Ollscoil na Gaillimhe
University of Galway

# CT101 Computing Systems

Dr. Bharathi Raja Chakravarthi
Lecturer-above-the-bar
Email: bharathi.raja@universityofgalway.ie

University
ofGalway.ie

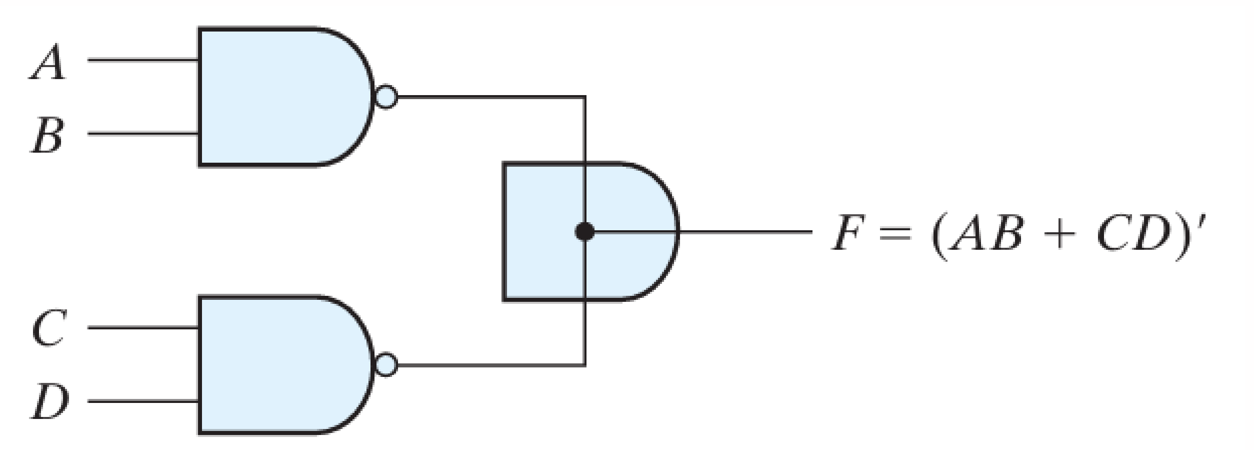
Ollscoil na Gaillimhe
University of Galway

# Other Two-Level Implementations

University
*of*Galway.ie

# Introduction

- In integrated circuits, NAND and NOR gates are commonly used.
- These gates often allow wire connections between their outputs to create specific logic functions, known as "**wired logic**."
- For example, open-collector TTL NAND gates perform wired-AND logic when connected together.
- The wired-AND logic performed with two NAND gates is depicted in Figure (a).



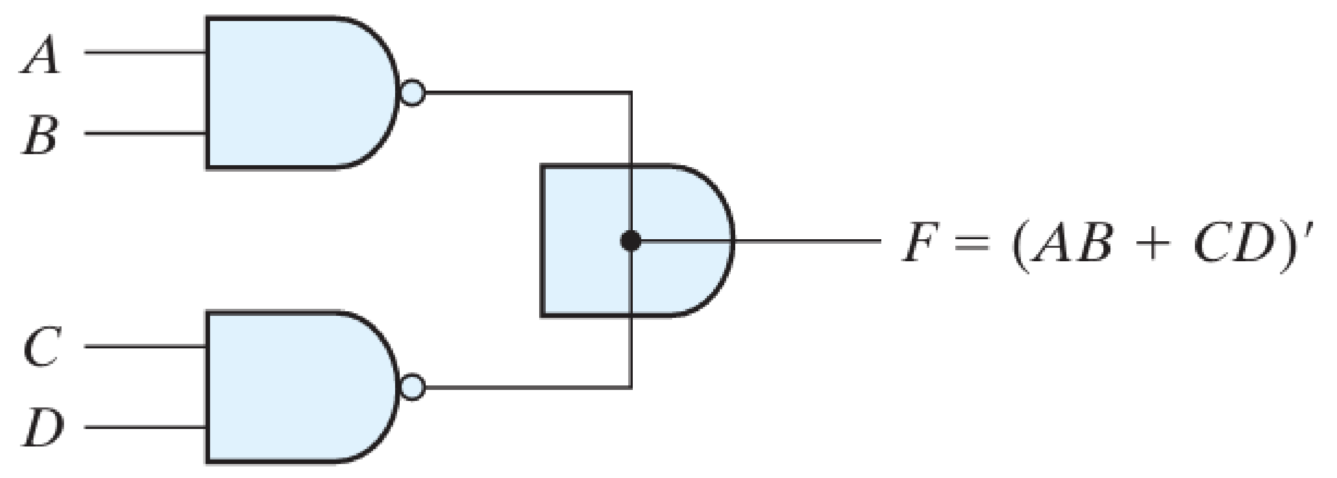
(a) Wired-AND in open-collector TTL NAND gates.

(AND–OR–INVERT)

# Introduction

- The wired-AND logic is symbolically represented, with lines passing through the center of the gate to distinguish it.
- The wired-AND gate is a symbolic representation of the function achieved through a specific wired connection, not a physical gate.
- The logic function implemented by this wired-AND gate is

  **F = (AB)'(CD) = (AB + CD) = (A + B)(C + D)**
- This function is referred to as an **AND-OR-INVERT function**.



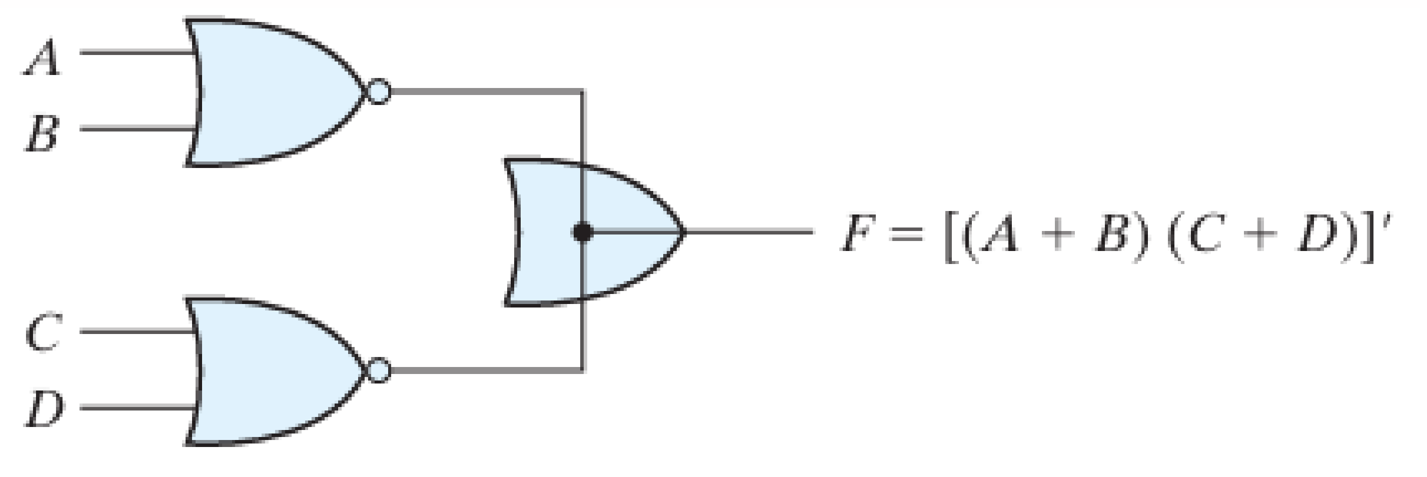
(a) Wired-AND in open-collector TTL NAND gates.

(AND–OR–INVERT)

## Introduction

- ECL gates with NOR outputs can be connected to perform a **wired-OR function**.
- The logic function implemented by the circuit of Figure (b) is and is called an **OR–AND–INVERT** function.

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



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

## Introduction

- A wired-logic gate does not produce a physical second-level gate, since it is just a wire connection.
- For discussion purposes, we will consider the circuits of Figure (a) and (b) as two-level implementations.
- The **first level** consists of NAND (or NOR) gates and the **second level** has a single AND (or OR) gate.
- The wired connection in the graphic symbol will be omitted in subsequent discussions.

# Nondegenerate Forms

- **Four types** of gates are considered: AND, OR, NAND, and NOR.
- When one type of gate is assigned for the first level and one for the second level, there are 16 possible combinations of two-level forms.

**Note:**

The same type of gate can be in the first and second levels, as in a NAND–NAND implementation.

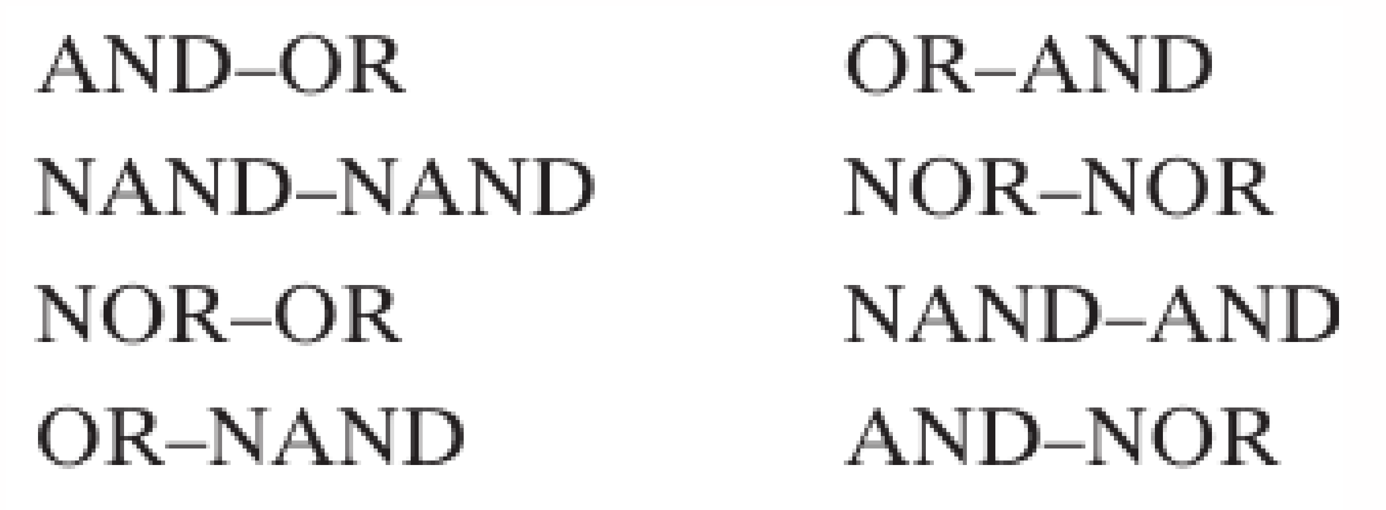
- Eight of these combinations are said to be degenerate forms because they degenerate to a single operation.

## Nondegenerate Forms

- This can be seen from a circuit with AND gates in the first level and an AND gate in the second level.
- The output of the circuit is merely the AND function of all input variables.
- The remaining eight nondegenerate forms produce an implementation in sum-of-products form or product-of-sums form.

