Cynthia Li
EE 371
April 22, 2021
Lab 2 Report

Procedure

The lab comprises of three tasks: 1) Design a 32x4 single port ram that can takes an input 5-bit address and has a four-bit data port, and a write control input, its address, data to write and data to read will be displayed on HEX in hexadecimal form.
2) Design a 32x4 2-port ram using IP catalog in Quartus that use one port supplying 5-bit address for read operation and the other port for 5-bit address for write operation. The read address should be internally updating by a designed counter on an approximately 1Hz frequency, and all addresses, the 4-bit data to read and data to write, should be display on HEX in hexadecimal format; 3) Design a FIFO that uses a 16x8 2 port ram to store data, and display the "queue's" full/empty status on LEDR9/8, and display read data, write data on HEX in hexadecimal form.

Task#1
By using built-in memory blocks, design a 32x4 single port ram that can takes inputs of a 5-bit address (SW[8:4]), a 4-bit data to write (SW[3:0]), a write control (SW[9]), and a self-operated clock (KEY[0]) and output a 4-bit data to read. The read/write address displayed on HEX5 and HEX4, data to write is displayed on HEX2, and data to read is displayed on HEX0. These are all in the hexadecimal form when displayed. The the ram can always read the data in the specified address, but only when write control is enabled, the input data can be written into the address on the next self-operated positive clock edge.

Task#2
This task is kind of built on task 1, as task 1 uses 1 port, and this one uses 2 port, a more advanced version. Using IP catalog in Quartus Design, design a 32x4 2-port ram that is able to use one port supplying 5-bit address for read operation (address specified by a internal counter and scroll through the memory locations with approximately1Hz frequency) and the other port for 5-bit address (SW[8:4]) for write operation, it also takes a 4bit data to write and a write control (KEY[3]). The write address is displayed on HEX5 and HEX4, read address is displayed on HEX3 and HEX2, data to write on HEX1 and data to read on HEX0, all in hexadecimal form. So the read and write operations are completely separate: counter and read works on a set frequency on its own, write operation is only working when wr_en (KEY[3]) is pressed.

Task #3
For this task, we want to design a FIFO that uses a 16x8 2 port ram to store data, and operates like Queue. It displays the "queue's" full/empty status on LEDR9/8, and

when read is enabled, it reads the oldest data we input and displays it on HEX1 and HEX0, when write is enabled, it writes the specified data (SW[7:0]) to the newest address, and display this value on HEX5 and HEX4. All HEX display are in hexadecimal form.

Results

Task 1:

I started the task by designing the ram32x4, I ran ModelSim on that schematic and produced the waveform seen in Fig 1.

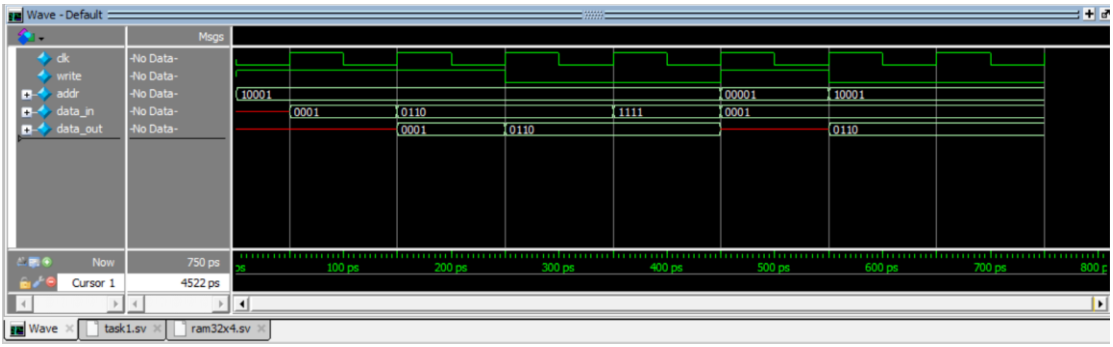

Fig 1. ram32x4_testbench simulation waveform

Test that for a specific address, if input data can be updated when write enable is true, if turning off write enable stops data updating, and test if data can be updated for other addresses. We expect to see when write is enabled, address is given, but there is no input, the output will be X, and when write is enabled, output data will be one cycle slower than input changes, and stops changes when write is disenabled. We also expect this to work the same for different addresses. More specific expectation as references to APPENDIX. A.i. The modelSim simulation waveform in Fig.1 matches what we have expected.

Then I worked on hexadecimal.sv which transforms the 4-bit binary code into its 7-bit hexadecimal value's seven segment expression. The modelSim simulation produces waveform seen in Fig 2-3.

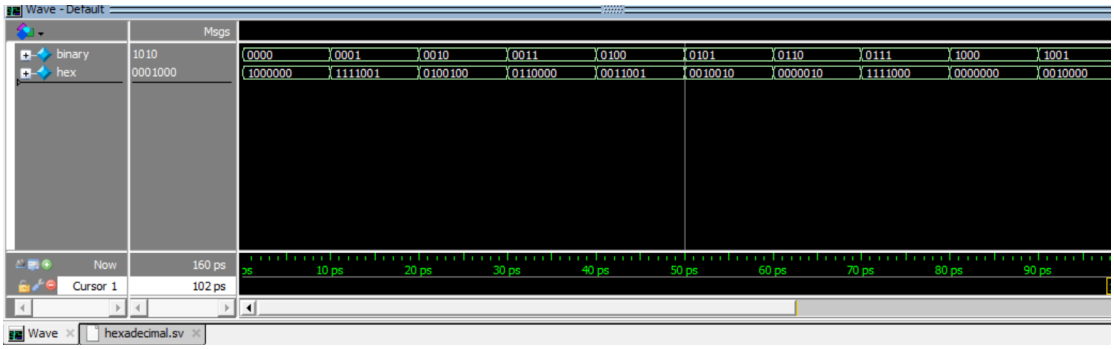

Fig. 2. Hexadecimal_testbench simulation waveform (part1)

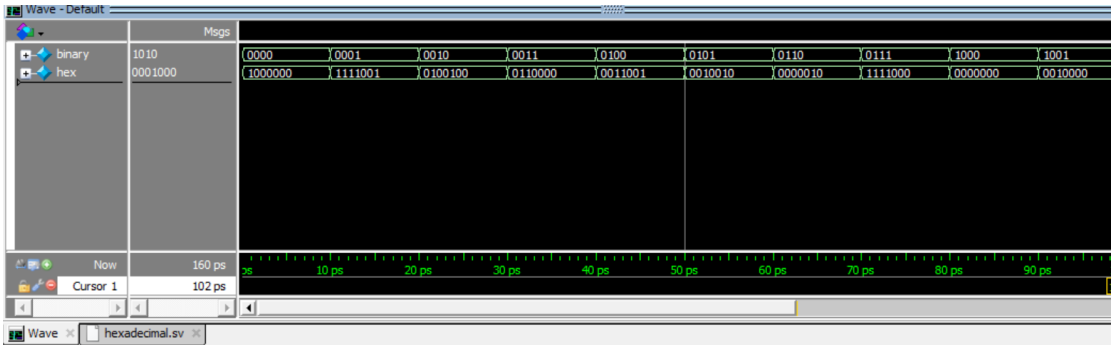

Fig. 3. Hexadecimal_testbench Simulation waveform (part2)

I tested all 16 cases of a 4bit binary code and expect to see the output matches their hexadecimal value's seven segment display expression, as in APPENDIX. A.ii. The modelSim simulation matches exactly as expected.

Then I got my code to process the KEY press to make it only true for one cycle in a long press and to process metastability from EE 271 (from previous lab, has been tested). Their modelSim simulation are as listed below in Fig. 4-5.

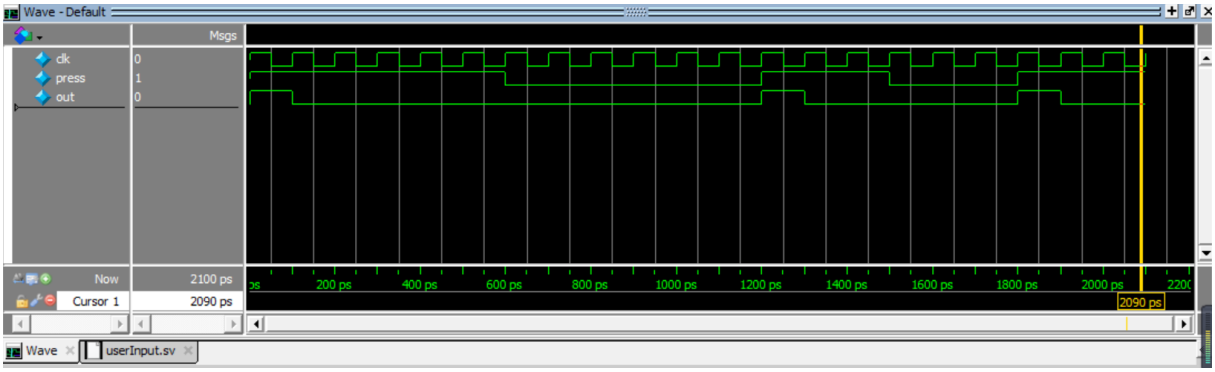

Fig. 4. userInput_testbench simulation waveform

I tested the transitions between the FSM (specific code please refer to APPENDIX A. iii) and we expect to see that every long press will only result in one true cycle, otherwise it stay false. The modelSim simulation matches what we expected.

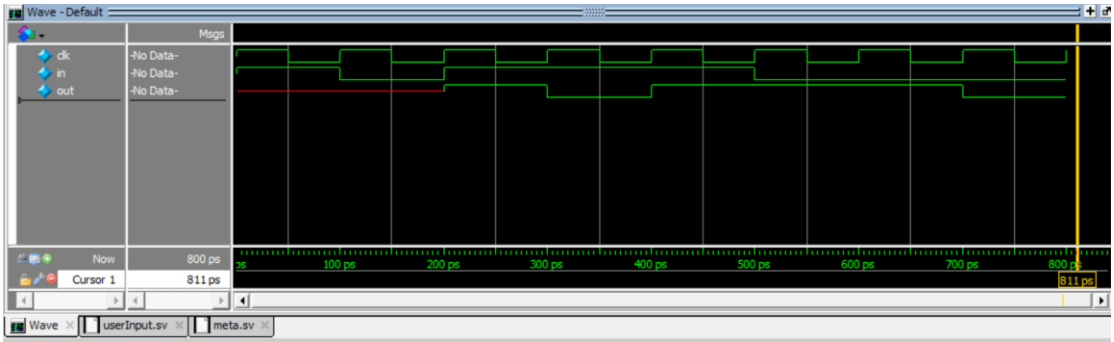

Fig. 5. meta_testbench simulation waveform

This is a very simple module that let the input to go through 2 DFF. Therefore, I just put in some values and expect to see that they are 2 cycles delayed in the output. Thus, we can see the modelSim simulation matches exactly with what we expected.

The last part is to instantiate all these modules and wires the inputs and outputs correctly in a top module. Task1's simulation produces the waveform in Fig.6-8.

