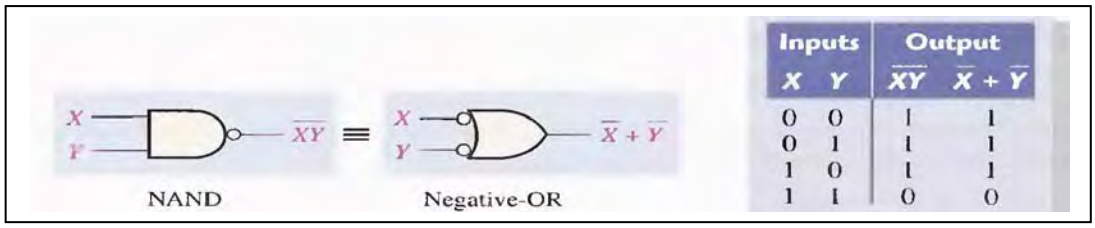
## DEMORGAN'S THEOREMS

**One of DeMorgan's** theorems stated as follows:
The complement of a product of variables is equal to the sum of the complements of the variables.

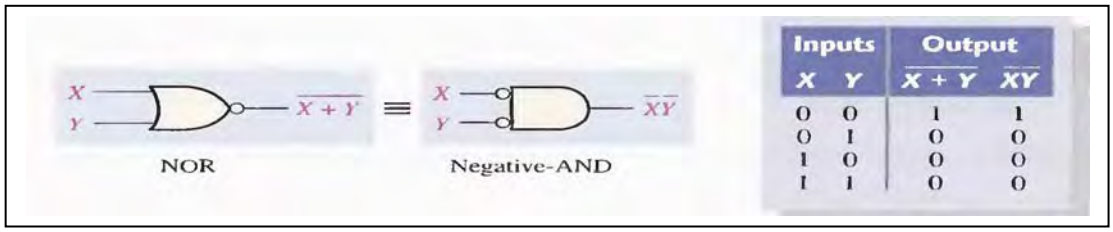
$$\overline{XY} = \overline{X} + \overline{Y}$$




| Inputs | | Output | |
|---|---|---|---|
| X | Y | $\overline{XY}$ | $\overline{X} + \overline{Y}$ |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

**DeMorgan's second** theorem is stated as follows:
The complement of a sum of variables is equal to the product of the complements of the variables.

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




| Inputs | | Output | |
|---|---|---|---|
| X | Y | $\overline{X + Y}$ | $\overline{XY}$ |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |

Example:

Apply DeMorgan's theorems to each of the following expressions:

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

*Solution* **(a)** Let $A + B + C = X$ and $D = Y$. The expression $\overline{(A + B + C)D}$ is of the form $\overline{XY} = \overline{X} + \overline{Y}$ and can be rewritten as

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

Next, apply DeMorgan's theorem 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 expression $\overline{ABC + DEF}$ is of the form $\overline{X + Y} = \overline{X}\,\overline{Y}$ and can be rewritten as

$$\overline{ABC + DEF} = (\overline{ABC})(\overline{DEF})$$

Next, apply DeMorgan's theorem to each of the terms $\overline{ABC}$ and $\overline{DEF}$.

$$(\overline{ABC})(\overline{DEF}) = (\overline{A} + \overline{B} + \overline{C})(\overline{D} + \overline{E} + \overline{F})$$

**(c)** Let $A\overline{B} = X$, $\overline{C}D = Y$, and $EF = Z$. The expression $\overline{A\overline{B} + \overline{C}D + EF}$ is of the form $\overline{X + Y + Z} = \overline{X}\,\overline{Y}\,\overline{Z}$ and can be rewritten as

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

Next, apply DeMorgan's theorem to each of the terms $\overline{A\overline{B}}$, $\overline{\overline{C}D}$, and $\overline{EF}$.

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

**SIMPLIFICATION USING BOOLEAN ALGEBRA**

A simplified Boolean expression uses the fewest gates possible to implement a given expression. Simplification means fewer gates for the same function

Example:

Using Boolean algebra techniques, simplify this expression:

$$AB + A(B + C) + B(B + C)$$

*Solution* The following is not necessarily the only approach.

**Step 1:** Apply the distributive law to the second and third terms in the expression, as follows:

$$AB + AB + AC + BB + BC$$

**Step 2:** Apply rule 7 ($BB = B$) to the fourth term.

$$AB + AB + AC + B + BC$$

**Step 3:** Apply rule 5 ($AB + AB = AB$) to the first two terms.

$$AB + AC + B + BC$$

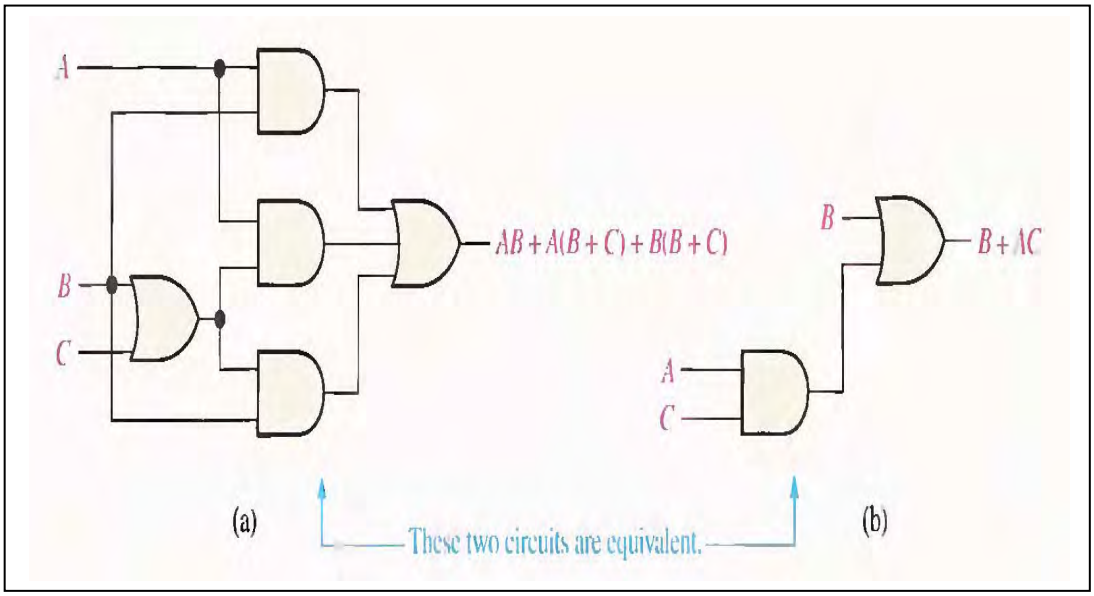
**Step 4:** Apply rule 10 ($B + BC = B$) to the last two terms.

$$AB + AC + B$$

**Step 5:** Apply rule 10 ($AB + B = B$) to the first and third terms.

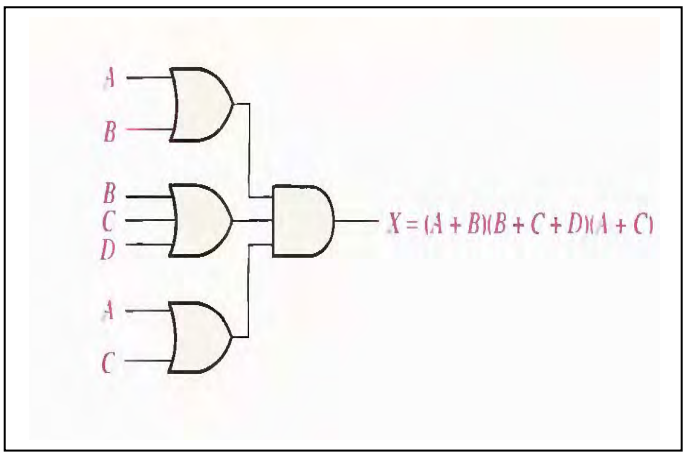
$$B + AC$$

At this point the expression is simplified as much as possible. Once you gain experience in applying Boolean algebra, you can often combine many individual steps.



## Product-of-Sums (POS) Form

When two or more sum terms are multiplied, the resulting expression is a product-of-sums (POS). Implementing a POS expression simply requires ANDing the outputs of two or more OR gates.




A POS expression is equal to 0 only if one or more of the sum terms in the expression is equal to 0.

Example

Determine the binary values of the variables for which the following standard POS expression is equal to 0:

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

*Solution* The term $A + B + C + D$ is equal to 0 when $A = 0, B = 0, C = 0$, and $D = 0$.

$$A + B + C + D = 0 + 0 + 0 + 0 = 0$$

The term $A + \overline{B} + \overline{C} + D$ is equal to 0 when $A = 0. B = 1, C = 1.$ and $D = 0.$

$$A + \overline{B} + \overline{C} + D = 0 + \overline{1} + \overline{1} + 0 = 0 + 0 + 0 + 0 = 0$$

The term $\overline{A} + \overline{B} + \overline{C} + \overline{D}$ is equal to 0 when $A = 1, B = 1, C = 1,$ and $D = 1.$

$$\overline{A} + \overline{B} + \overline{C} + \overline{D} = \overline{1} + \overline{1} + \overline{1} + \overline{1} = 0 + 0 + 0 + 0 = 0$$

The POS expression equals 0 when any of the three sum terms equals 0.

## BOOLEAN EXPRESSIONS AND TRUTH TABLES

### Converting SOP Expressions to Truth Table Format

The first step in constructing a truth table is to list all possible combinations of binary values of the variables in the expression. Next, convert the SOP expression to standard form if it is not already. Finally, place a 1 in the output column (X) for each binary value that makes the standard SOP expression a 1 and place a 0 for all the remaining binary values.

## Example

Develop a truth table for the standard SOP expression $\overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + ABC$.

*Solution* There are three variables in the domain, so there are eight possible combinations of binary values of the variables as listed in the left three columns of Table 4–6. The binary values that make the product terms in the expressions equal to 1 are $\overline{A}\,\overline{B}C$: 001; $A\overline{B}\,\overline{C}$: 100; and $ABC$: 111. For each of these binary values, place a 1 in the output column as shown in the table. For each of the remaining binary combinations, place a 0 in the output column.

| INPUTS | | | OUTPUT | |
|---|---|---|---|---|
| A | B | C | X | PRODUCT TERM |
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | $\overline{A}\,\overline{B}C$ |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | $A\overline{B}\,\overline{C}$ |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | |
| 1 | 1 | 1 | 1 | $ABC$ |

**Converting POS Expressions to Truth Table Format**

To construct a truth table from a POS expression, list all the possible combinations of binary values of the variables just as was done for the SOP expression. Next, convert the POS expression to standard form if it is not already. Finally, place a 0 in the output column (X) for each binary value that makes the expression a 0 and place a 1 for all the remaining binary values.

Determine the truth table for the following standard POS expression:

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

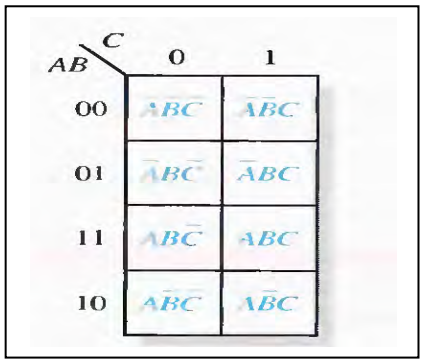
*Solution* There are three variables in the domain and the eight possible binary values are listed in the left three columns of Table 4–7. The binary values that make the sum terms in the expression equal to 0 are $A + B + C$: 000; $A + \overline{B} + C$: 010; $A + \overline{B} + \overline{C}$: 011; $\overline{A} + B + \overline{C}$: 101; and $\overline{A} + \overline{B} + C$: 110. For each of these binary values, place a 0 in the output column as shown in the table. For each of the remaining binary combinations, place a 1 in the output column.

| INPUTS | | | OUTPUT | |
|---|---|---|---|---|
| A | B | C | X | SUM TERM |
| 0 | 0 | 0 | 0 | $(A + B + C)$ |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | $(A + \overline{B} + C)$ |
| 0 | 1 | 1 | 0 | $(A + \overline{B} + \overline{C})$ |
| 1 | 0 | 0 | 1 | |
| 1 | 0 | 1 | 0 | $(\overline{A} + B + \overline{C})$ |
| 1 | 1 | 0 | 0 | $(\overline{A} + \overline{B} + C)$ |
| 1 | 1 | 1 | 1 | |

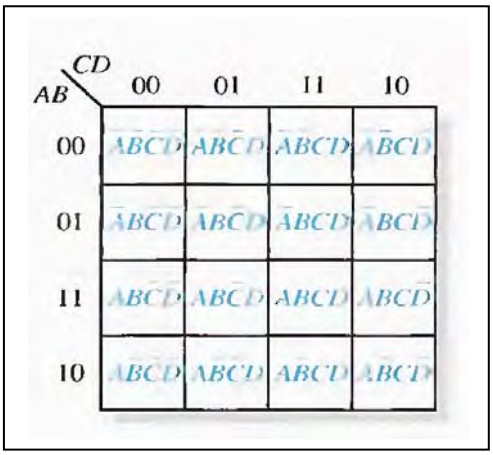
## THE KARNAUGH MAP

The purpose of a Karnaugh map is to simplify a Boolean expression.
The number of cells in a Karnaugh map is equal to the total number of possible input variable combinations as is the number of rows in a truth table. For three variables, the number of cells is $2^3 = 8$. For four variables, the number of cells is $2^4 = 16$.

### 3-Variable Karnaugh Map




### 4-Variable Karnaugh Map




### Karnaugh Map Simplification of SOP Expressions

### Determining the Minimum SOP Expression from the Map

**1.** Group the cells that have 1 s. Each group of cells containing 1 s creates one product term composed of all variables that occur in only one form (either un complemented or complemented) within the group. Variables that occur both un complemented and complemented within the group are eliminated. These called contradictory variables.
**2**. Determine the minimum product term for each group.
**A. For a 3-variable map**:
(1) l-cell group yields a 3-variable product term
(2) 2-cell group yields a 2-variable product term
(3) 4-cell group yields a 1-variable term
(4) 8-cell group yields a value of 1 for the expression
**B. For a 4-variable map:**
(1) 1-cell group yields a 4-variable product term
(2) 2-cell group yields a 3-variable product term
(3) 4-cell group yields a 2-variable product term
(4) 8-cell group yields a 1-variable term
(5) 16-cell group yields a value of 1 for the expression
3. When all the minimum product terms derived from the Karnaugh map, they summed to form the minimum SOP expression.

Example:

Determine the product terms for each of the Karnaugh maps in Figure 4–32 and write the resulting minimum SOP expression.

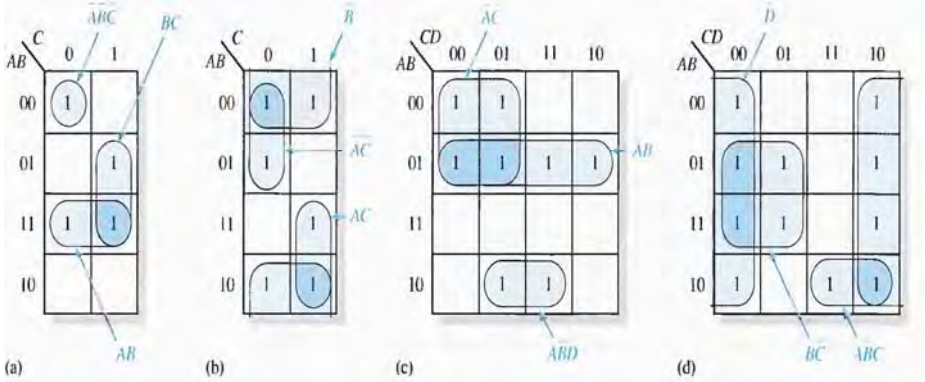



FIGURE 4–32

*Solution* The resulting minimum product term for each group is shown in Figure 4–32. The minimum SOP expressions for each of the Karnaugh maps in the figure are

(a) $AB + BC + \overline{A}\,\overline{B}\,\overline{C}$ (b) $\overline{B} + \overline{A}\,\overline{C} + AC$

(c) $\overline{A}B + \overline{A}\,\overline{C} + A\overline{B}D$ (d) $\overline{D} + A\overline{B}C + B\overline{C}$

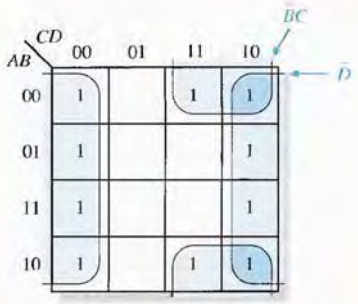
Example:

Use a Karnaugh map to minimize the following SOP expression:

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

*Solution* The first term $\overline{B}\,\overline{C}\,\overline{D}$ must be expanded into $A\overline{B}\,\overline{C}\,\overline{D}$ and $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ to get the standard SOP expression, which is then mapped; and the cells are grouped as shown in Figure 4–34.

FIGURE 4–34



Notice that both groups exhibit "wrap around" adjacency. The group of eight is formed because the cells in the outer columns are adjacent. The group of four is formed to pick up the remaining two 1s because the top and bottom cells are adjacent. The product term for each group is shown. The resulting minimum SOP expression is

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

Keep in mind that this minimum expression is equivalent to the original standard expression.

## KARNAUGH MAP POS MINIMIZATION

### Karnaugh Map Simplification of POS Expressions

The process for minimizing a POS expression is the same as for an SOP expression except that you group 0s to produce minimum sum terms instead of grouping 1s to produce minimum product terms.

Example

Use a Karnaugh map to minimize the following POS expression:

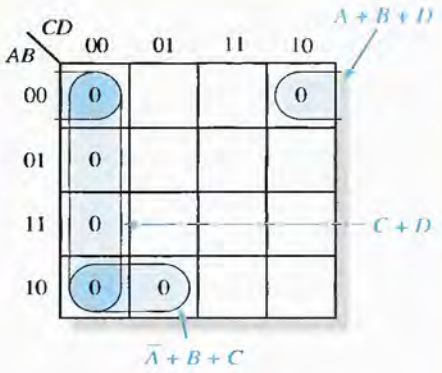
$$(B + C + D)(A + B + \overline{C} + D)(\overline{A} + B + C + \overline{D})(A + \overline{B} + C + D)(\overline{A} + \overline{B} + C + D)$$

*Solution* The first term must be expanded into $\overline{A} + B + C + D$ and $A + B + C + D$ to get a standard POS expression, which is then mapped; and the cells are grouped as shown in Figure 4–40. The sum term for each group is shown and the resulting minimum POS expression is

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

Keep in mind that this minimum POS expression is equivalent to the original standard POS expression.

FIGURE 4–40

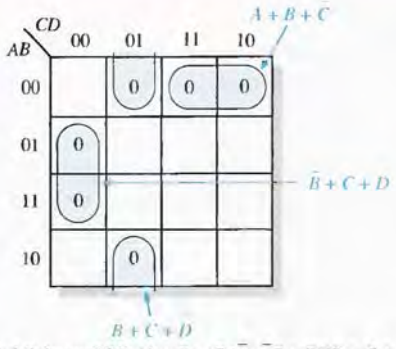


## Converting Between POS and SOP Using the Karnaugh Map

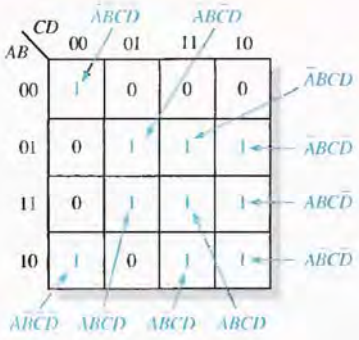
Using a Karnaugh map, convert the following standard POS expression into a minimum POS expression, a standard SOP expression, and a minimum SOP expression.

$$(\bar{A} + \bar{B} + C + D)(A + \bar{B} + C + D)(A + B + C + \bar{D})$$
$$(A + B + \bar{C} + \bar{D})(\bar{A} + B + C + \bar{D})(A + B + \bar{C} + D)$$

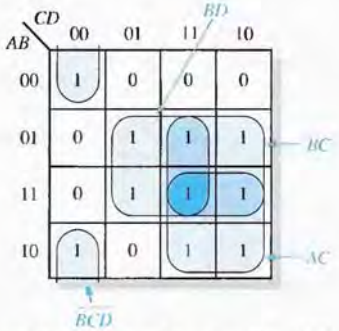
*Solution* The 0s for the standard POS expression are mapped and grouped to obtain the minimum POS expression in Figure 4–41(a). In Figure 4–41(b), 1s are added to the cells that do not contain 0s. From each cell containing a 1, a standard product term is obtained as indicated. These product terms form the standard SOP expression. In Figure 4–41(c), the 1s are grouped and a minimum SOP expression is obtained.




(a) Minimum POS: $(A + B + C)(\bar{B} + \bar{C} + D)(B + C + \bar{D})$




(b) Standard SOP:
$\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D} + AB\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D} + AB\bar{C}D + A\bar{B}CD + ABCD$



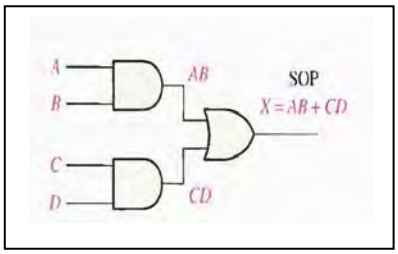

(c) Minimum SOP: $AC + BC + BD + \bar{B}\bar{C}\bar{D}$

## Don’t care Karnough condition

# Chapter 4
# COMBINATIONAL LOGIC CIRCUITS

- **AND-OR Logic**

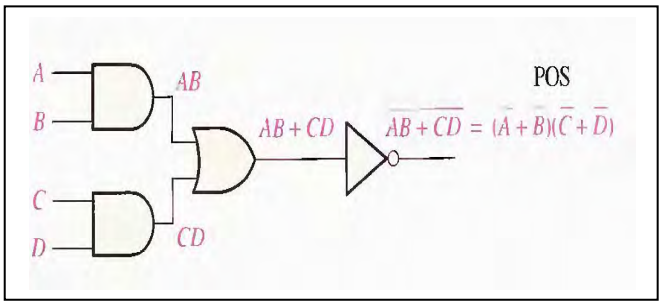
AND-OR circuit consisting of two (2-input) AND gates and one 2-input OR gate;




- **AND-OR-Invert Logic**

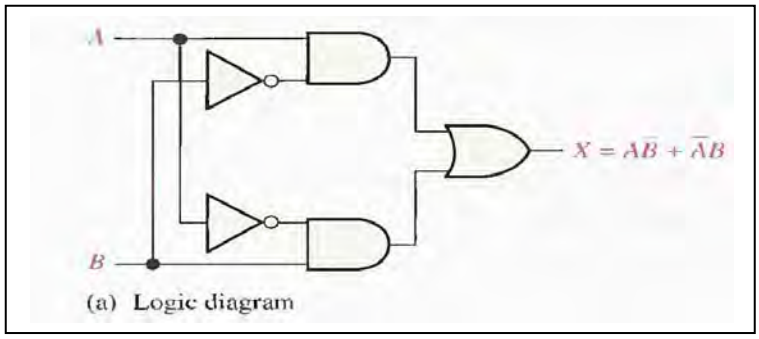
When the output of an AND-OR circuit is complemented (inverted), it results in an AND-OR-Invert circuit.

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

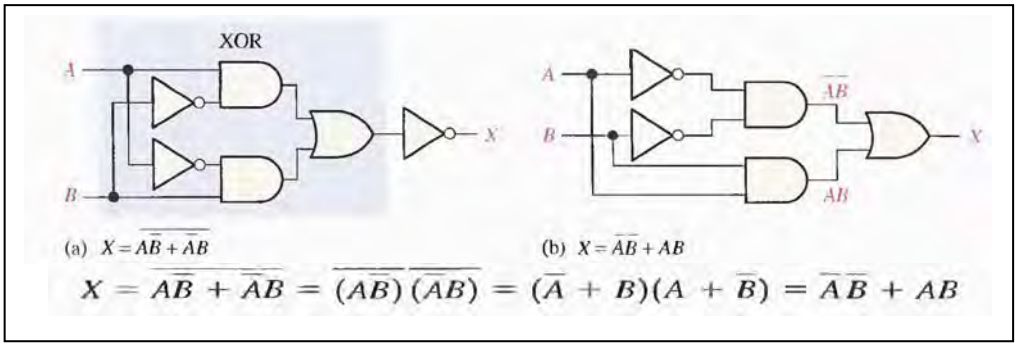



- **Exclusive-OR logic** we can used AND-OR to represent X-OR

$$X = A \oplus B$$




- **Exclusive-NOR Logic**



**Example**

Develop a logic circuit with four input variables that will only produce a 1 output when exactly three input variables are 1s.

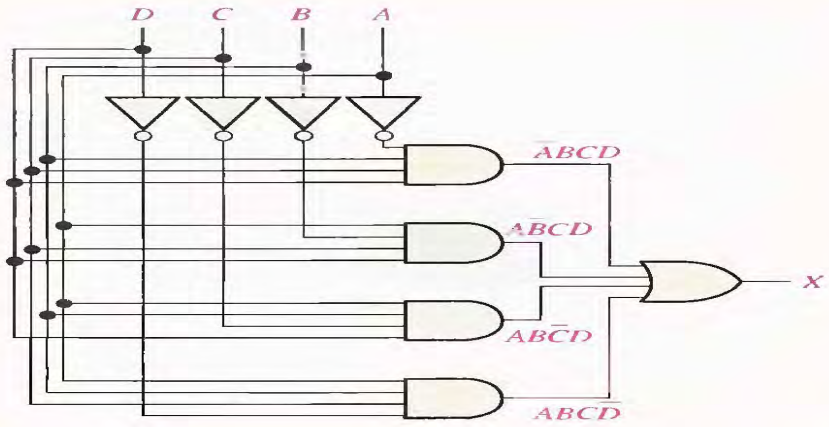
*Solution* Out of sixteen possible combinations of four variables, the combinations in which there are exactly three 1s are listed in Table 5–5, along with the corresponding product term for each.

| A | B | C | D | PRODUCT TERM |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | $\bar{A}BCD$ |
| 1 | 0 | 1 | 1 | $A\bar{B}CD$ |
| 1 | 1 | 0 | 1 | $AB\bar{C}D$ |
| 1 | 1 | 1 | 0 | $ABC\bar{D}$ |

The product terms are ORed to get the following expression:

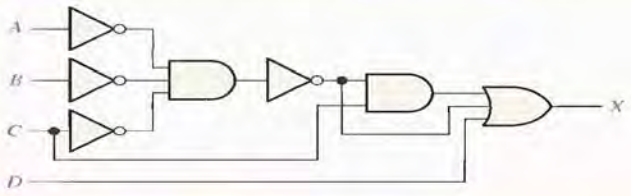
$$X = \bar{A}BCD + A\bar{B}CD + AB\bar{C}D + ABC\bar{D}$$

This expression is implemented in Figure 5–11 with AND-OR logic.




**Example**

Reduce the combinational logic circuit in Figure 5–12 to a minimum form.




*Solution* The expression for the output of the circuit is

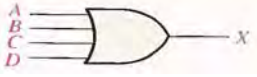
$$X = (\overline{\bar{A}\bar{B}\bar{C}})C + \overline{\bar{A}\bar{B}\bar{C}} + D$$

Applying DeMorgan's theorem and Boolean algebra,

$$\begin{aligned} X &= (\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}})C + \bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}} + D \\ &= AC + BC + CC + A + B + C + D \\ &= AC + BC + C + A + B + C + D \\ &= C(A + B + 1) + A + B + D \\ X &= A + B + C + D \end{aligned}$$

The simplified circuit is a 4-input OR gate as shown in Figure 5–13.

