1

Describing Logic Equations

Before you begin this journey into digital design, it is important that you understand the philosophy that will guide your study. If you have not read the Preface, do so now before you go on. There we discuss the issues that give rise to the need for good style and structure in digital design. Also, the Preface contains an outline of the book, which will give you a view of where you are heading and how you will get there. It is particularly important that you understand our approach to the details of digital hardware. The overriding emphasis is to let the problem solution dictate the hardware, rather than allowing premature commitments to hardware coerce the solution. This conscious suppression of hardware detail during most of the design pays big dividends. Chips and wires and power supplies are still important—they are vital to success, and you will need a good background in many areas of digital technology in order to transform your designs to a commercial product—but too often in digital design the hardware has dominated the solution to the problem. To head off this common malady, we stress the theory of sound design and leave broader issues of production technology to further study.

THE NEED FOR ABSTRACTION, FORMALISM, AND STYLE

Style

The human mind needs help when it tackles complex tasks. As we use the term, style is a method of partitioning a large problem into manageable subunits in a systematic and understandable way. The need for good style is more apparent as problems become larger. The most complex projects ever attempted by human beings have been computer programs; some, exceeding many million lines of code, are so large that no one person is able to encompass the entire program, or even a significant part of it. The study of programming style was forced upon practitioners of the art as a way of gaining control over their projects. Programming style has blossomed into a rather well defined set of techniques, bearing such names as "top-down" and "structured." The hardware of a large computer involves complexity on the same scale as these giant computer programs. The study of style in digital system design is not as well developed as its programming counterpart but is nonetheless essential to success. In this book, we emphasize style.

Here are some rules of good style in digital design:

Design from the top down. The design starts with the specification of the complete system in a form compact enough that one person can readily comprehend it. The design proceeds by sectioning the digital system into subunits such as memory, arithmetic elements, and control, with well-defined interrelationships. You may then describe each unit in more detail and still retain the ability to comprehend both the whole structure and the details of the units. This process continues until you have completely specified the system in detail, at which time construction may begin.

Use only foolproof techniques that will keep you on the narrow path of safety in the design process. Digital hardware allows a high degree of flexibility in design—so much flexibility that designers can bury themselves in clever and unusual circuits. Uncontrolled use of such flexibility promotes undisciplined, unintelligible, and incorrect design of products. This phenomenon has its counterpart (to a less severe degree) in computer software because assembly language programming offers access to the full power of a digital computer. Experience in the solution of hardware and software problems has shown that we must restrict our design tools and techniques to those that can be shown to work reliably and understandably under a variety of circumstances.

Use documentation techniques, at both the system level and the detailed circuit level, that clearly portray what you, the designer, were thinking when you reduced your problem first to an abstract solution and then to hardware. This often-violated precept boils down to common courtesy. Put yourself in the position of a user or a maintainer of your hardware design; in such a position you would be grateful for clear, complete documentation.

Abstraction

In our context, abstraction means dealing with digital design at the conceptual level. The concept of a memory, a central theme in every computer, is an abstraction. When starting a design we need to deal with conceptual elements and their interrelationships: it is only later in the design process that we need to worry about the realization of the concepts in hardware. This freedom from concern about the details of various hardware devices is absolutely essential if we are to get a good start on a new design of any complexity. Start at the top and begin reducing the problem to its natural conceptual elements. For example, a computer will need a memory, an input-output system, an arithmetic unit, and so on. We ordinarily begin a design at this highly abstract level, and carry the conceptual process down, level by level. Thus, at the next level we draw a block diagram of an arithmetic unit by interconnecting functional units such as registers, control units, and data paths. The initial abstraction is a critical part of any design, since bad early planning will inevitably lead to bad implementations. There is no way to rescue a bad design with clever tricks of Boolean algebra or exotic integrated circuits.

Formalism

Formalism is the theory of the behavior of a system. In a digital design, formalisms help us to establish systematic rules and procedures with known characteristics. Formalisms are important at all levels of design. In the traditional study of digital systems, one of the principal formalisms is Boolean algebra, the theory of binary logic, named after George Boole, who studied it long before the advent of digital computers. Boolean algebra is an essential tool for describing and simplifying the detailed logical processes at the root of digital design. Powerful and well developed as Boolean algebra is, it nevertheless becomes of less benefit as our level of abstraction increases. For instance, at the top ("systems") level of abstraction, where we are thinking in terms of the movement, storage, and manipulation of data, Boolean algebra is of little use. As we move closer to the detailed implementation—as our design becomes less abstract and more concrete Boolean algebra begins to be a useful tool.

At the systems level, the formalisms are less well developed, appearing as structures and rules rather than mathematical constructs. High-level formalisms are nevertheless of great importance to good design, for only by adopting systematic methods at all levels can we hope to transform correct concepts reliably into correct hardware.

LOGIC IN DIGITAL DESIGN

Imagine trying to speak without the words *and, or,* and *not.* Discarding these little words would severely handicap our ability to express complex thoughts. Although *and, or,* and *not* each have several meanings in English, the most profound uses describe logical combinations of thoughts: "I have money for gas *and* my car is running"; "There is a paper jam *or* the printer is out of paper"; "She will *not* fail." Our thought processes are molded by our language, and when we design digital systems we will use *and, or,* and *not* in the same sense as above. In this book, we denote the specific logical uses of *and, or,* and *not* by the symbols AND, OR, and NOT.

It is nearly always useful to formalize heavily used concepts; by so doing we achieve compactness and are able to handle more complexity than if we wrestle with informal concepts such as the normal English language uses of *and, or,* and *not. To* pave the way for a Boolean algebraic treatment of digital logic, we will formalize the concepts of logical constants, variables, and operators.

Logical Constants

The statement "There is a photodiode error" is either true or false. The operators AND, OR, and NOT are likewise concerned with two logical values, true and false.

We will concern ourselves only with logic systems that can be formalized with a binary set of logical constants, TRUE and FALSE. Since we use these concepts so heavily, it is worthwhile seeking abbreviations. We will represent TRUE by T or 1, and FALSE by F or 0. In this context 1 and 0 are not decimal numerical values; they are abbreviations for TRUE and FALSE, and nothing more. We will use 1 and T, 0 and F interchangeably in the text. Each abbreviation has its value, and both are widely used in digital design. In the study of hardware implementation of logic in Chapter 2, we will show a strong bias toward the T,F notation, to avoid a common point of confusion. On the other hand, in much of this chapter the 1,0 form for TRUE and FALSE is convenient. Be prepared to accept and use either form.

Logical Variables

Consider the declaration:

A = photodiode error

We use the *logical variable* A as an abbreviation for the cumbersome phrase "photodiode error" to achieve compactness. The variable A can have two values, T or F. If we do not have a photodiode error, then A = F; if we do, then A = T. Although it is possible to use single letters to represent logical elements, as we have above, it is usually better to use a more recognizable abbreviation, such as:

PDE = photodiode error

In general, the abbreviations should be a compromise between clarity and brevity. It is not really necessary to abbreviate at all. We could use PHOTODIODE.ERROR as the name of the logical variable, but it is too long to be convenient. A is short but conveys no meaning; PDE is a good compromise. We often use numbers in logical variables to indicate a particular member of a set of variables. A common unit of computer information is the 8-bit byte. If we needed to examine the individual bits of the byte, we might choose to assign distinct names to the bit values:

B7= left-most bit

.

.

.

B0=right-most bit

Other notations for sets of variables will suggest themselves. Instead of the distinct names for bits in the byte, we might choose to use a subscripted variable $B_7 \ldots B_0$ with equivalent meanings.

In this book we will capitalize the letters in the names of logical variables and will always start each name with a letter. To preserve the mnemonic value, names of variables may include periods as separators; GO.ON is an example.

Truth Tables

Consider a set of logical variables, each variable of which may have one of two values, T or F. In digital design we are interested in combining logical variables (e.g., using AND, OR, and NOT) to produce new variables that again have only two possible values, T or F, for any combination of given variable values. In other words, we wish to study binary functions of binary variables.

For a set of logical variables, we may define any desired function by giving the function value for each possible set of variable values. A tabular form with input variables on the left and the function on the right is useful for this display. For example, here is a logical function X of three variables A, B, and C, shown in both the 1,0 and the T,F notations.

