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 Verilog HDL 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 Verilog 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 Verilog 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 Verilog module 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 vectors in the Verilog code, as shown. We have used a single assignment statement for all 18 *LEDR* outputs, which is equivalent to the individual assignments

```
assign LEDR[17] = SW[17];
assign LEDR[16] = SW[16];
...
assign 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 Verilog 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 Verilog module. 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.

```
// Simple module that connects the SW switches to the LEDR lights
module  part1 (SW, LEDR);
    input [17:0] SW;         // toggle switches
    output [17:0] LEDR;      // red LEDs

    assign LEDR = SW;
endmodule
```

Figure 1. Verilog 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.

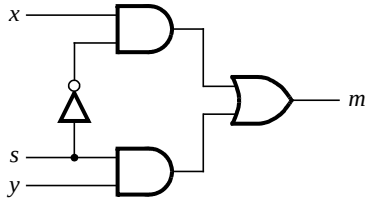
1. Create a new Quartus II project for your circuit. Select Cyclone II EP2C35F672C6 as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a Verilog module for the code in Figure 1 and include it in your project.

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

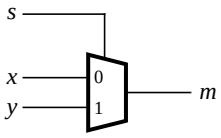
Figure 2a shows a sum-of-products circuit that implements a 2-to-1 *multiplexer* with a select input s. If $s = 0$ the multiplexer's output m is equal to the input x, and if $s = 1$ the output is equal to y. Part b of the figure gives a truth table for this multiplexer, and part c shows its circuit symbol.



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 Verilog statement:

assign m = (~s & x) | (s & y);

You are to write a Verilog module 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.

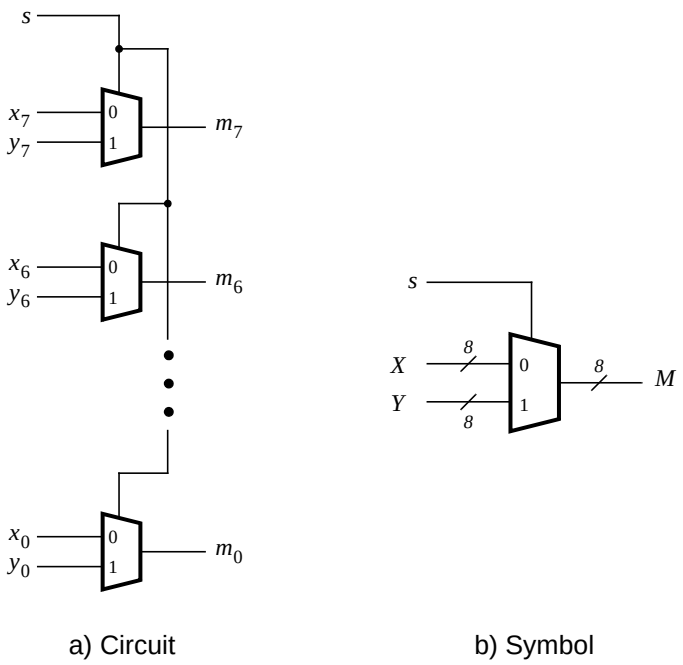


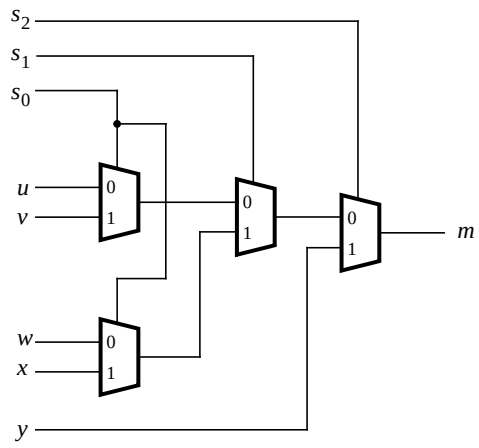
Figure 3. An eight-bit wide 2-to-1 multiplexer.

1. Create a new Quartus II project for your circuit.
2. Include your Verilog 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 Verilog code will use the pins on the Cyclone II FPGA that are connected to the *SW* switches, and the output ports of your Verilog 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.

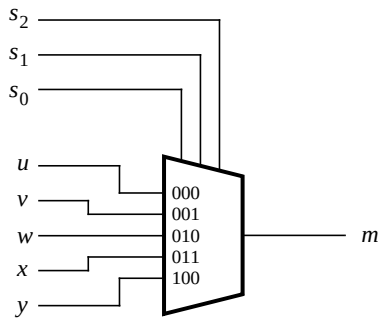
Recall from Figure 3 that an eight-bit wide 2-to-1 multiplexer can be built by using eight instances of a 2-to-1 multiplexer. Figure 5 applies this concept to define a three-bit wide 5-to-1 multiplexer. It contains three instances of the circuit in Figure 4a.



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.

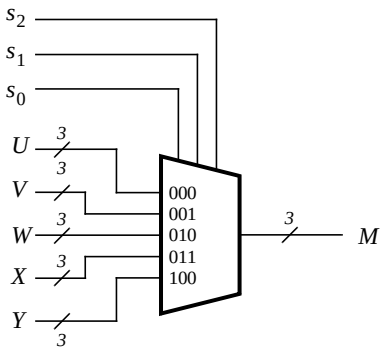


Figure 5. A three-bit wide 5-to-1 multiplexer.

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 Verilog module 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 Verilog module that implements logic functions that represent circuits needed to activate each of the seven segments. Use only simple Verilog **assign** statements in your code to specify each logic function using a Boolean expression.

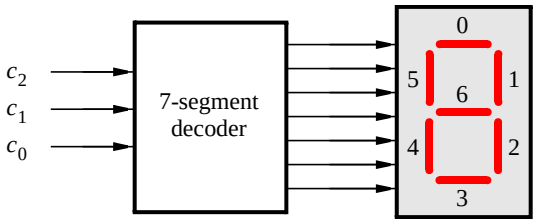


Figure 6. A 7-segment decoder.

$c_2c_1c_0$	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 Verilog module 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

   output [0:6] HEX0;

   in your Verilog 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 Verilog 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.

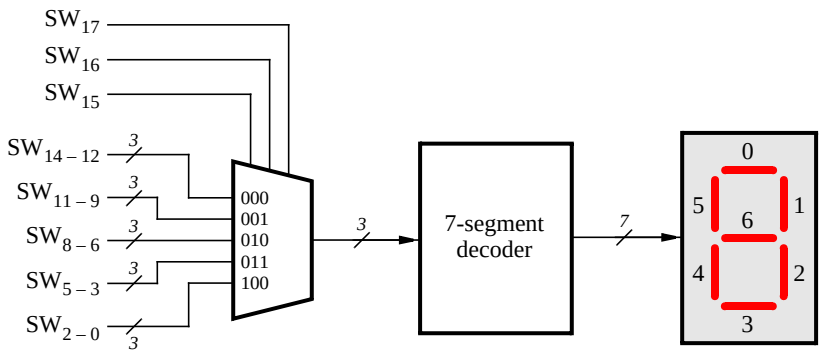


Figure 7. A circuit that can select and display one of five characters.

```
module  part5 (SW, HEX0);
    input [17:0] SW;        // toggle switches
    output [0:6] HEX0;      // 7-seg displays

    wire [2:0] M;

    mux_3bit_5to1 M0 (SW[17:15], SW[14:12], SW[11:9], SW[8:6], SW[5:3], SW[2:0], M);
    char_7seg H0 (M, HEX0);
endmodule

// implements a 3-bit wide 5-to-1 multiplexer
module mux_3bit_5to1 (S, U, V, W, X, Y, M);
    input [2:0] S, U, V, W, X, Y;
    output [2:0] M;

    . . . code not shown

endmodule

// implements a 7-segment decoder for H, E, L, O, and 'blank'
module char_7seg (C, Display);
    input [2:0] C;          // input code
    output [0:6] Display;   // output 7-seg code

    . . . code not shown

endmodule
```

Figure 8. Verilog 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 Verilog module 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 Verilog module 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.

```
// Simple module that connects the SW switches to the LEDR lights
module part1 (SW, LEDR);
   input [17:0] SW;           // toggle switches
   output [17:0] LEDR;        // red LEDs

   assign LEDR[17:0] = SW[17:0];
endmodule
```

```
// 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
module part2 (SW, LEDR, LEDG);
   input [17:0] SW;          // toggle switches
   output [17:0] LEDR;       // red LEDs
   output [7:0] LEDG;        // green LEDs

   wire Sel;
   wire [7:0] X, Y, M;

   assign LEDR = SW;
   assign X = SW[7:0];
   assign Y = SW[15:8];
   assign Sel = SW[17];

   assign M[0] = (~Sel & X[0]) | (Sel & Y[0]);
   assign M[1] = (~Sel & X[1]) | (Sel & Y[1]);
   assign M[2] = (~Sel & X[2]) | (Sel & Y[2]);
   assign M[3] = (~Sel & X[3]) | (Sel & Y[3]);
   assign M[4] = (~Sel & X[4]) | (Sel & Y[4]);
   assign M[5] = (~Sel & X[5]) | (Sel & Y[5]);
   assign M[6] = (~Sel & X[6]) | (Sel & Y[6]);
   assign M[7] = (~Sel & X[7]) | (Sel & Y[7]);
   assign LEDG[7:0] = M;
endmodule
```

```
// 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
module part3 (SW, LEDR, LEDG);
   input [17:0] SW;          // toggle switches
   output [17:0] LEDR;       // red LEDs
   output [2:0] LEDG;        // green LEDs

   wire [1:3] m_0, m_1, m_2;  // 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]
   wire [2:0] S, U, V, W, X, Y, M;  // M is the 3-bit 5-to-1 multiplexer

   assign S[2:0] = SW[17:15];
   assign U = SW[2:0];
   assign V = SW[5:3];
   assign W = SW[8:6];
   assign X = SW[11:9];
   assign Y = SW[14:12];

   assign LEDR = SW;

   // 5-to-1 multiplexer for bit 0
   assign m_0[1] = (~S[0] & U[0]) | (S[0] & V[0]);
   assign m_0[2] = (~S[0] & W[0]) | (S[0] & X[0]);
   assign m_0[3] = (~S[1] & m_0[1]) | (S[1] & m_0[2]);
   assign M[0] = (~S[2] & m_0[3]) | (S[2] & Y[0]); // 5-to-1 multiplexer output

   // 5-to-1 multiplexer for bit 1
   assign m_1[1] = (~S[0] & U[1]) | (S[0] & V[1]);
   assign m_1[2] = (~S[0] & W[1]) | (S[0] & X[1]);
   assign m_1[3] = (~S[1] & m_1[1]) | (S[1] & m_1[2]);
   assign M[1] = (~S[2] & m_1[3]) | (S[2] & Y[1]); // 5-to-1 multiplexer output

   // 5-to-1 multiplexer for bit 2
   assign m_2[1] = (~S[0] & U[2]) | (S[0] & V[2]);
   assign m_2[2] = (~S[0] & W[2]) | (S[0] & X[2]);
   assign m_2[3] = (~S[1] & m_2[1]) | (S[1] & m_2[2]);
   assign M[2] = (~S[2] & m_2[3]) | (S[2] & Y[2]); // 5-to-1 multiplexer output

   assign LEDG[2:0] = M;
endmodule
```

```
// 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
module part4 (SW, LEDR, HEX0);
   input [2:0] SW;            // toggle switches
   output [2:0] LEDR;         // red LEDs
   output [0:6] HEX0;         // 7-seg display

   wire [2:0] C;

   assign LEDR = SW;
   assign C[2:0] = SW[2:0];

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   // the following equations describe HEX0[0-6] in cannonical SOP form
   assign HEX0[0] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & C[0]));
   assign HEX0[1] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign HEX0[2] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign HEX0[3] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & ~C[0]) |
      (~C[2] & C[1] & C[0]));
   assign HEX0[4] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign HEX0[5] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign HEX0[6] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]));
endmodule
```

```
// 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")
module part5 (SW, LEDR, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [17:0] SW;           // toggle switches
   output [17:0] LEDR;        // red LEDs
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;   // 7-seg displays

   assign LEDR = SW;

   wire [2:0] Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank;
   wire [2:0] H4_Ch, H3_Ch, H2_Ch, H1_Ch, H0_Ch;
   assign Ch_Sel = SW[17:15];
   assign Ch1 = SW[14:12];
   assign Ch2 = SW[11:9];
   assign Ch3 = SW[8:6];
   assign Ch4 = SW[5:3];
   assign Ch5 = SW[2:0];
   assign Blank = 3'b111;  // used to blank a 7-seg display (see module char_7seg)

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

   // instantiate module char_7seg (C, Display);
   char_7seg H7 (Blank, HEX7);
   char_7seg H6 (Blank, HEX6);
   char_7seg H5 (Blank, HEX5);
   char_7seg H4 (H4_Ch, HEX4);
   char_7seg H3 (H3_Ch, HEX3);
   char_7seg H2 (H2_Ch, HEX2);
   char_7seg H1 (H1_Ch, HEX1);
   char_7seg H0 (H0_Ch, HEX0);
endmodule

// Implements a 3-bit wide 5-to-1 multiplexer
module mux_3bit_5to1 (S, U, V, W, X, Y, M);
   input [2:0] S, U, V, W, X, Y;
   output [2:0] M;
   wire [1:3] m_0, m_1, m_2;  // intermediate multiplexers

   // 5-to-1 multiplexer for bit 0
   assign m_0[1] = (~S[0] & U[0]) | (S[0] & V[0]);
   assign m_0[2] = (~S[0] & W[0]) | (S[0] & X[0]);
```

```
   assign m_0[3] = (~S[1] & m_0[1]) | (S[1] & m_0[2]);
   assign M[0] = (~S[2] & m_0[3]) | (S[2] & Y[0]); // 5-to-1 multiplexer output

   // 5-to-1 multiplexer for bit 1
   assign m_1[1] = (~S[0] & U[1]) | (S[0] & V[1]);
   assign m_1[2] = (~S[0] & W[1]) | (S[0] & X[1]);
   assign m_1[3] = (~S[1] & m_1[1]) | (S[1] & m_1[2]);
   assign M[1] = (~S[2] & m_1[3]) | (S[2] & Y[1]); // 5-to-1 multiplexer output

   // 5-to-1 multiplexer for bit 2
   assign m_2[1] = (~S[0] & U[2]) | (S[0] & V[2]);
   assign m_2[2] = (~S[0] & W[2]) | (S[0] & X[2]);
   assign m_2[3] = (~S[1] & m_2[1]) | (S[1] & m_2[2]);
   assign M[2] = (~S[2] & m_2[3]) | (S[2] & Y[2]); // 5-to-1 multiplexer output
endmodule

// 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
//
module char_7seg (C, Display);
   input [2:0] C;          // input code
   output [0:6] Display;   // output 7-seg code

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   // the following equations describe display functions in cannonical SOP form
   assign Display[0] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[1] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[2] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[3] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & ~C[0]) |
      (~C[2] & C[1] & C[0]));
   assign Display[4] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[5] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[6] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]));
endmodule
```

```
// 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
module part6 (SW, LEDR, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [17:0] SW;           // toggle switches
   output [17:0] LEDR;        // red LEDs
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;   // 7-seg displays

   assign LEDR = SW;

   wire [2:0] Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank;
   wire [2:0] H7_Ch, H6_Ch, H5_Ch, H4_Ch, H3_Ch, H2_Ch, H1_Ch, H0_Ch;
   assign Ch_Sel = SW[17:15];
   assign Ch1 = SW[14:12];
   assign Ch2 = SW[11:9];
   assign Ch3 = SW[8:6];
   assign Ch4 = SW[5:3];
   assign Ch5 = SW[2:0];
   assign Blank = 3'b111;  // 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
   mux_3bit_8to1 M7 (Ch_Sel, Blank, Blank, Blank, Ch1, Ch2, Ch3, Ch4, Ch5, H7_Ch);
   mux_3bit_8to1 M6 (Ch_Sel, Blank, Blank, Ch1, Ch2, Ch3, Ch4, Ch5, Blank, H6_Ch);
   mux_3bit_8to1 M5 (Ch_Sel, Blank, Ch1, Ch2, Ch3, Ch4, Ch5, Blank, Blank, H5_Ch);
   mux_3bit_8to1 M4 (Ch_Sel, Ch1, Ch2, Ch3, Ch4, Ch5, Blank, Blank, Blank, H4_Ch);
   mux_3bit_8to1 M3 (Ch_Sel, Ch2, Ch3, Ch4, Ch5, Blank, Blank, Blank, Ch1, H3_Ch);
   mux_3bit_8to1 M2 (Ch_Sel, Ch3, Ch4, Ch5, Blank, Blank, Blank, Ch1, Ch2, H2_Ch);
   mux_3bit_8to1 M1 (Ch_Sel, Ch4, Ch5, Blank, Blank, Blank, Ch1, Ch2, Ch3, H1_Ch);
   mux_3bit_8to1 M0 (Ch_Sel, Ch5, Blank, Blank, Blank, Ch1, Ch2, Ch3, Ch4, H0_Ch);

   // iinstantiate module char_7seg (C, Display) to drive the hex displays
   char_7seg H7 (H7_Ch, HEX7);
   char_7seg H6 (H6_Ch, HEX6);
   char_7seg H5 (H5_Ch, HEX5);
   char_7seg H4 (H4_Ch, HEX4);
   char_7seg H3 (H3_Ch, HEX3);
   char_7seg H2 (H2_Ch, HEX2);
   char_7seg H1 (H1_Ch, HEX1);
   char_7seg H0 (H0_Ch, HEX0);
endmodule

// implements a 3-bit wide 8-to-1 multiplexer
module mux_3bit_8to1 (S, G1, G2, G3, G4, G5, G6, G7, G8, M);
```

```
   input [2:0] S, G1, G2, G3, G4, G5, G6, G7, G8;
   output [2:0] M;
   wire [1:6] m_0, m_1, m_2;  // intermediate multiplexers

   // 8-to-1 multiplexer for bit 0
   assign m_0[1] = (~S[0] & G1[0]) | (S[0] & G2[0]);
   assign m_0[2] = (~S[0] & G3[0]) | (S[0] & G4[0]);
   assign m_0[3] = (~S[0] & G5[0]) | (S[0] & G6[0]);
   assign m_0[4] = (~S[0] & G7[0]) | (S[0] & G8[0]);
   assign m_0[5] = (~S[1] & m_0[1]) | (S[1] & m_0[2]);
   assign m_0[6] = (~S[1] & m_0[3]) | (S[1] & m_0[4]);
   assign M[0] = (~S[2] & m_0[5]) | (S[2] & m_0[6]);

   // 8-to-1 multiplexer for bit 1
   assign m_1[1] = (~S[0] & G1[1]) | (S[0] & G2[1]);
   assign m_1[2] = (~S[0] & G3[1]) | (S[0] & G4[1]);
   assign m_1[3] = (~S[0] & G5[1]) | (S[0] & G6[1]);
   assign m_1[4] = (~S[0] & G7[1]) | (S[0] & G8[1]);
   assign m_1[5] = (~S[1] & m_1[1]) | (S[1] & m_1[2]);
   assign m_1[6] = (~S[1] & m_1[3]) | (S[1] & m_1[4]);
   assign M[1] = (~S[2] & m_1[5]) | (S[2] & m_1[6]);

   // 8-to-1 multiplexer for bit 2
   assign m_2[1] = (~S[0] & G1[2]) | (S[0] & G2[2]);
   assign m_2[2] = (~S[0] & G3[2]) | (S[0] & G4[2]);
   assign m_2[3] = (~S[0] & G5[2]) | (S[0] & G6[2]);
   assign m_2[4] = (~S[0] & G7[2]) | (S[0] & G8[2]);
   assign m_2[5] = (~S[1] & m_2[1]) | (S[1] & m_2[2]);
   assign m_2[6] = (~S[1] & m_2[3]) | (S[1] & m_2[4]);
   assign M[2] = (~S[2] & m_2[5]) | (S[2] & m_2[6]);
endmodule

// 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
//
module char_7seg (C, Display);
   input [2:0] C;          // input code
   output [0:6] Display;         // output 7-seg code

