**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. Inlogic circuits, a product term is produced by an AND operation with no OR operations involved. Some examples of product terms are AB, AB, 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 ABCD 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

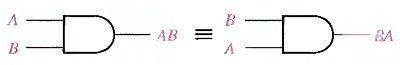
This law states that the order in which the variables are ANDed makes no difference. Fig.(4-2), il1ustrates this law as applied to the AND gate.

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.

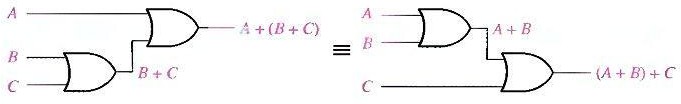


Fig.(4-3) Application of associative law of addition.

* 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.

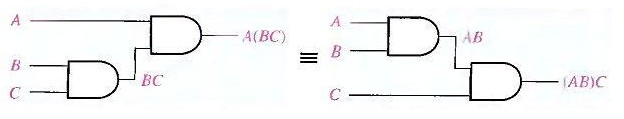


Fig.(4-4) Application of associative law of multiplication.

*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.

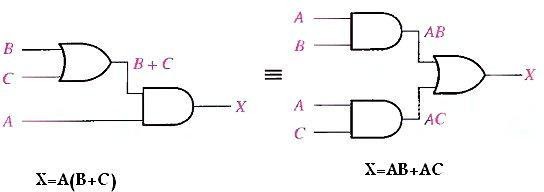
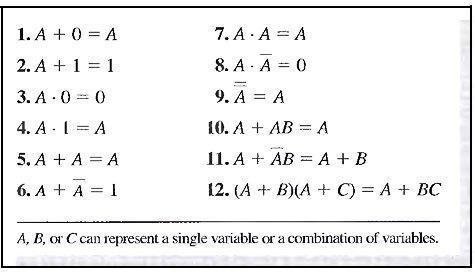


Fig.(4-5) Application of distributive law.

* ***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.



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.

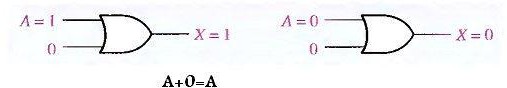


Fig.(4-6)

Rule 2. A + 1 = 1

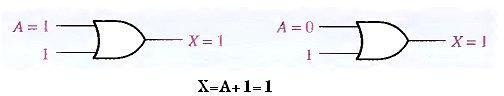
A variable ORed with 1 is always equal to 1. A 1 on an input to an OR gate produces a 1 on the output, regardless of the value of the variable on the other input. This rule is illustrated in Fig.(4-7), where the lower input is fixed at 1.

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.

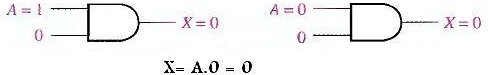


Fig.(4-8)

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.

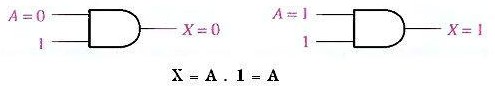


Fig.(4-9)

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.

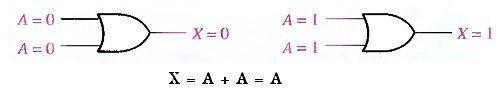


Fig.(4-10)



Rule 6. A + A = 1

A variable ORed with its complement is always equal to 1. If A is 0, then 0 + 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.

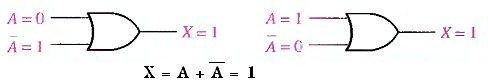


Fig.(4-11)

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.

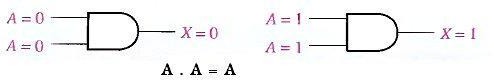


Fig.(4-12)



Rule 8. A . A = 0



A variable ANDed with its complement is always equal to 0. Either A or 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.

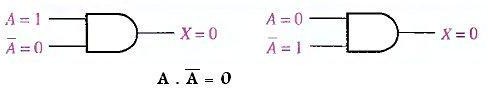


Fig.(4-13)



Rule 9 A = 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 A. If you then take A and complement (invert) it, you get A, which is the original variable. This rule is shown in Fig.(4-14) using inverters.

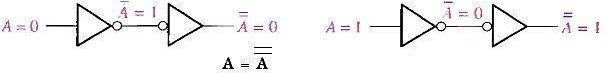


Fig.(4-14)

Rule 10. A + AB = A

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

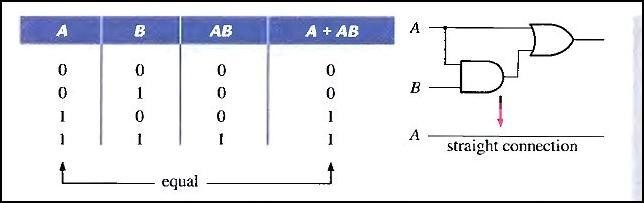
A + AB = A( 1 + B) Factoring (distributive law)

= 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





Rule 11. A + AB = A + B

This rule can be proved as follows:



A + AB = (A + AB) + AB Rule 10: A = A + AB



= (AA + AB) + AB Rule 7: A = AA



=AA +AB +AA +AB Rule 8: adding AA = 0



= (A + A)(A + B) Factoring

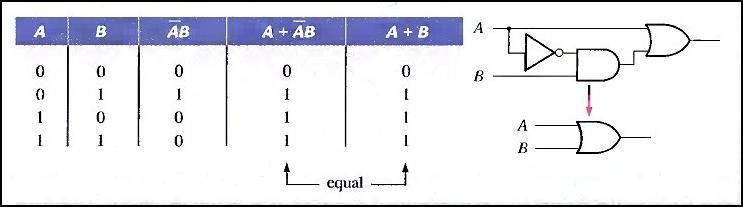


= 1. (A + B) Rule 6: A + A = 1

=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



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

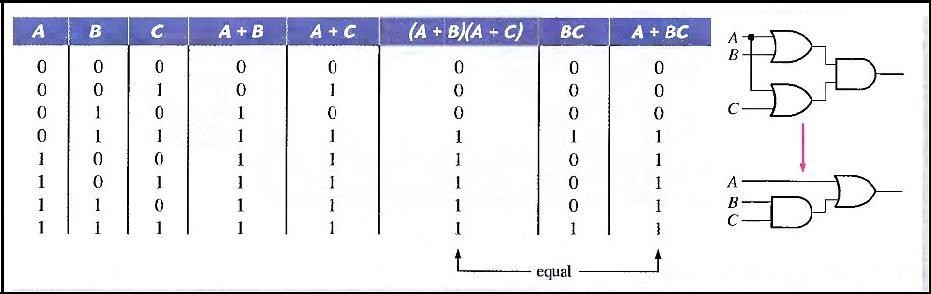
= A. 1 + BC Rule 4: A . 1 = A

##### = A + BC

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

simplification.

Table 4-4



# 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 = X + 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 X + Y = X Y

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

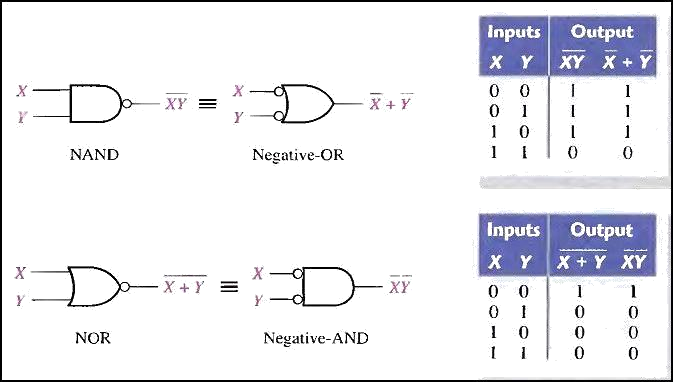


Fig.(4-15) Gate equivalencies and the corresponding truth tables that illustrate DeMorgan's theorems.

