Applied Logic

Seven-Segment Display

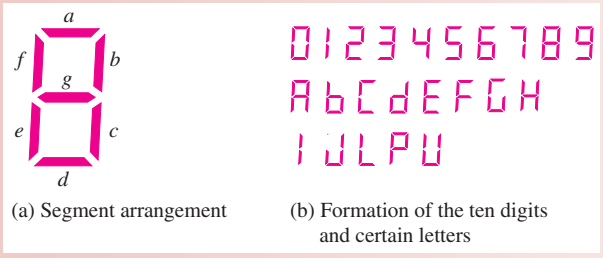
Seven-segment displays are used in many types of products that you see every day. A 7-segment display was used in the tablet-bottling system that was introduced in Chapter 1. The display in the bottling system is driven by logic circuits that decode a binary coded decimal (BCD) number and activate the appropriate digits on the display. BCD-to-7-segment decoder/drivers are readily available as single IC packages for activating the ten decimal digits.

In addition to the numbers from 0 to 9, the 7-segment display can show certain letters. For the tablet-bottling system, a requirement has been added to display the letters A, b, C, d, and E on a separate common-anode 7-segment display that uses a hexadecimal keypad for both the numerical inputs and the letters. These letters will be used to identify the type of vitamin tablet that is being bottled at any given time. In this application, the decoding logic for displaying the five letters is developed.

The 7-Segment Display

Two types of 7-segment displays are the LED and the LCD. Each of the seven segments in an LED display uses a light-emitting diode to produce a colored light when there is current through it and can be seen in the dark. An LCD or liquid-crystal display operates by polarizing light so that when a segment is not activated by a voltage, it reflects incident light and appears invisible against its background; however, when a segment is activated, it does not reflect light and appears black. LCD displays cannot be seen in the dark.

The seven segments in both LED and LCD displays are arranged as shown in Figure 4–50 and labeled *a*, *b*, *c*, *d*, *e*, *f*, and *g* as indicated in part (a). Selected segments are activated to create each of the ten decimal digits as well as certain letters of the alphabet, as shown in part (b). The letter *b* is shown as lowercase because a capital B would be the same as the digit 8. Similarly, for *d*, a capital letter would appear as a 0.



(a) Segment arrangement

(b) Formation of the ten digits and certain letters

FIGURE 4–50 Seven-segment display.

Exercise

1. List the segments used to form the digit 2.
2. List the segments used to form the digit 5.
3. List the segments used to form the letter A.
4. List the segments used to form the letter E.
5. Is there any one segment that is common to all digits?
6. Is there any one segment that is common to all letters?

Display Logic

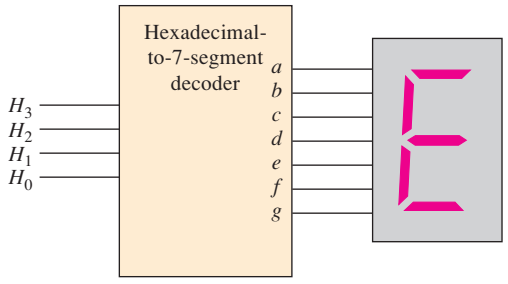
The segments in a 7-segment display can be used in the formation of various letters as shown in Figure 4–50(b). Each segment must be activated by its own decoding circuit that detects the code for any of the letters in which that segment is used. Because a common-anode display is used, the segments are turned *on* with a LOW (0) logic level and turned *off* with a HIGH (1) logic level. The active segments are shown for each of the letters required for the tablet-bottling system in Table 4–14. Even though the active level is LOW (lighting the LED), the logic expressions are developed exactly the same way as discussed in this chapter, by mapping the desired output (1, 0, or X) for every possible input, grouping the 1s on the map, and reading the SOP expression from the map. In effect, the reduced logic expression is the logic for keeping a given segment OFF. At first, this may sound confusing, but it is simple in practice and it avoids an output current capability issue with bipolar (TTL) logic (discussed in Chapter 15 on the website).

TABLE 4–14

Active segments for each of the five letters used in the system display.

Letter	Segments Activated
A	*a*, *b*, *c*, *e*, *f*, *g*
b	*c*, *d*, *e*, *f*, *g*
C	*a*, *d*, *e*, *f*
d	*b*, *c*, *d*, *e*, *g*
E	*a*, *d*, *e*, *f*, *g*

A block diagram of a 7-segment logic and display for generating the five letters is shown in Figure 4–51(a), and the truth table is shown in part (b). The logic has four hexadecimal inputs and seven outputs, one for each segment. Because the letter F is not used as an input, we will show it on the truth table with all outputs set to 1 (OFF).



(a)

Letter	Hexadecimal Inputs H_3	H_2	H_1	H_0	Segment Ouputs *a*	*b*	*c*	*d*	*e*	*f*	*g*
A	1	0	1	0	0	0	0	1	0	0	0
b	1	0	1	1	1	1	0	0	0	0	0
C	1	1	0	0	0	1	1	0	0	0	1
d	1	1	0	1	1	0	0	0	0	1	0
E	1	1	1	0	0	1	1	0	0	0	0
F	1	1	1	1	1	1	1	1	1	1	1

(b)

FIGURE 4–51 Hexadecimal-to-7-segment decoder for letters *A* through *E*, used in the system.

Karnaugh Maps and the Invalid BCD Code Detector

To develop the simplified logic for each segment, the truth table information in Figure 4–51 is mapped onto Karnaugh maps. Recall that the BCD numbers will not be shown on the letter display. For this reason, an entry that represents a BCD number will be entered as an "X" ("don't care") on the K-maps. This makes the logic much simpler but would put some strange outputs on the display unless steps are taken to eliminate that possibility. Because all of the letters are *invalid* BCD characters, the display is activated only when an invalid BCD code is entered into the keypad, thus allowing only letters to be displayed.

Expressions for the Segment Logic

Using the table in 4–51(b), a standard SOP expression can be written for each segment and then minimized using a K-map. The desired outputs from the truth table are entered in the appropriate cells representing the hex inputs. To obtain the minimum SOP expressions for the display logic, the 1s and Xs are grouped.

Segment a Segment *a* is used for the letters A, C, and E. For the letter A, the hexadecimal code is 1010 or, in terms of variables, $H_3\overline{H}_2H_1\overline{H}_0$. For the letter C, the hexadecimal code is 1100 or $H_3H_2\overline{H}_1\overline{H}_0$. For the letter E, the code is 1110 or $H_3H_2H_1\overline{H}_0$. The complete standard SOP expression for segment *a* is

$$a = H_3\overline{H}_2H_1\overline{H}_0 + H_3H_2\overline{H}_1\overline{H}_0 + H_3H_2H_1\overline{H}_0$$

Because a LOW is the active output state for each segment logic circuit, a 0 is entered on the Karnaugh map in each cell that represents the code for the letters in which the segment is *on*. The simplification of the expression for segment *a* is shown in Figure 4–52(a) after grouping the 1s and Xs.

Segment b Segment *b* is used for the letters A and d. The complete standard SOP expression for segment *b* is

$$b = H_3\overline{H}_2H_1\overline{H}_0 + H_3H_2\overline{H}_1H_0$$

The simplification of the expression for segment *b* is shown in Figure 4–52(b).

Segment c Segment *c* is used for the letters A, b, and d. The complete standard SOP expression for segment *c* is

$$c = H_3\overline{H}_2H_1\overline{H}_0 + H_3\overline{H}_2H_1H_0 + H_3H_2\overline{H}_1H_0$$

The simplification of the expression for segment *c* is shown in Figure 4–52(c).

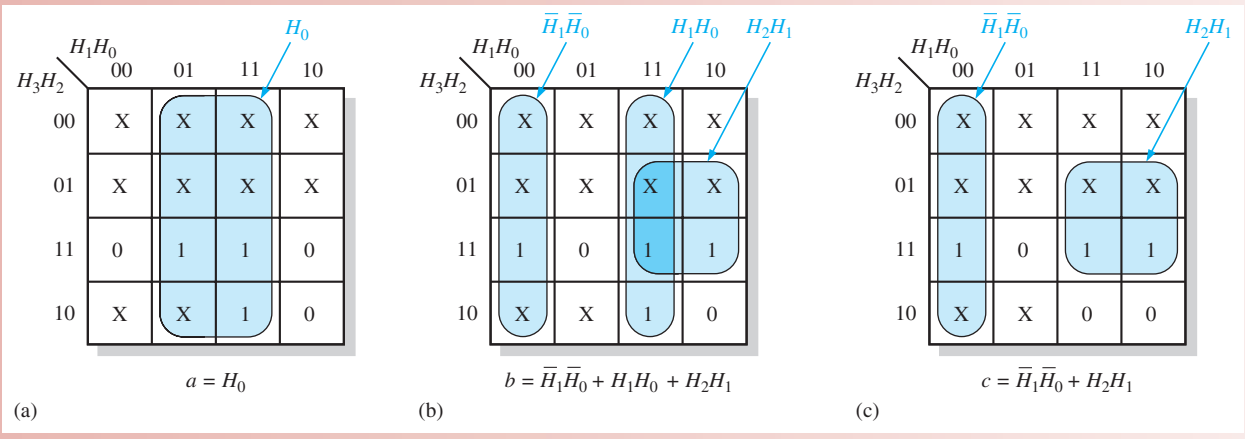


FIGURE 4–52 Minimization of the expressions for segments *a*, *b*, and *c*.

Exercise

7. Develop the minimum expression for segment *d*.
8. Develop the minimum expression for segment *e*.
9. Develop the minimum expression for segment *f*.
10. Develop the minimum expression for segment *g*.

The Logic Circuits

From the minimum expressions, the logic circuits for each segment can be implemented. For segment *a*, connect the H_0 input directly (no gate) to the *a* segment on the display. The segment *b* and segment *c* logic are shown in Figure 4–53 using AND or OR gates. Notice that two of the terms (H_2H_1 and $\overline{H}_1\overline{H}_0$) appear in the expressions for both *b* and *c* logic so two of the AND gates can be used in both, as indicated.

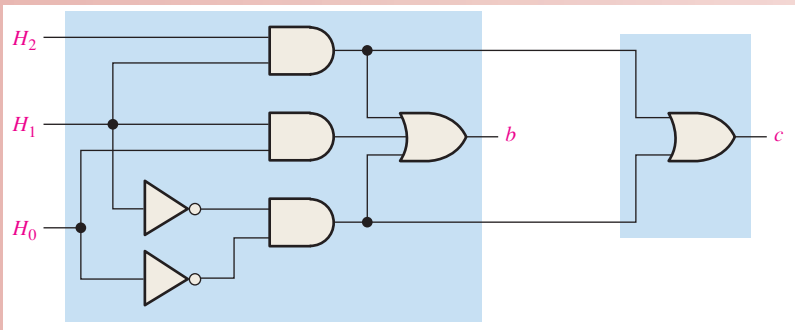


FIGURE 4–53 Segment-*b* and segment-*c* logic circuits.

Exercise

11. Show the logic for segment *d*.
12. Show the logic for segment *e*.
13. Show the logic for segment *f*.
14. Show the logic for segment *g*.

Describing the Decoding Logic with VHDL

The 7-segment decoding logic can be described using VHDL for implementation in a programmable logic device (PLD). The logic expressions for segments *a*, *b*, and *c* of the display are as follows:

$$a = H_0$$
$$b = \overline{H}_1\overline{H}_0 + H_1H_0 + H_2H_1$$
$$c = \overline{H}_1\overline{H}_0 + H_2H_1$$

- The VHDL code for segment *a* is

```
entity SEGLOGIC is
   port (H0: in bit; SEGa: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGa <= H0;
end architecture LogicFunction;
```

- The VHDL code for segment *b* is

```
entity SEGLOGIC is
   port (H0, H1, H2: in bit; SEGb: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGb <= (not H1 and not H0) or (H1 and H0) or (H2 and H1);
end architecture LogicFunction;
```

- The VHDL code for segment *c* is

```
entity SEGLOGIC is
   port (H0, H1, H2: in bit; SEGc: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGc <= (not H1 and not H0) or (H2 and H1);
end architecture LogicFunction;
```

Exercise

15. Write the VHDL code for segments *d*, *e*, *f*, and *g*.

Simulation

The decoder simulation using Multisim is shown in Figure 4–54 with the letter E selected. Subcircuits are used for the segment logic to be developed as activities or in the lab. The purpose of simulation is to verify proper operation of the circuit.

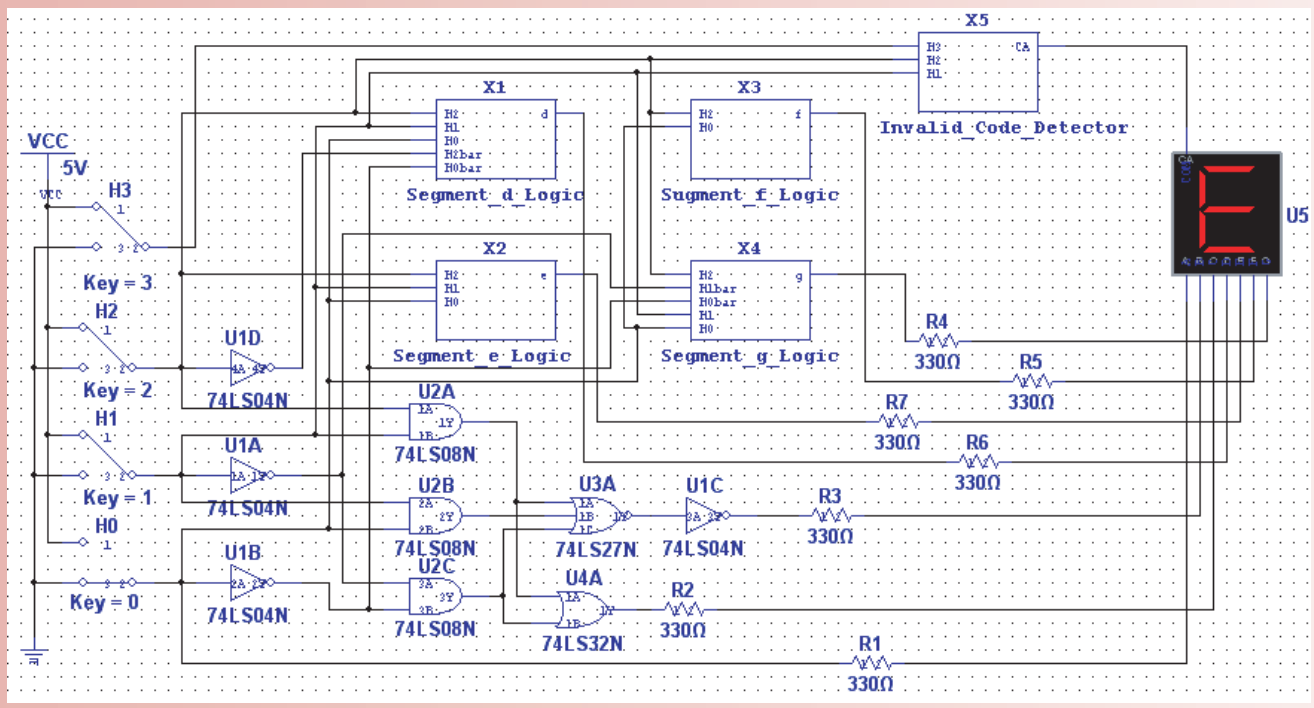


FIGURE 4–54 Multisim circuit screen for decoder and display.

MultiSim

Open file AL04 in the Applied Logic folder on the website. Run the simulation of the decoder and display using your Multisim software. Observe the operation for the specified letters.

Putting Your Knowledge to Work

How would you modify the decoder for a common-cathode 7-segment display?

SUMMARY

- Gate symbols and Boolean expressions for the outputs of an inverter and 2-input gates are shown in Figure 4–55.

FIGURE 4–55

As you know, testing and troubleshooting logic circuits often require observing and comparing two digital waveforms simultaneously, such as an input and the output of a device, on an oscilloscope. For digital waveforms, the scope should always be set to DC coupling on each channel input to avoid "shifting" the ground level. You should determine where the 0 V level is on the screen for both channels.

To compare the timing of the waveforms, the scope should be triggered from only one channel (don't use vertical mode or composite triggering). The channel selected for triggering should always be the one that has the lowest frequency waveform, if possible.

SECTION 5–7 CHECKUP

1. List four common internal failures in logic gates.
2. One input of a NOR gate is externally shorted to $+V_{CC}$. How does this condition affect the gate operation?
3. Determine the output of gate G_4 in Figure 5–49(a), with inputs as shown in part (b), for the following faults:
   - **(a)** one input to G_1 shorted to ground
   - **(b)** the inverter input shorted to ground
   - **(c)** an open output in G_3

Applied Logic

Tank Control

A storage tank system for a pancake syrup manufacturing company is shown in Figure 5–50. The control logic allows a volume of corn syrup to be preheated to a specified temperature to achieve the proper viscosity prior to being sent to a mixing vat where ingredients such as sugar, flavoring, preservative, and coloring are added. Level and temperature sensors in the tank and the flow sensor provide the inputs for the logic.

