7

COMBINATIONAL CIRCUITS

7.1 INTRODUCTION TO DIGITAL CIRCUITS

Digital design is concerned with the design of digital electronic circuits. Digital circuits are required to handle just two voltage levels, a *true* level and a *false* level. Because these circuits handle two basic levels as opposed to infinitely many voltage levels as analog circuits do, they are more reliable. They last longer. Also, they are more consistent than analog circuits by repeatedly generating the same results under the same input conditions. The best-known digital system today is the computer; many computer-based products are manufactured today. The low cost, the reliability, the versatility of such circuits allows incorporating computers in virtually all intelligent products at the present time. Two main classes of digital circuits cover the world of digital design. The first is combinational circuits. They are digital circuits that produce outputs when the inputs are presented to them. Such circuits have no memory. The second kind of digital circuit is the sequential circuit, or those digital circuits that have memory capability. Combinational circuits will be the subject of Chapters 7 and 8. Sequential circuits will be addressed in Chapter 9.

7.2 BINARY NUMBERS: A QUICK INTRODUCTION

This chapter assumes that the reader has some knowledge about numbering systems, in particular binary and hexadecimal numbering systems. We will

Table 7.1 List of all possible three-bit unsigned binary numbers with their decimal equivalents

3-Bit Binary Numbers (Base 2)	Decimal (Base 10)
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

move at a fast pace throughout this subject, hopefully not to bore anyone and at the same time present the needed fundamentals. A binary number is represented with two uniquely defined digits, *ones* and *zeros.* A binary digit is generically referred to as a *bit*, which stands for *binary term*. Any integer number can be represented with the appropriate number of ones and zeros.

Let us consider three-bit binary numbers first. If we exhaustively come up with all the binary combinations of three binary terms it is easy to see that the list in Table 7.1 contains all the possible binary combinations. In this chapter we will only address positive or unsigned binary numbers. In the next chapter we will cover positive as well as signed or negative numbers.

The algorithm to generate base 10 or decimal numbers is simple and we use it all the time, without even giving it a second thought. In base 10 we have 10 uniquely defined digits, 0 through 9. With those 10 digits we can write all possible integer numbers as long as we have the freedom of having enough digits to represent the largest number that we are interested in. For example, if we are asked to write all the possible 3-decimal digit integers; the first decimal number is $(000)_{10}$ while the largest one is $(999)_{10}$. We know that after $(000)_{10}$ comes $(001)_{10}$ then $(002)_{10}$ and so on until $(009)_{10}$. Now we ran out of uniquely defined digits so we reset the least significant decimal digit to zero and set to 1 the digit left to the rightmost decimal digit or the least significant decimal digit. We now compose $(010)_{10}$, then $(011)_{10}$, and so on until we reach $(019)_{10}$. This algorithm is repeated and we clearly know from grade school how to come up with all the 3-digit decimal numbers all the way up to $(999)_{10}$ or any other larger sequence of them.

Now if we want to do the same thing for a different base number, like for integer binary numbers, the algorithm is no different from what we already do with decimal numbers. The possibly "new" thing is that we only have two uniquely defined *bits*, so that we exhaust the use of each bit sooner than we do when dealing with the decimal numbering system. The sequence of all the possible 3-bit integer binary number was listed in Table 7.1. Note that the binary number 01100111 as an 8-bit number consists of eight bits; each bit position is referred to as: $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$, where bit b_0 is referred to as the *least significant bit* or *LSB* and b_7 is the *most significant bit* or *MSB* of our 8-bit wide number.

So since $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 = 01100111$, that means that every bit is weighted in the following fashion:

$$0\times 2^7 + 1\times 2^6 + 1\times 2^5 + 0\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0$$
$$= 64 + 32 + 4 + 2 + 1 = (103)_{10}.$$

After computing the sum above the equivalent decimal number is $(103)_{10}$

$$= 64 + 32 + 4 + 2 + 1 = (103)_{10}.$$

Continuing with 8-bit binary integers, 8 bits span $2^8 = 256$ uniquely defined 8-bit binary combinations. The reader should convince herself that zero in 8-bit binary is: 0000_0000 and 255 is 1111_1111, the largest possible 8-bit unsigned binary integer. The underscores used to separate four-bit groups is simply to enhance the readability of the number. The reader not too familiar with binary sequences is encouraged to write down the complete binary sequence starting at $(0000\_0000)_2$ ending at $(1111\_1111)_2$.

Hexadecimal numbers have 16 uniquely defined symbols:

0, 1, 2,…, 9, A, B, C, D, E, F. Table 7.2 lists the first 16 hexadecimal (or hex) numbers, their binary and decimal equivalents.

Table 7.2 List of 16 uniquely defined hex digits, their binary and decimal equivalents

Hexadecimal (Hex, Base 16)	Binary (Base 2)	Decimal (Base 10)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Example 7.1 Represent the decimal number $(183)_{10}$ in binary and in hex.

Solution to Example 7.1

Initially we are not sure how many bits the number $(183)_{10}$ requires in binary representation. To sort of figure out the number of bits required, let us list the powers of two:

Note from Table 7.3 that the number $(128)_{10}$ can be represented in binary simply by writing:

$$(1000\_0000)_2 = (128)_{10}.$$

Similarly 128 + 64 = $(192)_{10}$ is written as:

$$(1100\_0000)_2 = (192)_{10}.$$

Since we want to write the number *$(183)_{10}$* in binary representation we know that *8* bits will suffice. To convert from decimal to binary, algorithmically we proceed as follows:

We divide *$(183)_{10}$* by 2 to give an integer quotient of *91* and a remainder of ½. This process is repeated until the integer quotient becomes zero. We record all the operations as shown below:

Integer quotient		Remainder	Bit position	Weight
183/2 = 91	+	½	b0 = 1(LSB)	1
91/2 = 45	+	½	b1 = 1	2
45/2 = 22	+	½	b2 = 1	4
22/2 = 11	+	0	b3 = 0	8
11/2 = 5	+	½	b4 = 1	16
5/2 = 2	+	½	b5 = 1	32
2/2 = 1	+	0	b6 = 0	64
1/2 = 0	+	½	b7 = 1(MSB)	128

From above we conclude that $(183)_{10} = (1011\_0111)_2$.

Table 7.3 Some powers of two and bit position in a binary number

Powers of 2	Binary Bit Position
$2^0 = 1$	b_0
$2^1 = 2$	b_1
$2^2 = 4$	b_2
$2^3 = 8$	b_3
$2^4 = 16$	b_4
$2^5 = 32$	b_5
$2^6 = 64$	b_6
$2^7 = 128$	b_7

Now to convert $(183)_{10}$ to hex we simply translate each group of four bits into their hex equivalent starting with the LSB position according to Table 7.2, thus:

$$(183)_{10} = (1011\_0111)_2 = (B7)_{16}.$$

In Chapter 8 we will address some interesting ways of representing positive and negative binary numbers. This will be useful to design digital arithmetic circuits.

7.3 BOOLEAN ALGEBRA

In 1854, English mathematician and philosopher George Boole developed what is known today as Boolean algebra. Later on in 1938, American engineer and mathematician Claude Elwood Shannon also introduced a two-valued algebra he denominated switching algebra. Boolean algebra, also known as switching algebra, consists of binary variables and the logical operations among them. All logic variables that we will deal with have a binary value; that is, they can only take one out of two possible values, either 1 or 0, which we can associate with a true value and a false value, respectively, or vice-versa. Why do we need to deal with logic that only handles two values or two logic levels? Because it is easier and it is more reliable to develop, build, and use circuits that handle two values rather than circuits that handle infinitely many or many more values than just two. Circuits that handle infinitely many values are commonly referred to as analog circuits. Analog circuits are not as reliable, repeatable, and maintainable as digital circuits are.

The three most important logic operations are:

AND, OR, NOT

If we group these operators as follows: group (1) *AND, NOT* group (2) *OR, NOT* it is interesting to state that all possible binary or combinational functions, regardless of their length or complexity, can be implemented with just the operators of either group (1) or (2). We will come back to this concept once we study the fundamental logic rules and operations.

7.3.1 *AND* Logic Operation

Given a two-variable switching function $f(A, B)$, where A and B are binary-valued variables, function f can be exhaustively represented with the aid of a *truth table.*

Table 7.4 Truth table for the *AND* logic operator

Input A	Input B	Output $f(A,B)$ = A B
0	0	0
0	1	0
1	0	0
1	1	1

Binary-Valued Functions: Example 7.2 Let *A* and *B* be two-valued binary independent variables. Let us assume that variable *A* means: *it-is-a-sunny-day* and variable *B* means: *the-soil-is-dry.* Further assume that when *A* is true it takes the value 1, when *B* is true it also takes the value 1. We want to come up with a binary-valued function *f* of variables *A, B* that is true (1) when both *A and B* are both true, else *f* is false (0). Table 7.4 explicitly and fully describes that requirement.

Now let us assume that the meaning of function *f* is: "turn-on-watering-system." So it seems intuitive to think that *f* = *A And'ed with B* is true when both *A* and *B* are true, else *f* is false. The logic symbol for the *AND* operator is a dot or the absence of it. For example:

$$f(\text{A, B}) = \text{A And-ed with B} = \text{A.B} = \text{AB}.$$

We will interchangeably place the *AND* "." (dot) or leave it out trying to make the logic expression more readable.

A true means: *it-is-a-sunny-day* while, *A* false means: NOT *it-is-a-sunny-day,* or grammatically more pleasant, false *A* means "it is not a sunny day."

This can be written in two ways: $\overline{it-is-a-sunny-day}$ = (it-is-a-sunny-day)'

Similarly B true means: "the-soil-is-dry" while B false means: NOT "the-soil-is-dry"

Finally, it is also important to see that function *f* is binary-valued as well; refer to Table 7.4.

Example 7.3 Derive the truth table for of three-input variable binary-valued function *g (A, B, C)*. The *And-ing* of three or more variables does not change the significance of the *AND* operator. In general the *And-ing* of any number of binary-valued variables is true when the true value of each independent variable *(A, B, C, . . .)* is true. For all other cases our function is false. This is succinctly listed in Table 7.5.

7.3.2 *OR* Logic Operation (Also Called *Inclusive OR, or XNOR*)

Let us begin with a little more advanced logic function, the 3-variable *OR* or *inclusive OR.* Given the three binary-valued independent variables *A, B, C,* function $h(A,B,C) = A + B + C$ is true if and only if any one or more

Table 7.5 Three-variable *AND* truth table

Input C	Input B	Input A	Output g (*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

Table 7.6 Three-variable *OR* truth table

Input C	Input B	Input A	Output h (A, B, C) = A + B + C
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

independent variables are true, else *h (A,B,C)* is false. The "+" signs are not arithmetic signs, they represent the logic *OR* (or *inclusive OR*) operation. This *OR* is referred to as being inclusive because the output function is true not only whenever each independent variable is true, but also includes the case when one or more than one independent variable is true. We will cover shortly the two-variable exclusive *OR*, which requires that both independent variables have the opposite true value for the *exclusive OR* function to be true.

Table 7.6 depicts a three-variable inclusive OR function.

Exercise: Derive the truth table of a two-variable inclusive *OR* function.

7.3.3 NOT Logic Operation or Inversion—*NAND* and *NOR*

Inversion is the simplest of all logic operations. Given a binary-valued function *f* of an arbitrary number of independent binary-valued variables and its associated truth table, *NOT f or* $\overline{f}$ truth table is generated by changing function f output column *ones* with *zeros* and *zeros* with *ones*.

Table 7.7 Truth table for the *AND* & *NAND* logic operators

Input A	Input B	Output f(A,B) = AB	Output $\overline{f}(\mathrm{A,B}) = \overline{AB}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Table 7.8 Truth table for the *OR* & *NOR* logic operators

Input A	Input B	Output g(A,B) = A + B	Output $\overline{g} = \overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Example 7.4 Let us define a function $\overline{f} = \overline{AB}$ which we will refer to as *A NAND'ed* with *B*. The letter "*N*" in the acronym "*NAND*" stands for negation or not.

Referring to our originally studied function $f = AB$, in Table 7.4, function $\overline{f} = \overline{AB}$ is simply annotated in *truth table* of Table 7.7.

In the above table, output *f* column, *f(A,B) = AB,* has been complemented bit-by-bit to form the column of our *NAND* function that is $\overline{f} = \overline{AB}$.

Example 7.5 Let us define a function $\overline{g} = \overline{A+B}$ which we will refer to as *A NOR'ed* with *B*. The letter "*N*" in the acronym "*NOR*" stands for negation or not. Table 7.8 presents the truth table of a two-variable *OR* under column *g* and *NOR*, under column $\overline{g}$.

7.3.4 Exclusive *OR* Logic Operation or *XOR*

The two-variable *XOR* is defined as true whenever an independent variable is true while the other one is false. Additionally when both variables have the same true value the *XOR* is false. Table 7.9 below presents the truth table of a two-variable *exclusive-or* function and the two-variable *equivalence* function.

Another way of defining the two-variable *XOR* is:

$$A\ XOR\ B = A \oplus B = A\overline{B} + \overline{A}B. \tag{7.1}$$

Table 7.9 Exclusive or and equivalence truth tables

Input A	Input B	Output f(A,B) = A ⊕ B = A XOR B	Output $\overline{f}(\mathrm{A,B}) = \overline{A \oplus B}$ = A EQUIVALENCE B = A XNOR B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Equation (7.1) is interpreted from Table 7.9 as follows: *A* exclusive-or *B* is true whenever (*A* is true and *B* false) or (when *A* is false and *B* is true). The parentheses used in the previous sentence emphasize the precedence of the logical operations. Note that $A \oplus B$ is not true when both *A* and *B* have the same logic value. This is the reason why the *XOR* is referred to as an *exclusive-OR*, it excludes the cases for which both *A* and *B* have the same true value. (Two binary variables have the same true value whenever both are true or both are false.) Remember that *A* + *B* (*A* inclusive-OR *B*) is true when both *A* and *B* are true in addition to being true when either only **A** or **B** is true.

