UNIT-II
Boolean algebra and Logic Gates

BOOLEAN OPERATIONS AND EXPRESSIONS

Variable, complement, and literal are terms used in Boolean algebra. A variable is a symbol used to represent a logical quantity. Any single variable can have a 1 or a 0 value. The complement is the inverse of a variable and is indicated by a bar over variable (overbar). For example, the complement of the variable A is A. If A = 1, then A = 0. If A = 0, then A = 1. The complement of the variable A is read as "not A" or "A bar." Sometimes a prime symbol rather than an overbar is used to denote the complement of a variable; for example, B' indicates the complement of B. A literal is a variable or the complement of a variable.

Boolean Addition

Recall from part 3 that Boolean addition is equivalent to the OR operation. In Boolean algebra, a sum term is a sum of literals. In logic circuits, a sum term is produced by an OR operation with no AND operations involved. Some examples of sum terms are A + B, A + B, A +
B + C, and A + B + C + D.

A sum term is equal to 1 when one or more of the literals in the term are 1. A sum term is equal to 0 only if each of the literals is 0.

Example

Determine the values of A, B, C, and D that make the sum term
A + B + C + D equal to 0.

Boolean Multiplication

Also recall from part 3 that Boolean multiplication is equivalent to the AND operation. In Boolean algebra, a product term is the product of literals. In

logic circuits, a product term is produced by an AND operation with no OR operations involved. Some examples of product terms are AB, $A\overline{B}$, ABC, and ABCD.

A product term is equal to 1 only if each of the literals in the term is 1. A product term is equal to 0 when one or more of the literals are 0.

Example

Determine the values of A, B, C, and D that make the product term $\overline{A}BC\overline{D}$ equal to 1.

LAWS AND RULES OF BOOLEAN ALGEBRA

■ *Laws of Boolean Algebra*

The basic laws of Boolean algebra-the commutative laws for addition and multiplication, the associative laws for addition and multiplication, and the distributive law-are the same as in ordinary algebra.

Commutative Laws

►The commutative law of addition for two variables is written as

A+B = B+A

This law states that the order in which the variables are ORed makes no difference. Remember, in Boolean algebra as applied to logic circuits, addition and the OR operation are the same. Fig.(4-1) illustrates the commutative law as applied to the OR gate and shows that it doesn't matter to which input each variable is applied. (The symbol ≡ means "equivalent to.").

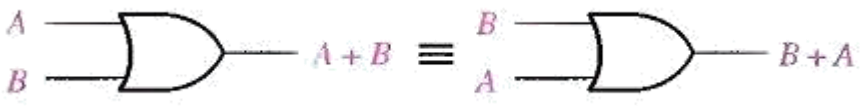


Fig.(4-1) Application of commutative law of addition.

►The commutative law of multiplication for two variables

is A.B = B.A

This law states that the order in which the variables are ANDed makes no difference. Fig.(4-2), il1ustrates this law as applied to the AND gate.

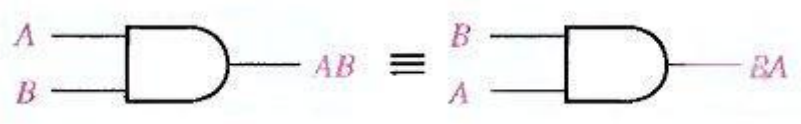


Fig.(4-2) Application of commutative law of multiplication.

Associative Laws :

►The associative law of addition is written as follows for three variables:

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

This law states that when ORing more than two variables, the result is the same regardless of the grouping of the variables. Fig.(4-3), illustrates this law as applied to 2-input OR gates.

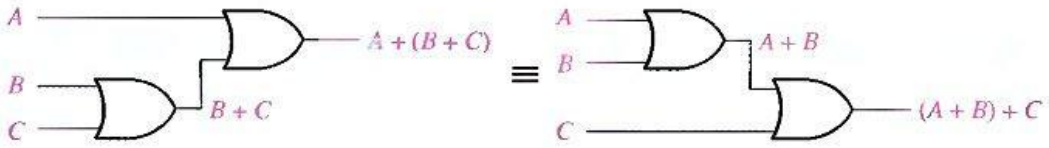


Fig.(4-3) Application of associative law of addition.

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

A(BC) = (AB)C

This law states that it makes no difference in what order the variables are grouped when ANDing more than two variables. Fig.(4-4) illustrates this law as applied to 2-input AND gates.

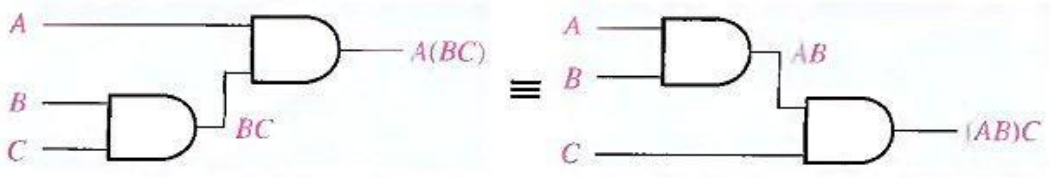


Fig.(4-4) Application of associative law of multiplication.

Distributive Law:

►The distributive law is written for three variables as follows:

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

This law states that ORing two or more variables and then ANDing the result with a single variable is equivalent to ANDing the single variable with each of the two or more variables and then ORing the products. The distributive law also expresses the process of factoring in which the common variable A is factored out of the product terms, for example,

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

Fig.(4-5) illustrates the distributive law in terms of gate implementation.

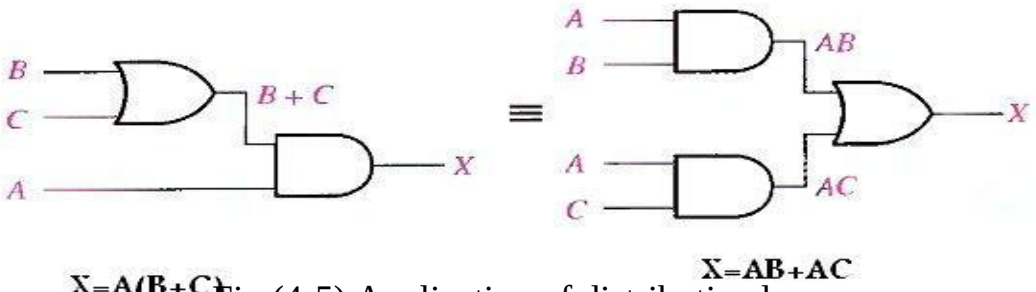


Fig.(4-5) Application of distributive law.

■ ***Rules of Boolean Algebra***

Table 4-1 lists 12 basic rules that are useful in manipulating and simplifying Boolean expressions. Rules 1 through 9 will be viewed in terms of their application to logic gates. Rules 10 through 12 will be derived in terms of the simpler rules and the laws previously discussed.

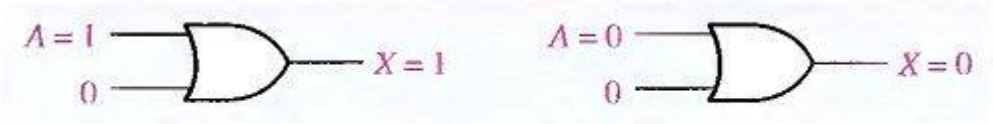
Table 4-1 Basic rules of Boolean algebra.

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 ORed with 0 is always equal to the variable. If the input variable A is 1, the output variable X is 1, which is equal to A. If A is 0, the output is 0, which is also equal to A. This rule is illustrated in Fig.(4-6), where the lower input is fixed at 0.



A+0=A

Fig.(4-6)

Rule 2. A + 1 = 1

A variable ORed with 1 is always equal to 1. A 1 on an input to an OR gate produces a 1 on the output, regardless of the value of the variable on the other input. This rule is illustrated in Fig.(4-7), where the lower input is fixed at 1.

X=A+1=1

Fig.(4-7)

Rule 3. A . 0 = 0

A variable ANDed with 0 is always equal to 0. Any time one input to an AND gate is 0, the output is 0, regardless of the value of the variable on the other input. This rule is illustrated in Fig.(4-8), where the lower input is fixed at 0.

X= A.0 = 0

Fig.(4-8)

Rule 4. A . 1 = A

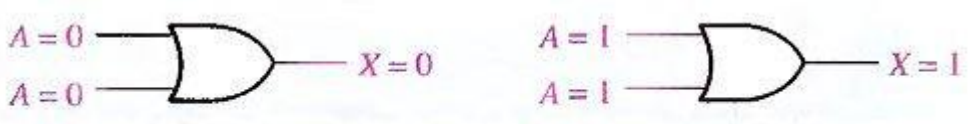
A variable ANDed with 1 is always equal to the variable. If A is 0 the output of the AND gate is 0. If A is 1, the output of the AND gate is 1 because both inputs are now 1s. This rule is shown in Fig.(4-9), where the lower input is fixed at 1.

X = A . 1 = A

Fig.(4-9)

Rule 5. A + A = A

A variable ORed with itself is always equal to the variable. If A is 0, then 0 + 0 = 0; and if A is 1, then 1 + 1 = 1. This is shown in Fig.(4-10), where both inputs are the same variable.

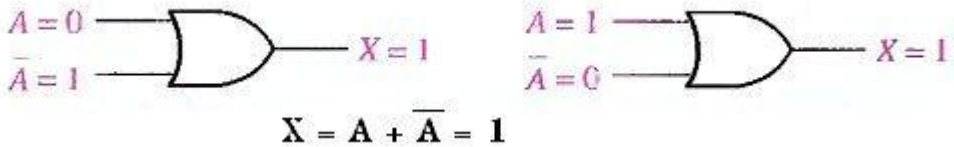


X = A + A = A

Fig.(4-10)

Rule 6. A + $\overline{A}$ = 1

A variable ORed with its complement is always equal to 1. If A is 0, then 0 + $\overline{0}$ = 0 + 1 = 1. If A is l, then 1 + 1 = 1+ 0 = 1. See Fig.(4-11), where one input is the complement of the other.

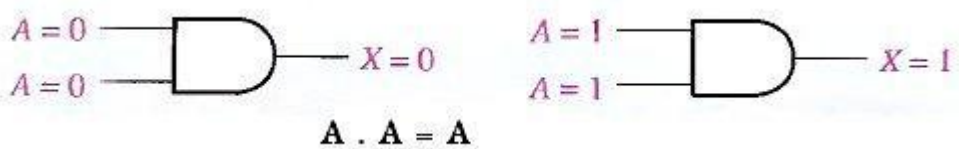


X = A + $\overline{A}$ = 1

Fig.(4-11)

Rule 7. A . A = A

A variable ANDed with itself is always equal to the variable. If A = 0, then 0.0 = 0; and if A = 1. then 1.1 = 1. Fig.(4-12) illustrates this rule.



A . A = A

Fig.(4-12)

Rule 8. $A \cdot \overline{A} = 0$

A variable ANDed with its complement is always equal to 0. Either A or $\overline{A}$ will always be 0: and when a 0 is applied to the input of an AND gate. the output will be 0 also. Fig.(4-13) illustrates this rule.

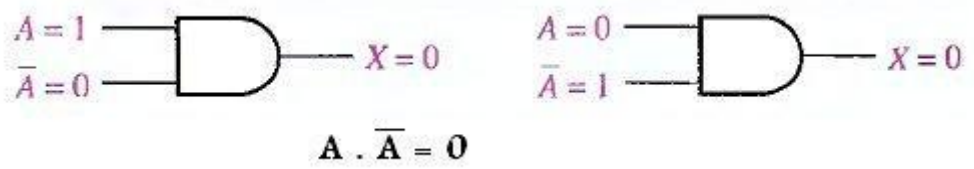


Fig.(4-13)

Rule 9 $A = \overline{\overline{A}}$

The double complement of a variable is always equal to the variable. If you start with the variable A and complement (invert) it once, you get $\overline{A}$. If you then take $\overline{A}$ and complement (invert) it, you get A, which is the original variable. This rule is shown in Fig.(4-14) using inverters.