# Nondegenerate Forms

- The **eight nondegenerate** forms are as follows:

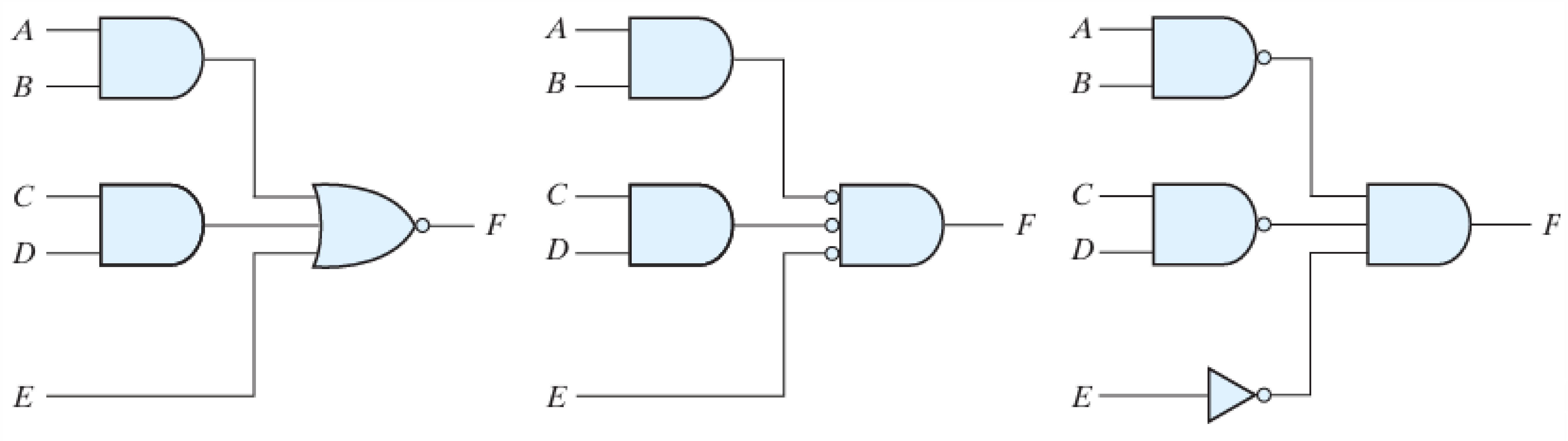


| | |
|---|---|
| AND–OR | OR–AND |
| NAND–NAND | NOR–NOR |
| NOR–OR | NAND–AND |
| OR–NAND | AND–NOR |

- The first gate listed in each of the forms constitutes the first level in the implementation.
- The second gate listed is a single gate placed on the second level.
- Note that any two forms listed on the same line are duals of each other.

# AND–OR–INVERT Implementation

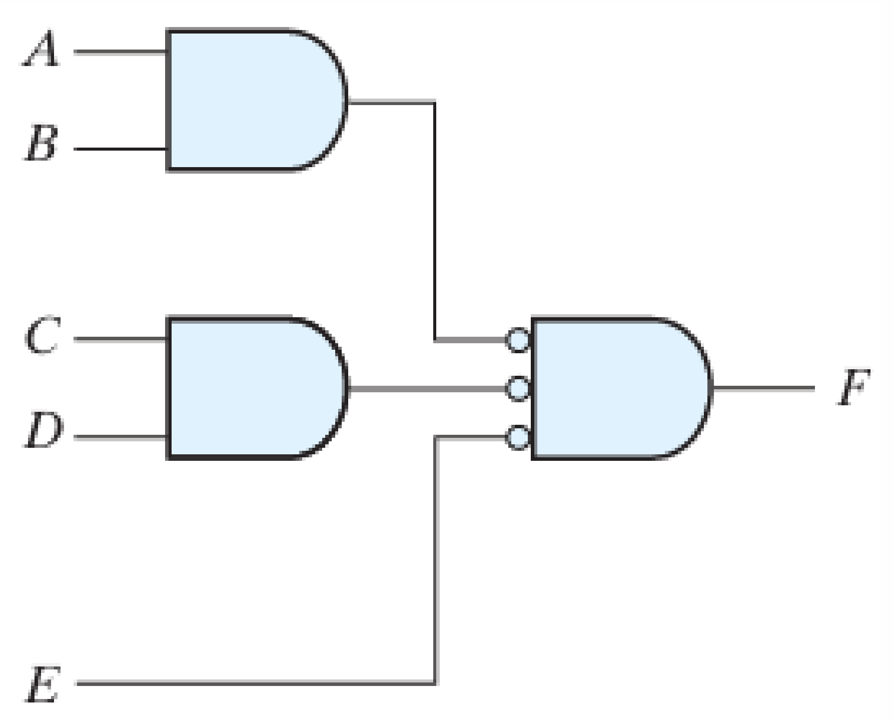
- NAND–AND and AND–NOR forms are equivalent and perform the AND–OR–INVERT function.
- The AND–NOR form resembles the AND–OR form but includes an inversion done by the bubble in the output of the NOR gate.
- It implements the function **F = (AB + CD + E)'**.



(a) AND–NOR (b) AND–NOR (c) NAND–AND

## AND–OR–INVERT Implementation

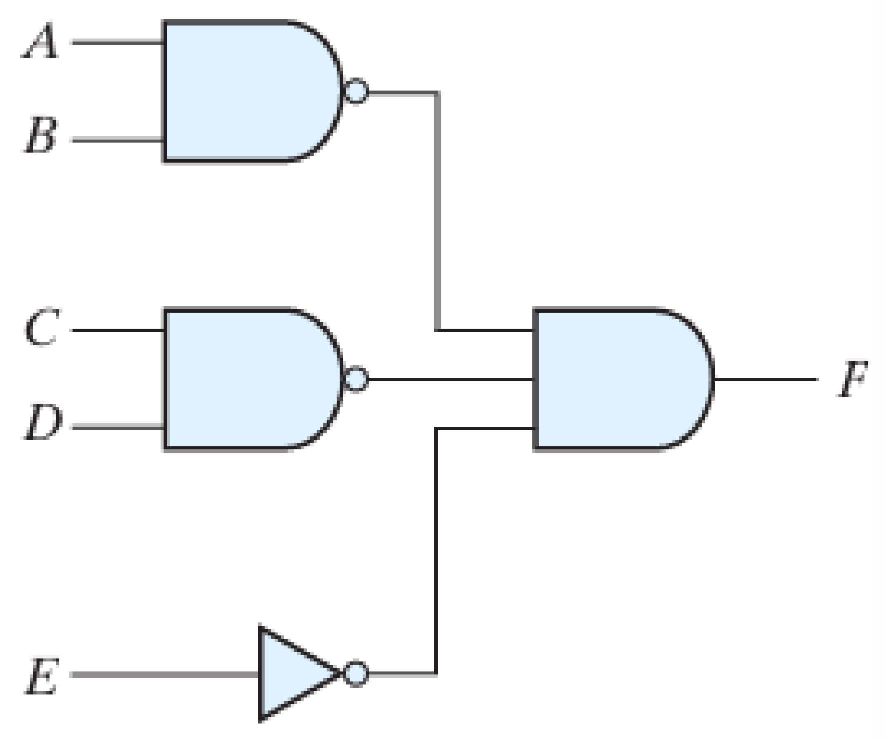
- The alternative graphic symbol for the NOR gate is used in the diagram (Figure b).
- A single variable, E, is not complemented in this diagram because the only change is in the graphic symbol of the NOR gate.
- The bubble from the input terminal of the second-level gate is moved to the output terminals of the first-level gates.



(b) AND–NOR

# AND–OR–INVERT Implementation

- An inverter is needed for the single variable to compensate for the bubble, or the inverter can be removed if input E is complemented.
- The circuit in Figure c is a **NAND–AND form**, which was previously shown to implement the AND–OR–INVERT function.



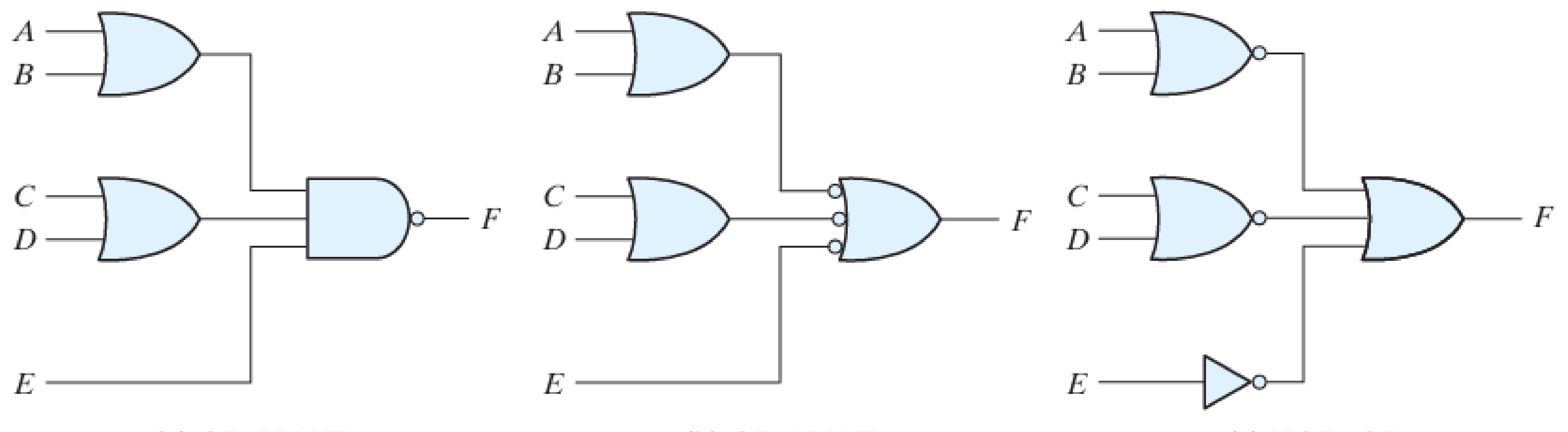
(c) NAND–AND

# AND–OR–INVERT Implementation

- An AND–OR implementation requires an expression in sum-of-products form.
- The AND–OR–INVERT implementation is similar to AND–OR, except for the inversion.
- If the complement of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement F' with the AND–OR part of the function.
- When F' passes through the always present output inversion (the INVERT part), it will generate the output F of the function.
- An example for the AND–OR–INVERT implementation will be shown subsequently.

