
Logic and Computer Design Fundamentals

Chapter 2 – Combinational Logic Circuits

Part 2 – Circuit Optimization

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology,
Zhejiang University

Overview

- **Part 1 – Gate Circuits and Boolean Equations**
 - Binary Logic and Gates
 - Boolean Algebra
 - Standard Forms
- **Part 2 – Circuit Optimization**
 - Two-Level Optimization
 - Map Manipulation
 - Multi-Level Circuit Optimization
- **Part 3 – Additional Gates and Circuits**
 - Other Gate Types
 - Exclusive-OR Operator and Gates
- **Part 4 – HDLs overview**
 - Logic Synthesis
 - HDL Representations—Verilog

Circuit Optimization

- **Goal: To obtain the simplest implementation for a given function**
- **Optimization is a more formal approach to simplification that is performed using a specific procedure or algorithm**
- **Optimization requires a cost criterion to measure the simplicity of a circuit**
- **Distinct cost criteria we will use:**
 - Literal cost (L)
 - Gate input cost (G)
 - Gate input cost with NOTs (GN)

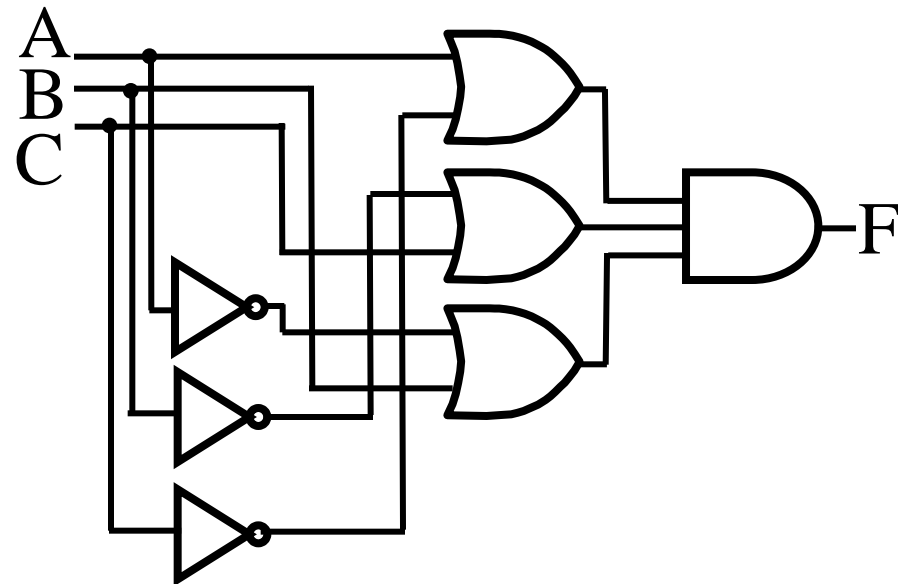
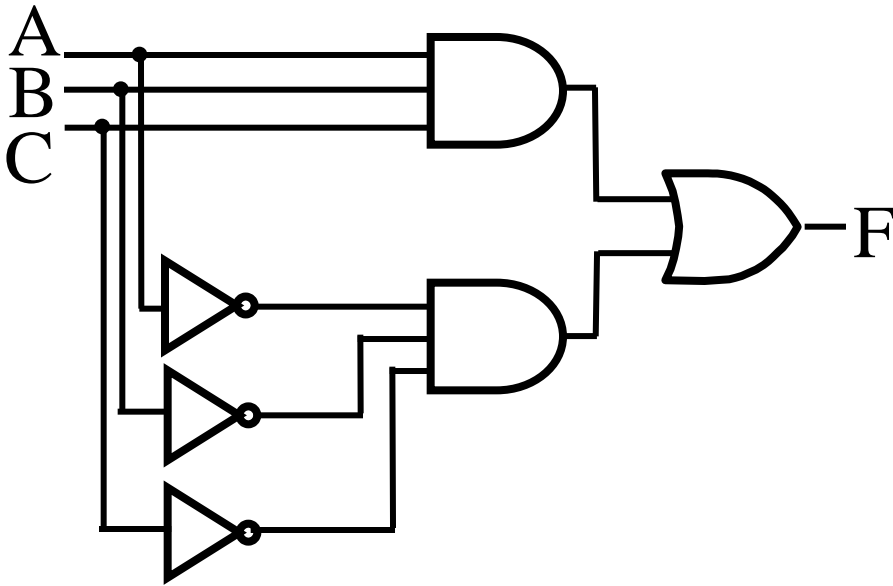
Literal Cost

- **Literal** – a variable or its complement
- **Literal cost** – the number of literal appearances in a Boolean expression corresponding to the logic circuit diagram
- **Examples:**
 - $F = BD + A\bar{B}C + A\bar{C}\bar{D}$ $L = 8$
 - $F = BD + A\bar{B}C + A\bar{B}\bar{D} + AB\bar{C}$ $L =$
 - $F = (A + B)(A + D)(B + C + \bar{D})(\bar{B} + \bar{C} + D)$ $L =$
 - Which solution is best?

Literal Cost

■ Another Example:

- $F = ABC + \bar{A}\bar{B}\bar{C}$ $L = 6$
- $F = (A + \bar{C})(\bar{B} + C)(\bar{A} + B)$ $L = 6$
- Which solution is best?

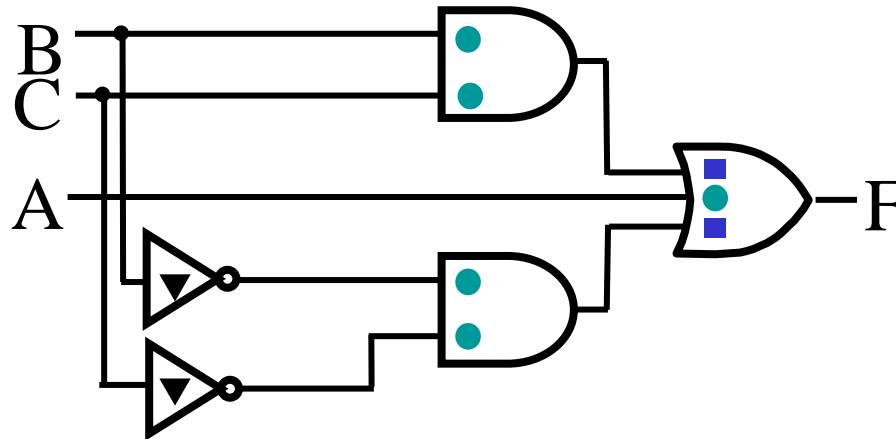


Gate Input Cost

- **Gate input costs** - the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations. (**G** - inverters not counted, **GN** - inverters counted)
- For SOP and POS equations, it can be found from the equation(s) by finding the sum of:
 - all literal appearances
 - the number of terms excluding single literal terms, (G) and
 - optionally, the number of distinct complemented single literals (GN).
- **Example:**
 - $F = BD + A\bar{B}C + A\bar{C}\bar{D}$ $G = 11, GN = 14$
 - $F = BD + A\bar{B}C + A\bar{B}\bar{D} + AB\bar{C}$ $G = 15, GN = 18$
 - $F = (A + \bar{B})(A + D)(B + C + \bar{D})(\bar{B} + \bar{C} + D)$ $G = 14, GN = 17$
 - Which solution is best?

Cost Criteria (continued)

- Example 1: $\nabla \nabla$ $GN = G + 2 = 9$
- $F = \overset{\bullet}{A} + \overset{\bullet}{B} \overset{\bullet}{C} + \overset{\bullet}{\bar{B}} \overset{\bullet}{\bar{C}}$ $L = 5$
- $G = L + 2 = 7$

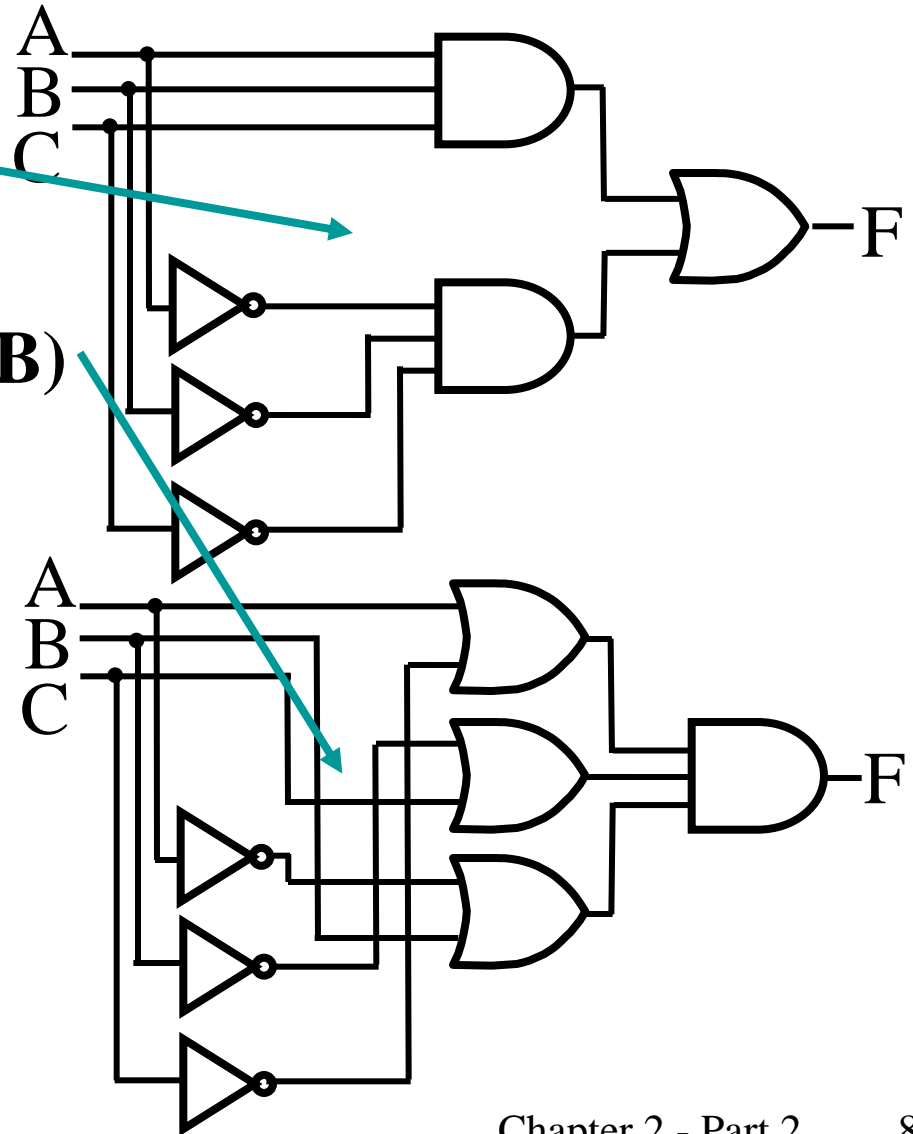


- L (literal count) counts the AND inputs and the single literal OR input.
- G (gate input count) adds the remaining OR gate inputs
- GN(gate input count with NOTs) adds the inverter inputs

Cost Criteria (continued)

Example 2:

- $F = A B C + \bar{A} \bar{B} \bar{C}$
 $L = 6 \quad G = 8 \quad GN = 11$
- $F = (A + \bar{C})(\bar{B} + C)(\bar{A} + B)$
 $L = 6 \quad G = 9 \quad GN = 12$
- Same function and same literal cost
- But first circuit has better gate input count and better gate input count with NOTs
- Select it!



Gate Input Count

- **Example 1**

$$F = \bar{A} \bar{C} \bar{D} + \bar{A} B \bar{C} + ABC + AC\bar{D} \quad G = 16$$

- **Example 2**

$$F = \bar{A} (\bar{C} \bar{D} + B\bar{C}) + A (BC + C\bar{D}) \quad G = 18$$

- **Example 3**

$$F = \bar{A} \bar{C} (B + \bar{D}) + AC (B + \bar{D}) \quad G = 12$$

- **Example 4**

$$F = (\bar{A} \bar{C} + AC) (B + \bar{D}) \quad G = ? \quad \text{💬}$$

Boolean Function Optimization

- Minimizing the gate input (or literal) cost of a (a set of) Boolean equation(s) reduces circuit cost.
- **We choose gate input cost.**
- Boolean Algebra and graphical techniques are tools to minimize cost criteria values.
- Some important questions:
 - **When do we stop trying to reduce the cost?**
 - **Do we know when we have a minimum cost?**
- Treat optimum or near-optimum cost functions for two-level (SOP and POS) circuits first.
- Introduce a graphical technique using Karnaugh maps (K-maps, for short)

Observations

X	Y	F	m_i
0	0	1	$\bar{X}\bar{Y}$
0	1	0	$\bar{X}Y$
1	0	1	$X\bar{Y}$
1	1	0	XY

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

- When a boolean function is expressed as a sum of minterms, optimization can be done using **minimization theorem**:

$$x \cdot y + \bar{x} \cdot y = y$$

K-Map and Truth Tables

- The K-Map is just a different form of the truth table.
- Example – Two variable function:
 - We choose m_0, m_1, m_2 and m_3 from the set $\{0,1\}$ to implement a particular function, $G(x,y)$.

