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:	$A \bullet 0 = 0$
Th6:	$A + 1 = 1$
Th7:	$(A + B)' = A' \bullet B'$
Th8:	$(A \bullet B)' = A' + B'$
Th9:	$A + A \bullet B = A$
Th10:	$A \bullet (A + B) = A$
Th11:	$A + A'B = A+B$
Th12:	$A' \bullet (A + B') = A'B'$
Th13:	$AB + AB' = A$
Th14:	$(A'+B') \bullet (A' + B) = A'$
Th15:	$A + A' = 1$
Th16:	$A \bullet A' = 0$

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 ($1 \bullet 0 + 1 = 1$).

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 $A \neq B$.
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:

F_{248} = (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:

$$G= (A' + B' + C') \bullet (A + B' + C') \bullet (A' + B + C')$$

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).

		A	
		0	1
B	0	B'A'	B'A
	1	BA'	BA

Two Variable Truth Table

		BA			
		00	01	11	10
C	0	C'B'A'	C'B'A	C'AB	C'BA'
	1	CB'A'	CB'A	CAB	CBA'

Three Variable Truth Table

		BA			
		00	01	11	10
DC	00	D'C'B'A'	D'C'B'A	D'C'AB	D'C'BA'
	01	D'CB'A'	D'CB'A	D'CAB	D'CBA'
	11	DCB'A'	DCB'A	DCAB	DCBA'
	10	DC'B'A'	DC'B'A	DC'AB	DC'BA'

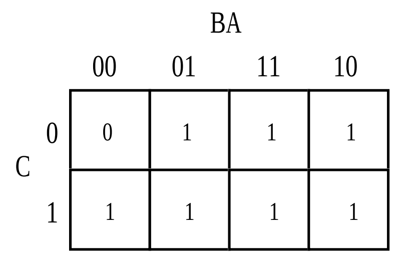
Four Variable Truth Table

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.



F=C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA.

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)

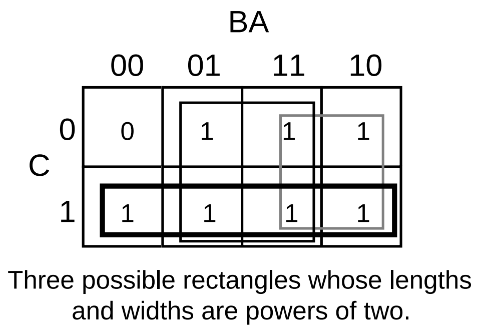

BA
00
01
11
10
C
0
1
0 1 1 1
1 1 1 1
Three possible rectangles whose lengths and widths are powers of two.

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.

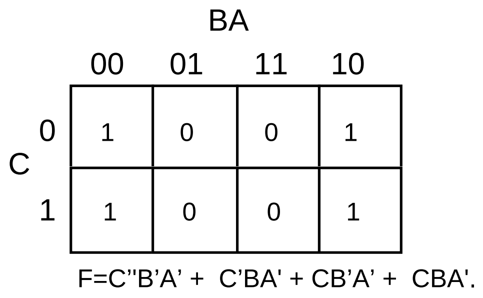


Figure 2.4 : Truth Map for F=C'B'A' + C'BA' + CB'A' + CBA'

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'

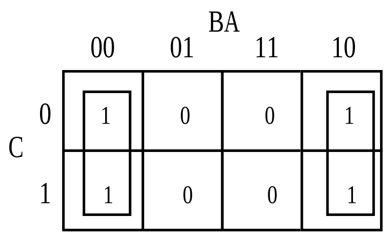


Figure 2.5 : First attempt at Surrounding Rectangles Formed by Ones

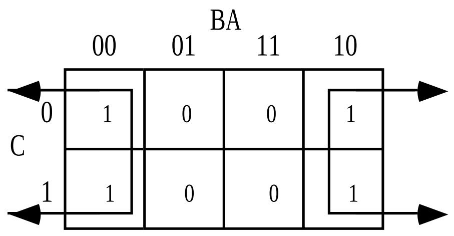


Figure 2.6 : Correct Rectangle for the Function

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.

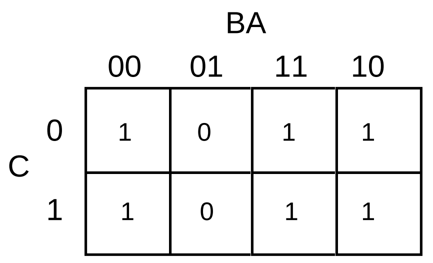


Figure 2.7 : Truth Map for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

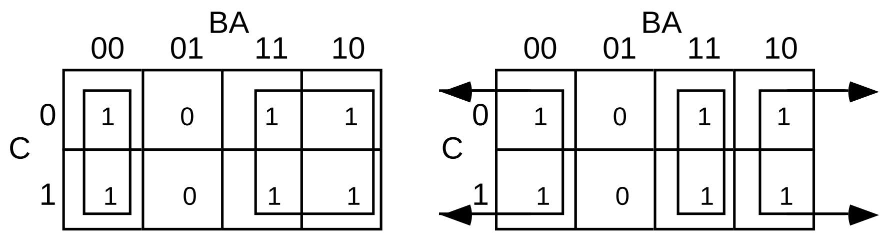


Figure 2.8 : Obvious Choices for Rectangles

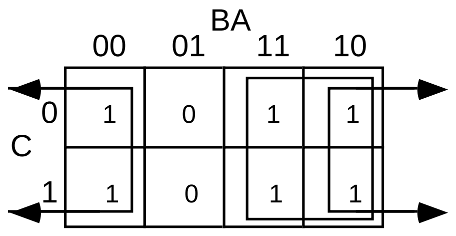


Figure 2.9 Correct Set of Rectangles for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

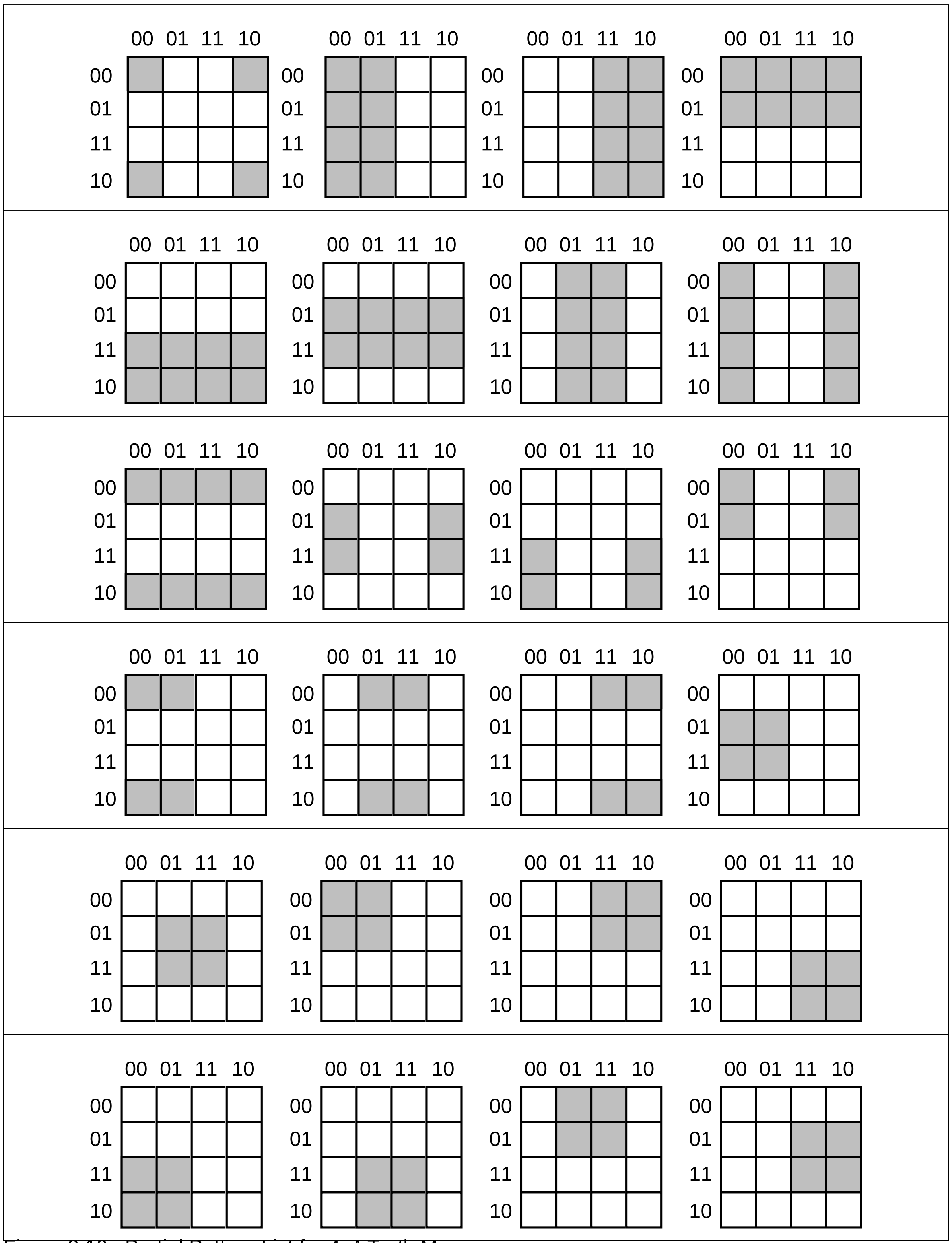


Figure 2.10 : Partial Pattern List for 4x4 Truth Map

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

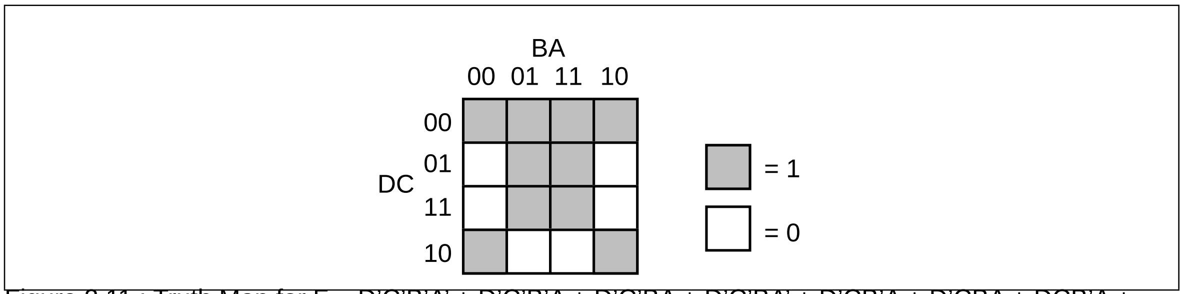


Figure 2.11 : Truth Map for F = D’C’B’A’ + D’C’B’A + D’C’BA + D’C’BA’ + D’CB’A + D’CBA + DCB’A + DCBA + DC’B’A’ + DC’BA’

