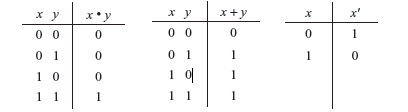
**UNIT 2 -DLCF**

**Two-Valued Boolean Algebra**

A two-valued Boolean algebra is defined on a set of two elements, *B* = {0, 1}, with rules for the two binary operators + and • as shown in the following operator tables (the rule for the complement operator is for verification of postulate 5):



These rules are exactly the same as the AND, OR, and NOT operations, respectively, defined in Table 1-6. We must now show that the Huntington postulates are valid for the set *B* = {0, 1} and the two binary operators defined above.

1. *Closure* is obvious from the tables since the result of each operation is either 1 or 0 and 1,

0 ∈ *B.*

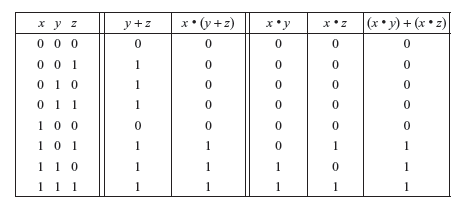
2. From the tables we see that:



which establishes the two *identity elements* 0 for + and 1 for • as defined by postulate 2.

3. The *commutative* laws are obvious from the symmetry of the binary operator tables.

4. (a) The *distribute* law *x* • (*y* + *z*) = (*x* • *y*) + (*x* • *z*) can be shown to hold true from the operator tables by forming a truth table of all possible values of *x, y,* and *z.* For each combination, we derive *x •* (*y* + *z*) and show that the value is the same as (*x* • *y*)+ (*x* • *z*)*.*



(b) The *distributive* law of + over • can be shown to hold true by means of a truth table similar to the one above.

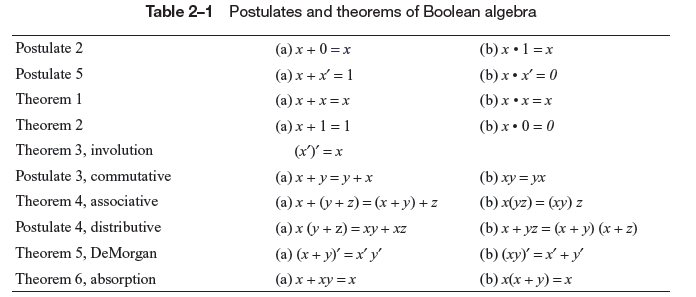
5. From the complement table it is easily shown that:

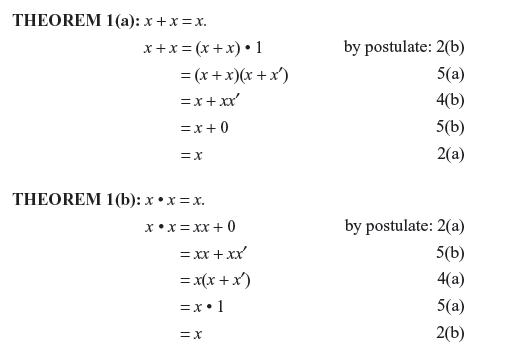
(a) *x* + *x*′ = 1, since 0 + 0′ = 0 + 1 = 1 and 1 + 1′ = 1 + 0 = 1

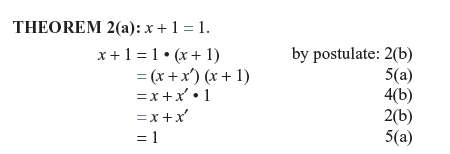
(b) *x* • *x*′ = 0, since 0 • 0′ = 0 • 1 = 0 and 1 • 1′ = 1 • 0 = 0 which verifies postulate 5.

6. Postulate 6 is satisfied because the two-valued Boolean algebra has two distinct elements 1 and 0 with 1 ¹ 0.

*Principle of duality:* It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, interchange OR and AND operators and replace 1’s by 0’s and 0’s by 1’s.

