CHAPTER 2

Review of Combinational Logic Design

This chapter will review manual methods for designing combinational logic. In Chapter 6 we will see how these steps can be automated with modern design tools.

2.1 Combinational Logic and Boolean Algebra

Combinational logic forms its outputs as Boolean functions of its input variables on an instantaneous basis. That is, at any time t the outputs y_1, y_2, and y_3 in Figure 2-1 depend on only the values of a, b, c, and d at time t. The outputs of combinational logic at any time t are a function of only the inputs at time t. The outputs of other circuits may depend on the history of the inputs up to time t, and they are called sequential circuits. Sequential circuits require memory elements in hardware.

The variables in a logic circuit are binary—they may have a value of 0 or 1. Hardware implementations of logic circuits use either positive logic, in which a high voltage level, say 5 volts, corresponds to a logical value of 1, and a low voltage, say 0, corresponds to a logical 0. In negative logic, a low electrical level corresponds to a logical 1, and a high electrical level corresponds to a 0.

Some common logic gates are shown in Figure 2-2, together with the Boolean equation that determines the value of the output of the gate as a function of its inputs, and Table 2-1 lists common symbols for hardware-based Boolean logic operations.[1]

2.1.1 ASIC Library Cells

Logic gates are implemented physically by a transistor-level circuit. For example, in CMOS (complementary metal-oxide semiconductor) technology, a logic inverter consists

[1] *Note*: The schematic symbol for the three-state buffer uses the symbol z to indicate the high impedance condition of the device.

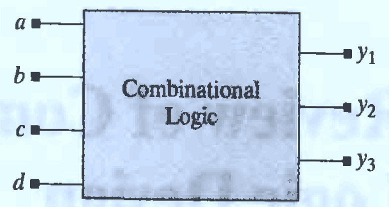

FIGURE 2-1 Block diagram symbol for combinational logic having four inputs and three outputs.

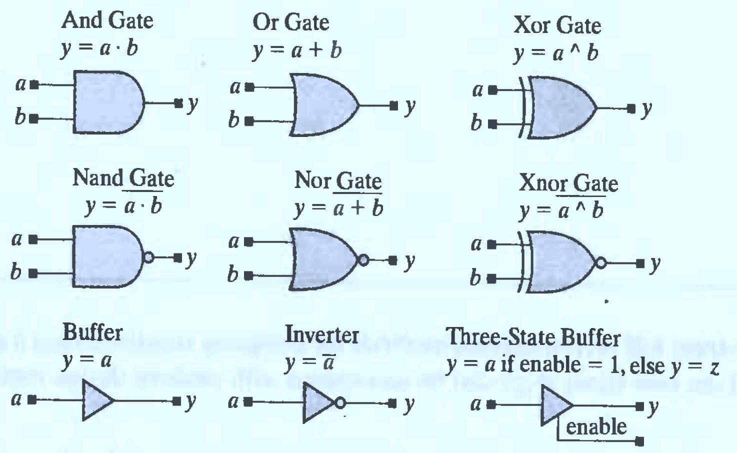

FIGURE 2-2 Schematic symbols and Boolean relationships for some common logic gates.

TABLE 2-1 Common Boolean logic symbols and operations.

Symbol	Logic Operation
+	Logic "or"
·	Logical "and"
⊕	Exclusive "or"
^	Exclusive "or"
'	Logical negation
–	Logical negation (overbar)

of a series connection of *p*-channel and *n*-channel MOS transistors having a common drain that serves as the output, and a common gate that serves as the input. When the input is low, the *p*-channel device conducts and the *n*-channel device is an open circuit. In this mode the output capacitor charges to V_{dd}. When the input is high, the *n*-channel device conducts, and the *p*-channel device is an open circuit. This discharges the output node capacitor to ground. Figure 2-3 shows (b) the pull-up and (c) pull-down paths for current in the inverter in (a).

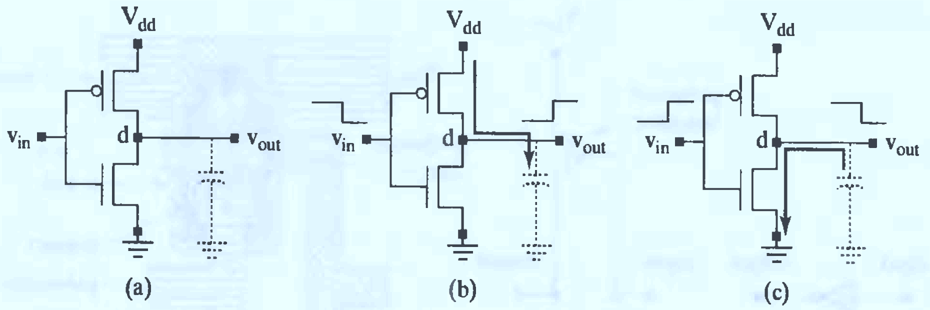

FIGURE 2-3 CMOS transistor-level schematics: (a) inverter with output load capacitance, (b) inverter with pull-up (charging) signal paths, and (c) inverter with pull-down (discharging) signal paths.

