Garrett Tashiro

EE 371

October 20, 2021

Lab 2 Report

Procedure

This lab comprised of three tasks: the first task was to design and implement a single port RAM unit that provides the address and the data associated with that address. The single port RAM could display an address, data wanting to be written, and data that was already written. The second task was to create a dual port RAM unit. The first port was for reading addresses and the data with it, and the second port was for writing data to an address. To create the 32x4 RAM, we used the IP catalog to create our Verilog file for the dual port RAM. The third task was to design a FIFO. The FIFO utilized an FSM to act like a queue and hold data that was written to the RAM. You can write to the RAM, and read from it as well. This lab gave an understanding about how single port RAM, dual port RAM, and the FIFO for a dual port RAM.

Task #1

The first task we were given was to design a single port RAM unit in SystemVerilog, and implement it on the DE1_SoC board. The way I approached this first task was I first read over the whole task at hand. I referred to Figure 1a on the lab doc to get an understanding of how the RAM module would be implemented. This task introduced a multidimensional array for out 32x4 memory module that is 32 words that are 4-bits long. Since in this task we were asked to make a single port RAM module, I knew I needed to have an input for address, input/output for data, and an input write enable. I wrote the code in an always_ff block that updated on the positive clock edge, and had read data be set to the data in the multidimensional memory array at the given address. If write enable was true, then the memory array would update that memory location with the new data that was being written. This can be seen in Figure 1 below.

```
//Create a 2-D array that is depth by width (32x4).
logic [width - 1:0] memory_array [depth - 1:0];

//always_ff updates on the positive clock edge. If wr_en is high
//memory_array at the current address will be written over with
//wr_data. re_data will always display the data at wr_adrr in
//memory_arry.
always_ff @(posedge clk) begin
   if(wr_en) begin
      memory_array[wr_addr] <= wr_data;
   end

   re_data <= memory_array[wr_addr];
end
```

Fig. 1: Multidimensional Memory Array with Read and Write

Since I knew I needed to display the data that was store in the memory, the data being written to memory, and the address in which the data is being written to, I created a HEX display module that had three input ports: dataIn, dataOut, and addr. These three inputs were the data that was being written to memory, the data at an address in memory, and the address in memory. I first copied over the localparams I made from the last lab for convenience. From there, I broke the address into two parts: upper and lower. Since hexadecimal only takes 4-bits (upto 15 in decimal) and the address being passed was 5-bits, I took bits 0-3 as lower, and bit 4 as upper. This allowed me to count from 0 to 31 and display the output on two HEX display. Figure 2 shows how I did this below.

```
//Creating two 4-bit logics to hold the upper and lower
//bits of addr to make two always_comb blocks
logic upper;
logic [3:0] lower;

//Assigning lower to bits 0-3 of addr for hex output
assign lower = addr[3:0];
//Assigning upper to bit 4 of addr for hex output
assign upper = addr[4];
```

Fig. 2: Upper and Lower Bits of addr

This was the best, and easiest way I could think of to display a max of 31 onto two separate HEX displays. I used four always_comb blocks to with case statements to set the HEX displays to the correct output for each input. This way of setting values to the HEX displays was similar to what I did in previous labs. The code for hexDisplays module can be seen in Appendix 1.C. After I had the module for the RAM and the HEX display module, I was able to use hierarchical calls to both modules in DE1_Soc module and implement the single port RAM on the board.

Task #2

For the second task, we had to implement a dual port RAM unit in SystemVerilog and use the IP Catalog in Quartus to create a dual port RAM module. How I did this was I first read over everything that was being asked in the task. I followed the instructions that were in the lab document on how to create the dual port RAM module through the IP Catalog and created the ram32x4.v module that was the code for the dual port RAM module. Following the steps for the IP Catalog I also created a .mif file in which I was able to set the data at each memory address in the RAM unit. This dual port RAM was similar to the single port RAM that we created in task one, but this module would read from addresses while you could write to them. This task asked us to display an address in which the module was reading from along with the data at that address on HEX displays 0, 2, and 3 for roughly one second, while on HEX displays 1, 4, and 5 you had to display the data that would, or could, be written and the address you were going to write it at and these would update every clock cycle.

Since I needed to have an address and data update every clock cycle, as well as an address and the data associated with it update roughly every second, I decided to use a clock divider. We used these in 271 a few times, and I still had the file for it from my 271 class. I used CLOCK_50 in the DE1_SoC and passed CLOCK_50 to every module except a module I called counter, and the modules that didn't need a clock, such as the HEX display module. CLOCK_50 is a 50MHz clock, and using the clock divider I was able to get a clock rate of 0.75Hz. The clock divider can be seen below in Figure 3, and the full code for this module can be found in Appendix 2.F.

```
// 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
```

Fig. 3: clock_divider Module

With the clock rate divided to allow the address and data reads from the memory to be about one second, I moved onto the counter. The counter module is the only module that directly used the lower clock rate. The counter module had an output read_out that was for the memory address. Using the slower clock speed, I had an always_ff that incremented read_out by one at every positive clock edge. This updated the address that was used for reading the data in memory roughly every second. The always_ff using the slower clock rate can be seen below in Figure 4.

```
//always_ff block to update on the positive clock edge. This block
//checks if 1-bit reset is high and if it is then it zeros out
//read_out, and if reset is not high then it will add 1 to read_out.
always_ff @(posedge clk_divide) begin
   if(reset) begin
      read_out <= '0;
   end

   else begin
      read_out <= read_out + 1;
   end
end
```

Fig. 4: The Counter Modules always_ff Updating the Address Read Roughly Every Second

Once I had the addresses and data associated to that address updating roughly every second, I utilized the hexDisplay module that was written in task 1 and modified it slightly for this task so I could show two addresses, as well as two sets of data. I knew I needed to have two DFF's in series as well for the inputs from the user, so I tweaked my doubleD module to be a parameterized module in order to take in more than a single bit at a time. After that, I put everything into the DE1_SoC top module and was able to test everything together in a testbench then run it on the board. The code for DE1_SoC module can be found in Appendix 2.I.

Task #3

For the third task we had to design a FIFO memory system. A FIFO is a first in, first out system that implements a queue. The FIFO can be written to if the queue is not full, and can read from if the queue is not empty. The data store will be read in the order in which it was stored in. The way I approached this task was I first read over the task in the lab document. I also referred to the lecture slides as well in order to get a better understanding of the FIFO. I used the FIFO skeleton code that was already written for us, and I knew that I needed the FIFO controller, as well as a dual port RAM module that was 16x8. I first started on the FIFO controller, and the lecture slides hinted towards using an FSM in order to implement the queue for the FIFO. I didn't know exactly where to start with the FSM since I was having troubles visualizing how it would be written in code. In Java we created queues from scratch and implemented it by using an array. I was stuck on the array aspect for a bit until I realized the outputs from FIFO_Control were the arrays being used to keep track of positions of the read, and write pointer. With that out of the way, I made my FSM for the FIFO_Control module which can is shown in Figure 5 below.

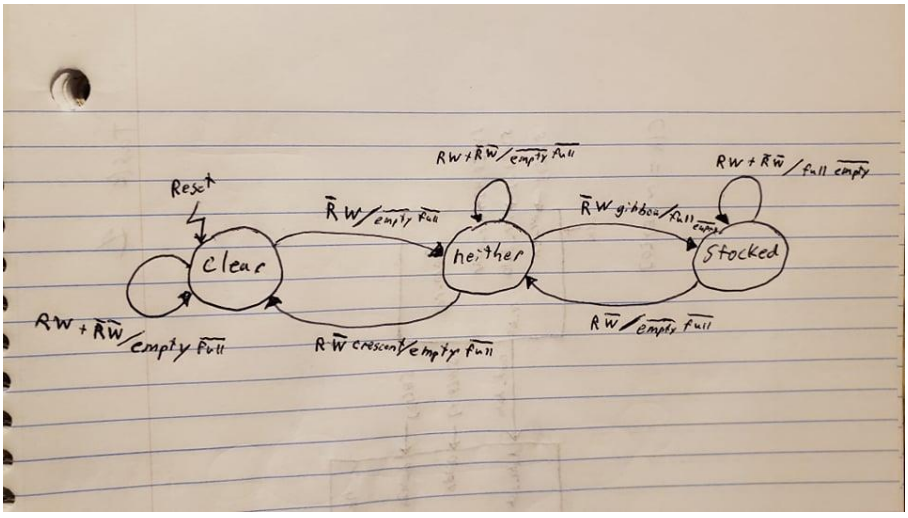

Fig. 5: Finite State Machine for FIFO_Control Module

The FSM has three states: clear, neither, and stocked. When in state clear, the FIFO is empty, and there is no data in memory, and empty is set to 1. If you are to try and read while in this state, nothing will happen, and

you won't transition to the neither state. The only way to transfer to the neither state is to write to memory. Once in the neither state, you can only leave it in one of two ways: you read without writing and crescent is 1, or you write without reading when gibbous is 1. The logic crescent is high if readAddr is one behind writeAddr. This is essentially a pointer setup to tell if you only have on piece of data left in the queue. The logic gibbous is the same as crescent, except it is to tell if writeAddr is one behind readAddr, so there is only one spot left in the queue before it is full. This is shown below in Figure 6.

```
//logic to be pointers for queue. Moon phase names.
//crescent: close to empty
//gibbous: close to full
logic crescent, gibbous;

//crescent points to one place behind of readAddr
//gibbous points to one place behind of writeAddr
assign crescent = (writeAddr == readAddr + 4'b0001);
assign gibbous = (readAddr == writeAddr + 4'b0001);
```

Fig. 6: Logic for Crescent and Gibbous

If you are to write without reading while in neither and gibbous is 1, you will go to the stocked state. In the stocked state full will be set to 1, and the only way to leave the stocked state is to read without a write.

Once FIFO and FIFO_Control modules were done I was able to move onto the HEX display module. The HEX display module was similar to both task 1 and task 2, so I copied it over and made a few modifications so that only two sets of data were being shown on the HEX displays, both being 8-bits in length and using two HEX displays. With all of those modules done, I was almost ready to put everything together in the DE1_SoC module. I knew I needed to have the input for read and write only be read once per button push, so I went back through my 271 labs to get an input buffer module. This module only allows the press from the buttons to be high for one clock cycle, and this was to prevent the system from constantly reading/writing to/from the memory with a 50MHz clock. Each press of the button would only allow one read, or one write to happen. After getting the buffer module, I was able to copy over the paramDFF module that I used in task 2 to prevent metastability from my inputs, and put everything together in DE1_SoC. The full code for DE1_SoC can be seen in Appendix 3.L.