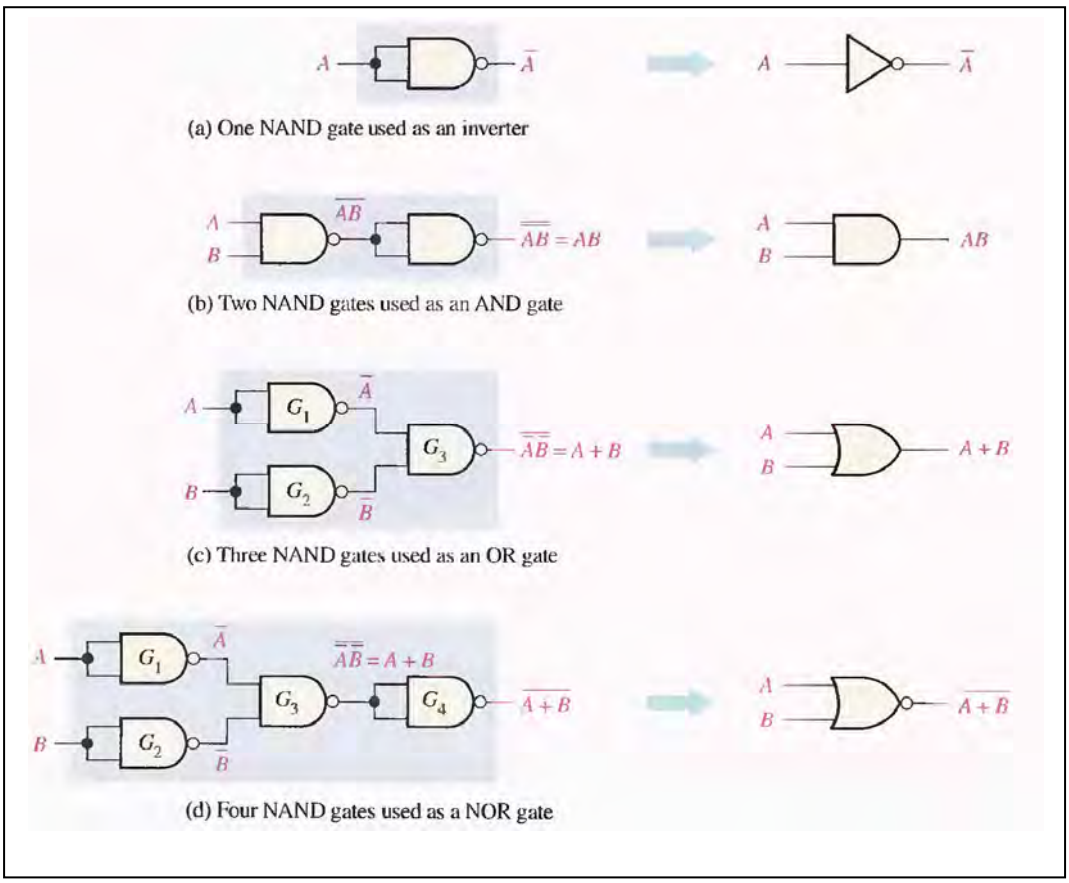
FIGURE 5–13



## THE UNIVERSAL PROPERTY OF NAND AND NOR GATES

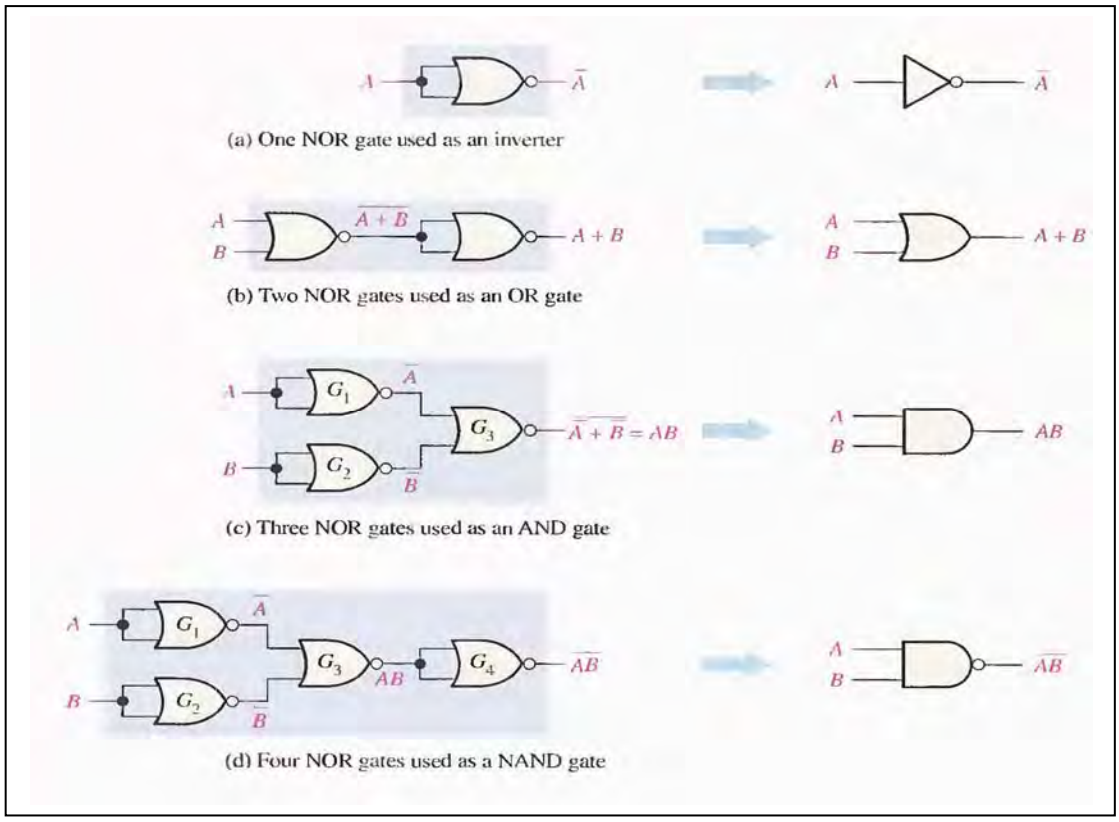
- **The NAND Gate as a Universal Logic Element**

The NAND gate is a universal gate because it can be used to produce the NOT, the AND, the OR, and the NOR functions. An inverter can be made from a NAND gate by connecting all of the inputs together and creating, in effect, a single input,



## The NOR Gate as a Universal Logic Element

Like the NAND gate, the NOR gate can be used to produce the NOT, AND. OR and NAND functions. As shown below:




## COMBINATIONAL LOGIC USING NAND AND NOR GATES

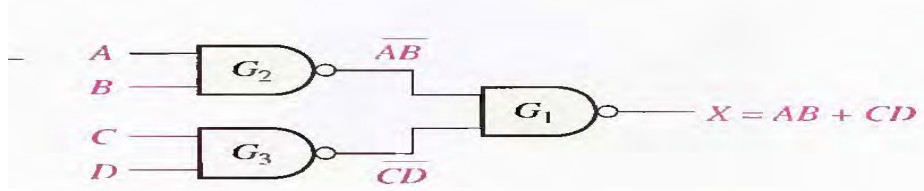
- **NAND Logic**

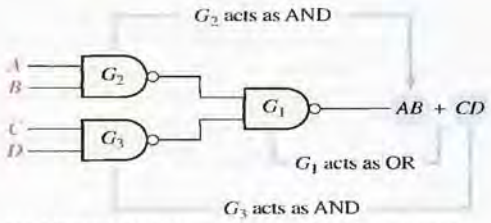
NAND gate can function as either a NAND or a negative-OR because, by DeMorgan"s theorem,

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

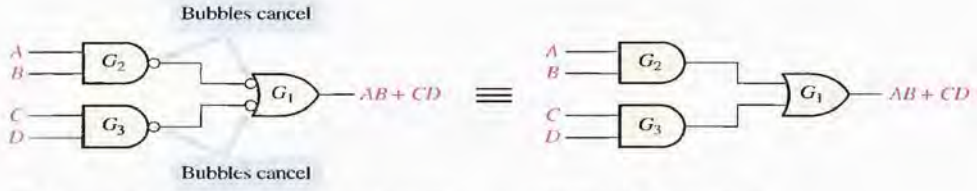
NAND — negative-OR

**Example:**




(a) Original NAND logic diagram showing effective gate operation relative to the output expression



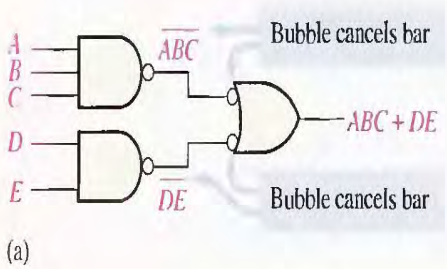

(b) Equivalent NAND/Negative-OR logic diagram (c) AND-OR equivalent

**Example:**

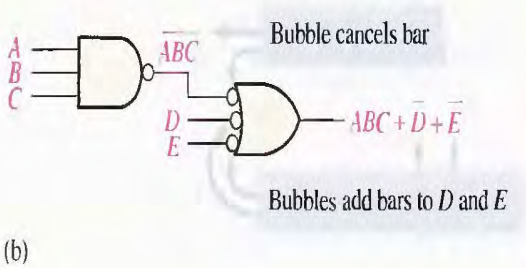
Implement each expression with NAND logic using appropriate dual symbols:

**(a)** $ABC + DE$ **(b)** $ABC + \overline{D} + \overline{E}$

Solution:




(a)




(b)

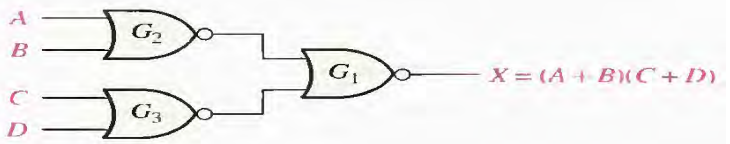
- **NOR Logic**

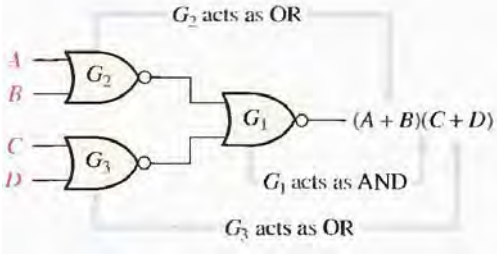
A NOR gate can function as either a NOR or a negative-AND, as shown by DeMorgan's theorem.

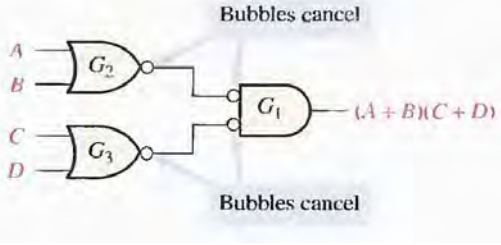
$$\overline{A + B} = \overline{A}\,\overline{B}$$

NOR — negative-AND

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



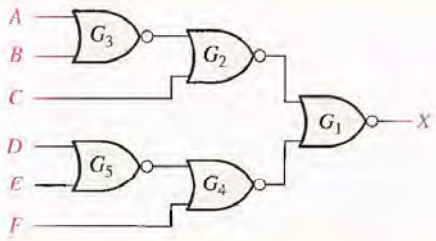




Example

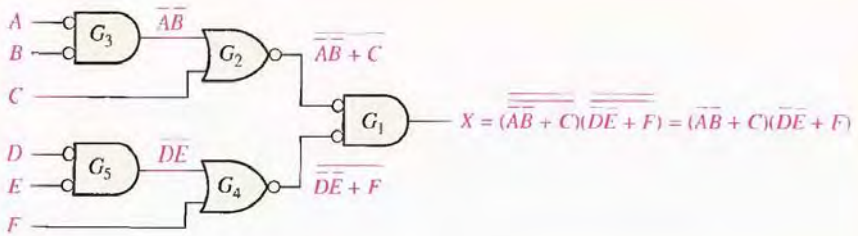
Using appropriate dual symbols, redraw the logic diagram and develop the output expression for the circuit in Figure 5–27.

**FIGURE 5–27**




*Solution* Redraw the logic diagram with the equivalent negative-AND symbols as shown in Figure 5–28. Writing the expression for $X$ directly from the indicated operation of each gate,

$$X = (\overline{A}\,\overline{B} + C)(\overline{D}\,\overline{E} + F)$$

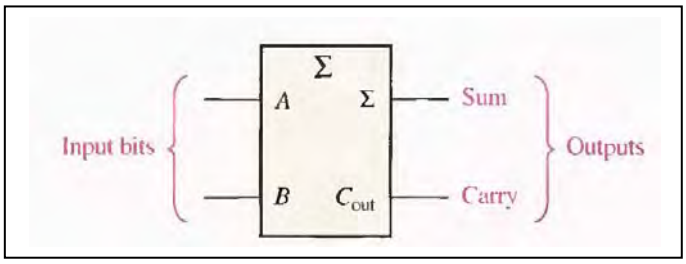


# CHAPYER FIVE BASIC ADDERS

## The Half-Adder

The half-adder accepts two binary digits on its inputs and produces two binary digits on its outputs, a **sum** bit and a **carry** bit.

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$




**Truth table for half adder**

From the operation of the half-adder as stated in Table

The carry output expression

Notice that the output Carry ($C_{out}$) is a 1 only when both A and B are 1 s: therefore. ($C_{out}$) can be expressed as the AND of the input variables.

| A | B | $C_{out}$ | Σ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

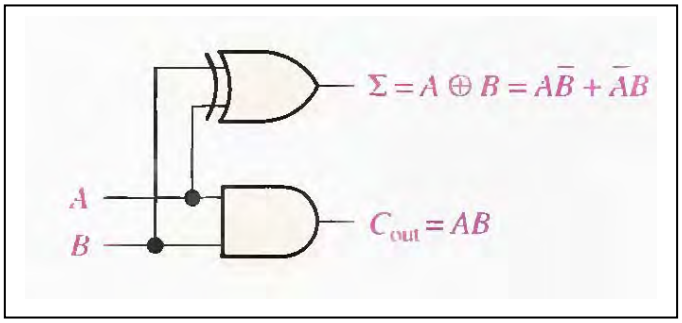
Σ = sum
$C_{out}$ = output carry
A and B = input variables (operands)

$$C_{out} = AB$$

Now observe that the sum output (Σ) is a 1 only if the input variables, *A* and *B*, are not equal. The sum can therefore be expressed as the exclusive-OR of the input variables.

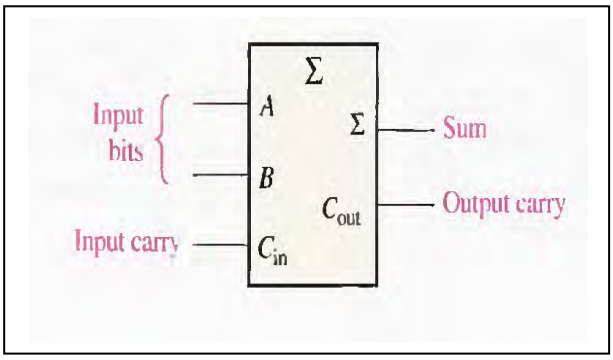
$$\Sigma = A \oplus B$$

## Half-adder logic diagram.



## The Full-Adder

The full-adder accepts two input bits and an input carry and generates a sum output and an output carry.




| A | B | $C_{in}$ | $C_{out}$ | Σ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$C_{in}$ = input carry, sometimes designated as *CI*

$C_{out}$ = output carry, sometimes designated as *CO*

Σ = sum

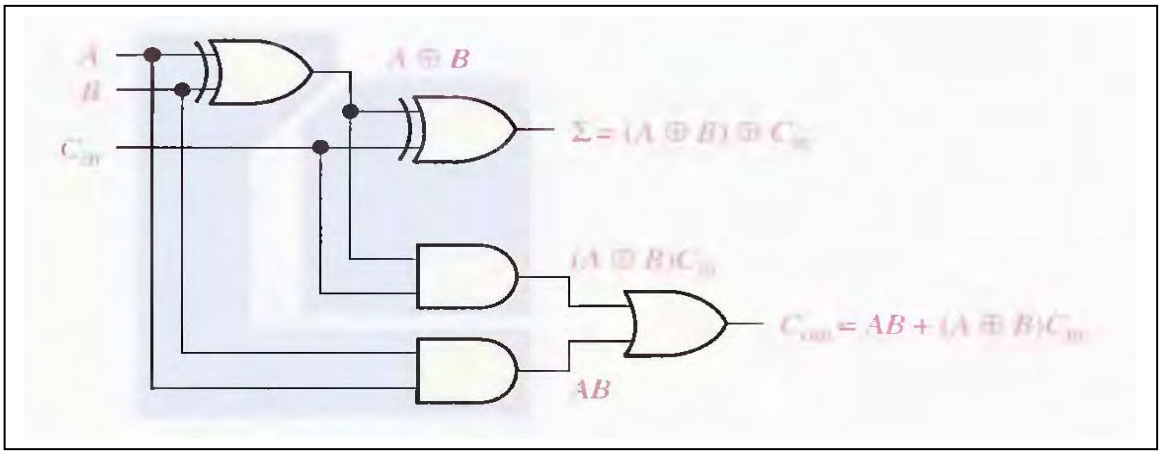
*A* and *B* = input variables (operands)

## Full-Adder Logic

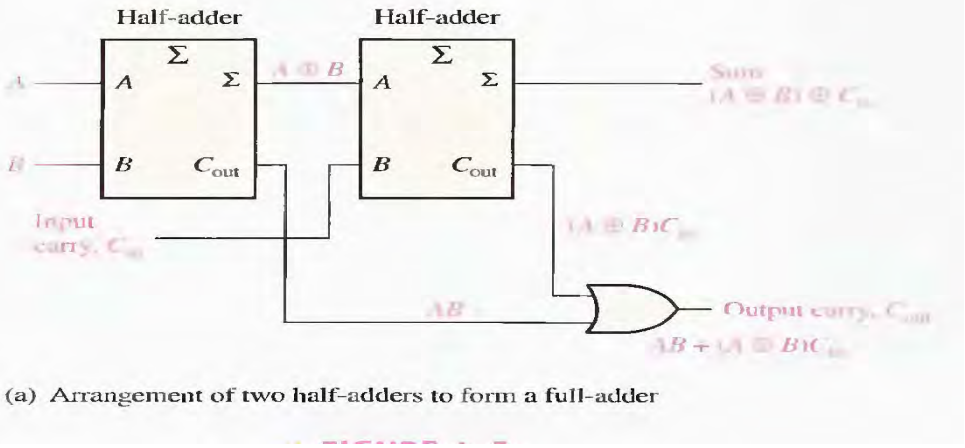
Full-Adder Logic The full-adder must add the two input bits and the input carry from the half-adder.

$$\Sigma = (A \oplus B) \oplus C_{in}$$

$$C_{out} = AB + (A \oplus B)C_{in}$$



There are two half-adders, connected as shown in the block diagram with their output carries ORed.




(a) Arrangement of two half-adders to form a full-adder

Example:

For each of the three full-adders in Figure 6–6, determine the outputs for the inputs shown.

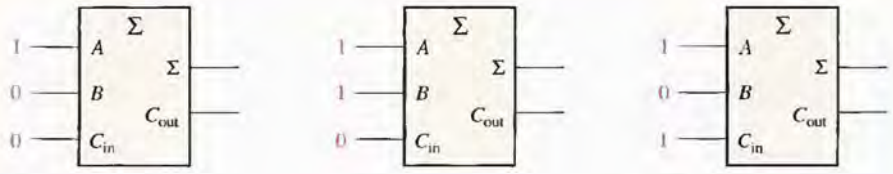



FIGURE 6–6

*Solution* **(a)** The input bits are $A = 1$, $B = 0$, and $C_{in} = 0$.

$1 + 0 + 0 = 1$ with no carry

Therefore, $\Sigma = 1$ and $C_{out} = 0$.

**(b)** The input bits are $A = 1$, $B = 1$, and $C_{in} = 0$.

$1 + 1 + 0 = 0$ with a carry of 1

Therefore, $\Sigma = 0$ and $C_{out} = 1$.

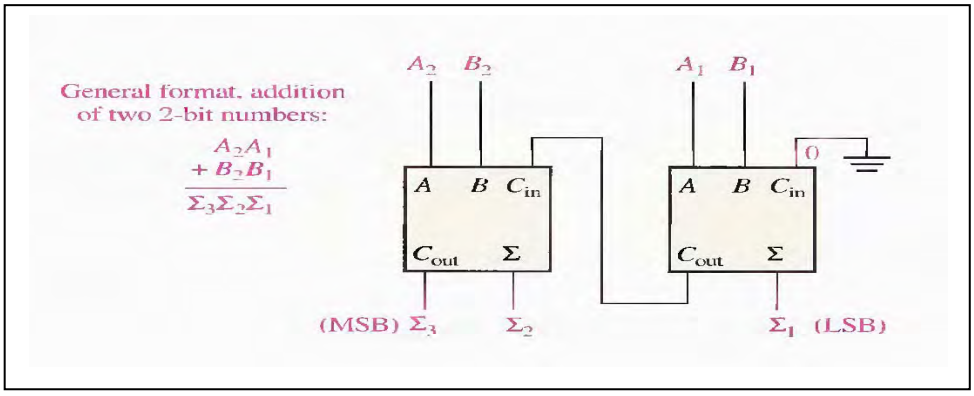
**(c)** The input bits are $A = 1$, $B = 0$, and $C_{in} = 1$.

$1 + 0 + 1 = 0$ with a carry of 1

Therefore, $\Sigma = 0$ and $C_{out} = 1$.

## PARALLEL BINARY ADDERS

Two or more full-adders have connected to form parallel binary adders.
To add two binary numbers, a full-adder is required for each bit in the numbers. So for 2-bit numbers, two adders are needed; for 4-bit numbers, four adders are used; and so on. The carry output of each adder is connected to the carry input of the next higher-order adder, as shown in Figure below for a 2-bit adder. Notice that either a half-adder can be used for the least significant position or the carry input of a full-adder can be made 0 (grounded) because there is no carry input to the least significant bit position.

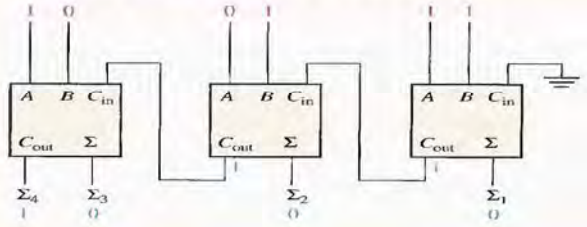



In Figure above the least significant bits (LSB) of the two numbers are represented by $A_1$ and B1. The next higher-order bits are represented by $A_2$ and $B_2$ . The three sum bits are $\sum_1$,$\sum_2$and$\sum_3$ ' Notice that the output carry from the left-most full-adder becomes the most significant bit (MSB) in the sum, $\sum_3$.

Example

Determine the sum generated by the 3-bit parallel adder in Figure 6–8 and show the intermediate carries when the binary numbers 101 and 011 are being added.

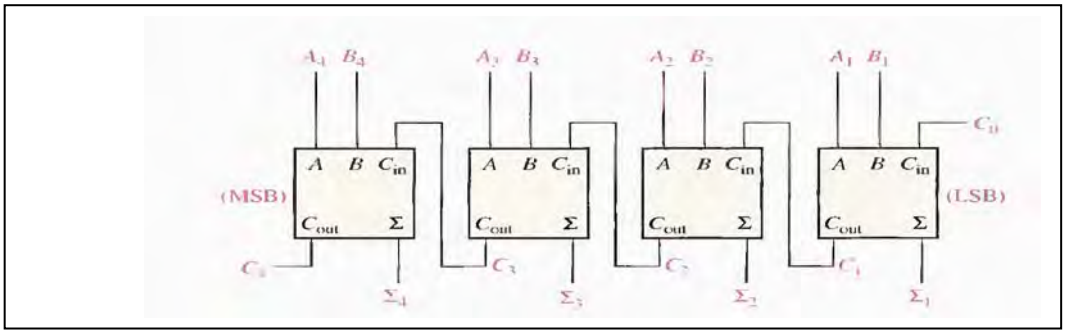
FIGURE 6–8




*Solution* The LSBs of the two numbers are added in the right-most full-adder. The sum bits and the intermediate carries are indicated in blue in Figure 6–8.

## Four-Bit Parallel Adders

A group of four bits is called a nibble. A basic 4-bit parallel adder is implemented with four full-adder stages as shown in Figure




## Truth Table for a 4-Bit Parallel Adder

Table 6-3 is the truth table for a 4-bit adder. On some data sheets, truth tables may be called function tables or functional truth tables. The subscript n represents the adder bit and can be 1, 2, 3, or 4 for the 4-bit adder. $C_{n-1}$ is the carry from the previous adder. Carries $C_1$ , $C_2$ and $C_3$ are generated internally. $C_n$ is an external carry input and $C_4$ is an output.

| $C_{n-1}$ | $A_n$ | $B_n$ | $\Sigma_n$ | $C_n$ |
|---|---|---|---|---|
| 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 |

**Example:**

Use the 4-bit parallel adder truth table (Table 6–3) to find the sum and output carry for the addition of the following two 4-bit numbers if the input carry ($C_{n-1}$) is 0:

$$A_4A_3A_2A_1 = 1100 \quad \text{and} \quad B_4B_3B_2B_1 = 1100$$

*Solution* For $n = 1$: $A_1 = 0$, $B_1 = 0$, and $C_{n-1} = 0$. From the 1st row of the table,

$$\Sigma_1 = 0 \quad \text{and} \quad C_1 = 0$$

For $n = 2$: $A_2 = 0$, $B_2 = 0$, and $C_{n-1} = 0$. From the 1st row of the table,

$$\Sigma_2 = 0 \quad \text{and} \quad C_2 = 0$$

For $n = 3$: $A_3 = 1$, $B_3 = 1$, and $C_{n-1} = 0$. From the 4th row of the table,

$$\Sigma_3 = 0 \quad \text{and} \quad C_3 = 1$$

For $n = 4$: $A_4 = 1$, $B_4 = 1$, and $C_{n-1} = 1$. From the last row of the table,

$$\Sigma_4 = 1 \quad \text{and} \quad C_4 = 1$$

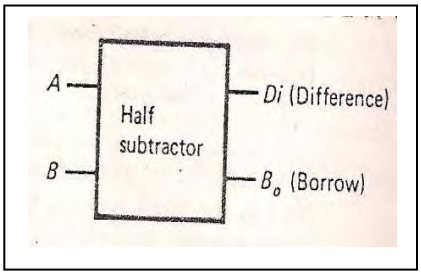
$C_4$ becomes the output carry; the sum of 1100 and 1100 is 11000.

## Half Subtractor

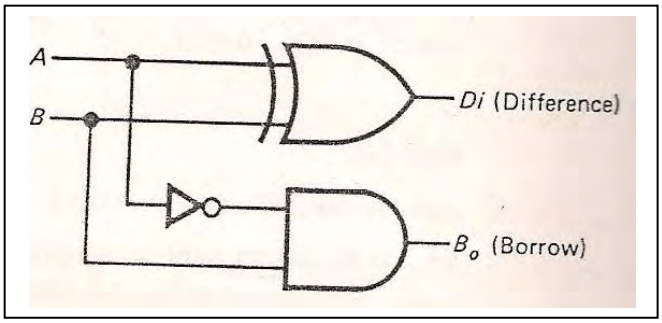
A subtracted from B the output is Di (Difference), and if B greater than A we need to borrow and labeled ($B_o$)
The truth table and block diagram of half subtractor as shown below

TRUTH TABLE

| INPUTS | | OUTPUTS | |
|---|---|---|---|
| A | B | Di | $B_o$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| A – B | | Difference | Borrow out |




Logic Diagram half subtractor.



