Boolean Algebra Chapter Two

Logic circuits are the basis for modern digital computer systems. To appreciate how computer systems operate you will need to understand digital logic and boolean algebra.

This Chapter provides only a basic introduction to boolean algebra. This subject alone is often the subject of an entire textbook. This Chapter will concentrate on those subject that support other chapters in this text.

2.0 Chapter Overview

Boolean logic forms the basis for computation in modern binary computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of boolean equations. This chapter provides a brief introduction to boolean algebra, truth tables, canonical representation, of boolean functions, boolean function simplification, logic design, combinatorial and sequential circuits, and hardware/software equivalence.

The material is especially important to those who want to design electronic circuits or write software that controls electronic circuits. Even if you never plan to design hardware or write software than controls hardware, the introduction to boolean algebra this chapter provides is still important since you can use such knowledge to optimize certain complex conditional expressions within IF, WHILE, and other conditional statements.

The section on minimizing (optimizing) logic functions uses *Veitch Diagrams* or *Karnaugh Maps*. The optimizing techniques this chapter uses reduce the number of *terms* in a boolean function. You should realize that many people consider this optimization technique obsolete because reducing the number of terms in an equation is not as important as it once was. This chapter uses the mapping method as an example of boolean function optimization, not as a technique one would regularly employ. If you are interested in circuit design and optimization, you will need to consult a text on logic design for better techniques.

Although this chapter is mainly hardware-oriented, keep in mind that many concepts in this text will use boolean equations (logic functions). Likewise, some programming exercises later in this text will assume this knowledge. Therefore, you should be able to deal with boolean functions before proceeding in this text.

2.1 Boolean Algebra

Boolean algebra is a deductive mathematical system closed over the values zero and one (false and true). A *binary operator* "°" defined over this set of values accepts a pair of boolean inputs and produces a single boolean value. For example, the boolean AND operator accepts two boolean inputs and produces a single boolean output (the logical AND of the two inputs).

For any given algebra system, there are some initial assumptions, or *postulates*, that the system follows. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the following postulates:

- *Closure*. The boolean system is *closed* with respect to a binary operator if for every pair of boolean values, it produces a boolean result. For example, logical AND is closed in the boolean system because it accepts only boolean operands and produces only boolean results.
- *Commutativity*. A binary operator "°" is said to be commutative if A°B = B°A for all possible boolean values A and B.

- *Associativity*. A binary operator "°" is said to be associative if

  $$(A \circ B) \circ C = A \circ (B \circ C)$$

  for all boolean values A, B, and C.

- *Distribution*. Two binary operators "°" and "%" are distributive if

  $$A \circ (B \,\%\, C) = (A \circ B) \,\%\, (A \circ C)$$

  for all boolean values A, B, and C.

- *Identity*. A boolean value I is said to be the *identity element* with respect to some binary operator "°" if $A \circ I = A$.

- *Inverse*. A boolean value I is said to be the *inverse element* with respect to some binary operator "°" if $A \circ I = B$ and $B \neq A$ (i.e., B is the opposite value of A in a boolean system).

For our purposes, we will base boolean algebra on the following set of operators and values:

The two possible values in the boolean system are zero and one. Often we will call these values false and true (respectively).

The symbol "•" represents the logical AND operation; e.g., A • B is the result of logically ANDing the boolean values A and B. When using single letter variable names, this text will drop the "•" symbol; Therefore, AB also represents the logical AND of the variables A and B (we will also call this the *product* of A and B).

The symbol "+" represents the logical OR operation; e.g., A + B is the result of logically ORing the boolean values A and B. (We will also call this the *sum* of A and B.)

Logical *complement*, *negation*, or *not*, is a unary operator. This text will use the (') symbol to denote logical negation. For example, A' denotes the logical NOT of A.

If several different operators appear in a single boolean expression, the result of the expression depends on the *precedence* of the operators. We'll use the following precedences (from highest to lowest) for the boolean operators: parenthesis, logical NOT, logical AND, then logical OR. The logical AND and OR operators are *left associative*. If two operators with the same precedence are adjacent, you must evaluate them from left to right. The logical NOT operation is right associative, although it would produce the same result using left or right associativity since it is a unary operator.

We will also use the following set of postulates:

P1 Boolean algebra is closed under the AND, OR, and NOT operations.

P2 The identity element with respect to • is one and + is zero. There is no identity element with respect to logical NOT.

P3 The • and + operators are commutative.

P4 • and + are distributive with respect to one another. That is, A • (B + C) = (A • B) + (A • C) and A + (B • C) = (A + B) • (A + C).

P5 For every value A there exists a value A' such that A•A' = 0 and A+A' = 1. This value is the logical complement (or NOT) of A.

P6 • and + are both associative. That is, (A•B)•C = A•(B•C) and (A+B)+C = A+(B+C).

You can prove all other theorems in boolean algebra using these postulates. This text will not go into the formal proofs of these theorems, however, it is a good idea to familiarize yourself with some important theorems in boolean algebra. A sampling include:

Th1: A + A = A

Th2: A • A = A

Th3: A + 0 = A

Th4: A • 1 = A

Th5: A • 0 = 0

Th6: A + 1 = 1

Th7: (A + B)' = A' • B'

Th8: (A • B)' = A' + B'

Th9: A + A•B = A

Th10: A •(A + B) = A

Th11: A + A'B = A+B

Th12: A' • (A + B') = A'B'

Th13: AB + AB' = A

Th14: (A'+B') • (A' + B) = A'

Th15: A + A' = 1

Th16: A • A' = 0

Theorems seven and eight above are known as *DeMorgan's Theorems* after the mathematician who discovered them.

The theorems above appear in pairs. Each pair (e.g., Th1 & Th2, Th3 & Th4, etc.) form a *dual*. An important principle in the boolean algebra system is that of *duality*. Any valid expression you can create using the postulates and theorems of boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the • and + operators and swap the 0 and 1 values in an expression, you will wind up with an expression that obeys all the rules of boolean algebra. *This does not mean the dual expression computes the same values*, it only means that both expressions are legal in the boolean algebra system. Therefore, this is an easy way to generate a second theorem for any fact you prove in the boolean algebra system.

