Logic Optimization

Mano & Ciletti Ch.3 /Brown & Vranesic Ch.4

Outline

- Truth tables, minterms, and canonical form
- Karnaugh map method
- From a cover to a network of gate
- Incompletely specified functions
- Product-of-sums implementation
- Verilog of Combinational Logic

Review

- The world is digital
  - Analog at the edges, hardware for demanding problems
- Digital signals
  - Encode discrete states in a continuous signal
  - Tolerate noise
- Boolean Algebra (0, 1, $\wedge$, $\vee$)
  - Axioms, properties, duality
  - Logic equations express binary functions
- Combinational logic
  - Output is a function only of current input
- Verilog
  - Defines hardware modules, assign, case

Boolean Algebra and Combinational Logic

- How to implement combinational logic by hand
  - Given a description of a logic function
  - Generate a gate-level circuit that realizes that function
- Knowing the basics
  - To understand how they are done
  - Demystifies what the synthesis tools do
  - Better understanding of what synthesis tools can and cannot do
- However, the general practice in modern days is:
  - Design using Verilog
  - Simulate with test cases
  - Generate gates with synthesis tools

Example of a Combinational Logic Circuit

- Prime detector
  - Specification:
    A circuit that outputs a 1 when its four-bit input represents a prime number in binary
  - Or
    f(d, c, b, a) = 1 if (d, c, b, a) is a prime
- Truth table
  - For an n-input function, a truth table has 2^n rows
    - 2^n input combinations

Truth Table

- f(d, c, b, a) = 1 if (d, c, b, a) is a prime

No	d c b a	f
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	1
3	0 0 1 1	1
4	0 1 0 0	0
5	0 1 0 1	1
6	0 1 1 0	0
7	0 1 1 1	1
8	1 0 0 0	0
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	1
12	1 1 0 0	0
13	1 1 0 1	1
14	1 1 1 0	0
15	1 1 1 1	0

Abbreviated Truth Table

No	d c b a	f
1	0 0 0 1	1
2	0 0 1 0	1
3	0 0 1 1	1
5	0 1 0 1	1
7	0 1 1 1	1
11	1 0 1 1	1
13	1 1 0 1	1
Otherwise		0

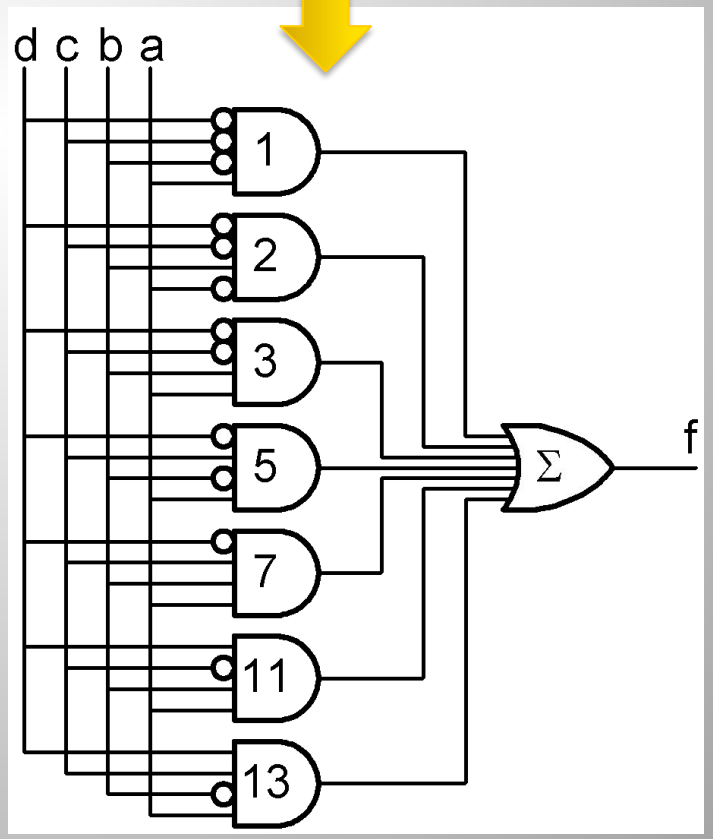
Canonical Form: Sum of Minterms

No	d	c	b	a	f
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
5	0	1	0	1	1
7	0	1	1	1	1
11	1	0	1	1	1
13	1	1	0	1	1
Otherwise					0

$$f = (\bar{d} \wedge \bar{c} \wedge \bar{b} \wedge a) \vee (\bar{d} \wedge \bar{c} \wedge b \wedge \bar{a}) \vee (\bar{d} \wedge \bar{c} \wedge b \wedge a) \vee (\bar{d} \wedge c \wedge \bar{b} \wedge a) \vee (\bar{d} \wedge c \wedge b \wedge a) \vee (d \wedge \bar{c} \wedge b \wedge a) \vee (d \wedge c \wedge \bar{b} \wedge a)$$

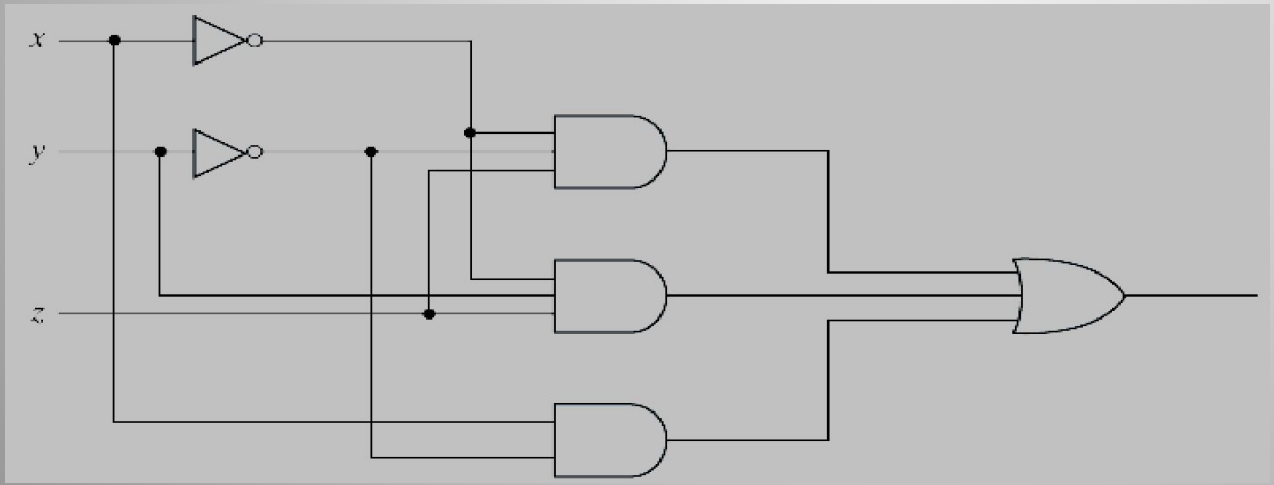
$$f = \sum_{(d,c,b,a)} m(1, 2, 3, 5, 7, 11, 13)$$

Schematic Logic Diagram



Gate-Level Minimization

- *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 required
  - the complexity of the algebraic expression



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

Two-Level Logic

. SOP/POS form corresponds to a two-level logic circuit (if each variable is provided in both their complemented and uncomplemented forms).

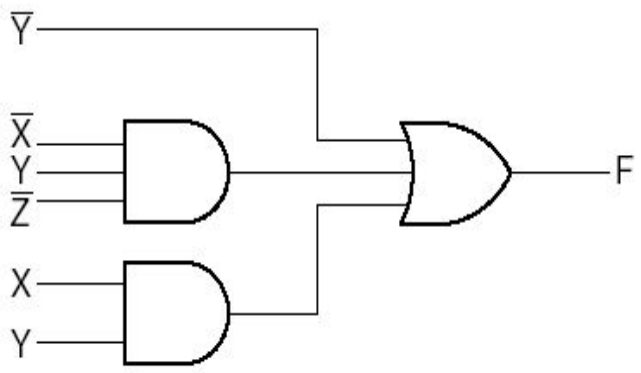


Fig. 2-5 Sum-of-Products Implementation

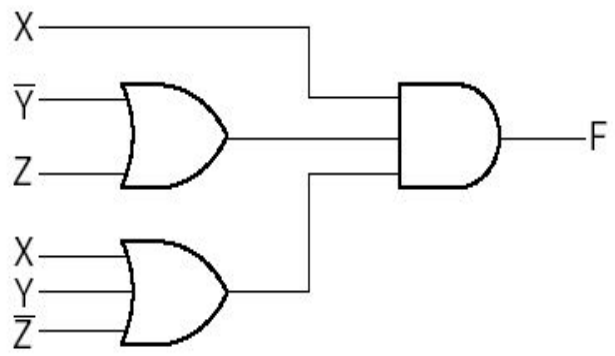


Fig. 2-7 Product-of-Sums Implementation

Minimization of Two-Level Logic

- Boolean function simplification
  - Sum of products (or product of sum) in the simplest form
    - minimum number of terms
    - minimum number of literals
  - The simplified expression may not be unique
- To find the best solution
  - Compare the minimum SOP and minimum POS forms

The Map Method

- Logic minimization methods
  - Algebraic approaches: lack specific rules
  - Karnaugh map approach
    - a simple straight forward procedure
    - a pictorial form of a truth table
    - applicable if the # of variables < 7
- Karnaugh map
  - a diagram made up of squares
  - each square represents one minterm

Two-Variable K-Map

- 2 variables => 4 minterms

m_0	m_1
m_2	m_3

(a)

X \ Y	0	1
0	$\overline{X}\,\overline{Y}$	$\overline{X}\,Y$
1	$X\,\overline{Y}$	$X\,Y$

(b)

Fig. 2-8 Two-Variable Map

Two-Variable Map Simplification

(a) Map for $F = XY$. (b) Map for $G = X'Y+XY'+XY$.