System Operation and Analysis

The tank holds corn syrup for use in a pancake syrup manufacturing process. In preparation for mixing, the temperature of the corn syrup when released from the tank into a mixing vat must be at a specified value for proper viscosity to produce required flow characteristics. This temperature can be selected via a keypad input. The control logic maintains the temperature at this value by turning a heater *on* and *off*. The analog output from the temperature transducer (T_{analog}) is converted to an 8-bit binary code by an analog-to-digital converter and then to an 8-bit BCD code. A temperature controller detects when the temperature falls below the specified value and turns the heater *on*. When the temperature reaches the specified value, the heater is turned *off*.

The level sensors produce a HIGH when the corn syrup is at or above the minimum or at the maximum level. The valve control logic detects when the maximum level (L_{max}) or minimum level (L_{min}) has been reached and when mixture is flowing into the tank (F_{inlet}). Based on these inputs, the control logic opens or closes each valve (V_{inlet} and V_{outlet}). New corn syrup can be

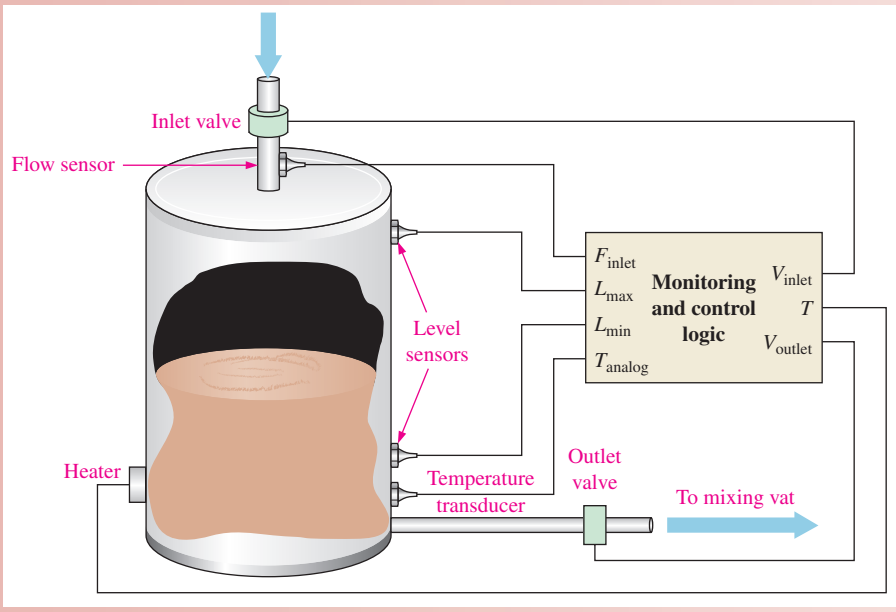


FIGURE 5–50 Tank with level and temperature sensors and controls.

added to the tank via the inlet valve only when the minimum level is reached. Once the inlet valve is opened, the level in the tank must reach the maximum point before the inlet valve is closed. Also, once the outlet valve is opened, the level must reach the minimum point before the outlet valve is closed. New syrup is always cooler than the syrup in the tank. Syrup cannot be released from the tank while it is being filled or its temperature is below the specified value.

Inlet Valve Control The conditions for which the inlet valve is open, allowing the tank to fill, are

- The solution level is at minimum (L_{min}).
- The tank is filling (F_{inlet}) but the maximum level has not been reached ($\overline{L}_{max}$).

Table 5–6 is the truth table for the inlet valve. A HIGH (1) is the active level for the inlet valve to be open (*on*).

TABLE 5–6

Truth table for inlet valve control.

Inputs			Output	
L_{max}	L_{min}	F_{inlet}	V_{inlet}	Description
0	0	0	1	Level below minimum. No inlet flow.
0	0	1	1	Level below minimum. Inlet flow.
0	1	0	0	Level above min and below max. No inlet flow.
0	1	1	1	Level above min and below max. Inlet flow.
1	0	0	X	Invalid
1	0	1	X	Invalid
1	1	0	0	Level at maximum. No inlet flow.
1	1	1	0	Level at maximum. Inlet flow.

Exercise

1. Explain why the two conditions indicated in the truth table are invalid.
2. Under how many input conditions is the inlet valve open?
3. Once the level drops below minimum and the tank starts refilling, when does the inlet valve turn *off*?

From the truth table, an expression for the inlet valve control output can be written.

$$V_{\text{inlet}} = \overline{L}_{\text{max}}\overline{L}_{\text{min}}\overline{F}_{\text{inlet}} + \overline{L}_{\text{max}}\overline{L}_{\text{min}}F_{\text{inlet}} + \overline{L}_{\text{max}}L_{\text{min}}F_{\text{inlet}}$$

The SOP expression for the inlet valve logic can be reduced to the following simplified expression using Boolean methods:

$$V_{\text{inlet}} = \overline{L}_{\text{min}} + \overline{L}_{\text{max}}F_{\text{inlet}}$$

Exercise

4. Using a K-map, prove that the simplified expression is correct.
5. Using the simplified expression, draw the logic diagram for the inlet valve control.

Outlet Valve Control The conditions for which the outlet valve is open allowing the tank to drain are

- The syrup level is above minimum and the tank is not filling.
- The temperature of the syrup is at the specified value.

Table 5–7 is the truth table for the outlet valve. A HIGH (1) is the active level for the outlet valve to be open (*on*). (*Note: T* is both an input and an output, T = Temp).

TABLE 5–7

Truth table for outlet valve control.

Inputs				Output	
L_{max}	L_{min}	F_{inlet}	T	V_{outlet}	Description
0	0	0	0	0	Level below minimum. No inlet flow. Temp low.
0	0	0	1	0	Level below minimum. No inlet flow. Temp correct.
0	0	1	0	0	Level below minimum. Inlet flow. Temp low.
0	0	1	1	0	Level below minimum. Inlet flow. Temp correct.
0	1	0	0	0	Level above min and below max. No inlet flow. Temp low.
0	1	0	1	1	Level above min and below max. No inlet flow. Temp correct.
0	1	1	0	0	Level above min and below max. Inlet flow. Temp low.
0	1	1	1	0	Level above min and below max. Inlet flow. Temp correct
1	0	0	0	X	Invalid
1	0	0	1	X	Invalid
1	0	1	0	X	Invalid
1	0	1	1	X	Invalid
1	1	0	0	0	Level at maximum. No inlet flow. Temp low.
1	1	0	1	1	Level at maximum. No inlet flow. Temp correct.
1	1	1	0	0	Level at maximum. Inlet flow. Temp low.
1	1	1	1	0	Level at maximum. Inlet flow. Temp correct.

Exercise

6. Why does the outlet valve control require four inputs and the inlet valve only three?
7. Under how many input conditions is the outlet valve open?
8. Once the level reaches maximum and the tank starts draining, when does the outlet valve turn off?

From the truth table, an expression for the outlet valve control can be written.

$$V_{\text{outlet}} = \overline{L}_{\text{max}}L_{\text{min}}\overline{F}_{\text{inlet}}\,T + L_{\text{max}}L_{\text{min}}\overline{F}_{\text{inlet}}T$$

The SOP expression for the outlet valve logic can be reduced to the following simplified expression:

$$V_{outlet} = L_{min}\overline{F}_{inlet}T$$

Exercise

9. Using a K-map, prove that the simplified expression is correct.

10. Using the simplified expression, draw the logic diagram for the outlet valve control.

Temperature Control The temperature control logic accepts an 8-bit BCD code representing the measured temperature and compares it to the BCD code for the specified temperature. A block diagram is shown in Figure 5–51.

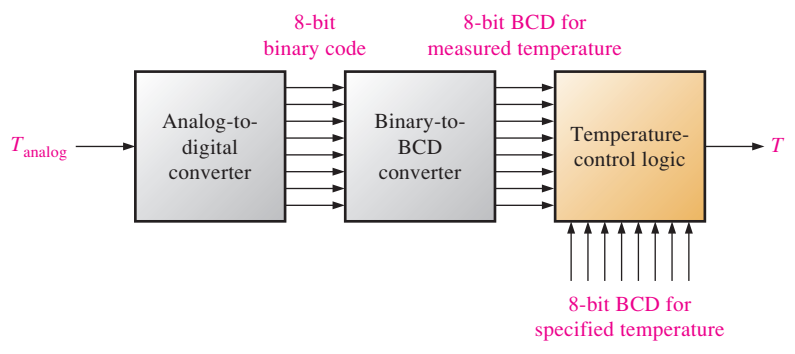


FIGURE 5–51 Block diagram for temperature control circuit.

When the measured temperature and the specified temperature are the same, the two BCD codes are equal and the T output is LOW (0). When the measured temperature falls below the specified value, there is a difference in the BCD codes and the T output is HIGH (1), which turns on the heater. The temperature control logic can be implemented with exclusive-OR gates, as shown in Figure 5–52. Each pair of corresponding bits from the two

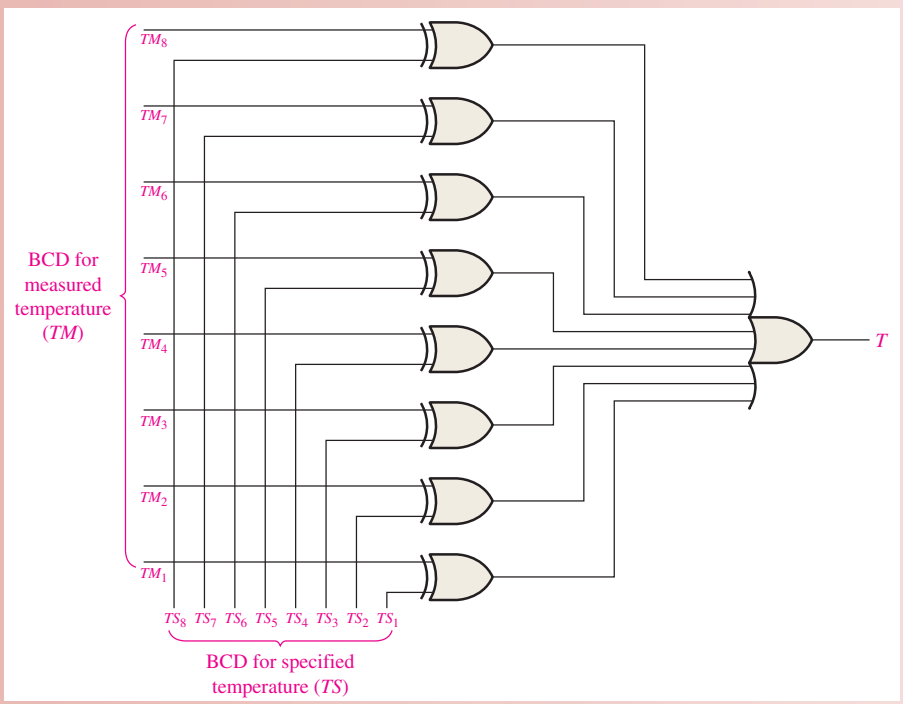


FIGURE 5–52 Logic diagram of the temperature control logic.

BCD codes is applied to an exclusive-OR gate. If the bits are the same, the output of the XOR gate is 0; and if they are different, the output of the XOR gate is 1. When one or more XOR outputs equal 1, the *T* output of the OR gate equals 1, causing the heater to turn on.

VHDL Code for Tank Control Logic

The control logic for the inlet valve, outlet valve, and temperature is described with VHDL using the data flow approach (which is based on the Boolean description of the logic). Exercise 11 requires the structural approach (which is based on the gates and how they are connected) for comparison.

```
entity TankControl is
  port (Finlet, Lmax, Lmin, TS1, TS2, TS3, TS4, TS5, TS6, TS7, TS8, TM1, TM2,
  TM3, TM4, TM5, TM6, TM7, TM8: in bit; Vinlet, Voutlet, T: out bit);
end entity TankControl;

architecture ValveTempLogic of Tank Control is

begin

  Vinlet <= not Lmin or (not Lmax and Finlet);
  Voutlet <= Lmin and not Finlet and T;
  T <= (TS1 xor TM1) or (TS2 xor TM2) or (TS3 xor TM3) or (TS4 xor TM4)
  or (TS5 xor TM5) or (TS6 xor TM6) or (TS7 xor TM7) or (TS8 xor TM8);

end architecture ValveTempLogic;
```

Exercise

11. Write the VHDL code for the tank control logic using the structural approach.

Simulation of the Valve Control Logic

The inlet and outlet valve control logic simulation screen is shown in Figure 5–53. SPDT switches are used to represent the level and flow sensor inputs and the temperature indication. Probes are used to indicate the output states.

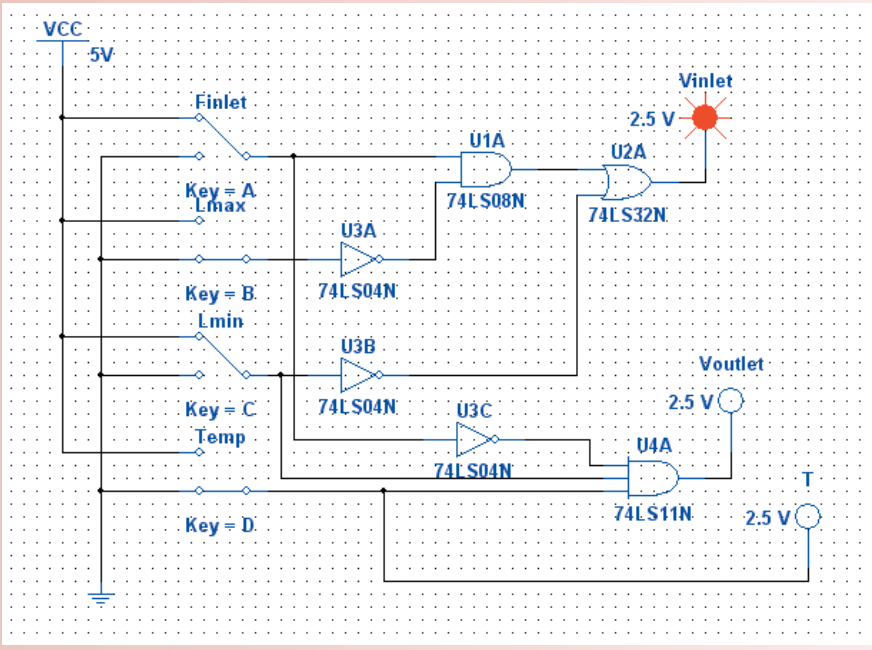


FIGURE 5–53 Multisim circuit screen for the valve control logic.

Open file AL05 in the Applied Logic folder on the website. Run the simulation of the valve-control logic using your Multisim software and observe the operation. Create a new Multisim file, connect the temperature control logic, and run the simulation.

MultiSim

Putting Your Knowledge to Work

If the temperature of the syrup can never be more than 9°C below the specified value, can the temperature control circuit be simplified? If so, how?

SUMMARY

- AND-OR logic produces an output expression in SOP form.
- AND-OR-Invert logic produces a complemented SOP form, which is actually a POS form.
- The operational symbol for exclusive-OR is $\oplus$. An exclusive-OR expression can be stated in two equivalent ways:

$$A\overline{B} + \overline{A}B = A \oplus B$$

- To do an analysis of a logic circuit, start with the logic circuit, and develop the Boolean output expression or the truth table or both.
- Implementation of a logic circuit is the process in which you start with the Boolean output expressions or the truth table and develop a logic circuit that produces the output function.
- All NAND or NOR logic diagrams should be drawn using appropriate dual symbols so that bubble outputs are connected to bubble inputs and nonbubble outputs are connected to nonbubble inputs.
- When two negation indicators (bubbles) are connected, they effectively cancel each other.
- A VHDL component is a predefined logic function stored for use throughout a program or in other programs.
- A component instantiation is used to call for a component in a program.
- A VHDL signal effectively acts as an internal interconnection in a VHDL structural description.

KEY TERMS

Key terms and other bold terms in the chapter are defined in the end-of-book glossary.

Component A VHDL feature that can be used to predefine a logic function for multiple use throughout a program or programs.

Negative-AND The dual operation of a NOR gate when the inputs are active-LOW.

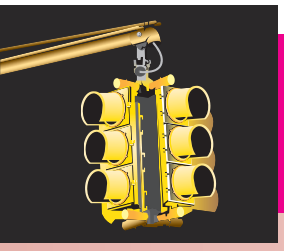
Negative-OR The dual operation of a NAND gate when the inputs are active-LOW.

Node A common connection point in a circuit in which a gate output is connected to one or more gate inputs.

Signal A waveform; a type of VHDL object that holds data.

Signal tracing A troubleshooting technique in which waveforms are observed in a step-by-step manner beginning at the input and working toward the output or vice versa. At each point the observed waveform is compared with the correct signal for that point.

Universal gate Either a NAND gate or a NOR gate. The term *universal* refers to the property of a gate that permits any logic function to be implemented by that gate or by a combination of that kind.



