VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

LOGIC DESIGN PROJECT REPORT

SIMPLE PROCESSOR ON BOARD DE2i-150

Major: Computer Engineering

LOGIC DESIGN PROJECT COMMITTEE 5

INSTRUCTOR: NGUYEN TRAN HUU NGUYEN

SECRETARY: PHAM HOANG ANH
REVIEWER: NGUYEN CAO TRI

STUDENTS: HUYNH TRUNG NHAT
DO HUU THANH THIEN

Winter Semester, AY 2022-2023

Contents

List of Tables

List of Figures

Chapter 1

Switches, Lights, and Multiplexers

1.1 Introduction

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 on the DE-series boards as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

1.2 Part I

REQUIREMENT

1. Create a new Quartus project for your circuit. Select the target chip that corresponds to your DE-series board. Refer to Table 1 for a list of devices.
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 your DE-series board, as discussed above. Compile the project.
4. Download the compiled circuit into the FPGA chip by using the Quartus Programmer tool (the procedure for using the Programmer tool is described in the tutorial Quartus Introduction). Test the functionality of the circuit by toggling the switches and observing the LEDs.

SOLUTION

```
module part2(out, X, Y,S);
    output  [3:0]out;
    input       [3:0]X,Y;
    input       S;

    assign out = ({4{S}}&X) | ({4{~S}}&Y);
endmodule

```

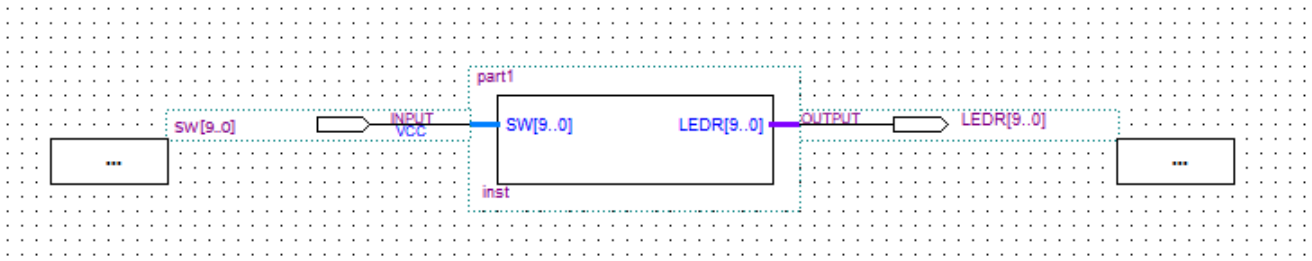


Figure 1.1: LAB 1: Schematic for part I

1.3 Part II

REQUIREMENT

1. Create a new Quartus project for your circuit.
2. Include your Verilog file for the four-bit wide 2-to-1 multiplexer in your project. Use switch SW_9 as the s input, switches SW_{3-0} as the X input and SW_{7-4} as the Y input. Display the value of the input s on $LEDR_9$, connect the output M to $LEDR_{3-0}$, and connect the unused LEDR lights to the constant value 0.
3. Include in your project the required pin assignments for your DE-series board. As discussed in Part I, these assignments ensure that the ports of your Verilog code will use the pins on the FPGA chip that are connected to the SW switches and LEDR lights.
4. Compile the project, and then download the resulting circuit into the FPGA chip. Test the functionality of the four-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs

SOLUTION

```
module part2(out, X, Y,S);
   output   [3:0]out;
   input    [3:0]X,Y;
   input    S;

   assign out = ({4{S}}&X) | ({4{~S}}&Y);

endmodule

```

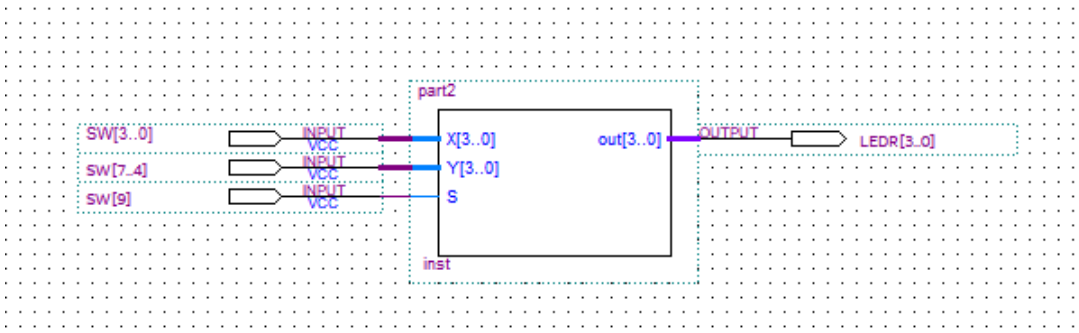


Figure 1.2: LAB 1: Schematic for part II

1.4 Part III

REQUIREMENT

1. Create a new Quartus project for your circuit.
2. Create a Verilog module for the two-bit wide 4-to-1 multiplexer. Connect its select inputs to switches SW_{9-8}, and use switches SW_{7-0} to provide the four 2-bit inputs U to X. Connect the output M to the red lights $LEDR_{1-0}$.
3. Include in your project the required pin assignments for your DE-series board. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the two-bit wide 4-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs U to X can be properly selected as the output M.

SOLUTITON

```
module part3(M,U,V,W,X,S_1,S_0);
    input   [1:0]U,V,W,X;
    input      S_1,S_0;
    output  [1:0]M;

    assign  M= ({2{S_0&S_1}}&X)|
                      ({2{~S_0&S_1}}&W)|
                      ({2{S_0&~S_1}}&V)|
                      ({2{~S_0&~S_1}}&U);

endmodule

```

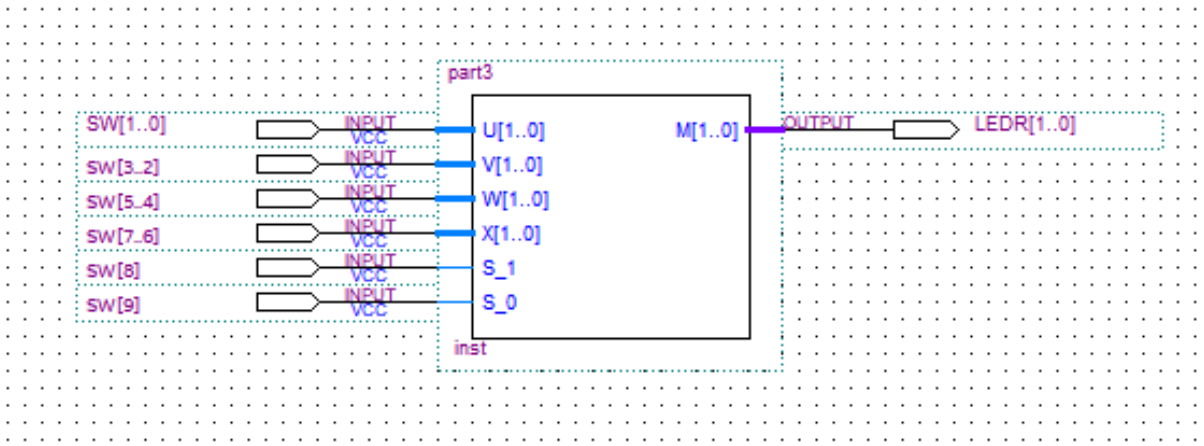


Figure 1.3: LAB 1: Schemactic for part III

1.5 Part IV

REQUIREMENT

1. The objective of this part is to display a character on a 7-segment display. This decoder produces seven outputs that are used to display a character on a 7-segment display. Table 2 lists the characters that should be displayed for each valuation of c1c0 for your DE-series board.

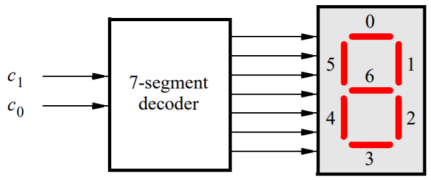


Figure 6: A 7-segment decoder.

c_1c_0	DE10-Lite	DE0-CV	DE1-SoC	DE2-115
00	d	d	d	d
01	E	E	E	E
10	1	0	1	2
11	0			

Table 2: Character codes for the DE-series boards.

Figure 1.4: LAB 1: Hint for part IV

SOLUTION

```
module  part4(OUT,C_IN);
    input   [1:0]C_IN;
    output  [6:0]OUT;

    assign  OUT = (C_IN==0) ? 7'b0100001:
                  (C_IN==1) ? 7'b0000110:
                  (C_IN==2) ? 7'b0100100:7'b1111011;
endmodule

```

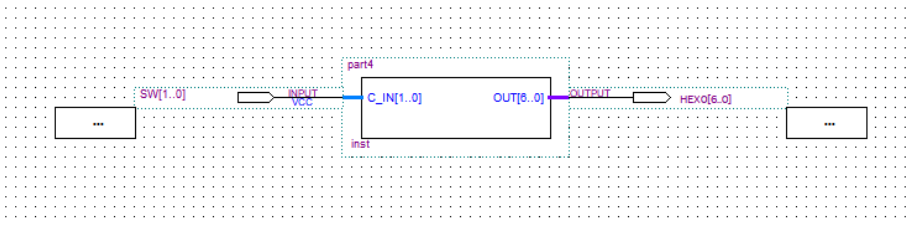


Figure 1.5: LAB 1: Schematic for part IV

1.6 Part V

REQUIREMENT

1. Consider the circuit shown in Figure 7. It uses a two-bit wide 4-to-1 multiplexer to enable the selection of four characters that are displayed on a 7-segment display. Using the 7-segment decoder from Part IV this circuit can display the characters d, E, 0, 1, 2, or 'blank' depending on your DE-series board. The character codes are set according to Table 2 by using the switches SW7-0, and a specific character is selected for display by setting the switches SW_{9-8}.
2. Note that we have used the circuits from Parts III and IV as subcircuits in this code. The purpose of your circuit is to display any word on the four 7-segment displays that is composed of the characters in Table 2, and be able to rotate this word in a circular fashion across the displays when the switches SW9-8 are toggled. As an example, if the displayed word is dE10, then your circuit should produce the output patterns illustrated in Table 3.

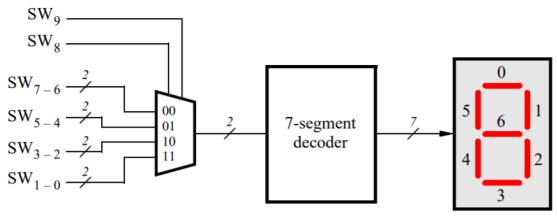


Figure 7: A circuit that can select and display one of four characters.

SW_{9-8}	Characters			
00	d	E	1	0
01	E	1	0	d
10	1	0	d	E
11	0	d	E	1

Table 3: Rotating the word dE10 on four displays.

Figure 1.6: LAB 1: Hint part V

SOLUTION In this part of Lab 1, we reuse part 3 and part 4 block to implement the circuit.

Part 3 block with 4 fix input and Pin S_0, S_1 to select the input

Part 4 block use the output of the part 3 as the input and generate signal for four 7-segment leds in DE2i-board.

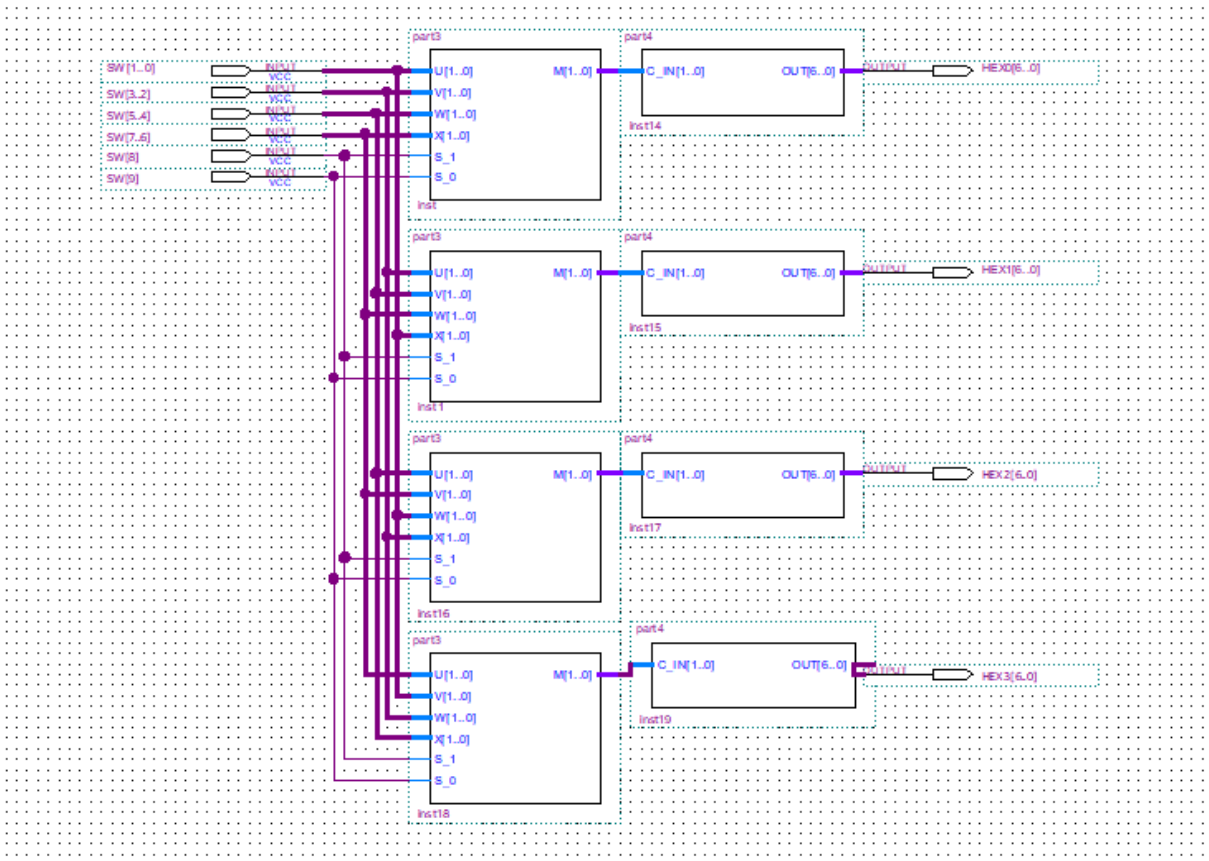


Figure 1.7: LAB 1: Schematic for part V

1.7 Part VI

REQUIREMENT

1. Extend your design from Part V so that is uses all 7-segment displays on your DE-series board. Your circuit needs to display a three- or four-letter word, corresponding to Table 2, using 'blank' characters for unused displays. Implement rotation of this word from right-to-left as indicated in Table 4 and Table 5.
2. Note that for the DE10-Lite you will need to use 3-bit codes for your characters, because five characters are needed when including the 'blank' character (your 7-segment decoder will have to use 3-bit codes, and you will need to use 3-bit wide 6-to-1 multiplexers).

SOLUTION

```
module part6(SW,HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7);
  input    [2:0]SW;
  output   [6:0]HEX0,HEX1,HEX2, HEX3, HEX4, HEX5, HEX6, HEX7;
  wire     [55:0]hex0,hex1,hex2,hex3,hex4,hex5,hex6,hex7;
  wire     [55:0]temp;

  assign hex0 ={7'b1111111,7'b1111111,7'b1111111,7'b1111111,
    7'b0100001,7'b0000110,7'b1111001,7'b1111111};
  assign hex1 = {7'b1111111,7'b1111111,7'b1111111,7'b0100001,
    7'b0000110,7'b1111001,7'b1111111,7'b1111111};
  assign hex2 = {7'b1111111,7'b1111111,7'b0100001,7'b0000110,
    7'b1111001,7'b1111111,7'b1111111,7'b1111111};
  assign hex3 = {7'b1111111,7'b0100001,7'b0000110,7'b1111001,
    7'b1111111,7'b1111111,7'b1111111,7'b1111111};
  assign hex4 = {7'b0100001,7'b0000110,7'b1111001,7'b1111111,
    7'b1111111,7'b1111111,7'b1111111,7'b1111111};
  assign hex5 = {7'b0000110,7'b1111001,7'b1111111,7'b1111111,
    7'b1111111,7'b1111111,7'b1111111,7'b0100001};
  assign hex6 = {7'b1111001,7'b1111111,7'b1111111,7'b1111111,
    7'b1111111,7'b1111111,7'b0100001,7'b0000110};
  assign hex7 = {7'b1111111,7'b1111111,7'b1111111,7'b1111111,
    7'b1111111,7'b0100001,7'b0000110,7'b1111001};

  assign temp = (SW==0)?hex0:
                (SW==1)?hex1:
                (SW==2)?hex2:
                (SW==3)?hex3:
                (SW==4)?hex4:
                (SW==5)?hex5:
                (SW==6)?hex6:hex7;

  assign HEX0 = temp[6:0];
```

```
33    assign HEX1 = temp[13:7];
34    assign HEX2 = temp[20:14];
35    assign HEX3 = temp[27:21];
36    assign HEX4 = temp[34:28];
37    assign HEX5 = temp[41:35];
38    assign HEX6 = temp[48:42];
39    assign HEX7 = temp[55:49];
40
41 endmodule
42
```

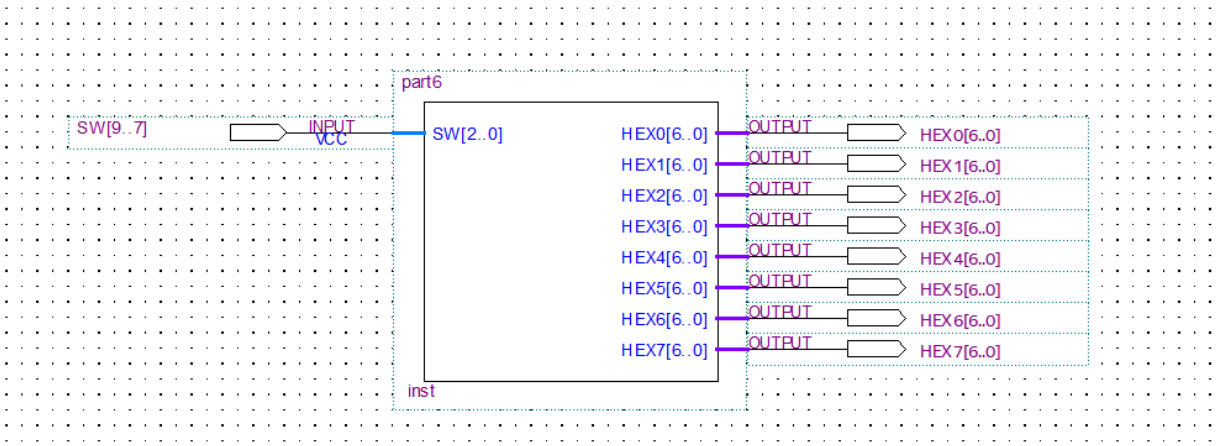


Figure 1.8: LAB 1: Schematic for part 6

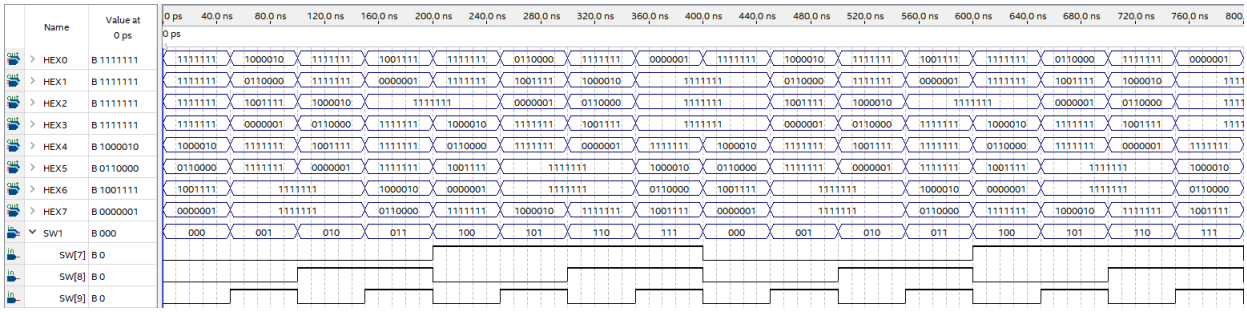


Figure 1.9: LAB 1: Simulation Result for part 6

Chapter 2

Numbers and Displays

2.1 Introduction

This is an exercise in designing combinational circuits that can perform binary-to-decimal number conversion and binary-coded-decimal (BCD) addition.
In previous parts, we design and implemented:

- Two 7-segment displays inputed by the switches SW_{7-0}.
- Two-digit decimal represented by 7-segment displays,inputed by the switches SW_{3-0}(includes a comparator).
- full adder.
- Circuit that adds the two BCD digits.

2.2 Part I

REQUIREMENT

1. We wish to display on the 7-segment displays $HEX1$ and $HEX0$ the values set by the switches SW_{7-0}. Let the values denoted by SW_{7-4} and SW_{3-0} be displayed on $HEX1$ 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.
2. Create a new project which will be used to implement the desired circuit on your Intel FPGA DE-series board. The intent of this exercise is to manually derive the logic functions needed for the 7-segment displays. Therefore, you should use only simple Verilog assign statements in your code and specify each logic function as a Boolean expression.
3. 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. Make sure to include the necessary
4. Compile the project and download the compiled circuit into the FPGA chip. pin assignments.

SOLUTION

```
1 module part1(SEG7_IN, SEG7_OUT);
2   input    [3:0]SEG7_IN;
3   output   [6:0]SEG7_OUT;
4   assign SEG7_OUT[6] =    ~SEG7_IN[3] & ~SEG7_IN[2]
5                         + SEG7_IN[2] &  SEG7_IN[1] & SEG7_IN[0];

6   assign SEG7_OUT[5] =     SEG7_IN[2] &  SEG7_IN[1] & SEG7_IN[0]
7                         +~SEG7_IN[3] & ~SEG7_IN[2] & SEG7_IN[0]
8                         +~SEG7_IN[3] &  SEG7_IN[2] & SEG7_IN[1];

9   assign SEG7_OUT[4] =     SEG7_IN[0]
10                        + SEG7_IN[2] & ~SEG7_IN[1];
11  assign SEG7_OUT[3] =    ~SEG7_IN[3] & ~SEG7_IN[2] & ~SEG7_IN[1] &
     SEG7_IN[0]
12                        + SEG7_IN[2] & ~SEG7_IN[1] & ~SEG7_IN[0]
13                        + SEG7_IN[2] &  SEG7_IN[1] &  SEG7_IN[0];

14  assign SEG7_OUT[2] =    ~SEG7_IN[3] & ~SEG7_IN[2] &  SEG7_IN[1] &
    ~SEG7_IN[0];
15  assign SEG7_OUT[1] =    SEG7_IN[3] & ~SEG7_IN[2] & ~SEG7_IN[1];
16  assign SEG7_OUT[0] =    ~SEG7_IN[3] & ~SEG7_IN[2] & ~SEG7_IN[1] &
     SEG7_IN[0]
17                        + SEG7_IN[2] & ~SEG7_IN[1] & ~SEG7_IN[0];
18 endmodule
```

2.3 Part II

REQUIREMENT

1. 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 $D = d_1d_0$. The table below shows the required output values. A partial design of this circuit is given in the figure below.