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 = X + Y + Z

X + y + Z = X Y Z

Example

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



WXYZ = W + X + y + Z



W + X + y + Z = W X Y Z

*Applying DeMorgan's Theorems*

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

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



Step 2. Since X + Y = X Y,



##### = (A + BC) (D(E + F))



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



##### (A + BC) (D(E + F)) = (A + BC)(D(E + F ))

Step 4. Applying DeMorgan's theorem to the second term, (A + BC)(D(E + F)) = (A + BC)(D + (E + F ))

Step 5. Use rule 9 (A = A) to cancel the double bars over the E + F part of the term.

##### (A + BC)(D + E + F) = (A + BC)(D + E + F)

Example

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



(a) (A + B + C)D (b) ABC + DEF (c) AB + CD + EF

Example



The Boolean expression for an exclusive-OR gate is AB + AB. 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.

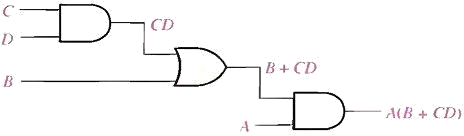


Fig.(4-16) A logic circuit showing the development of the Boolean expression for the output.

#### Constructing a Truth Table for a Logic Circuit

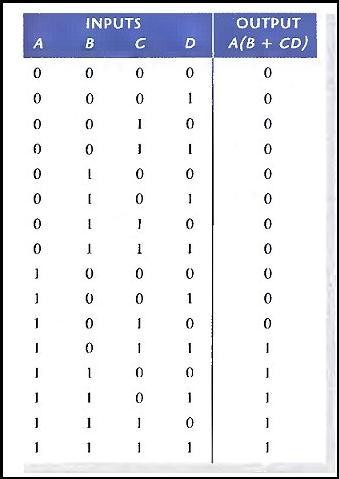
Once the Boolean expression for a given logic circuit has been determined, a truth table that shows the output for all possible values of the input variables can be developed. The procedure requires that you evaluate the Boolean expression for all possible combinations of values for the input variables. In the case of the circuit in Fig.(4-16), there are four input variables (A, B, C, and D) and therefore sixteen (24 = 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**



**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.

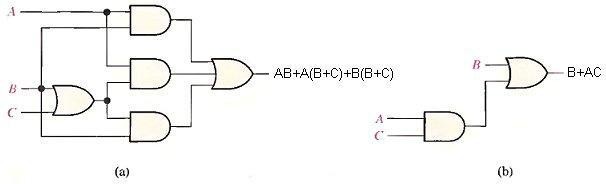


Fig.(4-17) Gate circuits for example above.

Example

Simplify the Boolean expressions:



##### 1- AB + A(B + C) + B(B + C).



1. [AB( C + BD) + A B]C

##### ABC + ABC + A B C + ABC + 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:

1. AB + B(CD + EF)
2. (A + B)(B + C + D)
3. (A + B) + C

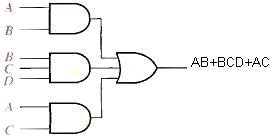


Fig.(4-18) Implementation of the SOP expression AB + BCD + AC.

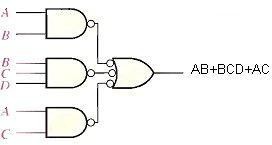


Fig.(4-19) This NAND/NAND implementation is equivalent to the AND/OR in figure above.

*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-

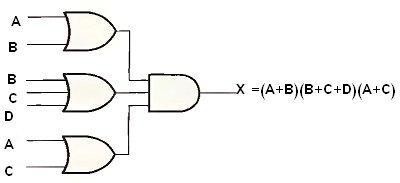
20) shows for the expression (A + B)(B + C + D)(A + C). The output X of the AND gate equals the POS expression.

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:

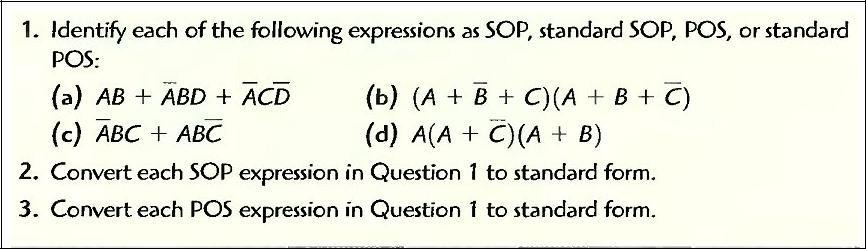
##### A + B + C = A + B + C + DD = (A + B + C + D)(A + B + C + D)

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:

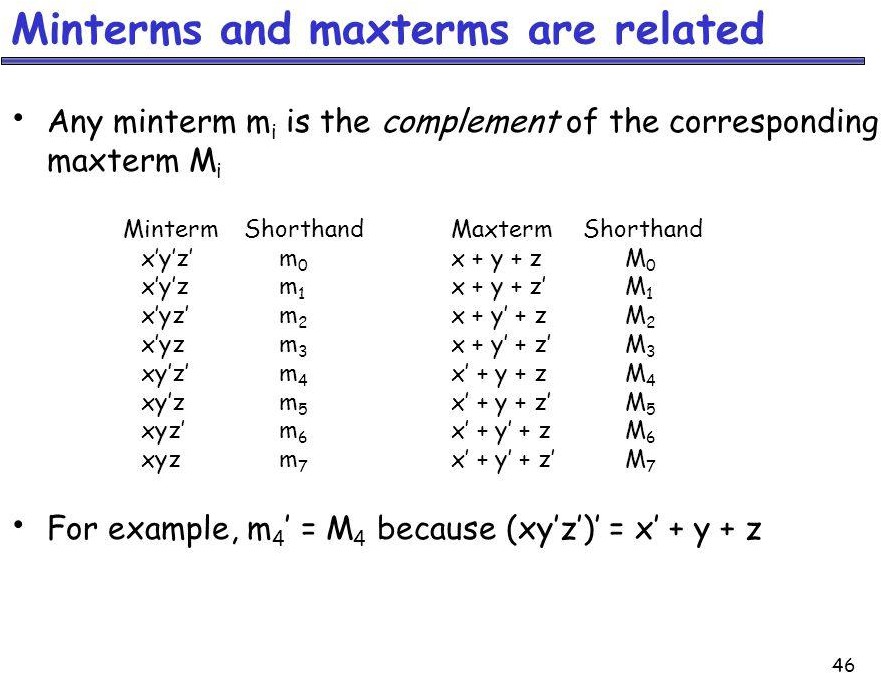
##### (A + B + C)(B + C + D)(A + B + C + D) = (A + B + C + D)(A + B + C + D) (A + B + C + D)(A + B + C + D) (A + B + C + D)

Examples:-

#### CANONICAL FORMS OF BOOLEAN EXPRESSIONS

n variables can be combined to form 2n minterms.

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



For example the function F

x y z F

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



F = x y z + x y z + x y z F = m1 + m4 + m7

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 + x y z + x y z + x y z + x y z

The complement of F = F = F



F = (x + y + z) (x + y + z) (x + y + z) (x + y + z) (x + y + z) F = M0 M2 M3 M5 M6

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

BC missing A, so

BC(A + A) = ABC + ABC AB(C + C) = ABC + ABC AB(C + C) = ABC + ABC

F = ABC + ABC + ABC + ABC + ABC + ABC



Because A + A = A

F = ABC + ABC + ABC + ABC + ABC

F = m7 + m6 + m5 + m4 + m1

In short notation

F(A, B, C) = ∑(1, 4, 5, 6, 7)

F(A, B, C) = ∑(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 + BC

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **B** |  | **BC** | **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 = M4 M5 M0 M2

F(x, y, z) = ∏(0, 2, 4, 5)

F(x, y, z) = ∏(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.

