Chapter 4: Combinational Logic Design

In this chapter we cover the techniques to synthesize, analyze, and manipulate logic functions. The purpose of these techniques is to ultimately create a logic circuit using the basic gates described in Chap. 3 from a truth table or word description. This process is called *combinational logic design*. Combinational logic refers to circuits where the output depends on the present value of the inputs. This simple definition implies that there is no storage capability in the circuitry and a change on the input immediately impacts the output. To begin, we first define the rules of Boolean algebra, which provide the framework for the legal operations and manipulations that can be taken on a two-valued number system (i.e., a binary system). We then explore a variety of logic design and manipulation techniques. These techniques allow us to directly create a logic circuit from a truth table and then to manipulate it to either reduce the number of gates necessary in the circuit or to convert the logic circuit into equivalent forms using alternate gates. The goal of this chapter is to provide an understanding of the basic principles of combinational logic design.

Learning Outcomes—After completing this chapter, you will be able to:

- 4.1 Describe the fundamental principles and theorems of Boolean algebra and how to use them to manipulate logic expressions.
- 4.2 Analyze a combinational logic circuit to determine its logic expression, truth table, and timing information.
- 4.3 Synthesize a logic circuit in canonical form (sum of products or product of sums) from a functional description including a truth table, minterm list, or maxterm list.
- 4.4 Synthesize a logic circuit in minimized form (sum of products or product of sums) through algebraic manipulation or with a Karnaugh map.
- 4.5 Describe the causes of timing hazards in digital logic circuits and the approaches to mitigate them.

4.1 Boolean Algebra

The term *algebra* refers to the rules of a number system. In Chap. 2 we discussed the number of symbols and relative values of some of the common number systems. Algebra defines the operations that are legal to perform on that system. Once we have defined the rules for a system, we can then use the system for more powerful mathematics such as solving for unknowns and manipulating into equivalent forms. The ability to manipulate into equivalent forms allows us to minimize the number of logic operations necessary and also put into a form that can be directly synthesized using modern logic circuits.

In 1854, English mathematician George Boole presented an abstract algebraic framework for a system that contained only two states, true and false. This framework essentially launched the field of computer science even before the existence of the modern integrated circuits that are used to implement digital logic today. In 1930, American mathematician Claude Shannon applied Boole's algebraic framework to his work on switching circuits at Bell Labs, thus launching the field of digital circuit design and information theory. Boole's original framework is still used extensively in modern digital circuit design and thus bears the name *Boolean algebra*. Today, the term Boolean algebra is often used to describe not only George Boole's original work, but all of those that contributed to the field after him.

B.J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*,
DOI 10.1007/978-3-319-34195-8_4

4.1.1 Operations

In Boolean algebra there are two valid states (true and false) and three core operations. The operations are conjunction ($\wedge$, equivalent to the AND operation), disjunction ($\vee$, equivalent to the OR operation), and negation ($\neg$, equivalent to the NOT operation). From these three operations, more sophisticated operations can be created including other logic functions (i.e., BUF, NAND, NOR, XOR, XNOR) and arithmetic. Engineers primarily use the terms AND, OR, and NOT instead of conjunction, disjunction, and negation. Similarly, engineers primarily use the symbols for these operators described in Chap. 3 (e.g., $\cdot$, +, and $'$) instead of $\wedge$, $\vee$, and $\neg$.

4.1.2 Axioms

An *axiom* is a statement of truth about a system that is accepted by the user. Axioms are very simple statements about a system, but need to be established before more complicated theorems can be proposed. Axioms are so basic that they do not need to be proved in order to be accepted. Axioms can be thought of as the basic *laws* of the algebraic framework. The terms *axiom* and *postulate* are synonymous and used interchangeably. In Boolean algebra there are five main axioms. These axioms will appear redundant with the description of basic gates from Chap. 3, but must be defined in this algebraic context so that more powerful theorems can be proposed.

4.1.2.1 Axiom #1: Logical Values

This axiom states that in Boolean algebra, a variable A can only take on one of the two values, 0 or 1. If the variable A is not 0, then it must be a 1, and conversely, if it is not a 1, then it must be a 0.

Axiom #1—Boolean Values: $A = 0$ if $A \neq 1$, conversely $A = 1$ if $A \neq 0$.

4.1.2.2 Axiom #2: Definition of Logical Negation

This axiom defines logical negation. Negation is also called the NOT operation or taking the *complement*. The negation operation is denoted using either a prime ($'$), an inversion bar, or the negation symbol ($\neg$). If the complement is taken on a 0, it becomes a 1. If the complement is taken on a 1, it becomes a 0.

Axiom #2—Definition of Logical Negation: if $A = 0$ then $A' = 1$, conversely, if $A = 1$ then $A' = 0$.

4.1.2.3 Axiom #3: Definition of a Logical Product

This axiom defines a logical product or multiplication. Logical multiplication is denoted using either a dot ($\cdot$), an ampersand (&), or the conjunction symbol ($\wedge$). The result of logical multiplication is true when *both* inputs are true and false otherwise.

Axiom #3—Definition of a Logical Product: *$A \cdot B = 1$ if $A = B = 1$ and $A \cdot B = 0$ otherwise.*

4.1.2.4 Axiom #4: Definition of a Logical Sum

This axiom defines a logical sum or addition. Logical addition is denoted using either a plus sign (+) or the disjunction symbol ($\vee$). The result of logical addition is true when *any* of the inputs are true and false otherwise.

Axiom #4—Definition of a Logical Sum: *A + B = 1 if A = 1 or B = 1 and A + B = 0 otherwise.*

4.1.2.5 Axiom #5: Logical Precedence

This axiom defines the order of precedence for the three operators. Unless the precedence is explicitly stated using parentheses, negation takes precedence over a logical product and a logical product takes precedence over a logical sum.

Axiom #5—Definition of Logical Precedence: *NOT precedes AND, AND precedes OR.*

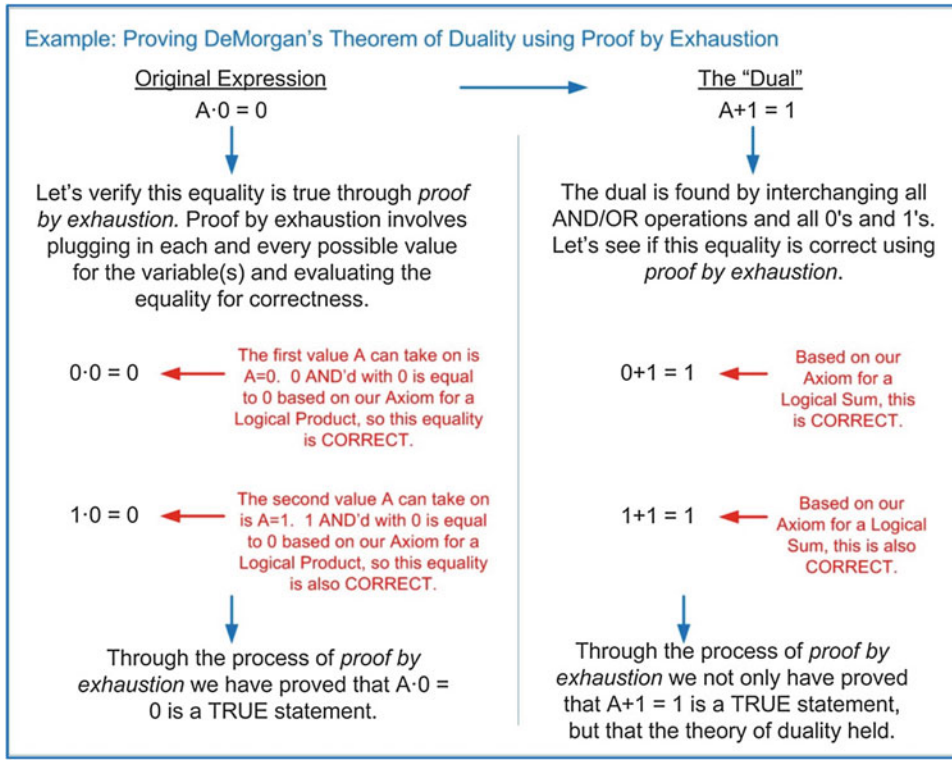
To illustrate Axiom #5, consider the logic function F = A′·B + C. In this function, the first operation that would take place is the NOT operation on A. This would be followed by the AND operation of A′ with B. Finally, the result would be OR'd with C. The precedence of any function can also be explicitly stated using parentheses such as F = (((A′) · B) + C).

4.1.3 Theorems

A theorem is a more sophisticated truth about a system that is not intuitively obvious. Theorems are proposed and then must be proved. Once proved, they can be accepted as a truth about the system going forward. Proving a theorem in Boolean algebra is much simpler than in our traditional decimal system due to the fact that variables can only take on one of the two values, true or false. Since the number of input possibilities is bounded, Boolean algebra theorems can be proved by simply testing the theorem using every possible input code. This is called **proof by exhaustion**. The following theorems are used widely in the manipulation of logic expressions and reduction of terms within an expression.

4.1.3.1 DeMorgan's Theorem of Duality

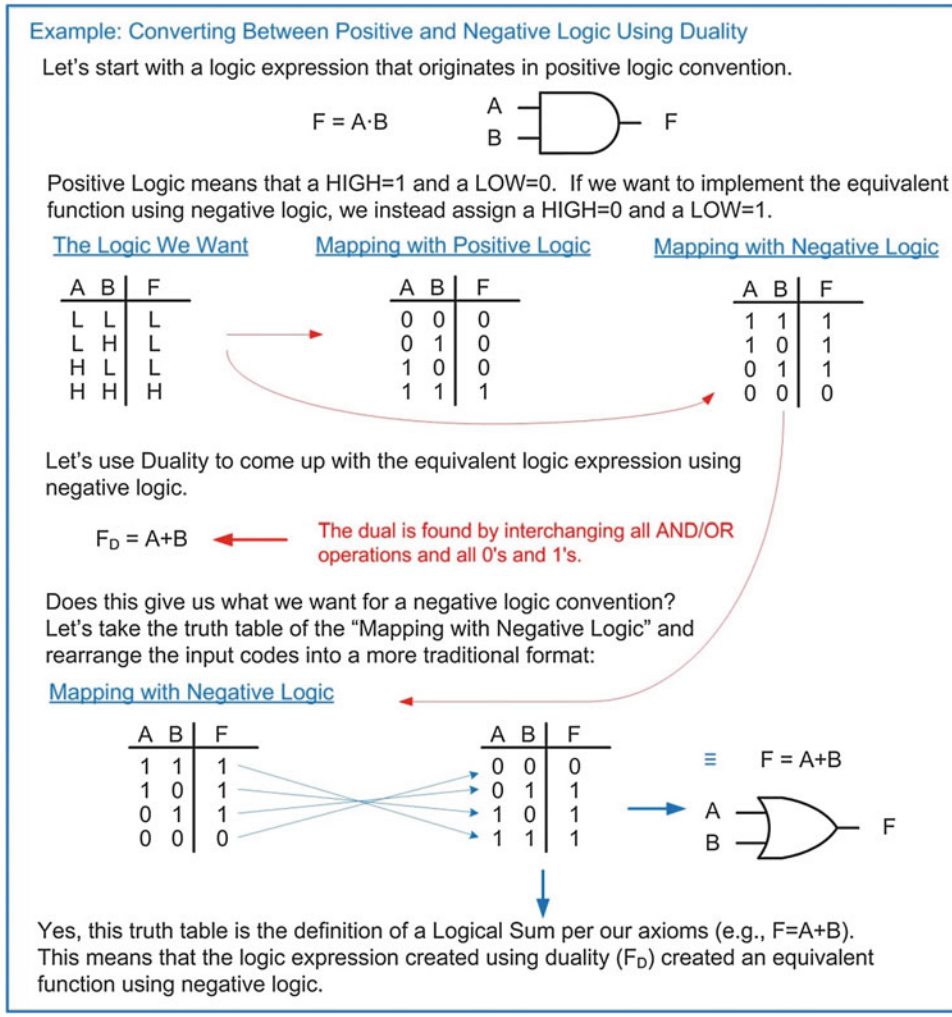
Augustus DeMorgan was a British mathematician and logician who lived during the time of George Boole. DeMorgan is best known for his contribution to the field of logic through the creation of what have been later called the *DeMorgan's Theorems* (often called DeMorgan's laws). There are two major theorems that DeMorgan proposed that expanded Boolean algebra. The first theorem is named *duality*. Duality states that an algebraic equality will remain true if all 0s and 1s are interchanged and all AND and OR operations are interchanged. The new expression is called the *dual* of the original expression. Example 4.1 shows the process of proving duality using proof by exhaustion.



Example 4.1
Proving DeMorgan's Theorem of Duality Using Proof by Exhaustion

Duality is important for two reasons. First, it doubles the impact of a theorem. If a theorem is proved to be true, then the dual of that theorem is also proved to be true. This, in essence, gives twice the theorem with the same amount of proving. Boolean algebra theorems are almost always given in pairs, the original and the dual. That is why duality is covered as the first theorem.

The second reason that duality is important is because it can be used to convert between positive and negative logic. Until now, we have used positive logic for all of our examples (i.e., a logic HIGH = true = 1 and a logic LOW = false = 0). As mentioned earlier, this convention is arbitrary and we could have easily chosen a HIGH to be false and a LOW to be true (i.e., negative logic). Duality allows us to take a logic expression that has been created using positive logic (F) and then convert it into an equivalent expression that is valid for negative logic (F_D). Example 4.2 shows the process for how this works.



Example 4.2
Converting Between Positive and Negative Logic Using Duality

One consideration when using duality is that the order of precedence follows the original function. This means that in the original function, the axiom for precedence states the order as NOT-AND-OR; however, this is not necessarily the correct precedence order in the dual. For example, if the original function was $F = A \cdot B + C$, the AND operation of A and B would take place first, and then the result would be OR'd with C. The dual of this expression is $F_D = A + B \cdot C$. If the expression for F_D was evaluated using traditional Boolean precedence, it would show that F_D does NOT give the correct result per the definition of a dual function (i.e., converting a function from positive to negative logic). The order of precedence for F_D must correlate to the precedence in the original function. Since in the original function A and B were operated on first, they must also be operated on first in the dual. In order to easily manage this issue, parentheses can be used to track the order of operations from the original function to the dual. If we put parentheses in the original function to explicitly state the precedence of the operations, it would take the form $F = (A \cdot B) + C$. These parentheses can be mapped directly to the dual yielding $F_D = (A + B) \cdot C$. This order of precedence in the dual is now correct.

Now that we have covered the duality operation, its usefulness, and its pitfalls, we can formally define this theorem as:

DeMorgan's Duality: An algebraic equality will remain true if all 0s and 1s are interchanged and all AND and OR operations are interchanged. Furthermore, taking the dual of a positive logic function will produce the equivalent function using negative logic if the original order of precedence is maintained.

4.1.3.2 Identity

An identity operation is one that when performed on a variable will yield itself regardless of the variable's value. The following is the formal definition of identity theorem. Figure 4.1 shows the gate-level depiction of this theorem.

Identity: OR'ing any variable with a logic 0 will yield the original variable. The dual: AND'ing any variable with a logic 1 will yield the original variable.

Original	Dual
A+0 = A	A•1 = A

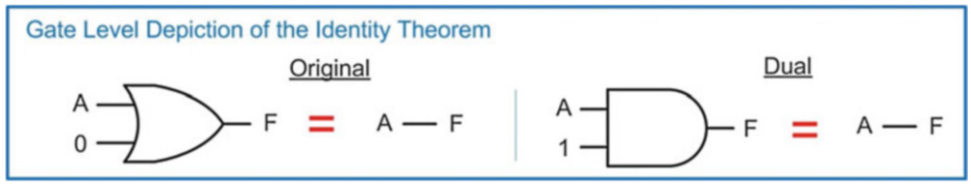


Fig. 4.1
Gate-level depiction of the identity theorem

The identity theorem is useful for reducing circuitry when it is discovered that a particular input will never change values. When this is the case, the static input variable can simply be removed from the logic expression making the entire circuit a simple wire from the remaining input variable to the output.

4.1.3.3 Null Element

A null element operation is one that, when performed on a constant value, will yield that same constant value regardless of the values of any variables within the same operation. The following is the formal definition of null element. Figure 4.2 shows the gate-level depiction of this theorem.

Null Element: OR'ing any variable with a logic 1 will yield a logic 1 regardless of the value of the input variable. The dual: AND'ing any variable with a logic 0 will yield a logic 0 regardless of the value of the input variable.

Original	Dual
A+1 = 1	A•0 = 0

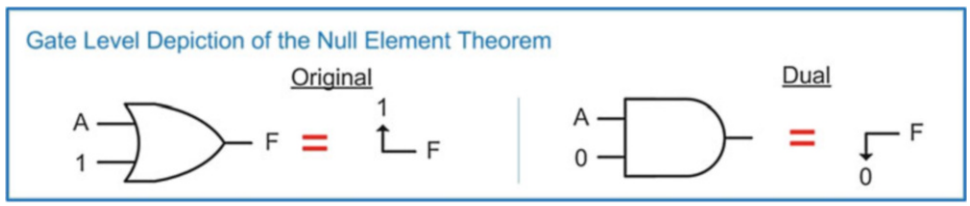


Fig. 4.2
Gate-level depiction of the null element theorem

The null element theorem is also useful for reducing circuitry when it is discovered that a particular input will never change values. It is also widely used in computer systems in order to set (i.e., force to a logic 1) or clear (i.e., force to a logic 0) the value of a storage element.

4.1.3.4 Idempotent

An idempotent operation is one that has no effect on the input, regardless of the number of times the operation is applied. The following is the formal definition of idempotence. Figure 4.3 shows the gate-level depiction of this theorem.

Idempotent: OR'ing a variable with itself results in itself. The dual: AND'ing a variable with itself results in itself.

Original	Dual
A+A = A	A•A = A

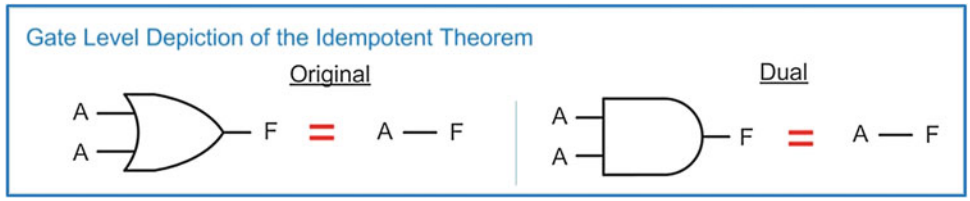


Fig. 4.3
Gate-level depiction of the idempotent theorem

This theorem also holds true for any number of operations such as $A + A + A + \ldots + A = A$ and $A \cdot A \cdot A \cdot \ldots \cdot A = A$.

4.1.3.5 Complements

This theorem describes an operation of a variable with the variable's own complement. The following is the formal definition of complements. Figure 4.4 shows the gate-level depiction of this theorem.

Complements: OR'ing a variable with its complement will produce a logic 1. The dual: AND'ing a variable with its complement will produce a logic 0.

Original	Dual
A+A' = 1	A•A' = 0

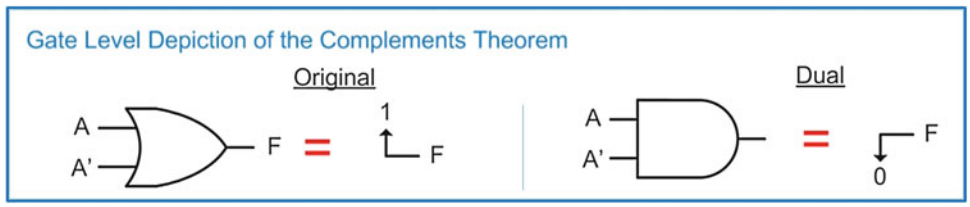


Fig. 4.4
Gate-level depiction of the complements theorem

The complement theorem is again useful for reducing circuitry when these types of logic expressions are discovered.

4.1.3.6 Involution

An involution operation describes the result of double negation. The following is the formal definition of involution. Figure 4.5 shows the gate-level depiction of this theorem.

Involution: Taking the double complement of a variable will result in the original variable.

Original
A'' = A

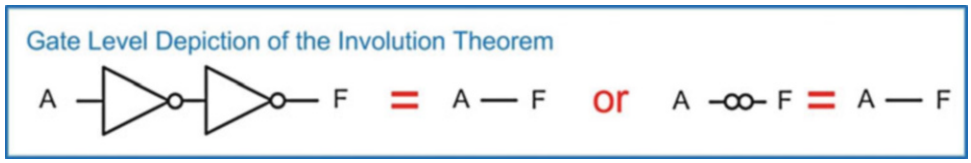


Fig. 4.5
Gate-level depiction of the involution theorem

This theorem is not only used to eliminate inverters but also provides us a powerful tool for *inserting* inverters in a circuit. We will see that this is used widely with the second of DeMorgan's laws that will be introduced at the end of this section.

4.1.3.7 Commutative Property

The term commutative is used to describe an operation in which the order of the quantities or variables in the operation has no impact on the result. The following is the formal definition of the commutative property. Figure 4.6 shows the gate-level depiction of this theorem.

Commutative Property: Changing the order of variables in an OR operation does not change the end result. The dual: Changing the order of variables in an AND operation does not change the end result.

