CHAPTER-4

Combinational Logic

Digital Design (with an introduction to the Verilog HDL) 6th Edition,
M. Morris Mano, Michael D. Ciletti

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

ECE, IIITDM Kancheepuram

Outline

- Combinational Circuits
- Analysis Procedure
- Design Procedure
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Encoders
- Multiplexer

Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.
- The diagram of a combinational circuit has logic gates with no feedback paths or memory elements.
- A combinationa that can be specified logicall

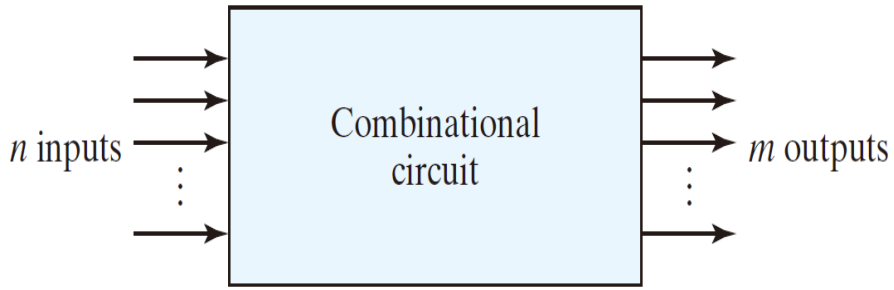


FIGURE 4.1
Block diagram of combinational circuit

Sequential Logic

- In contrast, sequential circuits employ storage elements in addition to logic gates.
- Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs.
- The circuit behavior must be specified by a time sequence of inputs and internal states.

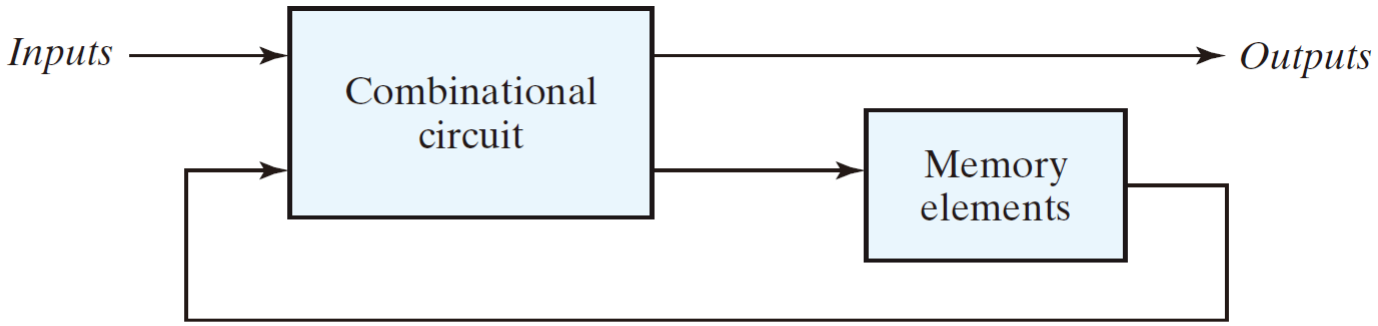


FIGURE 5.1
Block diagram of sequential circuit

Combinational Circuits

- A combinational circuit consists of
  - Input variables
  - Logic gates
  - Output variables

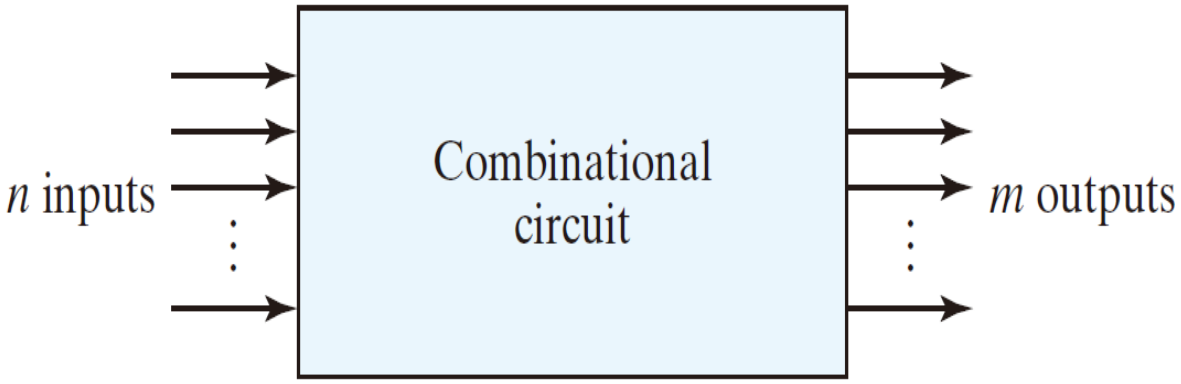


FIGURE 4.1

Block diagram of combinational circuit

Combinational Circuits

- Each input and output variable is a binary signal
  - Represent logic 1 and logic 0
- There are 2^n possible binary input combinations for n input variable
- Only one possible output value for each possible input combination
- Can be specified with a truth table
- Can also be described by m Boolean functions, one for each output variable
  - Each output function is expressed in terms of n input variables

Outline

- Combinational Circuits
- **Analysis Procedure**
- Design Procedure
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Encoders
- Multiplexer

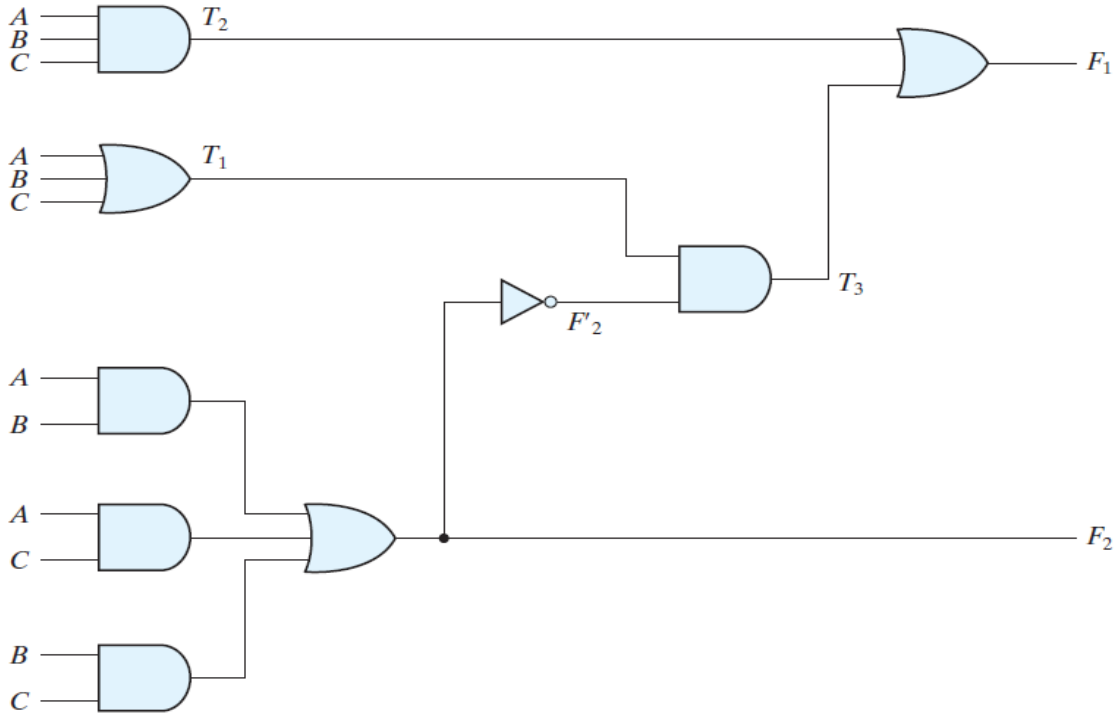
Analysis Procedure

- Analysis: determine the function that the circuit implements
  - Often start with a given logic diagram
- The analysis can be performed by
  - Manually finding Boolean functions
  - Manually finding truth table
  - Using a computer simulation program
- First step: make sure that circuit is combinational
  - Without feedback paths or memory elements
- Second step: obtain the output Boolean functions or the truth table

Analysis Procedure

Step 1:

- Label all gate outputs that are a function of input variables
- Determine Boolean functions for each gate output

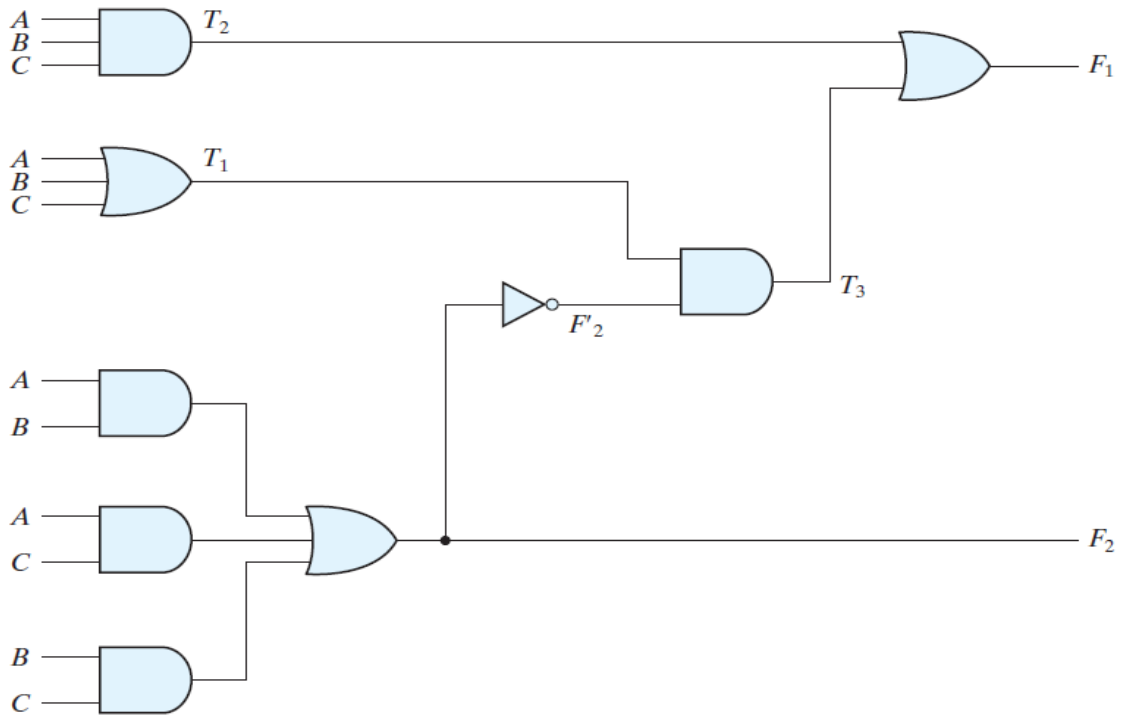


$$F_2 = AB + AC + BC$$
$$T_1 = A + B + C$$
$$T_2 = ABC$$

Analysis Procedure

Step 2:

- Label the gates that are a function of input variables and previously labeled gates
- Find the Boolean function for these gates



$$T_3 = F'_2 T_1$$

$$F_1 = T_3 + T_2$$

Analysis Procedure

Step 3:

- Obtain the output Boolean function in term of input variables
- By repeated substitution of previously defined functions

$$
\begin{aligned}
F_1 &= T_3 + T_2 = F'_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\
&= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\
&= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\
&= A'BC' + A'B'C + AB'C' + ABC
\end{aligned}
$$

Step 4:

- Build up the truth table

  As we know ahead F_1, $F_1 = \Sigma(1, 2, 4, 7)$, we can verify the function by truth table

Analysis Procedure

- To obtain the **truth table** from the logic diagram:

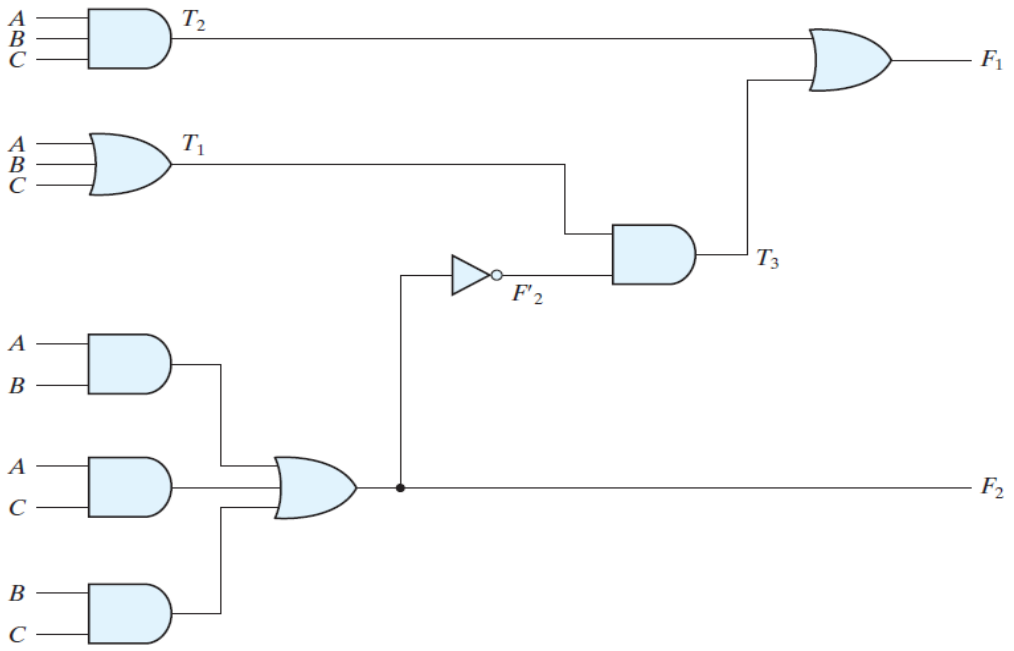
1. Determine the number of input variables For n inputs:
   - 2^n possible combinations
   - List the binary numbers from 0 to (2^n -1) in a table
2. Label the outputs of selected gates
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only
4. Obtain the truth table for those gates that are a function of previously defined variables at step 3
   - Repeatedly until all outputs are determined

Analysis Procedure

- **Realize the truth table "level-by-level" for verification.**

Table 4.1
Truth Table for the Logic Diagram of Fig. 4.2

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1



Outline

- Combinational Circuits
- Analysis Procedure
- **Design Procedure**
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Encoders
- Multiplexer

Design Procedure

- Design procedure:
  - start from the specification of the problem and
  - culminates in a logic circuit diagram or a set of Boolean functions
- Step 1: determine the required number of inputs and outputs from the specification
- Step 2: derive the truth table that defines the required relationship between inputs and outputs
- Step 3: obtain the simplified Boolean function for each output as a function of the input variables
- Step 4: draw the logic diagram and verify the correctness of the design

Code Conversion Example

Example: Build up a logic circuit to convert a BCD code to an Excess-3 code

- The 6 input combinations not listed are don't cares
- These values have no meaning in BCD
- We can arbitrary assign them to 1 or 0

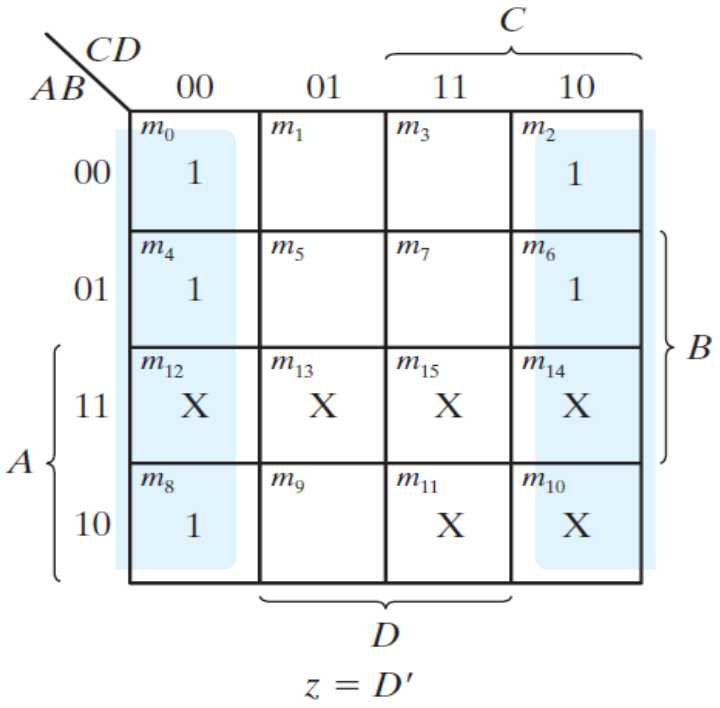
Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Note: Excess 3 code is the code with "additional" 3

Maps for Code Converter

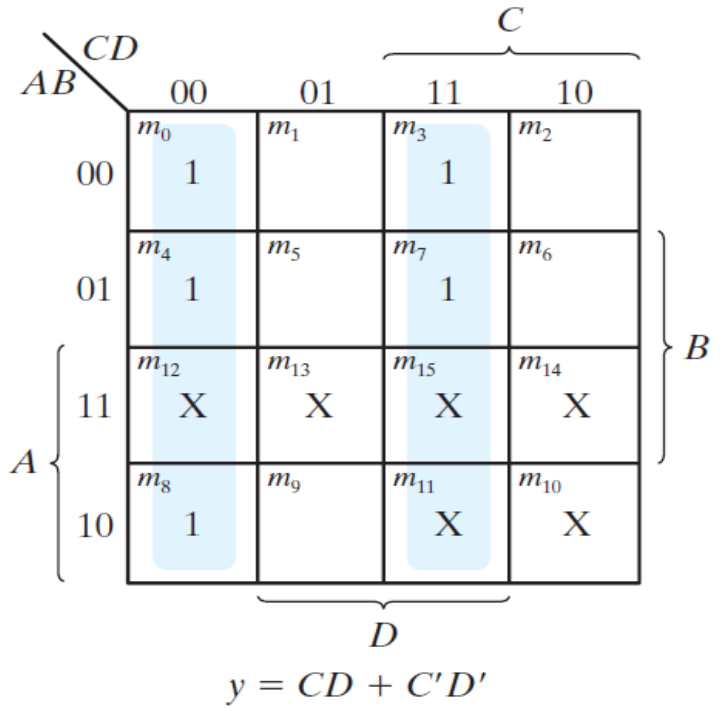
- **Step 1: Find inputs and outputs and the truth table.**
- **Step 2: Making a K-map for the inputs and outputs.**
- **Step 3: Find the logic expressions from the K-map.**

Z



$z = D'$

y



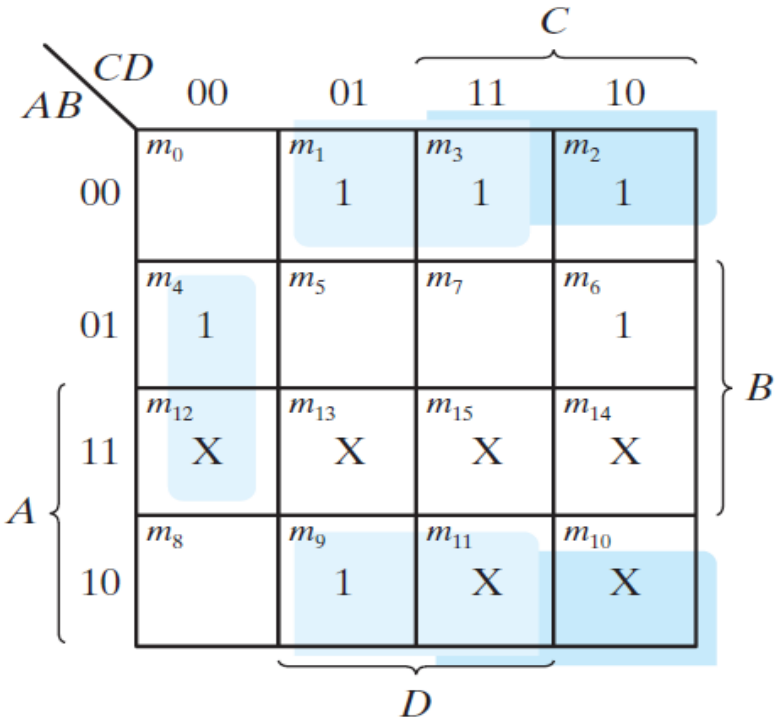
$y = CD + C'D'$

A	B	C	D	y	z
0	0	0	0	1	1
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	0	0

