Chapter 3

Gate-Level Minimization

3-1

Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

3-2

Why Logic Minimization ?

- Minimize the number of gates used
  - Reduce gate count = reduce cost
- Minimize total delay (critical path delay)
  - Reduce delay = improve performance
- Satisfy design constrains
  - Maximum fanins and fanouts, …
- Remove undesired circuit behavior
  - Hazard, race, …

3-3

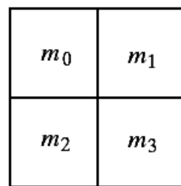
The Map Method

- The map method is also known as the **Karnaugh map** or **K-map**
- Provide a straightforward procedure for minimizing Boolean functions
- The simplified expressions are always in one of the two standard forms:
  - Sum of Products (SOP)
  - Product of Sums (POS)

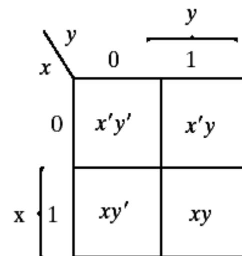
3-4

Two-Variable Map (1/2)

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

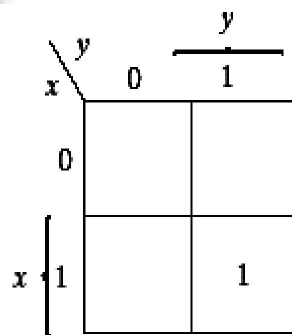


(a)

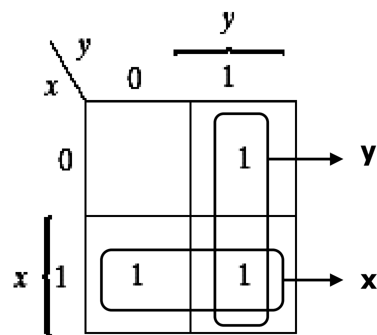


(b)

Two-Variable Map (2/2)



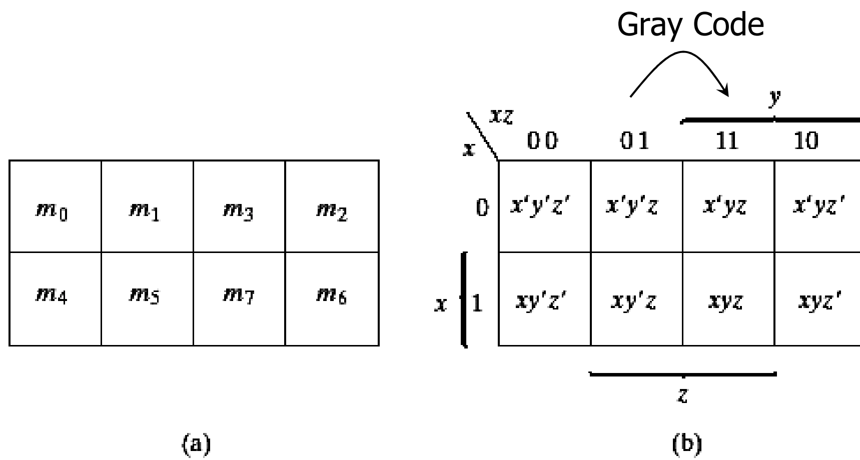
(a) xy



(b) x + y

$x + y = x(y+y') + y(x+x')$
$= xy\ (m_3) + xy'\ (m_2) + x'y\ (m_1)$
$= m_1 + m_2 + m_3$

Three-Variable Map (1/2)

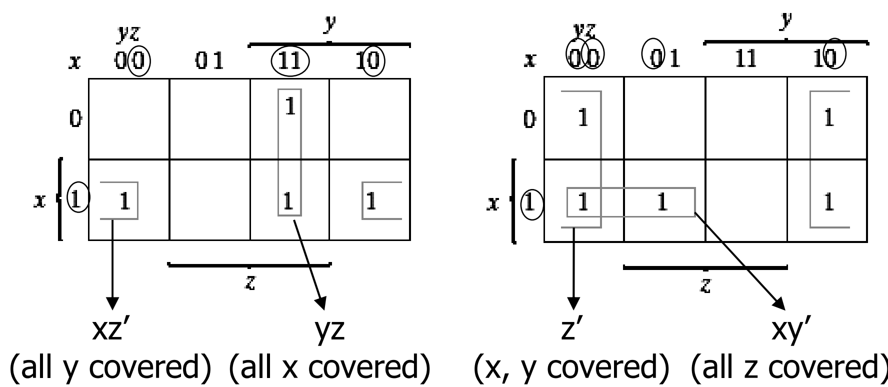


3-7

Three-Variable Map (2/2)

