CHAPTER-3

Gate-Level Minimization

Digital Design (with an introduction to the Verilog HDL) 6th Edition,
M. Morris Mano, Michael D. Ciletti

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

- Dr. Kalpana Settu
  Assistant Professor
  ECE, IIITDM Kancheepuram

Outline

- The Map Method
- Four Variable K-map
- Product of Sums
- Don’t Care Conditions
- NAND & NOR Implementation
- And-Or-Invert & Or-And-Invert
- Exclusive-OR Function
- Hardware Description Language

Why Logic Minimization?

- *Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.
- Minimize the number of gates used

  → Reduce gate count = reduce cost
- Minimize total delay (critical path delay)

  → Reduce delay = improve performance

The Map Method

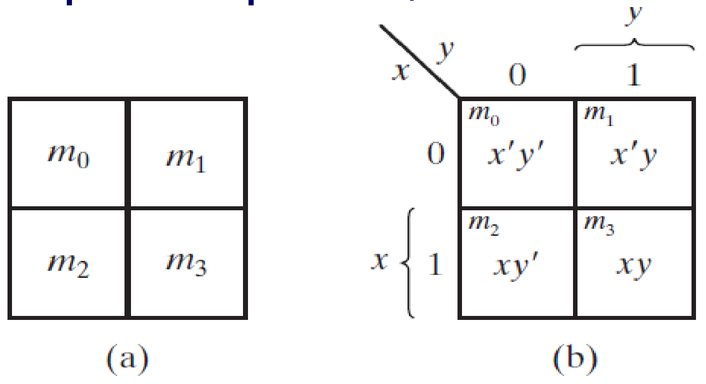
- The truth table representation of a function is unique, but the function can be expressed in many different algebraic forms.
- The complexity of the digital logic gates is directly related to the complexity of the algebraic expression of a function.
- Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions.
- The map method provides a simple, straightforward procedure for minimizing Boolean functions.

The Map Method (Karnaugh map or K-map)

- The map method is also known as the Karnaugh map or K-map.
- Provide a straightforward procedure for minimizing Boolean functions.

Maurice Karnaugh introduced it in 1953 as a refinement of Edward Veitch's 1952 Veitch diagram.

- A Karnaugh map (K-map) is a diagram made up of squares, with each square representing one minterm of the function.
- Example: Two-variable K-map (4 minterms)



- The simplified expressions are always in one of the two standard forms:
  - Sum of Products (SOP)
  - Product of Sums (POS)

Two-Variable K-Map

- Two-variable function has four minterms
  - Four squares in the map for those minterms
- The corresponding minterm of each square is determined by the bit status shown outside

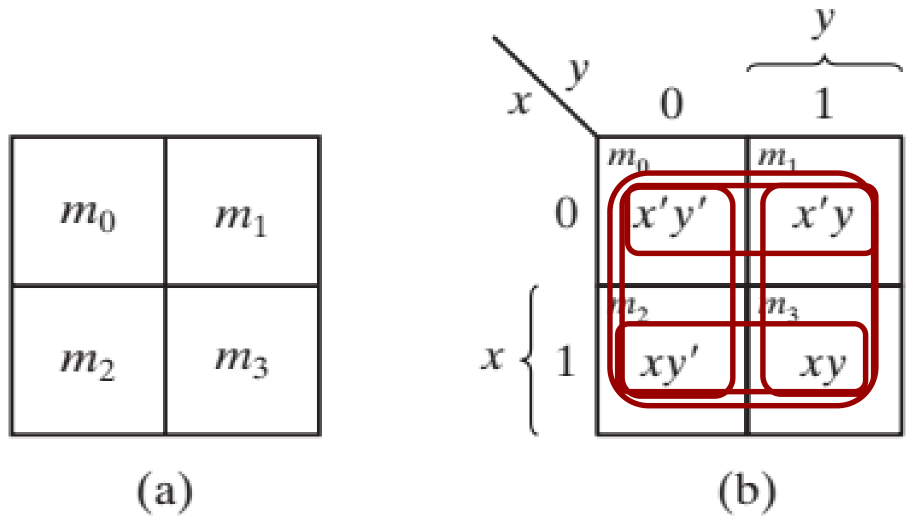


FIGURE 3.1
Two-variable K-map

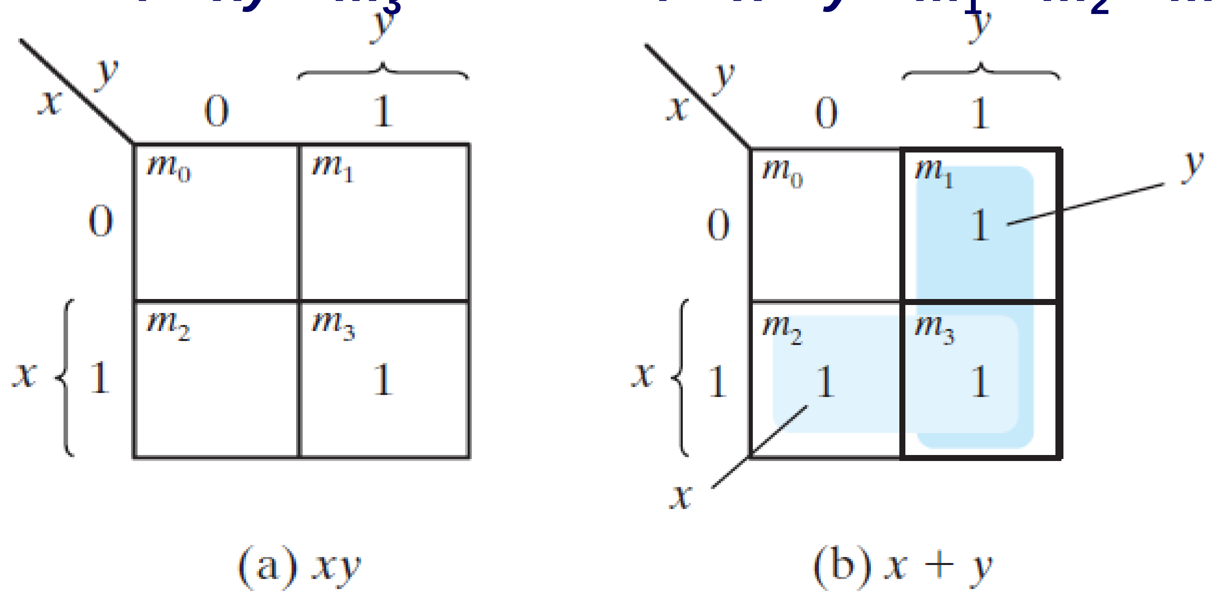
- In **K-map**, the **adjacent squares** represent minterms that differ by one variable
- Minterms in **adjacent squares can** be **combined**

- 1 square = 1 minterm = two literals
- 2 adjacent squares = 1 term = one literal
- 4 adjacent squares = 1

Two-Variable K-Map

- In the K-map, mark the squares whose minterms belong to a given function.
- Example: (a) *xy* (b) *x* + *y*

$f = xy = m_3$ $\qquad$ $f = x + y = m_1 + m_2 + m_3$



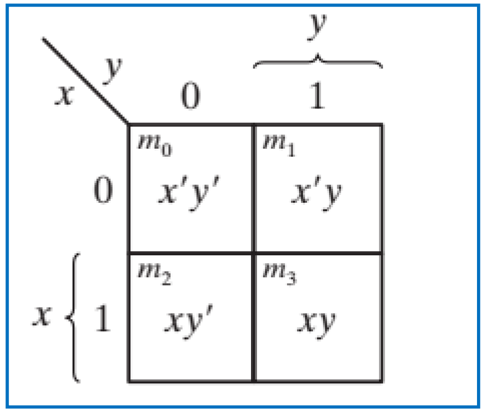
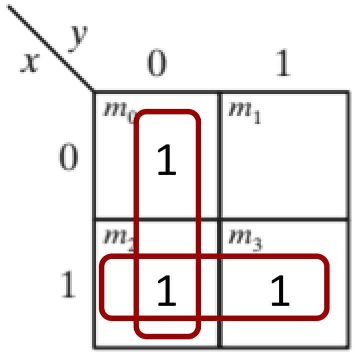


FIGURE 3.2
Representation of functions in the map

Two-Variable K-Map

- Simplify the Boolean function F(x,y) = $\sum(0,2,3)$



F(x,y) = x + y’

Three-Variable K-Map

- **In the three variable K-map, there are eight minterms for three binary variables.**
- **Watch the sequence!! Only one bit changes in value from one adjacent column to the next.**
- The minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code.

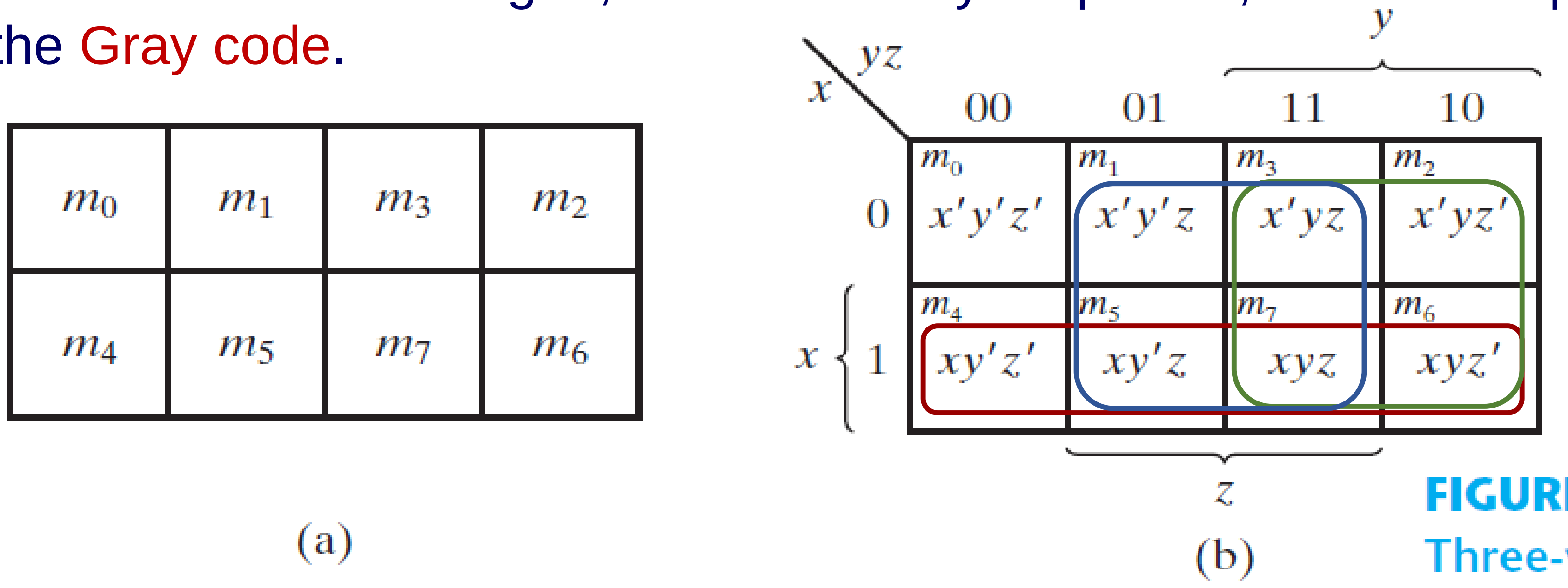
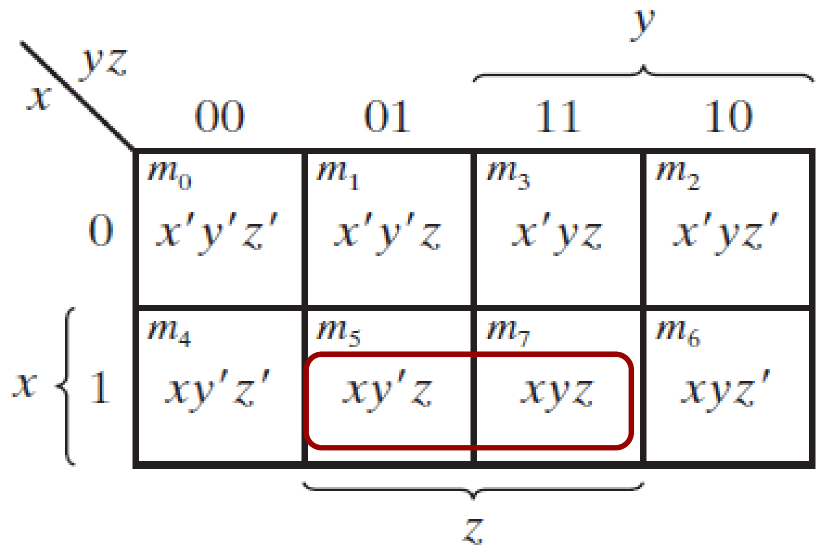


FIGURE 3.3
Three-variable K-map

Three-Variable K-Map

- **To simplify the final expression!**
  - → Recognize: Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other.
  - → **The sum of two minterms in adjacent squares can be simplified to a single product term.**



The two squares differ by the variable *y*, which can be removed when the sum of the two minterms are formed.

$$m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

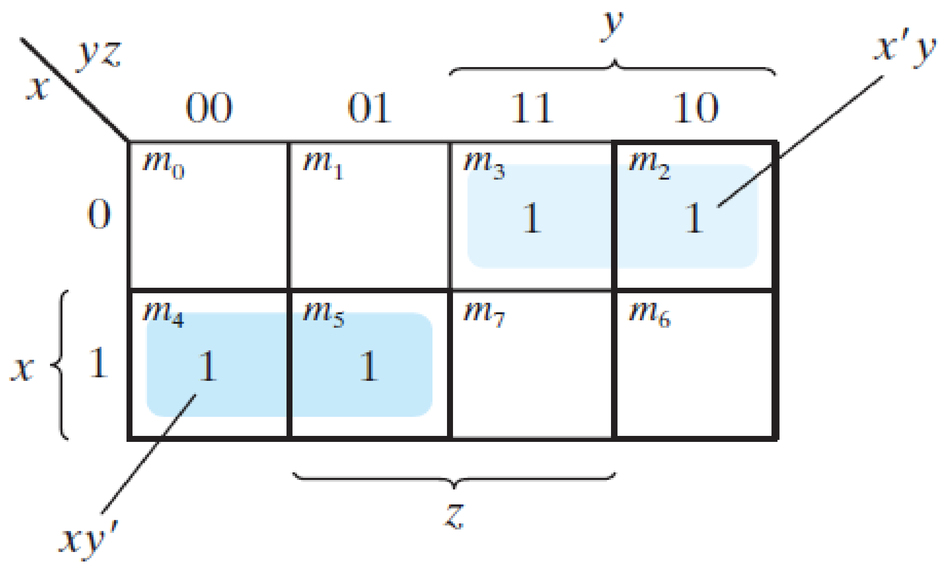
Three Variable K-Map

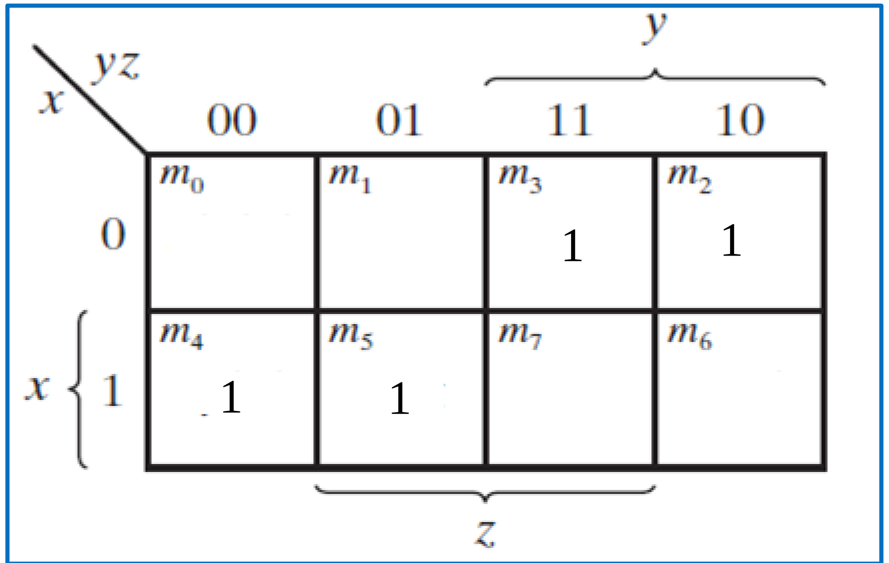
- A larger number of adjacent squares are combined, we obtain a product term with fewer literals.
  - 1 square = 1 minterm = three literals.
  - 2 adjacent squares = 1 term = two literals.
  - 4 adjacent squares = 1 term = one literal.
  - 8 adjacent squares encompass the entire map and produce a function that is always equal to 1.
- It is obviously to know the number of adjacent squares is combined in a power of two such as 1,2,4, and 8.

Example 3.1 of K-map

EXAMPLE 3.1

Simplify the Boolean function $F(x, y, z) = \Sigma(2, 3, 4, 5)$

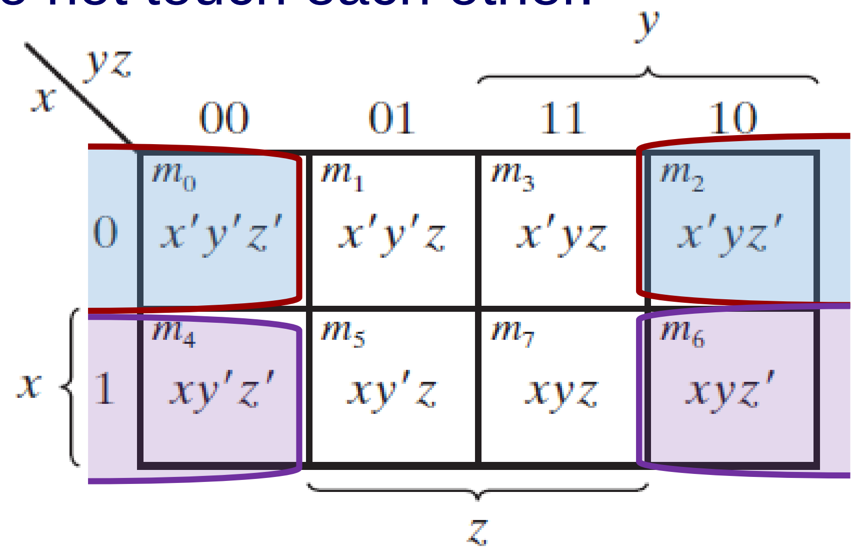




$$\boxed{F = x'y + xy'}$$

Three Variable K-Map

- In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other.



- m_0 is adjacent to m_2 because their minterms differ by one variable.

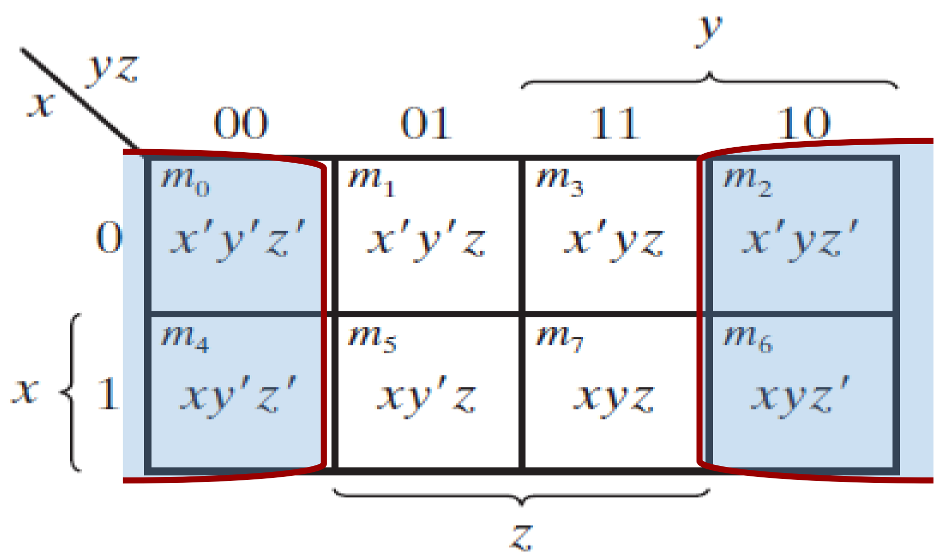
$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$

- m_4 is adjacent to m_6

$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

Three Variable K-Map

- Four adjacent squares in the three-variable map → represents the logical sum of four minterms → results in an expression with only one literal.