Original	Dual
A+B = B+A	A•B = B•A

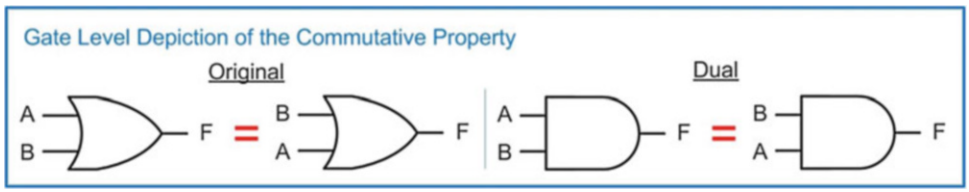
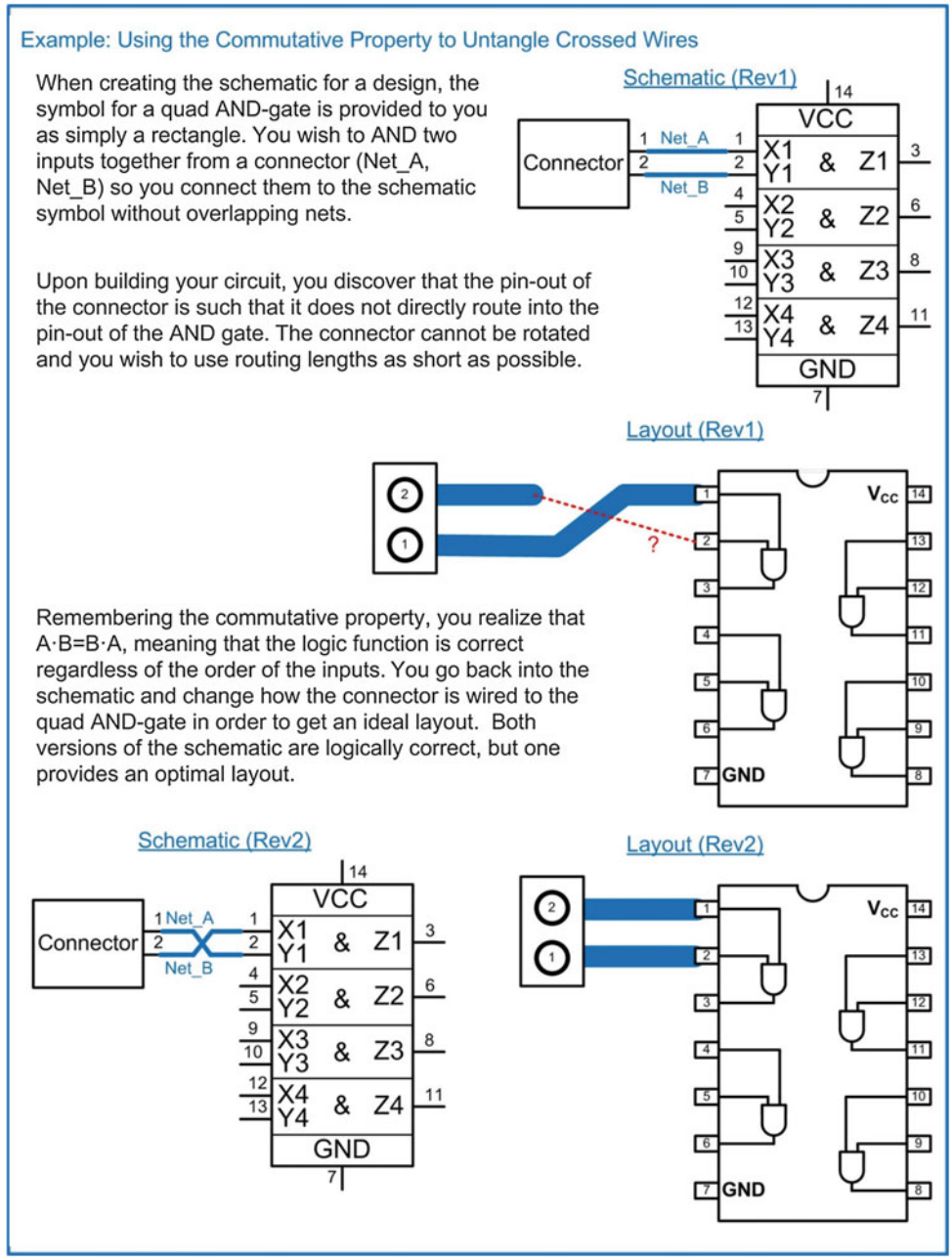


Fig. 4.6
Gate-level depiction of commutative property

One practical use of the commutative property is when wiring or routing logic circuitry together. Example 4.3 shows how the commutative property can be used to untangle crossed wires when implementing a digital system.



Example 4.3
Using the Commutative Property to Untangle Crossed Wires

4.1.3.8 Associative Property

The term associative is used to describe an operation in which the grouping of the quantities or variables in the operation has no impact on the result. The following is the formal definition of the associative property. Figure 4.7 shows the gate-level depiction of this theorem.

Associative Property: The grouping of variables doesn't impact the result of an OR operation. The dual: The grouping of variables doesn't impact the result of an AND operation.

Original	Dual
(A+B)+C = A+(B+C)	(A·B)·C = A·(B·C)

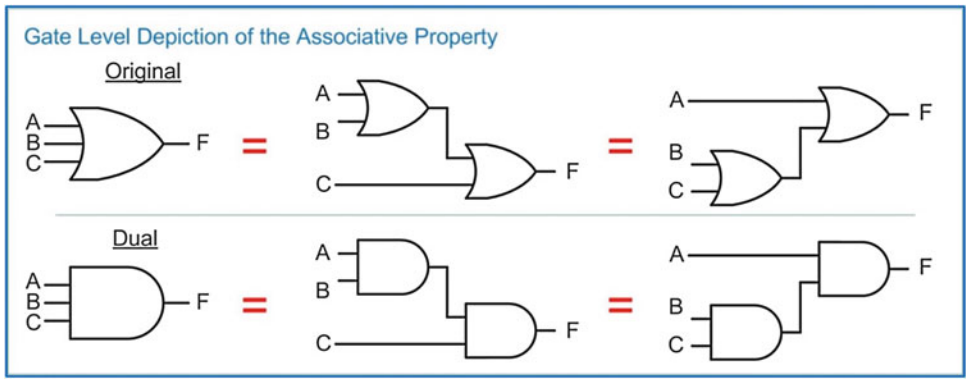
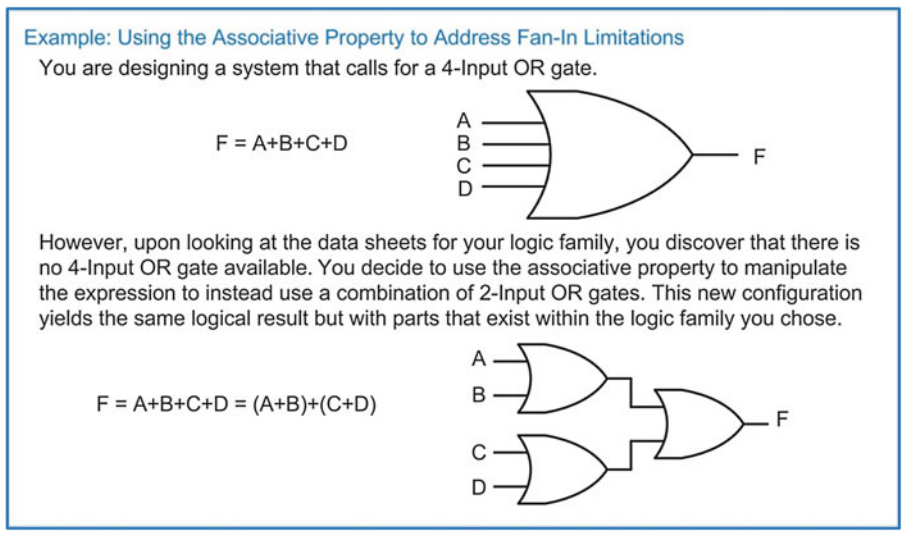


Fig. 4.7
Gate-level depiction of the associative property

One practical use of the associative property is addressing fan-in limitations of a logic family. Since the grouping of the input variables does not impact the result, we can accomplish operations with large numbers of inputs using multiple gates with fewer inputs. Example 4.4 shows the process of using the associative property to address a fan-in limitation.



Example 4.4
Using the Associative Property to Address Fan-In Limitations

4.1.3.9 Distributive Property

The term distributive describes how an operation on a parenthesized group of operations (or higher precedence operations) can be distributed through each term. The following is the formal definition of the distributive property. Figure 4.8 shows the gate-level depiction of this theorem.

Distributive Property: An operation on a parenthesized operation(s), or higher precedence operator, will distribute through each term.

Original	Dual
$A \cdot (B+C) = A \cdot B + A \cdot C$	$A+(B \cdot C) = (A+B) \cdot (A+C)$

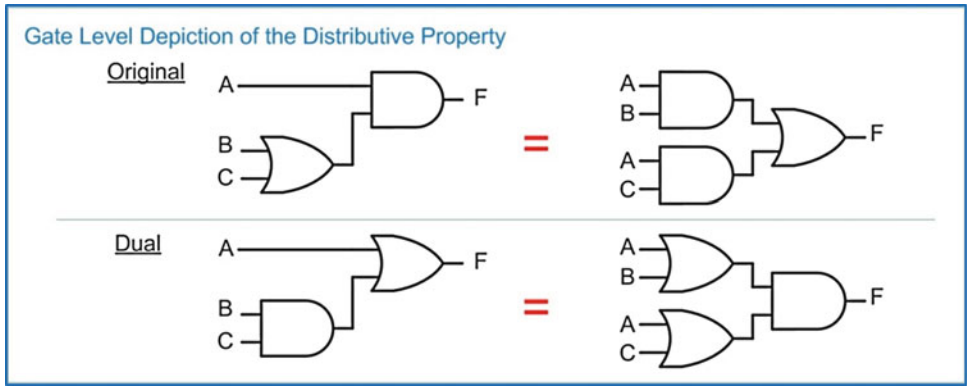
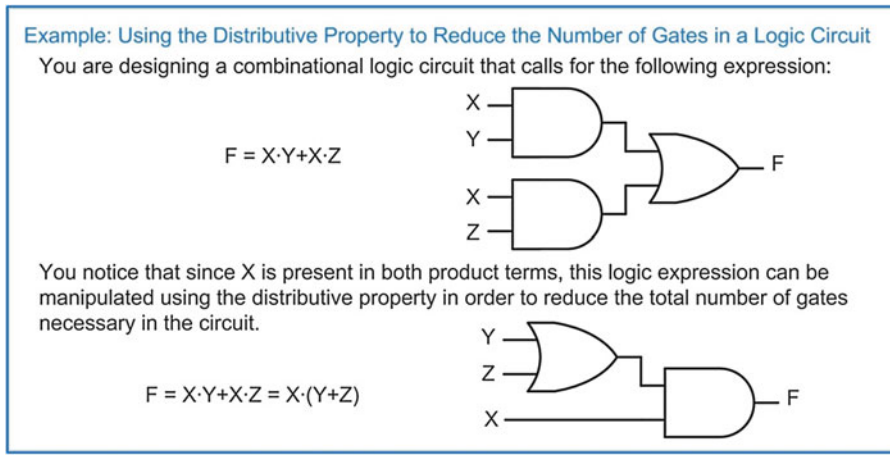


Fig. 4.8
Gate-level depiction of the distributive property

The distributive property is used as a logic manipulation technique. It can be used to put a logic expression into a form more suitable for direct circuit synthesis, or to reduce the number of logic gates necessary. Example 4.5 shows how to use the distributive property to reduce the number of gates in a logic circuit.



Example 4.5
Using the Distributive Property to Reduce the Number of Logic Gates in a Circuit

4.1.3.10 Absorption

The term absorption refers to when multiple logic terms within an expression produce the same results. This allows one of the terms to be eliminated from the expression, thus reducing the number of logic operations. The remaining terms essentially *absorb* the functionality of the eliminated term. This theorem is also called *covering* because the remaining term essentially covers the functionality of both itself and the eliminated term. The following is the formal definition of the absorption theorem. Figure 4.9 shows the gate-level depiction of this theorem.

Absorption: When a term within a logic expression produces the same output(s) as another term, the second term can be removed without affecting the result.

Original	Dual
A+A·B = A	A·(A+B) = A

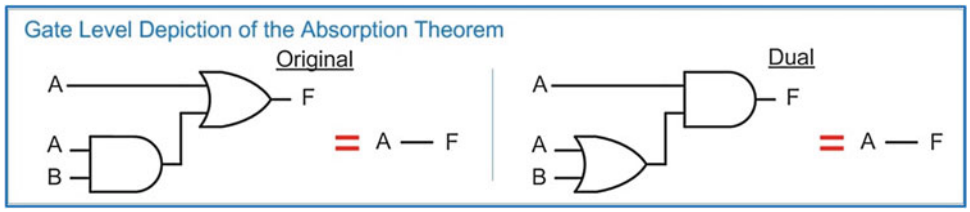


Fig. 4.9
Gate-level depiction of absorption

This theorem is better understood by looking at the evaluation of each term with respect to the original expression. Example 4.6 shows how the absorption theorem can be proven through proof by exhaustion by evaluating each term in a logic expression.

Example: Proving the Absorption Theorem using Proof by Exhaustion

Consider the expression F= A + A·B. Let's evaluate each of the two terms in the OR'd expression and then see how they relate to the output of the original expression.

A	B	A+A·B	A	A·B
0	0	0	0	0
0	1	0	0	0
1	0	1	1	0
1	1	1	1	1

The evaluation of the original expression
The evaluation of the term A
The evaluation of the term A·B

Notice that the term A will produce a result of 1 for the input code A=1, B=1. This result is sufficient to cover the result produced by the term A·B for this input code. When these two terms are OR'd together, the A·B term becomes unnecessary because its output will be fully covered by the term A. We can thus reduce the expression to simply A. We can say that the term A·B can be *absorbed* into A.

Example 4.6
Proving the Absorption Theorem Using Proof by Exhaustion

4.1.3.11 Uniting

The uniting theorem, also called *combining* or *minimization*, provides a way to remove variables from an expression when they have no impact on the outcome. This theorem is one of the most widely used

techniques for the reduction of the number of gates needed in a combinational logic circuit. The following is the formal definition of the uniting theorem. Figure 4.10 shows the gate-level depiction of this theorem.

Uniting: When a variable (B) and its complement (B′) appear in multiple product terms with a common variable (A) within a logical OR operation, the variable B does not have any effect on the result and can be removed.

Original	Dual
A·B +A·B' = A	(A+B)·(A+B') = A

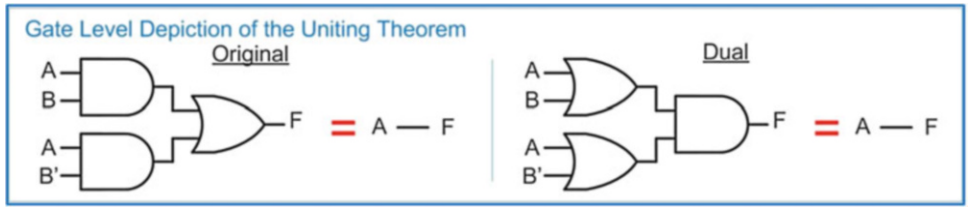


Fig. 4.10
Gate-level depiction of uniting

This theorem can be proved using prior theorems. Example 4.7 shows how the uniting theorem can be proved using a combination of the distributive property, the complements theorem, and the identity theorem.

Example: Proving the Uniting Theorem

Uniting theorem states that A·B + A·B' = A. Let's use the other Boolean algebra theorems to manipulate the original expression in order to prove this theorem.

The original expression:	F = A·B + A·B'
Using the distributive property, we can rewrite the expression as:	F = A·(B + B')
The "complements theorem" states that B+B'=1, so we can now rewrite the expression as:	F = A·1
The *identity theorem* states that A·1=A, so the expression can be written in its final form.	F = A

This proves that the uniting theorem holds true. Uniting theorem is also called *minimization* or *combining*.

Example 4.7
Proving of the Uniting Theorem

4.1.3.12 DeMorgan's Theorem

Now we look at the second of DeMorgan's laws. This second theorem is simply known as *DeMorgan's theorem*. This theorem provides a technique to manipulate a logic expression that uses AND gates into one that uses OR gates and vice versa. It can also be used to manipulate traditional Boolean logic expressions that use AND-OR-NOT operators, into equivalent forms that use NAND and NOR gates. The following is the formal definition of DeMorgan's theorem. Figure 4.11 shows the gate-level depiction of this theorem.

DeMorgan's Theorem: An OR operation with both inputs inverted is equivalent to an AND operation with the output inverted. The dual: An AND operation with both inputs inverted is equivalent to an OR operation with the output inverted.

Original	Dual
A' + B' = (A·B)'	A' · B' = (A + B)'

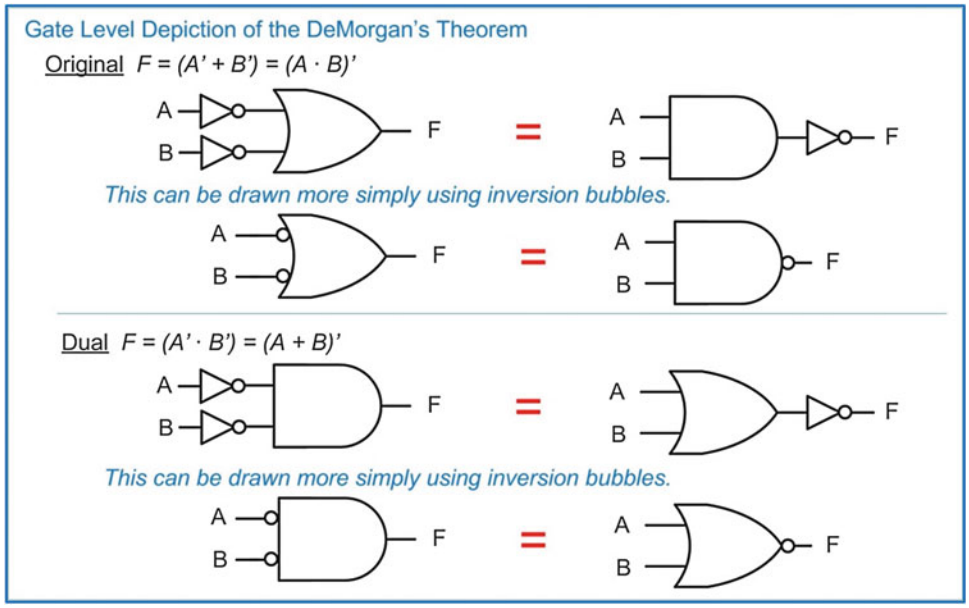


Fig. 4.11
Gate-level depiction of DeMorgan's theorem

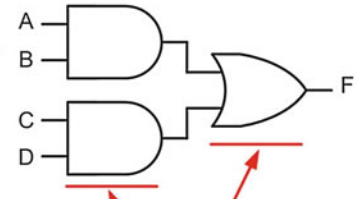
This theorem is used widely in modern logic design because it bridges the gap between the design of logic circuitry using Boolean algebra and the physical implementation of the circuitry using CMOS. Recall that Boolean algebra is defined for only three operations, the AND, the OR, and inversion. CMOS, on the other hand, can only directly implement negative-type gates such as NAND, NOR, and NOT. DeMorgan's theorem allows us to design logic circuitry using Boolean algebra and synthesize logic diagrams with AND, OR, and NOT gates, and then directly convert the logic diagrams into an equivalent form using NAND, NOR, and NOT gates. As we'll see in the next section, Boolean algebra produces logic expressions in two common forms. These are the **sum of products (SOP)** and the **product of sums (POS)** forms. Using a combination of involution and DeMorgan's theorem, SOP and POS forms can be converted into equivalent logic circuits that use only NAND and NOR gates. Example 4.8 shows a process to convert a sum of products form into one that uses only NAND gates.

Example: Converting a Sum of Products Form into One That Uses Only NAND Gates

You are designing a combinational logic circuit that will be implemented in CMOS. You use Boolean algebra to create a circuit in the form of a **sum of products (SOP)**.

$F = A \cdot B + C \cdot D$

These two logical products (e.g., AND operations) are summed together (e.g., OR operation) to form a Sum of Product form.

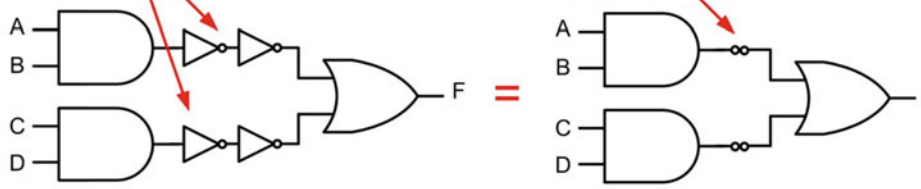


A Sum of Products at the gate level always has a stage of AND gates feeding into a single OR gate.

Since this logic needs to be implemented in CMOS, you need to convert it into a form that uses only NAND, NOR or NOT gates. You know that DeMorgan's Theorem allows an OR gate with its inputs inverted to be converted to an AND gate with its output inverted (e.g., a NAND gate). To prepare for this manipulation, you take advantage of the theory of involution, which allows you to put double inversions on any net without affecting the result.

Double inverters are placed on these nodes in order to create an OR gate with its inputs inverted.

These inverters can also be denoted using inversion bubbles (e.g., double bubbles).