C	A	B	X
0	0	0	0
0	0	1	1
1	0	0	1
1	0	1	0
0	1	0	1
0	1	1	0
1	1	0	0
1	1	1	0

A	C	B	X
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

Such a display is called *a truth table.* Having chosen an ordering of the input variables (A, C, B in this case), we list all possible combinations of the variables' values, in binary numeric order. Tabulation in this standard form is called a *canonical truth table.* "Canonical" means standard. For three variables, the canonical truth table has $2^3 = 8$ rows, arranged from binary 000 through binary 111. Since each binary bit pattern corresponds to a decimal number, we may describe a row of a canonical truth table by its decimal number equivalent. For example, the row corresponding to the variable values 0110 for a four-variable function may be called row 6. When convenient, you may write the row numbers on the left of the canonical truth table.

For canonical truth tables, we may compactly describe the function by a vector of function values. For example, the three-variable truth table for X above yields an eight-element vector

$$X(A,C,B) = (0,1,1,0,1,0,0,0)$$

Although we usually choose to list the values of variables in canonical order, any other order of rows displays the same information. The following two truth tables are equivalent, but the right-hand table lacks the useful uniformity of the canonical form on the left:

Row number	D	E	Y
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	0

Row number	D	E	Y
2	1	0	1
0	0	0	0
1	0	1	0
3	1	1	0

Logical Operators and Truth Tables

We will now give a precise definition of the three logical operators AND, OR, and NOT.

NOT. We represent logical NOT by the over-score. Thus, if PDE is a logical variable, then

$$\text{NOT (PDE)} = \overline{\text{PDE}}$$

In words, we would describe the notation $\overline{\text{PDE}}$ as "PDE not." We may define NOT by listing in a truth table all possible values for an arbitrary logical variable, and the corresponding values of the logical NOT of that variable. Since a logical variable A can have only two values, 1 and 0, the following list is exhaustive:

A	$\overline{A}$
0	1
1	0

We regard the formal definition of logical NOT to be given by its truth table. Remember that 1 and 0 represent TRUE and FALSE.

AND. We represent logical AND by a dot separating two logical variables we write B AND C as B•C. We shall faithfully use the "•" symbol to represent the AND operator even though some authors omit it when dealing with single letter logical variables. Thus, if you insist upon single-letter names, you might interpret BC as B•C. This is dangerous because we may want to name a single logical variable with the two-letter name BC. In real-world logic design, single letter names are not descriptive enough to be of use. There are only 26 possible single-letter names and a typical design will require many more than 26 names. We therefore give up the dubious advantage of having an implied AND for the real advantage of multi-letter names for logical variables.

We will define AND with a truth table. There are two independent variables in the logical AND, each of which can assume either of the two values, 1 and 0. Therefore, specifying the function value for each of the four combinations of inputs completely defines AND:

B	C	B•C
0	0	0
0	1	0
1	0	0
1	1	1

The table corresponds to our intuitive notion of AND in that B•C is true only if both B and C are simultaneously true.

Just as we may generalize the English use of *and* to encompass more than two variables, we can do so for the formal logical AND. The truth table for A•B•C is

A	B	C	A•B•C
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

For more than three variables, the truth table becomes unwieldy, and we revert to a verbal definition of the logical AND:

> The logical AND of several variables is true only when all the variables are simultaneously true.

OR. The symbol for logical OR is the + sign. Do not confuse this with the use of + in other contexts to represent arithmetic addition. Since in logic design the uses of logical OR will vastly outnumber the uses of an arithmetic plus, we choose a convenient single symbol for the OR operator and we use the more cumbersome word "plus" or the symbol "(+)" for an arithmetic plus. Here are two word statements translated into their corresponding logic design notations

A is true if B OR C is true	A=B + C
2 added to 3 is 5	5 = 2 plus 3, or 5 = 2 (+) 3

The defining truth table for a two-input logical OR is

B	C	B+C
0	0	0
0	1	1
1	0	1
1	1	1

As with the AND operator, we may generalize the definition of the logical OR to more than two input variables. In words, the output is true if at least one of the inputs is true. For instance,

AY	PDE	X	AY+PDE+X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

This completes our definition of AND, OR, and NOT. These logical operators operate on input variables to yield a single output. The NOT operates only on single variables, while AND and OR fundamentally operate on two inputs. For our convenience we may also think of AND and OR as multi-input operators. Truth tables are useful in many designs, but we need a more compact and powerful tool for representing logical manipulations. An algebra of logical operators, called *Boolean algebra* in honor of George Boole, who first explored the properties of the logical operators, is analogous to the familiar algebra of arithmetic operators. In the next section, we present some simple but important Boolean algebraic results.

ELEMENTS OF BOOLEAN ALGEBRA

Basic Manipulations

Boolean algebra is important to hardware designers because it allows the compact specification and simplification of logic formulas. Physical devices can perform the AND and OR functions, and it is this fact that raises Boolean algebra from the realm of interesting theory to the role of a vital design tool. The algebra may be developed from any of several starting points. Modern mathematicians derive Boolean algebra from a compact set of abstract postulates, producing an elegant and rigorous theory; however, in building a useful tool to assist structured digital design, we best achieve our goals by emphasizing the relationship of truth tables to logic equations. Truth tables and allied tabular displays play an important role in digital design and implementation. Therefore, we will assume as our starting point the existence of the two binary values TRUE (T or 1) and FALSE (F or 0), and the three operators AND, OR, and NOT, with behavior described by their truth tables.

Boolean algebraic formulas follow certain conventions. Our intuition tells us that we want the operators AND and OR to *commute* (e.g., $A + B = B + A$) and *associate* [e.g., $A + (B + C) = (A + B) + C$. The operators also *distribute* according to the relations:

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

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

The conventional hierarchy of operator action in complex expressions is

First: NOT
Then: AND
Last: OR

Our notation for logical NOT (the over score) explicitly shows the scope of action of the NOT operation, so the only possible confusion in evaluating expression would occur with AND and OR. As the hierarchy shows, AND takes precedence As an example, consider

$$X = \overline{A + B \bullet \overline{C}}$$

Evaluation is in the order specified below by the parentheses, innermost parenthesized expressions being evaluated first. Thus

$$X = \overline{((A + (B \bullet (\overline{C}))))}$$

Parentheses are useful in Boolean equations to override the normal hierarchy, just as we use them for similar purposes in conventional algebra.

Below are some fundamental relations of Boolean algebra; memorize these results.

$$\overline{\overline{A}} \equiv A \tag{1–1}$$

$$A \bullet T \equiv A \qquad A + F \equiv A \tag{1–2}$$

$$A \bullet F \equiv F \qquad A + T \equiv T \tag{1–3}$$

$$A \bullet A \equiv A \qquad A + A \equiv A \tag{1–4}$$

$$A \bullet \overline{A} \equiv F \qquad A + \overline{A} \equiv T \tag{1–5}$$

$$\overline{A \bullet B} \equiv \overline{A} + \overline{B} \qquad \overline{A + B} \equiv \overline{A} \bullet \overline{B} \tag{1–6}$$

Each identity involving AND or OR operators comes in two forms, one emphasizing AND, the other emphasizing OR. This *principle of duality* is a characteristic of Boolean algebra, and it has important applications in the study of logic and in the implementation of logic functions with physical devices. The dual identities are related by this rule:

Change each AND to OR, each OR to AND, each T to F, and each F to T.

Equation (1–6) is the well-known *De Morgan's law.* It is of special importance because it allows us to convert Boolean operators from AND to OR, and vice versa.

You may prove each of the foregoing identities by using the truth-table definitions of the logical operators. To illustrate the art of proving theorems with truth tables, we will prove the validity of De Morgan's law. Start with the form $\overline{A \bullet B} \equiv \overline{A} + \overline{B}$. Develop truth tables for the left-hand side and for the right-hand side of the identity. (When convenient, we may show several functions [outputs] in the same table: we write two or more truth tables in one package.)

A	B	$A \bullet B$	$\overline{A \bullet B}$	A	B	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
F	F	F	T	F	F	T	T	T
F	T	F	T	F	T	T	F	T
T	F	F	T	T	F	F	T	T
T	T	T	F	T	T	F	F	F

A truth table is an exhaustive list of function values for each possible combination of inputs; therefore, if two truth tables have identical rows, the functions behave identically. You see that the truth table for $\overline{A \bullet B}$ is the same as that for $\overline{A} + \overline{B}$; this proves Eq. (1–6).

