Chapter 6

Combinational Logic Functions

6.1 Decoders

- Decoder: A digital circuit designed to detect the presence of a particular digital state.
- Can have one output or multiple outputs.
- Example: 2-Input NAND Gate detects the presence of ‘11’ on the inputs to generate a ‘0’ output.

Address Decoder

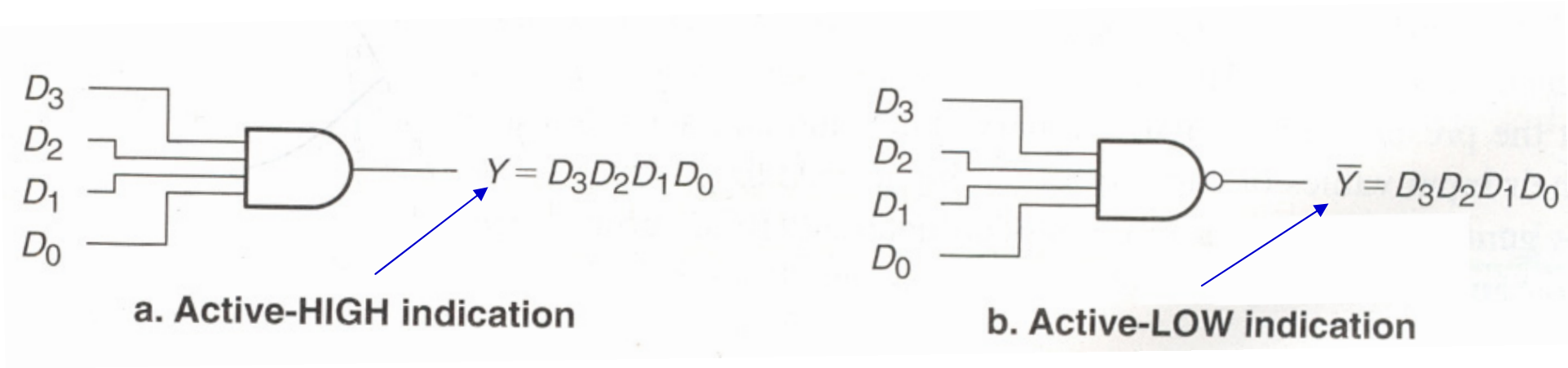
- A device that selects memory or peripheral devices for a block of addresses in a computer system

Single-Gate Decoders

- Uses single gates (AND/NAND) and some Inverters.
- Example: 4-Input AND detects '1111' on the inputs to generate a '1' output.
- Inputs are labeled D_3, D_2, D_1, and D_0, with D_3 the MSB (most significant bit) and D_0 the LSB (least significant bit).

Single-Gate Decoders

- Fig. a: generate a logic HIGH when its input is 1111
- Fig. b: generate a logic LOW when its input is 1111



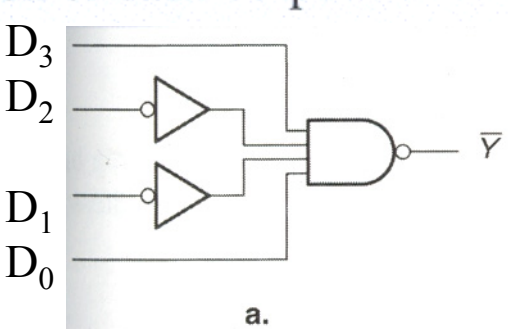
Single-Gate Examples

- If the inputs to a 4-Input NAND are given as $\overline{D_1}, \overline{D_2}, \overline{D_3}, D_4$, then the NAND detects the code 0001. The output is a 0 when the code 0001 is detected.

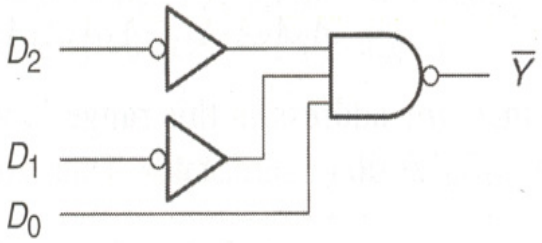
- This type of decoder is used in Address Decoding for a PC System Board.

Example 6.1

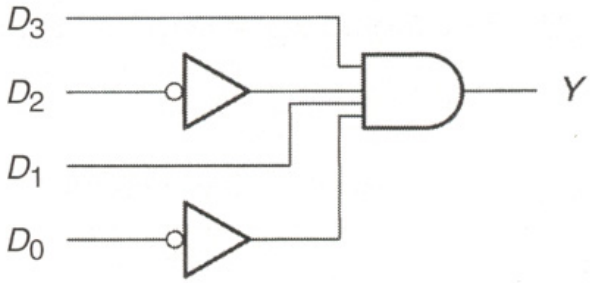
Figure 6.2 shows three single-gate decoders. For each one, state the output active level and the input code that activates the decoder. Also write the Boolean expression of each output.



a.



b.



c.

■ **Solution**

Each decoder is a NAND or AND gate. For each of these gates, the output is *active* when *all inputs are HIGH*. Because of the inverters, each circuit has a different code that fulfils this requirement.

Figure 6.2a: Output: Active LOW
Input code: $D_3D_2D_1D_0 = 1001$

$$\overline{Y} = D_3\overline{D}_2\overline{D}_1D_0$$

Figure 6.2b: Output: Active LOW:
Input code: $D_2D_1D_0 = 001$

$$\overline{Y} = \overline{D}_2\overline{D}_1D_0$$

Figure 6.2c: Output: Active HIGH
Input code: $D_3D_2D_1D_0 = 1010$

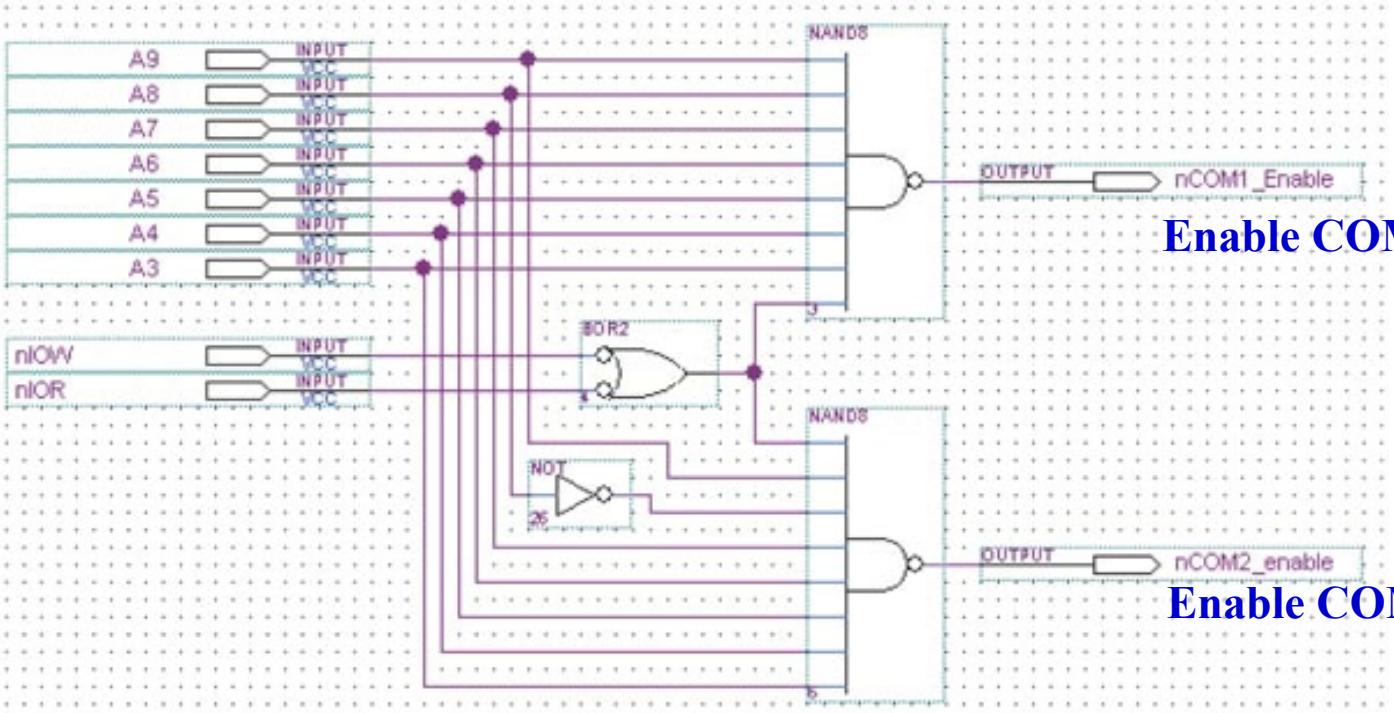
$$Y = D_3\overline{D}_2D_1\overline{D}_0$$

Example 6.2 -1

A PC has two serial port cards called COM1 and COM2. Each card is activated when either one of two control inputs called $\overline{IOR}$ *(Input/Output Read)* and $\overline{IOW}$ *(Input/Output Write)* are active and a unique 10-bit address is present. $\overline{IOR}$ and $\overline{IOW}$ are active-LOW. The address is specified by bits $A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$, which can be represented by three hexadecimal digits. The decoder outputs, $\overline{COM1\_Enable}$ and $\overline{COM2\_Enable}$ are both active-LOW.

The card for COM1 activates when ($\overline{IOR}$ OR $\overline{IOW}$ is LOW) AND the address is between 3F8H and 3FFH.

The card for COM2 activates when ($\overline{IOR}$ OR $\overline{IOW}$ is LOW) AND the address is between 2F8H and 2FFH.



Enable COM1

Enable COM2

Example 6.2 -2

Solution

The lowest address that activates COM1 is

$$A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 3F8H = 11\ 1111\ 1000$$

The highest COM1 address is

$$A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 3FFH = 11\ 1111\ 1111$$

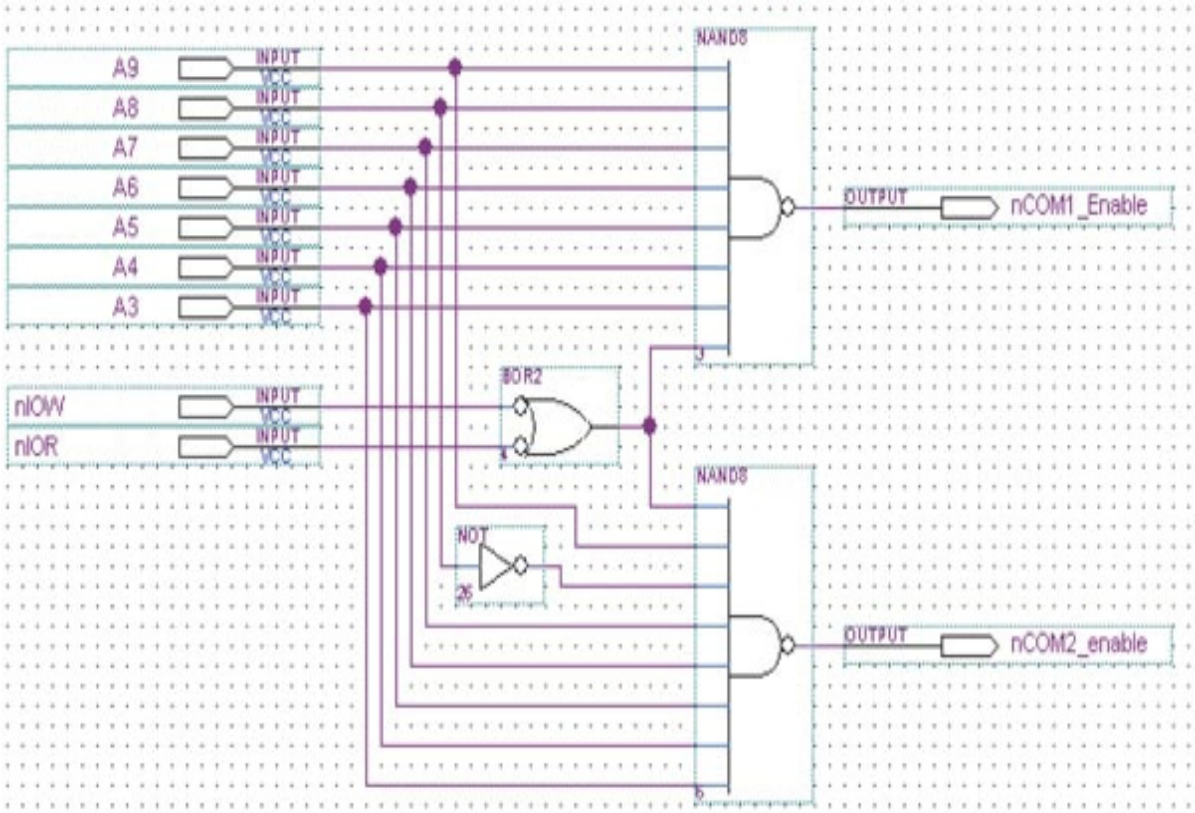
Since *any* address in this range is valid, we can represent the last three $A_2A_1A_0$, as don't care states. Thus, for COM1, we should decode the addres

$$A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 11\ 1111\ 1XXX$$

Similarly, for COM2:

Low address: $A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 2F8H = 10\ 1111\ 1000$
High address: $A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 2FFH = 10\ 1111\ 1111$
Decode: $A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 = 10\ 1111\ 1XXX$

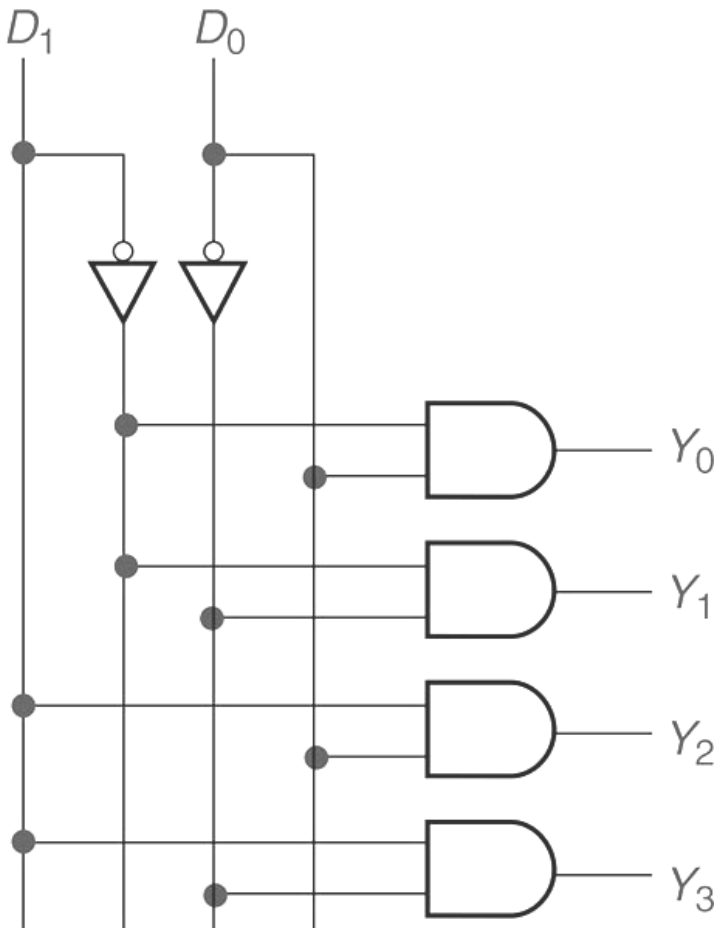
Figure 6.3 shows the **bdf** representation of the decoder circuit, including in for the control signals $\overline{IOR}$ and $\overline{IOW}$. The active-LOW inputs are indicated prefix, labeled *n* (e.g., *nIOR*), since we cannot draw the bar over the active-L input names.



Multiple-Output Decoders

- Decoder circuits often are constructed with multiple outputs. In effect, such a device is a collection of decoding gates controlled by the same inputs.
- A decoder circuit with n inputs can activate $m = 2^n$ load circuits.
- Called a *n*-line-to-*m*-line decoder, such as a 2-to-4 or a 3-to-8 decoder.
- Usually has an active low enable $\overline{G}$ that enables the decoder outputs.

2-Line-to-4-Line Decoder



- One and only one output is HIGH for any input combination

TABLE 6.1 Truth Table of a 2-to-4 Decoder with Enable

D_1	D_0	Y_0	Y_1	Y_2	Y_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

2-Line-to-4-Line Decoder with Active-LOW Enable

- Addition input $\overline{G}$

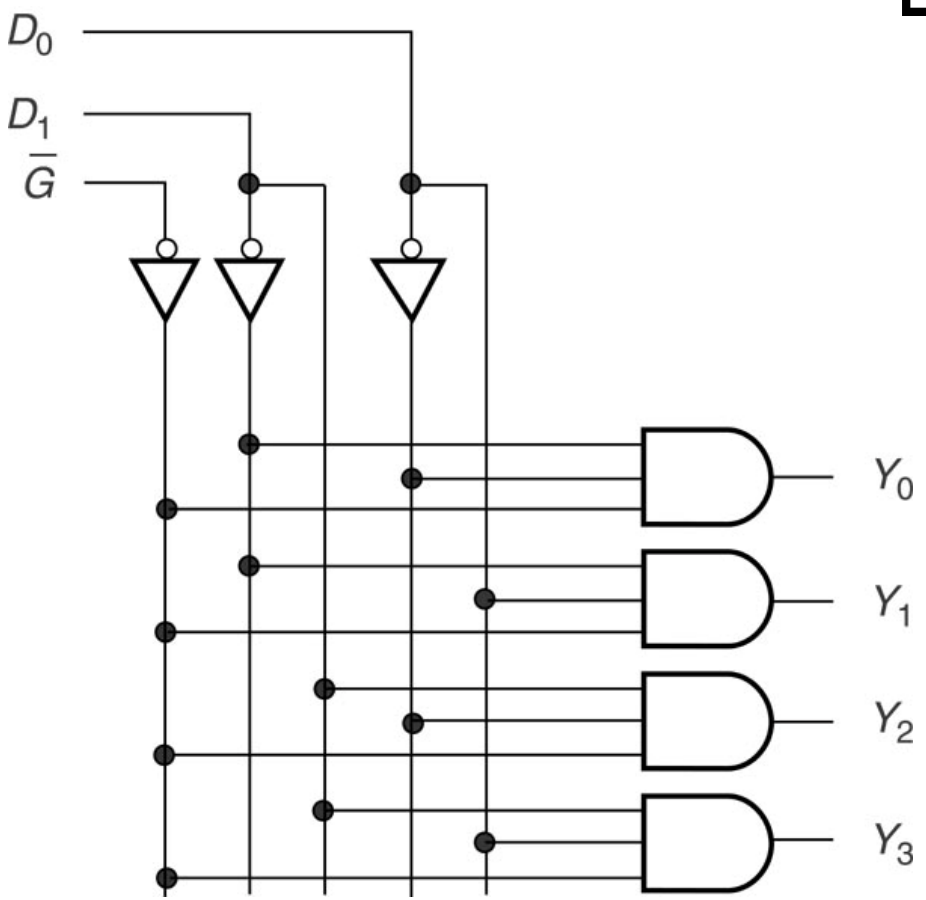


TABLE 6.2 Truth Table of a 2-Line-to-4-Line Decoder with Enable

$\overline{G}$	D_1	D_0	Y_0	Y_1	Y_2	Y_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

3-Line-to-8-Line Decoder with Enable

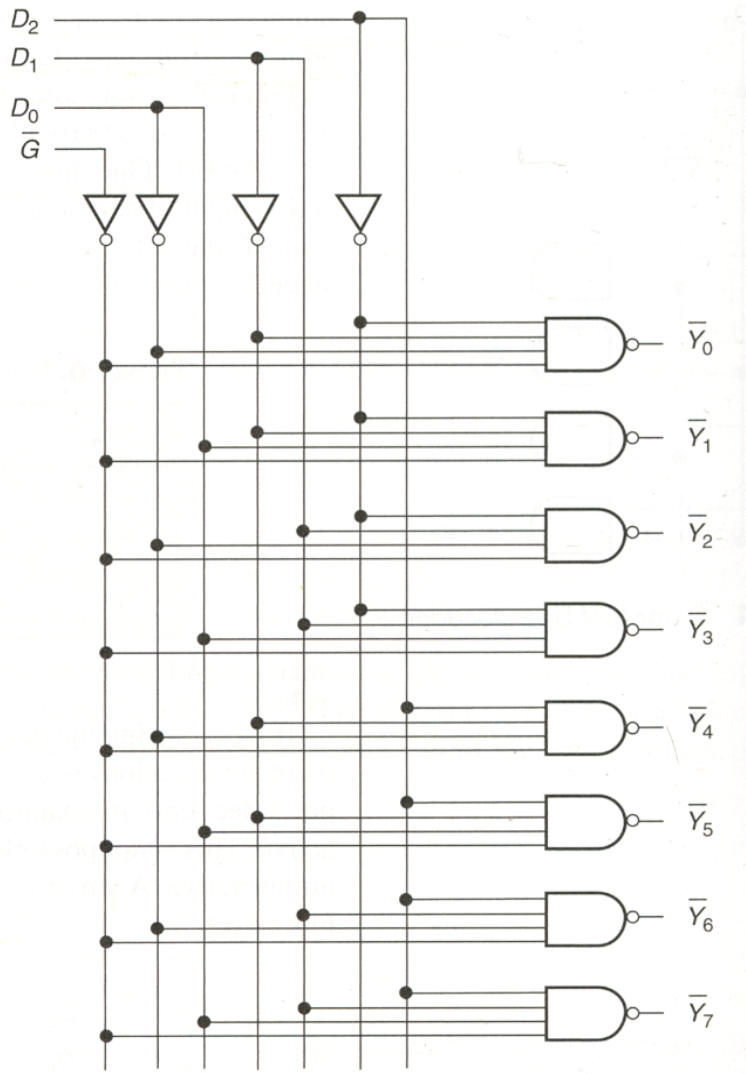


Fig. 6.6

Figure 6.6 shows the circuit for a 3-line-to-8-line decoder, again with an a LOW enable, $\overline{G}$. In this case, the decoder outputs are active LOW. One and onl output is active for any given combination of $D_2D_1D_0$. Table 6.3 shows the table for this decoder. Again if the enable line is HIGH, no output is active.

TABLE 6.3 Truth Table of a 3-to-8 Decoder with Enable

$\overline{G}$	D_2	D_1	D_0	$\overline{Y}_0$	$\overline{Y}_1$	$\overline{Y}_2$	$\overline{Y}_3$	$\overline{Y}_4$	$\overline{Y}_5$	$\overline{Y}_6$	$\overline{Y}_7$
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	0
1	X	X	X	1	1	1	1	1	1	1	1

3-Line-to-8-Line 74138 Decoder

A 3-line-to-8-line decoder that is commercially available in a single chip is the 74138 decoder, shown in Figure 6.7. Depending on the logic family, this device will be designated 74LS138, 74HC138, or some other such number.

This decoder has active-LOW outputs and three enable inputs, one active-HIGH and two active-LOW, all of which must be active to enable any of the outputs. The truth table for this decoder is shown in Table 6.4.

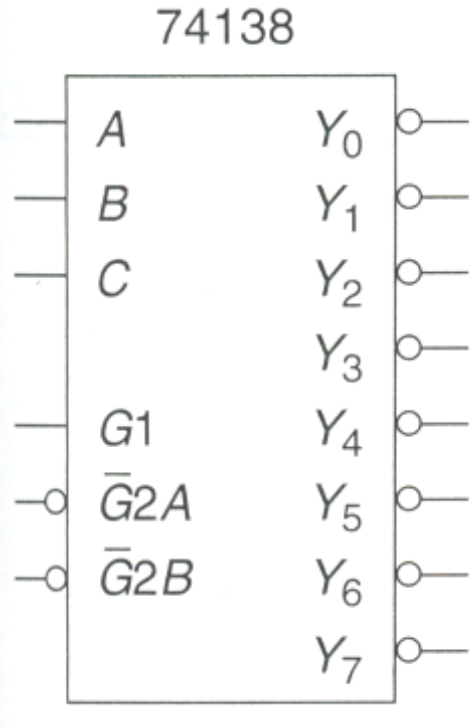


TABLE 6.4 Truth Table for a 74138 Decoder

$G1$	$\bar{G}2A$	$\bar{G}2B$	C	B	A	$\bar{Y}_0$	$\bar{Y}_1$	$\bar{Y}_2$	$\bar{Y}_3$	$\bar{Y}_4$	$\bar{Y}_5$	$\bar{Y}_6$	$\bar{Y}_7$
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	1	X	X	X	X	1	1	1	1	1	1	1	1
1	X	1	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

Notice that the Y outputs are active-LOW and that they are selected by a combination of inputs CBA, where C is the most significant bit.