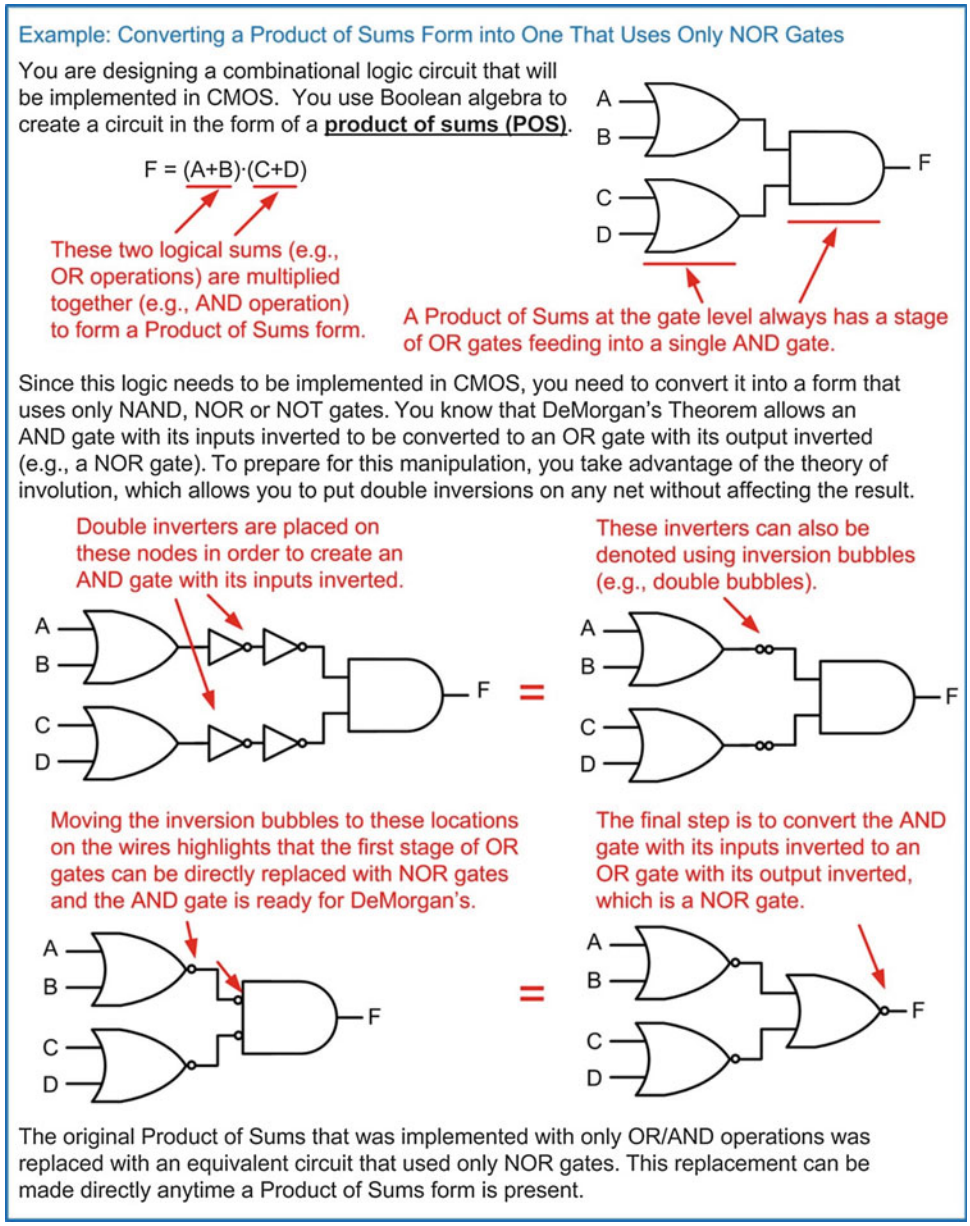
Moving the inversion bubbles to these locations on the wires highlights that the first stage of AND gates can be directly replaced with NAND gates and the OR gate is ready for DeMorgan's.

The final step is to convert the OR gate with its inputs inverted to an AND gate with its output inverted, which is a NAND gate.

The original Sum of Products that was implemented with only AND/OR operations was replaced with an equivalent circuit that used only NAND gates. This replacement can be made directly anytime a Sum of Products form is present.

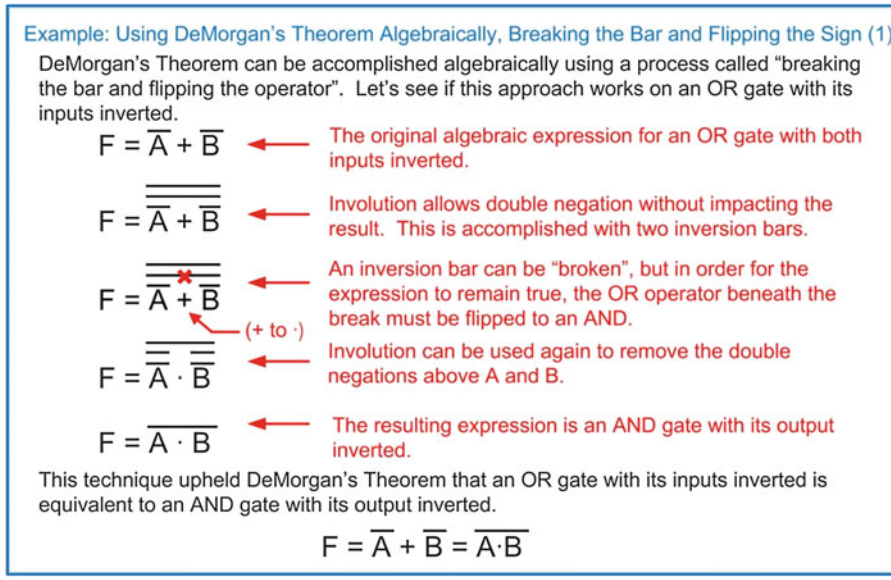
Example 4.8
Converting a Sum of Products Form into One That Uses Only NAND Gates

Example 4.9 shows a process to convert a product of sums form into one that uses only NOR gates.



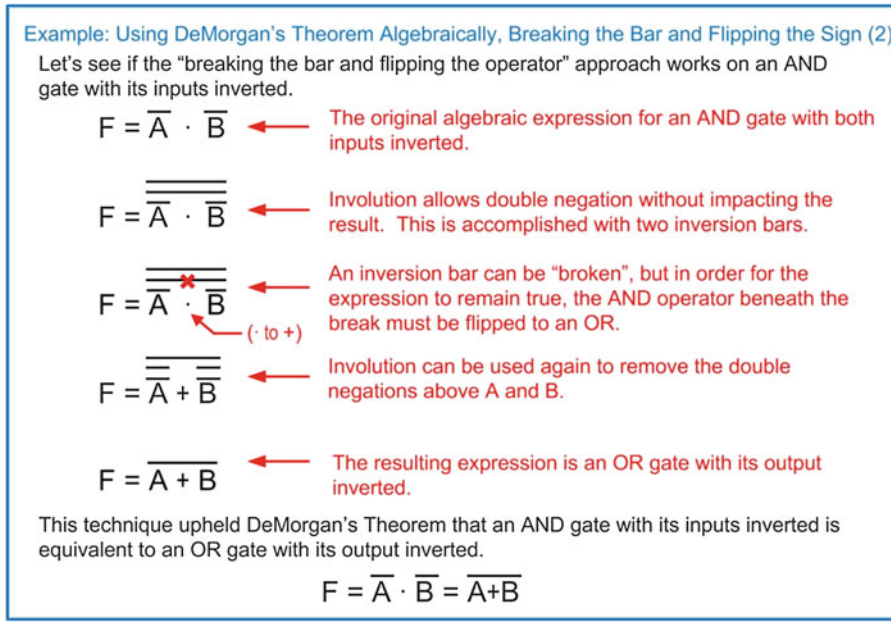
Example 4.9
Converting a Product of Sums Form into One That Uses Only NOR Gates

DeMorgan's theorem can also be accomplished algebraically using a process known as *breaking the bar and flipping the operator*. This process again takes advantage of the involution theorem, which allows double negation without impacting the result. When using this technique in algebraic form, involution takes the form of a double-inversion bar. If an inversion bar is *broken*, the expression will remain true as long as the operator directly below the break is flipped (AND to OR, OR to AND). Example 4.10 shows how to use this technique when converting an OR gate with its inputs inverted into an AND gate with its output inverted.



Example 4.10
Using DeMorgan's Theorem in Algebraic Form (1)

Example 4.11 shows how to use this technique when converting an AND gate with its inputs inverted into an OR gate with its output inverted.



Example 4.11
Using DeMorgan's Theorem in Algebraic Form (2)

Table 4.1 gives a summary of all the Boolean algebra theorems just covered. The theorems are grouped in this table with respect to the number of variables that they contain. This grouping is the most common way these theorems are presented.

Summary of Boolean Algebra Theorems

Single Variable Theorems	Original	Dual
Identity	$A+0 = A$	$A \cdot 1 = A$
Null Element	$A+1 = 1$	$A \cdot 0 = 0$
Idempotency	$A+A = A$	$A \cdot A = A$
Complements	$A+A' = 1$	$A \cdot A' = 0$
Involution	$A'' = A$	
Multiple Variable Theorems		
Commutative	$A+B = B+A$	$A \cdot B = B \cdot A$
Associative	$(A+B)+C = A+(B+C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
Distributive	$A \cdot (B+C) = A \cdot B + A \cdot C$	$A+(B \cdot C) = (A+B) \cdot (A+C)$
Absorption (or Covering)	$A+A \cdot B = A$	$A \cdot (A+B) = A$
Uniting (or Combining)	$A \cdot B + A \cdot B' = A$	$(A+B) \cdot (A+B') = A$
DeMorgan's	$A' \cdot B' = (A + B)'$	$A'+B' = (A \cdot B)'$

Table 4.1
Summary of Boolean algebra theorems

4.1.4 Functionally Complete Operation Sets

A set of Boolean operators is said to be *functionally complete* when the set can implement all possible logic functions. The set of operators {AND, OR, NOT} is functionally complete because every other operation can be implemented using these three operators (i.e., NAND, NOR, BUF, XOR, XNOR). The DeMorgan's theorem showed us that all AND and OR operations can be replaced with NAND and NOR operators. This means that NAND and NOR operations could be by themselves functionally complete if they could perform a NOT operation. Figure 4.12 shows how a NAND gate can be configured to perform a NOT operation. This configuration allows a NAND gate to be considered functionally complete because all other operations can be implemented.

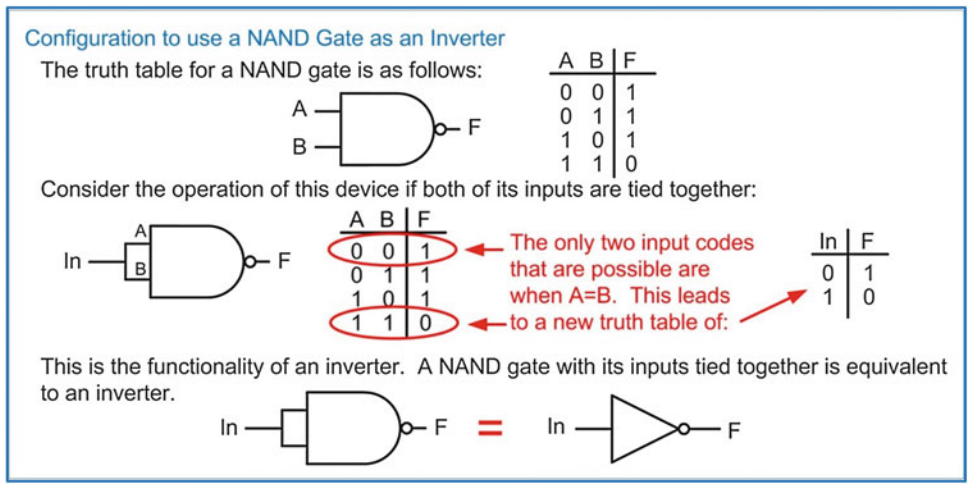


Fig. 4.12
Configuration to use a NAND gate as an inverter

This approach can also be used on a NOR gate to implement an inverter. Figure 4.13 shows how a NOR gate can be configured to perform a NOT operation, thus also making it functionally complete.

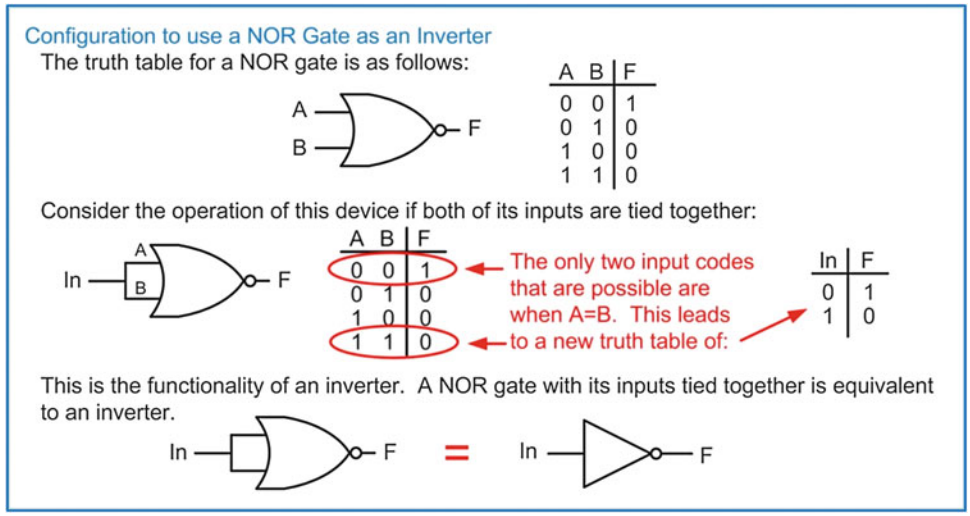


Fig. 4.13
Configuration to use a NOR gate as an inverter

CONCEPT CHECK

CC4.1 If the logic expression F=A·B·C·D·E·F·G·H is implemented with only 2-input AND gates, how many levels of logic will the final implementation have? Hint: Consider using the associative property to manipulate the logic expression to use only 2-input AND operations.

A) 2 B) 3 C) 4 D) 5

4.2 Combinational Logic Analysis

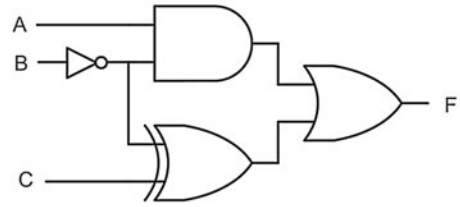
Combinational logic analysis refers to the act of deciphering the operation of a circuit from its final logic diagram. This is a useful skill that can aid designers when debugging their circuits. This can also be used to understand the timing performance of a circuit and to reverse-engineer an unknown design.

4.2.1 Finding the Logic Expression from a Logic Diagram

Combinational logic diagrams are typically written with their inputs on the left and their output on the right. As the inputs change, the intermediate *nodes*, or connections, within the diagram hold the interim computations that contribute to the ultimate circuit output. These computations propagate from left to right until ultimately the final output of the system reaches its final steady-state value. When analyzing the behavior of a combinational logic circuit a similar *left-to-right* approach is used. The first step is to label each intermediate node in the system. The second step is to write in the logic expression for each node based on the preceding logic operation(s). The logic expressions are written working left-to-right until the output of the system is reached and the final logic expression of the circuit has been found. Consider the example of this analysis in Example 4.12.

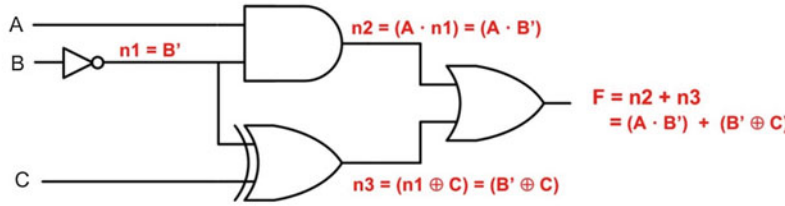
Example: Determining the Logic Expression from a Logic Diagram

Given: The following combinational logic diagram.



Find: The logic expression for the output F.

Solution: First, let's label each of the internal nodes of the circuit. We'll call these nodes n1, n2, and n3. Next, let's insert the logic expression for each node working from the left to the right. Finally, we can write the final output logic expression for F based on all of the prior internal node expressions. Substitutions can be made within each expression to put the logic in terms of only the input variable names (i.e., A, B, and C).



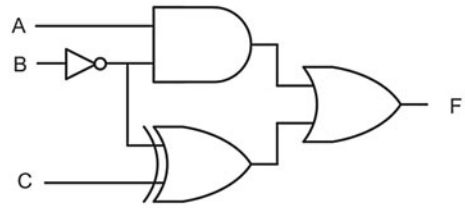
Example 4.12
Determining the Logic Expression from a Logic Diagram

4.2.2 Finding the Truth Table from a Logic Diagram

The final truth table of a circuit can also be found in a similar manner as the logic expression. Each internal node within the logic diagram can be evaluated working from the left to the right for each possible input code. Each subsequent node can then be evaluated using the values of the preceding nodes. Consider the example of this analysis in Example 4.13.

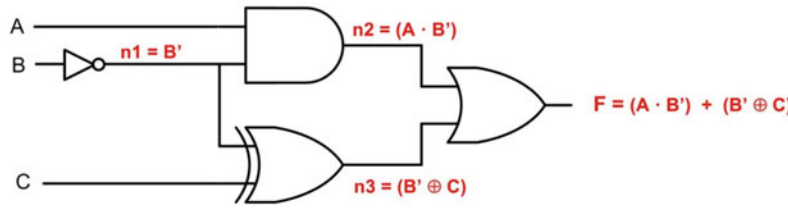
Example: Determining the Truth Table from a Logic Diagram

Given: The following combinational logic diagram.



Find: The truth table for the output F.

Solution: First, we label each internal node and record the intermediate logic expressions.



Next, we evaluate each node for all possible input codes working from the left to the right. This allows us to keep a record of the values of each intermediate node that can be used in the subsequent evaluations. We continue this process until we reach the final output F.

A	B	C	n1 = B'	n2 = A·B'	n3 =B'⊕C	F = (A·B') + (B'⊕C)
0	0	0	1	0	1	1
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	1	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

Notice that the intermediate computations can be used in the subsequent evaluations.

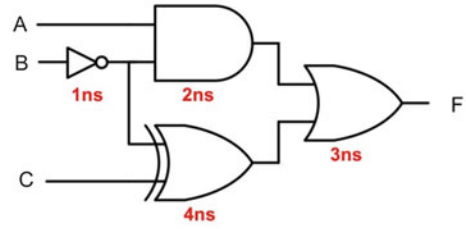
Example 4.13
Determining the Truth Table from a Logic Diagram

4.2.3 Timing Analysis of a Combinational Logic Circuit

Real logic gates have a propagation delay (t_{pd}, t_{PHL}, or t_{PLH}) as presented in Chap. 3. Performing a timing analysis on a combinational logic circuit refers to observing how long it takes for a change in the inputs to propagate to the output. Different paths through the combinational logic circuit will take different times to compute since they may use gates with different delays. When determining the delay of the entire combinational logic circuit we always consider the longest delay path. This is because this delay represents the worst-case scenario. As long as we wait for the longest path to propagate through the circuit, then we are ensured that the output will always be valid after this time. To determine which signal path has the longest delay, we map out each and every path the inputs can take to the output of the circuit. We then sum up the gate delay along each path. The path with the longest delay dictates the delay of the entire combinational logic circuit. Consider this analysis shown in Example 4.14.

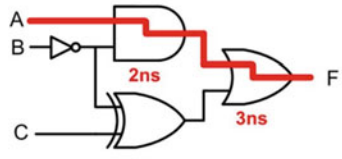
Example: Determining the Delay of a Combinational Logic Circuit

Given: The following combinational logic diagram with the associated gate delays.

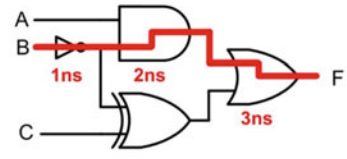


Find: The delay of the combinational logic circuit.

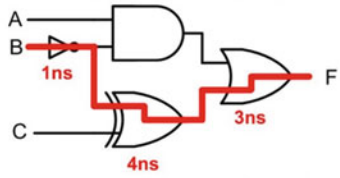
Solution: We begin by mapping the route of each and every path from the inputs to the output. For each path, we sum the delay through each gate that is used.



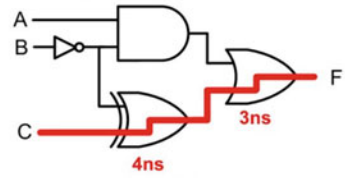
$t_{delay\text{-}1}$ = 2ns + 3ns = 5ns



$t_{delay\text{-}2}$ = 1ns + 2ns + 3ns = 6ns



$t_{delay\text{-}3}$ = 1ns + 4ns + 3ns = 8ns



$t_{delay\text{-}4}$ = 4ns + 3ns = 7ns

The longest delay path through this circuit is from B to F in which the signal traverses the inverter, XOR gate, and OR gate ($t_{delay\text{-}3}$). This path takes 8ns to compute. Since we must always consider the longest delay path when calculating how fast this circuit can operate, we can say that the delay of this combinational logic circuit is 8ns.

Example 4.14
Determining the Delay of a Combinational Logic Circuit

CONCEPT CHECK

CC4.2 Does the delay specification of a combinational logic circuit change based on the input values that the circuit is evaluating?

A) Yes. There are times when the inputs switch between inputs codes that use paths through the circuit with different delays.

B) No. The delay is always specified as the longest delay path.

C) Yes. The delay can vary between the longest delay path and zero. A delay of zero occurs when the inputs switch between two inputs codes that produce the same output.

D) No. The output is always produced at a time equal to the longest delay path.

4.3 Combinational Logic Synthesis

4.3.1 Canonical Sum of Products

One technique to directly synthesize a logic circuit from a truth table is to use a canonical sum of products topology based on **minterms**. The term *canonical* refers to this topology yielding potentially unminimized logic. A minterm is a product term (i.e., an AND operation) that will be true for one and only one input code. The minterm must contain every input variable in its expression. Complements are applied to the input variables as necessary in order to produce a true output for the individual input code. We define the word *literal* to describe an input variable which may or may not be complemented. This is a more useful word because if we say that a minterm "must include all variables," it implies that all variables are included in the term uncomplemented. A more useful statement is that a minterm "must include all literals." This now implies that each variable must be included, but it can be in the form of itself or its complement (e.g., A or A′). Figure 4.14 shows the definition and gate-level depiction of a minterm expression. Each minterm can be denoted using the lower case "m" with the row number as a subscript.

