Chapter 3
Gate-Level Minimization

Outline

- The Map Method
- Four-Variable Map
- Five-Variable Map
- Product-of-Sum Simplification
- Don’t Care Conditions
- NAND and NOR Implementation
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

The Map Method

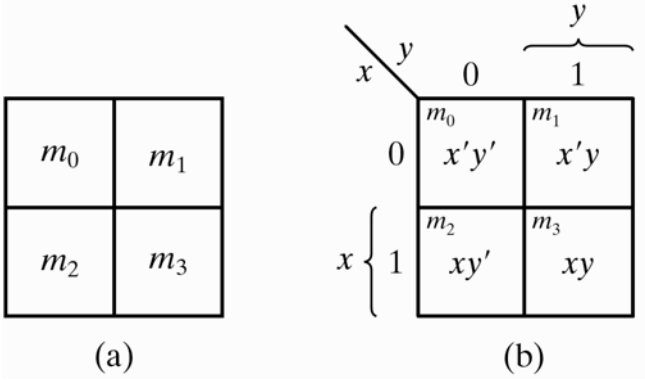
- *Gate-level minimization* refers to the design task of finding an optimal gate-level implementation of Boolean functions describing a digital circuit.
- The complexity of the digital logic gates has close relation with the complexity of the algebraic expression.
- The map method provides a straightforward logic minimization.
- Logic minimization
  - Algebraic approaches: lack specific rules
  - the Karnaugh map (K-map)
    - a simple straight forward procedure
    - a pictorial form of a truth table
    - applicable if the # of variables < 7

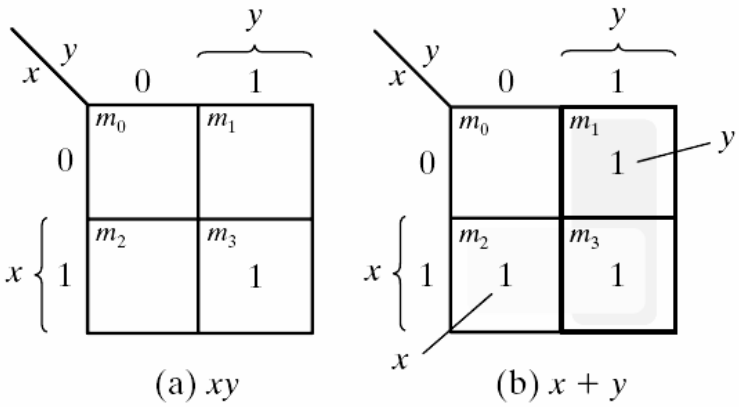
The Map Method

- A diagram made up of squares
  - each square represents one minterm
- Boolean function
  - sum of minterms
  - sum of products and product of sums are two basic standard algebraic expressions
  - The simplified algebraic expression is one with a minimum number of terms and a minimum number of literals
  - The simplified expression may not be unique

Two-Variable Map

- A two-variable map
  - Four minterms
  - x' = row 0; x = row 1
  - y' = column 0; y = column 1
  - a truth table in square diagram
  - $xy = \Sigma(m_3)$
  - $x + y = \Sigma(m_1, m_2, m_3)$

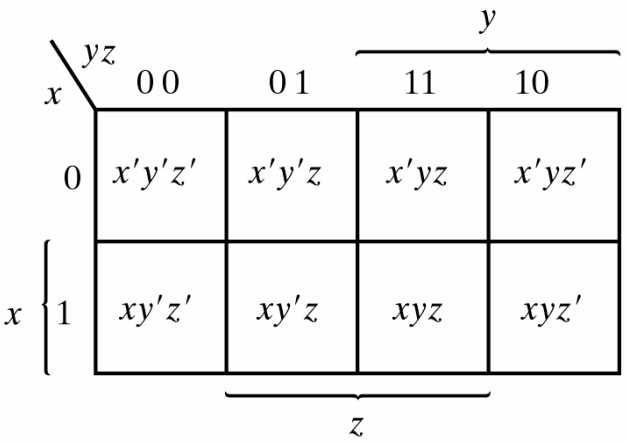




Three-Variable Map

- A three-variable map
  - Eight minterms
  - The Gray code sequence
  - Any two adjacent squares in the map differ by only one variable
    - primed in one square and unprimed in the other
  - Example:
    - m_5 and m_7 can be simplified
    - $m_5 + m_7 = xy'z + xyz$
      $= xz\,(y' + y) = xz$

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6



Three-Variable Map

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

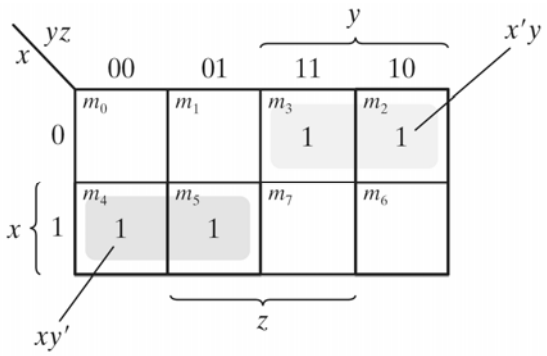
m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

x \ yz	00	01	11	10
0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
1	$xy'z'$	$xy'z$	xyz	xyz'

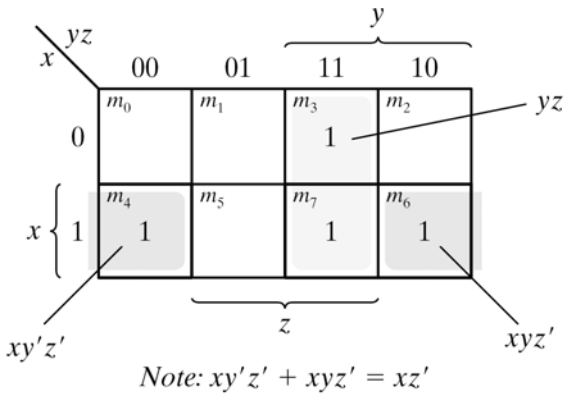
(y spans 11 and 10; z spans 01 and 11)

Examples

- Example 3-1
  - $F(x, y, z) = \Sigma(2, 3, 4, 5)$
  - $F = x'y + xy'$



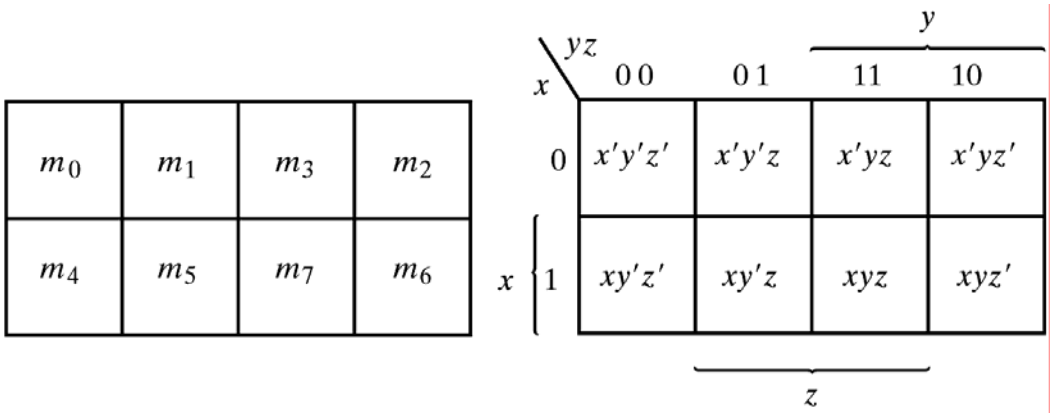
- Example 3-2
  - $F(x, y, z) = \Sigma(3, 4, 6, 7)$
    $= yz + xz'$



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

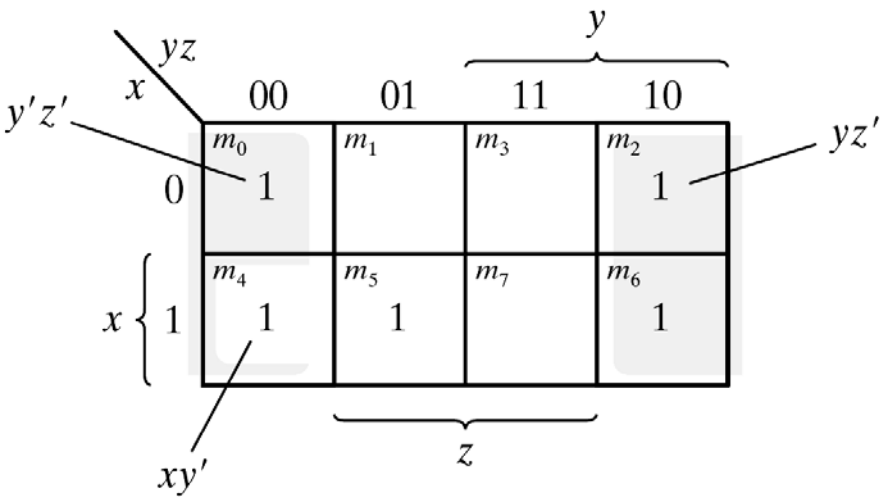
Four Adjacent Squares

- Any combination of 4 adjacent squares reduces to an expression with only one literal.
- $m_0 + m_2 + m_4 + m_6 = x'y'z' + x'yz' + xy'z' + xyz'$
  $= x'z'(y' + y) + xz'(y' + y)$
  $= x'z' + xz' = z'$
- $m_1 + m_3 + m_5 + m_7 = x'y'z + x'yz + xy'z + xyz$
  $= x'z(y' + y) + xz(y' + y) = x'z + xz = z$



Example 3-3

- $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = m_0 + m_2 + m_4 + m_5 + m_6$
- $F = z' + xy'$



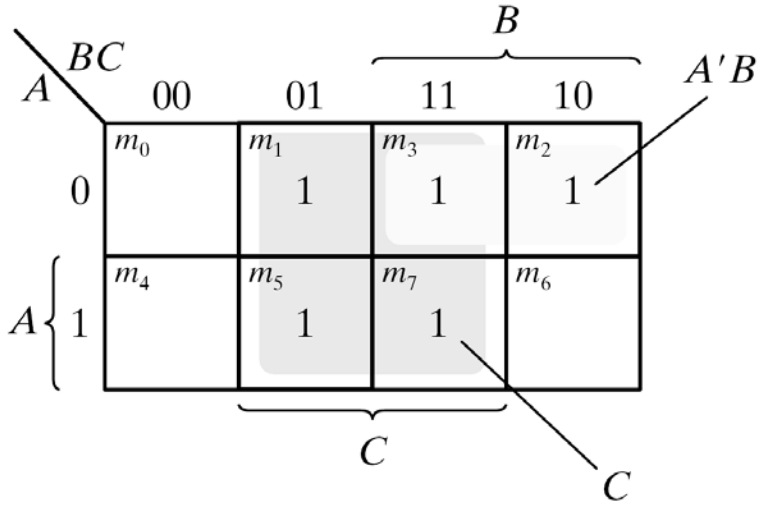
Note: $y'z' + yz' = z'$

Example 3-4

– $F = A'C + A'B + AB'C + BC$

a) express it in sum of minterms

b) find the minimal sum of products expression from maximam to minimum squares



Four-Variable Map

- Construction of the map:
  - consists of 16 minterms
  - checks the combinations of 2, 4, 8, and 16 adjacent squares.
  - The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns.