Example 6.3 -1
74138 as an Address Decoder

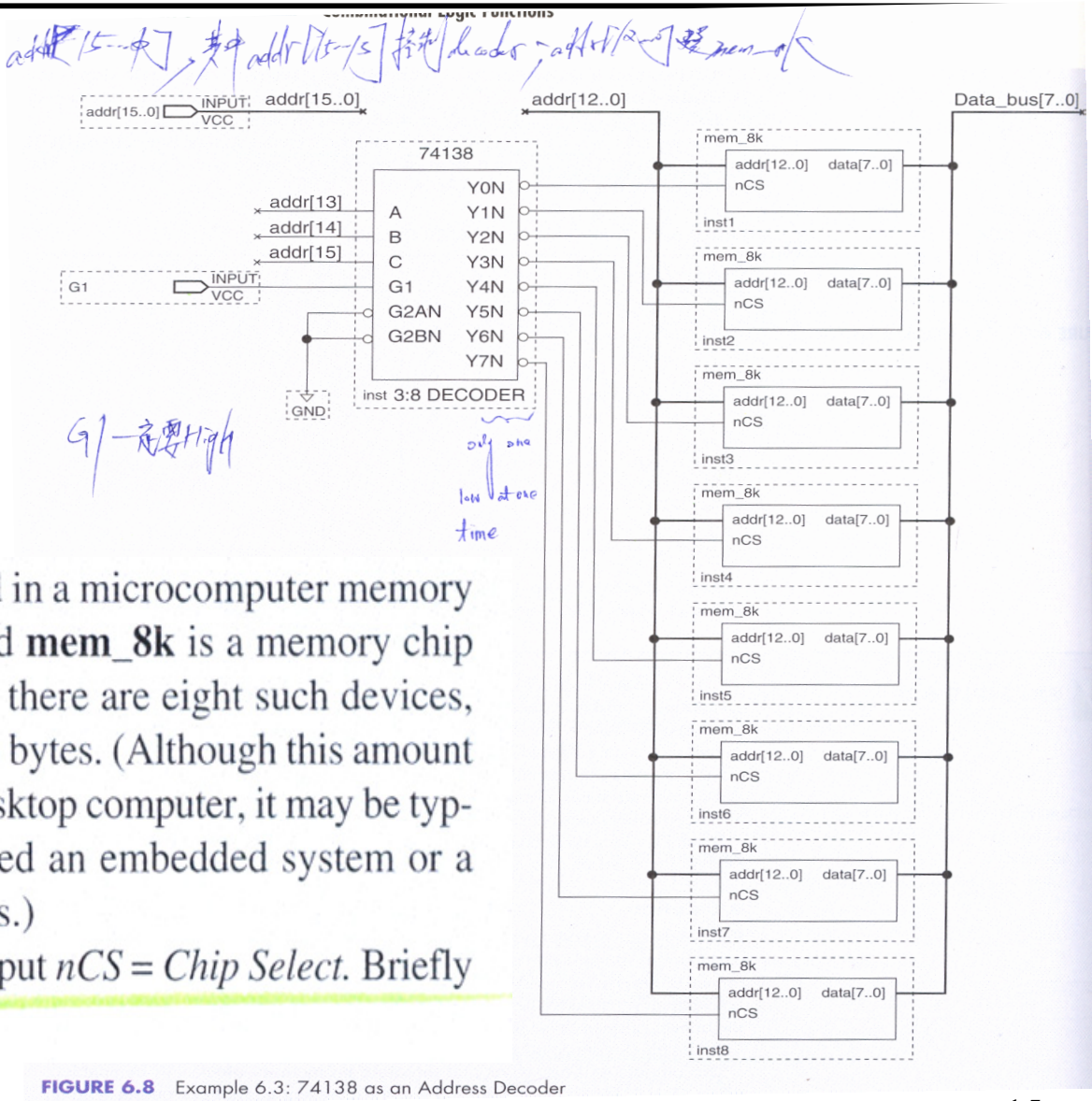
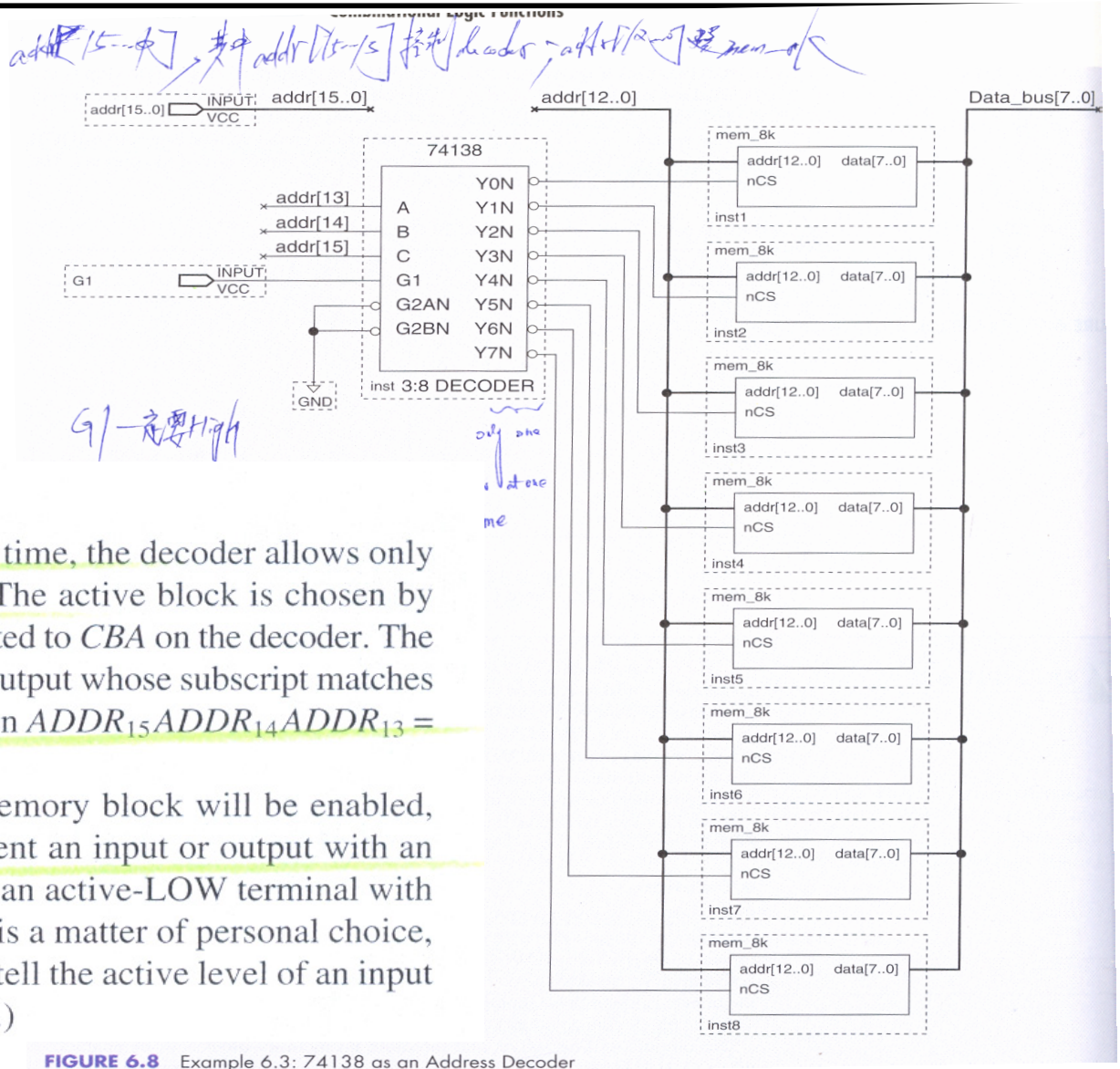


Figure 6.8 shows how a 74138 decoder can be used in a microcomputer memory system as an **address decoder.** Each block labeled **mem_8k** is a memory chip capable of holding 8192 (8K) bytes of data. Since there are eight such devices, the whole system can hold $8 \times 8192 = 65{,}536$ (64K) bytes. (Although this amount of memory may seem small by the standards of a desktop computer, it may be typical of a small stand-alone computer system (called an embedded system or a microcontroller) that is used in control applications.)

Each 8K block is enabled by a LOW at its *nCS* input *nCS* = *Chip Select.* Briefly explain the function of the decoder in the system.

FIGURE 6.8 Example 6.3: 74138 as an Address Decoder

Example 6.3 -2
74138 as an Address Decoder



Solution

Since only one decoder output is LOW at any one time, the decoder allows only one memory block to be active at any one time. The active block is chosen by inputs $ADDR_{15}ADDR_{14}ADDR_{13}$, which are connected to *CBA* on the decoder. The active memory block is the one connected to the *y* output whose subscript matches the binary value of these inputs. For example, when $ADDR_{15}ADDR_{14}ADDR_{13} = 110$, the block connected to Y_6 is active.

No outputs will be active, and therefore no memory block will be enabled, when $G_1 = 0$. (Note that Quartus II cannot represent an input or output with an inversion bar. Some conventions would represent an active-LOW terminal with an "*n*" prefix, indicating "NOT" (e.g., *nCS*). This is a matter of personal choice, but without such an indication it is not possible to tell the active level of an input or output from the Quartus II Block Diagram File.)

FIGURE 6.8 Example 6.3: 74138 as an Address Decoder

Simulation

KEY TERMS

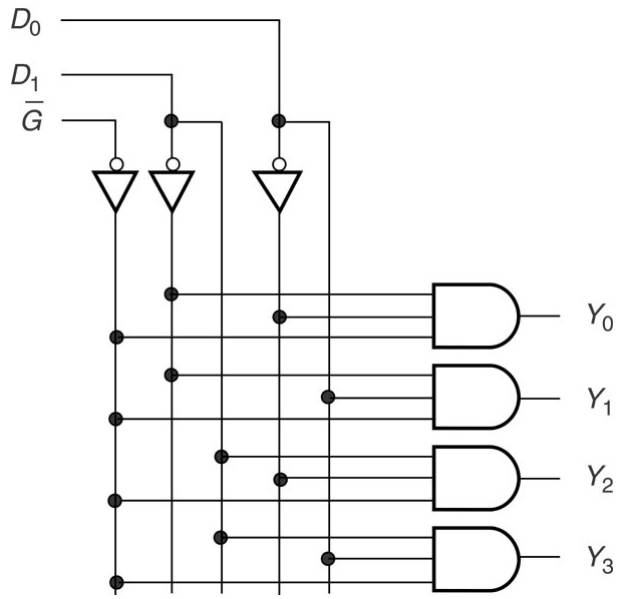
Simulation The verification, using timing diagrams, of the logic of a digital design before programming it into a PLD.

Timing Diagram A diagram showing how two or more digital waveforms in a system relate to each other over time.

Stimulus Waveforms A set of user-defined input waveforms in a simulator file designed to imitate input conditions of a digital circuit.

Response Waveforms A set of output waveforms generated by a simulator for a particular digital design in response to a set of stimulus waveforms.

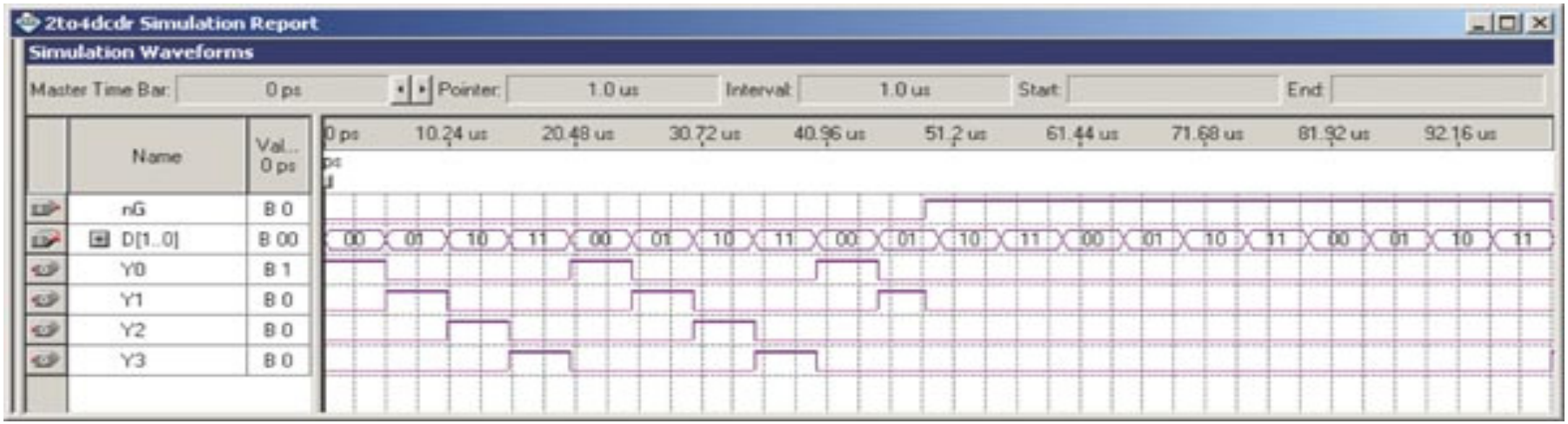
Simulation Waveform of a 2-Line-to-4-Line Decoder with Active-LOW Enable



- Addition input $\overline{G}$

TABLE 6.2 Truth Table of a 2-Line-to-4-Line Decoder with Enable

$\overline{G}$	D_1	D_0	Y_0	Y_1	Y_2	Y_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

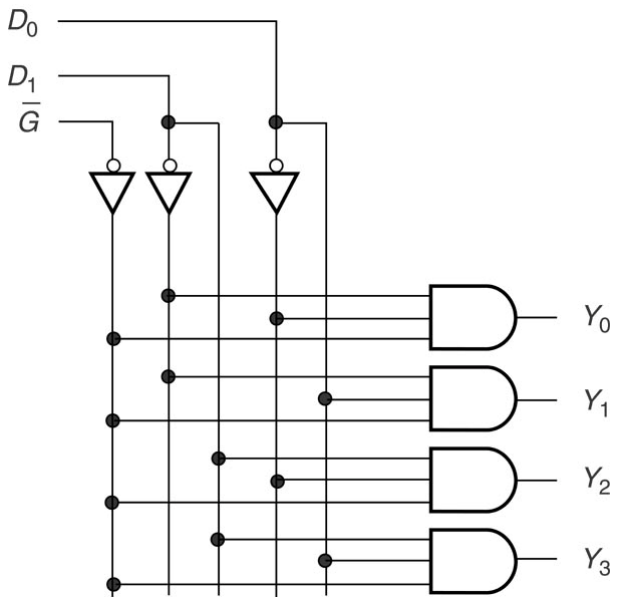


Simulation Criteria for n-Line-to-m-Line Decoders

- Each output must respond to its appropriate binary input by going HIGH wh selected.
- Only one output must be HIGH at any time.
- If an ascending 2-bit binary count is applied to inputs $D1$ and $D0$, the outpu will go HIGH in the sequence $Y0$, $Y1$, $Y2$, and $Y3$, then repeat.
- If $\overline{G} = 0$, the decoder outputs can activate as described in the previou criteria.
- If $\overline{G} = 1$, no output will activate for any value of $D1$ and $D0$.

Example 6.4 -1

The 2-line-to-4-line decoder in Figure 6.5 is tested with input waveforms meeting the simulation criteria previously discussed. The simulation result, shown in Figure 6.10, indicates an error condition in the decoder. What is likely to be the circuit fault?



Example 6.4 -2

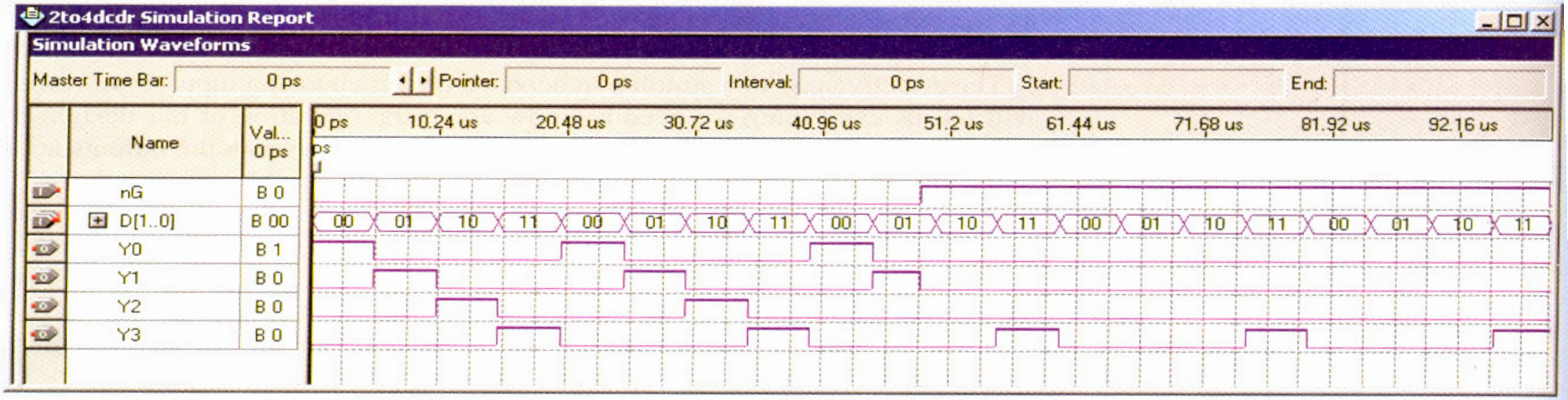


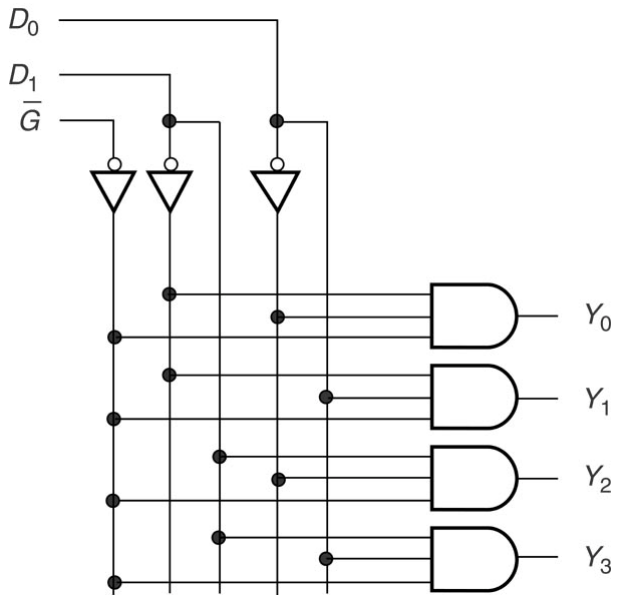
FIGURE 6.10 Example 6.4: Simulation of a 2-Line-to-4-Line Decoder with Error in Enable/Disable Function

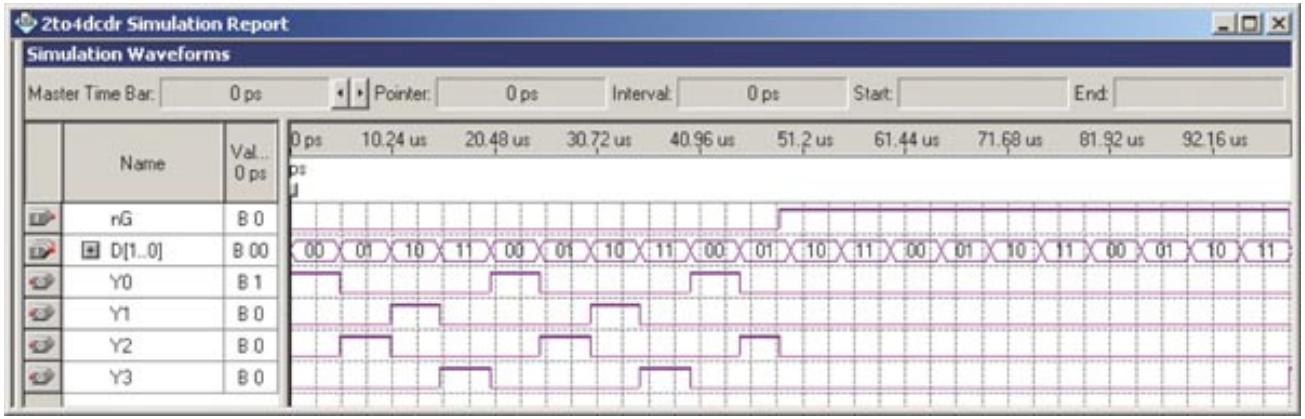
■ Solution

The simulation waveforms show the outputs behaving correctly when the $\overline{G}$ input is LOW, that is, when the decoder is enabled. When the decoder should be disabled (when $\overline{G}$ is HIGH), the $Y3$ output activates whenever $D1 = 1$ and $D0 = 1$. This is the correct decoding for this output, but it should not be decoding at all when the decoder is disabled. This could result from an improper connection from $\overline{G}$ to the gate that decodes $Y3$. The connection from $\overline{G}$ to $Y3$ is open-circuited or stuck HIGH.

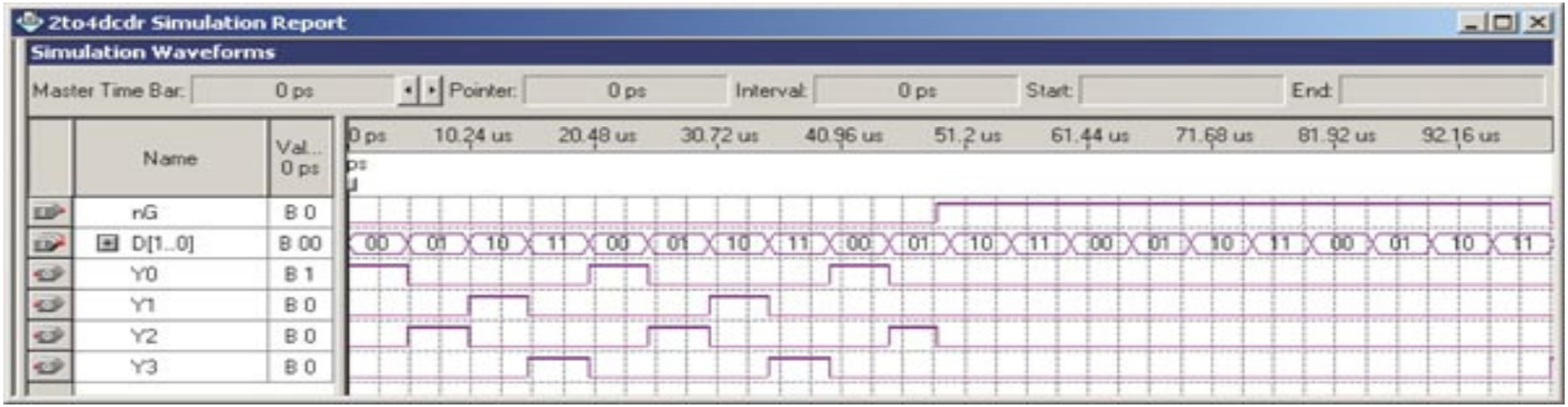
Example 6.5 -1

The 2-line-to-4-line decoder of Figure 6.5 is tested with input waveforms meeting appropriate simulation criteria. The simulation result, shown in Figure 6.11, indicates an error condition in the decoder. What is likely to be the circuit fault?





