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. That subject alone is often the subject of an entire textbook. This chapter concentrates on those subjects that support other chapters in this text.

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, and combinatorial and sequential circuits.

This 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.

3.1 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 ¡ B) ¡ C = A ¡ (B ¡ C)

¥

¥ for all boolean values A, B, and C.

¥ ¥*Distribution*. Two binary operators ¡ and % are distributive if

¥ A ¡(B % C) = (A ¡ B) % (A ¡ 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 ° I = A
for all boolean values A.

¥ ¥*Inverse*. A boolean value I is said to be the *inverse element* with respect to some binary operator ¡ if A ° I = B and B≠A (i.e., B is the opposite value of A in a boolean system)
for all boolean values A and B.

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 includes:

Th1: A + A = A

Th2: A ¥ A = A

Th3: A + 0 = A

Th4: A ¥ 1 = A

Th5: A ¥ 0 = 0

Th6: A + 1 = 1

Th7: (A + B) = A ¥ B

Th8: (A ¥ B) = A + B

Th9: A + A¥B = A

Th10: A ¥(A + B) = A

Th11: A + A B = A+B

Th12: A ¥ (A + B) = A B

Th13: AB + AB = A

Th14: (A +B) ¥ (A + B) = A

Th15: A + A = 1

Th16: A ¥ A = 0

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

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

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

3.2 Boolean Functions and Truth Tables

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

$$F_0 = AB+C$$

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

Another way to represent a boolean function is via a *truth table*. A previous chapter (see Logical Operations on Bits on page 65) used truth tables to represent the AND and OR functions. Those truth tables took the forms:

Table 13: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 14: 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:

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

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.

The following table 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.

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

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)[1] unique boolean functions of those *n* input values. For two input variables, $2^{**}(2^2) = 2^4$ or 16 different functions. With three input variables 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:

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 = AB’ (A, not B). Also equivalent to A>B or B < A.
3	NOT B. Ignores A and returns B’.
4	Inhibition = BA’ (B, not A). Also equivalent to B>A or A<B.
5	NOT A. Returns A’ and ignores B
6	Exclusive-or (XOR) = A ⊕ B. Also equivalent to A≠B.
7	Logical NAND (NOT (A AND B)) = (A•B)’
8	Logical AND = A•B. Returns A AND B.

1. In this context, the operator ** means exponentiation.

Function #	Description
9	Equivalence = (A = B). Also known as exclusive-NOR (not exclusive-or).
10	Copy A. Returns the value of A and ignores B's value.
11	Implication, B implies A, or A + B'. (if B then A). Also equivalent to B >= A.
12	Copy B. Returns the value of B and ignores A's value.
13	Implication, A implies B, or 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. 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  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.

3.3 Algebraic Manipulation of Boolean Expressions

You can transform one boolean expression into an equivalent expression by applying the postulates and 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 + b                  By Th11


(a′b + a′b′ + b′)‘      =  ( a′(b+b′) + b′)′      By P4
                        =  (a′•1 + b′)′           By P5
                        =  (a′ + b′)              By Th4
                        =  ( (ab)′ )′             By Th8
                        =  ab                     By definition of not


b(a+c) + ab′ + bc′ + c= ba + bc + ab′ + bc′ + cBy P4
                        =  a(b+b′) + b(c + c′) + cBy P4
                        =  a•1 + b•1 + cBy P5
                        =  a + b + cBy 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. We can use algebraic manipulation to produce canonical forms even though the canonical forms are rarely optimal.

3.4 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’, A’BC, 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:

