Ch1: Digital Systems and Binary Numbers
Ch2: Boolean Algebra and Logic Gates
Ch3: Gate-Level Minimization
Ch4: Combinational Logic
Ch5: Synchronous Sequential Logic
Ch6: Registers and Counters

Chapter 4
Combinational Logic

Switching Theory & Logic Design
1403271-4

Prof. Adnan Gutub

Main Ref: M. Morris Mano and Michael D. Ciletti, Digital Design, Prentice Hall

Content

Combinational Logic

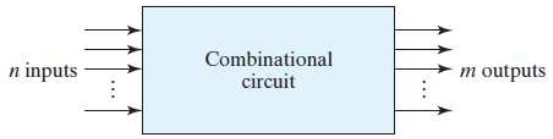
2

4.1 Introduction

- Logic Circuits: Combinational or Sequential
- Combinational
  - No Memory (No Storage)
  - Outputs fully based on combination of inputs
- Sequential
  - Memory (Storage) + Combinational Logic

3

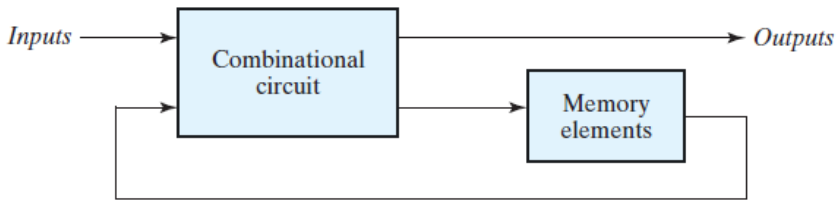
4.2 Combinational Circuits



Analysis

Design

Sequential



4

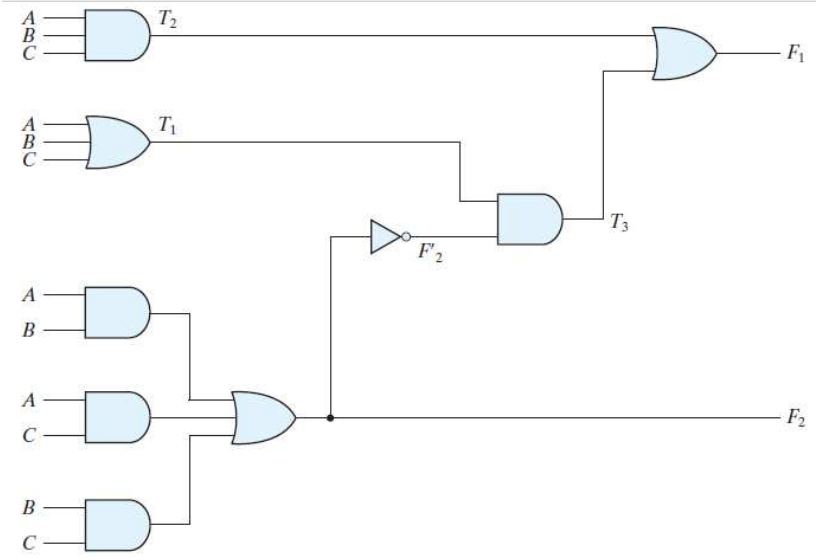
4.3 Analysis Procedure

1. Label all gate outputs as functions of input variables.
   - Use arbitrary symbols—with meaningful names.
2. Label gates as functions of input variables. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. Obtain output Boolean functions in terms of input variables.

5

Analysis Example

Find F_1 & F_2 in terms of inputs?



6

Truth Table

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

7

4.4 Design Procedure

- Determine required number of inputs and outputs → assign a symbol to each.
- Derive truth table relating inputs and outputs.
- Simplify outputs Boolean function.
- Derive Logic Diagram

8

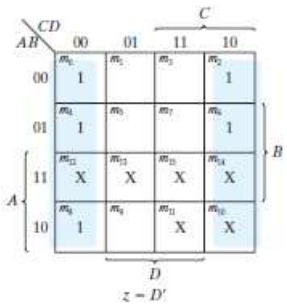
Code Conversion Example

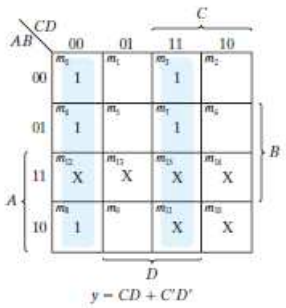
Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

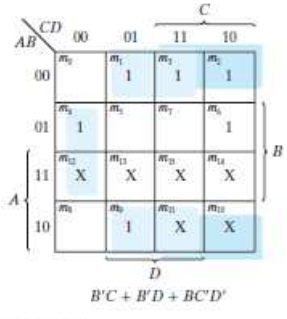
9

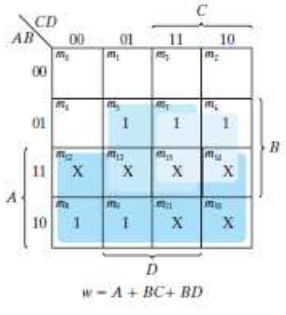
Code Conversion Example

- Utilize Don't Care
- Simplify Functions









$z = D'$

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

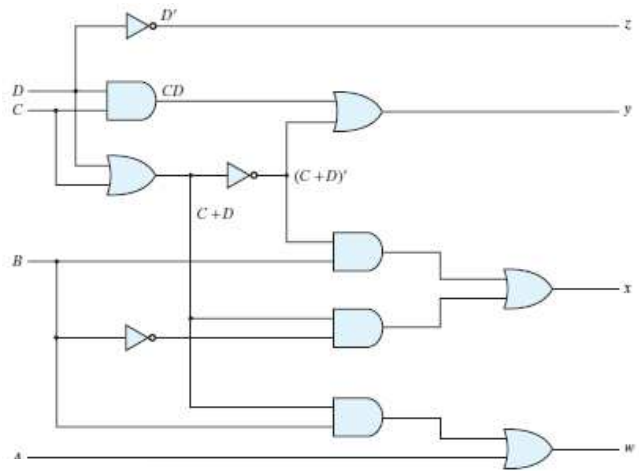
$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$

