THE LANGUAGE OF LOGIC

Digital circuits, no matter how complex, are composed of a small group of identical building blocks. These blocks are either basic gates or special circuits such as Schmitt triggers, special memory cells, and other structures for which gates are less suitable. The vast majority of the functional building blocks are gates or combinations of gates. A flip-flop, for example, can be considered as a functional block, but it too is composed of standard gates interconnected within the package.

A counter is a more complex functional unit than a flip-flop and is composed of flip-flops and control gates connected inside the package to perform a particular counting function. The flip-flops are made up of combinations of basic gates.

At a much higher level of organization, the microprocessor is the central processing unit of a computer in a package using thousands of gates, flip-flops, and memory cells. Even the memory cells are modified versions of basic gates.

In this chapter we will examine the functional (logical) properties of basic gates. In the next chapter we will examine the electronic properties of the most important gate circuits. In subsequent chapters we will see how these basic gates can be integrated into larger systems to perform all of the computing, counting, and control functions required by computers—as well as by smaller-scale digital systems.

In most electronics systems the schematic diagram is our most valuable symbolic tool. We also use block diagrams to explain systems behavior. In logic systems the schematic diagram is often an unsatisfactory form of communication because of the large number of individual circuits involved and because modem integrated circuits are fairly complex. Schematic diagrams of digital systems would be difficult to follow and to draw simply because of the vast number of components involved. There are only a half dozen, or so basic circuits for which we normally draw schematics. In this chapter we will be concerned with logic gates, but later we will encounter some building blocks that are not made up of basic gates alone.

BOOLEAN ALGEBRA

Boolean algebra is a two-value symbolic logic system. This symbolic tool allows us to design complex logic systems with the certainty that they will carry out their function exactly as intended. Conversely, we can use Boolean algebra to determine exactly what function an existing logic circuit was intended to perform. A very important aspect of the Boolean system is that it formalizes the difficult problem of getting the job done with an absolute minimum of hardware.
In applied Boolean, the symbols and equations of Boolean algebra are integrated with conventional electronic symbols. Digital waveforms called timing diagrams and digital block diagrams called logic diagrams are liberally mixed with their symbolic (Boolean) counterparts.

**BINARY ARITHMETIC**

Ordinary algebra is a generalization of our human system of arithmetic. Machine arithmetic is accomplished in a two-value (binary) number system, but Boolean algebra is not a generalization of the binary number system. The symbols 0, 1, +, and • are used in both systems but with totally different meanings in each system. Table 1-1 illustrates some important differences between them. There are more differences than similarities, and binary numbers and Boolean algebra must be treated as entirely different entities.

The use in one machine of two different systems having the same symbols but different meanings sounds confusing. In practice, both counting in binary and arithmetic in binary are performed by Boolean logic circuits. The actual circuits are strictly Boolean logic circuits, although the input to the circuit or the output from it may be binary. Binary numbers and arithmetic may be thought of as being external to the electronic logic circuitry. Because of the above-mentioned distinction, there is little real confusion generated.

Although we will not get into binary arithmetic until a later chapter, we need to understand right away how the basic number system works. We will use binary numbers to enumerate things, to count entries in Boolean truth tables, and so on.

**The Binary Number System**

The binary system contains only two symbols, 0 and 1, but any number that can be written in the decimal system can also be written in the binary system. The binary system is structurally similar to our decimal system.

In the decimal system we have the symbols for weight digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and the position of a digit in the number determines its value. We call the position values...
ones, tens, hundreds, thousands, and so on. Moving a given digit one position to the left increases its value by a factor of 10:

<table>
<thead>
<tr>
<th>Hundreds</th>
<th>Tens</th>
<th>Units</th>
<th>Position value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Decimal number</td>
</tr>
</tbody>
</table>

In the units position the digit 5 represents the decimal number 5 (5 units), in the tens position it represents 50 (5 x 10), and in the hundreds position it represents 500 (5 x 100).

In the binary system each position to the left increases the value of the symbol by a factor of 2. Thus the position values in the binary system are ones, twos, fours, and so on:

<table>
<thead>
<tr>
<th>Eights</th>
<th>Fours</th>
<th>Twos</th>
<th>Ones</th>
<th>Position value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Binary number</td>
</tr>
</tbody>
</table>

The digit 1 in the ones column represents 1, in the twos column it represents 2, in the fours column it stands for 4, and so on. Thus the number 1111 in the example would be equal to \((1 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1)\), or decimal 15. The binary number 101 would be equal to \((1 \times 4) + (0 \times 2) + (1 \times 1)\), or decimal 5.

The principles involved in converting binary numbers into our usual decimal numbers are illustrated by the following examples:

Example 1:

<table>
<thead>
<tr>
<th>(2^4)</th>
<th>(2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>Exponent value</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>Column value</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Binary number</td>
</tr>
</tbody>
</table>

\((1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (0 \times 1)\)  Decimal value

\(16 + 0 + 4 + 0 + 0 = 20\) (decimal)

Example 2:

\(8 + 4 + 2 + 1 = 11\) (decimal)
HOW TO CONVERT DECIMAL NUMBERS INTO BINARY NUMBERS
The following example shows how to convert decimal numbers into their binary equivalents.

Example: Write the decimal number 43 as its equivalent binary number.

Procedure
1. Set up the column values as shown.

<table>
<thead>
<tr>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
<th>Column heading</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4 + 2 + 0 = 6 (decimal)</td>
</tr>
<tr>
<td>(2) 0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8 + 0 + 0 + 1 = 9 (decimal)</td>
</tr>
<tr>
<td>(3) 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16 + 0 + 0 + 0 + 1 = 17 (decimal)</td>
</tr>
<tr>
<td>(4) 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>16 + 8 + 0 + 2 + 0 = 26 (decimal)</td>
</tr>
</tbody>
</table>

Table 1-2 lists the decimal numbers 0 through 7 and their binary equivalents.

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Column number</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>Decimal value of each column</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Binary number</td>
</tr>
</tbody>
</table>

2. Examine each column in turn starting with column 6 (the 64 column).

Column 6: 64. This is larger than 43, and we enter a 0 in column 6.

Column 5: 32. This is less than 43, and we enter a 1 in column 5.

Column 4: 16, and 16 + 32 = 48. This is larger than 43, and we enter a 0 in column 4.

Column 3: 8, and 32 + 8 = 40. This is less than 43, and we enter a 1 in column 3.

Column 2: 4, 40 + 4 is more than 43, and we enter a 0 in column 2.

Column 1: 2, and 40 + 2 = 42. This is less than 43, and we enter a 1 in column 1.

Column 0: 1, 42 + 1 == 43. We enter a 1 in column 0, and the job is done.

Thus, decimal 43 = 101011 in binary.
1-2 Logic Symbols, Functions, and Conventions

THE LOGIC DIAGRAM
Gates are the basic universal logic building blocks. Each gate has a special symbol that represents the circuitry in the gate. These logic symbols will be combined to form block diagrams called **logic diagrams**.

Although the logic diagram will be our primary method of symbolizing logic systems, there are also some supplementary forms of notation that serve to interpret the operation of logic circuits and systems.

While the logic diagram indicates how the various gates are interconnected, the supplementary forms tell us exactly what the particular logic circuit is intended to do and when.

THE TRUTH TABLE
One of the best statements of exactly what a particular circuit does (and does not do) is called a **truth table**. The truth table is a formal specifications sheet that describes the exact circuit behavior for every possible set of conditions. Logic circuits normally have a number of inputs and one or more outputs. Each input signal can be only one of two possible values designated as 1 or 0 in a logic discussion and as **high** or **low** when electronic conditions are being emphasized. All input signals are usually independent of each other so that $2^n$ possible combinations of input 0's and 1's exist. The base of 2 is used because there are two possible input conditions, 0 or 1. The exponent (n) is the number of independent inputs to a given circuit. Each output can produce one of two values (0 or 1) and is a function of the various combinations of input signals. The truth table, then, shows all possible combinations ($2^n$) of input conditions and the resultant condition of each output for all the input conditions.

The truth table is a complete statement of logic circuit functions because no condition can exist that is not included in the truth table.