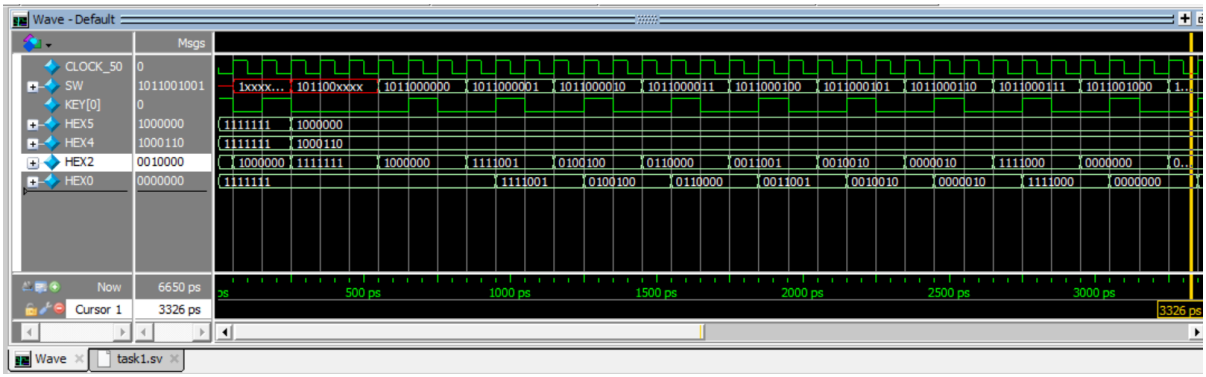

Fig. 6. Task1_testbench simulation waveform (part1)

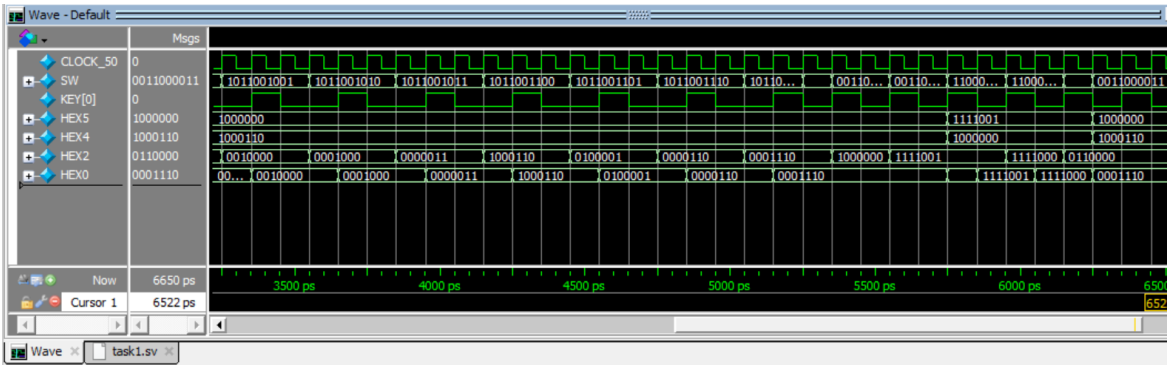

Fig. 7. Task1_testbench simulation waveform (part2)

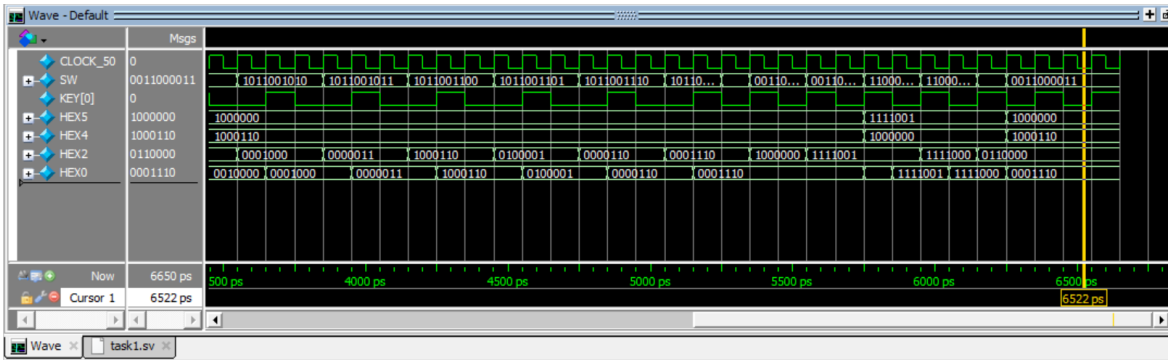

Fig. 8. Task1_testbench simulation waveform (part3)

I used basically the same testing method for this module as compared to ram32x4. I tested cases when only data or only address is given, and when a specific address is given, if input data can be updated when write enable is true, if turning off write enable stops data updating, if data can be updated for other addresses, and if data can be read again when re-accessing the addresss. We expect to see the HEX display turned off for lack of address or input; when write is enabled, output data will be one cycle slower than input changes (due to my testbench), and stops changes when write is disenabled. We also expect this to work the same for different addresses. The modelSim simulation waveform in Fig.6-8 matches what we have expected.

So basically task1 can function exactly as expected as required by the lab instruction. Here is a synthesis report for task1 in Fig.9.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pi
1	|task1	90 (0)	131 (0)	0	0	57	0
1	|hexadecimal:addr2|	7 (7)	0 (0)	0	0	0	0
2	|hexadecimal:data_read|	7 (7)	0 (0)	0	0	0	0
3	|hexadecimal:in|	7 (7)	0 (0)	0	0	0	0
4	|meta:clk_metatemp|	0 (0)	2 (2)	0	0	0	0
5	|ram32x4:RAM|	68 (68)	128 (128)	0	0	0	0
6	|userInput:human_clk|	1 (1)	1 (1)	0	0	0	0

Fig. 9. Task1 Synthesis report

Task 2:

In this task, I first followed the instruction on the specification to produce the mif file, and used the wizard to produce the ram .v file.

Then I implemented 32x4 two port ram, which produces waveform in Fig. 10-12.

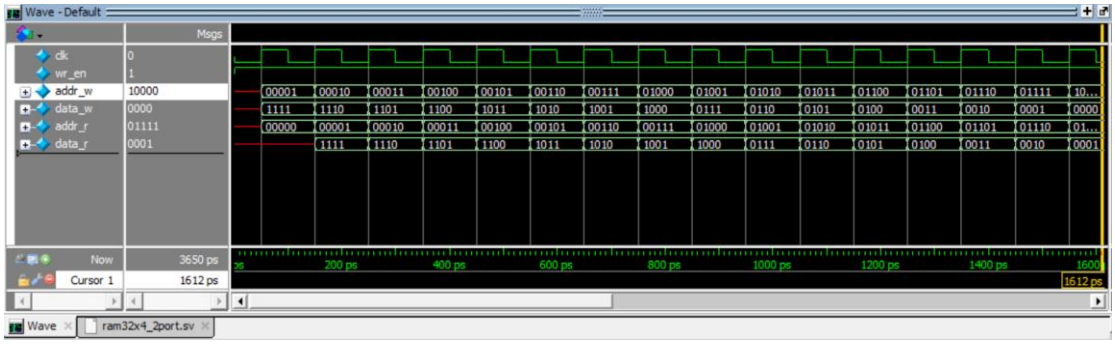

Fig. 10. Ram32x4_2port_testbench simulation waveform (part1)

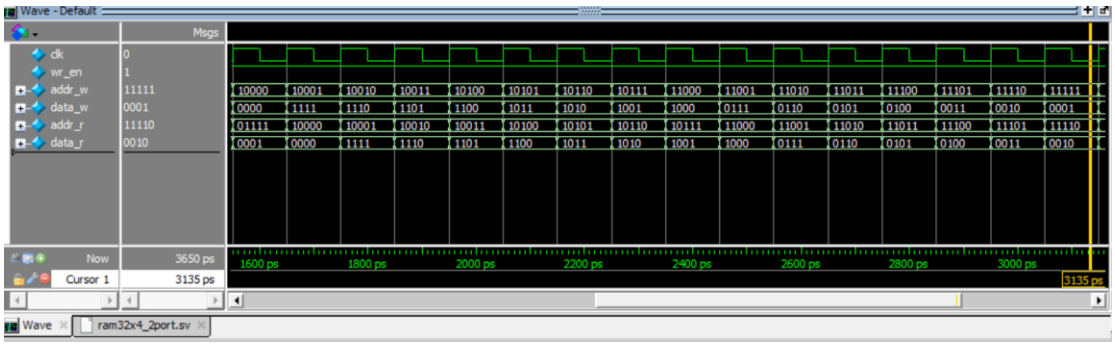

Fig. 11. Ram32x4_2port_testbench simulation waveform (part2)

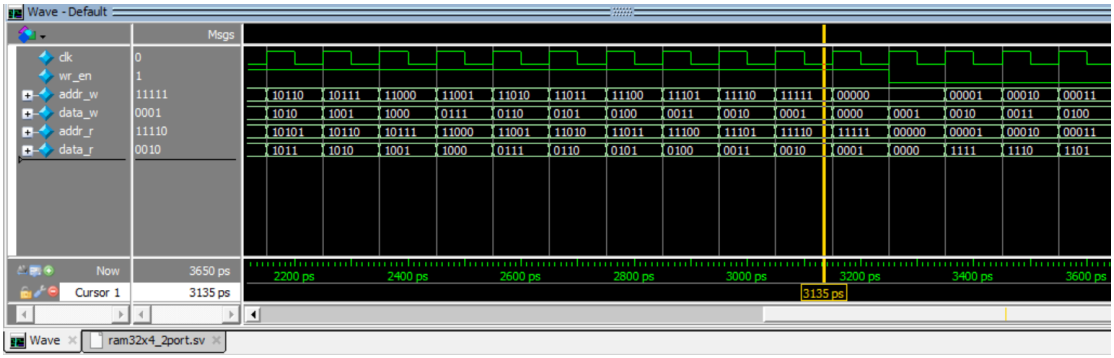

Fig. 12. Ram32x4_2port_testbench simulation waveform (part3)

I tested whether data can be written into each address (whether the stored value in the address updates) when write is enabled, and whether the address displays previous value if data has been input but write is disenabled. I expect to see data_r display the value one cycle after data_w (since we write data in the address 1 after read) and data_r should show previous value stored like in the first several cycles (3~4 cycles) after 33 cycles even though different data has been input. Thus, we can see that the modelSim simulation matches exactly with what we have expected.

Next I implemented the counter to scroll through the 32 addresses. Counter produces waveform in simulation as in Fig. 13-15.