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

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

10

Code Conversion Example Logic Diagram



$z = D'$
$y = CD + C'D' = CD + (C + D)'$
$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$
$= B'(C + D) + B(C + D)'$
$w = A + BC + BD = A + B(C + D)$

11

4.5 Binary Adder–Subtractor

- simple addition consists of four possible elementary operations:
  - 0+0 = 0
  - 0+1 = 1
  - 1+0 = 1
  - 1+1 = 10 ← Sum has Carry
  - Half Adder (two '2' input bits)
  - Full Adder (three '3' input bits)

12

Half Adder (Top Overview)

- Binary addition used frequently
- Addition Development:
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block,
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block,
  - *Ripple Carry Adder*, an iterative array to perform binary addition, and
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.

13

Functional Block: Half-Adder

- **A 2-input, 1-bit width binary adder that performs the following computations:**

X	**0**	**0**	**1**	**1**
+ Y	**+ 0**	**+ 1**	**+ 0**	**+ 1**
C S	**0 0**	**0 1**	**0 1**	**1 0**

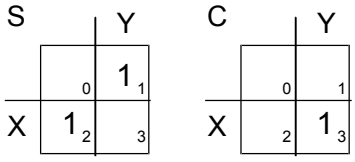
- **A half adder adds two bits to produce a two-bit sum**
- **The sum is expressed as a sum bit , S and a carry bit, C**
- **The half adder can be specified as a truth table for S and C ⇒**

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Chapter 4 14

Logic Simplification: Half-Adder

- The K-Map for S, C is:
- This is a pretty trivial map! By inspection:



$$S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y$$

$$S = (X + Y) \cdot \overline{(X} + \overline{Y)}$$

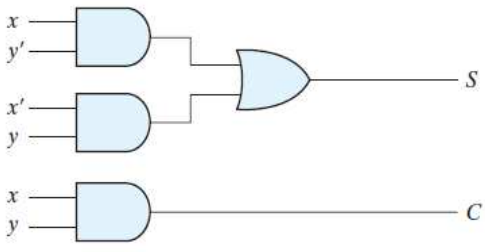
- and

$$C = X \cdot Y$$

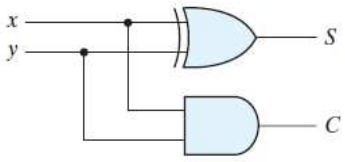
$$C = \overline{(\overline{X \cdot Y})}$$

- These equations lead to several implementations.

Half Adder Implementation



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



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

Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.
  - For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+Y	+0	+1	+0	+1
C S	0 0	0 1	0 1	1 0

  - For a carry- in (Z) of 1:

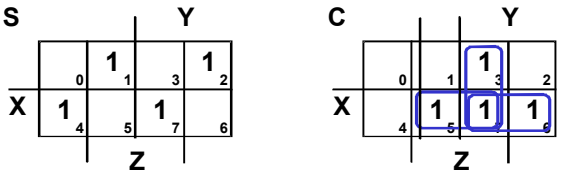
Z	1	1	1	1
X	0	0	1	1
+Y	+0	+1	+0	+1
C S	0 1	1 0	1 0	1 1

Logic Optimization: Full-Adder

- Full-Adder Truth Table:

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 K-Map:



Equations: Full-Adder

- From the K-Map, we get:

$$S = X\overline{Y}\,\overline{Z} + \overline{X}\,Y\,\overline{Z} + \overline{X}\,\overline{Y}\,Z + X\,Y\,Z$$
$$C = X\,Y + X\,Z + Y\,Z$$

- The S function is the three-bit XOR function (Odd Function):

$$S = X \oplus Y \oplus Z$$

- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:

$$C = X\,Y + (X \oplus Y)\,Z$$

- The term X·Y is *carry generate.*
- The term X⊕Y is *carry propagate.*

Logic and Computer Design Fundamentals, 4e
PowerPoint® Slides
© 2008 Pearson Education, Inc.