- The six don't care minterms (10~15) are marked with X

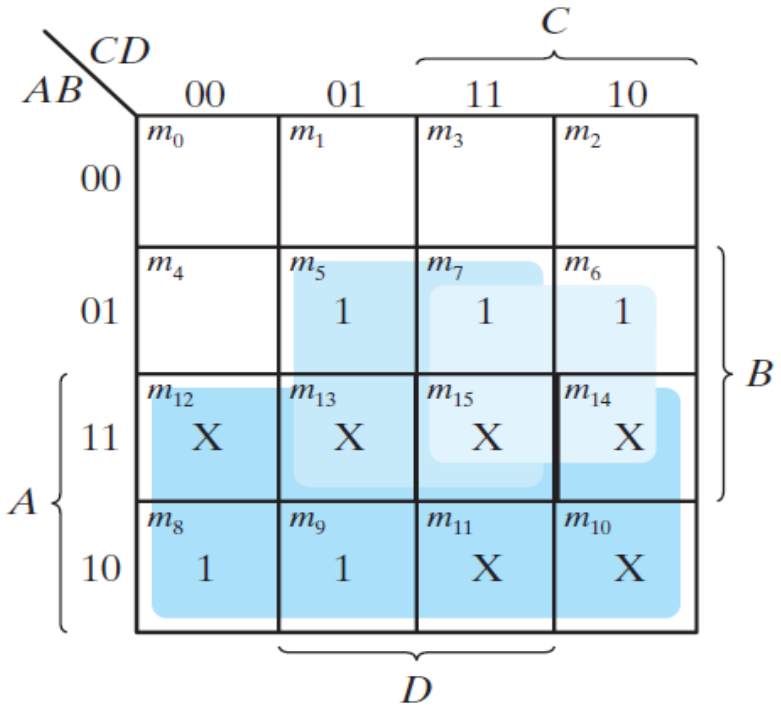
Maps for Code Converter

x



$$x = B'C + B'D + BC'D'$$

w



$$w = A + BC + BD$$

w	x
0	0
0	1
0	1
0	1
0	1
1	0
1	0
1	0
1	0
1	1

Logic Diagram for the Converter

- There are various possibilities for a logic diagram that implements a circuit
- A two-level logic diagram may be obtained directly from the Boolean expressions derived by the maps
- The expressions may be manipulated algebraically to use common gates for two or more outputs
  - Reduce the number of gates used

Simplified function

$z = D'$

$y = CD + C'D'$

$x = B'C + B'D + BC'D'$

$w = A + BC + BD$

Another implementation

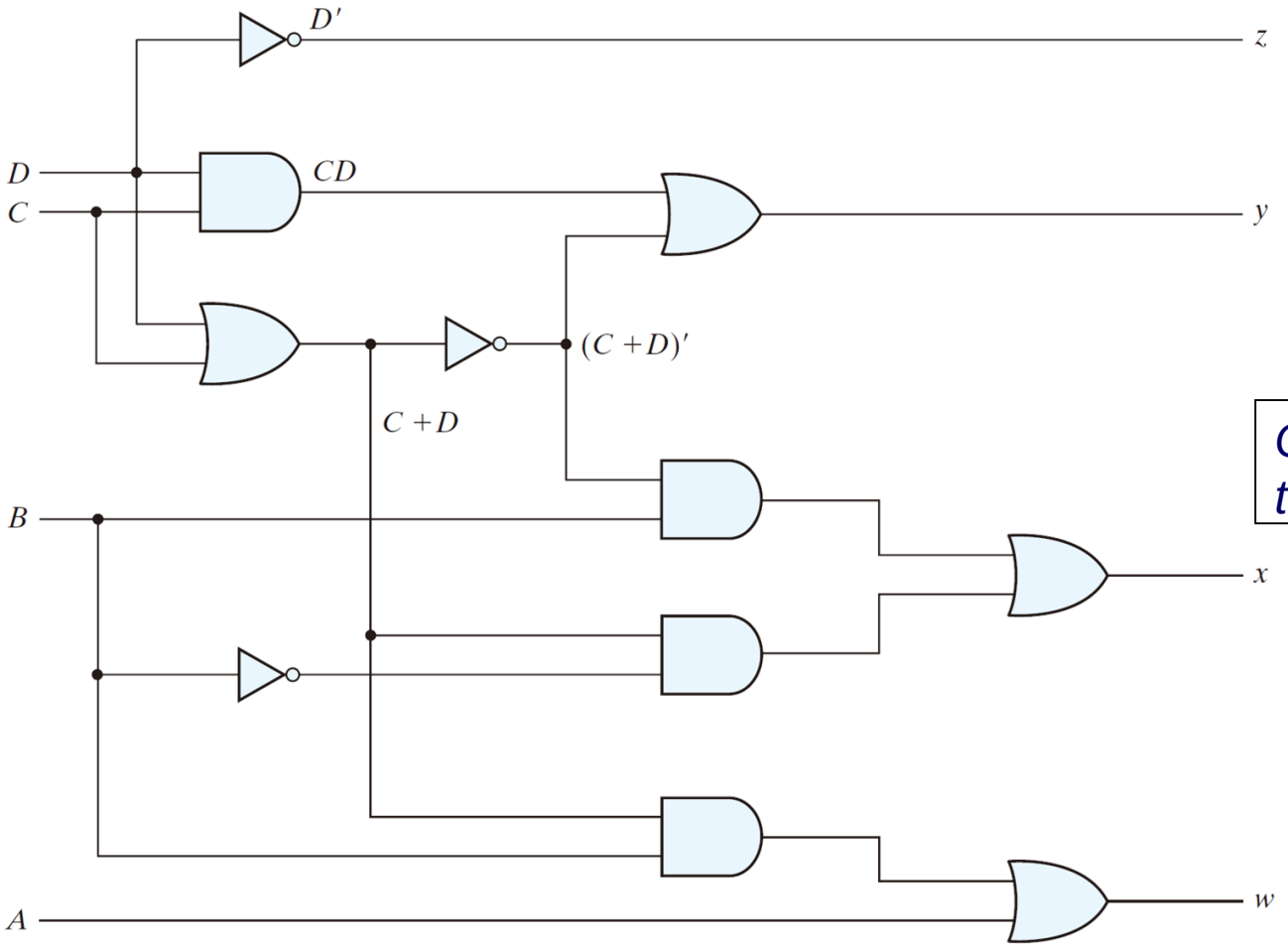
$z = D'$

$y = CD + C'D' = CD + (C+D)'$

$x = B'C + B'D + BC'D' = B'(C+D) + B(C+D)'$

$w = A + BC + BD = A + B(C+D)$

Logic Diagram for the Converter



z = D'

y = CD +C'D' = CD + (C+D)'

x = B'C + B'D+BC'D' = B'(C+D) +B(C+D)'

w = A+BC+BD = A+B(C+D)

C + D is commonly used to implement the three outputs

FIGURE 4.4
Logic diagram for BCD-to-excess-3 code converter

Outline

- Combinational Circuits
- Analysis Procedure
- Design Procedure
- **Binary Adder-Subtractor**
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders
- Encoders
- Multiplexer

Adder

- The most basic arithmetic operation is the addition of two binary digits
  - When both augend and addend bits are equal to 1, the binary sum consists of two digits (1 + 1 = 10)

    0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 0 = 1 ; 1 + 1 = 10

    Carry
  - The higher significant bit of this result is called a carry
- A combination circuit that performs the addition of two bits is half adder
- A adder performs the addition of 2 significant bits and a previous carry is called a full adder

Half Adder

- **A combinational circuit that performs the addition of two bits is called a half adder.**
  - two input variables: x, y
  - two output variables: C (carry), S (sum)
  - truth table

Table 4.3
Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

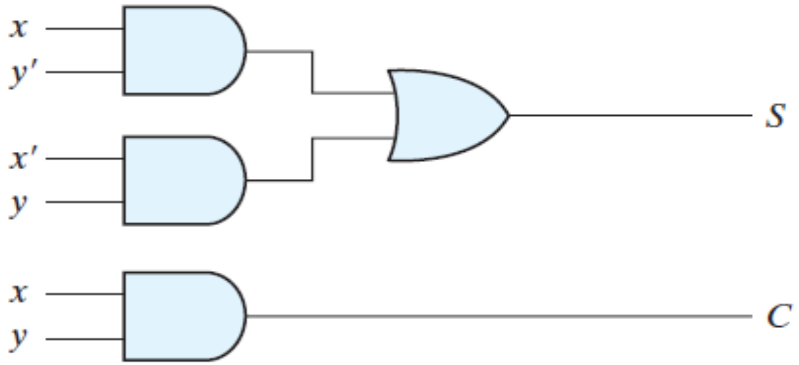
Half Adder

- **Circuit implementation of a half adder**

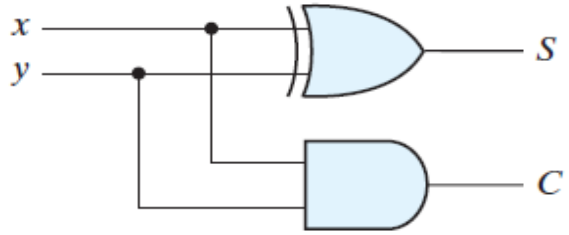
Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = x'y + xy'$$
$$C = xy$$



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

FIGURE 4.5
Implementation of half adder

Full Adder

- The arithmetic sum of three input bits
- Three input bits
  - x, y: two significant bits
  - z: the carry bit from the previous lower significant bit
- Two output bits: C, S

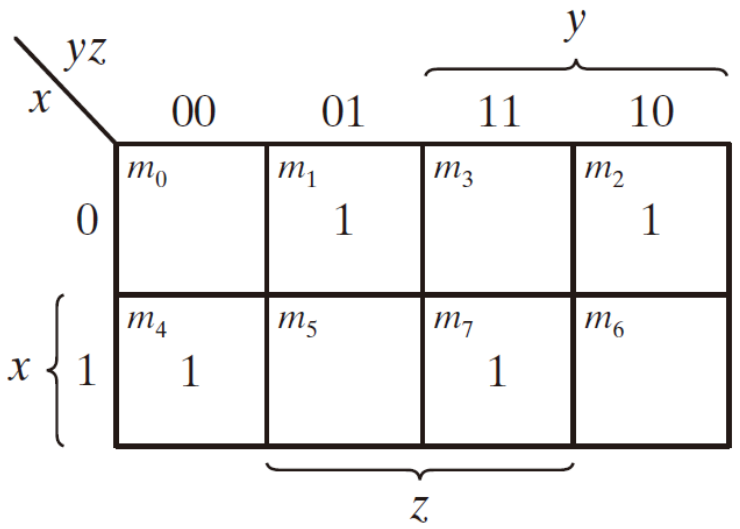
Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

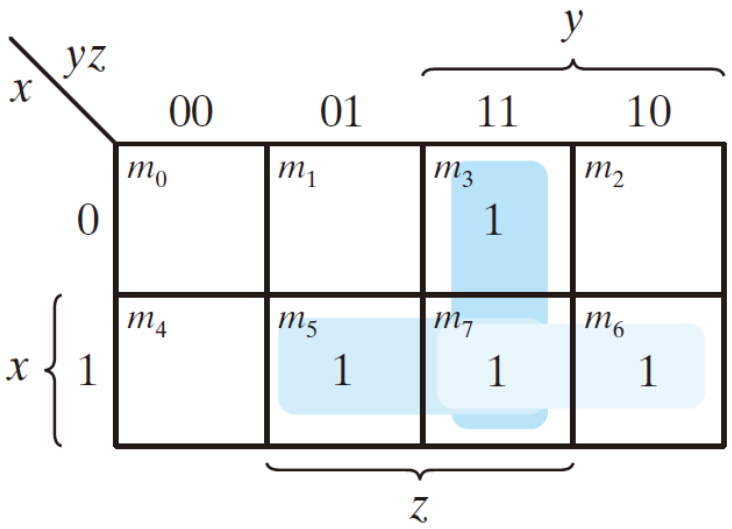
Full Adder

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



(a) $S = x'y'z + x'yz' + xy'z' + xyz$



(b) $C = xy + xz + yz$

FIGURE 4.6
K-Maps for full adder

Full Adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

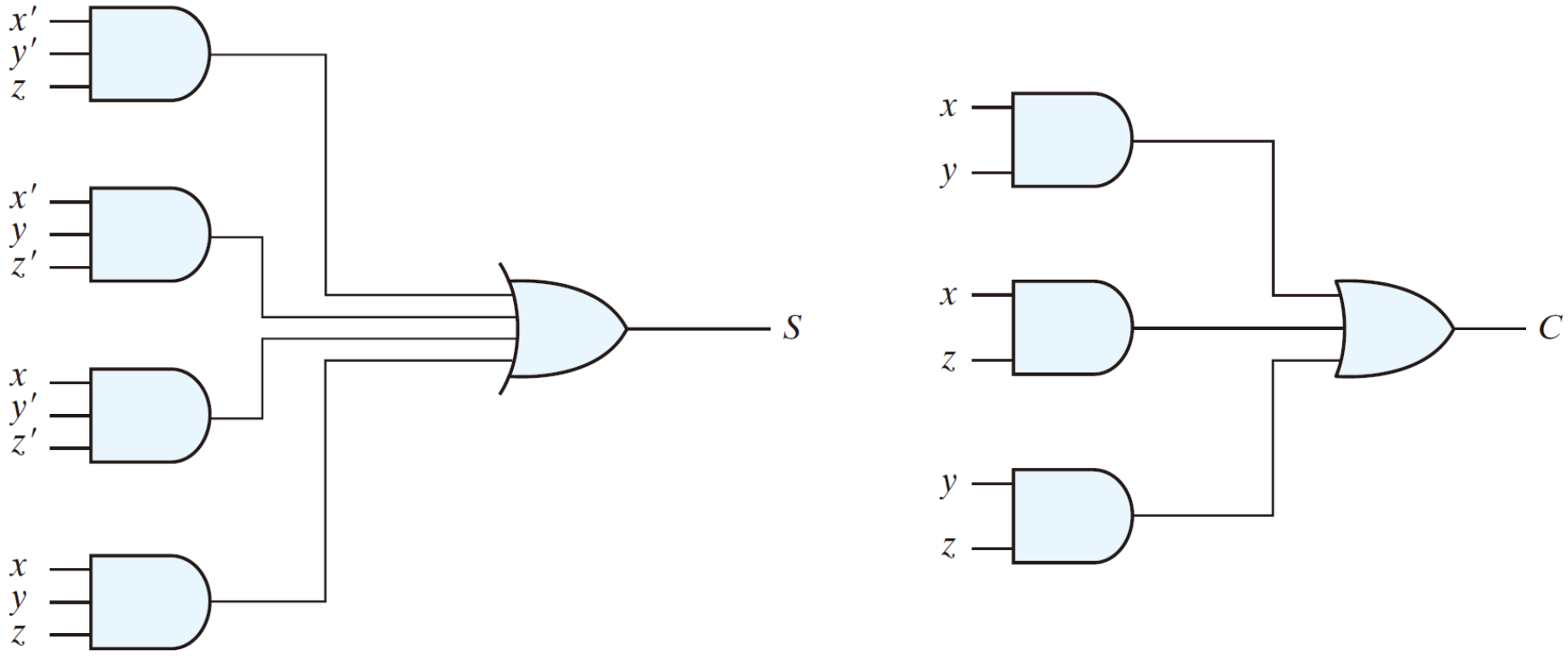


FIGURE 4.7
Implementation of full adder in sum-of-products form

Full Adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$= z'(x'y + xy') + z(x'y' + xy)$$
$$= z'(x'y + xy') + z[(x'y' + x)(x'y' + y)]$$
$$= z'(x'y + xy') + z[(x' + x)(y' + x)(x' + y)(y' + y)]$$
$$= z'(x'y + xy') + z[(yx')'(xy')']$$
$$= z'(x'y + xy') + z(yx' + xy')'$$
$$S = z \oplus (x'y + xy') = z \oplus (x \oplus y)$$

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= z'(xy' + x'y) + z(xy + x'y')$$
$$= xy'z' + x'yz' + xyz + x'y'z$$

Full Adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

Make a full adder using 2 half adders and one OR gate

$$S = z \oplus (x \oplus y)$$

Different?