Example 6.5 -2



■ Solution

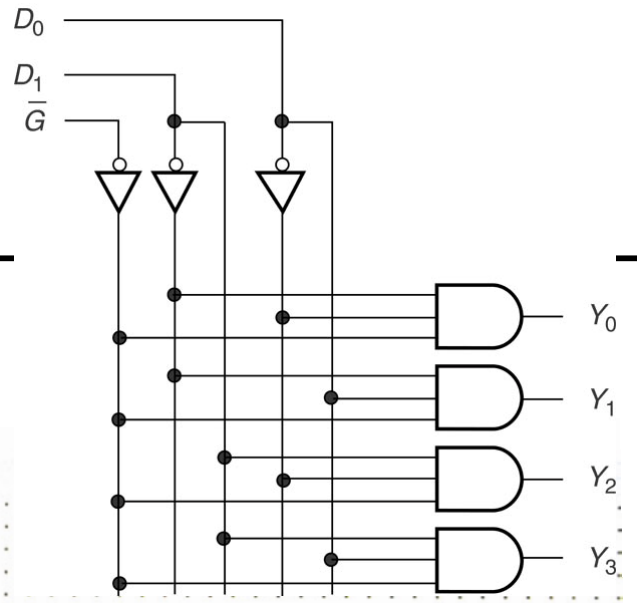
The outputs are enabled and disabled when they are supposed to be, but, when enabled, the outputs activate in the wrong order. We can sort this out by comparing the actual outputs to the expected outputs, as shown in Table 6.5.

TABLE 6.5 Actual and Expected Decoder Outputs

Output	Should activate when D =	Actually activates when D =
$Y0$	00	00
$Y1$	01	10
$Y2$	10	01
$Y3$	11	11

Notice that the values in the actual and expected columns are mirror images of one another. This implies that when we entered the design in the Quartus II Block Editor, we mixed up the order of $D1$ and $D0$, as shown in Figure 6.12. Reconnecting (or renaming) the inputs will fix the problem.

Example 6.5 -3



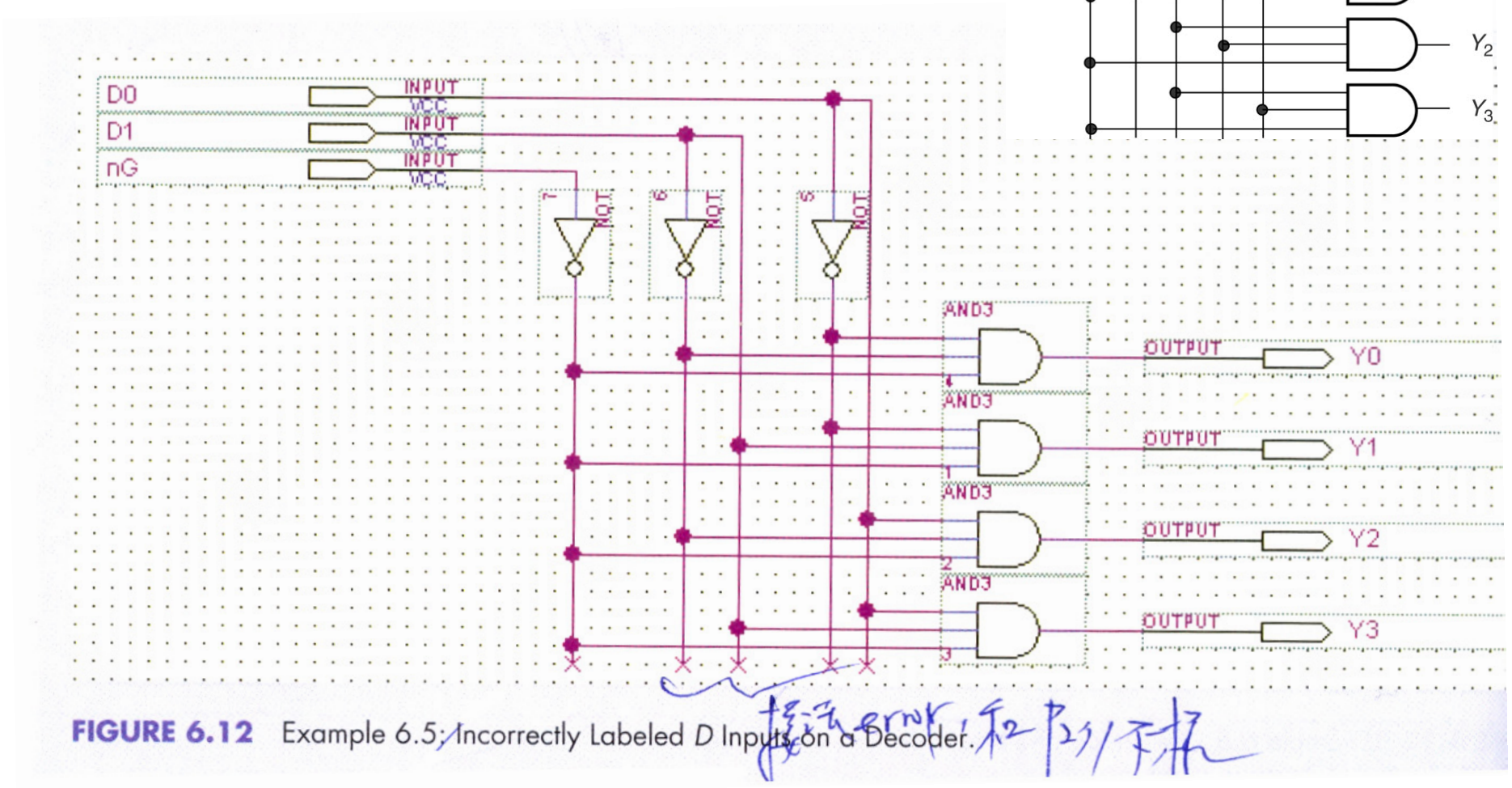
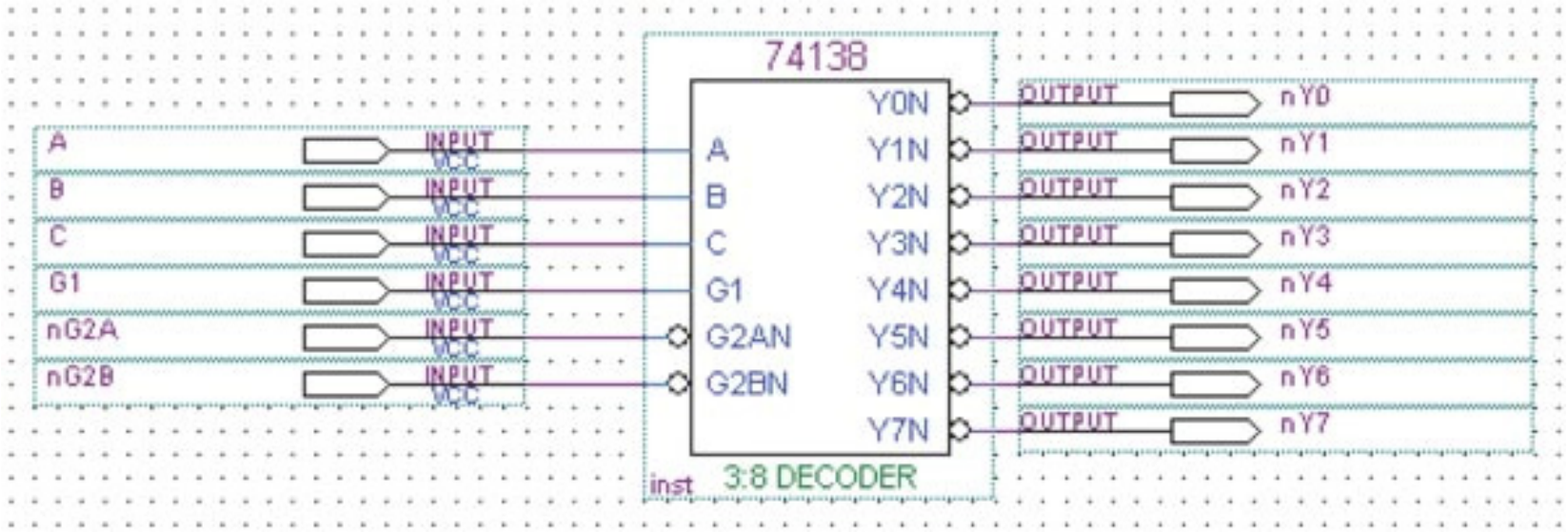


FIGURE 6.12 Example 6.5; Incorrectly Labeled *D* Inputs on a Decoder.

Example 6.6 -1

A 74138 3-line-to-8-line decoder is connected in a Quartus II Block Diagram as shown in Figure 6.13. Make a list of simulation criteria that will fully tes decoder and create a Quartus II simulation to verify the operation of the dev



Example 6.6 -2

SIMULATION CRITERIA

■ Solution

- Each decoder output should respond to its appropriate binary input value going LOW.
- Only one output should be LOW at any given time.
- An increasing 3-bit binary count on inputs *CBA* should make the outputs go LO one at a time, starting with *Y*0, followed by *Y*1, *Y*2, and so on until *Y*7, and th repeat.
- The enable inputs must all be active for the outputs to activate: $G1 = 1$, $\bar{G}$ $= 0$, and $\bar{G}2B = 0$
- If any enable input is inactive, no decoder output should be active, regardle of the values of the input *CBA*.

Example 6.6 -3

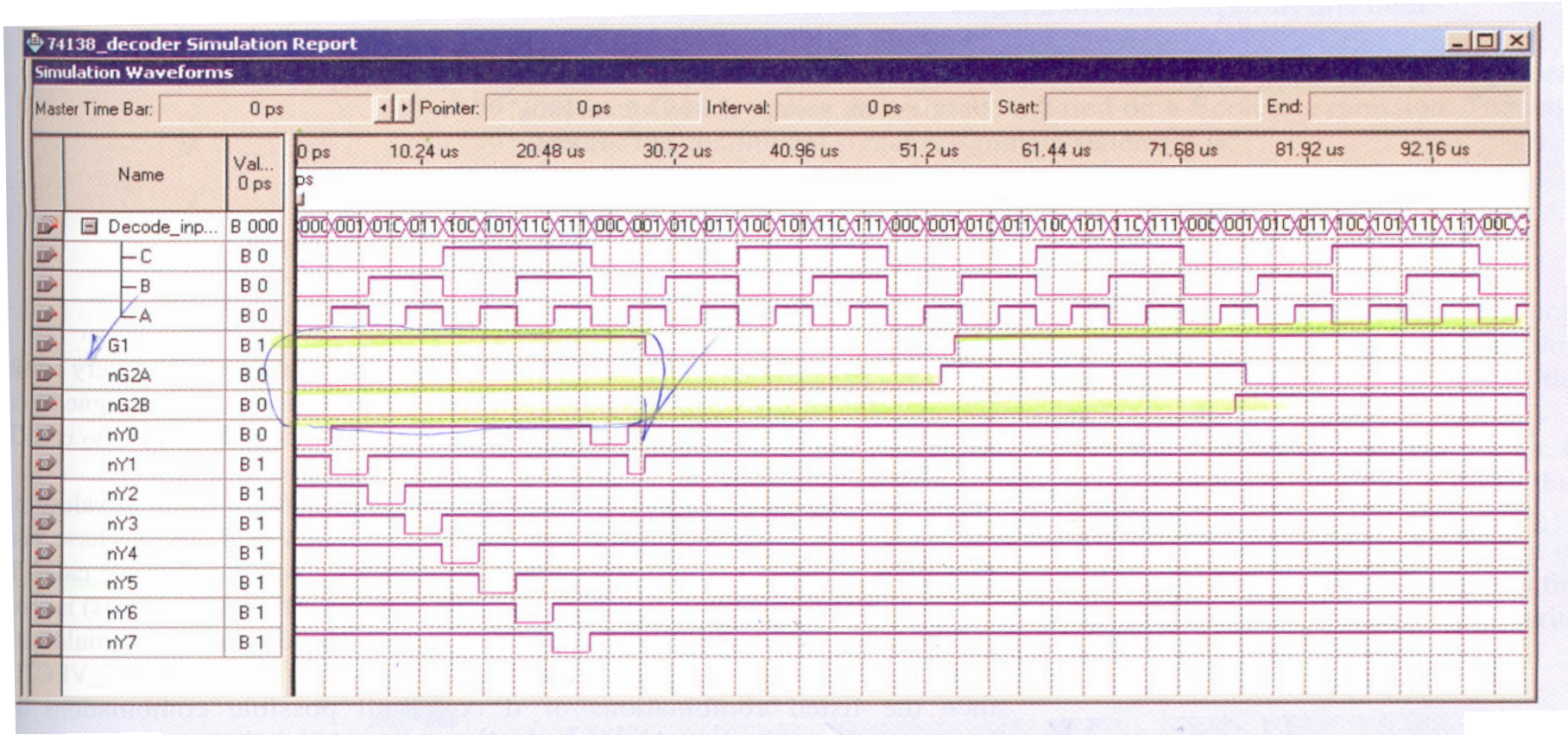


Figure 6.14 shows a simulation that meets these criteria. An increasing binary count on *CBA* (also shown grouped as *Decode inputs*), activates the outputs in sequence, as long as all three enables are active. Any one of the enables inactive prevents any output from activating. Notice that each of the enable inputs are inactive for the entire binary count range.

VHDL Binary Decoder

- Use **selected signal assignment statements** constructs or **conditional signal assignment statements** constructs.

- WITH SELECT

- WHEN ELSE

KEY TERMS

Selected Signal Assignment Statement A concurrent signal assignment in VHDL in which a value is assigned to a signal, depending on the alternative values of another signal or variable.

Conditional Signal Assignment Statement A concurrent VHDL construct that assigns a value to a signal, depending on a sequence of conditions being true or false.

2-to-4 Decoder VHDL -1

- Using a **select signal assignment statement:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY decode3 IS
   PORT(
       d : IN  STD_LOGIC_VECTOR (1 downto 0);
       y : OUT STD_LOGIC_VECTOR (3 downto 0));
END decode3;

ARCHITECTURE decoder OF decode3 IS
BEGIN
   WITH d SELECT
      y <=    "0001" WHEN "00",
              "0010" WHEN "01",
              "0100" WHEN "10",
              "1000" WHEN "11",
              "0000" WHEN others;
END decoder;
```

d(1)
d(0)
y(0)
y(3)
default case

2-to-4 Decoder VHDL -2

The selected signal assignment statement evaluates input **d.** For every possible combination of the 2-bit input vector, **d,** a particular value is assigned to the 4-bit vector, **y.** (For example, for the case $d_1d_0 = 10$ $(= 2_{10})$, the output y_2 is HIGH: $y_3y_2y_1y_0 = 0100$.)

The default case (**"WHEN others"**) is required because of the multivalued logic type STD_LOGIC_VECTOR. Since a STD_LOGIC_VECTOR can have values other than '0' and '1', the values listed for **d** don't cover all possible cases. The default output (which will never occur if we only use '0' and '1' inputs) is chosen such that no output is active in the default case. The default case would not be required if we chose to use BIT_VECTOR, rather than STD_LOGIC_VECTOR, since the listed combinations of **d** cover all possible combinations of a BIT_VECTOR.

Selected Signal Entity

- In the previous slide, the Entity used a STD LOGIC Array for Inputs and Outputs.
- The Y : OUT STD_LOGIC_VECTOR(3 downto 0) is equal to Y_3, Y_2, Y_1, Y_0.
- The STD_LOGIC Data Type is similar to BIT but has added state values such as Z, X, H, and L instead of just 0 and 1.

Selected Signal Assignments

- Uses a VHDL Architecture construct called WITH SELECT.

- Format is:
  - WITH (signal input(s)) SELECT.
  - Signal input states are used to define the output state changes.

2-to-4 Decoder VHDL Architecture

```
ARCHITECTURE  decoder  OF decode3
   IS
BEGIN
 WITH  d  SELECT
        y <= "0001" WHEN  "00",
             "0010" WHEN  "01",
             "0100" WHEN  "10",
             "1000" WHEN  "11",
             "0000" WHEN  others;
END  decoder;
```

Decoder Architecture

- The decoder Architecture used a SELECT to evaluate **d** to determine the Output **y**.
- Both **d** and **y** are defined as an Array (or bus or vector) Data Type.
- The last state for WHEN OTHERS is added for the other logic states (Z, X, H, L, etc.).

Include an Enable Input (g) -1

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY decode3a IS
   PORT(
       d  : IN  STD_LOGIC_VECTOR (1 downto 0);
       g  : IN  STD_LOGIC;
       y  : OUT STD_LOGIC_VECTOR (3 downto 0));
END decode3a;

ARCHITECTURE decoder OF decode3a IS
   SIGNAL inputs : STD LOGIC VECTOR (2 downto 0);
BEGIN
   inputs (2)           <= g;
   inputs (1 downto 0)  <= d;
   WITH inputs SELECT
      y <=     "0001" WHEN "000",
               "0010" WHEN "001",
               "0100" WHEN "010",
               "1000" WHEN "011",
               "0000" WHEN others;
END decoder;
```

Concatenate g (1 bit) and d (2 bits) to get 3-bit vector

g
d(1)
d(0)

default case

y(0)
y(3)

Include an Enable Input (g) -2

```
ARCHITECTURE decoder OF decode3a IS
   SIGNAL inputs : STD LOGIC VECTOR (2 downto 0);
BEGIN
   inputs (2)            <= g;
   inputs (1 downto 0)   <= d;
   WITH inputs SELECT
      y <=     "0001" WHEN "000",
               "0010" WHEN "001",
               "0100" WHEN "010",
               "1000" WHEN "011",
               "0000" WHEN others;
END decoder;
```

Concatenate g (1 bit) and d (2 bits) to get 3-bit vector

g

d(1)

d(0)

default case

y(0)

y(3)

To include **g** and **d** in a single vector, we create a signal called **inputs,** a vector with three elements in the sequence **g, d(1), d(0).** When assigning the **d** to the last two elements of **inputs,** we must be explicit about which elements of **inputs** we want to use. Since **d** only contains two elements and we are assigning them to two elements of **inputs,** we don't need to list the elements of **d** explicitly.

We can use a selected signal assignment statement to evaluate all inputs, including **g,** and assign outputs accordingly. When **g** = '0', the decoder outputs are assigned the same as they were in the example without the enable input. The

Conditional Signal Assignment Statement -1

```
__signal <= __expression WHEN __boolean_expression ELSE
            __expression WHEN __boolean_expression ELSE
            __expression;
```

The first Boolean expression in the statement is evaluated. If it is true, the corresponding expression is assigned to the signal. If false, the next Boolean expression is evaluated, and so on until a true Boolean expression is found. If none are true, the signal is assigned a default expression, listed last in the statement.

Conditional Signal Assignment Statement -2

```
--decode4g.vhd
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY decode4g IS
   PORT (
        d  : IN  INTEGER Range 0 to 3;
        g  : IN  STD_LOGIC;
        y  : OUT STD_LOGIC_VECTOR (0 to 3));
END decode4g;

ARCHITECTURE a OF decode4g IS
BEGIN
   y <=    "1000" WHEN (d=0 and g='0') ELSE
           "0100" WHEN (d=1 and g='0') ELSE
           "0010" WHEN (d=2 and g='0') ELSE
           "0001" WHEN (d=3 and g='0') ELSE
           "0000";
END a;
```

Assign this value to y

under this condition

STD_LOGIC

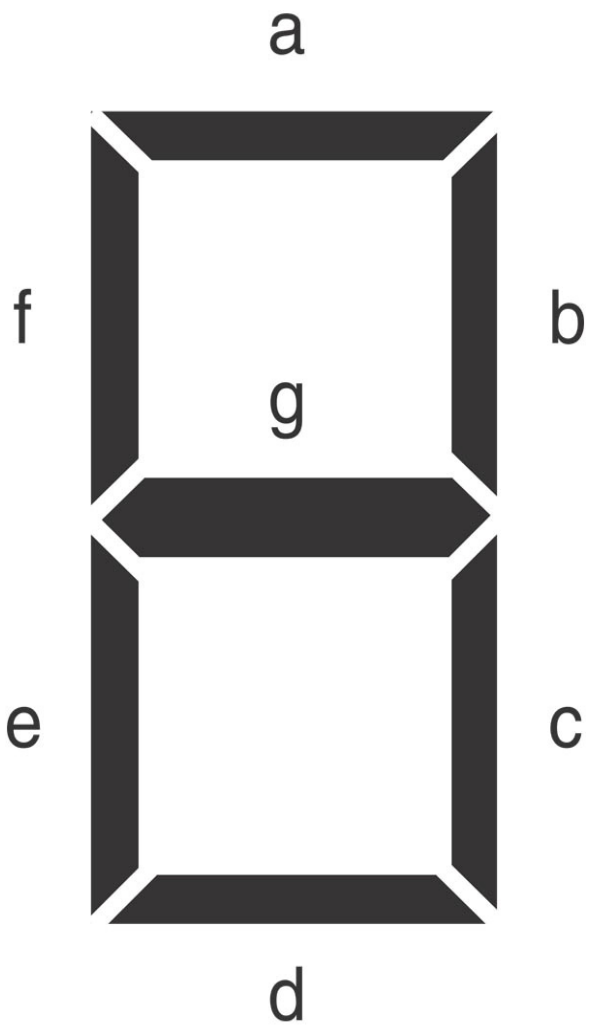
INTEGER

default case

Seven-Segment Displays

- **Seven-Segment Display:** An array of seven independently controlled LEDs shaped like an 8 that can be used to display decimal digits.

Seven-Segment Displays



Seven-Segment Displays

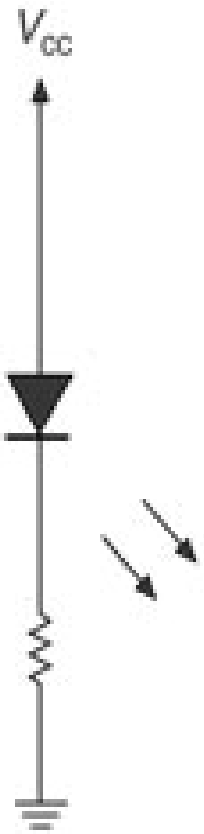
Common Anode Display

- Common Anode Display (CA): A seven-segment display where the anodes of all the LEDs are connected together to V_{CC} and a '0' turns on a segment (a to g).

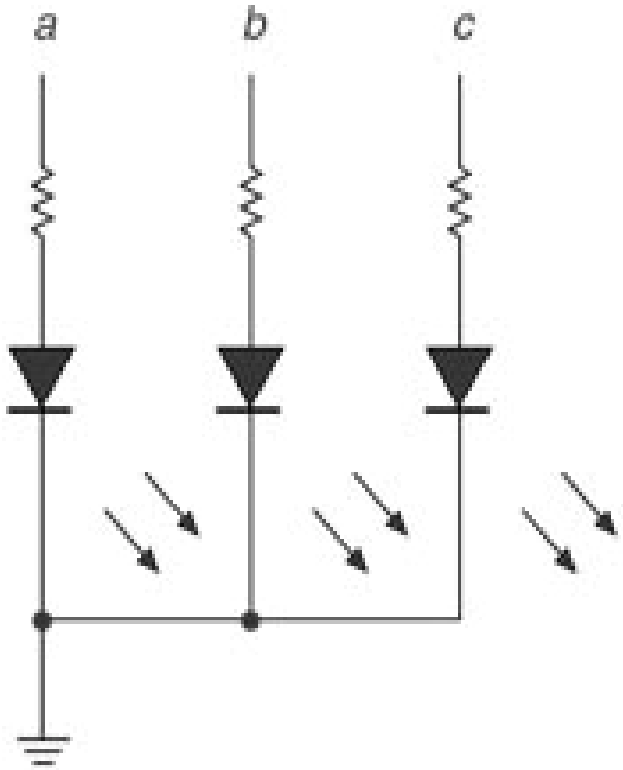
Common Cathode Display

- Common Cathode Display (CC): A seven-segment display where all the cathodes are connected and tied to ground, and a ‘1’ turns on a segment.