Top-Level Block Diagrams

Task #1

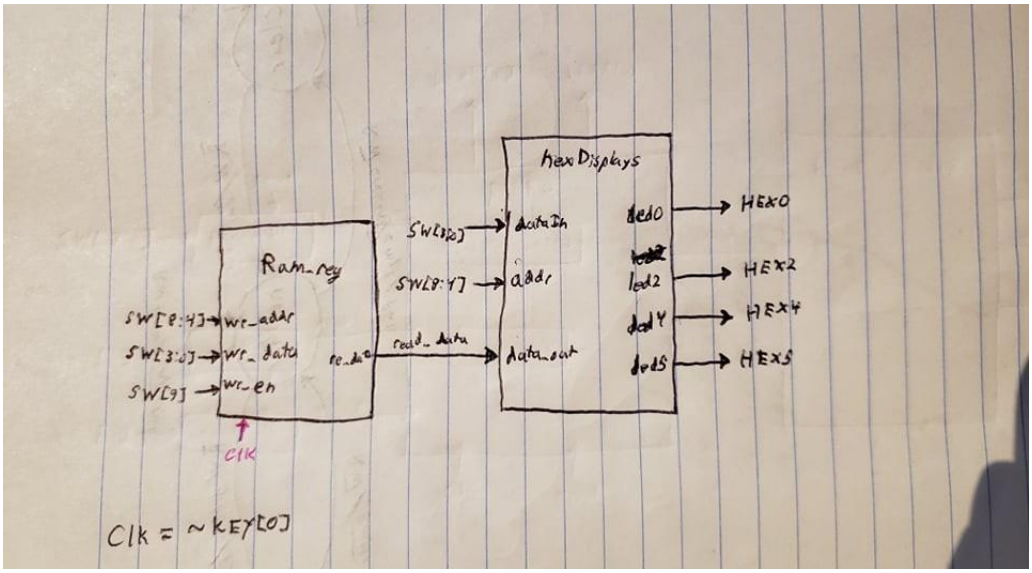

Fig. 7: DE1_SoC Block Diagram for Task 1

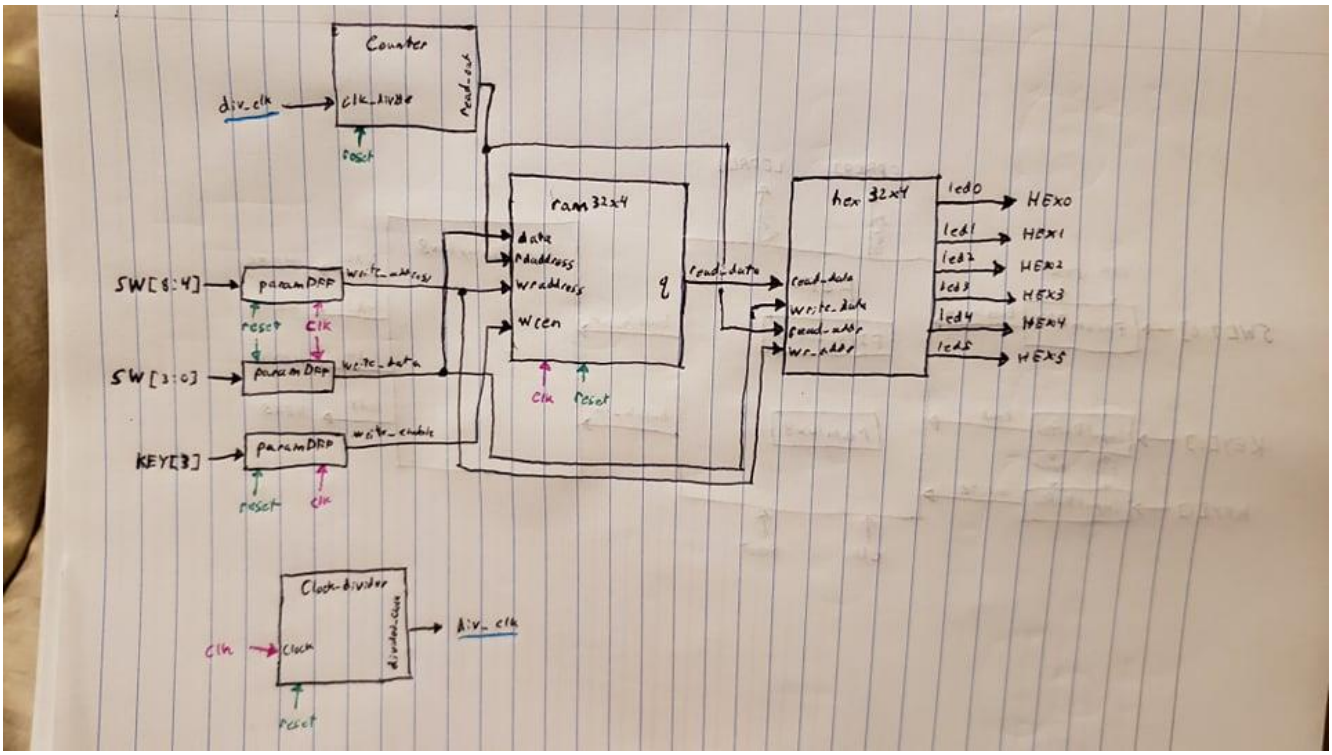

Fig. 8: DE1_SoC Block Diagram for Task 2

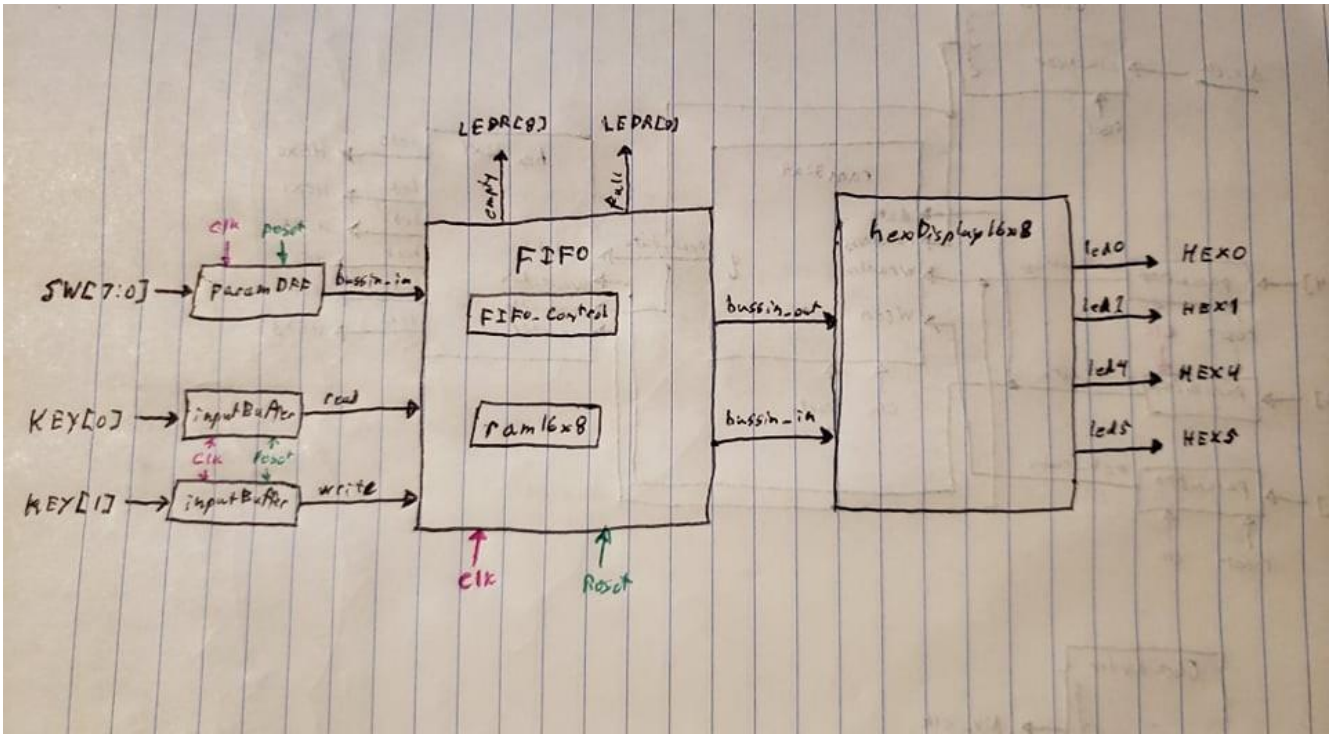

Fig. 9: DE1_SoC Block Diagram for Task 3

Results

Task #1

The first testbench I made for task 1 was for the RAM_reg module. I started the testbench by setting wr_en high, and then writing five pieces of data to five different addresses. I then set write enable low, and then checked those same addresses for the output as if I was reading them. After that, I set wr_en high again and I wrote over two addresses that I wrote data to initially. I then set wr_en low and checked those two addresses to see if updated accordingly. The results were turned out as expected. With wr_en high I was able to write data to each address, and when I went back to those addresses, I could read that data that was written prior. I also was not able to write data to an address if wr_en was low. This can be seen below in Figure 10. The code for the RAM_reg module testbench can be found in Appendix 1.B.

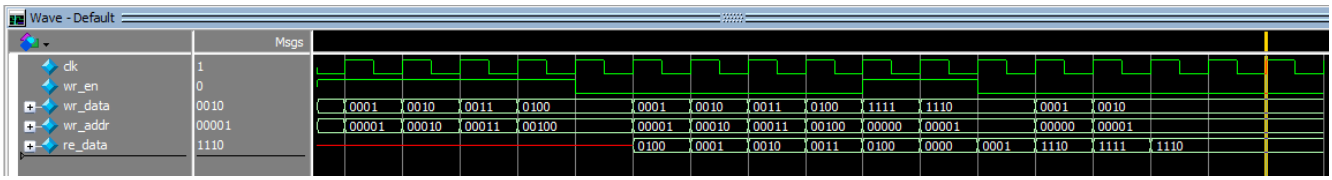

Fig. 10: RAM_reg ModelSim Simulation

Next, I made the hexDisplays module. For the testbench for this module I checked to see if HEX displays would update correctly when I changed the inputs dataIn, dataOut, and addr. I tested each one three times with different values. The results from this test were as expected. Each of the HEX displays updated accordingly. This can be seen below in Figure 11. The code for the testbench for the hexDisplays module can be found in Appendix 1.D.

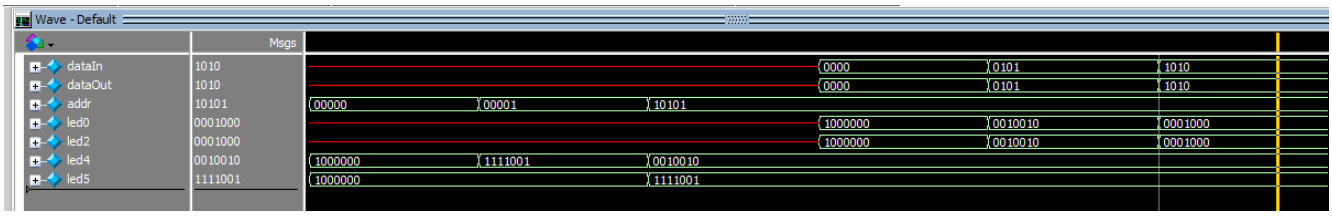

Fig. 11: hexDisplays ModelSim Simulation

The last testbench I created for task 1 was for DE1_SoC. I first set an address and had write not enabled. After that I set write high and had data written to the first address. I then set write low for a clock cycle. I set a new address, and wrote to this address. I then set write low. After that, I checked both addresses to see if the data I wrote was in them. The results from this test were as expected. With write enable, it will write to an address, and you can go back to that address later and the last thing you wrote to it will still be there. This can be seen below in Figure 12.

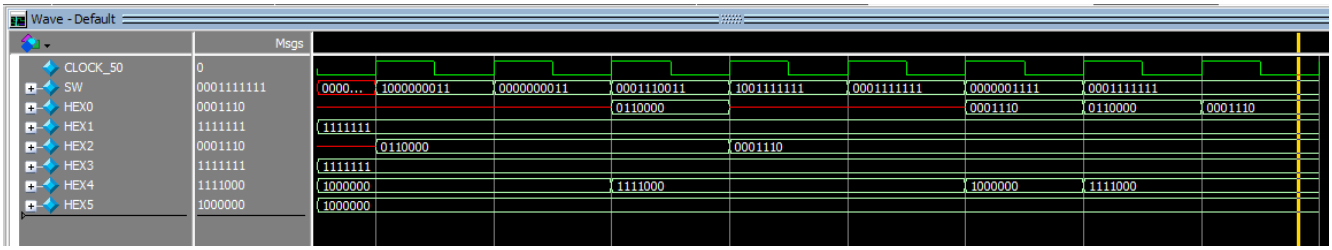

Fig. 12: DE1_SoC ModelSim Simulation

Task #2

The first testbench I started with in task 2 was testbench for hex32x4 module. I ran a similar test to the testbench for hexDisplays in task 1 since they are essentially the same. I had both 5-bit inputs set to three different values, and then I had the 4-bit inputs set to three different values to check and make sure the HEX displays were updating correctly. The results from this test were as expected, and the HEX displays updated correctly, and displayed the correct things. This can be seen below in Figure 13.

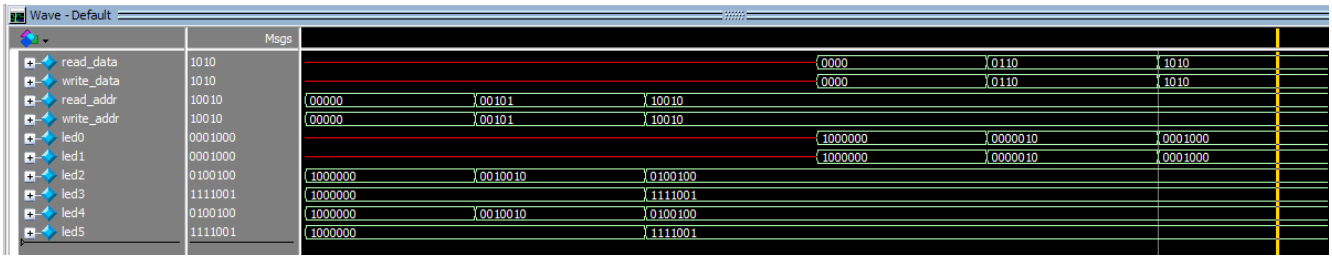

Fig. 13: hex32x4 ModelSim Simulation

Next, I worked on the counter and its testbench. For the testbench for the counter I set reset high, then low, and then I let 33 clock cycles pass. After that, I reset again, and let five clock cycles pass. The results were as expected. The counter did not count past 31, and after 31 it went to 0. If reset happens, then the counter will go back to 0. This can be seen in Figure 14, and a zoomed in portion in Figure 15 down below.

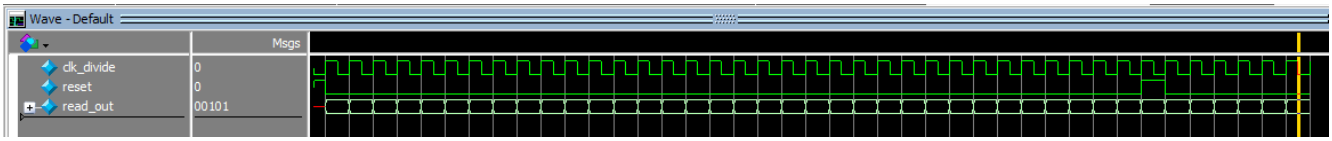

Fig. 14 counter ModelSim Simulation

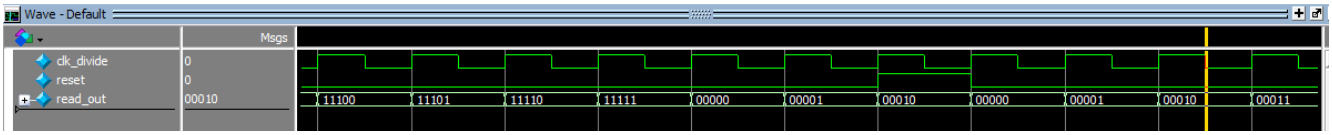

Fig. 15: Zoomed in ModelSim Simulation of counter

After the counter I moved onto paramDFF module, which is two DFF's in series. For this testbench, I set press to four different values to see if out would update correctly. The results from this test were as expected, out updated to the correct value, and did it after two clock cycles as well. This can be seen in Figure 16 below.

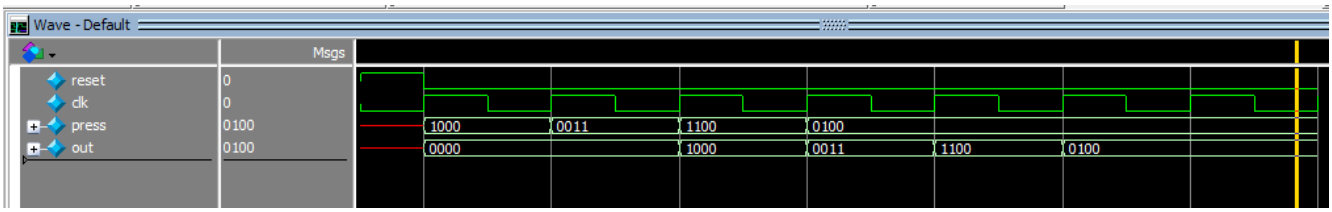

Fig. 16: ModelSim Simulation for Module paramDFF

From there I moved onto clock_divider. The clock_divider testbench is very simple. Reset is set high and then low, and then 100 clock cycles pass. When the div_clk array is full, the that will count as one clock cycle. Without this clock_divider I wouldn't have been able to display memory addresses and the associated data using CLOCK_50. The testbench for this can be seen below in Figure 17.