Function Table

Input Values (x,y)	Function Value $G(x,y)$
0 0	m_0 (0)
0 1	m_1 (1)
1 0	m_2 (1)
1 1	m_3 (1)

K-Map

	$y = 0$	$y = 1$
$x = 0$	m_0	m_1
$x = 1$	m_2	m_3

Two Variable Maps

- **Example: $F(x,y) = x$**

$F = x$	$y = 0$	$y = 1$
$x = 0$	0	0
$x = 1$	1	1

- For function $F(x,y)$, the two adjacent cells containing 1's can be combined using the **Minimization Theorem**:

$$F(x, y) = x\bar{y} + xy = x$$

Two Variable Maps

■ **Example:** $G(x,y) = x + y$

$G = x+y$	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	1

- For $G(x,y)$, two pairs of adjacent cells containing 1's can be combined using the Minimization Theorem:

$$G(x, y) = (x\bar{y} + xy) + (xy + \bar{x}y) = x + y$$

Duplicate xy

Three Variable Maps

- A three-variable K-map:

	yz=00	yz=01	yz=11	yz=10
x=0	m ₀	m ₁	m ₃	m ₂
x=1	m ₄	m ₅	m ₇	m ₆

- Where each minterm corresponds to the product terms:

	yz=00	yz=01	yz=11	yz=10
x=0	$\bar{x} \bar{y} \bar{z}$	$\bar{x} \bar{y} z$	$\bar{x} y z$	$\bar{x} y \bar{z}$
x=1	$x \bar{y} \bar{z}$	$x \bar{y} z$	$x y z$	$x y \bar{z}$

- Note that if the binary value for an index differs in one bit position, the minterms are adjacent on the K-Map

Karnaugh Maps (K-map)

- A K-map is a collection of squares
 - Each square represents a minterm
 - The collection of squares is a graphical representation of a Boolean function
 - Adjacent squares differ in the value of one variable (column and row headings must be in **Gray Code order**)
 - Alternative algebraic expressions for the same function are derived by recognizing **patterns of squares**
- The K-map can be viewed as
 - A reorganized version of the **truth table**
 - A clever way to **rewrite truth tables** to make it easier to figure out the logic.
 - A topologically-warped **Venn diagram** as used to visualize sets in algebra of sets

Some Uses of K-Maps

- **Provide a means for:**
 - **Finding optimum or near optimum**
 - **SOP and POS standard forms, and**
 - **two-level AND/OR and OR/AND circuit implementations**
 - for functions with 2 to 5 variables**
 - **Visualizing concepts related to manipulating Boolean expressions, and**
 - **Demonstrating concepts used by computer-aided design programs to simplify large circuits**

Alternative Map Labeling

- Map use largely involves:
 - **Entering values** into the map, and
 - **Reading off product terms** from the map.
- Alternate labelings are useful:

	\bar{y}		y	
\bar{x}	0	1	3	2
x	4	5	7	6
	\bar{z}	z		\bar{z}

		y			
		$x \backslash yz$	00	01	$\overbrace{11 \quad 10}$
x	0	0	1	3	2
	1	4	5	7	6
		$\underbrace{\hspace{1.5cm}}_z$			

Example Functions

- By convention, we represent the minterms of F by a "1" in the map and leave the minterms of \bar{F} blank

- Example:

$$F(x, y, z) = \Sigma_m(2,3,4,5)$$

			y
	0	1	3 1
			2 1
x	4 1	5 1	7
			6

- Example:

$$G(a, b, c) = \Sigma_m(3,4,6,7)$$

- Learn the locations of the 8 indices based on the variable order shown (x , most significant and z , least significant) on the map boundaries

			y
	0	1	3 1
			2
x	4 1	5	7 1
			6 1
		z	

Combining Squares

- By combining squares, we reduce number of literals in a product term, reducing the literal cost, thereby reducing the other two cost criteria
- On a 3-variable K-Map:
 - **One square** represents a minterm with three variables
 - **Two adjacent squares** represent a product term with two variables
 - **Four “adjacent” terms** represent a product term with one variable
 - **Eight “adjacent” terms** is the function of all ones (no variables) = 1.

Example: Combining Squares

- Example: Let $F = \Sigma m(2,3,6,7)$

	\bar{y}		y	
\bar{x}	0	1	3 1	2 1
x	4	5	7 1	6 1
	\bar{z}		z	\bar{z}

- Applying the Minimization Theorem three times:

$$\begin{aligned}
 F(x, y, z) &= \bar{x} y z + x y z + \bar{x} y \bar{z} + x y \bar{z} \\
 &= yz + y\bar{z} \\
 &= y
 \end{aligned}$$

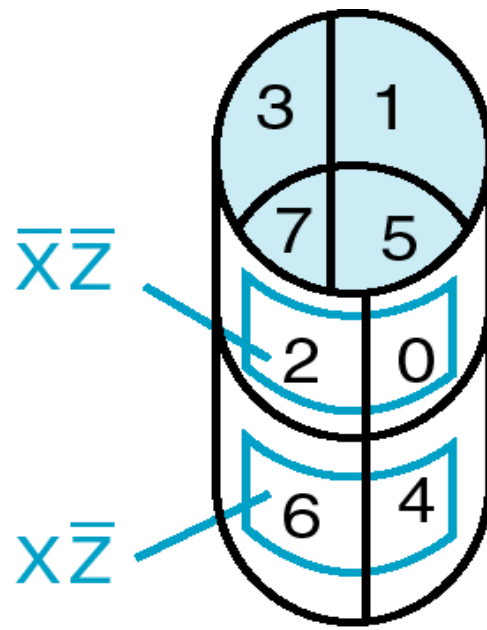
- Thus the four terms that form a 2×2 square correspond to the term "y".

Three Variable Maps

- Reduced literal product terms for SOP standard forms correspond to rectangles on K-maps containing cell counts that are powers of 2.
- Rectangles of 2 cells represent 2 adjacent minterms; of 4 cells represent 4 minterms that form a “pairwise adjacent” ring.
- Rectangles can contain **non-adjacent cells** as illustrated by the “pairwise adjacent” ring above.

Three Variable Maps

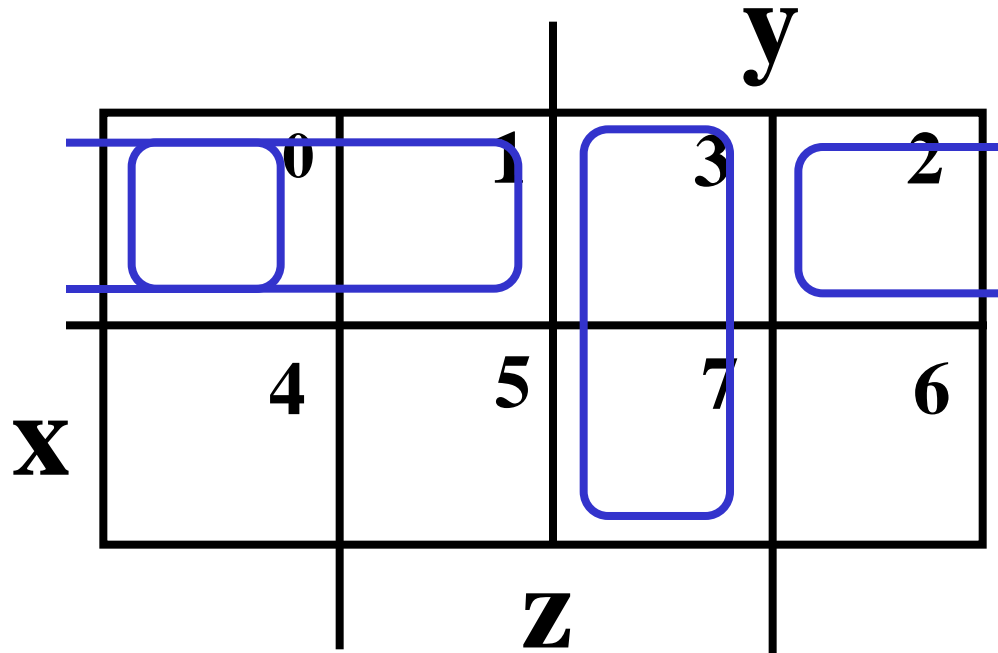
- Topological warps of 3-variable K-maps that show *all* adjacencies:
 - Cylinder



	\bar{y}		y	
\bar{x}	0 1	1	3	2 1
x	4 1	5	7	6 1
	\bar{z}	z		\bar{z}

Three Variable Maps

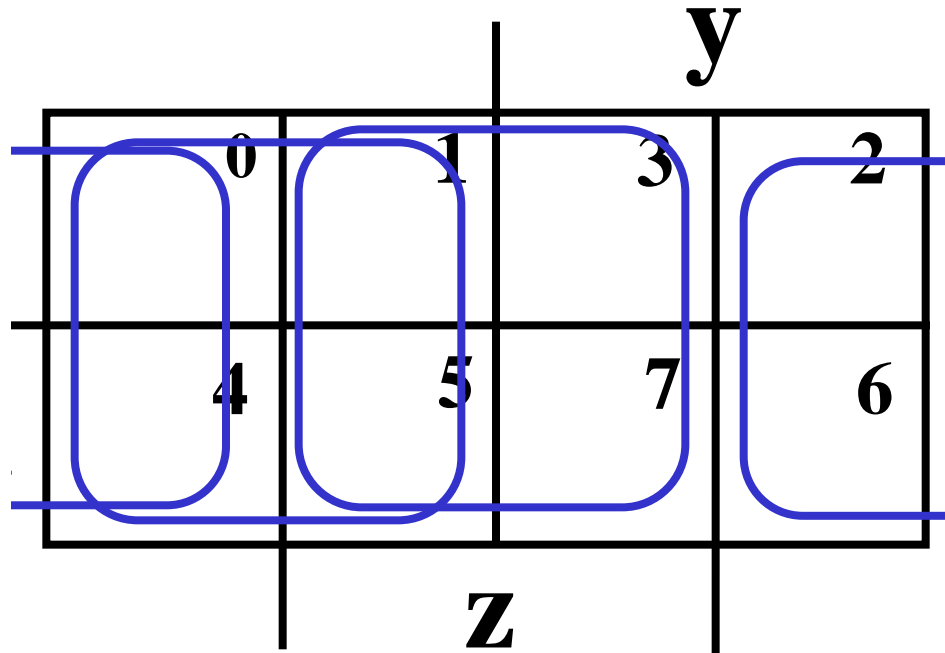
- **Example Shapes of 2-cell Rectangles:**



- **Read off the product terms for the rectangles shown**

Three Variable Maps

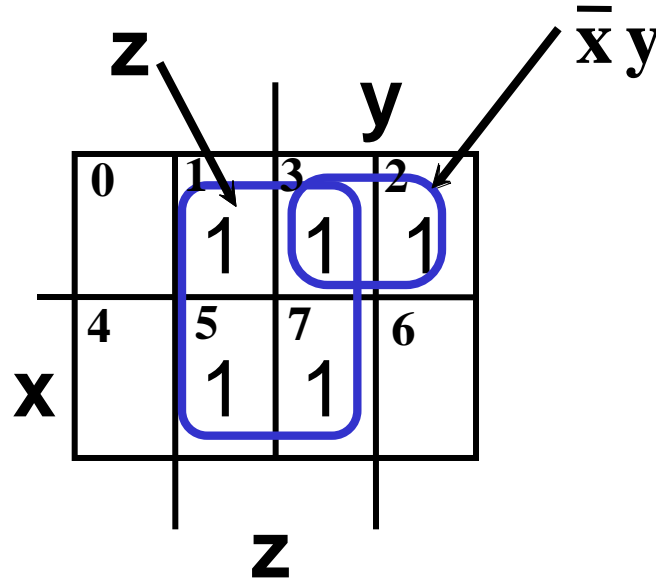
- **Example Shapes of 4-cell Rectangles:**



- **Read off the product terms for the rectangles shown**

Three Variable Maps

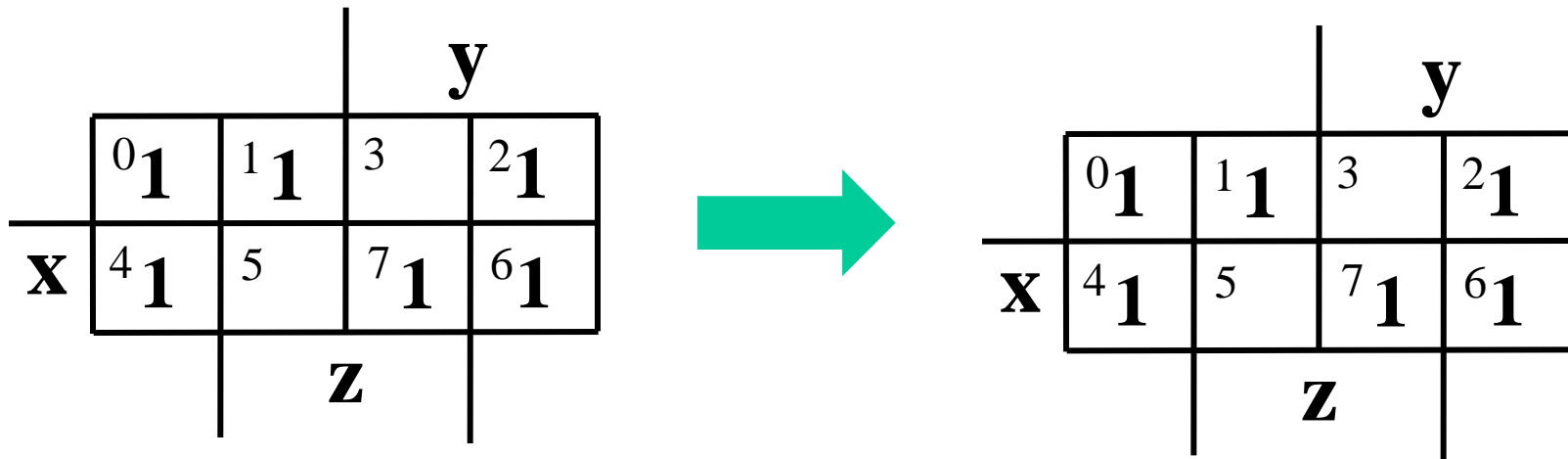
- K-Maps can be used to simplify Boolean functions by systematic methods. Terms are selected to cover the “1s” in the map.
- Example: Simplify $F(x, y, z) = \Sigma_m(1,2,3,5,7)$



$$F(x, y, z) = z + \bar{x}y$$

Three Variable Map Simplification

- Use a K-map to find an optimum SOP equation for $F(X, Y, Z) = \Sigma_m(0,1,2,4,6,7)$



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

Four Variable Maps

- Map and location of minterms:

Variable Order

WXYZ

	00	01	11	10	
00	0	1	3	2	
01	4	5	7	6	
11	12	13	15	14	X
10	8	9	11	10	
					Z

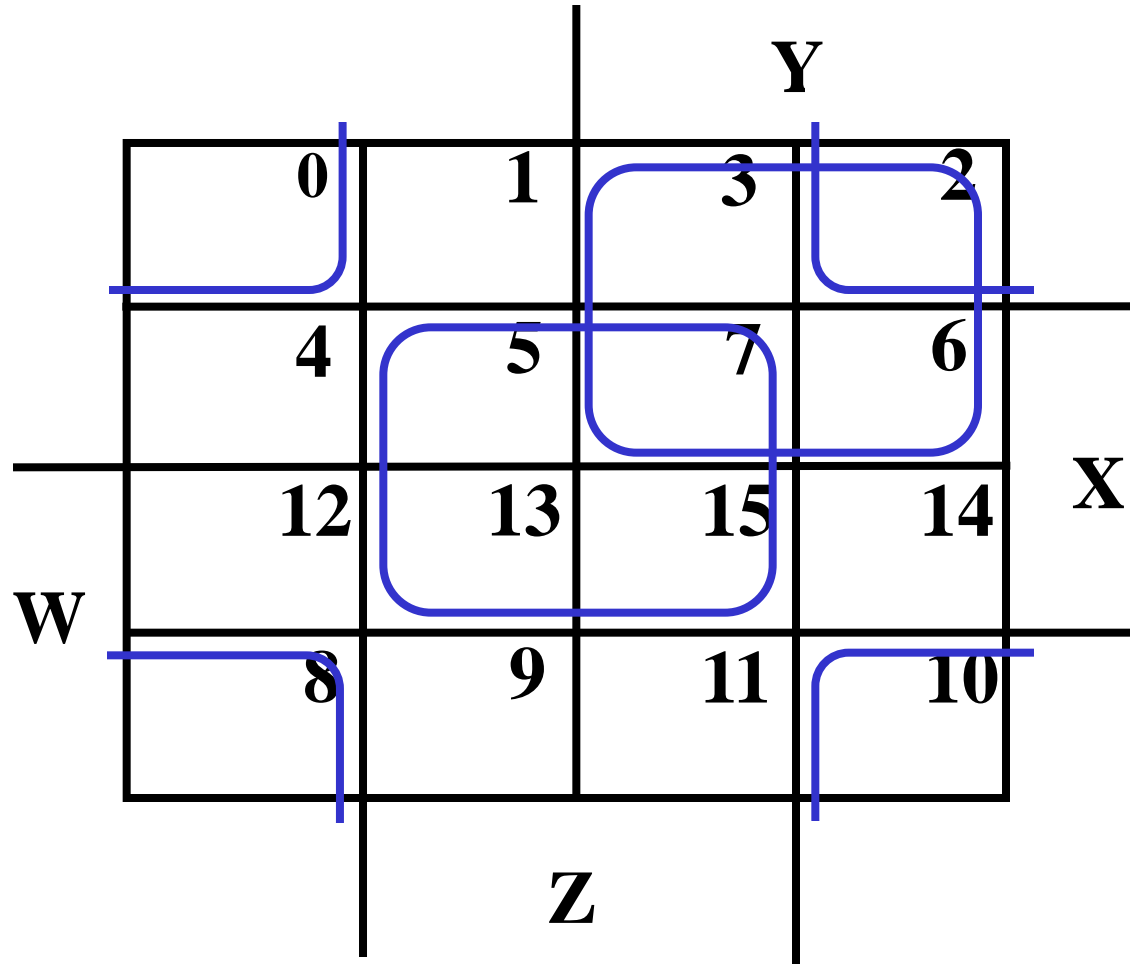
The diagram shows a 4x4 grid representing a four-variable map. The columns are labeled with the values of variables W and X (00, 01, 11, 10) and the rows are labeled with the values of variables Y and Z (00, 01, 11, 10). The minterm numbers are placed in the cells of the grid. The variable order is indicated by the text 'Variable Order' and 'WXYZ'.

Four Variable Terms

- **Four variable maps can have rectangles corresponding to:**
 - **A single 1 = 4 variables, (Minterm)**
 - **Two 1s = 3 variables,**
 - **Four 1s = 2 variables**
 - **Eight 1s = 1 variable,**
 - **Sixteen 1s = zero variables (Constant "1")**

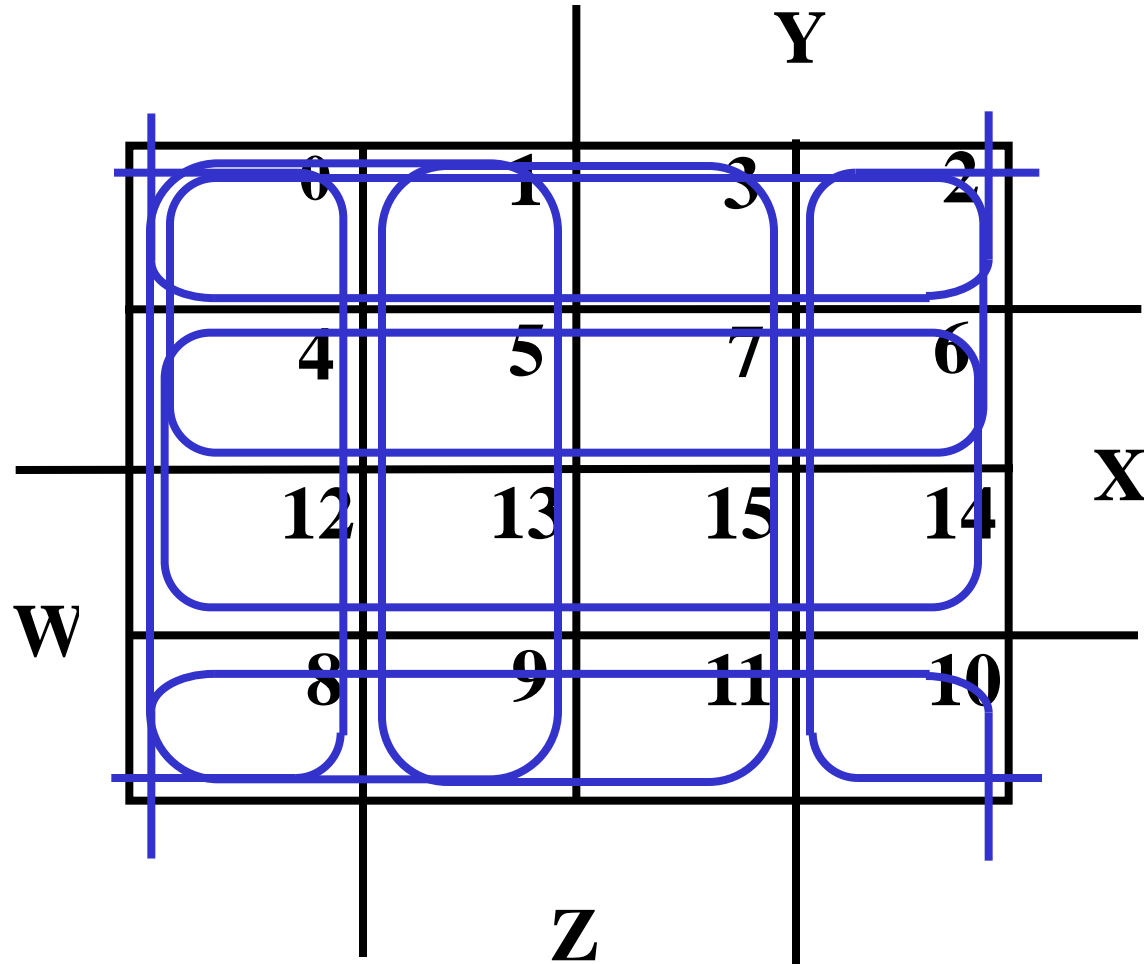
Four Variable Maps

- **Example Shapes of Rectangles:**



Four Variable Maps

- **Example Shapes of Rectangles:**



Four Variable Map Simplification

■ $F(W, X, Y, Z) = \Sigma_m (0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

WXYZ

	00	01	11	10	
	Y		Y		
00	0 1	1	3	2 1	
01	4 1	5 1	7 1	6 1	
11	12	13 1	15 1	14	X
10	8 1	9	11	10 1	
W	Z				


Four Variable Map Simplification

- $F(W, X, Y, Z) = \Sigma_m(3,4,5,7,9,13,14,15)$

WXYZ

		00	01	11	Y	10	
00		0	1	3	1	2	
01		4	5	7	1	6	
11		12	13	15	1	14	X
W							
10		8	9	11		10	
					Z		

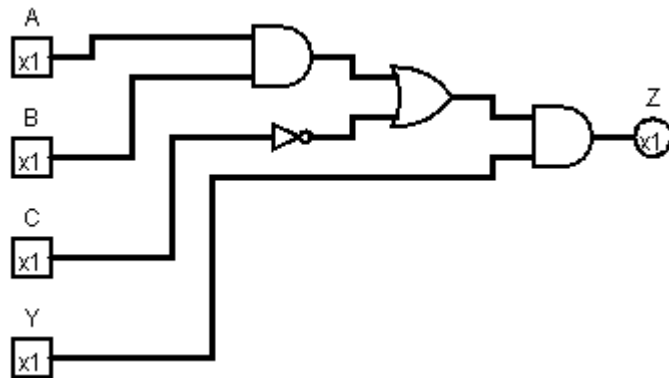
Don't Cares in K-Maps

- Sometimes a function table or map contains entries for which it is known:
 - The input values for the minterm will never occur, or
 - The output value for the minterm is not used
- In these cases, the output value need not be defined.
- The output value is defined as a “don't care” that may take on either a 0 or 1 value in resulting solutions.
- By placing “don't cares” (an “x” entry) in the function table or map, the cost of the logic circuit may be lowered.
- Example 1: A logic function having the binary codes for the BCD digits as its inputs. Only the codes for 0 through 9 are used. The six codes, 1010 through 1111 will never occur, so the output values for these codes are “don't cares.” 

Some input values will never occur!

Don't Cares in K-Maps

- **Example 2: A circuit that represents a very common situation that has two distinct sets of input variables and a single output Z:**



if $AB + \overline{C} = 1$ **then** $Z = Y$
else Z is a don't care

Some output values are not used!

- A, B, and C which take on all possible combinations, and
- Y which takes on values 0 or 1.

The circuit that receives the input Y observes it only for combinations of A, B, and C such $A = 1$ and $B = 1$ or $C = 0$, otherwise ignoring it.

- Thus, Z is specified only for those combinations, and for all other combinations of A, B, and C, Z is a don't care.
- Any minterm with value “x” *need not be covered* by a prime implicant.

Example: BCD “5 or More”

- The map below gives a function $F_1(w,x,y,z)$ which is defined as "5 or more" over BCD inputs. With the don't cares used for the 6 non-BCD combinations:

		y				
		0	1	3	2	
w	x	0	1	1	1	0
		4	5	7	6	
	X	X	X	X		
	12	13	15	14		
		1	1	X	X	
		8	9	11	10	
		z				

$$F_1(w,x,y,z) = w + xz + xy \quad G = 7$$

- This is much lower in cost than F_2 where the “don't cares” were treated as “0s.”

$$F_2(w, x, y, z) = \bar{w}xz + \bar{w}xy + w\bar{x}\bar{y} \quad G = 12$$

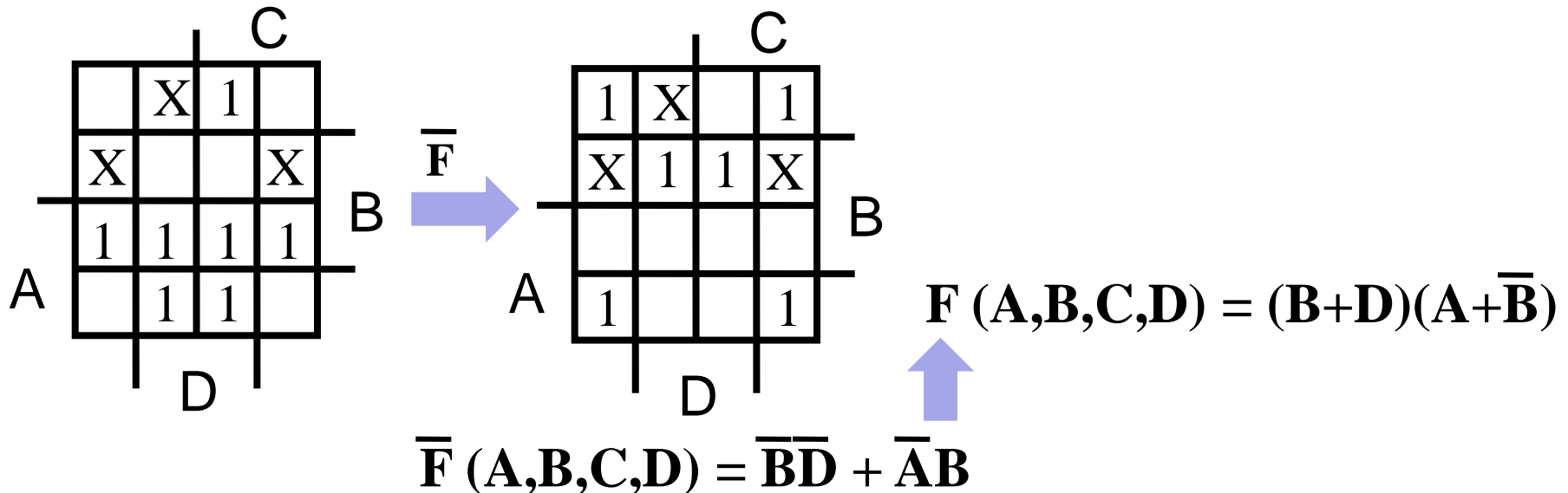
- For this particular function, cost G for the POS solution for $F_1(w,x,y,z)$ is not changed by using the don't cares.

Product of Sums Example

- Find the optimum POS solution:

$$F(A, B, C, D) = \Sigma_m(3, 9, 11, 12, 13, 14, 15) + \Sigma_d(1, 4, 6) \quad \leftarrow \text{don't cares}$$


- Hint: Use \bar{F} and complement it to get the result.



Five Variable Maps

- 2^5 Minterms, consists of 32 squares

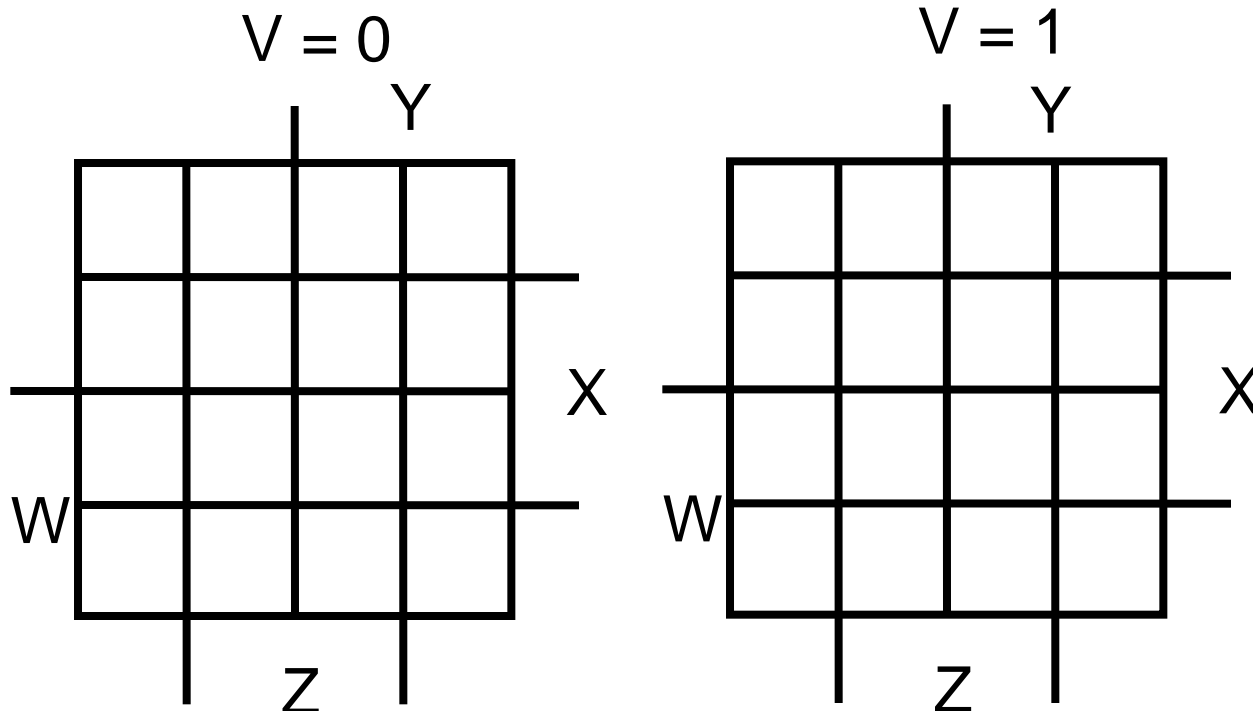
Gray Code order



XYZ VW	000	001	011	010	110	111	101	100
00	0	1	3	2	6	7	5	4
01	8	9	11	10	14	15	13	12
11	24	25	27	26	30	31	29	28
10	16	17	19	18	22	23	21	20