# OR–AND–INVERT Implementation

- The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function, as shown in Figure below.
- The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate.
- It implements the function **F = [(A + B)(C + D)E]’**

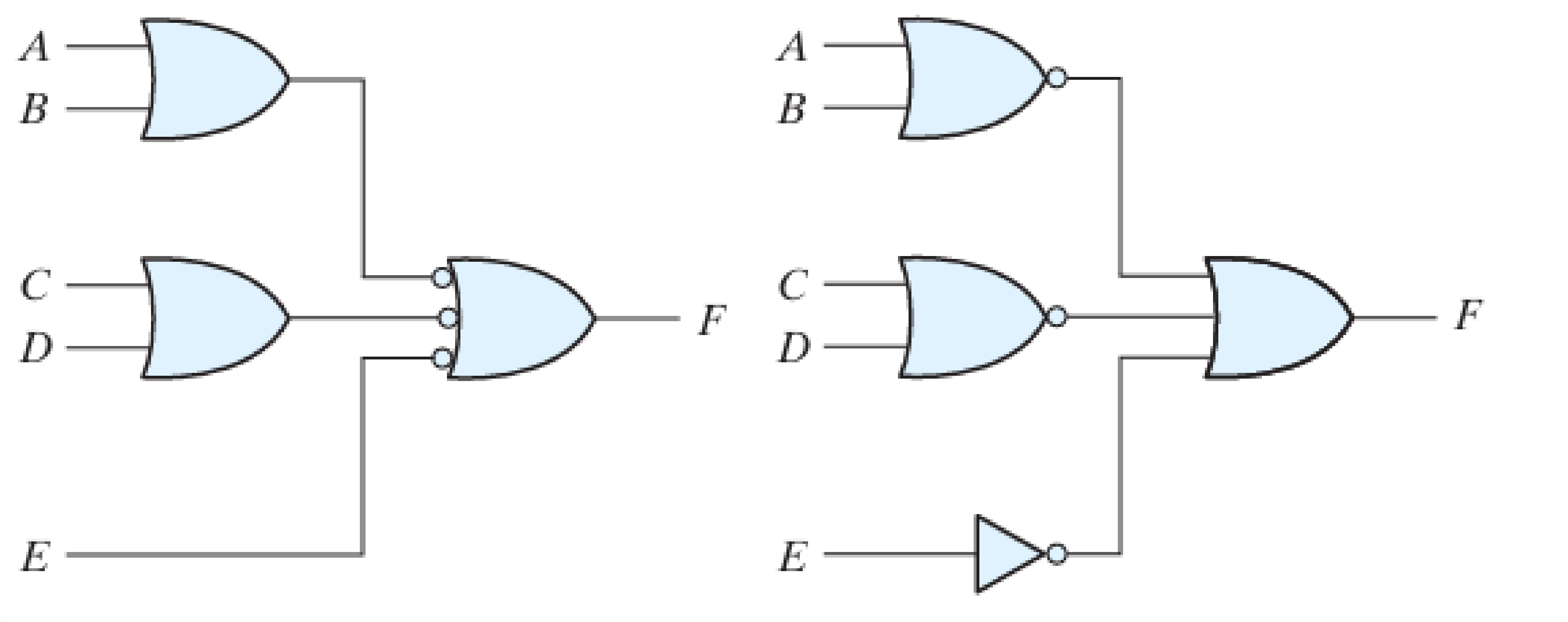


(a) OR–NAND (b) OR–NAND (c) NOR–OR

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

# OR–AND–INVERT Implementation

- By using the alternative graphic symbol for the NAND gate, we obtain the diagram of Figure (b).
- The circuit in Figure (c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates.
- The circuit of Figure (c) is a NOR–OR form.



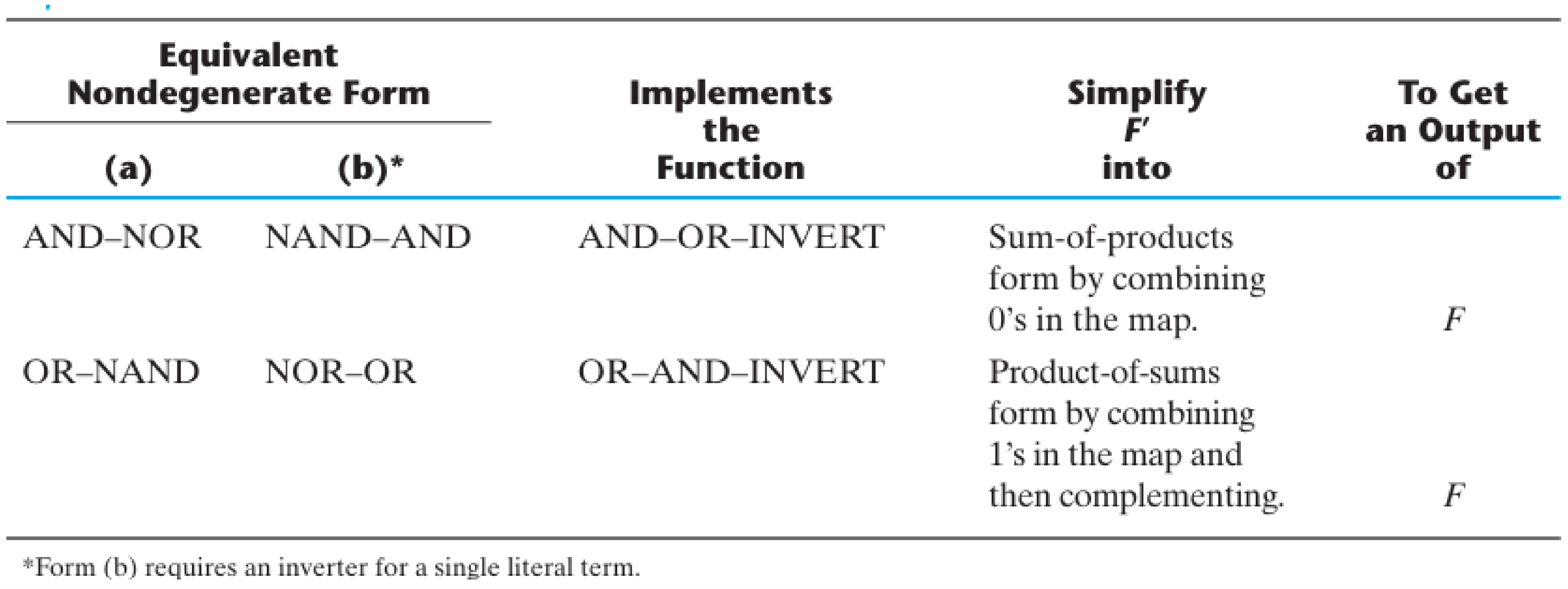
(b) OR–NAND (c) NOR–OR

# OR–AND–INVERT Implementation

- The OR–AND–INVERT implementation requires an expression in product-of-sums form.
- If the complement of the function is simplified into that form, we can implement F’ with the OR–AND part of the function.
- When F’ passes through the INVERT part, we obtain the complement of F’, or F, in the output.

# Tabular Summary

- The Table below summarizes the procedures for implementing a Boolean function in any one of the four 2-level forms.
- Because of the INVERT part in each case, it is convenient to use the simplification of F’ (the complement) of the function.

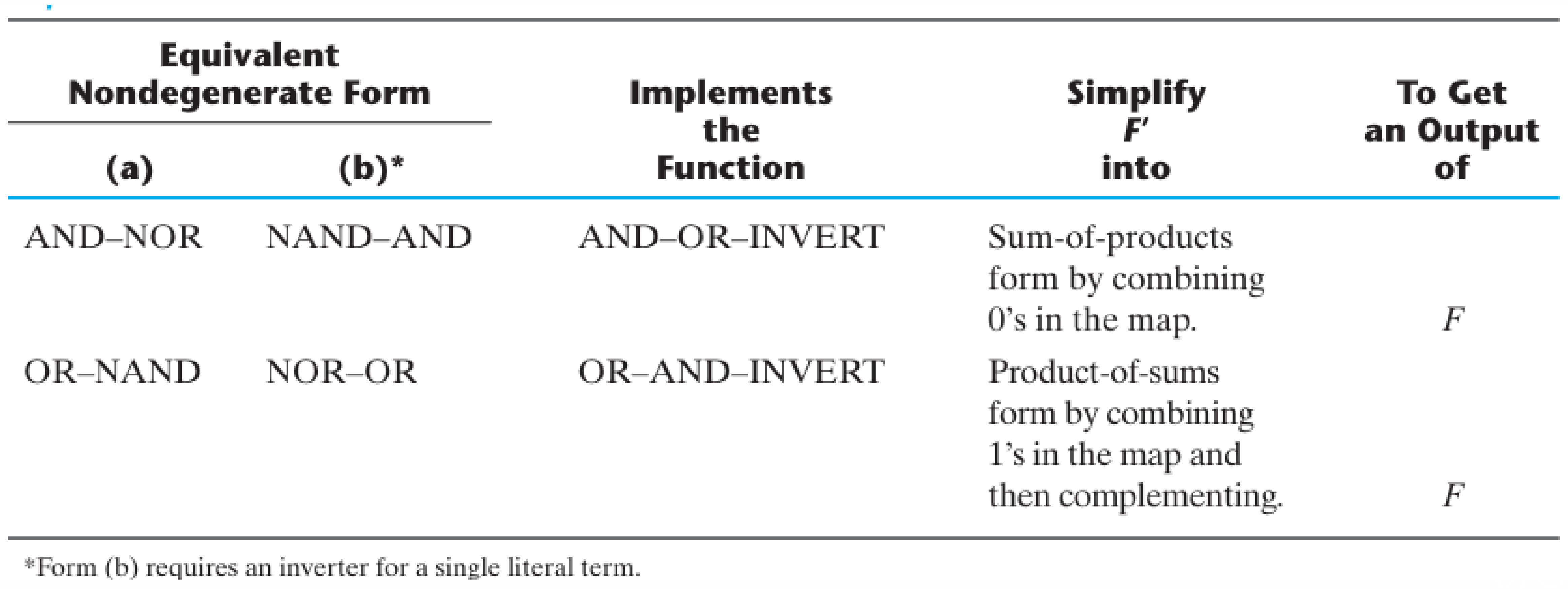


| **Equivalent Nondegenerate Form** | | **Implements the Function** | **Simplify *F*′ into** | **To Get an Output of** |
|---|---|---|---|---|
| **(a)** | **(b)*** | | | |
| AND–NOR | NAND–AND | AND–OR–INVERT | Sum-of-products form by combining 0’s in the map. | *F* |
| OR–NAND | NOR–OR | OR–AND–INVERT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

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

# Tabular Summary

- When F’ is implemented in one of these forms, we obtain the complement of the function in the **AND–OR or OR–AND** form.
- The four 2-level forms invert this function, giving an output that is the complement of F.
- This is the normal output F.



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

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

## Example

Implement the function of $F = x'y'z' + xyz'$ with the four 2-level forms.