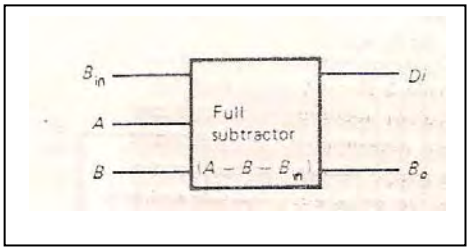

The Boolean expression for half subtractor

$D_i$ = $A \oplus B$

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

## Full subtractor

Full subtractor we have Barrow in ($B_{in}$) the truth table as shown below

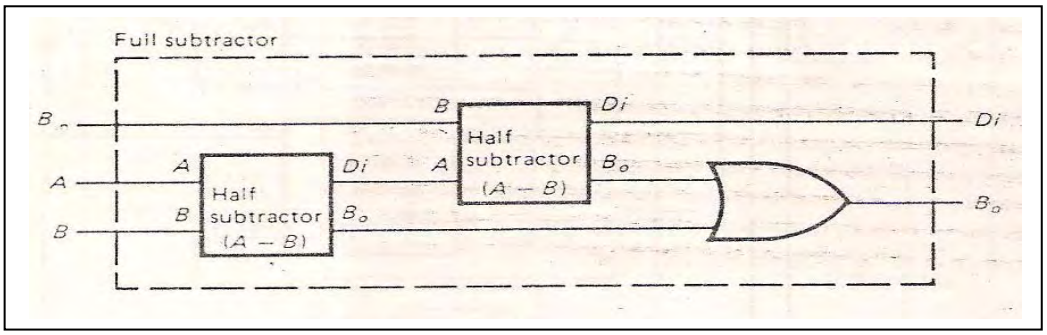



Block symbol

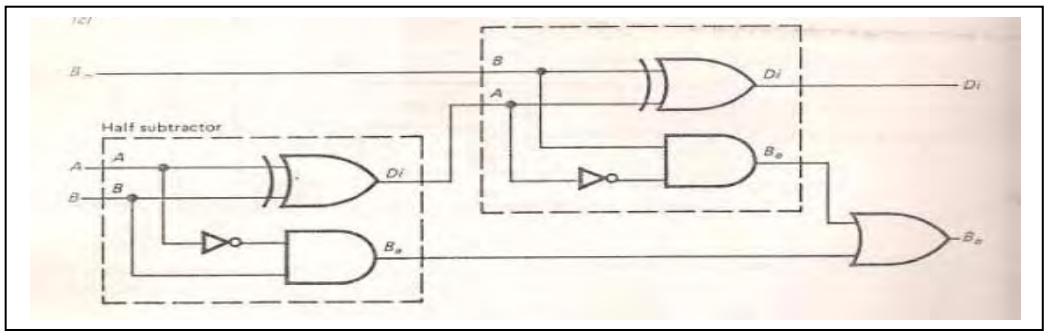
TRUTH TABLE

| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| A | B | $B_{in}$ | Di | $B_o$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| A – B – $B_{in}$ | | | Difference | Borrow out |

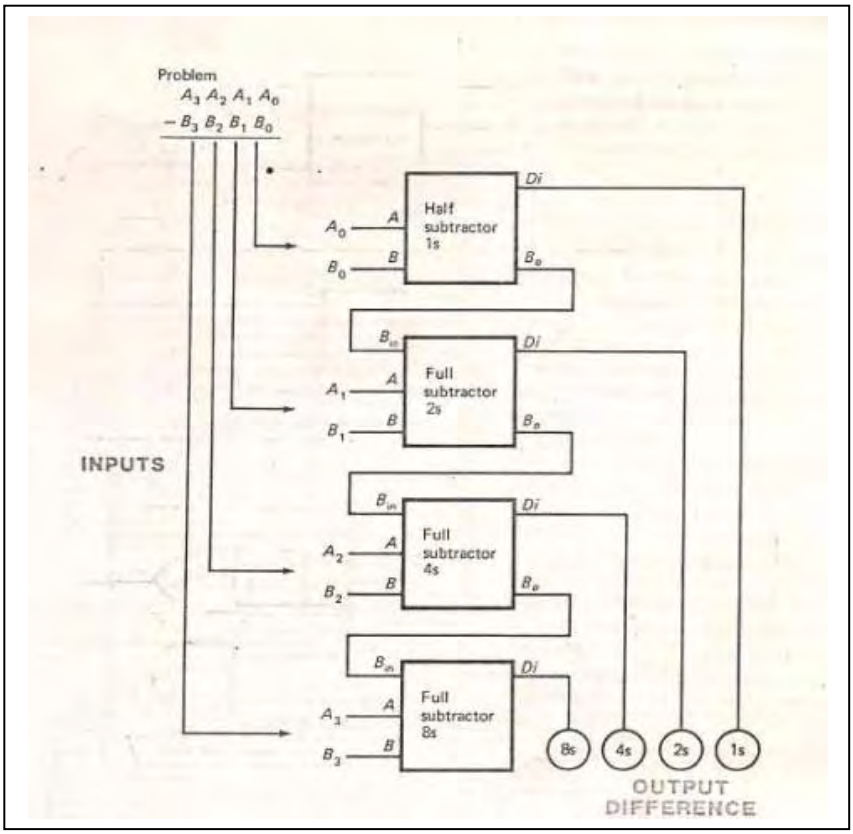
We can construct full subtractor by using half subtractor as shown below




**Logic diagram as shown**

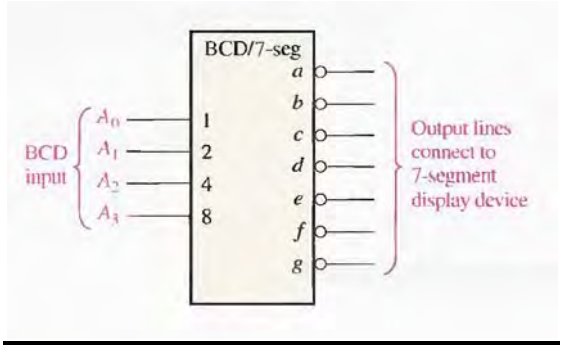



**4 bit parallel subtractor:** the form bellow 4 bit parallel subtractor that can be subtract binary number $B_3B_2B_1B_0$ from binary number $A_3A_2A_1A_0$, Notice that the top subtractor (half subtractor ) subtract the LSBs(1s place). The $B_o$ of the 1s subtractor is tied to next subtractor as $B_{in}$



## The BCD-to-7-Segment Decoder

The BCD-to-7-segment decoder accepts the BCD code on its inputs and provides outputs to drive 7-segment display devices to produce a decimal readout. The logic diagram for a basic 7-segment decoder is shown in Figure 6–34.

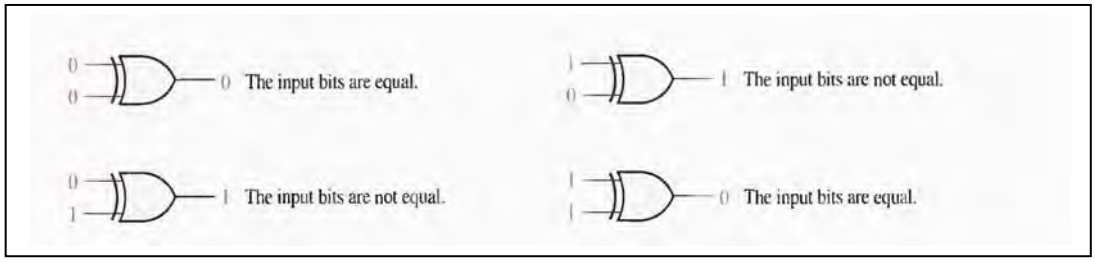


## COMPARATORS

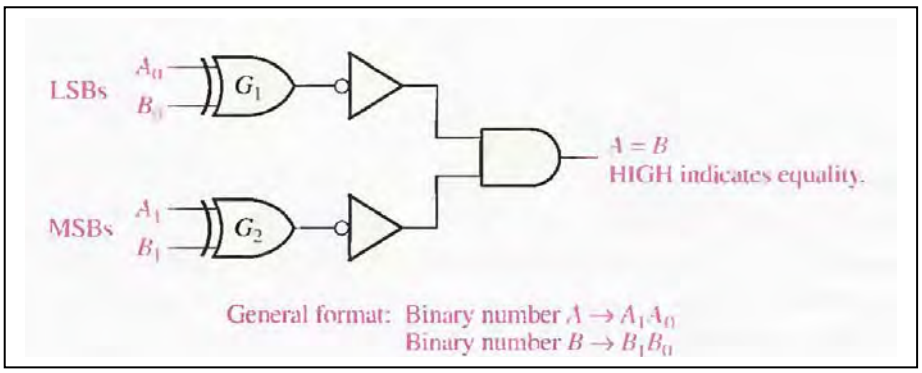
The basic function of a comparator is to compare the magnitudes of two binary quantities to determine the relationship of those quantities.

### Equality

Exclusive-OR gate can be used as a basic comparator because its output is a 1 if the two input bits are not equal and a 0 if the input bits are equal.




In order to compare binary numbers containing two bits each, an additional exclusive OR gate is necessary. The two least significant bits (LSBs) of the two numbers are compared by gate $G_1$. In addition, the two most significant bits (MSBs) are compared by gate $G_2$, as shown in Figure below. If the two numbers are equal, their corresponding bits are the same, and the output of each exclusive-OR gate is a 0. If the corresponding sets of bits are not equal. a 1 occurs on that exclusive-OR gate output. In order to produce a single output indicating an equality or inequality of two numbers, two inverters and an AND gate can be used,



**Example:**

Apply each of the following sets of binary numbers to the comparator inputs in Figure 6–21, and determine the output by following the logic levels through the circuit.

(a) 10 and 10 (b) 11 and 10

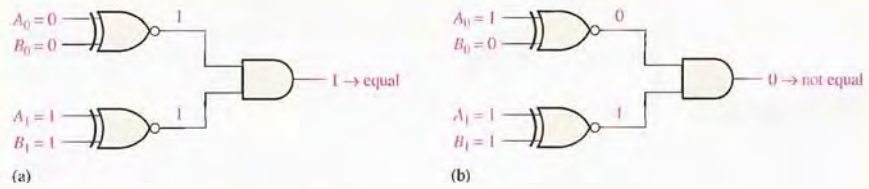



FIGURE 6–21

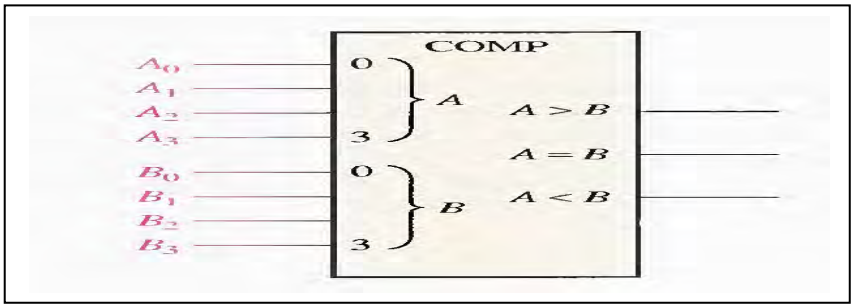
*Solution* (a) The output is 1 for inputs 10 and 10. as shown in Figure 6–21(a).

(b) The output is 0 for inputs 11 and 10, as shown in Figure 6–21(b).

**Inequality**

In addition to the equality output, many IC comparators provide additional outputs that indicate which of the two binary numbers being compared is the larger. That is, there is an output that indicates when number A is greater than number B (A > B) and an output that indicates when number A is less than number B (A < B), as shown in the logic symbol for a 4-bit comparator.

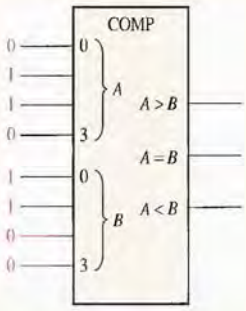
.




**Example:**

Determine the $A = B$, $A > B$, and $A < B$ outputs for the input numbers shown on the comparator in Figure 6–23.

FIGURE 6–23




*Solution* The number on the $A$ inputs is 0110 and the number on the $B$ inputs is 0011. The $A > B$ output is HIGH and the other outputs are LOW.

# CODE CONVERTERS

## BCD-to-Binary Conversion

One method of BCD-to-binary code conversion uses adder circuits. The basic conversion process is as follows:

1. The value, or weight, of each bit in the BCD number have represented by a binary number.
2. All of the binary representations of the weights of bits that are 1 s in the BCD number have added.
3. The result of this addition is the binary equivalent of the BCD number.

The binary numbers representing the weights of the BCD bits have summed to produce the total binary number.

| | Tens Digit | | | | Units Digit | | | |
|---|---|---|---|---|---|---|---|---|
| Weight: | 80 | 40 | 20 | 10 | 8 | 4 | 2 | 1 |
| Bit designation: | $B_3$ | $B_2$ | $B_1$ | $B_0$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |

Example:

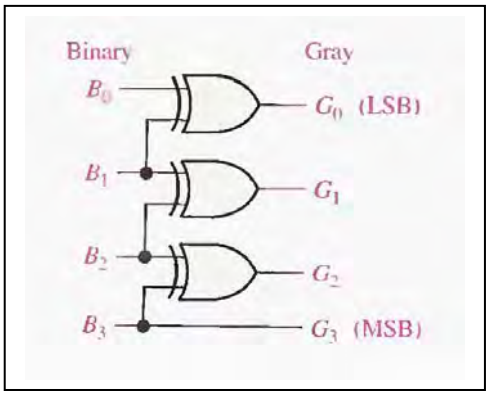
Convert the BCD numbers 00100111 (decimal 27) and 10011000 (decimal 98) to binary.

*Solution* Write the binary representations of the weights of all 1s appearing in the numbers, and then add them together.

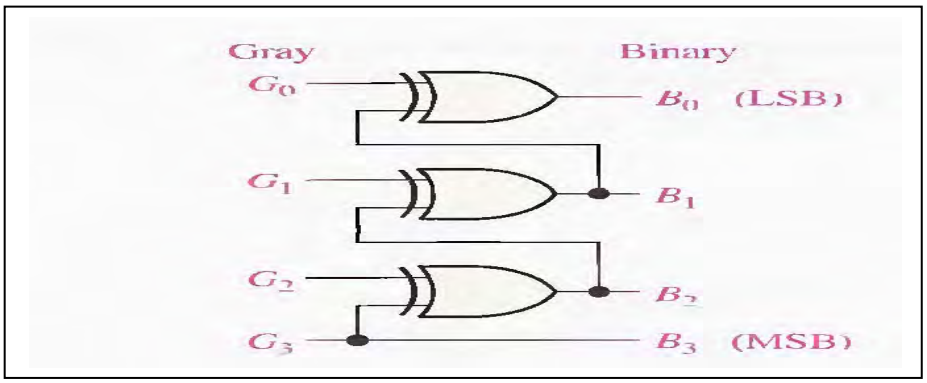
```
80 40 20 10 8 4 2 1
0  0  1  0  0 1 1 1
                    0000001   1
                    0000010   2
                    0000100   4
                  + 0010100  20
                    0011011   Binary number for decimal 27

80 40 20 10 8 4 2 1
1  0  0  1  1 0 0 0
                    0001000   8
                    0001010  10
                  + 1010000  80
                    1100010   Binary number for decimal 98
```

**Binary-to-Gray and Gray-to-Binary Conversion**

Figure bellow shows a 4-bit binary-to-Gray code converter.




In addition, Figure bellow illustrates a 4-bit Gray-to-binary converter.




Example:

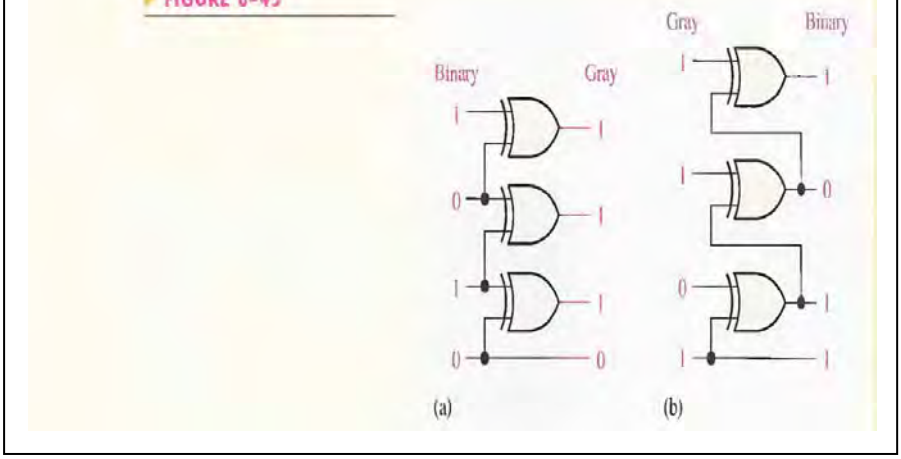
(a) Convert the binary number 0101 to Gray code with exclusive-OR gates.

(b) Convert the Gray code 1011 to binary with exclusive-OR gates.

*Solution* (a) $0101_2$ is 0111 Gray. See Figure 6–45(a).

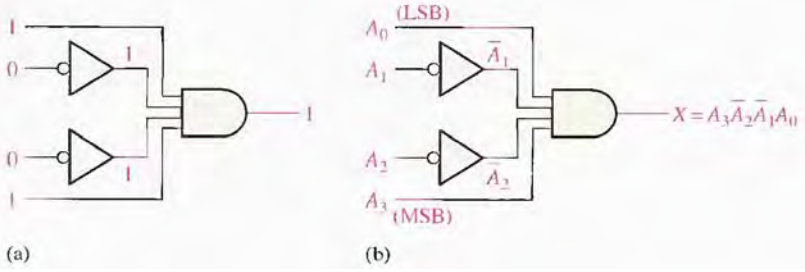
(b) 1011 Gray is $1101_2$. See Figure 6–45(b).

FIGURE 6-45



### The Basic Binary Decoder

Suppose you need to determine when a binary 1001 occurs on the inputs of a digital circuit. An AND gate can be used as the basic decoding element because it produces a HIGH output only when all of its inputs are HIGH. Therefore, you must make sure that all of the inputs to the AND gate are HIGH when the binary number 1001 occurs; this can be done by inverting the two middle bits (the 0s), as shown in Figure 6–26.




# ENCODER

An **encoder** is a combinational logic circuit that essentially performs a "reverse" decoder function. An encoder accepts an active level on one of its inputs representing digit, such as a decimal or octal digit, and converts it to a coded output, such as BCD or binary. Encoders can also be devised to encode various symbols and alphabetic characters. The process of converting from familiar symbols or numbers to a coded format is called *encoding*.

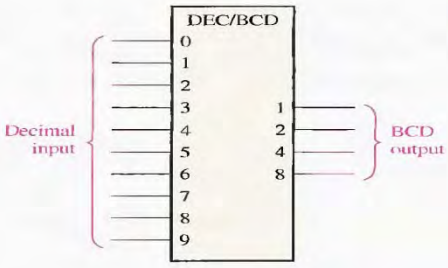



FIGURE 6–37
Logic symbol for a decimal-to-BCD encoder.

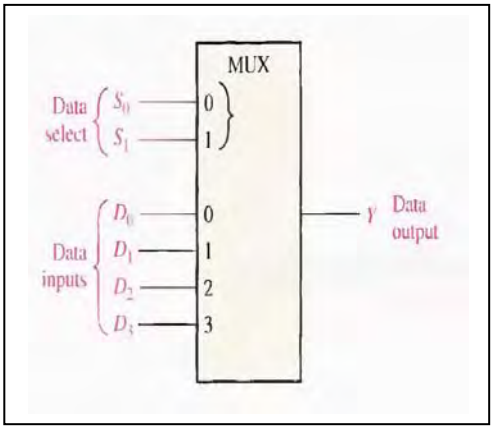
TABLE 6–6

| 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 |

## MULTIPLEXERS (DATA SELECTORS)

Logic symbol for a 4-input multiplexer (MUX) is shown in Figure bellow. Notice that there are two data-select lines because with two select bits. Any one of the four data input lines have selected.

2-bit code on the data-select (S) inputs will allow the data on the selected data input to pass through to the data output. If a binary 0 ($S_1$ = 0 and So = 0) is applied to the data-select lines, the data on input $D_0$ appear on the data-output line. If a binary 1 ($S_1$ = 0 and $S_0$ = 1) is applied to the data-select lines, the data on input $D_1$ appear on the data output. If a binary 2 ($S_1$ = 1 and So = 0) is applied, the data on $D_2$ appear on the output. If a binary 3 ($S_1$ = 1 and So = 1) is applied, the data on $D_3$




| DATA-SELECT INPUTS $S_1$ | $S_0$ | INPUT SELECTED |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

The data output is equal to $D_0$ only if $S_1 = 0$ and $S_0 = 0$: $Y = D_0\bar{S}_1\bar{S}_0$.

The data output is equal to $D_1$ only if $S_1 = 0$ and $S_0 = 1$: $Y = D_1\bar{S}_1S_0$.

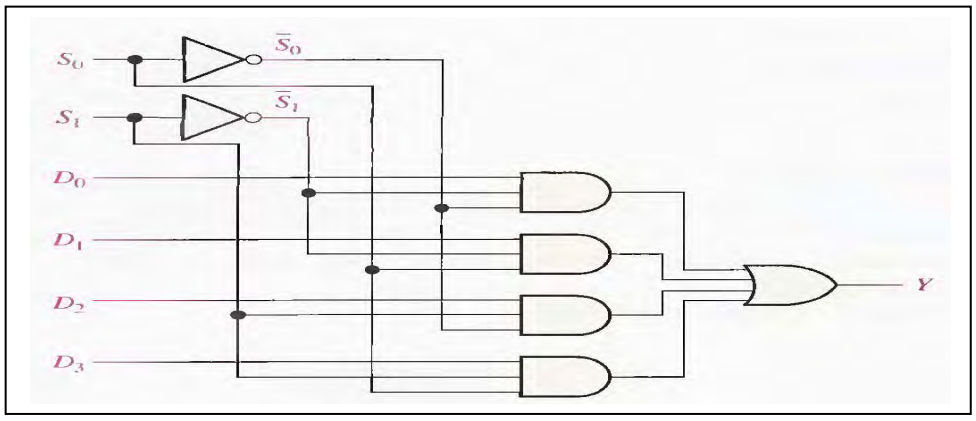
The data output is equal to $D_2$ only if $S_1 = 1$ and $S_0 = 0$: $Y = D_2S_1\bar{S}_0$.

The data output is equal to $D_3$ only if $S_1 = 1$ and $S_0 = 1$: $Y = D_3S_1S_0$.

When these terms are OR, the total expression for the data output is

$$Y = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1S_0 + D_2S_1\bar{S}_0 + D_3S_1S_0$$

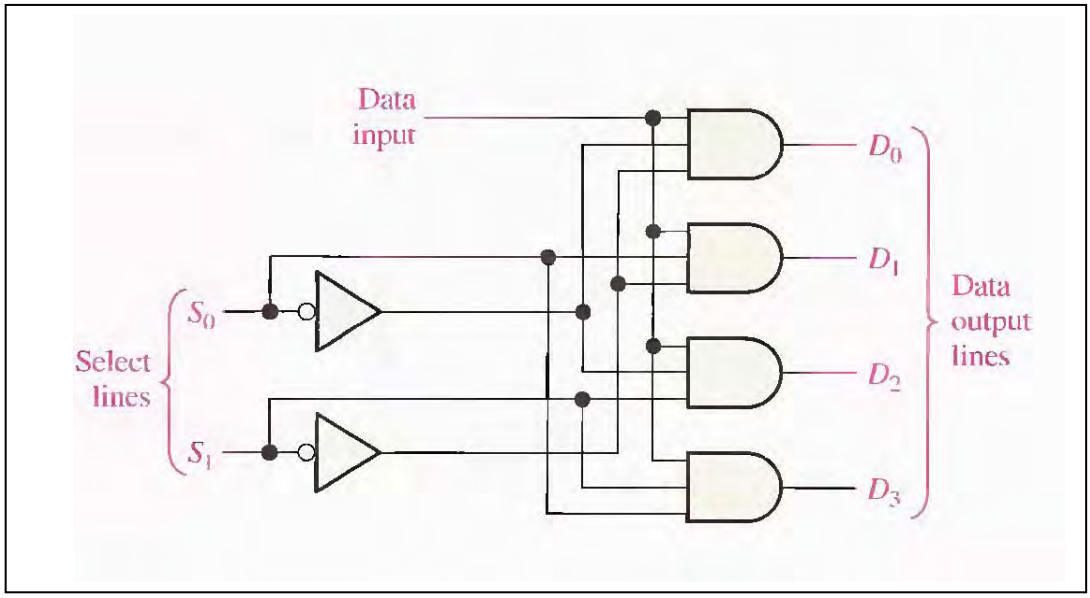
The implementation of this equation requires four 3-input AND gates, a 4-input OR gate, and two inverters to generate the complements of $S_1$and $S_2$




**DEMULTIPLEXERS**

A demultiplexer (DEMUX), basically reverses the multiplexing function. It takes digital information from one line and distributes it to a given number of output lines. For this reason, the demultiplexer has known as a data distributor.

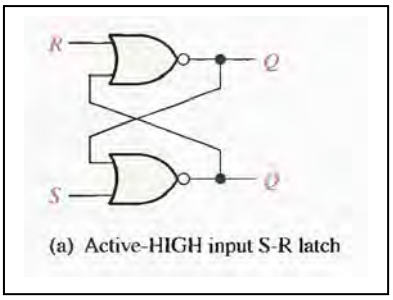
In Figure shows a l-line-to-4-line demultiplexer (DEMUX) circuit. The data-input line goes to all of the AND gates. The two data-select lines enable only one gate at a time, and the data appearing on the data-input line will pass through the selected gate to the associated data-output line.



# CHAPTER SIX/ Flip Flop

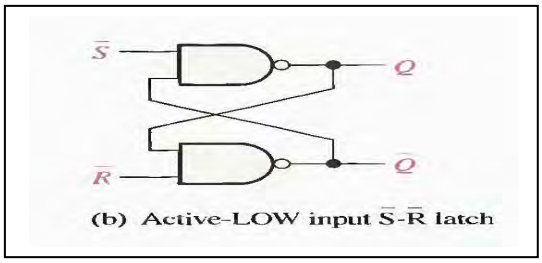
## The S-R (SET-RESET) Latch

An active-**HIGH** input S-R (SET-RESET) latch is formed with two cross-coupled NOR gates, as shown in Figure



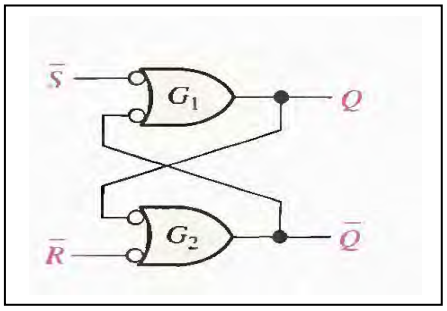
(a) Active-HIGH input S-R latch

An active-**LOW** $\overline{S}$-$\overline{R}$ input latch has formed with two cross-coupled NAND gates, as shown in figure



(b) Active-LOW input $\bar{S}$-$\bar{R}$ latch

We start explain the operation by using Negative OR gates as shown in figure



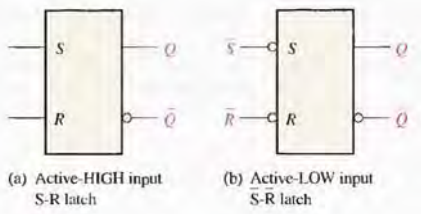

two inputs, $\bar{S}$ and $\bar{R}$, and two outputs, $Q$ and $\bar{Q}$. Let's start by assuming that both inputs and the $Q$ output are HIGH. Since the $Q$ output is connected back to an input of gate $G_2$, and the $\bar{R}$ input is HIGH, the output of $G_2$ must be LOW. This LOW output is coupled back to an input of gate $G_1$, ensuring that its output is HIGH.

When the $Q$ output is HIGH, the latch is in the SET state. It will remain in this state indefinitely until a LOW is temporarily applied to the $\bar{R}$ input. With a LOW on the $\bar{R}$ input and a HIGH on $\bar{S}$, the output of gate $G_2$ is forced HIGH. This HIGH on the $\bar{Q}$ output is coupled back to an input of $G_1$, and since the $\bar{S}$ input is HIGH, the output of $G_1$ goes LOW. This LOW on the $Q$ output is then coupled back to an input of $G_2$, ensuring that the $\bar{Q}$ output remains HIGH even when the LOW on the $\bar{R}$ input is removed. When the $Q$ output is LOW, the latch is in the RESET state. Now the latch remains indefinitely in the RESET state until a LOW is applied to the $\bar{S}$ input.