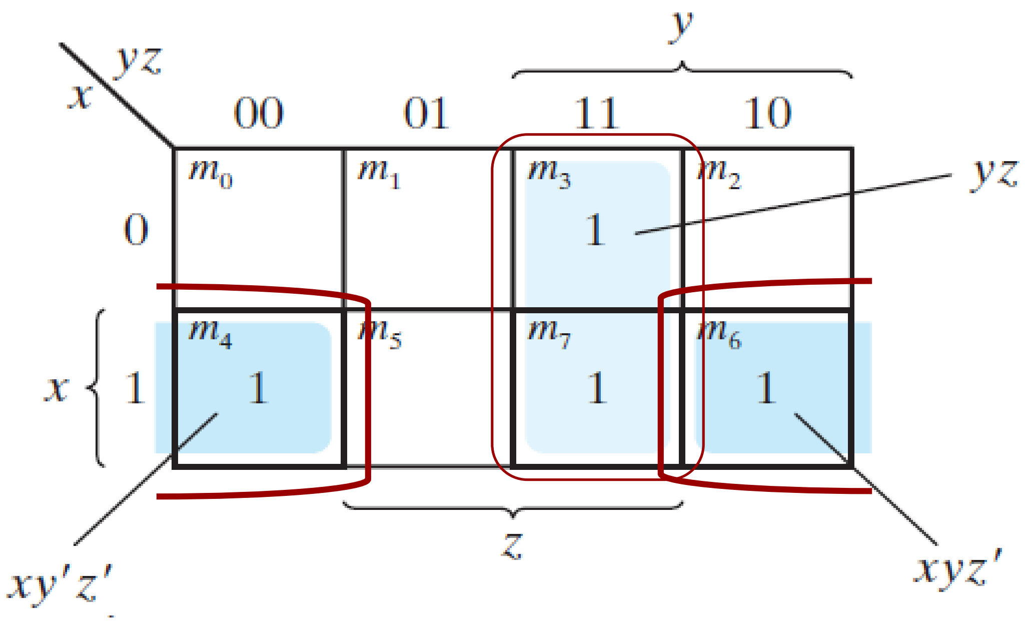


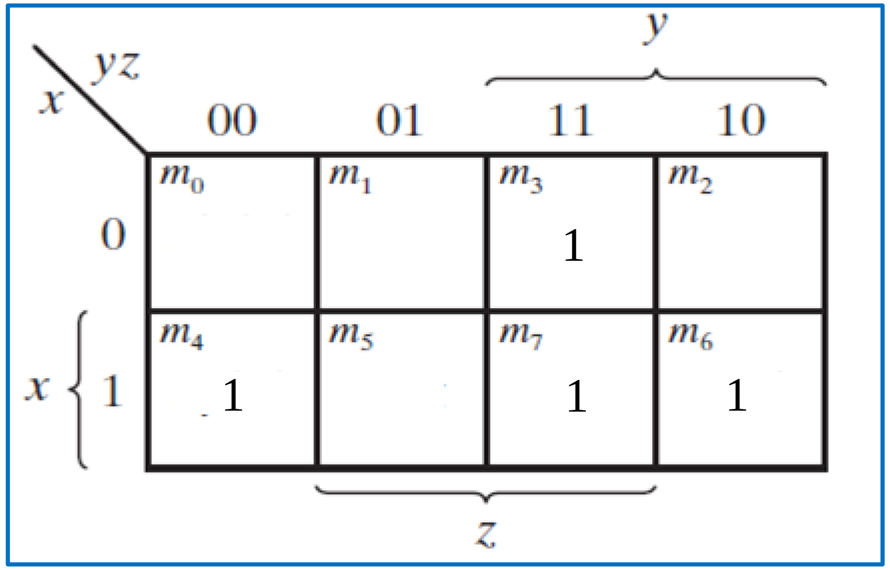
$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\ &= x'z'(y' + y) + xz'(y' + y) \\ &= x'z' + xz' = z'(x' + x) = z' \end{aligned}$$

Example 3.2 of K-map

EXAMPLE 3.2

Simplify the Boolean function $F(x, y, z) = \Sigma(3, 4, 6, 7)$





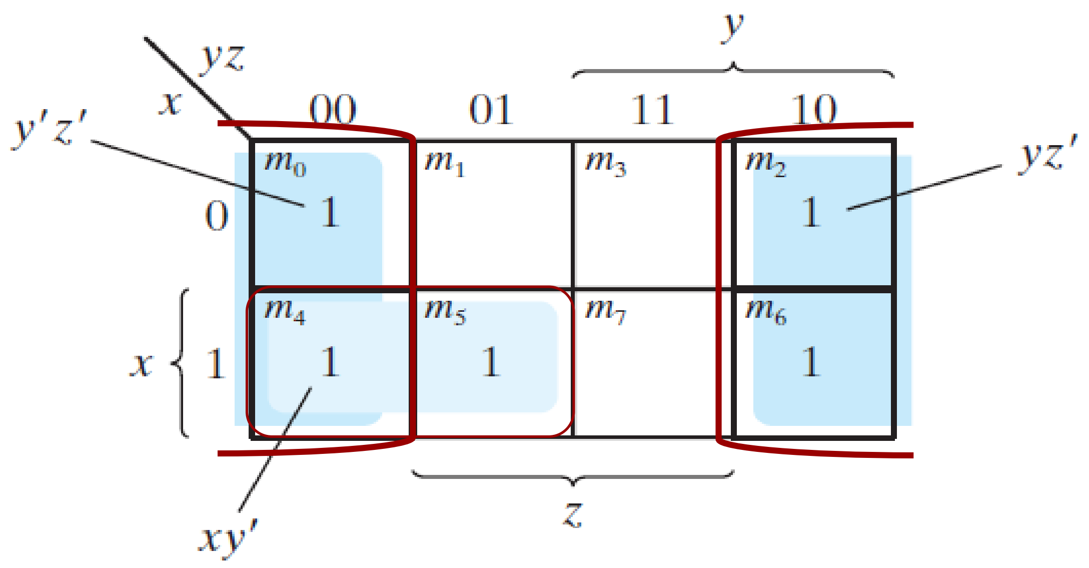
Note: $xy'z' + xyz' = xz'$

$$\boxed{F = yz + xz'}$$

Example 3.3 of K-map

EXAMPLE 3.3

Simplify the Boolean function $F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$



Note: $y'z' + yz' = z'$

$$\boxed{F = z' + xy'}$$

Example 3.4 of K-map

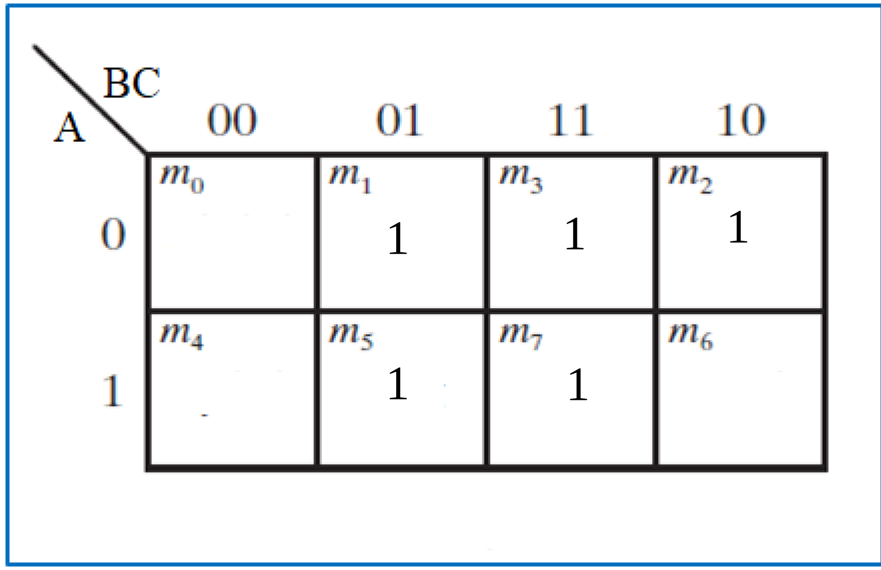
- If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms.

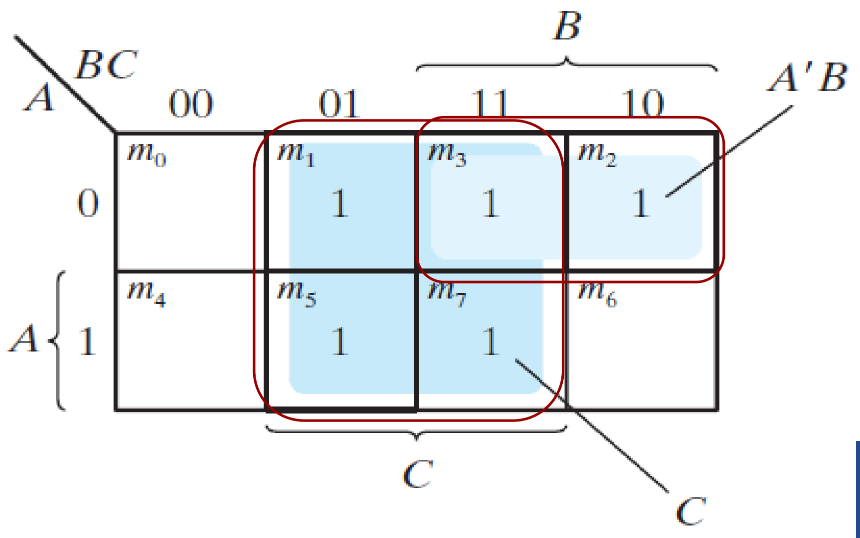
EXAMPLE 3.4

For the Boolean function

$$F = A'C + A'B + AB'C + BC$$

(a) Express this function as a sum of minterms.
(b) Find the minimal sum-of-products expression.





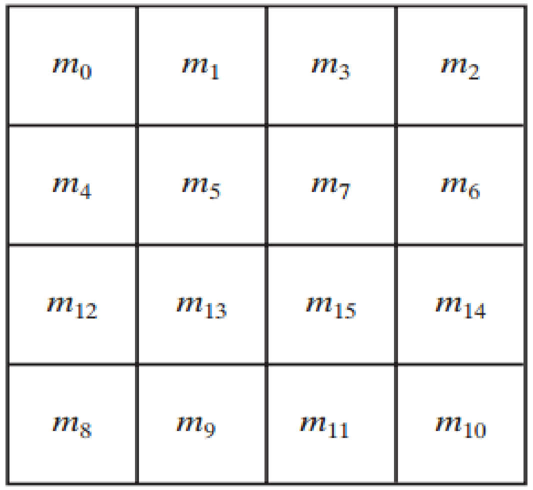
$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

$$F = C + A'B$$

Four Variable K-map

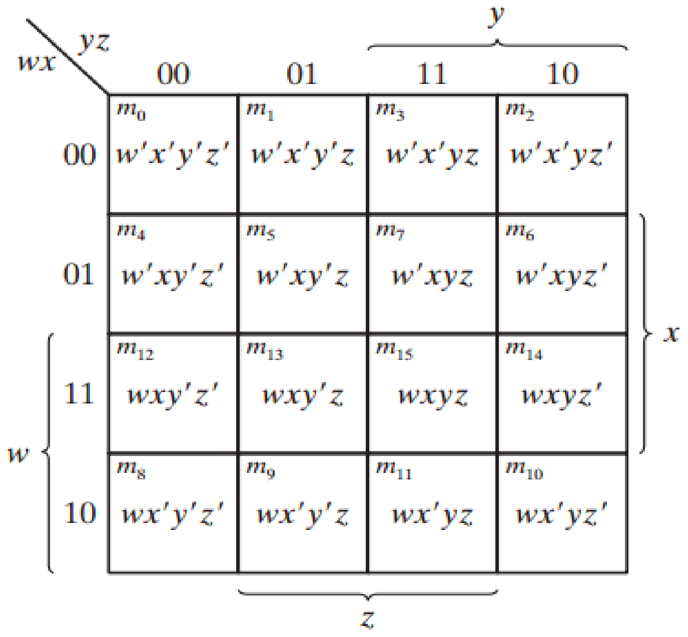
- The same rule: only one bit changes in value from one column to the next.

4 variables: *w*, *x*, *y*, *z*



m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

(a)



$wx \backslash yz$	00	01	11	10
00	m_0 $w'x'y'z'$	m_1 $w'x'y'z$	m_3 $w'x'yz$	m_2 $w'x'yz'$
01	m_4 $w'xy'z'$	m_5 $w'xy'z$	m_7 $w'xyz$	m_6 $w'xyz'$
11	m_{12} $wxy'z'$	m_{13} $wxy'z$	m_{15} $wxyz$	m_{14} $wxyz'$
10	m_8 $wx'y'z'$	m_9 $wx'y'z$	m_{11} $wx'yz$	m_{10} $wx'yz'$

(b)

1 square = 1 minterm = 4 literals
2 adjacent squares = 1 term = 3 literals
4 adjacent squares = 1 term = 2 literals
8 adjacent squares = 1 term = 1 literal
16 adjacent squares = 1

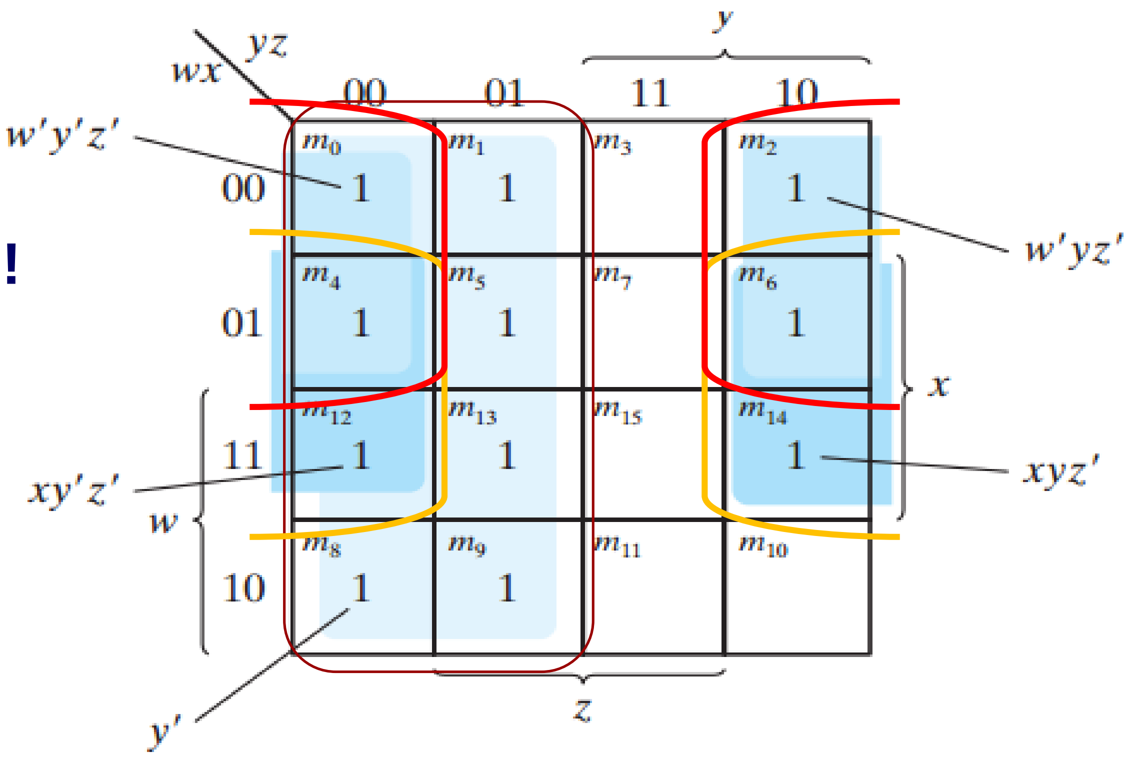
Example 3.5 of K-map

EXAMPLE 3.5

Simplify the Boolean function $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$

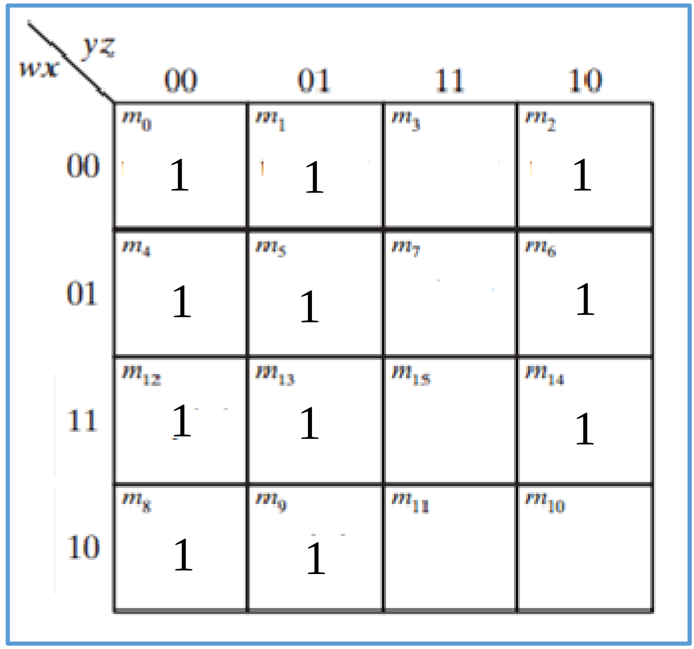
- **We can use the same square more than once!**

$$F = \boxed{y'} + \boxed{w'z'} + \boxed{xz'}$$



$$\begin{array}{c|cccc}
wx \backslash yz & 00 & 01 & 11 & 10 \\ \hline
00 & m_0\ 1 & m_1\ 1 & m_3 & m_2\ 1 \\
01 & m_4\ 1 & m_5\ 1 & m_7 & m_6\ 1 \\
11 & m_{12}\ 1 & m_{13}\ 1 & m_{15} & m_{14}\ 1 \\
10 & m_8\ 1 & m_9\ 1 & m_{11} & m_{10}
\end{array}$$

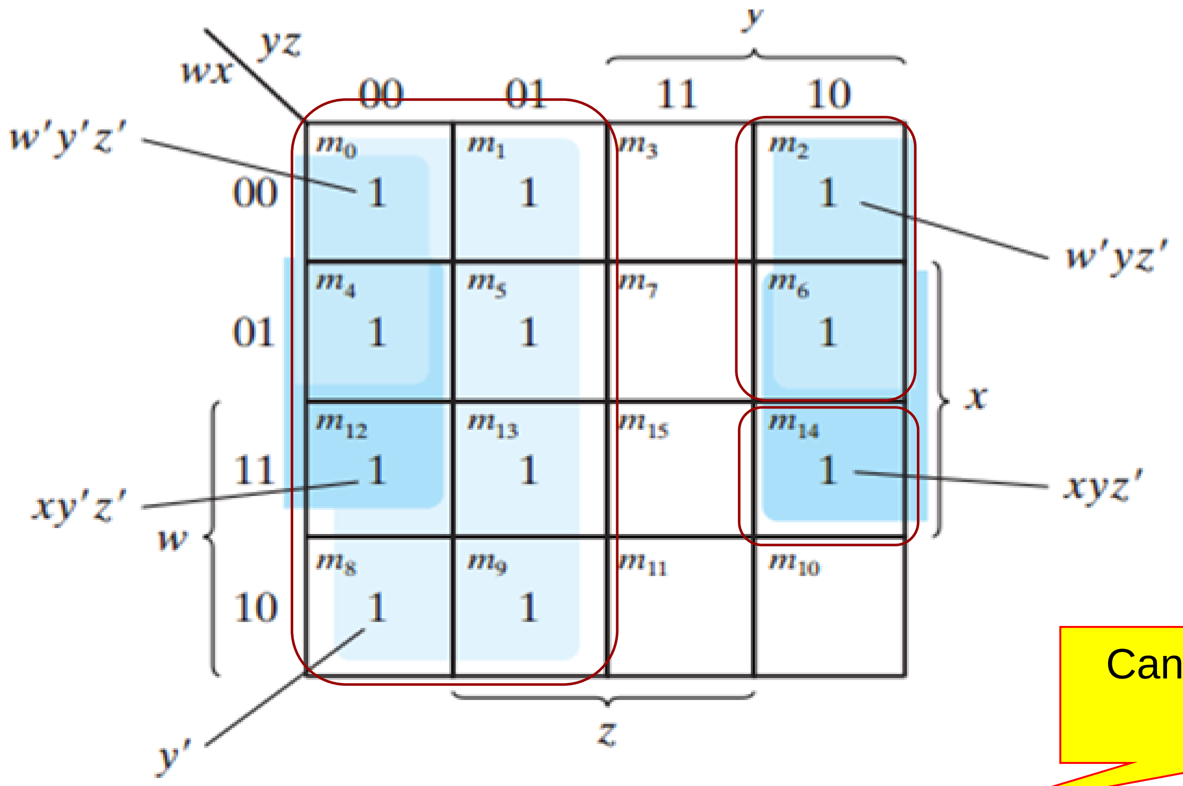
Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$



$$\begin{array}{c|cccc}
wx \backslash yz & 00 & 01 & 11 & 10 \\ \hline
00 & m_0\ 1 & m_1\ 1 & m_3 & m_2\ 1 \\
01 & m_4\ 1 & m_5\ 1 & m_7 & m_6\ 1 \\
11 & m_{12}\ 1 & m_{13}\ 1 & m_{15} & m_{14}\ 1 \\
10 & m_8\ 1 & m_9\ 1 & m_{11} & m_{10}
\end{array}$$

Example 3.5 of K-map

- Using a fewer number of squares isn't a good idea.