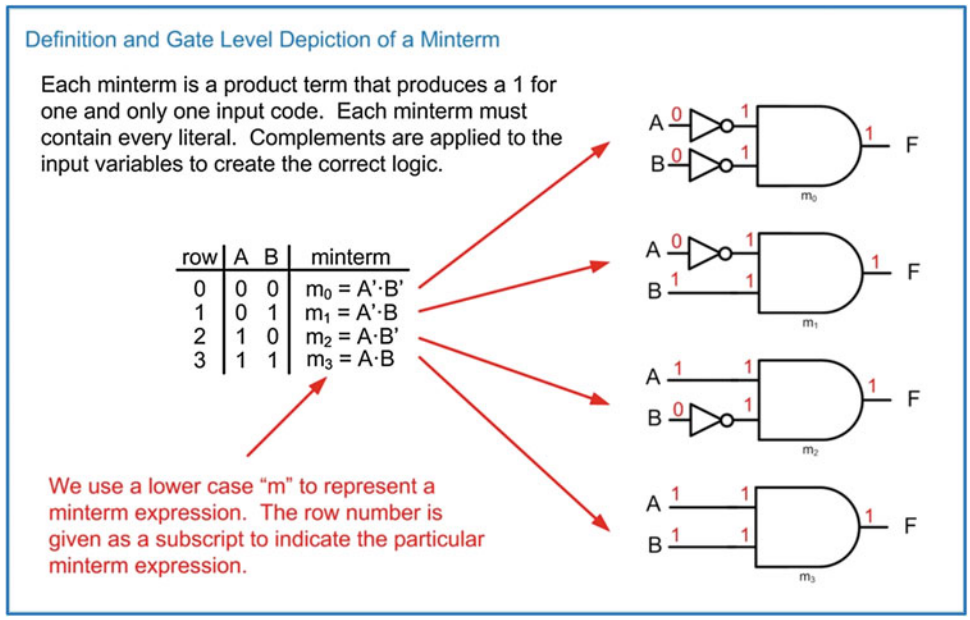


Fig. 4.14
Definition and gate-level depiction of a minterm

For an arbitrary truth table, a minterm can be used for each row corresponding to a true output. If each of these minterms' outputs are fed into a single OR gate, then a sum of products logic circuit is formed that will produce the logic listed in the truth table. In this topology, any input code that corresponds to an output of 1 will cause its corresponding minterm to output a 1. Since a 1 on any input of an OR gate will cause the output to go to a 1, the output of the minterm is passed to the final result. Example 4.15 shows this process. One important consideration of this approach is that no effort has been taken to minimize the logic expression. This unminimized logic expression is also called the **canonical sum**. The canonical sum is logically correct but uses the most amount of circuitry possible for a given truth table. This canonical sum can be the starting point for minimization using Boolean algebra.

Example: Creating a Canonical Sum of Products Logic Circuit using Minterms

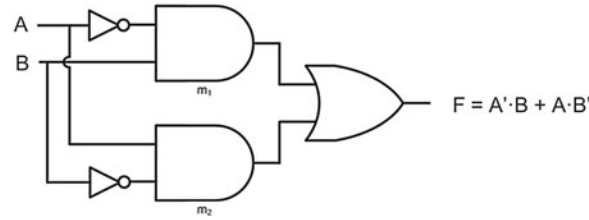
Given: The following truth table.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

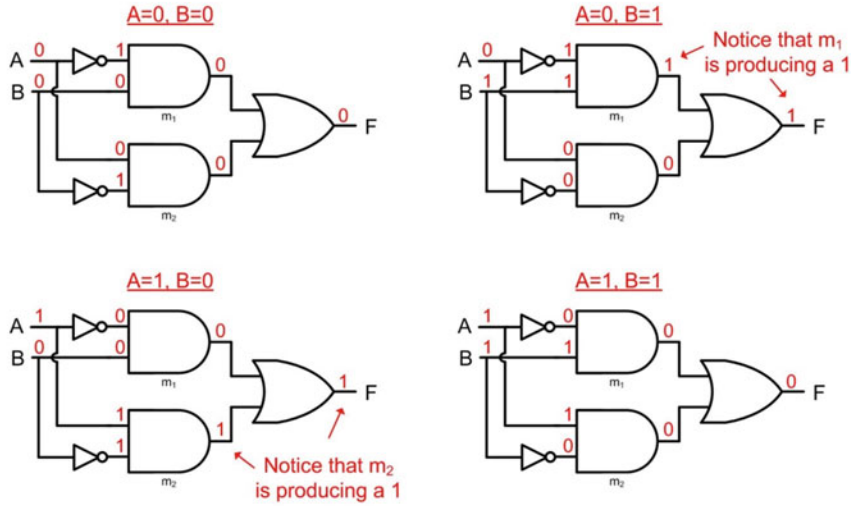
Find: The Canonical SOP

Solution: Let's first start by writing the minterms for the rows that correspond to a 1 on the output. These can then be implemented using inverters and AND gates. The final step is to feed the outputs of each minterm circuit into a single OR gate.

row	A	B	minterm
0	0	0	-
1	0	1	$m_1 = A' \cdot B$
2	1	0	$m_2 = A \cdot B'$
3	1	1	-



Let's now check that this circuit performs as intended by testing it under each input code for A and B and observing the output F.

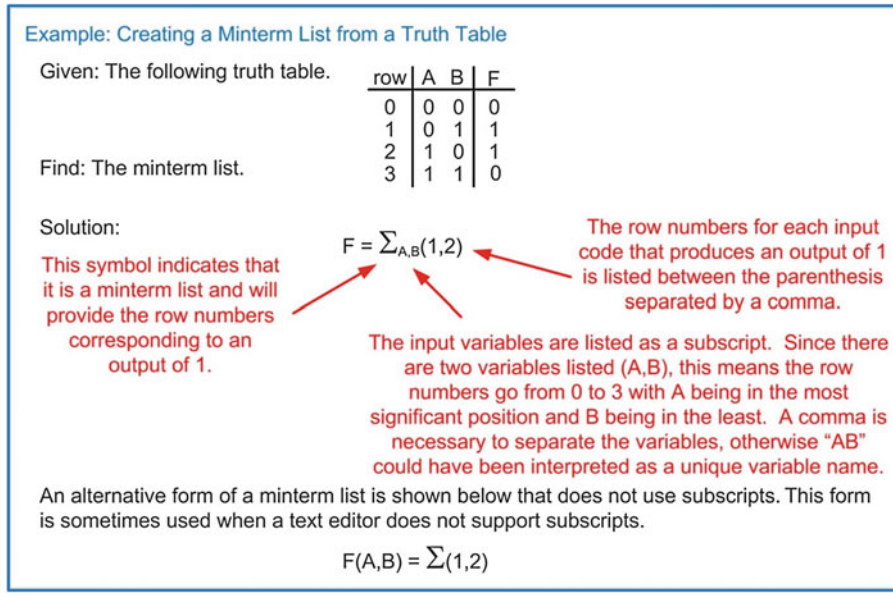


This circuit operates as intended.

Example 4.15
Creating a Canonical Sum of Products Logic Circuit Using Minterms

4.3.2 The Minterm List (Σ)

A *minterm list* is a compact way to describe the functionality of a logic circuit by simply listing the row numbers that correspond to an output of 1 in the truth table. The ∑ symbol is used to denote a minterm list. All input variables must be listed in the order they appear in the truth table. This is necessary because since a minterm list uses only the row numbers to indicate which input codes result in an output of 1, the minterm list must indicate how many variables comprise the row number, which variable is in the most significant position, and which is in the least significant position. After the ∑ symbol, the row numbers corresponding to a true output are listed in a comma-delimited format within parentheses. Example 4.16 shows the process for creating a minterm list from a truth table.



Example 4.16
Creating a Minterm List from a Truth Table

A minterm list contains the same information as the truth table, the canonical sum, and the canonical sum of products logic diagram. Since the minterms themselves are formally defined for an input code, it is trivial to go back and forth between the minterm list and these other forms. Example 4.17 shows how a minterm list can be used to generate an equivalent truth table, canonical sum, and canonical sum of products logic diagram.

Example: Creating Equivalent Functional Representations from a Minterm List

Given: The following minterm list. $F = \sum_{A,B,C}(0,3,7)$

Find: The truth table, canonical sum logic expression and the canonical sum of products logic diagram.

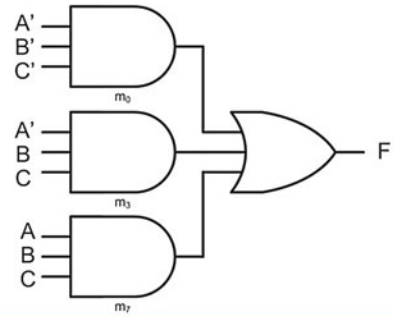
Solution: First, let's generate the **truth table**. From the minterm list subscripts, we know that there are three input variables named A, B and C. These will be listed in the truth table with A in the most significant position and C in the least significant position. We can fill in the input codes as a binary count and insert the row numbers. We can then list the output values that are true. From the minterm list we know that the true outputs are on rows 0, 3 and 7. Since we know we will need the minterm expressions for these rows in the canonical sum, we can also list them in the truth table.

row	A	B	C	F	minterm
0	0	0	0	1	$m_0 = A' \cdot B' \cdot C'$
1	0	0	1	0	-
2	0	1	0	0	-
3	0	1	1	1	$m_3 = A' \cdot B \cdot C$
4	1	0	0	0	-
5	1	0	1	0	-
6	1	1	0	0	-
7	1	1	1	1	$m_7 = A \cdot B \cdot C$

The **canonical sum** is simply the minterm expressions corresponding to a true output OR'd together. Since we already wrote the minterm expressions for rows 0, 3 and 7 (e.g., m_0, m_3 and m_7) in the truth table, we can write the canonical sum directly.

$$F = A' \cdot B' \cdot C' + A' \cdot B \cdot C + A \cdot B \cdot C$$

The **canonical sum of products logic diagram** is simply the gate level depiction of the canonical sum. When logic diagrams get larger, it is acceptable to indicate a variable's complement as a prime instead of placing individual inverters and drawing connection wires that cross each other. It is implied that multiple listings of a variable's complement (e.g., A' in m_0 and m_3) will come from the same inverter.



Example 4.17
Creating Equivalent Functional Representations from a Minterm List

4.3.3 Canonical Product of Sums (POS)

Another technique to directly synthesize a logic circuit from a truth table is to use a canonical product of sums topology based on **maxterms**. A maxterm is a sum term (i.e., an OR operation) that will be false for one and only one input code. The maxterm must contain every literal in its expression. Complements are applied to the input variables as necessary in order to produce a false output for the individual input code. Figure 4.15 shows the definition and gate-level depiction of a maxterm expression. Each maxterm can be denoted using the upper case "M" with the row number as a subscript.

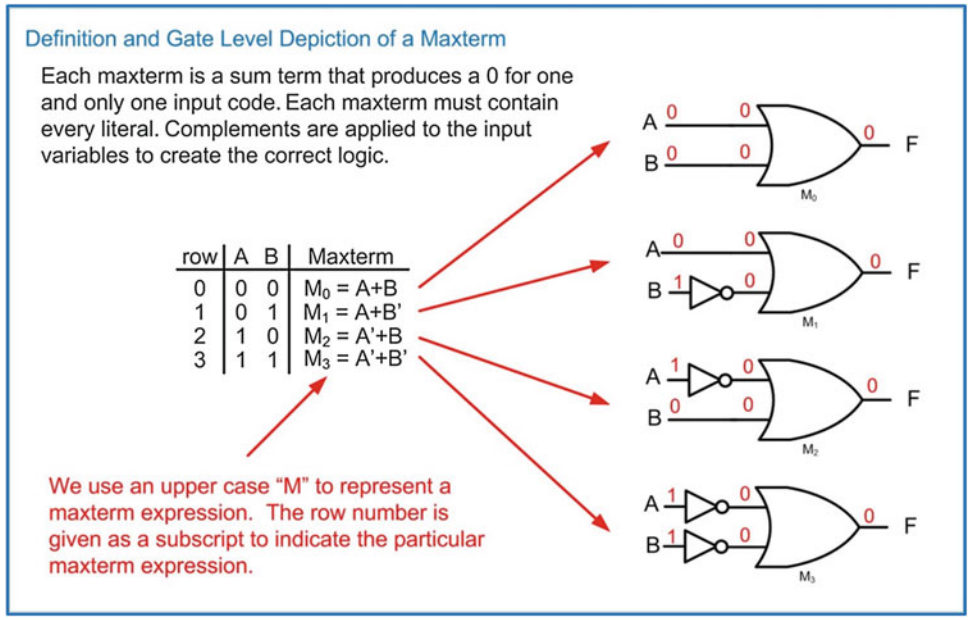


Fig. 4.15
Definition and gate level depiction of a maxterm

For an arbitrary truth table, a maxterm can be used for each row corresponding to a false output. If each of these maxterms outputs are fed into a single AND gate, then a product of sums logic circuit is formed that will produce the logic listed in the truth table. In this topology, any input code that corresponds to an output of 0 will cause its corresponding maxterm to output a 0. Since a 0 on any input of an AND gate will cause the output to go to a 0, the output of the maxterm is passed to the final result. Example 4.18 shows this process. This approach is complementary to the sum of products approach. In the sum of products approach based on minterms, the circuit operates by producing 1s that are passed to the output for the rows that require a true output. For all other rows, the output is false. A product of sums approach based on maxterms operates by producing 0s that are passed to the output for the rows that require a false output. For all other rows, the output is true. These two approaches produce the equivalent logic functionality. Again, at this point no effort has been taken to minimize the logic expression. This unminimized form is called a **canonical product**. The canonical product is logically correct, but uses the most amount of circuitry possible for a given truth table. This canonical product can be the starting point for minimization using the Boolean algebra theorems.

Example: Creating a Canonical Product of Sums Logic Circuit using Maxterms

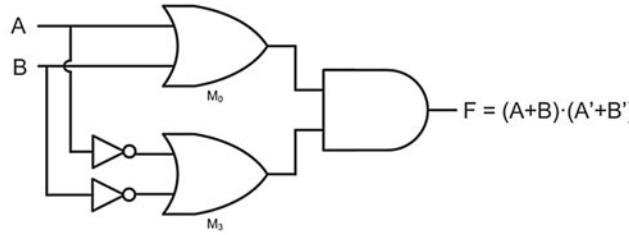
Given: The following truth table.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

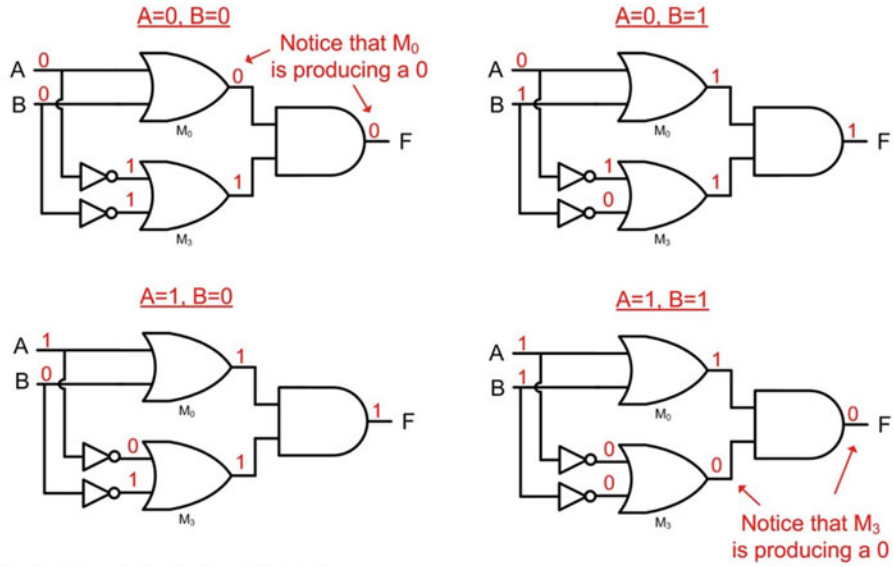
Find: The Canonical POS.

Solution: Let's first start by writing the maxterms for the rows that correspond to a 0 on the output. These can then be implemented using inverters and OR gates. The final step is to feed the outputs of each maxterm circuit into a single AND gate.

row	A	B	Maxterm
0	0	0	M_0=A+B
1	0	1	-
2	1	0	-
3	1	1	M_3=A'+B'



Let's now check that this circuit performs as intended by testing it under each input code for A and B and observing the output F.

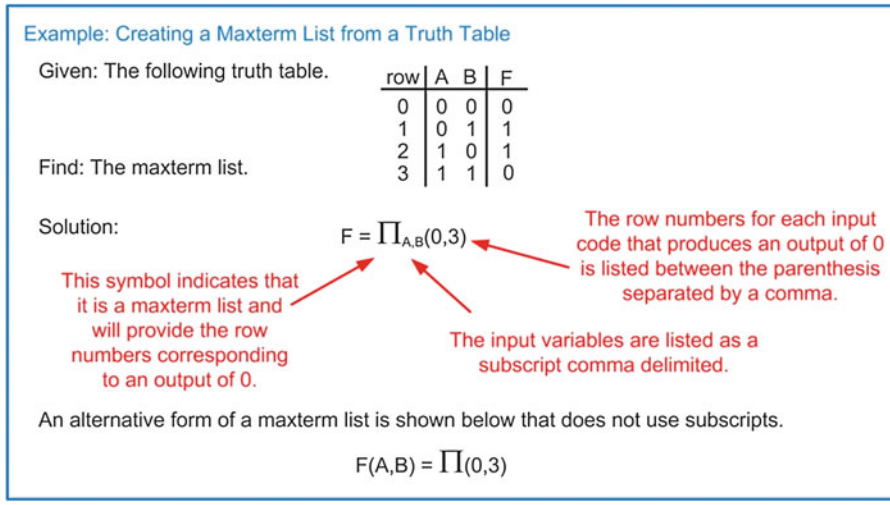


This circuit operates as intended.

Example 4.18
Creating a Product of Sums Logic Circuit Using Maxterms

4.3.4 The Maxterm List (Π)

A maxterm list is a compact way to describe the functionality of a logic circuit by simply listing the row numbers that correspond to an output of 0 in the truth table. The Π symbol is used to denote a maxterm list. All literals used in the logic expression must be listed in the order they appear in the truth table. After the Π symbol, the row numbers corresponding to a false output are listed in a comma-delimited format within parentheses. Example 4.19 shows the process for creating a maxterm list from a truth table.



Example 4.19
Creating a Maxterm List from a Truth Table

A maxterm list contains the same information as the truth table, the canonical product, and the canonical product of sums logic diagram. Example 4.20 shows how a maxterm list can be used to generate these equivalent forms.

Example: Creating Equivalent Functional Representations from a Maxterm List

Given: The following maxterm list. $F = \prod_{A,B,C}(1,2,4,5,6)$

Find: The truth table, canonical product logic expression and the canonical product of sums logic diagram.

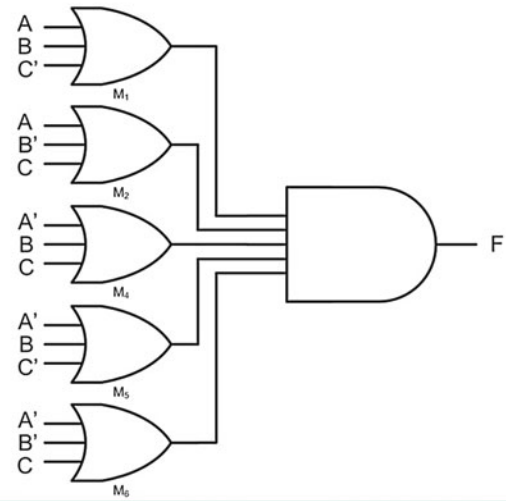
Solution: First, let's generate the **truth table**. From the maxterm list subscripts, we know that there are three input variables named A, B and C that will be used in the truth table in that order. We can fill in the input codes as a binary count and insert the row numbers. We then can list the output values that are false. From the maxterm list we know that the false outputs are on rows 1, 2, 4, 5 and 6. Since we know we will need the maxterm expressions for these rows in the canonical product, we can also list them in the truth table.

row	A	B	C	F	Maxterm
0	0	0	0	1	-
1	0	0	1	0	M_1=A+B+C'
2	0	1	0	0	M_2=A+B'+C
3	0	1	1	1	-
4	1	0	0	0	M_4=A'+B+C
5	1	0	1	0	M_5=A'+B+C'
6	1	1	0	0	M_6=A'+B'+C
7	1	1	1	1	-

The **canonical product** is simply the maxterm expressions corresponding to a false output AND'd together. Since we already wrote these maxterm expressions in the truth table (M_1, M_2, M_4, M_5 and M_6) we can write the canonical product directly.

$$F = (A+B+C')\cdot(A+B'+C)\cdot(A'+B+C)\cdot(A'+B+C')\cdot(A'+B'+C)$$

The **canonical product of sums logic diagram** is simply the gate level depiction of the canonical product.



Example 4.20
Creating Equivalent Functional Representations from a Maxterm List

4.3.5 Minterm and Maxterm List Equivalence

The examples in Examples 4.17 and 4.20 illustrate how minterm and maxterm lists produce the exact same logic functionality but in a complementary fashion. It is trivial to switch back and forth between minterm lists and maxterm lists. This is accomplished by simply changing the list type

(i.e., min to max, max to min) and then switching the row numbers between those listed and those not listed. Example 4.21 shows multiple techniques for representing equivalent logic functionality as a truth table.

Example: Creating Equivalent Forms to Represent Logic Functionality

Given: The following minterm list.

row	A	B	F
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

Find: All equivalent forms to describe the same functionality as the truth table.

Solution: Let's start by writing the **minterm list** and the **maxterm list**. These two lists are equivalent to each other. Remember that the minterm list provides the row numbers corresponding to an output of true while the maxterm list provides the row numbers corresponding to an output of false.

$$F = \sum_{A,B}(0,3) = \prod_{A,B}(1,2)$$

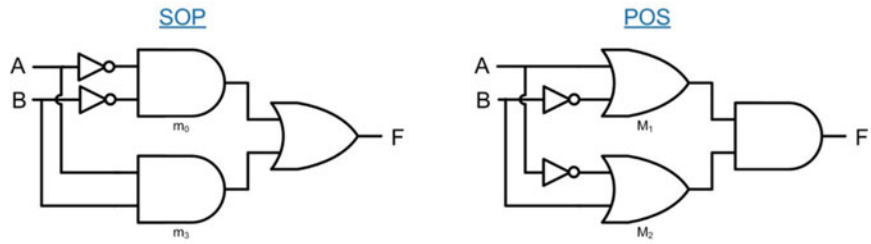
Let's write the minterm and maxterm expressions in the truth table. These will be used when creating the canonical sum and product expressions.

row	A	B	F	minterm	maxterm
0	0	0	1	$m_0 = A' \cdot B'$	-
1	0	1	0	-	$M_1 = A+B'$
2	1	0	0	-	$M_2 = A'+B$
3	1	1	1	$m_3 = A \cdot B$	-

Now let's write the **canonical sum** and **canonical product** logic expressions using these minterms and maxterms. Remember that a canonical sum is simply all of the minterms corresponding to an output of true OR'd together, and a canonical product is simply all of the maxterms corresponding to an output of false AND'd together.

$$F = A' \cdot B' + A \cdot B = (A+B') \cdot (A'+B)$$

Finally, let's draw the **canonical sum of products logic diagram** and the **canonical product of sums logic diagram**.