- The complement of the function is simplified into sum-of-products form by combining the 0's in the map:

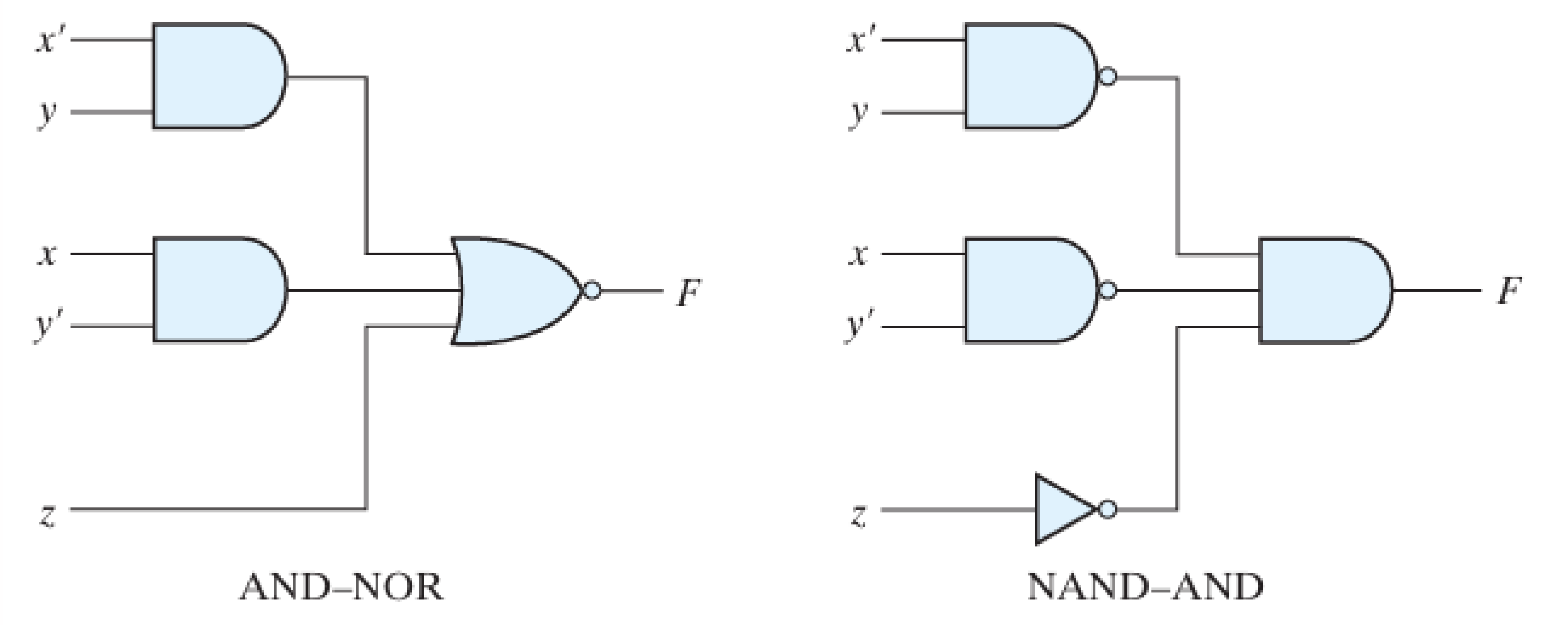
$$F' = x'y + xy' + z$$

- The normal output for this function can be expressed as

$$F = (xy + xy + z)$$

## Example

- The normal output is in the AND–OR–INVERT form. The AND–NOR and NAND–AND implementations are shown in Figure (b).
- Note that a one-input NAND, or inverter, gate is needed in the NAND–AND implementation, but not in the AND–NOR case.
- The inverter can be removed if we apply the input variable **z' instead of z**.



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

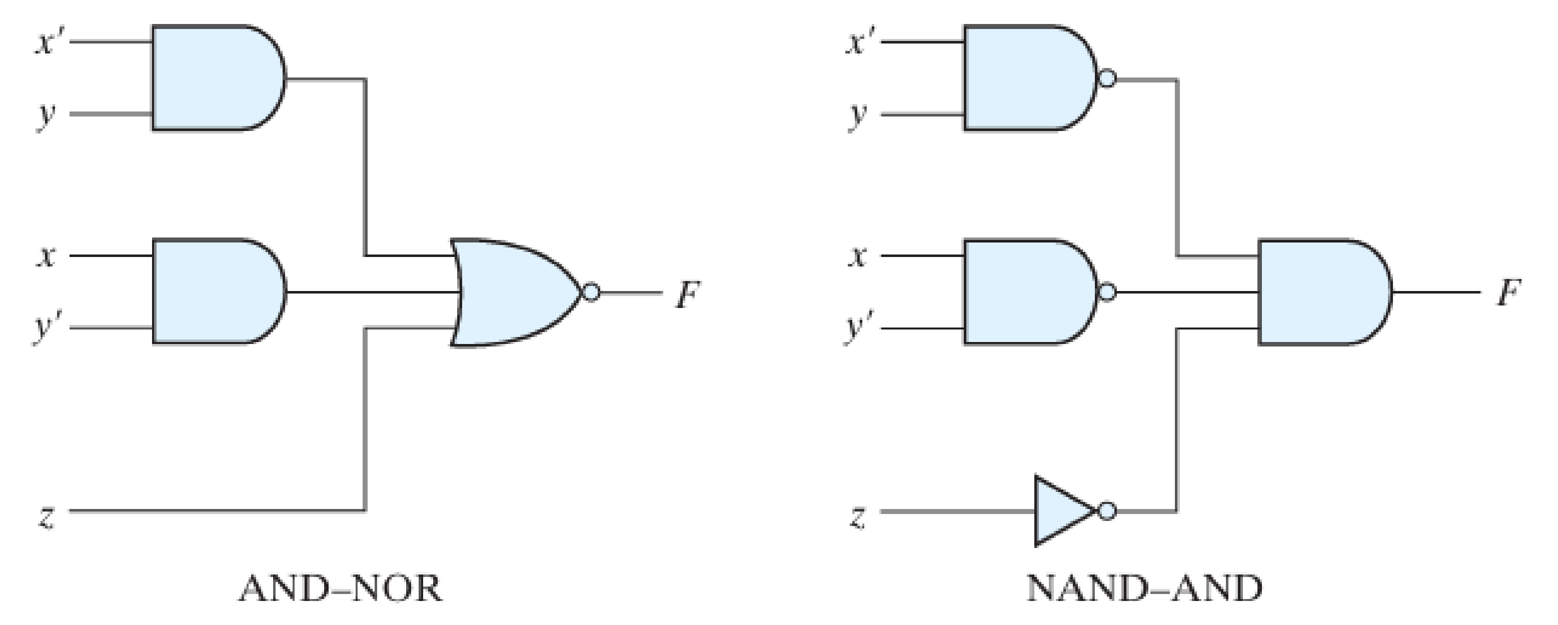
OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

## Example

- The OR–AND–INVERT forms require a simplified expression of the complement of the function in product-of-sum form.
- To obtain this expression, we first combine the 1's in the map:

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

- Then we take the complement of the function: **F' = (x + y + z)(x' + y' + z)**

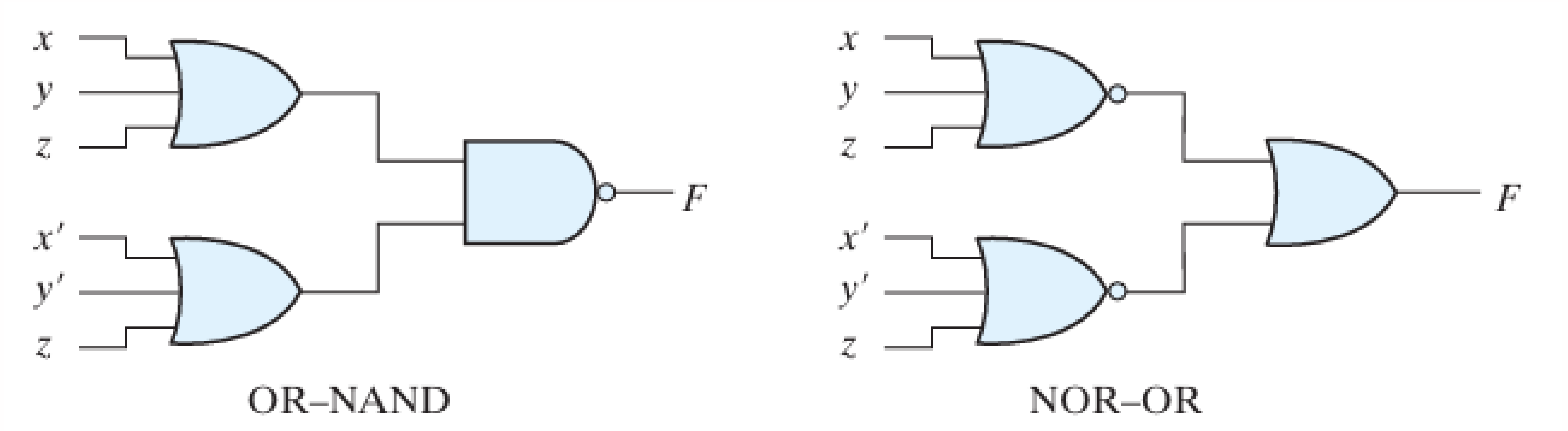


(b) $F = (x'y + xy' + z)'$

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

## Example

- The normal output F can now be expressed in the form **F= [(x+y+z)(x +y +z)]** which is the OR–AND–INVERT form.
- From this expression, we can implement the function in the **OR–NAND and NOR–OR** forms, as shown in Figure (c).



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

Ollscoil na Gaillimhe
University of Galway

# Exclusive-OR Function

University
of Galway.ie

# Introduction

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

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

- The exclusive-OR is equal to 1 if only x is equal to 1 or if only y is equal to 1 (i.e., x and y differ in value), but not when both are equal to 1 or when both are equal to 0.

- The exclusive NOR, also known as equivalence, performs the following Boolean operation:

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

## Introduction

- The exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0.
- The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

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

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

$$\begin{aligned} x \oplus 0 &= x \\ x \oplus 1 &= x' \\ x \oplus x &= 0 \\ x \oplus x' &= 1 \\ x \oplus y' &= x' \oplus y = (x \oplus y)' \end{aligned}$$

## Introduction

- Any of these identities can be proven with a truth table or by replacing the $\oplus$ operation by its equivalent Boolean expression.

- It can be shown that the exclusive-OR operation is both commutative and associative; that is,

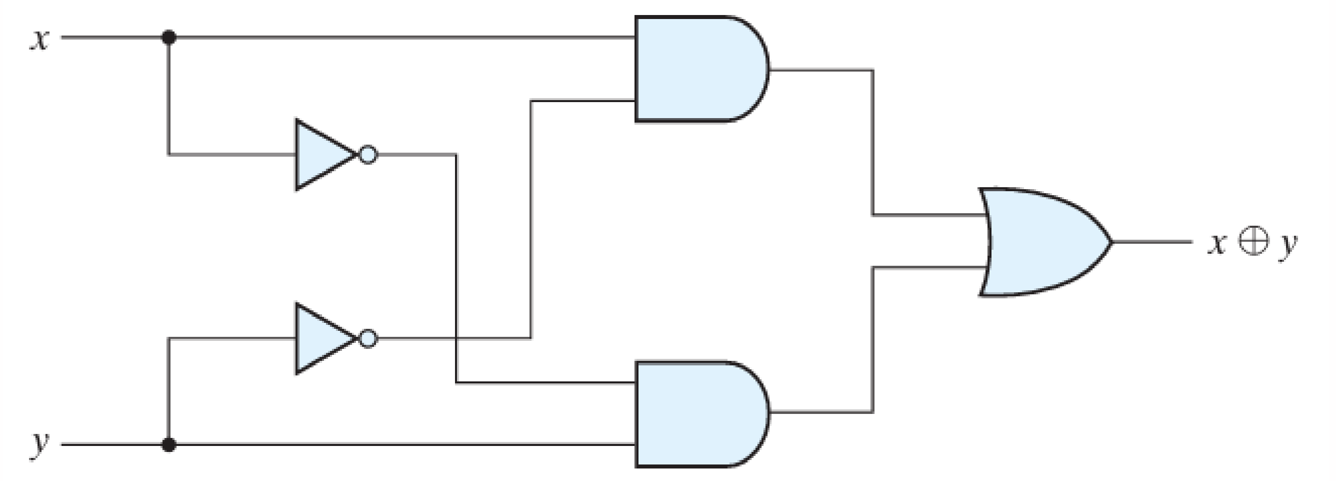
$$A \oplus B = B \oplus A \text{ and } (A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