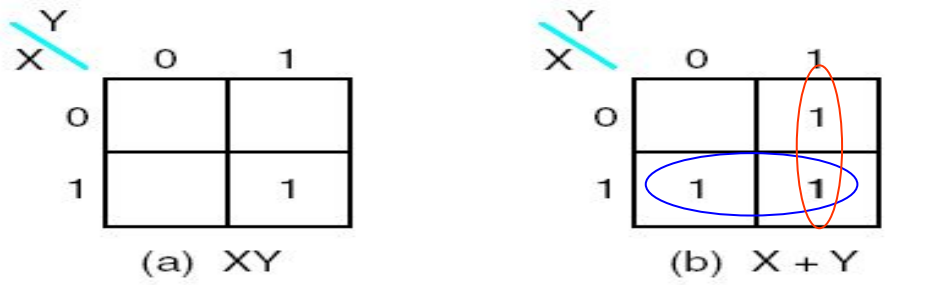


Fig. 2-9 Representation of Functions in the Map

- Minterms in adjacent cells can be combined into a single product term since they differ in only one variable.
- In (b), $XY'+XY$ can be combined into X, and $XY+X'Y$ can be combined into Y. Hence $G = X+Y$.

Three-Variable K-Map

- 3 variables => 8 minterms

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)

X \ YZ	00	01	11	10
0	$\bar{X}\bar{Y}\bar{Z}$	$\bar{X}\bar{Y}Z$	$\bar{X}YZ$	$\bar{X}Y\bar{Z}$
1	$X\bar{Y}\bar{Z}$	$X\bar{Y}Z$	XYZ	$XY\bar{Z}$

(b)

Fig. 2-10 Three-Variable Map

- Columns are identified by the sequence 00, 01, 11, 10 to ensure that adjacent cells only differ in the value of one variable.

- The first and the last columns are also considered as adjacent.

3-Variable Map Simplification Examples

- In a 3-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells.

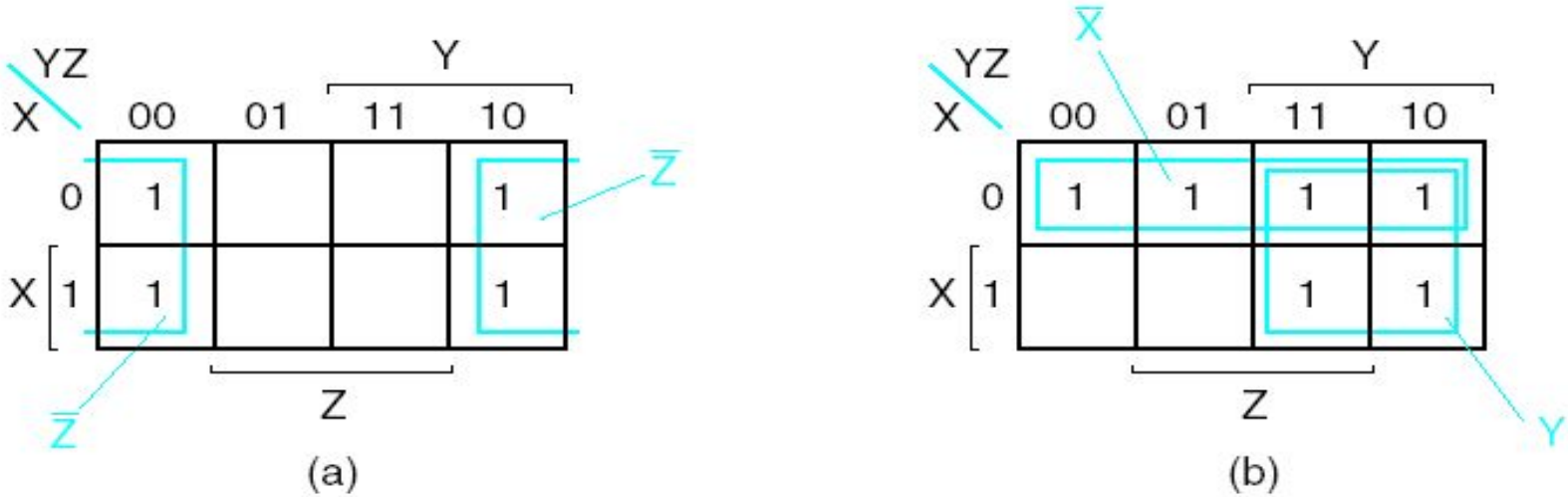
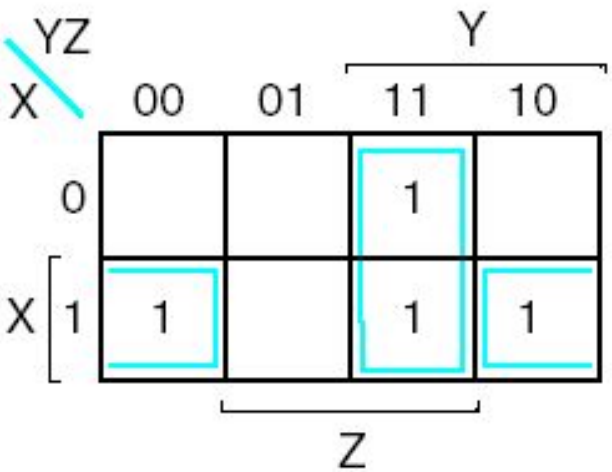
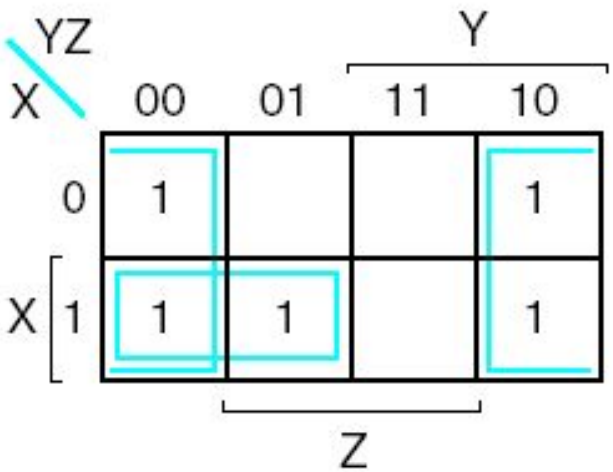


Fig. 2-13 Product Terms Using Four Minterms

More Simplification Examples



(a) $F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7)$
$= YZ + X\overline{Z}$



(b) $F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6)$
$= \overline{Z} + X\overline{Y}$

Fig. 2-14 Maps for Example 2-4

More Examples

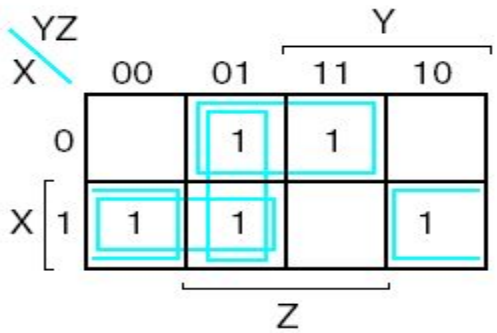


Fig. 2-15 $F(X, Y, Z) = \Sigma m(1, 3, 4, 5, 6$

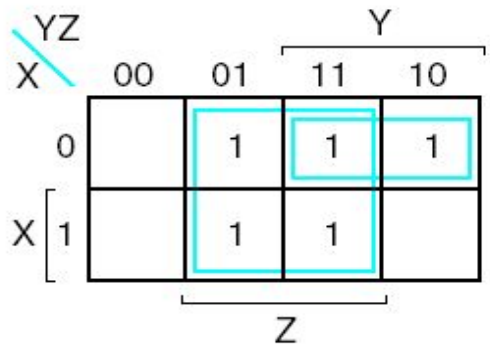


Fig. 2-16 $F(X, Y, Z) = \Sigma m\ (1, 2, 3, 5, 7)$

Two simplified solutions:

$F = X'Z + XZ' + XY'$

$F = X'Z + XZ' + Y'Z$

$F = Z + X'Y$

Q. Are they identical?

Q. Are they equivalent?

Four-Variable K Map

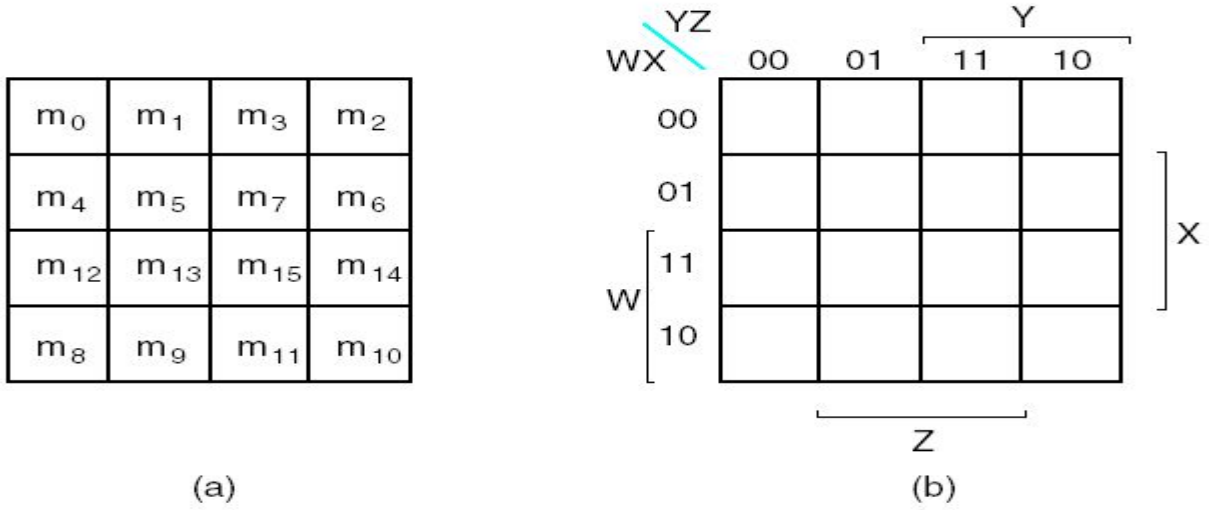


Fig. 2-17 Four-Variable Map

- The left and right edges of the map are adjacent in terms of assignment of variables, so are the top and bottom edges.
- The 4 corners of the map are adjacent to each other

4-Variable Map Simplification Example

Simplify $F(W,X,Y,Z) = \sum m(0,1,2,4,5,6,8,9,12,13,14)$.