Although we will not be proving any theorems for the sake of boolean algebra in this text, we will use these theorems to show that two boolean equations are identical. This is an important operation when attempting to produce *canonical representations* of a boolean expression or when simplifying a boolean expression.

2.2 Boolean Functions and Truth Tables

A boolean *expression* is a sequence of zeros, ones, and *literals* separated by boolean operators. A literal is a primed (negated) or unprimed variable name. For our purposes, all variable names will be a single alphabetic character. A boolean function is a specific boolean expression; we will generally give boolean functions the name "F" with a possible subscript. For example, consider the following boolean:

$$F_0 = AB+C$$

This function computes the logical AND of A and B and then logically ORs this result with C. If A=1, B=0, and C=1, then F_0 returns the value one (1•0 + 1 = 1).

Another way to represent a boolean function is via a *truth table*. The previous chapter used truth tables to represent the AND and OR functions. Those truth tables took the forms:

Table 6: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 7: OR Truth Table

OR	0	1
0	0	1
1	1	1

For binary operators and two input variables, this form of a truth table is very natural and convenient. However, reconsider the boolean function F_0 above. That function has *three* input variables, not two. Therefore, one cannot use the truth table format given above. Fortunately, it is still very easy to construct truth tables for three or more variables. The following example shows one way to do this for functions of three or four variables:

Table 8: Truth Table for a Function with Three Variables

F = AB + C		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

Table 9: Truth Table for a Function with Four Variables

F = AB + CD		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

In the truth tables above, the four columns represent the four possible combinations of zeros and ones for A & B (B is the H.O. or leftmost bit, A is the L.O. or rightmost bit). Likewise the four rows in the second truth table above represent the four possible combinations of zeros and ones for the C and D variables. As before, D is the H.O. bit and C is the L.O. bit.

Table 10 shows another way to represent truth tables. This form has two advantages over the forms above – it is easier to fill in the table and it provides a compact representation for two or more functions.

Note that the truth table above provides the values for three separate functions of three variables.

Although you can create an infinite variety of boolean functions, they are not all unique. For example, F=A and F=AA are two different functions. By theorem two, however, it is easy to show that these two functions are equivalent, that is, they produce exactly the same outputs for all input combinations. If you fix the number of input variables, there are a finite number of unique boolean functions possible. For example, there are only 16 unique boolean functions with two inputs and there are only 256 possible boolean functions of three input variables. Given *n* input variables, there are $2^{**}(2^n)$ (two raised to the two raised to the n^{th} power) unique boolean functions of those *n* input values. For two input variables, $2^{\wedge}(2^2) = 2^4$ or 16 different functions. With three input vari-

Table 10: Another Format for Truth Tables

C	B	A	F = ABC	F = AB + C	F = A+BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

ables there are $2^{**}(2^3) = 2^8$ or 256 possible functions. Four input variables create $2^{**}(2^4)$ or 2^{16}, or 65,536 different unique boolean functions.

When dealing with only 16 boolean functions, it's easy enough to name each function. The following table lists the 16 possible boolean functions of two input variables along with some common names for those functions:

Table 11: The 16 Possible Boolean Functions of Two Variables

Function #	Description
0	Zero or Clear. Always returns zero regardless of A and B input values.
1	Logical NOR (NOT (A OR B)) = (A+B)'
2	Inhibition = BA' (B, not A). Also equivalent to B>A or A < B.
3	NOT A. Ignores B and returns A'.
4	Inhibition = AB' (A, not B). Also equivalent to A>B or B<A.
5	NOT B. Returns B' and ignores A
6	Exclusive-or (XOR) = A $\oplus$ B. Also equivalent to A≠B.
7	Logical NAND (NOT (A AND B)) = (A•B)'
8	Logical AND = A•B. Returns A AND B.
9	Equivalence = (A = B). Also known as exclusive-NOR (not exclusive-or).
10	Copy B. Returns the value of B and ignores A's value.
11	Implication, B implies A = A + B'. (if B then A). Also equivalent to B >= A.
12	Copy A. Returns the value of A and ignores B's value.
13	Implication, A implies B = B + A' (if A then B). Also equivalent to A >= B.
14	Logical OR = A+B. Returns A OR B.
15	One or Set. Always returns one regardless of A and B input values.

Beyond two input variables there are too many functions to provide specific names. Therefore, we will refer to the function's number rather than the function's name. For example, F_8 denotes the logical AND of A and B for a two-input function and F_{14} is the logical OR operation. Of course, the only problem is to determine a function's number. For

example, given the function of three variables F=AB+C, what is the corresponding function number? This number is easy to compute by looking at the truth table for the function (see Table 14 on page 50). If we treat the values for A, B, and C as bits in a binary number with C being the H.O. bit and A being the L.O. bit, they produce the binary numbers in the range zero through seven. Associated with each of these binary strings is a zero or one function result. If we construct a binary value by placing the function result in the bit position specified by A, B, and C, the resulting binary number is that function's number. Consider the truth table for F=AB+C:

CBA:	7	6	5	4	3	2	1	0
F=AB+C:	1	1	1	1	1	0	0	0

If we treat the function values for F as a binary number, this produces the value $F8_{16}$ or 248_{10}. We will usually denote function numbers in decimal.

This also provides the insight into why there are $2^{**}2^n$ different functions of *n* variables: if you have *n* input variables, there are 2^n bits in function's number. If you have *m* bits, there are 2^m different values. Therefore, for *n* input variables there are $m=2^n$ possible bits and 2^m or $2^{**}2^n$ possible functions.

2.3 Algebraic Manipulation of Boolean Expressions

You can transform one boolean expression into an equivalent expression by applying the postulates the theorems of boolean algebra. This is important if you want to convert a given expression to a *canonical form* (a standardized form) or if you want to minimize the number of literals (primed or unprimed variables) or terms in an expression. Minimizing terms and expressions can be important because electrical circuits often consist of individual components that implement each term or literal for a given expression. Minimizing the expression allows the designer to use fewer electrical components and, therefore, can reduce the cost of the system.