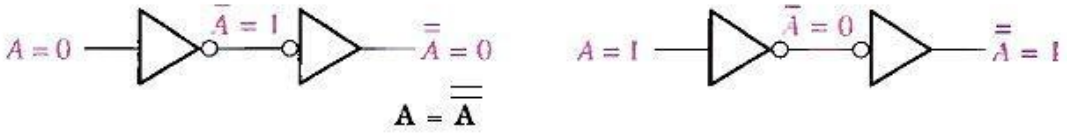


Fig.(4-14)

Rule 10. $A + AB = A$

This rule can be proved by applying the distributive law, rule 2, and rule 4 as follows:

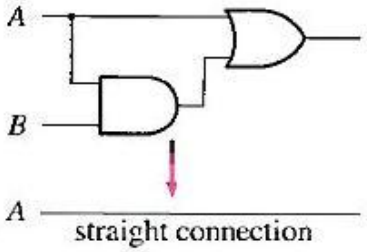
$A + AB = A(1 + B)$	Factoring (distributive law)
$= A . l$	Rule 2: (1 + B) = 1
$= A$	Rule 4: A . 1 = A

The proof is shown in Table 4-2, which shows the truth table and the resulting logic circuit simplification.

Table 4-2

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

equal



Rule 11. $A + A\overline{B} = A + B$

This rule can be proved as follows:

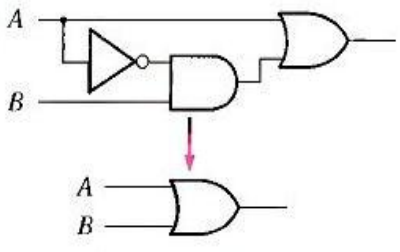
$A + \overline{A}B = (A + AB) + \overline{A}B$	Rule 10: A = A + AB
$= (AA + AB) + \overline{A}B$	Rule 7: A = AA
$=AA + AB + A\overline{A} + \overline{A}B$	Rule 8: adding $A\overline{A} = 0$
$= (A + \overline{A})(A + B)$	Factoring
$= 1. (A + B)$	Rule 6: $A + \overline{A} = 1$
$=A + B$	Rule 4: drop the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

The proof is shown in Table 4-3, which shows the truth table and the resulting logic circuit simplification.

Table 4-3

A	B	$\overline{A}B$	$A + \overline{A}B$	A + B
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

equal



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: $AA = A$

$= A(1 + C) + AB + BC$ Rule 2: $1 + C = 1$

$= A. 1 + AB + BC$ Factoring (distributive law)

$= A(1 + B) + BC$ Rule 2: $1 + B = 1$

$= A. 1 + BC$ Rule 4: $A . 1 = A$

$= A + BC$

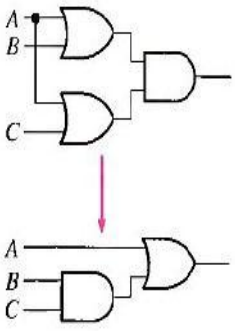
The proof is shown in Table 4-4, which shows the truth table and the resulting logic circuit

simplification.

Table 4-4

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

equal



DEMORGAN'S THEOREMS

DeMorgan, a mathematician who knew Boole, proposed two theorems that are an important part of Boolean algebra. In practical terms. DeMorgan's theorems provide mathematical verification of the equivalency of the NAND and negative-OR gates and the equivalency of the NOR and negative-AND gates, which were discussed in part 3.

One of DeMorgan's theorems is stated as follows:

The complement of a product of variables is equal to the sum of the complements of the variables,

Stated another way,

The complement of two or more ANDed variables is equivalent to the OR of the complements of the individual variables.

The formula for expressing this theorem for two variables is

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

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.

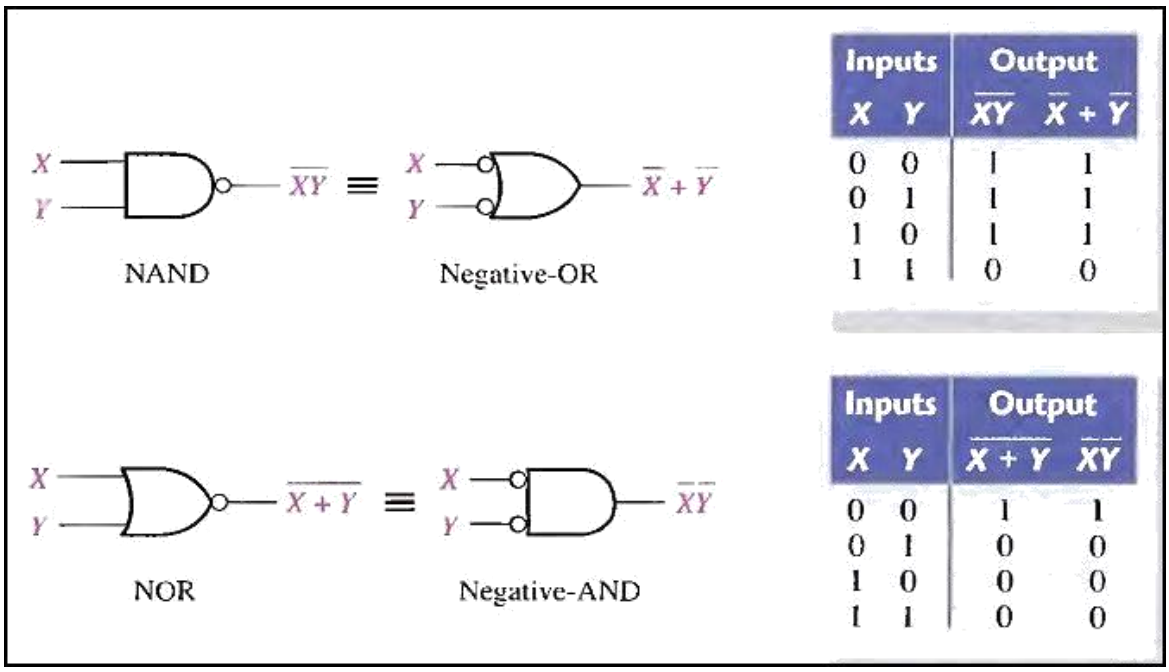
Stated another way,

The complement of two or more ORed variables is equivalent to the AND of the complements of the individual variables,

The formula for expressing this theorem for two variables is

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

Fig.(4-15) shows the gate equivalencies and truth tables for the two equations above.



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

Inputs		Output	
X	Y	$\overline{X + Y}$	$\overline{X}\,\overline{Y}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Fig.(4-15) Gate equivalencies and the corresponding truth tables that illustrate DeMorgan's theorems.

As stated, DeMorgan's theorems also apply to expressions in which there are more than two variables. The following examples illustrate the application of DeMorgan's theorems to 3-variable and 4-variable expressions.

Example

Apply DeMorgan's theorems to the expressions XYZ and X + Y + z.

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

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

Example

Apply DeMorgan's theorems to the expressions WXYZ and W + X + y + z.

$WXYZ = \overline{W} + \overline{X} + \overline{y} + \overline{Z}$

$\overline{W + X + y + Z} = \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z}$

Applying DeMorgan's Theorems

The following procedure illustrates the application of DeMorgan's theorems and Boolean algebra to the specific expression

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

Step l. Identify the terms to which you can apply DeMorgan's theorems, and think of each term as a single variable. Let A + $\overline{BC}$ = X and D(E + $\overline{F}$) = Y.

Step 2. Since $\overline{X + Y} = \overline{X}\,\overline{Y}$,

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

Step 3. Use rule 9 ($A = \overline{\overline{A}}$) to cancel the double bars over the left term (this is not part of DeMorgan's theorem).

$$(A + \overline{BC})\,(D(E + \overline{F})) = (A + \overline{BC})(D(E + \overline{F}))$$

Step 4. Applying DeMorgan's theorem to the second term,

$$(A + \overline{BC})(D(E + \overline{F})) = (A + \overline{BC})(\overline{D} + (E + \overline{F}))$$

Step 5. Use rule 9 ($A = \overline{\overline{A}}$) to cancel the double bars over the E + F $\overline{\text{part}}$ of the term.

$$(A + \overline{BC})(\overline{D} + \overline{E + F}) = (A + \overline{BC})(\overline{D} + E + \overline{F})$$

Example

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

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

Example

The Boolean expression for an exclusive-OR gate is $A\overline{B} + \overline{A}B$. With this as a starting point, use DeMorgan's theorems and any other rules or laws that are applicable to develop an expression for the exclusive-NOR gate.

BOOLEAN ANALYSIS OF LOGIC CIRCUITS

Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.

Boolean Expression for a Logic Circuit

To derive the Boolean expression for a given logic circuit, begin at the left-most inputs and work toward the final output, writing the expression for each gate. For the example circuit in Fig.(4-16), the Boolean expression is determined as follows:

The expression for the left-most AND gate with inputs C and D is CD. The output of the left-most AND gate is one of the inputs to the OR gate and B is the other input. Therefore, the expression for the OR gate is B + CD.

The output of the OR gate is one of the inputs to the right-most AND gate and A is the other input. Therefore, the expression for this AND gate is A(B + CD), which is the final output expression for the entire circuit.

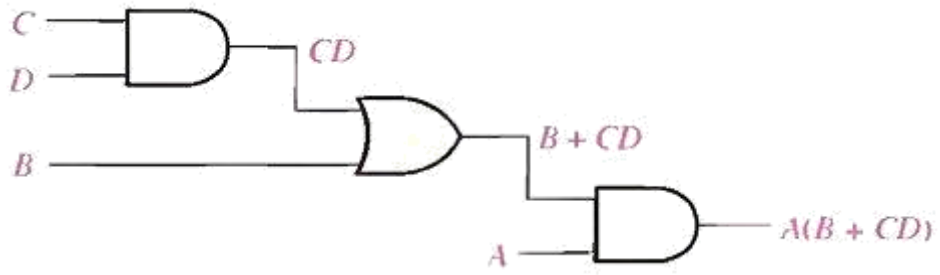


Fig.(4-16) A logic circuit showing the development of the Boolean expression for the output.

Constructing a Truth Table for a Logic Circuit

Once the Boolean expression for a given logic circuit has been determined, a truth table that shows the output for all possible values of the input variables can be developed. The procedure requires that you evaluate the Boolean expression for all possible combinations of values for the input variables. In the case of the circuit in Fig.(4-16), there are four input variables (A, B, C, and D) and therefore sixteen ($2^4 = 16$) combinations of values are possible.

Putting the Results in Truth Table format

The first step is to list the sixteen input variable combinations of 1s and 0s in a binary sequence as shown in Table 4-5. Next, place a 1 in the output column for each combination of input variables that was determined in the evaluation. Finally, place a 0 in the output column for all other combinations of input variables. These results are shown in the truth table in Table 4-5.

Table 4-5

INPUTS				OUTPUT
A	B	C	D	A(B + CD)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

SIMPLIFICATION USING BOOLEAN ALGEBRA

A simplified Boolean expression uses the fewest gates possible to implement a given expression.

Example

Using Boolean algebra techniques, simplify this expression:

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

Solution

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.