# Introduction

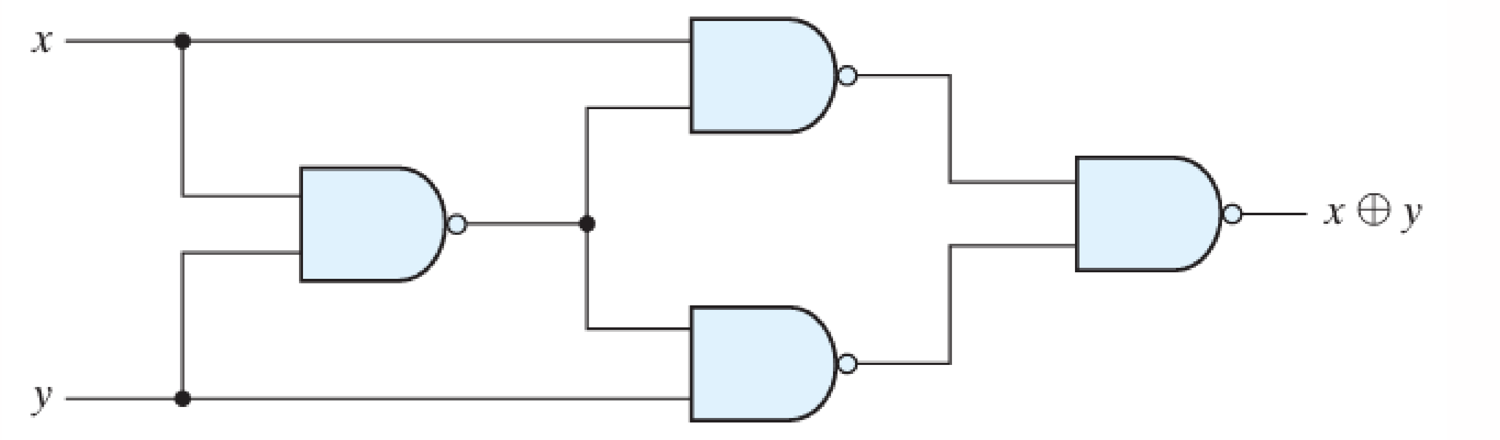
- In an exclusive-OR (XOR) gate, the two inputs can be interchanged without affecting the operation.
- A three-variable XOR operation can be evaluated in any order, allowing the expression of three or more variables without parentheses.
- This implies the possibility of using exclusive-OR gates with three or more inputs.
- However, **multiple-input exclusive-OR gates** are difficult to fabricate with hardware.
- Even a two-input function is typically constructed with other types of gates.

# Introduction

- A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Figure (a).



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



(b) Exclusive-OR with NAND gates

- Figure (b) shows the implementation of the exclusive-OR with four NAND gates.

- The first NAND gate performs the operation **$(xy)' = (x' + y')$.**

- The other two-level NAND circuit produces the sum of products of its inputs:

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

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

# Introduction

- Only a limited number of Boolean functions can be expressed in terms of exclusive-OR (XOR) operations.
- The **XOR function** is encountered frequently in the design of digital systems.
- It is particularly useful in **arithmetic operations** and **error detection and correction** circuits.

## Odd Function

- The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression.
- In particular, the three-variable case can be converted to a Boolean expression as follows:

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

## Odd Function

- Three-variable exclusive-OR (XOR) function evaluates to 1 under two conditions:
  - When only one variable is equal to 1.
  - When all three variables are equal to 1.

- Unlike the two-variable XOR, where only one variable must be 1, in the case of three or more variables, an odd number of variables (1, 3, 5, etc.) must be equal to 1 for the function to yield 1.

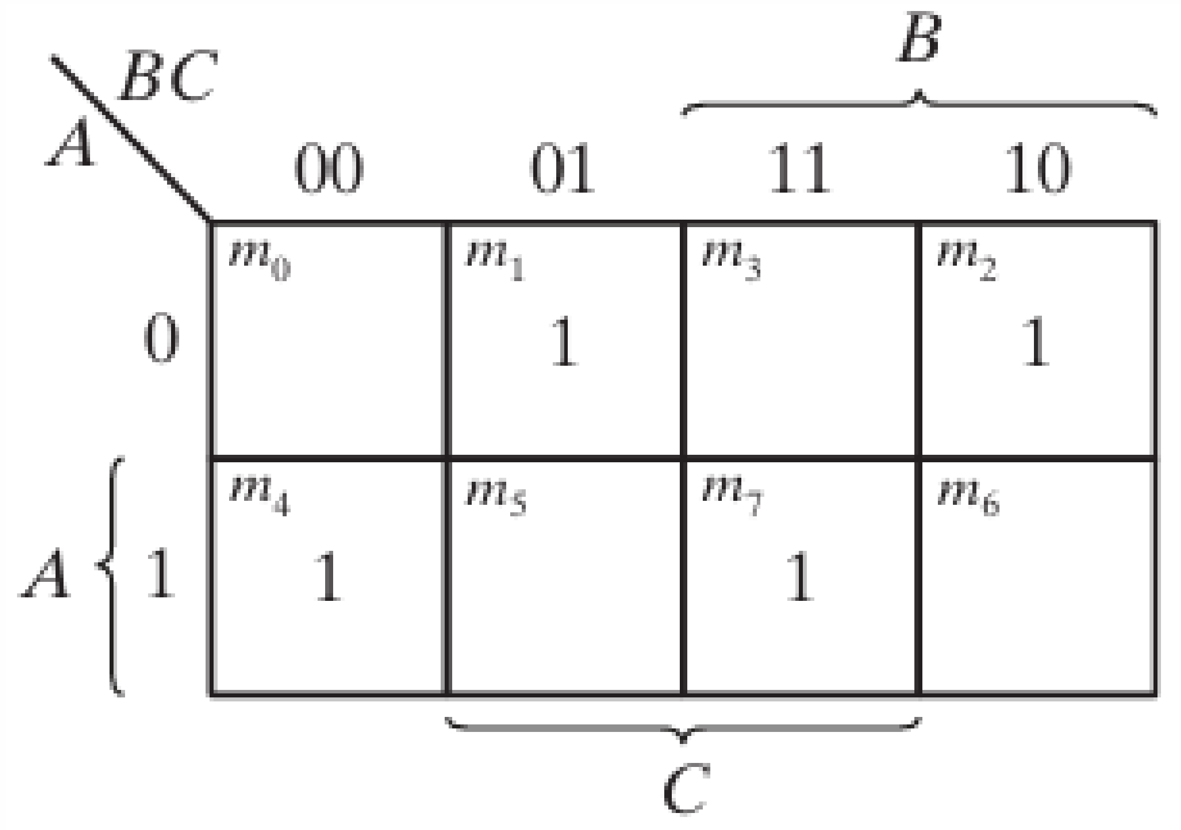
- The multiple-variable exclusive-OR operation is defined as an "**odd function**."

# Odd Function

- The three-variable exclusive-OR function is expressed as the logical sum of four minterms.
- These minterms have binary numerical values of **001, 010, 100, and 111**.
- All of these binary values have an odd number of 1's (1, 1, 1, and 3, respectively).
- The remaining four minterms (**000, 011, 101, and 110**) are not included in the function.
- These excluded minterms have an even number of 1's in their binary numerical values.
- In general, for an n-variable exclusive-OR function is an odd function defined as the logical sum of **2n/2** minterms whose binary numerical values have an odd number of 1's.

# Odd Function

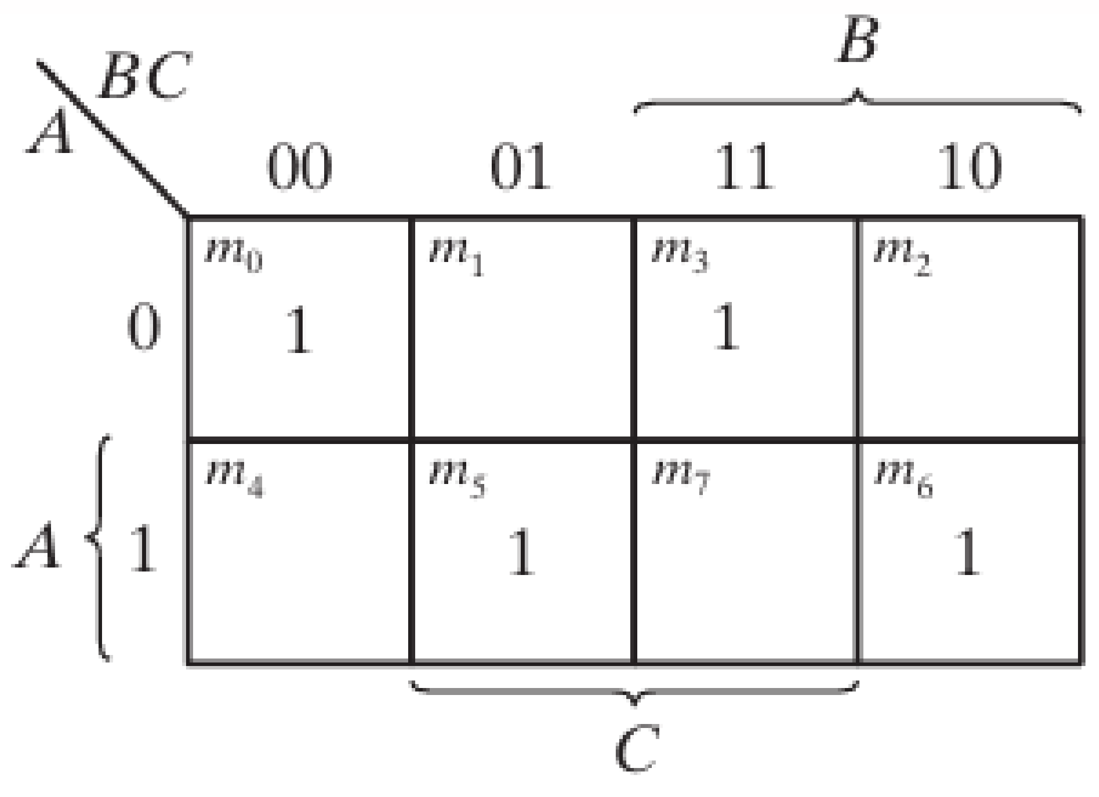
- The concept of an odd function can be clarified through mapping.
- Figure (a) illustrates the map for the three-variable exclusive-OR function.
- The four minterms of this function are equally spaced on the map.



