Bryan Ching
November 10, 2016
ECPE 174

Lab 7 Report

Problem Summary and System Specifications:

For this lab, I built upon my arithmetic logic unit (ALU) by implementing a "first-in, first-out" (FIFO) memory unit and a reset module. The ALU is expected to add and subtract 6-bit values the same way it did in lab 4. For the purposes of lab 7, I have added four logical operation modules into my design. The FIFO, which incorporates similar modules from lab 6, allows me to store operations into memory and execute these operations all at once. The reset function allows me to clear all operations from my system memory.

The following is a list of system design specifications:

1. Inputs:
    a. 2 6-bit two's complement numbers
    b. 1 3-bit operations number
        i. Add = 000
        ii. Subtract = 001
        iii. Equal = 100
        iv. Greater than = 101
        v. Less than = 110
        vi. A equal 0 = 111
    c. 1 mode switch to determine read or write state
    d. 1 push-button to read from or write to memory
    e. 1 reset switch
2. Outputs:
    a. 6 7-segment displays
        i. 2 for <u>each</u> of the 6-bit two's complement numbers
        ii. 2 for the result of the operation
    b. 6 LEDs
        i. 1 for indicating negative values for <u>each</u> of the 6-bit two's complement numbers
        ii. 1 for indicating a negative value for the result of the operation
        iii. 1 for indicating overflow of the result of the operation
        iv. 1 for indicating FIFO is full
        v. 1 for indicating FIFO is empty
3. 2 FSMs:

a. 1 for determining state of logical operation
b. 1 for determining read or write state

4. FIFO to store up to 8 operations at one time
5. Reset that uses asynchronous assert and synchronous deassert

System Design:

Essentially, this lab merges the ALU design of lab 4 with the memory feature of lab 6. In addition to being able to add and subtract, the ALU is capable of a series of comparison operations. These operations will output a "1" to the 7-segment LEDs if the calculation is true and "0" if the calculation is false. The *equal* operation compares the two input values, denoted as A and B, and determines if they are the same values. The *greater than* operation compares A and B and determines whether or not A is greater than B, while the *less than* operation determines if A is less than B. Lastly, the *zero* operation determines if A is equal to 0.

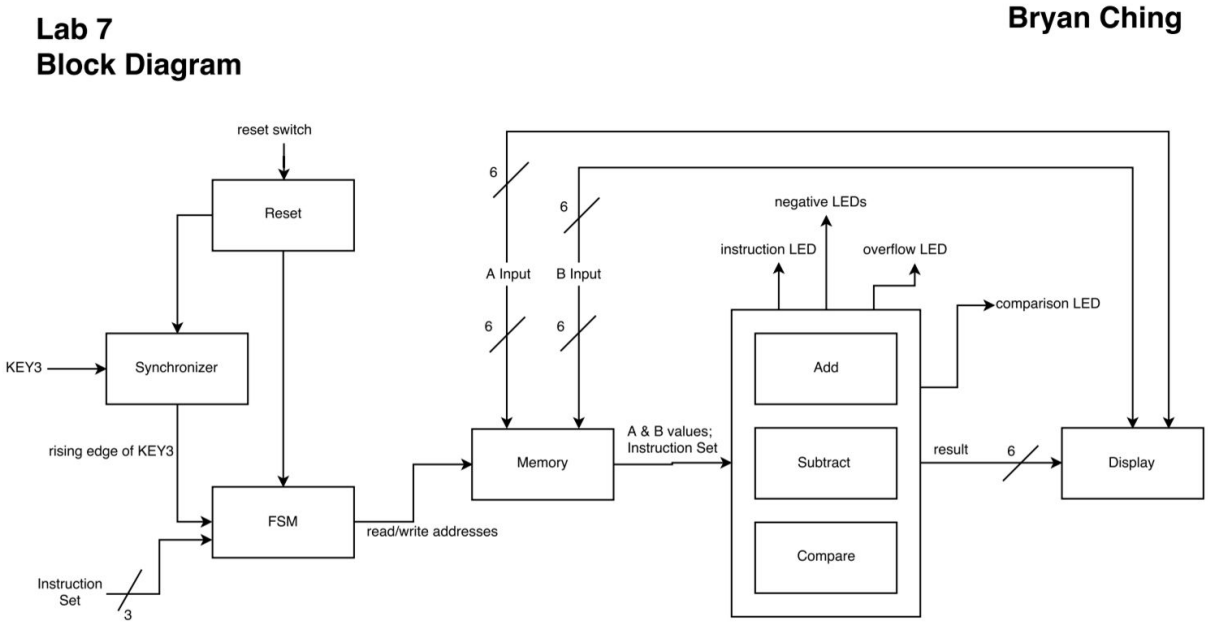

Figure 1: Lab 7 Block Diagram

The block diagram in Figure 1 depicts the initial layout of this lab, which incorporates a memory feature with the ALU. I operate this system by setting the input values, A and B, to whatever 6-bit values I want to calculate. Secondly, I select which 3-bit operation I want to perform. Next, I flip the input switch for writing to memory and press the push-button to perform the operation. If I wish to read the values out of memory and calculate the result, then I flip the switch to read mode and press the push-button again. The corresponding result along with the two input values

display on the 7-segment LEDs. Similar to lab 4, this lab will have LEDs that signify if the values are negative. I added the reset feature at the end of my design in order to revert the entire system back to its initial state, which emptied all values stored in memory.