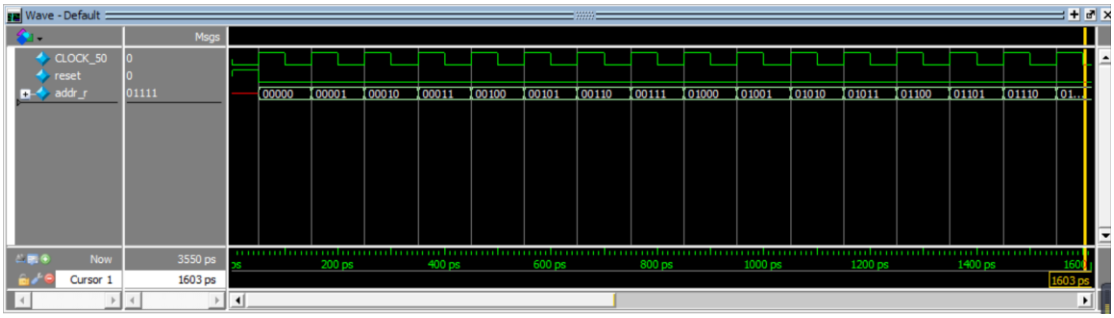

Fig. 13. counter_testbench simulation waveform (part1)

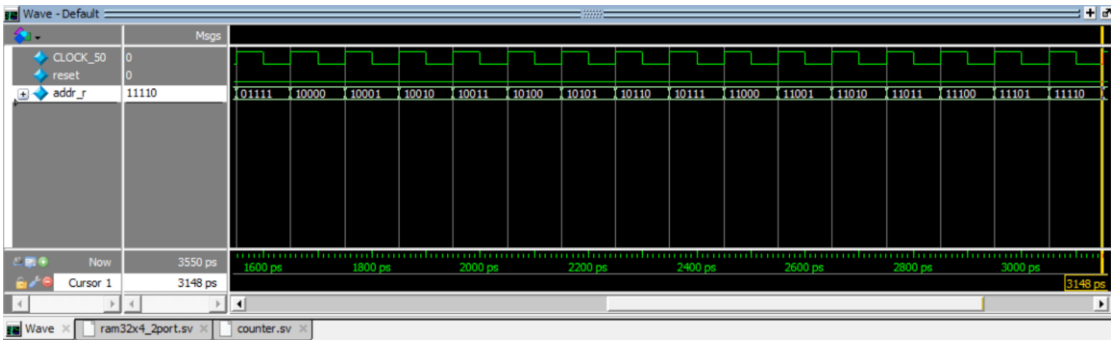

Fig. 14. counter_testbench simulation waveform (part2)

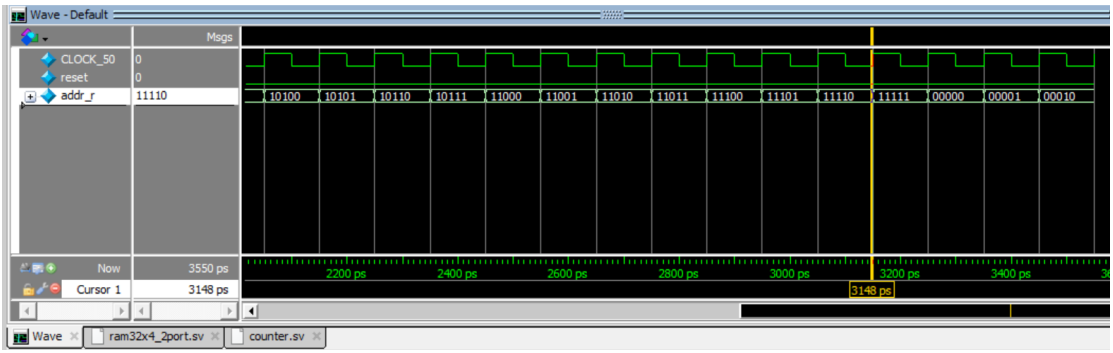

Fig. 15. counter_testbench simulation waveform (part3)

I simply tested after reset, whether counter actually counts to 0 to 31 and whether it will start over again. We expect to see addr_r becomes 0 after reset, and increment every clock cycle until reaching 31, which is when it start over. The modelSim simulation shows it working as expected.

Since I have the code of clock_divider.sv from EE 271 provided by the professor, I didn't implement a testbench for it. And we can use the hexadecimal.sv from task 1. So now everything is ready, I started to instantiate the modules and wires input and outputs in the top module task2.sv. which produced waveforms as in Fig. 16-19.

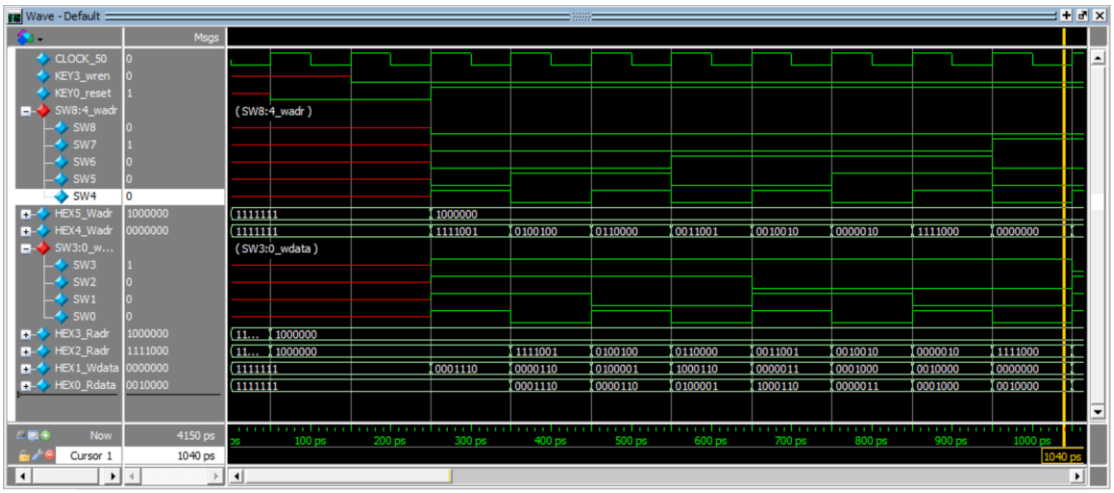

Fig. 16. Task2_testbench simulation waveform (part1)

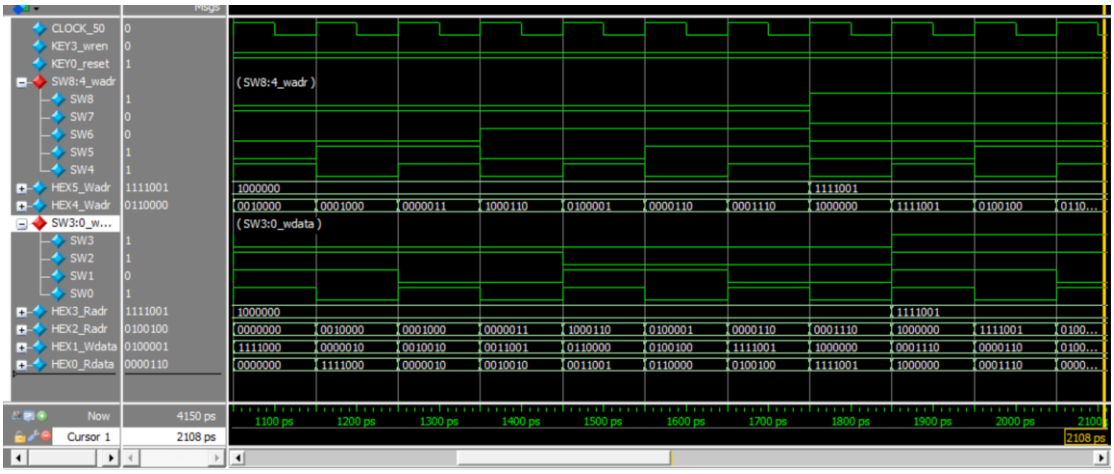

Fig. 17. Task2_testbench simulation waveform (part2)

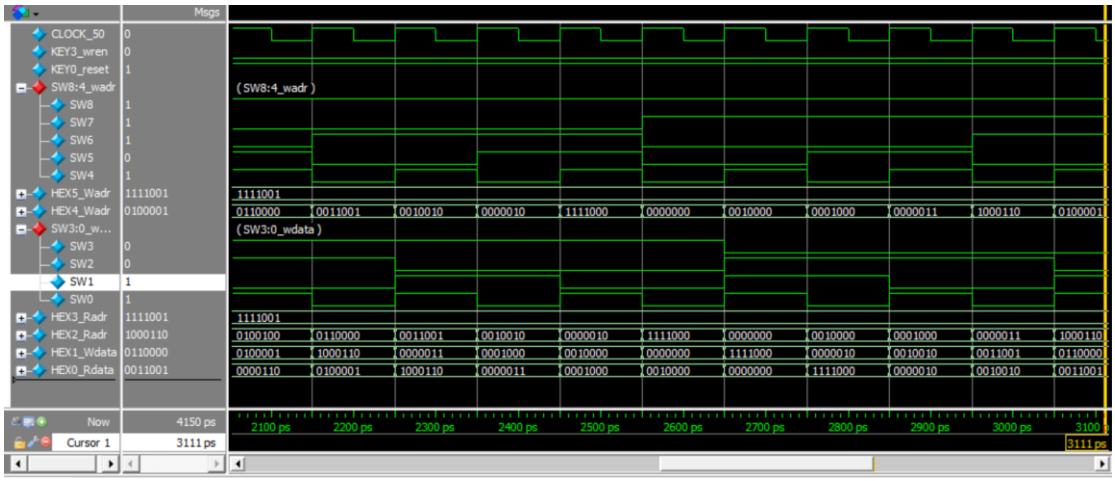

Fig. 18. Task2_testbench simulation waveform (part3)

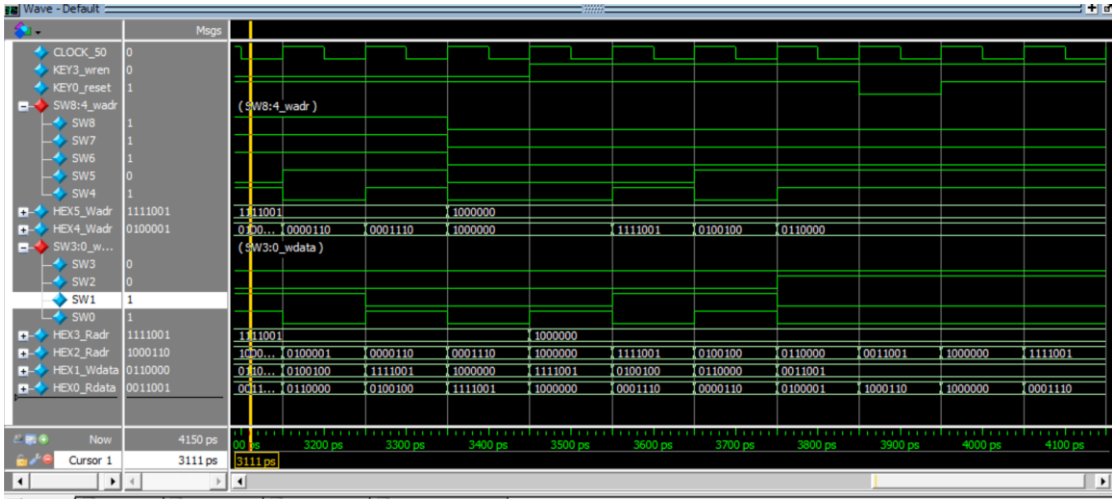

Fig. 19. Task2_testbench simulation waveform (part4)

I used basically the same testing method for this module as compared to ram32x4_2port. I tested whether data can be written into each address (whether the stored value in the address updates) when write is enabled, whether the address displays previous value if data has been input but write is disenabled, whether counter is outputting correct address so that the update can goes on smoothly, and whether reset works. I expect to see HEX3,2 display the value one cycle after HEX5,4(since we write data in the address 1 after read), HEX0 one cycle after HEX1. Thus, we can see that the modelSim simulation matches exactly with what we have expected. The modelSim simulation waveform in Fig.16-19 matches what we have expected.