De Morgan's law extends to more than two variables. For example, the following identities are valid.

$$\overline{A + B + C} \equiv \overline{A} \bullet \overline{B} \bullet \overline{C} \qquad\qquad \overline{A \bullet B \bullet C} \equiv \overline{A} + \overline{B} + \overline{C}$$

Several other Boolean identities find frequent use in our design work. You may demonstrate each relationship using truth tables or using the previous Boolean identity.

$$A + A \bullet B \equiv A \qquad A \bullet (A + B) \equiv A \qquad (1\text{–}7)$$

$$A + \overline{A} \bullet B \equiv A + B \qquad A \bullet (\overline{A} + B) \equiv A \bullet B \qquad (1\text{–}8)$$

$$A \bullet B + \overline{A} \bullet B \equiv B \qquad (1\text{–}9)$$

The left-hand form of Eq. (1–8) is not immediately obvious, but it is of great help in reducing the complexity of commonly occurring Boolean expressions. After De Morgan's law, Eq. (1–9) is perhaps the most widely used relation. It is the basis for several systematic Boolean simplification procedures. Presently, we will develop the one simplification method, Karnaugh maps, that will be of most benefit to our design work.

Truth tables serve as an easy means of verifying the validity of small Boolean equations, whereas the Boolean identities presented above are useful in manipulating both large and small Boolean equations. Here is an example of Boolean algebraic manipulations.

$$\overline{A \bullet (B + C \bullet (B + \overline{A}))} \equiv \overline{A \bullet (B + C \bullet B + C \bullet \overline{A})} \qquad \text{(distribution law)}$$

$$\equiv \overline{A \bullet (B + C \bullet \overline{A})} \qquad \text{[Eq. (1–7)]}$$

$$\equiv \overline{A \bullet B + A \bullet (C \bullet \overline{A})} \qquad \text{(distribution law)}$$

$$\equiv \overline{A \bullet B + A \bullet (\overline{A} \bullet C)} \qquad \text{(commutation law)}$$

$$\equiv \overline{A \bullet B + (A \bullet \overline{A}) \bullet C} \qquad \text{(association law)}$$

$$\equiv \overline{A \bullet B + F \bullet C} \qquad \text{[Eq. (1–5)]}$$

$$\equiv \overline{A \bullet B} \qquad \text{[Eqs. (1–3) and (1–2)]}$$

$$\equiv \overline{A} + \overline{B} \qquad \text{(De Morgan's law)}$$

Equations from Truth Tables

If the truth table and logic equation are to work hand in hand as design aids, we must be able to derive a logic equation for a function from its truth table. Consider this example:

Row number	A	B	W
0	F	F	F
1	F	T	F
2	T	F	T
3	T	T	F

In words, W is true only if A is true and B is false (from row 2). More formally, $W = A \bullet \overline{B}$. Another example is

Row number	A	B	Y
0	F	F	T
1	F	T	F
2	T	F	T
3	T	T	T

Here, our intuitive understanding of the truth table is that Y is true whenever A is false and B is false (row 0), or whenever A is true and B is false (row 2), or whenever A is true and B is true (row 3). Thus

$$Y = \overline{A} \bullet \overline{B} + A \bullet \overline{B} + A \bullet B \qquad (1–10)$$

Incidentally, we may simplify this equation:

$$Y = \overline{B} + A \bullet B \qquad \text{[by Eq. (1–9)]} \qquad (1–11)$$

$$= \overline{B} + A \qquad \text{[by Eq. (1–8)]}$$

Viewing this truth table another way, we may say that Y is *false* only when A is false and B is true (row 1):

$$\overline{Y} = \overline{A} \bullet B \qquad (1–12)$$

We have two equations for Y derived from the same truth table–Eqs. (1–10) and (1–12). Can you show that the expressions for Y are equivalent?

Which way is best for writing equations from truth tables–reading true conditions for the function, or reading false conditions? Both ways result in equivalent expressions, usually in somewhat different form. For equations of more than two variables, when there are many rows in the truth table, you will usually wish to use the method that involves the fewer AND terms. In this example, Eq. (1–12), derived from the false function values, yields the more direct and simple result.

Sum-of-products form. Equation (1–10) (and also Eq. [1–11]) expresses the function *Y* in the *sum-of-products form.* This is the most common form for deriving Boolean equations from truth tables, and in this context the form fits nicely with our thought processes. The name "sum of products" comes from an analogy of the Boolean operator symbols with those of arithmetic: the expression is a sum (OR) of product (AND) terms.

In sum-of-products form, a product term consists of the logical AND of a set of operands, each operand being a logic variable or its negation. (A trivial form of product term consists of a single variable or its negation.) A variable's name must appear at most once in a product term. For example, $\overline{A}$, $A \bullet \overline{B}$, and $\overline{A} \bullet B \bullet \overline{C}$ are valid product terms, whereas $A \bullet \overline{A}$ and $\overline{A} \bullet B \bullet B \bullet C$, although valid Boolean expressions, are not proper product terms.

A product term containing exactly one occurrence of every variable (either asserted or negated) is called *a minterm* or *a canonical product term.* A function expressed as a logical OR of distinct minterms is in *canonical sum-of-products form* or *disjunctive normal form.* Our intuitive method for deriving equations from truth tables yields the canonical sum-of-products form, as in Eq. (1–10). An expression may be in sum-of-products form yet not be canonical. Equation (1–11) is a noncanonical sum-of-products expression.

We may now state formal prescriptions for deriving sum-of-products logic equations from canonical truth tables:

> To derive a sum-of-products form for a function from a canonical truth table, write the OR (sum) of the minterms for which the function is true. Similarly, to derive a sum-of-products form for the complement of a function from a canonical truth table, write the OR of the minterms for which the function is false.

Applying these rules to the previous truth table yields Eqs. (1–10) and (1–12).

For n-variable functions, there are 2^n possible minterms. A minterm is sometimes designated by $\mathbf{m_i}$, where *i* is the number of the single canonical truth table row for which the minterm yields truth. For example, $\mathbf{m_4} = A \bullet \overline{B} \bullet \overline{C}$ yields truth only for variable values A=1, B=0, C=0; this corresponds to row 4, since binary 100 is decimal 4. Again, $\mathbf{m_2} = \overline{A} \bullet B \bullet \overline{C}$ produces truth only for row 2 (binary 010). (The name *minterm* denotes that the term is true for only one row of the table.) With this notation, we may describe canonical sum-of-products equations as sums of minterms. Equation (1–10) becomes Y = $\mathbf{m_0} + \mathbf{m_2} + \mathbf{m_3}$. Although this notation is important in certain developments of Boolean algebra, we will not use it frequently in this book.

Equation (1–12) is a canonical sum-of-products equation for $\overline{Y}$, albeit a somewhat trivial form containing only one term. We may use the minterm notation for this form also:

$$\overline{Y} = \mathbf{m_1}$$

Product-of-sums form. There is another formulation of logic expressions from truth tables: the *product-of-sums form.* This form consists of the AND (the product) of a set of OR terms (the sums), such as $(\overline{A}+B)\bullet(A+B)\bullet(\overline{B})$. In a product-of-sums expression, a sum term consists of the logical OR of a set of operands, each operand being a logic variable or its negation, and each variable appearing at most once.

A sum term that contains exactly one occurrence of every variable (either asserted or negated) is called a *maxterm* or *canonical sum term.* A function expressed as a logical AND of distinct maxterms is in *canonical product-of-sums form* or *conjunctive normal form.* The product-of-sums notation is not much used in practical design, since it lacks the sum-of-products' easy kinship with our thought processes. An expression may be in product-of-sums form yet not be canonical, if one or more of the sum terms is not a maxterm. For a three-variable function,

> $A+\overline{B}+C$ is a maxterm; $\overline{B}+\overline{C}$ is not.
>
> $W = (P+\overline{Q}+\overline{R})\bullet(P+Q+R)$ is in canonical product-of-sums form;
>
> $W = (P+Q)\bullet(P+Q+R)$ is not canonical.

The prescriptions for forming product-of-sums logic equations from truth tables are:

> To derive a product-of-sums form of a function from a canonical truth table, write the AND (product) of each maxterm for which the function is false. Similarly, to derive a product-of-sums form for the complement of a function, write the AND of each maxterm for which the function is true.

Applying these rules to the previous truth table, we have

$$Y = (A+\overline{B})$$

$$\overline{Y} = (A+B)\bullet(\overline{A}+B)\bullet(\overline{A}+\overline{B}) \qquad (1\text{–}13)$$

The first equation agrees with Eq. (1–11), obtained by simplifying the sum-of-products form in Eq. (1–10). With the aid of the distributive law, we may simplify Eq. (1–13) to yield

$$\overline{Y} = \overline{A}\bullet B$$

in agreement with Eq. (1–12).