THE BOOLEAN EQUATION
**Boolean algebra** is a special logical algebra that provides the same information as the truth table in the form of equations. It has specific rules for manipulating equations to
discover several alternate logic structures that will perform the desired function. In addition, Boolean notation, in conjunction with a special kind of truth table, provides a tool to insure that any given logical requirements are met with an absolute minimum number of gates.

THE TIMING DIAGRAM
The logic diagram shows how the logic gates are interconnected to form the specific logic system; truth tables and Boolean equations are used to describe what the gates and the system accomplish. The timing diagram tells us when each gate or each part of the system responds with respect to a standard timing signal called the clock signal. These four systems—the logic diagram, the truth table, the Boolean equation, and the timing diagram—provide the methods of describing and explaining logic systems on paper and make up the symbolic language of logic.

ELECTRONIC GATES
Logic circuits are composed of combinations of high-speed electronic switches called gates. These gates are the electronic equivalent of simple conventional switches connected in series or parallel. Various systems combine groups of these series and parallel switches. We will use simple switches only for the purpose of explaining the basic logic functions and combinations.

In most digital circuits we cannot use mechanical switches (or relays) because they are much too slow. Electronic switches have been designed that can switch in much less than a microsecond (10^-6 sec) and, in most cases, in a few nanoseconds (10^-9 sec). To give you some idea of how fast this is, light travels at a speed of 186,000 miles per second or 982,080,000 feet per second. In 1 microsecond light will cover a distance of 982 feet, and in 1 nanosecond it will cover a little under 1 foot (0.982 ft). Many modern electronic switches are capable of switching in the time it takes light to travel the length of a man’s finger.

Input voltages used to represent logic 0’s and 1’s in electronic gates can be any pair of distinctly different voltage levels. However, in practice they have been well standardized, eliminating much of the confusion that existed at one time. There are two logic conventions in use: positive logic and negative logic. In positive logic a logic 1 is represented by the most positive of the two levels, and a logic zero is represented by the less positive level. In current transistor-transistor-logic (TTL) technology positive logic universally uses +5 volts for a logic 1 and zero volts for logic 0\(^1\). The less common emitter-coupled logic family uses positive logic but a —0.8 volt = logic 1 and -1.9 volts = logic 0. The level —0.8 volt is more positive than —1.9 volts.

In the ^-channel field-effect family of logic circuits the output voltages are negative for both levels, as they are in emitter-coupled logic. The absolute voltage levels are somewhat more variable and will be examined later. The logic is positive.

In the ^-channel field-effect family the levels are +5 volts = logic 1 and 0 volts == logic 0. These are the same levels used in the TTL family.

---

\(^1\) When this book was first written in the 1980’s this was certainly the case. Today, TTL logic is mostly obsolete and the dominant technology is called CMOS (Complementary Metal Oxide Silicon). Also, the logic levels have dropped significantly, so that the most common values are 0V (low) and 3.3V (high). AB
Since nearly all modern logic circuits use positive logic, negative logic devices are becoming increasingly rare. In negative logic, the most negative of the two voltage levels represents a logic 1, and the less negative voltage represents a logic 0. These two definitions are fairly recent and other definitions may sometimes be found in older texts.

GATE-CIRCUIT CONVENTIONS
a. A 1 on the input of an electronic gate can also be designated High, Hi, or H.
b. A 0 on the input of an electronic gate can also be designated Low, Lo, or L.
c. A 1 on the output of an electronic gate can also be designated by High, Hi, or H, and a 0 can be represented by Low, Lo, or L.
d. Inputs will be labeled, B, C, and so forth, and the output of a gate will be called f (for function).

SWITCHING CIRCUIT CONVENTIONS
a. An open switch is designated 0. If some switch A is open, A = 0.
b. A closed switch is designated 1. If some switch A is closed, A = 1.
c. If lamp/is lit, we designate that lit condition as a 1.
d. If lamp/is dark, we designate it by: f = 0.

BOOLEAN ALGEBRA CONVENTIONS
There are only three basic logical functions in digital circuits and Boolean algebra: AND, OR, and NOT. Variables in Boolean algebra correspond to the input conditions on a gate (or gates). They are designated A, B, C, and so on and can have only a value of either 0 (low) or 1 (high). Boolean expressions are normally written as equations with an f used (generally by itself) on the left side of the equal sign. The f corresponds to the output function of a gate (or system of gates) and can have either one of two values, 0 or 1. (See Figure 1-1a and c.)

Equations consist of variables joined by operators: ( ü) the Boolean product symbol and (+) the Boolean sum symbol, along with an equal sign and the f (function). For example, \( f = (A \cdot B) + (A \cdot C) \) is read as f equals (A and B) or (A and C).

It is important to understand that the ( ü) and (+) symbols do not describe the same operation as ordinary arithmetic multiplication and addition.

The ( ü) symbol is called AND. The equation \( f = A \cdot B \) is read as f equals A AND B. The (+) symbol is called the OR symbol. The equation \( f = A + B \) is read as f equals A OR B.

THE COMPLEMENT
The complement or NOT function is the third Boolean operator. A bar over a variable\(^2\), constant, or operator (• or +) indicates a NOT or complemented variable, constant, or operator.

If \( A = 1 \), \( \neg A \) (not A) = 0. If \( B = 0 \), \( \neg B \) (not B) = 1.

\(^2\) When writing with a word processor, it is sometimes difficult to write the NOT function with the bar over the variable. Other acceptable representations for NOT are \( \neg A \) and \( /A \).
The complement bar may be read in two ways: \( A \) is read as \( NOT A \), or as \( A \) NOT

**SUMMARY**
1. The AND function (symbolizing the AND gate):
\[
f = A \cdot B \text{ or } f = AB
\]
Read as: \( f = A \) AND \( B \)
2. The OR function (symbolizing the OR gate):
\[
f = A + B \text{ Read as: } f = A \text{ OR } B
\]
3. The complement or NOT function (symbolizing the inverter) is indicated by a bar over a variable (or larger segments of an equation). For example, \( A \) would be read as \( NOT A \), or \( A \) NOT.

There are no squares, square roots, or any other roots or powers in Boolean algebra nor are there any fractions or division operations. There are only two possible values for constants and two possible values for variables, 0 or 1.

**CONSTRUCTING TRUTH TABLES**

**Columns**
The truth table will have the following columns: one for each variable (\( A, B, C, D, \) and so on), one or more \( f \) columns, and one \( m \) column. The \( m \) column simply numbers the rows, always beginning with 0.

**Rows**
The rows contain every possible combination of 0's and 1's, with each row containing one combination. The number of rows on a given truth table depends on the number of variables involved in the corresponding Boolean equation. The number of rows \( = 2^n \), where \( n \) is the number of variables.

Example In the equation \( f = (A \cdot B) + (A \cdot C) \) there are three variables: \( A, B, \) and \( C \). The number of rows is as follows: \( m = 2^n \) where \( n = 3 \), and \( m = 2^3 = 8 \) rows. These would be numbered 0 through 7. We will use these conventions throughout the text.
Procedure (See Table 1-3)
1. Enter the \(2^n\) decimal numbers in the \(m\) column in Table 1-3. Start numbering with zero.
2. For each row, write the binary equivalent of the decimal number in the \(m\) column.
3. The \(f\) column will be used to specify the behavior of the particular logic circuit under consideration.

This procedure satisfies two important requirements: (1) it ensures against duplication and omissions of possible combinations of zeros and ones in the table; and (2) it provides a standard form for all truth tables. This makes it possible to compare all entries in a pair of truth tables by examining only the \(f\) column.

### The AND Gate

**FUNCTIONAL DEFINITION**

The AND gate will produce a 1 (high) output if and only if there is a logical 1 (high) on all inputs at the same time.

The Boolean representation of the AND gate is \(f = A \cdot B\) and is read \(f = A\ AND\ B\).

The AND gate is the electronic equivalent of series-connected switches. Figure 1-1 shows the AND switching circuit, the AND gate symbol, two forms of truth table, and the equation for a two-input and a three-input AND function. The AND gate can have any number of inputs.

**Example** To start a car with an automatic transmission, you must put the shift lever out of gear (neutral or park) AND turn the key to start the car. The conditions are: key turned to "start" AND "shift lever out of gear" equals start. Both conditions must exist at the same time if the car is to start. The equation for this example is as follows:

\[
\text{start} = \text{neutral} \cdot \text{key on}.
\]

**TIMING DIAGRAMS**

Digital circuits are normally driven by pulses derived from a master oscillator called a clock. Diagrams of these pulses in different parts of the system are called timing.
Timing diagrams are often a convenient way to illustrate graphically the behavior of a digital circuit. The timing diagram for a two-input AND gate is illustrated in Figure 1-2.

