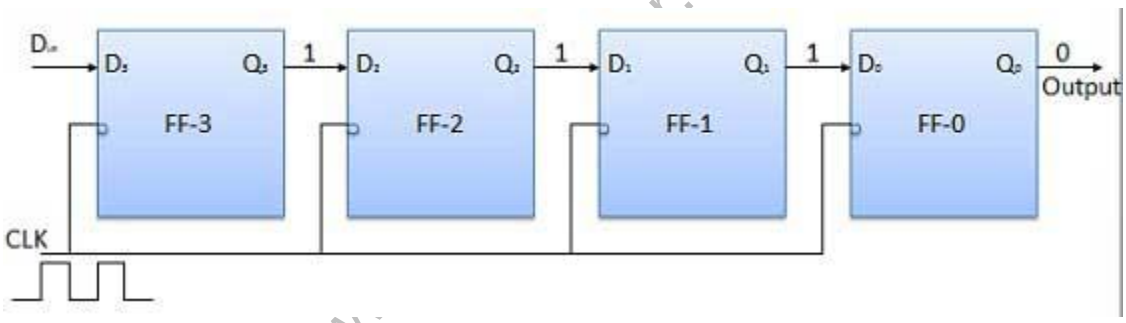


Apply the next bit to D_{in}. So $D_{in} = 1$. As soon as the next negative edge of the clock hits, FF-2 will set and t

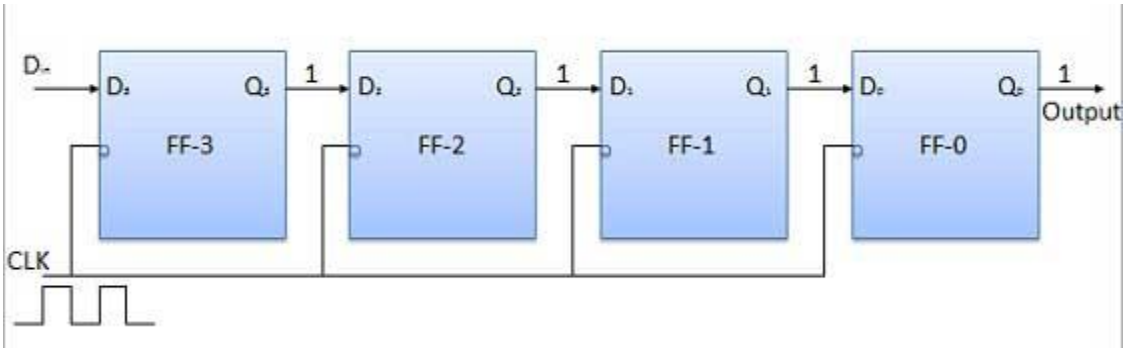
$Q_3\ Q_2\ Q_1\ Q_0 = 1100$.



Apply the next bit to be stored i.e. 1 to D_{in}. Apply the clock pulse. As soon as the third negative clock edge output will be modified to $Q_3\ Q_2\ Q_1\ Q_0 = 1110$.



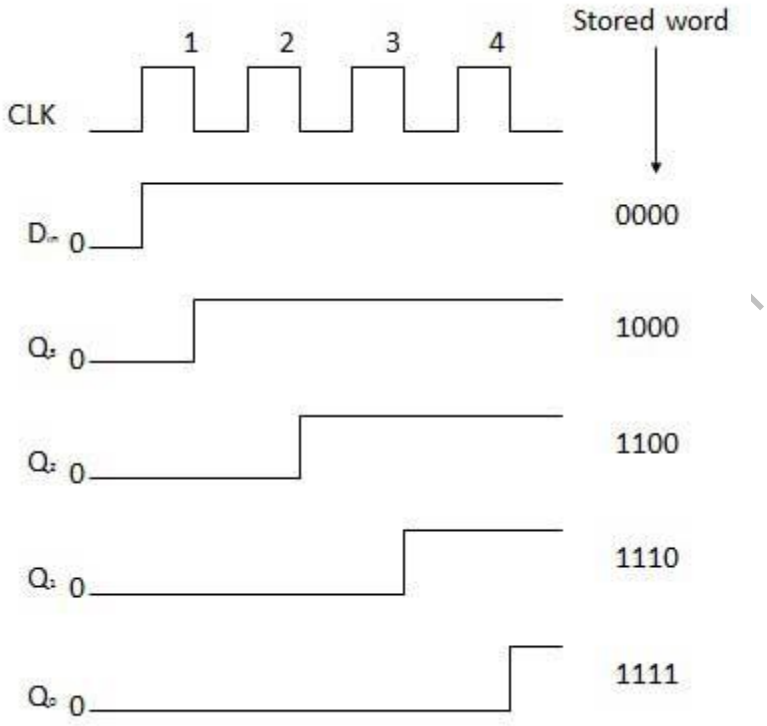
Similarly with $D_{in} = 1$ and with the fourth negative clock edge arriving, the stored word in the register is Q_3