A maxterm is true for every row of the canonical truth table except one; we sometimes specify the maxterm by $\mathbf{M_i}$, where i is the row number for which the maxterm is false. For example, $(\overline{A}+B+\overline{C})$is true for every combination of values of variable except A = 1, B = 0, C = 1, which designates row 5 (binary 101 = decimal 5). Maxterm $\mathbf{M_0}$ is (A + B + C), since only for variable values 000 does the term produce a false value. The name *maxterm* connotes that the term is true for all but one set of variable values. We occasionally write canonical product-of-sums expressions, analogous to the sum-of-products formalism, as products of maxterms. For instance, the products of sums above become

$$Y = (A+\overline{B}) = M_1$$

$$Y = (A+B)\bullet(\overline{A}+B)\bullet(\overline{A}+\overline{B}) = M_0\bullet M_2\bullet M_3$$

To illustrate the four rules for producing canonical equations, we derive the canonical equations for a function W of three variables:

Row number	J	K	L	W
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Sum of products on true outputs:

$$W = \overline{J}\bullet\overline{K}\bullet L + \overline{J}\bullet K\bullet\overline{L} + \overline{J}\bullet K\bullet L \tag{1–14}$$

Sum of products on false outputs:

$$\overline{W} = \overline{J}\bullet\overline{K}\bullet\overline{L} + J\bullet\overline{K}\bullet\overline{L} + J\bullet\overline{K}\bullet L + J\bullet K\bullet\overline{L} + J\bullet K\bullet L \tag{1–15}$$

Product of sums on false outputs

$$W = (J+K+L)\bullet(\overline{J}+K+L)\bullet(\overline{J}+K+\overline{L})\bullet(\overline{J}+\overline{K}+L)\bullet(\overline{J}+\overline{K}+\overline{L}) \tag{1–16}$$

Product of sums on true outputs:

$$\overline{W} = (J+K+\overline{L})\bullet(J+\overline{K}+L)\bullet(J+\overline{K}+\overline{L}) \tag{1–17}$$

You should simplify each equation, using algebraic identities, and verify that the equations are equivalent.

Truth Tables from Equations

Sometimes you will wish to convert a logic expression into its truth-table form. If the expression is in sum-of-products form, the conversion is easy. Each product term will form one or more truth-table rows having, a true function value. A canonical product term (one with all the variables in it; a minterm) produces one row with an output of TRUE. A product term with fewer variables yields more rows, since such a term is true for *any* values of the missing variables. Often more than one product term in the sum will contribute truth for a given row of the truth table. This is fine, double truth is still truth!

As an example, consider this equation of three variables:

$$Y = \underset{\text{Term1}}{J \bullet \overline{K}} \;+\; \underset{\text{Term2}}{\overline{J} \bullet K \bullet L} \;+\; \underset{\text{Term3}}{J \bullet K \bullet \overline{L}} \;+\; \underset{\text{Term4}}{K \bullet L}$$

The truth table for this equation is

J	K	L	Y		
0	0	0	0		
0	0	1	0		
0	1	0	0		
0	1	1	1	(Terms 2 and 4)	(1–19)
1	0	0	1	(Term 1)	
1	0	1	1	(Term 1)	
1	1	0	1	(Term 3)	
1	1	1	1	(Term 4)	

Terms 2 and 3 are canonical; each contributes a true output to one row of the table. Terms 1 and 4, having a missing variable, contribute true outputs to two rows each.

Equations in product-of-sums form are most easily translated into truth tables by focusing on the conditions for having false function values. Each sum term in the product will assure a false expression value whenever *all* its variables are the opposite of the form in the term. For example, consider the following equation of three variables:

$$G = \underset{\text{Term1}}{(\overline{A} + B + C)} \bullet \underset{\text{Term2}}{(\overline{A} + B)} \bullet \underset{\text{Term3}}{(\overline{A} + \overline{B} + \overline{C})}$$

Term 1 makes G false for row 4 (100) of the truth table; term 3 produces a result of false for row 7 (111). Term 2, with its missing variable C, produces a result of false for two rows, 4 (100) and 5 (101). Terms 1 and 3 are canonical: each contributes a false function value for one row; term 2 is not canonical.
Here is the truth table:

Row number	A	B	C	Term 1	Term 2	Term 3	G
0	0	0	0	1	1	1	1
1	0	0	1	1	1	1	1
2	0	1	0	1	1	1	1
3	0	1	1	1	1	1	1
4	1	0	0	0	0	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	1	1
7	1	1	1	1	1	0	0

For logic expressions of more general form than the sum of products or the product of sums, we fall back on the ultimate method of deriving truth tables: evaluating the function for every combination of values of the input variables.

This means that we explicitly determine each function value. The process is often laborious, but (barring error!) is foolproof. For example, from the equation

$$K = \overline{(L + G)} \bullet (\overline{L} + G)$$

We get the following truth table by computing the value of K for each of the four sets of values of L and G:

L	G	K
F	F	T
F	T	F
T	F	F
T	T	F

Another approach for forming a truth table from a general Boolean equation is to manipulate the expression (usually using De Morgan's law) until it becomes a sum-of-products or product-of-sums form, and then use the methods presented earlier in this section.

Condensing Truth Tables

A canonical truth table of n input variables has 2^n rows arranged in binary numerical order on its inputs, corresponding to all the possible values of the input variables. The canonical form explicitly displays the function's value for every possible set of input conditions. This form of truth table—the only one we have used so far—is the counterpart of the canonical forms of sum-of-products and product-of-sums equations. Just as we frequently use Boolean equations in a simplified form, we also sometimes wish to deal with a simplified or collapsed truth-table notation.

Consider Eq. (1–18) again:

$$Y = \underset{\text{Term1}}{J \bullet \overline{K}} + \underset{\text{Term2}}{\overline{J} \bullet K \bullet L} + \underset{\text{Term3}}{J \bullet K \bullet \overline{L}} + \underset{\text{Term4}}{K \bullet L}$$

The canonical terms (2 and 3) each contribute one row to the canonical truth table in Eq. (1–19). Term 1, however, is independent of the value of variable L (L does not appear), so term 1 contributes two rows with true output to the canonical truth table, one for each value of the missing variable. If we are willing to abandon the canonical form for the truth table, we may introduce a shorthand notation for this situation. We collapse these two rows for term 1 into a single row and place an X for the value of the missing variable L. The X means "both values" and implies that the function value is independent of that variable whenever the other inputs are in their stated conditions. Similar arguments apply to term 4, which lacks the variable J.

Applying this concept to the expression, we may derive a shortened truth table

J	K	L	Y	
0	0	0	0	
0	0	1	0	
0	1	0	0	
X	1	1	1	(Terms 2 and 4)
1	0	X	1	(Term 1)
1	1	0	1	(Term 3)

(1–20)

Note how this truth table yields the original equation in a direct manner.

The "X" is the truth-table equivalent of the important Boolean algebraic identity of Eq. (1–9)

$$A \bullet B + \overline{A} \bullet B \equiv B$$

There are various applications of this identity that we could introduce directly into the truth table of Eq. (1–20) if we desired. For instance, here are two more stages in the condensation of this table:

J	K	L	Y
0	0	X	0
0	1	0	0
X	1	1	1
1	0	X	1
1	1	0	1

J	K	L	Y
0	0	X	0
0	X	0	0
X	1	1	1
1	X	X	1

(1–21)

If we are presented with a truth table containing **X**'s, the derived Boolean equation will not be canonical, but will contain some simplified terms. A sum-of-products equation for the right-hand truth table in Eq. (1–21) is

$$Y = K \bullet L + J$$

Satisfy yourself that the original Eq. (1–18) is equivalent to this, and that all these truth tables based on Eq. (1–18) are equivalent.

Condensed tables are convenient, since the original statement of a problem will often lead in a natural way to the condensing of rows. The main virtue of the notation is in allowing the truth table to reflect its origins more faithfully. A secondary virtue is in the resultant shortening. Attempts, as in Eq. (1–21), to simplify truth tables by collapsing rows of a less simplified form are tricky and can lead the inexperienced into errors. Soon we will discuss a graphical method for simplifying logic functions that is easier for people to use.