$v_3v_2v_1v_0$	d_1	d_0
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

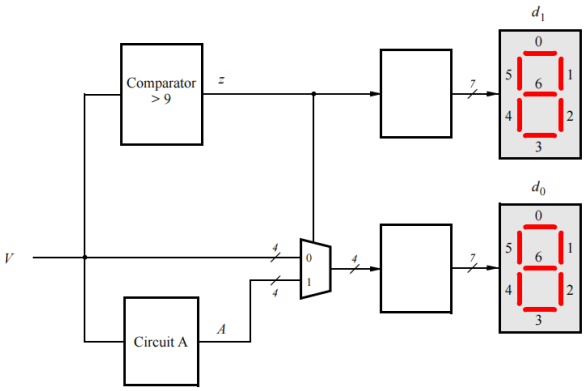


Figure 2.1: LAB 2: Hint for part II

2. For the input values V 9, **the circuit A** does not matter, because the multiplexer in Figure above just selects V in these cases. But for the input values V > 9, the multiplexer will select A.

SOLUTION

Circuit A is implemented using the hint of the requirement above.

```
module circuitA(A_in, A_out);
   input    [3:0]A_in;
   output   [3:0]A_out;

   assign   A_out[3] = 0;
   assign   A_out[2] = A_in[2]& A_in[1];
   assign   A_out[1] = A_in[2]&~A_in[1];
   assign   A_out[0] = A_in[2]& A_in[0]
                     +~A_in[2]&~A_in[0];

endmodule

```

comparew9 indicate if the input greater than 9.

```
module multi_4bits(out, in0, in1, s);
   input    [3:0]in0,in1;
   input    s;
   output   [3:0]out;

   assign out = (in0&{4{~s}})|(in1&{4{s}});
endmodule

```

one_to_4btis is used for translate the output of comparew9 block from 1 bits to 4 bits

```
module comparew9(com_in, com_out);
   input    [3:0]com_in;
   output   com_out;
   assign   com_out = com_in[3] & (com_in[2]|com_in[1]);
endmodule

```

multi_4bits is user to choose the signal from **circuitA** and input, if input > 9, choose the result of block **circuitA.**

```
module one_to_4bits(in,out);
   input    in;
   output   [3:0]out;
   assign   out = (4'b0000) | in;
endmodule

```

Finally, we wired these parts together.

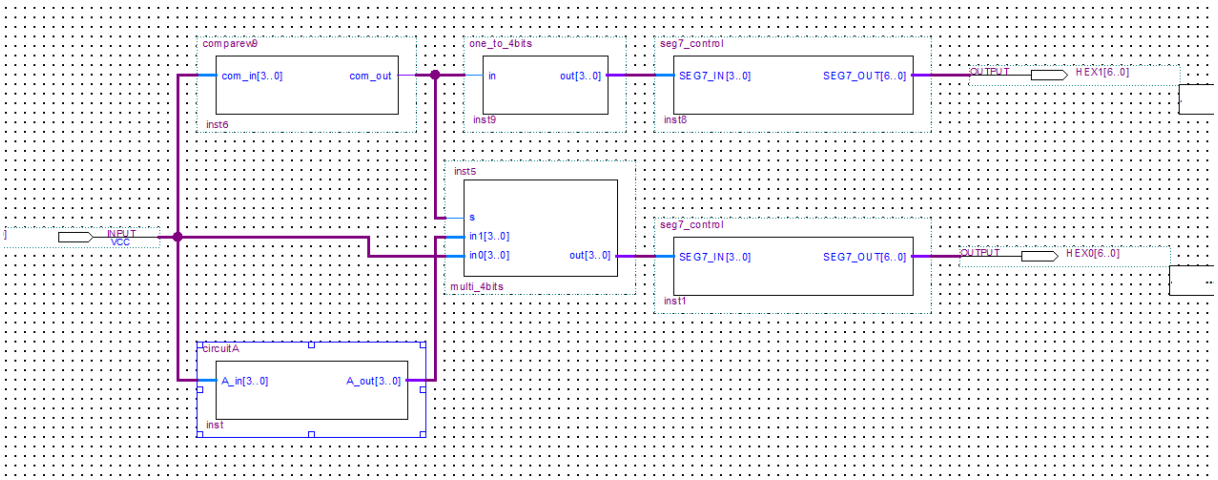


Figure 2.2: LAB 2: Schematic for part II

2.4 Part III

REQUIREMENT

1. Create a new Quartus 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 outputs of the adder, c_{out} and S, to the red lights $LEDR$
3. Include the necessary pin assignments for your DE-series board, compile the circuit, and download it into the FPGA chip.

SOLUTION

```
1  module part3(A,B,C_IN,SUM,C_OUT);
2    input    [3:0]A,B;
3    input    C_IN;
4    output   [3:0]SUM;
5    output   C_OUT;
6
7    wire     C_1, C_2, C_3;
8
9    full_adder inst1 (.a(A[0]), .b(B[0]), .c_i(C_IN), .s(SUM[0]),
     .c_o(C1));
10   full_adder inst2 (.a(A[1]), .b(B[1]), .c_i(C1), .s(SUM[1]),
     .c_o(C2));
11   full_adder inst3 (.a(A[2]), .b(B[2]), .c_i(C2), .s(SUM[2]),
     .c_o(C3));
12   full_adder inst4 (.a(A[3]), .b(B[3]), .c_i(C3), .s(SUM[3]),
     .c_o(C_OUT));
13
14 endmodule
15
16 module full_adder(a,b,c_i,s,c_o);
17   input    a,b,c_i;
18   output   s, c_o;
19
20   assign   s     = c_i ^ (a^b);
21   assign   c_o   = (c_i & (a^b)) | (b & ~(a^b));
22
23 endmodule
24
```

2.5 Part IV

REQUIREMENT

1. In part II we discussed the conversion of binary numbers into decimal digits. For this part you are to design a circuit that has two decimal digits, X and Y , as inputs. Each decimal digit is represented as a 4-bit number. In technical literature this is referred to as the binary coded decimal (BCD) representation.
2. You are to design a circuit that adds the two BCD digits. The inputs to your circuit are the numbers X and Y , plus a carry-in, c_{in}. When these inputs are added, the result will be a 5-bit binary number. But this result is to be displayed on 7-segment displays as a two-digit BCD sum S_1S_0.

SOLUTION

In this design, we use 2 block, namely: **sum4bits** and **display7SEG**.

Sum4bits is the combination of 4 full-adder block.

```
module sum4bits(A,B,C_IN,SUM,C_OUT);
   input    [3:0]A,B;
   input    C_IN;
   output   [3:0]SUM;
   output   C_OUT;
   wire     C_1, C_2, C_3;
   full_adder inst1 (.a(A[0]), .b(B[0]), .c_i(C_IN),
                .s(SUM[0]), .c_o(C1));
   full_adder inst2 (.a(A[1]), .b(B[1]), .c_i(C1),
                .s(SUM[1]), .c_o(C2));
   full_adder inst3 (.a(A[2]), .b(B[2]), .c_i(C2),
                .s(SUM[2]), .c_o(C3));
   full_adder inst4 (.a(A[3]), .b(B[3]), .c_i(C3),
                .s(SUM[3]), .c_o(C_OUT));
endmodule
module full_adder(a,b,c_i,s,c_o);
   input    a,b,c_i;
   output   s, c_o;
   assign   s     = c_i ^ (a^b);
   assign   c_o   = (c_i & (a^b)) | (b & ~(a^b));
endmodule

```

display7SEG

```
module display7SEG(num, carry , hex0, hex1);
   input     [3:0]num;
   input     carry;
   output    [6:0]hex0, hex1;
   wire      [3:0]A_out;
   wire      [3:0]num_plus6;
   wire      select;
   wire      com_out;
   wire      [3:0]seg7_1_in;
   wire      [3:0]seg7_0_in;
   circuitA        inst0 (.A_in(num), .A_out(A_out));
   comparew9       inst1 (.com_in(num),
                     .com_out(com_out));
   or select_char_1      (select,carry,com_out);
   multi_4bits     inst4 (.out(seg7_0_in), .in0(num),
                     .in1(A_out), .s(select));
   one_to_4bits    inst5 (.out(seg7_1_in),
                     .in(select));
   seg7_control    inst6 (.SEG7_IN(seg7_0_in),
                     .SEG7_OUT(hex0));
   seg7_control    inst7 (.SEG7_IN(seg7_1_in),
                     .SEG7_OUT(hex1));
endmodule

```

In **module part4**

```
module part4(X,Y,HEX0,HEX1);
   input     [3:0]X,Y;
   output    [6:0]HEX0,HEX1;
   wire [3:0]num;
   wire carry;
   sum4bits    inst0 (.A(X), .B(Y), .C_IN(0), .SUM(num),
   .C_OUT(carry));
   display7SEG inst1 (.num(num), .carry(carry), .hex0(HEX0),
   .hex1(HEX1));
endmodule

```

This is our design

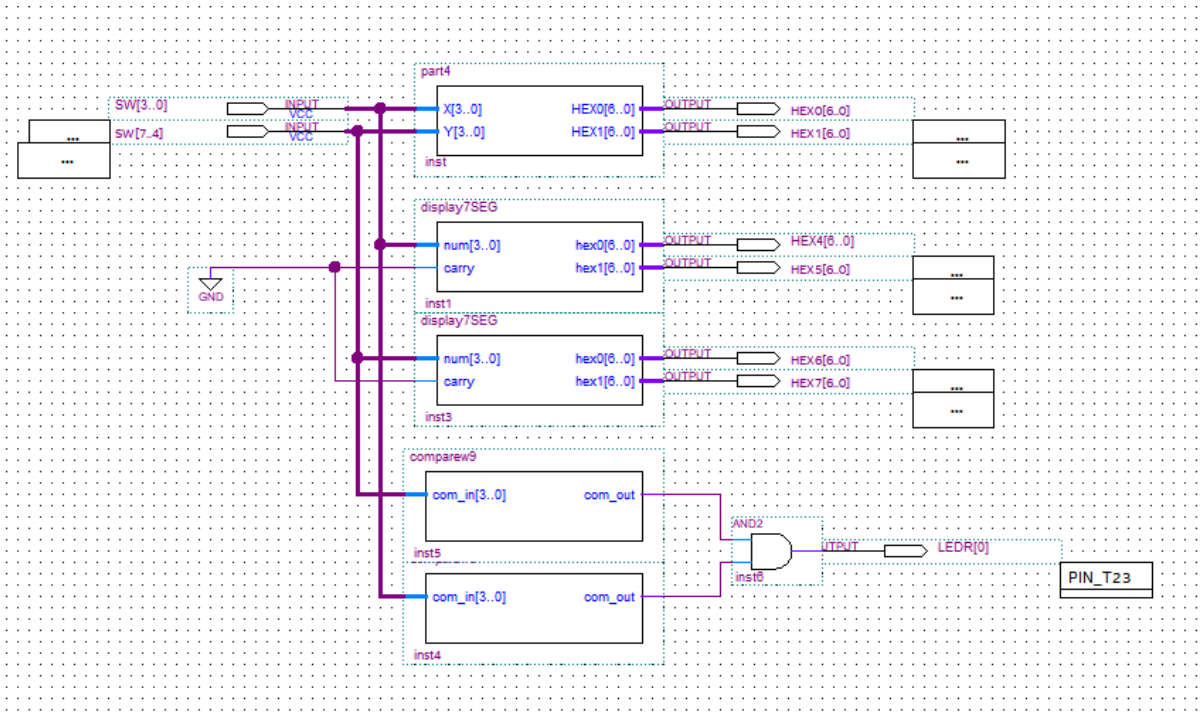


Figure 2.3: LAB 2: Schematic for part IV

Chapter 3

Latches, Flip-flops, and Registers

3.1 Introduction

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

Intel 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.

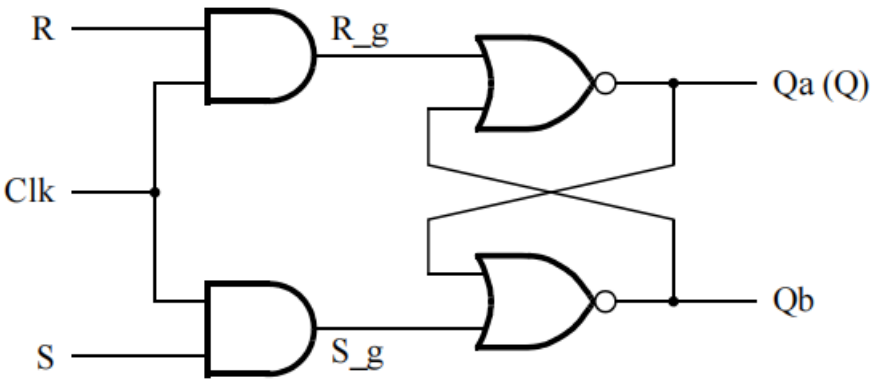


Figure 3.1: LAB 3: A logic design for D Latch

3.2 Part I

REQUIREMENT

1. Create a new Quartus project for your DE-series board.
2. Generate a Verilog file for the RS latch. Using **assign** or **primitive gate**.
3. Compile the code. Use the Quartus RTL Viewer tool to examine the gate-level circuit produced from the code, and use the Technology Map Viewer tool to verify that the latch is implemented.

SOLUTION

Using only assign

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

Using the primitive gate

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

VERIFICATION

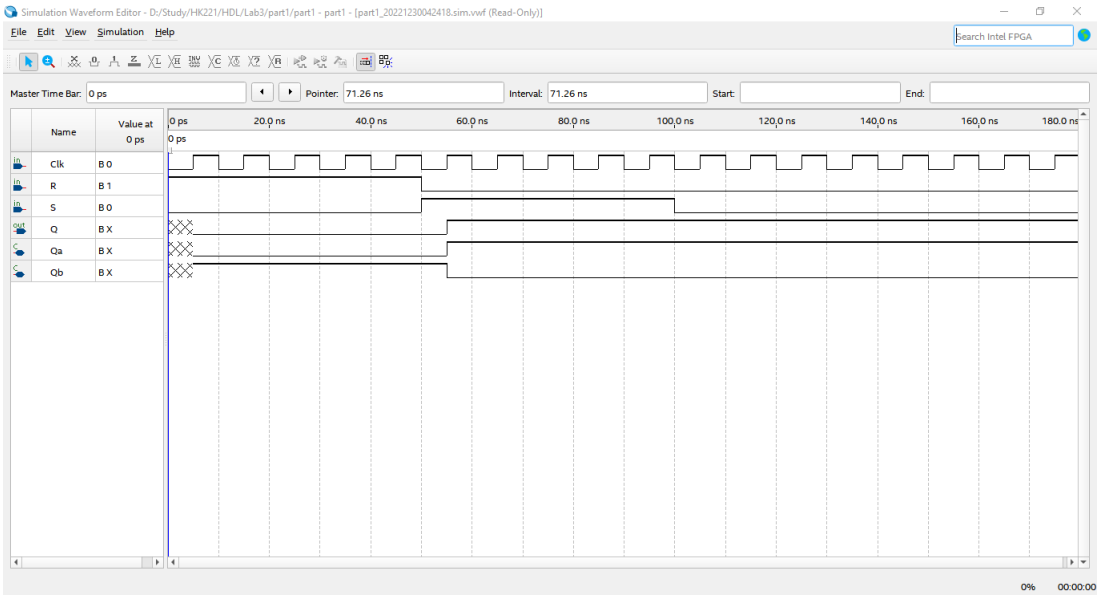


Figure 3.2: LAB 3: Simulation Result of part 1

3.3 Part II

REQUIREMENT

1. Create a new Quartus project. Generate a Verilog file using the style of code in Figure 3 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. 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.

SOLUTION

```
module part2(CLK, D, Q, Q_n);
   input    D, CLK;
   output   Q, Q_n;

   wire Qa, Qb /* synthesis keep */;
   //RS_latch inst0(CLK, ~D, D, Qa, Qb);
   wire s_g, r_g;

   assign s_g = ~(D & CLK);
   assign r_g = ~(~D & CLK);
   assign Qa  = ~(Qb & s_g);
   assign Qb  = ~(Qa & r_g);

   assign Q = Qa;
   assign Q_n = Qb;

endmodule

```

3.4 Part III

REQUIREMENT

1. Create a new Quartus 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 your DE-series 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 $LEDR_0$.

SOLUTION

```
module part3(CLK, D, Q, Q_n);
   input    D,CLK;
   output   Q, Q_n;
   wire     Q_n_master, Q_n_slave ;
   wire     Q_master, Q_slave;
   D_latch  master (.CLK(~CLK),  .D(D),          .Q(Q_master),
    .Q_n(Q_n_master));
   D_latch  slave  (.CLK( CLK),  .D(Q_master),   .Q(Q_slave) ,
    .Q_n(Q_n_slave));
   assign   Q      = Q_slave;
   assign   Q_n    = Q_n_slave;
endmodule
```

VERIFICATION

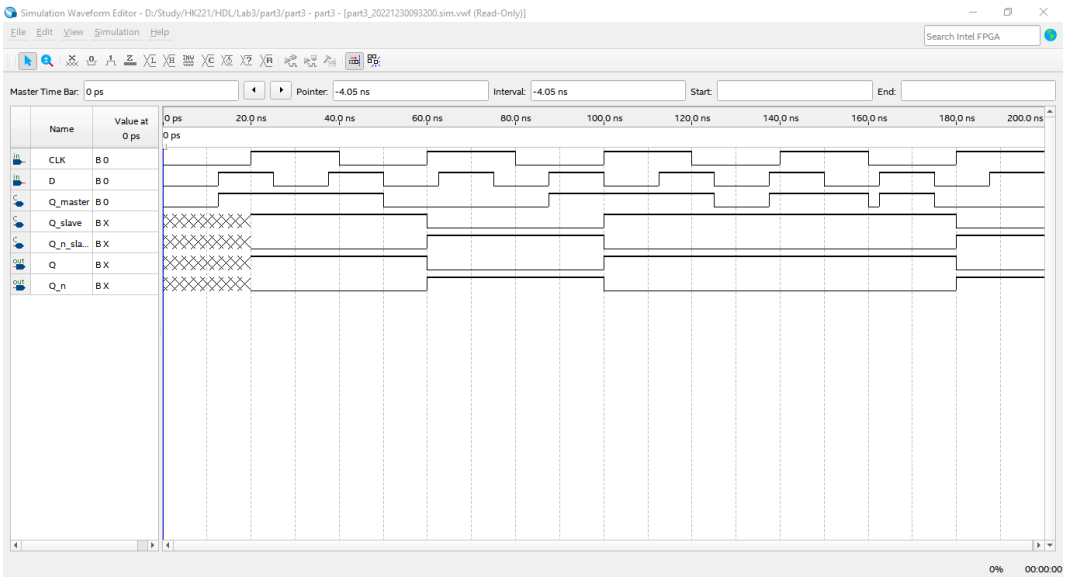


Figure 3.3: LAB 3: Simulation Result for part 3

3.5 Part IV

REQUIREMENT

1. Write a Verilog file that instantiates the three storage elements.
2. 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.

SOLUTION

```
module part4(D, CLK, Qa, Qb, Qc);
   input     [3:0]D;
   input     CLK;
   output    [3:0]Qa, Qb, Qc;
   D_latch inst1 (.CLK(CLK), .D(D), .Q(Qa));
   D_FF_P   inst2 (.CLK(CLK), .D(D), .Q(Qb));
   D_FF_N   inst3 (.CLK(CLK), .D(D), .Q(Qc));
endmodule

```

VERIFICATION

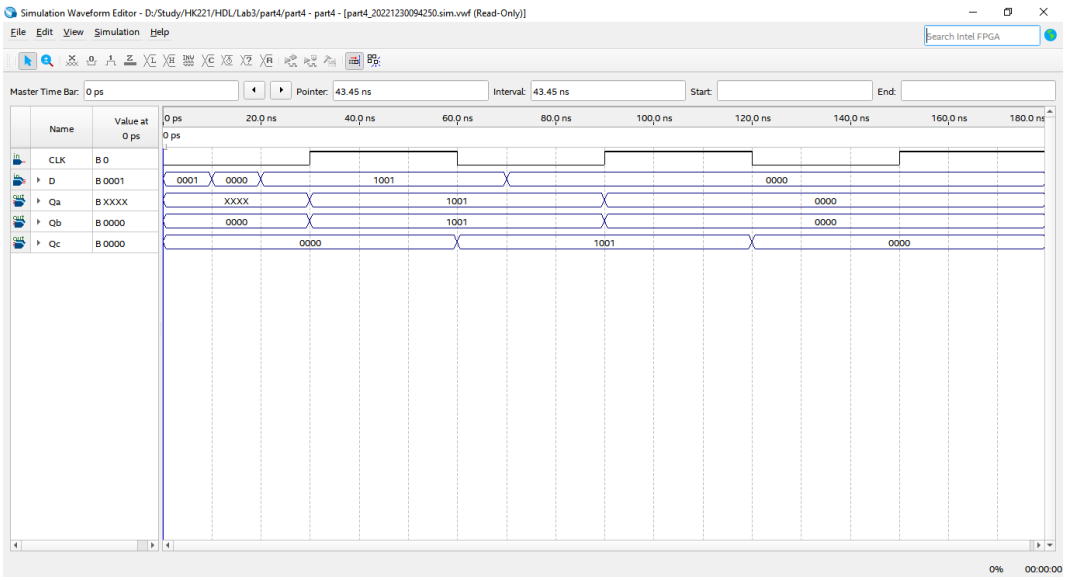


Figure 3.4: LAB 3: Simulation Result for part 4

3.6 Part V

REQUIREMENT

1. We wish to display the hexadecimal value of an 8-bit number A on the two 7-segment displays HEX_{3-2}. We also wish to display the hex value of an 8-bit number B on the two 7-segment displays HEX_{1-0}.

2. The values of A and B are inputs to the circuit which are provided by means of switches SW_{7-0}.Finally, use an adder to generate the arithmetic sum S = A + B, and display this sum on the 7-segment displays HEX_{5-4}. Show the carry-out produced by the adder on $LEDR[0]$.