Thus, I can say that task2 works as specified in the lab specification and as expected. Task2 has synthesis report as in Fig. 20.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	|DE1_SoC	60 (0)	32 (0)	128	0	57	0
1	|clock_divider:cdiv|	27 (27)	27 (27)	0	0	0	0
2	|counter:count|	5 (5)	5 (5)	0	0	0	0
3	|hexadecimal:r_addr2|	7 (7)	0 (0)	0	0	0	0
4	|hexadecimal:read|	7 (7)	0 (0)	0	0	0	0
5	|hexadecimal:w_addr2|	7 (7)	0 (0)	0	0	0	0
6	|hexadecimal:write|	7 (7)	0 (0)	0	0	0	0
7	|ram32x4_2...ram_2port|	0 (0)	0 (0)	128	0	0	0
1	|altsyncram:RAM_rtl_0|	0 (0)	0 (0)	128	0	0	0
1	|altsyncr...enerated|	0 (0)	0 (0)	128	0	0	0

Fig. 20. Task2 synthesis report

Task3:

In this task, I started implementing the ram, which produced the waveforms as in Fig. 21-24.

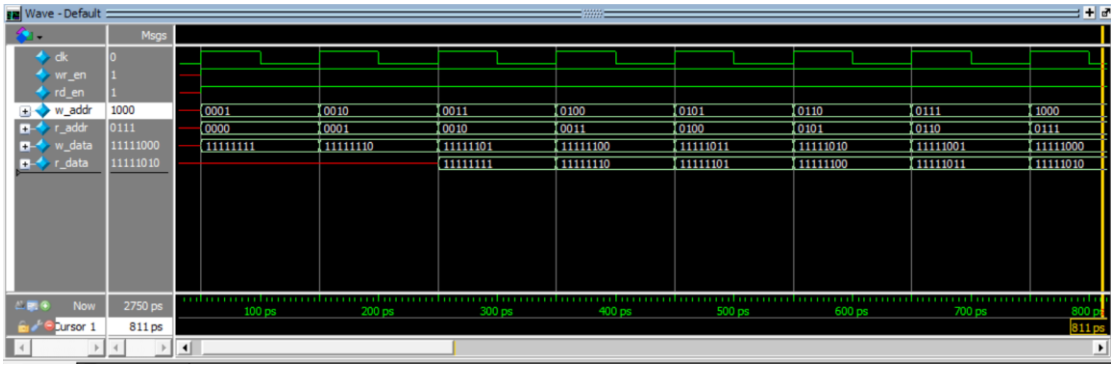

Fig. 21. Ram_2port_testbench simulation waveform (part1)

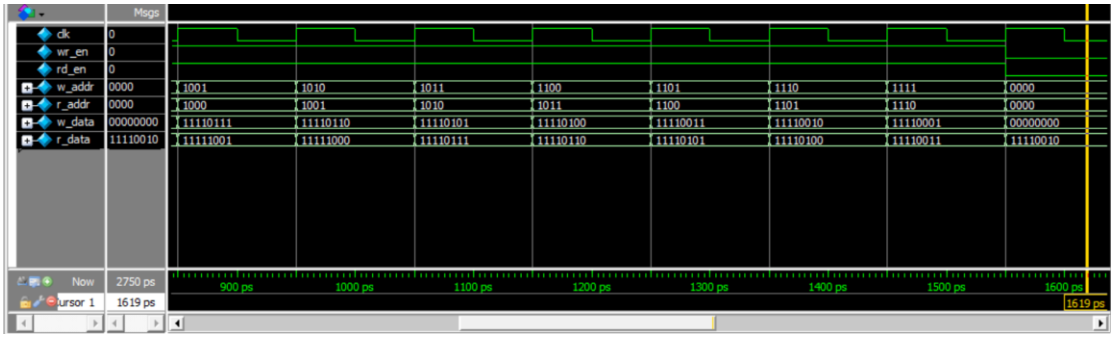

Fig. 22. Ram_2port_testbench simulation waveform (part2)

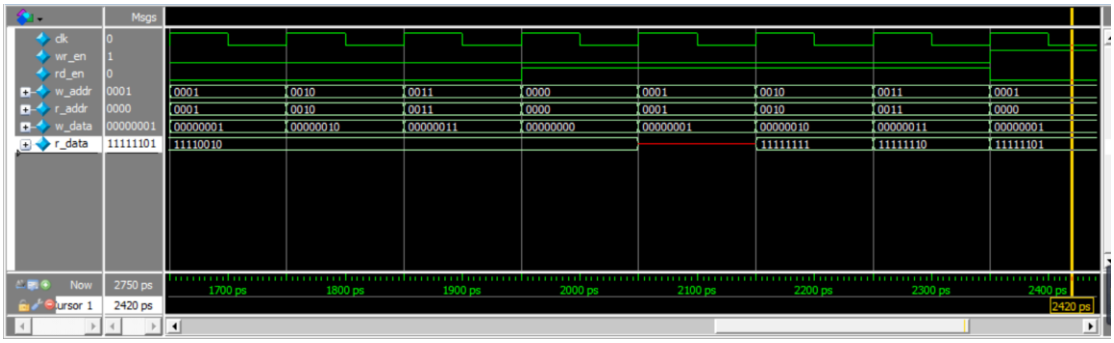

Fig. 23. Ram_2port_testbench simulation waveform (part3)

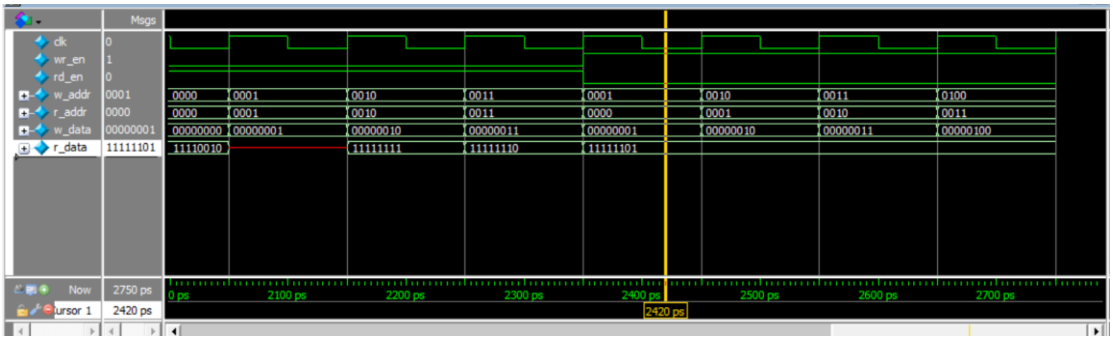

Fig. 24. Ram_2port_testbench simulation waveform (part4)

I tested cases of enabling both read and write updates data, disenabling both stops updating, enabling only read, enabling only write. Expect to see r_data updates 2 cycles after w_data for the first 15 cycles, then remains the same, then start to show values in the first 15 cycles, and then stay the same.

Then I started finishing the FIFO_Control, which produced the waveform in Fig. 25-28.

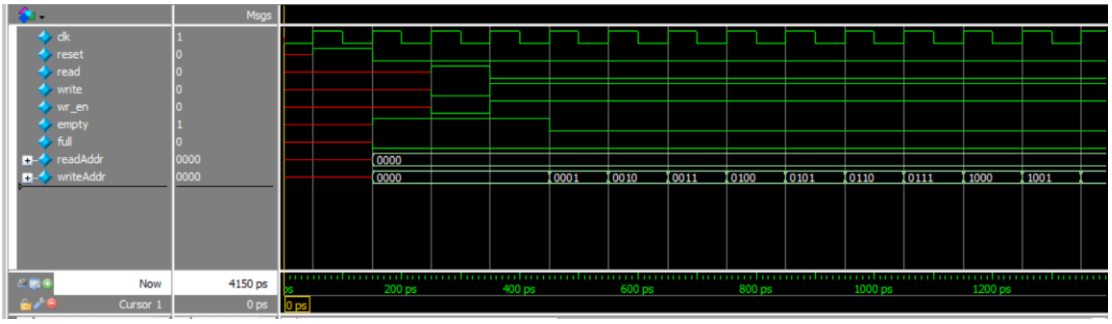

Fig.25 FIFO_Control_testbench simulation waveform(part1)

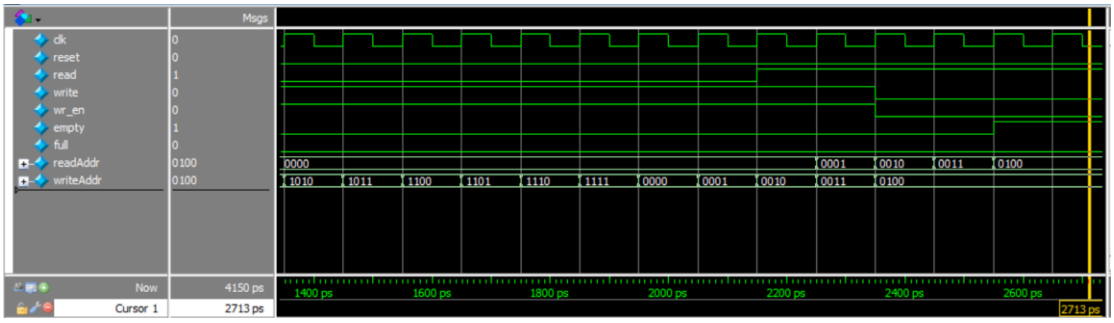

Fig.26 FIFO_Control_testbench simulation waveform(part2)

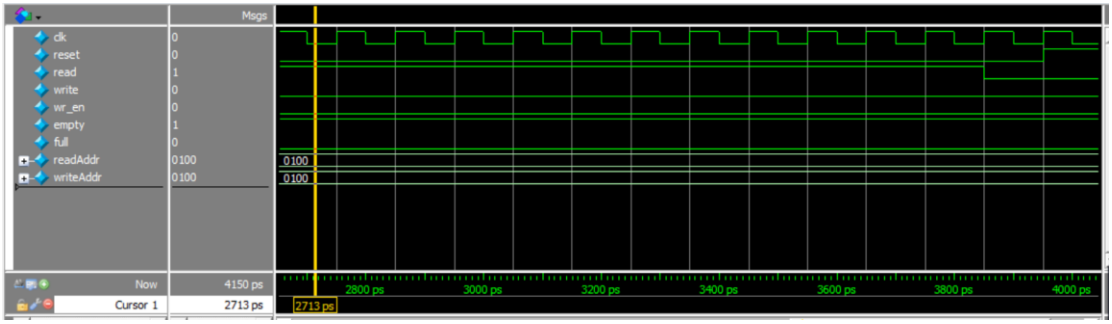

Fig.27 FIFO_Control_testbench simulation waveform(part3)

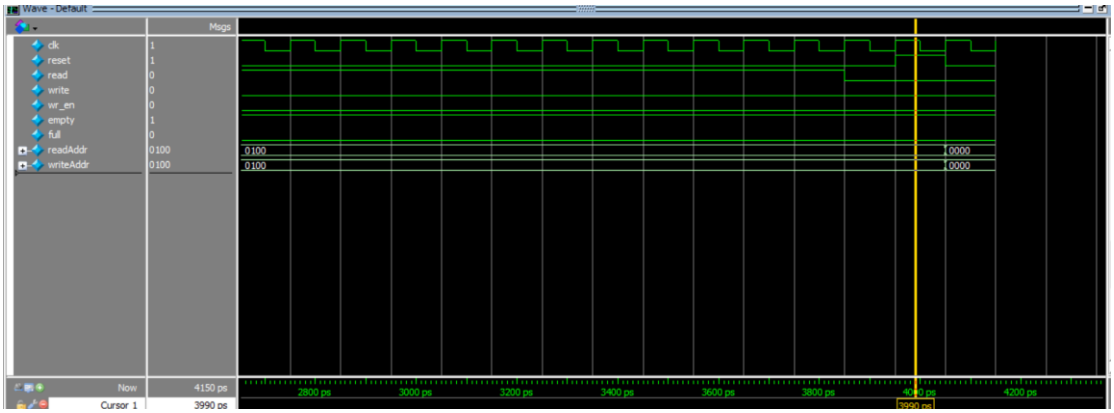

Fig.28 FIFO_Control_testbench simulation waveform (part4)

This tests whether the FIFO is able to output empty, do normal write, output full, do normal read, and test if the FIFO maintains its status if read and write are both disenabled. So we expect to see that write address increment, and after 22 cycles, it stops, then read address increments. Also, empty should be initialized at the beginning. Full should be reached in the middle. In simulation, we see that everything is working as expected except for full.