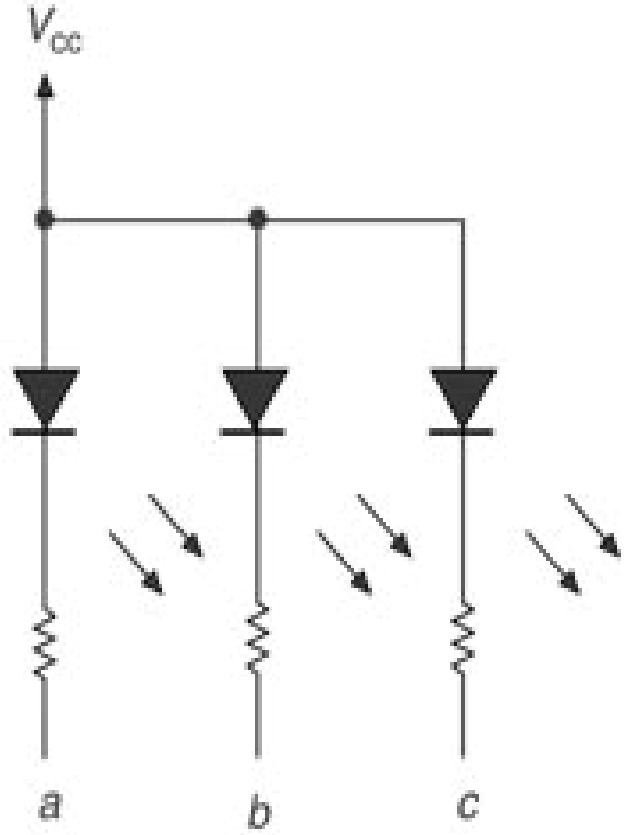
LED Display



a. Circuit requirements for an illuminated LED

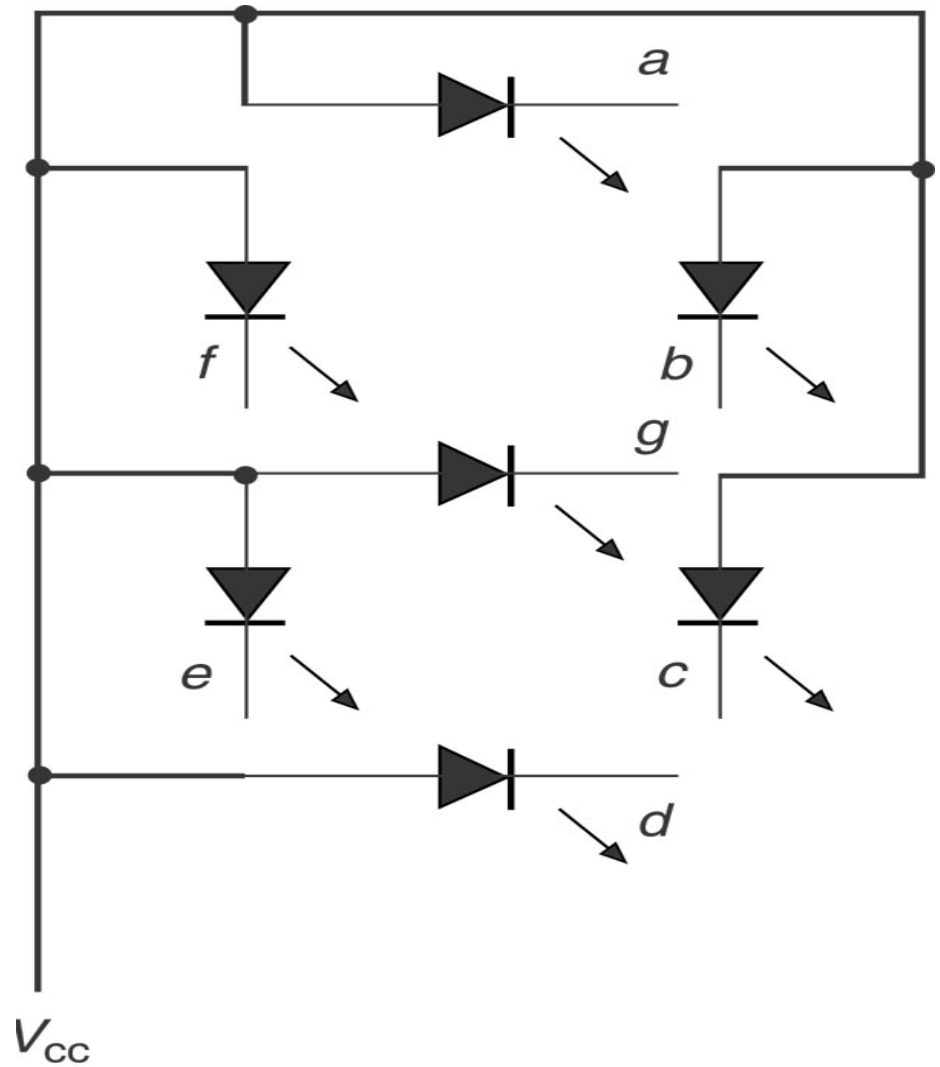


b. Common cathode



b. Common anode

Common Anode Display



Seven-Segment Decoder/Driver – 1

- Receives a BCD (Binary Coded Decimal) 4-Bit inputs, a BCD digit 0000 – 1001 (0 through 9).
- Generates Outputs (a–g) for each of the display LEDs.
- Requires a current limit series resistor for each segment.

Seven-Segment Decoder/Driver – 2

- Decoders for a CC-SS have **active high** outputs while decoders for a CA-SS have **active low** outputs (a to g).
- The outputs generated for the binary input combinations of 1010 to 1111 are "don't cares".
- The decoder can be designed with VHDL.

Example 6.7

Sketch the segment patterns required to display all 16 hexadecimal digits on a seven-segment display. What changes from the patterns in Figure 6.16 need to be made?

■ Solution

The segment patterns are shown in Figure 6.19.

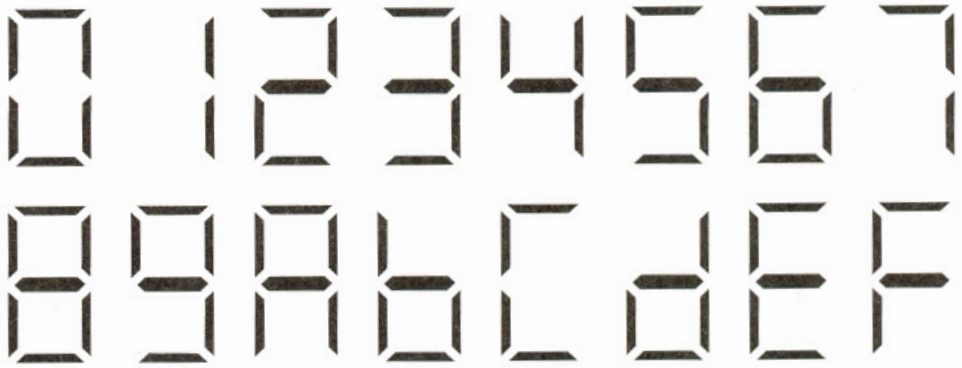


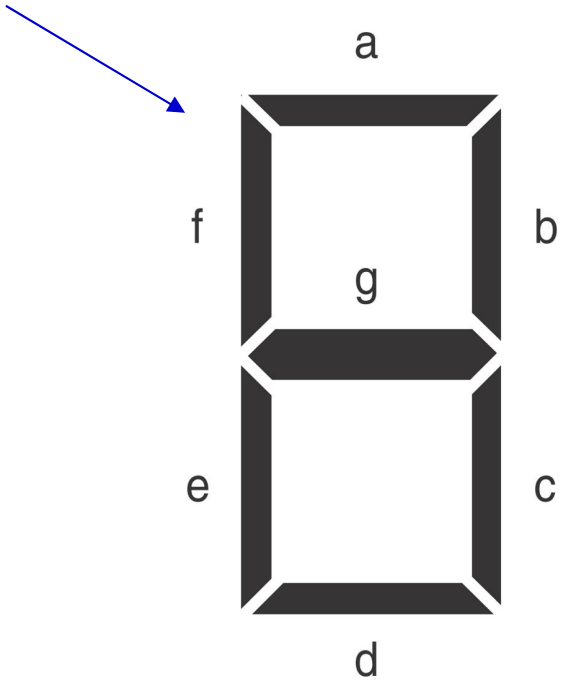
FIGURE 6.19 Example 6.7: Hexadecimal Digit Display Format

Hex digits B and D must be displayed as lowercase letters, b and d, to avoid confusion between B and 8 and between D and 0. To make 6 distinct from b, 6 must be given a tail (segment *a*) and to make 6 and 9 symmetrical, 9 should also have a tail (segment *d*).

Common Anode BCD-to-Seven-Segment Decoder -1

TABLE 6.6 Truth Table for Common Anode BCD-to-Seven-Segment Decoder

Digit	D_3	D_2	D_1	D_0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	1	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	1	1	0	0
Invalid Range	1	0	1	0	X	X	X	X	X	X	X
	1	0	1	1	X	X	X	X	X	X	X
	1	1	0	0	X	X	X	X	X	X	X
	1	1	0	1	X	X	X	X	X	X	X
	1	1	1	0	X	X	X	X	X	X	X
	1	1	1	1	X	X	X	X	X	X	X



The illumination of each segment is determined by a Boolean function of the input variables, $D_3D_2D_1D_0$. From the truth table, the function for segment a is

$$a = \bar{D}_3\bar{D}_2\bar{D}_1D_0 + \bar{D}_3D_2\bar{D}_1\bar{D}_0 + \bar{D}_3D_2D_1\bar{D}_0$$

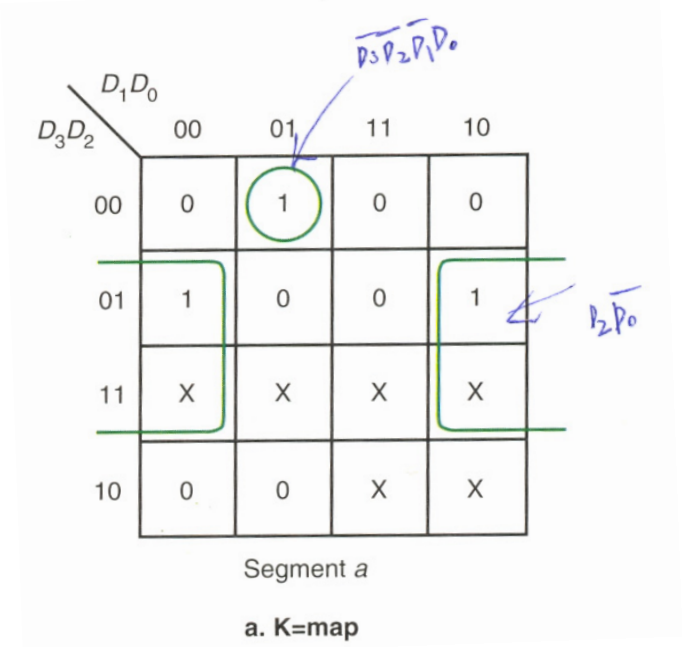
(Since the display is active-LOW, this means segment a is OFF for digits 1, 4, and 6.)

Common Anode BCD-to-Seven-Segment Decoder -2

If we assume that inputs 1010 to 1111 are never going to be used ("don't care states," symbolized by X), we can make any of these states produce HIGH or LOW outputs, depending on which is most convenient for simplifying the segment functions. Figure 6.20a shows a Karnaugh map simplification for segment *a*. The resultant function is

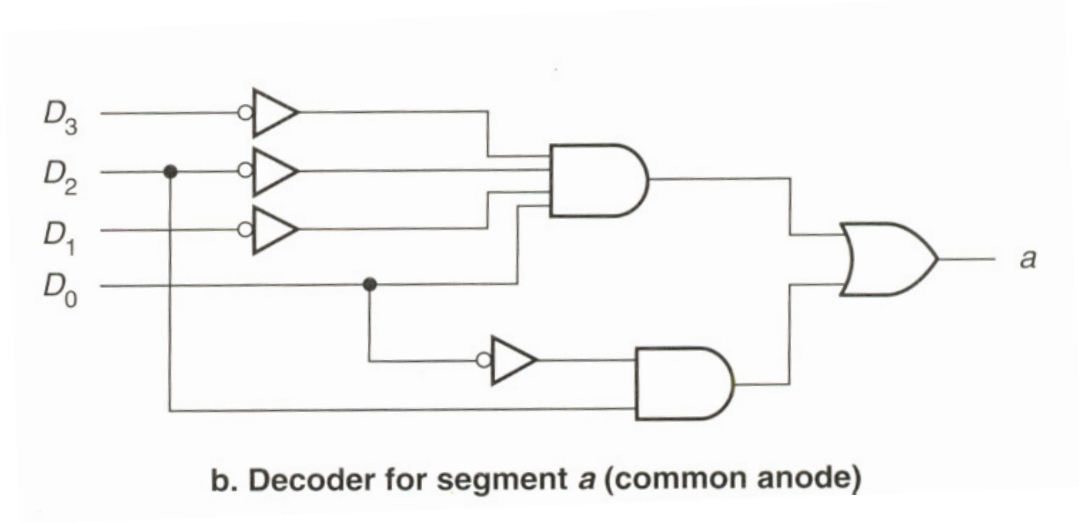
$$a = \bar{D}_3\bar{D}_2\bar{D}_1D_0 + D_2\bar{D}_0$$

The corresponding partial decoder is shown in Figure 6.20b.



Segment *a*

a. K=map



b. Decoder for segment *a* (common anode)

Common Anode BCD-to-Seven-Segment Decoder -3

```
--bcd_7seg.vhd
--BCD-to-seven-segment decoder
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bcd_7seg IS
   PORT(
       d3, d2, d1, d0       : IN  STD_LOGIC;
       a, b, c, d, e, f, g : OUT STD_LOGIC);
END bcd_7seg;

ARCHITECTURE seven_segment OF bcd_7seg IS
   SIGNAL input  : STD_LOGIC_VECTOR (3 downto 0);
   SIGNAL output : STD_LOGIC_VECTOR (6 downto 0);
BEGIN
   input <= d3 & d2 & d1 & d0;
   WITH input SELECT
      output <= "0000001" WHEN "0000", -- display 0
                "1001111" WHEN "0001", -- display 1
                "0010010" WHEN "0010", -- display 2
                "0000110" WHEN "0011", -- display 3
                "1001100" WHEN "0100", -- display 4
                "0100100" WHEN "0101", -- display 5
                "1100000" WHEN "0110", -- display 6
                "0001111" WHEN "0111", -- display 7
                "0000000" WHEN "1000", -- display 8
                "0001100" WHEN "1001", -- display 9
                "1111111" WHEN others;
```

```
   a <= output (6);
   b <= output (5);
   c <= output (4);
   d <= output (3);
   e <= output (2);
   f <= output (1);
   g <= output (0);

END seven_segment;
```

SS VHDL File Description

- In the preceding example file, a concurrent select signal assignment was used (WITH (signals) SELECT.

- The intermediate output signals were mapped to the segments (a to g).

Ripple Blanking – 1

- Ripple Blanking: A technique used in a multiple-digit display that suppresses leading/trailing zeros but allows internal zeros to be displayed.

- Uses a $\overline{RBI}$ (active-LOW) Input and a $\overline{RBO}$ (active-LOW) output.

- When $D_0 - D_3 = 0000$ and $\overline{RBI} = 0$, the display is blank(空白).

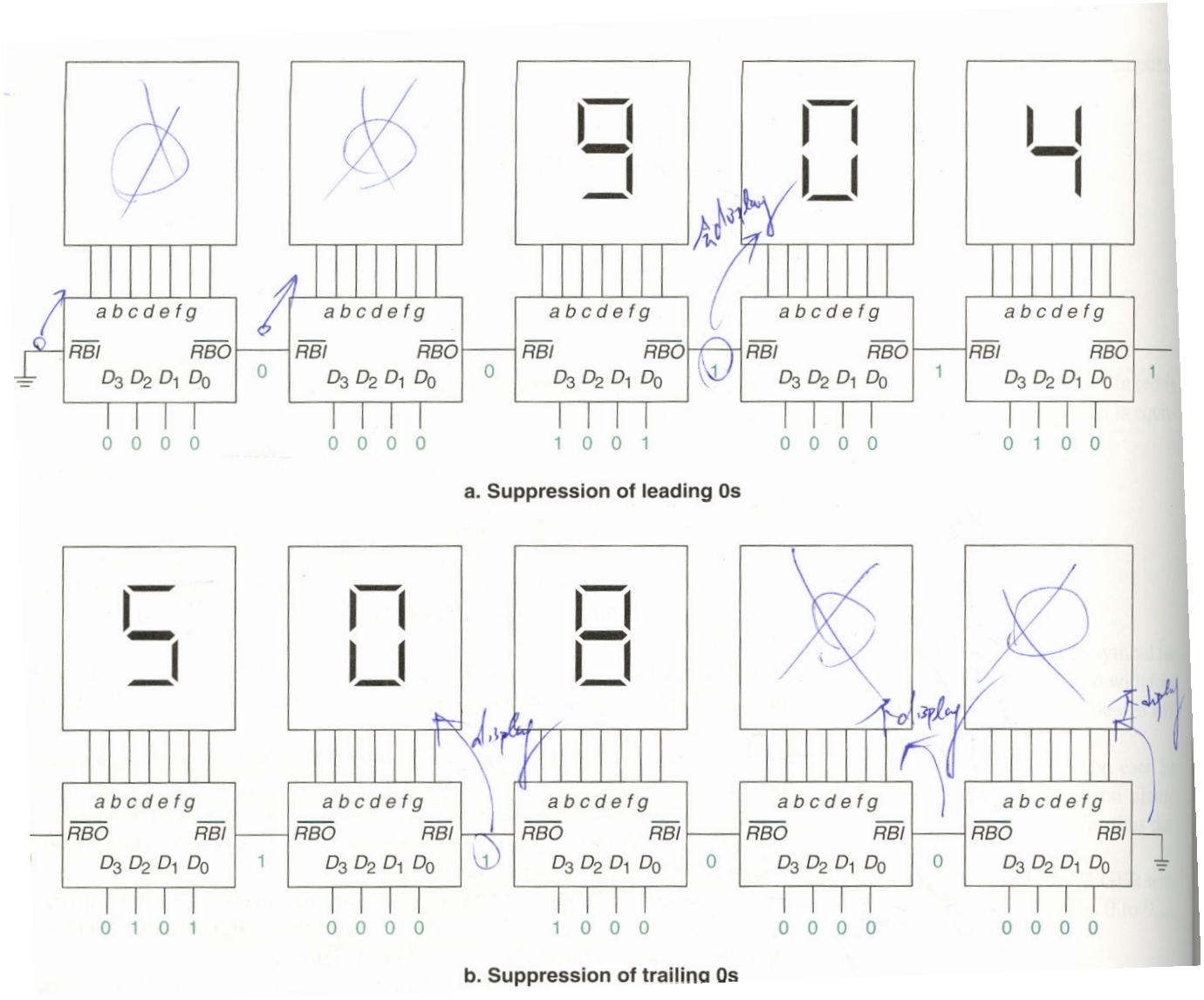
Ripple Blanking – 2

- When $D_0 - D_3 = 0000$ and $\overline{RBI} = 1$, the display shows a '0' (zero).

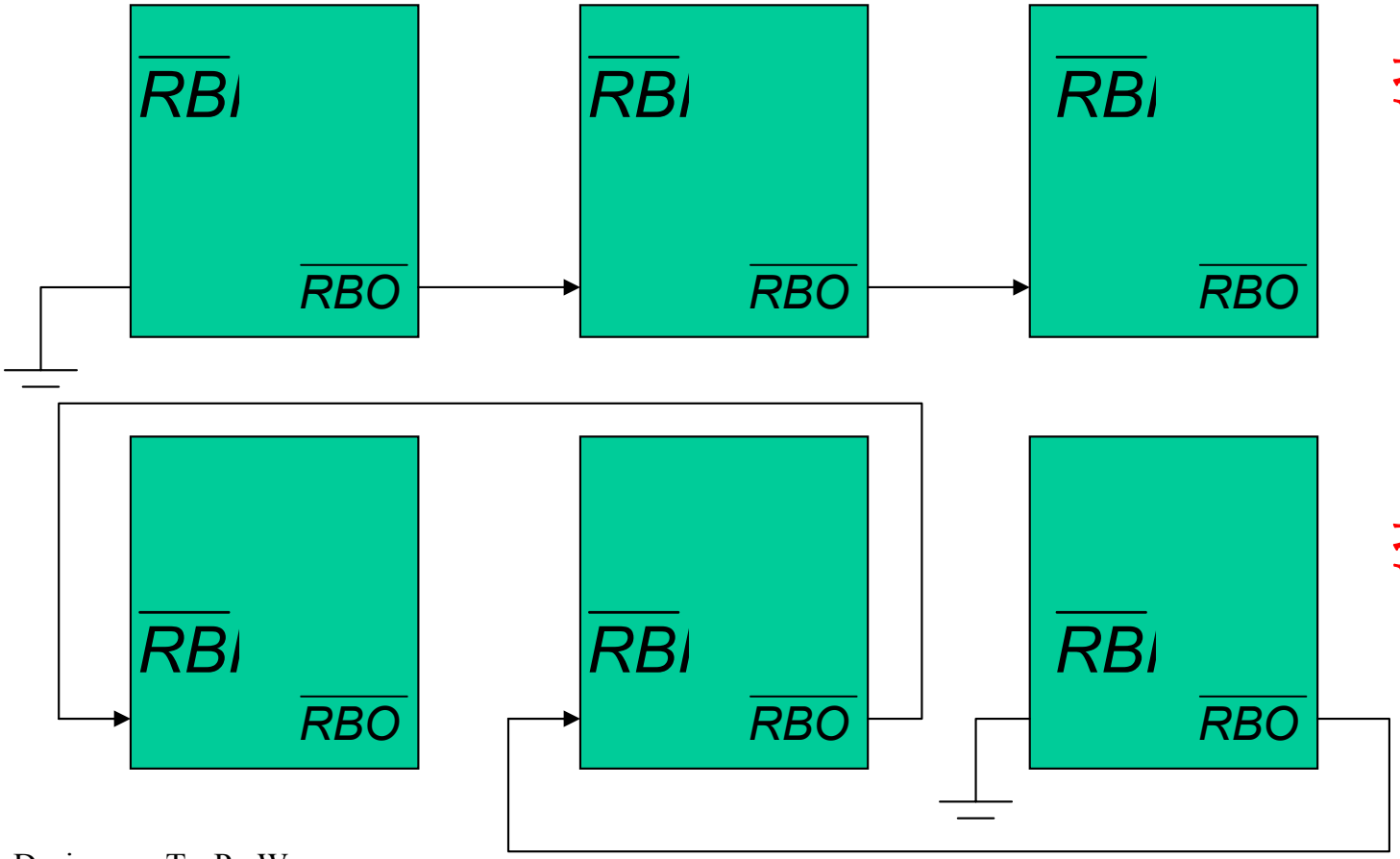
- If $\overline{RBI} = 1$ or $D_0 - D_3 \neq 0000$, then $\overline{RBO} = 1$.

- To suppress leading zeros, connect $\overline{RBI}$ of MS Digit to Gnd, and $\overline{RBO}$ to $\overline{RBI}$ of the next least significant display.

Ripple Blanking – 3



Ripple Blanking – 4



Sequential Process in VHDL

- A VHDL Process is a construct that encloses sequential statements that are executed when a signal in a **sensitivity list** changes.