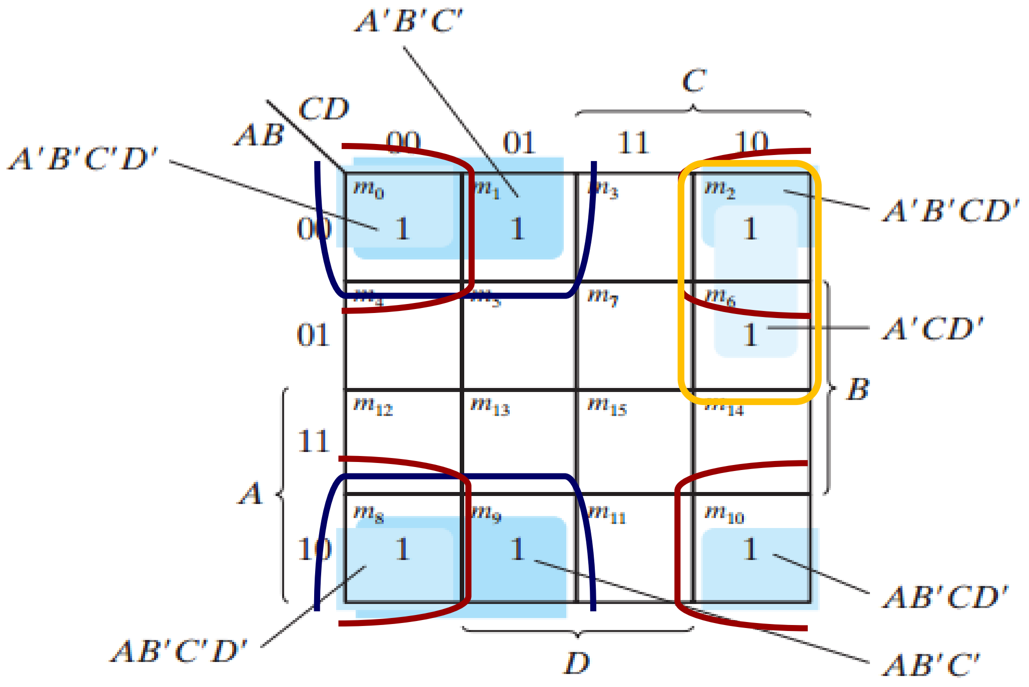
$F = y' + w'yz' + wxyz'$

Can be Simplified further!

Example 3.6 of K-map

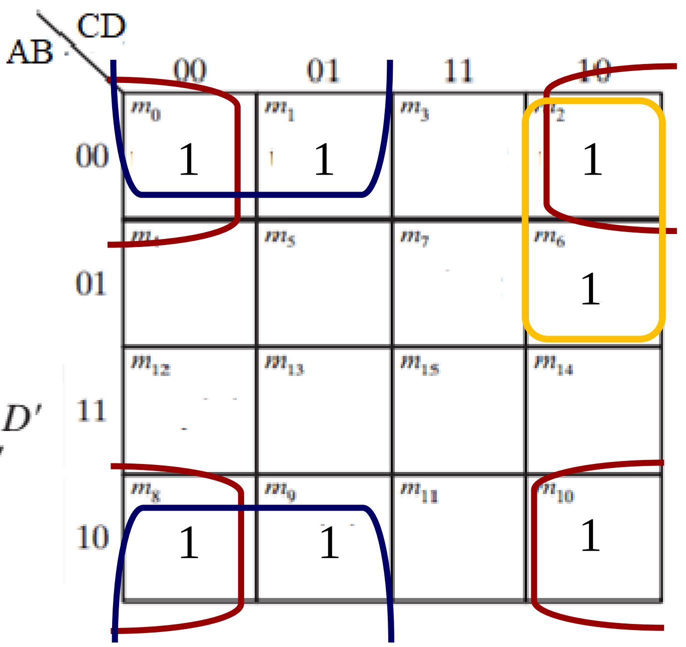
EXAMPLE 3.6

Simplify the Boolean function $F = A'B'C' + B'CD' + A'BCD' + AB'C'$



$$F = \boxed{B'D'} + \boxed{B'C'} + \boxed{A'CD'}$$

Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$



$$F = B'D' + B'C' + A'CD'$$

Prime Implicants

- In choosing adjacent squares in a map, we must ensure that

  (1) all the minterms of the function are covered when we combine the squares,

  (2) the number of terms in the expression is minimized, and

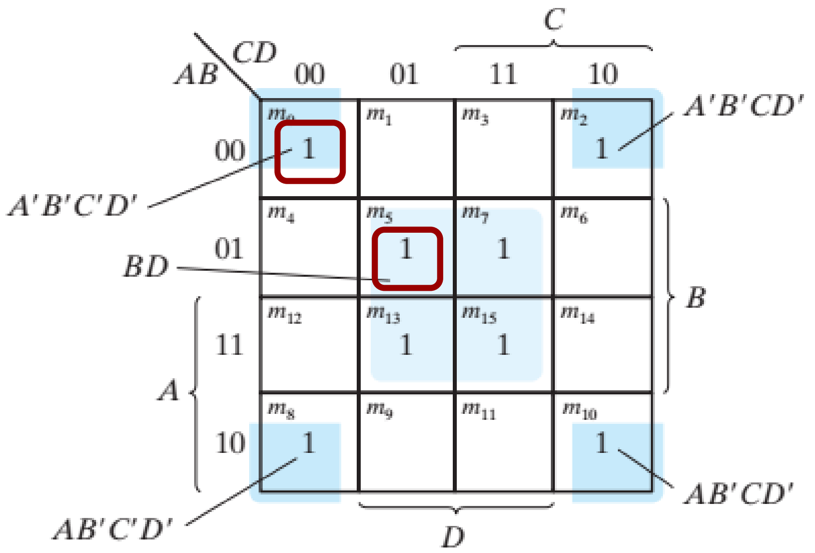
  (3) there are no redundant terms (i.e., minterms already covered by other terms).

- A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

Prime Implicants

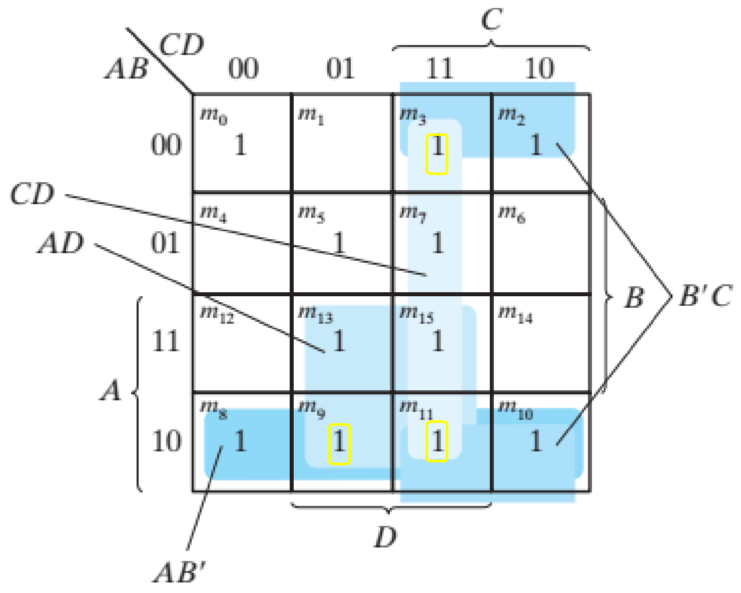
- A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

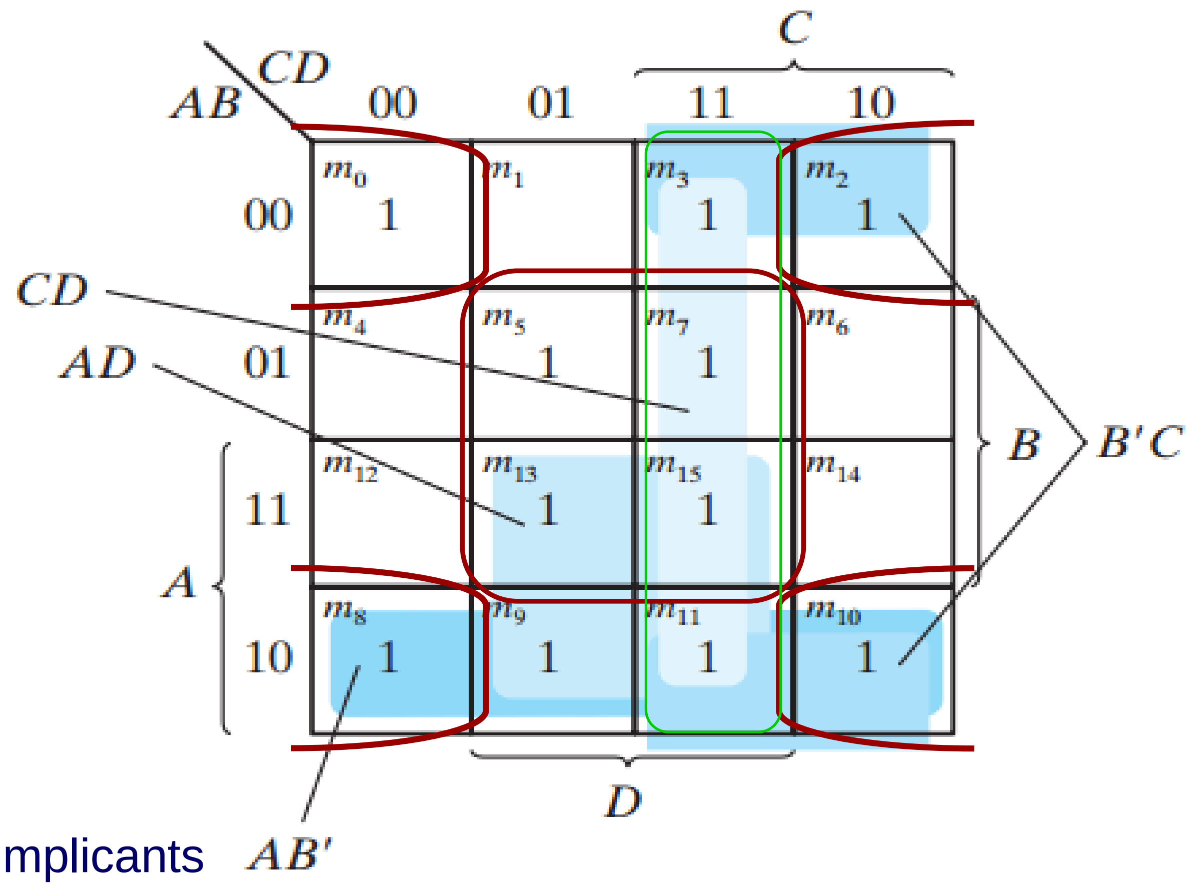
(a) Essential prime implicants BD and $B'D'$



(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE 3.11
Simplification using prime implicants

Prime Implicants

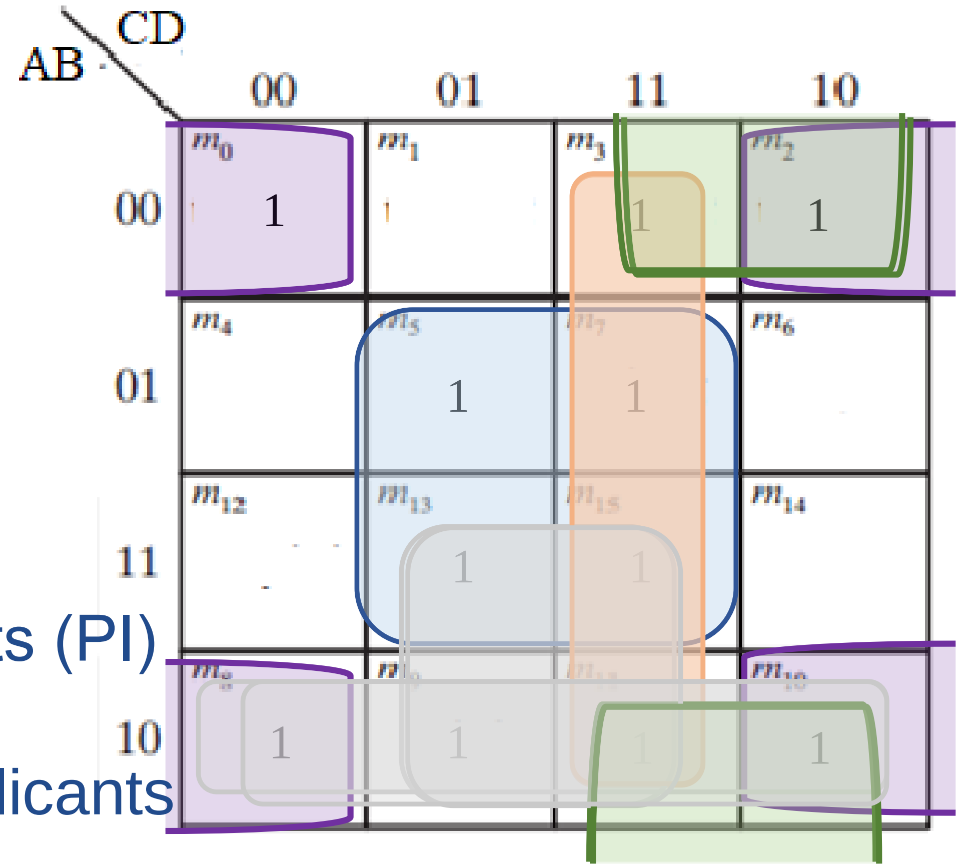


$$
\begin{aligned}
F &= BD + B'D' + CD + AD \\
&= BD + B'D' + CD + AB' \\
&= BD + B'D' + B'C + AD \\
&= BD + B'D' + B'C + AB'
\end{aligned}
$$

- For finding the simplified expression
  - first determine all the essential prime implicants
  - logical sum of all the essential prime implicants, plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants.

Prime Implicants

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$



$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

BD, B'D', CD, AD, AB', B'C : Prime Implicants (PI)

BD, B'D' : Essential Prime Implicants (EPI)

CD, AD, AB', B'C : Non-Essential Prime Implicants

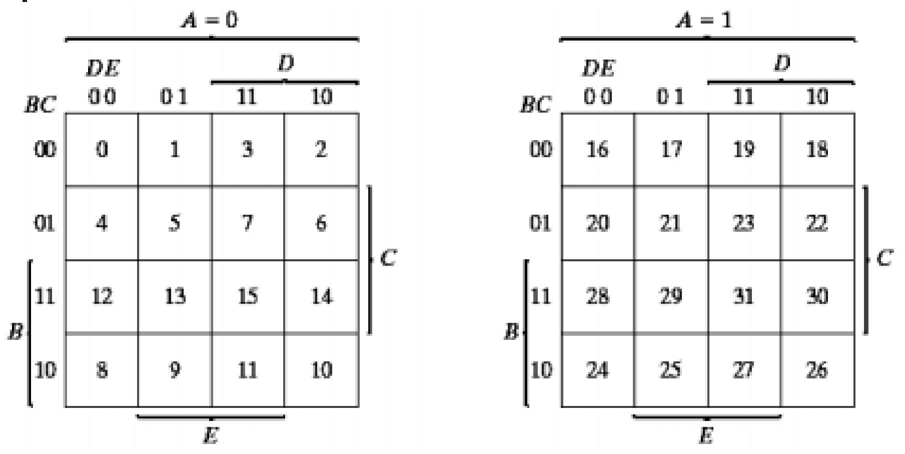
or

Selective Prime Implicant (SPI)

Five-Variable K-Map

- Maps for more than four variables are not as simple to use as maps for four or fewer variables.
- A five-variable map needs 32 squares and a six-variable map needs 64 squares.
- Maps with four or more variables need too many squares and are impractical to use.

Maps with six or more variables need too many squares and are impractical to use.



$A = 0$

BC \ DE	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

$A = 1$

BC \ DE	00	01	11	10
00	16	17	19	18
01	20	21	23	22
11	28	29	31	30
10	24	25	27	26

Product of Sums

- K-map is mainly designed for expressing a function in sum-of-products. We need to take a few extra steps for expressing a function in product-of-sums.

Steps:

1. Express the function *F* in minterms. Fill them in K-map with "1". Fill the rest with "0".
2. Use the same way to combine the squares marked "0".
3. The obtained result will be *F*'. Apply the DeMorgan's theorem and we can obtain *F* in product-of-sums.