TRUTH TABLE

	CLK	$D_n = Q_3$	$Q_3 = D_2$	$Q_2 = D_1$	$Q_1 = D_0$	Q_0
Initially			0	0	0	0
(i)	↓	1	1	0	0	0
(ii)	↓	1	1	1	0	0
(iii)	↓	1	1	1	1	0
(iv)	↓	1	1	1	1	1

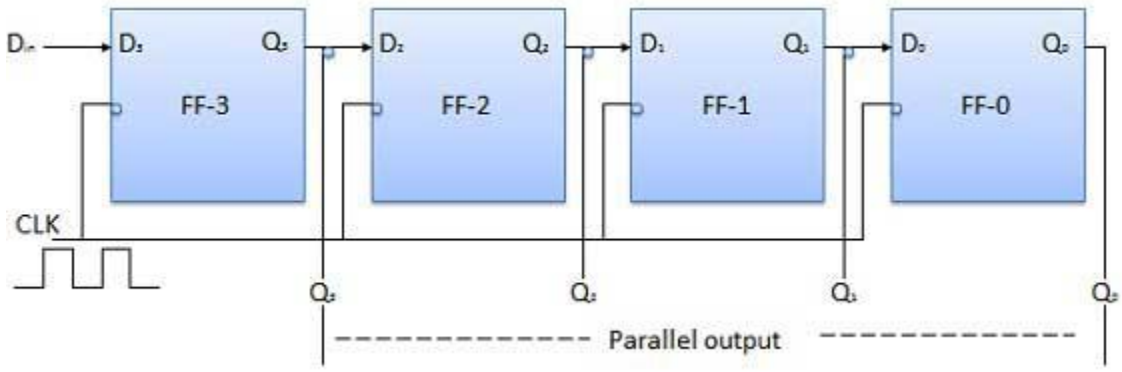
→ Direction of data travel



Serial Input Parallel Output

- In such types of operations, the data is entered serially and taken out in parallel fashion.
- Data is loaded bit by bit. The outputs are disabled as long as the data is loading.
- As soon as the data loading gets completed, all the flip-flops contain their required data, the outputs a loaded data is made available over all the output lines at the same time.
- 4 clock cycles are required to load a four bit word. Hence the speed of operation of SIPO mode is same as

Block Diagram



Parallel Input Serial Output (PISO)

- Data bits are entered in parallel fashion.
- The circuit shown below is a four bit parallel input serial output register.
- Output of previous Flip Flop is connected to the input of the next one via a combinational circuit.
- The binary input word B_0, B_1, B_2, B_3 is applied though the same combinational circuit.
- There are two modes in which this circuit can work namely - shift mode or load mode.