Don't-Care Outputs in Truth Tables

Truth tables have a useful property that a logic equation cannot express. We often know from the nature of the problem that the function's value is irrelevant for certain combinations of the input variables. This situation usually arises when we know that the inputs will never legitimately assume certain sets of values. We may use a small dash "–" for the truth-table function value in such cases. The "–" means "don't care." In deriving a logic equation from the truth table, we are free to use either a T or F value for the don't-care dash; whichever will yield the more useful form. For instance, look at the condensed truth table below:

A	B	Y
F	X	T
T	F	–
T	T	F

Of the various equations that we may derive from this truth table, a choice of T for the don't-care output might yield the sum-of-products form

$$Y = \overline{A} + A \bullet \overline{B}$$

which can be simplified to

$$Y = \overline{A} + \overline{B}$$

whereas a choice of F for the don't-care gives the quite different equation

$$Y = \overline{A}$$

You may use either equation; your choice may depend on other factors in the problem design.

KARNAUGH MAPS

In the early years of digital computers, each logic element in a circuit was large and cumbersome by modern standards, and consumed considerable power. There was a natural emphasis on reducing the number of circuit elements to the bare minimum so as to cut the total cost. Designers developed many elaborate techniques for simplifying logic expressions, and much effort went into perfecting these tools. Today, hardware for digital logic is inexpensive, and in modern design work the emphasis on circuit minimization has given way to a concern for modularity and clarity in the design process.

One result of this shift in technology and design style is that circuit building blocks have become larger and more powerful, while the "glue" that holds them together (the logic equations) has become simpler and less voluminous. Although the minimization of complex Boolean equations is no longer of paramount importance, simplification of small and manageable equations remains a routine task that we should make as easy and mechanical as practical. Manipulating equations through the Boolean algebraic identities, as we have done in the previous sections, is an arcane art. There are few guidelines to follow other than our intuition (based on experience) and trial and error. You have seen that sum-of-products is a common form of Boolean expression. This form results from truth-table derivations and occurs in other steps of our design process. The Boolean identity that is of most frequent use in simplifying sum-of-products forms is Eq. (1–9), which allows the elimination of a variable and its complement when these have a common factor:

$$A \bullet B + \overline{A} \bullet B = (A + \overline{A}) \bullet B = T \bullet B = B$$

The Karnaugh map (commonly called a K-map, and also known as a Vietch diagram) is a graphic display whose visual impact assists us in the systematic application of this identity. A Karnaugh map is a canonical truth table rearranged in form. Figure 1–1 is a truth table and its K-map for an arbitrary function of two variables. The Karnaugh map has a square for each truth-table row; each combination of variables identifies a square in the map. In the two-variable case, the values of one variable appear as the labels for the columns, (B = 0 and B = 1 in Fig. 1–1), and the other variable's values mark the rows, (A = 0 and A = 1). Each square contains the value of the given function (Y_i) corresponding to the appropriate truth-table row, as specified by the labels on the edges of the K-map. For instance, the lower left square in the K-map of Fig. 1–1, corresponding to A = 1, B = 0, has the function value Y_2

Row number	A	B	Y
0	0	0	Y_0
1	0	1	Y_1
2	1	0	Y_2
3	1	1	Y_3

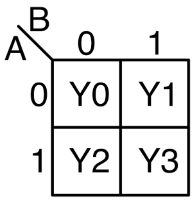


Figure 1–1. A Truth Table and its Karnaugh map

Functions of more than two variables also have K-map representations. Three and four-variable functions are easy to manage; with more than four variables, the K-map technique becomes unwieldy, but fortunately most of the simplifications of design equations that we encounter in practice involve no more than four variables. The three-variable map contains eight squares, corresponding to the eight rows in the canonical truth table. Figure 1-2 shows two notations for K-maps of three variables. Both forms are equivalent, and both are in common use. Our mild preference is for the left-hand form, but you should be familiar with each. We like the left-hand form because it has an explicit display of the values of the variables on each edge of the map. The right-hand form explicitly labels only the location of *true* values for each variable. Some people prefer this form; take your pick. Conventionally, the first variable (lefttmost in the truth table) appears on the vertical edge, while the horizontal edge displays the second and third variables

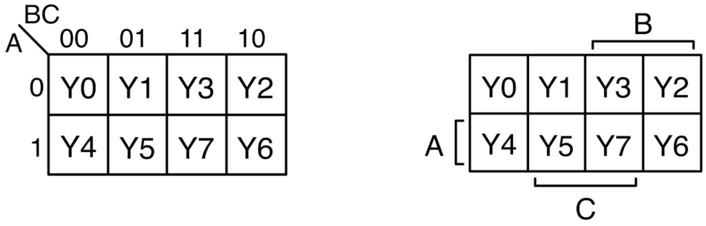


Figure 1–2. Two forms of a K-map of three variables

Note carefully the order of the labels on the top edge. In moving from square to square across a row (and around the corner, also), the value of only one variable changes at a time. As you will see, this *unit distance* property gives the K-map its virtue in simplifying logic expressions.

Extending the K-map to four variables adds an additional variable to the vertical edge, resulting in 16 squares. Figure 1–3 illustrates both notations for Karnaugh maps of four variables.

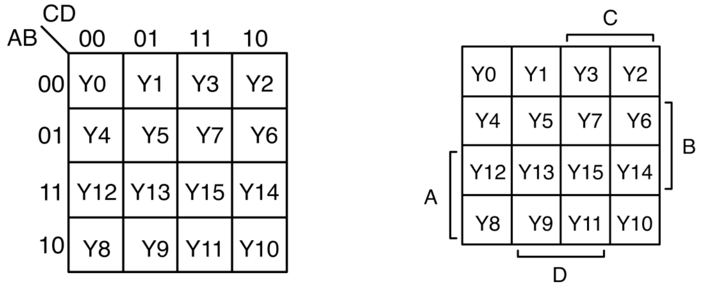


Figure 1–3. Two forms of a K-map of four variables

Building K-Maps

There is a one-to-one correspondence between Karnaugh map squares and canonical truth-table rows. Deriving either the map or the table from the other is just a matter of rearranging the information. Don't-care outputs from truth tables are directly transferable to the appropriate K-map squares. Condensed truth-table rows yield values for more than one K-map square, in an obvious way.

We may view the K-map as a representation of the canonical sum-of-products form of a Boolean expression. Just as we may derive a truth table from a logic equation, we may move directly to a K-map from an equation, when this is appropriate. Figure 1–4 shows the three forms for a function of two variables.

The techniques for creating a truth table from a logic equation will also yield the K-map for the equation. The expression in Fig. 1–4 is already in canonical form, so the transformations among table, K-map, and equation are easy.

$$X = \overline{A} \bullet \overline{B} + A \bullet \overline{B}$$

A	B	X
0	0	1
0	1	0
1	0	1
1	1	0

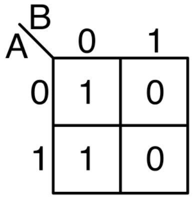
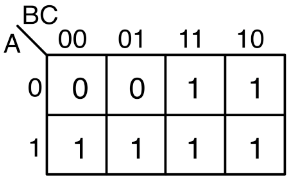


Figure 1–4. Three representations of a Boolean function

Consider now the three-variable equation

$$V = A \bullet \overline{B} + B + A \bullet \overline{B} \bullet C$$

The first term in the sum yields l's in the K-map when A,B = 10 and C = anything. To give a true value, the second term requires B = 1, but A and C may be anything. The third term yields 1 for A,B,C = 101, which already appears in the map because of the A•B term. The resulting map is Fig. 1-5.



A \ BC	00	01	11	10
0	0	0	1	1
1	1	1	1	1

Figure 1–5. K-map for $V = A \bullet \overline{B} + B + A \bullet \overline{B} \bullet C$

Simplifying with K-Maps

Why bother with these maps? You may get a clue from Fig. 1–4. You have probably noticed that the expression for X can be simplified with Eq. (1–9):

$$\begin{aligned} X &= A \bullet \overline{B} + \overline{A} \bullet \overline{B} \\ &= (A + \overline{A}) \bullet \overline{B} \\ &= T \bullet \overline{B} \\ &= \overline{B} \end{aligned}$$

How does the K-map display this simplification? The key point is that a certain term ($\overline{B}$ in this case) is ANDed with both A and $\overline{A}$. On the K-map this results in l's in both the A = 0 and A = 1 squares for B = 0. Let's circle these adjacent l's to remind us that the A variable disappears from the simplified expression because it appears as both A and $\overline{A}$ (see Fig. 1–6). Note how the simplified form A = $\overline{B}$ stands out more clearly: the condition for X to be true is that B is false (or $\overline{B}$ is true). Thus X = $\overline{B}$.