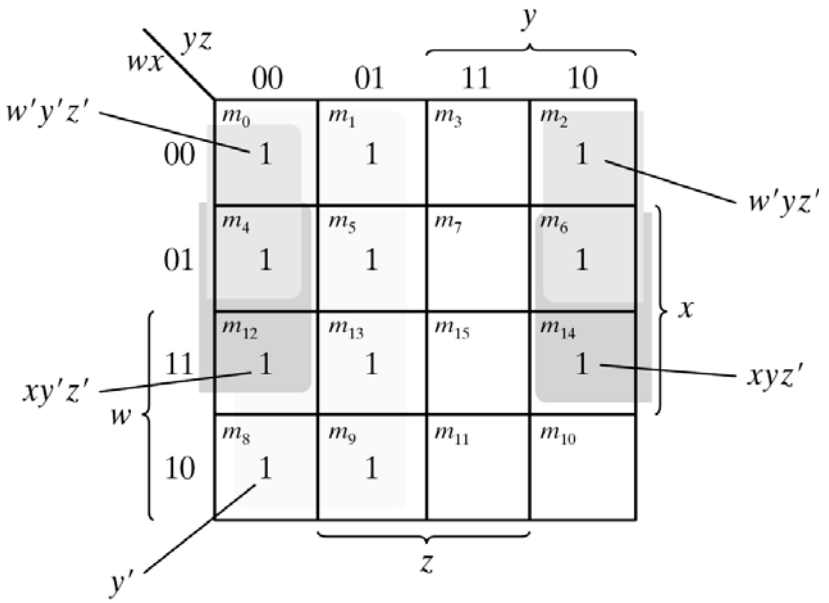
m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

wx \ yz	00	01	11	10
00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

y (11, 10); x; w (11, 10); z

Example 3-5

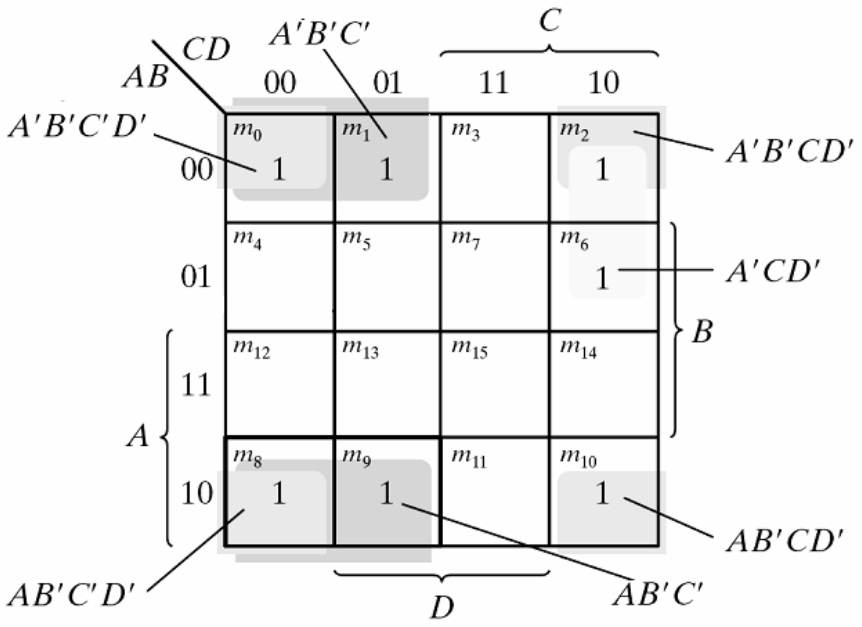
- $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$
- $F = y' + w'z' + xz'$



Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$

Example 3-6

- $F = A'B'C' + B'CD' + A'BCD' + AB'C'$
- Simplified: $F = B'D' + B'C' + A'CD'$

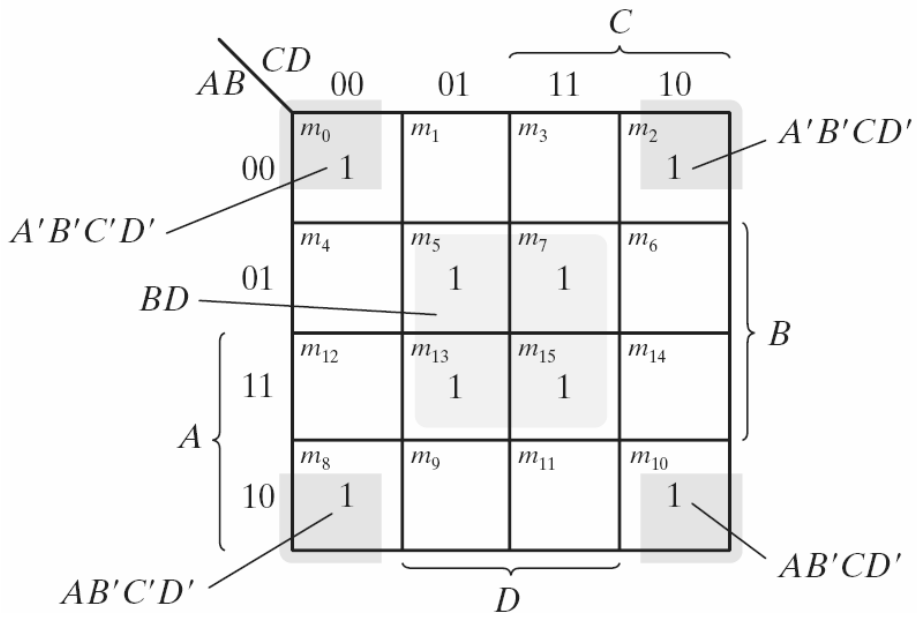


Prime Implicants

- We have to ensure:
  - All the minterms are covered
  - Minimize the number of terms
  - No redundant terms (i.e., minterms already covered by other terms)
- A prime implicant: a product term obtained by combining the maximum possible number of adjacent squares (combining all possible maximum numbers of squares)
- Essential prime implicant: a minterm is covered by only one prime implicant
- The simplified expression is obtained from the logical sum of:
  - All the essential prime implicants and
  - Other prime implicants that need to cover other remaining minterms that are not covered by essential prime implicants.

Essential Prime Implicants

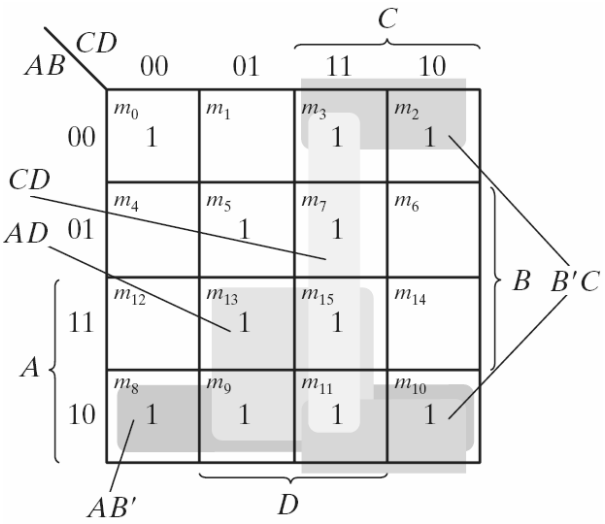
- Example: $F(A, B, C, D) = \Sigma(0, 2, 5, 7, 8, 10, 13, 15)$



Prime Implicants

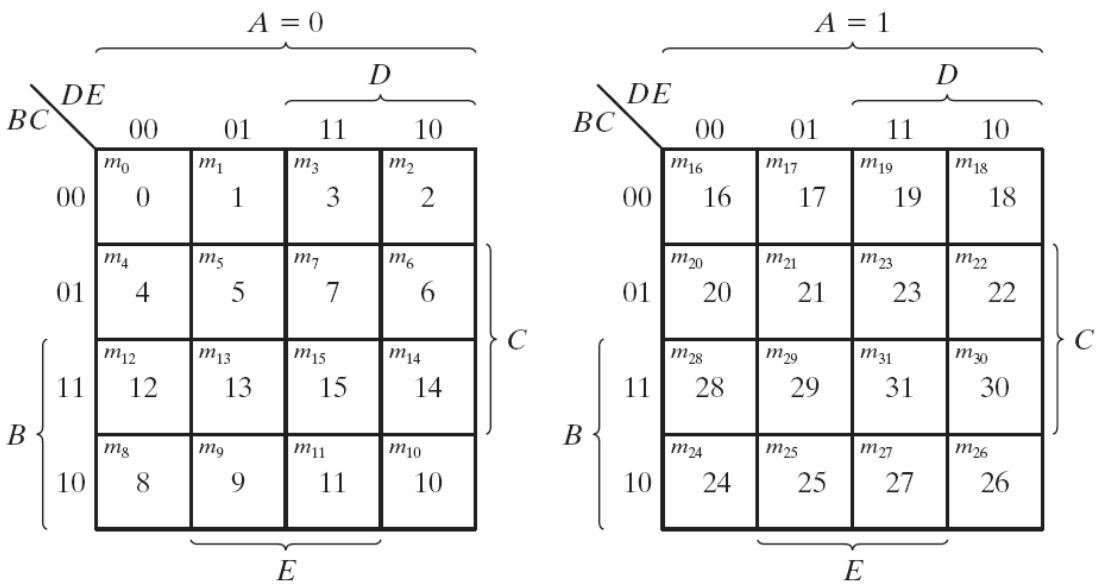
- Example: $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$
  - the simplified expression may not be unique
  - $F = BD + B'D' + CD + AD = BD + B'D' + CD + AB'$

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



Five-Variable Map

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



Example 3-7

- $F = \Sigma(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$
- $F = A'B'E' + BD'E + ACE$