The negation of $A \oplus B$ or $\overline{A \oplus B}$ is referred to as *A equivalence B* or *A XNOR B*. Refer one more time to Table 7.9. Similarly, $\overline{A \oplus B}$ = *A equivalence B* is defined as true whenever ($\overline{A}$ and $\overline{B}$ are true) or (*A* and *B* are true), else *A* equivalence *B* is false.

7.3.5 DeMorgan's Laws, Rules, and Theorems

DeMorgan's laws are the most powerful Boolean algebra rules. There are two of them. First we will state the two-variable laws, then we will present the generalized multivariable rules. Let *A* and *B* be two-valued independent variables, then

$$\text{Rule (1) } \overline{A+B} = \overline{A}.\overline{B} \tag{7.2}$$

$$\text{Rule (2) } \overline{A.B} = \overline{A}+\overline{B}. \tag{7.3}$$

It is appropriate to prove these rules and we will do that using truth tables.

We build Table 7.10 containing independent variables *A* and *B,* then we will generate columns corresponding to the following functions: $\overline{A}$, $\overline{B}$, *A* + *B*, $\overline{A+B}$, $\overline{A}\overline{B}$. Shall the column corresponding to $\overline{A+B}$ equal to column $\overline{A}\overline{B}$ we can affirm that Rule (1) holds.

Adding functions: $\overline{A.B}$, $(\overline{A}+\overline{B})$ under columns *viii* and *ix* we also prove Rule (2).

Looking at the results of Table 7.10 we observe that columns *vi* and *vii* are identical, this proves that: *Rule (1)* $\overline{A+B} = \overline{A}\overline{B}$ is true. Similarly, we observe that columns *viii* and *ix* are identical, thus proving *Rule (2)* $\overline{A.B} = \overline{A}+\overline{B}$.

Table 7.10 Truth table to prove 2-variable DeMorgan's rules

i	ii	iii	iv	v	vi	vii	viii	ix
A	B	$\bar{A}$	$\bar{B}$	A + B	$\overline{A+B}$	$\bar{A}\bar{B}$	$\overline{A.B}$	$(\bar{A}+\bar{B})$
0	0	1	1	0	1	1	1	1
0	1	1	0	1	0	0	1	1
1	0	0	1	1	0	0	1	1
1	1	0	0	1	0	0	0	0

Both Rules (1) and (2) can be generalized for *n* binary valued variables W_0, W_1, W_2, . . . W_{n-1}, where *n* is an integer.

Generalized DeMorgan Rule (1)

$$\overline{W_0 + W_1 + W_2 + \ldots + W_{n-1}} = \overline{W_0}.\overline{W_1}\,\overline{W_2}.\ldots.\overline{W_{n-1}}. \tag{7.4}$$

Generalized DeMorgan Rule (2)

$$\overline{W_0.W_1.W_2.\ldots.W_{n-1}} = \overline{W_0} + \overline{W_1} + \overline{W_2} + \cdots + \overline{W_{n-1}}. \tag{7.5}$$

Exercise: Prove using truth tables that DeMorgan's Rules (1) and (2) hold for four variables. Hint: Four variables will span 16 unique binary combinations.

7.3.6 Other Boolean Algebra Postulates and Theorems

We present in this section some other basics postulates and theorems used in Boolean algebra. Most of them are quite intuitive and a few others are not so intuitive. Postulates need not be proven, theorems generally are. Only some less intuitive theorems will be proven.

Assume that Table 7.11 uses binary-valued variables: *X, Y,* and *Z*.

Example 7.6 Using Boolean postulates and theorems, find the complement the following Boolean expressions. Reduce the expressions as much as possible.

(a) $f(X,Y) = \bar{X}Y + \bar{Y}X$

(b) $f(\mathrm{W},X,Y,Z) = (\mathrm{W}\bar{X} + Y)\bar{Z}$

(a) We first apply DeMorgan's rules to the complement of equation (a) and then apply the distributive postulate to the complemented function, we obtain:

$$\bar{f} = XY + \bar{X}\bar{Y}.$$

Table 7.11 Boolean algebra postulates and theorems

	Part (1)	Part (2): Its Dual
Identity postulate	$X+0=X$	$X.1=X$
Idem-potent postulate	$X+\bar{X}=1$	$X.X=X$
Neutral element postulate	$X+1=1$	$X.0=0$
Complements postulate	$X+\bar{X}=1$	$X.\bar{X}=0$
Distributive postulate	$X(Y+Z)=XY+XZ$	$X+YZ=(X+Y)(X+Z)$
Involution theorem	(Double negation) $\bar{\bar{X}}=X$	
Commutative theorem	$X+Y=Y+X$	$X.Y=Y.X$
Associative theorem	$X+(Y+Z)=(X+Y)+Z$	$X(YZ)=(XY)Z$
Absorption theorem	$X+XY=X$	$X(X+Y)=X$

Note that all the postulates and theorems only apply to *ANDs, ORs, or NOTs operators.* Some of the above postulates and theorems may or may not apply to exclusive- and/or equivalence operations.

(b) Applying distribution, complementing, and applying De Morgan's rule to Equation (b), we obtain:

$$f=W\bar{X}\bar{Z}+Y\bar{Z},\text{ then }\bar{f}=\overline{W\bar{X}\bar{Z}}.\overline{Y\bar{Z}}=(\bar{W}+X+Z)(\bar{Y}+Z).$$

Let us use distribution one more time, thus:

$$\bar{f}=(\bar{W}+X+Z)(\bar{Y}+Z)=\bar{W}\bar{Y}+X\bar{Y}+\bar{Y}Z+\bar{W}Z+XZ+Z=$$

Observe the four right-most terms above; let us apply absorption three consecutive times, among terms: $\bar{Y}Z+\bar{W}Z+XZ+Z$.

Hence:

$$\bar{Y}Z+\bar{W}Z+XZ+Z=\bar{Y}Z+\bar{W}Z+Z=\bar{Y}Z+Z=Z.$$

Finally, $\bar{f}=\bar{W}\bar{Y}+X\bar{Y}+Z$.

7.3.7 The Duality Principle

Table 7.11 lists the most important postulates and theorems in Boolean algebra. The previously covered DeMorgan Laws present a good example of duality. Let us look back at Equations (7.2) and (7.3). The duality principle states that one rule (e.g., rule 1) may be obtained for the other one (2) by interchanging operators and identity elements. For our example, using DeMorgan Laws, we interchange *OR* and *AND* operators and replace 1's with 0's and 0's with 1's. Also refer to the complements postulate to see how operators are interchanged and 1's and 0's are swapped.

7.3.8 Venn Diagrams

In order to prove some of the theorems in Table 7.11, instead of using *truth tables*, we will use *Venn diagrams. Venn diagrams* provide a graphical methodology to visually perform logic operations in a very intuitive fashion. Each variable such as A, B, and so on is represented with a circle. All variables must be within a single rectangular area, which is referred to as the *universal* set, which is a frame of reference where all variables reside. The purpose of having a *universal* set is to easily draw A and its complement $\bar{A}$. Figure 7.1 (a) depicts variable A, which is represented by the area within the circle ($\bar{A}$ is the area outside of A but within the universal set), (b) shows A *inclusive-or* B (note the complete area of both variables A, B), (c) depicts the common area to both A and B, thus area AB is cross-hatched, (d) shows A *XOR* B (observe that the cross-hatched areas are also equivalent to $A\bar{B} + \bar{A}B$), and finally (e) shows the complement of A *XOR* B.

Using the concepts just learned about Venn diagrams we will use them to prove graphically some of the theorems listed in Table 7.11.

Figure 7.2 depicts the graphical justification of the absorption theorem, part 1. Part (a) shows X, part (b) shows XY, and part (c) shows the *ORing* of $X + XY = X$.

Figure 7.3 shows the graphical representation of the dual of the first absorption part 2. $X\ (X + Y) = X$ (from Table 7.11).

Exercise: Show using Venn diagram the validity of DeMorgan Rules (1) and (2).

7.4 MINTERMS: STANDARD OR CANONICAL SUM OF PRODUCTS (SOP) FORM

Binary variables may appear in their normal form, sometimes referred to as their true form, and their complemented form. For example, given A a binary-valued variable, we can have A and $\bar{A}$. If we consider two binary variables, such as A and B, since each one of them can take its true and complemented value, both variables together span four unique binary combinations. Table 7.12 depicts two variables and their four combinations and also three variables and their eight unique binary combinations.

Each of those binary combinations of the *ANDed* variables is referred to as a *minterm*. Generalizing the preceding concept given n variables, such n variables can span 2^n unique binary combinations. Each of those combinations is shown in Table 7.12 for 3 and 2 variables.

Note that each minterm is the logic product or the *ANDing* of the variable in question, not complemented when they represent ones and complemented when they represent zeros. It is also important to say that variable A was

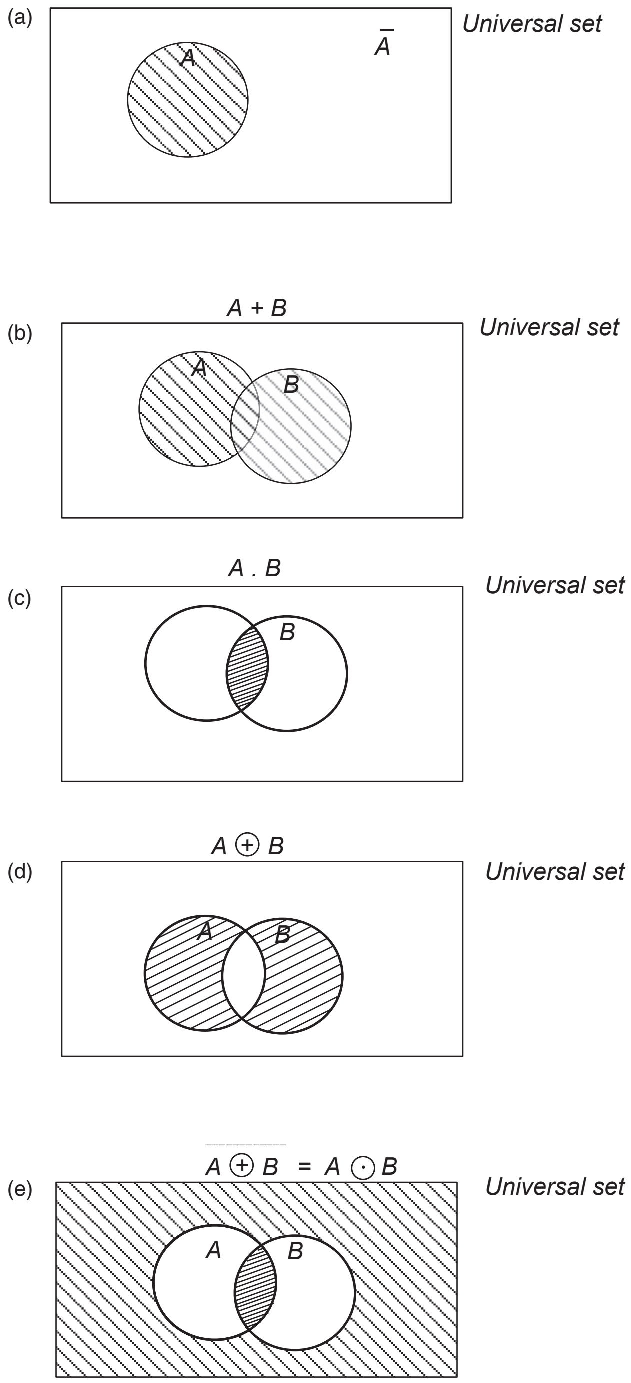


Figure 7.1 Venn diagrams (a) A and its complement $\bar{A}$; (b) $A + B = A$ *Inclusive-or* B; (c) $AB = A$ *and* B; (d) $A \oplus B = A$ *Exclusive-or* B; (e) $\overline{A \oplus B} = A$ *Equivalence* B.

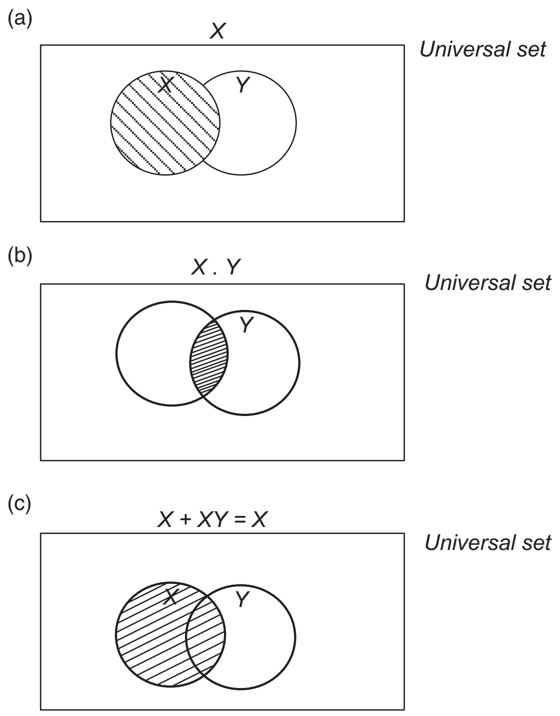


Figure 7.2 Absorption Theorem (a) *X*; (b) *XY*; (c) $X + XY = X$.

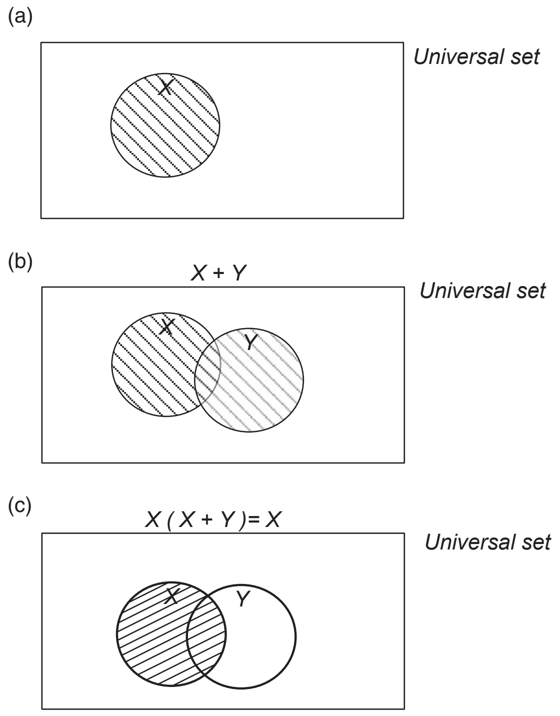


Figure 7.3 Absorption theorem part (2) (a) *X*; (*b*) $X + Y$; (c) $X\,(X + Y) = X$.