Applied Logic

Traffic Signal Controller: Part 1

The control logic is developed for a traffic signal at the intersection of a busy main street and a lightly used side street. The system requirements are established, and a general block diagram is developed. Also, a state diagram is introduced to define the sequence of operation. The combinational logic unit of the controller is developed in this chapter, and the remaining units are developed in Chapter 7.

Timing Requirements

The control logic establishes the sequencing of the lights for a traffic signal at the intersection of a busy main street and an occasionally used side street. The following are the timing requirements:

- The green light for the main street will stay on for a minimum of 25 s or as long as there is no vehicle on the side street.
- The green light for the side street will stay on until there is no vehicle on the side street up to a maximum of 25 s.
- The yellow caution light will stay on for 4 s between changes from green to red on both the main street and the side street.

The State Diagram

From the timing requirements, a state diagram can be developed to describe the complete operation. A state diagram graphically shows the sequence of states, the conditions for each state, and the requirements for transitions from one state to the next.

Defining the Variables The variables that determine how the system sequences through the various states are defined as follows:

- V_s A vehicle is present on the side street.
- T_L The 25 s timer (long timer) is *on.*
- T_S The 4 s timer (short timer) is *on.*

A complemented variable indicates the opposite condition.

State Descriptions A state diagram is shown in Figure 6–63. Each of the four states is assigned a 2-bit Gray code as indicated. A looping arrow means that the system remains in a state, and an arrow between states means that the system transitions to the next state. The Boolean expression or variable associated with each of the arrows in the state diagram indicate the condition under which the system remains in a state or transitions to the next state.

First State The Gray code is 00. In this state, the light is green on the main street and red on the side street for 25 s when the long timer is *on* or there is no vehicle on the side street. This condition is expressed as $T_L + \overline{V}_s$. The system transitions to the next state when the long timer goes *off* and there is a vehicle on the side street. This condition is expressed as $\overline{T}_L V_s$.

Second State The Gray code is 01. In this state, the light is yellow on the main street and red on the side street. The system remains in this state for 4 s when the short timer is *on.* This condition is expressed as T_S. The system transitions to the next state when the short timer goes *off*. This condition is expressed as $\overline{T}_S$.

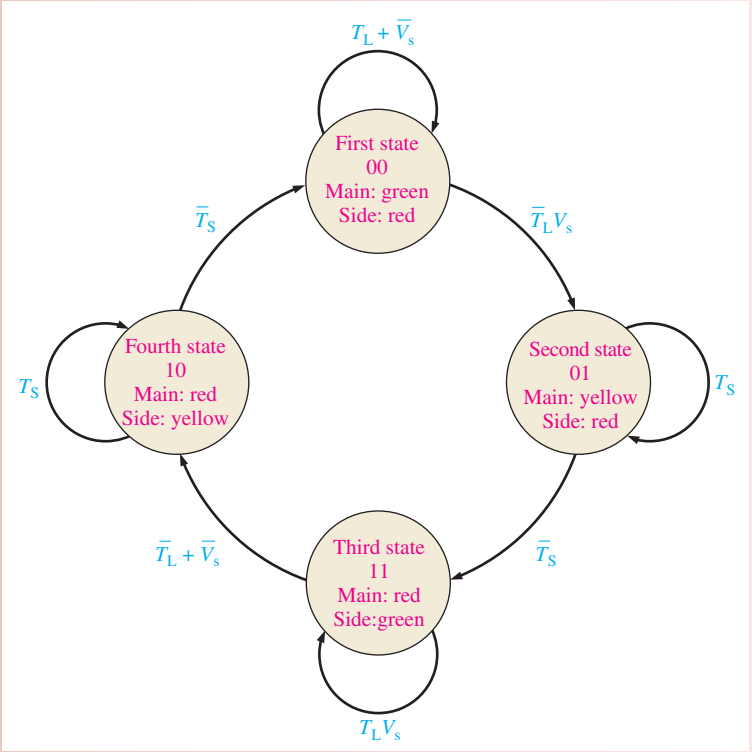


FIGURE 6–63 State diagram for the traffic signal control.

Third State The Gray code is 11. In this state, the light is red on the main street and green on the side street for 25 s when the long timer is *on* as long as there is a vehicle on the side street. This condition is expressed as $T_L V_s$. The system transitions to the next state when the long timer goes *off* or when there is no vehicle on the side street. This condition is expressed as $\overline{T}_L + \overline{V}_s$.

Fourth State The Gray code is 10. In this state, the light is red on the main street and yellow on the side street. The system remains in this state for 4 s when the short timer is *on.* This condition is expressed as T_S. The system transitions back to the first state when the short timer goes *off*. This condition is expressed as $\overline{T}_S$.

Exercise

1. How long can the system remain in the first state?
2. How long can the system remain in the fourth state?
3. Write the expression for the condition that produces a transition from the first state to the second state.
4. Write the expression for the condition that keeps the system in the second state.

Block Diagram

The traffic signal controller consists of three units: combinational logic, sequential logic, and timing circuits, as shown in Figure 6–64. The combinational logic unit provides outputs to turn the signal lights on and off. It also provides trigger outputs to start the long and short timers. The input sequence to this logic represents the four states described by the state diagram. The timing circuits unit provides the 25 s and the 4 s timing outputs. A frequency divider in the timing circuits unit divides the system clock down to a 1 Hz clock for use in producing the 25 s and 4 s signals. The sequential logic unit produces the sequence of 2-bit Gray codes representing the four states.

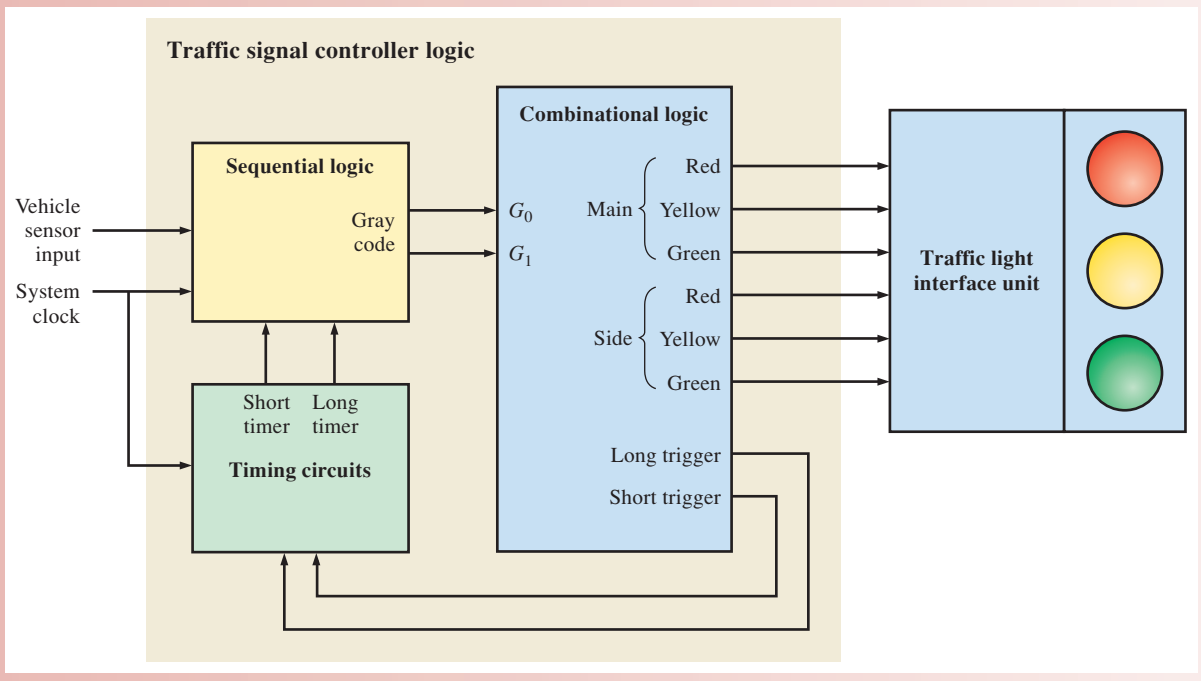


FIGURE 6–64 Block diagram of the traffic signal controller.

The Combinational Logic

The combinational logic consists of a state decoder, light output logic, and trigger logic, as shown in Figure 6–65.

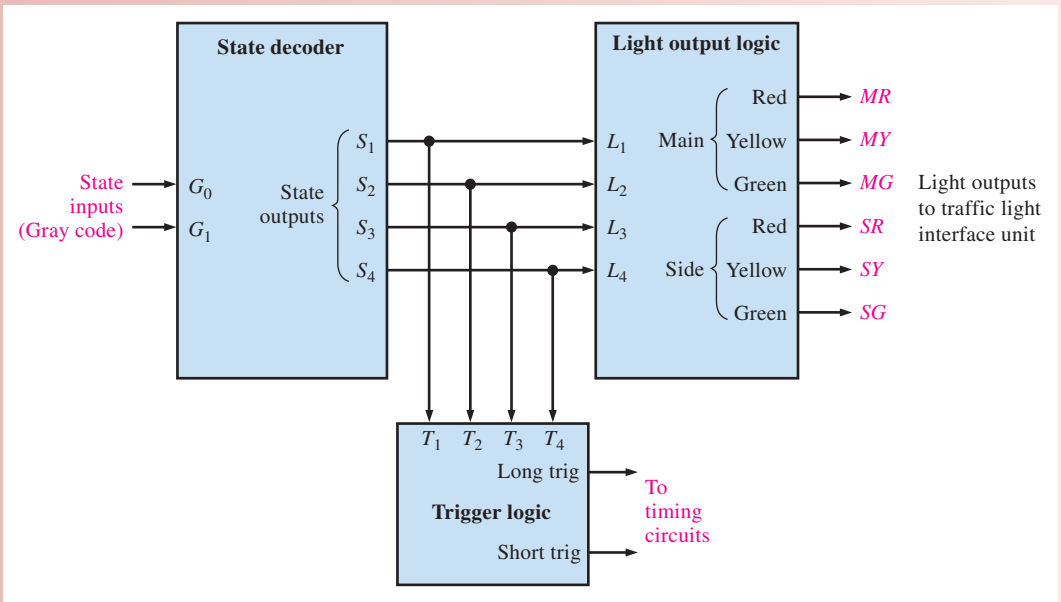


FIGURE 6–65 Block diagram of the combinational logic unit.

State Decoder This logic decodes the 2-bit Gray code from the sequential logic to determine which of the four states the system is in. The inputs to the state decoder are the two Gray code bits G_1 and G_0. There are four state outputs S_1, S_2, S_3, and S_4. For each of the

four input codes, one and only one of the outputs is activated. The Boolean expressions for the state outputs in terms of the inputs are

$$S_1 = \overline{G}_1\overline{G}_0$$
$$S_2 = \overline{G}_1 G_0$$
$$S_3 = G_1 G_0$$
$$S_4 = G_1\overline{G}_0$$

The truth table for the state decoder logic is shown in Table 6–11, and the logic diagram is shown in Figure 6–66.

TABLE 6–11

Truth table for the state decoder.

State Inputs (Gray Code)		State Outputs			
G_1	G_0	S_1	S_2	S_3	S_4
0	0	1	0	0	0
0	1	0	1	0	0
1	1	0	0	1	0
1	0	0	0	0	1

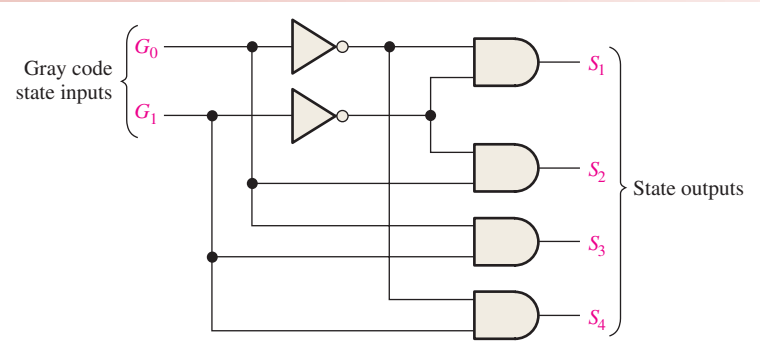


FIGURE 6–66 State decoder logic.

Light Output Logic This logic has the four state outputs (S_1–S_4) of the state decoder as its inputs (L_1–L_4) and produces six outputs to turn the traffic lights on and off. These outputs are designated *MR*, *MY*, *MG* (main red, main yellow, main green) and *SR*, *SY*, *SG* (side red, side yellow, side green).

The state diagram shows that the main red is *on* in the third state (L_3) or in the fourth state (L_4), so the Boolean expression is

$$MR = L_3 + L_4$$

The main yellow is *on* in the second state (L_2), so the expression is

$$MY = L_2$$

The main green is *on* in the first state (L_1), so the expression is

$$MG = L_1$$

Similarly, the state diagram is used to obtain the following expressions for the side street:

$$SR = L_1 + L_2$$
$$SY = L_4$$
$$SG = L_3$$

The logic circuit is shown in Figure 6–67.

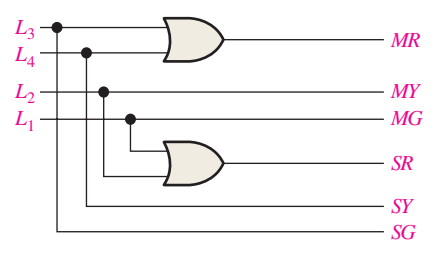


FIGURE 6–67 Light output logic.

Exercise

5. Show the logic diagram for the light output logic using specific IC devices with pin numbers.
6. Develop a truth table for the light output logic.

Trigger Logic The trigger logic produces two outputs, the long trigger output and the short trigger output. The long trigger output initiates the 25 s timer on a LOW-to-HIGH transition at the beginning of the first or third states. The short trigger output initiates the 4 s timer on a LOW-to-HIGH transition at the beginning of the second or fourth states. The Boolean expressions for this logic are

$$LongTrig = T_1 + T_3$$
$$ShortTrig = T_2 + T_4$$

Equivalently,

$$LongTrig = T_1 + T_3$$
$$ShortTrig = \overline{T_1 + T_3}$$

The logic circuit is shown in Figure 6–68.

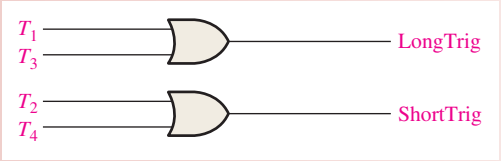


FIGURE 6–68 Trigger logic.

Exercise

7. Show the logic diagram for the trigger logic using specific IC devices with pin numbers.
8. Develop a truth table for the trigger logic.
9. Show the complete combinational logic by combining the state decoder, light output logic, and trigger logic. Include specific IC devices and pin numbers.

VHDL Descriptions

The VHDL program for the combinational logic unit of the traffic signal controller can be written using the data flow approach to describe each of the three functional blocks of the combinational logic unit. These functional blocks are the state decoder, the light output logic, and the trigger logic, as shown in Figure 6–65.

- The VHDL program code for the state decoder is as follows:

```
entity StateDecoder is
   port (G0, G1: in bit; S1, S2, S3, S4: out bit);
end entity StateDecoder;

architecture LogicOperation of StateDecoder is
begin
   S1 <= not G0 and not G1;
   S2 <= G0 and not G1;
   S3 <= G0 and G1;
   S4 <= not G0 and G1;
end architecture LogicOperation;
```

G0, G1: Gray code inputs
S1–S4: State outputs

Boolean expressions for state decoder outputs

- The VHDL program code for the light output logic is as follows:

```
entity LightOutputLogic is
   port (L1, L2, L3, L4: in bit; MR, MY, MG, SR, SY, SG: out bit);
end entity LightOutputLogic;

architecture LogicOperation of LightOutputLogic is
begin
   MR <= L3 or L4;
   MY <= L2;
   MG <= L1;
   SR <= L1 or L2;
   SY <= L4;
   SG <= L3;
end architecture LogicOperation;
```

Inputs and outputs declared

Boolean expressions for light output logic outputs

- The VHDL program code for the trigger logic is as follows:

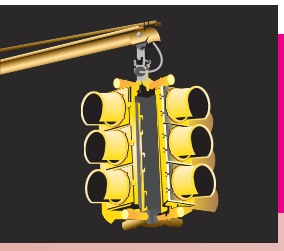
```
entity TriggerLogic is
   port (T1, T2, T3, T4: in bit; LongTrig, ShortTrig: out bit);
end entity TriggerLogic;

architecture LogicOperation of TriggerLogic is
begin
   LongTrig <= T1 or T3;
   ShortTrig <= T2 or T4;
end architecture LogicOperation;
```

Inputs and outputs declared

Boolean expressions for trigger logic outputs

Development of the traffic signal controller will continue in the Applied Logic in Chapter 7.



Applied Logic

Traffic Signal Controller: Part 2

The combinational logic unit of the traffic signal controller was completed in Chapter 6. Now, the timing circuits and sequential logic are developed. Recall that the timing circuits produce a 25 s time interval for the red and green lights and a 4 s interval for the yellow caution light. These outputs will be used by the sequential logic. The block diagram of the complete traffic signal controller is shown in Figure 7–64.