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
```

```
   // the following equations describe display functions in cannonical SOP form
   assign Display[0] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[1] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[2] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[3] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & ~C[0]) |
      (~C[2] & C[1] & C[0]));
   assign Display[4] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[5] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
      (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
   assign Display[6] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]));
endmodule
```

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 Verilog **assign** statements in your code and specify each logic function as a Boolean expression.
2. Write a Verilog 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 Verilog 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 Verilog module which includes the comparator, multiplexers, and circuit *A* (do not include circuit *B* or the 7-segment decoder at this point). Your Verilog module 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 Verilog **assign** statements to specify the required logic functions using Boolean expressions. Your Verilog 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 Verilog module.
2. Compile the circuit and use functional simulation to verify the correct operation of your comparator, multiplexers, and circuit *A*.
3. Augment your Verilog 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.

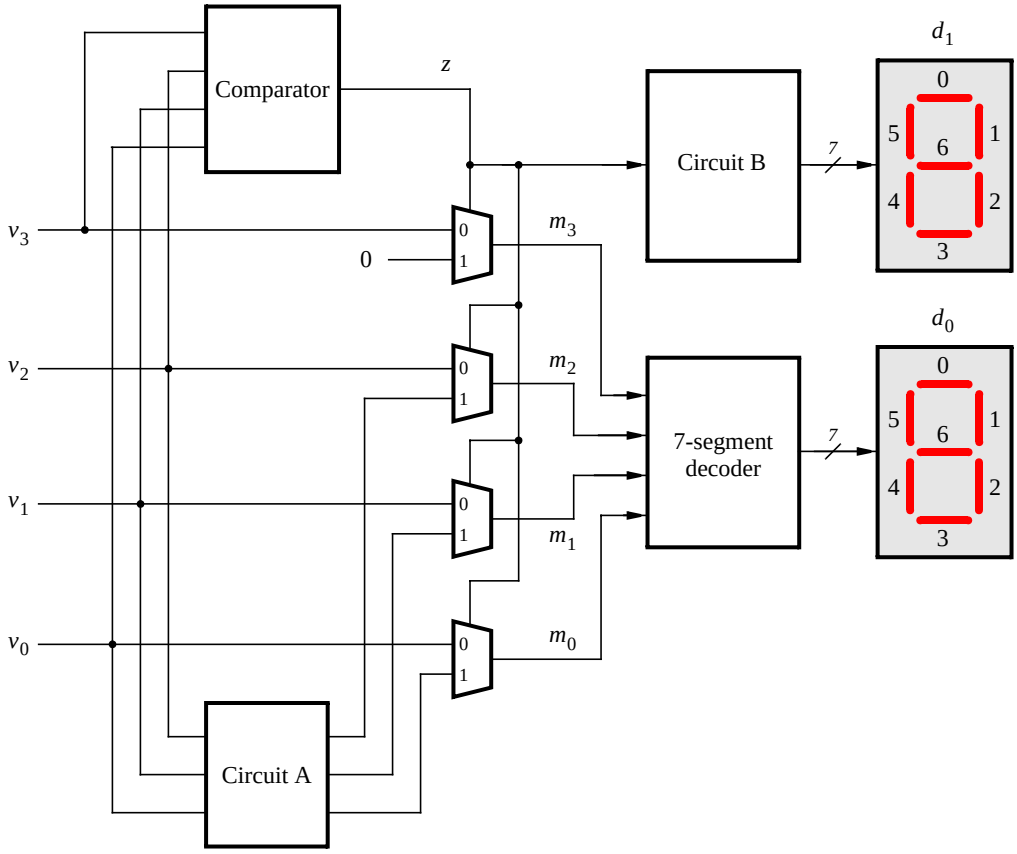
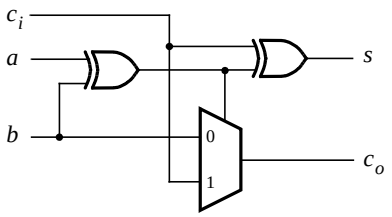


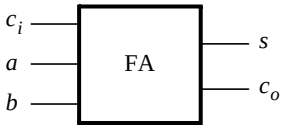
Figure 1. Partial design of the binary-to-decimal conversion circuit.

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 module 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 Verilog code that implements this circuit, as described below.



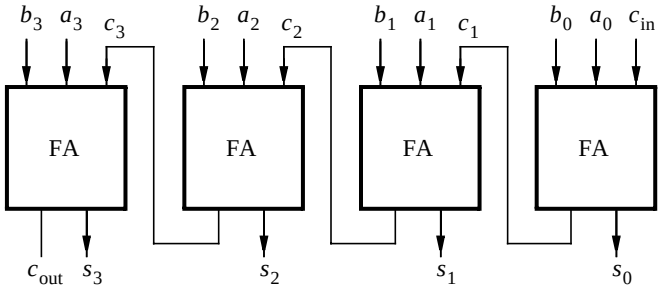
a) Full adder circuit



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 Verilog module for the full adder subcircuit and write a top-level Verilog module 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 Verilog code using simple **assign** statements to specify the required logic functions–do not use other types of Verilog 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 Verilog code for a two-digit BCD adder by using two instances of the Verilog code for a one-digit BCD adder from part IV. A different approach for describing the two-digit BCD adder in Verilog 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 Verilog code that corresponds to this pseudo-code. Note that you can perform addition operations in your Verilog 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 Verilog compiler to design the circuit by using **if-else** statements along with the Verilog $>$ and $+$ operators. Perform the following steps:

1. Create a new Quartus II project for your Verilog 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 Verilog 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.
module part1 (SW, LEDR, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [15:0] LEDR;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   assign LEDR = SW;

   // drive the displays through 7-seg decoders
   bcd7seg digit3 (SW[15:12], HEX3);
   bcd7seg digit2 (SW[11:8], HEX2);
   bcd7seg digit1 (SW[7:4], HEX1);
   bcd7seg digit0 (SW[3:0], HEX0);

endmodule

module bcd7seg (B, H);
   input [3:0] B;
   output [0:6] H;

   wire [0:6] H;

   /*
    *       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;
   assign H[0] = (B[2] & ~B[0]) | (~B[3] & ~B[2] & ~B[1] & B[0]);
   assign H[1] = (B[2] & ~B[1] & B[0]) | (B[2] & B[1] & ~B[0]);
   assign H[2] = (~B[2] & B[1] & ~B[0]);
   assign H[3] = (~B[2] & ~B[1] & B[0]) | (B[2] & ~B[1] & ~B[0]) |
      (B[2] & B[1] & B[0]);
   assign H[4] = (~B[1] & B[0]) | (~B[3] & B[0]) | (~B[3] & B[2] & ~B[1]);
   assign H[5] = (B[1] & B[0]) | (~B[2] & B[1]) | (~B[3] & ~B[2] & B[0]);
   assign H[6] = (B[2] & B[1] & B[0]) | (~B[3] & ~B[2] & ~B[1]);
endmodule
```

```
// bcd-to-decimal converter
module part2 (SW, HEX1, HEX0);
   input [3:0] SW;
   output [0:6] HEX1, HEX0;

   wire[3:0] V, M;
   wire[2:0] B;
   wire z;

   assign V = SW;

   // circuit A
   assign z = (V[3] & V[2]) | (V[3] & V[1]);

   // Circuit B
   assign B[2] = V[2] & V[1];
   assign B[1] = V[2] & ~V[1];
   assign B[0] = (V[1] & V[0]) | (V[2] & V[0]);

   // multiplexers
   assign M[3] = ~z & V[3];
   assign M[2] = (~z & V[2]) | (z & B[2]);
   assign M[1] = (~z & V[1]) | (z & B[1]);
   assign M[0] = (~z & V[0]) | (z & B[0]);

   // Circuit D
   bcd7seg Circuit_D (M, HEX0);

   // Circuit C
   assign HEX1 = {1'b1, ~z, ~z, 4'b1111}; // display a blank or the digit 1

endmodule

module bcd7seg (B, H);
   input [3:0] B;
   output [0:6] H;

   wire [0:6] H;

   /*
    *       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;
```

```
   assign H[0] = (B[2] & ~B[0]) | (~B[3] & ~B[2] & ~B[1] & B[0]);
   assign H[1] = (B[2] & ~B[1] & B[0]) | (B[2] & B[1] & ~B[0]);
   assign H[2] = (~B[2] & B[1] & ~B[0]);
   assign H[3] = (~B[2] & ~B[1] & B[0]) | (B[2] & ~B[1] & ~B[0]) |
      (B[2] & B[1] & B[0]);
   assign H[4] = (~B[1] & B[0]) | (~B[3] & B[0]) | (~B[3] & B[2] & ~B[1]);
   assign H[5] = (B[1] & B[0]) | (~B[2] & B[1]) | (~B[3] & ~B[2] & B[0]);
   assign H[6] = (B[2] & B[1] & B[0]) | (~B[3] & ~B[2] & ~B[1]);
endmodule
```

```
// 4-bit ripple-carry adder
module part3 (SW, LEDR, LEDG);
   input [7:0] SW;
   output [7:0] LEDR;
   output [4:0] LEDG;

   wire [3:0] A, B, S;
   wire [4:1] C;     // carries

   assign A = SW[7:4];
   assign B = SW[3:0];

   fa bit0 (A[0], B[0], 1'b0, S[0], C[1]);
   fa bit1 (A[1], B[1], C[1], S[1], C[2]);
   fa bit2 (A[2], B[2], C[2], S[2], C[3]);
   fa bit3 (A[3], B[3], C[3], S[3], C[4]);

   // Display the inputs
   assign LEDR = SW;
   assign LEDG = {C[4], S};

endmodule

module fa (a, b, ci, s, co);
   input a, b, ci;
   output s, co;

   wire a_xor_b;

   assign a_xor_b = a ^ b;
   assign s = a_xor_b ^ ci;
   assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule
```

```
// 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 HEX0
module part4 (SW, LEDR, LEDG, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [8:0] SW;
   output [8:0] LEDR;
   output [8:0] LEDG;

   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [3:0] A, B, S;        // S is the sum output of the adder
   wire Cin;                  // carry in
   wire [4:1] C;              // internal carries
   wire [3:0] S0;
   wire [3:0] S0_M;           // modified S0 for sums 16, 17, 18, 19
   wire S1;

   assign A = SW[7:4];
   assign B = SW[3:0];
   assign Cin = SW[8];
   assign LEDR = SW;

   fa bit0 (A[0], B[0], Cin, S[0], C[1]);
   fa bit1 (A[1], B[1], C[1], S[1], C[2]);
   fa bit2 (A[2], B[2], C[2], S[2], C[3]);
   fa bit3 (A[3], B[3], C[3], S[3], C[4]);
   assign LEDG[4:0] = {C[4], S};

   // Display the inputs
   bcd7seg H_6 (A, HEX6);
   assign HEX7 = {7'b1111111};   // display blank

   bcd7seg H_4 (B, HEX4);
   assign HEX5 = {7'b1111111};   // display blank

   // Detect illegal inputs, display on LEDG[8]
   assign LEDG[8] = (A[3] & A[2]) | (A[3] & A[1]) |
      (B[3] & B[2]) | (B[3] & B[1]);
   assign LEDG[7:5] = 3'b000;

   // Display the sum
   // module bcd_decimal (V, z, M);
   bcd_decimal BCD_S (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:
   assign S0_M[3] = (~C[4] & S0[3]) | (C[4] & S0[1]);
   assign S0_M[2] = (~C[4] & S0[2]) | (C[4] & ~S0[1]);
   assign S0_M[1] = (~C[4] & S0[1]) | (C[4] & ~S0[1]);
   assign S0_M[0] = S0[0];
   bcd7seg H_0 (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
   assign HEX1 = {1'b1, ~(S1 | C[4]), ~(S1 | C[4]), 4'b1111};  // display blank or 1
   assign HEX2 = {7'b1111111};   // display blank
   assign HEX3 = {7'b1111111};   // display blank

endmodule

module fa (a, b, ci, s, co);
```

```
   input a, b, ci;
   output s, co;

   wire a_xor_b;

   assign a_xor_b = a ^ b;
   assign s = a_xor_b ^ ci;
   assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule

// bcd-to-decimal converter
module bcd_decimal (V, z, M);
   input [3:0] V;
   output z;
   output [3:0] M;

   wire[2:0] B;

   // circuit A
   assign z = (V[3] & V[2]) | (V[3] & V[1]);

   // Circuit B
   assign B[2] = V[2] & V[1];
   assign B[1] = V[2] & ~V[1];
   assign B[0] = (V[1] & V[0]) | (V[2] & V[0]);

   // multiplexers
   assign M[3] = ~z & V[3];
   assign M[2] = (~z & V[2]) | (z & B[2]);
   assign M[1] = (~z & V[1]) | (z & B[1]);
   assign M[0] = (~z & V[0]) | (z & B[0]);

endmodule

module bcd7seg (B, H);
   input [3:0] B;
   output [0:6] H;

   wire [0:6] H;

   /*
    *       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;
   assign H[0] = (B[2] & ~B[0]) | (~B[3] & ~B[2] & ~B[1] & B[0]);
   assign H[1] = (B[2] & ~B[1] & B[0]) | (B[2] & B[1] & ~B[0]);
   assign H[2] = (~B[2] & B[1] & ~B[0]);
   assign H[3] = (~B[2] & ~B[1] & B[0]) | (B[2] & ~B[1] & ~B[0]) |
      (B[2] & B[1] & B[0]);
   assign H[4] = (~B[1] & B[0]) | (~B[3] & B[0]) | (~B[3] & B[2] & ~B[1]);
   assign H[5] = (B[1] & B[0]) | (~B[2] & B[1]) | (~B[3] & ~B[2] & B[0]);
   assign H[6] = (B[2] & B[1] & B[0]) | (~B[3] & ~B[2] & ~B[1]);
endmodule
```

```
// 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
module part5 (SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire[3:0] A1, A0, B1, B0;
   assign A1 = SW[15:12];
   assign A0 = SW[11:8];
   assign B1 = SW[7:4];
   assign B0 = SW[3:0];

   wire [3:0] S0, S1;
   wire C1, C2, S2;

   // module part4 (A, B, Cin, S1, S0);
   part4 BCD_0 (A0, B0, 1'b0, C1, S0);
   part4 BCD_1 (A1, B1, C1, C2, S1);
   assign S2 = C2;

   // drive the displays through 7-seg decoders
   bcd7seg digit7 (A1, HEX7);
   bcd7seg digit6 (A0, HEX6);
   bcd7seg digit5 (B1, HEX5);
   bcd7seg digit4 (B0, HEX4);
   bcd7seg digit2 ({3'b000, S2}, HEX2);
   bcd7seg digit1 (S1, HEX1);
   bcd7seg digit0 (S0, HEX0);

   assign HEX3 = 7'b1111111;  // turn off HEX3

endmodule

// one digit BCD adder S1 S0 = A + B + Cin
module part4 (A, B, Cin, S1, S0);
   input [3:0] A, B;
   input Cin;
   output S1;              // S1 is really just a carry out
   output [3:0] S0;        // S0 is the least signisum output of the adder

   wire [4:1] C;           // internal carries
   wire [3:0] S0_M;        // used because S0 has to be modified for sums 16, 17, 18,
19
   wire S1_M;              // used because S1 has to be modified for sums 16, 17, 18,
19
   wire [3:0] S;           // S is the sum produced by the adder

   fa bit0 (A[0], B[0], Cin, S[0], C[1]);
   fa bit1 (A[1], B[1], C[1], S[1], C[2]);
   fa bit2 (A[2], B[2], C[2], S[2], C[3]);
   fa bit3 (A[3], B[3], C[3], S[3], C[4]);

   // convert the sum to BCD
   bcd_decimal BCD_S (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:
   assign S0[3] = (~C[4] & S0_M[3]) | (C[4] & S0_M[1]);
   assign S0[2] = (~C[4] & S0_M[2]) | (C[4] & ~S0_M[1]);
   assign S0[1] = (~C[4] & S0_M[1]) | (C[4] & ~S0_M[1]);
```

```
   assign 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
   assign S1 = S1_M | C[4];
endmodule

module fa (a, b, ci, s, co);
   input a, b, ci;
   output s, co;

   wire a_xor_b;

   assign a_xor_b = a ^ b;
   assign s = a_xor_b ^ ci;
   assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule

// bcd-to-decimal converter
module bcd_decimal (V, z, M);
   input [3:0] V;
   output z;
   output [3:0] M;

   wire[2:0] B;
   wire z;

   // circuit A
   assign z = (V[3] & V[2]) | (V[3] & V[1]);

   // Circuit B
   assign B[2] = V[2] & V[1];
   assign B[1] = V[2] & ~V[1];
   assign B[0] = (V[1] & V[0]) | (V[2] & V[0]);

   // multiplexers
   assign M[3] = ~z & V[3];
   assign M[2] = (~z & V[2]) | (z & B[2]);
   assign M[1] = (~z & V[1]) | (z & B[1]);
   assign M[0] = (~z & V[0]) | (z & B[0]);

endmodule

module bcd7seg (B, H);
   input [3:0] B;
   output [0:6] H;

   wire [0:6] H;

   /*
    *       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;
    assign H[0] = (B[2] & ~B[0]) | (~B[3] & ~B[2] & ~B[1] & B[0]);
    assign H[1] = (B[2] & ~B[1] & B[0]) | (B[2] & B[1] & ~B[0]);
    assign H[2] = (~B[2] & B[1] & ~B[0]);
    assign H[3] = (~B[2] & ~B[1] & B[0]) | (B[2] & ~B[1] & ~B[0]) |
       (B[2] & B[1] & B[0]);
    assign H[4] = (~B[1] & B[0]) | (~B[3] & B[0]) | (~B[3] & B[2] & ~B[1]);
    assign H[5] = (B[1] & B[0]) | (~B[2] & B[1]) | (~B[3] & ~B[2] & B[0]);
    assign H[6] = (B[2] & B[1] & B[0]) | (~B[3] & ~B[2] & ~B[1]);
endmodule
```

```
// 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
module part6 (SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire[3:0] A1, A0, B1, B0;
   assign A1 = SW[15:12];
   assign A0 = SW[11:8];
   assign B1 = SW[7:4];
   assign B0 = SW[3:0];

   wire [3:0] S0, S1;
   wire S2;
   reg C1, C2;

   wire [4:0] T1, T0;         // used for bcd addition
   reg [3:0] Z1, Z0;       // used for bcd addition

   // add lower two bcd digits. Result is five bits: C1,S0
   assign T0 = {1'b0,A0} + {1'b0,B0};
   always @ (T0)
   begin
      if (T0 > 5'd9)
      begin
         Z0 = 4'd6;  // we will add +6 instead of -10
         C1 = 1'b1;
      end
      else
      begin
         Z0 = 4'd0;
         C1 = 1'b0;
      end
   end
   assign S0 = T0[3:0] + Z0;  // using 4 bits, + 6 is same as - 10

   // add upper two bcd digits plus C1
   assign T1 = {1'b0,A1} + {1'b0,B1} + C1;
   always @ (T1)
   begin
      if (T1 > 5'd9)
      begin
         Z1 = 4'd6;  // we will add +6 instead of -10
         C2 = 1'b1;
      end
      else
      begin
         Z1 = 4'd0;
         C2 = 1'b0;
      end
   end
   assign S1 = T1[3:0] + Z1; // using 4 bits, + 6 is same as - 10
   assign S2 = C2;

   // drive the displays through 7-seg decoders
   bcd7seg digit7 (A1, HEX7);
   bcd7seg digit6 (A0, HEX6);
   bcd7seg digit5 (B1, HEX5);
   bcd7seg digit4 (B0, HEX4);
   bcd7seg digit2 ({3'b000, S2}, HEX2);
```

```
   bcd7seg digit1 (S1, HEX1);
   bcd7seg digit0 (S0, HEX0);

   assign HEX3 = 7'b1111111;  // turn off HEX3
endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'b1111111;
      endcase
endmodule
```

```
// input six bits using SW toggle switches, and convert to decimal (2-digit bcd)
module part7 (SW, HEX3, HEX2, HEX1, HEX0);
   input [5:0] SW;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   reg [3:0] bcd_h, bcd_l;
   wire [5:0] bin6;

   assign bin6 = SW;

   // Check various ranges and set bcd digits. Note that we work with
   // bin6[3:0] just to prevent compiler warnings about bit size
   // truncation. This is not really necessary
   always @ (bin6)
   begin
      if (bin6 < 10)
      begin
         bcd_h = 4'h0;
         bcd_l = bin6[3:0];
      end
      else if (bin6 < 20)
      begin
         bcd_h = 4'h1;
         bcd_l = /* bin6 - 10 */ bin6[3:0] + 4'h6; // -10 = 11110110. So, add 6
      end
      else if (bin6 < 30)
      begin
         bcd_h = 4'h2;
         bcd_l = /* bin6 - 20 */ bin6[3:0] + 4'hC; // -20 = 11101100. So, add 12
      end
      else if (bin6 < 40)
      begin
         bcd_h = 4'h3;
         bcd_l = /* bin6 - 30 */ bin6[3:0] + 4'h2; // -30 = 11100010. So, add 2
      end
      else if (bin6 < 50)
      begin
         bcd_h = 4'h4;
         bcd_l = /* bin6 - 40 */ bin6[3:0] + 4'h8; // -40 = 11011000. So, add 8
      end
      else if (bin6 < 60)
      begin
         bcd_h = 4'h5;
         bcd_l = /* bin6 - 50 */ bin6[3:0] + 4'hE; // -50 = 11001110. So, add 14
      end
      else
      begin
         bcd_h = 4'h6;
         bcd_l = /* bin6 - 60 */ bin6[3:0] + 4'h4; // -60 = 11000100. So, add 4
      end
   end

   // drive the displays
   bcd7seg digit1 (bcd_h, HEX1);
   bcd7seg digit0 (bcd_l, HEX0);

   // blank the adjacent displays
   bcd7seg digit3 (4'hF, HEX3);
   bcd7seg digit2 (4'hF, HEX2);

endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
```

```
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'b1111111;
      endcase
endmodule
```

Laboratory Exercise 3

Multipliers

The purpose of this exercise is to design a combinational circuit that can multiply two unsigned numbers. First, you will design the multiplier by writing Verilog code that describes the desired circuit. Then, you will use a predefined subcircuit for the multiplier from Altera's library of parameterized modules (LPMs), and compare the results achieved. To make the design task more manageable, start with a simple case of 4-bit numbers.

Part I

Figure 1a 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 1c shows how each summand can be formed by using the Boolean AND operation of A with the appropriate bit in B.

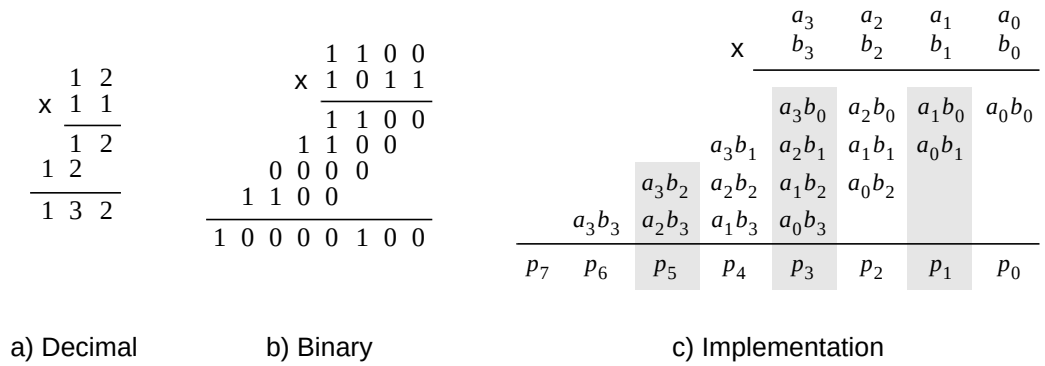


Figure 1. Multiplication of binary numbers.

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 2. 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 1c. 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.

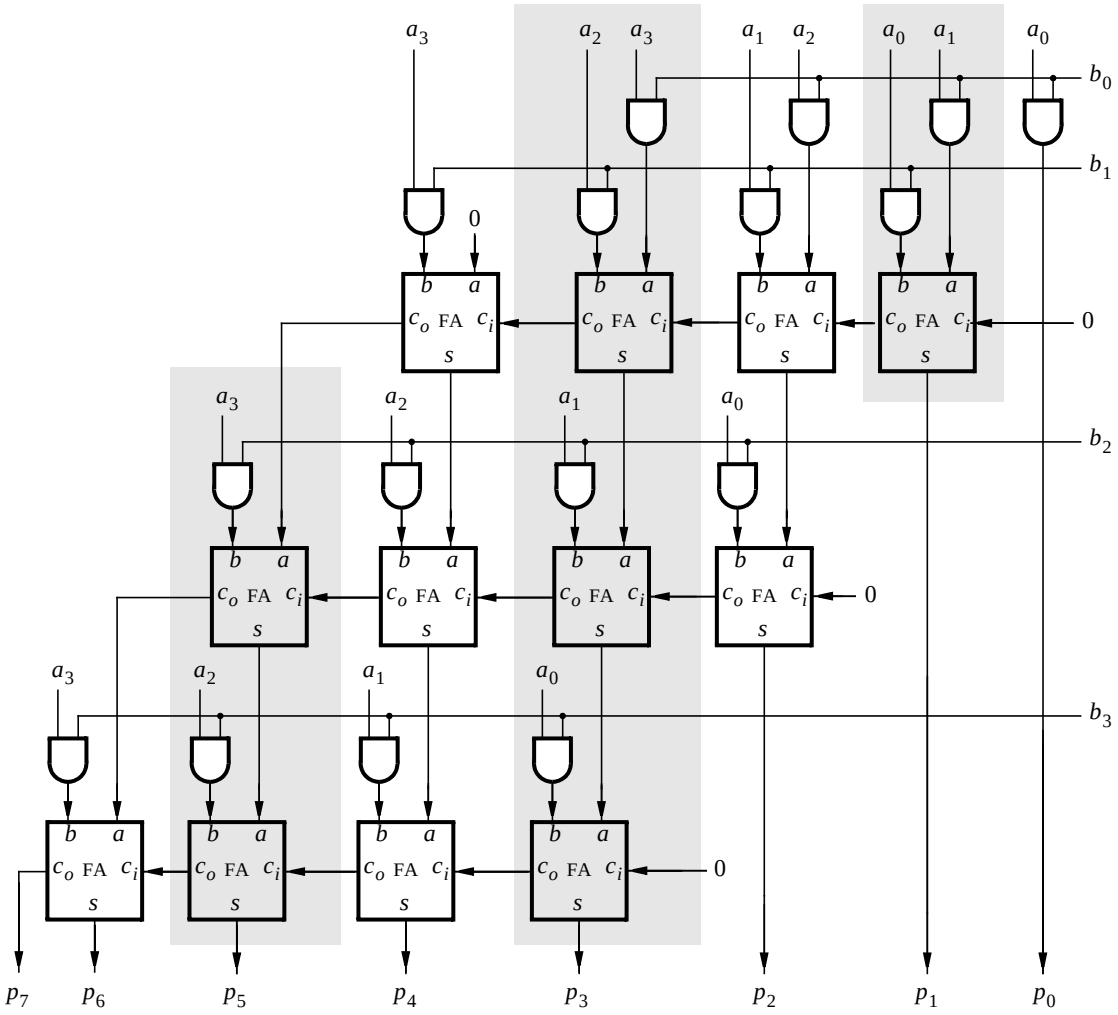


Figure 2. An array multiplier circuit.

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 Verilog 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 II

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*.