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

$$C = z(x \oplus y) + xy$$

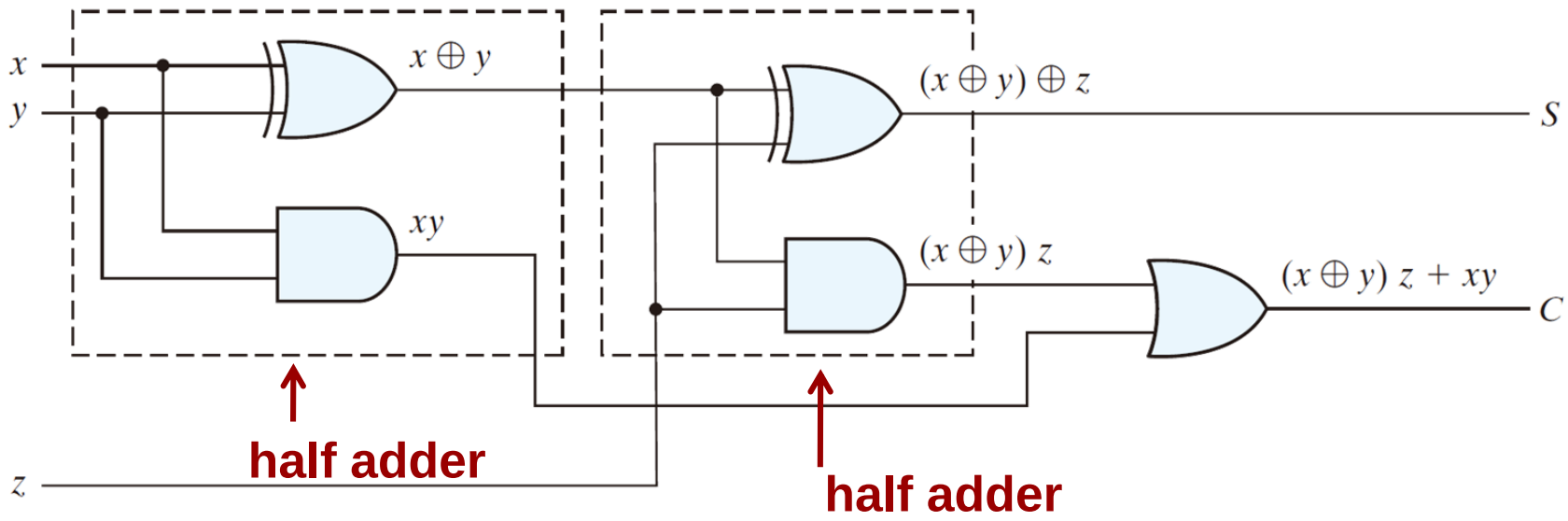
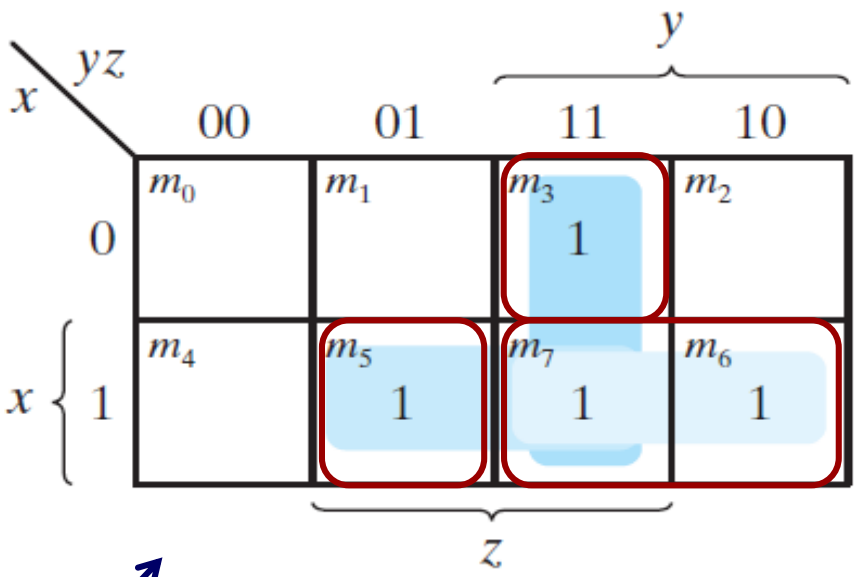


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

Full Adder

Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

$$C = xy + xz + yz$$

Full Adder

- **A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.**
- Can be constructed with full adders connected in cascade
  - The output carry from each full adder is connected to input carry of the next full adder in the chain
  - n-bit binary ripple carry adder is connected by n FAs

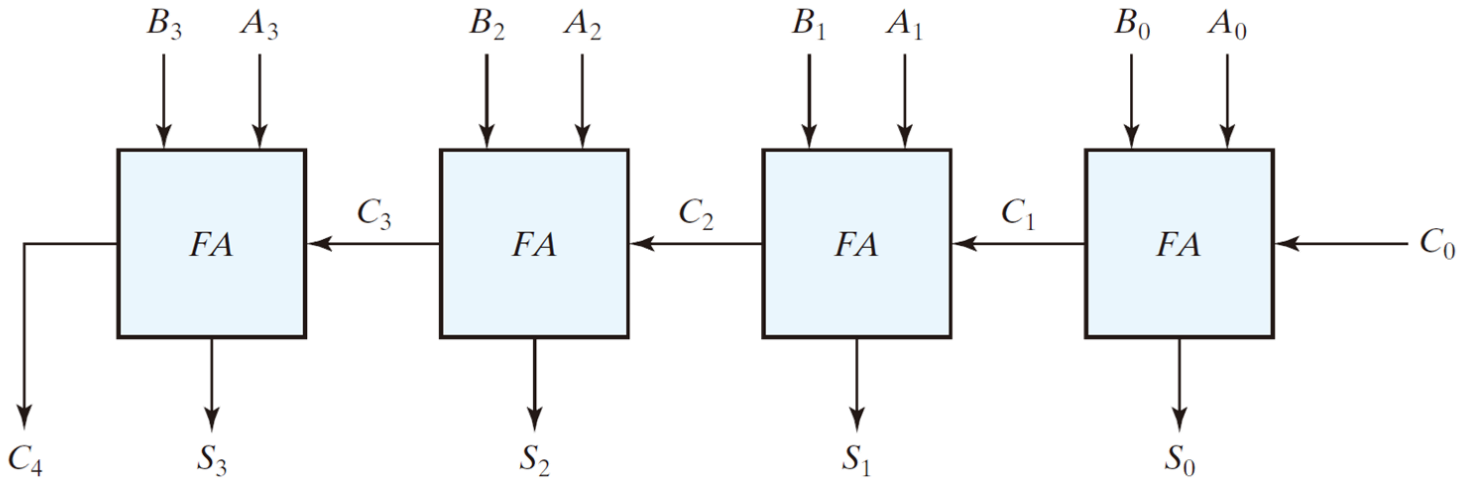


FIGURE 4.9
Four-bit adder

Full Adder

- Consider two binary number A = 1011 and B = 0011

Example:

4 bit binary adder

Subscript *i*:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

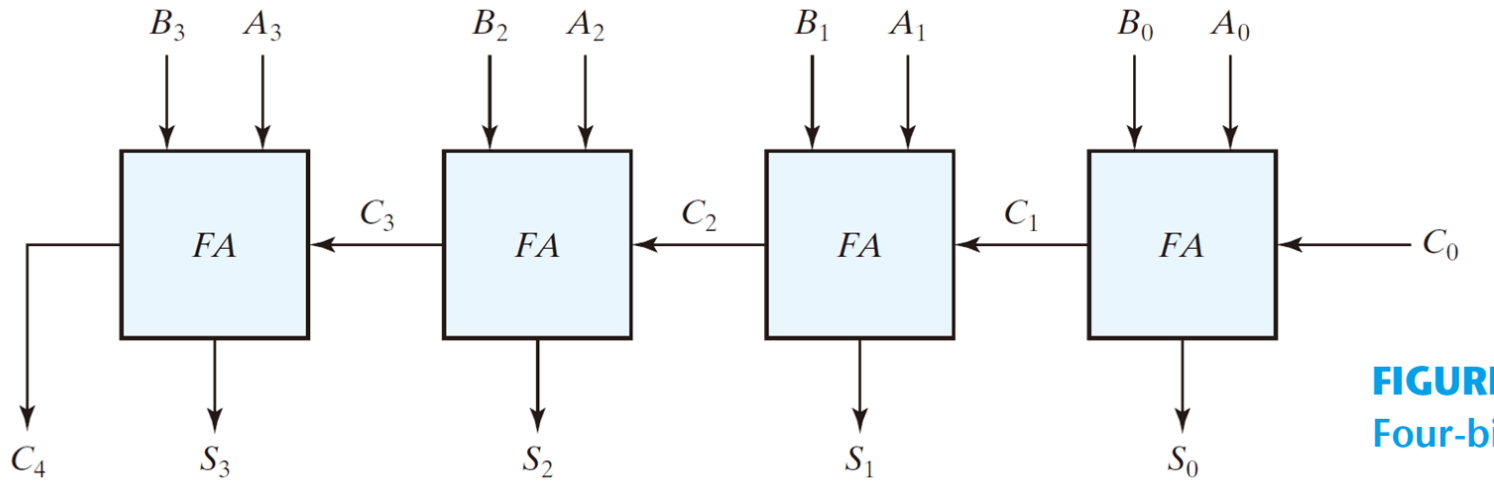
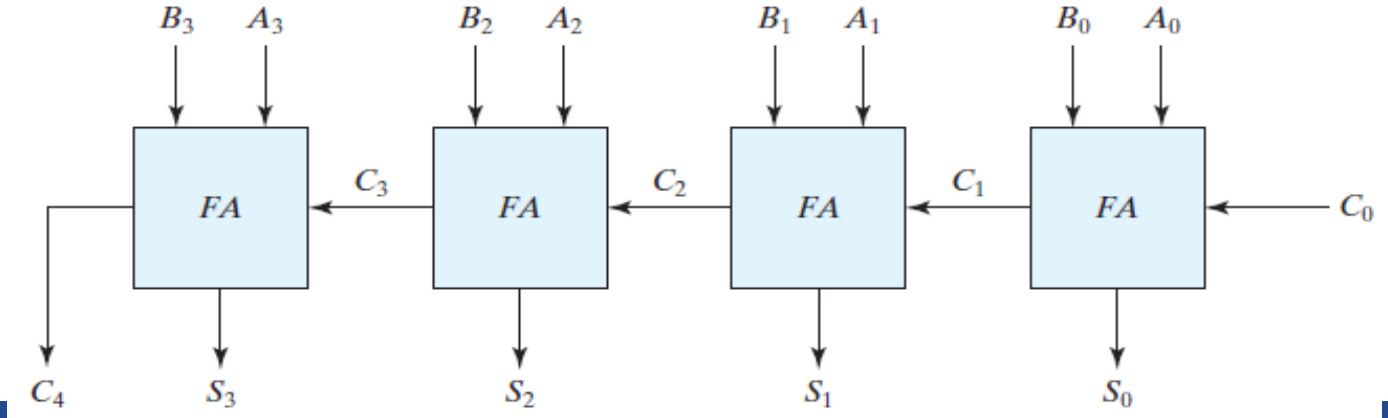


FIGURE 4.9
Four-bit adder

Carry Propagation

- As in any combinational circuit, the signal must propagate through the gates before the output is available.
- The total propagation time is equal to the propagation delay of a typical gate times the number of levels in the circuit.
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders.
- Each bit of the sum output depends on the value of the input carry
  - The value of S_i will be in final value only after the input carry C_i has been propagated



Full Adder with P and G

- The full adder can be redrawn with two internal signals P (propagation) and G (generation)
- The signal from input carry C_i to output carry C_{i+1} propagates through an AND and a OR gate (2 gate levels)
- For n-bit adder, there are 2n gate levels for the carry to propagate from input to output

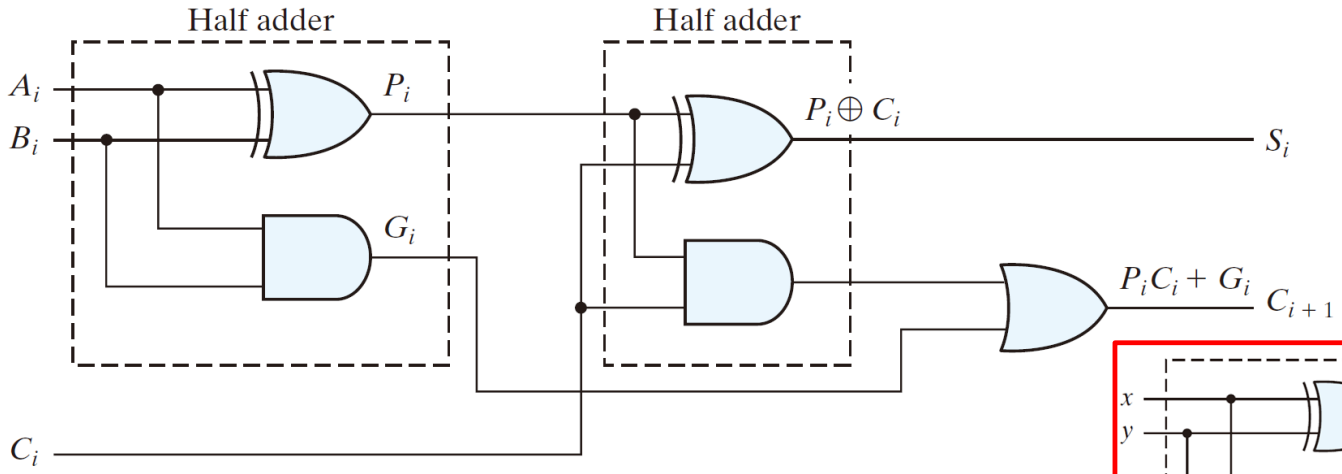
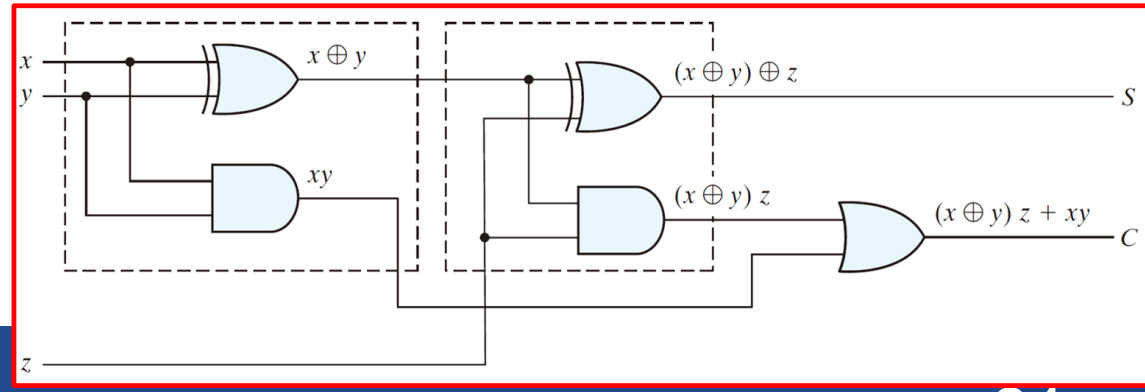


FIGURE 4.10
Full adder with *P* and *G* shown



Carry Propagation

- The carry propagation time is a limiting factor on the speed with which two numbers are added
- All other arithmetic operations are implemented by successive additions
  - The time consumed during the addition is very critical
- To reduce the carry propagation delay
  - Employ faster gates with reduced delays
  - Increase the equipment complexity
- Several techniques for reducing the carry propagation time in a parallel adder
  - The most widely used technique employs the principle of carry lookahead

Carry Propagation and Generation

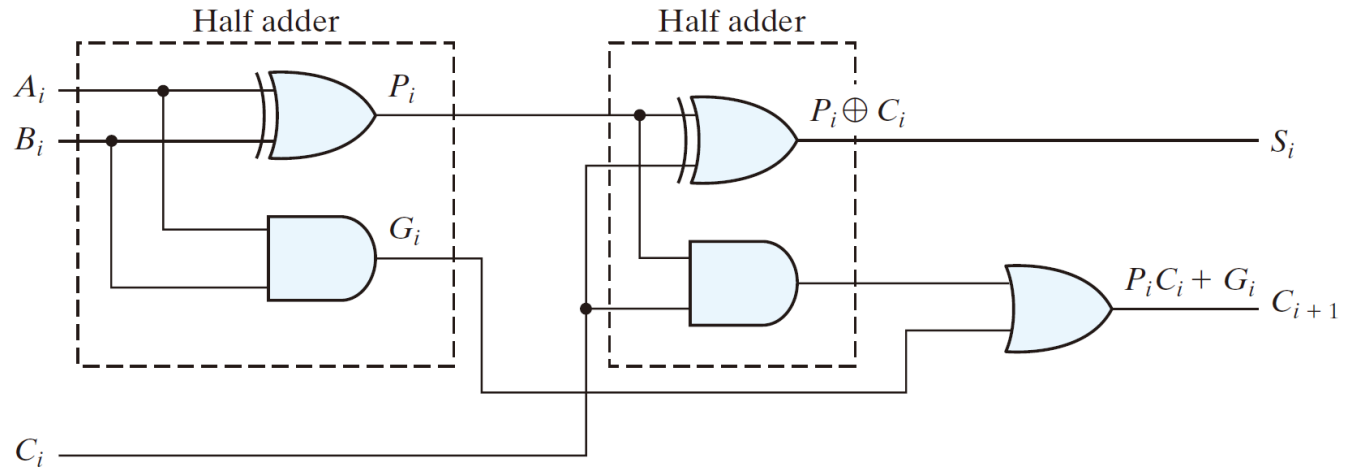


FIGURE 4.10
Full adder with *P* and *G* shown

$$P_i = A_i \oplus B_i \qquad S_i = P_i \oplus C_i$$
$$G_i = A_i B_i \qquad C_{i+1} = G_i + P_i C_i$$

- **G_i, called "carry generate", produces a carry 1 when A_i and B_i are 1.**
- **P_i is called a "carry propagate", because it determines whether a carry into stage *i* will propagate into stage *i* + 1.**

Carry Lookahead Logic

$$P_i = A_i \oplus B_i \qquad S_i = P_i \oplus C_i$$
$$G_i = A_i B_i \qquad C_{i+1} = G_i + P_i C_i$$

C_0 = input carry

$$C_1 = G_0 + P_0 C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 C_1 \\ &= G_1 + P_1 (G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$

$$\begin{aligned} C_3 &= G_2 + P_2 C_2 \\ &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{aligned}$$

C_3 is propagated at the same time as C_1 and C_2.

- **Use "carry lookahead logic" to reduce the propagation.**

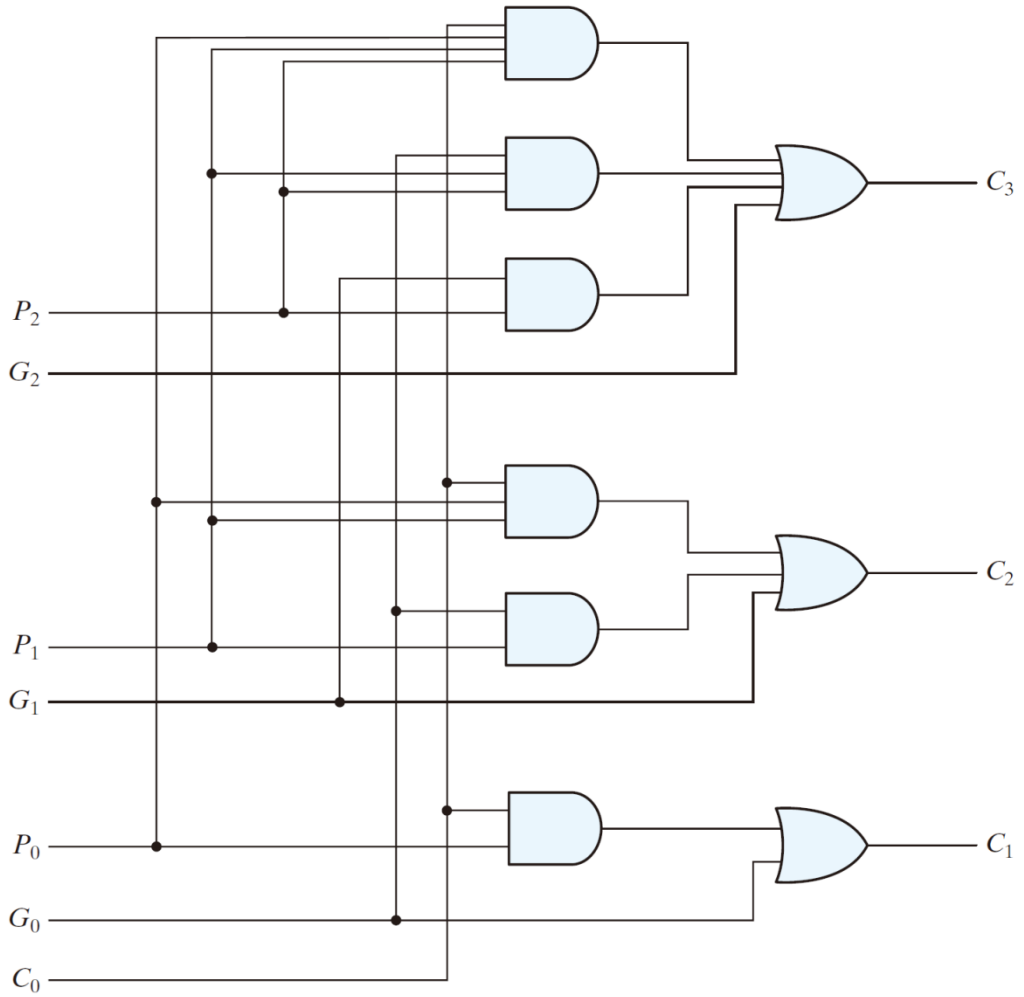
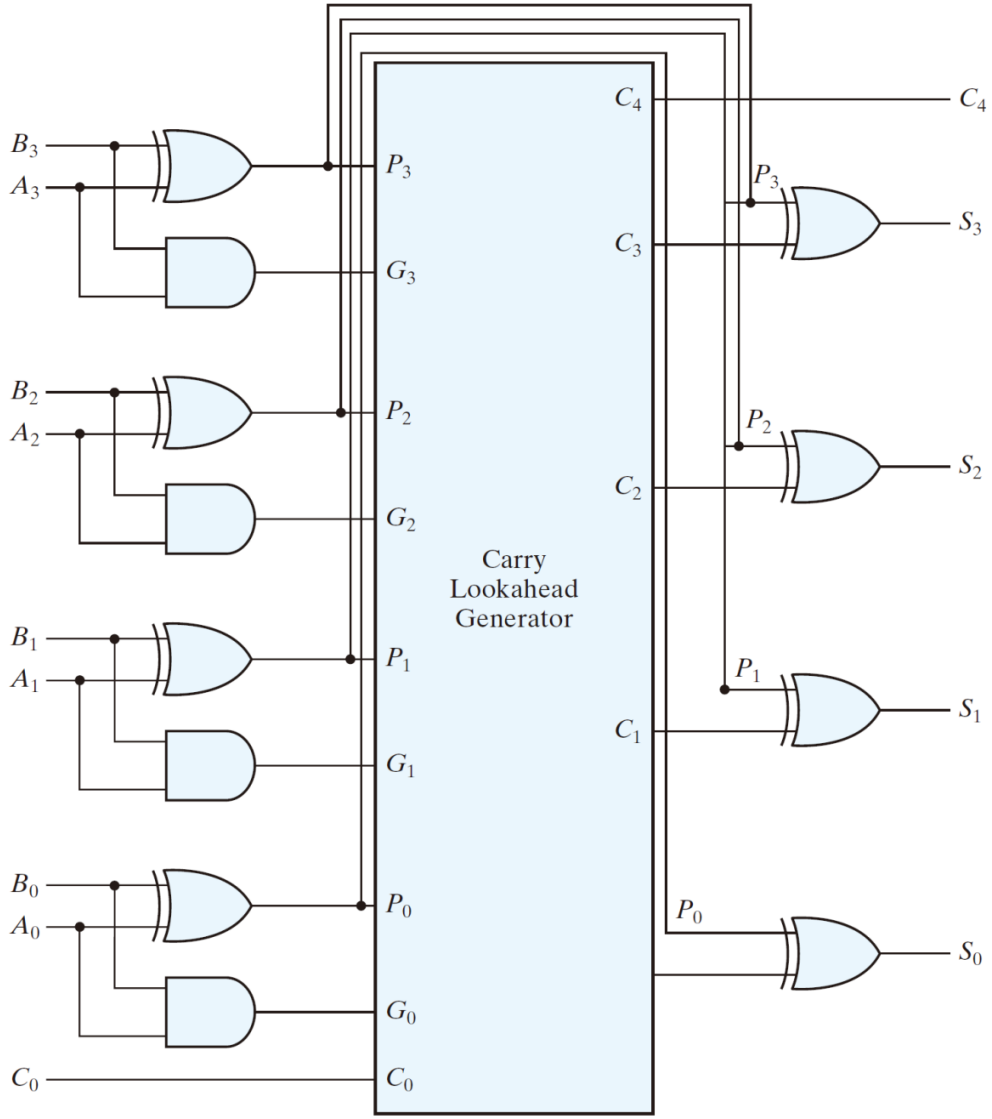