Other logic gates can be implemented using the same basic principles of pull-up and pull-down logic. Figure 2-4 shows the transistor-level schematic for a three-input NAND gate. If one or more of the inputs is low, the output node Y is pulled up to V_{dd}; all inputs must be high to pull the output to ground.

Very-large-scale integrated (VLSI) circuits that implement logic gates are fabricated by a series of processing steps in which photomasks are used to selectively dope a silicon wafer to form and connect transistors. Figure 2-5 shows a composite view of the basic masks used in an elementary process to fabricate a CMOS inverter by implanting semiconductor dopants and depositing metal and polycrystalline silicon. The masking steps are performed in a well-defined sequence, beginning with the implantation of a dopant to form the *n*-well, a region that is heavily doped with an n-type material (e.g., arsenic). A p-channel transistor is formed in the n-well by implanting a p-type (e.g., boron) source and drain regions, and an n-channel transistor is formed in the host silicon substrate. Polycrystalline silicon is deposited to form the gates of the transistors, and metal is deposited to form interconnections in and between devices. The actual processes involve many more steps and can have several more layers of metal than these simple structures. Figure 2-6 shows a cross section of a simple ASIC cell for an inverter, revealing the doped regions.

Circuits that implement basic and moderately complex Boolean functions are characterized for their functional, electrical, and timing properties, and packaged in

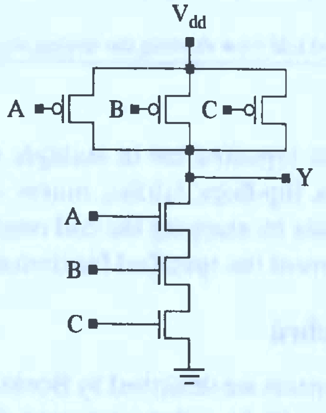

FIGURE 2-4 Transistor-level schematic for three-input CMOS Nand gate.

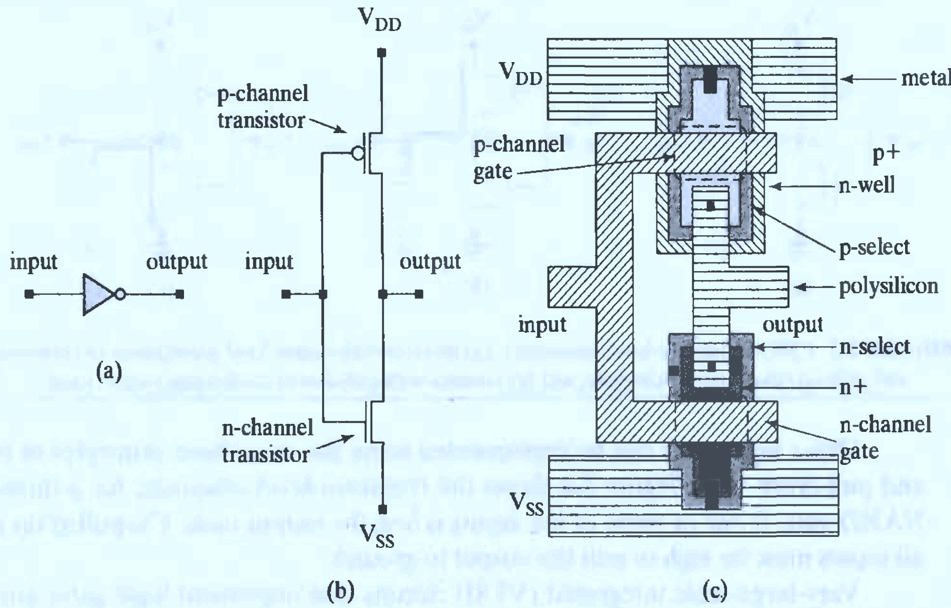

FIGURE 2-5 Views of a CMOS inverter: (a) circuit-symbol view, (b) transistor-schematic view, and (c) simplified composite fabrication mask view.