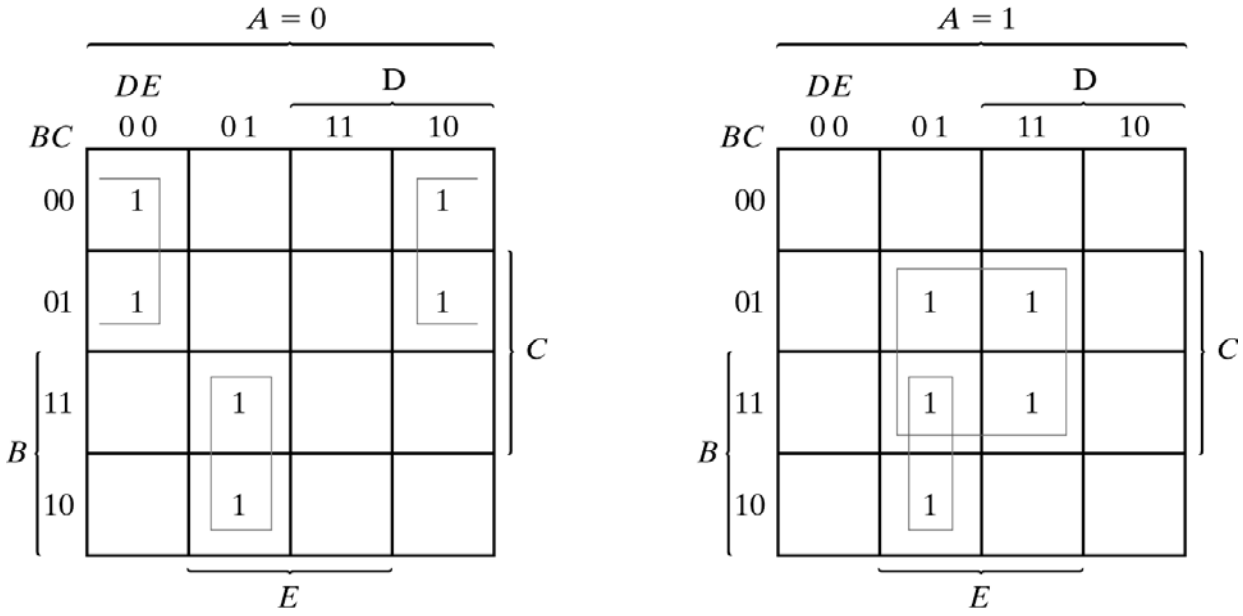
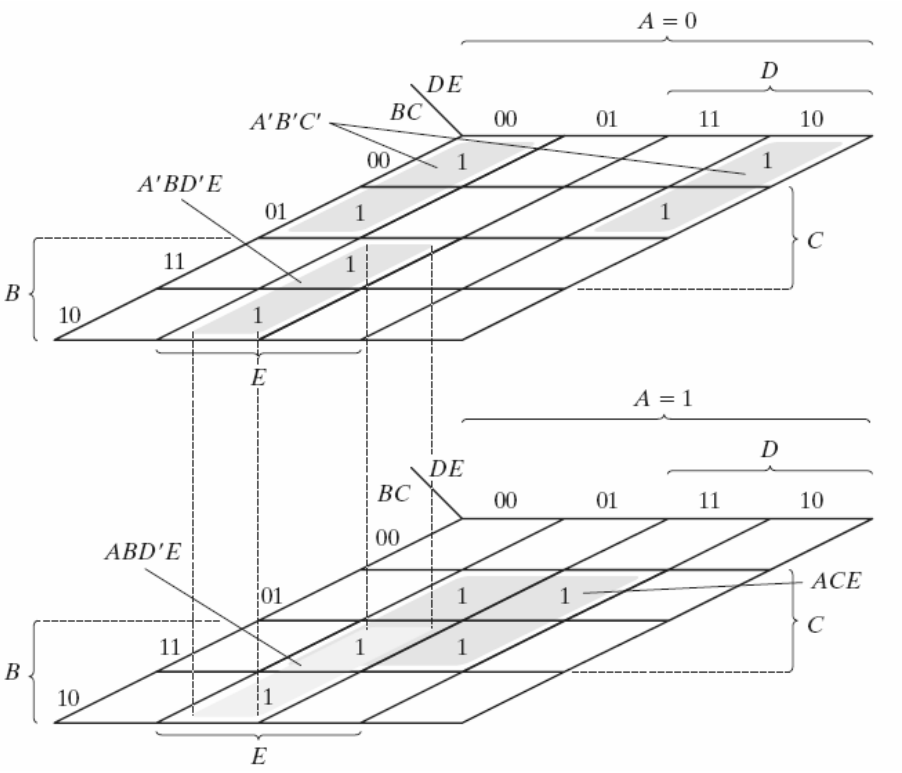


Fig. 3-13 Map for Example 3-7; $F = A'B'E' + BD'E + ACE$

Another Map for Example 3-7



The K-Map Conclusion

- The relationship between the number of adjacent squares and the number of literals in the term:

	Number of Adjacent Squares	Number of Literals in a Term in an n-variable map			
K	2^k	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0	1	2	3	4	5
1	2	1	2	3	4
2	4	0	1	2	3
3	8		0	1	2
4	16			0	1
5	32				0

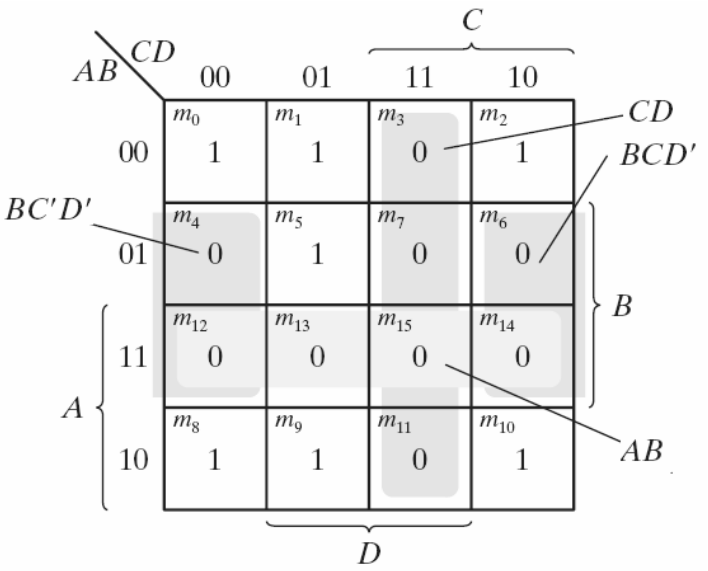
Product of Sums Simplification

- Approach #1: The product of sums simplification is based on the generalized DeMorgan's theorem.
  - (0's in the K-map): Simplified F' in the form of sum of products.
  - (1's in the K-map): Apply DeMorgan's theorem $F = (F')'$
  - F': sum of products $\Rightarrow$ F: product of sums
- Approach #2: duality
  - combinations of maxterms (it was minterms)
  - $M_0M_1 = (A + B + C + D)(A + B + C + D')$
    $= (A + B + C) + (DD')$
    $= A + B + C$

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

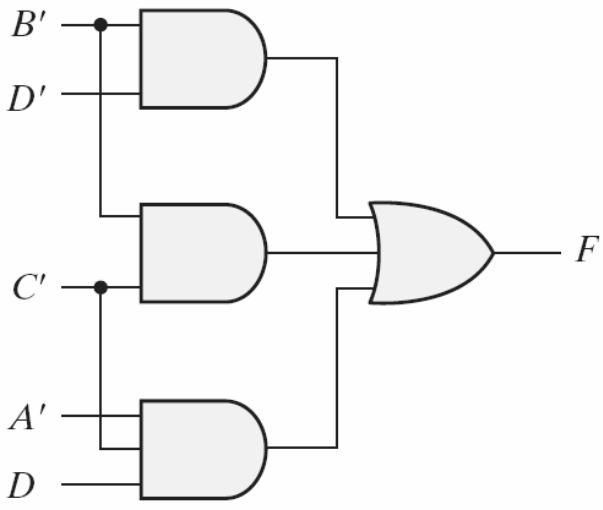
Example 3-8

- Simplified the function: $F = \Sigma(0, 1, 2, 5, 8, 9, 10)$
- a) Sum of products:
  $F = B'D'+B'C'+A'C'D$

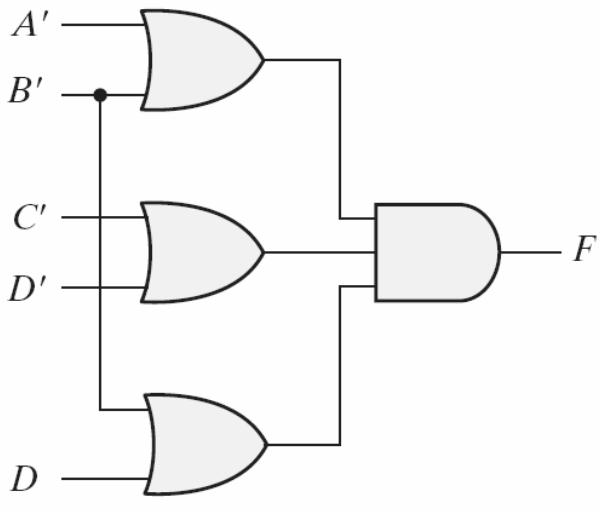


- b) Product of sums
- $F' = AB+CD+BD'$
- Apply DeMorgan's theorem;
  $F = (A'+B')(C'+D')(B'+D)$
- or think in terms of maxterms

Gate Implementation of the Function of Example 3-8



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



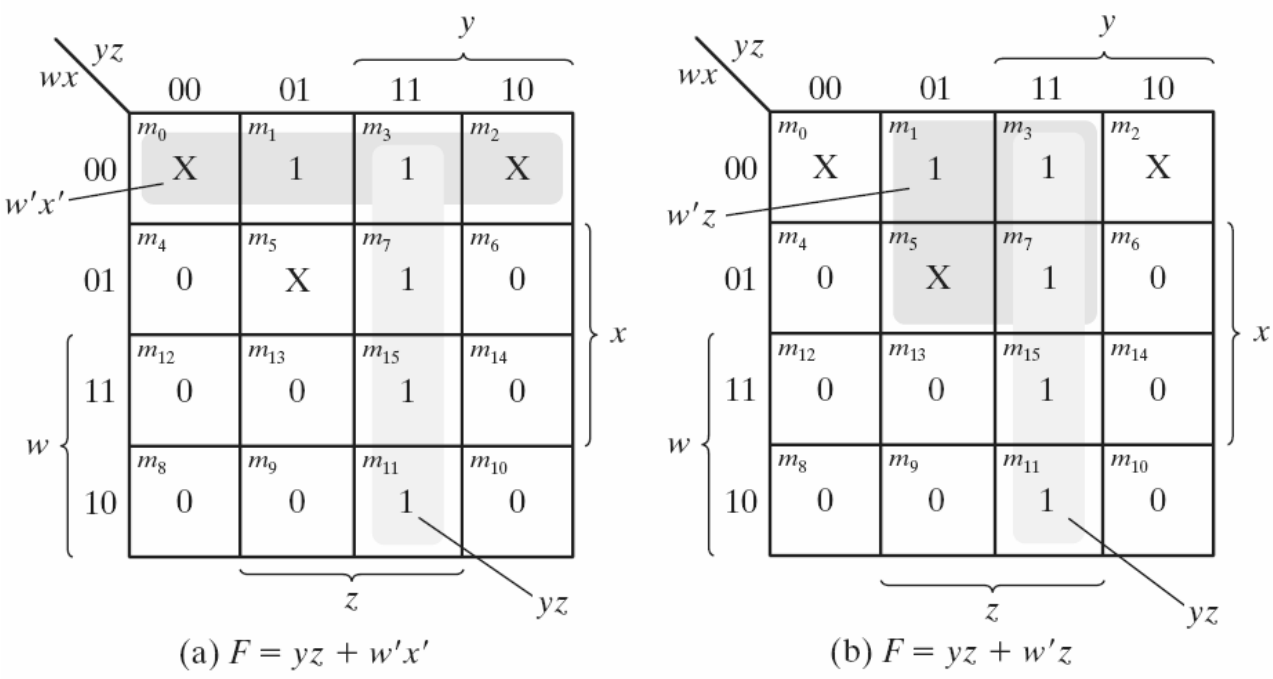
(b) $F = (A' + B')(C' + D')(B' + D)$

Two-Level Logic Implementation!!