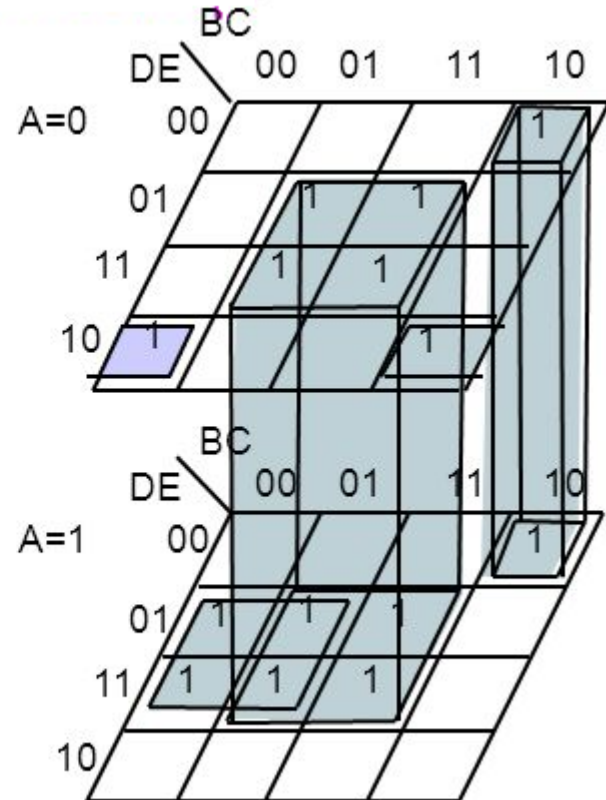
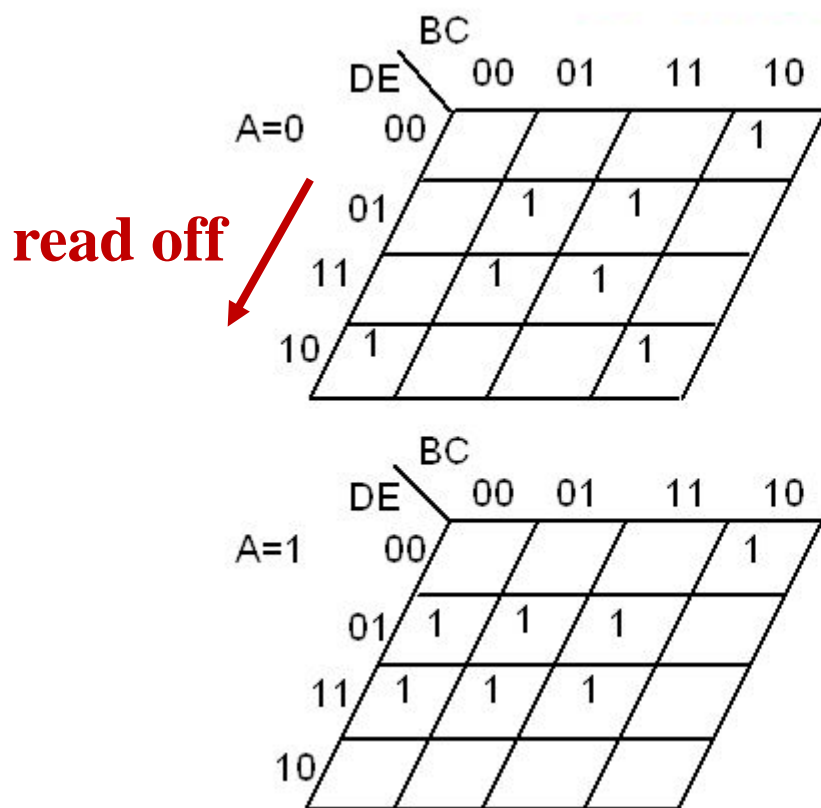
Five Variable Maps

- For five variable problems, we use *two adjacent K-maps*. It becomes harder to visualize adjacent minterms for selecting PIs. You can extend the problem to six variables by using four K-Maps.



Five Variable Maps

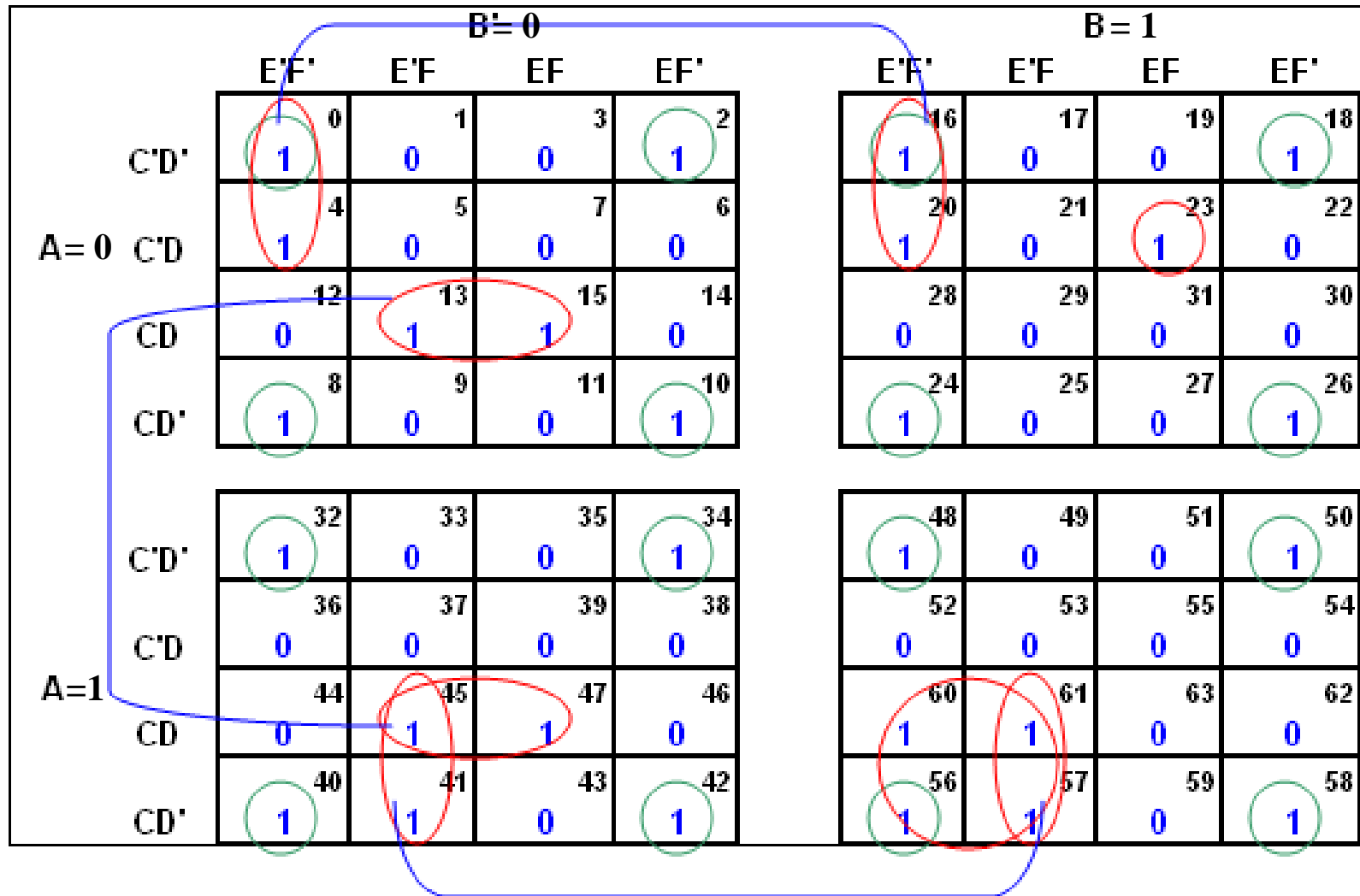
- The **overlay mode** of the Karnaugh map



$$F(ABCDE) = \sum_m(2, 5, 7, 8, 10, 13, 15, 17, 19, 21, 23, 24, 29, 31)$$

$$F(ABCDE) = CE + A\bar{B}E + B\bar{C}\bar{D}\bar{E} + \bar{A}\bar{C}D\bar{E}$$

Six Variable Maps



Advantages of K-Maps

- **The K-map simplification technique is simpler and less error-prone compared to the method of solving the logical expressions using Boolean laws.**
- **It prevents the need to remember each and every Boolean algebraic theorem.**
- **It involves fewer steps than the algebraic minimization technique to arrive at a simplified expression.**
- **K-map simplification technique always results in minimum expression if carried out properly.**

Disadvantages of K-Maps

- As the number of variables in the logical expression increases, the K-map simplification process becomes complicated.
- The minimum logical expression **may or may not be unique** depending on the choices made while forming the groups.

		BC			
		00	01	11	10
A	0	1	1	0	0
	1	1	0	1	1

(a)

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

		BC			
		00	01	11	10
A	0	1	1	0	0
	1	1	0	1	1

(b)

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

K-map with a non-unique solution

Optimization Algorithm

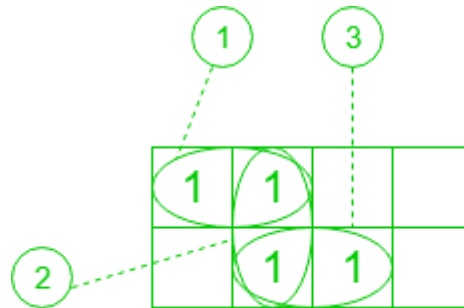
- Minimizing boolean functions **by hand** using the classical Karnaugh maps is a laborious, tedious and error prone process.
- It isn't suited for more than six input variables and **practical only for up to 5 variables**, while product term sharing for multiple output functions is even harder to carry out.
- Moreover, this method doesn't lead itself to be **automated** in the form of a computer program.

Example of Boolean Minimization

- Truth Table Solver is a program that solves the truth table and output all the possible minimized boolean expressions.
- It uses **Quine-McCluskey algorithm** (the method of **prime implicants**) for boolean minimization.
- Quine-McCluskey Solver:
<http://www.quinemccluskey.com>

Systematic Simplification

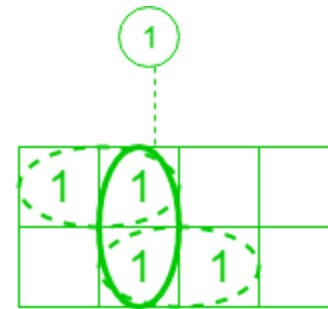
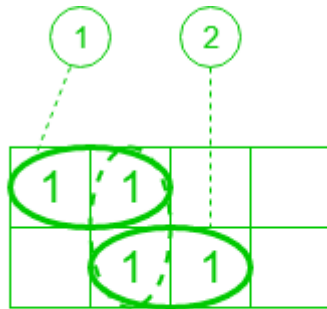
- **Implicant** is a minterm/product term in SOP or maxterm/sum term in POS of a Boolean function. For example:
 $F = AB + ABC + BC$. Implicants are AB, ABC and BC.
- The group of “1s” is called **implicant**. There are two types of implicants: **Prime Implicant** and **Essential Prime Implicant**.
- A **Prime Implicant** is a product term obtained by combining the **maximum possible number** of adjacent squares in the map into a rectangle with the number of squares a power of 2.



No. of Prime Implicants = 3

Systematic Simplification (1/2)

- An ***Essential Prime Implicant*** is a prime implicant that covers one or more minterms that no combination of other prime implicants are able to include.

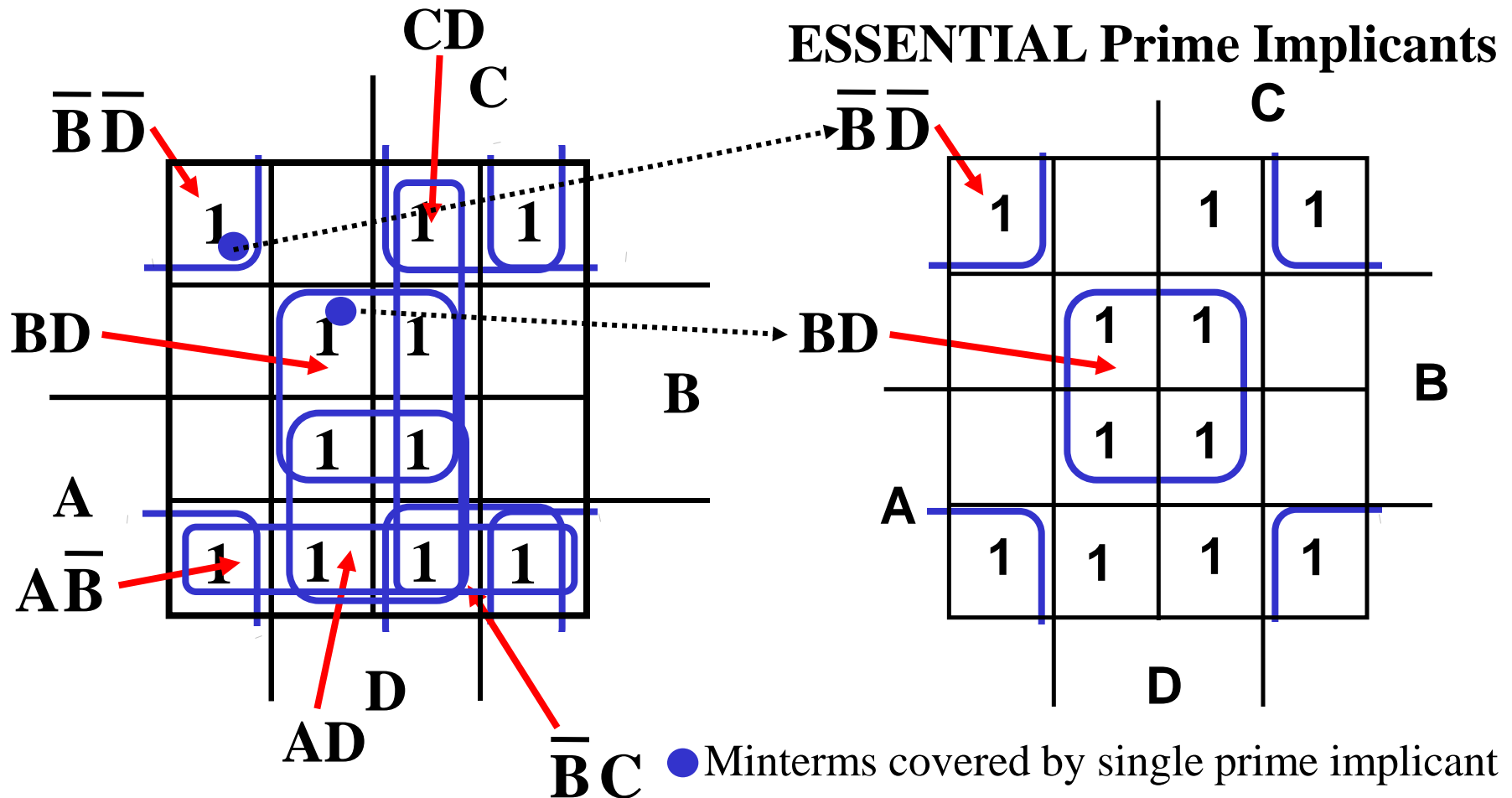


No. of Essential Prime Implicants = 2 No. of Redundant Prime Implicants = 1

- A set of prime implicants "*covers all minterms*" if, for each minterm of the function, at least one prime implicant in the set of prime implicants includes the minterm.