SOLUTION

For this part, we use 2 different D_latch with 1 Enable pin to store the input into A and B

```
D_latch     inst1 (.CLK(CLK),  .D(Num), .Q(A));
D_latch     inst2 (.CLK(~CLK), .D(Num), .Q(B));
```

Then, we use block sum8bits which is the combination of 2 sum4bits blocks we mentioned before, to do the sum between A and B.

```
sum8bits   sum   (.A(A), .B(B), .C_IN(0), .SUM(Sum),
.C_OUT(C_out));
```

VERIFICATION

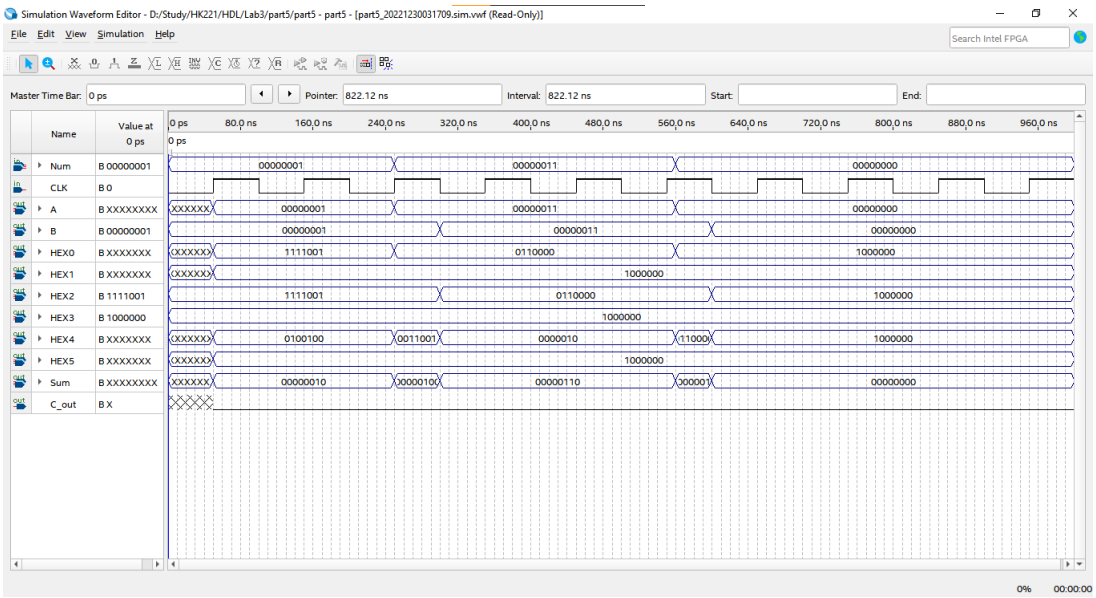


Figure 3.5: LAB 3: Simulation Result for part V

Chapter 4

Counters

4.1 Introduction

The purpose of this exercise is to build and use counters. The designed circuits are to be implemented on an Intel FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2-115 Board.

Students are expected to have a basic understanding of counters and sufficient familiarity with the Verilog hardware description language to implement various types of latches and flip-flops.

4.2 Part I

REQIREMENT

1. Write a Verilog file that defines an 8-bit counter. Your code should include a T flip-flop module that is instantiated eight times to create the counter. Compile the circuit. How many logic elements (LEs) are used to implement your circuit?
2. Simulate your circuit to verify its correctness.
3. Augment your Verilog file to use the pushbutton KEY_0 as the Clock input and switches SW_1 and SW_0 as *Enable* and *Clear* inputs, and 7-segment displays HEX_{1-0} to display the hexadecimal count as your circuit operates. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
4. Download your circuit into the FPGA chip and test its functionality by operating the switches.

SOLUTION

```
output   [6:0]HEX0, HEX1;
  reg      [7:0]T;
  reg      [7:0]pre;
  always   @(posedge CLK) begin
     T[0] <= EN;
     T[7:1] <= C[6:0] & T[6:0];
  end
  T_FF inst0 (.CLK(CLK), .CLR(~CLR), .T(T[0]), .Q(C[0]));
  T_FF inst1 (.CLK(CLK), .CLR(~CLR), .T(T[1]), .Q(C[1]));
  T_FF inst2 (.CLK(CLK), .CLR(~CLR), .T(T[2]), .Q(C[2]));
  T_FF inst3 (.CLK(CLK), .CLR(~CLR), .T(T[3]), .Q(C[3]));
  T_FF inst4 (.CLK(CLK), .CLR(~CLR), .T(T[4]), .Q(C[4]));
  T_FF inst5 (.CLK(CLK), .CLR(~CLR), .T(T[5]), .Q(C[5]));
  T_FF inst6 (.CLK(CLK), .CLR(~CLR), .T(T[6]), .Q(C[6]));
  T_FF inst7 (.CLK(CLK), .CLR(~CLR), .T(T[7]), .Q(C[7]));
  display7SEG(.in(C/16),.out(HEX1));
  display7SEG(.in(C%16),.out(HEX0));
endmodule
module T_FF(CLK, CLR, T, Q);
  input    T, CLK, CLR;
  output   reg Q;
  reg temp /*synthesis keep*/;
  always @(posedge CLK)  begin
     if (CLR) Q <= 0;
     else Q <= T ^ temp;
  end
  always @(*) temp = Q;
endmodule
```

4.3 Part II

REQUIREMENT

1. Another way to specify a counter is by using a register and adding 1 to its value. This can be accomplished using the following Verilog statement: Q<=Q+1;
2. Compile a 16-bit version of this counter and determine the number of LEs needed. Implement the counter on your DE-series board, using the displays HEX_{3-0} to show the counter value.

SOLUTION

```
module part2(CLK, CLR, EN, C, HEX0, HEX1, HEX2, HEX3);
   input    CLK, CLR, EN;
   output   reg [15:0]C;
   output   [6:0]HEX0, HEX1, HEX2, HEX3;
   always @(posedge CLK) begin
      if (!CLR) C <= 0;
      else if (EN) C <= C + 1;
   end
   // display hexadecima
   display7SEG(.in(C[15:12]),.out(HEX3));
   display7SEG(.in(C[11:8]),.out(HEX2));
   display7SEG(.in(C[7:4]),.out(HEX1));
   display7SEG(.in(C[3:0]),.out(HEX0));
endmodule

```

VERIFICATION

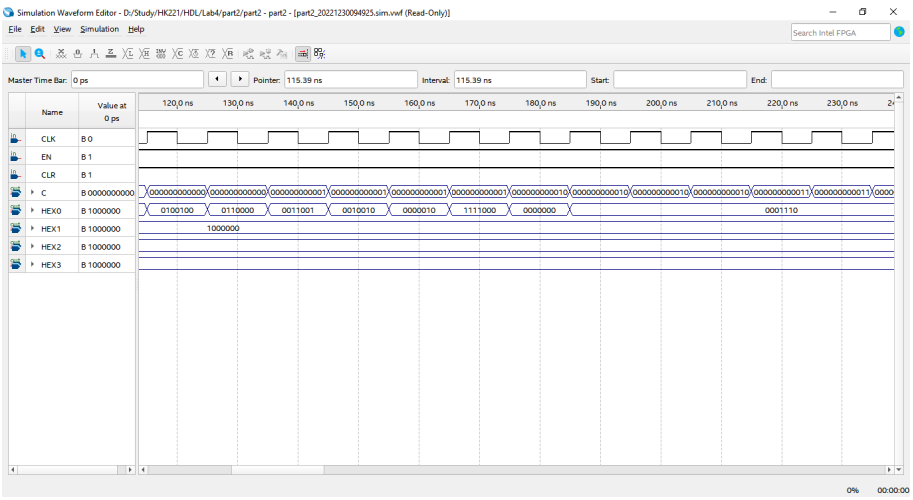


Figure 4.1: LAB 4: Simulation result for part 2

4.4 Part III

REQUIREMENT

1. Design and implement a circuit that successively flashes digits 0 through 9 on the 7-segment display HEX_0. Each digit should be displayed for about one second.
2. Use a counter to determine the one-second intervals. The counter should be incremented by the $50-MHzclock$ signal provided on the DE-series boards. Do not derive any other clock signals in your design–make sure that all flip-flops in your circuit are clocked directly by the $50-MHzclock$ signal.

SOLUTION

```
module part3(CLK, CLR, EN, HEX0);
   input    CLK, CLR, EN;
   output   [6:0]HEX0;
   reg      flag;
   reg      [3:0]C;
   reg      [25:0]Q;

    // 1s signal clock
   always @(posedge CLK) begin
      if (Q == 50000000) flag<=1;
      else flag <= 0;
      if (!CLR | Q>50000000)  Q<= 0;
      else if (EN) Q <= Q + 1;
   end

   always @(posedge flag) begin
      if (!CLR | C>9)  C<= 0;
      else if (EN) C <= C + 1;
   end

    // display hexadecimal
   display7SEG(.in(C[3:0]), .out(HEX0));
endmodule

```

VERIFICATION

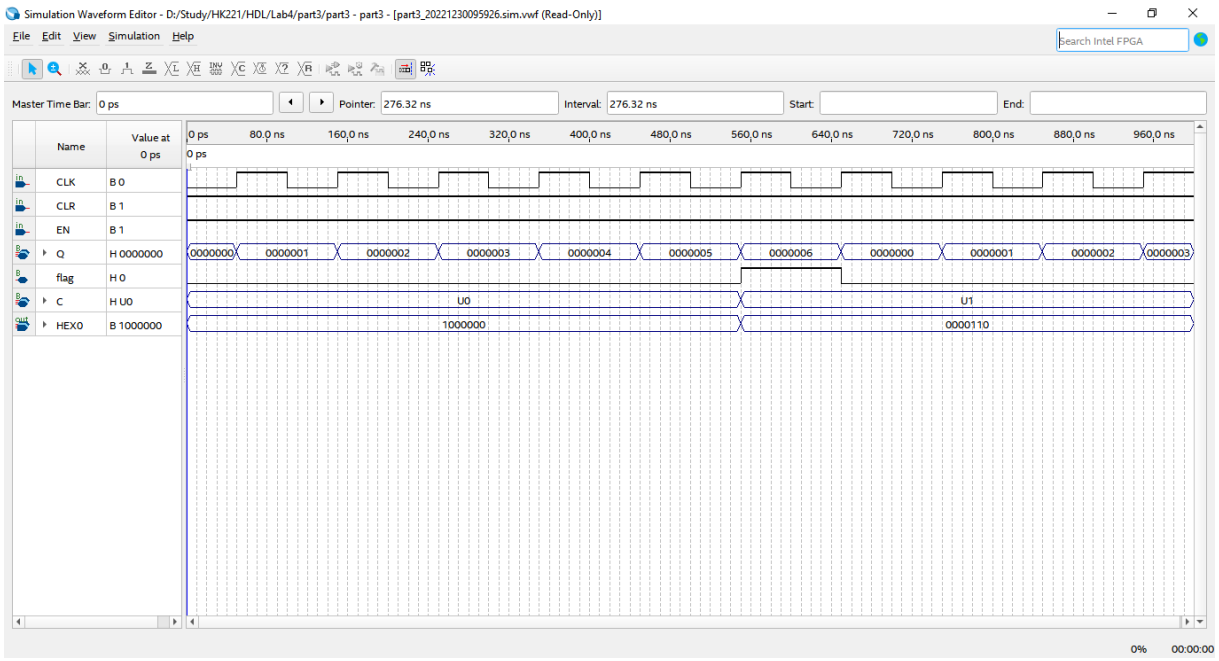


Figure 4.2: LAB 4: Simulation Result for part 4

4.5 Part IV

REQUIREMENT

1. Design and implement a circuit that displays a word on four 7-segment displays $HEX3-0$. The word to be displayed for your DE-series board
2. Make the letters rotate from right to left in intervals of about one second.

SOLUTION

```
            module part4(CLK, CLR, EN, HEX0, HEX1, HEX2, HEX3);
  input     CLK, CLR, EN;
  output    [6:0]HEX0, HEX1, HEX2, HEX3;

  reg       flag;
  reg       [2:0]C;
  reg       [25:0]Q;


  always @(posedge CLK) begin
     if (Q == 50000000) flag<=1;
     else flag <= 0;

     if (!CLR | Q>50000000)  Q<= 0;
     else if (EN) Q <= Q + 1;
  end

  always @(posedge flag) begin
     if (!CLR | C>4)  C<= 0;
     else if (EN) C <= C + 1;
  end

  assign {HEX3,HEX2,HEX1,HEX0} =
             (C==0)?{7'b0100001,7'b0000110,7'b0100100,7'b1111011}:
             (C==1)?{7'b1111011,7'b0100001,7'b0000110,7'b0100100}:
             (C==2)?{7'b0100100,7'b1111011,7'b0100001,7'b0000110}:
                    {7'b0000110,7'b0100100,7'b1111011,7'b0100001};


endmodule

```

VERIFICATION

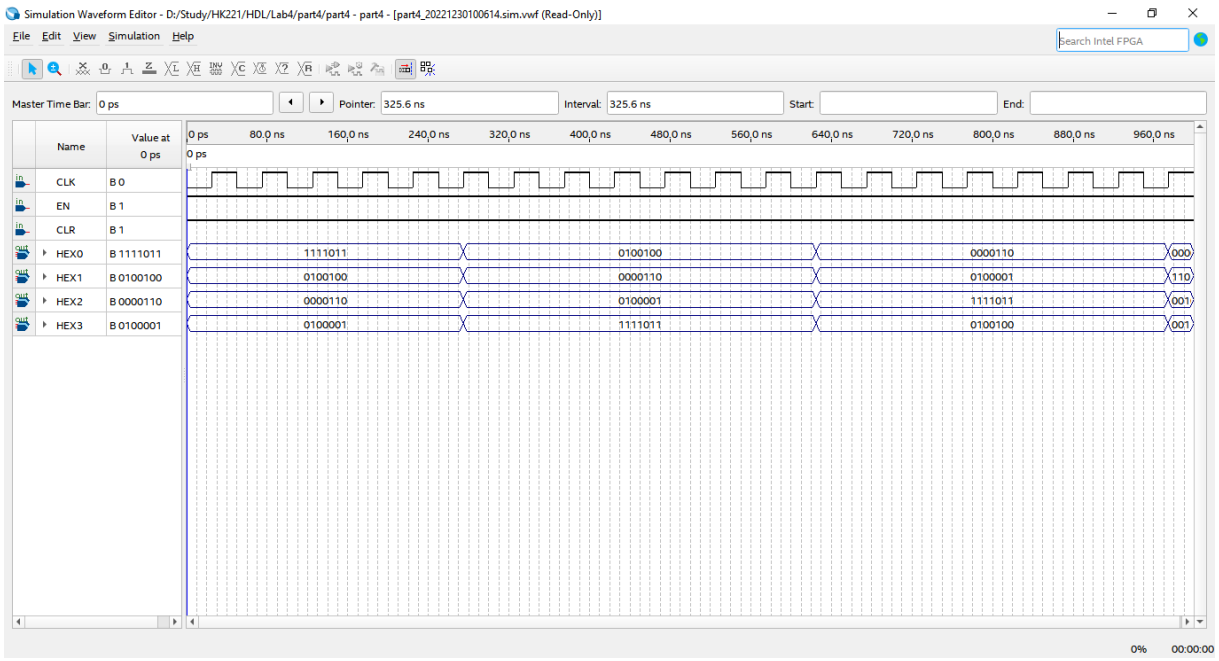


Figure 4.3: LAB 4: Simulation Result for part IV

4.6 Part V

REQUIREMENT

1. Augment your circuit from Part IV so that it can rotate the word over all of the 7-segment displays on your DE-series board.
2. The shifting pattern for the DE10-Lite is shown in Table below. Your can base on this table to create your own table for DE2i-150.

Count	Character pattern					
000			d	E	1	0
001		d	E	1	0	
010	d	E	1	0		
011	E	1	0			d
100	1	0			d	E
101	0			d	E	1

Figure 4.4: LAB 4: Hint for Part IV (Rotating the word on six display)

SOLUTION

To use the clock of the board which has the frequency equal to 50MHz, we have to create the counter with in the input is the board's clock and the output which just equal to 1 when

```
always @(posedge CLK) begin
    if (Q == 50000000) flag<=1;
    else flag <= 50000000;
    if (!CLR | Q>50000000)  Q<= 0;
    else if (EN) Q <= Q + 1;
end
always @(posedge flag) begin
    if (!CLR | C>7)  C<= 0;
    else if (EN) C <= C + 1;
end

```

Each time the output of the counter equal to 1, we update the value of 8 7-segment leds by follow module.

```
module update_display(out, val);
   input [2:0]val;
   output[55:0]out;
```

```
    wire [55:0]temp =
    56'b11111111111111111111111111111010000100001100100100111101 1;
    wire [55:0]templong = temp<<(7*val);
    wire [55:0]tempshort = temp>>(56-7*val);
    assign out = templong | tempshort;
endmodule

```

Ansd this is the entire design

```
always @(posedge CLK) begin
    if (Q == 50000000) flag<=1;
    else flag <= 50000000;

    if (!CLR | Q>50000000)  Q<= 0;
    else if (EN) Q <= Q + 1;
end
always @(posedge flag) begin
    if (!CLR | C>7)  C<= 0;
    else if (EN) C <= C + 1;
end
always @(HEX_BUS) HEX = HEX_BUS;
update_display inst1(.out(HEX_BUS), .val(C[2:0]));
assign {HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0} = HEX;

```

VERIFICATION

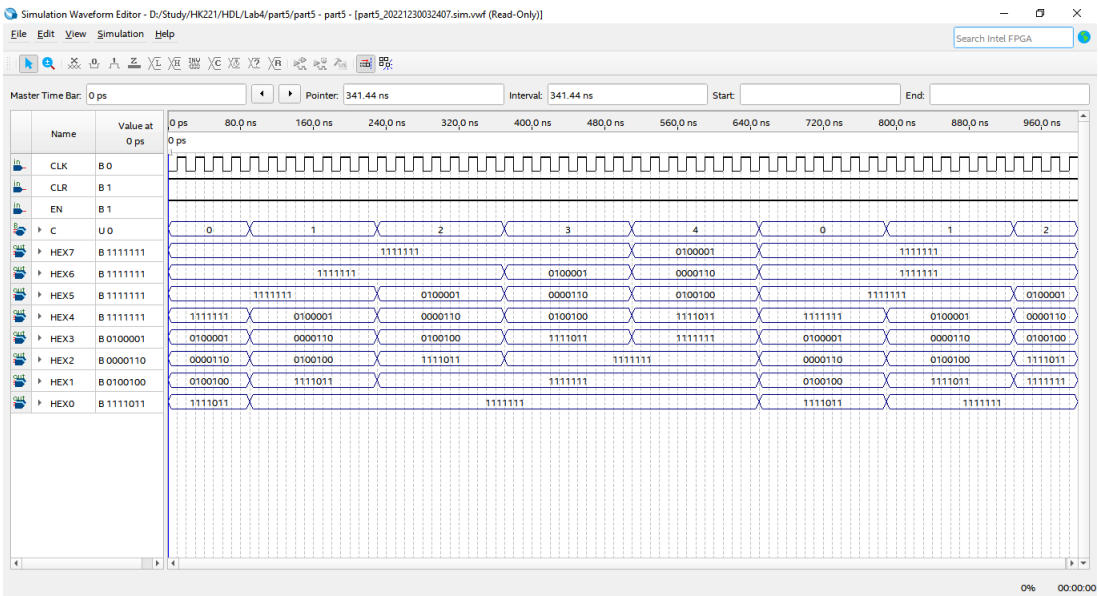


Figure 4.5: LAB 4: Simulation result for part V

Chapter 5

Timers and Real-time Clock

5.1 Introduction

The purpose of this exercise is to study the use of clocks in timed circuits. The designed circuits are to be implemented on an Intel FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2-115 board.

5.2 Part I

REQUIREMENT

1. Create a modulo-k counter by modifying the design of an 8-bit counter to contain an additional parameter. The counter should count from 0 to $k-1$. When the counter reaches the value $k1$, then the next counter value should be 0. Include an output from the counter called $rollover$ and set this output to 1 in the clock cycle where the count value is equal to $k-1$.
2. Write a Verilog file that specifies the circuit for $k = 20$, and an appropriate value of n. Your circuit should use pushbutton KEY_0 as an asynchronous reset and KEY_1 as a manual clock input. The contents of the counter should be displayed on the red lights $LEDR$. Also, display the rollover signal on one of the LEDR lights

SOLUTION

```
module part1 (CLK, RESET, ROLLOVER, COUNTER);
   input    CLK, RESET;
   output   ROLLOVER;
   output   [5:0]COUNTER;

   counter k_bits (.CLK(CLK), .RESET(RESET), .K(20) ,
    .rollover(ROLLOVER), .Q(COUNTER));
   defparam k_bits.n = 5;

endmodule
module counter (CLK, RESET, K, rollover, Q);
   parameter n=4;
   input    [n-1:0]K;
   input    CLK, RESET;
   output   reg [n-1:0]Q;
   output   reg rollover;
   always @(posedge CLK) begin
      if (Q==(K-2)) rollover=1;
      else rollover = 0;
   end
   always @(posedge CLK or negedge RESET) begin
      if (!RESET) Q<=0;
      else
         if (rollover) Q<=0;
         else Q<=Q+1;
   end
endmodule
```

5.3 Part II

REQUIREMENT

1. Using your modulo-counter from Part I as a subcircuit, implement a 3-digit BCD counter (hint: use multiple counters, not just one).
2. Display the contents of the counter on the 7-segment displays, HEX_{2-0}. Connect all of the counters in your circuit to the $50-MHzclock$ signal on your DE-series board, and make the BCD counter increment at one-second intervals. Use the pushbutton switch KEY_0 to reset the BCD counter to 0.

