Altera Digital Logic Lab Exercises

Introduction

This document presents a set of ten laboratory exercises for use on the Altera DE2 Development and Education board. These exercises are intended for use in a first course on Digital Logic Design, which is included as part of the curriculum in most Computer Engineering, Electrical Engineering, and Computer Science programs.

Circuits are designed for implementation on the DE2 board by using Altera's state-of-the-art *Quartus II* CAD system. To teach students how to use this software, which is provided at no charge for educational use, we have developed a set of step-by-step tutorials that are available in the *Educational Materials* section of Altera's *University Program* web site. There is a tutorial that introduces the use of the DE2 board, and a set of tutorials that show how to develop circuits with the Quartus II software by using VHDL as the design entry method. Students should work through these tutorials as a preparation for the lab exercises.

There is also a basic tutorial on using schematic capture in the Quartus II software. Although we do not provide a version of the lab exercises that use schematic capture, a course Instructor could easily adapt the material for this purpose if desired.

Overview of Lab Exercises

The ten exercises begin with fundamental concepts and perform simple operations on the DE2 board, like using switches and controlling LEDs and 7-segment displays. These exercises assume that students are just beginning to learn about digital logic concepts, and require solutions that use simple logic expressions. Subsequent exercises progress to more advanced topics such as arithmetic circuits, flip-flops, counters, state machines, memory devices, data paths, and simple processors. Instructors of courses may choose to adopt the entire sequence of exercises, only selected exercises, or just parts of some exercises. We have tried to make the material as modular as possible so that instructors can combine these exercises with their own teaching material.

Each exercise consists of multiple parts. In most cases the solution required for the early parts can be reused in a modular fashion for later parts. Also, the solutions produced for early exercises are often reusable for parts of more advanced exercises. Our basic approach is to encourage students to develop their circuits in small increments and to build larger circuits in a modular, hierarchical fashion. As an aid for the Instructor, this document provides complete solutions in VHDL code for all ten lab exercises.

Obtaining the Source Code Files

To make it easier for Instructors to modify the lab exercises as needed, we provide the original source files that were used to create this PDF document. The lab exercises are written in ASCII text files that include formatting information for the LaTex word processing system. Instructors who are not familiar with LaTex may choose to import the text into some other word processing system of their choice. The figures used in the exercises were created using Adobe FrameMaker. They are provided to course Instructors in both the FrameMaker format as well as in Adobe PDF format, which can be edited using programs other than FrameMaker.

We also provide the VHDL source files for all of the suggested solutions, and the Quartus II project files that are needed to compile the code for implementation on the DE2 board.

To obtain the source files for both the lab exercises and solutions, go to the *Educational Materials* section of Altera's *University Program* web site at *www.altera.com*. You will find instructions for obtaining this information, which is protected from access by students. Please do not freely distribute the suggested solutions on the Internet, and protect this material as you would other similar educational materials that are assigned by Instructors to students as a part of their course grade.

Laboratory Exercise 1

Switches, Lights, and Multiplexers

The purpose of this exercise is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices. We will use the switches SW_{17-0} on the DE2 board as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

Part I

The DE2 board provides 18 toggle switches, called SW_{17-0}, that can be used as inputs to a circuit, and 18 red lights, called $LEDR_{17-0}$, that can be used to display output values. Figure 1 shows a simple VHDL entity that uses these switches and shows their states on the LEDs. Since there are 18 switches and lights it is convenient to represent them as arrays in the VHDL code, as shown. We have used a single assignment statement for all 18 *LEDR* outputs, which is equivalent to the individual assignments

```
LEDR(17) <= SW(17);
LEDR(16) <= SW(16);
...
LEDR(0) <= SW(0);
```

The DE2 board has hardwired connections between its FPGA chip and the switches and lights. To use SW_{17-0} and $LEDR_{17-0}$ it is necessary to include in your Quartus II project the correct pin assignments, which are given in the *DE2 User Manual*. For example, the manual specifies that SW_0 is connected to the FPGA pin *N25* and $LEDR_0$ is connected to pin *AE23*. A good way to make the required pin assignments is to import into the Quartus II software the file called *DE2_pin_assignments.csv*, which is provided on the *DE2 System CD* and in the University Program section of Altera's web site. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction using VHDL Design*, which is also available from Altera.

It is important to realize that the pin assignments in the *DE2_pin_assignments.csv* file are useful only if the pin names given in the file are exactly the same as the port names used in your VHDL entity. The file uses the names *SW*[0] ... *SW*[17] and *LEDR*[0] ... *LEDR*[17] for the switches and lights, which is the reason we used these names in Figure 1 (note that the Quartus II software uses [ ] square brackets for array elements, while the VHDL syntax uses ( ) round brackets).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Simple module that connects the SW switches to the LEDR lights
ENTITY part1 IS
   PORT ( SW   : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
          LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0)); -- red LEDs
END part1;

ARCHITECTURE Behavior OF part1 IS
BEGIN
   LEDR <= SW;
END Behavior
```

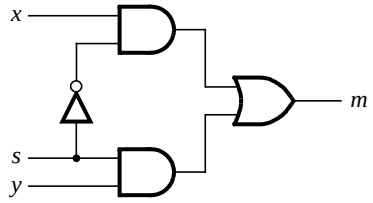
Figure 1. VHDL code that uses the DE2 board switches and lights.

Perform the following steps to implement a circuit corresponding to the code in Figure 1 on the DE2 board.

1. Create a new Quartus II project for your circuit. Select Cyclone II EP2C35F672C6 as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in Figure 1 and include it in your project.
3. Include in your project the required pin assignments for the DE2 board, as discussed above. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part II

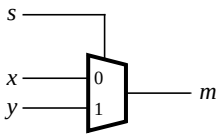
Figure 2a shows a sum-of-products circuit that implements a 2-to-1 *multiplexer* with a select input s. If $s = 0$ the multiplexer's output m is equal to the input x, and if $s = 1$ the output is equal to y. Part b of the figure gives a truth table for this multiplexer, and part c shows its circuit symbol.



a) Circuit

s	m
0	x
1	y

b) Truth table



c) Symbol

Figure 2. A 2-to-1 multiplexer.

The multiplexer can be described by the following VHDL statement:

```
m <= (NOT (s) AND x) OR (s AND y);
```

You are to write a VHDL entity that includes eight assignment statements like the one shown above to describe the circuit given in Figure 3a. This circuit has two eight-bit inputs, X and Y, and produces the eight-bit output M. If $s = 0$ then $M = X$, while if $s = 1$ then $M = Y$. We refer to this circuit as an eight-bit wide 2-to-1 multiplexer. It has the circuit symbol shown in Figure 3b, in which X, Y, and M are depicted as eight-bit wires. Perform the steps shown below.

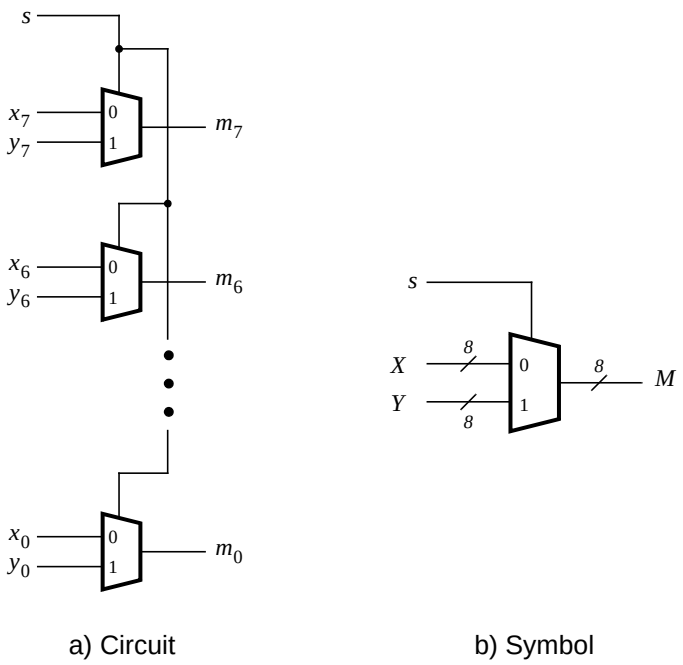


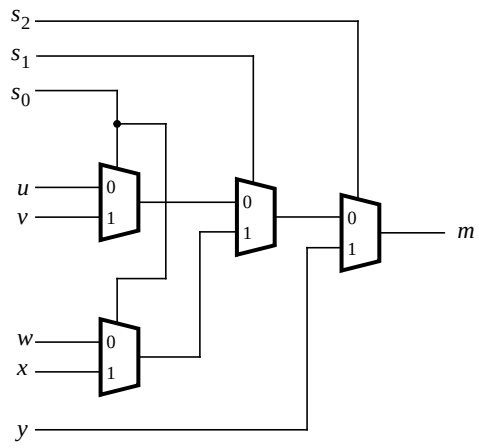
Figure 3. An eight-bit wide 2-to-1 multiplexer.

1. Create a new Quartus II project for your circuit.
2. Include your VHDL file for the eight-bit wide 2-to-1 multiplexer in your project. Use switch SW_{17} on the DE2 board as the s input, switches SW_{7-0} as the X input and SW_{15-8} as the Y input. Connect the *SW* switches to the red lights *LEDR* and connect the output M to the green lights $LEDG_{7-0}$.
3. Include in your project the required pin assignments for the DE2 board. As discussed in Part I, these assignments ensure that the input ports of your VHDL code will use the pins on the Cyclone II FPGA that are connected to the *SW* switches, and the output ports of your VHDL code will use the FPGA pins connected to the *LEDR* and *LEDG* lights.
4. Compile the project.
5. Download the compiled circuit into the FPGA chip. Test the functionality of the eight-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.

Part III

In Figure 2 we showed a 2-to-1 multiplexer that selects between the two inputs *x* and *y*. For this part consider a circuit in which the output *m* has to be selected from five inputs u, v, w, x, and y. Part a of Figure 4 shows how we can build the required 5-to-1 multiplexer by using four 2-to-1 multiplexers. The circuit uses a 3-bit select input $s_2s_1s_0$ and implements the truth table shown in Figure 4b. A circuit symbol for this multiplexer is given in part c of the figure.

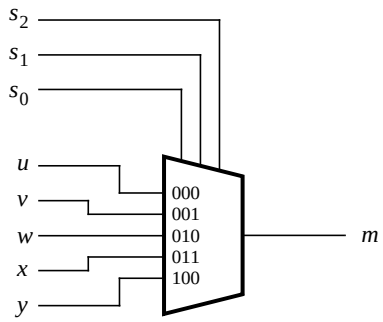
Recall from Figure 3 that an eight-bit wide 2-to-1 multiplexer can be built by using eight instances of a 2-to-1 multiplexer. Figure 5 applies this concept to define a three-bit wide 5-to-1 multiplexer. It contains three instances of the circuit in Figure 4a.



a) Circuit

s_2	s_1	s_0	m
0	0	0	u
0	0	1	v
0	1	0	w
0	1	1	x
1	0	0	y
1	0	1	y
1	1	0	y
1	1	1	y

b) Truth table



c) Symbol

Figure 4. A 5-to-1 multiplexer.

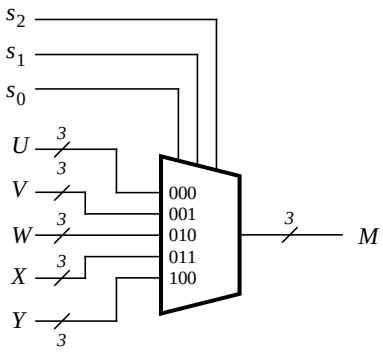


Figure 5. A three-bit wide 5-to-1 multiplexer.

Perform the following steps to implement the three-bit wide 5-to-1 multiplexer.

1. Create a new Quartus II project for your circuit.

2. Create a VHDL entity for the three-bit wide 5-to-1 multiplexer. Connect its select inputs to switches SW_{17-15}, and use the remaining 15 switches SW_{14-0} to provide the five 3-bit inputs U to Y. Connect the *SW* switches to the red lights *LEDR* and connect the output M to the green lights $LEDG_{2-0}$.

3. Include in your project the required pin assignments for the DE2 board. Compile the project.

4. Download the compiled circuit into the FPGA chip. Test the functionality of the three-bit wide 5-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs U to Y can be properly selected as the output M.

Part IV

Figure 6 shows a *7-segment* decoder module that has the three-bit input $c_2c_1c_0$. This decoder produces seven outputs that are used to display a character on a 7-segment display. Table 1 lists the characters that should be displayed for each valuation of $c_2c_1c_0$. To keep the design simple, only four characters are included in the table (plus the 'blank' character, which is selected for codes $100 - 111$).

The seven segments in the display are identified by the indices 0 to 6 shown in the figure. Each segment is illuminated by driving it to the logic value 0. You are to write a VHDL entity that implements logic functions that represent circuits needed to activate each of the seven segments. Use only simple VHDL assignment statements in your code to specify each logic function using a Boolean expression.

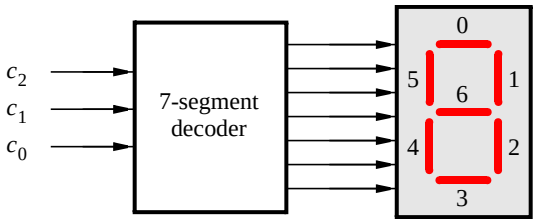


Figure 6. A 7-segment decoder.

$c_2c_1c_0$	Character
000	H
001	E
010	L
011	O
100	
101	
110	
111	

Table 1. Character codes.

Perform the following steps:

1. Create a new Quartus II project for your circuit.

2. Create a VHDL entity for the 7-segment decoder. Connect the $c_2c_1c_0$ inputs to switches SW_{2-0}, and connect the outputs of the decoder to the *HEX0* display on the DE2 board. The segments in this display are called $HEX0_0$, $HEX0_1$, . . ., $HEX0_6$, corresponding to Figure 6. You should declare the 7-bit port

   HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6);

   in your VHDL code so that the names of these outputs match the corresponding names in the *DE2 User Manual* and the *DE2_pin_assignments.csv* file.

3. After making the required DE2 board pin assignments, compile the project.

4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the SW_{2-0} switches and observing the 7-segment display.

Part V

Consider the circuit shown in Figure 7. It uses a three-bit wide 5-to-1 multiplexer to enable the selection of five characters that are displayed on a 7-segment display. Using the 7-segment decoder from Part IV this circuit can display any of the characters H, E, L, O, and 'blank'. The character codes are set according to Table 1 by using the switches SW_{14-0}, and a specific character is selected for display by setting the switches SW_{17-15}.

An outline of the VHDL code that represents this circuit is provided in Figure 8. Note that we have used the circuits from Parts III and IV as subcircuits in this code. You are to extend the code in Figure 8 so that it uses five 7-segment displays rather than just one. You will need to use five instances of each of the subcircuits. The purpose of your circuit is to display any word on the five displays that is composed of the characters in Table 1, and be able to rotate this word in a circular fashion across the displays when the switches SW_{17-15} are toggled. As an example, if the displayed word is HELLO, then your circuit should produce the output patterns illustrated in Table 2.

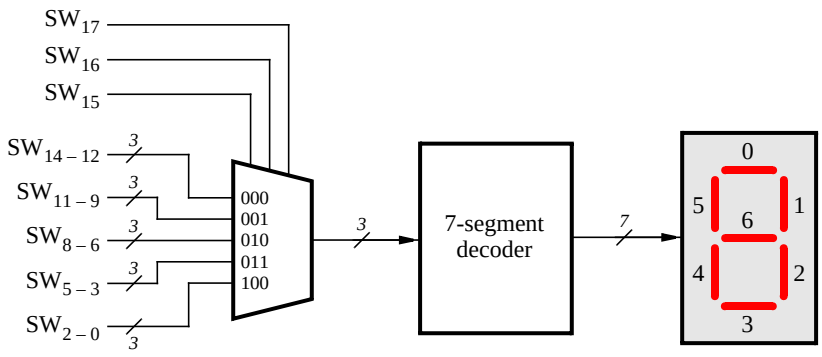


Figure 7. A circuit that can select and display one of five characters.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part5 IS
   PORT ( SW     : IN     STD_LOGIC_VECTOR(17 DOWNTO 0);
          HEX0 : OUT    STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT mux_3bit_5to1
      PORT ( S, U, V, W, X, Y     : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
             M                    : OUT    STD_LOGIC_VECTOR(2 DOWNTO 0));
   END COMPONENT;
   COMPONENT char_7seg
      PORT ( C           : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
             Display     : OUT    STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL M : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   M0: mux_3bit_5to1 PORT MAP (SW(17 DOWNTO 15), SW(14 DOWNTO 12), SW(11 DOWNTO 9),
      SW(8 DOWNTO 6), SW(5 DOWNTO 3), SW(2 DOWNTO 0), M);
   H0: char_7seg PORT MAP (M, HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- implements a 3-bit wide 5-to-1 multiplexer
ENTITY mux_3bit_5to1 IS
   PORT ( S, U, V, W, X, Y     : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
          M                    : OUT    STD_LOGIC_VECTOR(2 DOWNTO 0));
END mux_3bit_5to1;

ARCHITECTURE Behavior OF mux_3bit_5to1 IS

   . . . code not shown

END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY char_7seg IS
   PORT ( C           : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
          Display     : OUT    STD_LOGIC_VECTOR(0 TO 6));
END char_7seg;

ARCHITECTURE Behavior OF char_7seg IS

   . . . code not shown

END Behavior;
```

Figure 8. VHDL code for the circuit in Figure 7.

SW_{17} SW_{16} SW_{15}	Character pattern				
000	H	E	L	L	O
001	E	L	L	O	H
010	L	L	O	H	E
011	L	O	H	E	L
100	O	H	E	L	L

Table 2. Rotating the word HELLO on five displays.

Perform the following steps.

1. Create a new Quartus II project for your circuit.
2. Include your VHDL entity in the Quartus II project. Connect the switches SW_{17-15} to the select inputs of each of the five instances of the three-bit wide 5-to-1 multiplexers. Also connect SW_{14-0} to each instance of the multiplexers as required to produce the patterns of characters shown in Table 2. Connect the outputs of the five multiplexers to the 7-segment displays *HEX4*, *HEX3*, *HEX2*, *HEX1*, and *HEX0*.
3. Include the required pin assignments for the DE2 board for all switches, LEDs, and 7-segment displays. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by setting the proper character codes on the switches SW_{14-0} and then toggling SW_{17-15} to observe the rotation of the characters.

Part VI

Extend your design from Part V so that is uses all eight 7-segment displays on the DE2 board. Your circuit should be able to display words with five (or fewer) characters on the eight displays, and rotate the displayed word when the switches SW_{17-15} are toggled. If the displayed word is HELLO, then your circuit should produce the patterns shown in Table 3.

SW_{17} SW_{16} SW_{15}	Character pattern							
000				H	E	L	L	O
001			H	E	L	L	O	
010		H	E	L	L	O		
011	H	E	L	L	O			
100	E	L	L	O				H
101	L	L	O				H	E
110	L	O				H	E	L
111	O				H	E	L	L

Table 3. Rotating the word HELLO on eight displays.

Perform the following steps:

1. Create a new Quartus II project for your circuit and select as the target chip the Cyclone II EP2C35F672C6.

2. Include your VHDL entity in the Quartus II project. Connect the switches SW_{17-15} to the select inputs of each instance of the multiplexers in your circuit. Also connect SW_{14-0} to each instance of the multiplexers as required to produce the patterns of characters shown in Table 3. (Hint: for some inputs of the multiplexers you will want to select the 'blank' character.) Connect the outputs of your multiplexers to the 7-segment displays *HEX7*, . . ., *HEX0*.

3. Include the required pin assignments for the DE2 board for all switches, LEDs, and 7-segment displays. Compile the project.

4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by setting the proper character codes on the switches SW_{14-0} and then toggling SW_{17-15} to observe the rotation of the characters.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Simple module that connects the SW switches to the LEDR lights
ENTITY part1 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0));  -- red LEDs
END part1;

ARCHITECTURE Structure OF part1 IS
BEGIN
   LEDR <= SW;
END Structure;
```

```
-- Implements eight 2-to-1 multiplexers.
-- inputs:  SW7-0 represent the 8-bit input X, and SW15-8 represent Y
--          SW17 selects either X or Y to drive the output LEDs
-- outputs: LEDR17-0 show the states of the switches
--          LEDG7-0 shows the outputs of the multiplexers
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Simple module that connects the SW switches to the LEDR lights
ENTITY part2 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);  -- red LEDs
            LEDG  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  -- green LEDs
END part2;

ARCHITECTURE Structure OF part2 IS
   SIGNAL Sel : STD_LOGIC;
   SIGNAL X, Y, M : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   LEDR <= SW;
   X <= SW(7 DOWNTO 0);
   Y <= SW(15 DOWNTO 8);
   Sel <= SW(17);

   M(0) <= (NOT(Sel) AND X(0)) OR (Sel AND Y(0));
   M(1) <= (NOT(Sel) AND X(1)) OR (Sel AND Y(1));
   M(2) <= (NOT(Sel) AND X(2)) OR (Sel AND Y(2));
   M(3) <= (NOT(Sel) AND X(3)) OR (Sel AND Y(3));
   M(4) <= (NOT(Sel) AND X(4)) OR (Sel AND Y(4));
   M(5) <= (NOT(Sel) AND X(5)) OR (Sel AND Y(5));
   M(6) <= (NOT(Sel) AND X(6)) OR (Sel AND Y(6));
   M(7) <= (NOT(Sel) AND X(7)) OR (Sel AND Y(7));
   LEDG(7 DOWNTO 0) <= M;
END Structure;
```

```
-- Implements a 3-bit wide 5-to-1 multiplexer.
-- inputs:  SW14-0 represent data in 5 groups, U-Y
--          SW17-15 selects one group from U to Y
-- outputs: LEDR17-0 show the states of the switches
--          LEDG2-0 displays the selected group
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Simple module that connects the SW switches to the LEDR lights
ENTITY part3 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);  -- red LEDs
            LEDG  : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));  -- green LEDs
END part3;

ARCHITECTURE Structure OF part3 IS
   SIGNAL m_0, m_1, m_2 : STD_LOGIC_VECTOR(1 TO 3);
      -- m_0 is used for 3 intermediate
      -- multiplexers to produce the
      -- 5-to-1 multiplexer M(0), m_1 is
      -- for M(1), and m_2 is for M(2)
   SIGNAL S, U, V, W, X, Y, M : STD_LOGIC_VECTOR(2 DOWNTO 0);
      -- M is the 3-bit 5-to-1 multiplexer
BEGIN
   S(2 DOWNTO 0) <= SW(17 DOWNTO 15);
   U <= SW(2 DOWNTO 0);
   V <= SW(5 DOWNTO 3);
   W <= SW(8 DOWNTO 6);
   X <= SW(11 DOWNTO 9);
   Y <= SW(14 DOWNTO 12);

   LEDR <= SW;

   -- 5-to-1 multiplexer for bit 0
   m_0(1) <= (NOT(S(0)) AND U(0)) OR (S(0) AND V(0));
   m_0(2) <= (NOT(S(0)) AND W(0)) OR (S(0) AND X(0));
   m_0(3) <= (NOT(S(1)) AND m_0(1)) OR (S(1) AND m_0(2));
   M(0) <= (NOT(S(2)) AND m_0(3)) OR (S(2) AND Y(0)); -- 5-to-1 multiplexer output

   -- 5-to-1 multiplexer for bit 1
   m_1(1) <= (NOT(S(0)) AND U(1)) OR (S(0) AND V(1));
   m_1(2) <= (NOT(S(0)) AND W(1)) OR (S(0) AND X(1));
   m_1(3) <= (NOT(S(1)) AND m_1(1)) OR (S(1) AND m_1(2));
   M(1) <= (NOT(S(2)) AND m_1(3)) OR (S(2) AND Y(1)); -- 5-to-1 multiplexer output

   -- 5-to-1 multiplexer for bit 2
   m_2(1) <= (NOT(S(0)) AND U(2)) OR (S(0) AND V(2));
   m_2(2) <= (NOT(S(0)) AND W(2)) OR (S(0) AND X(2));
   m_2(3) <= (NOT(S(1)) AND m_2(1)) OR (S(1) AND m_2(2));
   M(2) <= (NOT(S(2)) AND m_2(3)) OR (S(2) AND Y(2)); -- 5-to-1 multiplexer output

   LEDG(2 DOWNTO 0) <= M;
END Structure;
```

```
-- Implements a circuit that can display five characters on a 7-segment
-- display.
-- inputs:  SW2-0 selects the letter to display. The characters are:
--    SW 2 1 0    Char
--    ----------------
--       0 0 0    'H'
--       0 0 1    'E'
--       0 1 0    'L'
--       0 1 1    'O'
--       1 0 0    ' ' Blank
--       1 0 1    ' ' Blank
--       1 1 0    ' ' Blank
--       1 1 1    ' ' Blank
--
-- outputs: LEDR2-0 show the states of the switches
--          HEX0 displays the selected character
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part4 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0); -- toggle switches
            LEDR  : OUT STD_LOGIC_VECTOR(2 DOWNTO 0); -- red LEDs
            HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));    -- 7-seg display
END part4;

ARCHITECTURE Structure OF part4 IS
   SIGNAL C : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   LEDR <= SW;
   C(2 DOWNTO 0) <= SW(2 DOWNTO 0);
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   -- the following equations describe display functions in (inverted)
   -- cannonical SOP form
   HEX0(0) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(1) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(2) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(3) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(4) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(5) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   HEX0(6) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) );
END Structure;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Implements a circuit that can display different 5-letter words on five 7-segment
-- displays. The character selected for each display is chosen by
-- a multiplexer, and these multiplexers are connected to the characters
-- in a way that allows a word to be rotated across the displays from
-- right-to-left as the multiplexer select lines are changed through the
-- sequence 000, 001, 010, 011, 100, 000, etc. Using the four characters H,
-- E, L, O, the displays can scroll any 5-letter word using these letters, such
-- as "HELLO", as follows:
--
-- SW 17 16 15    Displayed characters
--     0  0  0    HELLO
--     0  0  1    ELLOH
--     0  1  0    LLOHE
--     0  1  1    LOHEL
--     1  0  0    OHELL
--
-- inputs: SW17-15 provide the multiplexer select lines
--         SW14-0 provide five 3-bit codes used to select characters
-- outputs: LEDR shows the states of the switches
--          HEX4 - HEX0 displays the characters (HEX7 - HEX5 are set to "blank")
ENTITY part5 IS
   PORT (   SW                      : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDR                    : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Structure OF part5 IS
   COMPONENT mux_3bit_5to1
      PORT (   S, U, V, W, X, Y  : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               M                 : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
   END COMPONENT;
   COMPONENT char_7seg
      PORT (   C        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               Display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL H4_Ch, H3_Ch, H2_Ch, H1_Ch, H0_Ch : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   LEDR <= SW;

   Ch_Sel <= SW(17 DOWNTO 15);
   Ch1 <= SW(14 DOWNTO 12);
   Ch2 <= SW(11 DOWNTO 9);
   Ch3 <= SW(8 DOWNTO 6);
   Ch4 <= SW(5 DOWNTO 3);
   Ch5 <= SW(2 DOWNTO 0);
   Blank <= "111";   -- used to blank a 7-seg display (see module char_7seg)

   -- instantiate mux_3bit_5to1 (S, U, V, W, X, Y, M);
   M4: mux_3bit_5to1 PORT MAP (Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, H4_Ch);
   M3: mux_3bit_5to1 PORT MAP (Ch_Sel, Ch2, Ch3, Ch4, Ch5, Ch1, H3_Ch);
   M2: mux_3bit_5to1 PORT MAP (Ch_Sel, Ch3, Ch4, Ch5, Ch1, Ch2, H2_Ch);
   M1: mux_3bit_5to1 PORT MAP (Ch_Sel, Ch4, Ch5, Ch1, Ch2, Ch3, H1_Ch);
   M0: mux_3bit_5to1 PORT MAP (Ch_Sel, Ch5, Ch1, Ch2, Ch3, Ch4, H0_Ch);

   -- instantiate char_7seg (C, Display);
   H7: char_7seg PORT MAP (Blank, HEX7);
   H6: char_7seg PORT MAP (Blank, HEX6);
   H5: char_7seg PORT MAP (Blank, HEX5);
   H4: char_7seg PORT MAP (H4_Ch, HEX4);
```

```
   H3: char_7seg PORT MAP (H3_Ch, HEX3);
   H2: char_7seg PORT MAP (H2_Ch, HEX2);
   H1: char_7seg PORT MAP (H1_Ch, HEX1);
   H0: char_7seg PORT MAP (H0_Ch, HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Implements a 3-bit wide 5-to-1 multiplexer
ENTITY mux_3bit_5to1 IS
   PORT (   S, U, V, W, X, Y  : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            M                 : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END mux_3bit_5to1;

ARCHITECTURE Behavior OF mux_3bit_5to1 IS
   SIGNAL m_0, m_1, m_2 : STD_LOGIC_VECTOR(1 TO 3); -- intermediate multiplexers
BEGIN
   -- 5-to-1 multiplexer for bit 0
   m_0(1) <= (NOT(S(0)) AND U(0)) OR (S(0) AND V(0));
   m_0(2) <= (NOT(S(0)) AND W(0)) OR (S(0) AND X(0));
   m_0(3) <= (NOT(S(1)) AND m_0(1)) OR (S(1) AND m_0(2));
   M(0) <= (NOT(S(2)) AND m_0(3)) OR (S(2) AND Y(0)); -- 5-to-1 multiplexer output

   -- 5-to-1 multiplexer for bit 1
   m_1(1) <= (NOT(S(0)) AND U(1)) OR (S(0) AND V(1));
   m_1(2) <= (NOT(S(0)) AND W(1)) OR (S(0) AND X(1));
   m_1(3) <= (NOT(S(1)) AND m_1(1)) OR (S(1) AND m_1(2));
   M(1) <= (NOT(S(2)) AND m_1(3)) OR (S(2) AND Y(1)); -- 5-to-1 multiplexer output

   -- 5-to-1 multiplexer for bit 2
   m_2(1) <= (NOT(S(0)) AND U(2)) OR (S(0) AND V(2));
   m_2(2) <= (NOT(S(0)) AND W(2)) OR (S(0) AND X(2));
   m_2(3) <= (NOT(S(1)) AND m_2(1)) OR (S(1) AND m_2(2));
   M(2) <= (NOT(S(2)) AND m_2(3)) OR (S(2) AND Y(2)); -- 5-to-1 multiplexer output
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Converts 3-bit input code on C2-0 into 7-bit code that produces
-- a character on a 7-segment display. The conversion is defined by:
--     C 2 1 0    Char
--    ----------------
--       0 0 0    'H'
--       0 0 1    'E'
--       0 1 0    'L'
--       0 1 1    'O'
--       1 0 0    ' ' Blank
--       1 0 1    ' ' Blank
--       1 1 0    ' ' Blank
--       1 1 1    ' ' Blank
--
--    Codes 100, 101, 110 are not used
--
ENTITY char_7seg IS
   PORT (   C        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            Display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END char_7seg;
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
```

```
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
ARCHITECTURE Behavior OF char_7seg IS
BEGIN
   -- the following equations describe display functions in (inverted) cannonical SOP
form
   Display(0) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(1) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(2) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(3) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(4) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   Display(5) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   Display(6) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) );
END Behavior;
```

```
-- Implements a circuit that can display different 5-letter words on the eight
-- 7-segment displays. The character selected for each display is chosen by
-- a multiplexer, and these multiplexers are connected to the characters
-- in a way that allows a word to be scrolled across the displays from
-- right-to-left as the multiplexer select lines are changed through the
-- sequence 000, 001, ..., 111, 000, 001, etc. Using the four characters H,
-- E, L, O, -, where - means "blank". the displays can scroll any 5-letter word using
-- these letters, such as "HELLO---", as follows:
--
-- SW 17 16 15    Displayed characters
--     0  0  0    ---HELLO
--     0  0  1    --HELLO-
--     0  1  0    -HELLO--
--     0  1  1    HELLO---
--     1  0  0    ELLO---H
--     1  0  1    LLO---HE
--     1  1  0    LO---HEL
--     1  1  1    O---HELL
--
-- inputs: SW17-15 provide the multiplexer select lines
--         SW14-0 provide five different codes used to select characters
-- outputs: LEDR shows the states of the switches
--          HEX7 - HEX0 displays the characters
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part6 IS
   PORT (   SW                      : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDR                    : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part6;

ARCHITECTURE Structure OF part6 IS
   COMPONENT mux_3bit_8to1
      PORT (   S, G1, G2, G3, G4, G5, G6, G7, G8   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               M                                   : OUT STD_LOGIC_VECTOR(2 DOWNTO 0))
;
   END COMPONENT;
   COMPONENT char_7seg
      PORT (   C        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               Display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL H7_Ch, H6_Ch, H5_Ch, H4_Ch : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL H3_Ch, H2_Ch, H1_Ch, H0_Ch : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   LEDR <= SW;

   Ch_Sel <= SW(17 DOWNTO 15);
   Ch1 <= SW(14 DOWNTO 12);
   Ch2 <= SW(11 DOWNTO 9);
   Ch3 <= SW(8 DOWNTO 6);
   Ch4 <= SW(5 DOWNTO 3);
   Ch5 <= SW(2 DOWNTO 0);
   Blank <= "111";   -- used to blank a 7-seg display (see module char_7seg)

   -- instantiate module mux_3bit_8to1 (S, G1, G2, G3, G4, G5, G6, G7, G8, M) to
   -- create the multiplexer for each hex display
   M7: mux_3bit_8to1 PORT MAP (Ch_Sel, Blank, Blank, Blank, Ch1, Ch2, Ch3,
      Ch4, Ch5, H7_Ch);
   M6: mux_3bit_8to1 PORT MAP (Ch_Sel, Blank, Blank, Ch1, Ch2, Ch3, Ch4,
      Ch5, Blank, H6_Ch);
   M5: mux_3bit_8to1 PORT MAP (Ch_Sel, Blank, Ch1, Ch2, Ch3, Ch4, Ch5,
```

```
      Blank, Blank, H5_Ch);
   M4: mux_3bit_8to1 PORT MAP (Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank,
      Blank, Blank, H4_Ch);
   M3: mux_3bit_8to1 PORT MAP (Ch_Sel, Ch2, Ch3, Ch4, Ch5, Blank, Blank,
      Blank, Ch1, H3_Ch);
   M2: mux_3bit_8to1 PORT MAP (Ch_Sel, Ch3, Ch4, Ch5, Blank, Blank, Blank,
      Ch1, Ch2, H2_Ch);
   M1: mux_3bit_8to1 PORT MAP (Ch_Sel, Ch4, Ch5, Blank, Blank, Blank, Ch1,
      Ch2, Ch3, H1_Ch);
   M0: mux_3bit_8to1 PORT MAP (Ch_Sel, Ch5, Blank, Blank, Blank, Ch1, Ch2,
      Ch3, Ch4, H0_Ch);

   -- instantiate module char_7seg (C, Display) to drive the hex displays
   H7: char_7seg PORT MAP (H7_Ch, HEX7);
   H6: char_7seg PORT MAP (H6_Ch, HEX6);
   H5: char_7seg PORT MAP (H5_Ch, HEX5);
   H4: char_7seg PORT MAP (H4_Ch, HEX4);
   H3: char_7seg PORT MAP (H3_Ch, HEX3);
   H2: char_7seg PORT MAP (H2_Ch, HEX2);
   H1: char_7seg PORT MAP (H1_Ch, HEX1);
   H0: char_7seg PORT MAP (H0_Ch, HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- implements a 3-bit wide 8-to-1 multiplexer
ENTITY mux_3bit_8to1 IS
   PORT (   S, G1, G2, G3, G4, G5, G6, G7, G8   : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
            M                                   : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END mux_3bit_8to1;

ARCHITECTURE Behavior OF mux_3bit_8to1 IS
   SIGNAL m_0, m_1, m_2 : STD_LOGIC_VECTOR(1 TO 6); -- intermediate multiplexers
BEGIN
   -- 8-to-1 multiplexer for bit 0
   m_0(1) <= (NOT(S(0)) AND G1(0)) OR (S(0) AND G2(0));
   m_0(2) <= (NOT(S(0)) AND G3(0)) OR (S(0) AND G4(0));
   m_0(3) <= (NOT(S(0)) AND G5(0)) OR (S(0) AND G6(0));
   m_0(4) <= (NOT(S(0)) AND G7(0)) OR (S(0) AND G8(0));
   m_0(5) <= (NOT(S(1)) AND m_0(1)) OR (S(1) AND m_0(2));
   m_0(6) <= (NOT(S(1)) AND m_0(3)) OR (S(1) AND m_0(4));
   M(0) <= (NOT(S(2)) AND m_0(5)) OR (S(2) AND m_0(6));

   -- 8-to-1 multiplexer for bit 1
   m_1(1) <= (NOT(S(0)) AND G1(1)) OR (S(0) AND G2(1));
   m_1(2) <= (NOT(S(0)) AND G3(1)) OR (S(0) AND G4(1));
   m_1(3) <= (NOT(S(0)) AND G5(1)) OR (S(0) AND G6(1));
   m_1(4) <= (NOT(S(0)) AND G7(1)) OR (S(0) AND G8(1));
   m_1(5) <= (NOT(S(1)) AND m_1(1)) OR (S(1) AND m_1(2));
   m_1(6) <= (NOT(S(1)) AND m_1(3)) OR (S(1) AND m_1(4));
   M(1) <= (NOT(S(2)) AND m_1(5)) OR (S(2) AND m_1(6));

   -- 8-to-1 multiplexer for bit 2
   m_2(1) <= (NOT(S(0)) AND G1(2)) OR (S(0) AND G2(2));
   m_2(2) <= (NOT(S(0)) AND G3(2)) OR (S(0) AND G4(2));
   m_2(3) <= (NOT(S(0)) AND G5(2)) OR (S(0) AND G6(2));
   m_2(4) <= (NOT(S(0)) AND G7(2)) OR (S(0) AND G8(2));
   m_2(5) <= (NOT(S(1)) AND m_2(1)) OR (S(1) AND m_2(2));
   m_2(6) <= (NOT(S(1)) AND m_2(3)) OR (S(1) AND m_2(4));
   M(2) <= (NOT(S(2)) AND m_2(5)) OR (S(2) AND m_2(6));
END Behavior;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

-- Converts 3-bit input code on C2-0 into 7-bit code that produces
-- a character on a 7-segment display. The conversion is defined by:
--     C 2 1 0    Char
--    ----------------
--       0 0 0    'H'
--       0 0 1    'E'
--       0 1 0    'L'
--       0 1 1    'O'
--       1 0 0    ' ' Blank
--       1 0 1    ' ' Blank
--       1 1 0    ' ' Blank
--       1 1 1    ' ' Blank
--
--    Codes 100, 101, 110 are not used
--
ENTITY char_7seg IS
   PORT (   C        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            Display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END char_7seg;
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
ARCHITECTURE Behavior OF char_7seg IS
BEGIN
   -- the following equations describe display functions in (inverted) cannonical SOP
form
   Display(0) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(1) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(2) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(3) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR
      (NOT(C(2)) AND C(1) AND C(0)) );
   Display(4) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   Display(5) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) OR
      (NOT(C(2)) AND C(1) AND NOT(C(0))) OR (NOT(C(2)) AND C(1) AND C(0)) );
   Display(6) <= NOT( (NOT(C(2)) AND NOT(C(1)) AND NOT(C(0))) OR
      (NOT(C(2)) AND NOT(C(1)) AND C(0)) );
END Behavior;
```

Laboratory Exercise 2

Numbers and Displays

This is an exercise in designing combinational circuits that can perform binary-to-decimal number conversion and binary-coded-decimal (BCD) addition.

Part I

We wish to display on the 7-segment displays *HEX3* to *HEX0* the values set by the switches SW_{15-0}. Let the values denoted by SW_{15-12}, SW_{11-8}, SW_{7-4} and SW_{3-0} be displayed on *HEX3*, *HEX2*, *HEX1* and *HEX0*, respectively. Your circuit should be able to display the digits from 0 to 9, and should treat the valuations 1010 to 1111 as don't-cares.

1. Create a new project which will be used to implement the desired circuit on the Altera DE2 board. The intent of this exercise is to manually derive the logic functions needed for the 7-segment displays. You should use only simple VHDL assignment statements in your code and specify each logic function as a Boolean expression.
2. Write a VHDL file that provides the necessary functionality. Include this file in your project and assign the pins on the FPGA to connect to the switches and 7-segment displays, as indicated in the User Manual for the DE2 board. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction using VHDL Design*, which is available on the *DE2 System CD* and in the University Program section of Altera's web site.
3. Compile the project and download the compiled circuit into the FPGA chip.
4. Test the functionality of your design by toggling the switches and observing the displays.

Part II

You are to design a circuit that converts a four-bit binary number $V = v_3v_2v_1v_0$ into its two-digit decimal equivalent $D = d_1d_0$. Table 1 shows the required output values. A partial design of this circuit is given in Figure 1. It includes a comparator that checks when the value of V is greater than 9, and uses the output of this comparator in the control of the 7-segment displays. You are to complete the design of this circuit by creating a VHDL entity which includes the comparator, multiplexers, and circuit *A* (do not include circuit *B* or the 7-segment decoder at this point). Your VHDL entity should have the four-bit input V, the four-bit output M and the output *z*. The intent of this exercise is to use simple VHDL assignment statements to specify the required logic functions using Boolean expressions. Your VHDL code should not include any IF-ELSE, CASE, or similar statements.

Binary value	Decimal digits	
0000	0	0
0001	0	1
0010	0	2
...	...	...
1001	0	9
1010	1	0
1011	1	1
1100	1	2
1101	1	3
1110	1	4
1111	1	5

Table 1. Binary-to-decimal conversion values.

Perform the following steps:

1. Make a Quartus II project for your VHDL entity.
2. Compile the circuit and use functional simulation to verify the correct operation of your comparator, multiplexers, and circuit *A*.
3. Augment your VHDL code to include circuit *B* in Figure 1 as well as the 7-segment decoder. Change the inputs and outputs of your code to use switches SW_{3-0} on the DE2 board to represent the binary number V, and the displays *HEX1* and *HEX0* to show the values of decimal digits d_1 and d_0. Make sure to include in your project the required pin assignments for the DE2 board.
4. Recompile the project, and then download the circuit into the FPGA chip.
5. Test your circuit by trying all possible values of V and observing the output displays.

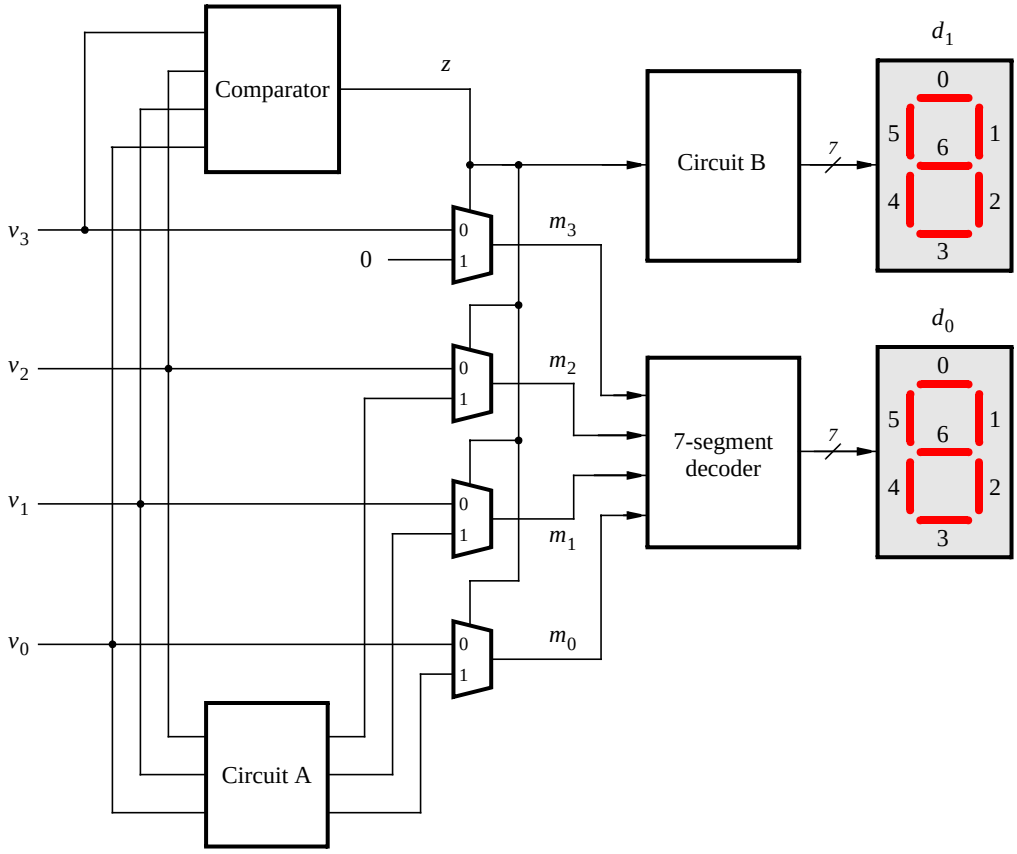
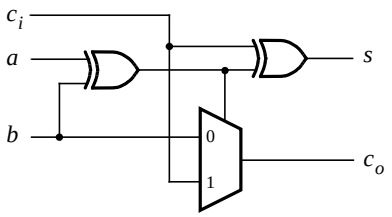


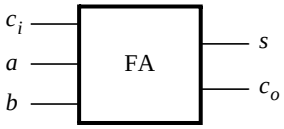
Figure 1. Partial design of the binary-to-decimal conversion circuit.

Part III

Figure 2a shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and c_o. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$. Figure 2d shows how four instances of this full adder entity can be used to design a circuit that adds two four-bit numbers. This type of circuit is usually called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next. Write VHDL code that implements this circuit, as described below.



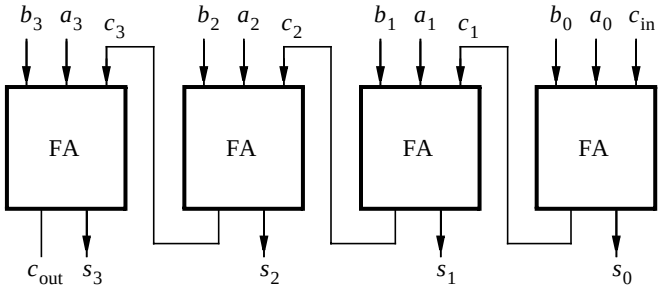
a) Full adder circuit



b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Figure 2. A ripple-carry adder circuit.

1. Create a new Quartus II project for the adder circuit. Write a VHDL entity for the full adder subcircuit and write a top-level VHDL entity that instantiates four instances of this full adder.
2. Use switches SW_{7-4} and SW_{3-0} to represent the inputs A and B, respectively. Use SW_8 for the carry-in c_{in} of the adder. Connect the *SW* switches to their corresponding red lights LEDR, and connect the outputs of the adder, c_{out} and S, to the green lights LEDG.
3. Include the necessary pin assignments for the DE2 board, compile the circuit, and download it into the FPGA chip.
4. Test your circuit by trying different values for numbers A, B, and c_{in}.

Part IV

In part II we discussed the conversion of binary numbers into decimal digits. It is sometimes useful to build circuits that use this method of representing decimal numbers, in which each decimal digit is represented using four bits. This scheme is known as the *binary coded decimal* (BCD) representation. As an example, the decimal value 59 is encoded in BCD form as 0101 1001.

You are to design a circuit that adds two BCD digits. The inputs to the circuit are BCD numbers A and B, plus a carry-in, c_{in}. The output should be a two-digit BCD sum S_1S_0. Note that the largest sum that needs to be handled by this circuit is $S_1S_0 = 9 + 9 + 1 = 19$. Perform the steps given below.

1. Create a new Quartus II project for your BCD adder. You should use the four-bit adder circuit from part III to produce a four-bit sum and carry-out for the operation $A + B$. A circuit that converts this five-bit result, which has the maximum value 19, into two BCD digits S_1S_0 can be designed in a very similar way as the binary-to-decimal converter from part II. Write your VHDL code using simple assignment statements to specify the required logic functions–do not use other types of VHDL statements such as IF-ELSE or CASE statements for this part of the exercise.

2. Use switches SW_{7-4} and SW_{3-0} for the inputs A and B, respectively, and use SW_8 for the carry-in. Connect the *SW* switches to their corresponding red lights LEDR, and connect the four-bit sum and carry-out produced by the operation $A + B$ to the green lights LEDG. Display the BCD values of A and B on the 7-segment displays *HEX6* and *HEX4*, and display the result S_1S_0 on *HEX1* and *HEX0*.

3. Since your circuit handles only BCD digits, check for the cases when the input A or B is greater than nine. If this occurs, indicate an error by turning on the green light *LEDG*$_8$.

4. Include the necessary pin assignments for the DE2 board, compile the circuit, and download it into the FPGA chip.

5. Test your circuit by trying different values for numbers A, B, and c_{in}.

Part V

Design a circuit that can add two 2-digit BCD numbers, A_1A_0 and B_1B_0 to produce the three-digit BCD sum $S_2S_1S_0$. Use two instances of your circuit from part IV to build this two-digit BCD adder. Perform the steps below:

1. Use switches SW_{15-8} and SW_{7-0} to represent 2-digit BCD numbers A_1A_0 and B_1B_0, respectively. The value of A_1A_0 should be displayed on the 7-segment displays *HEX7* and *HEX6*, while B_1B_0 should be on *HEX5* and *HEX4*. Display the BCD sum, $S_2S_1S_0$, on the 7-segment displays *HEX2*, *HEX1* and *HEX0*.

2. Make the necessary pin assignments and compile the circuit.

3. Download the circuit into the FPGA chip, and test its operation.

Part VI

In part V you created VHDL code for a two-digit BCD adder by using two instances of the VHDL code for a one-digit BCD adder from part IV. A different approach for describing the two-digit BCD adder in VHDL code is to specify an algorithm like the one represented by the following pseudo-code:

```
1   T0 = A0 + B0
2   if (T0 > 9) then
3       Z0 = 10;
4       c1 = 1;
5   else
6       Z0 = 0;
7       c1 = 0;
8   end if
9   S0 = T0 − Z0

10  T1 = A1 + B1 + c1
11  if (T1 > 9) then
12      Z1 = 10;
13      c2 = 1;
14  else
15      Z1 = 0;
16      c2 = 0;
17  end if
18  S1 = T1 − Z1
19  S2 = c2
```

It is reasonably straightforward to see what circuit could be used to implement this pseudo-code. Lines 1, 9, 10, and 18 represent adders, lines 2-8 and 11-17 correspond to multiplexers, and testing for the conditions $T_0 > 9$ and $T_1 > 9$ requires comparators. You are to write VHDL code that corresponds to this pseudo-code. Note that you can perform addition operations in your VHDL code instead of the subtractions shown in lines 9 and 18. The intent of this part of the exercise is to examine the effects of relying more on the VHDL compiler to design the circuit by using IF-ELSE statements along with the VHDL $>$ and $+$ operators. Perform the following steps:

1. Create a new Quartus II project for your VHDL code. Use the same switches, lights, and displays as in part V. Compile your circuit.
2. Use the Quartus II RTL Viewer tool to examine the circuit produced by compiling your VHDL code. Compare the circuit to the one you designed in Part V.
3. Download your circuit onto the DE2 board and test it by trying different values for numbers A_1A_0 and B_1B_0.

Part VII

Design a combinational circuit that converts a 6-bit binary number into a 2-digit decimal number represented in the BCD form. Use switches SW_{5-0} to input the binary number and 7-segment displays *HEX1* and *HEX0* to display the decimal number. Implement your circuit on the DE2 board and demonstrate its functionality.