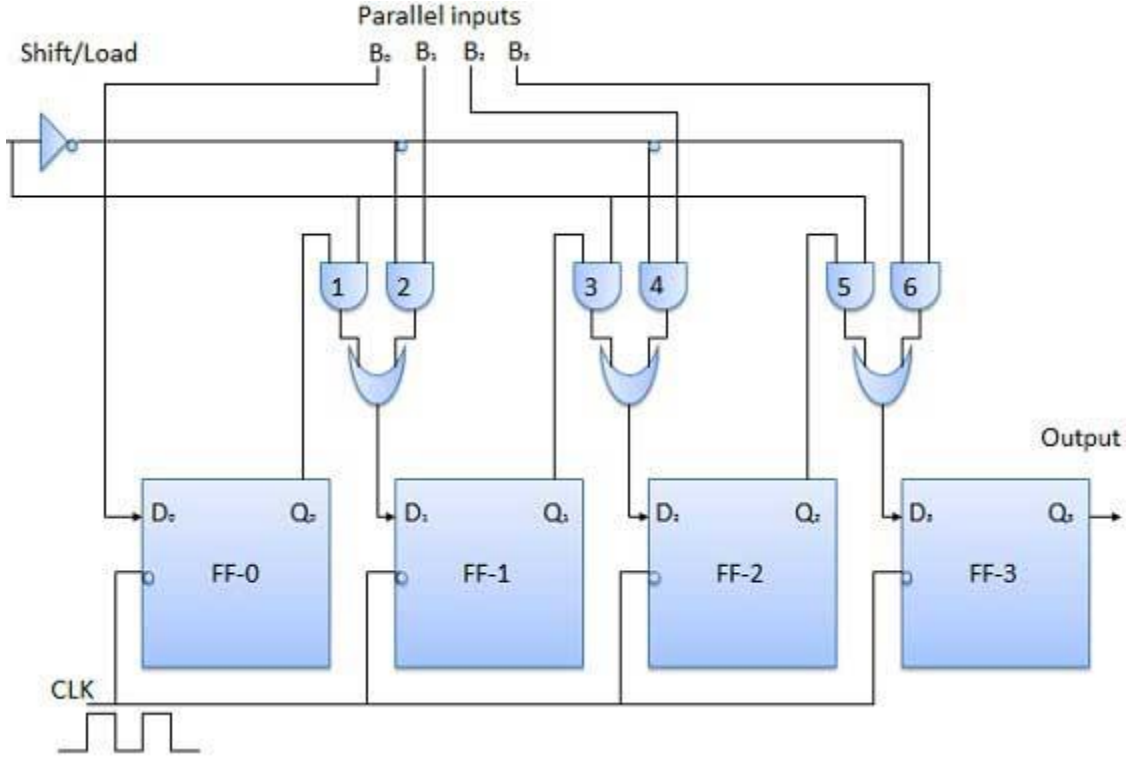
Load mode

When the shift/load bar line is low (0), the AND gate 2, 4 and 6 become active they will pass B_1, B_2, B_3 flip-flops. On the low going edge of clock, the binary input B_0, B_1, B_2, B_3 will get loaded into the corres parallel loading takes place.

Shift mode

When the shift/load bar line is low (1), the AND gate 2, 4 and 6 become inactive. Hence the parallel loa impossible. But the AND gate 1,3 and 5 become active. Therefore the shifting of data from left to right bi clock pulses. Thus the parallel in serial out operation takes place.

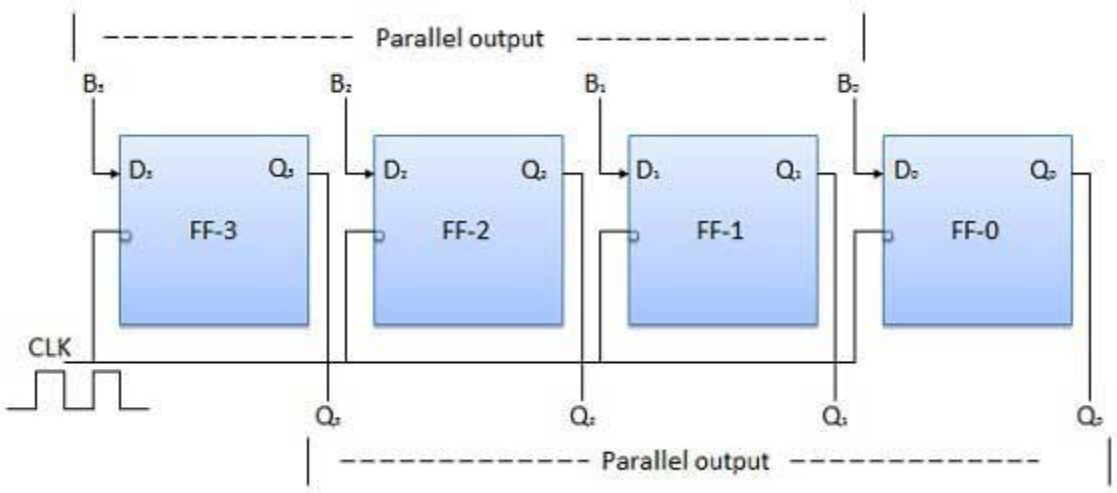
Block Diagram



Parallel Input Parallel Output (PIPO)

In this mode, the 4 bit binary input B_0, B_1, B_2, B_3 is applied to the data inputs D_0, D_1, D_2, D_3 respectively soon as a negative clock edge is applied, the input binary bits will be loaded into the flip-flops simultaneou appear simultaneously to the output side. Only clock pulse is essential to load all the bits.

