Chapter 3 Combinational Logic Design

M. Morris Mano, Charles R. Kime. (2015). *Logic and computer design fundamentals* (5th ed.). Pearson.

Contents

Contents

1. Combinational Functional Blocks

- Corresponding to each of the functions is a combinational circuit implementation called a *functional block*.
- In the past, functional blocks were packaged as small-scale-integrated (SSI), medium-scale integrated (MSI), and large-scale-integrated (LSI) circuits.
- Today, they are often simply implemented within a very-large-scale-integrated (VLSI) circuit.

2. Rudimentary Logic Functions

- Functions of a single variable X
  - Four different functions are possible

TABLE 3-1
Functions of One Variable

X	F = 0	F = X	F = $\overline{X}$	F = 1
0	0	0	1	1
1	0	1	0	1
	Value fixing	***Transferring***	***Inverting***	***Value fixing***

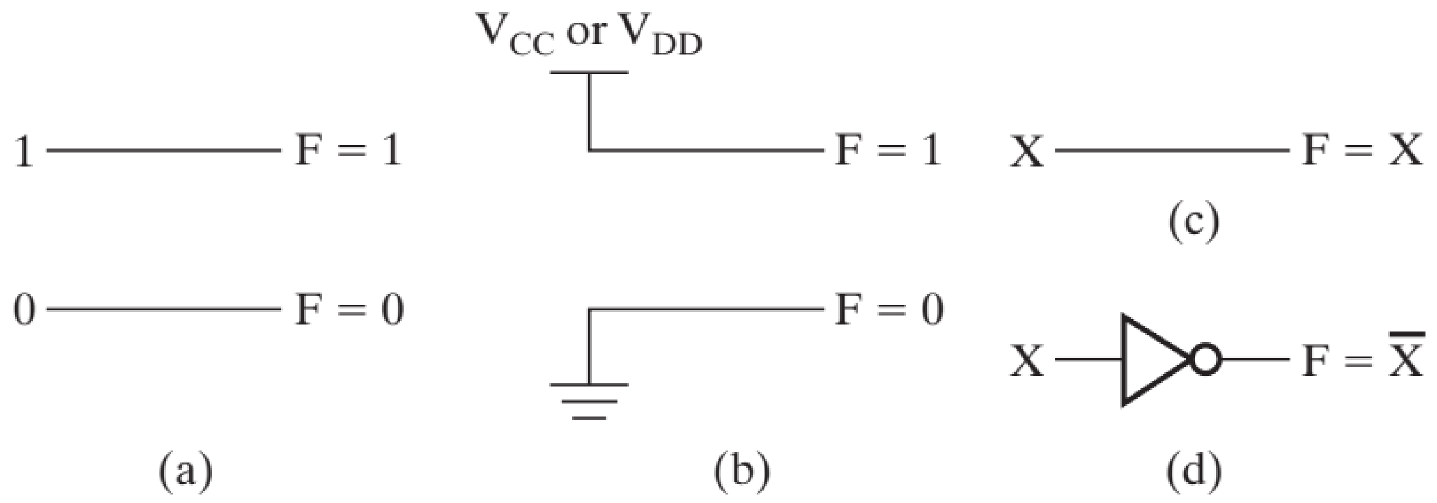
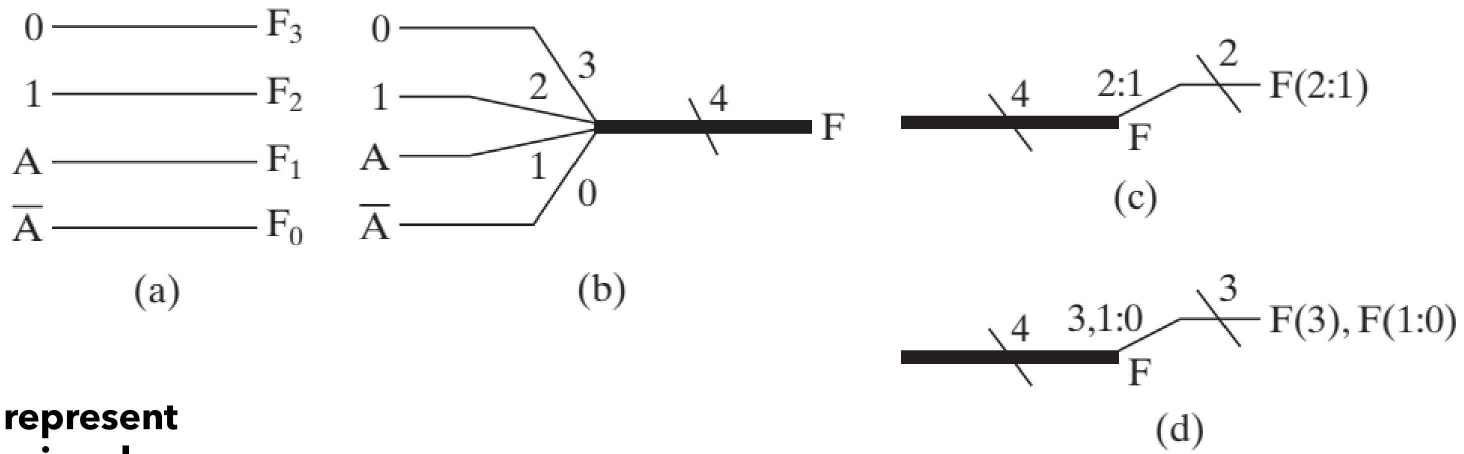


FIGURE 3-7
Implementation of Functions of a Single Variable *X*

Multiple-bit Rudimentary Functions

- Multiple-bit functions as vectors of single-bit functions
- We can order the four functions with F_3 as the most significant bit and F_0 the least significant bit, providing the vector $F = (F_3, F_2, F_1, F_0)$.
  - Suppose that F consists of rudimentary functions $F_3 = 0$, $F_2 = 1$, $F_1 = A$, and $F_0 = \overline{A}$. Then we can write F as the vector $(0, 1, A, \overline{A})$.



A wide line is used to represent a *bus* which is a vector signal.

FIGURE 3-8
Implementation of Multibit Rudimentary Functions

Multiple-bit Rudimentary Functions

- In (b) of the example, *F* = (F_3, F_2, F_1, F_0) is a bus.
- The bus can be split into individual bits as shown in (b)
- Sets of bits can be split from the bus as shown in (c) for bits 2 and 1 of *F*.
- The sets of bits need not be continuous as shown in (d) for bits 3, 1, and 0 of *F*.

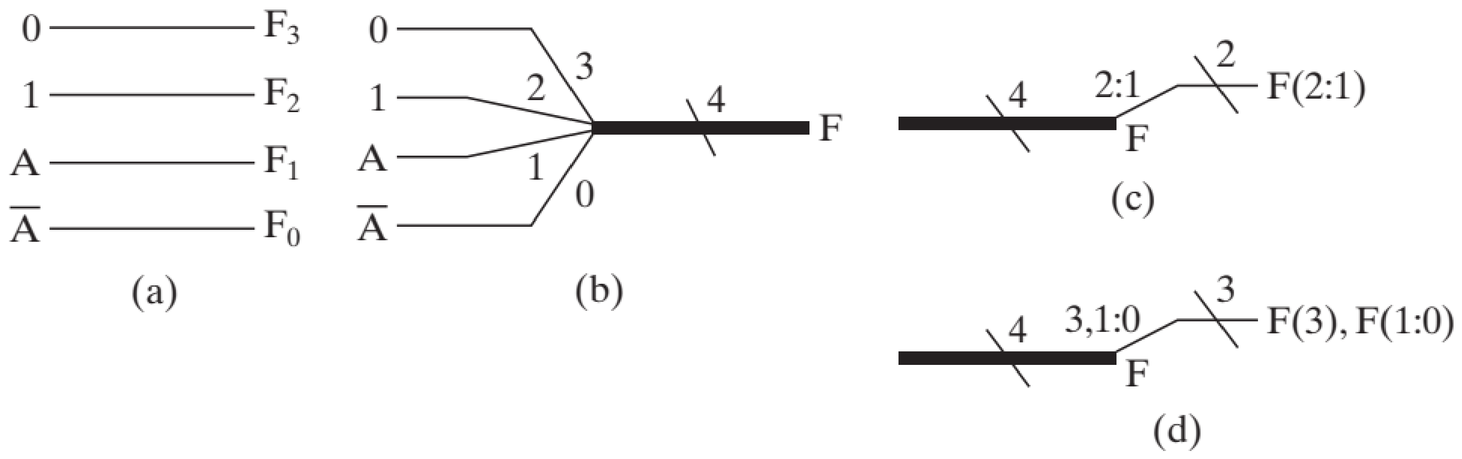


FIGURE 3-8
Implementation of Multibit Rudimentary Functions

Enabling

- *Enabling* permits an input signal to pass through to an output.
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value.
  - The value on the output when it is disable can be Hi-Z, 0 , or 1. (disabled value)

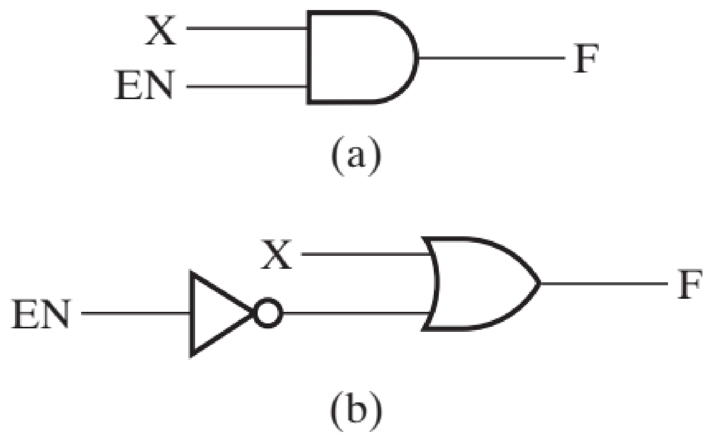


FIGURE 3-10
Enabling Circuits

Figure (a) : Disabled value is 0

If EN is 1, the input X reaches the output (enabled).

If EN is 0 (disabled) , the output is fixed at 0.

Figure (b) : Disabled value is 1

If EN is 1, the input X reaches the output (enabled).

If EN is 0 (disabled) , the output is fixed at 1.