Table 7.12 Three-variable and two-variable *minterms*

3-Variable Minterms					2-Variable Minterms			
C	B	A	Minterm	Acronym	B	A	Minterm	Acronym
0	0	0	$\bar{C}\bar{B}\bar{A}$	m_0	0	0	$\bar{B}\bar{A}$	m_0
0	0	1	$\bar{C}\bar{B}A$	m_1	0	1	$\bar{B}A$	m_1
0	1	0	$\bar{C}B\bar{A}$	m_2	1	0	$B\bar{A}$	m_2
0	1	1	$\bar{C}BA$	m_3	1	1	BA	m_3
1	0	0	$C\bar{B}\bar{A}$	m_4				
1	0	1	$C\bar{B}A$	m_5				
1	1	0	$CB\bar{A}$	m_6				
1	1	1	CBA	m_7				

chosen to represent the *LSB* of each minterm. Without loss of generality, any variable name can represent any bit position. The ordering is simply a matter of convenience. As long as one chooses a way of doing things, it is better and less error-prone to stick to a methodical way of defining your minterms.

Example 7.7 Given a three-variable function $f(C, B, A) = CBA + \bar{C}\bar{B}A + C\bar{B}\bar{A}$, write the function as a sum of minterms. Table 7.13 presents a 3-variable function table listing all of its minterms. By inspection of function *f(C, B, A)* we can identify that the function contains three minterms that are *ORed* (or logically summed). The first minterm is m_7, the second one is m_1 and the third and last one is m_4.

So our function $f(C, B, A) = CBA + \bar{C}\bar{B}A + C\bar{B}\bar{A}$ can also be written in a so-called sum-of-products form (*SOP*), and after rearranging its minterms in an ascending order we obtain:

$$f(C, B, A) = m_1 + m_4 + m_7 \tag{7.6}$$

which in a more compact form can be written as:

$$f(C, B, A) = \sum(1, 4, 7). \tag{7.7}$$

Function *f*(*C, B, A*) is written by Equation (7.7) in a *canonical or standard sum-of-products form (SOP).* Each minterm is generically referred to as a *product*, the logic *AND* is equivalently called a logic product because of its similarity with regular arithmetic. And it is a sum-of-product because each minterm present in the function is *ORed* or logically added. The *OR* operation is also called a *logic sum* or simply a *sum* if the context clearly is that of a logic *OR-ing*.

Table 7.13 Three-variable function of Example 7.7

3-Variable Minterms				
C	B	A	Minterm	$f(C, B, A)$
0	0	0	$\bar{C}\bar{B}\bar{A}$	0
0	0	1	$\bar{C}\bar{B}A$	1
0	1	0	$\bar{C}B\bar{A}$	0
0	1	1	$\bar{C}BA$	0
1	0	0	$C\bar{B}\bar{A}$	1
1	0	1	$C\bar{B}A$	0
1	1	0	$CB\bar{A}$	0
1	1	1	CBA	1

Example 7.8 Given function $f(C, B, A) = CA + BA$, expand it to represent it in its canonical *SOP* form.

Solution to Example 7.8

The given function clearly is not already given as a sum of its minterms.

What we need to do is to create "*logic redundancies*" that do not affect the original logic of the function. For example, *ANDing* terms like $(C\bar{C})$, since $(C\bar{C}) = 1$ and *ANDing* 1 to any logical expression does not alter its original logic, is a way of creating such redundancy. Another type of possible redundancy is *ANDing* terms like $(B + \bar{B})$ to the original function, which will not alter the initial logic of the function because $(B + \bar{B}) = 1$. Proceeding with our function f:

$$f(C, B, A) = CA + BA. \tag{7.8}$$

Since (Eq. 7.8) term CA is missing the literal B we *AND* the term $(B\bar{B})$ with the term CA without changing the original logic of function f. At the same time we create a redundancy to the term BA by *ANDing* the term $(C\bar{C})$ with the term BA. Hence:

$$f(C, B, A) = CA(B\bar{B}) + (C\bar{C})BA. \tag{7.9}$$

Applying logic product distribution and making sure that variables are consistently organized from C down to A (e.g., CBA)

$$f(C, B, A) = CBA + C\bar{B}A + CBA + \bar{C}BA. \tag{7.10}$$

Eliminating only the second instance of the term CBA because it is redundant yields:

$$f(C, B, A) = C\bar{B}A + CBA + \bar{C}BA. \tag{7.11}$$

Rewriting Equation (7.11) in *SOP* form and rearranging terms:

$$f(C,B,A) = \mathrm{m}_5 + \mathrm{m}_7 + \mathrm{m}_3 = \sum(3,5,7). \tag{7.12}$$

7.5 MAXTERMS: STANDARD OR CANONICAL PRODUCT OF SUMS (POS) FORM

Given any logic function in its standard *SOP* form, taking its complement we obtain a *product-of-sum* form (*POS*). Each *POS* term is referred to as a *maxterm*.

Table 7.14 lists all the *minterms* for a three-variable function and its corresponding complements. Such complements are defined as the function *maxterms*.

Binary-valued functions not only can be expressed in a *standard SOP* form, but also in a *standard product-of-sums* (*POS*) form. Let us explain with the following example.

Example 7.9 Given the following function in *standard SOP* form, convert it to its *standard POS* form.

$$f(C,B,A) = \mathrm{m}_6 + \mathrm{m}_7 + \mathrm{m}_3 = \sum(3,6,7). \tag{7.13}$$

The following steps will lead to the expected results:

It is intuitive to see that if a function *f* is given in *standard SOP* form, such as Equation (7.13), then its complement ($\bar{f}$) also in *standard SOP* form will list all those minterms that are not listed in function *f*. That is to say:

$$\bar{f}(C,B,A) = \mathrm{m}_0 + \mathrm{m}_1 + \mathrm{m}_2 + \mathrm{m}_4 + \mathrm{m}_5 = \sum(0,1,2,4,5). \tag{7.14}$$

Table 7.14 Minterms and maxterms for three-variable functions

C	B	A	Minterm	Minterm Acronym $\mathrm{m}_\mathrm{i} = \overline{M_i}$	Maxterm	Maxterm Acronym $\mathrm{M}_\mathrm{i} = \overline{m_i}$
0	0	0	$\bar{C}\bar{B}\bar{A}$	m_0	$C+B+A$	M_0
0	0	1	$\bar{C}\bar{B}A$	m_1	$C+B+\bar{A}$	M_1
0	1	0	$\bar{C}B\bar{A}$	m_2	$C+\bar{B}+A$	M_2
0	1	1	$\bar{C}BA$	m_3	$C+\bar{B}+\bar{A}$	M_3
1	0	0	$C\bar{B}\bar{A}$	m_4	$\bar{C}+B+A$	M_4
1	0	1	$C\bar{B}A$	m_5	$\bar{C}+B+\bar{A}$	M_5
1	1	0	$CB\bar{A}$	m_6	$\bar{C}+\bar{B}+A$	M_6
1	1	1	CBA	m_7	$\bar{C}+\bar{B}+\bar{A}$	M_7

Note that from Table 7.11 of postulates and theorems, *idem potent,* which states that:

$X + \bar{X} = 1$, does justify f and $\bar{f}$ expressed by Equations (7.13) and (7.14).

Now let us proceed to take the complement of $\bar{f}$, thus from Equation (7.14):

$$\bar{\bar{f}}(C, B, A) = \overline{m_0 + m_1 + m_2 + m_4 + m_5}.$$

Applying Boolean algebra on $\bar{\bar{f}}$, using the *minterm* and *maxterm* definitions from Table 7.14 we obtain:

$$\bar{\bar{f}} = (C + B + A) + (C + B + \bar{A}) + (C + \bar{B} + A) + (\bar{C} + B + A) + (\bar{C} + B + \bar{A}). \quad (7.15)$$

Identifying every maxterm from Equation (7.15) with the aid of Table 7.14 we find that:

$$\bar{\bar{f}} = f = \mathrm{M}_0\mathrm{M}_1\mathrm{M}_2\mathrm{M}_4\mathrm{M}_5. \quad (7.16)$$

Equation (7.16) can be written in *POS* compact form using the symbol π to indicate multiplication, hence:

$$f = \prod(0, 1, 2, 4, 5). \quad (7.17)$$

Equation (7.17) is a standard *POS* form for the originally given function *f*. It is very important to remember that the π symbol indicates that the numerals within the parentheses are *maxterms* and not *minterms*.

Example 7.10 Let us consider designing a logic block that receives three input binary variables *X, Y,* and *Z* and we want it to have a single output which detects whenever two or more of the inputs are ones, else we want to output to be zero. Derive the truth table for such circuit.

Let us draw a truth table with three inputs and one output. Simply follow the example requirements, whenever we see two or more ones in any *Z Y X* row we must write a one at the output, all other cases require a zero output. A digital circuit such as the one just described is called a *majority detector* circuit. Refer to Table 7.15 for a truth table of the majority circuit.

The *standard SOP* form for our majority detector circuit is:

$$F(Z, Y, X) = \sum(3, 5, 6, 7). \quad (7.18)$$

7.6 KARNAUGH MAPS AND DESIGN EXAMPLES

Unless we work with one or two variables, logic equations can become quite complex, particularly when we need to simplify them and express them as less

Table 7.15 Truth table of a majority detector circuit

Input Z	Input Y	Input X	Output F
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	1

complicated expressions. Rather than using the postulates and theorems of Table 7.11, which can easily become cumbersome and lengthy, there is a methodology attributed to *Karnaugh,* referred to as solving or simplifying logic equations using *Karnaugh* maps. This is the topic of this section. Let us begin defining the *Karnaugh* (K) map construction. For a two-variable map, we need to have a map with as many cells as minterms the number of variables spans. A two-variable K. map has $2^2 = 4$ cells. A three-variable K. map has $2^3 = 8$ cells, and so on. Figure 7.4 depicts a *2, 3,* and *4-cell Karnaugh* maps. We will start covering 2-variable maps progressing onto 3 and 4-variables.

7.6.1 Two-Variable Karnaugh Maps

The two-variable K. map is shown in Figure 7.4a. The map clearly shows the relationship between its cells (squares) and the two variables A and B.

Note that the 2-variable map is drawn such that the rightmost vertical column corresponds to variable A (i.e., $A = 1$), in true value or noncomplemented. The leftmost vertical column corresponds to $\overline{A}$, A false or A complemented. Similarly the bottom row corresponds to B ($B = 1$), and the top row corresponds to $\overline{B}$ ($B = 0$). The four-cell map becomes fully defined. Note that the cell at $B = 0$ and $A = 0$, corresponds to minterm *0*. Cell at $B = 0$ and $A = 1$ corresponds to minterm 1. Similarly, cell at $B = 1$ and $A = 0$ corresponds to minterm 2, and cell at $B = 1$ and $A = 1$ corresponds to minterm 3. So let us solve some problems to see the 2-variable *Karnaugh* map at work.

We will solve a handful of 2-variable maps in a somewhat intuitive fashion. Problem solving and K. map simplification will become clearer using 3 and 4-variable maps. In some ways, 2-variable maps are too simple to appreciate the properties of K. map method. We will address simplification in the *Karnaugh map* sense more thoroughly after all the 2-variable examples.

Example 7.11 Refer to Figure 7.5a for this example.

Given function $f(B, A) = m_0 + m_1 + m_2 + m_3$, find a maximally simplified *SOP form* logic expression. As we discussed in earlier sections, canonical or standard *SOP* forms express a logic function as a logic sum (*Or-ing*) of the appropriate

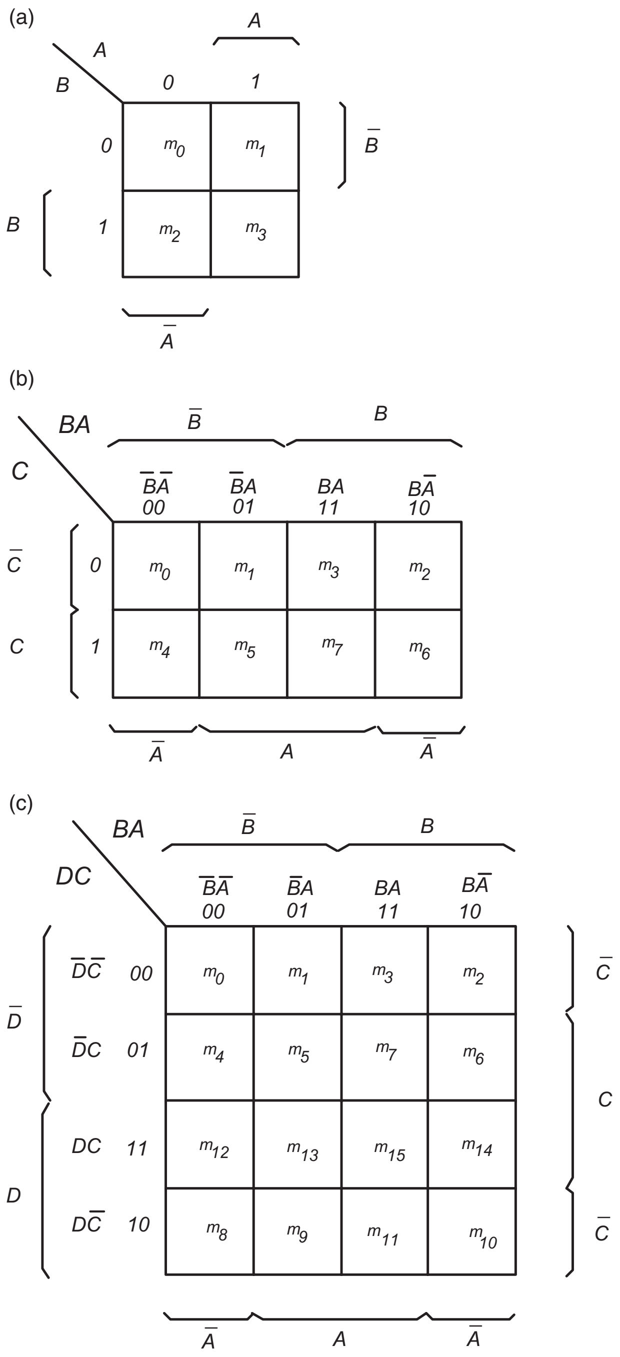


Figure 7.4 Two, three, and four-variable Karnaugh maps definition.

