Chapter 3: Gate-Level Minimization

3.1 Introduction

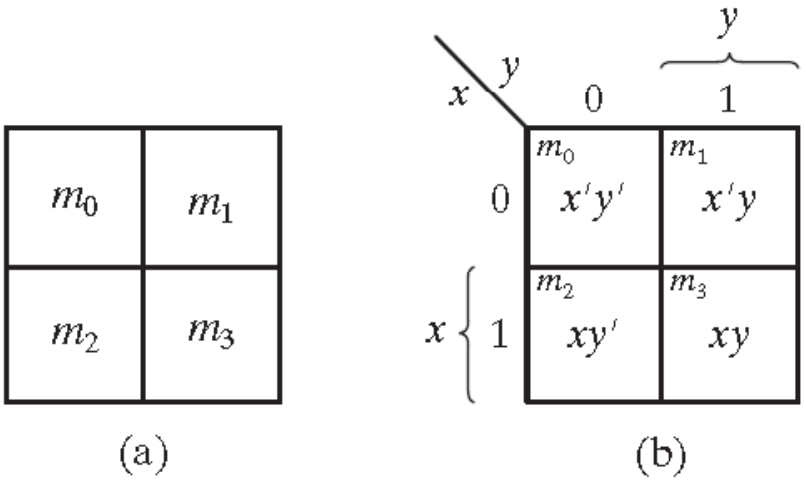
- *Gate-level minimization*: a digital circuit design task of finding an optimal gate-level implementation of the Boolean functions.
- This task is well understood, but is difficult to execute by manual design methods when the logic has more than a few inputs (typically five inputs).
- Fortunately, computer-based *logic synthesis* tools can minimize a large set of Boolean equations efficiently and quickly.
- Nevertheless, it is important that a designer understands the underlying mathematical description and solution of the problem.
- This chapter will enable you to execute a manual design of simple circuits, preparing you for skilled use of modern design tools.
- Hardware description language (HDL), such as Verilog and VHDL, is used by modern design tools.

3.2 The Map Method

- When implementing a Boolean function: the complexity of the digital logic gates is directly related to the complexity of its algebraic expression.
- Truth table representation is unique, but many equivalent algebraic expressions may exist.
- *Karnaugh map* (*K-map*): a simple and straightforward procedure for minimizing Boolean functions, which may be regarded as a pictorial form of a truth table.
- *K-map* is a diagram made up of squares, and each square represents one minterm.
- Boolean function
  - can be expressed as a sum of minterms
  - has simplified expression in sum of products (or product of sums)
  - exists the simplest algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term, which will produce a circuit with a minimum number of gates and the minimum number of inputs to each gate
  - may not have unique simplified expression

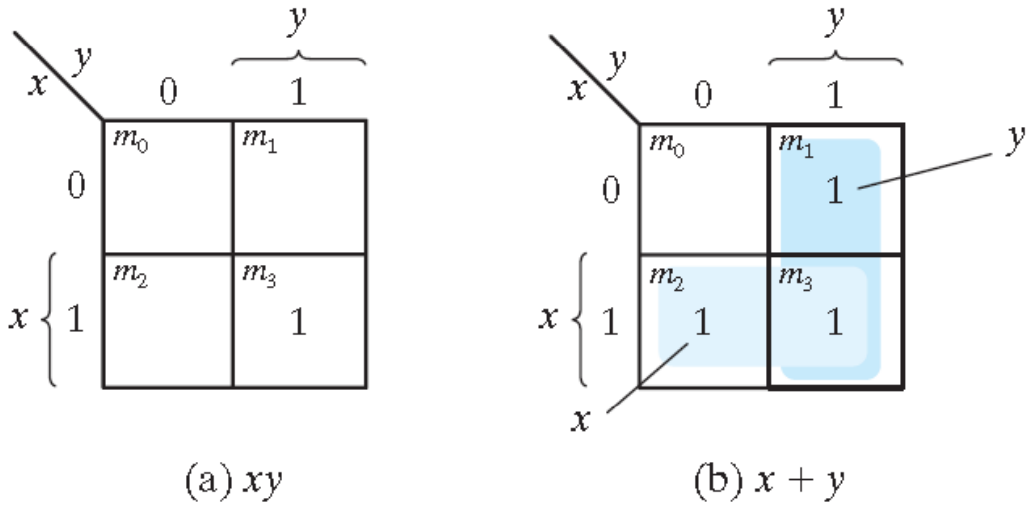
Two-Variable Map

- Two-variable map has four squares for four minterms.



- Representation of functions in the map

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$

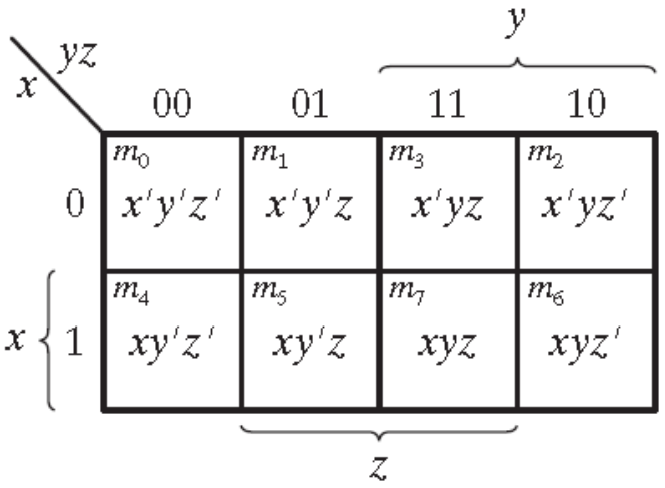


Three-Variable map

- Three variables function has eight minterms.
- Arranged in Gray code sequence
- Any two adjacent squares in the map differ by only one variable
  - primed in one square and unprimed in the other
  - e.g. m_5 and m_7 can be simplified
  - $m_5 + m_7 = xy'z + xyz = xz\,(y'+y) = xz$ *i.e.* $x = 1$ and $z = 1$.

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)

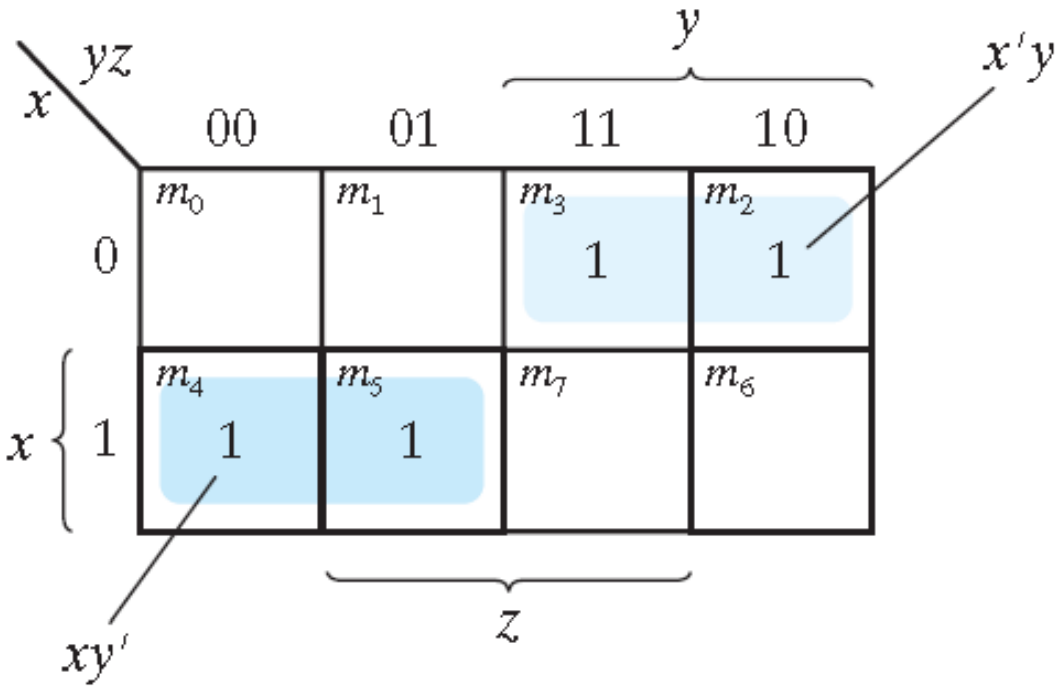


(b)

Example 3.1

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

$F = m_2+m_3+m_4+m_5 = x'yz' + x'yz + xy'z' + xy'z = x'y + xy'$



- m_0 and m_2 (m_4 and m_6) are adjacent
- $m_0 + m_2 = x'y'z' + x'yz' = x'z'(y'+y) = x'z'$
- $m_4 + m_6 = xy'z' + xyz' = xz'(y'+y) = xz'$

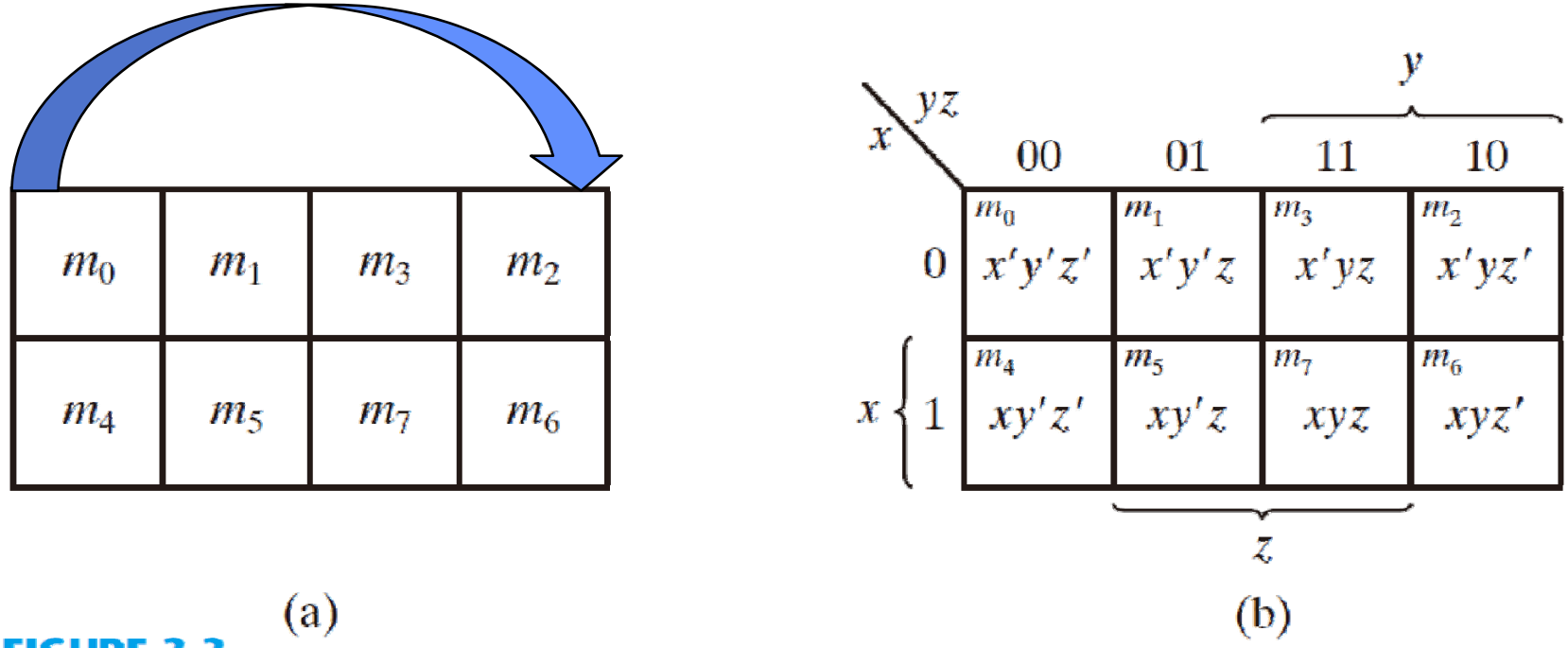
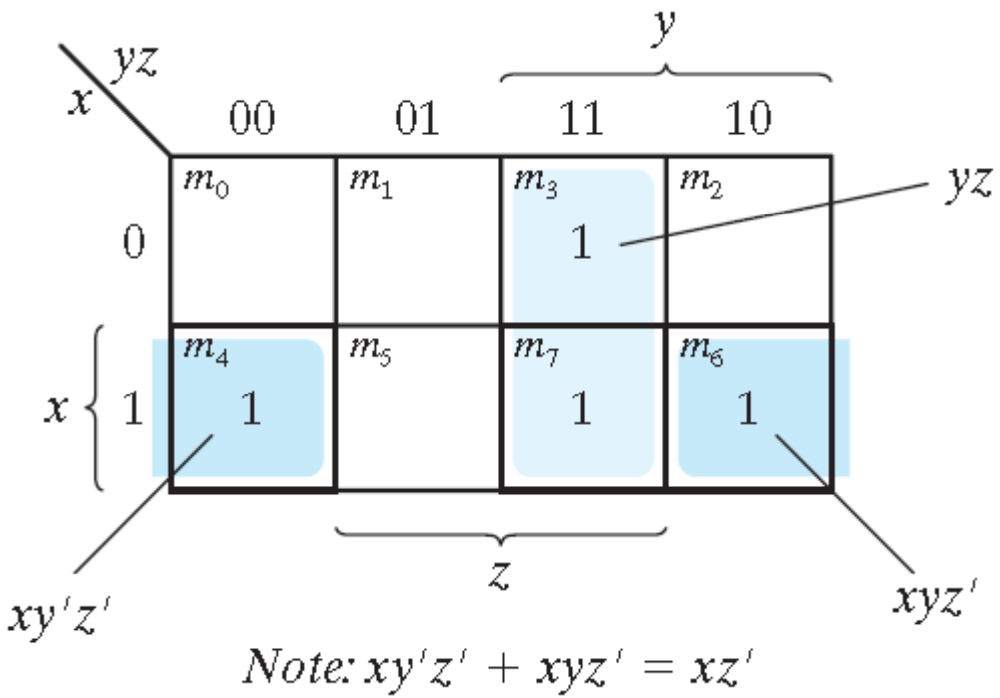