Enabling

- Example 3-5. **Car Electrical Control using Enabling**
  - The problem: In most automobiles, the lights, radio, and power windows operate only if the ignition switch is turned on. In this case, the ignition switch acts as an "enabling" signal. Suppose that we model this automotive subsystem using the following variables and definitions:
  - Input Switches (inputs)
    - Ignition switch IG: Value 0 if off and value 1 if on
    - Light switch LS: Value 0 if off and value 1 if on
    - Radio switch RS: Value 0 if off and value 1 if on
    - Power window switch WS: Value 0 if off and value 1 if on
  - Accessory Control (outputs)
    - Lights L: Value 0 if off and value 1 if on
    - Radio R: Value 0 if off and value 1 if on
    - Power windows W: Value 0 if off and value 1 if on

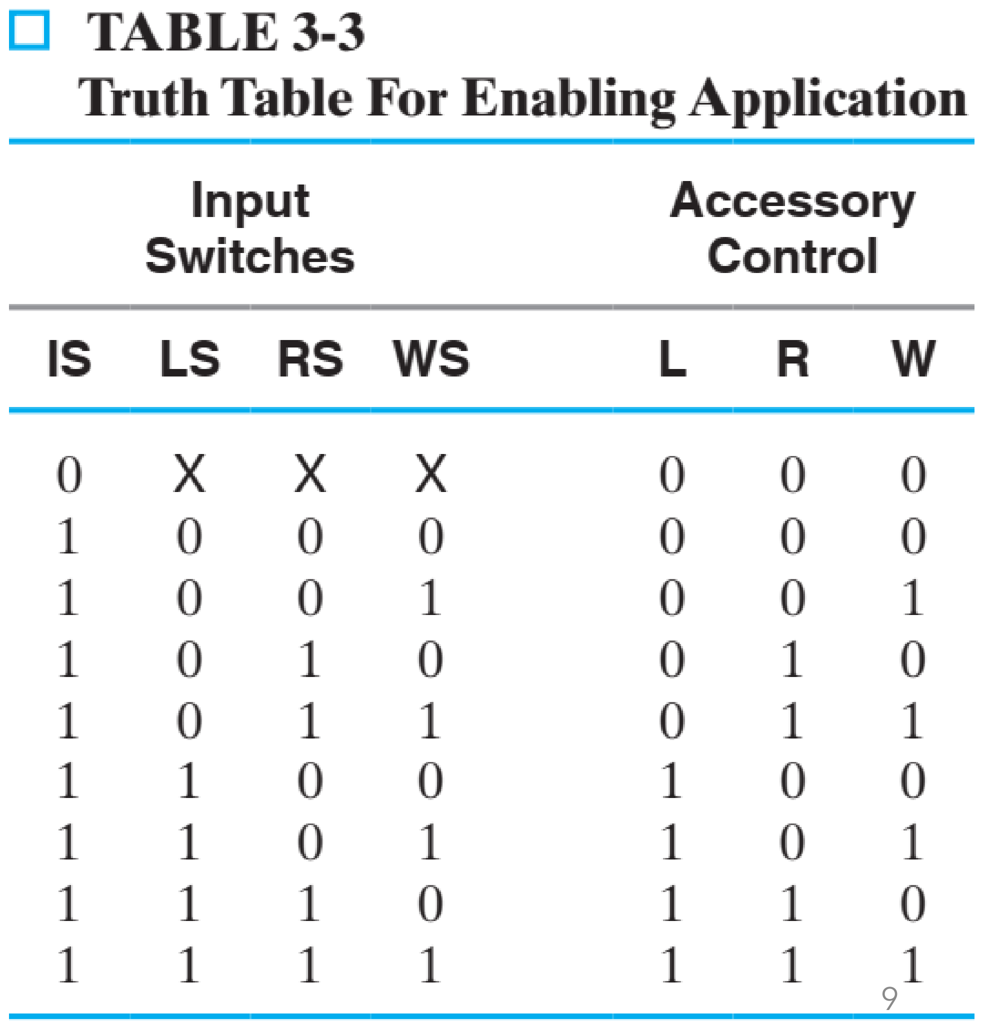


TABLE 3-3
Truth Table For Enabling Application

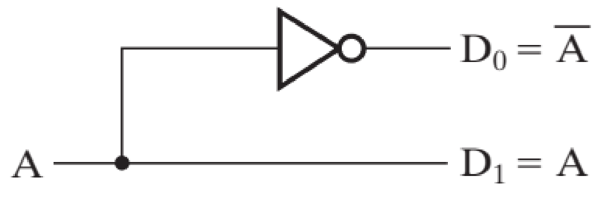
Input Switches				Accessory Control		
IS	LS	RS	WS	L	R	W
0	X	X	X	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

3. Decoding

- Decoding is the conversion of an *n*-bit input code to an *m*-bit output code with $n \le m \le 2^n$, such that each valid input code word produces **a unique output code**.
- Circuits that perform decoding are called *decoders*.
- **1-to-2-Line Decoder**

A	D_0	D_1
0	1	0
1	0	1

(a)



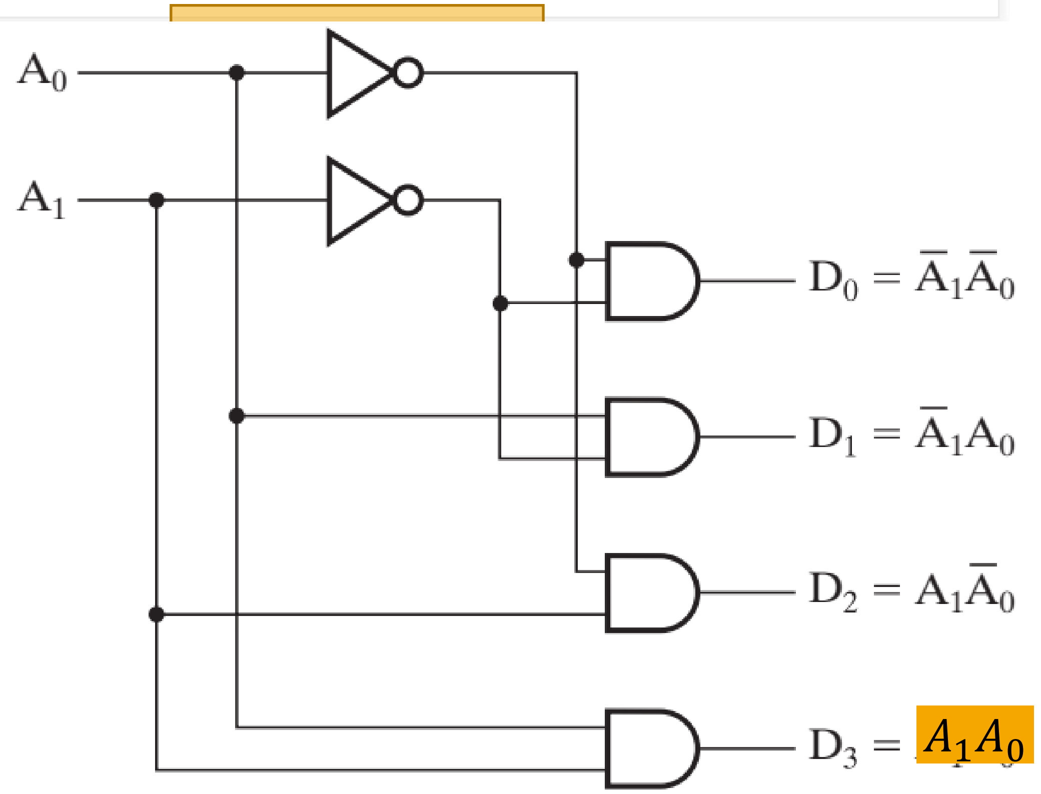
(b)

FIGURE 3-12
A 1–to–2-Line Decoder

3. Decoding

- **2-to-4-Line Decoder**

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

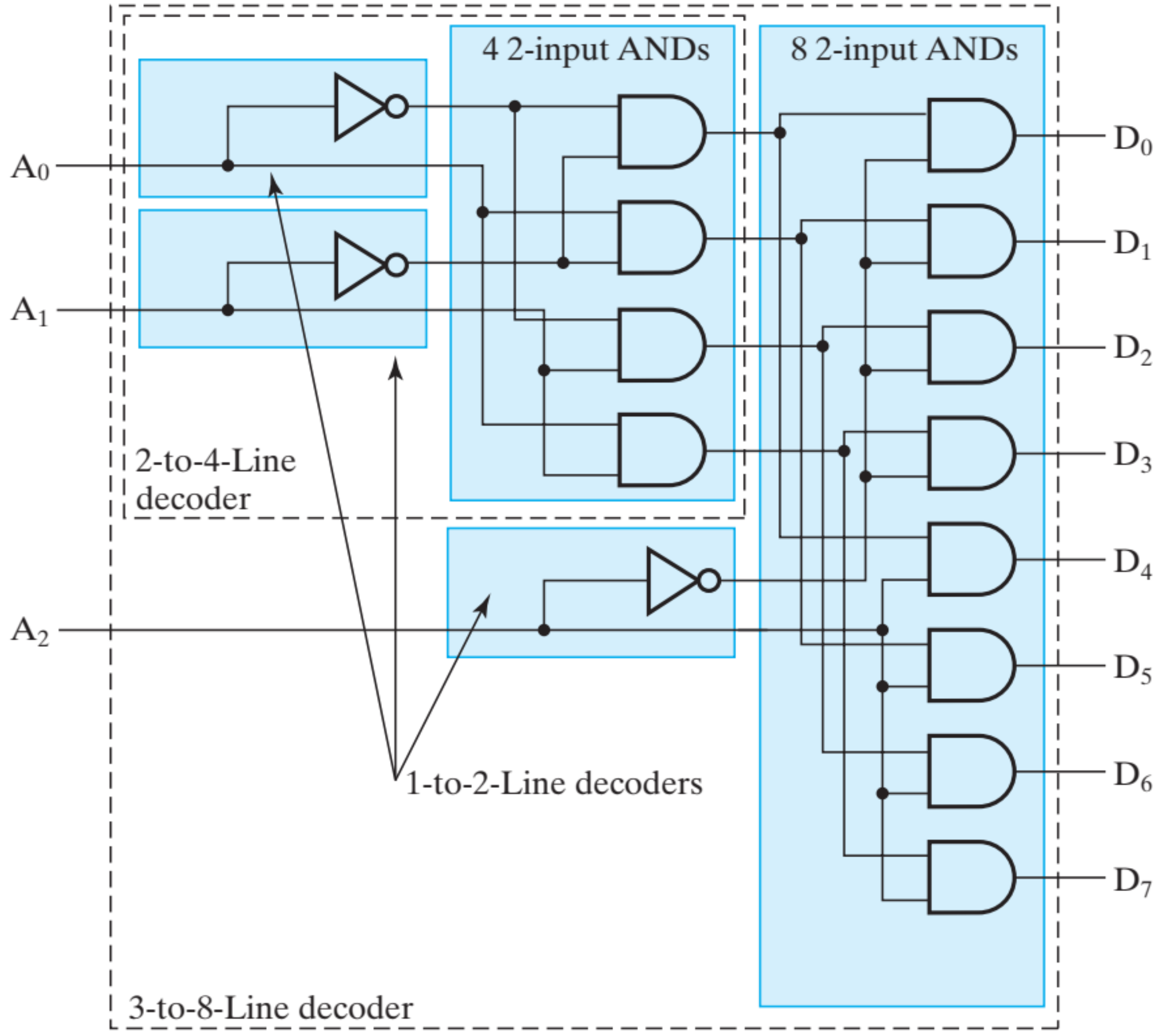


Note that the 2-4-line decoder made up of 2 1-to-2-line decoders and 4 AND gates.