$F(x, y, z) = \sum(3,4,6,7) = yz + xz'$

$F(x, y, z) = \sum(0,2,4,5,6) = z' + xy'$

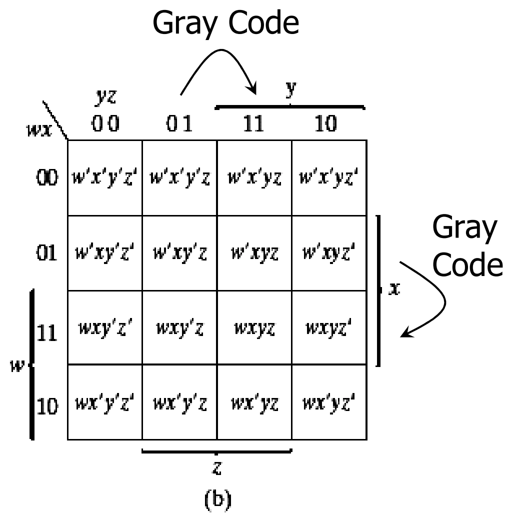


3-8

Four-Variable Map (1/2)

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

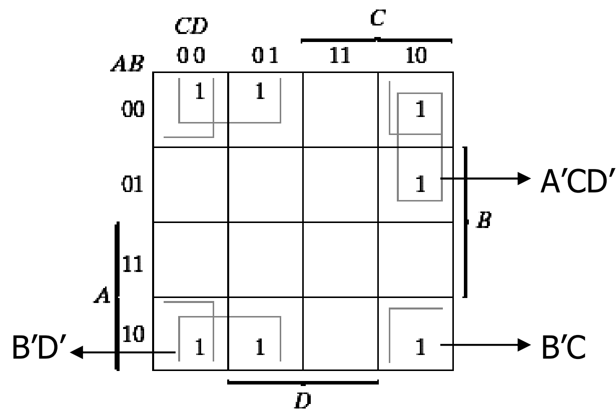
(a)



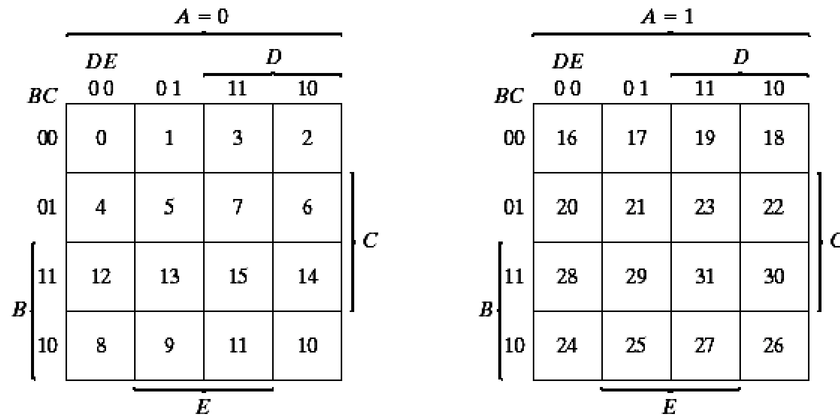
(b)

Four-Variable Map (2/2)

F = A′B′C′ + B′CD′ + A′BCD′ + AB′C′ = $\sum(0,1,2,6,8,9,10)$
= B′D′ + B′C′ + A′CD′



Five-Variable Map (1/2)



