Chapter 2 Combinational Logic Circuits

M. Morris Mano, Charles R. Kime. (2015). *Logic and computer design fundamentals* (5th ed.). Pearson.

Contents

1. Binary Logic and Gates

- Digital circuits are hardware components that manipulate binary information.
  - The circuits are implemented using transistors and interconnections in complex semi-conductor devices called *integrated circuits*.
- Each basic circuit is referred to as a *logic gate*.
  - For simplicity in design, we model the transistor-based electronic circuits as logic gates.
    - Thus, the designer need not be concerned with the internal electronics of the individual gates, but only with their external logic properties.
    - Each gate performs a specific logical operation.
    - The outputs of gates are applied to the inputs of other gates to form a digital circuit

Binary Logic

- Binary logic deals with binary variables, which take on two discrete values, and with the operations of mathematical logic applied to these variables.
- Logical operators operate on binary values and binary variables.
    - Basic logical operators are the logic functions AND, OR and NOT.

Binary Variables

- Two binary values have different names:
  - True/False
  - On/Off
  - Yes/No
  - 1/0
- We use 1 and 0 to denote the two values.
- Variable identifier examples:
  - A, B, C, X, Y, Z
  - RESET, START_IT, or ADD1

Logical Operations

- The three basic logical operations are:
  - AND : represented by a dot (.) or by the absence of an operator
    - $Z = X.Y$ or $Z = XY$ is read "Z is equal to X AND Y."
  - OR : represented by a plus symbol (+)
    - $Z = X + Y$ is read "Z is equal to X OR Y."
  - NOT : represented by a bar over the variable (¯), a single quote mark (') after, or (~) before the variable.
    - $Z = \overline{X}$ is read "Z is equal to NOT X"

Logical Operations

- AND operation

$0 . 0 = 0$

$0 . 1 = 0$

$1 . 0 = 0$

$1 . 1 = 1$

OR operation

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 1$

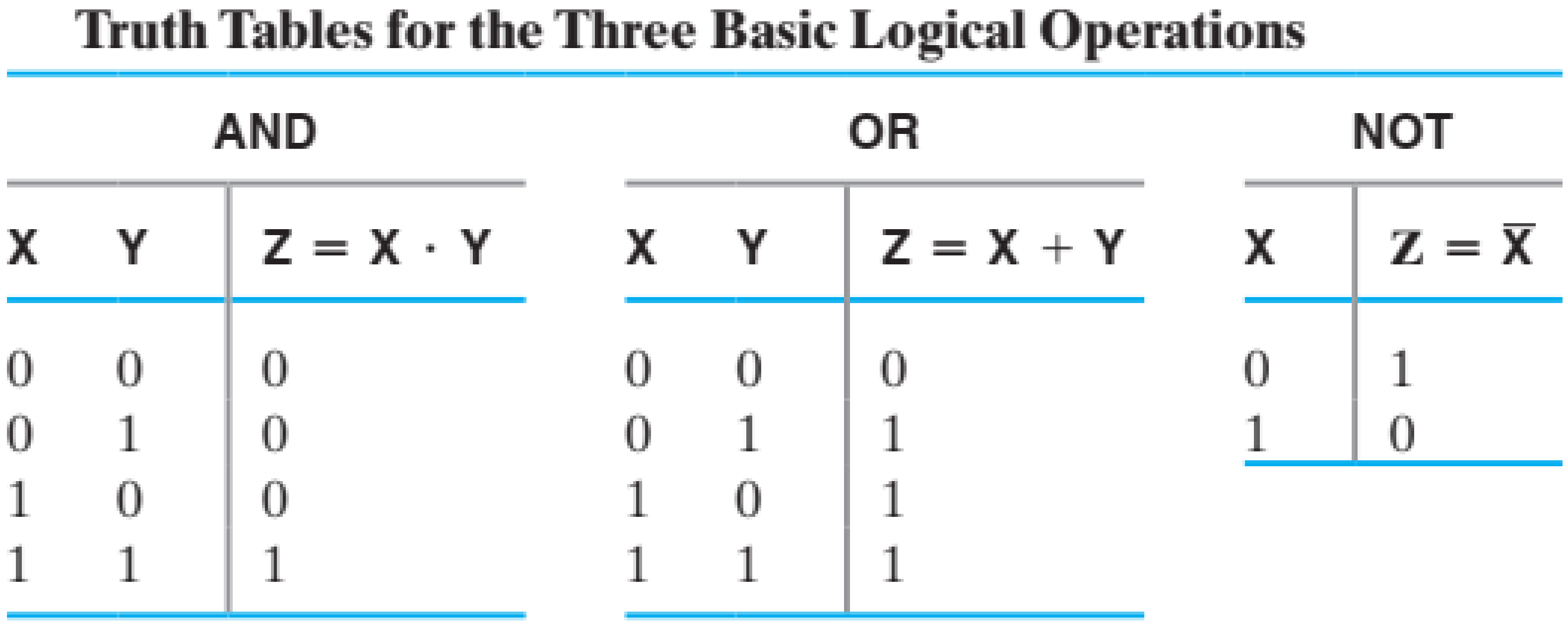
NOT operation

$\overline{0} = 1$

$\overline{1} = 0$

Logical Operations

- A *truth table* for an operation is a table of combinations of the binary variables showing the relationship between the values that the variables take on and the values of the result of the operation.



Truth Tables for the Three Basic Logical Operations

AND

X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

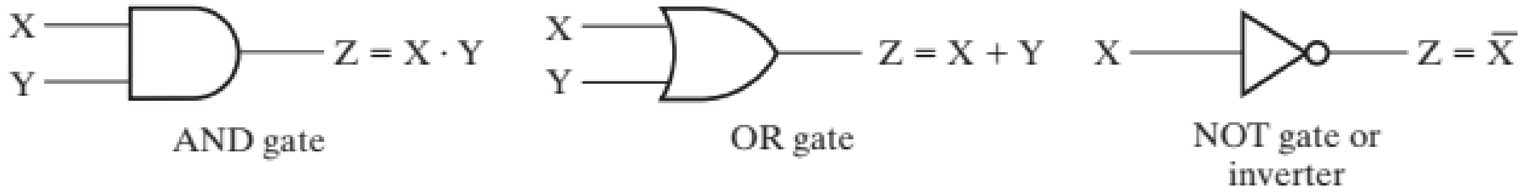
NOT

X	$Z = \overline{X}$
0	1
1	0

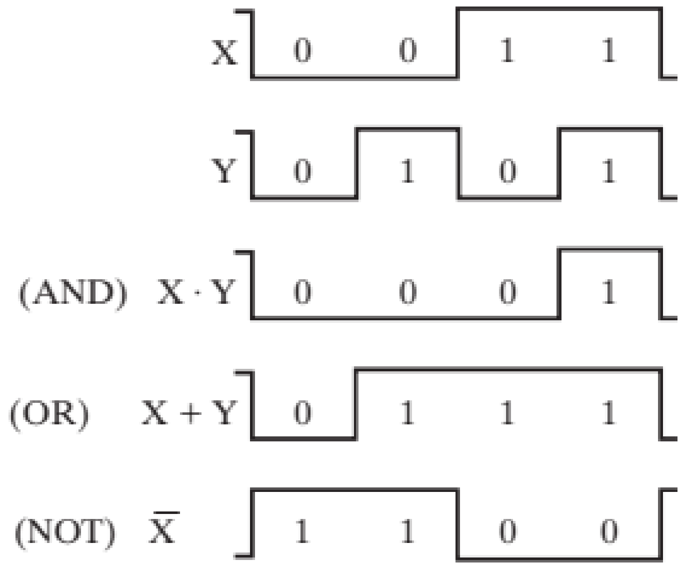
Logic Gates

- Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.
    - Voltage-operated circuits respond to two separate voltage ranges that represent a binary variable equal to logic 1 or logic 0.
    - The input terminals of logic gates accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within a specified range.
    - The intermediate regions between the allowed ranges in the figure are crossed only during changes from 1 to 0 or from 0 to 1.
        - These changes are called *transitions*.

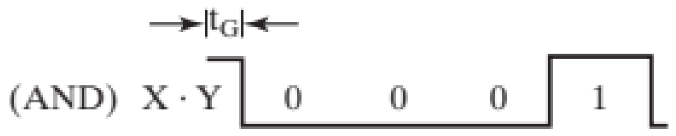
Logic Gates



(a) Graphic symbols

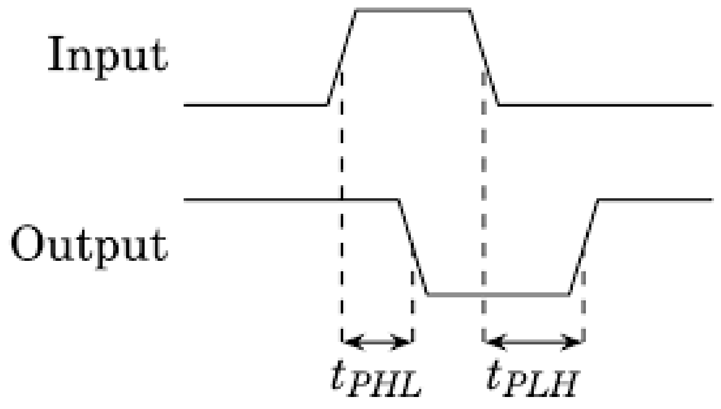


(b) Timing diagram



(c) AND timing diagram with gate delay t_G

- Each gate has another very important property called gate delay, the length of time it takes for an input change to result in the corresponding output change.



Logic Gates

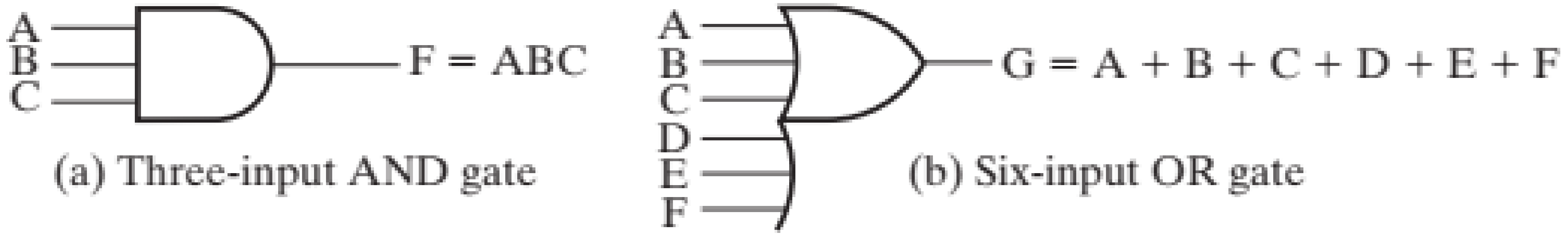
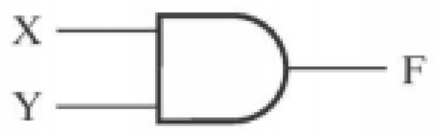
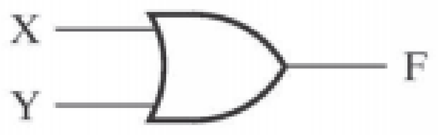
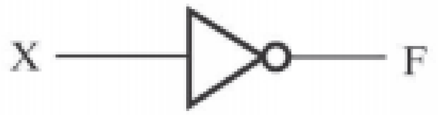
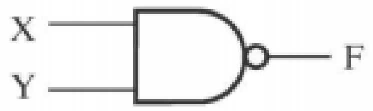
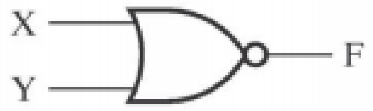
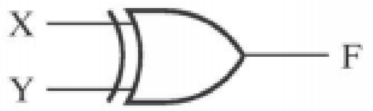
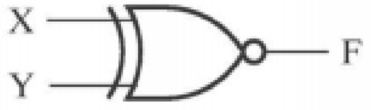


FIGURE 2-2
Gates with More than Two Inputs

Commonly Used Logic Gates

Name	Distinctive-Shape Graphics Symbol	Algebraic Equation	Truth Table
AND		$F = XY$	$\begin{array}{cc\|c} X & Y & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$
OR		$F = X + Y$	$\begin{array}{cc\|c} X & Y & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
NOT (inverter)		$F = \overline{X}$	$\begin{array}{c\|c} X & F \\ \hline 0 & 1 \\ 1 & 0 \end{array}$

Commonly Used Logic Gates

			X	Y	F
NAND		$F = \overline{X \cdot Y}$	0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR		$F = \overline{X + Y}$	0	0	1
			0	1	0
			1	0	0
			1	1	0
Exclusive-OR (XOR)		$F = X\overline{Y} + \overline{X}Y$ $= X \oplus Y$	0	0	0
			0	1	1
			1	0	1
			1	1	0
Exclusive-NOR (XNOR)		$F = XY + \overline{X}\,\overline{Y}$ $= \overline{X \oplus Y}$	0	0	1
			0	1	0
			1	0	0
			1	1	1

Logic Gates

- If we consider the inverter as a degenerate version of NAND and NOR gates with just one input, NAND gates alone or NOR gates alone can implement any Boolean function.
  - Thus, these gate types are much more widely used than AND and OR gates in actual logic circuits. As a consequence, actual circuit implementations are often done in terms of these gate types.

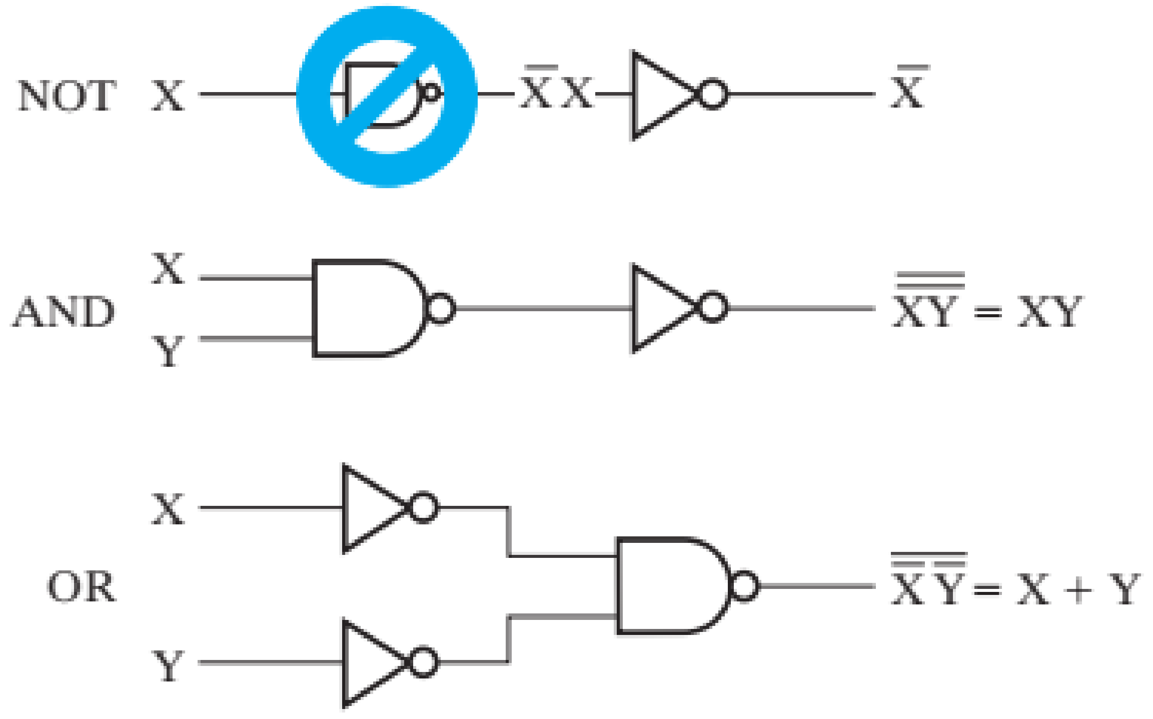

NOT
X
$\bar{X}$
X
$\bar{X}$
AND
X
Y
$\overline{\overline{XY}} = XY$
OR
X
Y
$\overline{\bar{X}\,\bar{Y}} = X + Y$

FIGURE 2-4
Logical Operations with NAND Gates

HDL Representations of Gates

- While schematics using the basic logic gates are sufficient for describing small circuits, they are impractical for designing more complex digital systems.
- In contemporary computer systems design, HDL (Hardware Description Language) has become intrinsic to the design process.
  - VHDL
  - Verilog

HDL Representations of Gates

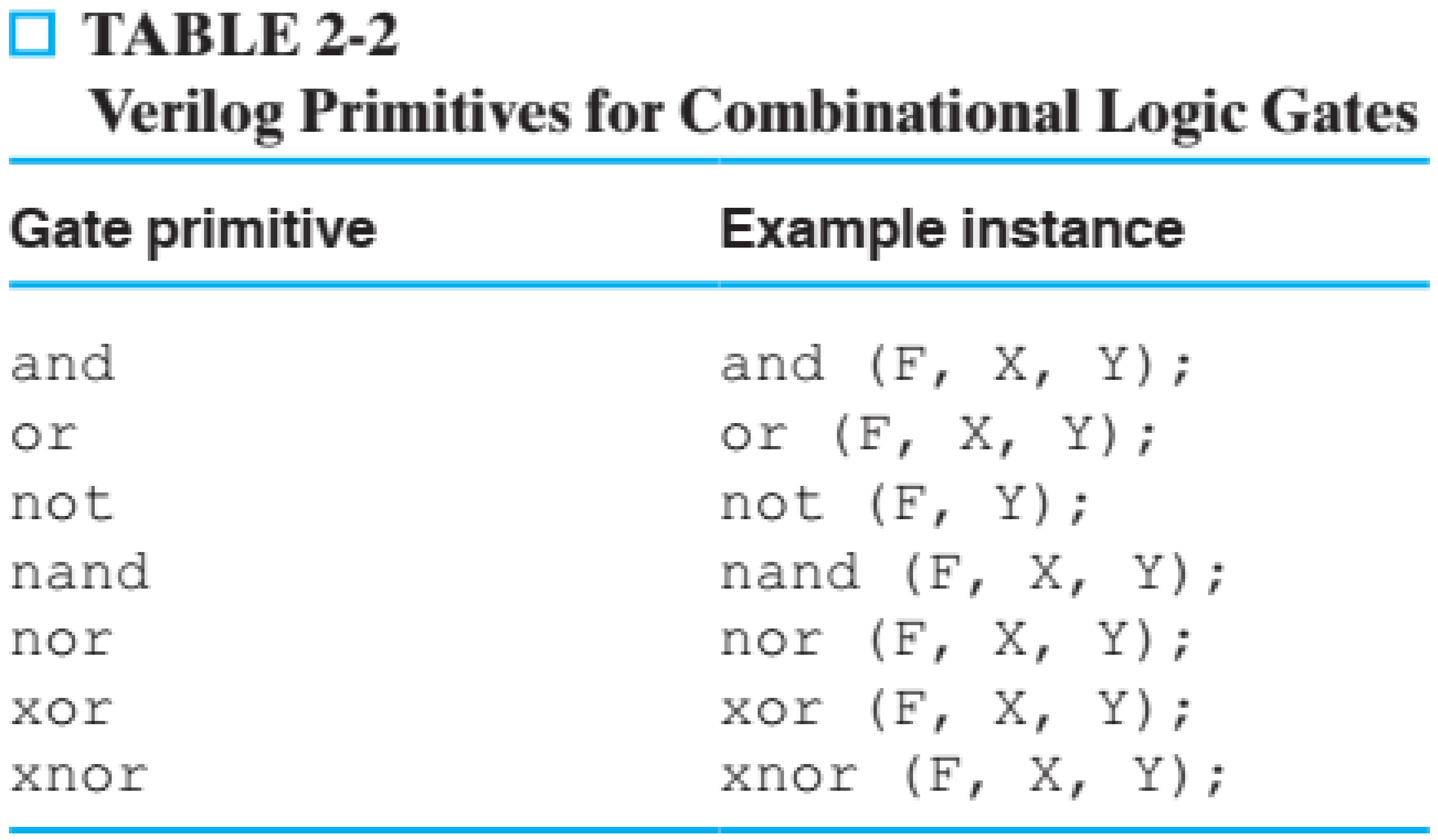


TABLE 2-2
Verilog Primitives for Combinational Logic Gates

Gate primitive	Example instance
`and`	`and (F, X, Y);`
`or`	`or (F, X, Y);`
`not`	`not (F, Y);`
`nand`	`nand (F, X, Y);`
`nor`	`nor (F, X, Y);`
`xor`	`xor (F, X, Y);`
`xnor`	`xnor (F, X, Y);`

HDL Representations of Gates

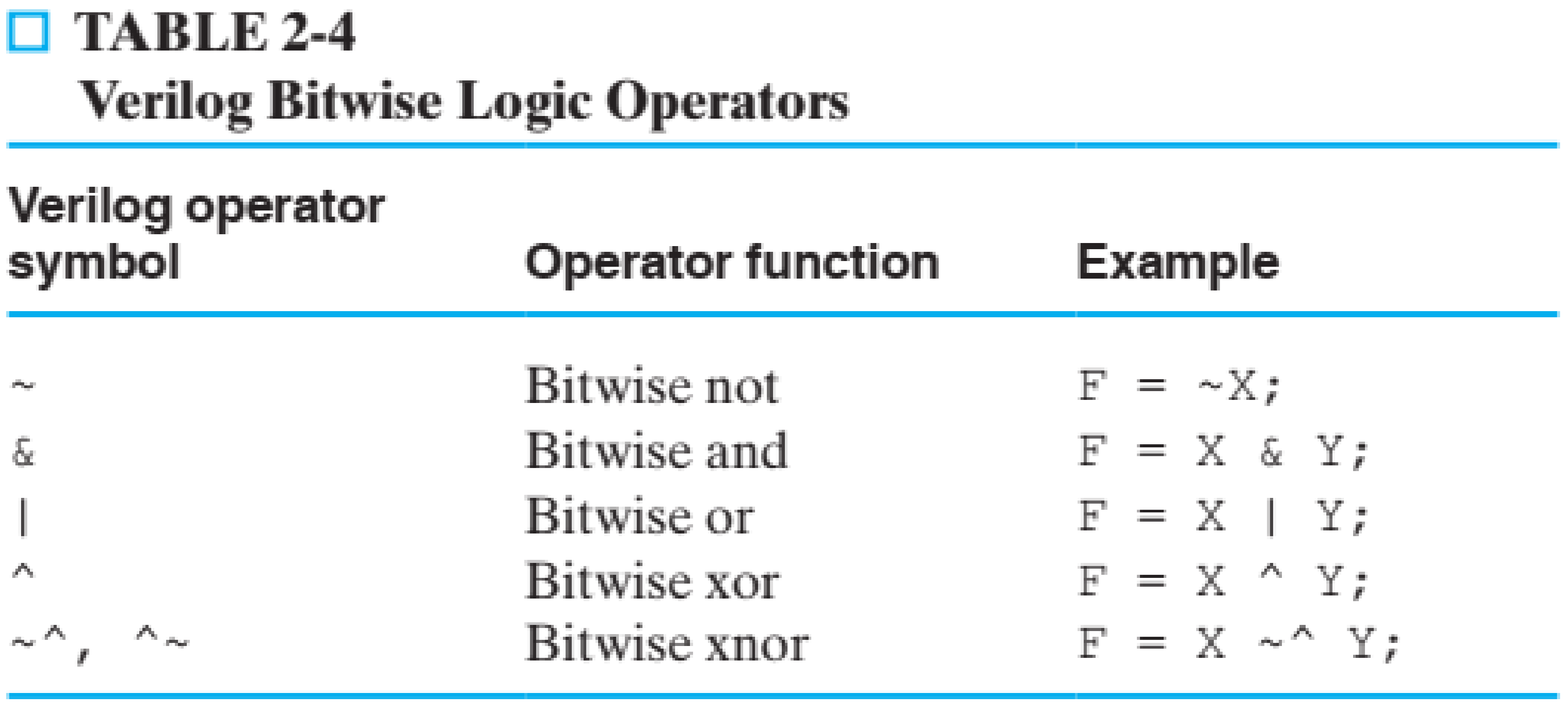


TABLE 2-4
Verilog Bitwise Logic Operators

Verilog operator symbol	Operator function	Example
`~`	Bitwise not	`F = ~X;`
`&`	Bitwise and	`F = X & Y;`
`|`	Bitwise or	`F = X | Y;`
`^`	Bitwise xor	`F = X ^ Y;`
`~^, ^~`	Bitwise xnor	`F = X ~^ Y;`

2. Boolean Algebra

- The *Boolean algebra* is an algebra dealing with binary variables and logic operations.
  - The variables are designated by letters of the alphabet.
  - The three basic logic operations are AND, OR, and NOT.
- A *Boolean expression* is an algebraic expression formed by using
  - Binary variables
  - Constants 0 and 1
  - Logic operation symbols
  - Parentheses.
- Boolean function

$$L(D,X,A) = D\bar{X} + A$$

2. Boolean Algebra

- A Boolean function can be represented by a truth table.
- A *truth table* for a function is a list of all combinations of 1s and 0s that can be assigned to the binary variables and a list that shows the value of the function for each binary combination.
- The number of rows in a truth table is 2^n, where n is the number of variables in the function.
  - The binary combinations for the truth table are the n-bit binary numbers that correspond to counting in decimal from 0 through $2^n - 1$.