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

## Odd Function

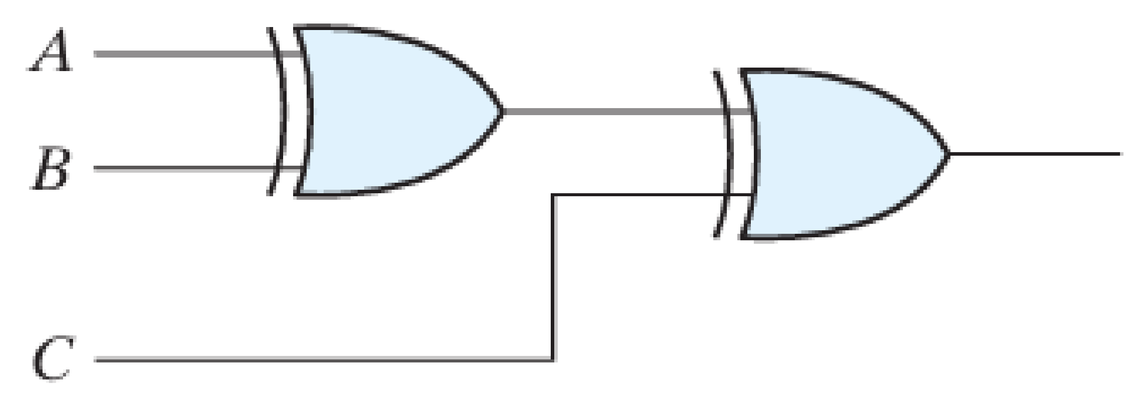
- Odd functions are identified by the binary values of their minterms having an odd number of 1's.
- The complement of an odd function results in an even function.
- Figure (b) depicts a three-variable even function, which evaluates to 1 when an even number of its variables are set to 1, including the case where none of the variables is 1.



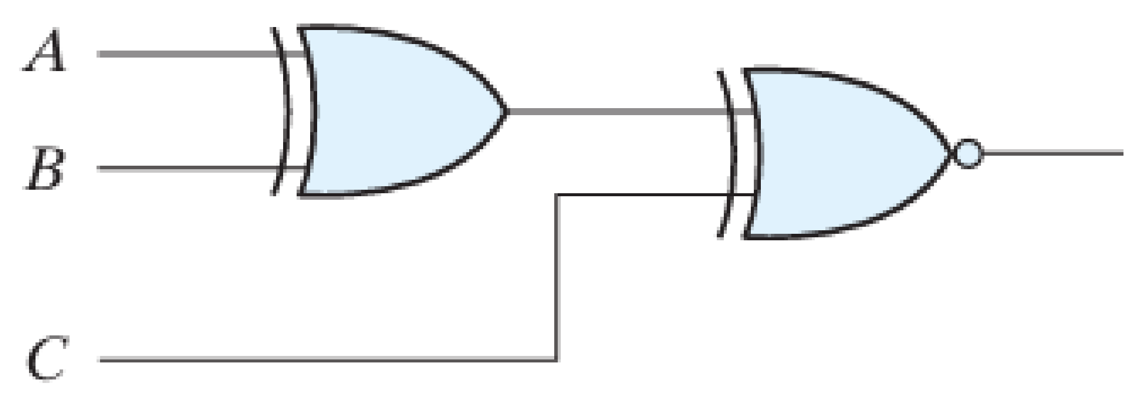
(b) Even function $F = (A \oplus B \oplus C)'$

# Odd Function

- A **three-input odd** function is implemented using **two-input exclusive-OR** gates (Figure (a)).
- The complement of an odd function is achieved by replacing the output gate with an exclusive-NOR gate (Figure (b)).
- Now, let's consider the four-variable exclusive-OR operation.



(a) 3-input odd function



(b) 3-input even function

## Odd Function

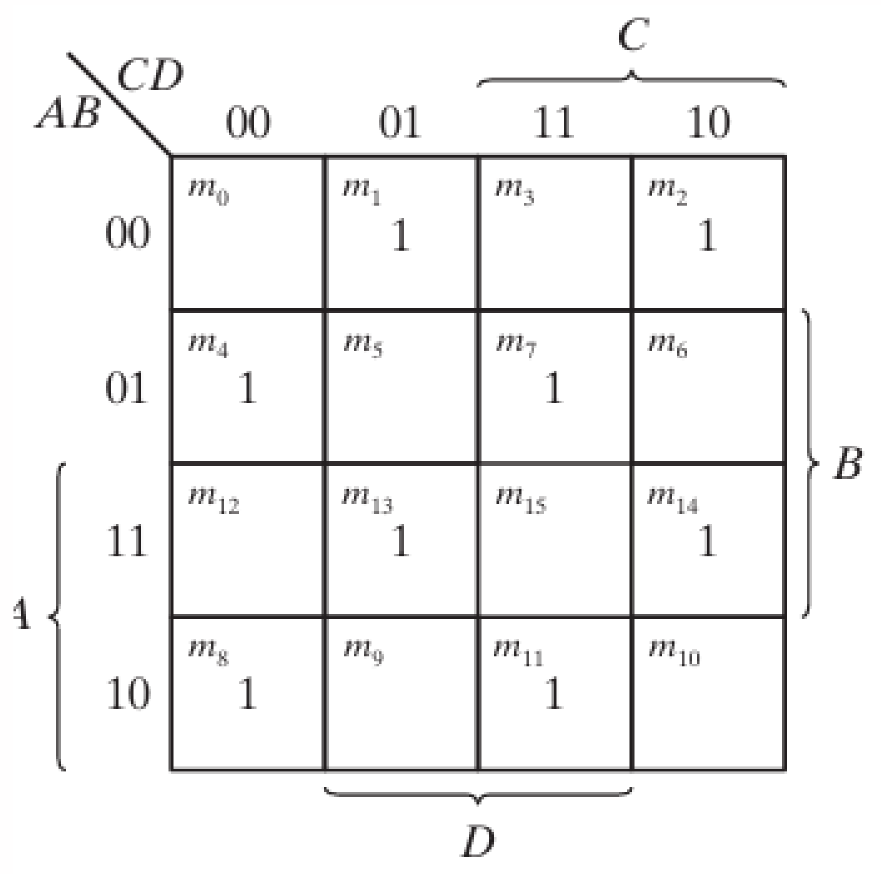
- we can obtain the sum of minterms for this function:

$$\begin{aligned} A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\ &= \Sigma(1, 2, 4, 7, 8, 11, 13, 14) \end{aligned}$$

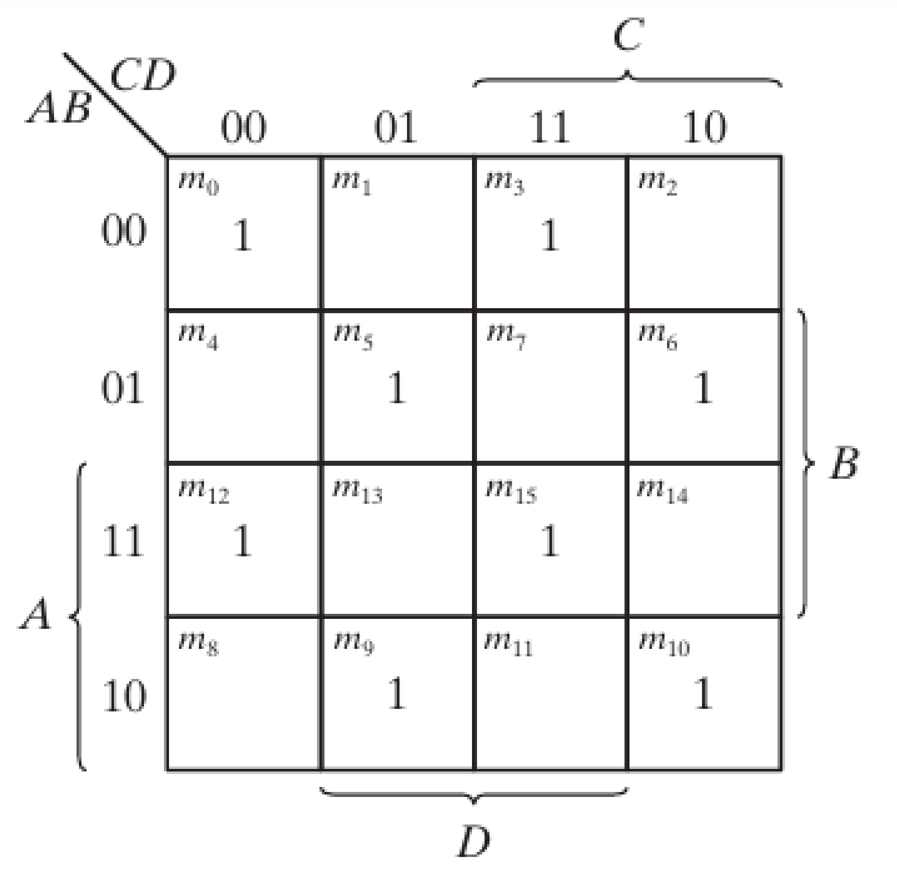
- For a four-variable Boolean function, there are 16 minterms.
- Half of the minterms have binary numerical values with an odd number of 1's, and the other half have an even number of 1’s.

# Odd Function

- The binary value of a minterm is determined by its position in the map, based on row and column numbers.
- The map in Figure (a) represents the four-variable exclusive-OR function, which is an odd function.
- The complement of an odd function is an even function, as shown in Figure (b).



