srivenkateshwaraa
College of Engineering & Technology
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University, Puducherry)
13-A, Pondy - Villupuram Main Road, Ariyur, Puducherry - 605 102.
ASPIRE TO EXCEL

## DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

# EC T45 DIGITAL CIRCUITS NOTES

## II YEAR/ IV SEM

# UNIT- I

**Number System:** Review of Binary, Octal and Hexadecimal Number Systems – Conversion methods. Number Representations – Signed Numbers and Complements, Unsigned, Fixed point, and Floating point numbers. Addition and subtraction with 1's and 2's complements.

**Codes:** Binary code for decimal numbers- Gray code-Codes for detecting and correcting errors: Even and Odd parity codes, Hamming Codes, Checksum codes, m-out–of-n-codes,

## NUMBER SYSTEM

### Introduction to Number Systems

- The number system we generally use in our everyday lives is a decimal place value system; that is, it is based on powers of ten: 1, 10, 100, 1000, etc.
- In the field of computer science, however, it is often useful to represent numbers in binary, octal, or hexadecimal notation.
- Table 1 below compares how the numbers from 0 through 24 are expressed in each of these number systems. Notice that, in decimal notation, we use ten different digits to express numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- In binary notation, we use just two different digits: 0 and 1.
- In octal notation, we use eight digits: 0, 1, 2, 3, 4, 5, 6, and 7; and in
- hexadecimal notation, we must come up with sixteen different digits to use: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

### Decimal Number System

- The decimal number system is a radix-10 number system and therefore has 10 different digits or symbols. These are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
- The place values of different digits in a mixed decimal number, starting from the decimal point, are $10^0$, $10^1$, $10^2$ and so on (for the integer part) and $10^{-1}$, $10^{-2}$, $10^{-3}$ and so on (for the fractional part). As an illustration, in the case of the decimal number 3586.265, the integer part (i.e. 3586) can be expressed as

$$3586 = 6\times10^0 + 8\times10^1 + 5\times10^2 + 3\times10^3 = 6+80+500+3000 = 3586$$

- and the fractional part can be expressed as

$$265 = 2\times10^{-1} + 6\times10^{-2} + 5\times10^{-3} = 0.2+0.06+0.005 = 0.265$$

### Binary Number System

- The binary number system is a radix-2 number system with '0' and '1' as the two independent digits.

- All larger binary numbers are represented in terms of ‘0’ and ‘1’.

## Octal Number System

- The octal number system has a radix of 8 and therefore has eight distinct digits. All higher-order numbers are expressed as a combination of these on the same pattern as the one followed in the case of the binary and decimal number systems

- The place values for the different digits in the octal number system are $8^0$, $8^1$, $8^2$ and so on (for the integer part) and $8^{-1}$, $8^{-2}$, $8^{-3}$ and so on (for the fractional part).

## Hexadecimal Number System

- The hexadecimal number system is a radix-16 number system and its 16 basic digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.
- The place values or weights of different digits in a mixed hexadecimal number are $16^0$, $16^1$, $16^2$ and so on (for the integer part) and 16–, $16^{-2}$, $16^{-3}$ and so on (for the fractional part).
- The decimal equivalent of A, B, C, D, E and F are 10, 11, 12, 13, 14 and 15 respectively, for obvious reasons.

## Octal and Hexadecimal Number Systems

- The conversion from and to binary. octal. and hexadecimal plays an important role in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.
- The first 16 numbers in the decimal, binary. octal, and hexadecimal number systems are listed in The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three digit s each. starting from the binary point and proceeding to the left and to the rig ht. The corresponding octal digit is then assigned to each group.
- The following example illustrates the procedure.
  $( 10\ 110\ 00\ 1\ 101\ 011\ .\ 111\ 100\ 000\ 110 )_2 = (26\ 153.7406)_8$
  2 6 1 5 3 7 4 0 6

Table1
Numbers with Different Bases

| **Decimal (base 10)** | **Binary** (base 2) | **Octal** (base 8) | **Hexadecimal** (base 16 ) |
|---|---|---|---|
| 00 | 0000 | 00 | 0 |
| 01 | 0001 | 01 | 1 |
| 02 | 0010 | 02 | 2 |
| 03 | 0011 | 03 | 3 |
| 04 | 0100 | 04 | 4 |
| 05 | 0101 | 05 | 5 |
| 06 | 0 110 | 06 | 6 |
| 07 | 0 111 | 07 | 7 |
| 08 | 1000 | 10 | 8 |
| 09 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| I5 | 1111 | 17 | F |

Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of four digits :
$( 10\ \ 1100\ \ 0110\ \ 1011\ .1111\ 0010)_2 = (2C6B.F2)_{16}$
2 C 6 B F

- The corresponding hexadecimal (or Octal) digit for each group of binary digits is easily remembered from the values listed in Table.
- Conversion from octal or hexadecimal to binary is done by reversing the preceding procedure Each octal digit is convened to its three-digit binary equivalent. Similarly, each hexadecimal digit is converted to its four-digit binary equivalent. The procedure is illus treated in the following examples;
  $(673.12.)_8$ =( 110 111 011 . 001 010 100$)_2$
  6 7 3 1 2 4

and

$(306.D)_{16}$ = (0001 0000 0110 .1101$)_2$

## Binary-to-Decimal Conversion

- The decimal equivalent of the binary number $(1001.0101)_2$ is determined as follows:
- The integer part = 1001
- The decimal equivalent = $1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 + 0 + 0 + 8 = 9$
- The fractional part = .0101
- Therefore, the decimal equivalent = $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2{-4}^4 =$
  $0 + 0.25 + 0 + 0.0625 = 0.3125$
- Therefore, the decimal equivalent of $(1001.0101)_2 = 9.3125$

## Octal-to-Decimal Conversion

- The decimal equivalent of the octal number $(137.21)_8$ is determined as follows
- The integer part = 137
- The decimal equivalent = $7 \times 8^0 + 3 \times 8^1 + 1 \times 8^2 = 7 + 24 + 64 = 95$
- The fractional part = .21
- The decimal equivalent = $2 \times 8^{-1} + 1 \times 8^{-2} = 0.265$
- Therefore, the decimal equivalent of $(137.21)_8 = (95.265)_{10}$

## Hexadecimal-to-Decimal Conversion

The decimal equivalent of the hexadecimal number $(1E0.2A)_{16}$ is determined as follows

- The integer part = 1E0 The decimal equivalent = $0 \times 16^0 + 14 \times 16^1 + 1 \times 16^2 = 0 + 224 + 256 = 480$
- The fractional part = .2A
- The decimal equivalent = $2 \times 16^{-1} + 10 \times 16^{-2} = 0.164$
- Therefore, the decimal equivalent of $(1E0.2A)_{16} = (480.164)_{10}$

## Decimal-to-Binary Conversion

- This method of decimal–binary conversion is popularly known as the double-dabble method.
- We will find the binary equivalent of $(13.375)_{10}$.

**Solution**

The integer part = 13

| Divisor | Dividend | Remainder |
|---|---|---|
| 2 | 13 | — |
| 2 | 6 | 1 |
| 2 | 3 | 0 |
| 2 | 1 | 1 |
| — | 0 | 1 |

- The binary equivalent of $(13)_{10}$ is therefore $(1101)_2$

The fractional part = .375

$0.375 \times 2 = 0.75$ with a carry of 0
$0.75 \times 2 = 0.5$ with a carry of 1
$0.5 \times 2 = 0$ with a carry of 1

The binary equivalent of $(0.375)_{10} = (.011)2$
Therefore, the binary equivalent of $(13.375)_{10} = (1101.011)_2$

## Decimal-to-Octal Conversion

- The process of decimal-to-octal conversion is similar to that of decimal-to-binary conversion.
- The progressive division in the case of the integer part and the progressive multiplication while working on the fractional part here are by '8' which is the radix of the octal number system.

## Example

We will find the octal equivalent of $(73.75)_{10}$

**Solution**

• The integer part = 73

| Divisor | Dividend | Remainder |
|---|---|---|
| 8 | 73 | — |
| 8 | 9 | 1 |
| 8 | 1 | 1 |
| — | 0 | 1 |

• The octal equivalent of $(73)_{10} = (111)_8$

- The fractional part = 0.75
- $0.75 \times 8 = 0$ with a carry of 6
- The octal equivalent of $(0.75)_{10} = (.6)_8$
- Therefore, the octal equivalent of $(73.75)_{10} = (111.6)_8$

## Decimal-to-Hexadecimal Conversion

- The process of decimal-to-hexadecimal conversion is also similar. Since the hexadecimal number system has a base of 16, the progressive division and multiplication factor in this case is 16.

  **Example**

  Let us determine the hexadecimal equivalent of $(82.25)_{10}$

  **Solution**

  The integer part = 82

  | Divisor | Dividend | Remainder |
  |---|---|---|
  | 16 | 82 | — |
  | 16 | 5 | 2 |
  | — | 0 | 5 |

  The hexadecimal equivalent of $(82)_{10} = (52)_{16}$

  The fractional part = 0.25

  $0.25 \times 16 = 0$ with a carry of 4

  Therefore, the hexadecimal equivalent of $(82.25)_{10} = (52.4)_{16}$

## Binary–Octal and Octal–Binary Conversions

- An octal number can be converted into its binary equivalent by replacing each octal digit with its three-bit binary equivalent.
- We take the three-bit equivalent because the base of the octal number system is 8 and it is the third power of the base of the binary number system is 2.

## Example

Let us find the binary equivalent of $(374.26)_8$ and the octal equivalent of $(1110100.0100111)_2$

- **Solution**

  The given octal number = $(374.26)_8$

  The binary equivalent = $(011\ 111\ 100.010\ 110)_2 = (011111100.010110)_2$

  - Any 0s on the extreme left of the integer part and extreme right of the fractional part of the equivalent binary number should be omitted.
  - Therefore, $(011111100.010110)_2 = (11111100.01011)_2$
  - The given binary number = $(1110100.0100111)_2$
  - $(1110100.0100111)_2 = (1\ 110\ 100.010\ 011\ 1)_2 = (001\ 110\ 100.010\ 011\ 100)_2 = (164.234)_8$

## Hex–Binary and Binary–Hex Conversions

- A hexadecimal number can be converted into its binary equivalent by replacing each hex digit with its four-bit binary equivalent.
- We take the four-bit equivalent because the base of the hexadecimal number system is 16 and it is the fourth power of the base of the binary number system.

**Example**

Let us find the binary equivalent of $(17E.F6)_{16}$ and the hex equivalent of $(1011001110.011011101)_2$.

**Solution**

• The given hex number = $(17E.F6)_{16}$
• The binary equivalent = $(0001\ 0111\ 1110.1111\ 0110)_2$
= $(000101111110.11110110)_2$
= $(101111110.1111011)_2$

The 0s on the extreme left of the integer part and on the extreme right of the fractional part have been omitted.

The given binary number = $(1011001110.011011101)_2$
= $(10\ 1100\ 1110.0110\ 1110\ 1)_2$

The hex equivalent = $(0010\ 1100\ 1110.0110\ 1110\ 1000)_2$ = $(2CE.6E8)_{16}$

## Hex–Octal and Octal–Hex Conversions

- For hexadecimal–octal conversion, the given hex number is firstly converted into its binary equivalent which is further converted into its octal equivalent.

**Example**

Let us find the octal equivalent of (2F.C4)16 and the hex equivalent of $(762.013)_8$.

**Solution**

The given hex number = $(2F.C4)_{16}$.
The binary equivalent = $(0010\ 1111.1100\ 0100)_2$ = $(00101111.11000100)_2$
= $(101111.110001)_2$= $(101\ 111.110\ 001)_2$ = $(57.61)_8$.
The given octal number = $(762.013)_8$.
The octal number = $(762.013)_8$ = $(111\ 110\ 010.000\ 001\ 011)_2$
= $(111110010.000001011)_2$
= $(0001\ 1111\ 0010.0000\ 0101\ 1000)_2$= $(1F2.058)_{16}$.

**Number Representation in Binary:**

Different formats used for binary representation of both positive and negative decimal numbers include the sign-bit magnitude method, the 1’s complement method and the 2’s complement method.

**Sign-Bit Magnitude:**

In the sign-bit magnitude representation of positive and negative decimal numbers, the MSB represents the ‘sign’, with a ‘0’ denoting a plus sign and a ‘1’ denoting a minus sign. The remaining bits represent the magnitude. In eight-bit representation, while MSB represents the sign, the remaining seven bits represent the magnitude. For example, the eight-bit representation

of +9 would be 00001001, and that for −9 would be 10001001. An n−bit binary representation can be used to represent decimal numbers in the range of −(2n−1−1) to +(2n−1−1). That is, eight-bit representation can be used to represent decimal numbers in the range from −127 to +127 using the sign-bit magnitude format.

## BINARY CODE:

Binary codes are codes which are represented in binary system with modification from the original ones. Below we will be seeing the following:

- Weighted Binary Systems
- Non Weighted Codes

**Weighted Binary system:**

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example

| Decimal | 8421 | 2421 | 5211 | Excess-3 |
|---|---|---|---|---|
| 0 | 0000 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0010 | 0011 | 0101 |
| 3 | 0011 | 0011 | 0101 | 0110 |
| 4 | 0100 | 0100 | 0111 | 0111 |
| 5 | 0101 | 1011 | 1000 | 1000 |
| 6 | 0110 | 1100 | 1010 | 1001 |
| 7 | 0111 | 1101 | 1100 | 1010 |
| 8 | 1000 | 1110 | 1110 | 1011 |
| 9 | 1001 | 1111 | 1111 | 1100 |

## 8421 Code/BCD Code:

- The BCD (Binary Coded Decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8,4,2,1. In BCD, a digit is usually represented by four bits which, in general, represent the values/digits/characters 0–9. Other bit combinations are sometimes used for a sign or other indications.
- Its main virtue is that it allows easy conversion to decimal digits for printing or display, and allows faster decimal calculations. Its drawbacks are a small increase in the complexity of circuits needed to implement mathematical operations. Uncompressed

BCD is also a relatively inefficient encoding—it occupies more space than a purely binary representation.

**Example:** The bit assignment 1001, can be seen by its weights to represent the decimal 9 because: 1x8+0x4+0x2+1x1 = 9

**Non Weighted Codes:**

- Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value

## Excess-3 Code:

- Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).
- Excess-3 binary-coded decimal (XS-3), also called biased representation or Excess-N, is a numeral system used on some older computers that uses a pre-specified number N as a biasing value.
- It is a way to represent values with a balanced number of positive and negative numbers. In XS-3, numbers are represented as decimal digits, and each digit is represented by four bits as the BCD value plus 3 (the "excess" amount):
- The smallest binary number represents the smallest value. (i.e. 0 − Excess Value) The greatest binary number represents the largest value. (i.e. 2N − Excess Value − 1) The primary advantage of XS-3 coding over BCD coding is that a decimal number can benines' complemented (for subtraction) as easily as a binary number can be ones' complemented; just invert all bits.
- Adding Excess-3 works on a different algorithm than BCD coding or regular binary numbers. When you add two XS-3 numbers together, the result is not an XS-3 number. For instance, when you add 1 and 0 in XS-3 the answer seems to be 4 instead of 1. In order to correct this problem, when you are finished adding each digit, you have to subtract 3 (binary 11) if the digit is less than decimal 10 and add three if the number is greater than or equal to decimal 10.

**Example:** 1000 of 8421 = 1011 in Excess-3

## Gray Code:

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code. In digital Gray code has got a special place.

**USES:**

- A typical use of Gray code counters is building a FIFO (first-in, first-out) data buffer that has read and write ports that exist in different clock domains.
- Gray codes are used in position encoders (linear encoders and rotary encoders), in preference to straightforward binary encoding. This avoids the possibility that, when several bits change in the binary representation of an angle, a misread could result from some of the bits changing before others. Rotary encoders benefit from the cyclic nature of Gray codes, because the first and last values of the sequence differ by only one bit.
- Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

| 2-bit Gray code<br><br>00<br>01<br>11<br>10 |
|---|
| 3-bit Gray code<br>000<br>001<br>011<br>010<br>110<br>111<br>101<br>100 |
| 4-bit Gray code<br>0000<br>0001<br>0011<br>0010<br>0110<br>0111<br>0101<br>0100<br>1100<br>1101<br>1111<br>1110<br>1010<br>1011<br>1001<br>1000 |

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

## Excess 3 Gray code:

- In many applications, it is desirable to have a code that is BCD as well as unit distance. A unit distance code derives its name from the fact that there is only one bit change between two consecutive numbers.
- The excess 3 gray code is such a code, the values for zero and nine differ in only 1 bit, and so do all values for successive numbers. Outputs from linear devices or angular encoders may be coded in excess 3 gray code to obtain multi-digit BCD numbers.

| Decimal | Gray |
|---|---|
| **0** | 0010 |
| **1** | 0110 |
| **2** | 0111 |
| **3** | 0101 |
| **4** | 0100 |
| **5** | 1100 |
| **6** | 1101 |
| **7** | 1111 |
| **8** | 1110 |
| **9** | 1010 |

## BCD TO EXCESS 3 CODE CONVERSION:

| TRUTH TABLE | | | | | | | |
|---|---|---|---|---|---|---|---|
| Input (BCD) | | | | Output (Excess-3) | | | |
| **A** | **B** | **C** | **D** | **W** | **X** | **Y** | **Z** |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

## SIGNED NUMBERS:

- In real life we have to represent signed numbers ( like: -12, -45, 78).
- The difference between signed and unsigned numbers is the sign.
- A scheme is needed to represent the sign as part of the binary representation.
- There are a number of schemes for representing signed numbers in binary format.
  - sign-magnitude representation
  - The two's-complement representation.

SIGN-MAGNITUDE REPRESENTATION:

- In this representation, the leftmost bit of a binary code represents the sign of the value:
  - 0 for positive,
  - 1 for negative;

  the remaining bits represent the numeric value.
- To Compute negative values using Sign/Magnitude (signmag) representation.
- Begin with the binary representation of the positive value, then flip the leftmost zero bit.

**Ex 1. Find the signmag representation of $-6_{10}$**

Step1: find binary representation using 8 bits

$6_{10} = 00000110_2$

Step2: if the number is a negative number flip left most bit

10000110

So: $-6_{10} = 10000110_2$ (in 8-bit sign/magnitude form)

**Ex 2. Find the signmag representation of $-36_{10}$**

Step 1: find binary representation using 8 bits

$36_{10} = 00100100_2$

Step 2: if the number is a negative number flip left most bit

10100100

So: $-36_{10} = 10100100_2$ (in 8-bit sign/magnitude form)

**Ex 3. Find the signmag representation of $70_{10}$**

Step 1: find binary representation using 8 bits

$70_{10} = 01000110_2$

Step 2: if the number is a negative number flip left most bit

01000110 (no flipping, since it is +ve)

So: $70_{10} = 01000110_2$ (in 8-bit sign/magnitude form)

## TWO'S COMPLEMENT REPRESENTATION:

- Another scheme to represent negative numbers
- The leftmost bit serves as a sign bit:

  0 for positive numbers,

  1 for negative numbers.

- To Compute negative values using two's Complement representation, begin with the binary representation of the positive value, complement (flip each bit if it is 0 make it 1 and visa versa) the entire positive number, and then add one.

## Ex. Find the two's complement representation of $-6_{10}$

Step1: find binary representation in 8 bits

$$6_{10} = 00000110_2$$

Step 2: Complement the entire positive number, and then add one

```
                          00000110
(complemented)     ->     11111001
     (add one) ->       +        1
                        ----------
                          11111010
```

So: $-6_{10} = 11111010_2$ (in 2's complement form, using any of above methods)

## ALTERNATIVE METHOD FOR STEP 2

- Scan binary representation from right too left, find first one bit, from low-order (right) end, and complement the remaining pattern to the left.

```
                             00000110
(left complemented)  -->     11111010
```

## Ex 2: Find the Two's Complement of $-76_{10}$

Step 1: Find the 8 bit binary representation of the positive value.

$76_{10} = 01001100_2$

Step 2: Find first one bit, from low-order (right) end, and complement the pattern to the left.

```
                          01001100
(left complemented) ->    10110100
```

So: $-76_{10} = 10110100_2$ (in 2's complement form, using any of above methods)

## EVEN/ODD PARITY:

- Computers can sometimes make errors when they transmit data.

**EVEN/ODD PARITY:**

- is basic method for detecting if an odd number of bits has been switched by accident.

**ODD PARITY:**

- The number of 1-bit must add up to an odd number

**EVEN PARITY:**

- The number of 1-bit must add up to an even number

# UNIT – II

**Boolean Algebra:** Basic theorems- Postulates- Duality – Canonical form.

**Simplification of Boolean Function:** Karnaugh map method – Incompletely specified functions. Realization of logic functions - NAND gate realization - NOR gate realization - Multilevel synthesis

## Boolean Algebra:

- Boolean algebra. like any other deductive mathematical system. may be defined with a Set of elements. a set of operators. and a number of unproved axioms or postulates.
- A *set* of elements is any collection of object s. usually having a common property. If S is a set, and *x* and y are certain objects. then *x* € S means that *x* is a member of the set S and *y* ₡ S means that Y is not an element of S.
- A set with a denumerable number of elements is specified by braces:
- A = {1, 2, 3, 4 } indicates that the elements of set A are the numbers 1, 2, 3, and 4.
- A binary operator defined on a set S of elements is a rule that assigns, to each pair of elements from S,
- a unique element from S. As an example, consider the relation a * b = c. We say that is a binary operator if it specifies a rule for finding c from the pair (a , b) and also if a, b, C € S. However,* is not a binary operator if a , b € , if C ₡ S.

## Boolean Postulates

- A Boolean algebra is an algebra consisting of a set B (which consists of at least two values 0 and 1) together with three operations, the AND (Boolean product) operation•, the OR (Boolean sum) operation +, and NOT (Complement) operation defined on the set, such that for any objects/elements x, y and z of B, x ∈ B denotes that x is an element of the set B, and y /∈ B denotes that y is not an element of B.

  AND operator x • y (the product of x and y)
  OR operator x + y (the sum of x and y)
  NOT operator $x'$ or $x'$ (the complement of x)
- Here + and • are binary operators and (not) is a unary operators the postulates of mathematical system form the assumptions from which it is possible to deduce the rules, theorems and properties of the system. For Boolean algebra, the following postulates/axioms are used.

## Postulate 1: Closure

- A set B is closed with respect to a binary operator, if for every pair of elements of
- B, the binary operator specifies a rule for obtaining a unique element of B.

For example:

- the set of natural numbers N = {1,2,3,4, · · ·} is closed with respect to the binary operator plus (+) by the rules of arithmetic addition, since
- for any a , b ∈ N we obtain a unique c ∈ N by the operation a+b = c.

## Postulate 2: Associative Law

- A binary operator * (or • ) and + on a set *B* are said to be associative whenever
  (x•y)•z = x•(y •z) :law for multiplication
  (x+y)+z = x+(y+z) :law for addition

## Postulate 3: Commutative Law

- The binary operator + and• on a set *B* are said to be commutative whenever
  x • y = y • x : law for multiplication
  x+y = y+x : law for addition

## Postulate 4: Identity element

- A set B is said to have an identity element with respect to binary operator + and ∗ on B if there exists an element c ∈ B with the property
  c ∗ x = x ∗ c = x for every c ∈ B

## Postulate 5: Inverse

- A set B having the identity element e with respect to a binary operator is said to have an inverse whenever for every x ∈ B, there exists an element y ∈ B such that,
  x • y = e

## Postulate 6: Distributive Law

- A Binary operator • in a set B, is said to be distributive over + whenever
  x • (y+z) = x • y+x • z for x, y, z ∈ B
  x+(y • z) = (x+y) • (x+z)

## Postulate 7: Absorptive Law

- The binary operators • and + on a set B are said to be absorptive whenever

  $x \bullet (x+y) = x$

  $x+(x \bullet y) = x$

## Postulate 8: Union Law

- Zero (null, smallest) and one (universal, largest) elements: there exists a unique elements (the one element) $1 \in B$ such that for each element $x \in B$

  $x \bullet 1 = 1 \bullet x = x$

- There exists a unique element (the zero) $0 \in B$ such that for each element $x \in B$

  $X + 0 = 0 + x = x$

## Postulate 9: complementary

- For each $x \in B$ there exists a unique element $x \in B$ called the complement of x such that

  $x \bullet x' = 0$

  $x + x' = 1$


## Postulate 7: Absorptive Law

- The binary operators • and + on a set B are said to be absorptive whenever

  $x \bullet (x+y) = x$

  $x+(x \bullet y) = x$

## Postulate 8: Union Law

- Zero (null, smallest) and one (universal, largest) elements: there exists a unique elements (the one element) $1 \in B$ such that for each element $x \in B$

  $x \bullet 1 = 1 \bullet x = x$

- There exists a unique element (the zero) $0 \in B$ such that for each element $x \in B$

  $X + 0 = 0 + x = x$

## Postulate 9: complementary

- For each $x \in B$ there exists a unique element $x \in B$ called the complement of x such that

  $x \bullet x' = 0$

  $x + x' = 1$


## Huntington Postulates

- E.V Huntington has formulated the following postulates for Boolean algebra which is set B elements a,b,c and has two operators + and • .

  (a) Closure with respect to the operator +

  (b) Closure with respect to the operator•

- Identity element with respect to + , designated by 0 :
  $x+0 = 0+x = x$
- (b) Identity element with respect to • , designated by 1:
  $x \bullet 1 = 1 \bullet x = x$
- (a) commutative with respect to operator +
  $x+y = y+x$
  (b) Commutative with respect to operator •
  $x \bullet y = y \bullet x$
- (a) • is distributive over +
  $x \bullet (y+z) = (x \bullet y)+(x \bullet z)$
- (b) + is distributive over •
  $x+(y \bullet z) = (x+y) \bullet (x+z)$
- For every element x ∈ B, there exists an element x ∈ B (complement of x)
- such that
  $x+x' = 1$
  $x \bullet x' = 0$
- There exist at least two elements x, y ∈ B such that x ≠ y.

## Two - Valued Boolean algebra

- The two - valued Boolean algebra is defined on a set of two elements B = {0,1}, with rules for the two binary operators + and • (OR, AND logic function) and inversion shown in the following table 1.1

**Table 1.1(a)**

| AND function | | |
|---|---|---|
| x | y | x•y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table 1.1(b)**

| OR function | | |
|---|---|---|
| x | y | x+y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Table 1.1(c)**

| Inversion function | |
|---|---|
| x | $\overline{x}$ |
| 0 | 1 |
| 1 | 0 |

Validation of Huntington postulate for two-valued set B = {0,1} are

1. Closure is obvious form tables since the result of each operation is either 1 or 0 and 1, 0 ∈ B.
2. Form tables, we can see that

   (a) $0+0 = 0$
   $0+1 = 1+ \quad = 1$ 0→identity

   (b) $1 \bullet 1 = 1$
   $1 \bullet 0 = 0 \bullet 1 \quad = 0$ 1→identity

Which establish the two identity elements, 0 for + and 1 for ., as defined by postulate 2.

The commutative and distributive laws are proved as shown in the

**Table 1.2**

| x | y | z | x+y | x+z | x+(y.z) | (x+y).(x+z) | (x.y) | (x.z) | x.y+x.z |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Complement operation is

1. $x+x' = 1$ since $0+0' = 0+1 = 1$
   $1+1' = 1+0 = 1$
2. $x \bullet x' = 0$ since $0 \bullet 0' = 0 \bullet 1 = 0$
   $1 \bullet 1' = 1 \bullet 0 = 0$

## Basic Theorems and Properties of Boolean AlgebraDuality theorem

- The duality theorem states that starting with a Boolean relation we can derive another Boolean relation by
- Changing OR (operation) i.e., +(plus) sign to an and (operation) i.e., •(dot) and vice - versa
- Complement any 0 or 1 appearing in the expression i.e., replacing contains 0 and 1 by 1 and 0 respectively
- Any expression has this property called dual and some dual identity are given below.

| S.No | Given Expression | Dual |
|---|---|---|
| 1 | $0' = 1$ | $1' = 0$ |
| 2 | A.0 = 0 | A+1 = 1 |
| 3 | A.(B.C)=(A.B).C | A+ (B +C) = (A+B)+C |
| 4 | A.(A.B)= A.B | A+ A+B = A+B |
| 5 | A(A+B) = A | A + AB = A |

## Basic Theorems

- The theorems and postulates listed in table 1.3 are most basic relationships in Boolean algebra.
- The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates.

Table 1.3

| | | |
|---|---|---|
| Postulate 2 | (a) $x+0=x$ | (b) $x.1=x$ |
| Postulate 5 | (a) $x+\bar{x}=1$ | (b) $x.\bar{x}=0$ |
| Theorem 1 | (a) $x+x=x$ | (b) $x.x=x$ |
| Theorem 2 | (a) $x+1=1$ | (b) $x.0=0$ |
| Theorem 3, involution | (a) $\overline{\overline{(x)}}=x$ | |
| Postulate 3,Commutative | (a) $x+y=y+x$ | (b) $xy=yx$ |
| Postulate 4, Associative | (a) $x+(y+z)=(x+y)+z$ | (b) $x(yz)=(xy)z$ |
| Postulate 4, Distributive | (a) $x(y+z)=xy+xz$ | (b) $x+yz=(x+y).(x+z)$ |
| Postulate 5, Demorgan | (a) $\overline{x+y}=\bar{x}+\bar{y}$ | (b) $\overline{x.y}=\bar{x}+\bar{y}$ |
| Postulate 4, absorption | (a) $x+xy=x$ | (b) $x(x+y)=x$ |

**Proof of theorem - 1(a):** x+x = x

x+x = (x+x) • 1
= (x+x) • (x+x') [x+x' = 1]
= x • x+x' • x+x • x+x • x'
= x+0+x+0 ∵ x • x = x, x • x' = 0
= x+x
= x

**Proof of theorem 1(b):** x . x = x

x • x = xx+0

(addition of zero does not change the equation)

= xx+xx'
= x(x+x') [x+x' = 1]
= x(1)
= x

**Proof of theorem 2(a):** x+1 = 1

x+1 = 1 • (x+1) [∵ 1+x = 1]
= (x+x)(x+1) [x+x = 1]
= x x+x + x'x+x'
= x+x+0+x [x+x' = x]
= x+0+x'
= 1+0 [x+x = 1]
= 1

**Proof of theorem 6(a):** x+xy = x

x+xy = x+xy

= x(1+y)        [1+y = 1]

= x • 1

= x

## DeMorgan's Theorems

- The DeMorgan's theorems provide mathematical verification as follows.

### Theorem 1

- It states that the complement of a product of variables is equal to the sum of the complements of the variables.

A'•B' = A'+B'

### Theorem 2

- It states that the complement of a sum of variables is equal to the product of the complements of the variables.

A'+B' = A'•B'

## Proof of theorem 1:

| A | B | $\overline{A.B}$ | $\overline{A}$ | $\overline{B}$ | $\overline{A}+\overline{B}$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |

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

- From the above table, we can observe that the column A'.B' and column A'+B' are same.

## Proof of theorem 2:

| A | B | A+B | $\overline{A+B}$ | $\overline{A}$ | $\overline{B}$ | $\overline{A}.\overline{B}$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

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

From the above table, we can observe that the column A'+B' and column A'.B' are same.

## Applications of DeMorgan's Theorem

- The De Morgan theorems change +(plus) sign to •(dot) sign and vice-versa i.e., AND operation to OR operation and vice-versa
- 2. These theorems are used to convert Boolean expression of SOP form to POS form and vice-versa.

**Example 1.1** Apply DeMorgans theorem to the following expressions

(a) $\overline{(A+B+C)D}$ (b) $\overline{ABC+DEF}$

☐ **Solution**

(a) Let $A+B+C=X$ and $D=Y$.

The expression $\overline{(A+B+C)D}$ is in the form of theorem 1.