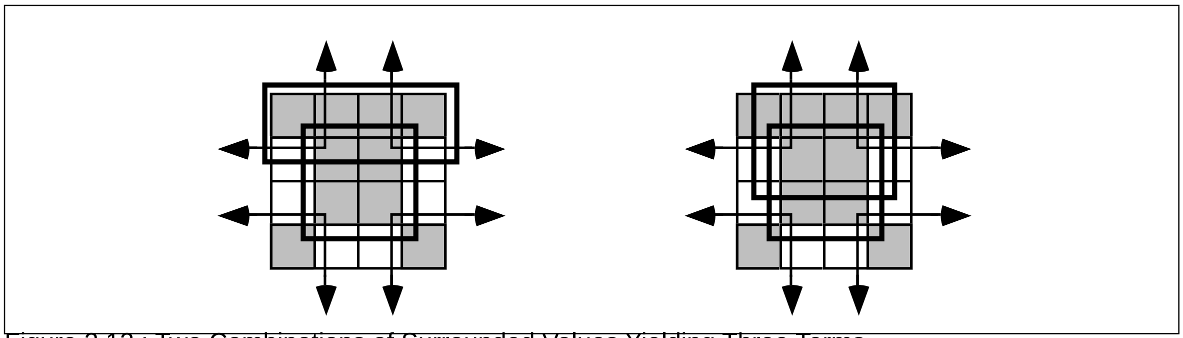


Figure 2.12 : Two Combinations of Surrounded Values Yielding Three Terms

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.

2.6 What Does This Have To Do With Computers, Anyway?

Although there is a tenuous relationship between boolean functions and boolean expressions in programming languages like C or Pascal, it is fair to wonder why we’re spending so much time on this material. However, the relationship between boolean logic and computer systems is much stronger. There is a one-to-one relationship between boolean functions and electronic circuits. Electrical engineers who design CPUs and other computer related circuits need to be intimately familiar with this stuff. Even if you never intend to design your own electronic circuits, understanding this relationship is important if you want to make the most of any computer system.

2.6.1 Correspondence Between Electronic Circuits and Boolean Functions

There is a one-to-one correspondence between an electrical circuits and boolean functions. For any boolean function you can design an electronic circuit and vice versa. Since boolean functions only require the AND, OR, and NOT boolean operators, we can construct any electronic circuit using these operations exclusively. The boolean AND, OR, and NOT functions correspond to the following electronic circuits, the AND, OR, and inverter (NOT) gates (see Figure 2.13).

One interesting fact is that you only need a single gate type to implement *any* electronic circuit. This gate is the *NAND* gate, shown in Figure 2.14.

To prove that we can construct any boolean function using only NAND gates, we need only show how to build an inverter (NOT), AND gate, and OR gate from a NAND (since we can create any boolean function using only AND, NOT, and OR). Building an inverter is easy, just connect the two inputs together (see Figure 2.15).

Once we can build an inverter, building an AND gate is easy – just invert the output of a NAND gate. After all, NOT (NOT (A AND B)) is equivalent to A AND B (see). Of course, this takes *two* NAND gates to construct a single AND gate, but no one said that

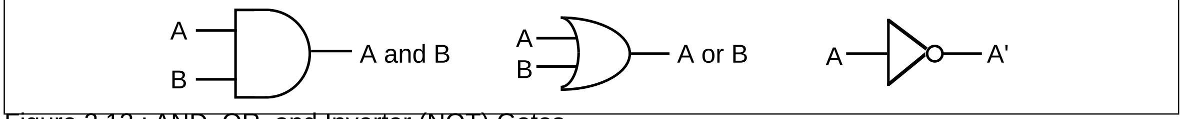


Figure 2.13 : AND, OR, and Inverter (NOT) Gates

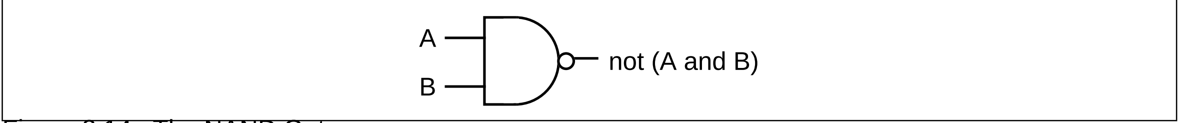


Figure 2.14 : The NAND Gate

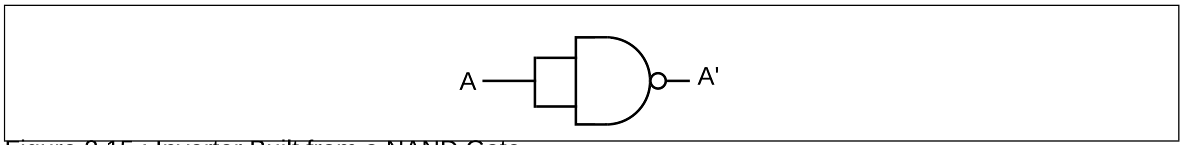


Figure 2.15 : Inverter Built from a NAND Gate

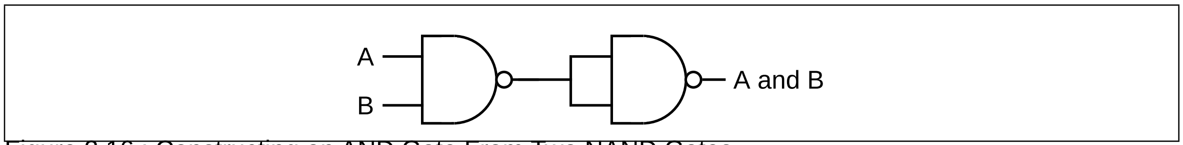


Figure 2.16 : Constructing an AND Gate From Two NAND Gates

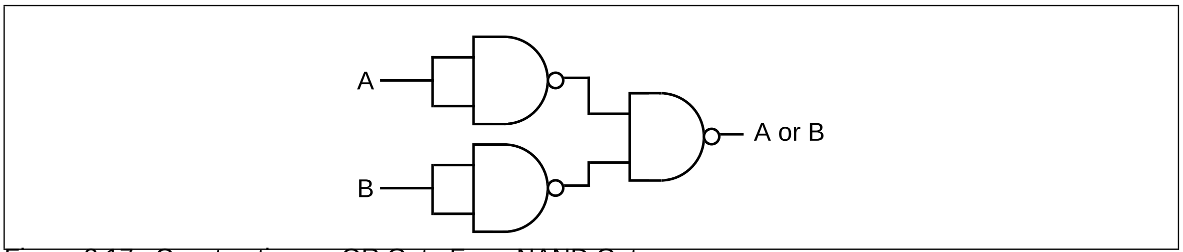


Figure 2.17 : Constructing an OR Gate From NAND Gates

circuits constructed only with NAND gates would be optimal, only that it is possible to do.

The remaining gate we need to synthesize is the logical-OR gate. We can easily construct an OR gate from NAND gates by applying DeMorgan’s theorems.

```
(A or B)’      =          A’ and B’        DeMorgan’s Theorem.
A or B         =          (A’ and B’)’     Invert both sides of the equation.
A or B         =          A’ nand B’       Definition of NAND operation.
```

By applying these transformations, you get the circuit in Figure 2.17.

Now you might be wondering why we would even bother with this. After all, why not just use logical AND, OR, and inverter gates directly? There are two reasons for this. First, NAND gates are generally less expensive to build than other gates. Second, it is also much easier to build up complex integrated circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

Note, by the way, that it is possible to construct any logic circuit using only NOR gates[3]. The correspondence between NAND and NOR logic is orthogonal to the correspondence between the two canonical forms appearing in this chapter (sum of minterms vs. product of maxterms). While NOR logic is useful for many circuits, most electronic designs use NAND logic. See the exercises for more examples.

2.6.2 Combinatorial Circuits

A combinatorial circuit is a system containing basic boolean operations (AND, OR, NOT), some inputs, and a set of outputs. Since each output corresponds to an individual logic function, a combinatorial circuit often implements several different boolean functions. It is very important that you remember this fact – each output represents a different boolean function.