2. Boolean Algebra

- Function $L(D, X, A) = D\bar{X} + A$

TABLE 2-5
Truth Table for the Function $L = D\overline{X} + A$

D	X	A	L
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

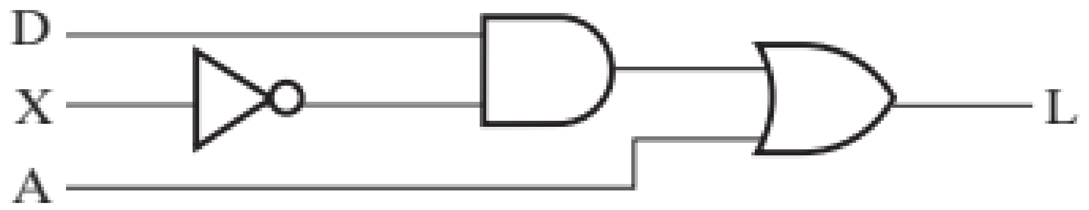
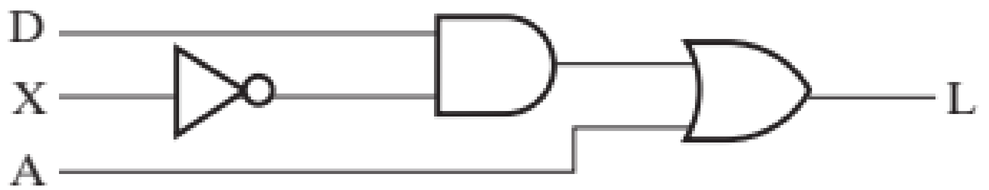


FIGURE 2-5
Logic Circuit Diagram for $L = D\overline{X} + A$

2. Boolean Algebra

- Function $L(D,X,A) = D\bar{X} + A$



❑ **FIGURE 2-5**
Logic Circuit Diagram for $L = D\overline{X} + A$

```
module fig2_5 (L, D, X, A);
    input D, X, A;
    output L;
    wire X_n, t2;

    not (X_n, X);
    and (t2, D, X_n);
    or (L, t2, A);
endmodule
```

❑ **FIGURE 2-6**
Verilog Model for the Logic Circuit of Figure 2-5

Boolean Algebra

- Circuit gates are interconnected by wires that carry logic signals. Logic circuits of this type are called combinational logic circuits, since the variables are “combined” by the logical operations.
- There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic equation form, it can be expressed in a variety of ways.

Basic Identities of Boolean Algebra

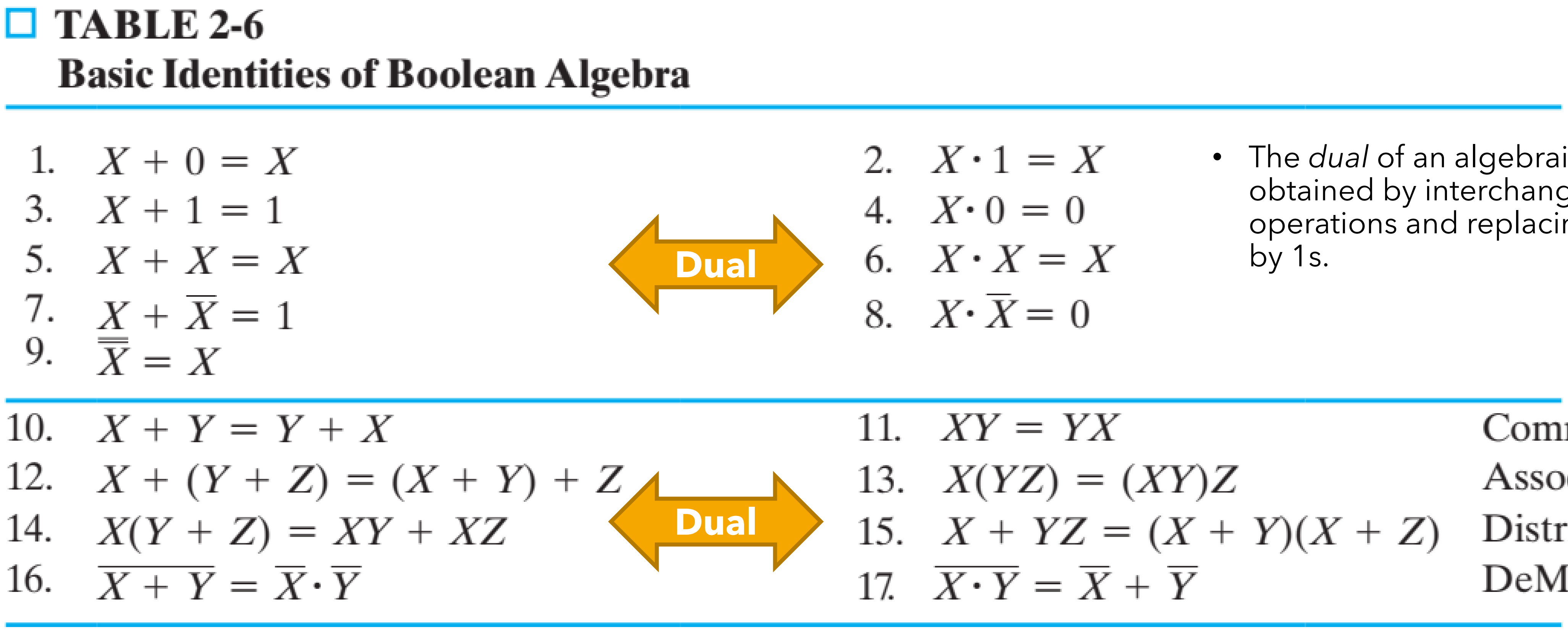


TABLE 2-6
Basic Identities of Boolean Algebra

1. $X + 0 = X$	Dual	2. $X \cdot 1 = X$	
3. $X + 1 = 1$		4. $X \cdot 0 = 0$	
5. $X + X = X$		6. $X \cdot X = X$	
7. $X + \overline{X} = 1$		8. $X \cdot \overline{X} = 0$	
9. $\overline{\overline{X}} = X$			
10. $X + Y = Y + X$	Dual	11. $XY = YX$	Commutative
12. $X + (Y + Z) = (X + Y) + Z$		13. $X(YZ) = (XY)Z$	Associative
14. $X(Y + Z) = XY + XZ$		15. $X + YZ = (X + Y)(X + Z)$	Distributive
16. $\overline{X + Y} = \overline{X} \cdot \overline{Y}$		17. $\overline{X \cdot Y} = \overline{X} + \overline{Y}$	DeMorgan's

- The *dual* of an algebraic expression is obtained by interchanging OR and AND operations and replacing 1s by 0s and 0s by 1s.

Basic Identities of Boolean ALgebra

- DeMorgan's theorem can be extended to three or more variables.

$$\overline{X_1 + X_2 + \ldots + X_n} = \overline{X}_1\overline{X}_2\ldots\overline{X}_n$$

$$\overline{X_1X_2\ldots X_n} = \overline{X}_1 + \overline{X}_2 + \ldots + \overline{X}_n$$

  - For example

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

Basic Identities of Boolean Algebra

- Using truth tables to prove the logical equivalence

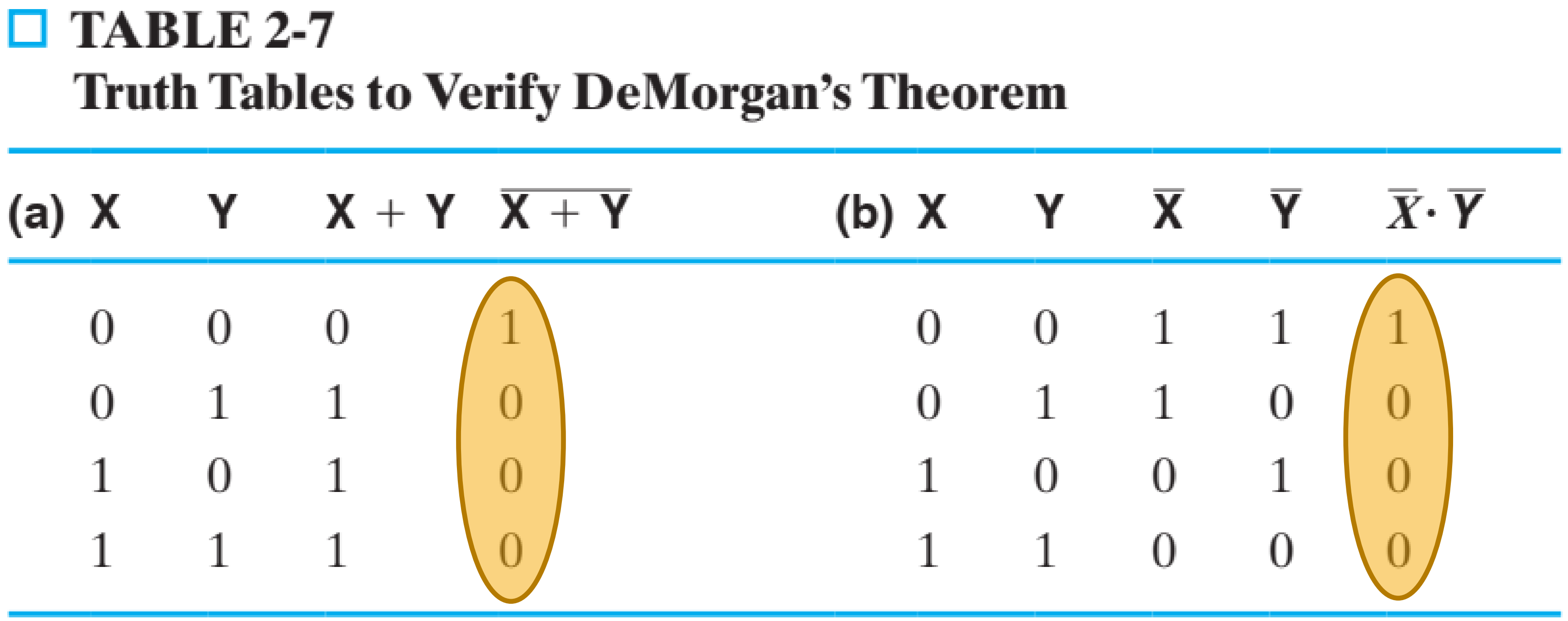


TABLE 2-7
Truth Tables to Verify DeMorgan's Theorem

(a) X	Y	X + Y	$\overline{X + Y}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

(b) X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} \cdot \overline{Y}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

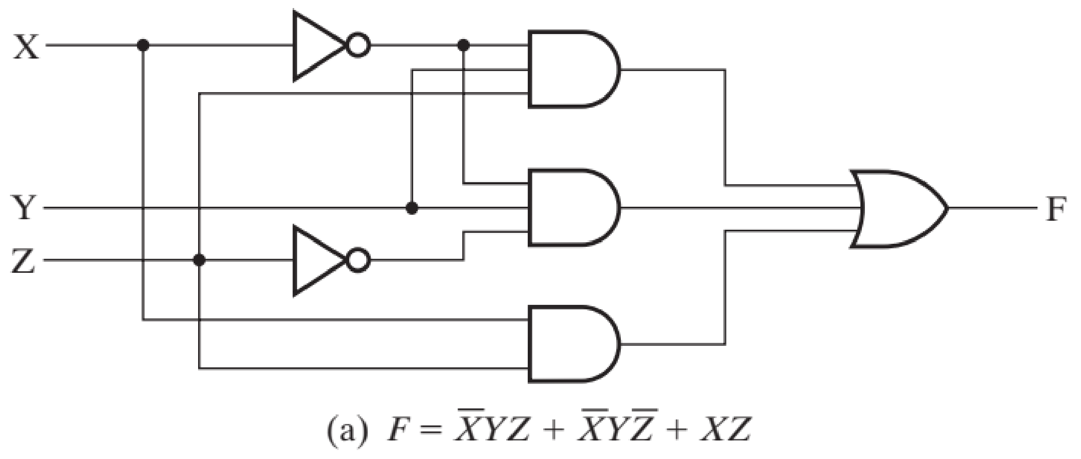
Algebraic Manipulation

- Boolean algebra is a useful tool for simplifying digital circuits.

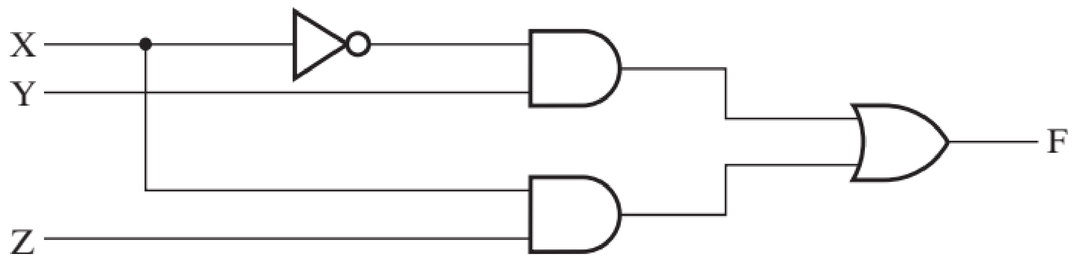
$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

$$\begin{aligned} F &= \overline{X}YZ + \overline{X}Y\overline{Z} + XZ & \\ &= \overline{X}Y(Z + \overline{Z}) + XZ & \text{by identity 14} \\ &= \overline{X}Y \cdot 1 + XZ & \text{by identity 7} \\ &= \overline{X}Y + XZ & \text{by identity 2} \end{aligned}$$

Algebraic Manipulation



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$



(b) $F = \overline{X}Y + XZ$

FIGURE 2-8
Implementation of Boolean Function with Gates

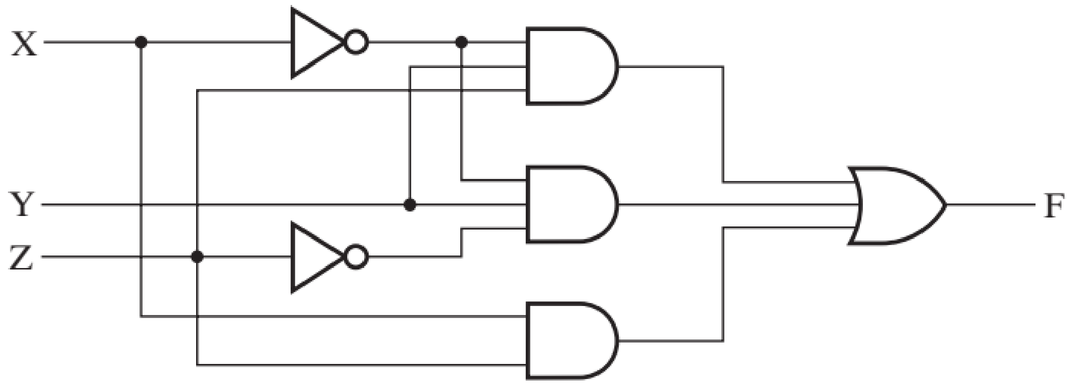
TABLE 2-8
Truth Table for Boolean Function

X	Y	Z	(a) F	(b) F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

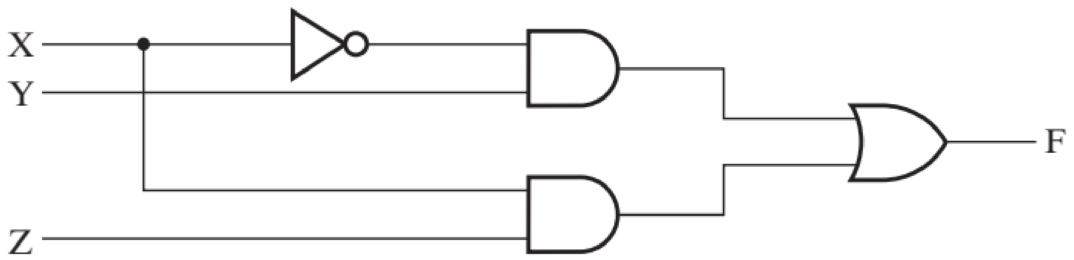
=

- Each circuit implements the same function, but the one with fewer gates and/or fewer gate inputs is preferable because it requires fewer components

Algebraic Manipulation



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$



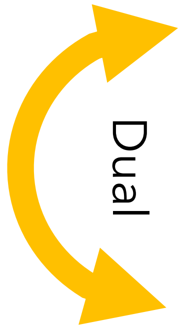
(b) $F = \overline{X}Y + XZ$

FIGURE 2-8
Implementation of Boolean Function with Gates

- When a Boolean equation is implemented with logic gates, each term requires a gate, and each variable within the term designates an input to the gate. We define a *literal* as a single variable within a term that may or may not be complemented.
  - Figure 2-8(a): 3 terms, 8 literals
  - Figure 2-8(b): 2 terms, 4 literals
- By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. Boolean algebra is applied to reduce an expression for the purpose of obtaining a simpler circuit.

Algebraic Manipulation

- Some useful equalities



1. $X + XY = X \cdot 1 + XY = X(1 + Y) = X \cdot 1 = X$
2. $XY + X\overline{Y} = X(Y + \overline{Y}) = X \cdot 1 = X$
3. $X + \overline{X}Y = (X + \overline{X})(X + Y) = 1 \cdot (X + Y) = X + Y$
4. $X(X + Y) = X \cdot X + X \cdot Y = X + XY = X(1 + Y) = X \cdot 1 = X$
5. $(X + Y)(X + \overline{Y}) = X + Y\overline{Y} = X + 0 = X$
6. $X(\overline{X} + Y) = X\overline{X} + XY = 0 + XY = XY$

- Consensus theorem

$$XY + \overline{X}Z + YZ = XY + \overline{X}Z$$

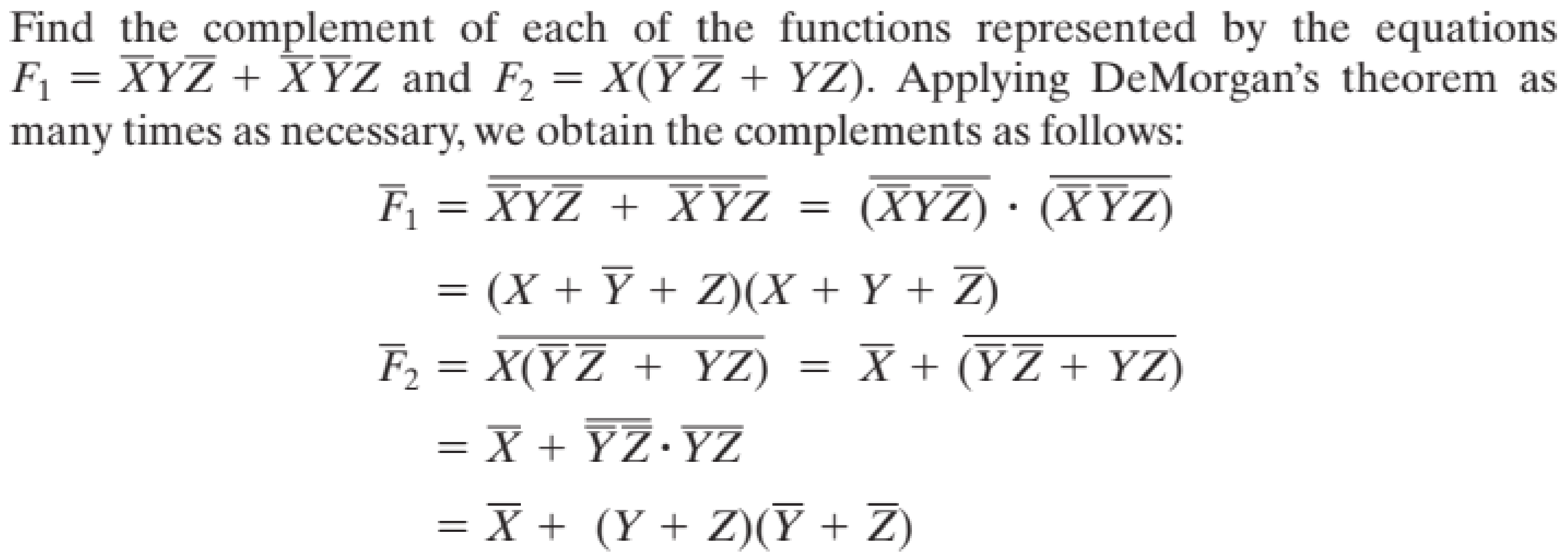
$$(X + Y)(\overline{X} + Z)(Y + Z) = (X + Y)(\overline{X} + Z)$$

Complement of a Function

- 1. The complement of a function can be derived algebraically by applying DeMorgan's theorem.
- 2. A simpler method for deriving the complement of a function is to take the dual of the function equation and complement each literal.
- 3. The complement representation for a function *F*, $\bar{F}$, obtained from an interchange of 1s to 0s and 0s to 1s for the values of *F* in the truth table.