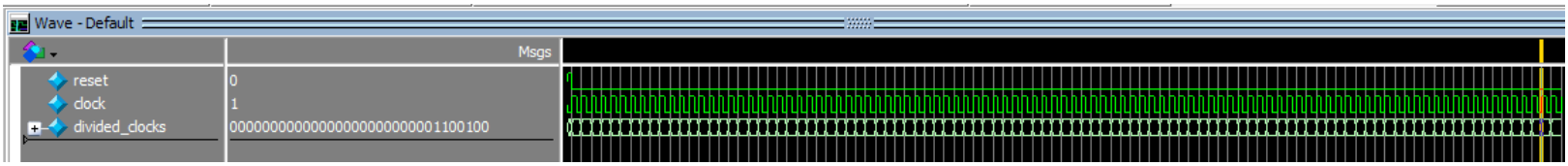

Fig. 17: The clock_divider Module ModelSim Simulation

The last module I worked on in task 2 was DE1_SoC. The testbench for DE1_SoC sets the address to 5'd0, and the data to 4'd0. Reset is set high and then low, and then 33 clock cycles pass. After 33 clock cycles, write enable is set low (which is high since it is a KEY), and data is written to an address. Write enable is then set high (which is low since it is a KEY) and I tried writing to data to another address. The results from this test were as expected, during the 33 cycles, the HEX displays connected to the counter and one second clock updated correctly up to memory address 31, and then went back to memory address 0. I was about to write to an address when writing was enabled, see the change, and then not wrote to another address when writing wasn't enabled. This can be seen below in Figure 18.

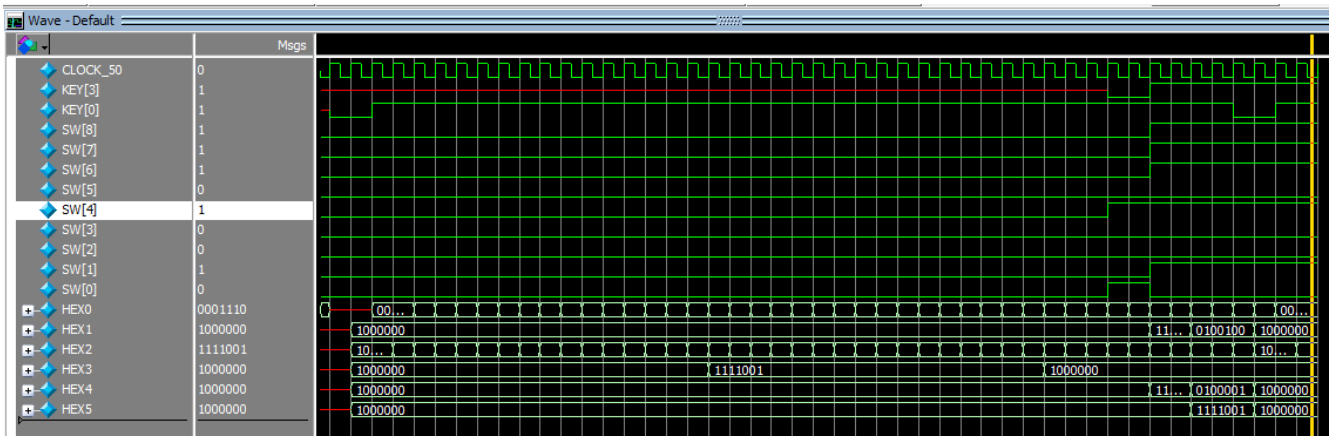

Fig. 18: DE1_SoC Testbench Simulation on ModelSim

Task 3

The first testbench I made for task 3 was for FIFO_Control. In this testbench I was testing for expected, unexpected, and edge case behavior. I first had read and write set to 0, and then reset. After that, I had write go high for a clock cycle, then go low for a clock cycle. This was repeated 17 times to see if the queue was working properly. Next, I had read go high for a clock cycle, and then go low for a clock cycle. This was repeated 18 times to see if the queue was working properly. I finished off the testbench by "writing" once and resetting to see if the queue would become empty. The results from this were as expected. You cannot write more than the queue allowed, and once you are at the max amount in the queue, full would go high. You cannot read more than what is in the queue. Once everything is read from the queue, empty will go high. Resetting does empty the queue. This can be seen below in Figure 19.

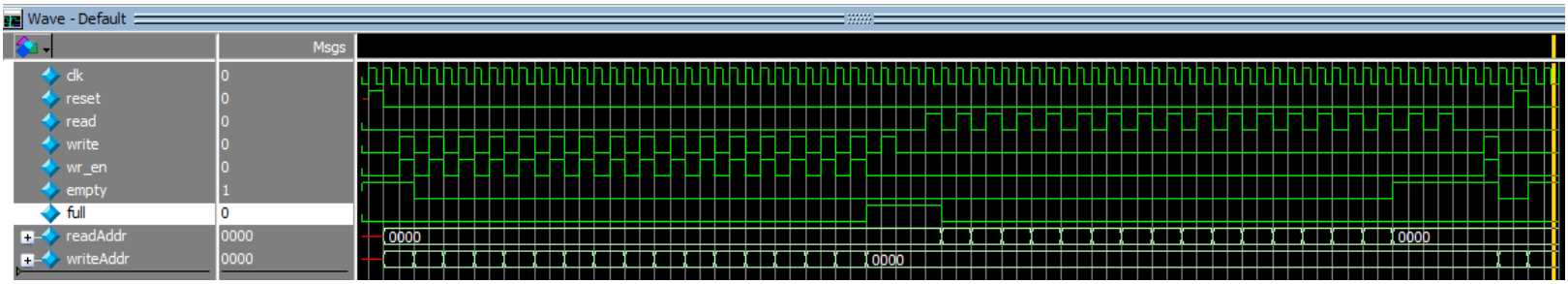

Fig. 19: FIFO_Control ModelSim Simulation

The next module I worked on for task 3 was FIFO. For the testbench for FIFO I was testing to see if you could read more than what was in the queue, as well as write more than there was space in the queue. Similar to the testbench in FIFO_Control. Reset was set high then low, and then I used a for loop with write set to 1 inside and inputBus set to i, and I had i < 17. I did the same thing with read and outputBus to see if empty and full would go high. The results from this test were as expected. Both empty and full went high at the correct times. This can be seen in Figure 20 below.

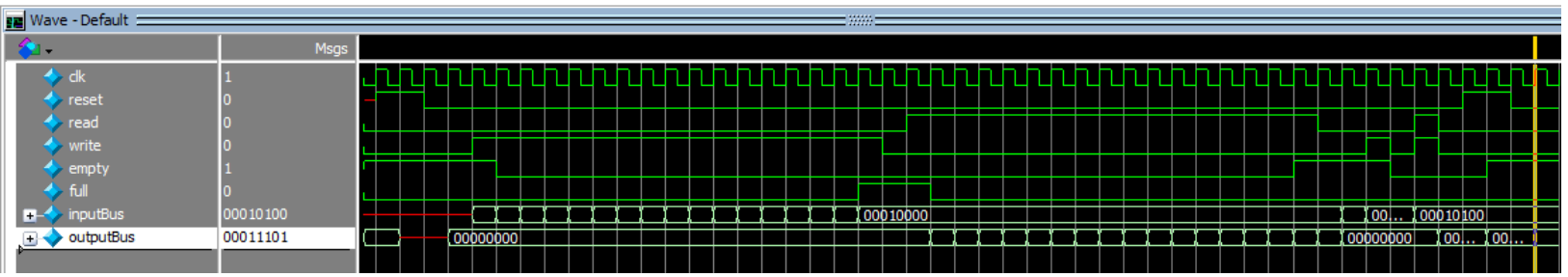

Fig. 20: Simulated Testbench for FIFO Module

After I wrote FIFO module, I moved onto hexDisplay16x8 module. This is essentially the exact same HEX display module as in task 1 and task 2. In the testbench for this module I set dataInput and dataOutput to four different values. The results from this test were as expected. The HEX displays were updating accordingly. The testbench for this can be seen below in Figure 21.

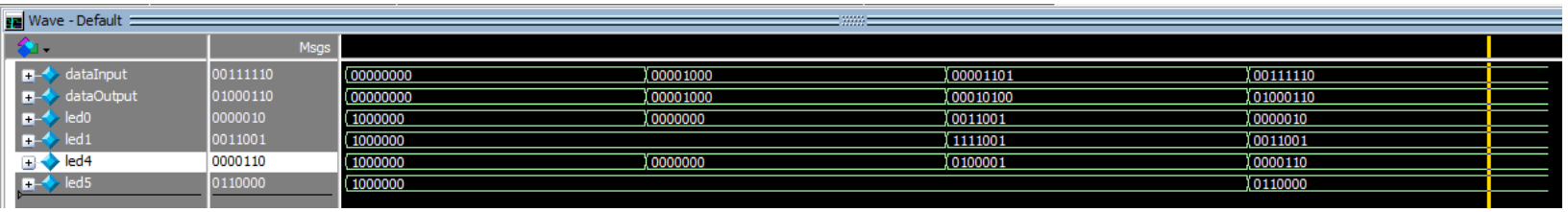

Fig. 21: Simulated Testbench for hexDisplay16x8 Module

Once I had the hexDisplay16x8 module done, I then brought in the paramDFF module. This is the exact same paramDFF module that was used in task 2. For this testbench, I set press to four different values to see if out would update correctly. The results from this test were as expected, out updated to the correct value, and did it after two clock cycles as well. This can be seen in Figure 22 below.

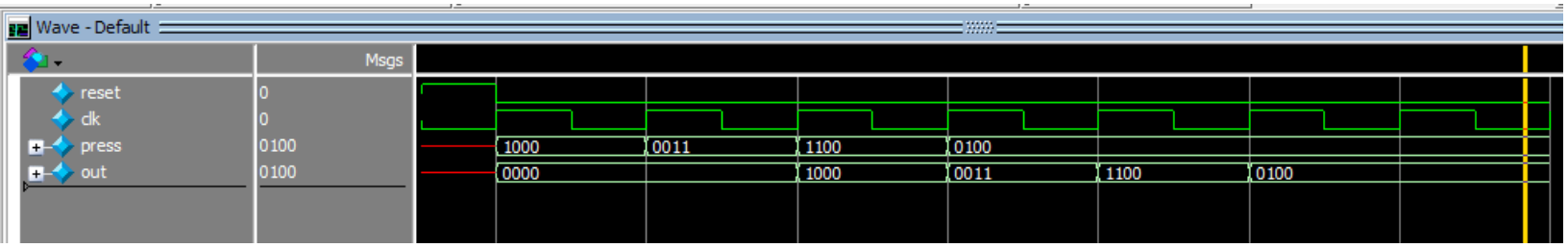

Fig. 22: ModelSim Simulation for Module paramDFF

Next, I moved onto the inputBuffer module. This module was a simple one, and one that was used in 271. For the testbench I set reset to high, and then low. After that I set press high for two clock cycles and observed out. The results from this test were as expected. Out can only be high for 1 clock cycle, and press was high for two clock cycles. The output out went low during the second clock cycle. This can be seen in Figure 23 below.

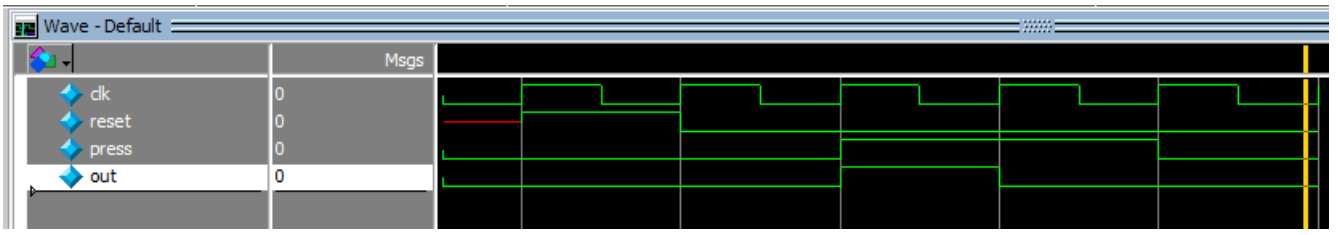

Fig. 23: ModelSim Simulation for inputBuffer

The last module I test was DE1_SoC. I had a similar testbench for DE1_SoC as I did with FIFO. I used for loops to see if you could write once the queue was full, and if you could read once the queue was empty. The results from these tests were as expected. Trying to write to the queue when it is full will do nothing, and if you try and read from the queue when it is full it will do nothing. This showed that the queue is working properly, and that the HEX displays are as well. This can be seen below in Figure 24.

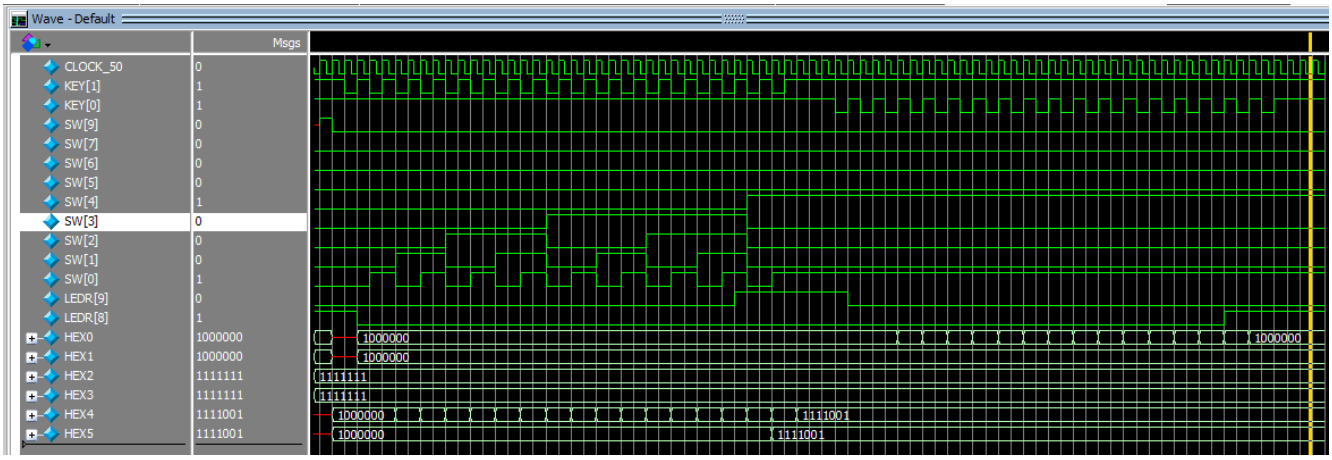

Fig. 24: DE1_SoC ModelSim Simulation

Final Project

The goal of this lab was to learn about RAM units and how to implement them on the DE1_SoC board by using SystemVerilog. In this lab we learned about single port, and dual port RAM. We also learned about FIFO, and how FIFO works with dual port RAM. The FIFO is a queue for the dual port RAM. The most challenging part of this lab for me was figuring out how to design the FSM for FIFO_Control. That part of the lab made me really think about what was needed in order for the system to work. I had never written a queue in SystemVerilog prior to this lab, and I also had never used the IP Catalog. I didn't even know the IP Catalog even existed.