The input and output conditions are shown on the timing diagram in the same order as they are listed on the truth table.

---

diagrams. Timing diagrams are often a convenient way to illustrate graphically the behavior of a digital circuit. The timing diagram for a two-input AND gate is illustrated in Figure 1-2.

The input and output conditions are shown on the timing diagram in the same order as they are listed on the truth table.
The OR Gate

The OR gate is the electronic equivalent of switches connected in parallel. Figure 1-3 shows the OR switching circuit, the OR gate symbols, the truth tables, and the equations. The truth table indicates that a logic 1 (high) on one or more inputs at the same time will produce a 1 (high) output. The OR gate will output a 0 only when all inputs are 0 (low) at the same time. The OR gate can have any number of inputs.

Figure 1-4 shows a four-input OR switching circuit) gate symbol, truth table, and equation. Figure 1-5 is the timing diagram for a two-input OR gate. The extension wings in Figure 1-4c are sometimes used to prevent the input lines from appearing too crowded on the drawing.

The AND gate symbol may also be extended in a similar fashion as shown in Figure 1-6. These extension lines can also have a special symbolic meaning as an indication that the

---

3 Sometimes called inclusive OR.
gate shown—is actually constructed of more than one gate. When more inputs are required than are available in a readily obtainable gate package, several gates may be connected to function as a single basic gate. Only experience can tell you whether this meaning is intended in a given situation.

Example: You and an associate have a joint checking account. Either of you can write checks without consulting the other. The bank will honor your signature or your associate's. If both signatures appear on the check, the bank will still honor the check.

![The OR switching circuit](image)

**Figure 1-3 The OR function**

The inverter, the simplest of the three gates, has one input and one output. The output is always the complement of the input. The complement of a 1 is 0; the complement of 0 is 1. A 1 (high) at the input of an inverter produces a 0 (low) at the inverter output. A 0 (low) input produces a 1 (high) output.

Notice that Figure 1-7 shows two different symbols, one with a circle on the input and the other with the circle on the output. The circle, often called a bubble, is always a part of the inverter symbol, and identifies the symbol as an inverting amplifier.

In the case of multiple inverters in series: (1) An even number of inverters yields the same results as no inverters at all, and (2) an odd number of inverters is equivalent to a single inverter. (See Fig. 1-8.) Figure 1-9 shows the inverter timing diagram.
Figure 1-4 The Four input OR Gate

Figure 1-5 The two-input OR gate timing diagram

Figure 1-6 AND Gate Symbol with extensions to accommodate several inputs without crowding the drawing
Basic Boolean Operations

Because Boolean equations are such a common and useful form for describing digital circuits, it is important to be familiar with the basic laws of Boolean algebra and how they are applied to digital circuits. Some of these properties have already been briefly examined, but now we will formalize the already familiar ones and examine some that we have not yet encountered in this text.

1. The AND operation, equation: \( f = A \cdot B \)
This operation is often called the *Boolean product*. The AND function is unique to Boolean algebra and should not be confused with binary arithmetic multiplication. Table 1-4 shows the truth tables for Boolean product, combinations.

2. The OR operation equation: \( f = A + B \)
The OR operation is referred to as a *Boolean sum*. Again, this is a unique Boolean function not to be confused with arithmetic addition. Figure 1-11 shows the electronic implementation of the OR function and Table 1-5 provides truth tables of Boolean sum combinations.

![Figure 1-10 Electronic Implementation of A and B](image)

![Table 1-4 Tables of Boolean Product Combination](image)

3. The NOT operation and the laws of complementation:
   First law of complementation:
   - If \( A = 0 \), \( \sim A = 1 \)
   - If \( A = 1 \), \( \sim A = 0 \)
Second law of complementation:
\[ A \cdot \sim A = 0 \]

Third law of complementation:
\[ A + \sim A = 1 \]

Law of double complementation:
\[ \sim \sim A = A \]

Figure 1-12 illustrates the electronic interpretation of the laws of complementation.

---

The Commutative Laws

There are two commutative laws in Boolean algebra, one for the logical AND function and one for the logical OR function. Both commutative laws state that order is not important, that A may be ORed or ANDed to B, or B ORed or ANDed to A with the same result.

Commutative law for the OR function:
\[ A + B = B + A \]
Commutative, law for the AND function:

\[ A \land B = B \land A \]

**The Associative Laws**
The associative laws apply when three or more variables (or constants) are to be combined by the AND or OR function. The associative laws state that the variables may be combined in any order without changing the outcome.

Associative law for the OR function:

\[ A + (B + C) = C + (A + B) \]

Associative law for the AND function:

\[ A \land (B \land C) = C \land (A \land B) \]

**The Distributive Laws**
In addition to specifying rules for grouping and multiplying out, the distributive laws also lead to a kind of factoring that will prove to be of value later on.

First distributive law:

\[ A \land (B + C) = (A \land B) + (A \land C) \]

Second distributive law:

\[ A + (B \cdot C) = (A + B) \cdot (A + C) \]

**The Laws of Tautology**
First law of tautology:

\[ A \land A = A \]

because:

If \( A = 1 \), \( 1 \land 1 = 1 \), and
If \( A = 0 \), \( 0 \land 0 = 0 \)

Second law of tautology:

\[ A + A = A \]

because:

If \( A = 1 \), \( 1 \lor 1 = 1 \), and
If \( A = 0 \), \( 0 \lor 0 = 0 \)
When constants are involved:

\[
\begin{align*}
A \cdot 1 &= A \\
A \cdot 0 &= 0 \\
A + 1 &= 1 \\
A + 0 &= A
\end{align*}
\]

**The Laws of Absorption**

First law of absorption:

\[A \cdot (A + B) = A\]

Second law of absorption:

\[A + (A \cdot B) = A\]

In both cases in Figure 1-13, a 1 on A will produce a 1 at f. B can be either 1 or 0 without affecting the output at f. If there is a 0 on A, the output will be 0 for either \(B = 0\) or \(B = 1\). The B has no influence over the output result and can be dropped from the system.

**DeMorgan's Laws**

DeMorgan's laws describe the dual nature of Boolean algebra. When DeMorgan's laws are applied to the implementation of Boolean functions using electronic gates, they can be interpreted to mean: *Any Boolean function can be accomplished using either AND gates and inverters or OR gates and inverters.*

Boolean expressions are often written initially in a form that implies the use of all three gate forms (AND, OR, Invert) to implement the function. DeMorgan's laws provide the basis for the use of most modern electronic gate families. These commercial gate families are based on, for example, an AND gate and a built-in inverter. In this example the OR gate would exist in the family but would probably be more expensive and not as frequently used.

DeMorgan's laws imply that the OR gate would not actually be necessary and that any circuit requirements could be met using only the AND-inverter gate package.
**DEMORGAN'S LAW: CASE I**

\[ \neg(A \cdot B) = \neg A + \neg B \]

Figure 1-14 shows the gate circuits corresponding to the expression on each side of equation 1. The truth tables shown are identical. Because every possible condition is included on the truth table, identical truth tables provide proof that the two circuits are functionally identical.

The truth table entries in Figure 1-14a show all of the possible conditions of 0's and 1's for inputs \( A \) and \( B \). Under the heading "\( A \cdot B \), output," the entries define the functional operation of any AND gate.

The tilde in front of \( A \cdot B \) \( \neg(A \cdot B) \) means that the entire expression is complemented by "passing" it through an inverter. The complete logic function \( A \cdot B \) requires the use of an AND gate and an inverter as shown in Figure 1-14a.

The rightmost column in the truth table, the result of the complete function, is the defined function. The column adjacent (to the left) of the \( \neg(A \cdot B) \) column is shown here for the purpose of explanation only. It would not normally be included because it merely represents one of the steps in obtaining the desired function \( f = \neg(A \cdot B) \).