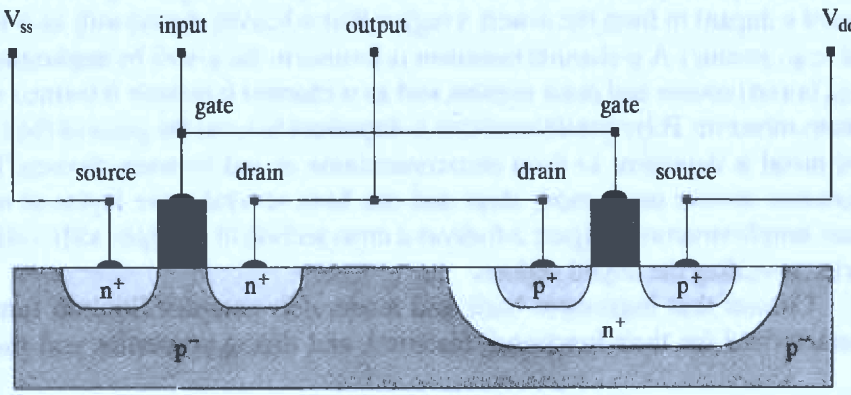

FIGURE 2-6 Simplified side view showing the doping regions of a CMOS inverter.

standard-cell libraries for repeated use in multiple designs. Such libraries commonly contain basic logic gates, flip-flops, latches, muxes, and adders. Synthesis tools build complex integrated circuits by mapping the end result of logic synthesis onto the parts of a cell library to implement the specified functionality with acceptable performance.

2.1.2 Boolean Algebra

The operations of logic circuits are described by Boolean algebra. A Boolean algebra consists of a set of values $\mathbf{B} = \{0, 1\}$ and the operators "+" and "·". The operator "+" is

TABLE 2-2 Laws of Boolean algebra.

Laws of Boolean Algebra	SOP Form	POS Form
Combinations with 0, 1	$a + 0 = a$	$a \cdot 1 = a$
	$a + 1 = 1$	$a \cdot 0 = 0$
Commutative	$a + b = b + a$	$ab = ba$
Associative	$(a + b) + c = a + (b + c)$ $= a + b + c$	$(ab)c = a(bc) = abc$
Distributive	$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$
Idempote	$a + a = a$	$a \cdot a = a$
Involution	$(a')' = a$	
Complementarity	$a + a' = 1$	$a \cdot a' = 0$

called the sum operator, the "OR" operator, or the disjunction operator. The operator "$\cdot$" is called the product operator, the "AND" operator, or the conjunction operator. The operators in a Boolean algebra have commutative and distributive properties such that for two Boolean variables A and B having values in $\mathbf{B}$, $a + b = b + a$, and $a \cdot b = b \cdot a$. The operators "+" and "$\cdot$" have identity elements 0 and 1, respectively, such that for any Boolean variable a, $a + 0 = a$, and $a \cdot 1 = a$. Each Boolean variable a has a complement, denoted by a', such that $a + a' = 1$, and $a \cdot a' = 0$. Table 2-2 summarizes the laws of Boolean algebra for sum-of-products (SOP) and product-of-sums (POS) Boolean expressions (more on this later). For simplicity, we have omitted showing the "$\cdot$" operator and will do so freely in the remaining examples.

A multidimensional space spanned by a set of n Boolean variables is denoted by $\mathbf{B}^n$. A point in $\mathbf{B}^n$ is called a *vertex* and is represented by an n-dimensional vector of binary valued elements, for example, (100). A binary variable can be associated with the dimensions of a Boolean space, and a point is identified with the values of the variables. A Boolean variable is represented symbolically by a literal, such as a. A literal is an instance (e.g., a) of a variable or its complement (e.g., a'). Boolean expressions are formed by strings of literals and Boolean operators. A product of literals, such as $ab'c$ is a *cube*. A cube is associated with a set of vertices, and a cube is said to "contain" one or more vertices. Figure 2-7 illustrates how each point in $\mathbf{B}^3$ can be represented (a) by a vector of binary values (e.g., its coordinates), and (b) by a cube of literals.

A *completely specified* m-dimensional Boolean function with n inputs is a mapping from $\mathbf{B}^n$ into $\mathbf{B}^m$, denoted by f: $\mathbf{B}^n \rightarrow \mathbf{B}^m$. An *incompletely specified* function is defined over a subset of $\mathbf{B}^n$, and is considered to have a value of *don't-care* at points outside the domain of definition: f: $\mathbf{B}^n \rightarrow \{0, 1, *\}$, where * denotes don't-care.