A computer’s CPU is built up from various combinatorial circuits. For example, you can implement an addition circuit using boolean functions. Suppose you have two one-bit

3. NOR is NOT (A OR B).

numbers, A and B. You can produce the one-bit sum and the one-bit carry of this addition using the two boolean functions:

```
S =   AB' + A'B                 Sum of A and B.
C =   AB                        Carry from addition of A and B.
```

These two boolean functions implement a *half-adder*. Electrical engineers call it a half adder because it adds two bits together but cannot add in a carry from a previous operation. A *full adder* adds three one-bit inputs (two bits plus a carry from a previous addition) and produces two outputs: the sum and the carry. The two logic equations for a full adder are

$$S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Although these logic equations only produce a single bit result (ignoring the carry), it is easy to construct an n-bit sum by combining adder circuits (see Figure 2.18). So, as this example clearly illustrates, we can use logic functions to implement arithmetic and boolean operations.

Another common combinatorial circuit is the *seven-segment decoder*. This is a combinatorial circuit that accepts four inputs and determines which of the seven segments on a seven-segment LED display should be on (logic one) or off (logic zero). Since a seven segment display contains seven output values (one for each segment), there will be seven logic functions associated with the display (segment zero through segment six). See Figure 2.19 for the segment assignments. Figure 2.20 shows the segment assignments for each of the ten decimal values.

The four inputs to each of these seven boolean functions are the four bits from a binary number in the range 0..9. Let D be the H.O. bit of this number and A be the L.O. bit of this number. Each logic function should produce a one (segment on) for a given input if that particular segment should be illuminated. For example S_4 (segment four) should be

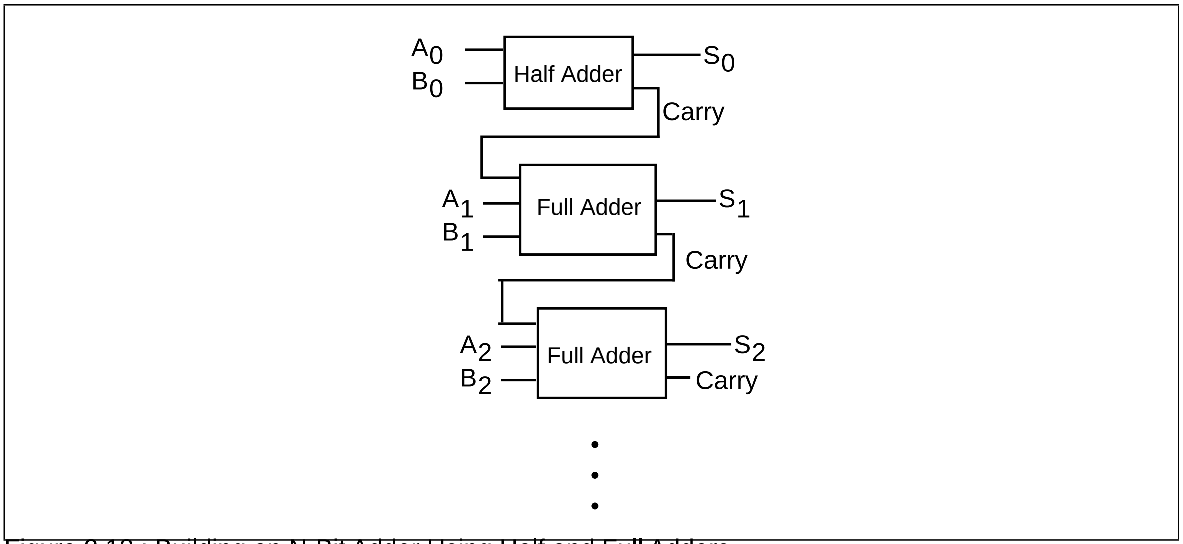


Figure 2.18 : Building an N-Bit Adder Using Half and Full Adders

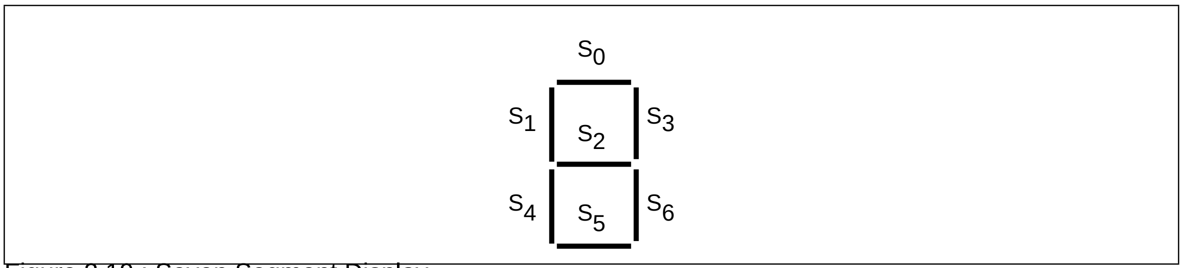


Figure 2.19 : Seven Segment Display

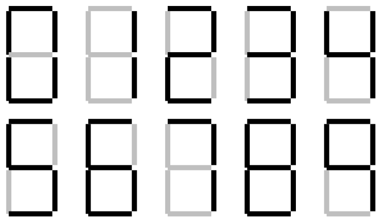


Figure 2.20 : Seven Segment Values for “0” through “9”.

on for binary values 0000, 0010, 0110, and 1000. For each value that illuminates a segment, you will have one minterm in the logic equation:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'.$$

S_o, as a second example, is on for values zero, two, three, five, six, seven, eight, and nine. Therefore, the logic function for S_0 is

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

You can generate the other five logic functions in a similar fashion (see the exercises).

Combinatorial circuits are the basis for many components of a basic computer system. You can construct circuits for addition, subtraction, comparison, multiplication, division, and many other operations using combinatorial logic.

2.6.3 Sequential and Clocked Logic

One major problem with combinatorial logic is that it is *memoryless*. In theory, all logic function outputs depend only on the current inputs. Any change in the input values is immediately reflected in the outputs[4]. Unfortunately, computers need the ability to *remember* the results of past computations. This is the domain of sequential or clocked logic.

A *memory cell* is an electronic circuit that remembers an input value after the removal of that input value. The most basic memory unit is the *set/reset flip-flop*. You can construct an *SR flip-flop* using two NAND gates, as shown in Figure 2.21.

The S and R inputs are normally high. If you *temporarily* set the S input to zero and then bring it back to one (*toggle* the S input), this forces the Q output to one. Likewise, if you toggle the R input from one to zero back to one, this sets the Q output to zero. The Q’ input is generally the inverse of the Q output.

Note that if both S and R are one, then the Q output depends upon Q. That is, whatever Q happens to be, the top NAND gate continues to output that value. If Q was originally one, then there are two ones as inputs to the bottom flip-flop (Q nand R). This

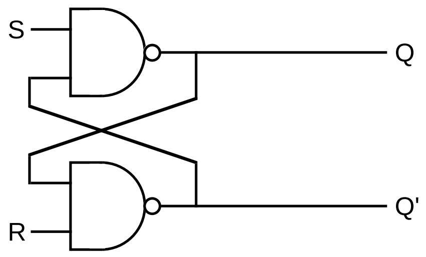


Figure 2.21 : Set/Reset Flip Flop Constructed From NAND Gates

4. In practice, there is a short *propagation delay* between a change in the inputs and the corresponding outputs in any electronic implementation of a boolean function.

produces an output of zero (Q’). Therefore, the two inputs to the top NAND gate are zero and one. This produces the value one as an output (matching the original value for Q).

If the original value for Q was zero, then the inputs to the bottom NAND gate are Q=0 and R=1. Therefore, the output of this NAND gate is one. The inputs to the top NAND gate, therefore, are S=1 and Q’=1. This produces a zero output, the original value of Q.

Suppose Q is zero, S is zero and R is one. This sets the two inputs to the top flip-flop to one and zero, forcing the output (Q) to one. Returning S to the high state does not change the output at all. You can obtain this same result if Q is one, S is zero, and R is one. Again, this produces an output value of one. This value remains one even when S switches from zero to one. Therefore, toggling the S input from one to zero and then back to one produces a one on the output (i.e., *sets* the flip-flop). The same idea applies to the R input, except it forces the Q output to zero rather than to one.

There is one catch to this circuit. It does not operate properly if you set both the S and R inputs to zero simultaneously. This forces both the Q and Q’ outputs to one (which is logically inconsistent). Whichever input remains zero the longest determines the final state of the flip-flop. A flip-flop operating in this mode is said to be *unstable*.

The only problem with the S/R flip-flop is that you must use separate inputs to remember a zero or a one value. A memory cell would be more valuable to us if we could specify the data value to remember on one input and provide a *clock input* to *latch* the input value. This type of flip-flop, the D flip-flop (for *data*) uses the circuit in Figure 2.22.

Assuming you fix the Q and Q’ outputs to either 0/1 or 1/0, sending a *clock pulse* that goes from zero to one back to zero will copy the D input to the Q output. It will also copy D’ to Q’. The exercises at the end of this chapter will expect you to describe this operation in detail, so study this diagram carefully.

Although remembering a single bit is often important, in most computer systems you will want to remember a group of bits. You can remember a sequence of bits by combining several D flip-flops in parallel. Concatenating flip-flops to store an n-bit value forms a *register*. The electronic schematic in Figure 2.23 shows how to build an eight-bit register from a set of D flip-flops.