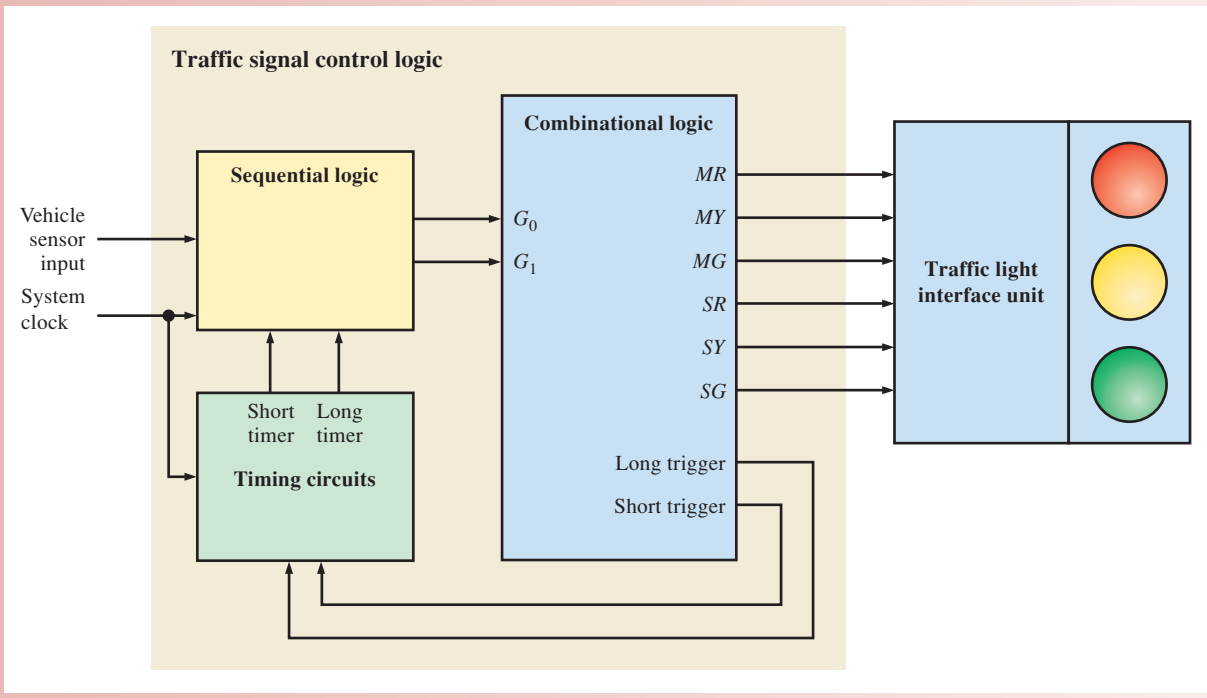


FIGURE 7–64 Block diagram of the traffic signal controller.

Timing Circuits

The timing circuits unit of the traffic signal controller consists of a 25 s timer and a 4 s timer and a clock generator. One way to implement this unit is with two 555 timers configured as one-shots and one 555 timer configured as an astable multivibrator (oscillator), as discussed earlier in this chapter. Component values are calculated based on the formulas given.

Another way to implement the timing circuits is shown in Figure 7–65. An external 24 MHz system clock (arbitrary value) is divided down to an accurate 1 Hz clock by the frequency divider. The 1 Hz clock is then used to establish the 25 s and the 4 s intervals by counting the 1 Hz pulses. This approach lends itself better to a VHDL description.

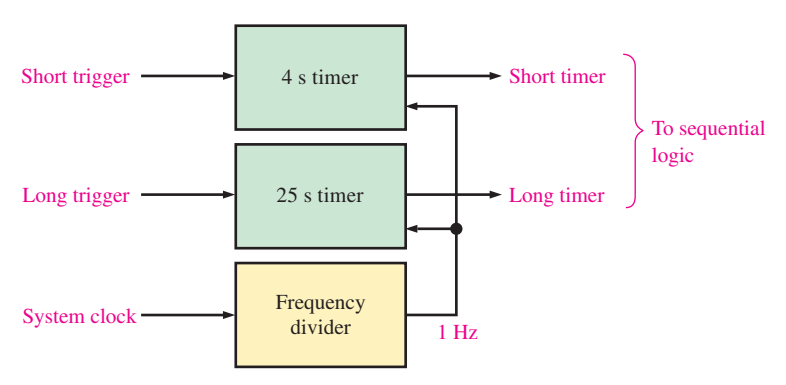


FIGURE 7–65 Block diagram of the timing circuits unit.

Exercise

1. Determine the values for the resistor and capacitor in a 25 s 555 timer.
2. Determine the values for the resistor and capacitor in a 4 s 555 timer.
3. What is the purpose of the frequency divider?

Controller Programming with VHDL

A programming model for the traffic signal controller is shown in Figure 7–66, where all the input and output labels are given. Notice that the Timing circuits block is split into two parts; the Frequency divider and the Timer circuits; and the Combinational logic block is divided into the State decoder and two logic sections (Light output logic and Trigger logic). This model will be used to develop the VHDL program codes.

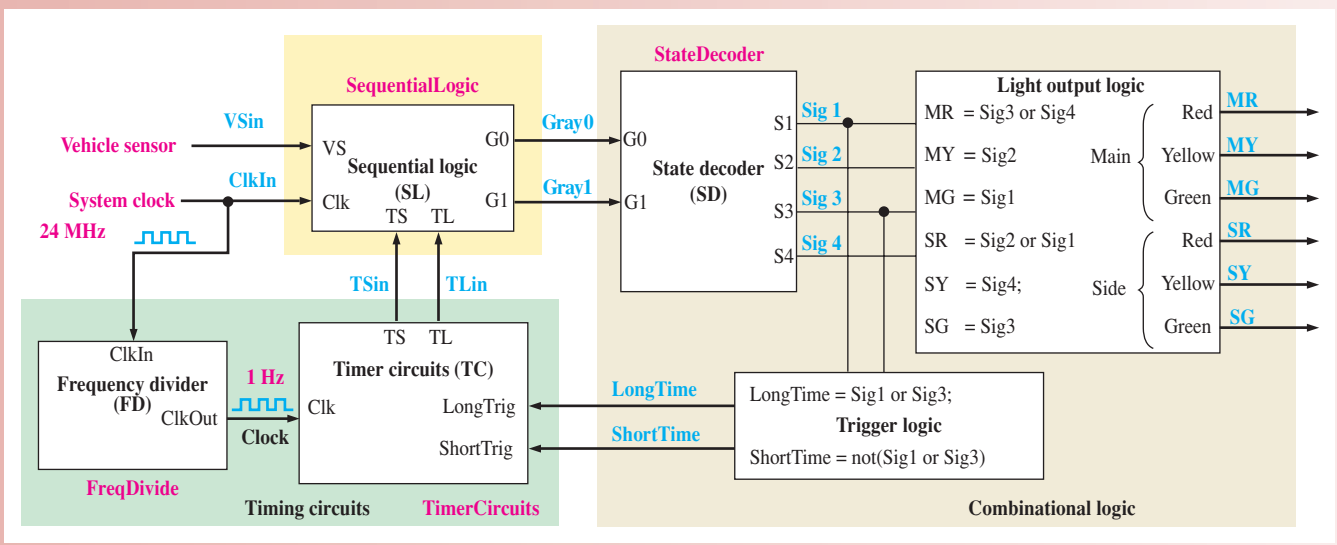


FIGURE 7–66 Programming model for the traffic signal controller.

Frequency Divider The purpose of the frequency divider is to produce a 1 Hz clock for the timer circuits. The input ClkIn in this application is a 24.00 MHz oscillator that drives the program code. SetCount is used to initialize the count for a 1 Hz interval. The program

FreqDivide counts up from zero to the value assigned to SetCount (one-half the oscillator speed) and inverts the output identifier ClkOut.

The integer value Cnt is set to zero prior to operation. The clock pulses are counted and compared to the value assigned to SetCount. When the number of pulses counted reaches the value in SetCount, the output ClkOut is checked to see if it is currently set to a 1 or 0. If ClkOut is currently 0, ClkOut is assigned a 1; otherwise, ClkIn is set to 1. Cnt is assigned a value of 0 and the process repeats. Toggling the output ClkOut each time the value of SetCount is reached creates a 1 Hz clock output with a 50% duty cycle.

The VHDL program code for the frequency divider is as follows:

```
library ieee;                                          Clkln: 24.00 MHz clock driver
use ieee.std_logic_1164.all;                           ClkOut: Output at 1 Hz

entity FreqDivide is
port(Clkln, in std_logic;                              Cnt: Counts up to value in SetCount
     ClkOut: buffer std_logic);                        SetCount: Holds 1/2 timer interval value
end entity FreqDivide;

architecture FreqDivide Behavior of FreqDivide is
begin
  FreqDivide: process(Clkln)
  variable Cnt: integer := 0;
  variable SetCount: integer;

begin
  SetCount := 12000000; -- 1/2 duty cycle              SetCount is assigned a value equal to half the system clock to produce a 1 Hz output. In this case, a 24 MHz system clock is used.
  if (ClkIn'EVENT and ClkIn = '1') then  <-- The if statement causes program to wait for a clock event and clock = 1 to start operation.
    if (Cnt = SetCount) then
      if ClkOut = '0' then                             Check that the terminal value in SetCount has been reached at which time ClkOut is toggled and Cnt is reset to 0.
        ClkOut <= '1'; --Output high 50%
      else
        ClkOut <= '0'; --Output Low 50%
      end if;
      Cnt := 0;
    else
      Cnt := Cnt + 1;  <-- If terminal value has not been reached, Cnt is incremented.
    end if;
  end if;
end process;
end architecture FreqDivideBehavior;
```

Timer Circuits The program TimerCircuits uses two one-shot instances consisting of a 25 s timer (TLong) and a 4 s timer (TShort). The 25 s and the 4 s timers are triggered by long trigger (LongTrig) and short trigger (ShortTrig). In the VHDL program, countdown timers driven by a 1 Hz clock input (Clk) replicate the one-shot components TLong and TShort. The values stored in SetCountLong and SetCountShort are assigned to the Duration inputs of one-shot components TLong and TShort, setting the 25-second and 4-second timeouts. When Enable is set LOW, the one-shot timer is initiated and output QOut is set HIGH. When the one-shot timers time out, QOut is set LOW. The output of one-shot component TLong is sent to TimerCircuits identifier TL. The output of one-shot component TShort is sent to TimerCircuits identifier TS.

The VHDL program code for the timing circuits is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity TimerCircuits is
   port(LongTrig, ShortTrig, Clk: in std_logic;
        TS, TL: buffer std_logic);
end entity TimerCircuits;

architecture TimerBehavior of TimerCircuits is
component OneShot is
   port(Enable, Clk: in std_logic;
        Duration :in integer range 0 to 25;
        QOut     :buffer std_logic);
end component OneShot;

signal SetCountLong, SetCountShort: integer range 0 to 25;
begin
   SetCountLong <= 25;
   SetCountShort <= 4;
   TLong:OneShot port map(Enable=>LongTrig, Clk=>Clk, Duration=>SetCountLong, QOut=>TL);
   TShort:OneShotport map(Enable=>ShortTrig, Clk=>Clk, Duration=>SetCountShort, QOut=>TS);
end architecture TimerBehavior;
```

LongTrig: Long timeout timer enable input
ShortTrig: Short timeout timer enable input
Clk: 1 Hz Clock input
TS: Short timer timeout signal
TL: Long timer timeout signal

Component declaration for OneShot.

SetCountLong: Holds long timer duration
SetCountShort: Holds short timer duration

Long and short count times are hard-coded to 25 and 4 based on a 1 Hz clock.

← Instantiation TLong

← Instantiation TShort

Sequential Logic

The sequential logic unit controls the sequencing of the traffic lights, based on inputs from the timing circuits and the side street vehicle sensor. The sequential logic produces a 2-bit Gray code sequence for each of the four states that were described in Chapter 6.

The Counter The sequential logic consists of a 2-bit Gray code counter and the associated input logic, as shown in Figure 7–67. The counter produces the four-state sequence on outputs G_0 and G_1. Transitions from one state to the next are determined by the short timer (T_S), the long timer (T_L), and vehicle sensor (V_s) inputs.

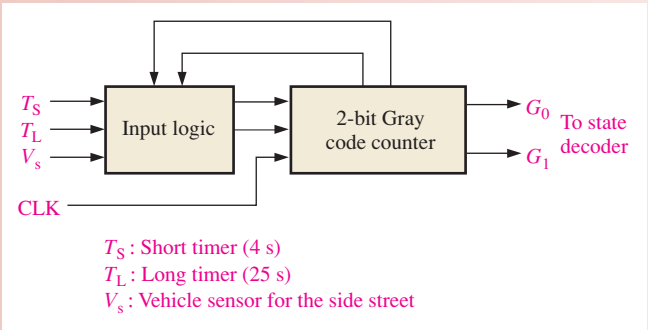


FIGURE 7–67 Block diagram of the sequential logic.

The diagram in Figure 7–68 shows how two D flip-flops can be used to implement the Gray code counter. Outputs from the input logic provide the *D* inputs to the flip-flops so they sequence through the proper states.

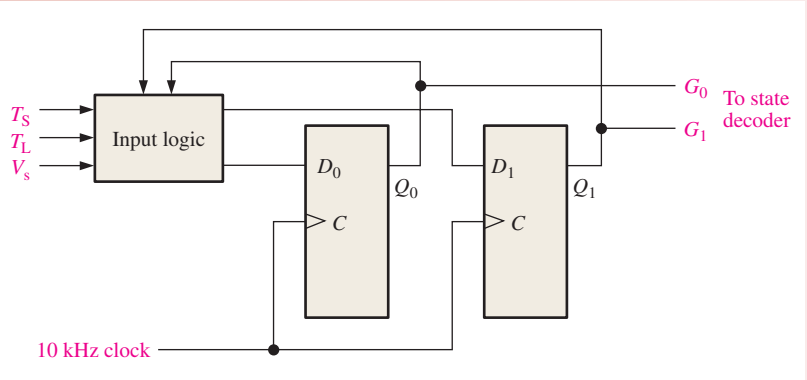


FIGURE 7–68 Sequential logic diagram with two *D* flip-flops used to implement the 2-bit Gray code counter.

The D flip-flop transition table is shown in Table 7–5. A next-state table developed from the state diagram in Chapter 6 Applied Logic is shown in Table 7–6. The subject of counter design is covered further in Chapter 8.

TABLE 7–5

D flip-flop transition table. Q_N is the output before clock pulse. Q_{N+1} is output after clock pulse.

Output Transitions			Flip-Flop Input
Q_N		Q_{N+1}	D
0	⟶	0	0
0	⟶	1	1
1	⟶	0	0
1	⟶	1	1

TABLE 7–6

Next-state table for the counter.

Present State		Next State			FF Inputs	
Q_1	Q_0	Q_1	Q_0	Input Conditions	D_1	D_0
0	0	0	0	$T_L + \overline{V}_s$	0	0
0	0	0	1	$\overline{T}_L V_s$	0	1
0	1	0	1	T_S	0	1
0	1	1	1	$\overline{T}_S$	1	1
1	1	1	1	$T_L V_s$	1	1
1	1	1	0	$\overline{T}_L + \overline{V}_s$	1	0
1	0	1	0	T_S	1	0
1	0	0	0	$\overline{T}_S$	0	0

The Input Logic Using Tables 7–5 and 7–6, the conditions required for each flip-flop to go to the 1 state can be determined. For example, G_0 goes from 0 to 1 when the present state is 00 and the condition on input D_0 is $\overline{T}_L V_s$, as indicated on the second row of Table 7–6. D_0 must be a 1 to make G_0 go to a 1 or to remain a 1 on the next clock pulse. A Boolean expression describing the conditions that make D_0 a 1 is derived from Table 7–6 as follows:

$$D_0 = \overline{G}_1\overline{G}_0\overline{T}_L V_s + \overline{G}_1 G_0 T_S + \overline{G}_1 G_0 \overline{T}_S + G_1 G_0 T_L V_s$$

In the two middle terms, the T_S and the $\overline{T}_S$ variables cancel, leaving the expression

$$D_0 = \overline{G}_1\overline{G}_0\overline{T}_L V_s + \overline{G}_1 G_0 + G_1 G_0 T_L V_s$$

Also, from Table 7–6, an expression for D_1 can be developed as follows:

$$D_1 = \overline{G}_1 G_0 \overline{T}_S + G_1 G_0 T_L V_s + G_1 G_0 \overline{T}_L + G_1 G_0 \overline{V}_s + G_1 \overline{G}_0 T_S$$

Based on the minimized expression for D_0 and D_1, the complete sequential logic diagram is shown in Figure 7–69.

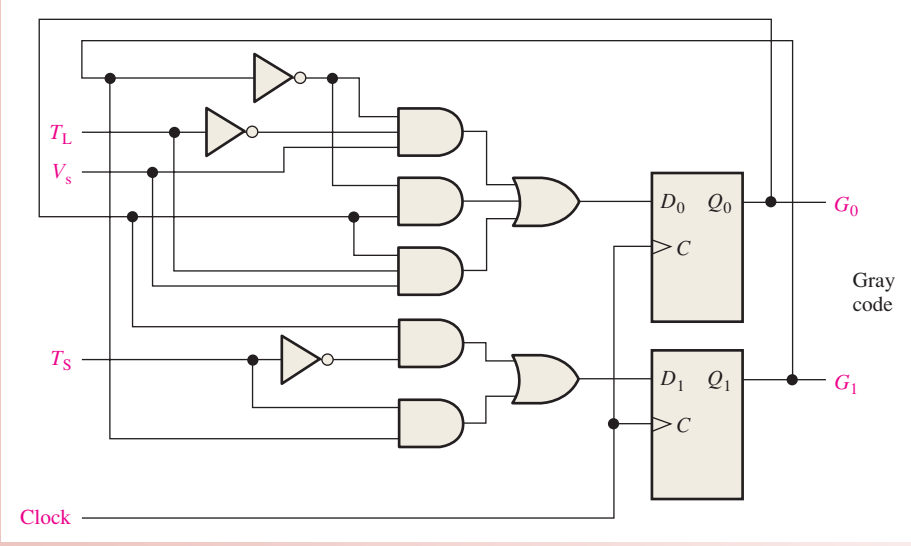


FIGURE 7–69 Complete diagram for the sequential logic.

Exercise

4. State the Boolean law and rule that permits the cancellation of T_S and $\overline{T}_S$ in the expression for D_0.
5. Use the Karnaugh map to reduce the D_0 expression further to a minimum form.
6. Use Boolean laws, rules, and/or the Karnaugh map to reduce the D_1 expression to a minimum form.
7. Do your minimized expressions for D_0 and D_1 agree with the logic shown in Figure 7–69?

The Sequential Logic with VHDL

The program SequentialLogic describes the Gray code logic needed to drive the traffic signal controller based on input from the timing circuits and the side street vehicle sensor. The sequential logic code produces a 2-bit Gray code sequence for each of the

four sequence states. The component definition dff is used to instantiate two D flip-flop instances DFF0 and DFF1. DFF0 and DFF1 produce the two-bit Gray code. The Gray code output sequences the traffic signal controller through each of four states. Internal variables D0 and D1 store the results of the D0 and D1 Boolean expressions developed in this chapter. The stored results in D0 and D1 are assigned to D flip-flops DFF0 and DFF1 along with the system clock to drive outputs G0 and G1 from the D flip-flop *Q* outputs.

The VHDL program code for the sequential logic is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity SequentialLogic is
port(VS, TL, TS, Clk: in std_logic; G0, G1: inout std_logic);
end entity SequentialLogic;

architecture SequenceBehavior of SequentialLogic is

component dff is
port (D, Clk: in std_logic; Q: out std_logic);
end component dff;

signal D0, D1: std_logic;
begin
D1 <= (G0 and not TS) or (G1 and TS);
D0 <= (not G1 and not TL and VS) or (not G1 and G0)
      or (G0 and TL and VS);

DFF0: dff port map(D=> D0, Clk => Clk, Q => G0);
DFF1: dff port map(D=> D1, Clk => Clk, Q => G1);

end architecture SequenceBehavior;
```