Don't-Care Conditions

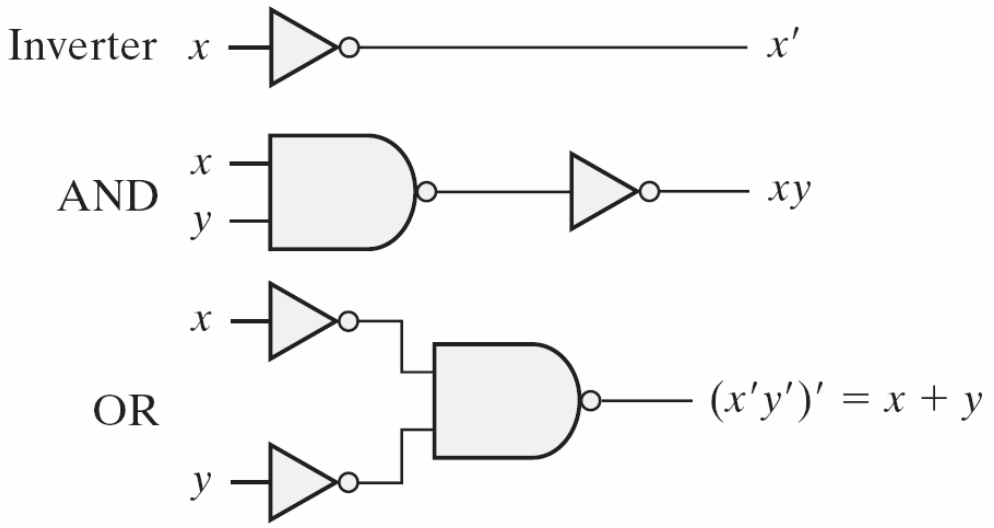
- The value of a function is not specified for certain combinations of variables
  - BCD: 1010 ~ 1111 are don't care terms
- The don't care conditions can be utilized in logic minimization
  - Because either 0 or 1 can be implemented; the logic optimization is more flexible for simplified gates.
- Example 3-9
  - Boolean function: $F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$
  - Don't care condition: $d(w, x, y, z) = \Sigma(0, 2, 5)$
  - $F = yz + w'x'$; $F = yz + w'z$
  - $F = \Sigma(0,1,2,3,7,11,15)$; $F = \Sigma(1,3,5,7,11,15)$
  - either expression is acceptable

Example of Don't-Care Conditions



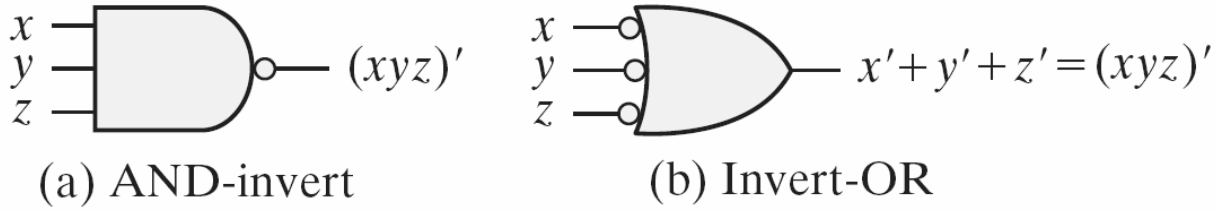
NAND and NOR Implementation

- NAND and NOR gates are easier to fabricate with electronic components than AND and OR gates.
- NAND gate is a universal gate because any operation can be implemented by it.



Equivalent NAND Gates

- The AND-invert and Invert-OR are equivalent, following the DeMorgan's theorem.
- Two graphic symbols are for a NAND gate



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

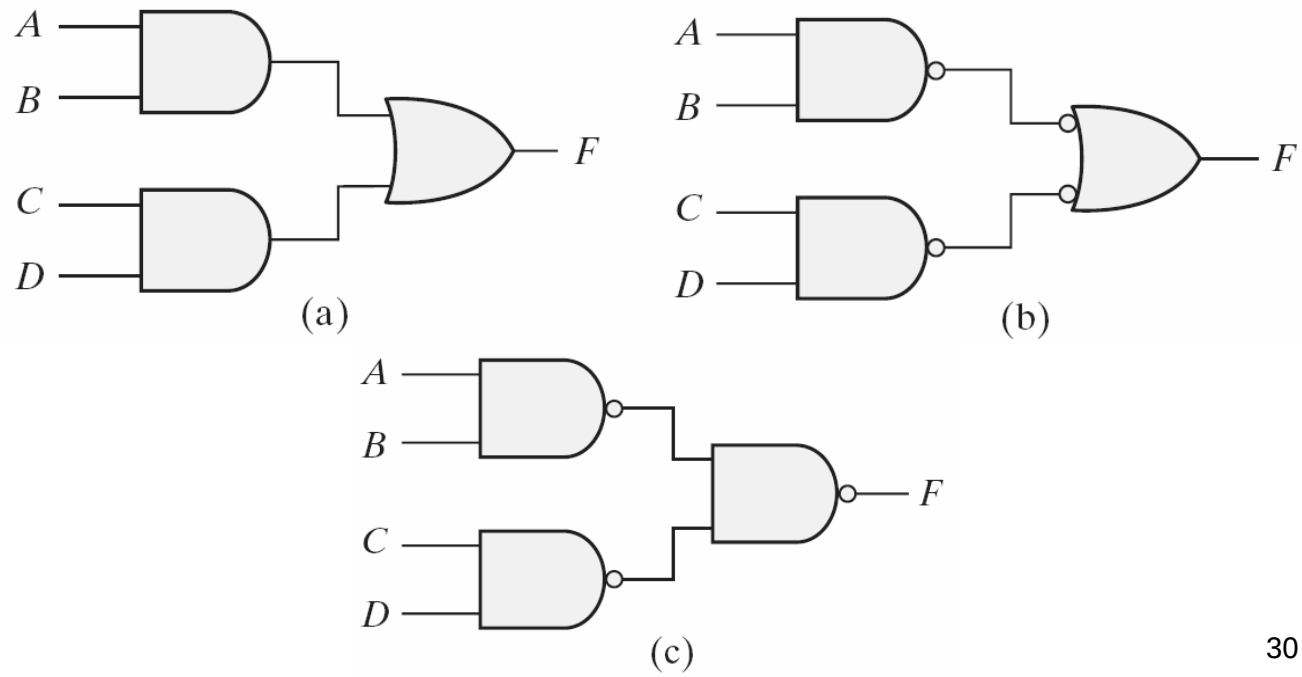
- The conversion between AND-Invert and Invert-OR makes the NAND implementation.

NAND-NAND Implementation

- Two-level NAND implementation procedure
  - 1. Simplify the function in the form of sum of products.
  - 2. Draw a NAND gate for each product term. The inputs to each NAND gate are the literals of the term forming a group of 1st level gate.
  - 3. Draw gates using AND-Invert or the Invert-OR in the second level. (Note: keep function by DeMorgan's theorem)
  - 4. Single literal must be complemented for first or second level.

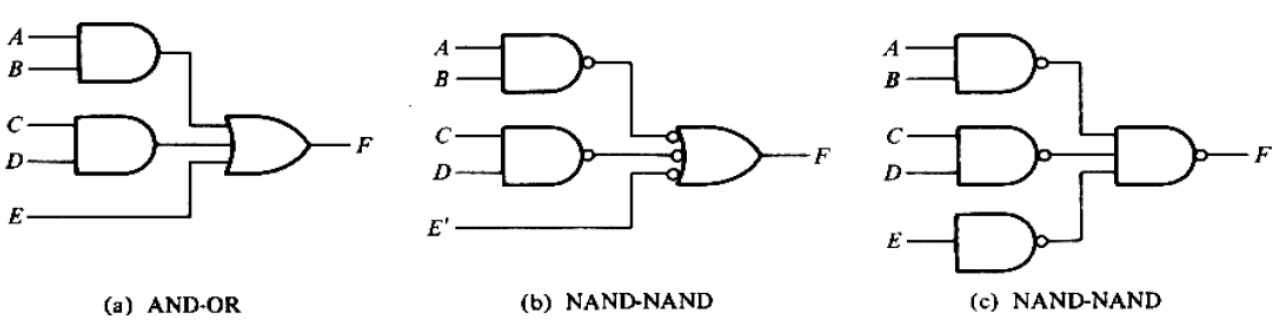
Example

- $F = AB + CD$
  - Three ways to implement F



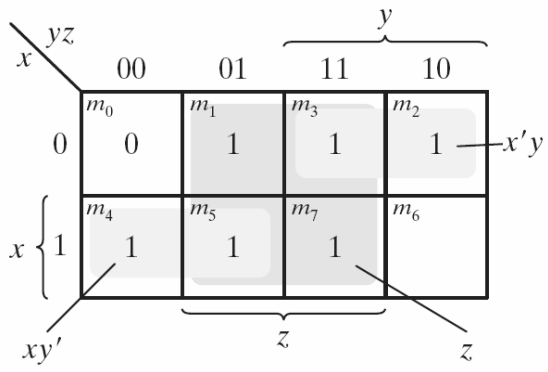
Two-Level NAND Implementation

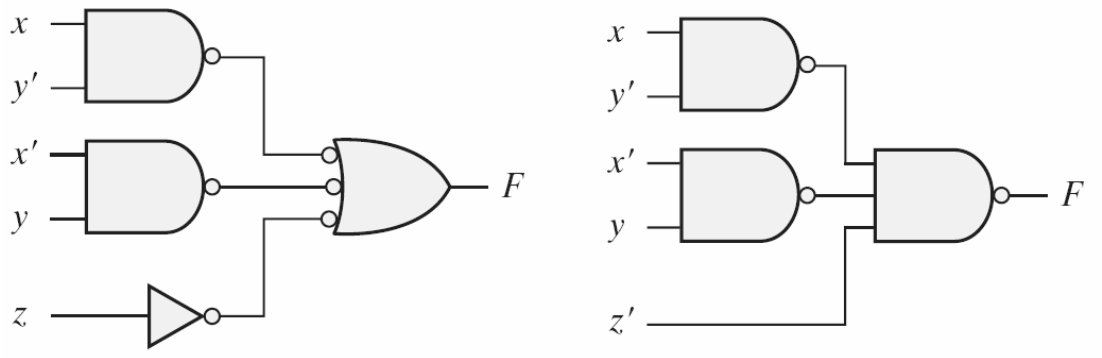
- Two-level logic
- NAND-NAND = sum of products
- Example: $F = AB + CD + E$
- $F = ((AB)'\,(CD)'\,E')' = AB + CD + E$