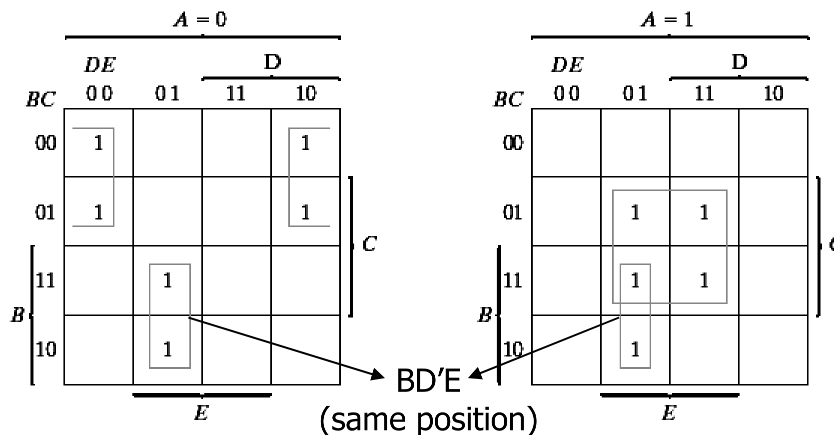
* Maps with six or more variables need too many squares and are impractical to use.

3-11

Five-Variable Map (2/2)

F(A,B,C,D,E) = (0,2,4,6,9,13,21,23,25,29,31)

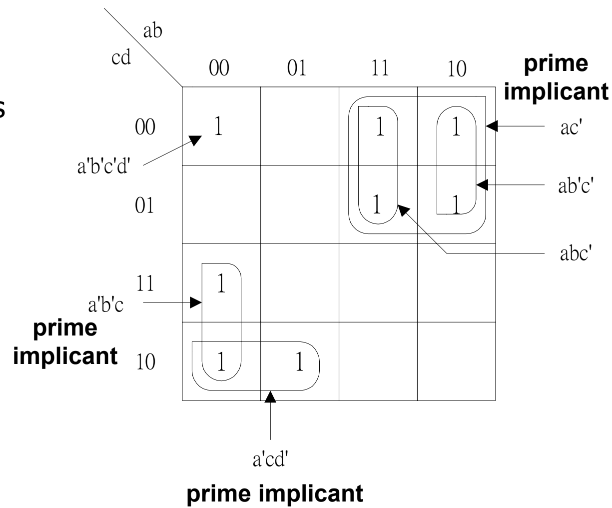
= A′B′E′ + BD′E + ACE



3-12

Prime Implicants

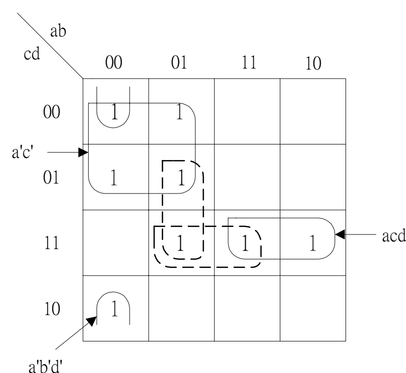
- Implicant (cube) :
  A group of minterms that form a cube
- Prime implicant :
  Combine maximum possible number of adjacent squares in the map



3-13

Essential Prime Implicants

- If a minterm is covered by only one prime implicant, that prime implicant is **essential** and must be included

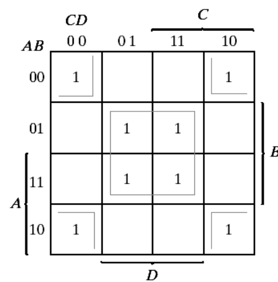


Note: 1's in red color are covered by only one prime implicant. All other 1's are covered by at least two prime implicants

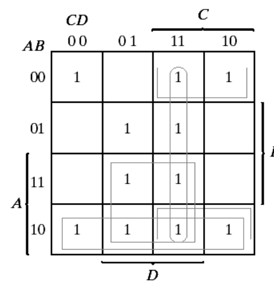
3-14

Systematic Simplification

- Identify all prime implicants on the k-map
- Select all essential prime implicants
- Select a minimum subset of the remaining prime implicants that cover all 1's
- Ex: $F(A,B,C,D)=\sum(0,2,3,5,7,8,9,10,11,13,15)$



(a) Essential prime implicants BD and B′D′

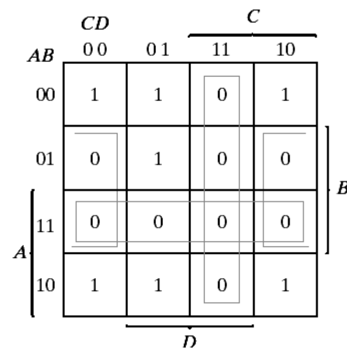


(b) Prime implicants CD, B′C AD, and AB′

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

Product of Sums Simplification

- The complement of a function is represented in the map by the squares not marked by 1's
  - Choose 1 → sum of products (minterms)
  - Choose 0 → product of sums (Maxterms)