VS: Vehicle sensor input
TL: Long timer input
TS: Short timer input
Clk: System clock
G0: Gray code output bit 0
G1: Gray code output bit 1
D0: Logic for DFlipFlop DFF0
D1: Logic for DFlipFlop DFF1

Component declaration for D flip-flop (dff)

Logic definitions for D flip-flop inputs D0 and D1 derived from Boolean expressions developed in this chapter.

Component instantiations

The Complete Traffic Signal Controller

The program TrafficLights completes the traffic signal controller. Components FreqDivide, TimerCircuits, SequentialLogic, and StateDecoder are used to compose the completed system. Signal CLKin from the TrafficLights program source code is the clock input to the FreqDivide component. The frequency divided output ClkOut is stored as local variable Clock and is the divided clock input to the TimerCircuits and SequentialLogic components. TimerCircuits is controlled by local variables LongTime and ShortTime, which are controlled by the outputs Sig1 and Sig3 from component StateDecoder. StateDecoder also provides outputs Sig1 through Sig4 to control the traffic lights MG, SG, MY, SY, MR, and SR. TimerCircuit timeout signals TS and TL are stored in variables TLin (timer long in) and TSin (timer short in).

Signals TSin and TLin from TimerCircuits are used along with vehicle sensor VSin as inputs to the SequentialLogic component. The outputs from SequentialLogic G0 and G1 are stored in variables Gray0 and Gray1 as inputs to component StateDecoder. Component StateDecoder returns signals S1 through S4 which are in turn passed to variables Sig1 through Sig4. The light output logic and trigger logic developed in Chapter 6 are not used as components in this program, but are stated as logic expressions. The values stored in variables Sig1 through Sig4 provide the logic for outputs MG, SG, MY, SY, MR, SR; and local timer triggers LongTime and ShortTime are sent to TimerCircuits.

The VHDL program code for the traffic signal controller is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity TrafficLights is
port(VSin, ClkIn: in std_logic; MR, SR, MY, SY, MG, SG: out std_logic);
end entity TrafficLights;

architecture TrafficLightsBehavior of TrafficLights is

component StateDecoder is
port(G0, G1: in std_logic; S1, S2, S3, S4: out std_logic);
end component StateDecoder;

component SequentialLogic is
port(VS, TL, TS, Clk: in std_logic; G0, G1: inout std_logic);
end component SequentialLogic;

component TimerCircuits is
port(LongTrig, ShortTrig, Clk: In std_logic; TS, TL: buffer std_logic);
end component TimerCircuits;

component FreqDivide is
port(Clkin: in std_logic; ClkOut: buffer std_logic);
end component FreqDivide;

signal Sig1, Sig2, Sig3, Sig4, Gray0, Gray1: std_logic;
signal LongTime, ShortTime, TLin, TSin, Clock: std_logic;

begin
MR <= Sig3 or Sig4;
SR <= Sig2 or Sig1;
MY <= Sig2;
SY <= Sig4;
MG <= Sig1;
SG <= Sig3;

LongTime <= Sig1 or Sig3;
ShortTime <= not(Sig1 or Sig3);

SD: StateDecoder     port map (G0 => Gray0, G1 => Gray1, S1 => Sig1, S2 => Sig2, S3 => Sig3, S4 => Sig4);
SL: SequentialLogic  port map (VS => VSin, TL => TLin, TS => TSin, Clk => Fout, G0 => Gray0, G1 => Gray1);
TC: TimerCircuits    port map (LongTrig=>LongTime, ShortTrig=>ShortTime, Clk=>Clock, TS=>TSin, TL=>TLin);
FD: FreqDivide       port map (Clkln => CLKin, ClkOut =>-Clock);