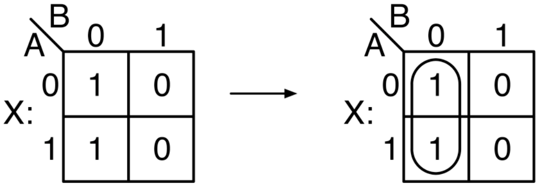


Figure 1–6. Circling adjacent 1’s

The drawing of circles (really ovals) among adjacent l's is the basis for using K-maps in Boolean simplification. On K-maps of two variables, there are five ways to display applications of the basic identity of Eq. (1–9) with circles. Figure 1–7 shows these forms.

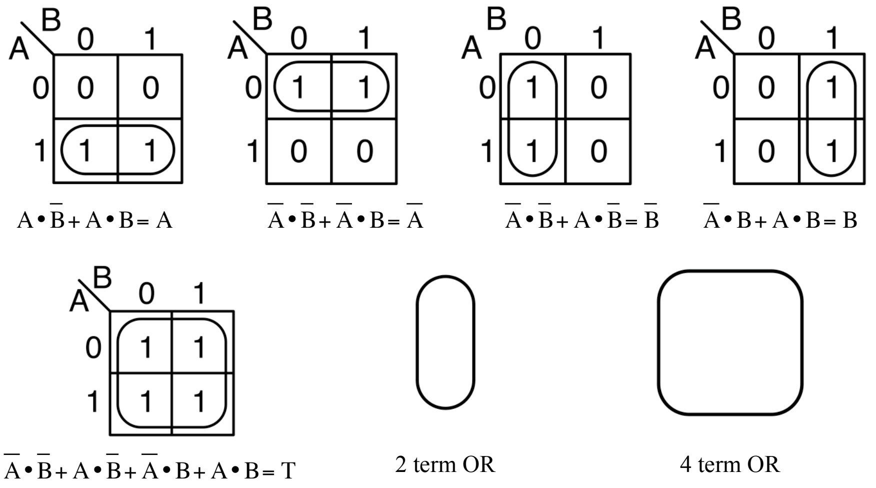

A
B
0
1
$A\bullet\overline{B}+A\bullet B=A$
$\overline{A}\bullet\overline{B}+\overline{A}\bullet B=\overline{A}$
$\overline{A}\bullet\overline{B}+A\bullet\overline{B}=\overline{B}$
$\overline{A}\bullet B+A\bullet B=B$
$\overline{A}\bullet\overline{B}+A\bullet\overline{B}+\overline{A}\bullet B+A\bullet B=T$
2 term OR
4 term OR

Figure 1–7. Simplifications for K-maps of 2 variables

In using the K-map for simplification, we look for applications of the rules for circling. Depending on the position of the l's, we may have several circles, each spanning a grouping of one, two, four, eight.... 1's. The K-map method requires that each 1 in the map appear in at least one circle, even if it is by itself. Circling two l's causes two canonical terms to collapse into one term; one variable drops out. Four circled l's bring four terms into one term, eliminating two variables. A proper group of eight circled l's drops three variables, and so on.

On K-maps of three or more variables, some applications of the simplifying identity do not involve physically adjacent l's. In these cases, we must draw "around-the-corner" circles. Figure 1–8a shows some typical ordinary circle patterns for a K-map of three variables; Fig. 1–8b gives all the around-the-corner patterns of two 1's; and Fig. l-8c shows the only around-the-corner pattern involving four l's.

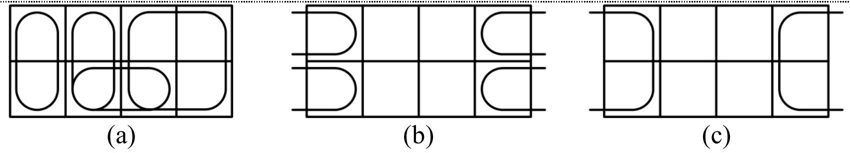

(a) (b) (c)

Figure 1–8. Typical circlings for 3-variable K-maps

Figure 1–9 shows three improper circlings. Diagonal or L-shaped arrangements do not correspond to applications of Eq. (1–9), nor does the circling of three 1's, since 3 is not a power of 2.

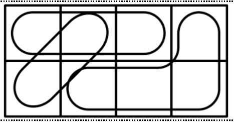


Figure 1–9. Improper K-map circlings

K-maps of four variables are similar to the three-variable variety. Figure 1–10 shows some forms involving correct circlings of four and eight 1's. You should inspect these patterns until you are comfortable with their meaning.

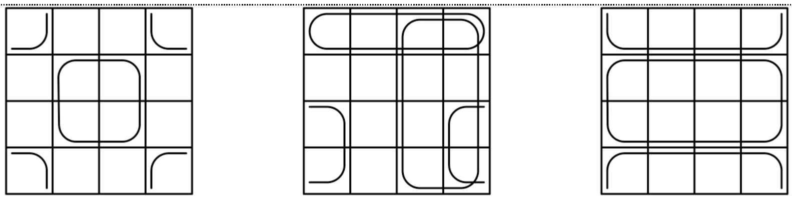
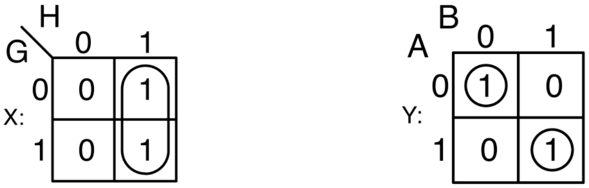


Figure 1–10. Typical 4&8 circlings on 4-variable K-maps

Here is the prescription for circling l's in a K-map: Draw circles (ovals or around-the-corner patterns) around properly positioned collections of l's, starting with the largest possible circles, and working toward smaller circles. Overlapping circles are appropriate when they allow a larger circle to appear. (Do not draw a circle that is completely within a larger circle; this would result in a redundant term and an incompletely simplified function.) Drawing the largest circles possible, cover all the l's on the map. Use don't-care dashes "–" as either a 1 or a 0, as convenient. The point of using K-maps is to let the drawing display the simplified result in a systematic and mechanical fashion. When you have finished drawing circles, read off the simplified function as a sum of products, in which each circle contributes one product term to the sum.

Figure 1–11 shows two examples of functions of two variables, derived from their K-maps. Figure 1–11b yields no simplification.



$X = H$

$Y = \overline{A} \bullet \overline{B} + A \bullet B$

(a) Simplifies

(b) Does not simplify

Figure 1–11. Simplifiable and unsimplifiable functions of 2 variables

The K-map method allows a simple derivation of the important identity of Eq. (1–8); Fig. 1–12 shows the process.

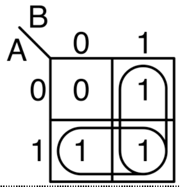
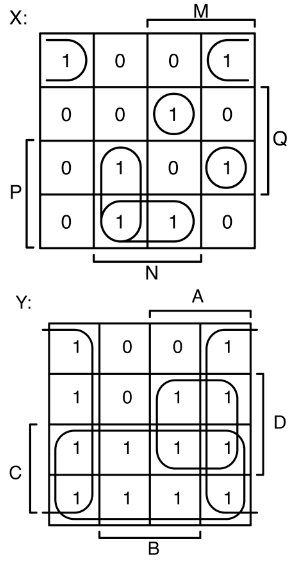


Figure 1–12. Proof of identity $A + \overline{A} \bullet B = A + B$

Figure 1–13 shows the simplifications of two functions of four variables. Notice the use of overlapping circles to achieve the largest circles. Some l's must be circled by themselves, yielding unsimplified four-variable terms.