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

Next applying DeMorgans theorem 2 to the term $\overline{A+B+C}$

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

(b) Let $ABC=X$ and $DEF=Y$

The given expression is in the form of theorem 2 and it can be rewritten as

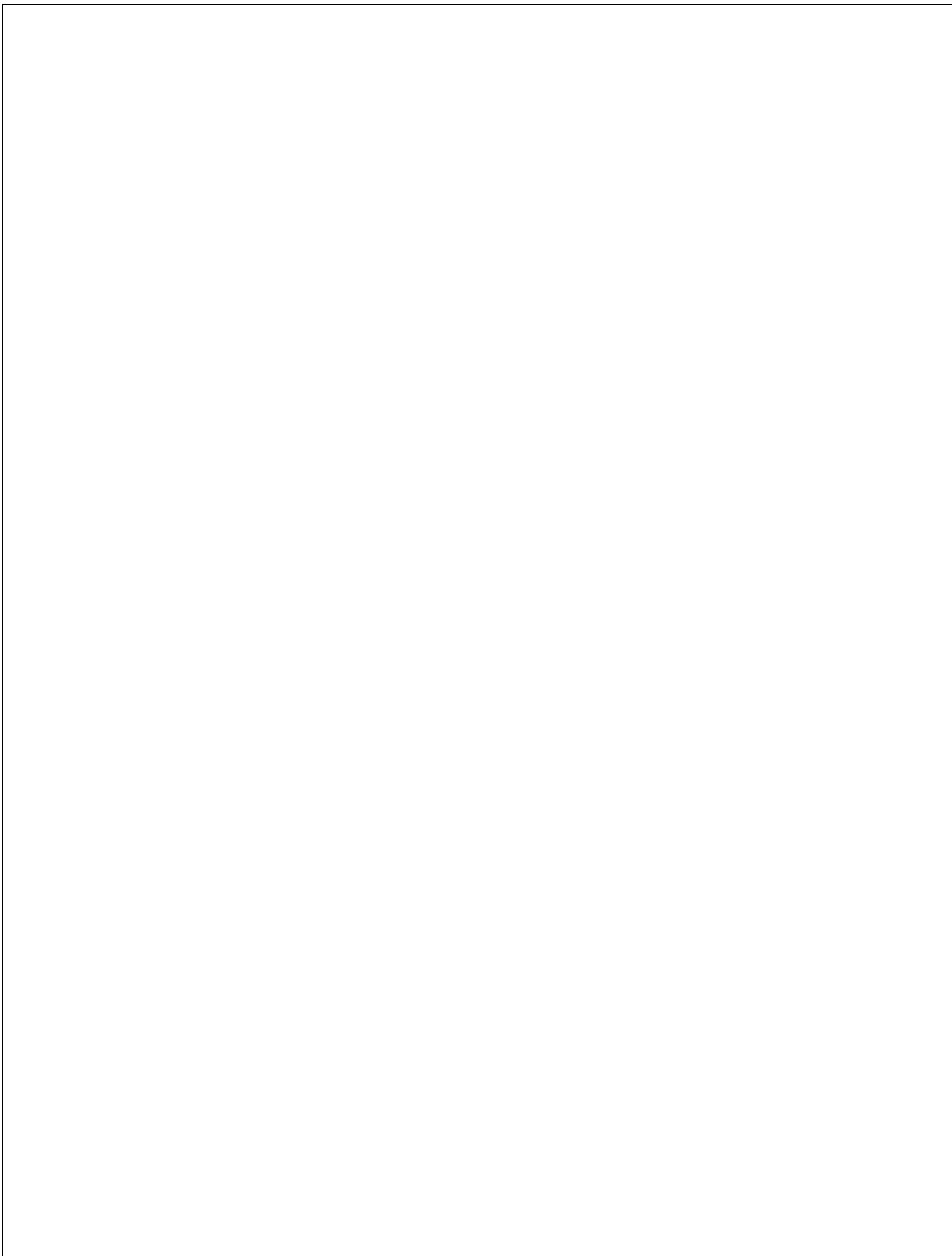
**Example 1.2** Simplify the expressions

(a) $\overline{\overline{A+B}+\overline{C}}$ (b) $\overline{\overline{A+B+\overline{C}}+D\left(\overline{E+F}\right)}$

☐ **Solution**

(a)

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



## CANONICAL FORM

- The **Quine–McCluskey algorithm** (or **the method of prime implicants**) is a method used for minimization of boolean functions which was developed by W.V. Quine and Edward J. McCluskey.
- It is functionally identical to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a Boolean function has been reached. It is sometimes referred to as the tabulation method.

## **The method involves two steps**:

- Finding all prime implicants of the function.
- Use those prime implicants in a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function.

## Step 1: finding prime implicants:

Minimizing an arbitrary function:

$$f(A,B,C,D) = \sum m(4,8,10,11,12,15) + d(9,14).$$

| | A B C D | f |
|---|---|---|
| m0 | 0 0 0 0 | 0 |
| m1 | 0 0 0 1 | 0 |
| m2 | 0 0 1 0 | 0 |
| m3 | 0 0 1 1 | 0 |
| m4 | 0 1 0 0 | 1 |
| m5 | 0 1 0 1 | 0 |
| m6 | 0 1 1 0 | 0 |
| m7 | 0 1 1 1 | 0 |
| m8 | 1 0 0 0 | 1 |
| m9 | 1 0 0 1 | x |
| m10 | 1 0 1 0 | 1 |
| m11 | 1 0 1 1 | 1 |
| m12 | 1 1 0 0 | 1 |
| m13 | 1 1 0 1 | 0 |
| m14 | 1 1 1 0 | x |
| m15 | 1 1 1 1 | 1 |

- One can easily form the canonical sum of products expression from this table, simply by summing the minterms (leaving out don't-care terms) where the function evaluates to one:
- $f_{A,B,C,D}$ = A'BC'D' + AB'C'D' + AB'CD' + AB'CD + ABC'D' + ABCD.
- Of course, that's certainly not minimal. So to optimize, all minterms that evaluate to one are first placed in a minterm table. Don't-care terms are also added into this table, so they can be combined with minterms:

| Number of 1s | Minterm | Binary Representation |
|---|---|---|
| 1 | m4 | 0100 |
| | m8 | 1000 |
| 2 | m9 | 1001 |
| | m10 | 1010 |
| | m12 | 1100 |
| 3 | m11 | 1011 |
| | m14 | 1110 |
| 4 | m15 | 1111 |

- At this point, one can start combining minterms with other minterms. If two terms vary by only a single digit changing, that digit can be replaced with a dash indicating that the digit doesn't matter. Terms that can't be combined any more are marked with a "*". When going from Size 2 to Size 4, treat '-' as a third bit value. Ex: -110 and -100 or -11- can be combined, but not -110 and 011-. (Trick: Match up the '-' first.)

| Number of 1s | Minterm | 0-Cube Size | 2 Implicants | Size 4 Implicants |
|---|---|---|---|---|
| 1 | m4 | 0100 | m(4,12) -100* | m(8,9,10,11) 10--* |
| | m8 | 1000 | m(8,9) 100- | m(8,10,12,14) 1--0* |
| -- | -- | -- | m(8,10) 10-0 | -- |
| 2 | m9 | 1001 | m(8,12) 1-00 | m(10,11,14,15) 1-1-* |
| | m10 | 1010 | -- | -- |
| | m12 | 1100 | m(9,11) 10-1 | -- |
| -- | -- | -- | m(10,11) 101- | -- |
| 3 | m11 | 1011 | m(10,14) 1-10 | -- |
| | m14 | 1110 | m(12,14) 11-0 | -- |
| 4 | m15 | 1111 | m(11,15) 1-11 | -- |
| | | | m(14,15) 111- | |

Note: In this example, none of the terms in the size 4 implicants table can be combined any further. Be aware that this processing should be continued otherwise (size 8 etc).

## Step 2: prime implicant chart:

- None of the terms can be combined any further than this, so at this point we construct an essential prime implicant table. Along the side goes the prime implicants that have just been generated, and along the top go the minterms specified earlier. The don't care terms are not placed on top - they are omitted from this section because they are not necessary inputs.
- Here, each of the essential prime implicants has been starred - the second prime implicant can be 'covered' by the third and fourth, and the third prime implicant can be 'covered' by the second and first, and neither is thus essential.
- If a prime implicant is essential then, as would be expected, it is necessary to include it in the minimized boolean equation. In some cases, the essential prime implicants do not cover all minterms, in which case additional procedures for chart reduction can be employed.
- The simplest "additional procedure" is trial and error, but a more systematic way is Petrick's Method. In the current example, the essential prime implicants do not handle all of the minterms, so, in this case, one can combine the essential implicants with one of the two non-essential ones to yield one of these two equations

- 
- 

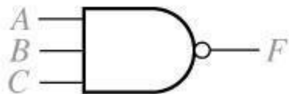
  Both of those final equations are functionally equivalent to the original, verbose equation:

  $f_{A,B,C,D} = A'BC'D' + AB'C'D' + AB'C'D + AB'CD' + AB'CD + ABC'D' + $

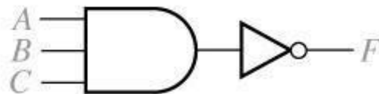
solve another example : f(a,b,c,d)=sum(0,1,2,5,8,14)+sum d(4,10,13) by the quine McClusky method.

## Realization of logic functions - NAND gate realization - NOR gate realization

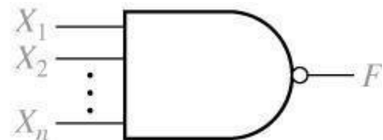
### NAND gate



(a) 3-input NAND gate



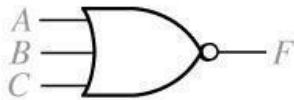
(b) NAND gate equivalent



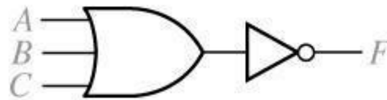
(c) *n*-input NAND gate

– F(ABC)=(ABC)’=A’+B’+C’

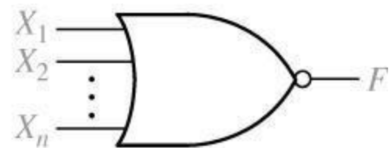
### NOR gate



(a) 3-input NOR gate



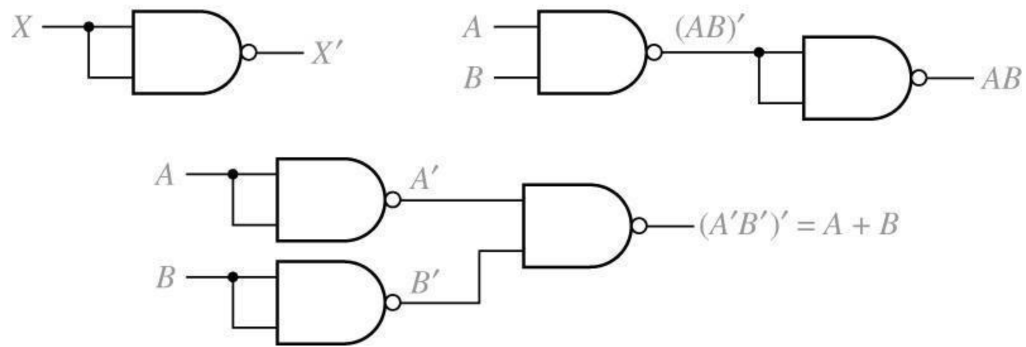
(b) NOR gate equivalent



(c) *n*-input NOR gate

F(ABC)=(A+B+C)’=A’B’C’

- Any logic function can be implemented using only NAND or NOR gates.
- If we can use NAND or NOR gates to implement AND, OR, and inverter, then we can prove that any logic function can be expressed using only NAND or NO gates.
- A set of logic operations is said to be functionally complete if any Boolean function can be expressed in terms of this set of operations.
- The set AND, OR, and NOT is obviously functionally complete.




X’=X nand X
AB=(A nand B) nand (A nand B)
A+B=(A nand A) nand (B nand B)

- Actually, as long as we could show NAND can express OR and NOT (AND and NOT), we can show NAND is functionally complete.
  - XY=(X’+Y’)’
  - X+Y=(X’Y’)’
- Can you prove that NOR is functionally complete?

## Design of Two-Level Circuits Using NAND and NOR Gates

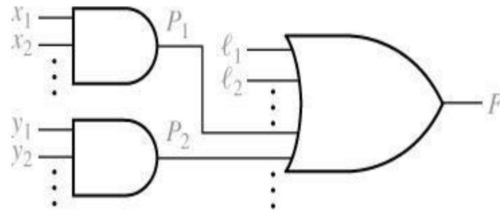
- The conversion from circuits composed of AND and OR gates to circuits composed of NAND or NOR gates is carried out by using F=(F’)’ and then applying
- DeMorgan’s laws:
  - (X1+X2+…+Xn)’=X1’X2’…Xn’
  - (X1X2…Xn)’= X1’+X2’+…+Xn’

- F=A+BC’+B’CD
- =[(A+BC’+B’CD)’]’ (F’)’
- =[A’(BC’)’(B’CD)’]’ NAND-NAND
- =[A’(B’+C)(B+C’+D’)]’ OR-NAND
- =A+(B’+C)’+(B+C’+D’)’ NOR-OR

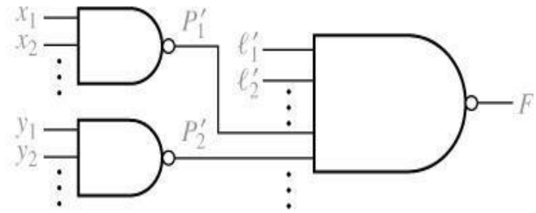
We started with the minimum sum-of-products expression.

### Procedure for designing a minimum two-level NAND-NAND circuit

- Find a minimum sum-of-products expression for F: Karnaugh maps, Quinne-McCluskey and Petrick methods
- Draw the corresponding two-level AND-OR circuit
- Replace all gates with NAND gates leaving the gate
- interconnections unchanged.
- If the output gate has any single literals as inputs, complement these literals.



(a) Before transformation



(b) After transformation

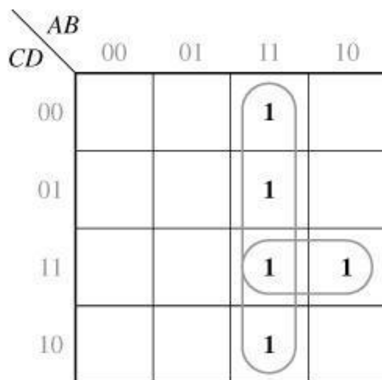
F=(A+B+C)(A+B'+C')(A+C'+D)

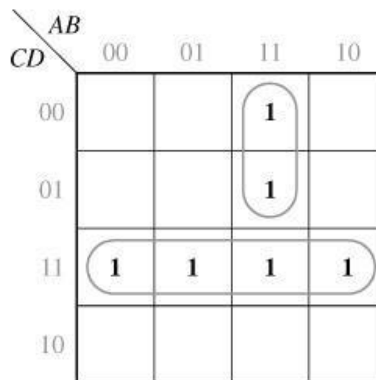
- ={[(A+B+C)(A+B'+C')(A+C'+D)]'}' (F')'
- =[(A+B+C)'+(A+B'+C')'+(A+C'+D)']' NOR-NOR
- =(A'B'C'+A'BC+A'CD)' AND-NOR
- =(A'B'C')'(A'BC)'(A'CD)' NAND-AND
- We started with the minimum product-of-sums expression.

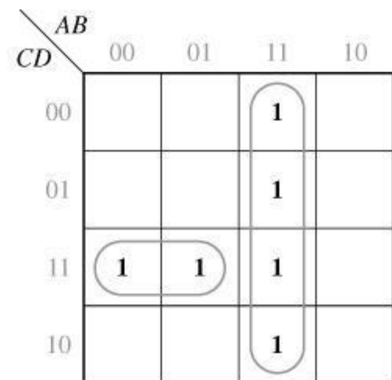
Procedure for designing a minimum twolevel NOR-NOR circuit

- Find a minimum product-of-sums expression for F
- Draw the corresponding two-level OR-AND for F
- Replace all gates with NOR gates leaving the gate interconnections unchanged.
- If the output gate has any single literals as inputs, complement these literals.
- Given a logic function, we can simplify it using some methods.
- But if we need to design a circuit to implement several functions, it may not be enough to simplify each function separately.

- Example:
  - F1(A,B,C,D)=sum[m(11,12,13,14,15)]
  - F2(A,B,C,D)=sum[m(3,7,11,12,13,15)]
  - F3(A,B,C,D)=sum[m(3,7,12,13,14,15)]

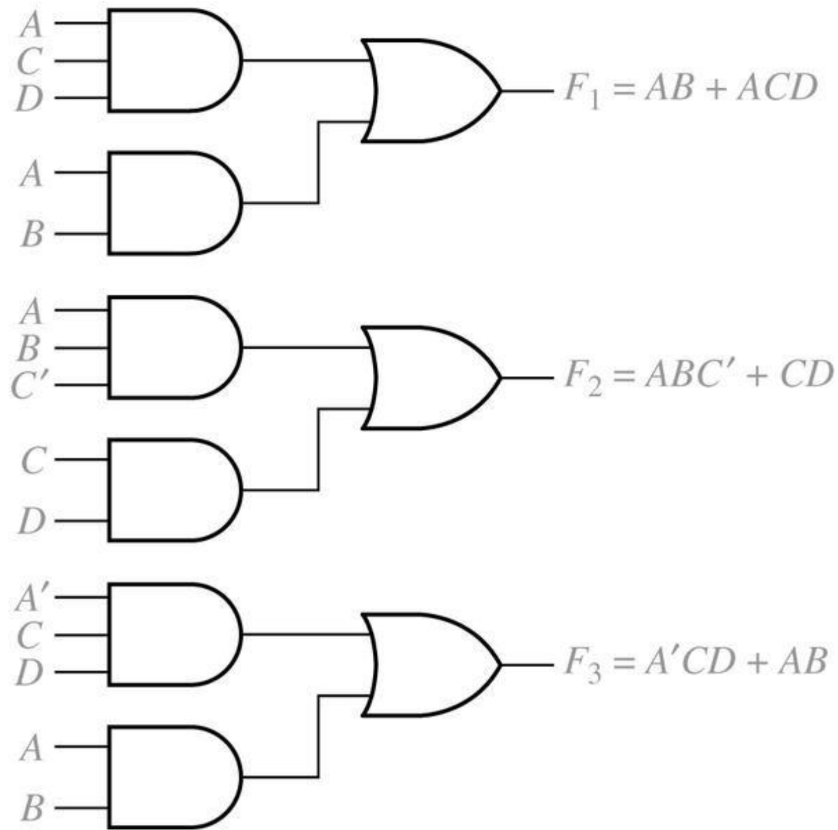


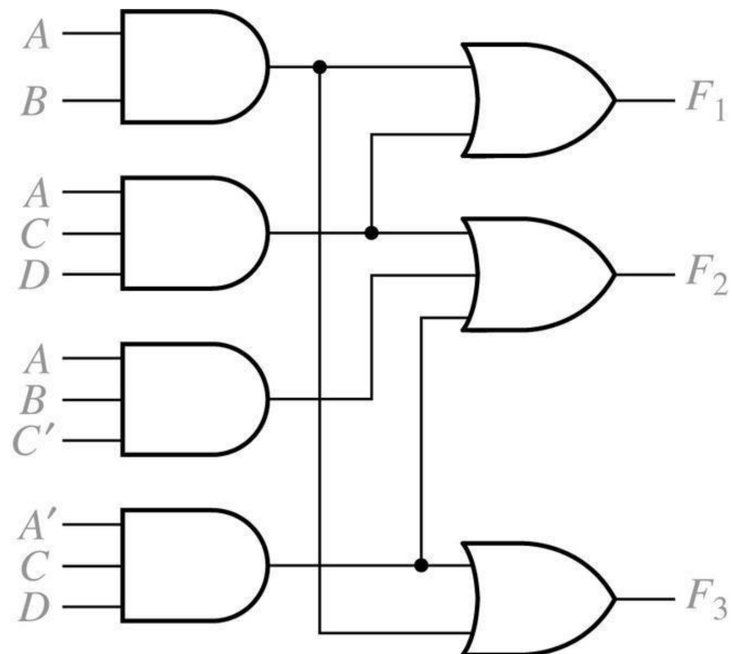




F1=AB+ACD F2=ABC'+CD F3=A'CD+AB

**Design of Two-Level, Multiple-Output Circuits**




Design of Two-Level, Multiple- Output Circuits



## QUINE-MCCLUSKY MINIMIZATION PROCEDURE:

Step 1: List the minterms grouped according to the number of 1's in their binary representation in the decimal format.

Step 2: Compare each minterm with larger minterms in the next group down. If they differ by a power of 2 then they pair-off. Check both minterms and form a second table by the minterms paired and substitute the decimal difference of the corresponding minterms in the bracket, i.e. mx, my (y-x).

Step 3: Compare each element of the group in the new table with elements of the next lower group and select numbers that have the same numbers in parenthesis. If the lowest minterm number of the table formed in the lower group is greater than the corresponding number by a power of 2 then they combine; on the right of both elements.

Step 4: Form a second table by all four minterms followed by both powers of 2 in parentheses, i.e. the previous value (the difference) and the power of 2 that is greater.

Step 5: Select the common literals from each prime implicant by comparison.

Step 6: Write the minimal SOP from the prime implicant that are not checked .

Example 1.

- Minimize the function f given below by Quine-McClusky method using decimal notation. f (A,B,C,D) ABCD ABCD ABCD ABCD ABCD ABCD ABCD ABCD ABCD.

Solution Step 1: Organize minterm as follows: f (A,B,C,D) $\sum$ m(0,5,6,7,9,10,13,14,15)

Arrange minterms to correspond to their number of 1's as shown in E1.1a

| | 1's | Minterms | | |
|---|---|---|---|---|
| * | 0 | 0 | | ...(a) |
| | 2 | 5 | ✓ | |
| | | 6 | ✓ | |
| | | 9 | ✓ | |
| | | 10 | ✓ | |
| | 3 | 7 | ✓ | |
| | | 13 | ✓ | |
| | | 14 | ✓ | |
| | 4 | 15 | ✓ | |

Table E1.1a

| | minterm | paired | | |
|---|---|---|---|---|
| | $5^{\Phi}, 7^{\Phi}$ | $(2)^{\dagger}$ | ✓ | |
| | 5, 13 | (8) | ✓ | |
| | 6, 7 | (1) | ✓ | |
| | 6, 14 | (8) | ✓ | |
| * | 9, 13 | (4) | | ...(b) |
| * | 10, 14 | (4) | | ...(c) |
| | 7, 15 | (8) | ✓ | |
| | 13, 15 | (2) | ✓ | |
| | 14, 15 | (1) | ✓ | |

Table E1.1b

| | minterms paired | | |
|---|---|---|---|
| * | 5,7-13,15' | (2,8) | ...(d) |
| * | 6,14-7,15 | (1,8) | ...(e) |

Table E1.1c

$^{\Phi}$ - squares combined (2 squares);

Consider the function: $Z = f(A,B,C) = \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}C + A\bar{B}\,\bar{C} + A\bar{B}C$

To make things easier, change the function into binary notation with index value and decimal value.

$f(A, B, C) = \sum (000, 001, 100, 101)$ —— Binary notation

0 1 1 2 —— Index

0 1 4 5 —— Decimal value

Tabulate the index groups in a colunm and insert the decimal value alongside.

| | First List | | Second List | | Third List | |
|---|---|---|---|---|---|---|
| | | A B C | | A B C | | A B C |
| Index 0 → | 0 | 0 0 0 ✓ | 0,1 | 0 0 – ✓ | 0,1,4,5 | – 0 – |
| Index 1 → | 1 | 0 0 1 ✓ | 0,4 | – 0 0 ✓ | 0,4,1,5 | – 0 – |
| | 4 | 1 0 0 ✓ | 1,5 | – 0 1 ✓ | | |
| Index 2 → | 5 | 1 0 0 ✓ | 4,5 | 1 0 – ✓ | | |

- From the first list, we combine terms that differ by 1 digit only from one index group to the next. These terms from the first list are then seperated into groups in the second list. Note that the ticks are just there to show that one term has been combined with another term. From the second list we can see that the expression is now reduced to:

  $Z = \quad + \quad + \bar{B}C + A\bar{B}$

- From the second list note that the term having an index of 0 can be combined with the terms of index 1. Bear in mind that the dash indicates a missing variable and must line up in order to get a third list. The final simplified expression is: $Z = \bar{B}$
- Bear in mind that any unticked terms in any list must be included in the final expression (none occured here except from the last list).
- Note that the only prime implicant here is $Z = \bar{B}$.
- The tabular method reduces the function to a set of prime implicants.

Note that the above solution can be derived algebraically. Attempt this in your notes.

**Example 2:**

Consider the function $f(A, B, C, D) = \sum (0,1,2,3,5,7,8,10,12,13,15)$, note that this is in decimal form.

**First List**

| | A B C D | |
|---|---|---|
| 0 | 0 0 0 0 | ✓ |
| 1 | 0 0 0 1 | ✓ |
| 2 | 0 0 1 0 | ✓ |
| 8 | 1 0 0 0 | ✓ |
| 3 | 0 0 1 1 | ✓ |
| 5 | 0 1 0 1 | ✓ |
| 10 | 1 0 1 0 | ✓ |
| 12 | 1 1 0 0 | ✓ |
| 7 | 0 1 1 1 | ✓ |
| 13 | 1 1 1 1 | ✓ |
| 15 | 1 1 1 1 | ✓ |

**Second List**

| | A B C D | |
|---|---|---|
| 0,1 | 0 0 0 – | ✓ |
| 0,2 | 0 0 – 0 | ✓ |
| 0,8 | – 0 0 0 | ✓ |
| 1,3 | 0 0 – 1 | ✓ |
| 1,5 | 0 – 0 1 | ✓ |
| 2,3 | 0 0 1 – | ✓ |
| 2,10 | – 0 1 0 | ✓ |
| 8,10 | 1 0 – 0 | ✓ |
| 8,12 | 1 – 0 0 | $A\bar{C}\bar{D}$ |
| 3,7 | 0 – 1 1 | ✓ |
| 5,7 | 0 1 – 1 | ✓ |
| 5,13 | – 1 0 1 | ✓ |
| 12,13 | 1 1 0 – | $AB\bar{C}$ |
| 7,15 | – 1 1 1 | ✓ |
| 13,15 | 1 1 – 1 | ✓ |

**Third List**

| | A B C D | |
|---|---|---|
| 0, 1 2 3 | 0 0 – – | $\bar{A}\bar{B}$ |
| 0, 2 1 3 | 0 0 – – | |
| 0, 2, 8, 10 | – 0 – 0 | $\bar{B}\bar{D}$ |
| 0, 8, 2, 10 | – 0 – 0 | |
| 1, 3, 5, 7 | 0 – – 1 | $\bar{A}D$ |
| 1, 5, 3, 7 | 0 – – 1 | |
| 5, 7, 13, 15 | – 1 – 1 | $BD$ |
| 5, 13, 7, 15 | – 1 – 1 | |

The prime implicants are: $+$ $+ \bar{A}D + BD + A$ $+ AB\bar{C}$

- The chart is used to remove redundant prime implicants. A grid is prepared having all the prime implicants listed at the left and all the minterms of the function along the top. Each minterm covered by a given prime implicant is marked in the appropriate position.

| | 0 | 1 | 2 | 3 | 5 | 7 | 8 | 10 | 12 | 13 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\bar{A}\bar{B}$ | X | X | X | X | | | | | | | |
| $\bar{B}\bar{D}$ | X | | X | | | | X | (X) | | | |
| $\bar{A}D$ | | X | | X | X | X | | | | | |
| $BD$ | | | | | X | X | | | | X | (X) |
| $A\bar{C}\bar{D}$ | | | | | | | X | | X | | |
| $AB\bar{C}$ | | | | | | | | | X | X | |
| Essential | X | | X | | X | X | X | (X) | | X | (X) |

- From the above chart, BD is an essential prime implicant. It is the only prime implicant that covers the minterm decimal 15 and it also includes 5, 7 and 13. $\bar{B}\ \bar{D}$ is also an essential prime implicant.

- It is the only prime implicant that covers the minterm denoted by decimal 10 and it also includes the terms 0, 2 and 8. The other minterms of the function are 1, 3 and 12. Minterm 1 is present in $\overline{A}\,\overline{B}$ and $\overline{A}D$. Similarly for minterm 3. We can therefore use either of these prime implicants for these minterms. Minterm 12 is present in $A\overline{C}\,\overline{D}$ and $AB\overline{C}$, so again either can be used.

  Thus, one minimal solution is: $Z = \overline{B}\,\overline{D} + BD + \overline{A}\,\overline{B} + A\overline{C}\,\overline{D}$

**UNIT - III**

**Combinational Logic Design:** Half adder - Full adder- Full- subtractor – Parallel Adder- Carry Look Ahead Adder – BCD Adder – Magnitude Comparator – Encoders and Decoders – Multiplexers – Code converters – Parity generator, Parity checker- Combinational circuit implementation using multiplexers and decoders.

**Programmable Logic Devices:** PROM – EPROM – EEPROM- Programmable Logic Array (PLA) – Programmable Array Logic (PAL) -Realization of combinational circuits using PLDs.

## Adder

## Definition:

Half adder**:**

- The circuit that will add two bits and produce a sum and a carry bit.

Full adder**:**

- The circuit that will add three bits and produce a sum and a carry bit.
- Adder is a combinational logic circuit, which performs the addition of two binary bits. For, any adder produces two outputs; sum and carry. Adder circuits can be classified into two types depending on the number of bits in the input variable.
  1. Half adder
  2. Full adder

## Half Adder

- A logic circuit which performs the addition of two bits is called a half adder. The half adder needs two inputs augend and addend bits; and two binary output: sum and carry.
- Fig.1.13 shows the block diagram of half adder and Table 1.12 shows the truth table of half adder. The next step is determining the expression for output variables (sum and
- carry) by using K-map. The two variables map is used to determine the expression because the half adder is having two input variables.

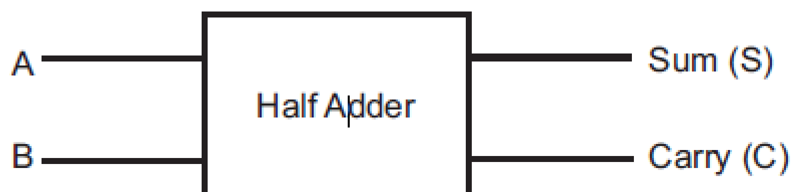



**Fig 1.13 Block diagram of Half adder**

**Table 1.12 Truth table of Half adder**

| Inputs | | Outputs | |
|---|---|---|---|
| A | B | S | C |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Expression for Sum (S)

| A \ B | 0 | 1 |
|---|---|---|
| 0 | (0) | (1) 1 |
| 1 | (2) 1 | (3) |

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

$S = A \oplus B$

Expression for Carry (C)

| A \ B | 0 | 1 |
|---|---|---|
| 0 | (0) | (1) |
| 1 | (2) | (3) 1 |

$C = A.B$

## Logic diagram

Obtain the logic circuit shown in Fig.1.14 by using the above sum and carry expression

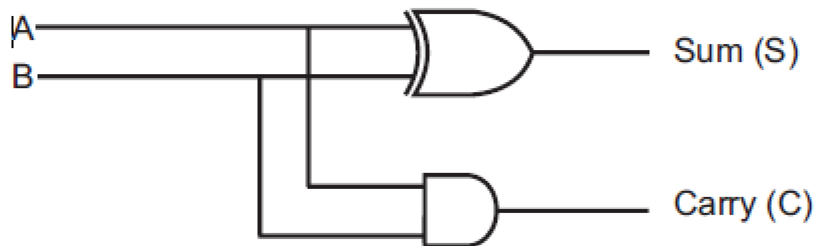



**Fig 1.14 Logic diagram of Half adder**

## Full adder

- A half adder has only two inputs and there is no provision to add a carry coming from the lower order bits when multi addition is performed. For this purpose, a full adder is designed.
- A full adder is a combinational logic circuit that performs the arithmetical sum of three inputs bits. It consists of three inputs and two outputs.
- Two of the input variables denoted by A and B represent the two significant bits to be added. The third input Cin represents the carry from the previous lower significant position.

- Fig. 1.15 shows the block diagram of a full adder and Table 1.13 shows the truth table for a full adder.

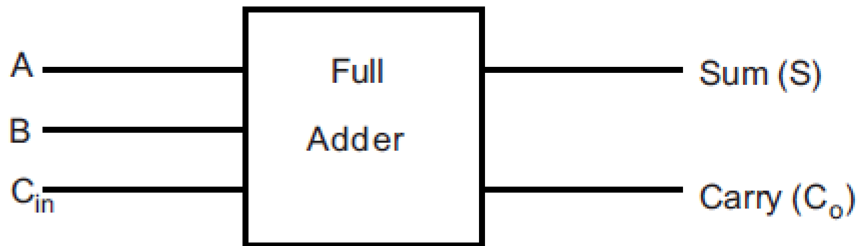



**Fig 1.15 Block diagram of a full adder**

**Table 1.13 Truth table of full adder**

| Inputs | | | Outputs | |
|---|---|---|---|---|
| A | B | $C_{in}$ | S | $C_o$ |
| 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 |

## K-map simplification

**Expression for Sum (S)**

| A \ $BC_{in}$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1: 1 | 3 | 2: 1 |
| 1 | 4: 1 | 5 | 7: 1 | 6 |

$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$

**Expression for Carry ($C_o$)**

| A \ $BC_{in}$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 3: 1 | 2 |
| 1 | 4 | 5: 1 | 7: 1 | 6: 1 |

$C_o = AB + AC_{in} + BC_{in}$

## Logic diagram

- The logic diagram is constructed by using logic gates for the sum and carry of a full adder circuit. Fig.1.16 shows the logic diagram of a full adder.

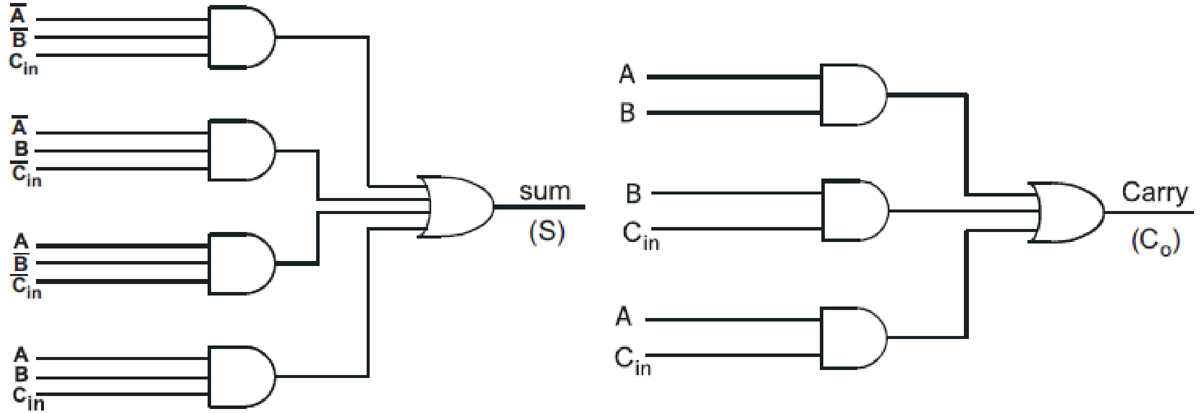



**Fig 1.16 Logic diagram of Full adder**

The Boolean function for sum can be further simplified as follows.