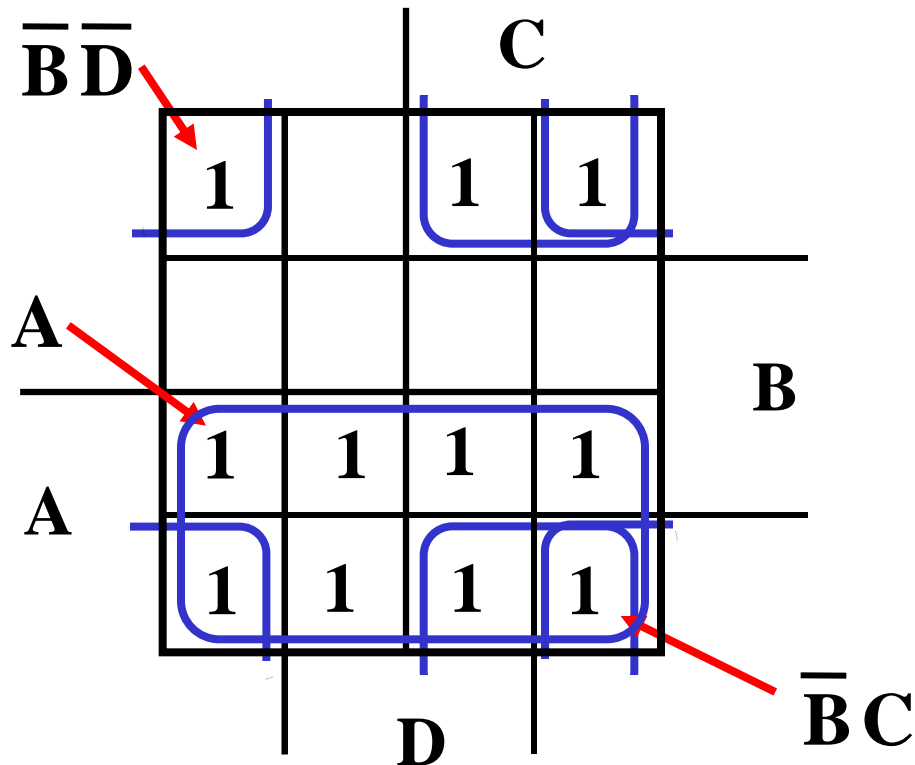
Example of Prime Implicants

- Find ALL Prime Implicants



Prime Implicant Practice

- Find all prime implicants for:
 $F(A, B, C, D) = \Sigma_m(0, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15)$



Another Example

- Find all prime implicants for:
 $G(A, B, C, D) = \Sigma_m(0, 2, 3, 4, 7, 12, 13, 14, 15)$
 - Hint: There are seven prime implicants!

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

Quine–McCluskey Algorithm

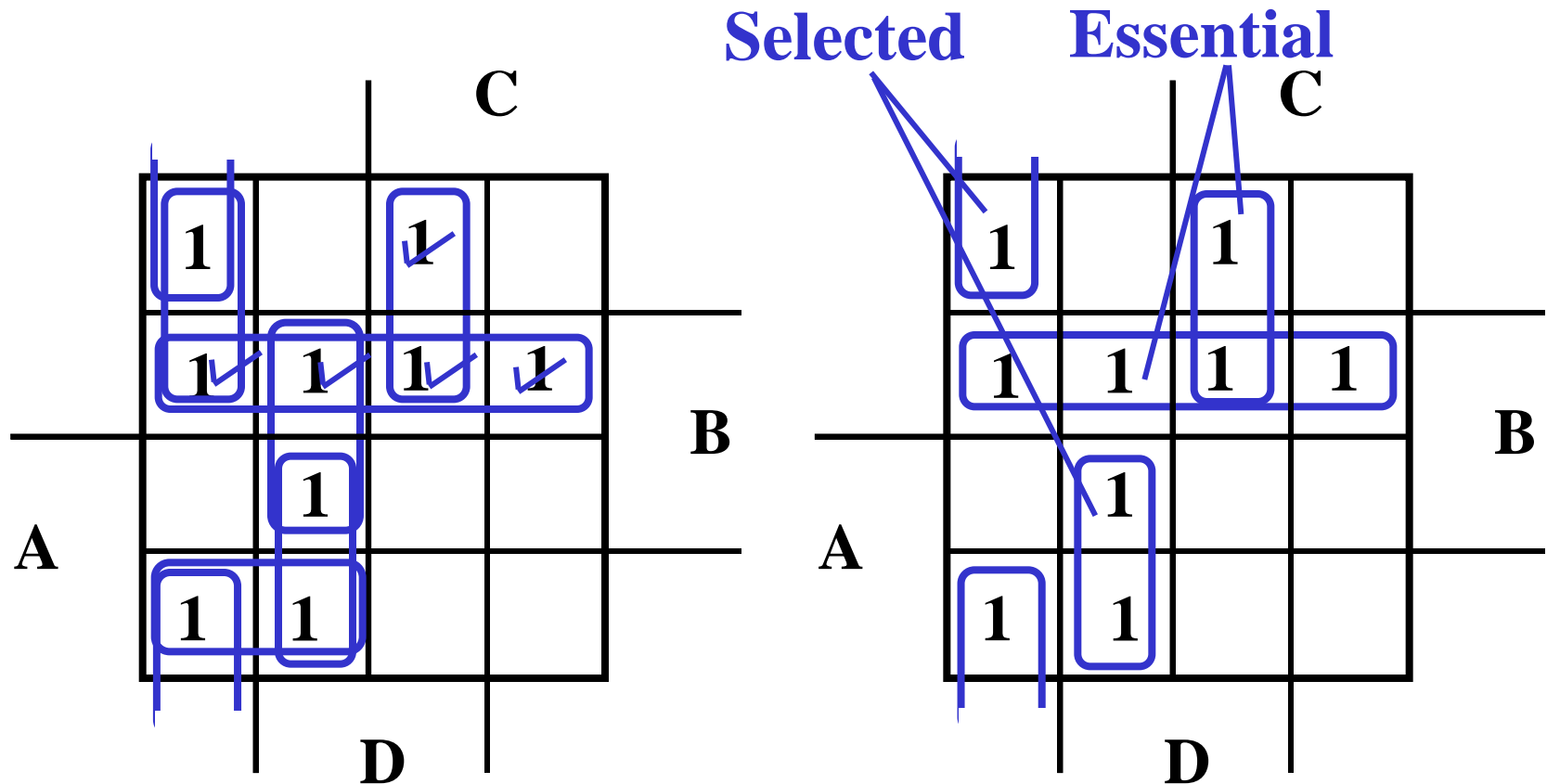
- Find **all prime implicants**.
- Include **all essential prime implicants** in the solution
- Select **a minimum cost set of non-essential prime implicants** to cover all minterms not yet covered:
 - Obtaining an optimum solution: See Reading Supplement - More on Optimization
 - Obtaining a good simplified solution: Use the Selection Rule
- Appendix A: **Quine–McCluskey Solver**

Prime Implicant Selection Rule

- **Minimize the overlap among prime implicants as much as possible. In particular, in the final solution, make sure that each prime implicant selected includes at least one minterm not included in any other prime implicant selected.**

Selection Rule Example

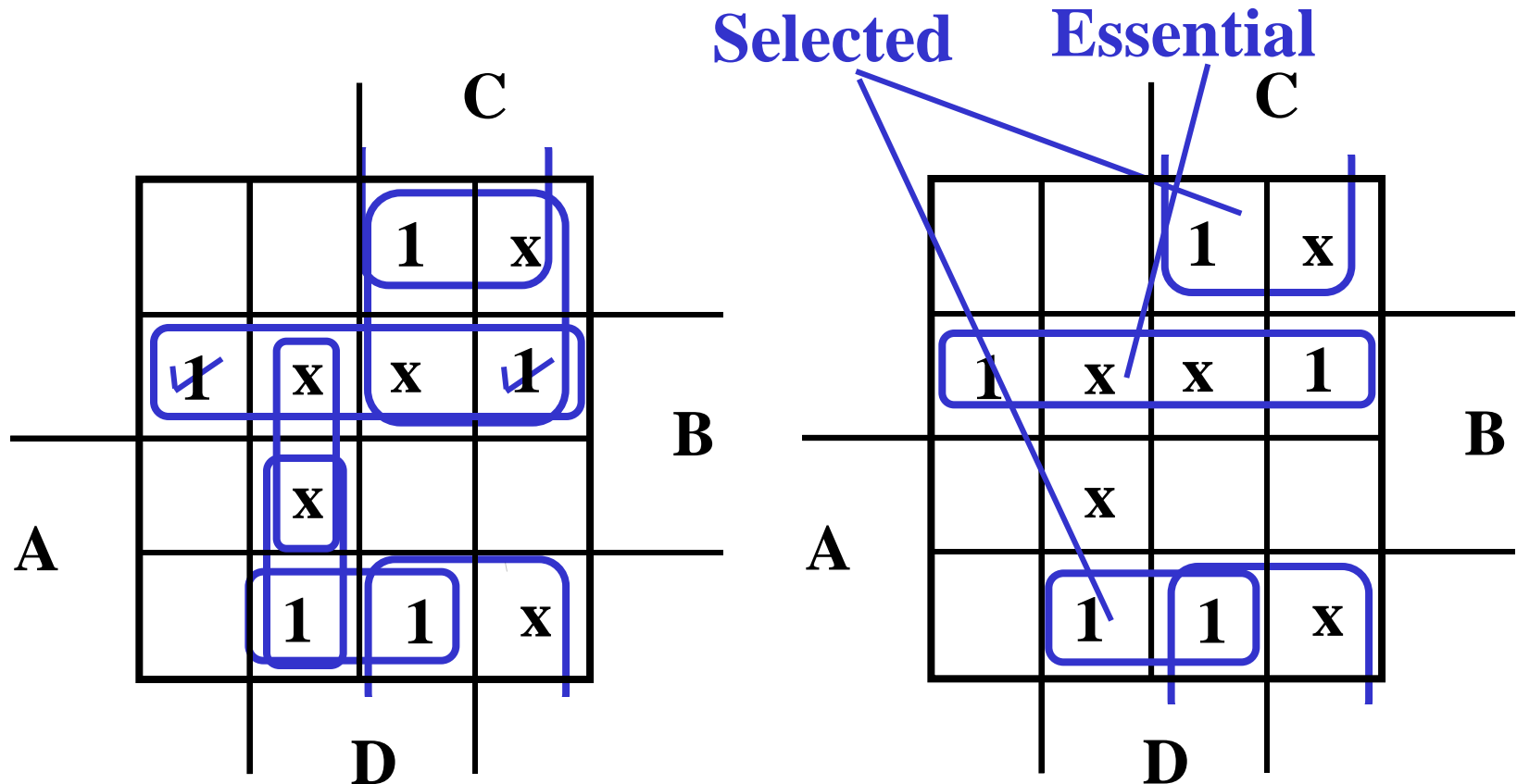
- Simplify $F(A, B, C, D)$ given on the K-map.



✓ Minterms covered by essential prime implicants

Selection Rule Example with Don't Cares

- Simplify $F(A, B, C, D)$ given on the K-map.



✓ Minterms covered by essential prime implicants

Multiple-Level Optimization

- **Multiple-level circuits - circuits that are not two-level (with or without input and/or output inverters)**
- **Multiple-level circuits can have reduced gate input cost compared to two-level (SOP and POS) circuits**
- **Multiple-level optimization is performed by **applying transformations** to circuits represented by equations while evaluating cost**

Transformation Examples

- **Algebraic Factoring**

$$F = \bar{A} \bar{C} \bar{D} + \bar{A} B \bar{C} + ABC + AC\bar{D} \quad G = 16$$

- **Factoring:**

$$F = \bar{A} (\bar{C} \bar{D} + B\bar{C}) + A (BC + C\bar{D}) \quad G = 18$$

- **Factoring again:**

$$F = \bar{A} \bar{C} (B + \bar{D}) + AC (B + \bar{D}) \quad G = 12$$

- **Factoring again:**

$$F = (\bar{A} \bar{C} + AC) (B + \bar{D}) \quad G = 10$$

Overview

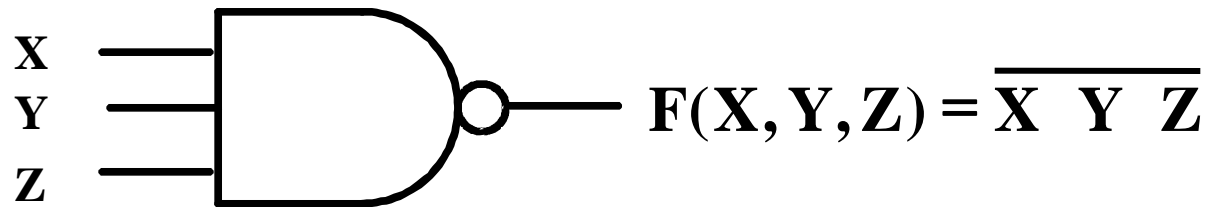
- **Part 1 – Gate Circuits and Boolean Equations**
 - **Binary Logic and Gates**
 - **Boolean Algebra**
 - **Standard Forms**
- **Part 2 – Circuit Optimization**
 - **Two-Level Optimization**
 - **Map Manipulation**
 - **Practical Optimization**
 - **Multi-Level Circuit Optimization**
- **Part 3 – Additional Gates and Circuits**
 - **Other Gate Types**
 - **Exclusive-OR Operator and Gates**
 - **High-Impedance Outputs**

Other Gate Types

- **Why?**
 - **Implementation feasibility and low cost**
 - **Power in implementing Boolean functions**
 - **Convenient conceptual representation**
- **Gate classifications**
 - **Primitive gate** - a gate that can be described using a single primitive operation type (AND or OR) plus an optional inversion(s).
 - **Complex gate** - a gate that requires more than one primitive operation type for its description
- **Primitive gates will be covered first**

NAND Gate

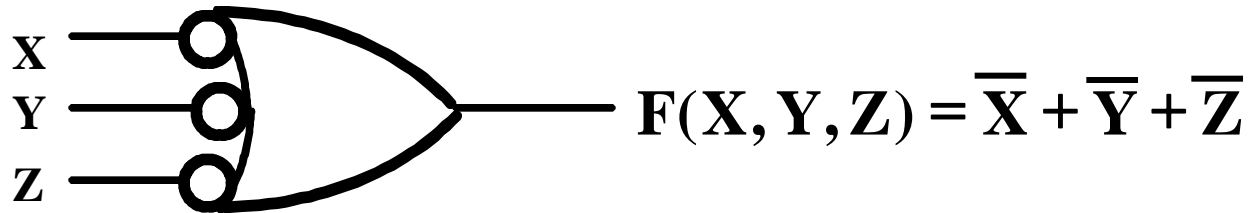
- The basic NAND gate has the following symbol, illustrated for three inputs:
 - **AND-Invert (NAND)**



- **NAND** represents NOT AND, i. e., the AND function with a NOT applied. The symbol shown is an AND-Invert. The small circle (“bubble”) represents the invert function.