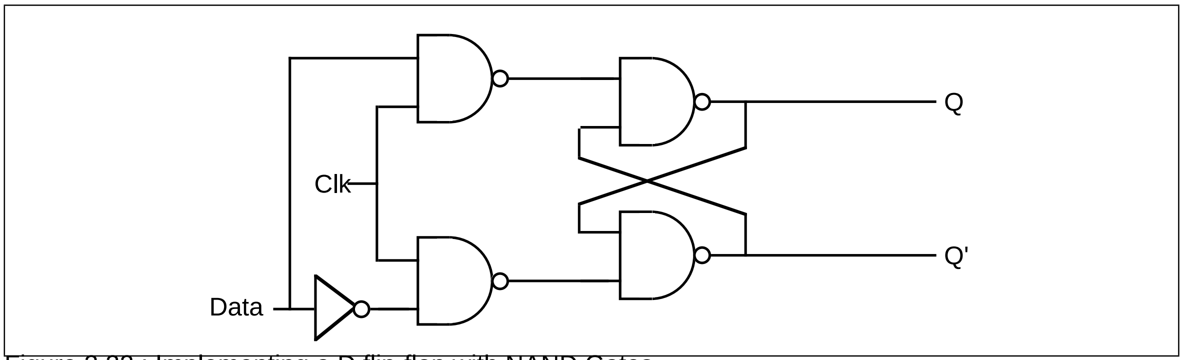


Figure 2.22 : Implementing a D flip-flop with NAND Gates

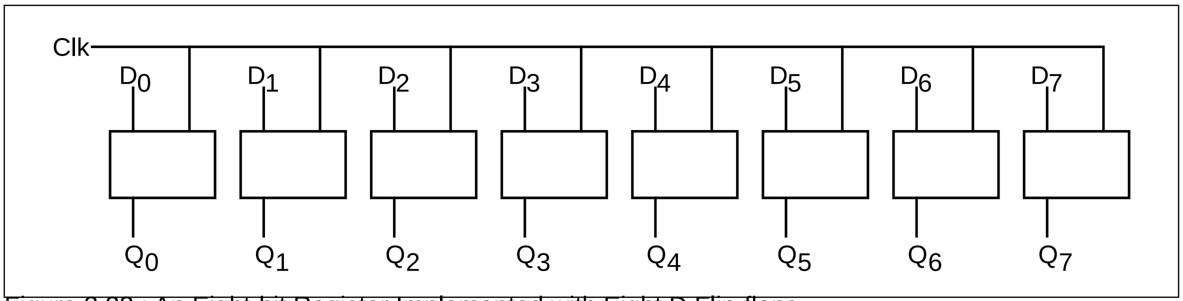


Figure 2.23 : An Eight-bit Register Implemented with Eight D Flip-flops

Note that the eight D flip-flops use a common clock line. This diagram does not show the Q' outputs on the flip-flops since they are rarely required in a register.

D flip-flops are useful for building many sequential circuits above and beyond simple registers. For example, you can build a *shift register* that shifts the bits one position to the left on each clock pulse. A four-bit shift register appears in Figure 2.24.

You can even build a *counter*, that counts the number of times the clock toggles from one to zero and back to one using flip-flops. The circuit in Figure 2.25 implements a four bit counter using D flip-flops.

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits beyond these.

2.7 Okay, What Does It Have To Do With Programming, Then?

Once you have registers, counters, and shift registers, you can build *state machines*. The implementation of an algorithm in hardware using state machines is well beyond the scope of this text. However, one important point must be made with respect to such circuitry – *any algorithm you can implement in software you can also implement directly in hardware*. This suggests that boolean logic is the basis for computation on all modern computer systems. Any program you can write, you can specify as a sequence of boolean equations.

Of course, it is much easier to specify a solution to a programming problem using languages like Pascal, C, or even assembly language than it is to specify the solution using boolean equations. Therefore, it is unlikely that you would ever implement an entire program using a set of state machines and other logic circuitry. Nevertheless, there are times when a hardware implementation is better. A hardware solution can be one, two, three, or more *orders of magnitude* faster than an equivalent software solution. Therefore, some time critical operations may require a hardware solution.

A more interesting fact is that the converse of the above statement is also true. Not only can you implement all software functions in hardware, but it is also possible to *implement all hardware functions in software*. This is an important revelation because many operations you would normally implement in hardware are *much cheaper* to implement using software on a microprocessor. Indeed, this is a primary use of *assembly language* in modern

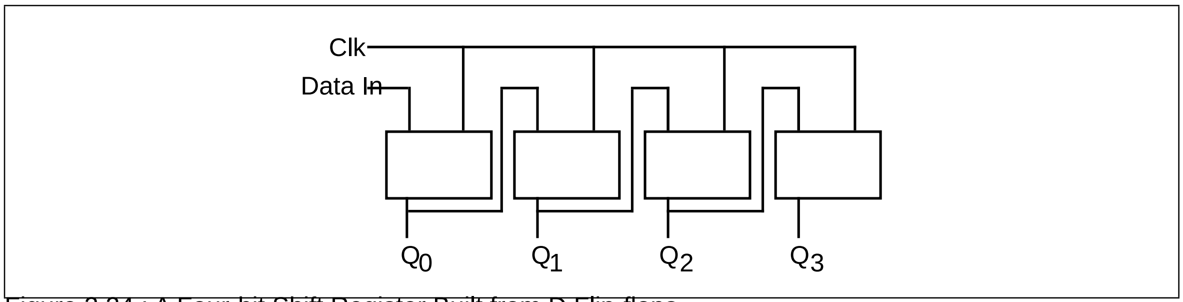


Figure 2.24 : A Four-bit Shift Register Built from D Flip-flops

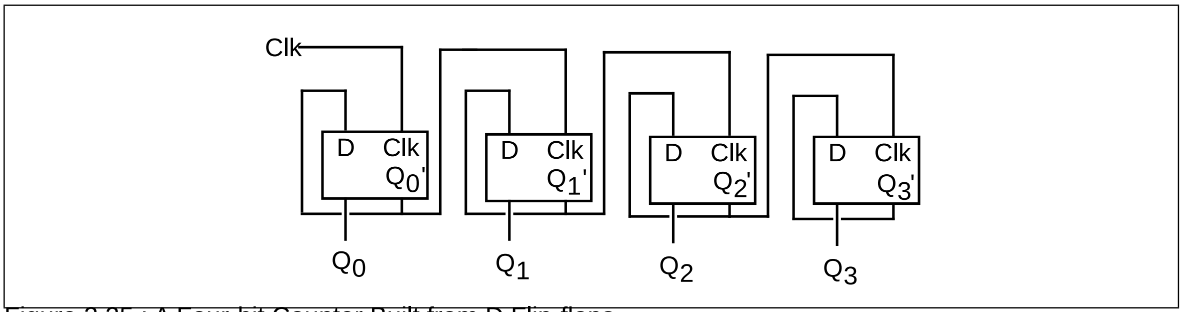


Figure 2.25 : A Four-bit Counter Built from D Flip-flops

systems – to inexpensively replace a complex electronic circuit. It is often possible to replace many tens or hundreds of dollars of electronic components with a single $25 microcomputer chip. The whole field of *embedded systems* deals with this very problem. Embedded systems are computer systems embedded in other products. For example, most microwave ovens, TV sets, video games, CD players, and other consumer devices contain one or more complete computer systems whose sole purpose is to replace a complex hardware design. Engineers use computers for this purpose because they are *less expensive* and *easier to design with* than traditional electronic circuitry.

You can easily design software that reads switches (input variables) and turns on motors, LEDs or lights, locks or unlocks a door, etc. (output functions). To write such software, you will need an understanding of boolean functions and how to implement such functions in software.

Of course, there is one other reason for studying boolean functions, even if you never intend to write software intended for an embedded system or write software that manipulates real-world devices. Many high level languages process boolean expressions (e.g., those expressions that control an if statement or while loop). By applying transformations like DeMorgan's theorems or a mapping optimization it is often possible to improve the performance of high level language code. Therefore, studying boolean functions *is* important even if you never intend to design an electronic circuit. It can help you write better code in a traditional programming language.

For example, suppose you have the following statement in Pascal:

```
if ((x=y) and (a <> b)) or ((x=y) and (c <= d)) then SomeStmt;
```

You can use the distributive law to simplify this to:

```
if ((x=y) and ((a <> b) or (c <= d)) then SomeStmt;
```

Likewise, we can use DeMorgan's theorem to reduce

```
while (not((a=b) and (c=d)) do Something;
```

to

```
while (a <> b) or (c <> d) do Something;
```

2.8 Generic Boolean Functions

For a specific application, you can create a logic function that achieves some specific result. Suppose, however, that you wanted to write a program to *simulate* any possible boolean function? For example, on the companion diskette, there is a program that lets you enter an arbitrary boolean function with one to four different variables. This program will read the inputs and produce and necessary function results. Since the number of unique four variable functions is large (65,536, to be exact), it is not practical to include a specific solution for each one in a program. What is necessary is a *generic logic function*, one that will compute the results for any arbitrary function. This section describes how to write such a function.

A generic boolean function of four variables requires five parameters – the four input parameters and a fifth parameter that specifies the function to compute. While there are lots of ways to specify the function to compute, we'll pass the boolean function's number as this fifth parameter.