Figure 1-14b shows the logic diagram for \( f = \neg A + \neg B \). The truth table shows the headings of the two leftmost columns as \( \neg A \) and \( \neg B \). Each 1 appearing in the truth table in part a of this figure, columns, \( A \) and \( B \), shows up as a zero in the table in part b. Each 0 in the truth table in part a (columns \( A \) and \( B \)) shows up in part b as a 1. The \( f \) column defines the functional behavior of any OR gate for all possible combinations of 0's and 1's for variables \( A \) and \( B \).

These two configurations are called logic duals because, although they are structurally different, they perform identical functions.

**DEMORGAN'S LAW: CASE 2**

\[ f = \neg(A + B) = \neg A \cdot \neg B \]

Figure 1-14, Case 2a shows the truth table and logic diagram for this pair of duals. In both cases of DeMorgan's laws the following applies:

<table>
<thead>
<tr>
<th>Inverted Output</th>
<th>Inverted Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Positive Logic</strong></td>
<td><strong>Logic Dual</strong></td>
</tr>
<tr>
<td>( f = \neg(A \cdot B) )</td>
<td>( f = \neg A + \neg B )</td>
</tr>
<tr>
<td>( f = \neg(A+B) )</td>
<td>( f = \neg A \cdot \neg B )</td>
</tr>
</tbody>
</table>
Figure 1-14  DeMorgan's Laws: Case 1 and Case 2

Case 1

a. Equation: \( f = A \cdot B \)  
(Read as \( f \) equals \( A \) AND \( B \), NOT)

b. The logic dual of \( A \cdot \overline{B} \)

Case 2

a. Equation: \( f = \overline{A} + \overline{B} \)  

b. Equation: \( f = \overline{A} \cdot \overline{B} \)
SUMMARY OF FUNDAMENTAL LAWS

1. Laws of tautology
   (1) $A \cdot A = A$
   (2) $A + A = A$

Constants
   (1) $A \cdot 1 = A$
   (2) $A \cdot 0 = 0$
   (3) $A + 1 = 1$
   (4) $A + 0 = A$

2. Laws of complementation
   (1) $A \cdot \neg A = 0$
   (2) $A + \neg A = 1$
   (3) $\neg \neg A = A$ (double complement)

3. DeMorgan's laws
   (1) $\neg(A \cdot B) = \neg A + \neg B$
   (2) $\neg(A+B) = \neg A \cdot \neg B$

4. Commutative laws
   (1) $A \cdot B = B \cdot A$
   (2) $A + B = B + A$

5. Distributive laws
   (1) $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
   (2) $A + (B \cdot C) = (A + B) \cdot (A + C)$

6. Associative laws
   (1) $A \cdot (B \cdot C) = C \cdot (A \cdot B)$
   (2) $A + (B + C) = C + (A + B)$

7. Laws of absorption
   (1) $A \cdot (A + B) = A$
   (2) $A + (A \cdot B) = A$

The NAND Gate
Two of the most popular modern gate structures are the NAND (not AND) and the NOR (not OR) gates. Any digital circuit can be constructed using only NAND gates or only NOR gates.

The NAND gate is an AND gate with a built-in inverter in the output line. Figure 1-15 shows the NAND gate symbol, its AND-inverter equivalent circuit, and how the law of the double complement can be used to convert a NAND gate into an AND gate. The bubble on the output represents the built-in inverter and is part of the NAND symbol. The AND function is not directly accessible in the NAND gate package.
In Figure 1-15b, notice that the equivalent circuit (and equation) is identical to Figure 1-14, Case 1a.

**The NOR Gate**

The NOR gate is an OR gate with a built-in inverter in the output line. The NOR gate is a NOT-OR gate. The OR function is not available from a NOR gate. Figure 1-16 shows the NOR gate symbol, its OR-inverter equivalent circuit, and how the law of the double complement can be used to obtain the OR function from a NOR gate. The bubble on the output is an integral part of the NOR gate symbol.

Notice that the NOR equivalent circuit (Figure 1-16b) is identical to the Case 2a logic diagram and equation in Figure 1-14. (Note: Properly connected, either the NAND gate or the NOR gate can serve as an inverter. Most logic families that feature NAND or NOR gates as the primary gate form also provide inverters for use when it is more convenient to use them.)
1-15 Using NAND and NOR Gates as Inverters
The drawings in Figure 1-17 indicate methods for using NAND and NOR gates as inverters. It is important that unused inputs on any gate be tied to some low-impedance source generally to the positive side of the power supply or ground. An open input circuit sees a nearly infinite driving impedance and is an invitation for noise or unwanted pulses to sneak in. A circuit may function in a lab with an open OR gate input, for example, but may become erratic in the field where electrical noise levels are higher.
Preferred method

Figure 1-17 Synthesized Gates

Note: The High (Hi) on the upper leg of gate 2 is normally a fixed voltage power supply line, in most cases the same line that supplies power to the gates.

Alternate method

\[ f = \overline{A \cdot B} = A + B \]

NAND gate used as an inverter

a. \((A \cdot B)\) synthesized with NAND gates

Preferred method

\[ f = \overline{A + B} = A \cdot B \]

NOR gate used as an inverter

Note: The GND (Lo) on the upper leg of gate 2 is normally a fixed voltage power supply line, in most cases the same line that supplies power to the gates.

Alternate method

\[ f = \overline{A + B} = A + B \]

NOR gate used as an inverter

b. \((A+B)\) synthesized with NOR gates

Figure 1-17 Synthesized Gates
Figure 1-18 summarizes NAND and NOR gates and their various equivalent forms.

**PART 1**
Defined

- a. NOR gate
  - NOR truth table
- b. AND equivalent

**PART 2**
The NOR gate and its equivalent circuits

- c. AND equivalent
- d. AND equivalent
  - AND equivalent truth table

**Figure 1-18 NOR and NAND Gate Circuits and Equivalents**
PART 3
NAND-OR equivalents

a. OR gate

b. NAND equivalent
c. NAND equivalent

d. NAND equivalent

NAND equivalent truth table

PART 4
The OR-NAND equivalents

Figure 1-18 (continued) NOR and NAND Gate Circuits and Equivalents
1-16 Bubble Notation
The use of bubble notation makes logic diagrams less cluttered, easier to read, and easier to draw.

SOME RULES FOR BUBBLE NOTATION
1. The bubble on the output of a gate is apart of that particular symbol and the indicated inverter is built into the gate.
2. The input bubbles do not indicate whether the inverters are internal to the gate or connected externally. In general, for basic gates the inverters are connected externally and are not, a part of the gate circuit. The best interpretation of an input bubble is to consider that input as a low active input. It takes a zero (low) instead of a one (high) on a bubbled input leg to activate that input. Some logic circuits, particularly flip-flops, do have built-in low active inputs.

DeMorgan's Law and Logic Gate Equivalents
The equivalent circuits in Figure 1-19 are based on the two cases of DeMorgan's law:

Case 1: \((A \cdot B) = \overline{A} + \overline{B}\)
Case 2: \((A + B) = \overline{A} \cdot \overline{B}\)

![Figure 1-19 DeMorgan's Law Gate Equivalents](image)
Figure 1-19 shows the equivalent logic gates (logic duals) based on the two cases of DeMorgan’s law. The symbols differ from Figure 1-13 in that inverters are symbolized by circles on the inputs and outputs of the gates. This method of presentation is called *bubble notation*, derived from MIL-STD 806B and the American National Standards Institute (ANSI) y32.14.

**Introduction to Gates and Pulses**

Most logic systems are pulse-operated. Combinations of pulses arriving at the inputs of a logic circuit produce a string of output pulses according to the operating rules for the particular circuit. In pulse operation both high and low levels and timing are important considerations.

Figure 1-20 illustrates pulse operation of four basic gates.

**A Simple Logic Circuit Example**

The simple logic circuit in Figure 1-21 makes the "decision"-to sound or not to sound a warning buzzer. The buzzer will sound when:

The seatbelt is *not* fastened AND 
The ignition switch is *on* AND 
The gearshift is in *reverse* OR *drive*.