NAND Gates (continued)

- Applying DeMorgan's Law gives Invert-OR (NAND)



- This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together.
- AND-Invert and Invert-OR both represent the NAND gate. Having both makes visualization of circuit function easier.
- A NAND gate with one input degenerates to an inverter.

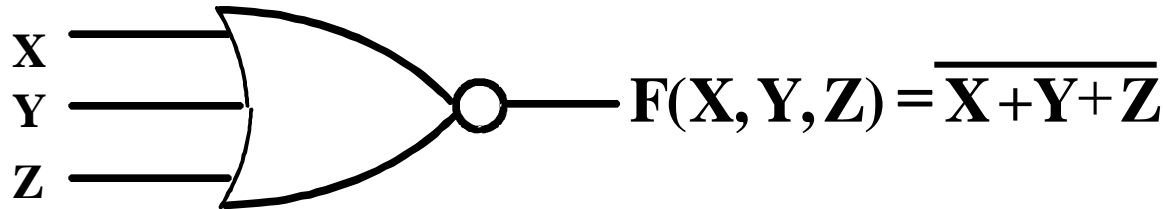
NAND Gates (continued)

- The NAND gate is the natural implementation for CMOS technology in terms of chip area and speed.
- *Universal gate* - a gate type that can implement any Boolean function.
- The NAND gate is a universal gate as shown in Figure 2-4 of the text.
- NAND usually does not have an operation symbol defined since
 - the NAND operation is not associative, and
 - we have difficulty dealing with non-associative mathematics!

NOR Gate

- The basic NOR gate has the following symbol, illustrated for three inputs:

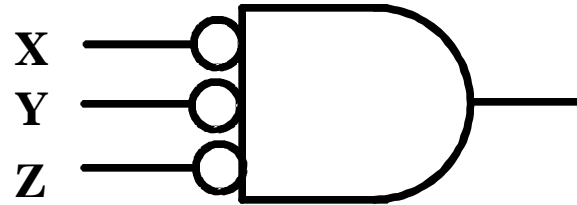
- OR-Invert (NOR)



- NOR represents NOT - OR, i. e., the OR function with a NOT applied. The symbol shown is an OR-Invert. The small circle (“bubble”) represents the invert function.

NOR Gate (continued)

- Applying DeMorgan's Law gives Invert-AND (NOR)






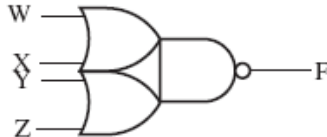

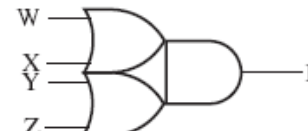
- This NOR symbol is called Invert-AND, since inputs are inverted and then ANDed together.
- OR-Invert and Invert-AND both represent the NOR gate. Having both makes visualization of circuit function easier.
- A NOR gate with one input degenerates to an inverter.

NOR Gate (continued)

- The NOR gate is a natural implementation for some technologies other than CMOS in terms of chip area and speed.
- The NOR gate is a **universal gate**
- NOR usually does not have a defined operation symbol since
 - the NOR operation is not associative, and
 - we have difficulty dealing with **non-associative** mathematics!

Other Gate Types

- Other gate types aim to lower the cost and transmit time in circuit.

Name	Distinctive shape symbol	Algebraic equation	Truth table															
Exclusive-OR (XOR)		$F = X\bar{Y} + \bar{X}Y$ $= X \oplus Y$	<table> <tr> <th>X</th> <th>Y</th> <th>F</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	X	Y	F	0	0	0	0	1	1	1	0	1	1	1	0
X	Y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR (XNOR)		$F = XY + \bar{X}\bar{Y}$ $= \overline{X \oplus Y}$	<table> <tr> <th>X</th> <th>Y</th> <th>F</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	X	Y	F	0	0	1	0	1	0	1	0	0	1	1	1
X	Y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																
AND-OR-INVERT (AOI)		$F = \overline{WX + YZ}$																
OR-AND -INVERT (OAI)		$F = \overline{(W + X)(Y + Z)}$																
AND-OR (AO)		$F = WX + YZ$																
OR-AND (OA)		$F = (W + X)(Y + Z)$																

Exclusive OR/Exclusive NOR

- The *eXclusive OR (XOR)* function is an important Boolean function used extensively in logic circuits.
- The XOR function may be:
 - implemented directly as an electronic circuit (truly a gate) or
 - implemented by interconnecting other gate types (used as a convenient representation)
- The *eXclusive NOR* function is the complement of the XOR function
- XOR and XNOR gates are both **complex gates**.

Truth Tables for XOR/XNOR

- Operator Rules: XOR

	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	0

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

X	Y	$\overline{(X \oplus Y)}$ or $X \equiv Y$
0	0	1
0	1	0
1	0	0
1	1	1

- The XOR function means:

X OR Y, but NOT BOTH

- Why is the XNOR function also known as the *equivalence* function, denoted by the operator \equiv ?

Exclusive OR/ Exclusive NOR

- **Definitions**

- The XOR function is: $X \oplus Y = X \bar{Y} + \bar{X} Y$
- The eXclusive NOR (XNOR) function, otherwise known as *equivalence* is: $\overline{X \oplus Y} = X Y + \bar{X} \bar{Y}$

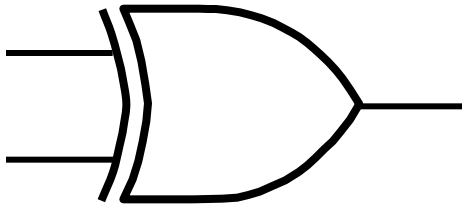
- **Uses for the XOR and XNORs gate include:**

- Adders/subtractors/multipliers
- Counters/incrementers/decrementers
- Parity generators/checkers

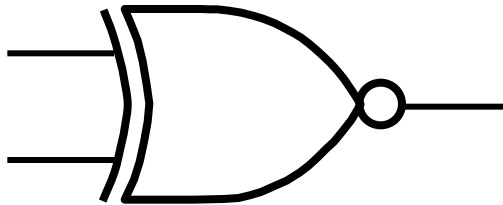
- **Strictly speaking, XOR and XNOR gates do not exist for more than two inputs. Instead, they are replaced by *odd* and *even* functions.**

Symbols For XOR and XNOR

- **XOR symbol:**



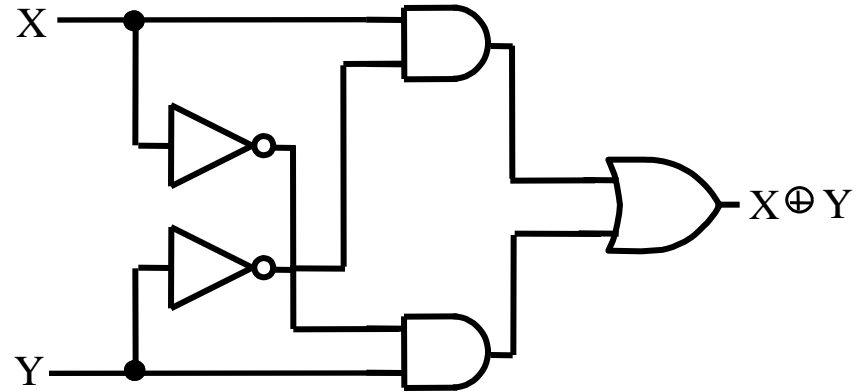
- **XNOR symbol:**



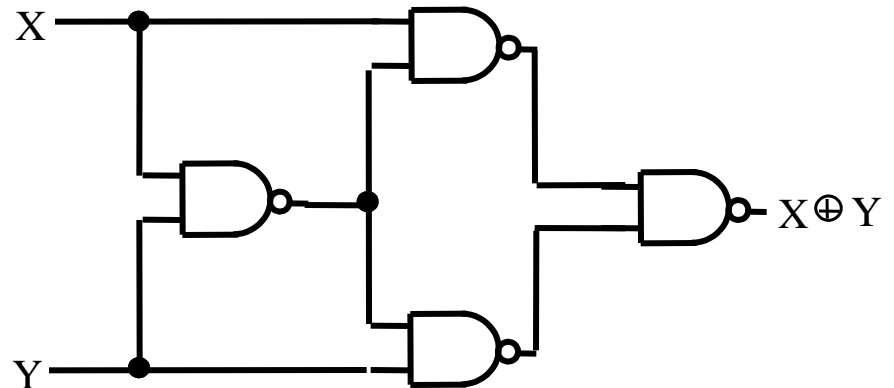
- **Shaped symbols exist only for two inputs**

XOR Implementations

- The simple SOP implementation uses the following structure:



- A NAND only implementation is:



XOR/XNOR (Continued)

- The XOR identities:

$$X \oplus 0 = X$$

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

$$X \oplus X = 0$$

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

$$X \oplus Y = Y \oplus X$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$$

- XOR and XNOR are **associative** operations.

Find the Element that Appears Once

- An array is consisting of integer numbers. Each number except one appears twice. The remaining number appears only once. How to find a number which occurs only once?
- Example: Suppose that an array contains values **1, 4, 2, 1, 3, 4, 2**. The value 3 appears only once.

```
Function Singular(a, N)
    value = 0
    for i = 0, N-1
        value = value XOR a[i]
    return value
```

XOR/XNOR (Continued)

- The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called an *odd function* or *modulo 2 sum (Mod 2 sum)*, not an XOR:

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

- The complement of the odd function is the *even function*.

Odd and Even Functions

- The odd and even functions on a K-map form “checkerboard” patterns.

		YZ			
		00	01	11	10
X	0		1		1
	1	1		1	

$X \oplus Y \oplus Z$

		YZ			
		00	01	11	10
WX	00		1		1
	01	1		1	
	11		1		1
	10	1		1	

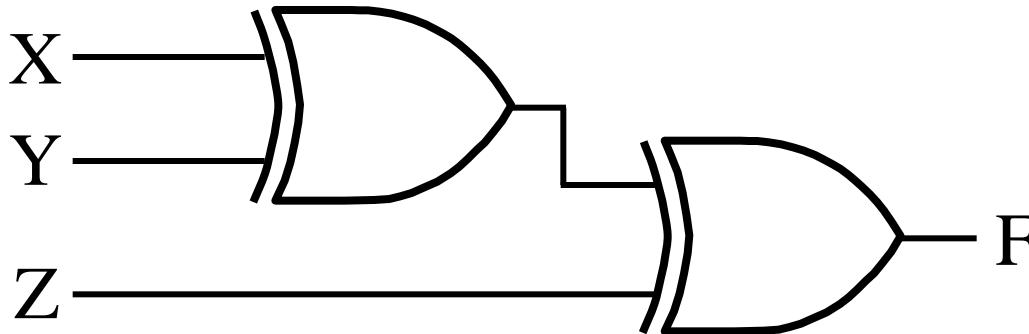
$W \oplus X \oplus Y \oplus Z$

Odd and Even Functions (continued)

- The 1s of an **odd function** correspond to minterms having **an index with an odd number of 1s**.
- The 1s of an **even function** correspond to minterms having **an index with an even number of 1s**.
- Implementation of odd and even functions for greater than four variables as a two-level circuit is difficult, so we use “**trees**” made up of :
 - 2-input XOR or XNORs
 - 3- or 4-input odd or even functions

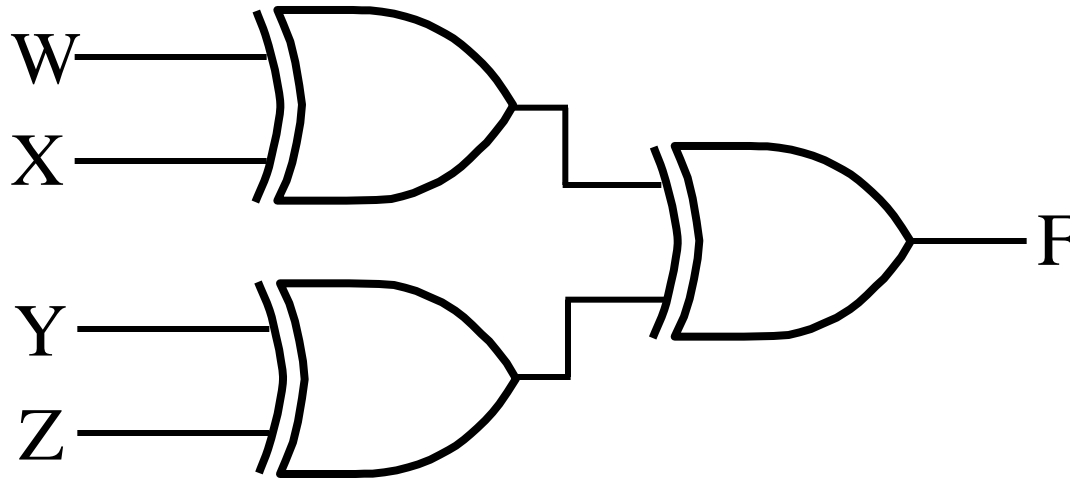
Example: Odd Function Implementation

- Design a 3-input odd function $F = X \oplus Y \oplus Z$ with 2-input XOR gates
- Factoring, $F = (X \oplus Y) \oplus Z$
- The circuit:



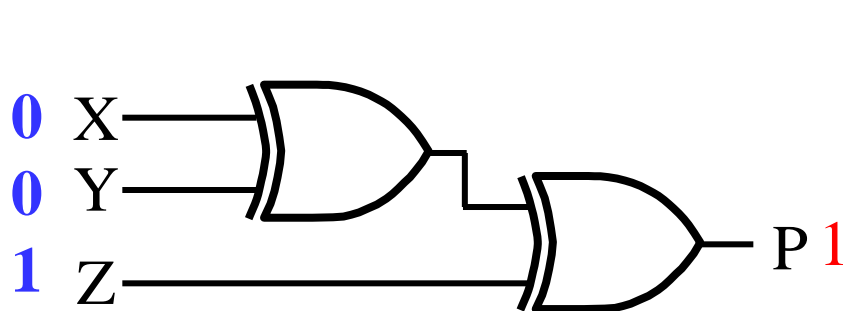
Example: Odd Function Implementation

- Design a 4-input odd function $F = W \oplus X \oplus Y \oplus Z$ with 2-input XOR gates
- Factoring, $F = (W \oplus X) \oplus (Y \oplus Z)$
- The circuit:

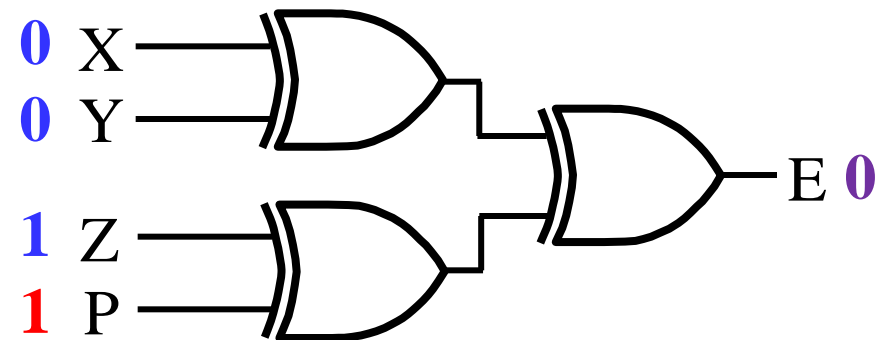


Parity Generator & Checkers

- Design an **even parity generator and checker** for 3-bit codes
- **Solution:** Use 3-bit **odd function** to generate even parity bit
- Use 4-bit **odd function** to check for errors in even parity codes
- **Operation:** $(X,Y,Z) = (0,0,1)$ gives $(X,Y,Z,P) = (0,0,1,1)$ and $E = 0$
- If Y changes from 0 to 1 between generator and checker, then $E = 1$ indicates an error

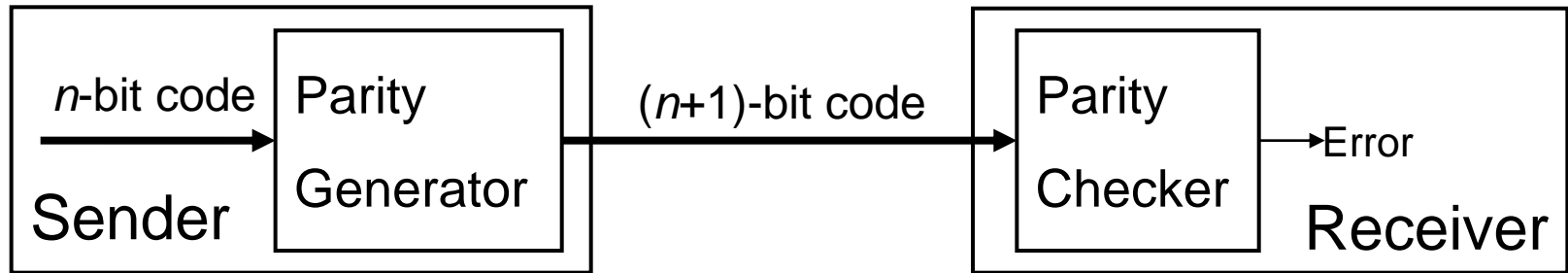


even parity generator



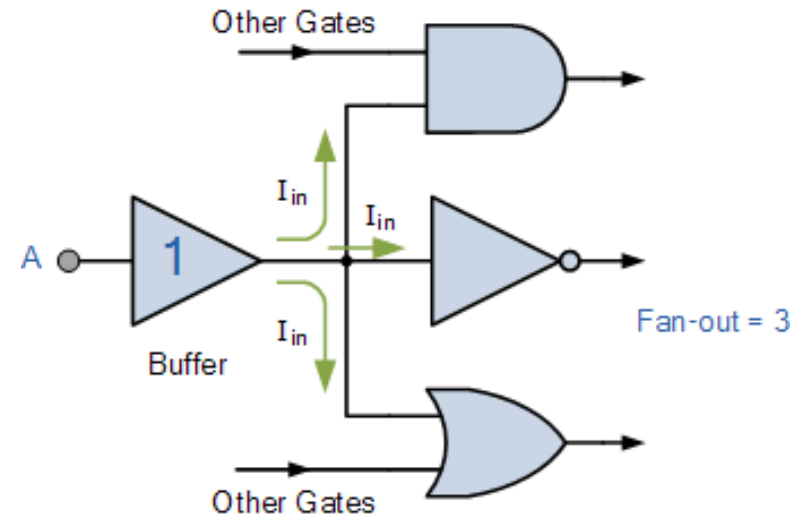
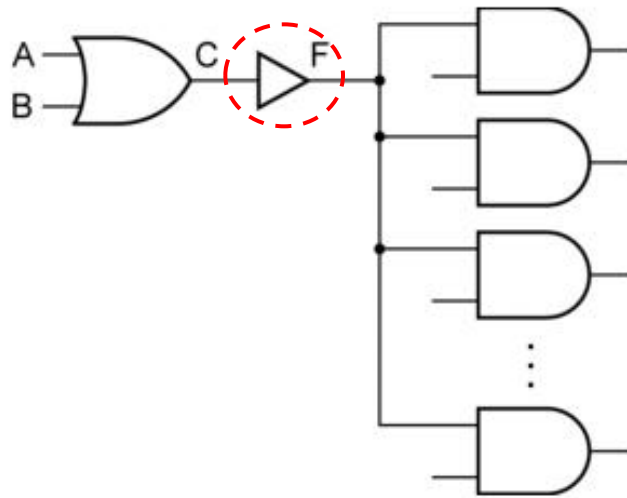
even parity checker

Parity Generator & Checkers

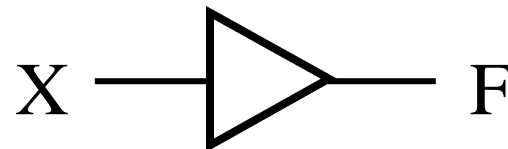


- **Even Parity bit: count of 1s in $(n+1)$ -bit code is even**
 - So use an **odd function** to generate the **even parity bit**
- **To check for even parity**
 - Use an **odd function** to check the $(n+1)$ -bit code
- **Odd Parity bit: count of 1s in $(n+1)$ -bit code is odd**
 - So use an **even function** to generate the **odd parity bit**
- **To check for odd parity**
 - Use an **even function** to check the $(n+1)$ -bit code

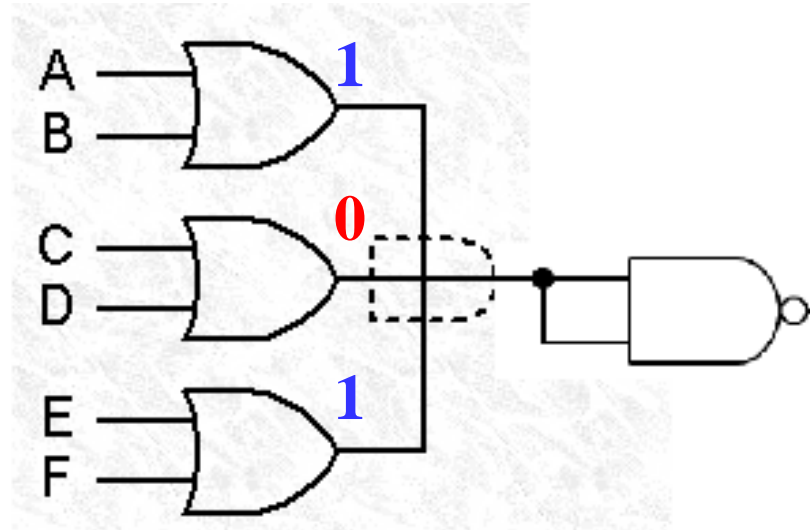
Buffer



- A buffer is an electronic **amplifier** used to **improve circuit voltage levels** and **increase the speed of circuit operation**.
- The buffer is a gate with the function $F = X$



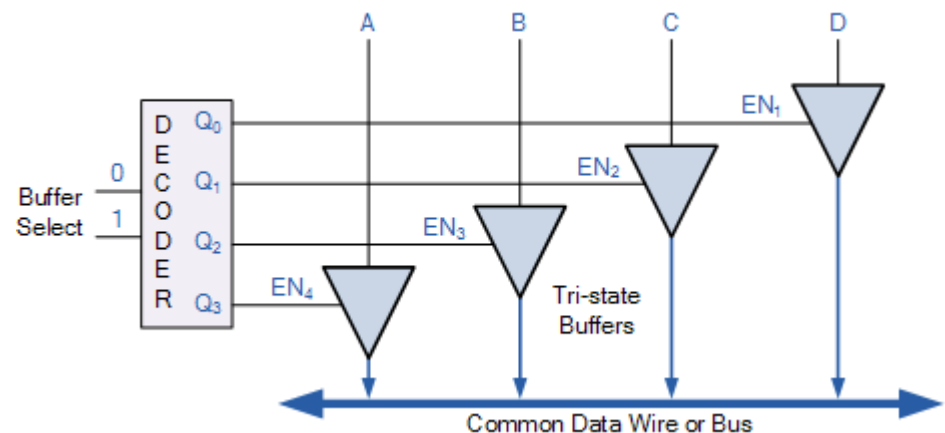
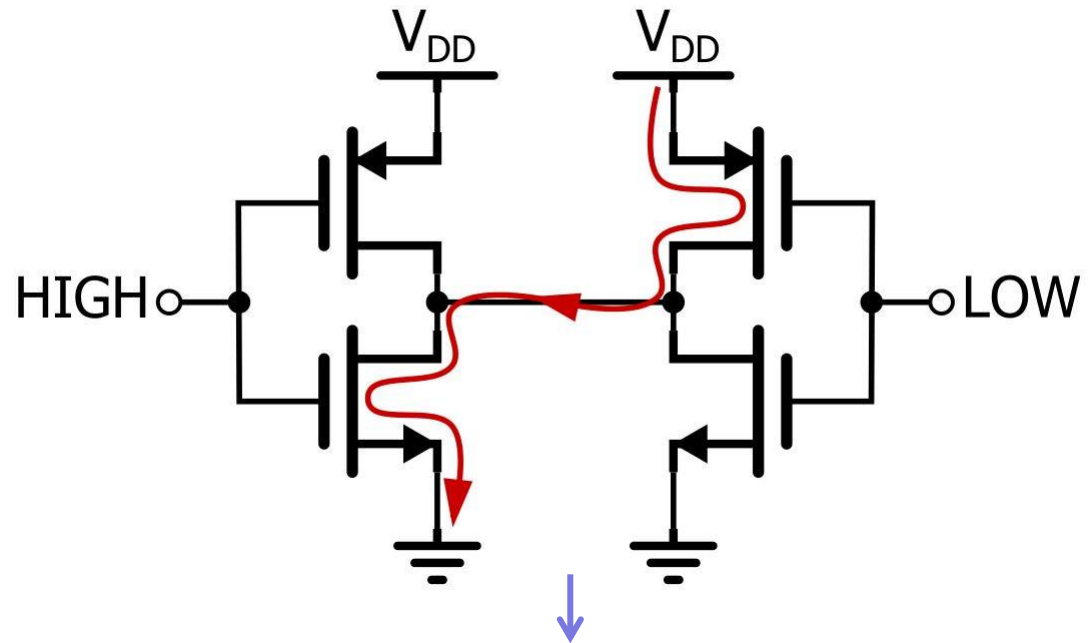
Wired Output



- Can we connect the output of multiple gates together?
- Logic gates introduced thus far ...
 - Have 1 and 0 output values
 - Cannot have their outputs connected together

Wired Output (continued)

- In the case of two inverters wired together, if one is logic high and the other is logic low, the current will flow freely from V_{DD} to ground!