FIGURE 4.11
Logic diagram of carry lookahead generator

Four-bit Adder with Carry Lookahead



$$P_i = A_i \oplus B_i$$
$$S_i = P_i \oplus C_i$$

Output S1 to S3 can have equal propagation delay times

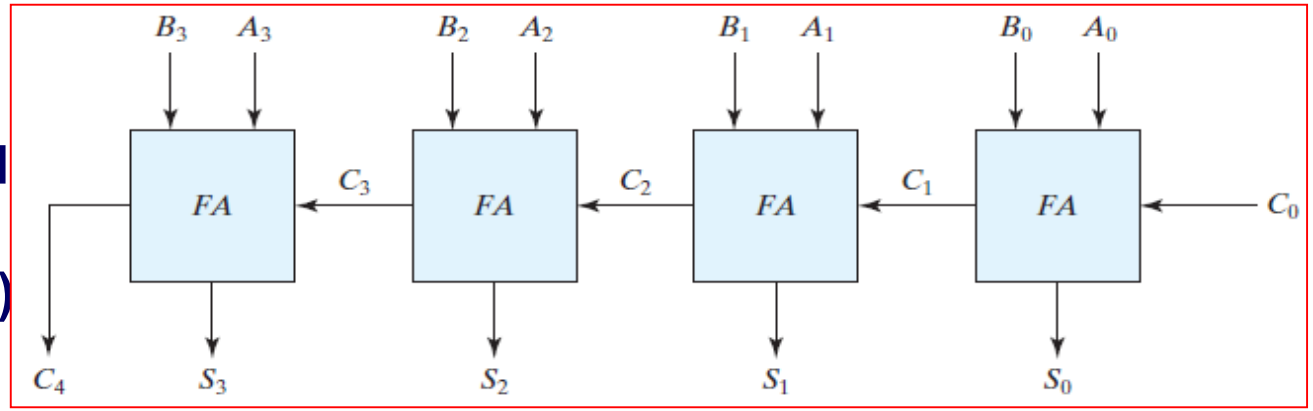
FIGURE 4.12
Four-bit adder with carry lookahead

Binary Subtractor

- The subtraction *A* − *B* can be done by taking the 2's complement of *B* and adding it to *A* .
  - The 2's complement can be obtained by taking the 1's complement and adding 1.
  - A - B = A + (B′ + 1)
- The 1's complement can be implemented with inverters.

Example: Find 2's complement of 1011001

	01001100	1's complement (inverted)
+	1	
	01001101	2's complement



- Place inverters between each data input *B* and the corresponding input of the full adder.
- $C_0 = 1$

4-bit Adder-subtractor

- A – B = A + (2's complement of B)
- M=0 (Adder) [M=0, A+B]
  - Input of FA is A and B (B ⊕ 0 = B), and C_0 is 0
- M=1 (Subtractor) [M=1, A+B′+1]
  - Input of FA is A and B′ (B ⊕ 1 = B′), and C_0 is 1

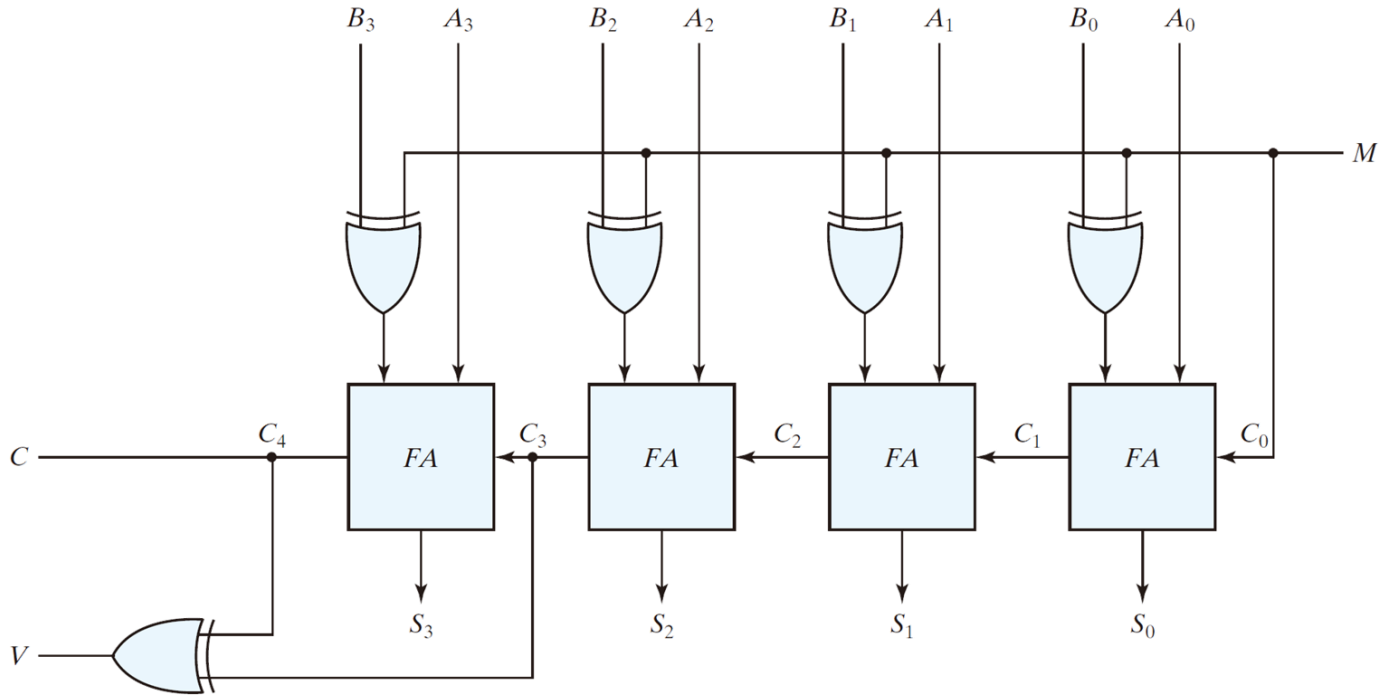


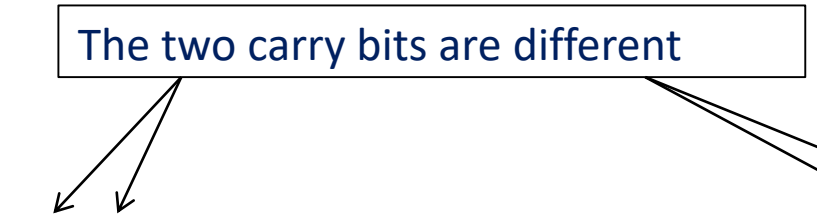
FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

M-mode input

Overflow

- An overflow occurs when two number of n digits each are added and the sum occupies n+1 digits
- An overflow can only occur when two numbers added are both positive or both negative

Example:



carries:	0 1		carries:	1 0	
+70	0 1000110		−70	1 0111010	
+80	0 1010000		−80	1 0110000	
+150	1 0010110	(−106)	−150	0 1101010	(+106)
(010010110)			(101101010)		

Overflow detection

- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position
  - If these two carries are not equal, and overflow has occurred
  - If the output V is equal to 1, an overflow is detected

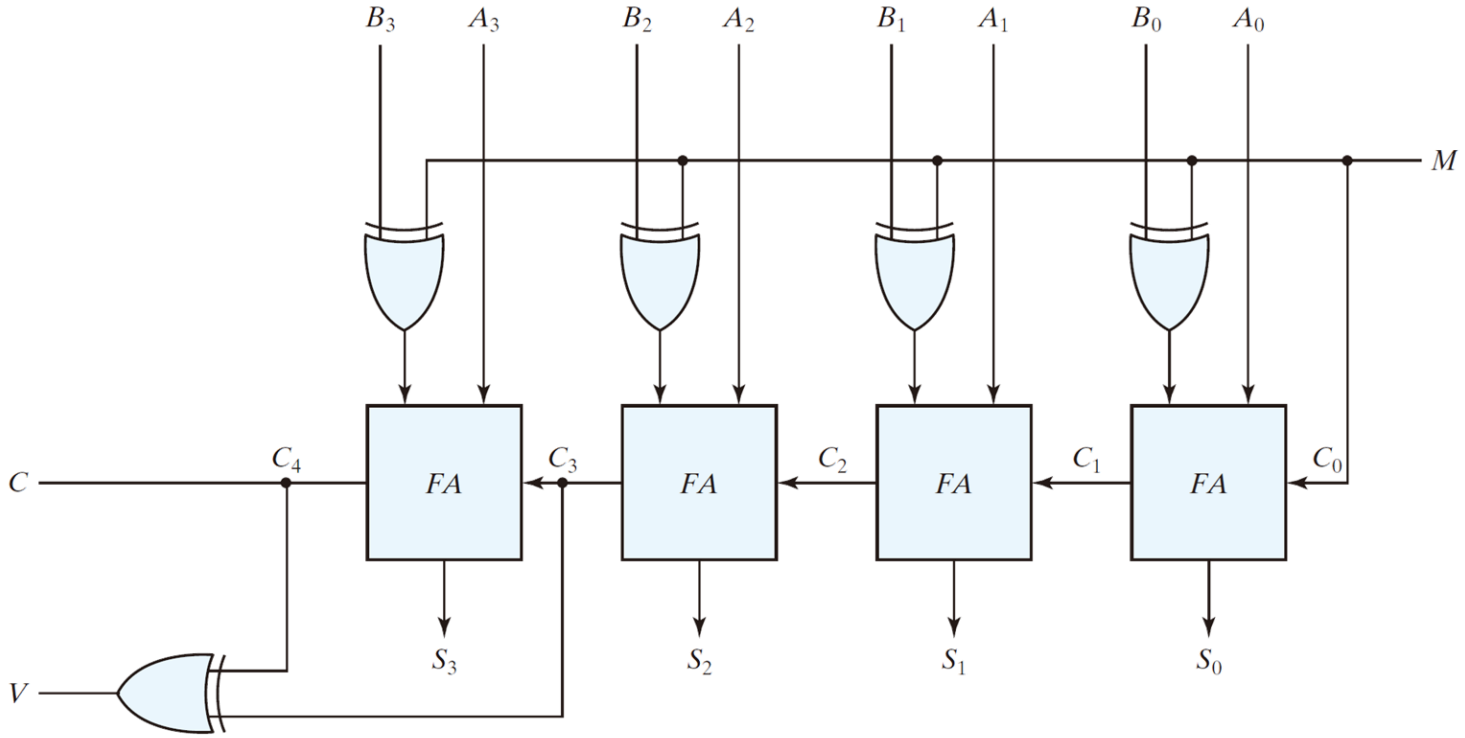


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

Adder-subtractor circuit

- Unsigned
  - C bit detects a carry after addition or a borrow after subtraction
- Signed
  - V bit detects an overflow

    0: no overflow; n -bit result is correct

    1: overflow; result of the operation contains n + 1 bits

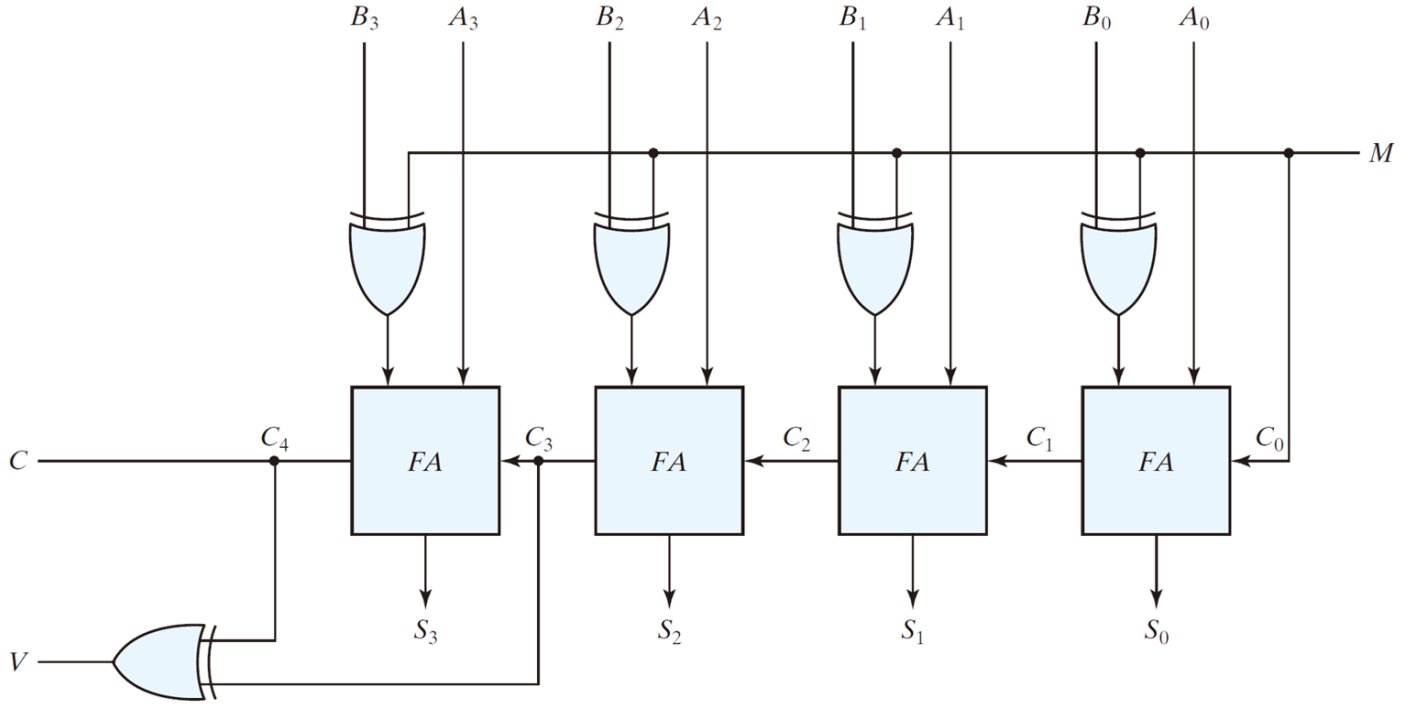


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

Decimal Adder

- A decimal adder requires a minimum of 9 inputs and 5 outputs
  - 1 digit requires 4-bit
  - Input: 2 digits + 1-bit carry
  - Output: 1 digit + 1-bit carry

BCD Adder

- How do we implement a BCD adder? Let's see the relationship between a binary sum to a BCD sum (the table in the next page).
- The maximum value for a BCD sum will be 9 + 9 + 1, while 1 is a carry from the previous bit.
- Remember: If the sum is 10, 11, 12, ..19. We need to add binary 6 (i.e. 0110) to the binary sum in order to obtain a correct BCD code.

BCD Adder

Binary Sum						BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1	
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0		0	0	1	0	0	4
0	0	1	0	1	same	0	0	1	0	1	5
0	0	1	1	0		0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
0	1	0	1	0		1	0	0	0	0	10
0	1	0	1	1		1	0	0	0	1	11
0	1	1	0	0		1	0	0	1	0	12
0	1	1	0	1	Δ = 6	1	0	0	1	1	13
0	1	1	1	0		1	0	1	0	0	14
0	1	1	1	1		1	0	1	0	1	15
1	0	0	0	0		1	0	1	1	0	16
1	0	0	0	1		1	0	1	1	1	17
1	0	0	1	0		1	1	0	0	0	18
1	0	0	1	1		1	1	0	0	1	19

- When the binary sum is equal to or less than 1001_b
  - BCD Sum = Binary Sum
  - C = 0
- When the binary sum is greater than 1001_b
  - BCD Sum = Binary Sum + 0110_b
  - C = 1
- If binary sum $Z_8Z_4Z_2Z_1 > 9$, we should add 0110 to the sum, set $C = 1$, and get the right BCD code. A way t $C = K + Z_8Z_4 + Z_8Z_2$
- Think how we get expression of C (use "observation" to guess the result).