At first glance you might wonder how we can compute a function using the function's number. However, keep in mind that the bits that make up the function's number come directly from the truth table for that function. Therefore, if we extract the bits from the function's number, we can construct the truth table for that function. Indeed, if we just select the i^{th} bit of the function number, where i = D*8 + C*4 + B*2 +A you will get the function result for that particular value of A, B, C, and D[5]. The following examples, in C and Pascal, show how to write such functions:

```
/************************************************************************/
/*                                                                      */
/* This C program demonstrates how to write a generic logic function    */
/* that can compute any logic function of four variables.  Given C’s    */
/* bit manipulation operators, along with hexadecimal I/O, this is an   */
/* easy task to accomplish in the C programming language.               */
/*                                                                      */
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>


/* Generic logic function.  The “Func” parameter contains the 16-bit    */
/* logical function number.  This is actually an encoded truth table    */
/* for the function.  The a, b, c, and d parameters are the inputs to   */
/* the logic function.  If we treat “func” as a 2x2x2x2 array of bits,  */
/* this particular function selects bit “func[d,c,b,a]” from func.      */

int
generic(int func, int a, int b, int c, int d)
{
        /* Return the bit specified by a, b, c, and d */

        return (func >> (a + b*2 + c*4 + d*8)) & 1;
}




/* Main program to drive the generic logic function written in C.       */

main()
{
        int func, a, b, c, d;

        /* Repeat the following until the user enters zero.             */

        do
        {
                /* Get the function’s number (truth table)              */

                printf(“Enter function value (hex): “);
                scanf(“%x”, &func);

                /* If the user specified zero as the function #, stop   */
                /* the program.                                          */

                if (func != 0)
                {
                        printf(“Enter values for d, c, b, & a: “);
                        scanf(“%d%d%d%d”,
                                &d, &c, &b, &a);

                        printf(“The result is %d\n”, generic(func,a,b,c,d));
                        printf(“Func = %x, A=%d, B=%d, C=%d, D=%d\n”,
                                func, a, b, c, d);
                }

        } while (func !=0);

}
```

The following Pascal program is written for *Standard Pascal*. Standard Pascal does not provide any bit manipulation operations, so this program is lengthy since it has to simulate bits using an array of integers. Most modern Pascals (especially Turbo Pascal) provide built-in bit operations or library routines that operate on bits. This program would be much easier to write using such non-standard features.

5. Chapter Five explains why this multiplication works.

```
program GenericFunc(input,output);

(* Since standard Pascal does not provide an easy way to directly man-  *)
(* ipulate bits in an integer, we will simulate the function number     *)
(* using an array of 16 integers. “GFTYPE” is the type of that array.   *)

type
    gftype = array [0..15] of integer;

var
   a, b, c, d:integer;
   fresult:integer;
   func: gftype;


(* Standard Pascal does not provide the ability to shift integer data   *)
(* to the left or right.  Therefore, we will simulate a 16-bit value    *)
(* using an array of 16 integers.  We can simulate shifts by moving     *)
(* data around in the array.                                            *)
(*                                                                      *)
(* Note that Turbo Pascal *does* provide shl and shr operators.  How-   *)
(* ever, this code is written to work with standard Pascal, not just    *)
(* Turbo Pascal.                                                        *)
(*                                                                      *)
(* ShiftLeft shifts the values in func on position to the left and in-  *)
(* serts the shiftin value into “bit position” zero.                    *)

procedure ShiftLeft(shiftin:integer);
var i:integer;
begin

     for i := 15 downto 1 do func[i] := func[i-1];
     func[0] := shiftin;

end;


(* ShiftNibble shifts the data in func to the left four positions and   *)
(* inserts the four bits a (L.O.), b, c, and d (H.O.) into the vacated  *)
(* positions.                                                           *)

procedure ShiftNibble(d,c,b,a:integer);
begin

     ShiftLeft(d);
     ShiftLeft(c);
     ShiftLeft(b);
     ShiftLeft(a);
end;


(* ShiftRight shifts the data in func one position to the right.  It    *)
(* shifts a zero into the H.O. bit of the array.                        *)

procedure ShiftRight;
var i:integer;
begin

     for i := 0 to 14 do func[i] := func[i+1];
     func[15] := 0;

end;


(* ToUpper converts a lower case character to upper case.               *)

procedure toupper(var ch:char);
begin

     if (ch in [‘a’..’z’]) then ch := chr(ord(ch) - 32);

end;


(* ReadFunc reads a hexadecimal function number from the user and puts  *)
(* this value into the func array (bit by bit).                         *)

function ReadFunc:integer;
```

```
var ch:char;
    i, val:integer;
begin

     write(‘Enter function number (hexadecimal): ‘);
     for i := 0 to 15 do func[i] := 0;
     repeat

           read(ch);
           if not eoln then begin

                      toupper(ch);
                      case ch of
                           ‘0’: ShiftNibble(0,0,0,0);
                           ‘1’: ShiftNibble(0,0,0,1);
                           ‘2’: ShiftNibble(0,0,1,0);
                           ‘3’: ShiftNibble(0,0,1,1);
                           ‘4’: ShiftNibble(0,1,0,0);
                           ‘5’: ShiftNibble(0,1,0,1);
                           ‘6’: ShiftNibble(0,1,1,0);
                           ‘7’: ShiftNibble(0,1,1,1);
                           ‘8’: ShiftNibble(1,0,0,0);
                           ‘9’: ShiftNibble(1,0,0,1);
                           ‘A’: ShiftNibble(1,0,1,0);
                           ‘B’: ShiftNibble(1,0,1,1);
                           ‘C’: ShiftNibble(1,1,0,0);
                           ‘D’: ShiftNibble(1,1,0,1);
                           ‘E’: ShiftNibble(1,1,1,0);
                           ‘F’: ShiftNibble(1,1,1,1);
                           else write(chr(7),chr(8));
                      end;
           end;
     until eoln;
     val := 0;
     for i := 0 to 15 do val := val + func[i];
     ReadFunc := val;
end;


(* Generic - Computes the generic logical function specified by *)
(*           the function number “func” on the four input vars  *)
(*           a, b, c, and d.  It does this by returning bit     *)
(*           d*8 + c*4 + b*2 + a from func.                     *)

function Generic(var func:gftype; a,b,c,d:integer):integer;
begin
          Generic := func[a + b*2 + c*4 + d*8];
end;


begin (* main *)

      repeat

            fresult := ReadFunc;
            if (fresult <> 0) then begin

               write(‘Enter values for D, C, B, & A (0/1):’);
               readln(d, c, b, a);
               writeln(‘The result is ‘,Generic(func,a,b,c,d));

            end;
      until fresult = 0;

end.
```

The following code demonstrates the power of bit manipulation operations. This version of the code above uses special features present in the Turbo Pascal programming language that allows programmers to shift left or right and do a bitwise logical AND on integer variables:

```
program GenericFunc(input,output);
const
     hex = [‘a’..’f’, ‘A’..’F’];
     decimal = [‘0’..’9’];

var
```

```
   a, b, c, d:integer;
   fresult:integer;
   func: integer;


(* Here is a second version of the Pascal generic function that uses *)
(* the features of Turbo Pascal to simplify the program.             *)


function ReadFunc:integer;
var ch:char;
    i, val:integer;
begin

     write('Enter function number (hexadecimal): ');
     repeat

           read(ch);
           func := 0;
           if not eoln then begin

              if (ch in Hex) then
                func := (func shl 4) + (ord(ch) and 15) + 9
              else if (ch in Decimal) then
                func := (func shl 4) + (ord(ch) and 15)
              else write(chr(7));

           end;
     until eoln;
     ReadFunc := func;
end;


(* Generic - Computes the generic logical function specified by *)
(*           the function number "func" on the four input vars  *)
(*           a, b, c, and d.  It does this by returning bit     *)
(*           d*8 + c*4 + b*2 + a from func.  This version re-   *)
(*           lies on Turbo Pascal's shift right operator and    *)
(*           its ability to do bitwise operations on integers.  *)

function Generic(func,a,b,c,d:integer):integer;
begin
          Generic := (func shr (a + b*2 + c*4 + d*8)) and 1;
end;


begin (* main *)

      repeat

            fresult := ReadFunc;
            if (fresult <> 0) then begin

               write('Enter values for D, C, B, & A (0/1):');
               readln(d, c, b, a);
               writeln('The result is ',Generic(func,a,b,c,d));

            end;
      until fresult = 0;

end.
```

2.9 Laboratory Exercises

This laboratory uses several Windows programs to manipulate truth tables and logic expressions, optimize logic equations, and simulate logic equations. These programs will help you understand the relationship between logic equations and truth tables as well as gain a fuller understanding of logic systems.

The *WLOGIC.EXE* program simulates logic circuitry. WLOGIC stores several logic equations that describe an electronic circuit and then it simulates that circuit using "switches" as inputs and "LEDs" as outputs. For those who would like a more "real-world" laboratory, there is an optional program you can run from DOS, *LOGICEV.EXE*, that controls a real set of LEDs and switches that you construct and attach to the PC's parallel port. The directions for constructing this hardware appear in the appendices. The use of *either* program will let you easily observe the behavior of a set of logic functions.

If you haven't done so already, please install the software for this text on your machine. See the laboratory exercises in Chapter One for more details.

2.9.1 Truth Tables and Logic Equations Exercises

In this laboratory exercise you will create several different truth tables of two, three, and four variables. The TRUTHTBL.EXE program (found in the CH2 subdirectory) will automatically convert the truth tables you input into logic equations in the sum of minterms canonical form.

The TRUTHTBL.EXE file is a Windows program; it requires some version of Windows for proper operation. In particular, it will not run properly from DOS. It should, however, work just fine with any version of Windows from Windows 3.1 on up.

The TRUTHTBL.EXE program provides three buttons that let you choose a two variable, three variable, or four variable truth table. Pressing one of these buttons rearranges the truth table in an appropriate fashion. By default, the TRUTHTBL program assumes you want to work with a four variable truth table. Try pressing the *Two Variables, Three Variables,* and *Four Variables* buttons and observe the results. Describe what happens in your lab report.