SOLUTION

```
module part2 (CLK, RESET, HEX0, HEX1, HEX2);
   input    CLK, RESET;
   output   [6:0]HEX0, HEX1, HEX2;

   wire     [11:0]Q /*synthesis keep*/;
   wire     clk1, clk2, clk3 /*synthesis keep*/;
   reg      clk1_bus, clk2_bus, clk3_bus /*synthesis keep*/;

   counter bcd_0 (.CLK(CLK),  .RESET(RESET), .K(50000000) ,
    .rollover(clk1), . Q());
   defparam bcd_0.n = 26;

   counter bcd_1 (.CLK(clk1), .RESET(RESET), .K(10) ,
    .rollover(clk2), . Q(Q[3:0]));
   defparam bcd_1.n = 5;
   counter bcd_2 (.CLK(clk2), .RESET(RESET), .K(10) ,
    .rollover(clk3), . Q(Q[7:4]));
   defparam bcd_2.n = 5;
   counter bcd_3 (.CLK(clk3), .RESET(RESET), .K(10) , .rollover(),
       . Q(Q[11:8]));
   defparam bcd_3.n = 5;

   display7SEG(.in(Q[3:0]), .out(HEX0));
   display7SEG(.in(Q[7:4]), .out(HEX1));
   display7SEG(.in(Q[11:8]), .out(HEX2));
endmodule

```

VERIFICATION

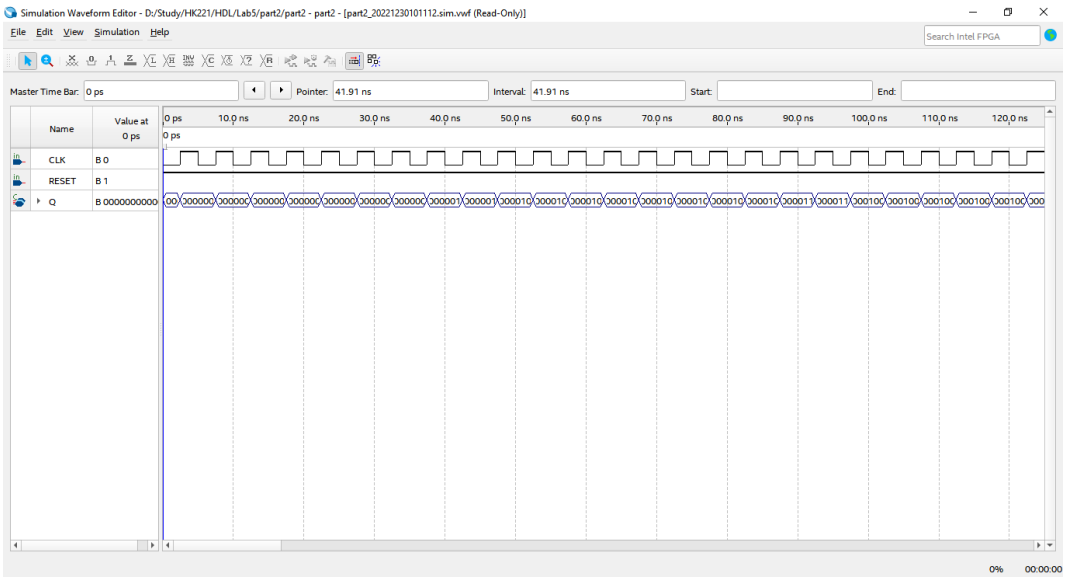


Figure 5.1: LAB 5: Simulation Result for part II

5.4 Part III

REQUIREMENT

1. Design and implement a circuit on your DE-series board that acts as a real-time clock. It should display the minutes (from 0 to 59) on HEX_{5-4}, the seconds (from 0 to 59) on HEX_{3-2}, and hundredths of a second (from 0 to 99) on HEX_{1-0}.
2. Use the switches SW_{7-0} to preset the minute part of the time displayed by the clock when KEY_1 is pressed. Stop the clock whenever KEY_0 is being pressed and continue the clock when KEY_0 is released.

SOLUTION

```
module part3 (CLK, RESET, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);
  input    CLK, RESET;
  output   [6:0]HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;

  wire     [19:0]Q /*synthesis keep*/;
  wire     clk0, clk1, clk2, clk3 /*synthesis keep*/;
  reg      clk1_bus, clk2_bus, clk3_bus /*synthesis keep*/;

  counter sub_clk (.CLK(CLK), .RESET(RESET), .K(500000),
  .rollover(clk0), .Q());
  defparam sub_clk.n = 19;

  counter bcd_1 (.CLK(clk0), .RESET(RESET), .K(100) ,
  .rollover(clk1), . Q(Q[6:0]));
  defparam bcd_1.n = 7;
  counter bcd_2 (.CLK(clk1), .RESET(RESET), .K(60) ,
  .rollover(clk2), . Q(Q[12:7]));
  defparam bcd_2.n = 6;
  counter bcd_3 (.CLK(clk2), .RESET(RESET), .K(60) ,
  .rollover(clk3), . Q(Q[18:13]));
  defparam bcd_3.n = 6;

  display7SEG inst0 (.in(Q[6:0]%10),     .out(HEX0));
  display7SEG inst1 (.in(Q[6:0]/10),     .out(HEX1));
  display7SEG inst2 (.in(Q[12:7]%10),    .out(HEX2));
  display7SEG inst3 (.in(Q[12:7]/10),    .out(HEX3));
  display7SEG inst4 (.in(Q[18:13]%10),   .out(HEX4));
  display7SEG inst5 (.in(Q[18:13]/10),   .out(HEX5));

endmodule

```

5.5 Part IV

An early method of telegraph communication was based on the Morse code. This code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse).

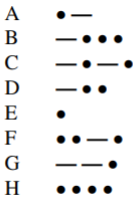


Figure 5.2: LAB 5: Hint for part IV

REQUIREMENT

1. Design and implement a circuit that takes as input one of the first eight letters of the alphabet and displays the Morse code for it on a red LED.
2. Your circuit should use switches SW_{2-0} and pushbuttons KEY_{1-0} as inputs. When a user presses KEY_1, the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes.
3. Pushbutton KEY_0 should function as an asynchronous reset. A high-level schematic diagram of the circuit is shown in Figure 2.

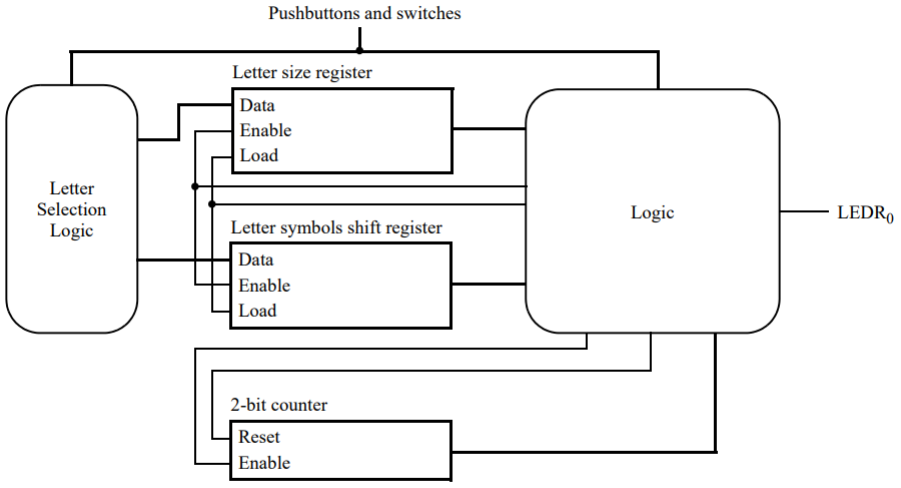


Figure 2: High-level schematic diagram of the circuit for part IV.

Figure 5.3: LAB 5: Hint for part IV (Diagram)

SOLUTION

We base on the hint given; we translate the input which has a 3-bit length into the signal 13 bits.

```
module translate_signal(in, out);
   input    [3:0]in;
   output   reg [12:0]out;

   always begin
      if (in==0) out=13'b0000000111010;
      if (in==1) out=13'b0001010101110;
      if (in==2) out=13'b0101110101110;
      if (in==3) out=13'b0000010101110;
      if (in==5) out=13'b0001011101010;
      if (in==4) out=13'b0000000000010;
      if (in==6) out=13'b0001110111010;
      if (in==7) out=13'b0000010101010;
   end
endmodule

```

After translating, we shift left 13 times, each time, if that bit is 1 we turn on the led, if that bit is equal to 0, we turn off the led. The period for each clock is 0.5 seconds.

```
module part4 (CLK, RESET, EN, code, led);
   input    CLK, RESET, EN;
   input    [2:0]code;
   output   led;

   reg      [2:0]numb;
   wire     [12:0]decode    /*synthesis keep*/;
   wire     time_start      /*synthesis keep*/;



   counter     sub_clk  (.CLK(CLK), .RESET(RESET), .K(25000000),
   .rollover(time_start), .Q());
   translate_signal  inst0    (.in(code), .out(decode));
   defparam sub_clk.n = 25;

   always @(posedge time_start) numb<=numb+1;
   assign led = decode[numb] & EN;


endmodule

```

Chapter 6

Adders, Subtractors, and Multipliers

6.1 Introduction

The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each circuit will be described in Verilog and implemented on an Intel FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2- 115 board.

6.2 Part I, II

REQUIREMENT

1. Generate the required Verilog file. Use switches SW_{7-4} 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 HEX_2 and HEX_0, respectively. The result P = A × B is to be displayed on HEX_{5-4}.
2. Use simulation to verify your design.

SOLUTION

```
module part1(CLK, RESET, IN, IN_LSB, IN_MSB, OUT_LSB, OUT_MSB, SUM,
    CARRY, OVERFLOW);
   input    CLK, RESET;
   input    [7:0]IN;
   output   [7:0]SUM;
   output   [6:0]IN_LSB, IN_MSB, OUT_LSB, OUT_MSB;
   output   OVERFLOW, CARRY;

   reg   [7:0]A,S;
   wire  [7:0]sum_ff_out, input_ff_out, sum_ff_in;

   always begin
      A = IN;
      S = sum_ff_out;
   end

   assign SUM = sum_ff_in;

   D_FF input_ff     (.CLK(CLK), .RESET(RESET), .D(A),
    .Q(input_ff_out));
   D_FF overflow_ff  (.CLK(CLK), .RESET(RESET), .D(overflow_ff_in),
    .Q(OVERFLOW));
   D_FF carry_ff     (.CLK(CLK), .RESET(RESET), .D(carry_ff_in),
    .Q(CARRY));
   D_FF sum_ff       (.CLK(CLK), .RESET(RESET), .D(sum_ff_in),
    .Q(sum_ff_out));

   defparam input_ff.n = 8;
   defparam sum_ff.n = 8;
   defparam overflow_ff.n = 8;

   sum4bits       inst0 (.X(input_ff_out), .Y(sum_ff_out),
    .SUM(sum_ff_in), .CARRY(carry_ff_in));
   overflow_bits  inst1 (.X(input_ff_out), .Y(sum_ff_in),
    .SUM(sum_ff_out), .OVERFLOW(overflow_ff_in));

```

```
30    display7SEG    A_lsb (.in(A[3:0]),.out(IN_LSB));
31    display7SEG    A_msb (.in(A[7:4]),.out(IN_MSB));
32    display7SEG    S_lsb (.in(S[3:0]),.out(OUT_LSB));
33    display7SEG    S_msb (.in(S[7:4]),.out(OUT_MSB));
34
35 endmodule
36 module sum4bits(X,Y,SUM,CARRY);
37    input     [7:0]X,Y;
38    output    [7:0]SUM;
39    output    CARRY;
40
41    assign {CARRY,SUM} = X+Y;
42 endmodule
43
44 module overflow_bits(X,Y,SUM,OVERFLOW);
45    input     [7:0]X,Y,SUM;
46    output    OVERFLOW;
47
48    assign OVERFLOW = (X>=128 | Y>=128 | SUM>=128);
49
50 endmodule
51
```

6.3 Part III, IV

REQUIREMENT

1. In Part III, an array multiplier was implemented using full adder modules. At a higher level, a row of full adders functions as an n-bit adder and the array multiplier circuit can be represented as shown in Figure 5.

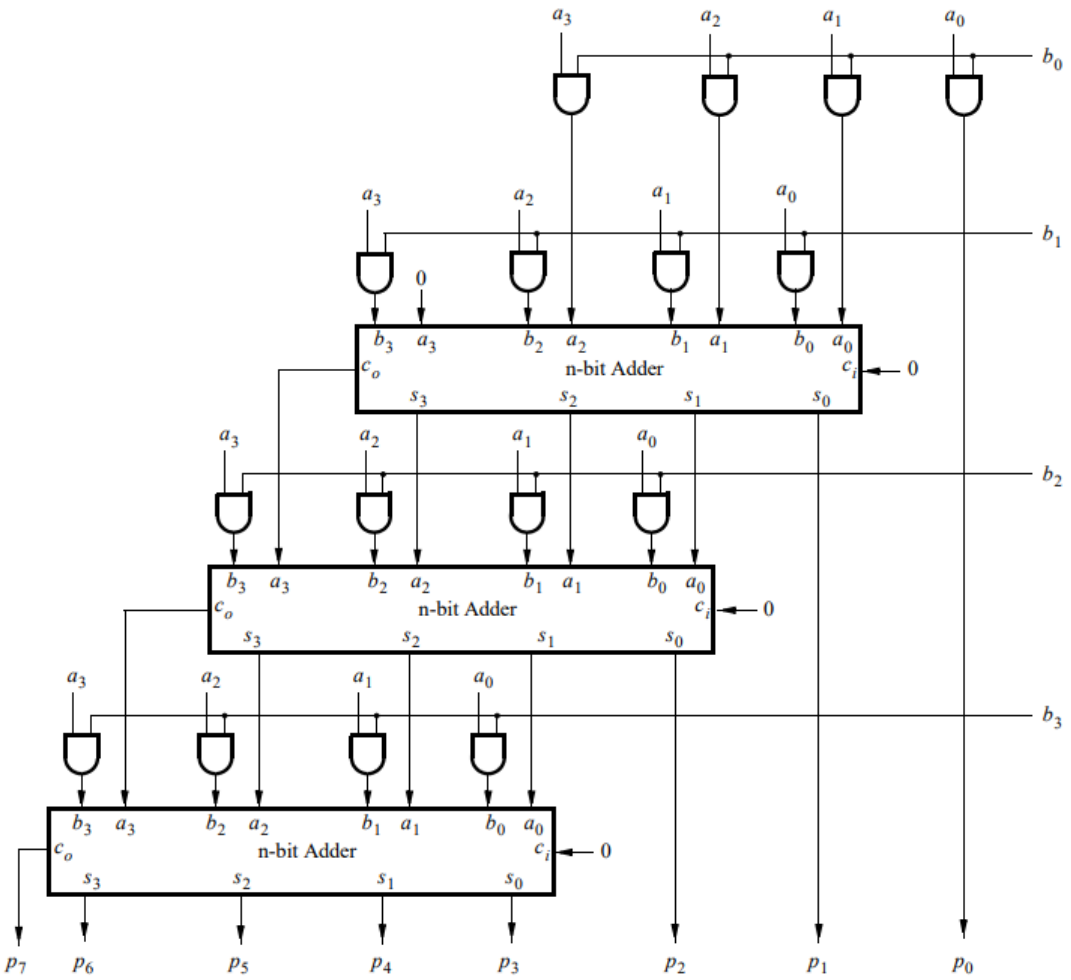


Figure 5: An array multiplier implemented using n-bit adders.

Figure 6.1: LAB 6: Hint for part IV (4 bits implementation)

2. Each n-bit adder adds a shifted version of A for a given row and the partial product of the row above. Abstracting the multiplier circuit as a sequence of additions allows us to build larger multipliers. The multiplier should consist of n-bit adders arranged in a structure shown in Figure 5. Use this approach to implement an 8 x 8 multiplier circuit with registered inputs and outputs, as shown in Figure 6.

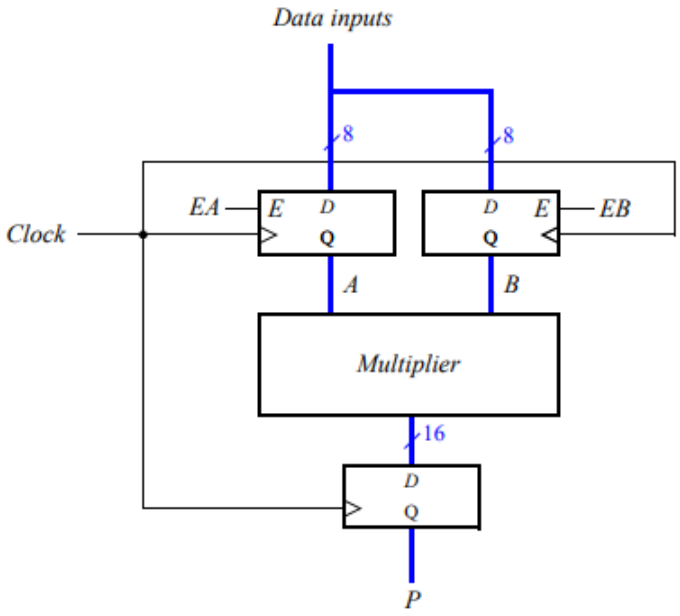


Figure 6: A registered multiplier circuit.

Figure 6.2: LAB 6: Hint for part IV (Diagram)

SOLUTION

```
module part4(CLK, IN,EA,EB,P);
  input    CLK, EA, EB;
  input    [7:0]IN;
  output   [15:0]P;

  wire  [7:0] A,B;
  wire  [15:0]SUM;
  wire  [7:0] adder1_in0, adder1_in1;
  wire  [7:0] adder2_in0, adder2_in1;
  wire  [7:0] adder3_in0, adder3_in1;
  wire  [7:0] adder4_in0, adder4_in1;
  wire  [7:0] adder5_in0, adder5_in1;
  wire  [7:0] adder6_in0, adder6_in1;
  wire  [7:0] adder7_in0, adder7_in1;

  D_FF  inputa_ff (.CLK(CLK), .EN(EA), .D(IN),  .Q(A));
  D_FF  inputb_ff (.CLK(CLK), .EN(EB), .D(IN),  .Q(B));
```

```
18    D_FF  output_ff (.CLK(CLK), .EN(1),  .D(SUM), .Q(P));
19
20    multiply_Nx1   inst0    (.A(A), .B(B[0]),
      .P({adder1_in0[6:0],SUM[0]}));
21    multiply_Nx1   inst1    (.A(A), .B(B[1]), .P(adder1_in1));
22    multiply_Nx1   inst2    (.A(A), .B(B[2]), .P(adder2_in1));
23    multiply_Nx1   inst3    (.A(A), .B(B[3]), .P(adder3_in1));
24    multiply_Nx1   inst4    (.A(A), .B(B[4]), .P(adder4_in1));
25    multiply_Nx1   inst5    (.A(A), .B(B[5]), .P(adder5_in1));
26    multiply_Nx1   inst6    (.A(A), .B(B[6]), .P(adder6_in1));
27    multiply_Nx1   inst7    (.A(A), .B(B[7]), .P(adder7_in1));
28
29    sumNbits adder1 (.A(adder1_in0), .B(adder1_in1), .C_IN(0)
      ,.SUM({adder2_in0[2:0],SUM[1]}) ,.C_OUT(adder2_in0[3]));
30    sumNbits adder2 (.A(adder2_in0), .B(adder2_in1), .C_IN(0)
      ,.SUM({adder3_in0[2:0],SUM[2]}) ,.C_OUT(adder3_in0[3]));
31    sumNbits adder3 (.A(adder3_in0), .B(adder3_in1), .C_IN(0)
      ,.SUM({adder4_in0[2:0],SUM[3]}) ,.C_OUT(adder4_in0[3]));
32    sumNbits adder4 (.A(adder4_in0), .B(adder4_in1), .C_IN(0)
      ,.SUM({adder5_in0[2:0],SUM[4]}) ,.C_OUT(adder5_in0[3]));
33    sumNbits adder5 (.A(adder5_in0), .B(adder5_in1), .C_IN(0)
      ,.SUM({adder6_in0[2:0],SUM[5]}) ,.C_OUT(adder6_in0[3]));
34    sumNbits adder6 (.A(adder6_in0), .B(adder6_in1), .C_IN(0)
      ,.SUM({adder7_in0[2:0],SUM[6]}) ,.C_OUT(adder7_in0[3]));
35    sumNbits adder7 (.A(adder7_in0), .B(adder7_in1), .C_IN(0)
      ,.SUM(SUM[14:7])          ,.C_OUT(SUM[15]));
36
37    defparam inputa_ff.n_bits = 8;
38    defparam inputb_ff.n_bits = 8;
39    defparam output_ff.n_bits = 16;
40
41    defparam inst0.n_bits = 8;
42    defparam inst1.n_bits = 8;
43    defparam inst2.n_bits = 8;
44    defparam inst3.n_bits = 8;
45    defparam inst4.n_bits = 8;
46    defparam inst5.n_bits = 8;
47    defparam inst6.n_bits = 8;
48    defparam inst7.n_bits = 8;
49
50    defparam adder1.n_bits = 8;
51    defparam adder2.n_bits = 8;
52    defparam adder3.n_bits = 8;
53    defparam adder4.n_bits = 8;
54    defparam adder5.n_bits = 8;
55    defparam adder6.n_bits = 8;
56    defparam adder7.n_bits = 8;
57endmodule
```

VERIFICATION

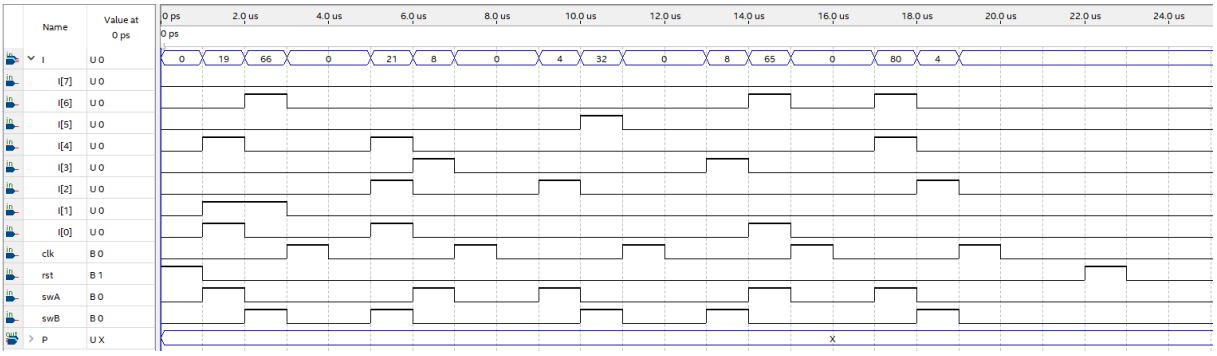


Figure 6.3: LAB 6: Simulation Result for part IV

6.4 Part V

REQUIREMENT

1. Part IV showed how to implement multiplication A × B as a sequence of additions, by accumulating the shiftedversions of A one row at a time. Another way to implement this circuit is to perform addition using an adder tree. An adder tree is a method of adding several numbers together in a parallel fashion.

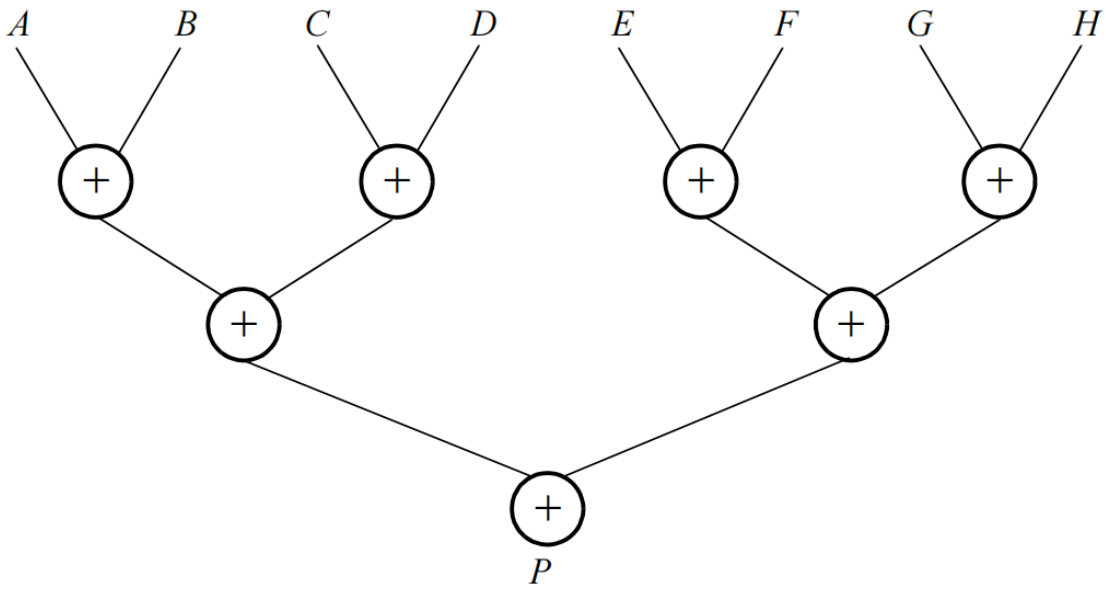


Figure 6.4: LAB 6: Hint for part V (Adder tree)

2. In this part you are to implement an 8 x 8 multiplier circuit by using the adder-tree approach. Inputs A and B, as well as the output P should be registered as in Part IV

SOLUTION

In this part we try to implement multiplication by adder tree or we will add in parallel rather than sequence additions

So firstly I create **add_8bit** module with use to calculate the sum of two numbers 8 bit.