Part III

Change your Verilog 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.

```
// 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
module part1 (SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [3:0] A, B;
   wire [7:0] P;
   wire [3:1] C_b1;  // carries for row that ANDs with B1
   wire [5:2] PP1;   // partial products from row that ANDs with B1
   wire [3:1] C_b2;  // carries for row that ANDs with B2
   wire [6:3] PP2;   // partial products from row that ANDs with B2
   wire [3:1] C_b3;  // carries for row that ANDs with B3

   assign A = SW[11:8];
   assign B = SW[3:0];
   assign P[0] = A[0] & B[0];

   // module fa (a, b, ci, s, co);
   fa b1_a0 (A[1] & B[0], A[0] & B[1], 1'b0,    P[1],   C_b1[1]);
   fa b1_a1 (A[2] & B[0], A[1] & B[1], C_b1[1], PP1[2], C_b1[2]);
   fa b1_a2 (A[3] & B[0], A[2] & B[1], C_b1[2], PP1[3], C_b1[3]);
   fa b1_a3 (1'b0,        A[3] & B[1], C_b1[3], PP1[4], PP1[5]);

   // module fa (a, b, ci, s, co);
   fa b2_a0 (PP1[2], A[0] & B[2], 1'b0,    P[2],   C_b2[1]);
   fa b2_a1 (PP1[3], A[1] & B[2], C_b2[1], PP2[3], C_b2[2]);
   fa b2_a2 (PP1[4], A[2] & B[2], C_b2[2], PP2[4], C_b2[3]);
   fa b2_a3 (PP1[5], A[3] & B[2], C_b2[3], PP2[5], PP2[6]);

   // module fa (a, b, ci, s, co);
   fa b3_a0 (PP2[3], A[0] & B[3], 1'b0,    P[3], C_b3[1]);
   fa b3_a1 (PP2[4], A[1] & B[3], C_b3[1], P[4], C_b3[2]);
   fa b3_a2 (PP2[5], A[2] & B[3], C_b3[2], P[5], C_b3[3]);
   fa b3_a3 (PP2[6], A[3] & B[3], C_b3[3], P[6], P[7]);

   // drive the display through a 7-seg decoder
   hex7seg digit_7 (4'h0, HEX7);
   hex7seg digit_6 (A, HEX6);

   hex7seg digit_5 (4'h0, HEX5);
   hex7seg digit_4 (B, HEX4);

   hex7seg digit_3 (4'h0, HEX3);
   hex7seg digit_2 (4'h0, HEX2);
   hex7seg digit_1 (P[7:4], HEX1);
   hex7seg digit_0 (P[3:0], HEX0);

endmodule

module fa (a, b, ci, s, co);
   input a, b, ci;
   output s, co;

   wire a_xor_b;

   assign a_xor_b = a ^ b;
   assign s = a_xor_b ^ ci;
   assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
```

```
    output [0:6] display;

    reg [0:6] display;

    /*
     *       0
     *      ---
     *     |   |
     *    5|   |1
     *     | 6 |
     *      ---
     *     |   |
     *    4|   |2
     *     |   |
     *      ---
     *       3
     */
    always @ (hex)
       case (hex)
          4'h0: display = 7'b0000001;
          4'h1: display = 7'b1001111;
          4'h2: display = 7'b0010010;
          4'h3: display = 7'b0000110;
          4'h4: display = 7'b1001100;
          4'h5: display = 7'b0100100;
          4'h6: display = 7'b1100000;
          4'h7: display = 7'b0001111;
          4'h8: display = 7'b0000000;
          4'h9: display = 7'b0001100;
          4'hA: display = 7'b0001000;
          4'hb: display = 7'b1100000;
          4'hC: display = 7'b0110001;
          4'hd: display = 7'b1000010;
          4'hE: display = 7'b0110000;
          4'hF: display = 7'b0111000;
       endcase
endmodule
```

```
// 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
module part2 (SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [7:0] A, B;
   wire [15:0] P;
   wire [7:1] C_b1;  // carries for row that ANDs with B1
   wire [9:2] PP1;   // partial products from row that ANDs with B1
   wire [7:1] C_b2;  // carries for row that ANDs with B2
   wire [10:3] PP2;  // partial products from row that ANDs with B2
   wire [7:1] C_b3;  // carries for row that ANDs with B3
   wire [11:4] PP3;  // partial products from row that ANDs with B3
   wire [7:1] C_b4;  // carries for row that ANDs with B4
   wire [12:5] PP4;  // partial products from row that ANDs with B4
   wire [7:1] C_b5;  // carries for row that ANDs with B5
   wire [13:6] PP5;  // partial products from row that ANDs with B5
   wire [7:1] C_b6;  // carries for row that ANDs with B6
   wire [14:7] PP6;  // partial products from row that ANDs with B6
   wire [7:1] C_b7;  // carries for row that ANDs with B7

   assign A = SW[15:8];
   assign B = SW[7:0];
   assign P[0] = A[0] & B[0];

   // module fa (a, b, ci, s, co);
   fa b1_a0 (A[1] & B[0], A[0] & B[1], 1'b0,    P[1],   C_b1[1]);
   fa b1_a1 (A[2] & B[0], A[1] & B[1], C_b1[1], PP1[2], C_b1[2]);
   fa b1_a2 (A[3] & B[0], A[2] & B[1], C_b1[2], PP1[3], C_b1[3]);
   fa b1_a3 (A[4] & B[0], A[3] & B[1], C_b1[3], PP1[4], C_b1[4]);
   fa b1_a4 (A[5] & B[0], A[4] & B[1], C_b1[4], PP1[5], C_b1[5]);
   fa b1_a5 (A[6] & B[0], A[5] & B[1], C_b1[5], PP1[6], C_b1[6]);
   fa b1_a6 (A[7] & B[0], A[6] & B[1], C_b1[6], PP1[7], C_b1[7]);
   fa b1_a7 (1'b0,        A[7] & B[1], C_b1[7], PP1[8], PP1[9]);

   // module fa (a, b, ci, s, co);
   fa b2_a0 (PP1[2], A[0] & B[2], 1'b0,    P[2],   C_b2[1]);
   fa b2_a1 (PP1[3], A[1] & B[2], C_b2[1], PP2[3], C_b2[2]);
   fa b2_a2 (PP1[4], A[2] & B[2], C_b2[2], PP2[4], C_b2[3]);
   fa b2_a3 (PP1[5], A[3] & B[2], C_b2[3], PP2[5], C_b2[4]);
   fa b2_a4 (PP1[6], A[4] & B[2], C_b2[4], PP2[6], C_b2[5]);
   fa b2_a5 (PP1[7], A[5] & B[2], C_b2[5], PP2[7], C_b2[6]);
   fa b2_a6 (PP1[8], A[6] & B[2], C_b2[6], PP2[8], C_b2[7]);
   fa b2_a7 (PP1[9], A[7] & B[2], C_b2[7], PP2[9], PP2[10]);

   // module fa (a, b, ci, s, co);
   fa b3_a0 (PP2[3],  A[0] & B[3], 1'b0,    P[3],    C_b3[1]);
   fa b3_a1 (PP2[4],  A[1] & B[3], C_b3[1], PP3[4],  C_b3[2]);
   fa b3_a2 (PP2[5],  A[2] & B[3], C_b3[2], PP3[5],  C_b3[3]);
   fa b3_a3 (PP2[6],  A[3] & B[3], C_b3[3], PP3[6],  C_b3[4]);
   fa b3_a4 (PP2[7],  A[4] & B[3], C_b3[4], PP3[7],  C_b3[5]);
   fa b3_a5 (PP2[8],  A[5] & B[3], C_b3[5], PP3[8],  C_b3[6]);
   fa b3_a6 (PP2[9],  A[6] & B[3], C_b3[6], PP3[9],  C_b3[7]);
   fa b3_a7 (PP2[10], A[7] & B[3], C_b3[7], PP3[10], PP3[11]);

   // module fa (a, b, ci, s, co);
   fa b4_a0 (PP3[4],  A[0] & B[4], 1'b0,    P[4],    C_b4[1]);
   fa b4_a1 (PP3[5],  A[1] & B[4], C_b4[1], PP4[5],  C_b4[2]);
   fa b4_a2 (PP3[6],  A[2] & B[4], C_b4[2], PP4[6],  C_b4[3]);
   fa b4_a3 (PP3[7],  A[3] & B[4], C_b4[3], PP4[7],  C_b4[4]);
   fa b4_a4 (PP3[8],  A[4] & B[4], C_b4[4], PP4[8],  C_b4[5]);
   fa b4_a5 (PP3[9],  A[5] & B[4], C_b4[5], PP4[9],  C_b4[6]);
```

```
    fa b4_a6 (PP3[10], A[6] & B[4], C_b4[6], PP4[10], C_b4[7]);
    fa b4_a7 (PP3[11], A[7] & B[4], C_b4[7], PP4[11], PP4[12]);

    // module fa (a, b, ci, s, co);
    fa b5_a0 (PP4[5],  A[0] & B[5], 1'b0,    P[5],    C_b5[1]);
    fa b5_a1 (PP4[6],  A[1] & B[5], C_b5[1], PP5[6],  C_b5[2]);
    fa b5_a2 (PP4[7],  A[2] & B[5], C_b5[2], PP5[7],  C_b5[3]);
    fa b5_a3 (PP4[8],  A[3] & B[5], C_b5[3], PP5[8],  C_b5[4]);
    fa b5_a4 (PP4[9],  A[4] & B[5], C_b5[4], PP5[9],  C_b5[5]);
    fa b5_a5 (PP4[10], A[5] & B[5], C_b5[5], PP5[10], C_b5[6]);
    fa b5_a6 (PP4[11], A[6] & B[5], C_b5[6], PP5[11], C_b5[7]);
    fa b5_a7 (PP4[12], A[7] & B[5], C_b5[7], PP5[12], PP5[13]);

    // module fa (a, b, ci, s, co);
    fa b6_a0 (PP5[6],  A[0] & B[6], 1'b0,    P[6],    C_b6[1]);
    fa b6_a1 (PP5[7],  A[1] & B[6], C_b6[1], PP6[7],  C_b6[2]);
    fa b6_a2 (PP5[8],  A[2] & B[6], C_b6[2], PP6[8],  C_b6[3]);
    fa b6_a3 (PP5[9],  A[3] & B[6], C_b6[3], PP6[9],  C_b6[4]);
    fa b6_a4 (PP5[10], A[4] & B[6], C_b6[4], PP6[10], C_b6[5]);
    fa b6_a5 (PP5[11], A[5] & B[6], C_b6[5], PP6[11], C_b6[6]);
    fa b6_a6 (PP5[12], A[6] & B[6], C_b6[6], PP6[12], C_b6[7]);
    fa b6_a7 (PP5[13], A[7] & B[6], C_b6[7], PP6[13], PP6[14]);

    // module fa (a, b, ci, s, co);
    fa b7_a0 (PP6[7],  A[0] & B[7], 1'b0,    P[7],  C_b7[1]);
    fa b7_a1 (PP6[8],  A[1] & B[7], C_b7[1], P[8],  C_b7[2]);
    fa b7_a2 (PP6[9],  A[2] & B[7], C_b7[2], P[9],  C_b7[3]);
    fa b7_a3 (PP6[10], A[3] & B[7], C_b7[3], P[10], C_b7[4]);
    fa b7_a4 (PP6[11], A[4] & B[7], C_b7[4], P[11], C_b7[5]);
    fa b7_a5 (PP6[12], A[5] & B[7], C_b7[5], P[12], C_b7[6]);
    fa b7_a6 (PP6[13], A[6] & B[7], C_b7[6], P[13], C_b7[7]);
    fa b7_a7 (PP6[14], A[7] & B[7], C_b7[7], P[14], P[15]);

    // drive the display through a 7-seg decoder
    // drive the display through a 7-seg decoder
    hex7seg digit_7 (A[7:4], HEX7);
    hex7seg digit_6 (A[3:0], HEX6);

    hex7seg digit_5 (B[7:4], HEX5);
    hex7seg digit_4 (B[3:0], HEX4);

    hex7seg digit_3 (P[15:12], HEX3);
    hex7seg digit_2 (P[11:8], HEX2);
    hex7seg digit_1 (P[7:4], HEX1);
    hex7seg digit_0 (P[3:0], HEX0);

endmodule

module fa (a, b, ci, s, co);
    input a, b, ci;
    output s, co;

    wire a_xor_b;

    assign a_xor_b = a ^ b;
    assign s = a_xor_b ^ ci;
    assign co = (~a_xor_b & b) | (a_xor_b & ci);
endmodule

module hex7seg (hex, display);
    input [3:0] hex;
    output [0:6] display;

    reg [0:6] display;
```

```
   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
// input two 8-bit numbers using the SW switches and display the numbers
// on the 7-seg digits.  Multiply and display the product on the
// other 7-seg digits
module part3 (SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [7:0] A, B;
   wire [15:0] P;

   assign A = SW[15:8];
   assign B = SW[7:0];

   lpm_mult16 u1 (A, B, P);

   // drive the display through a 7-seg decoder
   hex7seg digit_7 (A[7:4], HEX7);
   hex7seg digit_6 (A[3:0], HEX6);

   hex7seg digit_5 (B[7:4], HEX5);
   hex7seg digit_4 (B[3:0], HEX4);

   hex7seg digit_3 (P[15:12], HEX3);
   hex7seg digit_2 (P[11:8], HEX2);
   hex7seg digit_1 (P[7:4], HEX1);
   hex7seg digit_0 (P[3:0], HEX0);
endmodule

module hex7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
```

```
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

Laboratory Exercise 4

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 Part IV 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. Two styles of Verilog code that can be used to describe this circuit are given in Figure 2. Part a of the figure specifies the latch by instantiating logic gates, and part b uses logic expressions to create the same circuit. 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 3a.

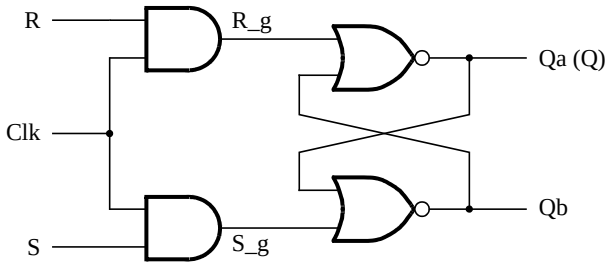


Figure 1. A gated RS latch circuit.

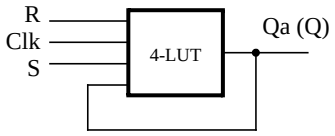
```
// A gated RS latch
module  part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;

    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

    and (R_g, R, Clk);
    and (S_g, S, Clk);
    nor (Qa, R_g, Qb);
    nor (Qb, S_g, Qa);

    assign Q = Qa;

endmodule
```

Figure 2a. Instantiating logic gates for the RS latch.

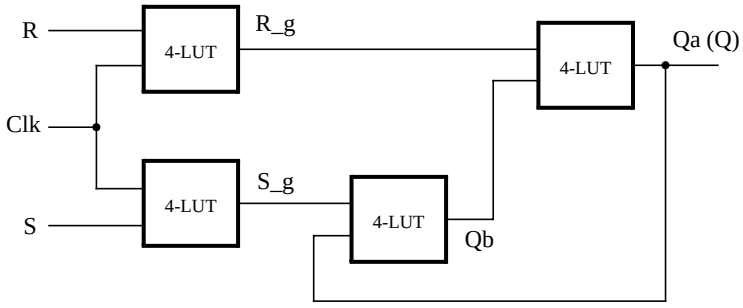
```
// A gated RS latch
module  part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;

    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

    assign R_g = R & Clk;
    assign S_g = S & Clk;
    assign Qa = ~(R_g | Qb);
    assign Qb = ~(S_g | Qa);

    assign Q = Qa;

endmodule
```

Figure 2b. 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 /* synthesis keep */ is included to instruct 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 Verilog file with the code in either part a or b of Figure 2 (both versions of the code should produce the same circuit) 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*, *Qa*, and *Qb*. 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.

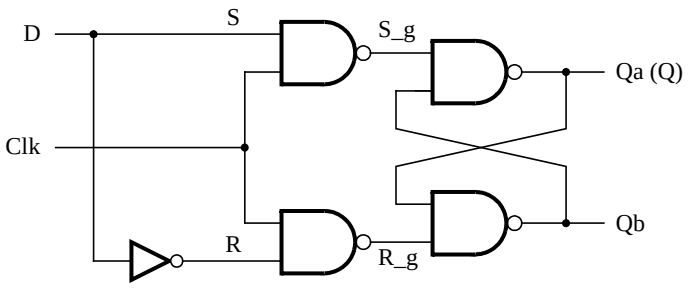


Figure 4. Circuit for a gated D latch.

Perform the following steps:

1. Create a new Quartus II project. Generate a Verilog file using the style of code in Figure 2b for the gated D latch. Use the /* synthesis 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 module that contains the appropriate input and output ports (pins) for the DE2 board. Instantiate your latch in this top-level module. 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.

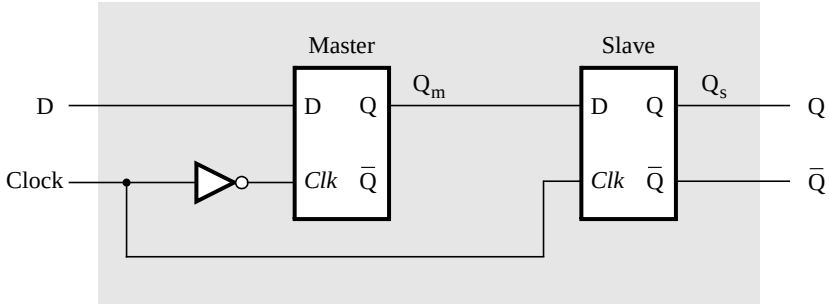


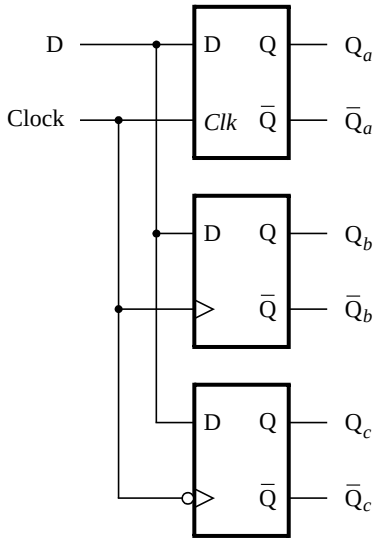
Figure 5. Circuit for a master-slave D flip-flop.

Perform the following:

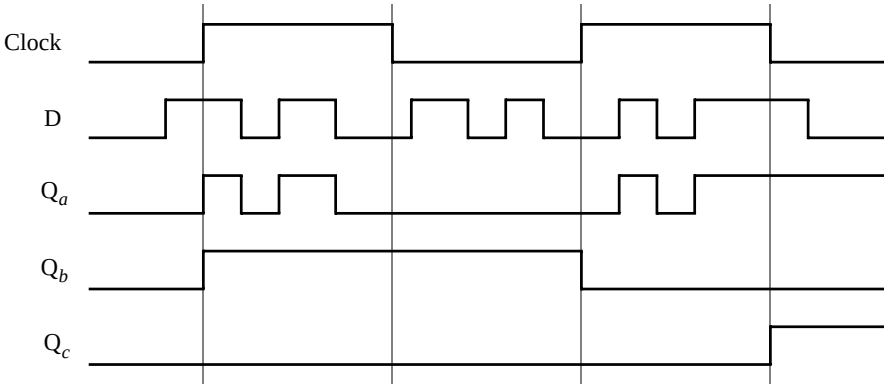
1. Create a new Quartus II project. Generate a Verilog file that instantiates two copies of your gated D latch module from Part II to implement the master-slave flip-flop.
2. Include in your project the appropriate input and output ports for the Altera DE2 board. Use switch SW_0 to drive the D input 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 Quartus II project.
2. Write a Verilog file that instantiates the three storage elements. For this part you should no longer use the /* synthesis keep */ directive from Parts I to III. Figure 7 gives a behavioral style of Verilog 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.

```
module D_latch  (D, Clk, Q);
   input D, Clk;
   output reg Q;

   always @ (D, Clk)
      if (Clk)
         Q = D;
endmodule
```

Figure 7. A behavioral style of Verilog 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 Verilog file that provides the necessary functionality. Use KEY_0 as an active-low asynchronous reset, and use KEY_1 as a clock input.
3. Include the Verilog file in your project and compile the circuit.
4. 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.
5. Recompile the circuit and download it into the FPGA chip.
6. Test the functionality of your design by toggling the switches and observing the output displays.

```
// A gated RS latch described the hard way
module part1 (Clk, R, S, Q);
   input Clk, R, S;
   output Q;

   wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

   and (R_g, R, Clk);
   and (S_g, S, Clk);
   nor (Qa, R_g, Qb);
   nor (Qb, S_g, Qa);

   assign Q = Qa;

endmodule
```

```
// A gated RS latch described the hard way
module part1 (Clk, R, S, Q);
   input Clk, R, S;
   output Q;

   wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

   assign R_g = R & Clk;
   assign S_g = S & Clk;
   assign Qa = ~(R_g | Qb);
   assign Qb = ~(S_g | Qa);

   assign Q = Qa;

endmodule
```

```
// A gated D latch described the hard way
module D_latch (Clk, D, Q);
   input Clk, D;
   output Q;

   wire R, S_g, R_g, Qa, Qb /* synthesis keep */ ;

   assign R = ~D;
   assign S_g = ~(D & Clk);
   assign R_g = ~(R & Clk);
   assign Qa = ~(S_g & Qb);
   assign Qb = ~(R_g & Qa);

   assign Q = Qa;

endmodule
```

```
// SW[0] is the latch's D input, SW[1] is the level-sensitive Clk, LEDR[0] is Q
module top (input [1:0] SW, output [0:0] LEDR);

   // module D_latch (input Clk, D, output Q);
   D_latch U1 (SW[1], SW[0], LEDR[0]);

endmodule

// A gated D latch described the hard way
module D_latch (input Clk, D, output Q);

   wire S, R, S_g, R_g, Qa, Qb /* synthesis keep */ ;

   assign S = D;
   assign R = ~D;
   assign S_g = ~(S & Clk);
   assign R_g = ~(R & Clk);
   assign Qa = ~(S_g & Qb);
   assign Qb = ~(R_g & Qa);

   assign Q = Qa;

endmodule
```

```
// SW[0] is the flip-flop's D input, SW[1] is the edge-sensitive Clock, LEDR[0] is Q
module part3 (input [1:0] SW, output [0:0] LEDR);

   wire Qm, Qs;
   // module D_latch (input Clk, D, output Q);
   D_latch U1 (~SW[1], SW[0], Qm);
   D_latch U2 (SW[1], Qm, Qs);

   assign LEDR[0] = Qs;

endmodule

// A D latch described the hard way
module D_latch (input Clk, D, output Q);

   wire S, R, S_g, R_g, Qa, Qb /* synthesis keep */ ;

   assign S = D;
   assign R = ~D;
   assign S_g = ~(S & Clk);
   assign R_g = ~(R & Clk);
   assign Qa = ~(S_g & Qb);
   assign Qb = ~(R_g & Qa);

   assign Q = Qa;

endmodule
```

```
// 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
module part4 (Clk, D, Qa, Qb, Qc);
   input Clk, D;
   output reg Qa, Qb, Qc;

   // gated D-latch
   always @( * )
      if (Clk == 1'b1)
         Qa = D;

   // positive-edge triggered D FF
   always @(posedge Clk)
      Qb <= D;

   // negative-edge triggered D FF
   always @(negedge Clk)
      Qc <= D;
endmodule
```

```
// KEY0 is resetn, KEY1 is the clock for reg_A
module part5 (SW, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [15:0] SW;
   input [3:0] KEY;     // Used for reset and enable for A_reg
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [15:0] A, B;

   // module regne (R, Clock, Resetn, En, Q);
   regne A_reg (SW, KEY[1], KEY[0], A);
   assign B = SW;

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

   hex7seg digit_3 (B[15:12], HEX3);
   hex7seg digit_2 (B[11:8], HEX2);
   hex7seg digit_1 (B[7:4], HEX1);
   hex7seg digit_0 (B[3:0], HEX0);
endmodule

module regne (R, Clock, Resetn, Q);
   parameter n = 16;
   input [n-1:0] R;
   input Clock, Resetn;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else
         Q <= R;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
```

```
            4'h5: display = 7'b0100100;
            4'h6: display = 7'b1100000;
            4'h7: display = 7'b0001111;
            4'h8: display = 7'b0000000;
            4'h9: display = 7'b0001100;
            4'hA: display = 7'b0001000;
            4'hb: display = 7'b1100000;
            4'hC: display = 7'b0110001;
            4'hd: display = 7'b1000010;
            4'hE: display = 7'b0110000;
            4'hF: display = 7'b0111000;
        endcase
endmodule
```

Laboratory Exercise 5

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.

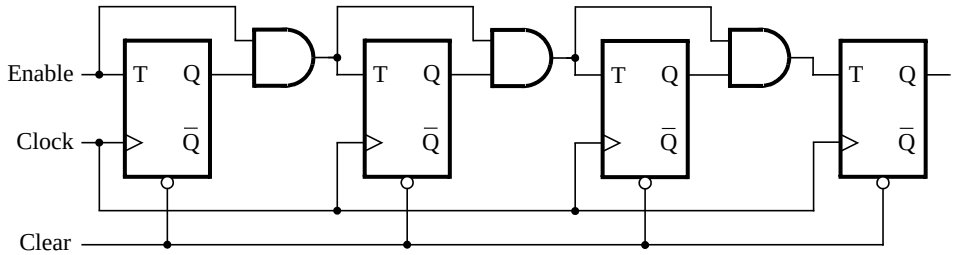


Figure 1. A 4-bit counter.

1. Write a Verilog file that defines a 16-bit counter by using the structure depicted in Figure 1. Your code should include a T flip-flop module that is instantiated 16 times to create the counter. 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 Verilog 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 needed to implement the circuit on the DE2 board, and compile the circuit.
4. Download your circuit into the FPGA chip 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 1?

Part II

Simplify your Verilog code so that the counter specification is based on the Verilog 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.