Chapter 4 19

Full Adder Implementation

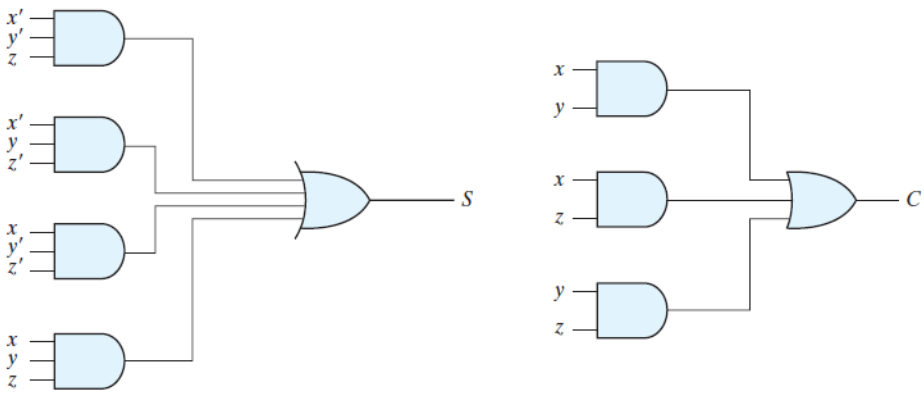


FIGURE 4.7
Implementation of full adder in sum-of-products form

Chapter 4 20

Full Adder as 2-Half Adders

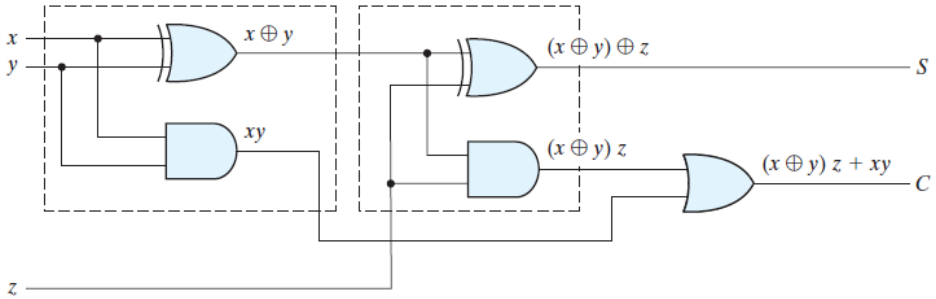


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

21

Binary Adders

- To add multiple operands, we "bundle" logical signals together into vectors and use functional blocks that operate on the vectors
- Example: 4-bit ripple carry adder: Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)
- Note: carry out of cell i becomes carry in of cell i + 1

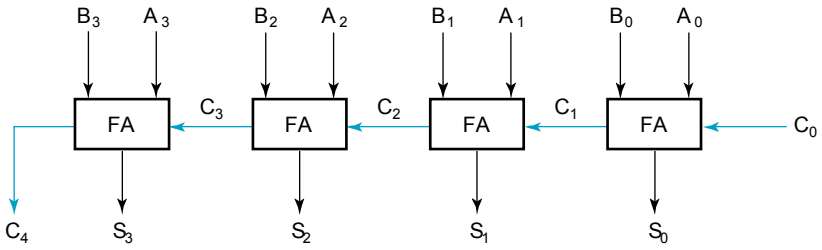
Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	Ci
Augend	1 0 1 1	Ai
Addend	0 0 1 1	Bi
Sum	1 1 1 0	Si
Carry out	0 0 1 1	Ci+1

Logic and Computer Design Fundamentals, 4e
PowerPoint® Slides
© 2008 Pearson Education, Inc.

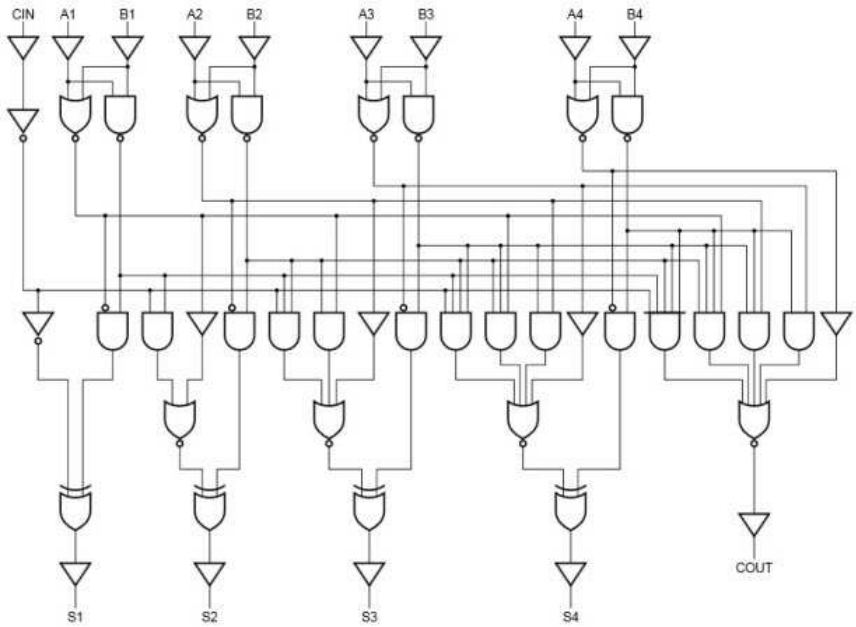
Chapter 4 22

4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:



Carry Propagation → Carry Lookahead Adder



Binary Subtraction: Signed Integers

- Positive numbers and zero can be represented by unsigned n-digit, radix *r* numbers. We need a representation for negative numbers.
- To represent a sign (+ or –) we need exactly one more bit of information (1 binary digit gives $2^1 = 2$ elements which is exactly what is needed).
- Since computers use binary numbers, by convention, the most significant bit is interpreted as a sign bit:

$$s\ a_{n-2} \ldots a_2 a_1 a_0$$

where:

s = 0 for Positive numbers

s = 1 for Negative numbers

and $a_i = 0$ or 1 represent the magnitude in some form.

Signed Integer Representations

- ***Signed-Magnitude*** – here the n – 1 digits are interpreted as a positive magnitude.
- ***Signed-Complement*** – here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:
  - ***Signed 1's Complement***
    - Uses 1's Complement Arithmetic
  - ***Signed 2's Complement***
    - Uses 2's Complement Arithmetic

Signed Integer Representation Example

- r =2, n=3

Number	Sign -Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
–0	100	111	—
–1	101	110	111
–2	110	101	110
–3	111	100	101
–4	—	—	100

Signed-Magnitude Arithmetic

- **If the parity of the three signs is 0:**
  1. Add the magnitudes.
  2. Check for overflow (a carry out of the MSB)
  3. The sign of the result is the same as the sign of the first operand.
- **If the parity of the three signs is 1:**
  1. Subtract the second magnitude from the first.
  2. If a borrow occurs:
     - take the two's complement of result
     - and make the result sign the complement of the sign of the first operand.
  3. Overflow will never occur.