FIGURE 3.3
Three-variable K-map

Example 3-2

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

$F = m_3 + m_4 + m_6 + m_7 = x'yz + xy'z' + xyz' + xyz = xz' + yz$



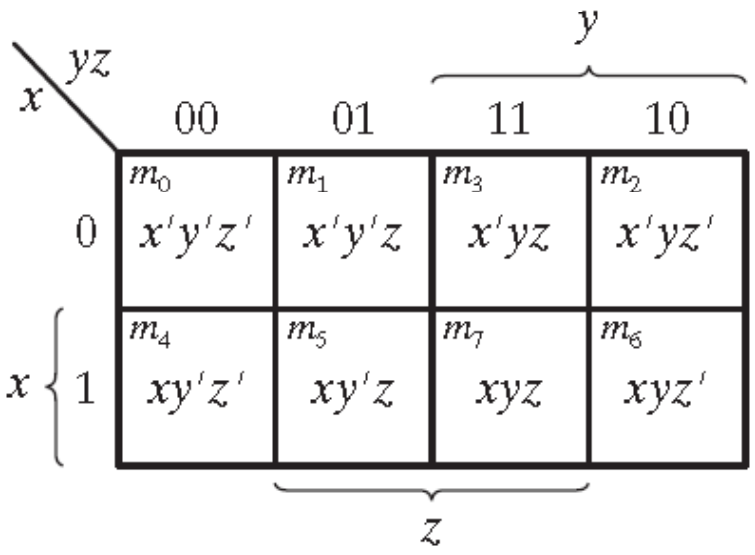
Note: $xy'z' + xyz' = xz'$

Four Adjacent Minterms

- Four adjacent minterms result in one literal expression.
- 2, 4, 8 and 16 squares are adjacent. (How many four adjacent squares? Ans: 6)
- $m_0+m_2+m_4+m_6 = x'y'z'+x'yz'+xy'z'+xyz' = x'z'(y'+y)+xz'(y'+y)$
  $= x'z' + xz' = z'$
- $m_1+m_3+m_5+m_7 = x'y'z+x'yz+xy'z+xyz = x'z(y'+y) + xz(y'+y)$
  $= x'z + xz = z$

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)

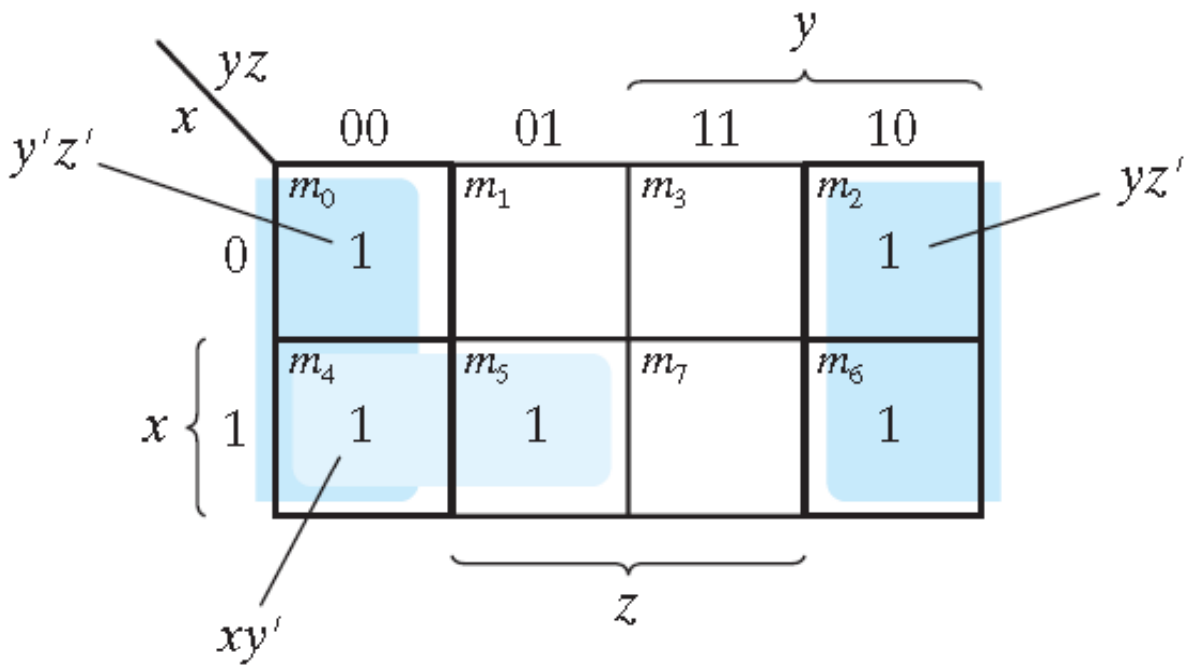


(b)

Example 3-3

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

$F = z' + xy'$



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

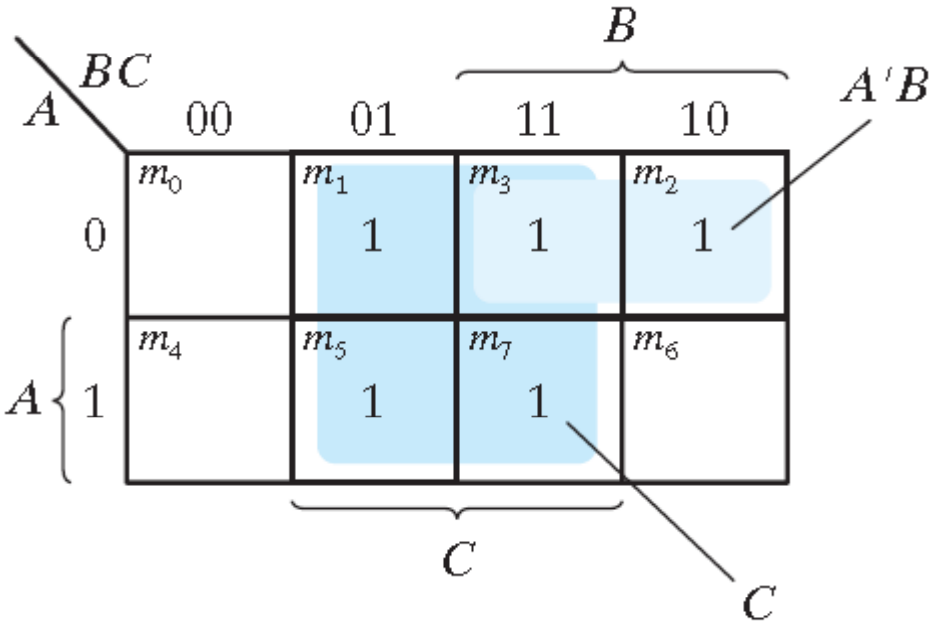
Example 3.4

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.

$F(A, B, C) = \sum(1, 2, 3, 5, 7) = C + A'B$



one-literal ➔ 4 squares

two-literal ➔ 2 squares

three-literal ➔ 1 square

$A'C$ ➔ m_1+m_3

$A'B$ ➔ m_2+m_3

$AB'C$ ➔ m_5

BC ➔ m_3+m_7

$m_1+m_3+ m_5+m_7$ ➔ C

m_2+m_3 ➔ $A'B$

3-3 Four-Variable Map

- Four variables (w,x,y,z) function has 16 minterms and squares
- Combinations of 2, 4, 8, and 16 adjacent squares
- 1 square ➔ 4-literal, 2 squares ➔ 3-literal,
  4 squares ➔ 2-literal, 8 squares ➔ 1-literal

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 \ 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} $wx'yz'$

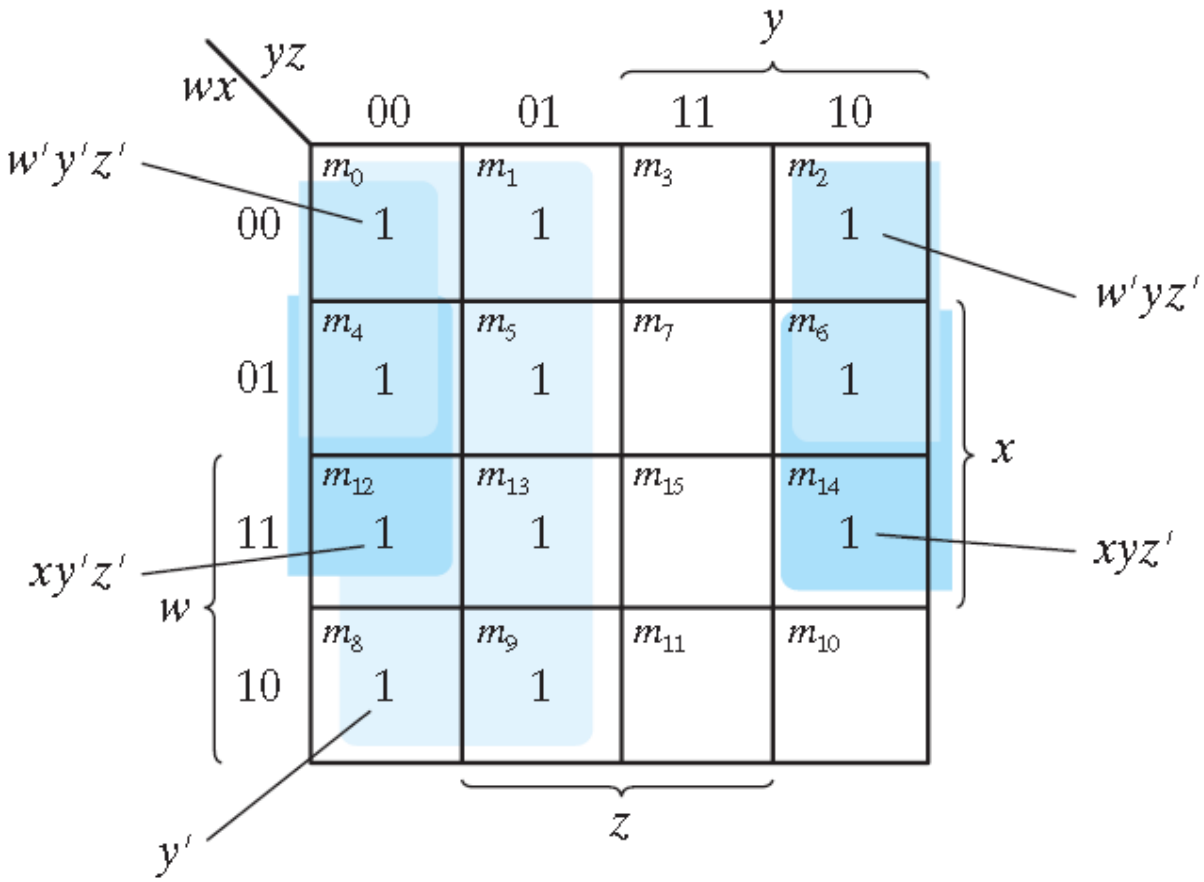
y (over 11, 10); x (rows 01, 11, 10); w (rows 11, 10); z (columns 01, 11)

(b)

Example 3-5

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

$F = y' + w'z' + xz'$



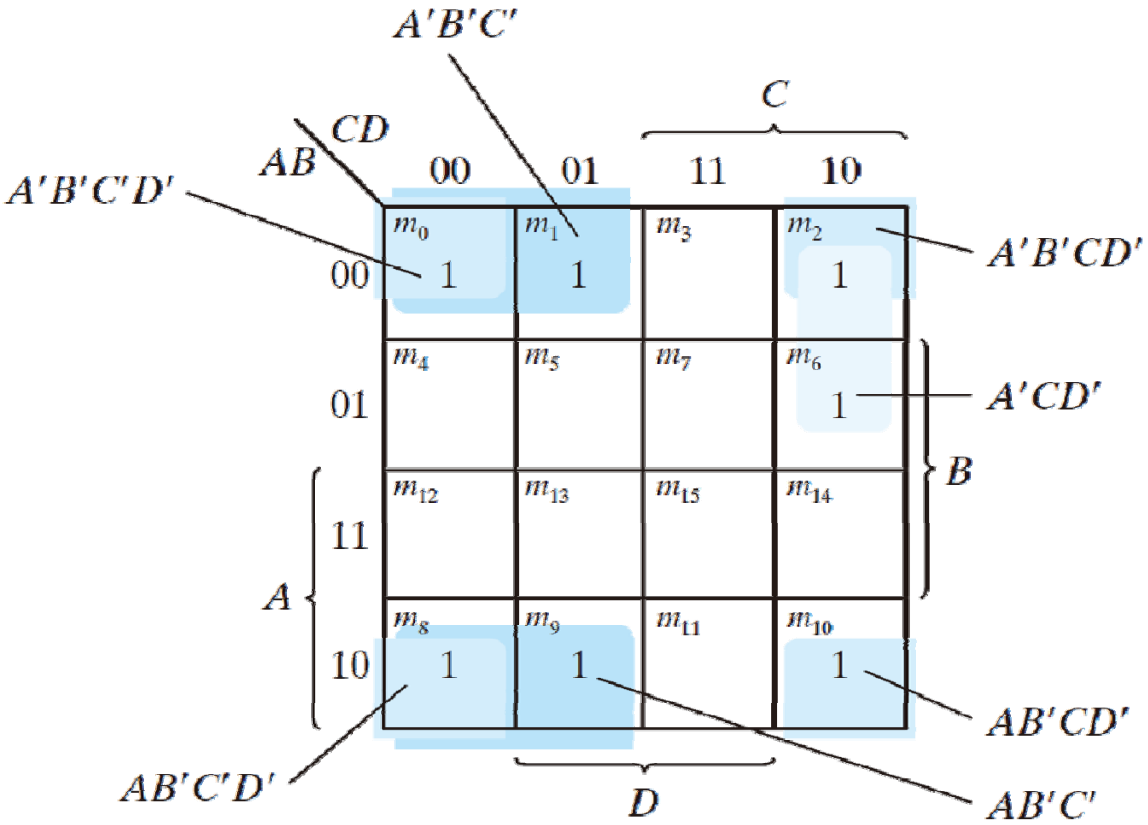
Note: $w'y'z' + w'yz' = w'z'$

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

Example 3-6

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

$F = B'D' + B'C' + A'CD'$ $\quad m_0+m_1 \quad m_2+m_{10} \quad m_6 \quad m_8+m_9$



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

Prime Implicants

- Choosing adjacent squares in K-map, we must ensure that

  (1) all the minterms of the function are covered,

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

  (3) there are no redundant terms.
- To develop a systematic procedure for combining squares in the map, two special types of terms are interested.
- ***Prime implicant***: 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***.
- The prime implicants can be obtained by combining all possible maximum numbers of squares.