```
-- Display digits from 0 to 9 on the 7-segment displays, using the SW
-- toggle switches as inputs.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part1 IS
   PORT (   SW                     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            LEDR                   : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);  -- red LEDs
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part1;

ARCHITECTURE Structure OF part1 IS
   COMPONENT bcd7seg
      PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               H  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
BEGIN
   LEDR <= SW;

   -- drive the displays through 7-seg decoders
   digit3: bcd7seg PORT MAP (SW(15 DOWNTO 12), HEX3);
   digit2: bcd7seg PORT MAP (SW(11 DOWNTO 8), HEX2);
   digit1: bcd7seg PORT MAP (SW(7 DOWNTO 4), HEX1);
   digit0: bcd7seg PORT MAP (SW(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            H  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Structure OF bcd7seg IS
BEGIN
   --
   --       0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --       3
   --
   -- B  H
   -- ----------
   -- 0  0000001;
   -- 1  1001111;
   -- 2  0010010;
   -- 3  0000110;
   -- 4  1001100;
   -- 5  0100100;
   -- 6  1100000;
   -- 7  0001111;
   -- 8  0000000;
   -- 9  0001100;
   H(0) <= (B(2) AND NOT(B(0))) OR
      (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)) AND B(0));
```

```
   H(1) <= (B(2) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND B(1) AND NOT(B(0)));
   H(2) <= (NOT(B(2)) AND B(1) AND NOT(B(0)));
   H(3) <= (NOT(B(2)) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND NOT(B(1)) AND NOT(B(0))) OR (B(2) AND B(1) AND B(0));
   H(4) <= (NOT(B(1)) AND B(0)) OR (NOT(B(3)) AND B(0)) OR
      (NOT(B(3)) AND B(2) AND NOT(B(1)));
   H(5) <= (B(1) AND B(0)) OR (NOT(B(2)) AND B(1)) OR
      (NOT(B(3)) AND NOT(B(2)) AND B(0));
   H(6) <= (B(2) AND B(1) AND B(0)) OR (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)));
END Structure;
```

```
-- bcd-to-decimal converter

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part2 IS
   PORT (   SW          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part2;

ARCHITECTURE Structure OF part2 IS
   COMPONENT bcd7seg
      PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               H  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL V, M : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL B : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL z : STD_LOGIC;
BEGIN
   V <= SW;

   -- circuit A
   z <= (V(3) AND V(2)) OR (V(3) AND V(1));

   -- Circuit B
   B(2) <= V(2) AND V(1);
   B(1) <= V(2) AND NOT(V(1));
   B(0) <= (V(1) AND V(0)) OR (V(2) AND V(0));

   -- multiplexers
   M(3) <= NOT(z) AND V(3);
   M(2) <= (NOT(z) AND V(2)) OR (z AND B(2));
   M(1) <= (NOT(z) AND V(1)) OR (z AND B(1));
   M(0) <= (NOT(z) AND V(0)) OR (z AND B(0));

   -- Circuit D
   Circuit_D: bcd7seg PORT MAP (M, HEX0);

   -- Circuit C
   HEX1 <= ('1' & NOT(z) & NOT(z) & "1111"); -- display a blank or the digit 1
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            H  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Structure OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
```

```
   --
   -- B  H
   -- ----------
   -- 0  0000001;
   -- 1  1001111;
   -- 2  0010010;
   -- 3  0000110;
   -- 4  1001100;
   -- 5  0100100;
   -- 6  1100000;
   -- 7  0001111;
   -- 8  0000000;
   -- 9  0001100;
   H(0) <= (B(2) AND NOT(B(0))) OR
      (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)) AND B(0));
   H(1) <= (B(2) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND B(1) AND NOT(B(0)));
   H(2) <= (NOT(B(2)) AND B(1) AND NOT(B(0)));
   H(3) <= (NOT(B(2)) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND NOT(B(1)) AND NOT(B(0))) OR (B(2) AND B(1) AND B(0));
   H(4) <= (NOT(B(1)) AND B(0)) OR (NOT(B(3)) AND B(0)) OR
      (NOT(B(3)) AND B(2) AND NOT(B(1)));
   H(5) <= (B(1) AND B(0)) OR (NOT(B(2)) AND B(1)) OR
      (NOT(B(3)) AND NOT(B(2)) AND B(0));
   H(6) <= (B(2) AND B(1) AND B(0)) OR (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)));
END Structure;
```

```
-- 4-bit ripple-carry adder

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part3 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDG  : OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
END part3;

ARCHITECTURE Structure OF part3 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL A, B, S : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL C : STD_LOGIC_VECTOR(4 DOWNTO 1);
BEGIN
   A <= SW(7 DOWNTO 4);
   B <= SW(3 DOWNTO 0);

   bit0: fa PORT MAP (A(0), B(0), '0', S(0), C(1));
   bit1: fa PORT MAP (A(1), B(1), C(1), S(1), C(2));
   bit2: fa PORT MAP (A(2), B(2), C(2), S(2), C(3));
   bit3: fa PORT MAP (A(3), B(3), C(3), S(3), C(4));

   -- Display the inputs
   LEDR <= SW;
   LEDG <= (C(4) & S);
END Structure;


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;
```

```
-- one-digit BCD adder S1 S0 = A + B + Cin
-- inputs: SW7-4 = A
--         SW3-0 = B
-- outputs: A is displayed on HEX6
--          B is displayed on HEX4
--          S1 S0 is displayed on HEX1 HEX

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part4 IS
   PORT (   SW                     : IN  STD_LOGIC_VECTOR(8 DOWNTO 0);
            LEDR                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);  -- red LEDs
            LEDG                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);  -- red LEDs
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);  -- 7-segs
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part4;

ARCHITECTURE Structure OF part4 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT bcd_decimal
      PORT (   V     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               z     : BUFFER STD_LOGIC;
               M     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));  -- 7-segs
   END COMPONENT;
   COMPONENT bcd7seg
      PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               H  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL A, B, S : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL Cin : STD_LOGIC;
   SIGNAL C : STD_LOGIC_VECTOR(4 DOWNTO 1);
   SIGNAL S0 : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL S0_M : STD_LOGIC_VECTOR(3 DOWNTO 0); -- modified S0 for sums > 15
   SIGNAL S1 : STD_LOGIC;
BEGIN
   A <= SW(7 DOWNTO 4);
   B <= SW(3 DOWNTO 0);
   Cin <= SW(8);

   bit0: fa PORT MAP (A(0), B(0), Cin, S(0), C(1));
   bit1: fa PORT MAP (A(1), B(1), C(1), S(1), C(2));
   bit2: fa PORT MAP (A(2), B(2), C(2), S(2), C(3));
   bit3: fa PORT MAP (A(3), B(3), C(3), S(3), C(4));

   -- Display the inputs
   LEDR <= SW;
   LEDG(4 DOWNTO 0) <= (C(4) & S);

   -- Display the inputs
   H_6: bcd7seg PORT MAP (A, HEX6);
   HEX7 <= ("1111111"); -- display blank

   H_4: bcd7seg PORT MAP (B, HEX4);
   HEX5 <= "1111111";   -- display blank

   -- Detect illegal inputs, display on LEDG(8)
   LEDG(8) <= (A(3) AND A(2)) OR (A(3) AND A(1)) OR
      (B(3) AND B(2)) OR (B(3) AND B(1));
   LEDG(7 DOWNTO 5) <= "000";
```

```
   -- Display the sum
   -- bcd_decimal (V, z, M);
   BCD_S: bcd_decimal PORT MAP (S, S1, S0);
   -- S is really a 5-bit # with the carry-out, but bcd_decimal handles only
   -- the lower four bit (sums 00-15).  To account for sums 16, 17, 18, 19 S0
   -- has to be modified in the cases that carry-out = 1. Use multiplexers:
   S0_M(3) <= (NOT(C(4)) AND S0(3)) OR (C(4) AND S0(1));
   S0_M(2) <= (NOT(C(4)) AND S0(2)) OR (C(4) AND NOT(S0(1)));
   S0_M(1) <= (NOT(C(4)) AND S0(1)) OR (C(4) AND NOT(S0(1)));
   S0_M(0) <= S0(0);
   H_0: bcd7seg PORT MAP (S0_M, HEX0);
   -- S is really a 5-bit #, but bcd_decimal works for only the lower four bits
   -- (sums 00-15). To account for sums 16, 17, 18, 19 S1 should be a 1 when
   -- the carry-out is a 1
   HEX1 <= ('1' & NOT(S1 OR C(4)) & NOT(S1 OR C(4)) & "1111"); -- display blank or 1
   HEX2 <= "1111111";   -- display blank
   HEX3 <= "1111111";   -- display blank
END Structure;


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd_decimal IS
   PORT (   V     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            z     : BUFFER STD_LOGIC;
            M     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));  -- 7-segs
END bcd_decimal;

ARCHITECTURE Structure OF bcd_decimal IS
   SIGNAL B : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   -- circuit A
   z <= (V(3) AND V(2)) OR (V(3) AND V(1));

   -- Circuit B
   B(2) <= V(2) AND V(1);
   B(1) <= V(2) AND NOT(V(1));
   B(0) <= (V(1) AND V(0)) OR (V(2) AND V(0));

   -- multiplexers
   M(3) <= NOT(z) AND V(3);
   M(2) <= (NOT(z) AND V(2)) OR (z AND B(2));
   M(1) <= (NOT(z) AND V(1)) OR (z AND B(1));
   M(0) <= (NOT(z) AND V(0)) OR (z AND B(0));
END Structure;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            H  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Structure OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   -- B  H
   -- ----------
   -- 0  0000001;
   -- 1  1001111;
   -- 2  0010010;
   -- 3  0000110;
   -- 4  1001100;
   -- 5  0100100;
   -- 6  1100000;
   -- 7  0001111;
   -- 8  0000000;
   -- 9  0001100;
   H(0) <= (B(2) AND NOT(B(0))) OR
      (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)) AND B(0));
   H(1) <= (B(2) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND B(1) AND NOT(B(0)));
   H(2) <= (NOT(B(2)) AND B(1) AND NOT(B(0)));
   H(3) <= (NOT(B(2)) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND NOT(B(1)) AND NOT(B(0))) OR (B(2) AND B(1) AND B(0));
   H(4) <= (NOT(B(1)) AND B(0)) OR (NOT(B(3)) AND B(0)) OR
      (NOT(B(3)) AND B(2) AND NOT(B(1)));
   H(5) <= (B(1) AND B(0)) OR (NOT(B(2)) AND B(1)) OR
      (NOT(B(3)) AND NOT(B(2)) AND B(0));
   H(6) <= (B(2) AND B(1) AND B(0)) OR (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)));
END Structure;
```

```
-- implements a two-digit bcd adder S2 S1 S0 = A1 A0 + B1 B0
-- inputs: SW15-8 = A1 A0
--         SW7-0 = B1 B0
-- outputs: A1 A0 is displayed on HEX7 HEX6
--          B1 B0 is displayed on HEX5 HEX4
--          S2 S1 S0 is displayed on HEX2 HEX1 HEX0

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part5 IS
   PORT (   SW                      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);  -- 7-segs
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part5;

ARCHITECTURE Structure OF part5 IS
   COMPONENT part4
      PORT (   A, B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               Cin   : IN  STD_LOGIC;
               S1    : OUT STD_LOGIC;
               S0    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
   END COMPONENT;
   COMPONENT bcd7seg
      PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               H  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;

   SIGNAL A1, A0, B1, B0, S1, S0 : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL C1, C2, S2 : STD_LOGIC;
BEGIN
   A1 <= SW(15 DOWNTO 12);
   A0 <= SW(11 DOWNTO 8);
   B1 <= SW(7 DOWNTO 4);
   B0 <= SW(3 DOWNTO 0);

   -- part4 (A, B, Cin, S1, S0);
   BCD_0: part4 PORT MAP (A0, B0, '0', C1, S0);
   BCD_1: part4 PORT MAP (A1, B1, C1, C2, S1);
   S2 <= C2;

   -- drive the displays through 7-seg decoders
   digit7: bcd7seg PORT MAP (A1, HEX7);
   digit6: bcd7seg PORT MAP (A0, HEX6);
   digit5: bcd7seg PORT MAP (B1, HEX5);
   digit4: bcd7seg PORT MAP (B0, HEX4);
   digit2: bcd7seg PORT MAP (("000" & S2), HEX2);
   digit1: bcd7seg PORT MAP (S1, HEX1);
   digit0: bcd7seg PORT MAP (S0, HEX0);

   HEX3 <= "1111111";   -- turn off HEX3
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- one digit BCD adder S1 S0 = A + B + Cin
ENTITY part4 IS
   PORT (   A, B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            Cin   : IN  STD_LOGIC;
            S1    : OUT STD_LOGIC;
            S0    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END part4;
```

```
ARCHITECTURE Structure OF part4 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT bcd_decimal
      PORT (   V     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               z     : BUFFER STD_LOGIC;
               M     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));  -- 7-segs
   END COMPONENT;

   SIGNAL C : STD_LOGIC_VECTOR(4 DOWNTO 1);
   SIGNAL S, S0_M : STD_LOGIC_VECTOR(3 DOWNTO 0); -- modified S0 for sums > 15
   SIGNAL S1_M : STD_LOGIC; -- used because S1 has to be modified for sums > 15
BEGIN
   bit0: fa PORT MAP (A(0), B(0), Cin, S(0), C(1));
   bit1: fa PORT MAP (A(1), B(1), C(1), S(1), C(2));
   bit2: fa PORT MAP (A(2), B(2), C(2), S(2), C(3));
   bit3: fa PORT MAP (A(3), B(3), C(3), S(3), C(4));

   -- convert the sum to BCD
   BCD_S: bcd_decimal PORT MAP (S, S1_M, S0_M);
   -- S is really a 5-bit # with the carry-out, but bcd_decimal handles only
   -- the lower four bits (sums 00-15).  To account for sums 16, 17, 18, 19 S0
   -- has to be modified in the cases that carry-out = 1. Use multiplexers:
   S0(3) <= (NOT(C(4)) AND S0_M(3)) OR (C(4) AND S0_M(1));
   S0(2) <= (NOT(C(4)) AND S0_M(2)) OR (C(4) AND NOT(S0_M(1)));
   S0(1) <= (NOT(C(4)) AND S0_M(1)) OR (C(4) AND NOT(S0_M(1)));
   S0(0) <= S0_M(0);
   -- S is really a 5-bit #, but bcd_decimal works for only the lower four bits
   -- (sums 00-15). To account for sums 16, 17, 18, 19 S1 should be a 1 when
   -- the carry-out is a 1
   S1 <= S1_M OR C(4);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd_decimal IS
   PORT (   V     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            z     : BUFFER STD_LOGIC;
            M     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));  -- 7-segs
END bcd_decimal;

ARCHITECTURE Structure OF bcd_decimal IS
   SIGNAL B : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   -- circuit A
```

```
   z <= (V(3) AND V(2)) OR (V(3) AND V(1));

   -- Circuit B
   B(2) <= V(2) AND V(1);
   B(1) <= V(2) AND NOT(V(1));
   B(0) <= (V(1) AND V(0)) OR (V(2) AND V(0));

   -- multiplexers
   M(3) <= NOT(z) AND V(3);
   M(2) <= (NOT(z) AND V(2)) OR (z AND B(2));
   M(1) <= (NOT(z) AND V(1)) OR (z AND B(1));
   M(0) <= (NOT(z) AND V(0)) OR (z AND B(0));
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   B  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            H  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Structure OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   -- B  H
   -- ----------
   -- 0  0000001;
   -- 1  1001111;
   -- 2  0010010;
   -- 3  0000110;
   -- 4  1001100;
   -- 5  0100100;
   -- 6  1100000;
   -- 7  0001111;
   -- 8  0000000;
   -- 9  0001100;
   H(0) <= (B(2) AND NOT(B(0))) OR
      (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)) AND B(0));
   H(1) <= (B(2) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND B(1) AND NOT(B(0)));
   H(2) <= (NOT(B(2)) AND B(1) AND NOT(B(0)));
   H(3) <= (NOT(B(2)) AND NOT(B(1)) AND B(0)) OR
      (B(2) AND NOT(B(1)) AND NOT(B(0))) OR (B(2) AND B(1) AND B(0));
   H(4) <= (NOT(B(1)) AND B(0)) OR (NOT(B(3)) AND B(0)) OR
      (NOT(B(3)) AND B(2) AND NOT(B(1)));
   H(5) <= (B(1) AND B(0)) OR (NOT(B(2)) AND B(1)) OR
      (NOT(B(3)) AND NOT(B(2)) AND B(0));
   H(6) <= (B(2) AND B(1) AND B(0)) OR (NOT(B(3)) AND NOT(B(2)) AND NOT(B(1)));
END Structure;
```

```
-- implements a two-digit bcd adder S2 S1 S0 = A1 A0 + B1 B0
-- inputs: SW15-8 = A1 A0
--         SW7-0 = B1 B0
-- outputs: A1 A0 is displayed on HEX7 HEX6
--          B1 B0 is displayed on HEX5 HEX4
--          S2 S1 S0 is displayed on HEX2 HEX1 HEX0

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY part6 IS
   PORT (   SW                      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);  -- 7-segs
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part6;

ARCHITECTURE Behavior OF part6 IS
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;

   SIGNAL A1, A0, B1, B0 : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL S2 : STD_LOGIC;
   SIGNAL S1, S0 : STD_LOGIC_VECTOR(3 DOWNTO 0);

   SIGNAL Z1, Z0 : STD_LOGIC_VECTOR(3 DOWNTO 0);   -- used for BCD addition
   SIGNAL T1, T0 : STD_LOGIC_VECTOR(4 DOWNTO 0);   -- used for BCD addition
   SIGNAL C1, C2 : STD_LOGIC; -- used for BCD addition
BEGIN
   A1 <= SW(15 DOWNTO 12);
   A0 <= SW(11 DOWNTO 8);
   B1 <= SW(7 DOWNTO 4);
   B0 <= SW(3 DOWNTO 0);

   -- add lower two bcd digits. Result is five bits: C1,S0
   T0 <= ('0' & A0) + ('0' & B0);
   PROCESS (T0)
   BEGIN
      IF (T0 > "01001") THEN
         Z0 <= "0110";  -- we will add +6 instead of -10
         C1 <= '1';
      ELSE
         Z0 <= "0000";  -- we will add +6 instead of -10
         C1 <= '0';
      END IF;
   END PROCESS;
   S0 <= T0(3 DOWNTO 0) + Z0; -- using 4 bits, + 6 is same as - 10

   -- add upper two bcd digits plus C1
   T1 <= ('0' & A1) + ('0' & B1) + C1;
   PROCESS (T1)
   BEGIN
      IF (T1 > "01001") THEN
         Z1 <= "0110";  -- we will add +6 instead of -10
         C2 <= '1';
      ELSE
         Z1 <= "0000";  -- we will add +6 instead of -10
         C2 <= '0';
      END IF;
   END PROCESS;
   S1 <= T1(3 DOWNTO 0) + Z1; -- using 4 bits, + 6 is same as - 10
   S2 <= C2;
```

```
   -- drive the displays through 7-seg decoders
   digit7: bcd7seg PORT MAP (A1, HEX7);
   digit6: bcd7seg PORT MAP (A0, HEX6);
   digit5: bcd7seg PORT MAP (B1, HEX5);
   digit4: bcd7seg PORT MAP (B0, HEX4);
   digit2: bcd7seg PORT MAP (("000" & S2), HEX2);
   digit1: bcd7seg PORT MAP (S1, HEX1);
   digit0: bcd7seg PORT MAP (S0, HEX0);

   HEX3 <= "1111111";   -- turn off HEX3
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- input six bits using SW toggle switches, and convert to decimal (2-digit bcd)
ENTITY part7 IS
   PORT (   SW                      : IN  STD_LOGIC_VECTOR(5 DOWNTO 0);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));  -- 7-segs
END part7;

ARCHITECTURE Behavior OF part7 IS
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;

   SIGNAL bcd_h, bcd_l : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL bin6 : STD_LOGIC_VECTOR(5 DOWNTO 0);
BEGIN
   bin6 <= SW;

   -- Check various ranges and set bcd digits. Note that we work with
   -- bin6(3 DOWNTO 0) just to prevent compiler warnings about bit size
   -- truncation. This is not really necessary
   PROCESS (bin6)
   BEGIN
      IF (bin6 < "001010") THEN
         bcd_h <= "0000";
         bcd_l <= bin6(3 DOWNTO 0);
      ELSIF (bin6 < "010100") THEN
         bcd_h <= "0001";
         bcd_l <= bin6(3 DOWNTO 0) + "0110"; -- -10 = -(00001010) = 11110110; add 6
      ELSIF (bin6 < "011110") THEN
         bcd_h <= "0010";
         bcd_l <= bin6(3 DOWNTO 0) + "1100"; -- -20 = -(00010100) = 11101100; add 12
      ELSIF (bin6 < "101000") THEN
         bcd_h <= "0011";
         bcd_l <= bin6(3 DOWNTO 0) + "0010"; -- -30 = -(00011110) = 11100010; add 2
      ELSIF (bin6 < "110010") THEN
         bcd_h <= "0100";
         bcd_l <= bin6(3 DOWNTO 0) + "1000"; -- -40 = -(00101000) = 11011000; add 8
      ELSIF (bin6 < "111100") THEN
         bcd_h <= "0101";
         bcd_l <= bin6(3 DOWNTO 0) + "1110"; -- -50 = -(00110010) = 11001110; add 14
      ELSE
         bcd_h <= "0110";
         bcd_l <= bin6(3 DOWNTO 0) + "0100"; -- -60 = -(00111100) = 11000100; add 4
      END IF;
   END PROCESS;

   -- drive the displays
   digit1: bcd7seg PORT MAP (bcd_h, HEX1);
   digit0: bcd7seg PORT MAP (bcd_l, HEX0);

   -- blank the adjacent displays
   digit3: bcd7seg PORT MAP ("1111", HEX3);
   digit2: bcd7seg PORT MAP ("1111", HEX2);

END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
```

```
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

Laboratory Exercise 3

Latches, Flip-flops, and Registers

The purpose of this exercise is to investigate latches, flip-flops, and registers.

Part I

Altera FPGAs include flip-flops that are available for implementing a user's circuit. We will show how to make use of these flip-flops in Parts IV to VII of this exercise. But first we will show how storage elements can be created in an FPGA without using its dedicated flip-flops.

Figure 1 depicts a gated RS latch circuit. A style of VHDL code that uses logic expressions to describe this circuit is given in Figure 2. If this latch is implemented in an FPGA that has 4-input lookup tables (LUTs), then only one lookup table is needed, as shown in Figure 3*a*.

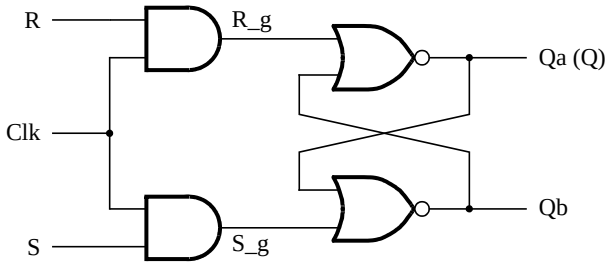
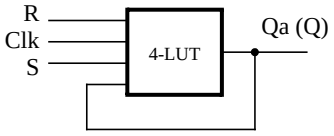


Figure 1. A gated RS latch circuit.

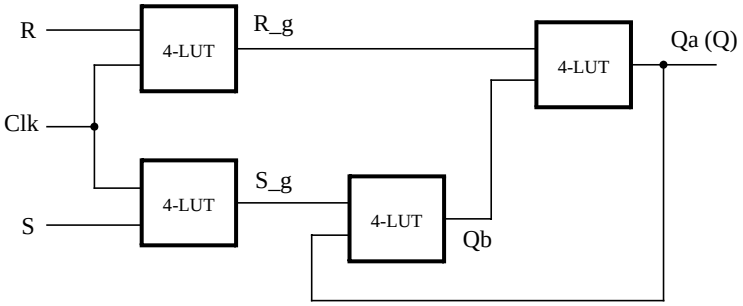
```
-- A gated RS latch desribed the hard way
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part1 IS
    PORT ( Clk, R, S : IN      STD_LOGIC;
           Q         : OUT     STD_LOGIC);
END part1;

ARCHITECTURE Structural OF part1 IS
    SIGNAL R_g, S_g, Qa, Qb : STD_LOGIC ;
    ATTRIBUTE keep : boolean;
    ATTRIBUTE keep of R_g, S_g, Qa, Qb : SIGNAL IS true;
BEGIN
    R_g <= R AND Clk;
    S_g <= S AND Clk;
    Qa <= NOT (R_g OR Qb);
    Qb <= NOT (S_g OR Qa);

    Q <= Qa;

END Structural;
```

Figure 2. Specifying the RS latch by using logic expressions.

Although the latch can be correctly realized in one 4-input LUT, this implementation does not allow its internal signals, such as R_g and S_g, to be observed, because they are not provided as outputs from the LUT. To preserve these internal signals in the implemented circuit, it is necessary to include a *compiler directive* in the code. In Figure 2 the directive *keep* is included by using a VHDL ATTRIBUTE statement; it instructs the Quartus II compiler to use separate logic elements for each of the signals R_g, S_g,Qa, and Qb. Compiling the code produces the circuit with four 4-LUTs depicted in Figure 3b.



(a) Using one 4-input lookup table for the RS latch.



(b) Using four 4-input lookup tables for the RS latch.

Figure 3. Implementation of the RS latch from Figure 1.

Create a Quartus II project for the RS latch circuit as follows:

1. Create a new project for the RS latch. Select as the target chip the Cyclone II EP2C35F672C6, which is the FPGA chip on the Altera DE2 board.
2. Generate a VHDL file with the code in Figure 2 and include it in the project.
3. Compile the code. Use the Quartus II RTL Viewer tool to examine the gate-level circuit produced from the code, and use the Technology Viewer tool to verify that the latch is implemented as shown in Figure 3b.
4. Create a Vector Waveform File (.vwf) which specifies the inputs and outputs of the circuit. Draw waveforms for the R and S inputs and use the Quartus II Simulator to produce the corresponding waveforms for *R_g*, *S_g*, Q*a*, and Q*b*. Verify that the latch works as expected using both functional and timing simulation.

Part II

Figure 4 shows the circuit for a gated D latch.

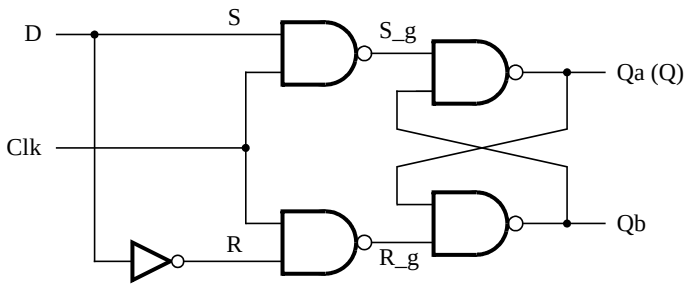


Figure 4. Circuit for a gated D latch.

Perform the following steps:

1. Create a new Quartus II project. Generate a VHDL file using the style of code in Figure 2 for the gated D latch. Use the *keep* directive to ensure that separate logic elements are used to implement the signals R, S_g, R_g,Qa, and Qb.
2. Select as the target chip the Cyclone II EP2C35F672C6 and compile the code. Use the Technology Viewer tool to examine the implemented circuit.
3. Verify that the latch works properly for all input conditions by using functional simulation. Examine the timing characteristics of the circuit by using timing simulation.
4. Create a new Quartus II project which will be used for implementation of the gated D latch on the DE2 board. This project should consist of a top-level entity that contains the appropriate input and output ports (pins) for the DE2 board. Instantiate your latch in this top-level entity. Use switch SW_0 to drive the *D* input of the latch, and use SW_1 as the *Clk* input. Connect the Q output to $LEDR_0$.
5. Recompile your project and download the compiled circuit onto the DE2 board.
6. Test the functionality of your circuit by toggling the D and *Clk* switches and observing the Q output.

Part III

Figure 5 shows the circuit for a master-slave D flip-flop.

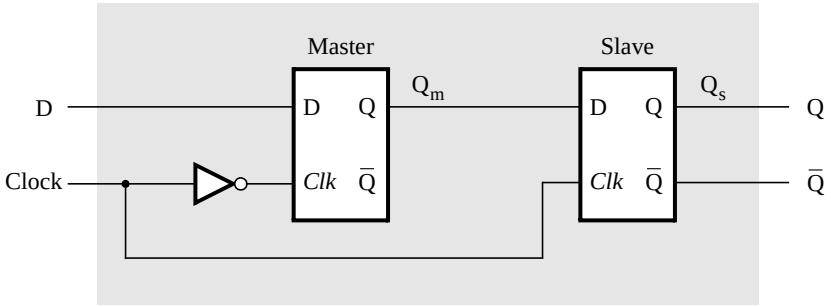


Figure 5. Circuit for a master-slave D flip-flop.

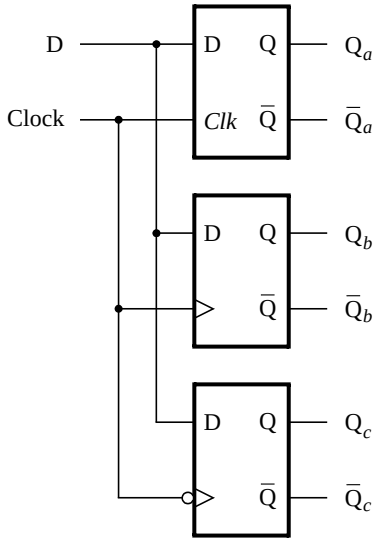
Perform the following:

1. Create a new Quartus II project. Generate a VHDL file that instantiates two copies of your gated D latch entity from Part II to implement the master-slave flip-flop.

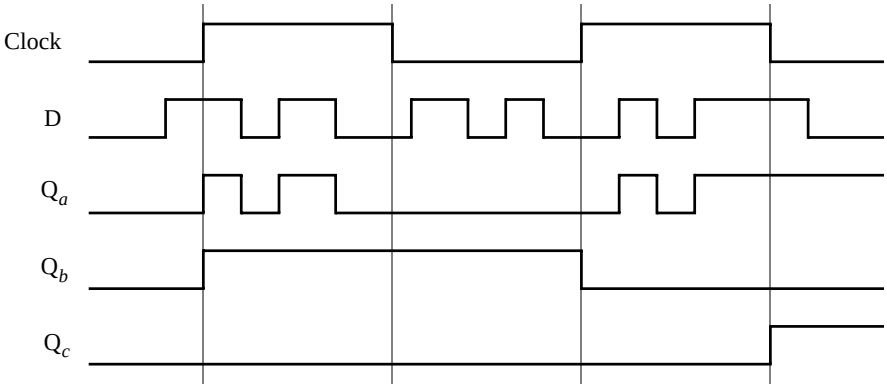
2. Include in your project the appropriate input and output ports for the Altera DE2 board. Use switch SW_0 to drive the D input of the flip-flop, and use SW_1 as the *Clock* input. Connect the Q output to $LEDR_0$.
3. Compile your project.
4. Use the Technology Viewer to examine the D flip-flop circuit, and use simulation to verify its correct operation.
5. Download the circuit onto the DE2 board and test its functionality by toggling the *D* and *Clock* switches and observing the Q output.

Part IV

Figure 6 shows a circuit with three different storage elements: a gated D latch, a positive-edge triggered D flip-flop, and a negative-edge triggered D flip-flop.



(a) Circuit



(b) Timing diagram

Figure 6. Circuit and waveforms for Part IV.

Implement and simulate this circuit using Quartus II software as follows:

1. Create a new project.
2. Write a VHDL file that instantiates the three storage elements. For this part you should no longer use the *keep* directive (that is, the VHDL ATTRIBUTE statement) from Parts I to III. Figure 7 gives a behavioral style of VHDL code that specifies the gated D latch in Figure 4. This latch can be implemented in one 4-input lookup table. Use a similar style of code to specify the flip-flops in Figure 6.
3. Compile your code and use the Technology Viewer to examine the implemented circuit. Verify that the latch uses one lookup table and that the flip-flops are implemented using the flip-flops provided in the target FPGA.
4. Create a Vector Waveform File (.vwf) which specifies the inputs and outputs of the circuit. Draw the inputs D and *Clock* as indicated in Figure 6. Use functional simulation to obtain the three output signals. Observe the different behavior of the three storage elements.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
   PORT (  D, Clk    : IN       STD_LOGIC ;
           Q         : OUT      STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
   PROCESS ( D, Clk )
   BEGIN
      IF Clk = '1' THEN
         Q <= D ;
      END IF ;
   END PROCESS ;
END Behavior ;
```

Figure 7. A behavioral style of VHDL code that specifies a gated D latch.

Part V

We wish to display the hexadecimal value of a 16-bit number A on the four 7-segment displays, $HEX7-4$. We also wish to display the hex value of a 16-bit number B on the four 7-segment displays, $HEX3-0$. The values of A and B are inputs to the circuit which are provided by means of switches SW_{15-0}. This is to be done by first setting the switches to the value of A and then setting the switches to the value of B; therefore, the value of A must be stored in the circuit.

1. Create a new Quartus II project which will be used to implement the desired circuit on the Altera DE2 board.
2. Write a VHDL file that provides the necessary functionality. Use KEY_0 as an active-low asynchronous reset, and use KEY_1 as a clock input. Include the VHDL file in your project and compile the circuit.
3. Assign the pins on the FPGA to connect to the switches and 7-segment displays, as indicated in the User Manual for the DE2 board.
4. Recompile the circuit and download it into the FPGA chip.
5. Test the functionality of your design by toggling the switches and observing the output displays.

```
-- A gated RS latch described the hard way
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part1 IS
   PORT (   Clk, R, S   : IN  STD_LOGIC;
            Q           : OUT STD_LOGIC);
END part1;

ARCHITECTURE Structural OF part1 IS
   SIGNAL R_g, S_g, Qa, Qb : STD_LOGIC ;
   ATTRIBUTE keep: boolean;
   ATTRIBUTE keep of R_g, S_g, Qa, Qb : signal is true;
BEGIN
   R_g <= R AND Clk;
   S_g <= S AND Clk;
   Qa <= NOT (R_g OR Qb);
   Qb <= NOT (S_g OR Qa);


   Q <= Qa;

END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- A gated D latch described the hard way
ENTITY D_latch IS
   PORT (   Clk, D   : IN  STD_LOGIC;
            Q        : OUT STD_LOGIC);
END D_latch;

ARCHITECTURE Structural OF D_latch IS
   SIGNAL R, R_g, S_g, Qa, Qb : STD_LOGIC ;
   ATTRIBUTE keep: boolean;
   ATTRIBUTE keep of R, R_g, S_g, Qa, Qb : signal is true;
BEGIN
   R <= NOT D;
   S_g <= NOT (D AND Clk);
   R_g <= NOT (R AND Clk);
   Qa <= NOT (S_g AND Qb);
   Qb <= NOT (R_g AND Qa);

   Q <= Qa;
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- SW[0] is the latch's D input, SW[1] is the level-sensitive Clk, LEDR[0] is Q
ENTITY top IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(0 TO 0));
END top;

ARCHITECTURE Structural OF top IS
   COMPONENT D_latch
      PORT (   Clk, D   : IN  STD_LOGIC;
               Q        : OUT STD_LOGIC);
   END COMPONENT;
BEGIN
   -- D_latch (input Clk, D, output Q)
   U1: D_latch PORT MAP (SW(1), SW(0), LEDR(0));

END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- A gated D latch described the hard way
ENTITY D_latch IS
   PORT (   Clk, D   : IN  STD_LOGIC;
            Q        : OUT STD_LOGIC);
END D_latch;

ARCHITECTURE Structural OF D_latch IS
   SIGNAL R, R_g, S_g, Qa, Qb : STD_LOGIC ;
   ATTRIBUTE keep: boolean;
   ATTRIBUTE keep of R, R_g, S_g, Qa, Qb : signal is true;
BEGIN
   R <= NOT D;
   S_g <= NOT (D AND Clk);
   R_g <= NOT (R AND Clk);
   Qa <= NOT (S_g AND Qb);
   Qb <= NOT (R_g AND Qa);

   Q <= Qa;
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- SW[0] is the flip-flop's D input, SW[1] is the edge-sensitive Clock, LEDR[0] is Q
ENTITY part3 IS
   PORT (   SW    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            LEDR  : OUT STD_LOGIC_VECTOR(0 TO 0));
END part3;

ARCHITECTURE Structural OF part3 IS
   COMPONENT D_latch
      PORT (   Clk, D   : IN  STD_LOGIC;
               Q        : OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL Qm, Qs : STD_LOGIC;
BEGIN
   -- D_latch (input Clk, D, output Q)
   U1: D_latch PORT MAP (NOT SW(1), SW(0), Qm);
   U2: D_latch PORT MAP (SW(1), Qm, Qs);

   LEDR(0) <= Qs;

END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- A gated D latch described the hard way
ENTITY D_latch IS
   PORT (   Clk, D   : IN  STD_LOGIC;
            Q        : OUT STD_LOGIC);
END D_latch;

ARCHITECTURE Structural OF D_latch IS
   SIGNAL R, R_g, S_g, Qa, Qb : STD_LOGIC ;
   ATTRIBUTE keep: boolean;
   ATTRIBUTE keep of R, R_g, S_g, Qa, Qb : signal is true;
BEGIN
   R <= NOT D;
   S_g <= NOT (D AND Clk);
   R_g <= NOT (R AND Clk);
   Qa <= NOT (S_g AND Qb);
   Qb <= NOT (R_g AND Qa);

   Q <= Qa;
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- inputs:
-- Clk: manual clock
-- D: data input
--
-- outputs:
-- Qa: gated D-latch output
-- Qb: positive edge-triggered D flip-flop output
-- Qc: negative edge-triggered D flip-flop output
ENTITY part4 IS
   PORT (   Clk, D      : IN  STD_LOGIC;
            Qa, Qb, Qc  : OUT STD_LOGIC);
END part4;

ARCHITECTURE Behavior OF part4 IS
BEGIN
   -- gated D-latch
   PROCESS (D, Clk)
   BEGIN
      IF (Clk = '1') THEN
         Qa <= D;
      END IF;
   END PROCESS;

   PROCESS (Clk)
   BEGIN
      IF (Clk'EVENT AND Clk = '1') THEN
         Qb <= D;
      END IF;
   END PROCESS;

   PROCESS (Clk)
   BEGIN
      IF (Clk'EVENT AND Clk = '0') THEN
         Qc <= D;
      END IF;
   END PROCESS;

END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- KEY0 is resetn, KEY1 is the clock for reg_A
ENTITY part5 IS
   PORT (   SW                         : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            KEY                        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4,
               HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   COMPONENT regn
      PORT (   R                 : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Clock, Resetn     : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
   END COMPONENT;
   SIGNAL A, B :  STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- regn (R, Clock, Resetn, Q)
   A_reg: regn PORT MAP (SW, KEY(1), KEY(0), A);
   B <= SW;

   -- drive the displays through 7-seg decoders
   digit_7: hex7seg PORT MAP (A(15 DOWNTO 12), HEX7);
   digit_6: hex7seg PORT MAP (A(11 DOWNTO 8), HEX6);
   digit_5: hex7seg PORT MAP (A(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (A(3 DOWNTO 0), HEX4);

   digit_3: hex7seg PORT MAP (B(15 DOWNTO 12), HEX3);
   digit_2: hex7seg PORT MAP (B(11 DOWNTO 8), HEX2);
   digit_1: hex7seg PORT MAP (B(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (B(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   PORT (   R                 : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            Clock, Resetn     : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
```

```
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

Laboratory Exercise 4

Counters

This is an exercise in using counters.

Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter which uses four T-type flip-flops. The counter increments its count on each positive edge of the clock if the Enable signal is asserted. The counter is reset to 0 by using the Reset signal. You are to implement a 16-bit counter of this type.

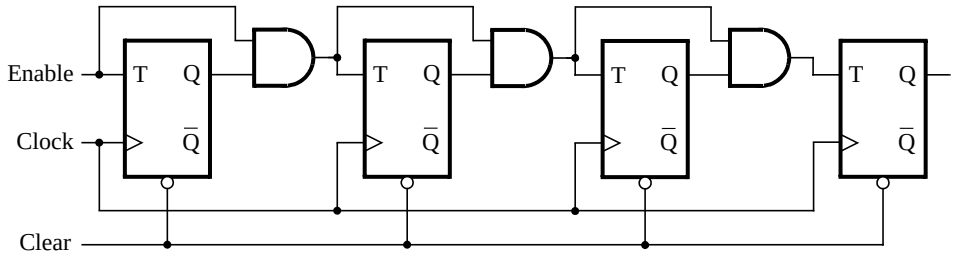


Figure 1. A 4-bit counter.

1. Write a VHDL file that defines a 16-bit counter by using the structure depicted in Figure 8, and compile the circuit. How many logic elements (LEs) are used to implement your circuit? What is the maximum frequency, *Fmax*, at which your circuit can be operated?
2. Simulate your circuit to verify its correctness.
3. Augment your VHDL file to use the pushbutton KEY_0 as the *Clock* input, switches SW_1 and SW_0 as *Enable* and *Reset* inputs, and 7-segment displays *HEX3-0* to display the hexadecimal count as your circuit operates. Make the necessary pin assignments and compile the circuit.
4. Implement your circuit on the DE2 board and test its functionality by operating the implemented switches.
5. Implement a 4-bit version of your circuit and use the Quartus II RTL Viewer to see how Quartus II software synthesized your circuit. What are the differences in comparison with Figure 8?

Part II

Simplify your VHDL code so that the counter specification is based on the VHDL statement

$$\mathrm{Q} <= \mathrm{Q} + 1;$$

Compile a 16-bit version of this counter and compare the number of LEs needed and the *Fmax* that is attainable. Use the RTL Viewer to see the structure of this implementation and comment on the differences with the design from Part I.

Part III

Use an LPM from the Library of Parameterized modules to implement a 16-bit counter. Choose the LPM options to be consistent with the above design, i.e. with enable and synchronous clear. How does this version compare with the previous designs?

Part IV

Design and implement a circuit that successively flashes digits 0 through 9 on the 7-segment display $HEX0$. Each digit should be displayed for about one second. Use a counter to determine the one-second intervals. The counter should be incremented by the 50-MHz clock signal provided on the DE2 board. Do not derive any other clock signals in your design–make sure that all flip-flops in your circuit are clocked directly by the 50 MHz clock signal.

Part V

Design and implement a circuit that displays the word HELLO, in ticker tape fashion, on the eight 7-segment displays $HEX7-0$. Make the letters move from right to left in intervals of about one second. The patterns that should be displayed in successive clock intervals are given in Table 1.

Clock cycle	Displayed pattern							
0				H	E	L	L	O
1			H	E	L	L	O	
2		H	E	L	L	O		
3	H	E	L	L	O			
4	E	L	L	O				H
5	L	L	O				H	E
6	L	O				H	E	L
7	O				H	E	L	L
8				H	E	L	L	O
...	and so on							

Table 1. Scrolling the word HELLO in ticker-tape fashion.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--
-- inputs:
-- KEY0: manual clock
-- SW0: active low reset
-- SW1: enable signal for the counter
--
-- outputs:
-- HEX0 - HEX3: hex segment displays
ENTITY part1 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX3, HEX2, HEX1, HEX0      : OUT STD_LOGIC_VECTOR(0 TO 6));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT ToggleFF
      PORT (   T, Clock, Resetn     : IN  STD_LOGIC;
               Q                    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn : STD_LOGIC;
   SIGNAL Count, Enable : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- 16-bit counter based on T-flip flops
   Clock <= KEY(0);
   Resetn <= SW(0);

   Enable(0) <= SW(1);
   TFF0: ToggleFF PORT MAP (Enable(0), Clock, Resetn, Count(0));
   Enable(1) <= Count(0) AND Enable(0);
   TFF1: ToggleFF PORT MAP (Enable(1), Clock, Resetn, Count(1));
   Enable(2) <= Count(1) AND Enable(1);
   TFF2: ToggleFF PORT MAP (Enable(2), Clock, Resetn, Count(2));
   Enable(3) <= Count(2) AND Enable(2);
   TFF3: ToggleFF PORT MAP (Enable(3), Clock, Resetn, Count(3));
   Enable(4) <= Count(3) AND Enable(3);
   TFF4: ToggleFF PORT MAP (Enable(4), Clock, Resetn, Count(4));
   Enable(5) <= Count(4) AND Enable(4);
   TFF5: ToggleFF PORT MAP (Enable(5), Clock, Resetn, Count(5));
   Enable(6) <= Count(5) AND Enable(5);
   TFF6: ToggleFF PORT MAP (Enable(6), Clock, Resetn, Count(6));
   Enable(7) <= Count(6) AND Enable(6);
   TFF7: ToggleFF PORT MAP (Enable(7), Clock, Resetn, Count(7));
   Enable(8) <= Count(7) AND Enable(7);
   TFF8: ToggleFF PORT MAP (Enable(8), Clock, Resetn, Count(8));
   Enable(9) <= Count(8) AND Enable(8);
   TFF9: ToggleFF PORT MAP (Enable(9), Clock, Resetn, Count(9));
   Enable(10) <= Count(9) AND Enable(9);
   TFF10: ToggleFF PORT MAP (Enable(10), Clock, Resetn, Count(10));
   Enable(11) <= Count(10) AND Enable(10);
   TFF11: ToggleFF PORT MAP (Enable(11), Clock, Resetn, Count(11));
   Enable(12) <= Count(11) AND Enable(11);
   TFF12: ToggleFF PORT MAP (Enable(12), Clock, Resetn, Count(12));
   Enable(13) <= Count(12) AND Enable(12);
   TFF13: ToggleFF PORT MAP (Enable(13), Clock, Resetn, Count(13));
   Enable(14) <= Count(13) AND Enable(13);
   TFF14: ToggleFF PORT MAP (Enable(14), Clock, Resetn, Count(14));
   Enable(15) <= Count(14) AND Enable(14);
   TFF15: ToggleFF PORT MAP (Enable(15), Clock, Resetn, Count(15));
```

```
   -- drive the displays
   digit3: hex7seg PORT MAP (Count(15 DOWNTO 12), HEX3);
   digit2: hex7seg PORT MAP (Count(11 DOWNTO 8), HEX2);
   digit1: hex7seg PORT MAP (Count(7 DOWNTO 4), HEX1);
   digit0: hex7seg PORT MAP (Count(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- T Flip-flop
ENTITY ToggleFF IS
   PORT (   T, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END ToggleFF;

ARCHITECTURE Behavior OF ToggleFF IS
   SIGNAL T_out : STD_LOGIC;
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            T_out <= '0';
         ELSIF (T = '1') THEN
            T_out <= NOT T_out;
         END IF;
      END IF;
   END PROCESS;
   Q <= T_out;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
```

```
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--
-- inputs:
-- KEY0: manual clock
-- SW0: active low reset
-- SW1: enable signal for the counter
--
-- outputs:
-- HEX0: hex segment display
ENTITY part1 IS
   PORT (   SW       : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY      : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX0     : OUT STD_LOGIC_VECTOR(0 TO 6));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT ToggleFF
      PORT (   T, Clock, Resetn     : IN  STD_LOGIC;
               Q                    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn : STD_LOGIC;
   SIGNAL Count, Enable : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   -- 4-bit counter based on T-flip flops
   Clock <= KEY(0);
   Resetn <= SW(0);

   Enable(0) <= SW(1);
   TFF0: ToggleFF PORT MAP (Enable(0), Clock, Resetn, Count(0));
   Enable(1) <= Count(0) AND Enable(0);
   TFF1: ToggleFF PORT MAP (Enable(1), Clock, Resetn, Count(1));
   Enable(2) <= Count(1) AND Enable(1);
   TFF2: ToggleFF PORT MAP (Enable(2), Clock, Resetn, Count(2));
   Enable(3) <= Count(2) AND Enable(2);
   TFF3: ToggleFF PORT MAP (Enable(3), Clock, Resetn, Count(3));

   -- drive the displays
   digit0: hex7seg PORT MAP (Count(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- T Flip-flop
ENTITY ToggleFF IS
   PORT (   T, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END ToggleFF;

ARCHITECTURE Behavior OF ToggleFF IS
   SIGNAL T_out : STD_LOGIC;
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            T_out <= '0';
         ELSIF (T = '1') THEN
            T_out <= NOT T_out;
```

```
         END IF;
      END IF;
   END PROCESS;
   Q <= T_out;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
--
-- inputs:
-- KEY0: manual clock
-- SW0: active low reset
-- SW1: enable signal for the counter
--
-- outputs:
-- HEX0 - HEX3: hex segment displays
ENTITY part2 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX3, HEX2, HEX1, HEX0      : OUT STD_LOGIC_VECTOR(0 TO 6));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn, Enable : STD_LOGIC;
   SIGNAL Count : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- 16-bit counter based on T-flip flops
   Clock <= KEY(0);
   Resetn <= SW(0);
   Enable <= SW(1);

   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= (OTHERS => '0');
         ELSIF (Enable = '1') THEN
            Count <= Count + '1';
         END IF;
      END IF;
   END PROCESS;

   -- drive the displays
   digit3: hex7seg PORT MAP (Count(15 DOWNTO 12), HEX3);
   digit2: hex7seg PORT MAP (Count(11 DOWNTO 8), HEX2);
   digit1: hex7seg PORT MAP (Count(7 DOWNTO 4), HEX1);
   digit0: hex7seg PORT MAP (Count(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
```

```
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
--
-- inputs:
-- KEY0: manual clock
-- SW0: active low reset
-- SW1: enable signal for the counter
--
-- outputs:
-- HEX0 - HEX3: hex segment displays
ENTITY part2 IS
   PORT (   SW       : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY      : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX0     : OUT STD_LOGIC_VECTOR(0 TO 6));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn, Enable : STD_LOGIC;
   SIGNAL Count : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   Enable <= SW(1);

   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= (OTHERS => '0');
         ELSIF (Enable = '1') THEN
            Count <= Count + '1';
         END IF;
      END IF;
   END PROCESS;

   -- drive the displays
   digit0: hex7seg PORT MAP (Count(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
```

```
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
--
-- inputs:
-- KEY0: manual clock
-- SW0: active low reset
-- SW1: enable signal for the counter
--
-- outputs:
-- HEX0 - HEX3: hex segment displays
ENTITY part3 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX3, HEX2, HEX1, HEX0      : OUT STD_LOGIC_VECTOR(0 TO 6));
END part3;

ARCHITECTURE Behavior OF part3 IS
   COMPONENT lpm_count
      PORT (   clock    : IN STD_LOGIC ;
               cnt_en   : IN STD_LOGIC ;
               sclr     : IN STD_LOGIC ;
               q        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn, Enable : STD_LOGIC;
   SIGNAL Count : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- 16-bit counter based on T-flip flops
   Clock <= KEY(0);
   Resetn <= SW(0);
   Enable <= SW(1);

   -- 16-bit counter
   -- lpm_count (clock, cnt_en, sclr, q)
   U1: lpm_count PORT MAP (Clock, Enable, NOT Resetn, Count);

   -- drive the displays
   digit3: hex7seg PORT MAP (Count(15 DOWNTO 12), HEX3);
   digit2: hex7seg PORT MAP (Count(11 DOWNTO 8), HEX2);
   digit1: hex7seg PORT MAP (Count(7 DOWNTO 4), HEX1);
   digit0: hex7seg PORT MAP (Count(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
```

```
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
-- uses a 1-digit bcd counter enabled at 1Hz
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY part4 IS
   PORT (   CLOCK_50    : IN  STD_LOGIC;
            HEX0        : OUT STD_LOGIC_VECTOR(0 TO 6));
END part4;

ARCHITECTURE Behavior OF part4 IS
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL bcd : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL digit_flipper : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

   -- Create a 1Hz 4-bit counter
   -- First, a large counter to produce a 1 second (approx) enable
   -- from the 50 MHz Clock
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- four-bit counter that uses a slow enable for selecting digit
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         IF (slow_count = 0) THEN
            IF (digit_flipper = "1001") THEN
               digit_flipper <= "0000";
            ELSE
               digit_flipper <= digit_flipper + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;

   bcd <= digit_flipper;
   -- drive the display through a 7-seg decoder
   digit_0: bcd7seg PORT MAP (bcd, HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
```

```
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- scans the word HELLO across the 7-seg displays. KEY3 causes a reset.
ENTITY part5 IS
   PORT (   CLOCK_50                  : IN  STD_LOGIC;
            KEY                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4,
               HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

-- KEY[3] is resetn
ARCHITECTURE Behavior OF part5 IS
   COMPONENT hello7seg
      PORT (   char     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL seg7_7, seg7_6, seg7_5, seg7_4, seg7_3, seg7_2, seg7_1, seg7_0 :
      STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL slow_count : STD_LOGIC_VECTOR(23 DOWNTO 0);
   SIGNAL digit_flipper : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   -- a large counter to produce a slow enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;
   -- 3-bit counter that uses a slow enable for selecting digit
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         IF (KEY(3) = '0') THEN
            digit_flipper <= "000";
         ELSIF (slow_count = 0) THEN
            digit_flipper <= digit_flipper + '1';
         END IF;
      END IF;
   END PROCESS;

   seg7_7 <= digit_flipper;
   seg7_6 <= digit_flipper + "001";
   seg7_5 <= digit_flipper + "010";
   seg7_4 <= digit_flipper + "011";
   seg7_3 <= digit_flipper + "100";
   seg7_2 <= digit_flipper + "101";
   seg7_1 <= digit_flipper + "110";
   seg7_0 <= digit_flipper + "111";

   -- drive the display through a 7-seg decoder designed specifically for letters
   -- 'h' 'e' 'l' 'o' and ' '
   digit_7: hello7seg PORT MAP (seg7_7, HEX7);
   digit_6: hello7seg PORT MAP (seg7_6, HEX6);
   digit_5: hello7seg PORT MAP (seg7_5, HEX5);
   digit_4: hello7seg PORT MAP (seg7_4, HEX4);
   digit_3: hello7seg PORT MAP (seg7_3, HEX3);
   digit_2: hello7seg PORT MAP (seg7_2, HEX2);
   digit_1: hello7seg PORT MAP (seg7_1, HEX1);
   digit_0: hello7seg PORT MAP (seg7_0, HEX0);
END Behavior;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

ENTITY hello7seg IS
   PORT (   char     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hello7seg;

ARCHITECTURE Behavior OF hello7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (char)
   BEGIN
      CASE char IS
         WHEN "000" => display <= "1001000";    -- 'H'
         WHEN "001" => display <= "0110000";    -- 'E'
         WHEN "010" => display <= "1110001";    -- 'L'
         WHEN "011" => display <= "1110001";    -- 'L'
         WHEN "100" => display <= "0000001";    -- 'O'
         WHEN "101" => display <= "1111111";    -- ' '
         WHEN "110" => display <= "1111111";    -- ' '
         WHEN OTHERS => display <= "1111111";   -- ' '
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- scans the word HELLO across the 7-seg displays. KEY(3) causes a reset.
-- KEY(0) loads the proper counter values
ENTITY part5 IS
   PORT (   CLOCK_50                  : IN  STD_LOGIC;
            KEY                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4,
               HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT upcount
      PORT (   R                       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               Resetn, Clock, L, E     : IN  STD_LOGIC;
               Q                       : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
   END COMPONENT;
   COMPONENT hello7seg
      PORT (   char     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL seg7_7, seg7_6, seg7_5, seg7_4, seg7_3, seg7_2, seg7_1, seg7_0 :
      STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL slow_count : STD_LOGIC_VECTOR(23 DOWNTO 0);
   SIGNAL Enable : STD_LOGIC;
BEGIN
   -- a large counter to produce a slow enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   Enable <= '1' WHEN (slow_count = 0) ELSE '0';
   -- upcount (R, Resetn, Clock, L, E, Q)
   up7: upcount PORT MAP ("000", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_7);
   up6: upcount PORT MAP ("001", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_6);
   up5: upcount PORT MAP ("010", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_5);
   up4: upcount PORT MAP ("011", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_4);
   up3: upcount PORT MAP ("100", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_3);
   up2: upcount PORT MAP ("101", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_2);
   up1: upcount PORT MAP ("110", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_1);
   up0: upcount PORT MAP ("111", KEY(3), CLOCK_50, NOT KEY(0), Enable, seg7_0);

   -- drive the display through a 7-seg decoder designed specifically for letters
   -- 'h' 'e' 'l' 'o' and ' '
   digit_7: hello7seg PORT MAP (seg7_7, HEX7);
   digit_6: hello7seg PORT MAP (seg7_6, HEX6);
   digit_5: hello7seg PORT MAP (seg7_5, HEX5);
   digit_4: hello7seg PORT MAP (seg7_4, HEX4);
   digit_3: hello7seg PORT MAP (seg7_3, HEX3);
   digit_2: hello7seg PORT MAP (seg7_2, HEX2);
   digit_1: hello7seg PORT MAP (seg7_1, HEX1);
   digit_0: hello7seg PORT MAP (seg7_0, HEX0);
END Behavior;

-- three-bit counter with parallel load and enable
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
```

```
-- scans the word HELLO across the 7-seg displays. KEY(3) causes a reset.
ENTITY upcount IS
   PORT (   R                      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            Resetn, Clock, L, E    : IN  STD_LOGIC;
            Q                      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END upcount;

ARCHITECTURE Behavior OF upcount IS
   SIGNAL Count : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= "000";
         ELSIF (L = '1') THEN
            Count <= R;
         ELSIF (E = '1') THEN
            Count <= Count + '1';
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hello7seg IS
   PORT (   char     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hello7seg;