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′ By P4
                         =  BC•1 +B′C•1 + ABC′    By Th15
                         =  C(B+B′) + ABC′        By P4
                         =  C + ABC′              By Th15 & Th4
                         =  C + AB                By Th11
```

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 posi - tion (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:

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:

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:

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:

$$G = (A+B+C) \cdot (A'+B+C) \cdot (A+B'+C).$$

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

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

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

$G= (A + B + C) ¥ (A + B + C) ¥ (A + B + C)$

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

$G = A B C + AB C + A BC$

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

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

Table 15: 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 vari-

ables 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.

3.5 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[2].

The first step in using the map method is to build a two-dimensional truth table for the function (see Figure 3.1)

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

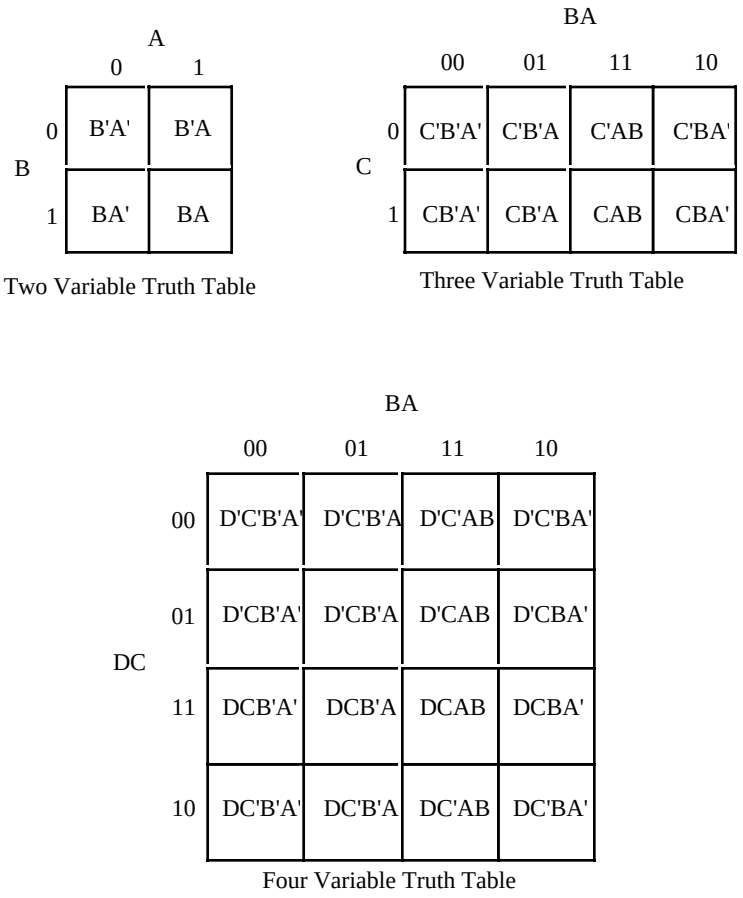
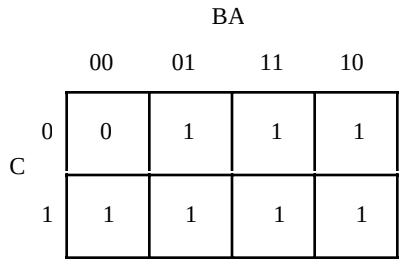


Figure 3.1 Two, Three, and Four Dimensional Truth Tables

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.

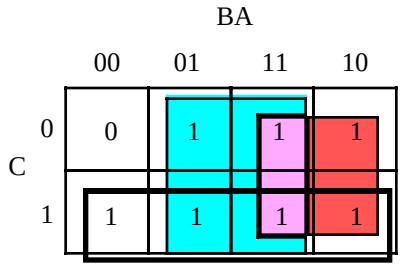
Assuming your boolean function is in canonical form (sum of minterms), insert ones for each of the truth map entries corresponding to a minterm in the function. Place zeros everywhere else. For example, consider the function of three variables F=C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA. Figure 3.2 shows the truth map for this function.



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

Figure 3.2 A Simple 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 3.3 there are three such rectangles (see Figure 3.3)



Three possible rectangles whose lengths and widths are powers of two.

Figure 3.3 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 blue square above. This rectangle includes C, C', B, B' and A. Therefore, it represents the single term A. Likewise, the red square 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 the remaining 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 3.4 shows the truth map for this function.