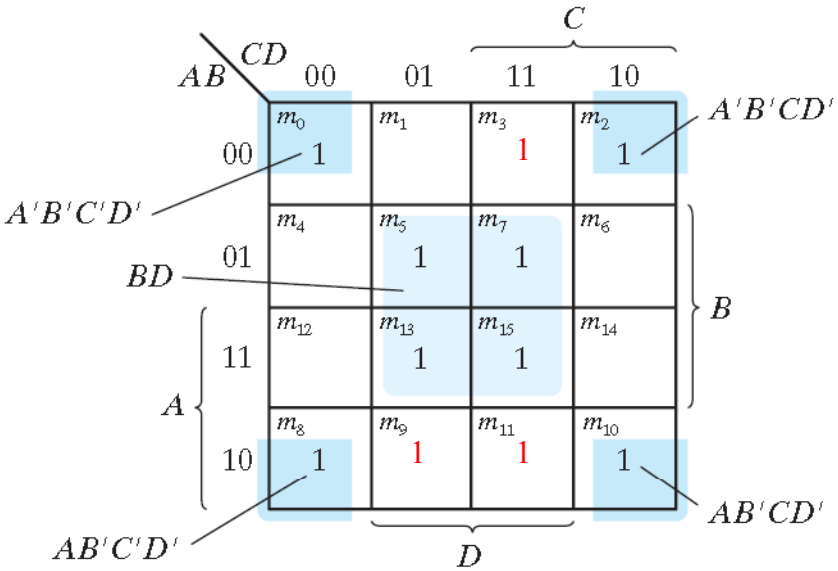
- Consider the following four-variable Boolean function:

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

- There are four possible expressions with four product terms of two literals each:

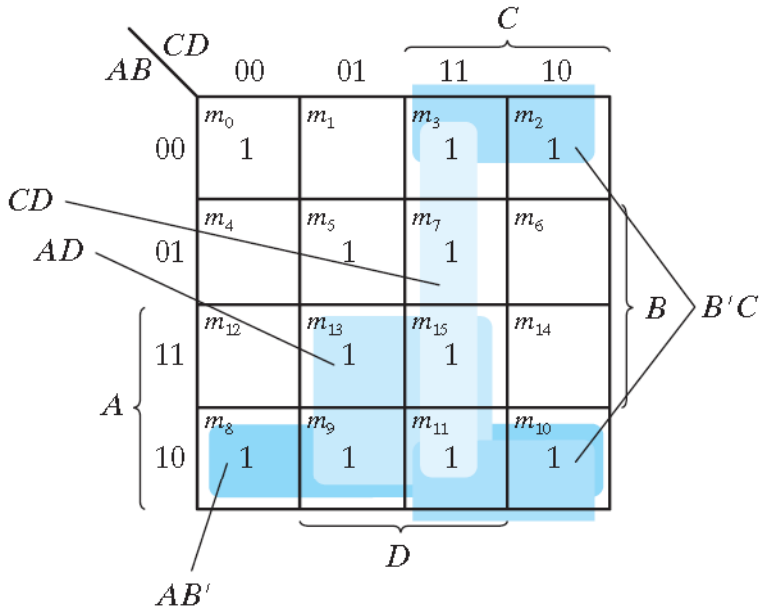
$$F = BD + B'D' + CD + AD$$
$$= BD + B'D' + CD + AB'$$
$$= BD + B'D' + B'C + AD$$
$$= BD + B'D' + B'C + AB'$$

m_3, m_9, and m_{11}



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

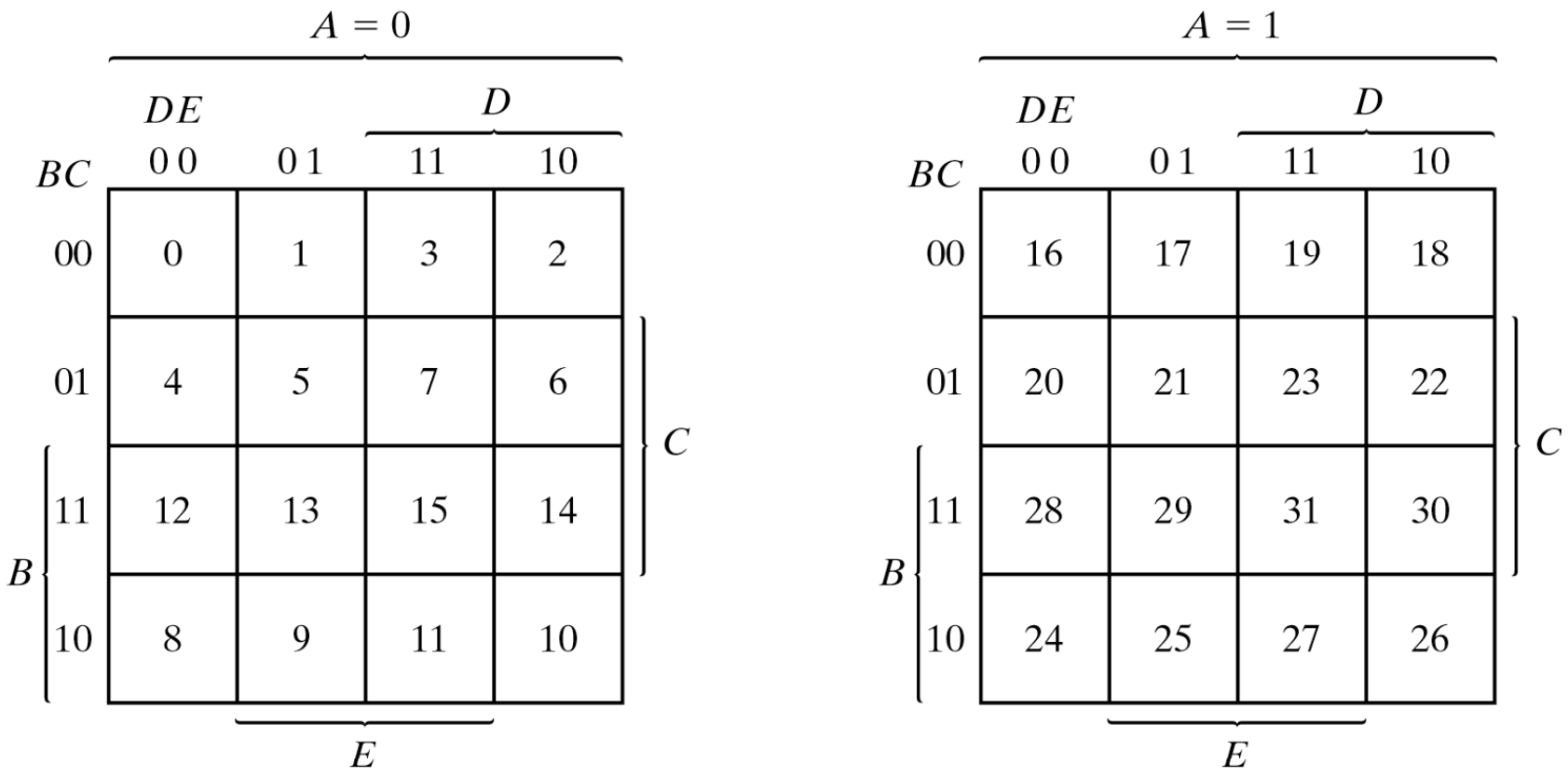
(a) Essential prime implicants BD and $B'D'$



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

Five-Variable Map

- Map for more than four variables becomes complicated
  - five-variable map: two four-variable map (one on the top of the other)



Example of 5-Variable Map

$F = \Sigma(0,2,4,6,9,13,21,23,25,29,31)$

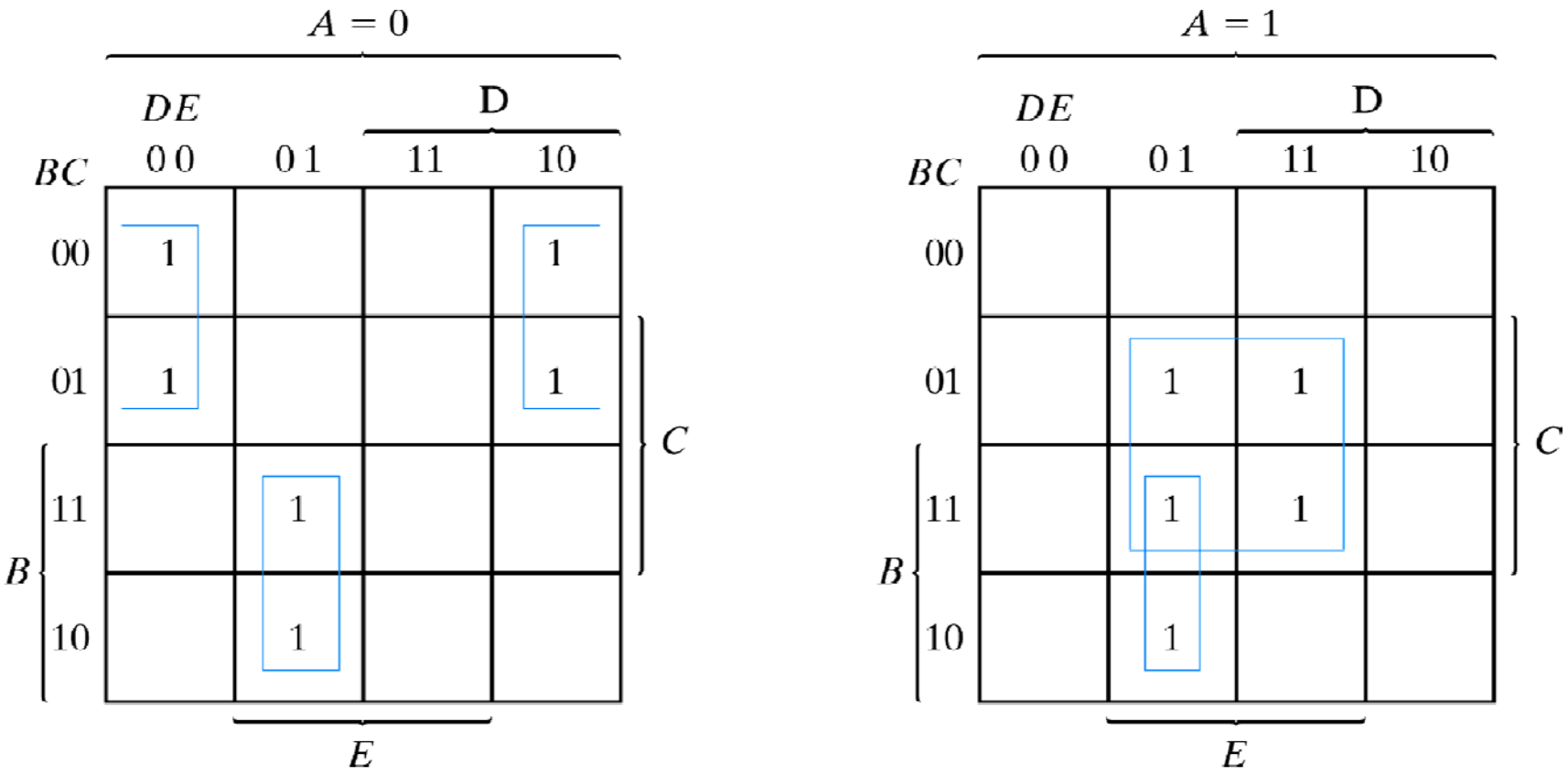
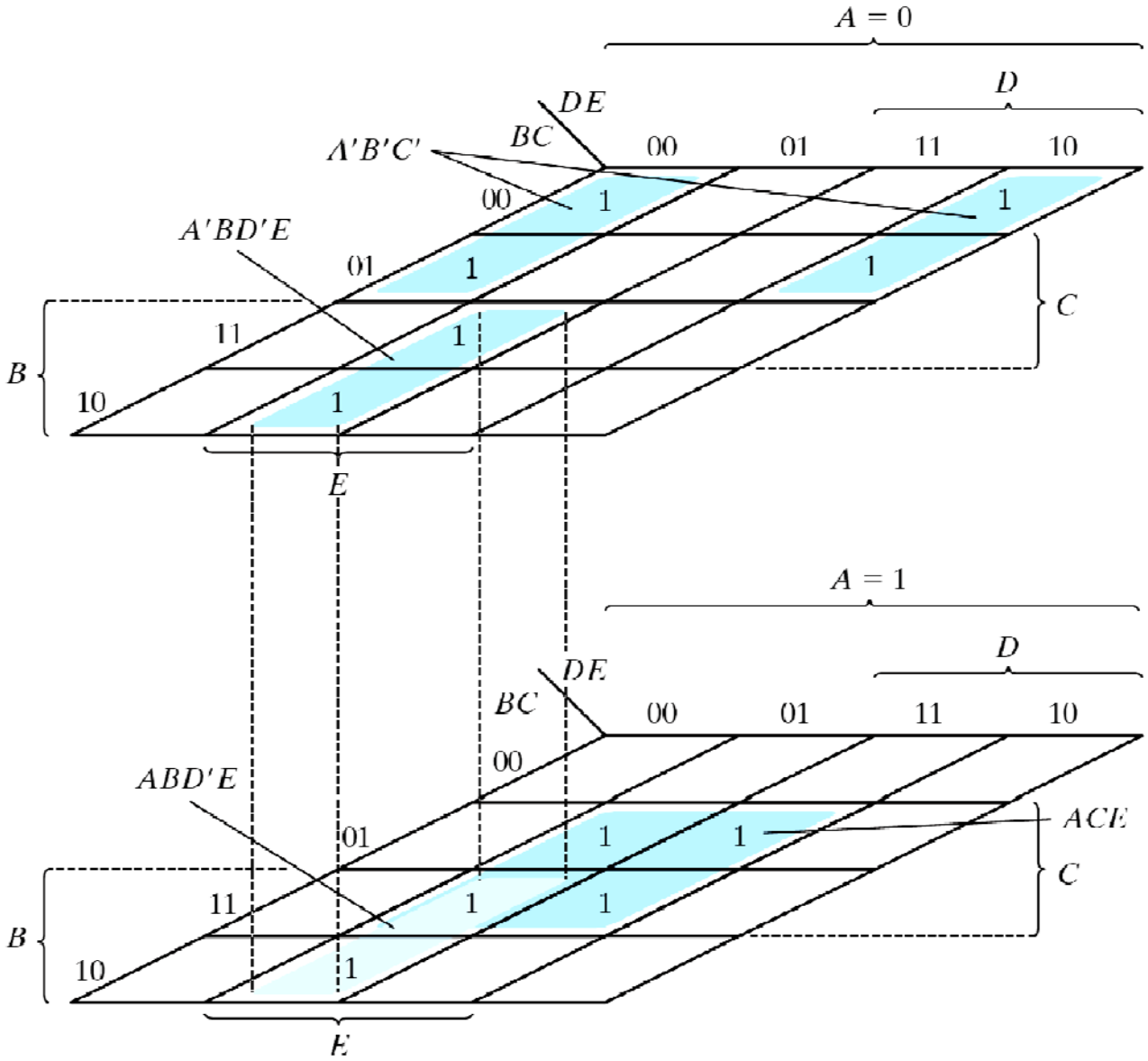