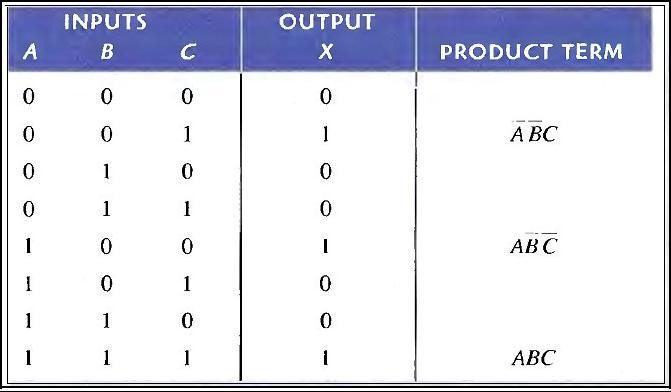
To convert from one canonical form to another, interchange the symbols ∑,

∏ and list those numbers missing from the original form.

F = M4 M5 M0 M2 = m1 + m3 + m6 + m7 F(x, y, z) = ∏(0, 2, 4, 5) = ∑(1, 3, 6, 7)

Example

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

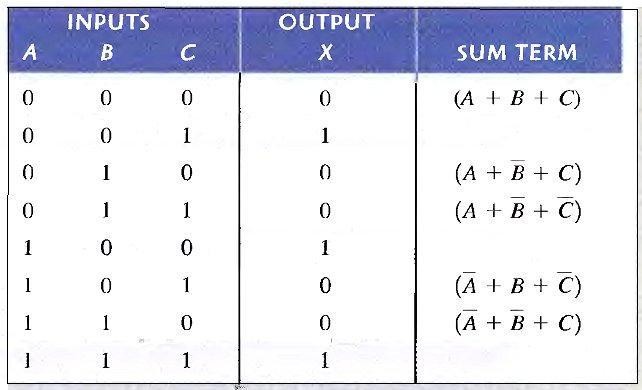
Example

Determine the truth table for the following standard POS expression:



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.



**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.

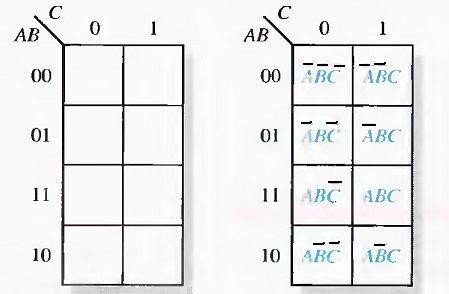
The number of cells in a Karnaugh map is equal to the total number of possible input variable combinations as is the number of rows in a truth table. For three variables, the number of cells is 23 = 8. For four variables, the

number of cells is 24 = 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.

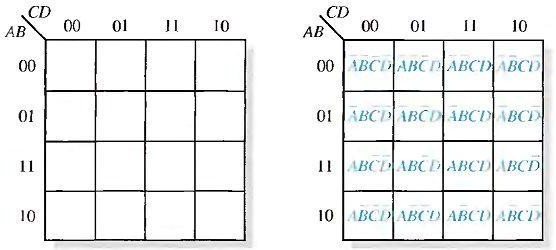


(a) (b) Fig.(5-1) A 3-variable Karnaugh map showing product terms.

***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.



*Cell Adjacency*

(a) (b)

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

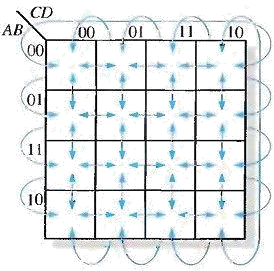
The cells in a Karnaugh map are arranged so that there is only a single- variable change between adjacent cells. Adjacency is defined by a single- variable change. In the 3-variable map the 010 cell is adjacent to the 000 cell, the 011 cell, and the 110 cell. The 010 cell is not adjacent to the 001 cell,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

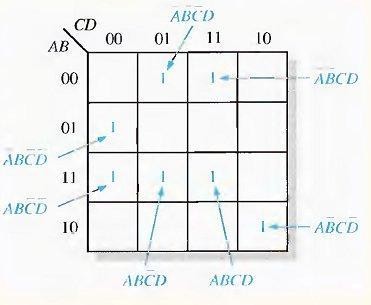
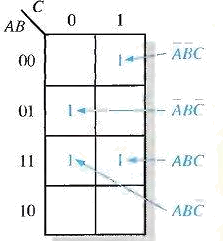
Map the following standard SOP expression on a Karnaugh map: see Fig.(5-4).

Example

Map the following standard SOP expression on a Karnaugh map:



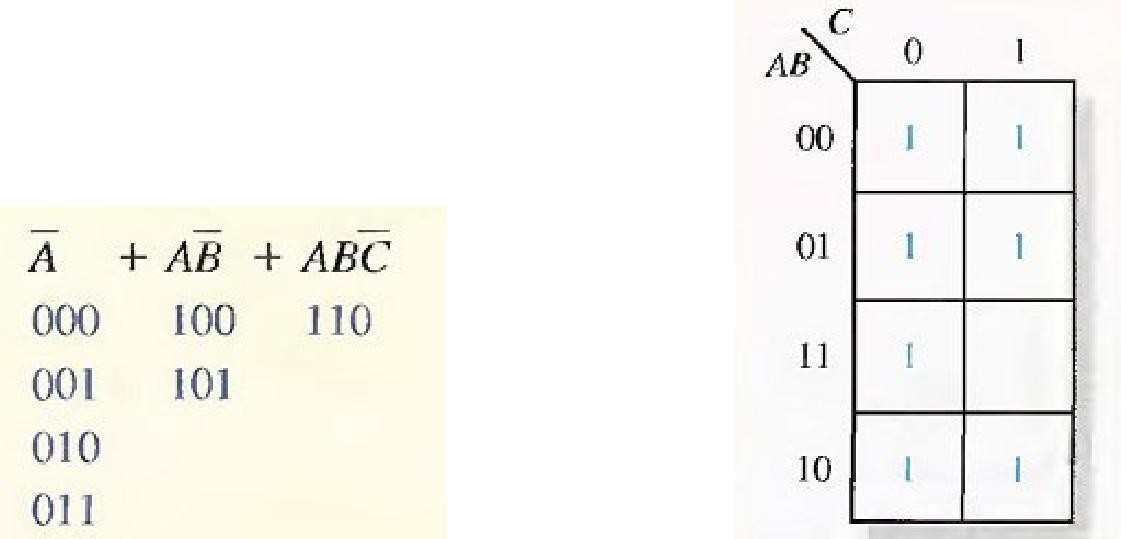
See Fig.(5-5).



Example

Map the following SOP expression on a Karnaugh map: 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:

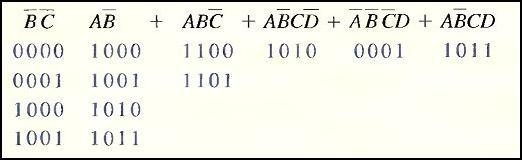
Example

Map the following SOP expression on a Karnaugh map:



Solution

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



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, 23 = 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).

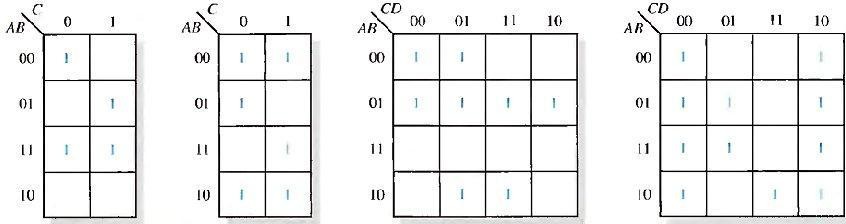


Fig.(5-6)

Solution:

**DSD UNIT 2 NOTES**

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.