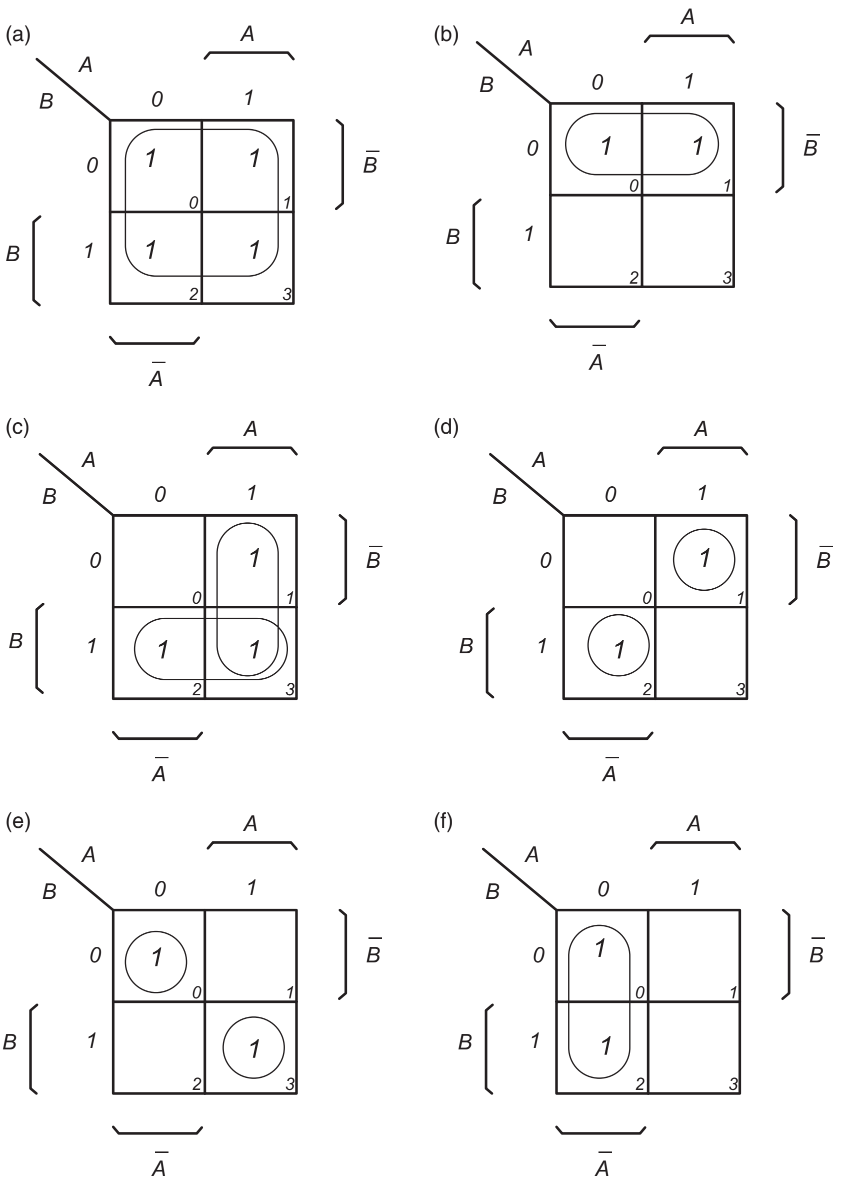


Figure 7.5 Two-variable Karnaugh Maps examples: (a), (b), (c), (d), (e), and (f).

minterms. Standard forms are not maximally simplified from the following criteria. Minterms show all possible variables that the function has. For example, 2-variable functions have two-bit minterms (e.g., such as $B\overline{A}$); three-variable functions have three-bit minterms, and so forth. Some functions can be expressed in a simplified *SOP* form by reducing the number of variables of some or all of its original standard form *SOP* form minterms. Additionally, upon logic simplification, some minterms may disappear from the simplified *SOP* form, this yielding a logic sum of less terms than its corresponding standard *SOP* form. This new form is referred to as a simplified *SOP* form. Let us go over these concepts simplifying our given function $f(B, A)$. By inspection of Figure 7.5a we can see that function f has ones in all of its minterms. It may not be clear right now, but it will become more obvious after we cover a few more examples that function $f(B, A)$ is always true. This means that regardless of the individual values of variables A and B, $f(B, A)$ is always true, that is $f(B, A) = 1$.

This fact is indicated in part (a) of Figure 7.5 by encircling all four minterms.

Summary of what a maximally simplified SOP form is in the K. map sense is:

1. *Not all variables will necessarily show in every term being OR-ed.*
2. *The total number of OR-ed terms will not necessarily be the same number as the number of OR-ed terms in the function's standard SOP form.*
3. *The simplified function will still have an SOP form, this means that it is implemented with just two-levels or logic like a standard SOP form. However, because of points (1) and (2), the number of OR terms will typically (although not all the time) be smaller and not all the variables will be present in each OR-ed term.*

Example 7.12 Given a new function $f(B, A) = m_0 + m_1$, find a maximally simplified *SOP* form. Refer to Figure 7.5b to observe the *K.* map of our function. Note that the *K.* map has four distinct areas: A, $\overline{A}$, B, and $\overline{B}$. Moreover, area A graphically corresponds to the *Or-ing* or *sum* of minterms m_1 and m_3, (i.e., $m_1 + m_3$). Area $\overline{A}$ corresponds to the logic *sum* of m_0 and m_2 (i.e., $m_0 + m_2$). Similarly, area B corresponds to $m_2 + m_3$. Finally, $\overline{B}$ corresponds to $m_0 + m_1$. We can easily identify that our function can simply be represented by area $\overline{B}$. This means that $f(B, A) = m_0 + m_1$ is logically equal to $f(B, A) = \overline{B}$. Other ways of proving that $f = \overline{B}$ is true is by logic simplification or using truth tables or *Venn* diagrams.

For example, let us prove using logic simplification that $f(B, A) = m_0 + m_1 = \overline{B}$. Since:

$$m_0 = \overline{B}\overline{A} \text{ and } m_1 = \overline{B}A,$$

we write:

$$f(B, A) = m_0 + m_1 = \overline{B}\overline{A} + \overline{B}A.$$

Using the distribution property we obtain:

$$f(B, A) = \overline{B}(\overline{A} + A)$$

and since from the idem-potent property from Table 7.11, $\overline{A} + A = 1$, hence:

$$f(B, A) = \overline{B}.$$

Exercise: Prove that $f(B, A) = m_0 + m_1 = \overline{B}$ using *Venn* diagrams.

Example 7.13 Now referring to the *K*. map of Figure 7.5c obtain a maximally simplified *SOP* form for $f(B, A) = m_1 + m_2 + m_3$.

This example is slightly more involved than the previous ones. Now is a good point to start presenting the concept of *adjacent* cells. *Adjacency* in the *K*. map sense are those cells whose binary representation only differs by one bit. Let us inspect any 2-variable *K*. map. Cell *00* (corresponding to minterm m_0) is adjacent to cell *01* because there is only *one bit difference between 00 and 01.* Additionally, cell *00* is adjacent to cell *10* because of the same reason. However, cell *00* is not adjacent to cell *11* (minterm m_3), because two bits differ between binary *00* and *11*. In a generalized and graphical way, adjacent cells are those cells that are above, below, left. and right of a cell. Those cells that are diagonally placed with respect to the cell in question are not adjacent cells. So back to Example 7.13, let us start encircling as many adjacent cells as possible, such that the number of encircled cells is a power of two. When we can no longer encircle more adjacent cells in the first round, we repeat the process again, until we run out of cells to encircle. We must attempt to produce the smallest number of encirclements possible. Every new encirclement of cells may re-encircle previously encircled cells; this process usually ensures that the largest possible number of adjacent cells is obtained. In summary, *Karnaugh* map cells encirclements for the purpose of simplification should follow the following basic steps:

1. *Combine the largest possible number of adjacent cells.*
2. *Such number of cells must be a power of two.*
3. *Minimize the overall number of encircled cells.*
4. *It may be convenient to re-encircle some previously encircled cells to reduce the overall number of variables that a term ends up having.*

It is important to be aware that the methodology described does not necessarily provide a unique solution. It will be up to the design engineer to adopt the most convenient solution for the application. It is also important to know that this technique highly depends on the expertise of the user. The more problems one solves, the better and the easier it will be to obtain a simplified solution. Initially, it is strongly recommended to solve a problem in at least two or more possible ways.

Referring again to Figure 7.5c let us encircle cells *1* and *3*. Clearly, we cannot encircle all three cells because *3* is not a power of *2*. After encircling cells *1* and *3,* only minterm *2* remains uncircled. Let us encircle minterm *2* re-encircling minterm *3* which is adjacent to *2*. We are done encircling all the minterms on the *K.* map that have *ones*. Looking again at Figure 7.5c, we can easily identify that the encirclement of cells *1* and *3* coincides with the area that corresponds to *A*. The encirclement of cells *2* and *3* corresponds to *B*. So the simplified function is:

$$f(B, A) = m_1 + m_2 + m_3 = A + B.$$

Example 7.14 Find maximally simplified *SOP* forms for the two functions given by Figure 7.5d,e. Both of these maps are handled at the same time because it is straightforward to see that one is the complement of the other. Addressing first Figure 7.5d, note that cells *01* and *10* are not adjacent in the *K.* map sense, because they differ by more than one bit position. Consequently, the best possible encirclement is to encircle each minterm separately. We can observe that this function:

$$m_1 + m_2 = \overline{B}A + B\overline{A} \tag{7.19}$$

is $A \oplus B$ from Equation (7.1). In a similar fashion, we can identify that Figure 7.5e function:

$$m_0 + m_3 = \overline{A \oplus B} = A \textit{ equivalence } B = A \textit{ NXOR } B. \tag{7.20}$$

Just as for the previous case, its minterms *00* and *11* are not adjacent.

From Equations (7.19) and (7.20) we conclude that the standard forms for *exclusive or* and *equivalence* cannot be simplified in the *K.* map sense, because for each function, none of its minterms is adjacent to any other minterm within the function.

Example 7.15 For the *K.* map of Figure 7.5f find a maximally simplified *SOP* form.

Now it is easy to see that minterms *00* and *10* are adjacent. We encircle them and notice that they correspond to area $\overline{A}$. Their simplified function is simply, $\overline{A}$.

7.6.2 Three-Variable Karnaugh Maps

The three-variable map was defined in Figure 7.4b. Let us observe the adjacent cells. For example, cell *000* corresponding to minterm m_0 has cells 001, 010, 100; note all these cells differ by no more than one bit position with respect

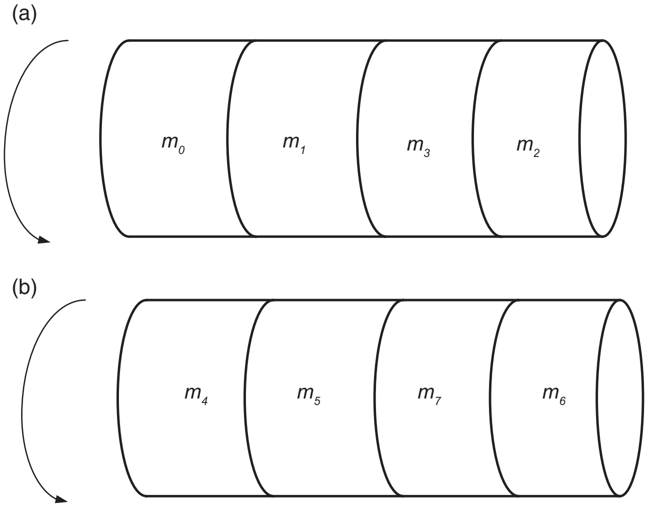


Figure 7.6 Spatial representation of a 3-variable map wrapped around a horizontal axis cylindrical surface. (a) Minterms 0, 1, 3, 2 on the front; (b) minterms 4, 5, 7, 6 on the front.

to cell *000*. Additionally, note that cells *011, 101, 110,* and *111* are not adjacent with respect to cell *000*.

Exercise: Explain why.

It is quite instructive to think about a 3-variable *K*. map as being rolled up on a cylindrical surface with a horizontal axis and also rolled up on a cylindrical surface with a vertical axis. Figure 7.6a depicts a 3-variable rolled up along a horizontal axis cylinder. Minterms m_0, m_1, m_3, *and* m_2 are respectively adjacent to minterms m_4, m_5, m_7, *and* m_6; either looking above or below minterms m_0, m_1, m_3, *and* m_2. Figure 7.6b depicts the same cylindrical surface whose axis is rotated 180°. In Figure 7.6a, minterms m_0, m_1, m_3, *and* m_2 are on the front of the cylinder while minterms m_4, m_5, m_7, *and* m_6 are on the back, thus the latter are not visible from the front. The opposite takes places in Figure 7.6b: minterms m_4, m_5, m_7, *and* m_6 are on the front while minterms m_0, m_1, m_3, *and* m_2 are on the back.