Sign-Magnitude Arithmetic Examples

- **Example 1:** 0010 + 0101
- **Example 2:** 0010 + 1101
- **Example 3:** 1010 − 0101

Signed-Complement Arithmetic

- **Addition:**

  1. Add the numbers including the sign bits, discarding a carry out of the sign bits (2's Complement), or using an end-around carry (1's Complement).
  2. If the sign bits were the same for both numbers and the sign of the result is different, an overflow has occurred.
  3. The sign of the result is computed in step 1.

- **Subtraction:**

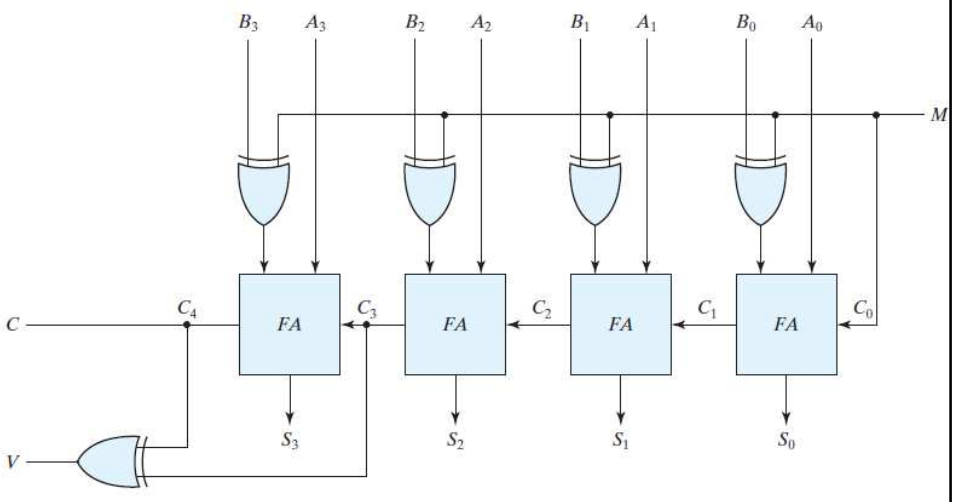
  Form the complement of the number you are subtracting and follow the rules for addition.

Signed 2's Complement Examples

- **Example 1: 1101**
  +0011

- **Example 2: 1101**
  −0011

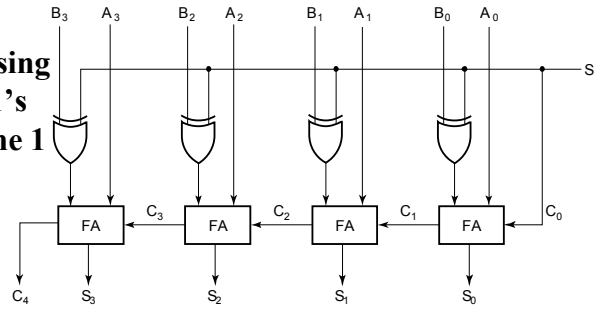
Binary Subtractor



32

2's Complement Adder/Subtractor

- Subtraction can be done by addition of the 2's Complement.
  1. Complement each bit (1's Complement.)
  2. Add 1 to the result.
- The circuit shown computes A + B and A – B:
- For S = 1, subtract, the 2's complement of B is formed by using XORs to form the 1's comp and adding the 1 applied to C_0.
- For S = 0, add, B is passed through unchanged



Overflow

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Overflow Detection

- *Overflow* occurs if $n+1$ bits are required to contain the result from an n-bit addition or subtraction
- Overflow can occur for:
  - Addition of two operands with the same sign
  - Subtraction of operands with different signs
- Signed number overflow cases with correct result sign

$$\begin{array}{rrrr} 0 & 0 & 1 & 1 \\ +0 & -1 & -0 & +1 \\ \hline 0 & 0 & 1 & 1 \end{array}$$

- Detection can be performed by examining the result signs which should match the signs of the top operand

Overflow Detection

- Signed number cases with carries C_n and C_{n-1} shown for correct result signs:

$$\begin{array}{rrrr} 0 & 0\,0 & 0\,1 & 1\,1 & 1 \\ 0 & 0 & 1 & 1 & \\ +0 & -1 & -0 & +1 & \\ \hline 0 & 0 & 1 & 1 & \end{array}$$

- Signed number cases with carries shown for erroneous result signs (indicating overflow):

$$\begin{array}{rrrr} 0 & 1\,0 & 1\,1 & 0\,1 & 0 \\ 0 & 0 & 1 & 1 & \\ +0 & -1 & -0 & +1 & \\ \hline 1 & 1 & 0 & 0 & \end{array}$$

- Simplest way to implement overflow $V = C_n \oplus C_{n-1}$
- This works correctly only if 1's complement and the addition of the carry in of 1 is used to implement the complementation! Otherwise fails for $-10\ldots0$