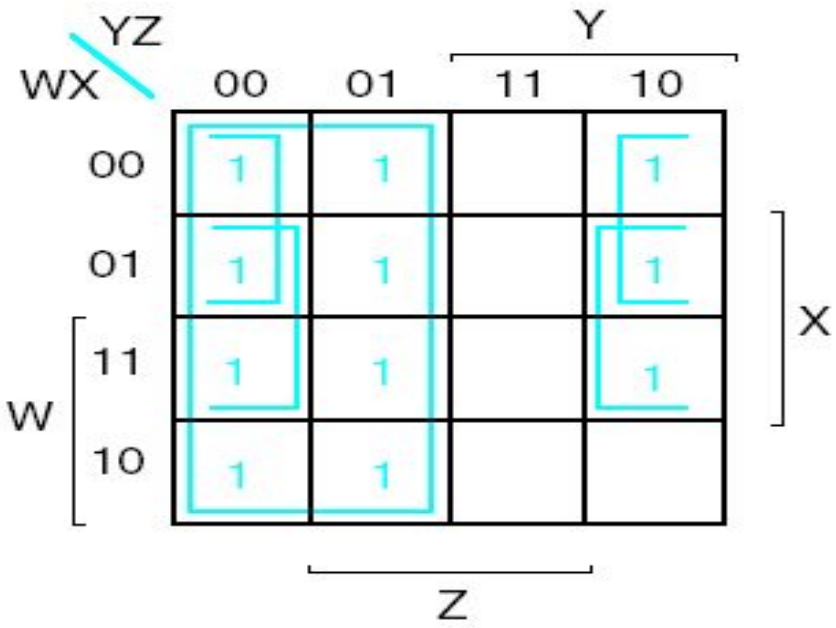


Fig. 2-19 Map for Example 2-5: $F = \overline{Y} + \overline{W}\overline{Z} + X\overline{Z}$

More Example

Simplify $F = A'B'C' + B'CD' + AB'C' + A'BCD'$.

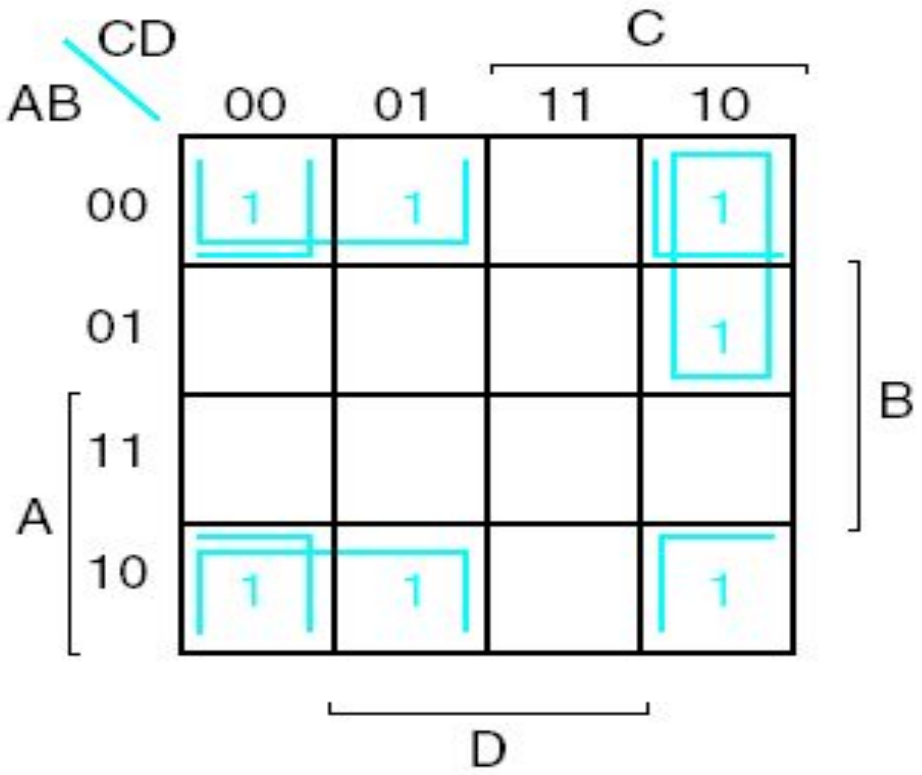


Fig. 2-20 Map for Example 2-6: $F = \overline{B}\overline{D} + \overline{B}\overline{C} + \overline{A}C\overline{D}$

Example: K-Map for Prime Detector

. The prime number detector

No	d c b a	f
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	1
3	0 0 1 1	1
4	0 1 0 0	0
5	0 1 0 1	1
6	0 1 1 0	0
7	0 1 1 1	1
8	1 0 0 0	0
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	1
12	1 1 0 0	0
13	1 1 0 1	1
14	1 1 1 0	0
15	1 1 1 1	0

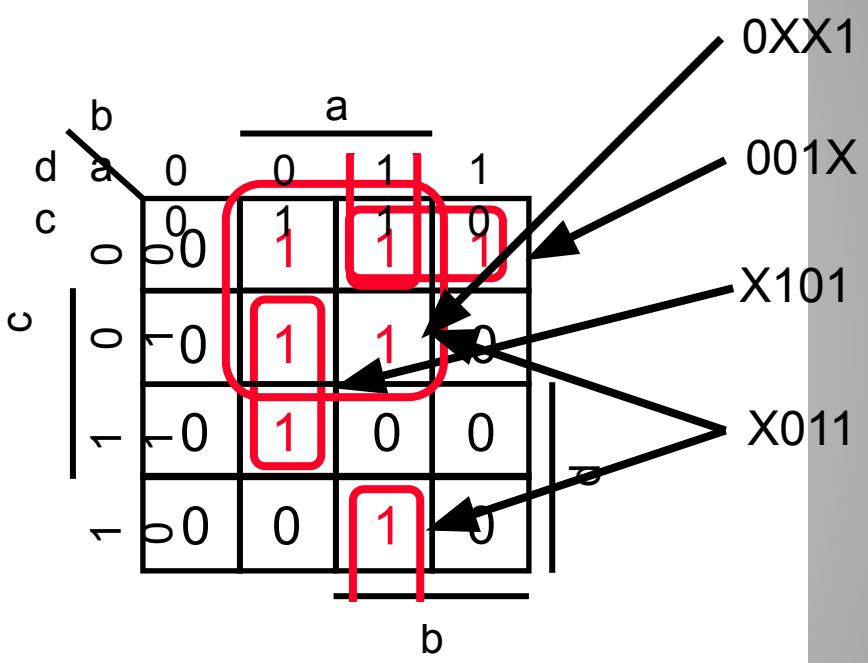
d c \ b a	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	0	1	0	0
10	0	0	1	0

K-Map for Prime Detector (cont)

. Obtain implicants as large as possible

– Fewer gates

dc \ ba	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	0	1	0	0
10	0	0	1	0

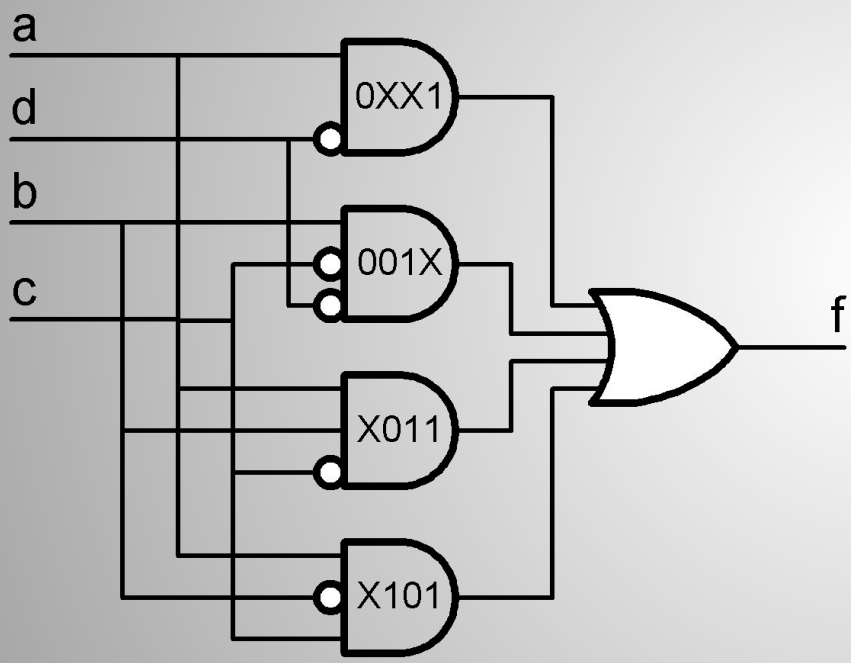


$$f = (b \wedge \overline{c} \wedge \overline{d}) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c}) \vee (a \wedge \overline{d})$$