```
//
// inputs:
// KEY0: manual clock
// SW0: active low reset
// SW1: enable signal for the counter
//
// outputs:
// HEX0 - HEX3: hex segment displays
module part1 (SW, KEY, HEX3, HEX2, HEX1, HEX0);
   input [1:0] SW ;
   input [0:0] KEY ;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   wire Clock = KEY[0];
   wire Resetn = SW[0];

   // 16-bit counter based on T-flip flops
   wire [15:0] Count;
   wire [15:0] Enable;

   assign Enable[0] = SW[1];
   ToggleFF (Enable[0], Clock, Resetn, Count[0]);
   assign Enable[1] = Count[0] & Enable[0];
   ToggleFF (Enable[1], Clock, Resetn, Count[1]);
   assign Enable[2] = Count[1] & Enable[1];
   ToggleFF (Enable[2], Clock, Resetn, Count[2]);
   assign Enable[3] = Count[2] & Enable[2];
   ToggleFF (Enable[3], Clock, Resetn, Count[3]);
   assign Enable[4] = Count[3] & Enable[3];
   ToggleFF (Enable[4], Clock, Resetn, Count[4]);
   assign Enable[5] = Count[4] & Enable[4];
   ToggleFF (Enable[5], Clock, Resetn, Count[5]);
   assign Enable[6] = Count[5] & Enable[5];
   ToggleFF (Enable[6], Clock, Resetn, Count[6]);
   assign Enable[7] = Count[6] & Enable[6];
   ToggleFF (Enable[7], Clock, Resetn, Count[7]);
   assign Enable[8] = Count[7] & Enable[7];
   ToggleFF (Enable[8], Clock, Resetn, Count[8]);
   assign Enable[9] = Count[8] & Enable[8];
   ToggleFF (Enable[9], Clock, Resetn, Count[9]);
   assign Enable[10] = Count[9] & Enable[9];
   ToggleFF (Enable[10], Clock, Resetn, Count[10]);
   assign Enable[11] = Count[10] & Enable[10];
   ToggleFF (Enable[11], Clock, Resetn, Count[11]);
   assign Enable[12] = Count[11] & Enable[11];
   ToggleFF (Enable[12], Clock, Resetn, Count[12]);
   assign Enable[13] = Count[12] & Enable[12];
   ToggleFF (Enable[13], Clock, Resetn, Count[13]);
   assign Enable[14] = Count[13] & Enable[13];
   ToggleFF (Enable[14], Clock, Resetn, Count[14]);
   assign Enable[15] = Count[14] & Enable[14];
   ToggleFF (Enable[15], Clock, Resetn, Count[15]);

   // drive the displays
   hex7seg digit3 (Count[15:12], HEX3);
   hex7seg digit2 (Count[11:8], HEX2);
   hex7seg digit1 (Count[7:4], HEX1);
   hex7seg digit0 (Count[3:0], HEX0);
endmodule

module ToggleFF(T, Clock, Resetn, Q);
   input T, Clock, Resetn;
   output reg Q;
```

```
   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         Q <= 1'b0;
      else if(T)
         Q <= ~Q;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *        0
    *       ---
    *      |   |
    *    5|   |1
    *      | 6 |
    *       ---
    *      |   |
    *    4|   |2
    *      |   |
    *       ---
    *        3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
//
// inputs:
// KEY0: manual clock
// SW0: active low reset
// SW1: enable signal for the counter
//
// outputs:
// HEX0: hex segment display
module part1 (SW, KEY, HEX0);
   input [1:0] SW ;
   input [0:0] KEY ;
   output [0:6] HEX0;

   wire Clock = KEY[0];
   wire Resetn = SW[0];

   // 4-bit counter based on T-flip flops
   wire [3:0] Count;
   wire [3:0] Enable;

   assign Enable[0] = SW[1];
   ToggleFF (Enable[0], Clock, Resetn, Count[0]);
   assign Enable[1] = Count[0] & Enable[0];
   ToggleFF (Enable[1], Clock, Resetn, Count[1]);
   assign Enable[2] = Count[1] & Enable[1];
   ToggleFF (Enable[2], Clock, Resetn, Count[2]);
   assign Enable[3] = Count[2] & Enable[2];
   ToggleFF (Enable[3], Clock, Resetn, Count[3]);

   // drive the displays
   hex7seg digit0 (Count[3:0], HEX0);
endmodule

module ToggleFF(T, Clock, Resetn, Q);
   input T, Clock, Resetn;
   output reg Q;

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         Q <= 1'b0;
      else if(T)
         Q <= ~Q;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
```

```
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
//
// inputs:
// KEY0: manual clock
// SW0: active low reset
// SW1: enable signal for the counter
//
// outputs:
// HEX0 - HEX3: hex segment displays
module part2 (SW, KEY, HEX3, HEX2, HEX1, HEX0);
   input [1:0] SW ;
   input [0:0] KEY ;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   wire Clock = KEY[0];
   wire Resetn = SW[0];
   wire Enable = SW[1];

   // 16-bit counter
   reg [15:0] Count;
   always @(posedge Clock)
      if (!Resetn)
         Count <= 0;
      else if (Enable)
         Count <= Count + 1'b1;

   // drive the displays
   hex7seg digit3 (Count[15:12], HEX3);
   hex7seg digit2 (Count[11:8], HEX2);
   hex7seg digit1 (Count[7:4], HEX1);
   hex7seg digit0 (Count[3:0], HEX0);
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
```

```
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
//
// inputs:
// KEY0: manual clock
// SW0: active low reset
// SW1: enable signal for the counter
//
// outputs:
// HEX0 - HEX3: hex segment displays
module part2 (SW, KEY, HEX0);
   input [1:0] SW ;
   input [0:0] KEY ;
   output [0:6] HEX0;

   wire Clock = KEY[0];
   wire Resetn = SW[0];
   wire Enable = SW[1];

   // 4-bit counter
   reg [3:0] Count;
   always @(posedge Clock)
      if (!Resetn)
         Count <= 0;
      else if (Enable)
         Count <= Count + 1'b1;

   // drive the displays
   hex7seg digit0 (Count[3:0], HEX0);
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
```

```
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
//
// inputs:
// KEY0: manual clock
// SW0: active low reset
// SW1: enable signal for the counter
//
// outputs:
// HEX0 - HEX3: hex segment displays
module part3 (SW, KEY, HEX3, HEX2, HEX1, HEX0);
   input [1:0] SW ;
   input [0:0] KEY ;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   wire Clock = KEY[0];
   wire Resetn = SW[0];
   wire Enable = SW[1];

   wire [15:0] Count;
   // 16-bit counter
   // module lpm_count (clock, cnt_en, sclr, q);
   lpm_count U1 (Clock, Enable, ~Resetn, Count);

   // drive the displays
   hex7seg digit3 (Count[15:12], HEX3);
   hex7seg digit2 (Count[11:8], HEX2);
   hex7seg digit1 (Count[7:4], HEX1);
   hex7seg digit0 (Count[3:0], HEX0);
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
```

```
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
// uses a 1-digit bcd counter enabled at 1Hz
module part4 (Clock, HEX0);
   input Clock;
   output [0:6] HEX0;

   wire [3:0] bcd;
   parameter m = 25;
   reg [m-1:0] slow_count;

   reg[3:0] digit_flipper;

   // Create a 1Hz 4-bit counter

   // A large counter to produce a 1 second (approx) enable from the 50 MHz Clock
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // four-bit counter that uses a slow enable for selecting digit
   always @ (posedge Clock)
      if (slow_count == 0)
         if (digit_flipper == 4'h9)
            digit_flipper <= 4'h0;
         else
            digit_flipper <= digit_flipper + 1'b1;

   assign bcd = digit_flipper;
   // drive the display through a 7-seg decoder
   bcd7seg digit_0 (bcd, HEX0);

endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'bx;
      endcase
```

```
endmodule
```

```
// uses a shift register enabled at 1Hz and a MUX. To start the circuit's
// operation, the shift register must be parallel loaded by pressing KEY[3]
module part4 (KEY, Clock, HEX0, LEDR);
   input [3:0] KEY;
   input Clock;
   output [0:6] HEX0;
   output [9:0] LEDR;

   wire [3:0] bcd;
   wire shift_enable;
   parameter m = 25;
   reg [m-1:0] slow_count;
   reg [3:0] digit_flipper;

   reg[9:0] rotate;

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   assign shift_enable = (slow_count == 0);
   // shift register with enable
   integer k;
   always @(posedge Clock)
   begin
      if (KEY[3] == 0)
         rotate <= 10'b0000000001;
      else if (shift_enable)
      begin
         for (k = 0; k < 9; k = k+1)
            rotate[k+1] <= rotate[k];
         rotate[0] <= rotate[k];
      end
   end

   assign LEDR = rotate;

   always @ (rotate)
      case (rotate)
         10'b0000000001:      digit_flipper = 4'h0;
         10'b0000000010:      digit_flipper = 4'h1;
         10'b0000000100:      digit_flipper = 4'h2;
         10'b0000001000:      digit_flipper = 4'h3;
         10'b0000010000:      digit_flipper = 4'h4;
         10'b0000100000:      digit_flipper = 4'h5;
         10'b0001000000:      digit_flipper = 4'h6;
         10'b0010000000:      digit_flipper = 4'h7;
         10'b0100000000:      digit_flipper = 4'h8;
         10'b1000000000:      digit_flipper = 4'h9;
         default:       digit_flipper = 4'bx;
      endcase

   assign bcd = digit_flipper;
   // drive the display through a 7-seg decoder
   bcd7seg digit_0 (bcd, HEX0);

endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;
```

```
   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'bx;
      endcase
endmodule
```

```
// scans the word HELLO across the 7-seg displays. KEY[3] causes a reset.
module part5 (Clock, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [3:0] KEY;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [2:0] seg7_7, seg7_6, seg7_5, seg7_4, seg7_3, seg7_2, seg7_1, seg7_0;
   parameter m = 24;
   reg [m-1:0] slow_count;

   reg[2:0] digit_flipper;

   // Create a 1Hz 4-bit counter

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // 3-bit counter that uses a slow enable for selecting digit
   always @ (posedge Clock)
      if (KEY[3] == 0)
         digit_flipper <= 3'b000;
      else if (slow_count == 0)
         digit_flipper <= digit_flipper + 1'b1;

   assign seg7_7 = digit_flipper;
   assign seg7_6 = digit_flipper + 3'b001;
   assign seg7_5 = digit_flipper + 3'b010;
   assign seg7_4 = digit_flipper + 3'b011;
   assign seg7_3 = digit_flipper + 3'b100;
   assign seg7_2 = digit_flipper + 3'b101;
   assign seg7_1 = digit_flipper + 3'b110;
   assign seg7_0 = digit_flipper + 3'b111;

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

module hello7seg (char, display);
   input [2:0] char;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
```

```
    */
   always @ (char)
      case (char)
         3'h0: display = 7'b1001000;      // 'H'
         3'h1: display = 7'b0110000;      // 'E'
         3'h2: display = 7'b1110001;      // 'L'
         3'h3: display = 7'b1110001;      // 'L'
         3'h4: display = 7'b0000001;      // 'O'
         3'h5: display = 7'b1111111;      // ' '
         3'h6: display = 7'b1111111;      // ' '
         3'h7: display = 7'b1111111;      // ' '
      endcase
endmodule
```

```
// scans the word HELLO across the 7-seg displays. KEY[3] causes a reset,
// and KEY[0] initializes the counters that select the characters in HELLO.
module part3 (Clock, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [3:0] KEY;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [3:0] seg7_7, seg7_6, seg7_5, seg7_4, seg7_3, seg7_2, seg7_1, seg7_0;
   wire Enable;
   parameter m = 24;
   reg [m-1:0] slow_count;

   // Create a 1Hz 4-bit counter

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   assign Enable = (slow_count == 0);
   // module upcount (R, Resetn, Clock, L, E, Q);
   upcount (3'b000, KEY[3], Clock, !KEY[0], Enable, seg7_7);
   upcount (3'b001, KEY[3], Clock, !KEY[0], Enable, seg7_6);
   upcount (3'b010, KEY[3], Clock, !KEY[0], Enable, seg7_5);
   upcount (3'b011, KEY[3], Clock, !KEY[0], Enable, seg7_4);
   upcount (3'b100, KEY[3], Clock, !KEY[0], Enable, seg7_3);
   upcount (3'b101, KEY[3], Clock, !KEY[0], Enable, seg7_2);
   upcount (3'b110, KEY[3], Clock, !KEY[0], Enable, seg7_1);
   upcount (3'b111, KEY[3], Clock, !KEY[0], Enable, seg7_0);

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

endmodule

// three-bit counter with parallel load and enable
module upcount (R, Resetn, Clock, L, E, Q);
   input [2:0] R;
   input Resetn, Clock, L, E;
   output [2:0] Q;
   reg [2:0] Q;

   always @ (posedge Clock)
      if (Resetn == 0)
         Q <= 3'b000;
      else if (L)
         Q <= R;
      else if (E)
         Q <= Q + 3'b001;
endmodule

module hello7seg (char, display);
   input [2:0] char;
   output [0:6] display;

   reg [0:6] display;
```

```
   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (char)
      case (char)
         3'h0: display = 7'b1001000;      // 'H'
         3'h1: display = 7'b0110000;      // 'E'
         3'h2: display = 7'b1110001;      // 'L'
         3'h3: display = 7'b1110001;      // 'L'
         3'h4: display = 7'b1000000;      // 'O'
         3'h5: display = 7'b1111111;      // ' '
         3'h6: display = 7'b1111111;      // ' '
         3'h7: display = 7'b1111111;      // ' '
      endcase
endmodule
```

Laboratory Exercise 6

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 Verilog file that specifies the desired circuit.
3. Include the Verilog 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*.

```
// 3-digit BCD counter.
module part1 (Clock, KEY, HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [3:0] KEY;
   output [0:6] HEX3, HEX2, HEX1, HEX0;

   parameter m = 25;
   reg [m-1:0] slow_count;

   reg[3:0] bcd_0, bcd_1, bcd_2;

   // Create a 1Hz 4-bit counter

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // 3-digit BCD counter that uses a slow enable
   always @ (posedge Clock)
      if (KEY[3] == 0)
      begin
         bcd_0 <= 4'h0;
         bcd_1 <= 4'h0;
         bcd_2 <= 4'h0;
      end
      else if (slow_count == 0)
      begin
         if (bcd_0 == 4'h9)
         begin
            bcd_0 <= 4'h0;
            if (bcd_1 == 4'h9)
            begin
               bcd_1 <= 4'h0;
               if (bcd_2 == 4'h9)
               begin
                  bcd_2 <= 4'h0;
               end
               else
               begin
                  bcd_2 <= bcd_2 + 1'b1;
               end
            end
            else
            begin
               bcd_1 <= bcd_1 + 1'b1;
            end
         end
         else
         begin
            bcd_0 <= bcd_0 + 1'b1;
         end
      end

   // drive the displays
   bcd7seg digit2 (bcd_2, HEX2);
   bcd7seg digit1 (bcd_1, HEX1);
   bcd7seg digit0 (bcd_0, HEX0);
   // blank the adjacent display
   bcd7seg digit3 (4'hF, HEX3);

endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
```

```
    output [0:6] display;

    reg [0:6] display;

    /*
     *       0
     *      ---
     *     |   |
     *    5|   |1
     *     | 6 |
     *      ---
     *     |   |
     *    4|   |2
     *     |   |
     *      ---
     *       3
     */
    always @ (bcd)
       case (bcd)
          4'h0: display = 7'b0000001;
          4'h1: display = 7'b1001111;
          4'h2: display = 7'b0010010;
          4'h3: display = 7'b0000110;
          4'h4: display = 7'b1001100;
          4'h5: display = 7'b0100100;
          4'h6: display = 7'b1100000;
          4'h7: display = 7'b0001111;
          4'h8: display = 7'b0000000;
          4'h9: display = 7'b0001100;
          default: display = 7'b1111111;
       endcase
endmodule
```

```
// Real time settable clock. Set SW[15:8] switches to 2-digit BCD number
// representing hours. Set SW[7:0] to 2-digit number representing minutes.
// Load initial time by pressing KEY[0].
module part2 (Clock, SW, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [15:0] SW;
   input [3:0] KEY;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   parameter m = 25;
   reg [m-1:0] slow_count;

   reg[3:0] hr_1, hr_0, min_1, min_0, sec_1, sec_0;
   wire E_sec_0, E_sec_1, E_min_0, E_min_1, E_hr_0, E_hr_1;
   wire [3:0] mod_hr_0;    // used to change hour (LSB) from 19 to 20 or 23 to 00

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // 6-digit clock display
   // module mod (R, M, Clock, Resetn, L, E, Q)
   assign E_sec_0 = (slow_count == 0);
   mod seconds_0 (4'h0, 4'h9, Clock, KEY[3], 1'b0, E_sec_0, sec_0);
   // module mod5 (R, Clock, Resetn, L, E, Q)
   assign E_sec_1 = (sec_0 == 9) && E_sec_0;
   mod seconds_1 (4'h0, 4'h5, Clock, KEY[3], 1'b0, E_sec_1, sec_1);

   assign E_min_0 = (sec_1 == 5) && E_sec_1;
   mod minutes_0 (SW[3:0], 4'h9, Clock, KEY[3], ~KEY[0], E_min_0, min_0);
   assign E_min_1 = (min_0 == 9) && E_min_0;
   mod minutes_1 (SW[7:4], 4'h5, Clock, KEY[3], ~KEY[0], E_min_1, min_1);

   assign E_hr_0 = (min_1 == 5) && E_min_1;
   assign mod_hr_0 = (hr_1 == 4'h2) ? 4'h3 : 4'h9;
   mod hour_0 (SW[11:8], mod_hr_0, Clock, KEY[3], ~KEY[0], E_hr_0, hr_0);
   assign E_hr_1 = ( ((hr_1 == 4'h2) && (hr_0 == 4'h3)) || (hr_0 == 9) ) && E_hr_0;
   mod hour_1 (SW[15:12], 4'h2, Clock, KEY[3], ~KEY[0], E_hr_1, hr_1);

   // drive the displays
   bcd7seg digit7 (hr_1, HEX7);
   bcd7seg digit6 (hr_0, HEX6);
   bcd7seg digit5 (min_1, HEX5);
   bcd7seg digit4 (min_0, HEX4);
   bcd7seg digit3 (sec_1, HEX3);
   bcd7seg digit2 (sec_0, HEX2);
   // blank the adjacent display
   bcd7seg digit1 (4'hF, HEX1);
   bcd7seg digit0 (4'hF, HEX0);

endmodule

module mod (R, M, Clock, Resetn, L, E, Q);
   input [3:0] R, M;
   input Clock, Resetn, L, E;
   output [3:0] Q;
   reg [3:0] Q;

   always @ (posedge Clock)
   begin
      if (Resetn == 0)
         Q <= 4'h0;
      else if (L)
         Q <= R;
```

```
      else if (E)
         if (Q == M)
            Q <= 4'h0;
         else
            Q <= Q + 1'b1;
   end
endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'b1111111;
      endcase
endmodule
```

```
// Real time settable clock. Set SW[15:8] switches to 2-digit BCD number
// representing hours. Set SW[7:0] to 2-digit number representing minutes.
// Load initial time by pressing KEY[3].
module part2 (Clock, SW, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [15:0] SW;
   input [3:0] KEY;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   parameter m = 25;
   reg [m-1:0] slow_count;

   reg[3:0] hr_1, hr_0, min_1, min_0, sec_1, sec_0;

   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // 6-digit clock display
   always @ (posedge Clock)
      if (KEY[3] == 0)
      begin
         hr_1 <= SW[15:12];
         hr_0 <= SW[11:8];
         min_1 <= SW[7:4];
         min_0 <= SW[3:0];
      end
      else
         if (slow_count == 0)
         begin
            if (sec_0 == 4'h9)
            begin
               sec_0 <= 4'h0;
               if (sec_1 == 4'h5)
               begin
                  sec_1 <= 4'h0;
                  if (min_0 == 4'h9)
                  begin
                     min_0 <= 4'h0;
                     if (min_1 == 4'h5)
                     begin
                        min_1 <= 4'h0;
                        if (hr_0 == 4'h3)
                        begin
                           if (hr_1 == 4'h2)
                           begin
                              hr_0 <= 4'h0;
                              hr_1 <= 4'h0;
                           end
                           else
                           begin
                              hr_0 <= hr_0 + 1'b1;
                           end
                        end
                        else if (hr_0 == 4'h9)
                        begin
                           hr_0 <= 4'h0;
                           hr_1 <= hr_1 + 1'b1;
                        end
                        else
                        begin
                           hr_0 <= hr_0 + 1'b1;
                        end
                     end
```

```
                    else
                    begin
                       min_1 <= min_1 + 1'b1;
                    end
                 end
                 else
                 begin
                    min_0 <= min_0 + 1'b1;
                 end
              end
              else
              begin
                 sec_1 <= sec_1 + 1'b1;
              end
           end
           else
           begin
              sec_0 <= sec_0 + 1'b1;
           end
        end

   // drive the displays
   bcd7seg digit7 (hr_1, HEX7);
   bcd7seg digit6 (hr_0, HEX6);
   bcd7seg digit5 (min_1, HEX5);
   bcd7seg digit4 (min_0, HEX4);
   bcd7seg digit3 (sec_1, HEX3);
   bcd7seg digit2 (sec_0, HEX2);
   // blank the adjacent display
   bcd7seg digit1 (4'hF, HEX1);
   bcd7seg digit0 (4'hF, HEX0);

endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
```

```
         4'h9: display = 7'b0001100;
         default: display = 7'b1111111;
      endcase
endmodule
```

```
// Press KEY[0] to reset. After a delay (#seconds set by the SW[7:0]
// switches), LEDR[0] turns on and the timer starts. Stop the timer by
// pressing KEY[3]
module part3 (Clock, SW, KEY, LEDR, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0);
   input Clock;
   input [7:0] SW;
   input [3:0] KEY;
   output [0:0] LEDR;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   parameter m = 16;
   reg [m-1:0] slow_count;

   parameter k = 25;
   reg [k-1:0] second;

   wire start;
   reg[7:0] pseudo;
   reg[3:0] sec_0, sec_1, min_0, min_1;

   regne #(.n(1)) reg_start (1'b1, Clock, KEY[3] & KEY[0], (pseudo == 0), start);
   assign LEDR[0] = start;

   // A large counter to produce a 1 sec second (approx) enable
   always @(posedge Clock)
      second <= second + 1'b1;

   // A large counter to produce a 1 msec second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // Pseudo random delay counter--loads SW switches and counts down
   always @(posedge Clock)
      if ( (KEY[3] && KEY[0]) == 0)
         pseudo <= SW;
      else if (second == 0)
         pseudo <= pseudo - 1'b1;

   // 4-digit time counter that uses a slow enable
   always @ (posedge Clock or negedge KEY[0])
      if (KEY[0] == 0)
      begin
         sec_0 <= 4'b0;
         sec_1 <= 4'b0;
         min_0 <= 4'b0;
         min_1 <= 4'b0;
      end
      else if ( (slow_count == 0) && start)
         begin
            if (sec_0 == 4'h9)
            begin
               sec_0 <= 4'h0;
               if (sec_1 == 4'h9)
               begin
                  sec_1 <= 4'h0;
                  if (min_0 == 4'h9)
                  begin
                     min_0 <= 4'h0;
                     if (min_1 == 4'h9)
                     begin
                        min_1 <= 4'h0;
                     end
                     else
```

```
                    begin
                       min_1 <= min_1 + 1'b1;
                    end
                 end
                 else
                 begin
                    min_0 <= min_0 + 1'b1;
                 end
              end
              else
              begin
                 sec_1 <= sec_1 + 1'b1;
              end
           end
           else
           begin
              sec_0 <= sec_0 + 1'b1;
           end
        end

   // drive the displays
   // blank the unused displays
   bcd7seg digit7 (4'hF, HEX7);
   bcd7seg digit6 (4'hF, HEX6);
   bcd7seg digit5 (4'hF, HEX5);
   bcd7seg digit4 (4'hF, HEX4);

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

endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 4;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E == 1)
         Q <= R;
endmodule

module bcd7seg (bcd, display);
   input [3:0] bcd;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
```

```
    *      ---
    *       3
    */
   always @ (bcd)
      case (bcd)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         default: display = 7'b1111111;
      endcase
endmodule
```

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*.

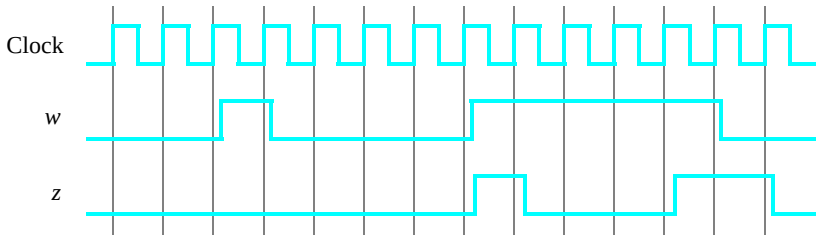


Figure 1. Required timing for the output *z*.

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.

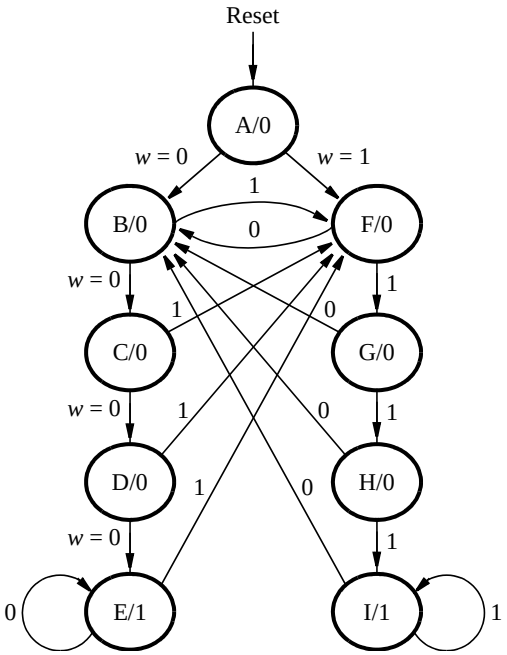


Figure 2. A state diagram for the FSM.

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 Verilog 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 **assign** statements in your Verilog 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 Verilog 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 Verilog 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 Verilog 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 Verilog **case** statement in an **always** block, and use another **always** block to instantiate the state flip-flops. You can use a third **always** block or simple assignment statements to specify the output *z*. To implement the FSM, use four state flip-flops $y_3, \ldots, y_0$ and binary codes, as shown in Table 3.

Name	State Code $y_3y_2y_1y_0$
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000

Table 3. Binary codes for the FSM.

A suggested skeleton of the Verilog code is given in Figure 3.

```
module  part2 ( . . . );
    . . . define input and output ports

    . . . define signals
    reg [3:0] y_Q, Y_D;      // y_Q represents current state, Y_D represents next state
    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
        F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;

    always @(w, y_Q)
    begin: state_table
        case (y_Q)
            A: if (!w) Y_D = B;
                else Y_D = F;
            . . . remainder of state table
            default: Y_D = 4'bxxxx;
        endcase
    end // state_table

    always @(posedge Clock)
    begin: state_FFs
        . . .
    end // state_FFS

    . . . assignments for output z and the LEDs
endmodule
```

Figure 3. Skeleton Verilog 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 Verilog 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 assign the state flip-flop outputs to the red LEDs $LEDR_3$ to $LEDR_0$. 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 necessary to explicitly tell the Synthesis tool in Quartus II that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code. If you do not explicitly give this setting to Quartus II, the Synthesis tool will automatically use a state assignment of its own choosing, and it will ignore the state codes specified in your Verilog code. To make this setting, 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 User-Encoded.
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 Verilog code. To see the result of removing 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 User-Encoded 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.

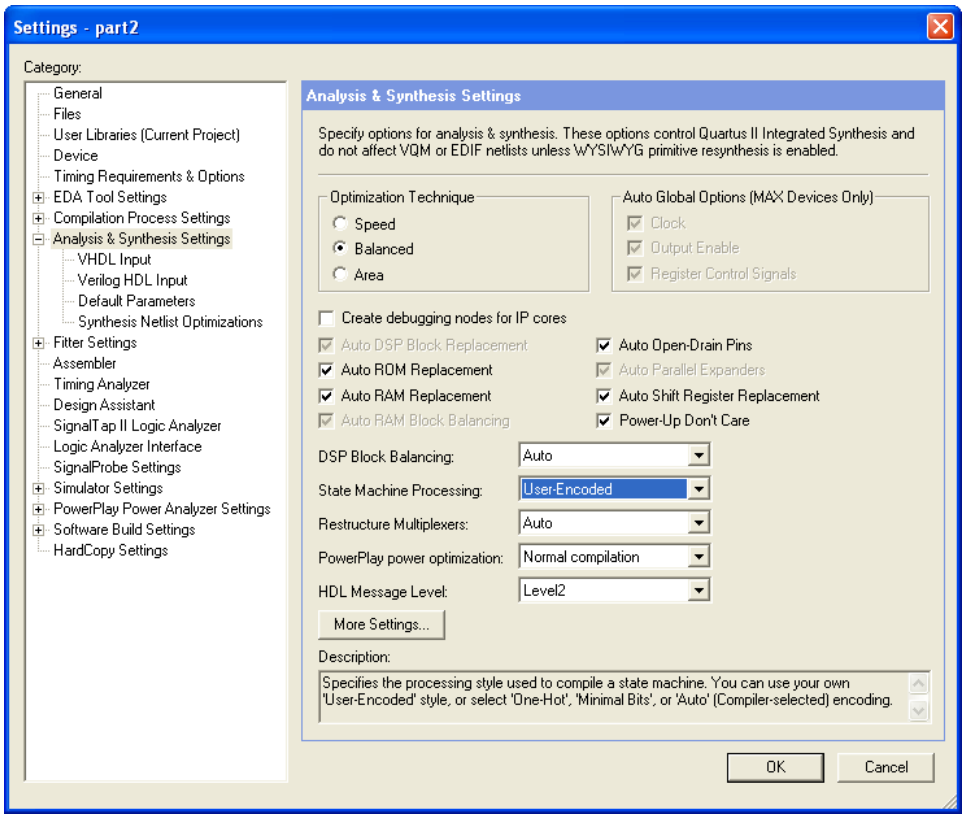


Figure 4. Specifying the state assignment method in Quartus II.

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 Verilog 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 Verilog file that defines the circuit. Use the style of code indicated in Figure 3 for your FSM.
3. Include the Verilog 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 Verilog 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 Verilog code in the style shown in Figure 3 for your finite state machine.

Compile your Verilog 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 Verilog 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 Verilog 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 Verilog 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 Verilog 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.