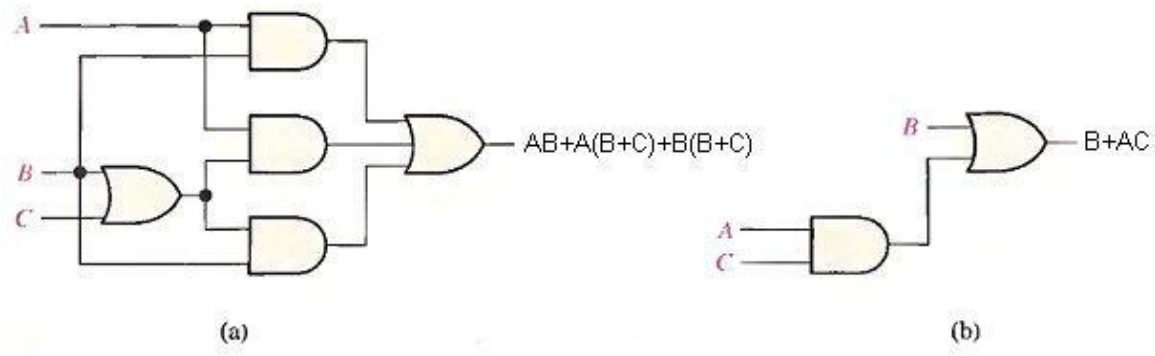


Fig.(4-17) Gate circuits for example above.

Example

Simplify the Boolean expressions:

1- $A\overline{B}$ + A(B + C) + B(B + C).

2- [$A\overline{B}$(C + BD) + $\overline{A}\,\overline{B}$]C

3- $\overline{A}$BC + AB$\overline{C}$ + $\overline{A}\,\overline{B}\,\overline{C}$ + A$\overline{B}$C + ABC

Standard and Canonical Forms:

STANDARD FORMS OF BOOLEAN EXPRESSIONS

All Boolean expressions, regardless of their form, can be converted into either of two standard forms: the sum-of-products form or the product-of-sums form. Standardization makes the evaluation, simplification, and implementation of Boolean expressions much more systematic and easier.

The Sum-of-Products (SOP) Form

When two or more product terms are summed by Boolean addition, the resulting expression is a sum-of-products (SOP). Some examples are:

AB + ABC

ABC + CDE + BCD

AB + BCD + AC

Also, an SOP expression can contain a single-variable term, as in

A + ABC + BCD.

In an SOP expression a single overbar cannot extend over more than one variable.

Example

Convert each of the following Boolean expressions to SOP form:

(a) AB + B(CD + EF)

(b) (A + B)(B + C + D)

(c) (A + B) + C

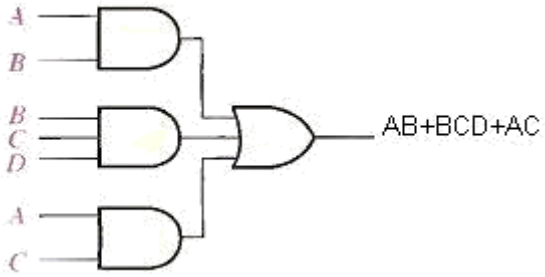


Fig.(4-18) Implementation of the SOP expression AB + BCD + AC.

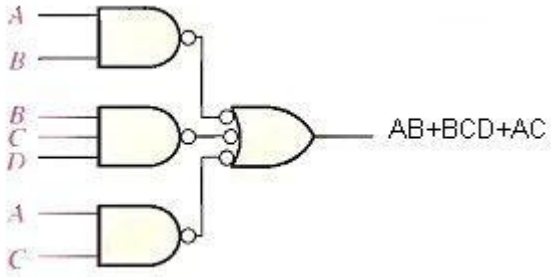


Fig.(4-19) This NAND/NAND implementation is equivalent to the AND/OR in figure above.

The Standard SOP Form

So far, you have seen SOP expressions in which some of the product terms do not contain all of the variables in the domain of the expression. For example, the expression ABC + ABD + ABCD has a domain made up of the variables A, B, C. and D. However, notice that the complete set of variables in the domain is not represented in the first two terms of the expression; that is, D or D is missing from the first term and C or C is missing from the second term.

A standard SOP expression is one in which all the variables in the domain appear in each product term in the expression. For example, ABCD + ABCD + ABCD is a standard SOP expression.

Converting Product Terms to Standard SOP:

Each product term in an SOP expression that does not contain all the variables in the domain can be expanded to standard SOP to include all variables in the domain and their complements. As stated in the following steps, a nonstandard SOP expression is converted into standard form using Boolean algebra rule 6 (A + A = 1) from Table 4-1: A variable added to its complement equals 1.

Step 1. Multiply each nonstandard product term by a term made up of the sum of a missing variable and its complement. This results in two product terms. As you know, you can multiply anything by 1 without changing its value.

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

Example

Convert the following Boolean expression into standard SOP form: ABC + AB + ABCD

Solution

The domain of this SOP expression A, B, C, D. Take one term at a time. The first term, ABC, is missing variable D or D, so multiply the first term by (D + D) as follows:

ABC = ABC(D + D) = ABCD + ABCD

In this case, two standard product terms are the result.

The second term, AB, is missing variables C or C and D or D, so first multiply the second term by C + C as follows:

AB = AB(C + C) = ABC + ABC

The two resulting terms are missing variable D or D, so multiply both terms by (D + D) as follows:

ABC(D + D) + ABC(D + D)

= A BCD + ABCD + ABCD + ABCD

In this case, four standard product terms are the result.

The third term, ABCD, is already in standard form. The complete standard SOP form of the original expression is as follows:

ABC + AB + ABCD = ABCD + ABCD + A BCD + ABCD + ABCD + ABCD + ABCD

The Product-of-Sums (POS) Form

A sum term was defined before as a term consisting of the sum (Boolean addition) of literals (variables or their complements). When two or more sum terms are multiplied, the resulting expression is a product-of-sums (POS). Some examples are

(A + B)(A + B + C)

(A + B + C)(C + D + E)(B + C + D)

(A + B)(A + B + C)(A + C)

A POS expression can contain a single-variable term, as in

A(A + B + C)(B + C + D).

In a POS expression, a single overbar cannot extend over more than one variable; however, more than one variable in a term can have an overbar. For example, a POS expression can have the term A + B + C but not A + B + C.

Implementation of a POS Expression simply requires ANDing the outputs of two or more OR gates. A sum term is produced by an OR operation and the product of two or more sum terms is produced by an AND operation. Fig.(4-

20) shows for the expression (A + B)(B + C + D)(A + C). The output X of the AND gate equals the POS expression.

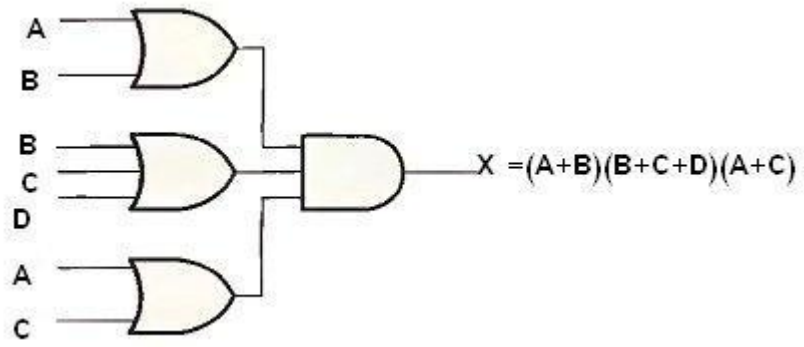


Fig.(4-20)

The Standard POS Form

So far, you have seen POS expressions in which some of the sum terms do not contain all of the variables in the domain of the expression. For example, the expression

(A + B + C) (A + B + D) (A + B + C + D)

has a domain made up of the variables A, B, C, and D. Notice that the complete set of variables in the domain is not represented in e first two terms of the expression; that is, D or D is missing from the first term and C or C is missing from the second term.

A standard POS expression is one in which all the variables in the domain appear in each sum term in the expression. For example,

(A + B + C + D)(A + B + C + D)(A + B + C + D)

is a standard POS expression. Any nonstandard POS expression (referred to simply as POS) can be converted to the standard form using Boolean algebra.

Converting a Sum Term to Standard POS

Each sum term in a POS expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements. As stated in the following steps, a

nonstandard POS expression is converted into standard form using Boolean algebra rule 8 (A A = 0) from Table 4-1:

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

Step 2. Apply rule 12 from Table 4-1: A + BC = (A + B)(A + C)

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

Example

Convert the following Boolean expression into standard POS form: (A + B + C)(B + C + D)(A + B + C + D)

Solution

The domain of this POS expression is A, B, C, D. Take one term at a time. The first term, A + B + C, is missing variable D or D, so add DD and apply rule 12 as follows:

A + B + C = A + B + C + DD = (A + B + C + D)(A + B + C + D)

The second term, B + C + D, is missing variable A or A, so add AA and apply rule 12 as follows:

B + C + D = B + C + D + AA = (A + B + C + D)(A + B + C + D)

The third term, A + B + C + D, is already in standard form. The standard POS form of the original expression is as follows:

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

Examples:-

1. Identify each of the following expressions as SOP, standard SOP, POS, or standard POS:
   (a) $AB + \overline{A}BD + \overline{A}C\overline{D}$ (b) $(A + \overline{B} + C)(A + B + \overline{C})$
   (c) $\overline{A}BC + AB\overline{C}$ (d) $A(A + \overline{C})(A + B)$
2. Convert each SOP expression in Question 1 to standard form.
3. Convert each POS expression in Question 1 to standard form.

CANONICAL FORMS OF BOOLEAN EXPRESSIONS

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

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

Minterms and maxterms are related

- Any minterm m_i is the *complement* of the corresponding maxterm M_i

Minterm	Shorthand	Maxterm	Shorthand
x'y'z'	m_0	x + y + z	M_0
x'y'z	m_1	x + y + z'	M_1
x'yz'	m_2	x + y' + z	M_2
x'yz	m_3	x + y' + z'	M_3
xy'z'	m_4	x' + y + z	M_4
xy'z	m_5	x' + y + z'	M_5
xyz'	m_6	x' + y' + z	M_6
xyz	m_7	x' + y' + z'	M_7

- For example, $m_4' = M_4$ because $(xy'z')' = x' + y + z$

46

For example the function F

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

$F = \bar{x}\,\bar{y}\,z + x\,\bar{y}\,\bar{z} + x\,y\,z$

$F = m_1 + m_4 + m_7$

Any Boolean function can be expressed as a sum of minterms (sum of products **SOP**) or product of maxterms (product of sums **POS**).

$F = x\,y\,z + \bar{x}\,y\,z + x\,y\,z + x\,y\,z + x\,y\,\bar{z}$

The complement of F = F = F

$F = (x + y + z)\,(x + y + z)\,(x + y + \bar{z})\,(x + y + z)\,(x + y + z)$ $F = M_0\,M_2\,M_3\,M_5\,M_6$

Example

Express the Boolean function F = A + BC in a sum of minterms (SOP).

Solution

The term A is missing two variables because the domain of F is (A, B, C)

A = A(B + B) = AB + AB because B + B = 1

$\overline{B}C$ missing A, so

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

$AB(C + \overline{C}) = ABC + AB\overline{C}$

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

$F = ABC + AB\overline{C} + \underline{A\overline{B}C} + A\overline{B}\overline{C} + \underline{A\overline{B}C} + \overline{A}\overline{B}C$

Because A + A = A

$F = ABC + AB\overline{C} + A\overline{B}C + A\overline{B}\overline{C} + \overline{A}\overline{B}C$

$F = m_7 + m_6 + m_5 + m_4 + m_1$

In short notation

$F(A, B, C) = \sum(1, 4, 5, 6, 7)$

$\overline{F}(A, B, C) = \sum(0, 2, 3)$

The complement of a function expressed as the sum of minterms equal to the sum of minterms missing from the original function.