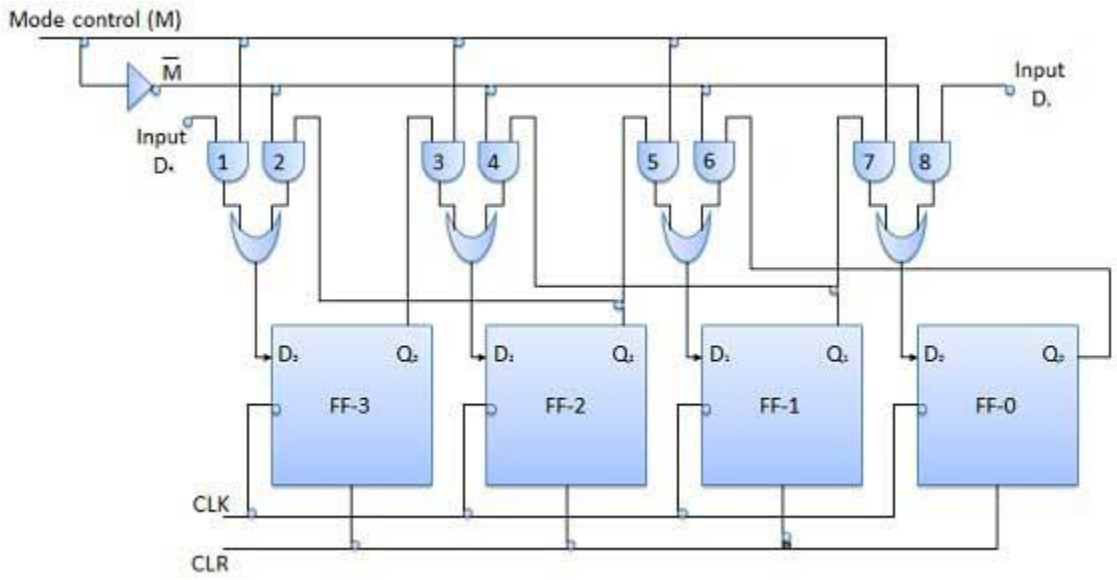
Block Diagram



Bidirectional Shift Register

- If a binary number is shifted left by one position then it is equivalent to multiplying the original number number is shifted right by one position then it is equivalent to dividing the original number by 2.
- Hence if we want to use the shift register to multiply and divide the given binary number, then we shoul in either left or right direction.
- Such a register is called bi-directional register. A four bit bi-directional shift register is shown in fig.
- There are two serial inputs namely the serial right shift data input DR, and the serial left shift data inp select input (M).

Block Diagram



Operation

S.N.	Condition	Operation
1	**With M = 1 – Shift right operation**	If M = 1, then the AND gates 1, 3, 5 and 7 are enabled whereas the remaining AND gates 2, 4, 6 and 8 will be disabled. The data at D_R is shifted to right bit by bit from FF-3 to FF-0 on the application of clock pulses. Thus with M = 1 we get the serial right shift operation.

2	**With M = 0 − Shift left operation**	When the mode control M is connected to 0 then the AND gates 2, 4, 6 and 8 are enabled while 1, 3, 5 and 7 are disabled. The data at D_L is shifted left bit by bit from FF-0 to FF-3 on the application of clock pulses. Thus with M = 0 we get the serial right shift operation.