```
// 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
module part1 (SW, KEY, LEDG, LEDR);
   input [1:0] SW;
   input [0:0] KEY;
   output [0:0] LEDG;
   output [8:0] LEDR;

   wire Clock, Resetn, w, z;
   wire [8:0] y_Q, Y_D;

   assign Clock = KEY[0];
   assign Resetn = SW[0];
   assign w = SW[1];

   assign Y_D[0] = 1'b0;
   flipflop (Y_D[0], Clock, 1'b1, Resetn, y_Q[0]);
   assign Y_D[1] = (y_Q[0] | y_Q[5] | y_Q[6] | y_Q[7] | y_Q[8]) & ~w;
   flipflop (Y_D[1], Clock, Resetn, 1'b1, y_Q[1]);
   assign Y_D[2] = y_Q[1] & ~w;
   flipflop (Y_D[2], Clock, Resetn, 1'b1, y_Q[2]);
   assign Y_D[3] = y_Q[2] & ~w;
   flipflop (Y_D[3], Clock, Resetn, 1'b1, y_Q[3]);
   assign Y_D[4] = (y_Q[3] | y_Q[4]) & ~w;
   flipflop (Y_D[4], Clock, Resetn, 1'b1, y_Q[4]);

   assign Y_D[5] = (y_Q[0] | y_Q[1] | y_Q[2] | y_Q[3] | y_Q[4]) & w;
   flipflop (Y_D[5], Clock, Resetn, 1'b1, y_Q[5]);
   assign Y_D[6] = y_Q[5] & w;
   flipflop (Y_D[6], Clock, Resetn, 1'b1, y_Q[6]);
   assign Y_D[7] = y_Q[6] & w;
   flipflop (Y_D[7], Clock, Resetn, 1'b1, y_Q[7]);
   assign Y_D[8] = (y_Q[7] | y_Q[8]) & w;
   flipflop (Y_D[8], Clock, Resetn, 1'b1, y_Q[8]);

   assign z = y_Q[4] | y_Q[8];
   assign LEDR[8:0] = y_Q[8:0];
   assign LEDG[0] = z;
endmodule

module flipflop (D, Clock, Resetn, Setn, Q);
   input D, Clock, Resetn, Setn;
   output reg Q;

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         Q <= 1'b0;
      else if (Setn  == 1'b0) // synchronous set
         Q <= 1'b1;
      else
         Q <= D;
endmodule
```

```
// a sequence detector FSM using one-hot encoding that is modifies so that
// the reset state uses code 000...0.
// 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
module part1 (SW, KEY, LEDG, LEDR);
   input [1:0] SW;
   input [0:0] KEY;
   output [0:0] LEDG;
   output [8:0] LEDR;

   wire Clock, Resetn, w, z;
   wire [8:0] y_Q, Y_D;

   assign Clock = KEY[0];
   assign Resetn = SW[0];
   assign w = SW[1];

   assign Y_D[0] = 1'b1;
   flipflop (Y_D[0], Clock, Resetn, y_Q[0]);
   assign Y_D[1] = (~y_Q[0] | y_Q[5] | y_Q[6] | y_Q[7] | y_Q[8]) & ~w;
   flipflop (Y_D[1], Clock, Resetn, y_Q[1]);
   assign Y_D[2] = y_Q[1] & ~w;
   flipflop (Y_D[2], Clock, Resetn, y_Q[2]);
   assign Y_D[3] = y_Q[2] & ~w;
   flipflop (Y_D[3], Clock, Resetn, y_Q[3]);
   assign Y_D[4] = (y_Q[3] | y_Q[4]) & ~w;
   flipflop (Y_D[4], Clock, Resetn, y_Q[4]);

   assign Y_D[5] = (~y_Q[0] | y_Q[1] | y_Q[2] | y_Q[3] | y_Q[4]) & w;
   flipflop (Y_D[5], Clock, Resetn, y_Q[5]);
   assign Y_D[6] = y_Q[5] & w;
   flipflop (Y_D[6], Clock, Resetn, y_Q[6]);
   assign Y_D[7] = y_Q[6] & w;
   flipflop (Y_D[7], Clock, Resetn, y_Q[7]);
   assign Y_D[8] = (y_Q[7] | y_Q[8]) & w;
   flipflop (Y_D[8], Clock, Resetn, y_Q[8]);

   assign z = y_Q[4] | y_Q[8];
   assign LEDR[8:0] = y_Q[8:0];
   assign LEDG[0] = z;
endmodule

module flipflop (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output reg Q;

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         Q <= 1'b0;
      else
         Q <= D;
endmodule
```

```
// a sequence detector FSM.
// SW1 is the active low synchronous reset, SW0 is the w input, and KEY0 is the clock.
// The z output appears on LEDG0, and the state FFs appear on LEDR3..0
module part2 (SW, KEY, LEDG, LEDR);
   input [1:0] SW;
   input [0:0] KEY;
   output [0:0] LEDG;
   output [3:0] LEDR;

   wire Clock, Resetn, w, z;
   reg [3:0] y_Q, Y_D;

   assign Clock = KEY[0];
   assign Resetn = SW[0];
   assign w = SW[1];

   parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
      F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;

   always @(w, y_Q)
   begin: state_table
      case (y_Q)
         A: if (!w) Y_D = B;
            else Y_D = F;
         B: if (!w) Y_D = C;
            else Y_D = F;
         C: if (!w) Y_D = D;
            else Y_D = F;
         D: if (!w) Y_D = E;
            else Y_D = F;
         E: if (!w) Y_D = E;
            else Y_D = F;
         F: if (!w) Y_D = B;
            else Y_D = G;
         G: if (!w) Y_D = B;
            else Y_D = H;
         H: if (!w) Y_D = B;
            else Y_D = I;
         I: if (!w) Y_D = B;
            else Y_D = I;
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         y_Q <= A;
      else
         y_Q <= Y_D;

   assign z = ((y_Q == E) | (y_Q == I)) ? 1'b1 : 1'b0;
   assign LEDR[3:0] = y_Q;
   assign LEDG[0] = z;
endmodule
```

```
// 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
module part3 (SW, KEY, LEDG, LEDR);
   input [1:0] SW;
   input [0:0] KEY;
   output [0:0] LEDG;
   output [7:0] LEDR;

   wire Clock, Resetn, w, z;
   reg [1:4] S4_0s;  // shift register for recognizing 4 0s
   reg [1:4] S4_1s;  // shift register for recognizing 4 1s

   assign Clock = KEY[0];
   assign Resetn = SW[0];
   assign w = SW[1];
   always @(posedge Clock)
   begin
      if (Resetn == 1'b0)
      begin
         S4_0s <= 4'b1111;
         S4_1s <= 4'b0000;
      end
      else
      begin
         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
   end

   assign z = ((S4_0s == 4'b0000) | (S4_1s == 4'b1111)) ? 1'b1 : 1'b0;
   assign LEDR[3:0] = S4_0s;
   assign LEDR[7:4] = S4_1s;
   assign LEDG[0] = z;
endmodule
```

```
// 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
module part4 (SW, KEY, HEX0);
   input [2:0] SW;
   input [0:0] KEY;
   output [0:6] HEX0;

   wire Clock, Resetn;
   wire [1:0] w;
   reg [3:0] y_Q, Y_D;

   assign Clock = KEY[0];
   assign Resetn = SW[0];
   assign w[1:0]= SW[2:1];

   parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
      F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000, J = 4'b1001;

   always @(w, y_Q)
   begin: state_table
      case (y_Q)
         A: case (w)
               2'b00: Y_D = A;
               2'b01: Y_D = B;
               2'b10: Y_D = C;
               2'b11: Y_D = J;
            endcase
         B: case (w)
               2'b00: Y_D = B;
               2'b01: Y_D = C;
               2'b10: Y_D = D;
               2'b11: Y_D = A;
            endcase
         C: case (w)
               2'b00: Y_D = C;
               2'b01: Y_D = D;
               2'b10: Y_D = E;
               2'b11: Y_D = B;
            endcase
         D: case (w)
               2'b00: Y_D = D;
               2'b01: Y_D = E;
               2'b10: Y_D = F;
               2'b11: Y_D = C;
            endcase
         E: case (w)
               2'b00: Y_D = E;
               2'b01: Y_D = F;
               2'b10: Y_D = G;
               2'b11: Y_D = D;
            endcase
         F: case (w)
               2'b00: Y_D = F;
               2'b01: Y_D = G;
               2'b10: Y_D = H;
               2'b11: Y_D = E;
            endcase
         G: case (w)
```

```
                2'b00: Y_D = G;
                2'b01: Y_D = H;
                2'b10: Y_D = I;
                2'b11: Y_D = F;
             endcase
          H: case (w)
                2'b00: Y_D = H;
                2'b01: Y_D = I;
                2'b10: Y_D = J;
                2'b11: Y_D = G;
             endcase
          I: case (w)
                2'b00: Y_D = I;
                2'b01: Y_D = J;
                2'b10: Y_D = A;
                2'b11: Y_D = H;
             endcase
          J: case (w)
                2'b00: Y_D = J;
                2'b01: Y_D = A;
                2'b10: Y_D = B;
                2'b11: Y_D = I;
             endcase
          default: Y_D = 4'bxxxx;
       endcase
    end // state_table

    always @(posedge Clock)
       if (Resetn  == 1'b0) // synchronous clear
          y_Q <= A;
       else
          y_Q <= Y_D;

    bcd7seg (y_Q, HEX0);

endmodule

module bcd7seg (bcd, display);
    input [3:0] bcd;
    output [0:6] display;

    reg [0:6] display;

    /*
     *       0
     *      ---
     *     |   |
     *    5|   |1
     *     | 6 |
     *      ---
     *     |   |
     *    4|   |2
     *     |   |
     *      ---
     *       3
     */
    always @ (bcd)
       case (bcd)
          4'h0: display = 7'b0000001;
          4'h1: display = 7'b1001111;
          4'h2: display = 7'b0010010;
          4'h3: display = 7'b0000110;
          4'h4: display = 7'b1001100;
          4'h5: display = 7'b0100100;
```

```
        4'h6: display = 7'b1100000;
        4'h7: display = 7'b0001111;
        4'h8: display = 7'b0000000;
        4'h9: display = 7'b0001100;
        default: display = 7'b1111111;
    endcase
endmodule
```

```
// 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
module part5 (SW, KEY, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [0:0] SW;
   input [0:0] KEY;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire Clock, Resetn;
   reg [3:0] y_Q, Y_D;
   reg [0:6] FSM_char;  // input to pipeline registers comes from FSM_char for
                        // the first 8 clock cycles, and then comes from the
                        // pipeline's last stage (HELLO travels in a loop)
   reg pipe_select;
   wire [0:6] pipe_input, pipe0, pipe1, pipe2, pipe3, pipe4, pipe5, pipe6, pipe7;

   assign Clock = KEY[0];
   assign Resetn = SW[0];

   parameter S0 = 4'b0000, S1 = 4'b0001, S2 = 4'b0010, S3 = 4'b0011, S4 = 4'b0100,
      S5 = 4'b0101, S6 = 4'b0110, S7 = 4'b0111, S8 = 4'b1000;
   parameter H = 7'b1001000, E = 7'b0110000, L = 7'b1110001, O = 7'b0000001,
      Blank = 7'b1111111;

   always @(y_Q)
   begin: state_table
      case (y_Q)
         S0:   Y_D = S1;
         S1:   Y_D = S2;
         S2:   Y_D = S3;
         S3:   Y_D = S4;
         S4:   Y_D = S5;
         S5:   Y_D = S6;
         S6:   Y_D = S7;
         S7:   Y_D = S8;
         S8:   Y_D = S8;
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         y_Q <= S0;
      else
         y_Q <= Y_D;

   always @(y_Q)
   begin: state_outputs
      pipe_select = 1'b0; FSM_char = 7'bxxxxxxx;
      case (y_Q)
         S0: FSM_char = H;
         S1: FSM_char = E;
         S2: FSM_char = L;
         S3: FSM_char = L;
         S4: FSM_char = O;
         S5: FSM_char = Blank;
         S6: FSM_char = Blank;
         S7: FSM_char = Blank;
         S8: pipe_select = 1'b1; // establish feedback loop
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table
```

```
   assign pipe_input = (pipe_select == 1'b0) ? FSM_char : pipe7;
   // module regne (R, Clock, Resetn, E, Q);
   regne (pipe_input, Clock, Resetn, pipe0);
   regne (pipe0, Clock, Resetn, pipe1);
   regne (pipe1, Clock, Resetn, pipe2);
   regne (pipe2, Clock, Resetn, pipe3);
   regne (pipe3, Clock, Resetn, pipe4);
   regne (pipe4, Clock, Resetn, pipe5);
   regne (pipe5, Clock, Resetn, pipe6);
   regne (pipe6, Clock, Resetn, pipe7);

   assign HEX0 = pipe0;
   assign HEX1 = pipe1;
   assign HEX2 = pipe2;
   assign HEX3 = pipe3;
   assign HEX4 = pipe4;
   assign HEX5 = pipe5;
   assign HEX6 = pipe6;
   assign HEX7 = pipe7;
endmodule

module regne (R, Clock, Resetn, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b1}};
      else
         Q <= R;
endmodule
```

```
// 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
module part6 (KEY, CLOCK_50, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input [0:0] KEY;
   input CLOCK_50;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire Clock, Resetn, Tick;
   reg [3:0] y_Q, Y_D;
   reg [0:6] FSM_char;  // input to pipeline registers comes from FSM_char for
                        // the first 8 clock cycles, and then comes from the
                        // pipeline's last stage (HELLO travels in a loop)
   reg pipe_select;
   wire [0:6] pipe_input, pipe0, pipe1, pipe2, pipe3, pipe4, pipe5, pipe6, pipe7;

   parameter m = 24;
   reg [m-1:0] slow_count;

   assign Clock = CLOCK_50;
   assign Resetn = KEY[0];

   parameter S0 = 4'b0000, S1 = 4'b0001, S2 = 4'b0010, S3 = 4'b0011, S4 = 4'b0100,
      S5 = 4'b0101, S6 = 4'b0110, S7 = 4'b0111, S8 = 4'b1000;
   parameter H = 7'b1001000, E = 7'b0110000, L = 7'b1110001, O = 7'b0000001,
      Blank = 7'b1111111;

   // A large counter to produce a 1 second (approx) enable, called Tick
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;
   assign Tick = ~| slow_count;

   always @(y_Q, Tick)
   begin: state_table
      case (y_Q)
         S0:   if (Tick) Y_D = S1;
               else Y_D = S0;
         S1:   if (Tick) Y_D = S2;
               else Y_D = S1;
         S2:   if (Tick) Y_D = S3;
               else Y_D = S2;
         S3:   if (Tick) Y_D = S4;
               else Y_D = S3;
         S4:   if (Tick) Y_D = S5;
               else Y_D = S4;
         S5:   if (Tick) Y_D = S6;
               else Y_D = S5;
         S6:   if (Tick) Y_D = S7;
               else Y_D = S6;
         S7:   if (Tick) Y_D = S8;
               else Y_D = S7;
         S8: Y_D = S8;
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         y_Q <= S0;
      else
         y_Q <= Y_D;
```

```
   always @(y_Q)
   begin: state_outputs
      pipe_select = 1'b0; FSM_char = 7'bxxxxxxx;
      case (y_Q)
         S0: FSM_char = H;
         S1: FSM_char = E;
         S2: FSM_char = L;
         S3: FSM_char = L;
         S4: FSM_char = O;
         S5: FSM_char = Blank;
         S6: FSM_char = Blank;
         S7: FSM_char = Blank;
         S8: pipe_select = 1'b1; // establish feedback loop
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   assign pipe_input = (pipe_select == 1'b0) ? FSM_char : pipe7;
   // module regne (R, Clock, Resetn, E, Q);
   regne (pipe_input, Clock, Resetn, Tick, pipe0);
   regne (pipe0, Clock, Resetn, Tick, pipe1);
   regne (pipe1, Clock, Resetn, Tick, pipe2);
   regne (pipe2, Clock, Resetn, Tick, pipe3);
   regne (pipe3, Clock, Resetn, Tick, pipe4);
   regne (pipe4, Clock, Resetn, Tick, pipe5);
   regne (pipe5, Clock, Resetn, Tick, pipe6);
   regne (pipe6, Clock, Resetn, Tick, pipe7);

   assign HEX0 = pipe0;
   assign HEX1 = pipe1;
   assign HEX2 = pipe2;
   assign HEX3 = pipe3;
   assign HEX4 = pipe4;
   assign HEX5 = pipe5;
   assign HEX6 = pipe6;
   assign HEX7 = pipe7;
endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b1}};
      else if (E)
         Q <= R;
endmodule
```

```
// 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
module part7 (KEY, CLOCK_50, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDG);
   input [2:0] KEY;
   input CLOCK_50;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [3:0] LEDG;

   wire Clock, Resetn, Tick;
   reg [3:0] y_Q, Y_D;     // This is the state machine that controls the pipeline
   wire Fast, Slow;        // Variable tick interval controls
   reg [3:0] yV_Q, YV_D;   // This is the state machine that it used to implement
                           // a variable tick interval
   reg [0:6] FSM_char;  // input to pipeline registers comes from FSM_char for
                        // the first 8 clock cycles, and then comes from the
                        // pipeline's last stage (HELLO travels in a loop)
   reg pipe_select;
   wire [0:6] pipe_input, pipe0, pipe1, pipe2, pipe3, pipe4, pipe5, pipe6, pipe7;

   reg var_count_sync;
   reg [3:0] var_count, Modulus; // used to implement a variable delay

   parameter m = 23;
   reg [m-1:0] slow_count;

   assign Clock = CLOCK_50;
   assign Resetn = KEY[0];

   // Synchronize the KEY inputs
   wire KEY_sync[2:1];
   regne #(.n(1)) (~KEY[1], Clock, Resetn, 1'b1, KEY_sync[1]);
   regne #(.n(1)) (KEY_sync[1], Clock, Resetn, 1'b1, Fast);
   regne #(.n(1)) (~KEY[2], Clock, Resetn, 1'b1, KEY_sync[2]);
   regne #(.n(1)) (KEY_sync[2], Clock, Resetn, 1'b1, Slow);

   // names of states for changing a counter value for implementing
   // a variable tick delay
   parameter Sync3 = 4'b0000, Speed3 = 4'b0001, Sync4 = 4'b0010, Speed4 = 4'b0011,
             Sync5 = 4'b0100, Speed5 = 4'b0101, Sync1 = 4'b0110, Speed1 = 4'b0111,
                Sync2 = 4'b1000, Speed2 = 4'b1001;
   parameter S0 = 4'b0000, S1 = 4'b0001, S2 = 4'b0010, S3 = 4'b0011, S4 = 4'b0100,
      S5 = 4'b0101, S6 = 4'b0110, S7 = 4'b0111, S8 = 4'b1000;
   parameter H = 7'b1001000, E = 7'b0110000, L = 7'b1110001, O = 7'b0000001,
      Blank = 7'b1111111;

   // 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
   always @(yV_Q, Fast, Slow)
   begin: state_table_speed
      case (yV_Q)
         Sync5:   if (Slow | Fast) YV_D = Sync5;   // wait for key to be released
                  else YV_D = Speed5;
         Speed5:  if (!Slow) YV_D = Speed5;  // fastest speed
                  else YV_D = Sync4;         // change to slower speed

         Sync4:   if (Slow | Fast) YV_D = Sync4;   // wait for key to be released
```

```
                else YV_D = Speed4;
         Speed4:  if (!Slow & !Fast) YV_D = Speed4;   // keep this speed
                  else if (Slow) YV_D = Sync3;        // change to slower speed
                  else YV_D = Sync5;         // change to faster speed

         Sync3:   if (Slow | Fast) YV_D = Sync3;   // wait for key to be released
                  else YV_D = Speed3;
         Speed3:  if (!Slow & !Fast) YV_D = Speed3;   // keep this speed
                  else if (Slow) YV_D = Sync2;        // change to slower speed
                  else YV_D = Sync4;         // change to faster speed

         Sync2:   if (Slow | Fast) YV_D = Sync2;   // wait for key to be released
                  else YV_D = Speed2;
         Speed2:  if (!Slow & !Fast) YV_D = Speed2;   // keep this speed
                  else if (Slow) YV_D = Sync1;        // change to slower speed
                  else YV_D = Sync3;         // change to faster speed

         Sync1:   if (Slow | Fast) YV_D = Sync1;   // wait for key to be released
                  else YV_D = Speed1;
         Speed1:  if (!Fast) YV_D = Speed1;     // keep this speed
                  else YV_D = Sync2;                // change to faster speed
         default: YV_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         yV_Q <= Sync3;    // middle speed
      else
         yV_Q <= YV_D;

   always @(yV_Q)
   begin: state_outputs_speed
      Modulus = 4'bxxxx; var_count_sync = 1'b1;
      case (yV_Q)
         Sync5: var_count_sync = 1'b0;
         Speed5: Modulus = 4'b0000;
         Sync4: var_count_sync = 1'b0;
         Speed4: Modulus = 4'b0001;
         Sync3: var_count_sync = 1'b0;
         Speed3: Modulus = 4'b0011;
         Sync2: var_count_sync = 1'b0;
         Speed2: Modulus = 4'b0110;
         Sync1: var_count_sync = 1'b0;
         Speed1: Modulus = 4'b1100;
      endcase
   end // state_table

   assign LEDG[3:0] = Modulus;

   // A large counter to produce an approx .25 second delay
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;

   // Counter that provides a variable delay
   always @(posedge Clock)
      if (var_count_sync == 1'b0)
         var_count <= 0;
      else if (slow_count == 0)
         if (var_count == Modulus)
            var_count <= 0;
         else
            var_count <= var_count + 1'b1;
```

```
   // 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.
   assign Tick = ~| var_count & ~| slow_count & var_count_sync;

   // state machine that controls the pipeline
   always @(y_Q, Tick)
   begin: state_table
      case (y_Q)
         S0:   if (Tick) Y_D = S1;
               else Y_D = S0;
         S1:   if (Tick) Y_D = S2;
               else Y_D = S1;
         S2:   if (Tick) Y_D = S3;
               else Y_D = S2;
         S3:   if (Tick) Y_D = S4;
               else Y_D = S3;
         S4:   if (Tick) Y_D = S5;
               else Y_D = S4;
         S5:   if (Tick) Y_D = S6;
               else Y_D = S5;
         S6:   if (Tick) Y_D = S7;
               else Y_D = S6;
         S7:   if (Tick) Y_D = S8;
               else Y_D = S7;
         S8: Y_D = S8;
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
      if (Resetn  == 1'b0) // synchronous clear
         y_Q <= S0;
      else
         y_Q <= Y_D;

   always @(y_Q)
   begin: state_outputs
      pipe_select = 1'b0; FSM_char = 7'bxxxxxxx;
      case (y_Q)
         S0: FSM_char = H;
         S1: FSM_char = E;
         S2: FSM_char = L;
         S3: FSM_char = L;
         S4: FSM_char = O;
         S5: FSM_char = Blank;
         S6: FSM_char = Blank;
         S7: FSM_char = Blank;
         S8: pipe_select = 1'b1; // establish feedback loop
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   assign pipe_input = (pipe_select == 1'b0) ? FSM_char : pipe7;
   // module regne (R, Clock, Resetn, E, Q);
   regne (pipe_input, Clock, Resetn, Tick, pipe0);
   regne (pipe0, Clock, Resetn, Tick, pipe1);
   regne (pipe1, Clock, Resetn, Tick, pipe2);
   regne (pipe2, Clock, Resetn, Tick, pipe3);
   regne (pipe3, Clock, Resetn, Tick, pipe4);
   regne (pipe4, Clock, Resetn, Tick, pipe5);
   regne (pipe5, Clock, Resetn, Tick, pipe6);
   regne (pipe6, Clock, Resetn, Tick, pipe7);
```