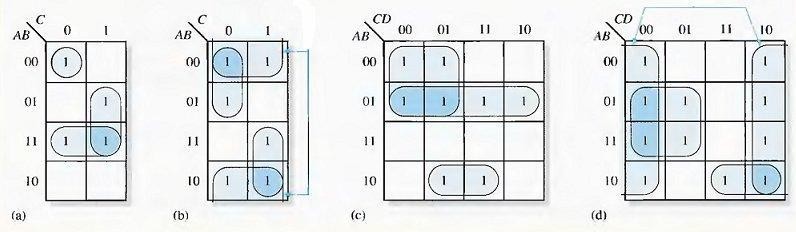


Fig.(5-7)

Determine the minimum product term for each group.

1. 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
2. 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.

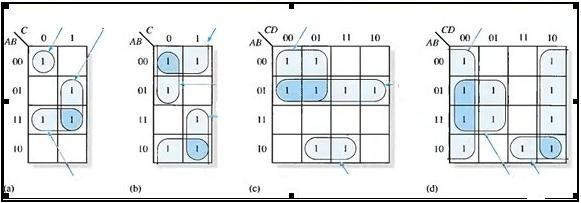


Fig.(5-8)

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:



* + 1. AB+BC+ABC (C) AB + AC + ABD



* + 1. B + A C + AC (d) D + ABC + BC

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:



**"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

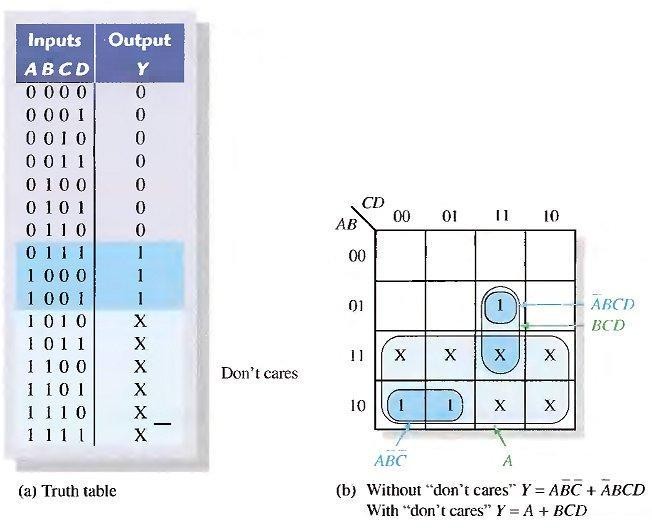
expression is ABC + ABCD: so you can see the advantage of using "don't care" terms to get the simplest expression.

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.

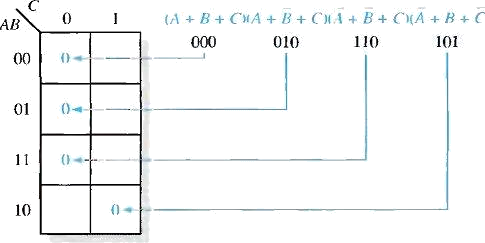
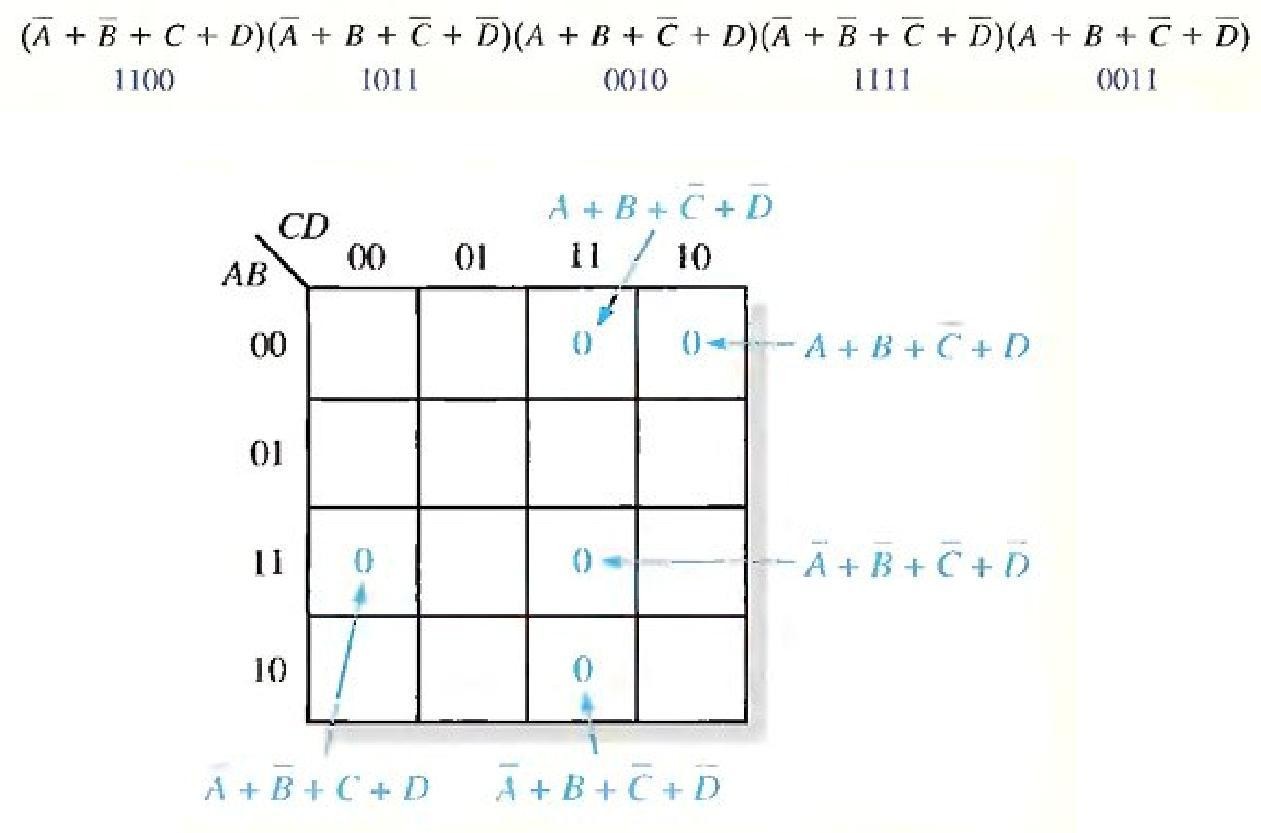


Fig.(5-10)

Example of mapping a standard POS expression.

Example:

Map the following standard POS expression on a Karnaugh map: Solution:

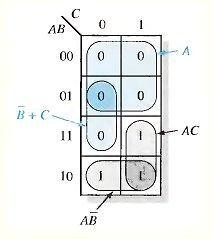
## Karnaugh Map Simplification of POS Expressions

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.

Solution:

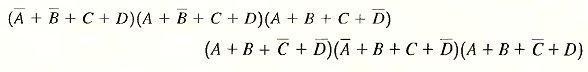


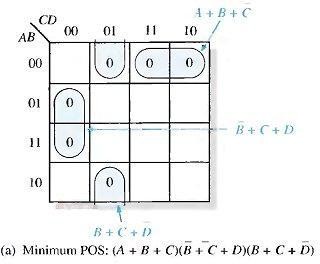
Example: Use a Karnaugh map to minimize the following POS expression:

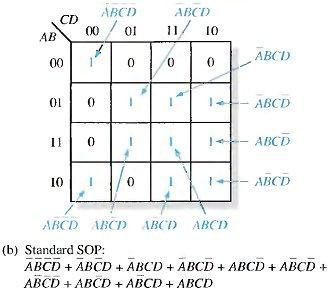


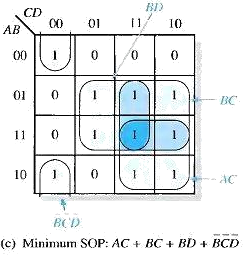
Example: Using a Karnaugh map, convert the following standard POS expression into a minimum POS expression, a standard SOP expression, and

a minimum SOP expression.