3. Decoding

- **3-to-8-Line Decoder**

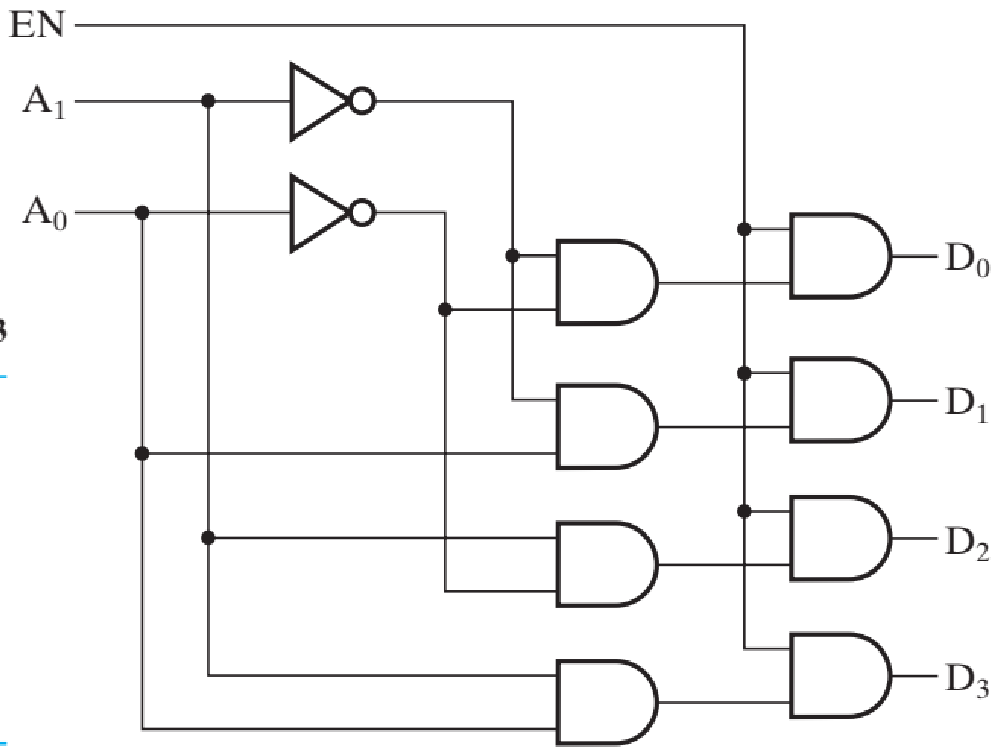
A_2	A_1	A_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Decoder and Enabling Combinations

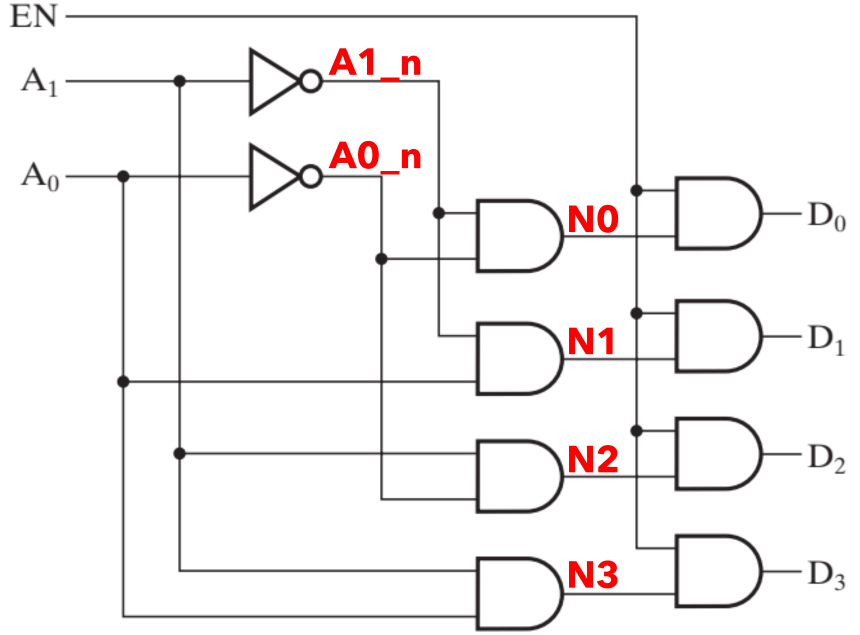
- See truth table below for function
  - Note use of X's to denote both 0 and 1
  - Combination containing two X's represent four binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs
  - In this case, called a *demultiplexer*

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



Structural Verilog Description of 2–to–4-Line Decoder

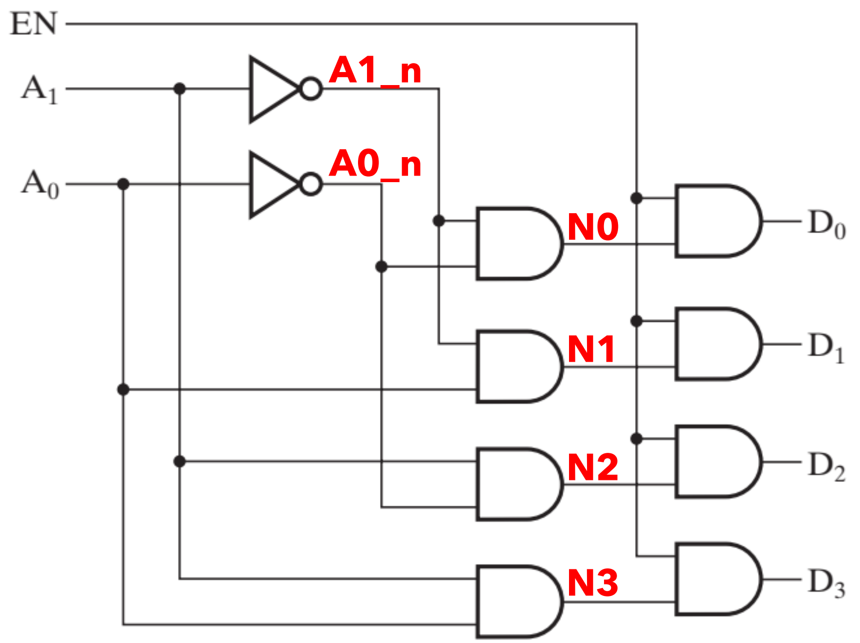
EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



```
// 2-to-4-Line Decoder with Enable: Structural Verilog Desc.
// (See Figure 3-16 for logic diagram)
module decoder_2_to_4_st_v (EN, A0, A1, D0, D1, D2, D3);
   input EN, A0, A1;
   output D0, D1, D2, D3;

   wire A0_n, A1_n, N0, N1, N2, N3;
   not
      g0(A0_n, A0),
      g1(A1_n, A1);
   and
      g3(N0, A0_n, A1_n),
      g4(N1, A0, A1_n),
      g5(N2, A0_n, A1),
      g6(N3, A0, A1),
      g7(D0, N0, EN),
      g8(D1, N1, EN),
      g9(D2, N2, EN),
      g10(D3, N3, EN);
endmodule
```

Dataflow Verilog Description of 2-to-4-Line Decoder

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



```
// 2-to-4-Line Decoder with Enable: Dataflow Verilog Desc.
// (See Example 3-16 for logic diagram)
module decoder_2_to_4_df_v(EN, A0, A1, D0, D1, D2, D3);
   input EN, A0, A1;
   output D0, D1, D2, D3;

   assign D0 = EN & ~A1 & ~A0;
   assign D1 = EN & ~A1 & A0;
   assign D2 = EN & A1 & ~A0;
   assign D3 = EN & A1 & A0;

endmodule
```

Decoder-Based Combinational Circuits

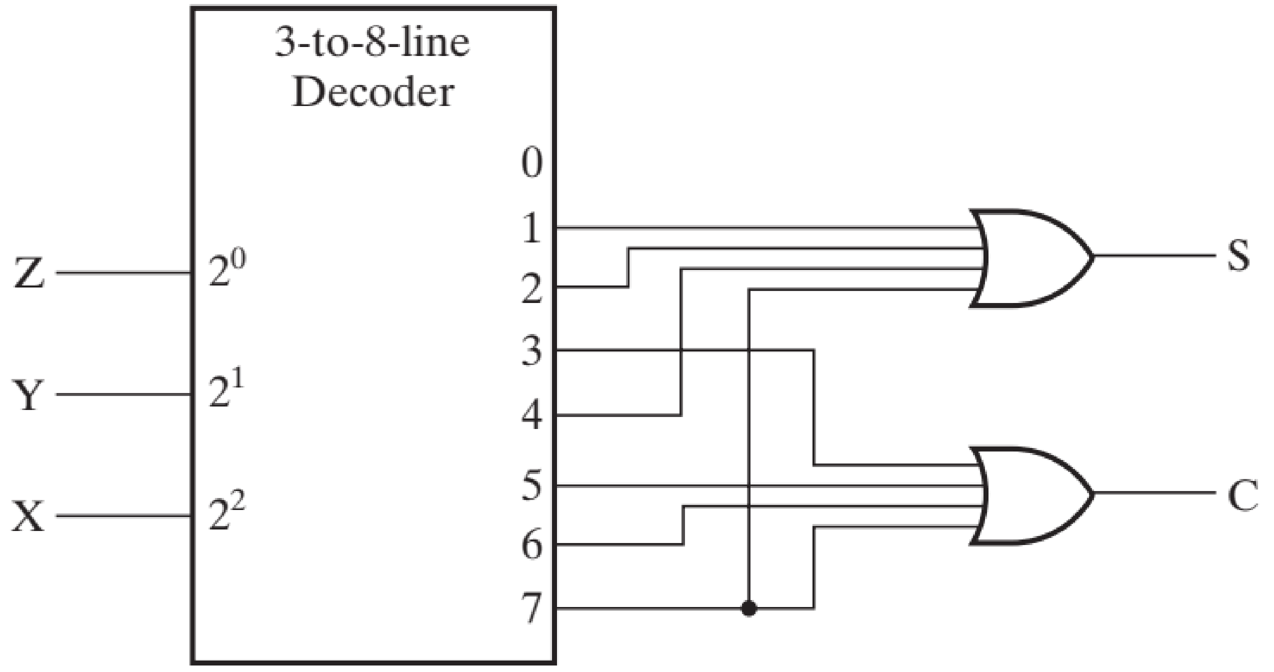
- Since any Boolean function can be expressed as a sum of minterms, one can use a decoder to generate the minterms and combine them with an external OR gate to form a sum-of-minterms implementation.
- In this way, any combinational circuit with n inputs and *m* outputs can be implemented with an *n*–to–2^n-line decoder and *m* OR gates.

Decoder-Based Combinational Circuits

- For example: Decoder and Or-gate implementation of a Binary Adder Bit

□ **TABLE 3-4**
Truth Table for 1-Bit Binary 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



□ **FIGURE 3-21**
Implementing a Binary Adder Using a Decoder

4. Encoding

- An encoder is a digital function that performs the inverse operation of a decoder.
- An encoder has 2^n (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value.

4. Encoding

- An example of an encoder is the octal-to-binary encoder.

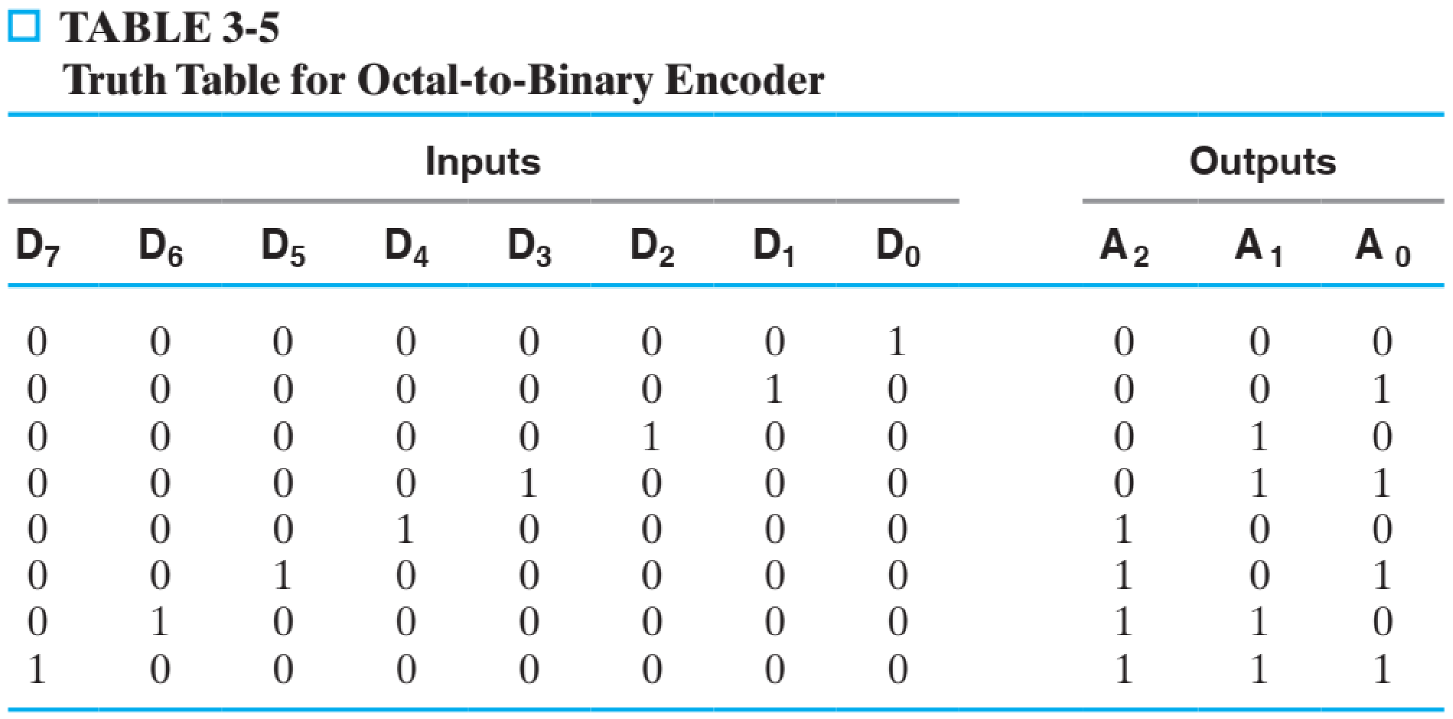


TABLE 3-5
Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

Priority Encoder

- A priority encoder is a combinational circuit that implements a priority function.
- The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority takes precedence.
- The valid output designated by V is set to 1 only when one or more of the inputs are equal to 1.