Complement of a Function

- 1. Complementing a function by applying DeMorgan’s theorem



Find the complement of each of the functions represented by the equations $F_1 = \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z$ and $F_2 = X(\bar{Y}\bar{Z} + YZ)$. Applying DeMorgan’s theorem as many times as necessary, we obtain the complements as follows:

$$\bar{F}_1 = \overline{\bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z} = (\overline{\bar{X}Y\bar{Z}}) \cdot (\overline{\bar{X}\bar{Y}Z})$$

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

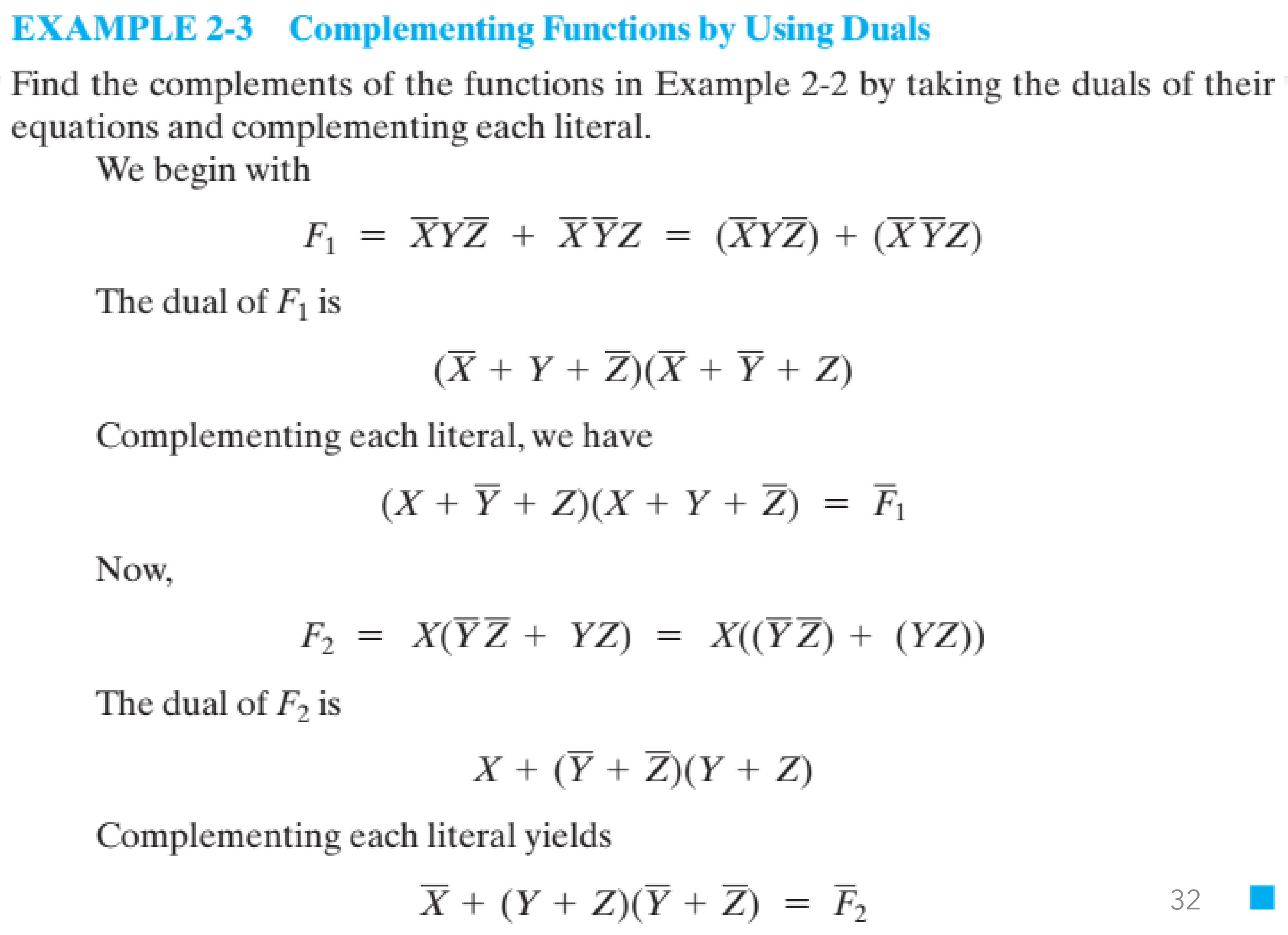
$$\bar{F}_2 = \overline{X(\bar{Y}\bar{Z} + YZ)} = \bar{X} + (\overline{\bar{Y}\bar{Z} + YZ})$$

$$= \bar{X} + \overline{\bar{Y}\bar{Z}} \cdot \overline{YZ}$$

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

Complement of a Function

- 3. Complementing a function by using dual



EXAMPLE 2-3 Complementing Functions by Using Duals

Find the complements of the functions in Example 2-2 by taking the duals of their equations and complementing each literal.

We begin with

$$F_1 = \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z = (\overline{X}Y\overline{Z}) + (\overline{X}\,\overline{Y}Z)$$

The dual of F_1 is

$$(\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + Z)$$

Complementing each literal, we have

$$(X + \overline{Y} + Z)(X + Y + \overline{Z}) = \overline{F}_1$$

Now,

$$F_2 = X(\overline{Y}\,\overline{Z} + YZ) = X((\overline{Y}\,\overline{Z}) + (YZ))$$

The dual of F_2 is

$$X + (\overline{Y} + \overline{Z})(Y + Z)$$

Complementing each literal yields

$$\overline{X} + (Y + Z)(\overline{Y} + \overline{Z}) = \overline{F}_2$$

Complement of a Function

- 3. The complement representation for a function F, $\bar{F}$, obtained from an interchange of 1s to 0s and 0s to 1s for the values of F in the truth table.

D	X	A	$L = D\bar{X} + A$	$\bar{L}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

3. Standard Forms

- The standard forms facilitate the simplification procedures for Boolean expressions and, in some cases, may result in more desirable expressions for implementing logic circuits.
- The standard forms contain product terms and sum terms.
  - Product term: This is a logical product consisting of an AND operation among three literals.

$$X\overline{Y}Z.$$

  - Sum term: This is a logical sum consisting of an OR operation among the literals.

$$X + Y + \overline{Z}.$$

Minterms and Maxterms

- Minterm: A product term in which **all** the variables appear exactly once, either complemented or uncomplemented, is called a *minterm*.
- Maxterm: A sum term that contains **all** the variables in complemented or uncomplemented form is called a *maxterm*.

Minterms and Maxterms

- *n* is the number of variable -> 2^n minsterms, 2^n maxterms

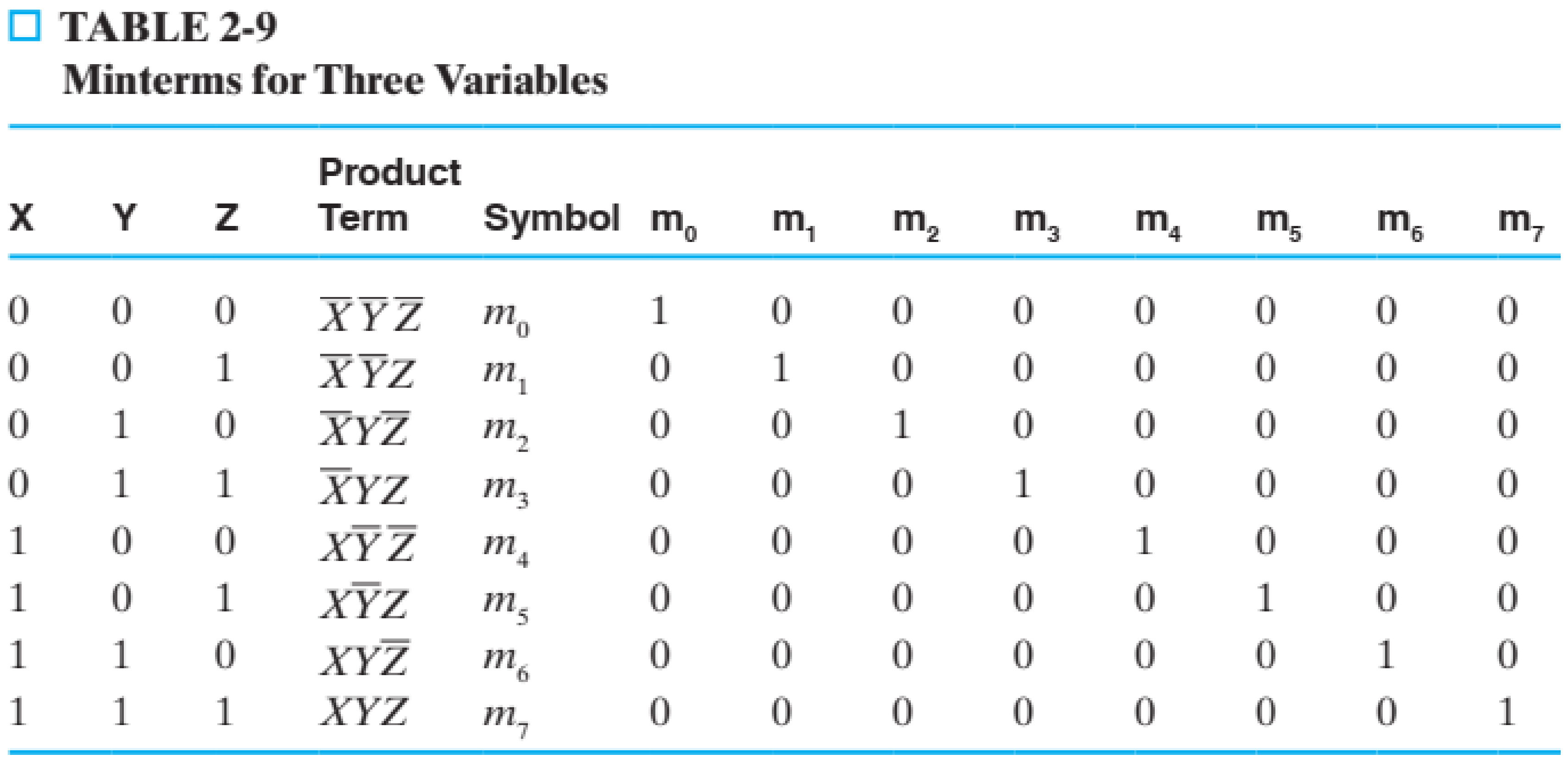


TABLE 2-9
Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\overline{Y}\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}Y\overline{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

Minterms and Maxterms

- 2^n maxterms

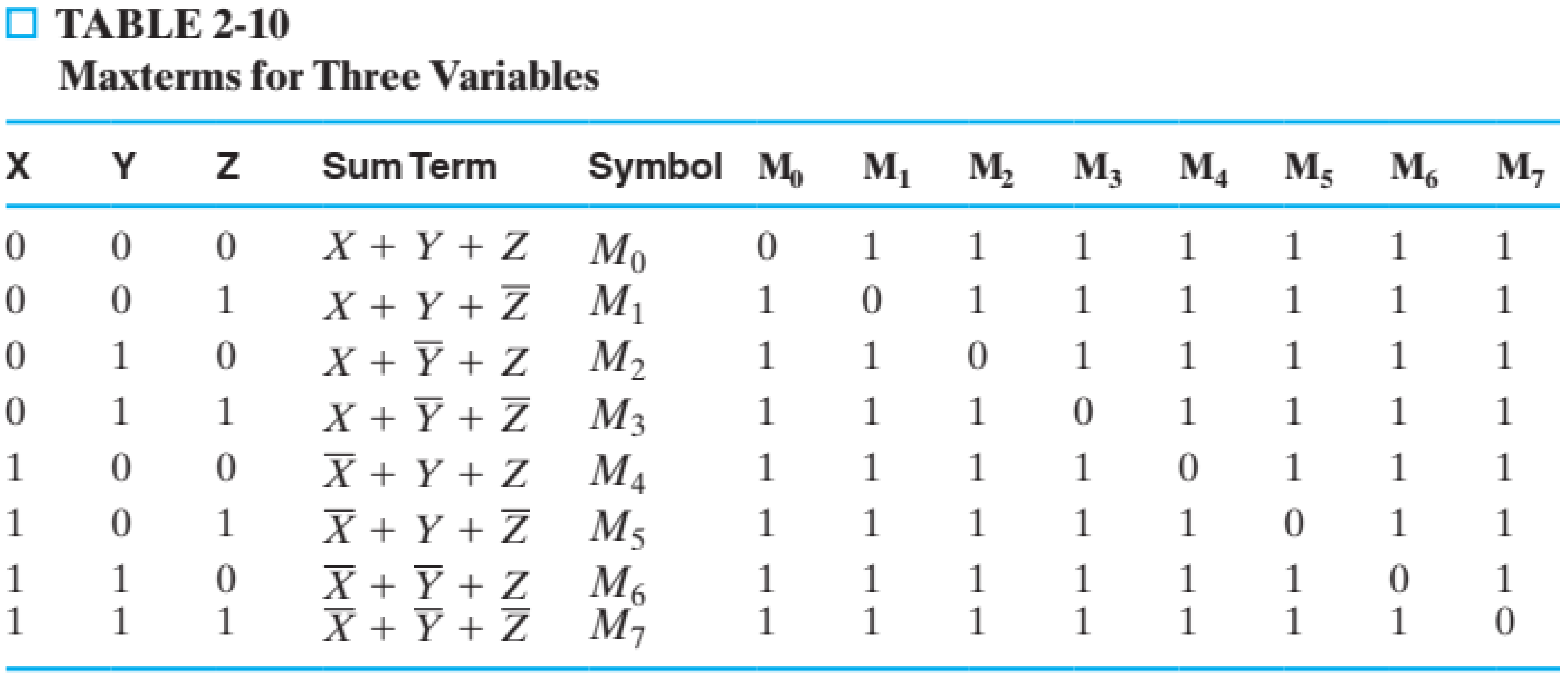


TABLE 2-10
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X + Y + Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X + \overline{Y} + Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X + \overline{Y} + \overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X} + Y + Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X} + Y + \overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X} + \overline{Y} + Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X} + \overline{Y} + \overline{Z}$	M_7	1	1	1	1	1	1	1	0

Minterms and Maxterms

- A Boolean function can be represented algebraically from a given truth table by
  - A *sum of minterms*
  - A product of *maxterms*

Standard Forms

- 1. ***Sum-of-Products Form***
  - sum of all the minterms that produce a **1** in the function

$$F = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

$$F(X, Y, Z) = \Sigma m(0, 2, 5, 7)$$

2. ***Product-of-Sums Form***

product of all the maxterms that produce a **0** in the function

$$F = (X + Y + \bar{Z})(X + \bar{Y} + \bar{Z})(\bar{X} + Y + Z)(\bar{X} + \bar{Y} + Z)$$

$$F(X, Y, Z) = \prod M(1, 3, 4, 6)$$

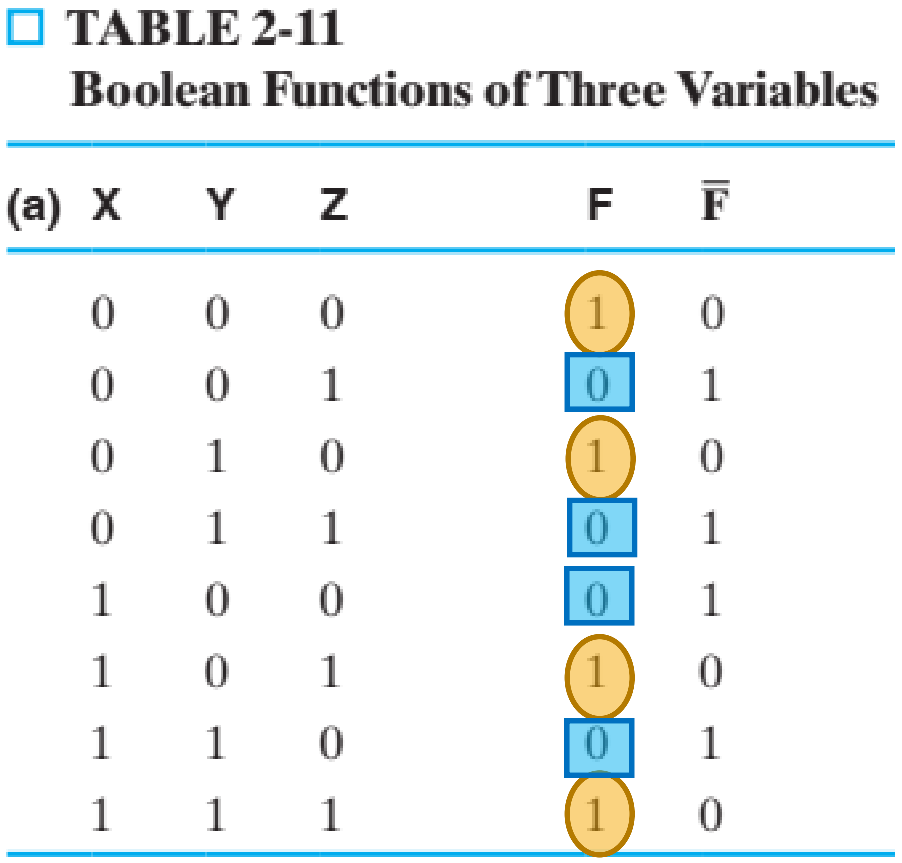
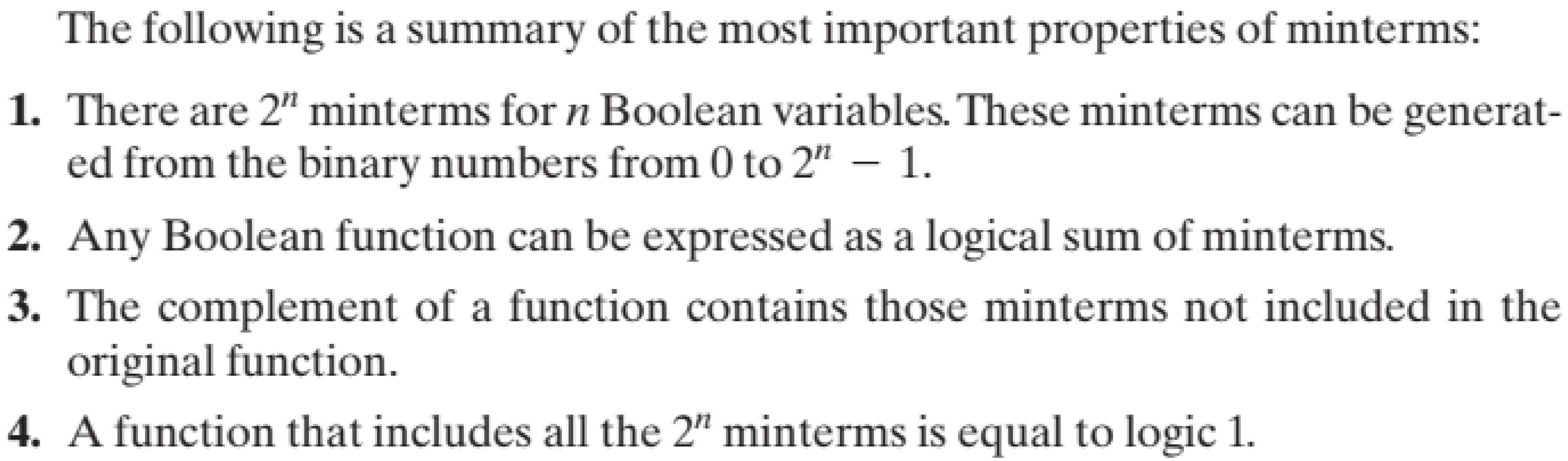


TABLE 2-11
Boolean Functions of Three Variables

(a) X	Y	Z	F	$\bar{F}$
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

Minterms and Maxterms



The following is a summary of the most important properties of minterms:

1. There are 2^n minterms for n Boolean variables. These minterms can be generated from the binary numbers from 0 to $2^n - 1$.
2. Any Boolean function can be expressed as a logical sum of minterms.
3. The complement of a function contains those minterms not included in the original function.
4. A function that includes all the 2^n minterms is equal to logic 1.

Sum of Products

$$F = \overline{Y} + \overline{X}Y\overline{Z} + XY$$

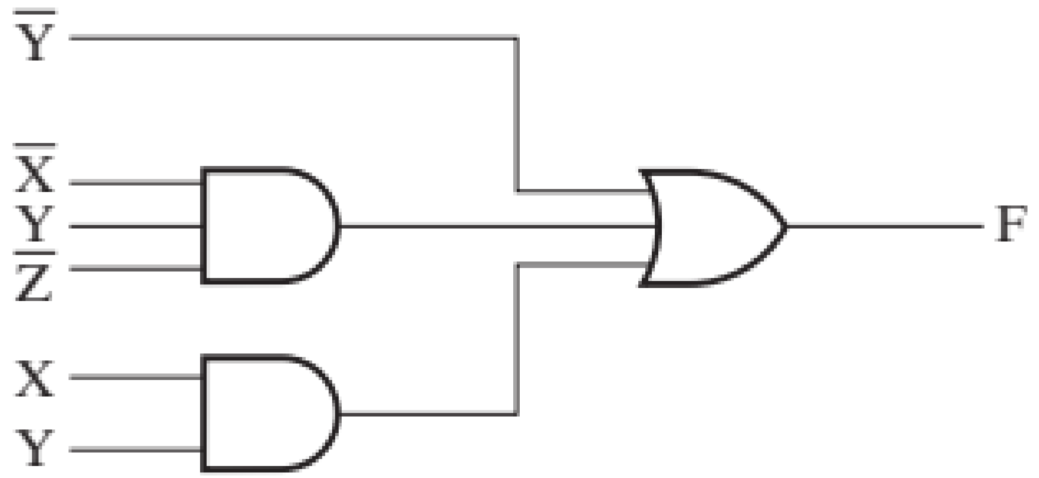


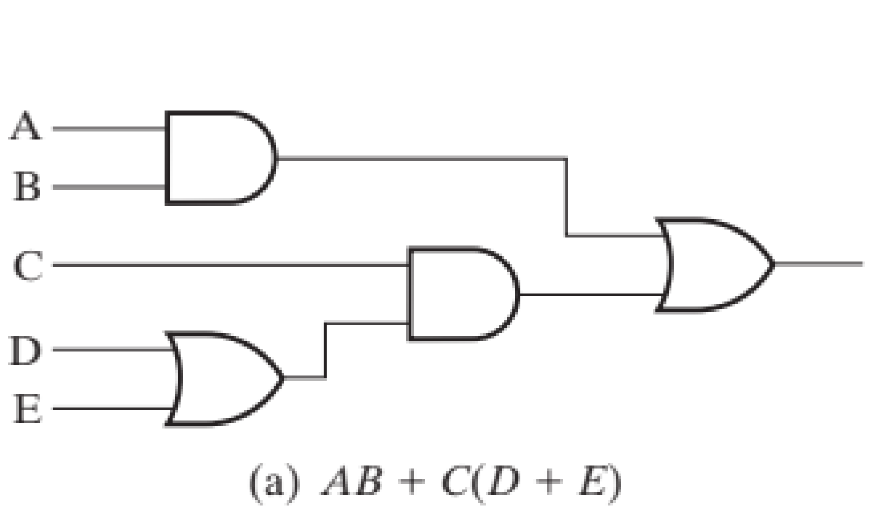
FIGURE 2-9
Sum-of-Products Implementation

- The AND gates followed by the OR gate form a circuit configuration referred to as a *two-level implementation* or *two-level circuit*.

Sum of Products

$$F = AB + C(D + E)$$

$$F = AB + C(D + E) = AB + CD + CE$$



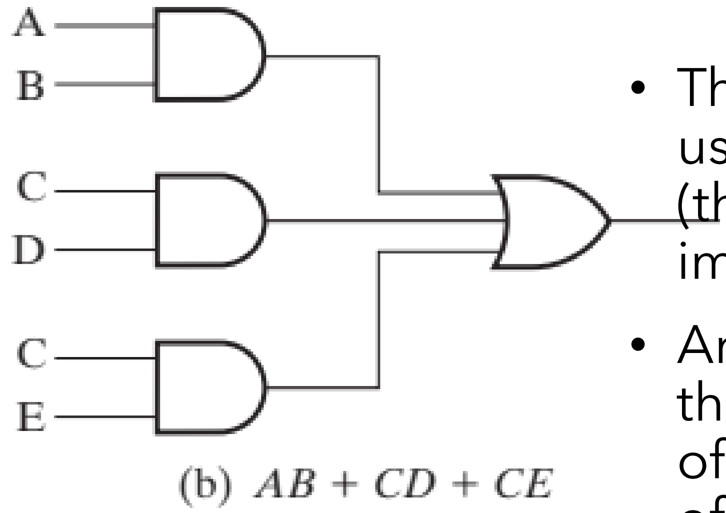


FIGURE 2-10
Three-Level and Two-Level Implementation

- The decision as to whether to use a two-level or multiple-level (three levels or more) implementation is complex.
- Among the issues involved are the number of gates, number of gate inputs, and the amount of delay between the time the input values are set and the time the resulting output values appear. Two-level implementations are the natural form for certain implementation technologies.