Fig. 3-13 Map for Example 3-7; $F = A'B'E' + BD'E + ACE$

Another Viewpoint of 5-Variable Map



3-4 Product of Sums Simplification

- Approach #1
  - Simplified F' in the form of sum of products
  - Apply DeMorgan's theorem $F = (F')'$
  - F': sum of products => F: product of sums
- Approach #2: duality
  - combinations of maxterms (it was minterms)
  - $M_0M_1 = (A+B+C+D)(A+B+C+D') = (A+B+C)+(DD') = A+B+C$

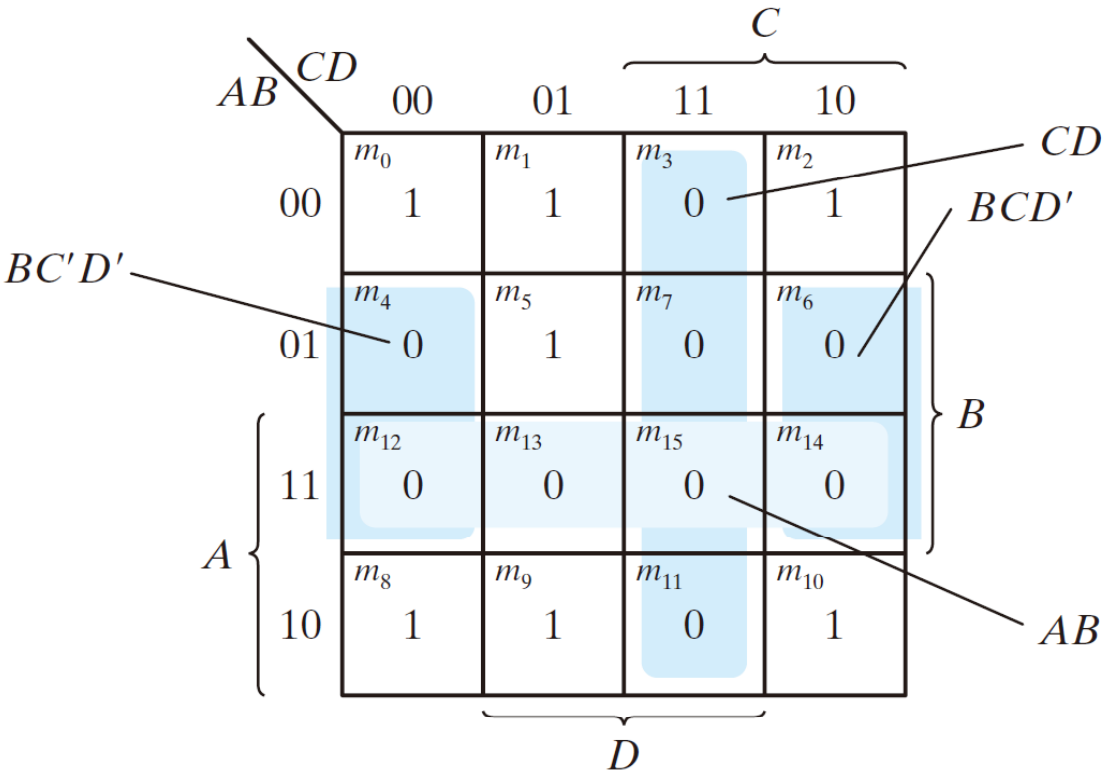
AB \ CD	00	01	11	10
00	M_0	M_1	M_3	M_2
01	M_4	M_5	M_7	M_6
11	M_{12}	M_{13}	M_{15}	M_{14}
10	M_8	M_9	M_{11}	M_{10}

Example 3-7

Simplify the Boolean function $F = \Sigma(0,1,2,5,8,9,10)$ into (a) sum-of-products form and (b) product-of-sums form.

$F' = AB+CD+BD'$ (adjacent squares 0 in the map)

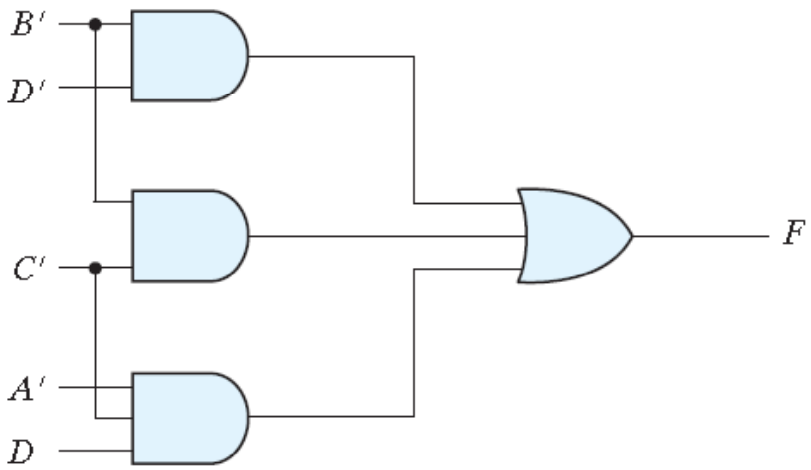
Apply DeMorgan's theorem; $F = (A'+B')(C'+D')(B'+D)$



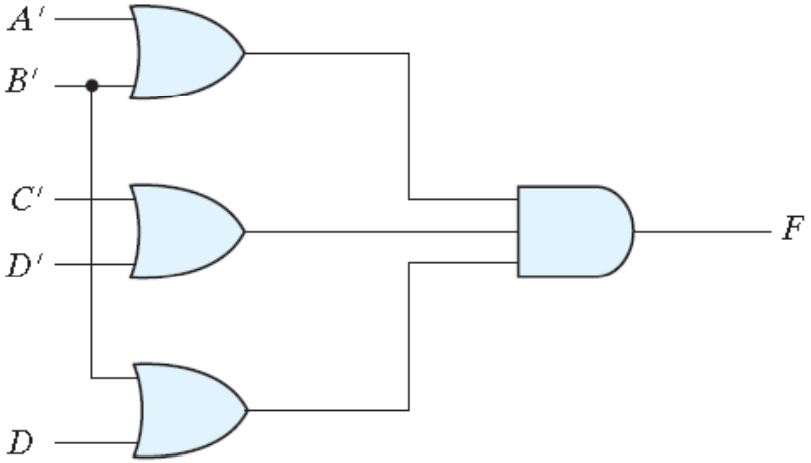
Note: $BC'D' + BCD' = BD'$

Two-Level Implementation of Example 3-7

- Two-level gate implementation: sum of products, and product of sums



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



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

Sum-of-Maxterm

- The 1's of the function represent the minterms and the 0's represent the maxterms.
- For Table 3.1

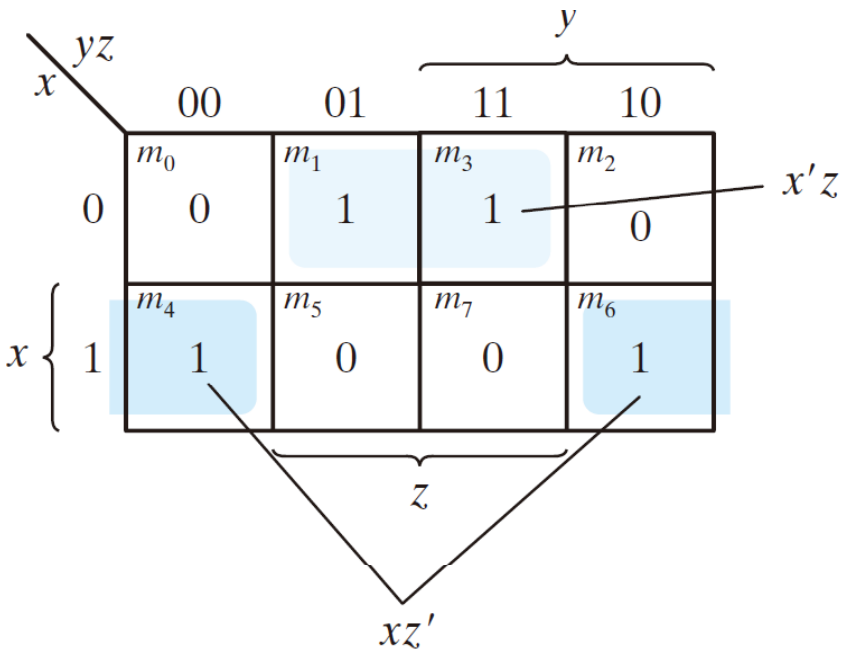
  In sum-of-minterm: $F(x, y, z) = \sum(1, 3, 4, 6) = x'z + xz'$

  In sum-of-maxterm: $F'(x, y, z) = \sum(0, 2, 5, 7) = (x'+z')\,(x+z)$

  Taking the complement of F' ➔ $F(x, y, z) = (x'+z')\,(x+z) = x'z + xz'$

Table 3.1
Truth Table of Function F

x	y	z	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



3-5 Don't-Care Conditions

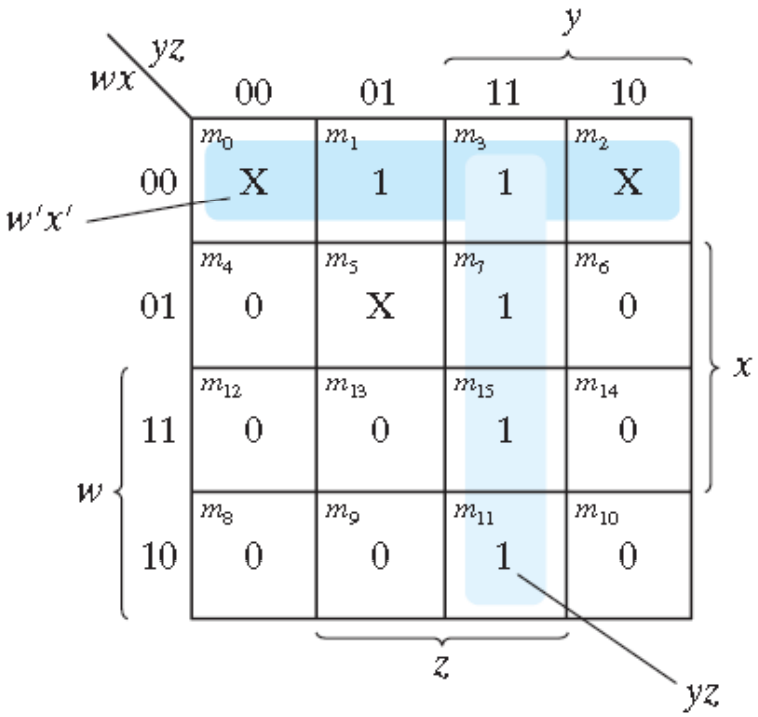
- In practice, functions with unspecified outputs for some input combinations are called *incompletely specified functions*. As an example, the BCD code has six combinations that are not used and considered to be unspecified.
- The unspecified minterms of a function is called *don't-care conditions*, because we simply don't care what value is assumed to the unspecified minterms.
- These don't-care conditions with symbol 'X' can be used on a map to provide further simplification of the Boolean expression.
- In choosing adjacent squares to simplify the function, the don't-care minterms may be assumed to be either 0 or 1.