Example 4.21
Creating Equivalent Forms to Represent Logic Functionality

CONCEPT CHECK

CC4.3 All logic functions can be implemented equivalently using either a canonical sum of products (SOP) or canonical product of sums (POS) topology. Which of these statements is true with respect to selecting a topology that requires the least amount of gates.

A) Since a minterm list and a maxterm list can both be written to describe the same logic functionality, the number of gates in an SOP and POS will always be the same.

B) If a minterm list has over half of its row numbers listed, an SOP topology will require fewer gates than a POS.

C) A POS topology always requires more gates because it needs additional logic to convert the inputs from positive to negative logic.

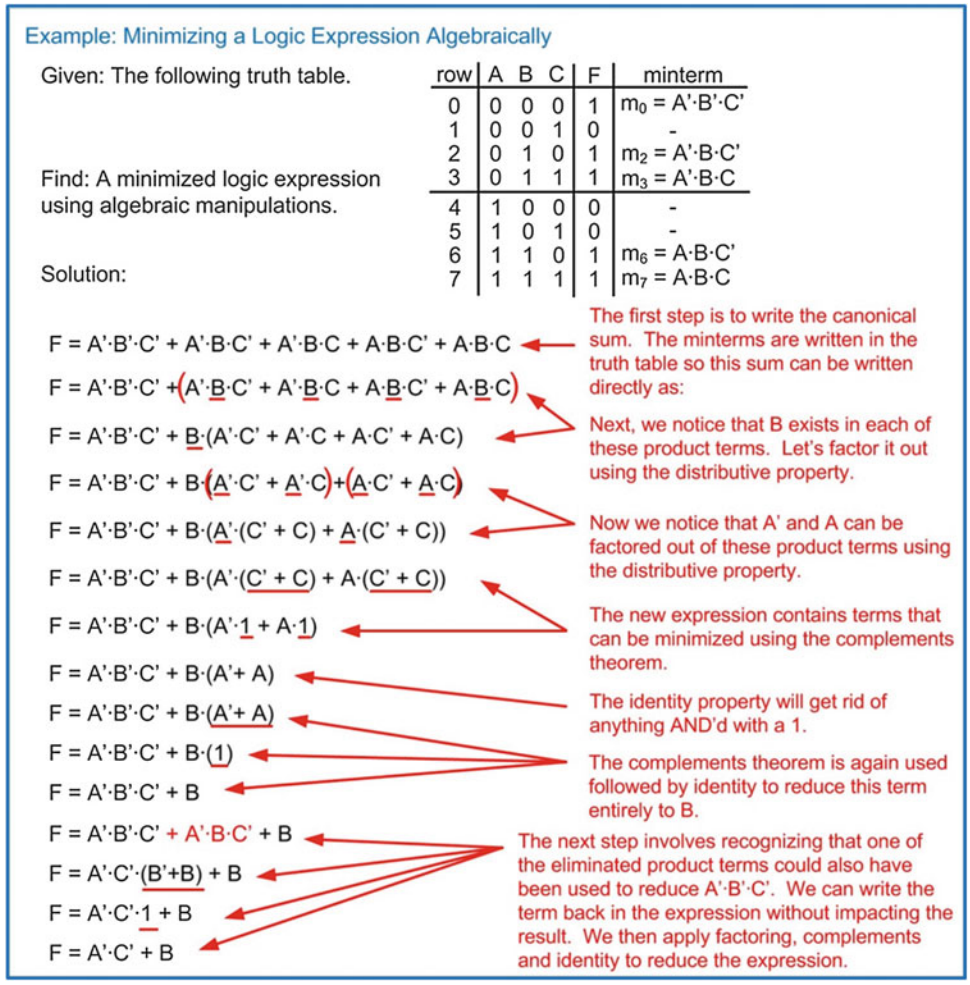
D) If a minterm list has over half of its row numbers listed, a POS topology will require fewer gates than SOP.

4.4 Logic Minimization

We now look at how to reduce the canonical expressions into equivalent forms that use less logic. This minimization is key to reducing the complexity of the logic prior to implementing in real circuitry. This reduces the amount of gates needed, placement area, wiring, and power consumption of the logic circuit.

4.4.1 Algebraic Minimization

Canonical expressions can be reduced algebraically by applying the theorems covered in prior sections. This process typically consists of a series of factoring based on the distributive property followed by replacing variables with constants (i.e., 0s and 1s) using the complements theorem. Finally, constants are removed using the identity theorem. Example 4.22 shows this process.



Example 4.22
Minimizing a Logic Expression Algebraically

The primary drawback of this approach is that it requires recognition of where the theorems can be applied. This can often lead to missed minimizations. Computer automation is often the best mechanism to perform this minimization for large logic expressions.

4.4.2 Minimization Using Karnaugh Maps

A Karnaugh map is a graphical way to minimize logic expressions. This technique is named after Maurice Karnaugh, American physicist, who introduced the map in its latest form in 1953 while working at Bell Labs. The Karnaugh map (or K-map) is a way to put a truth table into a form that allows logic minimization through a graphical process. This technique provides a graphical process that accomplishes the same result as factoring variables via the distributive property and removing variables via the complements and identity theorems. K-maps present a truth table in a form that allows variables to be removed from the final logic expression in a graphical manner.

4.4.2.1 Formation of a K-map

A K-map is constructed as a two-dimensional grid. Each *cell* within the map corresponds to the output for a specific input code. The cells are positioned such that neighboring cells only differ by one bit in their input codes. Neighboring cells are defined as cells immediately adjacent horizontally and immediately adjacent vertically. Two cells positioned diagonally next to each other are not considered neighbors. The input codes for each variable are listed along the top and side of the K-map. Consider the construction of a 2-input K-map shown in Fig. 4.16.

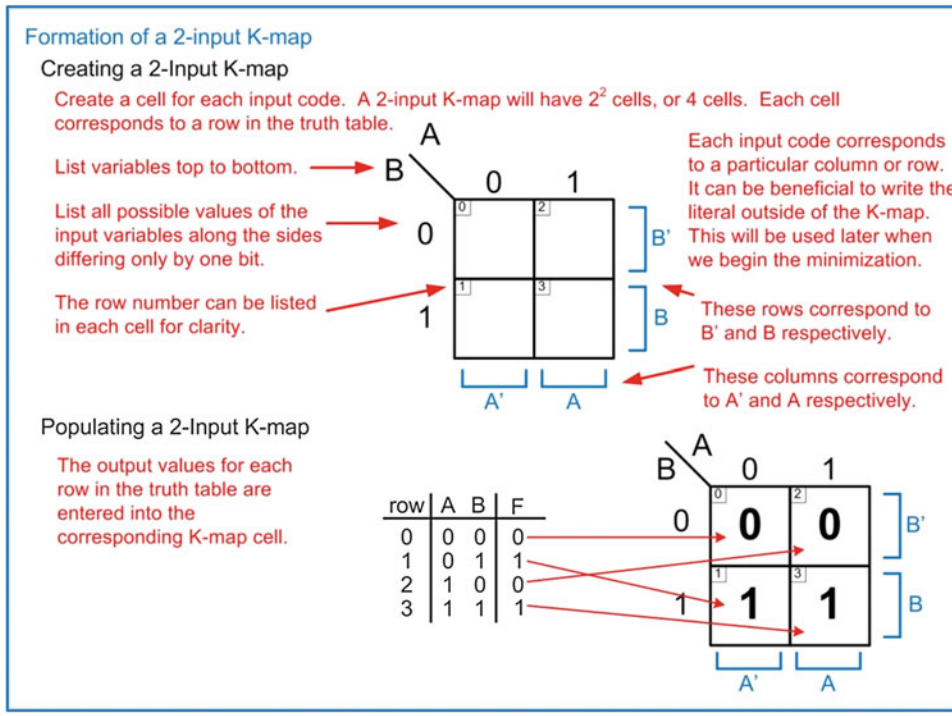


Fig. 4.16
Formation of a 2-input K-map

When constructing a 3-input K-map, it is important to remember that each input code can only differ from its neighbor by one bit. For example, the two codes 01 and 10 differ by two bits (i.e., the MSB is different and the LSB is different); thus they could not be neighbors; however, the codes 01-11 and 11-10 can be neighbors. As such, the input codes along the top of the 3-input K-map must be ordered accordingly (i.e., 00-01-11-10). Consider the construction of a 3-input K-map shown in Fig. 4.17. The rows and columns that correspond to the input literals can now span multiple rows and columns. Notice how in this 3-input K-map, the literals A, A′, B, and B′ all correspond to two columns. Also, notice that B′ spans two columns, but the columns are on different edges of the K-map. The side edges of the 3-input K-map are still considered neighbors because the input codes for these columns only differ by one bit. This is an important attribute once we get to the minimization of variables because it allows us to examine an input literal's impact not only within the obvious adjacent cells but also when the variables *wrap* around the edges of the K-map.

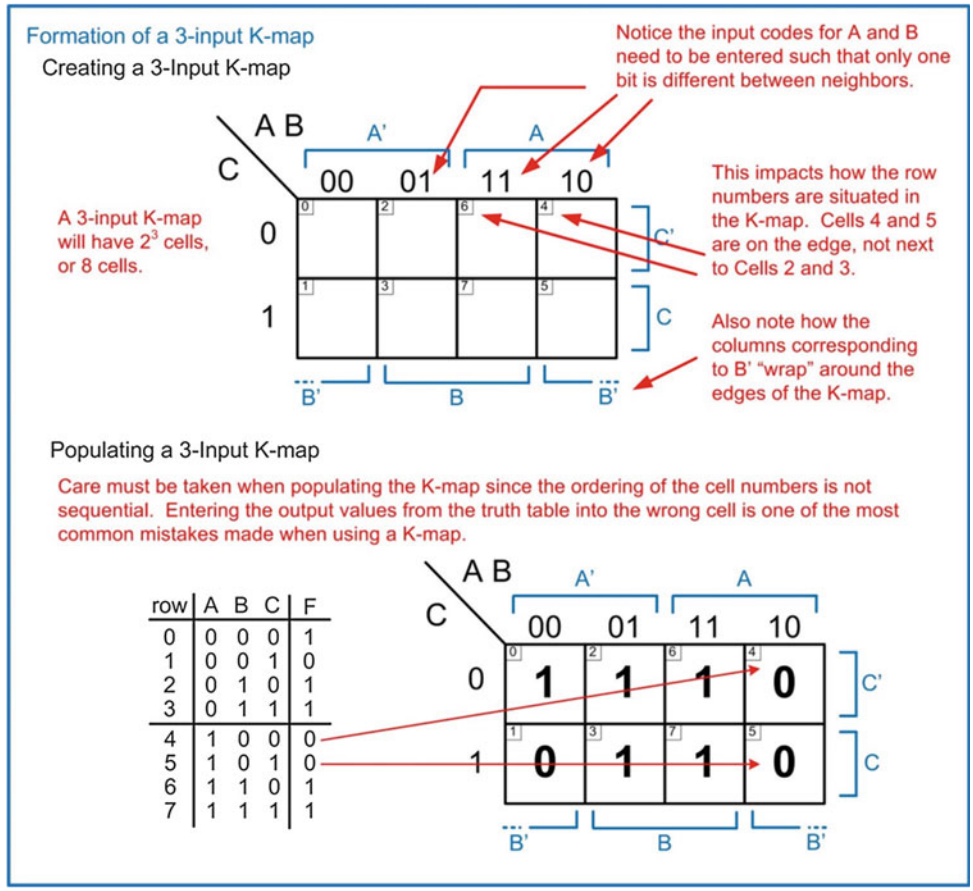


Fig. 4.17
Formation of a 3-input K-map

When constructing a 4-input K-map, the same rules apply that the input codes can only differ from their neighbors by one bit. Consider the construction of a 4-input K-map in Fig. 4.18. In a 4-input K-map, neighboring cells can wrap around both the top-to-bottom edges in addition to the side-to-side edges. Notice that all 16 cells are positioned within the map so that their neighbors on the top, bottom, and sides only differ by one bit in their input codes.

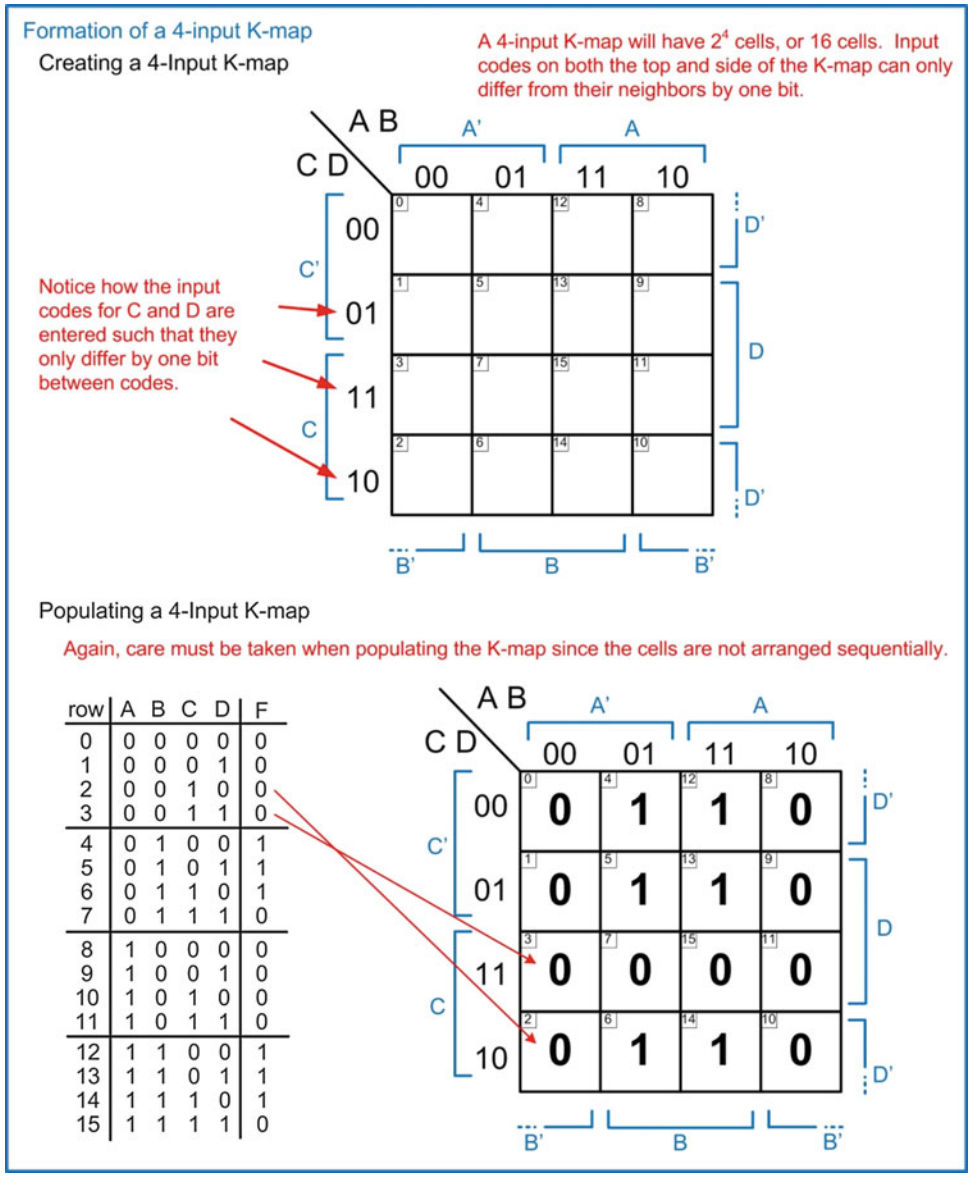


Fig. 4.18
Formation of a 4-input K-map

4.4.2.2 Logic Minimization Using K-maps (Sum of Products)

Now we look at using a K-map to create a minimized logic expression in an SOP form. Remember that each cell with an output of 1 has a minterm associated with it, just as in the truth table. When two neighboring cells have outputs of 1, it graphically indicates that the two minterms can be reduced into a minimized product term that will cover both outputs. Consider the example given in Fig. 4.19.

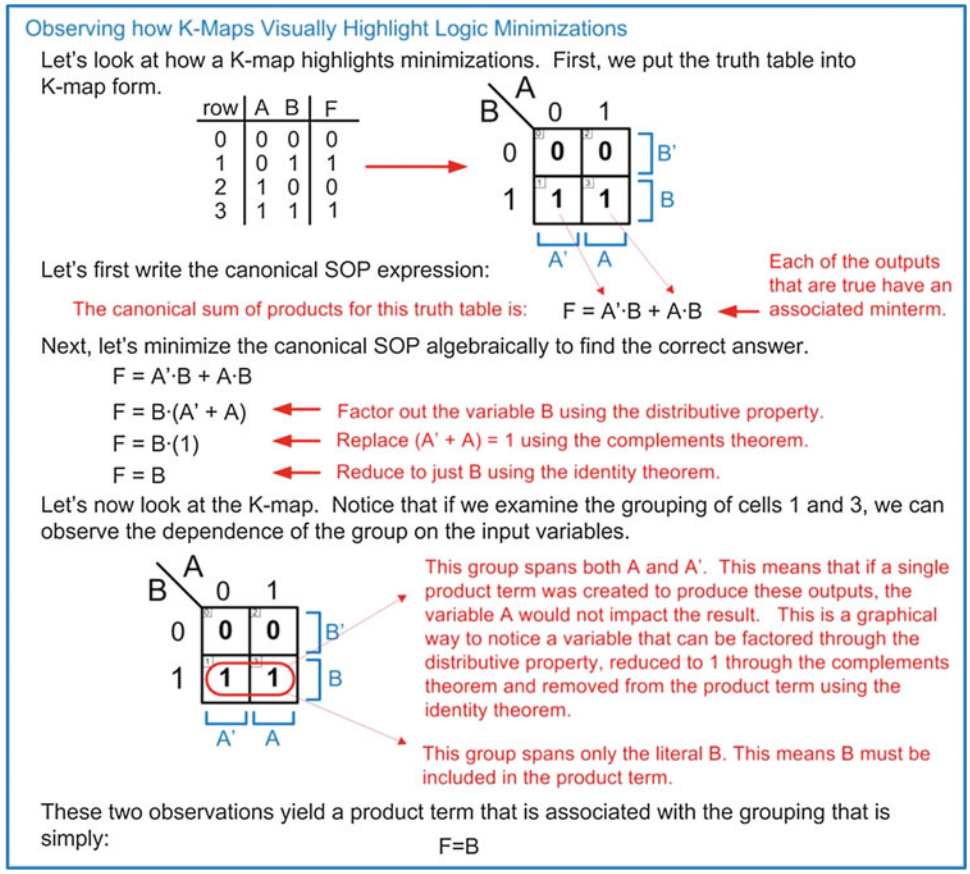


Fig. 4.19
Observing how K-maps visually highlight logic minimizations

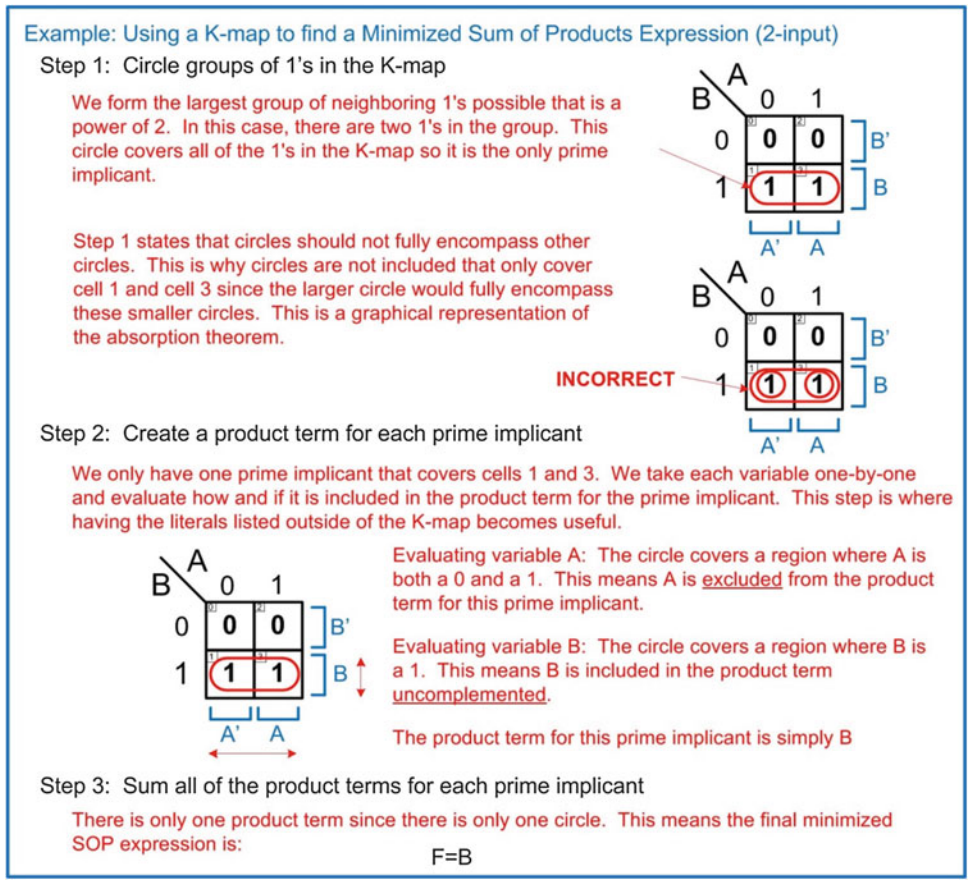
These observations can be put into a formal process to produce a minimized SOP logic expression using a K-map. The steps are as follows:

1. Circle groups of 1s in the K-map following the rules:
   - Each circle should contain the largest number of 1s possible.
   - The circles encompass only neighboring cells (i.e., side-to-side sides and/or top and bottom).
   - The circles must contain a number of 1s that is a power of 2 (i.e., 1, 2, 4, 8, or 16).
   - Enter as many circles as possible without having any circles fully cover another circle.
   - Each circle is called a *Prime Implicant*.
2. Create a product term for each prime implicant following the rules:
   - Each variable in the K-map is evaluated one by one.
   - If the circle covers a region where the input variable is a 1, then include it in the product term uncomplemented.
   - If the circle covers a region where the input variable is a 0, then include it in the product term complemented.