The *On_Set* of a Boolean function consists of the vertices at which the function is asserted (logically true), that is, $On\text{-}Set = \{x: x \in B^n \text{ and } f(x) = 1\}$. The *Off_Set* is the set of vertices at which the function is de-asserted (logically false): $Off\_Set = \{x: x \in B^n \text{ and } f(x) = 0\}$. The *Don't-Care-Set* is the set of vertices at which no significance is attached to the value of the function, so $Don't\text{-}Care\text{-}Set = \{x: x \in B^n \text{and } f(x) = *\}$. The Don't-Care-Set set accommodates input patterns that never occur or outputs that will not be observed.

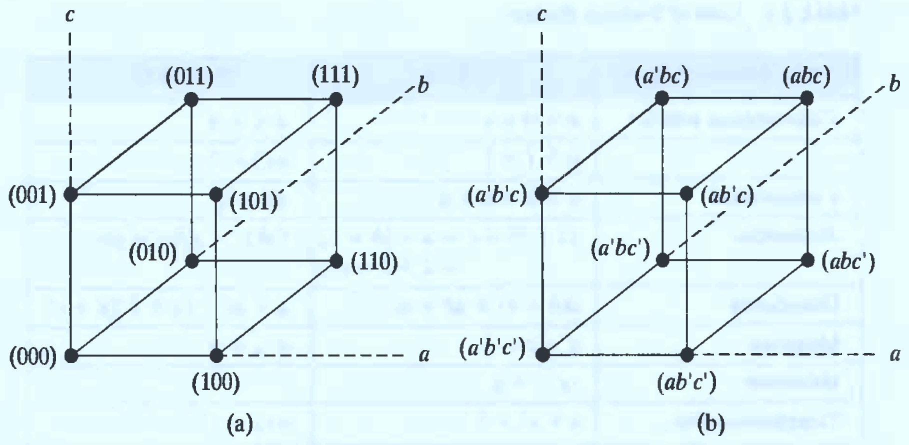

FIGURE 2-7 Points in a Boolean space: (a) represented by vectors of binary variables, and (b) represented symbolically.

2.1.3 DeMorgan's Laws

DeMorgan's laws allow us to transform a circuit from an SOP form to a POS form, and vice versa. The first form of the law specifies the complement of a sum of terms:

$$(a + b + c + \dots)' = a' \cdot b' \cdot c' \cdot \dots$$

For two variables the relationship specifies that:

$$(a + b)' = a' \cdot b'$$

The Venn diagrams in Figure 2-8 illustrate the operations of DeMorgan's laws for two variables.

The second form of DeMorgan's laws specifies the complement of the product of terms:

$$(a \cdot b \cdot c \dots)' = a' + b' + c' + \dots$$

For two variables, the law states that:

$$(a \cdot b)' = a' + b'$$

These relationships are illustrated by the Venn diagrams in Figure 2-9.

2.2 Theorems for Boolean Algebraic Minimization

Important theorems that are used to minimize Boolean algebraic expressions to produce efficient circuit realizations are shown in Figure 2-10, in POS and SOP form. Logical adjacency and the consensus theorem are illustrated by the Venn diagrams in Figure 2-11. The consensus term, *bc*, is redundant because it is covered by the union of *ab* and *a'c*. Laws that apply specifically to the exclusive-or operation are shown in Figure 2-12.

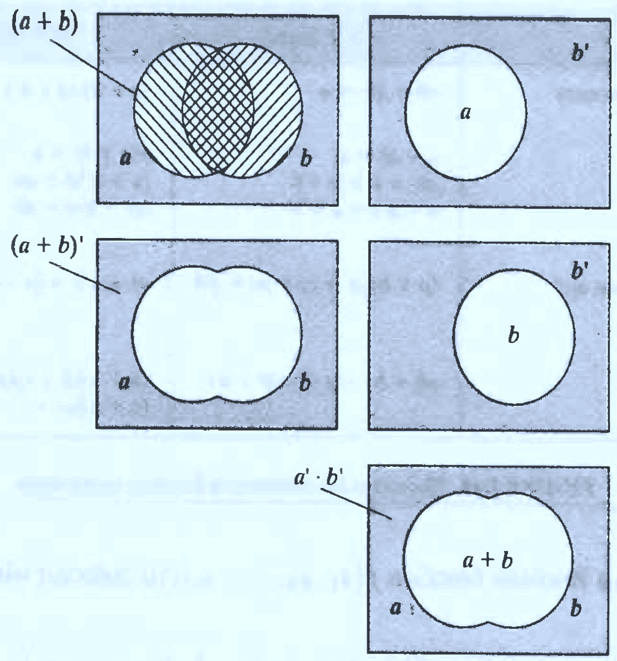

FIGURE 2-8 Venn diagrams illustrating DeMorgan's law: $(a + b)' = a' \cdot b'$.