Truth table for $F = A + \overline{B}C$

	A	**B**	**C**	$\overline{\mathbf{B}}$	$\overline{\mathbf{B}}\mathbf{C}$	**F**
0	0	0	0	1	0	0
1	0	0	1	1	1	1
2	0	1	0	0	0	0
3	0	1	1	0	0	0
4	1	0	0	1	0	1
5	1	0	1	1	1	1
6	1	1	0	0	0	1
7	1	1	1	0	0	1

Example

Express F = xy + xz in a product of maxterms form.

Solution

F = xy + xz = (xy + x)(xy + z) = (x + x)(y + x)(x + z)(y + z)

remember x + x = 1

F = (y + x)(x + z)(y + z)

F = (x + y + zz)(x + yy + z)(xx + y + z)

F = (x + y + z)(x + y + z)(x + y + z)(x + y + z)(x + y +z)(x + y + z)

F = (x + y + z)(x + y + z)(x + y + z)(x + y + z)

$F = M_4 M_5 M_0 M_2$

$F(x, y, z) = \prod(0, 2, 4, 5)$

$F(x, y, z) = \prod(1, 3, 6, 7)$

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

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

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

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

Example

Develop a truth table for the standard SOP expression ABC + ABC + ABC.

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

Reca11 that a POS expression is equal to 0 only if at least one of the sum terms is equal to 0. 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. This procedure is illustrated in Example below:

Example

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. The binary values that make the sum terms in the expression equal to 0 are A+ B + C: 000; A + B + C: 010: A + B + C: 011; A + B + C: 10l; and A + 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	

KARNAUGH MAP MINIMIZATION

A Karnaugh map provides a systematic method for simplifying Boolean expressions and, if properly used, will produce the simplest SOP or POS expression possible, known as the minimum expression. As you have seen, the effectiveness of algebraic simplification depends on your familiarity with all the laws, rules, and theorems of Boolean algebra and on your ability to apply them. The Karnaugh map, on the other hand, provides a "cookbook" method for simplification.

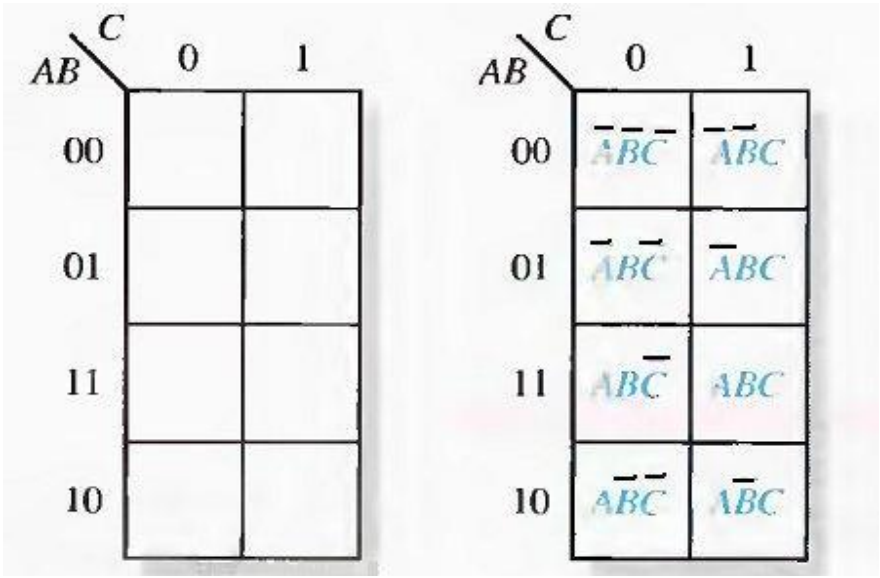
A Karnaugh map is similar to a truth table because it presents all of the possible values of input variables and the resulting output for each value. Instead of being organized into columns and rows like a truth table, the Karnaugh map is an array of cells in which each cell represents a binary value of the input variables. The cells are arranged in a way so that simplification of a given expression is simply a matter of properly grouping the cells. Karnaugh maps can be used for expressions with two, three, four. and five variables. Another method, called the Quine-McClusky method can be used for higher numbers of variables.

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 $2^4 = 16$.

The 3-Variable Karnaugh Map

The 3-variable Karnaugh map is an array of eight cells. as shown in Fig.(5-1)(a). In this case, A, B, and C are used for the variables although other letters could be used. Binary values of A and B are along the left side (notice

the sequence) and the values of C are across the top. The value of a given cell is the binary values of A and B at the left in the same row combined with the value of C at the top in the same column. For example, the cell in the upper left corner has a binary value of 000 and the cell in the lower right corner has a binary value of 101. Fig.(5-1)(b) shows the standard product terms that are represented by each cell in the Karnaugh map.

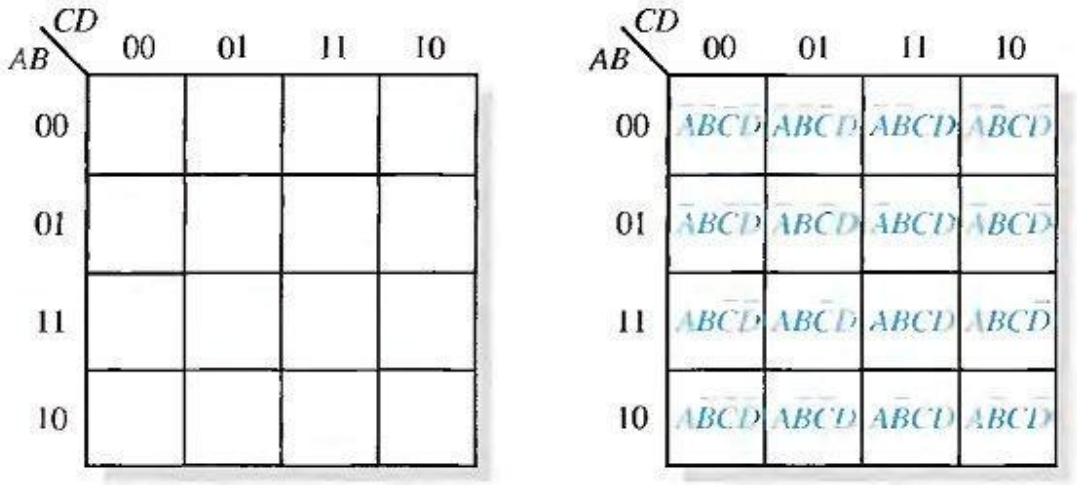


(a) (b) Fig.(5-1) A 3-variable Karnaugh map showing product terms.

The 4-Variable Karnaugh Map

The 4-variable Karnaugh map is an array of sixteen cells, as shown in Fig.(5-2)(a). Binary values of A and B are along the left side and the values of C and D are across the top. The value of a given cell is the binary values of A and B at the left in the same row combined with the binary values of C and D at the top in the same column. For example, the cell in the upper right corner has a binary value of 0010 and the cell in the lower right corner has a

binary value of 1010. Fig.(5-2)(b) shows the standard product terms that are represented by each cell in the 4-variable Karnaugh map.



(a) (b)

Fig.(5-2) A 4-variable Karnaugh map.

Cell Adjacency

The cells in a Karnaugh map are arranged so that there is only a single-variable change between adjacent cells. Adjacency is defined by a single-variable change. In the 3-variable map the 010 cell is adjacent to the 000 cell, the 011 cell, and the 110 cell. The 010 cell is not adjacent to the 001 cell, the 111 cell, the 100 cell, or the 101 cell.

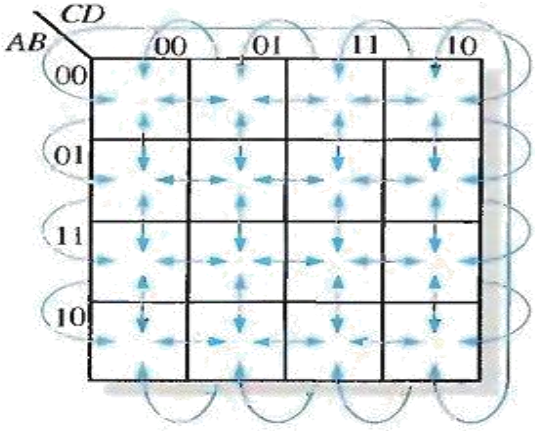


Fig.(5-3) Adjacent cells on a Karnaugh map are those that differ by only one variable. Arrows point between adjacent cells.

KARNAUGH MAP SOP MINIMIZATION

For an SOP expression in standard form, a 1 is placed on the Karnaugh map for each product term in the expression. Each 1 is placed in a cell corresponding to the value of a product term. For example, for the product term ABC, a 1 goes in the 10l cell on a 3-variable map.

Example

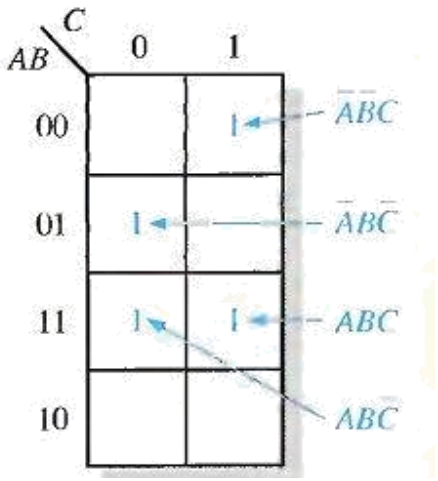
Map the following standard SOP expression on a Karnaugh map:
see Fig.(5-4).

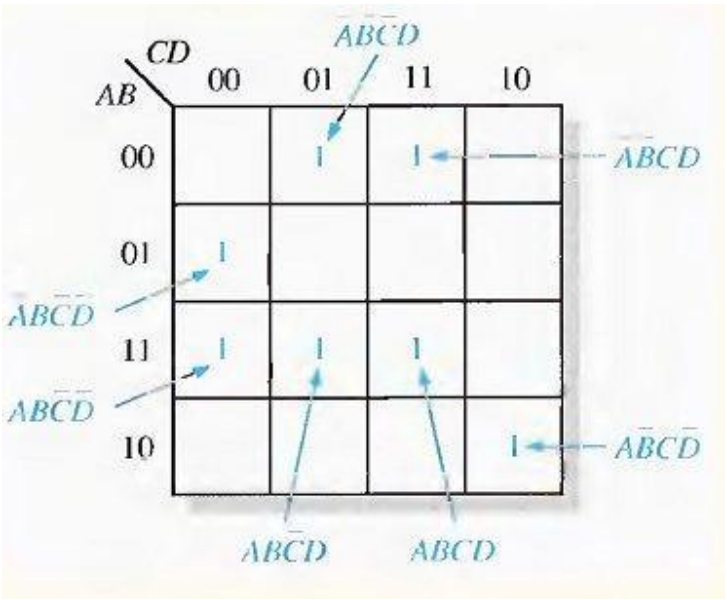
Example

Map the following standard SOP expression on a Karnaugh map:

$$\bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + AB\bar{C}D + ABCD + AB\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + A\bar{B}\bar{C}D$$

See Fig.(5-5).