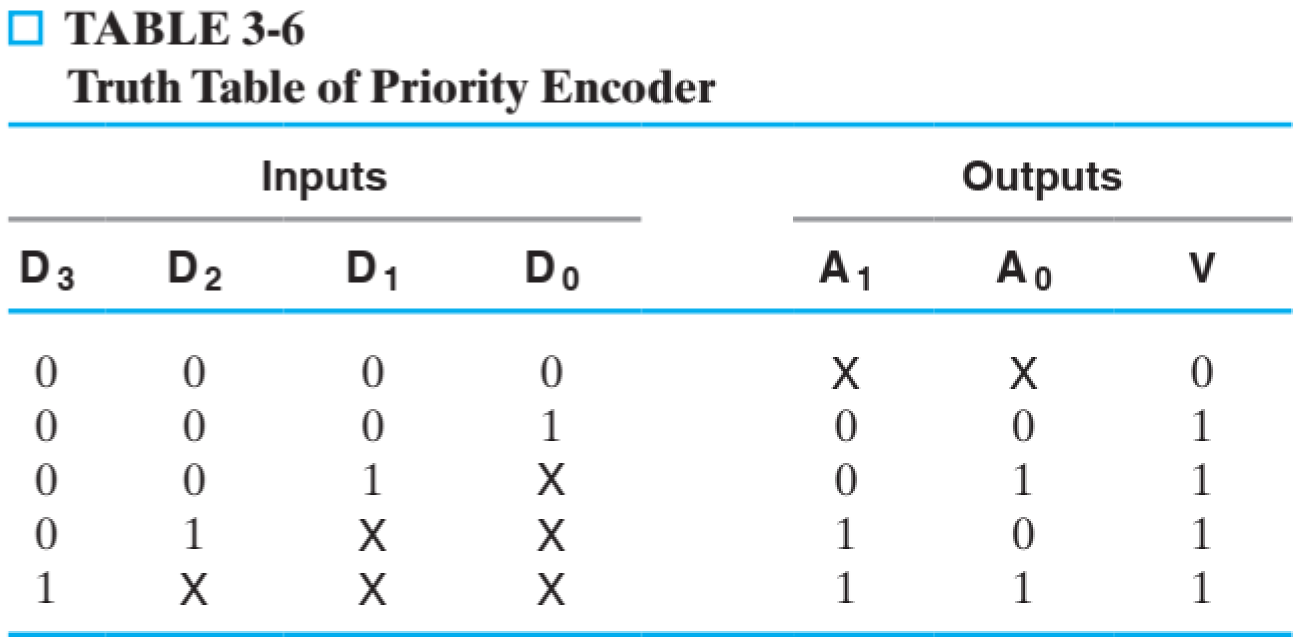


TABLE 3-6
Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Among the 1s that appear, it selects the **most significant input position**.

5. Selecting

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
  - A set of information inputs from which the selection is made
  - A single output
  - A set of control lines for making the selection
- Logic circuits that perform selecting are called *multiplexer*. Selecting can also be done by three-state logic or transmission gates.

Multiplexers

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs the information to a single output line.
- The selection of a particular input line is controlled by a set of input variables, called *selection inputs*.
- Normally, there are 2^n input lines and n selection inputs whose bit combinations determine which input is selected.

2-to-1-Line Multiplexer

- The single selection variable S has two values: (n = 1)
  - $S = 0$ selects input I_0
  - $S = 1$ selects input I_1
- The equation:

$$Y = \overline{S}I_0 + SI_1$$

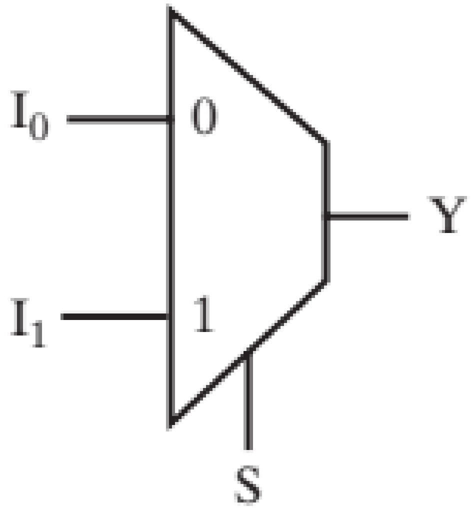


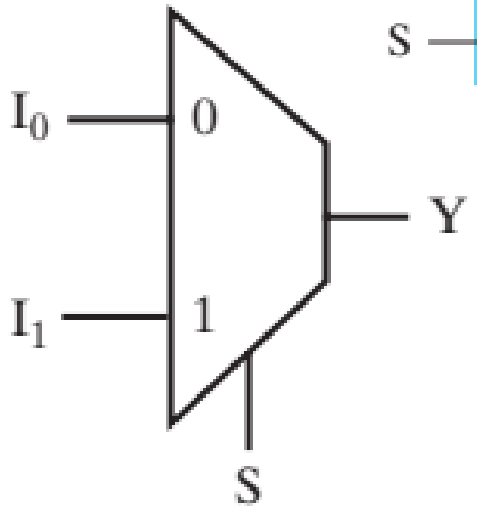
TABLE 3-7
Truth Table for 2–to–1-Line Multiplexer

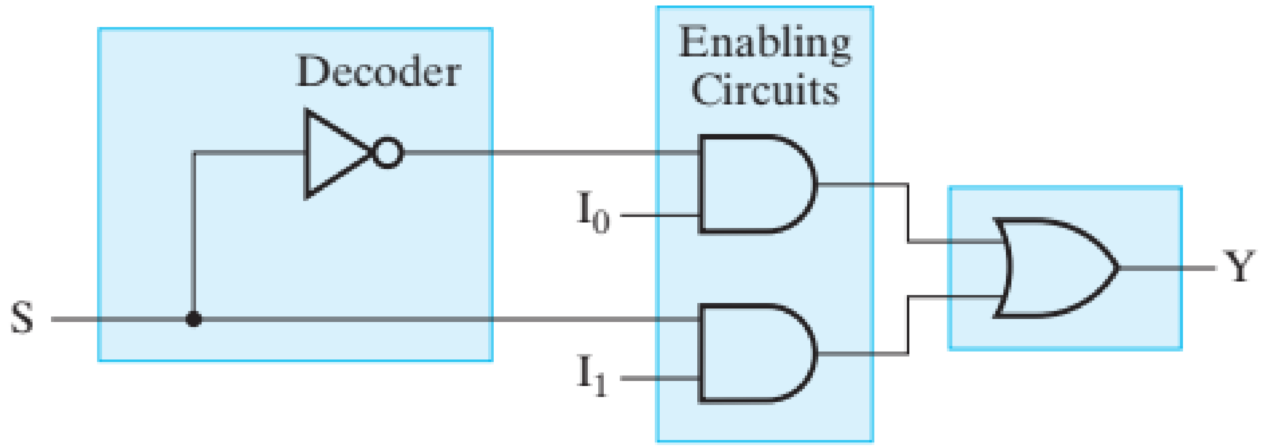
S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

2-to-1-Line Multiplexer

- The single selection variable S has two values: (n = 1)
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- The equation:

$$Y = \overline{S}I_0 + SI_1$$



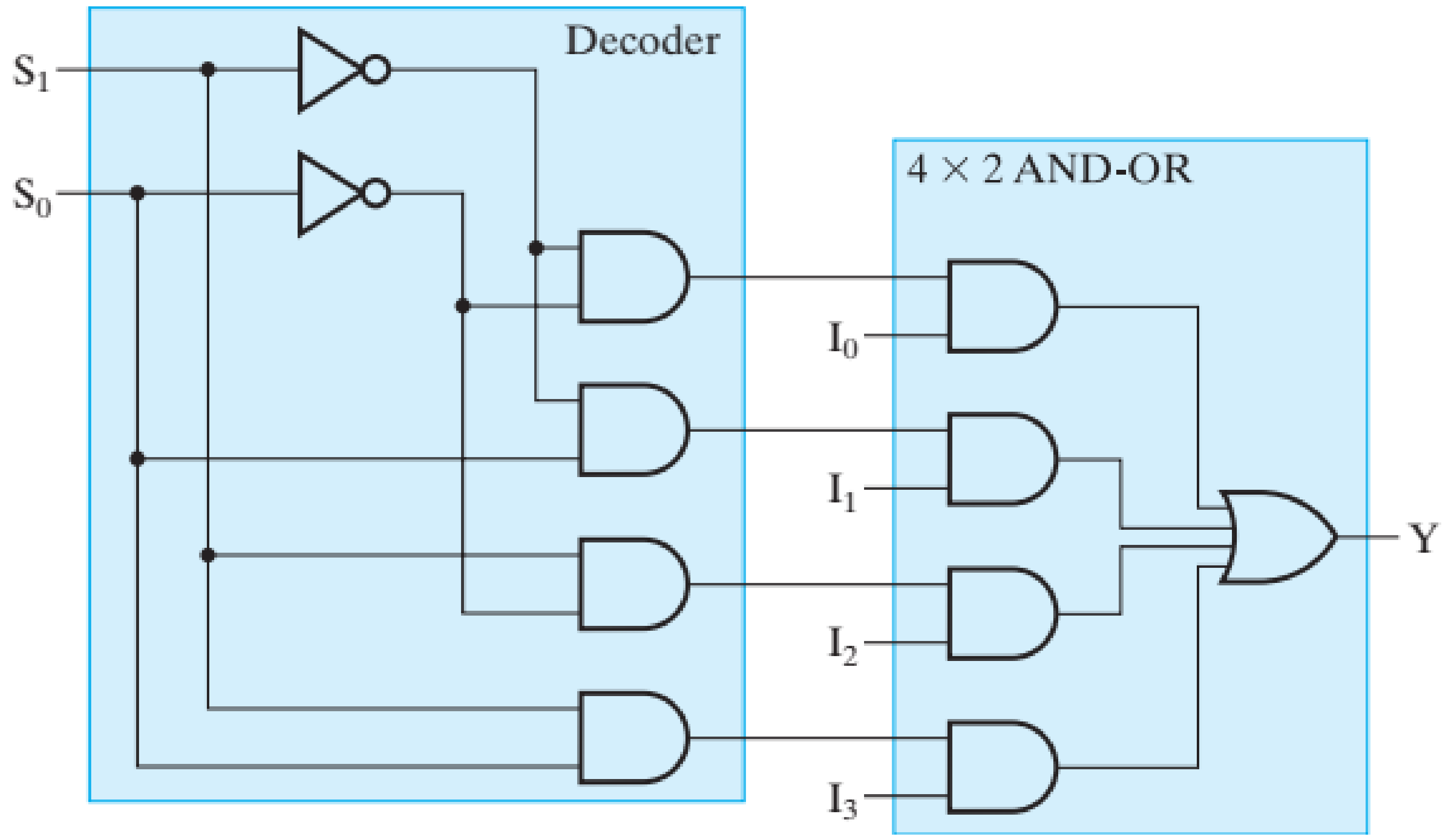


2^n-to-1-Line Multiplexer

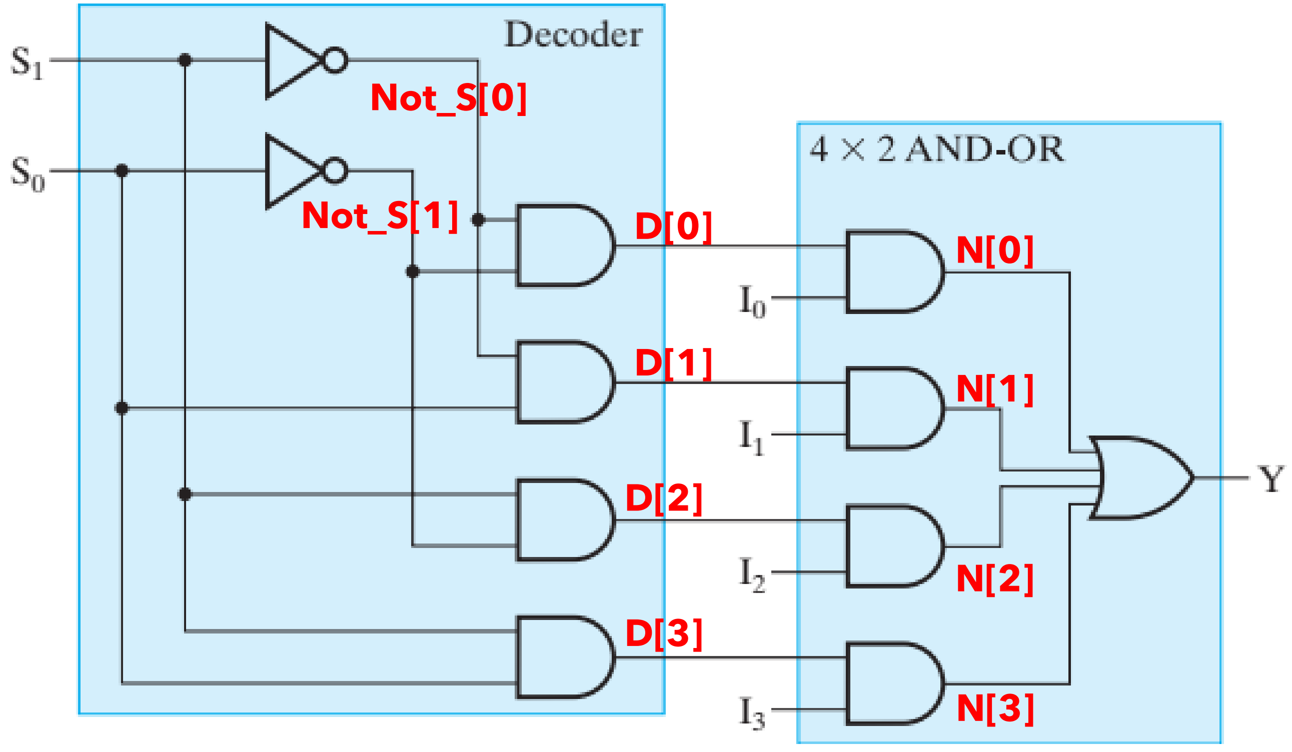
- In general, for an 2^n-to-1-line multiplexer:
  - n-to-2^n-line decoder
  - 2^n x 2 AND-OR

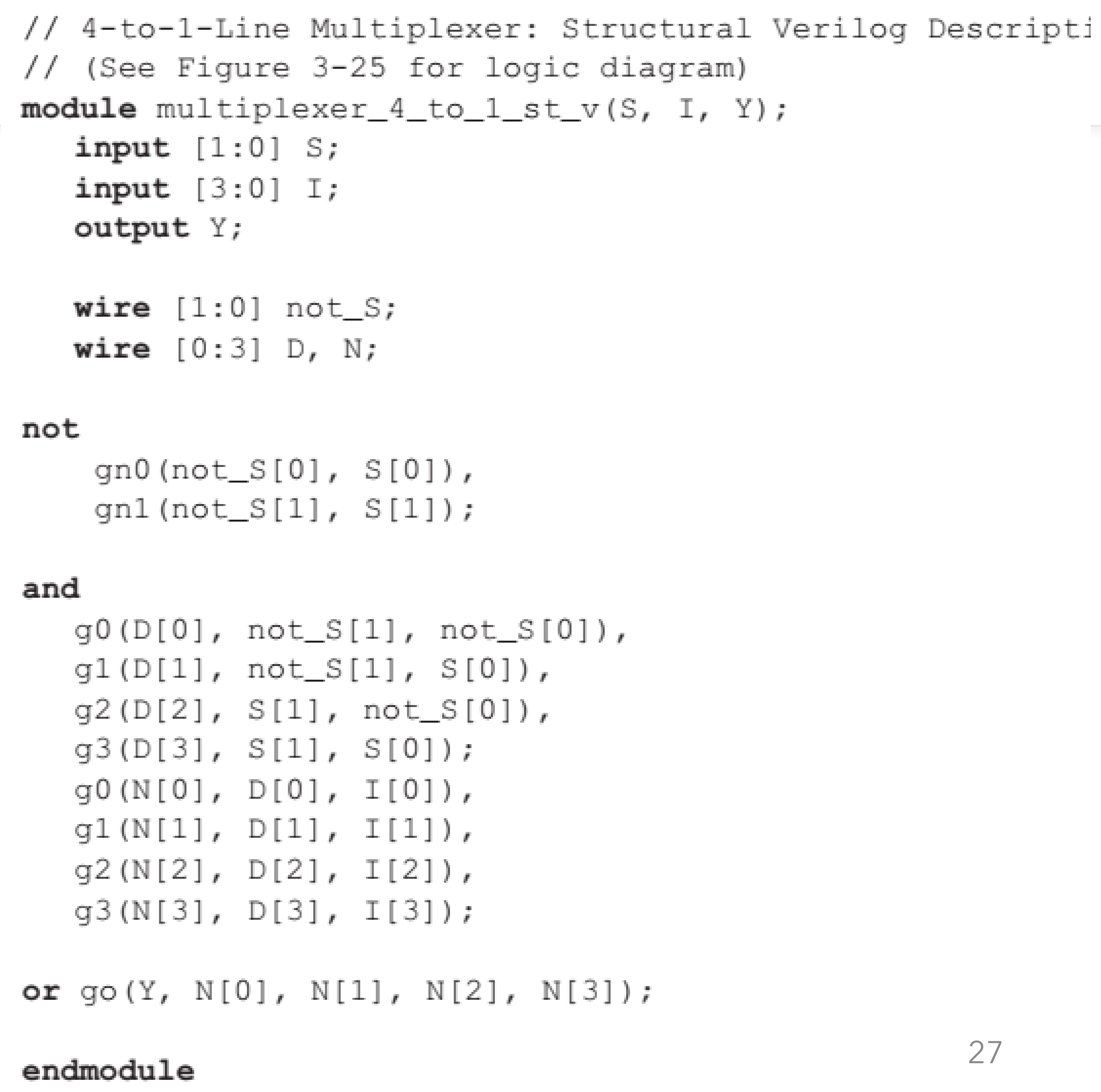
4-to-1-Line Multiplexer

- 2-to-2^2-line decoder
- $2^2 \times$ 2 AND-OR



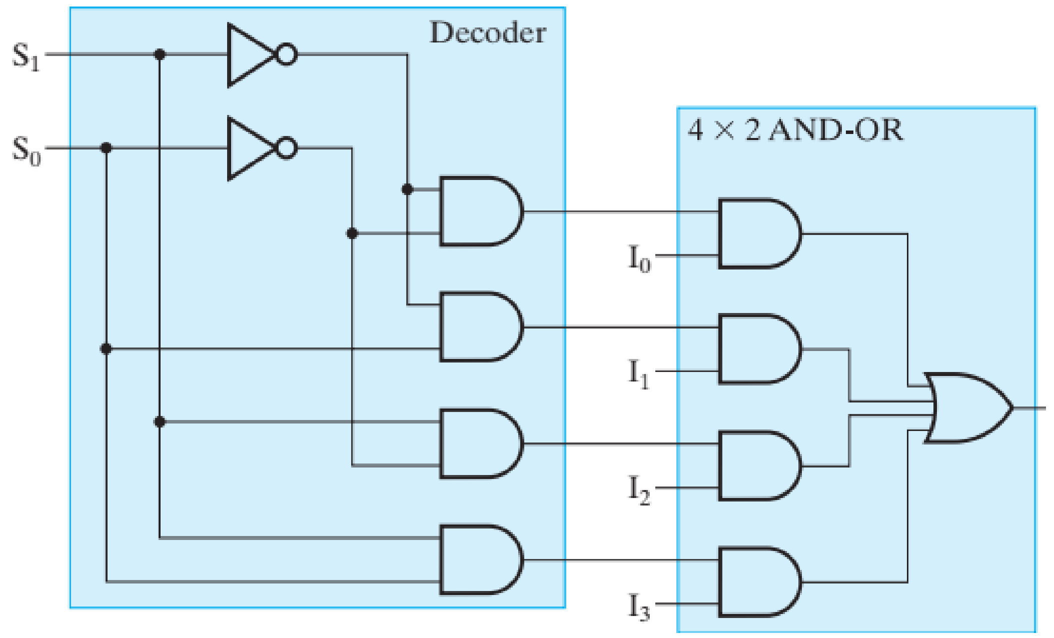
Structural Verilog Description for a 4-to-1-Line Multiplexer





```
// 4-to-1-Line Multiplexer: Structural Verilog Descripti
// (See Figure 3-25 for logic diagram)
module multiplexer_4_to_1_st_v(S, I, Y);
   input [1:0] S;
   input [3:0] I;
   output Y;

   wire [1:0] not_S;
   wire [0:3] D, N;

not
    gn0(not_S[0], S[0]),
    gn1(not_S[1], S[1]);

and
   g0(D[0], not_S[1], not_S[0]),
   g1(D[1], not_S[1], S[0]),
   g2(D[2], S[1], not_S[0]),
   g3(D[3], S[1], S[0]);
   g0(N[0], D[0], I[0]),
   g1(N[1], D[1], I[1]),
   g2(N[2], D[2], I[2]),
   g3(N[3], D[3], I[3]);

or go(Y, N[0], N[1], N[2], N[3]);

endmodule
```

Dataflow Verilog Description for a 4-to-1-Line Multiplexer



```
// 4-to-1-Line Multiplexer: Dataflow Verilog Description
// (See Table 3-8 for function table)
module multiplexer_4_to_1_tf_v(S, I, Y);
   input [1:0] S;
   input [3:0] I;
   output Y;

   assign Y = S[1] ? (S[0] ? I[3] : I[2]) :
                     (S[0] ? I[1] : I[0]);
endmodule
```