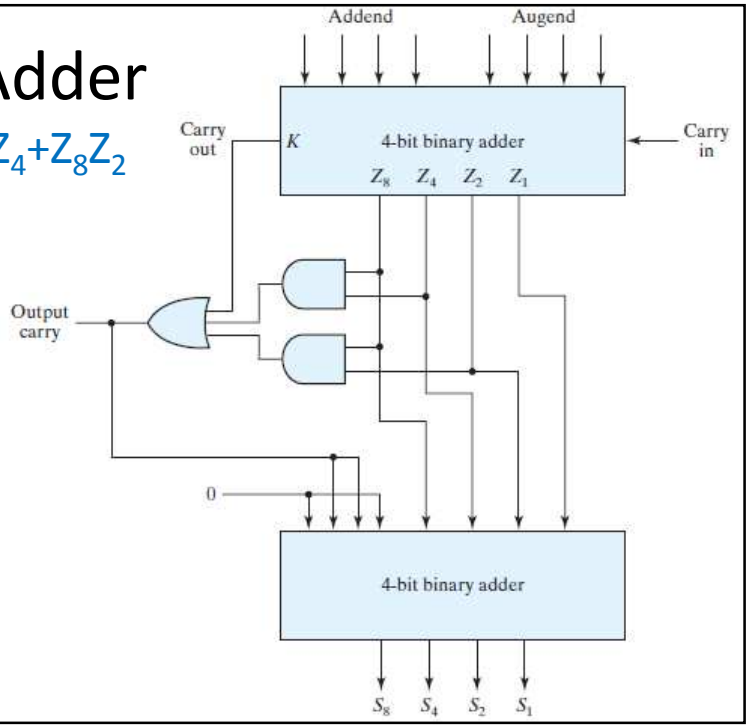
4.6 Decimal Adder: BCD Adder

Derivation of 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	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	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

BCD Adder

$C=K+Z_8Z_4+Z_8Z_2$



4.7 Binary Multiplication

The binary multiplication table is simple:

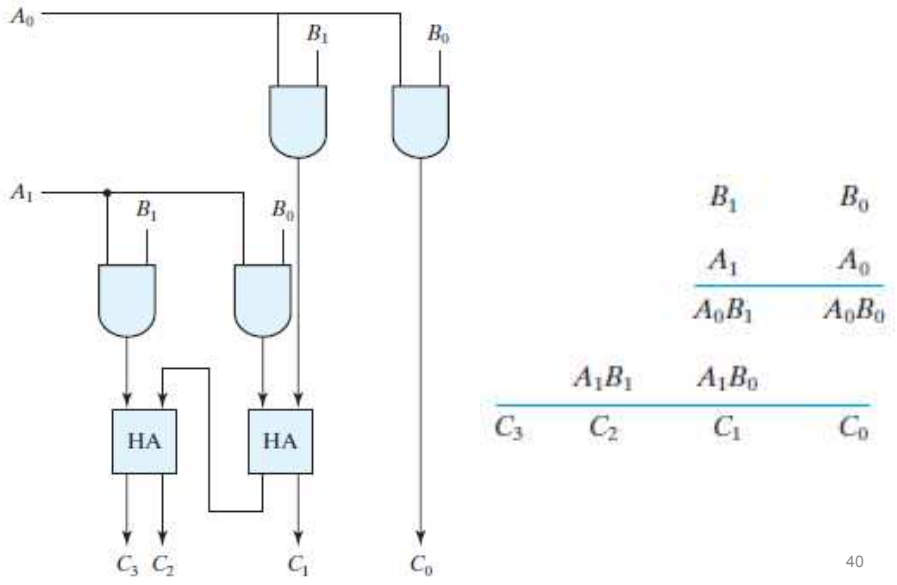
0 * 0 = 0 | 1 * 0 = 0 | 0 * 1 = 0 | 1 * 1 = 1

Extending multiplication to multiple digits:

Multiplicand	**1011**
Multiplier	**x 101**
Partial Products	**1011**
	0000 -
	1011 - -
Product	**110111**

39

4.7 Binary Multiplier



40

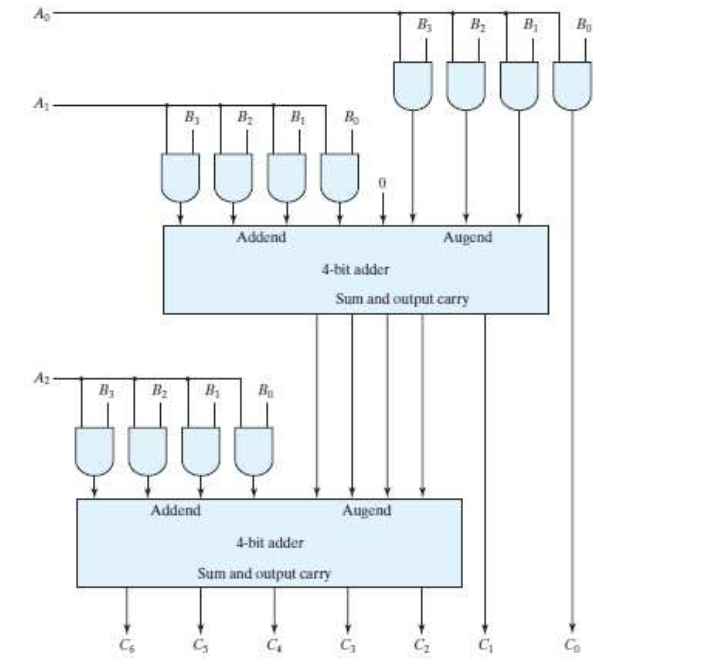
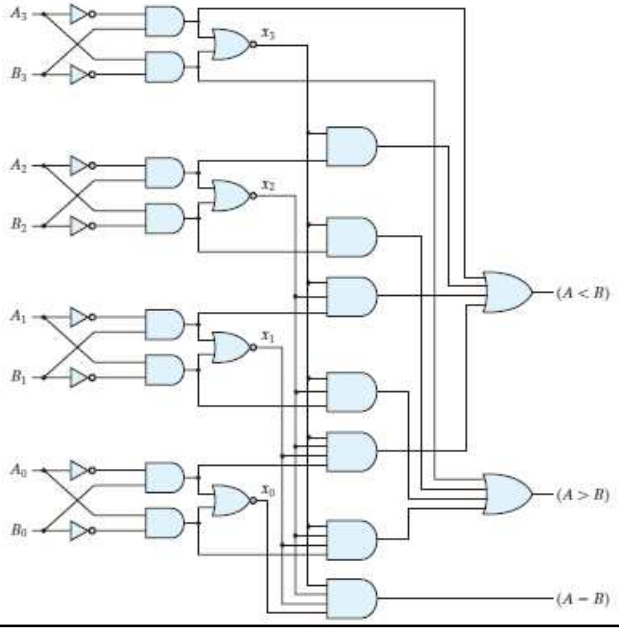