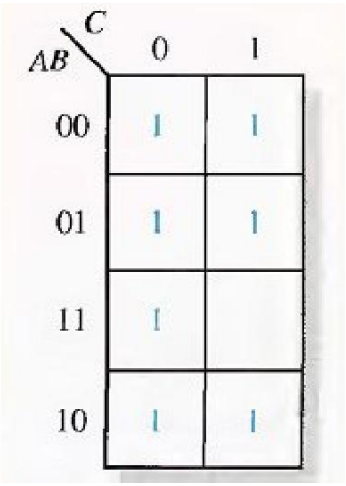
Example

Map the following SOP expression on a Karnaugh map: $\overline{A} + A\overline{B} + AB\overline{C}.$

Solution

The SOP expression is obviously not in standard form because each product term does not have three variables. The first term is missing two variables, the second term is missing one variable, and the third term is standard. First expand the terms numerically as follows:

$\overline{A}$	$+ A\overline{B}$	$+ AB\overline{C}$
000	100	110
001	101	
010		
011		



Example

Map the following SOP expression on a Karnaugh map:

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

Solution

The SOP expression is obviously not in standard form because each product term does not have four variables.

$\overline{B}\,\overline{C}$	$A\overline{B}$	$+ AB\overline{C}$	$+ A\overline{B}C\overline{D}$	$+ \overline{A}\,\overline{B}\,\overline{C}D$	$+ A\overline{B}CD$
0000	1000	1100	1010	0001	1011
0001	1001	1101			
1000	1010				
1001	1011				

Map each of the resulting binary values by placing a 1 in the appropriate cell of the 4- variable Karnaugh map.

Karnaugh Map Simplification of SOP Expressions

Grouping the 1s, you can group 1s on the Karnaugh map according to the following rules by enclosing those adjacent cells containing 1s. The goal is to maximize the size of the

groups and to minimize the number of groups.

A group must contain either 1, 2, 4, 8, or 16 cells, which are all

powers of two. In the case of a 3-variable map, 2^3 = 8 cells is the maximum group.

Each cell in a group must be adjacent to one or more cells in that same group.

Always include the largest possible number of 1s in a group in accordance with rule 1.

Each 1 on the map must be included in at least one group. The 1s already in a group can be included in another group as long as the overlapping groups include noncommon 1s.

Example:

Group the 1s in each of the Karnaugh maps in Fig.(5-6).

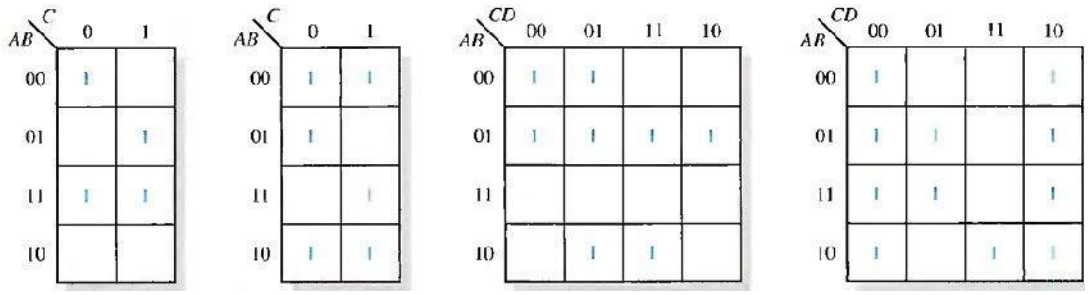


Fig.(5-6)

Solution:

The groupings are shown in Fig.(5-7). In some cases, there may be more than one way to group the 1s to form maximum groupings.

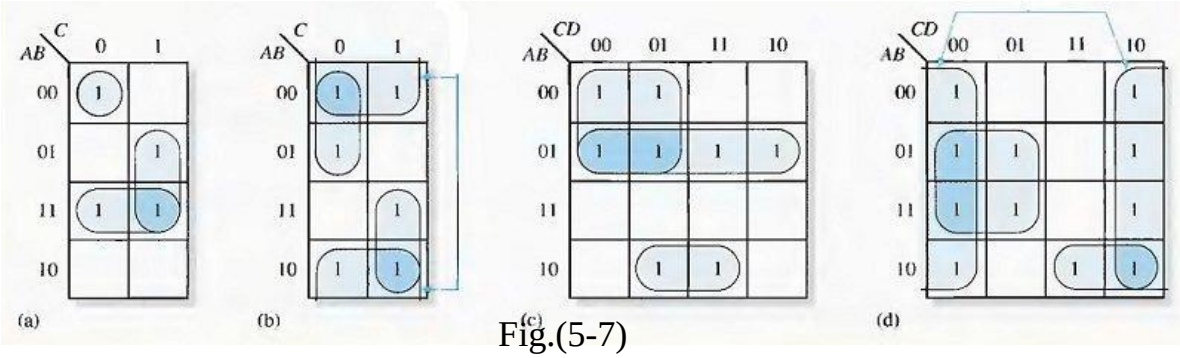


Fig.(5-7)

Determine the minimum product term for each group.

a. For a 3-variable map:

(1) A l-cell group yields a 3-variable product term

(2) A 2-cell group yields a 2-variable product term

(3) A 4-cell group yields a 1-variable term

(4) An 8-cell group yields a value of 1 for the expression

b. For a 4-variable map:

(1) A 1-cell group yields a 4-variable product term

(2) A 2-cell group yields a 3-variable product term

(3) A 4-cell group yields a 2-variable product term

(4) An 8-cell group yields a 1-variable term

(5) A 16-cell group yields a value of 1 for the expression

Example:

Determine the product terms for each of the Karnaugh maps in Fig.(5-7) and write the resulting minimum SOP expression.

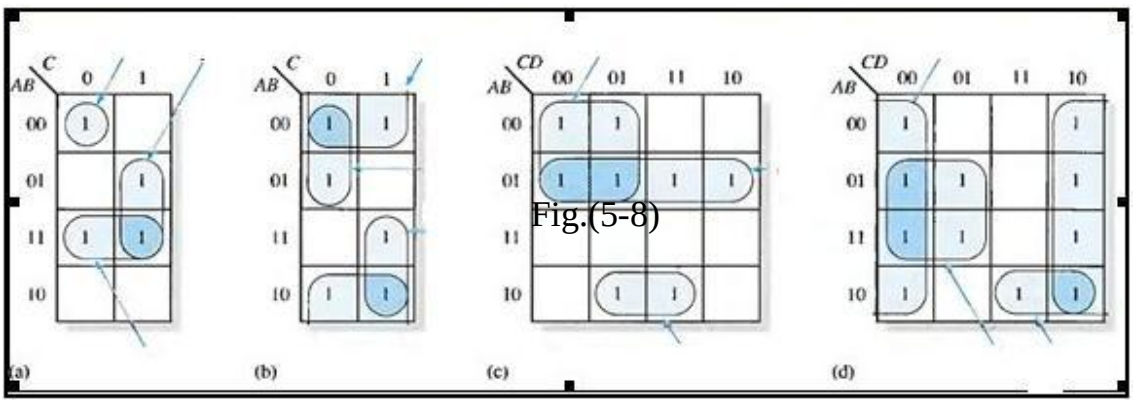


Fig.(5-8)

Solution:

The resulting minimum product term for each group is shown in Fig.(5-8).

The minimum SOP expressions for each of the Karnaugh maps in the figure are:

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

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

Example: Use a Karnaugh map to minimize the following standard SOP expression:

$$ABC + ABC + ABC + ABC + ABC$$

Example: Use a Karnaugh map to minimize the following SOP expression:

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

"Don't Care" Conditions

Sometimes a situation arises in which some input variable combinations are not allowed. For example, recall that in the BCD code there are six invalid combinations: 1010, 1011, 1100, 1101, 1110, and 1111. Since these unallowed states will never occur in an application involving the BCD code, they can be treated as "don't care" terms with respect to their effect on the output. That is, for these "don't care" terms either a 1 or a 0 may be assigned to the output: it really does not matter since they will never occur.

The "don't care" terms can be used to advantage on the Karnaugh map. Fig.(5-9) shows that for each "don't care" term, an X is placed in the cell. When grouping the 1 s, the Xs can be treated as 1s to make a larger grouping or as 0s if they cannot be used to advantage. The larger a group, the simpler the resulting term will be.

The truth table in Fig.(5-9)(a) describes a logic function that has a 1 output only when the BCD code for 7,8, or 9 is present on the inputs. If the "don't cares" are used as 1s, the resulting expression for the function is A + BCD, as indicated in part (b). If the "don't cares" are not used as 1s, the resulting

expression is ABC + ABCD: so you can see the advantage of using "don't care" terms to get the simplest expression.

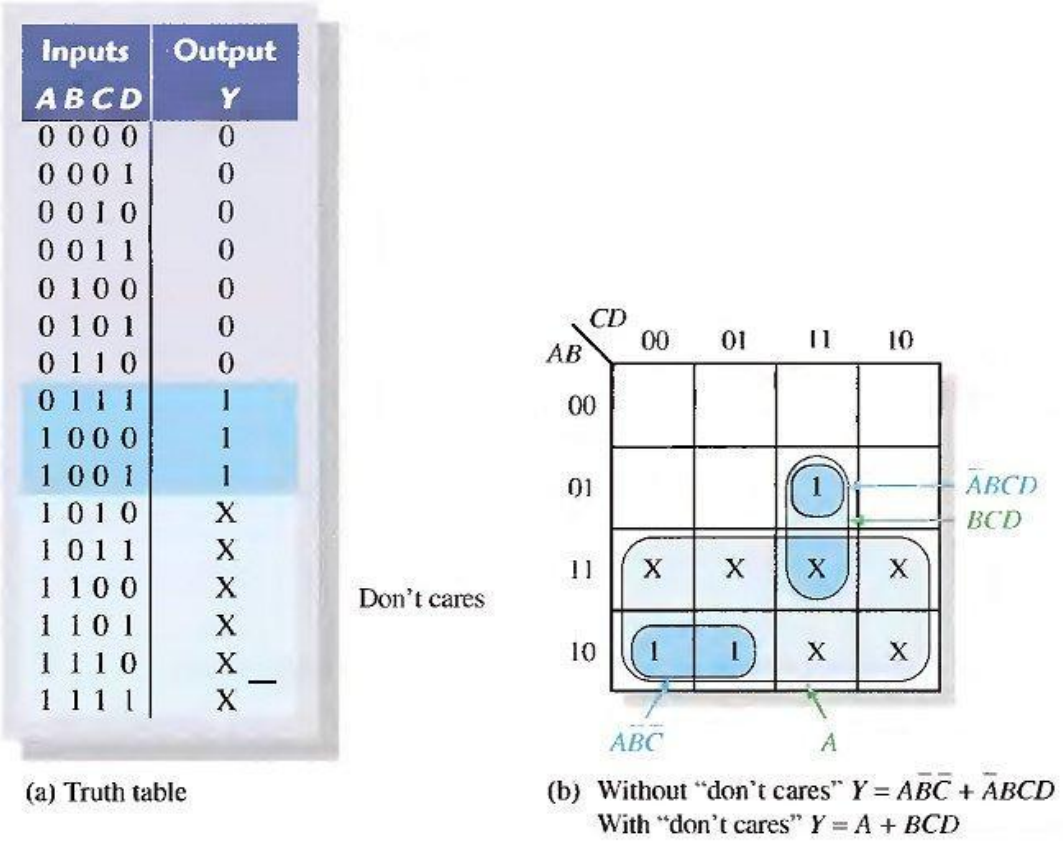


Fig.(5-9)

KARNAUGH MAP POS MINIMIZATION

In this section, we will focus on POS expressions. The approaches are much the same except that with POS expressions, 0s representing the standard sum terms are placed on the Karnaugh map instead of 1s.