In the end, I was able to produce the results that I wanted and I believe are sufficient and cover the requirements for lab 2.

Appendix

1.A) RAM_reg.sv

```
1  //Garrett Tashiro
2  //October 17, 2021
3  //EE 371
4  //Lab 2, Task 1.1
5
6  //RAM_reg has 1-bit clk, wr_en, 4-bit wr_data, and 5-bit wr_addr as inputs
7  //and returns 4-bit re_data as an output. This module is a parameterized
8  //module with two parameters that are used in the 2-D memory_array to create
9  //a single port 32x4 module. When the 1-bit input wr_en is high the memory_array
10 //at wr_addr will update to the value of wr_data on the positive clock edge.
11 //re_data always reads from memory_array at wr_addr on the positive edge of the clock.
12 module RAM_reg #(parameter depth = 32, width = 4)(clk, wr_en, wr_data, wr_addr, re_data);
13       input logic          clk, wr_en;  //wr = wr_en, en = enable
14       input logic [3:0]    wr_data;
15       input logic [4:0]    wr_addr;
16       output logic [3:0]   re_data;  //re = read
17
18       //Create a 2-D array that is depth by width (32x4).
19       logic [width - 1:0] memory_array [depth - 1:0];
20
21       //always_ff updates on the positive clock edge. If wr_en is high
22       //memory_array at the current address will be written over with
23       //wr_data. re_data will always display the data at wr_adrr in
24       //memory_arry.
25       always_ff @(posedge clk) begin
26          if(wr_en) begin
27             memory_array[wr_addr] <= wr_data;
28          end
29
30          re_data <= memory_array[wr_addr];
31       end
32 endmodule
33
```

1.B) RAM_reg.sv (testbench)

```
34 //RAM_reg_testbench tests for expected and unexpected behavior
35 //and the RAM_reg module. wr_en is initially high and 5 random
36 //data values are written to 5 different addresses. The next five
37 //tests are with wr_en low, to show that new data will be written
38 //to the memory addresses, and to show that data and be read at
39 //those addresses. Two tests to write over previously written
40 //addresses, and two tests to check if the data in those addresses
41 //were written over correctly.
42 module RAM_reg_testbench();
43       logic          clk, wr_en;
44       logic [3:0]    wr_data;
45       logic [4:0]    wr_addr;
46       logic [3:0]    re_data;
47
48       RAM_reg dut(.clk, .wr_en, .wr_data, .wr_addr, .re_data);
49
50       parameter clk_PERIOD = 100;
51       initial begin
52             clk <= 0;
53             forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
54       end
55
56       initial begin
57          wr_en <= 1;
58          wr_data <= 4'b0000; wr_addr <= 5'b00000;        repeat(1)   @(posedge    clk);
59          wr_data <= 4'b0001; wr_addr <= 5'b00001;        repeat(1)   @(posedge    clk);
60          wr_data <= 4'b0010; wr_addr <= 5'b00010;        repeat(1)   @(posedge    clk);
61          wr_data <= 4'b0011; wr_addr <= 5'b00011;        repeat(1)   @(posedge    clk);
62          wr_data <= 4'b0100; wr_addr <= 5'b00100;        repeat(1)   @(posedge    clk);
63          wr_en <= 0;                                     repeat(1)   @(posedge    clk);
64          wr_data <= 4'b0001; wr_addr <= 5'b00001;        repeat(1)   @(posedge    clk);
65          wr_data <= 4'b0010; wr_addr <= 5'b00010;        repeat(1)   @(posedge    clk);
66          wr_data <= 4'b0011; wr_addr <= 5'b00011;        repeat(1)   @(posedge    clk);
67          wr_data <= 4'b0100; wr_addr <= 5'b00100;        repeat(1)   @(posedge    clk);
68          wr_en <= 1;
69          wr_data <= 4'b1111; wr_addr <= 5'b00000;        repeat(1)   @(posedge    clk);
70          wr_data <= 4'b1110; wr_addr <= 5'b00001;        repeat(1)   @(posedge    clk);
71          wr_en <= 0;                                     repeat(1)   @(posedge    clk);
72          wr_data <= 4'b0001; wr_addr <= 5'b00000;        repeat(1)   @(posedge    clk);
73          wr_data <= 4'b0010; wr_addr <= 5'b00001;        repeat(1)   @(posedge    clk);
74                                                          repeat(3)   @(posedge    clk);
75          $stop; // End the simulation.
76       end
77 endmodule
```

1.C) hexDisplays.sv

```
1   //Garrett Tashiro
2   //October 17, 2021
3   //EE 371
4   //Lab 2, Task 1.2
5
6   //hexDisplays has 4-bit dataIn, dataOut, and 5-bit addr as inputs
7   //and returns 7-bit led0, led2, led4, and led5 as outputs. This
8   //module is designed to display the data that is in memory, getting
9   //written to memory, and the address to the HEX displays.
10  //each ledX goes the the corresponding HEX display number.
11  //HEX0 will display the data read out. HEX2 will display the data
12  //being input to memory. HEX4 and HEX5 will display the address.
13  module hexDisplays(dataIn, dataOut, addr, led0, led2, led4, led5);
14        input logic [3:0]    dataIn, dataOut;
15        input logic [4:0]    addr;
16        output logic [6:0]   led0, led2, led4, led5;
17
18
19        //localparams to hold 7-bits to logic for hex displays
20        localparam logic [6:0] zero = 7'b1000000;   //0
21        localparam logic [6:0] one = 7'b1111001;    //1
22        localparam logic [6:0] two = 7'b0100100;    //2
23        localparam logic [6:0] three = 7'b0110000;  //3
24        localparam logic [6:0] four = 7'b0011001;   //4
25        localparam logic [6:0] five = 7'b0010010;   //5
26        localparam logic [6:0] six = 7'b0000010;    //6
27        localparam logic [6:0] seven = 7'b1111000;  //7
28        localparam logic [6:0] eight = 7'b0000000;  //8
29        localparam logic [6:0] nine = 7'b0010000;   //9
30        localparam logic [6:0] A = 7'b0001000;      //A
31        localparam logic [6:0] B = 7'b0000011;      //b
32        localparam logic [6:0] C = 7'b1000110;      //C
33        localparam logic [6:0] D = 7'b0100001;      //d
34        localparam logic [6:0] E = 7'b0000110;      //E
35        localparam logic [6:0] F = 7'b0001110;      //F
36        localparam logic [6:0] blank = 7'b1111111;
37
38        //Creating two 4-bit logics to hold the upper and lower
39        //bits of addr to make two always_comb blocks
40        logic upper;
41        logic [3:0] lower;
42
43        //Assigning lower to bits 0-3 of addr for hex output
44        assign lower = addr[3:0];
45        //Assigning upper to bit 4 of addr for hex output
46        assign upper = addr[4];
47
48
```

```
50        //The case statement will set the output for HEX2 in hexidecimal for
51        //the data that is being input
52        always_comb begin
53           case(dataIn)
54
55              4'd0: begin
56                    led2 = zero;
57              end
58
59              4'd1: begin
60                    led2 = one;
61              end
62
63              4'd2: begin
64                    led2 = two;
65              end
66
67              4'd3: begin
68                    led2 = three;
69              end
70
71              4'd4: begin
72                    led2 = four;
73              end
74
75              4'd5: begin
76                    led2 = five;
77              end
78
79              4'd6: begin
80                    led2 = six;
81              end
82
83              4'd7: begin
84                    led2 = seven;
85              end
86
87              4'd8: begin
88                    led2 = eight;
89              end
90
91              4'd9: begin
92                    led2 = nine;
93              end
94
95              4'd10: begin
96                    led2 = A;
97              end
98
99              4'd11: begin
100                   led2 = B;
101             end
```

```
            end

            4'd12: begin
                  led2 = C;
            end

            4'd13: begin
                  led2 = D;
            end

            4'd14: begin
                  led2 = E;
            end

            4'd15: begin
                  led2 = F;
            end

            default: begin
                  led2 = 7'bx;
            end
         endcase
      end

      //always_comb for combinational logic with a case statement for dataOut.
      //The case statement will set the output for HEX0 in hexidecimal for
      //the data that is being read from an address location in the register.
      always_comb begin
         case(dataOut)

            4'd0: begin
                  led0 = zero;
            end

            4'd1: begin
                  led0 = one;
            end

            4'd2: begin
                  led0 = two;
            end

            4'd3: begin
                  led0 = three;
            end

            4'd4: begin
                  led0 = four;
            end

            4'd5: begin
                  led0 = five;
             end

            4'd6: begin
                  led0 = six;
             end

            4'd7: begin
                  led0 = seven;
             end

            4'd8: begin
                  led0 = eight;
             end

            4'd9: begin
                  led0 = nine;
             end

            4'd10: begin
                  led0 = A;
             end

            4'd11: begin
                  led0 = B;
             end

            4'd12: begin
                  led0 = C;
             end

            4'd13: begin
                  led0 = D;
             end

            4'd14: begin
                  led0 = E;
             end

            4'd15: begin
                  led0 = F;
             end

            default: begin
                  led0 = 7'bx;
             end
          endcase
       end
```

```
        //always_comb for combinational logic with a case statement for lower.
        //The case statement will set the output for HEX4 in hexidecimal for
        //the lower part of the memory address
        always_comb begin
           case(lower)

               4'd0: begin
                     led4 = zero;
               end

               4'd1: begin
                     led4 = one;
               end

               4'd2: begin
                     led4 = two;
               end

               4'd3: begin
                     led4 = three;
               end

               4'd4: begin
                     led4 = four;
               end

               4'd5: begin
                     led4 = five;
               end

               4'd6: begin
                     led4 = six;
               end

               4'd7: begin
                     led4 = seven;
               end

               4'd8: begin
                     led4 = eight;
               end

               4'd9: begin
                     led4 = nine;
               end

               4'd10: begin
                     led4 = A;
               end

               4'd11: begin
                     led4 = B;
```

```
               end

               4'd12: begin
                     led4 = C;
               end

               4'd13: begin
                     led4 = D;
               end

               4'd14: begin
                     led4 = E;
               end

               4'd15: begin
                     led4 = F;
               end

               default: begin
                     led4 = 7'bx;
               end
           endcase
        end

        //always_comb for combinational logic with a case statement for lower.
        //The case statement will set the output for HEX5 in hexidecimal for
        //the upper part of the memory address. Only going to 32, so this only
        //needs to check for the 5th bit being a 1 or 0.
        always_comb begin
           case(upper)

               1'd0: begin
                     led5 = zero;
               end

               1'd1: begin
                     led5 = one;
               end

               default: begin
                     led5 = 7'bx;
               end
           endcase
        end
    endmodule
```

1.D) hexDisplays.sv (testbench)

```
299   //hexDisplays_testbench tests for expect and unexpected
300   //behavior. The first three tests are to changing address.
301   //I tested 3 somewhat random inputs for address to make sure
302   //the HEX displays would update accordingly. The next three
303   //tests are on dataIn and dataOut. I chose 3 random numbers
304   //that were the same for the both of them since they are both
305   //4-bits and made sure the HEX displays updated accordingly.
306   module hexDisplays_testbench();
307         logic [3:0]    dataIn, dataOut;
308         logic [4:0]    addr;
309         logic [6:0]    led0, led2, led4, led5;
310
311         hexDisplays dut(.dataIn, .dataOut, .addr, .led0, .led2, .led4, .led5);
312
313         initial begin
314            addr = 5'd0;                        #10;
315            addr = 5'd1;                        #10;
316            addr = 5'd21;                       #10;
317
318            dataIn = 4'd0; dataOut = 4'd0;    #10;
319            dataIn = 4'd5; dataOut = 4'd5;    #10;
320            dataIn = 4'd10; dataOut = 4'd10; #10;
321         end
322   endmodule
```

1.E) DE1_SoC.sv

```
1     //Garrett Tashiro
2     //October 18, 2021
3     //EE 371
4     //Lab 2, Task 1.3
5
6     //DE1_SoC is the top level module. This module has 1-bit CLOCK_50,
7     //4-bit KEY (1-bit for each KEY), and 10-bit SW (1-bit for each sw 0-9)
8     //as inputs and returns 7-bit HEX0, HEX1, HEX2, HEX3, HEX4,  and HEX5
9     //as outputs. This module uses hierarchical calls to other modules in
10    //order to read and write from a memory array. The memory address is
11    //set with SW8-4, the data being written is set using SW3-0, SW9 is
12    //for enable to write data to an address. The address will be displayed
13    //on HEX5-4. The data being written will display on HEX2. The data
14    //read out of memory is on HEX0. This module is to implememnt a single
15    //port RAM reg.
16    module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, CLOCK_50);
17          input logic           CLOCK_50;
18          input logic [3:0]     KEY;
19          input logic [9:0]     SW;
20          output logic [6:0]    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
21
22          //Set HEX1 and HEX3 be blank
23          assign HEX1 = 7'b1111111;
24          assign HEX3 = 7'b1111111;
25
26          //4-bit logic to hold the output from re_data and pass it to
27          logic [3:0] read_data;
28
29
30          //1-bit logic to hold the output from KEY[0], or CLOCK_50.
31          //Similar setup to what was used in lab 5 in 271. When testing
32          //CLOCK_50 will be used, and when on the board output from
33          //KEY[0] will be used.
34          logic clk;
35          //assign clk = ~KEY[0];  //Uncomment for running code on the board
36          assign clk = CLOCK_50;  //Uncomment for simulation testbenches
37
38
39          //RAM_reg theReg takes 1-bit clk, enable, 4-bit SW[3:0], and 5-bit SW[8:4] as inputs
40          //and returns 4-bit read_data as an output. The output is passed to displayHEX.
41          RAM_reg theReg(.clk(clk),
42                         .wr_en(SW[9]),
43                         .wr_data(SW[3:0]),
44                         .wr_addr(SW[8:4]),
45                         .re_data(read_data));
46
47          //hexDisplays displayHEX has 4-bit read_data, SW[3:0], and 5-bit SW[8:4] as inputs
48          //and returns the corresponding information to HEX displays. The input from SW[3:0]
49          //is displayed on HEX2. The input from read_data is displayed on HEX0. The input from
50          //SW[8:4] is displayed on HEX4 and HEX5 since it is a 5-bit number and needs two
51          //displays to count up to 32 in hexidecimal.
52          hexDisplays displayHEX(.dataIn(SW[3:0]),
53                                 .dataOut(read_data),
54                                 .addr(SW[8:4]),
55                                 .led0(HEX0[6:0]),
56                                 .led2(HEX2[6:0]),
57                                 .led4(HEX4[6:0]),
58                                 .led5(HEX5[6:0]));
59    endmodule
```