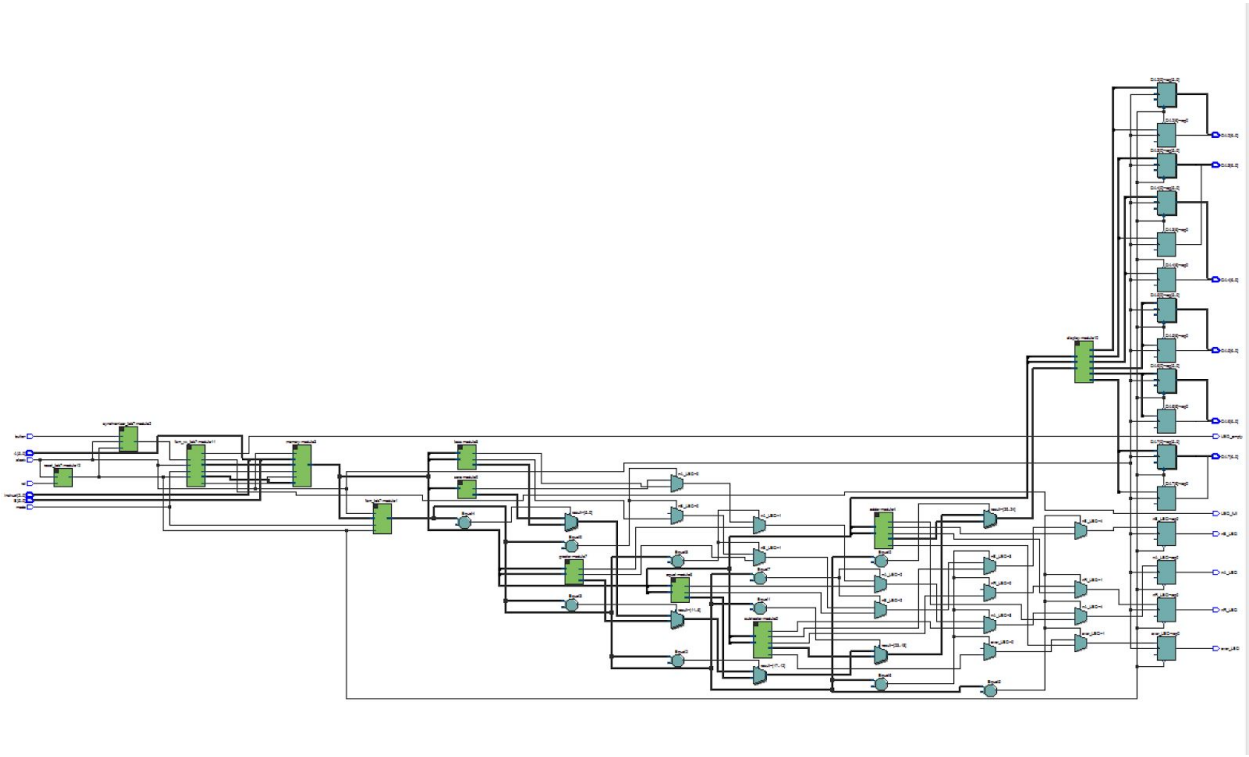

Figure 2: RTL Viewer

Figure 2 depicts the RTL viewer for the entire lab 7. There were a couple design changes between the initial conception and the final design. One of these changes is the inclusion of a second Finite State Machine (FSM). Originally, I had one FSM that determined which operation the system will perform. I neglected to include a FSM that determines whether the system is in read or write mode. Secondly, I had to connect my operations FSM differently than how I did in the initial design. Initially, the operations FSM fed into the memory module. I realized that I needed to select either the read or the write mode first, which meant feeding the output of the read/write FSM into the memory module. When I pushed the push-button, the values are read out of memory and feeds into the operations FSM. At this point, the operations FSM performs the proper calculations.

```
module fsm_lab7 (input logic clock,
                 input logic mode, rstsync,
                 input logic [14:0]dataOut,
                 output logic [2:0]out_instruct);

logic [2:0]instruct;

typedef enum logic [2:0]{Add, Subtract, Equal, Greater, Less, A_0} statetype;

statetype state, nextstate;

always_ff @(posedge clock or negedge rstsync)
   begin
      if (!rstsync)
         state <= Add;
      else
         state <= nextstate;
   end

assign instruct = dataOut[2:0];

always_comb
begin
   case(state)
      Add:
         if (instruct==3'b001 && mode==1'b0)
            nextstate = Subtract;
         else if (instruct==3'b100 && mode==1'b0)
            nextstate = Equal;
         else if (instruct==3'b101 && mode==1'b0)
            nextstate = Greater;
         else if (instruct==3'b110 && mode==1'b0)
            nextstate = Less;
         else if (instruct==3'b111 && mode==1'b0)
            nextstate = A_0;
```

Figure 3: Operations FSM Code Snippet

A snippet of code for the operations FSM can be found in Figure 3. The reset module has been implemented into its design. Additionally, each condition statement in the *Add* state as well as the other five states has a *mode* variable. This variable determines whether the system is in read or write mode. “0” denotes the read mode, while “1” denotes the write mode. None of these operations will occur unless the system is reading values from memory. Lastly, this FSM outputs the operation instruction that I input into memory. The top-level module uses this output.

The Read-Write FSM can be found in Appendix A under Figure A-1. I have not edited much in that FSM other than adding the reset functionality. Similarly, the synchronizer code snippet can be found in Appendix A under Figure A-2. The synchronizer does not differ much from my other synchronizer designs from pass labs. The push-button is the only component synchronized with three flip-flops in the whole lab.

```
module reset_lab7(input logic clock, rst,
                  output logic rstsync);

    logic intRst;
    always_ff@(posedge clock or negedge rst)
       if (!rst)
          begin
             intRst <= 1'b0;
             rstsync <= 1'b0;
          end
       else
          begin
             intRst <= 1'b1;
             rstsync <= intRst;
          end
 endmodule
```

Figure 4: Reset Module Code Snippet

```
module memory(input logic clock, ren, wen,
                  input logic [14:0] dataIn,
                  input logic [2:0] wrAddr, rdAddr,
                  output logic [14:0] dataOut);

    logic [14:0] memory [7:0];

    initial
    begin
       for (int i=15'b0; i<15'd8; i++)
           memory[i] <= i;
    end

    always_ff @(posedge clock)
       begin
       if(wen)
          memory[wrAddr] <= dataIn;

       if(ren)
          dataOut <= memory[rdAddr];
       end
endmodule