Cost Criteria

- Cost Criteria:
  - To formalize the way of measuring the simplicity of a logic circuit
  - *Literal cost* and *gate-input cost*
- Literal cost
  - The number of literal appearances in a Boolean expression corresponding exactly to the logic diagram

Literal Cost

$$F = AB + C(D + E) \text{ and } F = AB + CD + CE$$

Literal cost **5** **6**

- Literal cost has the advantage that it is very simple to evaluate by counting literal appearances.
- It does not, however, represent circuit complexity accurately in all cases, even for the comparison of different implementations of the same logic function.

$$G = ABCD + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \text{ and } G = (\overline{A} + B)(\overline{B} + C)(\overline{C} + D)(\overline{D} + A)$$

Literal cost **8** **8**

But, the first equation has two terms and the second has four. This suggests that the first equation has a lower cost than the second.

Gate-Input Cost

- Gate-input cost is the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations.
  - 1. all literal appearances,
  - 2. the number of terms excluding terms that consist only of a single literal, and, optionally,
  - 3. the number of distinct complemented single literals

$$G = ABCD + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \quad \text{and} \quad G = (\overline{A} + B)(\overline{B} + C)(\overline{C} + D)(\overline{D} + A)$$

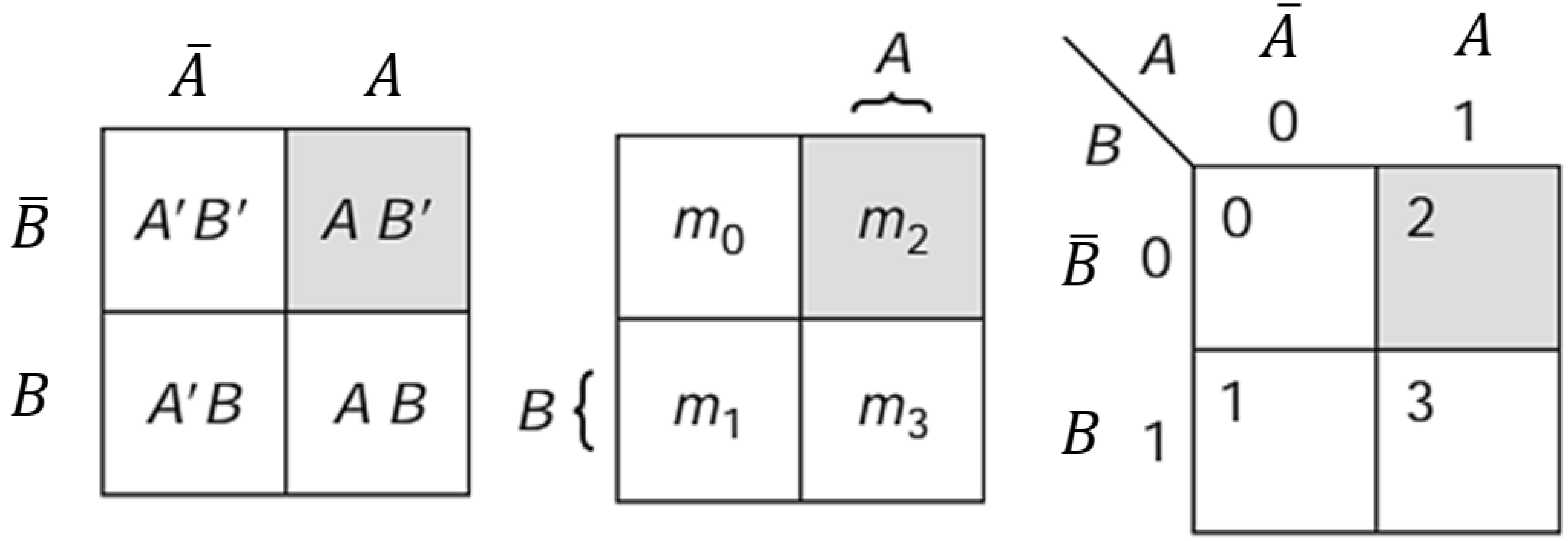
Gate-Input cost **8 + 2 + 4** **8 + 4 + 4**

Gate-input cost is currently a good measure for contemporary logic implementations, since it is proportional to the number of transistors and wires used in implementing a logic circuit

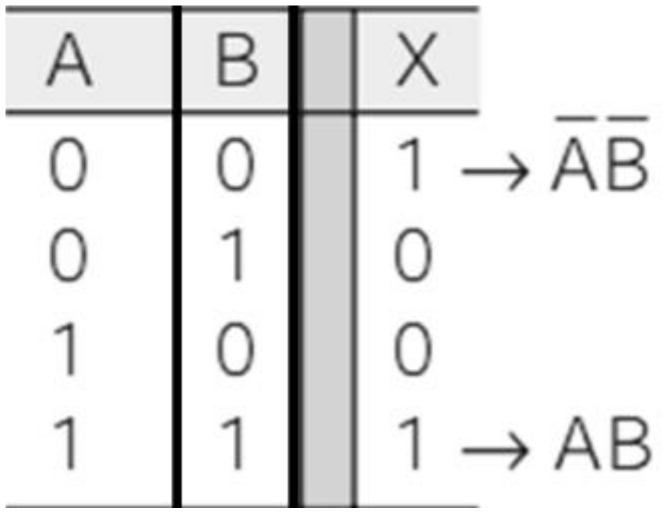
4. Karnaugh Map (K Map)

- The K map is a graphical tool
    - Simplify a logic expression
    - Convert a truth table into a simple one
- Each case in the truth table corresponds to a square in the K map.
- Horizontally or vertically, **adjacent square differs only in ONE variable.**
- Once a K map has been filled with 0s and 1s, the **sum-of-products** expression for the output X can be obtained by ORing together those squares that contain a 1.

Two-Variable K Map

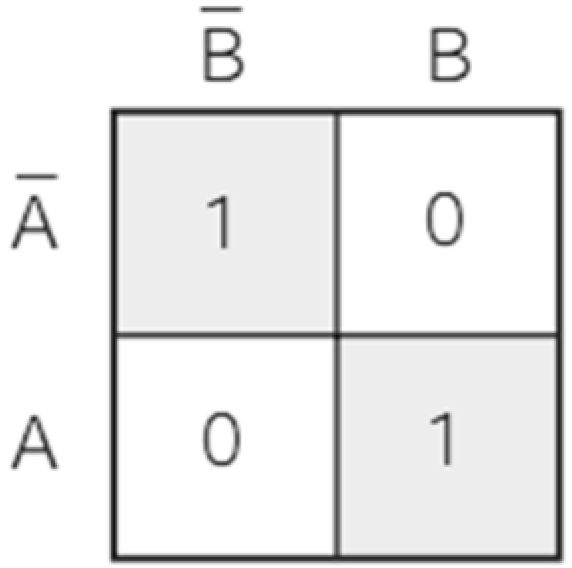


Two-Variable K Map



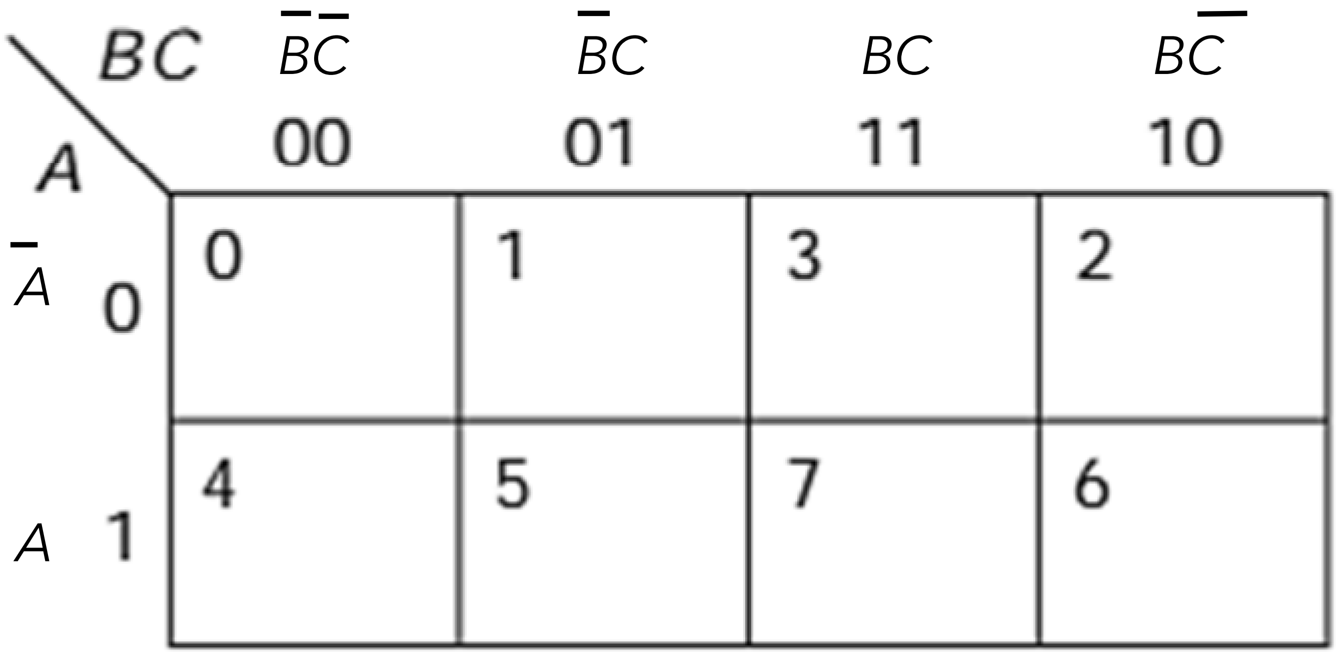
A	B	X	
0	0	1	$\rightarrow \bar{A}\bar{B}$
0	1	0	
1	0	0	
1	1	1	$\rightarrow AB$

$$\left\{ X = \bar{A}\bar{B} + AB \right\}$$



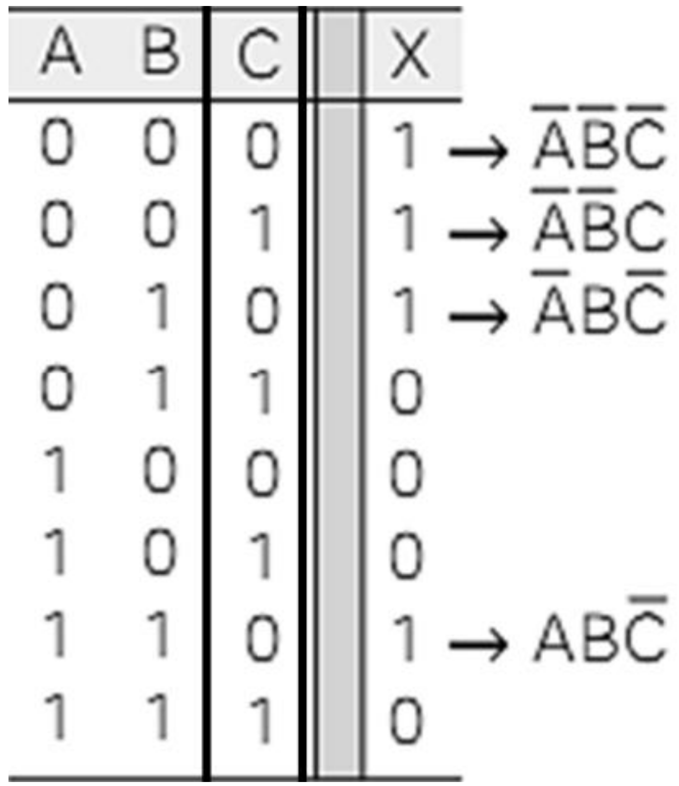
$$\begin{array}{c|cc} & \bar{B} & B \\ \hline \bar{A} & 1 & 0 \\ A & 0 & 1 \end{array}$$

Three-Variable K Map



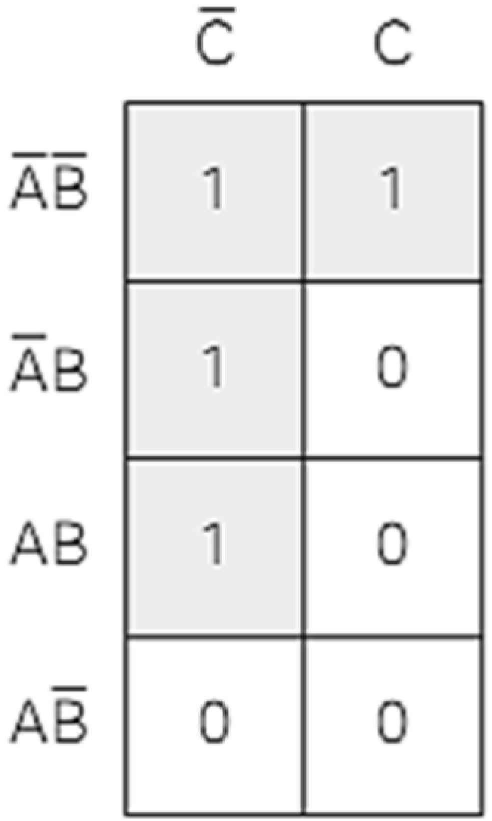
$A \backslash BC$	$\bar{B}\bar{C}$ 00	$\bar{B}C$ 01	BC 11	$B\bar{C}$ 10
$\bar{A}$ 0	0	1	3	2
A 1	4	5	7	6

Three-Variable K Map



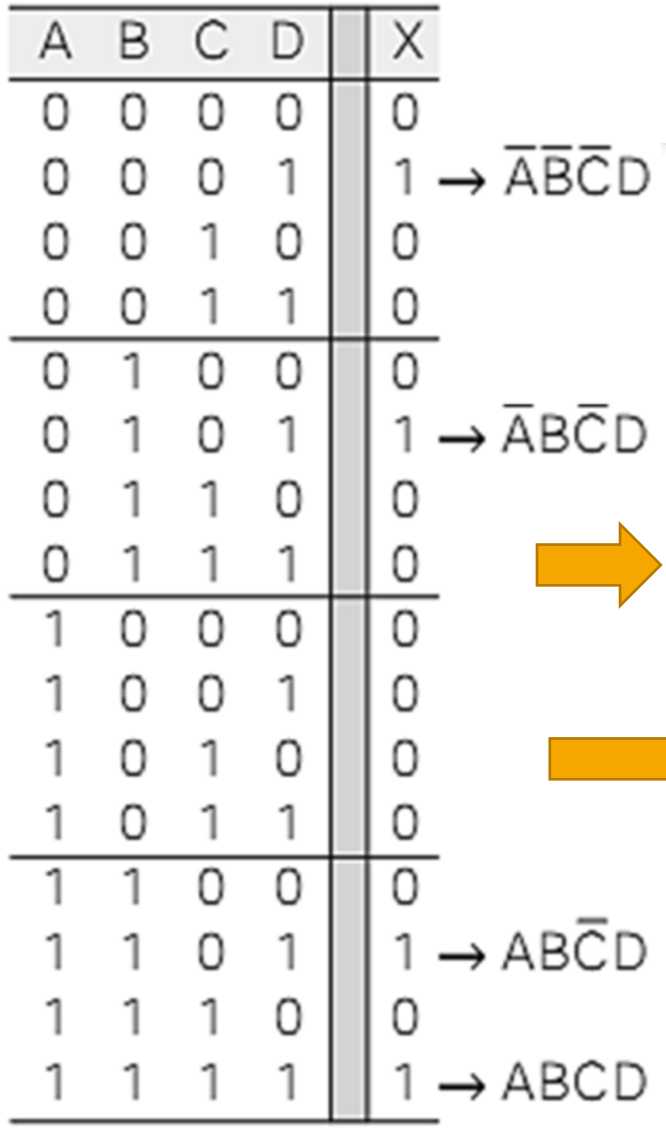
A	B	C	X	
0	0	0	1	$\rightarrow \bar{A}\bar{B}\bar{C}$
0	0	1	1	$\rightarrow \bar{A}\bar{B}C$
0	1	0	1	$\rightarrow \bar{A}B\bar{C}$
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	$\rightarrow AB\bar{C}$
1	1	1	0	

$$\left\{ \begin{aligned} X = {} & \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C \\ & + \bar{A}B\bar{C} + AB\bar{C} \end{aligned} \right\}$$



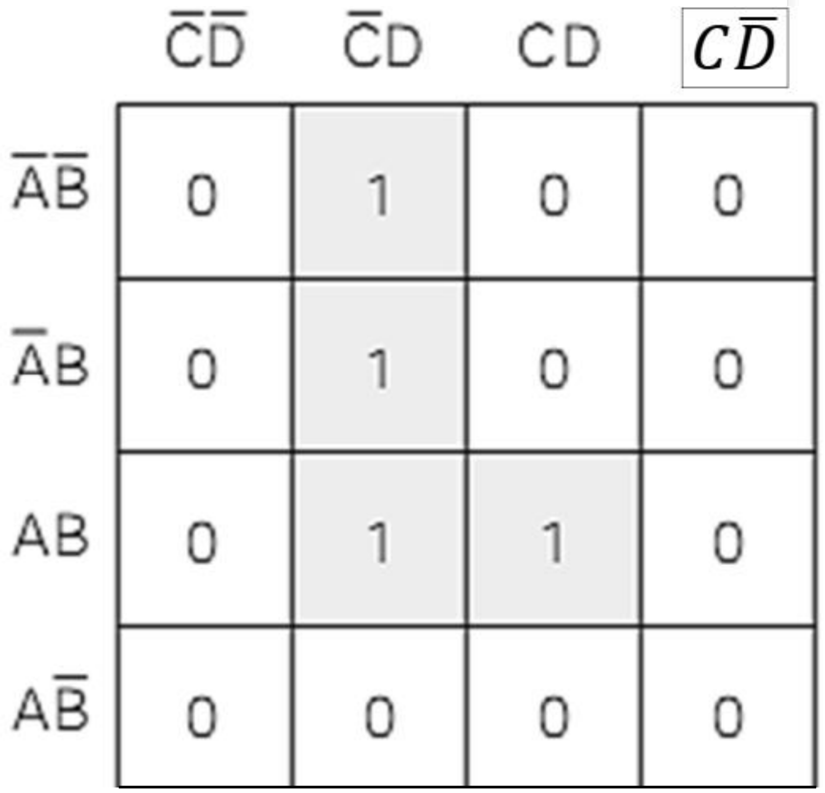
	$\bar{C}$	C
$\bar{A}\bar{B}$	1	1
$\bar{A}B$	1	0
AB	1	0
$A\bar{B}$	0	0

Four-Variable K Map



A	B	C	D	X	
0	0	0	0	0	
0	0	0	1	1	$\rightarrow \bar{A}\bar{B}\bar{C}D$
0	0	1	0	0	
0	0	1	1	0	
0	1	0	0	0	
0	1	0	1	1	$\rightarrow \bar{A}B\bar{C}D$
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	0	
1	1	0	0	0	
1	1	0	1	1	$\rightarrow AB\bar{C}D$
1	1	1	0	0	
1	1	1	1	1	$\rightarrow ABCD$

$$\left\{ \begin{aligned} X = \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}D \\ + AB\bar{C}D + ABCD \end{aligned} \right\}$$

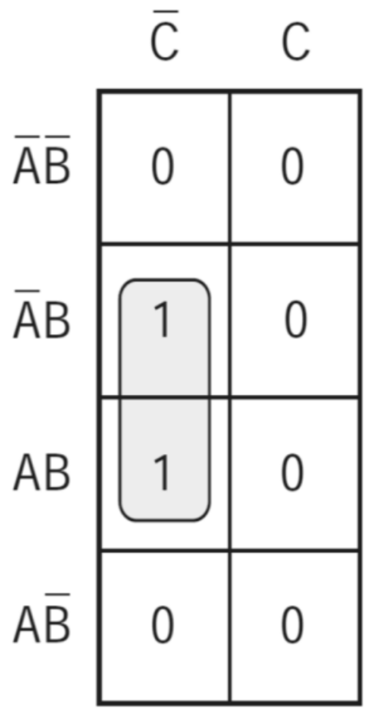


	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	0	0
AB	0	1	1	0
$A\bar{B}$	0	0	0	0

Karnaugh Map

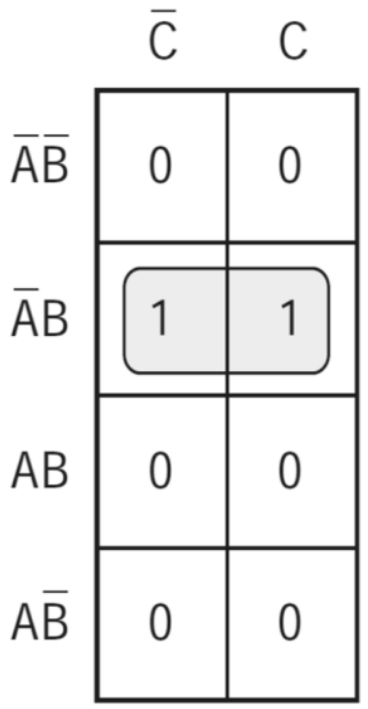
- **Looping**: The expression for the output X can be simplified by properly combining those squares in the K map that contain 1s. The process for combining these 1s is called “looping”.

Looping Groups of Two (pairs)



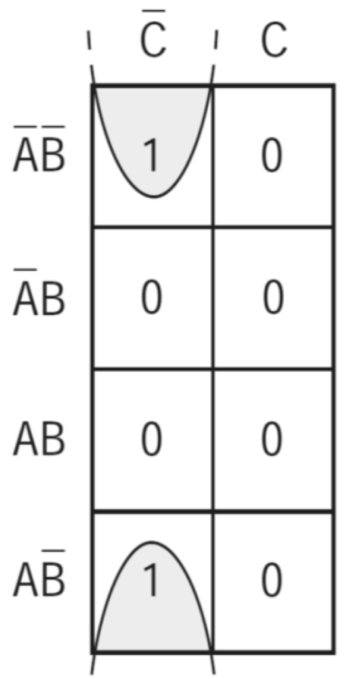
$X = \bar{A}B\bar{C} + AB\bar{C}$
$= B\bar{C}$

A pair of 1 are **vertically** adjacent to each other.



$X = \bar{A}B\bar{C} + \bar{A}BC$
$= \bar{A}B$

A pair of 1 are **horizontally** adjacent to each other.