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

$$= C_{in}(\overline{A}\,\overline{B} + AB) + \overline{C}_{in}(\overline{A}\,B + A\overline{B})$$

Let $\overline{A}B + A\overline{B} = x$

then $\overline{A}\,\overline{B} + AB = \overline{x}$

$$S = C_{in}\,(\overline{x}) + C_{in}x$$

$$= C_{in} \oplus x$$

Replacing $x$ by $A \oplus B$ $\quad S = C_{in} \oplus A \oplus B$

- From this simplified expression, we obtain the simplified full adder logic circuit as shown in Fig.1.17

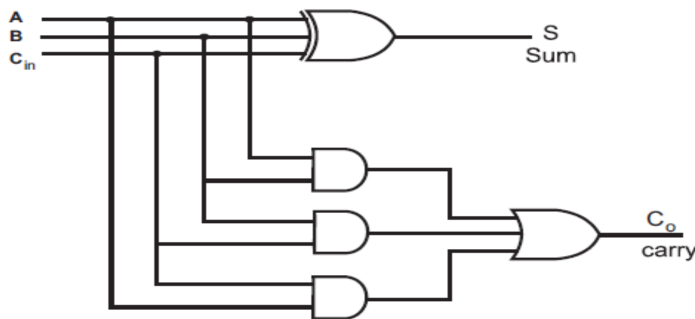



**Fig 1.17 Logic circuit of Simplified Full Adder**

- The full adder can also implemented with two half adders shown in Fig.1.18. The sum output from the second half adder is the exclusive OR of Cin and the output of first half adder

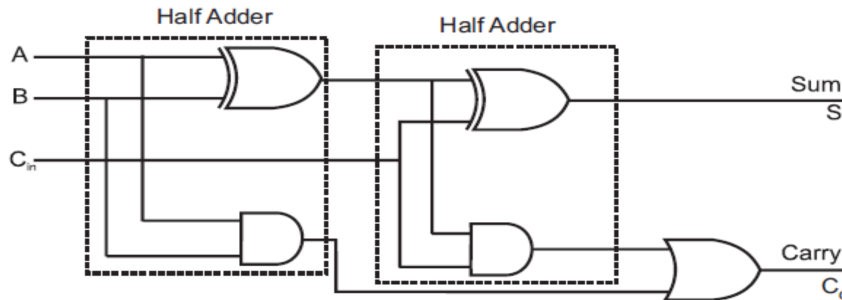



Fig 1.18 Implementation of Full adder Using two Half adders

# SUBTRACTOR:

## Definition:

### Half Subtractor:

- The circuit that will subtract two bits and produce a barrow and difference.

### Full subtractor:

- The circuit that will subtract three bits and produce a barrow and difference.

- Subtractor is a combinational logic circuit. It perform the subtraction operation. For, any subtractor will produce a difference and borrow. Subtractor is classified based on the number of bits performed.
  1. Half subtractor 2. Full subtractor.

## Half Subtractor:

- A half subtractor is a combinational logic circuit that subtracts two bits and produce their difference and barrow.
- The half subtractor needs two inputs: minuend and subtrahend bits and two output: barrow and difference. Fig.1.19 shows the block diagram of half subtractor and Table 1.14 shows the truth table of half subtractor.

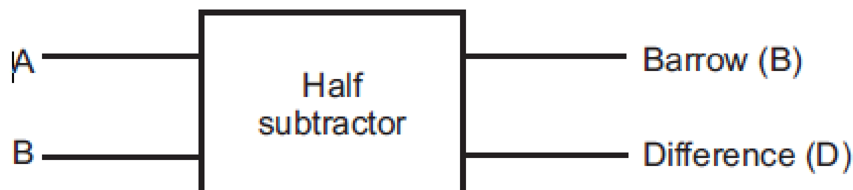
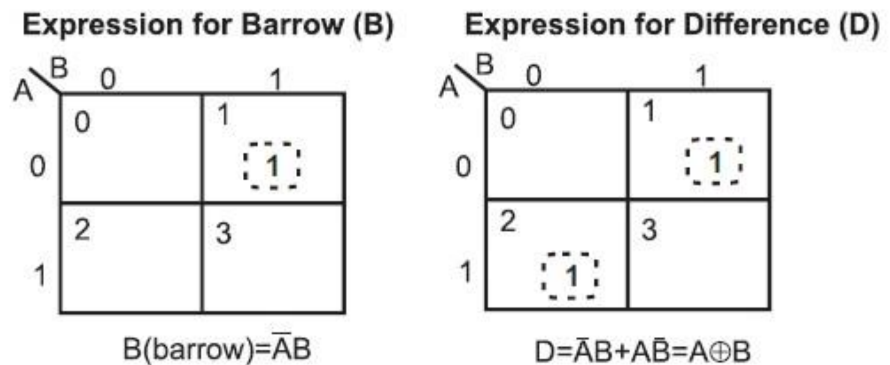



Fig 1.19 Block diagram of Half subtractor

**Table 1.14 Truth table of Half subtractor**

| Inputs | | Outputs | |
|---|---|---|---|
| A | B | Difference(D) | Barrow(B) |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

## K-map simplification




### Logic Diagram:

- The logic diagram is constructed for the above expression by using logic gate and it is shown in Fig.1.20

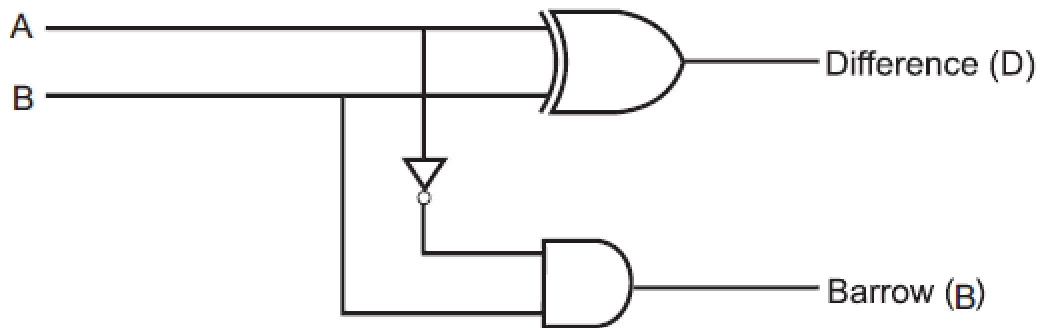



**Fig 1.20 Logic diagram of subtractor**

### Full Subtractor

- A full subtractor is a combinational logic circuit that performs subtraction involving three bits, namely minuend bit, subtrahend bit and the barrow from the previous stage.
- Fig.1.21 shows the block diagram of a full subtractor and table 1.15 shows the truth table for a full subtractor.

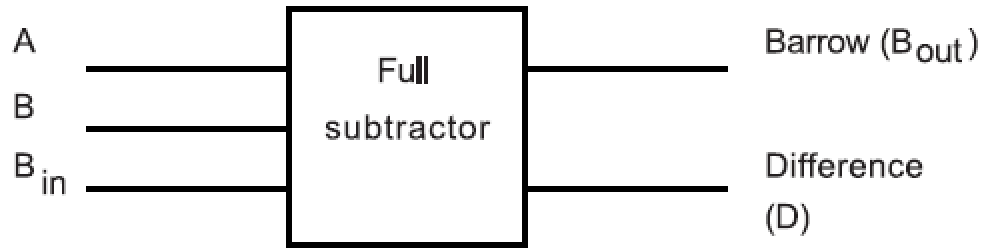



**Fig 1.21 Block diagram of full subtractor**

## Logic Diagram

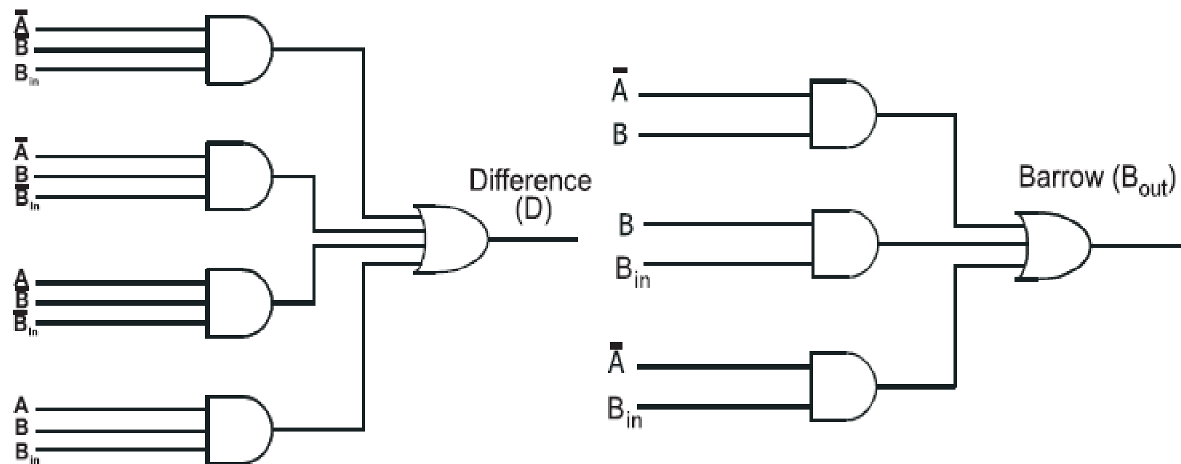



**Fig 1.22 Logic diagram of a full subtractor**

- Logic diagram is constructed for the above two expressions by using basic gates, as shown in Fig.1.22.
- The Boolean function for D (difference) can be further simplified of a full subtractor.

subtractor

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

$$= B_{in}\left(\overline{A}\,\overline{B} + AB\right) + B_{in}\left(\overline{A}B + A\overline{B}\right)$$

Let $\overline{A}B + A\overline{B} = X = A \oplus B$

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

$$D = B_{in}\left(\overline{X}\right) + \overline{B}_{in}(X)$$

Replacing $X$ by $A \oplus B$

$$D = B_{in} \oplus A \oplus B$$

- With this simplified Boolean function circuit (Fig.1.24), a full subtractor can be implemented. The full subtractor can also be implemented with two half subtractors and one OR gate, as shown in Fig.1.23.
- The difference in output from the second half subtractor is the exclusive OR of Bin and the output of the first half-subtractor, is same as the difference in the output of a full subtractor.

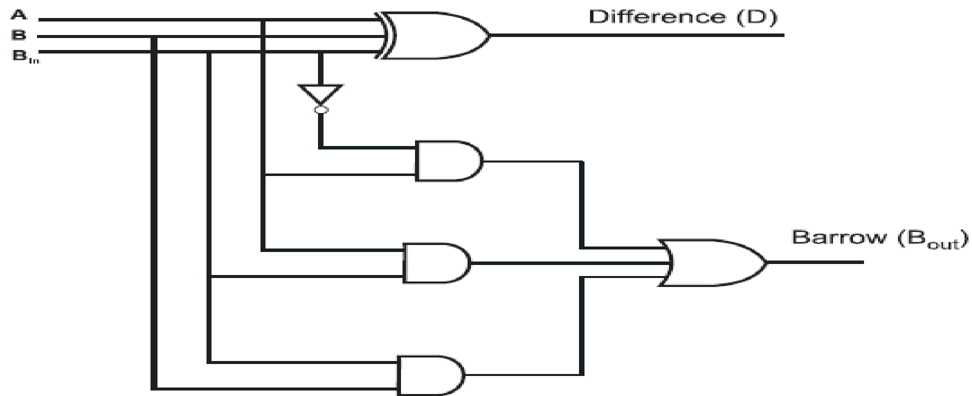



Fig 1.23 Simplified logic diagram for full subtractor

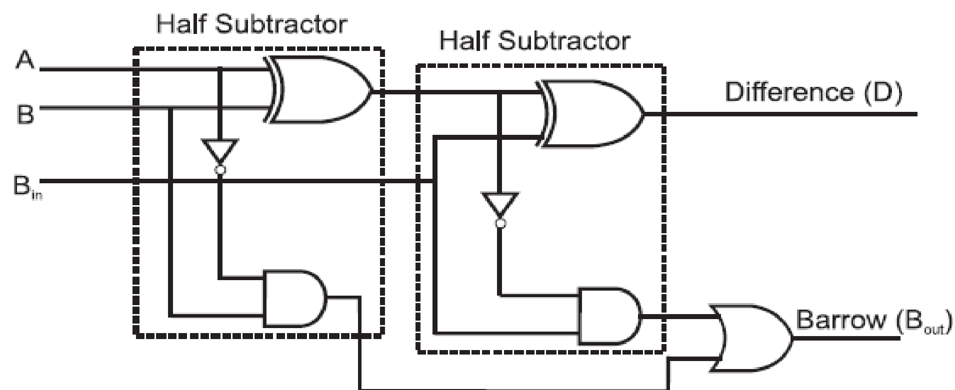



Fig 1.24 Implementation of full subtractor using two half subtractor.

## 4 bit Binary Parallel adder:

## Definition:

**Parallel binary adder:**

- A circuit, consisting of n full adders, that will add two n bit binary numbers. The output consists of n sum bits and a carry bit.

**Cascade:**

- To connect an output of one device to an input of another device, often for the purpose of expanding the number of bits available for a particular function.
- In most logic circuits, addition of more than one bit is carried out. The addition of multibit numbers can be accomplishedly using several full adders. The 4 - bit adder using full adder circuits, is capable of adding two 4 - bit numbers resulting in a 4 - bit sumand a carry output as shown in Fig.1.25.
- The addend and augend (4 bits) are fed into the 4 bit adder circuits simultaneously and the additions in each position take place at the same time. This circuit is known as parallel adder.

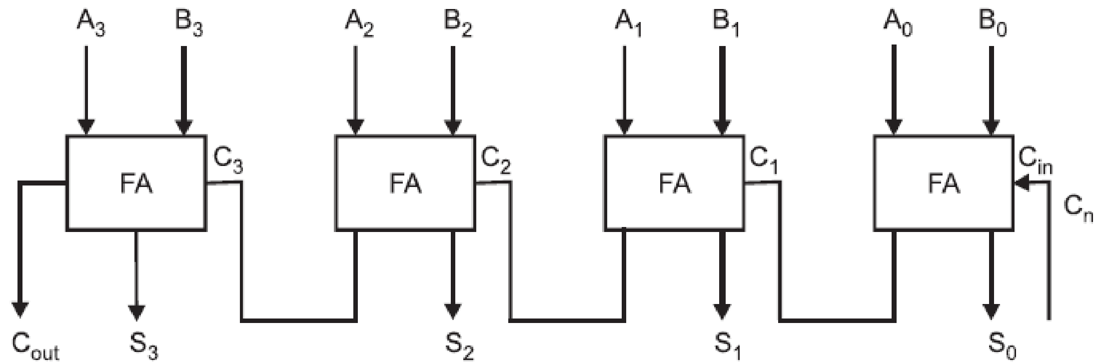



**Fig: 4 Bit parallel binary adder**

- Let the 4 bit words to are augend be represented by A3 A2 A1 A0 = 1111 and addend B3 B2 B1 B0 = 0011.

| | | |
|---|---|---|
| Input carry | | 1 1 1 0 |
| Augend | = | 1 1 1 1 |
| Addend | = | 0 0 1 1 |
| | 1 | 0 0 1 0 |
| | ↑ | |
| | carry out | 4 bit sum |

- In a 4 - bit parallel binary adder circuit, the input to each full adder will be Ai, Bi and Ci, and the output will be Si and Ci+1, where 'i' varies from 0 to 3.
- Also the carry output of the lower stage is connected to the carry input of next higher order stage. Hence this type of adder is called ripple-carry adder.
- The least significant stage, A0, B0 and C0 (C0 must 0) are added resulting in sum S0 and carry C1. This carry C1 becomes the carry input of second stage.
- Similarly, in the second stage, A1, B1 and C1 are added resulting in S2 and C2; in third stage, A2, B2 and C2 are added resulting in S2 and C3 in the fourth stage, A3, B3 and C3 are added resulting in S3 and C4 which is the output carry. Thus, the circuit results in a sum (S3 S2 S1 S0) and a carry output C4.
- In the parallel adder, each full adder carry input depends on the previous stage output. The propagation delay (tp) of a full-adder is the time difference between the instant at which the inputs (Ai, Bi & Ci) are applied and the outputs (Si and Ci+1) are generated.
  First stage of FA produced after 1tp
  Second stage of FA produced after 2tp
  Third stage of FA produced after 3tp
  Fourth stage of FA produced after 4tp
- If a full adder is having the propagation delay of 50 ns, the output in the fourth stage will be generated only after 4*tp* (4×50 = 200 ns)

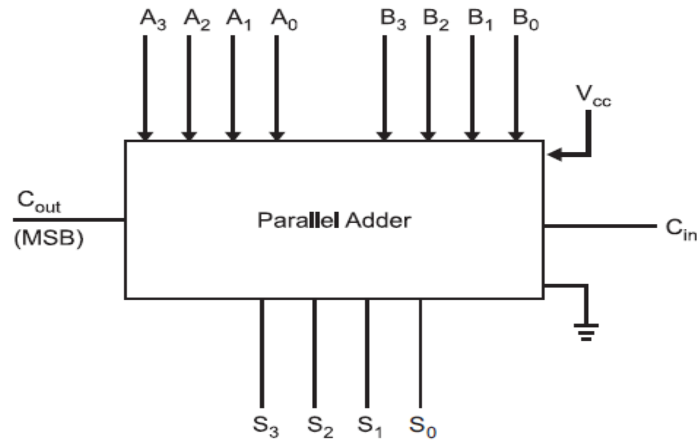



**Fig 1.26 Function symbol of 4-bit parallel adder**

- One of the methods of speeding up this process is to look ahead for carry addition which eliminates the ripple - carry delay.
- Fig.1.26 shows function symbol for the parallel adder. The inputs to this IC are two 4 bit numbers, *A*3*, A*2*, A*1*, A*0 and *B*3*, B*2*, B*1*, B*0 and carry *Cin* into LSB position. The outputs are the sum bits S3 S2 S1 S0 and the carry Cout,Cout of the MSB position.

## Ripple carry adder

- A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs (Ai, Bi; and Cin) and two outputs (Si and Cout) as illustrated in Fig.1.27 and the gate implementation of full adder is shown in Fig.1.27.

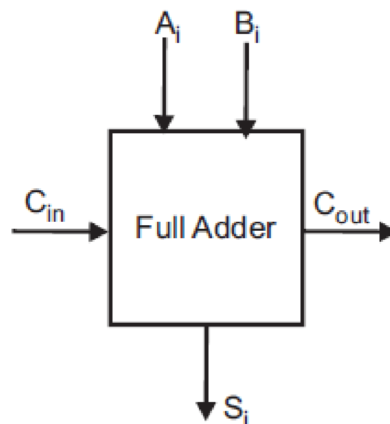



Fig 1.27 Block diagram of full adder

- A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It is also called as a Parallel adder.

- It can be constructed with full adders connected in cascade with the carry output from each full adder connected to the carry input of the next full adder in the chain. Fig.1.28 shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder.
- Notice from Fig.1.28 that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits A0 and B0 in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits S3− S0.

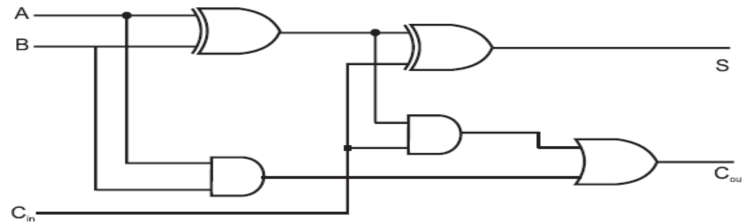



**Fig 1.28 Gate implementation of full adder**

- Thus, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage.
- This can be easily understood if one considers the addition of the two 4-bit words: $1111_2+0001_2$

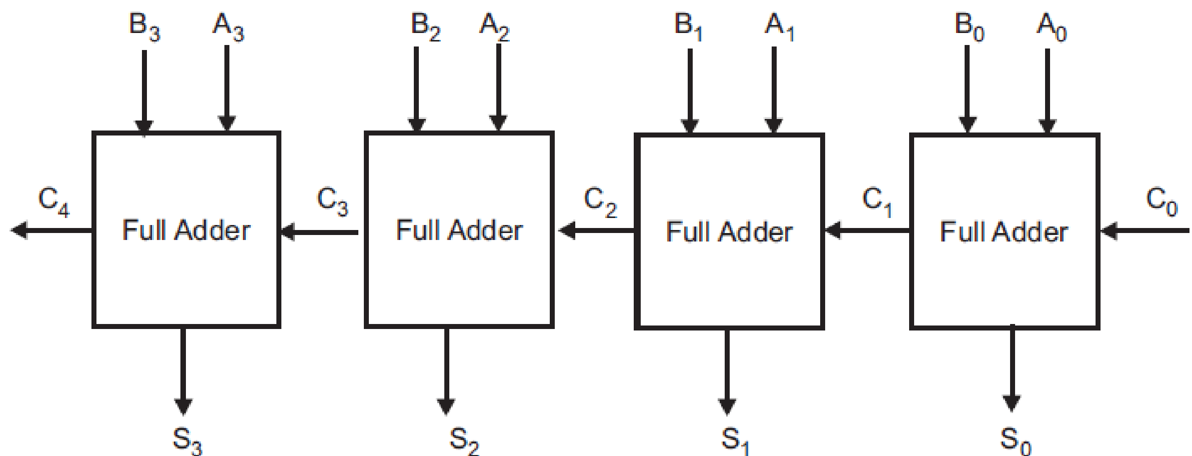



**Fig 1.29 Block diagram of parallel adder**

- In this case, the addition of (1+1 = 102) in the least significant stage causes a carry bit to be generated. This carry bit will consequently generate another carry bit in the next stage, and so on, until the final carry-out bit appears at the output.
- This requires the signal to travel (ripple) through all the stages of the adder as illustrated in Fig.1.29.

- As a result, the final Sum and Carry bits will be valid after a considerable delay. The carry-out bit of the first stage will be valid after 3 gate delays (1 associated with the XOR gate and 1 each associated with the AND and OR gates).
- From the schematic of Fig.1.29, one finds that the next carry-out (C2) will be valid after an additional 3 gate delays (associated with the AND and OR gates) for a total of 6 gate delays. In general the carry-out of a N-bit adder will be valid after 2N+2 gate delays. The sum bit will be valid after an additional 2 gate delays after the carry-in signal.
- Thus the sum of the most significant bit 2N-1 will be valid after 2(N-1) + 2 +2 = 2N +2 gate delays. This delay may be in addition to any delays associated with the interconnections.

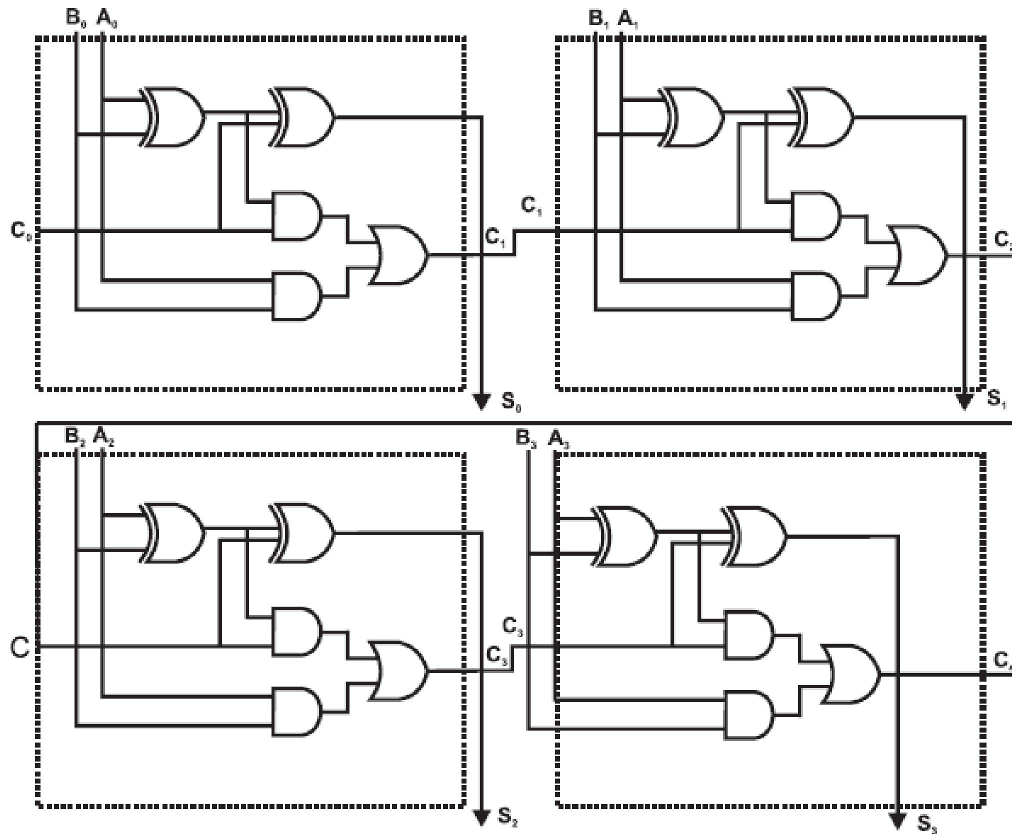



**Fig 1.30 Logic diagram for 4 bit Ripple carry adder**

- The disadvantage of the ripple-carry adder is that it can get very slow when we needs to add many bits. For fast applications, a better design is required.

## Carry look ahead carry adder:

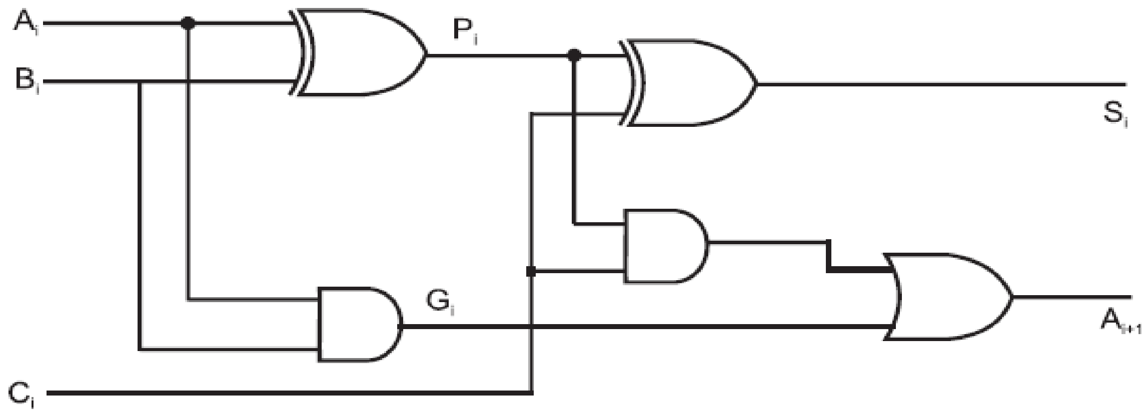



**Fig 1.31 Full adder at stage i with** Pi **and** Gi **shown**

- The carry-look-ahead adder solves this problem by calculating the carry signals in advance, based on the input signals.
- It is based on the fact that a carry signal will be generated in two cases:
  1. when both bits Ai and Bi are 1, or
  2. when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

  Thus, we can write, from Fig.1.31

$$Cout = C_{i+1} = A_i.B_i + (A_i \oplus B_i).C_i. \quad (1)$$

$$S_i = (A_i \oplus B_i) \oplus C_i. \quad (2)$$

The "$\oplus$" stands for exclusive OR or XOR. We can write this expression also as

$$C_{i+1} = G_i + P_i.C_i$$

in which

$$G_i = A_i.B_i$$

$$P_i = (A_i \oplus B_i)$$

- are called the carry generate and carry propagate term, respectively. Let us assume that the delay through an AND gate is one gate delay and through an XOR gate is two gate delays. Notice that the Propagate and Generate terms only depend on the input bits and thus will be valid after two and one gate delays, respectively.
- If we uses the above expression to calculate the carry signals, we need not wait for the carry to ripple through all the previous stages to find its proper value. Let's apply this to a 4-bit adder to make it clear.

$$C1 = G0+P0.C0 \quad (3)$$

$$C2 = G1+P1.C1 = G1+P1.G0+P1.P0.C0 \quad (4)$$

$$C3 = G2+P2.G1+P2.P1.G0+P2.P1.P0.C0 \quad (5)$$

$$C4 = G3+P3.G2+P3.P2.G1+P3\ P2.P1.G0+P3\ P2.P1.P0.C0 \quad (6)$$

The Sum signal can be calculated as follows,

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i. \quad (7)$$

- The carry-look ahead adder can be broken up in two modules: (1) the Partial Full Adder, PFA, which generates Si, Pi and Gi as defined by the above equations and the Carry Look-ahead Logic, which generates the carry-out bits according to equations 3 to 6.
- The 4-bit adder can then be built by using 4 FAs and the Carry Look-ahead logic block as shown in Fig.1.32

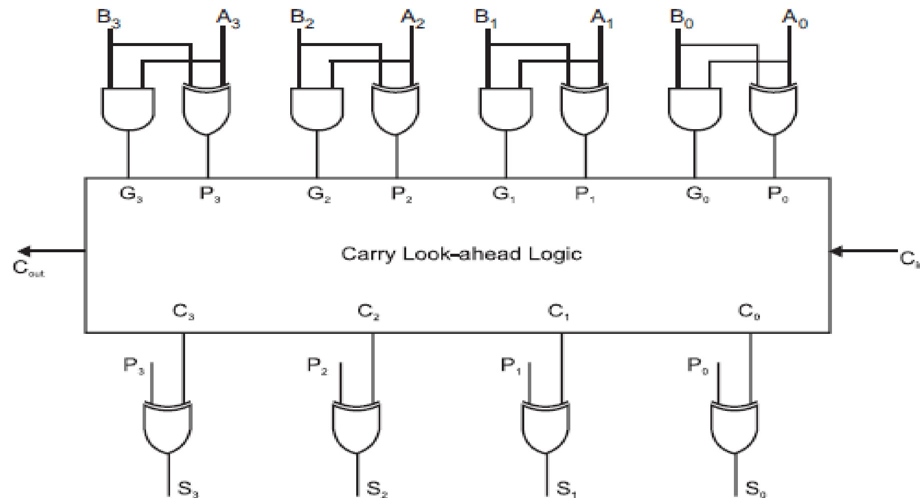



Fig 1.32 Logic diagram of Carry look ahead carry adder

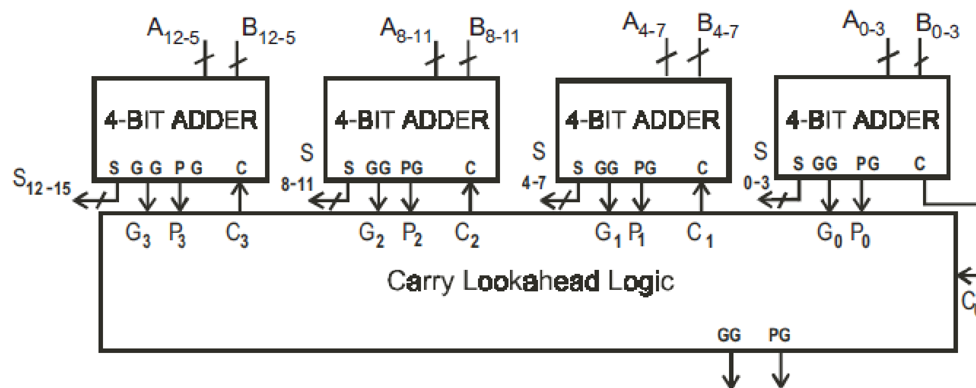



**Fig 1.33 Block diagram of a 16-bit CLA Adder**

- The disadvantage of the carry-look ahead adder is that the carry logic is getting quite complicated for more than 4 bits. For that reason, carry-look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits.
- Fig.1.33 shows the block diagram for a 16-bit CLA adder. The circuit makes use of the same CLA Logic block as the one used in the 4-bit adder. Notice that each 4-bit adder provides a group Propagate and Generate Signal, which is used by the

CLA Logic block. The group Propagate PG of a 4-bit adder will have the following expressions,

PG = P3.P2.P1.P0

GG = G3+P3 G2+P3.P2.G1.+P3.P2.P1.G0

## BCD

## Adder

## Definition:

### BCD adder:

A BCD adder is a combinational logical circuit which adds two BCD numbers.

- A BCD adder is a circuit that adds two BCD digits in parallel and produces sum which is also BCD. BCD number uses 10 symbols ( group of 4 bits 0000 to 1001). BCD adder circuit must be able to do the following and it is shown in Fig.1.41. Add two 4 bit BCD numbers using straight binary addition.
- If 4 bit sum is equal to or less than 9, the sum is valid BCD number and no correction needed. If the 4 bit sum is greater than 9, or if a carry is generated from the sum, the sum an invalid BCD number. Then, the digit 6 (0110) should be added to the sum to produce the valid BCD symbols.

**Table 1.17 Truth table of BCD adder**

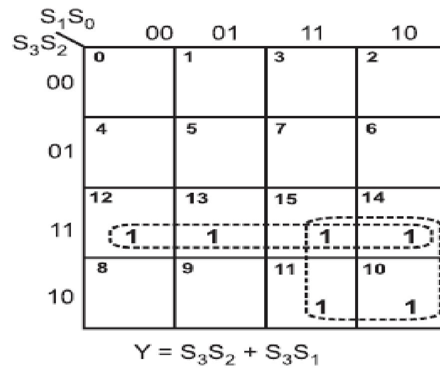
| Sums | | | | Correction required |
|---|---|---|---|---|
| $S_3$ | $S_2$ | $S_1$ | $S_0$ | Y |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Valid BCD sum

Invalid BCD sum
we need correction

- Table 1.17 shows the truth table. In this table, the inputs are sum bit from the two BCD additions and the output Y is indicating the required correction place.
- If Y = 0 (no correction needed) for 0000 to 1001 and Y = 1 (correction needed) for
- 1010 to 1111. From this truth table we get the simplified expression by using 4 variable K- map.