6. Iterative Combinational Circuits

- Arithmetic functions
  - Operate on binary vectors
  - Use the same subfunction in each bit position
- Can design functional block for subfunction and repeat to obtain functional block for overall function
  - Cell - subfunction block
  - Iterative array - a array of interconnected cells
- An iterative array can be in a single dimension (1D) or multiple dimensions

6. Iterative Combinational Circuits

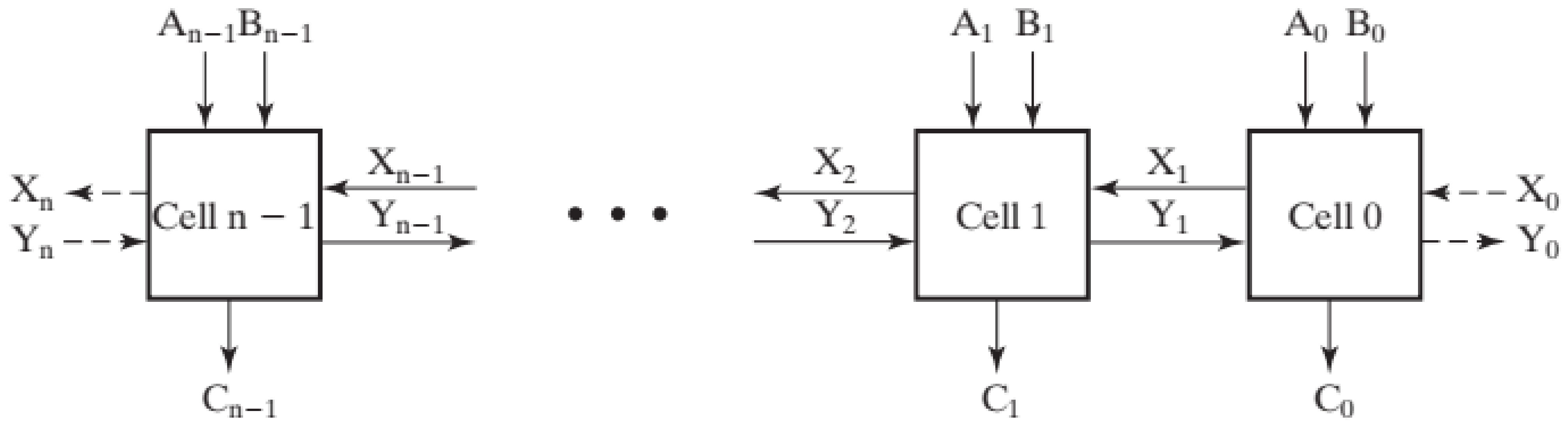


FIGURE 3-39
Block Diagram of an Iterative Circuit

6. Binary Adders

- Half-Adder (HA) is an arithmetic circuit that generates the sum of two binary digits.
  - The circuit has two inputs and two outputs.
- Full-Adder (FA) is a combinational circuit that forms the arithmetic sum of three input bits.
  - The circuit has three inputs and two outputs.
- Ripple Carry Adder, an iterative array to perform binary addition.

Half Adder

- A half adder is an arithmetic circuit that generates the sum of two binary digits. The circuit has two inputs and two outputs.
  - We assign the symbols X and Y to the two inputs and S (for "sum") and C (for "carry") to the outputs.

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

Half Adder

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

TABLE 3-11
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S		Y	
		0	1
X	0		**1**
	1	**1**	

C		Y	
		0	1
X	0		
	1		**1**

K-Maps

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

$$C = XY$$

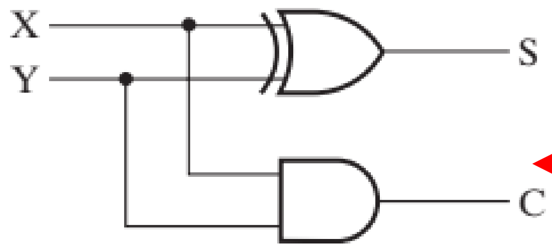


FIGURE 3-40
Logic Diagram of Half Adder

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
  - For a carry-in Z of 1

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
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

Full 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

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

TABLE 3-12
Truth Table of Full Adder

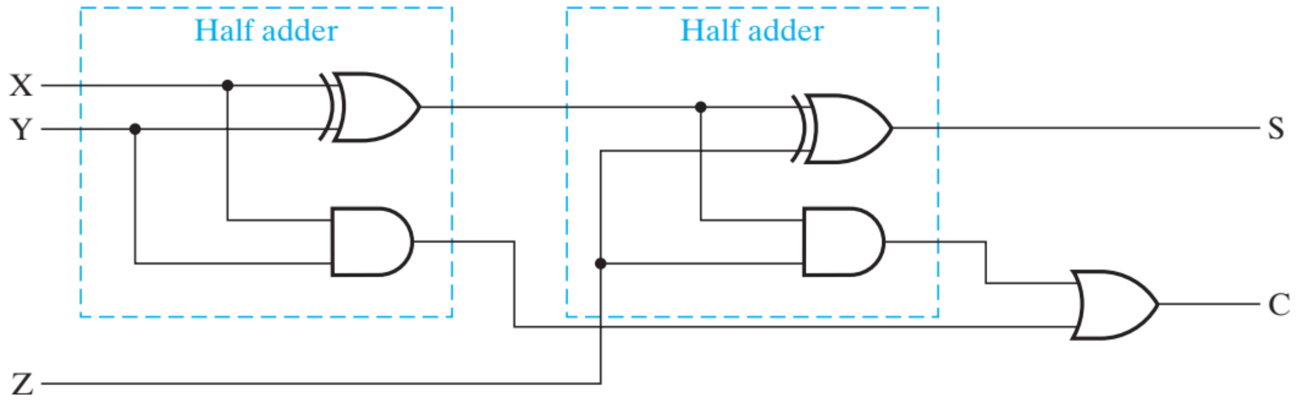
Inputs			Outputs	
X	**Y**	**Z**	**C**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S		YZ			
		00	01	11	10
X	0		1		1
	1	1		1	

C		YZ			
		00	01	11	10
X	0			1	
	1		1	1	1

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$

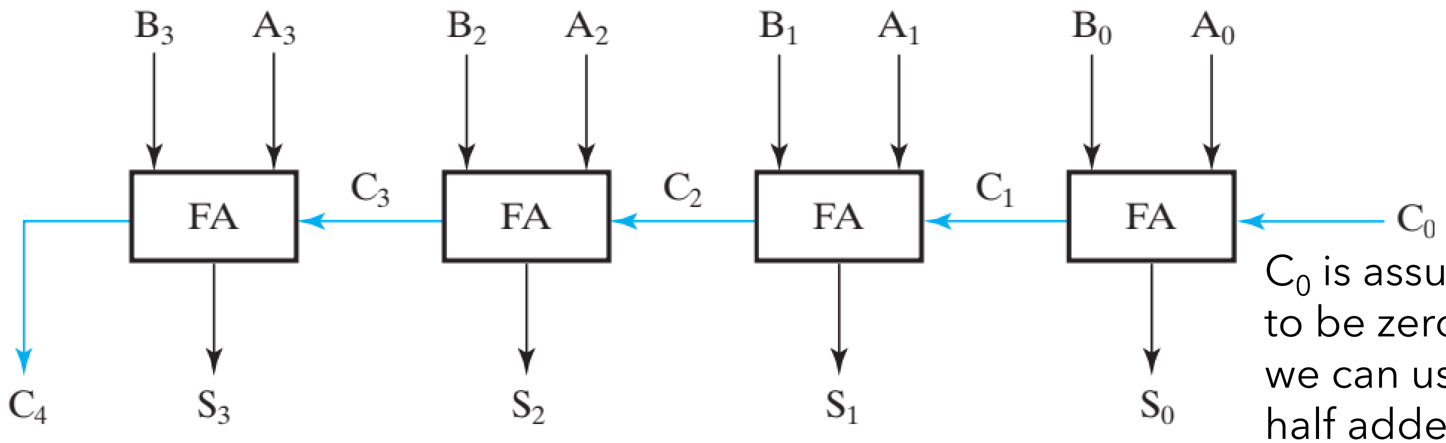
$$C = XY + XZ + YZ$$
$$= XY + Z(X\bar{Y} + \bar{X}Y)$$
$$= XY + Z(X \oplus Y)$$



Binary Ripple Carry Adder

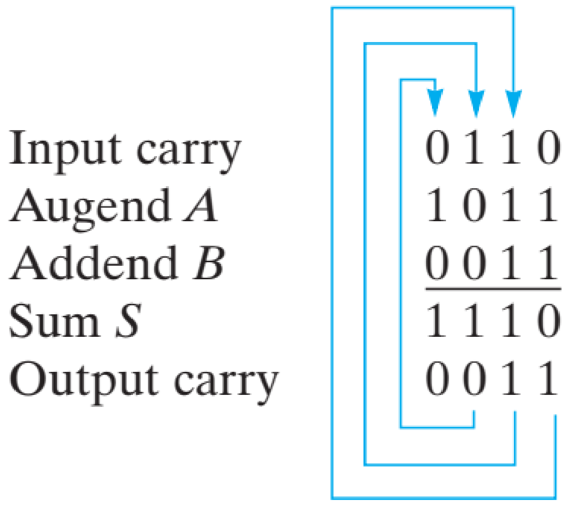
- Example: 4-bit ripple carry adder
  - A four-bit Ripple Carry Adder made from four 1-bit Full Adders

Description	Subscript 3 2 1 0	Name
Carry In	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
Carry out	0 0 1 1	C_{i+1}