The 3-State Buffer

- **Three-state logic** adds a third logic value:

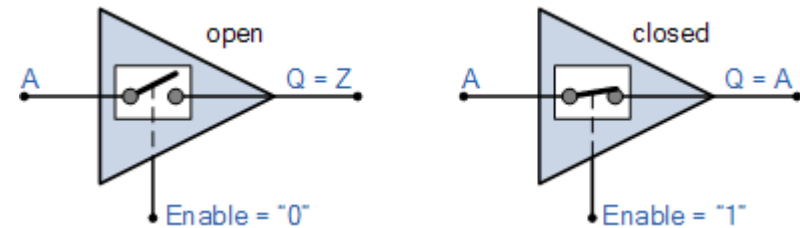
- Hi-Impedance output: **Hi-Z**

- **What is Hi-Impedance output?**

- The output appears to be **disconnected** from the input
- Behaves as an **open circuit** between gate input & output

- **Hi-Z state makes a gate output behave differently:**

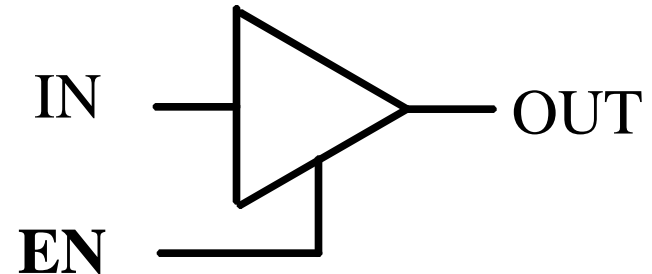
- Three output values: 1, 0, and Hi-Z
- Hi-impedance gates **can connect** their outputs together



The 3-State Buffer

- **IN** = data input
- **EN** = **Enable control input**
- **OUT** = data output
- **If EN = 0 then OUT = HI-Z**
 - Regardless of the value on IN
 - Output disconnected from input
- **If EN = 1, then OUT = IN**
 - Output follows the input value
- **Variations:**
 - EN can be inverted
 - OUT can be inverted
 - By addition of bubbles to signals

Symbol

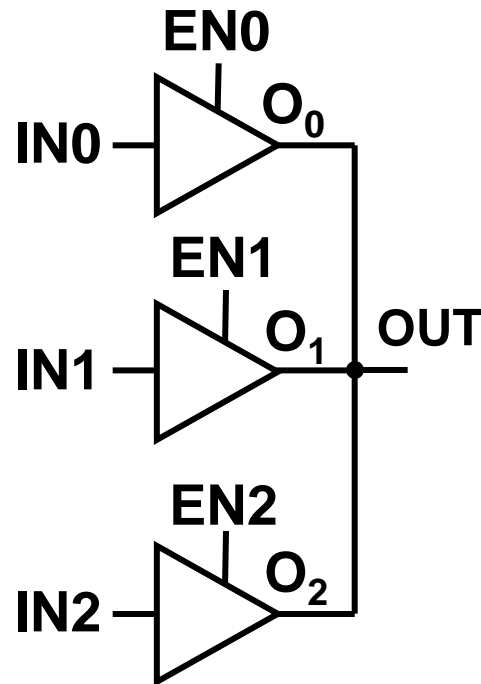


Truth Table

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

The 3-State Buffer: Resolving Output Value

- The output of 3-state buffers can be **wired together**
- **At most one 3-state buffer can be enabled.** Resolved output is equal to the output of the enabled 3-state buffer
- If multiple 3-state buffers are enabled at the same time then **conflicting outputs will burn the circuit**



Resolution Table			
O ₀	O ₁	O ₂	OUT
0 or 1	Hi-Z	Hi-Z	O0
Hi-Z	0 or 1	Hi-Z	O1
Hi-Z	Hi-Z	0 or 1	O2
Hi-Z	Hi-Z	Hi-Z	Hi-Z
0 or 1	0 or 1	0 or 1	Burn

Resolving 3-State Values on a Connection

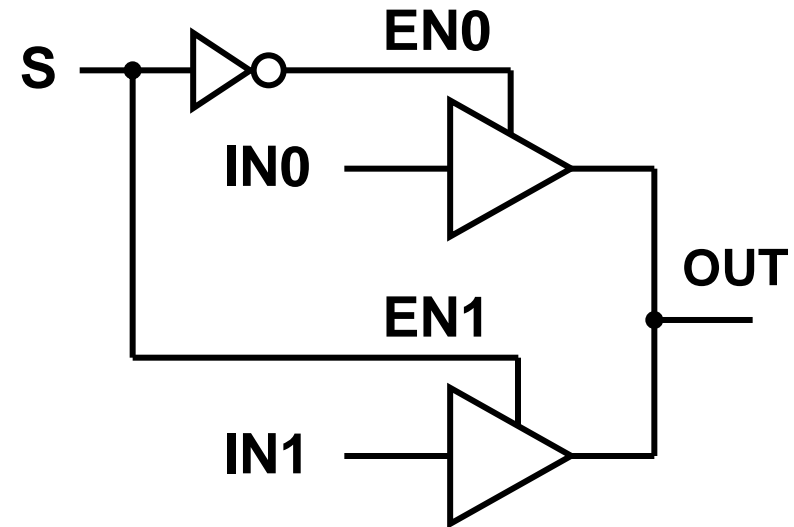
- **Connection** of two 3-state buffer outputs, B1 and B0, to a wire, OUT
- **Assumption:** Buffer data inputs can take any combination of values (0/1)
- **Resulting Rule:** At least one buffer output value must be Hi-Z.
- How many valid buffer output combinations exist? **5**
- What is the rule for n 3-state buffers connected to wire, OUT?
 $n-1$ buffer outputs must be Hi-Z
- How many valid buffer output combinations exist? **$2n+1$**

Resolution Table		
B1	B0	OUT
0	Hi-Z	0
1	Hi-Z	1
Hi-Z	0	0
Hi-Z	1	1
Hi-Z	Hi-Z	Hi-Z

Data Selection Circuit

- Performing data selection with 3-state buffers:
 - If $S = 0$ then $OUT = IN0$ else $OUT = IN1$

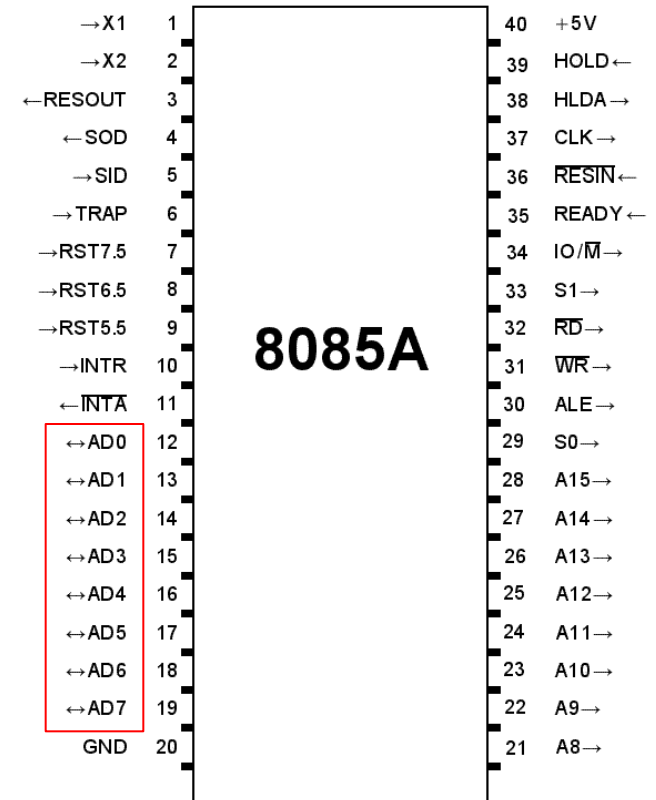
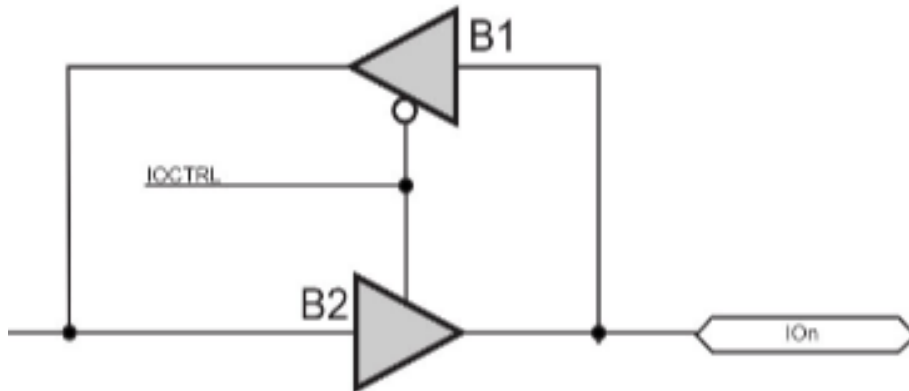
S	EN0	EN1	IN0	IN1	OUT
0	1	0	0	X	0
0	1	0	1	X	1
1	0	1	X	0	0
1	0	1	X	1	1



- The outputs of the 3-state buffers are **wired together**
- Since $EN0 = \overline{S}$ and $EN1 = S$, one of the two buffer outputs is always Hi-Z

Bidirectional

- Bidirectional means the data incoming and outgoing data flows over the same channel.
- 3-state buffers are always used to create bidirectional.

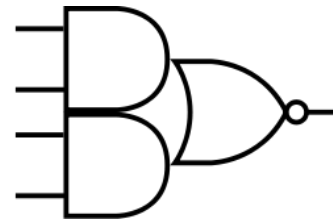


Pin 12-19

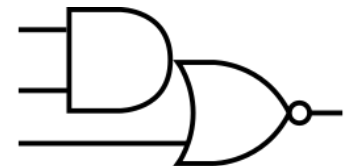
- multiplexed addr/data bus
- bidirectional

More Complex Gates

- The remaining complex gates are SOP or POS structures with and without an output inverter.
- The names are derived using:
 - A - AND
 - O - OR
 - I - Inverter
 - **Numbers of inputs** on first-level “gates” or directly to second-level “gates”



2-2 AOI

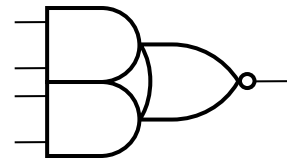


2-1 AOI

More Complex Gates (continued)

- Example: **2-2 AOI** consists of two 2-input ANDS driving an OR function which is inverted.

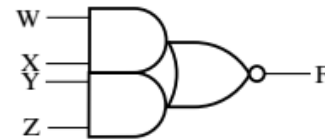
$$F = \overline{WX + YZ}$$



- Example: **2-2-1 AO** has two 2-input ANDS driving an OR with one additional OR input.

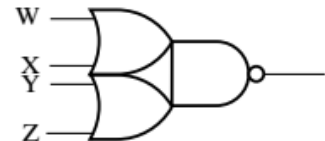
$$F = WX + YZ + V$$

AND-OR-INVERT
(AOI)



$$F = \overline{WX + YZ}$$

OR-AND -INVERT
(OAI)



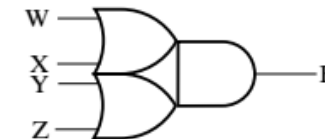
$$F = \overline{(W + X)(Y + Z)}$$

AND-OR
(AO)



$$F = WX + YZ$$

OR-AND
(OA)



$$F = (W + X)(Y + Z)$$

Assignments

Reading:

- 2.4-2.6

Problem assignment:

- 2-15c; 2-17b; 2-19a; 2-22a; 2-25b

Appendix A: Quine–McCluskey Solver

- **Quine-McCluskey Solver**

<http://www.quinemccluskey.com/>

- **Input**



- **Comparasion**



- **Prime Implicants**



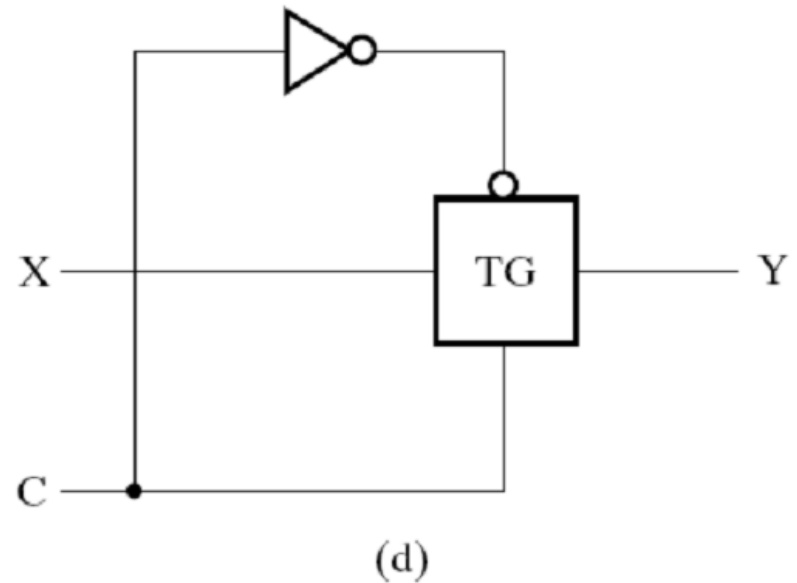
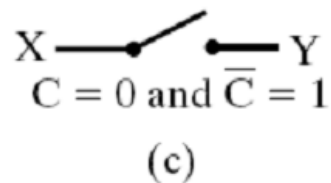
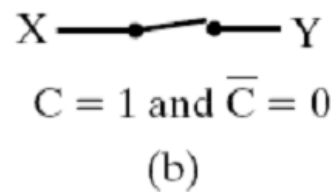
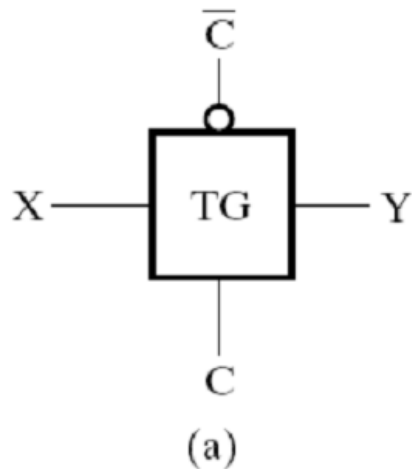
- **Coverage Table**



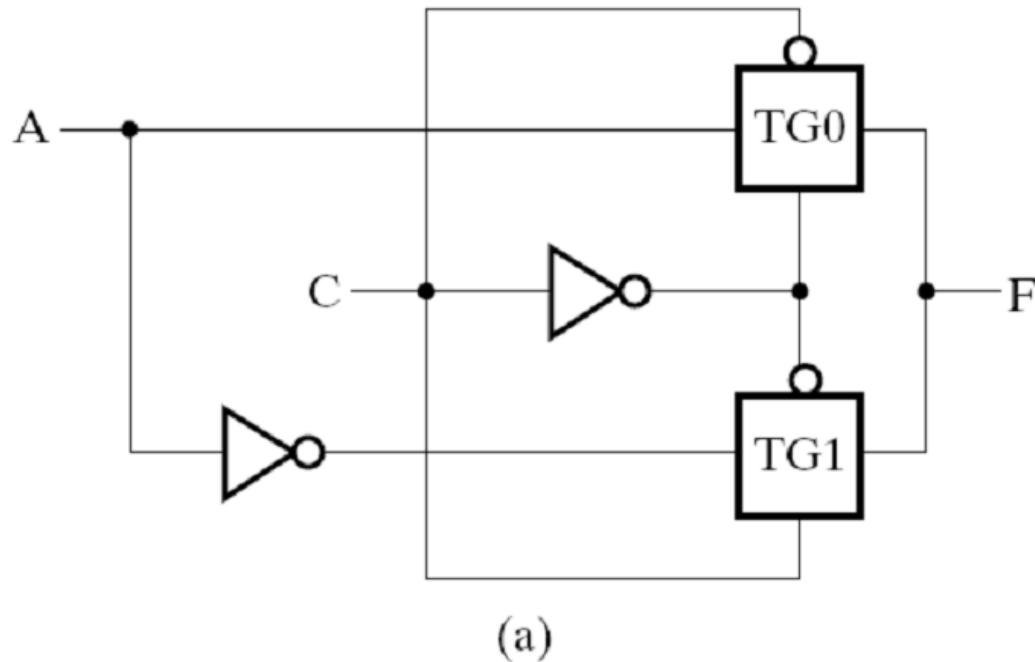
- **Essential Prime Implicants**

Appendix B: Transmission gate

- A transmission gate is similar to a **relay** that can conduct in both directions or block by a control signal with **almost any voltage potential**.



XOR Function with transmission gate



A	C	TG1	TG0	F
0	0	No path	Path	0
0	1	Path	No path	1
1	0	No path	Path	1
1	1	Path	No path	0

(b)