The truth table as shown in figure below

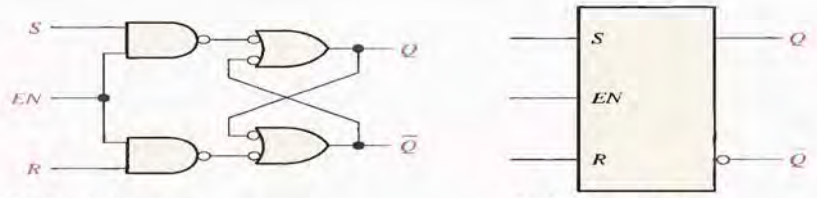
| INPUTS $\bar{S}$ | INPUTS $\bar{R}$ | OUTPUTS $Q$ | OUTPUTS $\bar{Q}$ | COMMENTS |
|---|---|---|---|---|
| 1 | 1 | NC | NC | No change. Latch remains in present state. |
| 0 | 1 | 1 | 0 | Latch SET. |
| 1 | 0 | 0 | 1 | Latch RESET. |
| 0 | 0 | 1 | 1 | Invalid condition |

Logic symbols for both the active-HIGH input and the active-LOW input latches are shown in figure




(a) Active-HIGH input S-R latch (b) Active-LOW input $\bar{S}$-$\bar{R}$ latch

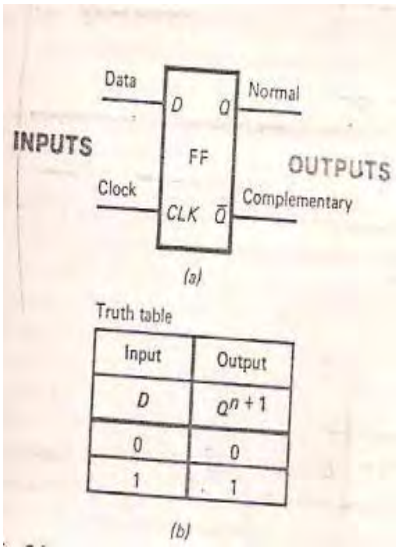
A gated latch requires an enable input, *EN* (*G* is also used to designate an enable input). The logic diagram and logic symbol for a gated S-R latch are shown in Figure 7–8. The *S* and *R* inputs control the state to which the latch will go when a HIGH level is applied to the *EN* input. The latch will not change until *EN* is HIGH; but as long as it remains HIGH, the output is controlled by the state of the *S* and *R* inputs. In this circuit, the invalid state occurs when both *S* and *R* are simultaneously HIGH.




(a) Logic diagram (b) Logic symbol

**D Flip Flop:**

Logic symbol for D- flip-flop has shown in figure
It has only one data input (D) and clock input (CLK) the output are labeled Q and $\overline{Q}$ the D flip-flop is often called delay flip flop because the data (0 or 1) at input D is delayed one clock pulse from getting to the output Q
From truth table Notice that output Q fallows input D after one clock pulse ($Q^{n+1}$)



We can make D flip flop from a clocked R-S flip flop by adding inverter as shown

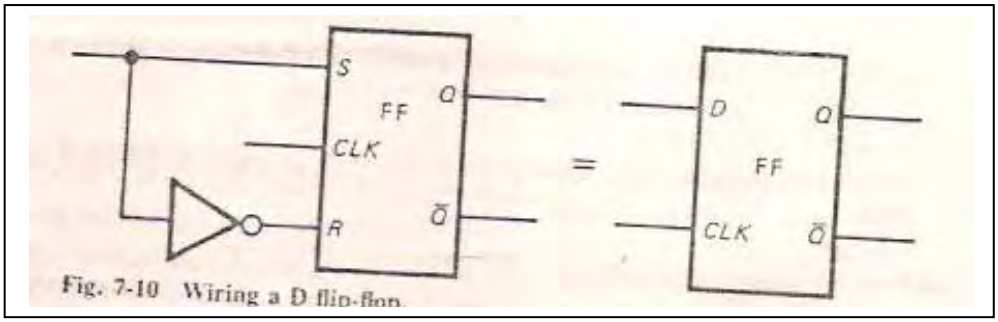
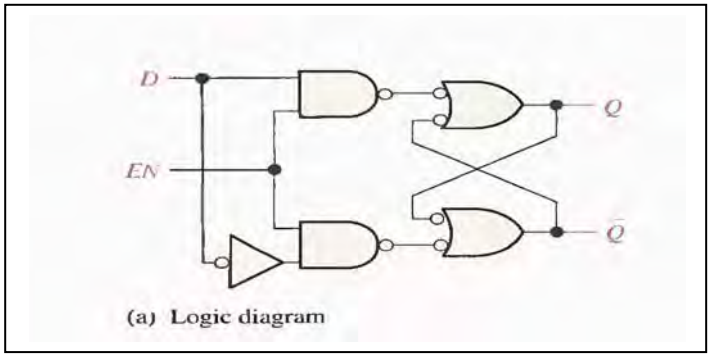



Fig. 7-10 Wiring a D flip-flop

Logic diagram of D flip flop As shown below

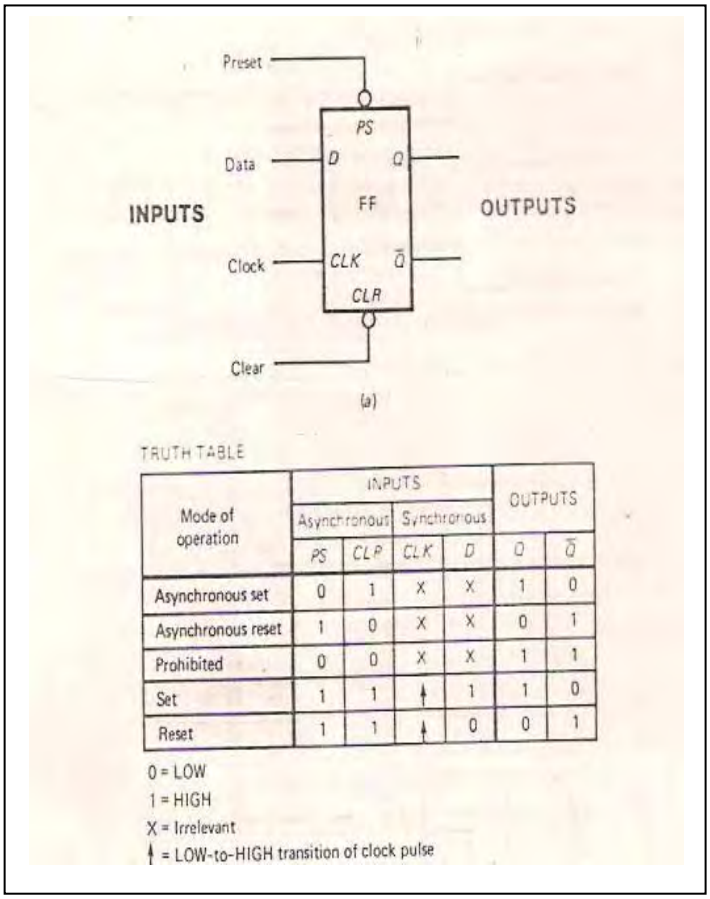



(a) Logic diagram

Always we use the D flip flop contained in IC as shown in figure

We have two extra input
[PS(preset) and CLR (clear) ]
PS input sets o/p Q=1 when enabled by logic 0
The CLR i/p clear o/p to 0 when enabled by logic 0
The PS and CLR override the D and CLK I/P
As shown in truth table

When we have the PS and CLR the flip flop operate as Asynchronous (not synchronous)
If the flip flop disable the PS and CLR therefore in synchronous operation and can be set and reset by D and CLK input this can be see from the last two line in truth table.




(a)

TRUTH TABLE

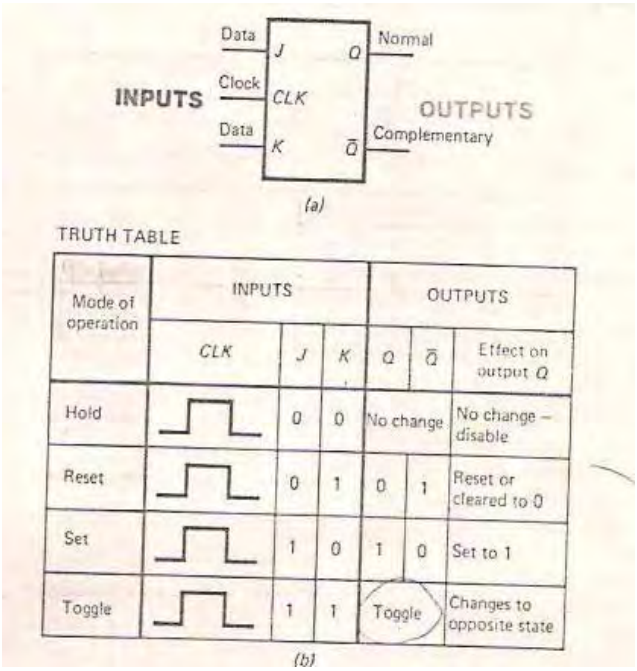
| Mode of operation | INPUTS | | | | OUTPUTS | |
|---|---|---|---|---|---|---|
| | Asynchronous | | Synchronous | | | |
| | PS | CLR | CLK | D | Q | Q̄ |
| Asynchronous set | 0 | 1 | X | X | 1 | 0 |
| Asynchronous reset | 1 | 0 | X | X | 0 | 1 |
| Prohibited | 0 | 0 | X | X | 1 | 1 |
| Set | 1 | 1 | ↑ | 1 | 1 | 0 |
| Reset | 1 | 1 | ↑ | 0 | 0 | 1 |

0 = LOW
1 = HIGH
X = Irrelevant
↑ = LOW-to-HIGH transition of clock pulse

## J-K Flip Flop

Most widely used and universal Flip Flop
The I/P label J and K are data input
CLK is the clock input
Q and $\overline{Q}$ are the normal and complementary o/p
From truth table we see

If J and K =0 the flip flop in hold mode the data input not effect on output then the output in hold on last data present line 1 in truth table
Line 2 and 3 is the reset and set condition for Q output
Line 4 toggle position of J-K flip flop when both data J=K=1 therefore the Q O/P will repeat clock pulses causes turn off-on-off-on and so on

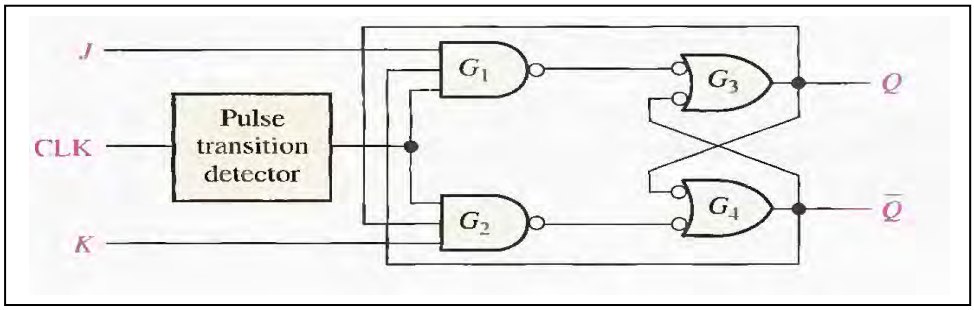



TRUTH TABLE

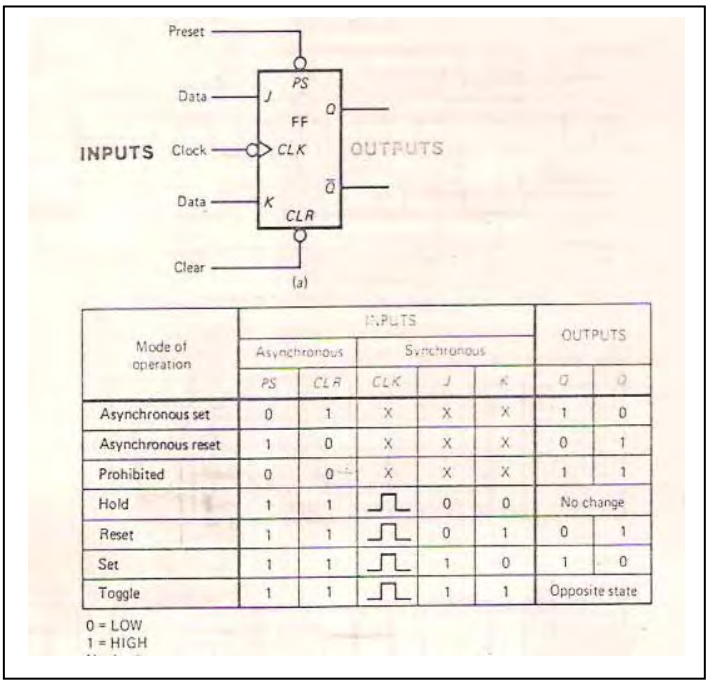
| Mode of operation | INPUTS | | | OUTPUTS | | |
|---|---|---|---|---|---|---|
| | CLK | J | K | Q | $\overline{Q}$ | Effect on output Q |
| Hold | ⊓ | 0 | 0 | No change | | No change – disable |
| Reset | ⊓ | 0 | 1 | 0 | 1 | Reset or cleared to 0 |
| Set | ⊓ | 1 | 0 | 1 | 0 | Set to 1 |
| Toggle | ⊓ | 1 | 1 | Toggle | | Changes to opposite state |

(b)

**The logic diagram of J-K flip-flop as shown**




The commercial logic symbols for J_K flip flop as shown
We see asynchronous input (preset and clear) and the synchronous is the clock input



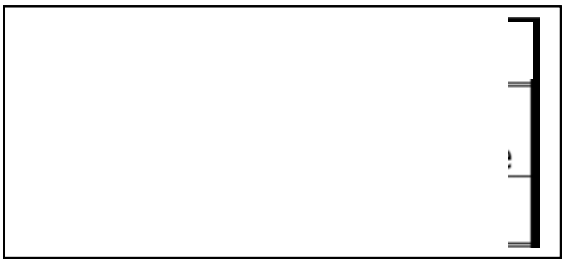

| Mode of operation | INPUTS | | | | | OUTPUTS | |
|---|---|---|---|---|---|---|---|
| | Asynchronous | | Synchronous | | | | |
| | PS | CLR | CLK | J | K | Q | $\overline{Q}$ |
| Asynchronous set | 0 | 1 | X | X | X | 1 | 0 |
| Asynchronous reset | 1 | 0 | X | X | X | 0 | 1 |
| Prohibited | 0 | 0 | X | X | X | 1 | 1 |
| Hold | 1 | 1 | ⊓ | 0 | 0 | No change | |
| Reset | 1 | 1 | ⊓ | 0 | 1 | 0 | 1 |
| Set | 1 | 1 | ⊓ | 1 | 0 | 1 | 0 |
| Toggle | 1 | 1 | ⊓ | 1 | 1 | Opposite state | |

0 = LOW
1 = HIGH

**T –Flip Flop**

The logic symbols for T- flip flop

The output is toggle if the input data T =1 with clock (reverse of previous state) as shown in truth table

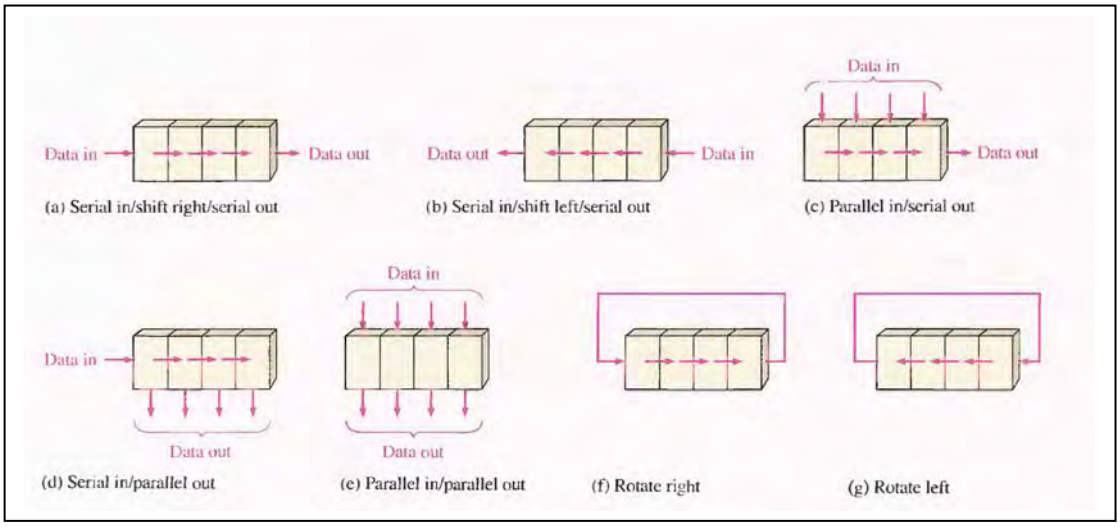


Also can be used the J-K flip flop to represent T flip flop by short J and K = 1

# CHAPTER SEVEN
# SHIFT REGISTER

BASIC SHIFT REGISTER FUNCTIONS

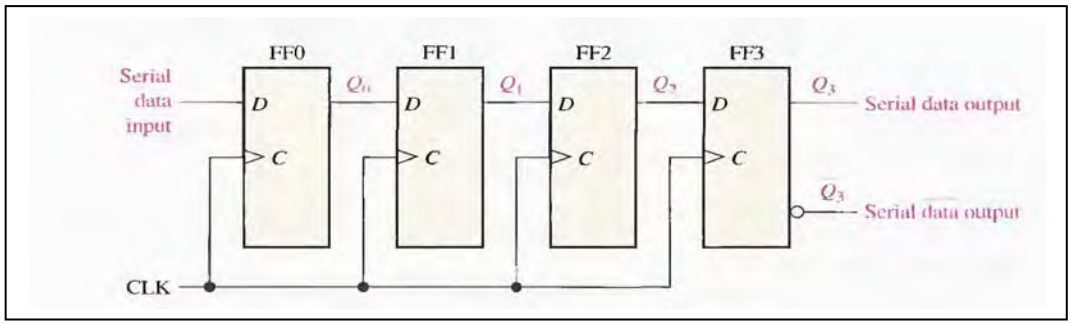
A register is a digital circuit with two basic functions: data storage and data movement. We have many type of shift register as shown




**Serial IN/Serial OUT SHIFT REGISTERS**

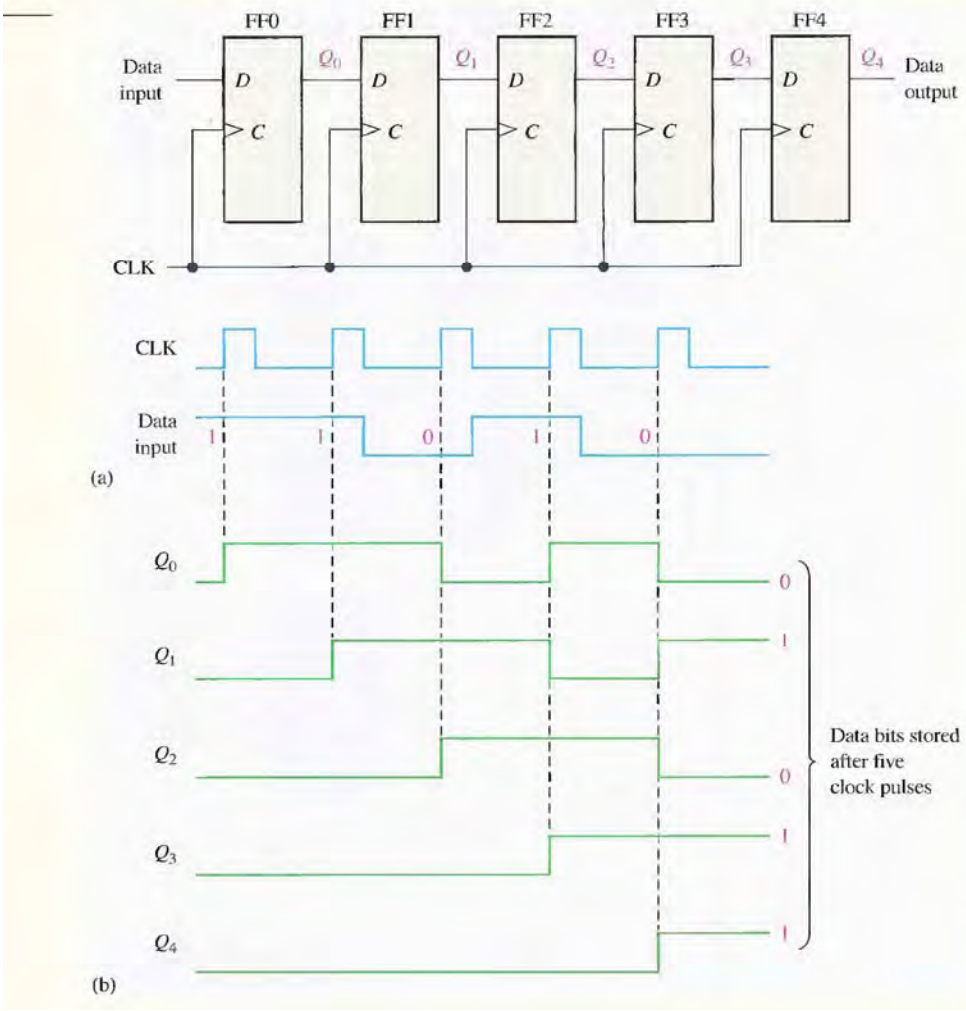
The serial in/serial out shift register accepts data serially-that is, one bit at a time on a single line. It produces the stored information on its output also in serial form.

Let first look at the serial entry of data into a typical shift register. Figure below shows a 4-bit device implemented with D flip-flops. With four stages, this register can store up to four bits of data.



Example:

Show the states of the 5-bit register in Figure 9–6(a) for the specified data input and clock waveforms. Assume that the register is initially cleared (all 0s).

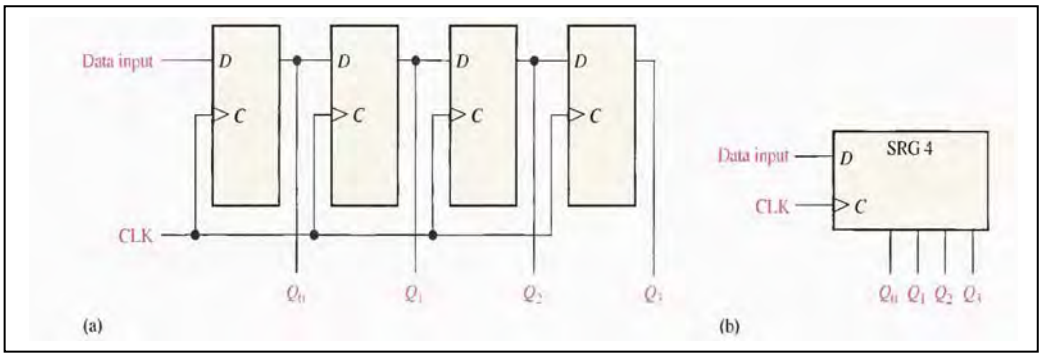



*Solution* The first data bit (1) is entered into the register on the first clock pulse and then shifted from left to right as the remaining bits are entered and shifted. The register contains $Q_4Q_3Q_2Q_1Q_0 = 11010$ after five clock pulses. See Figure 9–6(b).

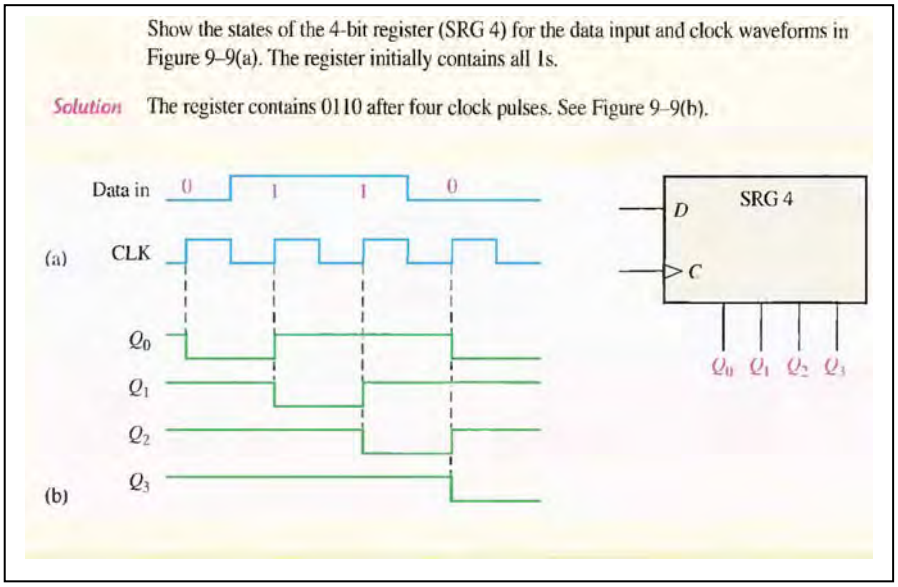
## SERIAL IN/PARALLEL OUT SHIFT REGISTERS

in the parallel output register. the output of each stage is available. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously, rather than on a bit-by-bit basis as with the serial

Figure below shows a 4-bit serial in/parallel out shift register and its logic block symbol




Example :




## PARALLEL IN/SERIAL OUT SHIFT REGISTERS

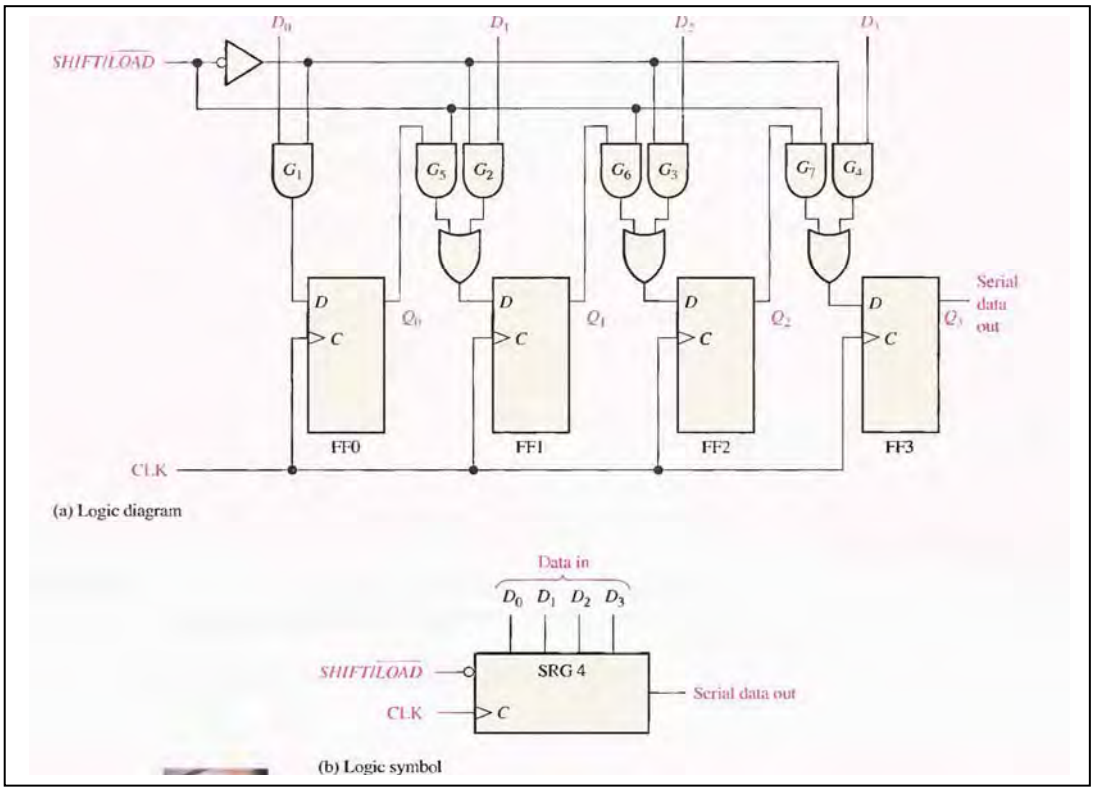
For parallel data, multiple bits have transferred at one time.

Figure below illustrates a 4-bit parallel in/serial out shift register and a typical logic symbol.

Notice that there are four data-input lines, $D_0$, $D_1$, $D_2$, and $D_3$,and SHIFT / $\overline{\text{LOAD}}$ input, which allows four bits of data to load in parallel into the register.

When SHIFT/$\overline{\text{LOAD}}$ is LOW, gates G 1 through G 4 are enabled, allowing each data bit to be applied to the D input of its respective flip-flop. When a clock pulse is applied, the flip-flops with D = 1will set and those with D = 0 will reset. Thereby storing all four bits simultaneously.

When SHIFT/$\overline{\text{LOAD}}$ is HIGH, gates $G_1$, through $G_4$ are disabled and gates G5; through G7 are enabled, allowing the data bits to shift right from one stage to the next. The OR gates allow either the normal shifting operation or the parallel data-entry operation, depending on which AND gates are enabled by the level on the SHIFT/$\overline{\text{LOAD}}$ input. Notice that FF0 has a single AND to disable the parallel input, $D_0$. It does not require an AND/OR arrangement because there is no serial data in.




Example

Show the data-output waveform for a 4-bit register with the parallel input data and the clock and *SHIFT*/$\overline{LOAD}$ waveforms given in Figure 9–13(a). Refer to Figure 9–12(a) for the logic diagram.