**Expression for Y**




## Logic Circuit

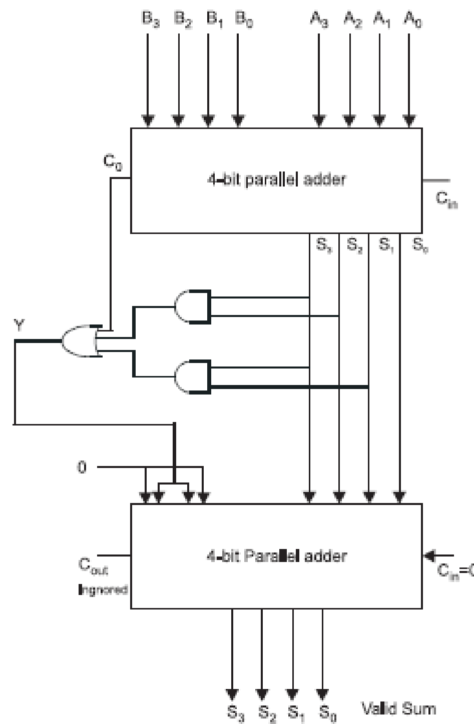



Fig 1.41 Logic diagram of BCD adder

## MAGNITUDE COMPARATOR

- The basic function of a comparator is to compare the magnitudes of two quantities to determine the relationship of those quantities.
- The XNOR gate can be used as a basic comparator because its output is a 1 when two inputs are equal. If the two input bits are not equal it is 0 if the input bits are equal.

## One bit magnitude comparator

- Magnitude comparator is to compare the three results: equal to, less than and greater than. Fig.1.44 shows the logic diagram for 2 bit comparator.

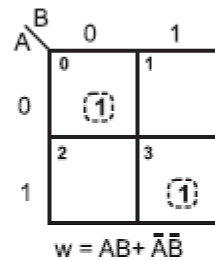
Number of required inputs = 2 (A,B)

Number of required outputs = 3 (A=B,  A<B,  A>B)
(w, x, y)

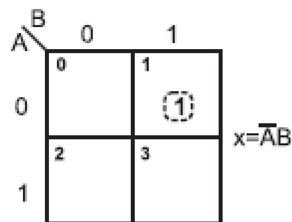
**Table 1.18 Truth table of 2-bit magnitude comparator**

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

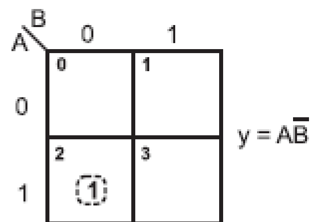
**K-map simplification**
**Expression for A=B(w)**




**K-map simplification**
**Expression for A<B(x)**




**K-map simplification**
**Expression for A<B(y)**



**Logic diagram**

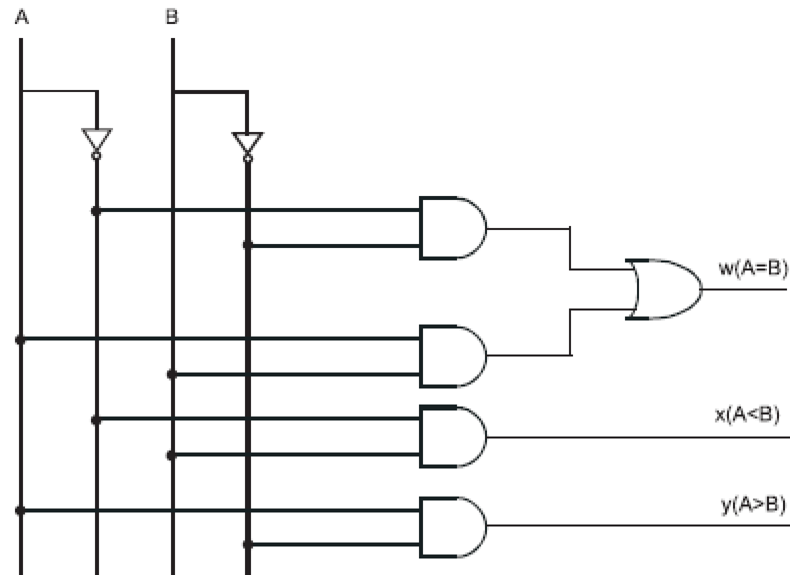



Fig 1.44 Logic diagram for 1 - bit comparator

## 2 - Bit Magnitude Comparator

Table 1.19 Truth table for 2 bit magnitude comparator

| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| $A_1$ | $A_0$ | $B_1$ | $B_0$ | $A_1A_0 = B_1B_0(w)$ | $A_1A_0 < B_1B_0(x)$ | $A_1A_0 > B_1B_0(y)$ |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 |

- A two bit comparator compares the magnitude of two bits and the logic diagram design is as follows. It is shown in Fig.1.45.

- Number of required inputs = 4 (*A*1 *B*1 *A*0 *B*0)
- Number of required outputs = 3 (w, x, y)
- when w is equal magnitude representation A1 A0 = B1 B0
- x is less than magnitude representation A1 A0 < B1 B0
- y is greater magnitude representation A1 A0 > B1B0

## Logic diagram

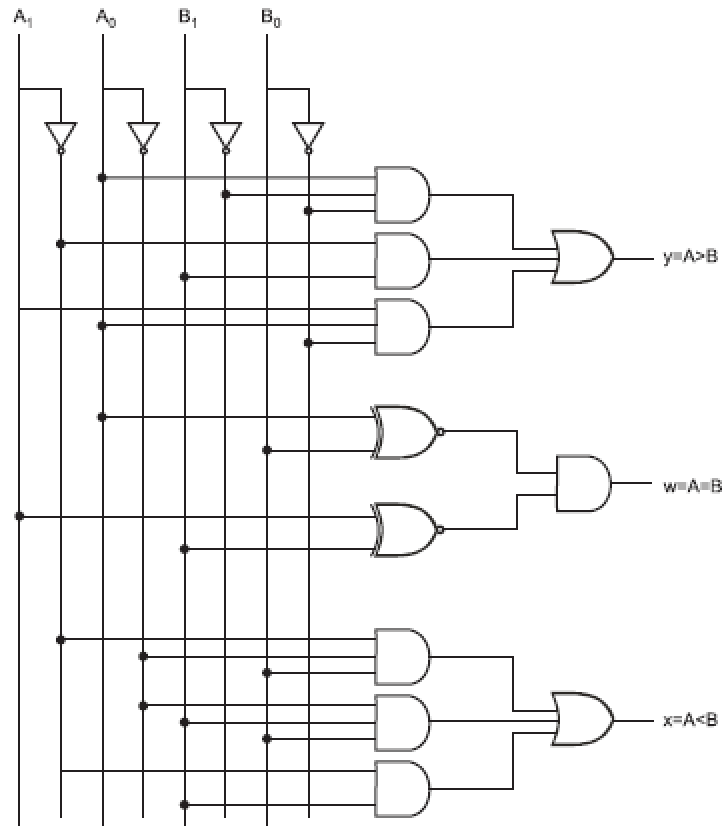



**Fig 1.45 Logic diagram for 4-bit magnitude comparator**

## DECODER

## Definition:

- A digital circuit designed to detect the presence of a particular digital state. It is a combinational logic circuit that converts binary information from n input lines to a maximum 2n unique output lines. Discrete quantities of information are represented in digital system by binary codes.
- A binary code of n bits is capable of representing upto 2n distinct elements of coded information. A decoder is a combinational logic circuit that converts binary information from n input lines to a maximum of 2n unique output lines.
- If the n-bit code information has unused combinations, the decoder may have fewer than 2n outputs. Fig.1.47 shows the block diagram of decoder.

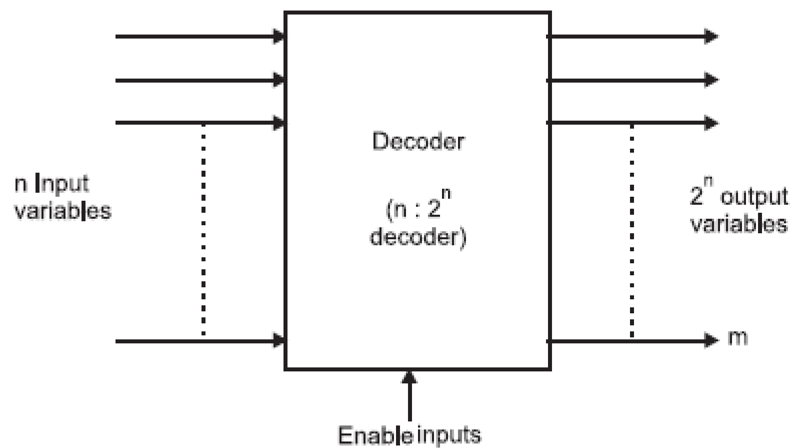



Fig 1.47 Block diagram of decoder

## 2 to 4 Binary decoder

- Fig.1.48 shows the 2 to 4 decoder. Here 2 represent the input lines and 4 represent output lines.
- The table 1.20 shows the truth table for a 2 to 4 decoder. If Enable is 1, one and only of the outputs Y0 to Y3 is active for a given input.
- The Y0 is active when inputs AB = 00, the output Y1 is active when the inputs AB = 01, similarly the outputs Y2 and Y3 are active when the input AB is 10 and 11 respectively. If Enable is 0 then all the outputs are 0.
- In general, a decoder may operate with complemented (or uncomplemented outputs. The enable input may be activated with 0 or with a 1 signal. Some decoders have two (or) more enable inputs that must satisfy a given logic condition in order to enable the circuit. A decoder with enable input can function as a demultiplexer.

**Table 1.20 Truth table for**

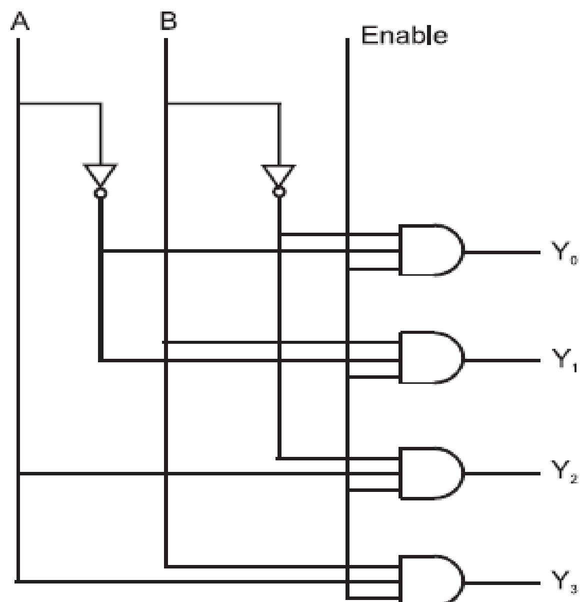



**Fig 1.48 logic diagram for 2 to 4 decoder**

Table 1.20 Truth table for a 2 to 4 decoder

| Inputs | | | Outputs | | | |
|---|---|---|---|---|---|---|
| E | A | B | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 0 | X | X | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

## 3 to 8 Decoder

- In the 3 to 8 decoder circuit, the 3 inputs are decoded into eight outputs represent the minterms of the 3 input variables.
- The three inverters provide the complement of the inputs and each one of the eight AND gates generates one of the minterms.
- The inputs are A, B, C. Fig.1.49 shows the logic diagram and the truth table of 3 to 8 decoder is shown in the Table 1.21 for active low output.

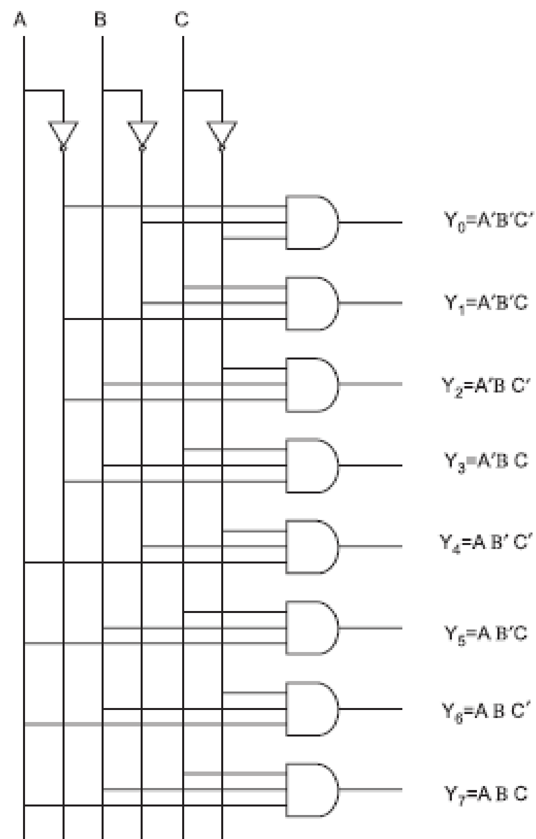



Fig 1.49 Logic diagram os 3 to 8 decoder

Table 1.21 Truth table of 3 to 8 decoder for active low output

| Inputs | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $A_2$ | $A_1$ | $A_0$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ | $Y_4$ | $Y_5$ | $Y_6$ | $Y_7$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

## ENCODER

### Definition of Encoder

- A circuit that generates a binary code at its outputs in response to one or more active input lines An encoder is a combinational logic circuit, it is a reverse decoder function. It has 2*n* (or fever) input lines and n output lines.
- In encoder accepts an active 1.80 Digital Logic Circuits level on one of its inputs representing a digit such as a decimal (on octal digit and comments it to a coded output). Fig.1.55 shows the block diagram of a encoder.

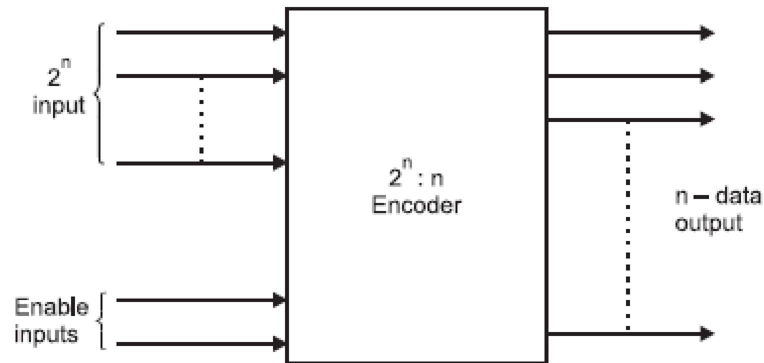



Fig 1.55 Block diagram of a encoder

## Octal to Binary Encoder

- In the octal to binary encoder has eight inputs, one for each octal digit and outputs that generates the corresponding binary code.
- In encoders it is assumed that only one input has a value of 1 at any given time. Table 1.23 shows the truth table of octal to binary encoder and Fig.1.56 shows the octal to binary encoder circuit.

Table 1.23 Truth table of octal to binary encoder

| Data Inputs ( octal digit) | | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ | A | B | C |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

From truth table out put A having 1 when the inputs 4, 5, 6 and 7 are appear so that the equations are written as follows.

A = D4+D5+D6+D7

Similarly, the Boolean expressions for output B and C are written as follows

B = D2+D3+D6+D7

C = D1+D3+D5+D7

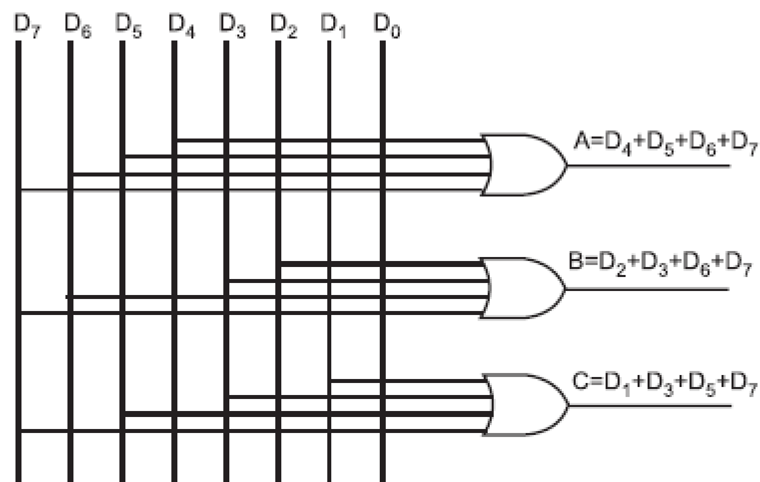



**Fig 1.56 Logic diagram of octal to binary encoder**

## Decimal to BCD encoder

- This type of encoder has ten inputs - one for each decimal digit and four outputs corresponding to the BCD code

Table 1.24 Truth table Decimal to BCD encoder

| Decimal digit | BCD Code | | | |
|---|---|---|---|---|
| | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 1 |

- Table 1.24 shown the relation between the decimal and BCD code. A3 is most significant bit of the BCD code. A3 is always 1 for decimal digit 8 or 9. The expression for bit A3 minterms of the decimal digits can written as.

  A3 = 8+9

  Bit A2 is always 1 for decimal digit 4,5,6 or 7 can be expressed as on OR function as follows. A2 = 4+5+6+7

  Bit A1 is always 1 for decimal digit 2, 3, 6 or 7 and can be expressed as

  A1 = 2+3+6+7

  Bit A0 is always 1 for decimal digit 1, 3, 5, 7 or 9 the expression for A0 is

  A0 = 1+3+5+7+9

  Now, we can draw the decimal to BCD encoder by using the above four expression Fig.1.57 shows the decimal to BCD encoder.

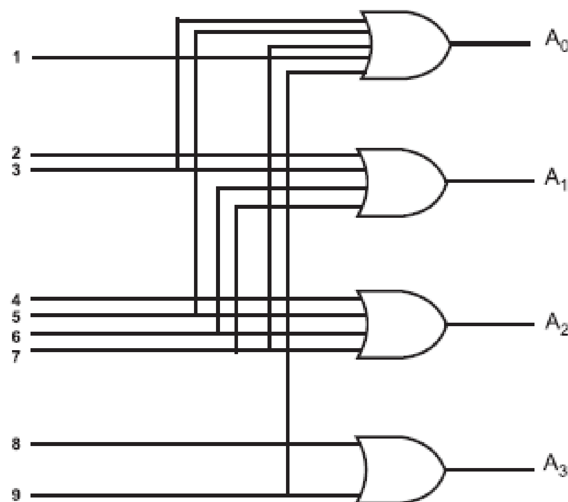



**Fig 1.57 Logic diagram for decimal to BCD encoder**

- When a HIGH is appears on one of the decimal digit input lines, the appropriate levels occur on the four BCD output lines.

## CODE CONVERTER:

- Code converter is a combinational logic circuit to convert one form of code to another form of code. Some of these codes are binary coded decimal, Excess -3 code, Gray code and so on. Many times it is required to convert one code to another.

### Binary to BCD Converter

Table 1.26 Truth Table for Binary to BCD converter

| Binary code | | | | BCD code | | | | |
|---|---|---|---|---|---|---|---|---|
| D | C | B | A | $B_4$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |

| Binary code | | | | BCD code | | | | |
|---|---|---|---|---|---|---|---|---|
| D | C | B | A | $B_4$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

- A code converter combinational circuit is designed to convert binary to BCD code. Fig.1.60 shows the logic diagram of Binary to BCD code converter.
- The input code of code converter is binary. The output code of code converter is BCD code.

## K-Map Simplification

**Expression for $B_0$**

| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1<br>1 | 3<br>1 | 2 |
| 01 | 4 | 5<br>1 | 7<br>1 | 6 |
| 11 | 12 | 13<br>1 | 15<br>1 | 14 |
| 10 | 8 | 9<br>1 | 11<br>1 | 10 |

$B_0 = A$

**Expression for $B_1$**

| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 3<br>1 | 2<br>1 |
| 01 | 4 | 5 | 7<br>1 | 6<br>1 |
| 11 | 12<br>1 | 13<br>1 | 15 | 14 |
| 10 | 8 | 9 | 11 | 10 |

$B_1 = DCB' + D'B$

**Expression for $B_2$**

| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 3 | 2 |
| 01 | 4<br>1 | 5<br>1 | 7<br>1 | 6<br>1 |
| 11 | 12 | 13 | 15<br>1 | 14<br>1 |
| 10 | 8 | 9 | 11 | 10 |

$B_2 = D'C + CB$

**Expression for $B_3$**

| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 3 | 2 |
| 01 | 4 | 5 | 7 | 6 |
| 11 | 12 | 13 | 15 | 14 |
| 10 | 8<br>1 | 9<br>1 | 11 | 10 |

$B_3 = DC'B'$

**Expression for $B_4$**

| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 3 | 2 |
| 01 | 4 | 5 | 7 | 6 |
| 11 | 12<br>1 | 13<br>1 | 15<br>1 | 14<br>1 |
| 10 | 8 | 9 | 11<br>1 | 10<br>1 |

$B_4 = DC + DB$

### Logic Diagram

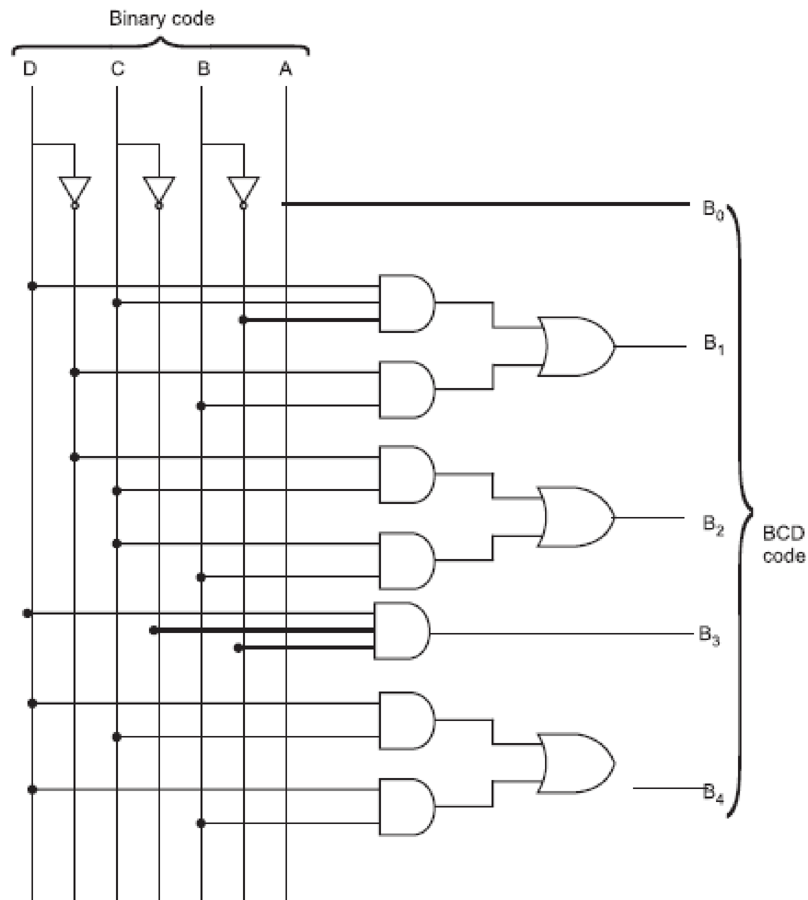
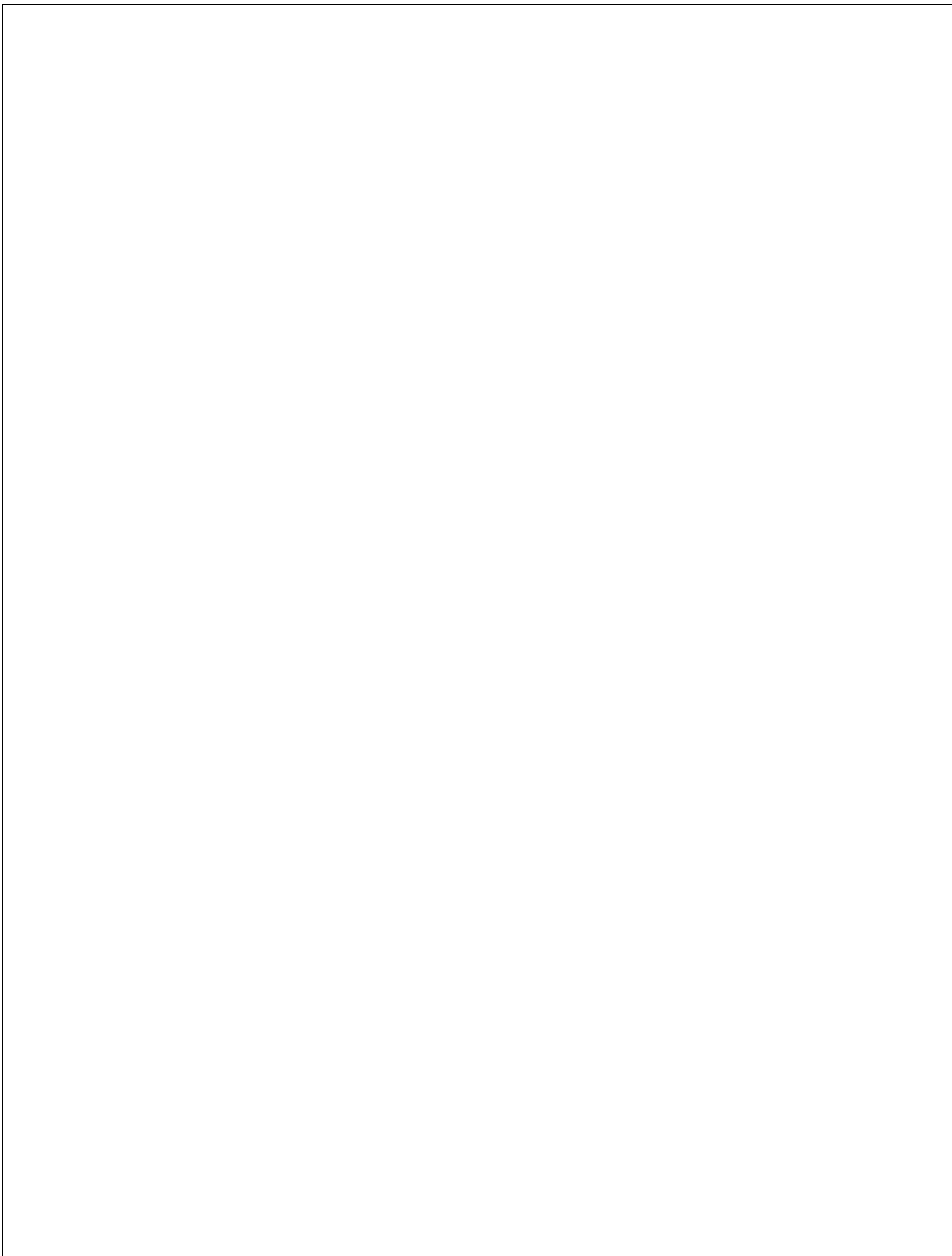



Fig 1.60 Logic diagram of Binary to BCD code converter

## BCD TO EXCESS -3 CODE CONVERTER

- A code converter combinational circuit is designed to convert BCD code to Excess 3 code. The input code of code converter is BCD code.
- The output code of code converter is Excess-3 code. Fig.1.62 shows the logic diagram of BCD t excess-3 code converter The unused states are 1010, 1011, 1100, 1101, 1110 and 1111. So place X (Don’t Care Condition) for the corresponding cells.



## Logic Diagram

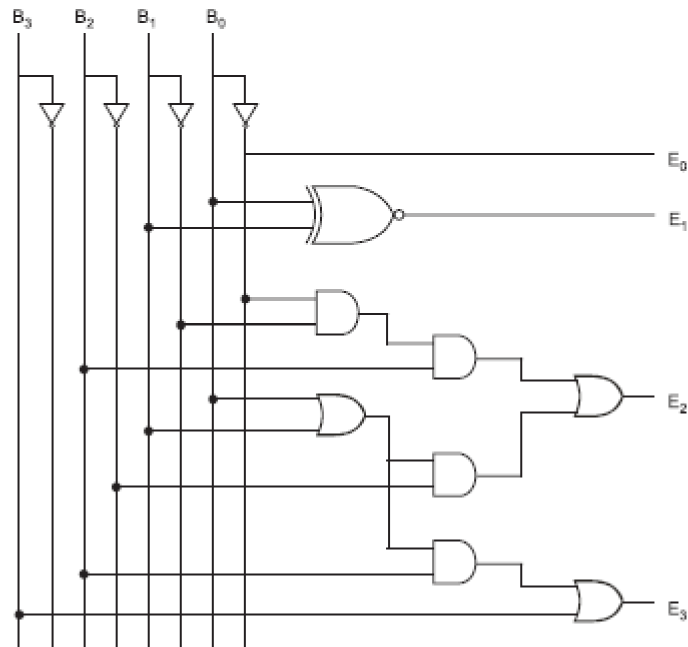



Fig 1.61 Logic diagram of BCD to Excess-3 code converter

## EXCESS-3 CODE TO BCD CODE CONVERTER

- A code converter combinational circuit is designed to convert Excess - 3 code to BCD code. The input code of code converter is Excess - 3. The output code of code converter is BCD code. Fig.1.61 shows the logic diagram of excess -3 code to BCD code converter.
- The unused states are 0000, 0001, 0010, 1101, 1110 and 1111. So place X (Don't Care Condition) for the corresponding above cells.

Table 1.28 Truth table for excess -3 code to BCD code

| Excess-3 code | | | | BCD Code | | | |
|---|---|---|---|---|---|---|---|
| $E_3$ | $E_2$ | $E_1$ | $E_0$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

## K-Map Simplification

**Expression for $B_0$**

| $E_3E_2$ \ $E_1E_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 X | 1 X | 3 | 2 X |
| 01 | 4 1 | 5 | 7 | 6 1 |
| 11 | 12 1 | 13 X | 15 X | 14 X |
| 10 | 8 1 | 9 | 11 | 10 1 |

$B_0 = E_0'$

**Expression for $B_1$**

| $E_3E_2$ \ $E_1E_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 X | 1 X | 3 | 2 X |
| 01 | 4 | 5 X | 7 | 6 X |
| 11 | 12 | 13 X | 15 X | 14 X |
| 10 | 8 | 9 1 | 11 X | 10 1 |

$B_1 = E_1'E_0 + E_1E_0' = E_1 \oplus E_0$

**Expression for $B_2$**

| $E_3E_2$ \ $E_1E_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 X | 1 X | 3 | 2 X |
| 01 | 4 | 5 | 7 1 | 6 |
| 11 | 12 | 13 X | 15 X | 14 X |
| 10 | 8 X | 9 X | 11 X | 10 1 |

$B_2 = E_2'E_1' + E_2E_1E_0 + E_3E_1E_0'$

**Expression for $B_3$**

| $E_3E_2$ \ $E_1E_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 X | 1 X | 3 | 2 X |
| 01 | 4 | 5 | 7 | 6 |
| 11 | 12 1 | 13 X | 15 X | 14 X |
| 10 | 8 | 9 | 11 1 | 10 |

$B_3 = E_3E_2 + E_3E_1E_0$

## Logic Diagram

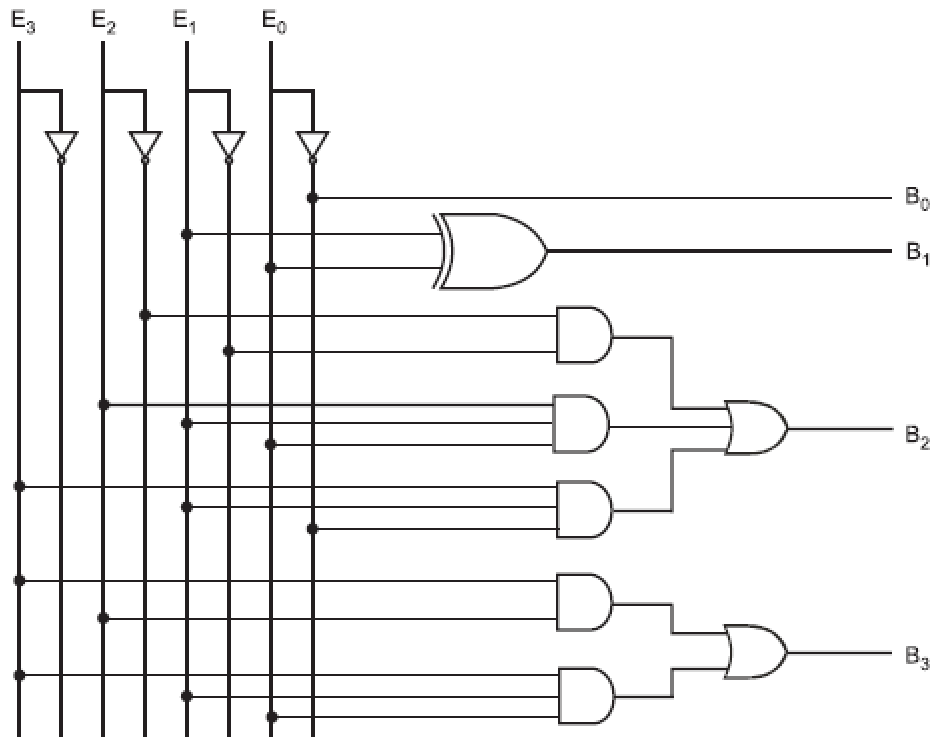



Fig 1.62 Logic diagram of Excess-3 to BCD code converter

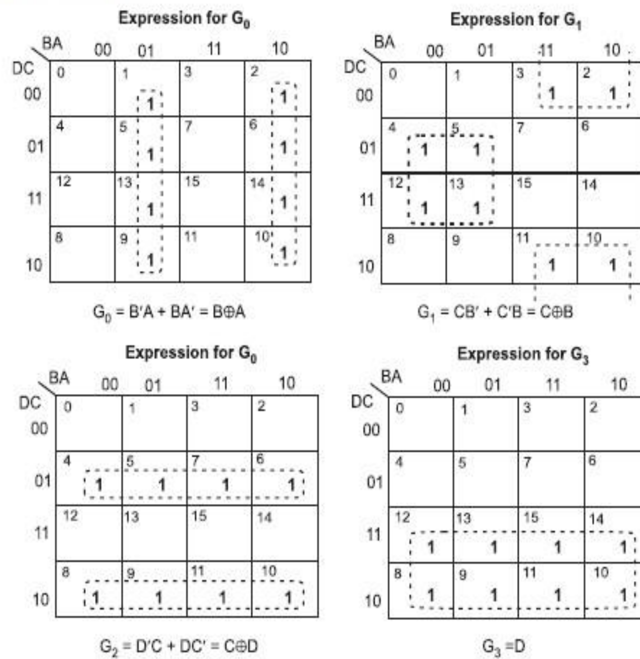
## BINARY CODE TO GRAY CODE CONVERTER:

- A code converter combinational circuit is designed to convert binary to gray code. Fig.1.63 shows the logic diagram of binary to gray code converter.
- The input code of code converter is binary and the output code of code converter is gray code.