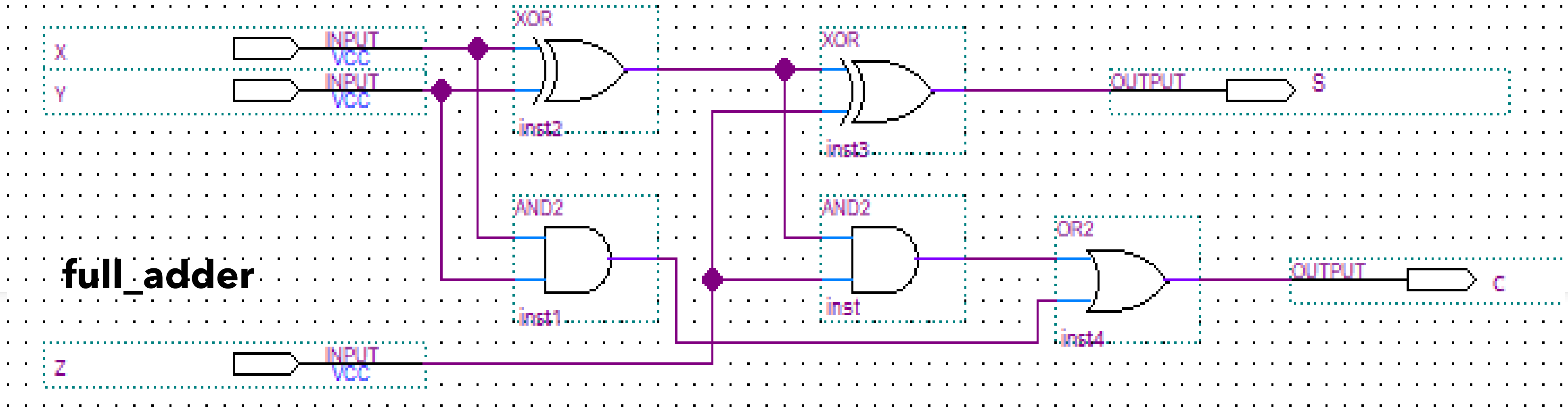


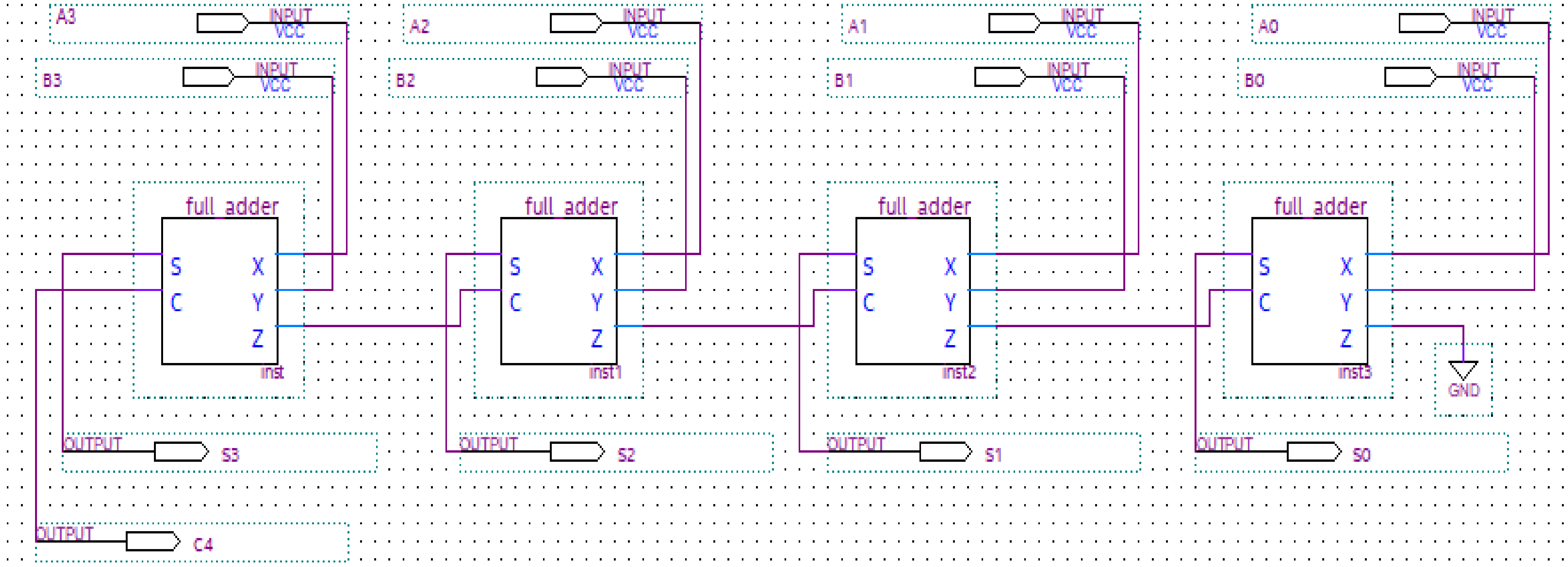
C_0 is assumed to be zero, or we can use a half adder for A_0 and B_0

FIGURE 3-43
4-Bit Ripple Carry Adder



Input carry	0 1 1 0
Augend A	1 0 1 1
Addend B	0 0 1 1
Sum S	1 1 1 0
Output carry	0 0 1 1


full_adder
X
Y
Z
INPUT
VCC
XOR
inst2
inst3
AND2
inst1
inst
OR2
inst4
OUTPUT
S
C


A3
B3
A2
B2
A1
B1
A0
B0
INPUT
VCC
full_adder
S
C
X
Y
Z
inst
inst1
inst2
inst3
GND
OUTPUT
S3
S2
S1
S0
C4

8. Binary Subtraction

- Unsigned numbers are all positive integers including zero.
- $M - N$
  1. Subtract the subtrahend N from the minuend M; result is n bits
  2. If no end borrow occurs, then $M \geq N$, and the result is a non-negative number and correct.
  3. If an end borrow occurs, then $N > M$, and the difference $M - N + 2^n$, is subtracted from 2^n, and a minus sign is appended to the result.
- Subtraction of a binary number from 2^n to obtain an n-digit result is called taking the 2s complement of the number. So in step 3, we are taking the 2s complement of the difference $M - N + 2^n$.

8. Binary Subtraction

```
•   0                            1            End Borrow
     1001                         0100
 -   0111                       - 0111
    00010                         1101
                            Step 3
                                 10000
                                - 1101
                             (-)  0011
```

Complements

- There are two types of complements for each base-*r* system: *the radix complement*, and the *diminished radix complement*.
- Diminished Radix Complement of *N*
  - (*r* - 1)'s complement for radix *r*
  - 1's complement for radix 2
  - Defined as $(r^n - 1) - N$
    - Obtained by complementing each individual bit (bitwise NOT).
- Radix Complement
  - *r*'s complement for radix *r*
  - 2's complement in binary
  - Defined as $r^n - N$
    - Is the 1's complement plus 1, a fact that can be used in designing hardware

Unsigned Subtraction with 2's Complement

- The subtraction of two *n*-digit unsigned numbers, $M - N$, in binary can be done as follows:
  1. Add the 2s complement of the subtrahend N to the minuend M. This performs $M + (2^n - N) = M - N + 2^n$.
  2. If $M \geq N$, the sum produces an end carry, 2^n. Discard the end carry, leaving result $M - N$.
  3. If $M < N$, the sum does not produce an end carry, since it is equal to $2^n - (N - M)$, the 2s complement of $N - M$. Perform a correction, taking the 2s complement of the sum and placing a minus sign in front to obtain the result $-(N - M)$.

Unsigned Subtraction with 2's Complement

- Find $01010100_2 - 01000011_2$

$$\begin{array}{rr} & 01010100 \\ - & \underline{01000011} \end{array} \xrightarrow{\textbf{2's complement}} \begin{array}{rr} \text{Carry} & \textbf{1} \\ & 01010100 \\ + & \underline{10111101} \\ & 00010001 \end{array}$$

- The carry of 1 indicates that no correction of the result is required.

Unsigned Subtraction with 2′s Complement

- Find $01000011_2 - 01010100_2$

$$\begin{array}{r} \\ 01000011 \\ -\ \underline{01010100} \\ \\ \end{array} \xrightarrow{\textbf{2's complement}} \begin{array}{r} \text{Carry}\quad \mathbf{0} \qquad\qquad\quad \\ 01000011 \\ +\ \underline{10101100} \\ 11101111 \end{array} \xrightarrow{\textbf{2's complement}} 00010001$$

- The carry of 0 indicates that a correction of the result is required.
- The result: -00010001

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 Integers

- **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 Integers

- $r = 2, n = 3$
- We need a sign bit with 2's complements to distinguish between positive and negative numbers.

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

Signed 2's Complement Arithmetic

- Addition:

  1. Add the numbers including the sign bits, discarding a carry out of the sign bits

  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:
  - Take the 2's complement of the subtrahend (including the sign bit) and **add** it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

Signed 2's Complement Arithmetic

- Example 1:

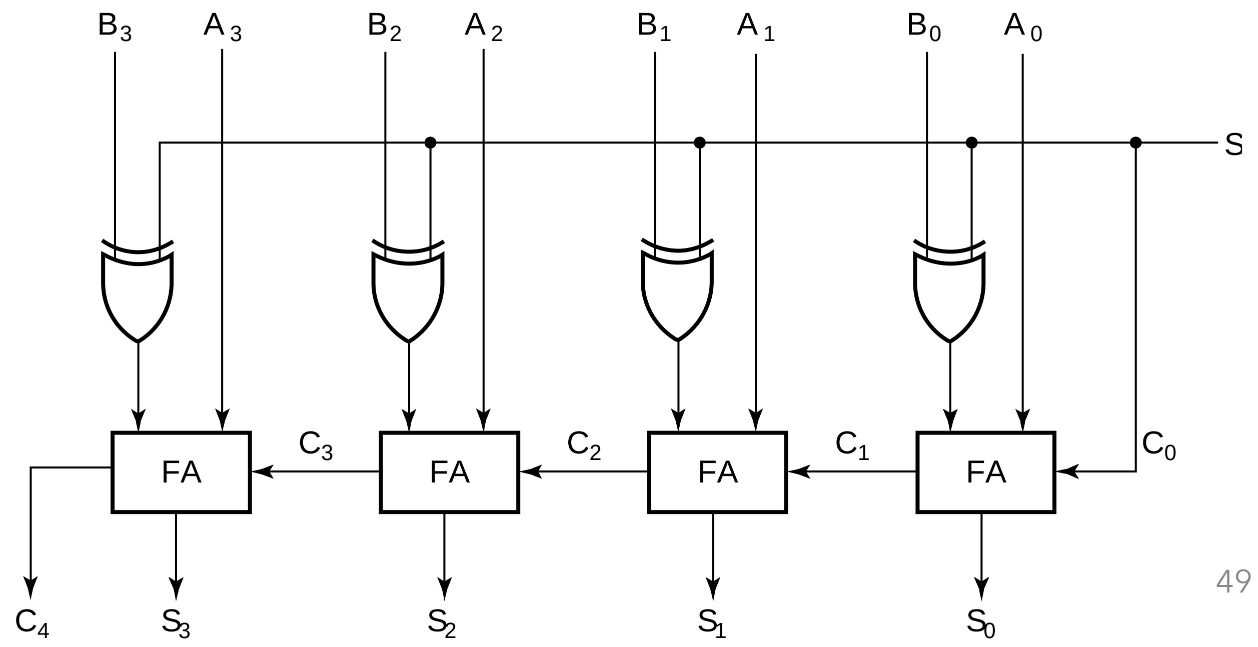
$$\begin{array}{r l} 1101 & (-3) \\ +\underline{0011} & (+3) \\ 0000 & \end{array}$$

- Example 2:

$$\begin{array}{r l} 1101 & (-3) \\ -\underline{0011} & (+3) \end{array} \xrightarrow{\textbf{2's comp.}} \begin{array}{r l} 1101 & (-3) \\ +\underline{1101} & (-3) \\ 1010 & (-6) \end{array}$$

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 C0.
  - For S = 0, add, B is passed through unchanged



Overflow Detection

- Overflow occurs if $n + 1$ bits are required to contain the result from an n-bit addition or subtraction
- Unsigned numbers
    - When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In unsigned subtraction, the magnitude of the result is always equal to or smaller than the larger of the original numbers, making overflow impossible.
- Signed Numbers
    - Overflow can occur for:
        - Addition of two operands with the same sign
        - Subtraction of operands with different signs

Overflow Detection

- Examples: 8-bit signed numbers

	Carries:	01
Overflow	+70	01000110
	+80	01010000
	+150	10010110

	Carries:	10
Overflow	−70	10111010
	−80	10110000
	−150	01101010

- 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, an overflow has occurred.

If V = 0 after a signed addition or subtraction, it indicates that no overflow has occurred and the result is correct. If V = 1, then an overflow has occurred.