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

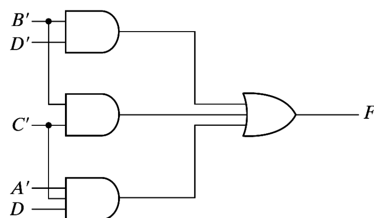
$=B'D'+B'C'+A'C'D$

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

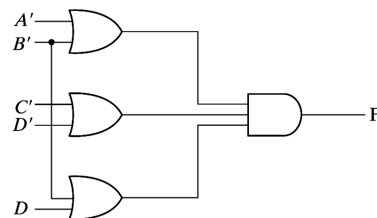
Two Gate Implementations

- Sometimes product-of-sums representations may have smaller implementations

7 literals, 4 gates	6 literals, 4 gates



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



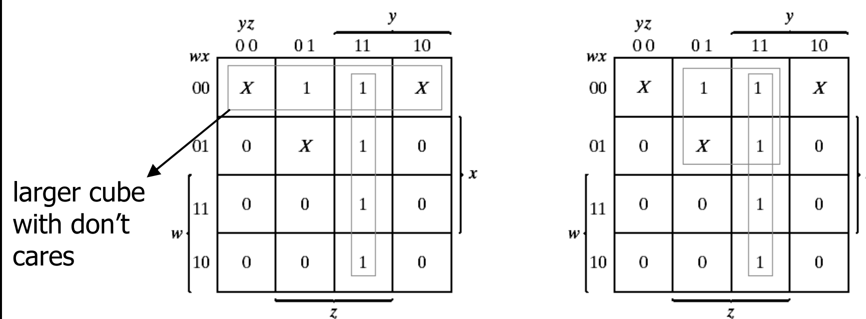
(b) $F = (A' + B')(C' + D')(B' + D)$

Fig. 3-15 Gate Implementation of the Function of Example 3-8

3-17

Don’t Care Conditions

- X = don't care (can be 0 or 1)
- Don’t cares can be included to form a larger cube, but not necessary to be completely covered
- Ex: $F(w,x,y,z) = \sum(1,3,7,11,15)$ $d(w,x,y,z) = \sum(0,2,5)$



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

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

3-18

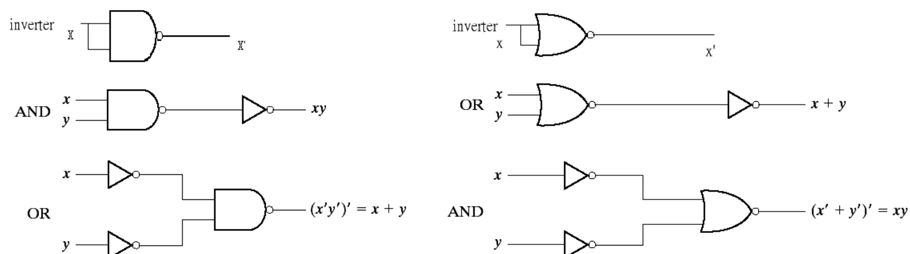
Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

3-19

NAND and NOR Implementation

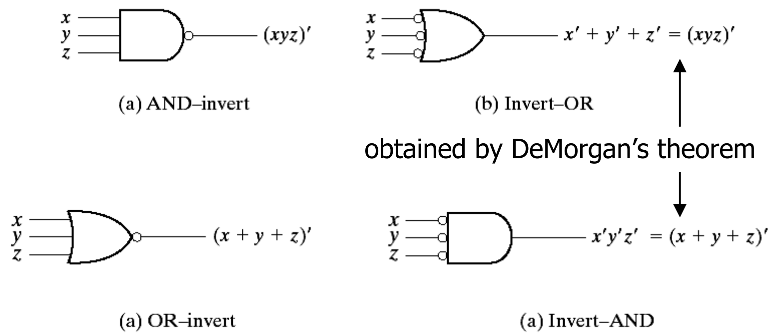
- Digital circuits are frequently constructed with **NAND** or **NOR** gates rather than with AND and OR gate
  - NAND and NOR gates are much easier to fabricate
- NAND or NOR gates are both universal gates
  - Any digital system can be implemented with only NAND gates or NOR gates