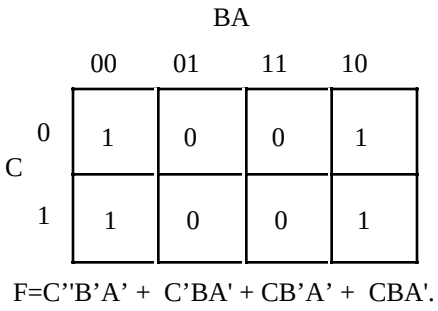


Figure 3.4 Truth Map for F=C'B'A' + C'BA' + CB'A' + CBA'

At first glance, you would think that there are two possible rectangles here as Figure 3.5 shows.

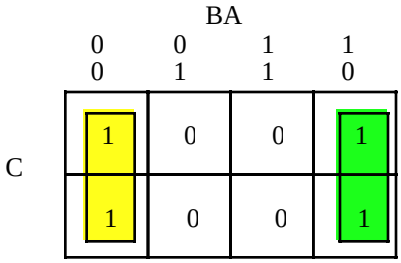


Figure 3.5 First Attempt at Surrounding Rectangles Formed by Ones

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 3.6 shows.

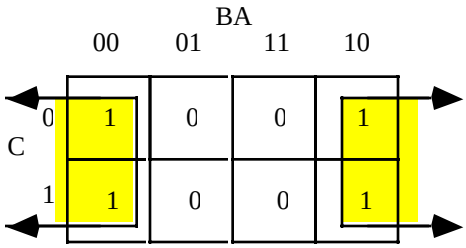


Figure 3.6 Correct Rectangle for the Function

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 (that is, the function number is 2^{n-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' 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 3.7.

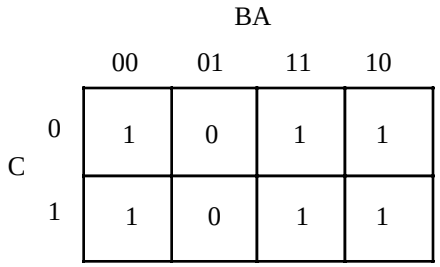


Figure 3.7 Truth Map for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

The initial temptation is to create one of the sets of rectangles found in Figure 3.8. However, the correct mapping appears in Figure 3.9

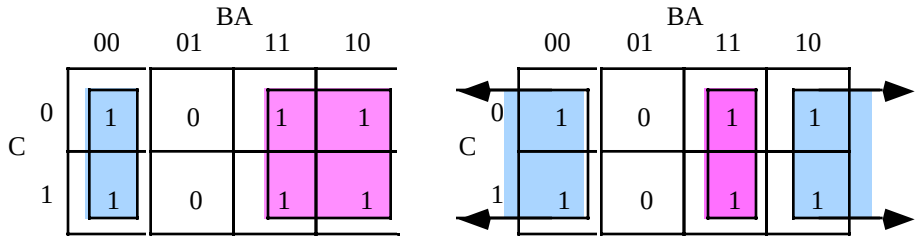


Figure 3.8 Obvious Choices for Rectangles

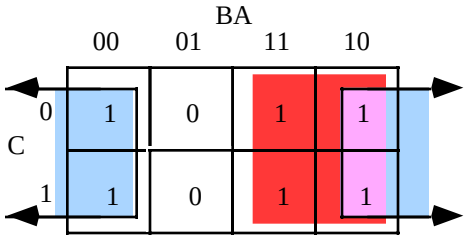


Figure 3.9 Correct Set of Rectangles for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

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 better optimized 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 (assuming we're working with functions of three variables).
- 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 3.10 shows some possible places rectangles can hide.

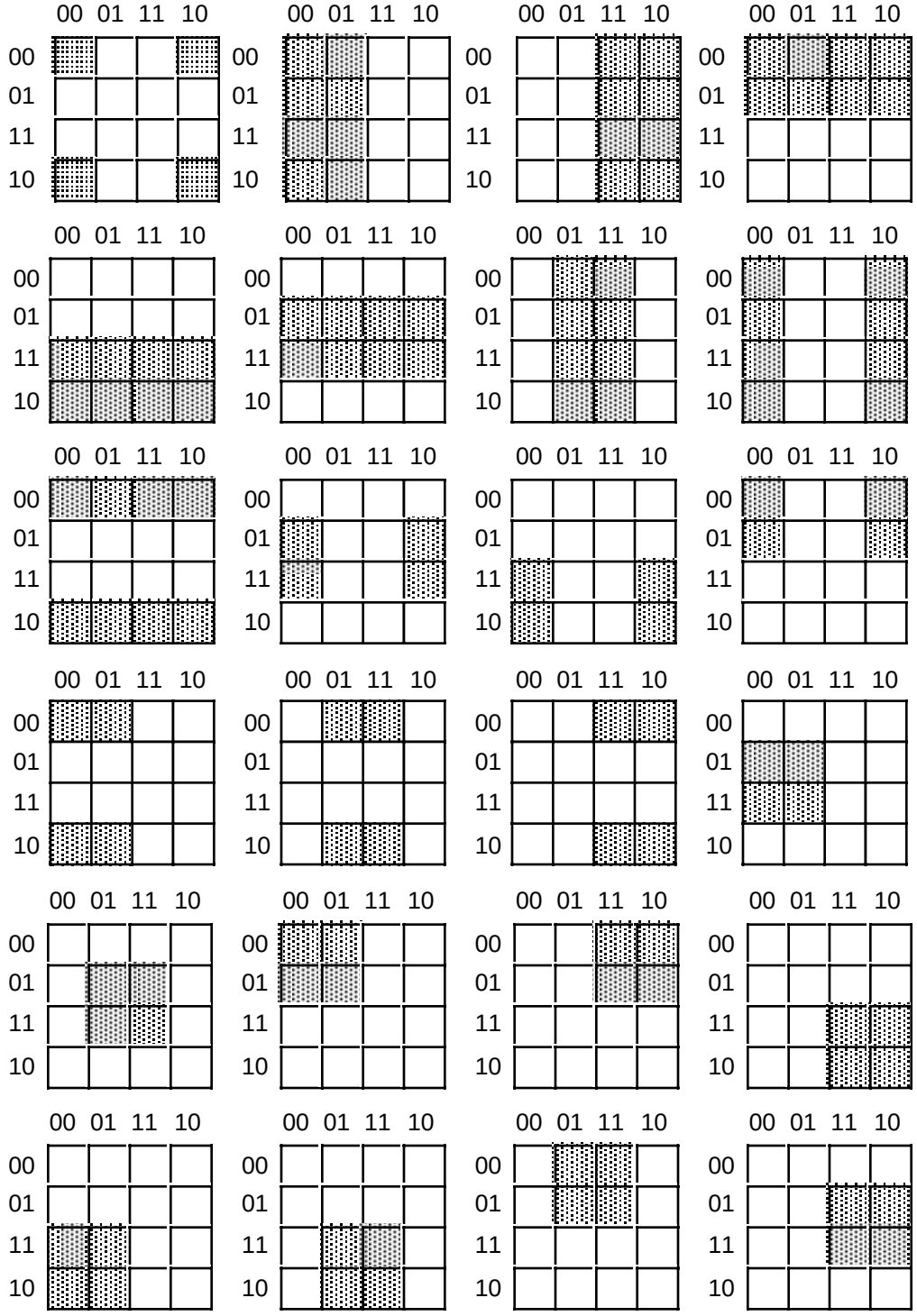


Figure 3.10 Partial Pattern List for 4x4 Truth Map

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 3.11.

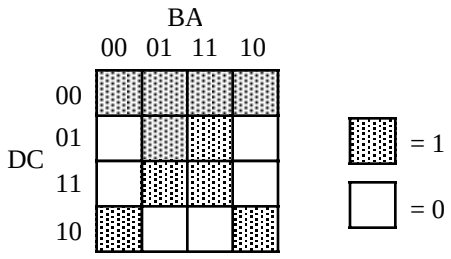


Figure 3.11 Truth Map for F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA

Here are two possible sets of maximal rectangles for this function, each producing three terms (see Figure 3.12). Both functions are equivalent; both are as optimal as you can get[3]. Either will suffice for our purposes.

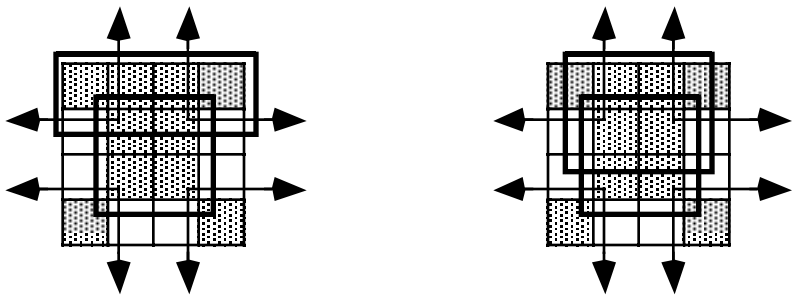


Figure 3.12 Two Combinations of Surrounded Values Yielding Three Terms

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'.

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

The second rectangle, common to both maps in Figure 3.12, 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 3.12 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 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 3.12 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.