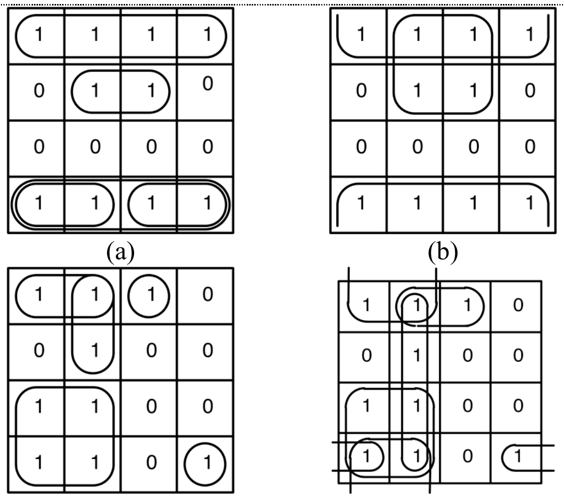
$$X = M\bullet N\bullet \overline{P}\bullet Q + M\bullet \overline{N}\bullet P\bullet Q$$
$$+ \overline{N}\bullet \overline{P}\bullet \overline{Q} + \overline{M}\bullet N\bullet P + N\bullet P\bullet \overline{Q}$$

$$Y = \overline{B} + C + A\bullet D$$

Figure 1–13. Simplification of two functions of four variables

K-Map Simplification Blunders

The most common error in simplifying expressions with K-maps is to fail to circle the largest possible groupings of l's. A less common error is to introduce a redundant smaller circle within a larger one. Figure 1–14 shows some typical blunders and the correct forms. Although Figs. 1–14a and 1–14c produce correct Boolean expressions, these expressions are not as simple as the K-map method allows. Design criteria may sometimes require the use of an incompletely simplified form, but these occasions are rare and we will not consider them here. Make all your circles as large as possible. Figure 1–14a has four circling errors: two incomplete circlings and two redundant circles. Figure 1–14c has three errors, all incomplete circlings.



(a)

1	1	1	1
0	1	1	0
0	0	0	0
1	1	1	1

(b)

1	1	1	1
0	1	1	0
0	0	0	0
1	1	1	1

(c)

1	1	1	0
0	1	0	0
1	1	0	0
1	1	0	1

(d)

1	1	1	0
0	1	0	0
1	1	0	0
1	1	0	1

Figure 1–14. Common blunders in circling K-maps

Other Ways of Reading K-Maps

We have stressed the method of circling l's and reading a sum of products for the function. If you are interested in developing the best facility with K-maps, you will want to investigate three other interpretations. These are analogous to the forms for reading expressions from truth tables shown earlier in this chapter. We give the three additional K-map methods below, with one example, leaving a comprehensive study of these techniques as grist for your mental mill.

(Hint: sums naturally emphasize 1's, products emphasize 0's.)

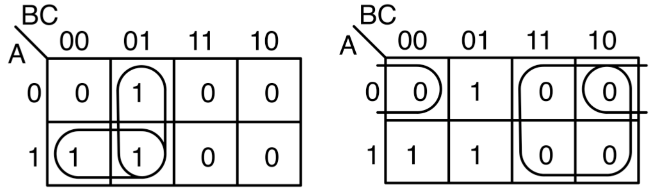
Method 1: Circle l's and read a sum of products for the function (the normal method).

Method 2: Circle 0's and read a sum of products for the inverse of the function (also a frequently used method).

Method 3: Circle 0's and show a way to generate a product of sums for the function.

Method 4: Circle l's and show a way to generate a product of sums for the inverse of the function.

Figure 1–15a shows a K-map with l's circled; Fig. 1–15b is the same map with 0's circled. The resulting equivalent simplified forms of the function are:



(a) Circling 1's (b) Circling 0's

Figure 1-15.

From method 1: $S = A \bullet \overline{B} + \overline{B} \bullet C$

From method 2: $\overline{S} = B + \overline{A} \bullet \overline{C}$

From method 3: $S = \overline{B} \bullet (A + C)$

From method 4: $\overline{S} = (B + \overline{A}) \bullet (B + \overline{C})$

CONCLUSION

You now have the knowledge of the foundations of digital logic that you need to continue your introduction to digital design. Boolean algebra and its allied techniques are a fascinating field of study, and you may wish to pursue these topics in more depth as your skill as a designer grows. We have just scratched the surface of the field, but the information in this chapter is sufficient for our purposes. Going deeper would deflect us from our goal, which is to build up the necessary tools as rapidly as possible in so you may quickly reach the study of the design process in later parts of the book.

Now it is time to develop tools for systematically translating logic expressions into hardware.

READINGS AND SOURCES

BLAKESLEE, THOMAS R., *Digital Design with Standard MSI and LSI,* 2nd ed. John Wiley& Sons, New York, 1979.

BOOLE, GEORGE, *An Investigation of the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probability,* 1849. Dover Publications, 31 East 2nd Street, Mineola, N.Y. 11501, 1954. Where it all started. A Dover reprint of the classic work.

DIETMEYER, DONALD L., *Logic Design of Digital Systems,* 2nd ed. Allyn & Bacon, Boston, 1978. Good exposition of traditional switching theory and minimizations.

FLETCHER, WILLIAM I., *An Engineering Approach to Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980. Chapter 3 contains a good exposition of Karnaugh maps and minimization.

HILL, FREDERICK J., and GERALD R. PETERSON, *Introduction to Switching Theory and Logical Design,* 3rd ed. John Wiley & Sons, New York, 1981.

KARNAUGH, M., "The map method for synthesis of combinational logic circuits," *Communications and Electronics,* No. 9, November 1953.

MILLER, RAYMOND E., *Switching Theory, Vol. 1: Combinational Circuits.* John Wiley & Sons, New York, 1966. An important early work.

SHANNON, C. E., "Symbolic analysis of relay and switching circuits," *Transactions AIEE,* Vol. 57, 1938, p. 713. Shannon's thesis has been called "possibly the most important and also the most famous master's thesis of the century"

VEITCH, E. W., "A chart method for simplifying truth functions," *Proceedings ACM,* Pittsburgh, 1952, p. 127.

EXERCISES

1-1. Read the preface.

1-2. Look up Boole, Karnaugh, and Shannon in (www.wikipedia.com)

1-3. What are the basic logical operators in digital design? What are the constants? What does the numeral 1 mean in digital design

1-4. How many different Boolean functions of two variables are there? Of three variables? Derive an expression for the number of different Boolean functions of *n* variables.

1-5. What is a canonical truth table? Give examples of a canonical truth table and a non-canonical truth table of three variables.

1-6. Give the operator hierarchies for AND, OR, and NOT. By inserting full parenthesis, show the order of evaluation of these functions:

(a) $\mathrm{B} \bullet \overline{\mathrm{A} \bullet \mathrm{C}} + \mathrm{D} + \overline{\mathrm{E}}$

(b) $\overline{\mathrm{A} + \mathrm{B}} \bullet \mathrm{C} + \mathrm{D}$

(c) $\overline{\mathrm{A} + \mathrm{B}} \bullet (\overline{\mathrm{C}} + \mathrm{D})$

1-7. By using the operator hierarchies, write the following expressions with as few parentheses as possible.

(a) $\overline{((\mathrm{Q} + (\mathrm{R} \bullet \mathrm{S}))} + \mathrm{U} \bullet \mathrm{V})$

(b) $((\mathrm{Q} \bullet (\overline{\mathrm{R}} + \mathrm{S})) \bullet (\mathrm{U} + \mathrm{V}))$

1-8. Prove the following identities by writing truth tables for both sides.

(a) $\mathrm{A} \bullet (\mathrm{B} + \mathrm{C}) \equiv (\mathrm{A} \bullet \mathrm{B}) + (\mathrm{A} \bullet \mathrm{C})$

(b) $\mathrm{A} + (\mathrm{B} \bullet \mathrm{C}) \equiv (\mathrm{A} + \mathrm{B}) \bullet (\mathrm{A} + \mathrm{C})$

(c) $\overline{\overline{\mathrm{A}}} \equiv \mathrm{A}$

(d) $\overline{\mathrm{A} \bullet \mathrm{B} \bullet \mathrm{C}} \equiv \overline{\mathrm{A}} + \overline{\mathrm{B}} + \overline{\mathrm{C}}$

(e) $\overline{\mathrm{A} + \mathrm{B} + \mathrm{C}} \equiv \overline{\mathrm{A}} \bullet \overline{\mathrm{B}} \bullet \overline{\mathrm{C}}$

(f) $\mathrm{A} + \overline{\mathrm{A}} \bullet \mathrm{B} \equiv \mathrm{A} + \mathrm{B}$

(g) $\mathrm{A} \bullet (\mathrm{A} + \mathrm{B}) \equiv \mathrm{A}$

1-9. The cancellation law of regular algebra states that

$$\text{If} \quad X\,(+)\,Y = X\,(+)\,Z, \quad \text{then } Y = Z$$

Show by giving counter examples that Boolean algebra has no equivalent cancellation law. In other words show the following statements are false:

$$\text{If} \quad X + Y = X + Z, \quad \text{then } Y = Z$$

$$\text{If} \quad X \bullet Y = Z \bullet X, \quad \text{then } Y = Z$$

1-10. NAND and NOR are Boolean functions used in design. NAND (NOT AND) is defined as AND followed by NOT; NOR (NOT OR) is defined as OR followed by NOT. Write the defining truth tables for A NAND B and A NOR B.