Example 3.8

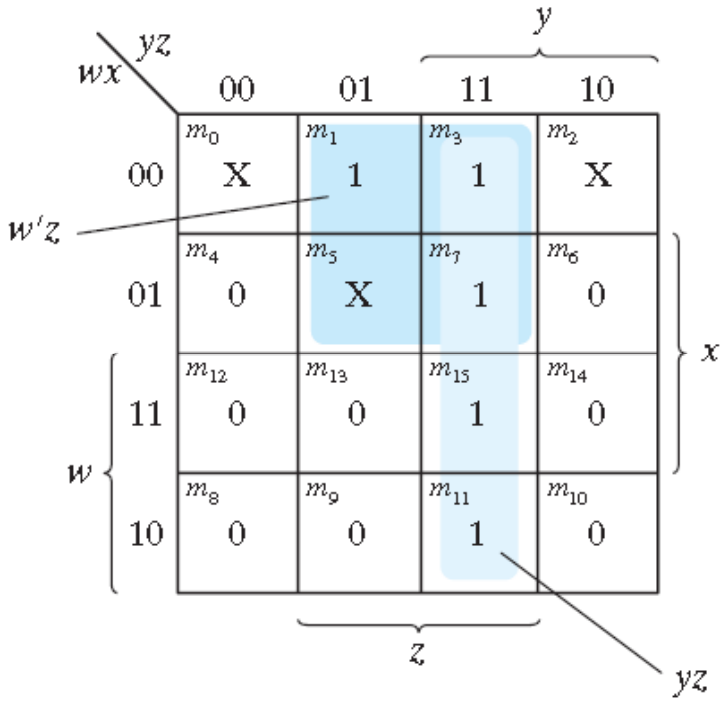
Simplify the Boolean function $F(w, x, y, z) = (1, 3, 7, 11, 15)$ which has the don't-care conditions $d(w, x, y, z) = (0, 2, 5)$.

Without using X, $F = yz + w'x'z$

Using X, (a) $F = yz + w'x'$ (b) $F = yz + w'z$



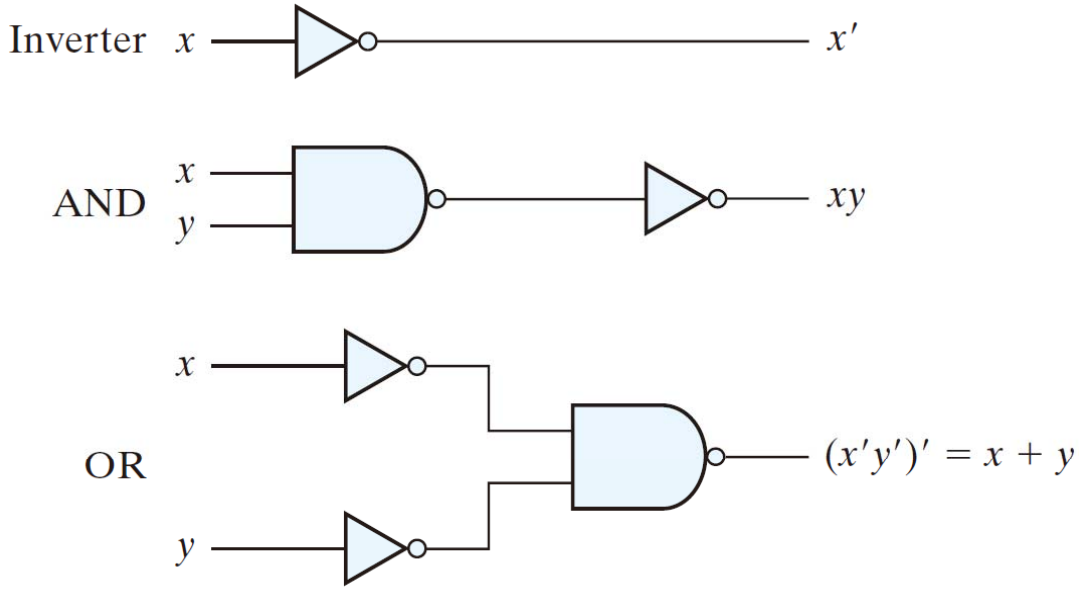
(a) $F = yz + w'x'$



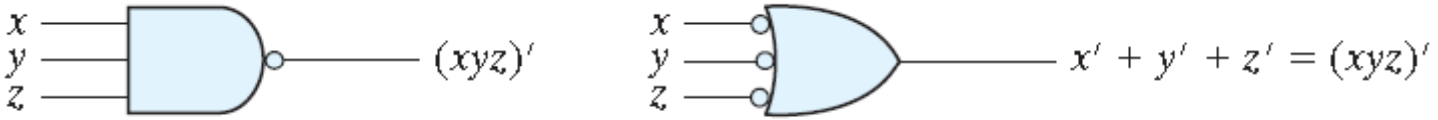
(b) $F = yz + w'z$

3-6 NAND and NOR Implementation

- NAND or NOR are preferred as the basic gates used in IC design.
- AND, OR, NOT, and other gates are translated to NAND and NOR gates.
- NAND gate is a universal gate, any function can be implemented with NAND.



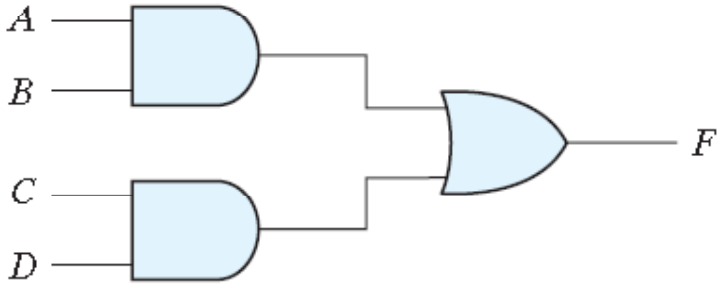
- Two graphic symbols for a NAND gate



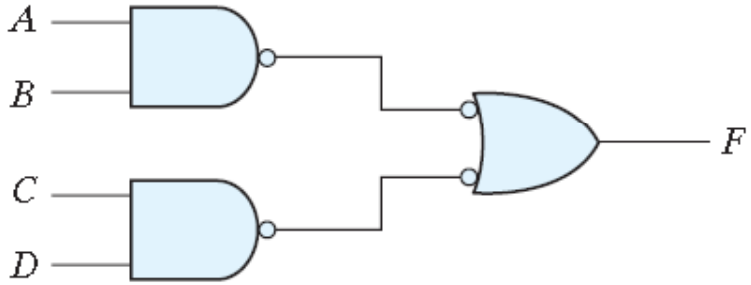
(a) AND-invert (b) Invert-OR

Two-level Implementation: NAND-NAND

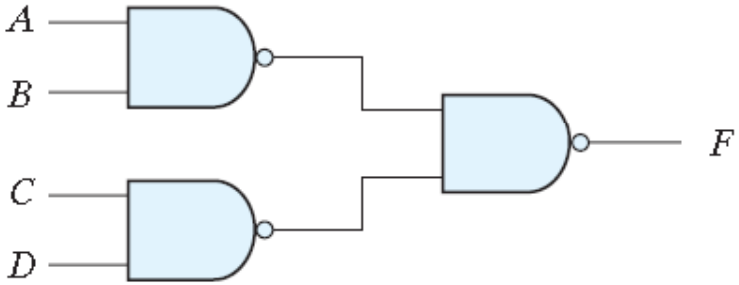
- The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.
- Two-level logic: NAND-NAND ⇔ sum of products
- Example: $F = AB + CD = ((AB)'(CD)')'$



(a)



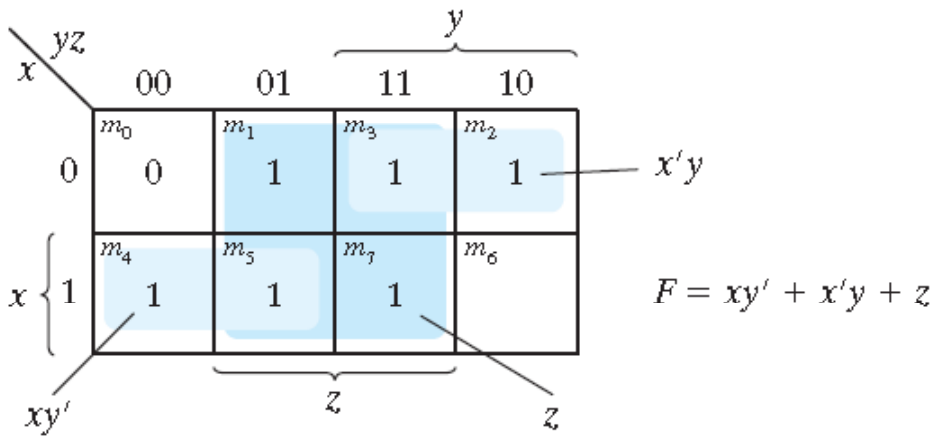
(b)



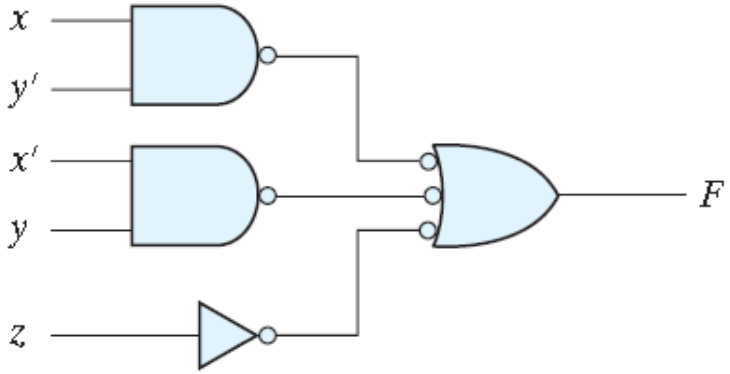
(c)

Example 3-9

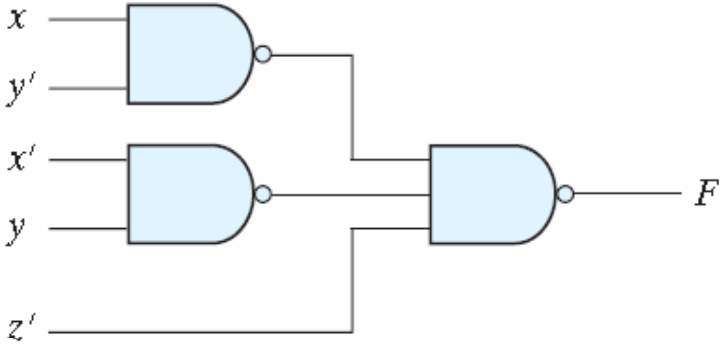
- Implement the following Boolean function with NAND gates:
  $F(x, y, z) = \sum(1, 2, 3, 4, 5, 7)$
- Fill the map and simplify, we get $F = xy' + x'y + z$



(a)



(b)



(c)

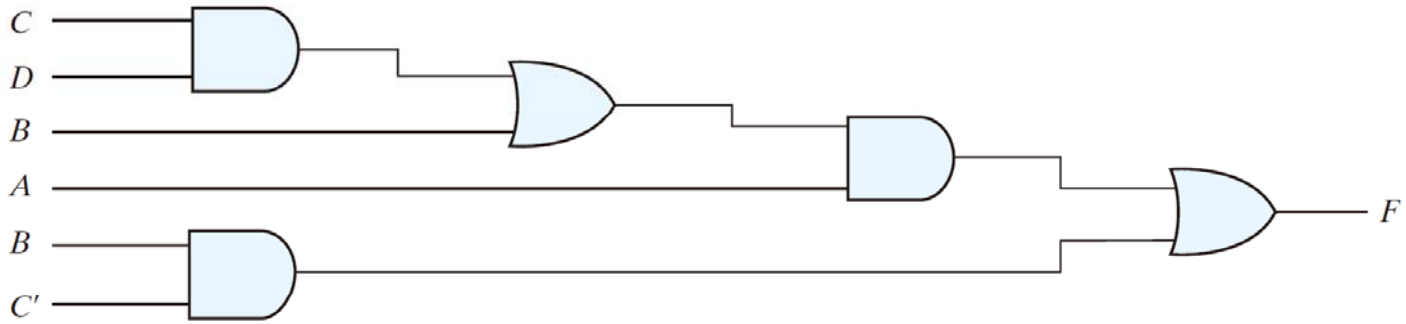
Two-level Implementation Procedure

- The standard form of expressing Boolean functions results in a two-level implementation.

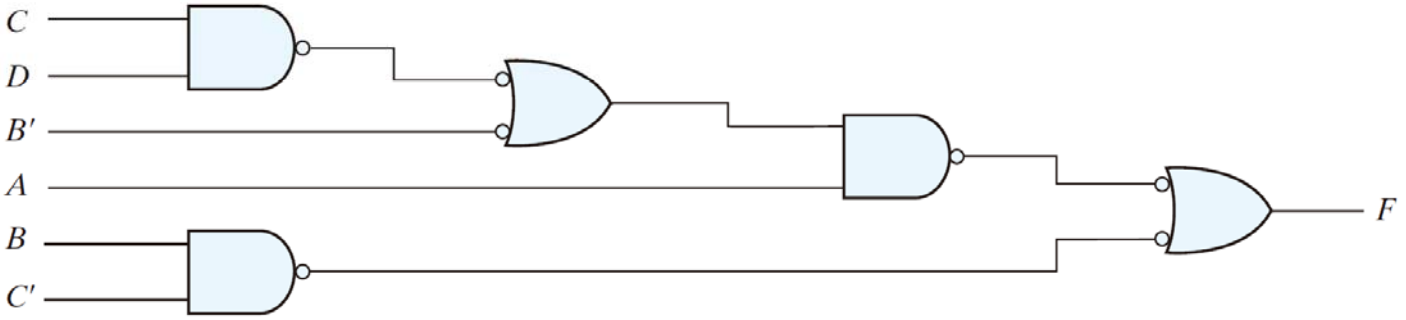
1. Simplify the function and express it in sum-of-products form.
2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.
3. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate.