```

Figure 5: Memory Module Code Snippet

This reset module is the exact same design as the one used in lab 6 for the memory functionality. This reset uses an asynchronous assert and a synchronous deassert. The asynchronous assert handles the issue of the reset assertion time window. The reset needs to be asserted for multiple clock cycles. The synchronous deassert handles the issue in which the reset deasserts within a recovery time window. The memory module, as depicted in Figure 5, is the only module that uses a flip-flop but does not implement the reset module. This is because Quartus looks for specific code structures to infer as memory. As a result, the reset module cannot be used in the memory module.

```
module greater (input logic [5:0]A,
                input logic [5:0]B,
                output logic [5:0]result,
                output logic nA_LED, nB_LED);

    logic [5:0]new_A;
    logic [5:0]new_B;

    always_comb
    begin
       if (A[5] == 1'b1)
          new_A = ~A+1'b1;
       else
          new_A = A;
    end

    always_comb
    begin
       if (B[5] == 1'b1)
          new_B = ~B+1'b1;
       else
          new_B = B;
    end

    always_comb
    begin
       if (A[5] == B[5] && A[5] == 1'b0)
          begin
             if (new_A > new_B)
                result = 6'd1;
             else
                result = 6'd0;
          end
       else if (A[5] == B[5] && A[5] == 1'b1)
          begin
             if (new_A > new_B)
```

Figure 6: Greater Than Module Code Snippet

In the *greater than* and *less than* modules, I first calculate the two's complement of both the input values, A and B. My combinational logic involves checking the most significant bit and checking if they are equal to "0" or "1". By doing so, I can confirm whether one or the other is positive or negative. If they are both either positive or negative, then I can do further comparisons. However, if they differ in value, then it is much simpler to determine which input value is greater or less than the other input value. In these two modules, I also determine whether the values are negative and set the negative LED values accordingly. The *less than* module is similar in design with slight difference. As a result, I have placed the screenshot of the *less than* module in Appendix A under Figure A-3.

```
module equal (input logic [5:0]A, B,
              output logic [5:0] result,
              output logic nA_LED, nB_LED);

 always_comb
begin
   if (A == B)
      result = 6'd1; // true
   else
      result = 6'd0; // false
end
```

Figure 7: Equal Module Code Snippet

```
module zero ( input logic [5:0]A,
              output logic [5:0]result,
              output logic nA_LED);

  always_comb
  begin
     if (A == 6'd0)
        result = 6'd1;
     else
        result = 6'd0;
  end
```

Figure 8: Zero Module Code Snippet

The *equal* and *zero* modules are very simple in design. The *equal* module checks if A and B are equal. The *zero* module checks if A is equal to "0". There are no additional code to determine the two's complement. These two modules also calculate whether or not their respective negative LEDs need to be turned on, which is later used in the top-level module.

The *adder* and *subtractor* modules are direct copies of the same modules used in lab 4. As such, screenshots of these modules have been included Their calculations are inherently simple. The *adder* adds the two input values. The *subtractor* needs to take the two's complement of the second input, B. The module then adds the input, A, and the two's complement of input B together. Additional calculations in both modules involve calculating whether or not there is overflow and if any of the inputs or output is negative.

```
module display ( input logic [5:0]A,
                 input logic [5:0]B,
                 input logic [5:0]result,
                 output logic [6:0]DA7, // A
                 output logic [6:0]DA6,
                 output logic [6:0]DA5, // B
                 output logic [6:0]DA4,
                 output logic [6:0]DA3, // Result
                 output logic [6:0]DA2);

  // For Result
  logic [5:0]new_result;
  logic [3:0]digit_1; // first digit (tens)
  logic [3:0]digit_2; // second digit (ones)
  logic [6:0]display_1; // 7 segment display
  logic [6:0]display_2;
```

Figure 9: Display Module Code Snippet

```
module decimal (input logic [3:0]digit,
                output logic [6:0]display);

 always_comb
begin
    if (digit == 4'd0)
       display = 7'b1000000; // 0
    else if (digit == 4'd1)
       display = 7'b1111001; // 1
    else if (digit == 4'd2)
       display = 7'b0100100; // 2
    else if (digit == 4'd3)
       display = 7'b0110000; // 3
    else if (digit == 4'd4)
       display = 7'b0011001; // 4
    else if (digit == 4'd5)
       display = 7'b0010010; // 5
    else if (digit == 4'd6)
       display = 7'b0000010; // 6
    else if (digit == 4'd7)
       display = 7'b1111000; // 7
    else if (digit == 4'd8)
       display = 7'b0000000; // 8
    else if (digit == 4'd9)
       display = 7'b0010000; // 9
    else
       display = 7'b1111111; // nothing
end

 endmodule

```

Figure 10: Decimal Module Code Snippet

Figure 9 and Figure 10 depict the SystemVerilog code snippets for displaying values on the 7-segment displays. I used six 7-segment displays in my overall design. Each input values and the calculation output value receive two 7-segment displays. The *display* module calculates the tens and the ones digit of the 6-bit inputs and result. Then, it sends those tens and ones digits to the *decimal* module that matches the digits to its corresponding 7-segment combination. The *decimal* module sends the 7-segment combination back to the *display* module to output.

```
    else
       begin
          nA_LED <= nA_LED_Zero;
          nB_LED <= 1'b0;
          nR_LED <= 1'b0;
          over_LED <= 1'b0;
          DA7 <= display7;
          DA6 <= display6;
          DA5 <= display5;
          DA4 <= display4;
          DA3 <= display3;
          DA2 <= display2;
       end
    end

    assign LED_full = full;
    assign LED_empty = empty;
    assign in_A = dataOut[14:9];
    assign in_B = dataOut[8:3];

    fsm_lab7 module1 (.clock(clock), .mode(mode), .rstsync(rstsync), .dataOut(dataOut), .out_instruct(out_instr

    fsm_rw_lab7 module11 (.clock(clock), .mode(mode), .button(rise), .rstsync(rstsync), .ren(ren), .wen(wen),
                          .empty(empty), .full(full), .r_addr(r_addr), .w_addr(w_addr));

    synchronizer_lab7 module2 (.clock(clock), .button(button), .rstsync(rstsync), .rise(rise));

    memory module3 (.clock(clock), .ren(ren), .wen(wen), .dataIn({A, B, instruct}), .wrAddr(w_addr),
                    .rdAddr(r_addr), .dataOut(dataOut));
```

Figure 11: Top-Level Module Code Snippet

Finally, a snippet of code for the top-level module is shown in Figure 11. The top-level combines every one of the aforementioned modules and creates a unified system. Additional calculations that I inputted for the top-level module includes determining the result variable, 7-segment displays, and LEDs based on which operation instruction I select. I determine the result variable value inside an “always_comb” block, while I determine the LEDs and 7-segment displays inside an “always_ff” block. Also, this “always_ff” block implements the reset functionality.