To change the truth table entries, all you need do is click on the square associated with the truth table value you want to change. Clicking on one of these boxes toggles (inverts) that value in that square. For example, try clicking on the DCBA square several times and observe the results.

Note that as you click on different truth table entries, the TRUTHTBL program automatically recomputes the sum of minterms canonical logic equation and displays it at the bottom of the window. What equation does the program display if you set all squares in the truth table to zero?[6]

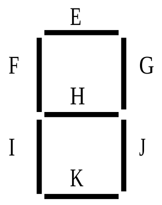
Set up the TRUTHTBL program to work with four variables. Set the DCBA square to one. Now press the *Two Variables* button. Press the *Four Variables* button and set *all* the squares to one. Now press the *Two Variables* button again. Finally, press the *Four Variables* button and examine the results. What does the TRUTHTBL program do when you switch between different sized truth tables? Feel free to try additional experiments to verify your hypothesis. Describe your results in your lab report.

Switch to two variable mode. Input the truth tables for the logical AND, OR, XOR, and NAND truth tables. Verify the correctness of the resulting logic equations. Write up the results in your lab report. Note: if there is a Windows-compatible printer attached to your computer, you can print each truth table you create by pressing the Print button in the window. This makes it very easy to include the truth table and corresponding logic equation in your lab report. **For additional credit:** input the truth tables for all 16 functions of two variables. In your lab report, present the results for these 16 functions.

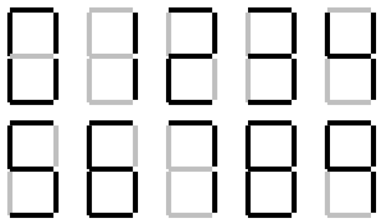
Design several two, three, and four variable truth tables by hand. Manually determine their logic equations in sum of minterms canonical form. Input the truth tables and verify the correctness of your logic equations. Include your hand designed truth tables and logic equations as well as the machine produced versions in your lab report.

6. Note: On initial entry to the program, TRUTHTBL does not display a logic equation. Therefore, you will need to set at least one square to one and then back to zero to see this equation.

Consider the following layout for a seven-segment display:



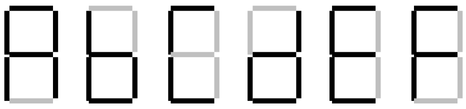
Here are the segments to light for the binary values DCBA = 0000 - 1001:



E = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A
F = D'C'B'A'+ D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
G = D'C'B'A' + D'C'B'A + D'C'BA' + D'C'BA + D'CB'A' + D'CBA + DC'B'A' + DC'B'A
H = D'C'BA' + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
I = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'
J = D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A' + D'CB'A +D'CBA' + D'CBA + DC'B'A' + DC'B'A
K = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + DC'B'A'

Convert each of these logic equations to a truth table by setting all entries in the table to zero and then clicking on each square corresponding to each minterm in the equation. Verify by observing the equation that TRUTHTBL produces that you've successfully converted each equation to a truth table. Describe the results and provide the truth tables in your lab report.

For Additional Credit: Modify the equations above to include the following hexadecimal characters. Determine the new truth tables and use the TRUTHTBL program to verify that your truth tables and logic equations are correct.



2.9.2 Canonical Logic Equations Exercises

In this laboratory you will enter several different logic equations and compute their canonical forms as well as generate their truth table. In a sense, this exercise is the opposite of the previous exercise where you generated a canonical logic equation from a truth table.

This exercise uses the CANON.EXE program found in the CH2 subdirectory. Run this program from Windows by double clicking on its icon. This program displays a text box, a truth table, and several buttons. Unlike the TRUTHTBL.EXE program from the previous exercise, you cannot modify the truth table in the CANON.EXE program; it is a display-only table. In this program you will enter logic equations in the text entry box and then press the "Compute" button to see the resulting truth table. This program also produces the sum of minterms canonical form for the logic equation you enter (hence this program's name).

Valid logic equations take the following form:

- A *term* is either a variable (A, B, C, or D) or a logic expression surrounded by parentheses.

- A *factor* is either a term, or a factor followed by the prime symbol (an apostrophe, i.e., "‘"). The prime symbol logically negates the factor immediately preceding it.
- A *product* is either a factor, or a factor concatenated with a product. The concatenation denotes logical AND operation.
- An expression is either a product or a product followed by a "+" (denoting logical OR) and followed by another expression.

Note that logical OR has the lowest precedence, logical AND has an intermediate precedence, and logical NOT has the highest precedence of these three operators. You can use parentheses to override operator precedence. The logical NOT operator, since its precedence is so high, applies only to a variable or a parenthesized expression. The following are all examples of legal expressions:

```
AB′C + D(B′+C′)
AB(C+D)′ + A′B′(C+D)
A′B′C′D′ + ABCD + A(B+C)
(A+B)′ + A′B′
```

For this set of exercises, you should create several logic expression and feed them through CANON.EXE. Include the truth tables and canonical logic forms in your lab report. Also verify that the theorems appearing in this chapter (See "Boolean Algebra" on page 43.) are valid by entering each side of the theorem and verifying that they both produce the same truth table (e.g., (AB)’ = A’ + B’). For additional credit, create several complex logic equations and generate their truth tables and canonical forms by hand. Then input them into the CANON.EXE program to verify your work.

2.9.3 Optimization Exercises

In this set of laboratory exercises, the OPTIMZP.EXE program (found in the CH2 subdirectory) will guide you through the steps of logic function optimization. The OPTIMZP.EXE program uses the Karnaugh Map technique to produce an equation with the minimal number of terms.

Run the OPTIMZP.EXE program by clicking on its icon or running the OPTIMZP.EXE program using the program manager’s File|Run menu option. This program lets you enter an arbitrary logic equation using the same syntax as the CANON.EXE program in the previous exercise.

After entering an equation press the "Optimize" button in the OPTIMZP.EXE window. This will construct the truth table, canonical equation, and an optimized form of the logic equation you enter. Once you have optimized the equation, OPTIMZP.EXE enables the "Step" button. Pressing this button walks you through the optimization process step-by-step.

For this exercise you should enter the seven equations for the seven-segment display. Generate and record the optimize versions of these equations for your lab report and the next set of exercises. Single step through each of the equations to make sure you understand how OPTIMZP.EXE produces the optimal expressions.

For additional credit: OPTIMZP.EXE generates a single optimal expression for any given logic function. Other optimal functions may exist. Using the Karnaugh mapping technique, see if you can determine if other, equivalent, optimal expressions exist. Feed the optimal equations OPTIMZP.EXE produces and your optimal expressions into the CANON.EXE program to verify that their canonical forms are identical (and, hence, the functions are equivalent.

2.9.4 Logic Evaluation Exercises

In this set of laboratory exercises you will use the LOGIC.EXE program to enter, edit, initialize, and evaluation logic expressions. This program lets you enter up to 22 distinct logic equations involving as many as 26 variables plus a clock value. LOGIC.EXE provides four input variables and 11 output variables (four simulated LEDs and a simulated seven-segment display). Note: this program requires that you install two files in your WINDOWS\SYSTEM directory. Please see the README.TXT file in the CH2 subdirectory for more details.

Execute the LOGIC.EXE program by double-clicking on its icon or using the program manager's "File | Run" menu option. This program consists of three main parts: an equation editor, an initialization screen, and an execution module. LOGIC.EVE uses a set of *tabbed notebook screens* to switch between these three modules. By clicking on the "Create", *Initialize*, and *Execute* tabs at the top of the screen with your mouse, you can select the specific module you want to use. Typically, you would first create a set of equations on the *Create* page and then execute those functions on the *Execute* page. Optionally, you can initialize any necessary logic variables (D-Z) on the *Initialize* page. At any time you can easily switch between modules by pressing on the appropriate notebook tab. For example, you could create a set of equations, execute them, and then go back and modify the equations (e.g., to correct any mistakes) by pressing on the *Create* tab.

The Create page lets you add, edit, and delete logic equations. Logic equations may use the variables A-Z plus the "#" symbol ("#" denotes the clock). The equations use a syntax that is very similar to the logic expressions you've used in previous exercises in this chapter. In fact, there are only two major differences between the functions LOGIC.EXE allows and the functions that the other programs allow. First, LOGIC.EXE lets you use the variables A-Z and "#" (the other programs only let you enter functions of four variables using A-D). The second difference is that LOGIC.EXE functions must take the form:

variable = *expression*

where *variable* is a single alphabetic character E-Z[7] and *expression* is a logic expression using the variables A-Z and #. An expression may use a maximum of four different variables (A-Z) plus the clock value (#). During the expression evaluation, the LOGIC.EXE program will evaluate the expression and store the result into the specified destination variable.

If you enter more than four variables, LOGIC.EXE will complain about your expression. LOGIC.EXE can only evaluation expressions that contain a maximum of four alphabetic characters (not counting the variable to the left of the equals sign). Note that the destination variable may appear within the expression; the following is perfectly legal:

```
F = FA+FB
```

This expression would use the *current* value of F, along with the current values of A and B to compute the new value for F.

Unlike a programming language like "C++", LOGIC.EXE does not evaluate this expression only once and store the result into F. *It will evaluate the expression several times until the value for F stabilizes.* That is, it will evaluate the expression several times until the evaluation produces the same result twice in a row. Certain expressions will produce an *infinite loop* since they will never produce the same value twice in a row. For example, the following function is unstable:

```
F = F'
```

Note that instabilities can cross function boundaries. Consider the following pair of equations:

```
 F = G
G = F'
```

LOGIC.EXE will attempt to execute this set of equations until the values for the variables stop changing. However, the system above will produce an infinite loop.