FIGURE 4.16
Four-bit by three-bit binary multiplier

41

4.8 Magnitude Comparator



42

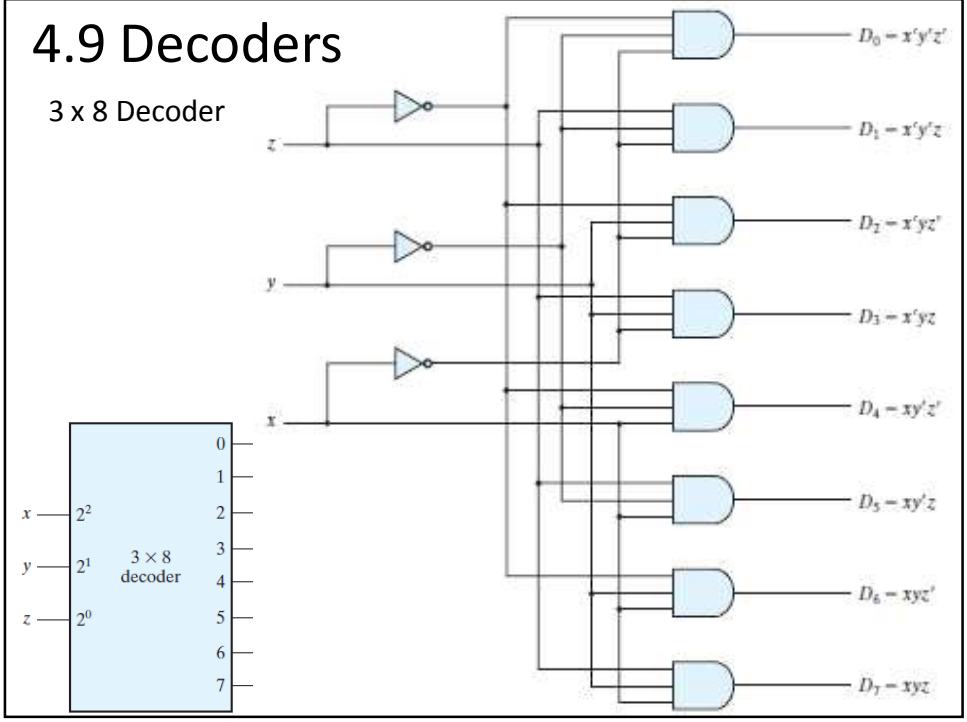
4.9 Decoders (3 x 8 Decoder)

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

43

4.9 Decoders

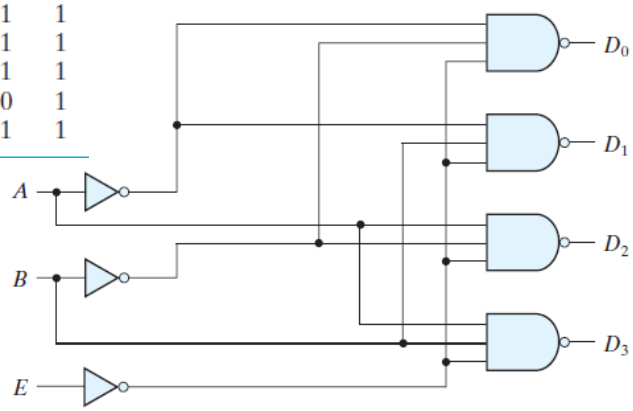
3 x 8 Decoder



Complemented Decoder

with Active Low Enable

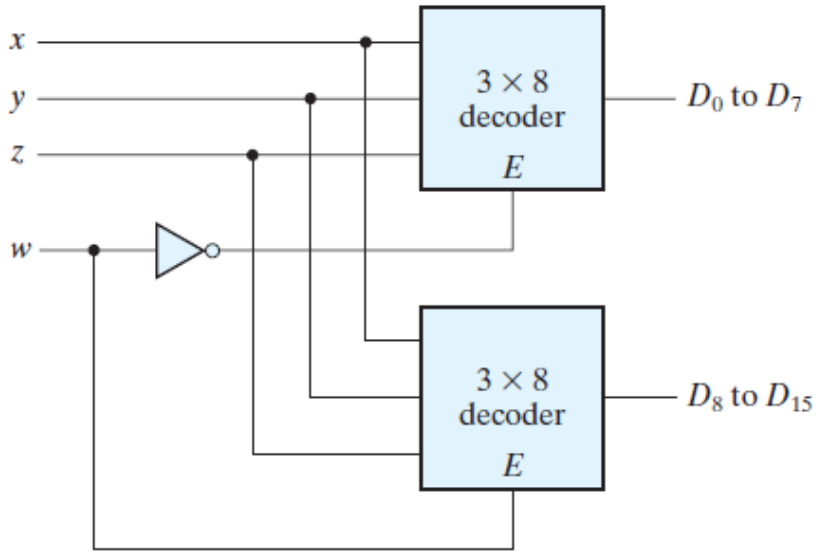
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1