Then create **MUL** module but now use parallel adding or (adder tree) so with A, B 8 bits number I will separate it into 8 layers and extern each layer to 8 bit.

After having separated layers represent A,B,C ... in picture then I do add operations like the image above.

```
module add_8_bit(Cout,S,A,B,Cin);
    input [7:0] A,B;
    input Cin;
    output [7:0]S;
    output Cout;
```

Figure 6.5: LAB 6: Module add_8_bits

```
assign LAYER1= A & {8{B[0]}};
assign LAYER2= A & {8{B[1]}};
assign LAYER3= A & {8{B[2]}};
assign LAYER4= A & {8{B[3]}};
assign LAYER5= A & {8{B[4]}};
assign LAYER6= A & {8{B[5]}};
assign LAYER7= A & {8{B[6]}};
assign LAYER8= A & {8{B[7]}};
```

Figure 6.6: LAB 6: Module MUL

```
assign P[0]=LAYER1[0];

add_8_bit ins2(C2,SUM2,LAYER2,{1'b0,LAYER1[7:1]},0);
assign P[1]=SUM2[0];
add_8_bit ins3(C3,SUM3,LAYER3,{C2,SUM2[7:1]},0);
assign P[2]=SUM3[0];
add_8_bit ins4(C4,SUM4,LAYER4,{C3,SUM3[7:1]},0);
assign P[3]=SUM4[0];
add_8_bit ins5(C5,SUM5,LAYER5,{C4,SUM4[7:1]},0);
assign P[4]=SUM5[0];
add_8_bit ins6(C6,SUM6,LAYER6,{C5,SUM5[7:1]},0);
assign P[5]=SUM6[0];
add_8_bit ins7(C7,SUM7,LAYER7,{C6,SUM6[7:1]},0);
assign P[6]=SUM7[0];
add_8_bit ins8(P[15],P[14:7],LAYER8,{C7,SUM7[7:1]},0);
```

Figure 6.7: LAB 6: Entire design for part V

Chapter 7

Finite State Machines

7.1 Introduction

This is an exercise in using finite state machines.

7.2 Part I

REQUIREMENT

1. 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.
2. Use the toggle switch SW_0 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 red light $LEDR_9$ as the output z, and assign the state flip-flop outputs to the red lights $LEDR_8$ to $LEDR_0$

SOLUTION

```
module part1(CLK, RESET, IN, OUT, STATE);
  input    CLK, RESET, IN;
  output   [8:0] STATE;
  output   OUT;

  wire  ff0_in, ff0_out;
  wire  ff1_in, ff1_out;
  wire  ff2_in, ff2_out;
  wire  ff3_in, ff3_out;
  wire  ff4_in, ff4_out;
  wire  ff5_in, ff5_out;
  wire  ff6_in, ff6_out;
  wire  ff7_in, ff7_out;
  wire  ff8_in, ff8_out;

  D_FF state0 (.CLK(CLK), .RESET(RESET), .D(ff0_in), .Q(ff0_out));
  D_FF state1 (.CLK(CLK), .RESET(RESET), .D(ff1_in), .Q(ff1_out));
  D_FF state2 (.CLK(CLK), .RESET(RESET), .D(ff2_in), .Q(ff2_out));
  D_FF state3 (.CLK(CLK), .RESET(RESET), .D(ff3_in), .Q(ff3_out));
  D_FF state4 (.CLK(CLK), .RESET(RESET), .D(ff4_in), .Q(ff4_out));
  D_FF state5 (.CLK(CLK), .RESET(RESET), .D(ff5_in), .Q(ff5_out));
  D_FF state6 (.CLK(CLK), .RESET(RESET), .D(ff6_in), .Q(ff6_out));
  D_FF state7 (.CLK(CLK), .RESET(RESET), .D(ff7_in), .Q(ff7_out));
  D_FF state8 (.CLK(CLK), .RESET(RESET), .D(ff8_in), .Q(ff8_out));

  assign ff0_in = ~(ff1_in | ff2_in | ff3_in | ff4_in | ff5_in |
  ff6_in | ff7_in | ff8_in);

  assign ff1_in =  ff0_out & ~IN |
                   ff2_out & ~IN |
                   ff4_out & ~IN |
                   ff6_out & ~IN |
```

```
32                  ff8_out & ~IN ;
33
34   assign ff2_in =  ff0_out &  IN |
35                    ff1_out &  IN |
36                    ff3_out &  IN |
37                    ff5_out &  IN |
38                    ff7_out &  IN ;
39
40   assign ff3_in =  ff1_out & ~IN ;
41   assign ff5_in =  ff3_out & ~IN ;
42   assign ff7_in =  ff5_out & ~IN |
43                    ff7_out & ~IN ;
44
45   assign ff4_in =  ff2_out &  IN ;
46   assign ff6_in =  ff4_out &  IN ;
47   assign ff8_in =  ff6_out &  IN |
48                    ff8_out &  IN ;
49
50
51   assign OUT = ff7_out | ff8_out;
52   assign STATE = {ff8_out, ff7_out, ff6_out, ff5_out, ff4_out,
     ff3_out, ff2_out, ff1_out, ff0_out};
53
54 endmodule
55
```

7.3 Part II

REQUIREMENT

1. 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.
2. A state diagram for this FSM is shown in Figure 2.
3. To implement the FSM use nine state flip-flops called y8, . . . , y0 and the one-hot state assignment given in Table 1.

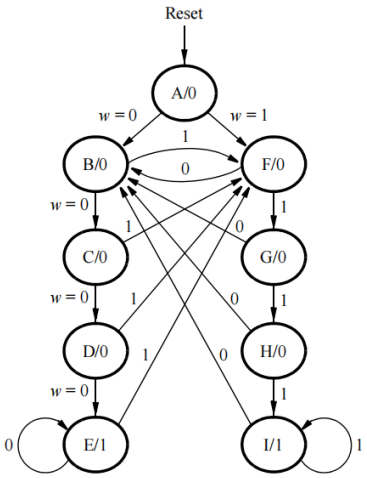


Figure 2: A state diagram for the FSM.

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.

LAB 7: Hint for part II

SOLUTION

```
module part2(CLK, RESET, IN, OUT, STATE);
   input    CLK, RESET, IN;
   output   OUT;
   output   [8:0]STATE;

   reg      [3:0]YQ, YD;
   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 @(IN, YQ) begin
      case (YQ)
         A: begin if (IN) YD = F; else YD = B; end
         B: begin if (IN) YD = F; else YD = C; end
         C: begin if (IN) YD = F; else YD = D; end
         D: begin if (IN) YD = F; else YD = E; end
         E: begin if (IN) YD = F; else YD = E; end
         F: begin if (IN) YD = G; else YD = B; end
         G: begin if (IN) YD = H; else YD = B; end
         H: begin if (IN) YD = I; else YD = B; end
         I: begin if (IN) YD = I; else YD = B; end
         default: YD = 4'bxxxx;
      endcase
   end

   always @(posedge CLK) begin
      if (RESET) YQ<=YD;
      else YQ<=A;
   end

   change_signal inst0 (.in(YQ), .out(STATE));
   assign OUT = (YQ==E) | (YQ==I);

endmodule

```

VERIFICATION

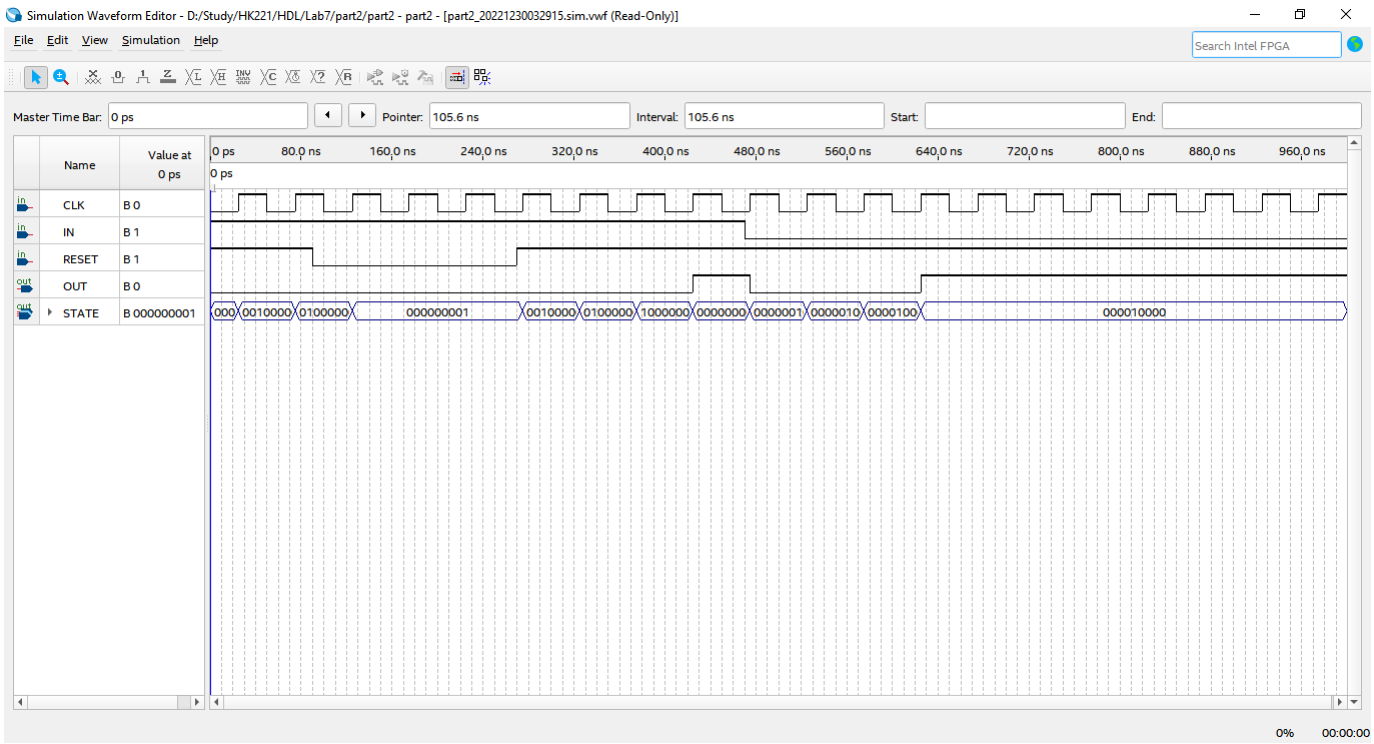


Figure 7.1: LAB 7: Simulation Result for part 2

7.4 Part III

REQUIREMENT

1. The sequence detector can be implemented in a straightforward manner 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.
2. 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.

SOLUTION

```
module part3(CLK, RESET, IN, OUT);
    input     CLK, RESET, IN;
    output    OUT;
    wire      out0, out1;

    assign OUT = out0 | out1;

    detector detect0 (CLK, RESET, ~IN, out0);
    detector detect1 (CLK, RESET,  IN, out1);

endmodule

module detector(CLK, RESET, IN, OUT);
    input     CLK, RESET, IN;
    output    OUT;

    wire  ff0_out, ff1_out, ff2_out, ff3_out;
    wire  ff0_in, ff1_in, ff2_in, ff3_in;

    D_FF state0 (.CLK(CLK), .RESET(RESET), .D(IN), .Q(ff1_in));
    D_FF state1 (.CLK(CLK), .RESET(RESET), .D(ff1_in), .Q(ff2_in));
    D_FF state2 (.CLK(CLK), .RESET(RESET), .D(ff2_in), .Q(ff3_in));
    D_FF state3 (.CLK(CLK), .RESET(RESET), .D(ff3_in), .Q(ff3_out));

    assign OUT = ff1_in & ff2_in & ff3_in & ff3_out;
endmodule

```

7.5 Part IV

REQUIREMENT

1. In this part of the exercise you are to implement a Morse-code encoder using an FSM. The Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

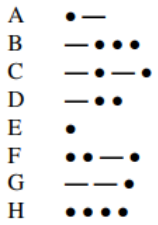


Figure 7.2: LAB 7: Hint for part IV

2. Design and implement a Morse-code encoder circuit using an FSM. Your circuit should take as input one of the first eight letters of the alphabet and display the Morse code for it on a red LED. Use switches SW_{2-0} and push-buttons KEY_{1-0} as inputs. When a user presses KEY_1, the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton KEY_0 should function as an asynchronous reset.

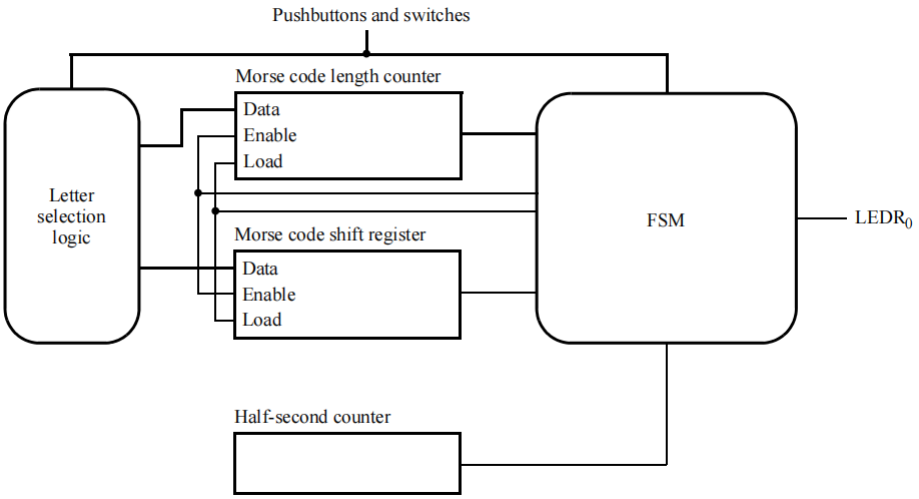


Figure 7.3: LAB 7: Hint for part IV (Diagram)

SOLUTION

Our idea is the same with the LAB5_PART_IV but now I try to use a state machine so firstly I create state.

```
parameter   A=3'b000,
        B=3'b001,
        C=3'b010,
        D=3'b011,
        E=3'b100,
        F=3'b101,
        G=4'b110,
        H=3'b111;

```

Whenever user change the input I will update the status of the machine

```
always@(SW)
begin
    case(SW)
        A: Y_D=A;
        B: Y_D=B;
        C: Y_D=C;
        D: Y_D=D;
        E: Y_D=E;
        F: Y_D=E;
        G: Y_D=G;
        H: Y_D=H;
    endcase
end

```

Then when they press button Key_1 I will update the OUTPUT_SIGNAL

```
always@(posedge KEY[1]) begin
        y_Q = Y_D;
        case (y_Q)
        0: SIGNAL = 14'b00101110000000; // A
        1: SIGNAL = 14'b00111010101000; // B
        2: SIGNAL = 14'b00111010111010; // C
        3: SIGNAL = 14'b00111010100000; // D
        4: SIGNAL = 14'b00100000000000; // E
        5: SIGNAL = 14'b00101011101000; // F
        6: SIGNAL = 14'b00111011101000; // G
        7: SIGNAL = 14'b00101010100000; // H
        default : SIGNAL=14'bxxxxxxxxxxxx;
        endcase
end
```

The value of Signal have meaning that because I use a counter to count half a second and I will to change the value of LED following the index of SIGNAL ("-"=3'b111=1.5 second, "."=1b'1=0.5 second)

```
counter_k_bit ins1(HALFSEC,Clk,KEY[0]);
defparam ins1.n=26;
defparam ins1.k=25000000;//25000000

always @(negedge Clk) begin
    if(HALFSEC==24999999) half=1;//24999999
    else half=0;
end

assign reset=KEY[1] && KEY[0];

counter_k_bit ins2(INDEX,half,reset);
defparam  ins2.n=4;
defparam  ins2.k=14;

always begin
   case (INDEX)
          0:LEDR = SIGNAL[13];
          1:LEDR= SIGNAL[12];
          2:LEDR= SIGNAL[11];
          3:LEDR= SIGNAL[10];
          4:LEDR= SIGNAL[9];
          5:LEDR= SIGNAL[8];
          6:LEDR= SIGNAL[7];
          7:LEDR= SIGNAL[6];
          8:LEDR= SIGNAL[5];
          9:LEDR= SIGNAL[4];
          10:LEDR= SIGNAL[3];
          11:LEDR= SIGNAL[2];
          12:LEDR= SIGNAL[1];
          13:LEDR= SIGNAL[0];
    endcase
end

```

VERIFICATION

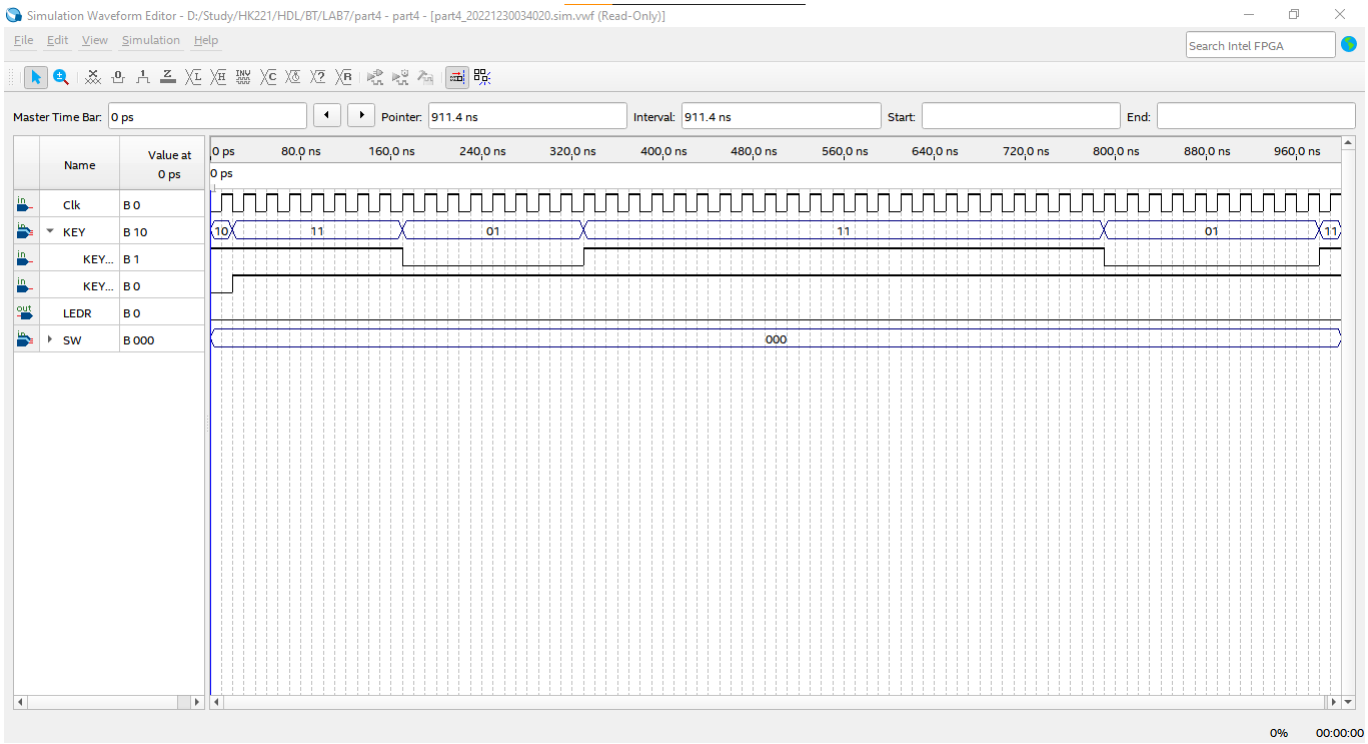


Figure 7.4: LAB 7: Simulation Result for part IV

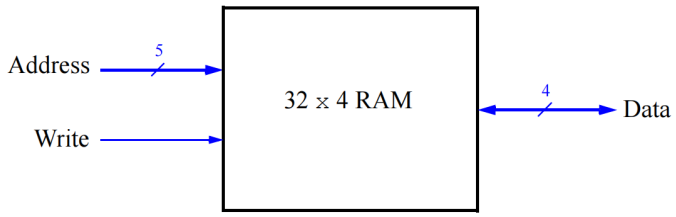
Chapter 8

Memory Blocks

8.1 Introduction

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. 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 1a. It contains 32 four-bit words (rows), which are accessed using a five-bit address port, a four-bit data port, and a write control input.



(*a*) RAM organization

Figure 8.1: LAB 8: Introduction to RAM

8.2 Part III

REQUIREMENT

Instead of creating a memory module subcircuit by using the IP Catalog, we can implement the required memory by specifying its structure in Verilog code. In a Verilog-specified design it is possible to define the memory as amultidimensional array.

SOLUTION

One thing to notice about the RAM IP - which is used in part I and part II is that at the time we write new data to the RAM, that value immediately appears on the output.

```
1 module part3 (CLK, ADDRESS, DATA_IN, WREN, display_add0,
      display_add1, display_in, display_out);
2     input     CLK, WREN;
3     input     [4:0]ADDRESS;
4     input     [3:0]DATA_IN;
5     output    [6:0]display_in, display_out, display_add0,
      display_add1;
6
7     reg       [3:0]RAM32X4[31:0];
8     reg       [3:0]DATA_OUT;
9
10    always @(posedge CLK)
11       if (WREN) RAM32X4[ADDRESS] <= DATA_IN;
12
13    always @(posedge CLK)
14       if (WREN) DATA_OUT <= DATA_IN;
15       else DATA_OUT <= {4{~WREN}} & RAM32X4[ADDRESS];
16
17    control7SEG data_in  (.IN(DATA_IN),     .OUT(display_in));
18    control7SEG data_out (.IN(DATA_OUT),    .OUT(display_out));
19    control7SEG address0 (.IN(ADDRESS%16), .OUT(display_add0));
20    control7SEG address1 (.IN(ADDRESS/16), .OUT(display_add1));
21
22 endmodule
23
```

8.3 Part IV

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 project for your circuit. To generate the desired memory module open the IP Catalog and select the RAM: 2-PORT module in the Basic Functions > On Chip Memory category. As shown in Figure 5, choose With one read port and one write port in the category called How will you be using the dual port ram?