3-20

Alternative Graphic Symbols

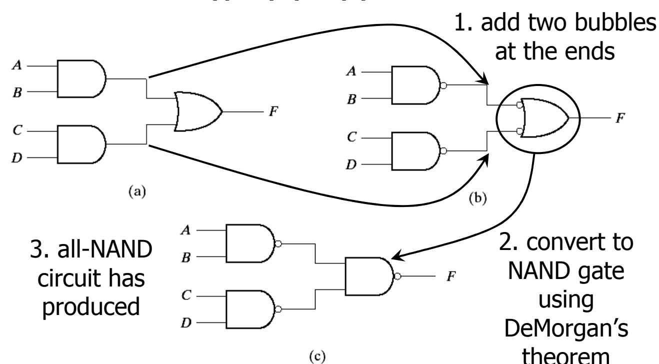
- To facilitate the conversion to NAND or NOR logic, it is convenient to define alternative graphic symbols
  - "Bubble" means complement



3-21

Two-Level Implementation (NAND)

- It's easy to implement a Boolean function with only NAND gates if converted from a **sum of products** form
- Ex: F = AB+CD = ((AB)'(CD)')'



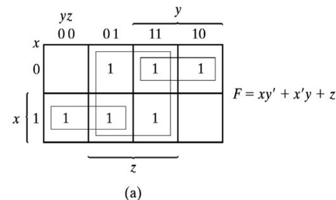
3-22

Example 3-10

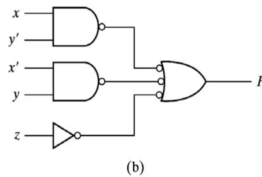
$$F(x, y, z) = \sum(1,2,3,4,5,7)$$

Procedures:

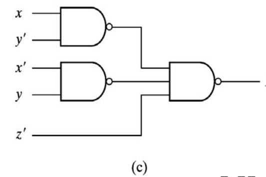
1. Simplify the function in sum of products
2. Draw NAND gates for the first level
3. Draw a single AND-invert or invert-OR in the second level
4. Add an inverter at the first level for the term with a single literal



(a)



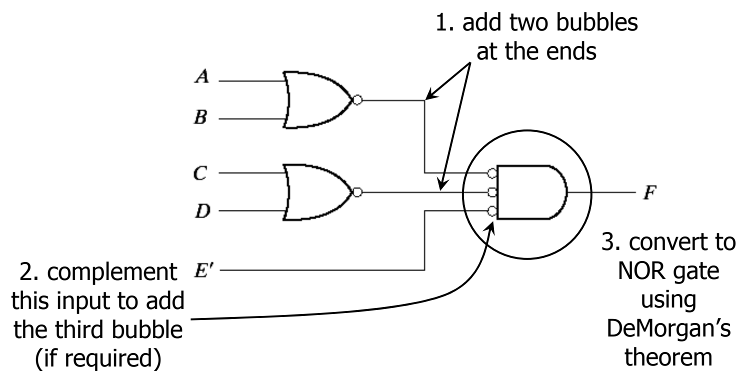
(b)



(c)

Two-Level Implementation (NOR)

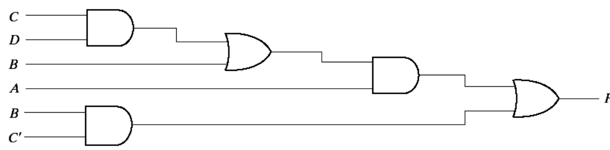
- It's easy to implement a Boolean function with only NOR gates if converted from a **product of sums** form
- Ex: F=(A+B)(C+D)E



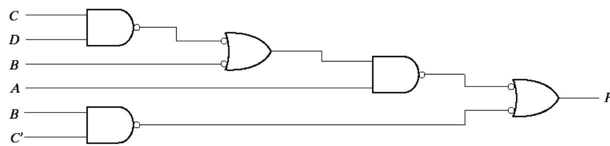
Multilevel NAND Circuits

F=A(CD+B)+BC′

- Procedures:
  1. Convert all AND gates to NAND gates with AND-invert symbols
  2. Convert all OR gates to NAND gates with invert-OR symbols
  3. Check all bubbles and insert an inverter for the bubble that are not compensated by another bubble