7. A-D are read-only values that you read from a set of switches. Therefore, you cannot store a value into these variables.

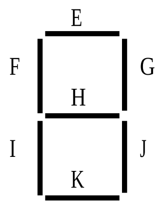
Sometimes a system of logic equations will only produce an infinite loop given certain data values. For example, consider the following of logic equation:

```
F = GF′ + G′F (F = G xor F)
```

If G’s value is one, this system is unstable. If G’s value is zero, this equation is stable. Unstable equations like this one are somewhat harder to discover.

LOGIC.EXE will detect and warn you about logic system instabilities when you attempt to execute the logic functions. Unfortunately, it will not pinpoint the problem for you; it will simply tell you that the problem exists and expect you to fix it.

The A-D, E-K, and W-Z variables are special. A-D are read-only input variables. E-K correspond to the seven segments of a simulated seven-segment display on the *Execute* page:



W-Z correspond to four output LEDs on the *Execute* page. If the variables E-K or W-Z contain a one, then the corresponding LED (or segment) turns red (on). If the variable contains zero, the corresponding LED is off.

The *Create* page contains three important buttons: *Add, Edit*, and *Delete*. When you press the *Add* button LOGIC.EXE opens a dialog box that lets you enter an equation. Type your equation (or edit the default equation) and press the *Okay* button. If there is a problem with the equation you enter, LOGIC.EXE will report the error and make you fix the problem, otherwise, LOGIC.EXE will attempt to add this equation to the system you are building. If a function already exists that has the same destination variable as the equation you’ve just added, LOGIC.EXE will ask you if you really want to replace that function before proceeding with the replacement. Once LOGIC.EXE adds your equation to its list, it also displays the truth table for that equation. You can add up to 22 equations to the system (since there are 22 possible destination variables, E-Z). LOGIC.EXE displays those functions in the list box on the right hand side of the window.

Once you’ve entered two or more logic functions, you can view the truth table for a given logic function by simply clicking on that function with the mouse in the function list box.

If you make a mistake in a logic function you can delete that function by selecting with the mouse and pressing the *delete* button, or you can edit it by selecting it with the mouse and pressing the *edit* button. You can also edit a function by double-clicking on the function in the expression list.

The *Initialize* page displays boxes for each of the 26 possible variables. It lets you view the current values for these 26 variables and change the values of the E-Z variables (remember, A-D are read-only). As a general rule, you will not need to initialize any of the variables, so you can skip this page if you don’t need to initialize any variables.

The *Execute* page contains five buttons of importance: *A-D* and *Pulse.*. The *A-D* toggle switches let you set the input values for the A-D variables. The *Pulse* switch toggles the clock value from zero to one and then back to zero, evaluating the system of logic functions while the clock is in each state.

In addition to the input buttons, there are several outputs on the *Execute* page. First, of course, are the four LEDs (W, X, Y, and Z) as well as the seven-segment display (output variables E-K as noted above). In addition to the LEDs, there is an *Instability* annunciator that turns red if LOGIC.EXE detects an instability in the system. There is also a small panel that displays the current values of all the system variables at the bottom of the window.

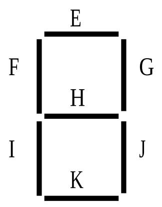
To execute the system of equations simply change one of the input values (A-D) or press the *Pulse* button. LOGIC.EXE will automatically reevaluate the system of equations whenever A-D or # changes.

To familiarize yourself with the LOGIC.EXE program, enter the following equations into the equation editor:

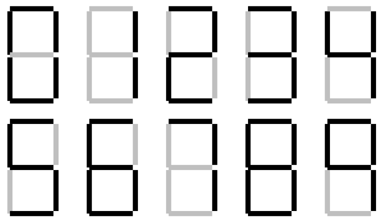
```
W = AB                      A and B
X = A + B                   A or B
Y = A'B + AB'               A xor B
Z = A'                      not A
```

After entering these equations, go to the execute page and enter the four values 00, 01, 10, and 11 for BA. Note the values for W, X, Y, and Z for your lab report.

The LOGIC.EXE program simulates a seven segment display. Variables E-K light the individual segments as follows:

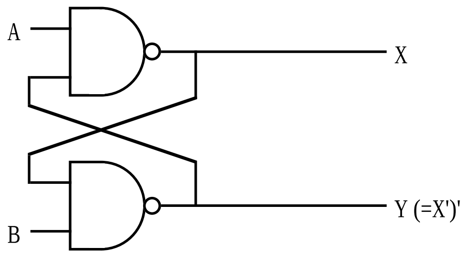


Here are the segments to light for the binary values DCBA = 0000 - 1001:



Enter the seven equations for these segments into LOGIC.EXE and try out each of the patterns (0000 through 1111). **Hint:** use the optimized equations you developed earlier. **Optional, for additional credit:** enter the equations for the 16 hexadecimal values and cycle through those 16 values. Include the results in your lab manual.

A simple sequential circuit. For this exercise you will enter the logic equations for a simple set / reset flip-flop. The circuit diagram is



A Set/Reset Flip-Flop

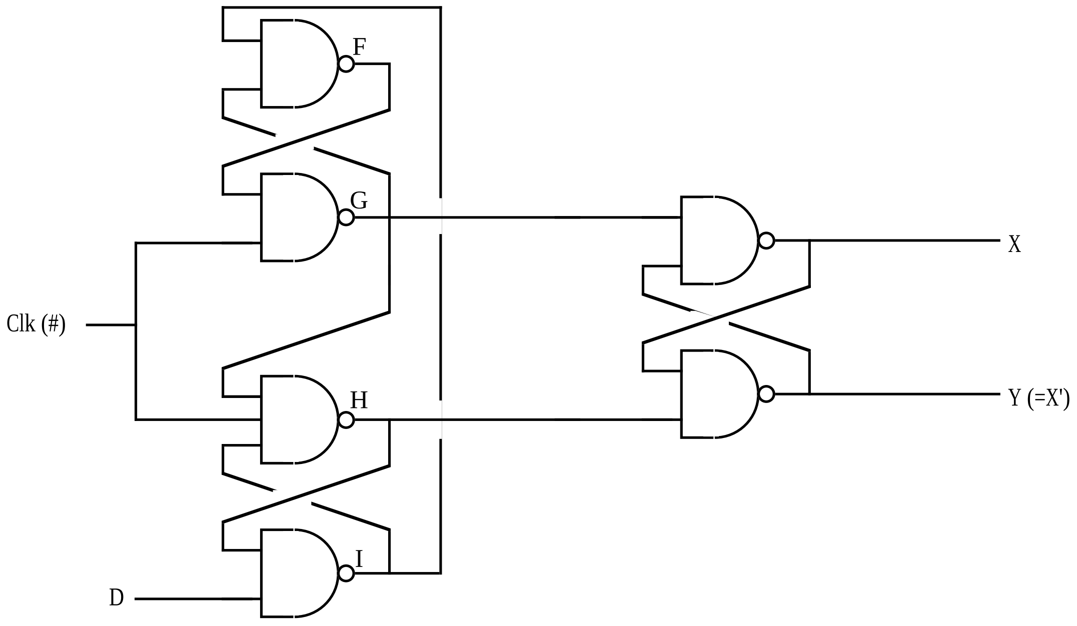
Since there are two outputs, this circuit has two corresponding logic equations. They are

```
X = (AY)'
Y = (BX)'
```

These two equations form a *sequential circuit* since they both use variables that are function outputs. In particular, Y uses the previous value for X and X uses the previous value for Y when computing new values for X and Y.

Enter these two equations into LOGIC.EXE. Set the A and B inputs to one (the normal or *quiescent* state) and run the logic simulation. Try setting the A switch to zero and determine what happens. Press the *Pulse* button several times with A still at zero to see what happens. Then switch A back to one and repeat this process. Now try this experiment again, this time setting B to zero. Finally, try setting *both* A and B to zero and then press the *Pulse* key several times while they are zero. Then set A back to one. Try setting both to zero and then set B back to one. **For your lab report:** provide diagrams for the switch settings and resultant LED values for each time you toggle one of the buttons.

A true D flip-flop only latches the data on the D input during a clock transition from low to high. In this exercise you will simulate a D flip-flop. The circuit diagram for a true D flip-flop is

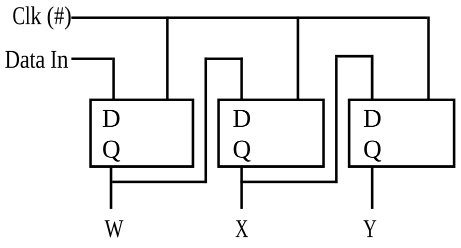


A True D flip-flop

```
F = (IG)’
G = (#F)’
H = (G#I)’
I = (DH)’
X = (GY)’
Y = (HX)’
```

Enter this set of equations and then test your flip-flop by entering different values on the D input switch and pressing the clock pulse button. Explain your results in your lab report.