Example 3-10

- $F(x, y, z) = \Sigma(1, 2, 3, 4, 5, 7) = xy' + x'y + z$



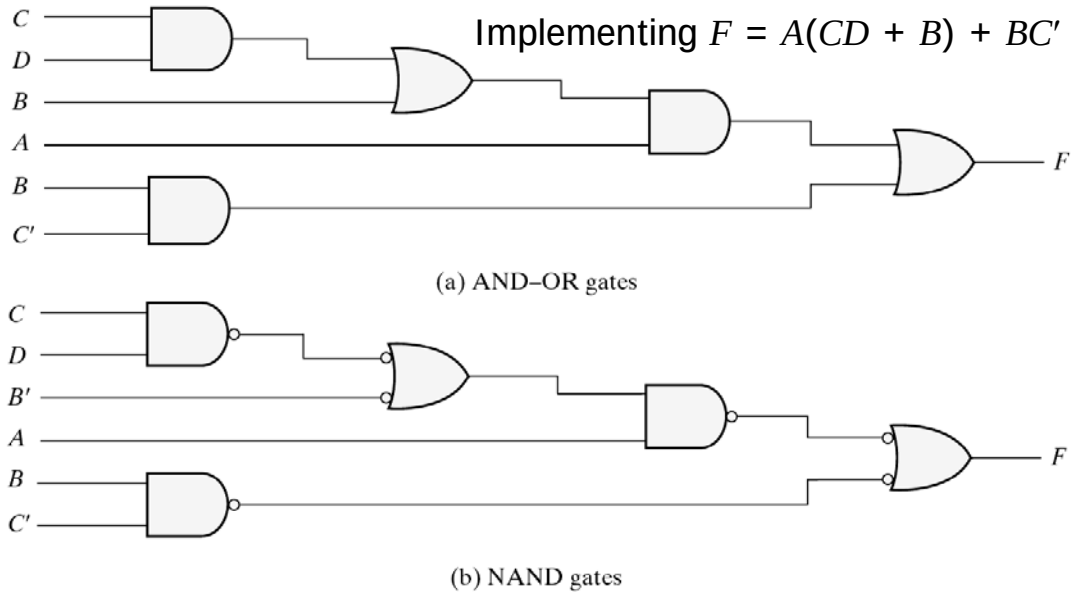


Multilevel NAND Circuits

- Multilevel NAND circuit implementation procedure
  - 1. Convert all AND gates to NAND gates with AND-Invert graphic symbols
  - 2. Convert all OR gates to NAND gates with Invert-OR graphic symbols
  - 3. Check all the bubbles (Inverter) in the diagram and insert possible inverter to keep the original function.

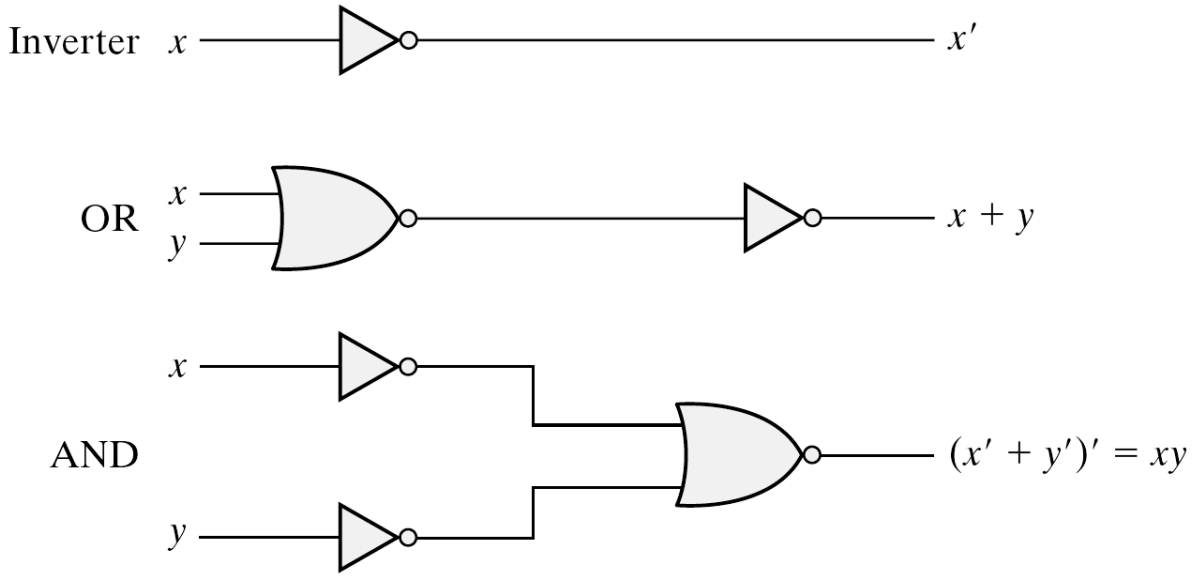
Multilevel NAND Circuits

- Boolean function implementation
  - AND-OR logic ⇒ NAND-NAND logic
    - AND ⇒ NAND + inverter
    - OR: inverter + OR = NAND

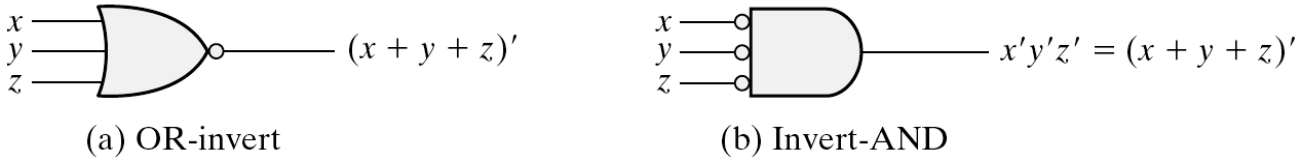


NOR Implementation

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

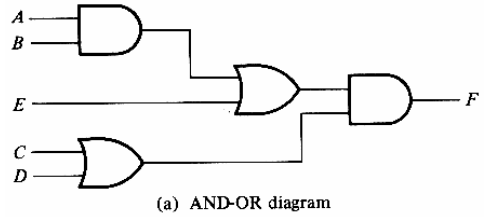


Two graphic symbols for a NOR gate

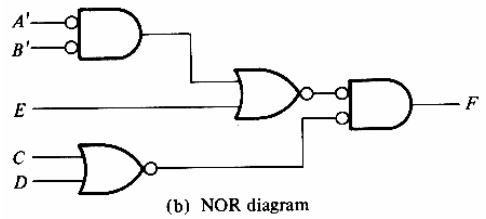


(a) OR-invert (b) Invert-AND

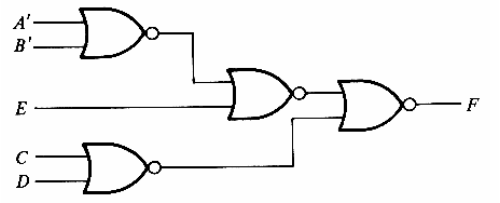
- Boolean-function implementation
  - OR ⇒ NOR + INV
  - AND: INV + AND = NOR



(a) AND-OR diagram



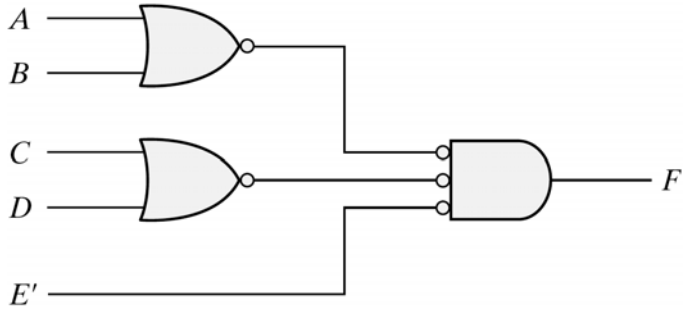
(b) NOR diagram



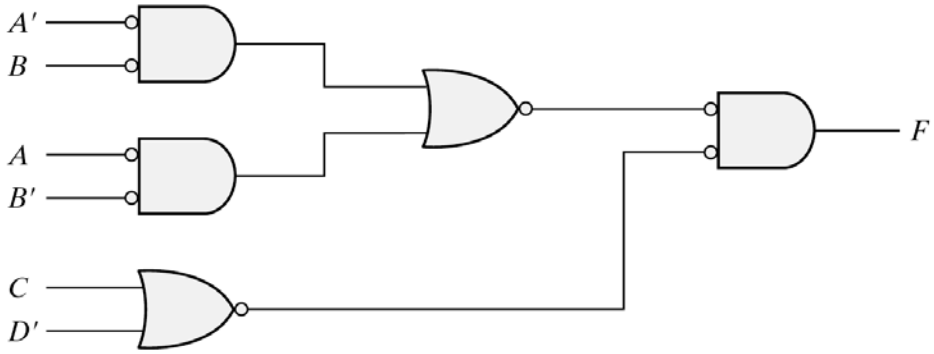
(c) Alternate NOR diagram

Example: Implementation with NOR Gate

$F = (A + B)(C + D)E$



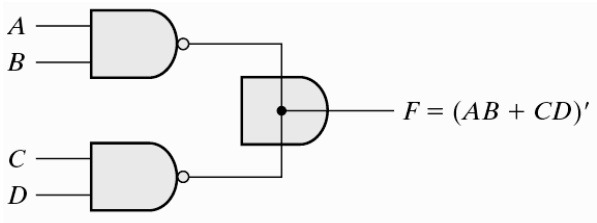
$F = (AB' + A'B)(C + D')$



Other Two-level Implementations

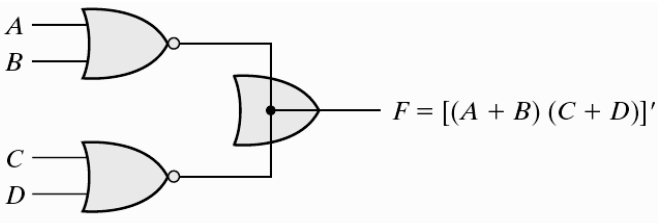
- Wired logic
  - a wire connection between the outputs of two gates (Not a physical two-level logic)
  - open-collector TTL NAND gates: wired-AND logic
  - the NOR output of ECL gates: wired-OR logic