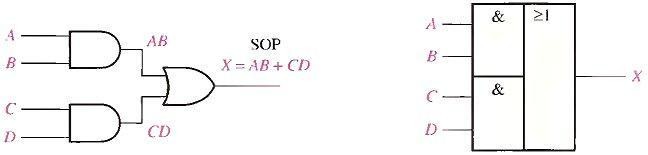


# Implimentation of logical circuit using NAND and NOR gates:

## 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.**

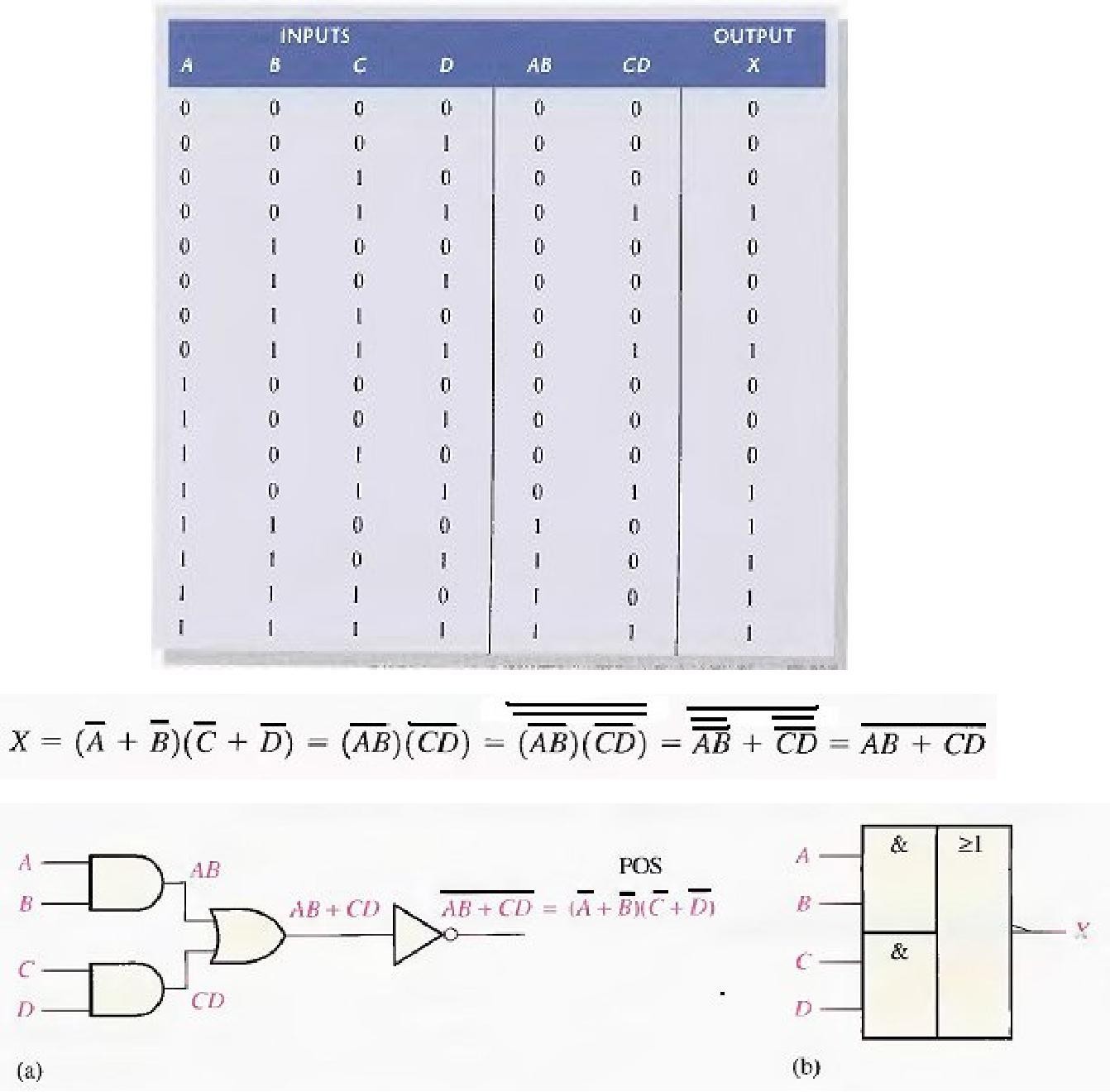
**Fig.(6-1)**

**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).**

## 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**

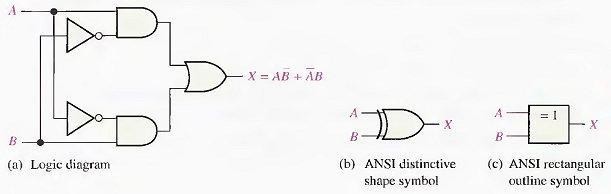


**Fig.(6-2)**

**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).**

## 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).



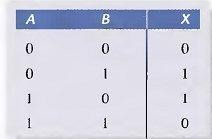
**Fig.(6-3)**

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



Can be written as 

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



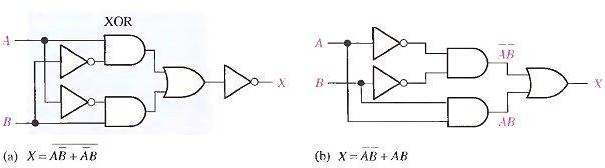
## Exclusive-NOR Logic

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.

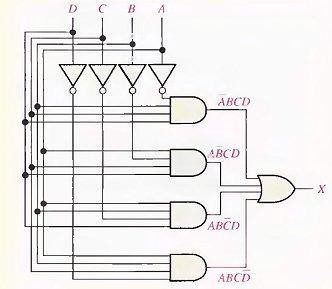
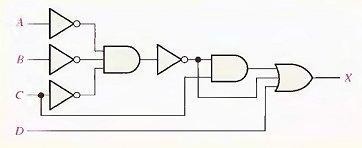


Fig.(6-5)

**Example**

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



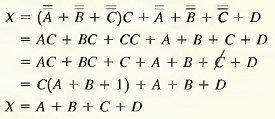
**Fig.(6-6)**

**Solution**

The expression for the output of the circuit is

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

Applying DeMorgan's theorem and Boolean algebra,



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

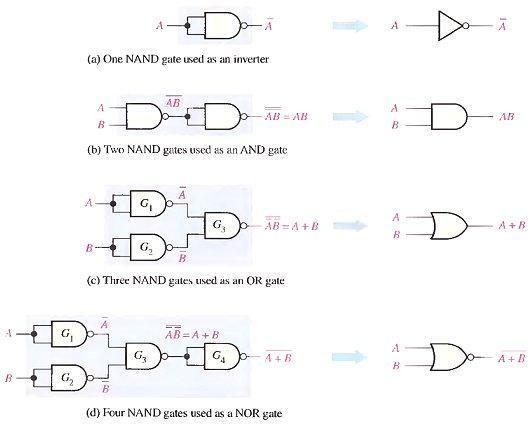


**Fig.(6-7)**

## 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).



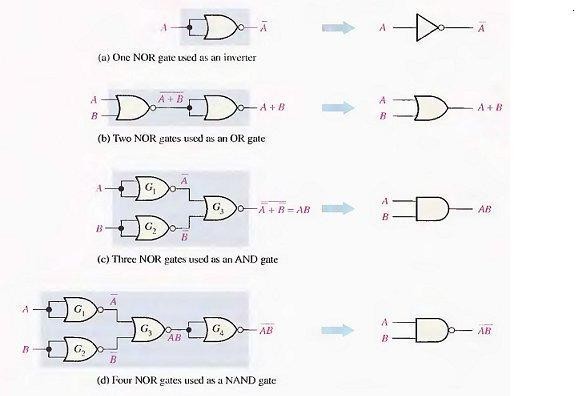
**Fig.(6-9)**

1. The NOR Gate as a Universal Logic Element

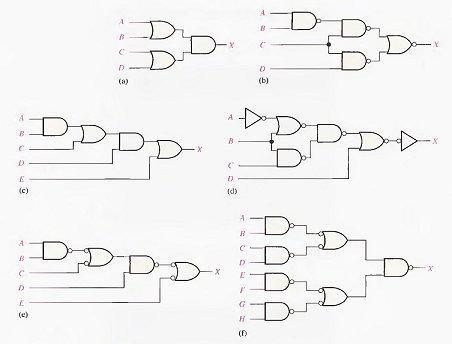
Like the NAND gate, the NOR gate can be used to produce the NOT, AND. OR and NAND functions. A NOT circuit, or inverter, can be made from a NOR gate by connecting all of the inputs together to effectively create a single input, as shown in Fig.(6-10)(a) with a 2-input example. Also, an OR gate can be 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