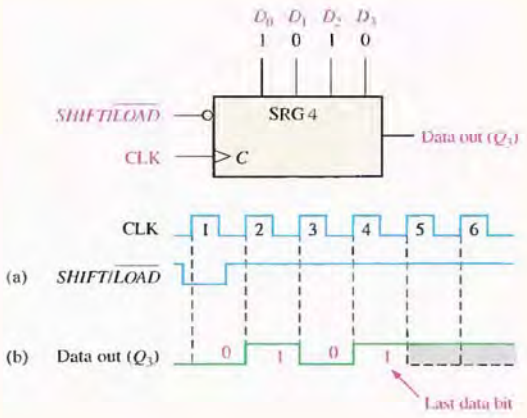
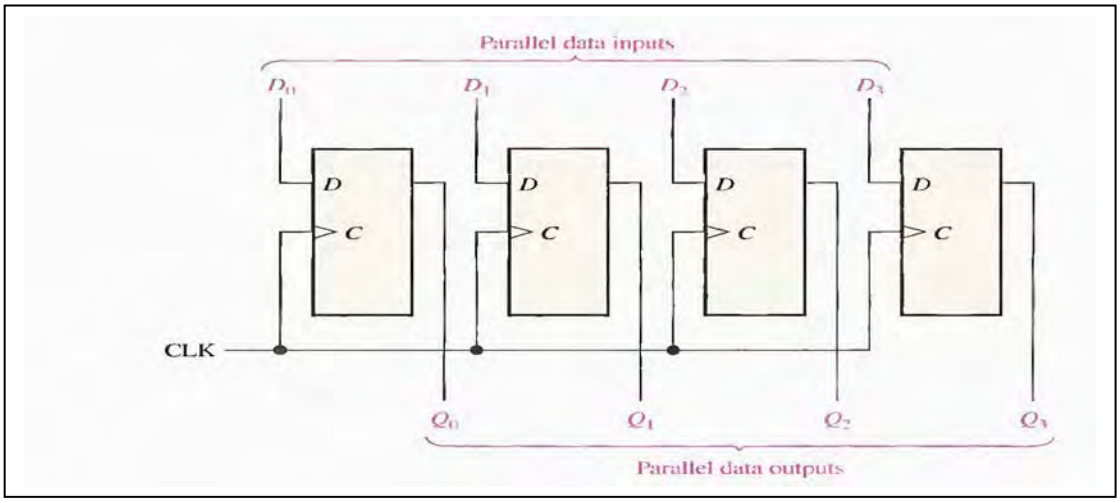



FIGURE 9–13

**Solution** On clock pulse 1, the parallel data ($D_0D_1D_2D_3 = 1010$) are loaded into the register, making $Q_3$ a 0. On clock pulse 2 the 1 from $Q_2$ is shifted onto $Q_3$; on clock pulse 3 the 0 is shifted onto $Q_3$; on clock pulse 4 the last data bit (1) is shifted onto $Q_3$; and on clock pulse 5, all data bits have been shifted out, and only 1s remain in the register (assuming the $D$ input remains a 1). See Figure 9–13(b).

## PARALLEL IN/PARALLEL OUT SHIFT REGISTERS

The parallel in/parallel out register, immediately following the simultaneous entry of all data bits, the bits appear on the parallel outputs.



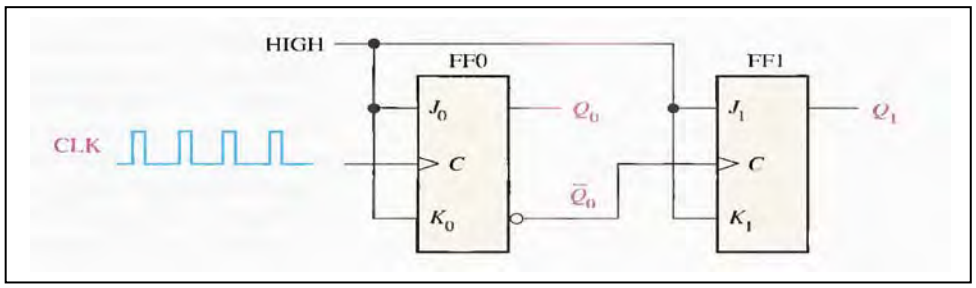
# CHAPTER EIGHT Counters

## ASYNCHRONOUS COUNTER OPERATION

The term asynchronous refers to events that do not have a fixed time relationship with each other and, generally, do not occur at the same time. An asynchronous counter is one in which the flip-flops (FF) within the counter do not change states at exactly the same time because they do not have a common clock pulse. A counter can have **$2^n$ states**, where **n** is the number of flip-flops.

## A 2-Bit Asynchronous Binary Counter

Figure below shows a 2-bit counter connected for asynchronous operation. Notice that the clock (CLK) has applied to the clock input (C) of only the first flop-flop, FF0, which is always the least significant bit (LSB). The second flip-flop, FF1, is triggered by the $Q_0$ out put of FF0. FFO changes state at the positive-going edge of each clock pulse. But FF1 changes only when triggered by a positive-going transition of the $Q_0$ output of FF0.




**The binary state sequence of 2 bit as shown in table below**

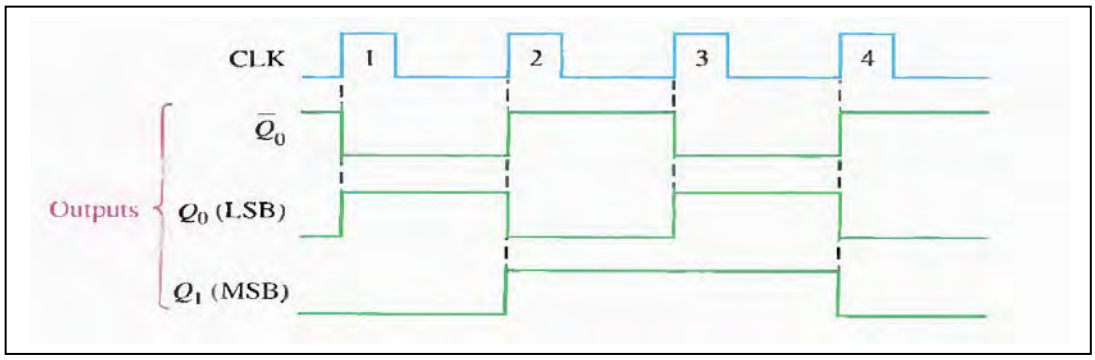
| CLOCK PULSE | $Q_1$ | $Q_0$ |
|---|---|---|
| Initially | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |
| 4 (recycles) | 0 | 0 |

Since it goes through a binary sequence, the counter in Figure 8-1 is a binary counter.
It actually counts the number of clock pulses up to three, and on the fourth pulse it recycles to its original state ($Q_0 = 0$, $Q_1 = 0$). The term recycle has commonly applied to counter operation; it refers to the transition of the counter from its final state back to its original state.

**The Timing Diagram**

The positive-going edge of CLKI (clock pulse]) causes the $Q_0$ output of FF0 to go HIGH, At the same time the $\overline{Q}_0$ output goes low. But it has no effect on FF1 because a positive -going transition must occur to trigger the flip-flop. After the

leading edge of CLK1, $Q_0 = 1$ and $Q_1 = 0$. The positive-going edge of CLK2 causes $Q_0$ to go LOW. Output $\overline{Q}_0$ goes HIGH and triggers FF1, causing $Q_1$ to go HIGH. After the leading edge of CLK2, $Q_0 = 0$ and $Q_1 = 1$. The positive-going edge of CLK3 causes $Q_0$ to go HIGH again. Output $\overline{Q}_0$ goes LOW and has no effect on FF1. Thus, after the leading edge of CLK3, $Q_0 = 1$ and $Q_1 = 1$. The positive-going edge of CLK4 causes $Q_0$ to go LOW, while $\overline{Q}_0$ goes HIGH and triggers FF1, causing $Q_1$ to go LOW. After the leading edge of CLK4, $Q_0 = 0$ and $Q_1 = 0$. The counter has now recycled to its original state (both flip-flops are RESET).



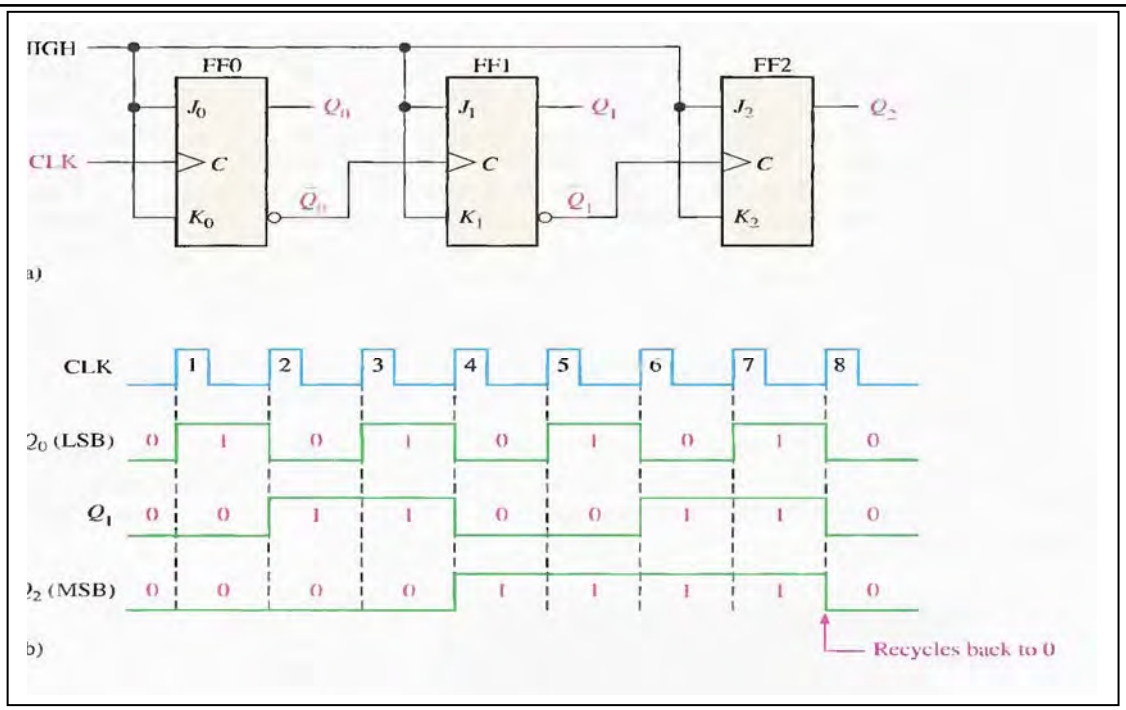

**A 3-Bit Asynchronous Binary Counter**

The state sequence for a 3-bit binary counter has listed in table below:

basic operation is the same as that of the 2-bit counter except that the 3-bit counter has eight states, due to its three flip-flops.

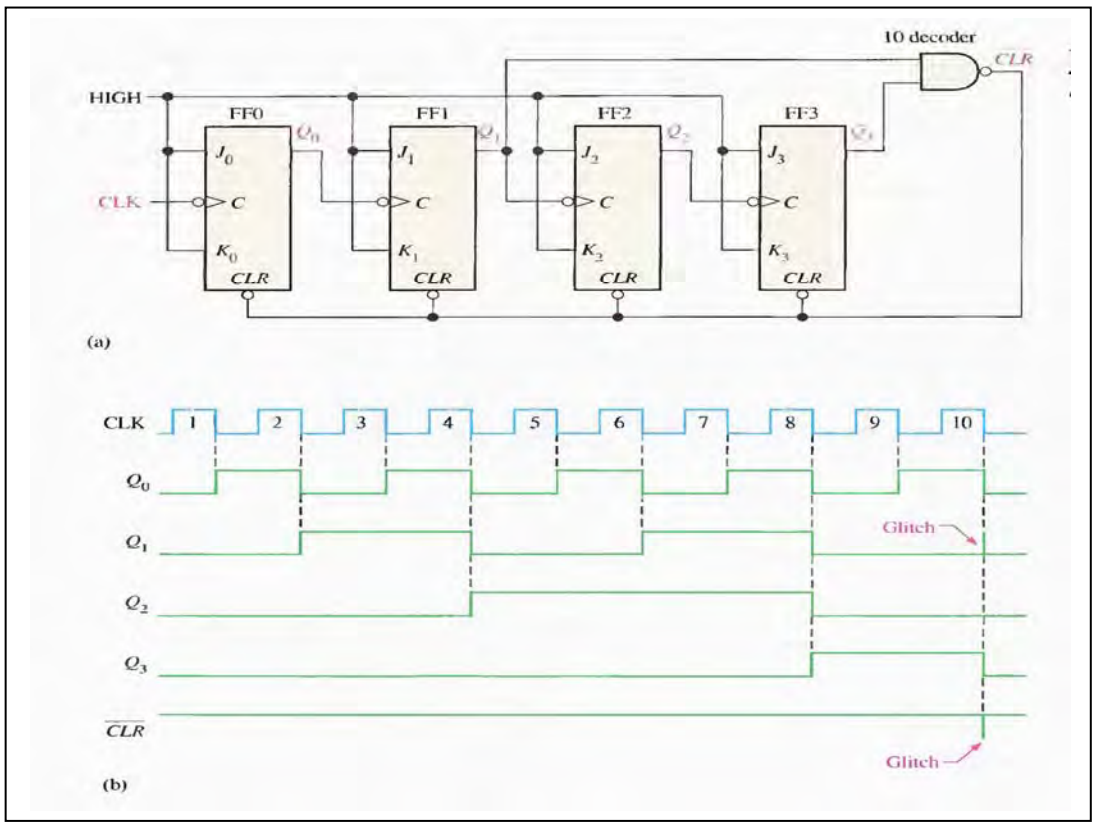
| CLOCK PULSE | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| Initially | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |
| 8 (recycles) | 0 | 0 | 0 |

Notice that the counter progresses through a binary count of zero through seven and then recycles to the zero state. This counter can be easily expanded for higher count, by connecting additional toggle flip-flops.




**Asynchronous Decade Counters**

One common modulus for counters with truncated sequences is ten (called MOD10). Counters with ten states in their sequence have called decade counters. A decade counter with a count sequence of zero (0000) through nine (1001) is a BCD decade counter because its ten-state sequence produces the BCD code. To obtain a truncated sequence, it is necessary to force the counter to recycle before going through all of its possible states. For example, the BCD decade counter must recycle back to the 0000 state after the 100l state. One way to make the counter recycle after the count of nine (1001) is to decode count ten (l010) with a NAND gate and connect the output of the NAND gate to the clear ( $\overline{CLR}$ ) inputs of the flip-flops, as shown



Example:

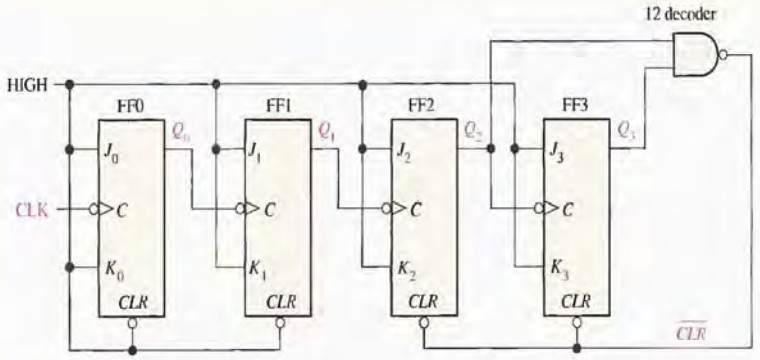
Show how an asynchronous counter can be implemented having a modulus of twelve with a straight binary sequence from 0000 through 1011.

*Solution* Since three flip-flops can produce a maximum of eight states, four flip-flops are required to produce any modulus greater than eight but less than or equal to sixteen.

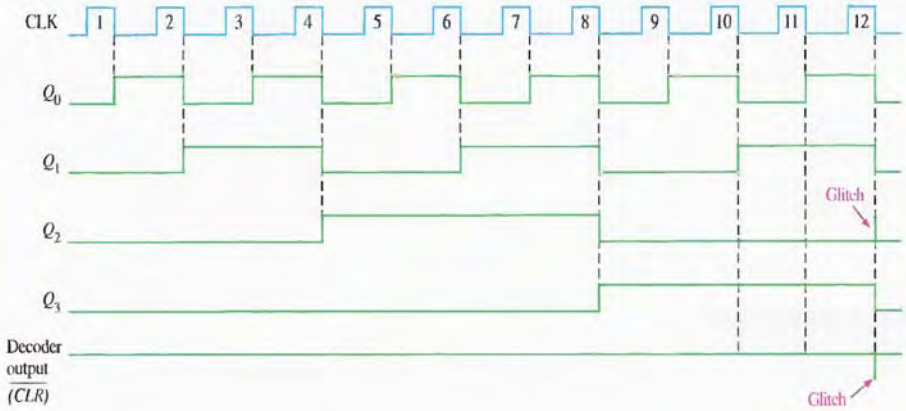
When the counter gets to its last state, 1011, it must recycle back to 0000 rather than going to its normal next state of 1100, as illustrated in the following sequence chart:

| $Q_3$ | $Q_2$ | $Q_1$ | $Q_0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ← |
| . | . | . | . | |
| . | . | . | . | Recycles |
| . | . | . | . | |
| 1 | 0 | 1 | 1 | ← |
| 1 | 1 | 0 | 0 | ← Normal next state |

Observe that $Q_0$ and $Q_1$ both go to 0 anyway, but $Q_2$ and $Q_3$ must be forced to 0 on the twelfth clock pulse. Figure 8–7(a) shows the modulus-12 counter. The NAND gate partially decodes count twelve (1100) and resets flip-flop 2 and flip-flop 3. Thus, on the twelfth clock pulse, the counter is forced to recycle from count eleven to count zero, as shown in the timing diagram of Figure 8–7(b). (It is in count twelve for only a few nanoseconds before it is reset by the glitch on $\overline{CLR}$.)




(a)



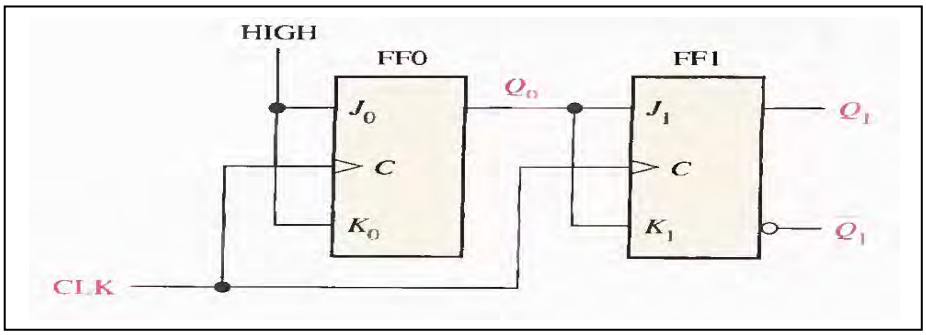

(b)

## SYNCHRONOUS COUNTER OPERATION

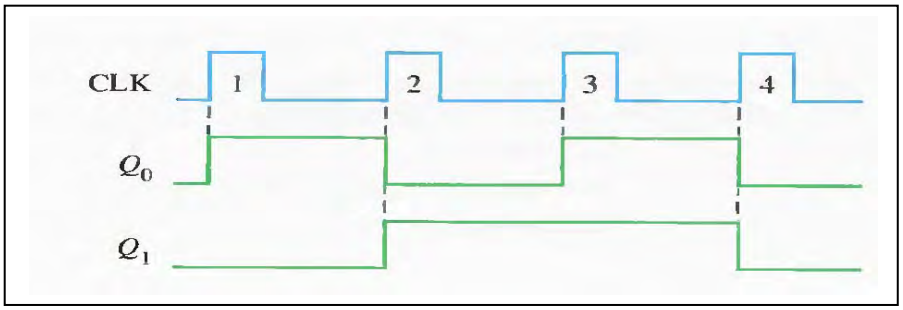
The term **synchronous** refers to events that have a fixed time relationship with each other. A **synchronous** counter is one in which all the flip-flops in the counter have clocked at the same time by a common clock pulse.

### A 2-Bit Synchronous Binary Counter

Figure below shows a 2-bit synchronous binary counter. Notice that an arrangement different from that for the asynchronous counter must be used for the $J_1$ and $K_1$ inputs of FF1 in order to achieve a binary sequence.



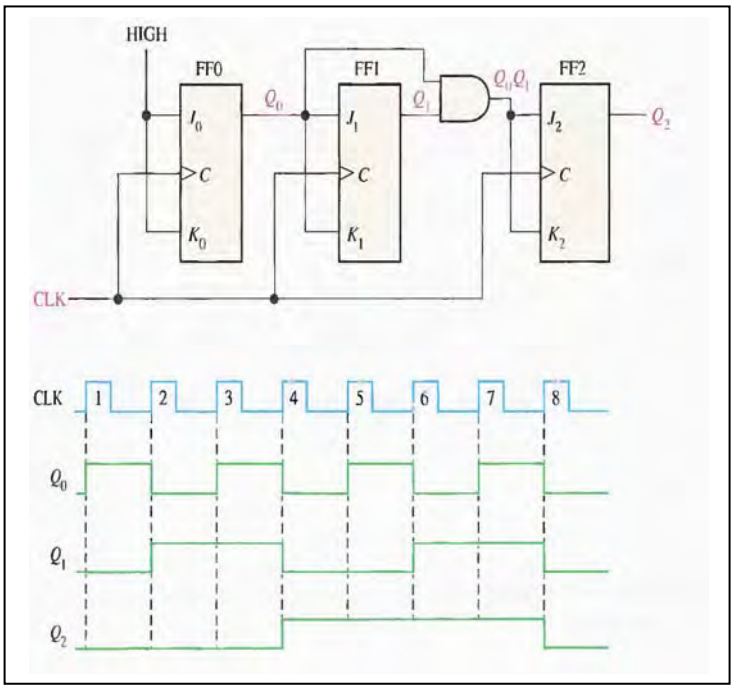

The operation of this synchronous counter is as follows:
First, assume that the counter is initially in the binary 0 state: that is. Both flip-flops are RESET. When the positive edge of the first clock pulse is applied, FF0 will toggle and $Q_0$ will therefore go HIGH. After CLK1, $Q_0$ = 1and $Q_1$ = 0. When the leading edge of CLK2 occurs, FF0 will toggle and $Q_0$ will go LOW. Since FF1 has a HIGH ($Q_0$ = 1) on its J1, and KI inputs at the triggering edge of this clock pulse, the flip-flop toggles and $Q_1$ goes HIGH. Thus, after CLK2, $Q_0$ = 0 and $Q_1$ = 1(which is a binary 2 state).
When the leading edge of CLK3 occurs. FF0 again toggles to the SET state ($Q_0$ = 1), and FF1 remains SET ($Q_1$ =1) because its $J_1$ and $K_1$ inputs are both LOW ($Q_0$ = 0). After this triggering edge, $Q_0$ = 1 and $Q_1$ = 1 (which is a binary 3 state).
Finally, at the leading edge of CLK4, $Q_0$ and $Q_1$ go LOW because they both have a toggle condition on their J and K inputs.



## A 3-Bit Synchronous Binary Counter

A 3-bit synchronous binary counter is shown in Figure below its timing diagram is shown in Figure You can understand this counter operation by examining its sequence of states as shown in table

| CLOCK PULSE | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| Initially | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |
| 8 (recycles) | 0 | 0 | 0 |

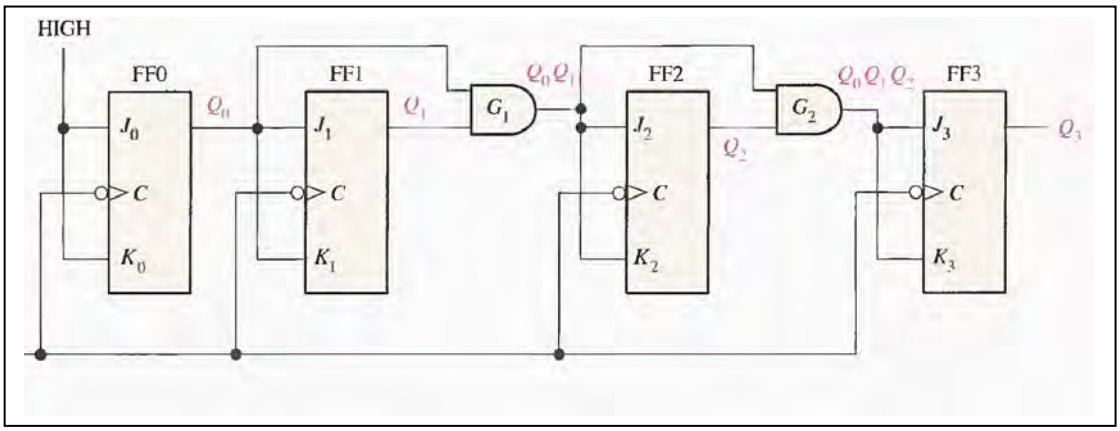



First, let's look at $Q_0$. Notice that $Q_0$ changes on each clock pulse as the counter progresses from its original state to its final state and then back to its original state.
To produce this operation, FF0 must be held in the toggle mode by constant HIGH on its $J_0$ and $K_0$ inputs. Notice that $Q_1$ goes to the opposite state following each time $Q_0$ is a 1. This change occurs at CLK2, CLK4, CLK6, and CLK8. The CLK8 pulse causes the counter to recycle. To produce this operation, $Q_0$ is connected to the $J_1$ and $K_1$ inputs of FF1. When $Q_0$ is a 1 and a clock pulse occurs, FF1 is in the toggle mode and therefore changes state. The other times, when $Q_0$ is a 0, FF1 is in the no-change mode and remains in its present state. Next, let's see how FF2 is made to change at the proper times according to the binary sequence. Notice that both times $Q_2$ changes state. it is preceded by the unique condition in which both $Q_0$ and $Q_1$ are HIGH. This condition is detected by the AND gate and applied to the $J_2$ and $K_2$ inputs of FF2. Whenever both $Q_0$ and $Q_1$ are HIGH, the output of the AND gate makes the $J_2$ and $K_2$ inputs of FF2 HIGH, and FF2 toggles on the following clock pulse. At all other times, the $J_2$ and $K_2$ inputs of FF2 are held LOW by the AND gate output, andFF2 does not change state.

## A 4-Bit Synchronous Binary Counter

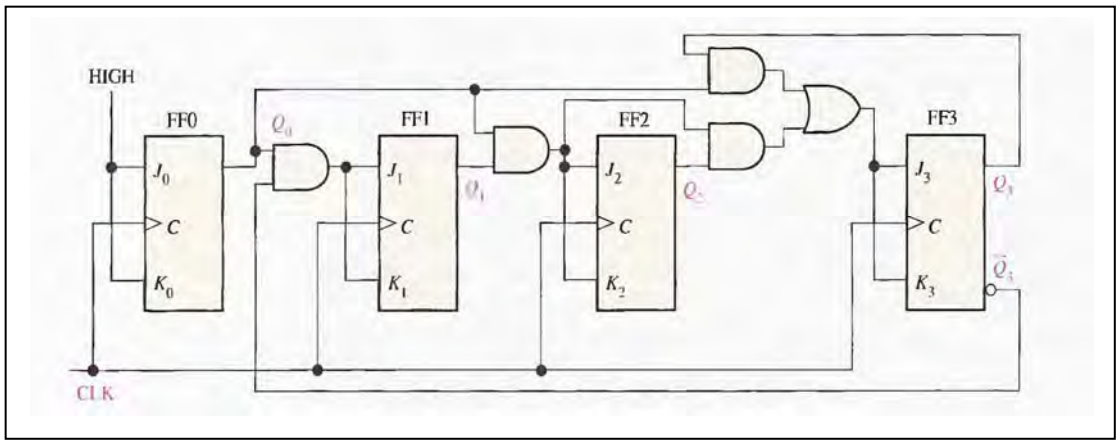
Figure below shows a 4-bit synchronous binary counter,
This particular counter is implemented with negative edge-triggered flip-

flops. The reasoning behind the $J$ and $K$ input control for the first three flip-flops is the same as previously discussed for the 3-bit counter. The fourth stage, FF3, changes only twice in the sequence. Notice that both of these transitions occur following the times that $Q_0$, $Q_1$, and $Q_2$ are all HIGH. This condition is decoded by AND gate $G_2$ so that when a clock pulse occurs, FF3 will change state. For all other times the $J_3$ and $K_3$ inputs of FF3 are LOW, and it is in a no-change condition.