45

4 x 16 Decoder

made from two 3 x 8 decoders

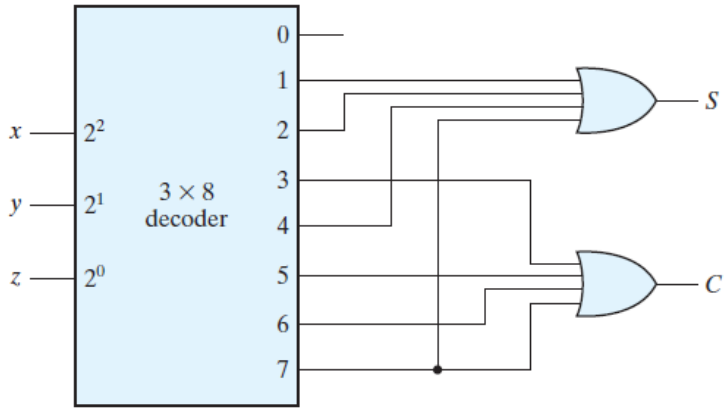


46

Implementation with Decoder

Ex: Full Adder Design

S(x,y,z) = ∑ (1,2,4,7) C(x,y,z) = ∑ (3,5,6,7)



47

4.10 Encoders

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

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

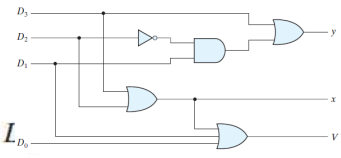
48

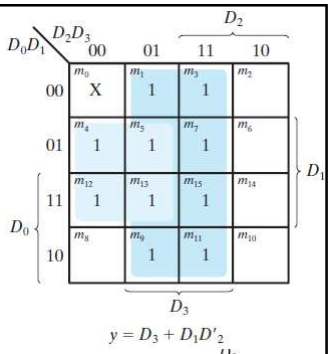
Priority Encoder

Truth Table of a Priority Encoder

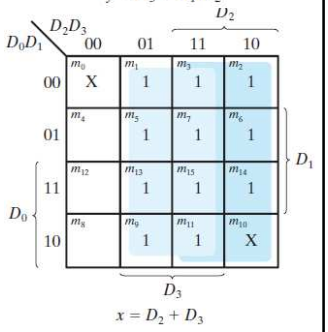
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D'_2$$
$$V = D_0 + D_1 + D_2 + D_3$$





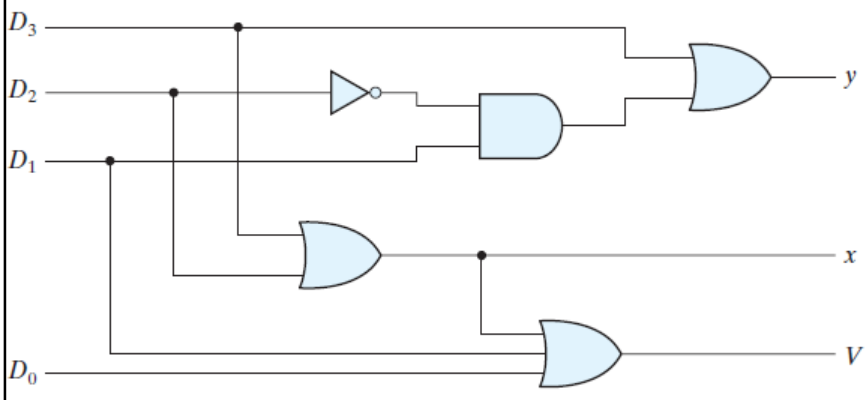
$y = D_3 + D_1D'_2$



$x = D_2 + D_3$

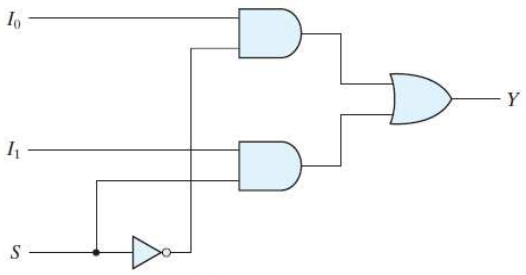
Implementation of Priority Encoder

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D'_2$$
$$V = D_0 + D_1 + D_2 + D_3$$