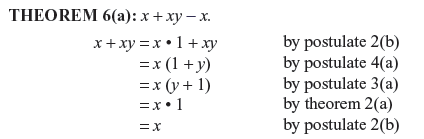






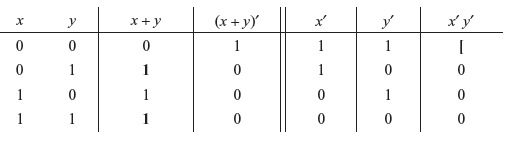
**THEOREM 2 (b):** *x* • 0 = 0 by duality.

**THEOREM 3**: (*x*′)′ = *x*. From postulate 5, we have *x* + *x*′ = 1 and *x* • *x*′ = 0, which defines the complement of *x*. The complement of *x*′ is *x* and is also (*x*′)′*.* Therefore, since the complement is unique, we have that (*x*′)′ = *x*.



**THEOREM 6(b):** *x*(*x* + *y*) = *x* by duality.

**De Morgan’s Theorem 5a and 5b: proof by TT**

****

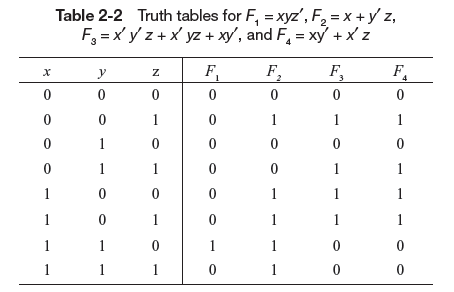
**Boolean Functions**

A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, parentheses, and equal sign. For a given value of the variables, the function can be either 0 or 1.

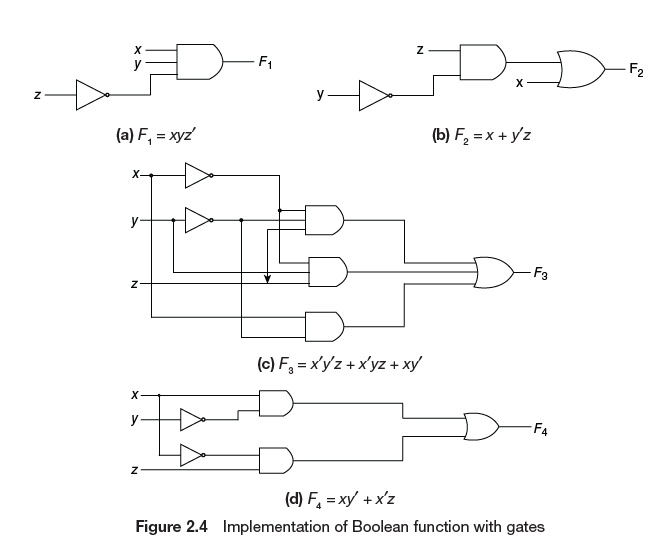
*Eg*: F1 = *xyz*′

The function *F*1 is equal to 1 if *x* = 1 *and y* = 1 *and z*′ = 1; otherwise *F*1 = 0

Any Boolean function can be represented in a truth table. The number of rows in the table is 2*n*, where *n* is the number of binary variables in the function. The 1’s and 0’s combinations for each row is easily obtained from the binary numbers by counting from 0 to *2n* - 1.



A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR, and NOT gates.



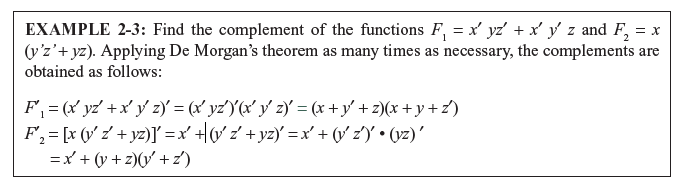
Some terms:

A *literal* is a primed or unprimed variable

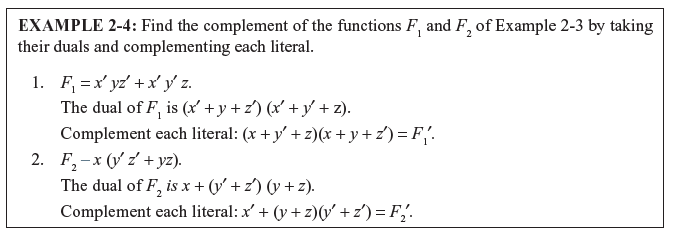
Note: The minimization of the number of literals and the number of terms results in a circuit with less equipment

**Complement of a Function**

**Method 1 (using De Morgan’s Theorem):** The complement of a function *F* is *F*′ and is obtained from an interchange of 0’s for 1’s and l’s for 0’s in the value of *F.* The complement of a function may be derived algebraically through De Morgan’s theorem.



**Method 1 :** Take the dual of the function and complement each literal.



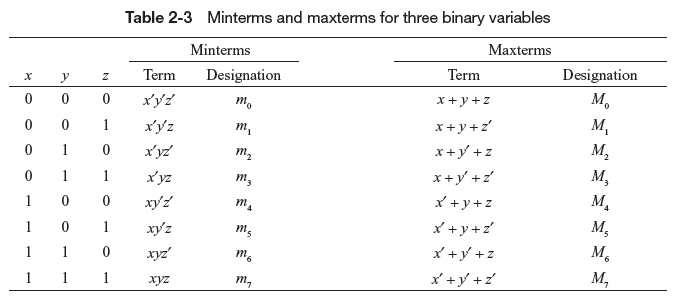
**Canonical Form:**

Boolean functions expressed as a sum of minterms or product of maxterms are said to be in ***canonical form.***

*Minterms and Maxterms*

A binary variable may appear either in its normal form (*x*) or in its complement form (*x*′)*,* and there are four possible combinations: *x*′*y*′*, x*′*y, xy*′*,* and *xy.* Each of these four AND is called a ***minterm***or a ***standard******product****.* n variables can he combined to form 2n minterms*.*

In a similar fashion, *n* variables forming an OR term, with each variable being primed or unprimed, provide 2npossible combinations, called ***maxterms***or ***standard******sums****.* Each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1.

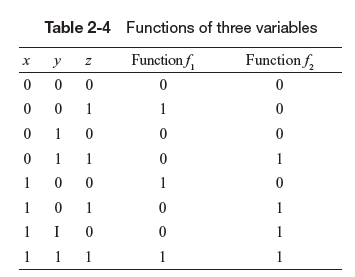


Note: each maxterm is the complement of its corresponding minterm, and vice versa

**Expressing a Function from a given truth table in sum of minterms:**

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of the variables which produces a 1 in the function, and then taking the OR of all those terms.

For example

**

*f*1 = *x*′*y*′*z* + *xy*′*z*′ + *xyz = m*1 + *m4* + *m7*

Similarly, it may be easily verified that:

*f*2= *x*′*yz* + *xy*′*z* + *xyz*′ + *xyz* = m3 + *m5* + *m*6 + *m7*

**Expressing a Function from a given truth table in product of maxterms:**

Take the complement of *f*1, by reading from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms. The complement of *f*1 is

*f1′* = *x*′*y*′*z*′ + *x*′*yz*′ + *x*′*yz* + *xy*′*z* + *xyz’*

To obtain the original function *f1,* take the complement ,

*f1′’= f1* = (*x* + *y* + *z*)(*x* + *y*′+ *z*)(*x* + *y*′ + *z*′)(*x*′ + *y* + *z*′)(*x*′ + *y*′ + *z*)= *M0*⋅*M2*⋅*M3*⋅*M5*⋅*M6*

Note: Any Boolean function can be expressed as a product of maxterms (by “product” is meant the ANDing of terms).

The procedure for obtaining the product of maxterms directly from the truth table is as follows.

* Form a maxterm for each combination of the variables which produces a 0 in the function, and then form the AND of all those maxterms.