- If the circle covers a region where the input variable is both a 0 and 1, then the variable is excluded from the product term.

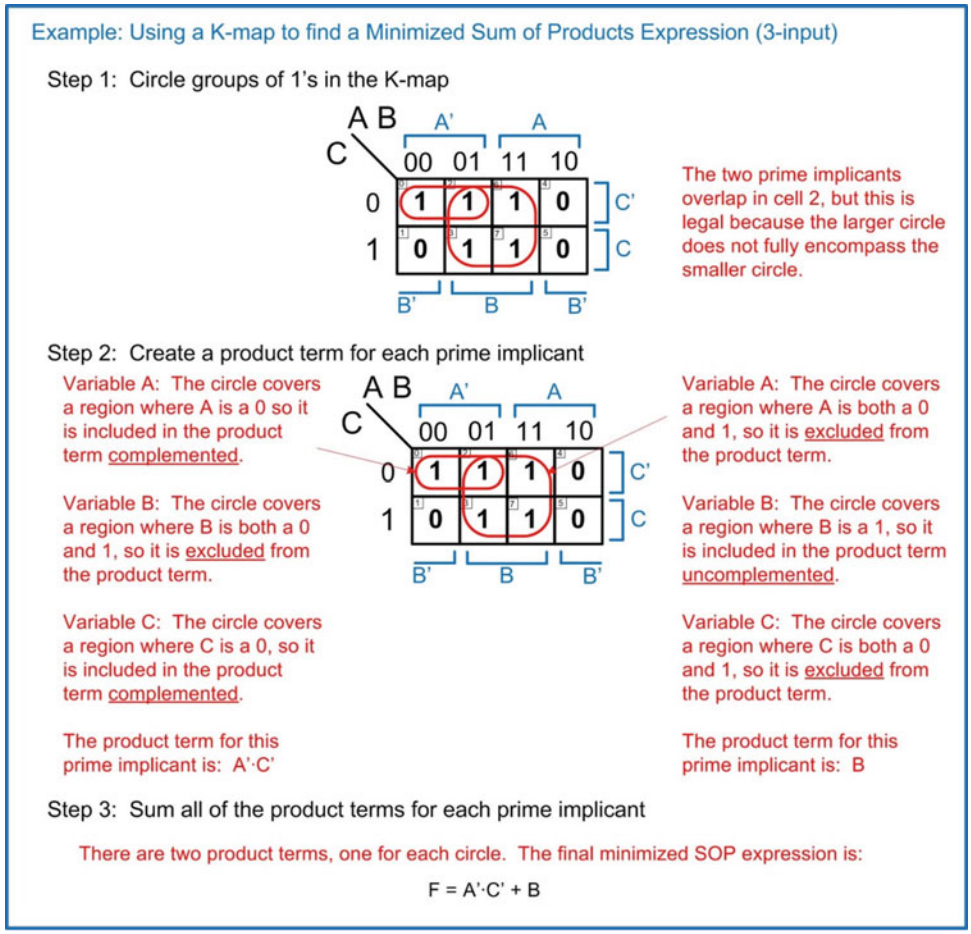
3. Sum all of the product terms for each prime implicant.

Let's apply this approach to our 2-input K-map example. Example 4.23 shows the process of finding a minimized sum of products logic expression for a 2-input logic circuit using a K-map. This process yielded the same SOP expression as the algebraic minimization and observations shown in Fig. 4.19, but with a formalized process.



Example 4.23
Using a K-map to Find a Minimized Sum of Products Expression (2-Input)

Let's now apply this process to our 3-input K-map example. Example 4.24 shows the process of finding a minimized sum of products logic expression for a 3-input logic circuit using a K-map. This example shows circles that overlap. This is legal as long as one circle does not fully encompass another. Overlapping circles are common since the K-map process dictates that circles should be drawn that group the largest number of ones possible as long as they are in powers of 2. Forming groups of ones using ones that have already been circled is perfectly legal to accomplish larger groupings. The larger the grouping of ones, the more chance there is for a variable to be excluded from the product term. This results in better minimization of the logic.

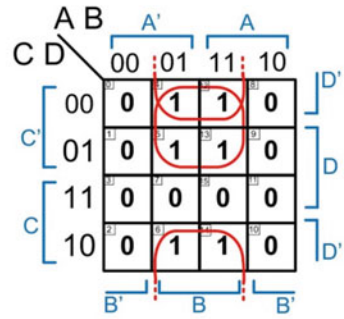


Example 4.24
Using a K-map to Find a Minimized Sum of Products Expression (3-Input)

Let's now apply this process to our 4-input K-map example. Example 4.25 shows the process of finding a minimized sum of products logic expression for a 4-input logic circuit using a K-map.

Example: Using a K-map to find a Minimized Sum of Products Expression (4-input)

Step 1: Circle groups of 1's in the K-map



Circles can be drawn that "wrap" around the edges. Notice that the input codes for cells 4 and 12 only differ by 1 bit from cells 6 and 14. This makes them neighbors and grouping these 4 cells together is legal.

Again, circles that overlap are legal as long as one circle does not fully encompass another.

Step 2: Create a product term for each prime implicant

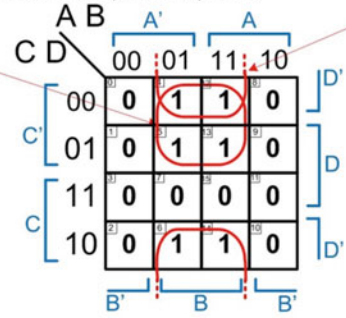
Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the product term.

Variable B: The circle covers a region where B is a 1, so it is included in the product term uncomplemented.

Variable C: The circle covers a region where C is a 0, so it is included in the product term complemented.

Variable D: The circle covers a region where D is both a 0 and 1, so it is excluded from the product term.

The product term for this prime implicant is: B·C'



Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the product term.

Variable B: The circle covers a region where B is a 1, so it is included in the product term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the product term.

Variable D: The circle covers a region where D is a 0, so it is included in the product term complemented.

The product term for this prime implicant is: B·D'

Step 3: Sum all of the product terms for each prime implicant

There are two product terms, one for each circle. The final minimized SOP expression is:

$$F = B \cdot C' + B \cdot D'$$

This expression could be further factored using the distributive property to F = B·(C' + D') to eliminate one more logic operation; however, since the problem asked for an SOP form, this last step was not necessary. Also, leaving a logic expression in an SOP form allows it to be directly converted into a NAND gate only implementation using DeMorgan's Theorem if the target logic family is CMOS.

Example 4.25
Using a K-map to Find a Minimized Sum of Products Expression (4-Input)

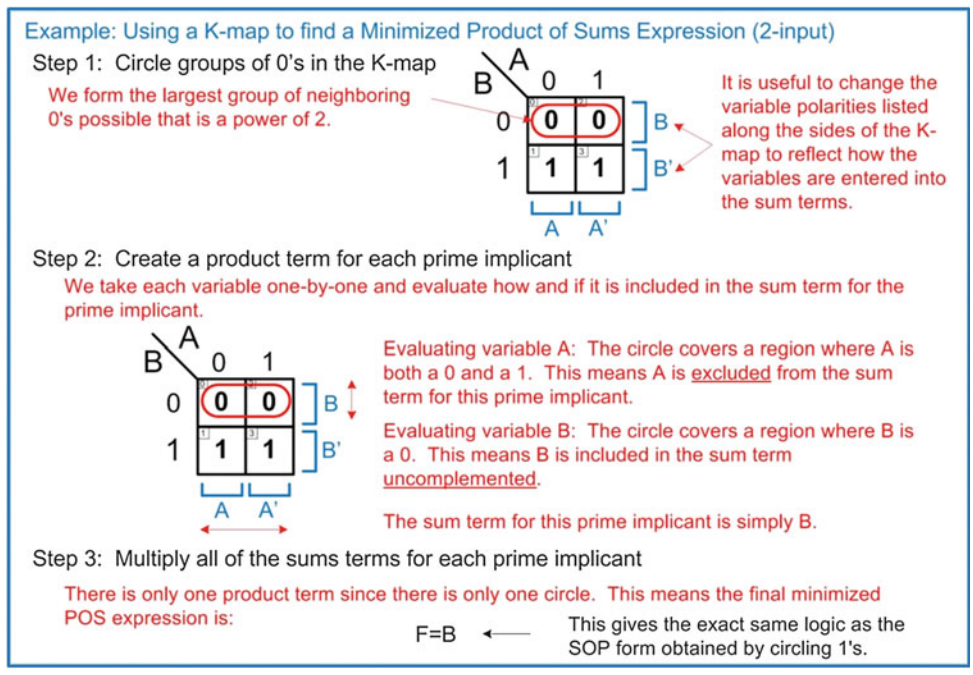
4.4.2.3 Logic Minimization Using K-maps (Product of Sums)

K-maps can also be used to create minimized product of sums logic expressions. This is the same concept as how a minterm list and maxterm list each produces the same logic function, but in complementary fashions. When creating a product of sums expression from a K-map, groups of 0s are circled. For each circle, a sum term is derived with a negation of variables similar to when forming a maxterm

(i.e., in the input variable is a 0, then it is included uncomplemented in the sum term and vice versa). The final step in forming the minimized POS expression is to AND all of the sum terms together. The formal process is as follows:

1. Circle groups of 0s in the K-map following the rules:
   - Each circle should contain the largest number of 0s possible.
   - The circles encompass only neighboring cells (i.e., side-to-side sides and/or top and bottom).
   - The circles must contain a number of 0s that is a power of 2 (i.e., 1, 2, 4, 8, or 16).
   - Enter as many circles as possible without having any circles fully cover another circle.
   - Each circle is called a *prime implicant*.
2. Create a sum term for each prime implicant following the rules:
   - Each variable in the K-map is evaluated one by one.
   - If the circle covers a region where the input variable is a 1, then include it in the sum term complemented.
   - If the circle covers a region where the input variable is a 0, then include it in the sum term uncomplemented.
   - If the circles cover a region where the input variable is both a 0 and 1, then the variable is excluded from the sum term.
3. Multiply all of the sum terms for each prime implicant.

Let's apply this approach to our 2-input K-map example. Example 4.26 shows the process of finding a minimized product of sums logic expression for a 2-input logic circuit using a K-map. Notice that this process yielded the same logic expression as the SOP approach shown in Example 4.23. This illustrates that both the POS and SOP expressions produce the correct logic for the circuit.

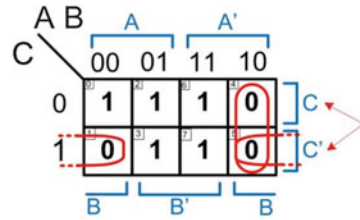


Example 4.26
Using a K-map to Find a Minimized Product of Sums Expression (2-Input)

Let's now apply this process to our 3-input K-map example. Example 4.27 shows the process of finding a minimized product of sums logic expression for a 3-input logic circuit using a K-map. Notice that the logic expression in POS form is not identical to the SOP expression found in Example 4.24; however, using a few steps of algebraic manipulation shows that the POS expression can be put into a form that *is* identical to the prior SOP expression. This illustrates that both the POS and SOP produce equivalent functionality for the circuit.

Example: Using a K-map to find a Minimized Product of Sums Expression (3-input)

Step 1: Circle groups of 0's in the K-map



Again, the polarities of the variables along K-map are changed to reflect how the variables are entered into the sum terms.

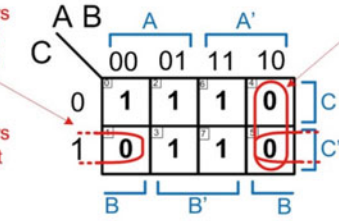
Step 2: Create a sum term for each prime implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is a 1, so it is included in the sum term complemented.

The sum term for this prime implicant is: B+C'



Variable A: The circle covers a region where A is a 1, so it is included in the sum term complemented.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the sum term.

The sum term for this prime implicant is: A'+B

Step 3: Multiply all of the sum terms for each prime implicant

There are two sum terms, one for each circle. The final minimized POS expression is:

$$F = (B+C')\cdot(A'+B)$$

Check: Is this equivalent to the logic expression obtained using the SOP approach?

From the prior example, the minimized SOP expression was: $F = A'\cdot C' + B$

$F = (B+C')\cdot(A'+B)$ ← Let's use the Boolean algebra theorems to see if this is equal to $A'\cdot C' + B$

$F = B+(C'\cdot A')$ ← Using the distributive property on the POS expression, we can factor out B.

$F = A'\cdot C' + B$ ← The commutative property allows us to rearrange terms to match the SOP expression exactly.

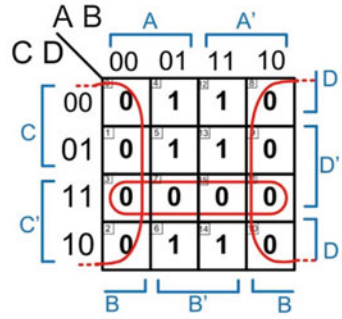
Yes, its POS expression is equivalent to the SOP expression.

Example 4.27
Using a K-map to Find a Minimized Product of Sums Expression (3-Input)

Let's now apply this process to our 4-input K-map example. Example 4.28 shows the process of finding a minimized product of sums logic expression for a 4-input logic circuit using a K-map.

Example: Using a K-map to find a Minimized Product of Sums Expression (4-input)

Step 1: Circle groups of 0's in the K-map



Again, the polarities of the variables along K-map are changed to reflect how the variables are entered into the sum terms.

Step 2: Create a sum term for each prime implicant

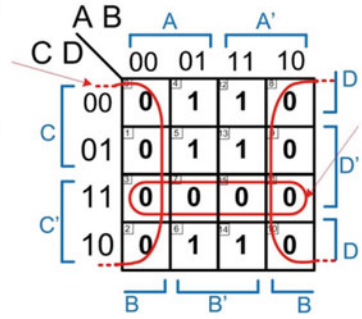
Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the sum term.

Variable D: The circle covers a region where D is both a 0 and 1, so it is excluded from the sum term.

The sum term for this prime implicant is: B



Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is both a 0 and 1, so it is excluded from the sum term.

Variable C: The circle covers a region where C is a 1, so it is included in the sum term complemented.

Variable D: The circle covers a region where D is a 1, so it is included in the sum term complemented.

The sum term for this prime implicant is: C'+D'

Step 3: Multiply all of the sum terms for each prime implicant

There are two sum terms, one for each circle. The final minimized POS expression is:

$$F = (B)\cdot(C'+D')$$

Check: Is this equivalent to the logic expression obtained using the SOP approach?

From the prior example, the minimized SOP expression was: $F = B\cdot C' + B\cdot D'$

$F = (B)\cdot(C'+D')$ ← Let's use the Boolean algebra theorems to see if this is equal to $B\cdot C' + B\cdot D'$

$F = B\cdot C' + B\cdot D'$ ← Using the distributive property on the POS expression shows that this is equal to the minimized SOP expression.

Example 4.28
Using a K-map to Find a Minimized Product of Sums Expression (4-Input)

4.4.2.4 Minimal Sum

One situation that arises when minimizing logic using a K-map is that some of the prime implicants may be redundant. Consider the example in Fig. 4.20.

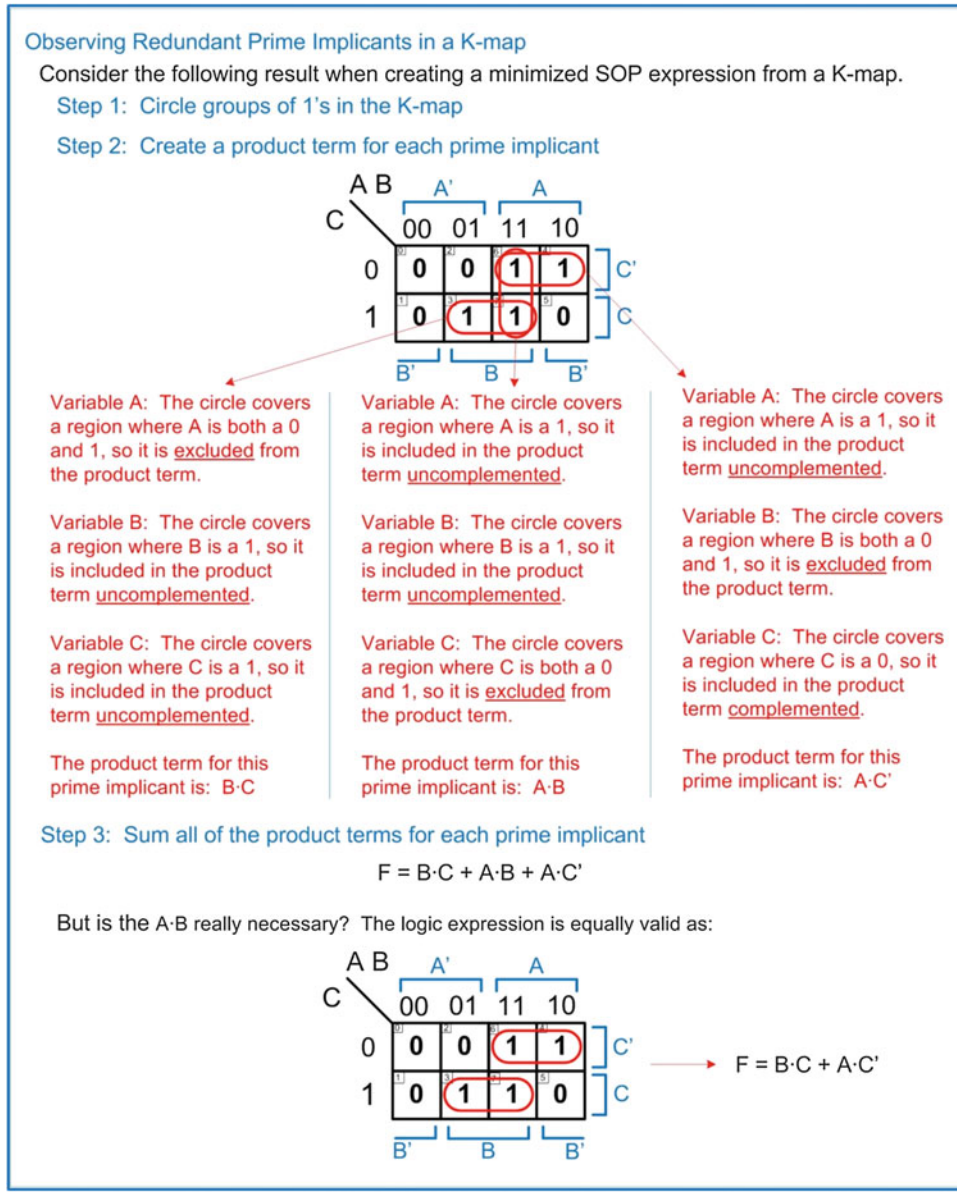
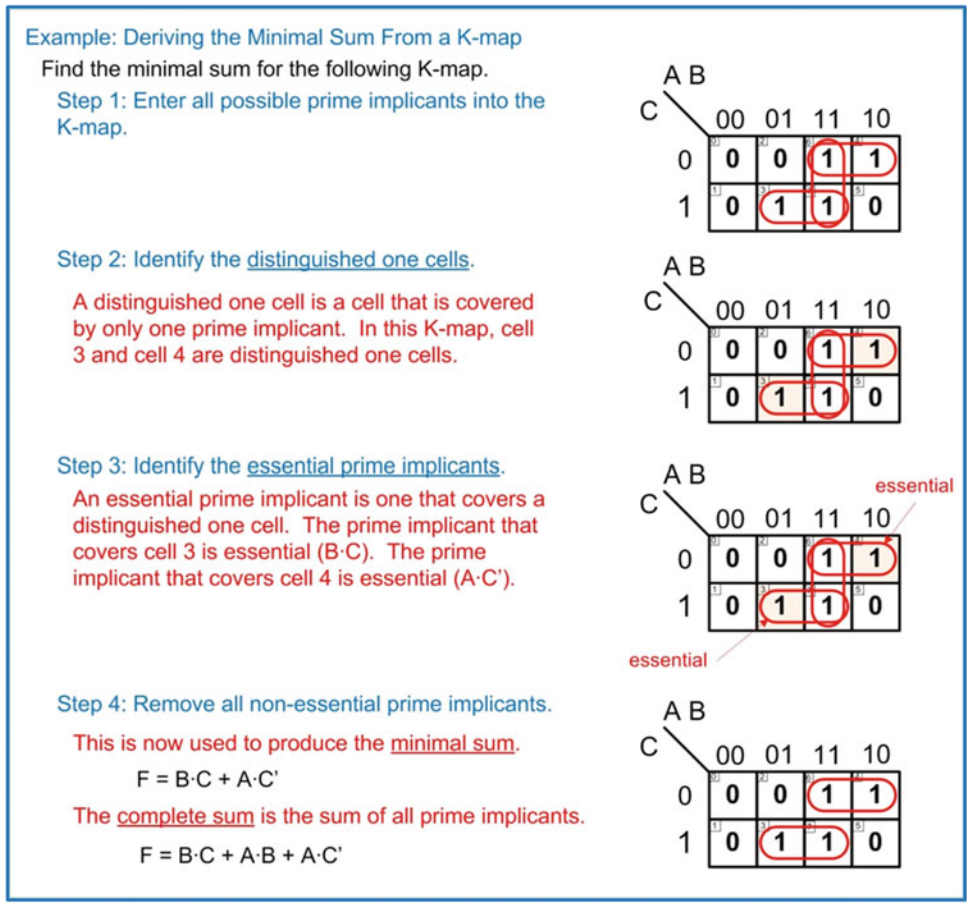


Fig. 4.20
Observing redundant prime implicants in a K-map

We need to define a formal process for identifying redundant prime implicants that can be removed without impacting the result of the logic expression. Let's start with examining the sum of products form. First, we define the term **essential prime implicant** as a prime implicant that *cannot* be removed from the logic expression without impacting its result. We then define the term **minimal sum** as a logic expression that represents the most minimal set of logic operations to accomplish a sum of products form. There may be multiple minimal sums for a given truth table, but each would have the same number of logic operations. In order to determine if a prime implicant is essential, we first put in each and every possible prime implicant into the K-map. This gives a logic expression known as the **complete sum**. From this point we identify any cells that have only one prime implicant covering them. These cells are

called **distinguished one cells**. Any prime implicant that covers a distinguished one cell is defined as an essential prime implicant. All prime implicants that are not essential are removed from the K-map. A minimal sum is then simply the sum of all remaining product terms associated with the essential prime implicants. Example 4.29 shows how to use this process.