2. Configure the memory size, clocking method, and registered ports the same way as Part II. As shown in Figure 6 select I do not care (The outputs will be undefined) for 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 during a write operation.

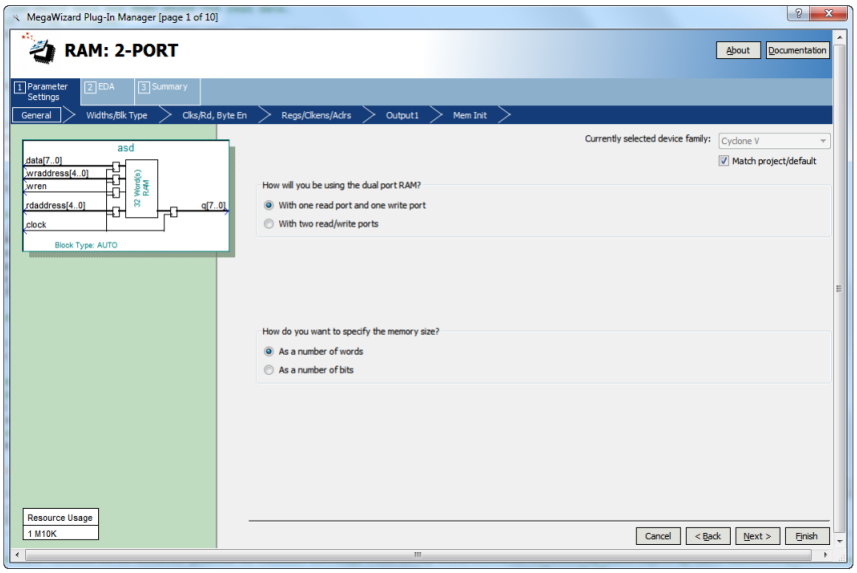


Figure 5: Configuring the two input ports of the RAM.

Figure 8.2: LAB 8: RAM IP configuration (1)

3. You will need to create a MIF file like the one in Figure 8 to test your circuit. Finish the Wizard and then examine the generated memory module in the file ram32x4.v.

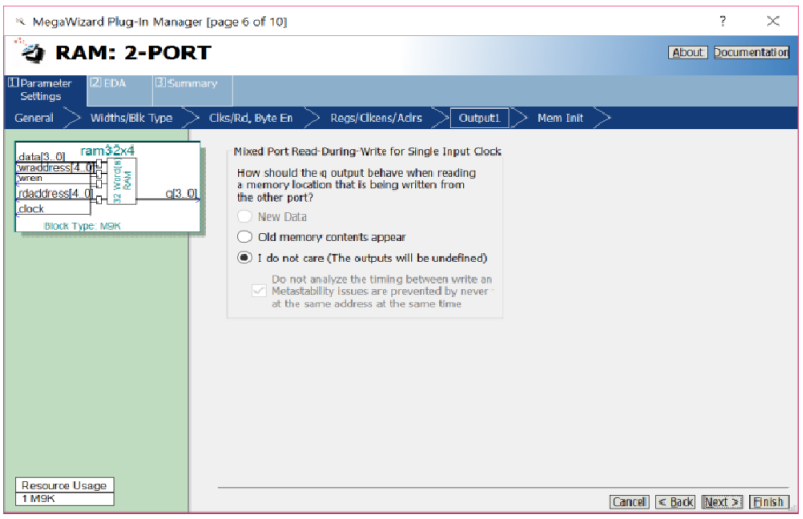


Figure 6: Configuring the output of the RAM when reading and writing to the same address.

Figure 8.3: LAB 8: RAM IP configuration (2)

4. 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 four-bit word (in hexadecimal format) on the 7-segment display HEX_0. Use a counter as a read address, and scroll through the memory locations by displaying each word for about one second. As each word is being displayed, show its address (in hex format) on the 7-segment displays HEX_{3-2}. Use the 50 MHz clock, $CLOCK50$, and use KEY_0 as a reset input. For the write address and corresponding data use switches SW_{8-4} and SW_{3-0}. Show the write address on HEX_{5-4} and show the write data on HEX_1.

5. Test your circuit and verify that the initial contents of the memory match your $ram32x4.mif file$.

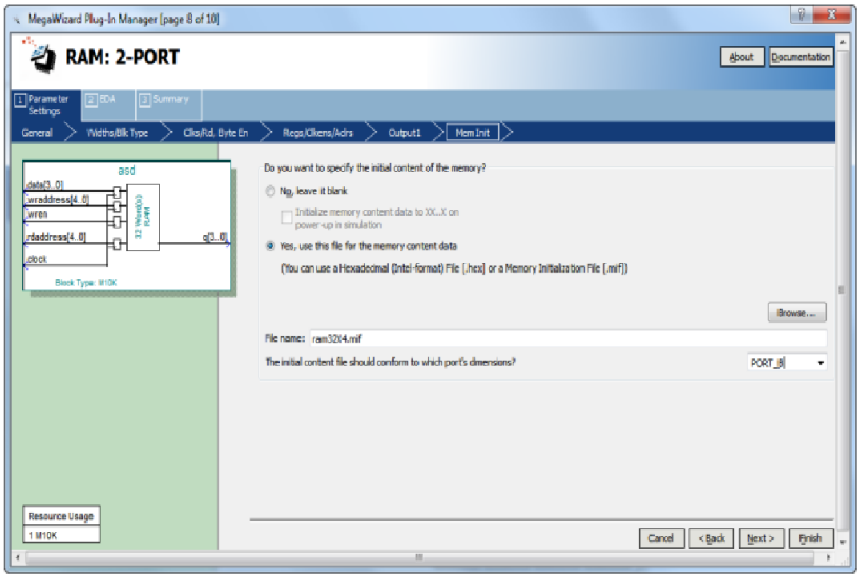


Figure 8.4: LAB 8: RAM IP configuration (3)

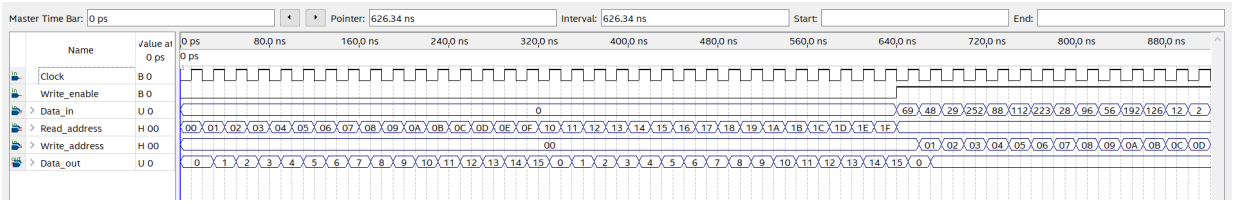


Figure 8.5: LAB 8: Simulation Resule for part IV

The following is the code of ram 32x4.mif file which is initialized for the memory.

```
WIDTH=4;
DEPTH=32;
ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;
CONTENT BEGIN
    0    :    1;
    1    :    2;
    2    :    0;
    3    :    5;
    4    :    0;
    5    :    1;
    [6..7]   :    0;
    [8..9]   :    1;
    10   :    0;
    11   :    1;
    12   :    10;
    13   :    0;
```

```
18    14  :   1;
19    [15..16]  :   0;
20    [17..18]  :   1;
21    [19..20]  :   0;
22    21  :   11;
23    22  :   0;
24    [23..25]  :   1;
25    26  :   0;
26    27  :   15;
27    [28..31]  :   0;
28END;
```

Chapter 9

A Simple Processor

9.1 Introduction

A **central processing unit (CPU)**, also called a **central processor**, **main processor** or just **processor**, is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program. This contrasts with external components such as main memory and I/O circuitry, and specialized processors such as graphics processing units (GPUs).

The form, design, and implementation of CPUs have changed over time, but their fundamental operation remains almost unchanged. Principal components of a CPU include the arithmetic logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations, and a control unit that orchestrates the fetching (from memory), decoding and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

The following figure depicts the internals of a CPU.

In which,

- The **resistors** R_0, R_1, R_2, etc. are basically the CPU's internal RAM, and it will only have a small number of these. These resistors can either store numeric values or specific functions.
- The **arithmetic logic unit** is responsible for performing additions, subtractions, logic ands and ors, and other source of computation.
- The **status flags** registor is a collection of bits which will gives us information about the status of the CPU and the ALU.
- The **program counter** is used to store the address of where we are up to in our program. So, as the program is executed sequentially, the program counter in-

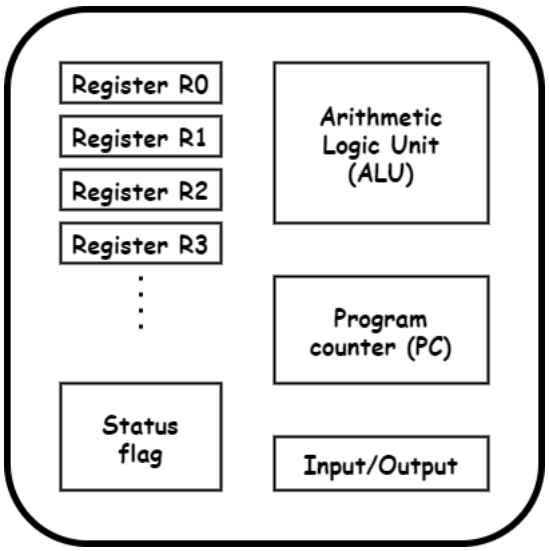


Figure 9.1: LAB 9: Hint for part I

creases. Furthermore, its value can be set depending on the result of something in the **status flags** register.

- **Input/Output** is how we are going to communicate, meaning getting the data in and out of the CPU.

9.2 Part I

The following figure shows a *processor* that contains a number of nine-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine). Data is input to this system via the nine-bit DIN input. This data can be loaded through the nine-bit wide multiplexer into the various registers, such as R_0, . . . , R_7 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 of signed numbers is performed by using the multiplexer to first place one nine-bit number onto the bus wires and loading this number into register A. Once this is done, a second nine-bit number is placed onto the bus, the adder/-subtractor 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

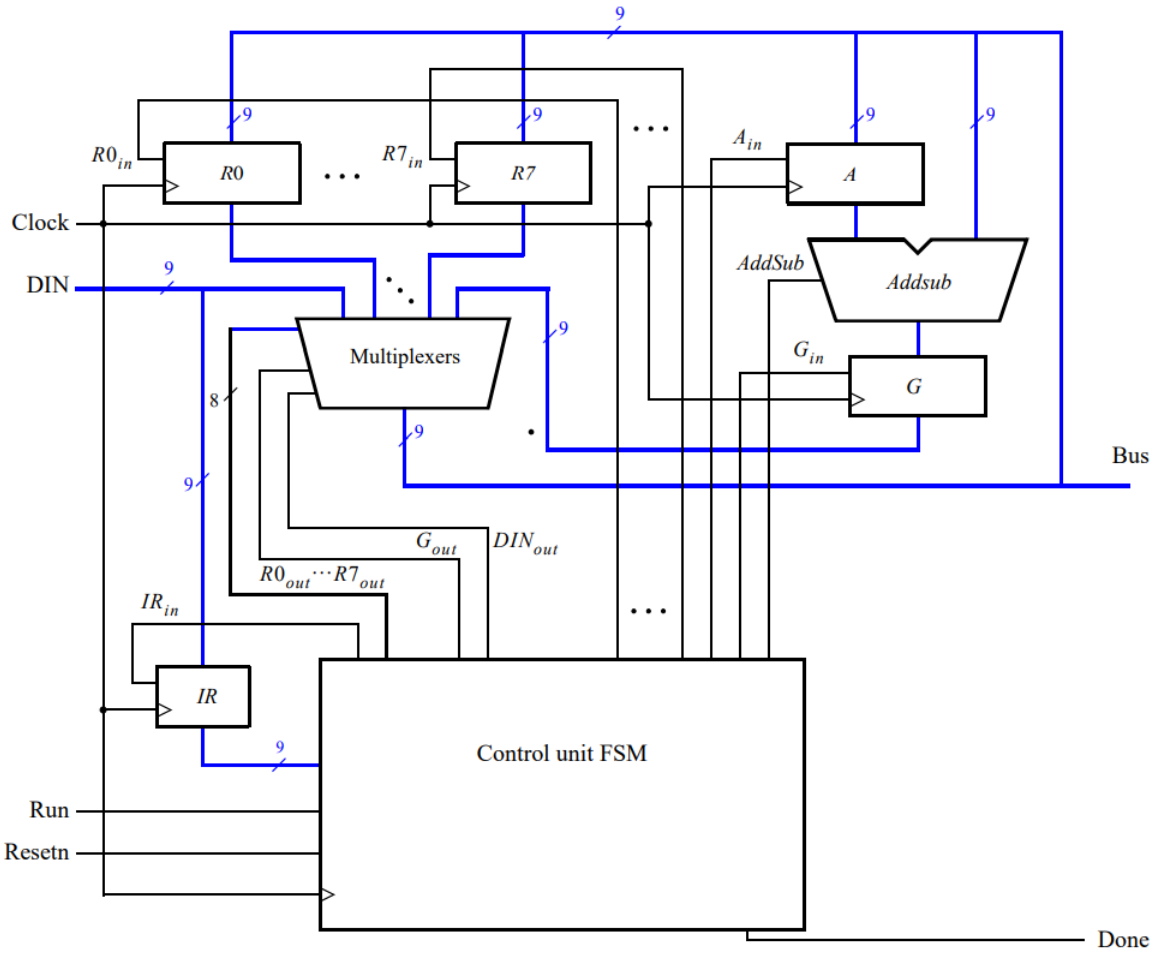


Figure 9.2: LAB 9: Hint for part I (Diagram)

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 R_0 onto the bus and this data will be loaded on the next active clock edge into A.

The *processor* executes operations specified in the form of *instructions*. The following table lists the instructions the processor has to support. The left column shows the name of an instruction and its operands. The meaning of the syntax $Rx \leftarrow [Ry]$ is that the contents of resister 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 $Rx \leftarrow D$ indicates that the nine-bit constant D is loaded into register Rx.

Each instruction can be encoded using the nine-bit format *IIIXXXYYY* where *III* specifies the instruction, *XXX* gives the Rx register, and *YYY* gives the Ry register. Al-

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx,Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Ry,Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 9.1: LAB 9: Table of instruction for simple processor

though only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor later. Assume that *III* = 000 for the **mv** instruction, 001 for **mvi**, 010 for **add**, and 011 for **sub**. Instructions are loaded from the the external input *DIN*, and stored into the *IR* register, using the connection indicated above. For the *mvi* instruction, the *YYY* field has no meaning, and the immediate data #*D* has to be supplied on the *DIN* input in the clock cycle 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 finite state machine in the control unit "steps through" such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. 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. The following table indicates the control signals that can be asserted in each time step to implement the instructions in the previous table. Note that the only control signal asserted in time step 0 is IR_{in}, so this time step is not shown in the table.

	T1	T2	T3
(**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}	G_{out}, Rx_{in}, *done*

Table 9.2: LAB 9: Detailed executing step for each instruction

The following block of **Verilog** code can be used to model the described processor.

```
module pro(CLK,D_IN, RUN, RESET, DONE, BUS);
  input    [8:0]    D_IN;
  input             CLK, RESET, RUN;
  output   [8:0]    BUS /*synthesis keep*/;
  output            DONE;

  wire     [8:0]    reg0_out, reg1_out, reg2_out, reg3_out, reg4_out,
   reg5_out, reg6_out, reg7_out, regA_out, regG_out /*synthesis keep*/;
  wire     [8:0]    addsub_out /*synthesis keep*/;
  wire     [9:0]    reg_en /*synthesis keep*/;
  wire     [9:0]    multi_select /*synthesis keep*/;
  wire     [8:0]    CODE /*synthesis keep*/;
  wire              MODE /*synthesis keep*/;

  regn  reg0 (.CLK(CLK), .EN(reg_en[0]), .IN(BUS),
   .OUT(reg0_out));
  regn  reg1 (.CLK(CLK), .EN(reg_en[1]), .IN(BUS), .OUT(reg1_out));
  regn  reg2 (.CLK(CLK), .EN(reg_en[2]), .IN(BUS), .OUT(reg2_out));
  regn  reg3 (.CLK(CLK), .EN(reg_en[3]), .IN(BUS), .OUT(reg3_out));
  regn  reg4 (.CLK(CLK), .EN(reg_en[4]), .IN(BUS), .OUT(reg4_out));
  regn  reg5 (.CLK(CLK), .EN(reg_en[5]), .IN(BUS), .OUT(reg5_out));
  regn  reg6 (.CLK(CLK), .EN(reg_en[6]), .IN(BUS), .OUT(reg6_out));
  regn  reg7 (.CLK(CLK), .EN(reg_en[7]), .IN(BUS), .OUT(reg7_out));
  regn  regA (.CLK(CLK), .EN(reg_en[8]), .IN(BUS), .OUT(regA_out));
  regn  regG (.CLK(CLK), .EN(reg_en[9]), .IN(addsub_out),
   .OUT(regG_out));
  regn  regI (.CLK(CLK), .EN(RUN), .IN(D_IN), .OUT(CODE));

  multiplexer inst0 (.IN0(reg0_out), .IN1(reg1_out), .IN2(reg2_out),
                     .IN3(reg3_out), .IN4(reg4_out), .IN5(reg5_out),
                     .IN6(reg6_out), .IN7(reg7_out), .IN8(regG_out),
                     .IN9(D_IN), .SELECT(multi_select), .OUT(BUS));

  addsub      inst1 (.MODE(MODE), .IN0(regA_out), .IN1(BUS),
                     .OUT(addsub_out));

  control     inst2 (.CLK(CLK), .RUN(RUN), .RESET(RESET), .DONE(DONE),
                     .CODE(CODE), .REG_EN(reg_en),
                     .MULTI_SELECT(multi_select), .MODE(MODE));
endmodule

module control(CLK, CODE, RUN, RESET, DONE, MODE, REG_EN, MULTI_SELECT);
  parameter MV=3'b000, MVI=3'b001, ADD=3'b010, SUB=3'b011, LDY=3'b100,
   UDX=3'b101, LDI=3'b110;
  input    [8:0]    CODE;
  input             RUN, RESET, CLK;
```

```
43
44  output   reg [9:0]   REG_EN;
45  output   reg [9:0]   MULTI_SELECT;
46  output   reg         DONE, MODE;
47
48  reg      [2:0]    fsm_in, fsm_out /*synthesis keep */;
49
50
51  //FSM
52  always @(posedge CLK) begin
53     fsm_out <= fsm_in;
54  end
55
56  always @(CODE) begin
57     case (fsm_out)
58        MV: fsm_in = CODE [8:6];
59        MVI:fsm_in = LDI;
60        ADD:fsm_in = LDY;
61        SUB:fsm_in = LDY;
62        LDY:fsm_in = UDX;
63        UDX:fsm_in = CODE [8:6];
64        default: fsm_in = CODE [8:6];
65     endcase
66  end
67
68  //Control
69  always @(fsm_in) begin
70        if (fsm_in==MV) begin
71           MULTI_SELECT = {10{1'b0}} | (1<<CODE[2:0]);
72           REG_EN       = {10{1'b0}} | (1<<CODE[5:3]);
73           DONE         = 1 ;
74        end else
75        if (fsm_in==MVI) begin
76           MULTI_SELECT = {10{1'b0}} | (1<<9);
77           REG_EN       = {10{1'b0}} | (1<<CODE[5:3]);
78           DONE         = 0 ;
79        end else
80        if (fsm_in==LDI) begin
81           REG_EN       = {10{1'b0}} | (0<<CODE[5:3]);
82           DONE         = 1 ;
83        end else
84        if (fsm_in==ADD | fsm_in==SUB) begin
85           MULTI_SELECT = {10{1'b0}} | (1<<CODE[5:3]);
86           REG_EN       = {10{1'b0}} | (1<<8);
87           DONE         = 0 ;
88        end else
89        if (fsm_in==LDY) begin
90           MULTI_SELECT = {10{1'b0}} | (1<<CODE[2:0]);
```

```
91              REG_EN       = {10{1'b0}} | (1<<9);
92              DONE         = 0 ;
93           end else
94           if (fsm_in==UDX) begin
95              MULTI_SELECT = {10{1'b0}} | (1<<8);
96              REG_EN       = {10{1'b0}} | (1<<CODE[5:3]);
97              DONE         = 1 ;
98           end
99
100          if (fsm_in==ADD) MODE = 1'b0 ;
101          if (fsm_out==SUB)MODE = 1'b1 ;
102
103       end
104
105 endmodule
106
107 module multiplexer(IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9,
      SELECT, OUT);
108    input     [8:0]     IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9;
109    input     [9:0]     SELECT;
110    output    [8:0]     OUT /*synthesis keep*/;
111
112    assign    OUT = IN0 & {9{SELECT[0]}} |
113                    IN1 & {9{SELECT[1]}} |
114                    IN2 & {9{SELECT[2]}} |
115                    IN3 & {9{SELECT[3]}} |
116                    IN4 & {9{SELECT[4]}} |
117                    IN5 & {9{SELECT[5]}} |
118                    IN6 & {9{SELECT[6]}} |
119                    IN7 & {9{SELECT[7]}} |
120                    IN8 & {9{SELECT[8]}} |
121                    IN9 & {9{SELECT[9]}} ;
122 endmodule
123
124 module addsub(MODE, IN0, IN1, OUT);
125    parameter ADD=1'b0, SUB=1'b1;
126
127    input     [8:0] IN0, IN1 /*synthesis keep*/;
128    input           MODE;
129    output    reg[8:0] OUT;
130
131
132    always @(IN0, IN1) begin
133       if (MODE==ADD) OUT = IN1 + IN0;
134       else OUT = IN1 - IN0;
135    end
136
137
```

```
endmodule

module regn(IN, EN, CLK, OUT);
parameter n = 9;
   input     [n-1:0] IN;
   input     EN, CLK;
   output    reg [n-1:0] OUT;

   always @(posedge CLK)
      if (EN)
         OUT <= IN;
endmodule
```

Using **Quartus**, we can generate a schematic block diagram and a waveform simulation, and the results are as followed.