**Table 1.29 Truth Table for Binary to Gray code converter**

| Binary code | | | | Gray code | | | |
|---|---|---|---|---|---|---|---|
| D | C | B | A | $G_3$ | $G_2$ | $G_1$ | $G_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

## K-Map Simplification




- We get the simplified boolean expression for the code convertor of Binary to Gray code.

$G_0 = B_A+BA_ = B\oplus A$

$G_1 = CB_+C_B = C\oplus B$

$G_2 = D_C+DC_ = C\oplus D$

$G_3 = D$

- By using the above expression we can construct the binary to gray code convertor as shown in Fig.1.63.

# Logic diagram

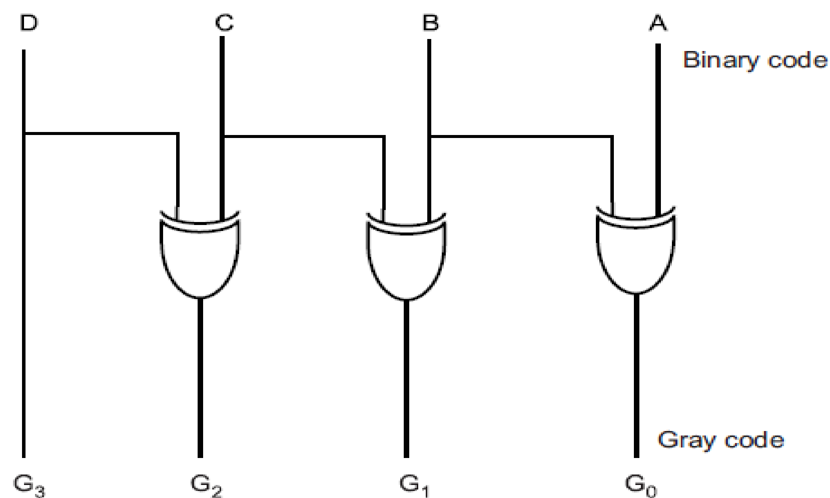



Fig 1.63 Logic diagram of Binary to gray code converter

## GRAY CODE TO BINARY CODE CONVERTER:

- A code converter combinational circuit is designed to convert gray code to binary code. The input code of code converter is gray. The output code of code converter is binary code. Fig.1.64 shows the logic diagram of gray to binary code converter.

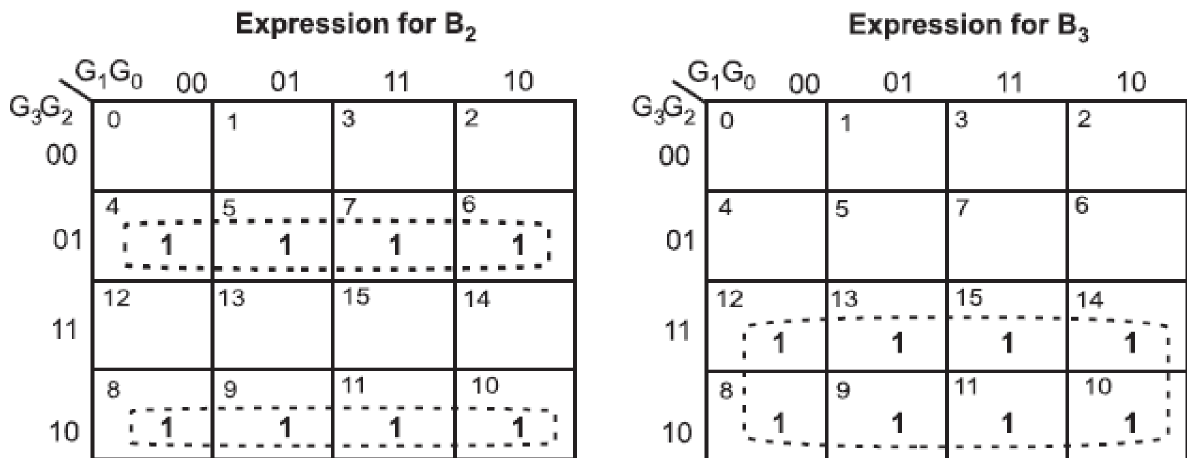
## K-Map Simplification

**Expression for $B_0$**

| $G_3G_2$ \ $G_1G_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1: 1 | 3 | 2: 1 |
| 01 | 4: 1 | 5 | 7: 1 | 6 |
| 11 | 12 | 13: 1 | 15 | 14: 1 |
| 10 | 8: 1 | 9 | 11: 1 | 10 |

**Expression for $B_1$**

| $G_3G_2$ \ $G_1G_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 3: 1 | 2: 1 |
| 01 | 4: 1 | 5: 1 | 7: 1 | 6 |
| 11 | 12 | 13 | 15: 1 | 14: 1 |
| 10 | 8: 1 | 9: 1 | 11 | 10 |




## Logic Diagram

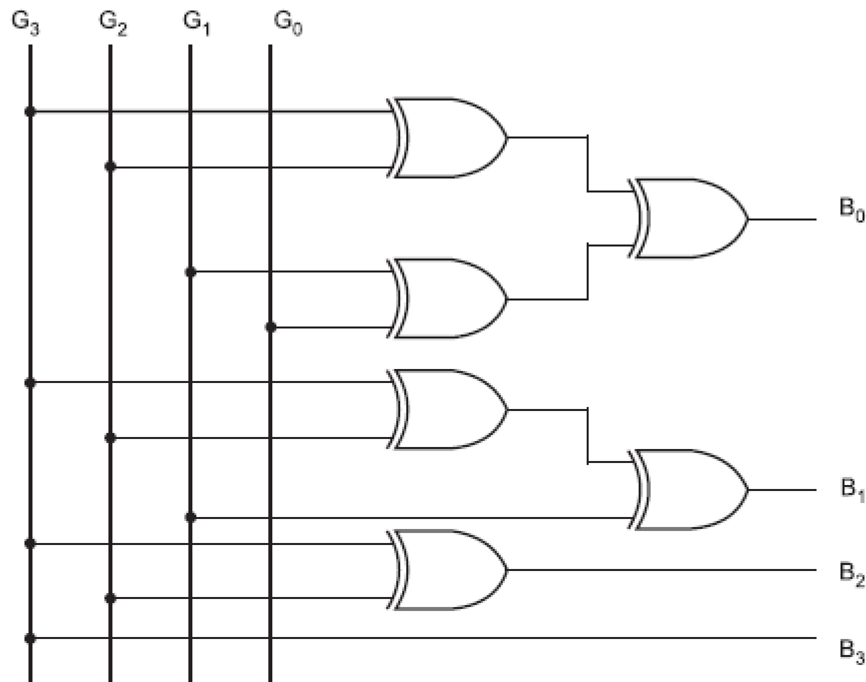



Fig 1.64 Logic diagram of Gray code to Binary code converter

## MULTIPLEXER:

### Definition of Multiplexer

- A circuit that directs one of several digital signals to a single output depending upon the state of several select inputs. Boolean Algebra and Combinational Circuits 1.95
- **Data inputs** The multiplexer inputs that feed a digital signal to the output when selected. (Maximum of inputs is $2^n$)
- **Select inputs** The multiplexer inputs that selects the digital input (Maximum "n" select lines).

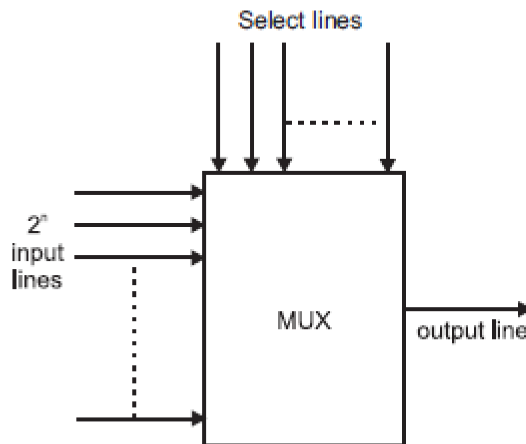



Fig 1.65 Block diagram of Multiplexer

- 'Multiplex' means many to one. Multiplexing is the process of transmitting a large number of information over a single line. A digital multiplexer (MUX) is a combinational logic circuit that selected information on a single output.
- A multiplexer is also called a data selector. Fig.1.65 shows the block diagram of multiplexer. Normally, there are 2n input lines and n selection lines and one output line. The selection of a particular input line is controlled by the set of select lines. The size of the multiplexer is specified by number 2n input lines and the single output line.

## 4 to 1 Multiplexer

- The 4 to 1 multiplexer, the 4 represent the number of inputs and one represent the output line. The two select lines (2n = 4;n = 2) S1 and S0 to select one of the four inputs. Table 1.31 shows the truth table of 4 to 1 multiplexer.
- From the truth table, the logical expression for the output in term of data input (I0, I1, I2, I3) and select lines can be derived as follows and Fig.1.66 shows the logic diagram of 4 : 1 MUX.

**Table 1.31 Truth table of 4:1 Multiplexer**

| Select lines | | Output |
|---|---|---|
| $S_1$ | $S_0$ | Y |
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

The data output $Y = I_0$, if only $S_1$ and $S_0 = 0$

Therefore $Y = I_0 \, \overline{S}_1 \, \overline{S}_0$

The data output $Y = I_1$ when $S_1 = 0$ and $S_0 = 1$

Therefore $Y = I_2 \, \overline{S}_1 \, S_0$

Similarly $Y = I_2$ when $S_1 = 1$ and $S_0 = 0$, therefore $Y = I_2 \, S_1 \, \overline{S}_0$

$Y = I_3 \, S_1 \, S_0$ when $S_1 = 1$ and $S_0 = 1$

The above output expressions are ORed, to get the final expression for the data output is given by

$$Y = I_0 \, \overline{S}_1 \, \overline{S}_0 + I_1 \overline{S}_1 \, S_0 + I_2 \, S_1 \, \overline{S}_0 + I_3 \, S_1 \, S_0$$

## Logic Diagram

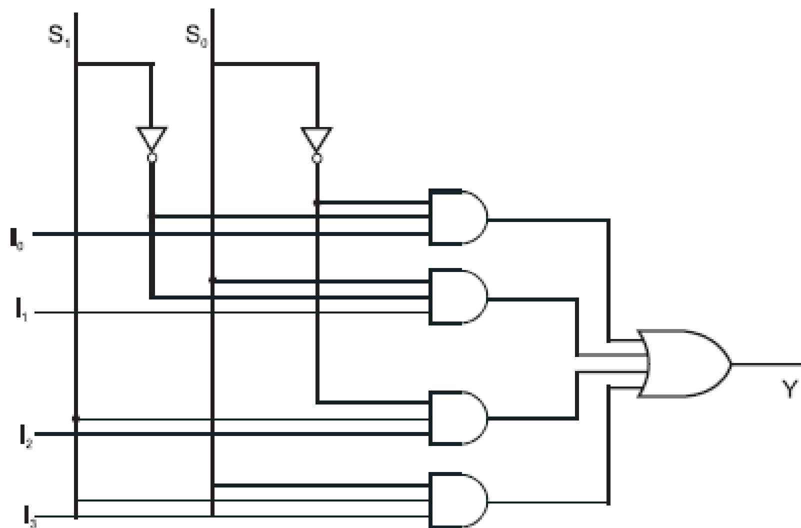
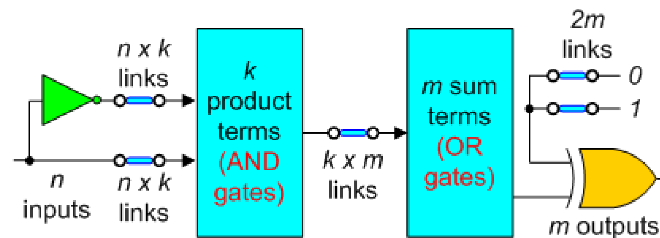



Fig 1.66 Logic diagram of multiplexer

## PLA (PROGRAMMABLE LOGIC ARRAY):

- In PLAs, instead of using a decoder as in PROMs, a number (**k**) of AND gates is used where $\mathbf{k < 2^n}$, (**n** is the number of inputs).
- Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM.
- The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them. The specific Boolean functions are implemented in sum of products form by opening appropriate links and leaving the desired connections.
- A block diagram of the PLA is shown in the figure. It consists of **n** inputs, **m** outputs, and **k** product terms.




The product terms constitute a group of **k** AND gates each of 2n inputs.

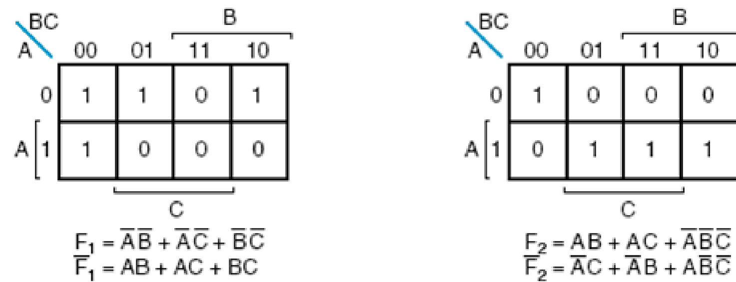
- Links are inserted between all **n** inputs and their complement values to each of the AND gates. Links are also provided between the outputs of the AND gates and the inputs of the OR gates.
- Since PLA has **m**-outputs, the number of OR gates is **m**.
- The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the **true** form or in the **complement** form.
- The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X'$).
- The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$).
- Thus, the total number of programmable links is **2n x k + k x m + 2m**.
- The size of the PLA is specified by the number of inputs **(n)**, the number of product terms **(k)**, and the number of outputs **(m)**, (the number of sum terms is equal to the number of outputs).

## Example:

- Implement the combinational circuit having the shown truth table, using PLA.

| *A* | *B* | *C* | $F_1$ | $F_2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 |

- Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.




The combination that gives a minimum number of product terms is:

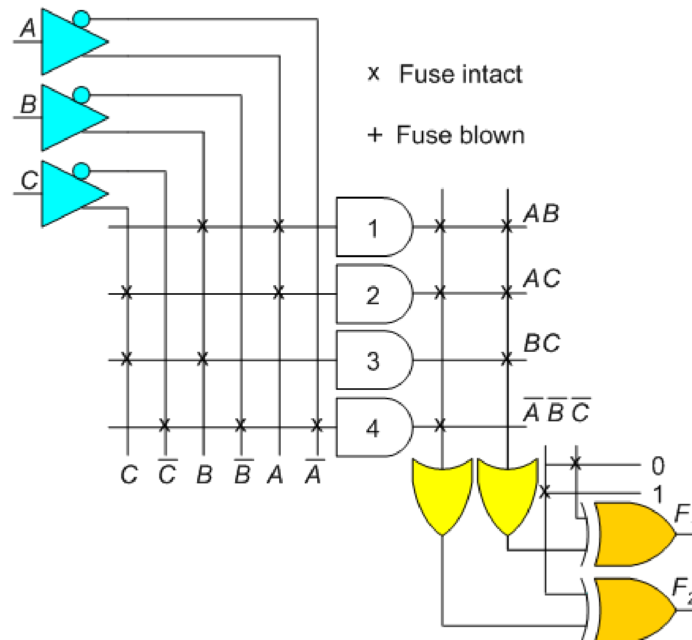
$F_1 = AB + AC + BC$ or $F_1 = (AB + AC + BC)'$

$F_2 = AB + AC + A'B'C'$

This gives only 4 distinct product terms: AB, AC, BC, and A'B'C'.

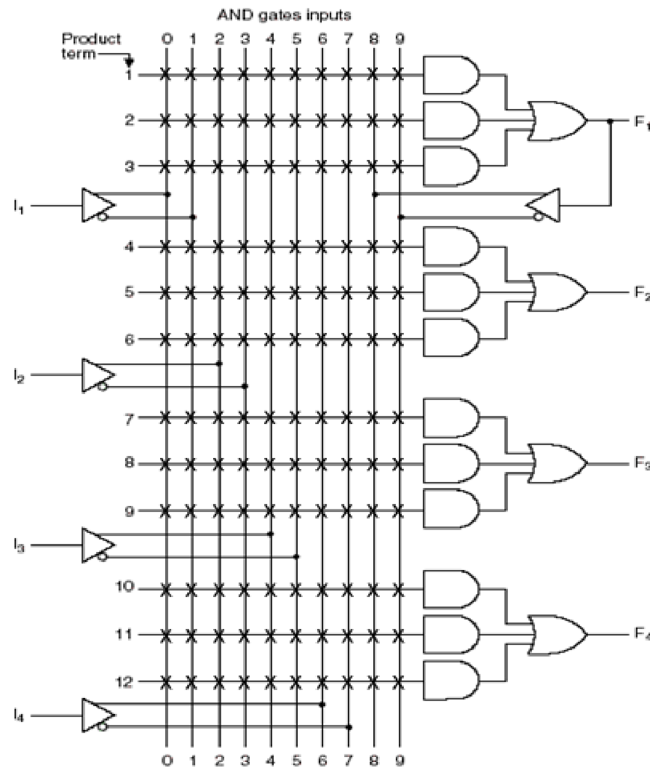
PLA programming table

| | Product term | Inputs A B C | Outputs (C) $F_1$ | (T) $F_2$ |
|---|---|---|---|---|
| AB | 1 | 1 1 – | 1 | 1 |
| AC | 2 | 1 – 1 | 1 | 1 |
| BC | 3 | – 1 1 | 1 | – |
| $\overline{A}\overline{B}\overline{C}$ | 4 | 0 0 0 | – | 1 |



## PAL (PROGRAMMABLE ARRAY LOGIC):

- The PAL device is a PLD with a fixed OR array and a programmable AND array.
- As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA.




- The device shown in the figure has 4 inputs and 4 outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate.
- The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section.
- Each AND gate has 10 programmable input connections indicating by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate.
- One of the outputs $\mathbf{F_1}$ is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections.

**Example:**
Implement the following Boolean functions using the PAL device as shown above:

$W(A, B, C, D) = \Sigma m(2, 12, 13)$
$X(A, B, C, D) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$
$Y(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$
$Z(A, B, C, D) = \Sigma m(1, 2, 8, 12, 13)$

Simplifying the 4 functions to a minimum number of terms results in the following Boolean functions:

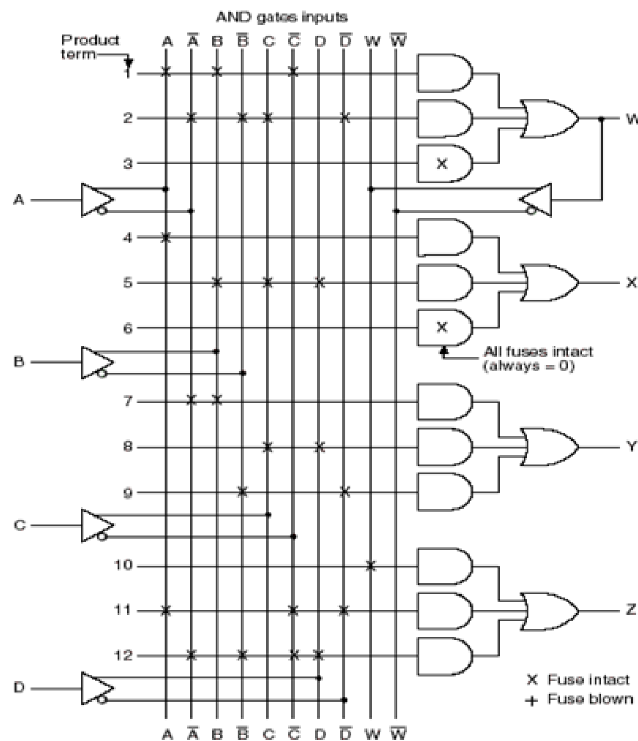
W = ABC’ + A’B’CD’X =

A + BCD

Y = A’B + CD + B’D’

Z = ABC’ + A’B’CD + AC’D’ + A’B’C’D =W +AC’D’ + A’B’C’D

| Product term | AND Inputs | | | | | Outputs |
|---|---|---|---|---|---|---|
| | A | B | C | D | W | |
| 1 | 1 | 1 | 0 | — | — | $W = AB\overline{C}$ |
| 2 | 0 | 0 | 1 | 0 | — | $+\overline{A}\,\overline{B}C\overline{D}$ |
| 3 | — | — | — | — | — | |
| 4 | 1 | — | — | — | — | $X = A$ |
| 5 | — | 1 | 1 | 1 | — | $+BCD$ |
| 6 | — | — | — | — | — | |
| 7 | 0 | 1 | — | — | — | $Y = \overline{A}B$ |
| 8 | — | — | 1 | 1 | — | $+CD$ |
| 9 | — | 0 | — | 0 | — | $+\overline{B}\,\overline{D}$ |
| 10 | — | — | — | — | 1 | $Z = W$ |
| 11 | 1 | — | 0 | 0 | — | $+A\overline{C}\,\overline{D}$ |
| 12 | 0 | 0 | 0 | 1 | — | $+\overline{A}\,\overline{B}\,\overline{C}D$ |



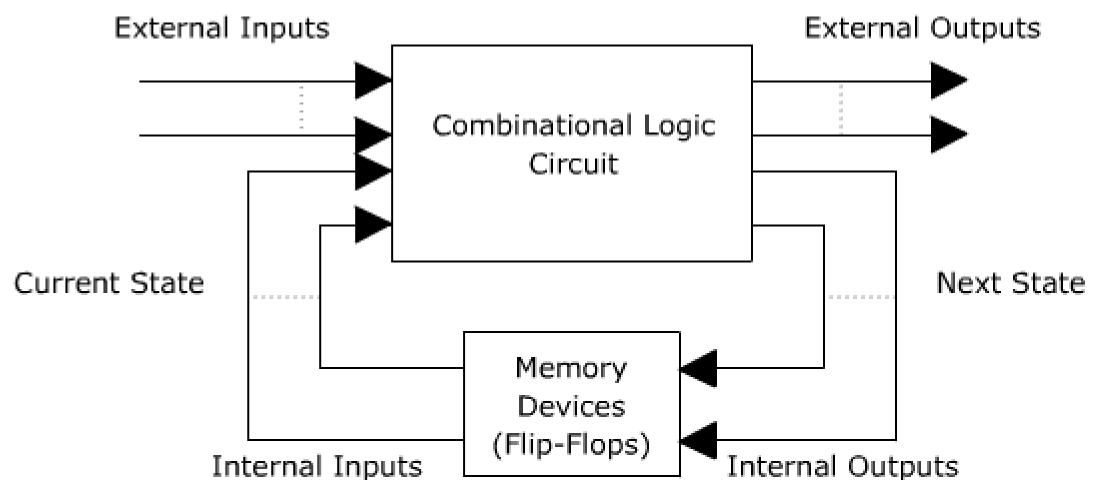

**UNIT – IV**

**Sequential Circuits:** General model of sequential circuits –latches – Master-slave Configuration- Flip-Flops - Concept of State – State diagram – State Table. Synchronous Sequential Circuits – Binary ripple counters-Design of Synchronous counters- binary counters-Arbitrary sequence counter - BCD counter – Shift Registers – Ring Counter – Johnson Counter – Timing diagram – Serial Adder – PN sequence generator.

**Sequential PLDs** - Introduction to CPLD and Field programmable Gate Array (FPGA).

## Sequential Circuits

- Combinational circuits and systems produce an output based on input variables only.
- Sequential circuits use current input variables and previous input variables by storing the information and putting back into the circuit on the next clock (activation) cycle.




- The figure above shows a theoretical view of how sequential circuits are made up from combinational logic and some storage elements.
- There are two types of input to the combinational logic; External inputs which come from outside the circuit design and are not controlled by the circuit; Internal inputs which are a function of a previous output states.

- The internal inputs and outputs are refered to as "secondaries" in the course notes. Secondary inputs are state variables produced by the storage elements, where as secondary outputs are excitations for the storage elements.

## Types of Sequential Circuits

- A sequential circuit is specified by a time sequence of inputs, outputs and internal states. The sequential circuits can be classified in two ways depending on the timing of their signals; There are:

  Synchronous sequential circuits.

  Asynchronous sequential circuits.
- A synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time, i.e., signals can affect the memory elements only at discrete instants of time. These are also called as “clocked sequential circuits”. Fig.2.2 shows the block diagram of synchronous clocked sequential circuit.

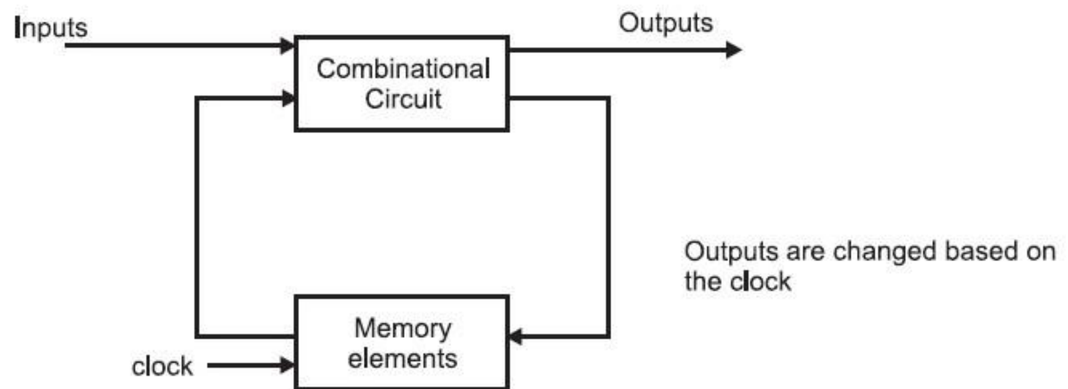



Fig 2.2 Synchronous clock sequential circuit

- In asynchronous sequential circuit, change in input signals can affect memory element at any instant of time, i.e., the behaviour of an asynchronous sequential circuit depends upon the order in which its input signals change and can be affect at any instant of time. These are also called as unclocked sequential circuits.
- The memory elements used in both cases are flip-flops which are capable of storing 1-bit binary information. A flip-flop circuit has two outputs one for the normal value and other for the complement value of the bit stored in it.

## Latches

- A flip-flop can maintain a binary state (either 0 or 1) as long as power is delivered to the flip-flop. The most basic types of flip-flops operate with signal levels and are referred to as latches.
- The latches are the basic circuits from which all flip-flops are constructed. Also latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use in synchronous sequential circuits.
- The basic difference between latches and flip-flops.

| | Latches | | Flip-flops |
|---|---|---|---|
| 1 | A latch checks all its inputs continuously and change its output accordingly at any time | 1 | Flip-flops samples its inputs and changes its outputs only at a time as determined by a clock signal |
| 2 | No clock is used | 2 | A clock is used. |
| 3 | It is used in asynchronous sequential logic circuit | 3 | It is used in synchronous sequential circuit |

## RS Latch

- The simplest latch is the Reset-Set latch (RS - latch). It can be constructed from either two NOR gates or two NAND gates.

## RS latch using NOR gates

- Fig.2.3 shows the RS latch using two NOR gates. The two NOR gates are cross coupled so that output of NOR gate 1 is connected to one of the inputs of NOR gate 2 and vice-versa.
- The latch has two outputs Q and Q and two inputs, set (S) and Reset (R). As we know that, logic 1 at any input of a NOR gate forces its output to a logic 0 immediately.

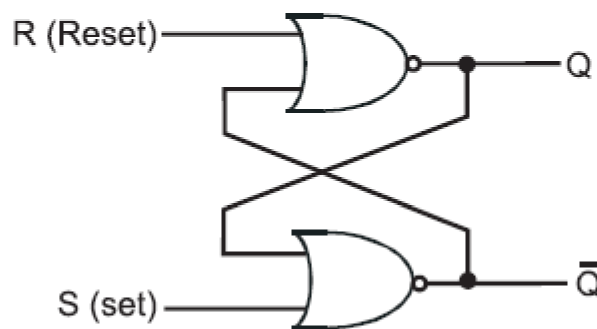



Fig 2.3 RS latch using NOR gates

## Operation of RS latch

- Let us see the operation of this circuit for various input possibilities

  Case (i) R = 0 and S = 1
- In this case, S input of NOR gate 2 is at logic 1, hence its output Q is at logic 0 which indicates that both inputs of NOR gate 1 are at logic 0. So that its output, Q is at logic 1 as shown in Fig.2.4(a).

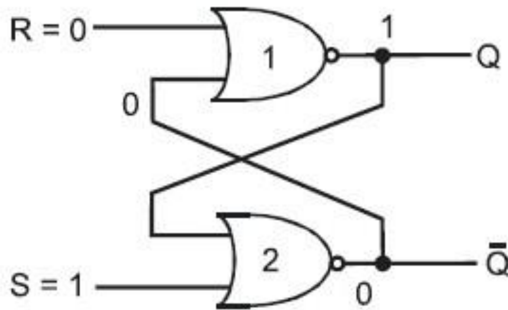



Fig.2.4(a)

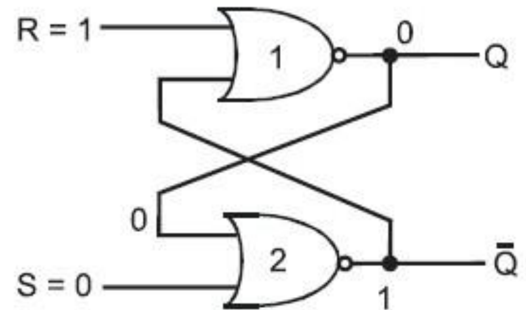



Fig.2.4(b)

### Case (ii) R = 1 and S = 0

- In this case, R input of the NOR gate 1 is at logic 1, hence its output Q is at logic 0.
- This indicates that both inputs of NOR gate 2 are now at logic 0, so its output Q s at logic 1 as shown in Fig.2.4(b).

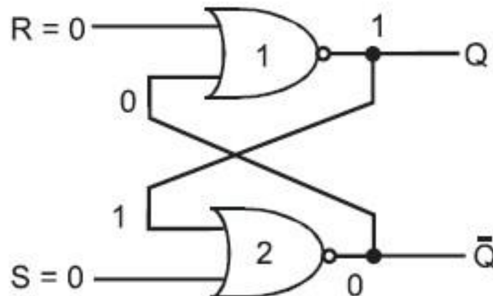



Fig.2.4(c)

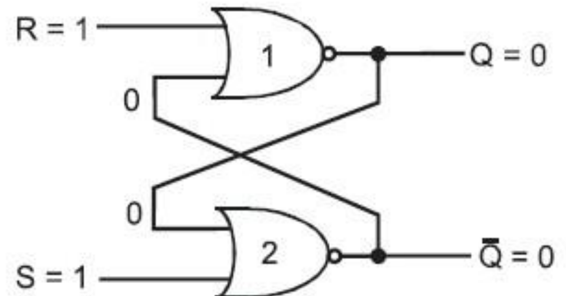



Fig.2.4(d)

### Case (iii) R = 0 and S = 0

- Let us assume that initially Q = 1 and Q = 0. With Q = 0, both inputs to NOR gate 1 are at logic 0. So its output, Q is at logic 1.
- With Q = 1, NOR gate 2 output Q is at logic 0. Thus the output is same as initial value. This same for Q = 0 and Q = 1 also. The diagram is shown in Fig.2.4(c).

### Case (iv) R = 1 and S = 1

- When both inputs are at logic 1, they force the outputs of both NOR gates to logic 0 i.e., Q = 0 and Q = 0. Since we know that Q is a logic binary variable and Q is its complement, then Q _= Q for any condition.
- So we call this condition or state as an "indeterminate state or prohibited state". Thus in normal operation, this condition must be avoided.

- This is shown in Fig.2.4(d). The symbol of RS latch is shown in Fig.2.5 and its truth table is shown in Table 2.1.