For a POS expression in standard form, a 0 is placed on the Karnaugh map for each sum term in the expression. Each 0 is placed in a cell corresponding to the value of a sum term. For example, for the sum term A + B + C, a 0 goes in the 0 1 0 cell

on a 3-variable map.

When a POS expression is completely mapped, there will be a number of 0s on the Karnaugh map equal to the number of sum terms in the standard POS expression. The cells that do not have a 0 are the cells for which the expression is 1. Usually, when working with POS expressions, the 1s are left off. The following steps and the illustration in Fig.(5-10) show the mapping process.

Step 1. Determine the binary value of each sum term in the standard POS expression. This is the binary value that makes the term equal to 0.

Step 2. As each sum term is evaluated, place a 0 on the Karnaugh map in the corresponding cell.

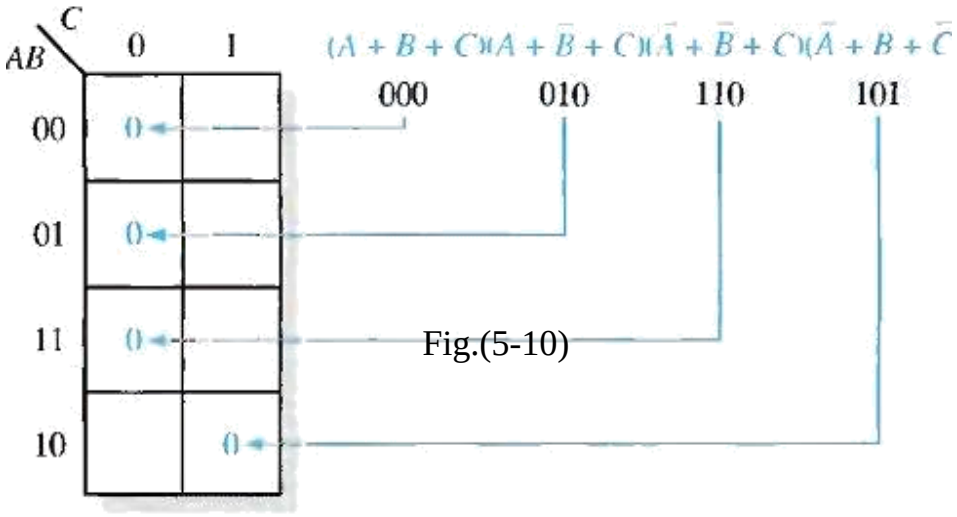


Fig.(5-10)

Example of mapping a standard POS expression.

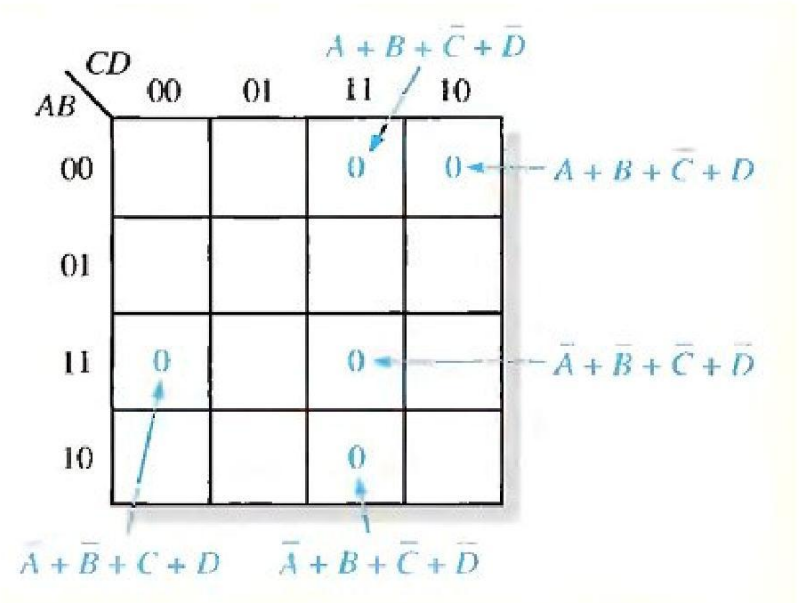
Example:

Map the following standard POS expression on a Karnaugh map:

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

Solution:

$$\underset{1100}{(\bar{A} + \bar{B} + C + D)}\underset{1011}{(\bar{A} + B + \bar{C} + \bar{D})}\underset{0010}{(A + B + \bar{C} + D)}\underset{1111}{(\bar{A} + \bar{B} + \bar{C} + \bar{D})}\underset{0011}{(A + B + \bar{C} + \bar{D})}$$



Karnaugh Map Simplification of POS Expressions

The process for minimizing a POS expression is basically 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. The rules for grouping the 0s are the same as those for grouping the 1s that you learned before.

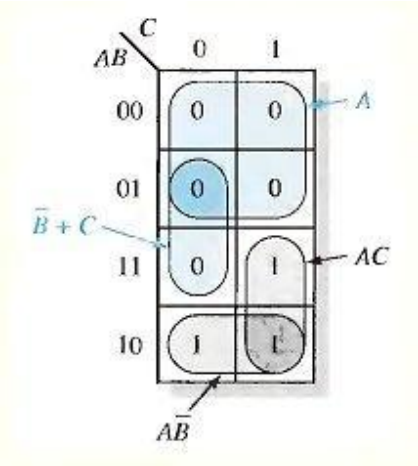
Example:

Use a Karnaugh map to minimize the following standard POS expression:

Also, derive the equivalent SOP expression.

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

Solution:



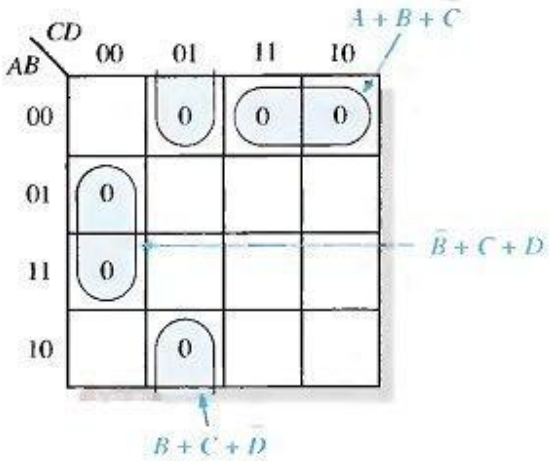
Example: Use a Karnaugh map to minimize the following POS expression:

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

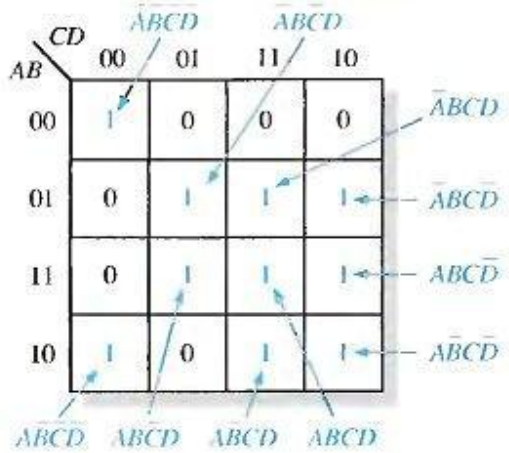
Example: Using a Karnaugh map, convert the following standard POS expression into a minimum POS expression, a standard SOP expression, and

a minimum SOP expression.

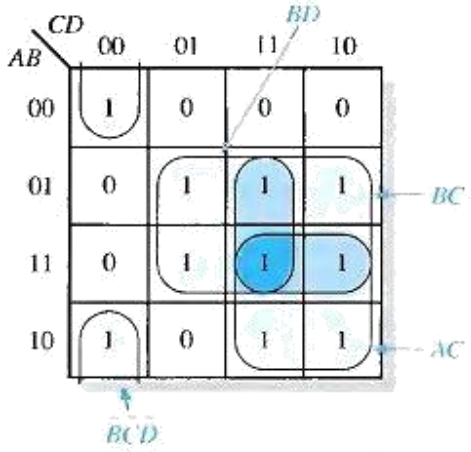
$$(\overline{A} + \overline{B} + C + D)(A + \overline{B} + C + D)(A + B + C + \overline{D})$$
$$(A + B + \overline{C} + \overline{D})(\overline{A} + B + C + \overline{D})(A + B + \overline{C} + D)$$



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



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



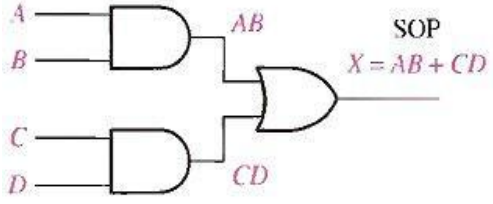
(c) Minimum SOP: $AC + BC + B\bar{D} + \bar{B}\bar{C}\bar{D}$

Implimentation of logical circuit using NAND and NOR gates:

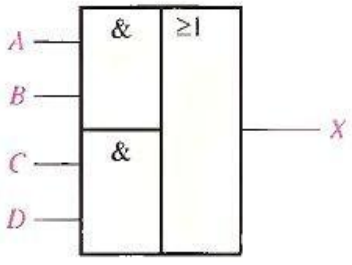
1- AND-OR Logic

Fig.(6-1)(a) shows an AND-OR circuit consisting of two 2-input AND gates and one 2-input OR gate; Fig.(6-1)(b) is the ANSI standard rectangular outline symbol. The Boolean expressions for the AND gate outputs and the resulting SOP expression for the output X are shown in the diagram. In general, all AND-OR circuit can have any number of AND gates each with any number of inputs.

The truth table for a 4-input AND-OR logic circuit is shown in Table 6-1. The intermediate AND gate outputs (AB and CD columns) are also shown in the table.



(a) Logic diagram



(b) ANSI standard rectangular outline symbol.

Fig.(6-1)

For a 4-input AND-OR logic circuit, the output X is HIGH (1) if both input A and input B are HIGH (1) or both input C and input D are HIGH (1).

2-AND-OR-Invert Logic

When the output of an AND-OR circuit is complemented (inverted), it results in an AND-OR-Invert circuit. Recall that AND-OR logic directly implements SOP expressions. POS expressions can be implemented with AND-OR-Invert logic. This is illustrated as follows, starting with a POS expression and developing the corresponding AND-OR-Invert expression.

Table 6-1

INPUTS						OUTPUT
A	B	C	D	AB	CD	X
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1

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

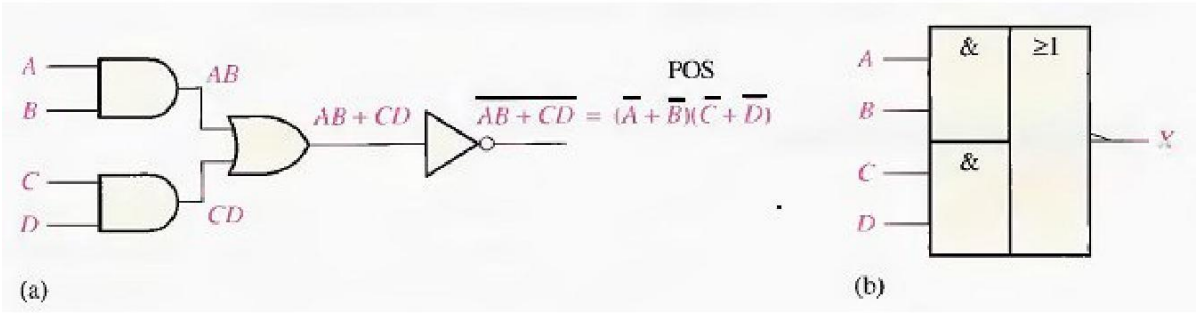


Fig.(6-2)

For a 4-input AND-OR-Invert logic circuit, the output X is LOW (0) if both input A and input B are HIGH (1) or both input C and input D are HIGH (1).