1.F) DE1_SoC.sv (testbench)

```
61
62    //DE1_SoC_testbench tests expected and unexpected behavior. I first set
63    //an address and have write not enabled for one clock cycle. I then set
64    //write enable high and passed data. I checked to see if the data could be
65    //read after written, and I did this a couple times.
66    module DE1_SoC_testbench();
67          logic       CLOCK_50;
68          logic [3:0] KEY;
69          logic [9:0] SW;
70          logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
71
72          DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);
73
74          parameter CLOCK_PERIOD = 100;
75          initial begin
76                CLOCK_50 <= 0;
77                forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;// Forever toggle the clock
78          end
79
80          initial begin
81             SW[8:4] <= 5'b00000; SW[9] <= 0; repeat(1)    @(posedge    CLOCK_50);
82             SW[9] <= 1; SW[3:0] <= 4'b0011;  repeat(1)    @(posedge    CLOCK_50);
83             SW[9] <= 0;                      repeat(1)    @(posedge    CLOCK_50);
84             SW[8:4] <= 5'b00111;             repeat(1)    @(posedge    CLOCK_50);
85             SW[9] <= 1; SW[3:0] <= 4'b1111;  repeat(1)    @(posedge    CLOCK_50);
86    //       SW[9] <= 1;                      repeat(1)    @(posedge    CLOCK_50);
87             SW[9] <= 0;                      repeat(1)    @(posedge    CLOCK_50);
88             SW[8:4] <= 5'b00000;             repeat(1)    @(posedge    CLOCK_50);
89             SW[8:4] <= 5'b00111;             repeat(1)    @(posedge    CLOCK_50);
90                                              repeat(1)    @(posedge    CLOCK_50);
91
92             $stop; // End the simulation.
93          end
94    endmodule
```

2.A) hex32x4.sv

```
  1    //Garrett Tashiro
  2    //October 18, 2021
  3    //EE 371
  4    //Lab 2, Task 2.1
  5
  6
  7
  8    //hex32x4 has 4-bit read_data, write_data, 5-bit read_addr and write_addr
  9    //as inputs and returns 7-bit led0, led1, led2, led3, led4, and led5
 10    //as outputs. This module displays the data being read from a memory address
 11    //as well as that memory address. This module also displays the data being
 12    //written as well as the memory address to which it is being written to.
 13    //Data being read is displayed to HEX0. Data being written is displayed to
 14    //HEX1. The address being read is displayed to HEX2 and HEX3. The address
 15    //being written to is displayed on HEX4 and HEX5. All addresses and data
 16    //are displayed in hexcidecimal.
 17    module hex32x4(read_data, write_data, read_addr, write_addr, led0, led1, led2, led3, led4, led5);
 18          input logic [3:0]    read_data, write_data;
 19          input logic [4:0]    read_addr, write_addr;
 20          output logic [6:0]   led0, led1, led2, led3, led4, led5;
 21
 22
 23          //localparams to hold 7-bits to logic for hex displays
 24          localparam logic [6:0] zero = 7'b1000000;   //0
 25          localparam logic [6:0] one = 7'b1111001;    //1
 26          localparam logic [6:0] two = 7'b0100100;    //2
 27          localparam logic [6:0] three = 7'b0110000;  //3
 28          localparam logic [6:0] four = 7'b0011001;   //4
 29          localparam logic [6:0] five = 7'b0010010;   //5
 30          localparam logic [6:0] six = 7'b0000010;    //6
 31          localparam logic [6:0] seven = 7'b1111000;  //7
 32          localparam logic [6:0] eight = 7'b0000000;  //8
 33          localparam logic [6:0] nine = 7'b0010000;   //9
 34          localparam logic [6:0] A = 7'b0001000;      //A
 35          localparam logic [6:0] B = 7'b0000011;      //b
 36          localparam logic [6:0] C = 7'b1000110;      //C
 37          localparam logic [6:0] D = 7'b0100001;      //d
 38          localparam logic [6:0] E = 7'b0000110;      //E
 39          localparam logic [6:0] F = 7'b0001110;      //F
 40          localparam logic [6:0] blank = 7'b1111111;
 41
 42
 43          //Creating two 4-bit and two 1-bit logics to hold the upper and lower
 44          //bits of read_addr and write_addr
 45          logic upperWrite, upperRead;
 46          logic [3:0] lowerWrite, lowerRead;
```

```
47
48          //Assigning lowerWrite to bits 0-3 of write_addr for hex output
49          assign lowerWrite = write_addr[3:0];
50          //Assigning upperWrite to bit 4 of write_addr for hex output
51          assign upperWrite = write_addr[4];
52
53          //Assign lowerRead to bits 0-3 of read_addr for hex output
54          assign lowerRead = read_addr[3:0];
55          //Assign upperRead to bit 4 of read_addr for hex output
56          assign upperRead = read_addr[4];
57
58
59          //always_comb for combinational logic with a case statement for write_data.
60          //The case statement will set the output for HEX2 in hexidecimal for
61          //the data that is being written to a memory address
62          always_comb begin
63             case(write_data)
64
65                4'd0: begin
66                      led1 = zero;
67                end
68
69                4'd1: begin
70                      led1 = one;
71                end
72
73                4'd2: begin
74                      led1 = two;
75                end
76
77                4'd3: begin
78                      led1 = three;
79                end
80
81                4'd4: begin
82                      led1 = four;
83                end
84
85                4'd5: begin
86                      led1 = five;
87                end
88
89                4'd6: begin
90                      led1 = six;
91                end
92
93                4'd7: begin
94                      led1 = seven;
95                end
96
97                4'd8: begin
98                      led1 = eight;
99                end
100
101               4'd9: begin
102                     led1 = nine;
103               end
104
105               4'd10: begin
106                     led1 = A;
107               end
108
109               4'd11: begin
110                     led1 = B;
111               end
112
113               4'd12: begin
114                     led1 = C;
115               end
116
117               4'd13: begin
118                     led1 = D;
119               end
120
121               4'd14: begin
122                     led1 = E;
123               end
124
125               4'd15: begin
126                     led1 = F;
127               end
128
129               default: begin
130                     led1 = 7'bx;
131               end
132            endcase
133         end
```

```
134
135         //always_comb for combinational logic with a case statement for read_data.
136         //The case statement will set the output for HEX0 in hexidecimal for
137         //the data that is being read from an read_address location in the register.
138         always_comb begin
139            case(read_data)
140
141               4'd0: begin
142                     led0 = zero;
143               end
144
145               4'd1: begin
146                     led0 = one;
147               end
148
149               4'd2: begin
150                     led0 = two;
151               end
152
153               4'd3: begin
154                     led0 = three;
155               end
156
157               4'd4: begin
158                     led0 = four;
159               end
160
161               4'd5: begin
162                     led0 = five;
163               end
164
165               4'd6: begin
166                     led0 = six;
167               end
168
169               4'd7: begin
170                     led0 = seven;
171               end
172
173               4'd8: begin
174                     led0 = eight;
175               end
176
177               4'd9: begin
178                     led0 = nine;
```

```
179               end
180
181               4'd10: begin
182                     led0 = A;
183               end
184
185               4'd11: begin
186                     led0 = B;
187               end
188
189               4'd12: begin
190                     led0 = C;
191               end
192
193               4'd13: begin
194                     led0 = D;
195               end
196
197               4'd14: begin
198                     led0 = E;
199               end
200
201               4'd15: begin
202                     led0 = F;
203               end
204
205               default: begin
206                     led0 = 7'bx;
207               end
208            endcase
209         end
210
211         //always_comb for combinational logic with a case statement for lowerWrite.
212         //The case statement will set the output for HEX4 in hexidecimal for
213         //the lower part of the memory write_address
214         always_comb begin
215            case(lowerwrite)
216
217               4'd0: begin
218                     led4 = zero;
219               end
220
221               4'd1: begin
222                     led4 = one;
223               end
```

```
            4'd2: begin
                led4 = two;
            end

            4'd3: begin
                led4 = three;
            end

            4'd4: begin
                led4 = four;
            end

            4'd5: begin
                led4 = five;
            end

            4'd6: begin
                led4 = six;
            end

            4'd7: begin
                led4 = seven;
            end

            4'd8: begin
                led4 = eight;
            end

            4'd9: begin
                led4 = nine;
            end

            4'd10: begin
                led4 = A;
            end

            4'd11: begin
                led4 = B;
            end

            4'd12: begin
                led4 = C;
            end

            4'd13: begin
                led4 = D;
            end

            4'd14: begin
                led4 = E;
            end

            4'd15: begin
                led4 = F;
            end

            default: begin
                led4 = 7'bx;
            end
        endcase
    end

    //always_comb for combinational logic with a case statement for upperWrite.
    //The case statement will set the output for HEX5 in hexidecimal for
    //the upper part of the memory write_address. Only going to 32, so this only
    //needs to check for the 5th bit being a 1 or 0.
    always_comb begin
        case(upperWrite)

            1'd0: begin
                led5 = zero;
            end

            1'd1: begin
                led5 = one;
            end

            default: begin
                led5 = 7'bx;
            end
        endcase
    end

    //always_comb for combinational logic with a case statement for lowerRead.
    //The case statement will set the output for HEX4 in hexidecimal for
    //the lower part of the memory read_address
    always_comb begin
        case(lowerRead)
```

```
312             case(lowerRead)
313
314                4'd0: begin
315                      led2 = zero;
316                end
317
318                4'd1: begin
319                      led2 = one;
320                end
321
322                4'd2: begin
323                      led2 = two;
324                end
325
326                4'd3: begin
327                      led2 = three;
328                end
329
330                4'd4: begin
331                      led2 = four;
332                end
333
334                4'd5: begin
335                      led2 = five;
336                end
337
338                4'd6: begin
339                      led2 = six;
340                end
341
342                4'd7: begin
343                      led2 = seven;
344                end
345
346                4'd8: begin
347                      led2 = eight;
348                end
349
350                4'd9: begin
351                      led2 = nine;
352                end
353
354                4'd10: begin
355                      led2 = A;
356                end
```

```
357
358                4'd11: begin
359                      led2 = B;
360                end
361
362                4'd12: begin
363                      led2 = C;
364                end
365
366                4'd13: begin
367                      led2 = D;
368                end
369
370                4'd14: begin
371                      led2 = E;
372                end
373
374                4'd15: begin
375                      led2 = F;
376                end
377
378                default: begin
379                      led2 = 7'bx;
380                end
381             endcase
382          end
383
384          //always_comb for combinational logic with a case statement for upperRead.
385          //The case statement will set the output for HEX5 in hexidecimal for
386          //the upper part of the memory read_address. Only going to 32, so this only
387          //needs to check for the 5th bit being a 1 or 0.
388          always_comb begin
389             case(upperRead)
390
391                1'd0: begin
392                      led3 = zero;
393                end
394
395                1'd1: begin
396                      led3 = one;
397                end
398
399                default: begin
400                      led3 = 7'bx;
401                end
402             endcase
403          end
404    endmodule
```

2.B) hex32x4.sv (testbench)

```
406  //hex32x4_testbench tests for expected and unexpected behavior for
407  //the hex displays. The first three tests are for the address being read
408  //and written since they are both 5-bits. The three tests done were at 0,
409  //5, and 18. This was to make sure the second HEX display for each address
410  //would update correctly. The next three tests are for read and write data.
411  //These tests do three numbers to see if the HEX displays for them update
412  //correctly.
413  module hex32x4_testbench();
414        logic [3:0]    read_data, write_data;
415        logic [4:0]    read_addr, write_addr;
416        logic [6:0]    led0, led1, led2, led3, led4, led5;
417
418        hex32x4 dut(.read_data, .write_data, .read_addr, .write_addr, .led0, .led1, .led2, .led3, .led4, .led5);
419
420        int i;
421
422        initial begin
423           read_addr = 5'd0; write_addr = 5'd0;   #10;
424           read_addr = 5'd5; write_addr = 5'd5;   #10;
425           read_addr = 5'd18; write_addr = 5'd18; #10;
426
427           read_data = 4'd0; write_data = 4'd0;   #10;
428           read_data = 4'd6; write_data = 4'd6;   #10;
429           read_data = 4'd10; write_data = 4'd10; #10;
430        end
431  endmodule
```

2.C) counter.sv

```
1   //Garrett Tashiro
2   //October 18, 2021
3   //EE 371
4   //Lab 2, Task 2.2
5
6   //counter is a parameterized module that has 1-bit clk_divide, and
7   //reset as inputs and returns a parameterized output read_out. For
8   //lab 2, parameter count is 5, so the output read_out will be 5-bits
9   //for the address. This module uses clk_divide to have a roughly 1 second
10  //clock speed to update the memory addresses being read in sequential order
11  //by increasing read_out by 1 on the positive clock edge of clk_divide.
12  module counter #(parameter count = 5) (clk_divide, reset, read_out);
13        input logic                    clk_divide, reset;
14        output logic [count - 1:0]     read_out;
15
16
17        //always_ff block to update on the positive clock edge. This block
18        //checks if 1-bit reset is high and if it is then it zeros out
19        //read_out, and if reset is not high then it will add 1 to read_out.
20        always_ff @(posedge clk_divide) begin
21           if(reset) begin
22              read_out <= '0;
23           end
24
25           else begin
26              read_out <= read_out + 1;
27           end
28        end
29  endmodule
```