end architecture TrafficLightsBehavior;
```

VSin : Vehicle sensor input
CLKin : System Clock
MR : Main red light output
SR : Side red light output
MY : Main yellow light output
SY : Side yellow light output
MG : Main green light output
SG : Side green light output

Component declaration for StateDecoder

Component declaration for SequentialLogic

Component declaration for TimerCircuits

Component declaration for FreqDivider

Logic definitions for the light output logic

Sig1-4 : Return values from StateDecoder
Gray0-1 : SequentialLogic Gray code return
LongTime : Trigger input to TimerCircuits
ShortTime : Trigger input to TimerCircuits
TLin : Store TimerCircuits long timeout
TSin : Store TimerCircuits Short timeout
Clock : Divided clock from FreqDivide

Logic definitions for the trigger logic

Component instantiations

Simulation

MultiSim

Open file AL07 in the Applied Logic folder on the website. Run the traffic signal controller simulation using your Multisim software and observe the operation. Lights will appear randomly when first turned on. Simulation times may vary.

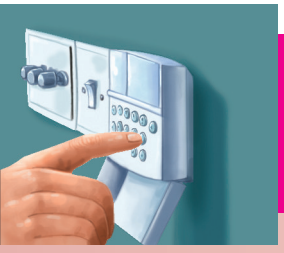
Putting Your Knowledge to Work

Add your modification for the pedestrian input developed in Chapter 6 and run a simulation.

When measuring digital signals with an oscilloscope, you should always use dc coupling, rather than ac coupling. The reason that ac coupling is not best for viewing digital signals is that the 0 V level of the signal will appear at the *average* level of the signal, not at true ground or 0 V level. It is much easier to find a "floating" ground or incorrect logic level with dc coupling. If you suspect an open ground in a digital circuit, increase the sensitivity of the scope to the maximum possible. A good ground will never appear to have noise under this condition, but an open will likely show some noise, which appears as a random fluctuation in the 0 V level.

SECTION 8–7 CHECKUP

1. What is the purpose of providing a test input to a sequential logic circuit?
2. Generally, when an output waveform is found to be incorrect, what is the next step to be taken?



Applied Logic

Security System

A security system that provides coded access to a secured area is developed. Once a 4-digit security code is stored in the system, access is achieved by entering the correct code on a keypad. A block diagram for the security system is shown in Figure 8–42. The system consists of the security code logic, the code-selection logic, and the keypad. The keypad is a standard numeric keypad.

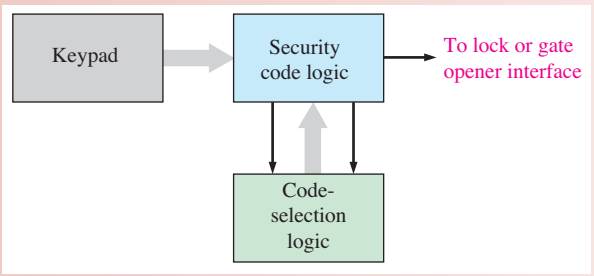


FIGURE 8–42 Block diagram of the security system.

Basic Operation

A 4-digit entry code is set into the memory with user-accessible DIP switches. Initially pressing the # key sets up the system for the first digit in the code. For entry, the code is entered one digit at a time on the keypad and converted to a BCD code for processing by the security code logic. If the entered code agrees with the stored code, the output activates the access mechanism and allows the door or gate, depending on the type of area that is secured, to be opened.

Exercise

1. Write the BCD code sequence produced by the code generator if the 4-digit access number 4739 is entered on the keypad.

The Security Code Logic

The security code logic compares the code entered on the keypad with the predetermined code from the code-selection logic. A logic diagram of the security code logic is shown in Figure 8–43.

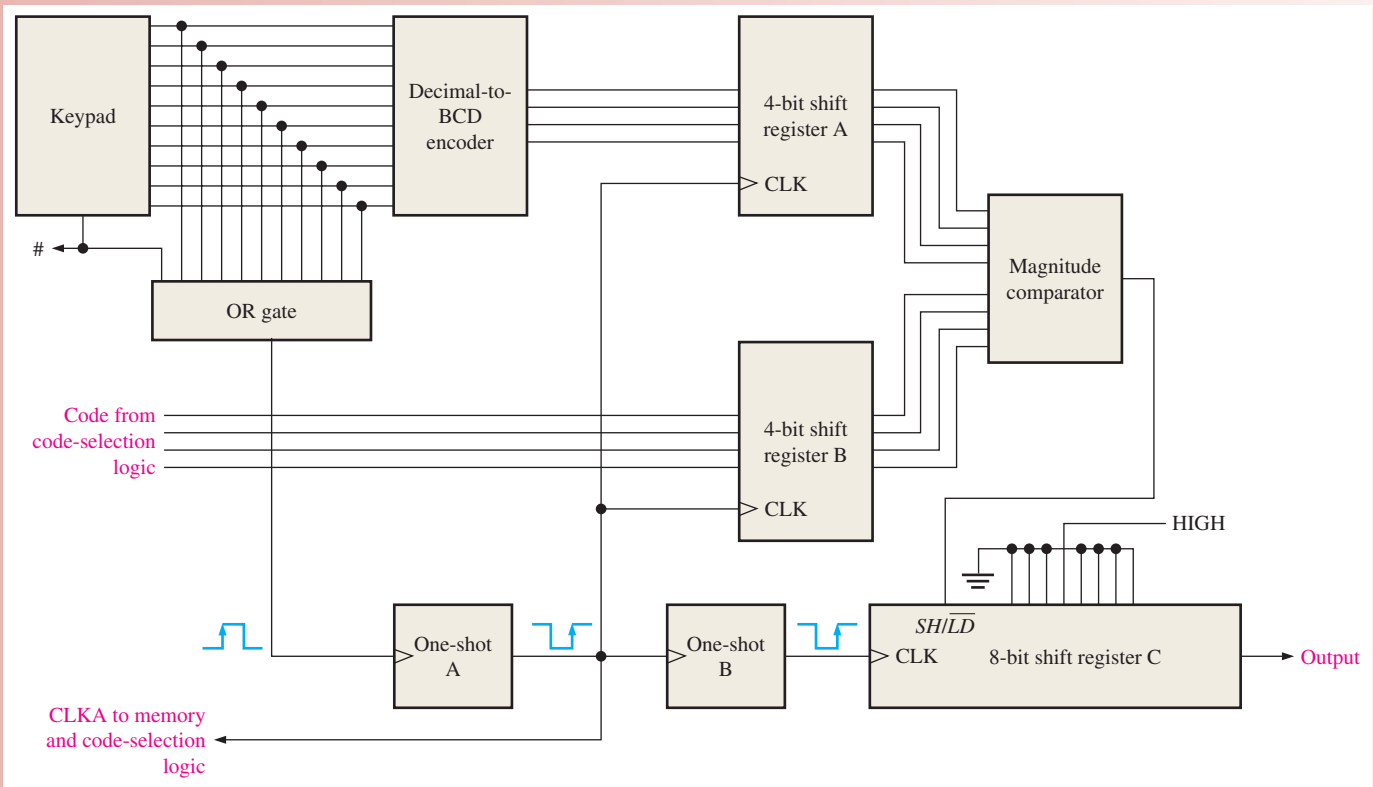


FIGURE 8–43 Block diagram of the security code logic with keypad.

In order to gain entry, first the # key on the keypad is pressed to trigger the one-shots, thus initializing the 8-bit register C with a preset pattern (00010000). Next the four digits of the code are entered in proper sequence on the keypad. As each digit is entered, it is converted to BCD by the decimal-to-BCD encoder, and a clock pulse is produced by one-shot A that shifts the 4-bit code into register A. The one-shot is triggered by a transition on the output of the OR gate when a key is pressed. At the same time, the corresponding digit from the code generator is shifted into register B. Also, one-shot B is triggered after one-shot A to provide a delayed clock pulse for register C to serially shift the preloaded pattern (00010000). The left-most three 0s are simply "fillers" and serve no purpose in the operation of the system. The outputs of registers A and B are applied to the comparator; if the codes are the same, the output of the comparator goes HIGH, placing shift register C in the SHIFT mode.

Each time an entered digit agrees with the preset digit, the 1 in shift register C is shifted right one position. On the fourth code agreement, the 1 appears on the output of the shift register and activates the mechanism to unlock the door or open the gate. If the code digits do not agree, the output of the comparator goes LOW, placing shift register C in the LOAD mode so the shift register is reinitialized to the preset pattern (00010000).

Exercise

2. What is the state of shift register C after two correct code digits are entered?
3. Explain the purpose of the OR gate.
4. If the digit 4 is entered on the keypad, what appears on the outputs of register A?

The Code-Selection Logic

A logic diagram of the code-selection logic is shown in Figure 8–44. This part of the system includes a set of DIP switches into which a 4-digit entry code is set. Initially pressing the # key sets up the system for the first digit in the code by causing a preset pattern to be loaded into the 4-bit shift register (0001). The four bits in the first code digit are selected by a HIGH on the Q_0 output of the shift register, enabling the four AND gates labeled A1–A4. As each digit of the code is entered on the keypad, the clock from the security code logic shifts the 1 in the shift register to sequentially enable each set of four AND gates. As a result, the BCD digits in the security code appear sequentially on the outputs.

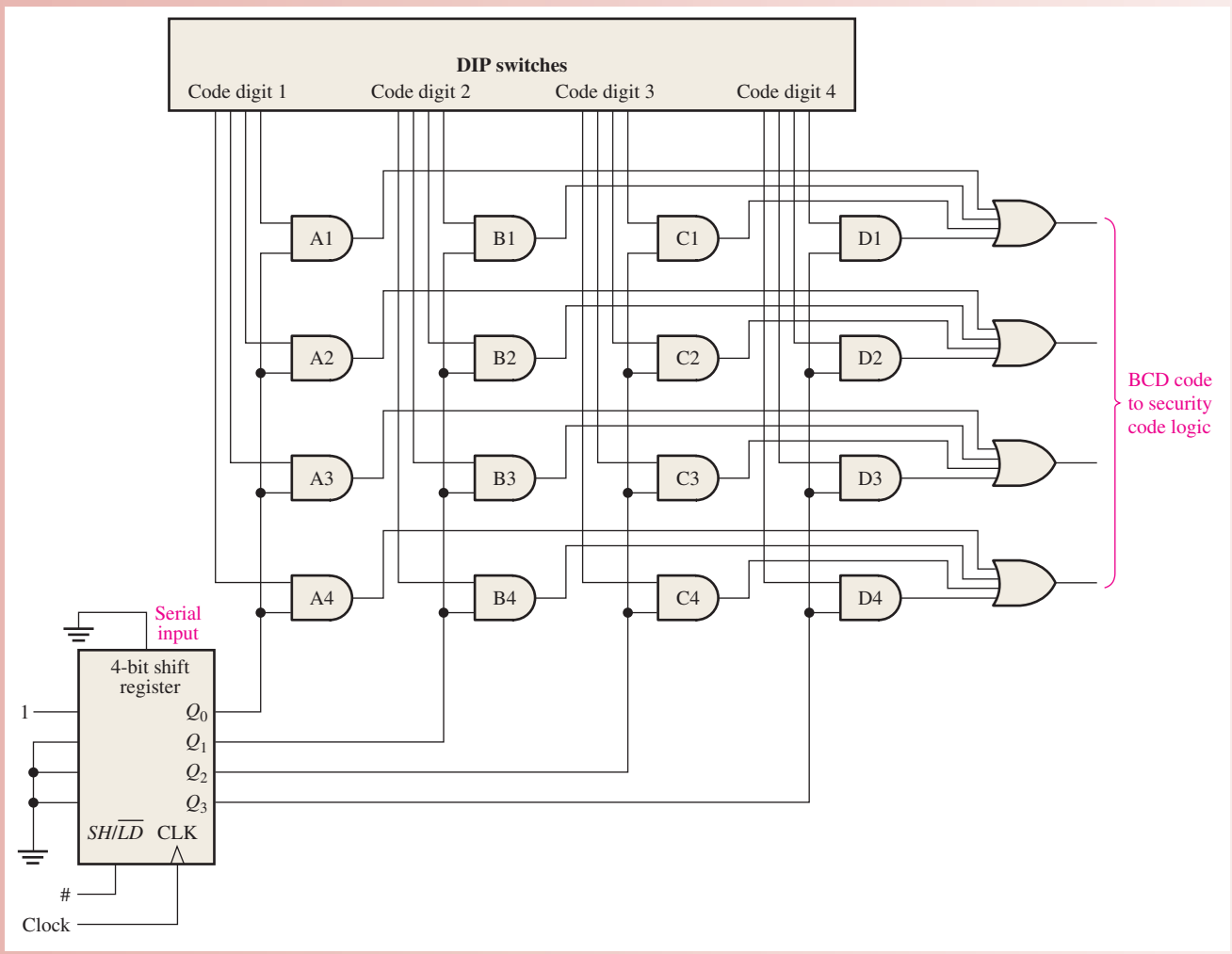


FIGURE 8–44 Logic diagram of the code-selection logic.

Security System with VHDL

The security system can be described using VHDL for implementation in a PLD. The three blocks of the system (keypad, security code logic, and code-selection logic) are combined in the program code to describe the complete system.

The security system block diagram is shown in Figure 8–45 as a programming model. Six program components perform the logical operations of the security system. Each component corresponds to a block or blocks in the figure. The security system program SecuritySystem contains the code that defines how the components interact.

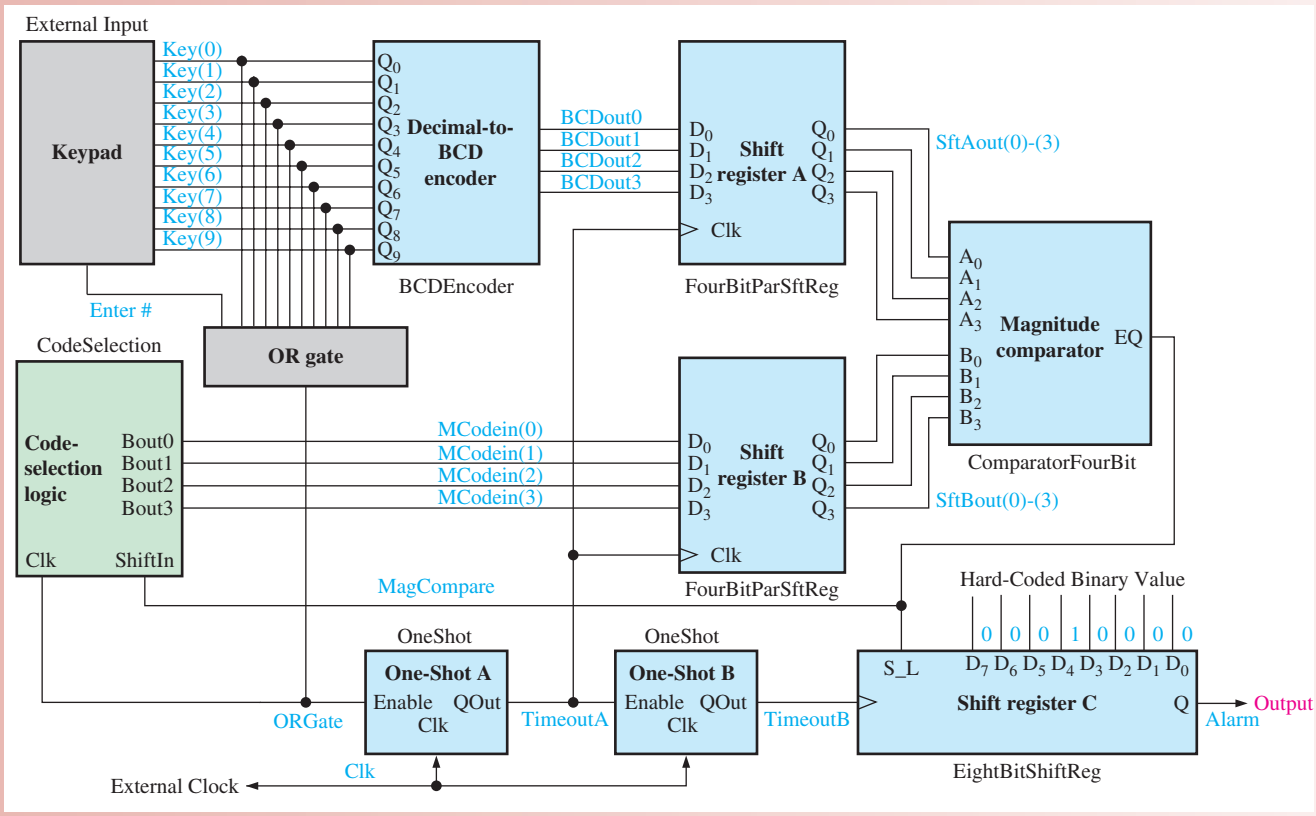


FIGURE 8–45 Security system block diagram as a programming model.

The security system includes a ten-bit input vector Key—one input bit for each decimal digit—and an input Enter, representing a typical numeric keypad. Once a key is pressed, the data stored in input array Key are sent to the decimal-to-BCD encoder (BCDEncoder). Its 4-bit output is then sent to the inputs of the 4-bit parallel in/parallel out shift register A (FourBitParSftReg). An external system clock applied to input Clk drives the overall security system. The Alarm output signal is set HIGH upon a successful arming operation.

Pressing the Enter key sends an initial HIGH clock signal to the code-selection logic block (CodeSelection), which loads an initial binary value of 1000 to shift register B. At this time, a binary 0000 is stored in shift register A, and the output of the magnitude comparator (ComparatorFourBit) is set LOW. The code-selection logic is now ready to present the first stored code value that is to be compared to the value of the first numeric keypad entry. At this time a LOW on the 8-bit parallel in/serial out shift register C (EightBitShiftReg) S_L input loads an initial value of 00010000.

When a numeric key is pressed, the output of the OR gate (ORGate) clocks the first stored value to the inputs of shift register B, and the output of the decimal-to-BCD encoder is sent to the inputs of shift register A. If the values in shift registers A and B match, the output of the magnitude comparator is set HIGH; and the code-selection logic is ready to clock in the next stored code value.

At the conclusion of four successful comparisons of stored code values against four correct keypad entries, the value 00010000 initially in shift register C will shift four places to the right, setting the Alarm output to a HIGH. An incorrect keypad entry will not match the stored code value in shift register B and the magnitude comparator will output a LOW. With the comparator output LOW, the code-selection logic will reset to the first stored code value; and the value 00010000 is reloaded into shift register C, starting the process over again.

To clock the keypad and the stored code values through the system, two one-shots (OneShot) are used. The one-shots allow data to stabilize before any action is taken. One-shot A receives an Enable signal from the keypad OR gate, which initiates the first timed process. The OR gate output is also sent to the code-selection logic, and the first code value from the code-selection logic is sent to the inputs of shift register A. When one-shot A times out, the selected keypad entry and the current code from the code-selection logic are stored in shift registers A and B for comparison by the magnitude comparator, and an Enable is sent to one-shot B. If the codes in shift registers A and B match, the value stored in shift register C shifts one place to the right after one-shot B times out.

The six components used in the security system program SecuritySystem are shown in Figure 8–46.

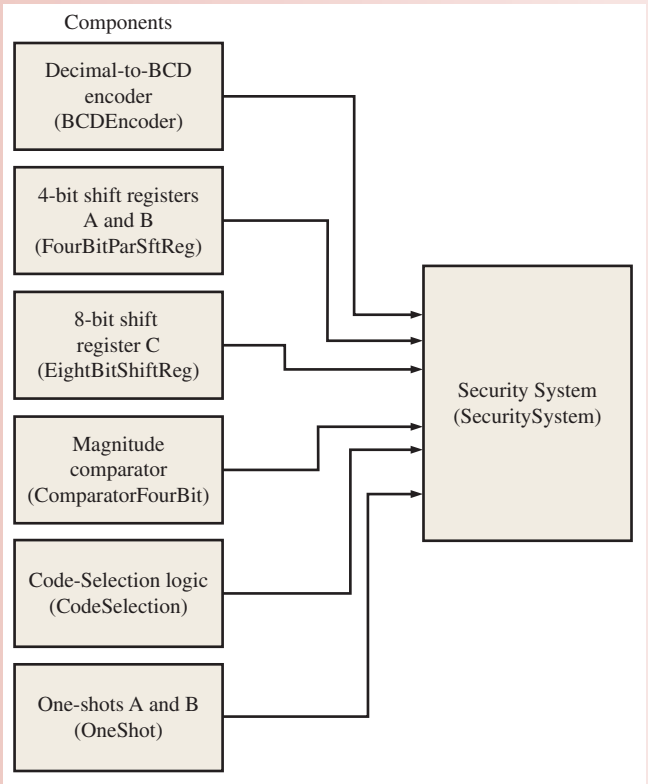


FIGURE 8–46 Security system components.

The VHDL program code for the security system is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
```
Key : 10 - Key input
Enter : # - Key input
Clk : System clock
Alarm : Alarm output

```
entity SecuritySystem is
port (key: in std_logic_vector(0 to 9); Enter: in std_logic;
      Clk: in std_logic; Alarm: out std_logic);
end entity SecuritySystem;

architecture SecuritySystemBehavior of SecuritySystem is

component BCDEncoder is
port(D: in std_logic_vector(0 to 9);
     Q: out std_logic_vector(0 to 3));
end component BCDEncoder;
```
Component declaration for BCDEncoder

```
component FourBitParSftReg is
port(D: in std_logic_vector(0 to 3);
     Clk: in std_logic;
     Q: out std_logic_vector(0 to 3));
end component FourBitParSftReg;
```
Component declaration for FourBitParSftReg

```
component ComparatorFourBit is
port(A, B: in std_logic_vector(0 to 3);
     EQ: out std_logic);
end component ComparatorFourBit;
```
Component declaration for ComparatorFourBit

```
component OneShot is
port(Enable, Clk: in std_logic;
     QOut: buffer std_logic);
end component OneShot;
```
Component declaration for OneShot

```
component EightBitShiftReg is
port(S_L, Clk: in std_logic;
     D: in std_logic_vector(0 to 7);
     Q: buffer std_logic);
end component EightBitShiftReg;
```
Component declaration for EightBitShiftReg

```
component CodeSelection is
port(ShiftIn, Clk: in std_logic;
     Bout: out std_logic_vector(1 to 4));
end component CodeSelection;
```
Component declaration for CodeSelection

```
signal BCDout: std_logic_vector(0 to 3);
signal SftAout: std_logic_vector(0 to 3);
signal SftBout: std_logic_vector(0 to 3);
signal MCodein: std_logic_vector(0 to 3);
signal ORgate: std_logic;
signal MagCompare: std_logic;
signal TimeoutA, TimeoutB: std_logic;
```
BDCout: BCD encoder return
SftAout: Shift Register A return
SftBout: Shift Register B return
MCodein: Security Code value
ORgate: OR output from 10-keypad
MagCompare: Key entry to code compare
TimeoutA/B: One-shot timer variables

```
begin
ORgate <= (Key(0) or Key(1) or Key(2) or Key(3) or Key(4)
          or key(5) or Key(6) or Key(7) or Key(8) or Key(9));
```

Logic definition for ORGate

```
BCD: BCDEncoder
port map(D(0)=>Key(0),D(1)=>Key(1),D(2)=>Key(2),D(3)=>Key(3),
         D(4)=>Key(4),D(5)=>Key(5),D(6)=>Key(6),D(7)=>Key(7),D(8)=>Key(8),D(9)=>Key(9),
         Q(0)=>BCDout(0),Q(1)=>BCDout(1),Q(2)=>BCDout(2),Q(3)=>BCDout(3));
ShiftRegisterA: FourBitParSftReg
port map(D(0)=>BCDout(0),D(1)=>BCDout(1),D(2)=>BCDout(2),D(3)=>BCDout(3),
          Clk=>not TimeoutA,Q(0)=>SftAout(0),Q(1)=>SftAout(1),Q(2)=>SftAout(2),Q(3)=>SftAout(3));
ShiftRegisterB: FourBitParSftReg
port map(D(0)=>MCodein(0),D(1)=>MCodein(1),D(2)=>MCodein(2),D(3)=>MCodein(3),
        Clk=>not TimeoutA,Q(0)=>SftBout(0),Q(1)=>SftBout(1),Q(2)=>SftBout(2),Q(3)=>SftBout(3));
Magnitude Comparator: ComparatorFourBit port map(A=>SftAout,B=>SftBout,EQ=>MagCompare);
OSA:OneShot port map(Enable=>Enter or ORgate,Clk=>Clk,QOut=>TimeoutA);
OSB:OneShot port map(Enable=>not TimeoutA,Clk=>Clk,QOut=>TimeoutB);
ShiftRegisterC:EightBitShiftReg
port map(S_L=>MagCompare,Clk=> TimeoutB,D(0)=>'0',D(1)=>'0',
        D(2)=>'0',D(3)=>'1',D(4)=>'0',D(5)=>'0',D(6)=>'0',D(7)=>'0',Q=>Alarm);
CodeSelectionA: CodeSelection
port map(ShiftIn=>MagCompare,Clk=>Enter or ORGate,Bout=>MCodein);
end architecture SecuritySystemBehavior;
```

Component instantiations

Simulation

MultiSim

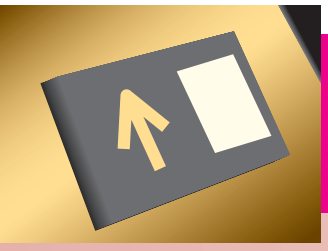
Open File AL08 in the Applied Logic folder on the website. Run the security code logic simulation using your Multisim software and observe the operation. A DIP switch is used to simulate the 10-digit keypad and switch J1 simulates the # key. Switches J2–J5 are used for test purposes to enter the code that is produced by the code selection logic in the complete system. Probe lights are used only for test purposes to indicate the states of registers A and B, the output of the comparator, and the output of register C.

Putting Your Knowledge to Work

Explain how the security code logic can be modified to accommodate a 5-digit code.

SUMMARY

- The basic types of data movement in shift registers are
  1. Serial in/shift right/serial out
  2. Serial in/shift left/serial out
  3. Parallel in/serial out
  4. Serial in/parallel out



Applied Logic

Elevator Controller: Part 1

This Applied Logic describes the operation and implementation of a service elevator controller for a seven-story building. The controller consists of logic that controls the elevator operation, a counter that determines the floor at which the elevator is located at any given time, and a floor number display. For simplicity, there is only one floor call and one floor request for each elevator cycle. A cycle occurs when the elevator is called to a given floor to pick up a passenger and the passenger is delivered to a requested floor. The elevator sequence for one cycle is shown in Figure 9–60.