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



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

## Parity Generation and Checking

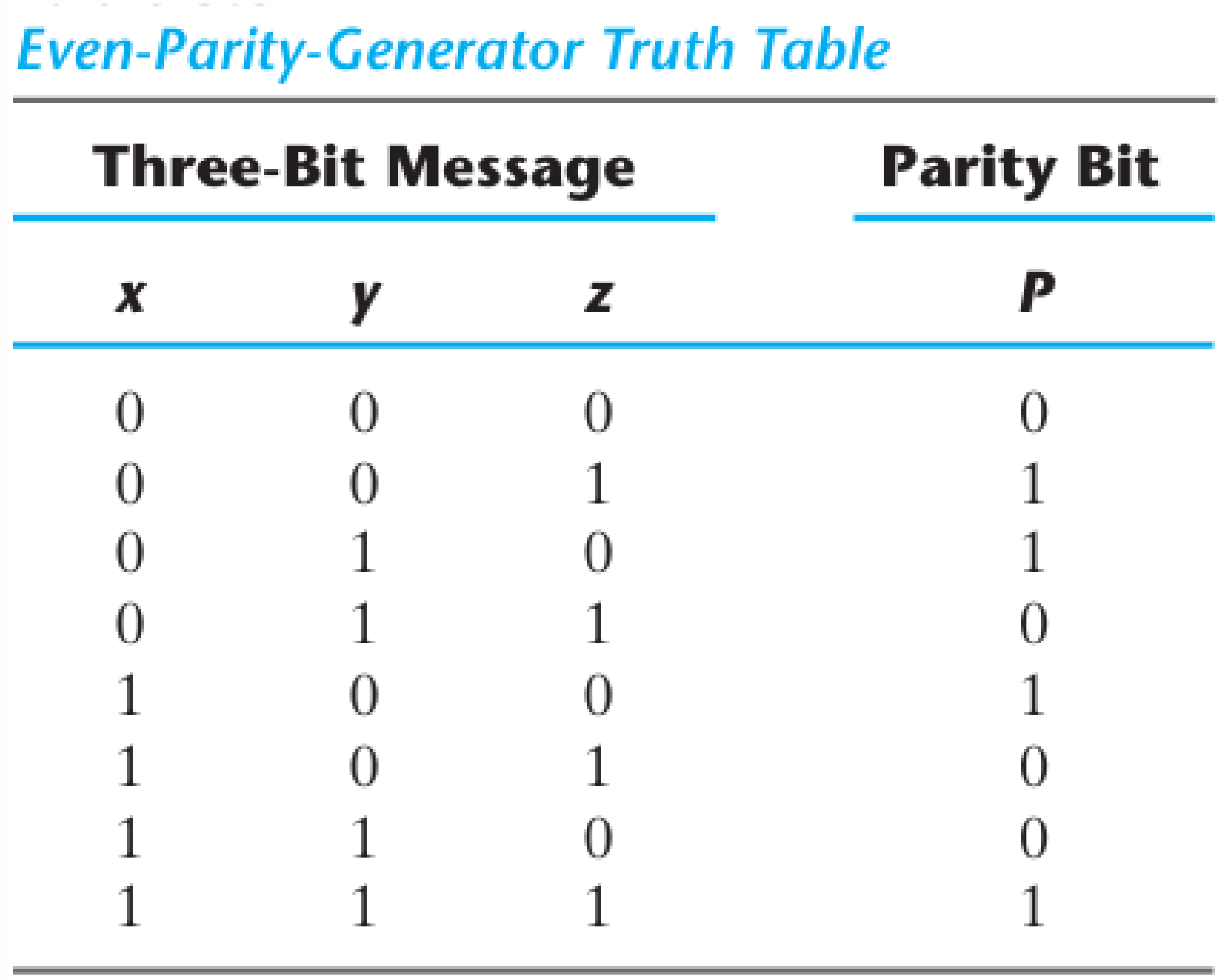
- Exclusive-OR (XOR) functions have practical applications in systems requiring error detection and correction codes.
- A parity bit is commonly used for error detection during the transmission of binary information.
- A parity bit is an additional bit added to a binary message to ensure that the total number of 1's in the message (including the parity bit) is either odd or even.

# Parity Generation and Checking

- The transmitted message, along with the parity bit, is received and checked for errors at the receiving end.
- Error detection occurs when the checked parity doesn't match the transmitted one.
- The circuit responsible for generating the parity bit in the transmitter is called a "**parity generator**."
- The circuit in the receiver that verifies the parity is known as a "**parity checker**."

# Parity Generation and Checking

- An example involves a three-bit message transmitted with an even-parity bit.
- The truth table for the parity generator is shown in the Table below.
- The three bits, labeled as x, y, and z, form the message and serve as inputs to the circuit.
- The output is the parity bit P.



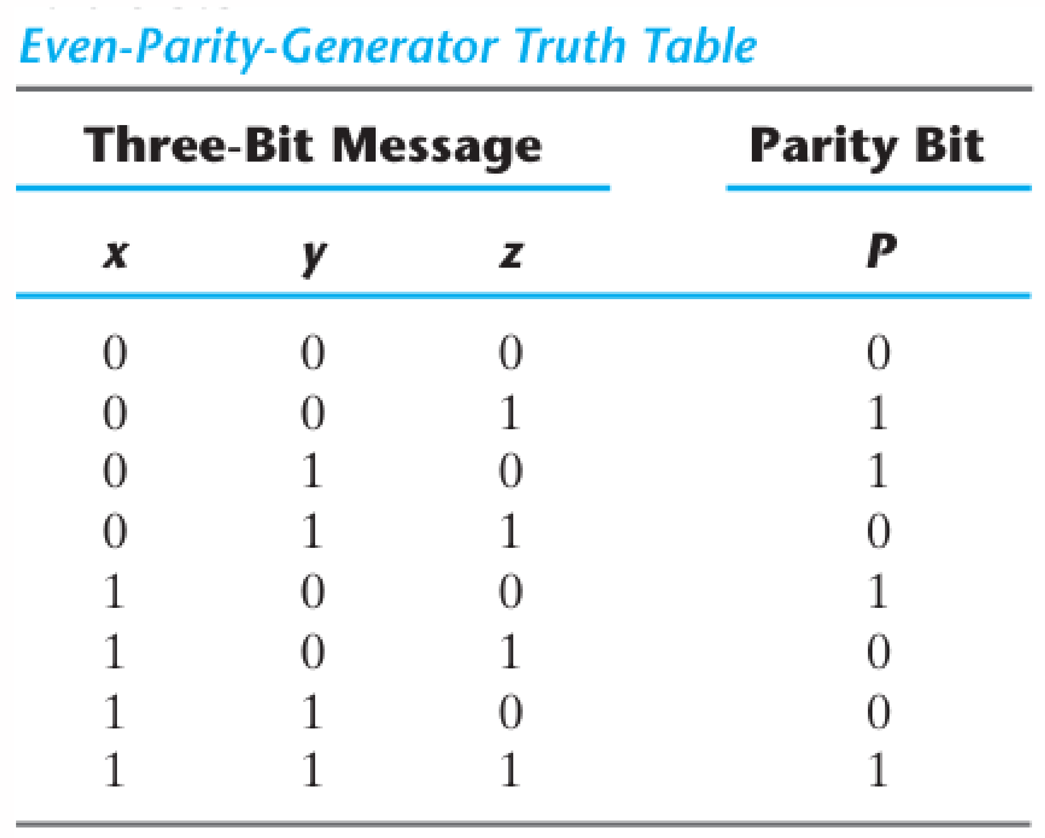
*Even-Parity-Generator Truth Table*

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

# Parity Generation and Checking

- For even parity, P is generated to ensure the total number of 1's (including P) is even.
- P is determined by minterms with an odd number of 1's in their binary values.
- This makes P an odd function, as it equals 1 for minterms with an odd number of 1’s.
- Therefore, P can be expressed as a three-variable exclusive-OR function:

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

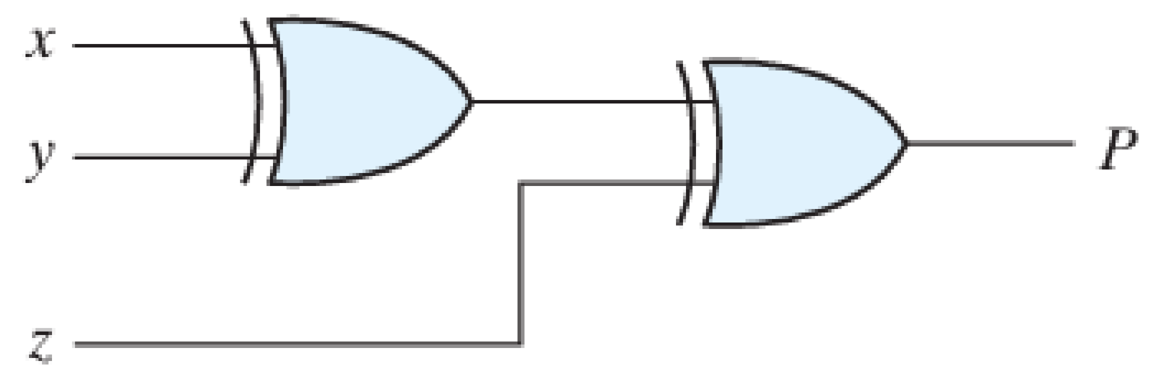


*Even-Parity-Generator Truth Table*

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

# Parity Generation and Checking

- The logic diagram for the parity generator is shown in Fig. (a)



(a) 3-bit even parity generator

- Data transmission involves sending three data bits along with a parity bit for error checking.
- At the destination, a parity-checker circuit is used to examine the received bits for potential errors.

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

# Parity Generation and Checking

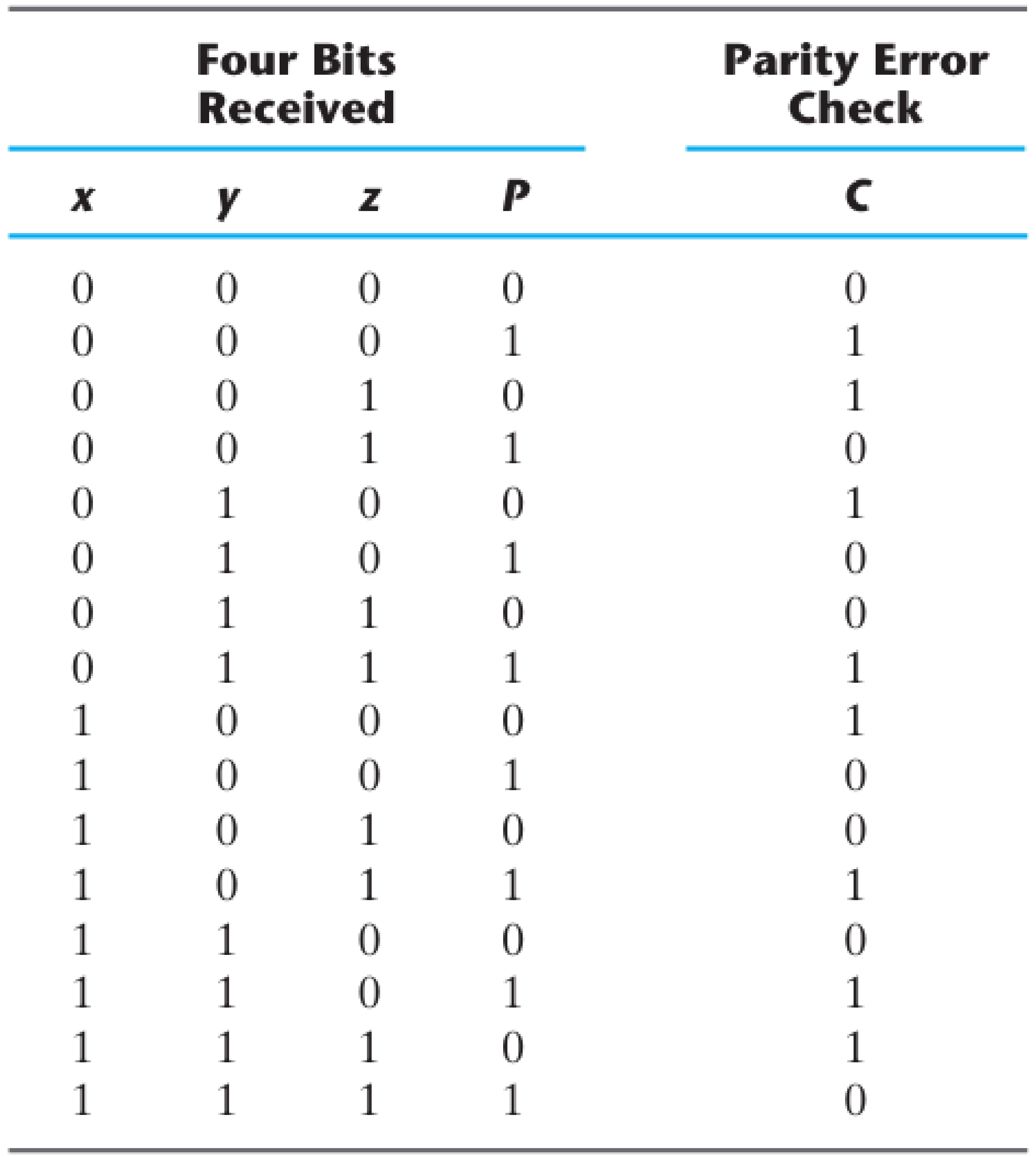
- In this case, even parity is utilized, meaning that the four bits received must contain an even number of 1’s.