Figure 7.7 depicts a 3-variable map wrapped around a vertical axis cylindrical surface. Since the 3-variable map has *4* cells per row and *2* rows, we can only see two cells of each row in Figure 7.7a.

The front half of the cylinder. This top half contains minterms m_0 *and* m_1 on the top row; and minterms m_4 and m_5 on the bottom row. To aid with the understanding of this spatial description, also refer to Figure 7.4b, which depicts the complete 3-variable map on a flat surface. Back to Figure 7.7a, as we rotate the cylinder 90° in the clockwise direction, while looking at the front of the cylinder, we see minterms m_1 and m_3 on the top front row, and minterms

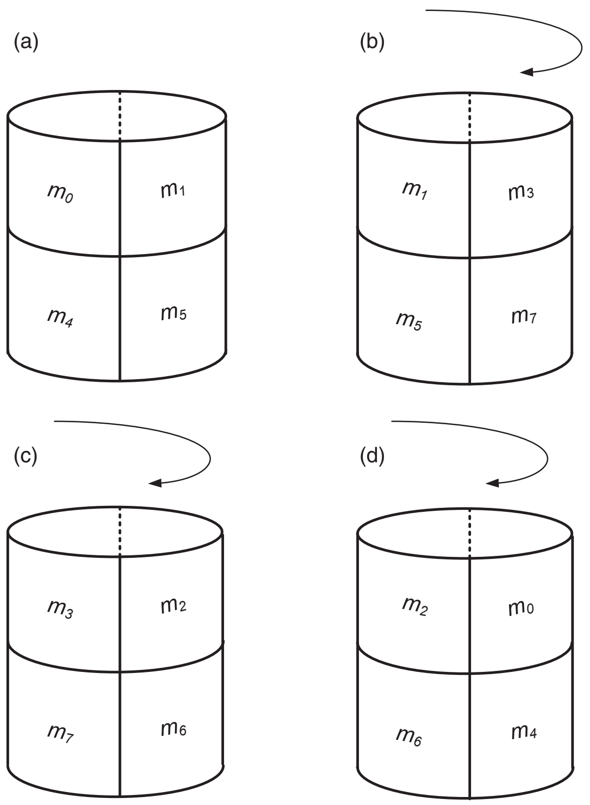


Figure 7.7 Spatial representation of a 3-variable map wrapped around a vertical axis cylindrical surface.

m_5 and m_7 on the second row; see Figure 7.7b. Another 90° rotation in the same direction is depicted in Figure 7.7c, and one last 90° rotation is depicted in Figure 7.7d. What is the purpose of all these cylindrical surfaces rotations for? The ideas that want to be conveyed are an aid to identify adjacent cells without having to figure this out by inspecting every bit of every minterm. Note that adjacent cells in the horizontal direction can be observed using Figures 7.7a through 7.7d, which depict the cylinder along a vertical axis. For example, minterms adjacent to m_1 are m_3 to the right, Figure 7.7b and m_0 to the left, Figure 7.7a. Similarly, this can also be appreciated when looking at the adjacent cells to minterm m_4. Minterm m_5 to the right of m_4 is adjacent to m_4, Figure 7.7a and m_6 to the left of m_4 is also adjacent to m_4, Figure 7.4d. We use the cylindrical picture with a horizontal axis, Figure 7.6a,b when we want to find the adjacent cells to any minterm by looking either above or below the minterm of interest. Obviously, for the 3-variable map case, the cells above and below a minterm of interest are basically the same thing because the

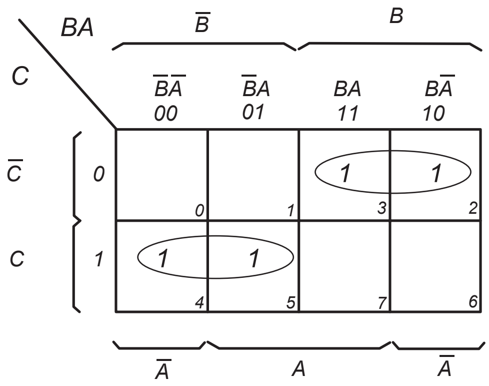


Figure 7.8 Three-variable map for Example 7.16.

3-variable map wraps around the cylinder, and it only has two rows. The *4*-variable K. map is really the first *K.* map that we are studying that will exhibit all the features seen in *Karnaugh* maps.

Example 7.16 Given a 3-variable function $g(C, B, A) = \Sigma(2, 3, 4, 5)$, find a maximally simplified *SOP* form using *K.* maps. Figure 7.8 depicts function g minterms on a 3-variable *K.* map.

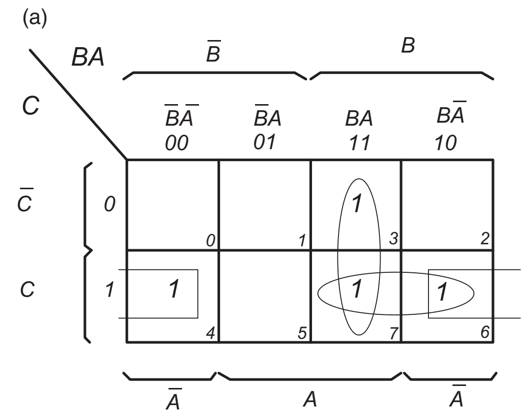
Solution to Example 7.16

Let us begin by encircling the largest possible number of adjacent minterms, such that that number is a power of two. We clearly obtain one grouping with minterms m_2 and m_3 and a second and last grouping produces the grouping of m_4 and m_5. The solution to this problem is particularly straightforward because we do not have other choices of encircling minterms that would produce similar results. Referring to Figure 7.8 we see that the maximally simplified *SOP* form yields:

$$g(C, B, A) = \sum(2, 3, 4, 5) = C\overline{B} + \overline{C}B.$$

Note: The solution of Example 7.16 is the exclusive or of which variables? Answer: *B, C.*

Example 7.17 Given function $w(C, B, A) = \Sigma(3, 4, 6, 7)$, find a maximally simplified *SOP* form using *K.* maps. Representing the given function on the 3-variable *K.* map of Figure 7.9a we encircle m_6 and m_7, then m_3 and m_7 and finally m_4 and m_6. We show this in Figure 7.9a and we obtained the following simplified function:



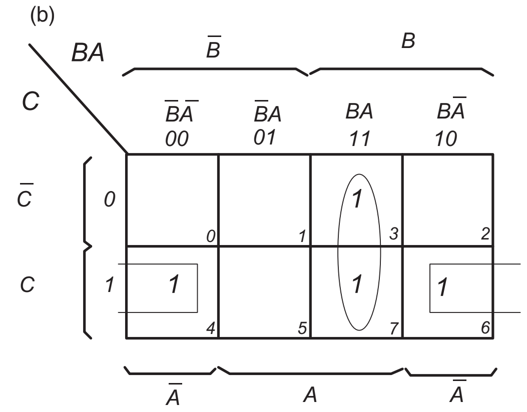


Figure 7.9 Three-variable map for Example 7.16: (a) redundant enclosure m_6 and m_7; (b) redundancy removed.

$$w(C, B, A) = BA + CB + C\bar{A}. \tag{7.21}$$

If we carefully examine Figure 7.9a we can see a redundant enclosure of minterms m_6 and m_7 given by Equation (7.21) term $C\ B$; $C\ B$ does not provide any more logical information. Figure 7.9b shows this term removed, so that our maximally simplified *SOP* form yields:

$$w(C, B, A) = BA + C\bar{A}. \tag{7.22}$$

Exercise: Prove using truth tables that Equations (7.21) and (7.22) are logically equivalent.

7.6.3 Four-Variable Karnaugh Maps

The four-variable map spans 16 cells or minterms. The map is depicted in Figure 7.4c. As usual, the minterms on this map have been encoded with variables:

DCBA, where *A* is the least significant bit variable and *D* is the most significant bit variable. It should be clear that $\bar{D}\bar{C}\bar{B}\bar{A}$ represents m_0, $\bar{D}\bar{C}\bar{B}A$ represents m_1 and so forth. Figure 7.4c also shows the groups of minterms for every one of its four variables. For the purpose of more easily visualizing adjacent cells, we can assume that the map is wrapped around a horizontal axis cylindrical surface as shown in Figure 7.10.

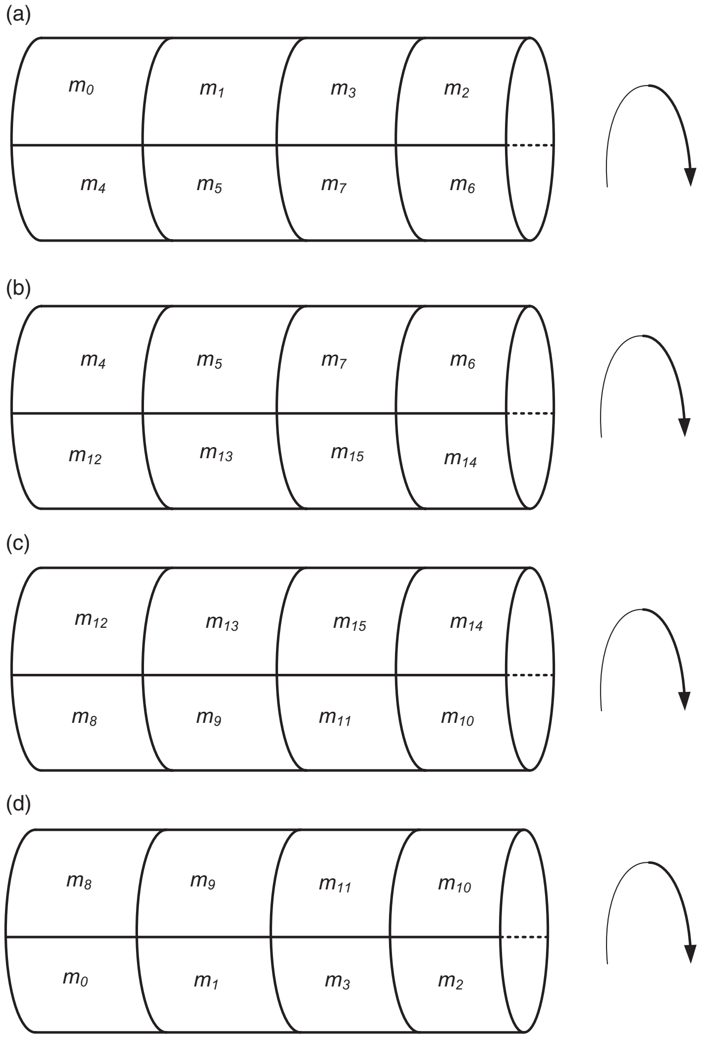


Figure 7.10 Four spatial views of a 4-variable map wrapped around a horizontal axis cylindrical surface.

Figure 7.10a depicts the 4-variable map wrapped around a horizontal cylinder. The two top rows are the only ones visible to the reader. That is, row with minterms m_0, m_1, m_3 & m_2 and row with minterms m_4, m_5, $m_{7,}$ & m_6. The other two rows are on the back of the cylindrical surface and cannot be seen by the reader. Figure 7.10b depicts the initial cylindrical surface rotated 90 degrees in the direction shown by the arrow. Rows with minterms m_4, m_5, m_7, & m_6 and minterms m_{12}, m_{13}, m_{15}, & m_{14} are visible to the reader; the other two rows of minterms are not visible. Similarly, after rotating another 90 degrees Figure 7.10b we obtain 7.10c which depicts the visible minterms and finally 7.10d depicts the visible minterms after rotating the cylinder one more quarter of a turn.

Figure 7.10 allows us to determine simply by visual inspection adjacent minterm to any cell of interest by looking at the cell above and below the cell in question. For example, for cell m_0, we see that m_8 is adjacent to it because m_0 is right below m_8 according to Figure 7.10d. Minterm m_4 is also adjacent to m_0, since it is located right below m_0 in Figure 7.10a.

To analyze the adjacencies left and right of any cell of interest we develop the spatial view of the 4-variable map wrapped around a vertical axis cylindrical surface; this is depicted in Figure 7.11.

In this case we wrap around a vertical axis cylindrical surface our 4-variable map. Figure 7.11a depicts two of the four columns of minterms that are visible to the reader. These are: *m_0, m_4, $m_{12,}$ & m_8 and minterms m_1, m_5, $m_{13,}$ & m_9.* As before, the other two columns are not visible to the reader, since they are on the back of the cylinder of Figure 7.11a. The next three Figure 7.11b,c,e show the previous view of the cylinder rotated around its vertical axis 90 degrees at a time. Each figure depicts the two front columns that are visible to the reader. Using Figure 7.11, it is easy to visualize adjacent cells to the left and to the right of the cell of interest. Between Figures 7.10 and 7.11, all adjacent cells to any one of the 16-cell, 4-variable map can easily be found.

Example 7.18 Given function $G(D, C, B, A) = \sum(1, 4, 6, 7, 8, 9, 10, 11, 15)$, find a maximally simplified *SOP* form. Function *G* is given in Figure 7.12.

Solution to Example 7.18

Using a 4-variable map, we write the unity minterms according to the given function *G*. We encircle groups of adjacent minterms as shown in Figure 7.12. Pay close attention to the choices made grouping adjacent minterms. The groupings shown provide the least number of terms and the least number of variables per term.