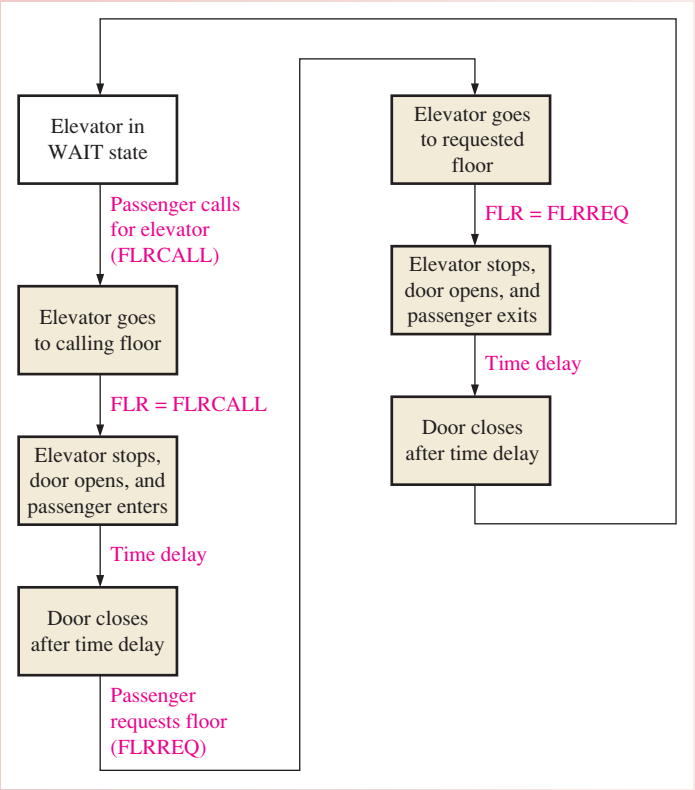


FIGURE 9–60 One cycle of the elevator operation.

The five states in the elevator control sequence are WAIT, DOWN, UP, STOP/OPEN, and CLOSE. In the WAIT state, the elevator is waiting on the last floor serviced for an external call button (FLRCALL) on any floor to be pressed. When there is a call for the elevator from any floor, the appropriate command (UP or DOWN) is issued. When the elevator arrives and stops at the calling floor, the door opens; the person enters and presses a number to request a destination floor. If the number of the requested floor is less than the number of the current floor, the elevator goes into the DOWN mode. If the number of the requested floor is greater than the number of the current floor, the elevator goes into the UP mode. The elevator goes to the STOP/OPEN mode at the requested floor to allow exit. After the door is open for a specified time, it closes and then goes back to the WAIT state until another floor call is received.

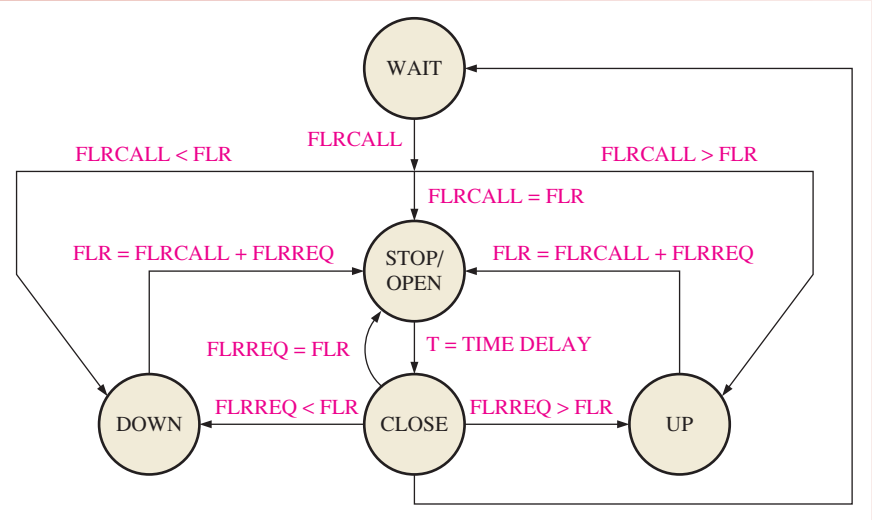


FIGURE 9–61 Elevator controller state diagram.

The following states are shown in the state diagram of Figure 9–61:

WAIT The system always begins in the WAIT state on the floor last serviced. When a floor call (FLRCALL) signal is received, the control logic determines if the number of the calling floor is greater than the current floor (FLRCALL > FLR), less than the current floor (FLRCALL < FLR), or equal to the current floor (FLRCALL = FLR) and puts the system in the UP mode, DOWN mode, or OPEN mode, respectively.

DOWN In this state, the elevator moves down toward the calling floor.

UP In this mode, the elevator moves up toward the calling floor.

STOP/OPEN This state occurs when the calling floor has been reached. When the number of the floor where the elevator is equals the number of the calling or requested floor, a signal is issued to stop the elevator and open the door.

CLOSE After a preset time (T) to allow entry or exit, the door closes.

The signals used by the elevator controller are defined as follows:

FLR Number of floor represented by a 3-bit binary code.

Floor sensor pulse A pulse issued at each floor to clock the floor counter to the next state.

FLRCALL Number of floor where a call for elevator service originates, represented by a 3-bit binary code.

Call pulse A pulse issued in conjunction with FLRCALL to clock the 3-bit code into a register.

FLRREQ Number of floor to which the passenger desires to go, represented by a 3-bit binary code.

Request pulse A pulse issued in conjunction with FLRREQ to clock the 3-bit code into a register.

UP A signal issued to the elevator motor control to cause the elevator to move from a lower floor to a higher floor.

DOWN A signal issued to the elevator motor control to cause the elevator to move from a higher floor to a lower floor.

STOP A signal issued to the elevator motor control to cause the elevator to stop.

OPEN A signal issued to door motor control to cause the door to open.

CLOSE A signal issued to the door motor control to cause the door to close.

Elevator Controller Block Diagram

Figure 9–62 shows the elevator controller block diagram, which consists of controller logic, a floor counter, and a floor number display. Assume that the elevator is on the first floor in

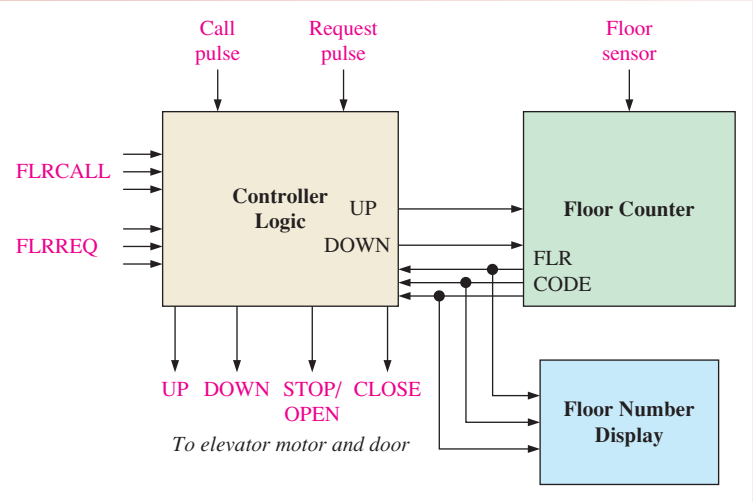


FIGURE 9–62 Elevator controller block diagram.

the WAIT state. The floor counter contains 001, which is the first floor code. Suppose the FLRCALL (101) comes in from the call button on the fifth floor. Since FLRCALL > FLR (101 > 001), the controller issues an UP command to the elevator motor. As the elevator moves up, the floor counter receives a floor sensor pulse as it reaches each floor which advances its state (001, 010, 011, 100, 101). When the fifth floor is reached and FLR = FLRCALL, the controller logic stops the elevator and opens its door. The process is repeated for a FLRREQ input.

The floor counter sequentially tracks the number of the floor and always contains the number of the current floor. It can count up or down and can reverse its state at any point under the direction of the state controller and the floor sensor input. A 3-bit counter is required since there are eight floors ($2^3 = 8$) including the basement, as shown in the floor counter state diagram in Figure 9–63.

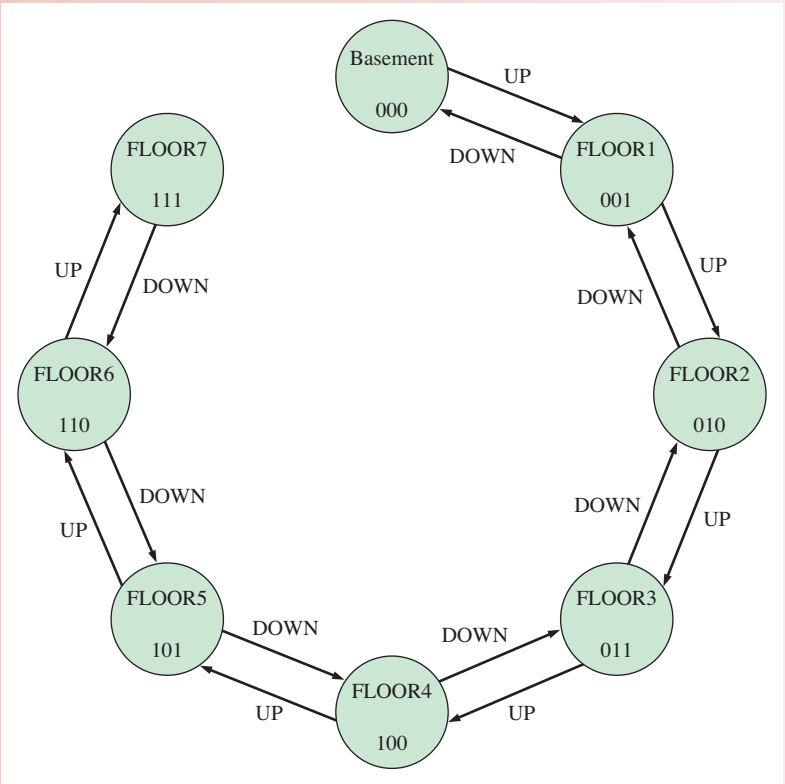


FIGURE 9–63 Floor counter state diagram.

Operation of Elevator Controller

The elevator controller logic diagram is shown in Figure 9–64. Elevator action is initiated by either a floor call (FLRCALL) or a floor request (FLRREQ). Keep in mind that FLRCALL is when a person calls the elevator to come to a particular floor. FLRREQ is when a passenger in the elevator requests to go to a specified floor. This simplified operation is based on a CALL/REQ sequence; that is, a call followed by a request followed by a call.

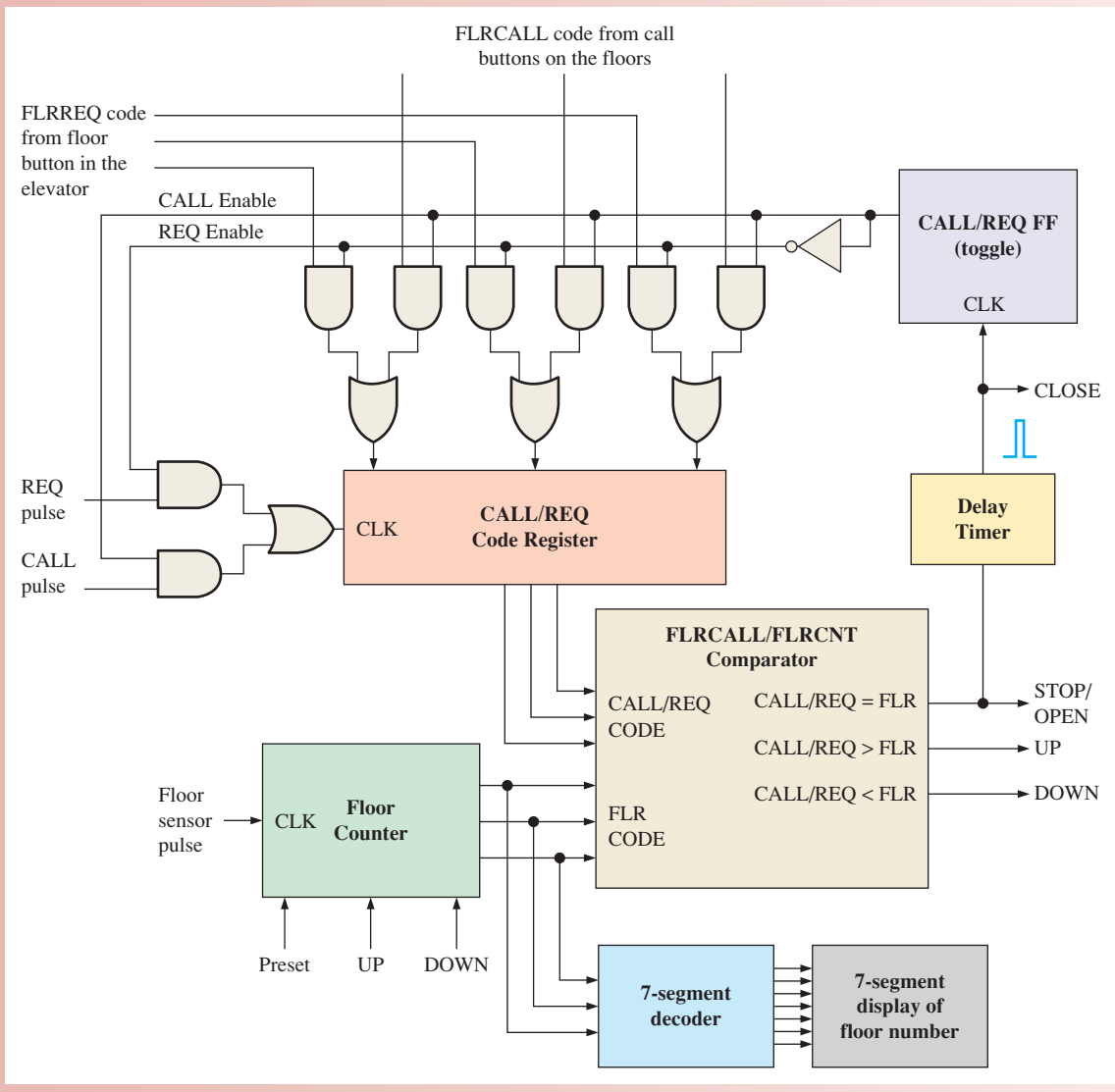


FIGURE 9–64 Elevator controller logic diagram.

As you know, FLRCALL and FLRREQ are 3-bit codes representing specific floors. When a person presses a call button on a given floor, the specific 3-bit code for that floor is placed on the inputs to the CALL/REQ code register and a CALL pulse is generated to enter the code into the register. The same process occurs when a request button is pressed inside the elevator. The code is input to the CALL/REQ code register, and a REQ pulse is generated to store the code in the register.

The elevator does not know the difference between a call and a request. The comparator determines if the destination floor number is greater than, less than, or equal to the current

floor where the elevator is located. As a result of this comparison, either an UP command, a DOWN command, or an OPEN command is issued to the elevator motor control. As the elevator moves toward the desired floor, the floor counter is either incremented at each floor as it goes up or decremented at each floor as it goes down. Once the elevator reaches the desired floor, a STOP/OPEN command is issued to the elevator motor control and to the door control. After a preset time, the delay timer issues a CLOSE signal to the elevator door control. As mentioned, this elevator design is limited to one floor call and one floor request per cycle.

Initialization The initial one-time setup requires that the elevator be placed at the basement level and the floor counter be preset to 000. After this, the counter will automatically move through the sequence of states determined by the elevator position.

Exercise

1. Explain the purpose of the floor counter.
2. Describe what happens during the WAIT mode.
3. How does the system know when the desired floor has been reached?
4. Discuss the limitations of the elevator design in Figure 9–64.

Implementation

The elevator controller can be implemented using fixed-function logic devices, a PLD programmed with a VHDL (or Verilog) code, or a programmed microcontroller or microprocessor. In the Chapter 10 Applied Logic, the VHDL program code for the elevator controller is presented. You will see how to program a PLD step by step.

Putting Your Knowledge to Work

What changes are required in the logic diagram of Figure 9–64 to upgrade the elevator controller for a ten-story building?

SUMMARY

- Asynchronous and synchronous counters differ only in the way in which they are clocked. The first stage of an asynchronous counter is driven by a clock pulse. Each succeeding stage is clocked by the output of the previous stage. In a synchronous counter, all stages are clocked by the same clock pulse. Synchronous counters can run at faster clock rates than asynchronous counters.
- The maximum modulus of a counter is the maximum number of possible states and is a function of the number of stages (flip-flops). Thus,

$$\text{Maximum modulus} = 2^n$$

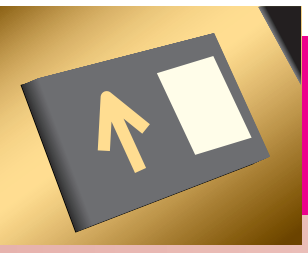
  where n is the number of stages in the counter. The modulus of a counter is the *actual* number of states in its sequence and can be equal to or less than the maximum modulus.
- The overall modulus of cascaded counters is equal to the product of the moduli of the individual counters.

KEY TERMS

Key terms and other bold terms in the chapter are defined in the end-of-book glossary.

Asynchronous Not occurring at the same time.

Cascade To connect "end-to-end" as when several counters are connected from the terminal count output of one counter to the enable input of the next counter.



Applied Logic

Elevator Controller: Part 2

In this section, the elevator controller that was introduced in the Applied Logic in Chapter 9 will be programmed for implementation in a PLD. Refer to Chapter 9 to review the elevator operation. The logic diagram is repeated in Figure 10–62 with labels changed to facilitate programming.