ARCHITECTURE Behavior OF hello7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (char)
   BEGIN
      CASE char IS
         WHEN "000" => display <= "1001000";    -- 'H'
         WHEN "001" => display <= "0110000";    -- 'E'
         WHEN "010" => display <= "1110001";    -- 'L'
         WHEN "011" => display <= "1110001";    -- 'L'
         WHEN "100" => display <= "0000001";    -- 'O'
         WHEN "101" => display <= "1111111";    -- ' '
         WHEN "110" => display <= "1111111";    -- ' '
         WHEN OTHERS => display <= "1111111";   -- ' '
      END CASE;
   END PROCESS;
END Behavior;
```

Laboratory Exercise 5

Clocks and Timers

This is an exercise in implementing and using a real-time clock.

Part I

Implement a 3-digit BCD counter. Display the contents of the counter on the 7-segment displays, *HEX2−0*. Derive a control signal, from the 50-MHz clock signal provided on the Altera DE2 board, to increment the contents of the counter at one-second intervals. Use the pushbutton switch KEY_0 to reset the counter to 0.

1. Create a new Quartus II project which will be used to implement the desired circuit on the DE2 board.
2. Write a VHDL file that specifies the desired circuit.
3. Include the VHDL file in your project and compile the circuit.
4. Simulate the designed circuit to verify its functionality.
5. Assign the pins on the FPGA to connect to the 7-segment displays and the pushbutton switch, as indicated in the User Manual for the DE2 board.
6. Recompile the circuit and download it into the FPGA chip.
7. Verify that your circuit works correctly by observing the display.

Part II

Design and implement a circuit on the DE2 board that acts as a time-of-day clock. It should display the hour (from 0 to 23) on the 7-segment displays *HEX7−6*, the minute (from 0 to 60) on *HEX5−4* and the second (from 0 to 60) on *HEX3−2*. Use the switches SW_{15-0} to preset the hour and minute parts of the time displayed by the clock.

Part III

Design and implement on the DE2 board a reaction-timer circuit. The circuit is to operate as follows:

1. The circuit is reset by pressing the pushbutton switch KEY_0.
2. After an elapsed time, the red light labeled $LEDR_0$ turns on and a four-digit BCD counter starts counting in intervals of milliseconds. The amount of time in seconds from when the circuit is reset until $LEDR_0$ is turned on is set by switches SW_{7-0}.
3. A person whose reflexes are being tested must press the pushbutton KEY_3 as quickly as possible to turn the LED off and freeze the counter in its present state. The count which shows the reaction time will be displayed on the 7-segment displays *HEX2-0*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- A 3-digit BCD counter.
ENTITY part1 IS
   PORT (   CLOCK_50                  : IN  STD_LOGIC;
            KEY                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX3, HEX2, HEX1, HEX0    : OUT STD_LOGIC_VECTOR(0 TO 6));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL bcd_0, bcd_1, bcd_2 : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   -- Create a 1Hz 4-bit counter
   -- First, a large counter to produce a 1 second (approx) enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- 3-digit BCD counter that uses a slow enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         IF (KEY(3) = '0') THEN
            bcd_0 <= "0000";
            bcd_1 <= "0000";
            bcd_2 <= "0000";
         ELSIF (slow_count = 0) THEN
            IF (bcd_0 = "1001") THEN
               bcd_0 <= "0000";
               IF (bcd_1 = "1001") THEN
                  bcd_1 <= "0000";
                  IF (bcd_2 = "1001") THEN
                     bcd_2 <= "0000";
                  ELSE
                     bcd_2 <= bcd_2 + '1';
                  END IF;
               ELSE
                  bcd_1 <= bcd_1 + '1';
               END IF;
            ELSE
               bcd_0 <= bcd_0 + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;

   -- drive the displays
   digit2: bcd7seg PORT MAP (bcd_2, HEX2);
   digit1: bcd7seg PORT MAP (bcd_1, HEX1);
   digit0: bcd7seg PORT MAP (bcd_0, HEX0);
   -- blank the adjacent display
   digit3: bcd7seg PORT MAP ("1111", HEX3);
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- Real time settable clock. Set SW(15 DOWNTO 8) switches to 2-digit BCD number
-- representing hours. Set SW(7 DOWNTO 0) to 2-digit number representing minutes.
-- Load initial time by pressing KEY(3).
ENTITY part2 IS
   PORT (   CLOCK_50                  : IN  STD_LOGIC;
            SW                        : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            KEY                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1,
               HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT modulo
   PORT (   R, M                    : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            Clock, Resetn, L, E     : IN  STD_LOGIC;
            Q                       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
   END COMPONENT;
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL hr_1, hr_0, min_1, min_0, sec_1, sec_0 : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL mod_hr_0 : STD_LOGIC_VECTOR(3 DOWNTO 0); -- used to change hour (LSB) from
                                                   -- 19 to 20 or 23 to 00
   SIGNAL E_sec_0, E_sec_1, E_min_0, E_min_1, E_hr_0, E_hr_1 : STD_LOGIC;
BEGIN
   -- Create a 1Hz 4-bit counter
   -- First, a large counter to produce a 1 second (approx) enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- 6-digit clock display
   -- modulo (R, M, CLOCK_50, Resetn, L, E, Q)
   E_sec_0 <= '1' WHEN (slow_count = 0) ELSE '0';
   seconds_0: modulo PORT MAP ("0000", "1001", CLOCK_50, KEY(3), '0', E_sec_0, sec_0);

   E_sec_1 <= '1' WHEN (sec_0 = 9) AND (E_sec_0 = '1') ELSE '0';
   seconds_1: modulo PORT MAP ("0000", "0101", CLOCK_50, KEY(3), '0', E_sec_1, sec_1);

   E_min_0 <= '1' WHEN (sec_1 = 5) AND (E_sec_1 = '1') ELSE '0';
   minutes_0: modulo PORT MAP (SW(3 DOWNTO 0), "1001", CLOCK_50, KEY(3), NOT KEY(0),
      E_min_0, min_0);
   E_min_1 <= '1' WHEN (min_0 = 9) AND (E_min_0 = '1') ELSE '0';
   minutes_1: modulo PORT MAP (SW(7 DOWNTO 4), "0101", CLOCK_50, KEY(3), NOT KEY(0),
      E_min_1, min_1);

   E_hr_0 <= '1' WHEN (min_1 = 5) AND (E_min_1 = '1') ELSE '0';
   mod_hr_0 <= "0011" WHEN (hr_1 = 2) ELSE "1001";
   hour_0: modulo PORT MAP (SW(11 DOWNTO 8), mod_hr_0, CLOCK_50, KEY(3), NOT KEY(0),
      E_hr_0, hr_0);
   E_hr_1 <= '1' WHEN ( ((hr_1 = 2) AND (hr_0 = 3)) OR (hr_0 = 9) ) AND (E_hr_0 = '1')
      ELSE '0';
   hour_1: modulo PORT MAP (SW(15 DOWNTO 12), "0010", CLOCK_50, KEY(3), NOT KEY(0),
      E_hr_1, hr_1);
```

```
   -- drive the displays
   digit7: bcd7seg PORT MAP (hr_1, HEX7);
   digit6: bcd7seg PORT MAP (hr_0, HEX6);
   digit5: bcd7seg PORT MAP (min_1, HEX5);
   digit4: bcd7seg PORT MAP (min_0, HEX4);
   digit3: bcd7seg PORT MAP (sec_1, HEX3);
   digit2: bcd7seg PORT MAP (sec_0, HEX2);
   -- blank the adjacent display
   digit1: bcd7seg PORT MAP ("1111", HEX1);
   digit0: bcd7seg PORT MAP ("1111", HEX0);

END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY modulo IS
   PORT (   R, M                    : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            Clock, Resetn, L, E     : IN  STD_LOGIC;
            Q                       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END modulo;

ARCHITECTURE Behavior OF modulo IS
   SIGNAL Count : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= "0000";
         ELSIF (L = '1') THEN
            Count <= R;
         ELSIF (E = '1') THEN
            IF (Count = M) THEN
               Count <= "0000";
            ELSE
               Count <= Count + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
```

```
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

-- Press KEY(0) to reset. After a delay (#seconds set by the SW(7 DOWNTO 0)
-- switches), LEDR(0) turns on and the timer starts. Stop the timer by
-- pressing KEY(3)
ENTITY part3 IS
   PORT (   CLOCK_50                  : IN  STD_LOGIC;
            SW                        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            KEY                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            LEDR                      : OUT STD_LOGIC_VECTOR(0 TO 0);
            HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1,
               HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part3;

ARCHITECTURE Behavior OF part3 IS
   COMPONENT regne
      PORT (   R, Clock, Resetn, E     : IN  STD_LOGIC;
               Q                       : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL slow_count : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL second : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL sec_1, sec_0, min_1, min_0 : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL pseudo : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL start, pseudo_0 : STD_LOGIC;
BEGIN
   -- regne (R, Clock, Resetn, E, Q)
   pseudo_0 <= '1' WHEN pseudo = 0 ELSE '0';
   reg_start: regne PORT MAP ('1', CLOCK_50, KEY(3) AND KEY(0), pseudo_0, start);
   LEDR(0) <= start;

   -- A large counter to produce a 1 sec second (approx) enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         second <= second + '1';
      END IF;
   END PROCESS;

   -- A large counter to produce a 1 msec second (approx) enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- Pseudo random delay counter; loads SW switches and counts down
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         IF ( KEY(3) = '0' OR KEY(0) = '0') THEN
            pseudo <= SW;
         ELSIF (second = 0) THEN
            pseudo <= pseudo - '1';
         END IF;
      END IF;
   END PROCESS;
```

```
   -- 4-digit BCD counter that uses a slow enable
   PROCESS (CLOCK_50)
   BEGIN
      IF (CLOCK_50'EVENT AND CLOCK_50 = '1') THEN
         IF (KEY(0) = '0') THEN
            sec_0 <= "0000";
            sec_1 <= "0000";
            min_0 <= "0000";
            min_1 <= "0000";
         ELSIF ( (slow_count = 0) AND (start = '1') ) THEN
            IF (sec_0 = "1001") THEN
               sec_0 <= "0000";
               IF (sec_1 = "1001") THEN
                  sec_1 <= "0000";
                  IF (min_0 = "1001") THEN
                     min_0 <= "0000";
                     IF (min_1 = "1001") THEN
                        min_1 <= "0000";
                     ELSE
                        min_1 <= min_1 + '1';
                     END IF;
                  ELSE
                     min_0 <= min_0 + '1';
                  END IF;
               ELSE
                  sec_1 <= sec_1 + '1';
               END IF;
            ELSE
               sec_0 <= sec_0 + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;

   -- drive the displays
   -- blank the unused displays
   digit7: bcd7seg PORT MAP ("1111", HEX7);
   digit6: bcd7seg PORT MAP ("1111", HEX6);
   digit5: bcd7seg PORT MAP ("1111", HEX5);
   digit4: bcd7seg PORT MAP ("1111", HEX4);

   digit3: bcd7seg PORT MAP (min_1, HEX3);
   digit2: bcd7seg PORT MAP (min_0, HEX2);
   digit1: bcd7seg PORT MAP (sec_1, HEX1);
   digit0: bcd7seg PORT MAP (sec_0, HEX0);

END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY regne IS
   PORT (   R, Clock, Resetn, E     : IN  STD_LOGIC;
            Q                       : OUT STD_LOGIC);
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Q <= '0';
```

```
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

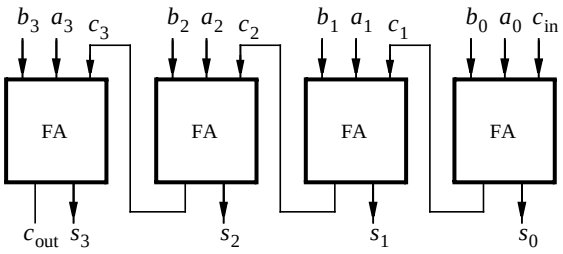
Laboratory Exercise 6

Adders, Subtractors, and Multipliers

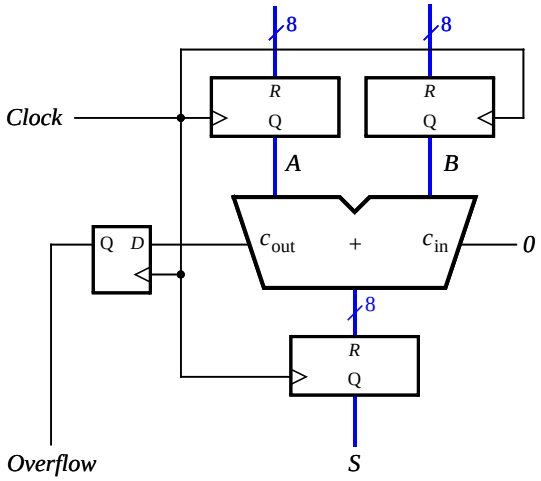
The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each type of circuit will be implemented in two ways: first by writing VHDL code that describes the required functionality, and second by making use of predefined subcircuits from Altera's library of parameterized modules (LPMs). The results produced for various implementations will be compared, both in terms of the circuit structure and its speed of operation.

Part I

Consider again the four-bit ripple-carry adder circuit that was used in lab exercise 2; a diagram of this circuit is reproduced in Figure 1a. You are to create an 8-bit version of the adder and include it in the circuit shown in Figure 1b. Your circuit should be designed to support signed numbers in 2's-complement form, and the *Overflow* output should be set to 1 whenever the sum produced by the adder does not provide the correct signed value. Perform the steps shown below.



a) Four-bit ripple-carry adder circuit



b) Eight-bit registered adder circuit

Figure 1. An 8-bit signed adder with registered inputs and outputs.

1. Make a new Quartus II project and write VHDL code that describes the circuit in Figure 1b. Use the circuit structure in Figure 1a to describe your adder.

2. Include the required input and output ports in your project to implement the adder circuit on the DE2 board. Connect the inputs A and B to switches SW_{15-8} and SW_{7-0}, respectively. Use KEY_0 as an active-low asynchronous reset input, and use KEY_1 as a manual clock input. Display the sum outputs of the adder on the red $LEDR_{7-0}$ lights and display the overflow output on the green $LEDG_8$ light. The hexadecimal values of A and B should be shown on the displays *HEX7-6* and *HEX5-4*, and the hexadecimal value of S should appear on *HEX1-0*.

3. Compile your code and use timing simulation to verify the correct operation of the circuit. Once the simulation works properly, download the circuit onto the DE2 board and test it by using different values of A and B. Be sure to check for proper functionality of the *Overflow* output.

4. Open the Quartus II Compilation Report and examine the results reported by the Timing Analyzer. What is the maximum operating frequency, *fmax*, of your circuit? What is the longest path in the circuit in terms of delay?

Part II

Modify your circuit from Part I so that it can perform both addition and subtraction of eight-bit numbers. Use switch SW_{16} to specify whether addition or subtraction should be performed. Connect the other switches, lights, and displays as described for Part I.

1. Simulate your adder/subtractor circuit to show that it functions properly, and then download it onto the DE2 board and test it by using different switch settings.

2. Open the Quartus II Compilation Report and examine the results reported by the Timing Analyzer. What is the fmax of your circuit? What is the longest path in the circuit in terms of delay?

Part III

Repeat Part I using the predefined adder circuit called *lpm_add_sub*, instead of your ripple-carry adder structure from Figure 1. The *lpm_add_sub* module can be found in Altera's library of parameterized modules (LPMs), which is provided as part of the Quartus II system. The procedure for using these predefined modules in Quartus II projects is described in the tutorial *Using Library Modules in VHDL Designs*, which is available on the *DE2 System CD* and in the University Program section of Altera's web site.

1. Configure the *lpm_add_sub* module so that it performs only addition, to make the functionality comparable to Part I. Store your configuration of the *lpm_add_sub* module in the file *lpm_add8.v*. After instantiating this module in your VHDL code, compile the project and use the Quartus II Chip Editor tool to examine some of the details of the implemented circuit.

   One way to examine the adder subcircuit using the Chip Editor tool is illustrated in Figure 2. In the Quartus II Project Navigator window right-click on the part of your circuit hierarchy that represents the *lpm_add8* subcircuit, and select the command Locate $>$ Locate in Chip Editor. This opens the Chip Editor window shown in Figure 3. The logic elements in the Cyclone II FPGA that are used to implement the adder are highlighted in blue in the Chip Editor tool. Position your mouse pointer over any of these logic elements and double-click to open the Resource Property Editor window displayed in Figure 4. In the box labeled Node name you can select any of the nine logic elements that implement the adder module. The Resource Property Editor allows you to examine the contents of a logic element and to see how one logic element is connected others.

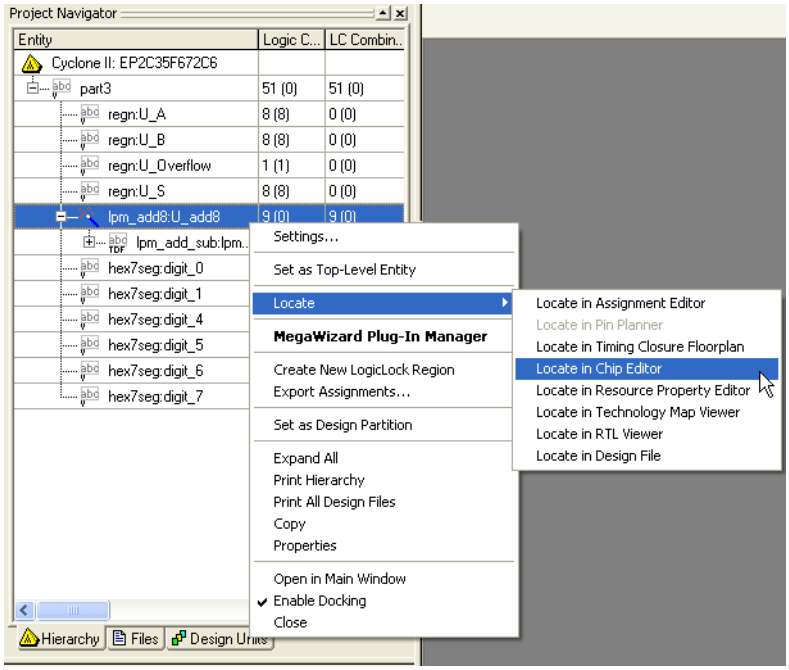


Figure 2. Locating the eight-bit adder in the Chip Editor tool.

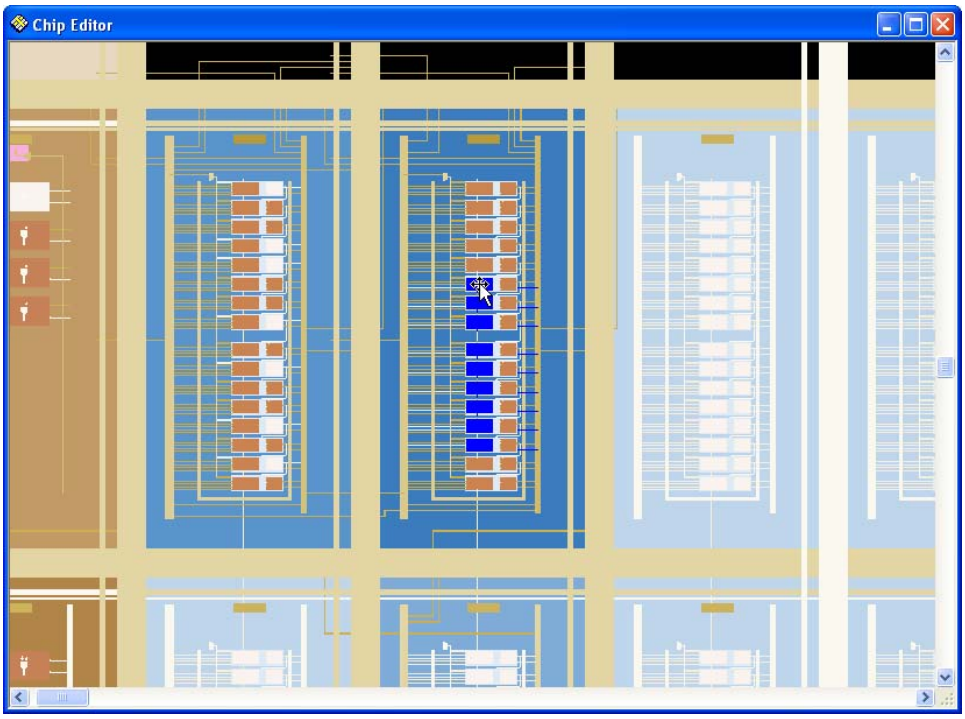


Figure 3. The highlighted logic elements for the eight-bit adder.

Using the tools described above, and referencing the Data Sheet information for the Cyclone II FPGA, describe the eight-bit adder circuit implemented with the *lpm_add_sub* module.

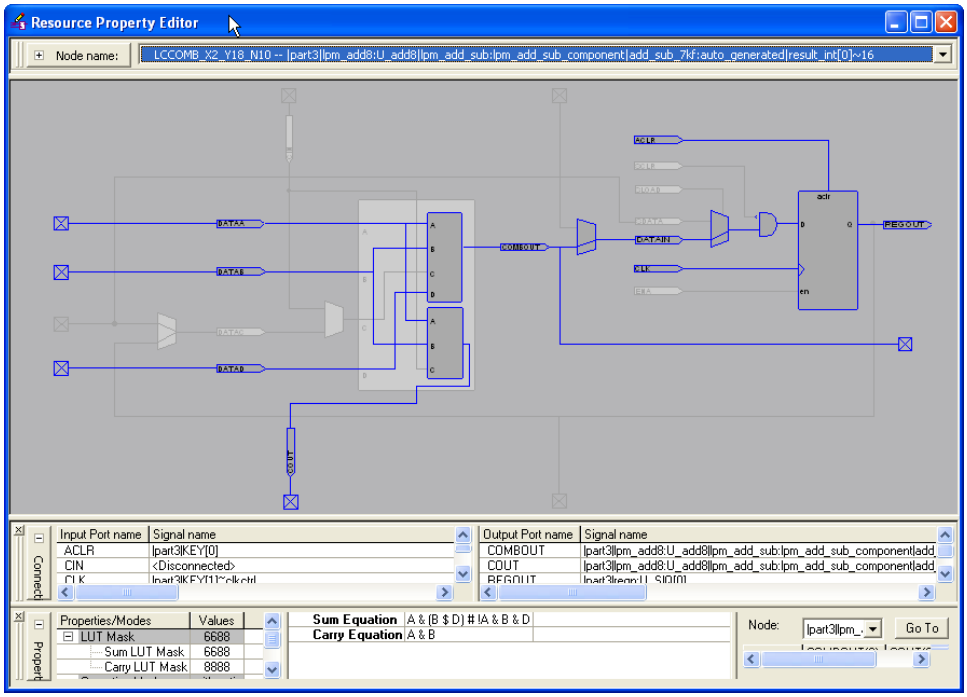


Figure 4. Examining details in a logic element using the Resource Property Editor.

2. Open the Quartus II Compilation Report and and compare the fmax of your adder circuit with the one designed in Part I. Discuss any differences in performance that are observed.

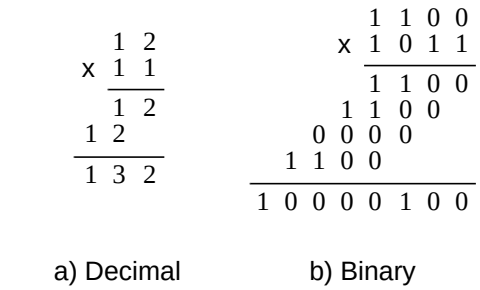
Part IV

Repeat Part II using the predefined adder circuit called *lpm_add_sub*, instead of your adder-subtractor circuit based on Figure 1.

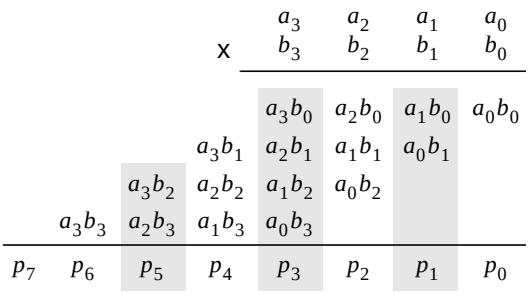
Comment briefly on the circuit structure obtained using the LPM module, and compare the fmax of this circuit to the one from Part II. Describe how the *lpm_add_sub* module implements the *Overflow* signal.

Part V

Figure 5a gives an example of the traditional paper-and-pencil multiplication $P = A \times B$, where $A = 12$ and $B = 11$. We need to add two summands that are shifted versions of A to form the product $P = 132$. Part b of the figure shows the same example using four-bit binary numbers. Since each digit in B is either 1 or 0, the summands are either shifted versions of A or 0000. Figure 5c shows how each summand can be formed by using the Boolean AND operation of A with the appropriate bit in B.



a) Decimal b) Binary



c) Implementation

Figure 5. Multiplication of binary numbers.

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 6. Because of its regular structure, this type of multiplier circuit is usually called an *array multiplier*. The shaded areas in the circuit correspond to the shaded columns in Figure 5c. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

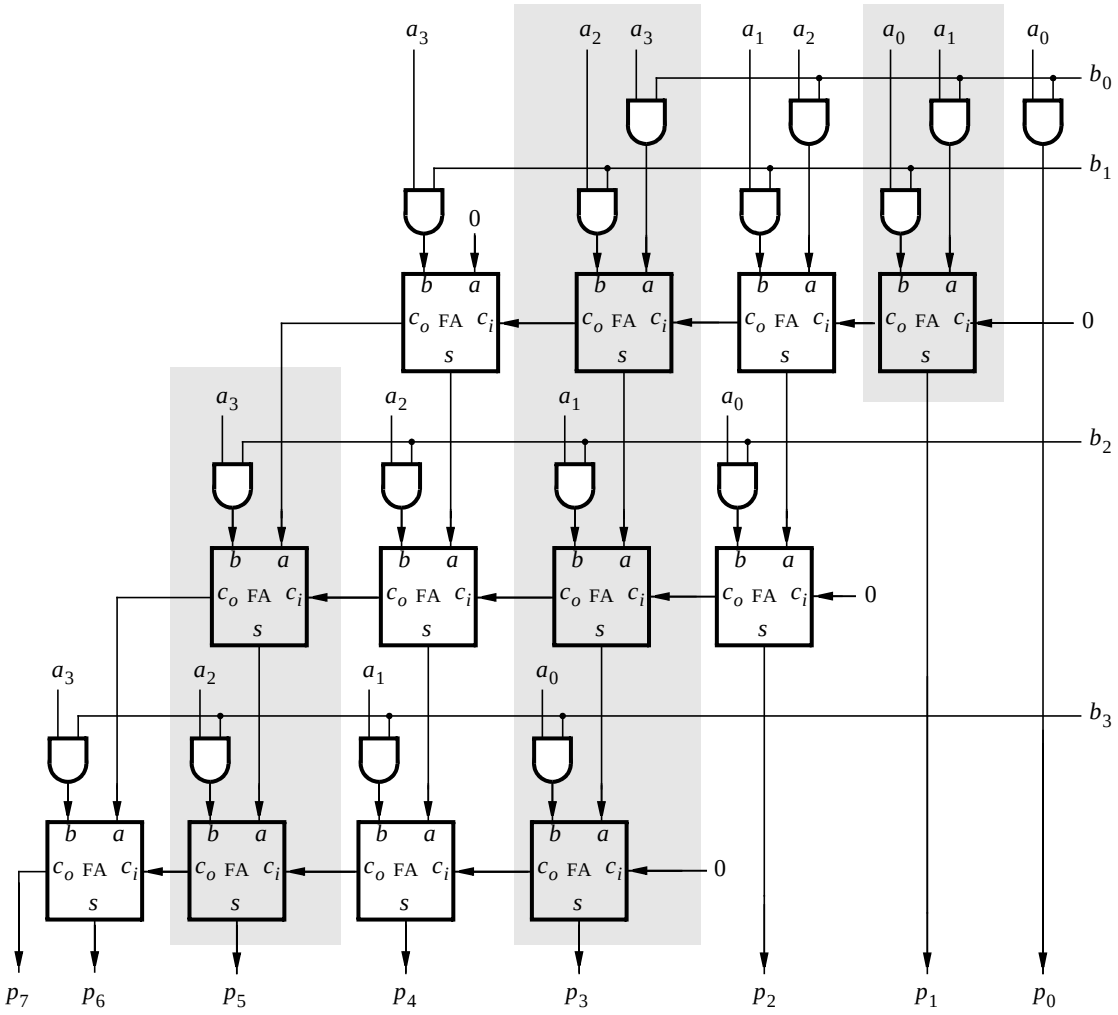


Figure 6. An array multiplier circuit.

Use the following steps to implement the array multiplier circuit:

1. Create a new Quartus II project which will be used to implement the desired circuit on the Altera DE2 board.
2. Generate the required VHDL file, include it in your project, and compile the circuit.
3. Use functional simulation to verify that your code is correct.
4. Augment your design to use switches SW_{11-8} to represent the number A and switches SW_{3-0} to represent B. The hexadecimal values of A and B are to be displayed on the 7-segment displays *HEX6* and *HEX4*, respectively. The result $P = A \times B$ is to be displayed on *HEX1* and *HEX0*.
5. Assign the pins on the FPGA to connect to the switches and 7-segment displays, as indicated in the User Manual for the DE2 board.
6. Recompile the circuit and download it into the FPGA chip.
7. Test the functionality of your design by toggling the switches and observing the 7-segment displays.

Part VI

Extend your multiplier to multiply 8-bit numbers and produce a 16-bit product. Use switches SW_{15-8} to represent the number A and switches SW_{7-0} to represent B. The hexadecimal values of A and B are to be displayed on the 7-segment displays *HEX7−6* and *HEX5−4*, respectively. The result $P = A \times B$ is to be displayed on *HEX3−0*. Add registers to your circuit to store the values of A, B, and the product P, using a similar structure as shown for the registered adder in Figure 1.

After successfully compiling and testing your multiplier circuit, examine the results produced by the Quartus II Timing Analyzer to determine the fmax of your circuit. What is the longest path in terms of delay between registers?

Part VII

Change your VHDL code to implement the 8 x 8 multiplier by using the *lpm_mult* module from the library of parameterized modules in the Quartus II system. Complete the design steps above. Compare the results in terms of the number of logic elements (LEs) needed and the circuit fmax.

Part VIII

It many applications of digital circuits it is useful to be able to perform some number of multiplications and then produce a summation of the results. For this part of the exercise you are to design a circuit that performs the calculation

$$S = (A \times B) + (C \times D)$$

The inputs A, B, C, and D are eight-bit unsigned numbers, and S provides a 16-bit result. Your circuit should also provide a carry-out signal, C_{out}. All of the inputs and outputs of the circuit should be registered, similar to the structure shown in Figure 1*b*.

1. Create a new Quartus II project which will be used to implement the desired circuit on the Altera DE2 board. Use the *lpm_mult* and *lpm_add_sub* modules to realize the multipliers and adders in your design.

2. Connect the inputs A and C to switches SW_{15-8} and connect the inputs B and D to switches SW_{7-0}. Use switch SW_{16} to select between these two sets of inputs: A, B or C, D. Also, use the switch SW_{17} as a *write enable* (WE) input. Setting *WE* to 1 should allow data to be loaded into the input registers when an active clock edge occurs, while setting *WE* to 0 should prevent loading of these registers.

3. Use KEY_0 as an active-low asynchronous reset input, and use KEY_1 as a manual clock input.

4. Display the hexadecimal value of either A or C, as selected by SW_{16}, on displays *HEX7-6* and display either B or D on *HEX5-4*. The sum S should be shown on *HEX3-0*, and the C_{out} signal should appear on $LEDG_8$.

5. Compile your code and use either functional or timing simulation to verify that your circuit works properly. Then download the circuit onto the DE2 board and test its operation.

6. It is often necessary to ensure that a digital circuit is able to meet certain speed requirements, such as a particular frequency of a signal applied to a clock input. Such requirements are provided to a CAD system in the form of *timing constraints*. The procedure for using timing constraints in the Quartus II CAD system is described in the tutorial *Timing Considerations with VHDL-Based Designs*, which is available on the *DE2 System CD* and in the University Program section of Altera's web site.

   For this exercise we are using a manual clock that is applied by a pushbutton switch, so no realistic timing requirements exist. But to demonstrate the design issues involved, assume that your circuit is required to operate with a clock frequency of 220 MHz. Enter this frequency as a timing constraint in the Quartus II software, and recompile your project. The Timing Analyzer should report that it is unable to meet the timing requirements due to the lengths of various register-to-register paths in the circuit. Examine the timing analysis report and describe briefly the timing violations observed.

7. One way to increase the speed of operation of a given circuit is to insert registers into the circuit in a way that shortens the lengths of its longest paths. This technique is referred to as *pipelining* a circuit, and the inserted registers are often called *pipeline registers*. Insert pipeline registers into your design between the multipliers and the adder. Recompile your project and discuss the results obtained.

Part IX

The Quartus II software includes a predesigned module called *altmult_add* that can perform calculations of the form $S = (A \times B) + (C \times D)$. Repeat Part VIII using this module instead of the *lpm_mult* and *lpm_add_sub* modules. Test your circuit using both simulation and by downloading the circuit onto the DE2 board.

Briefly describe how the implementation of your circuit differs when using the *altmult_add* module. Examine its performance both with and without the pipeline registers discussed in Part VIII.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Signed 8-bit ripple-carry adder S = A + B, with overflow detection
-- Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A, SW7-0 = B
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: LEDR7-0 shows S in binary form
--          HEX7-6 shows input A, HEX5-4 shows input B
--          HEX3-0 shows the output sum
--          LEDG8 shows overflow
ENTITY part1 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            LEDR                   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDG                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part1;

ARCHITECTURE Structure OF part1 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn    : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL A, B, S, S_reg : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C : STD_LOGIC_VECTOR(8 DOWNTO 1);     -- carries
   SIGNAL Clock, Resetn, Overflow, Overflow_reg : STD_LOGIC;
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   bit0: fa PORT MAP (A(0), B(0), '0', S(0), C(1));
   bit1: fa PORT MAP (A(1), B(1), C(1), S(1), C(2));
   bit2: fa PORT MAP (A(2), B(2), C(2), S(2), C(3));
   bit3: fa PORT MAP (A(3), B(3), C(3), S(3), C(4));
   bit4: fa PORT MAP (A(4), B(4), C(4), S(4), C(5));
   bit5: fa PORT MAP (A(5), B(5), C(5), S(5), C(6));
   bit6: fa PORT MAP (A(6), B(6), C(6), S(6), C(7));
   bit7: fa PORT MAP (A(7), B(7), C(7), S(7), C(8));
   -- Display the adder outputs
   LEDR <= S;

   -- instantiate regn (R, Clock, Resetn, Q);
   U_S: regn PORT MAP (S, Clock, Resetn, S_reg);
```

```
   -- check for Overflow
   Overflow <= C(8) XOR C(7);
   U_Overflow: flipflop PORT MAP (Overflow, Clock, Resetn, Overflow_reg);
   LEDG(8) <= Overflow_reg;

   -- drive the displays through 7-seg decoders
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   HEX3 <= "1111111";
   HEX2 <= "1111111";
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
   PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn     : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN -- asynchronous clear
            Q <= '0';
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= D;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Signed 8-bit ripple-carry adder/subtractor S = A +/- B, with overflow detection
-- Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A, SW7-0 = B
--          SW16 = Add_Subn
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: LEDR7-0 shows S in binary form
--          HEX7-6 shows input A, HEX5-4 shows input B
--          HEX3-0 shows the output sum
--          LEDG8 shows overflow
ENTITY part2 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(16 DOWNTO 0);
            LEDR                   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDG                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part2;

ARCHITECTURE Structure OF part2 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn    : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL A, B, S, S_reg : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C : STD_LOGIC_VECTOR(8 DOWNTO 1);     -- carries
   SIGNAL Clock, Resetn, Add_Subn, Overflow, Overflow_reg : STD_LOGIC;
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   Add_Subn <= SW(16);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   bit0: fa PORT MAP (A(0), B(0) XOR Add_Subn, Add_Subn, S(0), C(1));
   bit1: fa PORT MAP (A(1), B(1) XOR Add_Subn, C(1), S(1), C(2));
   bit2: fa PORT MAP (A(2), B(2) XOR Add_Subn, C(2), S(2), C(3));
   bit3: fa PORT MAP (A(3), B(3) XOR Add_Subn, C(3), S(3), C(4));
   bit4: fa PORT MAP (A(4), B(4) XOR Add_Subn, C(4), S(4), C(5));
   bit5: fa PORT MAP (A(5), B(5) XOR Add_Subn, C(5), S(5), C(6));
   bit6: fa PORT MAP (A(6), B(6) XOR Add_Subn, C(6), S(6), C(7));
   bit7: fa PORT MAP (A(7), B(7) XOR Add_Subn, C(7), S(7), C(8));
   -- Display the adder outputs
   LEDR <= S;
```

```
   -- instantiate regn (R, Clock, Resetn, Q);
   U_S: regn PORT MAP (S, Clock, Resetn, S_reg);

   -- check for Overflow
   Overflow <= C(8) XOR C(7);
   U_Overflow: flipflop PORT MAP (Overflow, Clock, Resetn, Overflow_reg);
   LEDG(8) <= Overflow_reg;

   -- drive the displays through 7-seg decoders
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   HEX3 <= "1111111";
   HEX2 <= "1111111";
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn    : IN STD_LOGIC;
            Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN -- asynchronous clear
            Q <= '0';
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= D;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Signed 8-bit ripple-carry adder S = A + B, with overflow detection
-- Uses lpm_add_sub. Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A, SW7-0 = B
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: LEDR7-0 shows S in binary form
--          HEX7-6 shows input A, HEX5-4 shows input B
--          HEX3-0 shows the output sum
--          LEDG8 shows overflow
ENTITY part3 IS
   PORT (   KEY                     : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            LEDR                    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDG                    : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part3;

ARCHITECTURE Structure OF part3 IS
   COMPONENT lpm_add8
   PORT (   dataa    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            datab    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            overflow : OUT STD_LOGIC ;
            result   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn    : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL A, B, S, S_reg : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C : STD_LOGIC_VECTOR(8 DOWNTO 1);     -- carries
   SIGNAL Clock, Resetn, Overflow, Overflow_reg : STD_LOGIC;
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   -- instantiate the lpm module lpm_add8 (dataa, datab, overflow, result);
   U_add8: lpm_add8 PORT MAP (A, B, Overflow, S);
   -- Display the adder outputs
   LEDR <= S;

   -- instantiate regn (R, Clock, Resetn, Q);
   U_S: regn PORT MAP (S, Clock, Resetn, S_reg);

   -- check for Overflow
   U_Overflow: flipflop PORT MAP (Overflow, Clock, Resetn, Overflow_reg);
   LEDG(8) <= Overflow_reg;
```

```
   -- drive the displays through 7-seg decoders
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   HEX3 <= "1111111";
   HEX2 <= "1111111";
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn    : IN STD_LOGIC;
            Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN -- asynchronous clear
            Q <= '0';
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= D;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;
```

```
ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --        0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Signed 8-bit ripple-carry adder/subtractor S = A +/- B, with overflow detection
-- Uses lpm_add_sub. Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A, SW7-0 = B
--          SW16 = Add_Subn
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: LEDR7-0 shows S in binary form
--          HEX7-6 shows input A, HEX5-4 shows input B
--          HEX3-0 shows the output sum
--          LEDG8 shows overflow
ENTITY part4 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(16 DOWNTO 0);
            LEDR                   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            LEDG                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part4;

ARCHITECTURE Structure OF part4 IS
   COMPONENT lpm_addsub8
   PORT (   add_sub  : IN STD_LOGIC ;
            dataa    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            datab    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            overflow : OUT STD_LOGIC ;
            result   : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn    : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL A, B, S, S_reg : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C : STD_LOGIC_VECTOR(8 DOWNTO 1);     -- carries
   SIGNAL Clock, Resetn, Add_Subn, Overflow, Overflow_reg : STD_LOGIC;
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   Add_Subn <= SW(16);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   -- instantiate the lpm module lpm_addsub8 (add_sub, dataa, datab, overflow, result)
;
   U_addsub: lpm_addsub8 PORT MAP (NOT(Add_Subn), A, B, Overflow, S);
   -- Display the adder outputs
   LEDR <= S;

   -- instantiate regn (R, Clock, Resetn, Q);
   U_S: regn PORT MAP (S, Clock, Resetn, S_reg);
```

```
   -- check for Overflow
   U_Overflow: flipflop PORT MAP (Overflow, Clock, Resetn, Overflow_reg);
   LEDG(8) <= Overflow_reg;

   -- drive the displays through 7-seg decoders
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   HEX3 <= "1111111";
   HEX2 <= "1111111";
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn    : IN STD_LOGIC;
            Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN -- asynchronous clear
            Q <= '0';
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= D;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
```

```
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Input two 4-bit numbers using the SW switches and display the numbers
-- on one digit of the two 2-digit 7-seg displays.  Multiply and display the
-- product on the two digits of the 4-digit 7-seg display
ENTITY part5 IS
   PORT (   SW                    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Structure OF part5 IS
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;

   SIGNAL A, B : STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL P : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL C_b1 : STD_LOGIC_VECTOR(3 DOWNTO 1); -- carries for row that ANDs with B1
   SIGNAL C_b2 : STD_LOGIC_VECTOR(3 DOWNTO 1); -- carries for row that ANDs with B2
   SIGNAL C_b3 : STD_LOGIC_VECTOR(3 DOWNTO 1); -- carries for row that ANDs with B3
   -- partial products from row that ANDs with B1:
   SIGNAL PP1 : STD_LOGIC_VECTOR(5 DOWNTO 2);
   -- partial products from row that ANDs with B2:
   SIGNAL PP2 : STD_LOGIC_VECTOR(6 DOWNTO 3);
BEGIN
   A <= SW(11 DOWNTO 8);
   B <= SW(3 DOWNTO 0);
   P(0) <= A(0) AND B(0);

   -- fa (a, b, ci, s, co);
   b1_a0: fa PORT MAP (A(1) AND B(0), A(0) AND B(1), '0',     P(1),   C_b1(1));
   b1_a1: fa PORT MAP (A(2) AND B(0), A(1) AND B(1), C_b1(1), PP1(2), C_b1(2));
   b1_a2: fa PORT MAP (A(3) AND B(0), A(2) AND B(1), C_b1(2), PP1(3), C_b1(3));
   b1_a3: fa PORT MAP ('0',           A(3) AND B(1), C_b1(3), PP1(4), PP1(5));

   -- fa (a, b, ci, s, co);
   b2_a0: fa PORT MAP (PP1(2), A(0) AND B(2), '0',     P(2),   C_b2(1));
   b2_a1: fa PORT MAP (PP1(3), A(1) AND B(2), C_b2(1), PP2(3), C_b2(2));
   b2_a2: fa PORT MAP (PP1(4), A(2) AND B(2), C_b2(2), PP2(4), C_b2(3));
   b2_a3: fa PORT MAP (PP1(5), A(3) AND B(2), C_b2(3), PP2(5), PP2(6));

   -- fa (a, b, ci, s, co);
   b3_a0: fa PORT MAP (PP2(3), A(0) AND B(3), '0',     P(3), C_b3(1));
   b3_a1: fa PORT MAP (PP2(4), A(1) AND B(3), C_b3(1), P(4), C_b3(2));
   b3_a2: fa PORT MAP (PP2(5), A(2) AND B(3), C_b3(2), P(5), C_b3(3));
   b3_a3: fa PORT MAP (PP2(6), A(3) AND B(3), C_b3(3), P(6), P(7));

   -- drive the display through a 7-seg decoder
   digit_7: hex7seg PORT MAP ("0000", HEX7);
   digit_6: hex7seg PORT MAP (A, HEX6);

   digit_5: hex7seg PORT MAP ("0000", HEX5);
   digit_4: hex7seg PORT MAP (B, HEX4);

   digit_3: hex7seg PORT MAP ("0000", HEX3);
   digit_2: hex7seg PORT MAP ("0000", HEX2);
   digit_1: hex7seg PORT MAP (P(7 DOWNTO 4), HEX1);
```

```
   digit_0: hex7seg PORT MAP (P(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Input two 4-bit numbers using the SW switches and display the numbers
-- on one digit of the two 2-digit 7-seg displays.  Multiply and display
-- the product on the two digits of the 4-digit 7-seg display
ENTITY part6 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part6;

ARCHITECTURE Structure OF part6 IS
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn    : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT fa
      PORT (   a, b, ci : IN  STD_LOGIC;
               s, co    : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;

   SIGNAL Clock, Resetn : STD_LOGIC;
   SIGNAL A, B : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL P, P_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL C_b1 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B1
   SIGNAL C_b2 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B2
   SIGNAL C_b3 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B3
   SIGNAL C_b4 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B4
   SIGNAL C_b5 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B5
   SIGNAL C_b6 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B6
   SIGNAL C_b7 : STD_LOGIC_VECTOR(7 DOWNTO 1); -- carries for row that ANDs with B7
   -- partial products from row that ANDs with B1:
   SIGNAL PP1 : STD_LOGIC_VECTOR(9 DOWNTO 2);
   -- partial products from row that ANDs with B2:
   SIGNAL PP2 : STD_LOGIC_VECTOR(10 DOWNTO 3);
   -- partial products from row that ANDs with B3:
   SIGNAL PP3 : STD_LOGIC_VECTOR(11 DOWNTO 4);
   -- partial products from row that ANDs with B4:
   SIGNAL PP4 : STD_LOGIC_VECTOR(12 DOWNTO 5);
   -- partial products from row that ANDs with B5:
   SIGNAL PP5 : STD_LOGIC_VECTOR(13 DOWNTO 6);
   -- partial products from row that ANDs with B6:
   SIGNAL PP6 : STD_LOGIC_VECTOR(14 DOWNTO 7);
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   P(0) <= A(0) AND B(0);

   --  fa (a, b, ci, s, co);
   b1_a0: fa PORT MAP (A(1) AND B(0), A(0) AND B(1), '0',     P(1),   C_b1(1));
   b1_a1: fa PORT MAP (A(2) AND B(0), A(1) AND B(1), C_b1(1), PP1(2), C_b1(2));
```

```
    b1_a2: fa PORT MAP (A(3) AND B(0), A(2) AND B(1), C_b1(2), PP1(3), C_b1(3));
    b1_a3: fa PORT MAP (A(4) AND B(0), A(3) AND B(1), C_b1(3), PP1(4), C_b1(4));
    b1_a4: fa PORT MAP (A(5) AND B(0), A(4) AND B(1), C_b1(4), PP1(5), C_b1(5));
    b1_a5: fa PORT MAP (A(6) AND B(0), A(5) AND B(1), C_b1(5), PP1(6), C_b1(6));
    b1_a6: fa PORT MAP (A(7) AND B(0), A(6) AND B(1), C_b1(6), PP1(7), C_b1(7));
    b1_a7: fa PORT MAP ('0',        A(7) AND B(1), C_b1(7), PP1(8), PP1(9));

    --  fa (a, b, ci, s, co);
    b2_a0: fa PORT MAP (PP1(2), A(0) AND B(2), '0',     P(2),   C_b2(1));
    b2_a1: fa PORT MAP (PP1(3), A(1) AND B(2), C_b2(1), PP2(3), C_b2(2));
    b2_a2: fa PORT MAP (PP1(4), A(2) AND B(2), C_b2(2), PP2(4), C_b2(3));
    b2_a3: fa PORT MAP (PP1(5), A(3) AND B(2), C_b2(3), PP2(5), C_b2(4));
    b2_a4: fa PORT MAP (PP1(6), A(4) AND B(2), C_b2(4), PP2(6), C_b2(5));
    b2_a5: fa PORT MAP (PP1(7), A(5) AND B(2), C_b2(5), PP2(7), C_b2(6));
    b2_a6: fa PORT MAP (PP1(8), A(6) AND B(2), C_b2(6), PP2(8), C_b2(7));
    b2_a7: fa PORT MAP (PP1(9), A(7) AND B(2), C_b2(7), PP2(9), PP2(10));

    --  fa (a, b, ci, s, co);
    b3_a0: fa PORT MAP (PP2(3),  A(0) AND B(3), '0',     P(3),    C_b3(1));
    b3_a1: fa PORT MAP (PP2(4),  A(1) AND B(3), C_b3(1), PP3(4),  C_b3(2));
    b3_a2: fa PORT MAP (PP2(5),  A(2) AND B(3), C_b3(2), PP3(5),  C_b3(3));
    b3_a3: fa PORT MAP (PP2(6),  A(3) AND B(3), C_b3(3), PP3(6),  C_b3(4));
    b3_a4: fa PORT MAP (PP2(7),  A(4) AND B(3), C_b3(4), PP3(7),  C_b3(5));
    b3_a5: fa PORT MAP (PP2(8),  A(5) AND B(3), C_b3(5), PP3(8),  C_b3(6));
    b3_a6: fa PORT MAP (PP2(9),  A(6) AND B(3), C_b3(6), PP3(9),  C_b3(7));
    b3_a7: fa PORT MAP (PP2(10), A(7) AND B(3), C_b3(7), PP3(10), PP3(11));

    --  fa (a, b, ci, s, co);
    b4_a0: fa PORT MAP (PP3(4),  A(0) AND B(4), '0',     P(4),    C_b4(1));
    b4_a1: fa PORT MAP (PP3(5),  A(1) AND B(4), C_b4(1), PP4(5),  C_b4(2));
    b4_a2: fa PORT MAP (PP3(6),  A(2) AND B(4), C_b4(2), PP4(6),  C_b4(3));
    b4_a3: fa PORT MAP (PP3(7),  A(3) AND B(4), C_b4(3), PP4(7),  C_b4(4));
    b4_a4: fa PORT MAP (PP3(8),  A(4) AND B(4), C_b4(4), PP4(8),  C_b4(5));
    b4_a5: fa PORT MAP (PP3(9),  A(5) AND B(4), C_b4(5), PP4(9),  C_b4(6));
    b4_a6: fa PORT MAP (PP3(10), A(6) AND B(4), C_b4(6), PP4(10), C_b4(7));
    b4_a7: fa PORT MAP (PP3(11), A(7) AND B(4), C_b4(7), PP4(11), PP4(12));

    --  fa (a, b, ci, s, co);
    b5_a0: fa PORT MAP (PP4(5),  A(0) AND B(5), '0',     P(5),    C_b5(1));
    b5_a1: fa PORT MAP (PP4(6),  A(1) AND B(5), C_b5(1), PP5(6),  C_b5(2));
    b5_a2: fa PORT MAP (PP4(7),  A(2) AND B(5), C_b5(2), PP5(7),  C_b5(3));
    b5_a3: fa PORT MAP (PP4(8),  A(3) AND B(5), C_b5(3), PP5(8),  C_b5(4));
    b5_a4: fa PORT MAP (PP4(9),  A(4) AND B(5), C_b5(4), PP5(9),  C_b5(5));
    b5_a5: fa PORT MAP (PP4(10), A(5) AND B(5), C_b5(5), PP5(10), C_b5(6));
    b5_a6: fa PORT MAP (PP4(11), A(6) AND B(5), C_b5(6), PP5(11), C_b5(7));
    b5_a7: fa PORT MAP (PP4(12), A(7) AND B(5), C_b5(7), PP5(12), PP5(13));

    --  fa (a, b, ci, s, co);
    b6_a0: fa PORT MAP (PP5(6),  A(0) AND B(6), '0',     P(6),    C_b6(1));
    b6_a1: fa PORT MAP (PP5(7),  A(1) AND B(6), C_b6(1), PP6(7),  C_b6(2));
    b6_a2: fa PORT MAP (PP5(8),  A(2) AND B(6), C_b6(2), PP6(8),  C_b6(3));
    b6_a3: fa PORT MAP (PP5(9),  A(3) AND B(6), C_b6(3), PP6(9),  C_b6(4));
    b6_a4: fa PORT MAP (PP5(10), A(4) AND B(6), C_b6(4), PP6(10), C_b6(5));
    b6_a5: fa PORT MAP (PP5(11), A(5) AND B(6), C_b6(5), PP6(11), C_b6(6));
    b6_a6: fa PORT MAP (PP5(12), A(6) AND B(6), C_b6(6), PP6(12), C_b6(7));
    b6_a7: fa PORT MAP (PP5(13), A(7) AND B(6), C_b6(7), PP6(13), PP6(14));

    --  fa (a, b, ci, s, co);
    b7_a0: fa PORT MAP (PP6(7),  A(0) AND B(7), '0',     P(7),  C_b7(1));
    b7_a1: fa PORT MAP (PP6(8),  A(1) AND B(7), C_b7(1), P(8),  C_b7(2));
    b7_a2: fa PORT MAP (PP6(9),  A(2) AND B(7), C_b7(2), P(9),  C_b7(3));
    b7_a3: fa PORT MAP (PP6(10), A(3) AND B(7), C_b7(3), P(10), C_b7(4));
    b7_a4: fa PORT MAP (PP6(11), A(4) AND B(7), C_b7(4), P(11), C_b7(5));
    b7_a5: fa PORT MAP (PP6(12), A(5) AND B(7), C_b7(5), P(12), C_b7(6));
```

```
   b7_a6: fa PORT MAP (PP6(13), A(6) AND B(7), C_b7(6), P(13), C_b7(7));
   b7_a7: fa PORT MAP (PP6(14), A(7) AND B(7), C_b7(7), P(14), P(15));

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_P: regn GENERIC MAP (N => 16) PORT MAP (P, Clock, Resetn, P_reg);

   -- drive the display through a 7-seg decoder
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   digit_3: hex7seg PORT MAP (P_reg(15 DOWNTO 12), HEX3);
   digit_2: hex7seg PORT MAP (P_reg(11 DOWNTO 8), HEX2);
   digit_1: hex7seg PORT MAP (P_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (P_reg(3 DOWNTO 0), HEX0);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn    : IN STD_LOGIC;
            Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fa IS
   PORT (   a, b, ci : IN  STD_LOGIC;
            s, co    : OUT STD_LOGIC);
END fa;

ARCHITECTURE Structure OF fa IS
   SIGNAL a_xor_b : STD_LOGIC;
BEGIN
   a_xor_b <= a XOR b;
   s <= a_xor_b XOR ci;
   co <= (NOT(a_xor_b) AND b) OR (a_xor_b AND ci);
END Structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;
```

```
ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --        0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Input two 8-bit numbers using the SW switches and display the numbers
-- on the 2-digit 7-seg displays.  Multiply and display the product on the
-- 4-digit 7-seg display
ENTITY part7 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part7;

ARCHITECTURE Behavior OF part7 IS
   COMPONENT regn
      GENERIC ( N : integer:= 8);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn     : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   COMPONENT lpm_mult16
      PORT (   dataa    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               datab    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               result      : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn : STD_LOGIC;
   SIGNAL A, B : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL P, P_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_A: regn PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, A);
   U_B: regn PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, B);

   -- lpm_mult16 (dataa, datab, result)
   u1: lpm_mult16 PORT MAP (A, B, P);

   -- instantiate module regn (R, Clock, Resetn, Q);
   U_P: regn GENERIC MAP (N => 16) PORT MAP (P, Clock, Resetn, P_reg);

   -- drive the display through a 7-seg decoder
   digit_7: hex7seg PORT MAP (A(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B(3 DOWNTO 0), HEX4);

   digit_3: hex7seg PORT MAP (P_reg(15 DOWNTO 12), HEX3);
   digit_2: hex7seg PORT MAP (P_reg(11 DOWNTO 8), HEX2);
   digit_1: hex7seg PORT MAP (P_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (P_reg(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC ( N : integer:= 8);
```

```
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn    : IN STD_LOGIC;
            Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF (Clock'EVENT AND Clock = '1') THEN
         Q <= R;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- unsigned 8-bit multiplier-adder S = (A x B) + (C x D)
-- Uses lpm_mult and lpm_add_sub. Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A or C, SW7-0 = B or D
--          SW16 = 0 loads A and B, SW16 = 1 loads C and D
--          SW17 = write enable. Setting this to 1 allows registers A, B or
--          C, D to be written
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: HEX7-6 shows input A when SW16 = 0, shows input C when SW16 = 1
--          HEX5-4 shows input B when SW16 = 0, shows input D when SW16 = 1
--          HEX3-0 shows the output sum
ENTITY part8 IS
   PORT (   KEY                     : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                      : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDG                    : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4  : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part8;

ARCHITECTURE Behavior OF part8 IS
   COMPONENT regne
      GENERIC ( N : integer:= 8);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   COMPONENT lpm_mult16
      PORT (   dataa    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               datab    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               result     : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT lpm_add16
      PORT (   dataa    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
               datab    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
               cout     : OUT STD_LOGIC ;
               result   : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn, sel_AB_CD, WE : STD_LOGIC;
                              -- sel_AB_CD selects regs, WE is write enable
   SIGNAL Cout : STD_LOGIC;   -- carry out for final sum
   SIGNAL A, B, C, D : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL P_AxB, P_CxD, S, S_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL A_C, B_D : STD_LOGIC_VECTOR(7 DOWNTO 0);
   -- uncomment line below for the pipelined version
   -- SIGNAL P_AxB_reg, P_CxD_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   sel_AB_CD <= SW(16);
   WE <= SW(17);

   -- instantiate module regne (R, Clock, Resetn, Q);
   U_A: regne PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, NOT(sel_AB_CD) AND WE, A);
   U_B: regne PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, NOT(sel_AB_CD) AND WE, B);
   U_C: regne PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, sel_AB_CD AND WE, C);
   U_D: regne PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, sel_AB_CD AND WE, D);
```

```
   -- instantiate module lpm_mult16 (dataa, datab, result);
   U_AxB: lpm_mult16 PORT MAP (A, B, P_AxB);
   U_CxD: lpm_mult16 PORT MAP (C, D, P_CxD);
   -- uncomment the two lines below to create the pipeline registers
   -- U_P_AxB: regne GENERIC MAP (N => 16) PORT MAP (P_AxB, Clock, Resetn, '1', P_AxB_
reg);
   -- U_P_CxD: regne GENERIC MAP (N => 16) PORT MAP (P_CxD, Clock, Resetn, '1', P_CxD_
reg);

   -- instantiate module lpm_add16 (dataa, datab, cout, result);
   U_Sum: lpm_add16 PORT MAP (P_AxB, P_CxD, Cout, S);
   -- uncomment the line below, comment out the line above for the pipelined version
   -- U_Sum: lpm_add16 PORT MAP (P_AxB_reg, P_CxD_reg, Cout, S);
   U_S: regne GENERIC MAP (N => 16) PORT MAP (S, Clock, Resetn, '1', S_reg);
   LEDG(8) <= Cout;

   A_C <= A WHEN (sel_AB_CD = '0') ELSE C;
   B_D <= B WHEN (sel_AB_CD = '0') ELSE D;

   -- drive the display through a 7-seg decoder
   digit_7: hex7seg PORT MAP (A_C(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A_C(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B_D(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B_D(3 DOWNTO 0), HEX4);

   digit_3: hex7seg PORT MAP (S_reg(15 DOWNTO 12), HEX3);
   digit_2: hex7seg PORT MAP (S_reg(11 DOWNTO 8), HEX2);
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF  (Clock'EVENT AND Clock = '1') THEN
         IF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
```

```
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --        0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- unsigned 8-bit multiplier-adder S = (A x B) + (C x D)
-- Uses lpm_multadd. Registers are included for all inputs and outputs.
-- inputs:  SW15-8 = A or C, SW7-0 = B or D
--          SW16 = 0 loads A and B, SW16 = 1 loads C and D
--          SW17 = write enable. Setting this to 1 allows registers A, B or
--          C, D to be written
--          KEY0 = active-low asynchronous reset
--          KEY1 = manual clock
-- outputs: HEX7-6 shows input A when SW16 = 0, shows input C when SW16 = 1
--          HEX5-4 shows input B when SW16 = 0, shows input D when SW16 = 1
--          HEX3-0 shows the output sum
ENTITY part9 IS
   PORT (   KEY                    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            SW                     : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
            LEDG                   : OUT STD_LOGIC_VECTOR(8 DOWNTO 8);
            HEX7, HEX6, HEX5, HEX4 : OUT STD_LOGIC_VECTOR(0 TO 6);
            HEX3, HEX2, HEX1, HEX0 : OUT STD_LOGIC_VECTOR(0 TO 6));
END part9;

ARCHITECTURE Behavior OF part9 IS
   COMPONENT regne
      GENERIC ( N : integer:= 8);
      PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E : STD_LOGIC;
               Q                : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   COMPONENT alt_multadd16
      PORT (   dataa_0  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               dataa_1  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               datab_0  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               datab_1  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               result   : OUT STD_LOGIC_VECTOR (16 DOWNTO 0));
   END COMPONENT;
   COMPONENT alt_multadd16_pipe
      PORT (   aclr0    : IN STD_LOGIC  := '0';
               clock0   : IN STD_LOGIC  := '1';
               dataa_0  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               dataa_1  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               datab_0  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               datab_1  : IN STD_LOGIC_VECTOR (7 DOWNTO 0) :=  (OTHERS => '0');
               result   : OUT STD_LOGIC_VECTOR (16 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn, sel_AB_CD, WE : STD_LOGIC;
                              -- sel_AB_CD selects regs, WE is write enable
   SIGNAL Cout : STD_LOGIC;   -- carry out for final sum
   SIGNAL A, B, C, D, A_C, B_D : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL S : STD_LOGIC_VECTOR(16 DOWNTO 0);
   SIGNAL S_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);

   sel_AB_CD <= SW(16);
   WE <= SW(17);

   -- instantiate module regne (R, Clock, Resetn, Q);
   U_A: regne PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, NOT(sel_AB_CD) AND WE, A);
```

```
   U_B: regne PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, NOT(sel_AB_CD) AND WE, B);
   U_C: regne PORT MAP (SW(15 DOWNTO 8), Clock, Resetn, sel_AB_CD AND WE, C);
   U_D: regne PORT MAP (SW(7 DOWNTO 0), Clock, Resetn, sel_AB_CD AND WE, D);

   -- instantiate module alt_multadd16 (dataa_0, dataa_1, datab_0, datab_1, result);
   U_Sum: alt_multadd16 PORT MAP (A, C, B, D, S);

   -- Use the configuration of alt_multadd below for the pipelined version
   -- instantiate module alt_multadd16_pipe (aclr0, clock0, dataa_0, dataa_1,
   --    datab_0, datab_1, result);
   -- U_Sum: alt_multadd16_pipe PORT MAP (NOT(Resetn), Clock, A, C, B, D, S);

   U_S: regne GENERIC MAP (N => 16) PORT MAP (S(15 DOWNTO 0), Clock, Resetn, '1', S_re
g);
   LEDG(8) <= S(16);

   A_C <= A WHEN (sel_AB_CD = '0') ELSE C;
   B_D <= B WHEN (sel_AB_CD = '0') ELSE D;

   -- drive the display through a 7-seg decoder
   digit_7: hex7seg PORT MAP (A_C(7 DOWNTO 4), HEX7);
   digit_6: hex7seg PORT MAP (A_C(3 DOWNTO 0), HEX6);

   digit_5: hex7seg PORT MAP (B_D(7 DOWNTO 4), HEX5);
   digit_4: hex7seg PORT MAP (B_D(3 DOWNTO 0), HEX4);

   digit_3: hex7seg PORT MAP (S_reg(15 DOWNTO 12), HEX3);
   digit_2: hex7seg PORT MAP (S_reg(11 DOWNTO 8), HEX2);
   digit_1: hex7seg PORT MAP (S_reg(7 DOWNTO 4), HEX1);
   digit_0: hex7seg PORT MAP (S_reg(3 DOWNTO 0), HEX0);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 8);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock, Resetn)
   BEGIN
      IF (Resetn = '0') THEN
         Q <= (OTHERS => '0');
      ELSIF  (Clock'EVENT AND Clock = '1') THEN
         IF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;
```

```
ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --        0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
   PROCESS (hex)
   BEGIN
      CASE hex IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

Laboratory Exercise 7

Finite State Machines

This is an exercise in using finite state machines.

Part I

We wish to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or four consecutive 0s. There is an input *w* and an output *z*. Whenever $w = 1$ or $w = 0$ for four consecutive clock pulses the value of *z* has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output *z* will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between *w* and *z*.

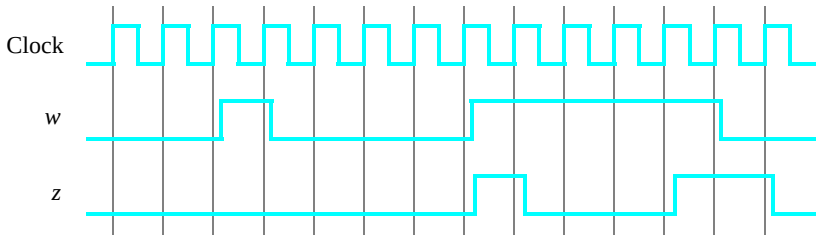


Figure 1. Required timing for the output *z*.

A state diagram for this FSM is shown in Figure 2. For this part you are to manually derive an FSM circuit that implements this state diagram, including the logic expressions that feed each of the state flip-flops. To implement the FSM use nine state flip-flops called $y_8, \ldots, y_0$ and the one-hot state assignment given in Table 1.

Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000001
B	000000010
C	000000100
D	000001000
E	000010000
F	000100000
G	001000000
H	010000000
I	100000000

Table 1. One-hot codes for the FSM.

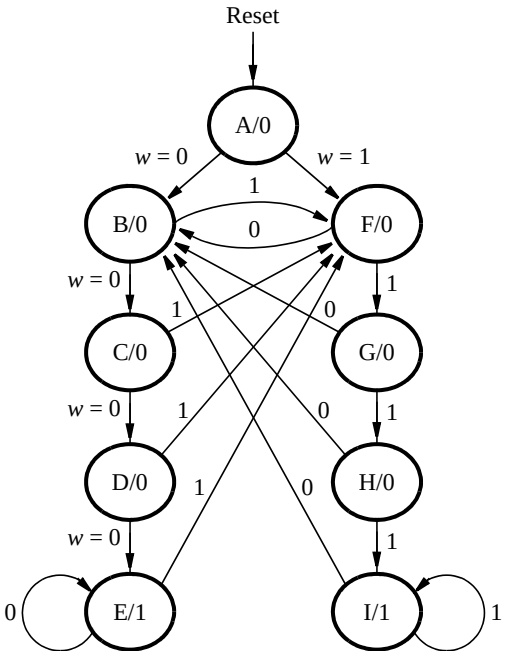


Figure 2. A state diagram for the FSM.

Design and implement your circuit on the DE2 board as follows.

1. Create a new Quartus II project for the FSM circuit. Select as the target chip the Cyclone II EP2C35F672C6, which is the FPGA chip on the Altera DE2 board.

2. Write a VHDL file that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple assignment statements in your VHDL code to specify the logic feeding the flip-flops. Note that the one-hot code enables you to derive these expressions by inspection.

   Use the toggle switch SW_0 on the Altera DE2 board as an active-low synchronous reset input for the FSM, use SW_1 as the *w* input, and the pushbutton KEY_0 as the clock input which is applied manually. Use the green LED $LEDG_0$ as the output *z*, and assign the state flip-flop outputs to the red LEDs $LEDR_8$ to $LEDR_0$.

3. Include the VHDL file in your project, and assign the pins on the FPGA to connect to the switches and the LEDs, as indicated in the User Manual for the DE2 board. Compile the circuit.

4. Simulate the behavior of your circuit.

5. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$.

6. Finally, consider a modification of the one-hot code given in Table 1. When an FSM is going to be implemented in an FPGA, the circuit can often be simplified if all flip-flop outputs are 0 when the FSM is in the reset state. This approach is preferable because the FPGA's flip-flops usually include a *clear* input port, which can be conveniently used to realize the reset state, but the flip-flops often do not include a *set* input port.

Table 2 shows a modified one-hot state assignment in which the reset state, *A*, uses all 0s. This is accomplished by inverting the state variable y_0. Create a modified version of your VHDL code that implements this state assignment. *Hint*: you should need to make very few changes to the logic expressions in your circuit to implement the modified codes. Compile your new circuit and test it both through simulation and by downloading it onto the DE2 board.

Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000000
B	000000011
C	000000101
D	000001001
E	000010001
F	000100001
G	001000001
H	010000001
I	100000001

Table 2. Modified one-hot codes for the FSM.

Part II

For this part you are to write another style of VHDL code for the FSM in Figure 2. In this version of the code you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a VHDL CASE statement in a PROCESS block, and use another PROCESS block to instantiate the state flip-flops. You can use a third PROCESS block or simple assignment statements to specify the output *z*.

A suggested skeleton of the VHDL code is given in Figure 3. Observe that the present and next state vectors for the FSM are defined as an enumerated type with possible values given by the symbols A to I. The VHDL compiler determines how many state flip-flops to use for the circuit, and it automatically chooses the state assignment.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part2 IS
    PORT (  . . . define input and output ports
            . . .);
END part2;

ARCHITECTURE Behavior OF part2 IS
    . . . declare signals
    TYPE State_type IS (A, B, C, D, E, F, G, H, I);
    SIGNAL y_Q, Y_D : State_type; - - y_Q is present state, y_D is next state
BEGIN
    . . .
    PROCESS (w, y_Q) - - state table
    BEGIN
        case y_Q IS
            WHEN A  IF (w = '0') THEN Y_D <= B;
                    ELSE Y_D <= F;
                    END IF;
            . . . other states
        END CASE;
    END PROCESS; - - state table

    PROCESS (Clock) - - state flip-flops
    BEGIN
        . . .
    END PROCESS;

    . . . assignments for output z and the LEDs
END Behavior;
```

Figure 3. Skeleton VHDL code for the FSM.

Implement your circuit as follows.

1. Create a new project for the FSM. Select as the target chip the Cyclone II EP2C35F672C6.
2. Include in the project your VHDL file that uses the style of code in Figure 3. Use the toggle switch SW_0 on the Altera DE2 board as an active-low synchronous reset input for the FSM, use SW_1 as the *w* input, and the pushbutton KEY_0 as the clock input which is applied manually. Use the green LED $LEDG_0$ as the output *z*, and use nine red LEDs, $LEDR_8$ to $LEDR_0$, to indicate the present state of the FSM. Assign the pins on the FPGA to connect to the switches and the LEDs, as indicated in the User Manual for the DE2 board.
3. Before compiling your code it is possible to tell the Synthesis tool in Quartus II what style of state assignment it should use. Choose Assignments > Settings in Quartus II, and then click on the Analysis and Synthesis item on the left side of the window. As indicated in Figure 4, change the parameter State Machine Processing to the setting Minimal Bits.
4. To examine the circuit produced by Quartus II open the RTL Viewer tool. Double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one in Figure 2. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines.
5. Simulate the behavior of your circuit.

6. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$.

7. In step 3 you instructed the Quartus II Synthesis tool to use the state assignment given in your VHDL code. To see the result of changing this setting, open again the Quartus II settings window by choosing Assignments > Settings, and click on the Analysis and Synthesis item. Change the setting for State Machine Processing from Minimal Bits to One-Hot. Recompile the circuit and then open the report file, select the Analysis and Synthesis section of the report, and click on State Machines. Compare the state codes shown to those given in Table 2, and discuss any differences that you observe.