3.6 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 than it first appears. 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.

3.6.1 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[4], 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 3.13).

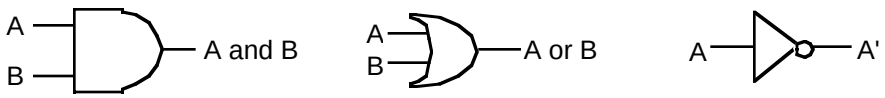


Figure 3.13 AND, OR, and Inverter (NOT) Gates

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 3.14.

4. We know this is true because these are the only operators that appear within canonical forms.

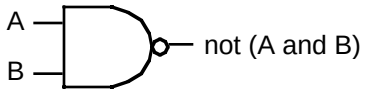


Figure 3.14 The NAND Gate

To prove that we can construct any boolean function using only NAND gates, we need only show how to build an inverter (NOT), an AND gate, and an 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 3.15).

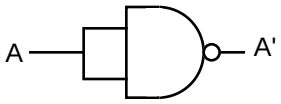


Figure 3.15 Inverter Built from a NAND Gate

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 Figure 3.16). Of course, this takes two NAND gates to construct a single AND gate, but no one said that circuits constructed only with NAND gates would be optimal, only that it is possible.

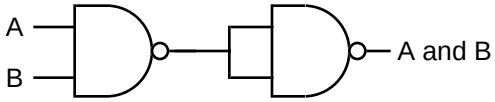


Figure 3.16 Constructing an AND Gate From Two NAND Gates

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 3.17.

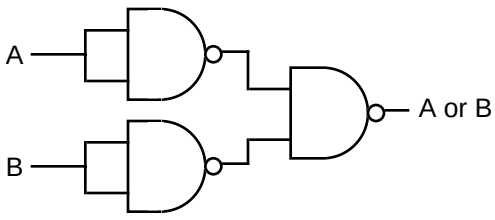


Figure 3.17 Constructing an OR Gate from NAND Gates

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[5]. 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.

3.6.2 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 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 3.18). So, as this example clearly illustrates, we can use logic functions to implement arithmetic and boolean operations.

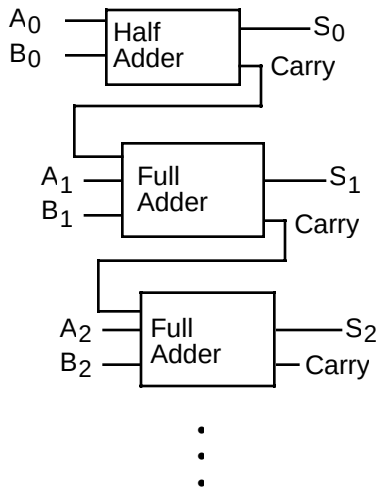


Figure 3.18 Building an N-Bit Adder Using Half and Full Adders

Another common combinatorial circuit is the *seven-segment decoder*. This is a combinatorial circuit that accepts four inputs and determines which of the 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 associ-

5. NOR is NOT (A OR B).

ated with the display (segment zero through segment six). See Figure 3.19 for the segment assignments. Figure 3.20 shows the segment assignments for each of the ten decimal values.

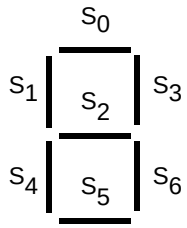


Figure 3.19 Seven Segment Display

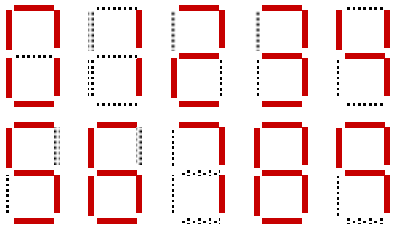


Figure 3.20 Seven Segment Values for “0” Through “9”

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 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_0, 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.