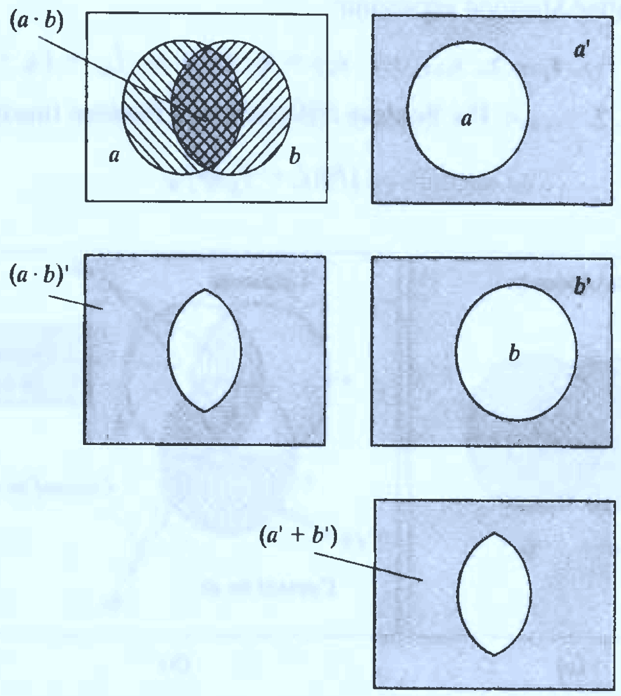

FIGURE 2-9 Venn diagrams illustrating DeMorgan's law: $(a \cdot b)' = a' + b'$.

Theorem	SOP Form	POS Form
Logical Adjacency	$ab + ab' = a$	$(a + b)(a + b') = a$
Absorption	$a + ab = a$	$a(a + b) = a$
	$ab' + b = a + b$	$(a + b')b = ab$
or:	$a + a'b = a + b$	$(a' + b)a = ab$
Multiplication and Factoring	$(a + b)(a' + c) = ac + a'b$	$ab + a'c = (a + c)(a' + b)$
Consensus	$ab + bc + a'c = ab + a'c$	$(a + b)(b + c)(a' + c) = (a + b)(a' + c)$

FIGURE 2-10 Theorems for minimizing Boolean expressions.

Given a Boolean function $f(x_1, x_2, \ldots, x_n)$, its cofactor with respect to variable x_i is

$$f_{xi} = f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n),$$

and its cofactor with respect to $x_{i'}$ is

$$f_{xi'} = f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n).$$

A binary-valued Boolean function can be represented by its cofactors as the following so-called Shannon expansion:

$$f(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = x_i \cdot f_{xi} + x_{i'} \cdot f_{xi'} = (x_i + f_{xi'}) \cdot (x_{i'} + f_{xi})$$

for all $i = 1, 2, \ldots, n$. The Boolean difference of a Boolean function f is given by

$$\partial f / \partial x_i = f_{xi} \oplus f_{xi'}$$

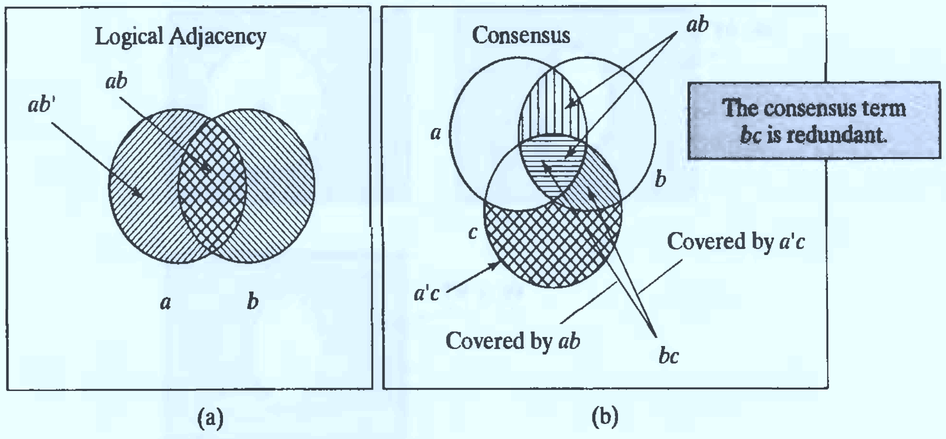

FIGURE 2-11 Venn diagrams: (a) Logical adjacency, and (b) Consensus.