The choices made lead to the following *SOP* simplification:

$$G(D, C, B, A) = \bar{C}\bar{B}A + \bar{D}C\bar{A} + CBA + D\bar{C}. \tag{7.23}$$

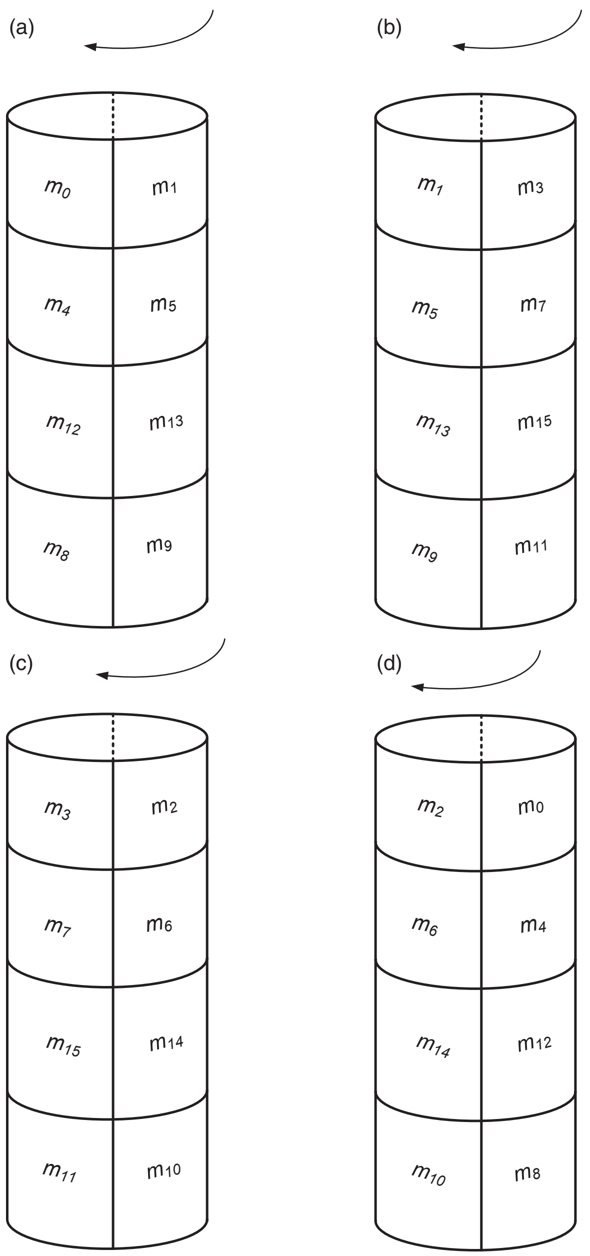


Figure 7.11 Four spatial views of a 4-variable map wrapped around a vertical axis cylindrical surface.

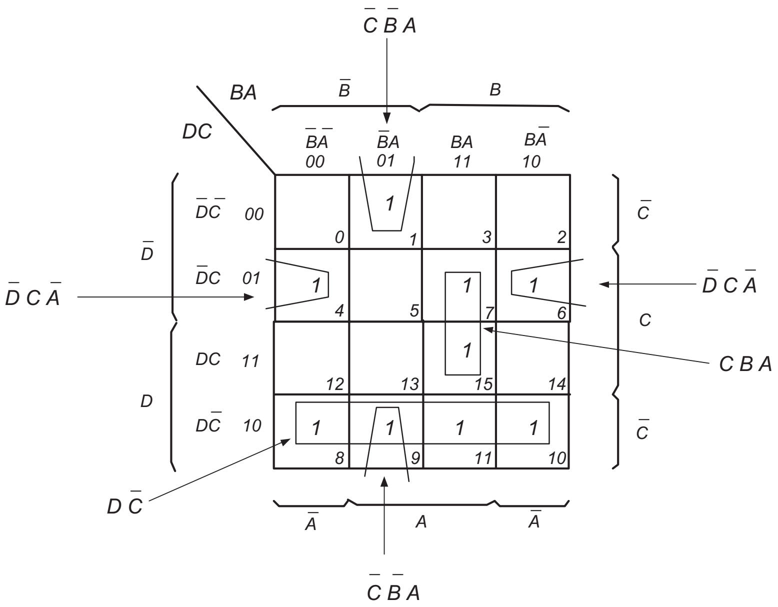


Figure 7.12 Four-variable Karnaugh map for Example 7.18.

7.6.4 Five-Variable Karnaugh Maps

A simple way of handling the 5-variable map, which spans 32 minterms, is to use two 4-variable maps. One of the maps is created for the most significant variable equated to zero (or false), and the second map is created for the most significant variable equated to one (or true). We will use variables *E, D, C, B, A* where *E* is the most significant variable bit and *A* is the least significant variable bit. Figure 7.13 depicts the basic *5*-variable, *32*-minterm, *Karnaugh* map. The upper map is used for $\bar{E}$ and the lower map for *E*. The adjacencies within each map are no different than those adjacencies defined for the *4*-variable map. The additional adjacency criterion of the *5*-variable map is across maps.

A cell on the $\bar{E}$ map that has the same relative position on the *E* map is by definition adjacent because their associated minterm would differ by only the most ignificant bit (variable *E*). Let us look at some examples: cell m_0 and m_{16} are adjacent because $m_0 = \bar{E}.\bar{D}.\bar{C}.\bar{B}.\bar{A} = 0\_0000$ and $m_{16} = E.\bar{D}.\bar{C}.\bar{B}.\bar{A} = 1\_0000$, the only difference is their most significant bit (MSB); thus, since there is a single bit difference they are adjacent. Another example is given by minterms m_{15} and m_{31}. $m_{15} = \bar{E}.D.C.B.A = 0\_1111$ and $m_{31} = E.D.C.B.A = 1\_1111$, again they only differ by one bit.

Exercise: List all other adjacent cells between $\bar{E}$ and the *E* maps.

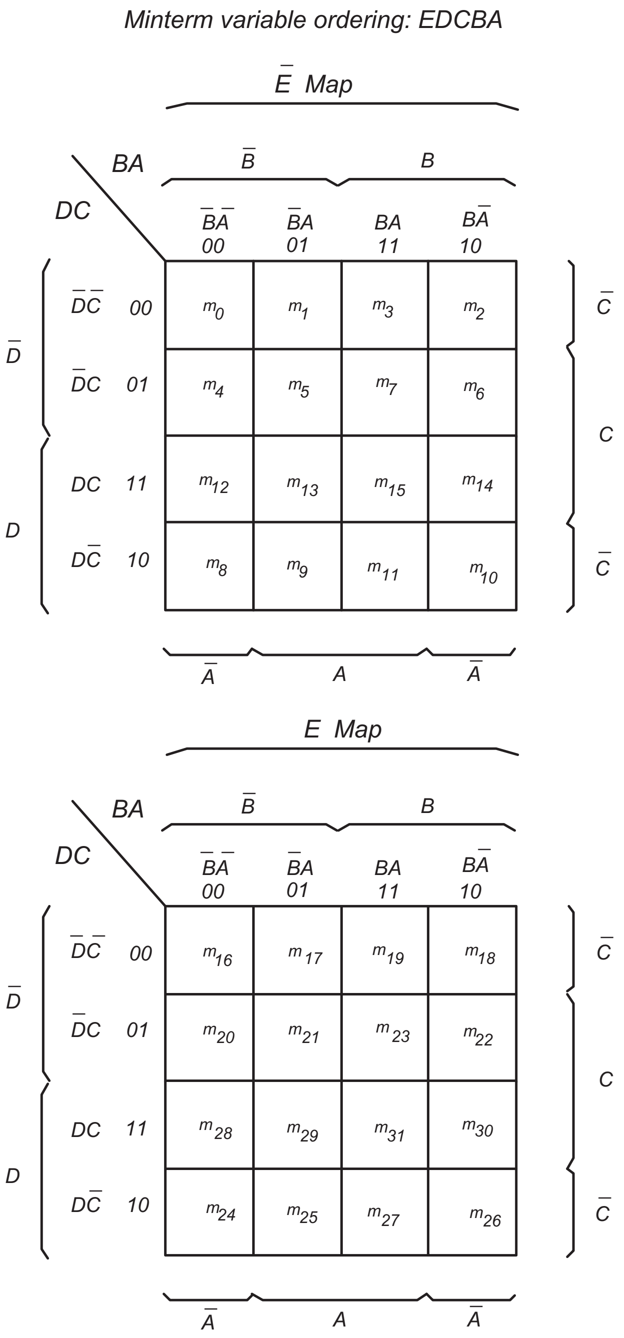


Figure 7.13 Five-variable Karnaugh map definition.

Example 7.19 Given the following function:

$$W(E, D, C, B, A) = \sum(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31), \qquad (7.24)$$

find a maximally simplified *SOP* form.

Figure 7.14 shows the minterms of function *W*. Minterms m_0, m_2, m_4, and m_6 encircled on the $\bar{E}$ map yield the simplified term $\bar{E}.\bar{D}.\bar{A}$. Since minterms m_0, m_2, m_4, and m_6 have no adjacent minterms in map E, the term $\bar{E}.\bar{D}.\bar{A}$ has

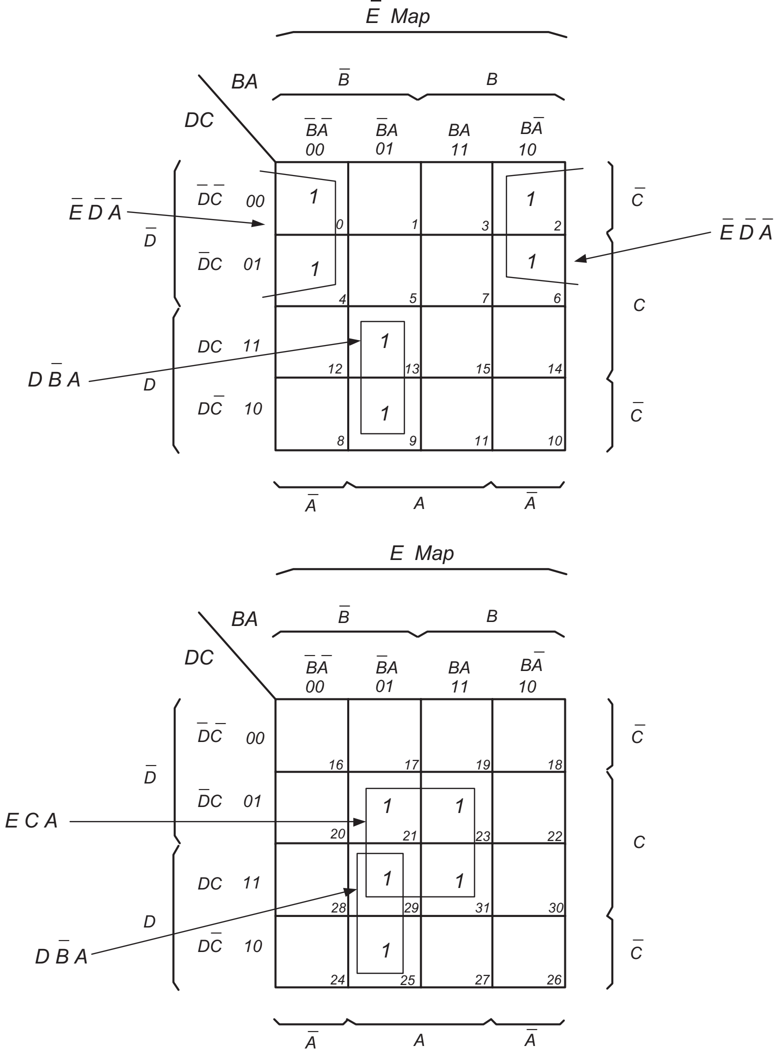


Figure 7.14 Five-variable map for Example 7.19.

a leading $\bar{E}$. Adjacent terms to the $\bar{E}$ map minterms m_0, m_2, m_4, and m_6 are E map minterms m_{16}, m_{18}, m_{20}, and m_{22}, but these last four minterms have zeroes (blank cells) on the E map. The first simplified term of function W is $\bar{E}.\bar{D}.\bar{A}$. The next group of minterms on the $\bar{E}$ map is m_{13} and m_9. The correspondingly adjacent group on the E map is m_{29} and m_{25}. For each of the maps, this group is represented by the term $D.\bar{B}.A$. Since this group is present on both maps, $D.\bar{B}.A$ does not have either a leading E or $\bar{E}$ literal. Alternatively, consider that the $D.\bar{B}.A$ group on the $\bar{E}$ map is annotated as $\bar{E}.D.\bar{B}.A$; also consider the $D.\bar{B}.A$ group on the E map is annotated as $E.D.\bar{B}.A$. Now in the final expression we would have to write:

$$\bar{E}.D.\bar{B}.A + E.D.\bar{B}.A, \tag{7.25a}$$

as part of the overall simplified expression for function W. But from Equation (7.25a), it is easy to observe that:

$$\bar{E}.D.\bar{B}.A + E.D.\bar{B}.A = (\bar{E} + E)(D.\bar{B}.A). \tag{7.25b}$$

Since from Table 7.11 we know that:

$$\bar{E} + E = 1. \tag{7.26}$$

Thus, Equation (7.25b) becomes:

$$D.\bar{B}.A. \tag{7.27}$$

Equation (7.27) is the second term of simplified function W, as shown by (7.28).

Finally, the third and last term of simplified W is obtained from the E map. Grouping minterms m_{21}, m_{23}, m_{29}, and m_{31}. This simplification turns out to be $C.A$ and since it is only present on the E map, the complete term is $E.C.A$, which is the third and last term of the maximally simplified *SOP* form of function W, as shown by (7.28).

The complete maximally simplified function W given by Equation (7.24) is then:

$$W(E, D, C, B, A) = \bar{E}.\bar{D}.\bar{A} + D.\bar{B}.A + E.C.A. \tag{7.28}$$

7.7 PRODUCT OF SUMS SIMPLIFICATIONS

All Karnaugh map simplifications covered so far yielded a *simplified sum of products* form (*SOP*). When we want to produce a *simplified product of sums* form (*POS*) some changes need to be taken into account. Let us address those with the next example.

Example 7.20 Given function:

$$f(D,C,B,A)=\sum(3,4,6,7,11,12,13,14,15) \tag{7.29}$$

The ones marked in Figure 7.15a represent the minterms of function *f.* The cells marked with zeros (actually cells left blank) are all the minterms not included in *f.*

As usual we can obtain a simplified *SOP* form for *f* and this is:

$$f(D,C,B,A)=\bar{C}.A+D.C+B.A. \tag{7.30}$$

Equation (7.30) is obtained with the map of Figure 7.15a.

Now let us consider the map of *f* complement (or simply $\bar{f}$). Refer to Figure 7.15b.