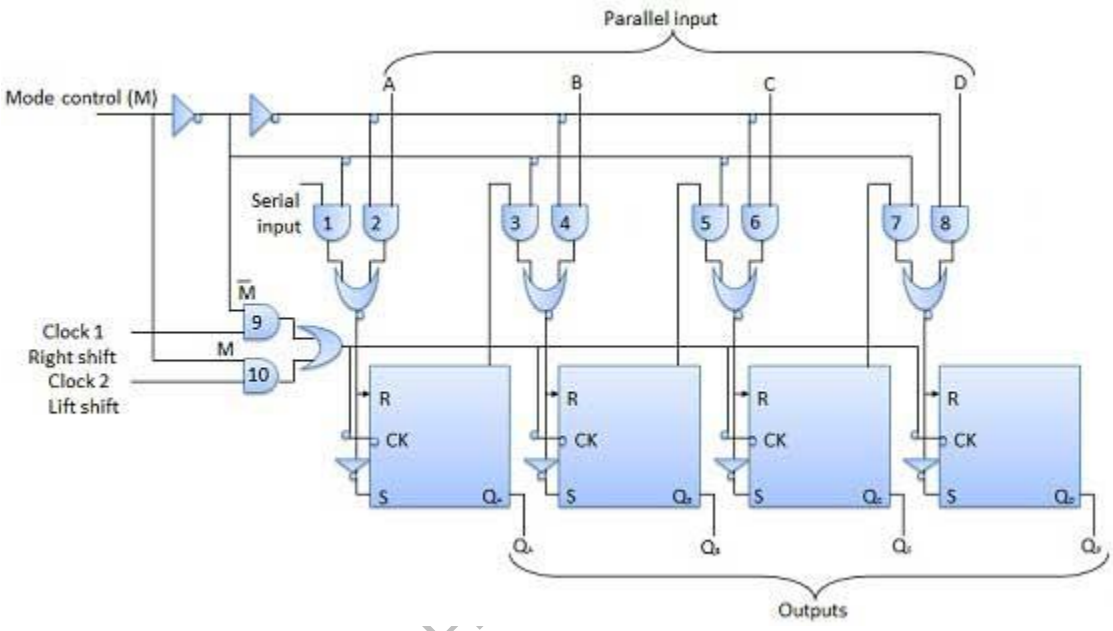
Universal Shift Register

A shift register which can shift the data in only one direction is called a uni-directional shift register. A shif the data in both directions is called a bi-directional shift register. Applying the same logic, a shift register w both directions as well as load it parallely, is known as a universal shift register. The shift register is ca following operation −

- Parallel loading
- Lift shifting
- Right shifting

The mode control input is connected to logic 1 for parallel loading operation whereas it is connected to 0 mode control pin connected to ground, the universal shift register acts as a bi-directional register. For serial l applied to the serial input which goes to AND gate-1 shown in figure. Whereas for the shift right operation, to D input.

Block Diagram



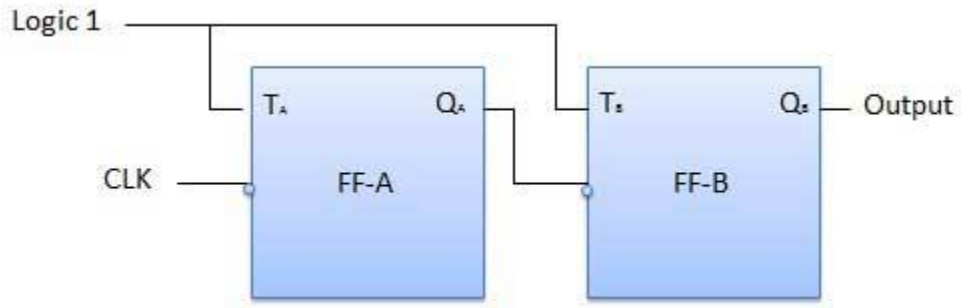
Counter is a sequential circuit. A digital circuit which is used for a counting pulses is known counte application of flip-flops. It is a group of flip-flops with a clock signal applied. Counters are of two types.

- Asynchronous or ripple counters.
- Synchronous counters.

Asynchronous or ripple counters

The logic diagram of a 2-bit ripple up counter is shown in figure. The toggle (T) flip-flop are being used. Bu
flop also with J and K connected permanently to logic 1. External clock is applied to the clock input of flip
applied to the clock input of the next flip-flop i.e. FF-B.