(a) AND-OR gates



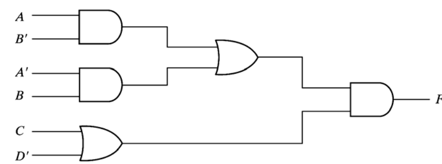
(a) NAND gates

3-25

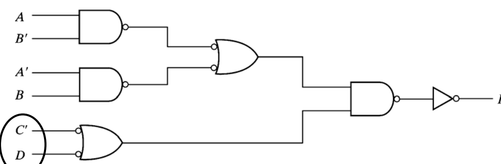
Multilevel NOR Circuits

- For NOR gates, AND → invert-AND, OR → OR-invert
- Other procedures are the same as those for NAND

F=(AB′+A′B)(C+D′)



(a) AND-OR gates



(b) NAND gates

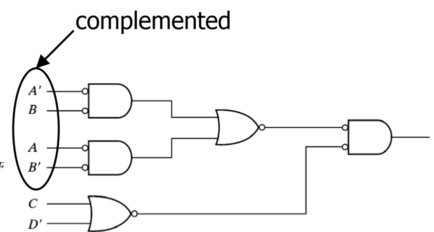


Fig. 3-27 Implementing $F = (AB' + A'B)(C + D')$ with NOR Gates

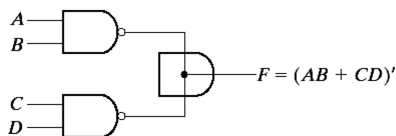
3-26

Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

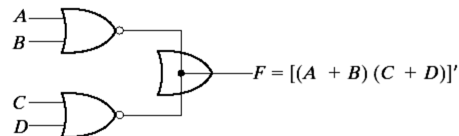
Wired Logic

- Wired logic: direct wire connection that results in a specific logic function
  - Wired-AND
  - Wired-OR
- Some NAND and NOR implementations (not all) have such a property



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

(AND-OR-INVERT)

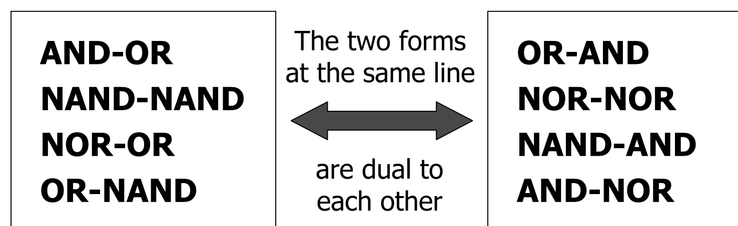


(b) Wired-OR in ECL gates

(OR-AND-INVERT)

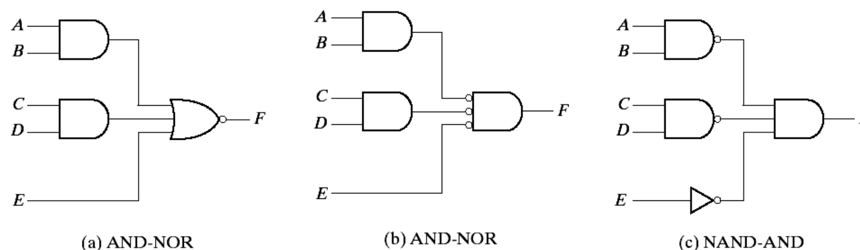
Nondegenerate Forms

- There are 16 possible combinations of two-level forms
  - Eight of these combinations will degenerate to a single operation
- The other eight **nondegenerate** forms produce an implementation in SOP or POS



AND-OR-INVERT Implementation

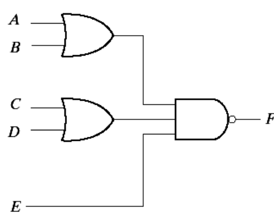
- NAND-AND and AND-NOR are equivalent and both perform the AND-OR-INVERT (**AOI**) function
- Require sum-of-products form in nature
  - When starting from product-of-sums form, complement it using DeMorgan's theorem to obtain sum-of-products form
- Ex: F = (AB + CD + E)'