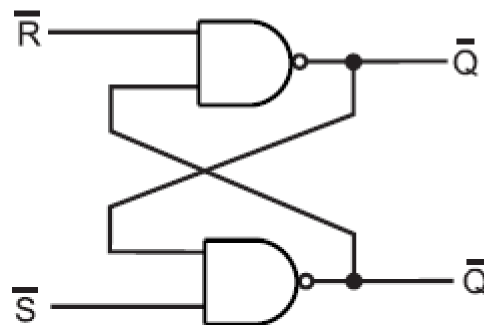
**Fig.2.5 Symbol of RS latch**

**Table 2.1 Truth of RS latch using NOR gates**

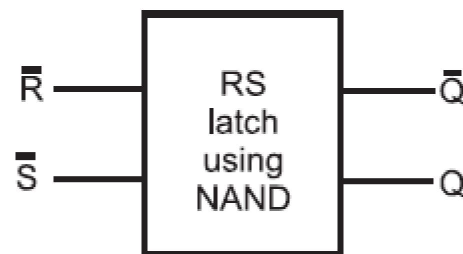
| Input | | Output | | State |
|---|---|---|---|---|
| R | S | Q | $\overline{Q}$ | |
| 0 | 0 | NC | NC | No change |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | X | X | Indeterminate |

## RS latch using NAND gates or $\overline{SR}$ Latch

- The active-LOW RS latch can be constructed using two cross-coupled NAND gates. Fig.2.6 shows the logic diagram of RS latch using NAND gates.




**(a) logic diagram**




**(b) Symbol**

**Fig 2.6 RS latch using NAND gates**

We know that a logic - 0 on any input to a NAND gate forces its output logic - 1.

## Operation

**Case (i)** If $\overline{S} = 1, \overline{R} = 0$, it will reset the $\overline{R}\,\overline{S}$ latch i.e., $Q = 0$ and $\overline{Q} = 1$

**Case (ii)** If $\overline{S} = 0, \overline{R} = 1$, it will set the $\overline{R}\,\overline{S}$ latch i.e., $Q = 1$ and $\overline{Q} = 0$

**Case (iii)** If $\overline{S} = \overline{R} = 1$, the latch will remain in its previous state

**Case (iv)** If $\overline{S} = \overline{R} = 0$, the latch is unpredictable

The truth table of $\overline{R}\,\overline{S}$ latch is shown in table 2.2.

**Table 2.2 Truth table $\overline{R}\,\overline{S}$ latch**

| Input | | Output | | State |
|---|---|---|---|---|
| $\overline{R}$ | $\overline{S}$ | Q | $\overline{Q}$ | |
| 1 | 1 | NC | NC | No change |
| 1 | 0 | 1 | 0 | Set |
| 0 | 1 | 0 | 1 | Reset |
| 0 | 0 | 1 | 1 | Indeterminate |

## Flip-Flops

- A flip-flop is a bistable multivibrator. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs.
- It is the basic storage element in sequential logic. Flip-flops and latches are a fundamental building block of digital electronics systems used in computers, communications, and many other types of systems.The types of flip-flops are:
  - RS flip-flop
  - JK flip-flop
  - D flip-flop
  - T flip-flop.

## Clocked RS flip-flop

- An RS flip-flop is shown in Fig.2.7(a). It consists of basic NOR latch circuit and two AND gates at the input.
- The AND gates remain '0' as long as the clock pulse CLK is '0' regardless of the R and S inputs. When the clock pulse goes to 1, information from R and S inputs is allowed to reach the basic RS latch. The basic symbol is also shown in Fig.2.7(b).

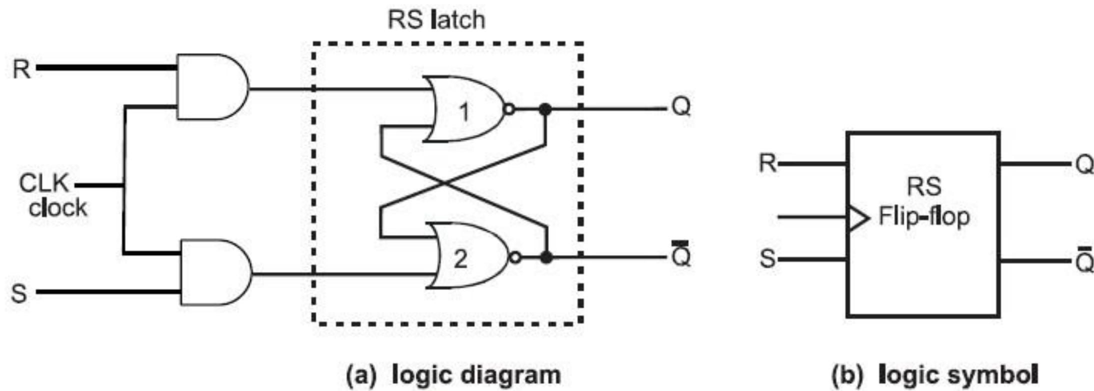


Fig 2.7 Clocked RS Flip-flop

## Operation

- When clock is absent or CLK =0, the circuit will retain same state.
- When clock is present or CLK =1, the RS flip-flop work as basic RS Latch.

## Case (i) If R = 0 and S = 0 and assume that Q = 1

- The output of both AND gates are ‘0’ which are given as one of inputs to the NOR gate. If Q =1 and $Q'$ =0, the output of NOR gate 1=1 and NOR gate 2=0 i.e., same as the previous values. This state is called “no change” state.

## Case (ii) If R = 1 and S = 0

- The output of AND gate 1 = 1 and AND gate 2 is ‘0’. With thes inputs, the NOR gate 1 output = 0 and NOR gate 2 output is 1. The state is called “reset” state.

## Case (iii) If R = 0 and S = 1

- The output of AND gate 1 = 0 and AND gate 2 is 1. With these inputs the NOR gate 1 output =1 and NOR gate 2 output is 0.
- This state is called “set” state. Case (iv) If R = 1 and S = 1 The output of both AND gates is 1. With these inputs, irrespective of other input, the output of both NOR gates is ‘0’ i.e., Q=0 and $Q'$=0 which is an invalid output. This state is called “indeterminate state”.
- Table 2.3 shows the truth table of RS flip-fop

Table 2.3 Truth table of RS flip-flop

| Inputs | | | Outputs | | state |
|---|---|---|---|---|---|
| CLK | R | S | Q | $\overline{Q}$ | |
| 0 | X | X | NC | NC | No change |
| 1 | 0 | 0 | NC | NC | No change |
| 1 | 0 | 1 | 1 | 1 | Set |
| 1 | 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 1 | X | X | Indeterminate |

## D Flip-Flop:

### Definition of Transparent Latch (D Flip-flop)

- A flip-flop whose output follows it data input when the clock is active. As shown in Fig.2.8, D input goes directly to the S input, and its complement is applied to the R input, through NOT.
- Therefore, only two input conditions exist, either S = 0 and R = 1 or S = 1 and R = 0. When D = 1, S = 1 and R = 0 and when D = 0, S = 0 and R = 1. Therefore, during the occurrence of clock pulse if Synchronous Sequential Circuits 2.9 D = 1, the Q output is set and if D = 0, the output is reset. Table 2.4 shows the truth table of D flip flop

Table 2.4 Truth table of Clocked D flip flop

| CLK | D | $Q_{n+1}$ |
|---|---|---|
| 0 | x | $Q_n$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$Q_{n+1} \Rightarrow$ Next state

$Q_n \Rightarrow$ Present state

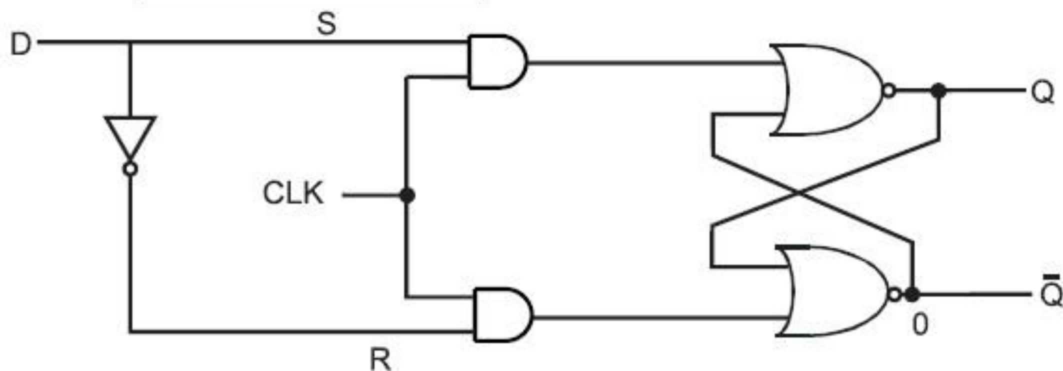



Fig 2.8 Clocked D flip-flop

- The output does not exist when the clock pulse absent. $Q_n$ is the binary state (present state) of the flip flop before the occurrence of a clock pulse.
- It is known as present state. $Q_{n+1}$ is the state of the flip flop after the occurrence of a clock pulse which is known as next state.

## JK Flip Flop:

## Definition:

- JK Flip-flop A JK Flip-flop is a refinement of the RS flip-flop. In JK flip-flop the unpredictable state in the RS flip-flop is defined. Toggle Alternate between opposite binary states with each applied clock pulse.

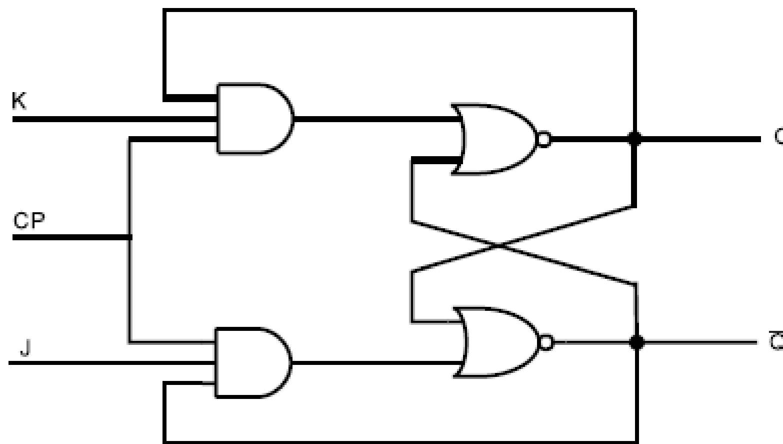



Fig 2.9 Clocked JK flip flop

- A JK flip flop is a refinement of the RS Flip flop. In JK flip flop the unpredictable state in the RS flip flop is defined inputs J and K behave like inputs S and R to set and reset the flip flop respectively as shown in Fig. 2.9.

## Case (i): J = K =0

- When J and K are both low, both AND gates are disabled. Therefore, clock pulses have no effect and, Q and Q' retain their last values. The truth table of JK FF as shown in Table 2.5.

## Case (ii): J = 0, K =1

- When J is low and K is high. The lower AND gate is disabled. So there is no way to set the flip flop. The only possibility is reset. When K is high the upper gate passes a RESET trigger as the next possible clock pulse arrives. This forces Q to become low.
- Therefore J = 0 and K = 1 means that the next positive clock pulse resets the flip flop unless Q is already reset.

## Case (iv): J = 1, K =0

- When J is high and K is low, the upper gate is disabled, so there is no way to reset the flip flop.
- The only possibility is to set the flip flop if it is not previously set with Q = 0, J is high and hence the lower gate passes a SET trigger on the next positive clock pulse.
- This drives Q into the high state. Therefore, J = 1 and K = 0 means that the next positive clock pulse set the flip flop unless Q is already set.

## Case (v): J = K =1

- When J and K are both high (Recall that this is an indeterminate condition with an RS flip flop) it's possible to set or reset the flip flop. If Q is high, the upper gate passes a RESET trigger on the next positive clock edge.
- On the other hand, when Q is low, the lower gate passes a SET trigger on the next positive clock edge. Either way, Q changes to the complement of the last state.
- Therefore, J = K = 1 means output of the flip flop will toggle on the next positive clock edge. When the inputs J and K are short circuited, the JK flip flop is act as a T flip flop. When Input T = 1, it complement the present input. When input T = 0, it maintain the present state.

**Table 2.5 Truth table of Clocked JK flip flop**

| FF inputs | | Present state | Next state |
|---|---|---|---|
| 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 |

## JK Flip-flop

- The truth table and excitation table for JK flip-flop are shown in Table 2.11(a) and (b) respectively. Let us examine of excitation table in each cas

**Table 2.11(a) Truth table**

| J | K | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q}_n$ |

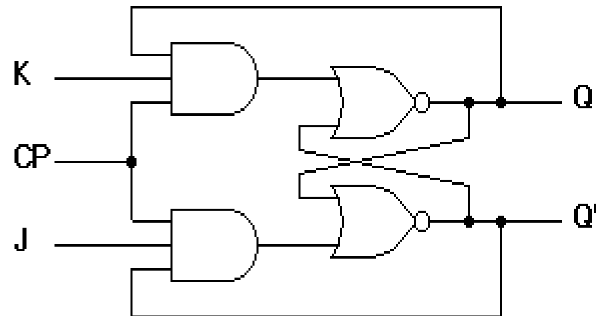
**Table 2.11(b) Excitation table**

| Present State | Next State | FF inputs | |
|---|---|---|---|
| $Q_n$ | $Q_{n+1}$ | J | K |
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | X |
| 1 | 0 | X | 1 |
| 1 | 1 | X | 0 |

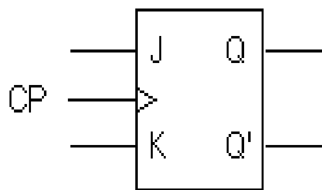
- When both present state (Qn) and next state (Qn+1) are 0, the J input must be 0 and the K input can be either 0 or 1, which is represented by X in the excitation table

| State transition | $0 \rightarrow 0$ | | |
|---|---|---|---|
| Possible input condition | J | K | |
| | 0 | 0 | No change condition |
| | 0 | 1 | Reset condtion |
| Excitation table entry | 0 | X | |

- A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate state of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop (note that in a JK flip-flop, the letter J is for set and the letter K is for clear).
- A clocked JK flip-flop is shown in Figure. Output Q is ANDed with K and CP inputs so that the flip-flop is cleared during a clock pulse only if Q was previously 1. Similarly, ouput Q' is ANDed with J and CP inputs so that the flip-flop is set with a clock pulse only if Q' was previously 1.
- Note that because of the feedback connection in the JK flip-flop, a CP signal which remains a 1 (while J=K=1) after the outputs have been complemented once will cause repeated and continuous transitions of the outputs.
- To avoid this, the clock pulses must have a time duration less than the propagation delay through the flip-flop. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction. The same reasoning also applies to the T flip-flop presented next.




**(a)** Logic diagram




**(b)** Graphical symbol

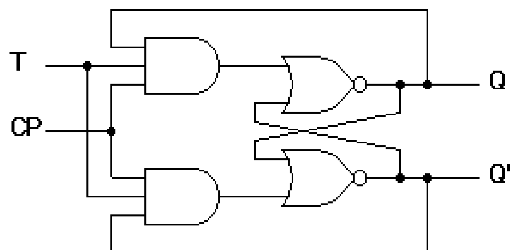
| Q | J | K | Q(t+1) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**(c)** Transition table

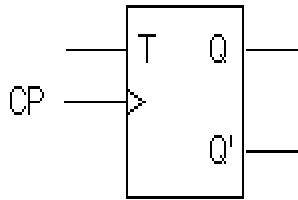
**Figure:**Clocked JK flip-flop

## T Flip-Flop

- The T flip-flop is a single input version of the JK flip-flop. As shown in Figure, the T flip-flop is obtained from the JK type if both inputs are tied together. The output of the T flip-flop "toggles" with each clock pulse.




**(a)** Logic diagram




**(b)** Graphical symbol

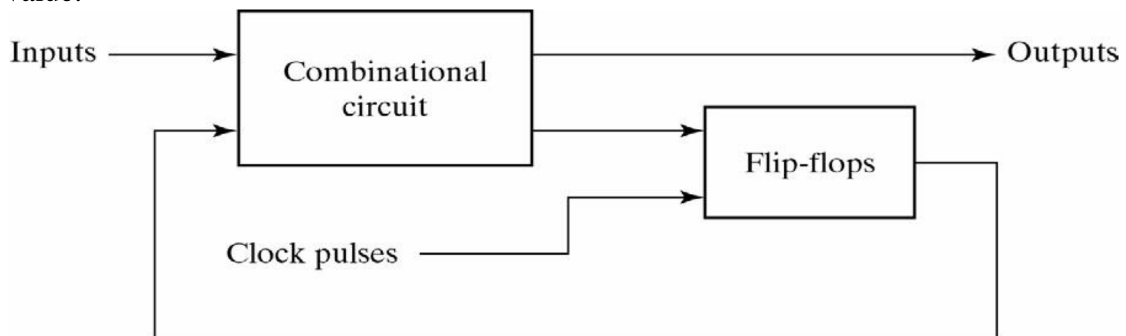
| Q | T | Q(t+1) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**(c)** Transition table

**Figure** : Clocked T flip-flop

## Clocked Sequential Circuits:

- Sequential circuits that use identical clock pulses in the inputs of all the flip-flops are called clocked sequential circuits.
- When a clock pulse is not active, the feedback loop is broken because the flip-flop outputs cannot change even if the combinational logic driving their inputs change in value.




(a) Block diagram

(b) Timing diagram of clock pulses

## Analysis of Clocked Sequential Circuits

- This consists of obtaining a table or a diagram for the time sequence of inputs, outputs and internal states. Boolean expressions can also be written.

## State Equations

- A state equation specifies the next state as a function of the present state and inputs. Consider the sequential circuit given below. Since the D input of a flip-flop determines the value of the next state, the equations for the next state are:

  $A(t+1) = A(t)x(t) + B(t)x(t)$

  $B(t+1) = A'(t)x(t)$

- The left-side of each equation denotes the next state of the flip-flop and the right-side specifies the present state and the conditions that make the next state equal to 1. These can be expressed in a more compact form by omitting the

  $(t)$: $A(t+1) = Ax+Bx$

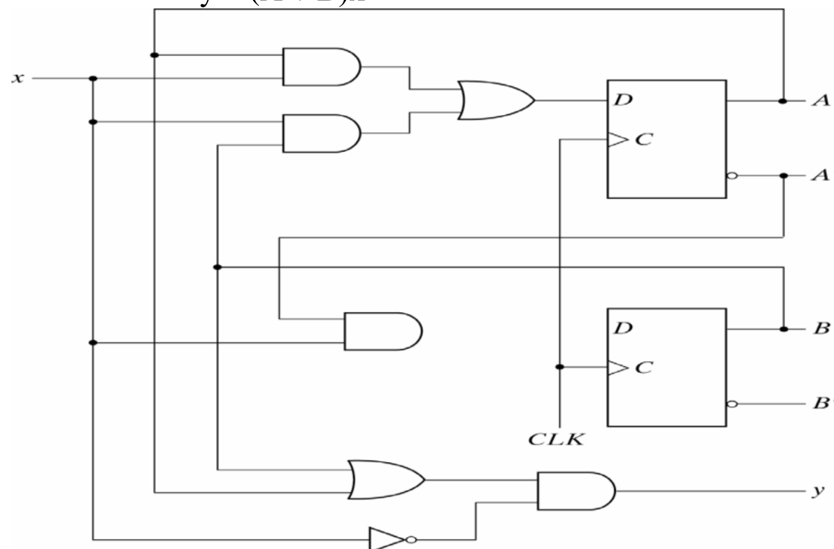
  $B(t+1) = A'x$

- The present state value of the output can be expressed as:

  $y(t) = [A(t) + B(t)]x'(t)$

- The above output equation can be expressed in a more compact form as:

  $y = (A + B)x'$




## State Table

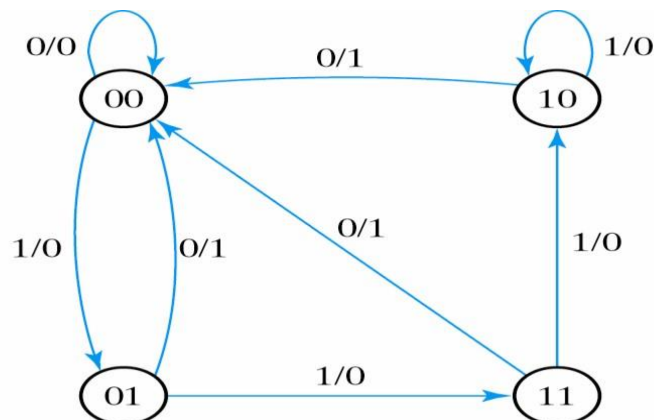
- The time sequence of inputs, outputs, and flip-flop states can be enumerated in a state table.
- This can be generated from the logic diagram or the state equations. Two alternative forms for the sequential circuit shown previously are as follows:

| Present state | | input | Next state | | output |
|---|---|---|---|---|---|
| **A** | **B** | **X** | **A** | **B** | **Y** |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

| PRESENT STATE | NEXT STATE | | OUTPUT | |
|---|---|---|---|---|
| | **X=0** | **X=1** | **X=0** | **X=1** |
| **AB** | **AB** | **AB** | **Y** | **Y** |
| 00 | 00 | 01 | 0 | 0 |
| 01 | 00 | 11 | 1 | 0 |
| 10 | 00 | 10 | 1 | 0 |
| 11 | 00 | 10 | 1 | 0 |

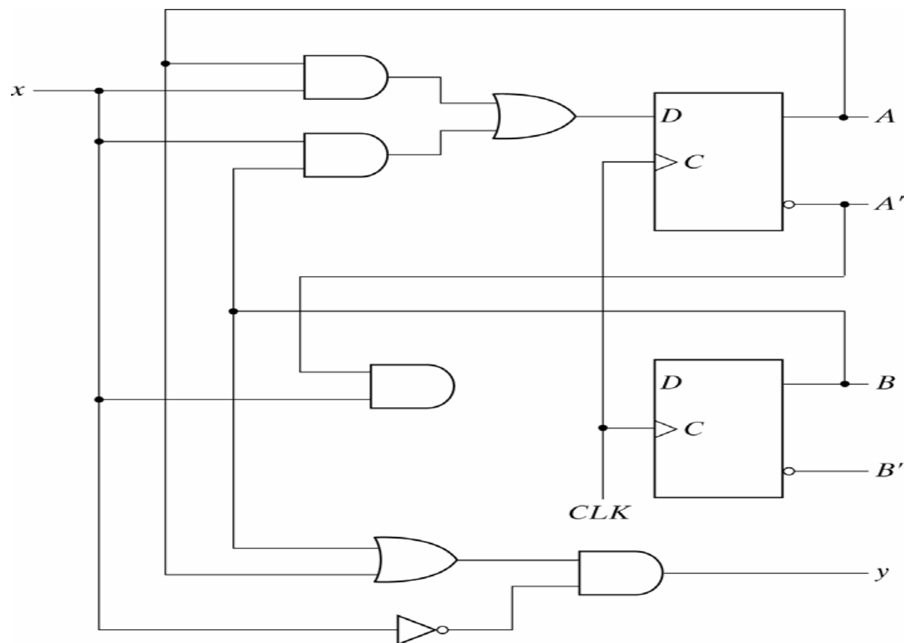
## State Diagram

- The information available in a state table can be represented graphically in a form of a state diagram. In this diagram, a state is represented by a circle, and the transitions between states by directed lines connecting the circles:
- Each directed lines are labelled with two binary numbers separated by a slash. The input value during the present state is labelled first, and the number after the slash gives the output during the present state with the given input. A directed line connecting a circle with itself indicates that no



## Flip-Flop Input Equations

- These fully specify the combinational logic that drives the flip-flops and they imply the type of flip-flop from the letter symbol.
- The input equations for the circuit analyzed before and shown below are:
    - DA=Ax+Bx
    - DB=A'x




- For a D flip-flop, the state equation is the same as the input equation. Input equations are sometimes called excitation equations.

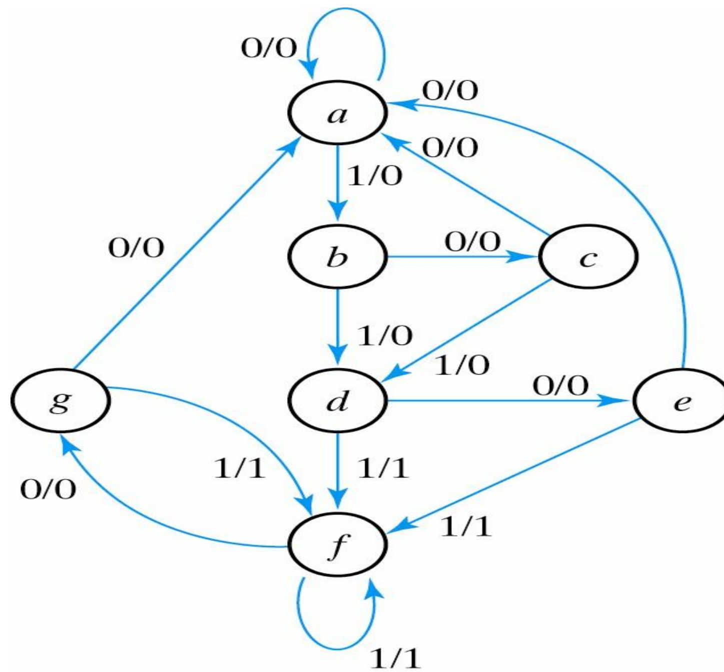
## Design Procedure

The procedure for designing synchronous sequential circuits is summarized by a list of recommended steps:

- 1.From the word description and specifications of the desired operation, derive a state diagram for the circuit.
- 2. Reduce the number of states if necessary.
- 3. Assign binary values to the states.
- 4. Obtain the binary-coded state table.
- 5. Choose the type of flip-flops to be used.
- 6. Derive the simplified flip-flop input equations and output equations.
- 7. Draw the logic diagram.

## State Reduction & Assignment

- Sometimes certain properties of sequential circuits may be used to reduce the number of gates and flip-flops during the design.
- The problem of state reduction is to find ways of reducing the number of states in a sequential circuit, while keeping the external input-output relationships unchanged.
- For example, suppose a sequential circuit is specified by the following seven-state diagram:




- There are an infinite number of input sequences that may be applied; each results in a unique output sequence. Consider the input sequence 01010110100 starting from the initial state a:

| **STATE** | **a** | **a** | **b** | **c** | **d** | **e** | **f** | **f** | **g** | **f** | **g** | **a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **INPUT** | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| **OUTPUT** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

- An algorithm for the state reduction quotes that: “Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.

Now apply this algorithm to the state table of the circuit:

| Present state | Next state | | Output | |
|---|---|---|---|---|
| | X=0 | X=1 | X=0 | X=1 |
| a | a | b | 0 | 0 |
| b | c | d | 0 | 0 |
| c | a | b | 0 | 0 |
| d | e | f | 0 | 1 |
| e | a | f | 0 | 1 |
| f | g | f | 0 | 1 |
| g | a | f | 0 | 1 |

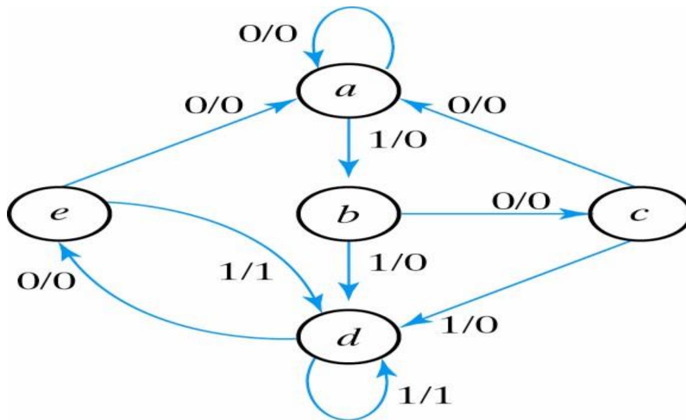
- States g and e both go to states a and f and have outputs of 0 and 1 for x = 0 and x = 1, respectively. The procedure for removing a state and replacing it by its equivalent is demonstrated in the following table :

| Present state | Next state | | Output | |
|---|---|---|---|---|
| | X=0 | X=1 | X=0 | X=1 |
| a | a | b | 0 | 0 |
| b | c | d | 0 | 0 |
| c | a | d | 0 | 0 |
| d | e | f | 0 | 1 |
| e | a | f | 0 | 1 |
| f | e | f | 0 | 1 |

- Thus, the row with present state g is removed and stage g is replaced by state e each time it occurs in the next state columns.
- Present state f now has next states e and f and outputs 0 and 1 for x = 0 and x = 1. The same next states and outputs appear in the row with present state d. Therefore, states f and d are equivalent and can be removed and replaced with d.
- The final reduced state table is:

| Present state | Next state | | Output | |
|---|---|---|---|---|
| | X=0 | X=1 | X=0 | X=1 |
| a | a | b | 0 | 0 |
| b | c | d | 0 | 0 |
| c | a | d | 0 | 0 |
| d | e | d | 0 | 1 |
| e | a | d | 0 | 1 |

The state diagram for the above reduced table is:




- This state diagram satisfies the original input output specifications. Applying the input sequence previously used, the following list is obtained:

| **STATE** | **a** | **a** | **b** | **c** | **d** | **e** | **d** | **d** | **e** | **d** | **e** | **a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **INPUT** | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| **OUTPUT** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

- Note that the same output sequence results, although the state sequence is different.
- To design a sequential circuit with real devices, it is necessary to assign coded binary values to the states. For a circuit with m states, the codes must contain n bits where 2n = ≥ m. For the reduced state table derived previously, only five states need binary assignment; three unused states are treated as don't care conditions.
- Three possible binary state assignments are:

| state | Assignment 1 binary | Assignment 2 Gray code | Assignment 3 One shot |
|---|---|---|---|
| a | 000 | 000 | 00001 |
| b | 001 | 001 | 00010 |
| c | 010 | 011 | 00100 |
| d | 011 | 010 | 01000 |
| e | 100 | 110 | 10000 |

The reduced table with binary assignment 1 is:

| Present state | Next state | | output | |
|---|---|---|---|---|
| | X=0 | X=1 | X=0 | X=1 |
| 000 | 000 | 001 | 0 | 0 |
| 001 | 010 | 011 | 0 | 0 |
| 010 | 000 | 011 | 0 | 0 |
| 011 | 100 | 011 | 0 | 1 |
| 100 | 000 | 011 | 0 | 1 |

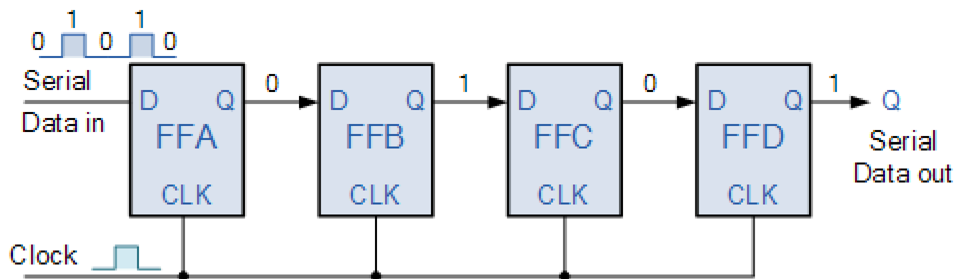
Sometimes, the name transition table is used for a state table with binary assignment

## SHIFT REGISTERS:

- The **Shift Register** is another type of sequential logic circuit that can be used for the storage or the transfer of data in the form of binary numbers. This sequential device loads the data present on its inputs and then moves or "shifts" it to its output once every clock cycle, hence the name "shift register".

## Serial-in, serial-out (SISO):

- These are the simplest kind of shift registers. The data string is presented at 'Data In', and is shifted right one stage each time 'Data Advance' is brought high.
- At each advance, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output. The bit on the far right (i.e. 'Data Out') is shifted out and lost.

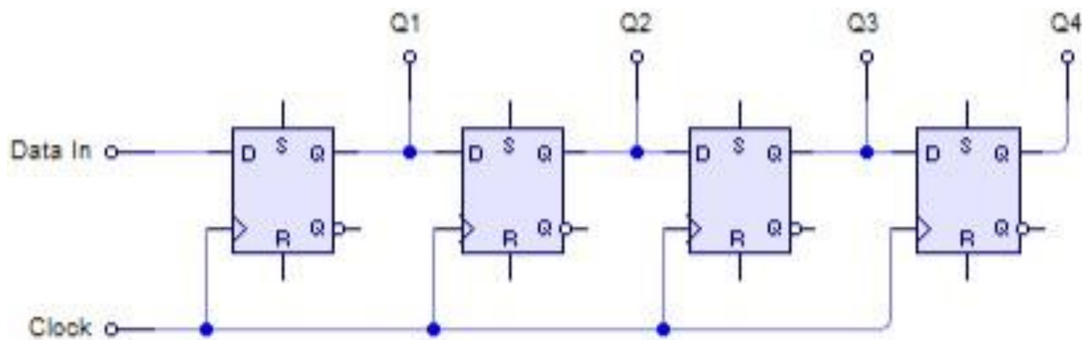


| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

- The data are stored after each flip-flop on the 'Q' output, so there are four storage 'slots' available in this arrangement, hence it is a 4-Bit Register. To give an idea of the shifting pattern, imagine that the register holds 0000 (so all storage slots are empty).
- As 'Data In' presents 1,0,1,1,0,0,0,0 (in that order, with a pulse at 'Data Advance' each time. This is called clocking or strobing) to the register, this is the result. The left hand column corresponds to the left-most flip-flop's output pin, and so on.