Decoder circuits are among the more important circuits in computer system design. They provide the ability to recognize (or ‘decode’) a string of bits. One very common use for a decoder is memory expansion. For example, suppose a system designer wishes to install four (identical) 256 MByte memory modules in a system to bring the total to one gigabyte of RAM. These 256 MByte memory modules have 28 address lines ($A_0..A_{27}$) assuming each memory module is eight bits wide (2^{28} x 8 bits is 256 MBytes)[6]. Unfortunately, if the system designer hooked up those four memory modules to the CPU’s address bus they would all respond to the same addresses on the bus. Pandemonium would result. To correct this problem, we need to select each memory module when a different set of addresses appear on the address bus. By adding a chip enable line to each of the memory modules and using a two-input, four-output decoder circuit, we can easily do this. See Figure 3.21 for the details.

6. Actually, most memory modules are wider than eight bits, so a real 256 MByte memory module will have fewer than 28 address lines, but we will ignore this technicality in this example.

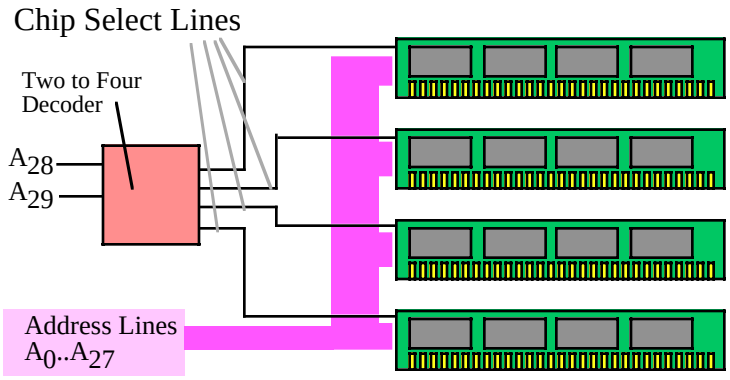


Figure 3.21 Adding Four 256 MByte Memory Modules to a System

The two-line to four-line decoder circuit in Figure 3.21 actually incorporates four different logic functions, one function for each of the outputs. Assume the inputs are A and B (A=A_{28} and B=A_{29}) then the four output functions have the following (simple) equations:

Q_0 = A′ B′
Q_1 = A B′
Q_2 = A′ B
Q_3 = A B

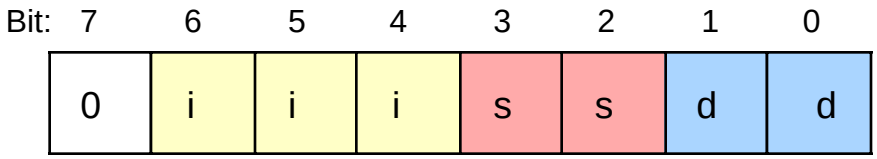
Following standard electronic circuit notation, these equations use “Q” to denote an output (electronic designers use “Q” for output rather than “O” because “Q” looks somewhat like an “O” and is more easily differentiated from zero). Also note that most circuit designers use *active low logic* for decoders and chip enables. This means that they enable a circuit with a low input value (zero) and disable the circuit with a high input value (one). Likewise, the output lines of a decoder chip are normally high and go low when the inputs select a given output line. This means that the equations above really need to be inverted for real-world examples. We’ll ignore this issue here and use positive (or active high) logic[7].

Another big use for decoding circuits is to decode a byte in memory that represents a machine instruction in order to activate the corresponding circuitry to perform whatever tasks the instruction requires. We’ll cover this subject in much greater depth in a later chapter, but a simple example at this point will provide another solid example for using decoders.