Multilevel NAND Circuits

- Boolean function implementation of three or more levels (multi-level)
- Express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit.
- Example: $F = A\,(CD + B) + BC'$



(a) AND–OR gates

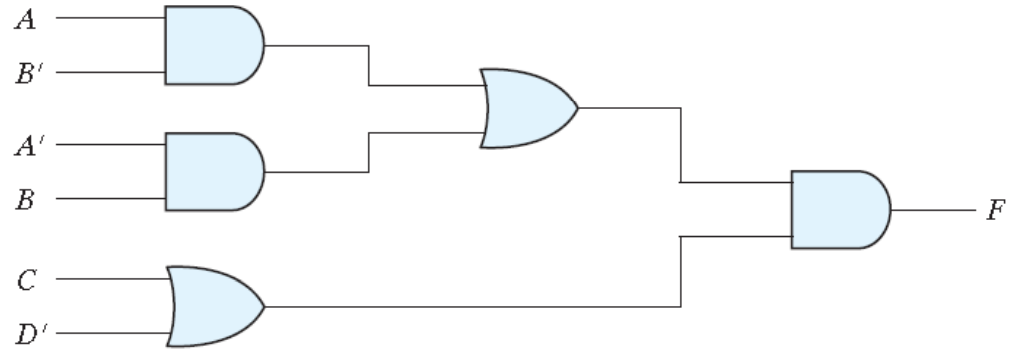


(b) NAND gates

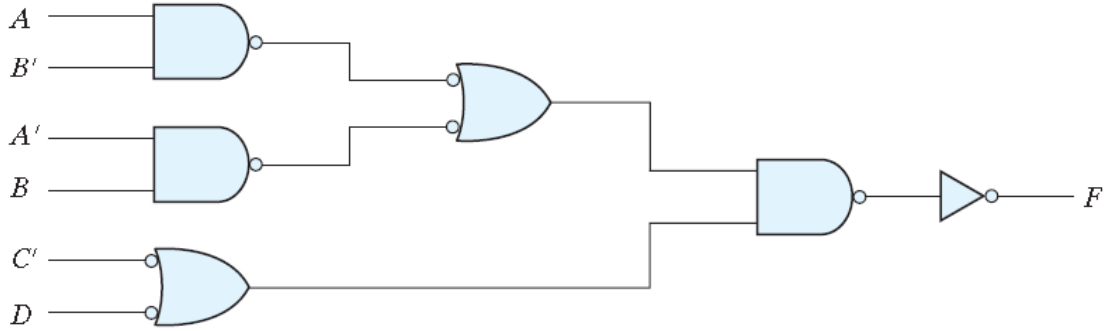
Multilevel AND-OR ➔ All-NAND

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.

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



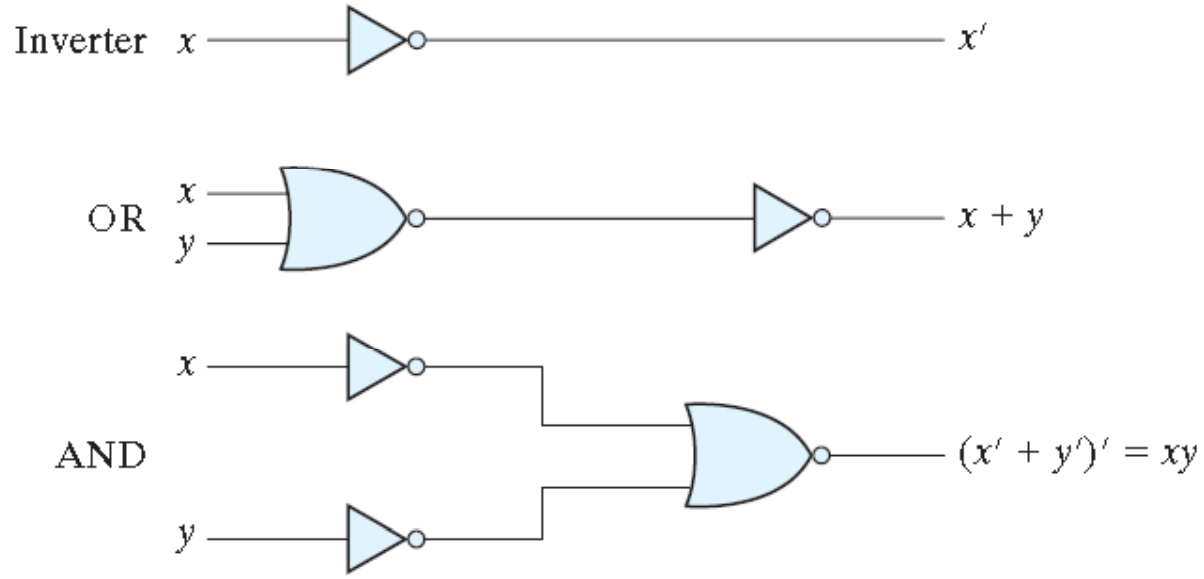
(a) AND–OR gates



(b) NAND gates

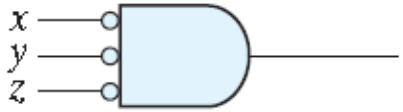
NOR Implementation

- NOR function is the dual of NAND function
- The NOR gate is also universal



- Two graphical symbols

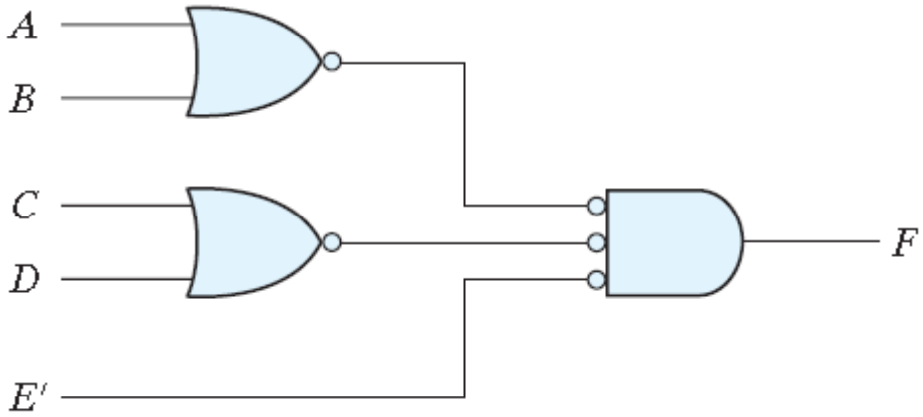
(a) OR-invert



(b) Invert-AND

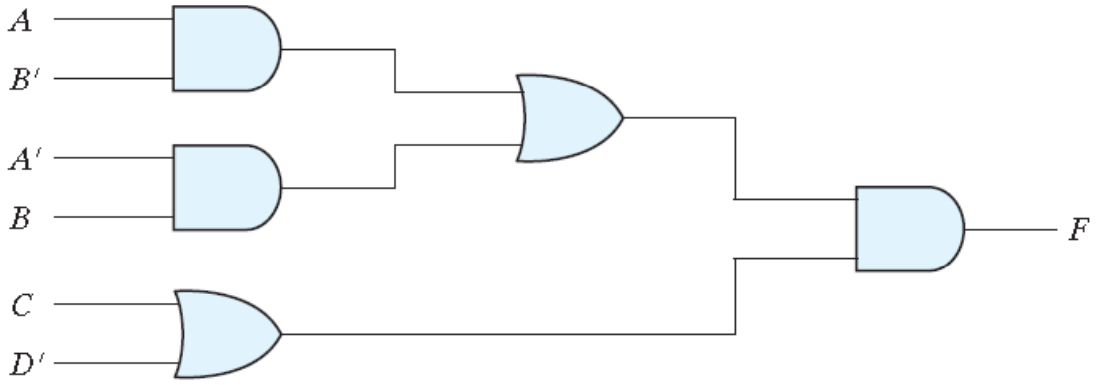
Two-level Implementation: NOR-NOR

- A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form.
- A product-of-sums expression is implemented with OR–AND circuit.
- NOR-NOR: changing the OR gates to NOR gates with OR-invert graphic symbols and the AND gate to a NOR gate with an invert-AND graphic symbol.
- Example: $F = (A + B)(C + D)E$

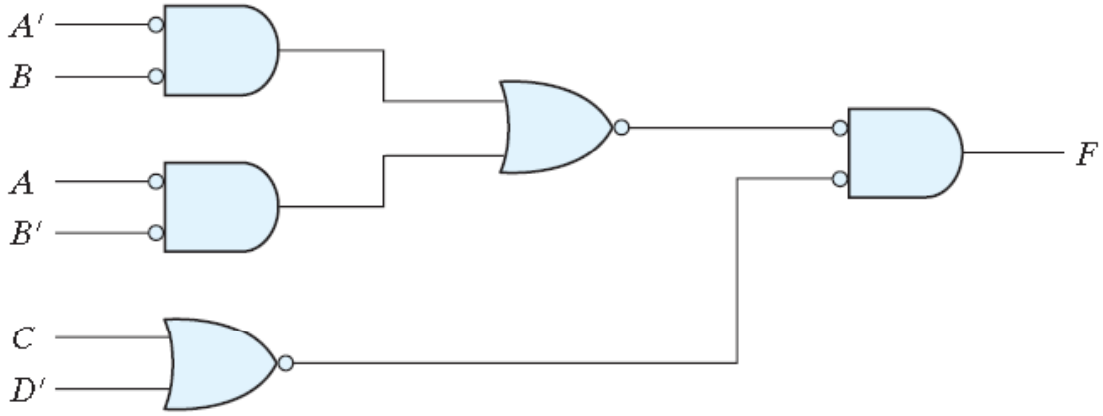


Transform AND-OR to All-NOR

- Convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol.
- Example: $F = (AB' + A'B)(C + D')$

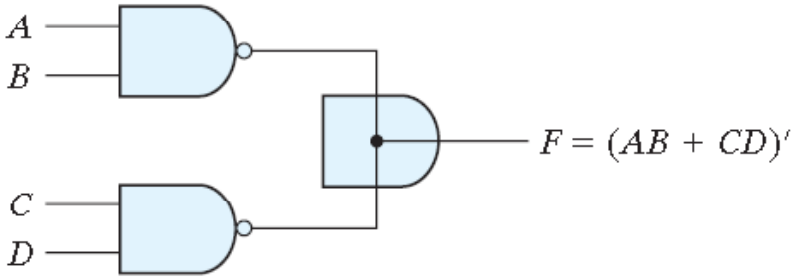


(a) AND–OR gates



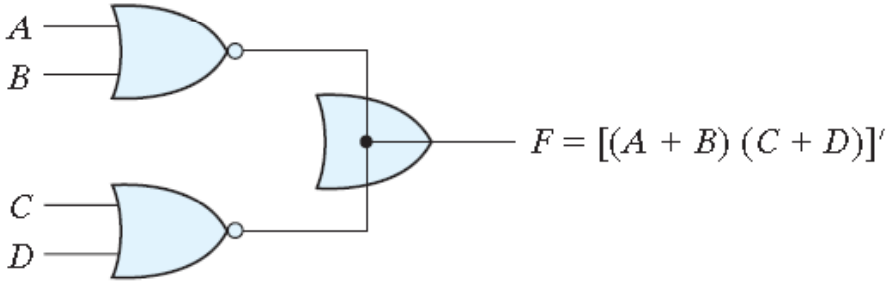
3-7 Other Two-level Implementations

- Wired-AND: when two open-collector TTL NAND gates ties together, AND function, not a physical gate, is formed.
- $F = (AB)'\,(CD)' = (AB + CD)' = (A' + B')(C' + D')$
- Also called an AND–OR–INVERT (AOI) function.



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

(AND–OR–INVERT)



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

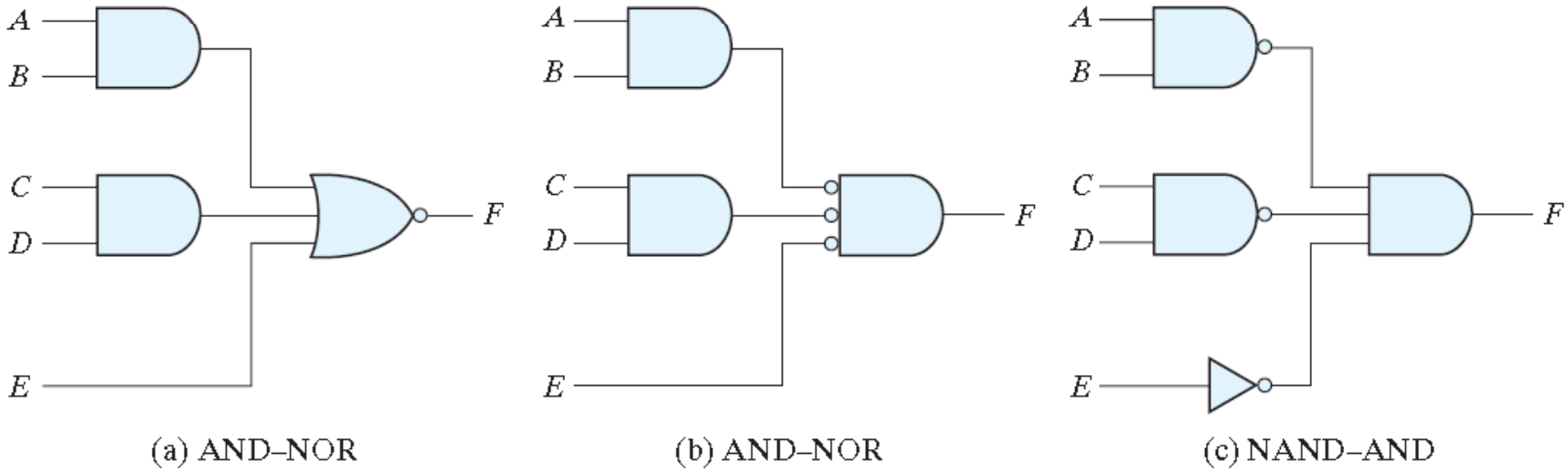
- Similarly, the NOR outputs of ECL gates can be tied together to perform a wired-OR function.
- $F = (A+B)'+(C+D)' = [(A+B)(C+D)]'$
- Also called an OR–AND–INVERT (OAI) function.