Sequential Process Basic Format

- Basic Format:

```
PROCESS(Sensitivity List)
BEGIN
        Sequential Statements;
END PROCESS;
```

VHDL for Ripple Blanking -1

```
-- sevsegrb.vhd
-- BCD-to-seven-segment decoder with ripple blanking
-- Leading or trailing zeroes can be suppressed in a
-- multi-digit display.
-- A zero is displayed IF input=0000 and nRBI=1.
-- A zero is suppressed (display blank) if input=0000 and nRBI=0.
-- nRBO = 0 for nrBI = 0 and input = 0000. Otherwise nRBO = 1.

ENTITY sevsegrb IS
    PORT (
        -- Use separate I/Os, not bus
        nRBI, d3, d2, d1, d0      : IN  BIT;
        a, b, c, d, e, f, g, nRBO: OUT BIT);
END sevsegrb;

ARCHITECTURE seven_segment OF sevsegrb IS
   SIGNAL input: BIT_VECTOR (3 DOWNTO 0);   -- Bit vectors for internal
   SIGNAL output: BIT_VECTOR (6 DOWNTO 0); -- use in decoder statements
BEGIN
   -- Concatenate inputs to make bit vector
   input  <=  d3 & d2 & d1 & d0;
```

VHDL for Ripple Blanking -2

```
-- Process Statement
PROCESS (input, nRBI)
BEGIN
IF(nRBI = '0' and input = "0000" THEN
   output  <= "1111111"; -- 0 suppressed
   nRBO    <= '0';       -- Next 0 suppressed
ELSE
    nRBO   <= '1';       -- Next 0 displayed
    CASE input IS
        WHEN "0000"   =>
            output <= "0000001"; -- 0 displayed
        WHEN "0001"   =>
            output <= "1001111"; -- 1
        WHEN "0010"   =>
            output <= "0010010"; -- 2
        WHEN "0011"   =>
            output <= "0000110"; -- 3
        WHEN "0100"   =>
            output <= "1001100"; -- 4
        WHEN "0101"   =>
            output <= "0100100"; -- 5
        WHEN "0110"   =>
            output <= "1100000"; -- 6
        WHEN "0111"   =>
            output <= "0001111"; -- 7
        WHEN "1000"   =>
            output <= "0000000"; -- 8
        WHEN "1001"   =>
            output <= "0001100"; -- 9
        WHEN others   =>
            output <= "1111111"; -- blank
    END CASE;
END IF;
END PROCESS assign_out;

-- Separate the output vector to make individual pin outputs.
      a  <= output (6);
      b  <= output (5);
      c  <= output (4);
      d  <= output (3);
      e  <= output (2);
      f  <= output (1);
      g  <= output (0);

END seven_segment;
```

CASE Statement

- In the RB Design, we replaced the Selected Signal Assignment (With Select) with a Case Statement to generate the outputs for the SS Display.

- The Process Steps are evaluated in sequence:
  - First the IF statements, then the Case, and so on.

IF THEN ELSE

- The IF THEN ELSE Statements were used for conditional testing of some inputs (the data and RBI).
- IF the data is this value THEN do these statements ELSE do this.
- This is a very simple statement that is used a great deal in sequential logic.

6.2 Encoders

- **Encoder:** A circuit that generates a binary code at its outputs in response to one or more active inputs.

- It is complementary in function to a decoder.

3-Bit Binary Encoder -1

Figure 6.22 shows a 3-bit binary encoder. The circuit generates a unique 3-bit binary output for every active input provided *only one input* is active at a time.

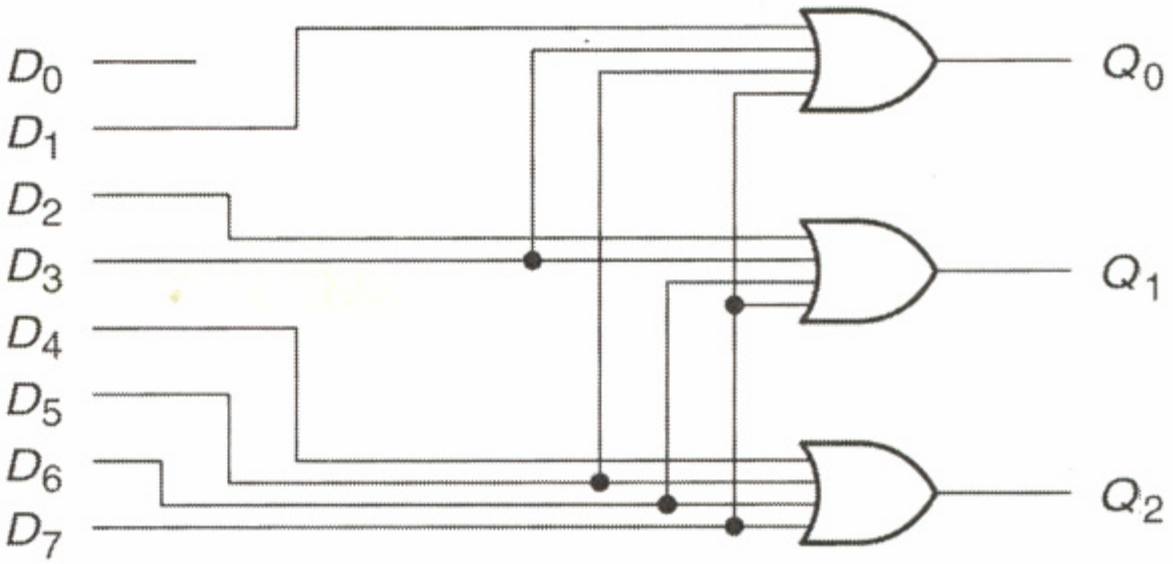


FIGURE 6.22 3-Bit Encoder (No Input Priority)

3-Bit Binary Encoder -2

The encoder has only 8 permitted input states out of a possible 256. Table 6.7 shows the allowable input states, which yield the Boolean equations used to design the encoder. These Boolean equations are:

$$Q_2 = D_7 + D_6 + D_5 + D_4$$
$$Q_1 = D_7 + D_6 + D_3 + D_2$$
$$Q_0 = D_7 + D_5 + D_3 + D_1$$

The D_0 input is not connected to any of the encoding gates, as all outputs are in their LOW (inactive) state when the 000 code is selected.

TABLE 6.7 Partial Truth Table for a 3-Bit Encoder

D_7	D_6	D_5	D_4	D_3	D_2	D_1	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	0	0	1	0
0	0	0	0	1	0	0	0	1	1
0	0	0	1	0	0	0	1	0	0
0	0	1	0	0	0	0	1	0	1
0	1	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	1	1	1

Priority Encoders

- **Priority Encoder:** An encoder that generates a code based on the highest-priority input.

- For example, if input D_3 = input D_5 =‘1’, then the output is 101, not 011. D_5 has a higher priority than D_3 and the output will respond accordingly.

Example 6.8 -1

Figures 6.23 shows a priority encoder with three different combinations of inputs. Determine the resultant output code for each figure. Inputs and outputs are active HIGH.

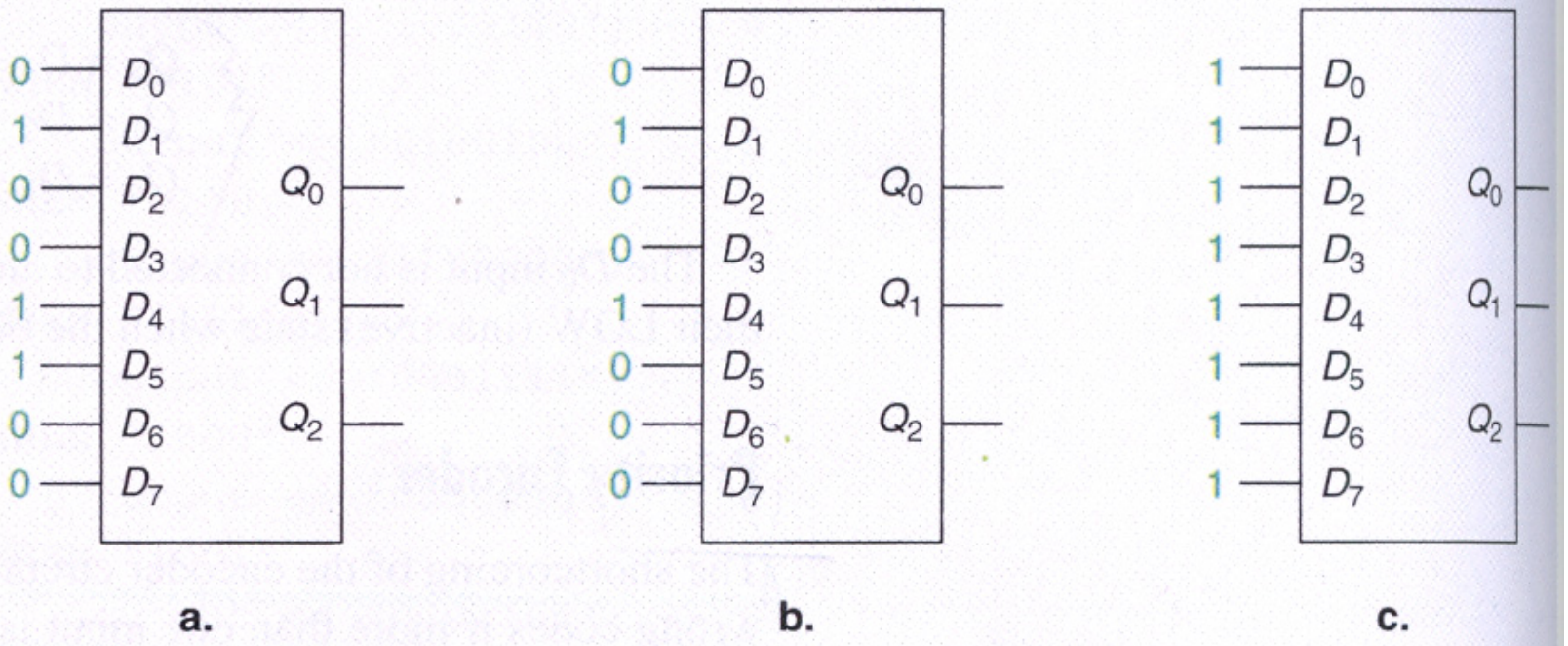


FIGURE 6.23 Example 6.8: Priority Encoder Inputs

Example 6.8 -2

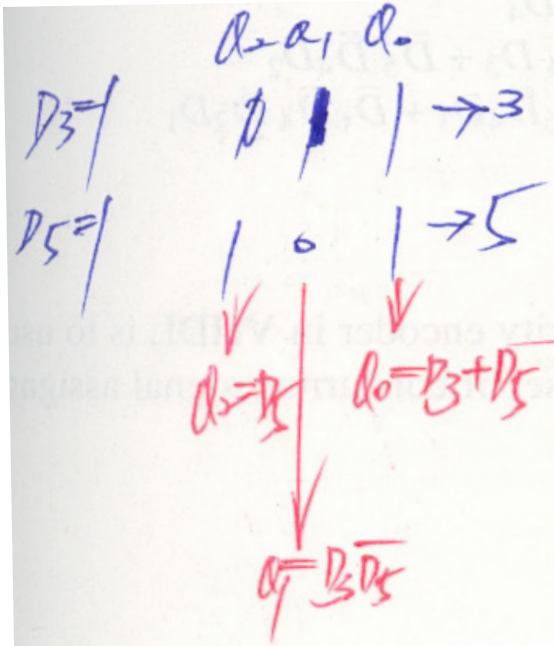
Solution

Figure 6.23a: The highest-priority active input is D_5. D_4 and D_1 are ignored. $Q_2Q_1Q_0 = 101$.

Figure 6.23b: The highest-priority active input is D_4. D_1 is ignored. $Q_2Q_1Q_0 = 100$.

Figure 6.23c: The highest-priority active input is D_7. All other inputs ignored. $Q_2Q_1Q_0 = 111$.

Encoding Principle



For example, if inputs D_3 and D_5 are both active, the correct output code is $Q_2Q_1Q_0 = 101$. The code for D_3 would be $Q_2Q_1Q_0 = 011$. Thus, D_3 must not make $Q_1 = 1$. The Boolean expressions for Q_2Q_1 and Q covering only these two codes are:

$Q_2 = D_5$ (HIGH if D_5 is active.)
$Q_1 = D_3\bar{D}_5$ (HIGH if D_3 is active AND D_5 is NOT active.)
$Q_0 = D_3 + D_5$ (HIGH if D_3 OR D_5 is active.)

Truth Table of 3-Bit Priority Encoder

The truth table of a 3-bit priority encoder is shown in Table 6.8.

TABLE 6.8 Truth Table for a 3-Bit Priority Encoder

D_7	D_6	D_5	D_4	D_3	D_2	D_1	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	X	0	1	0
0	0	0	0	1	X	X	0	1	1
0	0	0	1	X	X	X	1	0	0
0	0	1	X	X	X	X	1	0	1
0	1	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	1	1	1

TABLE 6.9 Binary Outputs and Corresponding Decimal Values

Q_2	Q_1	Q_0	Code Value
1	1	1	7
1	1	0	6
1	0	1	5
1	0	0	4
0	1	1	3
0	1	0	2
0	0	1	1
0	0	0	0

$$Q_2 = D_7 + D_6 + D_5 + D_4$$
$$Q_1 = D_7 + D_6 + \bar{D}_5\bar{D}_4 D_3 + \bar{D}_5 \bar{D}_4 D_2$$
$$Q_0 = D_7 + \bar{D}_6 D_5 + \bar{D}_6\bar{D}_4 D_3 + \bar{D}_6\bar{D}_4 \bar{D}_2 D_1$$

VHDL Priority Encoder

$$Q_2 = D_7 + D_6 + D_5 + D_4$$
$$Q_1 = D_7 + D_6 + \bar{D}_5\bar{D}_4 D_3 + \bar{D}_5 \bar{D}_4 D_2$$
$$Q_0 = D_7 + \bar{D}_6 D_5 + \bar{D}_6\bar{D}_4 D_3 + \bar{D}_6\bar{D}_4 \bar{D}_2 D_1$$

The most obvious way to program a priority encoder in VHDL is to use the equations derived in the previous section in a set of concurrent signal assignment statements, as follows.

```
-- hi_pri8a.vhd
ENTITY hi_pri8a IS

 PORT(
     d   : IN  BIT_VECTOR(7 downto 0);
     q   : OUT BIT_VECTOR (2 downto 0));
END hi_pri8a;

ARCHITECTURE a OF hi_pri8a IS
BEGIN
 -- Concurrent Signal Assignments
 q(2) <= d(7) or d(6) or d(5) or d(4);

 q(1) <= d(7) or d(6)
      or ((not d(5)) and (not d(4)) and d(3))
      or ((not d(5)) and (not d(4)) and d(2));

 q(0) <= d(7) or ((not d(6)) and d(5))
      or ((not d(6)) and (not d(4)) and d(3))
      or ((not d(6)) and (not d(4)) and (not d(2)) and d(1));
END a;
```

Another VHDL Priority Encoder -1

```
-- hi_pri8b.vhd
ENTITY hi_pri8b IS
 PORT(
     d  : IN  BIT_VECTOR (7 downto 0);
     q  : OUT INTEGER RANGE 0 to 7);
END hi_pri8b;

ARCHITECTURE a OF hi_pri8b IS
BEGIN
 --  Conditional Signal Assignment
encoder:
 q  <=  7 WHEN d(7)='1' ELSE
        6 WHEN d(6)='1' ELSE
        5 WHEN d(5)='1' ELSE
        4 WHEN d(4)='1' ELSE
        3 WHEN d(3)='1' ELSE
        2 WHEN d(2)='1' ELSE
        1 WHEN d(1)='1' ELSE
        0;
END a;
```

Another VHDL Priority Encoder -2

Output **q** is defined as type INTEGER. Since it ranges from 0 to 7, the VHDL compiler will automatically assign three outputs: Q_2, Q_1, and Q_0. The conditional signal assignment statement evaluates the first WHEN clause to determine if its condition `(d(7) = '1')` is true. If so, it assigns **q** the value of 7 ($Q_2Q_1Q_0 = 111$). If the first condition is false, the next WHEN clause is evaluated, assigning **q** the value 6 ($Q_2Q_1Q_0 = 110$) if true, and so on until all WHEN clauses have been evaluated. If no clause is true, then the default value (0: $Q_2Q_1Q_0 = 000$) is assigned to the output.

IF Statement

The effect is similar to that of an IF statement, where a sequence of conditions is evaluated, but only one output assignment is made. However, an IF statement must be used within a PROCESS statement, if we choose to use it. The IF statement for a priority encoder is as follows:

```
PROCESS (d)
BEGIN
 IF (d(7) = '1') THEN
        q <= 7;
    ELSIF (d(6) = '1') THEN
        q <= 6;
 .
 .
 .
    ELSIF (d(1) = '1' THEN
        q <= 1;
    ELSE
        q <= 0;
    END IF;
END PROCESS;
```

Example 6.9 -1

Write a set of simulation criteria for the 3-bit priority encoder using a conditional signal assignment statement. Use these criteria to create a simulation in Quartus II.

- With all input bits HIGH, d(7) is highest priority of active inputs. Expected output is 111_2 (= 7_{10}).
- With d(7) LOW and all other inputs HIGH, d(6) is now highest priority active input. Expected output is 110_2 (= 6_{10}).
- With d(7) and d(6) LOW and remaining bits HIGH, highest priority is d(5). Expected output is 101_2 (= 5_{10}).
- As each bit goes LOW in order, output code should count down by one with each step.

Example 6.9 -2

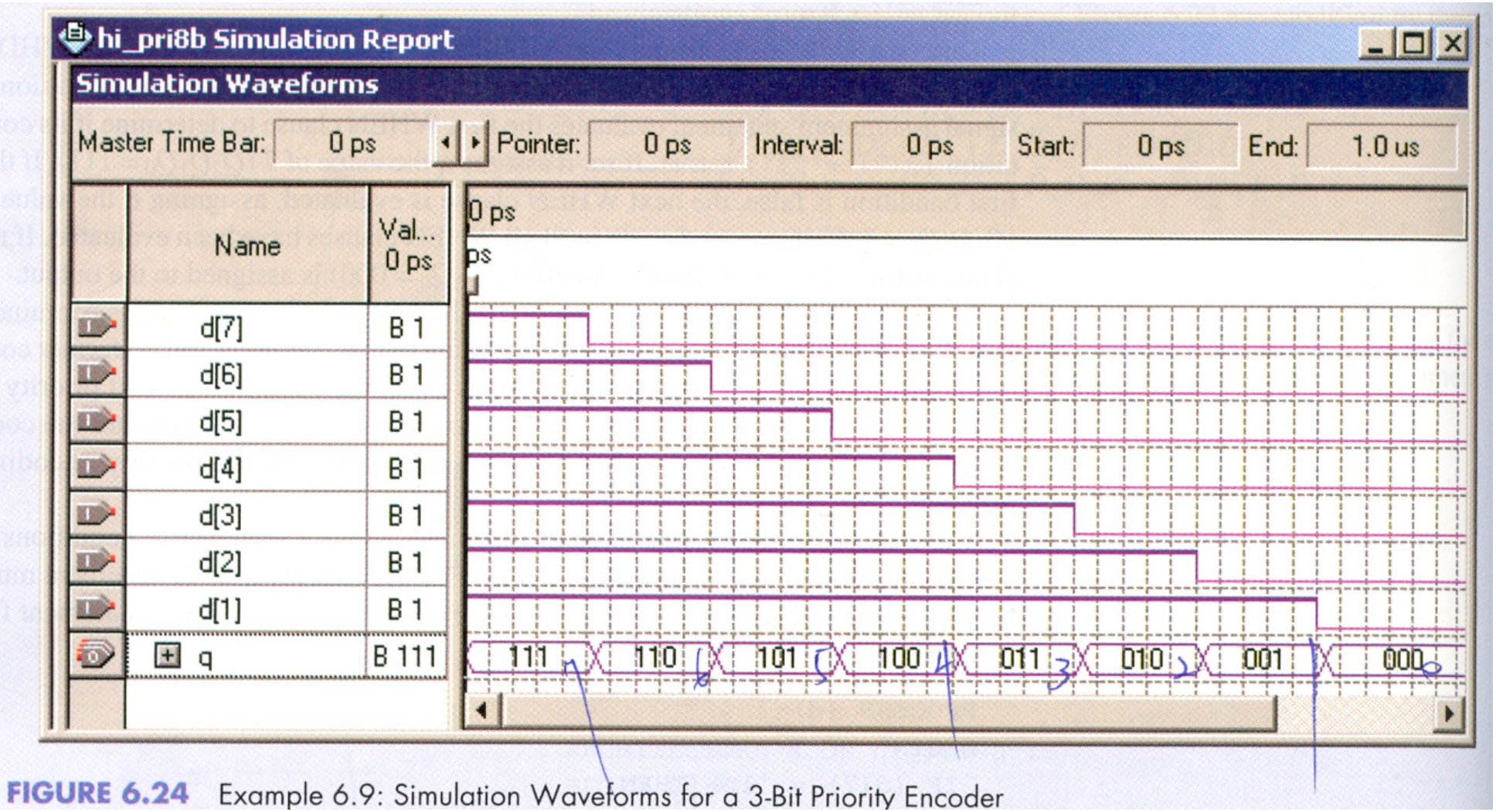


FIGURE 6.24 Example 6.9: Simulation Waveforms for a 3-Bit Priority Encoder

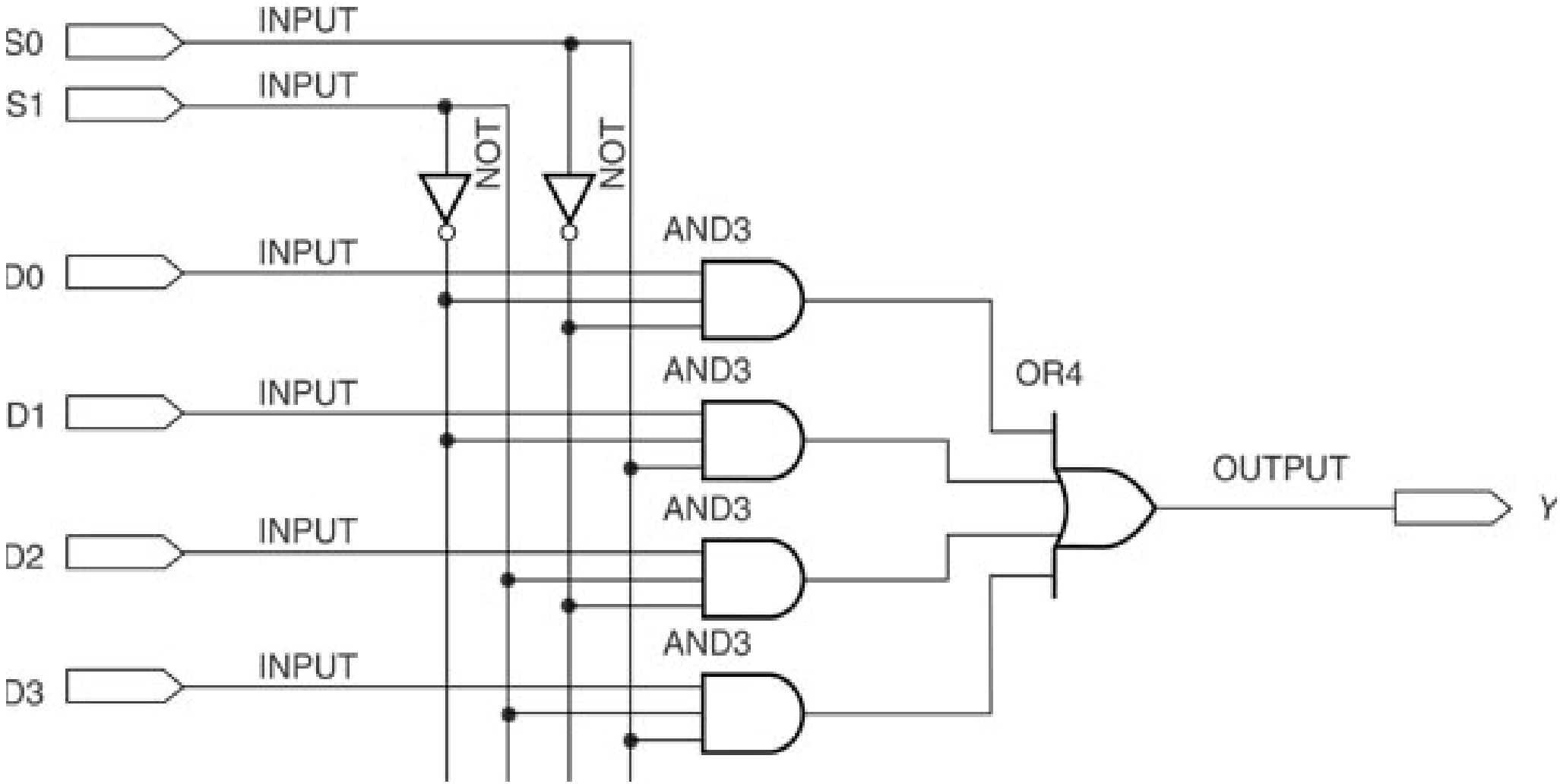
6.3 Basic Multiplexers (MUX)

- **(MUX):** A digital circuit that directs one of several inputs to a single output based on the state of several select inputs.