BCD Adder

Binary Sum						BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1	
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0		0	0	1	0	0	4
0	0	1	0	1	same	0	0	1	0	1	5
0	0	1	1	0		0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
0	1	0	1	0		1	0	0	0	0	10
0	1	0	1	1		1	0	0	0	1	11
0	1	1	0	0		1	0	0	1	0	12
0	1	1	0	1	Δ = 6	1	0	0	1	1	13
0	1	1	1	0		1	0	1	0	0	14
0	1	1	1	1		1	0	1	0	1	15
1	0	0	0	0		1	0	1	1	0	16
1	0	0	0	1		1	0	1	1	1	17
1	0	0	1	0		1	1	0	0	0	18
1	0	0	1	1		1	1	0	0	1	19

- The logic circuit that detects the necessary correction can be derived from the entries in the table.
- It is obvious that a correction is needed when the binary sum has an output carry K = 1.
- The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1.
- The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

BCD Adder

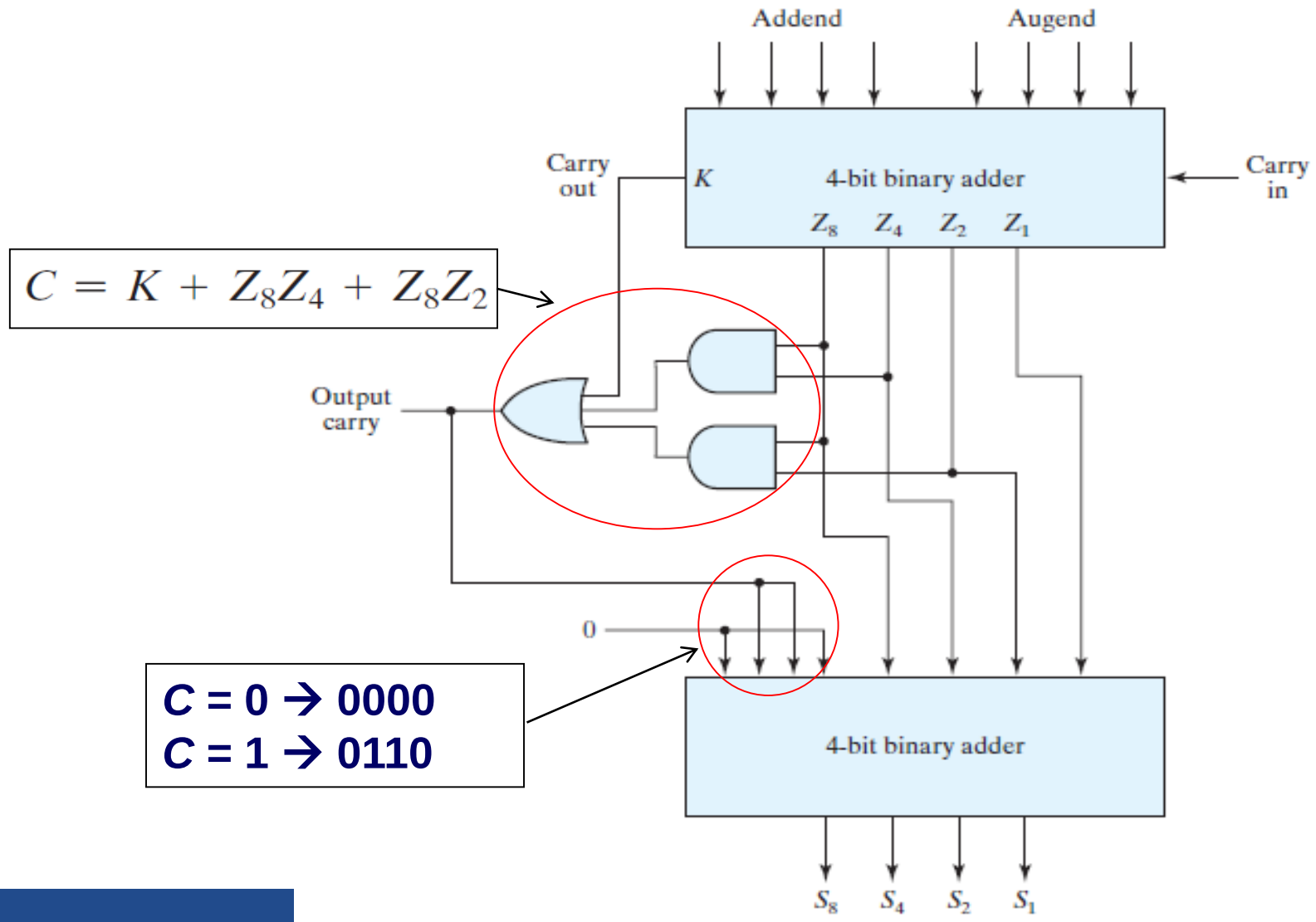


FIGURE 4.14
Block diagram of a BCD adder

Binary Multiplier

- **2 bits x 2 bits**

Multiplicand		B_1	B_0
Multiplier		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
Product	C_3	C_2	C_1	C_0

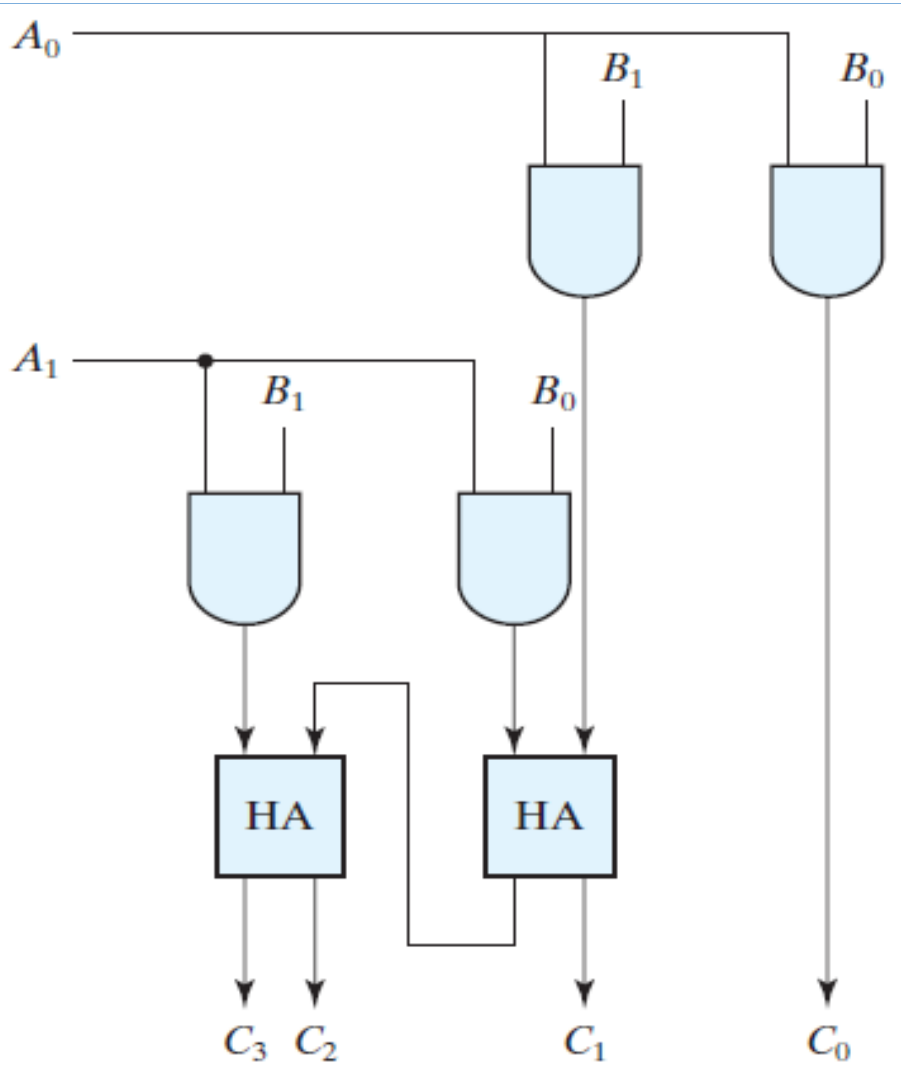


FIGURE 4.15
Two-bit by two-bit binary multiplier

4 Bit × 3 Bit Multiplier

$$
\begin{array}{cccccccc}
 & & & B_3 & B_2 & B_1 & B_0 & \\
 & & X & & A_2 & A_1 & A_0 & \\
\hline
 & & & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 & \\
 & & A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & & \\
 & A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & & \\
\hline
C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 &
\end{array}
$$

4 Bit × 3 Bit Multiplier

			B_3	B_2	B_1	B_0	
		X		A_2	A_1	A_0	
			A_0B_3	A_0B_2	A_0B_1	A_0B_0	
		A_1B_3	A_1B_2	A_1B_1	A_1B_0		
	A_2B_3	A_2B_2	A_2B_1	A_2B_0			
C_6	C_5	C_4	C_3	C_2	C_1	C_0	

For *J* multiplier bits (3) and *K* multiplicand bits (4), we need (J × K) AND gates and (*J* – 1) *K* -bit adders to produce a product of (*J* + *K*) bits.

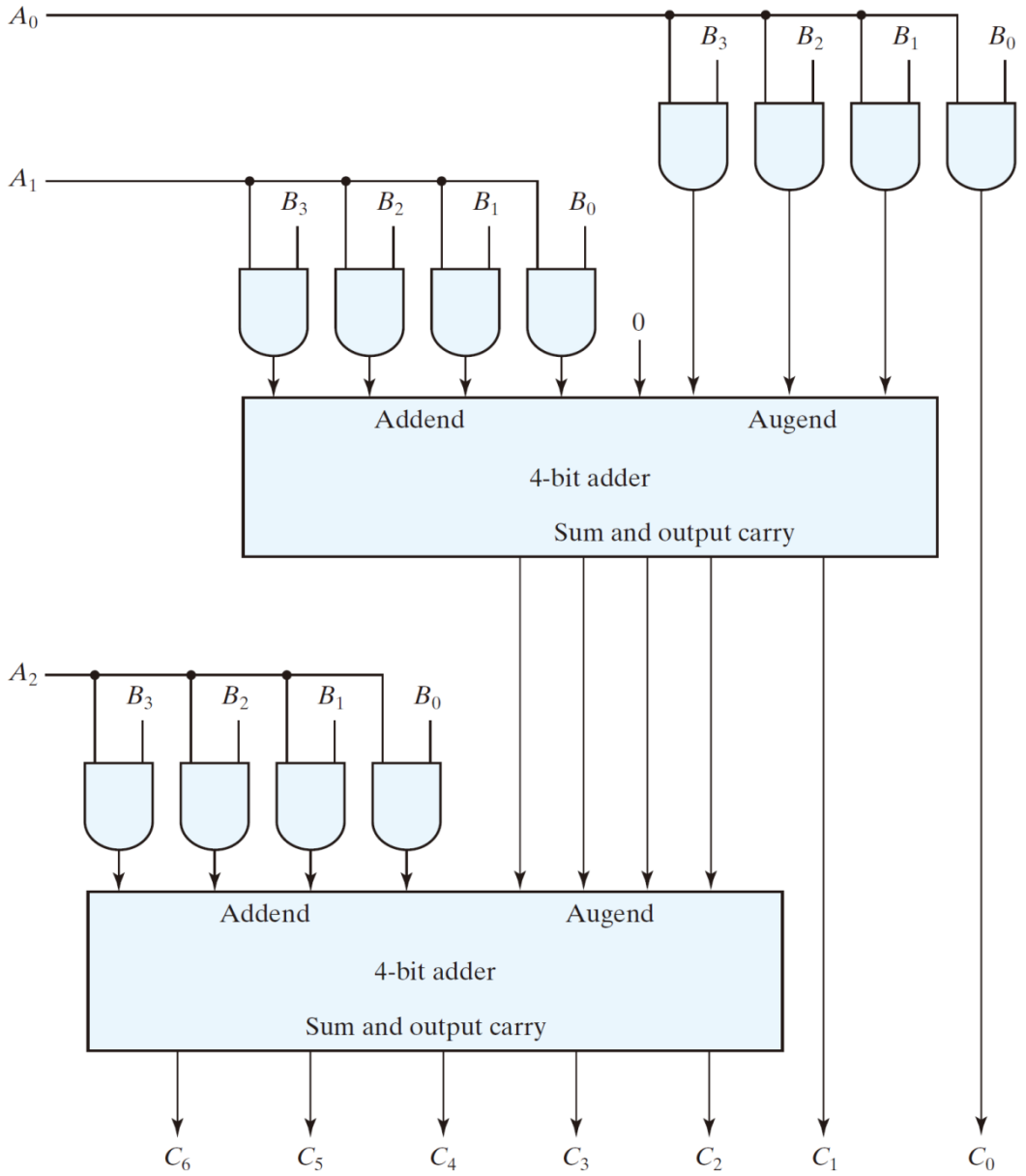


FIGURE 4.16
Four-bit by three-bit binary multiplier

Magnitude Comparator

- A **magnitude comparator** is a combinational circuit that compares two numbers A and B and determines their relative magnitudes.

$$A = A_3\, A_2\, A_1\, A_0$$

$$B = B_3\, B_2\, B_1\, B_0$$

- The comparison of two numbers

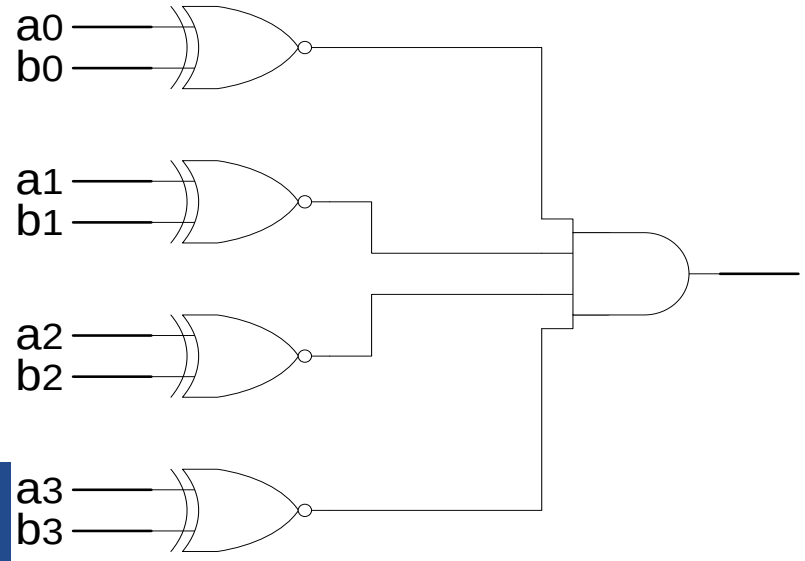
  outputs: A=B, A>B, A<B

- To test the equality of the 4-bit numbers the following circuit can be used

XNOR

a	b	z
0	0	1
0	1	0
1	0	0
1	1	1

XNOR → if $A = B$, $z_i = 1$



Magnitude Comparator

- Algorithm -> logic
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - A=B if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$ and $A_0=B_0$
    - equality: $x_i = A_iB_i + A_i'B_i'$, for i = 0, 1, 2, 3
    - $(A=B) = x_3x_2x_1x_0$
  - $(A>B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$
  - $(A<B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$
- Implementation
  - $x_i = (A_iB_i' + A_i'B_i)'$

- $A > B$ means: $A_3 = 1$ & $B_3 = 0$
  or $(A_3 = B_3)$ & $A_2 = 1$ & $B_2 = 0$
  or $(A_3 = B_3, A_2 = B_2)$ & $A_1 = 1$ & $B_1 = 0$
  or $(A_3 = B_3, A_2 = B_2, A_1 = B_1)$ & $A_0 = 1$ & $B_0 = 0$
- $A < B$ means: $A_3 = 0$ & $B_3 = 1$
  or $(A_3 = B_3)$ & $A_2 = 0$ & $B_2 = 1$
  or $(A_3 = B_3, A_2 = B_2)$ & $A_1 = 0$ & $B_1 = 1$
  or $(A_3 = B_3, A_2 = B_2, A_1 = B_1)$ & $A_0 = 0$ & $B_0 = 1$

Magnitude Comparator

- Algorithm -> logic
  - $A = A_3A_2A_1A_0$; $B = B_3B_2B_1B_0$
  - A=B if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$and $A_0=B_0$
    - equality: $x_i = A_iB_i+A_i'B_i'$, for i = 0, 1, 2, 3
    - $(A=B) = x_3x_2x_1x_0$
  - $(A>B) = A_3B_3'+x_3A_2B_2'+x_3x_2A_1B_1'+x_3x_2x_1A_0B_0'$
  - $(A<B) = A_3'B_3+x_3A_2'B_2+x_3x_2A_1'B_1+x_3x_2x_1A_0'B_0$
- Implementation
  - $x_i = (A_iB_i'+A_i'B_i)'$

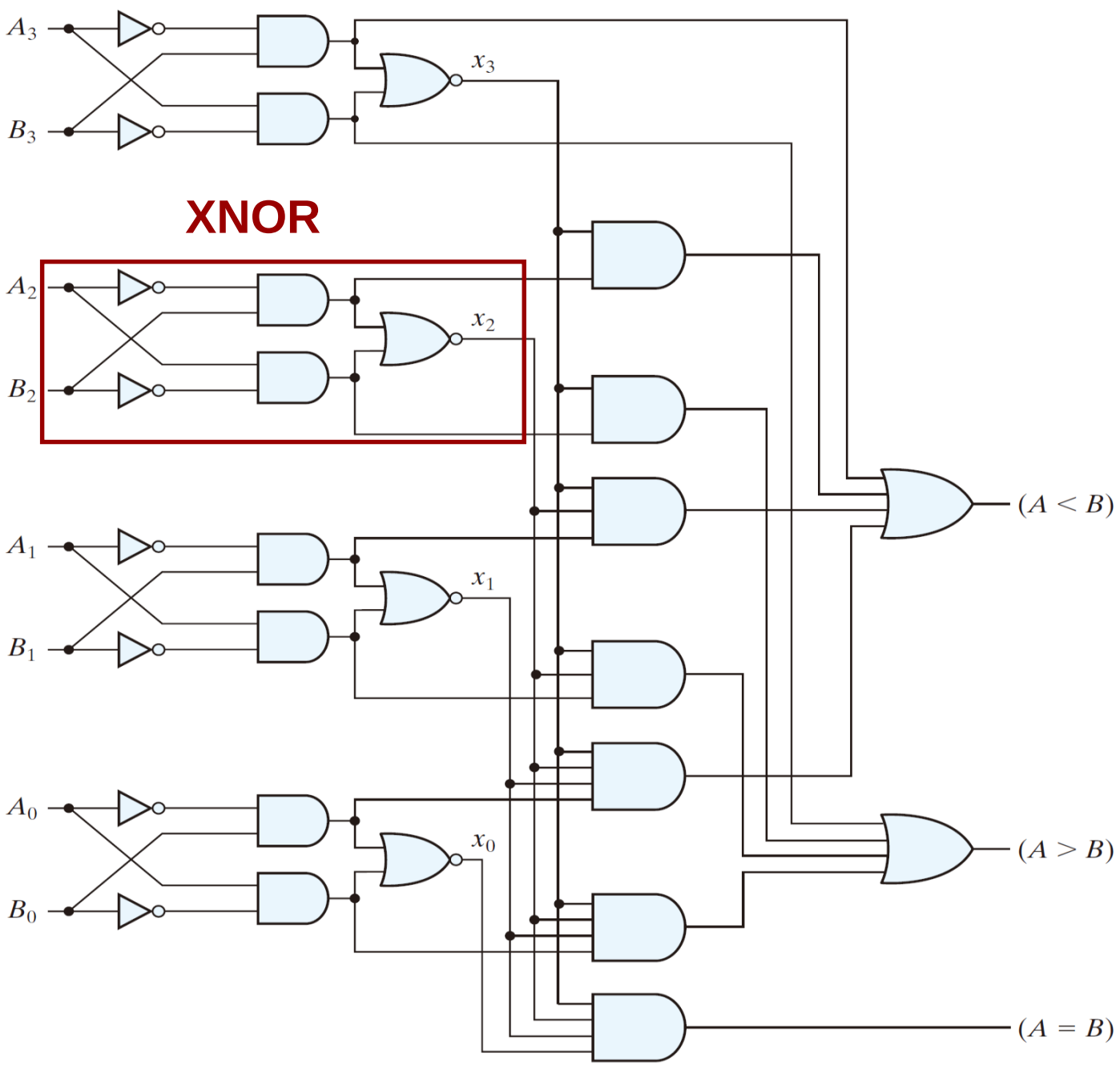


FIGURE 4.17
Four-bit magnitude comparator

Decoders

- A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines
  - May have fewer than 2^n outputs
- A n-to-m-line decoder ($m \leq 2^n$):
  - Generate the m minterns of n input variables
- For each possible input combination, there is only one output that is equal to 1
- The output whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines

3-to-8-Line Decoder

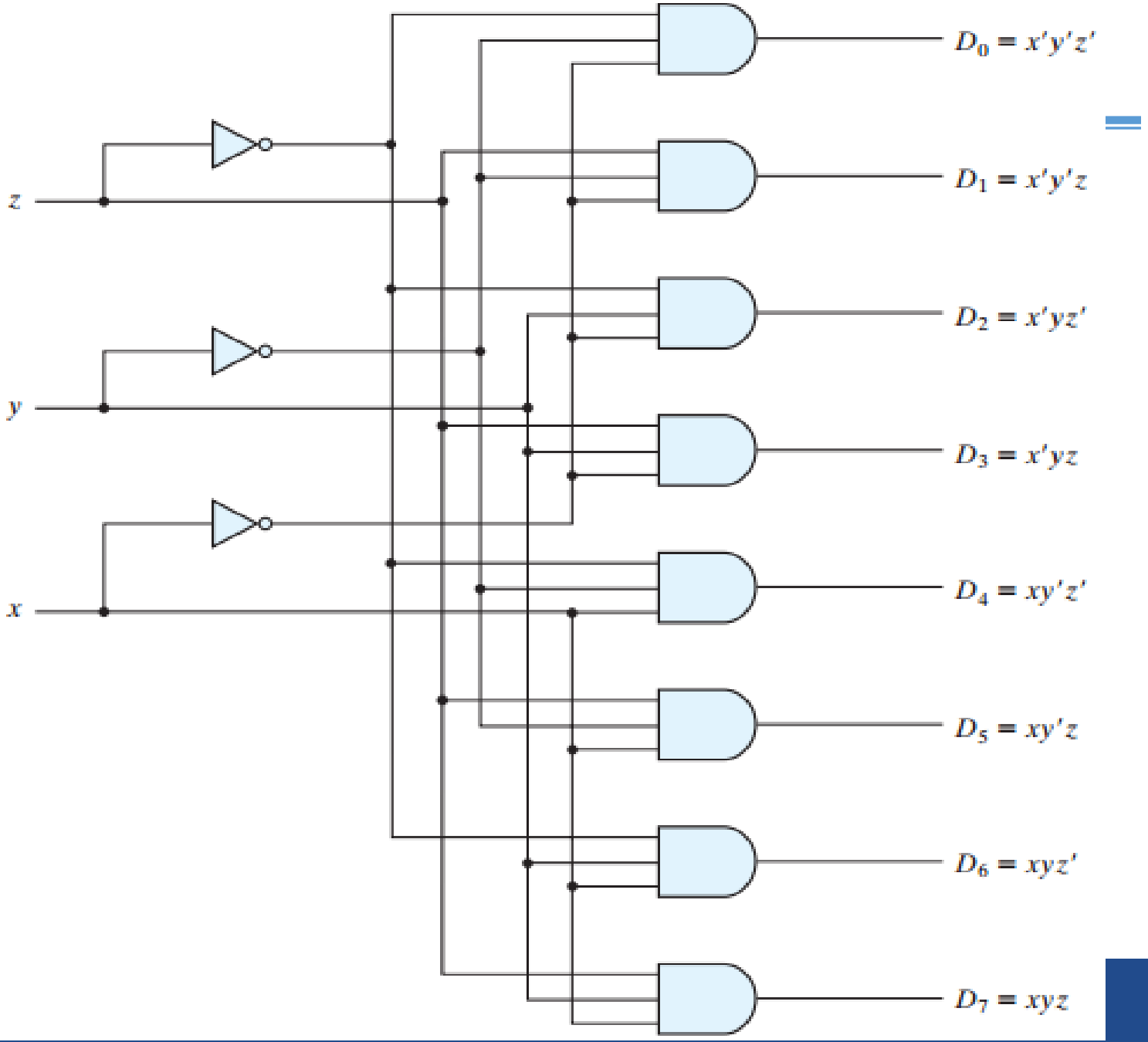
- The 3 inputs are decoded into 8 outputs
- Each represent one of the minterms of the inputs variables

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

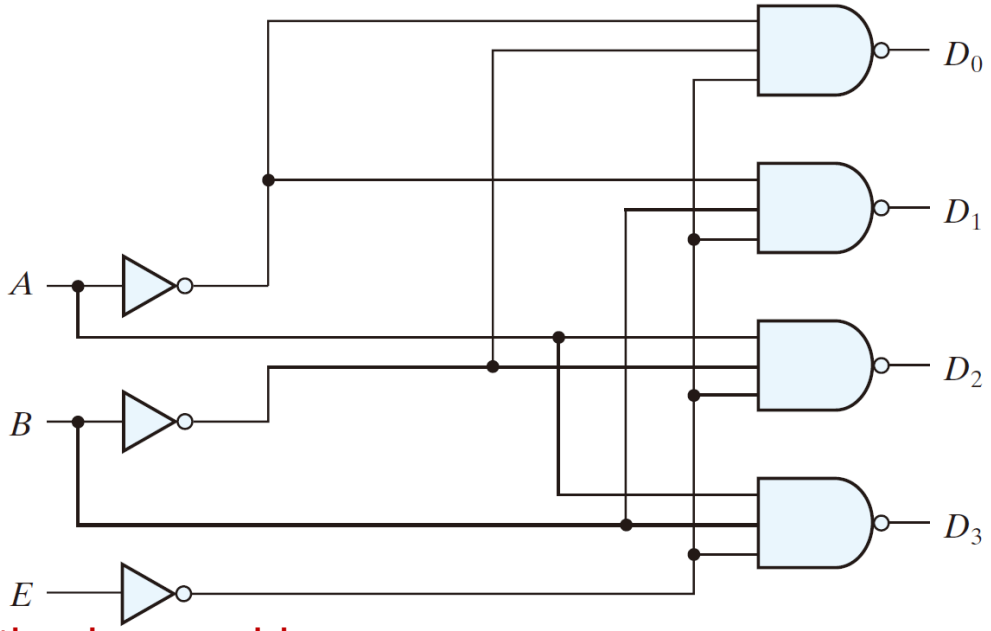
3-to-8-line decoder

The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables.



2-to-4-Line Decoder

- A circuit that receives information from a single line and directs it to one of 2^n possible output lines
- A decoder with enable input can function as a demultiplexer
  - Often referred to as a decoder/demultiplexer



(a) Logic diagram

Data input

Selection

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

- Only one output can be equal to 0 at any given time, all other outputs are equal to 1.
- The output whose value is equal to 0 represents the minterm selected by inputs *A* and *B* .

FIGURE 4.19
Two-to-four-line decoder with enable input

Construct Larger Decoders

- Decoders with enable inputs can be connected together to form a larger decoder
- The enable input is used as the most significant bit of the selection signal
- w=0: the top decoder is enabled
- w=1: the bottom one is enabled
- In general, enable inputs are a convenient feature for standard components to expand their numbers of inputs and outputs

Construct 4 × 16 decoder with two 3 × 8 decoders

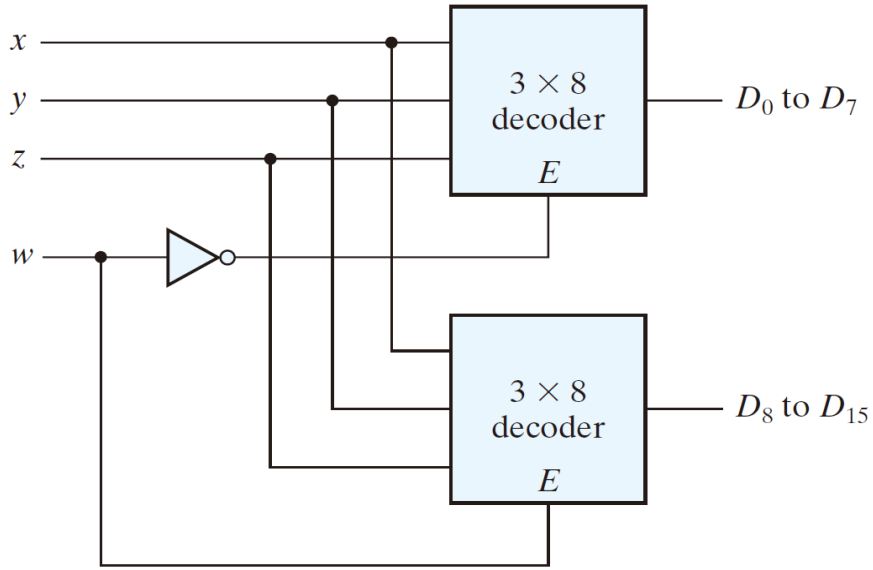


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

Combinational Logic Implementation

- Most of combinational logics can be implemented by the decoder, e.g., a full adder as shown below.
- Note that the decoder outputs represent all minterms.

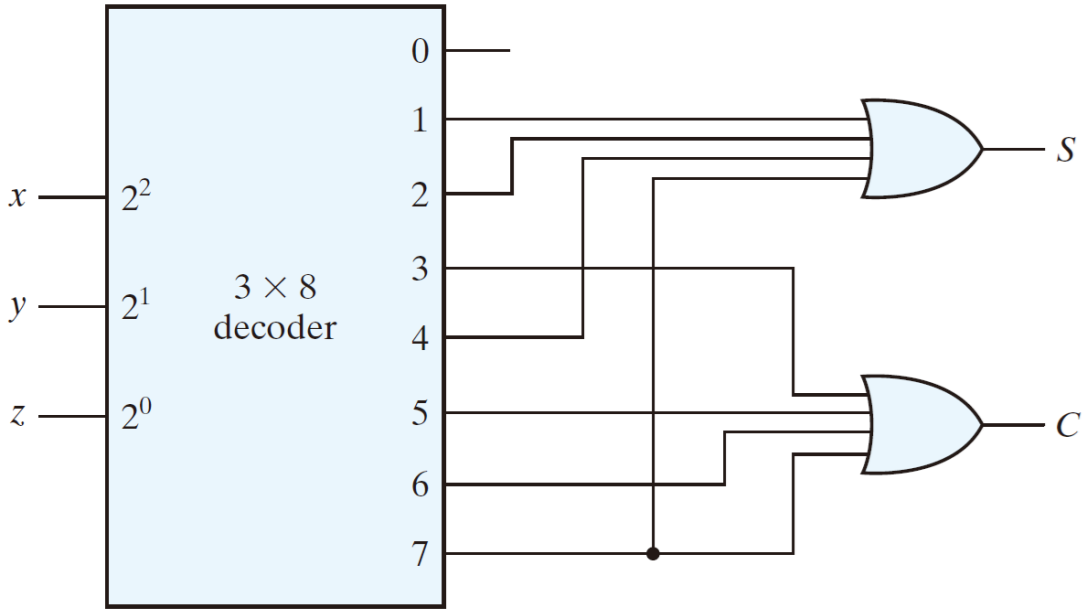


FIGURE 4.21
Implementation of a full adder with a decoder

Table 4.4
Full Adder

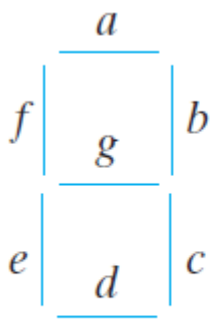
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$

$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

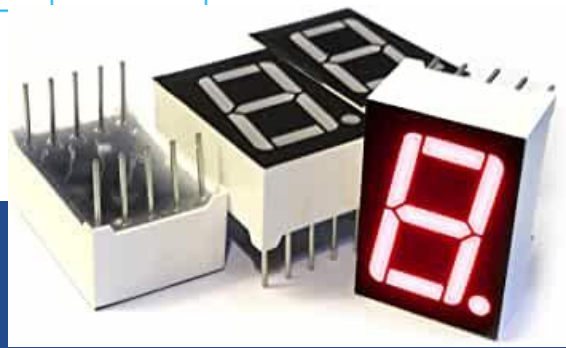
BCD to Seven-segment display

A **BCD–to–seven–segment decoder** is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in a display indicator used for displaying the decimal digit in a familiar form. The seven outputs of the decoder (a,b,c,d,e,f and g) select the corresponding segments in the display, as shown in the following figure (a). The numeric display chosen to represent the decimal digit is shown in figure (b). Using a truth table and K–maps, design the BCD–to–seven–segment decoder using a minimum number of gates. The six invalid combinations should result in a blank in a display.



(a) Segment designation

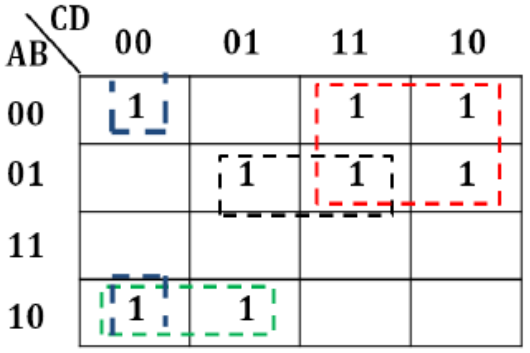
(b) Numerical designation for display

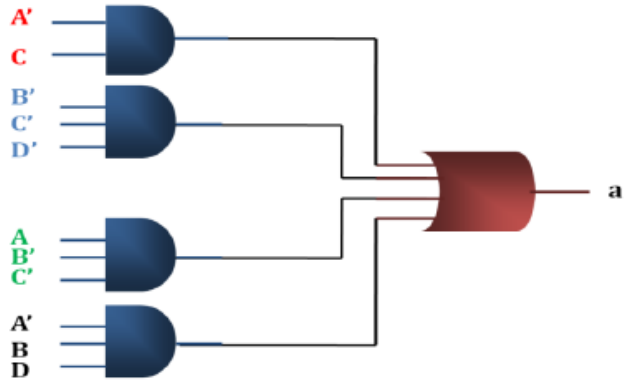


Using a truth table and K–maps, design the BCD–to–seven–segment decoder using a minimum number of gates. The six invalid combinations should result in a blank in a display.

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

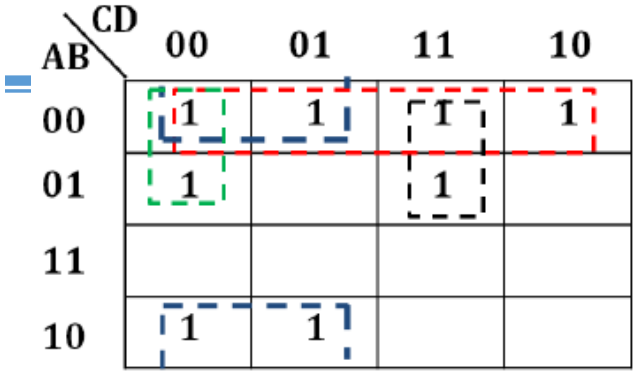
a

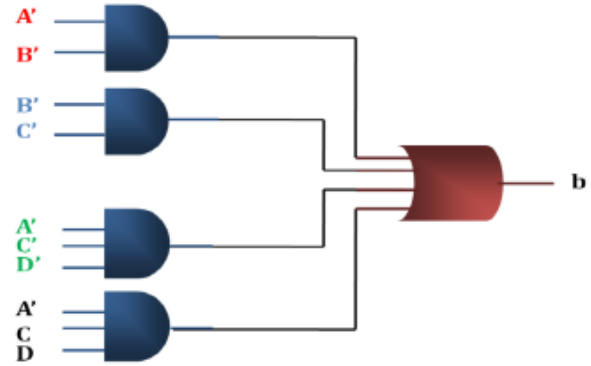




a = A'C+ AB'C' + B'C'D' + A'BD

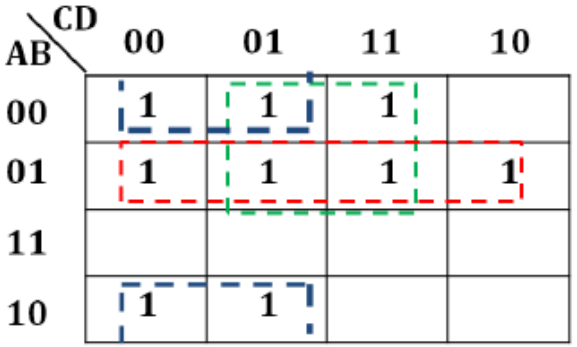
b

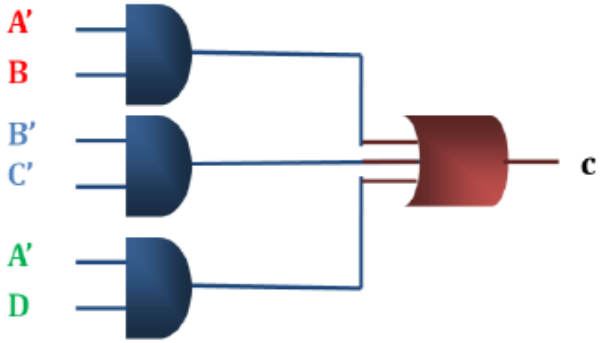




b = A'B'+ A'C'D' + B'C' + A'CD

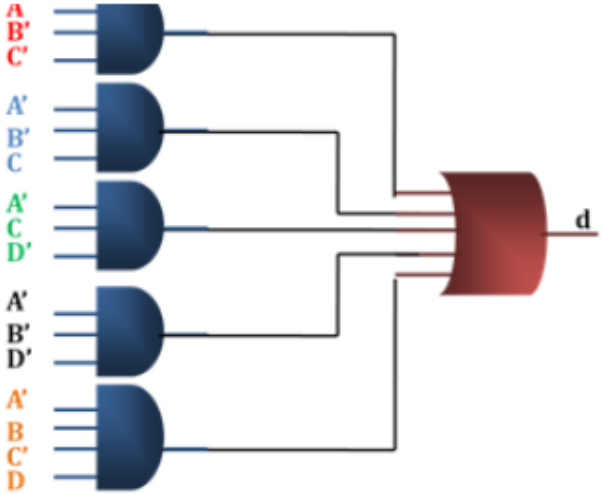
c





c = A'B+ A'D + B'C'

d

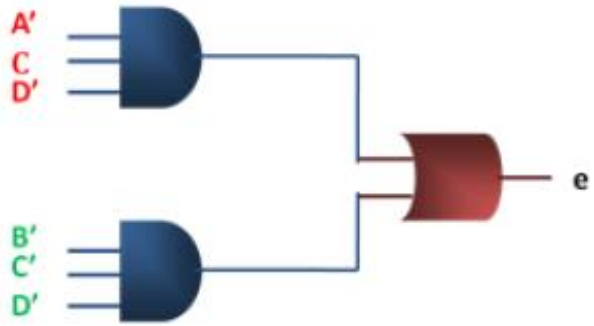


d = AB'C'+ A'CD' + A'B'C +
A'B'D' + A'BC'D

e

AB \ CD	00	01	11	10
00	1			1
01				1
11				
10	1			

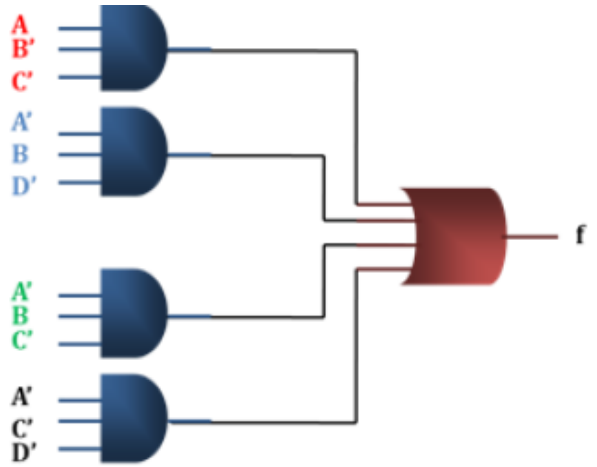
e = A'CD' + B'C'D'



f

AB \ CD	00	01	11	10
00	1			
01	1	1		1
11				
10	1	1		

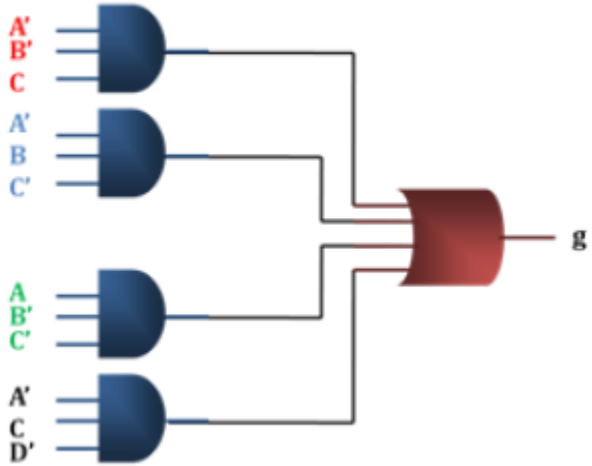
f = AB'C' + A'BC' + A'BD' + A'C'D'