Then I go on to instantiate the FIFO_Control and ram in the FIFO to achieve the desired functionality of FIFO, which produced the waveform in Fig. 29-32.

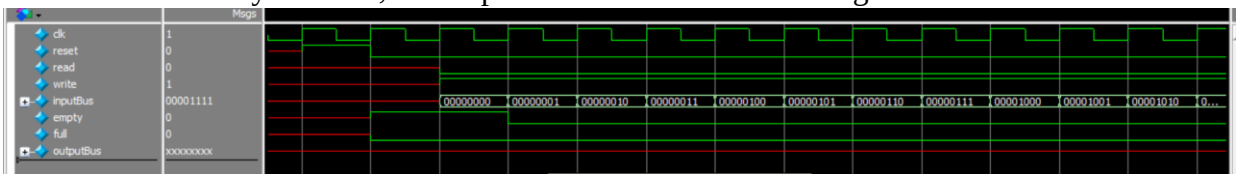

Fig.29 FIFO_testbench simulation waveform (part1)

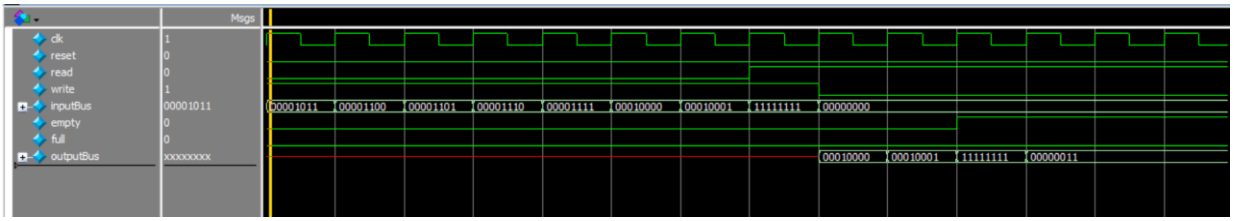

Fig.30 FIFO_testbench simulation waveform (part2)

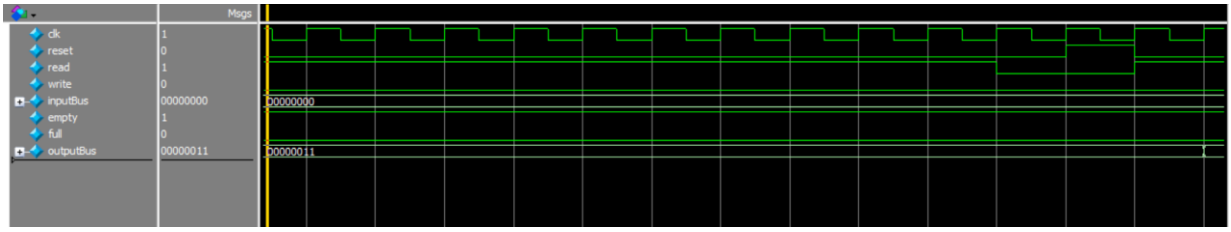

Fig.31 FIFO_testbench simulation waveform (part3)

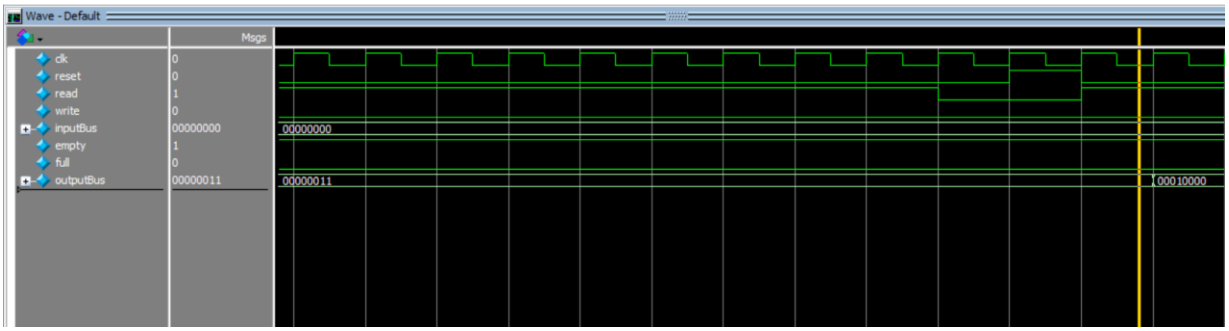

Fig.32 FIFO_testbench simulation waveform (part4)

FIFO tests the same cases as FIFO_Control as it works as FIFO_Control's upper level: it tests whether the FIFO is able to output empty, do normal write, output full, do normal read, and test if the FIFO maintains its status if read and write are both disenabled. We expect to see outputBus display what was shown in inputBus when write is enabled, and see that initially the FIFO is "empty." In the simulation, we see that the outputBus and empty works as expected, but the full is not.

Last, to fulfill the desired display on HEX, we wire the output of FIFO onto HEX in the top module task3.sv, which produced the waveform in Fig.33-36.

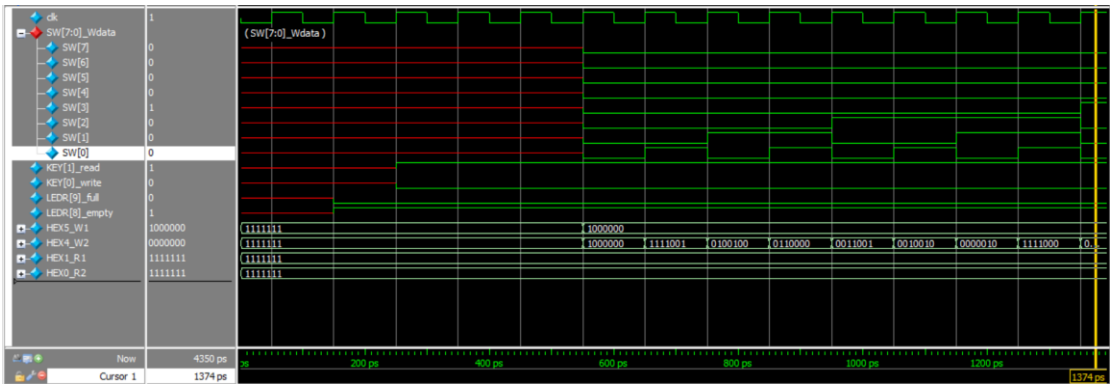

Fig.33 task3_testbench simulation waveform (part1)

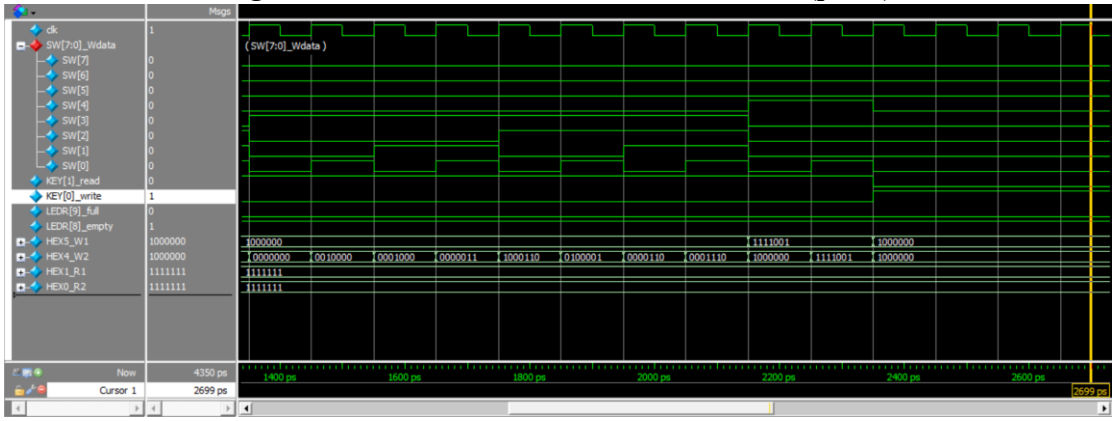

Fig.34 task3_testbench simulation waveform (part2)

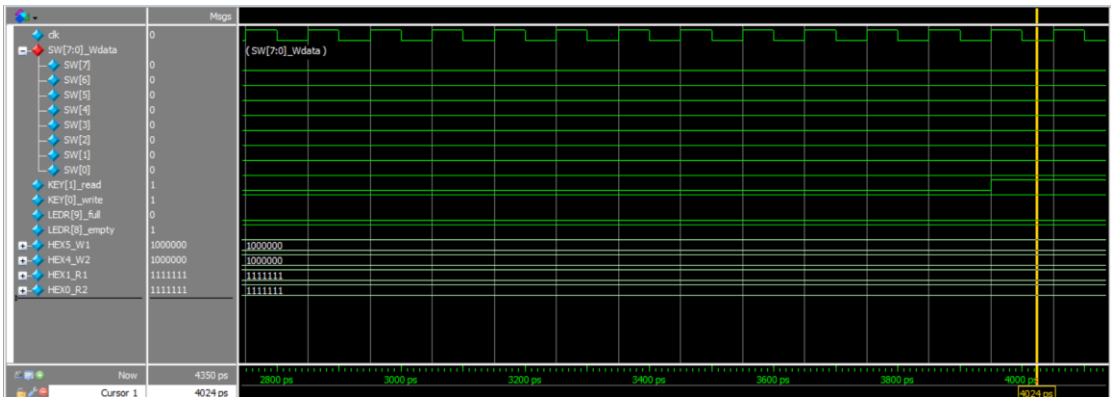

Fig.35 task3_testbench simulation waveform (part3)