$F = (AB)' \cdot (CD)' = (AB + CD)' = (A' + B')(C' + D')$	AND-OR-INVERT function
$F = (A + B)' + (C + D)' = [(A + B)(C + D)]'$	OR-AND-INVERT function



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

(AND–OR–INVERT)



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

Degenerate Forms

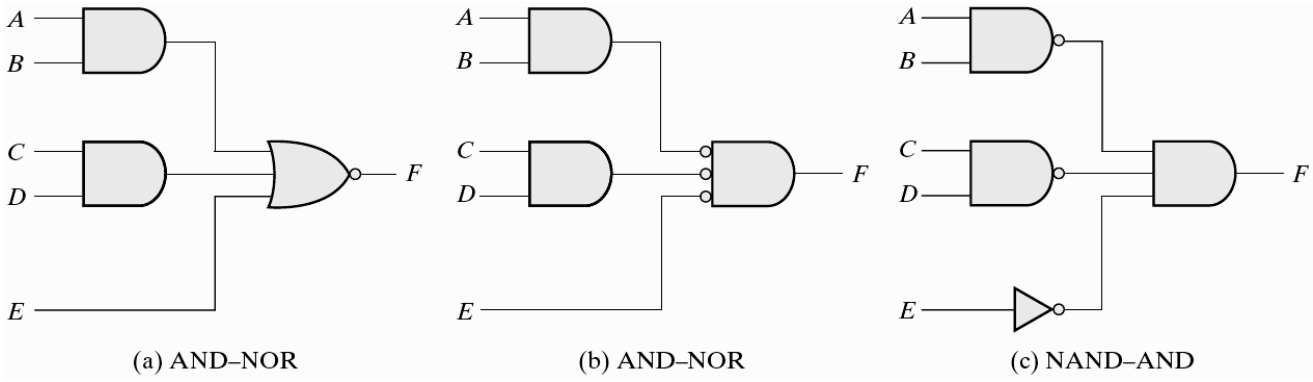
- AND/NAND/OR/NOR have 16 possible combinations of two-level forms
  - eight of them: degenerate forms $\Rightarrow$ a single operation
    - AND-AND $\Rightarrow$ AND
    - OR-OR $\Rightarrow$ OR
    - AND-NAND $\Rightarrow$ NAND
    - OR-NOR $\Rightarrow$ NOR
    - NAND-NOR $\Rightarrow$ AND
    - NOR-NAND $\Rightarrow$ OR
    - NAND-OR $\Rightarrow$ NAND
    - NOR-AND $\Rightarrow$ NOR

Nondegenerate Forms

- AND-OR $\Rightarrow$ standard sum-of-products
- NAND-NAND $\Rightarrow$ standard sum-of-products
- OR-AND $\Rightarrow$ standard product-of-sums
- NOR-NOR $\Rightarrow$ standard product-of-sums
- NAND-AND/AND-NOR $\Rightarrow$ AND-OR-INVERT (AOI) circuit (complement of sum-of-products)
- OR-NAND/NOR-OR $\Rightarrow$ OR-AND-INVERT (OAI) circuit (complement of product-of-sums)

AND-OR-Invert Implementation

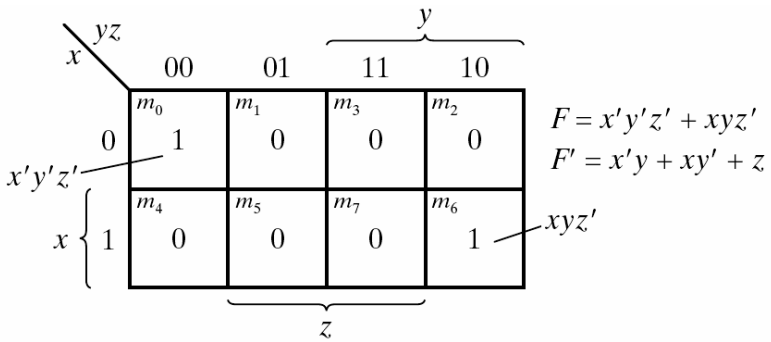
- AND-OR-INVERT (AOI) Implementation
  - NAND-AND = AND-NOR = AOI
  - $F = (AB + CD + E)'$
  - $F' = AB + CD + E$ (sum of products)



- Usage
  - Combining 0's in K-map to simplify F' in sum-of-products

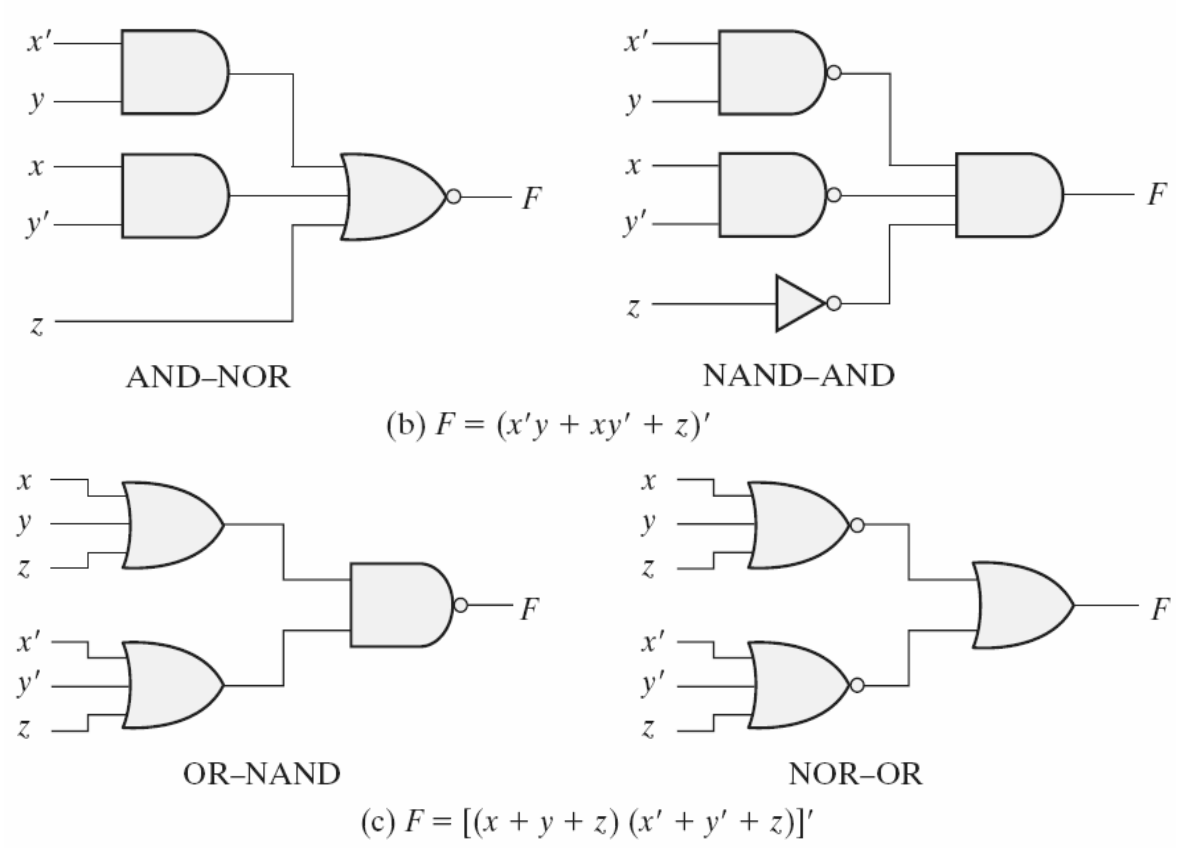
Example

- Example 3-11: Implement the function F using AOI and OAI two-level form



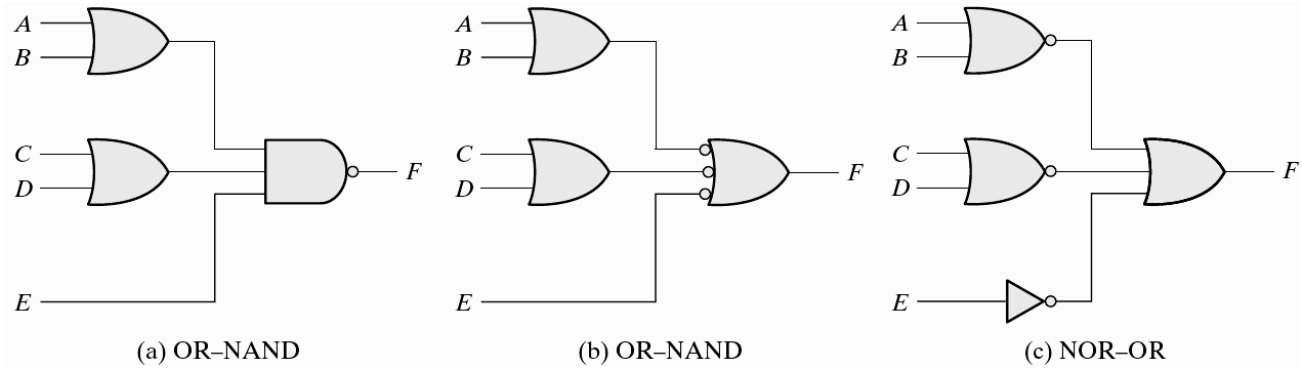
- $F' = x'y + xy' + z$ (F': sum of products)
- $F = (x'y + xy' + z)'$ (F: AOI implementation)
- $F = x'y'z' + xyz'$ (F: sum of products)
- $F' = (x + y + z)(x' + y' + z)$ (F': product of sums)
- $F = ((x + y + z)(x' + y' + z))'$ (F: OAI)