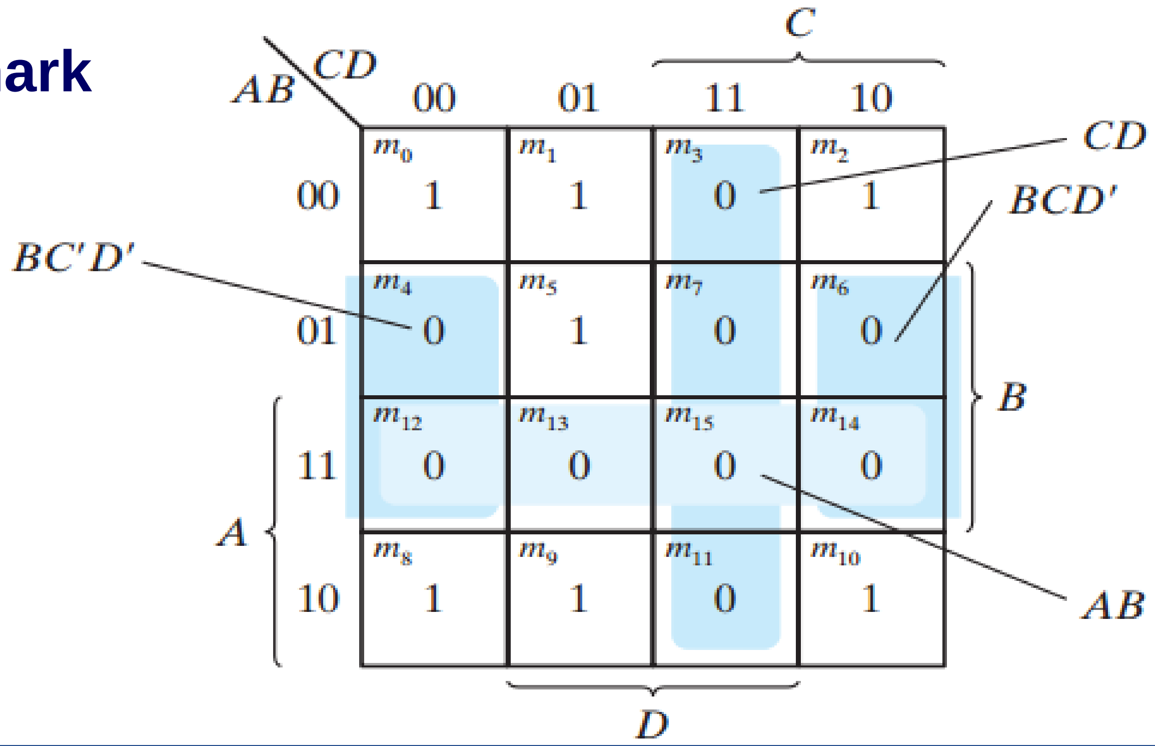
Example 3.7 of Product-of-Sums

EXAMPLE 3.7

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

- **Step-1: mark "0" & "1"**

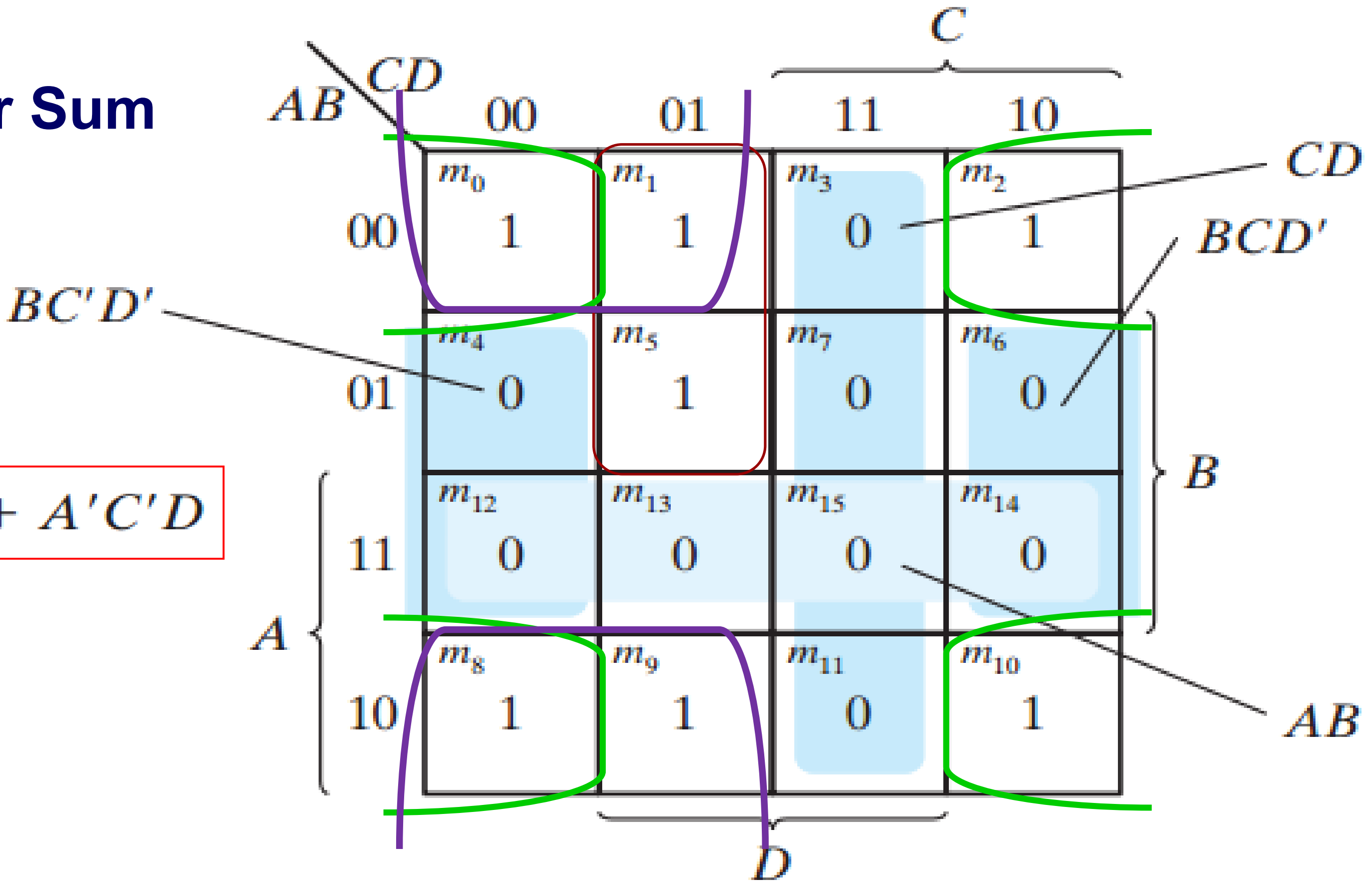


Example 3.7 of Product-of-Sums

- combine “1” for Sum of Products

Answer (a)

$$F = B'D' + B'C' + A'C'D$$



Example 3.7 of Product-of-Sums

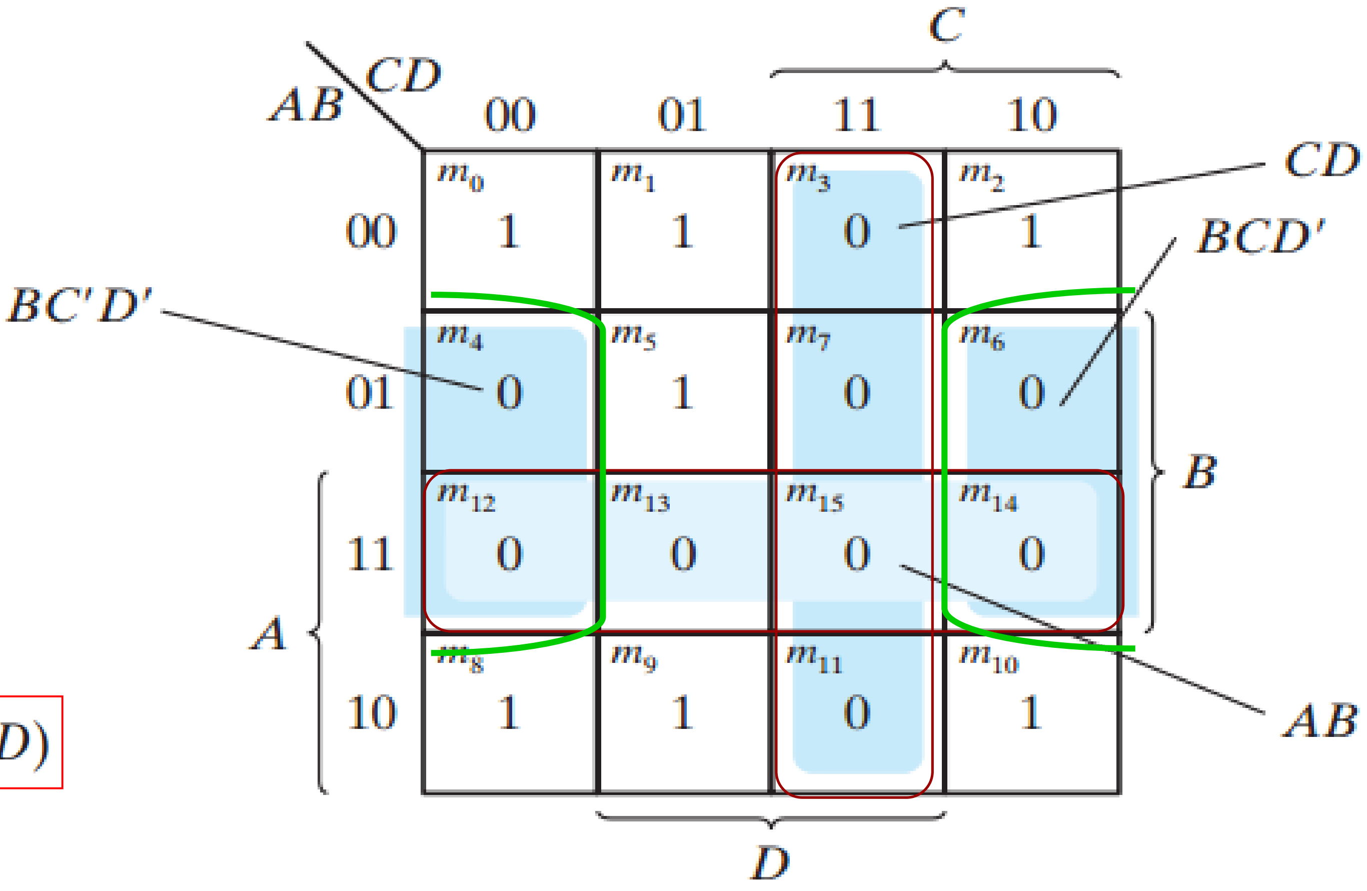
- **Step2, combine “0”**

$$F' = AB + CD + BD'$$

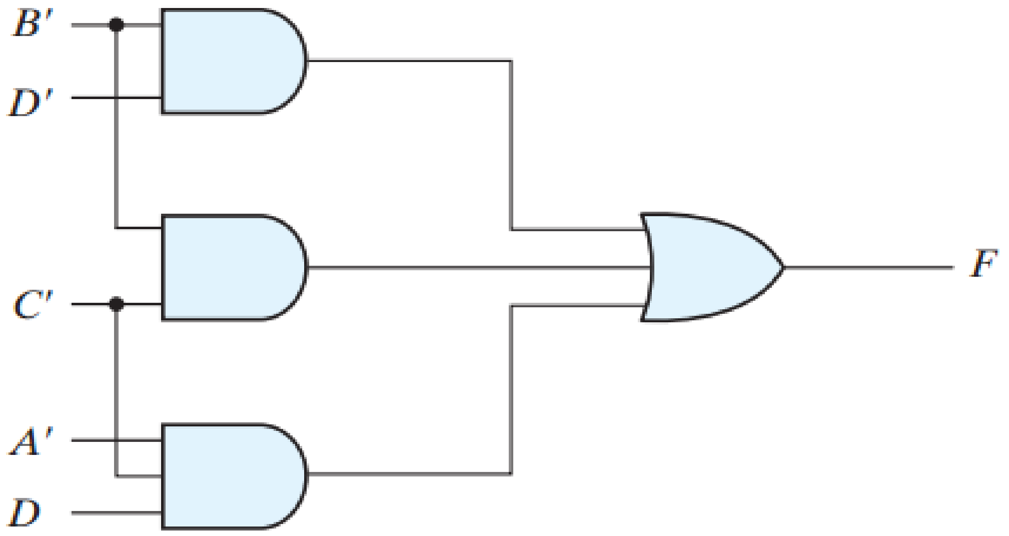
- **Step3, convert *F*’ into *F* by DeMorgan**

Answer (b)

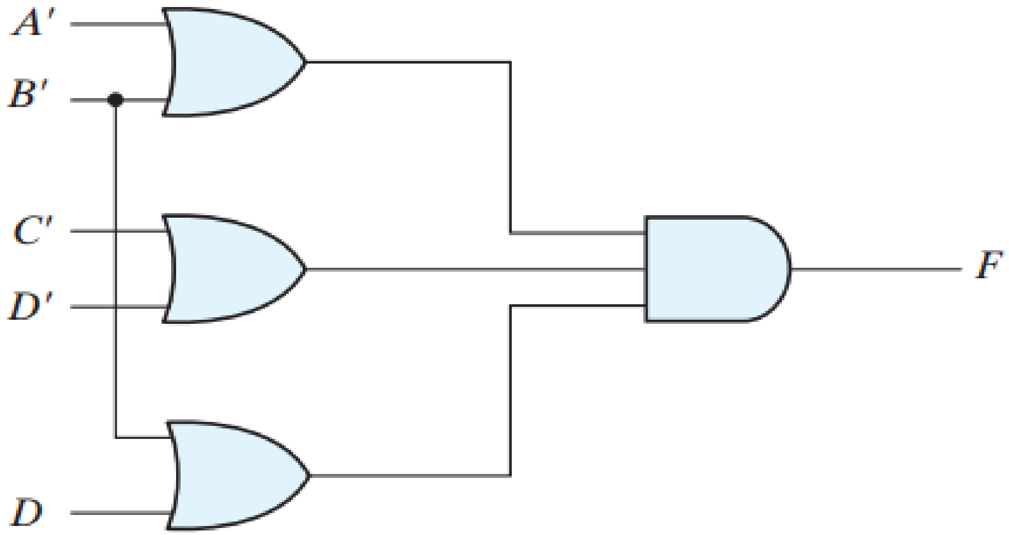
$$\boxed{F = (A' + B')(C' + D')(B' + D)}$$



Logic Gates of Example 3.7



(a) $F = B'D' + B'C' + A'C'D$



(b) $F = (A' + B')(C' + D')(B' + D)$

Simplify Product-of-Maxterms

- Example: Convert product-of-maxterms canonical form to a simplified function.

1. Step-1: mark "0" & "1"
2. Combine "1's" to get sum-of-products

$$F = x'z + xz'$$

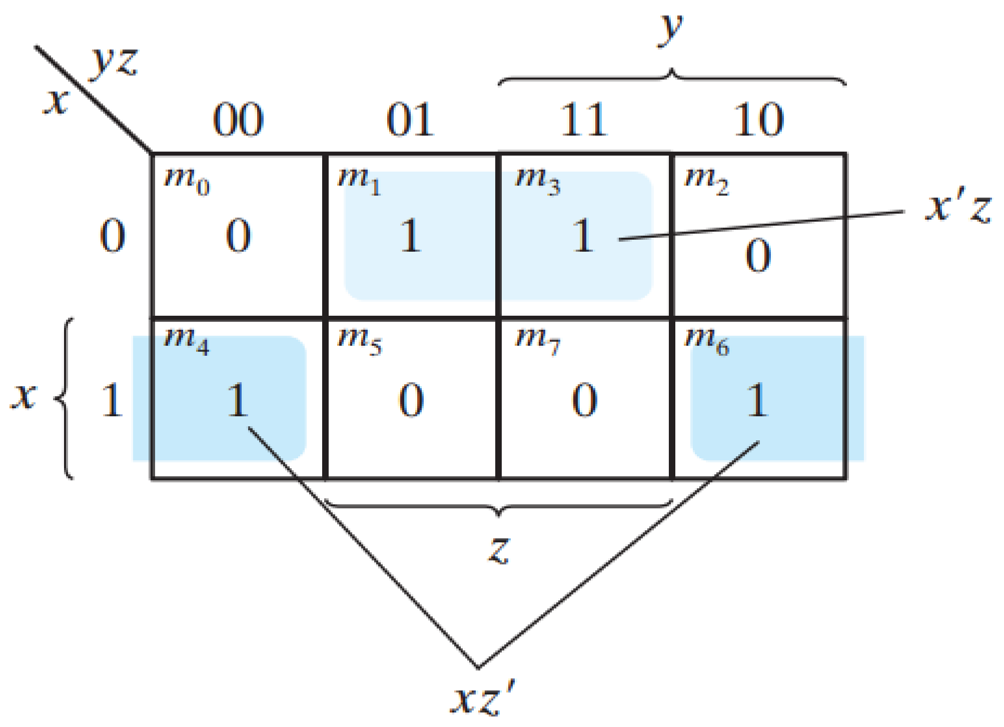
3. Merge "0" to get *F*' and use DeMorgan to obtain product-of-sums.

$$F' = xz + x'z'$$

$$\Downarrow$$

$$F = (x' + z')(x + z)$$

$$F(x, y, z) = \Pi(0, 2, 5, 7)$$



Don’t Care Conditions

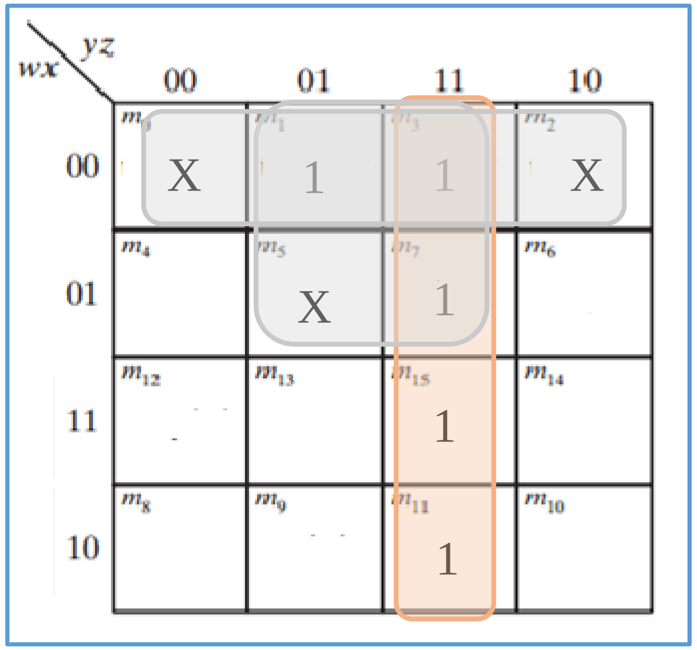
- In some applications, minterms of a function is not specified.
- It is customary to call the unspecified minterms of a function “don’t-care conditions”.
- To distinguish the don’t-care condition from 1’s and 0’s, an X is used.
- “Don’t-care minterms” may be assumed to be either 0 or 1.