**Conventions:**

- Buzzer sounds: $f=1$
- Closed switch = 1
- Open switch = 0
- $A =$ Seatbelt switch
- $B =$ Ignition switch
- $C =$ Drive switch
- $D =$ Reverse switch

**ADVANCED TOPICS**

The following Boolean techniques are not likely to be needed on a daily basis, but there will be times when you will not be able to solve the problem at hand without resorting to them.
Figure 1-21 Simple Logic Circuit Example

a. Switching diagram

b. Logic diagram

Sound buzzer \( f \) = ignition ON \( B \) AND seatbelt not fastened \( \overline{A} \) AND (gearshift in drive \( C \) OR in reverse \( D \))

c. The Boolean equation in words

\[ f = \overline{A} \cdot B \cdot (C + D) \]

d. The equation in symbols
Demorganizing an Equation

In more complex systems it sometimes becomes difficult to determine the exact function of the system without resorting to the formal technique of demorganing equations. The problem arises because entire Boolean terms or even complete equations are often inverted. When a minterm-form equation is inverted it becomes a maxterm-form equation. When a maxterm-form equation is inverted it becomes a minterm-form equation.

The most common form of Boolean equations is the minterm form, often called the sum-of-products form. For example, \( f = (A \cdot B \cdot C) + (A \cdot \sim B \cdot \sim C) \) is a minterm equation where variables within a term are ANDed and individual terms are ORed together. The maxterm form, also called the product-of-sums, takes the form \( f = (A + B + C) \cdot (A + B + C) \), where the variables within the terms are ORed and individual terms ANDed together.

Demorganization is simply an easy way to determine what happens to equations and parts of equations when they are passed through an inverter.

The Rules

1. Complement each variable, including the \( f \), on the left-hand side of the equation.
2. Change all AND operators to OR operators and all OR operators to AND operators.
3. Clear all double complements.

Example:

Demorganize the equation:

\[
f = (A \cdot B \cdot \sim C) + (A \cdot \sim B \cdot C) + (\sim A \cdot \sim B \cdot \sim C)
\]

Complementing both sides and changing operators, we have:

\[
\sim f = (\sim A + \sim B + \sim C) \cdot (\sim A + \sim B + \sim C) \cdot (\sim A + \sim B + \sim C)
\]

Since \( \sim \sim A = A \), we can clear the equation of double complements and write the equation as:

\[
\sim f = (\sim A + \sim B + C) \cdot (\sim A + B + \sim C) \cdot (A + B + C)
\]

Example: Demorganize the equation:

\[
\sim f = (\sim A + B + \sim C) \cdot (\sim A + B + \sim C)
\]

Complementing and changing operators:

\[
\sim \sim f = (\sim \sim A \cdot \sim B \cdot \sim C) + (\sim \sim A \cdot \sim B \cdot \sim C)
\]
Clearing the equation of double complements:

\[ f = (A \cdot \neg B \cdot C) + (A \cdot \neg B \cdot \neg C) \]

This is the finished minterm form equation.

**Complement-Bar Notation**

Because of the dual nature of Boolean algebra, complement bars can be used as an instruction to complement an entire term or equation as well as individual variables. Some examples of complement-bar notation, coupled with a little practice, will provide you with ample facility to handle this kind of notation.

Example: The long complement bar in this example is an implicit instruction to demorganize the equation.

Given the following equation, eliminate the long complement bar by performing the indicated operation (demorganization):

\[ f = (A + \neg B + C) \cdot (\neg A + B + \neg C) \]

Complementing all variables and changing operational signs, we get:

\[ \neg f = (\neg A \cdot \neg B \cdot \neg C) + (\neg A \cdot B \cdot \neg C) \]

Clearing the double complements, we get:

\[ f = (\neg A \cdot \neg B \cdot \neg C) + (A \cdot \neg B \cdot C) \]

Example Demorganize the equation:

\[ f = (\neg A \cdot B \cdot \neg C) + (A \cdot \neg B \cdot C) \]

Complementing all variables and changing operational signs, we get:

\[ \neg f = (\neg A + \neg B + \neg C) \cdot (\neg A + \neg B + \neg C) \]

Clearing the double complements yields:

\[ \neg f = (A + \neg B + C) \cdot (A + B + \neg C) \]

This example illustrates the case where a long complement bar exists over a single term. The complemented term is demorganized, but the rest of the equation is not altered.
Example: Given the following equation, remove the long complement bar by performing the operations indicated:

\[ f = (\sim A \cdot \sim B \cdot \sim C) + (A + \sim B + C) \]

Complementing the indicated variables and changing operational signs, we get:

\[ f = (\sim A \cdot \sim B \cdot \sim C) + (\sim A \cdot \sim B \cdot \sim C) \]

Clearing the double complement, we get:

\[ f = (\sim A \cdot \sim B \cdot \sim C) + (\sim A \cdot \sim B \cdot \sim C) \]

Example: This example covers the case where both of the previous cases are included in a single equation.

Given the following equation, remove the long complement bars by performing the indicated demorganizations:

\[ f = (\sim A \cdot B \cdot \sim C) + (A + B + C) \]

Removing the longer bar, we get:

\[ \sim f = (\sim A \cdot B \cdot C) \cdot (A + B + C) \]

Clearing the remaining long complement bars yields:

\[ \sim f = (A + \sim B + \sim C) \cdot (A + B + C) \]
A logic circuit is composed of one or more logic gates. The inputs to the gates consist of a high or low voltage, representing logic 1's and logic 0's. Circuit outputs are also either high or low, logic 1 or logic 0. There are two kinds of logic: combinatorial (also called asynchronous or direct logic) and sequential or synchronous logic. Synchronous logic responds to the input conditions only at specific times controlled by a master pulse generator called a clock. Circuit operation is synchronized to the clock. Asynchronous logic responds as the input conditions change. No clock input is provided and the circuit is not synchronized with the system clock.

We will examine logic operations starting with very basic ones and working up to fairly complex systems. At each level of complexity we will study the most appropriate methods of representing and manipulating equations and logic diagrams. At the lowest level of complexity we will examine the relationship among truth tables, equations, and logic diagrams. We will also investigate the dual nature of logic at this level.

At the next level of complexity we will work with Karnaugh mapping and simplification methods.

The various levels of complexity are dictated by the commercial availability of logic circuits, rather than purely by logic considerations. Certainly logic considerations have influenced manufacturing decisions, but their influence was tempered by practical considerations. Each method is best suited to a fairly specific complexity level and is either very difficult or completely impossible to use for greatly different levels. In Chapter 4 we will examine programmable logic on an MSI scale.

**Single-Input Logic**

There is only one valid logic function for a single input gate—inversion. Because there are two possible input values, 0 and 1, and two output values (also 0 and 1), we can make truth tables representing four different single-input gate functions as shown in Table 3-1.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3-1 All possible single input logic functions

Truth tables 1 and 2 are useless because the output condition is not influenced by input conditions and thus performs no logic function. Table 3 is a logic "do-nothing" circuit
because the output condition simply follows the input condition. Although the circuit represented by truth table 3 is a useful circuit element in the form of a non-inverting buffer amplifier, it cannot be considered as a logic element. The number 4 truth table is the table that describes the inverter function. Table number 4 is the only table which represents a valid logic function.

**Two-Input Logic**
For two-input devices it is possible to write sixteen different truth tables. Six of them are useless, six are common and most often used, and the remaining four are used only in very unusual situations. The following are the useless truth table forms:

1. Where all outputs are always 0.
2. Where all outputs are always 1.
3. Where the outputs are identical to input \( A \).
   - Here input \( B \) does not affect the output under any condition. The circuit is, in effect, a single-input circuit with input \( A \).
4. Where the outputs are identical to the inputs on \( B \). In this case, we have a single-input circuit with input \( B \).
5. Where the outputs are all inversions of input \( A \). The circuit has degenerated into a single-input inverter for input \( A \).
6. Where the outputs are the inversion of input \( B \). Here we have a single-input inverter with input \( B \).

These cases, in which the two-input circuit degenerates into a single-input non-inverting amplifier or an inverter, are useful in practice even though their logical significance is trivial.

Because actual IC gate packages come with several gates to a package it is often practical, for example, to use an extra NAND gate in an existing package rather than adding an additional inverter package.

**Logic State Definitions**
There are two kinds of logic: positive and negative.

Positive logic definition:
\[
+ = \text{Logic 1} \\
\text{Ground} = \text{Logic 0}
\]

Negative logic definition:
\[
\text{Ground} = \text{Logic 1} \\
+ = \text{Logic 0}
\]

We stated earlier that TTL and most other modern logic circuits are positive-logic devices. This is true in the sense that manufacturers define their products' functions in terms of the positive-logic definition. If a manufacturer had been making 7400 NAND gates and arbitrarily decided to change the logic definition from positive to negative, the 7400 would have to be listed in the new data manual as a NOR gate. The same electronic circuit with a different logic definition performs a different logic function. Fortunately,
manufacturers do not take such arbitrary liberties with the logic definition; however, it is often desirable for us to alter the logic definition for our own convenience.