Exclusive-Or Laws	
Combinations with 0, 1	$a \oplus 0 = a$
	$a \oplus 1 = a'$
	$a \oplus a = 0$
	$a \oplus a' = 1$
Commutative	$a \oplus b = b \oplus a$
Associative	$(a \oplus b) \oplus c = a \oplus (b \oplus c) = a \oplus b \oplus c$
Distributive	$a(b \oplus c) = ab \oplus ac$
Complement	$(a \oplus b)' = a \oplus b' = a' \oplus b = ab + a'b'$

FIGURE 2-12 Boolean relationships for the exclusive-or operation.

The Boolean difference of f with respect to x_i determines whether f is sensitive to a change in input variable x_i. This property has utility in algebraic methods to determine whether a test detects a fault in a circuit [1]. Also, a binary tree mapping of a Boolean function can be generated by recursively applying Shannon's expansion [2].

2.3 Representation of Combinational Logic

We will consider three common representations of combinational logic: (a) structural (i.e., gate-level) schematics, (b) truth tables, and (c) Boolean equations. An additional representation, a binary decision diagram (BDD) is a graphical representation of a Boolean function and contains the information needed to implement it [2, 3]. BDDs are used primarily within EDA software tools because they can be more efficient and easier to manipulate than truth tables. They can also be helpful in finding hazard covers [4]. We will not make use of BDDs here, but will rely on truth tables instead.

Example 2.1

The truth table for the combinational logic of a half adder is shown in Figure 2-13. The adder forms a sum and carry out bit from two data bits (without a carry in bit).

The Boolean equations describing the half adder can be derived from the truth table and written in SOP form:

$$sum = a'b + ab' = a \oplus b$$

$$c\_out = a \cdot b$$

End of Example 2.1

Inputs		Outputs	
a	*b*	*c_out*	*sum*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a)

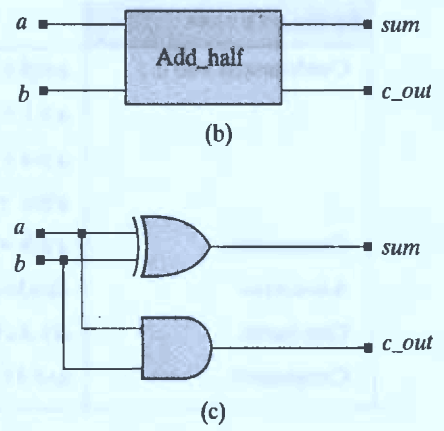

FIGURE 2-13 Half adder: (a) truth table, (b) block diagram symbol, and (c) schematic.

Example 2.2

A full adder forms a sum and carry out bit from two data bits and a carry in bit. The truth table for the combinational logic of a full adder is shown in Figure 2-14.

The Boolean equations describing *sum* and *c_out* bits are given by:

$$sum = a' \cdot b' \cdot c\_in + a' \cdot b \cdot c\_in' + a \cdot b' \cdot c\_in' + a \cdot b \cdot c\_in$$

$$c\_out = a' \cdot b \cdot c\_in + a \cdot b' \cdot c\_in + a \cdot b \cdot c\_in' + a \cdot b \cdot c\_in$$

Inputs			Outputs	
a	*b*	*c_in*	*c_out*	*sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a)

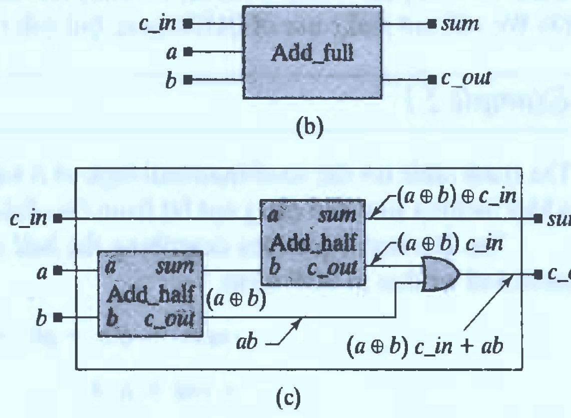

FIGURE 2-14 Full adder: (a) truth table, (b) block diagram symbol, and (c) schematic for a full adder composed of half adders and glue logic.