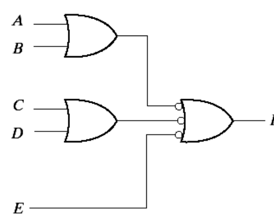
(a) AND-NOR (b) AND-NOR (c) NAND-AND

OR-AND-INVERT Implementation

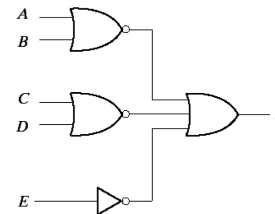
- OR-NAND and NOR-OR are equivalent and both perform the OR-AND-INVERT (**OAI**) function
- Require product-of-sums form in nature
  - When starting from sum-of-products form, complement it using DeMorgan's theorem to obtain product-of-sums form
- Ex: F = [(A+B)(C+D)E]'



(a) OR-NAND



(b) OR-NAND



(c) NOR-OR

3-31

Implement with Two-Level Forms

Table 3-3
Implementation with Other Two-Level Forms

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

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

3-32

Example 3-11

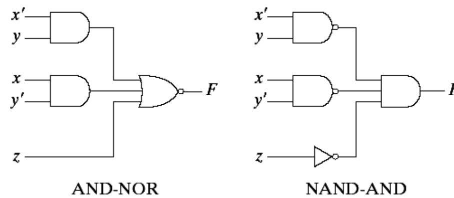
- F = x'y'z' + xyz'
  (choose 1 in K-map)
  = (x'y + xy' + z)'
  (choose 0 in K-map)

x \ yz	00	01	11	10
0	1	0	0	0
1	0	0	0	1

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

(a) Map simplification in sum of products.

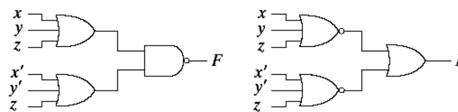
choose 0



AND-NOR NAND-AND

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

choose 1



OR-NAND NOR-OR

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

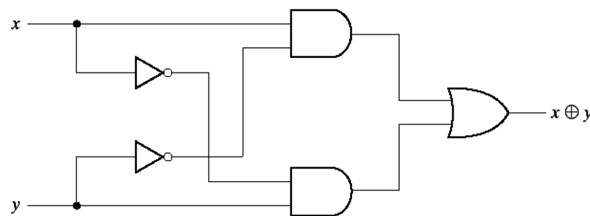
Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

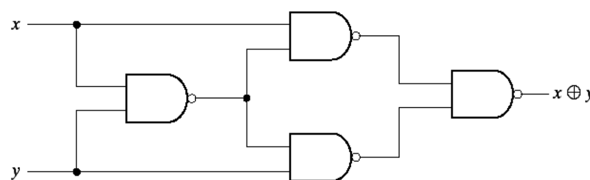
Exclusive-OR (XOR) Function

- XOR is often denoted by the symbol $\oplus$
- Logic operation of XOR
  - $X \oplus Y = XY' + X'Y$
  - Equal to 1 if **only x** is equal to 1 or if **only y** is equal to 1, but not when both are equal to 1
- It's complement, exclusive-NOR (XNOR), is often denoted by the symbol $\odot$
- Logic operation
  - $X \odot Y = XY + X'Y'$
  - It is equal to 1 if **both x and y** are equal to 1 or if both are equal to 0
- Seldom used in general Boolean functions
  - Particularly useful in arithmetic operations and error detection and correction circuits

Exclusive-OR Implementations



(a) With AND-OR-NOT gates

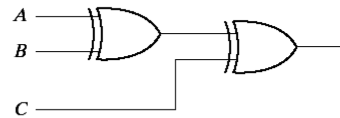


(b) With NAND gates

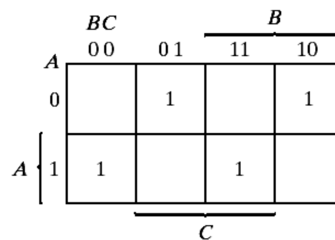
Odd Function

- The multiple-variable XOR operation is defined as an **odd function**
  - TRUE when no. of "1" in inputs is odd