## A 4-Bit Synchronous Decade Counter

As you know, a BCD decade counter exhibits a truncated binary sequence and goes from 0000 through the 1001 state. Rather than going from the 1001 state to the 1010 state, it recycles to the 0000 state. A synchronous BCD decade counter is shown in Figure below.




First, notice that FF0 ($Q_0$) toggles on each clock pulse, so the logic equation for its $J_0$ and $K_0$ inputs is **$J_0=K_0=1$**
This equation is implemented by connecting $J_0$ and $K_0$ to a constant HIGH level.
Next, notice in Table below that FF1 ($Q_1$) changes on the next clock pulse each time $Q_0 = 1$ and $Q_3 = 0$, so the logic equation for the $J_1$ and $K_1$ inputs is

$$J_1 = K_1 = Q_0\overline{Q}_3$$

This equation is implemented by ANDing $Q_0$ and $\overline{Q}_1$ and connecting the gate output to the $J_1$ and $K_1$ inputs of FF1.

Flip-flop 2 ($Q_2$) changes on the next clock pulse each time both $Q_0 = 1$ and $Q_1 = 1$. This requires an input logic equation as follows:

$$J_2 = K_2 = Q_0Q_1$$

This equation is implemented by ANDing $Q_0$ and $Q_1$ and connecting the gate output to the $J_2$ and $K_2$ inputs of FF2.

Finally, FF3 ($Q_3$) changes to the opposite state on the next clock pulse each time $Q_0 = 1$, $Q_1 = 1$, and $Q_2 = 1$ (state 7), or when $Q_0 = 1$ and $Q_3 = 1$ (state 9). The equation for this is as follows:

$$J_3 = K_3 = Q_0Q_1Q_2 + Q_0Q_3$$

This function is implemented with the AND/OR logic connected to the $J_3$ and $K_3$ inputs of FF3 as shown in the logic diagram

States of a BCD decade counter .table

| CLOCK PULSE | $Q_3$ | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|---|
| Initially | 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 |

## UP/DOWN SYNCHRONOUS COUNTERS

In general, most up/down counters can be reversed at any point in their sequence. Table below shows the complete up/down sequence for a 3-bit binary counter. The arrows indicate the state-to-state movement of the counter for both its UP and its DOWN modes of operation. An examination of $Q_0$ for both the up and down sequences shows that FF0 toggles on each clock pulse. Thus, the $J_0$ and $K_0$ inputs of FF0 are

$$\mathbf{J_0=K_0=1}$$

For the up sequence, $Q_1$ changes state on the next clock pulse when $Q_0 = 1$. For the down sequence, $Q_1$ changes on the next clock pulse when $Q_0 = 0$. Thus, the $J_1$ and $K_1$ inputs of FF1 must equal 1 under the conditions expressed by the following equation:

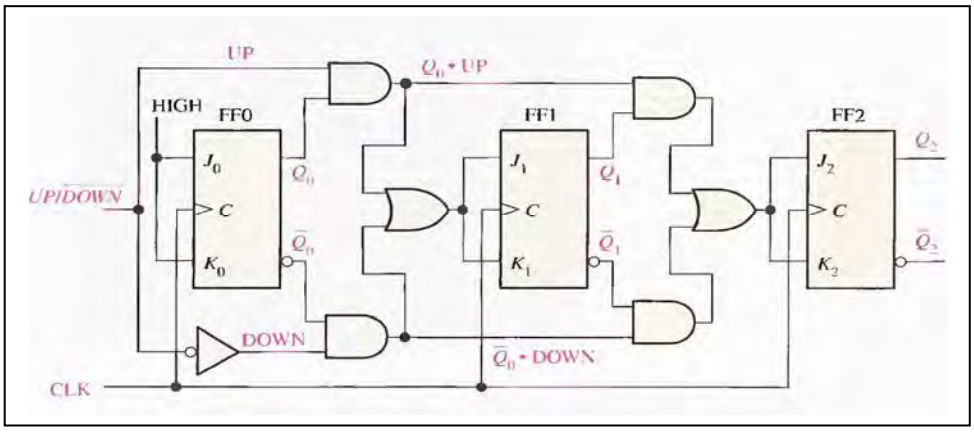
$$J_1 = K_1 = (Q_0 \cdot \text{UP}) + (\overline{Q}_0 \cdot \text{DOWN})$$

For the up sequence, $Q_2$ changes state on the next clock pulse when $Q_0 = Q_1 = 1$. For the down sequence, $Q_2$ changes on the next clock pulse when $Q_0 = Q_1 = 0$. Thus, the $J_2$ and $K_2$ inputs of FF2 must equal 1 under the conditions expressed by the following equation:

$$J_2 = K_2 = (Q_0 \cdot Q_1 \cdot \text{UP}) + (\overline{Q}_0 \cdot \overline{Q}_1 \cdot \text{DOWN})$$

**Figures below shows the logic diagram and truth table for UP/DOWN counter**

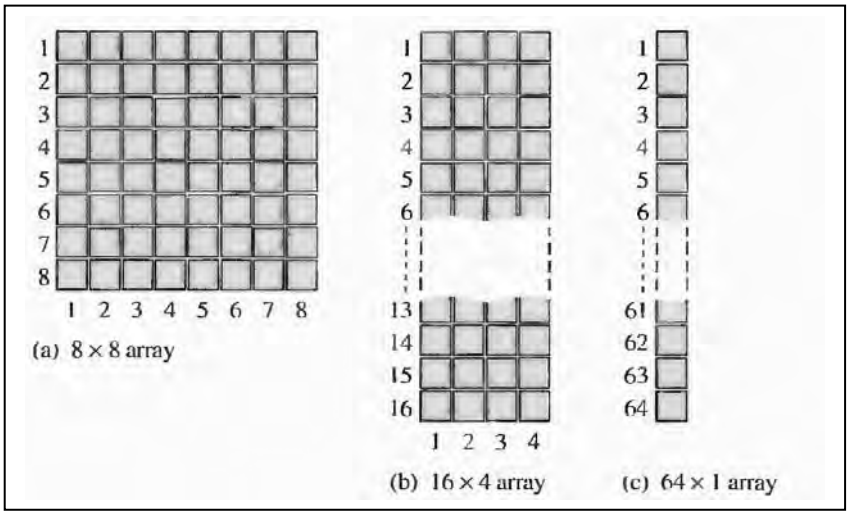
| CLOCK PULSE | UP | $Q_2$ | $Q_1$ | $Q_0$ | DOWN |
|---|---|---|---|---|---|
| 0 | | 0 | 0 | 0 | |
| 1 | | 0 | 0 | 1 | |
| 2 | | 0 | 1 | 0 | |
| 3 | | 0 | 1 | 1 | |
| 4 | | 1 | 0 | 0 | |
| 5 | | 1 | 0 | 1 | |
| 6 | | 1 | 1 | 0 | |
| 7 | | 1 | 1 | 1 | |



# CHAPTER NINE /Memories

**The Basic Semiconductor Memory Array**

Each storage element in a memory can retain either a I or a 0 and is called a cell. Memories made up of arrays of cells, as illustrated in Figure below Each block in the memory array represents one storage cell. and its location can be identified by specifying a row and a column.



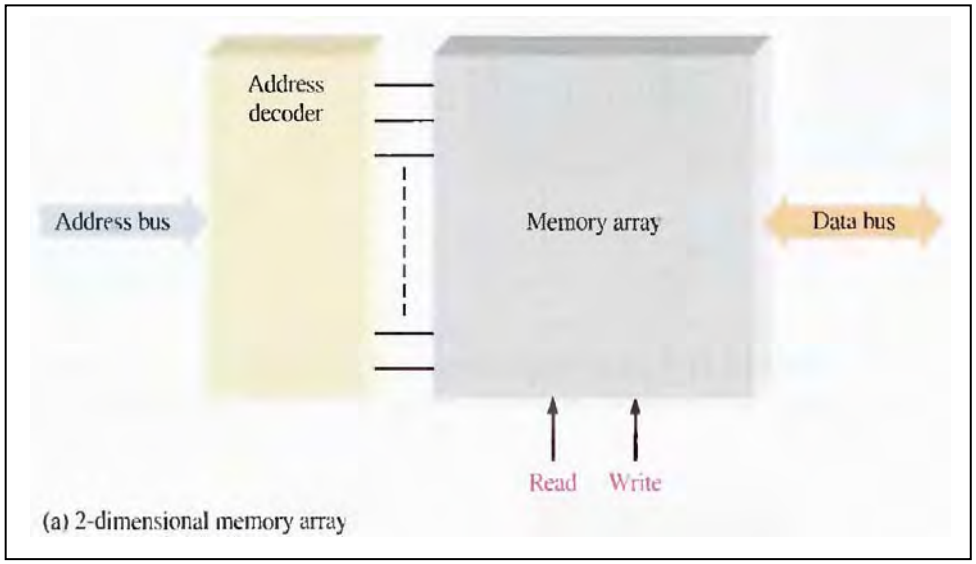

The location of a unit of data in a memory array called its address . The address of a byte specified only by the **row**.

**Basic Memory Operations**

Since a memory stores binary data. data must be put into the memory and data must be copied from the memory when needed. The write operation puts data into a specified address in the memory, and the read operation copies data out of a specified address in the memory. The addressing operation, which is part of both the write and the read operations, selects the specified memory address.

Data units go into the memory during a write operation and come out of the memory during a read operation on a set of lines called the data bus.

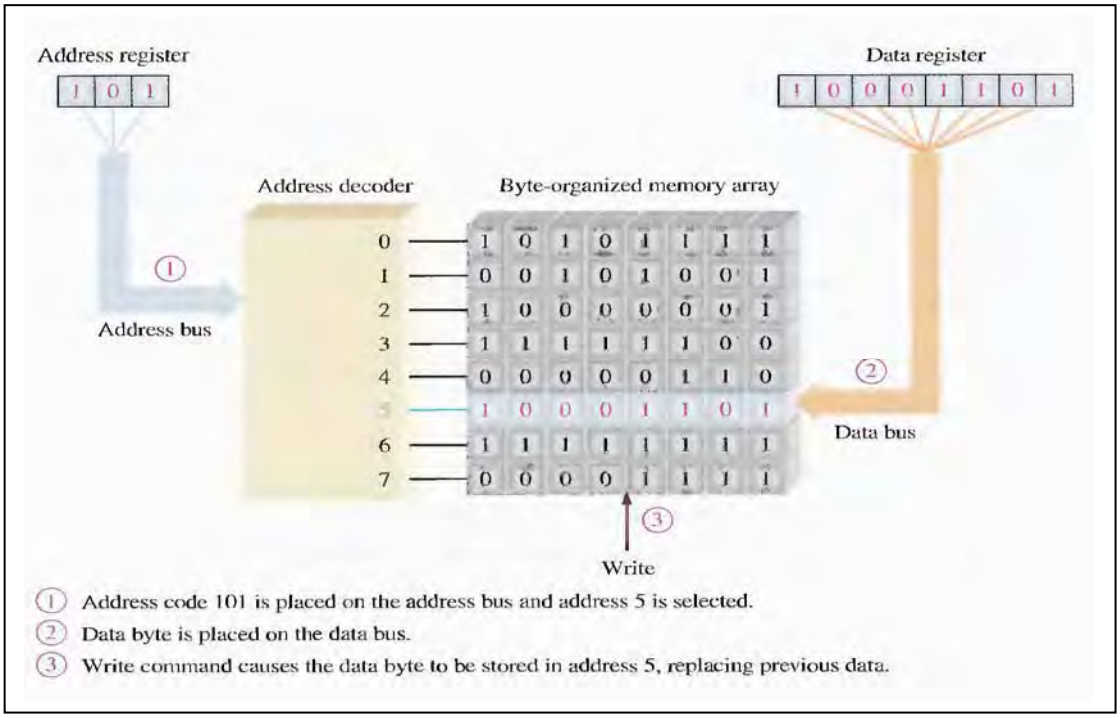
bus. As indicated in Figure below the data bus is bidirectional, which means that data can go in either direction (into the memory or out of the memory).




In this case, of byte-organized memories, the data bus has at least eight lines so that all eight bits in a selected address are transferred in parallel. For write or read operation. An address selected by placing a binary code representing the desired address on a set of lines called the address bus. The address code is decoded internally. And the appropriate address is selected.

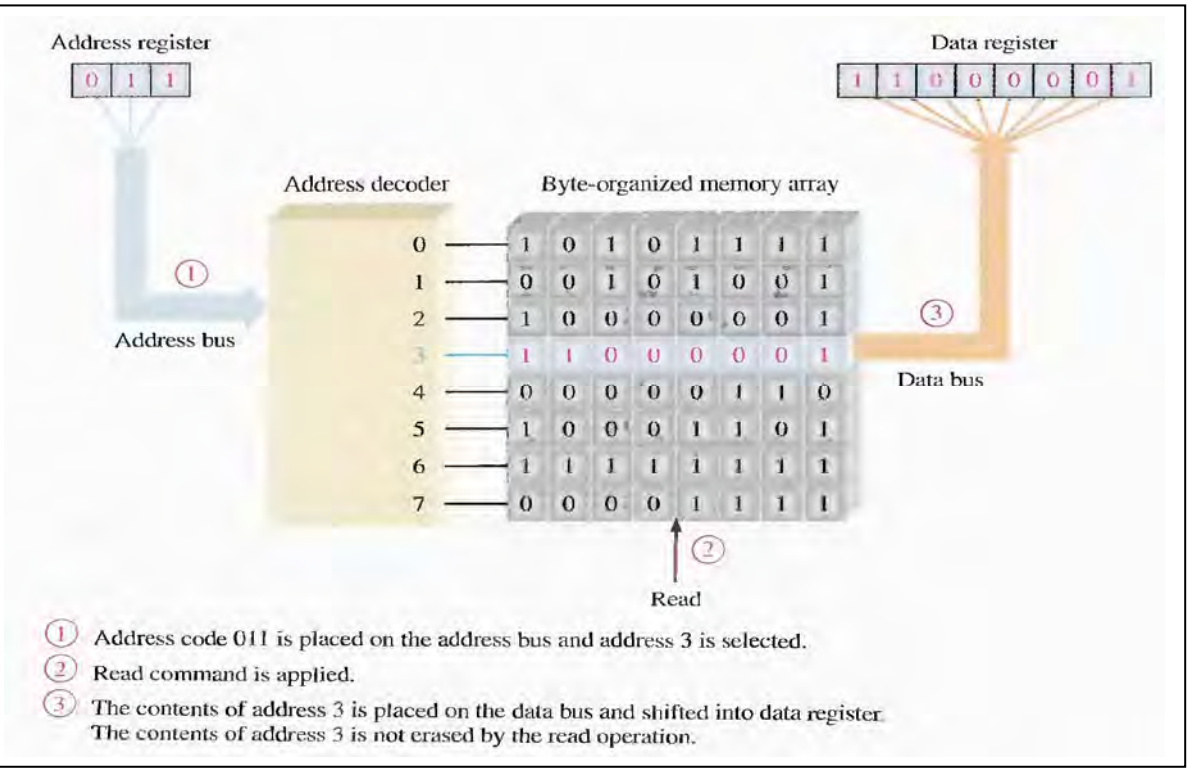
**The Write Operation**

A simplified write operation illustrated in Figure



**Read Operation**

**A simplified read operation is illustrated in Figure**



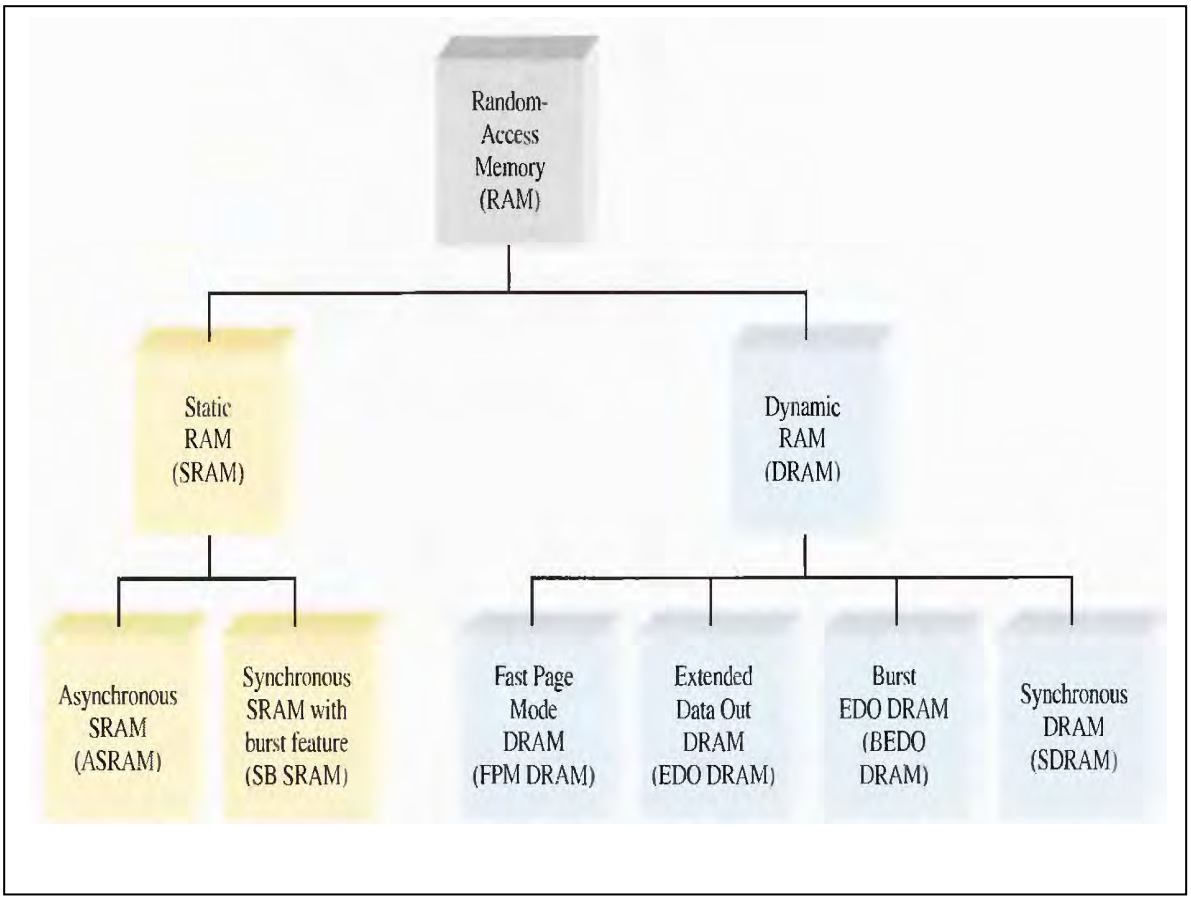

**Bit**: The smallest unit of binary data
**Byte**: data are handled in an 8-bit unit or in multiples of 8-bit units.
**Nibbles**: The byte can be split into two 4-bit units
**Word**: generally consists of one or more bytes.

**RANDOM-ACCESS MEMORIES (RAMs)**

RAM are read/write memories in which data can be written into or read from any selected address in any sequence. When a data unit is written into a given address in the RAM, the data unit previously stored at that address is replaced by the new data unit. When a data unit is read from a given address in the RAM, the data unit remains stored and is not erased by the read operation. This nondestructive read operation can be viewed as copying the content of an address while leaving the content intact. A
RAM is typically used for short-term data storage because it cannot retain stored data when power is turned off.

**The RAM Family**

The two categories of RAM are the static RAM **(SRAM)** and the dynamic RAM **(DRAM)**.
**Static RAMs** generally **use latches** as storage elements and can therefore store data indefinitely as long as dc power is applied.
**Dynamic RAMs use capacitors** as storage elements and cannot retain data very long without the capacitors being recharged by a process called refreshing.
Data can be read **much faster** from SRAMs than from DRAMs.
DRAMs can store much more data than SRAMs for a given physical size and cost because the DRAM cell is much simpler, and more cells can be crammed into a given chip area than in the SRAM. The type of RAM as shown in figure below


Random-Access Memory (RAM)
Static RAM (SRAM)
Dynamic RAM (DRAM)
Asynchronous SRAM (ASRAM)
Synchronous SRAM with burst feature (SB SRAM)
Fast Page Mode DRAM (FPM DRAM)
Extended Data Out DRAM (EDO DRAM)
Burst EDO DRAM (BEDO DRAM)
Synchronous DRAM (SDRAM)

**READ-ONLY MEMORIES (ROMs)**

ROM contain permanently or semi permanently stored data, Which can be read from the memory but either cannot be changed at all or cannot be changed without specialized equipment. ROMs retain stored data when the power is off and are therefore nonvolatile memories.

The **mask ROM** is the type in which the data are permanently stored in the memory during the manufacturing process.

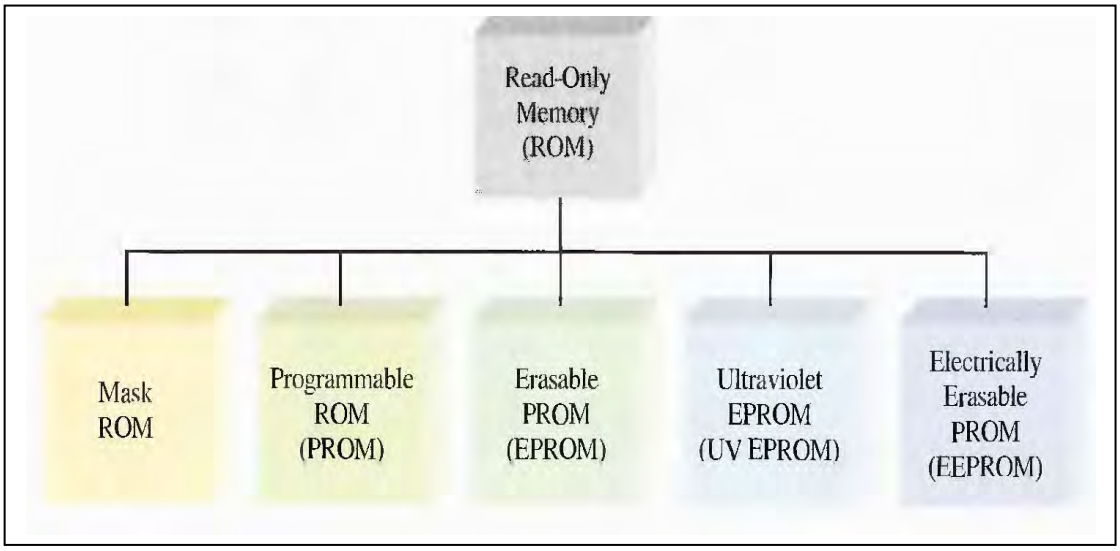
The **PROM**, or programmable ROM, is the type in which the data are electrically stored by the user with the aid of specialized equipment.

The **EPROM**, or erasable PROM.

The **UV EPROM** is electrically programmable by the user, but the stored data must be erased by exposure to ultraviolet light over a period of several minutes.

The electricallyerasable PROM (**EEPROM** or **$E^2$ PROM**) can be erased in a few milliseconds.

The types of ROM as shown in figure below



# Chapter one

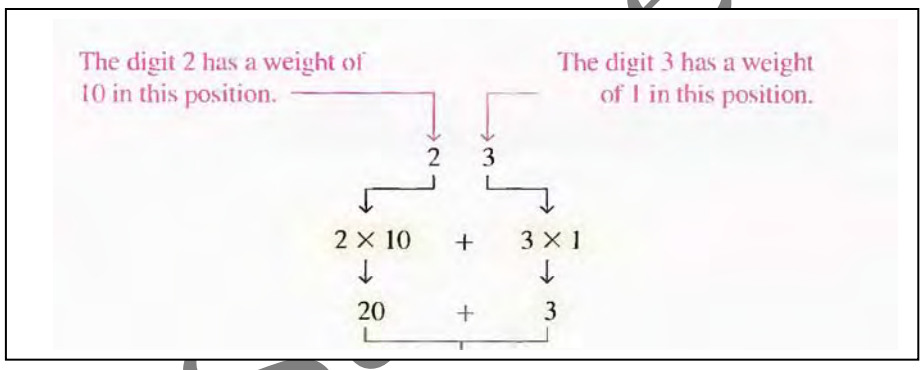
## 1- Number system and codes

The binary number system and digital codes are fundamental to computers and to digital electronics in general. The binary number system and its relationship to other number systems such as decimal, hexadecimal, and octal has presented. Arithmetic operations with binary numbers have covered to provide a basis for understanding how computers and many other types of digital systems work.

### 1.1 DECIMAL NUMBERS

We are familiar with the decimal number system because we use decimal numbers every day. The decimal number system has ten digits, 0 through 9. Represent a certain Therefore, these digits not limited because used in different position. As shown in example below.

For example




The position of each digit in a decimal number indicates the magnitude of the quantity represented and could assign a weight. The weights for whole numbers are positive powers of ten that increase from right to left, beginning with $10^0 = 1$.

$$\ldots 10^5\ 10^4\ 10^3\ 10^2\ 10^1\ 10^0$$

For fractional numbers, the weights are negative powers of ten that decrease from left to right beginning with $10^{-1}$.

$$10^2\ 10^1\ 10^0.10^{-1}\ 10^{-2}\ 10^{-3}\ldots$$
(Decimal point)

Example:

Express the decimal number 568.23 as a sum of the values of each digit.

The whole number digit 5 has a weight of 100, which is $10^2$, the digit 6 has a weight of 10, which is $10^1$, the digit 8 has a weight of 1, which is $10^0$, the fractional digit 2 has a weight of 0.1, which is $10^{-1}$, and the fractional digit 3 has a weight of 0.01, which is $10^{-2}$.

$$\begin{aligned} 568.23 &= (5 \times 10^2) + (6 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) + (3 \times 10^{-2}) \\ &= (5 \times 100) + (6 \times 10) + (8 \times 1) + (2 \times 0.1) + (3 \times 0.01) \\ &= 500 + 60 + 8 + 0.2 + 0.03 \end{aligned}$$

## 1.2 BINARY NUMBERS

The binary number has only two digits (bits) 1 and 0.

The position of a 1 or 0 in a binary number indicates its weight or value within the number,

The weights in a binary number have based on power of two.

| DECIMAL NUMBER | BINARY NUMBER | | | |
|---|---|---|---|---|
| 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 |
| 10 | 1 | 0 | 1 | 0 |
| 11 | 1 | 0 | 1 | 1 |
| 12 | 1 | 1 | 0 | 0 |
| 13 | 1 | 1 | 0 | 1 |
| 14 | 1 | 1 | 1 | 0 |
| 15 | 1 | 1 | 1 | 1 |

As we have seen in Table above, four bits are required to count from zero to 15.

In general, with n bits we can count to a number equal to $2^n$-1

Largest decimal number count = $2^n$-1

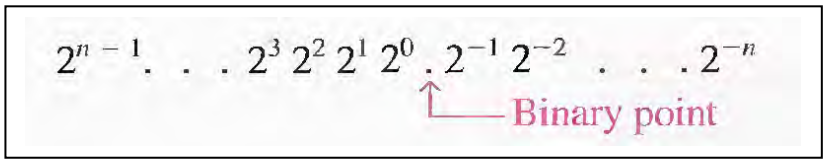
With six bits (n = 4) you can count from zero to sixty-three.

$$2^4\text{-}1 = 16\text{-}1 = 15$$

A binary number is a weighted number. The right most bit is the **LSB** (least significant bit) in a binary whole number and has a weight of $2^0 = 1$. The weights increase from **right to left** by a power of two for each bit. The left most bit is the **MSB** (most significant bit).

**Frictional** numbers can be represents in binary by placing bits to the right of the binary point. The **left**-most bit is the **MSB** in a binary fractional number, the fractional weights **decrease** from **left** to **right** by a negative power of two for each bit.

Figure below show the weights of binary fraction number where **n** is the **number of bits** from the binary point.



Binary weight table as shown below

| POSITIVE POWERS OF TWO (WHOLE NUMBERS) | | | | | | | | | NEGATIVE POWERS OF TWO (FRACTIONAL NUMBER) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ |
| 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 1/2 | 1/4 | 1/8 | 1/16 | 1/32 | 1/64 |
| | | | | | | | | | 0.5 | 0.25 | 0.125 | 0.0625 | 0.03125 | 0.015625 |

## 1.3 OCTAL NUMBERS

The octal number system is composed of eight digits, which are

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

Each octal number can be represented by three digits only 000 to 111

## 1.4 HEXADECIMAL NUMBERS

The hexadecimal number system consists of digits 0-9 and letters A-F.