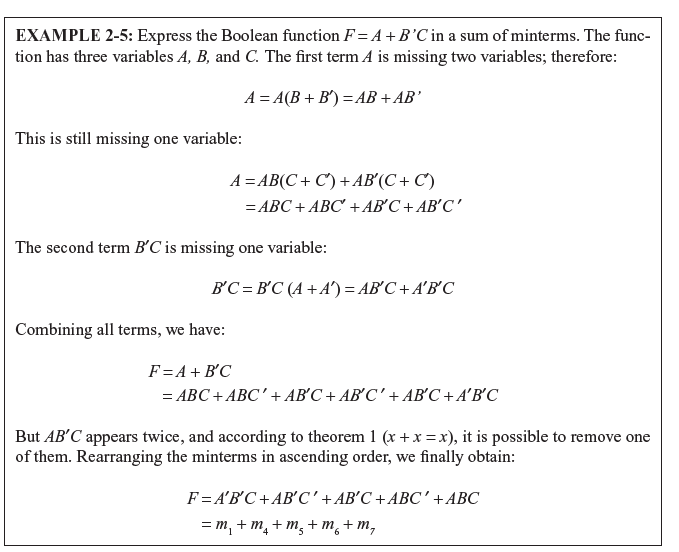
Thus by simply reading from the TT above, the expression for *f*2 in product of Maxterms is :

*f*2 = (*x* + *y* + *z*)(*x* + *y* +*z*′)(*x* + *y*′ + *z*)(*x*′ + *y* + *z*) = *M*0.*M*1.*M*2.*M*4

**A given function can be expressed in sum of minterms form in two ways:**

1. Algebraic manipulation
2. Drawing the function’s TT
3. Algebraic manipulation:

First expand the expression into a sum of AND terms. Each term is then checked to see if it contains all the variables. **If it misses one or more variables, it is ANDed with an expression** such as *x* + *x*′, where *x* is one of the missing variables



The sum of minterms form may also be expressed in the following short notation:

*F*(*A*,*B*,*C*) = Σ (1,4, 5,6,7)

The summation symbol Σ stands for the ORing of terms; the numbers following it are the minterms of the function.

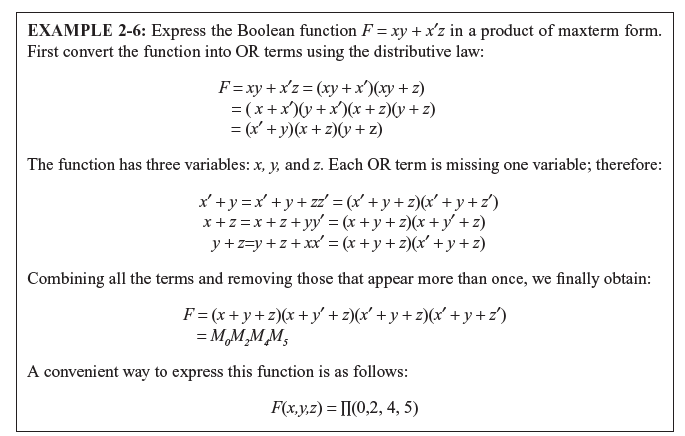
1. Drawing the function’s TT:

Draw the TT of the fuction and sum all the minterms where the function results in a 1.(as discussed earlier)

A given function can be expressed in product of maxterms form in two ways:

1. Algebraic manipulation
2. Drawing the function’s TT
3. Algebraic manipulation:

To express the Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law *x* + *yz* = (*x* + *y*)(*x* + z). Then any missing variable *x* in each OR term is ORed with *xx*′*.*

**

**Conversion between Canonical Forms**

To convert from one canonical form to another, interchange the symbols Σ and *Π* and list those numbers missing from the original form. As another example, the function:

*F*(*x*, *y*, *z*) = *Π* (0, 2, 4, 5)

is expressed in the product of maxterm form. Its conversion to sum of minterms is:

*F*(*x*, *y*, *z*) = Σ (1,3,6,7)

**Standard Forms**

In this form, the terms that form the function may contain one, two or any number of literals. There are two types of standard forms: **the sum of products and product of sums**.