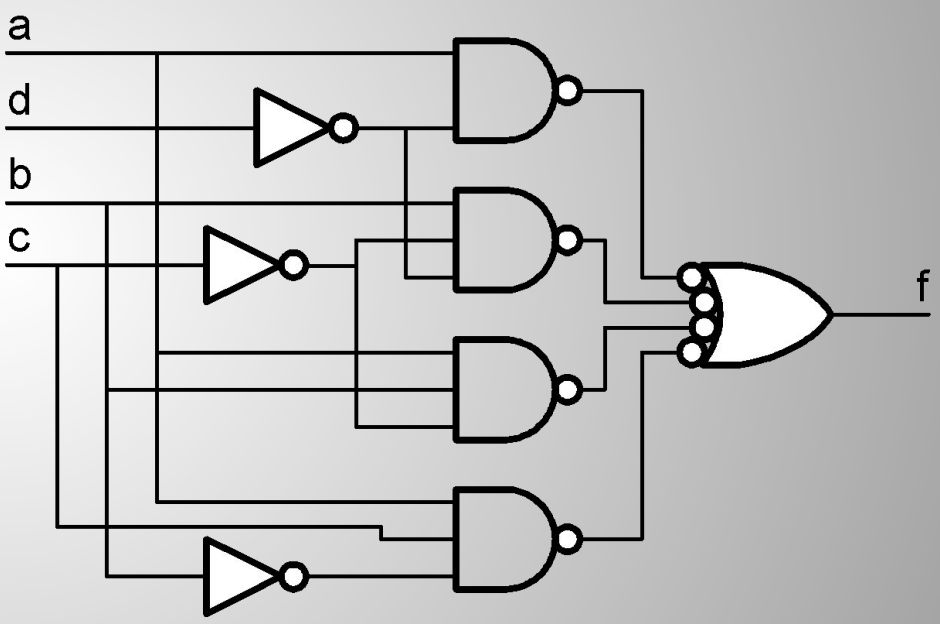
From Optimized Sum-of-Products Function to Gates

$$f = (b \wedge \overline{c} \wedge \overline{d}) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge \overline{c}) \vee (a \wedge \overline{d})$$

AND-OR



NAND-NAND



In practice, CMOS gates are always inverting, so the real circuit might use NAND-NAND design instead of AND-OR.

Summary of Some Definitions

- Minterm: a product term that includes every input variable or its complement.

- Implicant: a product term that if true implies the function is true.

- Prime Implicant: an implicant that cannot be made any larger and still be an implicant.

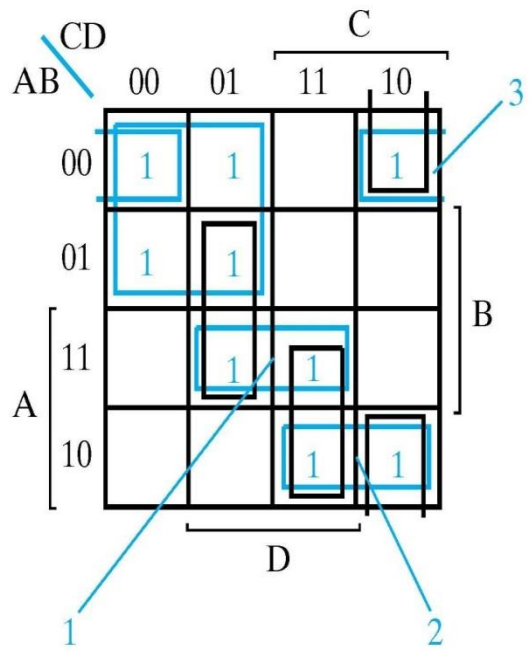
- Essential Prime Implicant: the only prime implicant that contains a particular minterm of the function.

Covering A Function

- A simple cost function of implicants:
  - # of its variables (literals)
- Procedure to select an inexpensive set of implicants
  - Start with an empty cover
  - Add all essential prime implicants to the cover
  - For each remaining uncovered minterm, add the largest implicant to cover it
- Good cover: no guarantee it is the *lowest-cost cover*

Covering A Function (cont)

. Simplify *F(A,B,C,D) = Σm(0,1,2,4,5,10,11,13,15).*



Essential prime implicant(s):

A'C'

Prime implicant(s) selected using the selection rule:

ABD (1), AB'C(2), A'B'D' (3)

Finally, is selected to form a complete cover.

$F =$

Incompletely Specified Functions

- The value of some function is not specified for certain combinations of variables
  - E.g. For binary-coded decimal, 1010-1111 are don't cares
- The don't care conditions can be utilized in logic minimization
  - can be implemented as 0 or 1

Simplification with Don't Cares

Simplify $F(A,B,C,D) = \Sigma m(1,3,7,11,15) + \Sigma d(0,2,5)$.

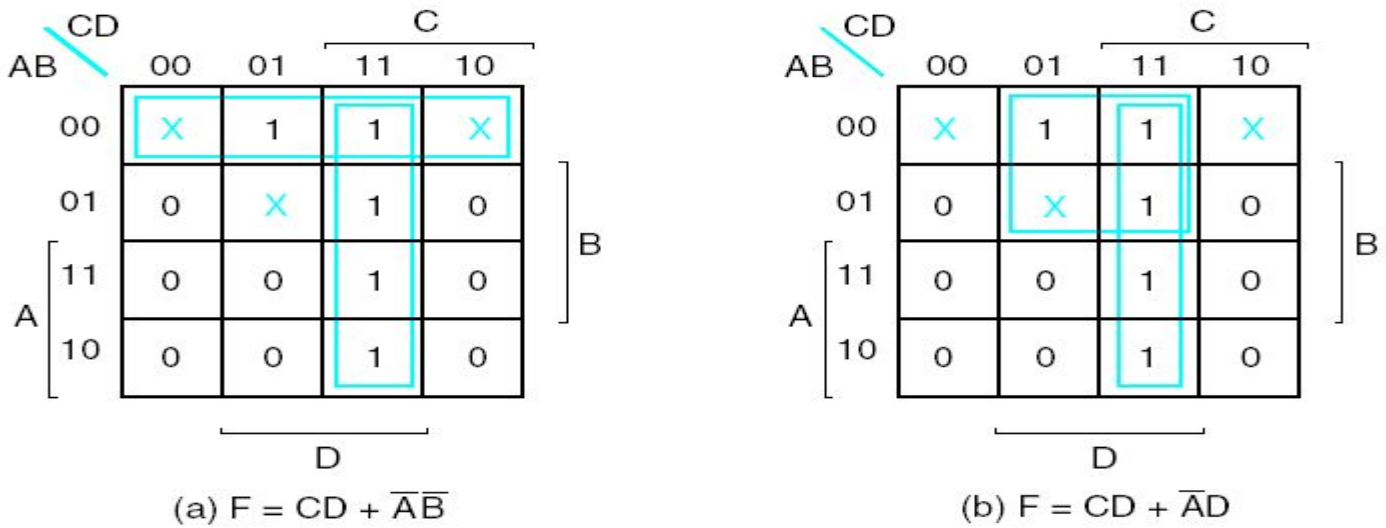


Fig. 2-25 Example with Don't-Care Conditions

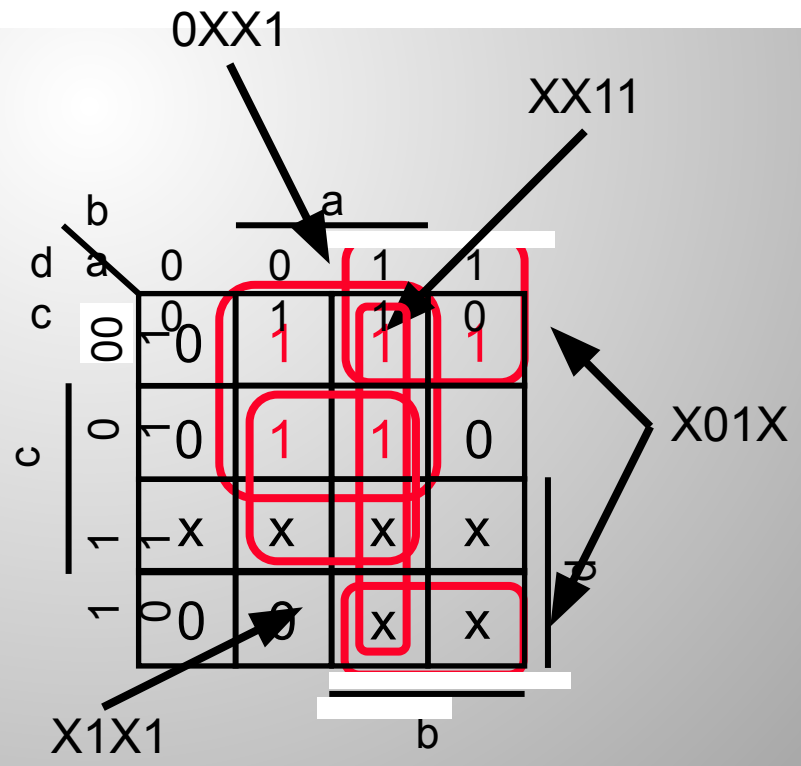
Q. Are CD+A'B' and CD+A'D algebraically equivalent?