**Gates and Logic Duals**
The concept of logic duals allows the selection of whichever logic definition best fits the problem at hand. The justification for logic duality is contained in the two laws of DeMorgan.

**LOGIC DUAL DEFINED**
From here on we will adopt the following definition for logic duals: *Two logic circuits are defined as logic duals when both have identical logic truth tables but opposite logic definitions.* One is defined as positive logic; the other is defined as negative logic.

With the preceding definition in mind, let us examine each of the most useful and common two-input logic circuits and their duals.

Figure 3-1 reviews bubble notation from Chapter 1.

Figure 3-2 shows positive-logic gates and their negative-logic equivalents. Bubbles on the gate inputs define the gate as a negative-logic gate. The negative-logic symbols in Figure 3-2 (b, d, f, h) are sometimes found in logic diagrams even though negative-logic gates are rarely available in integrated circuit hardware. For example, you might find the negative-logic OR gate (Fig. 3-2b) in a logic diagram, but the hardware would actually be its positive-logic equivalent, the AND gate in Figure 3-2a. A logic input with a bubble is also called an *active-low input.*

**Minterm and Maxterm Forms**
Minterm and maxterm are two common forms for logic equations and the associated arrangement of logic gates, one or both of which is the basis for every asynchronous logic circuit. The names refer to an obsolete form of truth table and are abbreviations of the full names which are no longer used—minimum-area-term and maximum-area-term. Figure 3-3 shows the equation and logic diagram for the minterm and maxterm forms. In practical circuits, inverters (when called for) are generally found in series with selected
input lines. Figure 3-4 shows the same circuits as Figure 3-3 but with inverters used to obtain the desired inverted input signals. In Figure 3-4a inputs are shown connected together—A on gate 1 to A on gate 2 and B on gate 1 to B on gate 2. In Figure 3-4b, the connections are implied by the letter designations. Both are common methods of drawing the logic diagram. The minterm circuit in Figure 3-3a and the maxterm circuit shown in part b both perform the same logic function.

<table>
<thead>
<tr>
<th>Positive-logic gates</th>
<th>Negative-logic gates</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 3-2 Bubble notation reviewed

Equation: \( f = \overline{(A \cdot B)} + (A \cdot \overline{B}) \)

Equation: \( f = (A + B) \cdot (\overline{A} + B) \)

Figure 3-3 Minterm and Maxterm forms
EXAMPLES OF MINTERM AND MAXTERM EQUATIONS

The minterm form may be identified as a sum-of-products form.

Examples:
1. \( f = (\neg A \cdot \neg B \cdot C) + (A \cdot \neg B \cdot C) + (A \cdot B \cdot C) \)
2. \( f = (A \cdot \neg C) + (B \cdot D) + (A \cdot \neg C \cdot \neg D) + (A \cdot B \cdot \neg C \cdot D) \)
3. \( f = (A \cdot B \cdot C) + (A \cdot \neg D) \)
4. \( f = (A \cdot B \cdot \neg C) + (A \cdot \neg B \cdot C) + (B \cdot \neg C) + (A \cdot \neg B) \)

The maxterm form may be identified as a product-of-sums form.

Examples:
1. \( f = ((A + B + \neg C) \cdot (\neg A + B + C) \cdot (A + \neg B + C)) \)
2. \( f = (\neg C \cdot (A + B + D) \cdot (A + B + C + D) \cdot (A + B + C + D)) \)
3. \( f = (A + B + C) \cdot (A + B + C + D) \cdot (A + B + C + D) \cdot (A + D) \)
4. \( f = (A + B + C) \cdot (A + B + C) \cdot (A + B + C) \cdot (A + B + C) \cdot (A + B + C) \cdot (A + B + C) \)

As a direct result of DeMorgan's law:
1. Any logic operation can be implemented in either a minterm or a maxterm form.
2. Any minterm-form logic circuit can be converted into a maxterm circuit.
3. Any maxterm-form logic circuit can be converted into a minterm circuit.

The two forms are complementary. In order to understand what is meant by complementary, let us examine Tables 3-2 and 3-3.

In Table 3-2 we have made a truth table for some arbitrary function. Later we will deal with more realistic situations.

THE MINTERM FORM
The minterm type equation, often called the sum-of-products form, can be written directly from the truth table by interpreting a 0 under the \( A \) heading as \( \neg A \) and a 1 in the \( A \) column as \( A \). Entries under the \( B \) heading are treated in the same way.
The conditions in the $A$, $B$, and other columns are interpreted and written only for rows in which there is a 1 in the $f$ column. In Table 3-2a the $m_0$ row has a 1 in the $f$ column. The 0,0 under $A,B$ is written as $(\overline{A} \cdot \overline{B})$. This is the first minterm. The next 1 entry in the $f$ column is row $m_2$ and is written as $(A \cdot \overline{B})$, as shown under the heading minterms. The final equation is formed by joining the minterms (AND-terms) with OR (+) symbols as shown in part c of this table. This is the minterm, or sum-of-products form, equation.
The logic diagram for the equation in $d$ is shown in two equally acceptable arrangements. The two input gates (I and 2) are AND gates with the necessary inverters (shown or implied) to form the hardware equivalent of the two minterms. The OR gate joins the two products in the way indicated by the equation.

Example:

Write the equation and draw the logic diagram as described by the truth table 3-3. If we combine the terms to form an equation we get:

$$f = (\sim A \cdot \sim B \cdot C) + (A \cdot \sim B \cdot C) + (A \cdot B \cdot \sim C)$$

Figure 3-5 shows the logic diagram.

THE MAXTERM FORM

Although the minterm form is the most common, the maxterm form is often used, so it is necessary to be able to translate from one form into the other. Truth tables can be assumed to be minterm truth tables unless otherwise specified. The maxterm truth table is primarily used only as an aid in translating from one form to another or for other special purposes.

In Table 3-4a we have repeated the minterm truth table from Table 3-2. Table 3-4b is the complementary (maxterm) truth table. Truth table 3 is formed by inverting all entries. All 0's are changed into 1's and all 1's become 0's. The maxterms in c are written from the maxterm truth table. The completed maxterm, or product-of-sums equation, is formed by joining the sum terms (maxterms) by an AND symbol as shown in Table 3-4d. The maxterm logic diagram is constructed as shown in Figure 3-7.

It is important to understand that the two logic circuits, the minterm and the maxterm forms, perform identical logic functions. From a Functional logic standpoint, they are identical and completely interchangeable.

Figure 3-4 Logic Circuit Design for Table 3-3
### Table 3-4 Maxterm Truth Tables, Equations and Logic Diagrams

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₀</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M₁</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>M₂</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>M₃</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M₄</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M₅</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>M₆</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>M₇</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ (A + \overline{B} + \overline{C}) \]

\[ (\overline{A} \cdot \overline{B} \cdot \overline{C}) \]

Note: Lower-case \( m \) indicates minterm and upper-case \( M \) indicates maxterm

### Table 3-5 Maxterm and Minterm Truth Tables

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₀</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m₂</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m₃</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m₄</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m₅</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m₆</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m₇</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ (\overline{A} \cdot \overline{B} \cdot \overline{C}) \]

\[ (A \cdot \overline{B} \cdot \overline{C}) \]

\[ (A \cdot \overline{B} \cdot \overline{C}) \]

\[ (A \cdot B \cdot C) \]

\[ (A \cdot B \cdot C) \]

\[ (A \cdot B \cdot C) \]

\[ (A \cdot B \cdot C) \]

\[ (A \cdot B \cdot C) \]

a. Maxterm truth table for \[ f = (\overline{A} + \overline{B} + \overline{C}) \cdot (A + \overline{B} + \overline{C}) \]

b. Inverting to a minterm table

c. Minterms

Table 3-5 Maxterm and Minterm Truth Tables
Equation: \( f = (A + \overline{B}) \cdot (\overline{A} + \overline{B}) \)

![Maxterm Logic Diagram](image)

Example: Given the following maxterm equation, convert it into its minterm equivalent:

a. The equation:
\[
f = (\overline{A} + B + \overline{C}) \cdot (A + \overline{B} + \overline{C})
\]

b. The maxterm truth table is shown in Table 3-5a.

c. Inverting all entries in the maxterm table a yields table b, the minterm table. The minterms corresponding to the 1 entries in the minterm table \( f \) column are indicated in part c. Connecting the minterms by + (OR) symbols yields the equation:
\[
f = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)
\]

On the surface it would appear that the minterm equation is far more complex than the maxterm version. What has actually happened is that a number of redundant terms are included that can be easily eliminated.