These can be rearranged as:

$$sum = a \oplus b \oplus c\_in$$

$$c\_out = (a \oplus b) \cdot c\_in + a \cdot b$$

The Venn diagram in Figure 2-15 shows the assertions of *sum* and *c_out* as a function of *a*, *b*, and *c_in*.

Truth tables become unwieldy for functions of several variables because the number of rows grows exponentially with the number of variables. Notice that the truth table of the full adder has twice as many rows as the table for the half adder.

End of Example 2.2

2.3.1 Sum of Products Representation

A cube is formed as the product of literals in which a literal appears in either uncomplemented or complemented form. For example, *ab'cd* is a cube but *ab'cbd* is not. A cube need not contain every literal. A *Boolean expression* is a set of cubes, and is typically expressed in *sum-of-products (SOP)* form as the "OR" of product terms (cubes), rather than in set notation.

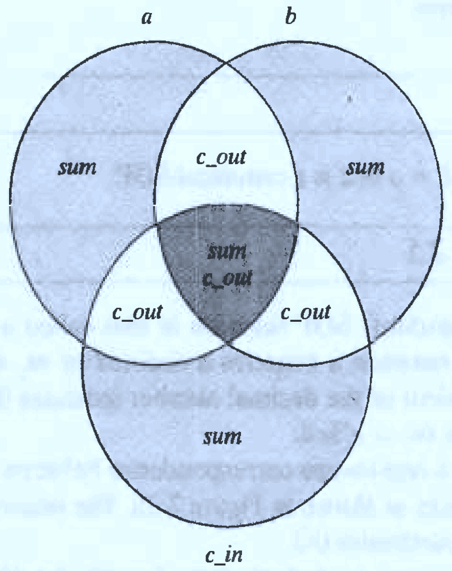

FIGURE 2-15 Venn diagram representation of the truth table for a full adder.

Example 2.3

The following expression is in sum-of-products form: $abc' + bd$.

End of Example 2.3

Each term of a Boolean expression in SOP form is called an *implicant* of the function. A *minterm* is a cube in which every variable appears. The variable will be in either true (uncomplemented) or complemented form (but not both). Thus, a minterm corresponds to a single point (vertex) in $\mathbf{B}^n$. A cube that is not a minterm represents two or more points in $\mathbf{B}^n$. The minterms of a Boolean function correspond to the rows of the truth table at which the function has a value of 1.

Example 2.4

The cube $ab'cd$ is a minterm in $\mathbf{B}^4$. The cube abc is not a minterm. It represents the pair of vertices defined by $abcd + abcd'$.

End of Example 2.4

A Boolean expression in SOP form is said to be canonical if every cube in the expression has a unique representation in which all of the literals are in complemented or uncomplemented form.

Example 2.5

The expression $abcd + a'bcd$ is a canonical SOP.

End of Example 2.5

A *canonic* (standard) SOP function is also called a standard sum-of-products (SSOP). In decimal notation, a minterm is denoted by m_i, and the pattern of 1s and 0s in the binary equivalent of the decimal number indicates the true and complemented literals. For example, $m_7 = a'bcd$.

In $\mathbf{B}^n$ there is a one-to-one correspondence between a minterm and a vertex of an n-dimensional cube, as shown in Figure 2-16. The minterm $m_3 = a'bc$ corresponds to the vertex with coordinates 011.

A *Boolean function* is a set of minterms (vertices) at which the function is asserted. A Boolean function in SOP form is expressed as a sum of minterms.

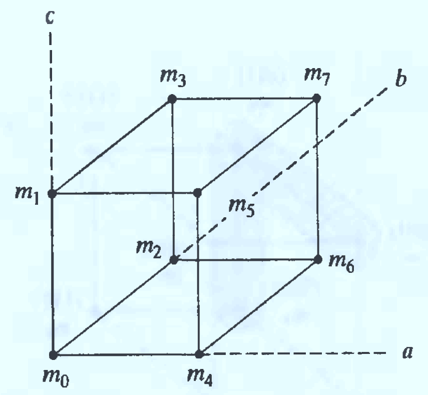

FIGURE 2-16 Correspondence between minterms and vertices in $\mathbf{B}^3$.