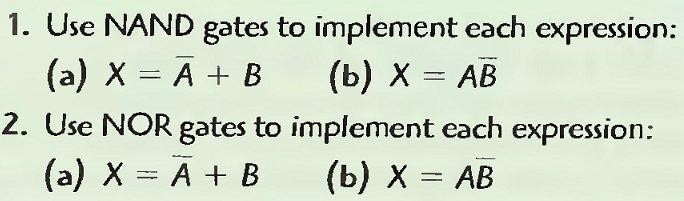
Fig.(6-10)(d) shows how NOR gates are used t0 form a NAND function.



**Fig.(6-10)**

**Example**

**NIT 2 NOTES**



**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.**

# Registers : These are Memory devices which accepts, stores and transfer the data.

# Accumulator Register(AC)

# Address Register(AR)

# Data Register(DR)

# Instruction Register(IR)

# Input Register(IR)

# Program Counter(PC)

# Temporary Register(TR)

# Output Register

# Functions Of Registers in Computer Architecture

# Data Storage: Registers are used for the temporary storage of data or instructions during the execution of the program. They are mainly used to store information for a small time so that the CPU can access it quickly.

# Data Processing: Various logical and arithmetic operations are performed with the help of a register..

# Addressing: Registers are used to store memory addresses, they can store the address of the instruction or the data that the CPU wants to access either to read from or write to.

# Data Transfer: Registers are used to transfer data between CPU and memory. Some of them hold the data that is read from or written to memory.

# Control: The control of the order of execution of instructions is handled by some registers as registers can store the current instruction and some will store the next instruction.

# Representing instructions

Computers must have instructions capable of performing the following four types of operations:

* + Data transfers between the memory and the processor registers
  + Arithmetic and logic operations on data
  + Program sequencing and control
  + I/O transfers

**Instruction Types**

Instructions are divided into the following five types:

* + ***Data-transfer instruction***, which copy information from one location to another location either in the processor’s internal register set or in the external main memory.
* *Operation: MOV, LOAD, STORE, XCH, PUSH, POP*
* *Eg: MOV A, R1 LOAD A STORE T XCH A PUSH A POPA*
  + ***Arithmetic instructions***, which perform operations on numerical data.
* Operation: ADD, ADD WITH CARRY,SUBTRACT,MULTIPLY, DIV
* *Eg: ADDA, B ADDC A,B SUBA,B MULA,B*
  + ***Logical instruction***s, which include Boolean and other non-numerical Operations.
* Operation: AND,OR,NOT,XOR, SHIFTLEFT, SHIFTRIGHT
* *Eg: ANDA, B OR A,B NOT A,B SHLA SHR A*
  + ***Program control instructions***, such as branch instruction, which change the sequence in which programs are executed.
* Operation: JUMP, RETURN, EXECUTE, SKIP, CONDITIONAL COMPARE, TEST,WAIT
* *Eg: JMP CALL RET SKP TST*
  + ***Input-output (IO) instructions***, which cause information to be transferred between the processor or its main memory and external IO devices.
* Operation: INPUT, OUTPUT, START IO, TEST IO, HALT IO

**load** means a transfer of information from memory up to the register,

**store** represents putting the information back to memory-chips.

**Instruction Set and Instruction Sequencing:**

**A computer must have instructions capable of performing 4 types of operations:**  
1) Data transfers between the memory and the registers (MOV, PUSH, POP, XCHG).  
2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT).  
3) Program sequencing and control(CALL.RET, LOOP, INT).  
4) I/0 transfers (IN, OUT).

**REGISTER TRANSFER NOTATION (RTN)**  
Here we describe the transfer of information from one location in a computer to another.Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locationssymbolically with convenient names.  
• The possible locations in which transfer of information occurs are:  
1) Memory-location  
2) Processor register &  
3) Registers in I/O device.

We can represent the instructions using the following ways:

**Register Transfer Notation**

* + Transfer of information from one location to another.
  + Possible locations involved:
* Memory locations
* Processor registers
* Registers in I/O subsystems
  + Contents of a location are denoted by placing square brackets around the name of the location
* R1←[LOC]
* R3 ←[R1]+[R2]
  + Register Transfer Notation(RTN):
* RHS→Value, LHS→location where value is to be stored.

**Assembly Language Notation**

* + Represent machine instructions and programs.
  + Assembly language format.
* Move LOC R1 = R1←[LOC]
* Add R1, R2, R3
* R3 ←[R1]+[R2]

## Basic Instruction Types/Instruction Formats:

**C = A + B**

**Three address instructions**

* + Syntax: opcode source1, source2, destination
  + Eg: ADD A,B, C
  + C ->[A]+[B]

**Two address instructions**

* + Syntax: opcode source, destination
  + Eg: ADD A, B
  + B [A]+[B]
  + MOVB,C C -> [B]
  + Need to add something to the above two-address instruction to finish: Move B, C = C ← [B]

**One-address instruction (to fit in one word length)**

* + **Syntax: opcode source**
  + Second address is implicit [accumulator]
  + Eg: Accumulator:
  + ADD A
  + LOAD A (copies content of memory location A to accumulator)
  + ACC [A]
  + ADD B
  + ACC [B]+[ACC]
  + STORE C (copies content of accumulator to memory location C)
  + C [ACC]

**Zero-address instructions stack operation**

* + Syntax: opcode
  + All addresses are implicit.
  + Eg: PUSH A
  + PUSH B ADD POP C

**Example: Evaluate X=(A+B) \* (C+D)**

*Three-Address*

* + 1. ADD A, B, R1; R1 ← M[A] + M[B]
    2. ADD C, D, R2 ; R2 ← M[C] + M[D]

3. MUL R1, R2, X; M[X] ← R1 \* R2

*Two-Address*

1. MOV A, R1; R1 ← M[A]
2. ADD B, R1 ; R1 ← R1 +M[B]
3. MOV C, R2; R2 ← M[C]
4. ADD D, R2 ; R2 ← R2 +M[D]

5. MUL R2, R1; R1 ← R1 \* R2

6. MOV R1, X; M[X] ← R1

*One-Address*

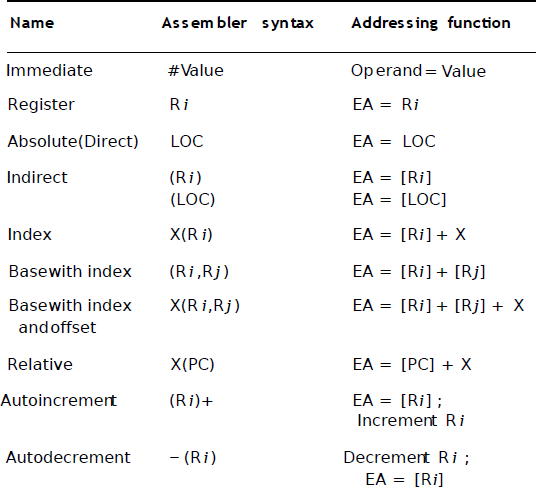
1. LOAD A ; AC← M[A]
2. ADD B ; AC← AC + M[B]
3. STORE T; M[T]← AC
4. LOAD C ; AC← M[C]
5. ADD D ; AC← AC + M[D]
6. MUL T ; AC← AC \*M[T]
7. STORE X ; M[X] ← AC

*Zero-Address*

1. PUSH A ; TOS← A
2. PUSH B ; TOS← B
3. ADD ; TOS← (A + B)
4. PUSH C ; TOS← C
5. PUSH D ; TOS← D
6. ADD ; TOS← (C + D)
7. MUL ; TOS← (C+D)\*(A+B)
8. POPX ; M[X] ← TOS

**Addressing Modes**

The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes. It is a method used to determine which part of memory is being referred by a machine instruction.