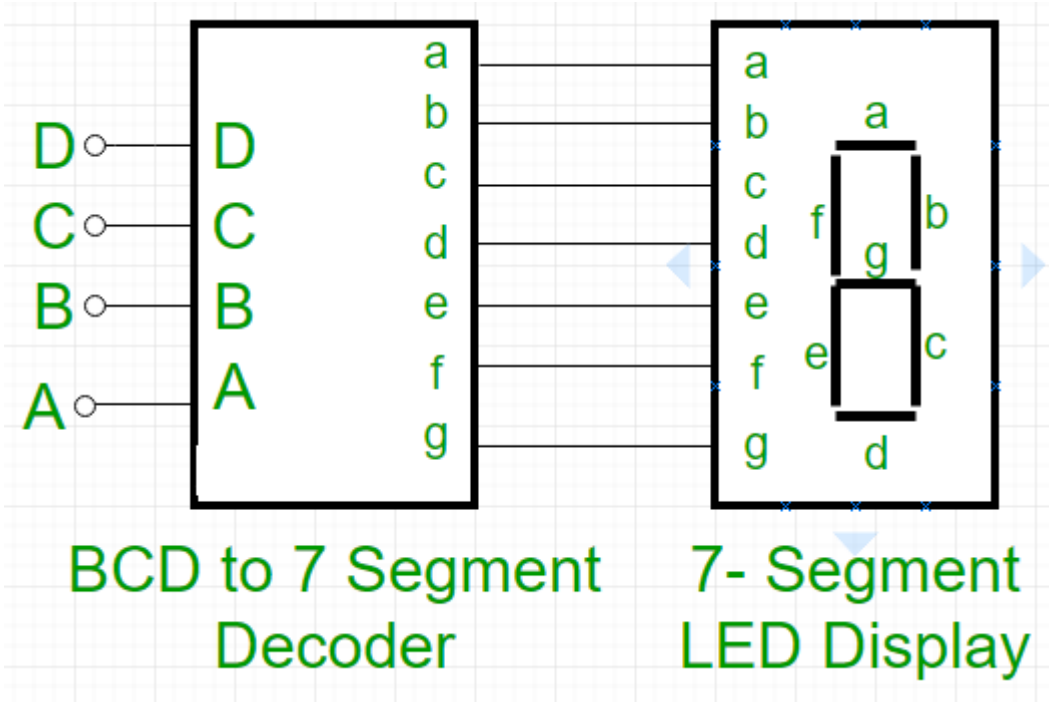
g

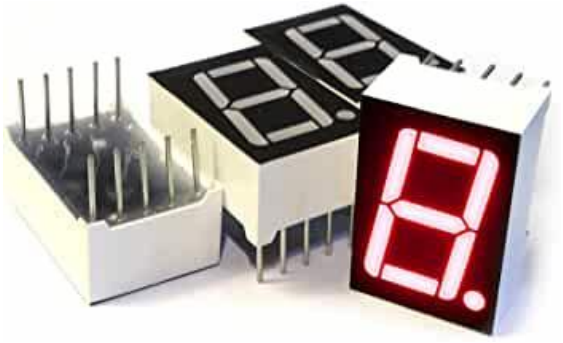
AB \ CD	00	01	11	10
00			1	1
01	1	1		1
11				
10	1	1		

g = A'B'C + AB'C' + A'BC' + A'CD'



BCD to Seven-segment display





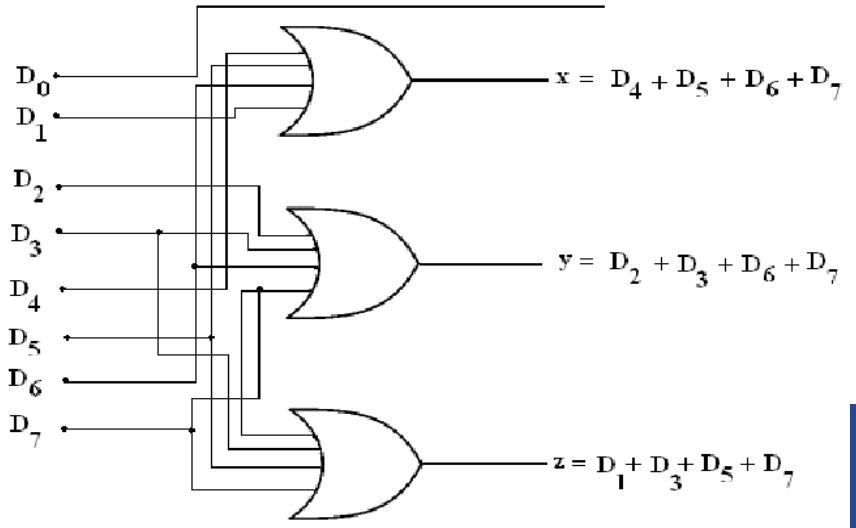
Encoders

- An encoder is a digital circuit that performs the inverse operation of a decoder.
- An encoder has 2^n input lines and n output lines.
- The output lines, as an aggregate, generate the binary code corresponding

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$



Priority Encoder

- The Boolean functions for the encoder in the last page have the limitation that only one input can be “1”.
- If there are two or more “1” in the inputs, the result will be wrong (e.q. D_1, D_3= 1 → ambiguous results!)
- Adding an input priority function can solve the ambiguity. If the inputs have more than two “1”, the priority function only consider the most significant bit.
- Priority Encoder: An encoder circuit that includes the priority function.

Priority Encoder

- An encoder circuit that includes the priority function
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence
- In the following truth table:

  D3 > D2 > D1 > D0

- The X's in output columns represent don't-care conditions
- The X's in input columns are useful for representing a truth table in condensed form

Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

V = 0 :
no valid inputs

Priority Encoder

The maps for simplifying outputs x and y

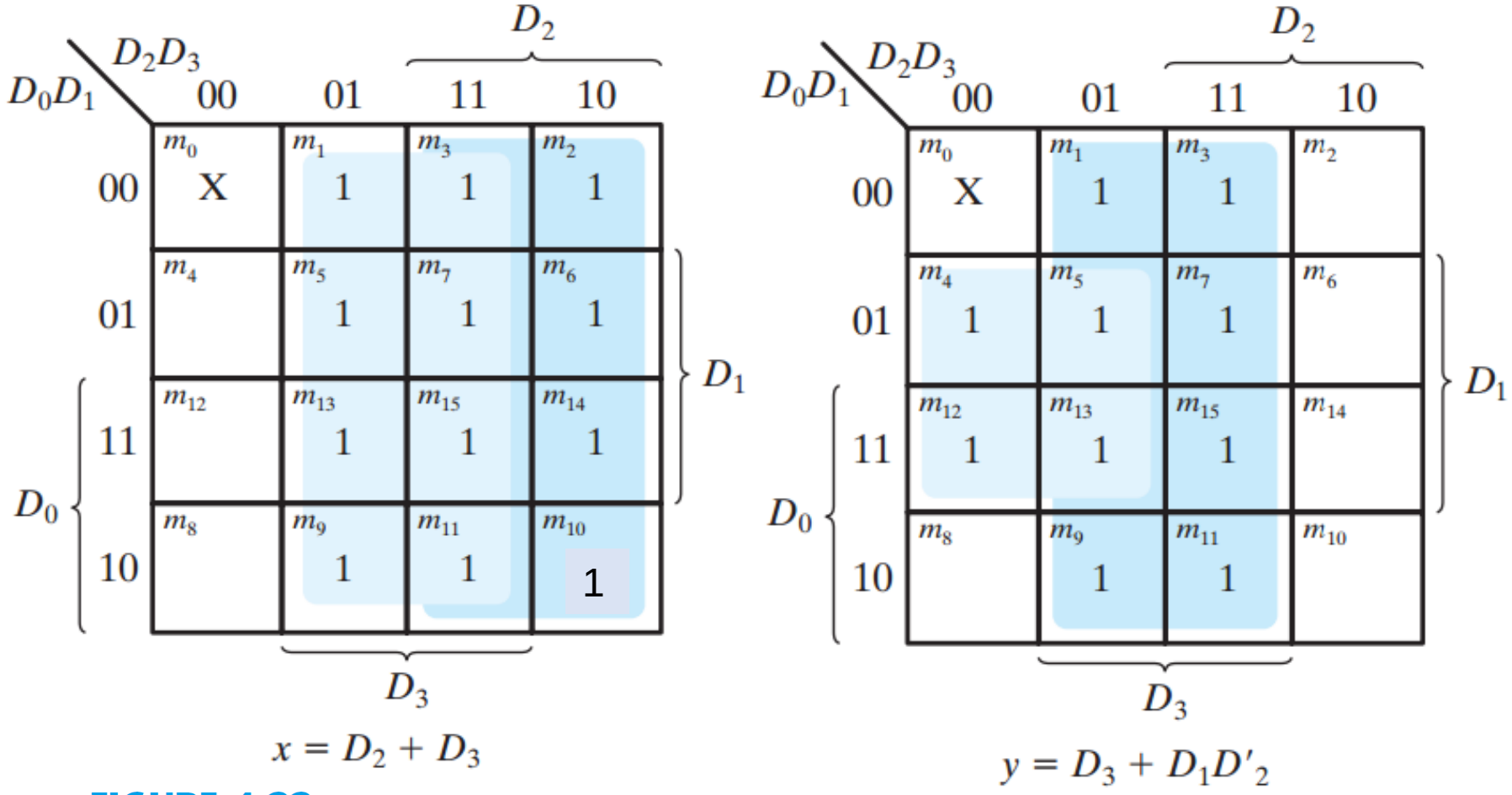
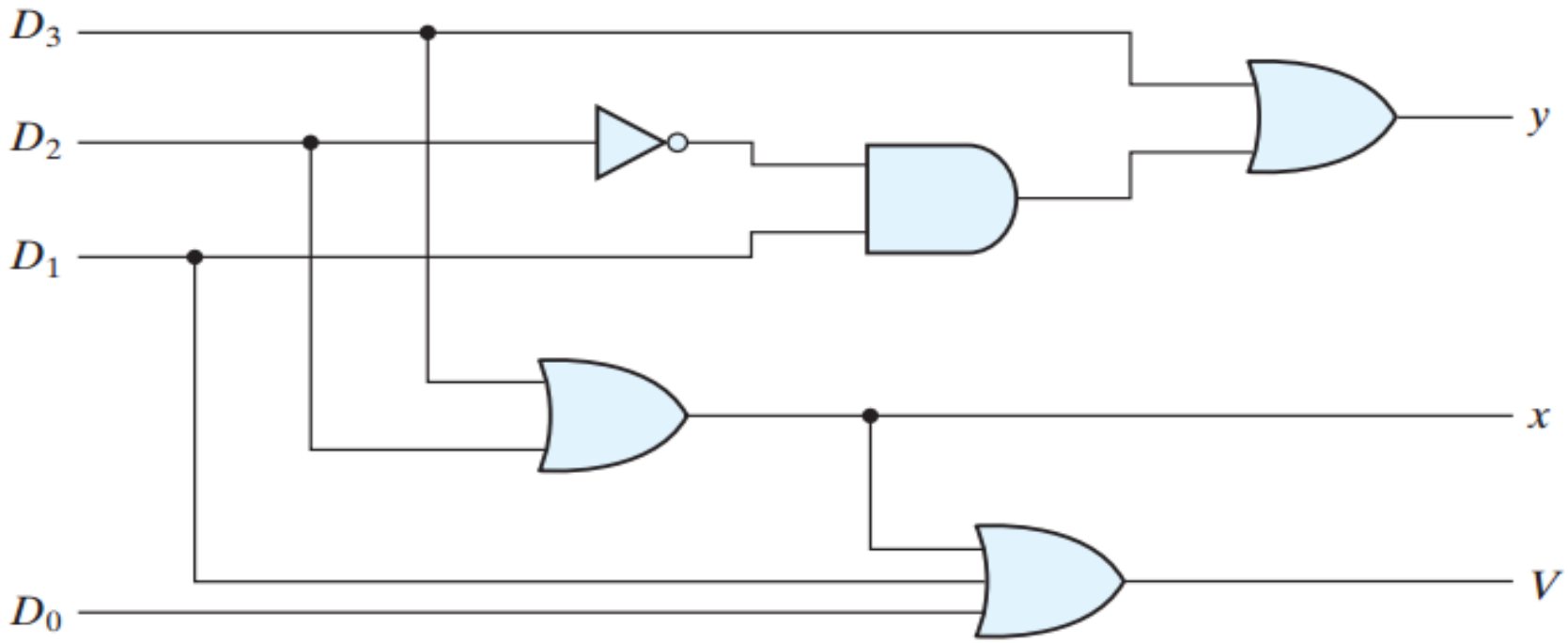


FIGURE 4.22
Maps for a priority encoder

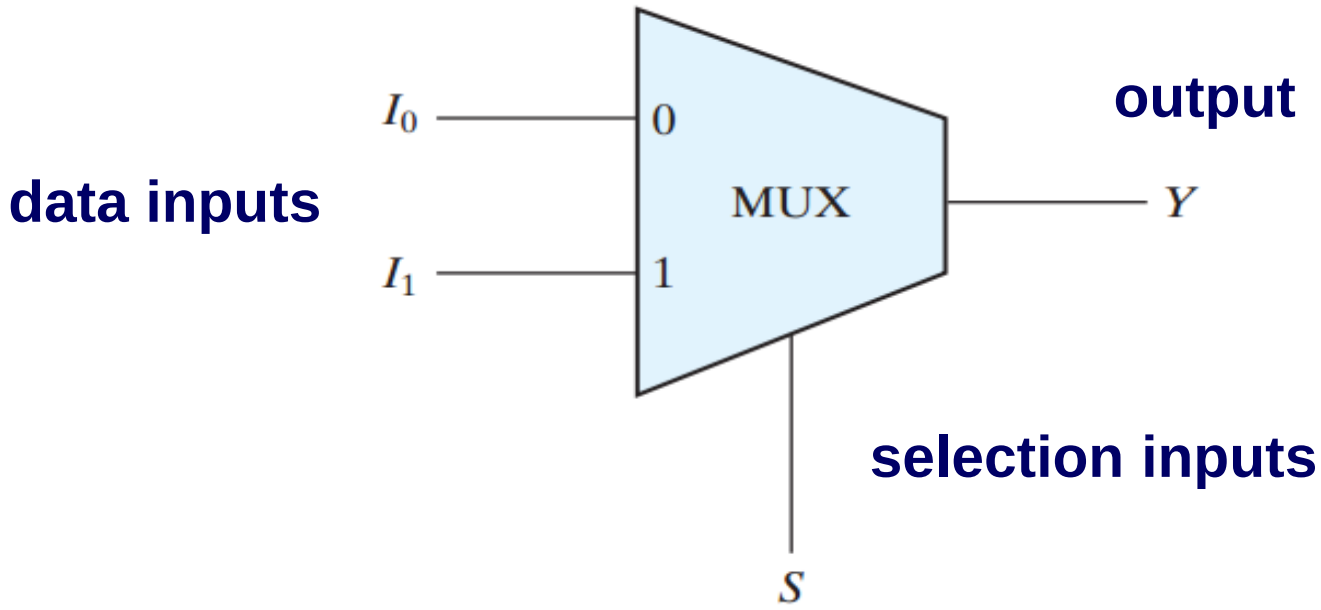
Priority Encoder

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$



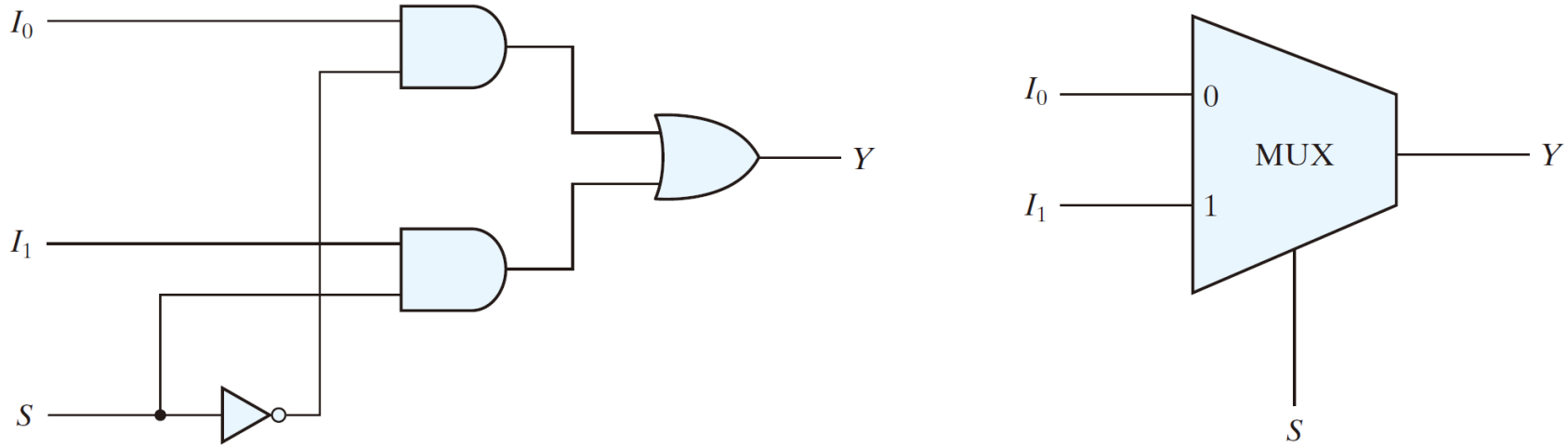
Multiplexer

- A multiplexer, often labeled "MUX", is a circuit that selects from one of many input lines and directs it to a single output line.
- It usually contains "data inputs", "selection inputs" and one "output".