2.C) counter.sv (testbench)

```
30
31  //counter_testbench tests for expected and unexpected behavior
32  //of the counter module. It first starts by setting reset high then
33  //low. After that it waits for 33 clock cycles for the counter has time
34  //to reach its max and go back down to 0. Then reset is set high then
35  //low again to show the count will reset to 0 and then continues to
36  //count for 5 more clock cycles.
37  module counter_testbench();
38        logic          clk_divide, reset;
39        logic [4:0]    read_out;
40
41
42        counter #(.count(5)) dut(.clk_divide, .reset, .read_out);
43
44        parameter clk_PERIOD = 100;
45        initial begin
46              clk_divide <= 0;
47              forever #(clk_PERIOD/2) clk_divide <= ~clk_divide;// Forever toggle the clk
48        end
49
50        initial begin
51           reset <= 1;          repeat(1)   @(posedge    clk_divide);
52           reset <= 0;          repeat(1)   @(posedge    clk_divide);
53                                repeat(33)  @(posedge    clk_divide);
54           reset <= 1;          repeat(1)   @(posedge    clk_divide);
55           reset <= 0;          repeat(1)   @(posedge    clk_divide);
56                                repeat(5)   @(posedge    clk_divide);
57           $stop; // End the simulation.
58        end
59  endmodule
```

2.D) paramDFF.sv

```
1  //Garrett Tashiro
2  //October 18, 2021
3  //EE 371
4  //Lab 2, Task 2.3
5
6  //doubleD has 1-bit clk, reset, and press as inputs, and
7  //returns 1-bit out. This is a parameterized module to be able
8  //to change the number of bits being passed. This module is a double
9  //DFF (two in series) that takes the input signal from a switches, or
10 //buttons to prevent metastability.
11 module paramDFF #(parameter itsy = 1)(clk, reset, press, out);
12       input logic                 reset, clk;
13       input logic [itsy - 1:0]    press;
14       output logic [itsy - 1:0]   out;
15
16       logic [itsy - 1:0] temp1;
17
18       //always_ff replicates a double DFF. The input press goes into the
19       //first DFF and the output from the first DFF is the input for the
20       //second DFF.
21       always_ff @(posedge clk) begin
22          if (reset) begin
23             temp1 <= '0;
24             out   <= '0;
25          end
26
27          else begin
28             temp1 <= press;
29             out   <= temp1;
30        end
31       end
32 endmodule
33
```

2.E) paramDFF.sv (testbench)

```
34 //paramDFF_testbench tests the behaivor of a parameterized double DFF
35 //module. The test first sets reset to high and then low. It then tests
36 //4 different values being passed through the DFF's in series. The updated
37 //output takes two clock cycles, so wait a few clock cycles at the end.
38 module paramDFF_testbench();
39       logic          reset, clk;
40       logic [3:0]    press, out;
41
42       paramDFF #(.itsy(4)) dut(.clk, .reset, .press, .out);
43
44       parameter clk_PERIOD = 100;
45       initial begin
46             clk <= 0;
47             forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
48       end
49
50       initial begin
51          reset <= 1;                          repeat(1)   @(posedge    clk);
52          reset <= 0; press <= 4'b1000;        repeat(1)   @(posedge    clk);
53          press <= 4'b0011;                    repeat(1)   @(posedge    clk);
54          press <= 4'b1100;                    repeat(1)   @(posedge    clk);
55          press <= 4'b0100;                    repeat(1)   @(posedge    clk);
56                                               repeat(3)   @(posedge    clk);
57
58          $stop; // End the simulation.
59       end
60 endmodule
```

2.F) clock_divider.sv

```
1   //Garrett Tashiro
2   //October 10, 2021
3   //EE 371
4   //Lab 2, Task 2.4
5
6   //clock_divider has 1-bit clock and 1-bit reset as inputs and returns
7   //divided_clocks as an output. This module allows you to change the
8   //clock rate of CLOCK_50 in order to be able to have HEX displays
9   //update roughly every second. This is a module written in 271.
10
11  // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
12  module clock_divider (clock, reset, divided_clocks);
13          input  logic          reset, clock;
14          output logic  [31:0]  divided_clocks = 0;
15
16          always_ff @(posedge clock) begin
17             divided_clocks <= divided_clocks + 1;
18          end
19  endmodule
20
```

2.G) clock_divider.sv (testbench)

```
20
21  //clock_divider_testbench tests for expected and unexpected behavior.
22  //This testbench resets and just runs for 100 clock cycles
23  module clock_divider_testbench();
24        logic          reset, clock;
25        logic [31:0]   divided_clocks;
26
27        clock_divider dut(.clock, .reset, .divided_clocks);
28
29        parameter clk_PERIOD = 100;
30        initial begin
31              clock <= 0;
32              forever #(clk_PERIOD/2) clock <= ~clock;// Forever toggle the clk
33        end
34
35        initial begin
36           reset <= 1;    repeat(1)   @(posedge    clock);
37           reset <= 0;    repeat(1)   @(posedge    clock);
38                          repeat(100)   @(posedge    clock);
39           $stop; // End the simulation.
40        end
41  endmodule
```

2.H) ram32x4.v

```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module ram32x4 (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input    clock;
    input [3:0]  data;
    input [4:0]  rdaddress;
    input [4:0]  wraddress;
    input    wren;
    output   [3:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1     clock;
    tri0     wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [3:0] sub_wire0;
    wire [3:0] q = sub_wire0[3:0];

    altsyncram  altsyncram_component (
              .address_a (wraddress),
              .address_b (rdaddress),
              .clock0 (clock),
              .data_a (data),
              .wren_a (wren),
              .q_b (sub_wire0),
              .aclr0 (1'b0),
              .aclr1 (1'b0),
              .addressstall_a (1'b0),
              .addressstall_b (1'b0),
              .byteena_a (1'b1),
              .byteena_b (1'b1),
              .clock1 (1'b1),
              .clocken0 (1'b1),
              .clocken1 (1'b1),
              .clocken2 (1'b1),
              .clocken3 (1'b1),
              .data_b ({4{1'b1}}),
              .eccstatus (),
              .q_a (),
              .rden_a (1'b1),
              .rden_b (1'b1),
              .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_b = "NONE",
        altsyncram_component.address_reg_b = "CLOCK0",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_input_b = "BYPASS",
        altsyncram_component.clock_enable_output_b = "BYPASS",
        altsyncram_component.init_file = "ram32x4.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32,
        altsyncram_component.numwords_b = 32,
        altsyncram_component.operation_mode = "DUAL_PORT",
        altsyncram_component.outdata_aclr_b = "NONE",
        altsyncram_component.outdata_reg_b = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
        altsyncram_component.widthad_a = 5,
        altsyncram_component.widthad_b = 5,
        altsyncram_component.width_a = 4,
        altsyncram_component.width_b = 4,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

2.I) DE1_SoC.sv

```
1   //Garrett Tashiro
2   //October 19, 2021
3   //EE 371
4   //Lab 2, Task 2.4
5
6   //DE1_SoC has implements a dual port RAM, and has hierarchical
7   //calls to paramDFF, clock_divider, counter, ram32x4, and hex32x4.
8   //This module takes inputs from KEY[0], KEY[3], and SW[8:0].
9   //Address for the RAM is taken from SW[8:4], data is input from
10  //SW[3:0]. The clock diver is used to cycle through all memory
11  //addresses at roughly 1 second and displays the data for all
12  //addresses. Each memory address can be written to, and all
13  //addresses as well as data is displayed on the seven segment
14  //HEX displays.
15
16  //DE1_SoC is the top level module.
17  module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, CLOCK_50);
18        input logic           CLOCK_50;
19        input logic [3:0]     KEY;
20        input logic [9:0]     SW;
21        output logic [6:0]    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
22
23
24        //32-bit logic to hold output from clock_divider
25        logic [31:0] div_clk;
26
27        //1-bit logic to hold value from KEY[0] for reset
28        logic reset;
29        assign reset = ~KEY[0];
30
31        //5-bit, 4-bit, and 1-nit logic to hold output values from DFF's
32        logic [4:0] write_address;
33        logic [3:0] write_data;
34        logic write_enable;
35
36        //5-bit logic count_addr to hold the output from counter module
37        logic [4:0] count_addr;
38
39        //4-bit logic to hold output from the ram32x4 to display on hex displays
40        logic [3:0] read_data;
```

```
41
42        //clock_divider has 1-bit CLOCK_50, and reset as inputs and returns 1-bit
43        //div_clk as an output.This module divides the clock to lower the frequency
44        //of CLOCK_50
45        clock_divider oneSec(.clock(CLOCK_50), .reset(reset), .divided_clocks(div_clk));
46
47        //1-bit logic clk for the clock on board or during simulation
48        logic clk;
49        //assign clk = CLOCK_50;      // for simulation
50        assign clk = div_clk[25];  // for board
51
52
53        //paramDFF address has 1-bit CLOCK_50, reset, and 5-bit SW[8:4] as inputs and
54        //returns 5-bit write_address. write_address is the address being written to,
55        //and this output is passed through two DFF's to prevent metastability.
56        paramDFF #(.itsy(5)) address(.clk(CLOCK_50), .reset(reset), .press(SW[8:4]), .out(write_address));
57
58        //paramDFF data has 1-bit CLOCK_50, reset, and 4-bit SW[3:0] as inputs and
59        //returns 4-bit write_data. write_data is the data being written to an address,
60        //and this output is passed through two DFF's to prevent metastability.
61        paramDFF #(.itsy(4)) data(.clk(CLOCK_50), .reset(reset), .press(SW[3:0]), .out(write_data));
62
63        //paramDFF data has 1-bit CLOCK_50, reset, and 1-bit ~KEY[3] as inputs and
64        //returns 1-bit write_enable. write_enable is the enable to be able to write
65        //to an address and this output is passed through two DFF's to prevent metastability.
66        paramDFF #(.itsy(1)) enable(.clk(CLOCK_50), .reset(reset), .press(~KEY[3]), .out(write_enable));
67
68        //counter read_mem has 1-bit clk, and reset as inputs and returns 5-bit count_addr
69        //as its output. This module is meant to count up to display the memory addresses.
70        counter #(.count(5)) read_mem(.clk_divide(clk), .reset(reset), .read_out(count_addr));
71
72        //ram32x4 duhRAM has 1-bit CLOCK_50, write_enable, 4-bit write_data, 5-bit count_addr,
73        //and write_address as inputs and returns 4-bit read_data. This module is the dual port
74        //RAM for this sytem.
75        ram32x4 duhRAM(.clock(CLOCK_50),
76                       .data(write_data),
77                       .rdaddress(count_addr),
78                       .wraddress(write_address),
79                       .wren(write_enable),
80                       .q(read_data));
81
```

```
82          //hex32x4 allDisplays has 4-bit read_data, write_data, 5-bit read_addr, and write_addr
83          //as inputs and returns 7-bit HEX0-HEX5. This module takes the data and addresses and
84          //displays them to the HEX displays on the board.
85          hex32x4 allDisplays(.read_data(read_data),
86                              .write_data(write_data),
87                              .read_addr(count_addr),
88                              .write_addr(write_address),
89                              .led0(HEX0),
90                              .led1(HEX1),
91                              .led2(HEX2),
92                              .led3(HEX3),
93                              .led4(HEX4),
94                              .led5(HEX5));
95
96      endmodule
```

2.J) DE1_SoC.sv (testbench)

```
98      //DE1_SoC_testbench test for expected and unexpected behavior.
99      //I first set SW inputs to 0, and reset the system. I let the
100     //system run for 33 clock cycles in order to make sure that
101     //the counter was working correctly and updating the HEX's.
102     //I then wrote to an address and with write enabled. I next
103     //tried writing to an address without write enabled. I reset
104     //again to see what was written.
105     `timescale 1 ps / 1 ps
106     module DE1_SoC_testbench();
107           logic CLOCK_50;
108           logic [3:0] KEY;
109           logic [9:0] SW;
110           logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
111
112           DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);
113
114           parameter CLOCK_PERIOD=100;
115           initial begin
116              CLOCK_50 <= 0;
117              forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
118           end
119
120           initial begin
121              SW[8:4] <= 5'b00000; SW[3:0] <= 4'b0000; repeat(1)   @(posedge    CLOCK_50);
122              KEY[0] <= 0;       repeat(2)   @(posedge    CLOCK_50);
123              KEY[0] <= 1;       repeat(2)   @(posedge    CLOCK_50);
124                                 repeat(33)   @(posedge    CLOCK_50);
125              SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; KEY[3] <= 0; repeat(2)   @(posedge    CLOCK_50);
126              SW[8:4] <= 5'b11101; SW[3:0] <= 4'b0010; KEY[3] <= 1; repeat(2)   @(posedge    CLOCK_50);
127                                                                    repeat(2)   @(posedge    CLOCK_50);
128              KEY[0] <= 0;       repeat(2)   @(posedge    CLOCK_50);
129              KEY[0] <= 1;       repeat(2)   @(posedge    CLOCK_50);
130
131              $stop; // End the simulation.
132           end
133     endmodule
```

3.A) FIFO_Control.sv

```
1       //Garrett Tashiro
2       //October 19, 2021
3       //EE 371
4       //Lab 2, Task 3.1
5
6
7       //FIFO_Control has 1-bit clk, reset, read, and write as inputs and returns
8       //1-bit wr_en, empty, full, 4-bit (set by parameter depth) readAddr
9       //and writeAddr as outputs. This module implements an FSM which acts like
10      //a queue for memory module. The FIFO follows first in, first out, so a queue is
11      //made through an FSM. The FSM tests if the memory is full or empty, if
12      //it is not full then it can be written to, and if it is not empty it
13      //can be read from.
14      module FIFO_Control #(parameter depth = 4)(clk, reset, read, write, wr_en, empty, full, readAddr, writeAddr);
15            input logic              clk, reset;
16            input logic              read, write;
17            output logic             wr_en;
18            output logic             empty, full;
19            output logic [depth-1:0] readAddr, writeAddr;
20
21
22
23          //logic to be pointers for queue. Moon phase names.
24          //crescent: close to empty
25          //gibbous: close to full
26          logic crescent, gibbous;
27
28          //crescent points to one place behind of readAddr
29          //gibbous points to one place behind of writeAddr
30          assign crescent = (writeAddr == readAddr + 4'b0001);
31          assign gibbous = (readAddr == writeAddr + 4'b0001);
```