Unfortunately, there are no fixed rules you can apply to optimize a given expression. Much like constructing mathematical proofs, an individual's ability to easily do these transformations is usually a function of experience. Nevertheless, a few examples can show the possibilities:

```
ab + ab' + a'b          =  a(b+b') + a'b              By P4
                        =  a•1 + a'b                  By P5
                        =  a + a'b                    By Th4
                        =  a + a'b + 0                By Th3
                        =  a + a'b + aa'              By P5
                        =  a + b(a + a')              By P4
                        =  a + b•1                    By P5
                        =  a + b                      By Th4


(a'b + a'b' + b')'      =  ( a'(b+b') + b')'          By P4
                        =  (a' + b')'                 By P5
                        =  ( (ab)' )'                 By Th8
                        =  ab                         By definition of not


b(a+c) + ab' + bc' + c  =  ba + bc + ab' + bc' + c    By P4
                        =  a(b+b') + b(c + c') + c    By P4
                        =  a•1 + b•1 + c              By P5
                        =  a + b + c                  By Th4
```

Although these examples all use algebraic transformations to simplify a boolean expression, we can also use algebraic operations for other purposes. For example, the next section describes a canonical form for boolean expressions. Canonical forms are rarely optimal.

2.4 Canonical Forms

Since there are a finite number of boolean functions of *n* input variables, yet an infinite number of possible logic expressions you can construct with those *n* input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms*. Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables, A and B, there are eight possible terms: A, B, A', B', A'B', A'B, AB', and AB. For three variables we have 26 different terms: A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly *n* literals. For example, the minterms for two variables are A'B', AB', A'B, and AB. Likewise, the minterms for three variables A, B, and C are A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. In general, there are 2^n minterms for *n* variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

Table 12: Minterms for Three Input Variables

Binary Equivalent (CBA)	Minterm
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

We can specify *any* boolean function using a sum (logical OR) of minterms. Given F_{248}=AB+C the equivalent canonical form is ABC+A'BC+AB'C+A'B'C+ABC'. Algebraically, we can show that these two are equivalent as follows:

```
ABC+A'BC+AB'C+A'B'C+ABC' =  BC(A+A') + B'C(A+A') + ABC'
                         =  BC•1 +B'C•1 + ABC'
                         =  C(B+B') + ABC'
                         =  C + ABC'
                         =  C + AB
```

Obviously, the canonical form is not the optimal form. On the other hand, there is a big advantage to the sum of minterms canonical form: it is very easy to generate the truth table for a function from this canonical form. Furthermore, it is also very easy to generate the logic equation from the truth table.

To build the truth table from the canonical form, simply convert each minterm into a binary value by substituting a "1" for unprimed variables and a "0" for primed variables.

Then place a “1” in the corresponding position (specified by the binary minterm value) in the truth table:

1) Convert minterms to binary equivalents:

F_{248}= CBA + CBA’ + CB’A + CB’A’ + C’BA

= 111 + 110 + 101 + 100 + 011

2) Substitute a one in the truth table for each entry above

Table 13: Creating a Truth Table from Minterms, Step One

C	B	A	F = AB+C
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Finally, put zeros in all the entries that you did not fill with ones in the first step above:

Table 14: Creating a Truth Table from Minterms, Step Two

C	B	A	F = AB+C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Going in the other direction, generating a logic function from a truth table, is almost as easy. First, locate all the entries in the truth table with a one. In the table above, these are the last five entries. The number of table entries containing ones determines the number of minterms in the canonical equation. To generate the individual minterms, substitute A, B, or C for ones and A’, B’, or C’ for zeros in the truth table above. Then compute the sum of these items. In the example above, F_{248} contains one for CBA = 111, 110, 101, 100, and 011. Therefore, F_{248} = CBA + CBA’ + CB’A + CB’A’ + C’AB. The first term, CBA, comes from the last entry in the table above. C, B, and A all contain ones so we generate the minterm CBA (or ABC, if you prefer). The second to last entry contains 110 for CBA, so we generate the minterm CBA’. Likewise, 101 produces CB’A; 100 produces CB’A’, and 011 produces C’BA. Of course, the logical OR and logical AND operations are both commutative, so we can rearrange the terms within the minterms as we please and we can rearrange the minterms within the sum as we see fit. This process works equally well for any number of

variables. Consider the function F_{53504} = ABCD + A'BCD + A'B'CD + A'B'C'D. Placing ones in the appropriate positions in the truth table generates the following:

Table 15: Creating a Truth Table with Four Variables from Minterms

D	C	B	A	F = ABCD + A'BCD + A'B'CD + A'B'C'D
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	1

The remaining elements in this truth table all contain zero.

Perhaps the easiest way to generate the canonical form of a boolean function is to first generate the truth table for that function and then build the canonical form from the truth table. We'll use this technique, for example, when converting between the two canonical forms this chapter presents. However, it is also a simple matter to generate the sum of minterms form algebraically. By using the distributive law and theorem 15 (A + A' = 1) makes this task easy. Consider F_{248} = AB + C. This function contains two terms, AB and C, but they are not minterms. Minterms contain each of the possible variables in a primed or unprimed form. We can convert the first term to a sum of minterms as follows:

```
AB    =    AB • 1                          By Th4
       =    AB • (C + C')                   By Th 15
       =    ABC + ABC'                      By distributive law
       =    CBA + C'BA                      By associative law
```

Similarly, we can convert the second term in F_{248} to a sum of minterms as follows:

```
C     =    C • 1                           By Th4
       =    C • (A + A')                    By Th15
       =    CA + CA'                        By distributive law
       =    CA•1 + CA'•1                    By Th4
       =    CA • (B + B') + CA' • (B + B')  By Th15
       =    CAB + CAB' + CA'B + CA'B'       By distributive law
       =    CBA + CBA' + CB'A + CB'A'       By associative law
```

The last step (rearranging the terms) in these two conversions is optional. To obtain the final canonical form for F_{248} we need only sum the results from these two conversions:

```
F248  =    (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A')
       =    CBA + CBA' + CB'A + CB'A' + C'BA
```

Another way to generate a canonical form is to use *products of maxterms*. A maxterm is the sum (logical OR) of all input variables, primed or unprimed. For example, consider the following logic function G of three variables:

```
G = (A+B+C) • (A'+B+C) • (A+B'+C).
```

Like the sum of minterms form, there is exactly one product of maxterms for each possible logic function. Of course, for every product of maxterms there is an equivalent sum of minterms form. In fact, the function G, above, is equivalent to

F_{248} = CBA + CBA′ + CB′A + CB′A′ + C′BA = AB +C.