Testing Approach:

For testing my lab design, I created testbenches for the newer comparison modules, the operations FSM, and the top-level module. The comparison modules that I tested include the *equal*, *greater than*, *less than*, and *zero* modules. For the comparison modules, I designed the testbenches to test basic functionality of each module. I achieved this by testing as many different types of inputs combinations as possible, which includes negative values to ensure the LEDs work as intended as well. With all these different input combinations, I ensure that the modules perform their intended purpose as stated in the system design.

```
module greater_testbench();

        // DUT signals
        logic clock = 1'b0;
        logic [5:0]A;
        logic [5:0]B;
        logic result;

        // Connect device to check
        greater dutlab7(.A(A), .B(B), .result(result));

        // Generate clock
        always #50 clock <= ~clock;

        // Generate inputs

        // Create task to loop through all possible combinations
        task testCase(input logic [5:0]in_A, input logic [5:0]in_B, input logic result_test);
                @(negedge clock);
                A <= in_A;
                B <= in_B;
                @(posedge clock);
                check: assert(result==result_test) $display("Output correct!");

        endtask

        initial
                begin
                        testCase(-6'd5, -6'd7, 1'b1); // -5 > -7
                        testCase(-6'd10, 6'd5, 1'b0); // -10 < 5
                        testCase(6'd10, 6'd2, 1'b1); // 10 > 2
                        testCase(6'd15, 6'd25, 1'b0); // 15 < 25
                        @(negedge clock) #1;
                        $stop;
                end
endmodule
```

Figure 12: Greater Than Module Testbench Code

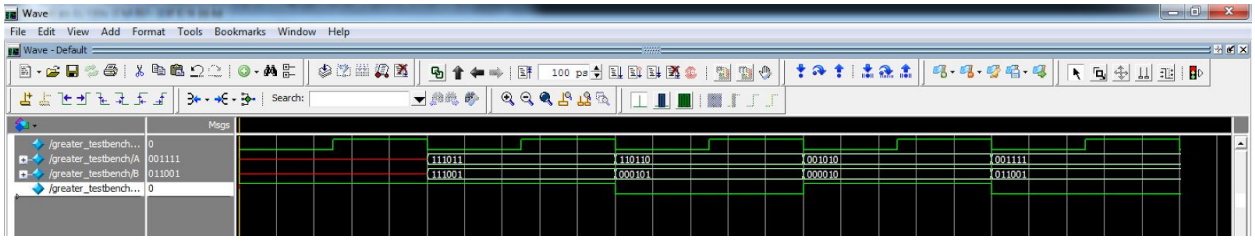

Figure 13: Greater Than Module Waveform Diagram

Figure 12 depicts an example of one of my testbenches, which is for the *greater than* module. Figure 13 depicts the waveform diagram of the *greater than* module. I have tested four cases with varying inputs and results. Figure 13 shows the correct results in that it wave goes high and low when it should. The other modules, such as *less than*, *equal*, and *zero*, are similar in design. As such, I have placed screenshots of them in Appendix B along with their respective waveform diagrams.

```
1   module fsm_testbench();
2
3           // DUT signals
4           logic clock = 1'b0;
5           logic [2:0]instruct;
6           logic [2:0]out_instruct;
7
8           // Connect device to check
9           fsm_lab7 dutlab7(.clock(clock), .instruct(instruct), .out_instruct(out_instruct));
10
11          // Generate clock
12          always #50 clock <= ~clock;
13
14          // Generate inputs
15
16          // Create task to loop through all possible combinations
17          task testCase(input logic [2:0]instruct_test, input logic [2:0]out_instruct_test);
18                  @(negedge clock);
19                  instruct <= instruct_test;
20                  @(posedge clock);
21                  check: assert(instruct_test == out_instruct_test) $display("Output correct!");
22
23          endtask
24
25          initial
26                  begin
27                          testCase(3'b000, 3'b000);
28                          testCase(3'b001, 3'b001);
29                          testCase(3'b100, 3'b100);
30                          testCase(3'b101, 3'b101);
31                          testCase(3'b110, 3'b110);
32                          testCase(3'b111, 3'b111);
33                          @(negedge clock) #1;
34                          $stop;
35                  end
36  endmodule
```