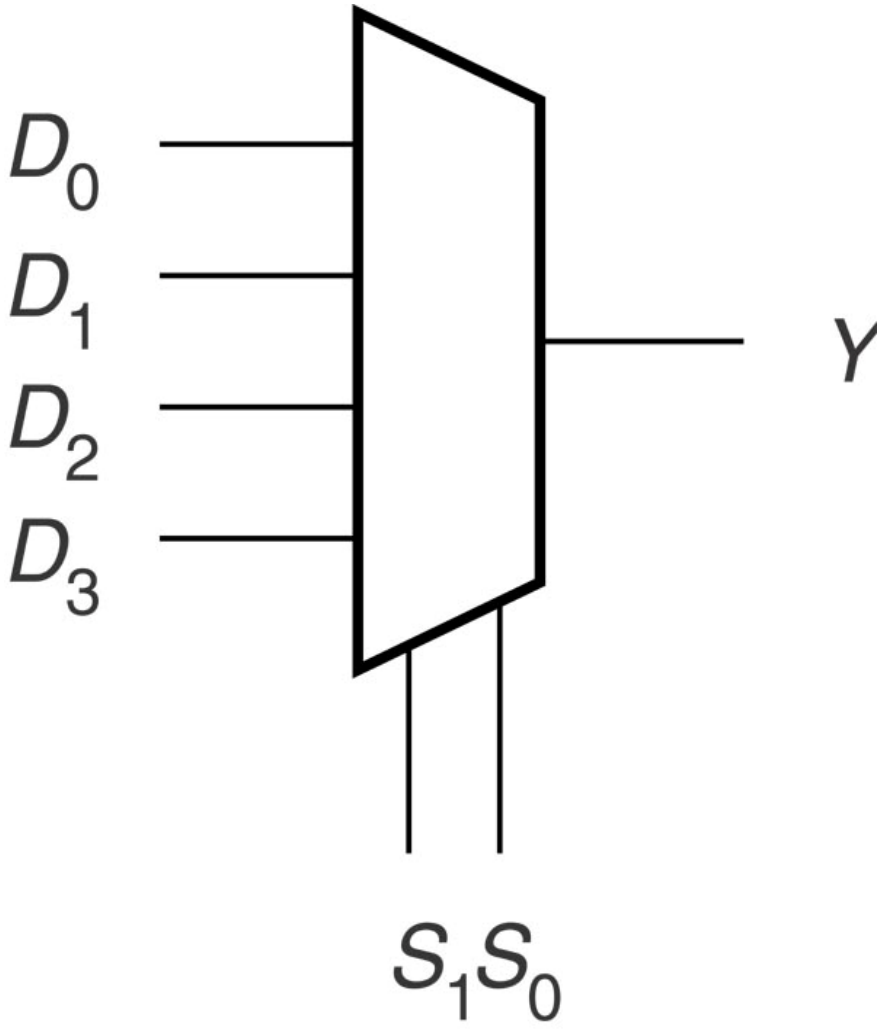
- A MUX is called a *m*-to-1 MUX.

- A MUX with *n* select inputs will require $m = 2^n$ data inputs (e.g., a 4-to-1 MUX requires 2 select inputs S_1 and S_0).

Basic Multiplexers (MUX)



Basic Multiplexers (MUX)



4-to-1 Multiplexers Truth Table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

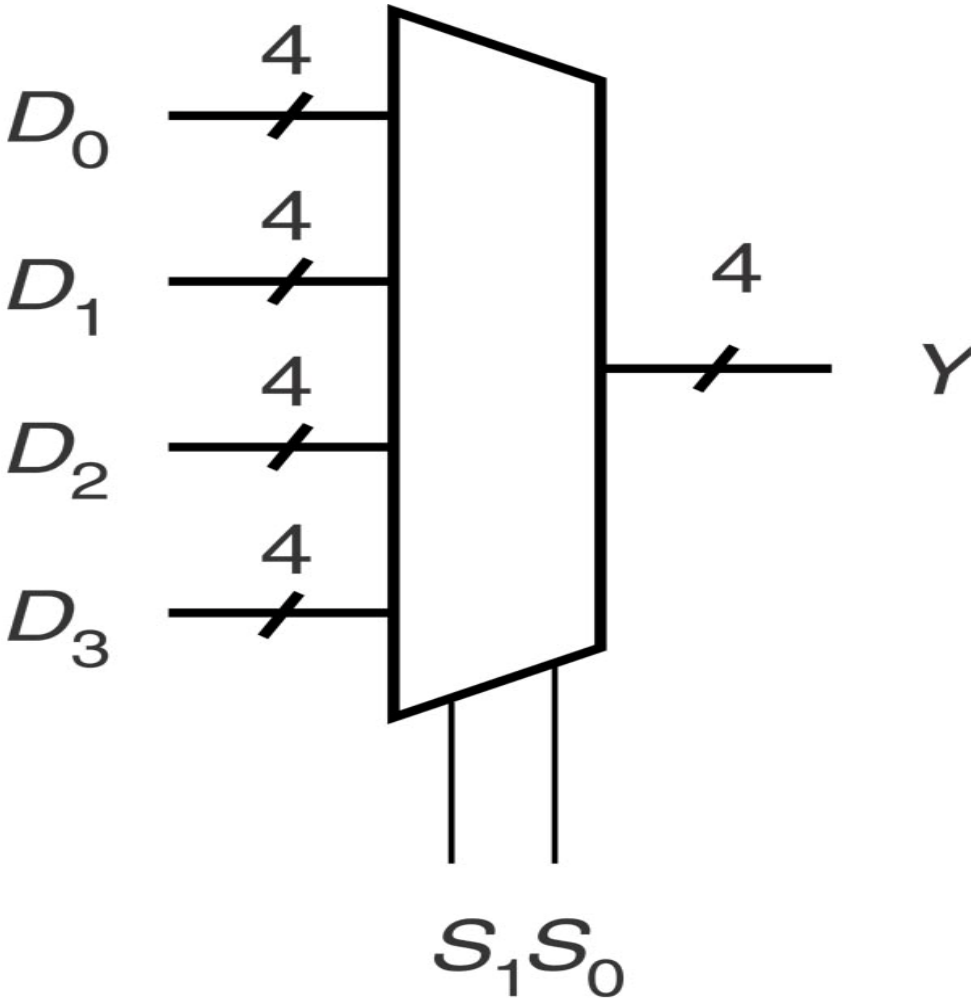
Multiplexer Logic

- Boolean expression for a 4-to-1 MUX is

$$Y = D_0 \overline{S_1}\,\overline{S_0} + D_1 \overline{S_1} S_0 + D_2 S_1 \overline{S_0} + D_3 S_1 S_0$$

- This expression can be expanded to any size MUX so the VHDL architecture could use a very long concurrent Boolean statement.

4-to-1 4-bit Bus MUX



Truth Table for a 4-to-1 4-bit Bus MUX

S_1	S_0	$Y_3 \quad Y_2 \quad Y_1 \quad Y_0$
0	0	$D_{03}\, D_{02}\, D_{01}\, D_{00}$
0	1	$D_{13}\, D_{12}\, D_{11}\, D_{10}$
1	0	$D_{23}\, D_{22}\, D_{21}\, D_{20}$
1	1	$D_{33}\, D_{32}\, D_{31}\, D_{30}$

Double Subscript Notation

- Naming convention in which variables are bundled in numerically related groups, the elements of which are themselves numbered.

- The first subscript identifies the group that a variable belongs to (D_{01}, D_{00}).

- The second subscript indicates which element of the group a variable represents.

VHDL Implementation of Multiplexers

- The following three VHDL constructs can be used to describe the Multiplexer:
  - Concurrent Signal Assignment Statement
  - Selected Signal Assignment Statement
  - CASE Statement within a Process

Concurrent Signal Assignment

Recall that the concurrent signal assignment statement takes the form:

```
__signal <= __expression;
```

```
-- mux4.vhd
-- 4-to-1 multiplexer
-- Directs one of four input signals (d0 to d3) to output,
--   depending on status of select bits (s1, s0).
ENTITY mux4 IS
PORT(
    d0, d1, d2, d3 : IN  BIT;
    s              : IN  BIT_VECTOR (1 downto 0);
    y              : OUT BIT);
END mux4;

ARCHITECTURE mux4to1 OF mux4 IS
BEGIN
-- Concurrent Signal Assignment
y <=  ((not s(1)) and (not s(0)) and d0)
   or ((not s(1)) and (     s(0)) and d1)
   or ((     s(1)) and (not s(0)) and d2)
   or ((     s(1)) and (     s(0)) and d3);
END mux4to1;
```

Selected Signal Assignment

This construct has the following form (the label is optional):

```
__label:
WITH __expression SELECT
__signal <= __expression WHEN __constant_value,
            __expression WHEN __constant_value,
            __expression WHEN __constant_value,
            __expression WHEN __constant_value;
```

```
ENTITY mux4sel IS
 PORT(
     d0, d1, d2, d3 : IN  BIT;
     s              : IN  BIT_VECTOR (1 downto 0);
     y              : OUT BIT);
END mux4sel;

ARCHITECTURE mux4to1 OF mux4sel IS
BEGIN
M: WITH s SELECT
   y  <=  d0 WHEN "00",
          d1 WHEN "01",
          d2 WHEN "10",
          d3 WHEN "11";
END mux4to1;
```

CASE Statement within a Process -1

A CASE statement within a PROCESS has the following form (the label is optional):

```
__process_label:
PROCESS (sensitivity list)
BEGIN
  CASE __expression IS
       WHEN __constant_value =>
            __statement;
            __statement;
       WHEN __constant_value =>
            __statement;
            __statement;
       WHEN OTHERS =>
            __statement;
            __statement;
  END CASE;
END PROCESS __process_label;
```

CASE Statement within a Process -2

```
-- Define inputs and outputs
ENTITY mux4case IS
    PORT(
        d0, d1, d2, d3 : IN   BIT;                        -- data inputs
        s              : IN   BIT_VECTOR (1 downto 0); -- select inpu
        y              : OUT  BIT);
END mux4case;
ARCHITECTURE mux4to1 OF mux4case IS
BEGIN
   -- Monitor select inputs and execute if they change
   PROCESS(s)
   BEGIN
      CASE s IS
         WHEN "00" =>
            y   <= d0;
         WHEN "01" =>
            y   <= d1;
         WHEN "10" =>
            y   <= d2;
         WHEN "11" =>
            y   <= d3;
         WHEN others =>
            y   <= '0';
      END CASE;
   END PROCESS;
END mux4to1;
```

Multiplexer Applications

- Used in directing multiple data sources to a single processing element such as multiple CD Player Streams to a DSP.

- Used in Time Division Multiplexing (TDM) by the Phone Service to multiplex multiple voice channels on a single coax line (or fiber).

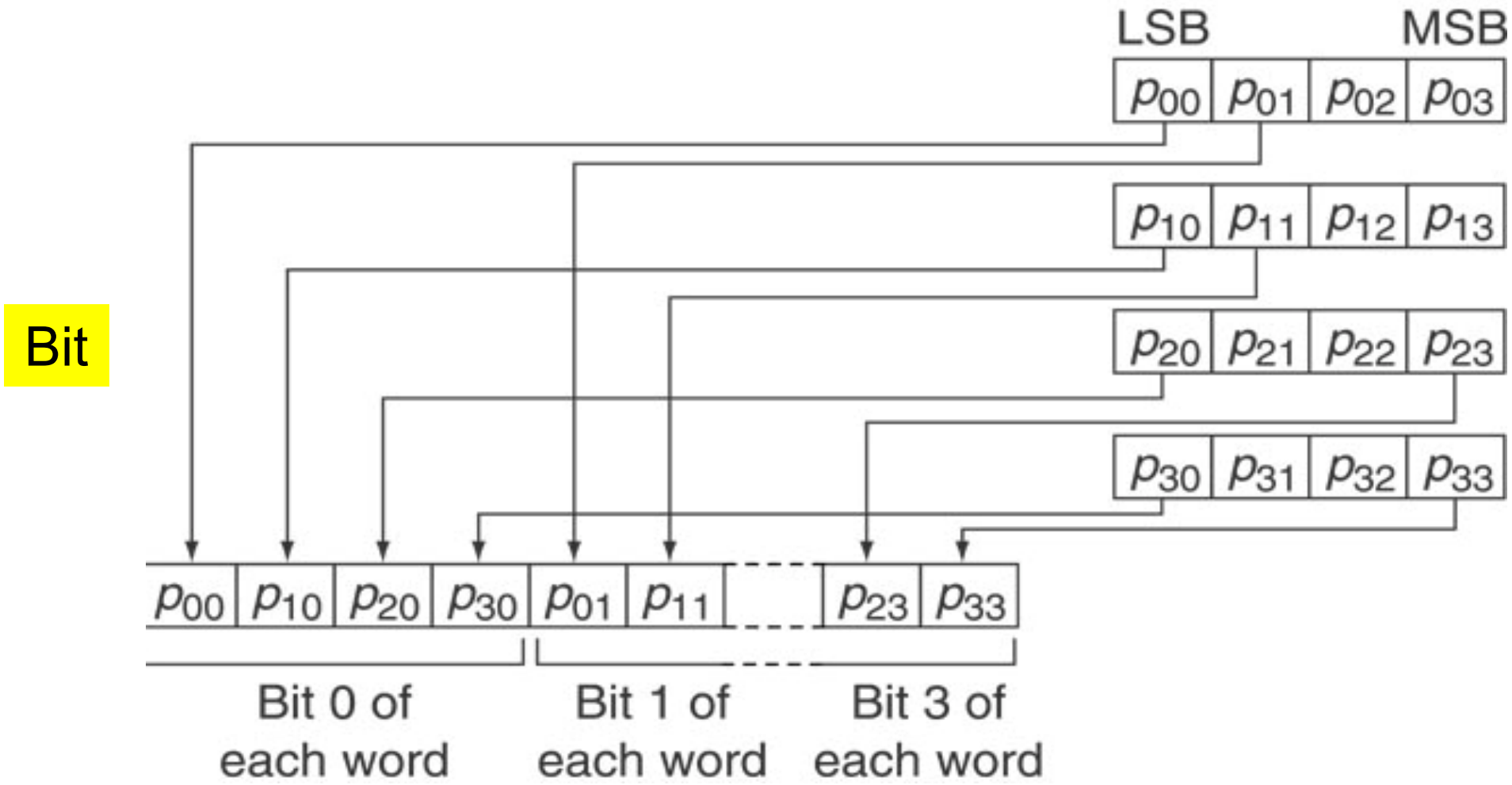
Time Division Multiplexing (TDM)

- Each user has a specific time slot in a TDM data frame. Each frame has 24 users.
- TDM requires a time-dependent (counter) source to synchronize the select lines.
- Each user's time slot repeats on the next frame for more data.
- The links are called T-Carriers (such as a T1 Line).

TDM Data Streams

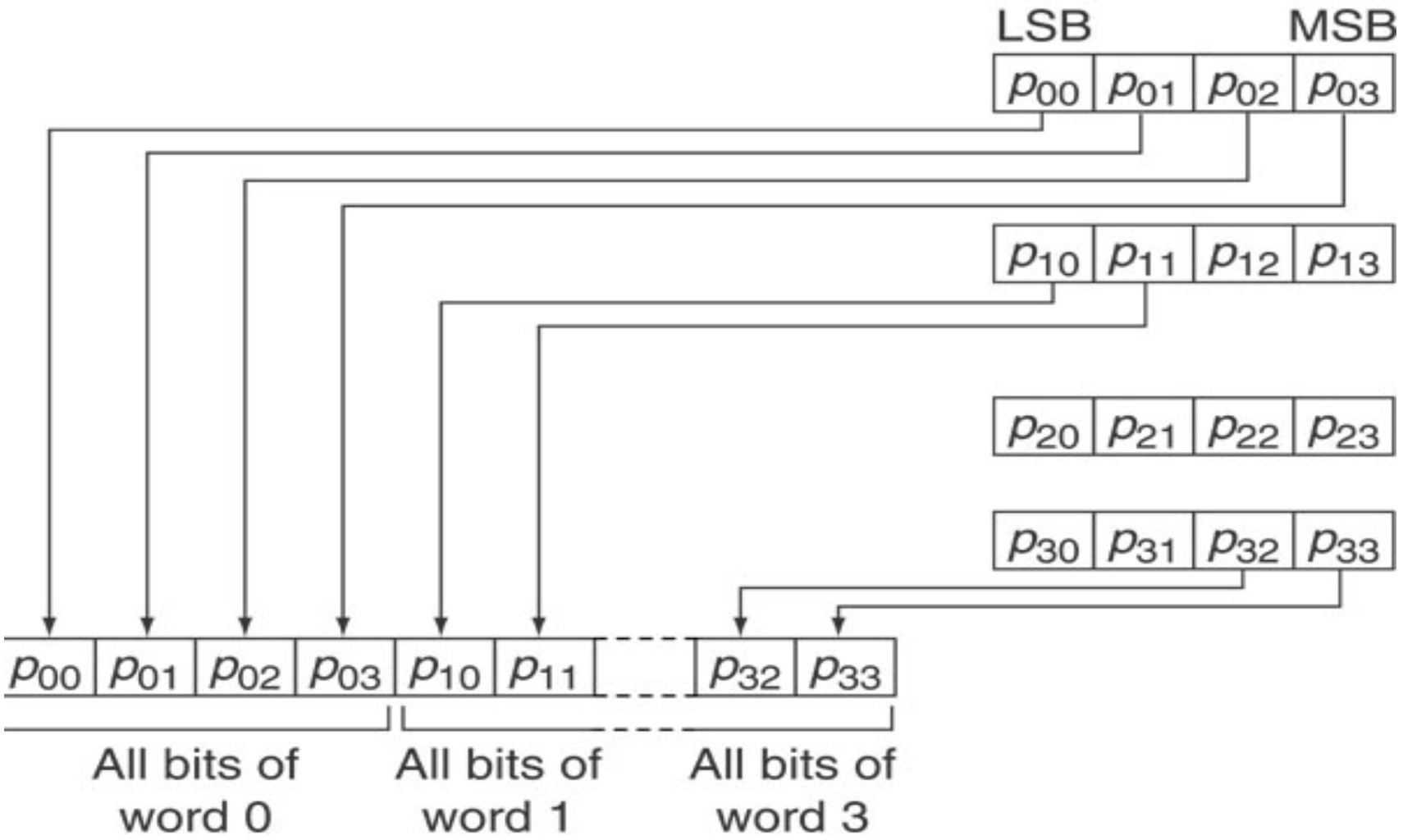
- Two methods in which data is transmitted:
  - Bit Multiplexing: One bit is sent at a time from the channel during the channel's assigned time slot
  - Word Multiplexing: One byte is sent at a time from the channel during the channel's assigned time slot

TDM Data Streams

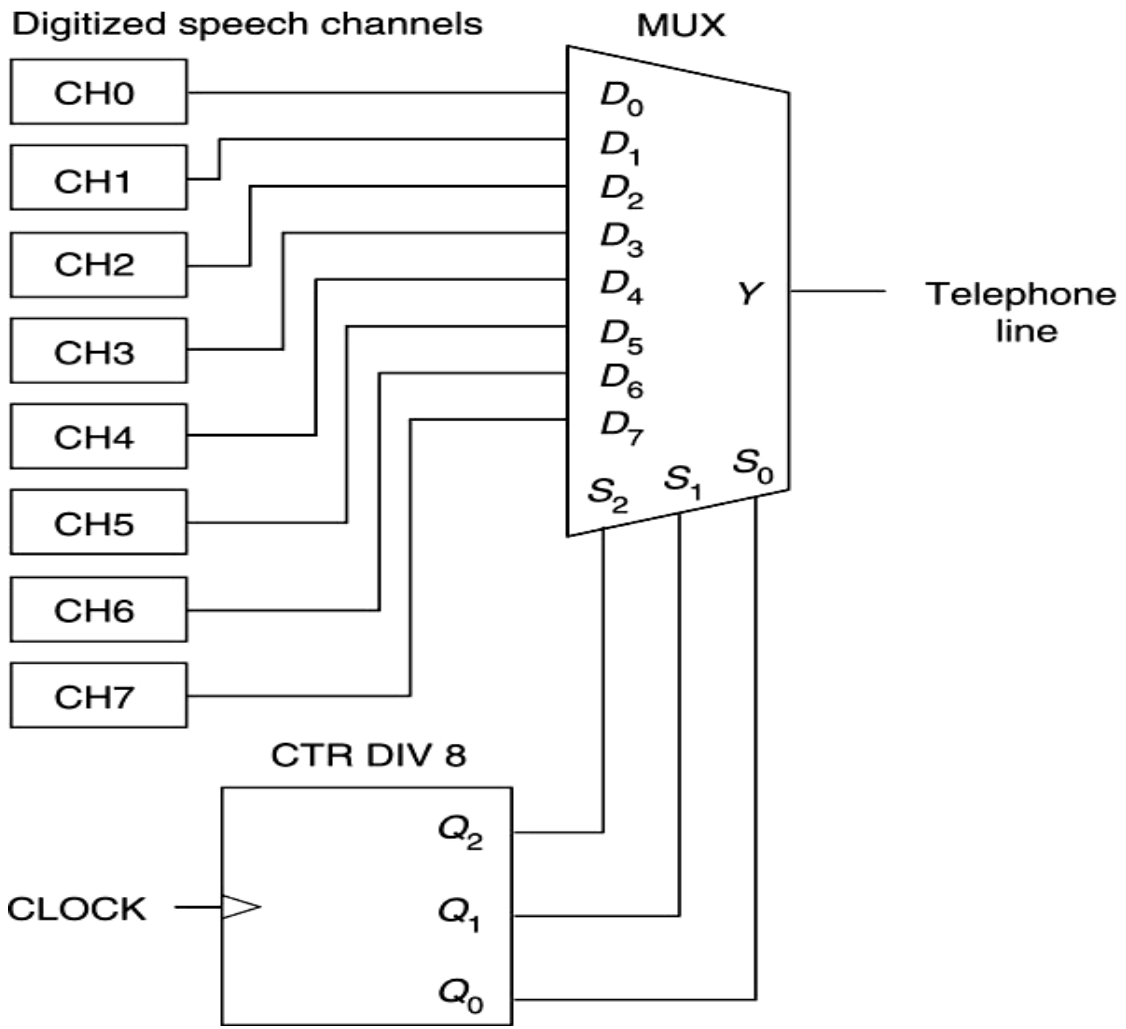


TDM Data Streams

Word



TDM Data Streams

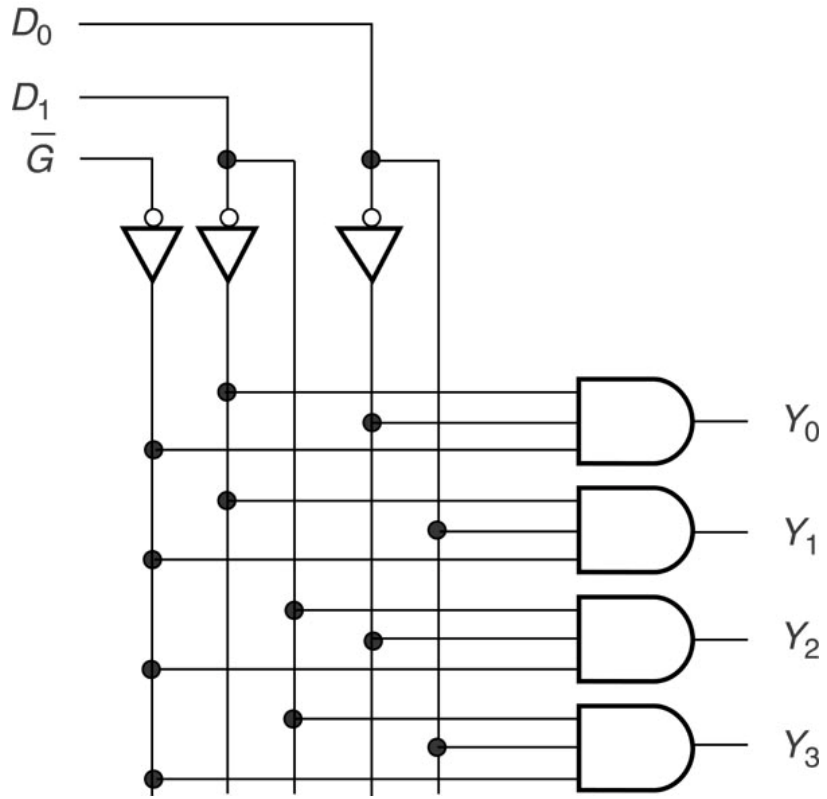


6.4 Demultiplexer Basics – 1

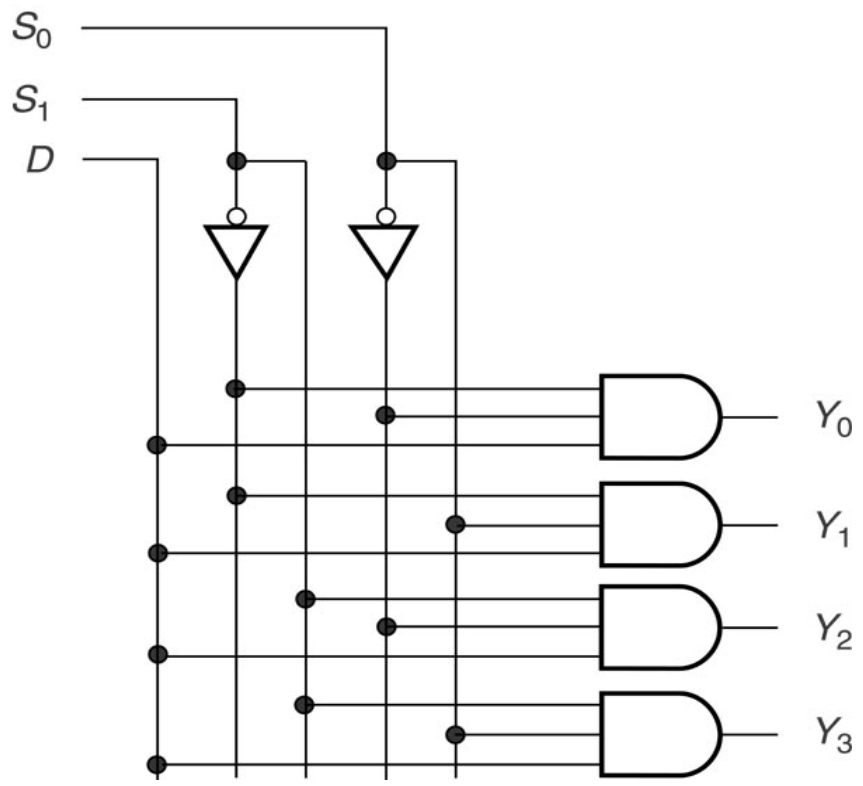
- **Demultiplexer (DMUX):** A circuit that uses a binary decoder to direct a single input signal to one of several outputs.
- A DEMUX performs the reverse operation of a MUX.
- The selected input or output is chosen by the state of an internal decoder.