EXAMPLE 3.8

Simplify the Boolean function $F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$
which has the don't-care conditions $d(w, x, y, z) = \Sigma(0, 2, 5)$

F = yz + w'x'

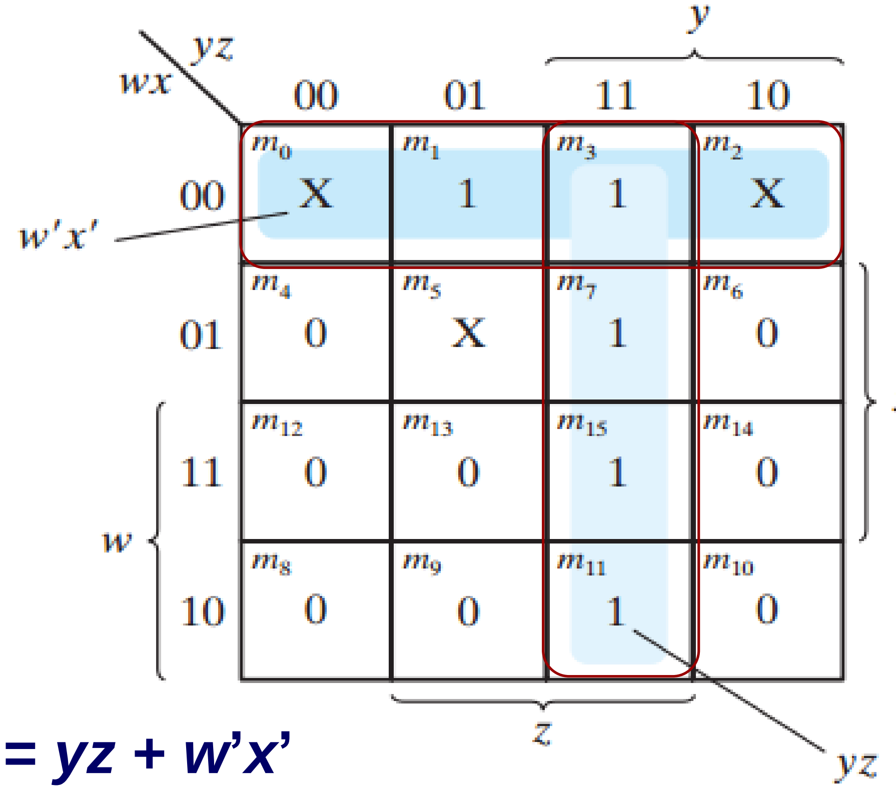
F = yz + w'z



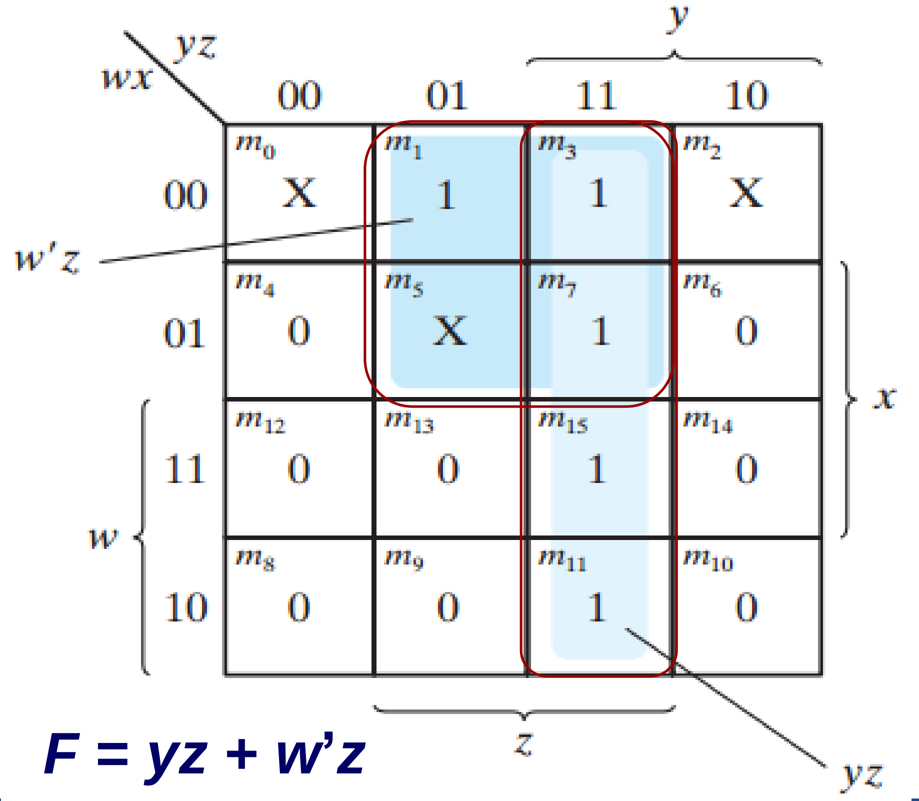
Example of "Don't Care Condition"

EXAMPLE 3.8

Simplify the Boolean function $F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$ which has the don't-care conditions $d(w, x, y, z) = \Sigma(0, 2, 5)$



F = yz + w'x'



F = yz + w'z

Quine-McCluskey (QM) Method
for Logic Minimization

- Minimizing Boolean functions using Karnaugh maps is practical only for up to four or five variables. Also, the Karnaugh map method does not lend itself to be automated in the form of a computer program.
- The **Quine-McCluskey** method is more practical for logic simplification of functions with more than four or five variables. It also has the advantage of being easily implemented with a computer or programmable calculator.
- The Quine-McCluskey method is functionally similar to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms.
- This method is sometimes referred to as the ***tabulation method***.

Quine-McCluskey (QM) Method

- To apply the Quine-McCluskey method, first write the function in standard **minterm** (SOP) form.

$$X = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + A\overline{B}C\overline{D} + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD$$

- **STEP-1**: The minterms that appear in the function are listed in the right column.

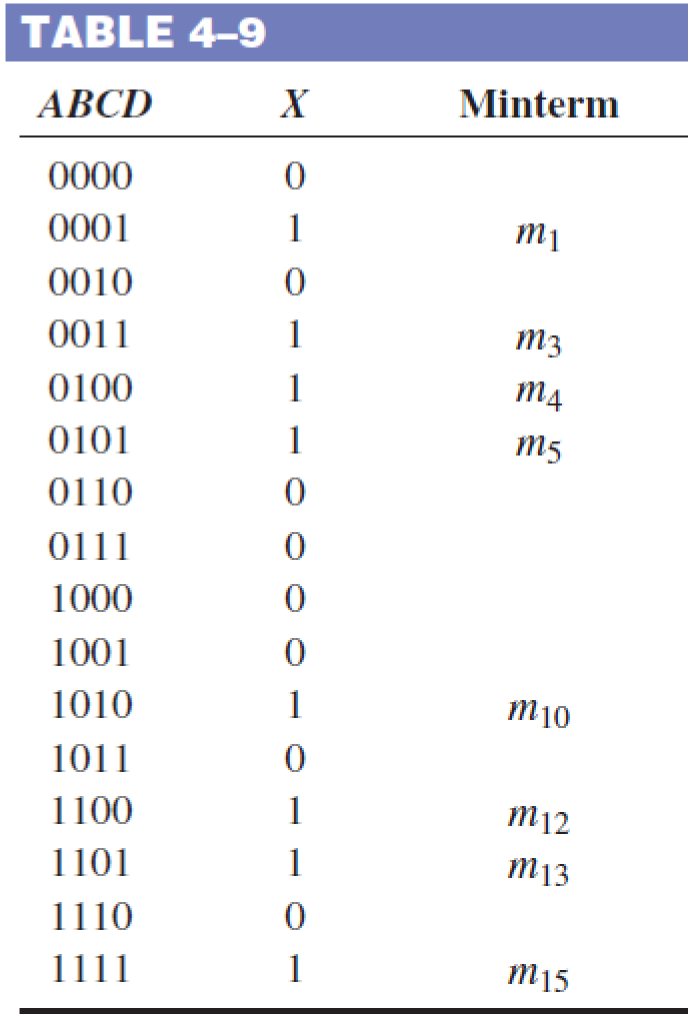


TABLE 4–9

$ABCD$	X	Minterm
0000	0	
0001	1	m_1
0010	0	
0011	1	m_3
0100	1	m_4
0101	1	m_5
0110	0	
0111	0	
1000	0	
1001	0	
1010	1	m_{10}
1011	0	
1100	1	m_{12}
1101	1	m_{13}
1110	0	
1111	1	m_{15}

Quine-McCluskey (QM) Method

STEP-1

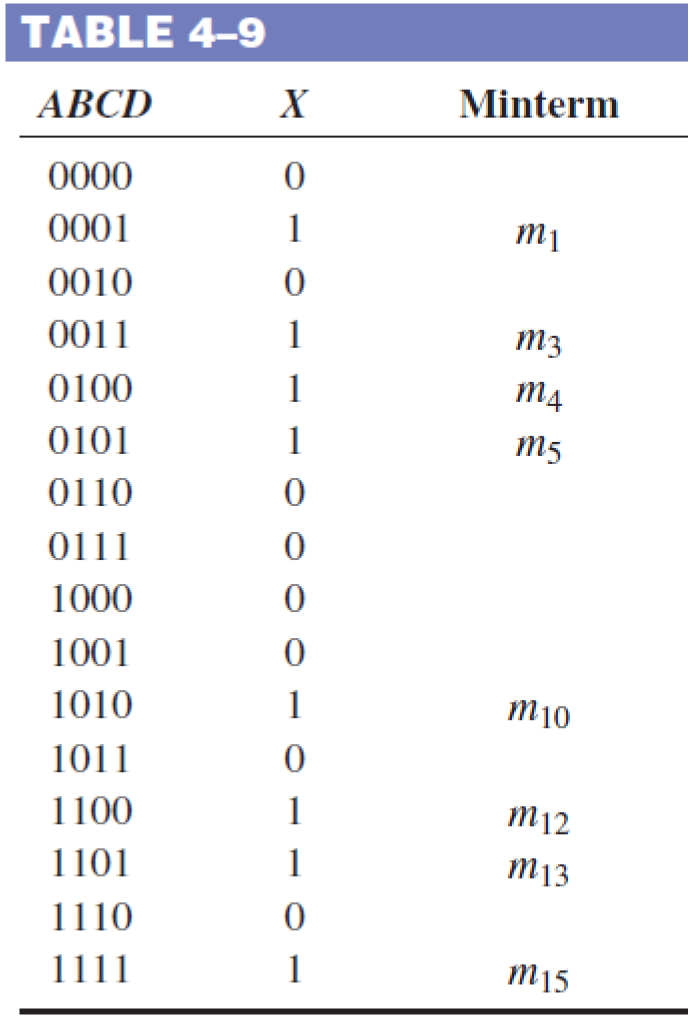


TABLE 4–9

$ABCD$	X	Minterm
0000	0	
0001	1	m_1
0010	0	
0011	1	m_3
0100	1	m_4
0101	1	m_5
0110	0	
0111	0	
1000	0	
1001	0	
1010	1	m_{10}
1011	0	
1100	1	m_{12}
1101	1	m_{13}
1110	0	
1111	1	m_{15}

- **STEP-2**: Arrange the minterms in the original expression in groups according to the number of 1s in each minterm.

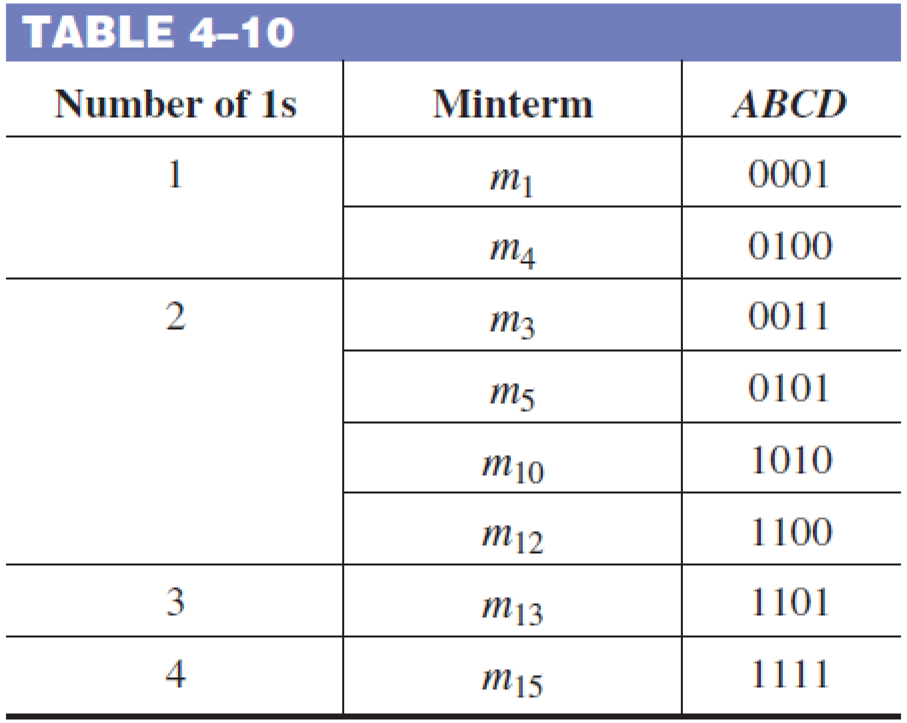


TABLE 4–10

Number of 1s	Minterm	$ABCD$
1	m_1	0001
	m_4	0100
2	m_3	0011
	m_5	0101
	m_{10}	1010
	m_{12}	1100
3	m_{13}	1101
4	m_{15}	1111

STEP-2

Quine-McCluskey (QM) Method

- **STEP-3**: Third, compare adjacent groups, looking to see if any minterms are the same in every position except one. If they are, place a checkmark by those two minterms, as shown in Table 4–11.
- You should check each minterm against all others in the following group, but it is not necessary to check any groups that are not adjacent.
- In the column labeled First Level, you will have a list of the minterm names and the binary equivalent with an x as the placeholder for the literal that differs.
  - In the example, minterm m_1 in Group 1 (0001) is identical to m_3 in Group 2 (0011) except for the C position, so place a check mark by these two minterms and enter 00x1 in the column labeled First Level.
  - If a given term can be used more than once, it should be.

TABLE 4–10

Number of 1s	Minterm	*ABCD*
1	m_1	0001
	m_4	0100
2	m_3	0011
	m_5	0101
	m_{10}	1010
	m_{12}	1100
3	m_{13}	1101
4	m_{15}	1111

TABLE 4–11

Number of 1s in Minterm	Minterm	ABCD	First Level
1	m_1	0001 ✓	(m_1, m_3) 00x1
	m_4	0100 ✓	(m_1, m_5) 0x01
2	m_3	0011 ✓	(m_4, m_5) 010x
	m_5	0101 ✓	(m_4, m_{12}) x100
	m_{10}	1010	(m_5, m_{13}) x101
	m_{12}	1100 ✓	(m_{12}, m_{13}) 110x
3	m_{13}	1101 ✓	(m_{13}, m_{15}) 11x1
4	m_{15}	1111 ✓	

Essential Prime Implicants (EPI)

Quine-McCluskey (QM) Method

STEP-4

- The terms listed in the *First Level* have been used to form a reduced table (Table 4–12) with one less group than before.
- The number of 1s remaining in the *First Level* are counted and used to form three new groups.
- Terms in the new groups are compared against terms in the adjacent group down. We need to compare these terms only if the x is in the same relative position in adjacent groups; otherwise go on.
- If the two expressions differ by exactly one position, a check mark is placed next to both terms as before and all of the minterms are listed in the Second Level list.

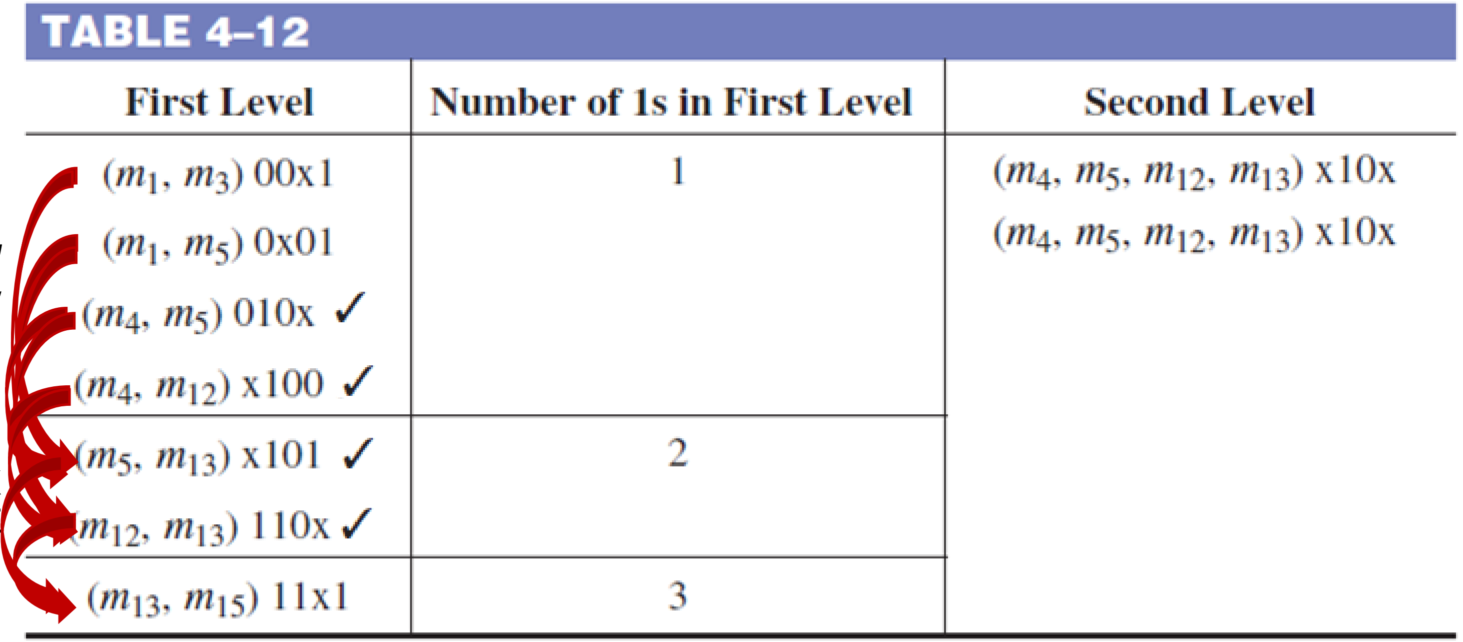


TABLE 4–12

First Level	Number of 1s in First Level	Second Level
(m_1, m_3) 00x1	1	$(m_4, m_5, m_{12}, m_{13})$ x10x
(m_1, m_5) 0x01		$(m_4, m_5, m_{12}, m_{13})$ x10x
(m_4, m_5) 010x ✓		
(m_4, m_{12}) x100 ✓		
(m_5, m_{13}) x101 ✓	2	
(m_{12}, m_{13}) 110x ✓		
(m_{13}, m_{15}) 11x1	3	

The terms that are unchecked will form other terms in the final reduced expression. The first unchecked term is read as *A'B'D*. The next one is read as *A'C'D*. The last unchecked term is *ABD*. Recall that m_{10} was an essential prime implicant, so is picked up in the final expression.

$$X = B\overline{C} + \overline{A}\,\overline{B}D + \overline{A}\,\overline{C}D + ABD + A\overline{B}C\overline{D}$$

Quine-McCluskey (QM) Method

$$X = B\overline{C} + \overline{A}\,\overline{B}D + \overline{A}\,\overline{C}D + ABD + A\overline{B}C\overline{D}$$

- Although this expression is correct, it may not be the minimum possible expression. There is a final check that can eliminate any unnecessary terms. The terms for the expression are written into a prime implicant table, with minterms for each prime implicant checked, as shown in Table 4–1

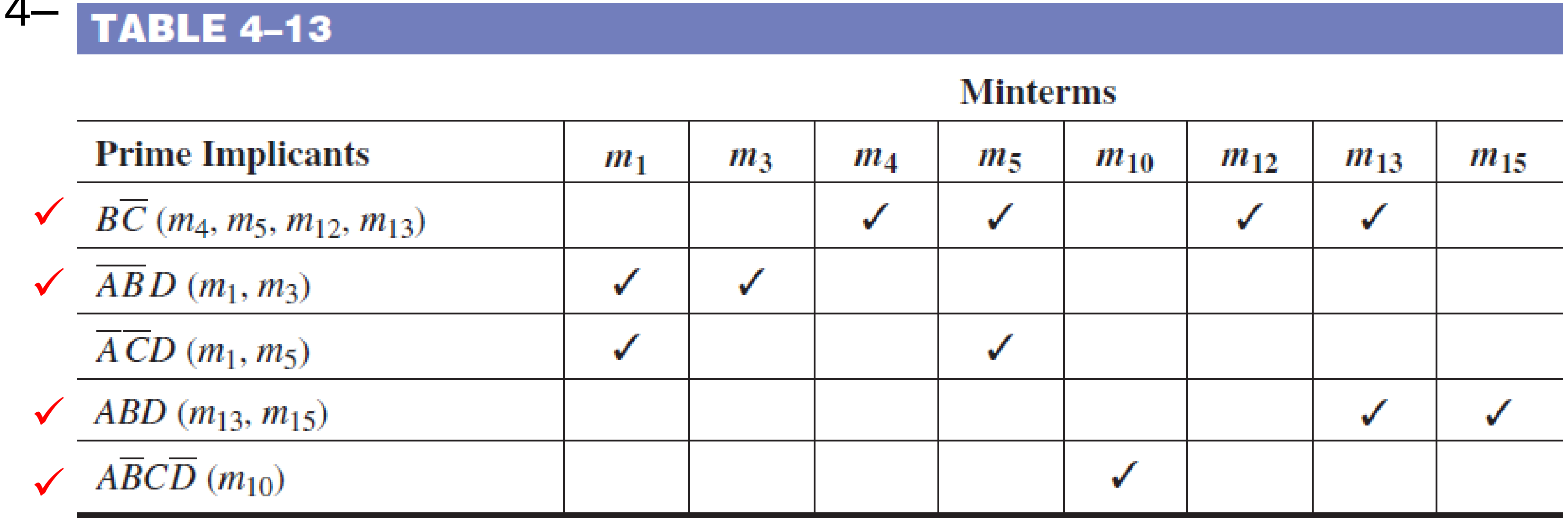


TABLE 4–13

		Minterms							
	Prime Implicants	m_1	m_3	m_4	m_5	m_{10}	m_{12}	m_{13}	m_{15}
✓	$B\overline{C}\ (m_4, m_5, m_{12}, m_{13})$			✓	✓		✓	✓	
✓	$\overline{A}\,\overline{B}D\ (m_1, m_3)$	✓	✓						
	$\overline{A}\,\overline{C}D\ (m_1, m_5)$	✓			✓				
✓	$ABD\ (m_{13}, m_{15})$							✓	✓
✓	$A\overline{B}C\overline{D}\ (m_{10})$					✓			