3-Exclusive-OR logic

The exclusive-OR gate was introduced before. Although, because of its importance, this circuit is considered a type of logic gate with its own unique symbol it is actually a combination of two AND gates, one OR gate, and two inverters, as shown in Fig.(6-3)(a). The two standard logic symbols are shown in parts (b) and (c).

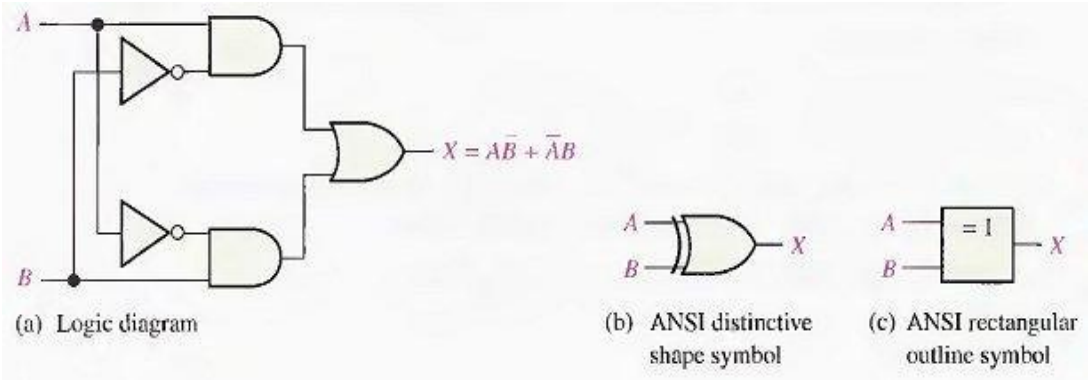


Fig.(6-3)

The output expression for the circuit in Fig.(6-3) is

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

Can be written as $X = A \oplus B$

Table 6-2 Truth table for an exclusive-OR.

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

4- Exclusive-NOR Logic

As you know, the complement of the exclusive-OR function is the exclusive-NOR, whichis derived as follows:

$$X = AB + AB = (AB)\,(AB) = (A + B)(A + B) = AB + AB$$

Notice that the output X is HIGH only when the two inputs, A and B, are at the same level.

The exclusive-NOR can be implemented by simply inverting the output of an exclusive- OR, as shown in Fig(6-4)(a), or by directly implementing the expression AB + AB, as shown in part (b).

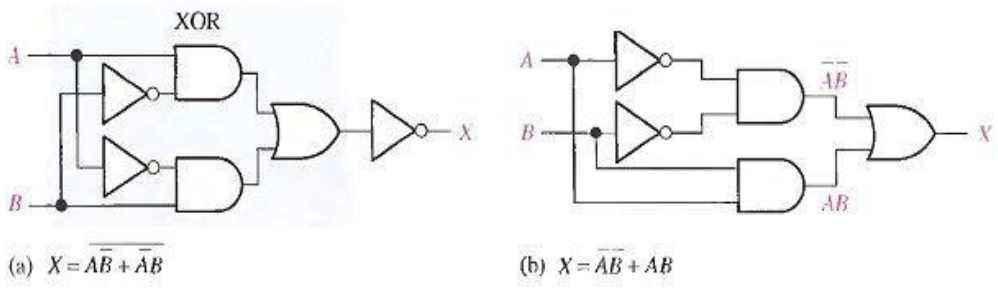


Fig.(6-4)

Example

Develop a logic circuit with four input variables that will only produce a 1 output when exactly three input variables are 1s. Fig.(6-5) shows the circuit.

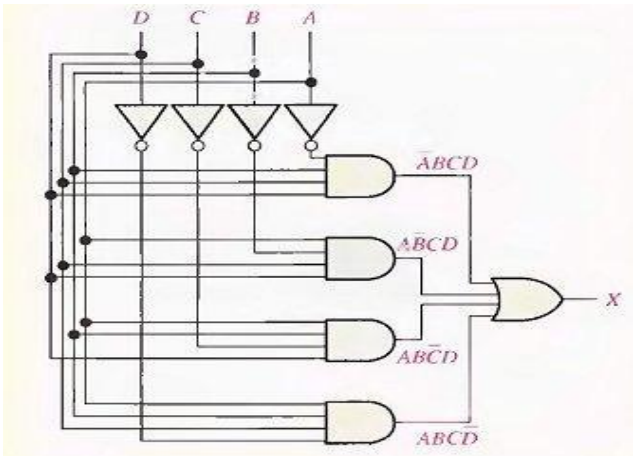


Fig.(6-5)

Example

Reduce the combinational logic circuit in Fig.(6-6) to a minimum form.

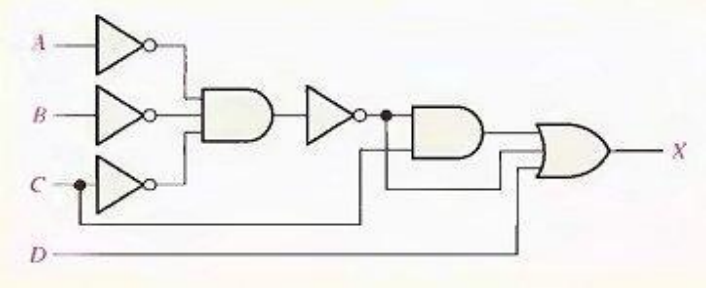


Fig.(6-6)

Solution

The expression for the output of the circuit is

X = (A B C) C + ABC + D

Applying DeMorgan's theorem and Boolean algebra,

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

The simplified circuit is a 4-input OR gate as shown in Fig.(6-7).

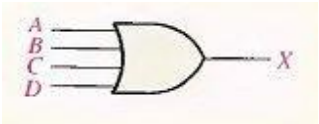
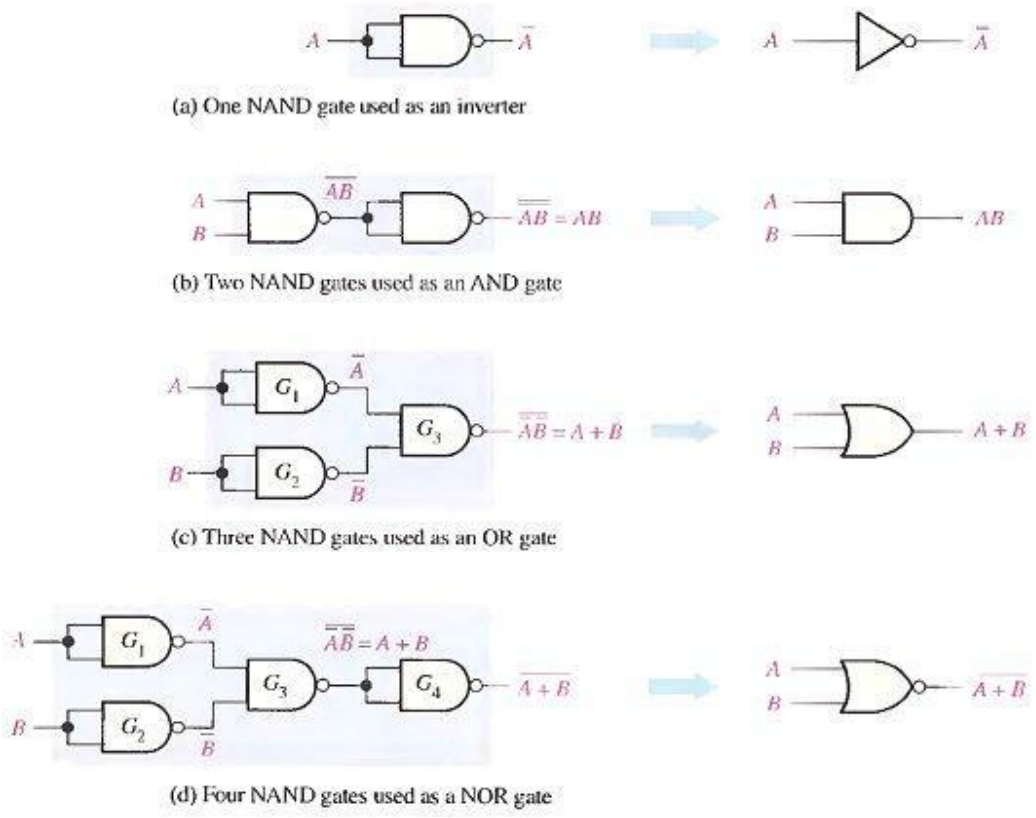


Fig.(6-7)

THE UNIVERSAL PROPERTY OF NAND AND NOR GATES

1- 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, as shown in Fig.(6-8)(a) for a 2-input gate. An AND function can be generated by the use of NAND gates alone, as shown in Fig.(6-8)(b). An OR function can be produced with only NAND gates, as illustrated in part (c). Finally. a NOR function is produced as shown in part (d).



(a) One NAND gate used as an inverter

(b) Two NAND gates used as an AND gate

(c) Three NAND gates used as an OR gate

(d) Four NAND gates used as a NOR gate

Fig.(6-9)

2- 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. A NOT circuit, or inverter, can be made from a NOR gate by connecting all of the inputs together to effectively create a single input, as shown in Fig.(6-10)(a) with a 2-input example. Also, an OR gate can be produced from NOR gates, as illustrated in Fig.(6-10)(b). An AND gate can be constructed by the use of NOR gates, as shown in Fig.(6-10)(c). In this case the NOR gates G 1 and G 2 are used as inverters, and the final output is derived by the use of DeMorgan's theorem as follows:

X=A+B=AB

Fig.(6-10)(d) shows how NOR gates are used t0 form a NAND function.

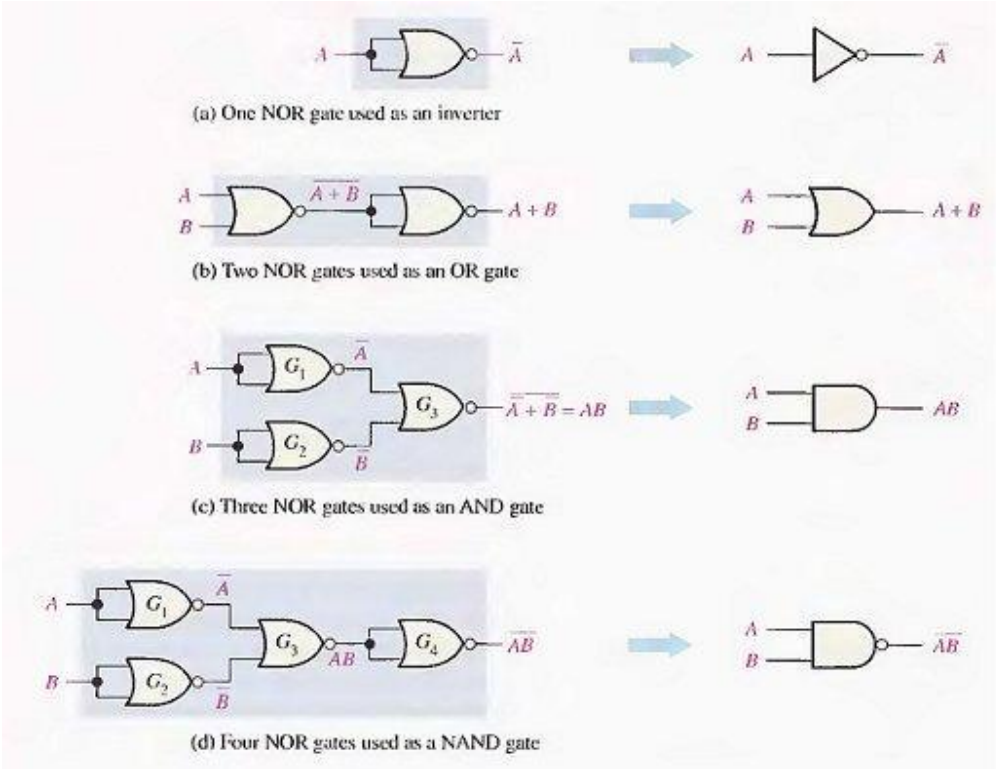


Fig.(6-10)