Figure 14: Operations FSM Testbench Code

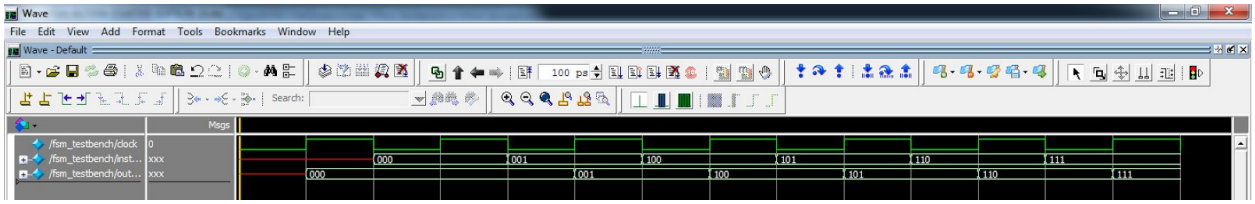

Figure 15: Operations FSM Waveform Diagram

Figure 14 depicts the testbench code that I wrote for my operations FSM. Essentially, I needed to ensure that the instructions that get inputted into the FSM is the same as the output. I tested each of the six different operations and got the expected results from the waveform diagram as shown in Figure 15.

```
8         logic mode, rst;
9         logic nA_LED, nB_LED, nR_LED, over_LED, LED_full, LED_empty;
10        logic [6:0] DA7, DA6, DA5, DA4, DA3, DA2;
11
12        // Connect device to check
13        toplevel_lab7 dutlab7(.clock(clock), .instruct(instruct), .A(A), .B(B), .button(button), .mode(mode), .rst(rst), .nA_LED(nA_LED), .nB_LED(nB_LED),
14                              .nR_LED(nR_LED), .over_LED(over_LED), .LED_full(LED_full), .LED_empty(LED_empty), .DA7(DA7), .DA6(DA6), .DA5(DA5), .DA4(DA4)
15
16        // Generate clock
17        always #50 clock <= ~clock;
18
19        // Generate inputs
20
21        // Create task to loop through all possible combinations
22        task Write(input logic [2:0]instruct_test, input logic [5:0]in_A, input logic [5:0]in_B, input logic mode_test, input logic rst_test);
23                @(negedge clock);
24                mode <= mode_test;
25                instruct <= instruct_test;
26                A <= in_A;
27                B <= in_B;
28                rst <= rst_test;
29                repeat(3)@(negedge clock);
30                button <= 1'b1;
31                repeat(3)@(negedge clock);
32                button <= 1'b0;
33                repeat(5)@(negedge clock);
34                //@(posedge clock);
35                //check: assert() $display("Output correct!");
36
37        endtask
```

Figure 16: Top-Level Module Testbench Code

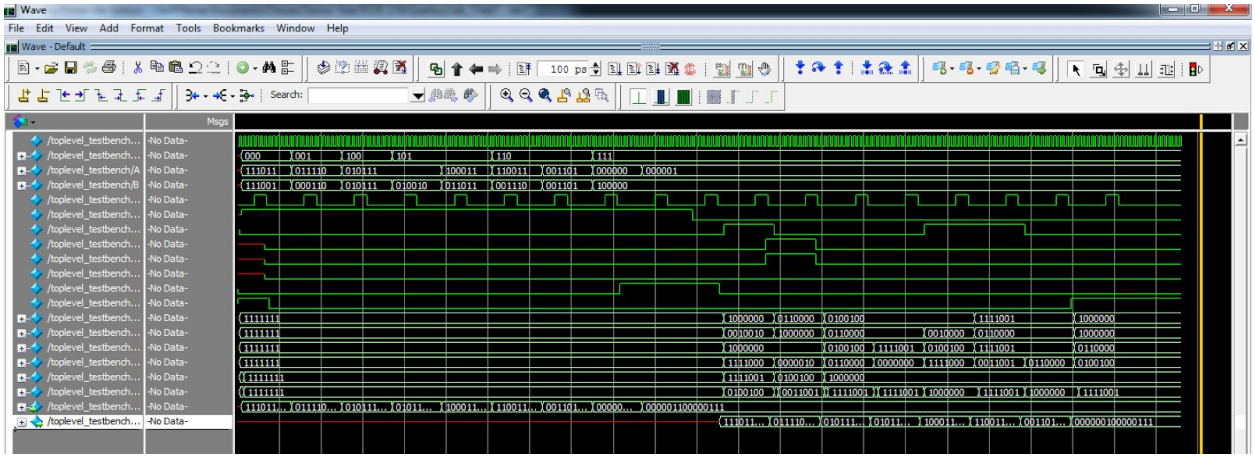

Figure 17: Top-Level Module Waveform Diagram

The top-level testbench is the most important testbench of this lab because it tests the functionality of every component and module. Initially, I created a testbench in which I manually inputted the values I wanted to test. By doing so, I was able to manually check the values myself for accuracy. I tested up to eight different values. I then checked each values corresponding output to make sure the correct LEDs went high and the operation modules correctly calculated

the results. After I proved that my static testbench was fully operational, I created a new testbench that used randomization.

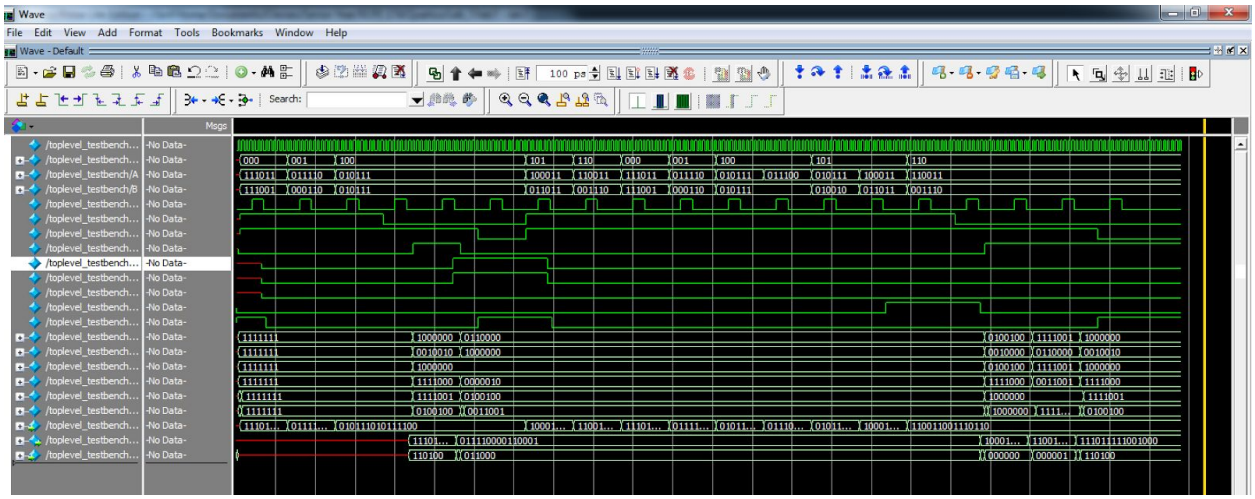

Figure 18: Top-Level Module with Randomization Waveform Diagram

Figure 18 includes the randomization in my new top-level testbench. I had a *for* loop that looped through my testbench for a random amount of time. I set the lower limit for the *for* loop to "10" and the upper limit to "300". As for the inputs, I randomized them all except the button presses. The button presses needed to remain pressed for a certain amount of clock cycles. There is also not a lot of negative values that get tested throughout this testbench. This is probably due to the excess amount of randomization between read and write modes and the input values. Overall, the randomized testbench worked correctly. A snippet of the testbench code can be found in Appendix B under Figure B-7.