Logical Diagram



Operation

S.N.	Condition	Operation
1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will be equal to 1. Q_A is connected to clock input of FF-B. Since Q_A has changed from 0 to 1, it is treated as the positive clock edge by FF-

		B. There is no change in Q_B because FF-B is a negative edge triggered FF. Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A toggles again and Q_A = 0. The change in Q_A acts as a negative clock edge for FF-B. So it will also toggle, and $Q_B$will be 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On the arrival of 3rd negative clock edge, FF-A toggles again and Q_A become 1 from 0. Since this is a positive going change, FF-B does not respond to it and remains inactive. So Q_B does not change and continues to be equal to 1. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On the arrival of 4th negative clock edge,

		FF-A toggles again and Q_A becomes 1 from 0. This negative change in $Q_A$acts as clock pulse for FF-B. Hence it toggles to change $Q_B$from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

Truth Table

Clock	Counter output		State number	Deciimal Counter output
	Q_B	Q_A		
Initially	0	0	—	0
1st	0	1	1	1
2nd	1	0	2	2
3rd	1	1	3	3
4th	0	0	4	0

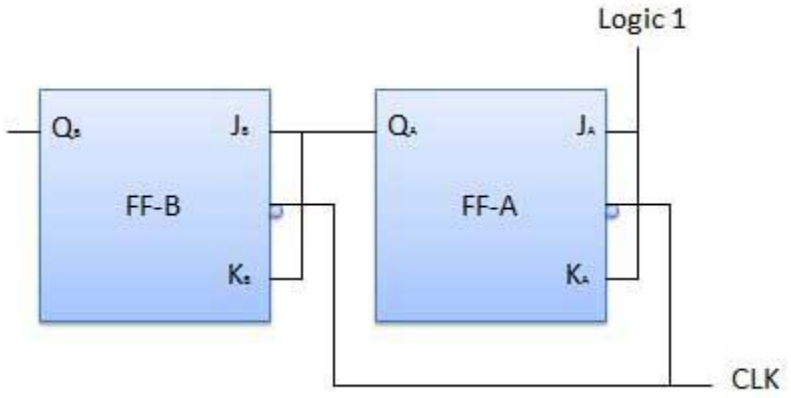
Synchronous counters

If the "clock" pulses are applied to all the flip-flops in a counter simultaneously, then such a counter is called

2-bit Synchronous up counter

The J_A and K_A inputs of FF-A are tied to logic 1. So FF-A will work as a toggle flip-flop. The J_B and K_B inpu

Logical Diagram



Operation

S.N.	Condition	Operation
1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially.
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will change from 0 to 1. But at the instant of application of negative clock edge, Q_A , J_B = K_B = 0. Hence FF-B will not change its state. So Q_B will remain 0.

		Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A toggles again and Q_A changes from 1 to 0. But at this instant Q_A was 1. So $J_B = K_B$= 1 and FF-B will toggle. Hence Q_B changes from 0 to 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On application of the third falling clock edge, FF-A will toggle from 0 to 1 but there is no change of state for FF-B. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On application of the next clock pulse, Q_A will change from 1 to 0 as Q_B will also change from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

CLASSIFICATION OF COUNTERS

Depending on the way in which the counting progresses, the synchronous or asynchronous counters are class

- Up counters
- Down counters
- Up/Down counters

UP/DOWN Counter

Up counter and down counter is combined together to obtain an UP/DOWN counter. A mode control (M) select either up or down mode. A combinational circuit is required to be designed and used between each pa achieve the up/down operation.

- Type of up/down counters
- UP/DOWN ripple counters
- UP/DOWN synchronous counter

UP/DOWN Ripple Counters

In the UP/DOWN ripple counter all the FFs operate in the toggle mode. So either T flip-flops or JK flip- LSB flip-flop receives clock directly. But the clock to every other FF is obtained from (Q = Q bar) output of

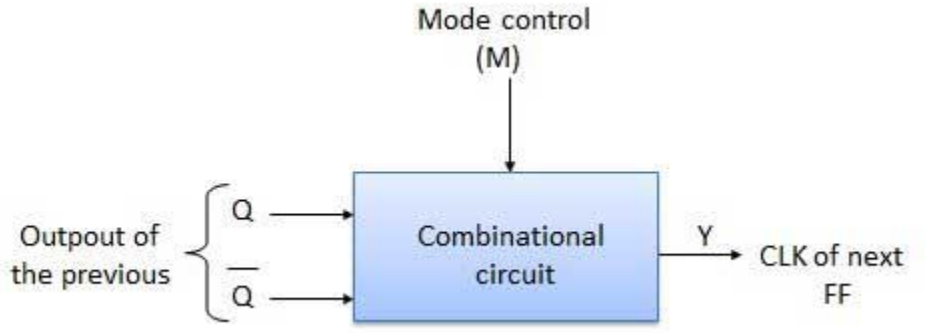
- **UP counting mode (M=0)** − The Q output of the preceding FF is connected to the clock of the next sta achieved. For this mode, the mode select input M is at logic 0 (M=0).
- **DOWN counting mode (M=1)** − If M = 1, then the Q bar output of the preceding FF is connected to the the counter in the counting mode.