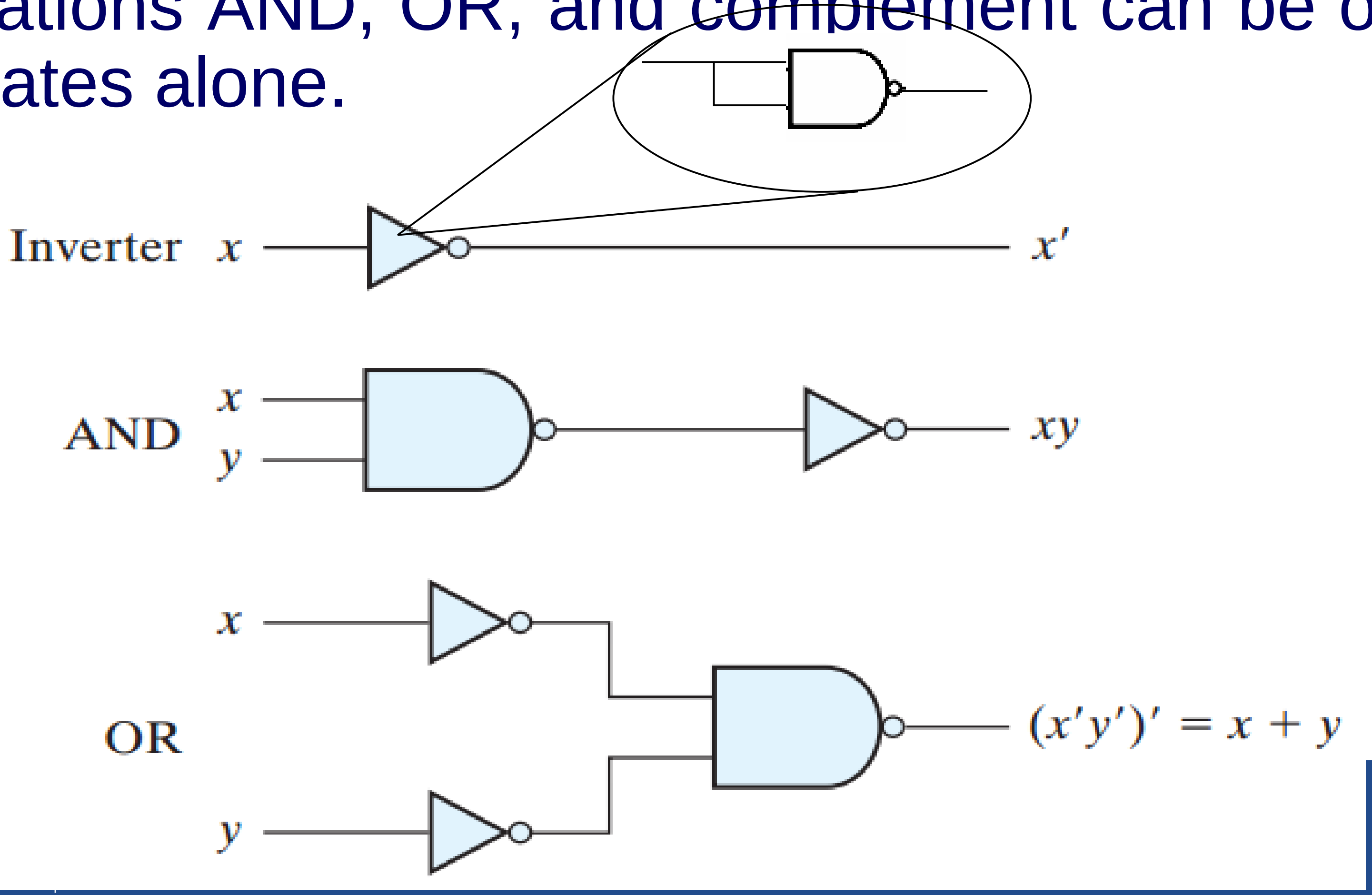
- If a minterm has a single check mark, then the prime implicant is essential and must be included in the final expression. The term ABD must be included because m_{15} is only covered by it. Likewise m_{10} is only covered by AB’CD’, so it must be in the final expression. Notice that the two minterms in A’C’D are covered by the prime implicants in the first two rows, so this term is unnecessary. $\boxed{X = B\overline{C} + \overline{A}\,\overline{B}D + ABD + A\overline{B}C\overline{D}}$

NAND & NOR Implementation

- Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates.
- NAND and NOR gates are easier to fabricate and are the basic gates used in IC digital logic families.
- The NAND and NOR gates are said to be universal gate because any logic circuit can be implemented with it.

NAND Circuits

- The NAND gate is said to be a universal gate because any logic circuit can be implemented with it.
- Logical operations AND, OR, and complement can be obtained with NAND gates alone.



NAND Circuits

- A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic.
- AND-invert → NAND; invert-OR →NAND
- In part (b), we can place a bubble (NOT) in each input and apply the DeMorgan's theorem, then get a Boolean function in NAND type.

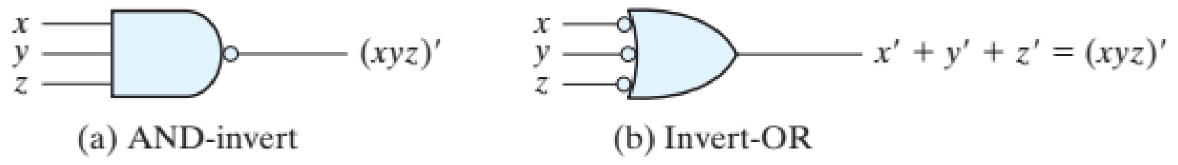
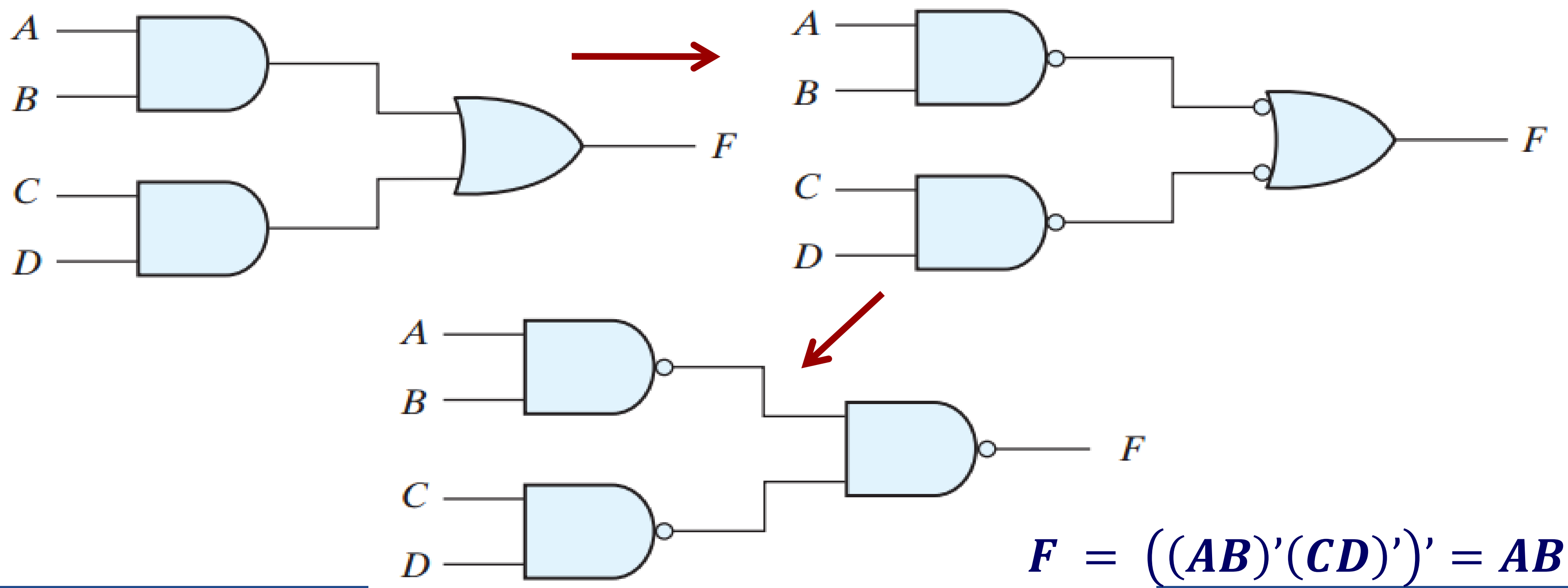


FIGURE 3.17
Two graphic symbols for a three-input NAND gate

Converted to NAND Gates

- The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.

$$F = AB + CD$$



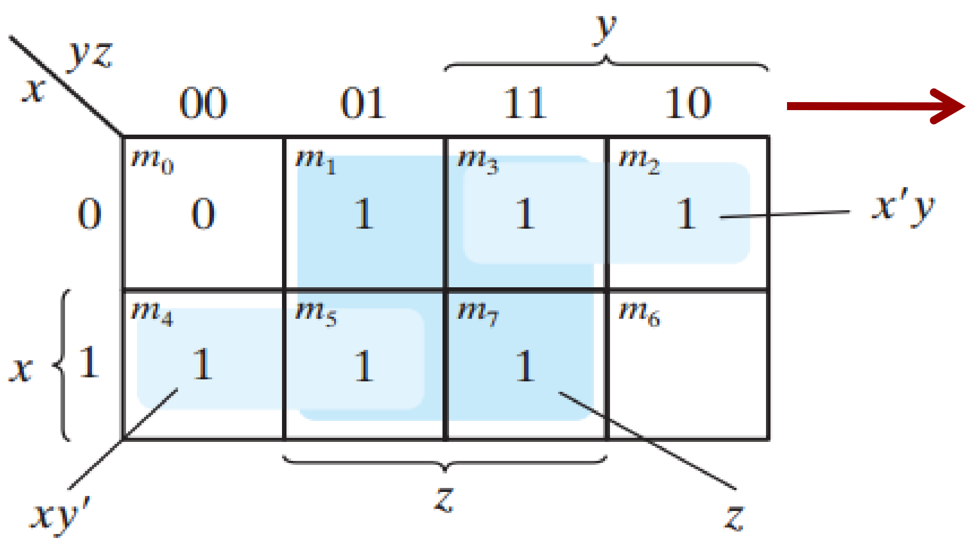
$$F = \left((AB)'(CD)'\right)' = AB + CD$$

Two Level NAND Gates Implementation

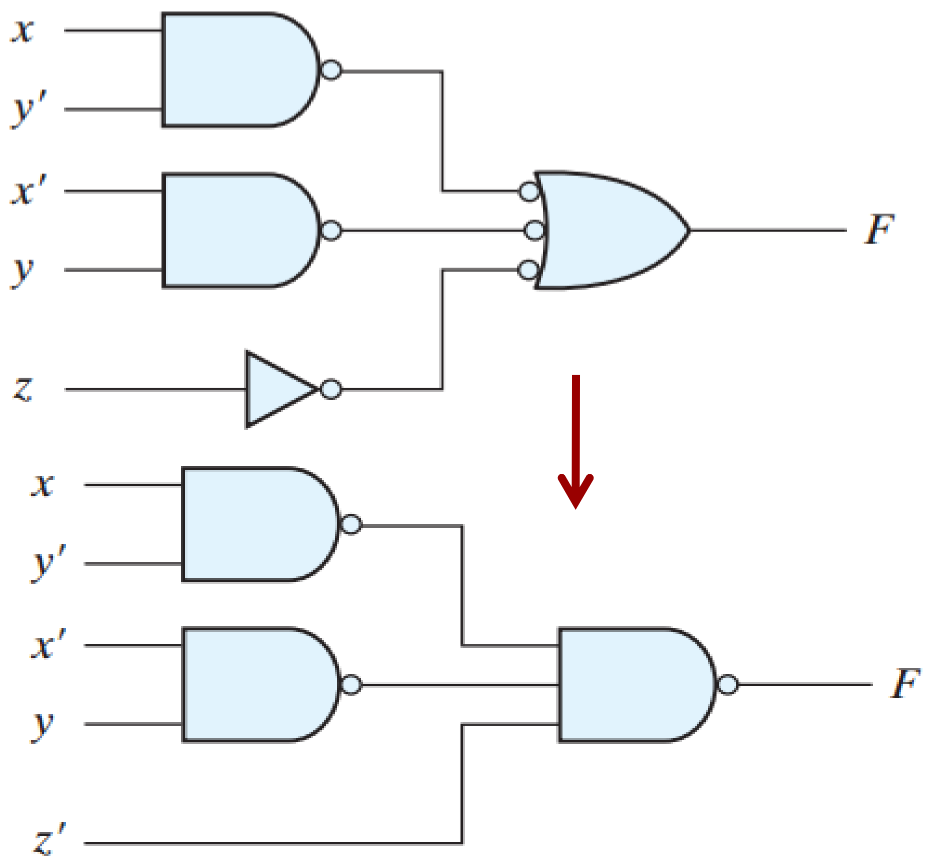
EXAMPLE 3.9

Implement the following Boolean function with NAND gates:

$$F(x, y, z) = \Sigma(1, 2, 3, 4, 5, 7)$$

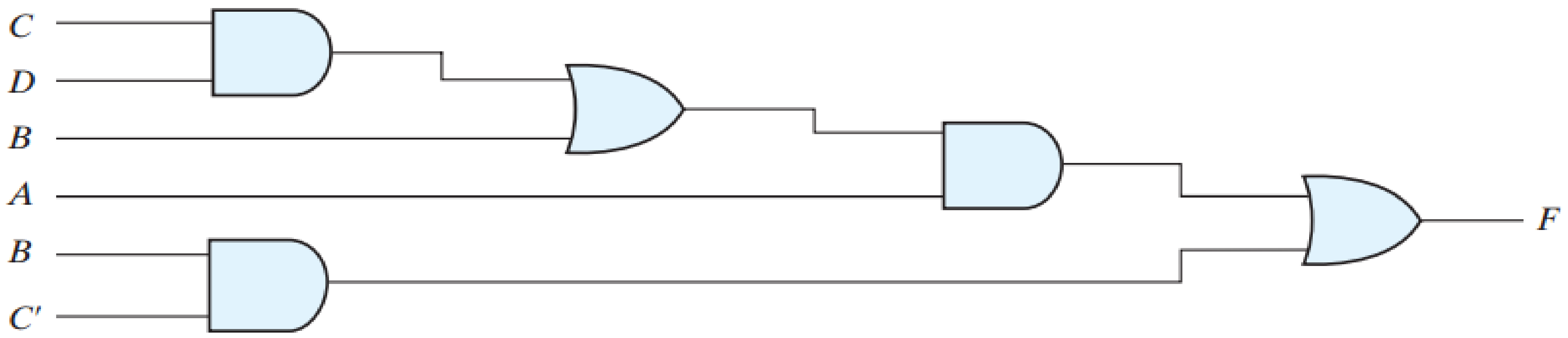


$$F = xy' + x'y + z$$

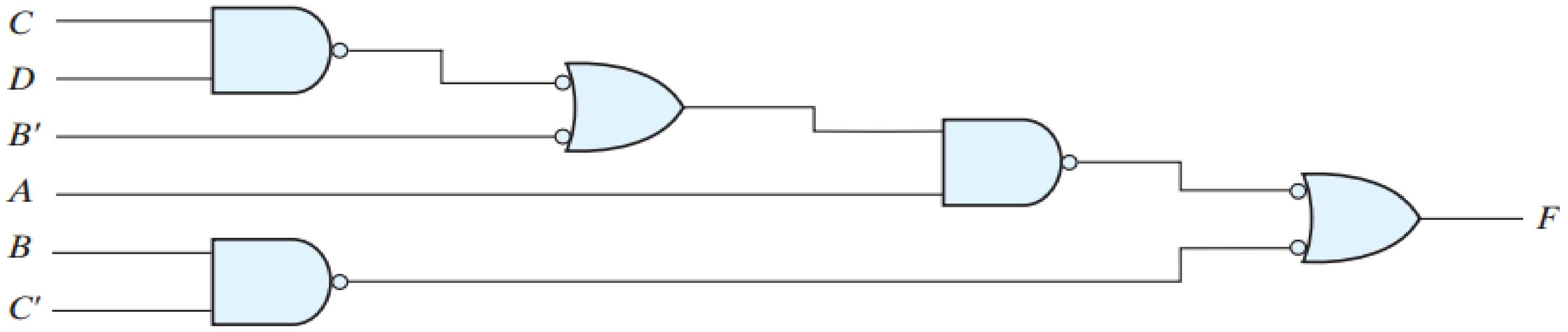


Multilevel NAND Circuits

$$F = A(CD + B) + BC'$$



(a) AND–OR gates



(b) NAND gates

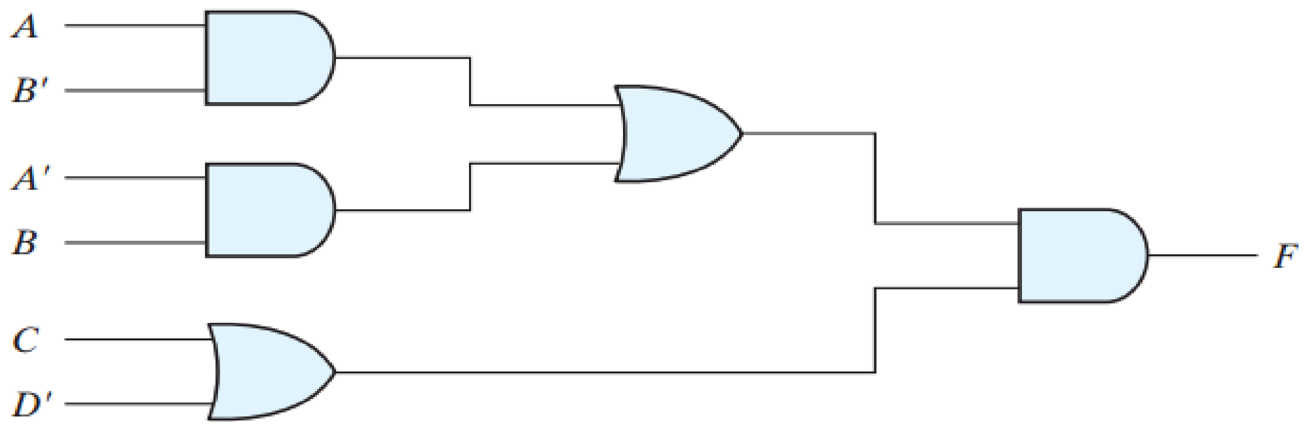
Multilevel NAND Circuits

The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

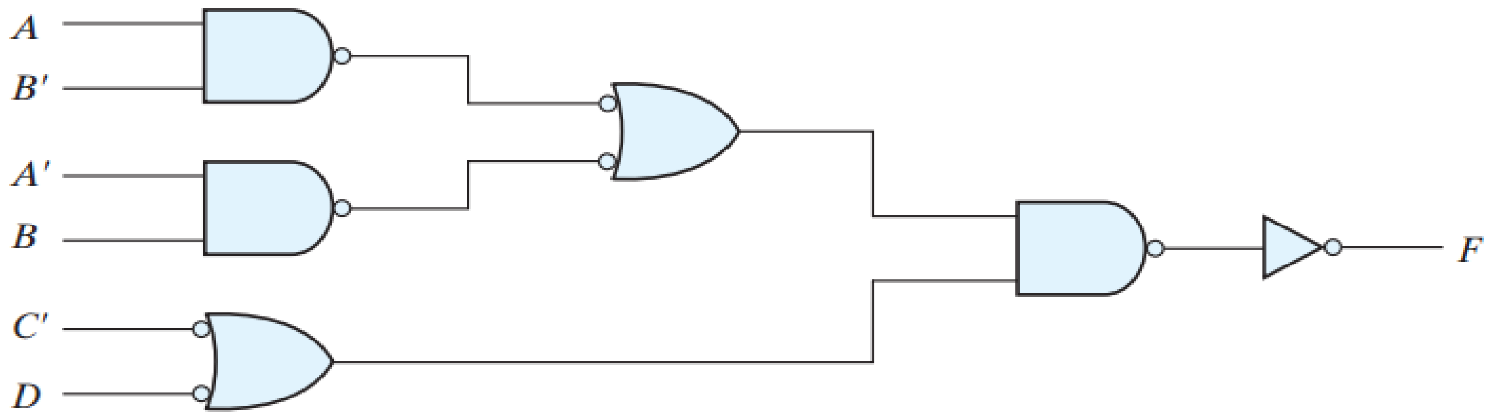
1. Convert all **AND** gates to NAND gates with **AND-invert** graphic symbols.
2. Convert all **OR** gates to NAND gates with **invert-OR** graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