Results:

For the final design of this lab, I implemented a reset module. The reason I did not include the reset module initially in my FSMs and top-level modules was because I wanted to make sure those modules worked fine without the reset. I added the reset to every module that has a flip-flop except the memory module.

When I programmed the Cyclone Board, nearly every component worked. The 7-segment and LEDs displayed properly. The different operations all calculated the correct values. The input switches all worked and could be stored in memory. However, there was a slight problem with my with my push-button and delays. I had the issue in which I had to press the push-button twice in order for the result to display on the 7-segment display. I had to debug for a portion of the lab as to what could be wrong.

```
else if (out_instruct == 3'b000 and rise == 1'b1)
   begin
      nA_LED <= nA_LED_Add;
      nB_LED <= nB_LED_Add;
      nR_LED <= nR_LED_Add;
      over_LED <= over_LED_Add;
      DA7 <= display7;
      DA6 <= display6;
      DA5 <= display5;
      DA4 <= display4;
      DA3 <= display3;
      DA2 <= display2;
   end
```

Figure 19: Top-Level Design Error

Figure 19 depicts what caused the delay problem in my design. For each condition state, I added the rising edge of the push-button. The problem with adding the rising edge in the top-level is that I already had added it into my Read-Write FSM. By including the rising edge in both modules, I had created a race condition in which some of the values got displayed on the 7-segment while others did not. After removing the rising edge button push in the top-level, my output displayed on the 7-segment displays right after I pressed the push-button.

Analysis:

For my testing procedure, I created two different testbenches for my top-level design. I created a static testbench for directed testing and a randomized testbench to see the results of my overall design. The randomized testbench is good for scalability of testbench for a large project. However, in terms of debugging and analyzing if the design outputs correctly, a directed testbench works better. By having a directed testbench, I can easily target functions that I think are not operating correctly and fix them accordingly.

Unit	Setup Time (ns)	Hold Time (ns)	Maximum Clock Rate	Total Logic Elements	Total Register
Adder	-1.990	3.301	N/A	7	0
Subtractor	-2.992	3.404	N/A	13	0
Operations FSM	6.347	0.672	2.25*10^8	14	5
Top-Level	-5.739	0.184	2.26*10^8	510	198

Figure 20: Timing Analyzer Data

I have obtained the data on the adder and subtractor through the timing analyzer. From my analysis, the subtractor takes more time and uses more space overall. The subtractor has a higher setup time and hold time, which means it takes longer to before data gets inputted into my system and calculations start. Because the adder and subtractor do not have flip-flops, they do not have any clock rates to report. The subtractor takes up almost twice as much space, or logical elements, than the adder. The reason that the subtractor takes more space and time is because it performs a two's complement calculation on the input variable, B before doing any calculations. The adder simply adds the two input values.