Example

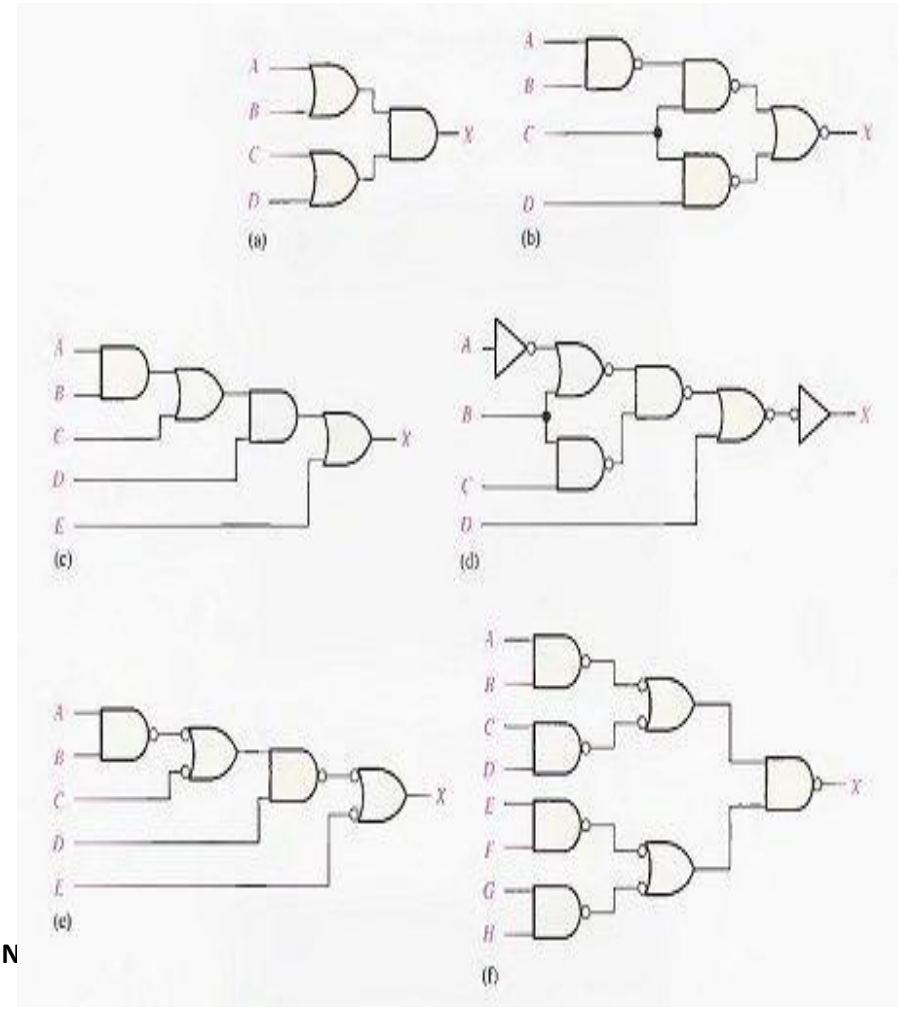
1. Use NAND gates to implement each expression:
   (a) $X = \overline{A} + B$ (b) $X = A\overline{B}$
2. Use NOR gates to implement each expression:
   (a) $X = \overline{A} + B$ (b) $X = A\overline{B}$

Example

1- Write the output expression for each circuit as it appears in Fig.(6-11) and then change each circuit to an equivalent AND-OR configuration.

2- Develop the truth table for circuit in Fig.(6-11)(a-b).

3- Show that an exclusive-NOR circuit produces a POS output.



Integrated Circuits "IC"

- It's a small silicon semiconductor, called a chip, containing the electronic components for the digital gates. The gates are interconnected inside the chip to form the required circuit.

Levels of Integration

- *Small-scale Integration* (SSI): contains several independent gates in a single package. The number of gates is usually fewer than 10.
- *Medium-scale Integration* (MSI): have a complexity between 10 and 100 gates in a single package. They perform specific digital operations such as decoders, adders, and multiplexers.
- *Large-scale Integration* (LSI): contains between 100 and 1000s gates in a single package. The number of gates is usually fewer than 10. They include processors, memory chips, and programmable logic devices.
- *Very Large-scale Integration* (VLSI): contains thousands of gates in a single package. Examples are large memory arrays and complex microcomputer chips.

Basically, there are two types of semiconductor devices: bipolar and unipolar. Based on these devices, digital integrated circuits have been made which are commercially available. Various digital functions are being fabricated in a variety of forms using bipolar and unipolar technologies. A group of compatible ICs with the same logic levels and supply voltages for performing various logic functions have been fabricated using a specific circuit configuration which is referred to as a *logic family*.

The saturated bipolar logic families are:

Resistor–transistor logic (RTL),
Direct–coupled transistor logic (DCTL),
Integrated–injection logic (I^2L),
Diode–transistor logic (DTL),
High–threshold logic (HTL), and
Transistor-transistor logic (TTL).

The non-saturated bipolar logic families are:

(1) Schottky TTL, and
(2)Emitter-coupled logic (ECL).

MOS devices are unipolar devices and only MOSFETs are employed in MOS logic circuits. The

MOS logic families are:

1. PMOS,
2. NMOS and CMOS

CHARACTERISTICS OF DIGITAL ICs

IC Classification	*Equivalent individual basic gates*	*Number of components*
Small-scale integration (SSI)	Less than 12	Up to 99
Medium-scale integration (MSI)	12–99	100–999
Large-scale integration (LSI)	100–999	1,000–9,999
Very large-scale integration (VLSI)	Above 1,000	Above 10,000

The various characteristics of digital ICs used to compare their performances are:

1. Speed of operation,
2. Power dissipation,
3. Figure of merit,
4. Fan-out,
5. Current and voltage parameters,
6. Noise immunity,
7. Operating temperature range,
8. Power supply requirements, and
9. Flexibilities available.

Families of logic gates

There are several different families of logic gates. Each family has its capabilities and limitations, its advantages and disadvantages. The following list describes the main logic families and their characteristics. You can follow the links to see the circuit construction of gates of each family.

- Diode Logic (DL)

Diode logic gates use diodes to perform AND and OR logic functions. Diodes have the property of easily passing an electrical current in one direction, but not the other. Thus, diodes can act as a logical switch.

Diode logic gates are very simple and inexpensive, and can be used effectively in specific situations. However, they cannot be used extensively, as they tend to degrade digital signals rapidly. In addition, they cannot perform a NOT function, so their usefulness is quite limited.

- Resistor-Transistor Logic (RTL)

Resistor-transistor logic gates use Transistors to combine multiple input signals, which also amplify and invert the resulting combined signal. Often an additional transistor is included to re-invert the output signal. This combination provides clean output signals and either inversion or non-inversion as needed.

RTL gates are almost as simple as DL gates, and remain inexpensive. They also are handy because both normal and inverted signals are often available. However, they do draw a significant amount of current from the power supply for each gate. Another limitation is that RTL gates cannot switch at the high speeds used by today's computers, although they are still useful in slower applications.

Although they are not designed for linear operation, RTL integrated circuits are sometimes used as inexpensive small- signal amplifiers, or as interface devices between linear and digital circuits.

- Diode-Transistor Logic (DTL)

By letting diodes perform the logical AND or OR function and then amplifying the result with a transistor, we can avoid some of the limitations of RTL. DTL takes diode logic gates and adds a transistor to the output, in order to provide logic inversion and to restore the signal to full logic levels.

- Transistor-Transistor Logic (TTL)

The physical construction of integrated circuits made it more effective to replace all the input diodes in a DTL gate with a transistor, built with multiple emitters. The result is transistor-transistor logic, which became the standard logic circuit in most applications for a number of years.

As the state of the art improved, TTL integrated circuits were adapted slightly to handle a wider range of requirements, but their basic functions remained the same. These devices comprise the 7400 family of digital ICs.

- Emitter-Coupled Logic (ECL)

Also known as Current Mode Logic (CML), ECL gates are specifically designed to operate at extremely high speeds, by avoiding the "lag" inherent when transistors are allowed to

become saturated. Because of this, however, these gates demand substantial amounts of electrical current to operate correctly.

- CMOS Logic

One factor is common to all of the logic families we have listed above: they use significant amounts of electrical power. Many applications, especially portable, battery-powered ones, require that the use of power be absolutely minimized. To accomplish this, the CMOS (Complementary Metal-Oxide-Semiconductor) logic family was developed. This family uses enhancement-mode MOSFETs as its transistors, and is so designed that it requires almost no current to operate.

CMOS gates are, however, severely limited in their speed of operation. Nevertheless, they are highly useful and effective in a wide range of battery-powered applications.

Most logic families share a common characteristic: their inputs require a certain amount of current in order to operate correctly. CMOS gates work a bit differently, but still represent a capacitance that must be charged or discharged when the input changes state. The current required to drive any input must come from the output supplying the logic signal. Therefore, we need to know how much current an input requires, and how much current an output can reliably supply, in order to determine how many inputs may be connected to a single output.

However, making such calculations can be tedious, and can bog down logic circuit design. Therefore, we use a different technique. Rather than working constantly with actual currents, we determine the amount of current required to drive one standard input, and designate that as a standard load on any output. Now we can define the number of standard loads a given output can drive, and identify it that way. Unfortunately, some inputs for specialized circuits require more than the usual input current, and some gates, known as buffers, are deliberately designed to be able to drive more inputs than usual. For an easy way to define input current requirements and output drive capabilities, we define two new terms: fan-in and fan-out

Fan-in

Fan-in is a term that defines the maximum number of digital inputs that a single logic gate can accept. Most transistor- transistor logic (TTL) gates have one or two inputs, although some have more than two. A typical logic gate has a fan- in of 1 or 2.

In some digital systems, it is necessary for a single TTL logic gate to drive several devices

with fan-in numbers greater than 1. If the total number of inputs a transistor-transistor logic (TTL) device must drive is greater than 10, a device called a buffer can be used between the TTL gate output and the inputs of the devices it must drive. A logical inverter (also called a NOT gate) can serve this function in most digital circuits.

Fan-out

Fan-out is a term that defines the maximum number of digital inputs that the output of a single logic gate can feed. Most transistor-transistor logic (TTL) gates can feed up to 10 other digital gates or devices. Thus, a typical TTL gate has a fan-out of 10.

In some digital systems, it is necessary for a single TTL logic gate to drive more than 10 other gates or devices. When this is the case, a device called a buffer can be used between the TTL gate and the multiple devices it must drive. A buffer of this type has a fan-out of 25 to 30. A logical inverter (also called a NOT gate) can serve this function in most digital circuits.

Remember, fan-in and fan-out apply directly only within a given logic family. If for any reason you need to interface between two different logic families, be careful to note and meet the drive requirements and limitations of both families, within the interface circuitry