Generating a truth table from the product of maxterms is no more difficult than building it from the sum of minterms. You use the duality principle to accomplish this. Remember, the duality principle says to swap AND for OR and zeros for ones (and vice versa). Therefore, to build the truth table, you would first swap primed and non-primed literals. In G above, this would yield:

G= (A′ + B′ + C′) • (A + B′ + C′) • (A′ + B + C′)

The next step is to swap the logical OR and logical AND operators. This produces

G = A′B′C′ + AB′C′ + A′BC′

Finally, you need to swap all zeros and ones. This means that you store *zeros* into the truth table for each of the above entries and then fill in the rest of the truth table with ones. This will place a zero in entries zero, one, and two in the truth table. Filling the remaining entries with ones produces F_{248}.

You can easily convert between these two canonical forms by generating the truth table for one form and working backwards from the truth table to produce the other form. For example, consider the function of two variables, F_7 = A + B. The sum of minterms form is F_7 = A'B + AB' + AB. The truth table takes the form:

Table 16: F_7 (OR) Truth Table for Two Variables

F_7	A	B
0	0	0
0	1	0
1	0	1
1	1	1

Working backwards to get the product of maxterms, we locate all entries that have a zero result. This is the entry with A and B equal to zero. This gives us the first step of G=A'B'. However, we still need to invert all the variables to obtain G=AB. By the duality principle we need to swap the logical OR and logical AND operators obtaining G=A+B. This is the canonical *product of maxterms* form.

Since working with the product of maxterms is a little messier than working with sums of minterms, this text will generally use the sum of minterms form. Furthermore, the sum of minterms form is more common in boolean logic work. However, you will encounter both forms when studying logic design.

2.5 Simplification of Boolean Functions

Since there are an infinite variety of boolean functions of *n* variables, but only a finite number of unique boolean functions of those *n* variables, you might wonder if there is some method that will simplify a given boolean function to produce the optimal form. Of course, you can always use algebraic transformations to produce the optimal form, but using heuristics does not guarantee an optimal transformation. There are, however, two methods that *will* reduce a given boolean function to its optimal form: the map method and the prime implicants method. In this text we will only cover the mapping method, see any text on logic design for other methods.

Since for any logic function some optimal form must exist, you may wonder why we don't use the optimal form for the canonical form. There are two reasons. First, there may be several optimal forms. They are not guaranteed to be unique. Second, it is easy to convert between the canonical and truth table forms.

Using the map method to optimize boolean functions is practical only for functions of two, three, or four variables. With care, you can use it for functions of five or six variables, but the map method is cumbersome to use at that point. For more than six variables, attempting map simplifications by hand would not be wise[1].

The first step in using the map method is to build a two-dimensional truth table for the function (see Figure 2.1).