## Serial-in, parallel-out (SIPO):

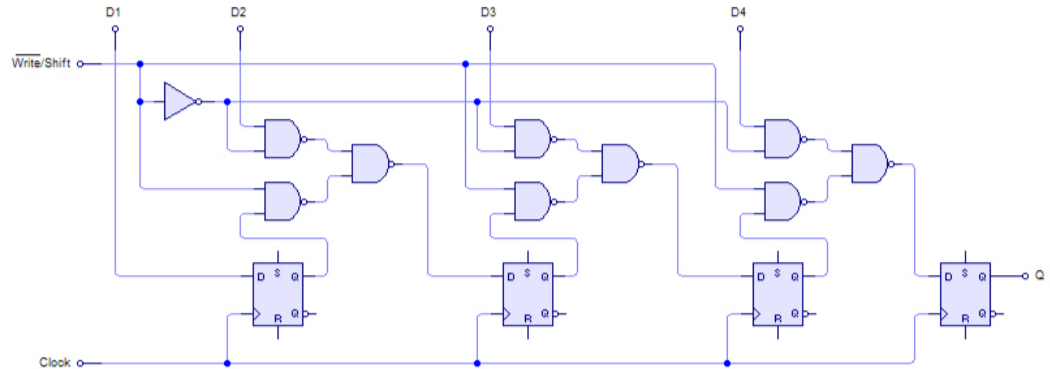
- This configuration allows conversion from serial to parallel format. Data is input serially, as described in the SISO section above.
- Once the data has been input, it may be either read off at each output simultaneously, or it can be shifted out and replaced.




4-Bit SIPO Shift Register

## Parallel-in, serial-out (PISO):

- This configuration has the data input on lines D1 through D4 in parallel format. To write the data to the register, the Write/Shift control line must be held LOW.
- To shift the data, the W/S control line is brought HIGH and the registers are clocked. The arrangement now acts as a PISO shift register, with D1 as the Data Input.
- However, as long as the number of clock cycles is not more than the length of the data-string, the Data Output, Q, will be the parallel data read off in order.

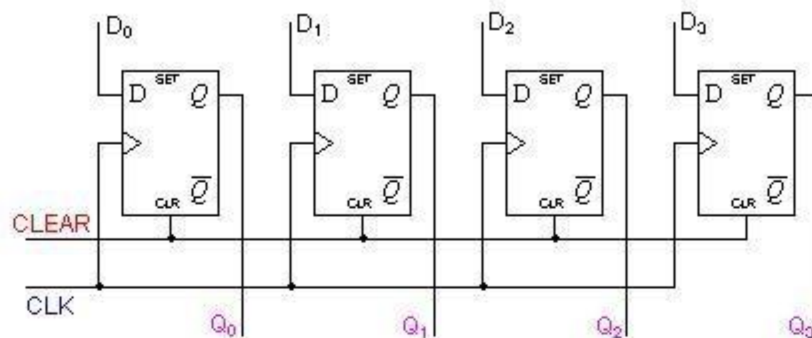



**4-Bit PISO Shift Register**

- The animation below shows the write/shift sequence, including the internal state of the shift register.

## Parallel in parallel out register:

- For parallel in - parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits.
- The following circuit is a four-bit parallel in - parallel out shift register constructed by D flip-flops.



- The D's are the parallel inputs and the Q's are the parallel outputs. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously.

## Uses:

- One of the most common uses of a shift register is to convert between serial and parallel interfaces. This is useful as many circuits work on groups of bits in parallel, but serial interfaces are simpler to construct. Shift registers can be used as simple delay circuits. Several bi-directional shift registers could also be connected in parallel for a hardware implementation of a stack.
- In early computers, shift registers were used to handle data processing: two numbers to be added were stored in two shift registers and clocked out into an arithmetic and logic unit (ALU) with the result being fed back to the input of one of the shift registers (the Accumulator) which was one bit longer since binary addition can only result in an answer that is the same size or one bit longer.
- Many computer languages include instructions to 'shift right' and 'shift left' the data in a register, effectively dividing by two or multiplying by two for each place shifted.
- Very large serial-in serial-out shift registers (thousands of bits in size) were used in a similar manner to the earlier delay line memory in some devices built in the early 1970s.

## THE MASTER-SLAVE JK FLIP-FLOP:

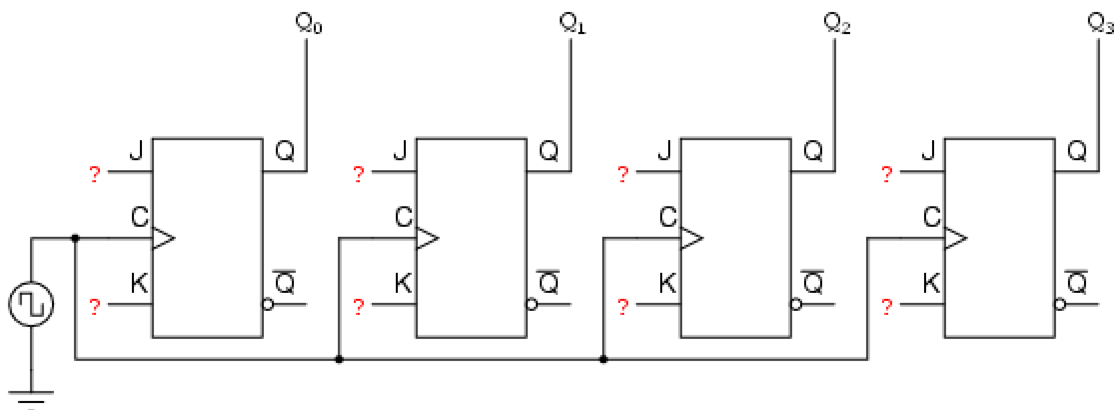
- The **Master-Slave Flip-Flop** is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse.
- The outputs from Q and Q from the "Slave" flip-flop are fed back to the inputs of the "Master" with the outputs of the "Master" flip flop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

## The Master-Slave JK Flip Flop

- The input signals J and K are connected to the gated "master" SR flip flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip flop is the inverse (complement) of the "master" clock input, the "slave" SR flip flop does not toggle. The outputs from the "master" flip flop are only "seen" by the gated "slave" flip flop when the clock input goes "LOW" to logic level "0".
- When the clock is "LOW", the outputs from the "master" flip flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip flop now responds to the state of its inputs passed over by the "master" section.
- Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flip flop are fed through to the gated inputs of the "slave" flip flop and on the "High-to-Low" transition the same inputs are reflected on the output of the "slave" making this type of flip flop edge or pulse-triggered.
- Then, the circuit accepts input data when the clock signal is "HIGH", and passes the data to the output on the falling-edge of the clock signal. In other words, the **Master-Slave JK Flip flop** is a "Synchronous" device as it only passes data with the timing of the clock signal.

## SYNCHRONOUS COUNTERS:

- A synchronous counter, in contrast to an asynchronous counter, is one whose output bits change state simultaneously, with no ripple.
- The only way we can build such a counter circuit from J-K flip-flops is to connect all the clock inputs together, so that each and every flip-flop receives the exact same clock pulse at the exact same time:

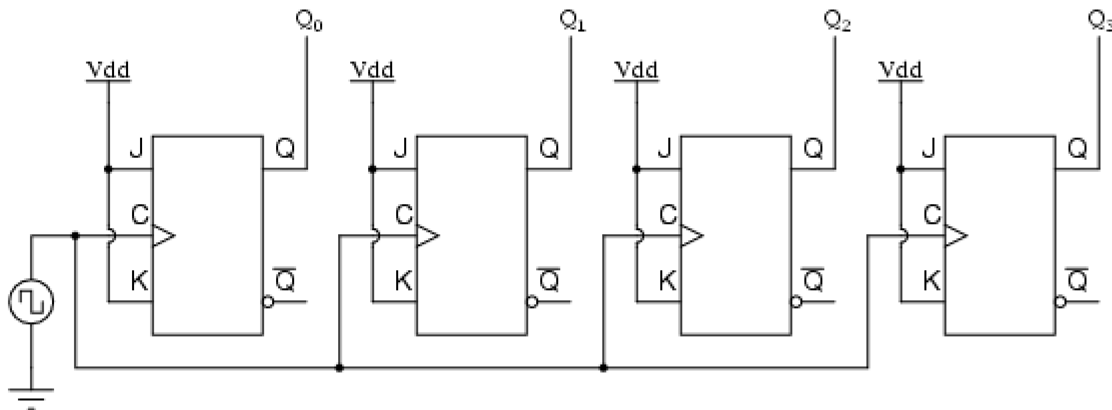



- Now, the question is, what do we do with the J and K inputs? We know that we still have to maintain the same divide-by-two frequency pattern in order to count in a binary

sequence, and that this pattern is best achieved utilizing the "toggle" mode of the flip-flop, so the fact that the J and K inputs must both be (at times) "high" is clear.

- However, if we simply connect all the J and K inputs to the positive rail of the power supply as we did in the asynchronous circuit, this would clearly not work because all the flip-flops would toggle at the same time: with each and every clock pulse!

***This circuit will not function as a counter!***



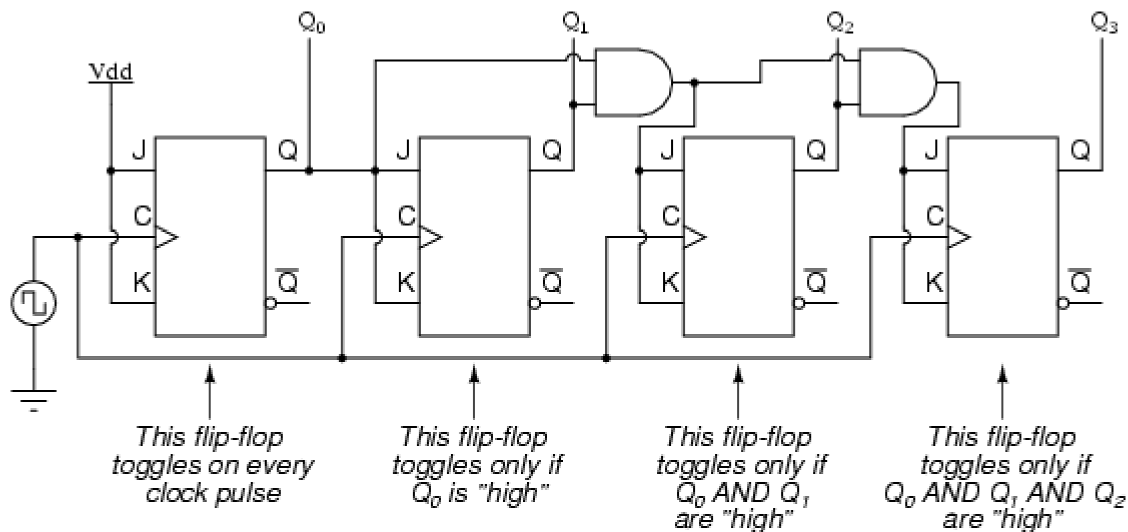

- Let's examine the four-bit binary counting sequence again, and see if there are any other patterns that predict the toggling of a bit. Asynchronous counter circuit design is based on the fact that each bit toggle happens at the same time that the preceding bit toggles from a "high" to a "low" (from 1 to 0). Since we cannot clock the toggling of a bit based on the toggling of a previous bit in a synchronous counter circuit (to do so would create a ripple effect) we must find some other pattern in the counting sequence that can be used to trigger a bit toggle:
- Examining the four-bit binary count sequence, another predictive pattern can be seen. Notice that just before a bit toggles, all preceding bits are "high:"

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 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
```

- This pattern is also something we can exploit in designing a counter circuit. If we enable each J-K flip-flop to toggle based on whether or not all preceding flip-flop outputs (Q)

are "high," we can obtain the same counting sequence as the asynchronous circuit without the ripple effect, since each flip-flop in this circuit will be clocked at exactly the same time:

*A four-bit synchronous "up" counter*

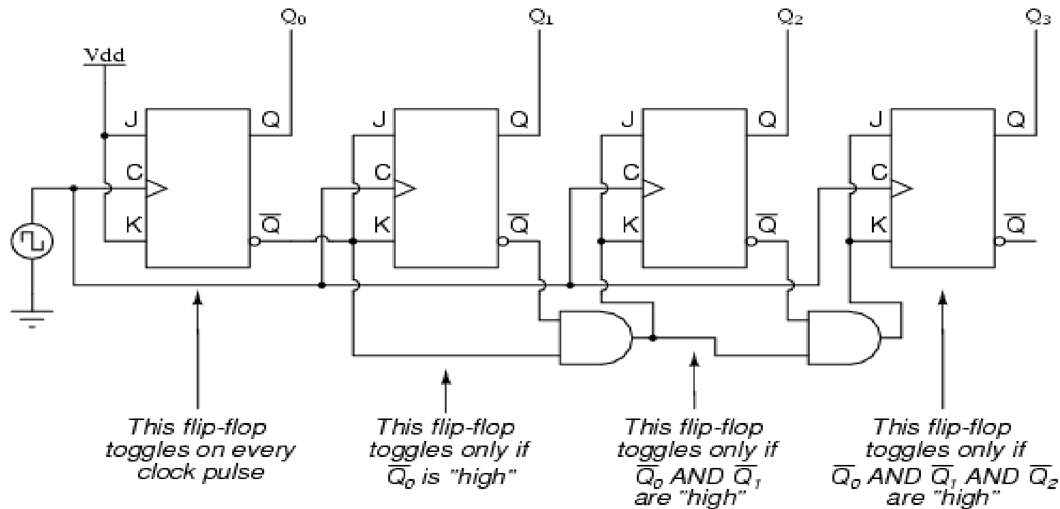



- The result is a four-bit synchronous "up" counter. Each of the higher-order flip-flops are made ready to toggle (both J and K inputs "high") if the Q outputs of all previous flip-flops are "high."
- Otherwise, the J and K inputs for that flip-flop will both be "low," placing it into the "latch" mode where it will maintain its present output state at the next clock pulse.
- Since the first (LSB) flip-flop needs to toggle at every clock pulse, its J and K inputs are connected to $V_{cc}$ or $V_{dd}$, where they will be "high" all the time.
- The next flip-flop need only "recognize" that the first flip-flop's Q output is high to be made ready to toggle, so no AND gate is needed. However, the remaining flip-flops should be made ready to toggle only when *all* lower-order output bits are "high," thus the need for AND gates.

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 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
```

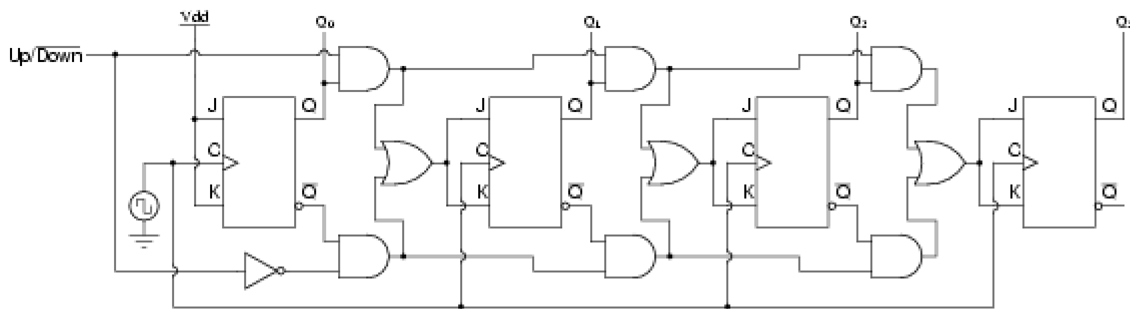
- Since each J-K flip-flop comes equipped with a Q' output as well as a Q output, we can use the Q' outputs to enable the toggle mode on each succeeding flip-flop, being that each Q' will be "high" every time that the respective Q is "low:"

*A four-bit synchronous "down" counter*




- Taking this idea one step further, we can build a counter circuit with selectable between "up" and "down" count modes by having dual lines of AND gates detecting the appropriate bit conditions for an "up" and a "down" counting sequence, respectively, then use OR gates to combine the AND gate outputs to the J and K inputs of each succeeding flip-flop:

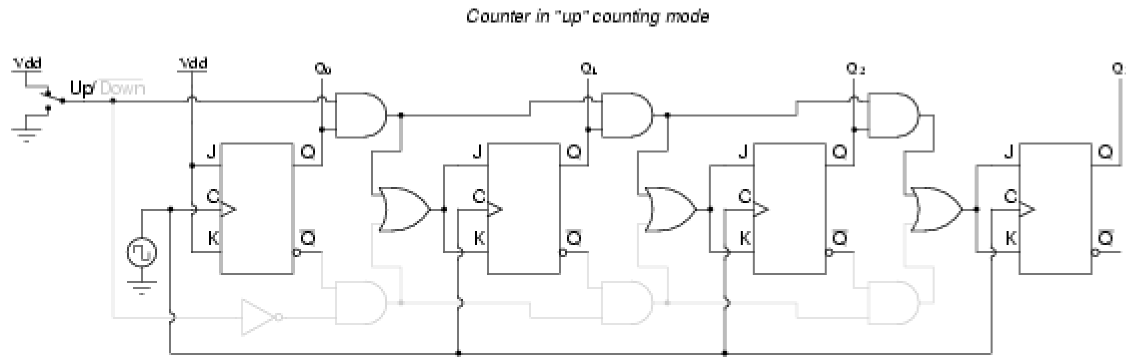
*A four-bit synchronous "up/down" counter*



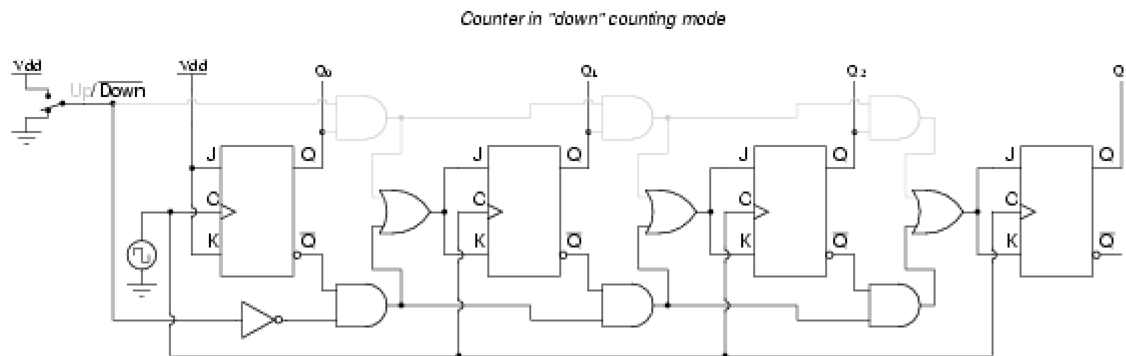

- This circuit isn't as complex as it might first appear. The Up/Down control input line simply enables either the upper string or lower string of AND gates to pass the Q/Q' outputs to the succeeding stages of flip-flops. If the Up/Down control line is "high," the top AND gates become enabled, and the circuit functions exactly the same as the first ("up") synchronous counter circuit shown in this section.

- If the Up/Down control line is made "low," the bottom AND gates become enabled, and the circuit functions identically to the second ("down" counter) circuit shown in this section.

To illustrate, here is a diagram showing the circuit in the "up" counting mode (all disabled circuitry shown in grey rather than black):




Here, shown in the "down" counting mode, with the same grey coloring representing disabled circuitry:




- Up/down counter circuits are very useful devices. A common application is in machine motion control, where devices called rotary shaft encoders convert mechanical rotation into a series of electrical pulses, these pulses "clocking" a counter circuit to track total motion.

## Counters:

In digital logic and computing, a counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. In practice, there are two types of counters:

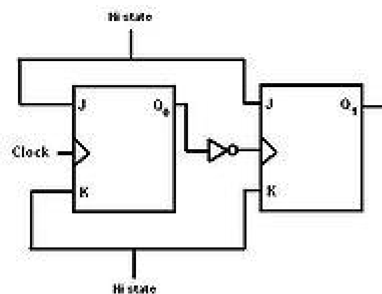
- Up counters, which increase (increment) in value
- Down counters, which decrease (decrement) in value

## In electronics:

In electronics, counters can be implemented quite easily using register-type circuits such as the flip-flop, and a wide variety of designs exist, e.g:

- Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
- Synchronous counter – all state bits change under control of a single clock
- Decade counter – counts through ten states per stage
- Up–down counter – counts both up and down, under command of a control input
- Ring counter – formed by a shift register with feedback connection in a ring
- Johnson counter – a twisted ring counter
- Cascaded counter

## Asynchronous (ripple) counter:

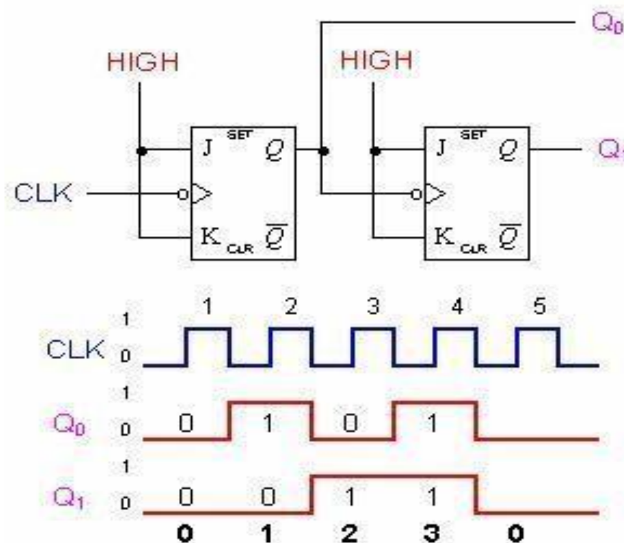



## Asynchronous counter created from two JK flip-flops

- An asynchronous (ripple) counter is a single D-type flip-flop, with its D (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0).
- This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock.
- If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), you will get another 1 bit counter that counts half as fast. Putting them together yields a two bit counter:

| Cycle | Q1 | Q0 | (Q1:Q0)dec |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 2 |
| 3 | 1 | 1 | 3 |
| 4 | 0 | 0 | 0 |

- You can continue to add additional flip-flops, always inverting the output to its own input, and using the output from the previous flip-flop as the clock signal. The result is called a ripple counter, which can count to $2^n$-1 where n is the number of bits (flip-flop stages) in the counter.
- Ripple counters suffer from unstable outputs as the overflows "ripple" from stage to stage, but they do find frequent application as dividers for clock signals, where the instantaneous count is unimportant, but the division ratio overall is. (To clarify this, a 1-bit counter is exactly equivalent to a divide by two circuit; the output frequency is exactly half that of the input when fed with a regular train of clock pulses).
- The use of flip-flop outputs as clocks leads to timing skew between the count data bits, making this ripple technique incompatible with normal synchronous circuit design styles.



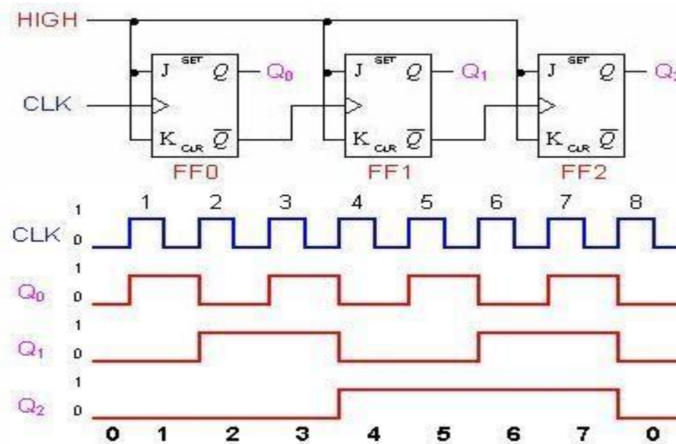

- A two-bit asynchronous counter is shown on the left. The external clock is connected to the clock input of the first flip-flop (FF0) only. So, FF0 changes state at the falling edge of each clock pulse, but FF1 changes only when triggered by the falling edge of the Q

output of FF0.

- Because of the inherent propagation delay through a flip-flop, the transition of the input clock pulse and a transition of the Q output of FF0 can never occur at exactly the same time. Therefore, the flip-flops cannot be triggered simultaneously, producing an asynchronous operation.

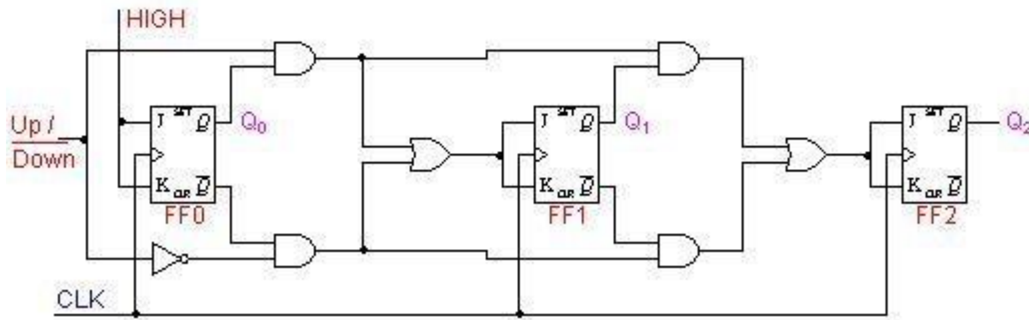
- A mod-n counter may also described as a divide-by-*n* counter. This is because the most significant flip-flop (the furthest flip-flop from the original clock pulse) produces one pulse for every *n* pulses at the clock input of the least significant flip-flop (the one triggers by the clock pulse). Thus, the above counter is an example of a divide-by-4 counter.

- The following is a three-bit asynchronous binary counter 0 and its timing diagram for one cycle. It works exactly the same way as a two-bit asynchronous binary counter mentioned above, except it has eight states due to the third flip-flop.



## Up Down counters:

- A circuit of a 3-bit synchronous up-down counter and a table of its sequence are shown below. Similar to an asynchronous up-down counter, a synchronous up-down counter also has an up-down control input. It is used to control the direction of the counter through a certain sequence.




An examination of the sequence table shows:

Up ↓ / Down ↑

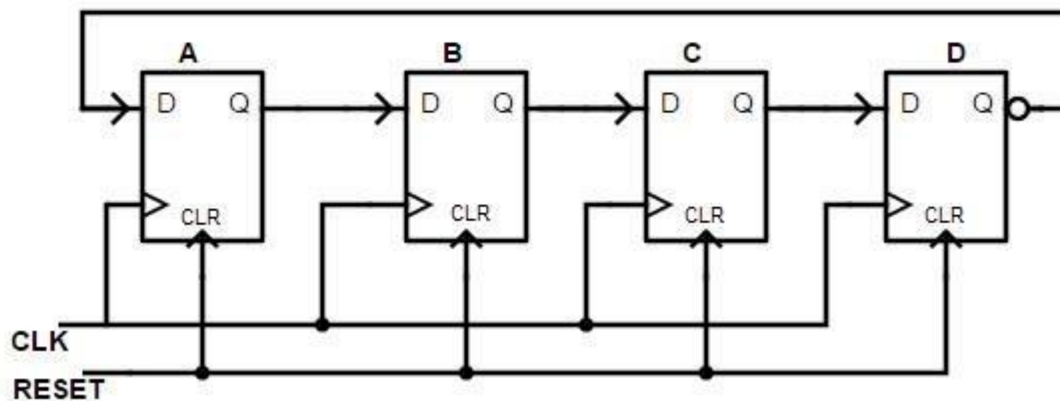
| Q2 | Q1 | Q0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

- for both the UP and DOWN sequences, Q0 toggles on each clock pulse.
- for the UP sequence, Q1 changes state on the next clock pulse when Q0=1.
- for the DOWN sequence, Q1 changes state on the next clock pulse when Q0=0.
- for the UP sequence, Q2 changes state on the next clock pulse when Q0=Q1=1.
- for the DOWN sequence, Q2 changes state on the next clock pulse when Q0=Q1=0.

- These characteristics are implemented with the AND, OR & NOT logic connected as shown in the logic diagram above.

## Ring counters:

- Ring counter is a sequential logic circuit that is constructed using shift register. Same data recirculates in the counter depending on the clock pulse.
- The ring counter is a cascaded connection of flip flops, in which the output of last flip flop is connected to input of first flip flop. In ring counter if the output of any stage is 1, then its reminder is 0. The Ring counters transfers the same output throughout the circuit.
- That means if the output of the first flip flop is 1, then this is transferred to its next stage i.e. 2nd flip flop. By transferring the output to its next stage, the output of first flip flop becomes 0. And this process continues for all the stages of a ring counter. If we use n flip flops in the ring counter, the '1' is circulated for every n clock cycles.




- Here we design the ring counter by using D flip flop. This is a Mod 4 ring counter which has 4 D flip flops connected in series. The clock signal is applied to clock input of each flip flop, simultaneously and the RESET pulse is applied to the CLR inputs of all the flip flops.

## Operation of Ring Counter

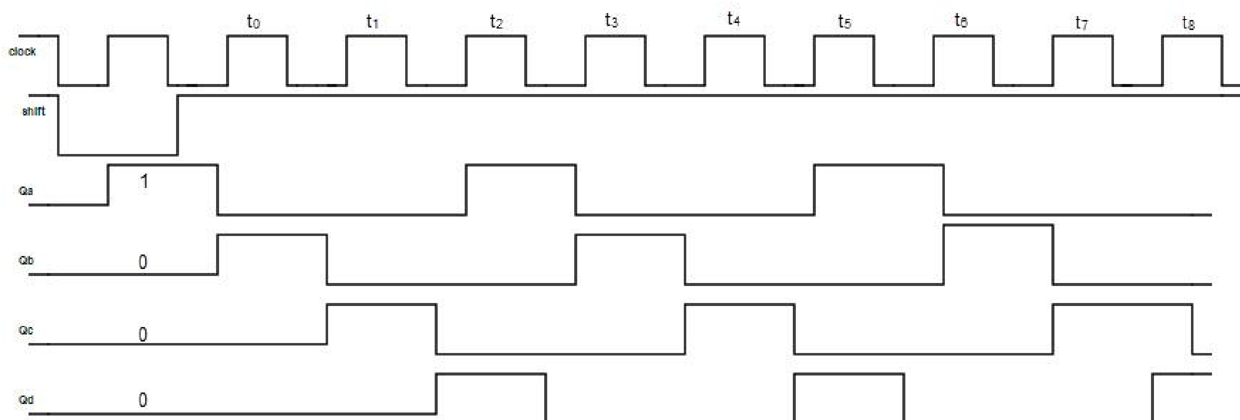
- Initially, all the flip flops in ring counter are reset to 0 by applying CLEAR signal. Before applying the clock pulse, we apply the PRESET pulse to the flip flops which assigns the value '1' to the ring counter circuit. For each clock signal, the data circulates among all the 4 flip flop stages of ring counter.
- This 4 staged ring counter is called Mod 4 ring counter or 4 bit ring counter. To circulate the data correctly in the ring counter, we must load the counter with required values like all 0's or all 1's.

## Truth table of ring counter

| $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

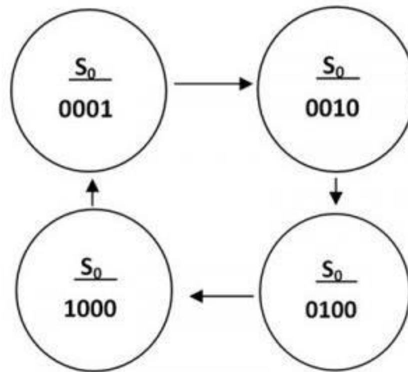
- When CLEAR input CLR = 0, then all flip flops are set to 1. When CLEAR input CLR = 1, the ring counter starts its operation.
- For one clock signal, the counter starts its operation. On next clock signal, the counter again resets to 0000. Ring counter has 4 sequences: 0001, 0010, 0100, 1000, 000.

## Timing diagram of Ring Counter




- The timing diagram of the Ring counter will explain that the clock signal changes the output of every stage of the counter, so that CLK signal will help the data to circulate from one flip flop to another.
- As the 4 bit ring counter (4 stages or 4 flip flops) circulates the preset digit within one clock signal, the output frequency of each flip flop is ¼ th of the main clock frequency.

## State diagram of ring counter




- The state diagram of the 4 bit ring counter is shown in above picture. It denotes that the position of the preset digit (in this case preset digit is 1) is changing its position from LSB to MSB, for one clock signal.

## Advantages

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

## JOHNSON RING COUNTER:

- The **Johnson Ring Counter** or "Twisted Ring Counters", is another shift register with feedback exactly the same as the standard Ring Counter above, except that this time the inverted output Qof the last flip-flop is now connected back to the input D of the first flip-flop as shown below.
- The main advantage of this type of ring counter is that it only needs half the number of flip-flops compared to the standard ring counter then its modulo number is halved. So a "n-stage" Johnson counter will circulate a single data bit giving sequence of 2n different states and can therefore be considered as a "mod-2n counter".

**4-bit Johnson Ring Counter**

- This inversion of Q before it is fed back to input D causes the counter to "count" in a different way. Instead of counting through a fixed set of patterns like the normal ring counter such as for a 4-bit counter, "0001"(1), "0010"(2), "0100"(4), "1000"(8) and repeat, the Johnson counter counts up and then down as the initial logic "1" passes through it to the right replacing the preceding logic "0".
- A 4-bit Johnson ring counter passes blocks of four logic "0" and then four logic "1" thereby producing an 8-bit pattern. As the inverted output Q is connected to the input D this 8-bit pattern continually repeats. For example, "1000", "1100", "1110", "1111", "0111", "0011", "0001", "0000" and this is demonstrated in the following table below.