Incompletely Specified Functions

. Decimal prime detector, with don't cares

$$f = \sum_{(d,c,b,a)} m(1,2,3,5,7) + D(10,11,12,13,14,15)$$

dc \ ba	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	x	x	x	x
10	0	0	x	x

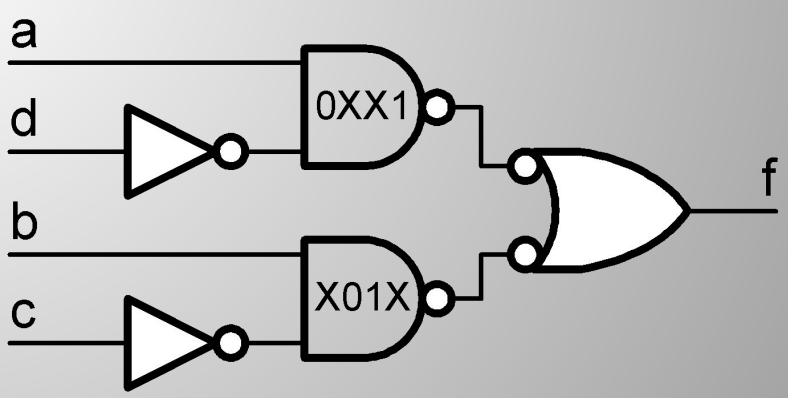


Incompletely Specified Functions (cont)

. Decimal prime detector, with don't cares

dc \ ba	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	x	x	x	x
10	0	0	x	x

$$f = (a \wedge \overline{d}) \vee (b \wedge \overline{c})$$



Finding Minimum POS Form

- Approach
  - Simplify F' in sum of products form
- Apply DeMorgan's theorem $F = (F')'$
- F': sum of products => F: product of sums

Finding Minimum POS Form Example

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

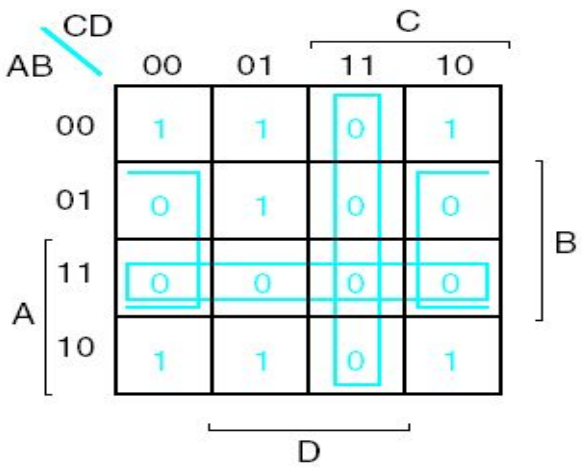


Fig. 2-24 Map for Example 2-10: $F = (\overline{A} + \overline{B})\ (\overline{C} + \overline{D})\ (\overline{B} + D)$

$$F' = AB + CD + BD'$$

$$\Rightarrow F = (A'+B')(C'+D')(B'+D)$$

Functionally Complete Set of Logic Gates

- A set of logic operations is said to be *functionally complete* if any Boolean function can be expressed in terms of these operations.
- The set AND, OR, and NOT is functionally complete as any Boolean function can be expressed in SOP and POS forms.
- Hence, any set of logic gates which can realize AND, OR, and NOT is also functionally complete.

  e.g. AND and NOT form a functionally complete set.

  e.g. OR and NOT form a functionally complete set.

NAND and NOR Gates

- Note that NAND gate forms a functionally complete set by itself. *(Why?)*
- So NAND gate is said to be a *universal gate*.
- Is NOR gate a universal gate?

NAND Circuit

- A Boolean function in terms of AND, OR, and NOT can be readily converted into NAND logic.
- We can use alternative gate symbols to facilitate the conversion:

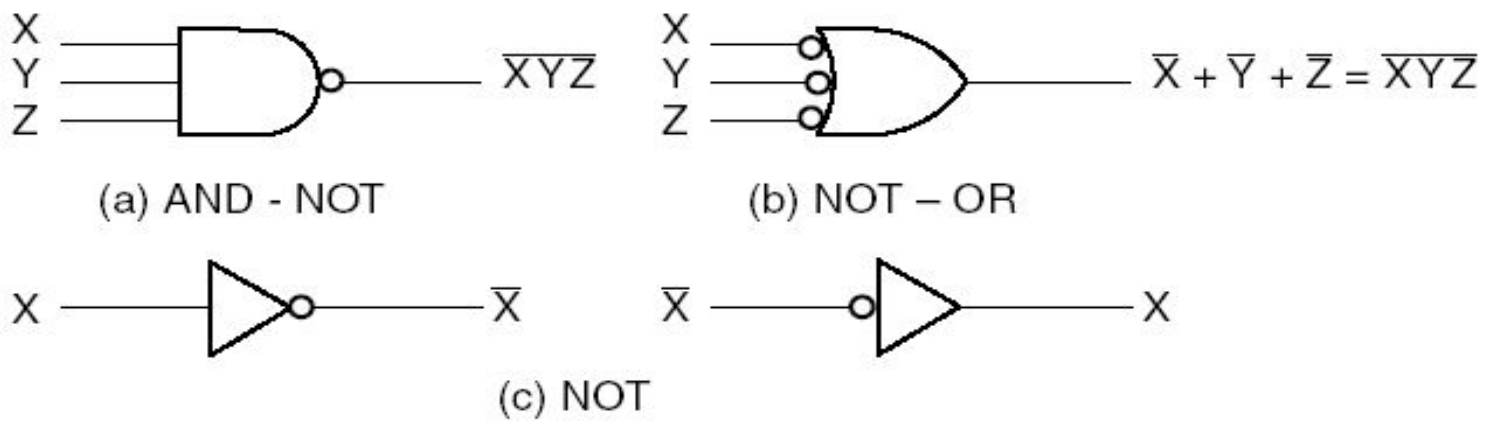


Fig. 2-28 Alternative Graphics Symbols for NAND and NOT Gates

Conversion into NAND Circuit: Example 1

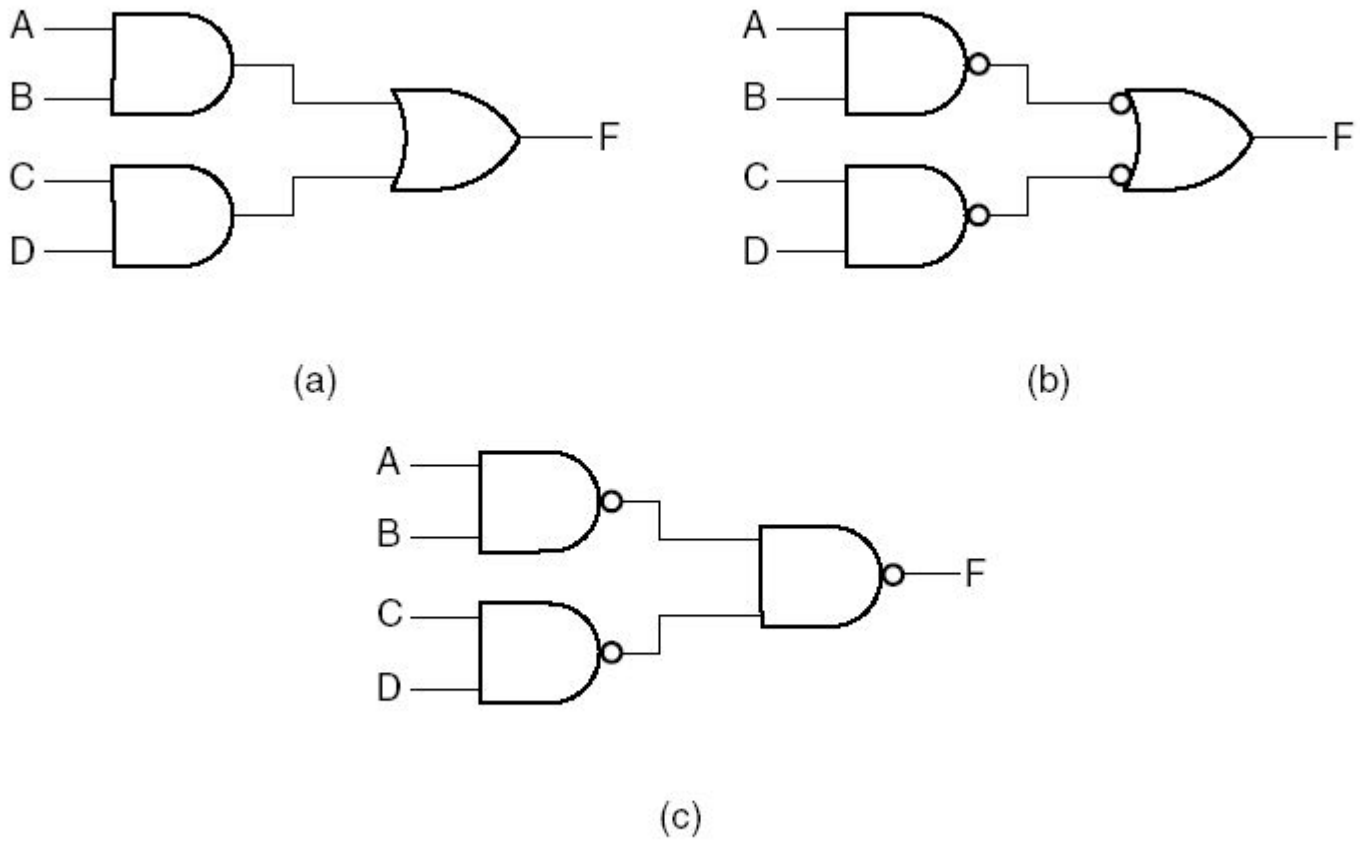
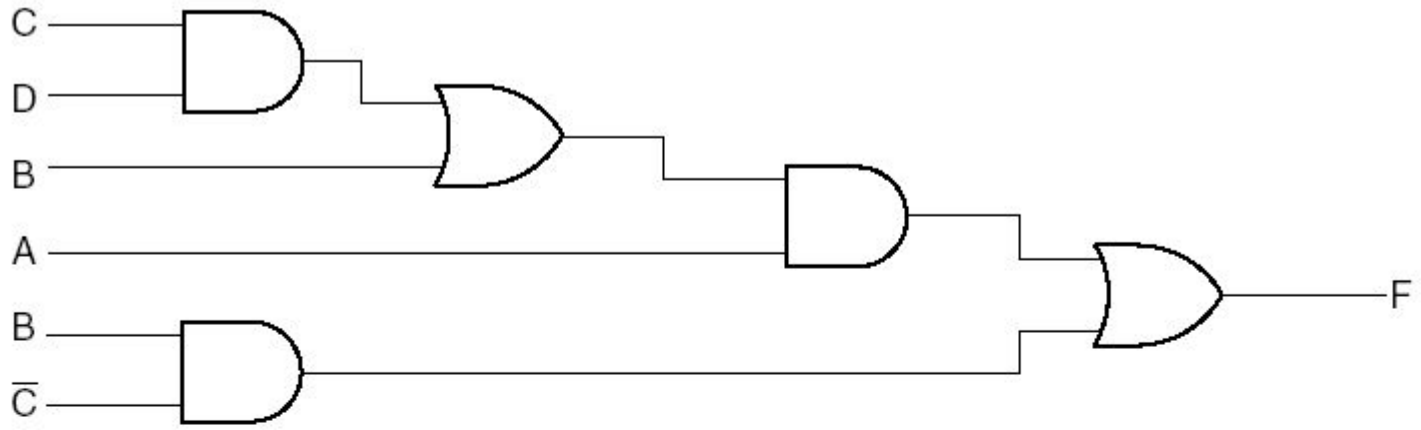
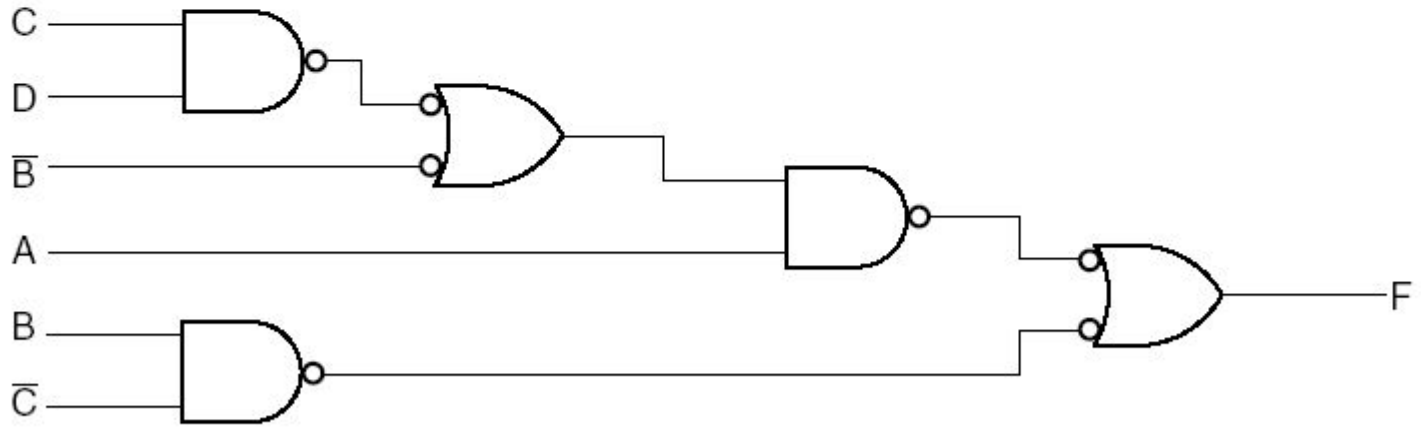