It

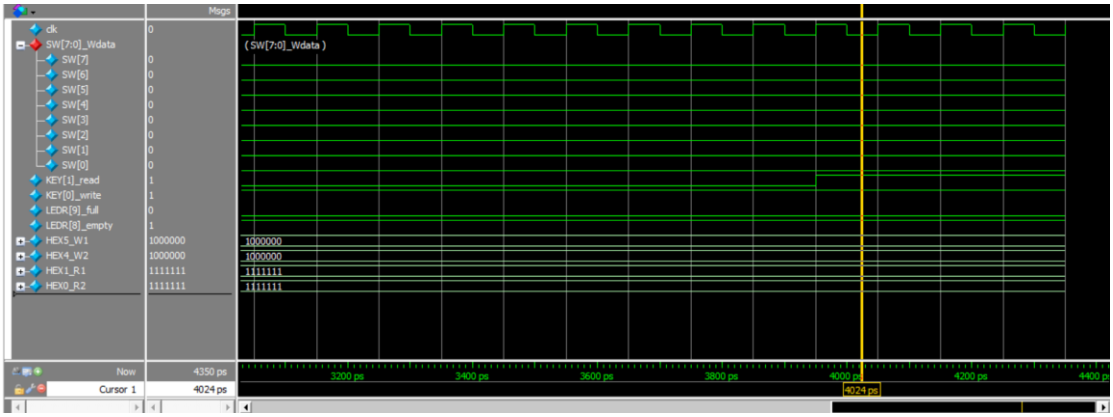

Fig.36 task3_testbench simulation waveform (part4)

Task3 still tests the same cases as the previous two: normal read and write operation, whether full and empty correctly asserts, whether the FIFO maintain its status when read and write are disenabled. We expect to see the HEX display correct hexadecimal value for read value and write value, and LEDR asserts true when it is empty or full. However, we see that in simulation, everything works except full is not changing

In general, task1 and task2 works as expected, but task3 is not very nicely dealing with metastability: after I implemented userInput to process a long pulse into only one-cycle true status and added 2 DFF to reduce metastability, the full is always malfunctioning, while empty, w_data, w_address, r_ address work fine. However, expcept this error, task3 works as expected to meet the standard of FIFO.

APPENDIX: (System Verilog Code)

A. (task1)

i. (ram32x4.sv)

```
// Cynthia Li
// 4/20/21
// EE 371 LAB2 TASK1

// Module ram32x4 defines a single port ram that uses the input 5-bit
// address to update the stored value in this address in the ram when input
// write is true, and always reads the value stored in this address.
module ram32x4 (clk, write, addr, data_in, data_out);
   input logic clk, write;
   input logic [4:0] addr;
   input logic [3:0] data_in;

   output logic [3:0] data_out;

   // the following defines the size of the built-in memory block
   logic [3:0] RAM [31:0];

   // the following is the write operation
   always_ff @(posedge clk) begin
      if (write) begin
         RAM[addr] <= data_in;
      end
   end

   // the following is the read operation
   assign data_out = RAM[addr];
endmodule
```

```
module ram32x4_testbench();
   logic clk, write;
   logic [4:0] addr;
   logic [3:0] data_in;

   logic [3:0] data_out;

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
   end

   ram32x4 dut (.clk, .write, .addr, .data_in, .data_out);

   initial begin
      write <= 1'b1; addr <= 5'b10001;    @(posedge clk); // test enabled write + no input
                                                         // expect X;
      data_in <= 4'b0001;                 @(posedge clk); // test enabled write + input
                                                         // expect X;
      data_in <= 4'b0110;                 @(posedge clk); // test if data continues to be updated
                                                         // expect 4'b0001
      write <= 1'b0;                      @(posedge clk); // test disenable write + no input
                                                         // expect 4'b0110
      data_in <= 4'b1111;                 @(posedge clk); // test disenable write + input
                                                         // expect  4'b0110
                                                         // expect 4'b0110
      write <= 1'b1; addr <= 5'b00001;                   // test if other address's value can be changed
      data_in <= 4'b0001;                 @(posedge clk); // expect
      write <= 1'b0; addr <= 5'b10001;    @(posedge clk); // expect 4'b0001
                                          @(posedge clk); // expect 4'b0110

      $stop;
   end
endmodule
```

ii. (hexadecimal.sv)

***** this module will be used for task 2 and task3, same module applies and will not be listed again**

```
// Cynthia Li
// 4/20/21
// EE 371 LAB2 TASK1

// Module hexadecimal takes a 4bit input value, and output its
// corresponding hexadecimal value seven segment display.
module hexadecimal (binary, hex);
   input logic [3:0] binary;
   output logic [6:0] hex;

   // the following is the logic that link each 4bit binary value
   // to its hexadecimal seven segment display expression
   always_comb begin
      case(binary)               // display
         4'b0000: hex = 7'b1000000; // 0
         4'b0001: hex = 7'b1111001; // 1
         4'b0010: hex = 7'b0100100; // 2
         4'b0011: hex = 7'b0110000; // 3
         4'b0100: hex = 7'b0011001; // 4
         4'b0101: hex = 7'b0010010; // 5
         4'b0110: hex = 7'b0000010; // 6
         4'b0111: hex = 7'b1111000; // 7
         4'b1000: hex = 7'b0000000; // 8
         4'b1001: hex = 7'b0010000; // 9
         4'b1010: hex = 7'b0001000; // A
         4'b1011: hex = 7'b0000011; // b
         4'b1100: hex = 7'b1000110; // C
         4'b1101: hex = 7'b0100001; // d
         4'b1110: hex = 7'b0000110; // E
         4'b1111: hex = 7'b0001110; // F
         default: hex = 7'b1111111; // turn off display
      endcase
   end
endmodule

module hexadecimal_testbench();

   logic [3:0] binary;
   logic [6:0] hex;

   hexadecimal dut (.binary, .hex);

   initial begin
      binary = 4'b0000; #10;
      binary = 4'b0001; #10;
      binary = 4'b0010; #10;
      binary = 4'b0011; #10;
      binary = 4'b0100; #10;
      binary = 4'b0101; #10;
      binary = 4'b0110; #10;
      binary = 4'b0111; #10;
      binary = 4'b1000; #10;
      binary = 4'b1001; #10;
      binary = 4'b1010; #10;
      binary = 4'b1011; #10;
      binary = 4'b1100; #10;
      binary = 4'b1101; #10;
      binary = 4'b1110; #10;
      binary = 4'b1111; #10;
      $stop;
   end
endmodule
```

iii. (userInput.sv)

***** this module will be used for task3, same module applies and will not be listed again**

```
1  // Cynthia Li
2  // 4/20/21
3  // EE 371 LAB2 TASK1
4
5  // Module userInput takes a input press that indicates the status
6  // of a KEY, and should process it so that every press is true for
7  // only one cycle and otherwise stays false. It will output this
8  // processed KEY status to out.
9  module userInput (clk, press, out);
10    input logic clk, press;
11    // press: true when the KEY is pressed
12    output logic out;
13
14    enum {on, off} ps, ns;
15
16    // the next state and output logic
17    always_comb
18       case(ps)
19          on: if (press) begin // key contiuously being pressed
20                   ns = on;    // & out has been true for 1 cycle
21                   out = 0;
22                end
23             else begin        // key is released
24                   ns = off;
25                   out= 0;
26                end
27          off:  if (press) begin  // key is pressed (the action)
28                      ns = on;    // make out true for one cycle
29                      out= 1;
30                   end
31                else begin         // key is not pressed
32                      ns = off;
33                      out = 0;
34                   end
35          default: begin
36                      ns = off;
37                      out = 0;
38                   end
39       endcase
40
41    always_ff @(posedge clk)
42       ps <= ns;
43
44 endmodule
45
46 module userInput_testbench();
47    logic clk, press;
48    logic out;
49
50    userInput dut (.clk, .press, .out);
51
52    // Set up the clock.
53    parameter CLOCK_PERIOD=100;
54    initial clk=1;
55    always begin
56       #(CLOCK_PERIOD/2);
57       clk = ~clk;
58    end
59
60    // Set up the inputs to the design. Each line is a clock cycle.
61    initial begin
62    press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed
63    press <= 1;   repeat(3) @(posedge clk); // on state: key is pressed
64    press <= 0;   repeat(3) @(posedge clk); // on state: key is released
65    press <= 0;   repeat(3) @(posedge clk); // off state: key is unpressed
66    press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed
67    press <= 0;   repeat(3) @(posedge clk); // on state: key is released
68    press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed
69
70    $stop; // End the simulation.
71    end
72 endmodule
```

iv. (meta.sv)

***** this module will be used for task3, same module applies and will not be listed again**

```
// Cynthia Li
// 4/20/21
// EE 371 LAB2 TASK1

// Module meta takes an input, pass it through
// two DFF to prevent metastability.
module meta (clk, in, out);
   input logic clk, in;
   output logic out;
   logic temp;

   // the following passes in through 2 DFF
   always_ff @(posedge clk)
         temp <= in;
   always_ff @(posedge clk)
         out <= temp;
endmodule

module meta_testbench();
   logic clk, in;
   logic out;

   // Set up the clock.
   parameter CLOCK_PERIOD=100;
   initial clk=1;
   always begin
      #(CLOCK_PERIOD/2);
      clk = ~clk;
   end

   meta dut (.*);

   initial begin
   in <= 1; @(posedge clk);
   in <= 0; @(posedge clk);
   in <= 1; repeat(3)@(posedge clk);
   in <= 0; repeat(3)@(posedge clk);
   $stop; // End the simulation.
   end
endmodule
```