```
   assign HEX0 = pipe0;
   assign HEX1 = pipe1;
   assign HEX2 = pipe2;
   assign HEX3 = pipe3;
   assign HEX4 = pipe4;
   assign HEX5 = pipe5;
   assign HEX6 = pipe6;
   assign HEX7 = pipe7;
endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b1}};
      else if (E)
         Q <= R;
endmodule
```

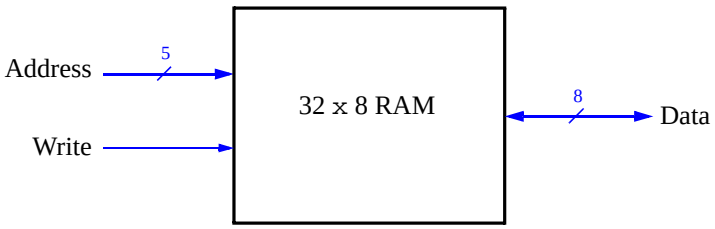
Laboratory Exercise 8

Memory Blocks

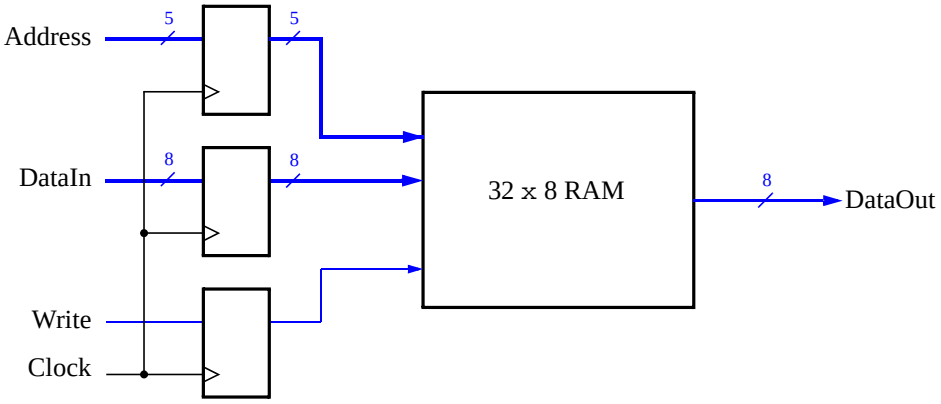
In computer systems it is necessary to provide a substantial amount of memory. If a system is implemented using FPGA technology it is possible to provide some amount of memory by using the memory resources that exist in the FPGA device. If additional memory is needed, it has to be implemented by connecting 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 Verilog 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 Verilog HDL as the type of output file to create, and give the file the name *ramlpm.v*. 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 Verilog file the module generated in *ramlpm.v*. Include appropriate input and output signals in your Verilog code for the memory ports given in Figure 1*b*.

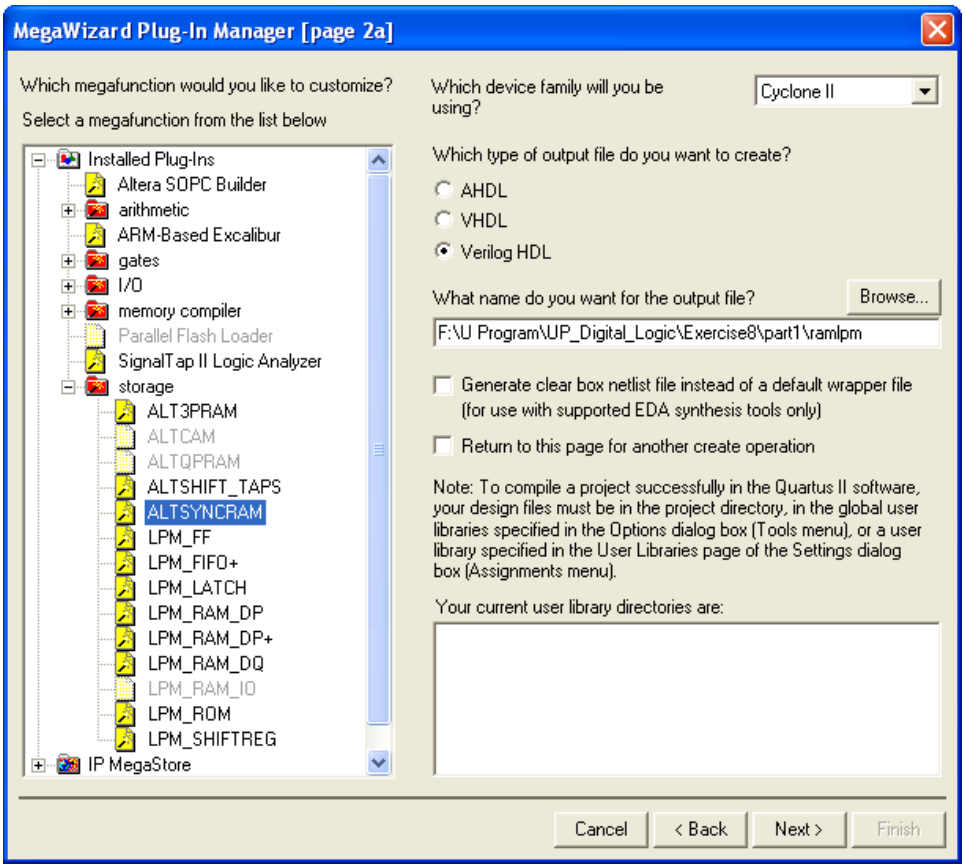


Figure 2. Choosing the *altsyncram* LPM.

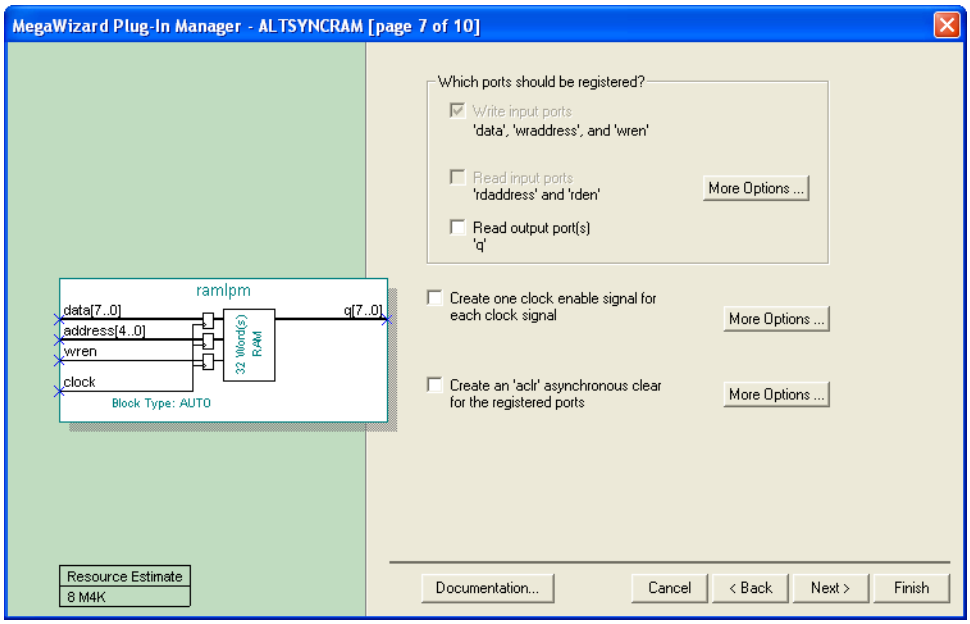


Figure 3. Configuring input and output ports on the *altsyncram* LPM.

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 Verilog 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 Verilog code. In a Verilog-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 statement

```
reg [7:0]  memory_array [31:0];
```

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 Verilog 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 Verilog 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 Verilog 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.

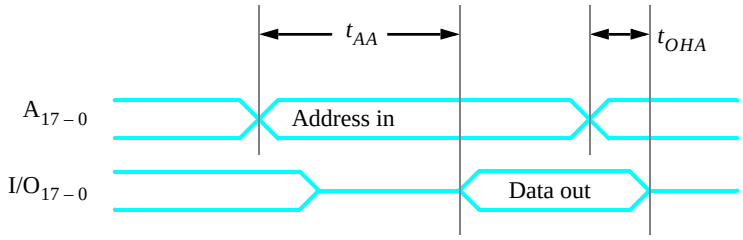
Part IV

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.

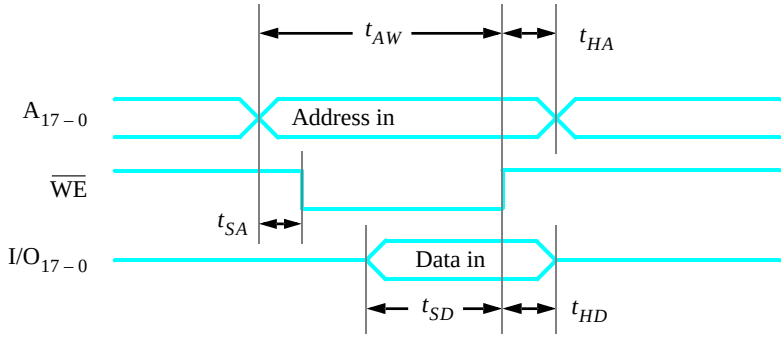
Name	Purpose
$\overline{CE}$	Chip enable−asserted low during all SRAM operations
$\overline{OE}$	Output enable−can be asserted low during only read operations, or during all operations
$\overline{WE}$	Write enable−asserted low during a write operation
$\overline{UB}$	Upper byte−asserted low to read or 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.

Parameter	Value 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 Verilog 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 Verilog module.

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.v*.

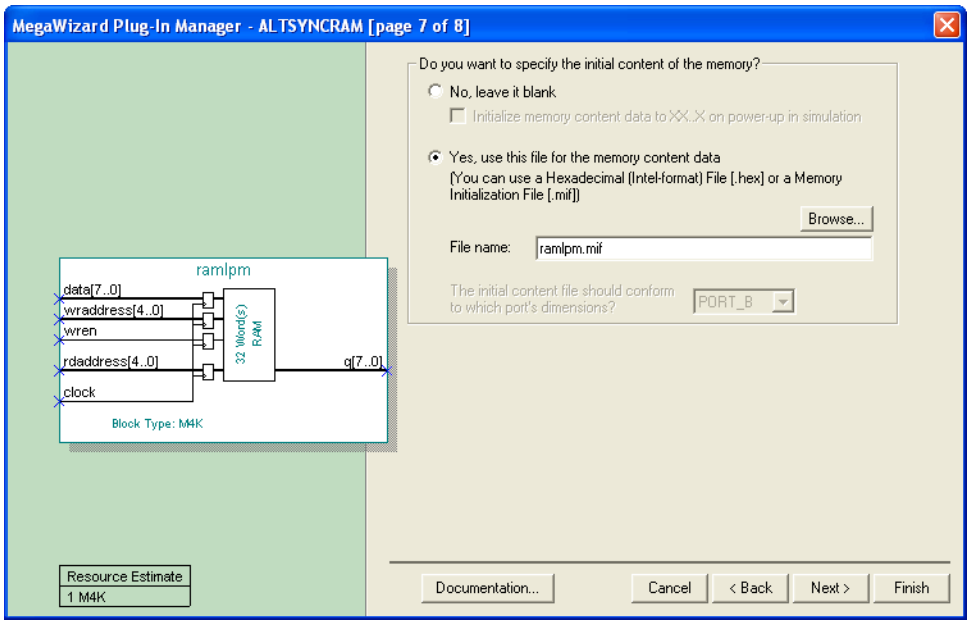


Figure 5. Specifying a memory initialization file (MIF).

2. Write a Verilog 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.

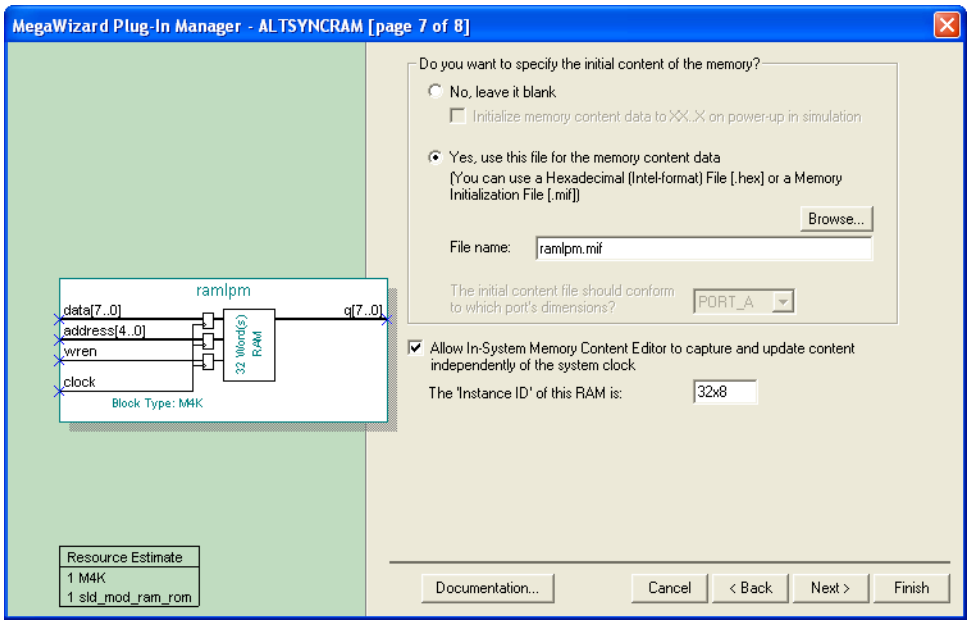


Figure 6. Configuring *altsyncram* for use with the In-System Memory Content Editor.

2. Write a Verilog 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.

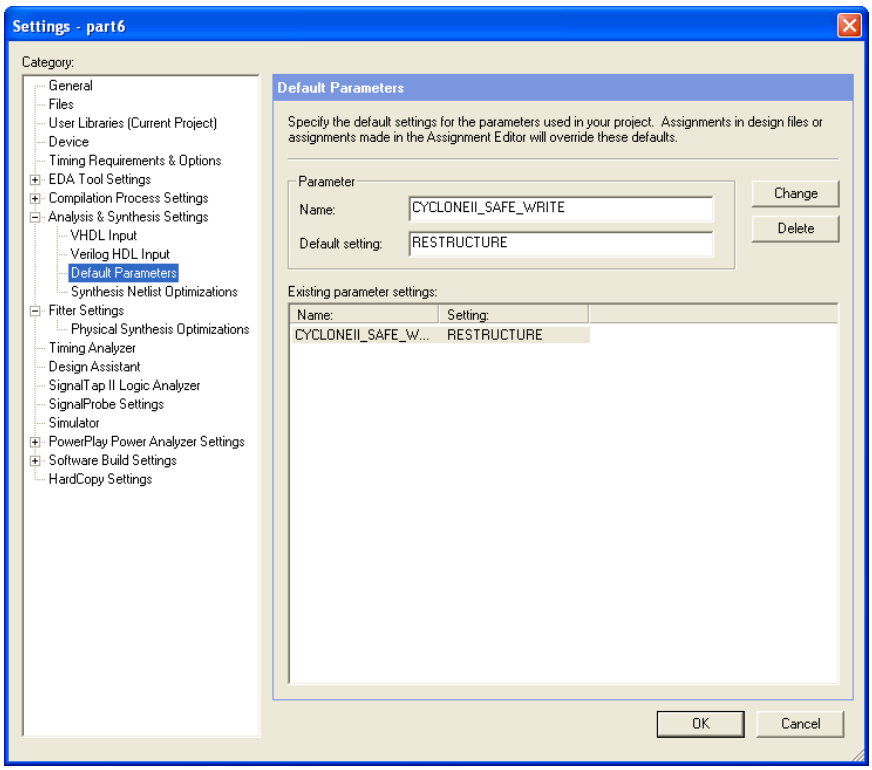


Figure 7. Setting the *CYCLONEII_SAFE_WRITE* parameter.

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.

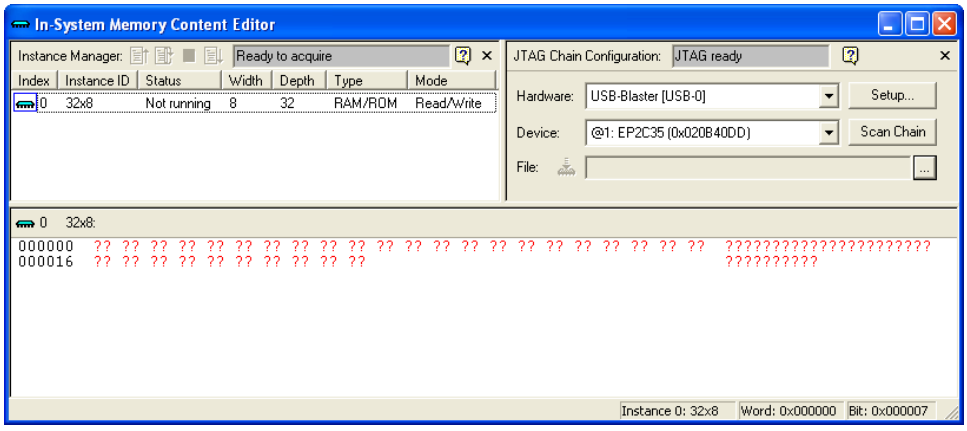


Figure 8. The In-System Memory Content Editor window.

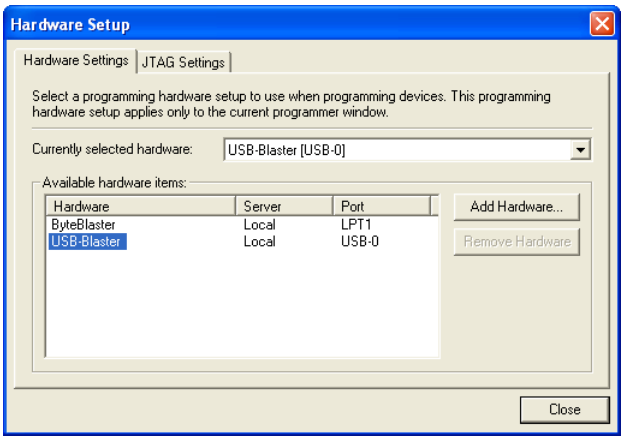


Figure 9. The Hardware Setup window.

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.

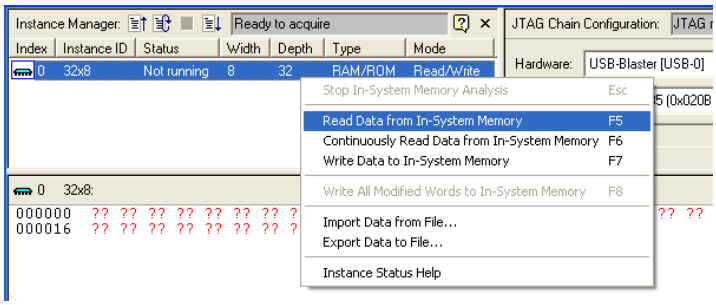


Figure 10. Using the In-System Memory Content Editor tool.

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.

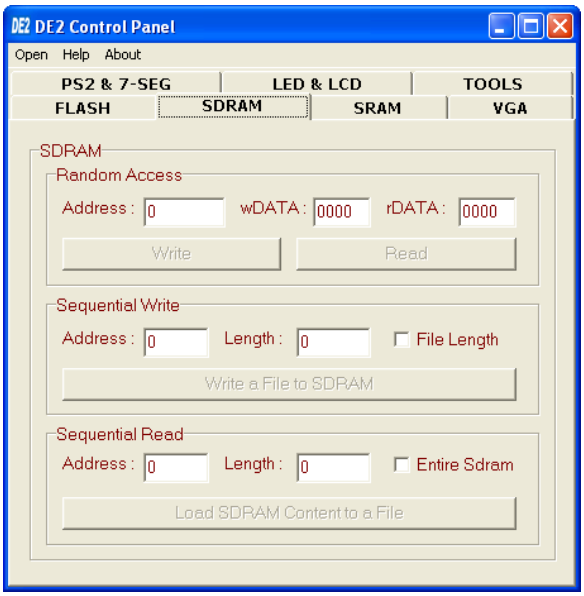


Figure 11. The DE2 Control Panel software.

```
// LPM RAM module
// inputs:
//    Clock
//    Address
//    Write: asserted to perform a write
//    DataIn: data to be written
//
// outputs:
//    DataOut: data read
module part1 (Clock, DataIn, DataOut, Address, Write);
   input Clock, Write;
   input [7:0] DataIn;
   output [7:0] DataOut;
   input [4:0] Address;

   // instantiate LPM module
   // module ramlpm (address, clock, data, wren, q);
   ramlpm m32x8 (Address, Clock, DataIn, Write, DataOut);
endmodule
```

```
// 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.
module part2 (KEY, SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDG);
   input [0:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;

   wire Clock, Write;
   wire [4:0] Address;
   wire [7:0] DataIn, DataOut;

   assign Clock = KEY[0];
   assign Write = SW[17];
   assign DataIn = SW[7:0];
   assign Address = SW[15:11];

   // instantiate LPM module
   // module ramlpm (address, clock, data, wren, q);
   ramlpm (Address, Clock, DataIn, Write, DataOut);

   // display the data input, data output, and address on the 7-segs
   // display the data input, data output, and address on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   assign HEX2 = 7'b1111111;  // blank
   assign HEX3 = 7'b1111111;  // blank
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Address[4]}), HEX7);

   assign LEDG[0] = Write;
endmodule

// the B input blanks the display when B = 1
module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
```

```
            4'h2: display = 7'b0010010;
            4'h3: display = 7'b0000110;
            4'h4: display = 7'b1001100;
            4'h5: display = 7'b0100100;
            4'h6: display = 7'b1100000;
            4'h7: display = 7'b0001111;
            4'h8: display = 7'b0000000;
            4'h9: display = 7'b0001100;
            4'hA: display = 7'b0001000;
            4'hb: display = 7'b1100000;
            4'hC: display = 7'b0110001;
            4'hd: display = 7'b1000010;
            4'hE: display = 7'b0110000;
            4'hF: display = 7'b0111000;
        endcase
endmodule
```

```
// This code implements an inferred memory block
//
// 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.
module part3 (KEY, SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDG);
   input [0:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;

   wire Clock, Write;
   wire [4:0] Address;
   wire [7:0] DataIn, DataOut;

   assign Clock = KEY[0];
   assign Write = SW[17];
   assign DataIn = SW[7:0];
   assign Address = SW[15:11];

   reg [7:0] memory_array [31:0];
   reg [4:0] Address_reg;

   // infer RAM module
   always @(posedge Clock)
   begin
      if (Write)
         memory_array[Address] <= DataIn;
      Address_reg <= Address;
   end

   assign DataOut = memory_array[Address_reg];

   // display the data input, data output, and address on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Address[4]}), HEX7);

   assign LEDG[0] = Write;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
```

```
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
// 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.
module part4 (KEY, SW, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDG,
   SRAM_ADDR, SRAM_DQ, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, SRAM_LB_N);
   input [1:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;
   output [17:0] SRAM_ADDR;
   inout [15:0] SRAM_DQ;
   output SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, SRAM_LB_N;

   wire Resetn, Clock, Write, CE;
   wire [4:0] Address;
   wire [7:0] DataIn, DataOut;
   wire [15:0] DataIn_reg;

   assign Resetn = KEY[0];
   assign Clock = KEY[1];

   assign Write = SW[17];
   regne #(.n(1)) R1 (~Write, Clock, Resetn, 1'b1, SRAM_WE_N);
   assign Address = SW[15:11];
   regne #(.n(5)) R2 (Address, Clock, Resetn, 1'b1, SRAM_ADDR[4:0]);
   assign SRAM_ADDR[17:5] = 13'b0;

   assign DataIn = SW[7:0];
   regne #(.n(8)) R3 (DataIn, Clock, Resetn, 1'b1, DataIn_reg[7:0]);
   assign DataIn_reg[15:8] = 8'b0;

   assign SRAM_DQ = (SRAM_WE_N == 1'b0) ? DataIn_reg : 16'bz;

   // hold CE_N to 1 at power-up, to avoid an accidental write
   regne #(.n(1)) R4 (1'b1, Clock, Resetn, 1'b1, CE);
   assign SRAM_CE_N = ~CE;

   assign SRAM_OE_N = 1'b0;
   assign SRAM_UB_N = 1'b0;
   assign SRAM_LB_N = 1'b0;

   assign DataOut = SRAM_DQ[7:0];

   // display the data input, data output, and address on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Address[4]}), HEX7);

   assign LEDG[0] = Write;
endmodule
```

```
module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E)
         Q <= R;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
// 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.
module part5 (CLOCK_50, KEY, SW, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0, LEDG);
   input CLOCK_50;
   input [0:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;

   wire Clock, Resetn, Write, Write_sync;
   wire [4:0] Write_address, Write_address_sync;
   wire [7:0] DataIn, DataIn_sync, DataOut;

   assign Resetn = KEY[0];
   assign Clock = CLOCK_50;

   // synchronize all asynchronous inputs to the clock
   regne #(.n(1)) (SW[17], Clock, Resetn, 1'b1, Write_sync);
   regne #(.n(1)) (Write_sync, Clock, Resetn, 1'b1, Write);
   regne #(.n(5)) (SW[15:11], Clock, Resetn, 1'b1, Write_address_sync);
   regne #(.n(5)) (Write_address_sync, Clock, Resetn, 1'b1, Write_address);
   regne #(.n(8)) (SW[7:0], Clock, Resetn, 1'b1, DataIn_sync);
   regne #(.n(8)) (DataIn_sync, Clock, Resetn, 1'b1, DataIn);

   // one second cycle counter
   parameter m = 25;
   reg [m-1:0] slow_count;
   reg [4:0] Read_address; // cycles from addresses 0 to 31 at one second per address

   // Create a 1Hz 5-bit address counter
   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;
   // the read address counter
   always @ (posedge Clock)
      if (Resetn == 0)
         Read_address <= 5'b0;
      else if (slow_count == 0)
         Read_address <= Read_address + 1'b1;

   // instantiate LPM module
   // module ramlpm (clock, data, rdaddress, wraddress, wren, q);
   ramlpm (Clock, DataIn, Read_address, Write_address, Write, DataOut);