```
33      //Three states for the FSM.
34      //clear = empty
35      //stocked = full
36      enum{clear, neither, stocked} ps, ns;
37
38      //always_comb for and FSM for the register. This FSM
39      //implements a queue for reading and writting to addresses
40      //in memory. This FSM has states clear, neither, and stocked.
41      //Each state checks if a read or write is happening, and
42      //neither takes into account if the queue is becoming full
43      //or empty with logics crescent and gibbous.
44      always_comb begin
45         case(ps)
46
47            clear: begin
48               if(read == 0 && write == 1) begin
49                  ns = neither;
50               end
51
52               else begin
53                  ns = clear;
54               end
55            end
56
57            neither: begin
58               if(read == 1 && write == 0 && crescent == 1) begin
59                  ns = clear;
60               end
61
62               else if(read == 0 && write == 1 && gibbous == 1) begin
63                  ns = stocked;
64               end
65
66               else begin
67                  ns = neither;
68               end
69            end
70
71            stocked: begin
72               if(read == 1 && write == 0) begin
73                  ns = neither;
74               end
75
76               else begin
77                  ns = stocked;
78               end
79            end
80         endcase
81      end
```

```
 82
 83      //Assignment for write enable to be 1 when
 84      //write is 1 and full is 0.
 85      assign wr_en = (write == 1 && full == 0);
 86
 87      assign empty = (ps == clear);
 88      assign full = (ps == stocked);
 89
 90      //This always_ff assigns will set the value for the
 91      //1-bit outputs empty and full. This is done by
 92      //testing if reset is high, or depending on what
 93      //state the FSM is in.
 94  // always_ff @(posedge clk) begin
 95  //     if(reset) begin
 96  //         empty <= 1;
 97  //         full <= 0;
 98  //     end
 99  //
100  //     else if(ps == neither) begin
101  //         empty <= 0;
102  //         full <= 0;
103  //     end
104  //     else if(ps == clear) begin
105  //         empty <= 1;
106  //         full <= 0;
107  //     end
108  //
109  //     else if(ps == stocked) begin
110  //         empty <= 0;
111  //         full <= 1;
112  //     end
113  // end
114
115      //This always_ff updates ps as well as 4-bit outputs
116      //readAddr and writeAddr. Upon reset, readAddr and
117      //writeAddr are zeroed out and ps is set to state clear.
118      //If/else if statements are used to see if the queue is
119      //empty or full and update readAddr and writeAddr if queue
120      //is not full and ps will then always be updated to ns.
121      always_ff @(posedge clk) begin
122          if(reset) begin
123              readAddr <= '0;
124              writeAddr <= '0;
125              ps <= clear;
126          end
127
128          else if(empty == 0 && read == 1) begin
129              readAddr <= readAddr + 4'b0001;
130              ps <= ns;
131          end
132
133           else if(full == 0 && write == 1) begin
134               writeAddr <= writeAddr + 4'b0001;
135               ps <= ns;
136           end
137
138           else begin
139               ps <= ns;
140           end
141       end
142  endmodule
```

3.B) FIFO_Control.sv (testbench)

```
144  //FIFO_Control_testbench tests for expected and unexpected behavior.
145  //1-bit inputs read and write are first set to 0, then reset is set
146  //to 1 then 0 for one clock cycle. 1-bit input write is set to 1 then
147  //0 for one clock cycle each. This is repeated 5 times to test that
148  //4-bit writeAddr and 1-bit empty outputs updated correctly. 1-bit
149  //read input is set to 1 for one clock cycle then set to 0 for 1 clock
150  //cycle. This is repeated 5 times to test if 4-bit readAddr updated
151  //correctly and to see if 1-bit output empty goes high when the queue
152  //is empty.
153  module FIFO_Control_testbench();
154        logic             clk, reset;
155        logic             read, write;
156        logic             wr_en;
157        logic             empty, full;
158        logic [3:0] readAddr, writeAddr;
159
160        FIFO_Control  dut(.clk, .reset, .read, .write, .wr_en, .empty, .full, .readAddr, .writeAddr);
161
162        parameter clk_PERIOD = 100;
163        initial begin
164              clk <= 0;
165              forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
166        end
167
168        initial begin
169           read <= 0; write <= 0; repeat(1)   @(posedge    clk);
170           reset <= 1;            repeat(1)   @(posedge    clk);
171           reset <= 0;            repeat(1)   @(posedge    clk);
172
173           repeat(17) begin
174           write <= 1;            repeat(1)   @(posedge    clk);
175           write <= 0;            repeat(1)   @(posedge    clk);
176           end
177                                  repeat(1)   @(posedge    clk);
178           repeat(18) begin
179           read <= 1;             repeat(1)   @(posedge    clk);
180           read <= 0;             repeat(1)   @(posedge    clk);
181           end
182                                  repeat(1)   @(posedge    clk);
183           write <= 1;            repeat(1)   @(posedge    clk);
184           write <= 0;            repeat(1)   @(posedge    clk);
185           reset <= 1;            repeat(1)   @(posedge    clk);
186           reset <= 0;            repeat(1)   @(posedge    clk);
187                                  repeat(1)   @(posedge    clk);
188
189           $stop; // End the simulation.
190        end
191  endmodule
```

3.C) FIFO.sv

```
1   //Garrett Tashiro
2   //October 19, 2021
3   //EE 371
4   //Lab 2, Task 3.2
5
6   //FIFO is a parameterized module that has 1-bit clk, reset, read,
7   //write, and 8-bit (based on parameter size) inputBus as inputs and
8   //returns 1-bit empty, full, and 8-bit (based on parameter size)
9   //outputBus. This module has two hierarchical calls, one is to
10  //ram16x8 with is the RAM module created from the IP Cat., and the
11  //other is FIFO_Control which imlpements an FSM that is a queue
12  //for data being stored into RAM.
13  module FIFO #(
14                parameter depth = 4,
15                parameter width = 8
16                )(
17                  input logic clk, reset,
18                  input logic read, write,
19                  input logic [width-1:0] inputBus,  //8-bit
20                 output logic empty, full,
21                 output logic [width-1:0] outputBus  //8-bit
22                 );
23
24     ///1-bit logic for enable to be the output from FC into ram16x8
25     //readAddr, and writeAddr to hold values from output of FC and
26     //input into ram16x8
27     logic enable;
28     logic [depth - 1:0] readAddr, writeAddr;
29
30     /*       Instantiate_Your_Dual-Port_RAM_Here       */
31     //Dual port RAM created with the IP cat. This is a 16x8 RAM reg.
32     ram16x8 nawFam(.clock(clk), .data(inputBus), .rdaddress(readAddr), .wraddress(writeAddr), .wren(enable), .q(outputBus));
33
34
35     /*       FIFO-Control_Module        */
36     //FC is a control module for the FIFO. This module implements an FSM
37     //to create a queue for reads and writes to addresses in the memory.
38     FIFO_Control #(depth) FC (.clk(clk), .reset(reset),
39                               .read(read),
40                               .write(write),
41                               .wr_en(enable),
42                               .empty(empty),
43                               .full(full),
44                               .readAddr(readAddr),
45                               .writeAddr(writeAddr)
46                              );
47
48  endmodule
```

3.D) FIFO.sv (testbench)

```
50  //FIFO_testbench tests expected, unexpected, and edgecase behavior.
51  //The first test is to see if you can go from empty to full without
52  //writing again once full. The next test is to go from full to empty
53  //to see if you can write when empty. Next test is to see if you can
54  //write without write being high. After that, data is written once,
55  //and then read and write happen at the same time. Reset at the end
56  //to make sure it goes back to empty.
57
58  `timescale 1ps/1ps
59  module FIFO_testbench();
60
61     parameter depth = 4, width = 8;
62
63     logic clk, reset;
64     logic read, write;
65     logic [width-1:0] inputBus;
66     logic empty, full;
67     logic [width-1:0] outputBus;
68
69     FIFO #(depth, width) dut (.*);
70
71     parameter CLK_Period = 100;
72
73     initial begin
74        clk <= 1'b0;
75        forever #(CLK_Period/2) clk <= ~clk;
76     end
77
78     int i;
79
80     initial begin
81        read <= 0; write <= 0;          repeat(1)   @(posedge    clk);
82        reset <= 1;                     repeat(2)   @(posedge    clk);
83        reset <= 0;                     repeat(2)   @(posedge    clk);
84
85        for(i = 0; i < 17; i++) begin
86
87           write <= 1; inputBus <= i;   repeat(1)   @(posedge    clk);
88        end
89
90        write <= 0;                        repeat(1)   @(posedge    clk);
91
92        for(i = 0; i < 17; i++) begin
93
94           read <= 1; outputBus <= i;    repeat(1)   @(posedge    clk);
95        end
96
97        read <= 0;                         repeat(1)   @(posedge    clk);
98        write <= 0; inputBus <= 8'd20;     repeat(1)   @(posedge    clk);
99        write <= 1; inputBus <= 8'd29;     repeat(1)   @(posedge    clk);
100       write <= 0;                        repeat(1)   @(posedge    clk);
101
102       read <= 1; write <= 1; inputBus <= 8'd20; repeat(1)   @(posedge    clk);
103
104       read <= 0; write <= 0;          repeat(1)   @(posedge    clk);
105
106       reset <= 1;                     repeat(2)   @(posedge    clk);
107       reset <= 0;                     repeat(2)   @(posedge    clk);
108
109
110       $stop; // End the simulation.
111    end
112 endmodule
```

3.E) hexDisplay16x8.sv

```
1   //Garrett Tashiro
2   //October 19, 2021
3   //EE 371
4   //Lab 2, Task 3.3
5
6   //hexDisplay16x8 has 8-bit dataInput, and dataOutput as inputs
7   //and returns 7-bit led0, led1, led4, and led5 as outputs.
8   //This module drives HEX diplays 0, 1, 4, and 5. The data being
9   //input to the FIFO is displayed on HEX5-4. The data output from the
10  //FIFO is displayed on HEX1-0. All data is displayed in hexcidecimal.
11  module hexDisplay16x8(dataInput, dataOutput, led0, led1, led4, led5);
12        input logic [7:0]    dataInput, dataOutput;
13        output logic [6:0]   led0, led1, led4, led5;
14
15
16        //localparams to hold 7-bits to logic for hex displays
17        localparam logic [6:0] zero = 7'b1000000;   //0
18        localparam logic [6:0] one = 7'b1111001;    //1
19        localparam logic [6:0] two = 7'b0100100;    //2
20        localparam logic [6:0] three = 7'b0110000;  //3
21        localparam logic [6:0] four = 7'b0011001;   //4
22        localparam logic [6:0] five = 7'b0010010;   //5
23        localparam logic [6:0] six = 7'b0000010;    //6
24        localparam logic [6:0] seven = 7'b1111000;  //7
25        localparam logic [6:0] eight = 7'b0000000;  //8
26        localparam logic [6:0] nine = 7'b0010000;   //9
27        localparam logic [6:0] A = 7'b0001000;      //A
28        localparam logic [6:0] B = 7'b0000011;      //b
29        localparam logic [6:0] C = 7'b1000110;      //C
30        localparam logic [6:0] D = 7'b0100001;      //d
31        localparam logic [6:0] E = 7'b0000110;      //E
32        localparam logic [6:0] F = 7'b0001110;      //F
33
34
35        //4-bit logic to break the 8-bit dataInput and dataOutput into
36        //two 4-bit chunks. 4-bits of data for each hex display.
37        logic [3:0] lower_input, lower_output, upper_input, upper_output;
38
39        //assign bits 0-3 to each lower_x logic, and bits 4-7 to each
40        //upper_x logic respectfully.
41        assign lower_input = dataInput[3:0];
42        assign upper_input = dataInput[7:4];
43
44        assign lower_output = dataOutput[3:0];
45        assign upper_output = dataOutput[7:4];
```

```
47        //always_comb for combinational logic with a case statement for upper_output.
48        //The case statement will set the output for HEX1 in hexidecimal for
49        //the data that is being read from the FIFO.
50        always_comb begin
51           case(upper_output)
52
53              4'd0: begin
54                    led1 = zero;
55              end
56
57              4'd1: begin
58                    led1 = one;
59              end
60
61              4'd2: begin
62                    led1 = two;
63              end
64
65              4'd3: begin
66                    led1 = three;
67              end
68
69              4'd4: begin
70                    led1 = four;
71              end
72
73              4'd5: begin
74                    led1 = five;
75              end
76
77              4'd6: begin
78                    led1 = six;
79              end
80
81              4'd7: begin
82                    led1 = seven;
83              end
84
85              4'd8: begin
86                    led1 = eight;
87              end
88
89              4'd9: begin
90                    led1 = nine;
91              end
92
93              4'd10: begin
94                    led1 = A;
95              end
```

```
            4'd11: begin
                  led1 = B;
            end

            4'd12: begin
                  led1 = C;
            end

            4'd13: begin
                  led1 = D;
            end

            4'd14: begin
                  led1 = E;
            end

            4'd15: begin
                  led1 = F;
            end

            default: begin
                  led1 = 7'bx;
            end
         endcase
      end

      //always_comb for combinational logic with a case statement for lower_output.
      //The case statement will set the output for HEX0 in hexidecimal for
      //the data that is being read out of the FIFO.
      always_comb begin
         case(lower_output)

            4'd0: begin
                  led0 = zero;
            end

            4'd1: begin
                  led0 = one;
            end

            4'd2: begin
                  led0 = two;
            end

             4'd3: begin
                   led0 = three;
             end

             4'd4: begin
                   led0 = four;
             end

             4'd5: begin
                   led0 = five;
             end

             4'd6: begin
                   led0 = six;
             end

             4'd7: begin
                   led0 = seven;
             end

             4'd8: begin
                   led0 = eight;
             end

             4'd9: begin
                   led0 = nine;
             end

             4'd10: begin
                   led0 = A;
             end

             4'd11: begin
                   led0 = B;
             end

             4'd12: begin
                   led0 = C;
             end

             4'd13: begin
                   led0 = D;
             end

             4'd14: begin
                   led0 = E;
             end

             4'd15: begin
                   led0 = F;
```

```
                    led0 = F;
                end

                default: begin
                    led0 = 7'bx;
                end
            endcase
        end