Example

3-bit binary up/down ripple counter.

- 3-bit − hence three FFs are required.
- UP/DOWN − So a mode control input is essential.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple down counter, the Q bar output of preceding FF is connected to the clock input of the next one
- Let the selection of Q and Q bar output of the preceding FF be controlled by the mode control input M counting. So connect Q to CLK. If M = 1, DOWN counting. So connect Q bar to CLK.

Block Diagram



Truth Table

Inputs			Outputs	
M	Q	$\overline{Q}$	Y	
0	0	0	0	Y = Q for up counter
0	0	1	0	
0	1	0	1	
0	1	1	1	
1	0	0	0	$Y = \overline{Q}$ for up counter
1	0	1	1	
1	1	0	0	
1	1	1	1	

Operation

S.N.	Condition	Operation
1	**Case 1 − With M = 0 (Up counting mode)**	If M = 0 and M bar = 1, then the AND gates 1 and 3 in fig. will be enabled whereas the AND gates 2 and 4 will be disabled. Hence Q_A gets connected to the clock input of FF-B and $Q_B$gets connected to the clock input of FF-C. These connections are same as those for

		the normal up counter. Thus with M = 0 the circuit work as an up counter.
2	**Case 2: With M = 1 (Down counting mode)**	If M = 1, then AND gates 2 and 4 in fig. are enabled whereas the AND gates 1 and 3 are disabled. Hence Q_A bar gets connected to the clock input of FF-B and Q_B bar gets connected to the clock input of FF-C. These connections will produce a down counter. Thus with M = 1 the circuit works as a down counter.

Modulus Counter (MOD-N Counter)

The 2-bit ripple counter is called as MOD-4 counter and 3-bit ripple counter is called as MOD-8 counte ripple counter is called as modulo-N counter. Where, MOD number = 2^n.

Type of modulus

- 2-bit up or down (MOD-4)
- 3-bit up or down (MOD-8)
- 4-bit up or down (MOD-16)

APPLICATIONS OF COUNTERS

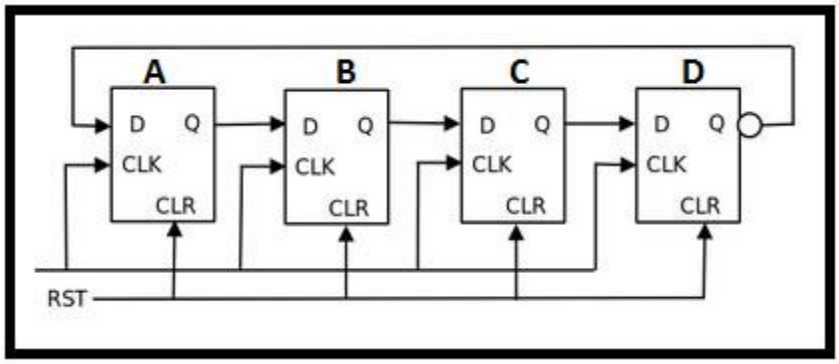
- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator.

Ring and Johnson Counter

Two most important types of shift register counters are Johnson counter and Ring counter. These shift register c
are connected to serial inputs to produce particular pattern of sequences. These shift registers are used as counte
sequence of states.

Ring Counter

Ring counter is a basic application of shift registers. It is formed by the feedback of the output to its own input
where N denotes the number of **flip-flops** in the ring counter.



Q_A	Q_B	Q_C	Q_D
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

A 4 bit ring counter circuit is shown in the figure above. It consists of 4 D-flipflops, FFA, FFB, FFC and FFD. has an input D and output Q. At first, a **CLEAR** signal is applied to the flip-flops to RESET the outputs to zero applied to the flip-flop FFA before the clock pulse is given. This step allows putting the value '1' to the ring coun the clock pulse is given, the counter circulates the data among all the four flip-flops. Modulo-4 or mod-4 counter i ring counter. Each output value of this counter has a frequency ¼ th of the main frequency value.