When a term accomplishes nothing that is not already accomplished by other terms, it is said to be redundant and can be dropped from the equation. We will find little or no difference between the complexity of the minterm and maxterm equation once redundant terms are eliminated. In the next section we will examine formal simplification (minimization) techniques that are used to eliminate the redundant terms. Redundancies can crop up in either or both forms of the equation.

**Simplification**

The process that we have been using to derive equations and logic diagrams from truth tables always yields valid results. But more often than not circuits derived from the truth table contain more hardware than is really necessary to get the job done. The problem is simply one of overlapping functions where part of a term will do the work of two or more complete terms.
Suppose we have the equation:

\[ f = (A \cdot B \cdot C) + (A \cdot B \cdot \neg C). \]

The equation says that we get the desired function when we have \((A \cdot B)\) combined with either \(C\) or \(\neg C\). If either \(C\) or \(\neg C\) will do, and every possibility includes one or the other, the term \(C\) is useless (complemented or uncomplemented). It can therefore be dropped from the equation. This leaves:

\[ f = (A \cdot B) + (A \cdot B). \]

Because the two terms are identical, one of them is a duplicate, or redundant, term. Therefore, we keep only one of them. The function can be performed by simply \(f = (A \cdot B)\) just as well as by the more complex \(f = (A \cdot B \cdot C) + (A \cdot B \cdot \neg C).\)

**AN ANALOGY**

\(f = \) the requirements for driving a nail:

\(f = \) (a nail AND a hammer AND an apron) OR (a nail AND a hammer AND NO apron).

The apron is obviously unimportant. Forgetting about the apron, we have:

\(f = \) (a nail AND a hammer) OR (a nail AND a hammer).

Obviously, what we need is simply a nail and a hammer. \(f = \) (a nail AND a hammer).

The analogy is fine as far as it goes, but we need to build on a more rigorous premise. We find the foundation for simplification methods in the basic Boolean laws we examined in Chapter 1. The third law of complementation, \(A + \neg A = 1\), forms the foundation for the discussion that follows.

**THE RULES FOR COMBINING TERMS**

Two terms may be combined whenever:

1. Each term contains exactly the same variables.
2. The terms to be combined are identical with the exception that one—and only one—variable appears in the complemented (barred) form in one term and in the uncomplemented form in the other.

**Examples:**

1. \( f = (A \cdot B \cdot C) \cdot (\neg A \cdot B \cdot C) \)

Because \(A\) appears in both complemented and uncomplemented forms and because \(A + \neg A = 1\), the \(A\) drops out, leaving \(f = (B \cdot C) + (B \cdot C)\). Because the two new terms are identical, one of them is said to be redundant and is dropped. The final simplification of the equation is \(f = (B \cdot C)\).

2. \( f = (\neg A \cdot B \cdot \neg C \cdot D) + (\neg A \cdot B \cdot C \cdot D) \)

The \(C\) appears in the complemented form in the first term and in the uncomplemented form in the second term. The \(C\) drops out, leaving \(f = (A \cdot B \cdot D) + (A \cdot B \cdot D)\). Because the two terms are identical, one of them is redundant and is dropped, leaving only \(f = (A \cdot B \cdot D)\).
3. Simplify the following equation by combining reducible pairs:
\[ f = (~A \cdot ~B \cdot ~C) + (~A \cdot B \cdot ~C) + (~A \cdot ~B \cdot C) + (~A \cdot B \cdot C) \]

   a. If we combine terms 1 and 2:
   \[ \sim A \cdot ~B \cdot ~C \]
   \[ \sim A \cdot B \cdot ~C \]
   \[ \sim A \cdot B \cdot ~C \]
   \[ \sim A \cdot B \cdot C \]
   \[ \sim A \cdot C \]

   b. If we combine terms 3 and 4:
   \[ \sim A \cdot ~B \cdot C \]
   \[ \sim A \cdot B \cdot C \]
   \[ \sim A \cdot C \]

   c. If we rewrite the equation using the simplified terms:
   \[ f = (~A \cdot ~C) + (~A \cdot C) \]

   d. On inspection we can see that these new terms can be combined to further simplify the equation:
   \[ ~A \cdot ~C \]
   \[ ~A \cdot C \]
   \[ A \]

   e. The final equation reduces to: \( f = A \), the equivalent of a single inverter for A.

4. An alternate pairing for example 3:
\[ f = (~A \cdot ~B \cdot ~C) + (~A \cdot B \cdot ~C) + (~A \cdot ~B \cdot C) + (~A \cdot B \cdot C) \]

   a. Combining terms 1 and 3 we get:
   \[ ~A \cdot ~B \cdot ~C \]
   \[ ~A \cdot ~B \cdot C \]
   \[ ~A \cdot ~B \]

   b. Combining terms 2 and 4:
   \[ ~A \cdot B \cdot ~C \]
   \[ ~A \cdot B \cdot C \]
   \[ ~A \cdot B \]

   c. The simplified equation:
   \[ f = (~A \cdot ~B) + (~A \cdot B) \]
d. Combining the two simplified terms:

\[
\begin{align*}
\overline{A} \cdot \overline{B} & \quad \overline{A} \\
\overline{A} \cdot B & \quad \overline{A}
\end{align*}
\]

e. The simplified equation:

\[f = \overline{A}\]

In this case the end result is the same whatever our selection of reducible pairs. This is not always true. Sometimes we can come up with two or more equally simple but different simplified equations. So far the examples have been trivial and this pairing technique worked well, but as equations get more complex, the method becomes tedious and prone to error.

**THE KARNAUGH MAP**

This is a special truth table designed specifically for simplifying equations. There are several variations of this table—(sometimes called a map), but they are all constructed so that all adjacent entries on the table represent reducible minterm pairs. Although there is more than one way that a Karnaugh map can be labeled and arranged, all versions are used and interpreted in exactly the same way.

Because the Karnaugh map is constructed so that all adjacent squares represent reducible pairs, it is really only an aid in using the pairing technique. The map makes the options obvious and clear because of their highly visual nature. It also allows us to perform multiple pairings in a single operation. Table 3-6 shows the Karnaugh maps for use with two, three, and four variable equations. A careful examination of all the maps shows that each square represents one of the possible minterms on the standard truth table, and that any pair of adjacent squares represents a reducible pair of minterms. For the purpose of this discussion, the indicated minterm has been written for each square, but these are not normally included. Instead, l's taken directly from the "f" column on the standard truth table are entered in the appropriate squares. The rest of the squares are left blank. The size of the map is determined by the number of variables involved and will have as many squares as the standard truth table has rows: \(2^n\) where \(n\) is the number of variables. In your examination of Table 3-6 you may have overlooked the fact that variables on the upper and lower edges and on the left and right edges are also reducible. In addition, the four corners are considered adjacent in all combinations except on the diagonal. No pairs of terms on a diagonal are reducible.

**LOOPING TERMS AND READING OUT SIMPLIFIED EQUATIONS**

**Looping Rules**

1. Each loop should be drawn around the largest group of 2, 4, 8, and so on—adjacent entries possible. The number of entries in a loop must be an integral power of 2.
2. An entry may be involved in any number of loops, but a new loop should not be added unless it includes at least one entry not included in any other loops. Typical looping patterns are shown in Table 3-7.
3. After looping, inspect the map for any loop that encloses entries where all of the l's in the loop are involved in other loops. Remove any such loops.
Reading Out the Simplified Equation

1. Each loop represents a new simplified minterm for the simplified equation. All simplified minterms will be ORed together (connected by the + symbol).