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

# Conversion between systems

- **Decimal –to- binary conversion.**

We have two method discussed below

### 1. Sum-of-Weights Method

One way to find the binary number that is equivalent to a given decimal number is to determine the set of binary weights whose **sum is equal** to the decimal number.

Example:

Convert the following decimal numbers to binary:

(a) 12 (b) 25 (c) 58 (d) 82

*Solution*

(a) $12 = 8 + 4 = 2^3 + 2^2$ ⟶ 1100

(b) $25 = 16 + 8 + 1 = 2^4 + 2^3 + 2^0$ ⟶ 11001

(c) $58 = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1$ ⟶ 111010

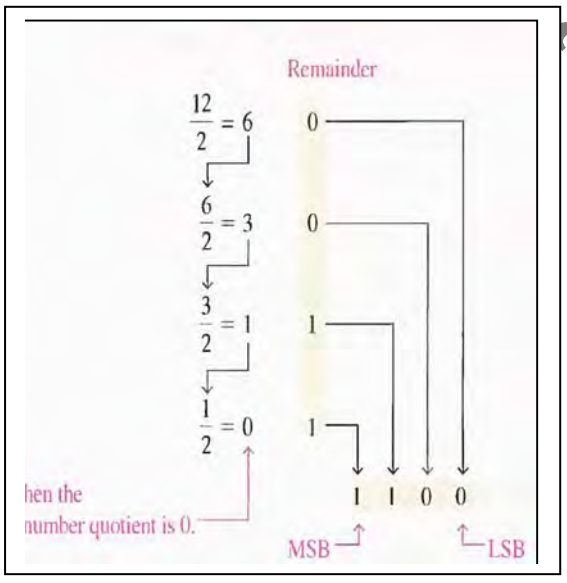
(d) $82 = 64 + 16 + 2 = 2^6 + 2^4 + 2^1$ ⟶ 1010010

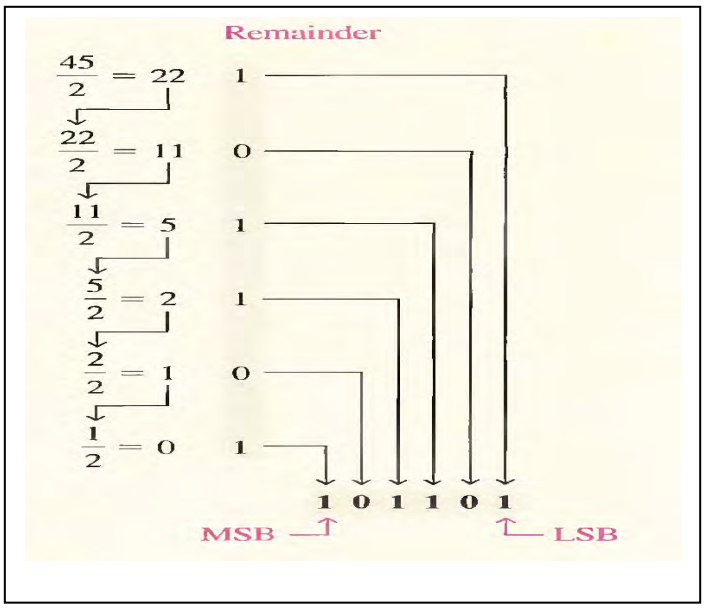
### 2. Repeated Division-by-2 Method

To get the binary number for a given decimal number, divide the decimal number by 2 until the quotient is zero. Remainders form is the binary number.

Examples below explain the process for more detail.

Examples :



- **Converting Decimal Fractions -to Binary**

An easy way to remember fractional binary weights is that the most significant weight is 0.5, which is $2^{-1}$ and that by halving any weight, you get the next lower weight; thus a list of four fractional binary weights would be 0.5, 0.25, 0.125, 0.0625.

**1. Sum-of-Weights**

The sum-of-weights method could apply to fractional decimal numbers, we determine the fraction binary wait whose sum equal to decimal number as shown in the following example:

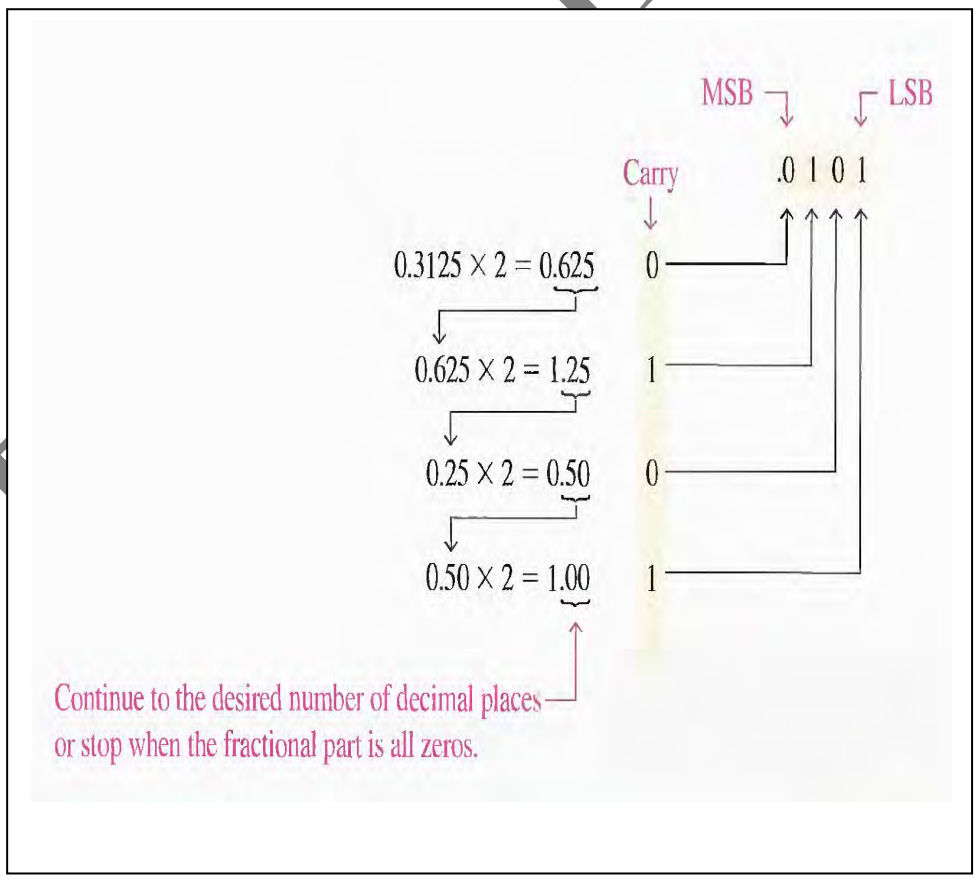
Example:

$$0.625 = 0.5 + 0.125 = 2^{-1} + 2^{-3} = 0.101$$

There is a 1 in the $2^{-1}$ position, a 0 in the $2^{-2}$ position, and a 1 in the $2^{-3}$ position.

**2. Repeated Multiplication by 2**

As you have seen, decimal whole numbers can be converted to binary by repeated division by 2. Decimal fractions can be converted to binary by repeated multiplication by 2, the carry digits are the binary number we stop multiplication when the fraction part of multiplication equal to zero For example,



**Binary-to-Decimal Conversion**

The decimal value of any binary number can be founds by **adding** the weights of all bits that are **1** and **discarding** the weights of all bits that are **zero**. Examples below more detail to conversion.

Example1:

Convert the binary whole number 1101101 to decimal.

*Solution* Determine the weight of each bit that is a 1, and then find the sum of the weights to get the decimal number.

Weight: $2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

Binary number: 1 1 0 1 1 0 1

$1101101 = 2^6 + 2^5 + 2^3 + 2^2 + 2^0$

$= 64 + 32 + 8 + 4 + 1 = \mathbf{109}$

Example2:

Convert the fractional binary number 0.1011 to decimal.

*Solution* Determine the weight of each bit that is a 1, and then sum the weights to get th decimal fraction.

Weight: $2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$

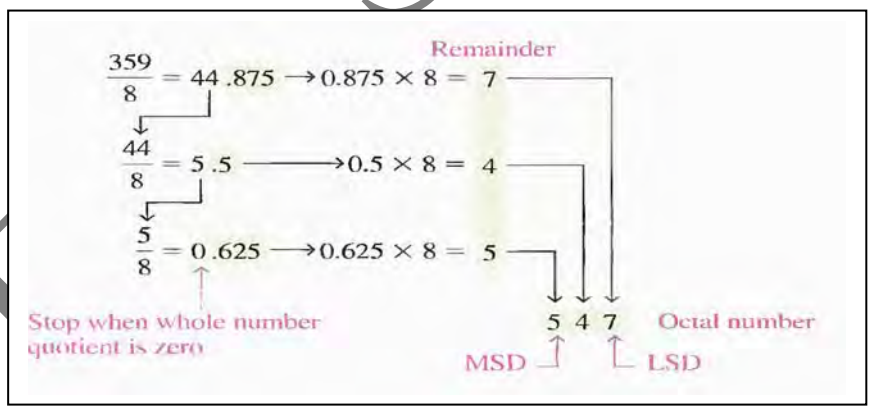
Binary number: 0 . 1 0 1 1

$0.1011 = 2^{-1} + 2^{-3} + 2^{-4}$

$= 0.5 + 0.125 + 0.0625 = \mathbf{0.6875}$

**Decimal-to-Octal Conversion :** A method of converting a decimal number to an octal number is the repeated division-by- 8

Example



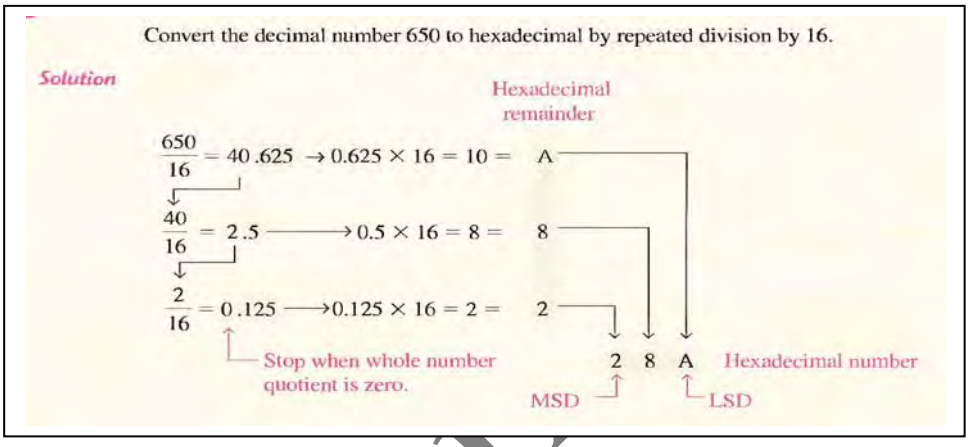

**Octal-to-Decimal Conversion**

The evaluation of an octal number in terms of its decimal equivalent has accomplished by multiplying each digit by its weight and summing the products.

$$2374_8 = (2 \times 8^3) + (3 \times 8^2) + (7 \times 8^1) + (4 \times 8^0)$$
$$= (2 \times 512) + (3 \times 64) + (7 \times 8) + (4 \times 1)$$
$$= 1024 + 192 + 56 + 4 = 1276_{10}$$

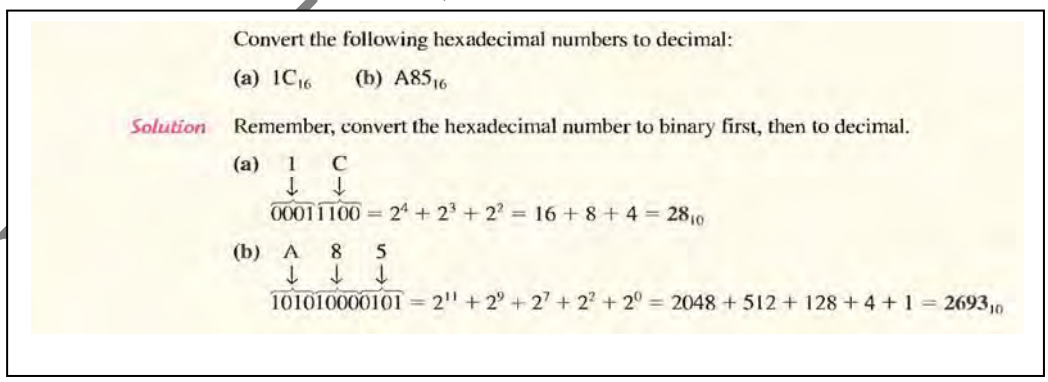
**Decimal to Hexadecimal Conversion**

Repeated division of a decimal number by 16 will produce the equivalent hexadecimal number, formed by the remainders of the divisions. The first remainder produced is the least significant digit (LSD). Each successive division by 16 yields a remainder that becomes digit in the equivalent hexadecimal number. Note that when a quotient has a fractional part, the fractional part has multiplied by the divisor to get the remainder.




**Hexadecimal-to-Decimal Conversion**

One way to find the decimal equivalent of a hexadecimal number is to first convert the




hexadecimal number to binary and then convert from binary to decimal.

Another way to convert a hexadecimal number to its decimal equivalent is to multiply

The decimal value of each hexadecimal digit by its weight and then take the sum of these products, the weights of a hexadecimal number are increasing powers of 16 (from right to left). For a 4-digit hexadecimal number, the weights are

| $16^3$ | $16^2$ | $16^1$ | $16^0$ |
|---|---|---|---|
| 4096 | 256 | 16 | 1 |

Example:

Convert the following hexadecimal numbers to decimal:

(a) $E5_{16}$ (b) $B2F8_{16}$

*Solution* Recall from Table 2–3 that letters A through F represent decimal numbers 10 through 15, respectively.

(a) $E5_{16} = (E \times 16) + (5 \times 1) = (14 \times 16) + (5 \times 1) = 224 + 5 = 229_{10}$

(b) $B2F8_{16} = (B \times 4096) + (2 \times 256) + (F \times 16) + (8 \times 1)$
$= (11 \times 4096) + (2 \times 256) + (15 \times 16) + (8 \times 1)$
$= 45{,}056 + 512 + 240 + 8 = 45{,}816_{10}$

**Binary-to-Octal Conversion** of a binary number to an octal number is the reverse of the octal-to-binary conversion

Example

Convert each of the following binary numbers to octal:

(a) 110101 (b) 101111001 (c) 100110011010 (d) 11010000100

*Solution*

(a) 110101 → 6 5 = $65_8$

(b) 101111001 → 5 7 1 = $571_8$

(c) 100110011010 → 4 6 3 2 = $4632_8$

(d) 011010000100 → 3 2 0 4 = $3204_8$

**Octal-to-Binary Conversion** : Because each octal digit can be represented by a 3-bit binary number,

| OCTAL DIGIT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| BINARY | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

Example:

Convert each of the following octal numbers to binary:

(a) $13_8$ (b) $25_8$ (c) $140_8$ (d) $7526_8$

*Solution*

(a) 1 3 → 001011

(b) 2 5 → 010101

(c) 1 4 0 → 001100000

(d) 7 5 2 6 → 111101010110

**Binary-to-Hexadecimal Conversion**

Converting a binary number to hexadecimal is a straightforward procedure. Simply break the binary number into 4-bit groups. Starting at the right-most bit, replace each 4-bit group with the equivalent hexadecimal symbol.

Convert the following binary numbers to hexadecimal:

(a) 1100101001010111 (b) 1111110001011010 01

*Solution*

(a) 1100 1010 0101 0111 → C A 5 7 = $CA57_{16}$

(b) 0011 1111 0001 0110 1001 → 3 F 1 6 9 = $3F169_{16}$

Two zeros have been added in part (b) to complete a 4-bit group at the left.

**Hexadecimal-to-Binary Conversion**

To convert from a hexadecimal number to a binary number, reverse the process and replace each hexadecimal symbol with the appropriate four bits.

Example:

Determine the binary numbers for the following hexadecimal numbers:

(a) $10A4_{16}$ (b) $CF8E_{16}$ (c) $9742_{16}$

*Solution*

(a) 1 0 A 4 → 1000 0101 0010 0 (b) C F 8 E → 1100 1111 1000 1110 (c) 9 7 4 2 → 1001 0111 0100 0010

In part (a), the MSB is understood to have three zeros preceding it, thus forming a 4-bit group.

**Octal to Hexadecimal Conversion**

To convert the octal to hex number by the fallowing steps

1- Convert the octal number to binary
2- Make group 4 digit and we add 0 to MSB
3- Convert the number to Hex

**Example:** convert $754_8$ to Hexadecimal number

| | | | |
|---|---|---|---|
| Octal | 7 | 5 | 4 |
| Binary | 00 ( 1 1 1 | 101 | 100 ) |
| | 0001 | 1110 | 1100 |
| **HEX** | **(1** | **E** | **C)$_{16}$** |

**Hexadecimal to Octal Conversion**

To convert the hex number to octal by

1- Convert the Hex number to binary
2- Make group for 3 digit and add 0 to MSB
3- Convert the number to octal

**Example:** convert the Hex $(FD4)_{16}$ to octal

(F D 4)
(1111 1101 0100)
(111 111 010 100)
**(7 7 2 4 )$_8$**

## BINARY ARITHMETIC

### Binary Addition

The four basic rules for adding binary digits (bits) are as follows:

| | |
|---|---|
| 0 + 0 = 0 | Sum of 0 with a carry of 0 |
| 0 + 1 = 1 | Sum of 1 with a carry of 0 |
| 1 + 0 = 1 | Sum of 1 with a carry of 0 |
| 1 + 1 = 10 | Sum of 0 with a carry of 1 |

When there is a carry of 1, you have a situation in which three bits are being added (bit in each of the two numbers and a carry bit). This situation has illustrated as follows:

Carry bits ↓

| | |
|---|---|
| 1 + 0 + 0 = 01 | Sum of 1 with a carry of 0 |
| 1 + 1 + 0 = 10 | Sum of 0 with a carry of 1 |
| 1 + 0 + 1 = 10 | Sum of 0 with a carry of 1 |
| 1 + 1 + 1 = 11 | Sum of 1 with a carry of 1 |

Example:

Add the following binary numbers:

(a) 11 + 11 (b) 100 + 10 (c) 111 + 11 (d) 110 + 100

*Solution* The equivalent decimal addition is also shown for reference.

| (a) | | | (b) | | | (c) | | | (d) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 3 | | 100 | 4 | | 111 | 7 | | 110 | 6 | |
| +11 | +3 | | +10 | +2 | | +11 | +3 | | +100 | +4 | |
| **110** | 6 | | **110** | 6 | | **1010** | 10 | | **1010** | 10 | |

### Addition in octal

When we add two octal number if greater than 7 we subtract 8 from result digit

Example:

71

+47

---

140

1+7 = 8 >7 than 8-8=0 with carry 1

7+4+1 =12 >7 than 12-8 = 4 with carry 1

**Hexadecimal Addition :** use the following rules:

1. In any given column of an addition problem, think of the two hexadecimal digits in terms of their decimal values. For instance, $5_{16} = 5_{10}$ and $C_{16} = 12_{10}$.
2. If the sum of these two digits is $15_{10}$ or less, bring down the corresponding hexadecimal digit.
3. If the sum of these two digits is greater than $15_{10}$, bring down the amount of the sum that exceeds $16_{10}$ and carry a 1 to the next column.

**Example:**

Add the following hexadecimal numbers:

(a) $23_{16} + 16_{16}$ (b) $58_{16} + 22_{16}$ (c) $2B_{16} + 84_{16}$ (d) $DF_{16} + AC_{16}$

*Solution*

(a) $23_{16}$ + $16_{16}$ = $39_{16}$

right column: $3_{16} + 6_{16} = 3_{10} + 6_{10} = 9_{10} = 9_{16}$

left column: $2_{16} + 1_{16} = 2_{10} + 1_{10} = 3_{10} = 3_{16}$

(b) $58_{16}$ + $22_{16}$ = $7A_{16}$

right column: $8_{16} + 2_{16} = 8_{10} + 2_{10} = 10_{10} = A_{16}$

left column: $5_{16} + 2_{16} = 5_{10} + 2_{10} = 7_{10} = 7_{16}$

(c) $2B_{16}$ + $84_{16}$ = $AF_{16}$

right column: $B_{16} + 4_{16} = 11_{10} + 4_{10} = 15_{10} = F_{16}$

left column: $2_{16} + 8_{16} = 2_{10} + 8_{10} = 10_{10} = A_{16}$

(d) $DF_{16}$ + $AC_{16}$ = $18B_{16}$

right column: $F_{16} + C_{16} = 15_{10} + 12_{10} = 27_{10}$

$27_{10} - 16_{10} = 11_{10} = B_{16}$ with a 1 carry

left column: $D_{16} + A_{16} + 1_{16} = 13_{10} + 10_{10} + 1_{10} = 24_{10}$

$24_{10} - 16_{10} = 8_{10} = 8_{16}$ with a 1 carry

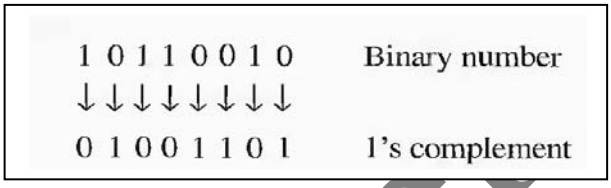
## 1-3-2 : COMPLEMENTS:

### 1'S AND 2'S COMPLEMENTS OF BINARY NUMBERS

The l's complement and the 2's complement of a binary number are important because they permit the representation of negative numbers. The method of 2's complement arithmetic has commonly used in computers to handle negative numbers.

### Finding the 1's Complement

The 1's complement of a binary number were found by changing all 1s to 0s and all 0s to 1s,

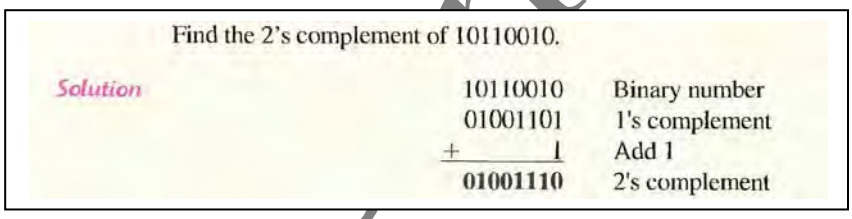



As illustrated:

### The 2' s Complement

**2's complement = (1's complement) + 1**

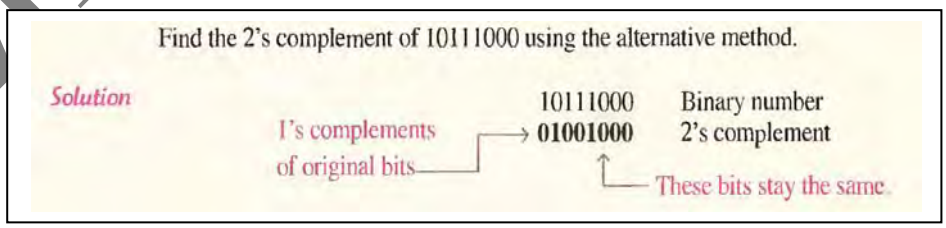
Example**:**




An alternative method of finding the 2's complement of a binary number is as follows:

**1**. Start at the right with the LSB and write the bits as they are up to and including the first 1.

**2.** Take the 1's complements of the remaining bits.



## 1st And 2nd complement in decimal

Express the decimal number −39 as an 8-bit number in the sign-magnitude, 1's complement, and 2's complement forms.

*Solution* First, write the 8-bit number for +39.

00100111

In the *sign-magnitude form*, −39 is produced by changing the sign bit to a 1 and leaving the magnitude bits as they are. The number is

**10100111**

In the *1's complement form*, −39 is produced by taking the 1's complement of +39 (00100111).

**11011000**

In the *2's complement form*, −39 is produced by taking the 2's complement of +39 (00100111) as follows:

```
 11011000   1's complement
+       1
 11011001   2's complement
```

## 1st And 2nd complement in hexadecimal number

**Method 1.** Convert the hexadecimal number to binary. Take the 2's complement of the binary number. Convert the result to hexadecimal. This is illustrated in Figure 2–4.

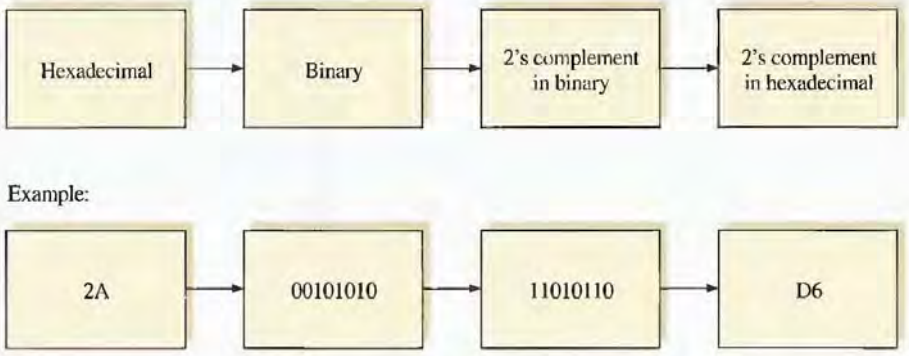



**FIGURE 2–4**
Getting the 2's complement of a hexadecimal number, Method 1.

**Method 2.** Subtract the hexadecimal number from the maximum hexadecimal number and add 1. This is illustrated in Figure 2–5.

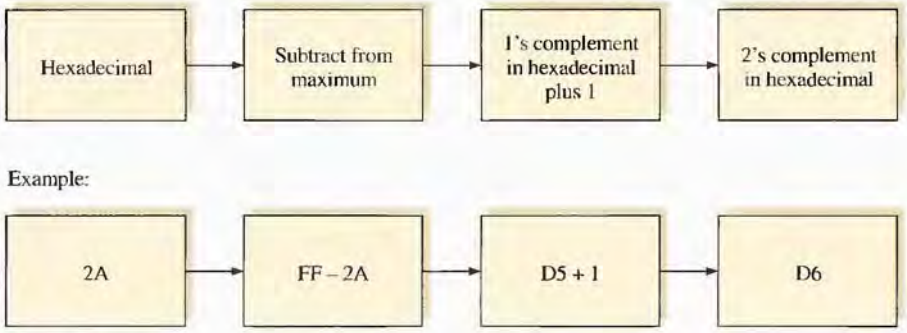



FIGURE 2–5

Getting the 2's complement of a hexadecimal number, Method 2.

**Method three :**

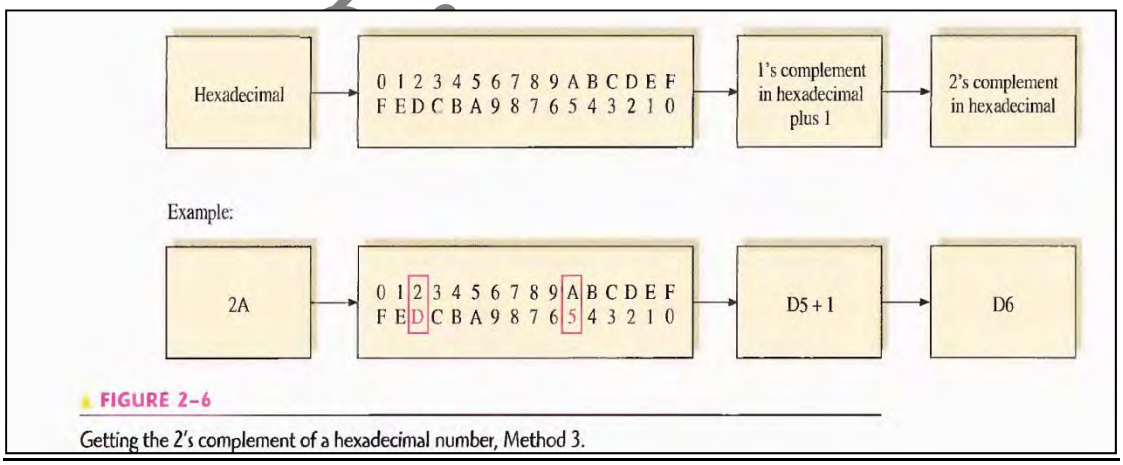



FIGURE 2–6

Getting the 2's complement of a hexadecimal number, Method 3.