v. (task1.sv)

```
// Cynthia Li
// EE 371 LAB2 TASK1

// Module task1 is the top module for task1 project
// it uses a 32x4 single port ram that can takes inputs of a 5-bit address (SW[8:4]),
// a 4-bit data to write (SW[3:0]), a write control (SW[9]), and a self-operated
// clock (KEY[0]) and output a 4-bit data stored in the address to read.
// The read/write address is displayed on HEX5 and HEX4, data to write is
// displayed on HEX2, and data to read is displayed on HEX0.
module task1(CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input logic CLOCK_50;
   input logic [9:0] SW;
   input logic [3:0] KEY;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   // the following turns off the display for HEX3 and HEX1
   assign HEX3 = 7'b1111111;
   assign HEX1 = 7'b1111111;

   logic [3:0] out;
   logic clk_temp, clk;

   // the following process the actual key press to prevent metastability
   userInput human_clk (.clk(CLOCK_50), .press(~KEY[0]), .out(clk_temp));
   meta clk_metatemp (.clk(CLOCK_50), .in(clk_temp), .out(clk));

   ram32x4 RAM (.clk, .write(SW[9]), .addr(SW[8:4]), .data_in(SW[3:0]), .data_out(out));

   // the following turns on the hexadecimal display on HEX5-4 for input address
   hexadecimal addr1 (.binary({3'b000, SW[8]}), .hex(HEX5));
   hexadecimal addr2 (.binary(SW[7:4]), .hex(HEX4));

   // the following turns on the hexadecimal display on HEX2 for input data_to_write
   hexadecimal in (.binary(SW[3:0]), .hex(HEX2));

   // the following turns on the hexadecimal display on HEX0 for output data_to_read
   hexadecimal data_read (.binary(out), .hex(HEX0));

endmodule

module task1_testbench();
   logic [9:0] SW;
   logic [3:0] KEY;
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   logic CLOCK_50;
   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   task1 dut (.*);

   logic clk;
   assign clk = CLOCK_50;
   integer i;

   initial begin
   KEY[0] <= 1;                  @(posedge clk);
   KEY[0] <= 0; SW[9] <= 1;
   SW[3:0] <= 4'b0000;           @(posedge clk);         // test only data is given
   KEY[0] <= 1;                  @(posedge clk);
   KEY[0] <= 0; SW[8:4] <= 5'b01100;
   SW[3:0] <= 4'bX;              @(posedge clk);         // test only address is given

   SW[9] <= 1; SW[8:4] <= 5'b01100;    @(posedge clk);// test if for a ramdom address
   for (i = 0; i < 16; i++) begin                      // if the stored value updates
      KEY[0] <= 0;               @(posedge clk);
      SW[3:0] <= i;              @(posedge clk);
      KEY[0] <= 1;               @(posedge clk);
   end
   KEY[0] <= 0; SW[9] <= 0;            @(posedge clk);// disenabled write
   KEY[0] <= 1; SW[3:0] <= 4'b0000;    @(posedge clk);// test if display stop update
   KEY[0] <= 0;                        @(posedge clk);
   KEY[0] <= 1; SW[3:0] <= 4'b0001;    @(posedge clk);
   KEY[0] <= 0;                        @(posedge clk);

   SW[9] <= 1; SW[8:4] <= 5'b10000;                    // enable write
   KEY[0] <= 1;                        @(posedge clk);
   KEY[0] <= 0;                        @(posedge clk);// test if display updates
   KEY[0] <= 1; SW[3:0] <= 4'b0111;    @(posedge clk);
   KEY[0] <= 0;                        @(posedge clk);// test if display updates
   KEY[0] <= 1; SW[3:0] <= 4'b0011;    @(posedge clk);
```

```
    SW[9] <= 0; SW[8:4] <= 5'b01100;
    KEY[0] <= 0;              @(posedge clk);
    KEY[0] <= 1;              @(posedge clk);
    KEY[0] <= 0;              @(posedge clk);
    KEY[0] <= 1;              @(posedge clk);
    $stop;
    end
endmodule
```

B. (task2)

i.(ram32x4_2port.sv)

```
// Cynthia Li
// 4/20/21
// EE 371 LAB2 TASK1

// Module ram32x4_2port defines a two port 32x4 ram that uses two input
// 5-bit one for write operation, to update the value in the address with
// input 4-bit data_w, when input wr_en is true; the other for read
// operation,it outputs the 4-bit value stored in this address to data_r.

`timescale 1 ps / 1 ps
module ram32x4_2port(clk, wr_en, addr_w, addr_r, data_w, data_r);
    input logic clk, wr_en;
    input logic [4:0] addr_w, addr_r;
    input logic [3:0] data_w;
    output logic [3:0] data_r;

    // the following defines the size of the memory block
    logic [3:0] RAM [31:0];

    // the following is the write operation
    always_ff @(posedge clk) begin
        if (wr_en) begin
            RAM[addr_w] <= data_w;
        end
    end

    // the following is the read operation
    assign data_r = RAM[addr_r];
endmodule
```

```
`timescale 1 ps / 1 ps
module ram32x4_2port_testbench();
    logic clk, wr_en;
    logic [4:0] addr_w, addr_r;
    logic [3:0] data_w;
    logic [3:0] data_r;

    // Set up a simulated clock.
    parameter CLOCK_PERIOD=100;

    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    ram32x4_2port ram32x4_2port_test(clk, wr_en, addr_w, addr_r, data_w, data_r);

    integer i;

    initial begin
        wr_en <= 1'b1;                          @(posedge clk);
        for (i = 0; i < 32; i++)  begin
            addr_w <= i + 1;
            data_w <= 31 - i;
            addr_r <= i;                        @(posedge clk); // test enabled write + no input
        end
        wr_en <= 1'b0;
```

```
        for (i = 0; i < 4; i++)  begin
            addr_w <= i;
            data_w <= i + 1;
            addr_r <= i;                         @(posedge clk); // test enabled write + no input
        end
        $stop;
    end
endmodule
```

ii.(counter.sv)

```
// Cynthia Li
// 4/20/21
// EE 371 LAB2 TASK1

// module counter takes an input 50mHz clock and a reset signal
// and output a 5-bit value to addr_r. The value is updated from
// 0 to 32 as a counter and will start over afterwards. reset will
// let counter start over as wel.
module counter (clk, reset, addr_r);
   input logic clk, reset; // should be a slow clock with a 1s/ period
   output logic [4:0] addr_r;

   // the following defines when counter increment
   // and when to start over
   always_ff @(posedge clk) begin
      if (reset) begin
         addr_r <= 0; // addr_r is initially set to 0
      end
      else begin
         if (addr_r < 31) // when not reaching the end
            addr_r <= addr_r + 1; // addr_r increment
         else             // when reaching the end
            addr_r <= 0;          // addr_r start over
      end
   end
endmodule
```

```
module counter_testbench();
   logic CLOCK_50, reset;
   logic [4:0] addr_r;

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   counter count (.clk(CLOCK_50), .reset, .addr_r);

   integer i;
   initial begin
      reset <= 1;    @(posedge CLOCK_50);
      reset <= 0;    @(posedge CLOCK_50);
      repeat(34)     @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

iii. clock_divider.sv

```
// Cynthia Li
// EE 371 LAB2 TASK2

// divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
module clock_divider (clock, reset, divided_clocks);
   input logic reset, clock;
   output logic [31:0] divided_clocks = 0;

   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule
```

*** since this is provided by EE 271 instruction team, I didn't implement a testbench for it.

iv. task2.sv

```
// Cynthia Li
// EE 371 LAB2 TASK2

// Module task2 is the top module for task1 project
// it uses a 32x4 single port ram that can takes inputs of a 5-bit address (SW[8:4]),
// a 4-bit data to write (SW[3:0]), a write control (SW[9]), and a self-operated
// clock (KEY[0]) and output a 4-bit data stored in the address to read.
// The read/write address is displayed on HEX5 and HEX4, data to write is
// displayed on HEX2, and data to read is displayed on HEX0.

`timescale 1 ps / 1 ps
module task2(CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input logic CLOCK_50;
   input logic [9:0] SW;
   input logic [3:0] KEY;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   // the following produces a slowed clock
   parameter whichClock = 25; // 0.365 Hz clock
   logic [31:0] div_clk;
   clock_divider cdiv (.clock(CLOCK_50),
                       .reset(reset),
                       .divided_clocks(div_clk));

   // clk; allows for easy switching between simulation and board clocks
   logic clk;
   // assign clk = CLOCK_50; // for simulation
   assign clk = div_clk[whichClock]; // for board

   logic [3:0] data_read, data_w;
   logic [4:0] addr_read;

   assign data_w = SW[3:0];

   // the following sets up the counter for read addresses
   counter count (.clk, .reset(~KEY[0]), .addr_r(addr_read));
   // the following sets up the ram
   ram32x4_2port ram_2port (.clk(CLOCK_50), .wr_en(~KEY[3]), .addr_w(SW[8:4]),
                            .addr_r(addr_read), .data_w, .data_r(data_read));
   // the following sets up HEX displays for addresses and datas
   hexadecimal read (.binary(data_read), .hex(HEX0));

   hexadecimal write (.binary(data_w), .hex(HEX1));

   hexadecimal r_addr1 (.binary({3'b0, addr_read[4]}), .hex(HEX3));
   hexadecimal r_addr2 (.binary(addr_read[3:0]), .hex(HEX2));

   hexadecimal w_addr1 (.binary({3'b0, SW[8]}), .hex(HEX5));
   hexadecimal w_addr2 (.binary(SW[7:4]), .hex(HEX4));
endmodule

`timescale 1 ps / 1 ps
module task2_testbench();
   logic [9:0] SW;
   logic [3:0] KEY;
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;


   logic CLOCK_50, clk;
   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

```

```
63    initial begin
64       CLOCK_50 <= 0;
65       forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
66    end
67
68    assign clk = CLOCK_50;
69
70    task2 dut (.CLOCK_50, .SW, .KEY, .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);
71
72    integer i;
73
74    initial begin
75                                  @(posedge clk);
76    KEY[0] <= 0;                  @(posedge clk);
77    KEY[3] <= 0;                  @(posedge clk); // reset counter, enable wr_en
78
79    for (i = 0; i < 32; i++)  begin
80       KEY[0] <= 1;
81       SW[8:4] <= i + 1;
82       SW[3:0] <= 31 - i;         @(posedge clk); // test enabled write + no input
83    end
84    KEY[3] <= 1;
85    for (i = 0; i < 4; i++)  begin
86       SW[8:4] <= i;
87       SW[3:0] <= i + 1;          @(posedge clk); // test enabled write + no input
88       end
89    KEY[0] <= 0;                  @(posedge clk); // see if reset works
90    KEY[0] <= 1;   repeat(2)      @(posedge clk);
91    $stop;
92    end
93 endmodule
```

C. (task3)

i. (FIFO_Control.sv)

```
1  // This module defines when the FIFO is empty and full
2  // it input the read/ write enable, and output wr_en,
3  // emty, full, status, and read, write address
4  module FIFO_Control #(parameter depth = 4)
5                       (input logic clk, reset,
6                        input logic read, write,
7                        output logic wr_en,
8                        output logic empty, full,
9                        output logic [depth-1:0] readAddr, writeAddr);
10
11    /*    Define_Variables_Here      */
12
13    /*    Combinational_Logic_Here   */
14    assign wr_en = write & ~full;
15
16    /*    Sequential_Logic_Here      */
17    // the following is the logic that defines when the FIFO is full and empty
18    // and when the
19    always_ff @(posedge clk) begin
20       if(reset) begin   // defin initialization
21          readAddr <= '0;
22          writeAddr <= '0;
23          full <= 1'b0;
24          empty <= 1'b1;
25        end else if (((writeAddr + 1) == readAddr) & write & ~ read) begin
26                                                  // define reaching "full"
27          readAddr <= readAddr;
28          writeAddr <= writeAddr + 1;
29          full <= 1'b1;
30          empty <= 1'b0;
31       end else if (((readAddr + 1) == writeAddr) & read & ~write) begin
```