Nondegenerate Forms

- There are 16 possible combinations of two-level forms
  - Eight of them are degenerate forms = a single operation
  - The eight nondegenerate forms are
    - AND-OR, OR-AND, NAND-NAND, NOR-NOR, NOR-OR, NAND-AND, OR-AND, AND-OR
    - AND-OR and NAND-NAND = sum of products
    - OR-AND and NOR-NOR = product of sums
    - NOR-OR, NAND-AND, OR-AND, AND-OR = ?

AND-OR-Invert Implementation

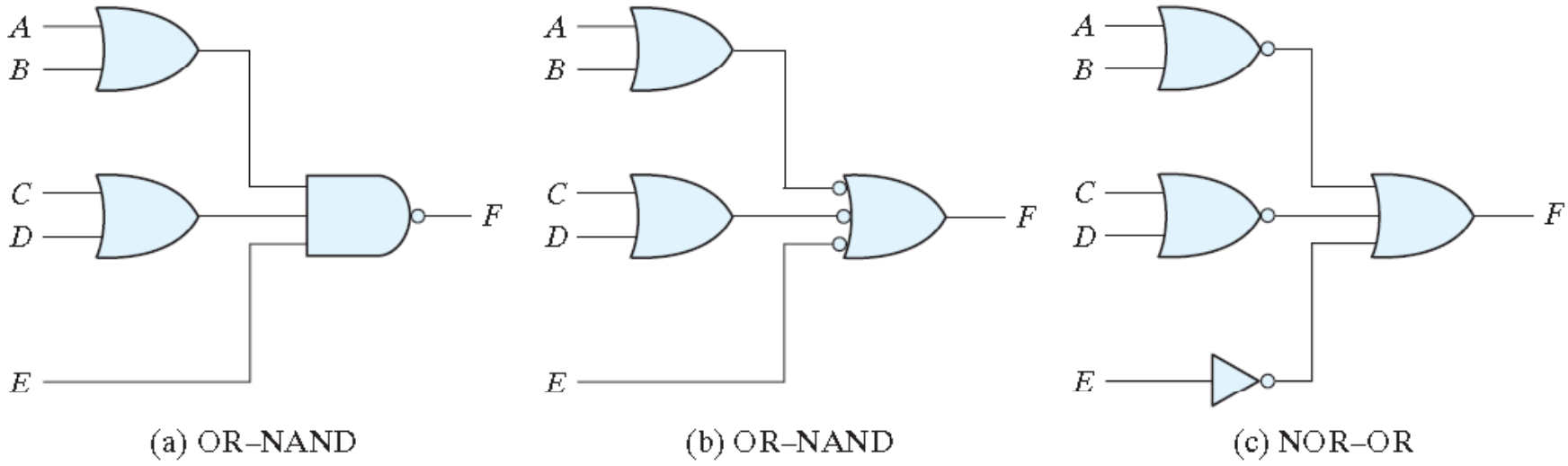
- NAND-AND = AND-NOR = AOI
- Example: $F = (AB + CD + E)'$



(a) AND–NOR (b) AND–NOR (c) NAND–AND

OR-AND-INVERT (OAI) Implementation

- OR-NAND = NOR-OR = OAI
- Example: $F = ((A + B)(C + D)E)'$



(a) OR–NAND (b) OR–NAND (c) NOR–OR

Tabular Summary

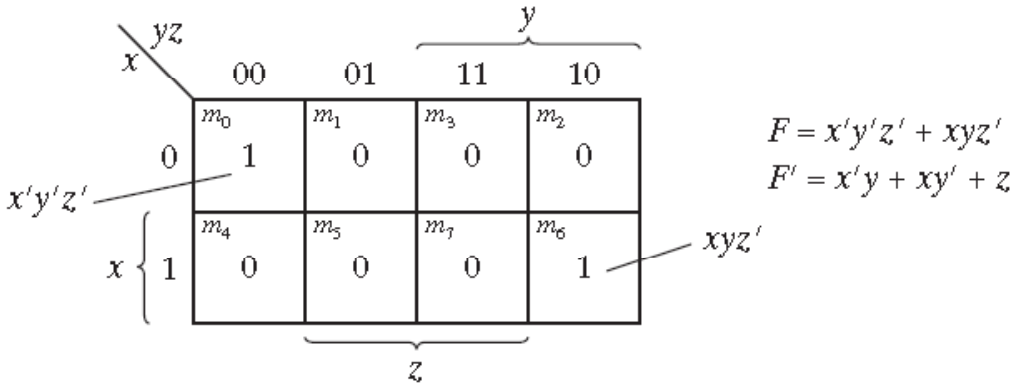
Table 3.2
Implementation with Other Two-Level Forms

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 3-10

Implement the function of Fig. (a) with the four 2-level forms listed in Table 3.2 .

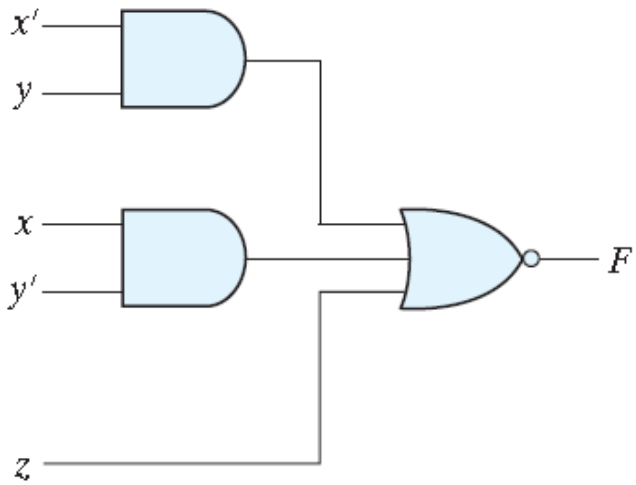


(a) Map simplification in sum of products

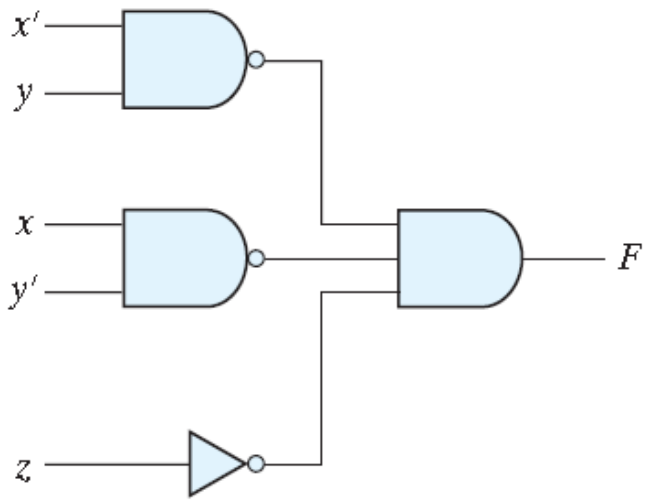
- $F' = x'y + xy' + z$ (F': sum of products; 0's)
- $F = (x'y + xy' + z)'$ (F: AOI implementation)

- $F = x'y'z' + xyz'$ (F: sum of products; 1's)
- $F' = (x + y + z)(x' + y' + z)$ (F': product of sums)
- $F = ((x + y + z)(x' + y' + z))'$ (F: OAI)

Other Two-level Implementations

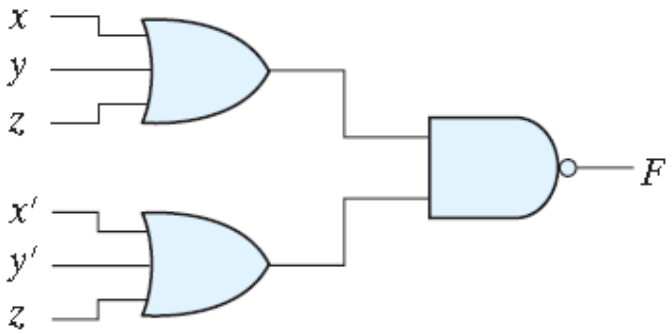


AND–NOR

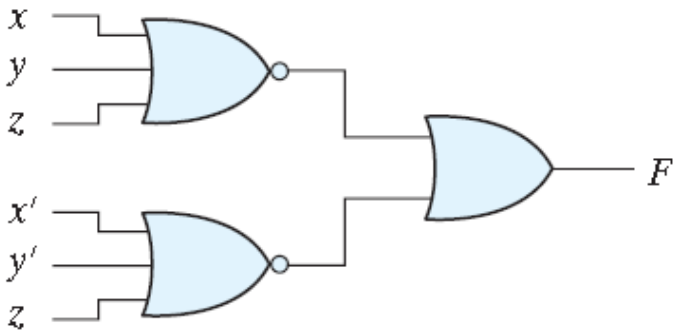


NAND–AND

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



OR–NAND



NOR–OR

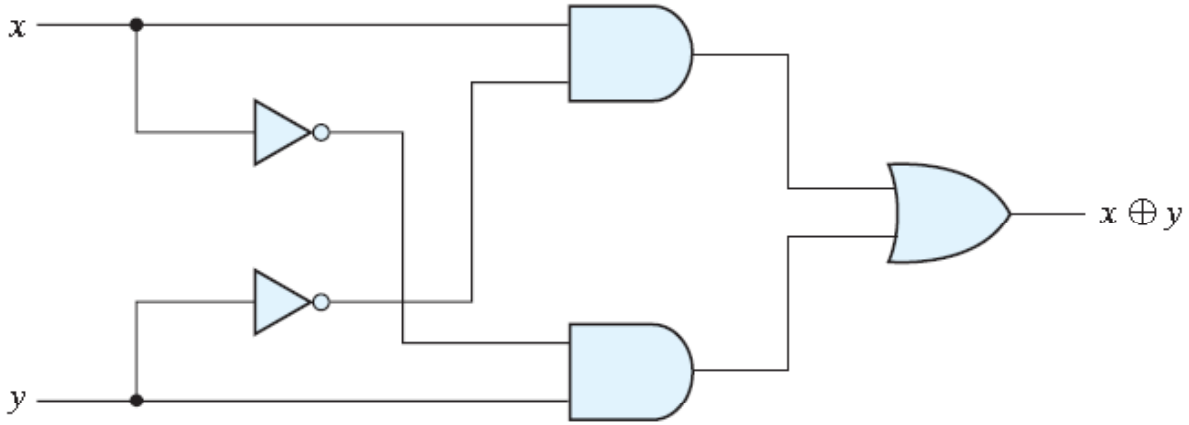
(c) $F = [(x + y + z)(x' + y' + z)]'$

3-8 Exclusive-OR Function

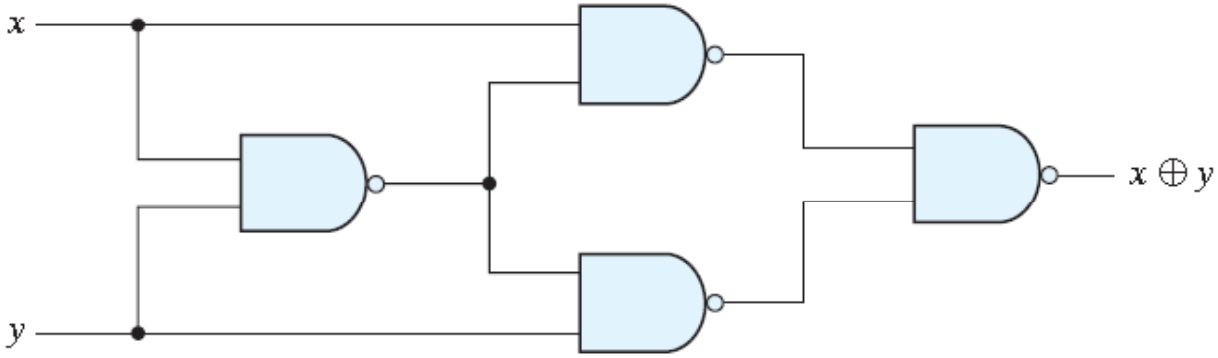
- Exclusive-OR (XOR)
  - $x \oplus y = xy' + x'y$
- Exclusive-NOR (XNOR) (some uses symbol $\odot$)
  - $(x \oplus y)' = xy + x'y'$
- Some identities
  - $x \oplus 0 = x$
  - $x \oplus 1 = x'$
  - $x \oplus x = 0$
  - $x \oplus x' = 1$
  - $x \oplus y' = (x \oplus y)'$
  - $x' \oplus y = (x \oplus y)'$
- Commutative and associative
  - $A \oplus B = B \oplus A$
  - $(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$