2. Any variable in a given loop which appears in both the complemented and the uncomplemented form drops out.

---

Table 3-6 Karnaugh Maps

Table 3-7 Looping Patterns
3. The variables left make up the simplified term for that loop. Individual variables are ANDed together Q’oined by the • symbol).

Example: Write the simplified terms for each loop in Table 3-7a through f and combine the loops to get the simplified equation for each map.

<table>
<thead>
<tr>
<th>Map</th>
<th>Loop</th>
<th>Term</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>(~A \cdot \sim C)</td>
<td>(f = \sim A \cdot \sim C + B \cdot \sim C + \sim A \cdot B)</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>(\sim A \cdot B)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>(\sim B \cdot \sim C)</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>(A \cdot \sim C)</td>
<td>(f = \sim B \cdot \sim C + A \cdot \sim C + A \cdot B)</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>(A \cdot B)</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>(\sim A \cdot C)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>b</td>
<td>(B)</td>
<td>(f = \sim A \cdot C + B + A \cdot \sim C)</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>(A \cdot \sim C)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>(B \cdot \sim C)</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>(A \cdot B)</td>
<td>(f = B \cdot \sim C + A \cdot B + A \cdot \sim C + \sim B \cdot C)</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>(A \cdot C)</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>(\sim B \cdot C)</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>a</td>
<td>(\sim C)</td>
<td>(f = \sim C + \sim A \cdot \sim B)</td>
</tr>
<tr>
<td>e</td>
<td>b</td>
<td>(\sim A \cdot \sim B)</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>a</td>
<td>(\sim B)</td>
<td>(f = \sim B + A \cdot \sim C)</td>
</tr>
<tr>
<td>f</td>
<td>b</td>
<td>(A \cdot \sim C)</td>
<td></td>
</tr>
</tbody>
</table>

**Don't Cares**

In some situations some of the combinations on a truth table are not defined. For example, a binary code for the decimal digits 0 through 9 requires that only ten of the combinations on a sixteen-row truth table be defined. The left over combinations are said to be don't-care terms. Because they will never occur as a part of the code, it is academic what the logic circuit would do if they did occur. We "don't care" whether the logic circuit produces a 1 or a 0 output for such will-never-happen combinations.

We can enter either a 0 or a 1 in the \(f\) column of the truth table for don't-care conditions. However, since the value we assign can make a difference in the simplification process, don't-care terms on the truth table are designated by an uncommitted symbol such as 0 or X. When loops are being formed on the Karnaugh map, 0 symbols are included in a loop...
if they serve to enlarge a loop—in this case the \( f \) is treated as a 1 entry. If the 0 entry cannot be used to enlarge a loop, it is treated as a 0 and ignored.

As a general rule, the more complex the specifications are (the more truth table entries involved), the more profitable simplification is likely to be. The following example illustrates the point.

Example: Given the following truth table and Karnaugh map (Table 3-13), write the unsimplified and the simplified equations and draw both the simplified and unsimplified logic diagrams.

Table 3-13 A case for simplification

a. The unsimplified equation

\[
f = (\neg A \cdot \neg B \cdot \neg C \cdot \neg D) + (\neg A \cdot \neg B \cdot \neg C \cdot D) + \\
(\neg A \cdot \neg B \cdot C \cdot \neg D) + (\neg A \cdot B \cdot \neg C \cdot \neg D) + \\
(\neg A \cdot B \cdot \neg C \cdot D) + (A \cdot \neg B \cdot \neg C \cdot \neg D) + \\
(A \cdot \neg B \cdot \neg C \cdot D) + (A \cdot B \cdot \neg C \cdot D) \]

Note: b is a redundant term.

Note: Loops b and c represent alternate terms. One or the other must be included in the equation, but not both.
b. The simplified equation:

\[ f = \sim C + (\sim B \cdot \sim D) \]

The unsimplified logic diagram is shown in Figure 3-9a and the simplified version is shown in part b. Notice that the variable \( A \) has dropped out of the system entirely.

**Figure 3-9 Logic Diagram for the minterm type 1 equation**

### Additional Methods of Minimizing Hardware

1. When more than one gate input in a logic structure requires a particular inverted variable, a single inverter can be used in place of multiple inverters as long as fan-out rules are observed. (See Figure 3-10.)
2. The distributive laws can sometimes be taken advantage of when differences in gate delay times for the different variables can be tolerated. An example of the first distributive law is shown in Figure 3-11 and the second distributive law in Figure 3-12.

\[
A \cdot (B + C) = (A \cdot B) + (A \cdot C)
\]

\[
A + (B \cdot C) = (A + B) \cdot (A + C)
\]

Figure 3-10 Combining Inverters

**Minterm and Maxterm Forms Using NAND and NOR Logic Gates**

**MINTERM CIRCUITS**

In Figure 3-1 it was demonstrated that the truth table outputs \( f \) for a positive logic 'OR' and an inverted negative logic AND (NAND) are identical. When NAND logic is used exclusively to form a minterm equation, a NAND gate can replace the OR gate. In Figure 3-13a, a minterm logic circuit is shown using AND-OR-NOT logic. In part b the NAND gates (1 and 2) produce inverted outputs, so what we need is an inverted
negative logic OR gate, which cannot be found in any of the modern logic families. However, the logic dual of a negative-logic OR is a positive-logic NAND (see Figure 3-1). If we substitute a positive-logic NAND for the negative-logic OR, we get the circuit shown in Figure 3-13b.

Because the NAND gate is the basic gate in TTL, most minterm circuits using TTL will be implemented as shown in Figure 3-13b instead of in the form shown in part a. In TTL AND and OR gates are more expensive and less available than NAND gates.

![Figure 3-13 Minterm Circuits Using NAND Logic](image)

**MAXTERM CIRCUITS**

The maxterm form using NOR gates is less frequently used than the NAND minterm form, but it does occur. Figure 3-14a shows a max-term circuit using AND-OR-NOT gates, and part b shows the NOR equivalent circuit.

Because the NOR gates (1 and 2) in Figure 3-14b invert the logic gate, 3 must be a negative-logic AND gate or its dual (inverted) a positive-logic NOR. (See Figure 3-18 for a summary of logic duals.)

**The Exclusive-OR Gate**

The exclusive-OR is a two-input gate that finds a great many applications in digital systems. The truth table for the exclusive-OR is shown. In Table 3-15, and Figure 3-15 is the X-OR (exclusive-OR) logic diagram and symbol. Each circuit is symbolized in Figure 3-15d.
Among the many applications for exclusive-OR gates, two important examples bear mentioning here.

1. Controlled Complementer
   The controllable complement circuit is sometimes called a \textit{controlled inverter} or \textit{true-invert gate}. In this application one input is used as a signal input that produces either a true (non-inverted) output signal or an inverted (complemented) signal, depending upon whether the second input is high or low. Look at truth table 3-15. Let input $B$ be the
signal input and put a fixed 0 level on input $A$. A zero on $B$ produces zero out, and one on $B$ produces a one out—the signal out is not inverted. Now, if input $A$ is set to a fixed 1 level, a zero input on $B$ produces a one output, and a one input produces a zero output—the signal is inverted.

An examination of the truth table (3-15) indicates that control and signal inputs can be exchanged with the same results as before.

2. Binary Adder
The truth table for binary addition and the exclusive-OR table are coincidentally identical, which makes the X-OR an ideal circuit element when binary addition is to be performed. We will examine these and other applications in some detail as we encounter them later in the text.

The X-OR gate, like the inverter, is independent of logic definition. It performs the same logic function for both negative and positive logic. X-OR gates are not available with more than two inputs.

**Expanding the Number of Gate Inputs**
It is frequently desirable to combine several gates with few inputs each to obtain a larger number of inputs, rather than to purchase a fairly expensive package with the required number of inputs. Figure 3-16 shows how this is accomplished.
a. A four-input OR gate composed of three two-input gates

\[ I = A + B + C + D \]

b. A four-input AND gate composed of three two-input gates

\[ I = A \cdot B \cdot C \cdot D \]

c. A three-input NAND gate composed of three two-input gates

\[ I = \overline{A \cdot B \cdot C} \]

d. A three-input NOR gate composed of three two-input gates

\[ I = \overline{A + B + C} \]

e. "Wired AND" three-input NOR gate (open collector)

\[ I = \overline{A + B + C} \]

Figure 3-16 Multiple Input Gates