From this figure we obtain a simplified *SOP* form for $\bar{f}$, which is:

$$\bar{f}(D,C,B,A)=\bar{C}.\bar{A}+\bar{C}.\bar{B}+\bar{D}.\bar{B}.A. \tag{7.31}$$

Now let us take the complement of Equation (7.31) and applying DeMorgan rules we obtain:

$$\overline{\overline{f(D,C,B,A)}}=f(D,C,B,A)=(C+A)(C+B)(D+B+\bar{A}). \tag{7.32}$$

Equation (7.32) is a maximally *simplified product of sums* for function *f(D, C, B, A).*

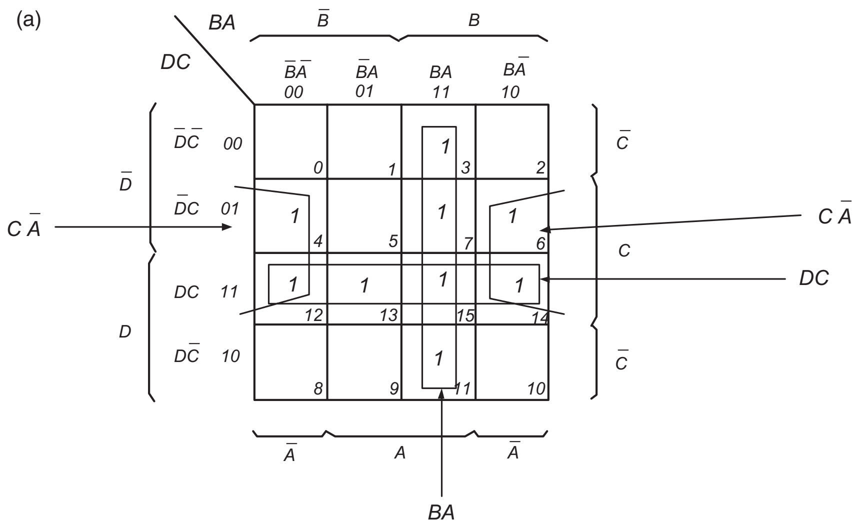
Figure 7.15 depicts the maps for Example 7.20.

7.8 DON'T CARE CONDITIONS

When a logic circuit is designed, we obtain its truth table and we transform the standard sum of products form into a simplified sum of products or product of sums form.

The assumption always has been up until now, that all minterms were defined. This means that minterms were either one's or zero's. There are some applications where not all the possible binary combinations that a number of binary-valued variables spanned are actually used. When this is the case, it is convenient to define the unused binary combination as a third and not previously defined state. We call such state a *don't care.* A *don't care* is typically represented with an *X*. The advantage of defining this *don't care* is convenient because the logic simplification can lead to an easier and more compact simplified *SOP* or *POS* form. Let us address this with an example.

Assume that we have a *4*-bit binary-coded-decimal (*BCD)* number. A single digit *4*-bit *BCD* number ranges from $0000_2=0_{10}$ to $1001_2=9_{10}$. If one wants to



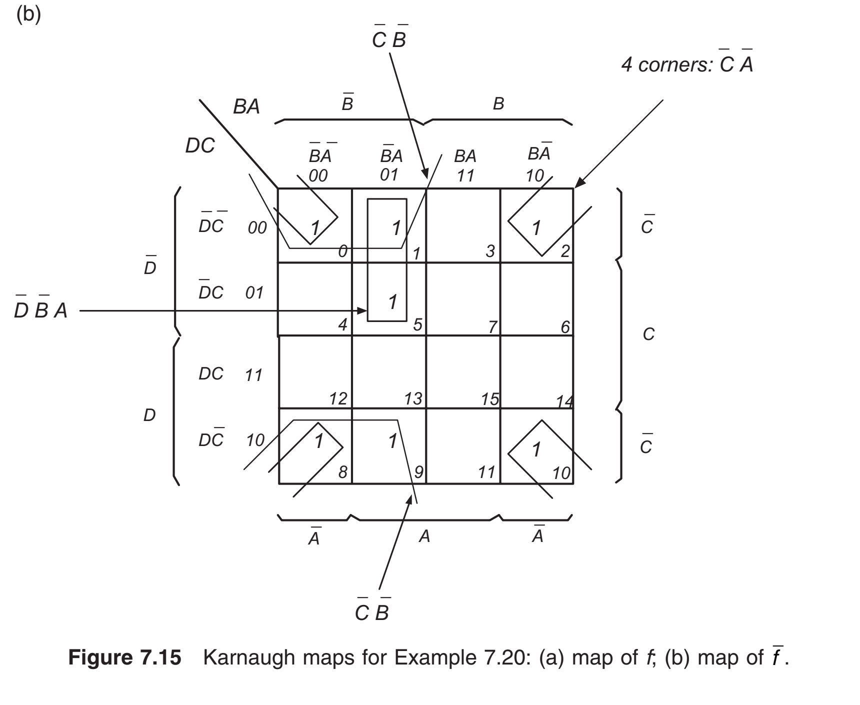


Figure 7.15 Karnaugh maps for Example 7.20: (a) map of f; (b) map of $\bar{f}$.

Table 7.16 Some Binary Coded Decimal Numbers and their decimal and binary representations

BCD	Decimal	Binary
0000_0000	0	0000_0000
0000_0001	1	0000_0001
0000_0010	2	0000_0010
0000_0011	3	0000_0011
0000_0100	4	0000_0100
0000_0101	5	0000_0101
0000_0110	6	0000_0110
0000_0111	7	0000_0111
0000_1000	8	0000_1000
0000_1001	9	0000_1001
0001_0000	10	0000_1010
0001_0001	11	0000_1011
0001_0010	12	0000_1100
⋮	⋮	⋮
0001_1000	18	0001_0010
0001_1001	19	0001_0011
0010_0000	20	0001_0100
0010_0001	21	0001_0101
0010_0010	22	0001_0110

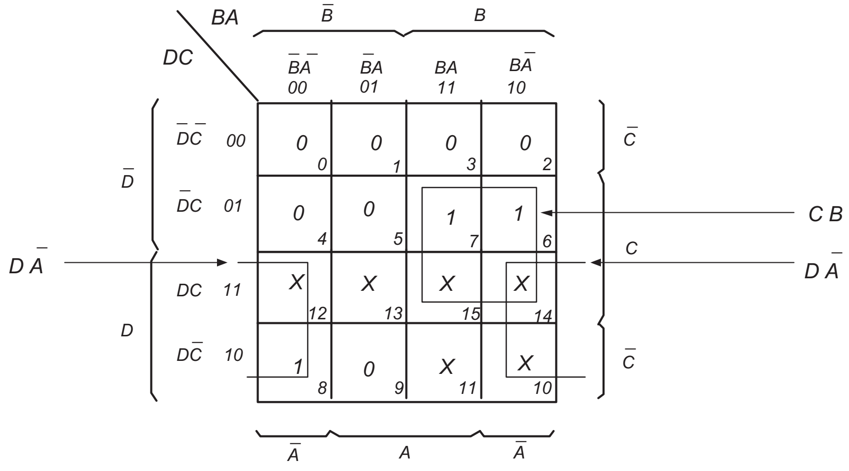
express the number 10_{10} in *BCD*, one needs an extra *BCD* digit (a new *4*-bit set) to represent $10_{10} = 0001\_1001_{BCD}$. Table 7.16 lists some double digit *BCD* numbers, followed by their decimal and binary representations.

Example 7.21 Design a BCD digit range detector: Our problem assumes that we want to design a combinational circuit that receives as input a single-digit *4*-bit *BCD* number. When the BCD digit is either 6, 7, or 8, we want the output of the range detector circuit to be a "*1,*" else we want such output to be "*0.*" We want to come up with a truth table for the range detector circuit. Additionally provide a maximally simplified *SOP* form for the designed range detector circuit. Based on the requirements the truth table follows below. We define the BCD digit having bits *DCBA,* where *A* is the *LSB* and *D* is the *MSB*. Note that the problem implicitly assumes that the six *4*-bit binary combinations *1010* through *1111* will not be present at the inputs; refer to last six rows of Table 7.17.

From the truth table of Table 7.17 we can easily start filling out a four-variable *Karnaugh* map with the values of output F. Figure 7.16 depicts the four-variable map for our *BDC* range detector. First, carefully observe the six *don't cares* on minterms m_{10} through m_{15}. As expected minterms m_6, m_7, and m_8 are 1's. All other minterms are 0's.

Table 7.17 Truth table for a *BCD* range detector

Input D	Input C	Input B	Input A	Output F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X
X	X	X	X	X



$$\begin{array}{c|cccc} & 00 & 01 & 11 & 10 \\ \hline 00 & 0_{0} & 0_{1} & 0_{3} & 0_{2} \\ 01 & 0_{4} & 0_{5} & 1_{7} & 1_{6} \\ 11 & X_{12} & X_{13} & X_{15} & X_{14} \\ 10 & 1_{8} & 0_{9} & X_{11} & X_{10} \end{array}$$

Figure 7.16 Four-variable for map for *BCD* range detector.

Here comes the most important part about simplifying with *don't care* conditions.

Since the *don't care* minterms will never be present at the *DCBA* inputs, it is to the designer's advantage, to most conveniently adopt either a value of 1 or 0 for the *don't cares* in such way that it maximally simplifies the terms to be encircled.

Figure 7.16 shows a possible way of encircling cells. For those *don't cares* that end up within the picked enclosures of minterms we assume that they are valued as 1's. For all other *don't cares* we assume they are 0's. However, in doing that we still do not change the *don't care* notation on the *Karnaugh* map, that is, we leave the *X*.

The maximally *SOP* form for the *BCD* range detector is:

$$F(D, C, B, A) = C.B + D.\bar{A}. \tag{7.33}$$

Exercise: Try other encirclements selecting other don't cares and compare your results against Equation (7.33). What can you tell about your findings?

7.9 LOGIC GATES: ELECTRICAL AND TIMING CHARACTERISTICS

Logic gates are available in integrated circuit packages or as macros or combinational logic building blocks in Application Specific Integrated Circuits (ASICs), Complex Programmable Logic Devices (CPLDs), Field Programmable Gate Arrays (FPGAs), and other devices. Figure 7.17 depicts the most common schematic symbols of the most commonly used logic gates.

All of the above gates are conceptually and sometimes physically available with more than two inputs. There may be three, four, and more inputs in a gate. Using DeMorgan's rules we will justify the logic equivalences given in Figure 7.17. For a positive *AND* gate, *A ANDed* with *B* is *A.B*. From DeMorgan's rule (Eq. 7.3)

$$\overline{A.B} = \bar{A} + \bar{B}. \tag{7.34}$$

Equation (7.34) justifies the logic equivalence between Figure 7.17c,d.

Complementing Equation (7.34) yields:

$$AB = \overline{\overline{A.B}} = \overline{\bar{A} + \bar{B}}. \tag{7.35}$$

Equation (7.35) justifies the logic equivalence of Figure 7.17a,b.

From the other DeMorgan rule (Eq. 7.2) we have that:

$$\overline{A + B} = \bar{A}\bar{B}. \tag{7.36}$$

Equation (7.36) justifies the logic equivalence between Figure 7.17g,h. Now complementing Equation (7.36) yields:

$$\overline{\overline{A + B}} = A + B = \overline{\bar{A}.\bar{B}}. \tag{7.37}$$

Equation (7.37) justifies the logic equivalence between Figure 7.17e,f. Note that neither (i) nor (j) are logically equivalent. The same is true for (k) and (l). Figure 7.17i is the logic complement of Figure 7.17j. So is Figure 7.17k,l.

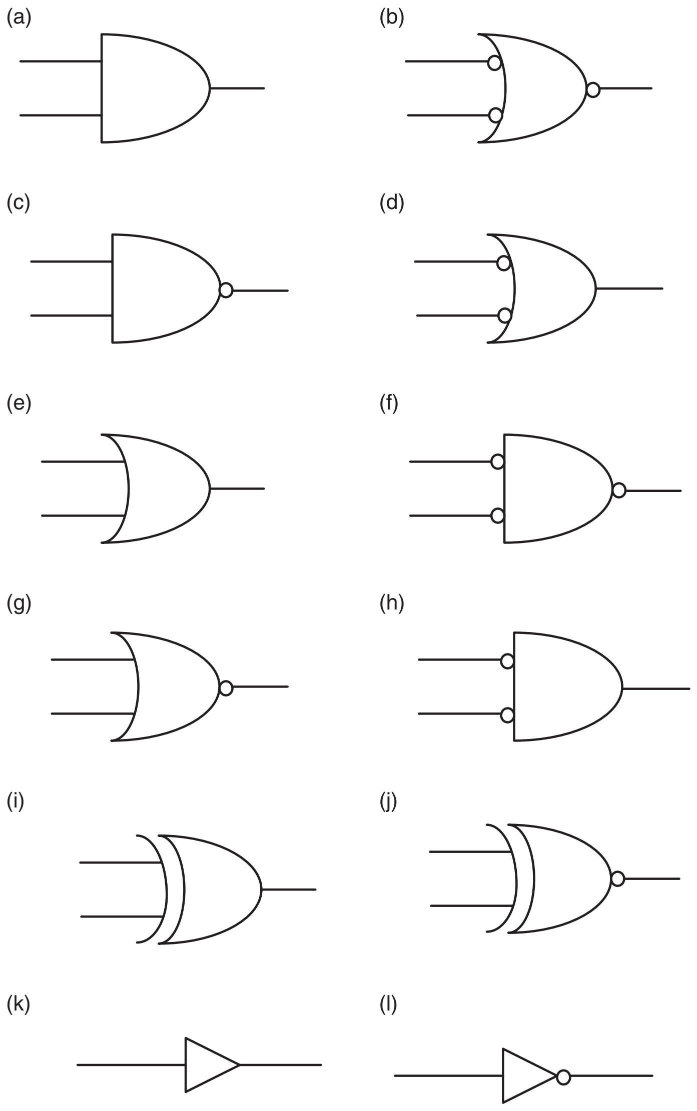


Figure 7.17 (a) Positive *AND* gate; (b) negative *OR* gate; (c) *NAND* gate; (d) DeMorganized *NAND* gate; (e) positive OR gate; (f) negative *AND*; (g) *NOR* gate; (h) DeMorganized *NOR* gate; (i) Exclusive *OR* gate; (j) Exclusive *NOR* gate or Equivalence gate; (k) buffer, no inversion; (i) inverting buffer or inverter.