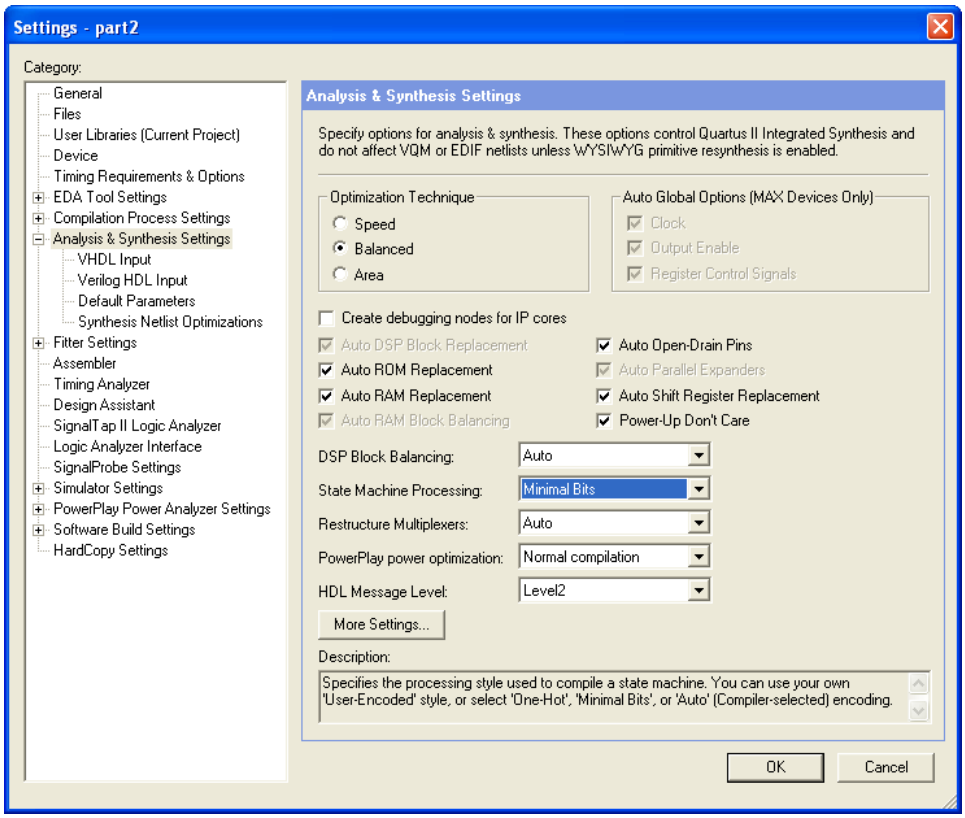


Figure 4. Specifying the state assignment method in Quartus II.

Part III

For this part you are to implement the sequence-detector FSM by using shift registers, instead of using the more formal approach described above. Create VHDL code that instantiates two 4-bit shift registers; one is for recognizing a sequence of four 0s, and the other for four 1s. Include the appropriate logic expressions in your design to produce the output z. Make a Quartus II project for your design and implement the circuit on the DE2 board. Use the switches and LEDs on the board in a similar way as you did for Parts I and II and observe the behavior of your shift registers and the output z. Answer the following question: could you use just one 4-bit shift register, rather than two? Explain your answer.

Part IV

We want to design a modulo-10 counter-like circuit that behaves as follows. It is reset to 0 by the *Reset* input. It has two inputs, w_1 and w_0, which control its counting operation. If $w_1w_0 = 00$, the count remains the same. If $w_1w_0 = 01$, the count is incremented by 1. If $w_1w_0 = 10$, the count is incremented by 2. If $w_1w_0 = 11$, the count is decremented by 1. All changes take place on the active edge of a *Clock* input. Use toggle switches SW_2 and SW_1 for inputs w_1 and w_0. Use toggle switch SW_0 as an active-low synchronous reset, and use the pushbutton KEY_0 as a manual clock. Display the decimal contents of the counter on the 7-segment display *HEX0*.

1. Create a new project which will be used to implement the circuit on the DE2 board.
2. Write a VHDL file that defines the circuit. Use the style of code indicated in Figure 3 for your FSM.
3. Include the VHDL file in your project and compile the circuit.
4. Simulate the behavior of your circuit.
5. Assign the pins on the FPGA to connect to the switches and the 7-segment display.
6. Recompile the circuit and download it into the FPGA chip.
7. Test the functionality of your design by applying some inputs and observing the output display.

Part V

For this part you are to design a circuit for the DE2 board that scrolls the word "HELLO" in ticker-tape fashion on the eight 7-segment displays $HEX7 - 0$. The letters should move from right to left each time you apply a manual clock pulse to the circuit. After the word "HELLO" scrolls off the left side of the displays it then starts again on the right side.

Design your circuit by using eight 7-bit registers connected in a queue-like fashion, such that the outputs of the first register feed the inputs of the second, the second feeds the third, and so on. This type of connection between registers is often called a *pipeline*. Each register's outputs should directly drive the seven segments of one display. You are to design a finite state machine that controls the pipeline in two ways:

1. For the first eight clock pulses after the system is reset, the FSM inserts the correct characters (H,E,L,L,0, , ,) into the first of the 7-bit registers in the pipeline.
2. After step 1 is complete, the FSM configures the pipeline into a loop that connects the last register back to the first one, so that the letters continue to scroll indefinitely.

Write VHDL code for the ticker-tape circuit and create a Quartus II project for your design. Use KEY_0 on the DE2 board to clock the FSM and pipeline registers and use SW_0 as a synchronous active-low reset input. Write VHDL code in the style shown in Figure 3 for your finite state machine.

Compile your VHDL code, download it onto the DE2 board and test the circuit.

Part VI

For this part you are to modify your circuit from Part V so that it no longer requires manually-applied clock pulses. Your circuit should scroll the word "HELLO" such that the letters move from right to left in intervals of about one second. Scrolling should continue indefinitely; after the word "HELLO" scrolls off the left side of the displays it should start again on the right side.

Write VHDL code for the ticker-tape circuit and create a Quartus II project for your design. Use the 50-MHz clock signal, *CLOCK_50*, on the DE2 board to clock the FSM and pipeline registers and use KEY_0 as a synchronous active-low reset input. Write VHDL code in the style shown in Figure 3 for your finite state machine, and ensure that all flip-flops in your circuit are clocked directly by the *CLOCK_50* input. Do not derive or use any other clock signals in your circuit.

Compile your VHDL code, download it onto the DE2 board and test the circuit.

Part VII

Augment your design from Part VI so that under the control of pushbuttons KEY_2 and KEY_1 the rate at which the letters move from right to left can be changed. If KEY_1 is pressed, the letters should move twice as fast. If KEY_2 is pressed, the rate has to be reduced by a factor of 2.

Note that the KEY_2 and KEY_1 switches are debounced and will produce exactly one low pulse when pressed. However, there is no way of knowing how long a switch may remain depressed, which means that the pulse duration can be arbitrarily long. A good approach for designing this circuit is to include a second FSM in your VHDL code that properly responds to the pressed keys. The outputs of this FSM can change appropriately when a key is pressed, and the FSM can wait for each key press to end before continuing. The outputs produced by this second FSM can be used as part of the scheme for creating a variable time interval in your circuit. Note that KEY_2 and KEY_1 are asynchronous inputs to your circuit, so be sure to synchronize them to the clock signal before using these signals as inputs to your finite state machine.

The ticker tape should operate as follows. When the circuit is reset, scrolling occurs at about one second intervals. Pressing KEY_1 repeatedly causes the scrolling speed to double to a maximum of four letters per second. Pressing KEY_2 repeatedly causes the scrolling speed to slow down to a minimum of one letter every four seconds.

Implement your circuit on the DE2 board and demonstrate that it works properly.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- A sequence detector FSM using one-hot encoding.
-- SW0 is the active low synchronous reset, SW1 is the w input, and KEY0 is the clock.
-- The z output appears on LEDG0, and the state FFs appear on LEDR8..0
ENTITY part1 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDG                        : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDR                        : OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT flipflop
      PORT (   D, Clock, Resetn, Setn  : IN  STD_LOGIC;
               Q                       : OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL Clock, Resetn, w, z : STD_LOGIC;
   SIGNAL y_Q, Y_D : STD_LOGIC_VECTOR(8 DOWNTO 0);
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   w <= SW(1);

   Y_D(0) <= '0';
   U0: flipflop PORT MAP (Y_D(0), Clock, '1', Resetn, y_Q(0));
   Y_D(1) <= (y_Q(0) OR y_Q(5) OR y_Q(6) OR y_Q(7) OR y_Q(8)) AND NOT (w);
   U1: flipflop PORT MAP (Y_D(1), Clock, Resetn, '1', y_Q(1));
   Y_D(2) <= y_Q(1) AND NOT (w);
   U2: flipflop PORT MAP (Y_D(2), Clock, Resetn, '1', y_Q(2));
   Y_D(3) <= y_Q(2) AND NOT (w);
   U3: flipflop PORT MAP (Y_D(3), Clock, Resetn, '1', y_Q(3));
   Y_D(4) <= (y_Q(3) OR y_Q(4)) AND NOT (w);
   U4: flipflop PORT MAP (Y_D(4), Clock, Resetn, '1', y_Q(4));

   Y_D(5) <= (y_Q(0) OR y_Q(1) OR y_Q(2) OR y_Q(3) OR y_Q(4)) AND (w);
   U5: flipflop PORT MAP (Y_D(5), Clock, Resetn, '1', y_Q(5));
   Y_D(6) <= y_Q(5) AND (w);
   U6: flipflop PORT MAP (Y_D(6), Clock, Resetn, '1', y_Q(6));
   Y_D(7) <= y_Q(6) AND (w);
   U7: flipflop PORT MAP (Y_D(7), Clock, Resetn, '1', y_Q(7));
   Y_D(8) <= (y_Q(7) OR y_Q(8)) AND (w);
   U8: flipflop PORT MAP (Y_D(8), Clock, Resetn, '1', y_Q(8));

   z <= y_Q(4) OR y_Q(8);
   LEDR(8 DOWNTO 0) <= y_Q(8 DOWNTO 0);
   LEDG(0) <= z;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn, Setn  : IN  STD_LOGIC;
            Q                       : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN -- synchronous clear
            Q <= '0';
```

```
         ELSIF (Setn  = '0') THEN -- synchronous set
            Q <= '1';
         ELSE
            Q <= D;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- A sequence detector FSM using one-hot encoding.
-- SW0 is the active low synchronous reset, SW1 is the w input, and KEY0 is the clock.
-- The z output appears on LEDG0, and the state FFs appear on LEDR8..0
ENTITY part1 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDG                        : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDR                        : OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT flipflop
      PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL Clock, Resetn, w, z : STD_LOGIC;
   SIGNAL y_Q, Y_D : STD_LOGIC_VECTOR(8 DOWNTO 0);
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   w <= SW(1);

   -- Three changes are needed from the version that uses the traditional one-hot code
:
   -- 1. Change equation for Y0 to Y0 = 1.
   -- 2. Change logic equations for Y1 and Y5 to use NOT Y0
   -- 3. Change flipflops to not have reset input
   Y_D(0) <= '1';
   U0: flipflop PORT MAP (Y_D(0), Clock, Resetn, y_Q(0));
   Y_D(1) <= (NOT (y_Q(0)) OR y_Q(5) OR y_Q(6) OR y_Q(7) OR y_Q(8)) AND NOT (w);
   U1: flipflop PORT MAP (Y_D(1), Clock, Resetn, y_Q(1));
   Y_D(2) <= y_Q(1) AND NOT (w);
   U2: flipflop PORT MAP (Y_D(2), Clock, Resetn, y_Q(2));
   Y_D(3) <= y_Q(2) AND NOT (w);
   U3: flipflop PORT MAP (Y_D(3), Clock, Resetn, y_Q(3));
   Y_D(4) <= (y_Q(3) OR y_Q(4)) AND NOT (w);
   U4: flipflop PORT MAP (Y_D(4), Clock, Resetn, y_Q(4));

   Y_D(5) <= (NOT (y_Q(0)) OR y_Q(1) OR y_Q(2) OR y_Q(3) OR y_Q(4)) AND (w);
   U5: flipflop PORT MAP (Y_D(5), Clock, Resetn, y_Q(5));
   Y_D(6) <= y_Q(5) AND (w);
   U6: flipflop PORT MAP (Y_D(6), Clock, Resetn, y_Q(6));
   Y_D(7) <= y_Q(6) AND (w);
   U7: flipflop PORT MAP (Y_D(7), Clock, Resetn, y_Q(7));
   Y_D(8) <= (y_Q(7) OR y_Q(8)) AND (w);
   U8: flipflop PORT MAP (Y_D(8), Clock, Resetn, y_Q(8));

   z <= y_Q(4) OR y_Q(8);
   LEDR(8 DOWNTO 0) <= y_Q(8 DOWNTO 0);
   LEDG(0) <= z;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
```

```
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN -- synchronous clear
            Q <= '0';
         ELSE
            Q <= D;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- A sequence detector FSM
-- SW0 is the active low synchronous reset, SW1 is the w input, and KEY0 is the clock.
-- The z output appears on LEDG0, and the state is indicated on LEDR8..0
ENTITY part2 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDG                        : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDR                        : OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END part2;

ARCHITECTURE Behavior OF part2 IS
   SIGNAL Clock, Resetn, w, z : STD_LOGIC;
   TYPE State_type IS (A, B, C, D, E, F, G, H, I);
   SIGNAL y_Q, Y_D : State_type;
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   w <= SW(1);

   PROCESS (w, y_Q) -- state table
   BEGIN
      CASE y_Q IS
         WHEN A =>   IF (w = '0') THEN Y_D <= B;
                     ELSE Y_D <= F;
                     END IF;
         WHEN B =>   IF (w = '0') THEN Y_D <= C;
                     ELSE Y_D <= F;
                     END IF;
         WHEN C =>   IF (w = '0') THEN Y_D <= D;
                     ELSE Y_D <= F;
                     END IF;
         WHEN D =>   IF (w = '0') THEN Y_D <= E;
                     ELSE Y_D <= F;
                     END IF;
         WHEN E =>   IF (w = '0') THEN Y_D <= E;
                     ELSE Y_D <= F;
                     END IF;
         WHEN F =>   IF (w = '0') THEN Y_D <= B;
                     ELSE Y_D <= G;
                     END IF;
         WHEN G =>   IF (w = '0') THEN Y_D <= B;
                     ELSE Y_D <= H;
                     END IF;
         WHEN H =>   IF (w = '0') THEN Y_D <= B;
                     ELSE Y_D <= I;
                     END IF;
         WHEN I =>   IF (w = '0') THEN Y_D <= B;
                     ELSE Y_D <= I;
                     END IF;
         WHEN OTHERS => Y_D <= A;
      END CASE;
   END PROCESS; -- state_table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            y_Q <= A;
         ELSE
            y_Q <= Y_D;
         END IF;
      END IF;
```

```
   END PROCESS;

   z <= '1' WHEN ((y_Q = E) OR (y_Q = I)) ELSE '0';
   LEDG(0) <= z;

   PROCESS (y_Q) -- drive the red LEDs for each state
   BEGIN
      LEDR <= "000000000";
      case y_Q IS
         WHEN A =>   LEDR <= "000000001";
         WHEN B =>   LEDR <= "000000010";
         WHEN C =>   LEDR <= "000000100";
         WHEN D =>   LEDR <= "000001000";
         WHEN E =>   LEDR <= "000010000";
         WHEN F =>   LEDR <= "000100000";
         WHEN G =>   LEDR <= "001000000";
         WHEN H =>   LEDR <= "010000000";
         WHEN I =>   LEDR <= "100000000";
      END CASE;
   END PROCESS; -- LEDs

END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- a sequence detector FSM using a shift register
-- SW0 is the active low synchronous reset, SW1 is the w input, and KEY0 is the clock.
-- The z output appears on LEDG0, and the shift register FFs appear on LEDR3..0
-- a sequence detector shift register
-- inputs: Resetn is
ENTITY part3 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDG                        : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
            LEDR                        : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END part3;

ARCHITECTURE Behavior OF part3 IS
   SIGNAL Clock, Resetn, w, z : STD_LOGIC;
   SIGNAL S4_0s : STD_LOGIC_VECTOR(1 TO 4); -- shift register for recognizing 4 0s
   SIGNAL S4_1s : STD_LOGIC_VECTOR(1 TO 4); -- shift register for recognizing 4 1s
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   w <= SW(1);

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            S4_0s <= "1111";
            S4_1s <= "0000";
         ELSE
            S4_0s(1) <= w;
            S4_0s(2) <= S4_0s(1);
            S4_0s(3) <= S4_0s(2);
            S4_0s(4) <= S4_0s(3);

            S4_1s(1) <= w;
            S4_1s(2) <= S4_1s(1);
            S4_1s(3) <= S4_1s(2);
            S4_1s(4) <= S4_1s(3);
         END IF;
      END IF;
   END PROCESS;

   z <= '1' WHEN ((S4_0s = "0000") OR (S4_1s = "1111")) ELSE '0';
   LEDR(3 DOWNTO 0) <= S4_0s;
   LEDR(7 DOWNTO 4) <= S4_1s;
   LEDG(0) <= z;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- A mod-10 counter
-- inputs: SW0 is the active low synchronous reset, and KEY0 is the clock.
-- SW2 SW1 are the w1 w0 inputs.
-- output: if w1 w0 == 00, keep count the same
--    if w1 w0 == 01, increment count by 1
--    if w1 w0 == 10, increment count by 2
--    if w1 w0 == 11, decrement count by 1
-- drive count to digit HEX0
ENTITY part4 IS
   PORT (   SW                          : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            KEY                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
            HEX0                        : OUT STD_LOGIC_VECTOR(0 TO 6));
END part4;

ARCHITECTURE Behavior OF part4 IS
   COMPONENT bcd7seg
      PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn : STD_LOGIC;
   SIGNAL w : STD_LOGIC_VECTOR(1 DOWNTO 0);
   TYPE State_type IS (A, B, C, D, E, F, G, H, I, J);
   SIGNAL y_Q, Y_D : State_type;
   SIGNAL Count : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   Clock <= KEY(0);
   Resetn <= SW(0);
   w(1 DOWNTO 0) <= SW(2 DOWNTO 1);

   PROCESS (w, y_Q) -- state table
   BEGIN
      CASE y_Q IS
         WHEN A =>   CASE w IS
                        WHEN "00" => Y_D <= A;
                        WHEN "01" => Y_D <= B;
                        WHEN "10" => Y_D <= C;
                        WHEN "11" => Y_D <= J;
                     END CASE;
         WHEN B =>   CASE w IS
                        WHEN "00" =>  Y_D <= B;
                        WHEN "01" =>  Y_D <= C;
                        WHEN "10" =>  Y_D <= D;
                        WHEN "11" =>  Y_D <= A;
                     END CASE;
         WHEN C =>   CASE w IS
                        WHEN "00" =>  Y_D <= C;
                        WHEN "01" =>  Y_D <= D;
                        WHEN "10" =>  Y_D <= E;
                        WHEN "11" =>  Y_D <= B;
                     END CASE;
         WHEN D =>   CASE w IS
                        WHEN "00" =>  Y_D <= D;
                        WHEN "01" =>  Y_D <= E;
                        WHEN "10" =>  Y_D <= F;
                        WHEN "11" =>  Y_D <= C;
                     END CASE;
         WHEN E =>   CASE w IS
                        WHEN "00" =>  Y_D <= E;
                        WHEN "01" =>  Y_D <= F;
                        WHEN "10" =>  Y_D <= G;
                        WHEN "11" =>  Y_D <= D;
                     END CASE;
```

```
         WHEN F =>   CASE w IS
                        WHEN "00" =>  Y_D <= F;
                        WHEN "01" =>  Y_D <= G;
                        WHEN "10" =>  Y_D <= H;
                        WHEN "11" =>  Y_D <= E;
                     END CASE;
         WHEN G =>   CASE w IS
                        WHEN "00" =>  Y_D <= G;
                        WHEN "01" =>  Y_D <= H;
                        WHEN "10" =>  Y_D <= I;
                        WHEN "11" =>  Y_D <= F;
                     END CASE;
         WHEN H =>   CASE w IS
                        WHEN "00" =>  Y_D <= H;
                        WHEN "01" =>  Y_D <= I;
                        WHEN "10" =>  Y_D <= J;
                        WHEN "11" =>  Y_D <= G;
                     END CASE;
         WHEN I =>   CASE w IS
                        WHEN "00" =>  Y_D <= I;
                        WHEN "01" =>  Y_D <= J;
                        WHEN "10" =>  Y_D <= A;
                        WHEN "11" =>  Y_D <= H;
                     END CASE;
         WHEN J =>   CASE w IS
                        WHEN "00" =>  Y_D <= J;
                        WHEN "01" =>  Y_D <= A;
                        WHEN "10" =>  Y_D <= B;
                        WHEN "11" =>  Y_D <= I;
                     END CASE;
      END CASE;
   END PROCESS; -- state table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            y_Q <= A;
         ELSE
            y_Q <= Y_D;
         END IF;
      END IF;
   END PROCESS;

   PROCESS (y_Q) -- state outputs
   BEGIN
      CASE y_Q IS
         WHEN A =>   Count <= "0000";
         WHEN B =>   Count <= "0001";
         WHEN C =>   Count <= "0010";
         WHEN D =>   Count <= "0011";
         WHEN E =>   Count <= "0100";
         WHEN F =>   Count <= "0101";
         WHEN G =>   Count <= "0110";
         WHEN H =>   Count <= "0111";
         WHEN I =>   Count <= "1000";
         WHEN J =>   Count <= "1001";
      END CASE;
   END PROCESS; -- state output

   digit0: bcd7seg PORT MAP (Count, HEX0);
END Behavior;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY bcd7seg IS
   PORT (   bcd      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END bcd7seg;

ARCHITECTURE Behavior OF bcd7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (bcd)
   BEGIN
      CASE bcd IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN OTHERS => display <= "1111111";
      END CASE;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- scrolls the word HELLO across the 7-seg displays. An FSM inserts the
-- display values into a pipeline that drives the 8 displays.
-- inputs: KEY0 is the manual clock, and SW0 is the reset input
-- outputs: 7-seg displays HEX7 ... HEX0
ENTITY part5 IS
PORT (   SW                         : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT regne
      PORT (   R              : IN  STD_LOGIC_VECTOR(6 DOWNTO 0);
               Clock, Resetn  : STD_LOGIC;
               Q              : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn : STD_LOGIC;
   TYPE State_type IS (S0, S1, S2, S3, S4, S5, S6, S7, S8);
   SIGNAL y_Q, Y_D : State_type;
   SIGNAL FSM_char : STD_LOGIC_VECTOR(0 TO 6);
      -- input to pipeline regs comes from
      -- FSM_char for the first 8 clock cycles,
      -- and then comes from the pipeline's
      -- last stage (HELLO travels in a loop)
   SIGNAL pipe_select : STD_LOGIC;
   SIGNAL pipe_input : STD_LOGIC_VECTOR(6 DOWNTO 0);
   SIGNAL pipe0, pipe1, pipe2, pipe3, pipe4, pipe5,
      pipe6, pipe7 : STD_LOGIC_VECTOR(6 DOWNTO 0);

   CONSTANT H : STD_LOGIC_VECTOR(0 TO 6) := "1001000";
   CONSTANT E : STD_LOGIC_VECTOR(0 TO 6) := "0110000";
   CONSTANT L : STD_LOGIC_VECTOR(0 TO 6) := "1110001";
   CONSTANT O : STD_LOGIC_VECTOR(0 TO 6) := "0000001";
   CONSTANT Blank : STD_LOGIC_VECTOR(0 TO 6) := "1111111";
BEGIN

   Clock <= KEY(0);
   Resetn <= SW(0);

   PROCESS (y_Q) -- state table
   BEGIN
      CASE y_Q IS
         WHEN S0 =>  Y_D <= S1;
         WHEN S1 =>  Y_D <= S2;
         WHEN S2 =>  Y_D <= S3;
         WHEN S3 =>  Y_D <= S4;
         WHEN S4 =>  Y_D <= S5;
         WHEN S5 =>  Y_D <= S6;
         WHEN S6 =>  Y_D <= S7;
         WHEN S7 =>  Y_D <= S8;
         WHEN S8 =>  Y_D <= S8;
      END CASE;
   END PROCESS; -- state_table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            y_Q <= S0;
         ELSE
            y_Q <= Y_D;
```

```
         END IF;
      END IF;
   END PROCESS;

   PROCESS (y_Q) -- state outputs
   BEGIN
      pipe_select <= '0'; FSM_char <= "-------";
      CASE y_Q IS
         WHEN S0 =>  FSM_char <= H;
         WHEN S1 =>  FSM_char <= E;
         WHEN S2 =>  FSM_char <= L;
         WHEN S3 =>  FSM_char <= L;
         WHEN S4 =>  FSM_char <= O;
         WHEN S5 =>  FSM_char <= Blank;
         WHEN S6 =>  FSM_char <= Blank;
         WHEN S7 =>  FSM_char <= Blank;
         WHEN S8 =>  pipe_select <= '1';
      END CASE;
   END PROCESS; -- state output

   pipe_input <= FSM_char WHEN (pipe_select = '0') ELSE pipe7;
   -- regne (R, Clock, Resetn, E, Q);
   U0: regne PORT MAP (pipe_input, Clock, Resetn, pipe0);
   U1: regne PORT MAP (pipe0, Clock, Resetn, pipe1);
   U2: regne PORT MAP (pipe1, Clock, Resetn, pipe2);
   U3: regne PORT MAP (pipe2, Clock, Resetn, pipe3);
   U4: regne PORT MAP (pipe3, Clock, Resetn, pipe4);
   U5: regne PORT MAP (pipe4, Clock, Resetn, pipe5);
   U6: regne PORT MAP (pipe5, Clock, Resetn, pipe6);
   U7: regne PORT MAP (pipe6, Clock, Resetn, pipe7);

   HEX0 <= pipe0;
   HEX1 <= pipe1;
   HEX2 <= pipe2;
   HEX3 <= pipe3;
   HEX4 <= pipe4;
   HEX5 <= pipe5;
   HEX6 <= pipe6;
   HEX7 <= pipe7;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   PORT (   R              : IN  STD_LOGIC_VECTOR(6 DOWNTO 0);
            Clock, Resetn  : IN STD_LOGIC;
            Q              : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '1');
         ELSE
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
-- scrolls the word HELLO across the 7-seg displays. An FSM inserts the
-- display values into a pipeline that drives the 8 displays; each display
-- is driven for about 1 second before changing to the next character
-- inputs: 50 MHz clock, KEY0 is reset input
-- outputs: 7-seg displays HEX7 ... HEX0
ENTITY part6 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         CLOCK_50                   : IN  STD_LOGIC;
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
END part6;

ARCHITECTURE Behavior OF part6 IS
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn, Tick : STD_LOGIC;
   TYPE State_type IS (S0, S1, S2, S3, S4, S5, S6, S7, S8);
   SIGNAL y_Q, Y_D : State_type;
   SIGNAL FSM_char : STD_LOGIC_VECTOR(0 TO 6);
      -- input to pipeline regs comes from
      -- FSM_char for the first 8 clock cycles,
      -- and then comes from the pipeline's
      -- last stage (HELLO travels in a loop)
   SIGNAL pipe_select : STD_LOGIC;
   SIGNAL pipe_input : STD_LOGIC_VECTOR(0 TO 6);
   SIGNAL pipe0, pipe1, pipe2, pipe3, pipe4, pipe5,
      pipe6, pipe7 : STD_LOGIC_VECTOR(6 DOWNTO 0);
   SIGNAL slow_count : STD_LOGIC_VECTOR(23 DOWNTO 0);

   CONSTANT H : STD_LOGIC_VECTOR(0 TO 6) := "1001000";
   CONSTANT E : STD_LOGIC_VECTOR(0 TO 6) := "0110000";
   CONSTANT L : STD_LOGIC_VECTOR(0 TO 6) := "1110001";
   CONSTANT O : STD_LOGIC_VECTOR(0 TO 6) := "0000001";
   CONSTANT Blank : STD_LOGIC_VECTOR(6 DOWNTO 0) := "1111111";
BEGIN
   Clock <= CLOCK_50;
   Resetn <= KEY(0);

   -- A large counter to produce a 1 second (approx) enable, called Tick
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;
   Tick <= '1' WHEN (slow_count = 0) ELSE '0';

   PROCESS (y_Q, Tick) -- state table
   BEGIN
      CASE y_Q IS
         WHEN S0 =>  IF (Tick = '1') THEN Y_D <= S1;
                     ELSE Y_D <= S0;
                     END IF;
         WHEN S1 =>  IF (Tick = '1') THEN Y_D <= S2;
                     ELSE Y_D <= S1;
                     END IF;
         WHEN S2 =>  IF (Tick = '1') THEN Y_D <= S3;
```

```
                    ELSE Y_D <= S2;
                    END IF;
         WHEN S3 =>  IF (Tick = '1') THEN Y_D <= S4;
                    ELSE Y_D <= S3;
                    END IF;
         WHEN S4 =>  IF (Tick = '1') THEN Y_D <= S5;
                    ELSE Y_D <= S4;
                    END IF;
         WHEN S5 =>  IF (Tick = '1') THEN Y_D <= S6;
                    ELSE Y_D <= S5;
                    END IF;
         WHEN S6 =>  IF (Tick = '1') THEN Y_D <= S7;
                    ELSE Y_D <= S6;
                    END IF;
         WHEN S7 =>  IF (Tick = '1') THEN Y_D <= S8;
                    ELSE Y_D <= S7;
                    END IF;
         WHEN S8 =>  Y_D <= S8;
      END CASE;
   END PROCESS; -- state_table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            y_Q <= S0;
         ELSE
            y_Q <= Y_D;
         END IF;
      END IF;
   END PROCESS;

   PROCESS (y_Q) -- state outputs
   BEGIN
      pipe_select <= '0'; FSM_char <= "-------";
      CASE y_Q IS
         WHEN S0 =>  FSM_char <= H;
         WHEN S1 =>  FSM_char <= E;
         WHEN S2 =>  FSM_char <= L;
         WHEN S3 =>  FSM_char <= L;
         WHEN S4 =>  FSM_char <= O;
         WHEN S5 =>  FSM_char <= Blank;
         WHEN S6 =>  FSM_char <= Blank;
         WHEN S7 =>  FSM_char <= Blank;
         WHEN S8 =>  pipe_select <= '1';
      END CASE;
   END PROCESS; -- state output

   pipe_input <= FSM_char WHEN (pipe_select = '0') ELSE pipe7;
   -- regne (R, Clock, Resetn, E, Q);
   U0: regne PORT MAP (pipe_input, Clock, Resetn, Tick, pipe0);
   U1: regne PORT MAP (pipe0, Clock, Resetn, Tick, pipe1);
   U2: regne PORT MAP (pipe1, Clock, Resetn, Tick, pipe2);
   U3: regne PORT MAP (pipe2, Clock, Resetn, Tick, pipe3);
   U4: regne PORT MAP (pipe3, Clock, Resetn, Tick, pipe4);
   U5: regne PORT MAP (pipe4, Clock, Resetn, Tick, pipe5);
   U6: regne PORT MAP (pipe5, Clock, Resetn, Tick, pipe6);
   U7: regne PORT MAP (pipe6, Clock, Resetn, Tick, pipe7);

   HEX0 <= pipe0;
   HEX1 <= pipe1;
   HEX2 <= pipe2;
   HEX3 <= pipe3;
   HEX4 <= pipe4;
```

```
   HEX5 <= pipe5;
   HEX6 <= pipe6;
   HEX7 <= pipe7;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '1');
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
-- scrolls the word HELLO across the 7-seg displays. An FSM inserts the
-- display values into a pipeline that drives the 8 displays; each display
-- is driven for about 1 second before changing to the next character
-- inputs: 50 MHz clock, KEY0 is reset input, KEY1 doubles the speed of the
--   display, KEY2 halves the speed
-- outputs: 7-seg displays HEX7 ... HEX0
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY part7 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
         CLOCK_50                   : IN  STD_LOGIC;
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END part7;

ARCHITECTURE Behavior OF part7 IS
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   SIGNAL Clock, Resetn, Tick : STD_LOGIC;

   SIGNAL Fast, Slow : STD_LOGIC;   -- Variable tick interval controls
   -- Names of states for changing a counter value for implementing a variable
   -- tick delay
   TYPE Var_Delay_State_type IS (Sync3, Speed3, Sync4, Speed4, Sync5, Speed5, Sync1,
      Speed1, Sync2, Speed2);
   SIGNAL yV_Q, YV_D : Var_Delay_State_Type;
      -- This is the state machine that it used to implement
      -- a variable tick interval
   -- Names of states for the pipeline control machine
   TYPE State_type IS (S0, S1, S2, S3, S4, S5, S6, S7, S8);
   SIGNAL y_Q, Y_D : State_type;
   SIGNAL FSM_char : STD_LOGIC_VECTOR(0 TO 6);
      -- input to pipeline regs comes from
      -- FSM_char for the first 8 clock cycles,
      -- and then comes from the pipeline's
      -- last stage (HELLO travels in a loop)
   SIGNAL pipe_select : STD_LOGIC;
   SIGNAL pipe_input : STD_LOGIC_VECTOR(0 TO 6);
   SIGNAL pipe0, pipe1, pipe2, pipe3, pipe4,
      pipe5, pipe6, pipe7 : STD_LOGIC_VECTOR(0 TO 6);

   SIGNAL var_count, Modulus : STD_LOGIC_VECTOR(3 DOWNTO 0);
      -- used to implement a variable delay
   SIGNAL var_count_sync : STD_LOGIC;
   SIGNAL slow_count : STD_LOGIC_VECTOR(22 DOWNTO 0);

   CONSTANT H : STD_LOGIC_VECTOR(0 TO 6) := "1001000";
   CONSTANT E : STD_LOGIC_VECTOR(0 TO 6) := "0110000";
   CONSTANT L : STD_LOGIC_VECTOR(0 TO 6) := "1110001";
   CONSTANT O : STD_LOGIC_VECTOR(0 TO 6) := "0000001";
   CONSTANT Blank : STD_LOGIC_VECTOR(6 DOWNTO 0) := "1111111";

   SIGNAL KEY1_vector, KEY1_sync : STD_LOGIC_VECTOR(0 TO 0); -- For synchronizing KEY(
1)
   SIGNAL KEY2_vector, KEY2_sync : STD_LOGIC_VECTOR(0 TO 0); -- For synchronizing KEY(
2)
   SIGNAL Fast_vector, Slow_vector : STD_LOGIC_VECTOR(0 TO 0);
      -- Needed so we can use the regne function with N = 1 for a simple
```

```
      -- flipflop (VHDL type checking)
BEGIN
   Clock <= CLOCK_50;
   Resetn <= KEY(0);

   KEY1_vector(0) <= NOT (KEY(1));
   SYNCFAST1: regne GENERIC MAP (N => 1) PORT MAP
      (KEY1_vector, Clock, Resetn, '1', KEY1_sync);
   SYNCFAST2: regne GENERIC MAP (N => 1) PORT MAP
      (KEY1_sync, Clock, Resetn, '1', Fast_vector);
   Fast <= Fast_vector(0); -- silly VHDL type checking

   KEY2_vector(0) <= NOT (KEY(2));
   SYNCSLOW1: regne GENERIC MAP (N => 1) PORT MAP
      (KEY2_vector, Clock, Resetn, '1', KEY2_sync);
   SYNCSLOW2: regne GENERIC MAP (N => 1) PORT MAP
      (KEY2_sync, Clock, Resetn, '1', Slow_vector);
   Slow <= Slow_vector(0); -- needed for silly VHDL type checking

   -- state machine that produces a variable delay
   -- Each state will produce an output value that is used to modulo
   -- a counter value. Speed 5 gives the lowest modulo value, and hence the
   -- smallest delay, while Speed 1 gives the largest modulo value, and
   -- hence the largest delay. There is a synchronization state before each
   -- Speed state so that we can wait for the slow switch pressing to end
   PROCESS (yV_Q, Fast, Slow) -- state table speed
   BEGIN
      CASE yV_Q IS
         WHEN Sync5  => IF ((Slow = '1') OR (Fast = '1')) THEN
                           YV_D <= Sync5; -- wait for any depressed key to be released
                        ELSE
                           YV_D <= Speed5;
                        END IF;
         WHEN Speed5 => IF (Slow = '0') THEN
                           YV_D <= Speed5;   -- fastest speed
                        ELSE
                           YV_D <= Sync4; -- change to slower speed
                        END IF;
         WHEN Sync4  => IF ((Slow = '1') OR (Fast = '1')) THEN
                           YV_D <= Sync4; -- wait for a depressed key to be released
                        ELSE
                           YV_D <= Speed4;
                        END IF;
         WHEN Speed4 => IF ((Slow = '0') AND (Fast = '0')) THEN
                           YV_D <= Speed4;   -- keep this speed
                        ELSIF (Slow = '1') THEN
                           YV_D <= Sync3;       -- change to slower speed
                        ELSE
                           YV_D <= Sync5;    -- change to faster speed
                        END IF;
         WHEN Sync3  => IF ((Slow = '1') OR (Fast = '1')) THEN
                           YV_D <= Sync3; -- wait for a depressed key to be released
                        ELSE
                           YV_D <= Speed3;
                        END IF;
         WHEN Speed3 => IF ((Slow = '0') AND (Fast = '0')) THEN
                           YV_D <= Speed3;   -- keep this speed
                        ELSIF (Slow = '1') THEN
                           YV_D <= Sync2;       -- change to slower speed
                        ELSE
                           YV_D <= Sync4;       -- change to faster speed
                        END IF;
         WHEN Sync2  => IF ((Slow = '1') OR (Fast = '1')) THEN
                           YV_D <= Sync2; -- wait for a depressed key to be released
```

```
                        ELSE
                           YV_D <= Speed2;
                        END IF;
         WHEN Speed2 => IF ((Slow = '0') AND (Fast = '0')) THEN
                           YV_D <= Speed2;   -- keep this speed
                        ELSIF (Slow = '1') THEN
                           YV_D <= Sync1;       -- change to slower speed
                        ELSE
                           YV_D <= Sync3;       -- change to faster speed
                        END IF;
         WHEN Sync1  => IF ((Slow = '1') OR (Fast = '1')) THEN
                           YV_D <= Sync1; -- wait for a depressed key to be released
                        ELSE
                           YV_D <= Speed1;
                        END IF;
         WHEN Speed1 => IF (Fast = '0') THEN
                           YV_D <= Speed1;      -- keep this speed
                        ELSE
                           YV_D <= Sync2;       -- change to faster speed
                        END IF;
      END CASE;
   END PROCESS; -- state_table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            yV_Q <= Speed3;
         ELSE
            yV_Q <= YV_D;
         END IF;
      END IF;
   END PROCESS;

   PROCESS (yV_Q)
   BEGIN -- state_outputs_speed
      Modulus <= "----"; var_count_sync <= '1';
      CASE yV_Q IS
         WHEN Sync5  => var_count_sync <= '0';
         WHEN Speed5 => Modulus <= "0000";
         WHEN Sync4  => var_count_sync <= '0';
         WHEN Speed4 => Modulus <= "0001";
         WHEN Sync3  => var_count_sync <= '0';
         WHEN Speed3 => Modulus <= "0011";
         WHEN Sync2  => var_count_sync <= '0';
         WHEN Speed2 => Modulus <= "0110";
         WHEN Sync1  => var_count_sync <= '0';
         WHEN Speed1 => Modulus <= "1100";
      END CASE;
   END PROCESS; -- state_table

   LEDG(3 DOWNTO 0) <= Modulus;

   -- A large counter to produce a .25 second (approx) enable, called Tick
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- Counter that provides a variable delay
   PROCESS (Clock)
   BEGIN
```

```
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (var_count_sync = '0') THEN
            var_count <= (OTHERS => '0');
         ELSIF (slow_count = 0) THEN
            IF (var_count = Modulus) THEN
               var_count <= (OTHERS => '0');
            ELSE
               var_count <= var_count + '1';
            END IF;
         END IF;
      END IF;
   END PROCESS;

   -- Tick advances the scrolling letters when var_count = slow_count = 0.
   -- The var_count_sync is used to prevent scrolling when a Fast or Slow
   -- key is being held down.
   Tick <= '1' WHEN ((var_count = 0) AND (slow_count = 0) AND (var_count_sync = '1'))
ELSE '0';

   PROCESS (y_Q, Tick) -- state table
   BEGIN
      CASE y_Q IS
         WHEN S0 =>  IF (Tick = '1') THEN Y_D <= S1;
                     ELSE Y_D <= S0;
                     END IF;
         WHEN S1 =>  IF (Tick = '1') THEN Y_D <= S2;
                     ELSE Y_D <= S1;
                     END IF;
         WHEN S2 =>  IF (Tick = '1') THEN Y_D <= S3;
                     ELSE Y_D <= S2;
                     END IF;
         WHEN S3 =>  IF (Tick = '1') THEN Y_D <= S4;
                     ELSE Y_D <= S3;
                     END IF;
         WHEN S4 =>  IF (Tick = '1') THEN Y_D <= S5;
                     ELSE Y_D <= S4;
                     END IF;
         WHEN S5 =>  IF (Tick = '1') THEN Y_D <= S6;
                     ELSE Y_D <= S5;
                     END IF;
         WHEN S6 =>  IF (Tick = '1') THEN Y_D <= S7;
                     ELSE Y_D <= S6;
                     END IF;
         WHEN S7 =>  IF (Tick = '1') THEN Y_D <= S8;
                     ELSE Y_D <= S7;
                     END IF;
         WHEN S8 =>  Y_D <= S8;
      END CASE;
   END PROCESS; -- state_table

   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            y_Q <= S0;
         ELSE
            y_Q <= Y_D;
         END IF;
      END IF;
   END PROCESS;

   PROCESS (y_Q) -- state outputs
   BEGIN
      pipe_select <= '0'; FSM_char <= "-------";
```

```
      CASE y_Q IS
         WHEN S0 =>  FSM_char <= H;
         WHEN S1 =>  FSM_char <= E;
         WHEN S2 =>  FSM_char <= L;
         WHEN S3 =>  FSM_char <= L;
         WHEN S4 =>  FSM_char <= O;
         WHEN S5 =>  FSM_char <= Blank;
         WHEN S6 =>  FSM_char <= Blank;
         WHEN S7 =>  FSM_char <= Blank;
         WHEN S8 =>  pipe_select <= '1';
      END CASE;
   END PROCESS; -- state output

   pipe_input <= FSM_char WHEN (pipe_select = '0') ELSE pipe7;
   -- regne (R, Clock, Resetn, E, Q);
   U0: regne PORT MAP (pipe_input, Clock, Resetn, Tick, pipe0);
   U1: regne PORT MAP (pipe0, Clock, Resetn, Tick, pipe1);
   U2: regne PORT MAP (pipe1, Clock, Resetn, Tick, pipe2);
   U3: regne PORT MAP (pipe2, Clock, Resetn, Tick, pipe3);
   U4: regne PORT MAP (pipe3, Clock, Resetn, Tick, pipe4);
   U5: regne PORT MAP (pipe4, Clock, Resetn, Tick, pipe5);
   U6: regne PORT MAP (pipe5, Clock, Resetn, Tick, pipe6);
   U7: regne PORT MAP (pipe6, Clock, Resetn, Tick, pipe7);

   HEX0 <= pipe0;
   HEX1 <= pipe1;
   HEX2 <= pipe2;
   HEX3 <= pipe3;
   HEX4 <= pipe4;
   HEX5 <= pipe5;
   HEX6 <= pipe6;
   HEX7 <= pipe7;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '1');
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

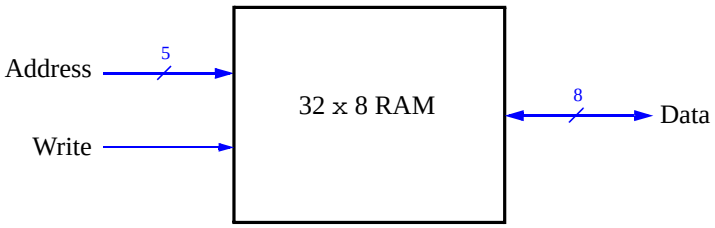
Laboratory Exercise 8

Memory Blocks

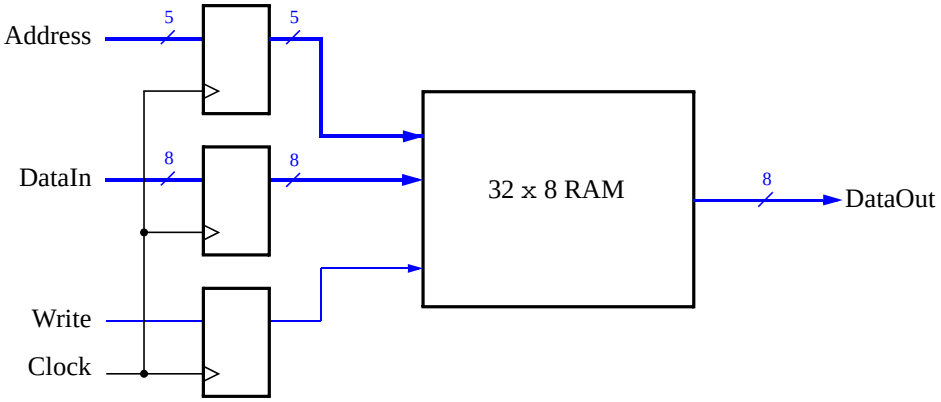
In computer systems it is necessary to provide a substantial amount of memory. If a system is implemented using FPGA technology it is possible to provide some amount of memory by using the memory resources that exist in the FPGA device. If additional memory is needed, it has to be implemented by connecting external memory chips to the FPGA. In this exercise we will examine the general issues involved in implementing such memory.