Example 2.6

The sum and carry bits of the full adder can be expressed as a sum of minterms, ordered as $\{a, b, c\_in\}$ in $\mathbf{B}^3$:

$$sum = m_1 + m_2 + m_4 + m_7 = \Sigma m(1, 2, 4, 7)$$

$$c\_out = m_3 + m_5 + m_6 + m_7 = \Sigma m(3, 5, 6, 7)$$

End of Example 2.6

Example 2.7

The set of shaded vertices in Figure 2-17 define $f = m_1 + m_2 = m_3 = a'b'c + a'bc' + a'bc$.

End of Example 2.7

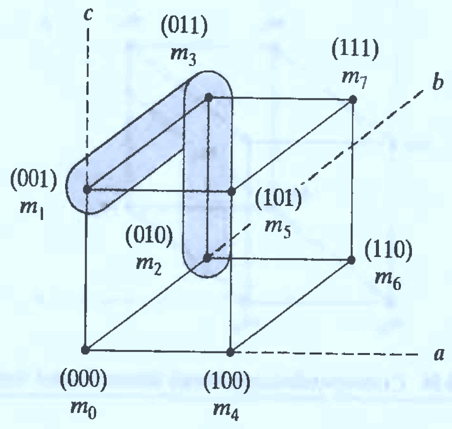

FIGURE 2-17 Set of minterms for $f = a'b'c + a'bc' + a'bc$.

2.3.2 Product-of-Sums Representation

A Boolean function can also be expressed in a POS form in which the expression is written as a product of Boolean factors, each of which is a sum of literals.

Example 2.8

The POS representation of the *c_out* bit in a full adder circuit is formed by expressing the 0s of the truth table in SOP form (see Figure 2-14):

$$c\_out' = a'b'c\_in' + a'b'c\_in + a'bc\_in' + ab'c\_in'$$

Then the expression for c_out' is complemented, giving

$$c\_out = (a'b'c\_in' + a'b'c\_in + a'bc\_in' + ab'c\_in')'$$

DeMorgan's Laws can be applied to *c_out* to give the POS expression shown below:

$$c\_out = (a'b'c\_in')' \cdot (a'b'c\_in)' \cdot (a'bc\_in')' \cdot (ab'c\_in')'$$

$$c\_out = (a + b + c\_in) \cdot (a + b + c\_in') \cdot (a + b' + c\_in) \cdot (a' + b + c\_in)$$

End of Example 2.8

A Boolean expression in POS form is said to be *canonical* (i.e., a unique representation for a given function) if each factor has all of the literals in complemented or uncomplemented form, but not both.

A *maxterm* is an OR-ed sum of literals in which each variable appears exactly once in true or complemented form (e.g., $a + b + c\_in$ is a maxterm in the POS expression for *c_out*). A canonical POS expansion consists of a product of the maxterms of the truth table of a function. The decimal notation of a maxterm is based on the rows of the truth table at which the function is zero (i.e., where f' is asserted). The variables are complemented when forming the POS expression.

Example 2.9

The SOP form of the *c_out'* bit of a full adder was given in the previous example. The decimal notation for *c_out* is given by the product of the maxterms that corresponds to the cubes of *c_out'* as

$$c\_out' = a'b'c\_in' + a'b'c\_in + a'bc\_in' + ab'c\_in'$$

$$c\_out = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi M(0, 1, 2, 4)$$

$$c\_out = (a + b + c\_in) \cdot (a + b + c\_in') \cdot (a + b' + c\_in) \cdot (a' + b + c\_in)$$

A canonical SOP expression can be a very efficient representation of a Boolean function because there might be very few terms at which the function is asserted. Alternatively, f' expressed as a POS expression might be very efficient because there are only a few terms at which the function is de-asserted.

End of Example 2.9

2.4 Simplification of Boolean Expressions

An SOP expression can be implemented in hardware as a two-level AND-OR logic circuit. Although a Boolean expression can always be expressed in a canonical form, with every cube containing every literal (in complemented or uncomplemented form), such descriptions are usually inefficient and waste hardware. In practice, minimization is important because the cost of hardware implementing a Boolean expression is related to the number of terms in the expression and to the number of literals in a term, that is, in a cube in an SOP expression.

A Boolean expression in SOP form is said to be *minimal* if it contains a minimal number of product terms and literals (i.e., a given term cannot be replaced by another that has fewer literals). A minimum SOP form corresponds to a two-level logic circuit having the fewest gates and the fewest number of gate inputs.

There are four common approaches to simplifying a Boolean expression. The first is a manual graphical method that is guided by Karnaugh maps displaying logical adjacencies of the function. Manual methods are feasible only for functions that have