Most modern (Von Neumann) computer systems represent machine instructions via values in memory. To execute an instruction the CPU fetches a value from memory, decodes that value, and the does the appropriate activity the instruction specifies. Obviously, the CPU uses decoding circuitry to decode the instruction. To see how this is done, let’s create a very simple CPU with a very simple instruction set. Figure 3.22 provides the instruction format (that is, it specifies all the numeric codes) for our simple CPU.

7. Electronic circuits often use active low logic because the circuits that employ them typically require fewer transistors to implement.

Instruction (opcode) Format:



iii

000 = MOV
001 = ADD
010 = SUB
011 = MUL
100 = DIV
101 = AND
110 = OR
111 = XOR

ss & dd

00 = EAX
01 = EBX
10 = ECX
11 = EDX

Figure 3.22 Instruction (opcode) Format for a Very Simple CPU

To determine the eight-bit operation code (opcode) for a given instruction, the first thing you do is choose the instruction you want to encode. Let's pick "MOV(EAX, EBX);" as our simple example. To convert this instruction to its numeric equivalent we must first look up the value for MOV in the iii table above; the corresponding value is 000. Therefore, we must substitute 000 for iii in the opcode byte.

Second, we consider our source operand. The source operand is EAX, whose encoding in the source operand table (ss & dd) is 00. Therefore, we substitute 00 for ss in the instruction opcode.

Next, we need to convert the destination operand to its numeric equivalent. Once again, we look up the value for this operand in the ss & dd table. The destination operand is EBX and it's value is 01. So we substitute 01 for dd in our opcode byte. Assembling these three fields into the opcode byte (a packed data type), we obtain the following bit value: %00000001. Therefore, the numeric value $1 is the value for the "MOV(EAX, EBX);" instruction (see Figure 3.23).

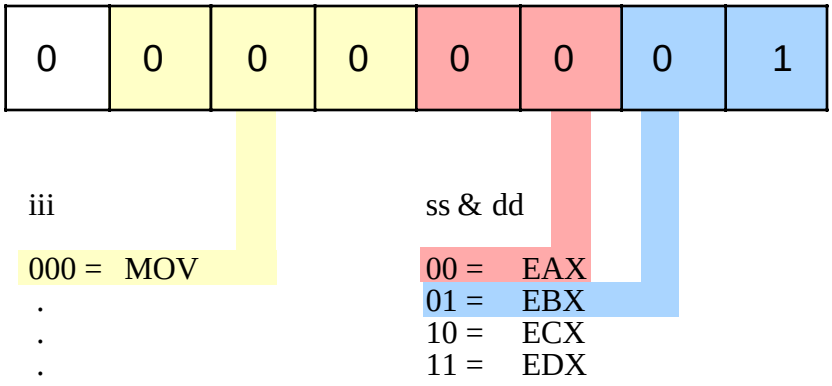
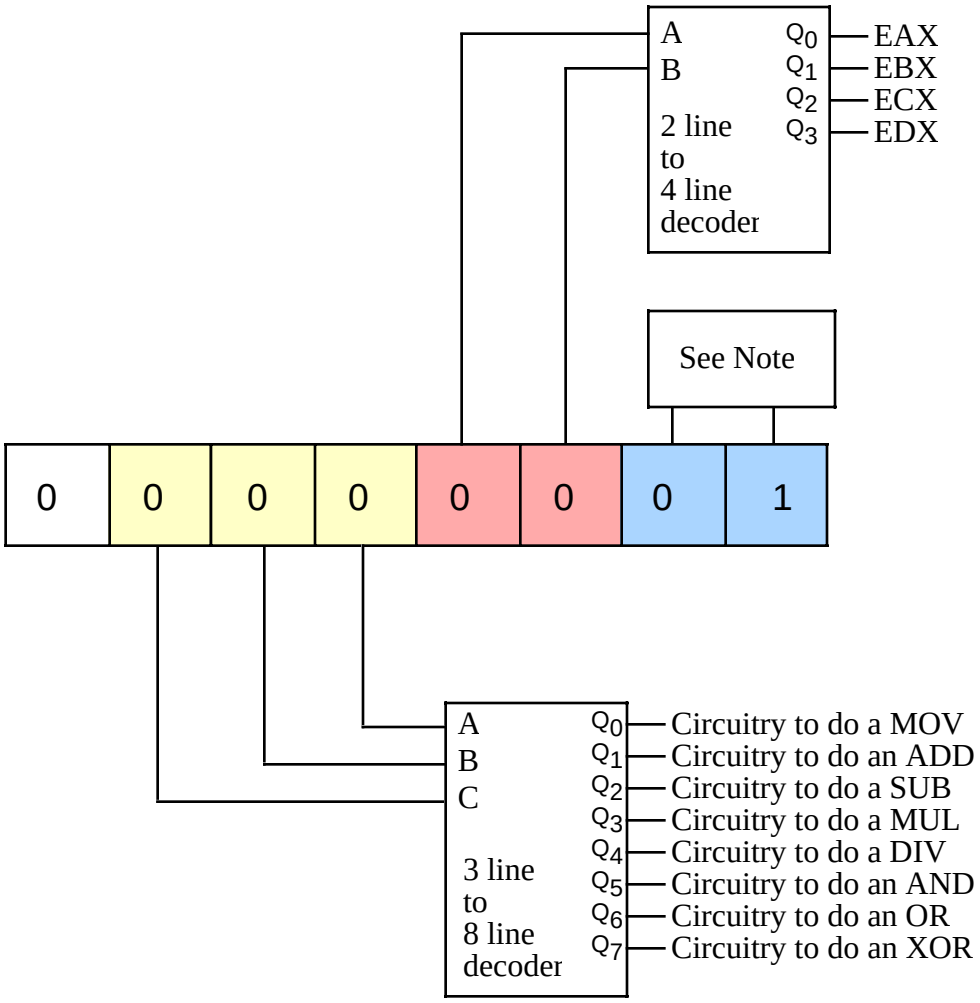


Figure 3.23 Encoding the MOV(EAX, EBX); Instruction

As another example, consider the "AND(EDX, ECX);" instruction. For this instruction the iii field is %101, the ss field is %11, and the dd field is %10. This yields the opcode %01011110 or $5E. You may easily create other opcodes for our simple instruction set using this same technique.