Multilevel NAND Circuits

$F = (AB' + A'B)(C + D')$



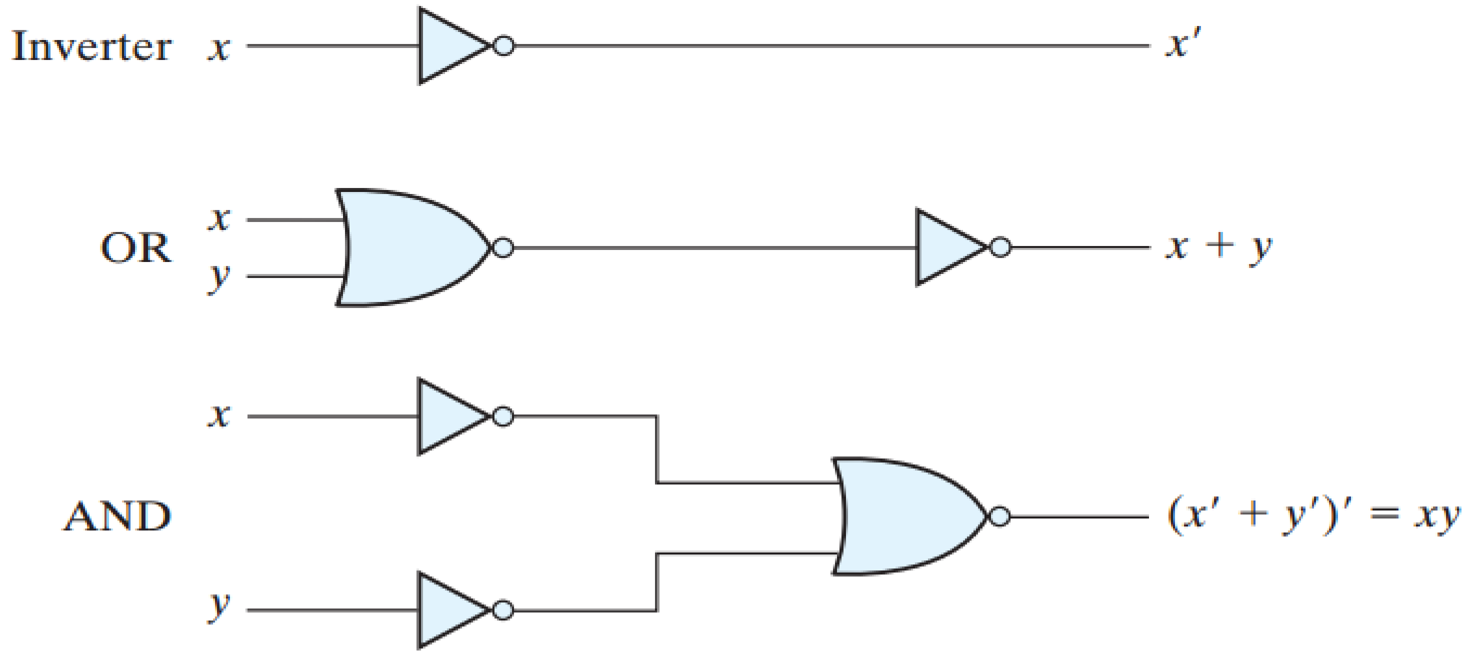
(a) AND–OR gates



(b) NAND gates

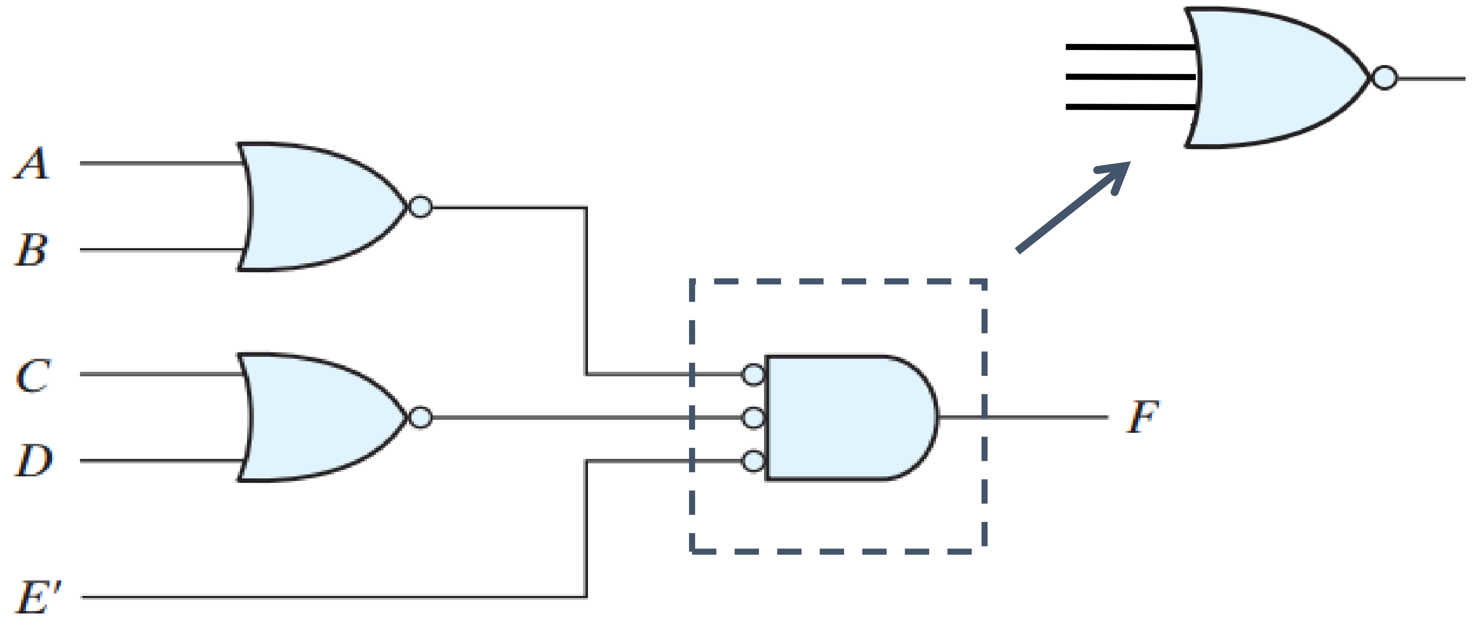
NOR Implementation

- The NOR gate is another universal gate that can be used to implement any Boolean function.
- A two-level implementation with NOR gates requires a function in a form of product-of-sums.



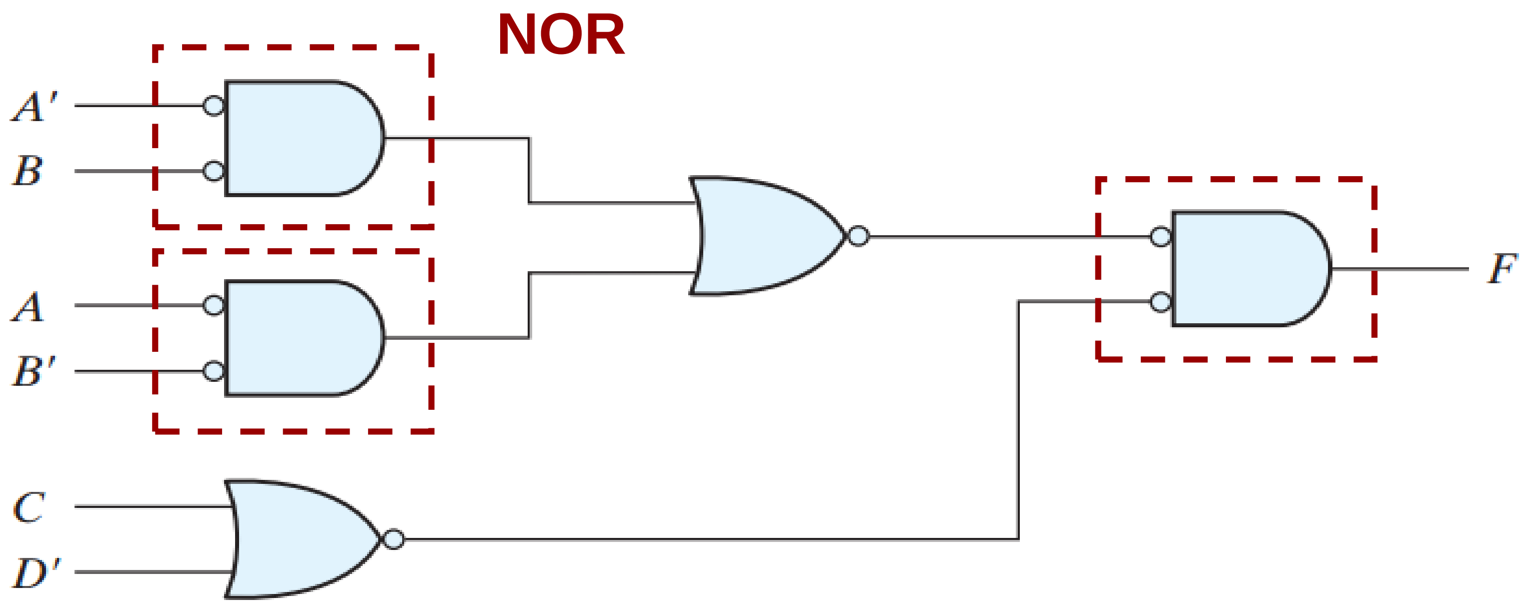
Two-Level NOR Implementation

$$F = (A + B)(C + D)E$$



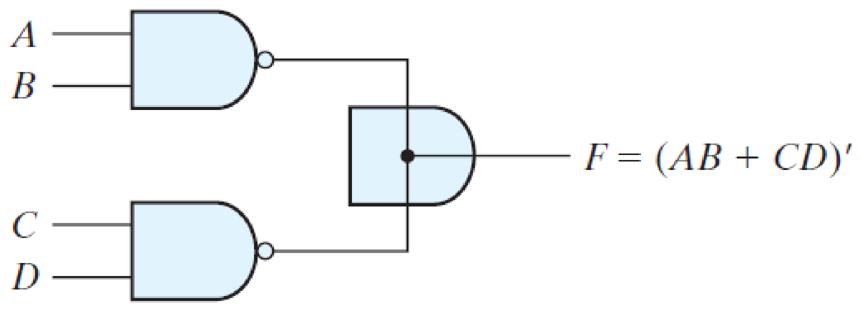
Multi-Level NOR Implementation

$$F = (AB' + A'B)(C + D')$$



OTHER TWO-LEVEL IMPLEMENTATIONS

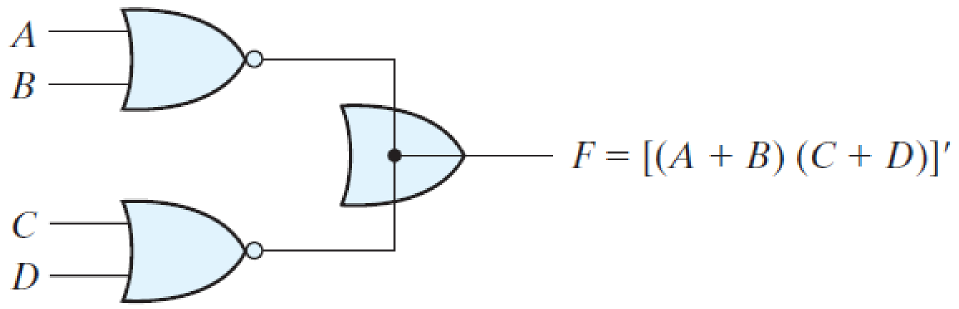
- Some (but not all) NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function.
- This type of logic is called *wired logic.*



(a) Wired-AND in open-collector TTL NAND gates.

(AND–OR–INVERT)

$$F = (AB)' \cdots (CD)' = (AB + CD)'$$



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

$$F = (A + B)' + (C + D)' \\ = [(A + B)(C + D)]'$$

FIGURE 3.26
Wired logic
(a) Wired-AND logic with two NAND gates
(b) Wired-OR in emitter-coupled logic (ECL) gates

Nondegenerate Forms

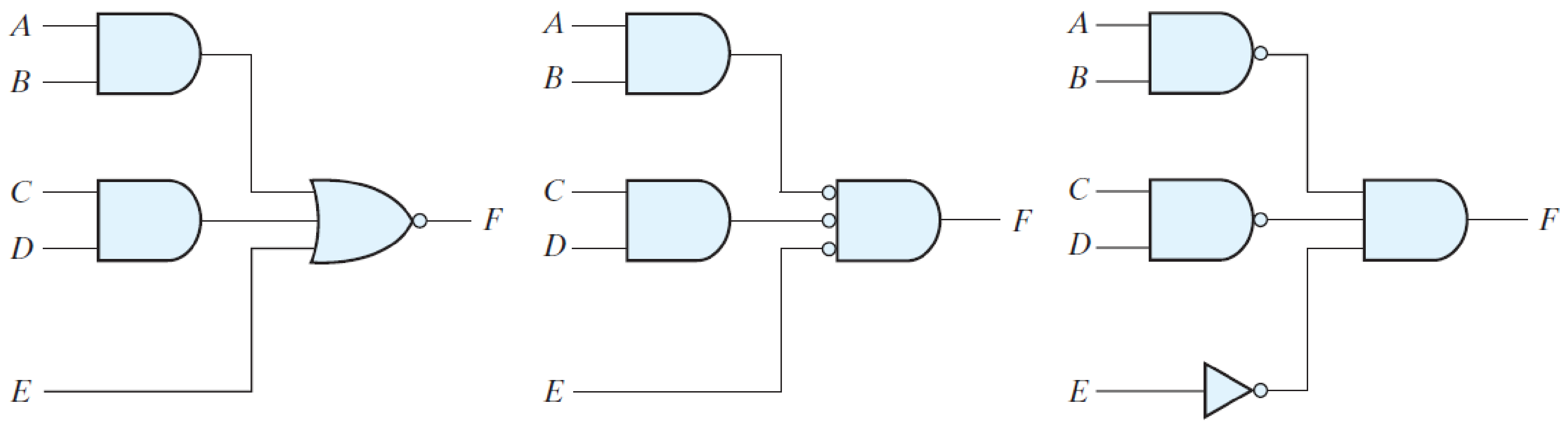
- Eight of these combinations are said to be *degenerate* forms because they degenerate to a single operation.
- The remaining eight *nondegenerate* forms produce an implementation in sum-of-products form or product-of-sums form.

AND-AND (AND)	OR-AND	NAND-AND	NOR-AND (NOR)
AND-OR	OR-OR (OR)	NAND-OR (NAND)	NOR-OR
AND-NAND (NAND)	OR-NAND	NAND-NAND	NOR-NAND (OR)
AND-NOR	OR-NOR (NOR)	NAND-NOR (AND)	NOR-NOR