Multiplexer

- **A multiplexer, often labeled "MUX", is a circuit that selects from one of many input lines and directs it to a single output line.**
  - Have 2^n input lines and n selection lines
  - Act like an electronic switch (also called a data selector)
- For the following 2-to-1-line multiplexer:
  - S=0 → Y = I_0 ; S=1 →Y = I_1

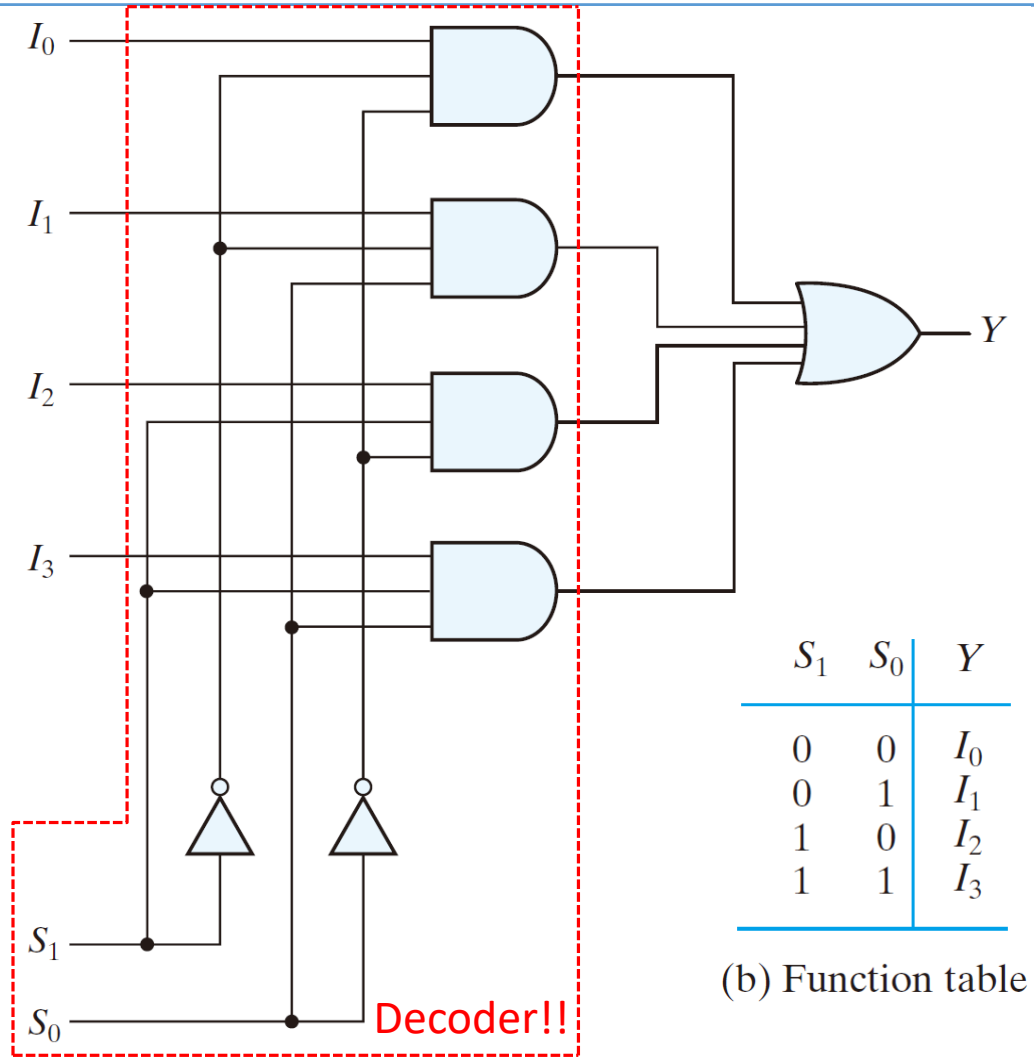


(a) Logic diagram (b) Block diagram

FIGURE 4.24
Two-to-one-line multiplexer

4-to-1-line Multiplexer

- **2^n inputs require *n* selection bits.**
- **A multiplexer can be modified from a decoder.**
- The combinations of S_0 and S_1 control each AND gates
- Part of the multiplexer resembles a decoder
- To construct a multiplexer:
  - Start with an n-to-2^n decoder
  - Add 2^n input lines, one to each AND gate
  - The outputs of the AND gates are applied to a single OR gate



(a) Logic diagram

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

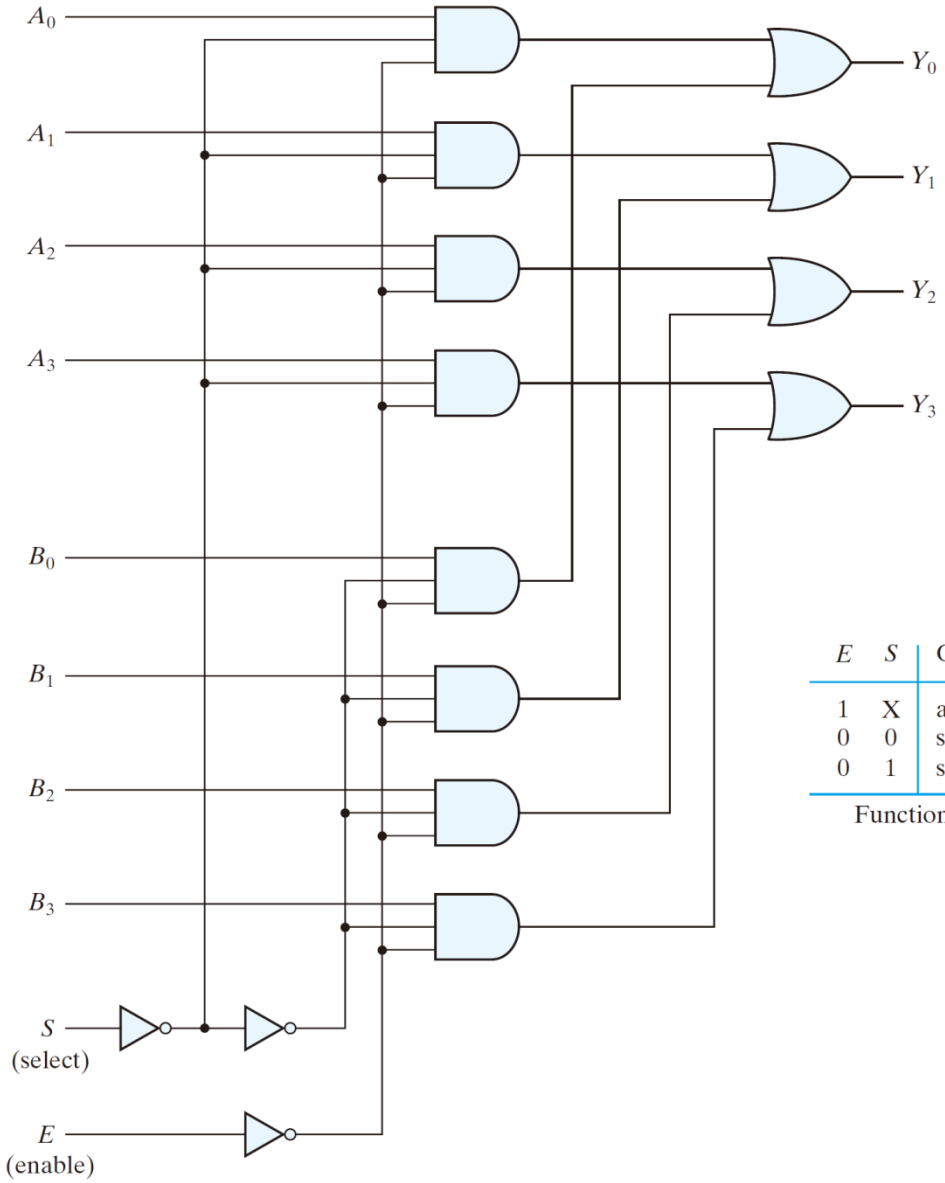
(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

Quadruple 2-to-1-line Multiplexer

- Multiplexers can be combined with common selection inputs to provide multiple-bit selection logic
- Quadruple 2-to-1-line multiplexer:
  - Four 2-to-1-line multiplexers
  - Each capable of selecting one bit of two 4-bit inputs
  - E: enable input

    E=1: disable the circuit (all outputs are 0)



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

FIGURE 4.26
Quadruple two-to-one-line multiplexer

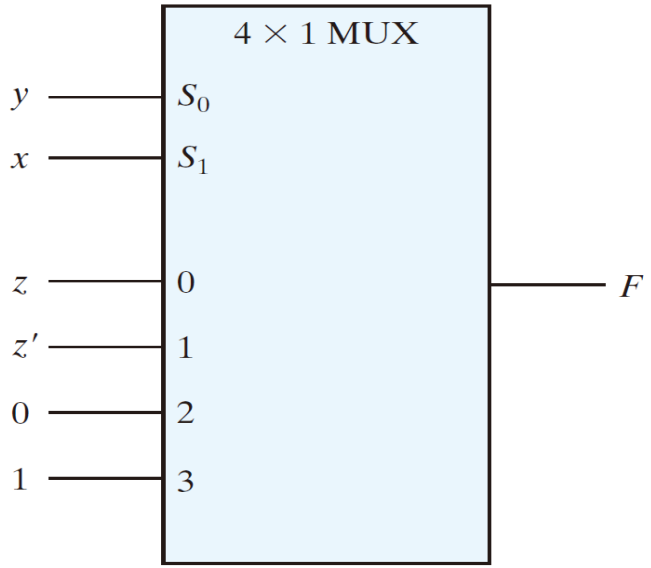
Boolean Function Implementation

- A multiplexer is essentially a decoder with an external OR gate
  - Can be used to implement Boolean functions without extra logic
- To implement a Boolean function of n variables:
  - Use a multiplexer with n – 1 selection inputs
  - The first n – 1 variables are connected to the selection inputs
  - The remaining variable is used for the data inputs

Example: F(x, y, z) = Σ(1,2,6,7)

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

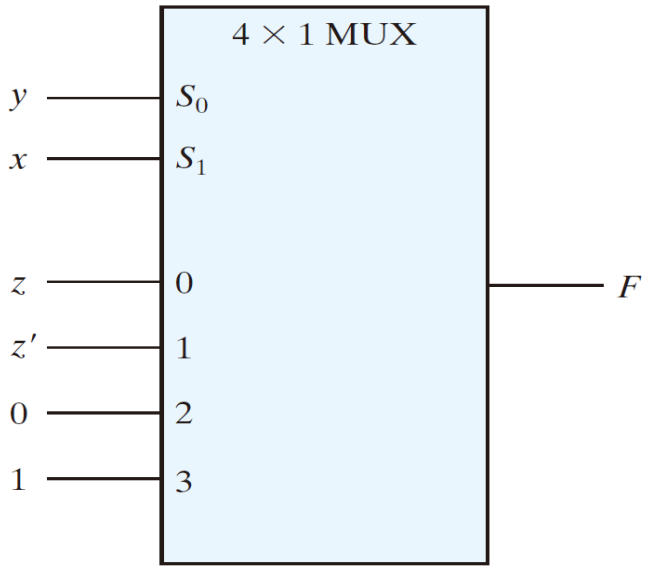
FIGURE 4.27
Implementing a Boolean function with a multiplexer

Boolean Function Implementation

Example: F(x, y, z) = Σ(1,2,6,7)

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

- The two variables *x* and *y* are applied to the selection lines in that order; *x* is connected to the S_1 input and *y* to the S_0 input.
- The values for the data input lines are determined from the truth table of the function.
- When *xy* = 00, output *F* is equal to *z* because *F* = 0 when *z* = 0 and *F* = 1 when *z* = 1. This requires that variable *z* be applied to data input 0.
- The operation of the multiplexer is such that when *xy* = 00, data input 0 has a path to the output, and that makes *F* equal to *z*.
- In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of *F* when *xy* = 01, 10, and 11, respectively.

Boolean Function Implementation

- The general procedure for implementing any Boolean function of *n* variables with a multiplexer with *n* - 1 selection inputs and 2^{n-1} data inputs.
- To begin with, Boolean function is listed in a truth table.
- Then first *n* - 1 variables in the table are applied to the selection inputs of the multiplexer.
- For each combination of the selection variables, we evaluate the output as a function of the last variable.
- This function can be 0, 1, the variable, or the complement of the variable.
- These values are then applied to the data inputs in the proper order.

Boolean Function Implementation

Implementing a 4-Input Function

Example: F(A, B, C, D) = Σ(1, 3, 4, 11, 12, 13, 14, 15)

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

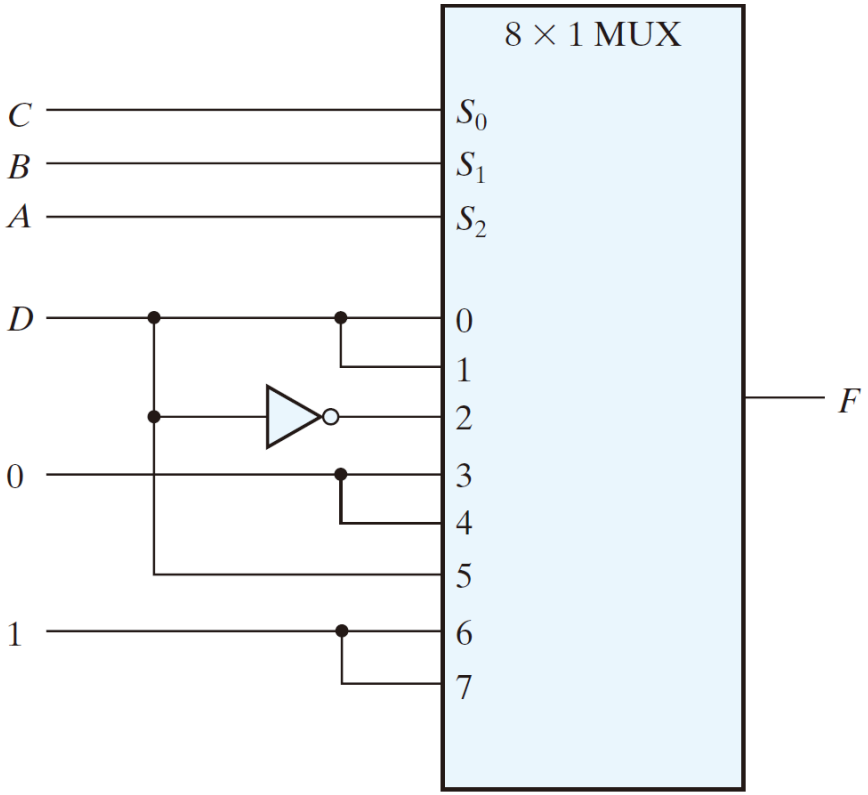


FIGURE 4.28
Implementing a four-input function with a multiplexer

Three-State Gates

- A circuit that exhibits three states
  - logic 1, logic 0, and high-impedance (z)
- The high-impedance state acts like an open circuit (disconnected)
- The most commonly used three-state gate is the buffer gate
  - C=0 → disabled (high-impedance) ; C=1 → enabled (pass)
  - Can be used at the output of a function without altering the internal implementation

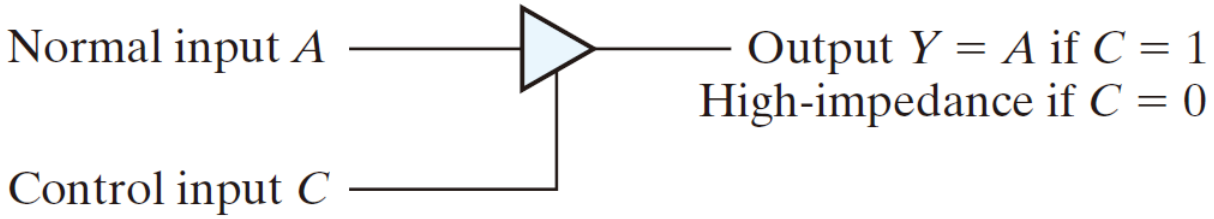


FIGURE 4.29
Graphic symbol for a three-state buffer

Three-State Gates

- A large number of three-state gate outputs can be connected with wires to form a common line (bus) without logic conflicts
  - Very convenient for implementing some circuits (ex: multiplexer)
  - Only one buffer can be in the active state at any given time
- One way to ensure that no more than one control input is active at any given time is to use a decoder

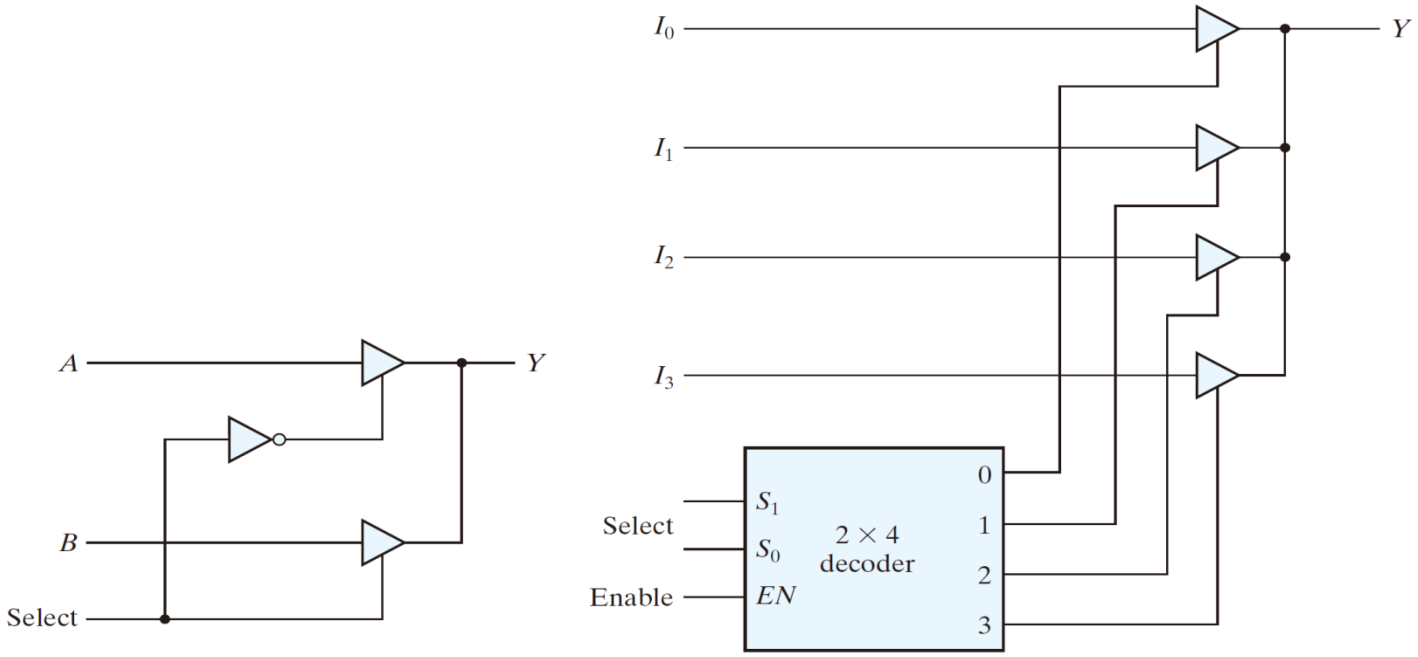


FIGURE 4.30
Multiplexers with three-state gates

The End

Reference:

1. **Digital Design (with an introduction to the Verilog HDL) 6th Edition, M. Morris Mano, Michael D. Ciletti**