> **Warning**: please do not come to the conclusion that these encodings apply to the 80x86 instruction set. The encodings in this examples are highly simplified in order to demonstrate instruction decoding. They do not correspond to any real-life CPU, and they especially don't apply to the x86 family.

In these past few examples we were actually *encoding* the instructions. Of course, the real purpose of this exercise is to discover how the CPU can use a decoder circuit to decode these instructions and execute them at run time. A typical set of decoder circuits for this might look like that in Figure 3.24:



Note: the circuitry attached to the destination register bits is identical to the circuitry for the source register bits.

Figure 3.24 Decoding Simple Machine Instructions

Notice how this circuit uses three separate decoders to decode the individual fields of the opcode. This is much less complex than creating a seven-line to 128-line decoder to decode each individual opcode. Of course, all that the circuit above will do is tell you which instruction and what operands a given opcode specifies. To actually execute this instruction you must supply additional circuitry to select the source and destination operands from an array of registers and act accordingly upon those operands. Such circuitry is beyond the scope of this chapter, so we'll save the juicy details for later.

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.

3.6.3 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[8]. 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 3.25.

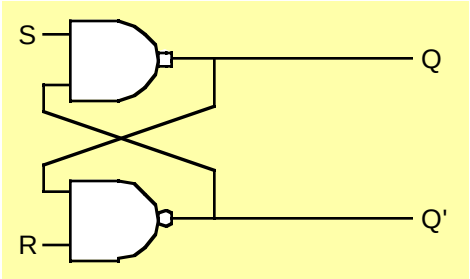


Figure 3.25 Set/Reset Flip Flop Constructed from NAND Gates

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 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 3.26.

8. 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.

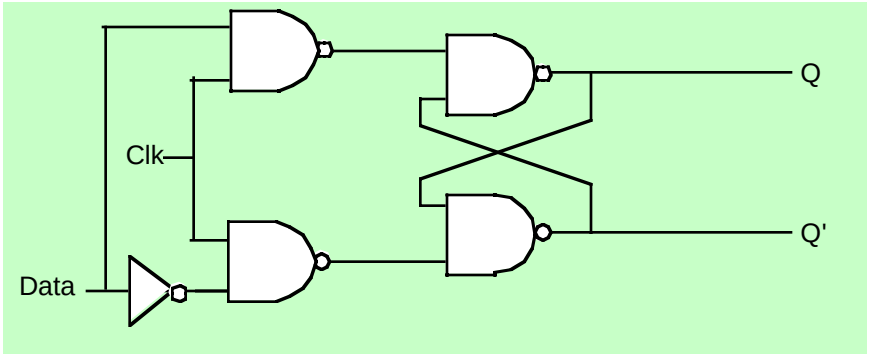


Figure 3.26 Implementing a D flip-flop with NAND Gates

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 topic section 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 3.27 shows how to build an eight-bit register from a set of D flip-flops.

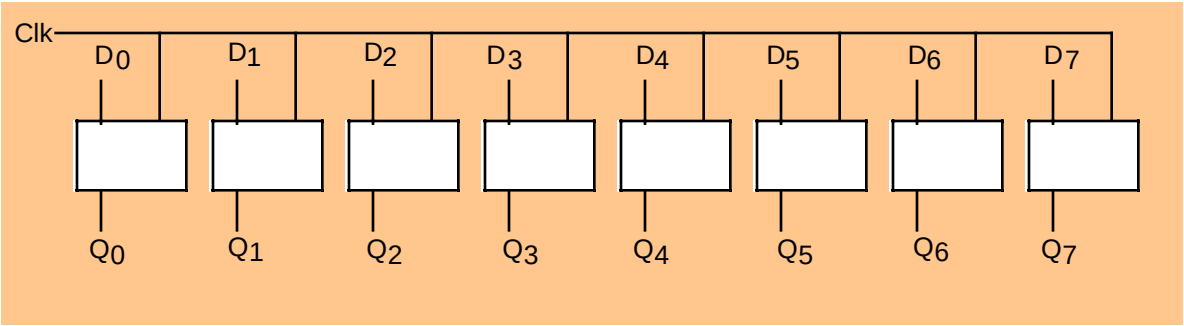


Figure 3.27 An Eight-bit Register Implemented with Eight D Flip-flops

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 3.28.

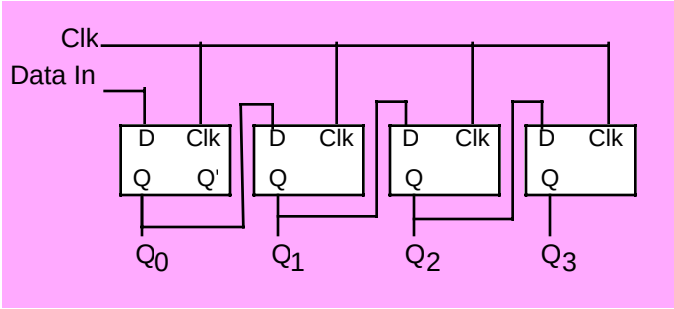


Figure 3.28 A Four-bit Shift Register Built from D Flip-flops

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 3.29 implements a four bit counter using D flip-flops.

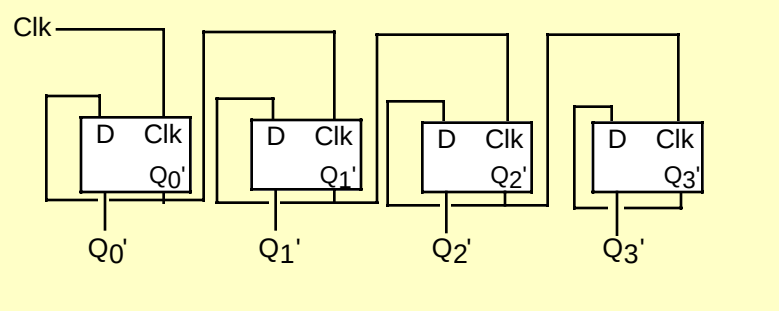