- An error during transmission is identified if the four received bits contain an odd number of 1's, suggesting that a bit has changed value during the transmission.

- The output of the parity checker is denoted as "C," and it equals 1 if an error occurs, indicating that the four received bits have an odd number of 1's.

# Parity Generation and Checking

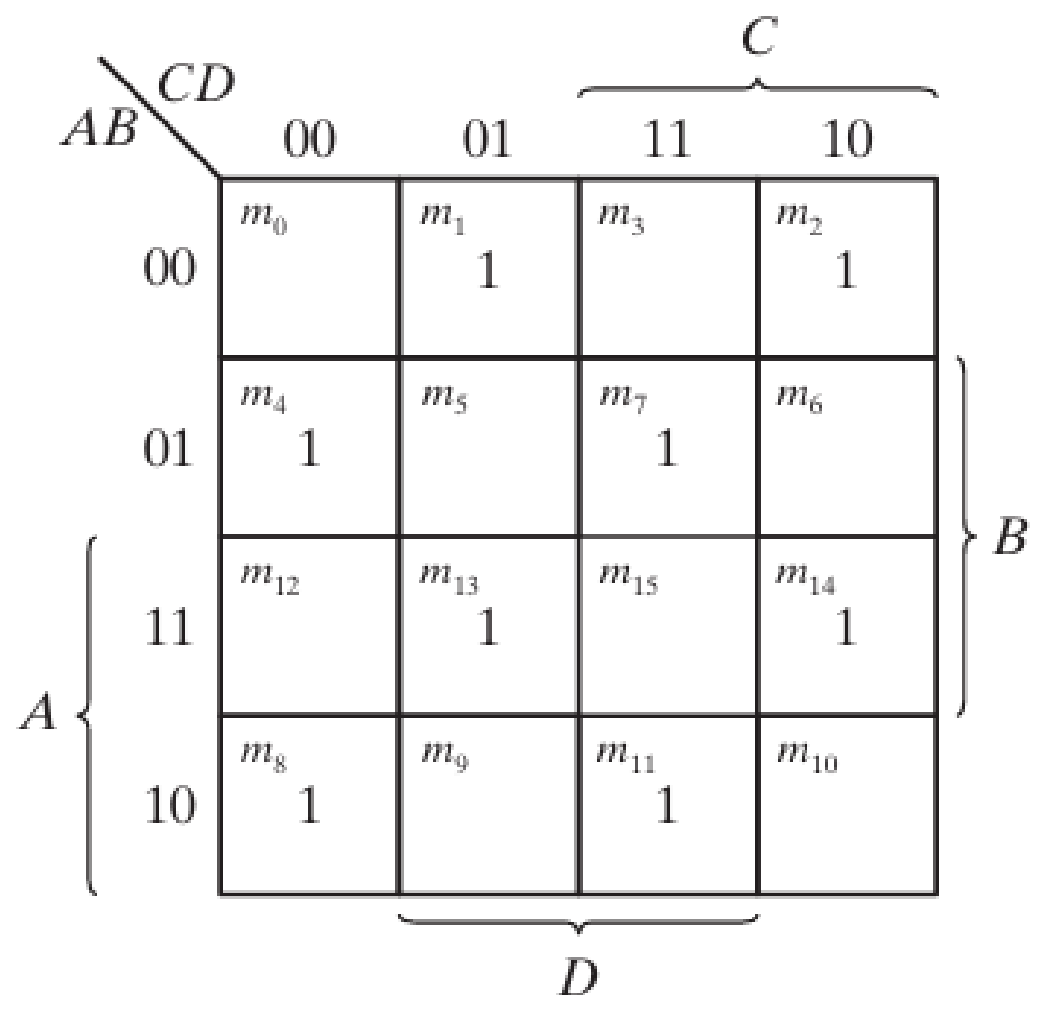
- The table presents the truth table for the even-parity checker, outlining the conditions under which C is equal to 1.



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

# Parity Generation and Checking

- The function C is defined by the eight minterms with binary numerical values that have an odd number of 1's.
- This truth table corresponds to the map shown in Fig. (a), which represents an odd function.



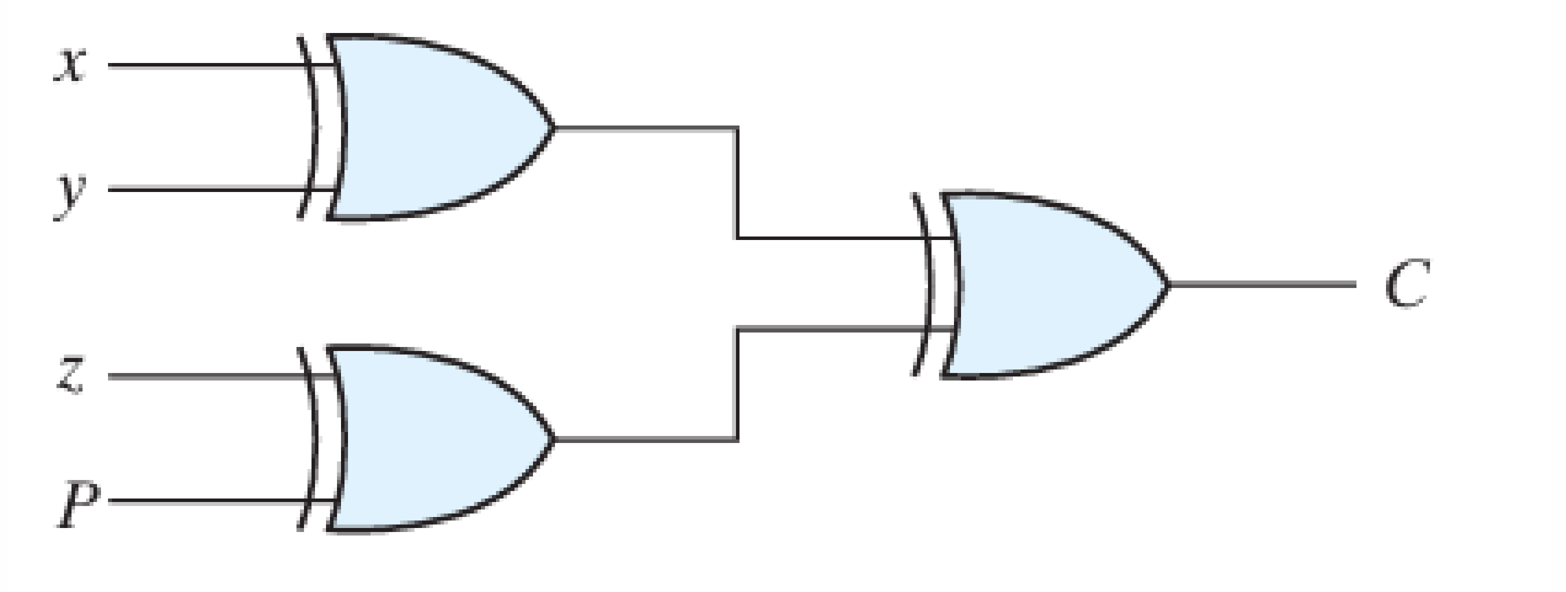
(a) Odd function $F = A \oplus B \oplus C \oplus D$

## Parity Generation and Checking

- The parity checker can be implemented with exclusive OR gates:

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

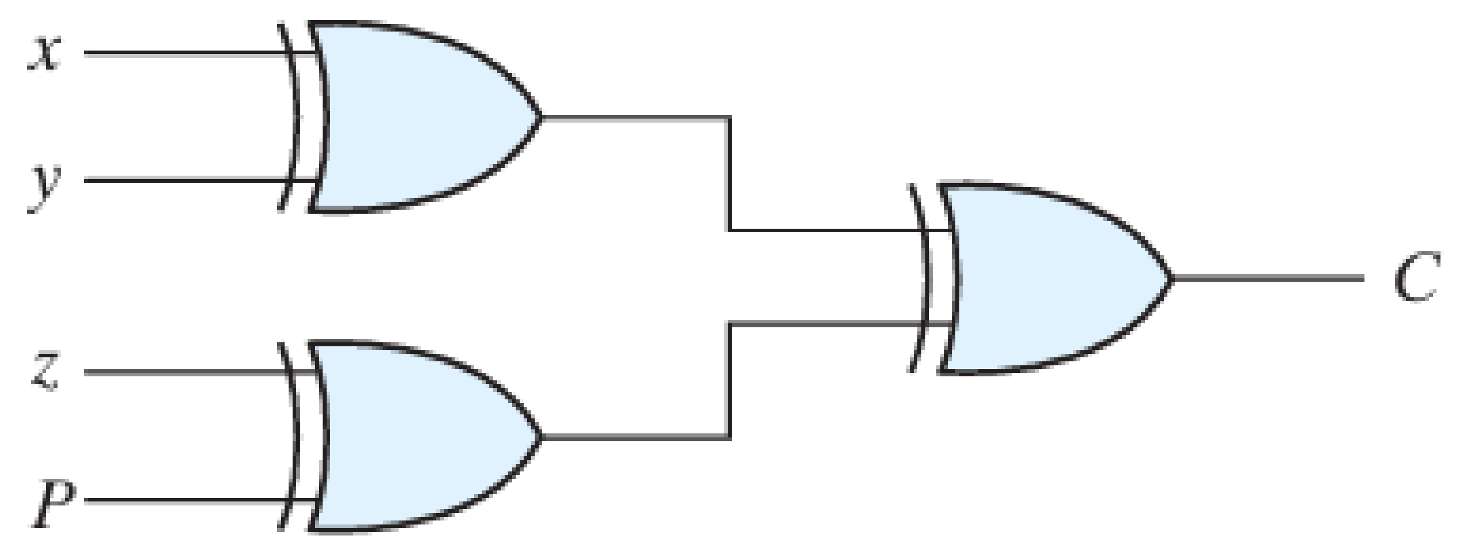
- The logic diagram of the parity checker is shown in Fig. (b).



(b) 4-bit even parity checker

# Parity Generation and Checking

- The parity generator can be efficiently implemented using the circuit depicted in Fig. (b) when the input P is set to logic 0, and the output is designated as P.
- This implementation is feasible because **$z \oplus 0 = z$**, meaning that the value of z passes through the gate without alteration.
- An important advantage of this approach is its versatility, as the same circuit can serve the dual purpose of both parity generation and checking.



(b) 4-bit even parity checker

## References

- Computer Organization and Architecture Designing for Performance Tenth Edition by William Stallings
- Digital Design With an Introduction to the Verilog HDL FIFTH EDITION by M Morris, M. and Michael, D., 2013.

Ollscoil na Gaillimhe
University of Galway

# Thank *you*

University
*of*Galway.ie