Example



OR-AND-INVERT (OAI) Implementation

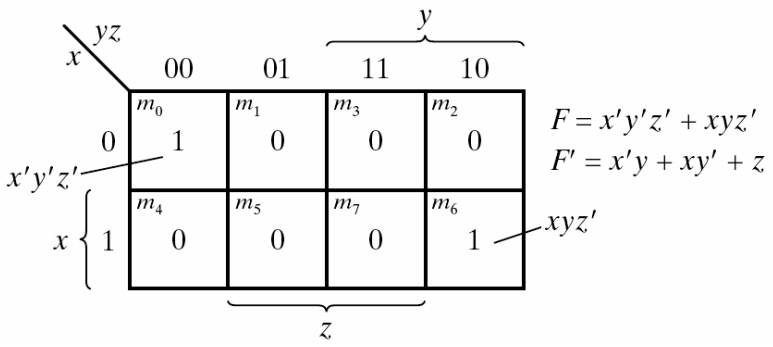
- OR-AND-INVERT (OAI) Implementation
  - OR-NAND = NOR-OR = OAI
  - $F = ((A + B)(C + D)E)'$
  - $F' = (A + B)(C + D)E$ (product of sums)



- Usage
  - Combining 1's in K-map to simplified F' in product-of-sums and then inverting the result (hint: DeMorgan's theorem)

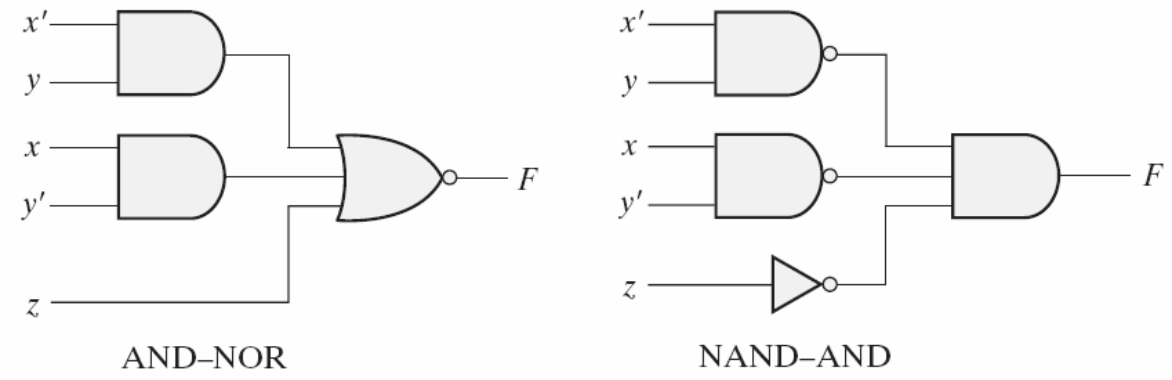
Example

- Example 3-11: Implement the function F using AOI and OAI two-level form

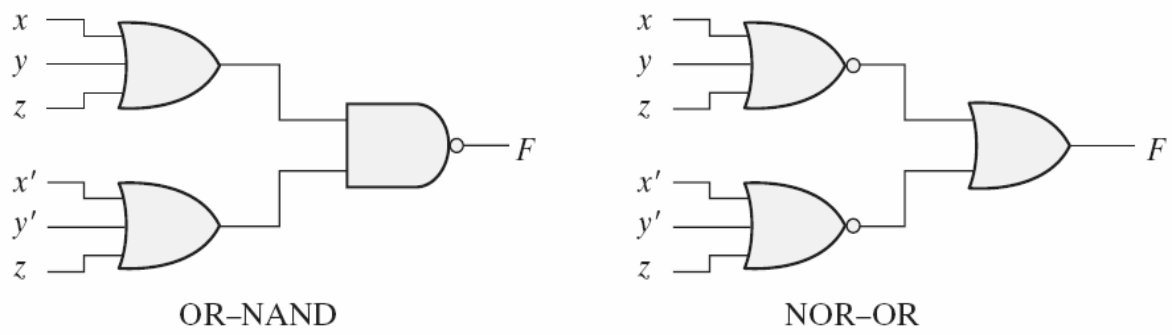


- $F' = x'y + xy' + z$ (F': sum of products)
- $F = (x'y + xy' + z)'$ (F: AOI implementation)
- $F = x'y'z' + xyz'$ (F: sum of products)
- $F' = (x + y + z)(x' + y' + z)$ (F': product of sums)
- $F = ((x + y + z)(x' + y' + z))'$ (F: OAI)

Example



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



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

Tabular Summary

Table 3.3
Implementation with Other Two-Level Forms

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

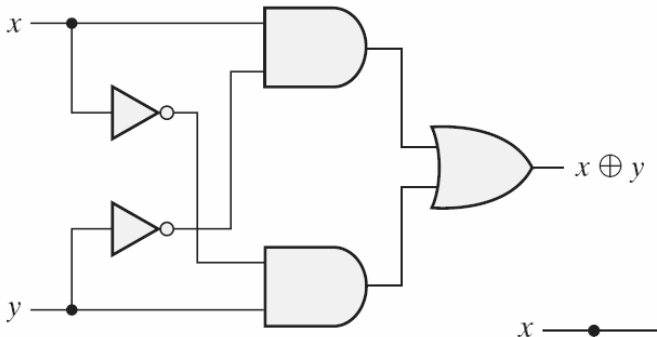
*Form (b) requires an inverter for a single literal term.

Exclusive-OR Function

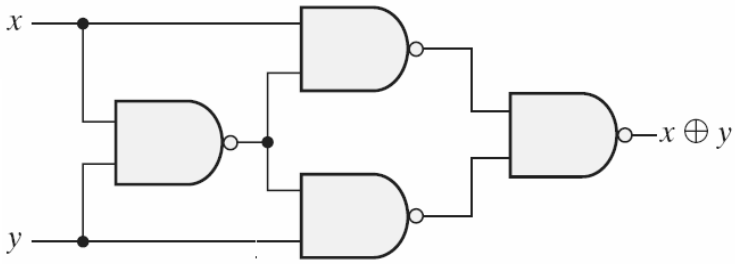
- Exclusive-OR (XOR)
  - $x \oplus y = xy' + x'y$
- Exclusive-NOR (XNOR)
  - $(x \oplus y)' = xy + x'y'$
- Some identities
  - $x \oplus 0 = x$
  - $x \oplus 1 = x'$
  - $x \oplus x = 0$
  - $x \oplus x' = 1$
  - $x \oplus y' = (x \oplus y)'$
  - $x' \oplus y = (x \oplus y)'$
- Commutative and associative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

XOR Implementations

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



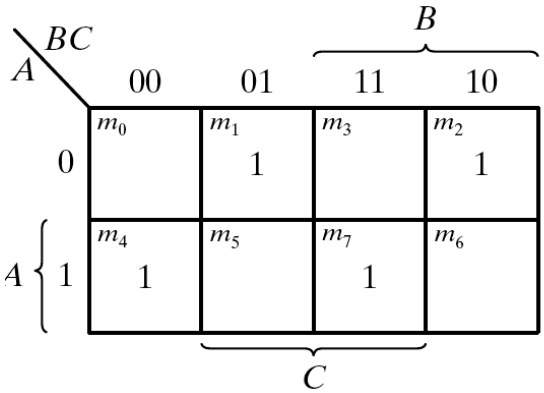
(a) With AND–OR–NOT gates



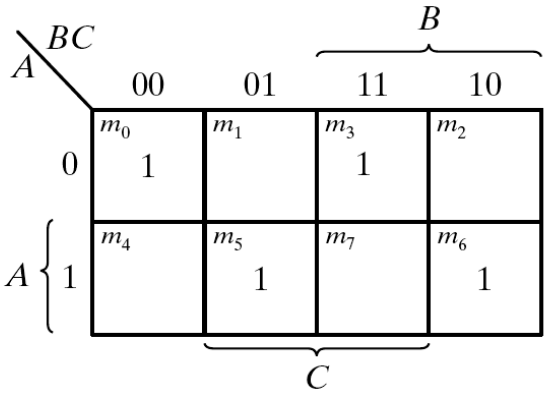
(b) With NAND gates

Odd / Even Function

- $A \oplus B \oplus C = (AB' + A'B)C' + (AB + A'B')C$
  $= AB'C' + A'BC' + ABC + A'B'C$
  $= \Sigma(1, 2, 4, 7)$
- an odd number of 1's



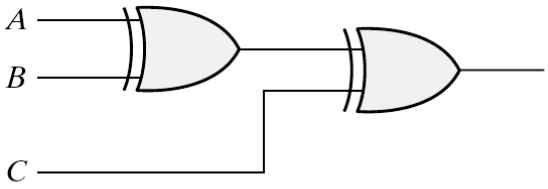
(a) Odd function $F = A \oplus B \oplus C$



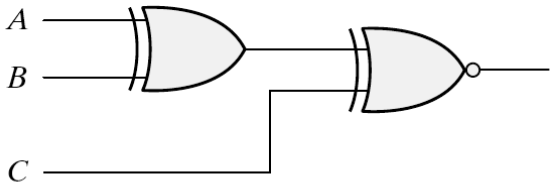
(b) Even function $F = (A \oplus B \oplus C)'$

XOR Functions

- Logic Diagrams of Odd/Even Functions



(a) 3-input odd function



(b) 3-input even function

- Four-variable Exclusive-OR function
  - $A \oplus B \oplus C \oplus D = (AB' + A'B) \oplus (CD' + C'D)$
    $= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D)$

Odd

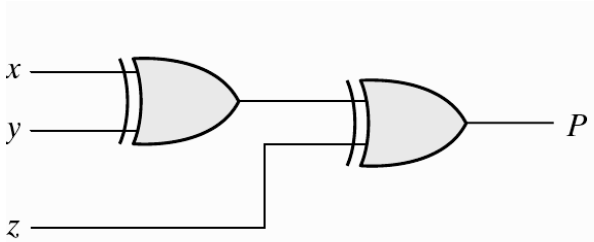
	00	01	11	10
00	m_0	m_1 1	m_3	m_2 1
01	m_4 1	m_5	m_7 1	m_6
11	m_{12}	m_{13} 1	m_{15}	m_{14} 1
10	m_8 1	m_9	m_{11} 1	m_{10}

Even

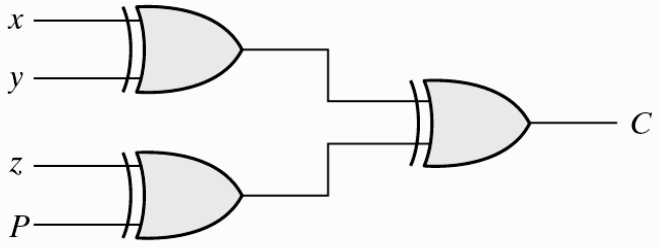
	00	01	11	10
00	m_0 1	m_1	m_3 1	m_2
01	m_4	m_5 1	m_7	m_6 1
11	m_{12} 1	m_{13}	m_{15} 1	m_{14}
10	m_8	m_9 1	m_{11}	m_{10} 1

Parity Generation and Checking

- Parity Generation (at Tx) and Parity Checker (at Rx)
  - a parity bit: $P = x \oplus y \oplus z$
  - parity check: $C = x \oplus y \oplus z \oplus P$
    - C = 1: an odd number of data bit error
    - C = 0: correct or an even # of data bit error