## Binary Subtraction

Subtraction is addition with the sign of the subtrahend changed, and adds it to the minuend. The result of a subtraction has called the difference.

To subtract two signed numbers, take the 2's complement of the subtrahend and add. Discard any final carry bit.

**Example:**

Perform each of the following subtractions of the signed numbers:

(a) 00001000 − 00000011 (b) 00001100 − 11110111

(c) 11100111 − 00010011 (d) 10001000 − 11100010

*Solution* Like in other examples, the equivalent decimal subtractions are given for reference.

(a) In this case, 8 − 3 = 8 + (−3) = 5.

```
                00001000    Minuend (+8)
              + 11111101    2's complement of subtrahend (−3)
Discard carry ⟶ 1 00000101  Difference (+5)
```

(b) In this case, 12 − (−9) = 12 + 9 = 21.

```
  00001100    Minuend (+12)
+ 00001001    2's complement of subtrahend (+9)
  00010101    Difference (+21)
```

(c) In this case, −25 − (+19) = −25 + (−19) = −44.

```
                11100111    Minuend (−25)
              + 11101101    2's complement of subtrahend (−19)
Discard carry ⟶ 1 11010100  Difference (−44)
```

(d) In this case, −120 − (−30) = −120 + 30 = −90.

```
  10001000    Minuend (−120)
+ 00011110    2's complement of subtrahend (+30)
  10100110    Difference (−90)
```

**Multiplication**

The sign of the product of a multiplication depends on the signs of the multiplicand and the multiplier according to the following two rules:

If the signs are the same, the product is positive.

If the signs are different, the product is negative.

The basic steps in the partial products method of binary multiplication are as follows:

Step 1. Determine if the signs of the multiplicand and multiplier are the same or different. This determines what the sign of the product will be.

Step 2. Change any negative number to true (uncomplemented) form. Because most computers store negative numbers in 2's complement, a 2's complement operation is required to get the negative number into true form.

Step 3. Starting with the least significant multiplier bit, generate the partial products. When the multiplier bit is 1, the partial product is the same as the multiplicand. When the multiplier bit is 0, the partial product is zero. Shift each successive partial product one bit to the left.

Step 4. Add each successive partial product to the sum of the previous partial products to get the final product.

Step 5. if the sign bit that was determined in step 1 is negative. Take the 2's complement of the product. if positive. Leave the product in true form. Attach the sign bit to the product.

Example

Multiply the signed binary numbers: 01010011 (multiplicand) and 11000101 (multiplier).

*Solution*

**Step 1:** The sign bit of the multiplicand is 0 and the sign bit of the multiplier is 1. The sign bit of the product will be 1 (negative).

**Step 2:** Take the 2's complement of the multiplier to put it in true form.

11000101 ⟶ 00111011

**Steps 3 and 4:** The multiplication proceeds as follows. Notice that only the magnitude bits are used in these steps.

| | |
|---|---|
| 1010011 | Multiplicand |
| × 0111011 | Multiplier |
| 1010011 | 1st partial product |
| + 1010011 | 2nd partial product |
| 11111001 | Sum of 1st and 2nd |
| + 0000000 | 3rd partial product |
| 011111001 | Sum |
| + 1010011 | 4th partial product |
| 1110010001 | Sum |
| + 1010011 | 5th partial product |
| 1000011000001 | Sum |
| + 1010011 | 6th partial product |
| 1001100100001 | Sum |
| + 0000000 | 7th partial product |
| 1001100100001 | Final product |

**Step 5:** Since the sign of the product is a 1 as determined in step 1, take the 2's complement of the product.

1001100100001 ⟶ 0110011011111

Attach the sign bit

⟶ **1 0110011011111**

## Division

The numbers in a division are the **dividend**, the **divisor**, and the **quotient.** These are illustrated in the following standard division format.

$$\frac{\text{dividend}}{\text{divisor}} = \text{quotient}$$

The sign of the quotient depends on the signs of the dividend and the divisor according to the following two rules:

**If the signs are the same, the quotient is positive.**

**If the signs are different, the quotient is negative.**

When two binary numbers are divided, both numbers must be in true (uncomplememed) form. The basic steps in a division process are as follows:

**Step 1.** Determine if the signs of the dividend and divisor are the same or different. This determines what the sign of the quotient will be. Initialize the quotient to zero.

**Step 2.** Subtract the divisor from the dividend using 2's complement addition to get the first partial remainder and add 1 to the quotient. If this partial remainder is positive, go to step 3. If the partial remainder is zero or negative, the division is complete.

**Step 3.** Subtract the divisor from the partial remainder and add 1 to the quotient. If the result is positive, repeat for the next partial remainder. If the result is zero or negative, the division is complete.

Example

Divide 01100100 by 00011001.

*Solution* Step 1: The signs of both numbers are positive, so the quotient will be positive. The quotient is initially zero: 00000000.

**Step 2:** Subtract the divisor from the dividend using 2's complement addition (remember that final carries are discarded).

| | |
|---|---|
| 01100100 | Dividend |
| + 11100111 | 2's complement of divisor |
| 01001011 | Positive 1st partial remainder |

Add 1 to quotient: 00000000 + 00000001 = 00000001.

**Step 3:** Subtract the divisor from the 1st partial remainder using 2's complement addition.

| | |
|---|---|
| 01001011 | 1st partial remainder |
| + 11100111 | 2's complement of divisor |
| 00110010 | Positive 2nd partial remainder |

**Step 4:** Subtract the divisor from the 2nd partial remainder using 2's complement addition.

| | |
|---|---|
| 00110010 | 2nd partial remainder |
| + 11100111 | 2's complement of divisor |
| 00011001 | Positive 3rd partial remainder |

Add 1 to quotient: 00000010 + 00000001 = 00000011.

**Step 5:** Subtract the divisor from the 3rd partial remainder using 2's complement addition.

| | |
|---|---|
| 00011001 | 3rd partial remainder |
| + 11100111 | 2's complement of divisor |
| 00000000 | Zero remainder |

Add 1 to quotient: 00000011 + 00000001 = **00000100** (final quotient). The process is complete.

## Chapter two

- **BINARY CODED DECIMAL (BCD)** :

The 8421 code is a type of BCD (binary coded decimal) code. Binary coded decimal means that each decimal digit, 0 through 9, is represented by a binary code of four bits.

| DECIMAL DIGIT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| BCD | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |

**Example:**

Convert each of the following decimal numbers to BCD:

(a) 35 (b) 98 (c) 170 (d) 2469

*Solution*

(a) 3 5 → 00110101

(b) 9 8 → 10011000

(c) 1 7 0 → 000101110000

(d) 2 4 6 9 → 0010010001101001

BCD Addition

Step 1. Add the two BCD numbers, using the rules for binary addition.

Step 2. If a 4-bit sum is equal to or less than 9, it is a valid BCD number.

Step 3. If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added. simply add the carry to the next 4-bit group.

**Examples**

(a)

| | | |
|---|---|---|
| 1001 | | 9 |
| + 0100 | | +4 |
| 1101 | Invalid BCD number (>9) | 13 |
| + 0110 | Add 6 | |
| 0001 0011 | Valid BCD number | |
| 1 3 | | |

(d)

| | | |
|---|---|---|
| 0100 0101 0000 | | 450 |
| + 0100 0001 0111 | | + 417 |
| 1000 0110 0111 | | 867 |

## Exess 3

The *Excess–3* code also uses 4 bits to represent the decimal numbers 0 through 9 and these are shown in the Table 4.2.

*TABLE 4.2 The Excess–3 code*

| Decimal | Excess–3 |
|---|---|
| 0 | 0011 |
| 1 | 0100 |
| 2 | 0101 |
| 3 | 0110 |
| 4 | 0111 |
| 5 | 1000 |
| 6 | 1001 |
| 7 | 1010 |
| 8 | 1011 |
| 9 | 1100 |

The Excess–3 Code derives its name from the fact that each decimal representation in Excess–3 code is larger than the BCD code by three. The advantage of the Excess–3 code over the BCD code is that the Excess–3 code is a *self–complementing code* as illustrated below.

- **The Gray Code**

The Gray code is un weighted and is not an arithmetic code; that is, there are no specific weights assigned to the bit positions.

| DECIMAL | BINARY | GRAY CODE | DECIMAL | BINARY | GRAY CODE |
|---|---|---|---|---|---|
| 0 | 0000 | 0000 | 8 | 1000 | 1100 |
| 1 | 0001 | 0001 | 9 | 1001 | 1101 |
| 2 | 0010 | 0011 | 10 | 1010 | 1111 |
| 3 | 0011 | 0010 | 11 | 1011 | 1110 |
| 4 | 0100 | 0110 | 12 | 1100 | 1010 |
| 5 | 0101 | 0111 | 13 | 1101 | 1011 |
| 6 | 0110 | 0101 | 14 | 1110 | 1001 |
| 7 | 0111 | 0100 | 15 | 1111 | 1000 |

**Binary-to-Gray Code Conversion**

1. The most significant bit (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

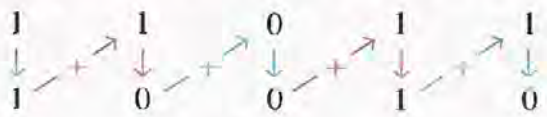
For example, the conversion of the binary number 10110 to Gray code is as follows:


The Gray code is 11101.

**Gray-to-Binary Conversion**

1. The most significant bit (left-most) in the binary code is the same as the corresponding bit in the Gray code.
2. Add each binary code bit generated to the Gray code bit in the next adjacent position. Discard carries.

For example, the conversion of the Gray code word 11011 to binary is as follows:




The binary number is 10010.

**Parity code**

A given system operates with even or odd parity, but not both. For instance, if a system operates with even parity, a check is made on each group of bits received to make sure the total number of 1s in that group is even. If there is an odd number of 1s, an error has occurred.

As an illustration of how parity bits are attached to a code, Table 2–10 lists the parity bits for each BCD number for both even and odd parity. The parity bit for each BCD number is in the *P* column.

TABLE 2–10

The BCD code with parity bits.

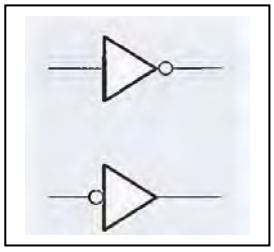
| EVEN PARITY | | ODD PARITY | |
|---|---|---|---|
| *P* | BCD | *P* | BCD |
| 0 | 0000 | 1 | 0000 |
| 1 | 0001 | 0 | 0001 |
| 1 | 0010 | 0 | 0010 |
| 0 | 0011 | 1 | 0011 |
| 1 | 0100 | 0 | 0100 |
| 0 | 0101 | 1 | 0101 |
| 0 | 0110 | 1 | 0110 |
| 1 | 0111 | 0 | 0111 |
| 1 | 1000 | 0 | 1000 |
| 0 | 1001 | 1 | 1001 |

The parity bit can be attached to the code at either the beginning or the end, depending on system design. Notice that the total number of 1s, including the parity bit, is always even for even parity and always odd for odd parity.

# Chapter three
# Boolean algebra

- **INVERTER**

Standard logic symbols for the inverter are shown in Figure below:



**Inverter Truth Table**

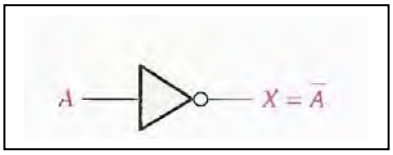
When a HIGH level is applied to an inverter input, a LOW level will appear on its output. When a LOW level is applied to its input, a HIGH will appear on its output.

| INPUT | OUTPUT |
|---|---|
| LOW (0) | HIGH (1) |
| HIGH (1) | LOW (0) |

**logic Expression for an Inverter**

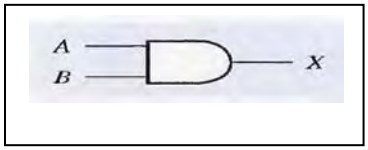
In Boolean algebra, which is the mathematics of logic circuits the operation of an inverter (NOT circuit) can be expressed as follows: If the input variable is called, **A** and the output variable is called X, then

$$X = \overline{A}$$




- **AND GATE**

An AND gate produces a HIGH output only when all of the inputs are HIGH, otherwise any of the inputs is LOW or both low, the output is LOW.



**AND Gate Truth Table**

The total number of possible combinations of binary inputs to a gate is determined by the following formula:

$$N= 2^n$$

The operation of a 2-input AND gate can be expressed in equation form as follows: If one input variable is A, the other input variable is B, and the output variable is X, then the Boolean expression is

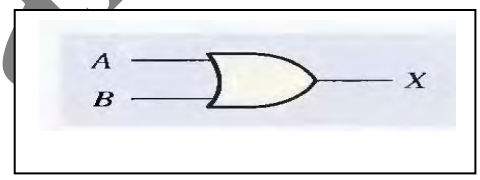
$$X=A.B$$

| INPUTS A | INPUTS B | OUTPUT X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

1 = HIGH, 0 = LOW

- **OR GATE**

An OR gate symbol as shown in figure




**OR Gate Truth Table**

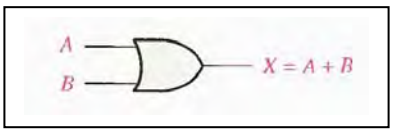
For a 2-input OR gate, output X is HIGH when either input A or input B is HIGH, or when both A and B are HIGH; X is LOW only when both A and B are LOW.

| INPUTS A | INPUTS B | OUTPUT X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1 = HIGH, 0 = LOW

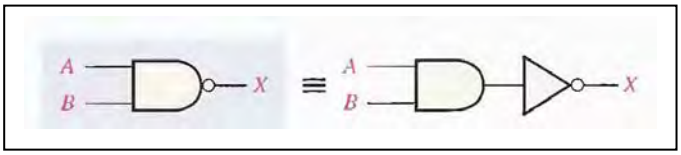
**Logic Expressions for an OR Gate**

The logical OR function of two variables is represented mathematically by a (+) between the two variables, for example, A + B.



**NAND GATE**

The term NAND is a contraction of NOT-AND and implies an AND function with a complemented (inverted) output.




**Operation of a NAND Gate**

For a 2-input NAND gate, output X is LOW only when inputs A and B are HIGH; X is HIGH when either A or B is LOW, or when both A and B are LOW.

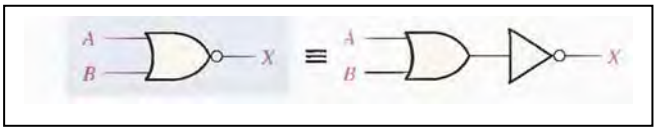
**Logic Expressions for a NAND Gate**

| INPUTS | | OUTPUT |
|---|---|---|
| A | B | X |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

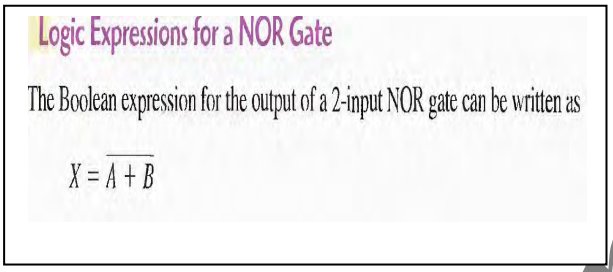
1 = HIGH, 0 = LOW.

**NOR GATE**

The NOR is the same as the OR except the output is inverted.




For a 2-input NOR gate, output X is LOW when either input A or input B is HIGH, or when both A and B are HIGH; X is HIGH only when both A and B are LOW.

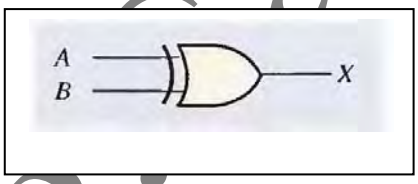



| INPUTS A | INPUTS B | OUTPUT X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

1 = HIGH, 0 = LOW.

- **Exclusive-OR Gate**

Standard symbols for an exclusive-OR (XOR) gate is shown in Figure



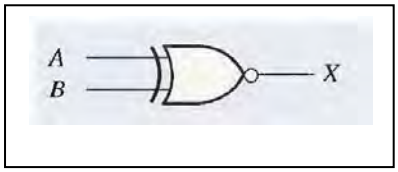

Output X is HIGH when input A is LOW and input B is HIGH, or when input A is HIGH and input B is LOW: X is LOW when A and B are both HIGH or both LOW.

| INPUTS A | INPUTS B | OUTPUT X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

- **Exclusive-NOR Gate**

Standard symbols for an exclusive-NOR (XNOR) gate are shown in Figure
Output X is LOW when input A is LOW and input B is HIGH, or when A is HIGH and B is LOW; X is HIGH when A and B are both HIGH or both LOW.

| INPUTS A | INPUTS B | OUTPUT X |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



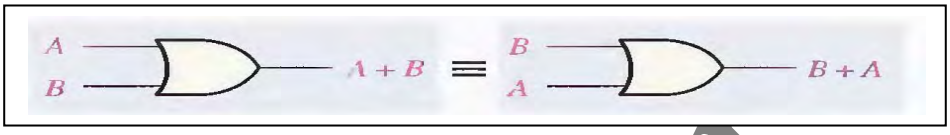
- **LAWS AND RULES OF BOOLEAN ALGEBRA**

**Laws of Boolean Algebra**

**Equation 1**

*Commutative Laws* The *commutative law of addition* for two variables is written as

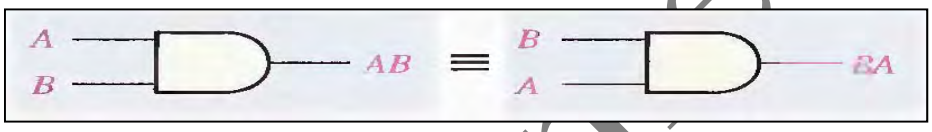
$$A + B = B + A$$




**Equation 2**

The *commutative law of multiplication* for two variables is

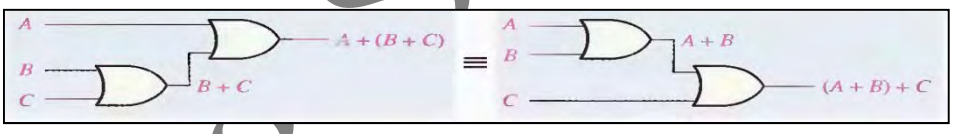
$$AB = BA$$




**Equation 3**

*Associative Laws* The *associative law of addition* is written as follows for three variables:

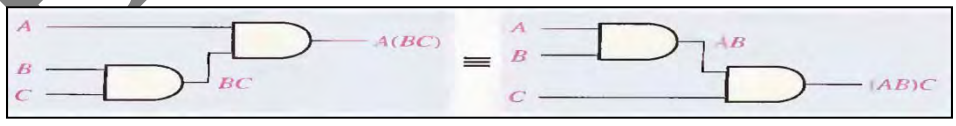
$$A + (B + C) = (A + B) + C$$




**Equation 4**

The *associative law of multiplication* is written as follows for three variables:

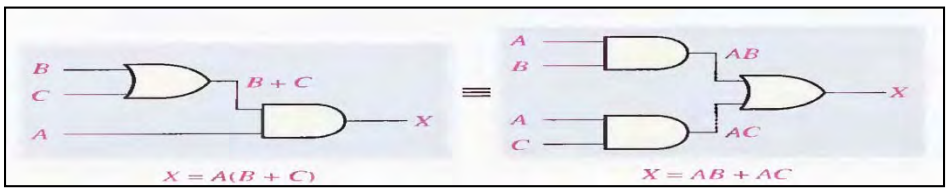
$$A(BC) = (AB)C$$




**Equation 5**

*Distributive Law* The distributive law is written for three variables as follows

$$A(B + C) = AB + AC$$



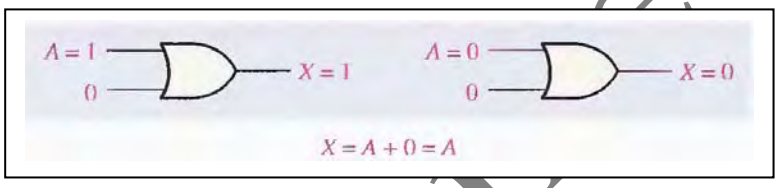
**Rules of Boolean algebra**

Table below lists 12 basic rules that are useful in manipulating and simplifying Boolean expressions.

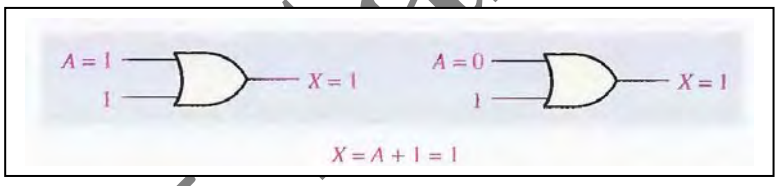
| | |
|---|---|
| 1. $A + 0 = A$ | 7. $A \cdot A = A$ |
| 2. $A + 1 = 1$ | 8. $A \cdot \overline{A} = 0$ |
| 3. $A \cdot 0 = 0$ | 9. $\overline{\overline{A}} = A$ |
| 4. $A \cdot 1 = A$ | 10. $A + AB = A$ |
| 5. $A + A = A$ | 11. $A + \overline{A}B = A + B$ |
| 6. $A + \overline{A} = 1$ | 12. $(A + B)(A + C) = A + BC$ |

*A, B,* or *C* can represent a single variable or a combination of variables.

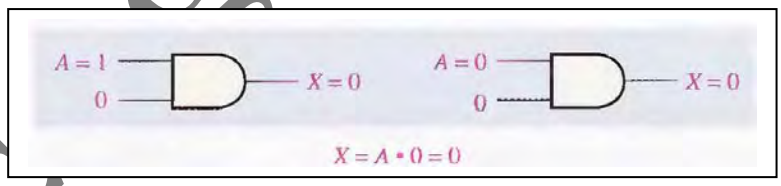
**Rule 1. A + 0 = A : A** variable OR with 0 is always equal to the variable.



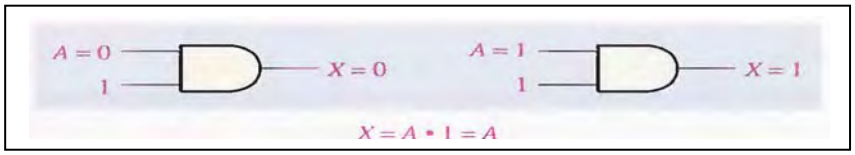

**Rule 2.** A + 1 = 1 : A variable OR with 1 is always equal to 1.



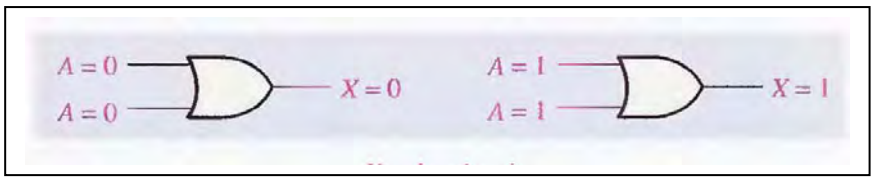

**Rule 3.** A . 0 = 0 A variable AND with 0 is always equal to 0.




**Rule 4.** A . 1 = A :A variable AND with 1 is always equal to the variable.

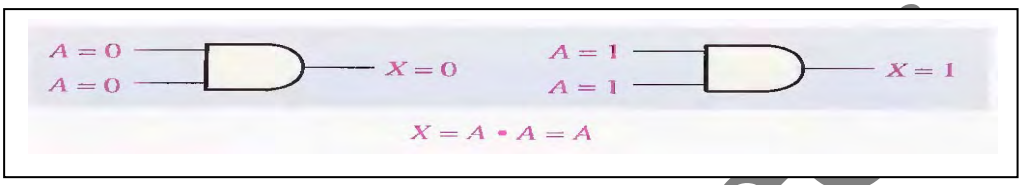



**Rule 5.** A + A = A : A variable OR with itself is always equal to the variable.

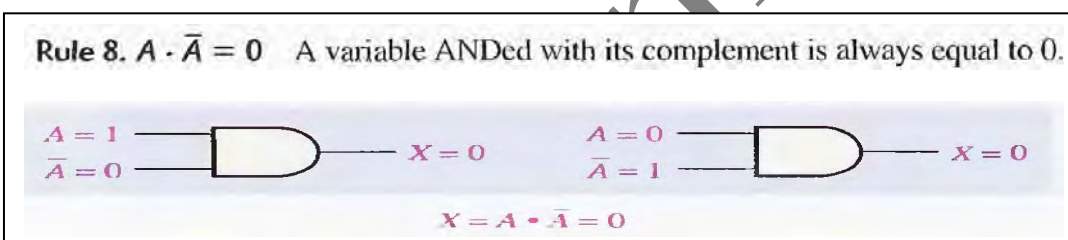


**Rule 6.** $A + \bar{A} = 1$ A variable ORed with its complement is always equal to 1

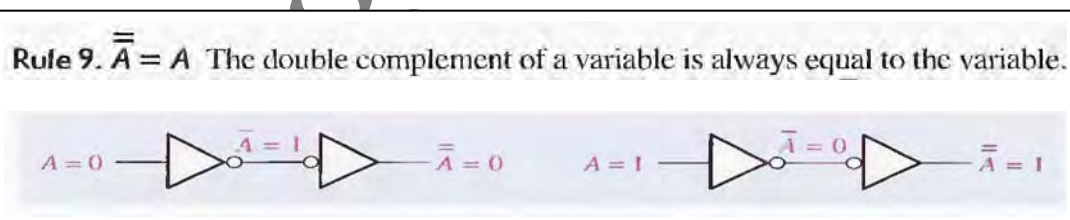

**Rule 7.** **A . A = A**: A variable AND with itself is always equal to the variable.




**Rule 8.** $A \cdot \bar{A} = 0$ A variable ANDed with its complement is always equal to 0.

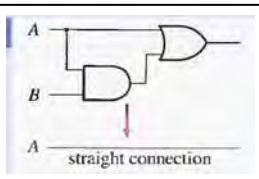


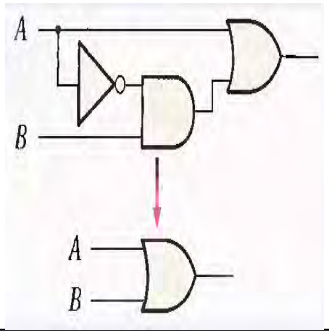

**Rule 9.** $\bar{\bar{A}} = A$ The double complement of a variable is always equal to the variable.




**Rule 10.** $A + AB = A$ This rule can be proved by applying the distributive law, rule 2, and rule 4 as follows:

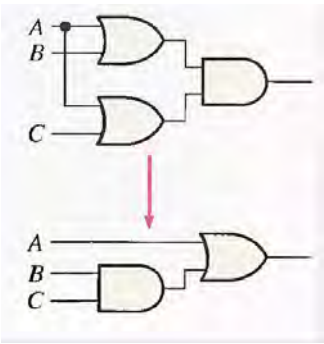
$$A + AB = A(1 + B) \quad \text{Factoring (distributive law)}$$
$$= A \cdot 1 \quad \text{Rule 2: } (1 + B) = 1$$
$$= A \quad \text{Rule 4: } A \cdot 1 = A$$






**Rule 11.** $A + \bar{A}B = A + B$ This rule can be proved as follows:

| | |
|---|---|
| $A + \bar{A}B = (A + AB) + \bar{A}B$ | Rule 10: $A = A + AB$ |
| $= (AA + AB) + \bar{A}B$ | Rule 7: $A = AA$ |
| $= AA + AB + A\bar{A} + \bar{A}B$ | Rule 8: adding $A\bar{A} = 0$ |
| $= (A + \bar{A})(A + B)$ | Factoring |
| $= 1 \cdot (A + B)$ | Rule 6: $A + \bar{A} = 1$ |
| $= A + B$ | Rule 4: drop the 1 |




**Rule 12.** $(A + B)(A + C) = A + BC$ This rule can be proved as follows:

| | |
|---|---|
| $(A + B)(A + C) = AA + AC + AB + BC$ | Distributive law |
| $= A + AC + AB + BC$ | Rule 7: $AA = A$ |
| $= A(1 + C) + AB + BC$ | Factoring (distributive law) |
| $= A \cdot 1 + AB + BC$ | Rule 2: $1 + C = 1$ |
| $= A(1 + B) + BC$ | Factoring (distributive law) |
| $= A \cdot 1 + BC$ | Rule 2: $1 + B = 1$ |
| $= A + BC$ | Rule 4: $A \cdot 1 = A$ |