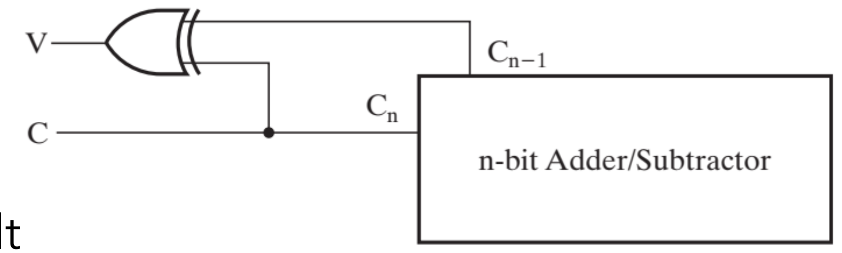
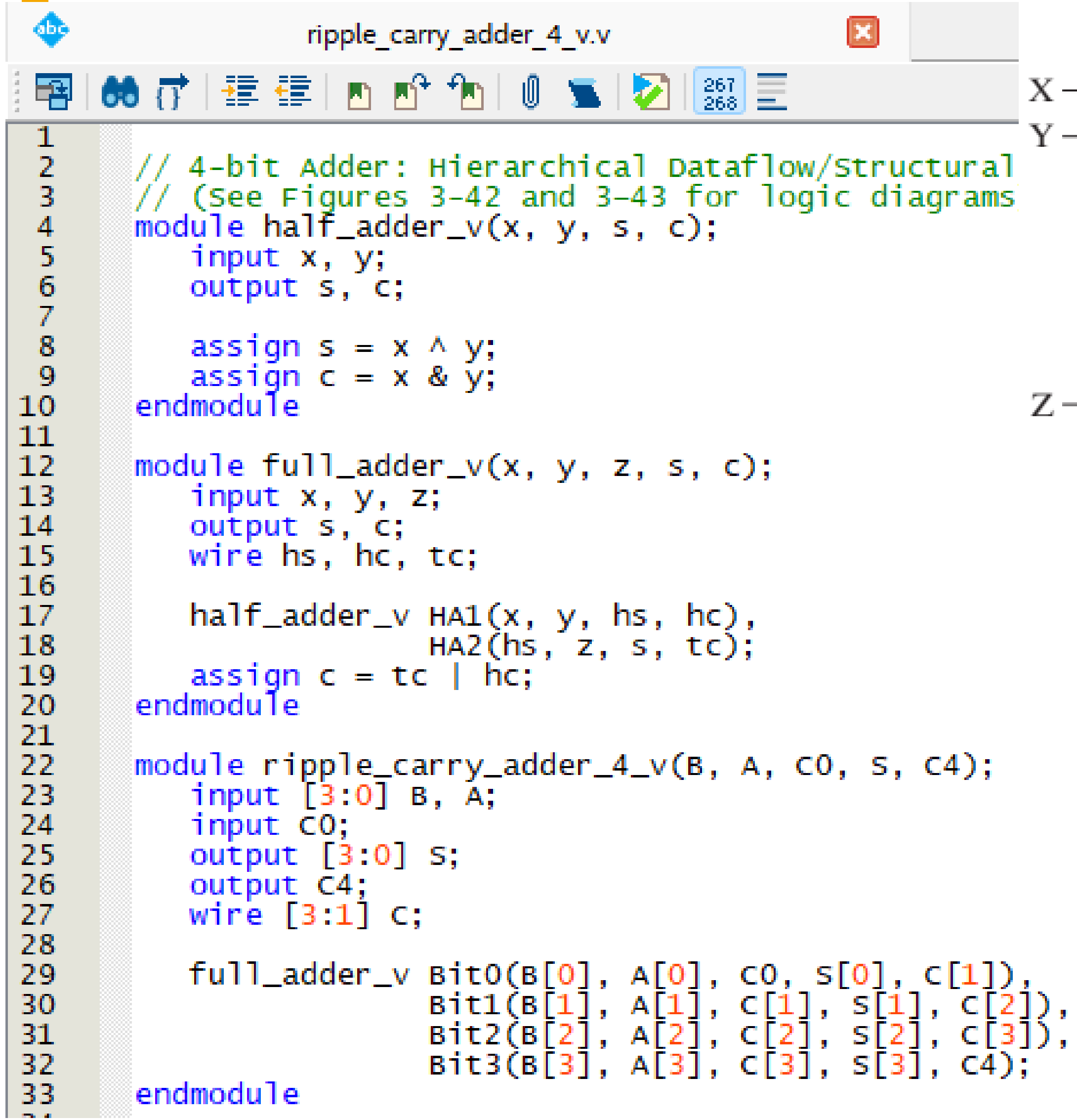


FIGURE 3-46
Overflow Detection Logic for Addition and Subtraction

Verilog HDL Models of Adders



ripple_carry_adder_4_v.v

```verilog

// 4-bit Adder: Hierarchical Dataflow/Structural
// (See Figures 3-42 and 3-43 for logic diagrams
module half_adder_v(x, y, s, c);
   input x, y;
   output s, c;

   assign s = x ^ y;
   assign c = x & y;
endmodule

module full_adder_v(x, y, z, s, c);
   input x, y, z;
   output s, c;
   wire hs, hc, tc;

   half_adder_v HA1(x, y, hs, hc),
                HA2(hs, z, s, tc);
   assign c = tc | hc;
endmodule

module ripple_carry_adder_4_v(B, A, C0, S, C4);
   input [3:0] B, A;
   input C0;
   output [3:0] S;
   output C4;
   wire [3:1] C;

   full_adder_v Bit0(B[0], A[0], C0, S[0], C[1]),
                Bit1(B[1], A[1], C[1], S[1], C[2]),
                Bit2(B[2], A[2], C[2], S[2], C[3]),
                Bit3(B[3], A[3], C[3], S[3], C4);
endmodule
```

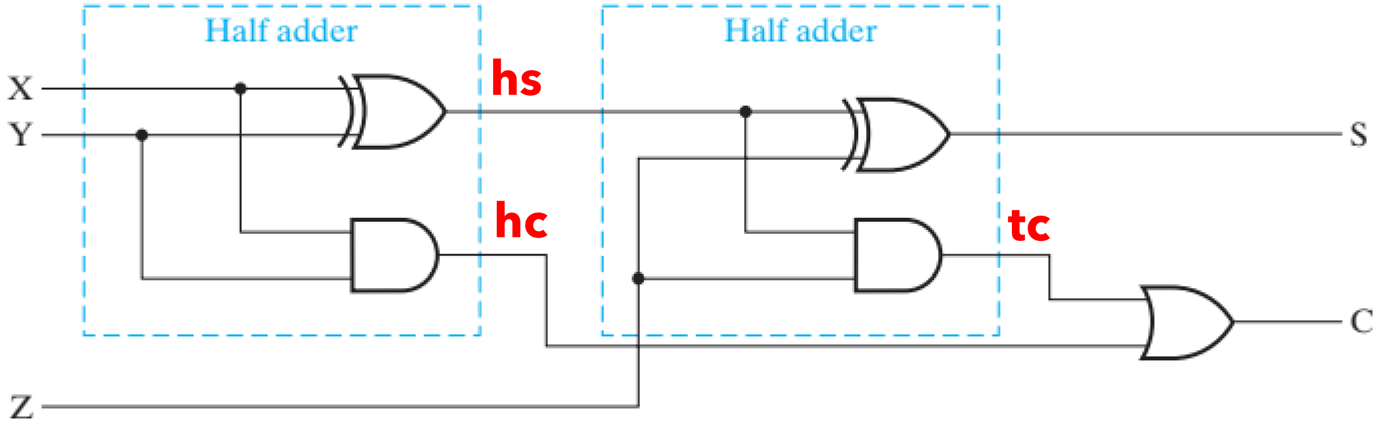


FIGURE 3-42
Logic Diagram of Full Adder

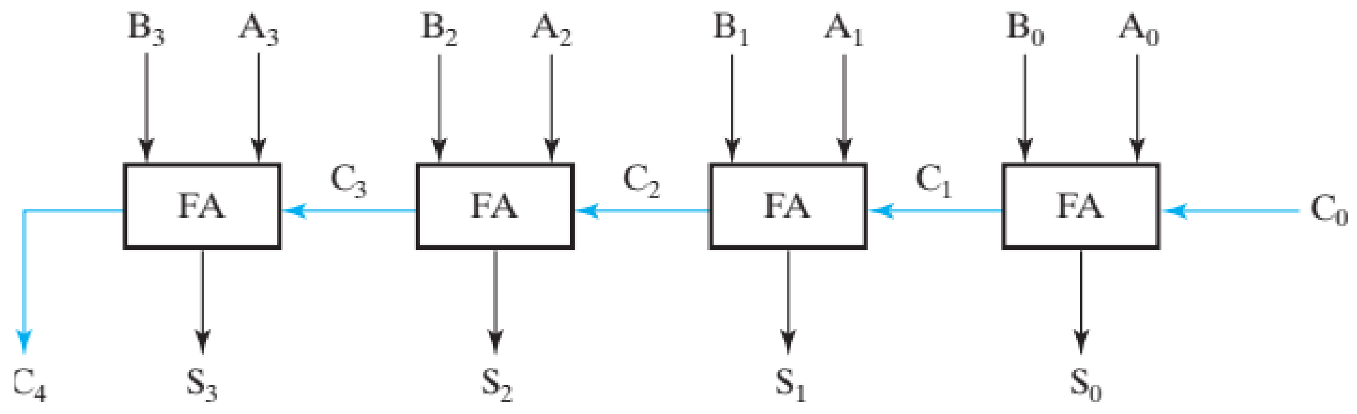
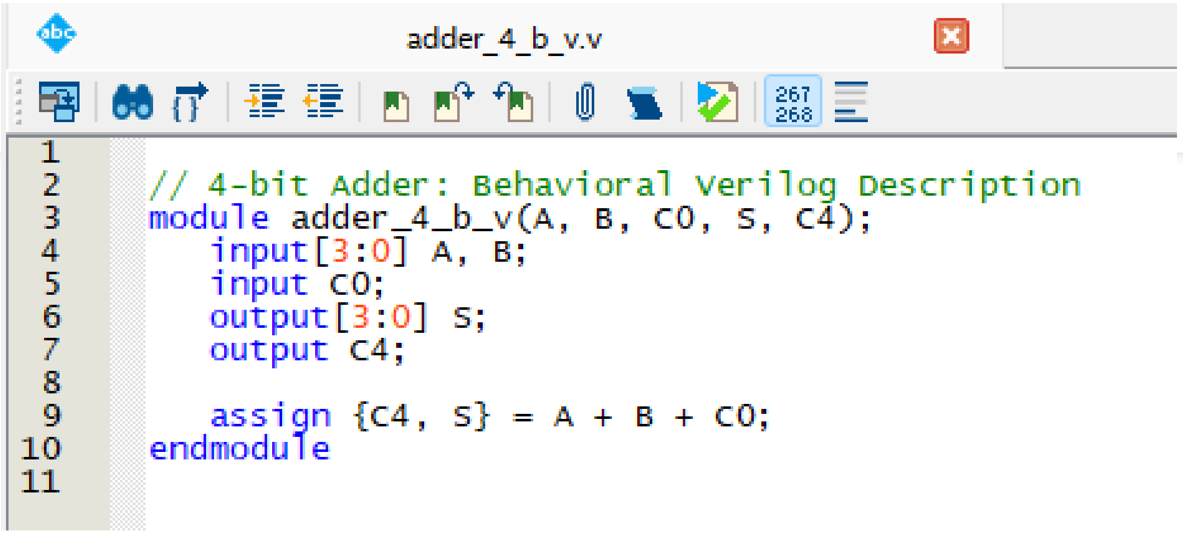


FIGURE 3-43
4-Bit Ripple Carry Adder

Behavioral Verilog for a 4-Bit Ripple Carry Adder


adder_4_b_v.v

```
// 4-bit Adder: Behavioral Verilog Description
module adder_4_b_v(A, B, C0, S, C4);
   input[3:0] A, B;
   input C0;
   output[3:0] S;
   output C4;

   assign {C4, S} = A + B + C0;
endmodule
```

- The + represents addition and the {} represents an operation called *concatenation*.
- The operation + performed on wire data types is **unsigned**.
- Concatenation combines two signals into a single signal having its number of bits equal to the sum of the number of bits in the original signals.
  - In the example, {C4,S} represents the signal vector

    C4 S[3] S[2] S[1] S[0]

  with 1 + 4 = 5 signals.
  - Note that C4, which appears on the left in the concatenation expression, appears on the left in the signal listing