Fig. 2-29 Three Ways to Implement F = AB + CD

Conversion into NAND Circuit: Example 2



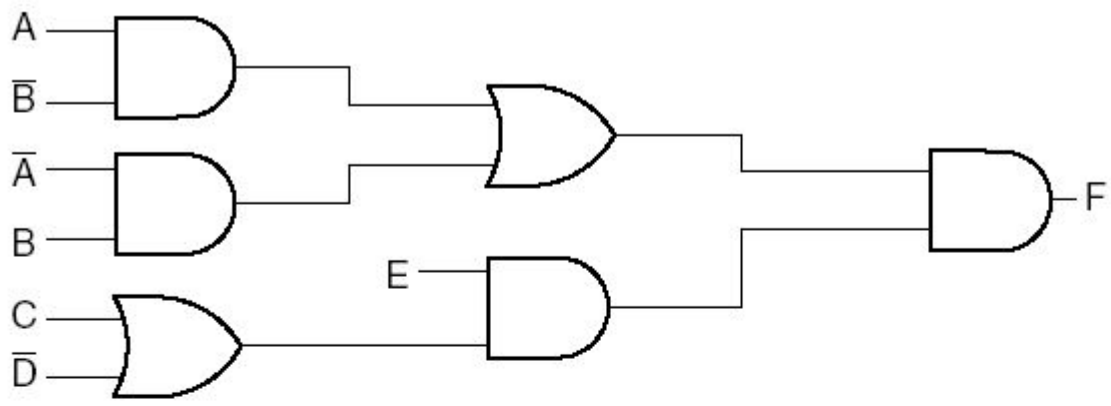
(a) AND – OR gates



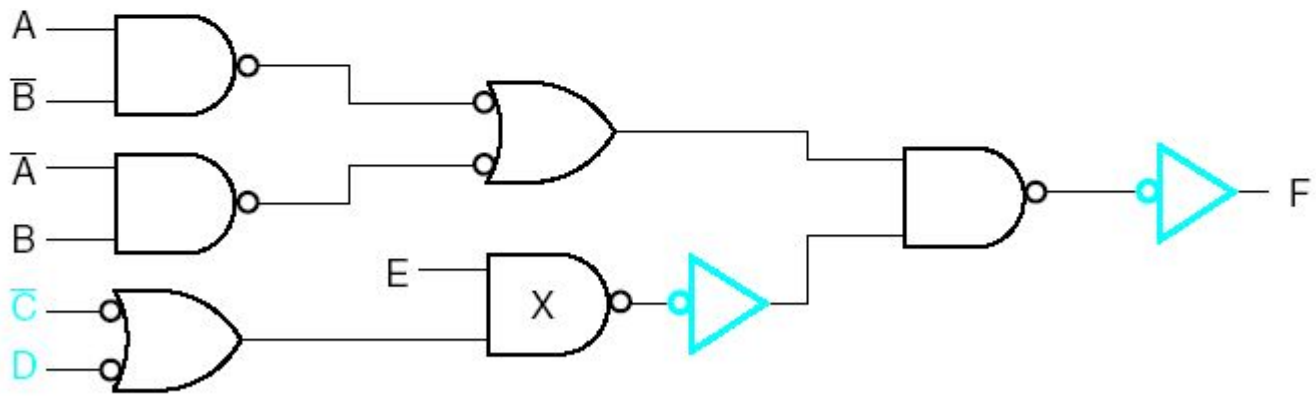
(b) NAND gates

Fig. 2-31 Implementing $F = A\,(CD + B) + B\overline{C}$

Conversion into NAND Circuit: Example 3



(a) AND – OR gates

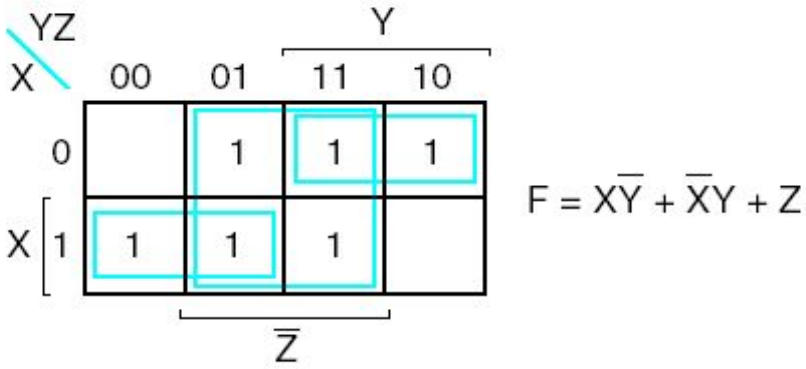


(b) NAND gates

Fig. 2-32 Implementing $F = (A\overline{B} + \overline{A}B)E(C + \overline{D})$

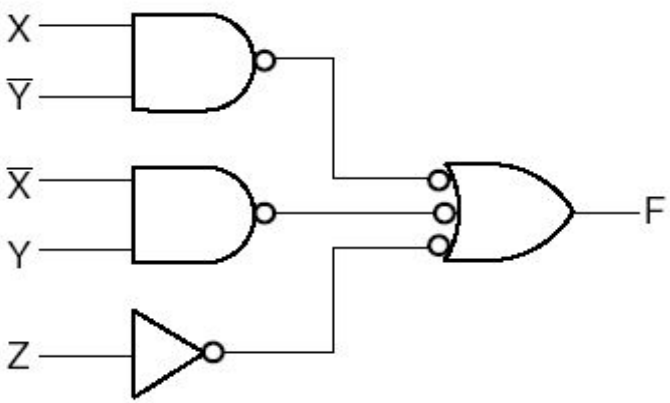
Designing with NAND Gates

e.g.