Example 4.29
Deriving the Minimal Sum from a K-map

This process is identical for the product of sums form to produce the **minimal product**.

4.4.3 Don't Cares

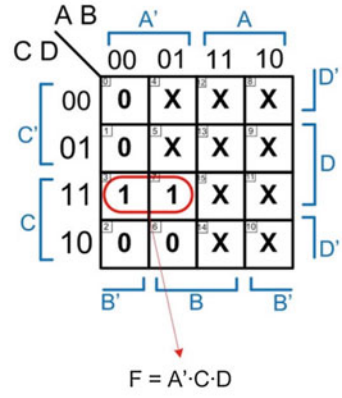
There are often times when framing a design problem that there are specific input codes that require exact output values, but there are other codes where the output value doesn't matter. This can occur for a variety of reasons, such as knowing that certain input codes will never occur due to the nature of the problem or that the output of the circuit will only be used under certain input codes. We can take advantage of this situation to produce a more minimal logic circuit. We define an output as a **don't care** when it doesn't matter whether it is a 1 or 0 for the particular input code. The symbol for a don't care is "X." We take advantage of don't cares when performing logic minimization by treating them as whatever output value will produce a minimal logic expression. Example 4.30 shows how to use this process.

Example: Using Don't Cares to Produce a Minimal SOP Logic Expression

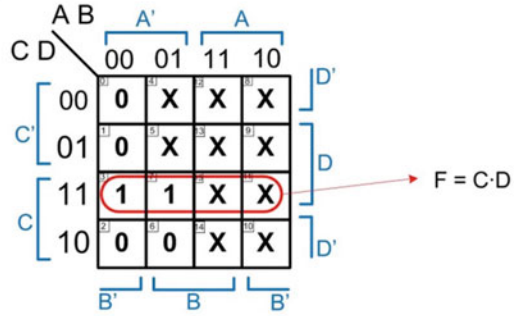
Let's create a minimized sum of products expression by taking advantage of don't cares.

Don't cares are indicated using the symbol "X". These go directly into the K-map. If we initially just circle 1's, we get the following logic expression:

row	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	X
5	0	1	0	1	X
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	X
9	1	0	0	1	X
10	1	0	1	0	X
11	1	0	1	1	X
12	1	1	0	0	X
13	1	1	0	1	X
14	1	1	1	0	X
15	1	1	1	1	X



However, we can take advantage of the don't cares that are in cells 5 and 11. If we treat them as 1's, we can include them in the prime implicant giving a more minimal logic expression.



Don't cares do not need to be circled. They are only included in a grouping if they help produce a more minimal product term for that prime implicant.

Example 4.30
Using Don't Cares to Produce a Minimal SOP Logic Expression

4.4.4 Using XOR Gates

While Boolean algebra does not include the exclusive-OR and exclusive-NOR operations, XOR and XNOR gates do indeed exist in modern electronics. They can be a useful tool to provide logic circuitry with less operations, sometimes even compared to a minimal sum or product synthesized using the techniques just described. An XOR/XNOR operation can be identified by putting the values from a truth table into a K-map. The XOR/XNOR operations will result in a characteristic checkerboard pattern in the K-map. Consider the following patterns for XOR and XNOR gates in Figs. 4.21, 4.22, 4.23, and 4.24.

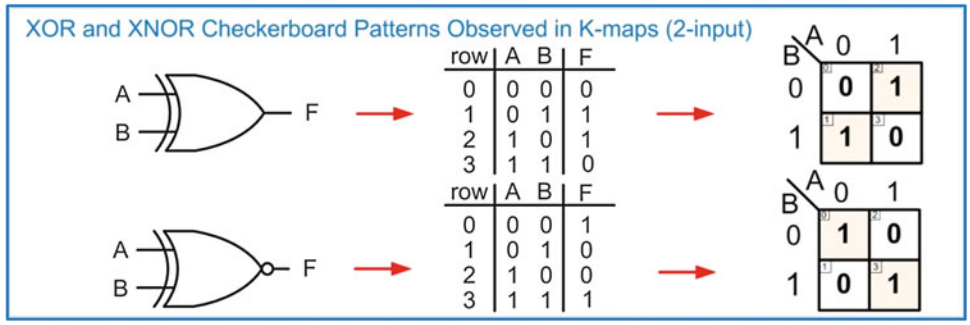


Fig. 4.21
XOR and XNOR checkerboard patterns observed in K-maps (2-input)

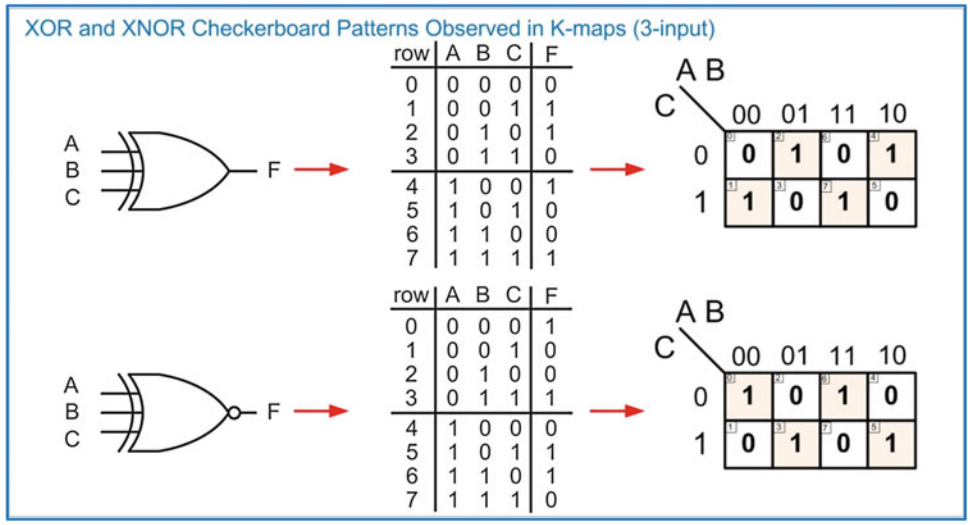


Fig. 4.22
XOR and XNOR checkerboard patterns observed in K-maps (3-input)

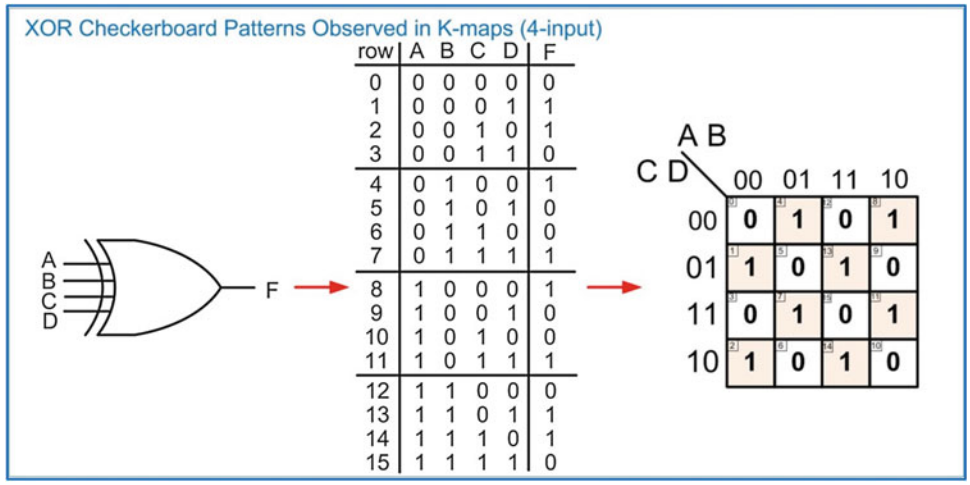


Fig. 4.23
XOR checkerboard pattern observed in K-maps (4-input)

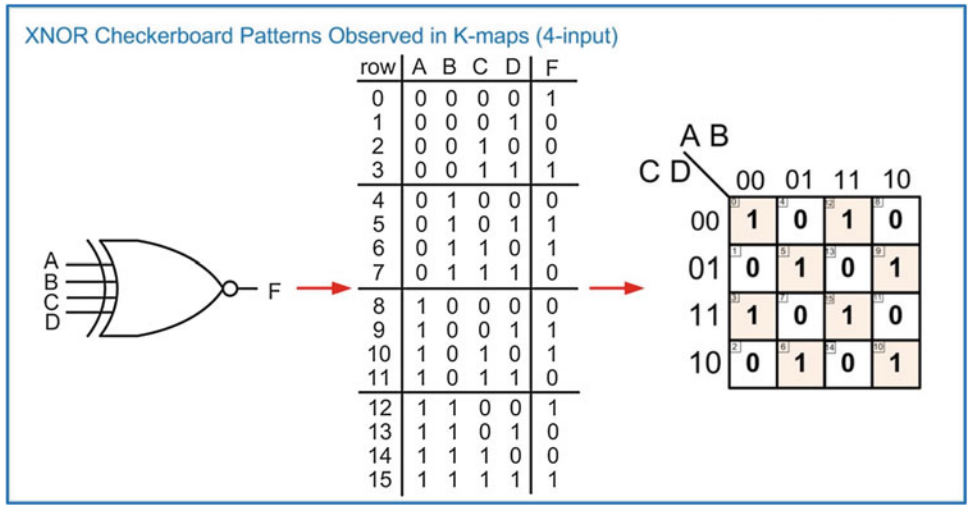


Fig. 4.24
XNOR checkerboard pattern observed in K-maps (4-input)

CONCEPT CHECK

CC4.4(a) Logic minimization is accomplished by removing variables from the original canonical logic expression that don't impact the result. How does a Karnaugh map graphically show what variables can be removed?

A) K-maps contain the same information as a truth table but the data is formatted as a grid. This allows variables to be removed by inspection.

B) K-maps rearrange a truth table so that adjacent cells have one and only one input variable changing at a time. If adjacent cells have the same output value when an input variable is both a 0 and a 1, that variable has no impact on the interim result and can be eliminated.

C) K-maps list both the rows with outputs of 1's and 0's simultaneously. This allows minimization to occur for a SOP and POS topology that each have the same, but minimal, number of gates.

D) K-maps display the truth table information in a grid format, which is a more compact way of presenting the behavior of a circuit.

CC4.4(b) A "Don't Care" can be used to minimize a logic expression by assigning the output of a row to either a 1 or a 0 in order to form larger groupings within a K-map. How does the output of the circuit behave when it processes the input code for a row containing a don't care?

A) The output will be whatever value was needed to form the largest grouping in the K-map.

B) The output will go to either a 0 or a 1, but the final value is random.

C) The output can toggle between a 0 and a 1 when this input code is present.

D) The output will be driven to exactly halfway between a 0 and a 1.

4.5 Timing Hazards and Glitches

Timing hazards, or glitches, refer to unwanted transitions on the output of a combinational logic circuit. These are most commonly due to different delay paths through the gates in the circuit. In real circuitry there is always a finite propagation delay through each gate. Consider the circuit shown in Fig. 4.25 where gate delays are included and how they can produce unwanted transitions.

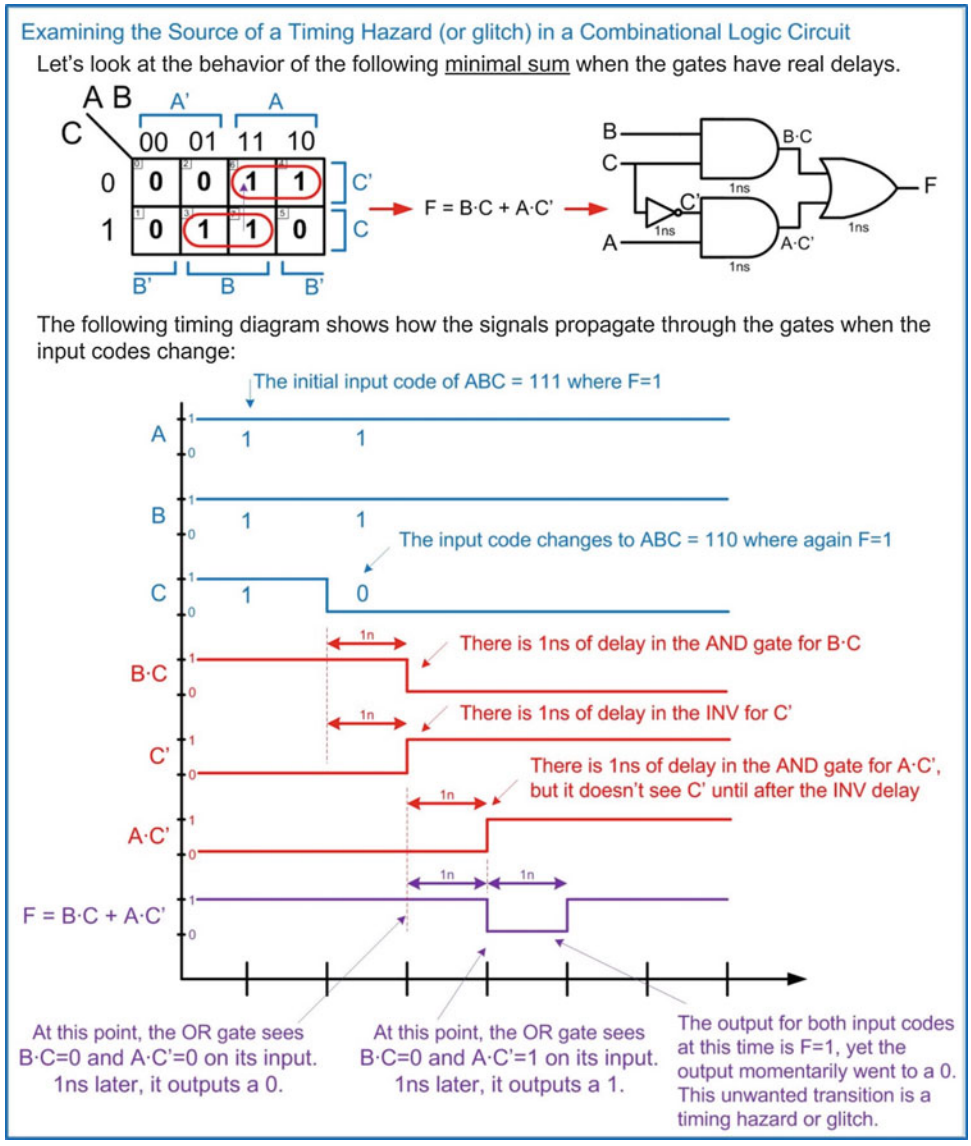


Fig. 4.25
Examining the source of a timing hazard (or glitch) in a combinational logic circuit

These timing hazards are given unique names based on the type of transition that occurs. A **static 0** timing hazard is when the input switches between two input codes that both yield an output of 0 but the output momentarily switches to a 1. A **static 1** timing hazard is when the input switches between two

input codes that both yield an output of 1 but the output momentarily switches to a 0. A **dynamic hazard** is when the input switches between two input codes that result in a real transition on the output (i.e., 0 to 1 or 1 to 0), but the output has a momentary glitch before reaching its final value. These definitions are shown in Fig. 4.26.

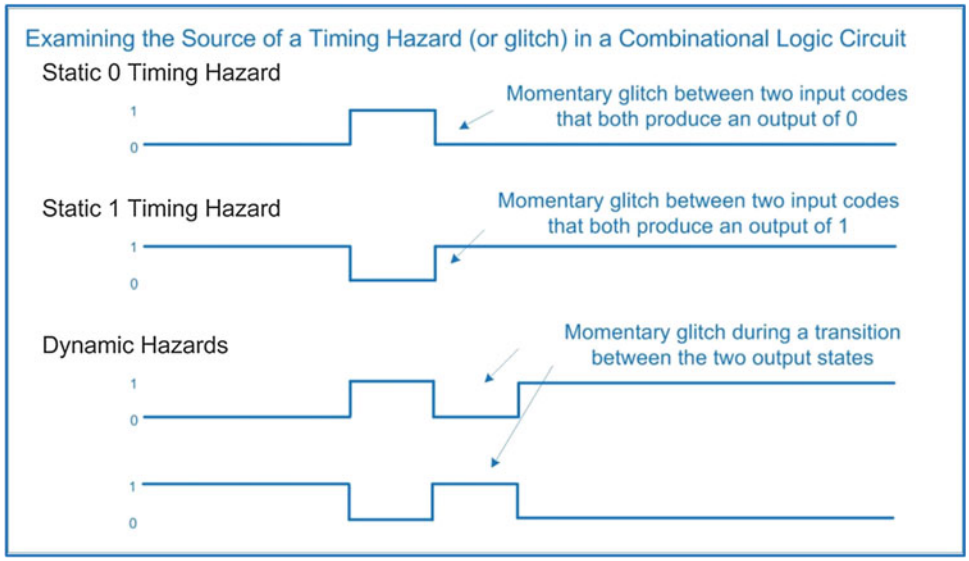
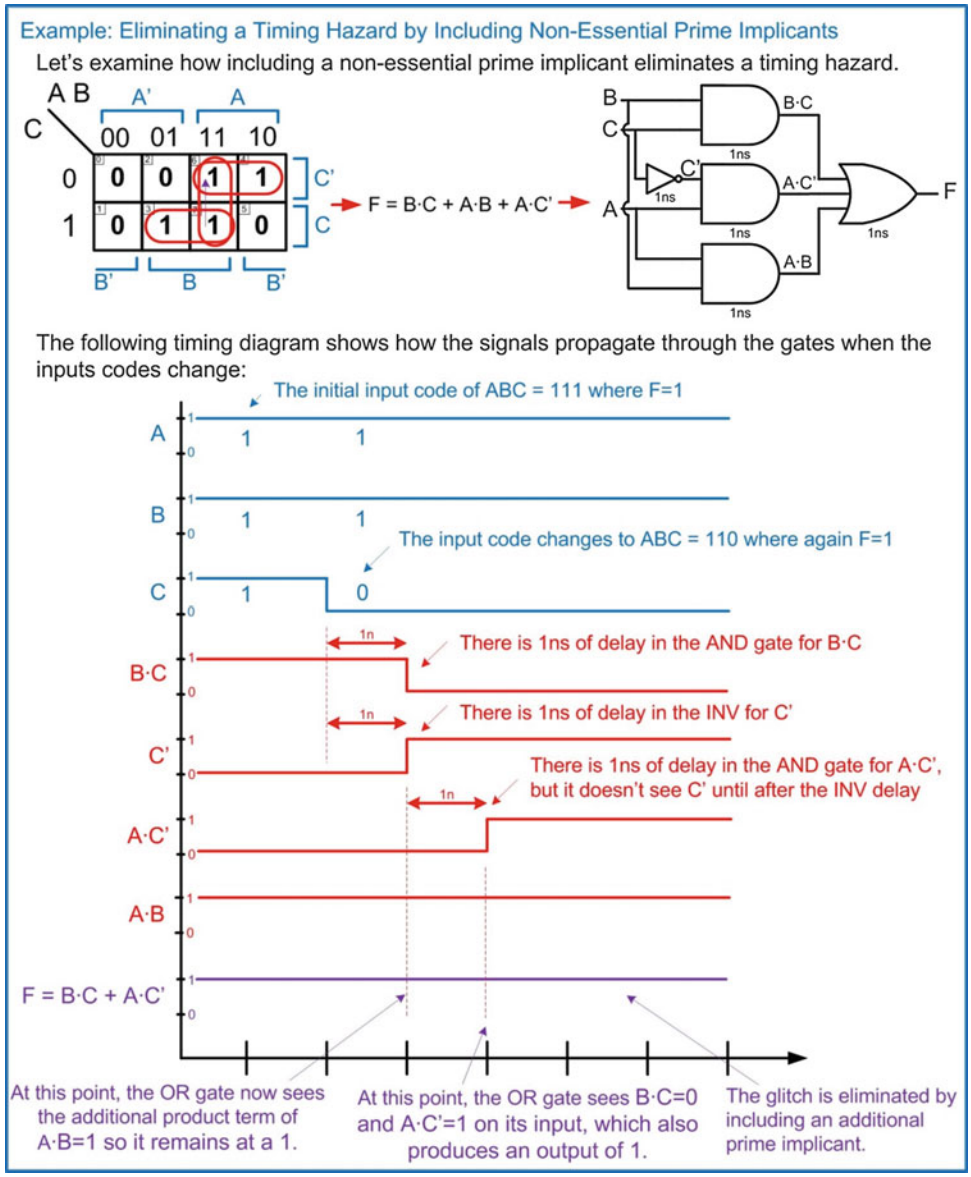


Fig. 4.26
Timing hazard definitions

Timing hazards can be addressed in a variety of ways. One way is to try to match the propagation delays through each path of the logic circuit. This can be difficult, particularly in modern logic families such as CMOS. In the example in Fig. 4.25, the root cause of the different propagation delays was due to an inverter on one of the variables. It seems obvious that this could be addressed by putting buffers on the other inputs with equal delays as the inverter. This would create a situation where all input codes would arrive at the first stage of AND gates at the same time regardless of whether they were inverted or not and eliminate the hazards; however, CMOS implements a buffer as two inverters in series, so it is difficult to insert a buffer in a circuit with an equal delay to an inverter. Addressing timing hazards in this way is possible, but it involves a time-consuming and tedious process of adjusting the transistors used to create the buffer and inverter to have equal delays.