Advantages

- Can be implemented using D and JK flip-flops.
- It is a self-decoding circuit.

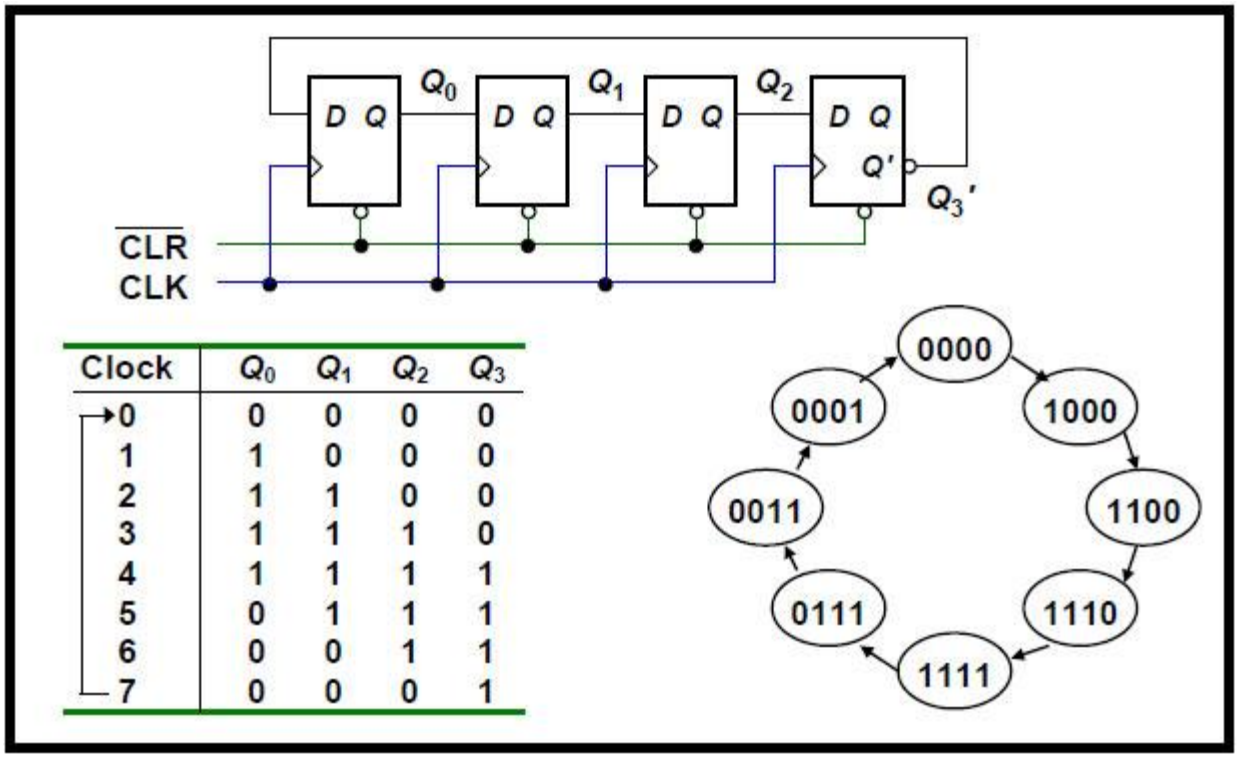
Disadvantages

- Only four of the 15 states are being utilized.

Johnson Counter

Johnson Counter also known as **Twisted Ring Counter** is another basic application of shift registers with a feed given from the inverted output of the last flip flop to the input of the first flip-flop.Figure below shows a 4-bit Joh

four flip-flops FF0, FF1, FF2 and FF3. Here the inverted output of the last flip-flop FF3 is given as feedback to th FF0. Here, at first four logic zeros will be passed to the flip-flops. When clock pulses are given "1000", "1100 "0011", "0001", "0000" outputs will be obtained and the sequence will repeat for the next clock pulses.



Clock	Q_0	Q_1	Q_2	Q_3
→0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

Advantages

- More outputs than ring counter.
- Disadvantages

- Only 8 of the 15 states are being used.