7.9.1 Gates Key Electrical Characteristics

For the sake of brevity we will only consider gates that operate with 3.3 V TTL logic levels. TTL or *Transistor-Transistor-Logic* is a class of digital circuits built with *bipolar* transistors and resistors. TTL became at one point in time the most widespread logic family used in computers and almost all other electronic equipment. Within our context TTL is used to mean TTL-compatible-logic-levels. The actual logic implementation may not necessarily be TTL, it just means that its input and output logic levels comply with the TTL family of integrated circuits levels. Other families of integrated circuits are *CMOS* and *ECL*. Today one can say that CMOS is the most widespread logic family of integrated circuits.

Table 7.18 defines the voltage logic levels for a *zero (low voltage level)* input and output and for a *one (high voltage level)* input and output. Note that expressing the state of an input or an output with a *voltage level* makes it independent as to whether the application that uses such gate or circuit with high true or low true signals.

Table 7.18 is a simplified real-device data sheet characteristics for the reader's convenience. Figure 7.18 depicts a gate output driving another gate input. Both the high and the low levels are shown. Let us concentrate on the logic

Table 7.18 Some electrical characteristics of low voltage TTL (LVT)

			Limits Temperature Range: –40°C to +85°C			
Symbol	Parameter	Test Conditions	MIN	TYP	MAX	Units
Recommended operating conditions						
V_{CC}	DC supply voltage		2.7		3.3	V
V_{IH}	High-Level input voltage		2.0			V
V_{IL}	Low-level input voltage				0.8	V
I_{OH}	High-level output current				–20	mA
I_{OL}	Low-level output current				32	mA
Electrical characteristics						
V_{OH}	High-level output voltage	$V_{CC} = 2.7$ V, $I_{OH} = -6$ mA	2.4			V
V_{OL}	Low-level output voltage	$V_{CC} = 2.7$ V, $I_{OL} = 32$ mA			0.4	V
I_L	Input leakage current applies to I_{IH} and I_{IL}	$V_{CC} = 3.6$V, $V_I = V_{CC}$ or GND (0 V)			±1	μA

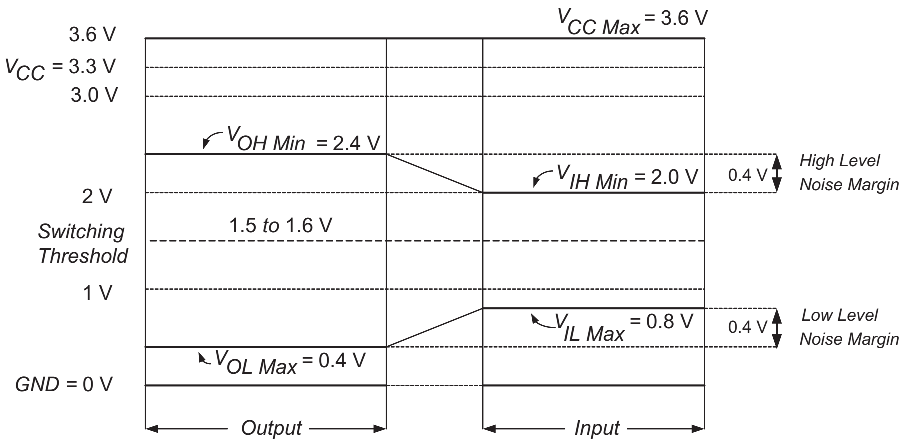


Figure 7.18 TTL output driving a TTL input.

zero or the low level. When a gate output drives a *low level* to the input of another gate, the output must not exceed $V_{OL\ MAX}$ voltage level which is 0.4 V for TTL compatible logic. While at the same time the input gate must be capable of accepting a *low level* that does not exceed a maximum level of $V_{IL\ MAX}$ of 0.8 V. Note that the difference between $V_{IL\ MAX}$ and $V_{OL\ MAX}$ is actually 0.4 V (400 mV), and it is referred to as the low-level noise margin for TTL-compatible logic. Similarly, when the output of a gate drives a *high level* to the input of another gate, the output must not be below $V_{OH\ MIN}$ of 2.4 V. It is also the case that a high output driving an input also has a 400 mV noise margin. The noise margin is a desirable voltage to have to account for system noise, power supply ripple, and other sources of noise that can couple onto the driving and the receiving lines of each gate.

Now what about the current specifications? That is, I_{OH}, I_{OL}, I_{IH}, and I_{IL}? When an output is at a high voltage level, the driving gate *sources* a current to the input gate, the sourced current flows outward from the output. Conventionally, this current is negative (refer to the I_{OH} entry in Table 7.18). When an output is at a low voltage level, the driving gate *sinks* current and sunk current conventionally has a positive sign (refer to the I_{OL} entry in Table 7.18). I_{IH} is the current into an input terminal when a specified high voltage level is applied to it. I_{IL} is the current into an input terminal when a specified low voltage level is applied to it. I_{IH} and I_{IL} are typically found only on devices with bipolar inputs and that significantly have different levels of pull-down current to provide a logic low and pull-up current to provide a logic high. CMOS devices, however, just have an I_L or a leakage current at the input. Such levels of I_L are measured at both low and high bias conditions. Figure 7.18 depicts an output driving an input, indicating all the voltage levels.

For TTL logic levels, the switching threshold is around 1.5–1.6 V.

7.9.2 Gates Key Timing Characteristics

When a digital input signal is applied to a combinational circuit the output will not respond (or change) until the combinational circuit time-propagation delay elapses.

JEDEC, the Joint Electronic Device Engineering Council, is the semiconductor engineering standardization body of the Electronics Industries Alliance, a trade association that represents all areas of industry. JEDEC defines the *propagation delay time* as the time specified between reference points on the input and output voltage waveforms with the output changing from one defined level (either *high* or *low*) to the other defined level. Thus, there will be a t_{PHL} and a t_{PLH}, respectively, a high-to-low propagation delay (t_{PD}), and a low-to-high propagation delay. The maximum value of $\mathrm{t_{PD}}$ simply is the worst-case or longest case of t_{PHL} and a t_{PLH}. Figure 7.19 depicts the propagation delay time that exists in a *LVTTL* combinational circuit, or simply just a LVTTL gate between an input and an output. Output *1* depicts both low-to-high and high-to-low t_{PD}. Output *2* depicts the same delays assuming it is the complement of Output *1*. Note that the time references are measured at 1.5 V, or about half of the 3.3 V power supply rail. A 1.5 V is referred to as the LVTTL logic switching threshold. Although the switching threshold of the logic may vary, perhaps as much as ±0.5 V or more, what matters is that all the timing measurements be made consistently with respect to the same 1.5-V reference level.

Examples of t_{PD} of integrated circuit gate delays are anywhere around 10 ns (older TTL technology) to as little as a fraction of a nanosecond (for high speed CMOS technologies and ECL). Reference 5 in the Further Reading section has a discussion on TTL, CMOS, and ECL families of integrated circuits.

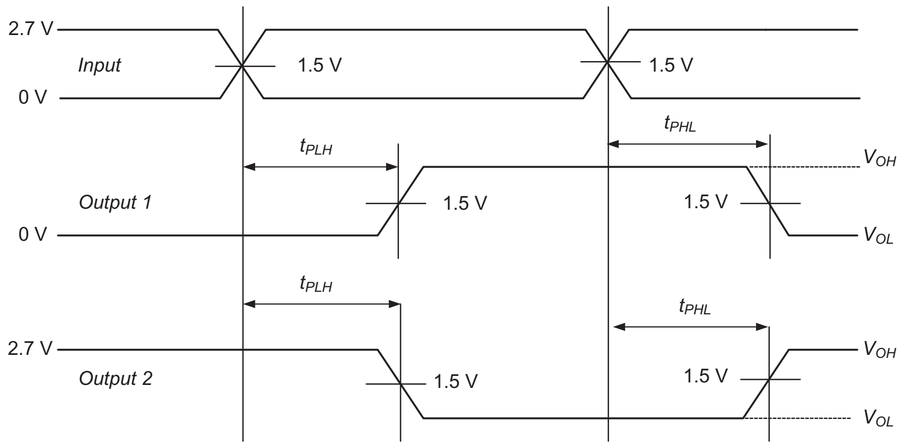


Figure 7.19 Logic gate propagation delay times.

7.10 SUMMARY

This chapter introduced the reader to combinational circuits, which are also referred to as circuits with no memory capability. Binary numbers were presented along with the essential elements of switching or Boolean algebra.

Standard or canonical SOP and POS forms are ways of representing logic functions. For the purpose of logic implementation, where usually we want the number of gates to be reduced as well as the number of inputs per gate to obtain simplified SOP and POS. Methods of simplification were presented by covering *2* through *5*-variable *Karnaugh* maps. *K* maps of *6* or more variables become somewhat impractical to use effectively. We will address other logic design techniques to overcome using huge *K*. maps. Finally, we studied the most basic electrical and timing characteristics of logic gates. The examples were centered around TTL-compatible logic levels gates, not necessarily implemented in TTL technology. Although it is completely true that TTL technology is obsolete, other logic families, like CMOS and BiCMOS, have adopted TTL levels to interface to the many devices, such as line drivers, that continue to use TTL levels.

FURTHER READING

1. M. Morris Mano, *Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1984.
2. John F. Wakerly, *Digital Design: Principles and Practices*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 2001.
3. Davis Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 2007.
4. Charles H. Roth Jr., *Logic Design*, 2nd ed., West Publishing Company, St. Paul, MN, 1979.
5. Ed Lipiansky, *Embedded Systems Hardware for Software Engineers*, McGraw-Hill, New York, 2011.

PROBLEMS

7.1 Convert the following 16-bit positive binary numbers to decimal:

(a) 1101_1111_1010_0001

(b) 1111_1111_1111_1111

(c) 1000_0000_0000_0000

(d) 1000_1000_1000_1000

(e) 1001_0110_1100_0111

7.2 Convert the following 16-bit 2’s complement numbers to decimal:

(a) 1101_1111_1010_0001

(b) 1111_1111_1111_1111

(c) 1000_0000_0000_0000

(d) 1000_1000_1000_1000

(e) 1001_0110_1100_0111

7.3 Convert the following 4-BCD-digit (16-bits) numbers to decimal:

(a) 1101_1111_1010_0001

(b) 1111_1111_1111_1111

(c) 1000_0000_0000_0000

(d) 1000_1000_1000_1000

(e) 1001_0110_1100_0111

7.4 Convert the following decimal numbers into 16-bit 2's complement.

(a) 1537

(b) –10418

(c) 32700

(d) 0

(e) –32700

7.5 Using De Morgan's rules, find simplified logic equivalent Boolean expressions:

(a) $\overline{A+B+C}$

(b) $\bar{A}.\overline{C+B}$

(c) $\bar{A}.\bar{B}+A.\bar{B}+\bar{A}.B+A.B$

(d) $A.\bar{B}.\bar{C}.\bar{D}+A.\bar{B}.\bar{C}.D+A.\bar{B}.C.D$

7.6 Express the following functions in SOP and POS canonical forms:

(a) $F=D(\overline{A+B})+\bar{B}D$

(b) $F=\bar{y}z+wx\bar{y}+wx\bar{z}+\bar{w}\bar{x}z$

(c) $F=(\bar{A}+B)(\bar{B}+C)$

(d) $F=1$

(e) $F=(xy+z)(y+xz)$

7.7 Using Karnaugh maps find simplified sum-of-product forms for the following logic functions:

(a) $f(A,B)=\Sigma(0,1,3)$

(b) $g(A,B,C,D)=\Sigma(0,1,4,58,9)$

(c) $h(A,B,C,D)=\Sigma(0,1,2,3,8,10,11,15)$

(d) $k(A,B,C,D)=\Sigma(1,3,4,7,8,9,10,11,14)$

7.8 Using Karnaugh maps find simplified product-of-sum forms for the following logic functions:

(a) $f(A,B)=\prod(1,2,3)$

(b) $g(A,B,C,D)=\prod(0,1,4,5,8,9)$

(c) $h(A,B,C,D)=\prod(1,2,5,6,9,12,13,14)$

(d) $k(A,B,C,D)=\prod(1,3,4,7,8,9,10,11,14)$

7.9 Graphically depict a 3-variable XOR using Venn Diagrams.

7.10 Graphically depict a 4-variable XNOR using Venn Diagrams.

7.11 Generate the truth table of a 4-variable XOR function.

7.12 Obtain the truth table for the following Boolean function: $F(X,Y,Z)=X.Y+X.\bar{Y}+\bar{Y}.Z$.

7.13 Write the sum-of-products form of a 3-variable XOR.

7.14 Write the product-of-sums form of a 3-variable XOR.

7.15 Obtain the truth table of a 4-variable majority logic circuit.
That is, majority is obtained whenever two or more variables are true, else majority is false.

7.16 Create a 2-level logic implementation of the majority function obtained in Problem 7.15.

7.17 Assume that you have an inverting gate with a 10 ns high-to-low and low-to-high propagation delay. If you connect the output of this gate to its input with a zero-delay wire, sketch the waveform that you would see with an oscilloscope. An oscilloscope is an instrument that allows one to visualize how an electric waveform varies with respect to time.

7.18 Given logic gates which all have a high-to-low and low-to-high 20-ns propagation delay, what is the maximum propagation delay of a function implemented in three levels of logic.

7.19 Given logic gates which all have a high-to-low and low-to-high 20-ns propagation delay, what is the maximum propagation delay of a function implemented in four levels of logic.

7.20 From doing Problems 7.18 and 7.19 what can you generalize when a logic function is implemented with more levels of logic? Clearly the propagation delay increases linearly with each new level of logic.