Figure 3.29 Four-bit Counter Built from D Flip-flops

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits beyond these. For example, you can build a simple state machine known as a sequencer by combining a counter and a decoder as shown in Figure 3.30. For each cycle of the clock this sequencer activates one of its output lines. Those lines, in turn, may control other circuitry. By "firing" these circuits on each of the 16 output lines of the decoder, we can control the order in which these 16 different circuits accomplish their tasks. This is a fundamental need in a CPU since we often need to control the sequence of various operations (for example, it wouldn't be a good thing if the "ADD(EAX, EBX);" instruction stored the result into EBX before fetching the source operand from EAX (or EBX). A simple sequencer such as this one can tell the CPU when to fetch the first operand, when to fetch the second operand, when to add them together, and when to store the result away. But we're getting a little ahead of ourselves, we'll discuss this in greater detail in a later chapter.

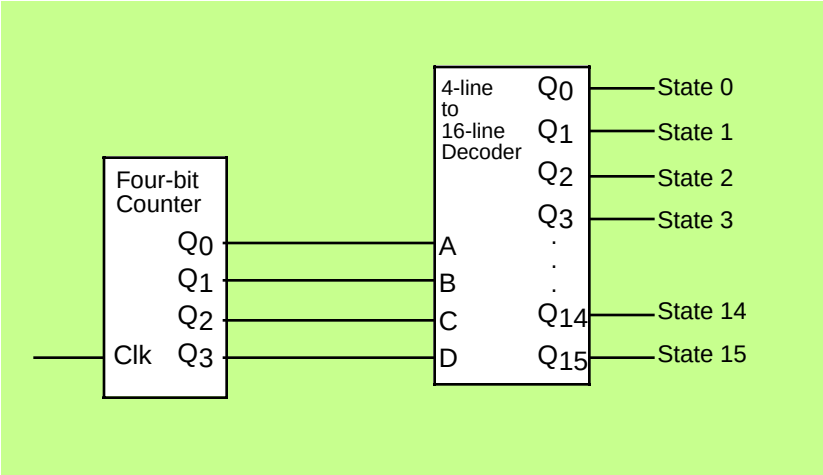


Figure 3.30 A Simple 16-State Sequencer

3.7 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 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 $5 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;
```

So as you can see, understanding a little boolean algebra can actually help you write better software.

3.8 Putting It All Together

A good understanding of boolean algebra and digital design is absolutely necessary for anyone who wants to understand the internal operation of a CPU. As an added bonus, programmers who understand digital design can write better assembly language (and high level language) programs. This chapter provides a basic introduction to boolean algebra and digital circuit design. Although a detailed knowledge of this material isn't necessary if you simply want to write assembly language programs, this knowledge will help explain why Intel chose to implement instructions in certain ways; questions that will undoubtedly arise as we begin to look at the low-level implementation of the CPU.

This chapter is not, by any means, a complete treatment of this subject. If you're interested in learning more about boolean algebra and digital circuit design, there are dozens and dozens of texts on this subject available. Since this is a text on assembly language programming, we cannot afford to spend additional time on this subject; please see one of these other texts for more information.