(a) 3-bit even parity generator



(b) 4-bit even parity checker

Parity Generation and Checking

- Truth Table (in the transmitter or storage input)

Three Bits 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

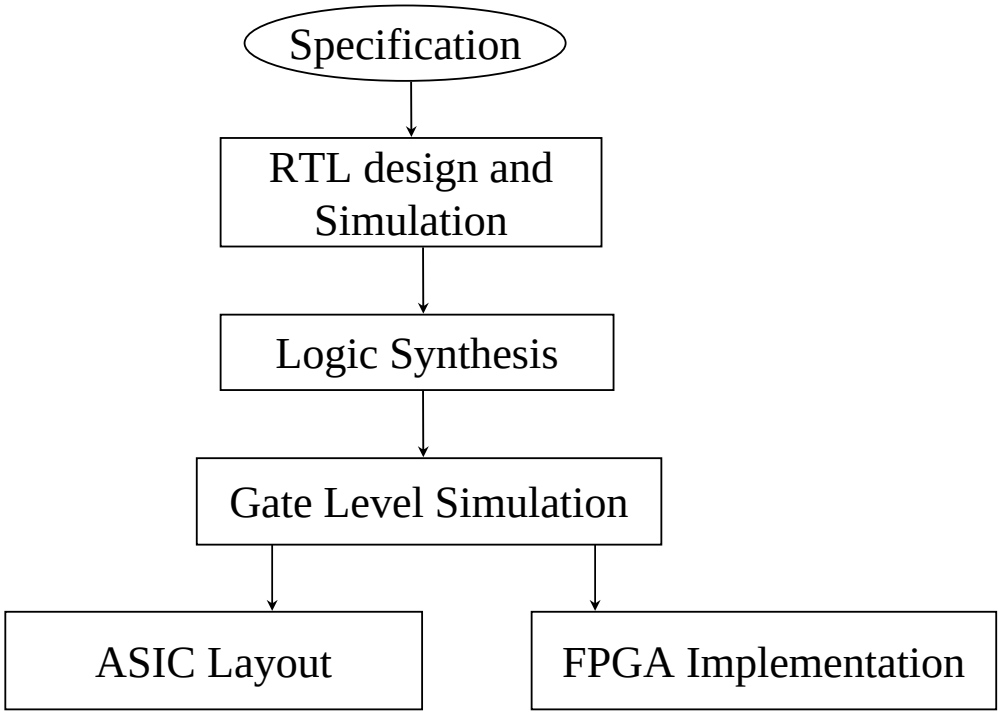
- Truth Table (in the receiver and storage output)

Four Bits Received				Parity Error Checker
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

Hardware Description Language (HDL)

- Describe the design of digital systems in a textual form (can be read by both humans and computers)
  - Design Entry: hardware structure
  - Logic Simulation: function/behavior simulations and verifications
  - Logic Synthesis: Process of deriving a list of physical components and their interconncets (netlist)
  - Timing Verification: speed test
  - Fault Simulation: identifying input stimuli to reveal the difference b/w faulty circuit and fault-free circuit
- VHDL (by DoD) and Verilog HDL (by Cadence, 益華電腦)

A Top-Down Design Flow



Verilog HDL

- ~100 keywords (lowercase):
  - Including: module, endmodule, input, output, and, or, not... (in bold face in the textbook)
  - Case sensitive: uppercase ≠ lowercase
- //: descriptions, comments (for single line)
  - /* descriptions for multilines... */

declaration

module simple_circuit(A, B, C, D, E);

⋮

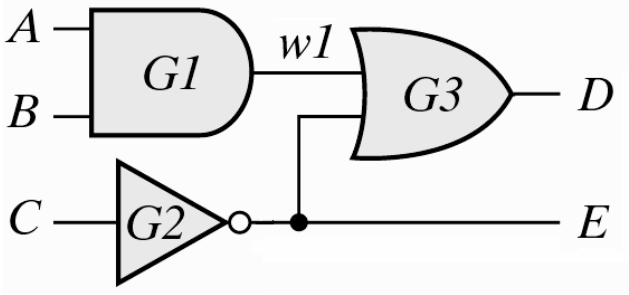
endmodule

Identifier:
1. Name of the module; 2. Composed of alphanumeric characters and underscores; 3. case sensitive; 4. start with an alphabet or underscore, can not be a number

HDL Example

- **module / endmodule** defined the building block of each design
- **input / output** define the input ports and output ports
- **and / not / or** are the **logic primitive** defined by verilog HDL; primitive_gate gate_instance(out, in)

```
// Description of simple circuit Fig. 3-37
module Simple_Circuit(A, B, C, D, E);
input A, B, C;
output D, E;
wire w1;
and G1(w1, A, B);
not G2(E, C);
or G3(D, w1, E);
endmodule
```

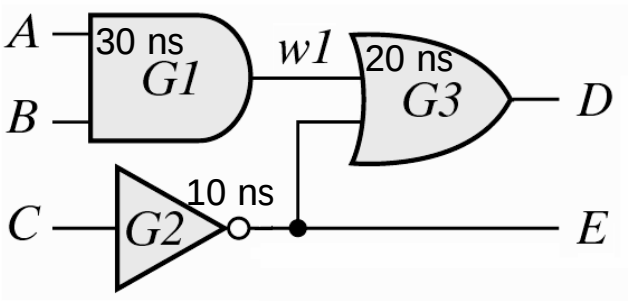


HDL with Gate Delays

- **`timescale** is a compiler directive used to specify the time unit and resolution.
- **#(time)** is a gate parameter used to specify the gate delay time.

```
// Description of simple circuit with gate delays
`timescale 1ns/100ps
module Simple_Circuit_prop_delay(A, B, C, D, E);
input A, B, C;
output D, E;
wire w1;
and #(30) G1(w1, A, B);
not #(10) G2(E, C);
or #(20) G3(D, w1, E);
endmodule
```


Output of Gates after Delay



Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	0 0 0	1 0 1
Change	—	1 1 1	1 0 1
	10	1 1 1	0 0 1
	20	1 1 1	0 0 1
	30	1 1 1	0 1 0
	40	1 1 1	0 1 0
	50	1 1 1	0 1 1

HDL Simulation Example

Test bench for simulating the circuit with delay

HDL Example 3.3

```
// Test bench for Simple_Circuit_prop_delay

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

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

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

initial #200 $finish;
endmodule
```

output
input

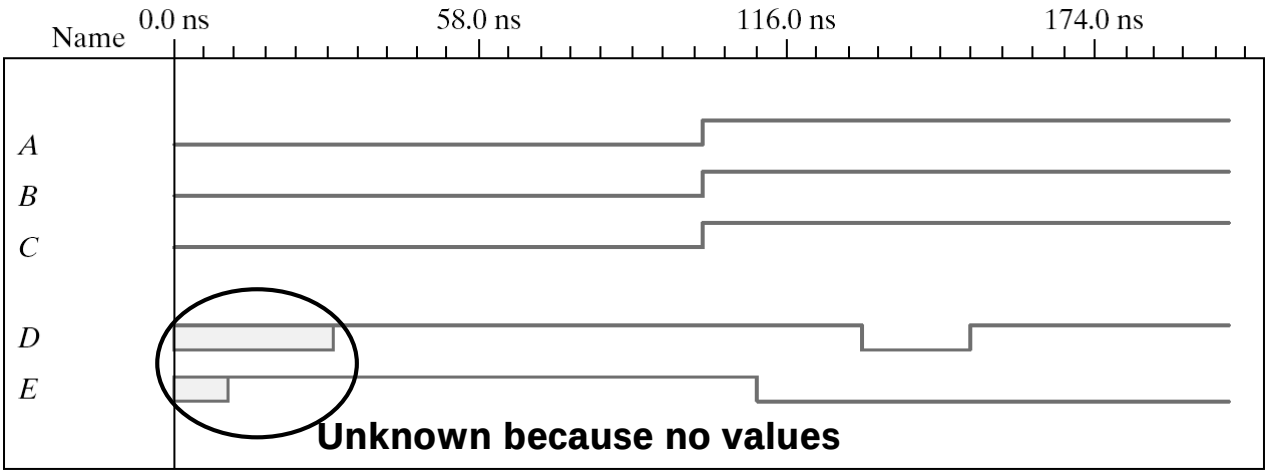
block statement

1 binary bit with value 0

After 100 ns
(A, B, C) = (1, 1, 1)

Terminate the execution after 200 ns

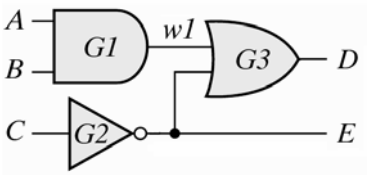
Simulation output for HDL Example



Boolean Expression

- Boolean expression for the circuit of Fig. 3.37

```
assign D = (A & B)|~C;
```



- Verilog HDL logic operator:
  - & ⇒ AND
  - | ⇒ OR
  - ~ ⇒ NOT (complement)
  - ^ ⇒ XOR
- Continuous assignment statement **assign** are used to describe the output function

$E = A + BC + B'D$
$F = B'C + BC'D'$

```
// Circuit specified with Boolean expressions
module circuit_boolean_CA(E, F, A, B, C, D);
input A, B, C, D;
output E, F;
assign E = A|(B & C)|(~B & D);
assign F = (~B & C)|(B & ~C & ~D);
endmodule
```


User-Defined Primitives

- General rules:
  - It is declared with the keyword **primitive**, followed by a name and port list.
  - There can be only one output, and it must be listed first in the port list and declared with keyword **output**.
  - There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
  - The truth table is enclosed within the keywords **table** and **endtable**.
  - The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
  - The declaration of a UDP ends with the keyword **endprimitive**.
- Declaration:

```
Circuit_with_UDP_02467 (E, F, A, B, C, D);
```

HDL Example 3.5

HDL Example 3.5

```
// Verilog model: User-defined Primitive

primitive UDP_02467 (D, A, B, C);
 output D;
 input   A, B, C;

// Truth table for D = f (A, B, C) = Σ (0, 2, 4, 6, 7);
 table
//     A       B       C       :       D       // Column header comment
       0       0       0       :       1;
       0       0       1       :       0;
       0       1       0       :       1;
       0       1       1       :       0;
       1       0       0       :       1;
       1       0       1       :       0;
       1       1       0       :       1;
       1       1       1       :       1;
 endtable
endprimitive
```


HDL Example 3.5

```
// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module Circuit_with_UDP_02467 (e, f, a, b, c, d);
 output       e, f;
 input        a, b, c, d;

 UDP_02467          (e, a, b, c);
 and                (f, e, d);          // Option gate instance name omitted
endmodule
```