   // display the data input, data output, and addresses on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   hex7seg digit2 (Read_address[3:0], HEX2);
   hex7seg digit3 (({3'b0,Read_address[4]}), HEX3);
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Write_address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Write_address[4]}), HEX7);

   assign LEDG[0] = Write;
endmodule
```

```
module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E)
         Q <= R;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
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.
module part6 (CLOCK_50, KEY, SW, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0, LEDG);
   input CLOCK_50;
   input [0:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;

   wire Clock, Resetn, Write, Write_sync;
   wire [4:0] Write_address, Write_address_sync, Address;
   wire [7:0] DataIn, DataIn_sync, DataOut;

   assign Resetn = KEY[0];
   assign Clock = CLOCK_50;

   // synchronize all asynchronous inputs to the clock
   regne #(.n(1)) R1 (SW[17], Clock, Resetn, 1'b1, Write_sync);
   regne #(.n(1)) R2 (Write_sync, Clock, Resetn, 1'b1, Write);
   regne #(.n(5)) R3 (SW[15:11], Clock, Resetn, 1'b1, Write_address_sync);
   regne #(.n(5)) R4 (Write_address_sync, Clock, Resetn, 1'b1, Write_address);
   regne #(.n(8)) R5 (SW[7:0], Clock, Resetn, 1'b1, DataIn_sync);
   regne #(.n(8)) R6 (DataIn_sync, Clock, Resetn, 1'b1, DataIn);

   // one second cycle counter
   parameter m = 25;
   reg [m-1:0] slow_count;
   reg [4:0] Read_address; // cycles from addresses 0 to 31 at one second per address

   // Create a 1Hz 5-bit address counter
   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;
   // the read address counter
   always @ (posedge Clock)
      if (Resetn == 0)
         Read_address <= 5'b0;
      else if (slow_count == 0)
         Read_address <= Read_address + 1'b1;

   assign Address = (Write == 1) ? Write_address : Read_address;
   // instantiate LPM module
   // module ramlpm (address, clock, data, wren, q);
   ramlpm m32x8 (Address, Clock, DataIn, Write, DataOut);

   // display the data input, data output, and addresses on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   hex7seg digit2 (Read_address[3:0], HEX2);
   hex7seg digit3 (({3'b0,Read_address[4]}), HEX3);
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Write_address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Write_address[4]}), HEX7);

   assign LEDG[0] = Write;
```

```
endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E)
         Q <= R;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
         4'h9: display = 7'b0001100;
         4'hA: display = 7'b0001000;
         4'hb: display = 7'b1100000;
         4'hC: display = 7'b0110001;
         4'hd: display = 7'b1000010;
         4'hE: display = 7'b0110000;
         4'hF: display = 7'b0111000;
      endcase
endmodule
```

```
// 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.
module part7 (CLOCK_50, KEY, SW, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0, LEDG,
   SRAM_ADDR, SRAM_DQ, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, SRAM_LB_N);
   input CLOCK_50;
   input [0:0] KEY;
   input [17:0] SW;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [0:0] LEDG;
   output [17:0] SRAM_ADDR;
   inout [15:0] SRAM_DQ;
   output SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, SRAM_LB_N;

   wire Resetn, Clock, Write_n_sync, CE, CE_1, CE_2;
   wire [4:0] Write_address, Write_address_sync;
   wire [7:0] DataIn, DataIn_sync, DataOut;

   assign Resetn = KEY[0];
   assign Clock = CLOCK_50;

   // synchronize all asynchronous inputs to the clock
   regne #(.n(1)) R1 (!SW[17], Clock, Resetn, 1'b1, Write_n_sync);
   regne #(.n(1)) R2 (Write_n_sync, Clock, Resetn, 1'b1, SRAM_WE_N);
   regne #(.n(5)) R3 (SW[15:11], Clock, Resetn, 1'b1, Write_address_sync);
   regne #(.n(5)) R4 (Write_address_sync, Clock, Resetn, 1'b1, Write_address);
   regne #(.n(8)) R5 (SW[7:0], Clock, Resetn, 1'b1, DataIn_sync);
   regne #(.n(8)) R6 (DataIn_sync, Clock, Resetn, 1'b1, DataIn);

   // one second cycle counter
   parameter m = 25;
   reg [m-1:0] slow_count;
   reg [4:0] Read_address; // cycles from addresses 0 to 31 at one second per address

   // Create a 1Hz 5-bit address counter
   // A large counter to produce a 1 second (approx) enable
   always @(posedge Clock)
      slow_count <= slow_count + 1'b1;
   // the read address counter
   always @ (posedge Clock)
      if (Resetn == 0)
         Read_address <= 5'b0;
      else if (slow_count == 0)
         Read_address <= Read_address + 1'b1;

   assign SRAM_ADDR = (SRAM_WE_N == 0) ? {13'b0,Write_address} : {13'b0,Read_address};

   assign SRAM_DQ = (SRAM_WE_N == 1'b0) ? {8'b0,DataIn} : 16'bz;

   // hold CE_N to 1 for two clock cycles after power-up, to avoid an accidental write
   regne #(.n(1)) R7 (1'b1, Clock, Resetn, 1'b1, CE_1);
   regne #(.n(1)) R8 (CE_1, Clock, Resetn, 1'b1, CE_2);
   regne #(.n(1)) R9 (CE_2, Clock, Resetn, 1'b1, CE);
   assign SRAM_CE_N = ~CE;
```

```
   assign SRAM_OE_N = 1'b0;
   assign SRAM_UB_N = 1'b0;
   assign SRAM_LB_N = 1'b0;

   assign DataOut = SRAM_DQ[7:0];

   // display the data input, data output, and address on the 7-segs
   hex7seg digit0 (DataOut[3:0], HEX0);
   hex7seg digit1 (DataOut[7:4], HEX1);
   hex7seg digit2 (Read_address[3:0], HEX2);
   hex7seg digit3 ({3'b0,Read_address[4]}, HEX3);
   hex7seg digit4 (DataIn[3:0], HEX4);
   hex7seg digit5 (DataIn[7:4], HEX5);
   hex7seg digit6 (Write_address[3:0], HEX6);
   hex7seg digit7 (({3'b0,Write_address[4]}), HEX7);

   assign LEDG[0] = SRAM_WE_N;
endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E)
         Q <= R;
endmodule

module hex7seg (hex, display);
   input [3:0] hex;
   output [0:6] display;

   reg [0:6] display;

   /*
    *       0
    *      ---
    *     |   |
    *    5|   |1
    *     | 6 |
    *      ---
    *     |   |
    *    4|   |2
    *     |   |
    *      ---
    *       3
    */
   always @ (hex)
      case (hex)
         4'h0: display = 7'b0000001;
         4'h1: display = 7'b1001111;
         4'h2: display = 7'b0010010;
         4'h3: display = 7'b0000110;
         4'h4: display = 7'b1001100;
         4'h5: display = 7'b0100100;
         4'h6: display = 7'b1100000;
         4'h7: display = 7'b0001111;
         4'h8: display = 7'b0000000;
```

```
            4'h9: display = 7'b0001100;
            4'hA: display = 7'b0001000;
            4'hb: display = 7'b1100000;
            4'hC: display = 7'b0110001;
            4'hd: display = 7'b1000010;
            4'hE: display = 7'b0110000;
            4'hF: display = 7'b0111000;
        endcase
endmodule
```

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.

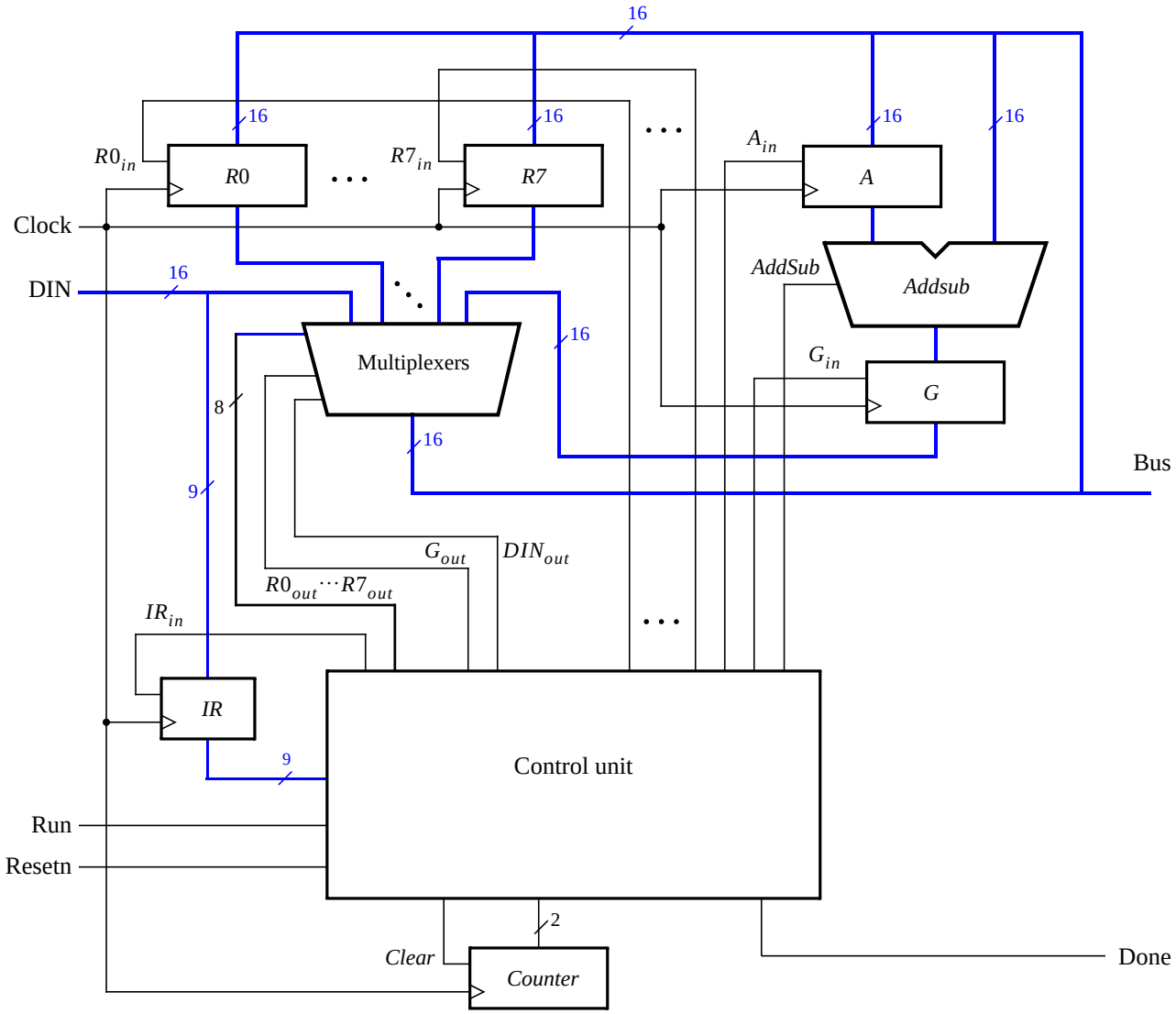


Figure 1. A digital system.

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 Verilog code as follows:

1. Create a new Quartus II project for this exercise.

2. Generate the required Verilog file, include it in your project, and compile the circuit. A suggested skeleton of the Verilog code is shown in Figure 2a, and some subcircuit modules that can be used in this code appear in Figure 2b.

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 module that contains the appropriate input and output ports for the Altera board. Instantiate your processor in this top-level module. 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.

```
module  proc (DIN, Resetn, Clock, Run, Done, BusWires);
    input [15:0] DIN;
    input Resetn, Clock, Run;
    output Done;
    output [15:0] BusWires;

    . . . declare variables

    wire Clear = . . .
    upcount Tstep (Clear, Clock, Tstep_Q);
    assign I = IR[1:3];
    dec3to8 decX (IR[4:6], 1'b1, Xreg);
    dec3to8 decY (IR[7:9], 1'b1, Yreg);

    always @(Tstep_Q or I or Xreg or Yreg)
    begin
        . . . specify initial values
        case (Tstep_Q)
            2'b00: // store DIN in IR in time step 0
            begin
                IRin = 1'b1;
            end
            2'b01: //define signals in time step 1
                case (I)
                    . . .
                endcase
            2'b10: //define signals in time step 2
                case (I)
                    . . .
                endcase
            2'b11: //define signals in time step 3
                case (I)
                    . . .
                endcase
        endcase
    end

    regn reg_0 (BusWires, Rin[0], Clock, R0);
    . . . instantiate other registers and the adder/subtracter unit

    . . . define the bus

endmodule
```

Figure 2a. Skeleton Verilog code for the processor.

```
module upcount(Clear, Clock, Q);
    input Clear, Clock;
    output [1:0] Q;
    reg [1:0] Q;

    always @(posedge Clock)
        if (Clear)
            Q <= 2'b0;
        else
            Q <= Q + 1'b1;
endmodule

module  dec3to8(W, En, Y);
    input [2:0] W;
    input En;
    output [0:7] Y;
    reg [0:7] Y;

    always @(W or En)
    begin
        if (En == 1)
            case (W)
                3'b000: Y = 8'b10000000;
                3'b001: Y = 8'b01000000;
                3'b010: Y = 8'b00100000;
                3'b011: Y = 8'b00010000;
                3'b100: Y = 8'b00001000;
                3'b101: Y = 8'b00000100;
                3'b110: Y = 8'b00000010;
                3'b111: Y = 8'b00000001;
            endcase
        else
            Y = 8'b00000000;
    end
endmodule

module  regn(R, Rin, Clock, Q);
    parameter n = 16;
    input [n-1:0] R;
    input Rin, Clock;
    output [n-1:0] Q;
    reg [n-1:0] Q;

    always @(posedge Clock)
        if (Rin)
            Q <= R;
endmodule
```

Figure 2*b*. Subcircuit modules for use in the processor.

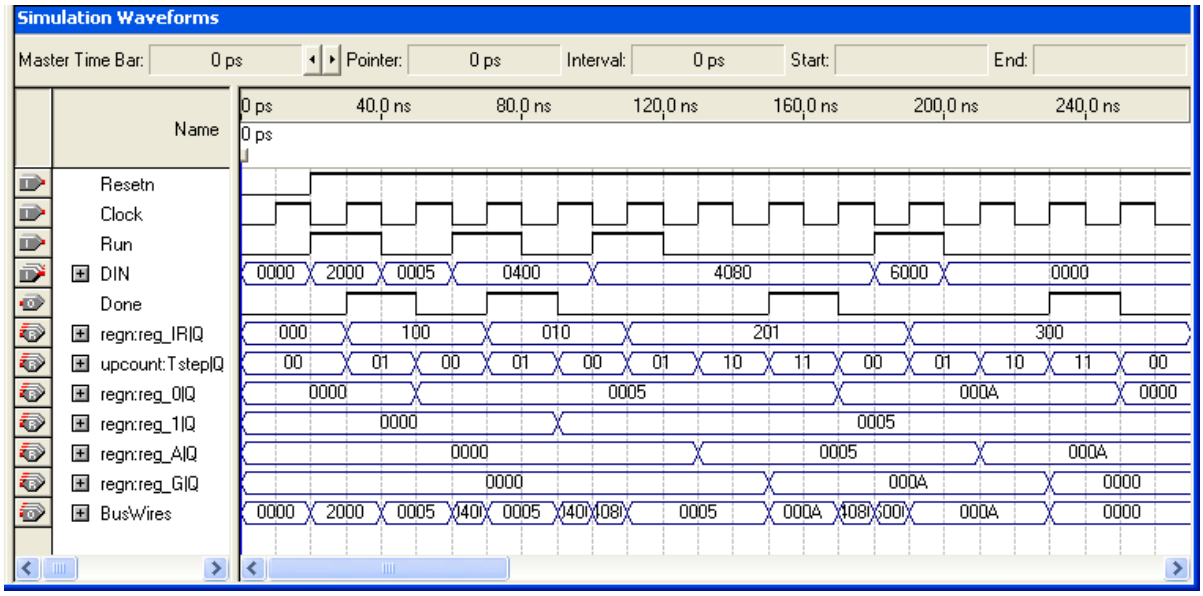


Figure 3. Simulation of the processor.

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.

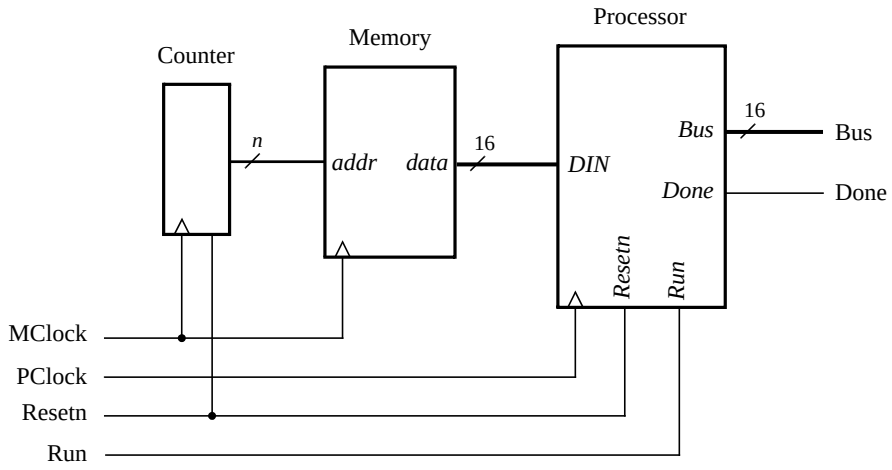


Figure 4. Connecting the processor to a memory and counter.

1. Create a new Quartus II project which will be used to test your circuit.

2. Generate a top-level Verilog 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.

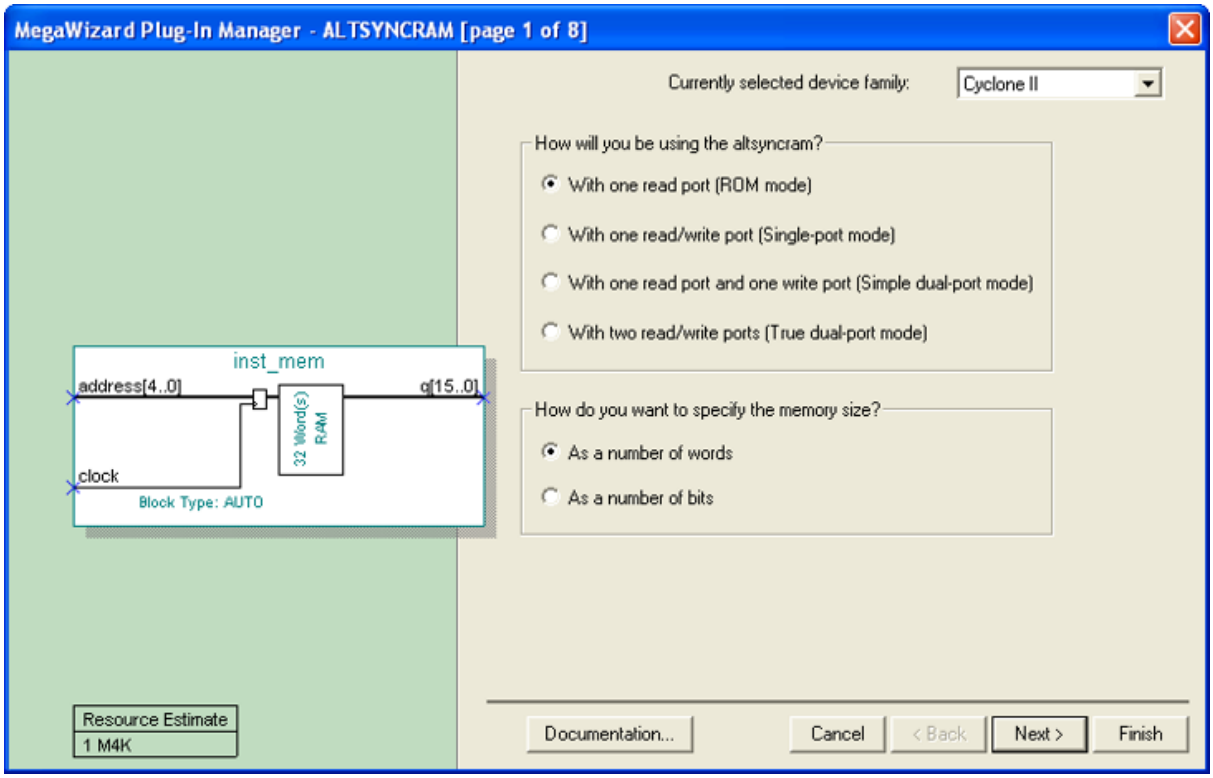


Figure 5. ALTSYNCRAM configuration.

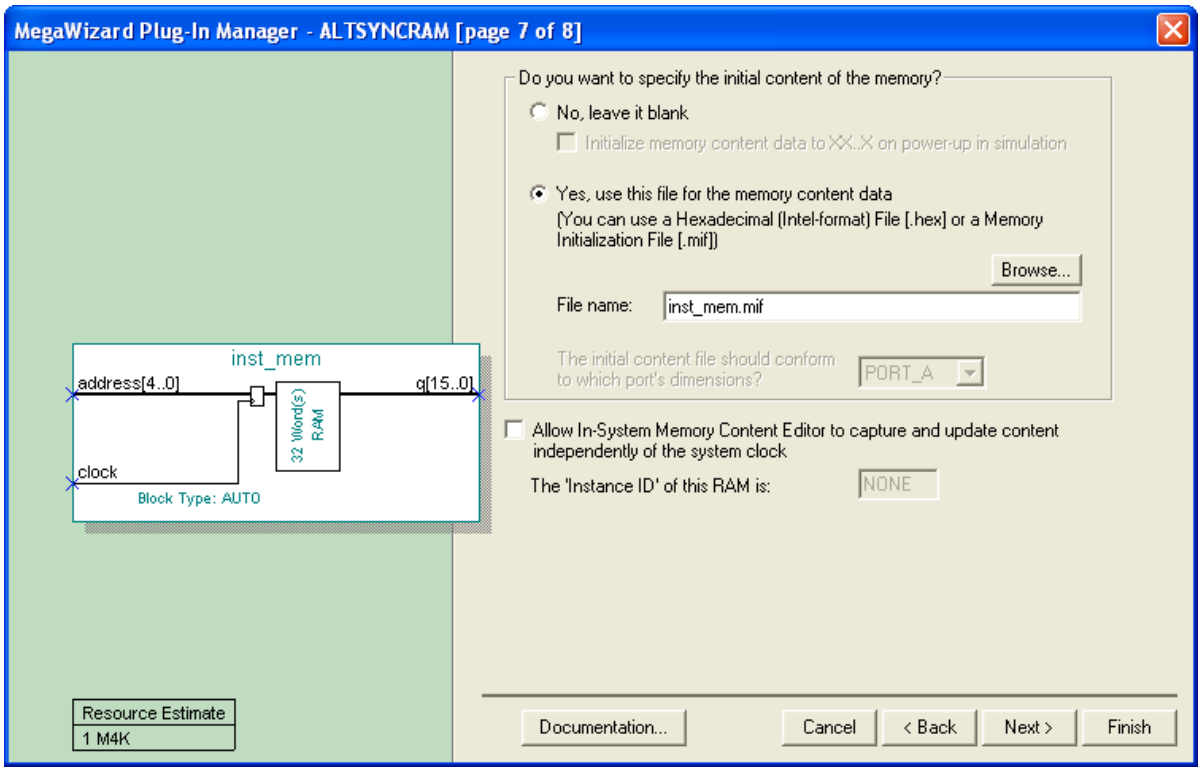


Figure 6. Specifying a memory initialization file (MIF).

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.