**Truth Table for a 4-bit Johnson Ring Counter**

| Clock Pulse No | FFA | FFB | FFC | FFD |
|---|---|---|---|---|
| 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 |

- As well as counting or rotating data around a continuous loop, ring counters can also be used to detect or recognise various patterns or number values within a set of data.
- By connecting simple logic gates such as the AND or the OR gates to the outputs of the flip-flops the circuit can be made to detect a set number or value.
- Standard 2, 3 or 4-stage **Johnson Ring Counters** can also be used to divide the frequency of the clock signal by varying their feedback connections and divide-by-3 or divide-by-5 outputs are also available

## CPLDs and FPGAs

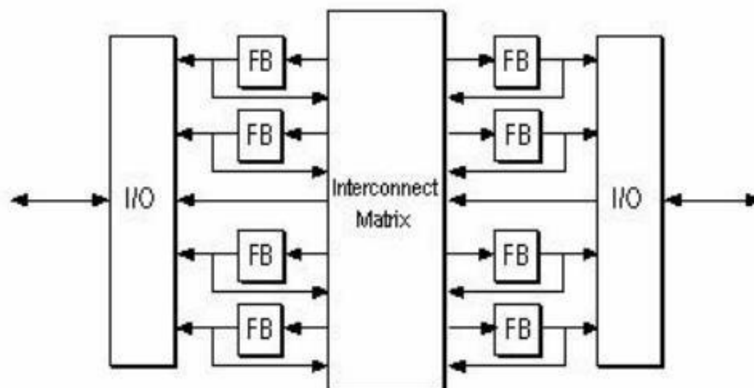
- Ideally, though, the hardware designer wanted something that gave him or her the flexibility and complexity of an ASIC but with the shorter turn-around time of a programmable device.
- The solution came in the form of two new devices - the Complex Programmable Logic Device (CPLD) and the Field Programmable Gate Array. As can be seen in Figure 4, CPLDs and FPGAs bridge the gap between PALs and Gate Arrays. CPLDs are as fast as PALs but more complex. FPGAs approach the complexity of Gate Arrays but are still

## Complex Programmable Logic Devices (CPLDs)

- Complex Programmable Logic Devices (CPLDs) are exactly what they claim to be. Essentially they are designed to appear just like a large number of PALs in a single chip, connected to each other through a crosspoint switch They use the same development tools and programmers, and are based on the same technologies, but they can handle much more complex logic and more of it.

## CPLD Architectures

- The diagram in Figure 5 shows the internal architecture of a typical CPLD. While each manufacturer has a different variation, in general they are all similar in that they consist of function blocks, input/output block, and an interconnect matrix.
- The devices are programmed using programmable elements that, depending on the technology of the manufacturer, can be EPROM cells, EEPROM cells, or Flash EPROM cells.

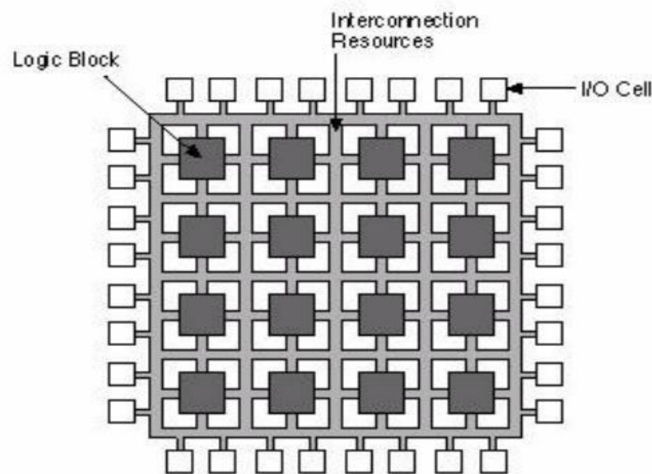



- Function Blocks A typical function block is shown in Figure 6. The AND plane still exists as shown by the crossing wires. The AND plane can accept inputs from the I/O blocks, other function blocks, or feedback from the same function block. The terms and then ORed together using a fixed number of OR gates, and terms are selected via a large multiplexer.
- The outputs of the mux can then be sent straight out of the block, or through a clocked flip-flop. This particular block includes additional logic such as a selectable exclusive OR and a master reset signal, in addition to being able to program the polarity at different stages.
- Usually, the function blocks are designed to be similar to existing PAL architectures, such as the 22V10, so that the designer can use familiar tools or even older designs without changing them.

## FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

- Field Programmable Gate Arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC.
- This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used. For example, an FPGA maybe used in a design that need to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost.

## FPGA Architectures




- Each FPGA vendor has its own FPGA architecture, but in general terms they are all a variation of that shown in Figure. The architecture consists of configurable logic blocks, configurable I/O blocks, and programmable interconnect.
- Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of programmable elements for an FPGA are Static RAM and anti-fuses.

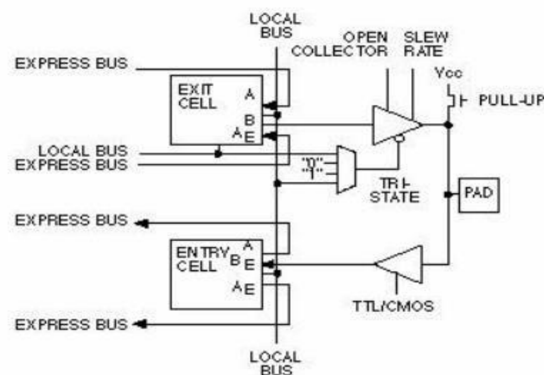
## Configurable Logic Blocks

- Configurable Logic Blocks contain the logic for the FPGA. In a large grain architecture, these CLBs will contain enough logic to create a small state machine. In a fine grain architecture, more like a true gate array ASIC, the CLB will contain only very basic logic.
- The diagram in Figure would be considered a large grain block. It contains RAM for creating arbitrary combinatorial logic functions. It also contains flip-flops for clocked storage elements, and multiplexers in order to route the logic within the block and to and

fromexternal resources. The muxes also allow polarity selection and reset and clear input selection.

## Configurable I/O Blocks

- A Configurable I/O Block, shown in Figure 10, is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three state and open collector output controls.
- Typically there are pull up resistors on the outputs and sometimes pull down resistors. The polarity of the output can usually be programmed for active high or active low output and often the slew rate of the output can be programmed for fast or slow rise and fall times. In addition, there is often a flip-flop on outputs so that clocked signals can be output directly to the pins without encountering significant delay.
- It is done for inputs so that there is not much delay on a signal before reaching a flip-flop which would increase the device hold time requirement.




## Example FPGA Families

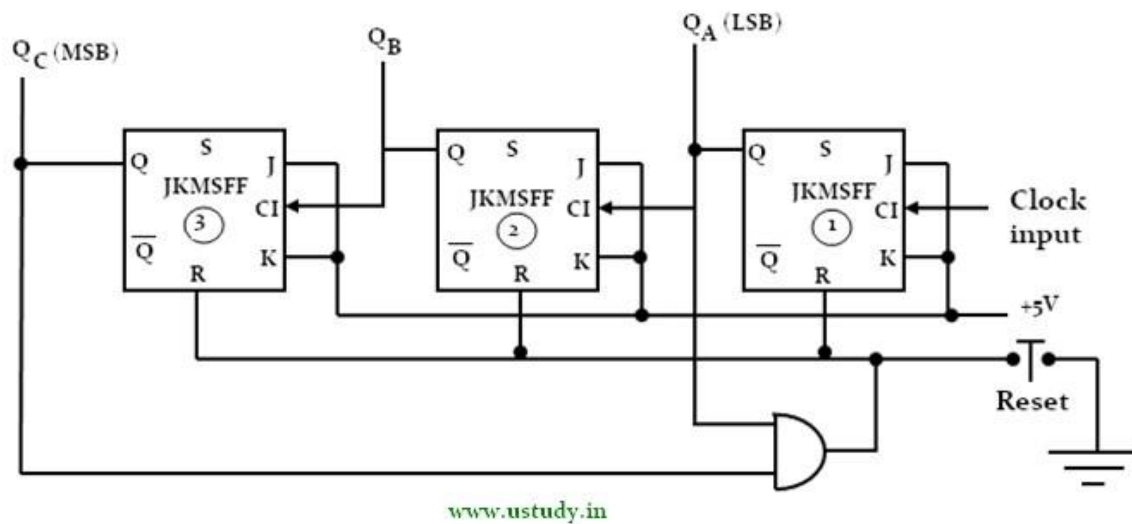
Examples of SRAM based FPGA families include the following:

- Altera FLEX family.
- Atmel AT6000 and AT40K families.
- Lucent Technologies ORCA family.
- Xilinx XC4000 and Virtex families.

## Modulus counter:

- A counter which is reset at the fifth clock pulse is called Mod 5 counter or Divide by 5 counter. The circuit diagram of Mod 5 counter is shown in the figure. This counter contains three JKMS flip-flop.

## Logic Diagram:




## Truth table:

| Clock | $Q_C$ | $Q_B$ | $Q_A$ |
|---|---|---|---|
| Reset | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 |

- A 3 bit binary counter is normally counting from 000 to 111. The actual output of a 3 bitbinary counter at the fifth clock pulse is 101.
- A two input NAND gate is used to make a Mod 5 counter.
- The outputs of the first and third flip flops ($Q_A$ and $Q_C$) are connected to the input of thegive NAND gate, and its output is connected to the RESET terminal of the counter,
- Hence the counter is reset at the fifth clock pulse, which produces the output $Q_C$,$Q_B$,$Q_A$ as 000. It is called divide by $5^{th}$ counter or **mod 5 counter.**

UNIT – V

# Memory

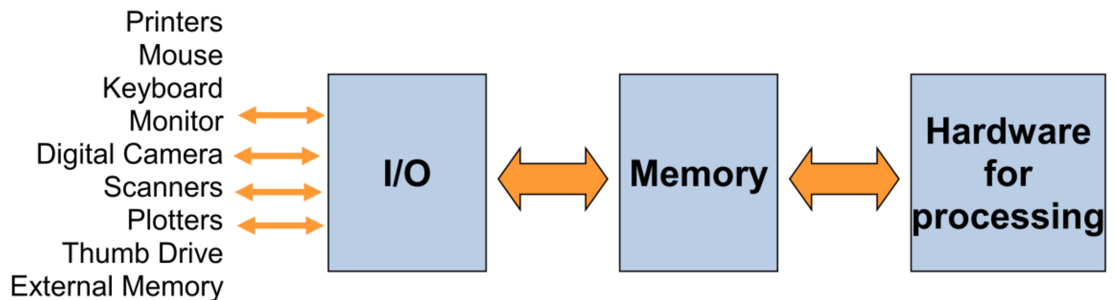
### Classification of Memories

### Memory Definitions

- Collection of cells capable of storing binary information
- Contains electronic circuits for storing & retrieve information
- Used to provide temporary or permanent storage capability

### Memory Basic Process

- Info/content from memory is send to h/w (usually consist of registers & combinational logic) to be processed
- The processed info is then returned to the same or different memory address
- Input and Output devices may also interact with memory



**Types of Memories**

- **Random Access Memory (RAM)**
  - Write operation – stores new info
  - Read operation – transfer the stored info out of memory
- **Read Only Memory (ROM)**

## Memory data elements

- Typical data elements are:
  - bit : a single binary digit
  - byte : a collection of eight (8) bits accessed together
  - word : a collection of binary bits whose size is a typical unit of access for the memory. (e.g., 1 byte, 2 bytes, 4 bytes, 8 bytes, etc.)
- Memory Data ─ a bit or a collection of bits to be stored into or accessed from memory cells.
- Memory Operations ─ operations on memory data supported by the memory unit. Typically, read and write operations over some data element (bit, byte, word, etc.).

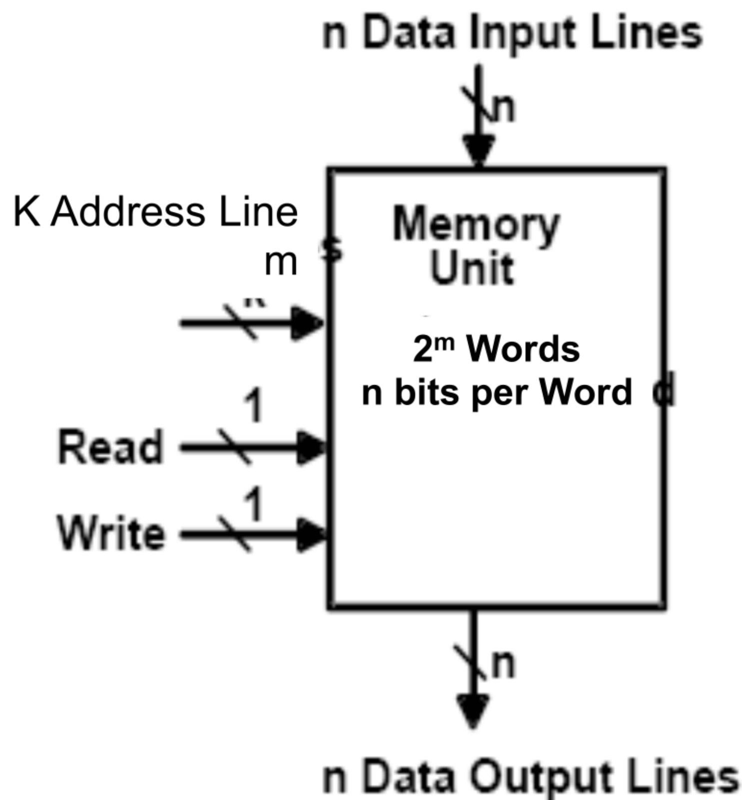
**Memory Organization**

- Organized as an indexed array of words. Value of the index for each word is the memory address.
- Often organized to fit the needs of a particular computer architecture. Some historically significant computer architectures and their associated memory organization.

**Digital Equipment Corporation PDP-8 (DEC Alpha)**

- used a 12-bit address to address 4096 12-bit words.
- **IBM 360**
- used a 24-bit address to address 16,777,216 8-bit (bytes), or 4,194,304 - 32-bit words.
- Intel 8080 (8-bit predecessor to the 8086 and the current Intel processors)
- used a 16-bit address to address 65,536 8-bit (bytes).

**Memory Block Diagram:**




**A basic memory system is shown here:**

- k address lines are decoded to address $2^m$ words of memory.
- Each word is n bits.

- Read and Write are single control lines defining the simplest of memory operations.

**Memory Organization**

| Memory Address Binary | Decimal | Memory Content |
|---|---|---|
| 0 0 0 | 0 | 1 0 0 0 1 1 1 1 |
| 0 0 1 | 1 | 1 1 1 1 1 1 1 1 |
| 0 1 0 | 2 | 1 0 1 1 0 0 0 1 |
| 0 1 1 | 3 | 0 0 0 0 0 0 0 0 |
| 1 0 0 | 4 | 1 0 1 1 1 0 0 1 |
| 1 0 1 | 5 | 1 0 0 0 0 1 1 0 |
| 1 1 0 | 6 | 0 0 1 1 0 0 1 1 |
| 1 1 1 | 7 | 1 1 0 0 1 1 0 0 |

**Example of memory contents above:**

- address bits = 3; m = 3
- data bits = 8; n = 8
- Therefore number of address lines = k = ($2^m$); $2^3 = 8$
- Address range = 0 to $2^m$ -1; therefore 0 to $2^3 - 1$, Address range = 0 to 7
- 1 word is the size of the memory content; so the memory above has 8 words of 8-bit data

| Memory address Binary | Decimal | Memory contents |
|---|---|---|
| 0000000000 | 0 | 10110101 01011100 |
| 0000000001 | 1 | 10101011 10001001 |
| 0000000010 | 2 | 00001101 01000110 |
| | . | . |
| | . | . |
| | . | . |
| | . | . |
| | . | . |
| 1111111101 | 1021 | 10011101 00010101 |
| 1111111110 | 1022 | 00001101 00011110 |
| 1111111111 | 1023 | 11011110 00100100 |

**Example:**

- address bits = m = 10
- data bits =16; n = 16
- Address line = ($2^m$)
- $2^{10}$ = 1024 or 1K, labeled 0 to 1023
- memory content = 16-bit
- so the memory has 1K words of 16-bit data or 1K x 16-bit memory

**Memory operations require the following:**

- Data
- Address
- An operation ─ Typical operations are READ and WRITE. (RAM)

**Read Memory ─ an operation that reads a data value stored inmemory: (takes from memory)**

- Place a valid address on the address lines
- Activate the Read input.
- Note : the content of the selected word are not changed by reading them

**Write Memory ─ an operation that writes a data value to memory:**

- Place a valid address on the address lines
- Apply data on the data lines
- Activate the Write input

**Other than Read/Write (R/W) Chip Select is used to enable a particularRAM. It is sometimes called Memory Enable.**

## Memory Enable

Control Inputs to a Memory Chip

| Chip select CS | Read/$\overline{Write}$ R/$\overline{W}$ | Memory operation |
|---|---|---|
| 0 | | None |
| 1 | 0 | Write to selected word |
| 1 | 1 | Read from selected word |

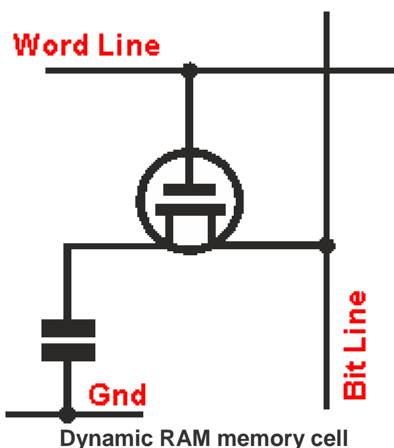
**RAM INTEGRATED CIRCUIT:**

## Types of random access memory (RAM)

- Static – information stored in latches
- Dynamic – information stored as electrical charges on capacitors
- Charge “leaks” off
- Periodic refresh of charge required
- Dependence on Power Supply
- Volatile – loses stored information when power turned off
  (example : FPGA – Flex10K)
- Non-volatile – retains information when power turned off
  (example : CPLD – MAX)

### DRAM OPERATION BASICS

- DRAM memory technology has MOS technology at the heart of the design, fabrication and operation. The basic dynamic RAM or DRAM memory cell uses a capacitor to store each bit of data and a transfer device - a MOSFET - that acts as a switch.
- The level of charge on the memory cell capacitor determines whether that particular bit is a logical "1" or "0" - the presence of charge in the capacitor indicates a logic "1" and the absence of charge indicates a logical "0".
- The basic dynamic RAM memory cell has the format that is shown below. It is very simple and as a result it can be densely packed on a silicon chip and this makes it very cheap.



Dynamic RAM memory cell

- Two lines are connected to each dynamic RAM cell - the Word Line (W/L) and the Bit Line (B/L) connect as shown so that the required cell within a matrix can have data read or written to it.

- The basic memory cell shown would be one of many thousands or millions of such cells in a complete memory chip. Memories may have capacities of 256 Mbit and more.
- To improve the write or read capabilities and speed, the overall dynamic RAM memory may be split into sub-arrays. The presence of multiple sub-arrays shortens the word and bit lines and this reduces the time to access the individual cells. For example a 256 Mbit dynamic RAM, DRAM may be split into 16 smaller 16Mbit arrays.
- The word lines control the gates of the transfer lines, while the bit bines are connected to the FET channel and are ultimately connected to the sense amplifiers.

There are two ways in which the bit lines can be organised:

- **Folded Bit Lines:** It is possible to consider a pair of adjacent bit lines as a single bit line folded in half with the connection on the fold broken and connected to a shared sense amplifier. This format provides additional noise immunity, but at the expense of being less compact.
- **Open Bit Lines:** In this configuration the sense lines are placed between two sub-arrays, thereby connecting each sense amplifier to one bit line in each array. This offers a more compact solution than the folded bit lines, but at the expense of noise immunity.

**DYNAMIC RAM READ / WRITE:**

One of the critical issues within the dynamic RAM is to ensure that the read and write functions are carried out effectively. As voltages on the charge capacitors are small, noise immunity is a key issue.

There are several lines that are used in the read and write operations:

- **RAS, the Row Address Strobe:** As the name implies, the /RAS line strobes the row to be addressed. The address inputs are captured on the falling edge of the /RAS line. The row is held open as long as /RAS remains low.
- **CAS, the Column Address Strobe:** This line selects the column to be addressed. The address inputs are captured on the falling edge of /CAS. It enables a column to be selected from the open row for read or write operations.
- **WE, Write Enable:** This signal determines whether a given falling edge of /CAS is a read or write. Low enables the write action, while high enables a

read action. If low (write), the data inputs are also captured on the falling edge of /CAS.

- **OE, Output Enable:** The /OE signal is typically used when controlling multiple memory chips in parallel. It controls the output to the data I/O pins. The data pins are driven by the DRAM chip if /RAS and /CAS are low, /WE is high, and /OE is low. In many applications, /OE can be permanently connected low, i.e. output always enabled if not required for example of chips are not wired in parallel.

**DYNAMIC RAM REFRESH:**

- One of the problems with this arrangement is that the capacitors do not hold their charge indefinitely as there is some leakage across the capacitor. It would not be acceptable for the memory to lose its data, and to overcome this problem the data is refreshed periodically. The data is sensed and written and this then ensures that any leakage is overcome, and the data is re-instated.
- One of the key elements of DRAM memory is the fact that the data is refreshed periodically to overcome the fact that charge on the storage capacitor leaks away and the data would disappear after a short while. Typically manufacturers specify that each row should be refreshed every 64 ms. This time interval falls in line with the JEDEC standards for dynamic RAM refresh periods.
- There are a number of ways in which the refresh activity can be accomplished. Some processor systems refresh every row together once every 64 ms. Other systems refresh one row at a time, but this has the disadvantage that for large memories the refresh rate becomes very fast. Some other systems (especially real time systems where speed is of the essence) adopt an approach whereby a portion of the semiconductor memory at a time based on an external timer that governs the operation of the rest of the system. In this way it does not interfere with the operation of the system.
- Whatever method is use, there is a necessity for a counter to be able to track the next row in the DRAM memory is to be refreshed. Some DRAM chips include a counter, otherwise it is necessary to include an additional counter for this purpose.
- It may appear that the refresh circuitry required for DRAM memory would over complicate the overall memory circuit making it more expensive. However it is found that DRAM the additional circuitry is not a major concern if it can be integrated into the memory chip itself. It is also found

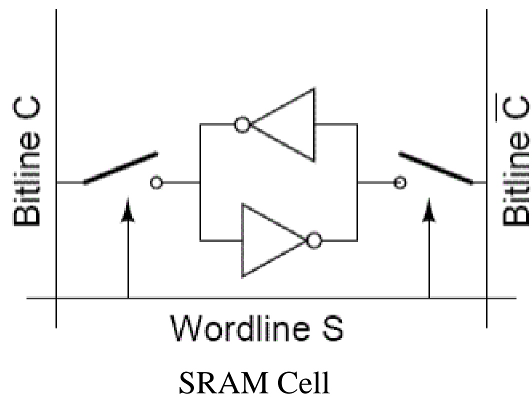
that DRAM memory is much cheaper and has a much greater capacity thanthe other major contender which might be Static RAM (SRAM).

**DRAM ADVANTAGES AND DISADVANTAGES:**

| ADVANTAGES | DISADVANTAGES |
| --- | --- |
| • Very dense<br>• Low cost per bit<br>• Simple memory cell structure | • Complex manufacturing process<br>• Data requires refreshing<br>• More complex external circuitry required (read and refresh periodically)<br>• Volatile memory<br>• Relatively slow operational speed |

## SRAM BASICS

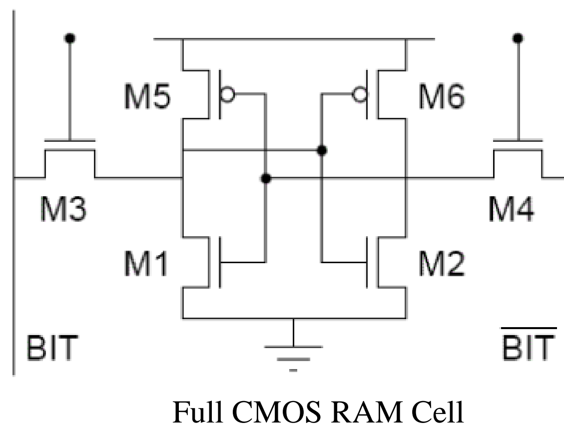
- The memory circuit is said to be static if the stored data can be retained indefinitely, as long as the power supply is on, without any need for periodic refresh operation.
- The data storage cell, i.e., the one-bit memory cell in the static RAM arrays, invariably consists of a simple latch circuit with two stable operating points. Depending on the preserved state of the two inverter latch circuit, the data being held in the memory cell will be interpreted either as logic '0' or as logic '1'.
- To access the data contained in the memory cell via a bit line, we need atleast one switch, which is controlled by the corresponding word line as shown in Figure



SRAM Cell

**CMOS SRAM Cell:**

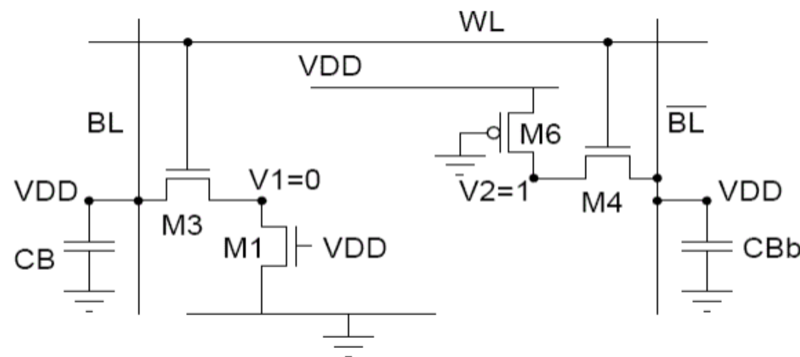
- A low power SRAM cell may be designed by using cross-coupled CMOS inverters. The most important advantage of this circuit topology is that the static power dissipation is very small; essentially, it is limited by small leakage current.
- Other advantages of this design are high noise immunity due to larger noise margins, and the ability to operate at lower power supply voltage. The major disadvantage of this topology is larger cell size.
- The circuit structure of the full CMOS static RAM cell is shown in Figure The memory cell consists of simple CMOS inverters connected back to back, and two access transistors. The access transistors are turned on whenever a word line is activated for read or write operation, connecting the cell to the complementary bit line columns.



Full CMOS RAM Cell

**READ OPERATION:**

- Consider a data read operation, shown in Figure, assuming that logic '0' is stored in the cell. The transistors M2 and M5 are turned off, while the transistors M1 and M6 operate in linear mode.
- Thus internal node voltages are V1 = 0 and V2 = VDD before the cell access transistors are turned on. The active transistors at the beginning of data read operation are shown in Figure



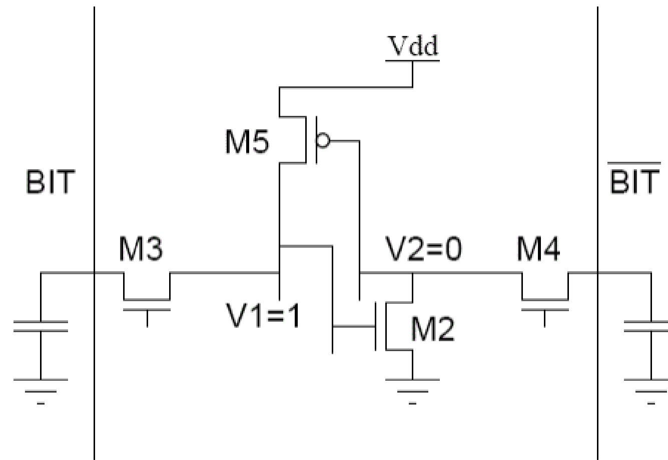
Read Operation

- After the pass transistors M3 and M4 are turned on by the row selection circuitry, the voltage CBb of will not change any significant variation since no current flows through M4.
- On the other hand M1 and M3 will conduct a nonzero current and the voltage level of CB will begin to drop slightly.
- The node voltage V1 will increase from its initial value of '0'V. The node voltage V1 may exceed the threshold voltage of M2 during this process, forcing an unintended change of the stored state.

**WRITE OPERATION:**

- Consider the write '0' operation assuming that logic '1' is stored in the SRAM cell initially. Figure shows the voltage levels in the CMOS SRAM cell at the beginning of the data write operation.
- The transistors M1 and M6 are turned off, while M2 and M5 are operating in the linear mode. Thus the internal node voltage V1 = VDD and V2 = 0 before the access transistors are turned on.

- The column voltage Vb is forced to '0' by the write circuitry. Once M3 and M4 are turned on, we expect the nodal voltage V2 to remain below the threshold voltage of M1, since M2 and M4 are designed according to




SRAM start of write '0'

- The voltage at node 2 would not be sufficient to turn on M1. To change the stored information, i.e., to force V1 = 0 and V2 = VDD, the node voltage V1 must be reduced below the threshold voltage of M2, so that M2 turns off. When the transistor M3 operates in linear region while M5 operates in saturation region.

**ADVANTAGES OF SRAM:**

- **Simplicity:** SRAMs don't require external refresh circuitry or other work in order for them to keep their data intact.
- **Speed:** SRAM is faster than DRAM.

**DISADVATNGES OF SRAM:**

- **Cost:** SRAM is, byte for byte, several times more expensive than DRAM.
- **Size:** SRAMs take up much more space than DRAMs.

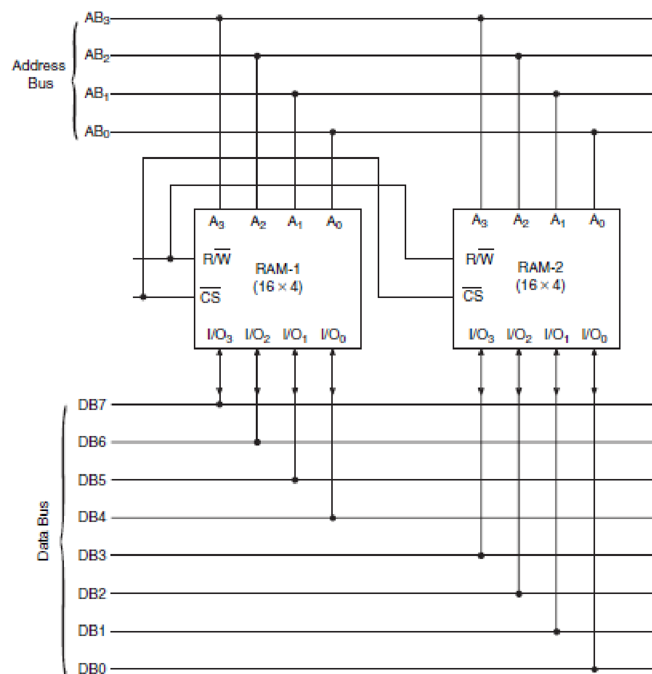
**EXPANDING MEMORY:**

- When a given application requires a RAM or ROM with a capacity that is larger than what is available on a single chip, more than one such chip can be used to achieve the objective.

- The required enhancement in capacity could be either in terms of increasing the word size or increasing the number of memory locations.
- How this can be achieved is illustrated in the following paragraphs with the help of examples.

**WORD SIZE EXPANSION:**

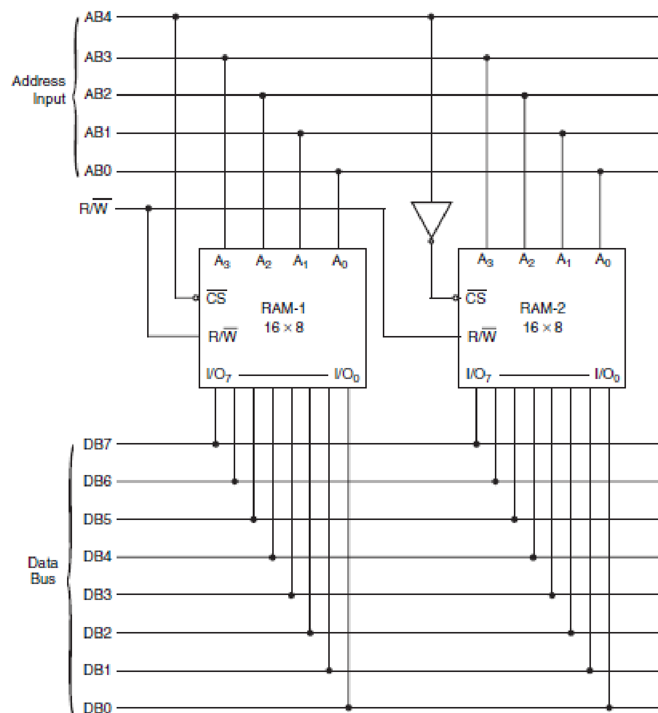
- Let us take up the task of expanding the word size of an available 16×4 RAM chip from four bits to eight bits. Figure shows a diagram where two such RAM chips have been used to achieve the desired effect.




- The arrangement is straightforward. Both chips are selected or deselected together. Also, the input that determines whether it is a ‘read’ or ‘write’ operation is common to both chips.
- That is, both chips are selected for ‘read’ or ‘write’ operation together. The address inputs to the two chips are also common.
- The memory locations corresponding to various address inputs store four higher-order bits in the case of RAM-1 and four lower-order bits in the case of RAM-2.
- In essence, each of the RAM chips stores half of the word. Since the address inputs are common, the same location in each chip is accessed at the same time.

**Memory Location Expansion:**

- Figure shows how more than one memory chip can be used to expand the number of memory locations. Let us consider the use of two 16×8 chips to get a 32×8 chip. A 32×8 chip would need five address input lines.
- Four of the five address inputs, other than the MSB address bit, are common to both 16×8 chips. The MSB bit feeds the input of one chip directly and the input of the other chip after inversion.
- The inputs to the two chips are common. Now, for first half of the memory locations corresponding to address inputs 00000 to 01111 (a total of 16 locations), the MSB bit of the address is '0', with the result that RAM-1 is selected and RAM-2 is deselected.




- For the remaining address inputs of 10000 to 11111 (again, a total of 16 locations), RAM-1 is deselected while RAM-2 is selected. Thus, the overall arrangement offers a total of 32 locations, 16 provided by RAM-1 and 16 provided by RAM-2. The overall capacity is thus 32×8.