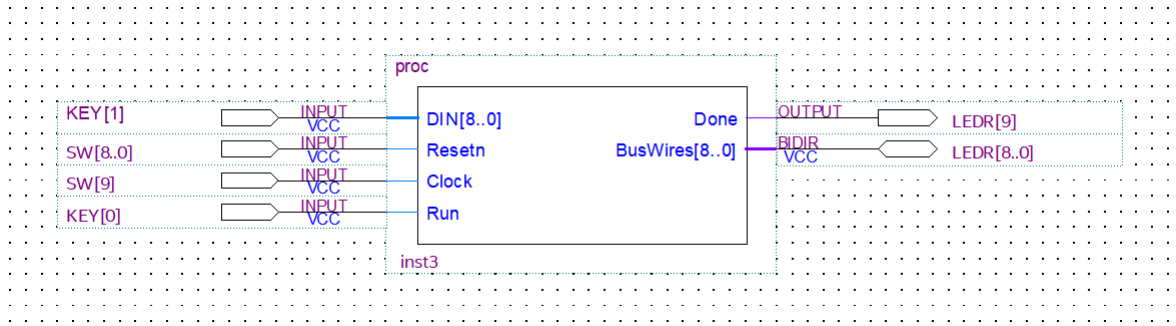


Figure 9.3: LAB 9: Schematic for part 1

Here are some explanations for the simulation result above:

- At $t = 10ns$, the instruction 120 (9'b001010000) $\Longleftrightarrow$ "**mvi** R2" is loaded into processor via **Run** signal, then at $t = 15ns$ it is executed **Run** signal. In the next clock pulse (at $t = 25ns$), the **value 050 is loaded into R2** and the signal **Done** is immediately returned. **This operation takes 2 clock pulses to complete.**
- At $t = 20ns$, the instruction 120 (9'b001011000) $\Longleftrightarrow$ "**mvi** R3" is loaded into processor via **Run** signal, then at $t = 25ns$ it is executed. In the next clock pulse (at $t = 35ns$), the **value 050 is loaded into R2** and the signal **Done** is immediately returned. **This operation takes 2 clock pulses to complete.**
- At $t = 50ns$, the instruction 110 (9'b001001000) $\Longleftrightarrow$ "**mvi** R1" is loaded into BUS_WIRE, then at $t = 55ns$ flows to the processor via the **Run** signal. In the next clock pulse (at $t = 65ns$), the **value 001 is loaded into R1** and the signal **Done** is immediately returned. **This operation takes 2 clock pulses to complete.**
- At $t = 70ns$, the instruction 332 (9'b011011010) $\Longleftrightarrow$ "**sub** R3,R2" is loaded into processor via **Run** signal, then it is executed at $t = 75ns$ - this is the reason why

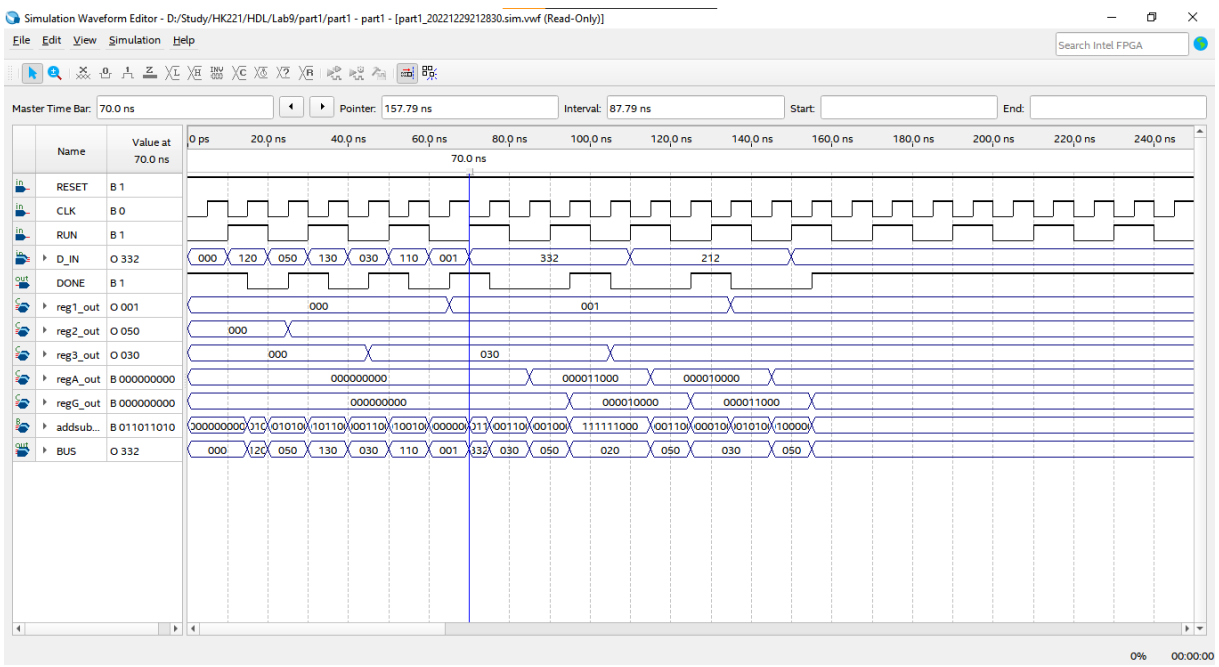


Figure 9.4: LAB 9: Stimulation Result for part I

at this time, the BUS_WIRE hold the value of R3 (020), immediately, this value flows to the **Addsub** module. At the next clock pulse, the value of the R_2 (050) is loaded into the BUS_WIRE, then flow to Addsub to compute the value, in this case, this module minus 020 from 050 and save into R_3. This operation takes 3 clock pulses to complete.

9.3 Part II

Let's take this one step further by adding a memory module and a counter to our processor. 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 will use separate clock signals, PClock and MClock, for the processor and the memory.

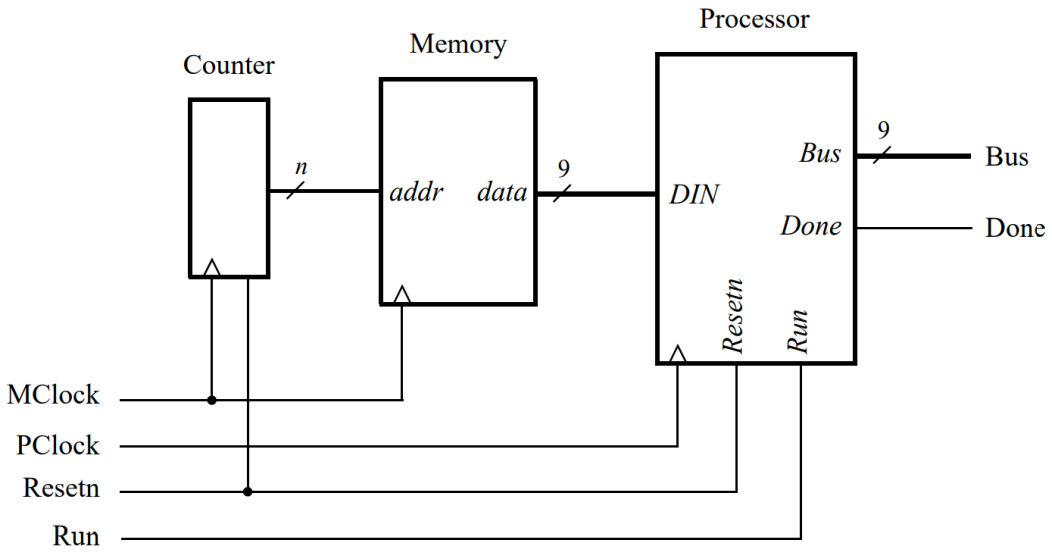


Figure 9.5: LAB 9: Hint for part II (Diagram)

The counter we will be using is just a simple mod-32 counter. The **verilog** code for the counter is as followed.

```
module counterv(CLK,RESET,Q);
   input CLK,RESET;
   output reg [4:0] Q;
   always@(posedge CLK) begin
      if(!RESET) Q<=0;
      else Q<=Q+1;
   end
endmodule
```

Next, the memory we will be using will be called a *synchronous read-only memory (synchronous ROM)* since it has only a read port and no write port. Using the Quartus IP Catalog tool, we can create this memory module as follow.

We can initialize the initial content of the memory by using a *Memory Initialization File* [.mif]. So we created such file, named *inst_mem.mif* and its content is as followed.

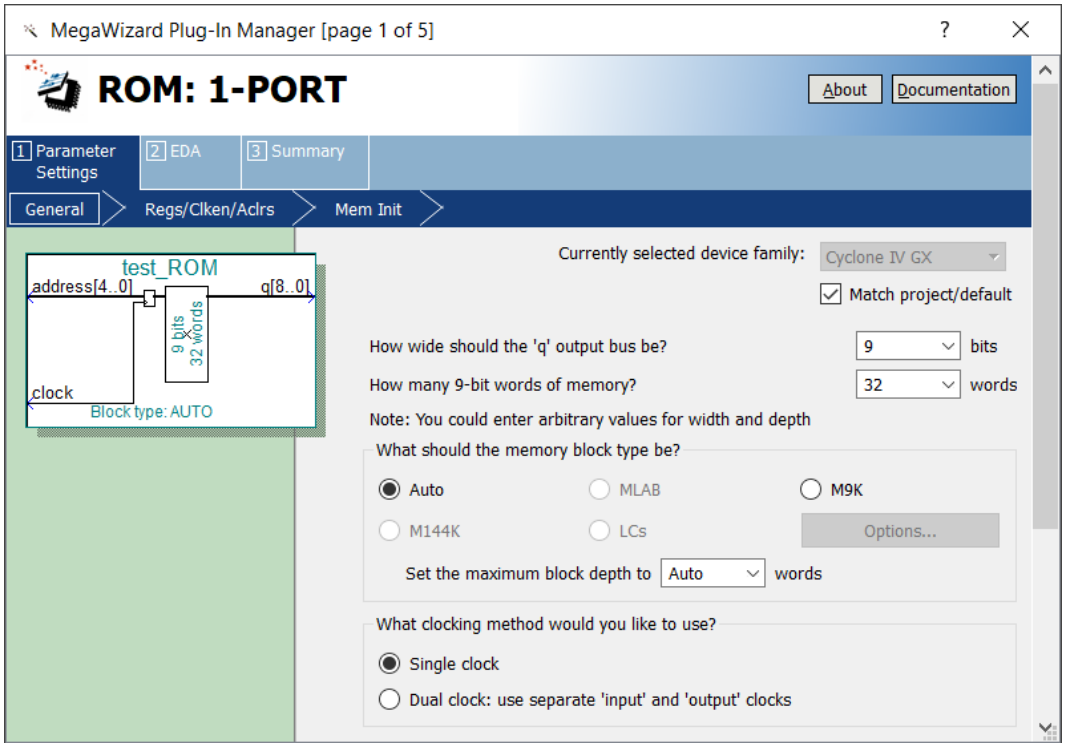


Figure 9.6: LAB 9: ROM IP configuration (1)

```
1
2 WIDTH=9;
3 DEPTH=32;
4
5 ADDRESS_RADIX=UNS;
6 DATA_RADIX=OCT;
7
8 CONTENT BEGIN
9     0   :   100;
10    1   :   005;
11    2   :   010;
12    3   :   201;
13    4   :   300;
14    [5..31]  :   000;
15 END;
```

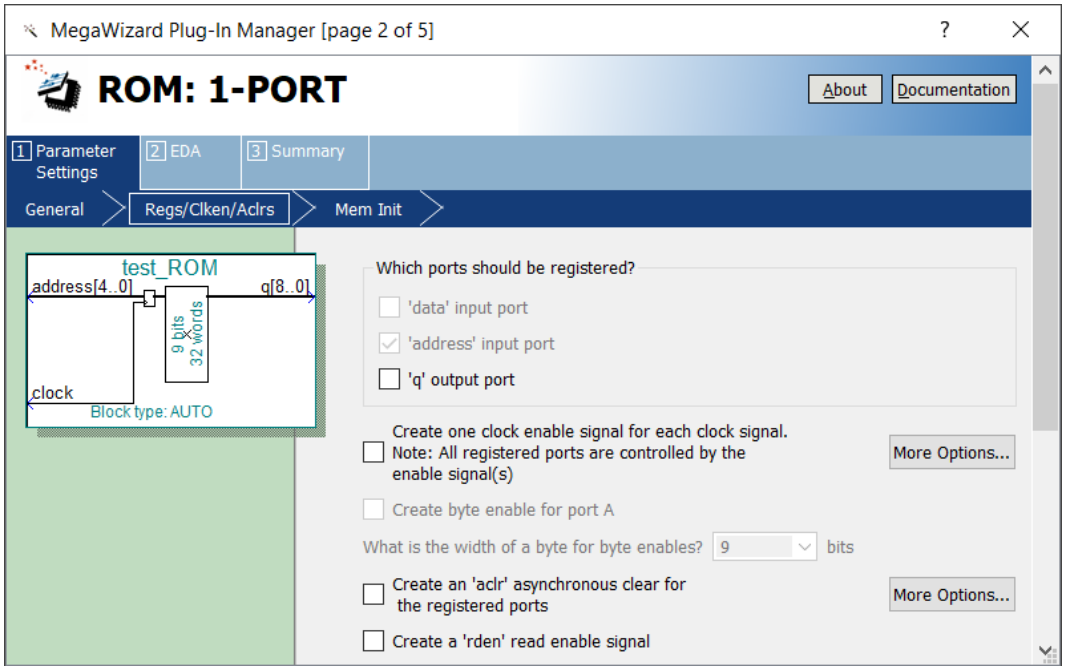


Figure 9.7: LAB 9: ROM IP configuration (2)

Using **Quartus**, we can generate a schematic block diagram and a waveform simulation.

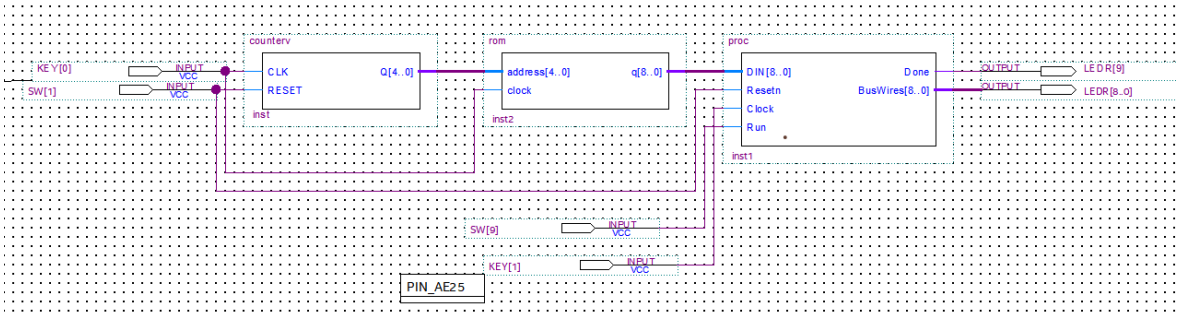


Figure 9.8: LAB 9: Schematic for part II

Chapter 10

An Enhanced Processor

10.1 Part III

REQUIREMENT

1. 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 below.

Operation	Function performed
ld Rx,$[Ry]$	$Rx \leftarrow [[Ry]]$
st Rx,$[Ry]$	$[Ry] \leftarrow [Rx]$
mvnz Rx, Ry	if G != 0, $Rx \leftarrow [Ry]$

Figure 10.1: LAB 10: Added instructions

2. A schematic of the enhanced processor is given in Figure 11. 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.

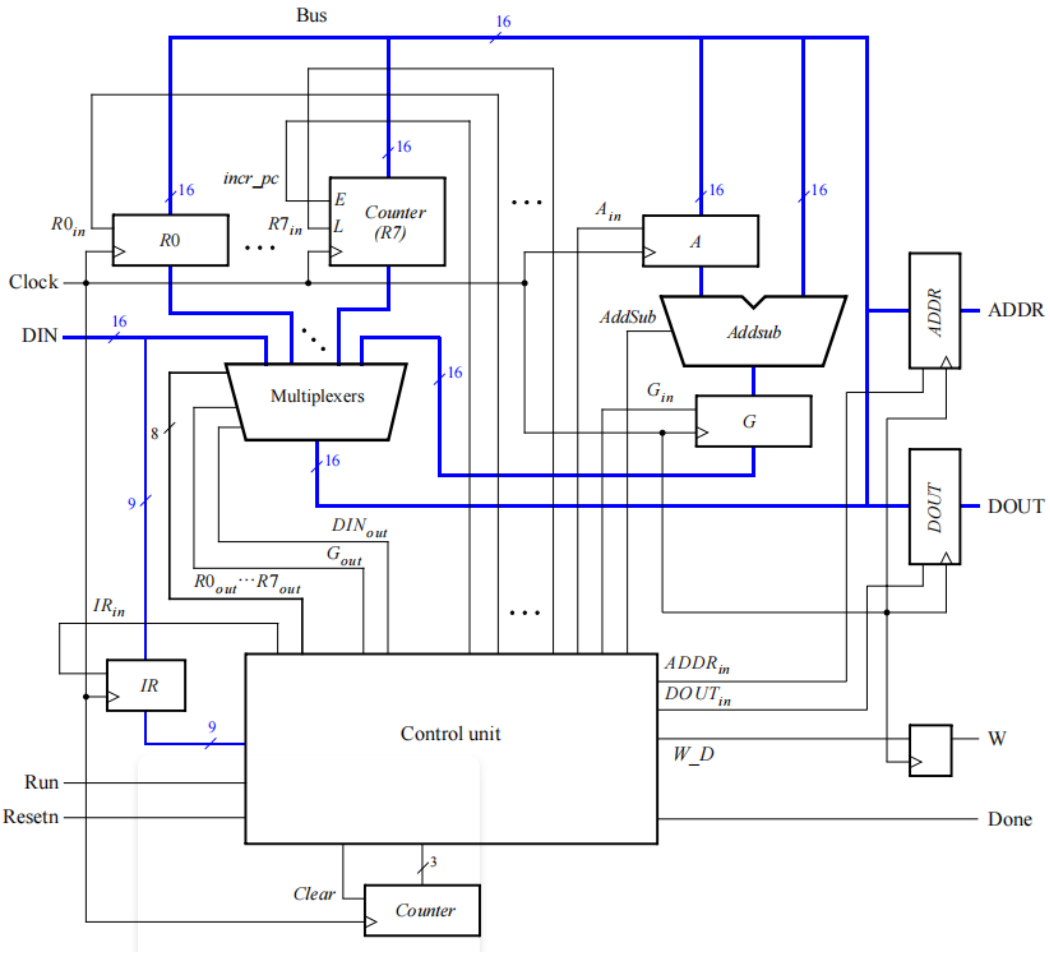


Figure 10.2: LAB 10: Hint for part III (Diagram)

3. 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 an **mv** or **mvi** instruction in which the destination register is specified as R7. In this case, the control unit uses the signal R7in 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
4. Figure 11 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 pro-

```
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 10.3: LAB 10: Example for Loops

vided 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.

5. Figure 12 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 static random access memory (synchronous SRAM). Figure 12 also includes a 9-bit register that can be used to store data 2 ADDR DOUT from the processor; this register might be connected to a set of LEDs to allow display of data on your DE-series 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 A8A7 = 00, then the memory module will be written at the address given on the lower address lines. Figure 12 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 A6 . . . A0. For addresses in which A8A7 = 01, the data written by the processor is loaded into the register whose outputs are called LEDs in Figure 12.

SOLUTION

Entire Design

```
1 module part3 (CLOCK_50, SW, KEY, LEDR);
2   input               CLOCK_50;
3   input         SW;
4   input         KEY;
```

```
  output  reg [8:0] LEDR;

  wire [8:0] DIN, ADDR, DOUT, LEDsOUT  ;
  wire Resetn, Run, W            ;
  wire newclock;
  reg LEDen, MEMen;
  counter_modk C_new (CLOCK_50, 1, newclock);
  defparam C_new.n = 26;
  defparam C_new.k = 2;

  assign Run      = SW;
  assign Resetn   = KEY;
  always begin
   if (LEDen==1'b1) LEDR = LEDsOUT;
  end

  always
  begin
    LEDen = W & ~(ADDR[8] | ~ADDR[7]);
    MEMen = W & ~(ADDR[8] | ADDR[7]);
  end

  proc2     P0 (DIN, Resetn, newclock, Run, ADDR, DOUT, W);
  regn      LEDs (DOUT, 1, newclock, LEDsOUT);
  ramlpm    Memory (ADDR, newclock, DOUT, MEMen, DIN);

endmodule

```

proc2 module: this processor is quite similar to the **proc** module we use in the simple processor, but it is added some state to handle new instruction.

Furthernmore, the register R7 is now modified to implement parallel load.