### Register mode

Operand is the content of a processor register. The register name/address is given in the instruction. Value of R2 is moved to R1.

Example: MOV R1, R2

### Absolute mode (direct)

Operand is in the memory location. Address of the location is given explicitly.

Here value in A is moved to 1000H.

Example: MOV 1000, A

### Immediate mode:

Address and data constants can be given explicitly in the instruction. Here value constant 200 is moved to R0 register.

Example: MOV #200, R0

### Indirect Mode:

The processor will read the register content (R1) in this case, which will not have direct value. Instead, it will be the address or location in which, the value will be stored. Then the fetched value is added with the value in R0 register.

Example: ADD (R1), R0

### Indexed / Relative Addressing Mode:

The processor will take R1 register address as base address and adds the value constant 20 (offset / displacement) with the base address to get the derived or actual memory location of the value i.e., stored in the memory. It fetches the value then adds the value to R2 register.

Example: ADD 20(R1), R2

### Auto increment mode and Auto decrement Mode:

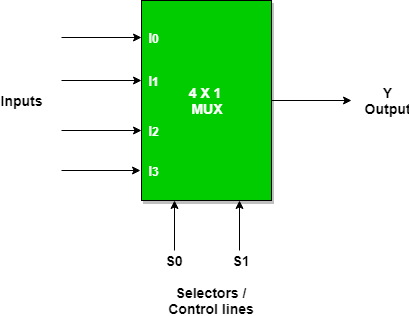
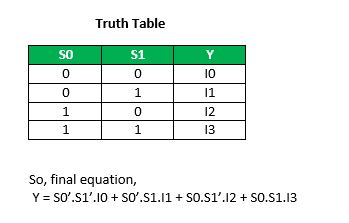
The value in the register / address that is supplied in the instruction is incremented or decremented.

Ex: Increment R1 (Increments the given register / address content by one) Ex: Decrement R2 (Decrements the given register / address content by one)

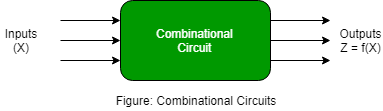
|  |  |  |  |
| --- | --- | --- | --- |
| **Addressing modes** | **Example Instruction** | **Meaning** | **When used** |
| **Register** | Add R4,R3 | R4 <- R4 + R3 | When a value is in a register |
| **Immediate** | Add R4, #3 | R4 <- R4 + 3 | For constants |
| **Displacement** | Add R4, 100(R1) | R4 <- R4 + M[100+R1] | Accessing local variables |
| **Register deffered** | Add R4,(R1) | R4 <- R4 + M[R1] | Accessing using a pointer or a  computed address |
| **Indexed** | Add R3, (R1 + R2) | R3 <- R3 + M[R1+R2] | Useful in array addressing: R1 - base of array  R2 - index amount |
| **Direct** | Add R1, (1001) | R1 <- R1 + M[1001] | Useful in accessing static data |
| **Memory deferred** | Add R1, @(R3) | R1 <- R1 + M[M[R3]] | If R3 is the address of a  pointer *p*, then mode yields *\*p* |
| **Auto- increment** | Add R1, (R2)+ | R1 <- R1 +M[R2] R2 <- R2 + *d* | Useful for stepping through arrays in a loop.  R2 - start of array  *d* - size of an element |
| **Auto- decrement** | Add R1,-(R2) | R2 <-R2-*d*  R1 <- R1 + M[R2] | Same as autoincrement. Both can also be used to implement a stack as push  and pop |
| **Scaled** | Add R1, 100(R2)[R3] | R1<- R1+M[100+R2+R3\**d*] | Used to index arrays. May be applied to any base addressing mode in some  machines. |

# Multiplexers in Digital Logic

It is a combinational circuit which have many data inputs and single output depending on control or select inputs.&#x200b For N input lines, log n (base2) selection lines, or we can say that for 2n input lines, n selection lines are required. Multiplexers are also known as **?ta n selector, parallel to serial convertor, many to one circuit, universal logic circuit&#x200b&#x201d**. Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.

Combinational Circuits:

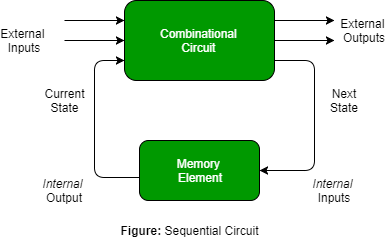


**Combinational Circuit Features:**

In this output depends only upon present input.

1. Speed is fast.
2. It is designed easy.
3. There is no feedback between input and output.
4. This is time independent.
5. Elementary building blocks: Logic gates
6. Used for arithmetic as well as boolean operations.
7. Combinational circuits don’t have capability to store any state.
8. As combinational circuits don’t have clock, they don’t require triggering.
9. These circuits do not have any memory element.
10. It is easy to use and handle.

**Sequential Citcuits:**



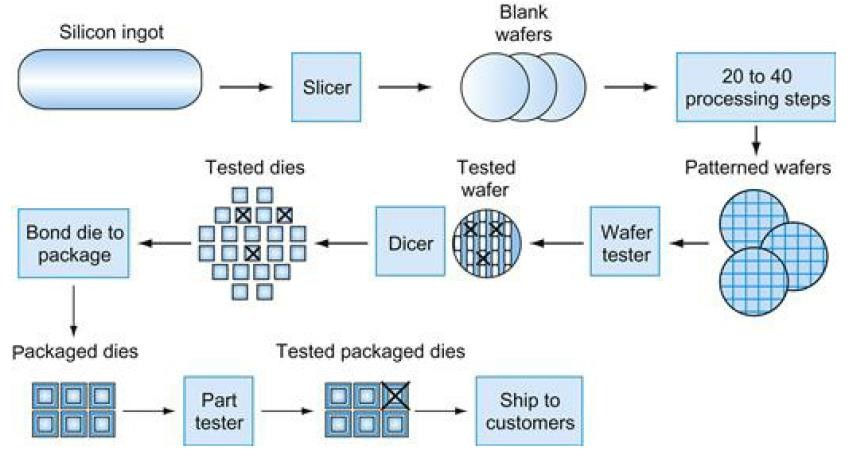
**Sequential Circuit Features–**

1. In this output depends upon present as well as past input.
2. Speed is slow.
3. It is designed tough as compared to combinational circuits.
4. There exists a feedback path between input and output.
5. This is time dependent.
6. Elementary building blocks: Flip-flops
7. Mainly used for storing data.
8. Sequential circuits have capability to store any state or to retain earlier state.
9. As sequential circuits are clock dependent they need triggering.
10. These circuits have memory element.
11. It is not easy to use and handle.

# Technologies for Building Processors and Memory

1. Transistor
2. very large-scale integrated (V L S I ) circuit
3. silicon
4. semiconductor
5. silicon crystal ingot
6. wafer
7. defect
8. die

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (I C) combined dozens to hundreds of transistors into a single chip.