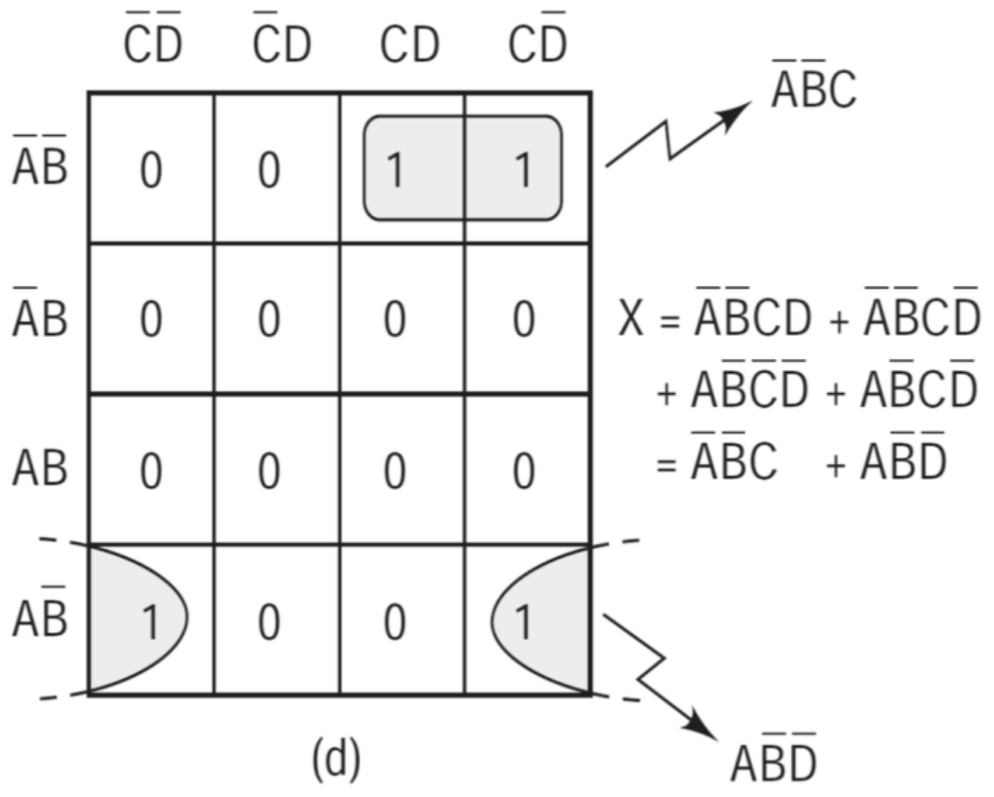


$X = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} = \bar{B}\bar{C}$

In a *K* map, the **top row and bottom row of squares** are considered to be **adjacent**.

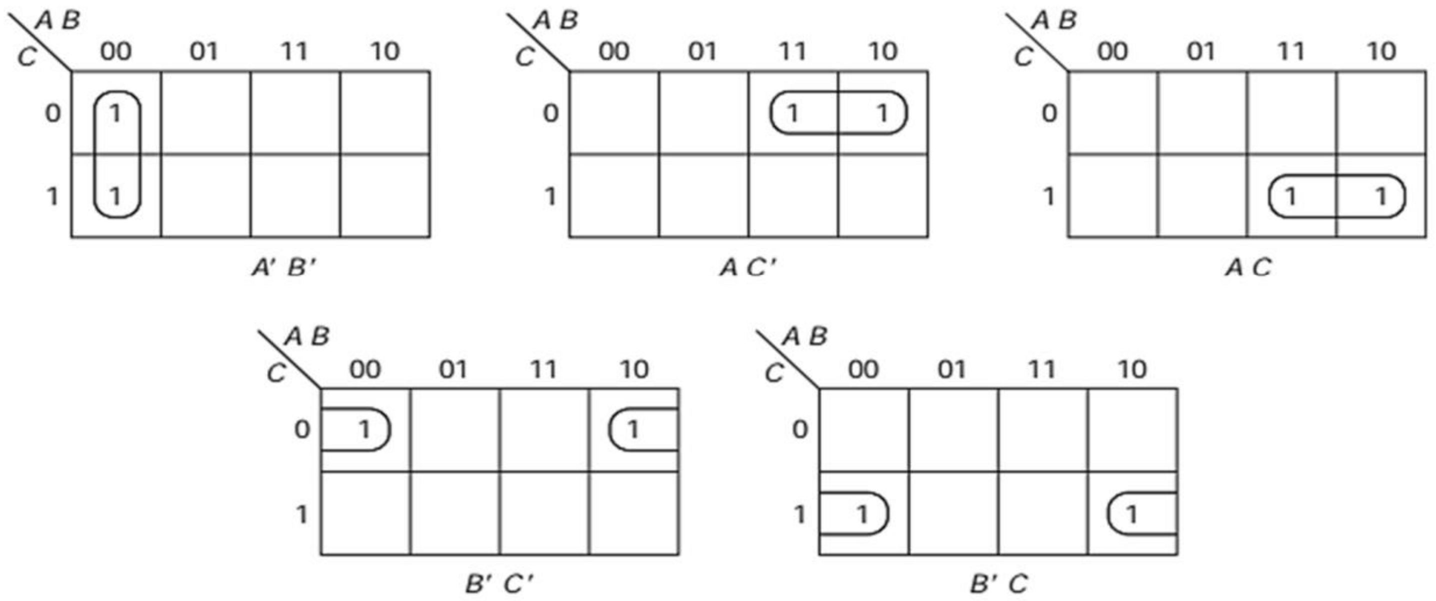
Looping Groups of Two (pairs)

- Looping a pair of adjacent 1s in a K map eliminates the variable that appears in complemented and un-complemented form.
- The two 1s in the top row are horizontally adjacent.
- The leftmost column and the rightmost column of squares are considered to be adjacent.

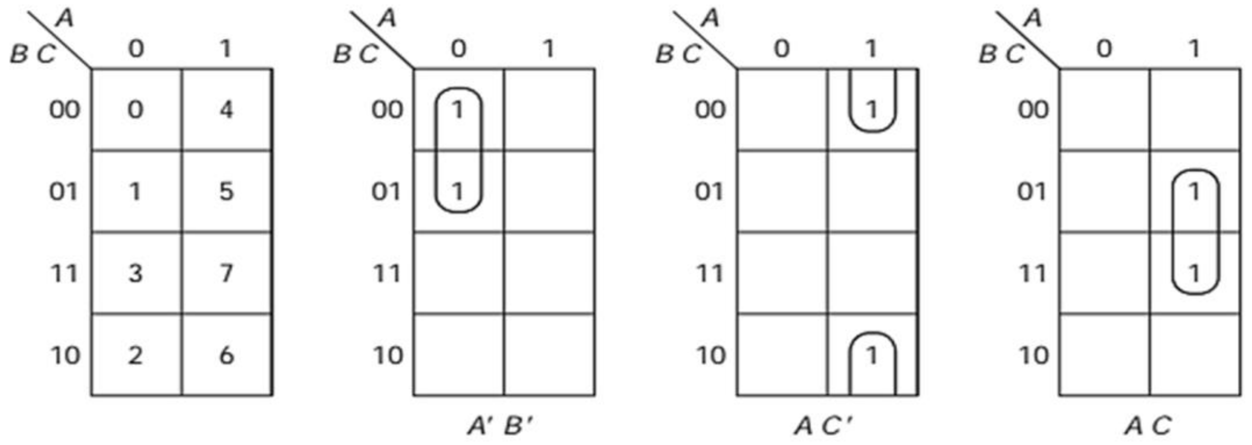


(d)

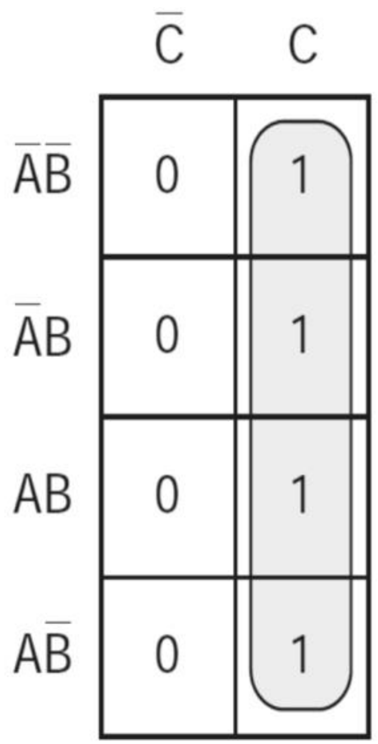
Looping Groups of Two (pairs)



Map 3.6 Vertical orientation of three-variable map.



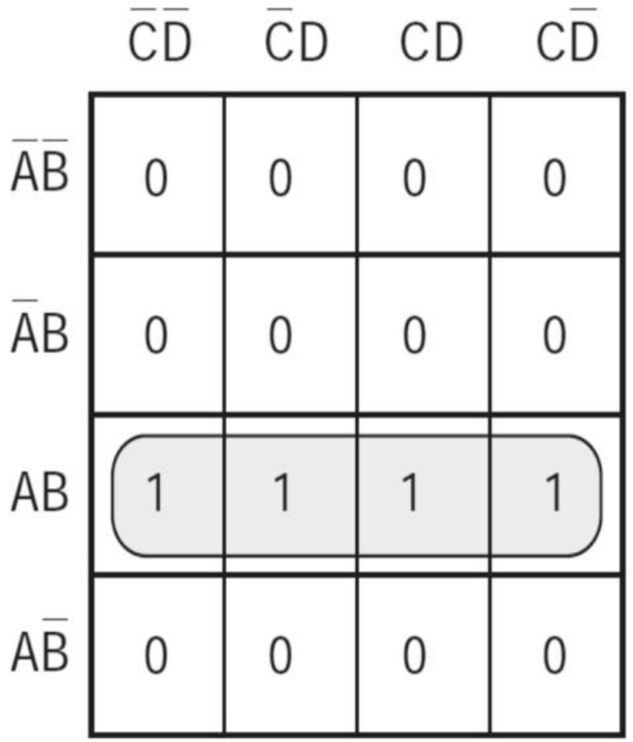
Looping Groups of Four (Quads)



	$\overline{C}$	C
$\overline{A}\overline{B}$	0	1
$\overline{A}B$	0	1
AB	0	1
$A\overline{B}$	0	1

X = C

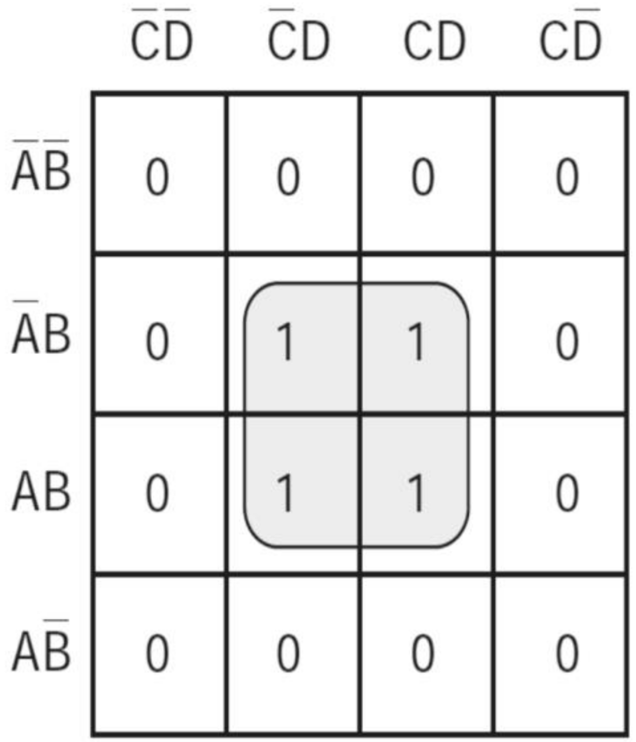
Vertically adjacent



	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	0	0
$\overline{A}B$	0	0	0	0
AB	1	1	1	1
$A\overline{B}$	0	0	0	0

X = AB

Horizontally adjacent



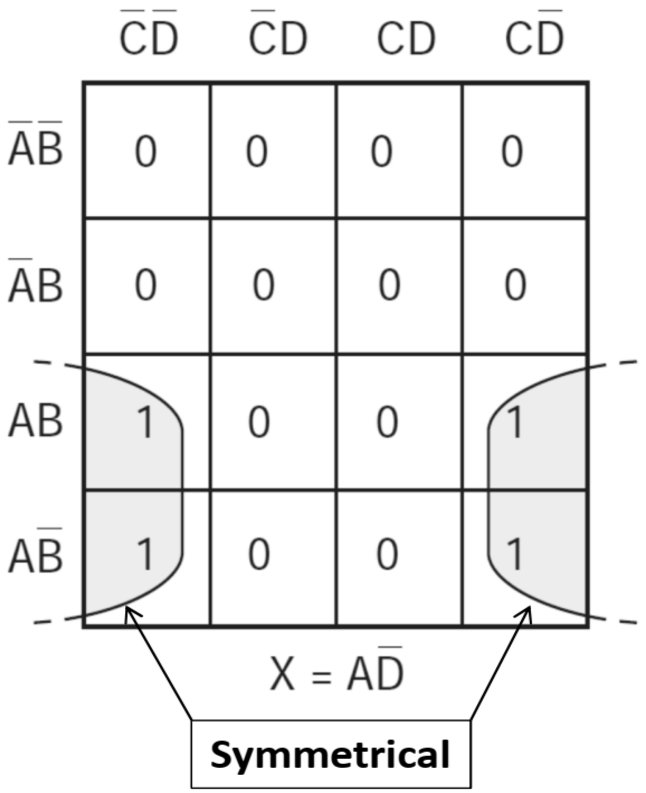
	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	0	0
$\overline{A}B$	0	1	1	0
AB	0	1	1	0
$A\overline{B}$	0	0	0	0

X = BD

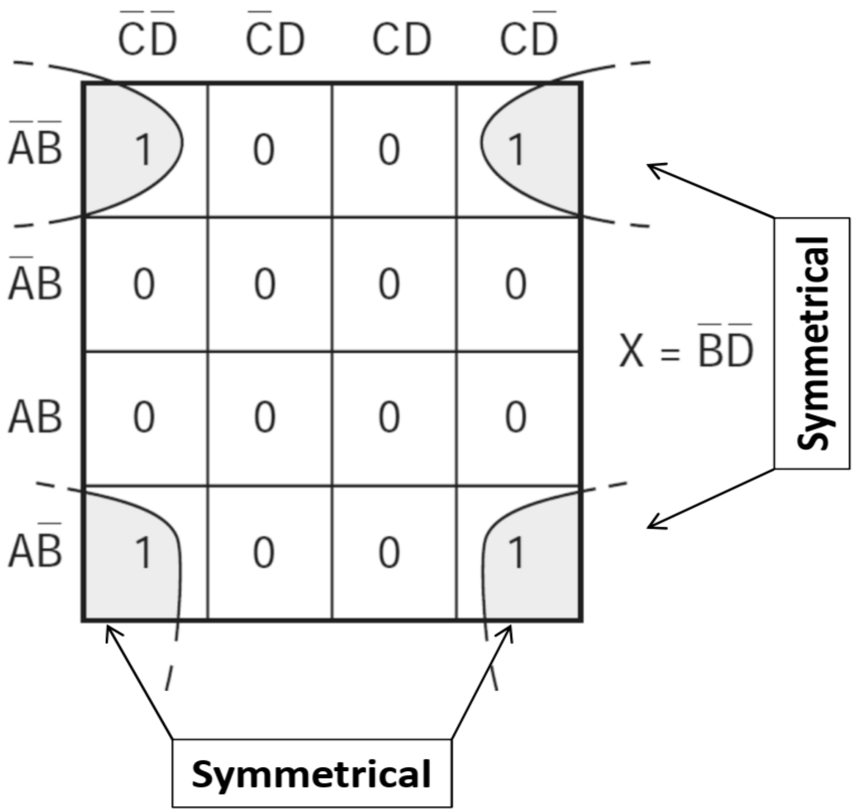
Four 1s in a square

Looping Groups of Four (Quads)

- Looping a quad of adjacent 1s eliminates the two variables that appear in both complemented and un-complemented form.



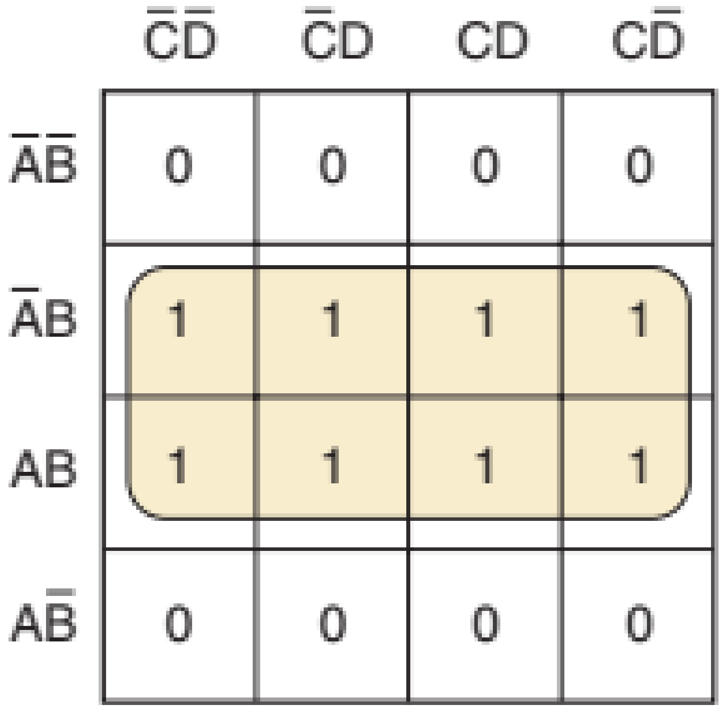
	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	1	0	0	1
$A\bar{B}$	1	0	0	1



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	0	0	1
$\bar{A}B$	0	0	0	0
AB	0	0	0	0
$A\bar{B}$	1	0	0	1

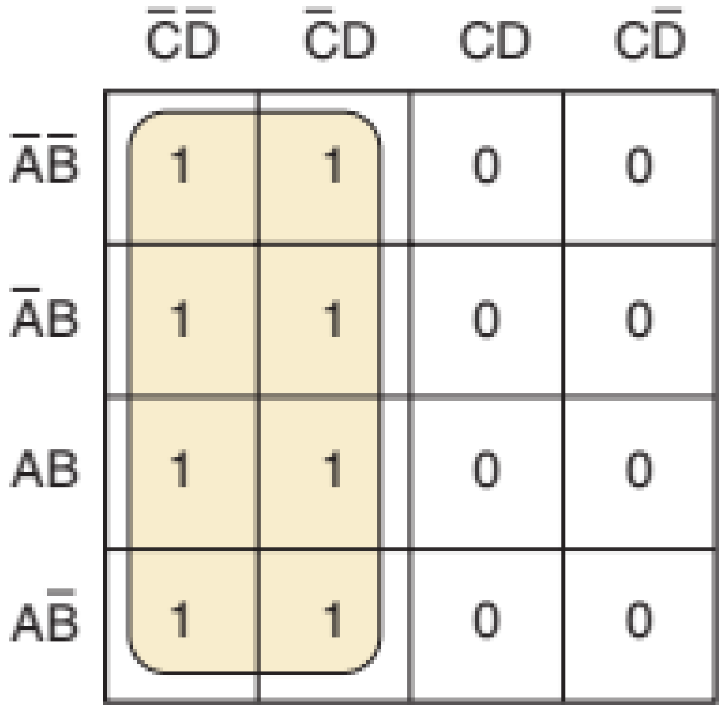
Looping Groups of Eight (Octets)

- Looping an eight of adjacent 1s eliminates the three variables that appear in both complemented and un-complemented form.



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	1	1	1	1
AB	1	1	1	1
$A\bar{B}$	0	0	0	0

$X = B$

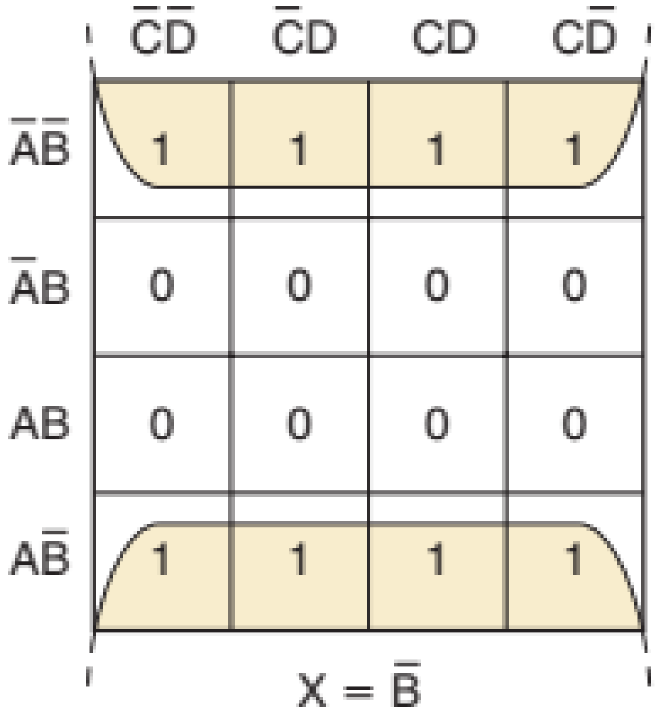


	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	1	0	0
$\bar{A}B$	1	1	0	0
AB	1	1	0	0
$A\bar{B}$	1	1	0	0

$X = \bar{C}$

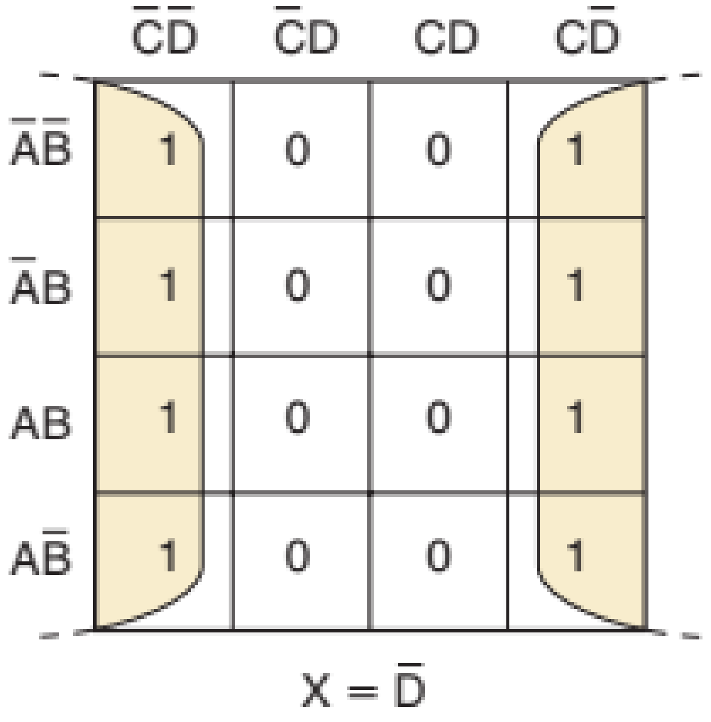
Looping Groups of Eight (Octets)

- Looping an eight of adjacent 1s eliminates the three variables that appear in both complemented and un-complemented form.



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	1	1	1
$\bar{A}B$	0	0	0	0
AB	0	0	0	0
$A\bar{B}$	1	1	1	1

$X = \bar{B}$



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	0	0	1
$\bar{A}B$	1	0	0	1
AB	1	0	0	1
$A\bar{B}$	1	0	0	1

$X = \bar{D}$

Karnaugh Map

- Simplification process

  1. Construct the K map and place 1s in those squares corresponding to the 1s in the truth table. Place 0s in the other squares.

  2. Examine the map for adjacent 1s and loop those 1s that are *not* adjacent to any other 1s. These are called *isolated* 1s.