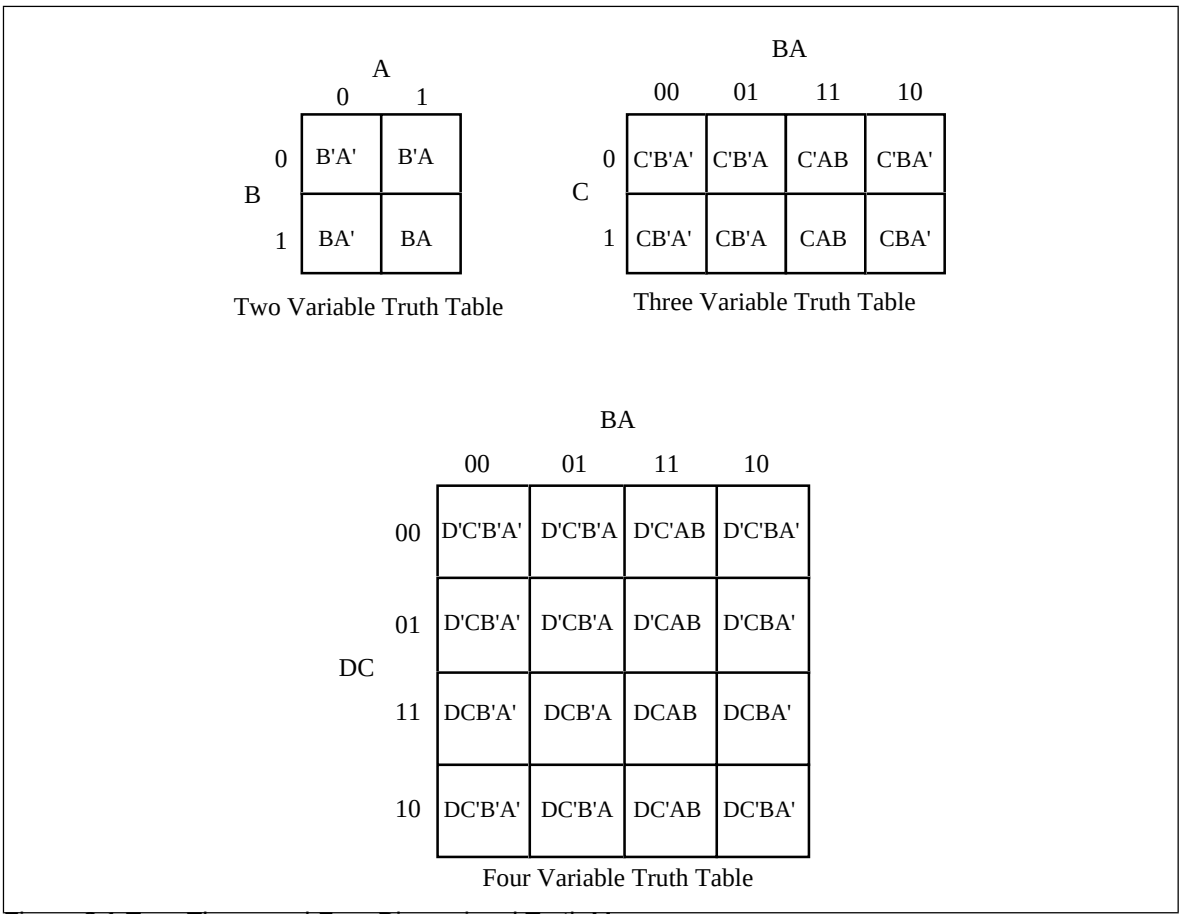

Figure 2.1 Two, Three, and Four Dimensional Truth Maps

Warning: Take a careful look at these truth tables. They do not use the same forms appearing earlier in this chapter. In particular, the progression of the values is 00, 01, 11, 10, not 00, 01, 10, 11. This is very important! If you organize the truth tables in a binary sequence, the mapping optimization method will not work properly. We will call this a *truth map* to distinguish it from the standard truth table.

Assuming your boolean function is in canonical form (sum of minterms), insert ones for each of the truth map entries corresponding to a minterm in the function. Place zeros everywhere else. For example, consider the function of three variables F=C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA. Figure 2.2 shows the truth map for this function.

1. However, it's probably quite reasonable to write a *program* that uses the map method for seven or more variables.

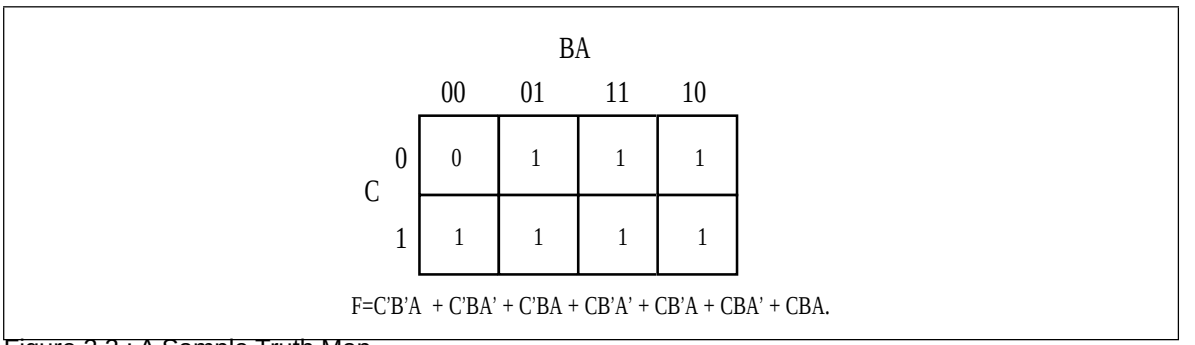

Figure 2.2 : A Sample Truth Map

The next step is to draw rectangles around rectangular groups of ones. The rectangles you enclose must have sides whose lengths are powers of two. For functions of three variables, the rectangles can have sides whose lengths are one, two, and four. The set of rectangles you draw must surround all cells containing ones in the truth map. The trick is to draw all possible rectangles unless a rectangle would be completely enclosed within another. Note that the rectangles may overlap if one does not enclose the other. In the truth map in Figure 2.2 there are three such rectangles (see Figure 2.3)

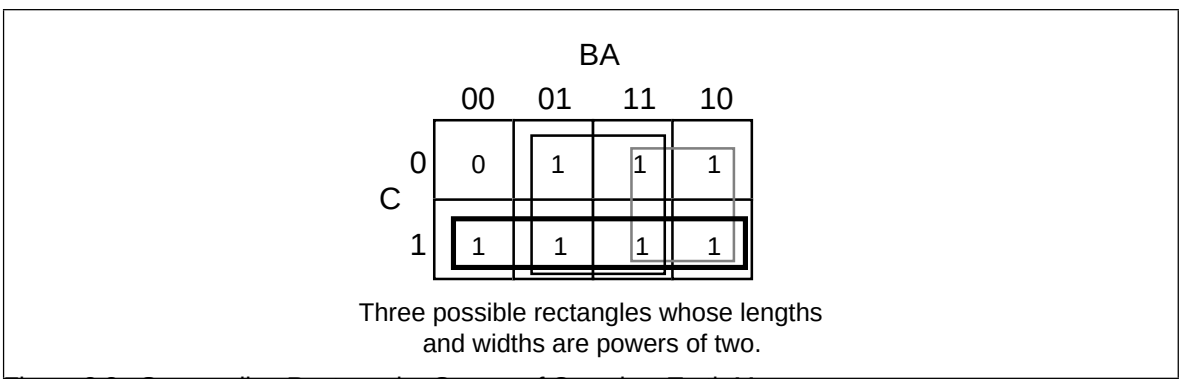

Figure 2.3 : Surrounding Rectangular Groups of Ones in a Truth Map

Each rectangle represents a term in the simplified boolean function. Therefore, the simplified boolean function will contain only three terms. You build each term using the process of elimination. You eliminate any variables whose primed and unprimed form both appear within the rectangle. Consider the long skinny rectangle above that is sitting in the row where C=1. This rectangle contains both A and B in primed and unprimed form. Therefore, we can eliminate A and B from the term. Since the rectangle sits in the C=1 region, this rectangle represents the single literal C.

Now consider the solid square above. This rectangle includes C, C', B, B' and A. Therefore, it represents the single term A. Likewise, the square with the dotted line above contains C, C', A, A' and B. Therefore, it represents the single term B.

The final, optimal, function is the sum (logical OR) of the terms represented by the three squares. Therefore, F= A + B + C. You do not have to consider squares containing zeros.

When enclosing groups of ones in the truth map, you must consider the fact that a truth map forms a *torus* (i.e., a doughnut shape). The right edge of the map *wraps around* to the left edge (and vice-versa). Likewise, the top edge *wraps around* to the bottom edge. This introduces additional possibilities when surrounding groups of ones in a map. Consider the boolean function F=C'B'A' + C'BA' + CB'A' + CBA'. Figure 2.4 shows the truth map for this function.