To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

To manufacture integrated circuits, we start at the beginning. The manufacture of a chip begins with **silicon**, silicon does not conduct electricity well, and it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

* Excellent conductors of electricity (using either microscopic copper or aluminum wire)
* Excellent insulators from electricity (like plastic sheathing or glass)
* Areas that can conduct or insulate under special conditions (as a switch)

### Process of making semiconductor devices

* The process starts with a **silicon crystal ingot**, which looks like a giant sausage.
* An ingot is finely sliced into **wafers** no more than 0.1 inches thick.
* These wafers then go through a series of processing steps, during which patterns of chemicals are placed on each wafer, creating the transistors, conductors, and insulators.
* A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer.
* The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components, called **dies** and more informally known as **chips**.
* Dicing enables to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer.
* This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

**power wall**

Both clock rate and power increased rapidly for decades, and then flattened off recently. The **energy metric joules** is a better measure than a **power rate like watts**, which is just joules/second.

Dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is **so-called dynamic energy**—that is, energy that is **consumed when transistors switch states from 0 to 1** and vice versa.

### Energy *α* Capacitive load X (Voltage)*2*

This equation is the energy of a pulse during the logic transition of 0 → 1 → 0 or 1 → 0 →

1. The energy of a single transition is then

### Energy *α* 1/2 X Capacitive load X (Voltage)*2*

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

### Power *α* Energy X Frequency switched [or]

**Power *α* 1/2 X Capacitive load X (Voltage)*2* X Frequency switched**

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the fan out) and the technology, which determines the capacitance of both wires and transistors.

Energy and thus power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. In 20 years, voltages have gone from **5 V to 1 V,** which is why the increase in power is only 30 times. To try to address the power problem, designers have already attached **large devices to increase cooling**, and they **turn off parts of the chip** that are not used in a given clock cycle.

# Uniprocessors to multiprocessors

SVCET

**Uniprocessor**

* + A type of architecture that is based on a single computing unit. All operations (additions, multiplications, etc.) are done sequentially on the unit.

**Multiprocessor**

* + A type of architecture that is based on multiple computing units. Some of the operations (not all, mind you) are done in parallel and the results are joined afterwards.

**Parameter Uniprocessor Systems Multiprocessor Systems**

|  |  |  |
| --- | --- | --- |
| **Description** | If a System contains only one processor for processing than it is  called single processor system. | If a System contains two or more than two processors for processing than it is called  multiprocessor system. |
| **Use of Co- Processors** | Yes, Single Processors uses multiple Controllers that are designed to handle special tasks and can execute limited instruction sets. E.g. DMA Controller, North/South Bridge. | In Multi-Processor Systems Two Types of approaches are Used:   1. Symmetric Multiprocessing(SMP) 2. Asymmetric Multiprocessing   In *Asymmetric Multiprocessing* one Processor works as Master and Second Processor act as Slave *In Symmetric Multiprocessing* each processor  performs all the tasks within the operating system. |
| **Throughput** | Throughput of Single Processor Systems is less than Multiprocessor Systems Because each and every task is performed by the same processor. | Throughput of Multiprocessor systems is greater than single processor systems. If a System Contains N Processors than its throughput will be slightly less than N because synchronization must be maintained between two processors and they also share resource which increases certain amount  of overhead. |
| **Cost Economic** | Single Processor Systems cost more because each processor requires separate resources .i.e. Mass Storage, Peripherals, Power  Supplies etc. | Multiprocessor Systems cost less than equivalent multiple single processor systems because they uses same resources on sharing basis. |
| **Design Process** | It is Easy to design Single Processor Systems. | It is difficult to design Multi Processor Systems because Synchronization must be maintained between processors otherwise it may result in overloading of one processor and another  processor may remain idle on the same time. |
| **Reliability** | Less reliable because failure in one processor will result in failure of  entire system. | More reliable because failure of one processor does not halt the entire system but only speed will be  slow down. |
| **Examples** | Most of Modern PCs. | Blade Servers. |

# Performance of a System:

**Computer performance** is characterized by the amount of useful work accomplished by a computer system or computer network compared to the time and resources used

* + As an individual computer user, you are interested in reducing **response time** — time between the start and completion of a task — also referred to as **execution time**.
  + Datacentre managers are often interested in increasing **throughput** or **bandwidth**—the total amount of work done in a given time.

**Response time**

Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I /O activities, operating system overhead, CPU execution time, and so on.

**Throughput**

Also called **bandwidth**. Another measure of performance, it is the number of tasks completed per unit time.

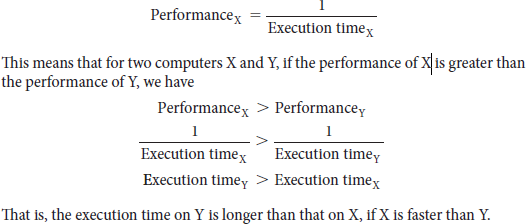
**Bandwidth**

The amount of data that can be carried from one point to another in a given time period (usually a second). This kind of bandwidth is usually expressed in bits (of data) per second (bps). Occasionally, it's expressed as bytes per second (Bps).

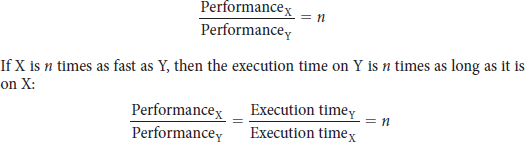
**Clock cycles per instruction (CPI):**

Average number of clock cycles per instruction for a program or program fragment.

*We can relate performance and execution time for a computer X:*



In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is *n* times faster than Y”— or equivalently “X is *n* times as fast as Y”—to mean



The most important measure of a computer is how quickly it can execute programs.

**Measuring Performance:**

* + The computer that performs the same amount of work in the **least time** is the fastest. Program execution time is **measured in seconds** per program.
  + **CPU execution time** or **simply CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and **does not include** time spent **waiting for I/O or running** other programs.
  + CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time.**
  + The term **system performance** to refer to **elapsed time on an unloaded system** and CPU performance to refer to **user CPU time**.

**CPU Performance and Its Factors:**

***CPU execution time for a program = CPU clock cycles for a program X Clock cycle***

***time***

Alternatively, because clock rate and clock cycle time are inverses,

***CPU execution time for a program = CPU clock cycles for a program***

***Clock rate***

* + This formula makes it clear that the hardware designer **can improve performance** by reducing the number of clock cycles required for a program or the length of the clock cycle.

**Instruction Performance:**

* + The performance equations above did not include any reference to the number of instructions needed for the program. The execution time must depend on the number of instructions in a program.
  + Here **execution time** is that it equals the number of instructions executed multiplied by the average time per instruction. **clock cycles** required for a program can be written as

**CPU clock cycles = Instructions for a program X Average clock cycles per instruction**

* + The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is **often abbreviated as CPI.**
  + CPI provides one way of **comparing two different implementations** of the same instruction set architecture, since the number of instructions executed for a program will be the same.

**The Classic CPU Performance Equation:**

The basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

***CPU time = Instruction count X CPI X Clock cycle time***

or, since the clock rate is the inverse of clock cycle time:

***CPU time = (Instruction count/Clock rate) X CPI***

These formulas are particularly useful because they separate the three key factors that affect performance.

|  |  |
| --- | --- |
| **Components of performance** | **Units of Measure** |
| CPU execution time for a program | Seconds for the program |
| Instruction count | Instructions executed for the program |
| Clock cycles per instruction (CPI) | Average number of clock cycles per |
| Clock cycle time | Seconds per clock cycle |

* + We can measure the **CPU execution time by running** the program, and the **clock cycle time is** usually published as part of the documentation for a computer.
  + **The instruction count and CPI** can be more **difficult to obtain**. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.