  3. Next, look for those 1s that are adjacent to only one other 1. Loop any pair containing such a 1.

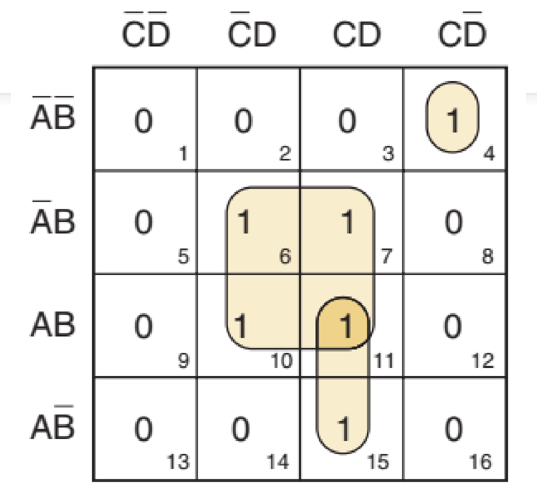
  4. Loop any octet even if it contains some 1s that have already been looped.

  5. Loop any quad that contains one or more 1s that have not already been looped, making sure to use the minimum number of loops.

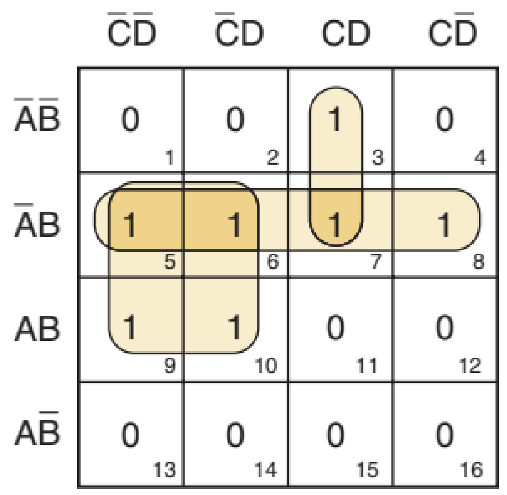
  6. Loop any pairs necessary to include any 1s that have not yet been looped, making sure to use the minimum number of loops.

  7. Form the OR sum of all the terms generated by each loop.

Karnaugh Map

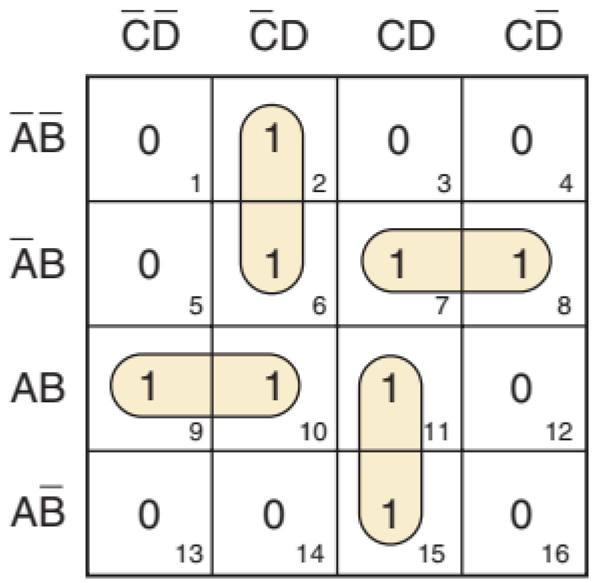


$$X = \underbrace{\bar{A}\bar{B}C\bar{D}}_{\text{loop 4}} + \underbrace{ACD}_{\text{loop 11, 15}} + \underbrace{BD}_{\text{loop 6, 7, 10, 11}}$$



$$X = \underbrace{\bar{A}B}_{\text{loop 5, 6, 7, 8}} + \underbrace{B\bar{C}}_{\text{loop 5, 6, 9, 10}} + \underbrace{\bar{A}CD}_{\text{loop 3, 7}}$$

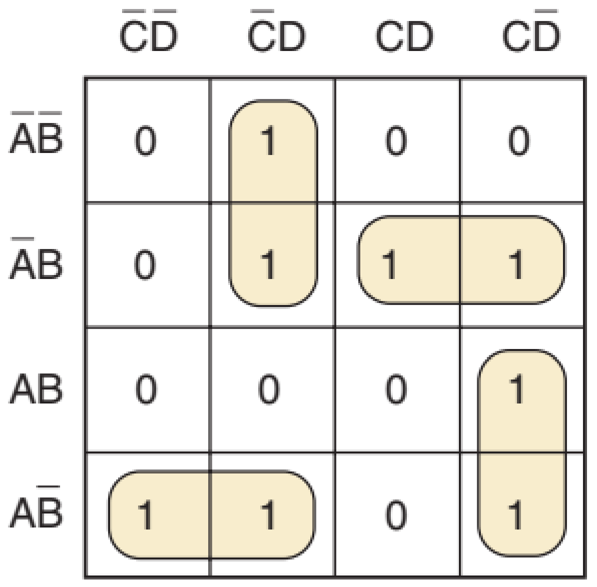
Karnaugh Map



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0_{1}	1_{2}	0_{3}	0_{4}
$\bar{A}B$	0_{5}	1_{6}	1_{7}	1_{8}
AB	1_{9}	1_{10}	1_{11}	0_{12}
$A\bar{B}$	0_{13}	0_{14}	1_{15}	0_{16}

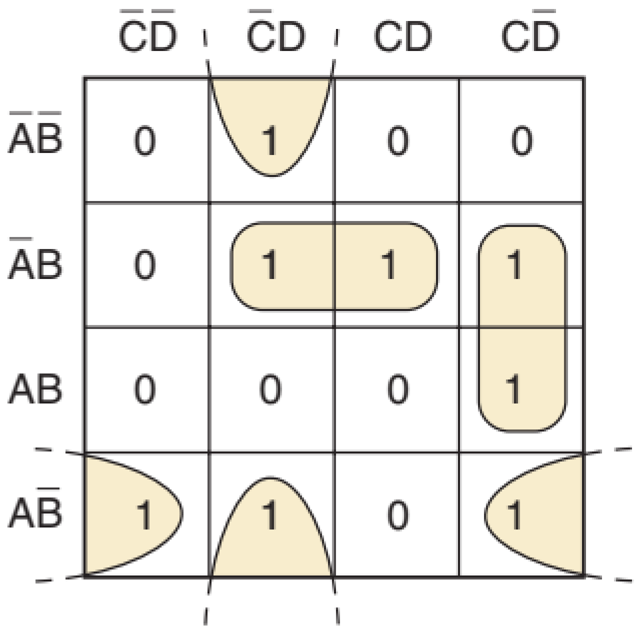
$$X = \underbrace{AB\bar{C}}_{9,\ 10} + \underbrace{\bar{A}\bar{C}D}_{2,\ 6} + \underbrace{\bar{A}BC}_{7,\ 8} + \underbrace{ACD}_{11,\ 15}$$

Karnaugh Map



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	1	1
AB	0	0	0	1
$A\bar{B}$	1	1	0	1

$$X = \bar{A}\bar{C}D + \bar{A}BC + A\bar{B}\bar{C} + AC\bar{D}$$



	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	1	1
AB	0	0	0	1
$A\bar{B}$	1	1	0	1

$$X = \bar{A}BD + BC\bar{D} + \bar{B}\bar{C}D + A\bar{B}\bar{D}$$

Karnaugh Map

- Filling a K Map from an Output Expression
  - 1. Get the expression into SOP form if it is not already in that form.
  - 2. For each product term in the SOP expression, place a 1 in each K-map square whose label contains the same combination of input variables. Place a 0 in all other squares.

Karnaugh Map

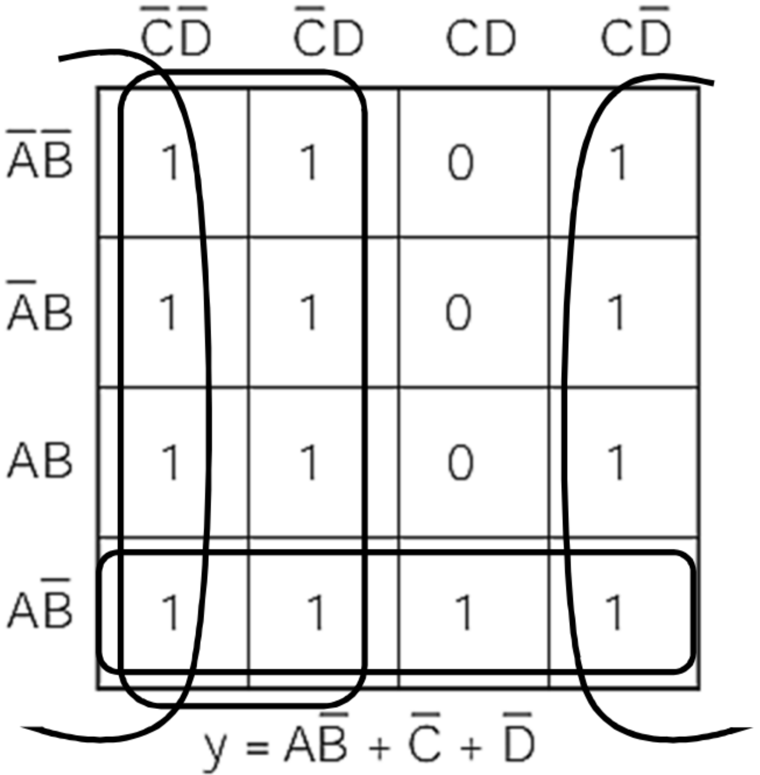
- Use a K map to simplify $y = \overline{C}(\overline{A}\,\overline{B}\,\overline{D} + D) + A\overline{B}C + \overline{D}$.

1. Multiply out the first term to get $y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{C}D + A\overline{B}C + \overline{D}$, which is now in SOP form.
2. For the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ term, simply put a 1 in the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ square of the K map (Figure 4-17). For the $\overline{C}D$ term, place a 1 in all squares with $\overline{C}D$ in their labels, that is, $\overline{A}\,\overline{B}\,\overline{C}D$, $\overline{A}B\overline{C}D$, $AB\overline{C}D$, $A\overline{B}\,\overline{C}D$. For the $A\overline{B}C$ term, place a 1 in all squares that have an $A\overline{B}C$ in their labels, that is, $A\overline{B}C\overline{D}$, $A\overline{B}CD$. For the $\overline{D}$ term, place a 1 in all squares that have a $\overline{D}$ in their labels, that is, all squares in the leftmost and rightmost columns.

	$\overline{C}\,\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\,\overline{B}$	1	1	0	1
$\overline{A}B$	1	1	0	1
AB	1	1	0	1
$A\overline{B}$	1	1	1	1

Karnaugh Map

- Use a K map to simplify $y = \overline{C}(\overline{A}\,\overline{B}\,\overline{D} + D) + A\overline{B}C + \overline{D}.$



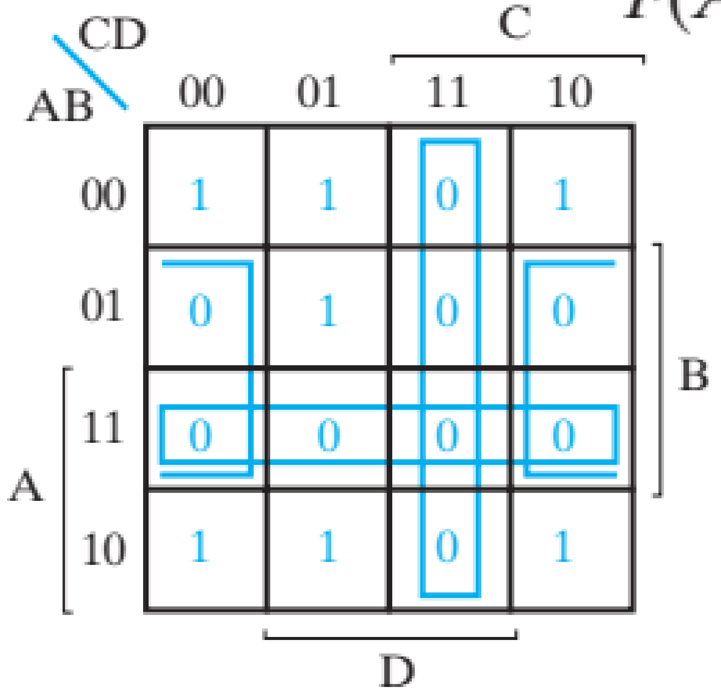
	$\overline{C}\,\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\,\overline{B}$	1	1	0	1
$\overline{A}B$	1	1	0	1
AB	1	1	0	1
$A\overline{B}$	1	1	1	1

$y = A\overline{B} + \overline{C} + \overline{D}$

Karnaugh Map

- Product-of-Sum Optimization
  - Simplify the following Boolean function in product-of-sums form:

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



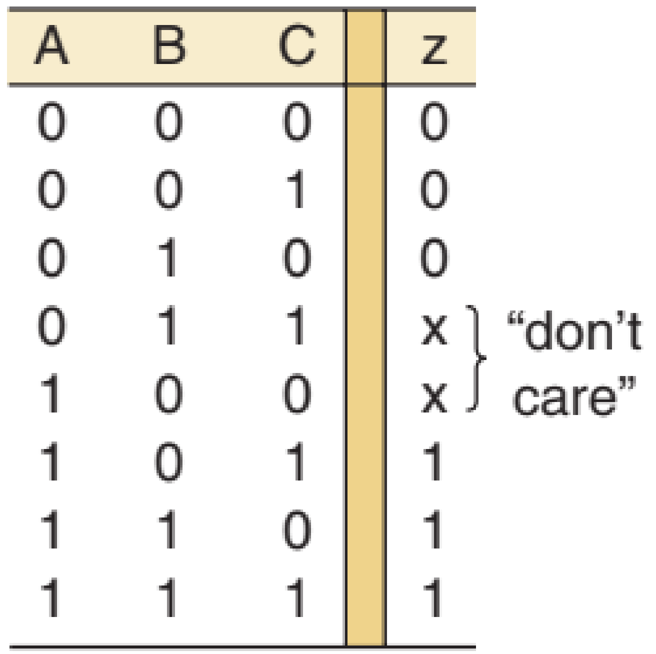
The squares marked with 0s represent the minterms not included in F and therefore denote the complement of F.
Combining the squares marked with 0s, we obtain the optimized complemented function

$$\overline{F} = AB + CD + B\overline{D}$$

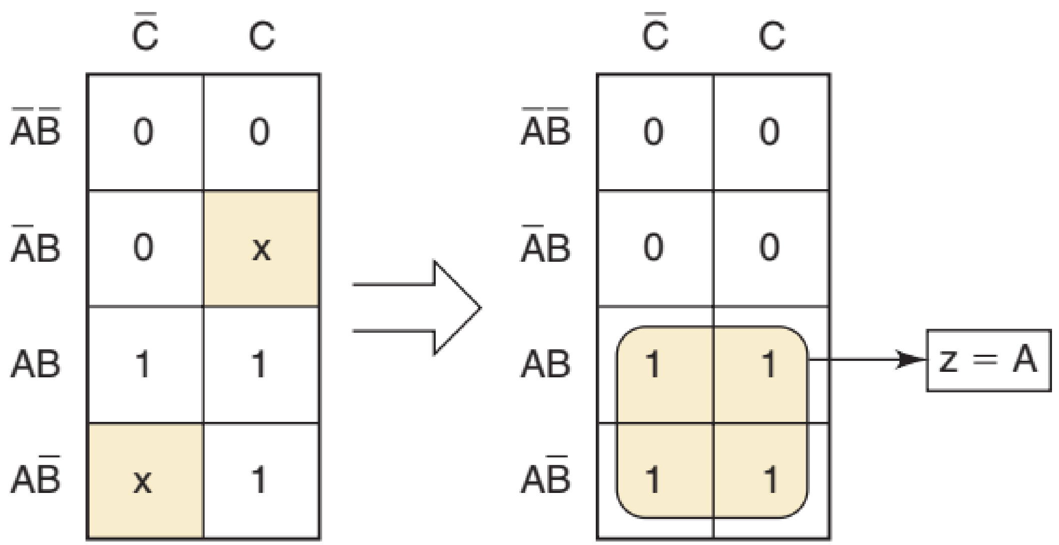
$$F = (\overline{A} + \overline{B})(\overline{C} + \overline{D})(\overline{B} + D)$$

Don’t-Care Conditions

- Some logic circuits can be designed so that we *“don’t care”* whether the output is 1 or 0.
- The **x** represents the don’t-care condition.
- A circuit designer is free to make the output for any don’t-care condition either a 0 or a 1 to produce the simplest output expression.



A	B	C	z	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	x	“don’t care”
1	0	0	x	
1	0	1	1	
1	1	0	1	
1	1	1	1	

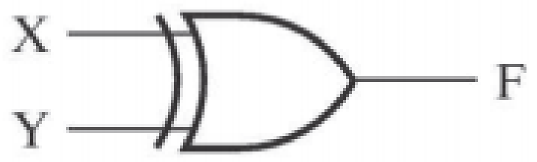


	$\bar{C}$	C
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	0	x
AB	1	1
$A\bar{B}$	x	1

	$\bar{C}$	C
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	0	0
AB	1	1
$A\bar{B}$	1	1

5. Exclusive-Or Operation and Gates

Exclusive-OR (XOR)



$$F = X\overline{Y} + \overline{X}Y = X \oplus Y$$

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

$$X \oplus Y = X\bar{Y} + \bar{X}Y$$

$$\overline{X \oplus Y} = \overline{X\bar{Y} + \bar{X}Y} = (\bar{X} + Y)(X + \bar{Y}) = XY + \bar{X}\bar{Y}$$

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

$$X \oplus 0 = X \qquad X \oplus 1 = \overline{X}$$

$$X \oplus X = 0 \qquad X \oplus \overline{X} = 1$$

$$X \oplus \overline{Y} = \overline{X \oplus Y} \qquad \overline{X} \oplus Y = \overline{X \oplus Y}$$

5. Exclusive-Or Operation and Gates

- Exclusive-OR operation is both commutative and associative:

$$A \oplus B = B \oplus A$$

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

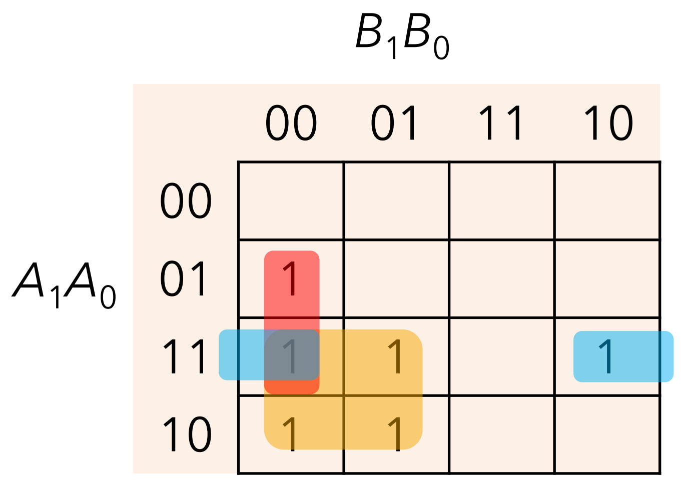
6. HDL Representation - Verilog

- **A Two-bit greater-than comparator** A > B
  - A > B : The output is 1
  - Otherwise: The output is 0

  - $A = A_1A_0$: A 2-digit binary number A
    - A_1: The most significant bit (MSB)
    - A_0: The lest significant bit (LSB)
  - $B = B_1B_0$: A 2-digit binary number B
    - B_1: The most significant bit (MSB)
    - B_0: The lest significant bit (LSB)

A1	A0	B1	B0	A is greater than B	
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**	$\overline{A_1}A_0\overline{B_1}\,\overline{B_0}$
0	1	0	1	**0**	
0	1	1	0	**0**	
0	1	1	1	**0**	
1	0	0	0	**1**	$A_1\overline{A_0}\,\overline{B_1}\,\overline{B_0}$
1	0	0	1	**1**	$A_1\overline{A_0}\,\overline{B_1}\,B_0$
1	0	1	0	**0**	
1	0	1	1	**0**	
1	1	0	0	**1**	$A_1A_0\,\overline{B_1}\,\overline{B_0}$
1	1	0	1	**1**	$A_1A_0\,\overline{B_1}\,B_0$
1	1	1	0	**1**	$A_1A_0\,B_1\overline{B_0}$
1	1	1	1	**0**	

$$A\_greater\_than\_B = \overline{A_1}A_0\overline{B_1}\,\overline{B_0} + A_1\overline{A_0}\,\overline{B_1}\,\overline{B_0} + A_1\overline{A_0}\,\overline{B_1}\,B_0 + A_1A_0\,\overline{B_1}\,\overline{B_0} + A_1A_0\,\overline{B_1}\,B_0 + A_1A_0\,B_1\overline{B_0}$$