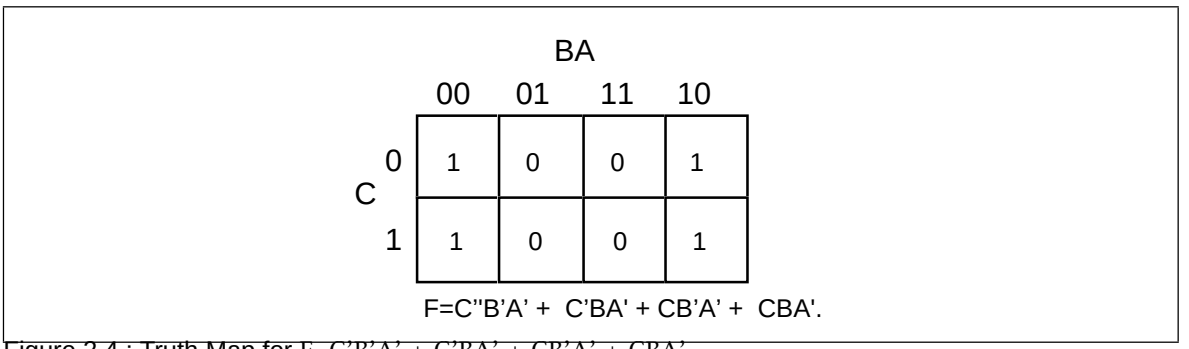

Figure 2.4 : Truth Map for F=C'B'A' + C'BA' + CB'A' + CBA'

At first glance, you would think that there are two possible rectangles here as Figure 2.5 shows. However, because the truth map is a continuous object with the right side and left sides connected, we can form a single, square rectangle, as Figure 2.6 shows.

So what? Why do we care if we have one rectangle or two in the truth map? The answer is because the larger the rectangles are, the more terms they will eliminate. The fewer rectangles that we have, the fewer terms will appear in the final boolean function. For example, the former example with two rectangles generates a function with two terms. The first rectangle (on the left) eliminates the C variable, leaving A'B' as its term. The second rectangle, on the right, also eliminates the C variable, leaving the term BA'. Therefore, this truth map would produce the equation F=A'B' + A'B. We know this is not optimal, see Th 13. Now consider the second truth map above. Here we have a single rectangle so our boolean function will only have a single term. Obviously this is more optimal than an equation with two terms. Since this rectangle includes both C and C' and also B and B', the only term left is A'. This boolean function, therefore, reduces to F=A'.

There are only two cases that the truth map method cannot handle properly: a truth map that contains all zeros or a truth map that contains all ones. These two cases correspond to the boolean functions F=0 and F=1, respectively. These functions are easy to generate by inspection of the truth map.

An important thing you must keep in mind when optimizing boolean functions using the mapping method is that you always want to pick the largest rectangles whose sides'

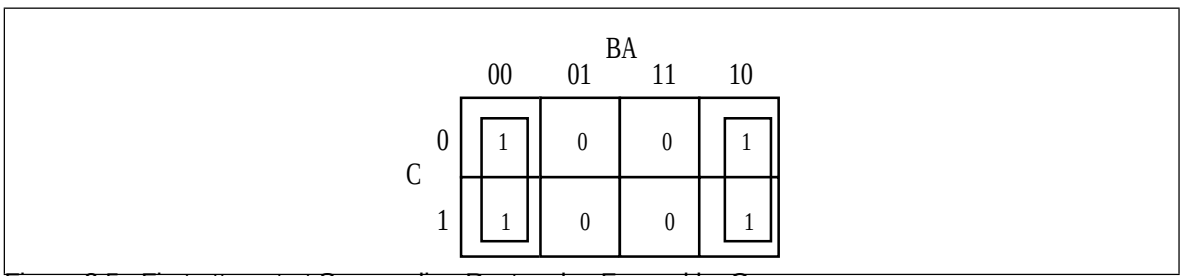

Figure 2.5 : First attempt at Surrounding Rectangles Formed by Ones

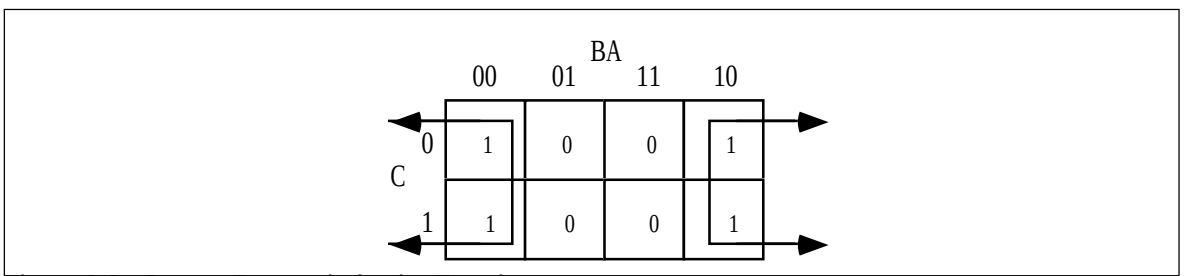

Figure 2.6 : Correct Rectangle for the Function

lengths are a power of two. You must do this even for overlapping rectangles (unless one rectangle encloses another). Consider the boolean function F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA. This produces the truth map appearing in Figure 2.7.

The initial temptation is to create one of the sets of rectangles found in Figure 2.8. However, the correct mapping appears in Figure 2.9.

All three mappings will produce a boolean function with two terms. However, the first two will produce the expressions F= B + A'B' and F = AB + A'. The third form produces F = B + A'. Obviously, this last form is more optimal than the other two forms (see theorems 11 and 12).

For functions of three variables, the size of the rectangle determines the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have three literals.
- A rectangle surrounding two squares containing ones represents a term containing two literals.
- A rectangle surrounding four squares containing ones represents a term containing a single literal.
- A rectangle surrounding eight squares represents the function F = 1.

Truth maps you create for functions of four variables are even trickier. This is because there are lots of places rectangles can hide from you along the edges. Figure 2.10 shows some possible places rectangles can hide.