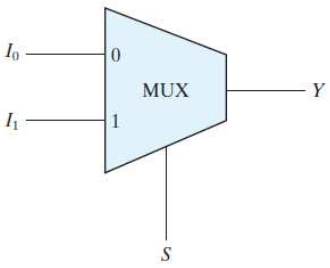


4.11 Multiplexers

2 x 1 Mux



(a) Logic diagram

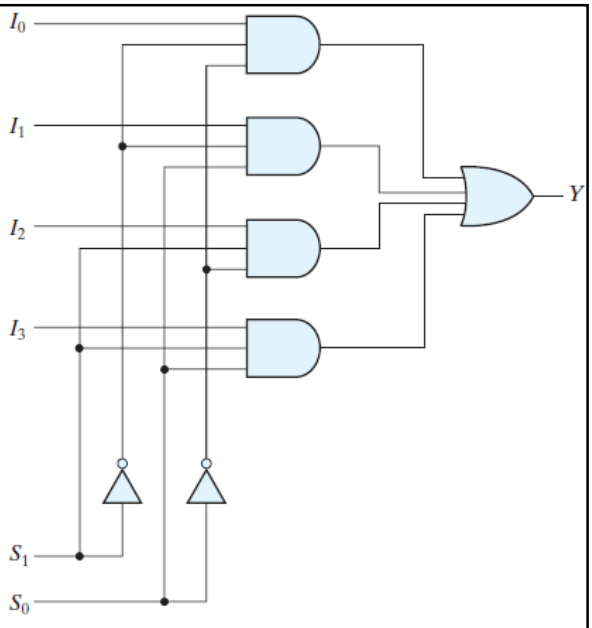


(b) Block diagram

51

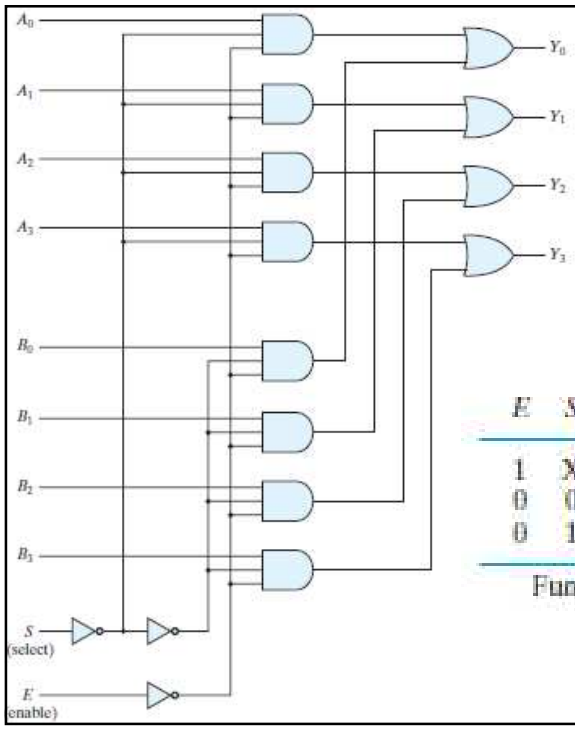
4 x 1 Mux

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



(a) Logic diagram

52



4-Bits 2x1 Mux

Quadruple 2x1 multiplexer

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

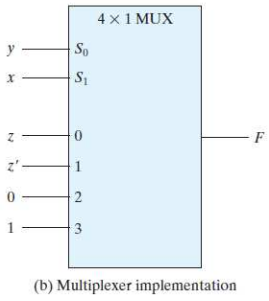
Function table

53

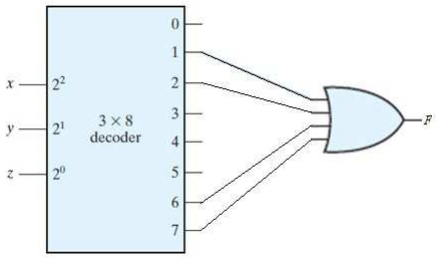
Boolean Function Implementation

F(x,y,z) = ∑ (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	



(b) Multiplexer implementation

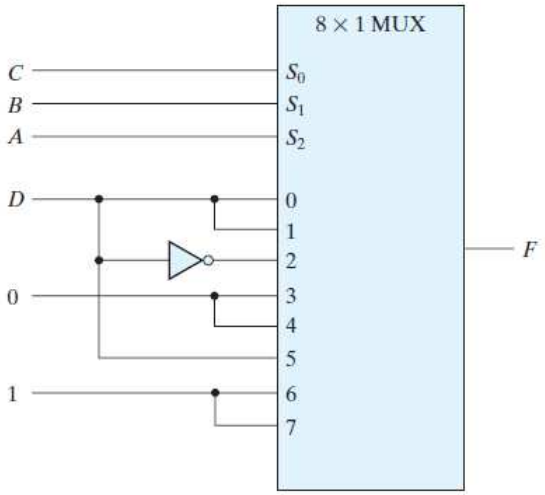


54

Example of Implementation with Mux

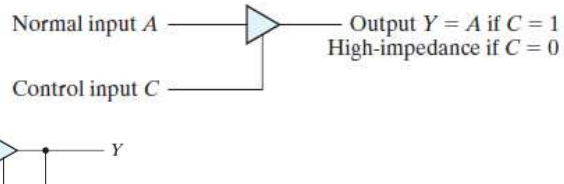
F(A,B,C,D)= ∑ (1,3,4,11,12,13,14,15)

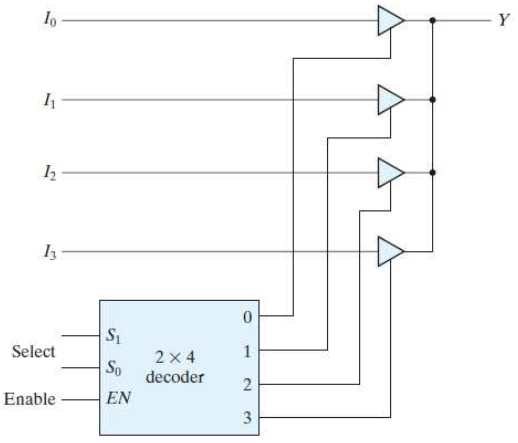
A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	



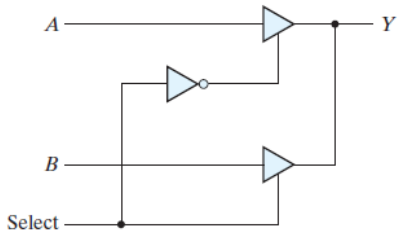
55

Three-State Gate (Tri-State Buffer)





(b) 4-to-1-line mux



(a) 2-to-1-line mux

56