        //always_comb for combinational logic with a case statement for upper_input.
        //The case statement will set the output for HEX1 in hexidecimal for
        //the data that is being input into the FIFO
        always_comb begin
            case(upper_input)

                4'd0: begin
                    led5 = zero;
                end

                4'd1: begin
                    led5 = one;
                end

                4'd2: begin
                    led5 = two;
                end

                4'd3: begin
                    led5 = three;
                end

                4'd4: begin
                    led5 = four;
                end

                4'd5: begin
                    led5 = five;
                end

                4'd6: begin
                    led5 = six;
                end

                4'd7: begin
                    led5 = seven;
                end

                4'd8: begin
                    led5 = eight;
                end
```

```
                4'd9: begin
                    led5 = nine;
                end

                4'd10: begin
                    led5 = A;
                end

                4'd11: begin
                    led5 = B;
                end

                4'd12: begin
                    led5 = C;
                end

                4'd13: begin
                    led5 = D;
                end

                4'd14: begin
                    led5 = E;
                end

                4'd15: begin
                    led5 = F;
                end

                default: begin
                    led5 = 7'bx;
                end
            endcase
        end
```

```
274
275         //always_comb for combinational logic with a case statement for lower_input.
276         //The case statement will set the output for HEX0 in hexidecimal for
277         //the data that is being input into the FIFO
278         always_comb begin
279            case(lower_input)
280
281               4'd0: begin
282                     led4 = zero;
283               end
284
285               4'd1: begin
286                     led4 = one;
287               end
288
289               4'd2: begin
290                     led4 = two;
291               end
292
293               4'd3: begin
294                     led4 = three;
295               end
296
297               4'd4: begin
298                     led4 = four;
299               end
300
301               4'd5: begin
302                     led4 = five;
303               end
304
305               4'd6: begin
306                     led4 = six;
307               end
308
309               4'd7: begin
310                     led4 = seven;
311               end
312
313               4'd8: begin
314                     led4 = eight;
315               end
316
317               4'd9: begin
318                     led4 = nine;
319               end
320
321               4'd10: begin
322                     led4 = A;
323               end
```

```
324
325               4'd11: begin
326                     led4 = B;
327               end
328
329               4'd12: begin
330                     led4 = C;
331               end
332
333               4'd13: begin
334                     led4 = D;
335               end
336
337               4'd14: begin
338                     led4 = E;
339               end
340
341               4'd15: begin
342                     led4 = F;
343               end
344
345               default: begin
346                     led4 = 7'bx;
347               end
348            endcase
349         end
350   endmodule
```

3.F) hexDisplay16x8.sv (testbench)

```
351
352   //hexDisplay16x8_testbench test four different inputs for
353   //dataInput, and dataOutput to make sure each HEX display
354   //updates accordingly. Two sets of numbers were the same
355   //and the other two sets of numbers to test as inputs were
356   //different to make sure the correct HEX displays were updating.
357   module hexDisplay16x8_testbench();
358         logic [7:0]    dataInput, dataOutput;
359         logic [6:0]    led0, led1, led4, led5;
360
361         hexDisplay16x8 dut(.dataInput, .dataOutput, .led0, .led1, .led4, .led5);
362
363         initial begin
364            dataInput = 8'd0; dataOutput = 8'd0;       #10;
365            dataInput = 8'd8; dataOutput = 8'd8;       #10;
366            dataInput = 8'd13;   dataOutput = 8'd20;   #10;
367            dataInput = 8'd62;   dataOutput = 8'd70;   #10;
368         end
369   endmodule
```

3.G) paramDFF.sv

```
1   //Garrett Tashiro
2   //October 20, 2021
3   //EE 371
4   //Lab 2, Task 3.4
5
6   //doubleD has 1-bit clk, reset, and press as inputs, and
7   //returns 1-bit out. This is a parameterized module to be able
8   //to change the number of bits being passed. This module is a double
9   //DFF (two in series) that takes the input signal from a switches, or
10  //buttons to prevent metastability.
11  module paramDFF #(parameter itsy = 1)(clk, reset, press, out);
12        input logic                reset, clk;
13        input logic [itsy - 1:0]   press;
14        output logic [itsy - 1:0]  out;
15
16        logic [itsy - 1:0] temp1;
17
18        //always_ff replicates a double DFF. The input press goes into the
19        //first DFF and the output from the first DFF is the input for the
20        //second DFF.
21        always_ff @(posedge clk) begin
22           if (reset) begin
23              temp1 <= '0;
24              out   <= '0;
25           end
26
27           else begin
28              temp1 <= press;
29              out   <= temp1;
30         end
31        end
32  endmodule
```

3.H) paramDFF.sv (testbench)

```
34  //paramDFF_testbench tests the behaivor of a parameterized double DFF
35  //module. The test first sets reset to high and then low. It then tests
36  //4 different values being passed through the DFF's in series. The updated
37  //output takes two clock cycles, so wait a few clock cycles at the end.
38  module paramDFF_testbench();
39        logic          reset, clk;
40        logic [3:0]    press, out;
41
42        paramDFF #(.itsy(4)) dut(.clk, .reset, .press, .out);
43
44        parameter clk_PERIOD = 100;
45        initial begin
46              clk <= 0;
47              forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
48        end
49
50        initial begin
51           reset <= 1;                          repeat(1)   @(posedge    clk);
52           reset <= 0; press <= 4'b1000;        repeat(1)   @(posedge    clk);
53           press <= 4'b0011;                    repeat(1)   @(posedge    clk);
54           press <= 4'b1100;                    repeat(1)   @(posedge    clk);
55           press <= 4'b0100;                    repeat(1)   @(posedge    clk);
56                                                repeat(3)   @(posedge    clk);
57
58           $stop; // End the simulation.
59        end
60  endmodule
```

3.I) inputBuffer.sv

```
1  //Garrett Tashiro
2  //October 20, 2021
3  //EE 371
4  //Lab 2, Task 3
5
6  //inputBuffer has 1-bit clk, reset, and press and
7  //returns 1-bit out. This module is a buffer so that
8  //the output from a button will act as one press.
9  //This file is from a 271 lab.
10 module inputBuffer(clk, reset, press, out);
11       input logic clk, reset, press;
12       output logic out;
13
14
15       logic ps, ns;
16
17       //This always_comb block sets ns to press and out to ~ps & press.
18       //The purpose of this block is to have the input create an output
19       //that is high for only one clock cycle
20       always_comb begin
21         ns = press;
22         out = (~ps & press);
23       end
24
25       //This always_ff block sets ps to 0 upon reset, otherwise it is set ns.
26       //if ns is high, then the always_comb will have the output only be high for
27       //one clock cycle.
28       always_ff @(posedge clk) begin
29         if(reset)
30             ps <= 0;
31         else
32             ps <= ns;
33       end
34 endmodule
```

3.J) inputBuffer.sv (testbench)

```
36 //inputBuffer_testbench tests expect and unexpected behavior.
37 //The test starts by resetting and setting press high for two
38 //clock cycles to make sure the output is only high for one
39 //clock cycle.
40 module inputBuffer_testbench();
41       logic clk, reset, press;
42       logic out;
43
44       inputBuffer dut(.clk, .reset, .press, .out);
45
46       parameter clk_PERIOD = 100;
47       initial begin
48             clk <= 0;
49             forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
50       end
51
52       initial begin
53          press <= 0;       repeat(1)   @(posedge    clk);
54          reset <= 1;       repeat(1)   @(posedge    clk);
55          reset <= 0;       repeat(1)   @(posedge    clk);
56          press <= 1;       repeat(1)   @(posedge    clk);
57          press <= 1;       repeat(1)   @(posedge    clk);
58          press <= 0;       repeat(1)   @(posedge    clk);
59
60          $stop; // End the simulation.
61       end
62 endmodule
```

3.K) ram16x8.v

```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module ram16x8 (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input    clock;
    input [7:0]  data;
    input [3:0]  rdaddress;
    input [3:0]  wraddress;
    input    wren;
    output   [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1     clock;
    tri0     wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    altsyncram  altsyncram_component (
                .address_a (wraddress),
                .address_b (rdaddress),
                .clock0 (clock),
                .data_a (data),
                .wren_a (wren),
                .q_b (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_b ({8{1'b1}}),
                .eccstatus (),
                .q_a (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_b = "NONE",
        altsyncram_component.address_reg_b = "CLOCK0",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_input_b = "BYPASS",
        altsyncram_component.clock_enable_output_b = "BYPASS",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 16,
        altsyncram_component.numwords_b = 16,
        altsyncram_component.operation_mode = "DUAL_PORT",
        altsyncram_component.outdata_aclr_b = "NONE",
        altsyncram_component.outdata_reg_b = "CLOCK0",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
        altsyncram_component.widthad_a = 4,
        altsyncram_component.widthad_b = 4,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_b = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

3.L) DE1_SoC.sv

```
//Garrett Tashiro
//October 20, 2021
//EE 371
//Lab 2, Task 3

//DE1_SoC has hierarchical calls to paramDFF, inputBuffer, FIFO, and
//hexDisplay16x8. This module takes inputs from switches and keys
//to write data to ram in the FIFO. DE1 implements dual port RAM
//with the ability to read and write. The FIFO has an FSM that is a queue
//to hold data being written to the RAM. Once full, LEDR9 will light up.
//If the RAM is empty, LEDR8 will light up. This is a 16x8 RAM module,
//so 16 inputs that are 8-bits long can be stored and read.

//DE1_SoC is the top level module for this task.
module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR, CLOCK_50);
      input logic          CLOCK_50;
      input logic [3:0]    KEY;
      input logic [9:0]    SW;
      output logic [9:0]   LEDR;
      output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;

      //1-bit logic for read, write, empty, and full to hold outputs
      logic read, write, full, empty;

      //8-bit logic for input from switches to inputBus, and for output outputBus
      logic [7:0] bussin_in, bussin_out;

      //Assign LEDR's to light up if full, or empty
      assign LEDR[8] = empty;
      assign LEDR[9] = full;

      //Have HEX2 and HEX3 be balnk since they aren't used to display data
      assign HEX2 = 7'b1111111;
      assign HEX3 = 7'b1111111;

      //paramDFF is two DFF's in series that is parameterized. This module takes in 1-bit CLOCK_50,
      //reset, and 8-bits from SW[7:0] as inputs and returns 8-bit bussin_in. This module assures
      //that inputs from SW[7:0] are not metastable. The output bussin_in is passed to FIFO and
      //hexDisplay16x8.
      paramDFF #(.itsy(8)) input_data(.clk(CLOCK_50), .reset(SW[9]), .press(SW[7:0]), .out(bussin_in));

      //inputBuffer read_data has 1-bit CLOCK_50, reset, and KEY[0] as inputs and returns 1-bit read
      //as an output. The output read is passed to FIFO. This module is to have just 1 single read
      //for a clock cycle.
      inputBuffer read_data(.clk(CLOCK_50), .reset(SW[9]), .press(~KEY[0]), .out(read));

      //inputBuffer write_data has 1-bit CLOCK_50, reset, and KEY[1] as inputs and returns 1-bit write
      //as an output. The output read is passed to FIFO. This module is to have just 1 single read
      //for a clock cycle.
      inputBuffer write_data(.clk(CLOCK_50), .reset(SW[9]), .press(~KEY[1]), .out(write));

      //FIFO has two hierarchical calls inside of it that has the daul port RAM as well as FIFO_Control
      //which has an FSM that is a queue for the RAM. This module takes in the output from paramDFF,
      //and both inputBuffer and returns 1-bit empty, full, and 8-bit bussin_out. logic empty and full
      //control LEDR[8] and LEDR[9], whiile bussin_out is passed to hexDisplay16x8.
      FIFO doing_stuff(.clk(CLOCK_50), .reset(SW[9]), .read(read), .write(write), .inputBus(bussin_in), .empty(empty), .full(full), .outputBus(bussin_out));

      //hexDisplay16x8 takes 8-bit bussin_in and bussin_out as inputs and returns the data from each
      //to HEX displays on the DE1_SoC board. bussin_in has its data displayed on HEX4-5, and bussin_out
      //has its data displayed on HEX0-1.
      hexDisplay16x8 show_it(.dataInput(bussin_in), .dataOutput(bussin_out), .led0(HEX0), .led1(HEX1), .led4(HEX4), .led5(HEX5));
endmodule
```

3.M) DE1_SoC.sv (testbench)

```
64  //DE1_SoC_testbench tests for expected, unexpected, and edgecase behavior.
65  //Reset is set high then low first. Next, a for loop is used to add data
66  //17 times to show that the system can only add 16 different sets of data.
67  //Inside the for loop writing is enable before data is written, and disable
68  //after. This fills up the memory and shows the queue has a max. After that,
69  //18 reads happen to check if the same data in comes out in the same order,
70  //and that you can't read more than you have in the memory.
71  `timescale 1ps/1ps
72  module DE1_SoC_testbench();
73        logic          CLOCK_50;
74        logic [3:0]    KEY;
75        logic [9:0]    SW;
76        logic [9:0]    LEDR;
77        logic [6:0]    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
78
79        DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .LEDR, .CLOCK_50);
80
81        parameter CLOCK_PERIOD = 100;
82        initial begin
83              CLOCK_50 <= 0;
84              forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;// Forever toggle the clock
85        end
86
87        int i;
88
89        initial begin
90           SW[7:0] <= '0; KEY[1] <= 1; KEY[0] <= 1;  repeat(1)   @(posedge    CLOCK_50);
91           SW[9] <= 1;                               repeat(1)   @(posedge    CLOCK_50);
92           SW[9] <= 0;                               repeat(1)   @(posedge    CLOCK_50);
93
94
95           for(i = 0; i < 18; i++) begin
96              KEY[1] <= 0; SW[7:0] <= i;             repeat(1)   @(posedge    CLOCK_50);
97              KEY[1] <= 1;                           repeat(1)   @(posedge    CLOCK_50);
98           end
99
100                                                    repeat(3)   @(posedge    CLOCK_50);
101
102          repeat(18) begin
103             KEY[0] <= 0;                           repeat(1)   @(posedge    CLOCK_50);
104             KEY[0] <= 1;                           repeat(1)   @(posedge    CLOCK_50);
105          end
106                                                    repeat(3)   @(posedge    CLOCK_50);
107
108          $stop; // End the simulation.
109       end
110 endmodule
```