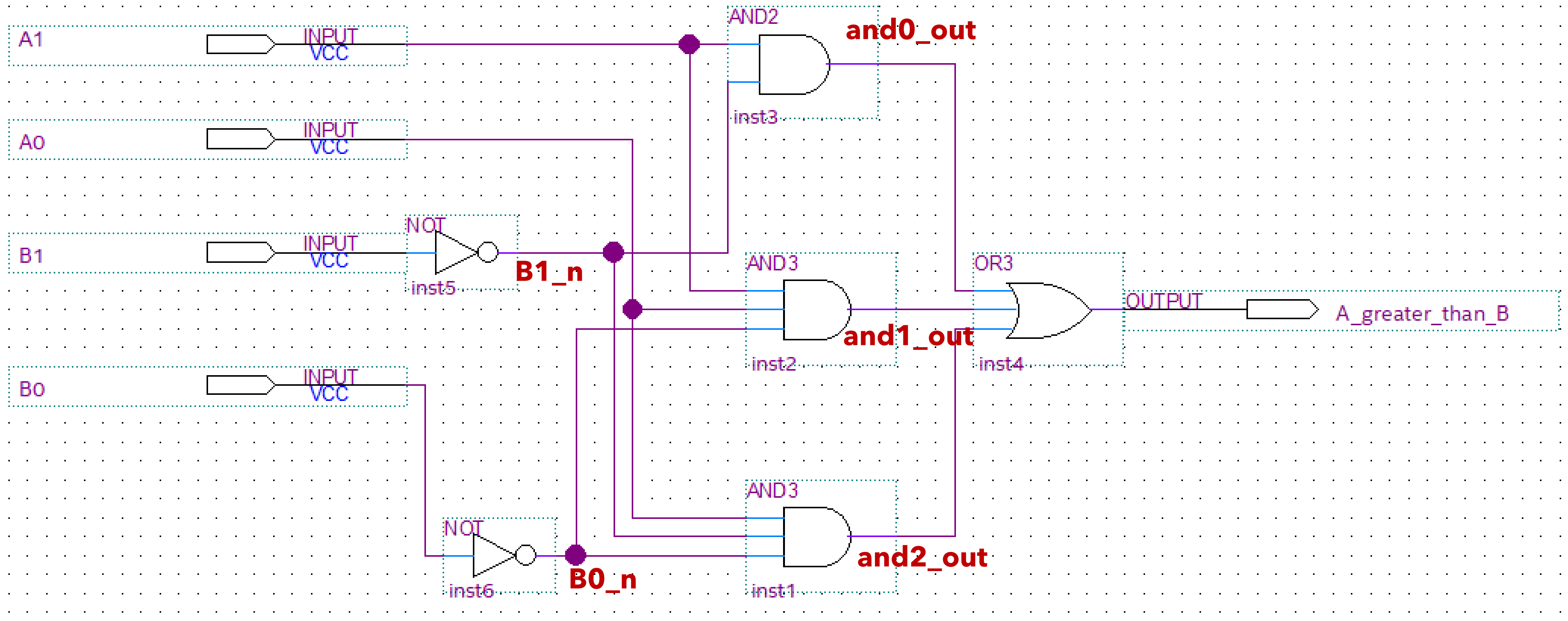


	00	01	11	10
00				
01	1			
11	1	1		1
10	1	1		

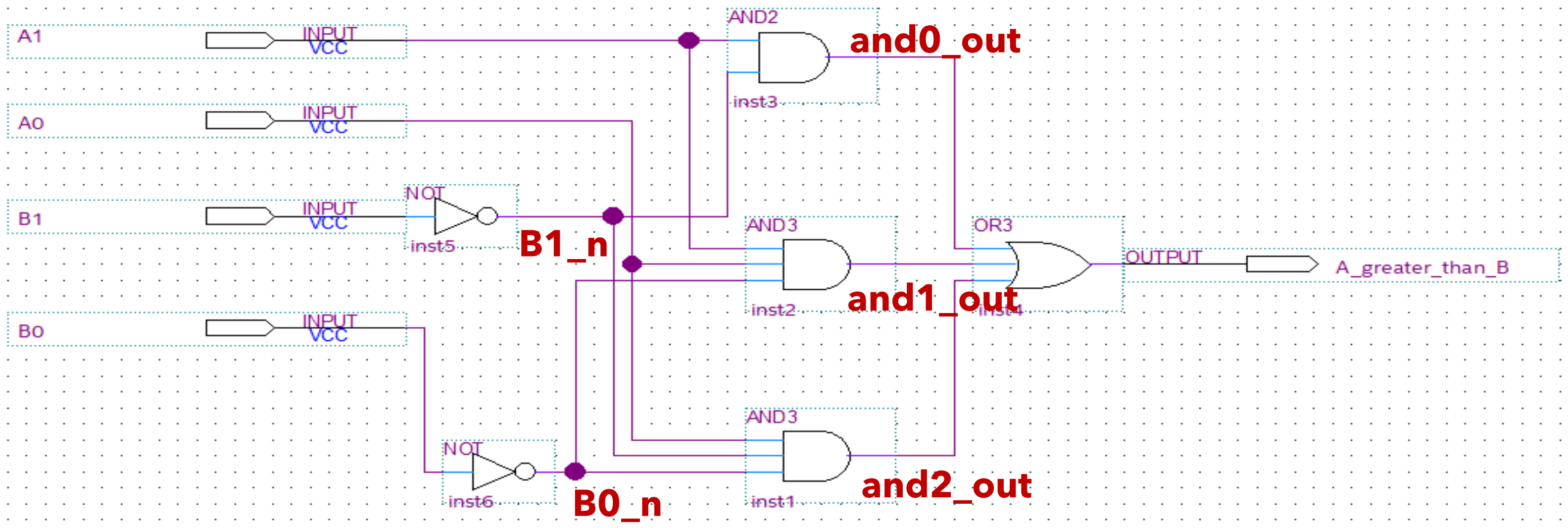
$$A\_greater\_than\_B = A_1\overline{B_1} + A_0\overline{B_1}\,\overline{B_0} + A_1A_0\,\overline{B_0}$$

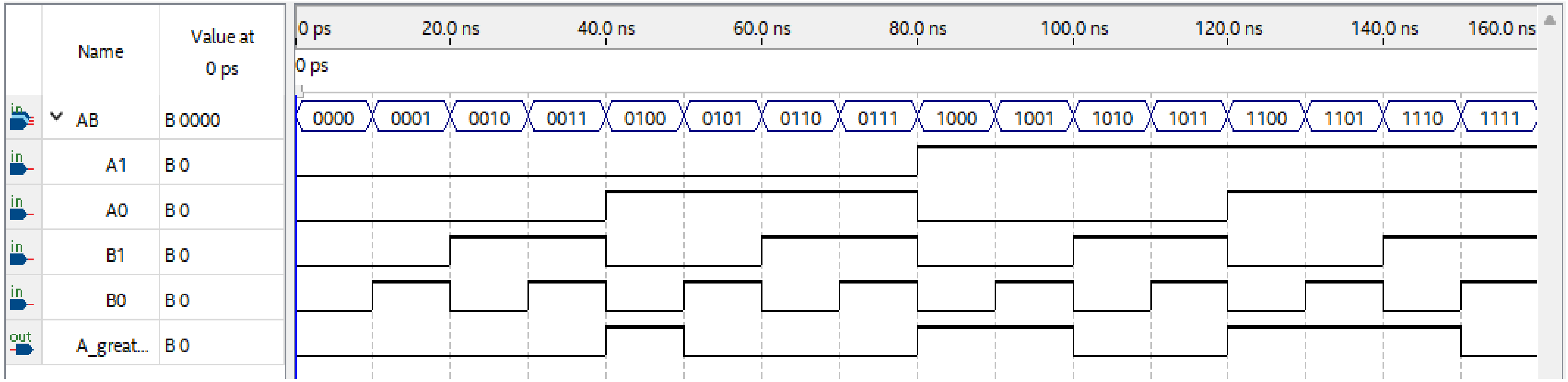
Two-Bit Greater-Than Circuit

$$A\_greater\_than\_B = A_1 \overline{B_1} + A_0 \overline{B_1}\,\overline{B_0} + A_1 A_0 \overline{B_0}$$



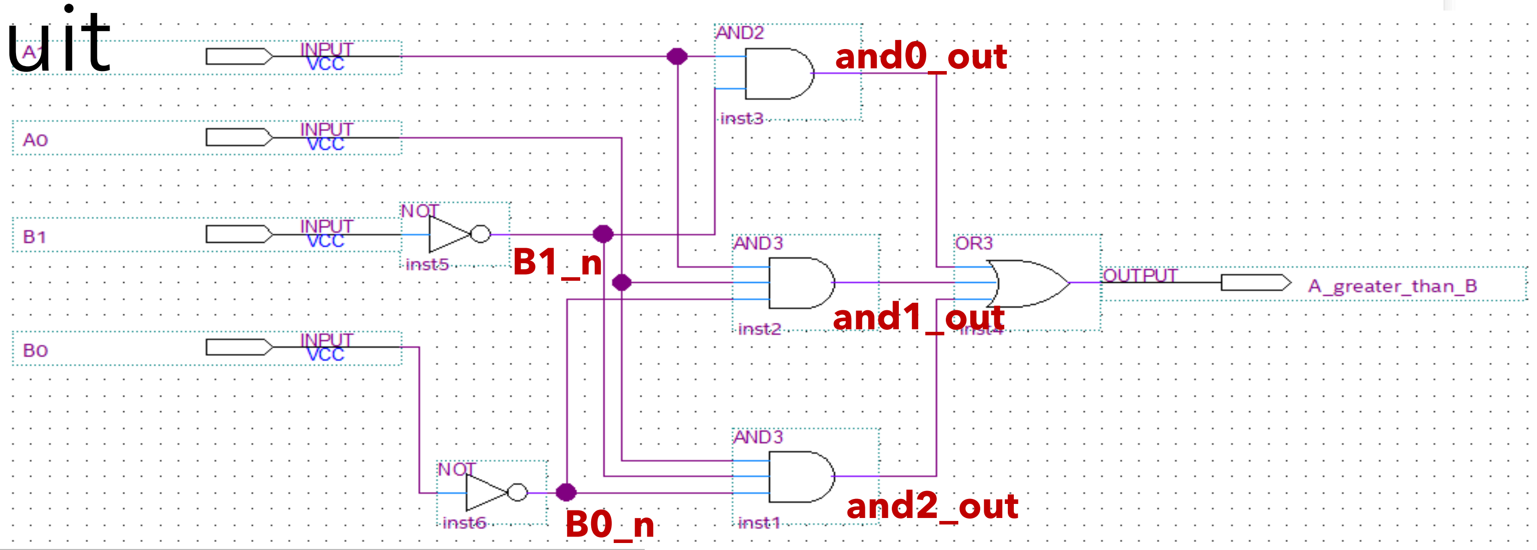
Two-Bit Greater-Than Circuit

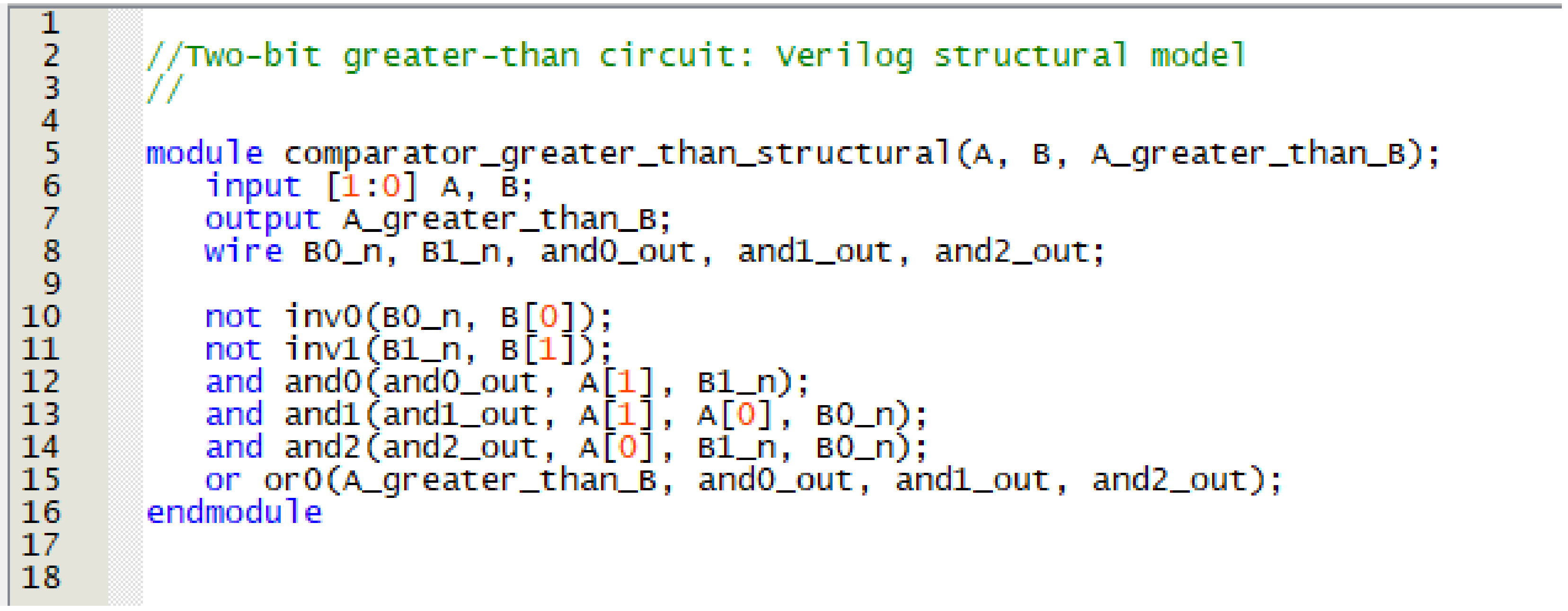




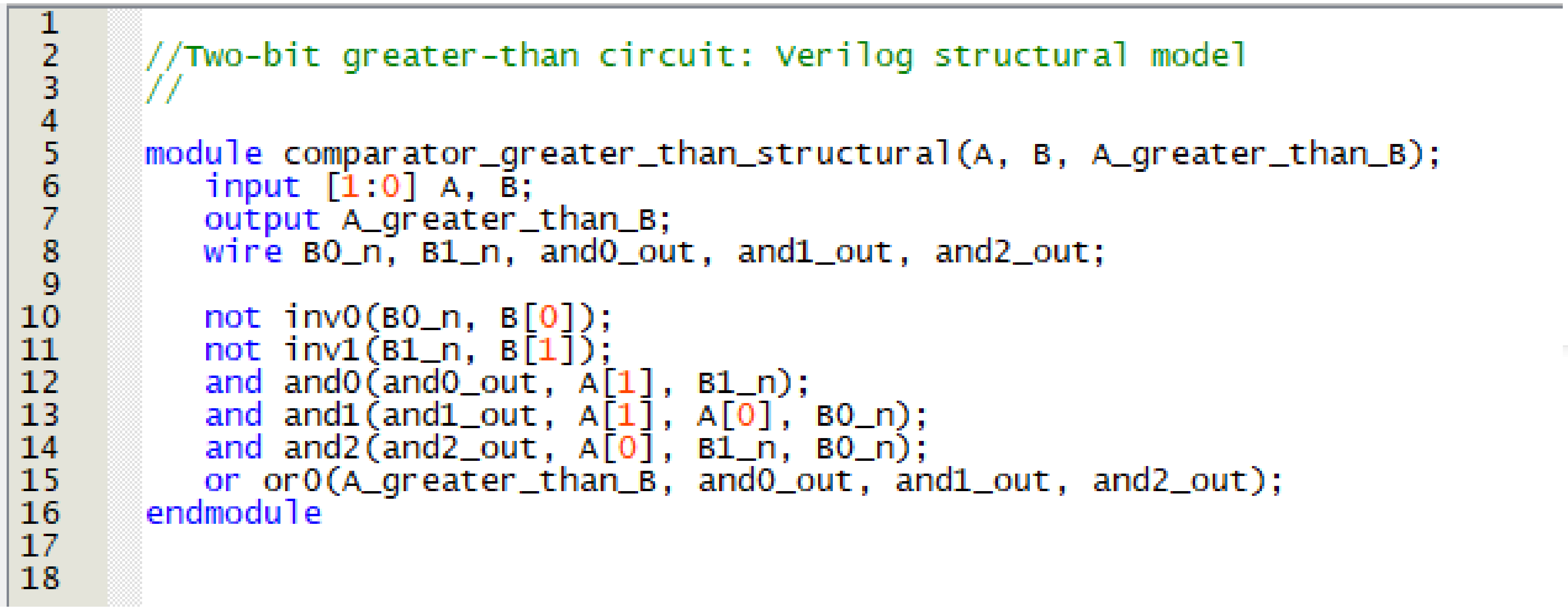
Structural Verilog Description of Two-Bit Greater-Than Circuit

- Equivalent to the circuit schematic
- Gate-level modeling





```
//Two-bit greater-than circuit: Verilog structural model
//

module comparator_greater_than_structural(A, B, A_greater_than_B);
   input [1:0] A, B;
   output A_greater_than_B;
   wire B0_n, B1_n, and0_out, and1_out, and2_out;

   not inv0(B0_n, B[0]);
   not inv1(B1_n, B[1]);
   and and0(and0_out, A[1], B1_n);
   and and1(and1_out, A[1], A[0], B0_n);
   and and2(and2_out, A[0], B1_n, B0_n);
   or or0(A_greater_than_B, and0_out, and1_out, and2_out);
endmodule
```



```
//Two-bit greater-than circuit: Verilog structural model
//

module comparator_greater_than_structural(A, B, A_greater_than_B);
   input [1:0] A, B;
   output A_greater_than_B;
   wire B0_n, B1_n, and0_out, and1_out, and2_out;

   not inv0(B0_n, B[0]);
   not inv1(B1_n, B[1]);
   and and0(and0_out, A[1], B1_n);
   and and1(and1_out, A[1], A[0], B0_n);
   and and2(and2_out, A[0], B1_n, B0_n);
   or or0(A_greater_than_B, and0_out, and1_out, and2_out);
endmodule
```

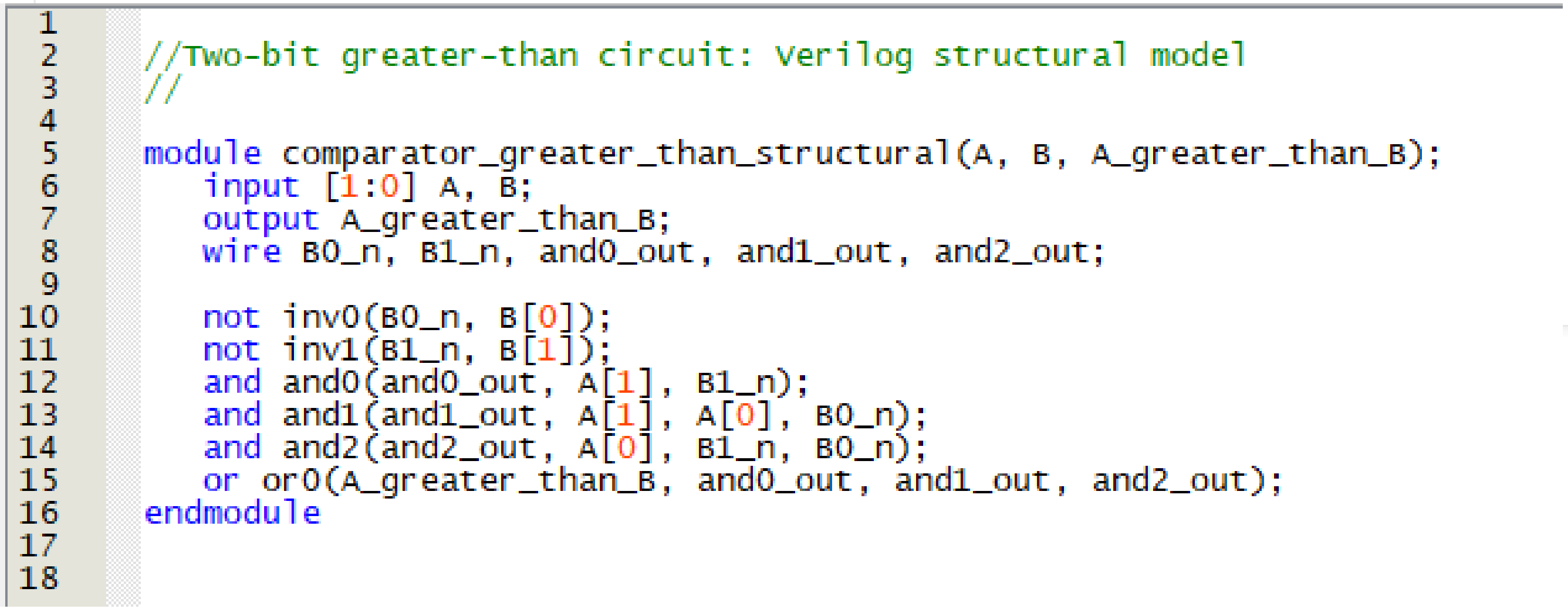
- **//** : Single line comment
- **/*** **..** : Multiline comment

***/**

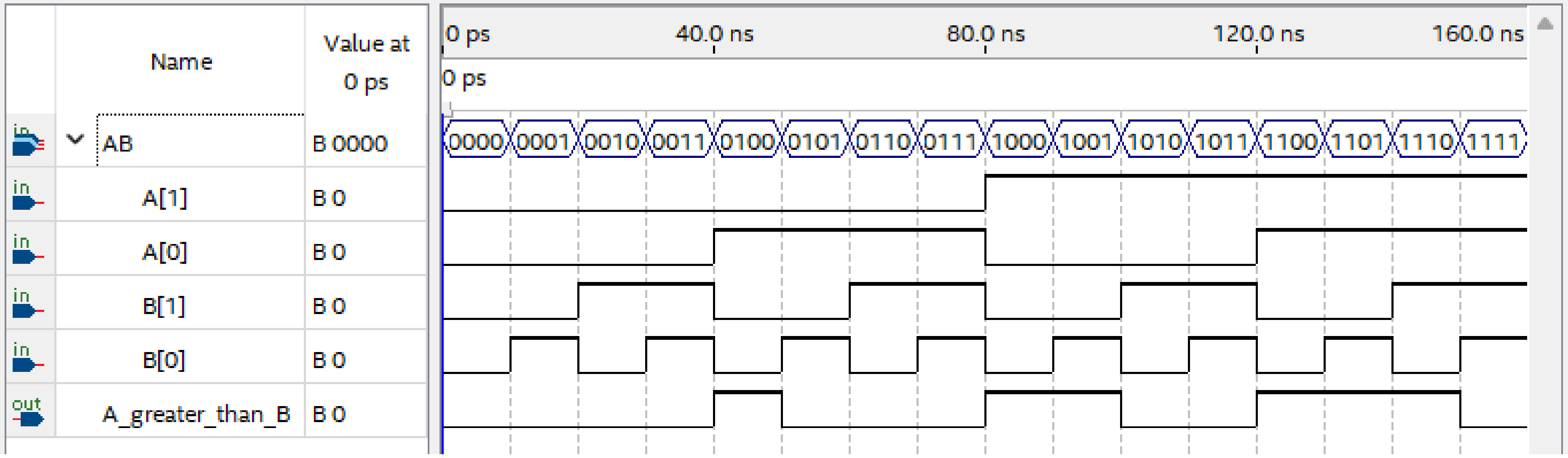
- **input [1:0] A, B;**
  - A and B are vectors with a width of two, with the most significant (leftmost) bit numbered 1 and least significant (rightmost) bit numbered 0.

A[1]
A[0]
A
B[1]
B[0]
B

- In Verilog, **wire** is the default net type. Notably, input and output ports have the default type **wire**.