In this exercise you will build a three-bit shift register using the logic equations for a true D flip-flop. To construct a shift register, you connect the outputs from each flip-flop to the input of the next flip-flop. The data input line provides the input to the first flip-flop, the last output line is the “carry out” of the circuit. Using a simple rectangle to represent a flip-flop and ignoring the Q’ output (since we don’t use it), the schematic for a four-bit shift register looks something like the following:



A Three-bit Shift Register Built from D Flip-flops

In the previous exercise you used six boolean expressions to define the D flip-flop. Therefore, we will need a total of 18 boolean expressions to implement a three-bit flip-flop. These expressions are

```
Flip-Flop #1:

          W = (GR)’
          F = (IG)’
          G = (F#)’
          H = (G#I)’
          I = (DH)’
          R = (HW)’

Flip-Flop #2:

          X = (KS)’
          J = (MK)’
          K = (J#)’
          L = (K#M)’
          M = (WL)’
          S = (LX)’

Flip-Flop #3:

          Y = (OT)’
          N = (QO)’
          O = (N#)’
          P = (O#Q)’
          Q = (XP)’
          T = (PY)’
```

Enter these equations into LOGIC.EXE. Initialize W, X, and Y to zero. Set D to one and press the *Pulse* button once to shift a one into W. Now set D to zero and press the pulse button several times to shift that single bit through each of the output bits. **For your lab report**: try shifting several bit patterns through the shift register. Describe the step-by-step operation in your lab report.

For additional credit: Describe how to create a *recirculating shift register.* One whose output from bit four feeds back into bit zero. What would be the logic equations for such a shift register? How could you initialize it (since you cannot use the D input) when using LOGIC.EXE?

Post-lab, for additional credit: Design a two-bit full adder that computes the sum of BA and DC and stores the binary result to the WXY LEDs. Include the equations and sample results in your lab report.

2.10 Programming Projects

You may write these programs in any HLL your instructor allows (typically C, C++, or some version of Borland Pascal or Delphi). You may use the generic logic functions appearing in this chapter if you so desire.

1) Write a program that reads four values from the user, I, J, K, and L, and plugs these values into a truth table with B’A’ = I, B’A = J, BA’ = K, and BA = L. Ensure that these input values are only zero or one. Then input a series of pairs of zeros or ones from the user and

plug them into the truth table. Display the result for each computation. Note: do *not* use the generic logic function for this program.

2) Write a program that, given a 4-bit logic function number, displays the truth table for that function of two variables.

3) Write a program that, given an 8-bit logic function number, displays the truth table for that function of three variables.

4) Write a program that, given a 16-bit logic function number, displays the truth table for that function of four variables.

5) Write a program that, given a 16-bit logic function number, displays the canonical equation for that function (hint: build the truth table).

2.11 Summary

Boolean algebra provides the foundation for both computer hardware and software. A cursory understanding of this mathematical system can help you better appreciate the connection between software and hardware.

Boolean algebra is a mathematical system with its own set of rules (postulates), theorems, and values. In many respects, boolean algebra is similar to the real-arithmetic algebra you studied in high school. In most respects, however, boolean algebra is actually *easier* to learn than real arithmetic algebra. This chapter begins by discussing features of any algebraic system including operators, closure, commutativity, associativity, distribution, identity, and inverse. Then it presents some important postulates and theorems from boolean algebra and discusses the *principle of duality* that lets you easily prove additional theorems in boolean algebra. For the details, see

- “Boolean Algebra” on page 43

The *Truth Table* is a convenient way to visually represent a boolean function or expression. Every boolean function (or expression) has a corresponding truth table that provides all possible results for any combination of input values. This chapter presents several different ways to construct boolean truth tables.

Although there are an infinite number of boolean functions you can create given *n* input values, it turns out that there are a finite number of unique functions possible for a given number of inputs. In particular, there are $2\text{\^{}}2^n$ unique boolean functions of *n* inputs. For example, there are 16 functions of two variables ($2\text{\^{}}2^2 = 16$).

Since there are so few boolean functions with only two inputs, it is easy to assign different names to each of these functions (e.g., AND, OR, NAND, etc.). For functions of three or more variables, the number of functions is too large to give each function its own name. Therefore, we’ll assign a number to these functions based on the bits appearing in the function’s truth table. For the details, see

- “Boolean Functions and Truth Tables” on page 45

We can manipulate boolean functions and expression algebraically. This allows us to prove new theorems in boolean algebra, simplify expressions, convert expressions to canonical form, or show that two expressions are equivalent. To see some examples of algebraic manipulation of boolean expressions, check out

- “Algebraic Manipulation of Boolean Expressions” on page 48

Since there are an infinite variety of possible boolean functions, yet a finite number of unique boolean functions (for a fixed number of inputs), clearly there are an infinite number of different functions that compute the same results. To avoid confusion, logic designers usually specify a boolean function using a *canonical form*. If two canonical equations are different, then they represent different boolean functions. This book describes two different canonical forms: the sum of minterms form and the product of maxterms form. To learn about these canonical forms, how to convert an arbitrary boolean equation to canonical form, and how to convert between the two canonical forms, see

- “Canonical Forms” on page 49

Although the canonical forms provide a unique representation for a given boolean function, expressions appearing in canonical form are rarely *optimal*. That is, canonical expressions often use more literals and operators than other, equivalent, expressions. When designing an electronic circuit or a section of software involving boolean expressions, most engineers prefer to use an optimized circuit or program since optimized versions are less expensive and, probably, faster. Therefore, knowing how to create an optimized form of a boolean expression is very important. This text discusses this subject in

- “Simplification of Boolean Functions” on page 52

Boolean algebra isn’t a system designed by some crazy mathematician that has little importance in the real world. Boolean algebra is the basis for digital logic that is, in turn, the basis for computer design. Furthermore, there is a one-to-one correspondence between digital hardware and computer software. Anything you can build in hardware you can construct with software, and vice versa. This text describes how to implement addition, decoders, memory, shift registers, and counters using these boolean functions. Likewise, this text describes how to improve the efficiency of software (e.g., a Pascal program) by applying the rules and theorems of boolean algebra. For all the details, see

- “What Does This Have To Do With Computers, Anyway?” on page 59
- “Correspondence Between Electronic Circuits and Boolean Functions” on page 59
- “Combinatorial Circuits” on page 60
- “Sequential and Clocked Logic” on page 62
- “Okay, What Does It Have To Do With Programming, Then?” on page 64

2.12 Questions

1. What is the identity element with respect to

   a) AND b) OR c) XOR d) NOT e) NAND f) NOR

2. Provide truth tables for the following functions of two input variables:

   a) AND b) OR c) XOR d) NAND e) NOR

   f) Equivalence g) A < B h) A > B i) A implies B

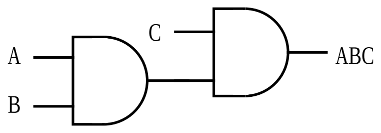
3. Provide the truth tables for the following functions of three input variables:

   a) ABC (and) b) A+B+C (OR) c) (ABC)' (NAND)d) (A+B+C)' (NOR)

   e) Equivalence (ABC) + (A'B'C') f) XOR (ABC + A'B'C')'

4. Provide schematics (electrical circuit diagrams) showing how to implement each of the functions in question three using only two-input gates and inverters. E.g.,

   A) ABC =



5. Provide implementations of an AND gate, OR gate, and inverter gate using one or more NOR gates.

6. What is the principle of duality? What does it do for us?

7. Build a single truth table that provides the outputs for the following three boolean functions of three variables:

   F_x= A + BC

   F_y - AB + C'B

   F_z = A'B'C' + ABC + C'B'A

8. Provide the function numbers for the three functions in question seven above.

9. How many possible (unique) boolean functions are there if the function has

   a) one input b) two inputs c) three inputs d) four inputs e) five inputs

10. Simplify the following boolean functions using algebraic transformations. Show your work.

    a) F = AB + AB' b) F = ABC + BC' + AC + ABC'

    c) F = A'B'C'D' + A'B'C'D + A'B'CD + A'B'CD'

    d) F = A'BC + ABC' + A'BC' + AB'C' + ABC + AB'C

11. Simplify the boolean functions in question 10 using the mapping method.

12. Provide the logic equations in canonical form for the boolean functions S_0..S_6 for the seven segment display (see "Combinatorial Circuits" on page 60).

13. Provide the truth tables for each of the functions in question 12

14. Minimize each of the functions in question 12 using the map method.

15. The logic equation for a half-adder (in canonical form) is

    Sum = AB' + A'B Carry = AB

    a) Provide an electronic circuit diagram for a half-adder using AND, OR, and Inverter gates

    b) Provide the circuit using only NAND gates

16. The canonical equations for a full adder take the form:

    Sum = A’B’C + A’BC’ + AB’C’ + ABC

    Carry = ABC + ABC’ + AB’C + A’BC

    a) Provide the schematic for these circuits using AND, OR, and inverter gates.

    b) Optimize these equations using the map method.

    c) Provide the electronic circuit for the optimized version (using AND, OR, and inverter gates).

17. Assume you have a D flip-flop (use this definition in this text) whose outputs currently are Q=1 and Q’=0. Describe, in minute detail, exactly what happens when the clock line goes

    a) from low to high with D=0

    b) from high to low with D=0

18. Rewrite the following Pascal statements to make them more efficient:

    a) if (x or (not x and y)) then write(‘1’);

    b) while(not x and not y) do somefunc(x,y);

    c) if not((x <> y) and (a = b)) then Something;

19. Provide canonical forms (sum of minterms) for each of the following:

    a) F(A,B,C) = A’BC + AB + BC   b) F(A,B,C,D) = A + B + CD’ + D

    c) F(A,B,C) = A’B + B’A   d) F(A,B,C,D) = A + BD’

    e) F(A,B,C,D) = A’B’C’D + AB’C’D’ + CD + A’BCD’

20. Convert the sum of minterms forms in question 19 to the product of maxterms forms.