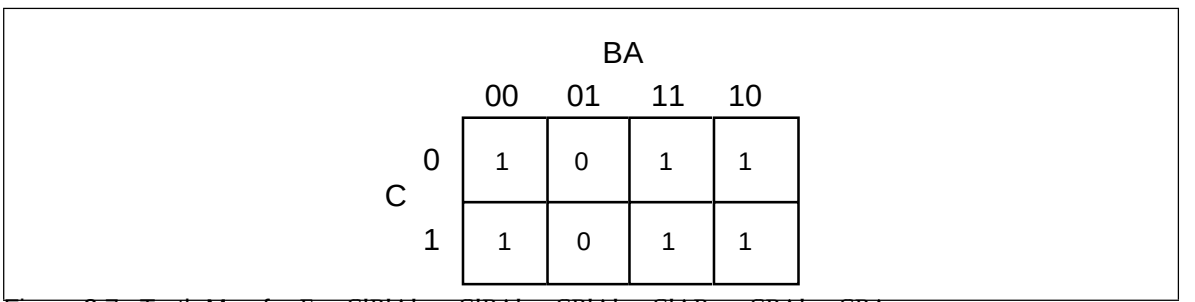

Figure 2.7 : Truth Map for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

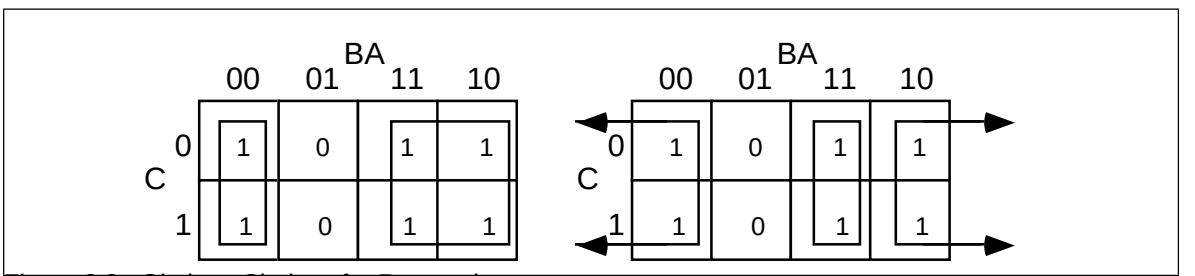

Figure 2.8 : Obvious Choices for Rectangles

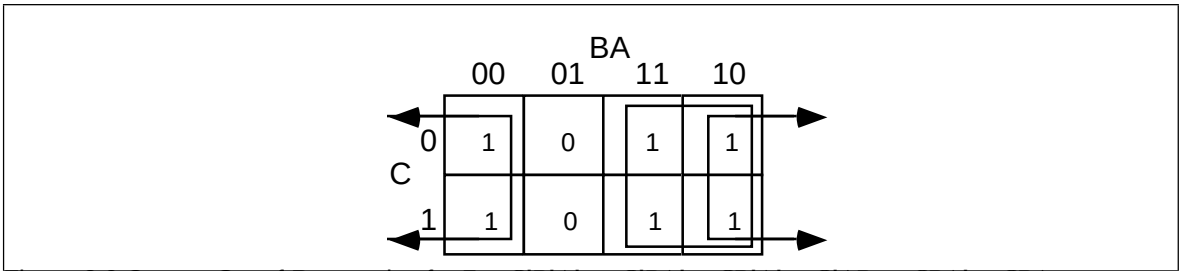

Figure 2.9 Correct Set of Rectangles for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