The ***sum of products***is a Boolean expression containing AND terms, called *product terms,*of one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed in sum of products is:

*F*1 = *y*′ + *xy* + *x*′*yz*′

A ***product of sums***is a Boolean expression containing OR terms, called *sum terms.* Each term may have any number of literals. The *product* denotes the AN Ding of these terms. An example of a function expressed in product of sums is:

*F2* = *x*(*y*′ + *z*)(*x*′ + *y* + *z*′ + *w*)

A Boolean function may be expressed in a **nonstandard form**. For example, the function:

*F*3 = (*AB* + *CD*)(*A*′*B*′ + *C*′*D*′)

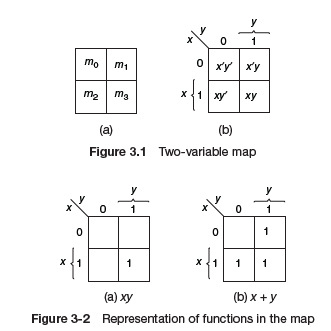
is neither in sum of products nor in product of sums. It can be changed to a standard form by using the distributive law to remove the parentheses:

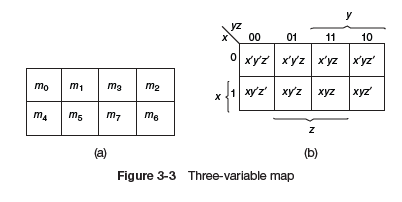
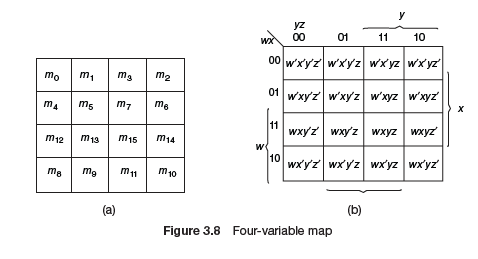
*F3* = *A*′*B*′*CD* + *ABC*′*D*′

**Simplification of Boolean Functions using Karnaugh Map Method**

The map method provides a simple straightforward procedure for minimizing Boolean functions. The map is a diagram made up of squares. Each square represents one minterm. A Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.

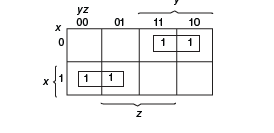
Labelling the KarnaughMap :



**EXAMPLE 3-1:** Simplify the Boolean function using K-map:

*F* = *x*′*yz* + *x*′*yz*′ *+ xy*′*z*′ + *xy*′*z*

 *F* = *x*′*y* + *xy*′

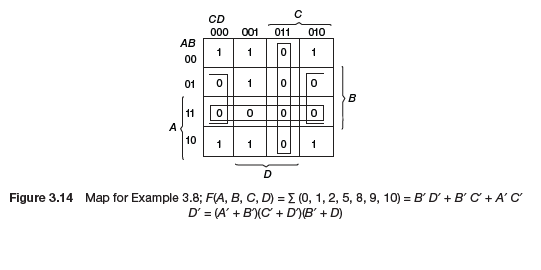
The above gives an SOP simplification from the K-map

For POS Simplification using the Kmap:

* Mark the empty squares by 0’s and combine them into valid adjacent squares . we obtain a simplified expression of the complement of the function, i.e., of *F*′*.*
* The complement of *F*′ gives us back the function in the product of sums form.

Example: Simplify the following function using K-map in POS form

*F*(*A*, *B*, *C*, *D*) = Σ (0, 1, 2, 5, 8, 9, 10)



F’(A,B,C,D)= AB+BD’+CD

Recomplement : F’’=F = [AB+BD’+CD]’

= (A’+B’)(B’+D)(C’+D’) (simplified in POS form)