A diagram of the random access memory (RAM) module that we will implement is shown in Figure 1*a*. It contains 32 eight-bit words (rows), which are accessed using a five-bit *address* port, an eight-bit *data* port, and a *write* control input. We will consider two different ways of implementing this memory: using dedicated memory blocks in an FPGA device, and using a separate memory chip.

The Cyclone II 2C35 FPGA that is included on the DE2 board provides dedicated memory resources called *M4K blocks*. Each M4K block contains 4096 memory bits, which can be configured to implement memories of various sizes. A common term used to specify the size of a memory is its *aspect ratio*, which gives the *depth* in words and the *width* in bits (depth x width). Some aspect ratios supported by the M4K block are 4K x 1, 2K x 2, 1K x 4, and 512 x 8. We will utilize the 512 x 8 mode in this exercise, using only the first 32 words in the memory. We should also mention that many other modes of operation are supported in an M4K block, but we will not discuss them here.



(*a*) RAM organization



(b) RAM implementation

Figure 1. A 32 x 8 RAM module.

There are two important features of the M4K block that have to be mentioned. First, it includes registers that can be used to synchronize all of the input and output signals to a clock input. Second, the M4K block has separate ports for data being written to the memory and data being read from the memory. A requirement for using the M4K block is that either its input ports, output port, or both, have to be synchronized to a clock input. Given these

requirements, we will implement the modified 32 x 8 RAM module shown in Figure 1*b*. It includes registers for the *address*, *data input*, and *write* ports, and uses a separate unregistered *data output* port.

Part I

Commonly used logic structures, such as adders, registers, counters and memories, can be implemented in an FPGA chip by using LPM modules from the Quartus II Library of Parameterized Modules. Altera recommends that a RAM module be implemented by using the *altsyncram* LPM. In this exercise you are to use this LPM to implement the memory module in Figure 1*b*.

1. Create a new Quartus II project to implement the memory module. Select as the target chip the Cyclone II EP2C35F672C6, which is the FPGA chip on the Altera DE2 board.

2. You can learn how the MegaWizard Plug-in Manager is used to generate a desired LPM module by reading the tutorial *Using Library Modules in VHDL Designs*. This tutorial is provided in the University Program section of Altera's web site. In the first screen of the MegaWizard Plug-in Manager choose the *altsyncram* LPM, which is found under the storage category. As indicated in Figure 2, select VHDL HDL as the type of output file to create, and give the file the name *ramlpm.vhd*. On the next page of the Wizard specify a memory size of 32 eight-bit words, and select M4K as the type of RAM block. Advance to the subsequent page and accept the default settings to use a single clock for the RAM's registers, and then advance again to the page shown in Figure 3. On this page *deselect* the setting called Read output port(s) under the category Which ports should be registered?. This setting creates a RAM module that matches the structure in Figure 1*b*, with registered input ports and unregistered output ports. Accept defaults for the rest of the settings in the Wizard, and then instantiate in your top-level VHDL file the entity generated in *ramlpm.vhd*. Include appropriate input and output signals in your VHDL code for the memory ports given in Figure 1*b*.

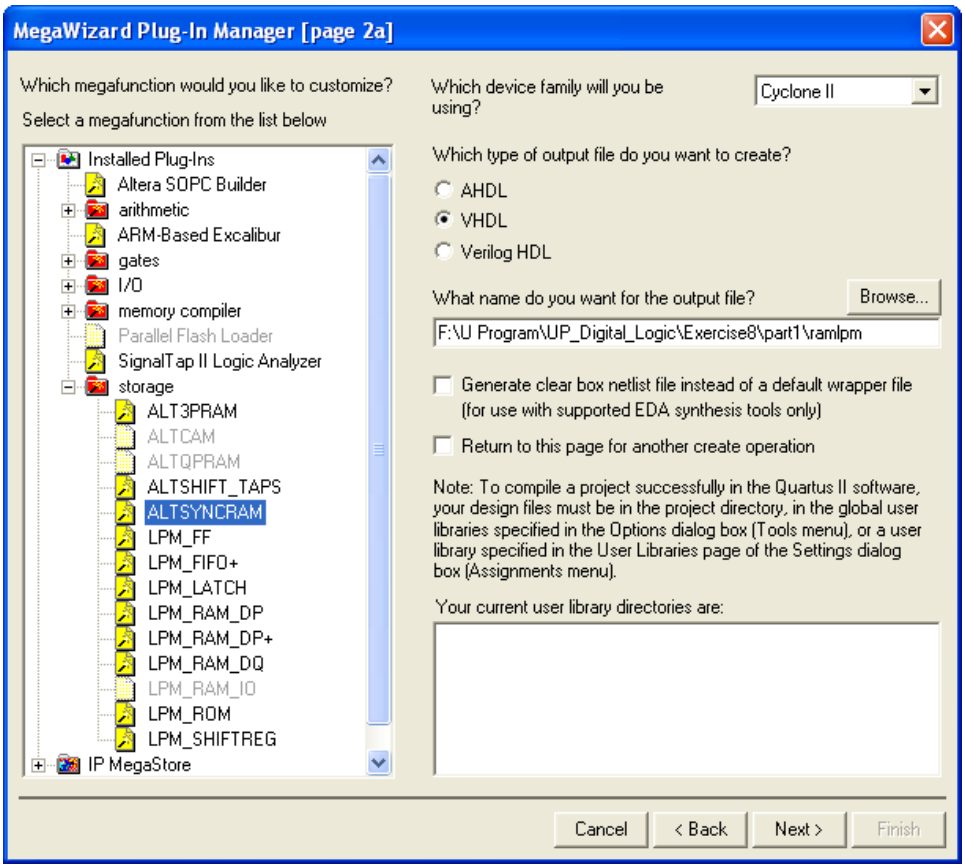


Figure 2. Choosing the *altsyncram* LPM.

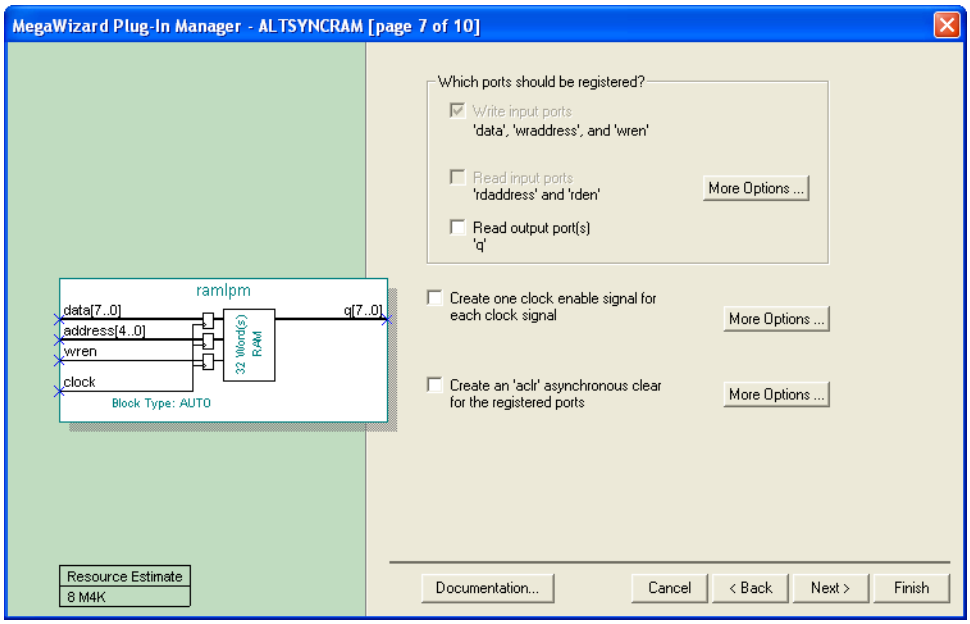


Figure 3. Configuring input and output ports on the *altsyncram* LPM.

3. Compile the circuit. Observe in the Compilation Report that the Quartus II Compiler uses 256 bits in one of the M4K memory blocks to implement the RAM circuit.
4. Simulate the behavior of your circuit and ensure that you can read and write data in the memory.

Part II

Now, we want to realize the memory circuit in the FPGA on the DE2 board, and use toggle switches to load some data into the created memory. We also want to display the contents of the RAM on the 7-segment displays.

1. Make a new Quartus II project which will be used to implement the desired circuit on the DE2 board.
2. Create another VHDL file that instantiates the *ramlpm* module and that includes the required input and output pins on the DE2 board. Use toggle switches SW_{7-0} to input a byte of data into the RAM location identified by a 5-bit address specified with toggle switches SW_{15-11}. Use SW_{17} as the *Write* signal and use KEY_0 as the *Clock* input. Display the value of the *Write* signal on $LEDG_0$. Show the address value on the 7-segment displays *HEX7* and *HEX6*, show the data being input to the memory on *HEX5* and *HEX4*, and show the data read out of the memory on *HEX1* and *HEX0*.
3. Test your circuit and make sure that all 32 locations can be loaded properly.

Part III

Instead of directly instantiating the LPM module, we can implement the required memory by specifying its structure in the VHDL code. In a VHDL-specified design it is possible to define the memory as a multidimensional array. A 32 x 8 array, which has 32 words with 8 bits per word, can be declared by the statements

```
TYPE mem IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL memory_array : mem;
```

In the Cyclone II FPGA, such an array can be implemented either by using the flip-flops that each logic element contains or, more efficiently, by using the M4K blocks. There are two ways of ensuring that the M4K blocks will be used. One is to use an LPM module from the Library of Parameterized Modules, as we saw in Part I. The other is to define the memory requirement by using a suitable style of VHDL code from which the Quartus II compiler can infer that a memory block should be used. Quartus II Help shows how this may be done with examples of VHDL code (search in the Help for "Inferred memory").

Perform the following steps:

1. Create a new project which will be used to implement the desired circuit on the DE2 board.
2. Write a VHDL file that provides the necessary functionality, including the ability to load the RAM and read its contents as done in Part II.
3. Assign the pins on the FPGA to connect to the switches and the 7-segment displays.
4. Compile the circuit and download it into the FPGA chip.
5. Test the functionality of your design by applying some inputs and observing the output. Describe any differences you observe in comparison to the circuit from Part II.

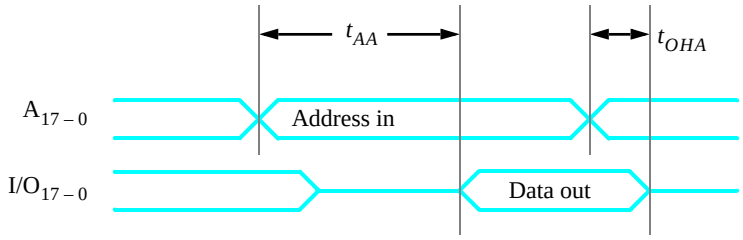
Part IV

The DE2 board includes an SRAM chip, called IS61LV25616AL-10, which is a static RAM having a capacity of 256K 16-bit words. The SRAM interface consists of an 18-bit address port, A_{17-0}, and a 16-bit bidirectional data port, I/O_{15-0}. It also has several control inputs, $\overline{CE}$, $\overline{OE}$, $\overline{WE}$, $\overline{UB}$, and $\overline{LB}$, which are described in Table 1.

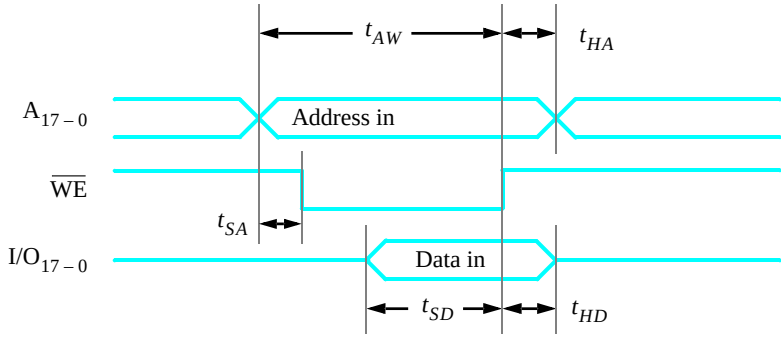
Name	Purpose
$\overline{CE}$	Chip enable−asserted low during all SRAM operations
$\overline{OE}$	Output enable−can be asserted low during only read operations, or during all operations
$\overline{WE}$	Write enable−asserted low during a write operation
$\overline{UB}$	Upper byte−asserted low to read or write the upper byte of an address
$\overline{LB}$	Lower byte−asserted low to read or write the lower byte of an address

Table 1. SRAM control inputs.

The operation of the IS61LV25616AL chip is described in its data sheet, which can obtained from the DE2 System CD that is included with the DE2 board, or by performing an Internet search. The data sheet describes a number of modes of operation of the memory and lists many timing parameters related to its use. For the purposes of this exercise a simple operating mode is to always assert (set to 0) the control inputs $\overline{CE}$, $\overline{OE}$, $\overline{UB}$, and $\overline{LB}$, and then to control reading and writing of the memory by using only the $\overline{WE}$ input. Simplified timing diagrams that correspond to this mode are given in Figure 4. Part (*a*) shows a read cycle, which begins when a valid address appears on A_{17-0} and the $\overline{WE}$ input is not asserted. The memory places valid data on the I/O_{15-0} port after the *address access* delay, t_{AA}. When the read cycle ends because of a change in the address value, the output data remains valid for the *output hold* time, t_{OHA}.



(*a*) SRAM read cycle timing



(*b*) SRAM write cycle timing

Figure 4. SRAM read and write cycles.

Figure 4*b* gives the timing for a write cycle. It begins when $\overline{WE}$ is set to 0, and it ends when $\overline{WE}$ is set back to 1. The address has to be valid for the *address setup* time, t_{AW}, and the data to be written has to be valid for the *data setup* time, t_{SD}, before the rising edge of $\overline{WE}$. Table 2 lists the minimum and maximum values of all timing parameters shown in Figure 4.

	Value	
Parameter	Min	Max
t_{AA}	–	10 ns
t_{OHA}	3 ns	–
t_{AW}	8 ns	–
t_{SD}	6 ns	–
t_{HA}	0	–
t_{SA}	0	–
t_{HD}	0	–

Table 2. SRAM timing parameter values.

You are to realize the 32 x 8 memory in Figure 1*a* by using the SRAM chip. It is a good approach to include in your design the registers shown in Figure 1*b*, by implementing these registers in the FPGA chip. Be careful to implement properly the bidirectional data port that connects to the memory.

1. Create a new Quartus II project for your circuit. Write a VHDL file that provides the necessary functionality, including the ability to load the memory and read its contents. Use the same switches, LEDs, and 7-segment displays on the DE2 board as in Parts II and III, and use the SRAM pin names shown in Table 3 to interface your circuit to the IS61LV25616AL chip (the SRAM pin names are also given in the *DE2 User Manual*).

Note that you will not use all of the address and data ports on the IS61LV25616AL chip for your 32 x 8 memory; connect the unneeded ports to 0 in your VHDL entity.

SRAM port name	DE2 pin name
A_{17-0}	SRAM_ADDR$_{17-0}$
I/O$_{15-0}$	SRAM_DQ$_{15-0}$
$\overline{CE}$	SRAM_CE_N
$\overline{OE}$	SRAM_OE_N
$\overline{WE}$	SRAM_WE_N
$\overline{UB}$	SRAM_UB_N
$\overline{LB}$	SRAM_LB_N

Table 3. DE2 pin names for the SRAM chip.

2. Compile the circuit and download it into the FPGA chip.

3. Test the functionality of your design by reading and writing values to several different memory locations.

Part V

The SRAM block in Figure 1 has a single port that provides the address for both read and write operations. For this part you will create a different type of memory module, in which there is one port for supplying the address for a read operation, and a separate port that gives the address for a write operation. Perform the following steps.

1. Create a new Quartus II project for your circuit. To generate the desired memory module open the MegaWizard Plug-in Manager and select again the *altsyncram* LPM in the storage category. On Page 1 of the Wizard choose the setting With one read port and one write port (simple dual-port mode) in the category called How will you be using the altsyncram?. Advance through Pages 2 to 5 and make the same choices as in Part II. On Page 6 choose the setting I don't care in the category Mixed Port Read-During-Write for Single Input Clock RAM. This setting specifies that it does not matter whether the memory outputs the new data being written, or the old data previously stored, in the case that the write and read addresses are the same.

   Page 7 of the Wizard is displayed in Figure 5. It makes use of a feature that allows the memory module to be loaded with initial data when the circuit is programmed into the FPGA chip. As shown in the figure, choose the setting Yes, use this file for the memory content data, and specify the filename *ramlpm.mif*. To learn about the format of a *memory initialization file* (MIF), see the Quartus II Help. You will need to create this file and specify some data values to be stored in the memory. Finish the Wizard and then examine the generated memory module in the file *ramlpm.vhd*.

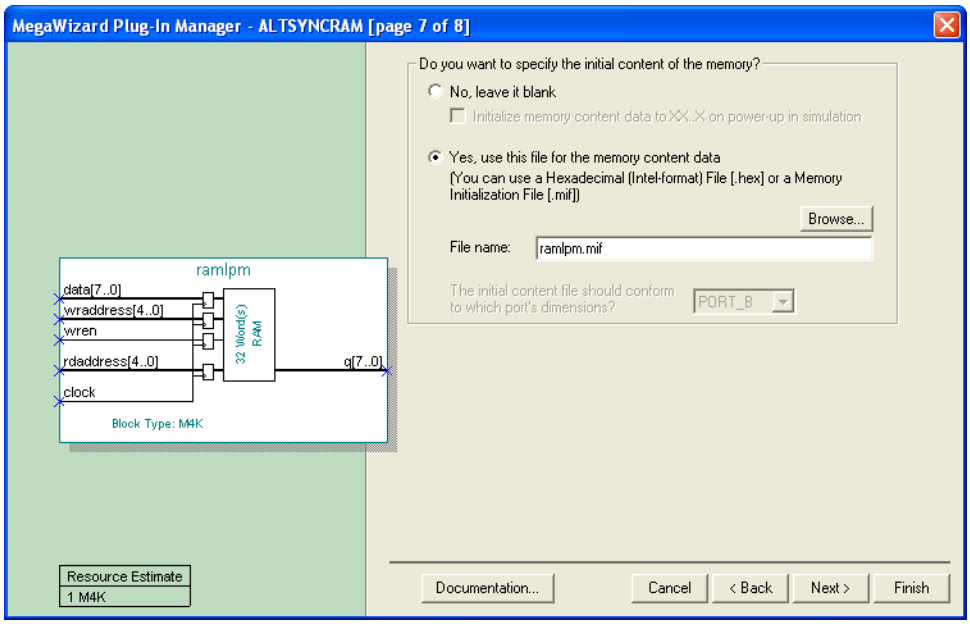


Figure 5. Specifying a memory initialization file (MIF).

2. Write a VHDL file that instantiates your dual-port memory. To see the RAM contents, add to your design a capability to display the content of each byte (in hexadecimal format) on the 7-segment displays *HEX1* and *HEX0*. Scroll through the memory locations by displaying each byte for about one second. As each byte is being displayed, show its address (in hex format) on the 7-segment displays *HEX3* and *HEX2*. Use the 50 MHz clock, *CLOCK_50*, on the DE2 board, and use KEY_0 as a reset input. For the write address and corresponding data use the same switches, LEDs, and 7-segment displays as in the previous parts of this exercise. Make sure that you properly synchronize the toggle switch inputs to the 50 MHz clock signal.

3. Test your circuit and verify that the initial contents of the memory match your *ramlpm.mif* file. Make sure that you can independently write data to any address by using the toggle switches.

Part VI

The dual-port memory created in Part V allows simultaneous read and write operations to occur, because it has two address ports. In this part of the exercise you should create a similar capability, but using a single-port RAM. Since there will be only one address port you will need to use multiplexing to select either a read or write address at any specific time. Perform the following steps.

1. Create a new Quartus II project for your circuit, and use the MegaWizard Plug-in Manager to again create a single-port version of the *altsyncram* LPM. For Pages 1 to 6 of the Wizard use the same settings as in Part I. On Page 7, shown in Figure 6, specify the *ramlpm.mif* file as you did in Part V, but also make the setting Allow In-System Memory Content Editor to capture and update content independently of the system clock. This option allows you to use a feature of the Quartus II CAD system called the In-System Memory Content Editor to view and manipulate the contents of the created RAM module. When using this tool you can optionally specify a four-character 'Instance ID' that serves as a name for the memory; in Figure 7 we gave the RAM module the name 32x8. Complete the final steps in the Wizard.

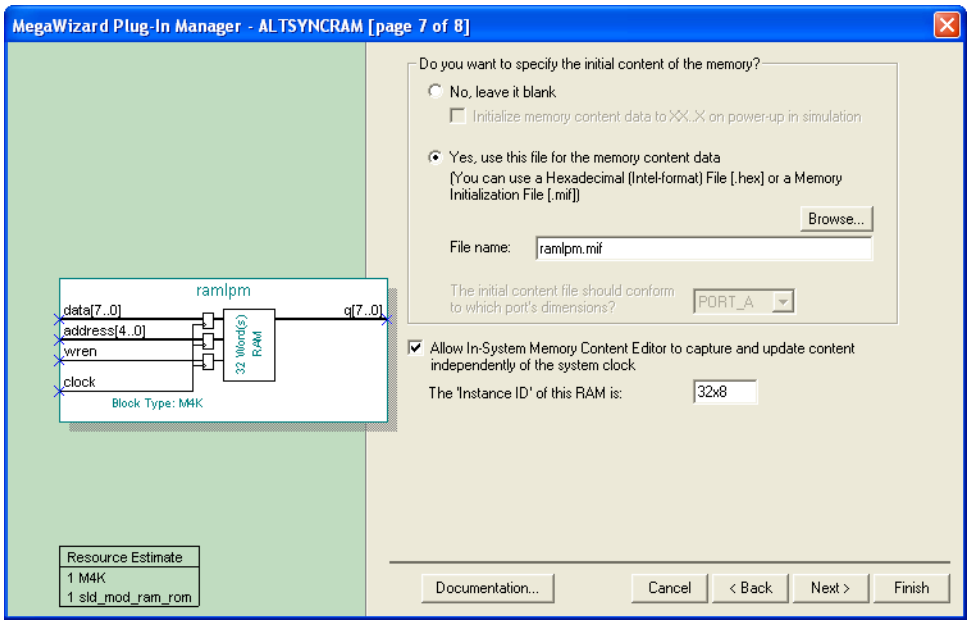


Figure 6. Configuring *altsyncram* for use with the In-System Memory Content Editor.

2. Write a VHDL file that instantiates your memory module. Include in your design the ability to scroll through the memory locations as in Part V. Use the same switches, LEDs, and 7-segment displays as you did previously.

3. Before you can use the In-System Memory Content Editor tool, one additional setting has to be made. In the Quartus II software select Assignments > Settings to open the window in Figure 7, and then open the item called Default Parameters under Analysis and Synthesis Settings. As shown in the figure, type the parameter name CYCLONEII_SAFE_WRITE and assign the value RESTRUCTURE. This parameter allows the Quartus II synthesis tools to modify the single-port RAM as needed to allow reading and writing of the memory by the In-System Memory Content Editor tool. Click OK to exit from the Settings window.

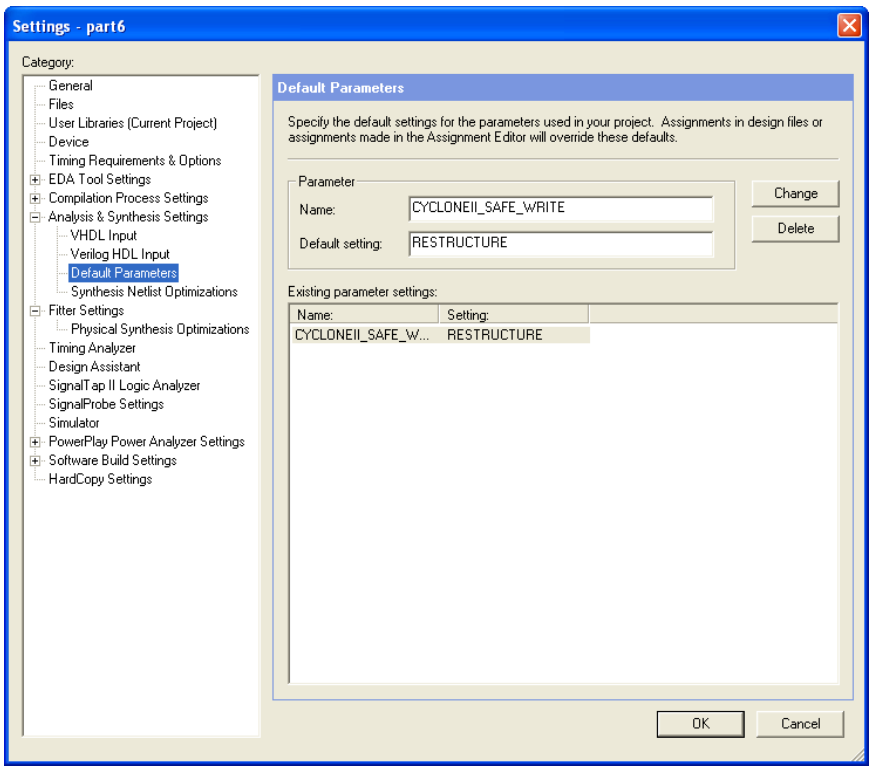


Figure 7. Setting the *CYCLONEII_SAFE_WRITE* parameter.

4. Compile your code and download the circuit onto the DE2 board. Test the circuit's operation and ensure that read and write operations work properly. Describe any differences you observe from the behavior of the circuit in Part V.

5. Select Tools > In-System Memory Content Editor, which opens the window in Figure 8. To specify the connection to your DE2 board click on the Setup button on the right side of the screen. In the window in Figure 9 select the USB-Blaster hardware, and then close the Hardware Setup dialog.

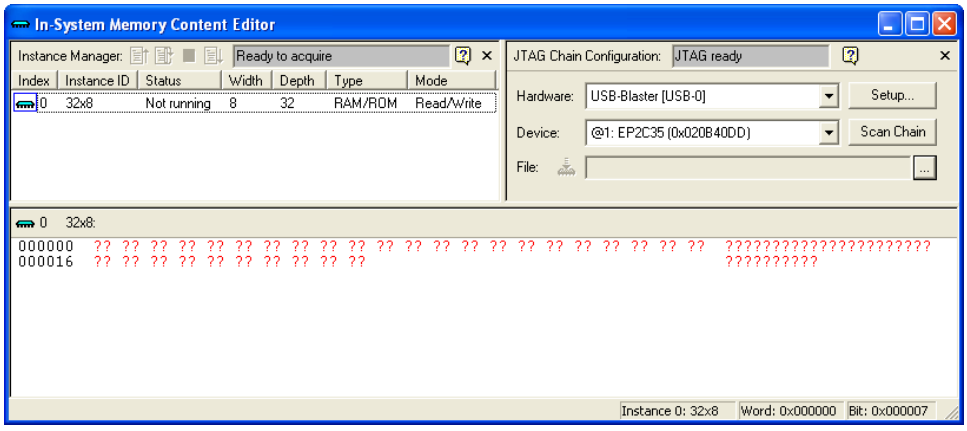


Figure 8. The In-System Memory Content Editor window.

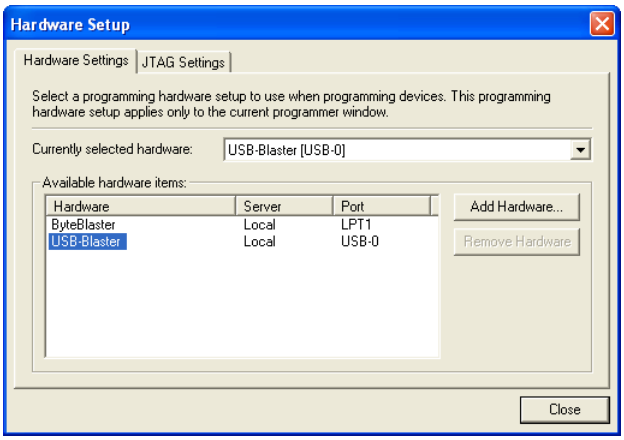


Figure 9. The Hardware Setup window.

Instructions for using the In-System Memory Content Editor tool can be found in the Quartus II Help. A simple operation is to right-click on the 32x8 memory module, as indicated in Figure 10, and select Read Data from In-System Memory. This action causes the contents of the memory to be displayed in the bottom part of the window. You can then edit any of the displayed values by typing over them. To actually write the new value to the RAM, right click again on the 32x8 memory module and select Write All Modified Words to In-System Memory.

Experiment by changing some memory values and observing that the data is properly displayed both on the 7-segment displays on the DE2 board and in the In-System Memory Content Editor window.

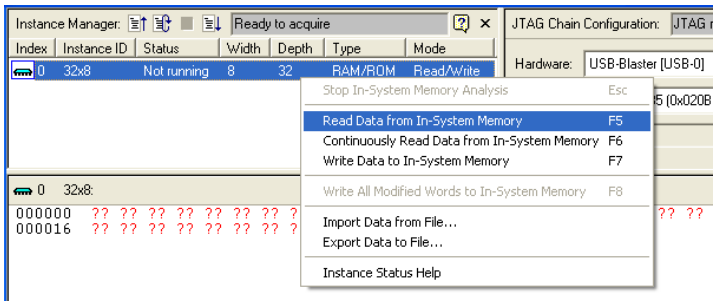


Figure 10. Using the In-System Memory Content Editor tool.

Part VII

For this part you are to modify your circuit from Part VI (and Part IV) to use the IS61LV25616AL SRAM chip instead of an M4K block. Create a Quartus II project for the new design, compile it, download it onto the DE2 boards, and test the circuit.

In Part VI you used a memory initialization file to specify the initial contents of the 32 x 8 RAM block, and you used the In-System Memory Content Editor tool to read and modify this data. This approach can be used only for the memory resources inside the FPGA chip. To perform equivalent operations using the external SRAM chip you can use a special capability of the DE2 board called the *DE2 Control Panel*. Chapter 3 of the *DE2 User Manual* shows how to use this tool. The procedure involves programming the FPGA with a special circuit that communicates with the Control Panel software application, which is illustrated in Figure 11, and using this setup

to load data into the SRAM chip. Subsequently, you can reprogram the FPGA with your own circuit, which will then have access to the data stored in the SRAM chip (reprogramming the FPGA has no effect on the external memory). Experiment with this capability and ensure that the results of read and write operations to the SRAM chip can be observed both in the your circuit and in the DE2 Control Panel software.

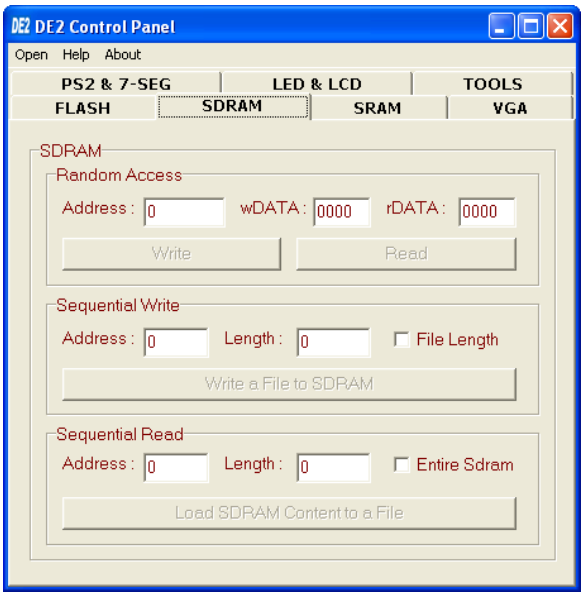


Figure 11. The DE2 Control Panel software.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- LPM RAM module
-- inputs:
--    Clock
--    Address
--    Write: asserted to perform a write
--    DataIn: data to be written
--
-- outputs:
--    DataOut: data read
ENTITY part1 IS
   PORT (   Clock, Write      : IN  STD_LOGIC;
            DataIn            : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            Address           : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
            DataOut           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT ramlpm
      PORT (   address  : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
BEGIN
   -- instantiate LPM module
   -- module ramlpm (address, clock, data, wren, q)
   m32x8: ramlpm PORT MAP (Address, Clock, DataIn, Write, DataOut);
END Behavior;
```

```
-- This code instantiates a 32 x 8 memory n the Cyclone II FPGA on the DE2 board.
--
-- inputs: KEY0 is the clock, SW7-SW0 provides data to write into memory.
-- SW15-SW11 provides the memory address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part2 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT ramlpm
      PORT (   address  : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Write : STD_LOGIC;
   SIGNAL Address : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL DataIn, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   Clock <= KEY(0);
   Write <= SW(17);
   DataIn <= SW(7 DOWNTO 0);
   Address <= SW(15 DOWNTO 11);

   -- instantiate LPM module
   -- module ramlpm (address, clock, data, wren, q)
   m32x8: ramlpm PORT MAP (Address, Clock, DataIn, Write, DataOut);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
   HEX2 <= "1111111";  -- blank
   HEX3 <= "1111111";  -- blank
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Address(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Address(4)), HEX7);

   LEDG(0) <= Write;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- the B input blanks the display when B = 1
ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;
```

```
ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --        0
   --       ---
   --      |   |
   --    5|   |1
   --      | 6 |
   --       ---
   --      |   |
   --    4|   |2
   --      |   |
   --       ---
   --        3
   --
   PROCESS (hex)
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
-- This code instantiates a 32 x 8 memory n the Cyclone II FPGA on the DE2 board.
--
-- inputs: KEY0 is the clock, SW7-SW0 provides data to write into memory.
-- SW15-SW11 provides the memory address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY part3 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
END part3;

ARCHITECTURE Behavior OF part3 IS
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Write : STD_LOGIC;
   SIGNAL Address, Address_reg : INTEGER RANGE 0 to 31;
   SIGNAL Address_STD : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL DataIn, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);

   -- declare memory array
   TYPE mem IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL memory_array : mem;
BEGIN
   Clock <= KEY(0);
   Write <= SW(17);
   DataIn <= SW(7 DOWNTO 0);
   Address_STD <= SW(15 DOWNTO 11);
   Address <= CONV_INTEGER(Address_STD);

   -- infer RAM module
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Write = '1') THEN
            memory_array(Address) <= DataIn;
         END IF;
         Address_reg <= Address;
      END IF;
   END PROCESS;

   DataOut <= memory_array(Address_reg);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
   HEX2 <= "1111111";
   HEX3 <= "1111111";
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Address_STD(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Address_STD(4)), HEX7);

   LEDG(0) <= Write;
```

```
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
-- This code implements a single port memory using the external SRAM chip.
--
-- inputs: KEY0 is the reset, KEY1 is the clock, SW7-SW0 provides data to
-- write into memory,
-- SW15-SW11 provides the memory address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
-- SRAM_ADDR provides the external SRAM chip address, SRAM_DQ is the data
-- input/output for the RAM, and the SRAM control signals are SRAM_WE_N,
-- SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, and SRAM_LB_N.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part4 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
         SRAM_ADDR                  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
         SRAM_DQ                    : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         SRAM_WE_N                  : BUFFER STD_LOGIC;
         SRAM_CE_N, SRAM_OE_N       : OUT STD_LOGIC;
         SRAM_UB_N, SRAM_LB_N       : OUT STD_LOGIC);
END part4;

ARCHITECTURE Behavior OF part4 IS
   COMPONENT flip_flop
      PORT (   R                 : IN  STD_LOGIC;
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Resetn, Clock, Write, CE : STD_LOGIC;
   SIGNAL Address : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL DataIn, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL DataIn_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= KEY(1);
   Write <= SW(17);
   R1: flip_flop PORT MAP (NOT (Write), Clock, Resetn, '1', SRAM_WE_N);

   Address <= SW(15 DOWNTO 11);
   R2: regne GENERIC MAP (N => 5) PORT MAP (Address, Clock, Resetn, '1',
      SRAM_ADDR(4 DOWNTO 0));
   SRAM_ADDR(17 DOWNTO 5) <= "0000000000000";

   DataIn <= SW(7 DOWNTO 0);
   R3: regne  GENERIC MAP (N => 8) PORT MAP (DataIn, Clock, Resetn, '1',
      DataIn_reg(7 DOWNTO 0));
   DataIn_reg(15 DOWNTO 8) <= "00000000";

   SRAM_DQ <= DataIn_reg WHEN (SRAM_WE_N = '0') ELSE "ZZZZZZZZZZZZZZZZ";
```

```
   -- hold CE_N to 1 at power-up, to avoid an accidental write
   R4: flip_flop PORT MAP ('1', Clock, Resetn, '1', CE);
   SRAM_CE_N <= NOT (CE);

   SRAM_OE_N <= '0';
   SRAM_UB_N <= '0';
   SRAM_LB_N <= '0';

   DataOut <= SRAM_DQ(7 DOWNTO 0);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
   HEX2 <= "1111111";
   HEX3 <= "1111111";
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Address(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Address(4)), HEX7);

   LEDG(0) <= Write;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '0');
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flip_flop IS
   PORT (   R                 : IN  STD_LOGIC;
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flip_flop;

ARCHITECTURE Behavior OF flip_flop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= '0';
```

```
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --       ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --       ---
   --     |   |
   --    4|   |2
   --     |   |
   --       ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
-- This code implements a simple dual-port memory using an M4K block
--
-- inputs: CLOCK_50 is the clock, KEY0 is Resetn, SW7-SW0 provides data to
-- write into memory.
-- SW15-SW11 provides the memory address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY part5 IS
PORT (   CLOCK_50                  : IN  STD_LOGIC;
         KEY                       : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                        : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                      : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT flip_flop
      PORT (   R                 : IN  STD_LOGIC;
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT ramlpm
      PORT (   clock       : IN STD_LOGIC ;
               data        : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               rdaddress   : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               wraddress   : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               wren        : IN STD_LOGIC  := '1';
               q           : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn, Write, Write_sync : STD_LOGIC;
   SIGNAL Write_address, Write_address_sync, Read_address :
      STD_LOGIC_VECTOR(4 DOWNTO 0);
   -- Read_address cycles from addresses 0 to 31 at one second per address

   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL DataIn, DataIn_sync, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= CLOCK_50;

   -- synchronize all asynchronous inputs to the clock
   R1: flip_flop PORT MAP (SW(17), Clock, Resetn, '1', Write_sync);
   R2: flip_flop PORT MAP (Write_sync, Clock, Resetn, '1', Write);

   R3: regne GENERIC MAP (N => 5)
      PORT MAP (SW(15 DOWNTO 11), Clock, Resetn, '1', Write_address_sync);
   R4: regne  GENERIC MAP (N => 5)
      PORT MAP (Write_address_sync, Clock, Resetn, '1', Write_address);
```

```
   R5: regne GENERIC MAP (N => 8) PORT MAP (SW(7 DOWNTO 0), Clock, Resetn,
      '1', DataIn_sync);
   R6: regne GENERIC MAP (N => 8) PORT MAP (DataIn_sync, Clock, Resetn,
      '1', DataIn);

   -- one second cycle counter
   -- Create a 1Hz 5-bit address counter
   -- A large counter to produce a 1 second (approx) enable
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- the read address counter
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Read_address <= (OTHERS => '0');
         ELSIF (slow_count = 0) THEN
            Read_address <= Read_address + '1';
         END IF;
      END IF;
   END PROCESS;

   -- instantiate LPM module
   -- module ramlpm (clock, data, rdaddress, wraddress, wren, q);
   m32x8_dual: ramlpm PORT MAP (Clock, DataIn, Read_address, Write_address,
      Write, DataOut);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
   digit2: hex7seg PORT MAP (Read_address(3 DOWNTO 0), HEX2);
   digit3: hex7seg PORT MAP (("000" & Read_address(4)), HEX3);
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Write_Address(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Write_Address(4)), HEX7);

   LEDG(0) <= Write;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '0');
         ELSIF (E = '1') THEN
```

```
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flip_flop IS
   PORT (   R                 : IN  STD_LOGIC;
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flip_flop;

ARCHITECTURE Behavior OF flip_flop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= '0';
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
```

```
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
DEPTH = 32;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 00000000;
01 : 00000001;
02 : 00000010;
03 : 00000011;
04 : 00000100;
05 : 00000101;
06 : 00000110;
07 : 00000111;
08 : 00001000;
09 : 00001001;
0A : 00001010;
0B : 00001011;
0C : 00001100;
0D : 00001101;
0E : 00001110;
0F : 00001111;
10 : 00010000;
11 : 00010001;
12 : 00010010;
13 : 00010011;
14 : 00010100;
15 : 00010101;
16 : 00010110;
17 : 00010111;
18 : 00011000;
19 : 00011001;
1A : 00011010;
1B : 00011011;
1C : 00011100;
1D : 00011101;
1E : 00011110;
1F : 00011111;

END;
```

```
-- This code implements a pseudo dual-port memory by using a multiplexer
-- for the write and read address, and using M4K for the memory
--
-- inputs: CLOCK_50 is the clock, KEY0 is Resetn, SW7-SW0 provides data to
-- write into memory.
-- SW15-SW11 provides the memory write address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY part6 IS
PORT (   CLOCK_50                  : IN  STD_LOGIC;
         KEY                       : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                        : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                      : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
END part6;

ARCHITECTURE Behavior OF part6 IS
   COMPONENT flip_flop
      PORT (   R                 : IN  STD_LOGIC;
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT ramlpm
      PORT (   address  : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Clock, Resetn, Write, Write_sync : STD_LOGIC;
   SIGNAL Write_address, Write_address_sync : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL Read_address, Address : STD_LOGIC_VECTOR(4 DOWNTO 0);
   -- Read_address cycles from addresses 0 to 31 at one second per address

   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL DataIn, DataIn_sync, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= CLOCK_50;

   -- synchronize all asynchronous inputs to the clock
   R1: flip_flop PORT MAP (SW(17), Clock, Resetn, '1', Write_sync);
   R2: flip_flop PORT MAP (Write_sync, Clock, Resetn, '1', Write);

   R3: regne GENERIC MAP (N => 5)
      PORT MAP (SW(15 DOWNTO 11), Clock, Resetn, '1', Write_address_sync);
   R4: regne  GENERIC MAP (N => 5)
      PORT MAP (Write_address_sync, Clock, Resetn, '1', Write_address);
```

```
   R5: regne GENERIC MAP (N => 8) PORT MAP (SW(7 DOWNTO 0), Clock, Resetn,
      '1', DataIn_sync);
   R6: regne GENERIC MAP (N => 8) PORT MAP (DataIn_sync, Clock, Resetn,
      '1', DataIn);

   -- one second cycle counter
   -- Create a 1Hz 5-bit address counter
   -- A large counter to produce a 1 second (approx) enable
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- the read address counter
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Read_address <= (OTHERS => '0');
         ELSIF (slow_count = 0) THEN
            Read_address <= Read_address + '1';
         END IF;
      END IF;
   END PROCESS;

   Address <= Write_address WHEN (Write = '1') ELSE Read_address;
   -- instantiate LPM module
   -- module ramlpm (address, clock, data, wren, q);
   m32x8: ramlpm PORT MAP (Address, Clock, DataIn, Write, DataOut);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
   digit2: hex7seg PORT MAP (Read_address(3 DOWNTO 0), HEX2);
   digit3: hex7seg PORT MAP (("000" & Read_address(4)), HEX3);
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Write_Address(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Write_Address(4)), HEX7);

   LEDG(0) <= Write;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '0');
         ELSIF (E = '1') THEN
```

```
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flip_flop IS
   PORT (   R                 : IN  STD_LOGIC;
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flip_flop;

ARCHITECTURE Behavior OF flip_flop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= '0';
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
```

```
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

```
-- This code implements a pseudo dual-port memory by using a multiplexer
-- for the write and read address, and using external SRAM.
--
-- inputs: CLOCK_50 is the clock, KEY0 is the reset, SW7-SW0 provides data to
-- write into memory,
-- SW15-SW11 provides the memory address, SW17 is the memory Write input.
-- outputs: 7-seg displays HEX7, HEX6 display the memory addres, HEX5, HEX4
-- displays the data input to the memory, and HEX1, HEX0 show the contents read
-- from the memory. LEDG0 shows the status of Write.
-- SRAM_ADDR provides the external SRAM chip address, SRAM_DQ is the data
-- input/output for the RAM, and the SRAM control signals are SRAM_WE_N,
-- SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, and SRAM_LB_N.
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY part7 IS
PORT (   CLOCK_50                   : IN  STD_LOGIC;
         KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
         SRAM_ADDR                  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
         SRAM_DQ                    : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         SRAM_WE_N                  : BUFFER STD_LOGIC;
         SRAM_CE_N, SRAM_OE_N       : OUT STD_LOGIC;
         SRAM_UB_N, SRAM_LB_N       : OUT STD_LOGIC);
END part7;

ARCHITECTURE Behavior OF part7 IS
   COMPONENT flip_flop
      PORT (   R                 : IN  STD_LOGIC;
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT regne
      GENERIC ( N : integer:= 7);
      PORT (   R                 : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
               Clock, Resetn, E  : STD_LOGIC;
               Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT hex7seg
      PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
               display  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Resetn, Clock, Write_n_sync, CE, CE_1, CE_2 : STD_LOGIC;
   SIGNAL Write_address, Write_address_sync : STD_LOGIC_VECTOR(4 DOWNTO 0);
   SIGNAL Read_address : STD_LOGIC_VECTOR(4 DOWNTO 0);
   -- Read_address cycles from addresses 0 to 31 at one second per address

   SIGNAL slow_count : STD_LOGIC_VECTOR(24 DOWNTO 0);
   SIGNAL DataIn, DataIn_sync, DataOut : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Clock <= CLOCK_50;

   -- synchronize all asynchronous inputs to the clock
   R1: flip_flop PORT MAP (NOT (SW(17)), Clock, Resetn, '1', Write_n_sync);
   R2: flip_flop PORT MAP (Write_n_sync, Clock, Resetn, '1', SRAM_WE_N);
   R3: regne GENERIC MAP (N => 5)
      PORT MAP (SW(15 DOWNTO 11), Clock, Resetn, '1', Write_address_sync);
   R4: regne GENERIC MAP (N => 5)
      PORT MAP (Write_address_sync, Clock, Resetn, '1', Write_address);
```

```
   R5: regne GENERIC MAP (N => 8) PORT MAP (SW(7 DOWNTO 0), Clock, Resetn,
      '1', DataIn_sync);
   R6: regne GENERIC MAP (N => 8) PORT MAP (DataIn_sync, Clock, Resetn,
      '1', DataIn);

   -- one second cycle counter
   -- Create a 1Hz 5-bit address counter
   -- A large counter to produce a 1 second (approx) enable
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         slow_count <= slow_count + '1';
      END IF;
   END PROCESS;

   -- the read address counter
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Read_address <= (OTHERS => '0');
         ELSIF (slow_count = 0) THEN
            Read_address <= Read_address + '1';
         END IF;
      END IF;
   END PROCESS;

   SRAM_ADDR <= ("0000000000000" & Write_address) WHEN (SRAM_WE_N = '0')
      ELSE ("0000000000000" & Read_address);

   SRAM_DQ <= ("00000000" & DataIn) WHEN (SRAM_WE_N = '0') ELSE "ZZZZZZZZZZZZZZZZ";

   -- hold CE_N to 1 for two clock cycles after power-up, to avoid an accidental write
   R7: flip_flop PORT MAP ('1', Clock, Resetn, '1', CE_1);
   R8: flip_flop PORT MAP (CE_1, Clock, Resetn, '1', CE_2);
   R9: flip_flop PORT MAP (CE_2, Clock, Resetn, '1', CE);
   SRAM_CE_N <= NOT (CE);

   SRAM_OE_N <= '0';
   SRAM_UB_N <= '0';
   SRAM_LB_N <= '0';

   DataOut <= SRAM_DQ(7 DOWNTO 0);

   -- display the data input, data output, and address on the 7-segs
   digit0: hex7seg PORT MAP (DataOut(3 DOWNTO 0), HEX0);
   digit1: hex7seg PORT MAP (DataOut(7 DOWNTO 4), HEX1);
      digit2: hex7seg PORT MAP (Read_address(3 DOWNTO 0), HEX2);
   digit3: hex7seg PORT MAP (("000" & Read_address(4)), HEX3);
   digit4: hex7seg PORT MAP (DataIn(3 DOWNTO 0), HEX4);
   digit5: hex7seg PORT MAP (DataIn(7 DOWNTO 4), HEX5);
   digit6: hex7seg PORT MAP (Write_Address(3 DOWNTO 0), HEX6);
   digit7: hex7seg PORT MAP (("000" & Write_Address(4)), HEX7);

   LEDG(0) <= SRAM_WE_N;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC ( N : integer:= 7);
   PORT (   R                : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Clock, Resetn, E  : IN STD_LOGIC;
```

```
            Q                 : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= (OTHERS => '0');
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flip_flop IS
   PORT (   R                 : IN  STD_LOGIC;
            Clock, Resetn, E  : IN STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flip_flop;

ARCHITECTURE Behavior OF flip_flop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF  (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN  -- synchronous clear
            Q <= '0';
         ELSIF (E = '1') THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
   PORT (   hex      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            display  : OUT STD_LOGIC_VECTOR(0 TO 6));
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
   --
   --       0
   --      ---
   --     |   |
   --    5|   |1
   --     | 6 |
   --      ---
   --     |   |
   --    4|   |2
   --     |   |
   --      ---
   --       3
   --
   PROCESS (hex)
```

```
   BEGIN
      CASE (hex) IS
         WHEN "0000" => display <= "0000001";
         WHEN "0001" => display <= "1001111";
         WHEN "0010" => display <= "0010010";
         WHEN "0011" => display <= "0000110";
         WHEN "0100" => display <= "1001100";
         WHEN "0101" => display <= "0100100";
         WHEN "0110" => display <= "1100000";
         WHEN "0111" => display <= "0001111";
         WHEN "1000" => display <= "0000000";
         WHEN "1001" => display <= "0001100";
         WHEN "1010" => display <= "0001000";
         WHEN "1011" => display <= "1100000";
         WHEN "1100" => display <= "0110001";
         WHEN "1101" => display <= "1000010";
         WHEN "1110" => display <= "0110000";
         WHEN OTHERS => display <= "0111000";
      END CASE;
   END PROCESS;
END Behavior;
```

Laboratory Exercise 9

A Simple Processor

Figure 1 shows a digital system that contains a number of 16-bit registers, a multiplexer, an adder/subtracter unit, a counter, and a control unit. Data is input to this system via the 16-bit *DIN* input. This data can be loaded through the 16-bit wide multiplexer into the various registers, such as $R0, \ldots, R7$ and *A*. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires are called a *bus* in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another.