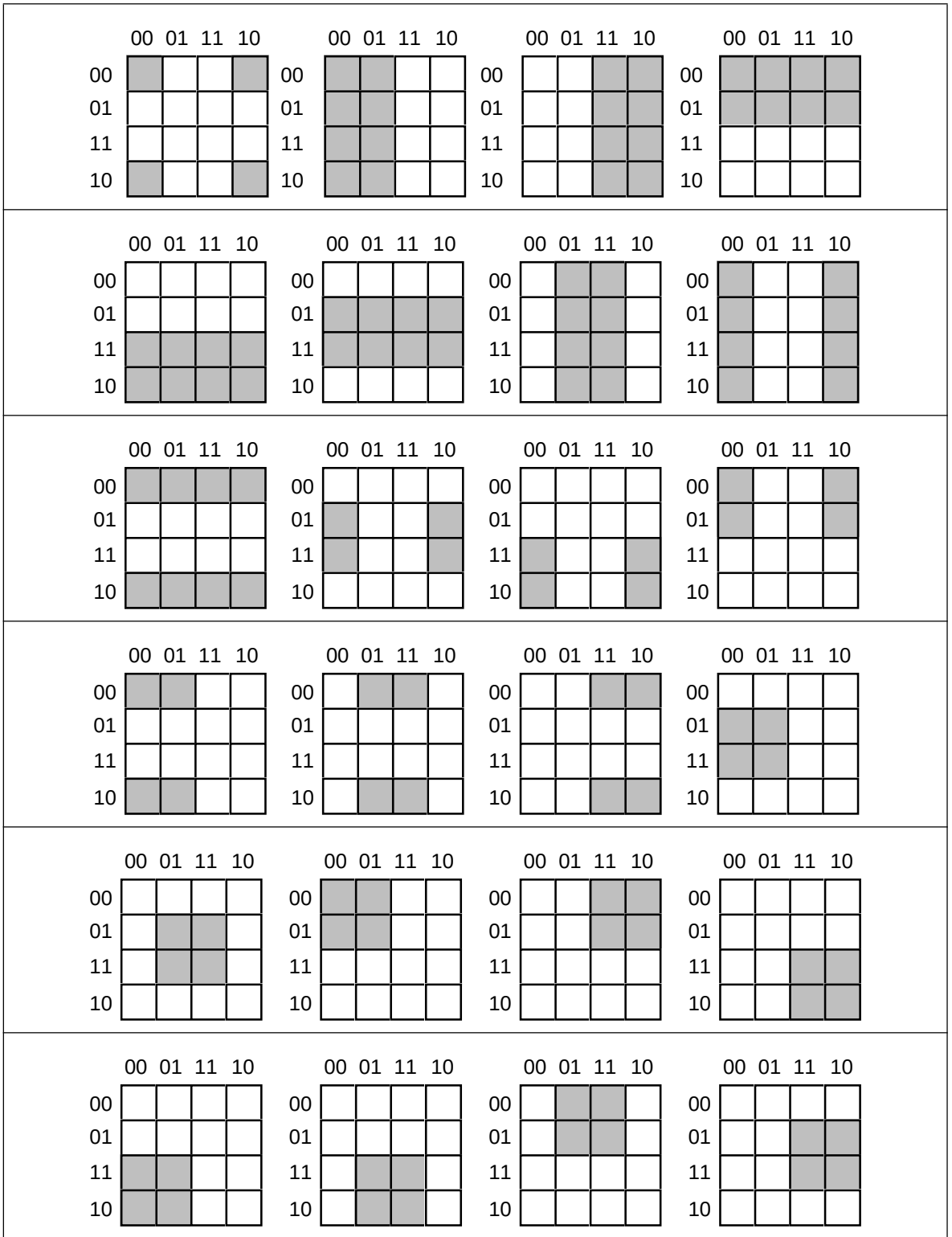

Figure 2.10 : Partial Pattern List for 4x4 Truth Map

This list of patterns doesn't even begin to cover all of them! For example, these diagrams show none of the 1x2 rectangles. You must exercise care when working with four variable maps to ensure you select the largest possible rectangles, especially when overlap occurs. This is particularly important with you have a rectangle next to an edge of the truth map.

As with functions of three variables, the size of the rectangle in a four variable truth map controls the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have four literals.
- A rectangle surrounding two squares containing ones represents a term containing three literals.
- A rectangle surrounding four squares containing ones represents a term containing two literals.
- A rectangle surrounding eight squares containing ones represents a term containing a single literal.
- A rectangle surrounding sixteen squares represents the function F=1.

This last example demonstrates an optimization of a function containing four variables. The function is F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA', the truth map appears in Figure 2.11.

Here are two possible sets of maximal rectangles for this function, each producing three terms (see Figure 2.12). Both functions are equivalent; both are as optimal as you can get[2]. Either will suffice for our purposes.

First, let's consider the term represented by the rectangle formed by the four corners. This rectangle contains B, B', D, and D'; so we can eliminate those terms. The remaining terms contained within these rectangles are C' and A', so this rectangle represents the term C'A'.

The second rectangle, common to both maps in Figure 2.12, is the rectangle formed by the middle four squares. This rectangle includes the terms A, B, B', C, D, and D'. Eliminating B, B', D, and D' (since both primed and unprimed terms exist), we obtain CA as the term for this rectangle.

The map on the left in Figure 2.12 has a third term represented by the top row. This term includes the variables A, A', B, B', C' and D'. Since it contains A, A', B, and B', we can

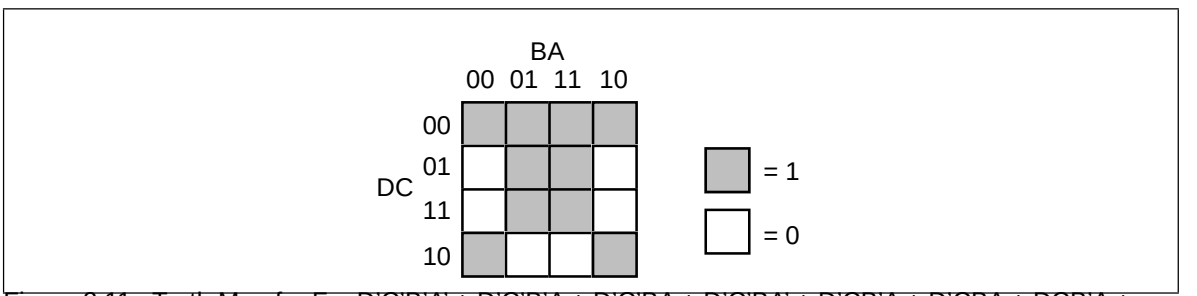

Figure 2.11 : Truth Map for F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'

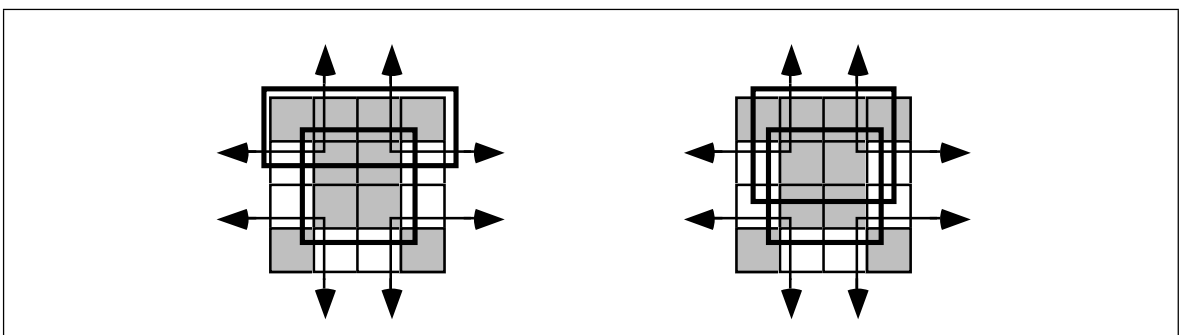

Figure 2.12 : Two Combinations of Surrounded Values Yielding Three Terms

2. Remember, there is no guarantee that there is a unique optimal solution.

eliminate these terms. This leaves the term C'D'. Therefore, the function represented by the map on the left is F=C'A' + CA + C'D'.

The map on the right in Figure 2.12 has a third term represented by the top / middle four squares. This rectangle subsumes the variables A, B, B', C, C', and D'. We can eliminate B, B', C, and C' since both primed and unprimed versions appear, this leaves the term AD. Therefore, the function represented by the function on the right is F=C'A' + CA + AD'.

Since both expressions are equivalent, contain the same number of terms, and the same number of operators, either form is equivalent. Unless there is another reason for choosing one over the other, you can use either form.