```
module proc2 (DIN, Resetn, Clock, Run, ADDR, DOUT, W);
  input [8:0] DIN;
  input Resetn, Clock, Run;
  reg Done;
  reg [8:0] BusWires /*synthesis keep*/;
  output [8:0] ADDR, DOUT;
  output W;

  //declare variables
  reg IRin, DINout, Ain, Gout, Gin, AddSub, incr_pc, ADDRin,
    DOUTin, W_D /*synthesis keep*/;
  reg [7:0] Rout, Rin;
  wire [7:0] Xreg, Yreg;
  wire [0:8] IR   /*synthesis keep*/;
```

```
14  wire [0:2] I    /*synthesis keep*/;
15  reg [9:0] MUXsel;
16  wire [8:0] R0, R1, R2, R3, R4, R5, R6, R7, result;
17  wire [8:0] A, G;
18  wire [2:0] Tstep_Q;
19
20  wire Clear = Done || ~Resetn;
21  upcount Tstep (Clear, Clock, Tstep_Q);
22  assign I = IR[0:2];
23  dec3to8 decX (IR[3:5], 1'b1, Xreg);
24  dec3to8 decY (IR[6:8], 1'b1, Yreg);
25  always @(Tstep_Q or I or Xreg or Yreg)
26  begin
27    //specify initial values
28    IRin = 1'b0;
29    Rout[7:0] = 8'b00000000;
30    Rin[7:0] = 8'b00000000;
31    DINout = 1'b0;
32    Ain = 1'b0;
33    Gout = 1'b0;
34    Gin = 1'b0;
35    AddSub = 1'b0;
36    DOUTin = 1'b0;
37    ADDRin = 1'b0;
38    W_D = 1'b0;
39    incr_pc = 1'b0;
40
41    Done = 1'b0;
42
43    case (Tstep_Q)
44      3'b000: // load next instruction in time step 0
45      begin
46        Rout = 8'b10000000;
47        ADDRin = 1'b1;
48        incr_pc = 1'b1;
49      end
50      3'b001: // store next instruction in time step 1
51      begin
52        IRin = 1'b1 & Run; // should this be ANDed with Run?
53        ADDRin = 1'b1;
54      end
55      3'b010: //define signals in time step 1
56        case (I)
57          3'b000: // mv
58          begin
59            Rout = Yreg;
60            Rin = Xreg;
61            Done = 1'b1;
```

```
62            end
63            3'b001: // mvi
64            begin
65              DINout = 1'b1;
66              Rin = Xreg;
67              Done = 1'b1;
68              incr_pc = 1'b1;
69            end
70            3'b010: // add
71            begin
72              Rout = Xreg;
73              Ain = 1'b1;
74            end
75            3'b011: // sub
76            begin
77              Rout = Xreg;
78              Ain = 1'b1;
79            end
80            3'b100: // ld
81            begin
82              Rout = Yreg;
83              ADDRin = 1'b1;
84            end
85            3'b101: // st
86            begin
87              Rout = Xreg;
88              DOUTin = 1'b1;
89            end
90            3'b110: // mvnz
91            begin
92              if (G != 0) begin
93                Rout = Yreg;
94                Rin = Xreg;
95              end
96              Done = 1'b1;
97            end
98          endcase
99        3'b011: //define signals in time step 2
100         case (I)
101           3'b010: // add
102           begin
103             Rout = Yreg;
104             Gin = 1'b1;
105           end
106           3'b011: // sub
107           begin
108             Rout = Yreg;
109             Gin = 1'b1;
```

```
110             AddSub = 1'b1;
111           end
112           3'b100: // ld
113           begin
114             DINout = 1'b1;
115             Rin = Xreg;
116             Done = 1'b1;
117           end
118           3'b101: // st
119           begin
120             Rout = Yreg;
121             ADDRin = 1'b1;
122             W_D = 1'b1;
123           end
124         endcase
125       3'b100: //define signals in time step 3
126         case (I)
127           3'b010: // add
128           begin
129             Gout = 1'b1;
130             Rin = Xreg;
131             Done = 1'b1;
132           end
133           3'b011: // sub
134           begin
135             Gout = 1'b1;
136             Rin = Xreg;
137             Done = 1'b1;
138           end
139         endcase
140     endcase
141   end
142
143   counterlpm reg_7 (1'b1, Clock, incr_pc, BusWires, ~Resetn,
       Rin[7], R7);
144   regn reg_0 (BusWires, Rin[0], Clock, R0);
145   regn reg_1 (BusWires, Rin[1], Clock, R1);
146   regn reg_2 (BusWires, Rin[2], Clock, R2);
147   regn reg_3 (BusWires, Rin[3], Clock, R3);
148   regn reg_4 (BusWires, Rin[4], Clock, R4);
149   regn reg_5 (BusWires, Rin[5], Clock, R5);
150   regn reg_6 (BusWires, Rin[6], Clock, R6);
151   regn reg_IR (DIN, IRin, Clock, IR);
152   defparam reg_IR.n = 9;
153   regn reg_A (BusWires, Ain, Clock, A);
154   regn reg_G (result, Gin, Clock, G);
155
156   regn reg_ADDR (BusWires, ADDRin, Clock, ADDR);
```

```
157   regn reg_DOUT (BusWires, 1, Clock, DOUT);
158   regn reg_W (W_D, 1'b1, Clock, W);
159   defparam reg_W.n = 1;
160
161   addsub AS (~AddSub, A, BusWires, result);
162
163   //define the bus
164   always @ (MUXsel or Rout or Gout or DINout)
165   begin
166     MUXsel[9:2] = Rout;
167     MUXsel[1] = Gout;
168     MUXsel[0] = DINout;
169
170     case (MUXsel)
171       10'b0000000001: BusWires = DIN;
172       10'b0000000010: BusWires = G;
173       10'b0000000100: BusWires = R0;
174       10'b0000001000: BusWires = R1;
175       10'b0000010000: BusWires = R2;
176       10'b0000100000: BusWires = R3;
177       10'b0001000000: BusWires = R4;
178       10'b0010000000: BusWires = R5;
179       10'b0100000000: BusWires = R6;
180       10'b1000000000: BusWires = R7;
181     endcase
182   end
183
184 endmodule
185
186 module upcount(Clear, Clock, Q);
187   input Clear, Clock;
188   output [2:0] Q;
189   reg [2:0] Q;
190
191   always @(posedge Clock)
192     if (Clear)
193       Q <= 3'b0;
194     else
195       Q <= Q + 1'b1;
196 endmodule
197
198 module dec3to8(W, En, Y);
199   input [2:0] W;
200   input En;
201   output [0:7] Y;
202   reg [0:7] Y;
203
204   always @(W or En)
```

```
205   begin
206     if (En == 1)
207       case (W)
208         3'b000: Y = 8'b10000000;
209         3'b001: Y = 8'b01000000;
210         3'b010: Y = 8'b00100000;
211         3'b011: Y = 8'b00010000;
212         3'b100: Y = 8'b00001000;
213         3'b101: Y = 8'b00000100;
214         3'b110: Y = 8'b00000010;
215         3'b111: Y = 8'b00000001;
216       endcase
217     else
218       Y = 8'b00000000;
219   end
220 endmodule
221
```

Memory file

```
1  WIDTH=9;
2  DEPTH=128;
3  ADDRESS_RADIX=HEX;
4  DATA_RADIX=BIN;
5  CONTENT BEGIN
6     00  :   001001000;
7     01  :   000000001;
8     02  :   001010000;
9     03  :   000000000;
10    04  :   001011000;
11    05  :   010000000;
12    06  :   101010011;
13    07  :   010010001;
14    08  :   001011000;
15    09  :   111111111;
16    0A  :   000101111;
17    0B  :   001100000;
18    0C  :   111111111;
19    0D  :   000000111;
20    0E  :   011100001;
21    0F  :   110111000;
22    10  :   011011001;
23    11  :   110111101;
24    12  :   001111000;
25    13  :   000000100;
26    [14..7F]  :   000000000;
27 END;
```

VERIFICATION

We failed to implementation the enhanced processor. The main reason is we cannot control the flow of data from memory by using new version of register R7.

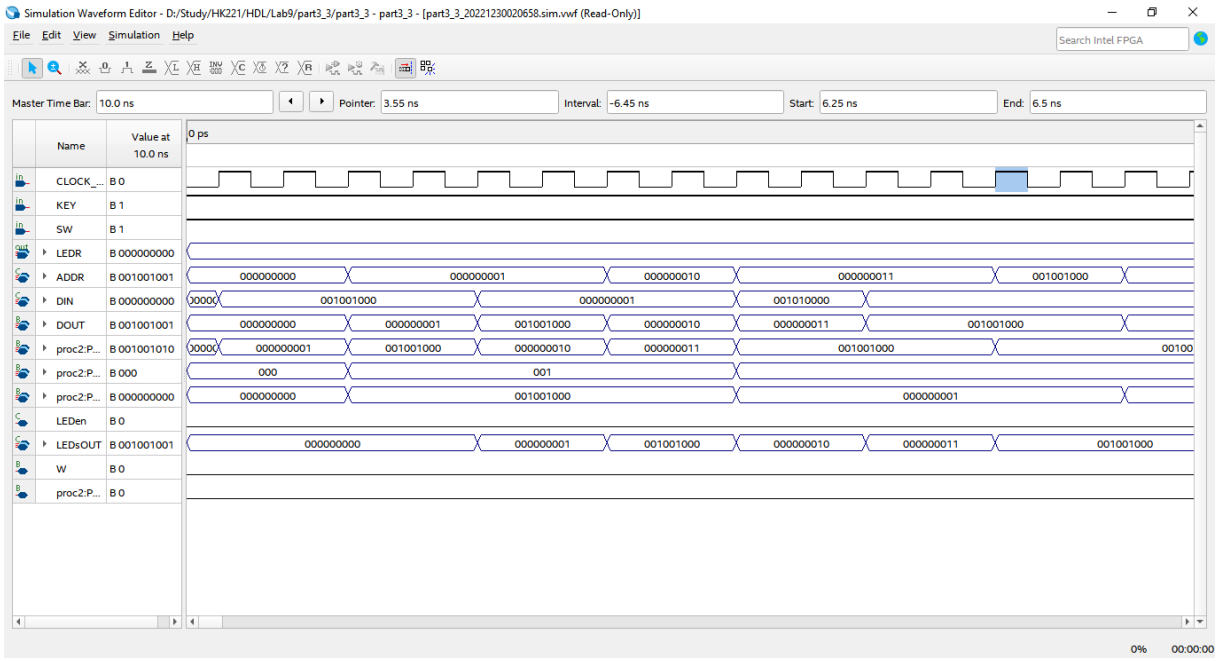


Figure 10.4: LAB 10: Stimulation Result for Part III

Chapter 11

Implementing Algorithms in Hardware

11.1 Introduction

This is an exercise in using algorithmic state machine charts to implement algorithms as hardware circuits.

Algorithmic State Machine (ASM) charts are a design tool that allow the specification of digital systems in a form similar to a flow chart. An example of an ASM chart is shown in Figure 11.1. It represents a circuit that counts the number of bits set to 1 in an n-bit input A ($A = a_{n-1}a_{n-2}...a_1a_0$). The rectangular boxes in this diagram represent the states of the digital system, and actions specified inside of a state box occur on each active clock edge in this state. Transitions between states are specified by arrows. The diamonds in the ASM chart represent conditional tests, and the ovals represent actions taken only if the corresponding conditions are either true (on an arrow labeled 1) or false (on an arrow labeled 0).

In this ASM chart, state S1 is the initial state. In this state the result is initialized to 0, and data is loaded into a register A, until a start signal, s, is asserted. The ASM chart then transitions to state S2, where it increments the result to count the number of 1's in register A. Since state S2 specifies a shifting operation, then A should be implemented as a shift register. Also, since the result is incremented, then this variable should be implemented as a counter. When register A contains 0 the ASM chart transitions to state S3, where it sets an output Done = 1 and waits for the signal s to be deasserted.

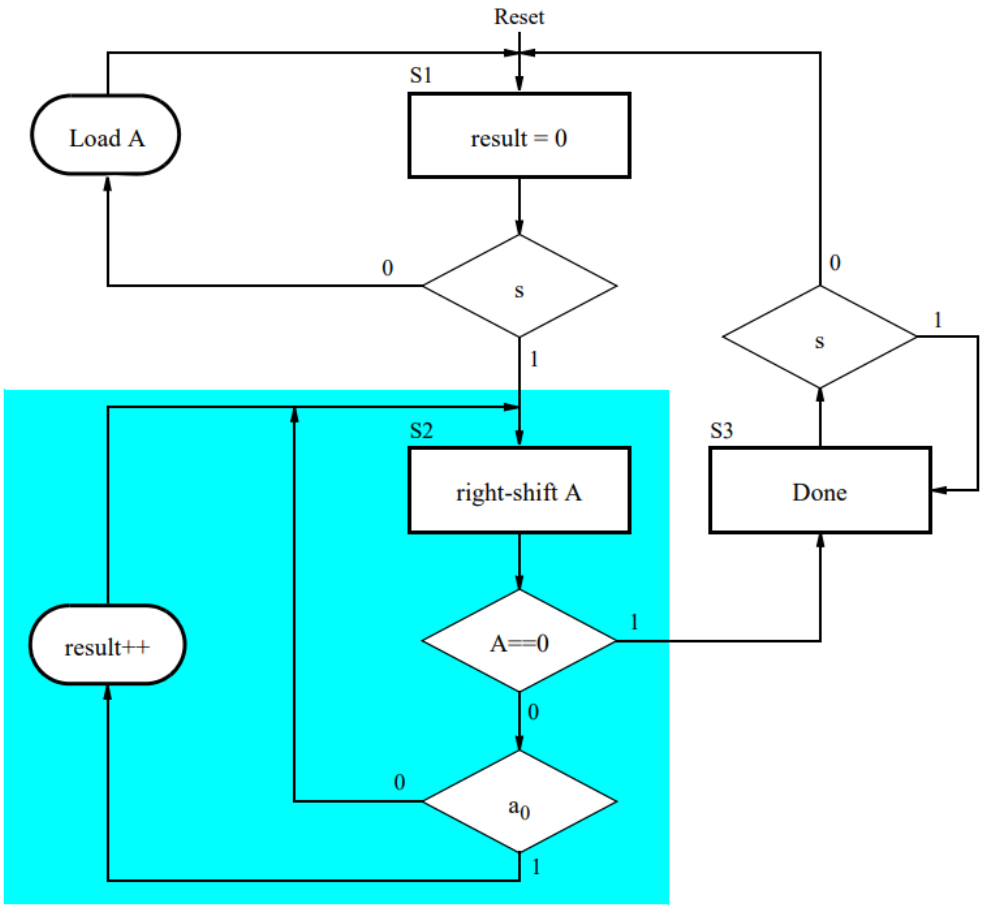


Figure 11.1: LAB 11: Hint for part I (ASM chart for bits counting)

11.2 Part I

REQUIREMENT

1. Write Verilog code to implement the bit-counting circuit using the ASM chart shown in Figure 1 on a DE-series board. Include in your Verilog code the datapath components needed, and make an FSM for the control circuit.
2. The inputs to your circuit should consist of an 8-bit input connected to slide switches W_{7-0}, a synchronous reset connected to KEY_0, and a start signal (s) connected to switch SW_9. Use the 50 MHz clock signal provided on the board as the clock input for your circuit. Be sure to synchronize the s signal to the clock.
3. Display the number of 1s counted in the input data on the 7-segment display HEX_0, and signal that the algorithm is finished by lighting up $LEDR_9$.

SOLUTION

In this lab, we were introduced to the **datapath** definition. A finite State Machine is used to control what is plane to be done in its state, another component in the design is the **datapath**.

Based on the idea of the datapath of part 1, we constructed the datapath as follows.

```
//Datapath
always @(posedge flag) begin
        case (status)
            INIT:    if (S==1'b1)   status <= BEGIN;
            BEGIN:   if (a==0)      status <= END;
            END:     if (S==1'b0)   status <= INIT;
            default: status <= INIT;
        endcase
end

```

The FSM for this part

```
//FSM Control
always @(posedge flag) begin
    case (status)
        INIT: begin
            count    <= 0;
            DONE     <= 0;
            a        <= DATA_IN;
        end
        BEGIN: begin
            if (a[0]==1'b1) count <= count + 1;
```

```
            a       <= a>>1;
        end
        END: begin
            DONE    <= 1;
        end
        default: status <= INIT;
    endcase
end

```

VERIFICATION

As we can see, our DIN is 00000011, which contains four 2s, and the result is exactly 2 at $t = 65ns$ (7'b0100100 is the signal representing 2).

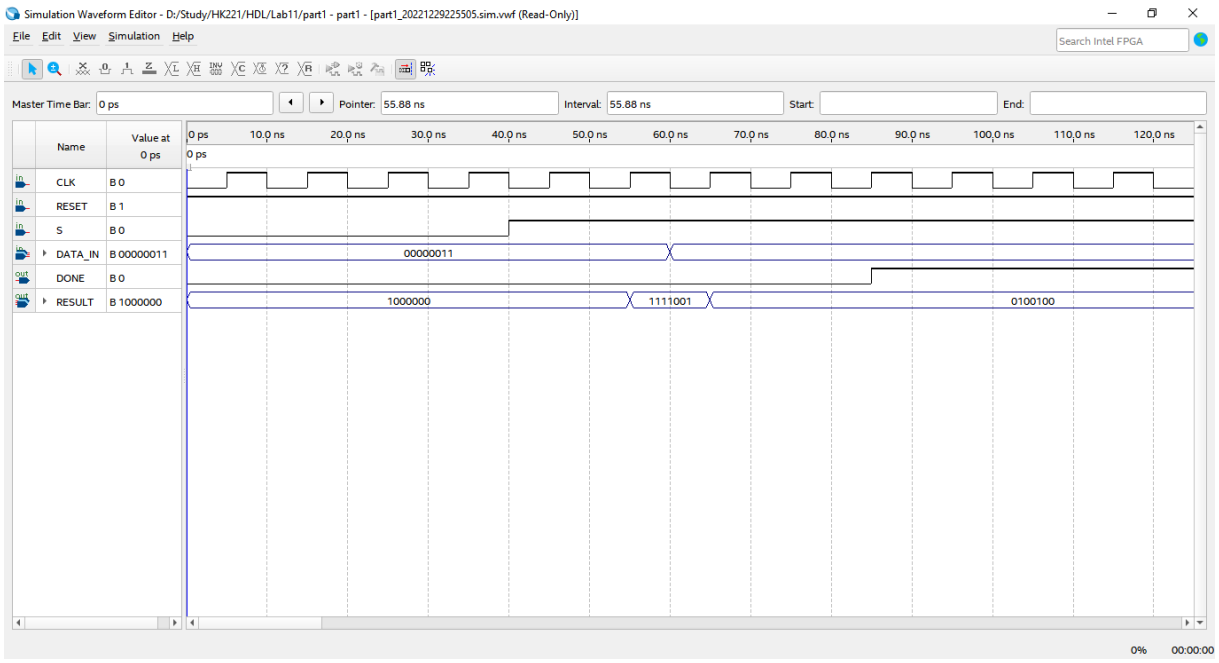


Figure 11.2: LAB 11: Simulation result for part I

11.3 Part II

REQUIREMENT

1. We wish to implement a binary search algorithm, which searches through an array to locate an 8-bit value *A* specified via switches

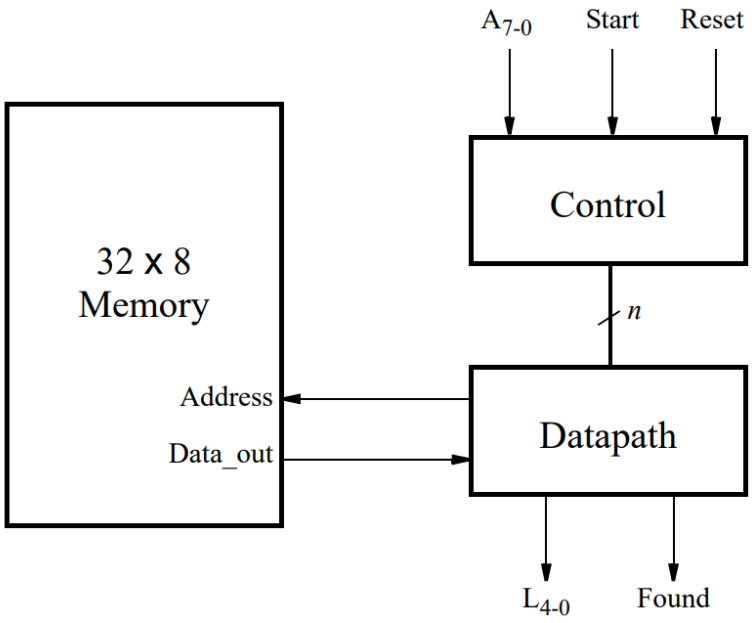


Figure 11.3: LAB 11: Hint for part II (Diagram)

2. The binary search algorithm works on a sorted array. Rather than comparing each value in the array to the one being sought, we first look at the middle element and compare the sought value to the middle element. If the middle element has a greater value, then we know that the element we seek must be in the first half of the array. Otherwise, the value we seek must be in the other half of the array. By applying this approach recursively, we can locate the sought element in only a few steps. The algorithm for a binary search goes like this:

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        return -1

```

3. In this circuit, the array is stored in a memory module that is implemented inside the FPGA chip. A diagram of the memory module that we need to create is depicted in Figure 11.5. This memory module has one read port and one write port, and is called a synchronous random-access memory (synchronous RAM). Note that the memory module includes registers for synchronously loading addresses, input data, and the Write input. These registers are required due to the design of the memory resources in the Intel FPGA chip.

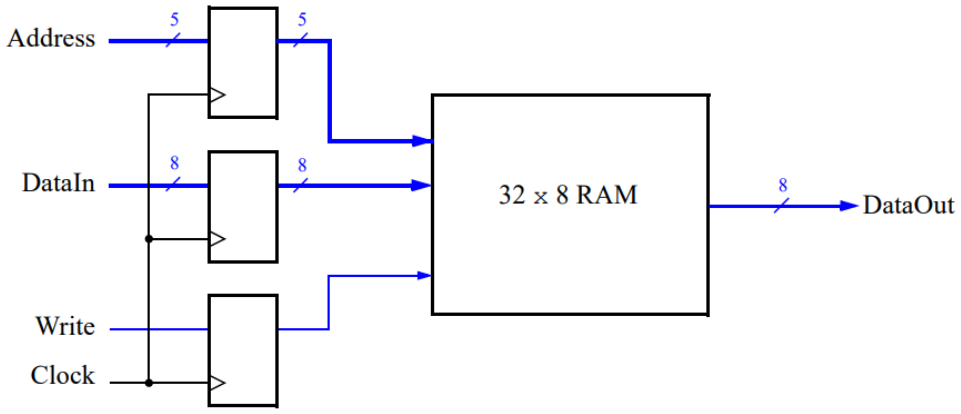


Figure 11.4: LAB 11: RAM IP introduction

SOLUTION

To place data into the memory, we need to specify initial values that should be stored in the memory once our circuit has been programmed into the FPGA chip. This can be done by initializing the memory using the contents of a memory initialization file (MIF). We have specified a file named my_array.mif, which then has to be created in the folder that contains the Quartus project. The memory initialization file is given in below. We set the contents of our MIF file such that it contains a sorted collection of integers.

```
WIDTH=8;
DEPTH=32;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
    00  :   01;
    01  :   02;
    02  :   03;
    03  :   05;
    [04..05]  :   06;
    06  :   07;
    [07..08]  :   08;
```

```
    09  :   0A;
    [0A..0B]  :   0B;
    0C  :   10;
    0D  :   11;
    [0E..0F]  :   12;
    10  :   13;
    11  :   14;
    12  :   15;
    13  :   17;
    14  :   18;
    [15..16]  :   19;
    17  :   1A;
    18  :   1B;
    19  :   1C;
    1A  :   1D;
    [1B..1C]  :   1E;
    [1D..1E]  :   1F;
    1F  :   20;
END;

```

After creating the MIF file, we started to code the actual Binary Search module.

Based on the idea of the datapath of part 1, we constructed the datapath as follows.

```
//Datapath
always @(posedge flag) begin
    case (status)
        INIT:     if (START==1)     status <= BEGIN;
        BEGIN: begin
                    if (done==1)       status <= END;
                    else               status <= WAIT;
                end
        WAIT:                        status <= FIND;
        FIND:                        status <= BEGIN;
        END:      if (START==0)      status <= INIT;
    endcase
end

```

About the FSM, we have 5 states. In the beginning, we just design with 4 states:

* **INIT**: initialize some value and load DATA_IN into a reg.
* **BEGIN**: calculate the address of the value needed to compare with the DATA_IN.

* **FIND**: execute the comparison
* **DONE**: our system change to this state if found the address of DATA_IN inside the memory, or even if it couldn't be found after looked the whole memory.

After try this state machine, we noticed that because every work is executed only in that state, so after changing the state, it has to wait for another clock to execute the job of the new state. So if we use the FSM above, the memory couldn't have enough time to load the value of the address we need.

We decided to add another state, named WAIT, to wait for the memory to load the value of the memory.

```
//FSM Control
always @(posedge flag) begin
    case (status)
        INIT: begin
            data_temp   <= DATA_IN;
            left        <= 5'b00000;
            right       <= 5'b11111;
            address_out <= 5'b00000;

            done        <= 0;
            FOUND       <= 0;
        end
        BEGIN: begin
            address_out <= (left+right)/2;
        end
        FIND: begin
            if (data_temp > value_out)       left  <=
    address_out+1;
            else if (data_temp < value_out)  right <=
    address_out-1;
            else if (data_temp == value_out)begin
                done <= 1;
                FOUND <= 1;
            end
            else if (left >= right) begin
                done    <= 1;
                FOUND <= 0;
            end
        end

    endcase
end

```