Addition or subtraction is performed by using the multiplexer to first place one 16-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second 16-bit number is placed onto the bus, the adder/subtracter unit performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred to one of the other registers as required.

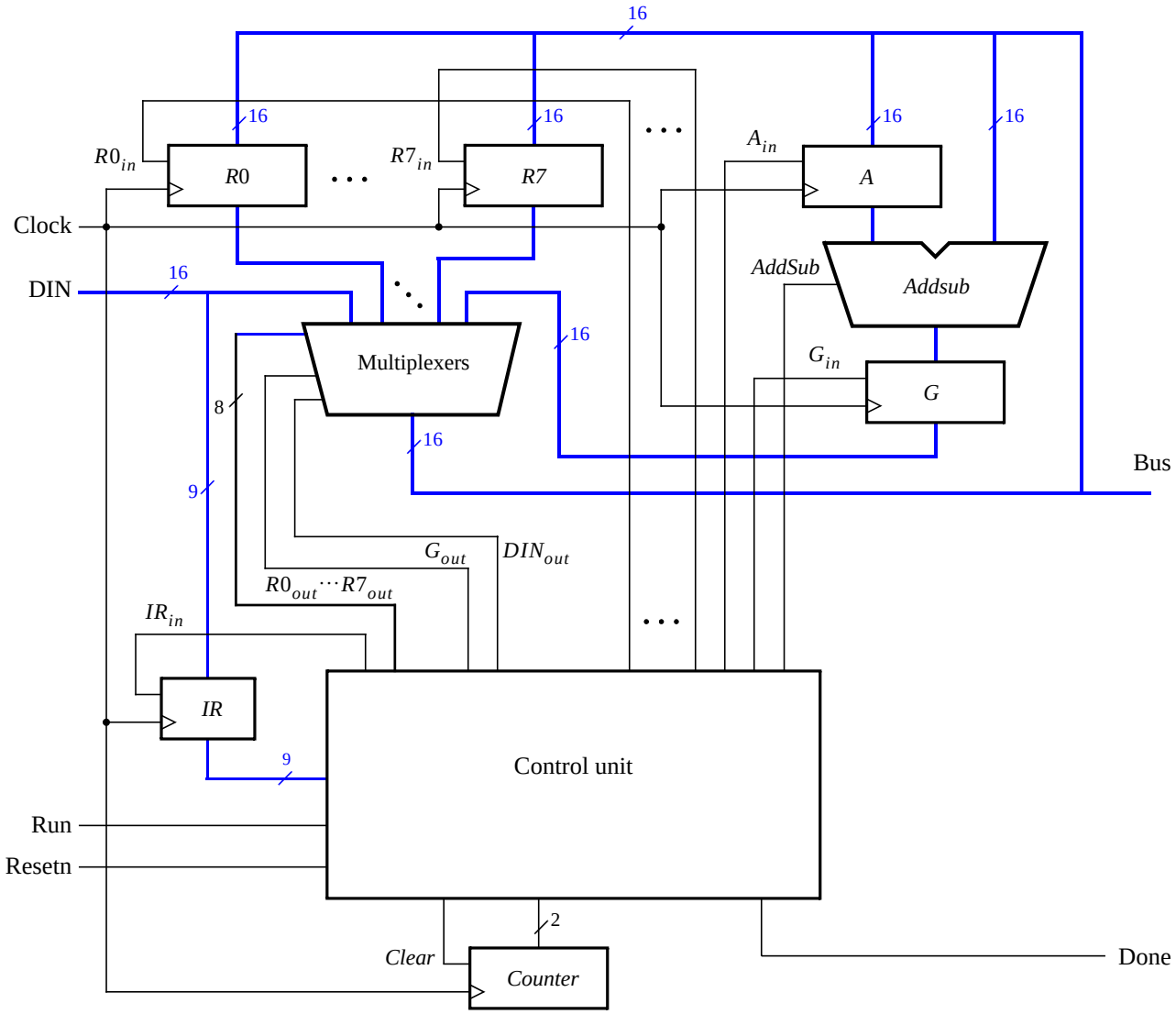


Figure 1. A digital system.

The system can perform different operations in each clock cycle, as governed by the *control unit*. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit asserts the signals $R0_{out}$ and A_{in}, then the multiplexer will place the contents of register $R0$ onto the bus and this data will be loaded by the next active clock edge into register *A*.

A system like this is often called a *processor*. It executes operations specified in the form of instructions. Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operand. The meaning of the syntax RX ← [RY] is that the contents of register RY are loaded into register RX. The **mv** (move) instruction allows data to be copied from one register to another. For the **mvi** (move immediate) instruction the expression RX ← D indicates that the 16-bit constant D is loaded into register RX.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 1. Instructions performed in the processor.

Each instruction can be encoded and stored in the *IR* register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the RX register, and YYY gives the RY register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Hence *IR* has to be connected to nine bits of the 16-bit *DIN* input, as indicated in Figure 1. For the **mvi** instruction the YYY field has no meaning, and the immediate data #D has to be supplied on the 16-bit *DIN* input after the **mvi** instruction word is stored into *IR*.

Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The control unit uses the two-bit counter shown in Figure 1 to enable it to "step through" such instructions. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is IR_{in}, so this time step is not shown in the table.

	T_1	T_2	T_3
(**mv**): I_0	RY_{out}, RX_{in}, *Done*		
(**mvi**): I_1	DIN_{out}, RX_{in}, *Done*		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(**sub**): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Table 2. Control signals asserted in each instruction/time step.

Part I

Design and implement the processor shown in Figure 1 using VHDL code as follows:

1. Create a new Quartus II project for this exercise.

2. Generate the required VHDL file, include it in your project, and compile the circuit. A suggested skeleton of the VHDL code is shown in parts a and b of Figure 2, and some subcircuit entities that can be used in this code appear in parts c and d.

3. Use functional simulation to verify that your code is correct. An example of the output produced by a functional simulation for a correctly-designed circuit is given in Figure 3. It shows the value $(2000)_{16}$ being loaded into *IR* from *DIN* at time 30 ns. This pattern represents the instruction **mvi** R0,#D, where the value $D = 5$ is loaded into $R0$ on the clock edge at 50 ns. The simulation then shows the instruction **mv** R1,R0 at 90 ns, **add** R0,R1 at 110 ns, and **sub** R0,R0 at 190 ns. Note that the simulation output shows *DIN* as a 4-digit hexadecimal number, and it shows the contents of *IR* as a 3-digit octal number.

4. Create a new Quartus II project which will be used for implementation of the circuit on the Altera DE2 board. This project should consist of a top-level entity that contains the appropriate input and output ports for the Altera board. Instantiate your processor in this top-level entity. Use switches SW_{15-0} to drive the *DIN* input port of the processor and use switch SW_{17} to drive the *Run* input. Also, use push button KEY_0 for *Resetn* and KEY_1 for *Clock*. Connect the processor bus wires to $LEDR_{15-0}$ and connect the *Done* signal to $LEDR_{17}$.

5. Add to your project the necessary pin assignments for the DE2 board. Compile the circuit and download it into the FPGA chip.

6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the processor's clock input is controlled by a push button switch, it is easy to step through the execution of instructions and observe the behavior of the circuit.

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY proc IS
   PORT ( DIN                 : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
          Resetn, Clock, Run  : IN     STD_LOGIC;
          Done                : BUFFER STD_LOGIC;
          BusWires            : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0));
END proc;

ARCHITECTURE Behavior OF proc IS
   . . . declare components
   . . . declare signals
BEGIN
   High <= '1';
   Clear <= . . .

   Tstep: upcount PORT MAP (Clear, Clock, Tstep_Q);
   I <= IR(1 TO 3);
   decX: dec3to8 PORT MAP (IR(4 TO 6), High, Xreg);
   decY: dec3to8 PORT MAP (IR(7 TO 9), High, Yreg);
```

Figure 2a. Skeleton VHDL code for the processor.

```
    controlsignals: PROCESS (Tstep_Q, I, Xreg, Yreg)
    BEGIN
        . . . specify initial values
        CASE Tstep_Q IS
            WHEN "00" => -- store DIN in IR as long as Tstep_Q = 0
                IRin <= '1';
            WHEN "01" => -- define signals in time step T1
                CASE I IS
                    . . .
                END CASE;
            WHEN "10" => -- define signals in time step T2
                CASE I IS
                    . . .
                END CASE;
            WHEN "11" => -- define signals in time step T3
                CASE I IS
                    . . .
                END CASE;
        END CASE;
    END PROCESS;

    reg_0: regn PORT MAP (BusWires, Rin(0), Clock, R0);
    . . . instantiate other registers and the adder/subtracter unit
    . . . define the bus
END Behavior;
```

Figure 2*b*. Skeleton VHDL code for the processor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY upcount IS
    PORT (  Clear, Clock    : IN    STD_LOGIC;
            Q               : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0));
END upcount;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
    PROCESS (Clock)
    BEGIN
        IF (Clock'EVENT AND Clock = '1') THEN
            IF Clear = '1' THEN
                Count <= "00";
            ELSE
                Count <= Count + 1;
            END IF;
        END IF;
    END PROCESS;
    Q <= Count;
END Behavior;
```

Figure 2*c*. Subcircuit entities for use in the processor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dec3to8 IS
   PORT ( W     : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
          En    : IN     STD_LOGIC;
          Y     : OUT    STD_LOGIC_VECTOR(0 TO 7));
END dec3to8;

ARCHITECTURE Behavior OF dec3to8 IS
BEGIN
   PROCESS (W, En)
   BEGIN
      IF En = '1' THEN
         CASE W IS
            WHEN "000" => Y <= "10000000";
            WHEN "001" => Y <= "01000000";
            WHEN "010" => Y <= "00100000";
            WHEN "011" => Y <= "00010000";
            WHEN "100" => Y <= "00001000";
            WHEN "101" => Y <= "00000100";
            WHEN "110" => Y <= "00000010";
            WHEN "111" => Y <= "00000001";
         END CASE;
      ELSE
         Y <= "00000000";
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC (n : INTEGER := 16);
   PORT ( R             : IN     STD_LOGIC_VECTOR(n-1 DOWNTO 0);
          Rin, Clock    : IN     STD_LOGIC;
          Q             : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         IF Rin = '1' THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

Figure 2*d*. Subcircuit entities for use in the processor.

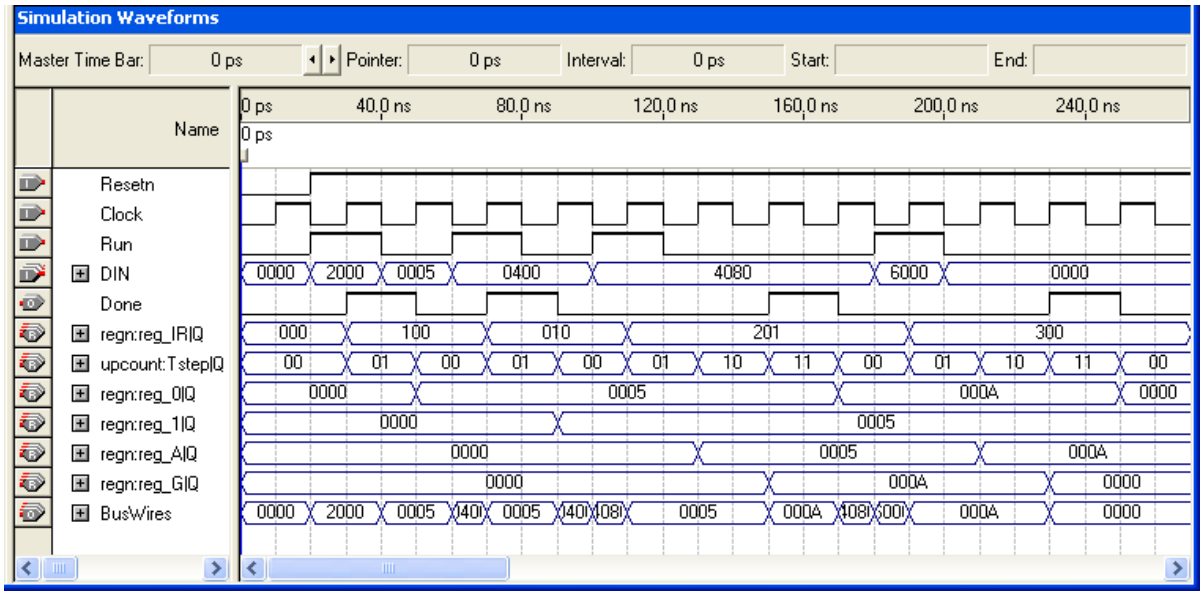


Figure 3. Simulation of the processor.

Part II

In this part you are to design the circuit depicted in Figure 4, in which a memory module and counter are connected to the processor from Part I. The counter is used to read the contents of successive addresses in the memory, and this data is provided to the processor as a stream of instructions. To simplify the design and testing of this circuit we have used separate clock signals, *PClock* and *MClock*, for the processor and memory.

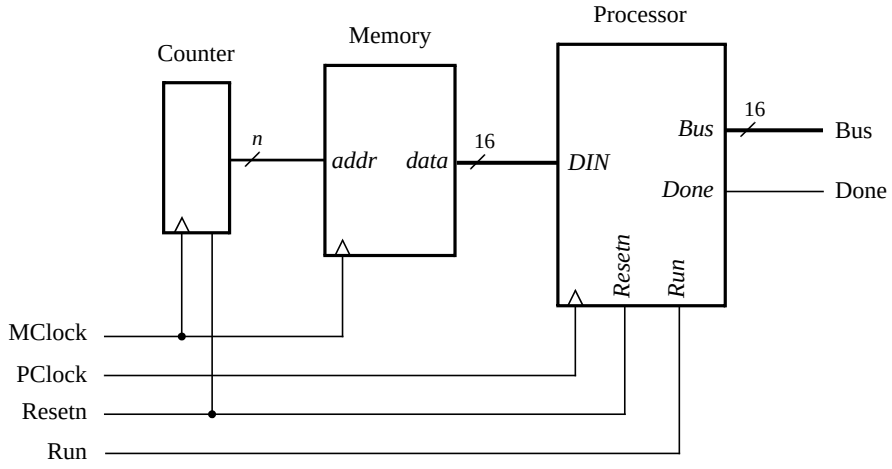


Figure 4. Connecting the processor to a memory and counter.

1. Create a new Quartus II project which will be used to test your circuit.

2. Generate a top-level VHDL file that instantiates the processor, memory, and counter. Use the Quartus II MegaWizard Plug-In Manager tool to create the memory module from the Altera library of parameterized modules (LPMs). The correct LPM is found under the *storage* category and is called *ALTSYNCRAM*. Follow the instructions provided by the wizard to create a memory that has one 16-bit wide read data port and is 32

words deep. The first screen of the wizard is shown in Figure 5. Since this memory has only a read port, and no write port, it is called a *synchronous read-only memory (synchronous ROM)*. Note that the memory includes a register for synchronously loading addresses. This register is required due to the design of the memory resources on the Cyclone II FPGA; account for the clocking of this address register in your design.

To place processor instructions into the memory, you need to specify *initial values* that should be stored in the memory once your circuit has been programmed into the FPGA chip. This can be done by telling the wizard to initialize the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen of the MegaWizard Plug-In Manager tool is illustrated in Figure 6. We have specified a file named *inst_mem.mif*, which then has to be created in the directory that contains the Quartus II project. Use the Quartus II on-line Help to learn about the format of the *MIF* file and create a file that has enough processor instructions to test your circuit.

3. Use functional simulation to test the circuit. Ensure that data is read properly out of the ROM and executed by the processor.

4. Make sure your project includes the necessary port names and pin location assignments to implement the circuit on the DE2 board. Use switch SW_{17} to drive the processor's *Run* input, use KEY_0 for *Resetn*, use KEY_1 for *MClock*, and use KEY_2 for *PClock*. Connect the processor bus wires to $LEDR_{15-0}$ and connect the *Done* signal to $LEDR_{17}$.

5. Compile the circuit and download it into the FPGA chip.

6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the circuit's clock inputs are controlled by push button switches, it is easy to step through the execution of instructions and observe the behavior of the circuit.

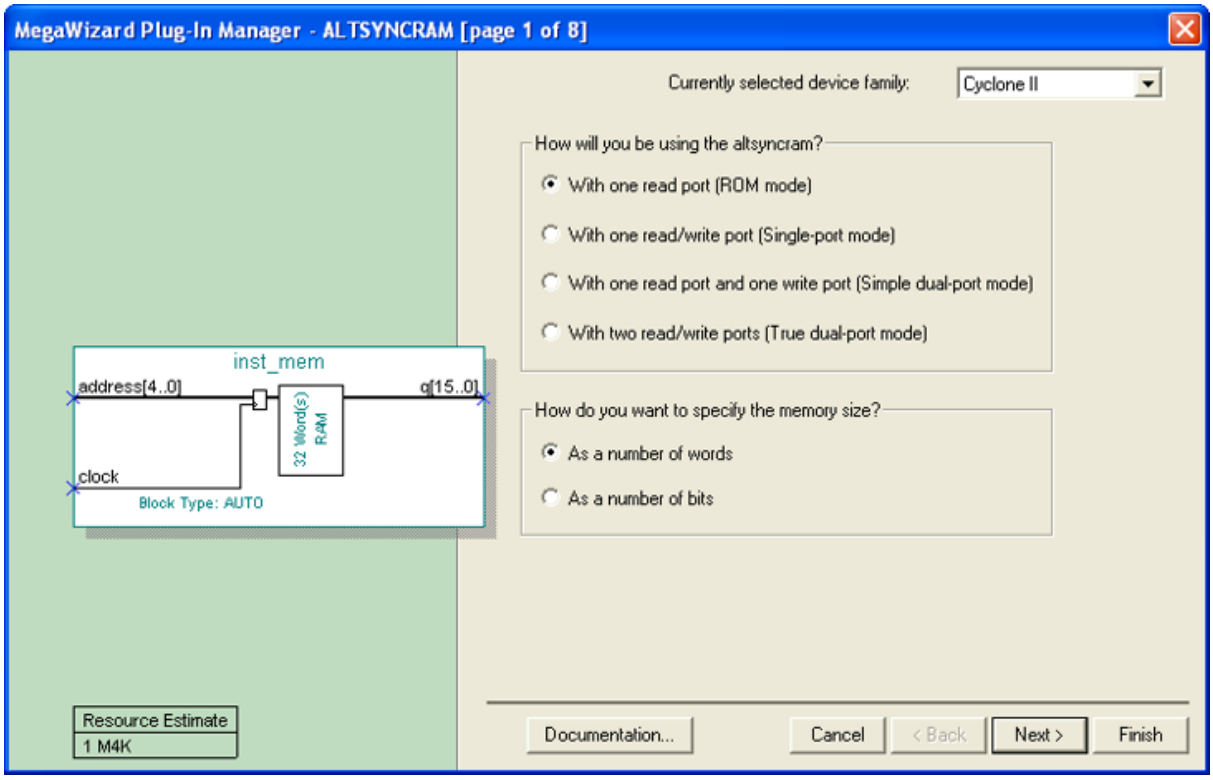


Figure 5. ALTSYNCRAM configuration.

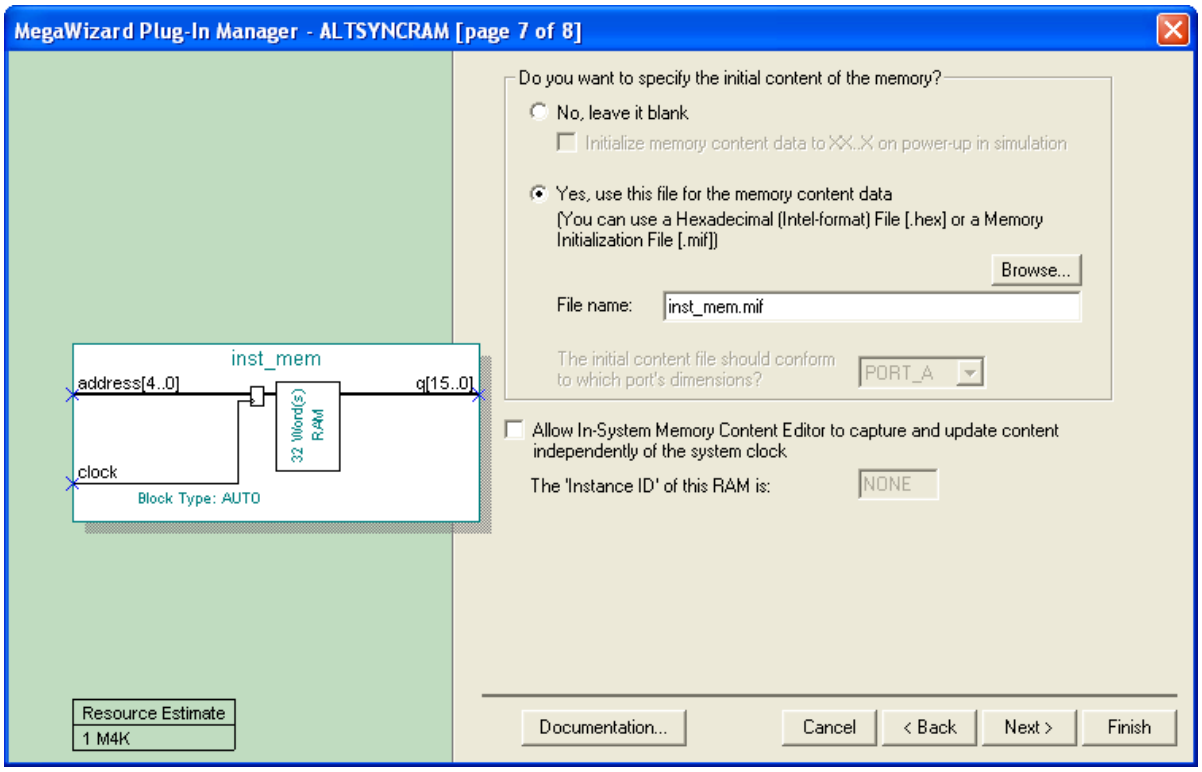


Figure 6. Specifying a memory initialization file (MIF).

Enhanced Processor

It is possible to enhance the capability of the processor so that the counter in Figure 4 is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. These enhancements involve adding new instructions to the processor and the programs that the processor executes are therefore more complex. Since these steps are beyond the scope of some logic design courses, they are described in a subsequent lab exercise available from Altera.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY proc IS
   PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            Resetn, Clock, Run   : IN  STD_LOGIC;
            Done                 : BUFFER STD_LOGIC;
            BusWires             : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0));
END proc;

ARCHITECTURE Behavior OF proc IS
   COMPONENT upcount
      PORT (   Clear, Clock   : IN     STD_LOGIC;
               Q              : BUFFER STD_LOGIC_VECTOR(1 DOWNTO 0));
   END COMPONENT;

   COMPONENT dec3to8
      PORT (   W     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               En    : IN  STD_LOGIC;
               Y     : OUT STD_LOGIC_VECTOR(0 TO 7));
   END COMPONENT;


   COMPONENT regn
      GENERIC (n : INTEGER := 16);
      PORT (   R           : IN     STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Rin, Clock  : IN     STD_LOGIC;
               Q           : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 7);
   SIGNAL Sum : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL Clear, High, IRin, DINout, Ain, Gin, Gout, AddSub : STD_LOGIC;
   SIGNAL Tstep_Q : STD_LOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL I : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL Xreg, Yreg : STD_LOGIC_VECTOR(0 TO 7);
   SIGNAL R0, R1, R2, R3, R4, R5, R6, R7, A, G : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL IR : STD_LOGIC_VECTOR(1 TO 9);
   SIGNAL Sel : STD_LOGIC_VECTOR(1 to 10); -- bus selector
BEGIN
   High <= '1';
   Clear <= NOT(Resetn) OR Done OR (NOT(Run) AND NOT(Tstep_Q(1)) AND NOT(Tstep_Q(0)));

   Tstep: upcount PORT MAP (Clear, Clock, Tstep_Q);
   I <= IR(1 TO 3);
   decX: dec3to8 PORT MAP (IR(4 TO 6), High, Xreg);
   decY: dec3to8 PORT MAP (IR(7 TO 9), High, Yreg);

   -- Instruction Table
   --    000: mv     Rx,Ry          : Rx <- [Ry]
   --    001: mvi    Rx,#D          : Rx <- D
   --    010: add    Rx,Ry          : Rx <- [Rx] + [Ry]
   --    011: sub    Rx,Ry          : Rx <- [Rx] - [Ry]
   --    OPCODE format: III XXX YYY, where
   --    III = instruction, XXX = Rx, and YYY = Ry. For mvi,
   --    a second word of data is loaded from DIN
   --
   controlsignals: PROCESS (Tstep_Q, I, Xreg, Yreg)
   BEGIN
      Done <= '0'; Ain <= '0'; Gin <= '0'; Gout <= '0'; AddSub <= '0';
      IRin <= '0'; DINout <= '0'; Rin <= "00000000"; Rout <= "00000000";
      CASE Tstep_Q IS
         WHEN "00" => -- store DIN in IR as long as Tstep_Q = 0
            IRin <= '1';
```

```
         WHEN "01" => -- define signals in time step T1
            CASE I IS
               WHEN "000" => -- mv Rx,Ry
                  Rout <= Yreg;
                  Rin <= Xreg;
                  Done <= '1';
               WHEN "001" => -- mvi Rx,#D
                  -- data is required to be on DIN
                  DINout <= '1';
                  Rin <= Xreg;
                  Done <= '1';
               WHEN "010" => -- add
                  Rout <= Xreg;
                  Ain <= '1';
               -- WHEN "011" => -- sub
               WHEN OTHERS => -- sub
                  Rout <= Xreg;
                  Ain <= '1';
               -- WHEN OTHERS => ;
            END CASE;
         WHEN "10" => -- define signals in time step T2
            CASE I IS
               WHEN "010" => -- add
                  Rout <= Yreg;
                  Gin <= '1';
               -- WHEN "011" => -- sub
               WHEN OTHERS => -- sub
                  Rout <= Yreg;
                  AddSub <= '1';
                  Gin <= '1';
               -- WHEN OTHERS => ;
            END CASE;
         WHEN "11" => -- define signals in time step T3
            CASE I IS
               WHEN "010" => -- add
                  Gout <= '1';
                  Rin <= Xreg;
                  Done <= '1';
               -- WHEN "011" => -- sub
               WHEN OTHERS => -- sub
                  Gout <= '1';
                  Rin <= Xreg;
                  Done <= '1';
               -- WHEN OTHERS => ;
            END CASE;
      END CASE;
   END PROCESS;

   reg_0: regn PORT MAP (BusWires, Rin(0), Clock, R0);
   reg_1: regn PORT MAP (BusWires, Rin(1), Clock, R1);
   reg_2: regn PORT MAP (BusWires, Rin(2), Clock, R2);
   reg_3: regn PORT MAP (BusWires, Rin(3), Clock, R3);
   reg_4: regn PORT MAP (BusWires, Rin(4), Clock, R4);
   reg_5: regn PORT MAP (BusWires, Rin(5), Clock, R5);
   reg_6: regn PORT MAP (BusWires, Rin(6), Clock, R6);
   reg_7: regn PORT MAP (BusWires, Rin(7), Clock, R7);
   reg_A: regn PORT MAP (BusWires, Ain, Clock, A);
   reg_IR: regn GENERIC MAP (n => 9) PORT MAP (DIN(15 DOWNTO 7), IRin, Clock, IR);

   -- alu
   alu: PROCESS (AddSub, A, BusWires)
   BEGIN
      IF AddSub = '0' THEN
         Sum <= A + BusWires;
```

```
      ELSE
         Sum <= A - BusWires;
      END IF;
   END PROCESS;

   reg_G: regn PORT MAP (Sum, Gin, Clock, G);

   -- define the internal processor bus
   Sel <= Rout & Gout & DINout;

   busmux: PROCESS (Sel, R0, R1, R2, R3, R4, R5, R6, R7, G, DIN)
   BEGIN
      IF Sel = "1000000000" THEN
         BusWires <= R0;
      ELSIF Sel = "0100000000" THEN
         BusWires <= R1;
      ELSIF Sel = "0010000000" THEN
         BusWires <= R2;
      ELSIF Sel = "0001000000" THEN
         BusWires <= R3;
      ELSIF Sel = "0000100000" THEN
         BusWires <= R4;
      ELSIF Sel = "0000010000" THEN
         BusWires <= R5;
      ELSIF Sel = "0000001000" THEN
         BusWires <= R6;
      ELSIF Sel = "0000000100" THEN
         BusWires <= R7;
      ELSIF Sel = "0000000010" THEN
         BusWires <= G;
      ELSE
         BusWires <= DIN;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY upcount IS
   PORT (   Clear, Clock   : IN     STD_LOGIC;
            Q              : OUT    STD_LOGIC_VECTOR(1 DOWNTO 0));
END upcount;

ARCHITECTURE Behavior OF upcount IS
   SIGNAL Count : STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF Clear  = '1' THEN
            Count <= "00";
         ELSE
            Count <= Count + 1;
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY dec3to8 IS
   PORT (   W     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            En    : IN  STD_LOGIC;
            Y     : OUT STD_LOGIC_VECTOR(0 TO 7));
END dec3to8;

ARCHITECTURE Behavior OF dec3to8 IS
BEGIN
   PROCESS (W, En)
   BEGIN
      IF En = '1' THEN
         CASE W IS
            WHEN "000" => Y <= "10000000";
            WHEN "001" => Y <= "01000000";
            WHEN "010" => Y <= "00100000";
            WHEN "011" => Y <= "00010000";
            WHEN "100" => Y <= "00001000";
            WHEN "101" => Y <= "00000100";
            WHEN "110" => Y <= "00000010";
            WHEN "111" => Y <= "00000001";
         END CASE;
      ELSE
         Y <= "00000000";
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC (n : INTEGER := 16);
   PORT (   R           : IN     STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            Rin, Clock  : IN     STD_LOGIC;
            Q           : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         IF Rin = '1' THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
-- KEY(0) is the reset input, and KEY(1) is the clock. SW15-0 are the instructions,
-- and SW(17) is the Run input. The processor bus appears on LEDR15-0 and
-- Done appears on LEDR17
-- This code instantiates a 32 x 8 memory n the Cyclone II FPGA on the DE2 board.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part1 IS
PORT (   KEY            : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
         SW             : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         LEDR           : OUT STD_LOGIC_VECTOR(17 DOWNTO 0));
END part1;

ARCHITECTURE Behavior OF part1 IS
   COMPONENT proc
      PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, Run   : IN  STD_LOGIC;
               Done                 : BUFFER STD_LOGIC;
               BusWires             : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0));
   END COMPONENT;
   SIGNAL Manual_Clock, Resetn, Run, Done : STD_LOGIC;
   SIGNAL DIN, BusWires : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   Manual_Clock <= KEY(1);
      -- Note: can't use name Clock because this is defined as
      -- the 50 MHz Clock coming into the FPGA from the board
   DIN <= SW(15 DOWNTO 0);
   Run <= SW(17);

   -- proc(DIN, Resetn, Clock, Run, Done, BusWires);
   U1: proc PORT MAP (DIN, Resetn, Manual_Clock, Run, Done, BusWires);

   LEDR(15 DOWNTO 0) <= BusWires;
   LEDR(16) <= '0';
   LEDR(17) <= Done;
END Behavior;
```

```
-- Reset with KEY(0). Clock counter and memory with KEY(2). Clock
-- each instuction into the processor with KEY(1). SW(17) is the Run input.
-- Use KEY(2) to advance the memory as needed before each processor KEY(1)
-- clock cycle.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part2 IS
PORT (   KEY            : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
         SW             : IN  STD_LOGIC_VECTOR(17 DOWNTO 17);
         LEDR           : OUT STD_LOGIC_VECTOR(17 DOWNTO 0));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT proc
      PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, Run   : IN  STD_LOGIC;
               Done                 : BUFFER STD_LOGIC;
               BusWires             : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0));
   END COMPONENT;
   COMPONENT inst_mem
      PORT (   address  : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               q        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT count5
      PORT (   Resetn, Clock  : IN  STD_LOGIC;
               Q              : OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
   END COMPONENT;

   SIGNAL Resetn, PClock, MClock, Run, Done : STD_LOGIC;
   SIGNAL DIN, BusWires : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL pc : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
   Resetn <= KEY(0);
   PClock <= KEY(1);
   MClock <= KEY(2);
   Run <= SW(17);
   -- proc(DIN, Resetn, Clock, Run, Done, BusWires);
   U1: proc PORT MAP (DIN, Resetn, PClock, Run, Done, BusWires);
   LEDR(15 DOWNTO 0) <= BusWires;
   LEDR(17) <= Done;

   U2: inst_mem PORT MAP (pc, MClock, DIN);
   U3: count5 PORT MAP (Resetn, MClock, pc);
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY count5 IS
PORT (   Resetn, Clock  : IN  STD_LOGIC;
         Q              : OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
END count5;

ARCHITECTURE Behavior OF count5 IS
   SIGNAL Count : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= "00000";
```

```
         ELSE
            Count <= Count + '1';
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;
```

```
DEPTH = 32;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
   00 :  0010000000000000;
   01 :  0000000000000101;
   02 :  0000010000000000;
   03 :  0100000010000000;
   04 :  0110000000000000;
   05 :  0000000000000000;
   06 :  0000000000000000;
   07 :  0000000000000000;
   08 :  0000000000000000;
   09 :  0000000000000000;
   0A :  0000000000000000;
   0B :  0000000000000000;
   0C :  0000000000000000;
   0D :  0000000000000000;
   0E :  0000000000000000;
   0F :  0000000000000000;
   10 :  0000000000000000;
   11 :  0000000000000000;
   12 :  0000000000000000;
   13 :  0000000000000000;
   14 :  0000000000000000;
   15 :  0000000000000000;
   16 :  0000000000000000;
   17 :  0000000000000000;
   18 :  0000000000000000;
   19 :  0000000000000000;
   1A :  0000000000000000;
   1B :  0000000000000000;
   1C :  0000000000000000;
   1D :  0000000000000000;
   1E :  0000000000000000;
   1F :  0000000000000000;
END;
```

Laboratory Exercise 10

An Enhanced Processor

In Laboratory Exercise 9 we described a simple processor. In Part I of that exercise the processor itself was designed, and in Part II the processor was connected to an external counter and a memory unit. This exercise describes subsequent parts of the processor design. Note that the numbering of figures and tables in this exercise are continued from those in Parts I and II in the preceding lab exercise.

Part III

In this part you will extend the capability of the processor so that the external counter is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. You will add three new types of instructions to the processor, as displayed in Table 3. The **ld** (load) instruction loads data into register RX from the external memory address specified in register RY. The **st** (store) instruction stores the data contained in register RX into the memory address found in RY. Finally, the instruction **mvnz** (move if not zero) allows a **mv** operation to be executed only under a certain condition; the condition is that the current contents of register G are not equal to 0.

Operation	Function performed
ld Rx,$[Ry]$	$Rx \leftarrow [[Ry]]$
st Rx,$[Ry]$	$[Ry] \leftarrow [Rx]$
mvnz Rx, Ry	if G != 0, $Rx \leftarrow [Ry]$

Table 3. New instructions performed in the processor.

A schematic of the enhanced processor is given in Figure 7. In this figure, registers $R0$ to $R6$ are the same as in Figure 1 of Laboratory Exercise 9, but register $R7$ has been changed to a counter. This counter is used to provide the addresses in the memory from which the processor's instructions are read; in the preceding lab exercise, a counter external to the processor was used for this purpose. We will refer to $R7$ as the processor's *program counter* (*PC*), because this terminology is common for real processors available in the industry. When the processor is reset, *PC* is set to address 0. At the start of each instruction (in time step 0) the contents of *PC* are used as an address to read an instruction from the memory. The instruction is stored in IR and the *PC* is automatically incremented to point to the next instruction (in the case of **mvi** the *PC* provides the address of the immediate data and is then incremented again).

The processor's control unit increments *PC* by using the *incr_PC* signal, which is just an enable on this counter. It is also possible to directly load an address into *PC* ($R7$) by having the processor execute a **mv** or **mvi** instruction in which the destination register is specified as $R7$. In this case the control unit uses the signal $R7_{in}$ to perform a parallel load of the counter. In this way, the processor can execute instructions at any address in memory, as opposed to only being able to execute instructions that are stored in successive addresses. Similarly, the current contents of *PC* can be copied into another register by using a **mv** instruction. An example of code that uses the *PC* register to implement a loop is shown below, where the text after the % on each line is just a comment. The instruction **mv** R5,R7 places into *R5* the address in memory of the instruction **sub** R4,R2. Then, the instruction **mvnz** R7,R5 causes the **sub** instruction to be executed repeatedly until R4 becomes 0. This type of loop could be used in a larger program as a way of creating a delay.