```
module proc(DIN, Resetn, Clock, Run, Done, BusWires);
   input [15:0] DIN;
   input Resetn, Clock, Run;
   output Done;
   output [15:0] BusWires;

   reg [15:0] BusWires;
   reg [0:7] Rin, Rout;
   reg [15:0] Sum;
   reg IRin, Done, DINout, Ain, Gin, Gout, AddSub;
   wire [1:0] Tstep_Q;
   wire [2:0] I;
   wire [0:7] Xreg, Yreg;
   wire [15:0] R0, R1, R2, R3, R4, R5, R6, R7, A, G;
   wire [1:9] IR;
   wire [1:10] Sel; // bus selector

   wire Clear = ~Resetn | Done | (~Run & ~Tstep_Q[1] & ~Tstep_Q[0]); // fix
   upcount Tstep (Clear, Clock, Tstep_Q);
   assign I = IR[1:3];
   dec3to8 decX (IR[4:6], 1'b1, Xreg);
   dec3to8 decY (IR[7:9], 1'b1, 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
   */
   always @(Tstep_Q or I or Xreg or Yreg)
   begin
      Done = 1'b0; Ain = 1'b0; Gin = 1'b0; Gout = 1'b0; AddSub = 1'b0;
      IRin = 1'b0; DINout = 1'b0; Rin = 8'b0; Rout = 8'b0;
      case (Tstep_Q)
         2'b00: // store DIN in IR as long as Tstep_Q == 0
            begin
               IRin = 1'b1;
            end
         2'b01:   //define signals in time step T1
            case (I)
               3'b000: // mv Rx,Ry
               begin
                  Rout = Yreg;
                  Rin = Xreg;
                  Done = 1'b1;
               end
               3'b001: // mvi Rx,#D
               begin
                  // data is required to be on DIN
                  DINout = 1'b1;
                  Rin = Xreg;
                  Done = 1'b1;
               end
               3'b010, 3'b11: //add, sub
               begin
                  Rout = Xreg;
                  Ain = 1'b1;
               end
               default: ;
               endcase
         2'b10:   //define signals in time step T2
```

```
            case (I)
               3'b010: // add
               begin
                  Rout = Yreg;
                  Gin = 1'b1;
               end
               3'b011: // sub
               begin
                  Rout = Yreg;
                  AddSub = 1'b1;
                  Gin = 1'b1;
               end
               default: ;
            endcase
         2'b11:   //define signals in time step T3
            case (I)
               3'b010, 3'b011: //add, sub
               begin
                  Gout = 1'b1;
                  Rin = Xreg;
                  Done = 1'b1;
               end
               default: ;
            endcase
      endcase
   end

   regn reg_0 (BusWires, Rin[0], Clock, R0);
   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);
   regn reg_4 (BusWires, Rin[4], Clock, R4);
   regn reg_5 (BusWires, Rin[5], Clock, R5);
   regn reg_6 (BusWires, Rin[6], Clock, R6);
   regn reg_7 (BusWires, Rin[7], Clock, R7);
   regn reg_A (BusWires, Ain, Clock, A);
   regn #(.n(9)) reg_IR (DIN[15:7], IRin, Clock, IR);

   // alu
   always @(AddSub or A or BusWires)
      begin
      if (!AddSub)
         Sum = A + BusWires;
       else
         Sum = A - BusWires;
      end

   regn reg_G (Sum, Gin, Clock, G);

   // define the internal processor bus
   assign Sel = {Rout, Gout, DINout};

   always @(*)
   begin
      if (Sel == 10'b1000000000)
         BusWires = R0;
      else if (Sel == 10'b0100000000)
         BusWires = R1;
      else if (Sel == 10'b0010000000)
         BusWires = R2;
      else if (Sel == 10'b0001000000)
         BusWires = R3;
      else if (Sel == 10'b0000100000)
         BusWires = R4;
```

```
      else if (Sel == 10'b0000010000)
         BusWires = R5;
      else if (Sel == 10'b0000001000)
         BusWires = R6;
      else if (Sel == 10'b0000000100)
         BusWires = R7;
      else if (Sel == 10'b0000000010)
         BusWires = G;
      else BusWires = DIN;
   end
endmodule

module upcount(Clear, Clock, Q);
   input Clear, Clock;
   output [1:0] Q;
   reg [1:0] Q;

   always @(posedge Clock)
      if (Clear)
         Q <= 2'b0;
      else
         Q <= Q + 1'b1;
endmodule

module dec3to8(W, En, Y);
   input [2:0] W;
   input En;
   output [0:7] Y;
   reg [0:7] Y;

   always @(W or En)
   begin
      if (En == 1)
         case (W)
            3'b000: Y = 8'b10000000;
            3'b001: Y = 8'b01000000;
            3'b010: Y = 8'b00100000;
            3'b011: Y = 8'b00010000;
            3'b100: Y = 8'b00001000;
            3'b101: Y = 8'b00000100;
            3'b110: Y = 8'b00000010;
            3'b111: Y = 8'b00000001;
         endcase
      else
         Y = 8'b00000000;
   end
endmodule

module regn(R, Rin, Clock, Q);
   parameter n = 16;
   input [n-1:0] R;
   input Rin, Clock;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Rin)
         Q <= R;
endmodule
```

```
// 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
module part1 (KEY, SW, LEDR);
   input [1:0] KEY;
   input [17:0] SW;
   output [17:0] LEDR;

   wire Resetn, Manual_Clock, Run, Done;
   wire [15:0] DIN, Bus;

   assign Resetn = KEY[0];
   assign 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
   assign DIN = SW[15:0];
   assign Run = SW[17];

   // module proc(DIN, Resetn, Clock, Run, Done, Bus);
   proc U1 (DIN, Resetn, Manual_Clock, Run, Done, Bus);

   assign LEDR[15:0] = Bus;
   assign LEDR[16] = 1'b0;
   assign LEDR[17] = Done;

endmodule
```

```
// 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.
module part2 (KEY, SW, LEDR);
   input [2:0] KEY;
   input [17:17] SW;
   output [17:0] LEDR;

   wire Done, Resetn, PClock, MClock, Run;
   wire [15:0] DIN, Bus;
   wire [4:0] pc;

   assign Resetn = KEY[0];
   assign PClock = KEY[1];
   assign MClock = KEY[2];
   assign Run = SW[17];

   // module proc(DIN, Resetn, Clock, Run, Done, BusWires);
   proc U1 (DIN, Resetn, PClock, Run, Done, Bus);
   assign LEDR[15:0] = Bus;
   assign LEDR[17] = Done;

   inst_mem U2 (pc, MClock, DIN);
   count5 U3 (Resetn, MClock, pc);

endmodule

module count5 (Resetn, Clock, Q);
   input Resetn, Clock;
   output reg [4:0] Q;

   always @ (posedge Clock)
      if (Resetn == 0)
         Q <= 5'b00000;
      else
         Q <= Q + 1'b1;
 endmodule
```

```
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
```

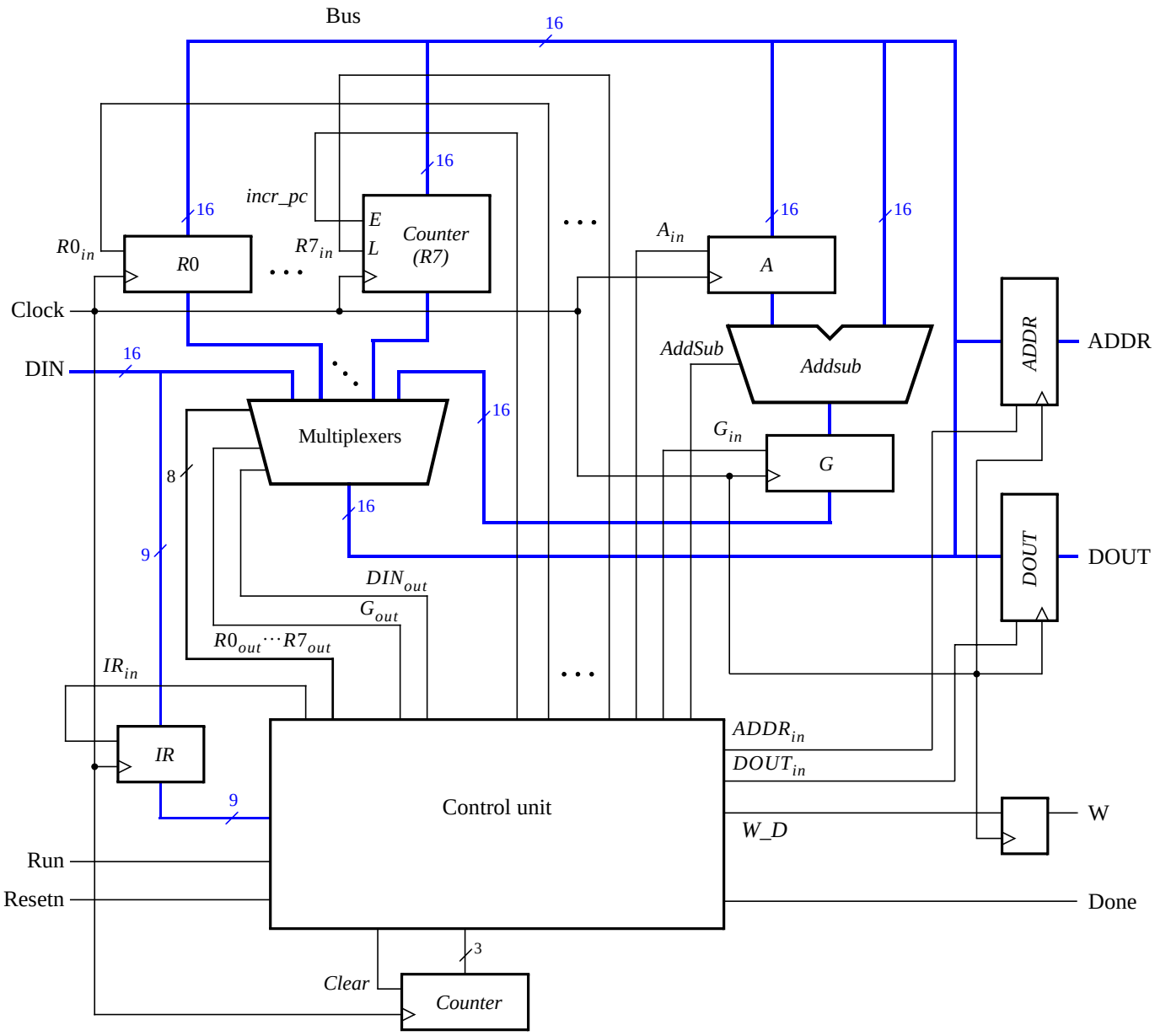


Figure 7. An enhanced version of the processor.

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.

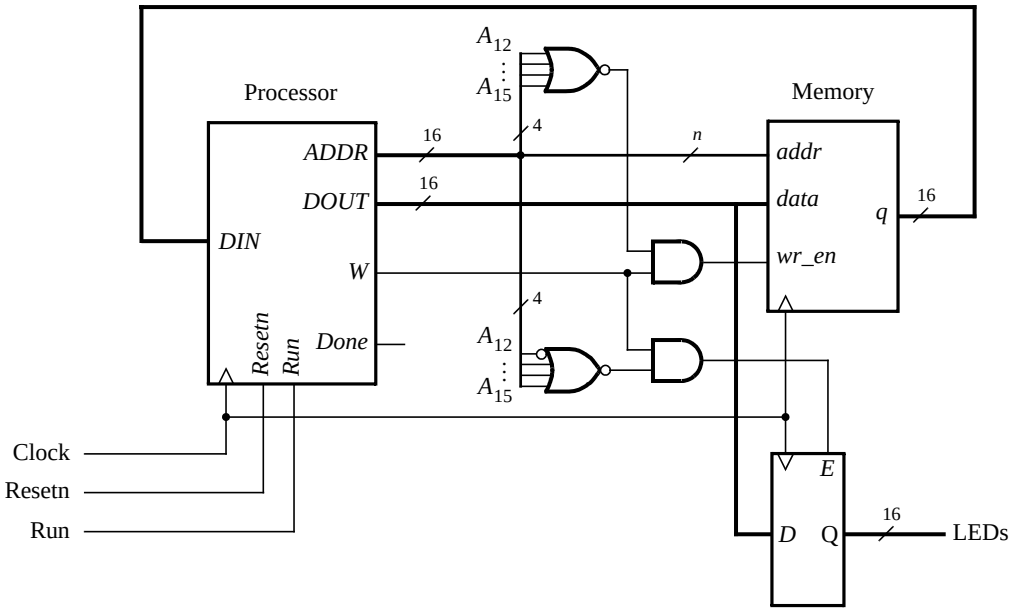


Figure 8. Connecting the enhanced processor to a memory and output register.

1. Create a new Quartus II project for the enhanced version of the processor.

2. Write Verilog 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 Verilog 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 Verilog 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 Verilog 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.

```
module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   input [15:0] DIN;
   input Resetn, Clock, Run;
   output wire [15:0] DOUT;
   output wire [15:0] ADDR;
   output wire W;
   output Done;

   reg [15:0] BusWires;
   reg [0:7] Rin, Rout;
   reg [15:0] Sum;
   reg IRin, ADDRin, Done, DINout, DOUTin, Ain, Gin, Gout, AddSub;
   wire [2:0] Tstep_Q;
   wire [2:0] I;
   wire [0:7] Xreg, Yreg;
   wire [15:0] R0, R1, R2, R3, R4, R5, R6, R7 /* pc */, A, G;
   wire [1:9] IR;
   wire [1:10] Sel; // bus selector
   reg pc_inc, W_D;
   wire Z, Z_D;

   wire Clear = ~Resetn | Done | (~Run & (Tstep_Q == 0));
   upcount Tstep (Clear, Clock, Tstep_Q);
   assign I = IR[1:3];
   dec3to8 decX (IR[4:6], 1'b1, Xreg);
   dec3to8 decY (IR[7:9], 1'b1, 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
   always @(Tstep_Q or I or Xreg or Yreg or Z or Run)
   begin
      Done = 1'b0; Ain = 1'b0; Gin = 1'b0; Gout = 1'b0; AddSub = 1'b0;
      IRin = 1'b0; DINout = 1'b0; DOUTin = 1'b0; ADDRin = 1'b0; W_D = 1'b0;
      Rin = 8'b0; Rout = 8'b0; pc_inc = 1'b0;
      case (Tstep_Q)
         3'b000: // fetch the instruction
            begin
               Rout = 8'b00000001;  // R7 is program counter (pc)
               ADDRin = 1'b1;
               pc_inc = Run; // to increment pc
            end
         3'b001: // wait cycle for synchronous memory
            // in case the instruction turns out to be mvi, read memory
            begin
               Rout = 8'b00000001;  // R7 is program counter (pc)
               ADDRin = 1'b1;
            end
         3'b010: // store DIN in IR
            begin
               IRin = 1'b1;
            end
         3'b011:   //define signals in T3
            case (I)
```

```
            3'b000: // mv Rx,Ry
            begin
               Rout = Yreg;
               Rin = Xreg;
               Done = 1'b1;
            end
            3'b001: // mvi Rx,#D
            begin
               // data is available now on DIN
               DINout = 1'b1;
               Rin = Xreg;
               pc_inc = 1'b1;
               Done = 1'b1;
            end
            3'b010, 3'b011: //add, sub
            begin
               Rout = Xreg;
               Ain = 1'b1;
            end
            3'b100: // ld Rx,[Ry]
            begin
               Rout = Yreg;
               ADDRin = 1'b1;
            end
            3'b101: // st [Ry],Rx
            begin
               Rout = Yreg;
               ADDRin = 1'b1;
            end
            3'b110: // mvnz Rx,Ry
            begin
               if (!Z)
               begin
                  Rout = Yreg;
                  Rin = Xreg;
               end
               else
               begin
                  Rout = 8'b0;
                  Rin = 8'b0;
               end
               Done = 1'b1;
            end
            default: ;
         endcase
      3'b100:   //define signals T4
         case (I)
            3'b010: // add
            begin
               Rout = Yreg;
               Gin = 1'b1;
            end
            3'b011: // sub
            begin
               Rout = Yreg;
               AddSub = 1'b1;
               Gin = 1'b1;
            end
            3'b100: // ld Rx,[Ry]
               ; // wait cycle for synchronous memory
            3'b101: // st [Ry],Rx
            begin
               Rout = Xreg;
               DOUTin = 1'b1;
```

```
                W_D = 1'b1;
             end
             default: ;
          endcase
       3'b101:   //define T5
          case (I)
             3'b010, 3'b011: //add, sub
             begin
                Gout = 1'b1;
                Rin = Xreg;
                Done = 1'b1;
             end
             3'b100: // ld Rx,[Ry]
             begin
                DINout = 1'b1;
                Rin = Xreg;
                Done = 1'b1;
             end
             3'b101: // st [Ry],Rx
             begin
                Done = 1'b1; // wait cycle for synhronous memory
             end
             default: ;
          endcase
       default: ;
    endcase
end

regn reg_0 (BusWires, Rin[0], Clock, R0);
regn reg_1 (BusWires, Rin[1], Clock, R1);
regn reg_2 (BusWires, Rin[2], Clock, R2);
regn reg_3 (BusWires, Rin[3], Clock, R3);
regn reg_4 (BusWires, Rin[4], Clock, R4);
regn reg_5 (BusWires, Rin[5], Clock, R5);
regn reg_6 (BusWires, Rin[6], Clock, R6);

// R7 is program counter
// module pc_count(R, Resetn, Clock, E, L, Q);
pc_count pc (BusWires, Resetn, Clock, pc_inc, Rin[7], R7);

regn reg_A (BusWires, Ain, Clock, A);
regn reg_DOUT (BusWires, DOUTin, Clock, DOUT);
regn reg_ADDR (BusWires, ADDRin, Clock, ADDR);
regn #(.n(9)) reg_IR (DIN[15:7], IRin, Clock, IR);

flipflop reg_W (W_D, Resetn, Clock, W);

// alu
always @(AddSub or A or BusWires)
   begin
   if (!AddSub)
      Sum = A + BusWires;
    else
      Sum = A - BusWires;
   end

regn reg_G (Sum, Gin, Clock, G);

assign Z_D = (G == 0) ? 1'b1 : 1'b0;
flipflop reg_Z (Z_D, Resetn, Clock, Z);

// define the internal processor bus
assign Sel = {Rout, Gout, DINout};
```

```
   always @(*)
   begin
      if (Sel == 10'b1000000000)
         BusWires = R0;
      else if (Sel == 10'b0100000000)
         BusWires = R1;
      else if (Sel == 10'b0010000000)
         BusWires = R2;
      else if (Sel == 10'b0001000000)
         BusWires = R3;
      else if (Sel == 10'b0000100000)
         BusWires = R4;
      else if (Sel == 10'b0000010000)
         BusWires = R5;
      else if (Sel == 10'b0000001000)
         BusWires = R6;
      else if (Sel == 10'b0000000100)
         BusWires = R7;
      else if (Sel == 10'b0000000010)
         BusWires = G;
      else BusWires = DIN;
   end
endmodule

module upcount(Clear, Clock, Q);
   input Clear, Clock;
   output [2:0] Q;
   reg [2:0] Q;

   always @(posedge Clock)
      if (Clear)
         Q <= 3'b0;
      else
         Q <= Q + 1'b1;
endmodule

module pc_count(R, Resetn, Clock, E, L, Q);
   input [15:0] R;
   input Resetn, Clock, E, L;
   output [15:0] Q;
   reg [15:0] Q;

   always @(posedge Clock)
      if (!Resetn)
         Q <= 16'b0;
      else if (L)
         Q <= R;
      else if (E)
         Q <= Q + 1'b1;
endmodule

module dec3to8(W, En, Y);
   input [2:0] W;
   input En;
   output [0:7] Y;
   reg [0:7] Y;

   always @(W or En)
   begin
      if (En == 1)
         case (W)
            3'b000: Y = 8'b10000000;
            3'b001: Y = 8'b01000000;
            3'b010: Y = 8'b00100000;
```

```
            3'b011: Y = 8'b00010000;
            3'b100: Y = 8'b00001000;
            3'b101: Y = 8'b00000100;
            3'b110: Y = 8'b00000010;
            3'b111: Y = 8'b00000001;
         endcase
      else
         Y = 4'b00000000;
   end
endmodule

module regn(R, Rin, Clock, Q);
   parameter n = 16;
   input [n-1:0] R;
   input Rin, Clock;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Rin)
         Q <= R;
endmodule
```

```
// 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.
module part3 (KEY, SW, Clock, LEDR, DOUT, ADDR, W, Done, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0, LEDG);
   input [0:0] KEY;
   input [17:17] SW;
   input Clock;
   output [15:0] LEDR;
   output [15:0] DOUT, ADDR;
   output W;
   output Done;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [8:0] LEDG;
   assign HEX7 = 7'b0000000;
   assign HEX6 = 7'b0000000;
   assign HEX5 = 7'b0000000;
   assign HEX4 = 7'b0000000;
   assign HEX3 = 7'b0000000;
   assign HEX2 = 7'b0000000;
   assign HEX1 = 7'b0000000;
   assign HEX0 = 7'b0000000;
   assign LEDG = 9'b000000000;

   wire [15:0] DIN;
   wire Sync, Run;
   wire inst_mem_cs, LED_reg_cs;
   wire [15:0] LED_reg;

   // synchronize the Run input
   flipflop U1 (SW[17], KEY[0], Clock, Sync);
   flipflop U2 (Sync, KEY[0], Clock, Run);

   // module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   proc U3 (DIN, KEY[0], Clock, Run, DOUT, ADDR, W, Done);

   assign inst_mem_cs = (ADDR[15:12] == 4'h0);
   // module inst_mem ( data, wren, address, clock, q);
   inst_mem U4 (DOUT, inst_mem_cs & W, ADDR[6:0], Clock, DIN);

   assign LED_reg_cs = (ADDR[15:12] == 4'h1);
   // module regn(R, Rin, Clock, Q);
   regn U6 (DOUT, LED_reg_cs & W, Clock, LED_reg);
   assign LEDR[15:0] = LED_reg[15:0];

endmodule
```

```
module flipflop (D, Resetn, Clock, Q);
   input D, Resetn, Clock;
   output Q;
   reg Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= 1'b0;
      else
         Q <= D;
endmodule
```

```
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;
```

```
// Reset with KEY[0]. SW[17] is Run. Set delay using SW15-0
module part4 (KEY, SW, Clock, HEX7, HEX6, HEX5, HEX4, HEX3,
   HEX2, HEX1, HEX0, LEDR, LEDG, DOUT, ADDR, W, Done);
   input [0:0] KEY;
   input [17:17] SW;
   input Clock;

   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [15:0] LEDR;
   output [8:0] LEDG;
   output [15:0] DOUT, ADDR;
   output W;
   output Done;

   wire [15:0] DIN;
   wire Sync, Run;
   wire inst_mem_cs, seg7_cs, LED_reg_cs;
   wire [15:0] LED_reg, inst_mem_q;

   // synchronize the Run input
   flipflop U1 (SW[17], KEY[0], Clock, Sync);
   flipflop U2 (Sync, KEY[0], Clock, Run);

   // module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   proc U3 (DIN, KEY[0], Clock, Run, DOUT, ADDR, W, Done);

   assign inst_mem_cs = (ADDR[15:12] == 4'h0);
   // module inst_mem ( data, wren, address, clock, q);
   inst_mem U4 (DOUT, inst_mem_cs & W, ADDR[6:0], Clock, inst_mem_q);

   assign DIN = inst_mem_q;

   assign LED_reg_cs = (ADDR[15:12] == 4'h1);
   // module regn(R, Rin, Clock, Q);
   regn U6 (DOUT, LED_reg_cs & W, Clock, LED_reg);
   assign LEDR[15:0] = LED_reg[15:0];
   assign LEDG = 9'b0;

   assign seg7_cs = (ADDR[15:12] == 4'h2);
   seg7_scroll U5 (DOUT[6:0], ADDR[2:0], seg7_cs & W, KEY[0], Clock,
      HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

endmodule
```

```
// Data written to registers R0 to R7 are sent to the HEX digits
module seg7_scroll (Data, Addr, Sel, Resetn, Clock, HEX7, HEX6, HEX5, HEX4,
   HEX3, HEX2, HEX1, HEX0);
   input [0:6] Data;
   input [2:0] Addr;
   input Sel, Resetn, Clock;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   wire [0:6] R0, R1, R2, R3, R4, R5, R6, R7;

   regne reg_R0 (Data, Clock, Resetn, Sel & (Addr == 3'b000), R0);
   regne reg_R1 (Data, Clock, Resetn, Sel & (Addr == 3'b001), R1);
   regne reg_R2 (Data, Clock, Resetn, Sel & (Addr == 3'b010), R2);
   regne reg_R3 (Data, Clock, Resetn, Sel & (Addr == 3'b011), R3);
   regne reg_R4 (Data, Clock, Resetn, Sel & (Addr == 3'b100), R4);
   regne reg_R5 (Data, Clock, Resetn, Sel & (Addr == 3'b101), R5);
   regne reg_R6 (Data, Clock, Resetn, Sel & (Addr == 3'b110), R6);
   regne reg_R7 (Data, Clock, Resetn, Sel & (Addr == 3'b111), R7);

   assign HEX7 = R0;
   assign HEX6 = R1;
   assign HEX5 = R2;
   assign HEX4 = R3;
   assign HEX3 = R4;
   assign HEX2 = R5;
   assign HEX1 = R6;
   assign HEX0 = R7;

endmodule

module regne (R, Clock, Resetn, E, Q);
   parameter n = 7;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= {n{1'b0}};
      else if (E)
         Q <= R;
endmodule
```

```
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;
```

```
// Reset with KEY[0]. SW[17] is Run
// The processor executes the instructions in the file inst_mem.mif
module part5 (KEY, SW, Clock, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0,
   LEDR, LEDG, DOUT, ADDR, W, Done);
   input [0:0] KEY;
   input [17:0] SW;
   input Clock;
   output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output [15:0] LEDR;
   output [8:0] LEDG;
   output [15:0] DOUT, ADDR;
   output W;
   output Done;

   reg [15:0] DIN;
   wire Sync, Run;
   wire inst_mem_cs, seg7_cs, LED_reg_cs;
   wire [15:0] LED_reg, SW_reg, inst_mem_q;

   // synchronize the Run input
   flipflop U1 (SW[17], KEY[0], Clock, Sync);
   flipflop U2 (Sync, KEY[0], Clock, Run);

   // module proc(DIN, Resetn, Clock, Run, DOUT, ADDR, W, Done);
   proc U3 (DIN, KEY[0], Clock, Run, DOUT, ADDR, W, Done);

   assign inst_mem_cs = (ADDR[15:12] == 4'h0);
   // module inst_mem ( data, wren, address, clock, q);
   inst_mem U4 (DOUT, inst_mem_cs & W, ADDR[6:0], Clock, inst_mem_q);

   always @ (inst_mem_q or SW_reg or inst_mem_cs)
   if (inst_mem_cs == 1'b1)
      DIN = inst_mem_q;
   else
      DIN = SW_reg;

   assign LED_reg_cs = (ADDR[15:12] == 4'h1);
   // module regn(R, Rin, Clock, Q);
   regn U6 (DOUT, LED_reg_cs & W, Clock, LED_reg);
   assign LEDR[15:0] = LED_reg[15:0];
   assign LEDG = 9'b0;

   assign seg7_cs = (ADDR[15:12] == 4'h2);
   seg7_scroll U5 (DOUT[6:0], ADDR[2:0], seg7_cs & W, KEY[0], Clock, HEX7, HEX6,
      HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

   // module regn(R, Rin, Clock, Q);
   regn U7 (SW[15:0], 1'b1, Clock, SW_reg);

endmodule
```

```
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;
```