```
31          end else if (((readAddr + 1) == writeAddr) & read & ~write) begin
32                                                      // define reaching empty
33             readAddr <= readAddr + 1;
34             writeAddr <= writeAddr;
35             full <= 1'b0;
36             empty <= 1'b1;
37          end else if (full & write & ~read) begin // define upper boundary
38             readAddr <= readAddr;
39             writeAddr <= writeAddr;
40             full <= 1'b1;
41             empty <= 1'b0;
42          end else if (empty & read & ~write) begin // define lower boundary
43             readAddr <= readAddr;
44             writeAddr <= writeAddr;
45             full <= 1'b0;
46             empty <= 1'b1;
47          end else if (read & ~write) begin
48             readAddr <= readAddr + 1;
49             writeAddr <= writeAddr;
50             full <= 1'b0;
51             empty <= 1'b0;
52          end else if (~read & write) begin
53             readAddr <= readAddr;
54             writeAddr <= writeAddr + 1;
55             full <= 1'b0;
56             empty <= 1'b0;
57          end else if (read & write) begin
58             readAddr <= readAddr + 1;
59             writeAddr <= writeAddr + 1;
60             empty <= empty;
61             full <= full;
62          end else begin
63             readAddr <= readAddr;
64             writeAddr <= writeAddr;
65             empty <= empty;
66             full <= full;
67          end
68       end
69    endmodule
70
71    module FIFO_Control1_testbench();
72       logic clk, reset;
73       logic read, write;
74       logic wr_en;
75       logic empty, full;
76       logic [3:0] readAddr, writeAddr;
77
78       // Set up a simulated clock.
79       parameter CLOCK_PERIOD=100;
80
81       initial begin
82          clk <= 0;
83          forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
84       end
85
86       FIFO_Control control (.*);
87
88       integer i;
89
90       initial begin
91                                                      @(posedge clk);
92          reset <= 1;                                 @(posedge clk);
93          reset <= 0;                                 @(posedge clk);
94          read <= 1; write <= 0;                      @(posedge clk); // test lower boundary
95          read <= 0; write <= 1;          repeat(18)  @(posedge clk); // test normal w/full/upper boundary
96          read <= 1; write <= 1;          repeat(2)   @(posedge clk); // test w&r
97          read <= 1; write <= 0;          repeat(15)  @(posedge clk); // test normal r
98          read <= 0; write <= 0;                      @(posedge clk); // test disenabling
99          reset <= 1;                                 @(posedge clk);
100         reset <= 0;                                 @(posedge clk);
101         $stop;
102      end
103   endmodule
104
```

ii. (FIFO.sv)

```
1  // This module uses a 2port ram as FIFO ("Queue")
2  // it takes input reset, read, and write, if it reads
3  // it's outputing a 8bit data to outputBus, if it write
4  // it update the most recent address to the value equal
5  // to inputBus.
6  module FIFO  #(parameter depth = 4,
7                 parameter width = 8)
8                (input logic clk, reset,
9                 input logic read, write,
10                input logic [width-1:0] inputBus,
11                output logic empty, full,
12                output logic [width-1:0] outputBus);
13
14    /*    Define_Variables_Here      */
15    logic [depth-1:0] w_addr, r_addr;
16    logic wr_en;
17
18    /*       Instantiate_Your_Dual-Port_RAM_Here       */
19
20    ram_2port RAM (.clk, .wr_en(wr_en), .rd_en(read), .w_addr(w_addr),
21                   .r_addr(r_addr), .w_data(inputBus), .r_data(outputBus));
22
23    /*       FIFO-Control_Module  */
24    FIFO_Control #(depth) FC (.clk, .reset, .read(read), .write(write), .wr_en, .empty,
25                              .full, .readAddr(r_addr), .writeAddr(w_addr));
26
27 endmodule
```

```
29 module FIFO_testbench();
30
31    parameter depth = 4, width = 8;
32
33    logic clk, reset;
34    logic read, write;
35    logic [width-1:0] inputBus;
36    logic empty, full;
37    logic [width-1:0] outputBus;
38
39    FIFO #(depth, width) dut (.*);
40
41    parameter CLK_Period = 100;
42
43    initial begin
44       clk <= 1'b0;
45       forever #(CLK_Period/2) clk <= ~clk;
46    end
47
48
49    integer i;
50
51    initial begin
52
53       reset <= 1;                                    @(posedge clk);
54       reset <= 0;                                    @(posedge clk);
55       for (i = 0; i < 18; i++) begin
56          read <= 0; write <= 1;
57          inputBus <= i;                              @(posedge clk); // test normal w/full/upper boundary
58       end
```

```
58       end
59       read <= 1; write <= 1;
60       inputBus <= 8'b11111111;                       @(posedge clk); // test w&r
61       read <= 1; write <= 0;
62       inputBus <= 8'b00000000;          repeat(16)   @(posedge clk); // test normal r
63       read <= 0; write <= 0;                         @(posedge clk); // test disenabling
64       reset <= 1;                                    @(posedge clk);
65       reset <= 0; read <= 1;            repeat(2)    @(posedge clk);
66       $stop;
67    end
68 endmodule
```

iii. (ram_2port.sv)

```
1  // this is a 2 port ram that works as a queue
2  // when it reads, it outputs the oldest data
3  // stored in r_addr to r_data, when it writes,
4  // it update the value stored in w_addr with
5  // w_data
6  module ram_2port #(parameter depth = 16,
7                     parameter width = 8)
8                    (clk, wr_en, rd_en, w_addr, r_addr, w_data, r_data);
9
10     input logic clk, wr_en, rd_en;
11     input logic [3:0] w_addr, r_addr;
12     input logic [7:0] w_data;
13
14     output logic [width-1:0] r_data;
15
16     logic [width-1:0] RAM [depth-1:0];
17
18     // the following defines the read and write operations
19     always_ff @(posedge clk) begin
20        if (wr_en & rd_en) begin
21           RAM[w_addr] <= w_data;
22           r_data <= RAM[r_addr];
23        end else if (wr_en & ~rd_en)begin
24           RAM[w_addr] <= w_data;
25           r_data <= r_data;
26        end else if (~wr_en & rd_en) begin
27           r_data <= RAM[r_addr];
28        end else if (~wr_en & ~rd_en) begin
29           r_data <= r_data;
30        end
31     end
```

```
34 module ram_2port_testbench();
35    logic clk, wr_en, rd_en;
36    logic [3:0] w_addr, r_addr;
37    logic [7:0] w_data;
38
39    logic [7:0] r_data;
40
41    // Set up a simulated clock.
42    parameter CLOCK_PERIOD=100;
43
44    initial begin
45       clk <= 0;
46       forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
47    end
48
49    ram_2port ram (.*);
50
51    integer i;
52
53    initial begin
54                                          @(posedge clk);
55       wr_en <= 1'b1; rd_en <= 1'b1;    // test if enable updates stored data and ouput
56       for (i = 0; i < 15; i++) begin   // expect to see r_data decrement 2 cycles
57          w_addr <= i + 1;              // after w_data
58          r_addr <= i;
59          w_data <= 255-i;                 @(posedge clk);
60       end
61       wr_en <= 1'b0; rd_en <= 1'b0;    // test if disenable stops updates
62       for (i = 0; i < 4; i++) begin    // expect to see r_data shows previous value
63          w_addr <= i;
64          r_addr <= i;
```

```
65             w_data <= i;                   @(posedge clk);
66       end
67       wr_en <= 1'b0; rd_en <= 1'b1;    // test if disenable wr_en stops updating write
68       for (i = 0; i < 4; i++) begin    // and rd_en updates output
69          w_addr <= i;                  // expect to see r_data with value in part1
70          r_addr <= i;
71          w_data <= i;                   @(posedge clk);
72       end
73       wr_en <= 1'b1; rd_en <= 1'b0;    // knowning wr_en can update stored data
74       for (i = 0; i < 4; i++) begin    // test if disenable rd_en stops updating output
75          w_addr <= i + 1;              // expect to see r_data shows previous value
76          r_addr <= i;
77          w_data <= i + 1;               @(posedge clk);
78       end
79       $stop;
80    end
81 endmodule
```

iv. (task3.sv)

```
// Cynthia Li
// EE 371 LAB2 TASK3

// This is the top module that uses a FIFO that uses a 16x8 2 port ram to store data,
// and operates like Queue. It displays the "queue's" full/empty status on LEDR9/8,
// and when read(KEY[1]) is enabled, it reads the oldest data we input and displays
// it on HEX1 and HEX0, when write(KEY[0]) is enabled, it writes the specified data
// (SW[7:0]) to the newest address, and display this value on HEX5 and HEX4.
module task3(CLOCK_50, SW, KEY, LEDR, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input logic CLOCK_50;
   input logic [9:0] SW;
   input logic [3:0] KEY;
   output logic [9:0] LEDR;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   logic empty, full;
   logic [7:0] outputBus;

   // the following produces a slowed clock
   parameter whichClock = 25; // 0.365 Hz clock
   logic [31:0] div_clk;
   clock_divider cdiv (.clock(CLOCK_50),
                       .reset(reset),
                       .divided_clocks(div_clk));

   // clk; allows for easy switching between simulation and board clocks
   logic clk;
   // assign clk = CLOCK_50; // for simulation
   assign clk = div_clk[whichClock]; // for board

   logic r, w, read, write;
   meta READ (.clk(CLOCK_50), .in(SW[9]), .out(r));
   meta WRITE (.clk(CLOCK_50), .in(SW[8]), .out(w));
   userInput R (.clk(CLOCK_50), .press(SW[9]), .out(read));
   userInput W (.clk(CLOCK_50), .press(SW[8]), .out(write));

   //FIFO #(4, 8) fifo (.clk(CLOCK_50), .reset(SW[9]), .read(read), .write(write),
            //  .inputBus(SW[7:0]), .empty, .full, .outputBus);

   FIFO #(4, 8) fifo (.clk(CLOCK_50), .reset(SW[9]), .read(read), .write(write),
              .inputBus(SW[7:0]), .empty, .full, .outputBus);

   // the following displays the full and empty status of FIFO on LEDR[9] and LEDR[8]
   assign LEDR[9] = full;
   assign LEDR[8] = empty;

   assign HEX3 = 7'b1111111;
   assign HEX2 = 7'b1111111;

   // the following displays hexadecimal value of current input value on HEX5-4
   hexadecimal writeData1 (.binary(SW[7:4]), .hex(HEX5));
   hexadecimal writeData2 (.binary(SW[3:0]), .hex(HEX4));

   // the following displays current value of the oldest data on HEX1-0
   hexadecimal readData1 (.binary(outputBus[7:4]), .hex(HEX1));
   hexadecimal readData2 (.binary(outputBus[3:0]), .hex(HEX0));
endmodule

module task3_testbench();
   logic CLOCK_50;
   logic [9:0] SW;
   logic [3:0] KEY;
   logic [9:0] LEDR;
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   task3 dut (.*);

   parameter CLK_Period = 100;

   initial begin
      CLOCK_50 <= 1'b0;
      forever #(CLK_Period/2) CLOCK_50 <= ~CLOCK_50;
   end

   logic clk;
   assign clk = CLOCK_50;

   integer i;

   initial begin
                                                    @(posedge clk);
      SW[9] <= 1;                                   @(posedge clk);
      SW[9] <= 0;                                   @(posedge clk);
      KEY[1] <= 1; KEY[0] <= 0;                     @(posedge clk);
                                                    @(posedge clk);
                                                    @(posedge clk);
      for (i = 0; i < 18; i++) begin
```

```
88
89          SW[7:0] <= i;                               @(posedge clk); // test normal w/full/upper boundary
90      end
91
92      KEY[1] <= 0; KEY[0] <= 1;
93      SW[7:0] <= 8'b00000000;       repeat(16)  @(posedge clk); // test normal r
94      KEY[1] <= 1; KEY[0] <= 1;                       @(posedge clk); // test disenabling
95      SW[9] <= 1;                                     @(posedge clk);
96      SW[9] <= 0; KEY[1] <= 1;            repeat(2)   @(posedge clk);
97
98      $stop;
99  end
100 endmodule
```