Nondegenerate Forms

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

AND–OR–INVERT function



(a) AND–NOR (b) AND–NOR (c) NAND–AND

FIGURE 3.27
AND–OR–INVERT circuits, $F = (AB + CD + E)'$

Nondegenerate Forms

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

OR–AND–INVERT function

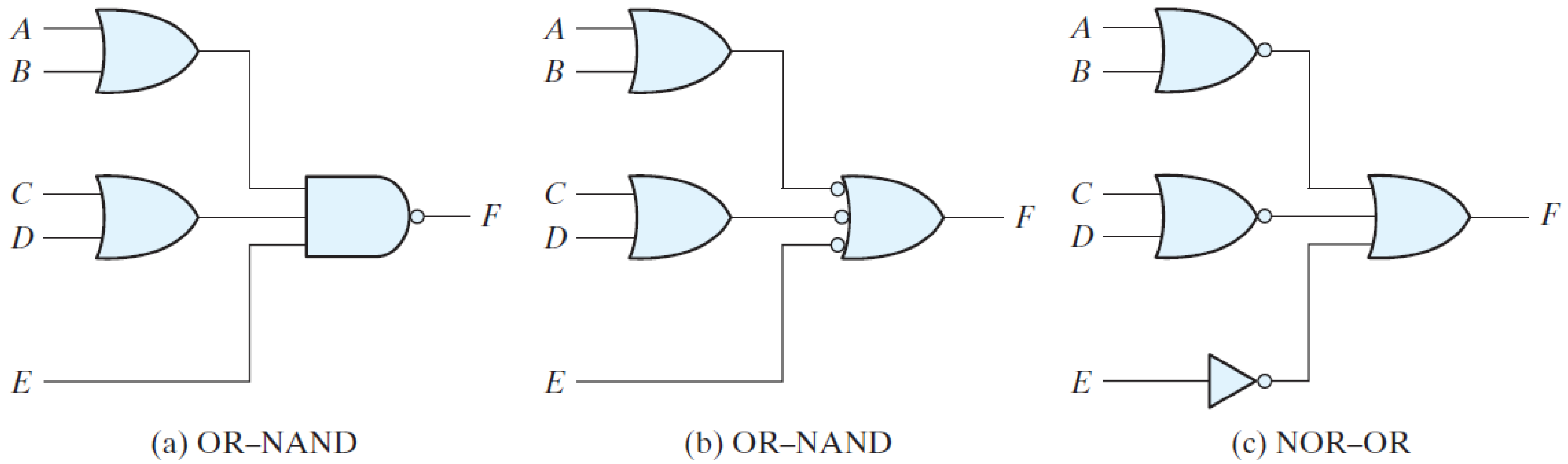


FIGURE 3.28
OR–AND–INVERT circuits, $F = [(A + B)(C + D)E]'$

Tabular Summary of AOI and OAI

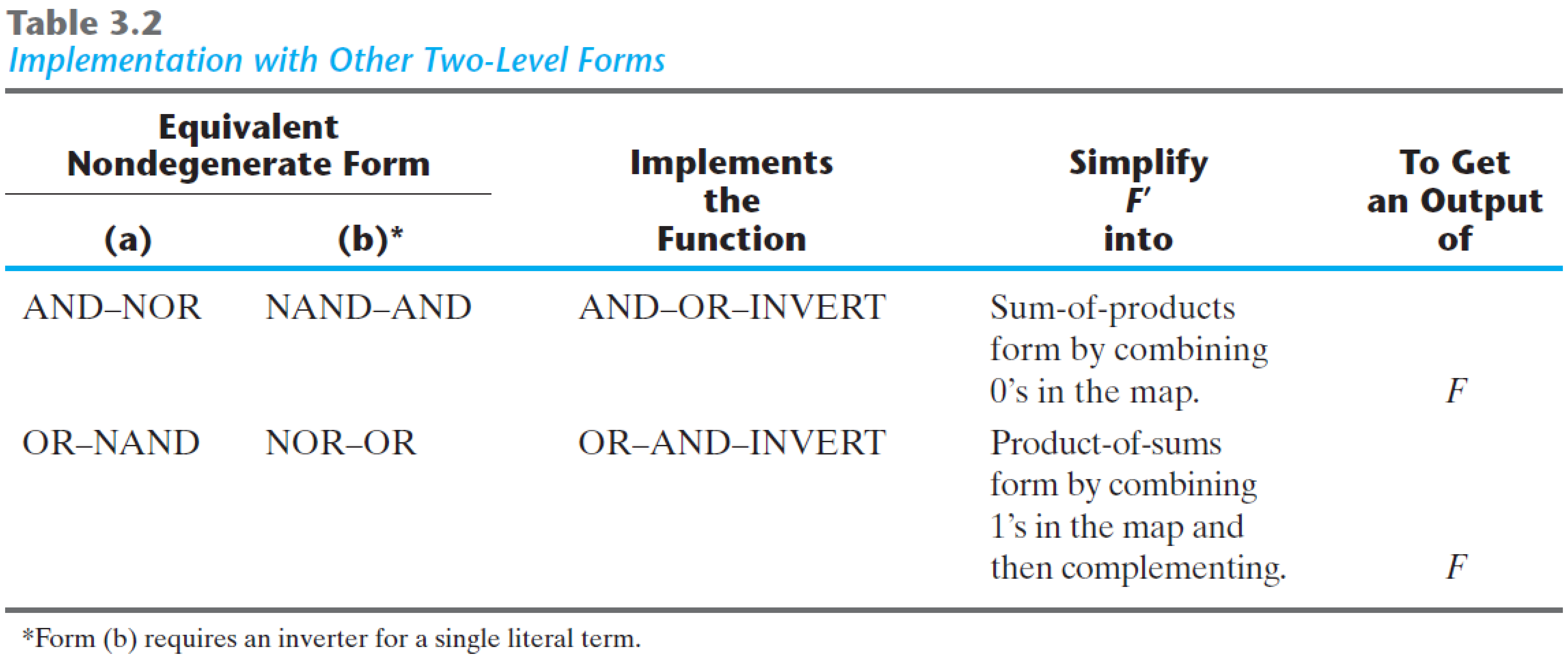


Table 3.2
Implementation with Other Two-Level Forms

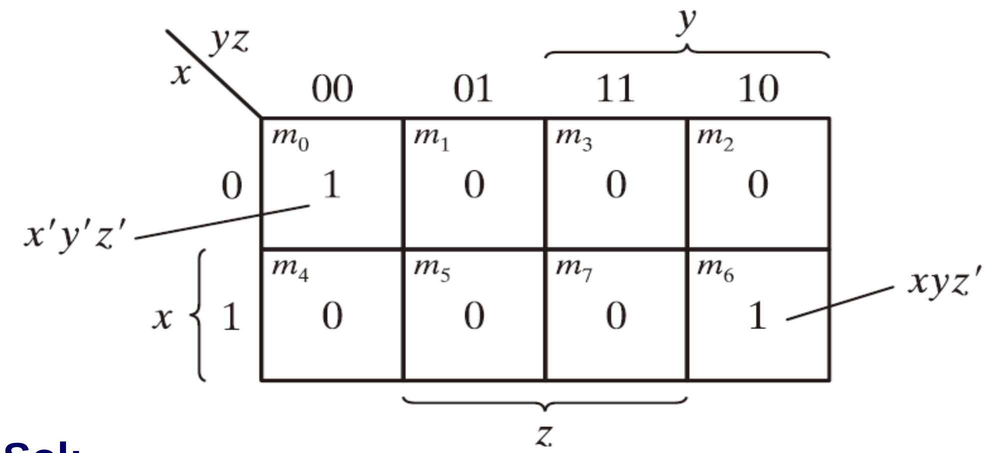
Equivalent Nondegenerate Form		**Implements the Function**	**Simplify F' into**	**To Get an Output of**
(a)	**(b)***			
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

Example of AOI and OAI

Example 3-10 Implement the function listed in the table below with:

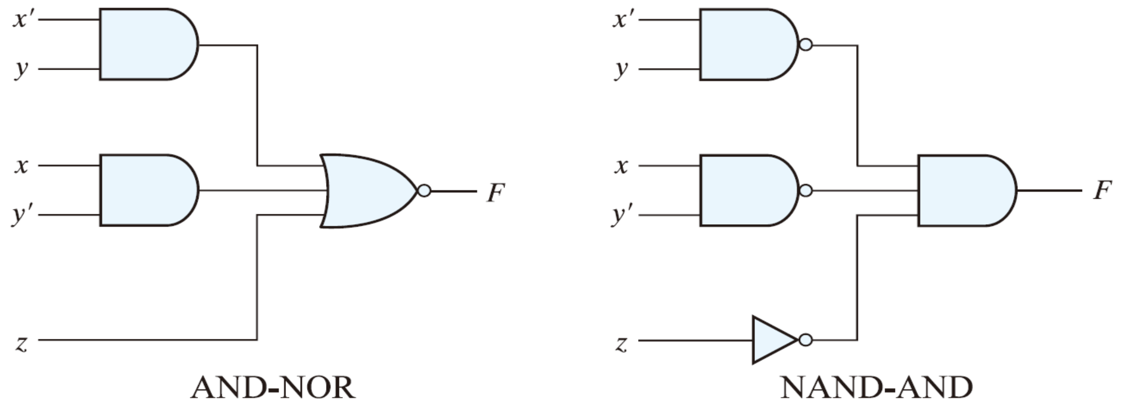
1. AND-NOR 2. NAND-AND 3. OR-NAND 4. NOR-OR



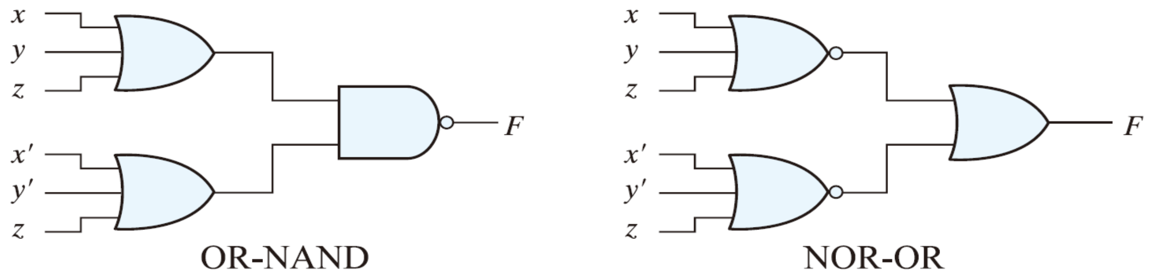
(a) Map simplification in sum of products

$F = x'y'z' + xyz'$

$F' = x'y + xy' + z$



(b) $F = (x'y + xy' + z)'$



(c) $F = [(x + y + z)(x' + y' + z)]'$

Sol:

- *F* ' = *x*'*y*+*xy*'+*z* (*F*': SOP, considering 0's in the map)
- *F* = (*x*'*y*+*xy*'+*z*) ' (*F*: AOI implementation)

- *F* = *x*'*y*'*z*' + *xyz*' (*F*: SOP, considering 1's in the map)
- *F* ' = (*x*+*y*+*z*)(*x*'+*y*'+*z*) (*F*': POS from DeMorgan)
- *F* = ((*x*+*y*+*z*)(*x*'+*y*'+*z*))' (*F*: OAI implementation)

Exclusive-OR function

The exclusive-OR (XOR), denoted by the symbol $\oplus$, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

- X-NOR

$$(x \oplus y)' = xy + x'y'$$

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

- The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

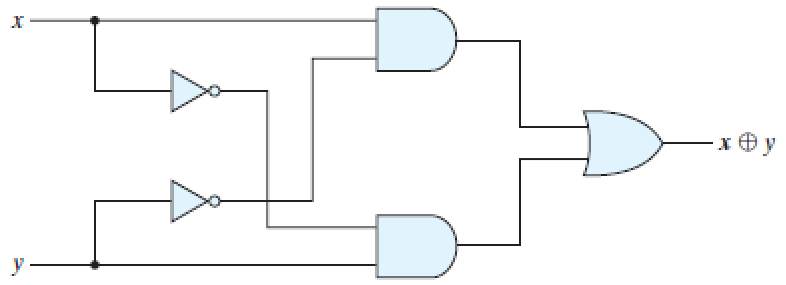
$$x \oplus y' = x' \oplus y = (x \oplus y)'$$

Exclusive-OR function

- The XOR and XNOR gates are commutative and associative

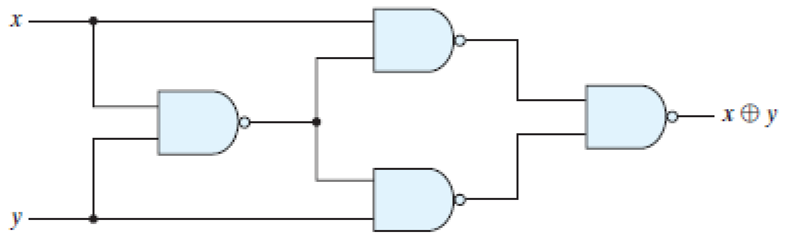
$$A \oplus B = B \oplus A$$
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

- Multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact, even a two-input function is usually constructed with other types of gates.



(a) Exclusive-OR with AND–OR–NOT gates

$$x \oplus y = xy' + x'y$$



(b) Exclusive-OR with NAND gates

$$(x' + y')x + (x' + y')y = xy' + x'y = x \oplus y$$

FIGURE 3.30
Exclusive-OR implementations

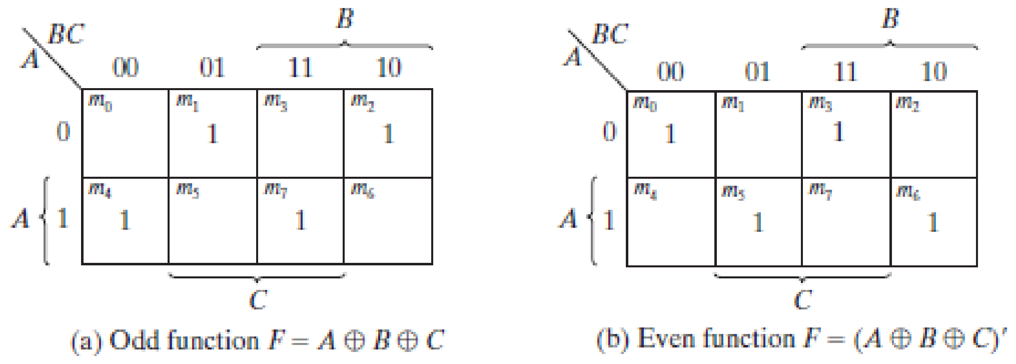
- Used in arithmetic operations and error detection and correction circuits.

Multiple variable Exclusive-OR (Odd function)

- The three-variable case can be converted to a Boolean expression as follows:

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

- The three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Hence, Odd function.



BC
A
B
00
01
11
10
0
1
A
C
m_0
m_1
m_3
m_2
m_4
m_5
m_7
m_6
1
(a) Odd function $F = A \oplus B \oplus C$
(b) Even function $F = (A \oplus B \oplus C)'$

FIGURE 3.31
Map for a three-variable exclusive-OR function

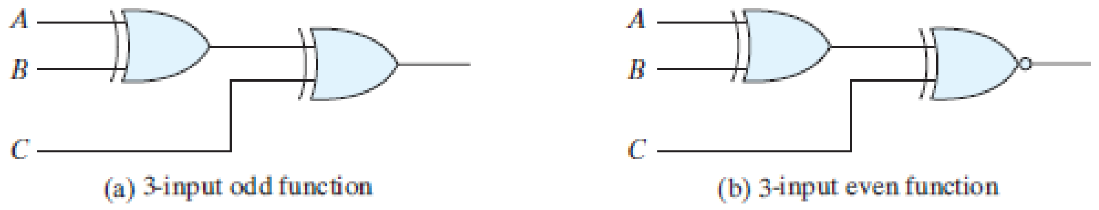
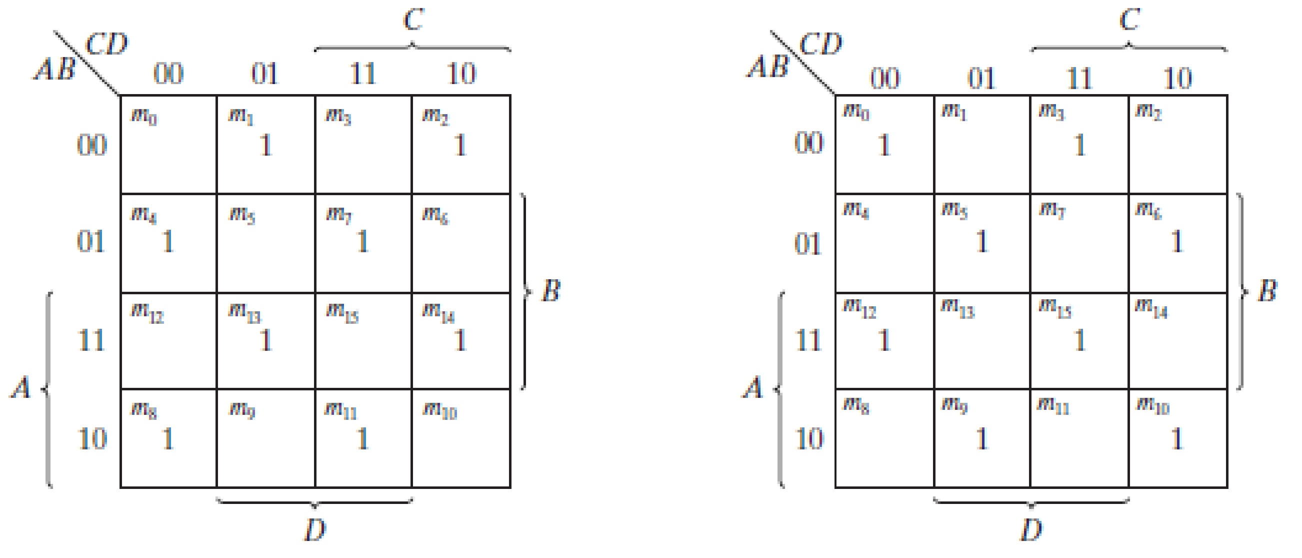


FIGURE 3.32
Logic diagram of odd and even functions

Multiple variable Exclusive-OR (Odd function)

- Odd function: The binary values of all the minterms have an odd number of 1's.
- The complement of an odd function is an even function.
- Even function: the four-variable even function is equal to 1 when an even number of its variables is equal to 1.



CD
AB
C
00 01 11 10
B
A
D
m_0 m_1 1 m_3 m_2 1
m_4 1 m_5 m_7 1 m_6
m_{12} m_{13} 1 m_{15} m_{14} 1
m_8 1 m_9 m_{11} 1 m_{10}
m_0 1 m_1 m_3 1 m_2
m_4 m_5 1 m_7 m_6 1
m_{12} 1 m_{13} m_{15} 1 m_{14}
m_8 m_9 1 m_{11} m_{10} 1

(a) Odd function $F = A \oplus B \oplus C \oplus D$　　(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

FIGURE 3.33
Map for a four-variable exclusive-OR function

Parity Generation and Checking

- A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even.
- The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.
- The circuit that generates the parity bit in the transmitter is called a parity generator. The circuit that checks the parity in the receiver is called a parity checker.

Parity Generation and Checking

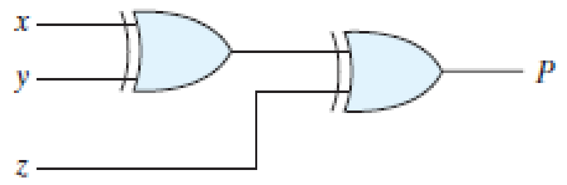
- As an example, consider a three-bit message to be transmitted together with an even-parity bit.

Table 3.3
Even-Parity-Generator Truth Table

Three-Bit Message			Parity Bit
x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

P: odd function

$$P = x \oplus y \oplus z$$



(a) 3-bit even parity generator

- The three bits— x, y, and z —constitute the message and are the inputs to the circuit. The parity bit P is the output.
- For even parity, the bit P must be generated to make the total number of 1's (including P) even.

Parity Generation and Checking

- Since the information was transmitted with even parity, the four bits received must have an even number of 1’s. An error occurs during the transmission if the four bits received have an odd number of 1’s, indicating that one bit has changed in value during transmission.
- The output of the parity checker (C) , will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1’s.

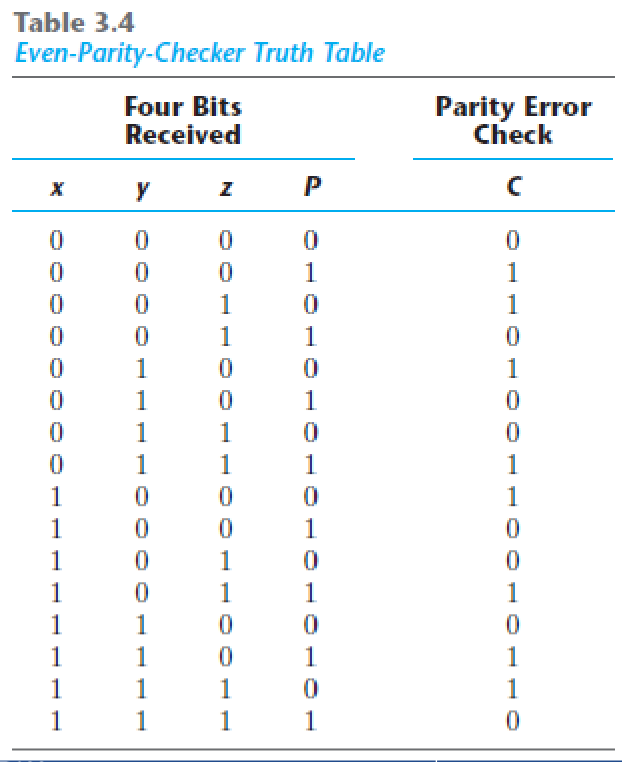
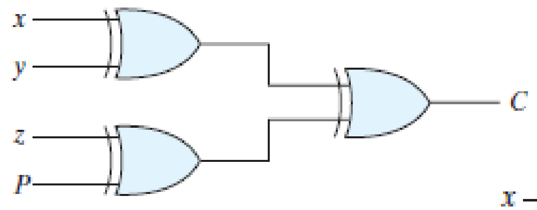


Table 3.4
Even-Parity-Checker Truth Table

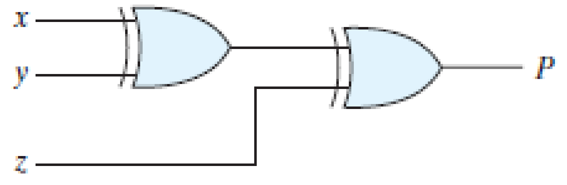
Four Bits Received				Parity Error Check
x	y	z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

C: odd function

$$C = x \oplus y \oplus z \oplus P$$



(b) 4-bit even parity checker



(a) 3-bit even parity generator

The End

Reference:

1. **Digital Design (with an introduction to the Verilog HDL) 6th Edition, M. Morris Mano, Michael D. Ciletti**