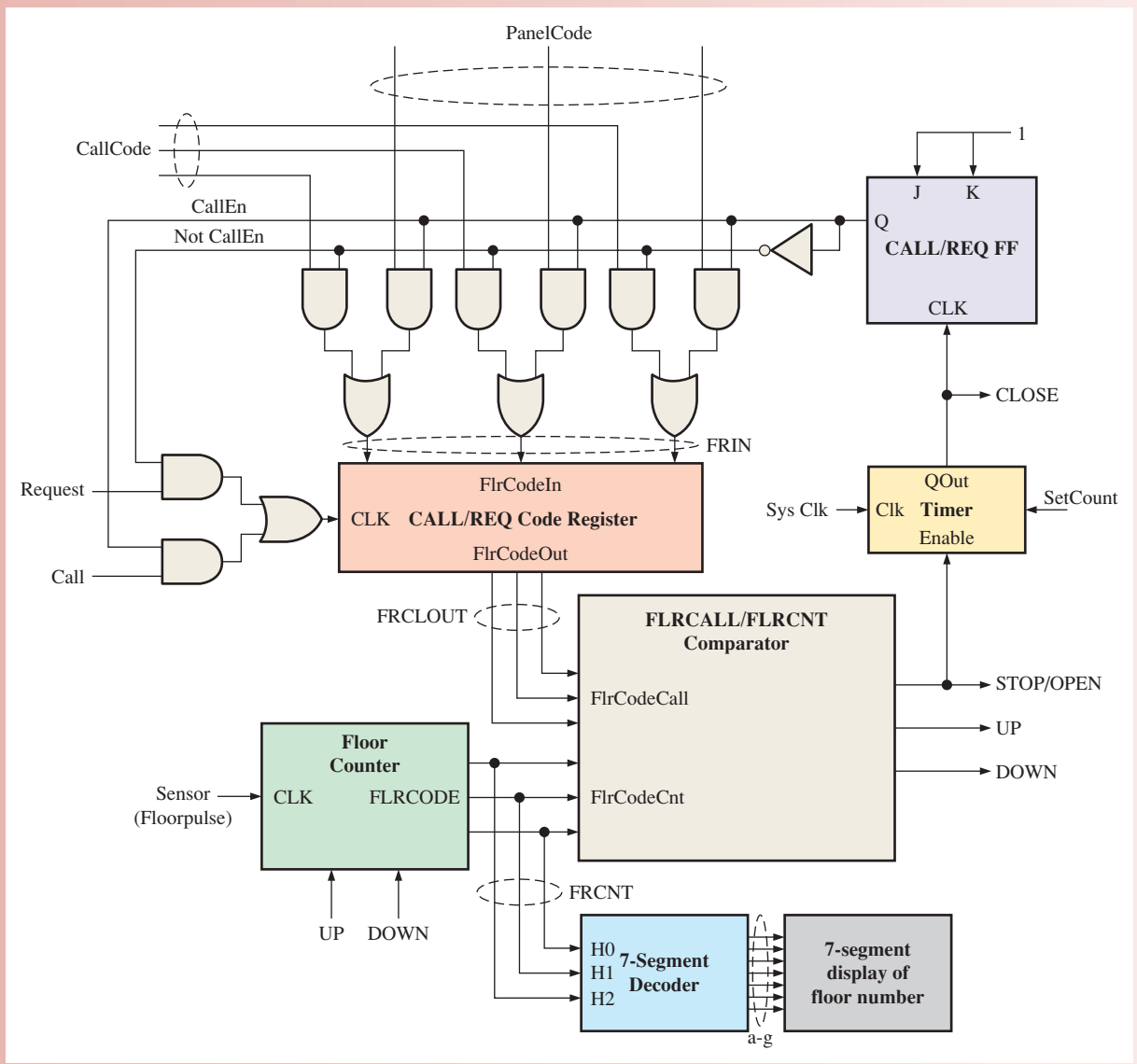


FIGURE 10–62 Programming model of the elevator controller.

The VHDL program code for the elevator controller will include component definitions for the Floor Counter, the FLRCALL/FLRCNT Comparator, the Code Register, the Timer, the Seven-Segment Decoder, and the CALL/REQ Flip-Flop. The VHDL program codes for these six components are as follows. (Blue annotated notes are not part of the program.)

Floor Counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

ieee.numeric_std_all is included to enable casting of unsigned identifier. Unsigned FloorCnt is converted to std_logic_vector.

```
entity FLOORCOUNTER is
   port (UP, DOWN, Sensor: in std_logic;
         FLRCODE: out std_logic_vector(2 downto 0));
end entity FLOORCOUNTER;
```

UP, DOWN: Floor count direction signals
Sensor: Elevator car floor sensor
FLRCODE: 3-digit floor count

```
architecture LogicOperation of FLOORCOUNTER is
signal FloorCnt: unsigned(2 downto 0) := "000";
```

Floor count is initialized to 000.

```
begin
   process(UP, DOWN, Sensor, FloorCnt)
   begin
      FLRCODE <= std_logic_vector(FloorCnt);
      if (Sensor'EVENT and Sensor = '1') then
         if UP = '1' and DOWN = '0' then
            FloorCnt <= FloorCnt + 1;
         elsif Up = '0' and DOWN = '1' then
            FloorCnt <= FloorCnt − 1;
         end if;
      end if;
   end process;
end architecture LogicOperation;
```

Numeric unsigned FloorCnt is converted to std_logic_vector data type and sent to std_logic_vector output FLRCODE.

Sensor event high pulse causes the floor count to increment when UP is set high or decrement by one when DOWN is set low.

FLRCALL/FLRCNT Comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
entity FLRCALLCOMPARATOR is
   port (FlrCodeCall, FlrCodeCnt: in std_logic_vector(2 downto 0);
         UP, DOWN, STOP: inout std_logic;
end entity FLRCALLCOMPARATOR;
```

FlrCodeCall, FlrCodeCnt: Compared values
UP, DOWN, STOP: Output control signals

```
architecture LogicOperation of FLRCALLCOMPARATOR is
begin
   STOP <= '1' when (FlrCodeCall = FlrCodeCnt) else '0';
   UP <= '1' when (FlrCodeCall > FlrCodeCnt) else '0';
   DOWN <= '1' when (FlrCodeCall < FlrCodeCnt) else '0';
end architecture LogicOperation;
```

STOP, UP, and DOWN signals are set or reset based on =, >, and < relational comparisons.

Code Register

```
library ieee;
use ieee.std_logic_1164.all;
entity CODEREGISTER is
   port (Clk: in std_logic;
      FlrCodeIn: in std_logic_vector(0 to 2);
      FlrCodeOut: out std_logic_vector(0 to 2));
end entity CODEREGISTER;

architecture LogicOperation of CODEREGISTER is
begin
   process(Clk)
   begin
      if (Clk 'event and Clk= '1') then
         FlrCodeOut <= FlrCodeIn;
      end if;
   end process;
end architecture LogicOperation;
```

Clk: Clk Pulse input
FlrCodeIn: 3-digit floor panel input
FlrCodeOut: 3-digit floor panel output

Clk event high pulse sends the FlrCodeIn floor number to FlrCodeOut.

Timer

```
library ieee;
use ieee.std_logic_1164.all;
entity Timer is
   port (Enable, Clk: in std_logic;
      SetCount: in integer range 0 to 1023;
      QOut: inout std_logic);
end entity Timer;

architecture TimerBehavior of Timer is
begin
   process(Enable, Clk)
   variable Cnt: integer range 0 to 1023;
   begin
      if (Clk'EVENT and Clk = '1') then
         if Enable = '0' then
            Cnt := 0; QOut <= '0';
         end if;
         if Cnt = SetCount then
            QOut <= '1';
            Cnt := 0;
         else
            Cnt := Cnt + 1;
         end if;
      end if;
   end process;
end architecture TimerBehavior;
```

Enable: Enable timer count input
Clk: Timer clock input
SetCount: Counter set input. Limit to 1023 for ten bits.
QOut: Counter output

Integer variable Cnt range limited to 1023 for ten bits used to count from 0 to terminal count from integer port input SetCount.

When a Clk clock event is HIGH, input Enable is checked for a '0' to clear Cnt and output Qout. If Cnt is equal to SetCount, then output QOut is set to '1' ending the count. If the terminal count in SetCount has not been reached, Cnt is incremented by one and the count process continues.

Seven Segment Decoder

```
library ieee;
use ieee.std_logic_1164.all;
```

a, b, c, d, e, f, g: Seven-segment display element output
H0, H1, H2: Hexadecimal count input

```
entity SevenSegment is
    port (a, b, c, d, e, f, g: out std_logic; H0, H1, H2: inout std_logic);
end entity SevenSegment;

architecture SevenSegmentBehavior of SevenSegment is
begin
```

Seven-segment logic operation

```
    a <= H1 or (H2 and H0) or (not H2 and not H0);
    b <= not H2 or (not H0 and not H1) or (H0 and H1);
    c <= H0 or not H1 or H2;
    d <= (not H0 and not H2) or (not H2 and H1) or
         (H1 and not H0) or (H2 and not H1 and H0);
    e <= (not H0 and not H2) or (H1 and not H0);
    f <= (not H1 and H2) or (not H1 and not H0) or (H2 and not H0);
    g <= (not H2 and H1) or (H1 and not H0) or (H2 and not H1);
end architecture SevenSegmentBehavior;
```

CALL/REQ FF

```
library ieee;
use ieee.std_logic_1164.all;

entity JKFlipFlop is
    port (J,K,Clk: in std_logic; Q: inout std_logic);
end entity JKFlipFlop;

architecture LogicOperation of JKFlipFlop is
signal QNot: std_logic := '1';
begin
    process (J, K, Clk)
    begin
        if (Clk'EVENT and Clk = '1') then
            if J = '1' and K = '0' then
                Q <= '1';
            elsif J = '0' and K = '1' then
                Q <= '0';
            elsif J = '1' and K = '1' then
                Q <= QNot;
            end if;
        end if;
    end process;
    QNot <= not Q;
end architecture LogicOperation;
```

The complete VHDL program code for the elevator controller using the previously defined components is as follows. Comments shown in green preceded by two hyphens are for explanatory purposes and are not recognized by the program for processing purposes.

Elevator Controller

```
library ieee;
use ieee.std_logic_1164.all;

entity ELEVATOR is
    port (CallCode, PanelCode: in std_logic_vector(2 downto 0);
          Call, Request, Sensor, Clk: in std_logic;
          UP, DOWN, STOPOPEN, CLOSE: inout std_logic;
          a, b, c, d, e, f, g: out std_logic);
end entity ELEVATOR;

architecture LogicOperation of ELEVATOR is
component FLOORCOUNTER is
    port (UP, DOWN, Sensor: in std_logic;
          FLRCODE: out std_logic_vector(2 downto 0));
end component FLOORCOUNTER;

component FLRCALLCOMPARATOR is
    port (FlrCodeCall, FlrCodeCnt: in std_logic_vector(2 downto 0);
          UP, DOWN, STOP : inout std_logic);
end component FLRCALLCOMPARATOR;

component CODEREGISTER
    port (Clk: in std_logic;
          FlrCodeIn: in std_logic_vector(0 to 2);
          FlrCodeOut: out std_logic_vector(0 to 2));
end component CODEREGISTER;

component Timer is
    port (Enable, Clk: in std_logic;
          SetCount: in integer range 0 to 1023;
          QOut: inout std_logic);
end component Timer;

component SevenSegment is
    Port (a, b, c, d, e, f, g: out std_logic;
          H0, H1, H2: inout std_logic);
end component SevenSegment;

component JKFlipFlop
    port (J, K, Clk: in std_logic;
          Q: out std_logic);
end component JKFlipFlop;
```

CallCode: Request number from floor
PanelCode: Request number from car
Call: Request pulse for CallCode
Request: Request pulse for PanelCode
Sensor: Floor level pulse input
Clk: Elevator system clock
UP, DOWN: Direction for elevator car
STOPOPEN: Motor stop and door open command
CLOSE: Door close command

Component definition for FLOOR COUNTER

Component definition for FLRCALL/FLRCNT COMPARATOR

Component definition for CODEREGISTER

Component defintion for Timer

Component definition for SevenSegment Decoder

Component definition for CALL/REQ flip-flop

```
-- Signal definitions used to interconnect components and output control signals
signal FRCNT, FRCLOUT, FRIN: std_logic_vector(0 to 2);
signal CallEn: std_logic;
begin
    Gnd <= '0';
    process (CallEn, CallCode, PanelCode) -- Select Floor or Panel call code based on
      begin                                  CALL/REQ
        if (CallEn = '1') then
            FRIN <= CallCode; -- If CALL Enabled, select code from call buttons from floor
        else
            FRIN <= PanelCode; -- If CALL not Enabled, select code from elevator
       end if;                    panel buttons
    end process;
-- Component instantiations
CALLREQ: JKFlipFlop port map(J=>'1', K=>'1', Clk=>Close, Q=> CallEn);
CODEREG: CODEREGISTER port map(Call => (Call and CallEn) or (Request and not
CallEn), FlrCodeIn=> FRIN, FlrCodeOut => FRCLOUT);
FLCLCOMP: FLRCALLCOMPARATOR port map(FlrCodeCall=> FRCL
FlrCodeCnt => FRCNT, Up=>UP, Down=>DOWN, Stop=>STOPOPEN);
FLRCNT: FLOORCOUNTER port map(UP=>UP, DOWN=>DOWN, Sensor=>Sensor,
FLRCODE=>FRCNT);
DISPLAY: SevenSegment port map(a=>a,b=>b,c=>c,d=>d,e=>e,f=>f,g=>g,
H0=>FRCNT(2),H1=>FRCNT(1),H2=>FRCNT(0));
TIMER1: Timer port map (Enable=>STOPOPEN, Clk=> Clk,SetCount=>10,
QOut=>Close);
end architecture LogicOperation;
```

The Programming and PLD Implementation Process

The elevator controller is implemented in a PLD using Altera Quartus II and ModelSim software. The Altera Quartus II software package is an integrated development environment (IDE) supplied by Altera for the creation of HDL applications combined with the ModelSim simulation software. A short summary of the programming process and PLD implementation follows. An expanded description of the elevator controller programming process can be found on the website as well as an Altera Quartus II tutorial. Altera Quartus II is available as a free download from Altera.com.

Project Creation To start the programming process, a project is created. A project allows the IDE to identify a location to store your application and to create self-generated support files needed to organize your application as well as to keep track of project preferences, rules, and definitions.

Project Definition To complete the project, you will need to respond to general questions defining the location of your project, the PLD device to be used, and the primary language. Additional questions will determine how you will simulate and verify your application.

Completed Project Definition With the project definitions completed, the VHDL program source code for the previously defined components and Elevator Controller files are added to your project.

Compiling the Application By compiling the program at this time, part of the input and output identifier information is automatically entered as you are now ready to make pin assignments to your I/O port identifiers. However, the basic design can be simulated before making the pin assignments.

Graphical Waveform Simulation In order to simulate the elevator controller design, first start the ModelSim application. Graphical waveform generation tools allow for the easy creation of stimulus waveforms. Graphical waveforms are created to provide the input stimulus to test the elevator controller application. Inputs call, request, callcode, panelcode, sensor, and clk will be created using graphical tools. Output identifiers *up, down, stopopen, close,* and seven-segment outputs *a* through *g* require no input stimulus.

Pin Assignments A pin assignment editor is used to associate an I/O port identifier with an external pin. Many newer pin editors utilize drag-and-drop features to allow the user to select an identifier with the mouse, then drag and drop to a graphic representation of the target device. Pin assignments can also be accomplished using traditional text entry.

Device Programming With the pins selected and saved, the project is recompiled once again, generating the output file to be loaded on the target device (PLD). The second compiling operation associates the selected pin to the program identifier. In order to program the target device, the project board on which it is mounted must be connected to the programming computer according to the project board manufacturer's instructions. The target device is typically JTAG compliant and connected through a USB port. Other JTAG compliant target boards may use other inputs such as Ethernet, serial, parallel, or FireWire as described by the manufacturer.

Downloading to the PLD With the simulation, pin assignment, and recompiling complete, it is time to download the application to the development environment (project board with PLD).

Hardware Testing With the project loaded, the application can be tested against actual hardware.

Putting Your Knowledge to Work

Modify the elevator controller program for a building with ten floors rather than eight.

SUMMARY

- A PAL is a one-time programmable (OTP) SPLD consisting of a programmable array of AND gates that connects to a fixed array of OR gates.
- The PAL structure allows any sum-of-products (SOP) logic expression with a defined number of variables to be implemented.
- The GAL is essentially a PAL that can be reprogrammed.
- In a PAL or GAL, a macrocell generally consists of one OR gate and some associated output logic.
- A CPLD is a complex programmable logic device that consists basically of multiple SPLD arrays with programmable interconnections.
- Each SPLD array in a CPLD is called a logic array block (LAB).
- A macrocell can be configured for either of two modes: the combinational mode or the registered mode.