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) 3-input odd function

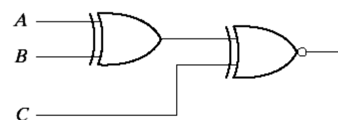


(a) Odd function
$F = A \oplus B \oplus C$

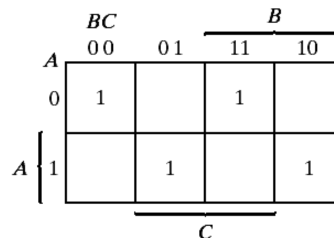
Even Function

- The multiple-variable XNOR operation is defined as an **even function**
  - TRUE when no. of "1" in inputs is even

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



(b) 3-input even function



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

Four-Variable XOR Function

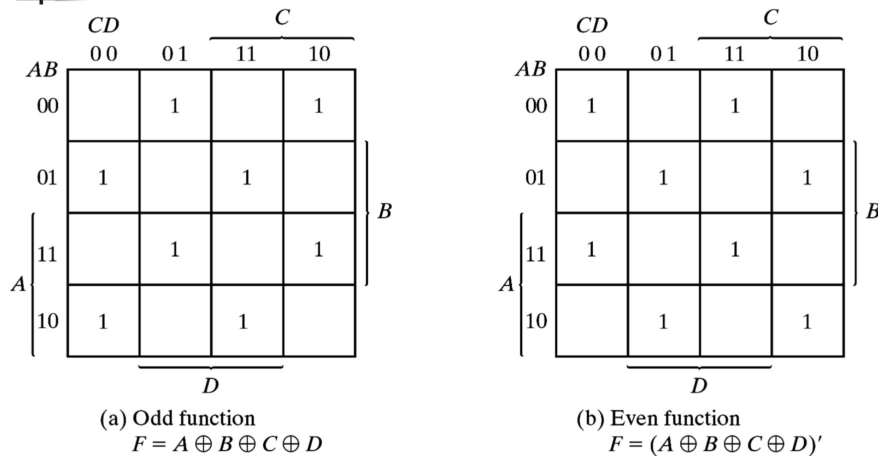


Fig. 3-35 Map for a Four-variable Exclusive-OR Function

3-39

Parity Generation and Checking

- An extra **parity** bit is often added and checked at the receiving end for error
- The circuit that generates the parity bit in the transmitter is called a **parity generator**
- The circuit that checks the parity in the receiver is called a **parity checker**
- Exclusive-OR functions are very useful to construct such circuits

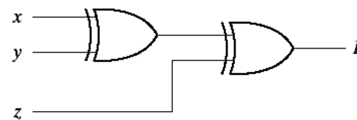
3-40

Parity Generator

- For even parity:
  - The total number of “1” (including P) is even
  - The number of “1” at inputs is odd
  - Generated with an **XOR** gate (odd function)
  - $P = x \oplus y \oplus z$ (for 3-bit message)
- Similarly, odd parity can be generated with an XNOR gate

Table 3-4
Even-Parity-Generator Truth Table

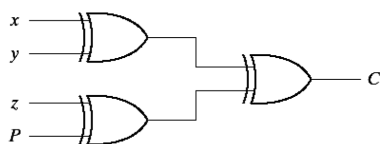
Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) 3-bit even parity generator

Parity Checker

- For even parity, the total number of “1” in the message is even
  - An error occurs when the received number of “1” is odd
- An XOR gate (odd function) can detect such an error
  - Has n+1 inputs



(a) 4-bit even parity checker

Table 3-5
Even-Parity-Checker Truth Table

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

Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

3-43

Hardware Description Language

- Have high-level language constructs to describe the functionality and connectivity of the circuit
- Can describe a design at some levels of abstraction
  - Behavioral, RTL, Gate-level, Switch
- Can describe functionality as well as timing
- Can model the concurrent actions in real hardware
- Can be used to document the complete system design tasks
  - testing, simulation … related activities
- Comprehensive and easy to learn
- Two popular languages: Verilog & VHDL
  - Will be taught in another course

3-44

Why Use an HDL ?

- Hard to design directly for complex systems
- Formal description using HDL
  - Verify the specification through simulation or verification
  - Easy to change
  - Enable automatic synthesis
- Allow architectural tradeoffs with short turnaround
- Reduce time for design capture
- Encourage focus on functionality
- Shorten the design verification loop

*HDL = Hardware Description Language