```
mvi   R2,#1
mvi   R4,#10000000   % binary delay value
mv    R5,R7          % save address of next instruction
sub   R4,R2          % decrement delay count
mvnz  R7,R5          % continue subtracting until delay count gets to 0
```

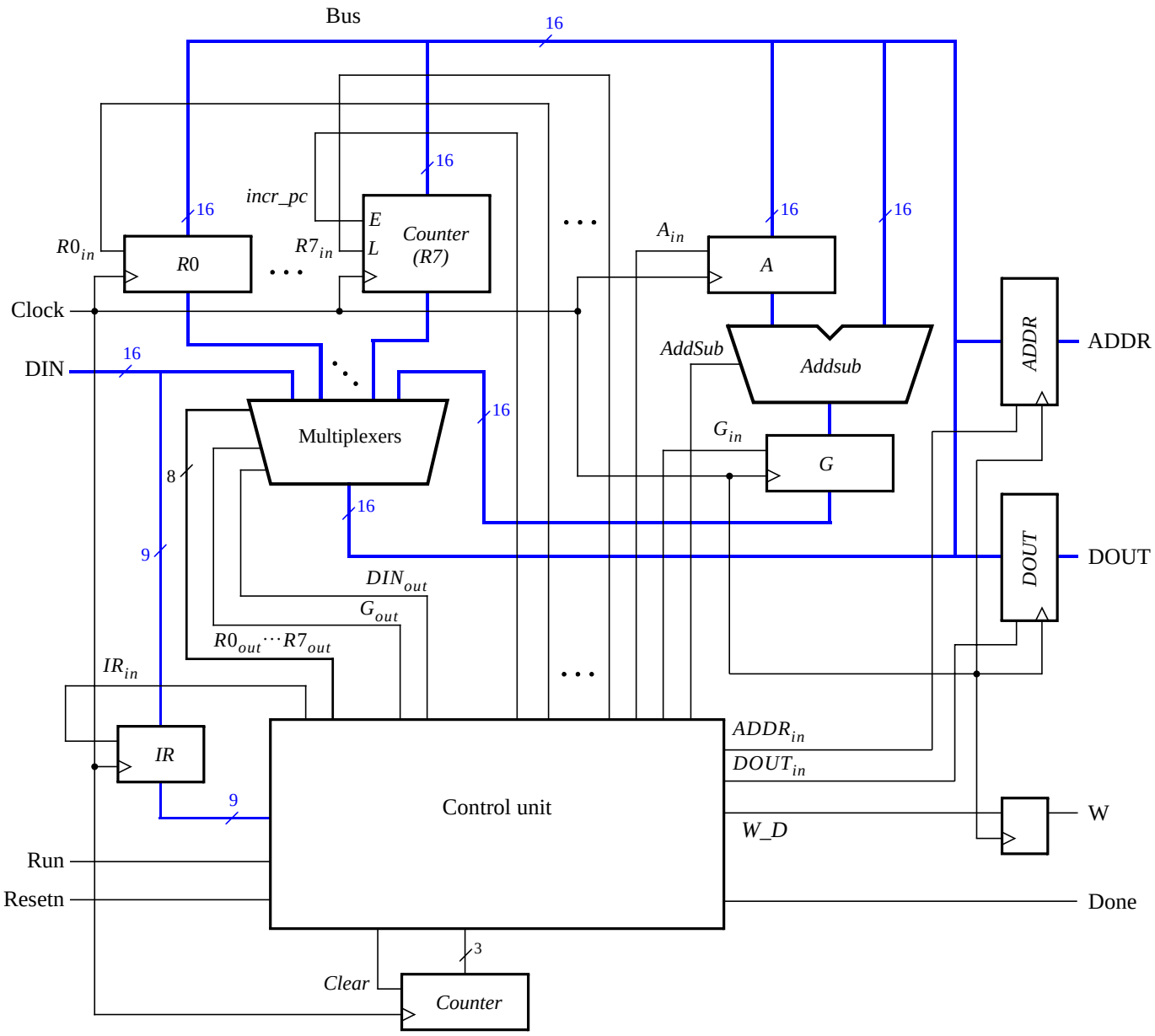


Figure 7. An enhanced version of the processor.

Figure 7 shows two registers in the processor that are used for data transfers. The *ADDR* register is used to send addresses to an external device, such as a memory module, and the *DOUT* register is used by the processor to provide data that can be stored outside the processor. One use of the *ADDR* register is for reading, or *fetching*, instructions from memory; when the processor wants to fetch an instruction, the contents of *PC* ($R7$) are transferred across the bus and loaded into *ADDR*. This address is provided to memory. In addition to fetching instructions, the processor can read data at any address by using the *ADDR* register. Both data and instructions are read into the processor on the *DIN* input port. The processor can write data for storage at an external address by placing this address into the *ADDR* register, placing the data to be stored into its *DOUT* register, and asserting the output of the *W* (write) flip-flop to 1.

Figure 8 illustrates how the enhanced processor is connected to memory and other devices. The memory unit in the figure supports both read and write operations and therefore has both address and data inputs, as well as a write enable input. The memory also has a clock input, because the address, data, and write enable inputs must be

loaded into the memory on an active clock edge. This type of memory unit is usually called a *synchronous random access memory (synchronous RAM)*. Figure 8 also includes a 16-bit register that can be used to store data from the processor; this register might be connected to a set of LEDs to allow display of data on the DE2 board. To allow the processor to select either the memory unit or register when performing a write operation, the circuit includes some logic gates that perform *address decoding*: if the upper address lines are $A_{15}A_{14}A_{13}A_{12} = 0000$, then the memory module will be written at the address given on the lower address lines. Figure 8 shows n lower address lines connected to the memory; for this exercise a memory with 128 words is probably sufficient, which implies that $n = 7$ and the memory address port is driven by $A_6 \ldots A_0$. For addresses in which $A_{15}A_{14}A_{13}A_{12} = 0001$, the data written by the processor is loaded into the register whose outputs are called *LEDs* in Figure 8.

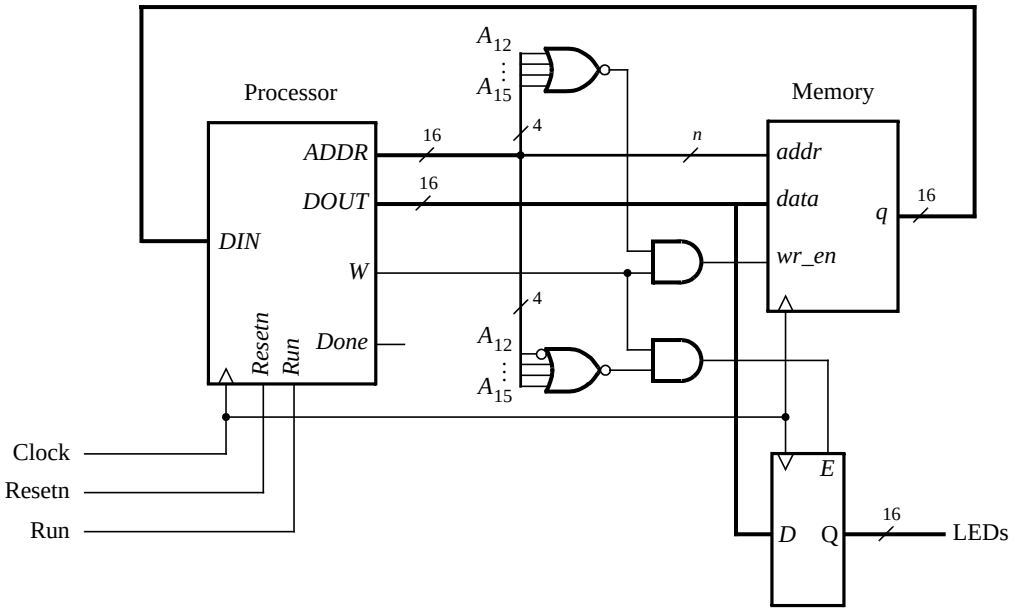


Figure 8. Connecting the enhanced processor to a memory and output register.

1. Create a new Quartus II project for the enhanced version of the processor.

2. Write VHDL code for the processor and test your circuit by using functional simulation: apply instructions to the *DIN* port and observe the internal processor signals as the instructions are executed. Pay careful attention to the timing of signals between your processor and external memory; account for the fact that the memory has registered input ports, as we discussed for Figure 8.

3. Create another Quartus II project that instantiates the processor, memory module, and register shown in Figure 8. Use the Quartus II MegaWizard Plug-In Manager tool to create the ALTSYNCRAM memory module. Follow the instructions provided by the wizard to create a memory that has one 16-bit wide read/write data port and is 128 words deep. Use a MIF file to store instructions in the memory that are to be executed by your processor.

4. Use functional simulation to test the circuit. Ensure that data is read properly from the RAM and executed by the processor.

5. Include in your project the necessary pin assignments to implement your circuit on the DE2 board. Use switch SW_{17} to drive the processor's *Run* input, use KEY_0 for *Resetn*, and use the board's 50 MHz clock signal as the *Clock* input. Since the circuit needs to run properly at 50 MHz, make sure that a timing constraint is set in Quartus II to constrain the circuit's clock to this frequency. Read the Report produced by the Quartus II Timing Analyzer to ensure that your circuit operates at this speed; if not, use the Quartus II tools to analyze your circuit and modify your VHDL code to make a more efficient design that meets the

50-MHz speed requirement. Also note that the *Run* input is asynchronous to the clock signal, so make sure to synchronize this input using flip-flops.

Connect the *LEDs* register in Figure 8 to $LEDR_{15-0}$ so that you can observe the output produced by the processor.

6. Compile the circuit and download it into the FPGA chip.
7. Test the functionality of your design by executing code from the RAM and observing the LEDs.

Part IV

In this part you are to connect an additional I/O module to your circuit from Part III and write code that is executed by your processor.

Add a module called *seg7_scroll* to your circuit. This module should contain one register for each 7-segment display on the DE2 board. Each register should directly drive the segment lights for one 7-segment display, so that the processor can write characters onto these displays. Create the necessary address decoding to allow the processor to write to the registers in the *seg7_scroll* module.

1. Create a Quartus II project for your circuit and write the VHDL code that includes the circuit from Figure 8 in addition to your *seg7_scroll* module.
2. Use functional simulation to test the circuit.
3. Add appropriate timing constraints and pin assignments to your project, and write a MIF file that allows the processor to write characters to the 7-segment displays. A simple program would write a word to the displays and then terminate, but a more interesting program could scroll a message across the displays, or scroll a word across the displays in the left, right, or both directions.
4. Test the functionality of your design by executing code from the RAM and observing the 7-segment displays.

Part V

Add to your circuit from Part IV another module, called *port_n*, that allows the processor to read the state of some switches on the board. The switch values should be stored into a register, and the processor should be able to read this register by using a **ld** instruction. You will have to use address decoding and multiplexers to allow the processor to read from either the RAM or *port_n* units, according to the address used.

1. Draw a circuit diagram that shows how the *port_n* unit is incorporated into the system.
2. Create a Quartus II project for your circuit, write the VHDL code, and write a MIF file that demonstrates use of the *port_n* module. One interesting application is to have the processor scroll a message across the 7-segment displays and use the values read from the *port_n* module to change the speed at which the message is scrolled.
3. Test your circuit both by using functional simulation and by downloading it and executing your processor code on the DE2 board.

Suggested Bonus Parts

The following are suggested bonus parts for this exercise.

1. Use the Quartus II tools to identify the critical paths in the processor circuit. Modify the processor design so that the circuit will operate at the highest clock frequency that you can achieve.
2. Extend the instructions supported by your processor to make it more flexible. Some suggested instruction types are logic instructions (AND, OR, etc), shift instructions, and branch instructions. You may also wish to add support for logical conditions other than “not zero” , as supported by **mvnz**, and the like.
3. Write an Assembler program for your processor. It should automatically produces a MIF file from assembler code.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY proc IS
   PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            Resetn, Clock, Run   : IN  STD_LOGIC;
            DOUT                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            ADDR                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            W                    : OUT STD_LOGIC;
            Done                 : BUFFER STD_LOGIC);
END proc;

ARCHITECTURE Behavior OF proc IS
   COMPONENT upcount
      PORT (   Clear, Clock   : IN  STD_LOGIC;
               Q              : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
   END COMPONENT;
   COMPONENT pc_count
      PORT (   R                    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, E, L  : IN  STD_LOGIC;
               Q                    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
   END COMPONENT;

   COMPONENT dec3to8
      PORT (   W     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               En    : IN  STD_LOGIC;
               Y     : OUT STD_LOGIC_VECTOR(0 TO 7));
   END COMPONENT;

   COMPONENT regn
      GENERIC (n : INTEGER := 16);
      PORT (   R           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Rin, Clock  : IN  STD_LOGIC;
               Q           : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Resetn, Clock  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;

   SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 7);
   SIGNAL BusWires, Sum : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL Clear, IRin, ADDRin, DINout, DOUTin, Ain, Gin, Gout, AddSub : STD_LOGIC;
   SIGNAL Tstep_Q : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL I : STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL Xreg, Yreg : STD_LOGIC_VECTOR(0 TO 7);
   SIGNAL R0, R1, R2, R3, R4, R5, R6, R7, A, G : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL IR : STD_LOGIC_VECTOR(1 TO 9);
   SIGNAL Sel : STD_LOGIC_VECTOR(1 to 10); -- bus selector
   SIGNAL pc_inc, W_D, Z, Z_D : STD_LOGIC;
BEGIN
   Clear <= NOT(Resetn) OR Done OR (NOT(Run) AND NOT(Tstep_Q(1)) AND NOT(Tstep_Q(0)));

   Tstep: upcount PORT MAP (Clear, Clock, Tstep_Q);
   I <= IR(1 TO 3);
   decX: dec3to8 PORT MAP (IR(4 TO 6), '1', Xreg);
   decY: dec3to8 PORT MAP (IR(7 TO 9), '1', Yreg);

   -- Instruction Table
   --    000: mv     Rx,Ry          : Rx <- [Ry]
   --    001: mvi    Rx,#D          : Rx <- D
   --    010: add    Rx,Ry          : Rx <- [Rx] + [Ry]
   --    011: sub    Rx,Ry          : Rx <- [Rx] - [Ry]
```

```
--    100: ld     Rx,[Ry]        : Rx <- [[Ry]]
--    101: st     Rx,[Ry]        : [Ry] <- [Rx]
--    110: mvnz   Rx,Ry          : if Z != 1, Rx <- [Ry]
--    OPCODE format: III XXX YYY UUUUUUU, where
--    III = instruction, XXX = Rx, YYY = Ry, and U = unused bit. For mvi,
--    a second word of data is read in the following clock cycle
--
-- R7 is the program counter
controlsignals: PROCESS (Tstep_Q, I, Xreg, Yreg, Z, Run)
BEGIN
   Done <= '0'; Ain <= '0'; Gin <= '0'; Gout <= '0'; AddSub <= '0';
   IRin <= '0'; DINout <= '0'; DOUTin <= '0'; ADDRin <= '0'; W_D <= '0';
   Rin <= "00000000"; Rout <= "00000000"; pc_inc <= '0';
   CASE Tstep_Q IS
      WHEN "000" => -- fetch the instruction
         Rout <= "00000001";  -- R7 is program counter (pc)
         ADDRin <= '1';
         pc_inc <= Run; -- to increment pc
      WHEN "001" => -- wait cycle for synchronous memory
         -- in case the instruction turns out to be mvi, read memory
         Rout <= "00000001";  -- R7 is program counter (pc)
         ADDRin <= '1';
      WHEN "010" => -- store DIN in IR
         IRin <= '1';
      WHEN "011" =>  -- define signals in T3
         CASE I IS
            WHEN "000" => -- mv Rx,Ry
               Rout <= Yreg;
               Rin <= Xreg;
               Done <= '1';
            WHEN "001" => -- mvi Rx,#D
               -- data is available now on DIN
               DINout <= '1';
               Rin <= Xreg;
               pc_inc <= '1';
               Done <= '1';
            WHEN "010" => -- add
               Rout <= Xreg;
               Ain <= '1';
            WHEN "011" => -- sub
               Rout <= Xreg;
               Ain <= '1';
            WHEN "100" => -- ld Rx,[Ry]
               Rout <= Yreg;
               ADDRin <= '1';
            WHEN "101" => -- st [Ry],Rx
               Rout <= Yreg;
               ADDRin <= '1';
            WHEN OTHERS => -- mvnz Rx,Ry
               IF Z = '0' THEN
                  Rout <= Yreg;
                  Rin <= Xreg;
               ELSE
                  Rout <= "00000000";
                  Rin <= "00000000";
               END IF;
               Done <= '1';
         END CASE;
      WHEN "100" => -- define signals T4
         CASE I IS
            WHEN "010" => -- add
               Rout <= Yreg;
               Gin <= '1';
            WHEN "011" => -- sub
```

```
                Rout <= Yreg;
                AddSub <= '1';
                Gin <= '1';
             WHEN "100" => -- ld Rx,[Ry]
                W_D <= '0'; -- do nothing--wait cycle for synchronous memory
             WHEN OTHERS => -- st [Ry],Rx
                Rout <= Xreg;
                DOUTin <= '1';
                W_D <= '1';
          END CASE;
       WHEN OTHERS =>  -- define T5
          CASE I IS
             WHEN "010" => -- add
                Gout <= '1';
                Rin <= Xreg;
                Done <= '1';
             WHEN "011" => -- sub
                Gout <= '1';
                Rin <= Xreg;
                Done <= '1';
             WHEN "100" => -- ld Rx,[Ry]
                DINout <= '1';
                Rin <= Xreg;
                Done <= '1';
             WHEN OTHERS => -- st [Ry],Rx
                Done <= '1'; -- wait cycle for synhronous memory
          END CASE;
   END CASE;
END PROCESS;

reg_0: regn PORT MAP (BusWires, Rin(0), Clock, R0);
reg_1: regn PORT MAP (BusWires, Rin(1), Clock, R1);
reg_2: regn PORT MAP (BusWires, Rin(2), Clock, R2);
reg_3: regn PORT MAP (BusWires, Rin(3), Clock, R3);
reg_4: regn PORT MAP (BusWires, Rin(4), Clock, R4);
reg_5: regn PORT MAP (BusWires, Rin(5), Clock, R5);
reg_6: regn PORT MAP (BusWires, Rin(6), Clock, R6);

-- R7 is program counter
-- pc_count(R, Resetn, Clock, E, L, Q);
pc: pc_count PORT MAP (BusWires, Resetn, Clock, pc_inc, Rin(7), R7);

reg_A: regn PORT MAP (BusWires, Ain, Clock, A);
reg_DOUT: regn PORT MAP (BusWires, DOUTin, Clock, DOUT);
reg_ADDR: regn PORT MAP (BusWires, ADDRin, Clock, ADDR);
reg_IR: regn GENERIC MAP (n => 9) PORT MAP (DIN(15 DOWNTO 7), IRin, Clock, IR);

reg_W: flipflop PORT MAP (W_D, Resetn, Clock, W);

-- alu
alu: PROCESS (AddSub, A, BusWires)
BEGIN
   IF AddSub = '0' THEN
      Sum <= A + BusWires;
   ELSE
      Sum <= A - BusWires;
   END IF;
END PROCESS;

reg_G: regn PORT MAP (Sum, Gin, Clock, G);

Z_D <= '1' WHEN (G = 0) ELSE '0';
reg_Z: flipflop PORT MAP (Z_D, Resetn, Clock, Z);
```

```
   -- define the internal processor bus
   Sel <= Rout & Gout & DINout;

   busmux: PROCESS (Sel, R0, R1, R2, R3, R4, R5, R6, R7, G, DIN)
   BEGIN
      IF Sel = "1000000000" THEN
         BusWires <= R0;
      ELSIF Sel = "0100000000" THEN
         BusWires <= R1;
      ELSIF Sel = "0010000000" THEN
         BusWires <= R2;
      ELSIF Sel = "0001000000" THEN
         BusWires <= R3;
      ELSIF Sel = "0000100000" THEN
         BusWires <= R4;
      ELSIF Sel = "0000010000" THEN
         BusWires <= R5;
      ELSIF Sel = "0000001000" THEN
         BusWires <= R6;
      ELSIF Sel = "0000000100" THEN
         BusWires <= R7;
      ELSIF Sel = "0000000010" THEN
         BusWires <= G;
      ELSE
         BusWires <= DIN;
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY upcount IS
   PORT (   Clear, Clock   : IN     STD_LOGIC;
            Q              : OUT    STD_LOGIC_VECTOR(2 DOWNTO 0));
END upcount;

ARCHITECTURE Behavior OF upcount IS
   SIGNAL Count : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF Clear  = '1' THEN
            Count <= "000";
         ELSE
            Count <= Count + 1;
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY pc_count IS
   PORT (   R                    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            Resetn, Clock, E, L  : IN  STD_LOGIC;
            Q                    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END pc_count;
```

```
ARCHITECTURE Behavior OF pc_count IS
   SIGNAL Count : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Count <= (OTHERS => '0');
         ELSIF (L = '1') THEN
            Count <= R;
         ELSIF (E = '1') THEN
            Count <= Count + 1;
         END IF;
      END IF;
   END PROCESS;
   Q <= Count;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dec3to8 IS
   PORT (   W     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            En    : IN  STD_LOGIC;
            Y     : OUT STD_LOGIC_VECTOR(0 TO 7));
END dec3to8;

ARCHITECTURE Behavior OF dec3to8 IS
BEGIN
   PROCESS (W, En)
   BEGIN
      IF En = '1' THEN
         CASE W IS
            WHEN "000" => Y <= "10000000";
            WHEN "001" => Y <= "01000000";
            WHEN "010" => Y <= "00100000";
            WHEN "011" => Y <= "00010000";
            WHEN "100" => Y <= "00001000";
            WHEN "101" => Y <= "00000100";
            WHEN "110" => Y <= "00000010";
            WHEN "111" => Y <= "00000001";
         END CASE;
      ELSE
         Y <= "00000000";
      END IF;
   END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regn IS
   GENERIC (n : INTEGER := 16);
   PORT (   R           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            Rin, Clock  : IN  STD_LOGIC;
            Q           : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         IF Rin = '1' THEN
```

```
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
-- Reset with KEY[0]. SW[17] is Run.
-- The DOUT, ADDR, W, and Done outputs are just for simulation; they
-- aren't connected to any DE2 resources. The HEX0-HEX7 and LEDG are driven
-- to constant values because letting them float causes some of the LEDs
-- to flash on and off for this circuit.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part3 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 17);
         Clock                      : IN  STD_LOGIC;
         LEDR                       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         DOUT, ADDR                 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
         W, Done                    : BUFFER STD_LOGIC;
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDG                       : OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END part3;

ARCHITECTURE Behavior OF part3 IS
   COMPONENT proc
      PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, Run   : IN  STD_LOGIC;
               DOUT                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               ADDR                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               W                    : OUT STD_LOGIC;
               Done                 : BUFFER STD_LOGIC);
   END COMPONENT;
   COMPONENT inst_mem
      PORT (   address  : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT regn
      GENERIC (n : INTEGER := 16);
      PORT (   R           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Rin, Clock  : IN  STD_LOGIC;
               Q           : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Resetn, Clock  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL Sync, Run, inst_mem_cs, LED_reg_cs : STD_LOGIC;
   SIGNAL DIN, LED_reg : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   HEX7 <= "0000000";
   HEX6 <= "0000000";
   HEX5 <= "0000000";
   HEX4 <= "0000000";
   HEX3 <= "0000000";
   HEX2 <= "0000000";
   HEX1 <= "0000000";
   HEX0 <= "0000000";
   LEDG <= "000000000";

   -- synchronize the Run input
   U1: flipflop PORT MAP (SW(17), KEY(0), Clock, Sync);
   U2: flipflop PORT MAP (Sync, KEY(0), Clock, Run);

   -- proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
```

```
    U3: proc PORT MAP (DIN, KEY(0), Clock, Run, DOUT, ADDR, W, Done);

    inst_mem_cs <= '1' WHEN (ADDR(15 DOWNTO 12) = "0000") ELSE '0';
    -- inst_mem ( address, clock, data, wren, q);
    U4: inst_mem PORT MAP (ADDR(6 DOWNTO 0), Clock, DOUT, inst_mem_cs AND W, DIN);

    LED_reg_cs <= '1' WHEN (ADDR(15 DOWNTO 12) = "0001") ELSE '0';
    -- regn(R, Rin, Clock, Q);
    U6: regn PORT MAP (DOUT, LED_reg_cs AND W, Clock, LED_reg);
    LEDR(15 DOWNTO 0) <= LED_reg(15 DOWNTO 0);
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
   PORT (   D, Resetn, Clock  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS (Clock)
   BEGIN
      IF (Clock'EVENT AND Clock = '1') THEN
         IF (Resetn = '0') THEN
            Q <= '0';
         ELSE
            Q <= D;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
DEPTH = 128;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

% This code displays a count (in register R2) on the red LEDs.
00 : 0010010000000000;     %        mvi   R1,#1                    1
01 : 0000000000000001;
02 : 0010100000000000;     %        mvi   R2,#0                    [LED]
03 : 0000000000000000;

04 : 0010110000000000;     % Loop   mvi   R3,#0001000000000000    LED reg address
05 : 0001000000000000;
06 : 1010100110000000;     %        st    R2,R3                    [LED]
07 : 0100100010000000;     %        add   R2,R1                    ++[LED]
08 : 0010110000000000;     %        mvi   R3,#1111111111111111    Delay
09 : 1111111111111111;
0A : 0001011110000000;     %        mv    R5,R7                    Save address of next
 instruction
0B : 0011000000000000;     % Outer  mvi   R4,#10100                Nested delay
0C : 0000000000010100;
0D : 0000001110000000;     %        mv    R0,R7                    Save address of next
 instruction
0E : 0111000010000000;     % Inner  sub   R4,R1                    Decrement R4
0F : 1101110000000000;     %        mvnz  R7,R0                    jnz Inner
10 : 0110110010000000;     %        sub   R3,R1                    Decrement R3
11 : 1101111010000000;     %        mvnz  R7,R5                    jnz Outer

12 : 0011110000000000;     %        mvi   R7,#Loop
13 : 0000000000000100;

END;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Reset with KEY0. SW17 is Run. Set delay using SW15-0
ENTITY part4 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 17);
         Clock                      : IN  STD_LOGIC;
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDR                       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         LEDG                       : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
         DOUT, ADDR                 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
         W, Done                    : BUFFER STD_LOGIC);
END part4;

ARCHITECTURE Behavior OF part4 IS
   COMPONENT proc
      PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, Run   : IN  STD_LOGIC;
               DOUT                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               ADDR                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               W                    : OUT STD_LOGIC;
               Done                 : BUFFER STD_LOGIC);
   END COMPONENT;
   COMPONENT inst_mem
      PORT (   address  : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT regn
      GENERIC (n : INTEGER := 16);
      PORT (   R           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Rin, Clock  : IN  STD_LOGIC;
               Q           : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Resetn, Clock  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT seg7_scroll
      PORT (   Data                       : IN  STD_LOGIC_VECTOR(0 TO 6);
               Addr                       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               Sel, Resetn, Clock         : IN  STD_LOGIC;
               HEX7, HEX6, HEX5, HEX4,
                  HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Sync, Run, inst_mem_cs, LED_reg_cs, seg7_cs : STD_LOGIC;
   SIGNAL DIN, LED_reg, inst_mem_q : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- synchronize the Run input
   U1: flipflop PORT MAP (SW(17), KEY(0), Clock, Sync);
   U2: flipflop PORT MAP (Sync, KEY(0), Clock, Run);

   -- proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   U3: proc PORT MAP (DIN, KEY(0), Clock, Run, DOUT, ADDR, W, Done);

   inst_mem_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0000" ELSE '0';
   -- inst_mem (address, clock, data, wren, q);
   U4: inst_mem PORT MAP (ADDR(6 DOWNTO 0), Clock, DOUT, inst_mem_cs AND W,
      inst_mem_q);
```

```
   DIN <= inst_mem_q;

   LED_reg_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0001" ELSE '0';
   -- regn(R, Rin, Clock, Q);
   U6: regn PORT MAP (DOUT, LED_reg_cs AND W, Clock, LED_reg);
   LEDR(15 DOWNTO 0) <= LED_reg(15 DOWNTO 0);
   LEDG <= "000000000";

   seg7_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0010" ELSE '0';
   U5: seg7_scroll PORT MAP (DOUT(6 DOWNTO 0), ADDR(2 DOWNTO 0), seg7_cs AND W,
      KEY(0), Clock, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
END Behavior;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Data written to registers R0 to R7 are sent to the HEX digits
ENTITY seg7_scroll IS
   PORT (   Data                        : IN  STD_LOGIC_VECTOR(0 TO 6);
            Addr                        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            Sel, Resetn, Clock          : IN  STD_LOGIC;
            HEX7, HEX6, HEX5, HEX4,
               HEX3, HEX2, HEX1, HEX0   : OUT STD_LOGIC_VECTOR(0 TO 6));
END seg7_scroll;

ARCHITECTURE Behavior OF seg7_scroll IS
   COMPONENT regne
      GENERIC (n : INTEGER := 7);
      PORT (   R                  : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Clock, Resetn, E   : IN  STD_LOGIC;
               Q                  : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   SIGNAL R0, R1, R2, R3, R4, R5, R6, R7 : STD_LOGIC_VECTOR(0 TO 6);
   SIGNAL R0_addr, R1_addr, R2_addr, R3_addr, R4_addr, R5_addr, R6_addr, R7_addr :
      STD_LOGIC;
BEGIN
   R0_addr <= '1' WHEN Addr = "000" ELSE '0';
   R1_addr <= '1' WHEN Addr = "001" ELSE '0';
   R2_addr <= '1' WHEN Addr = "010" ELSE '0';
   R3_addr <= '1' WHEN Addr = "011" ELSE '0';
   R4_addr <= '1' WHEN Addr = "100" ELSE '0';
   R5_addr <= '1' WHEN Addr = "101" ELSE '0';
   R6_addr <= '1' WHEN Addr = "110" ELSE '0';
   R7_addr <= '1' WHEN Addr = "111" ELSE '0';

   reg_R0: regne PORT MAP (Data, Clock, Resetn, Sel AND R0_Addr, R0);
   reg_R1: regne PORT MAP (Data, Clock, Resetn, Sel AND R1_Addr, R1);
   reg_R2: regne PORT MAP (Data, Clock, Resetn, Sel AND R2_Addr, R2);
   reg_R3: regne PORT MAP (Data, Clock, Resetn, Sel AND R3_Addr, R3);
   reg_R4: regne PORT MAP (Data, Clock, Resetn, Sel AND R4_Addr, R4);
   reg_R5: regne PORT MAP (Data, Clock, Resetn, Sel AND R5_Addr, R5);
   reg_R6: regne PORT MAP (Data, Clock, Resetn, Sel AND R6_Addr, R6);
   reg_R7: regne PORT MAP (Data, Clock, Resetn, Sel AND R7_Addr, R7);

   HEX7 <= R0;
   HEX6 <= R1;
   HEX5 <= R2;
   HEX4 <= R3;
   HEX3 <= R4;
   HEX2 <= R5;
   HEX1 <= R6;
   HEX0 <= R7;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY regne IS
   GENERIC (n : INTEGER := 7);
   PORT (   R                  : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            Clock, Resetn, E   : IN  STD_LOGIC;
            Q                  : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
END regne;

ARCHITECTURE Behavior OF regne IS
BEGIN
   PROCESS (Clock)
```

```
   BEGIN
      IF Clock'EVENT AND Clock = '1' THEN
         IF Resetn = '0' THEN
            Q <= (OTHERS => '0');
         ELSIF E = '1' THEN
            Q <= R;
         END IF;
      END IF;
   END PROCESS;
END Behavior;
```

```
DEPTH = 128;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

% This code scrolls back and forth the letters dE2 across the 7-segment displays
% and also displays a count (in register R2) on the red LEDs.
00 : 0010000000000000;     %        mvi   R0,#1                    K
01 : 0000000000000001;
02 : 0000010000000000;     %        mv    R1,R0                    1
03 : 0011100000000000;     %        mvi   R6,#Beta                 Q <- *'D'
04 : 0000000001100110;
05 : 0010100000000000;     %        mvi   R2,#0                    [LED]
06 : 0000000000000000;

07 : 0001011100000000;     % Loop   mv    R5,R6                    P <- Q
08 : 0011000000000000;     %        mvi   R4,#H7_address
09 : 0010000000000000;
0A : 1000111010000000;     %        ld    R3,R5                    [P]
0B : 1010111000000000;     %        st    R3,R4                    H7 <- [P]
0C : 0101010010000000;     %        add   R5,R1                    P++
0D : 0101000010000000;     %        add   R4,R1                    H6
0E : 1000111010000000;     %        ld    R3,R5                    [P]
0F : 1010111000000000;     %        st    R3,R4                    H6 <- [P]
10 : 0101010010000000;     %        add   R5,R1                    P++
11 : 0101000010000000;     %        add   R4,R1                    H5
12 : 1000111010000000;     %        ld    R3,R5                    [P]
13 : 1010111000000000;     %        st    R3,R4                    H5 <- [P]
14 : 0101010010000000;     %        add   R5,R1                    P++
15 : 0101000010000000;     %        add   R4,R1                    H4
16 : 1000111010000000;     %        ld    R3,R5                    [P]
17 : 1010111000000000;     %        st    R3,R4                    H4 <- [P]
18 : 0101010010000000;     %        add   R5,R1                    P++
19 : 0101000010000000;     %        add   R4,R1                    H3
1A : 1000111010000000;     %        ld    R3,R5                    [P]
1B : 1010111000000000;     %        st    R3,R4                    H3 <- [P]
1C : 0101010010000000;     %        add   R5,R1                    P++
1D : 0101000010000000;     %        add   R4,R1                    H2
1E : 1000111010000000;     %        ld    R3,R5                    [P]
1F : 1010111000000000;     %        st    R3,R4                    H2 <- [P]
20 : 0101010010000000;     %        add   R5,R1                    P++
21 : 0101000010000000;     %        add   R4,R1                    H1
22 : 1000111010000000;     %        ld    R3,R5                    [P]
23 : 1010111000000000;     %        st    R3,R4                    H1 <- [P]
24 : 0101010010000000;     %        add   R5,R1                    P++
25 : 0101000010000000;     %        add   R4,R1                    H0
26 : 1000111010000000;     %        ld    R3,R5                    [P]
27 : 1010111000000000;     %        st    R3,R4                    H0 <- [P]

28 : 0111100000000000;     %        sub   R6,R0                    Q <- Q - K
29 : 0011010000000000;     %        mvi   R5,#Alpha-1
2A : 0000000001100000;
2B : 0011000000000000;     %        mvi   R4,#Skip
2C : 0000000000110110;
2D : 0111011100000000;     %        sub   R5,R6                    Q == Alpha-1?
2E : 1101111000000000;     %        mvnz  R7,R4                    No
2F : 0101100010000000;     %        add   R6,R1
30 : 0101100010000000;     %        add   R6,R1                    Q <- Alpha+1
31 : 0011010000000000;     %        mvi   R5,#1111111111111111
32 : 1111111111111111;
33 : 0111010000000000;     %        sub   R5,R0
34 : 0101010010000000;     %        add   R5,R1
```

```
35 : 0000001010000000;     %        mv    R0,R5                   K <- -K

36 : 0011010000000000;     % Skip   mvi   R5,#Beta+1
37 : 0000000001100111;
38 : 0011000000000000;     %        mvi   R4,#Cont
39 : 0000000001000011;
3A : 0111011100000000;     %        sub   R5,R6                   Q == Beta+1?
3B : 1101111000000000;     %        mvnz  R7,R4                   No
3C : 0111100010000000;     %        sub   R6,R1
3D : 0111100010000000;     %        sub   R6,R1                   Q <- Beta-1
3E : 0011010000000000;     %        mvi   R5,#1111111111111111
3F : 1111111111111111;
40 : 0111010000000000;     %        sub   R5,R0
41 : 0101010010000000;     %        add   R5,R1
42 : 0000001010000000;     %        mv    R0,R5                   K <- -K

43 : 0011010000000000;     % Cont   mvi   R5,#Temp                 Save reg
44 : 0000000001110000;
45 : 1010001010000000;     %        st    R0,R5

46 : 0010110000000000;     %        mvi   R3,#LED                 LED reg address
47 : 0001000000000000;
48 : 1010100110000000;     %        st    R2,R3                   [LED]
49 : 0100100010000000;     %        add   R2,R1                   ++[LED]
4A : 0010110000000000;     %        mvi   R3,#11111111111111      Delay
4B : 0011111111111111;
4C : 0001011110000000;     %        mv    R5,R7                   Save address of next
 instruction
4D : 0011000000000000;     % Outer  mvi   R4,#101100             Inner loop delay
4E : 0000000000101100;
4F : 1000000000000000;     %        ld    R0,R0                   nop
50 : 0000001110000000;     %        mv    R0,R7                   Save address of next
 instruction
51 : 0111000010000000;     % Inner  sub   R4,R1                   Decrement R4
52 : 1101110000000000;     %        mvnz  R7,R0                   jnz Inner
53 : 0110110010000000;     %        sub   R3,R1                   Decrement R3
54 : 1101111010000000;     %        mvnz  R7,R5                   jnz Outer

55 : 0011010000000000;     %        mvi   R5,#Temp                Restore regs
56 : 0000000001110000;
57 : 1000001010000000;     %        ld    R0,R5

58 : 0011110000000000;     %        mvi   R7,#Loop
59 : 0000000000000111;

60 : 0000000000000000;     % Alpha-1
61 : 1111111111111111;     % Alpha  ' '
62 : 1111111111111111;     %        ' '
63 : 1111111111111111;     %        ' '
64 : 1111111111111111;     %        ' '
65 : 1111111111111111;     %        ' '
66 : 0000000001000010;     % Beta   'd'
67 : 0000000000110000;     % Beta+1 'E'
68 : 0000000000010010;     %        '2'
69 : 1111111111111111;     %        ' '
6A : 1111111111111111;     %        ' '
6B : 1111111111111111;     %        ' '
6C : 1111111111111111;     %        ' '
6D : 1111111111111111;     %        ' '

70 : 0000000000000000;     % Temp
71 : 0000000000000000;     % Temp

END;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Reset with KEY0. SW17 is Run
-- The processor executes the instructions in the file inst_mem.mif
ENTITY part5 IS
PORT (   KEY                        : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
         SW                         : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
         Clock                      : IN  STD_LOGIC;
         HEX7, HEX6, HEX5, HEX4,
            HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6);
         LEDR                       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
         LEDG                       : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
         DOUT, ADDR                 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
         W, Done                    : BUFFER STD_LOGIC);
END part5;

ARCHITECTURE Behavior OF part5 IS
   COMPONENT proc
      PORT (   DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
               Resetn, Clock, Run   : IN  STD_LOGIC;
               DOUT                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               ADDR                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
               W                    : OUT STD_LOGIC;
               Done                 : BUFFER STD_LOGIC);
   END COMPONENT;
   COMPONENT inst_mem
      PORT (   address  : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
               clock    : IN STD_LOGIC ;
               data     : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
               wren     : IN STD_LOGIC  := '1';
               q        : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT regn
      GENERIC (n : INTEGER := 16);
      PORT (   R           : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               Rin, Clock  : IN  STD_LOGIC;
               Q           : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
   END COMPONENT;
   COMPONENT flipflop
      PORT (   D, Resetn, Clock  : IN  STD_LOGIC;
               Q                 : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT seg7_scroll
      PORT (   Data                       : IN  STD_LOGIC_VECTOR(0 TO 6);
               Addr                       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
               Sel, Resetn, Clock         : IN  STD_LOGIC;
               HEX7, HEX6, HEX5, HEX4,
                  HEX3, HEX2, HEX1, HEX0  : OUT STD_LOGIC_VECTOR(0 TO 6));
   END COMPONENT;
   SIGNAL Sync, Run, inst_mem_cs, LED_reg_cs, seg7_cs : STD_LOGIC;
   SIGNAL DIN, LED_reg, SW_reg, inst_mem_q : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
   -- synchronize the Run input
   U1: flipflop PORT MAP (SW(17), KEY(0), Clock, Sync);
   U2: flipflop PORT MAP (Sync, KEY(0), Clock, Run);

   -- proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   U3: proc PORT MAP (DIN, KEY(0), Clock, Run, DOUT, ADDR, W, Done);

   inst_mem_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0000" ELSE '0';
   -- inst_mem (address, clock, data, wren, q);
   U4: inst_mem PORT MAP (ADDR(6 DOWNTO 0), Clock, DOUT, inst_mem_cs AND W,
      inst_mem_q);
```

```
   PROCESS (inst_mem_q, SW_reg, inst_mem_cs)
   BEGIN
      IF inst_mem_cs = '1' THEN
         DIN <= inst_mem_q;
      ELSE
         DIN <= SW_reg;
      END IF;
   END PROCESS;

   LED_reg_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0001" ELSE '0';
   -- regn(R, Rin, Clock, Q);
   U6: regn PORT MAP (DOUT, LED_reg_cs AND W, Clock, LED_reg);
   LEDR(15 DOWNTO 0) <= LED_reg(15 DOWNTO 0);
   LEDG <= "000000000";

   seg7_cs <= '1' WHEN ADDR(15 DOWNTO 12) = "0010" ELSE '0';
   U5: seg7_scroll PORT MAP (DOUT(6 DOWNTO 0), ADDR(2 DOWNTO 0), seg7_cs AND W,
      KEY(0), Clock, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

   -- regn(R, Rin, Clock, Q);
   U7: regn PORT MAP (SW(15 DOWNTO 0), '1', Clock, SW_reg);
END Behavior;
```

```
DEPTH = 128;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

% This code scrolls back and forth the letters dE2 across the 7-segment displays
% and also displays a count (in register R2) on the red LEDs. The speed of scrolling,
% and counting, is controlled by the 16-bit value read from switches SW15-0.
00 : 0010000000000000;     %        mvi   R0,#1                    K
01 : 0000000000000001;
02 : 0000010000000000;     %        mv    R1,R0                    1
03 : 0011100000000000;     %        mvi   R6,#Beta                 Q <- *'D'
04 : 0000000001100110;
05 : 0010100000000000;     %        mvi   R2,#0                    [LED]
06 : 0000000000000000;

07 : 0001011100000000;     % Loop   mv    R5,R6                    P <- Q
08 : 0011000000000000;     %        mvi   R4,#H7_address
09 : 0010000000000000;
0A : 1000111010000000;     %        ld    R3,R5                    [P]
0B : 1010111000000000;     %        st    R3,R4                    H7 <- [P]
0C : 0101010010000000;     %        add   R5,R1                    P++
0D : 0101000010000000;     %        add   R4,R1                    H6
0E : 1000111010000000;     %        ld    R3,R5                    [P]
0F : 1010111000000000;     %        st    R3,R4                    H6 <- [P]
10 : 0101010010000000;     %        add   R5,R1                    P++
11 : 0101000010000000;     %        add   R4,R1                    H5
12 : 1000111010000000;     %        ld    R3,R5                    [P]
13 : 1010111000000000;     %        st    R3,R4                    H5 <- [P]
14 : 0101010010000000;     %        add   R5,R1                    P++
15 : 0101000010000000;     %        add   R4,R1                    H4
16 : 1000111010000000;     %        ld    R3,R5                    [P]
17 : 1010111000000000;     %        st    R3,R4                    H4 <- [P]
18 : 0101010010000000;     %        add   R5,R1                    P++
19 : 0101000010000000;     %        add   R4,R1                    H3
1A : 1000111010000000;     %        ld    R3,R5                    [P]
1B : 1010111000000000;     %        st    R3,R4                    H3 <- [P]
1C : 0101010010000000;     %        add   R5,R1                    P++
1D : 0101000010000000;     %        add   R4,R1                    H2
1E : 1000111010000000;     %        ld    R3,R5                    [P]
1F : 1010111000000000;     %        st    R3,R4                    H2 <- [P]
20 : 0101010010000000;     %        add   R5,R1                    P++
21 : 0101000010000000;     %        add   R4,R1                    H1
22 : 1000111010000000;     %        ld    R3,R5                    [P]
23 : 1010111000000000;     %        st    R3,R4                    H1 <- [P]
24 : 0101010010000000;     %        add   R5,R1                    P++
25 : 0101000010000000;     %        add   R4,R1                    H0
26 : 1000111010000000;     %        ld    R3,R5                    [P]
27 : 1010111000000000;     %        st    R3,R4                    H0 <- [P]

28 : 0111100000000000;     %        sub   R6,R0                    Q <- Q - K
29 : 0011010000000000;     %        mvi   R5,#Alpha-1
2A : 0000000001100000;
2B : 0011000000000000;     %        mvi   R4,#Skip
2C : 0000000000110110;
2D : 0111011100000000;     %        sub   R5,R6                    Q == Alpha-1?
2E : 1101111000000000;     %        mvnz  R7,R4                    No
2F : 0101100010000000;     %        add   R6,R1
30 : 0101100010000000;     %        add   R6,R1                    Q <- Alpha+1
31 : 0011010000000000;     %        mvi   R5,#1111111111111111
32 : 1111111111111111;
33 : 0111010000000000;     %        sub   R5,R0
```

```
34 : 0101010010000000;     %        add   R5,R1
35 : 0000001010000000;     %        mv    R0,R5                    K <- -K

36 : 0011010000000000;     % Skip   mvi   R5,#Beta+1
37 : 0000000001100111;
38 : 0011000000000000;     %        mvi   R4,#Cont
39 : 0000000001000011;
3A : 0111011100000000;     %        sub   R5,R6                    Q == Beta+1?
3B : 1101111000000000;     %        mvnz  R7,R4                    No
3C : 0111100010000000;     %        sub   R6,R1
3D : 0111100010000000;     %        sub   R6,R1                    Q <- Beta-1
3E : 0011010000000000;     %        mvi   R5,#1111111111111111
3F : 1111111111111111;
40 : 0111010000000000;     %        sub   R5,R0
41 : 0101010010000000;     %        add   R5,R1
42 : 0000001010000000;     %        mv    R0,R5                    K <- -K

43 : 0011010000000000;     % Cont   mvi   R5,#Temp                 Save reg
44 : 0000000001110000;
45 : 1010001010000000;     %        st    R0,R5

46 : 0010110000000000;     %        mvi   R3,#LED                  LED reg address
47 : 0001000000000000;
48 : 1010100110000000;     %        st    R2,R3                    [LED]
49 : 0100100010000000;     %        add   R2,R1                    ++[LED]
4A : 0010110000000000;     %        mvi   R3,#11111111111111       Delay
4B : 0011111111111111;
4C : 0001011110000000;     %        mv    R5,R7                    Save address of next
 instruction
4D : 0010000000000000;     % Outer  mvi   R0,#SW                   Read switch valuatio
n
4E : 0011000000000000;
4F : 1001000000000000;     %        ld    R4,R0                    Nested delay
50 : 0000001110000000;     %        mv    R0,R7                    Save address of next
 instruction
51 : 0111000010000000;     % Inner  sub   R4,R1                    Decrement R4
52 : 1101110000000000;     %        mvnz  R7,R0                    jnz Inner
53 : 0110110010000000;     %        sub   R3,R1                    Decrement R3
54 : 1101111010000000;     %        mvnz  R7,R5                    jnz Outer

55 : 0011010000000000;     %        mvi   R5,#Temp                 Restore regs
56 : 0000000001110000;
57 : 1000001010000000;     %        ld    R0,R5

58 : 0011110000000000;     %        mvi   R7,#Loop
59 : 0000000000000111;

60 : 0000000000000000;     % Alpha-1
61 : 1111111111111111;     % Alpha  ' '
62 : 1111111111111111;     %        ' '
63 : 1111111111111111;     %        ' '
64 : 1111111111111111;     %        ' '
65 : 1111111111111111;     %        ' '
66 : 0000000001000010;     % Beta   'd'
67 : 0000000000110000;     % Beta+1 'E'
68 : 0000000000010010;     %        '2'
69 : 1111111111111111;     %        ' '
6A : 1111111111111111;     %        ' '
6B : 1111111111111111;     %        ' '
6C : 1111111111111111;     %        ' '
6D : 1111111111111111;     %        ' '

70 : 0000000000000000;     % Temp
71 : 0000000000000000;     % Temp
```

```
END;
```