XOR Implementation

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



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

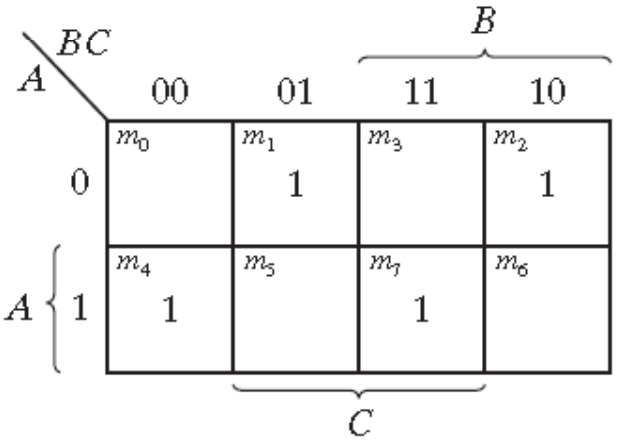


(b) Exclusive-OR with NAND gates

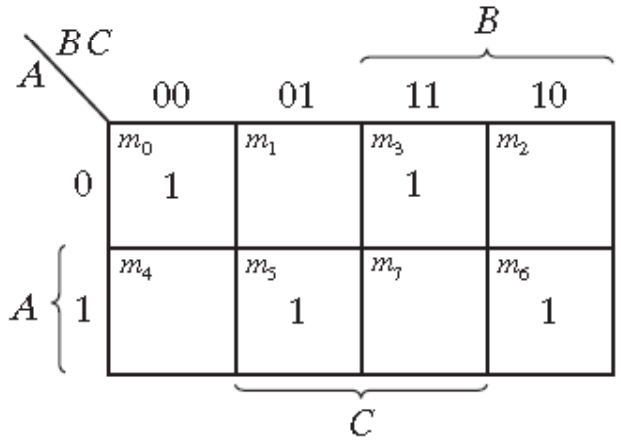
Odd Function

- $A \oplus B \oplus C = (AB' + A'B)C' + (AB + A'B')C = AB'C' + A'BC' + ABC + A'B'C$

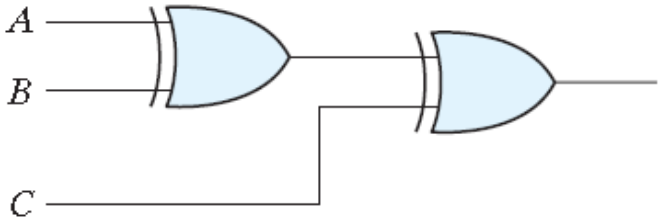
  $= \Sigma(1,2,4,7)$
- Odd number of 1's
- The complement is a even function.



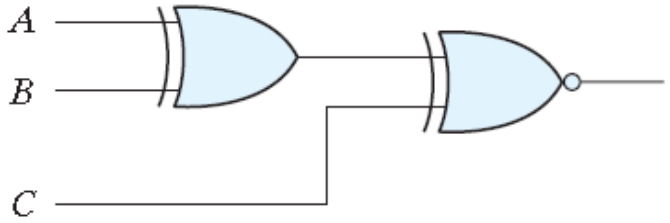
(a) Odd function $F = A \oplus B \oplus C$



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



(a) 3-input odd function



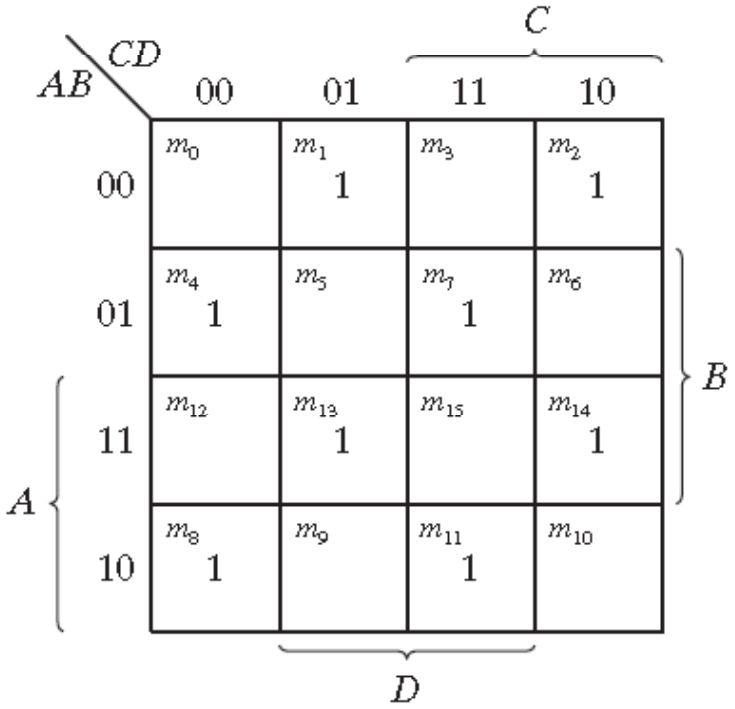
(b) 3-input even function

Four-variable Exclusive-OR

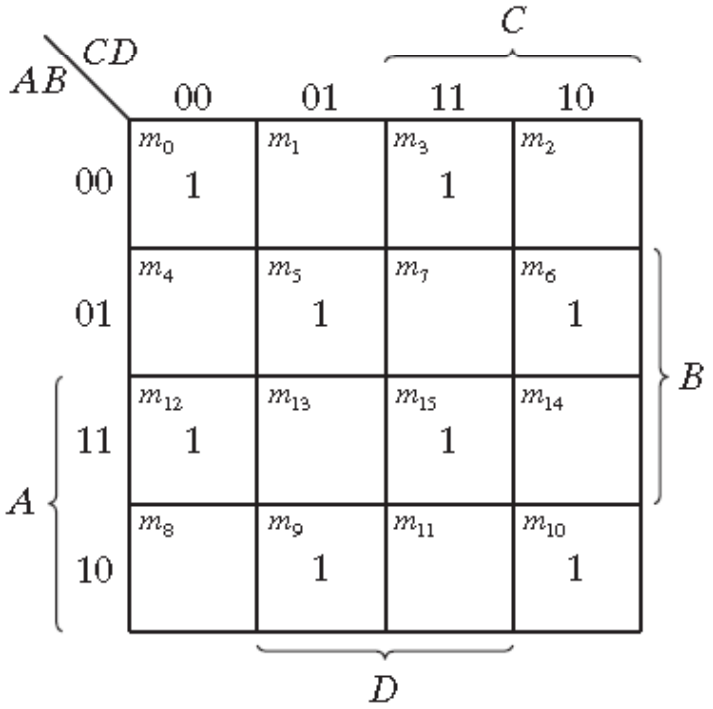
- $A \oplus B \oplus C \oplus D = (AB' + A'B) \oplus (CD' + C'D)$

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

$$= \sum(1, 2, 4, 7, 8, 11, 13, 14)$$



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



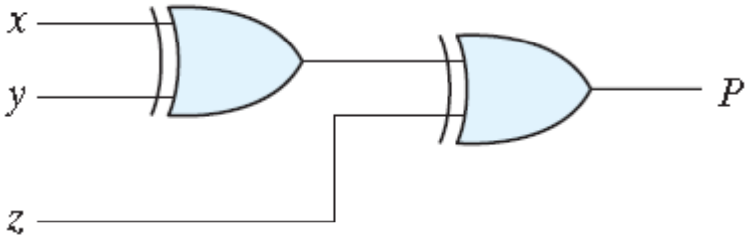
(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

Parity Generation

- XOR is used in error detection and correction codes; a parity bit is an extra bit included with a binary message to make the number of 1's either odd or even.
- Parity bit: $P = x \oplus y \oplus z$, is used for error detection

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



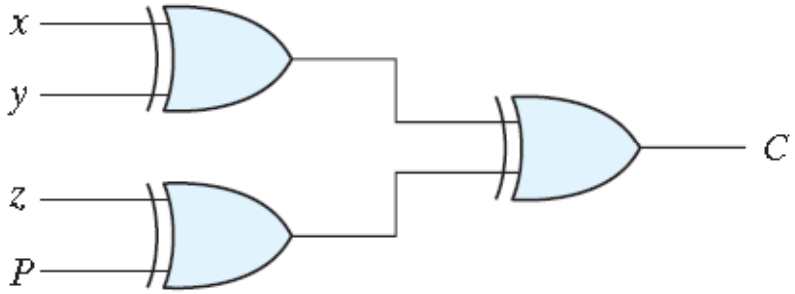
(a) 3-bit even parity generator

Parity Checking

- Parity check: $C = x \oplus y \oplus z \oplus P$
  - $C = 1$: an odd number of data bit error
  - $C = 0$: correct or an ever # of data bit error

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



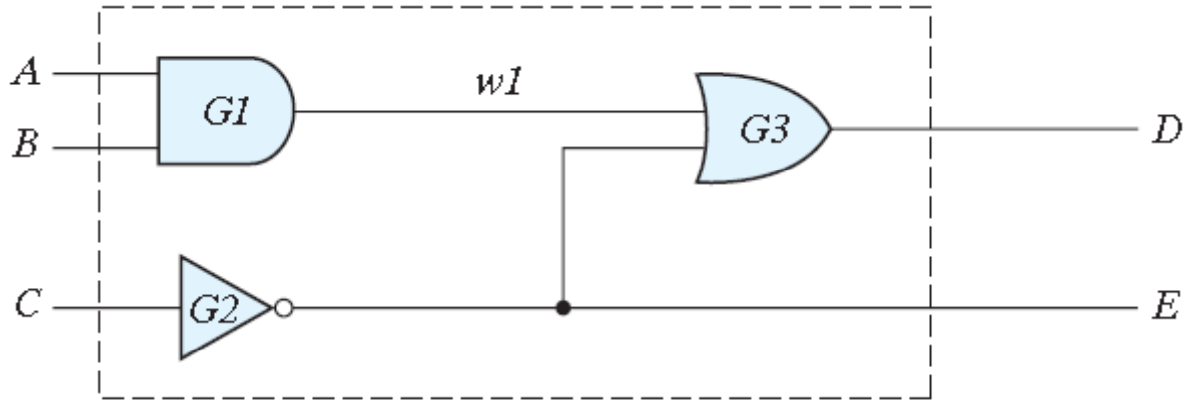
(b) 4-bit even parity checker

3.9 Hardware Description Language (HDL)

- Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language (HDL) to describe, design, and test a circuit in software before it is ever manufactured.
- Industry favored standards: VHDL and Verilog HDL
- A computer-based language that describes the hardware of digital systems in a textual form.
- As a *documentation language*, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.
- HDLs are used in several major steps in the design flow of an integrated circuit: *design entry*, *functional simulation* or *verification*, *logic synthesis*, *timing verification*, and *fault simulation*.

- *Design entry* creates an HDL-based description of the functionality that is to be implemented in hardware.
- *Logic simulation* displays the behavior of a digital system through the use of a computer.
- The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench.*
- *Logic synthesis* is the process of deriving a list of physical components and their interconnections (called a *netlist*) from the model of a digital system described in an HDL.
- *Timing verification* confirms that the fabricated, integrated circuit will operate at a specified speed.
- In VLSI circuit design, *fault simulation* compares the behavior of an ideal circuit with the behavior of a circuit that contains a process-induced flaw.

Example



HDL Example 3.2 (Gate-Level Model with Propagation Delays)

```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
  output D, E;
  input  A, B, C;
  wire   w1;

  and              #(30) G1 (w1, A, B);
  not              #(10) G2 (E, C);
  or               #(20) G3 (D, w1, E);
endmodule
```

HDL Example 3.3 (Test Bench)

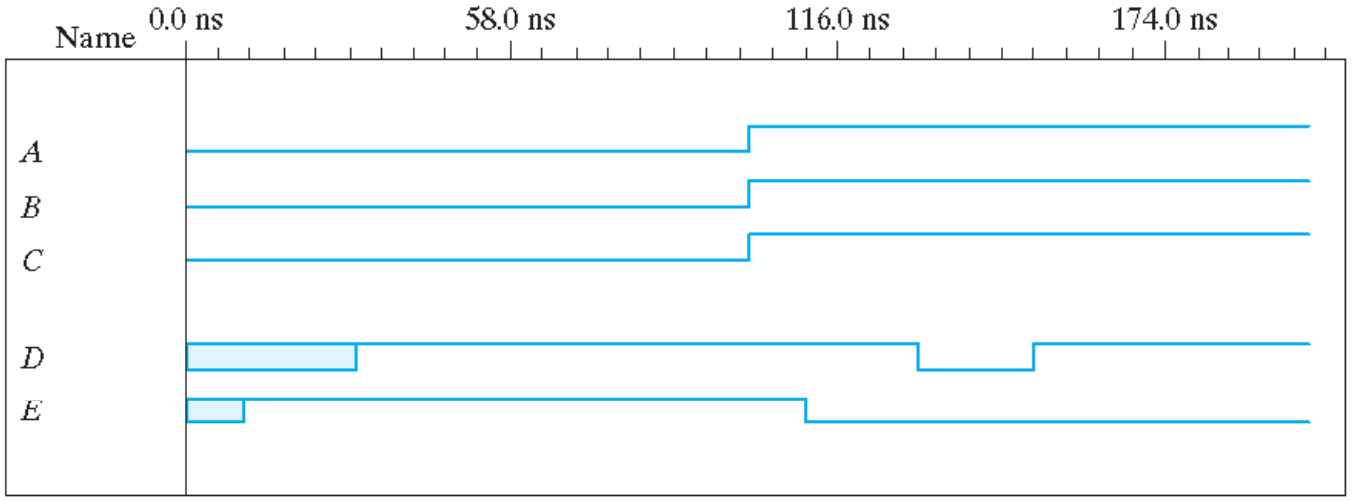
```
// Test bench for Simple_Circuit_prop_delay

module  t_Simple_Circuit_prop_delay;
  wire     D, E;
  reg      A, B, C;

Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

  initial #200 $finish;
endmodule
```



Homework #3

- 3.2 (c) (f)
- 3.6 (c) (d)
- 3.11
- 3.15 (c)
- 3.16 (a) (c)
- 3.21