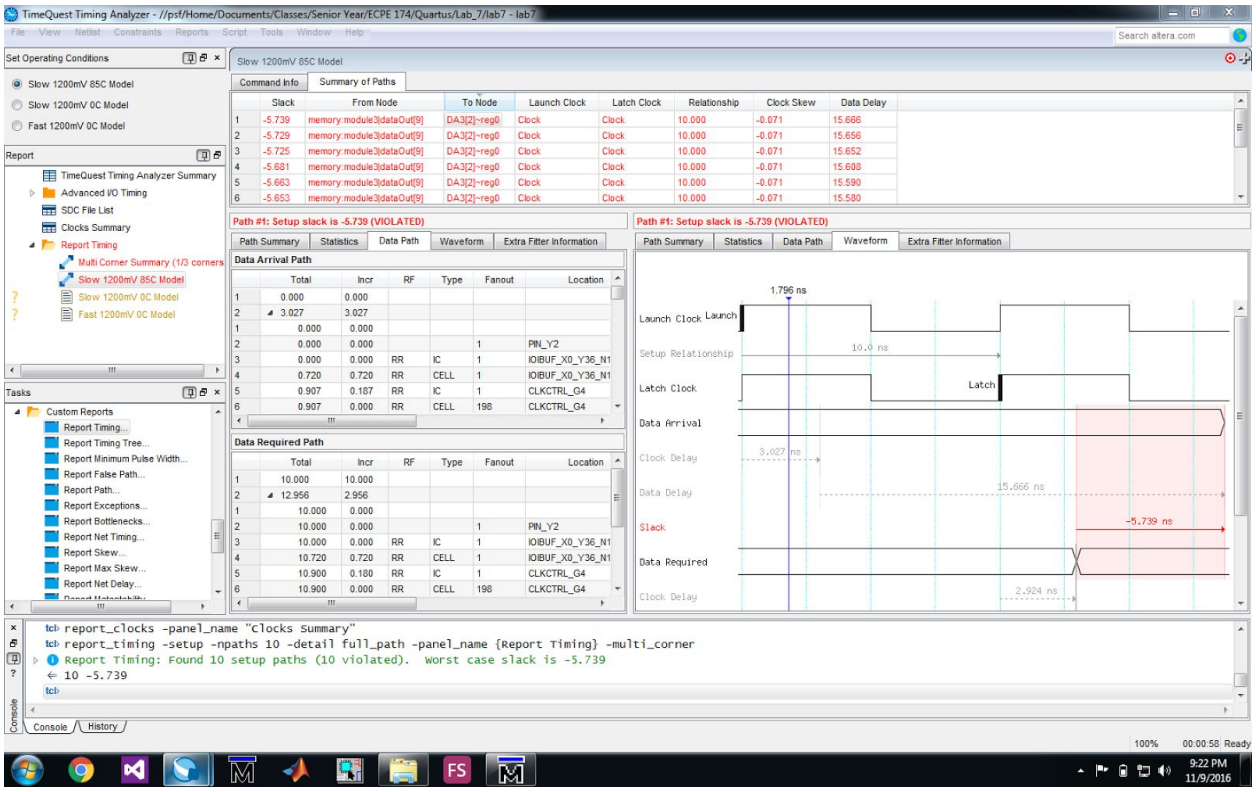

Figure 21: Timing Analyzer

Overall, my system design worked correctly and had been verified on the Cyclone Board. I used the timing analyzer to obtain data on the adder, subtractor, operations FSM, and top-level design. All the data can be found in Figure 20. The maximum clock frequency is taking the minimum pulse width and inverting it. So the equation is 1/(minimum pulse width). My longest path is from my memory module (memory:module3|dataOut[9]) to my display (DA3[2]~reg0). The longest propagation delay for my top-level design is 15.739ns The max slack is 5.739ns. I ran my timing analyzer for 10ns. Therefore, I added 10ns to 5.739ns for the longest path. The best way to reduce the propagation delay is to reduce the number of transitions throughout my design. It would involve making my code more efficient and using less flip-flops if they are not necessary.

Appendix A: Additional Design

```
19      always_comb
20         begin
21            case(state)
22               IDLE:
23                  if (mode==1'b0 && button==1'b1 && empty==1'b0)
24                     nextstate = READ;
25                  else if (mode==1'b1 && button==1'b1 && full==1'b0)
26                     nextstate = WRITE;
27                  else
28                     nextstate = IDLE;
29               READ:
30                  if (mode==1'b0 && button==1'b1 && empty==1'b0)
31                     nextstate = READ;
32                  else
33                     nextstate = IDLE;
34               WRITE:
35                  if (mode==1'b1 && button==1'b1 && full==1'b0)
36                     nextstate = WRITE;
37                  else
38                     nextstate = IDLE;
39               default: nextstate = IDLE;
40            endcase
41         end
```

Figure A-1: Read-Write FSM Code Snippet

```
1   module synchronizer_lab7 (input logic clock,
2                             input logic button, rstsync,
3                             output logic rise);
4
5       logic in_ff1, in_ff2, in_ff3; // in_ff2 == in_sync1, in_ff3 == in_sync2
6
7       assign rise = (in_ff2 && !in_ff3);
8
9       always_ff @ (posedge clock or negedge rstsync)
10      begin
11         if (rstsync == 1'b0)
12         begin
13            in_ff1 <= 1'b0;
14            in_ff2 <= 1'b0;
15            in_ff3 <= 1'b0;
16         end
17      else
18         begin
19            in_ff1 <= button;
20            in_ff2 <= in_ff1;
21            in_ff3 <= in_ff2;
22         end
23      end
24
25  endmodule
26
```

Figure A-2: Synchronizer Code Snippet

```
1   module less (input logic [5:0]A,
2                input logic [5:0]B,
3                output logic [5:0]result,
4                output logic nA_LED, nB_LED);
5
6       logic [5:0]new_A;
7       logic [5:0]new_B;
8
9       always_comb
10      begin
11         if (A[5] == 1'b1)
12            new_A = ~A+1'b1;
13         else
14            new_A = A;
15      end
16
17      always_comb
18      begin
19         if (B[5] == 1'b1)
20            new_B = ~B+1'b1;
21         else
22            new_B = B;
23      end
24
25      always_comb
26      begin
27         if (A[5] == B[5] && A[5] == 1'b0)
28            begin
29               if (new_A < new_B)
30                  result = 6'd1;
31               else
32                  result = 6'd0;
33            end
34         else if (A[5] == B[5] && A[5] == 1'b1)
35            begin
36               if (new_A < new_B)
```

Figure A-3: Less Than Module Code Snippet

```
1   module adder ( input logic [5:0]A,
2                  input logic [5:0]B,
3                  output logic [5:0]out_Val,
4                  output logic nA_LED, nB_LED, nR_LED,
5                  output logic over_LED);
6
7    assign out_Val = A + B;
```

Figure A-4: Adder Module Code Snippet

```
1   module subtractor (input logic [5:0]A,
2                      input logic [5:0]B,
3                      output logic [5:0]out_Val,
4                      output logic nA_LED, nB_LED, nR_LED,
5                      output logic over_LED);
6
7    logic [5:0]new_B;
8    assign new_B = (~B+1'b1);
9
10   assign out_Val = A + (~B+1'b1);
11
```

Figure A-5: Subtractor Module Code Snippet

Appendix B: ModelSim Testbenches and Waveform Diagrams

```
1  module less_testbench();
2
3          // DUT signals
4          logic clock = 1'b0;
5          logic [5:0]A;
6          logic [5:0]B;
7          logic result;
8
9          // Connect device to check
10         less dutlab7(.A(A), .B(B), .result(result));
11
12         // Generate clock
13         always #50 clock <= ~clock;
14
15         // Generate inputs
16
17         // Create task to loop through all possible combinations
18         task testCase(input logic [5:0]in_A, input logic [5:0]in_B, input logic result_test);
19                 @(negedge clock);
20                 A <= in_A;
21                 B <= in_B;
22                 @(posedge clock);
23                 check: assert(result==result_test) $display("Output correct!");
24
25         endtask
26
27         initial
28                 begin
29                         testCase(-6'd5, -6'd7, 1'b0); // -5 > -7
30                         testCase(-6'd10, 6'd5, 1'b1); // -10 < 5
31                         testCase(6'd10, 6'd2, 1'b0); // 10 > 2
32                         testCase(6'd15, 6'd25, 1'b1); // 15 < 25
33                         @(negedge clock) #1;
34                         $stop;
35                 end
36 endmodule
```