Another technique to address timing hazards is to place additional circuitry in the system that will ensure the correct output while the input codes switch. Consider how including a nonessential prime implicant can eliminate a timing hazard in Example 4.31. In this approach, the minimal sum from Fig. 4.25 is instead replaced with the complete sum. The use of the complete sum instead of the minimal sum can be shown to eliminate both static and dynamic timing hazards. The drawback of this approach is the addition of extra circuitry in the combinational logic circuit (i.e., nonessential prime implicants).



Example 4.31
Eliminating a Timing Hazard by Including Nonessential Product Terms

CONCEPT CHECK

CC4.5 How long do you need to wait for all hazards to settle out?

A) The time equal to the delay through the non-essential prime implicants.
B) The time equal to the delay through the essential prime implicants.
C) The time equal to the shortest delay path in the circuit.
D) The time equal to the longest delay path in the circuit.

Summary

- Boolean algebra defines the axioms and theorems that guide the operations that can be performed on a two-valued number system.
- Boolean algebra theorems allow logic expressions to be manipulated to make circuit synthesis simpler. They also allow logic expressions to be minimized.
- The delay of a combinational logic circuit is always dictated by the longest delay path from the inputs to the output.
- The *canonical form* of a logic expression is one that has not been minimized.
- A *canonical sum of products* form is a logic synthesis technique based on *minterms*. A minterm is a product term that will output a one for only one unique input code. A minterm is used for each row of a truth table corresponding to an output of a one. Each of the minterms is then summed together to create the final system output.
- A *minterm list* is a shorthand way of describing the information in a truth table. The symbol "Σ" is used to denote a minterm list. Each of the input variables is added to this symbol as comma-delimited subscripts. The row number is then listed for each row corresponding to an output of a one.
- A *canonical product of sums* form is a logic synthesis technique based on *maxterms*. A maxterm is a sum term that will output a zero for only one unique input code. A maxterm is used for each row of a truth table corresponding to an output of a zero. Each of the maxterms is then multiplied together to create the final system output.
- A *maxterm list* is a shorthand way of describing the information in a truth table. The symbol "Π" is used to denote a maxterm list. Each of the input variables is added to this symbol as comma-delimited subscripts. The row number is then listed for each row corresponding to an output of a zero.
- Canonical logic expressions can be minimized through a repetitive process of factoring common variables using the *distributive* property and then eliminating remaining variables using a combination of the *complements* and *identity* theorems.
- A *Karnaugh map* (K-map) is a graphical approach to minimizing logic expressions. A K-map arranges a truth table into a grid in which the neighboring cells have input codes that differ by only one bit. This allows the impact of an input variable on a group of outputs to be quickly identified.
- A *minimized sum of products* expression can be found from a K-map by circling neighboring ones to form groups that can be produced by a single product term. Each product term (aka *prime implicant*) is then summed together to form the circuit output.
- A *minimized product of sums* expression can be found from a K-map by circling neighboring zeros to form groups that can be produced by a single sum term. Each sum term (aka *prime implicant*) is then multiplied together to form the circuit output.
- A *minimal sum* or *minimal product* is a logic expression that contains only essential prime implicants and represents the smallest number of logic operations possible to produce the desired output.
- A *don't care* (X) can be used when the output of a truth table row can be either a zero or a one without affecting the system behavior. This typically occurs when some of the input codes of a truth table will never occur. The value for the row of a truth table containing a don't care output can be chosen to give the most minimal logic expression. In a K-map, don't cares can be included to form the largest groupings in order to give the least amount of logic.
- While exclusive-OR gates are not used in Boolean algebra, they can be visually identified in K-maps by looking for checkerboard patterns.
- Timing hazards are temporary glitches that occur on the output of a combinational logic circuit due to timing mismatches through different paths in the circuit. Hazards can be minimized by including additional circuitry in the system or by matching the delay of all signal paths.

Exercise Problems

Section 4.1: Boolean Algebra

4.1.1 Which Boolean algebra theorem describes the situation where *any variable OR'd with itself will yield itself*?

4.1.2 Which Boolean algebra theorem describes the situation where *any variable that is double complemented will yield itself*?

4.1.3 Which Boolean algebra theorem describes the situation where *any variable OR'd with a 1 will yield a 1*?

4.1.4 Which Boolean algebra theorem describes the situation where a variable that exists in multiple product terms can be factored out?

4.1.5 Which Boolean algebra theorem describes the situation where when output(s) corresponding to a term within an expression are handled by another term the original term can be removed?

4.1.6 Which Boolean algebra theorem describes the situation where *any variable AND'd with its complement will yield a 0*?

4.1.7 Which Boolean algebra theorem describes the situation where *any variable AND'd with a 0 will yield a 0*?

4.1.8 Which Boolean algebra theorem describes the situation where an AND gate with its inputs inverted is equivalent to an OR gate with its outputs inverted?

4.1.9 Which Boolean algebra theorem describes the situation where *a variable that exists in multiple sum terms can be factored out*?

4.1.10 Which Boolean algebra theorem describes the situation where an OR gate with its inputs inverted is equivalent to an AND gate with its outputs inverted?

4.1.11 Which Boolean algebra theorem describes the situation where the grouping of variables in an OR operation does not affect the result?

4.1.12 Which Boolean algebra theorem describes the situation where *any variable AND'd with itself will yield itself*?

4.1.13 Which Boolean algebra theorem describes the situation where the order of variables in an OR operation does not affect the result?

4.1.14 Which Boolean algebra theorem describes the situation where *any variable AND'd with a 1 will yield itself*?

4.1.15 Which Boolean algebra theorem describes the situation where the grouping of variables in an AND operation does not affect the result?

4.1.16 Which Boolean algebra theorem describes the situation where *any variable OR'd with its complement will yield a 1*?

4.1.17 Which Boolean algebra theorem describes the situation where the order of variables in an AND operation does not affect the result?

4.1.18 Which Boolean algebra theorem describes the situation where *a variable OR'd with a 0 will yield itself*?

4.1.19 Use proof by exhaustion to prove that an OR gate with its inputs inverted is equivalent to an AND gate with its outputs inverted.

4.1.20 Use proof by exhaustion to prove that an AND gate with its inputs inverted is equivalent to an OR gate with its outputs inverted.

Section 4.2: Combinational Logic Analysis

4.2.1 For the logic diagram given in Fig. 4.27, give the logic expression for the output F.

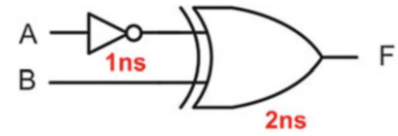


Fig. 4.27
Combinational Logic Analysis 1

4.2.2 For the logic diagram given in Fig. 4.27, give the truth table for the output F.

4.2.3 For the logic diagram given in Figure 4.27, give the delay.

4.2.4 For the logic diagram given in Fig. 4.28, give the logic expression for the output F.

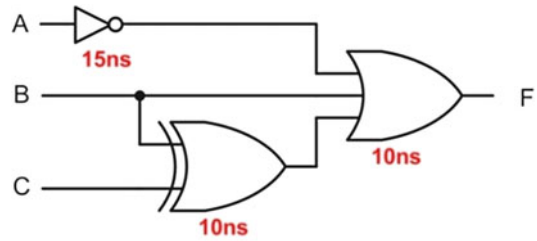


Fig. 4.28
Combinational Logic Analysis 2

4.2.5 For the logic diagram given in Fig. 4.28, give the truth table for the output F.

4.2.6 For the logic diagram given in Fig. 4.28, give the delay.

4.2.7 For the logic diagram given in Fig. 4.29, give the logic expression for the output F.

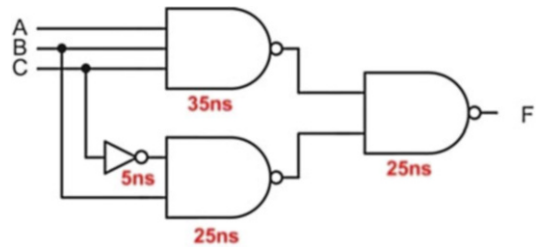


Fig. 4.29
Combinational Logic Analysis 3

4.2.8 For the logic diagram given in Fig. 4.29, give the truth table for the output F.

4.2.9 For the logic diagram given in Fig. 4.29, give the delay.

Section 4.3: Combinational Logic Synthesis

4.3.1 For the 2-input truth table in Fig. 4.30, give the canonical sum of products (SOP) logic expression.

A	B	F
0	0	0
0	1	1
1	0	0
1	1	1

Fig. 4.30
Combinational Logic Synthesis 1

4.3.2 For the 2-input truth table in Fig. 4.30, give the canonical sum of products (SOP) logic diagram.

4.3.3 For the 2-input truth table in Fig. 4.30, give the minterm list.

4.3.4 For the 2-input truth table in Fig. 4.30, give the canonical product of sums (POS) logic expression.

4.3.5 For the 2-input truth table in Fig. 4.30, give the canonical product of sums (POS) logic diagram.

4.3.6 For the 2-input truth table in Fig. 4.30, give the maxterm list.

4.3.7 For the 2-input minterm list in Fig. 4.31, give the canonical sum of products (SOP) logic expression.

$$F = \sum_{A,B}(1,2,3)$$

Fig. 4.31
Combinational Logic Synthesis 2

4.3.8 For the 2-input minterm list in Fig. 4.31, give the canonical sum of products (SOP) logic diagram.

4.3.9 For the 2-input minterm list in Fig. 4.31, give the truth table.

4.3.10 For the 2-input minterm list in Fig. 4.31, give the canonical product of sums (POS) logic expression.

4.3.11 For the 2-input minterm list in Fig. 4.31, give the canonical product of sums (POS) logic diagram.

4.3.12 For the 2-input minterm list in Fig. 4.31, give the maxterm list.

4.3.13 For the 2-input maxterm list in Fig. 4.32, give the canonical sum of products (SOP) logic expression.

$$F = \prod_{A,B}(1,2,3)$$

Fig. 4.32
Combinational Logic Synthesis 3

4.3.14 For the 2-input maxterm list in Fig. 4.32, give the canonical sum of products (SOP) logic diagram.

4.3.15 For the 2-input maxterm list in Fig. 4.32, give the minterm list.

4.3.16 For the 2-input maxterm list in Fig. 4.32, give the canonical product of sums (POS) logic expression.

4.3.17 For the 2-input maxterm list in Fig. 4.32, give the canonical product of sums (POS) logic diagram.

4.3.18 For the 2-input maxterm list in Fig. 4.32, give the truth table.

4.3.19 For the 3-input truth table in Fig. 4.33, give the canonical sum of products (SOP) logic expression.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Fig. 4.33
Combinational Logic Synthesis 4

4.3.20 For the 3-input truth table in Fig. 4.33, give the canonical sum of products (SOP) logic diagram.

4.3.21 For the 3-input truth table in Fig. 4.33, give the minterm list.

4.3.22 For the 3-input truth table in Fig. 4.33, give the canonical product of sums (POS) logic expression.

4.3.23 For the 3-input truth table in Fig. 4.33, give the canonical product of sums (POS) logic diagram.

4.3.24 For the 3-input truth table in Fig. 4.33, give the maxterm list.

4.3.25 For the 3-input minterm list in Fig. 4.34, give the canonical sum of products (SOP) logic expression.

$$F = \sum_{A,B,C}(2,4,6)$$

Fig. 4.34
Combinational Logic Synthesis 5

4.3.26 For the 3-input minterm list in Fig. 4.34, give the canonical sum of products (SOP) logic diagram.

4.3.27 For the 3-input minterm list in Fig. 4.34, give the truth table.

4.3.28 For the 3-input minterm list in Fig. 4.34, give the canonical product of sums (POS) logic expression.

4.3.29 For the 3-input minterm list in Fig. 4.34, give the canonical product of sums (POS) logic diagram.

4.3.30 For the 3-input minterm list in Fig. 4.34, give the maxterm list.

4.3.31 For the 3-input maxterm list in Fig. 4.35, give the canonical sum of products (SOP) logic expression.

$$F = \prod_{A,B,C}(2,3,5,6,7)$$

Fig. 4.35
Combinational Logic Synthesis 6

4.3.32 For the 3-input maxterm list in Fig. 4.35, give the canonical sum of products (SOP) logic diagram.

4.3.33 For the 3-input maxterm list in Fig. 4.35, give the minterm list.

4.3.34 For the 3-input maxterm list in Fig. 4.35, give the canonical product of sums (POS) logic expression.

4.3.35 For the 3-input maxterm list in Fig. 4.35, give the canonical product of sums (POS) logic diagram.

4.3.36 For the 3-input maxterm list in Fig. 4.35, give the truth table.

4.3.37 For the 4-input truth table in Fig. 4.36, give the canonical sum of products (SOP) logic expression.

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Fig. 4.36
Combinational Logic Synthesis 7

4.3.38 For the 4-input truth table in Fig. 4.36, give the canonical sum of products (SOP) logic diagram.

4.3.39 For the 4-input truth table in Fig. 4.36, give the minterm list.

4.3.40 For the 4-input truth table in Fig. 4.36, give the canonical product of sums (POS) logic expression.

4.3.41 For the 4-input truth table in Fig. 4.36, give the canonical product of sums (POS) logic diagram.

4.3.42 For the 4-input truth table in Fig. 4.36, give the maxterm list.

4.3.43 For the 4-input minterm list in Fig. 4.37, give the canonical sum of products (SOP) logic expression.

$$F = \sum_{A,B,C,D}(4,5,7,12,13,15)$$

Fig. 4.37
Combinational Logic Synthesis 8

4.3.44 For the 4-input minterm list in Fig. 4.37, give the canonical sum of products (SOP) logic diagram.

4.3.45 For the 4-input minterm list in Fig. 4.37, give the truth table.

4.3.46 For the 4-input minterm list in Fig. 4.37, give the canonical product of sums (POS) logic expression.

4.3.47 For the 4-input minterm list in Fig. 4.37, give the canonical product of sums (POS) logic diagram.

4.3.48 For the 4-input minterm list in Fig. 4.37, give the maxterm list.

4.3.49 For the 4-input maxterm list in Fig. 4.38, give the canonical sum of products (SOP) logic expression.

$$F = \prod_{A,B,C,D}(3,7,11,15)$$

Fig. 4.38
Combinational Logic Synthesis 9

4.3.50 For the 4-input maxterm list in Fig. 4.38, give the canonical sum of products (SOP) logic diagram.

4.3.51 For the 4-input maxterm list in Fig. 4.38, give the minterm list.

4.3.52 For the 4-input maxterm list in Fig. 4.38, give the canonical product of sums (POS) logic expression.

4.3.53 For the 4-input maxterm list in Fig. 4.38, give the canonical product of sums (POS) logic diagram.

4.3.54 For the 4-input maxterm list in Fig. 4.38, give the truth table.

Section 4.4: Logic Minimization

4.4.1 For the 2-input truth table in Fig. 4.39, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	F
0	0	0
0	1	1
1	0	0
1	1	1

Fig. 4.39
Logic Minimization 1

4.4.2 For the 2-input truth table in Fig. 4.39, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.3 For the 2-input truth table in Fig. 4.40, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 4.40
Logic Minimization 2

4.4.4 For the 2-input truth table in Fig. 4.41, use a K-map to derive a minimized product of sums (POS) logic expression.

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 4.41
Logic Minimization 3

4.4.5 For the 2-input truth table in Fig. 4.42, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	F
0	0	1
0	1	1
1	0	0
1	1	0

Fig. 4.42
Logic Minimization 4

4.4.6 For the 2-input truth table in Fig. 4.42, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.7 For the 3-input truth table in Fig. 4.43, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Fig. 4.43
Logic Minimization 5

4.4.8 For the 3-input truth table in Fig. 4.43, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.9 For the 3-input truth table in Fig. 4.44, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Fig. 4.44
Logic Minimization 6

4.4.10 For the 3-input truth table in Fig. 4.44, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.11 For the 3-input truth table in Fig. 4.45, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Fig. 4.45
Logic Minimization 7

4.4.12 For the 3-input truth table in Fig. 4.45, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.13 For the 3-input truth table in Fig. 4.46, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Fig. 4.46
Logic Minimization 8

4.4.14 For the 3-input truth table in Fig. 4.46, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.15 For the 4-input truth table in Fig. 4.47, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Fig. 4.47
Logic Minimization 9

4.4.16 For the 4-input truth table in Fig. 4.47, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.17 For the 4-input truth table in Fig. 4.48, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	D	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	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Fig. 4.48
Logic Minimization 10

4.4.18 For the 4-input truth table in Fig. 4.48, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.19 For the 4-input truth table in Fig. 4.49, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Fig. 4.49
Logic Minimization 11

4.4.20 For the 4-input truth table in Fig. 4.49, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.21 For the 4-input truth table in Fig. 4.50, use a K-map to derive a minimized sum of products (SOP) logic expression.

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Fig. 4.50
Logic Minimization 12

4.4.22 For the 4-input truth table in Fig. 4.50, use a K-map to derive a minimized product of sums (POS) logic expression.

4.4.23 For the 3-input truth table and K-map in Fig. 4.51, provide the row number(s) of any distinguished one-cells.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

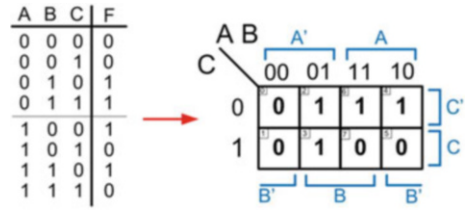


Fig. 4.51
Logic Minimization 13

4.4.24 For the 3-input truth table and K-map in Fig. 4.51, give the product terms for the essential prime implicants.

4.4.25 For the 3-input truth table and K-map in Fig. 4.51, give the minimal sum of products logic expression.

4.4.26 For the 3-input truth table and K-map in Fig. 4.51, give the complete sum of products logic expression.

4.4.27 For the 4-input truth table and K-map in Fig. 4.52, provide the row number(s) of any distinguished one-cells.

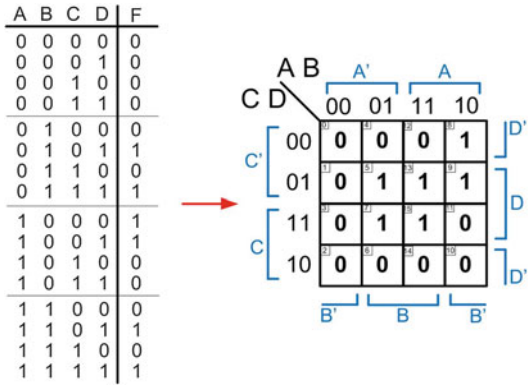


Fig. 4.52
Logic Minimization 14

4.4.28 For the 4-input truth table and K-map in Fig. 4.52, give the product terms for the essential prime implicants.

4.4.29 For the 4-input truth table and K-map in Fig. 4.52, give the minimal sum of products (SOP) logic expression.

4.4.30 For the 4-input truth table and K-map in Fig. 4.52, give the complete sum of products (SOP) logic expression.

4.4.31 For the 4-input truth table and K-map in Fig. 4.53, give the minimal sum of products (SOP) logic expression by exploiting "don't cares."

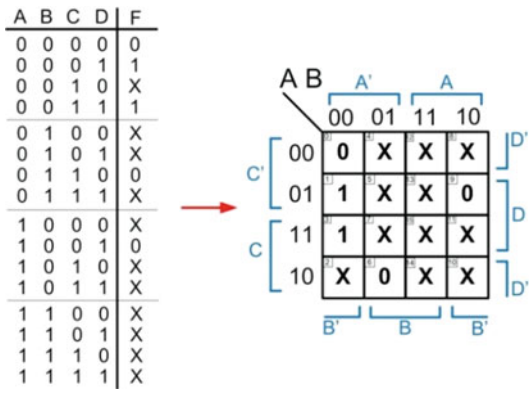


Fig. 4.53
Logic Minimization 15

4.4.32 For the 4-input truth table and K-map in Fig. 4.53, give the minimal product of sums (POS) logic expression by exploiting "don't cares."

4.4.33 For the 4-input truth table and K-map in Fig. 4.54, give the minimal product of sums (POS) logic expression by exploiting "don't cares."

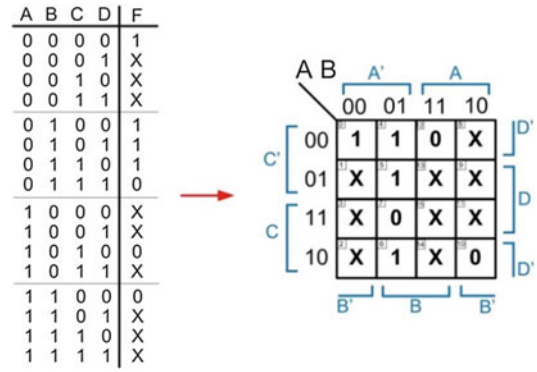


Fig. 4.54
Logic Minimization 16

4.4.34 For the 4-input truth table and K-map in Fig. 4.54, give the minimal product of sums (POS) logic expression by exploiting "don't cares".

Section 4.5: Timing Hazards and Glitches

4.5.1 Describe the situation in which a static-1 timing hazard may occur.

4.5.2 Describe the situation in which a static-0 timing hazard may occur.

4.5.3 In which topology will a static-1 timing hazard occur (SOP, POS, or both)?

4.5.4 In which topology will a static-0 timing hazard occur (SOP, POS, or both)?

4.5.5 For the 3-input truth table and K-map in Fig. 4.51, give the product term that helps eliminate static-1 timing hazards in this circuit.

4.5.6 For the 3-input truth table and K-map in Fig. 4.51, give the sum term that helps eliminate static-0 timing hazards in this circuit.

4.5.7 For the 4-input truth table and K-map in Fig. 4.52, give the product term that helps eliminate static-1 timing hazards in this circuit.

4.5.8 For the 4-input truth table and K-map in Fig. 4.52, give the sum term that helps eliminate static-0 timing hazards in this circuit.