Demultiplexer Basics – 2

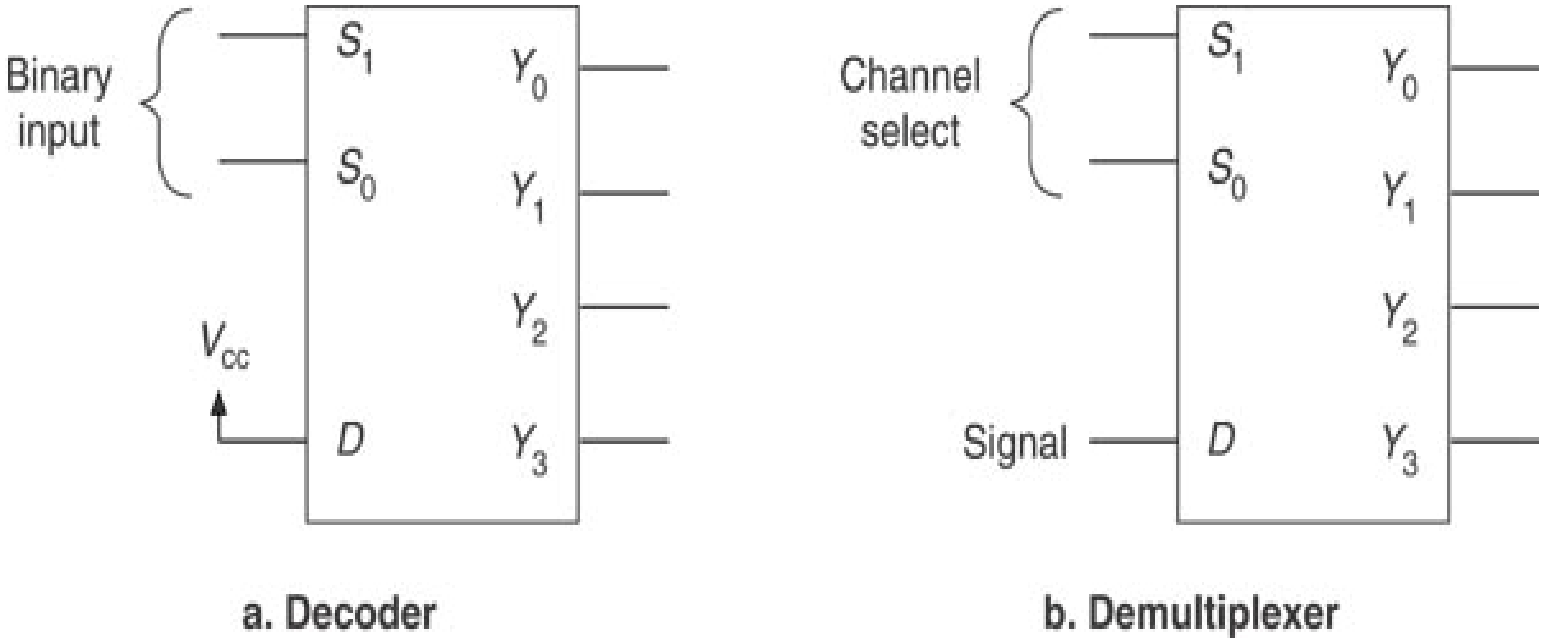
4-Bit Decoder



4-Bit Demultiplexer



Demultiplexer Basics – 3



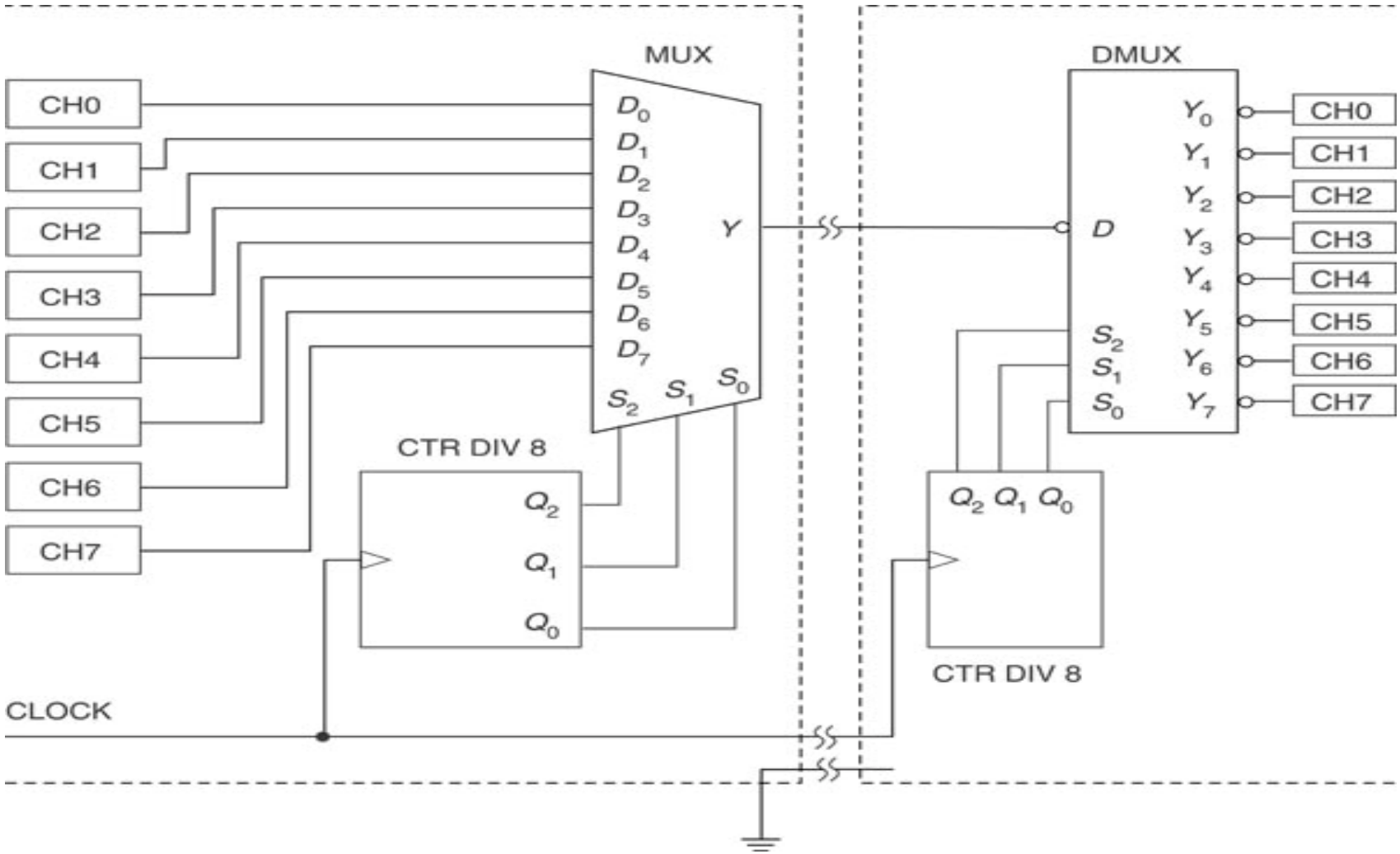
Demultiplexer Basics – 4

- Designated as a 1-to-*n* DEMUX that requires *m* select inputs such that *n* outputs = 2^m select inputs.
- 1-to-4 DEMUX Equations:

$$Y_{(0)} = D_0 \overline{S_1}\ \overline{S_0};\ Y_{(1)} = D_0 \overline{S_1}\ S_0;$$

$$Y_{(2)} = D_0\ S_1 \overline{S_0};\ Y_{(3)} = D_0\ S_1\ S_0.$$

Example 6.17 -1
Time-Division Multiplexing and Demultiplexing



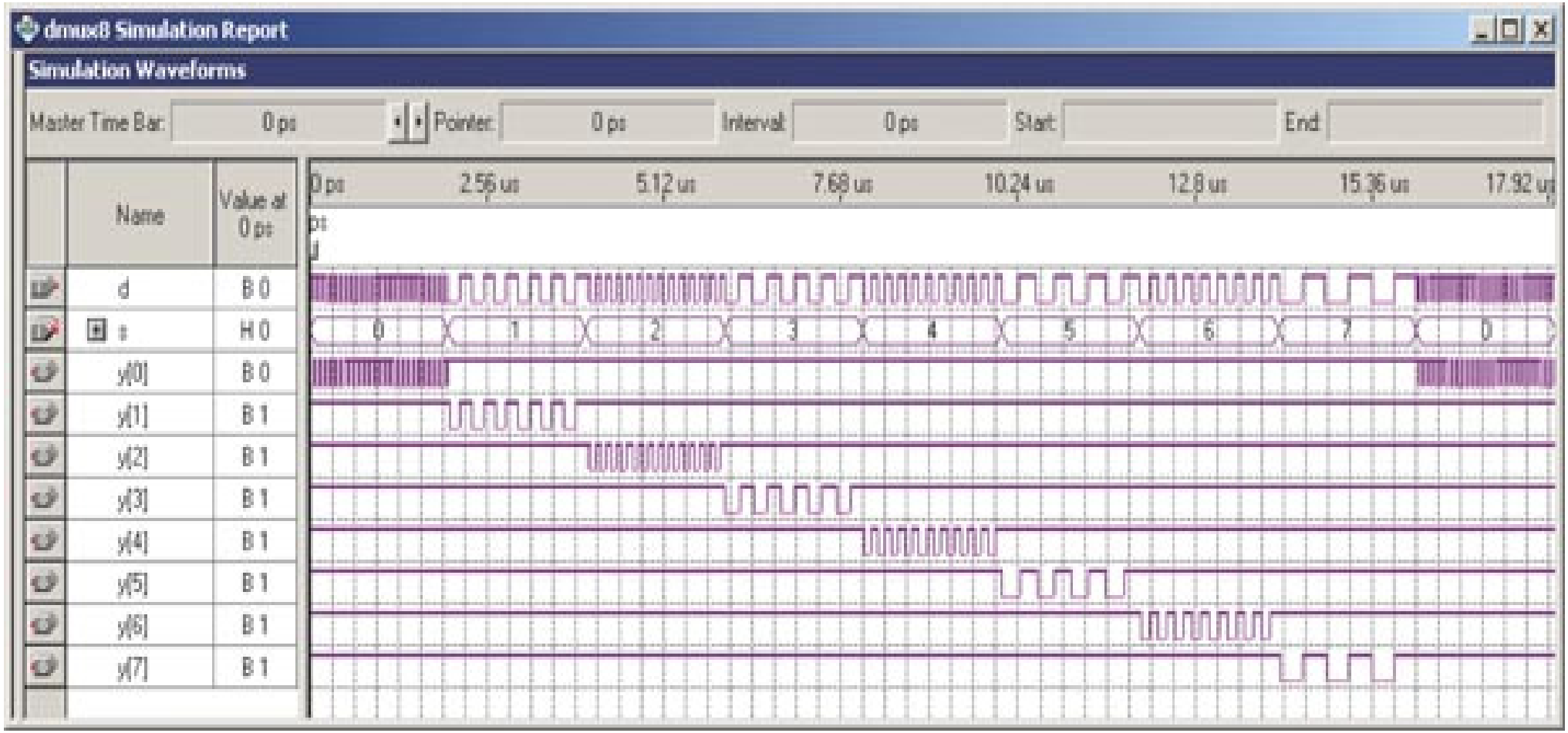
Example 6.17 -2
Demultiplexer VHDL Entity

```
ENTITY dmux8 IS
    PORT(
        s       : IN  STD_LOGIC_VECTOR (2  downto 0);
        d       : IN  STD_LOGIC;
        y       : OUT   STD_LOGIC_VECTOR (0 to 7));
END dmux8;
```

Example 6.17 -3
Demultiplexer VHDL Architecture

```
ARCHITECTURE  a  OF  dmux8  IS
        SIGNAL inputs  : STD_LOGIC_VECTOR (3 downto 0);
BEGIN
    inputs  <= d & s;
    WITH  inputs select
         Y  <=  "01111111"  WHEN  "0000",
                "10111111"  WHEN  "0001",
                      •          •          •
                      •          •          •
                "11111111"  WHEN  others;
END a;
```

Example 6.17 -4
Simulation Waveform

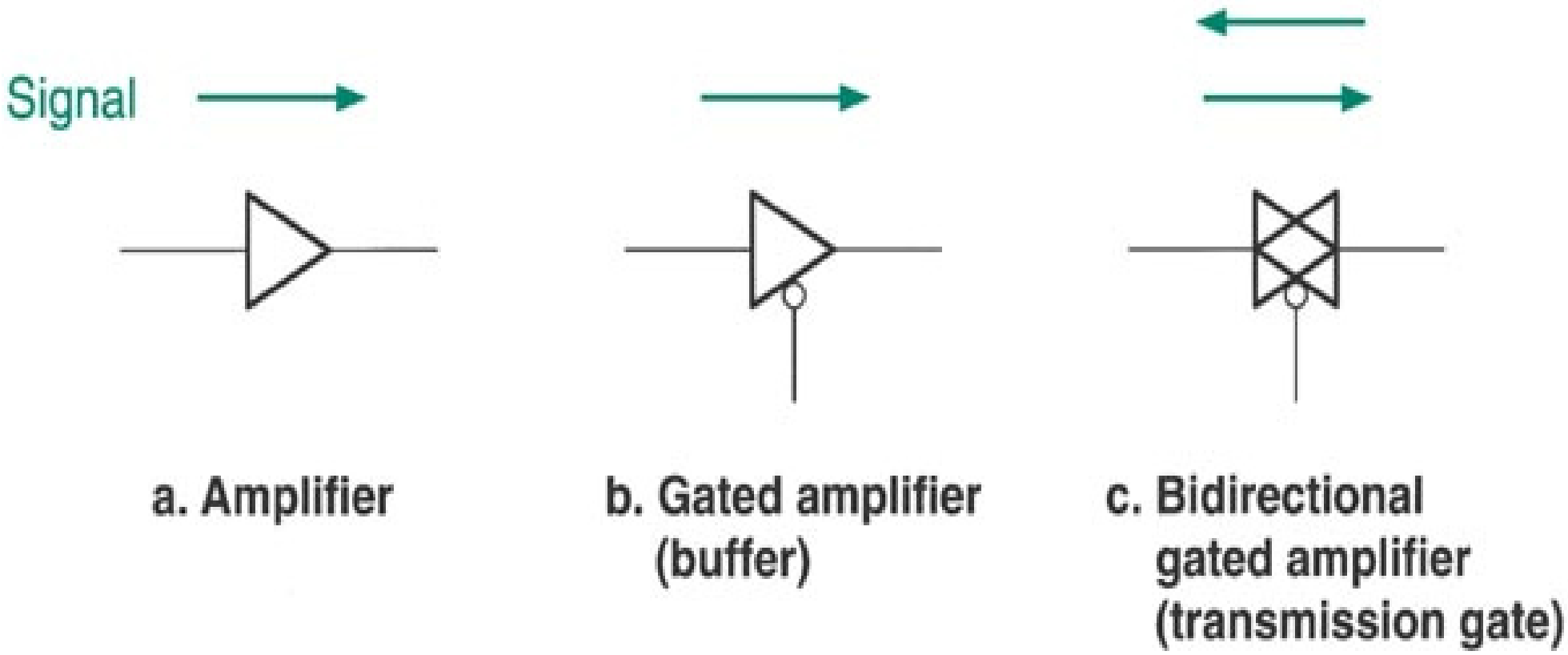


CMOS Analog MUX/DEMUX

- Uses a CMOS Switch or Transmission Gate that will allow a signal to pass in two directions.

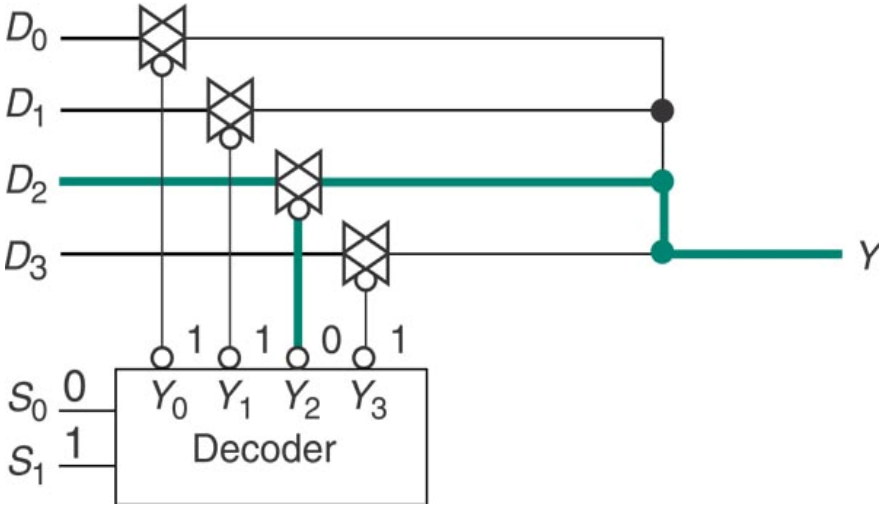
- Some commercial types such as a 4066B (standard CMOS) or 74HC4066 (high-speed CMOS).

Analog MUX/DEMUX



4-Channel CMOS MUX/DEMUX

MUX



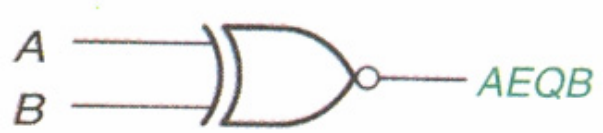
DMUX

6.5 Magnitude Comparators – 1

- Magnitude Comparator: A digital circuit that compares two *n*-Bit Binary Numbers and indicates if they are equal or which is greater.

- A very simple One-Bit Magnitude Comparator is the Two-Input XNOR Gate:
  - When both inputs are equal, the output is a 1; if they are not, it is a 0.

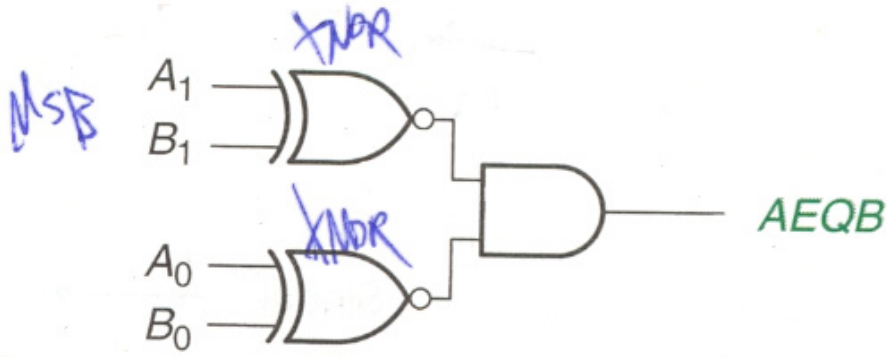
Magnitude Comparators – 2



Exclusive NOR Gate

XNOR Truth Table

A	B	AEQB
0	0	1
0	1	0
1	0	0
1	1	1



2-Bit Magnitude Comparator

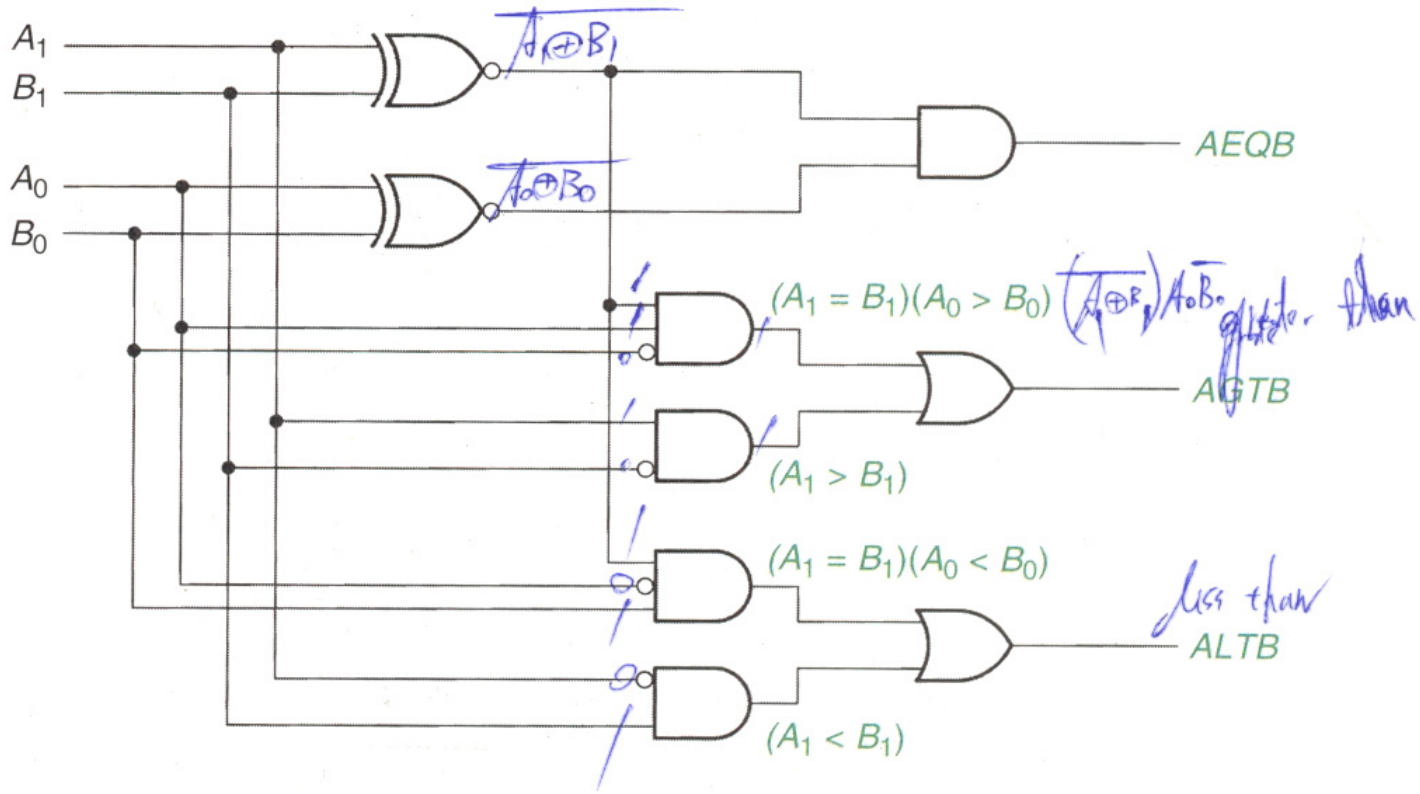
$A = B$, which is indicated by a HIGH at the AND output. This general principle applies to any number of bits:

$$AEQB = (\overline{A_{n-1} \oplus B_{n-1}}) \cdot (\overline{A_{n-1} \oplus B_{n-2}}) \ldots (\overline{A_1 \oplus B_1}) \cdot (\overline{A_0 \oplus B_0})$$

for two n-bit numbers, A and B.