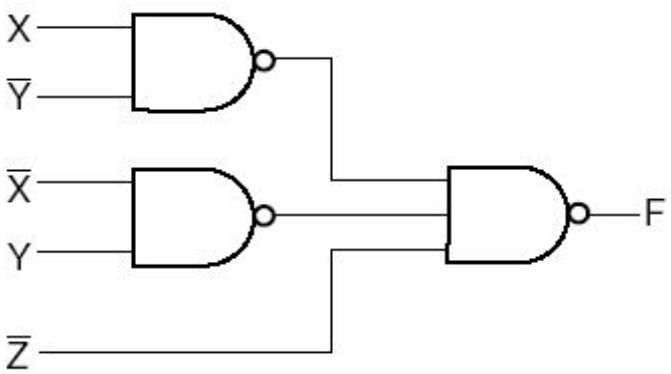


$F = X\overline{Y} + \overline{X}Y + Z$

(a)



(b)



(c)

Fig. 2-30 Solution to Example 2-12

NOR Circuit

- A Boolean function in terms of AND, OR, and NOT can be readily converted into NOR logic
- We can use alternative gate symbols to facilitate the conversion:

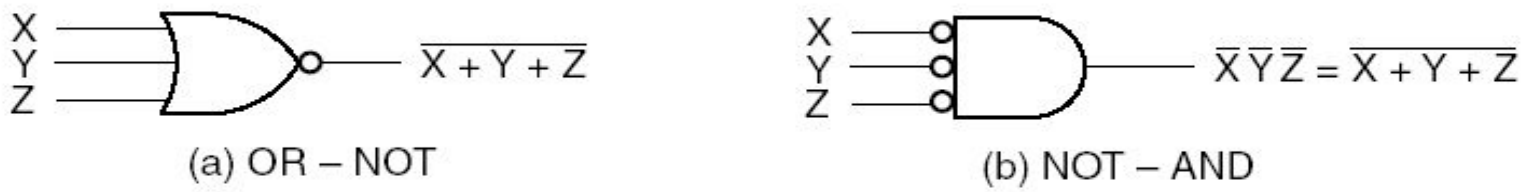


Fig. 2-34 Two Graphic Symbols for NOR Gate

Conversion into NOR Circuit: Example 1

Direct conversion from a product of sums into a NOR circuit:

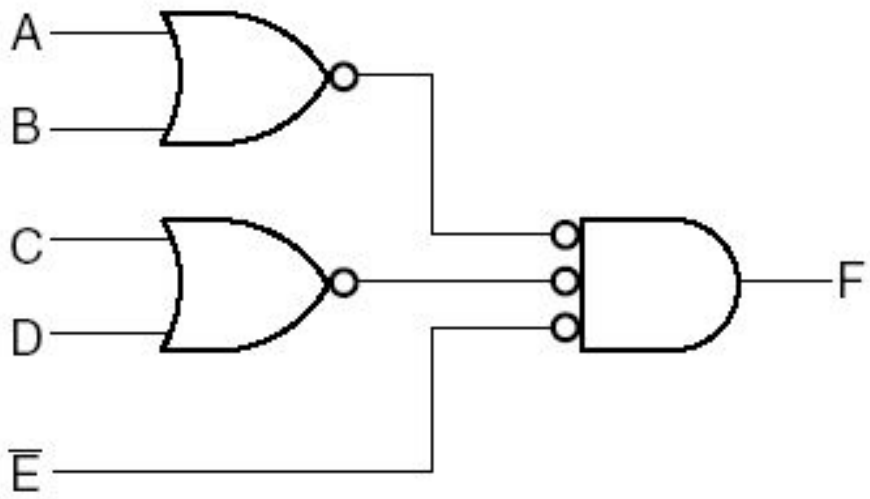


Fig. 2-35 Implementing $F = (A + B)(C + D)E$ with NOR Gates

Conversion into NOR Circuit: Example 2

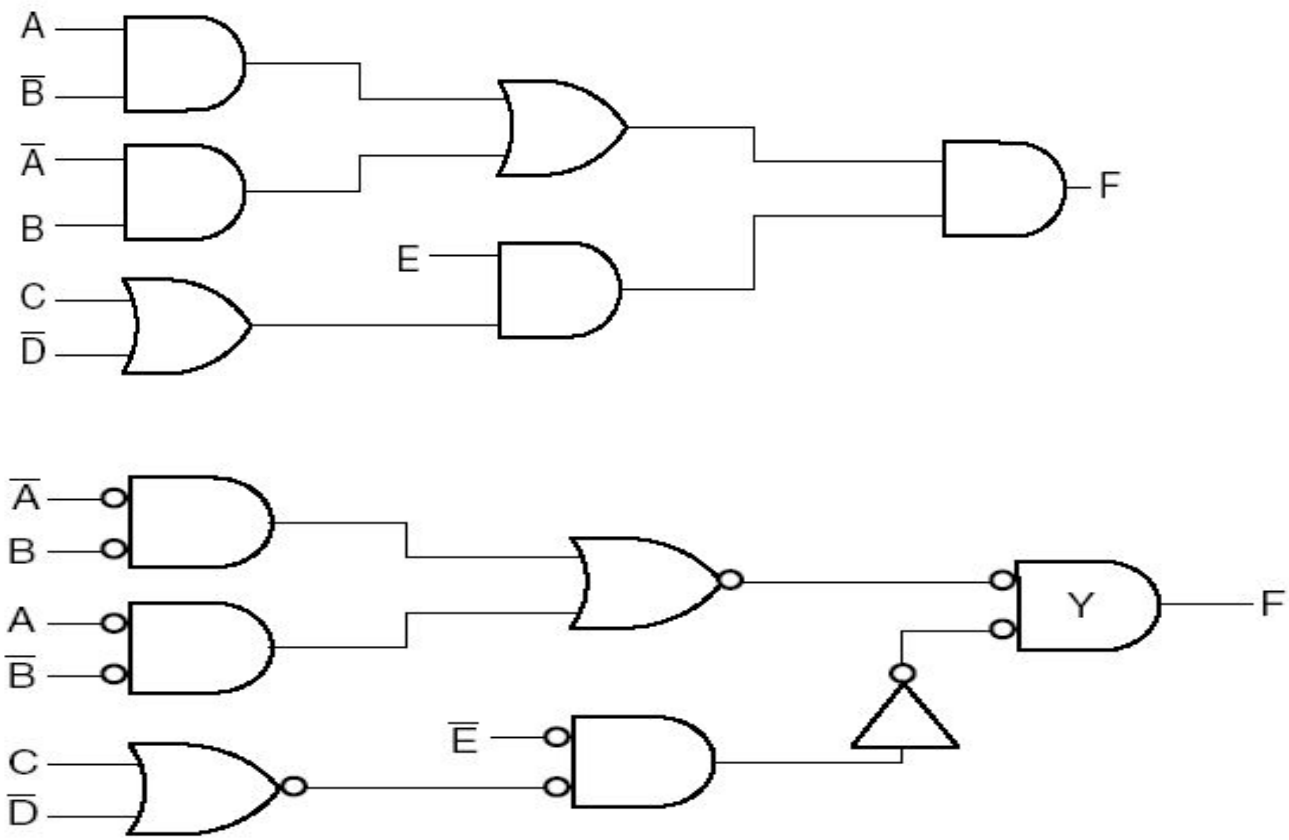


Fig. 2-36 Implementing $F = (A\overline{B} + \overline{A}B)\ E\ (C + \overline{D})$ with NOR Gates

Fig. 2-32 Implementing $F = (A\overline{B} + \overline{A}B)E(C + \overline{D})$

Exclusive-OR Gates

. XOR: $X \oplus Y = XY' + X'Y$

. It can be verified that

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

– $A \oplus B = B \oplus A$ commutative

Odd Function

- The output of an odd function is 1 iff there is an odd no. of input variables equal to 1.
- It can be built using XOR gates.

  e.g. $F = A \oplus B \oplus C \oplus D$ gives a 4-input odd function.

Parity Generation & Checking

e.g. Even parity generator

TABLE 2-9
Truth Table for an Even Parity Generator

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

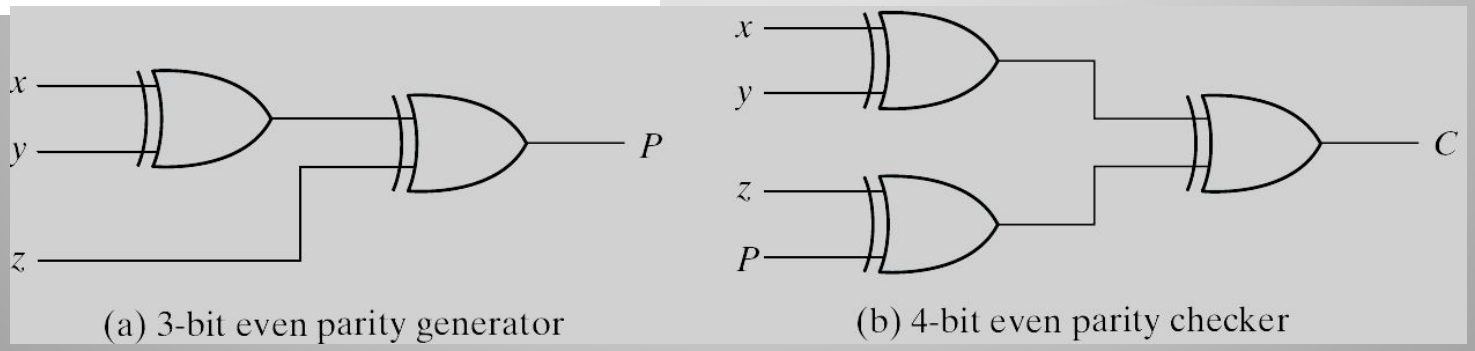
Table 2-9 Truth Table for an Even Parity Generator

Sender generates

$$P = X \oplus Y \oplus Z.$$

Receiver checks parity by

$$C = X \oplus Y \oplus Z \oplus P.$$



(a) 3-bit even parity generator (b) 4-bit even parity checker

Multi-Output Circuit

. Sharing common gates between multiple functions may reduce cost.

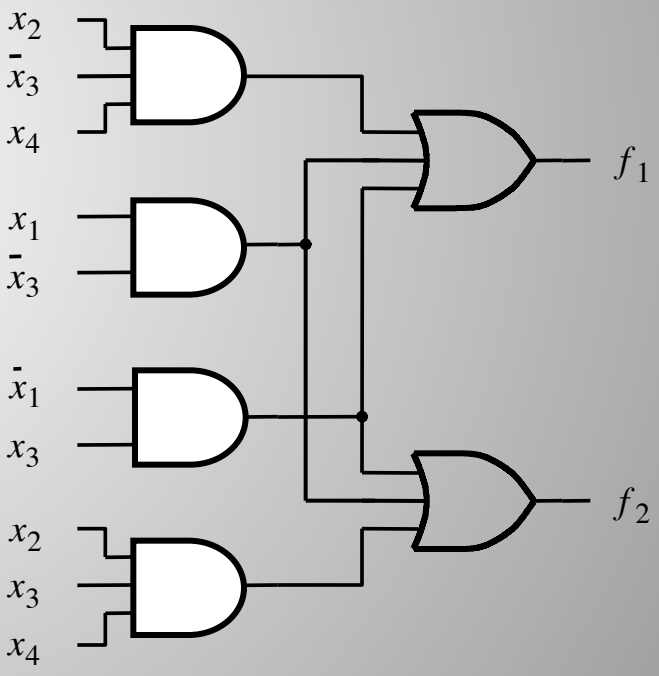
E.g.