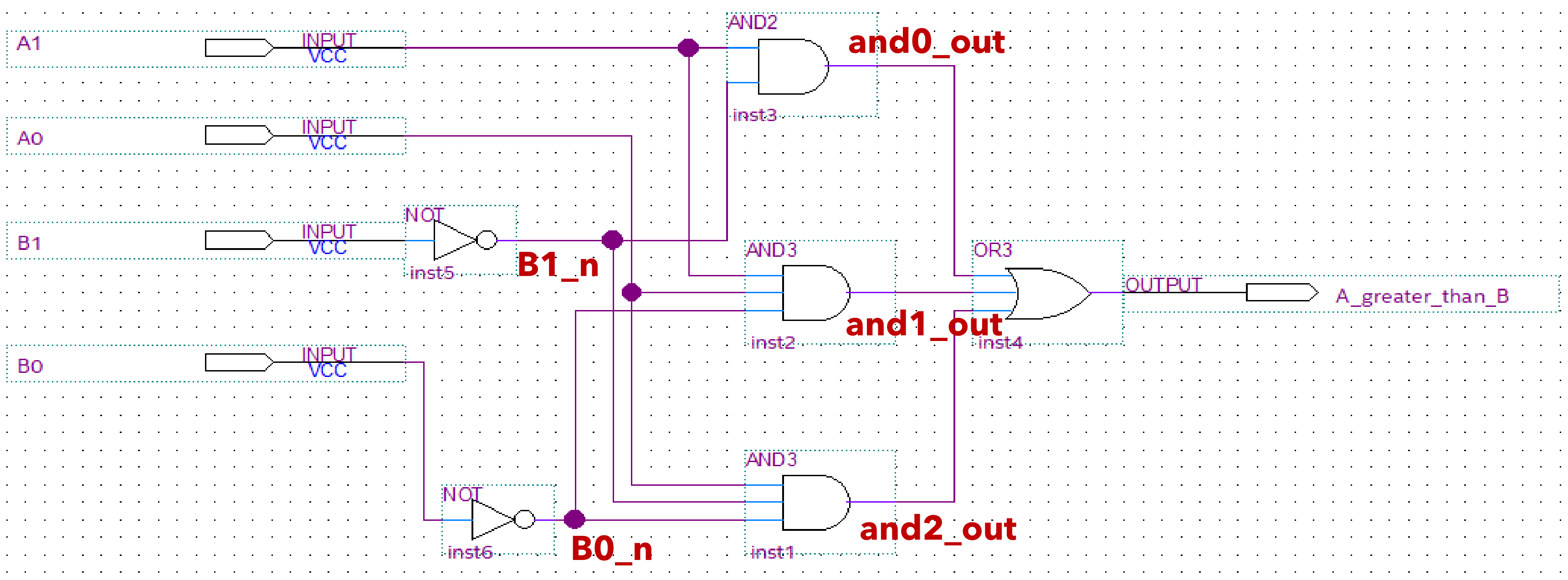
```verilog
//Two-bit greater-than circuit: Verilog structural model
//

module comparator_greater_than_structural(A, B, A_greater_than_B);
   input [1:0] A, B;
   output A_greater_than_B;
   wire B0_n, B1_n, and0_out, and1_out, and2_out;

   not inv0(B0_n, B[0]);
   not inv1(B1_n, B[1]);
   and and0(and0_out, A[1], B1_n);
   and and1(and1_out, A[1], A[0], B0_n);
   and and2(and2_out, A[0], B1_n, B0_n);
   or or0(A_greater_than_B, and0_out, and1_out, and2_out);
endmodule
```

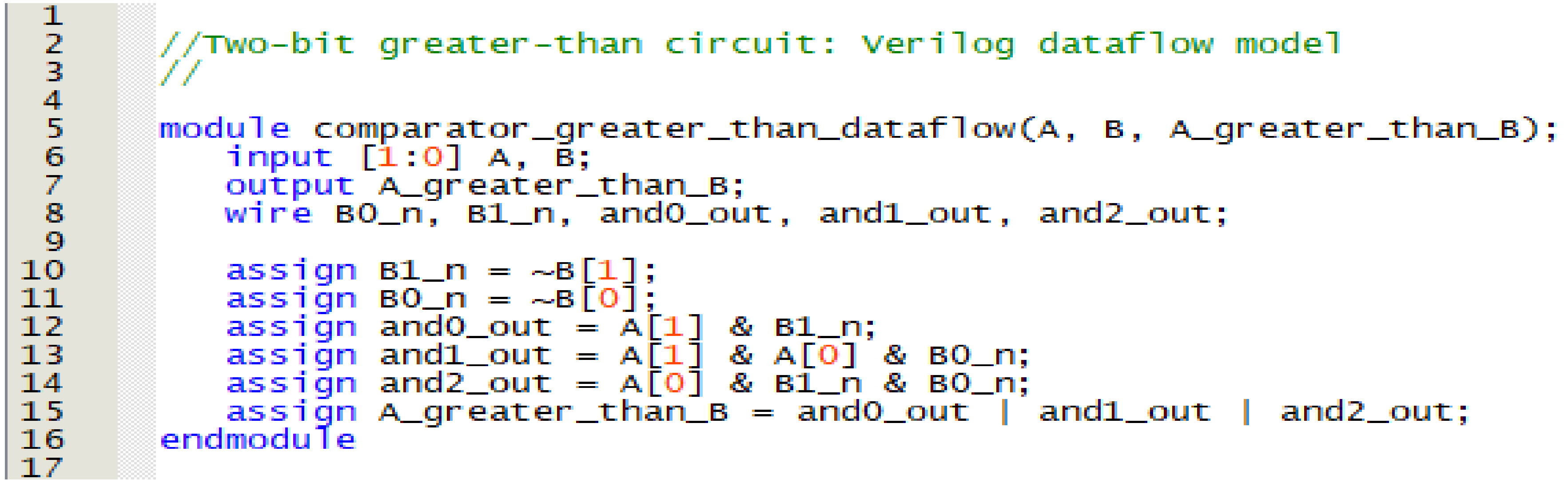


Dataflow Verilog Description

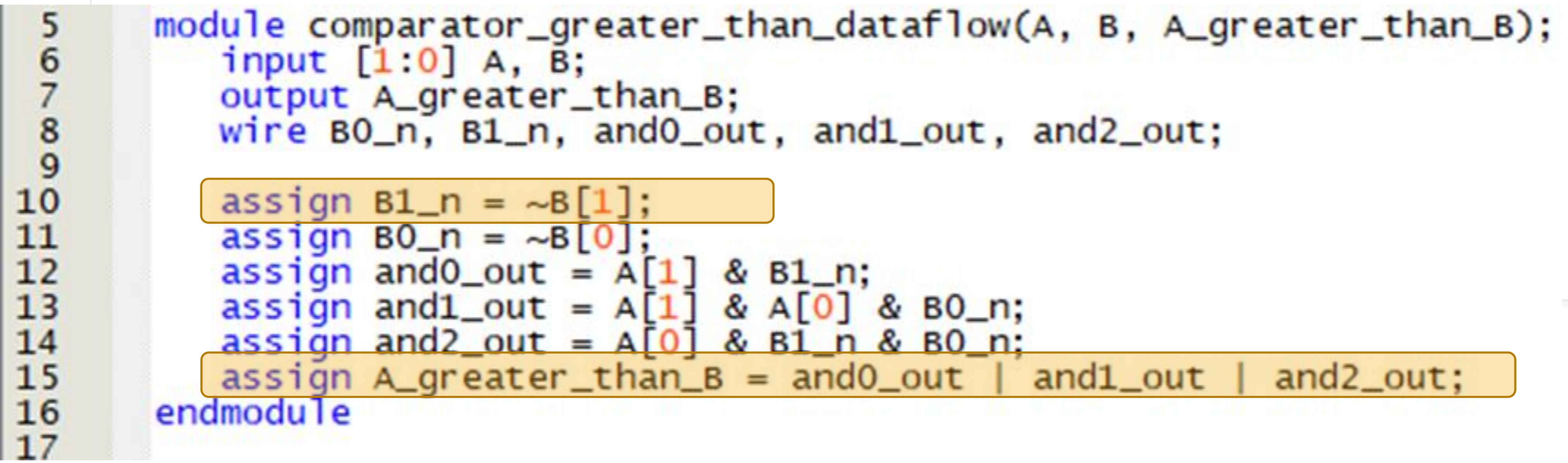
- A dataflow description describes a circuit in terms of function rather than structure and is made up of concurrent assignment statements or their equivalent.
- Concurrent assignment statements are executed concurrently (i.e., in parallel) whenever one of the values on the right-hand side of the statement changes.
  - For example, whenever a change occurs in a value on the right-hand side of a Boolean equation, the left-hand side is evaluated.

- Dataflow Modeling



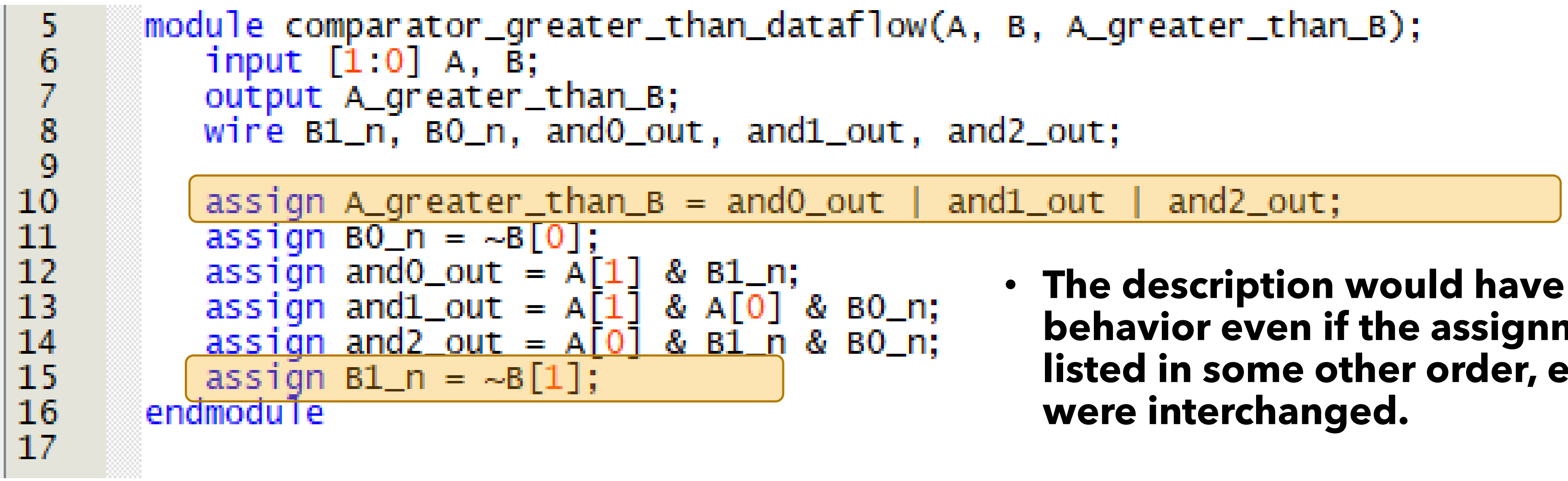


```
//Two-bit greater-than circuit: Verilog dataflow model
//

module comparator_greater_than_dataflow(A, B, A_greater_than_B);
   input [1:0] A, B;
   output A_greater_than_B;
   wire B0_n, B1_n, and0_out, and1_out, and2_out;

   assign B1_n = ~B[1];
   assign B0_n = ~B[0];
   assign and0_out = A[1] & B1_n;
   assign and1_out = A[1] & A[0] & B0_n;
   assign and2_out = A[0] & B1_n & B0_n;
   assign A_greater_than_B = and0_out | and1_out | and2_out;
endmodule
```



```
 5  module comparator_greater_than_dataflow(A, B, A_greater_than_B);
 6     input [1:0] A, B;
 7     output A_greater_than_B;
 8     wire B0_n, B1_n, and0_out, and1_out, and2_out;
 9
10     assign B1_n = ~B[1];
11     assign B0_n = ~B[0];
12     assign and0_out = A[1] & B1_n;
13     assign and1_out = A[1] & A[0] & B0_n;
14     assign and2_out = A[0] & B1_n & B0_n;
15     assign A_greater_than_B = and0_out | and1_out | and2_out;
16  endmodule
17
```

- **The order of execution of the assignment statements does not depend upon the order of their appearance in the model description, but rather on the order of changes of signals on the right-hand side of the assignment statements.**



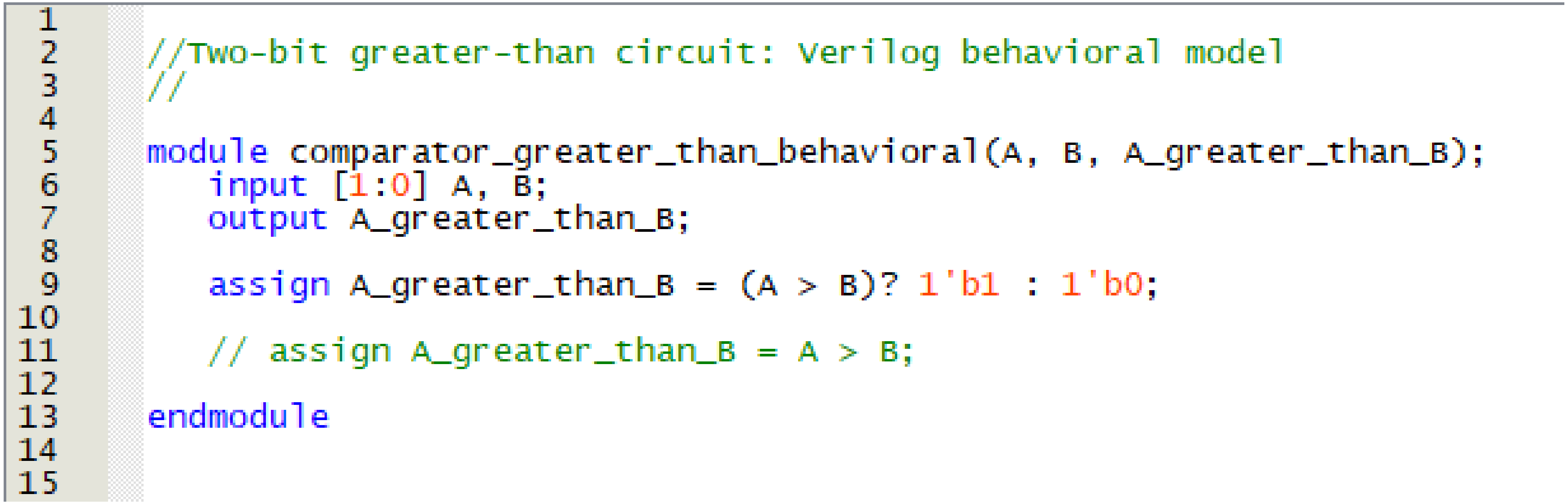
```
 5  module comparator_greater_than_dataflow(A, B, A_greater_than_B);
 6     input [1:0] A, B;
 7     output A_greater_than_B;
 8     wire B1_n, B0_n, and0_out, and1_out, and2_out;
 9
10     assign A_greater_than_B = and0_out | and1_out | and2_out;
11     assign B0_n = ~B[0];
12     assign and0_out = A[1] & B1_n;
13     assign and1_out = A[1] & A[0] & B0_n;
14     assign and2_out = A[0] & B1_n & B0_n;
15     assign B1_n = ~B[1];
16  endmodule
17
```

- **The description would have exactly the same behavior even if the assignment statements were listed in some other order, e.g., if lines 10 and 15 were interchanged.**

Behavioral Verilog Description

- Dataflow models using concurrent assignments are considered to be behavioral descriptions, because they describe the function of the circuit without describing its structure.
- Verilog provides ways to describe circuits at levels higher than the logic level.

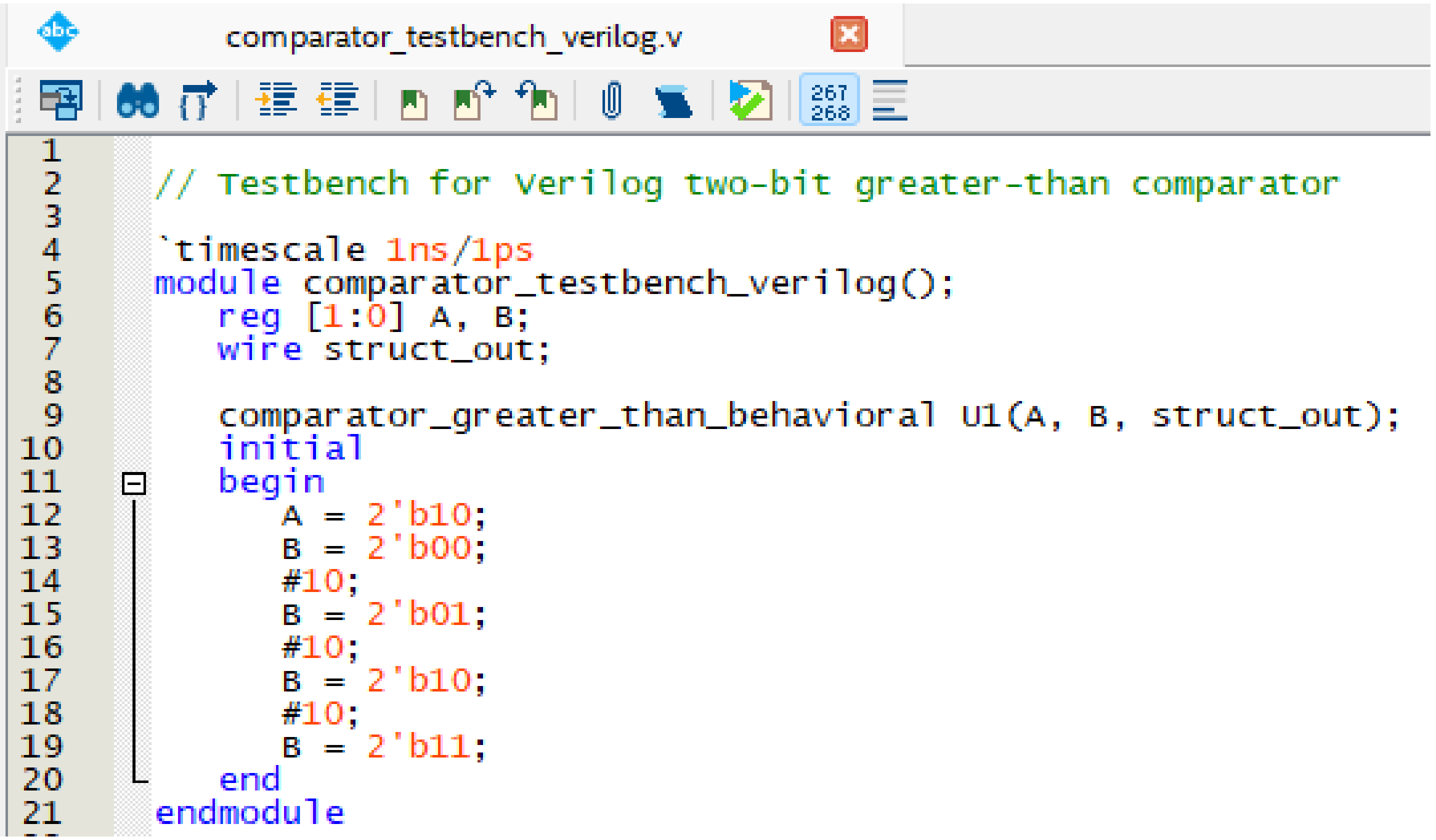
Behavioral Verilog Description



```verilog

//Two-bit greater-than circuit: Verilog behavioral model
//

module comparator_greater_than_behavioral(A, B, A_greater_than_B);
   input [1:0] A, B;
   output A_greater_than_B;

   assign A_greater_than_B = (A > B)? 1'b1 : 1'b0;

   // assign A_greater_than_B = A > B;

endmodule


```

Testbenches

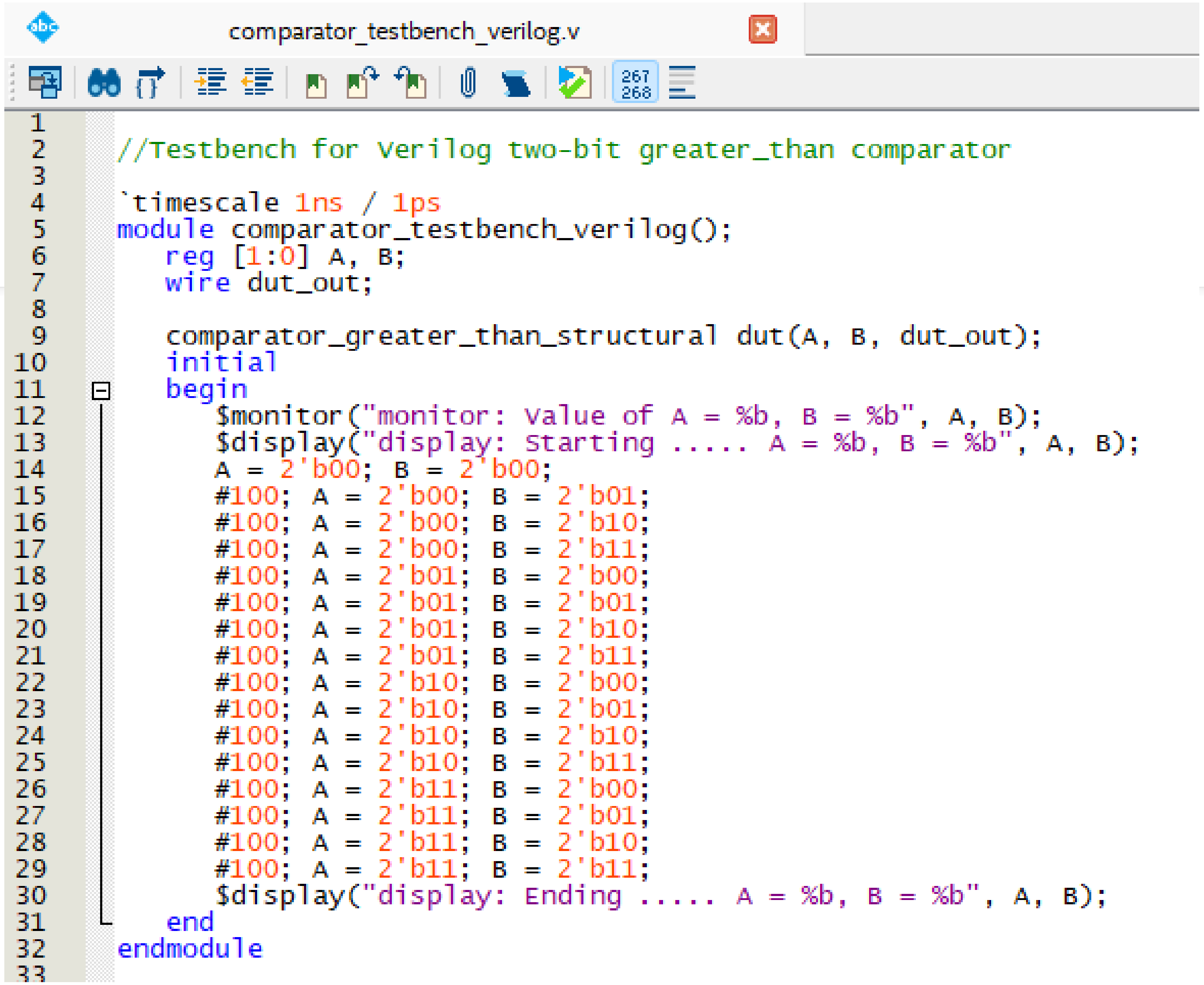
- A testbench is an HDL model whose purpose is to test another model, often called the *Device Under Test* (DUT), by applying stimuli to the inputs.
  - More complex testbenches will also analyze the output of the DUT for correctness.



```

// Testbench for Verilog two-bit greater-than comparator

`timescale 1ns/1ps
module comparator_testbench_verilog();
   reg [1:0] A, B;
   wire struct_out;

   comparator_greater_than_behavioral U1(A, B, struct_out);
   initial
   begin
      A = 2'b10;
      B = 2'b00;
      #10;
      B = 2'b01;
      #10;
      B = 2'b10;
      #10;
      B = 2'b11;
   end
endmodule
```

Testbenches

- **$display** : print the immediate values
- **$strobe** : print the values at the end of the current timestep
- **$monitor** : print the values at the end of the current timestep if any values changed.
    - $monitor can only be called once; sequential call will override the previous.



```
//Testbench for Verilog two-bit greater_than comparator

`timescale 1ns / 1ps
module comparator_testbench_verilog();
   reg [1:0] A, B;
   wire dut_out;

   comparator_greater_than_structural dut(A, B, dut_out);
   initial
   begin
      $monitor("monitor: Value of A = %b, B = %b", A, B);
      $display("display: Starting ..... A = %b, B = %b", A, B);
      A = 2'b00; B = 2'b00;
      #100; A = 2'b00; B = 2'b01;
      #100; A = 2'b00; B = 2'b10;
      #100; A = 2'b00; B = 2'b11;
      #100; A = 2'b01; B = 2'b00;
      #100; A = 2'b01; B = 2'b01;
      #100; A = 2'b01; B = 2'b10;
      #100; A = 2'b01; B = 2'b11;
      #100; A = 2'b10; B = 2'b00;
      #100; A = 2'b10; B = 2'b01;
      #100; A = 2'b10; B = 2'b10;
      #100; A = 2'b10; B = 2'b11;
      #100; A = 2'b11; B = 2'b00;
      #100; A = 2'b11; B = 2'b01;
      #100; A = 2'b11; B = 2'b10;
      #100; A = 2'b11; B = 2'b11;
      $display("display: Ending ..... A = %b, B = %b", A, B);
   end
endmodule
```