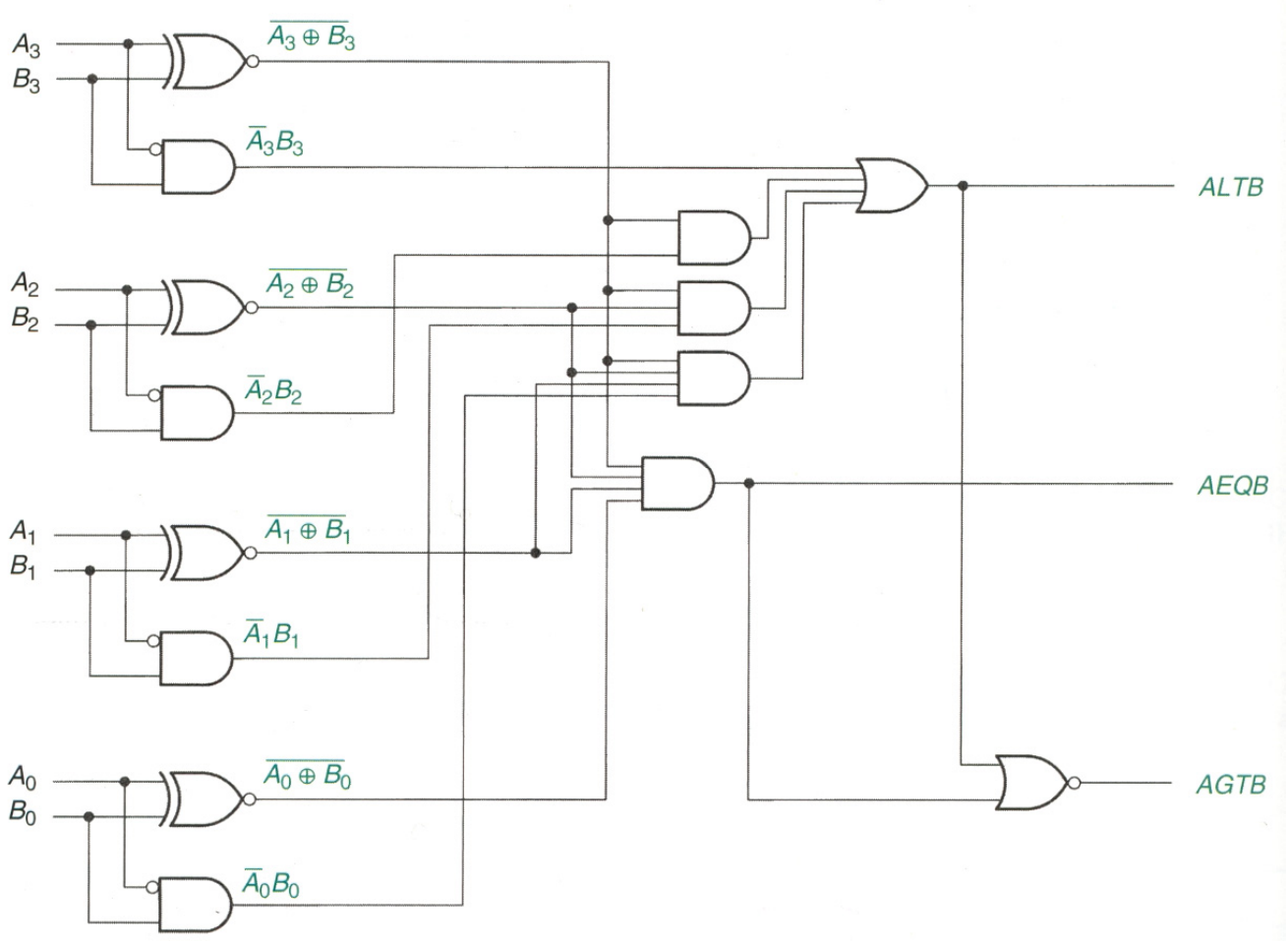
Magnitude Comparators – 3

2-Bit Comparator with *AEQB*, *AGTB*, and *ALTB* Outputs



Magnitude Comparators – 4

4-Bit Magnitude Comparator



$$AEQB = (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})(\overline{A_1 \oplus B_1})(\overline{A_0 \oplus B_0})$$

$$ALTB = \overline{A}_3 B_3 + (\overline{A_3 \oplus B_3})\overline{A}_2 B_2 + (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})\overline{A}_1 B_1 + (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})(\overline{A_1 \oplus B_1})\overline{A}_0 B_0$$

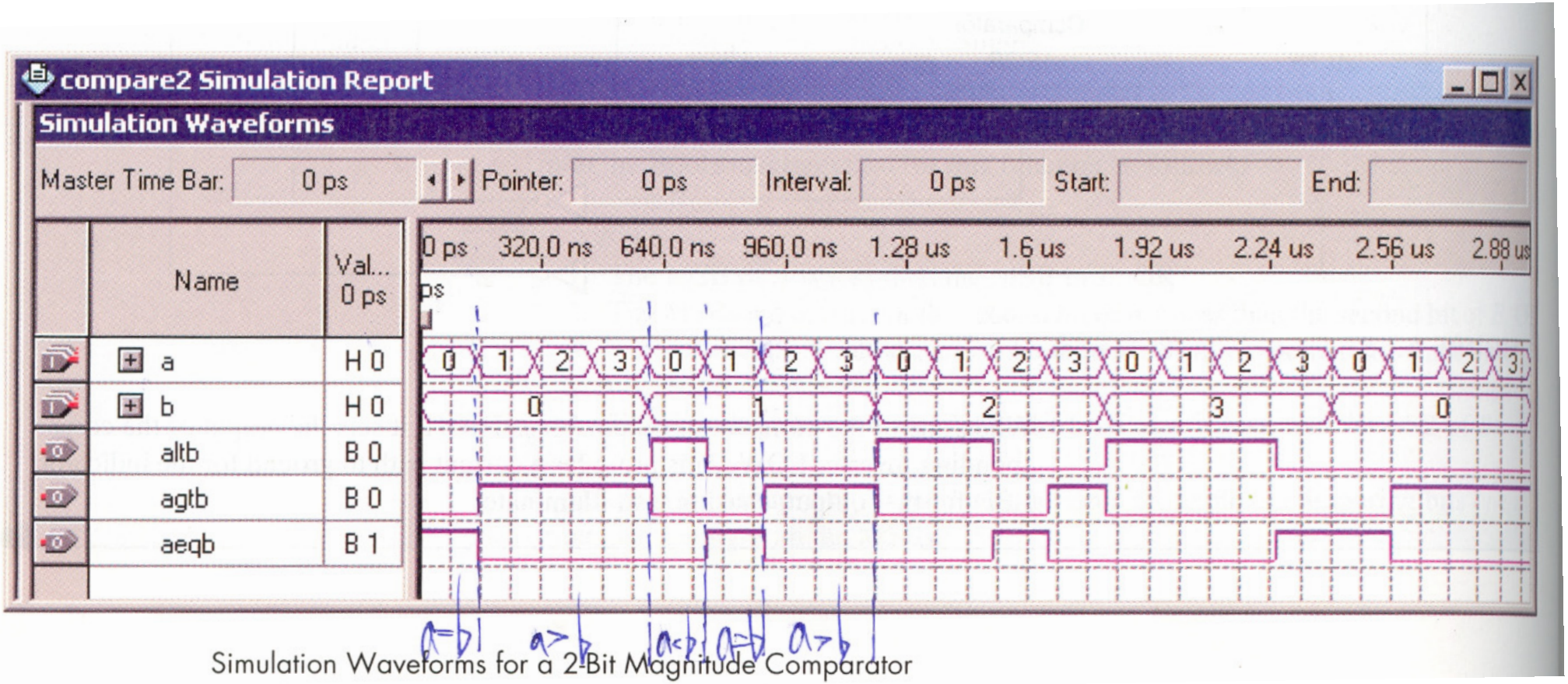
$$AGTB = \overline{AEQB + ALTB}$$

VHDL 2-Bit Magnitude Comparator – 1

```
-- compare2.vhd
ENTITY compare2 IS
 PORT(
     a, b                 : IN  BIT_VECTOR (1 downto 0);
     agtb, aeqb, altb     : OUT BIT);
END compare2;

ARCHITECTURE a OF compare2 IS
BEGIN
    altb <= (not (a(1)) and b(1))
          or ((not (a(1) xor b(1))) and (not (a(0)) and b(0)));
    aeqb <= (not (a(1) xor b(1))) and (not (a(0) xor b(0)));
    agtb <= (a(1) and not (b(1)))
          or ((not (a(1) xor b(1))) and (a(0) and not (b(0))));
END a;
```

VHDL 2-Bit Magnitude Comparator – 2



Simulation Waveforms for a 2-Bit Magnitude Comparator

6.6 Parity Basics – 1

- Parity: A digital system that checks for errors in a *n*-Bit Binary Number or Code.

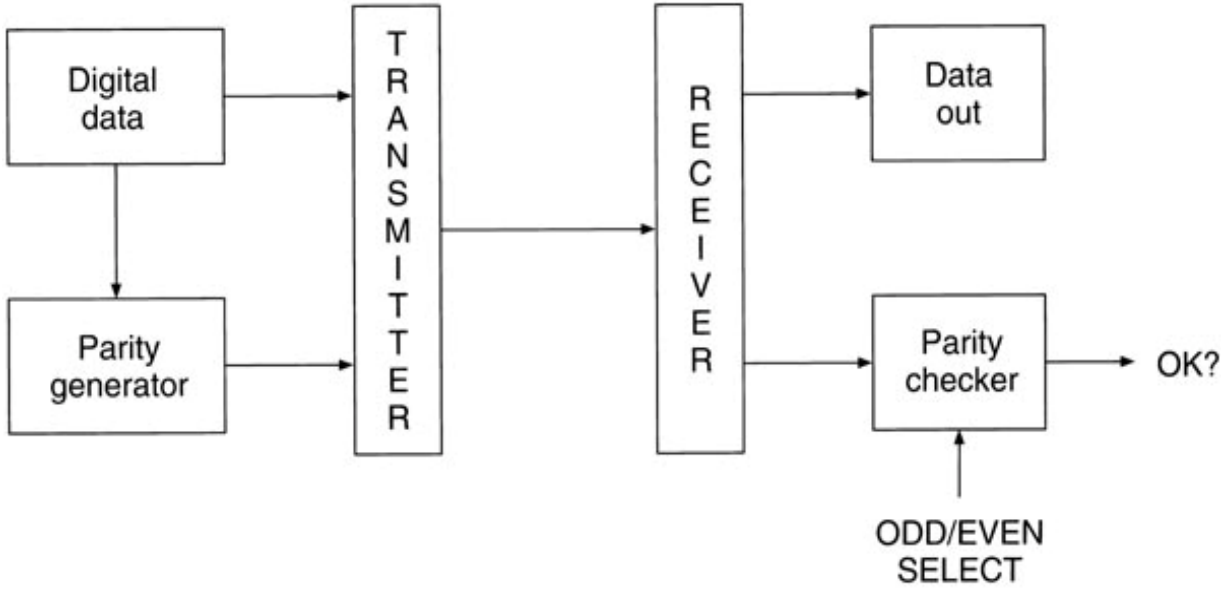
- Even Parity: A parity system that requires the binary number and the parity bit to have an even # of 1s.

- Odd Parity: A parity system that requires the binary number and the parity bit to have an Odd # of 1s.

Parity Basics – 2

- Parity Bit: A bit appended on the end of a binary number or code to make the # of 1s odd or even depending on the type of parity in the system.

- Parity is used in transmitting and receiving data by devices in a PC, that are on the COM Port.



Example 6.20

Data are transmitted from a PC serial port to a modem in groups of 7 data bits plus a parity bit. What should the parity bit, P, be for each of the following data if the parity is EVEN? If the parity is ODD?

a. 0110110
b. 1000000
c. 0010101

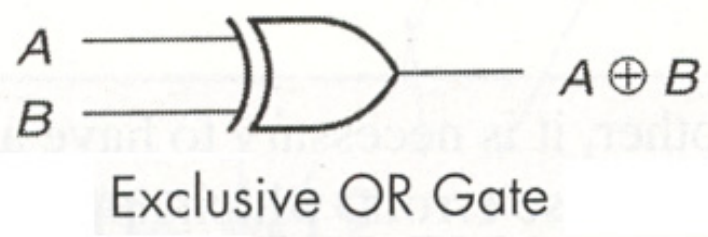
Solution

a. 0110110 Four 1s in data. (4 is an even number.)
EVEN parity: $P = 0$
ODD parity: $P = 1$

b. 1000000 One 1 in data. (1 is an odd number.)
EVEN parity: $P = 1$
ODD parity: $P = 0$

c. 0010101 Three 1s in data. (3 is an odd number.)
EVEN parity: $P = 1$
ODD parity: $P = 0$

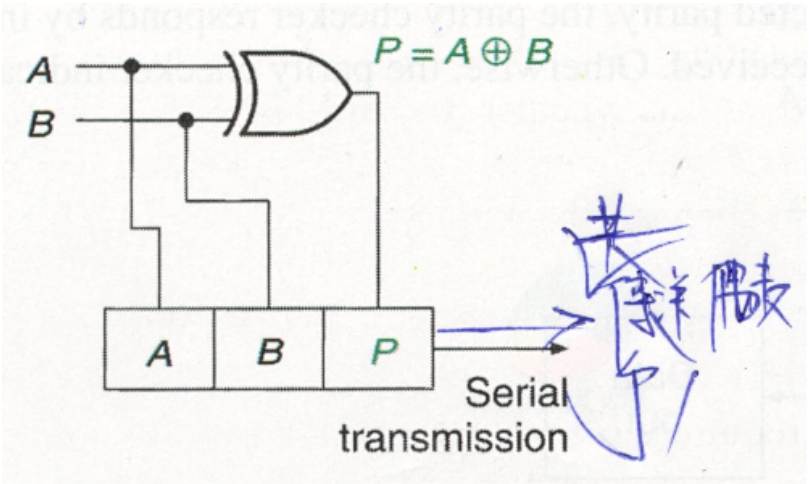
2-Bit Data: Parity Generation and Checking



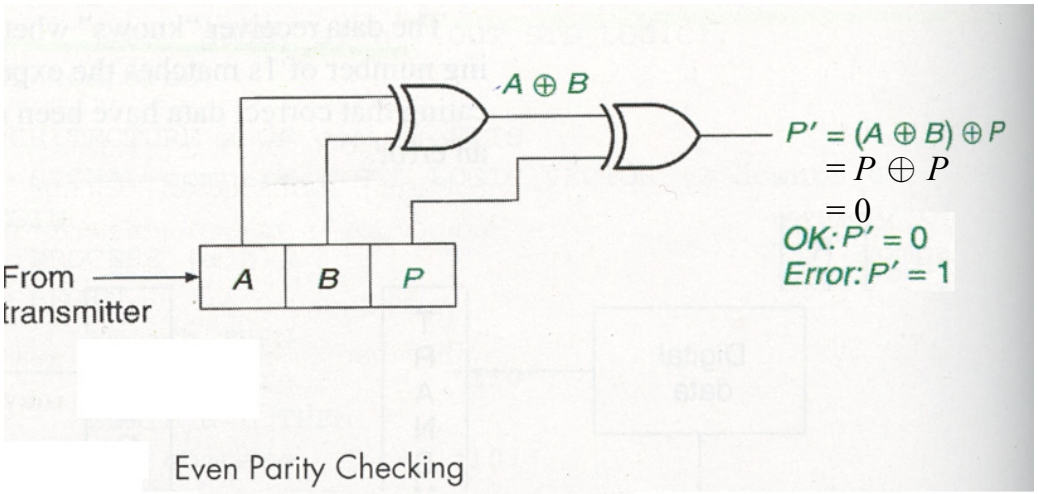
Exclusive OR Gate

Exclusive OR Truth Table

A	B	P=A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0



Even Parity Generation



Even Parity Checking

4-Bit Data: Parity Generation and Checking

Table 6.16

Even and Odd Parity Bits for 4-bit Data

A	B	C	D	$A \oplus B$	$C \oplus D$	P_E	P_O
0	0	0	0	0	0	0	1
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	0	0	0	1
0	1	0	0	1	0	1	0
0	1	0	1	1	1	0	1
0	1	1	0	1	1	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	0	1
1	0	1	1	1	0	1	0
1	1	0	0	0	0	0	1
1	1	0	1	0	1	1	0
1	1	1	0	0	1	1	0
1	1	1	1	0	0	0	1

$P_E = (A \oplus B) \oplus (C \oplus D)$

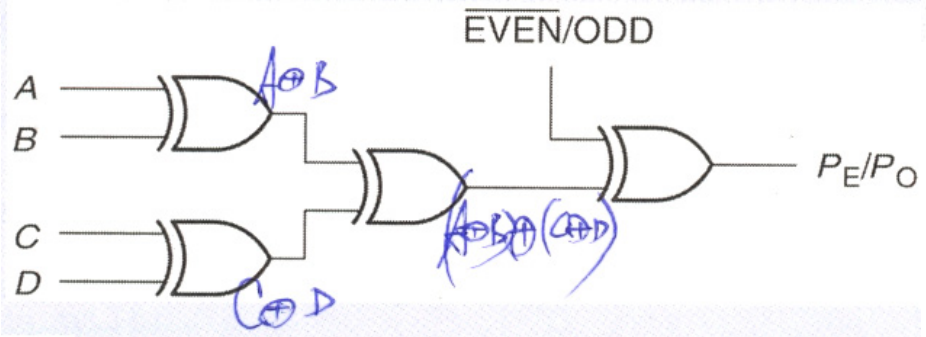
$P_O = \overline{P_E}$

$ABCDP_E$: 偶校字1

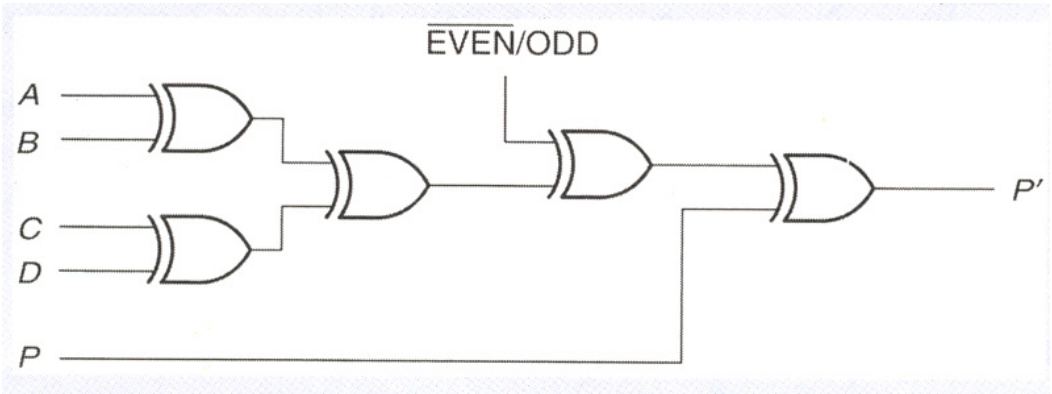
$ABCDP_O$: 奇校字1

Example 6.23

Use Table 6.16 to draw a 4-bit parity generator and a 4-bit parity checker that can generate and check either EVEN or ODD parity, depending on the state of one select input.

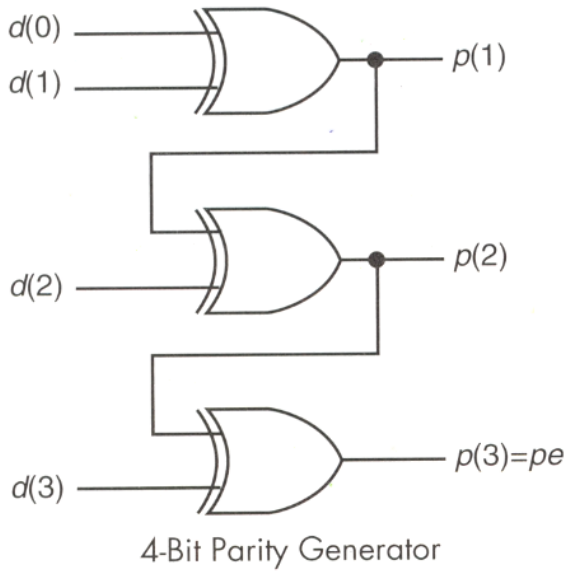


4-Bit Parity Generator



4-Bit Parity Checker

Parity Circuit in VHDL Using the Generate Statement



4-Bit Parity Generator

Figure 6.65

A simple way to make a small (e.g., 4-bit) parity generator in VHDL is to encode the Boolean expression as a concurrent signal assignment statement. For a 4-bit parity generator, we can define a set of inputs as **a, b, c, d** and write the equation as:

```
pe <= (a xor b) xor (c xor d);
```

This gets more complicated for a larger generator. For example, for an 8-bit circuit, we might write:

```
pe <= ((a xor b) xor (c xor d)) xor ((e xor f) xor (g xor h));
```

This is still manageable, but VHDL allows us to write statements that can easily be scaled to different sizes without rewriting large parts of the file. We can start building this kind of description by examining the 4-bit parity generator shown in Figure 6.65.

This circuit can be broken up into several pieces that can be described by the following concurrent signal assignment statements:

Book p300

```
p(1) <= d(1) xor d(0);
p(2) <= p(1) xor d(2);
p(3) <= p(2) xor d(3);
pe <= p(3);
```

VHDL GENERATE Statement

```
__generate_label:
 FOR __index_variable IN __range GENERATE
        __statement;
        __statement;
     END GENERATE;
```

4-Bit Parity Generator Using Generate Statement

```
-- 4-bit parity generator using GENERATE statement.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY parity4_gen IS
   PORT(
       d  : IN  STD_LOGIC_VECTOR(0 TO 3);
       pe : OUT STD_LOGIC);
END parity4_gen;

ARCHITECTURE parity OF parity4_gen IS
    SIGNAL p : STD_LOGIC_VECTOR(1 to 3);
BEGIN
   p(1) <= d(0) xor d(1);

   parity_generate:
   FOR i IN 2 to 3 GENERATE
       p(i) <= p(i-1) xor d(i);
   END GENERATE;

   pe <= p(3);
END parity;
```

HW