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01		1	1	1
11	1	1		
10	1	1		

(a) Function f_1

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01			1	1
11	1	1	1	
10	1	1		

(b) Function f_2



(c) Combined circuit for f_1 and f_2

Multi-Output Circuit Optimization

. Sometimes non-prime implicants may be shared advantageously.

E.g. Let $f_3 = x_1' x_4 + x_2 x_4 + x_1' x_2 x_3$

$f_4 = x_1 x_4 + x_2' x_4 + x_1' x_2 x_3 x_4'$

If optimized independently:

f_3

x_3x_4 \ x_1x_2	00	01	11	10
00				
01	1	1	1	
11	1	1	1	
10		1		

f_4

x_3x_4 \ x_1x_2	00	01	11	10
00				
01	1		1	1
11	1		1	1
10		1		

Multi-Output Circuit Optimization (cont)

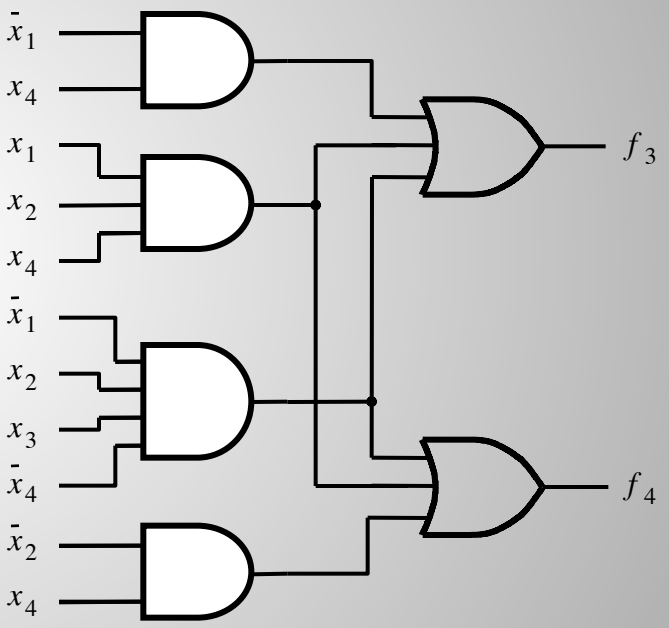
If f_3 and f_4 optimized together by sharing some implicants:

f_3

x_3x_4 \ x_1x_2	00	01	11	10
00				
01	1	1	1	
11	1	1	1	
10		1		

f_4

x_3x_4 \ x_1x_2	00	01	11	10
00				
01	1		1	1
11	1		1	1
10		1		



Combined circuit for f_3 and f_4.

VERILOG DESCRIPTIONS OF COMBINATIONAL LOGIC

4-bit Prime Detector in Verilog Using case

```
module prime(in, isprime);
  input [3:0] in;    // 4-bit input
  output      isprime;  // true if input is prime
  reg         isprime;

  always @(in) begin
    case(in)
   1,2,3,5,7,11,13: isprime = 1'b1;
      default: isprime = 1'b0;
    endcase
  end
endmodule
```

4-bit Prime Detector in Verilog Using casex

```
module prime(in, isprime);
  input [3:0] in;      // 4-bit input
  output      isprime; // true if input is prime
  reg         isprime;

  always @(in) begin
    casex(in)
      4'b0xx1: isprime = 1;
      4'b001x: isprime = 1;
      4'bx011: isprime = 1;
      4'bx101: isprime = 1;
      default: isprime = 0;
    endcase
  end
endmodule
```

4-bit Prime Detector in Verilog Using assign

- Assign can be combined with wire statement.

```
module prime(in, isprime);
  input [3:0] in;    // 4-bit input
  output      isprime; // true if input is prime

  wire isprime = (in[0] & ~in[3]) |
                 (in[1] & ~in[2] & ~in[3]) |
                 (in[0] & ~in[1] & in[2]) |
                 (in[0] & in[1] & ~in[2]);
endmodule
```

Structural Gate-Level Description

```
module prime3(in, isprime);
  input [3:0] in; // 4-bit input
  output      isprime;  // true if input is prime
  wire        a1, a2, a3, a4;

  and and1(a1, in[0], ~in[3]);
  and and2(a2, in[1], ~in[2], ~in[3]);
  and and3(a3, in[0], ~in[1], in[2]);
  and and4(a4, in[0], in[1], ~in[2]);
  or  or1(isprime, a1, a2, a3, a4);
endmodule
```

Test Stimulus (Testbench)

```
module test_prime;
  reg [3:0] in;
  wire isprime;

  // instantiate module to test
  prime p0(in, isprime);

  initial begin
    in = 0;
    repeat (16) begin
      #100
      $display("in = %2d isprime = %1b", in, isprime);
      in = in + 1;
    end
  end
endmodule
```

Style of Testbench

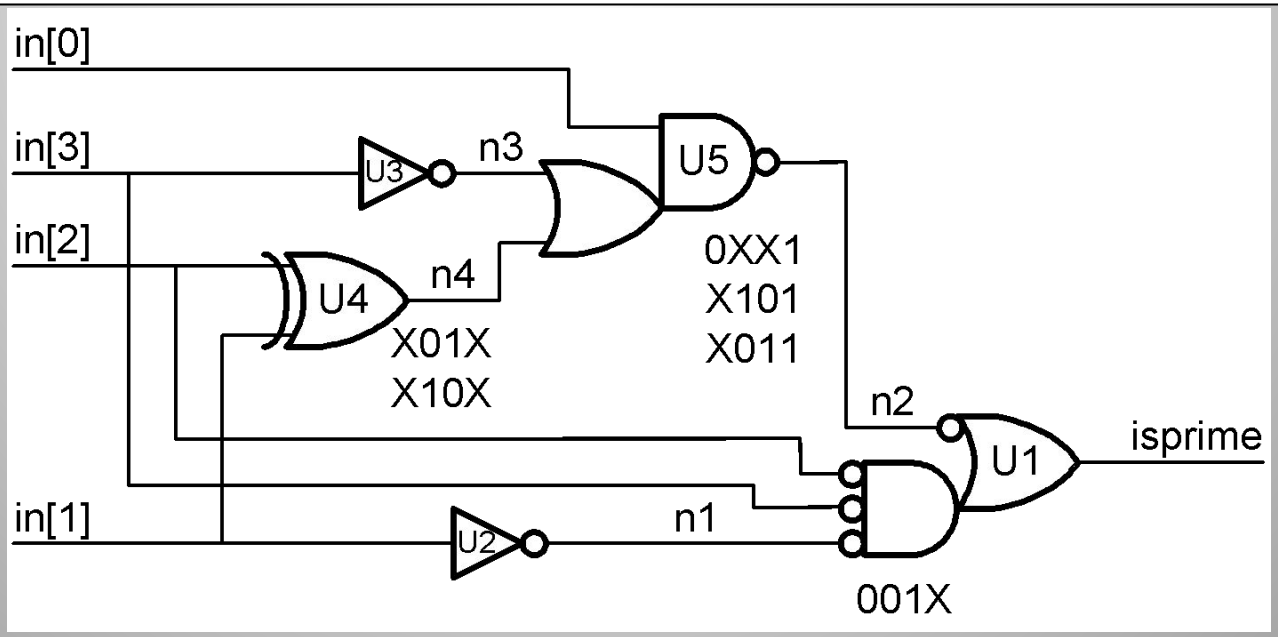
- Different style from describing the synthesizable modules
  - `initial` statements
  - `$display` task
  - `#delay`

Simulation Result

```
# in =  0 isprime = 0
# in =  1 isprime = 1
# in =  2 isprime = 1
# in =  3 isprime = 1
# in =  4 isprime = 0
# in =  5 isprime = 1
# in =  6 isprime = 0
# in =  7 isprime = 1
# in =  8 isprime = 0
# in =  9 isprime = 0
# in = 10 isprime = 0
# in = 11 isprime = 1
# in = 12 isprime = 0
# in = 13 isprime = 1
# in = 14 isprime = 0
# in = 15 isprime = 0
```

Synthesis Result (Gate-Level Structure)

```
module prime ( in, isprime );
input  [3:0] in;
output isprime;
    wire n1, n2, n3, n4;
    OAI13 U1 ( .A1(n2), .B1(n1), .B2(in[2]), .B3(in[3]), .Y(isprime) );
    INV   U2 ( .A(in[1]), .Y(n1) );
    INV   U3 ( .A(in[3]), .Y(n3) );
    XOR2  U4 ( .A(in[2]), .B(in[1]), .Y(n4) );
    OAI12 U5 ( .A1(in[0]), .B1(n3), .B2(n4), .Y(n2) );
endmodule
```



Decimal Prime Detector

```
module prime_dec(in, isprime);
  input [3:0] in;      // 4-bit input
  output      isprime; // true if input is prime
  reg         isprime;

  always @(in) begin
    case(in)
      0,4,6,8,9: isprime = 1’b0;
      1,2,3,5,7: isprime = 1’b1;
      default:   isprime = 1’bx;  // Is ‘default’ effective?
    endcase
  end
endmodule
```

Summary

- To minimize logic
  - Using K-map to find all prime implicants
  - Pick a minimal set of prime implicants that *covers* the function
- Hazards (or glitches) can be eliminated by covering transitions
- Verilog
  - Can describe logic using high-level statements such as case, casex, assign, etc., or logic gates structurally
  - Use the representation readable and maintainable
    - E.g., case for truth tables; assign for equations
    - Avoid doing the logic design yourself
  - Synthesis tool will do the optimization
  - Test benches check that the implementation meets its specification
    - Test benches use a different style of Verilog