(a) Write each of the following expressions in a form that has no AND operators:

$$\overline{(A + B)} \bullet \overline{C} \qquad\qquad A \bullet B \bullet \overline{C} + \overline{B} \bullet D$$

(b) Write each of the following expressions in a form that has no OR operators:

$$\overline{\overline{A} + \overline{B} + \overline{C}} \qquad\qquad \overline{A} + \overline{B} + \overline{C \bullet D + \overline{E}}$$

1-11. Define the following terms:

- **(a)** Canonical
- **(b)** Minterm
- **(c)** Maxterm
- **(d)** Sum-of-products form
- **(e)** Product-of sums form
- **(f)** Canonical sum-of-products form
- **(g)** Canonical product-of sums form

1-12. Which of the following expressions is in sum-of-products form? Which is in product-of-sums form

- **(a)** $A + \overline{B \bullet D}$
- **(b)** $C \bullet \overline{D} \bullet E + \overline{F} + D$
- **(c)** $(A + B) \bullet C$

1-13.

(a) Write a four-element vector describing the function X(A,B):

A	B	X
F	F	F
F	T	T
T	F	F
T	T	F

(b) Derive a logic equation for X directly from the truth table

(c) Derive a logic equation for X directly from your vector expression

(d) Show that the results from parts (b) and (c) are equivalent

1-14. Without formally deriving any logic equations, deduce the value of each function W, X, Y, and Z

A	B	C	W	X	Y	Z
0	0	0	0	1	0	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	1	0
1	1	0	0	1	1	1
1	1	1	0	1	1	0

1-15. Write the logic equations corresponding to the following

(a) $X(A,B,C) = \mathbf{m_0 + m_2 + m_5}$

(b) $X(P,Q) = \mathbf{M_1 \bullet M_3}$

1-16. Write the canonical truth tables for each of the following:

(a) $Y(V,W,X) = \mathbf{M_2 \bullet M_3 \bullet M_5 \bullet M_6}$

(b) $Y(C,B,G) = \mathbf{m_1 + m_2 + m_7}$

1-17. Show that Eq (1–13) reduces to $\overline{Y} = \overline{A} \bullet B$:

(a) By using Boolean algebraic reductions

(b) By developing the canonical truth table for each sum in Eq. (1–13) then performing the AND of the truth-table function values to produce a truth table for $\overline{\mathbf{Y}}$, and finally reading the sum-of–products logic equation for $\overline{Y}$ from the truth table

1-18. Consider the following truth table:

A	X	YZ	G
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Derive canonical equations for G and in the following forms:

(a) Sum-of-products on true outputs.

(b) Sum-of-products on false outputs.

(c) Product-of-sums on true outputs.

(d) Product-of-sums on false outputs.

1-19. Prove the correctness of your answers to Exercise 1–20 by reconstructing a truth table for G from each equation

1-20. Consider the following canonical truth table for two functions S and C:

P	Q	R	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Express S and C as eight-element vectors.

(b) Working directly from the vectors, write a canonical sum-of-products equation for *S* and a product-of-sums equation for $\overline{S}$. Show the equivalence of the equations by Boolean algebraic manipulations.

(c) Repeat part (b) for functions *C* and $\overline{C}$.

(d) Directly from the truth table, write a vector for the function S•C. Working from the vector, give a logic equation for $S \bullet C$.

(e) Repeat part (d) for the function S + C.

1-21. By Boolean algebraic transformations, show the equivalence of the forms in Eqs. (1–14) through (1–17).

1-22. Express Eqs. (1–14) through (1–17) using the minterm and maxterm notations.

1-23. Explain the use of X for "both values" and – for "don't care" in truth tables

1-24. Derive the canonical truth tables that correspond to each of the following K-maps:

BA C	00	01	11	10
0	0	0	0	0
1	0	0	1	0

BA DC	00	01	11	10
00	0	1	0	0
01	0	1	—	0
11	—	1	1	0
10	0	1	1	0

1-25. Plot the function in Exercise 1–20 on two K-maps, one map labeled as in Fig.1–2a and the other as in Fig. 1–2b. Simplify the function if possible

1-26. Draw a K-map for each of the truth tables below. Derive a simplified logic equation from each K-map.

A	B	C	M
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	–
1	1	0	0
1	1	1	0

A	B	C	M
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	–
1	0	1	1
1	1	0	0
1	1	1	0

A	B	C	D	M
0	X	0	X	1
0	X	1	1	0
0	0	1	0	-
X	1	1	0	0
1	0	0	0	1
1	X	X	1	–
1	0	1	0	1
1	1	0	0	0

1-27. Here is a K-map for a function S:

BA DC	00	01	11	10
00	0	0	1	0
01	1	0	1	1
11	1	0	0	1
10	0	0	0	0

By circling zeros, give a logic equation for $\bar{S}$ as a sum of products with each product term containing two variables.

1-28. By circling zeros, simplify the functions in Fig. 1–13.

1-29. Assuming that there are three inputs A, B, and C, write a truth table to describe each of these ideas:

- **(a)** The output should be true only when two or more of the input variables are true
- **(b)** The output should be true only when the number of true input variables is odd.
- **(c)** The output should be true only when the number of false input variables is even.
- **(d)** The output should be false only when exactly two of the input variables are true.

1-30. Simplify the functions derived in Exercise 1–29, using K-maps.

1-31. Often the natural formulation of a logic function is not in perfect sum-of-products or product-of-sums form. For example, consider the equation $M = \overline{A} \bullet (B + C) + A \bullet \overline{C}$ Simplify this equation using two different K-map circlings. Are the resulting sum-of-products forms less compact or more compact than the original? (A criterion for compactness is the number of binary AND and OR operators in the expression.) Can you perform elementary factorings on the K-map results that make the results more compact? Compare the original equation with each of the final equations.

1-32. Consider two 2-bit binary numbers, say, A,B and C,D. A function X is true only when the two numbers are different.

- **(a)** Construct a truth table for X.
- **(b)** Construct a four-variable K-map for X, directly from the word definition of X.
- **(c)** Derive a simplified logical expression for X.

1-33. You are installing an alarm bell to help protect a room at a museum from unauthorized entry. Sensor devices provide the following logic signals:

ARMED = The control system is active
DOOR = The room door is closed
OPEN = The museum is open to the public
MOTION = There is motion in the room

Devise a sensible logic expression for ringing the alarm bell.

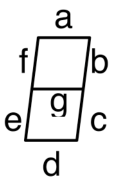
1-34. A large room has three doors, A, B, and C, each with a light switch that can turn the room light on or off. Flipping any switch will *change* the condition of the light

- **(a)** Assuming that the light is off when the switch variables have the values 0, 0, 0, write a truth table for a function LIGHT that can be used to direct the behavior of the light.
- **(b)** Derive a logic equation for LIGHT.

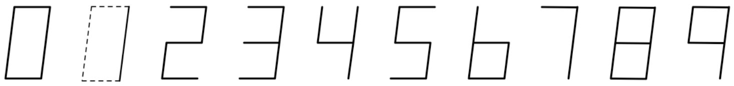
(c) Can you simplify this equation?

(d) How is this exercise related to Exercise 1–31?

1-35. Electronic watches display time by turning on a certain combination of seven light-bar segments to yield approximations of the shape of the decimal digits. For each digit position, the segments are labeled as follows:



The decimal digit displays have the form



For example, the digit 4 has segments b, c, f and g lighted. Internally, the watch represents a decimal digit by a 4-bit binary code, say, D,C,B,A. For example

		D	C	B	A
7	=	0	1	1	1

(a) Develop a multi-output truth table for lighting the segments. The truth table will have inputs D, C, B, and A, and outputs a, b, c, d, e, f, and g. Notice that don't-care conditions arise naturally, since 4 bits can encode 16 combinations, whereas the decimal digits use only 10 of them. Binary codes above 1001 will never occur.

(b) Plot the light segment outputs a through g on four-variable K-maps, and derive a simplified equation for each segment.

1-36. The university pool room has four pool tables lined up in a row. Although each table is far enough from the walls of the room, students have found that the tables are too close together for best play. The experts are willing to wait until they can reserve enough adjacent tables so that one game can proceed unencumbered by nearby tables. A light board visible outside the poolroom shows vacant tables. The manager has developed a digital circuit that will display an additional light whenever the experts' desired conditions arise. Give a logic equation for the assertion of the new light signal. Simplify the equation, using a K-map.