Figure B-1: Less Than Module Testbench Code

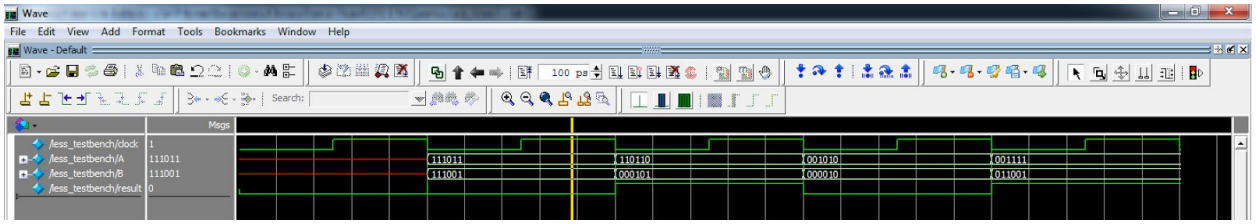

Figure B-2: Less Than Module Waveform Diagram

```
module equal_testbench();

        // DUT signals
        logic clock = 1'b0;
        logic [5:0]A;
        logic [5:0]B;
        logic result;

        // Connect device to check
        equal dutlab7(.A(A), .B(B), .result(result));

        // Generate clock
        always #50 clock <= ~clock;

        // Generate inputs

        // Create task to loop through all possible combinations
        task testCase(input logic [5:0]in_A, input logic [5:0]in_B, input logic result_test);
                @(negedge clock);
                A <= in_A;
                B <= in_B;
                @(posedge clock);
                check: assert(result==result_test) $display("Output correct!");

        endtask

        initial
                begin
                        testCase(-6'd5, -6'd7, 1'b0); // -5 != -7
                        testCase(-6'd10, 6'd5, 1'b0); // -10 != 5
                        testCase(6'd18, 6'd18, 1'b1);
                        testCase(-6'd10, -6'd10, 1'b1);
                        testCase(6'd10, 6'd2, 1'b0); // 10 != 2
                        testCase(6'd15, 6'd25, 1'b0); // 15 != 25
                        @(negedge clock) #1;
                        $stop;
                end
```

Figure B-3: Equal Module Testbench Code

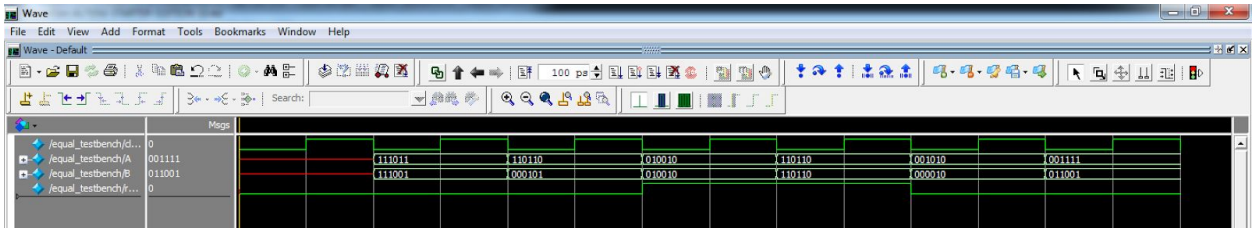

Figure B-4: Equal Module Waveform Diagram

```
module zero_testbench();

        // DUT signals
        logic clock = 1'b0;
        logic [5:0]A;
        logic result;

        // Connect device to check
        zero dutlab7(.A(A), .result(result));

        // Generate clock
        always #50 clock <= ~clock;

        // Generate inputs

        // Create task to loop through all possible combinations
        task testCase(input logic [5:0]in_A, input logic result_test);
                @(negedge clock);
                A <= in_A;
                @(posedge clock);
                check: assert(result==result_test) $display("Output correct!");

        endtask

        initial
                begin
                        testCase(-6'd5, 1'b0); // false
                        testCase(6'd0, 1'b1); // true
                        testCase(-6'd10, 1'b1); // false
                        testCase(6'd0, 1'b0); // true
                        @(negedge clock) #1;
                        $stop;
                end
endmodule
```

Figure B-5: Zero Module Testbench Code

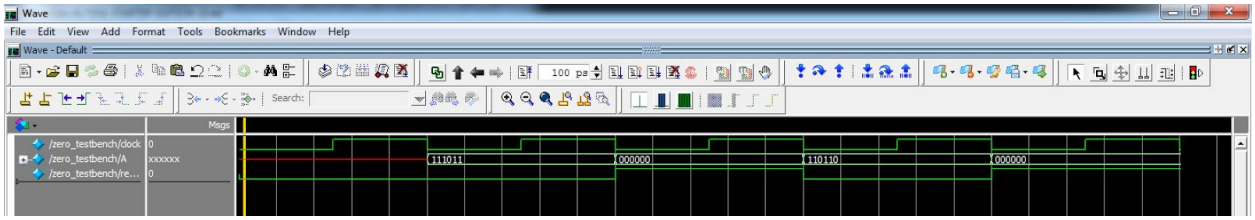

Figure B-6: Zero Module Waveform Diagram

```
43                  endcase*/
44                  instruct <= $urandom_range(7,0);
45                  in_A = $urandom_range(63,0);
46                  in_A = in_A - 6'd32;
47                  in_B = $urandom_range(63,0);
48                  in_B = in_B - 6'd32;
49
50                  //rst <= $urandom_range(1,0);
51                  rst <= 1'b1;
52                  mode <= $urandom_range(1,0);
53
54                  //instruct <= instructVal;
55                  //mode <= mode_test;
56                  A <= in_A;
57                  B <= in_B;
58                  //rst <= rst_test;
59                  repeat(3)@(negedge clock);
60                  button <= 1'b1;
61                  repeat($urandom_range(5,1))@(negedge clock);
62                  button <= 1'b0;
63                  repeat($urandom_range(5,1))@(negedge clock);
64          endtask
65
66          initial
67                  begin
68
69                  rand_loop=$urandom_range(300, 10);
70                  for(int i=0; i<rand_loop; i++)
71                          begin
72                                  TestEverything();
73                          end
74                  repeat(10)@(negedge clock) #1;
75                  $stop;
76
77                  end
78  endmodule
79
```

Figure B-7: Top-Level Module with Randomization Testbench Code