

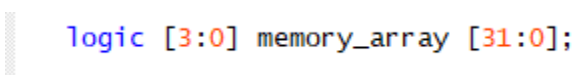
## **Lab 2 Report**

### **Procedure**

This lab was compromised of three tasks. The first task was to design a project that implements a memory unit, or in this case, a RAM unit with a single port that provides the address for both read and write operations onto the FPGA board by specifying its structure in SystemVerilog code. The second task was to create a different type of RAM unit that has two ports. One port supplies the address for reading operations and the other port supplies the address for writing operations. To do this, we used the IP catalog function from the Quartus software to create a Verilog file that represents the dual port RAM. The third and final task was to design a FIFO from the ground up. A FIFO uses a memory module to store data when written into and output data when read from. This can be done by implementing both a FIFO module and a FIFO Control module to better control the behavior of the dual port RAM and the FIFO I designed. Overall, this lab was about understanding how to design RAM units in SystemVerilog and Verilog and learning about the different implementations of RAM into the FPGA between single and dual port RAMs.

### **Task #1**

The first task was to design a single port RAM memory unit using SystemVerilog code, and then implementing it into the FPGA. My approach towards this task was rather simple. First, I needed to create a module for the RAM itself. We were given that the RAM must be of size 32x4, which means the RAM can fit 32 “words” that are 4 bits long each. Since this is a single port RAM unit, all we need is one input line for data, one input line for the single port address, and one input port for the writing enabler. All of these inputs should be synchronous with the clock, so I implemented my code in an “always\_ff” block. From here, the most important step is to create an intermediate logic that acts as the memory as shown in Figure 1. By manipulating this multi-dimensional array, we can both write into and read out of the RAM module. This intermediate logic is a multidimensional array that has 32, 4-bit data units which is sufficient for this task.



```
logic [3:0] memory_array [31:0];
```

Figure 1. Screenshot of multidimensional memory\_array

To actually implement this RAM unit, we have to code the functionalities within the “always\_ff” block. Since this is a single port RAM unit, we want to read the data within a specific address whenever we are at that address. The only time the input data is relevant is when the writing enabler is true, so there will be one if statement in the block allowing for the data at a specific address (or word location) to be rewritten. The “always\_ff” block is shown in Figure 2.

```

always_ff @(posedge clk) begin
    dataOut <= memory_array[address];
    if(write)
        memory_array[address] <= dataIn;
end

```

Figure 2. Reading and Writing from 32x4 Single-Port RAM Unit

After finishing the RAM module, I created the rest of the modules necessary to satisfy the requirements of the spec. I created a module that specifies what a HEX display should show given any 4-bit input. In this lab, every HEX display should display the hexadecimal value of the 4-bit input, so I made a module that implements this for any combination of the inputs shown in Figure 3.

```

module seg7hex (bcd, leds);
    input logic [3:0] bcd;
    output logic [6:0] leds;

    always_comb begin
        case (bcd)
            4'b0000: leds = 7'b1000000; // 0
            4'b0001: leds = 7'b1111001; // 1
            4'b0010: leds = 7'b0100100; // 2
            4'b0011: leds = 7'b0110000; // 3
            4'b0100: leds = 7'b0011001; // 4
            4'b0101: leds = 7'b0010010; // 5
            4'b0110: leds = 7'b0000010; // 6
            4'b0111: leds = 7'b1111000; // 7
            4'b1000: leds = 7'b0000000; // 8
            4'b1001: leds = 7'b0010000; // 9
            4'b1010: leds = 7'b0001000; // A
            4'b1011: leds = 7'b0000011; // B
            4'b1100: leds = 7'b1000110; // C
            4'b1101: leds = 7'b0100001; // D
            4'b1110: leds = 7'b0000110; // E
            4'b1111: leds = 7'b0001110; // F
            default: leds = 7'bx;
        endcase
    end
endmodule

```

Figure 3. Converting 4-bit binary input into hexadecimal value onto a 7-segment display

Then, I created an organizer to help me organize the HEX displays. While this module is somewhat unnecessary, it helps me because it reduces the amount of code in the top-level DE1\_SoC module. It is easy to see that this module is for HEX displays, and I can see exactly what logic is represented by which HEX display. This code is shown in appendix 1.C. After completing all of the modules, I tested the modules through ModelSim (see Task 1 in Results), and successfully implemented it into the FPGA board, completing task #1.

## Task #2

In the second task we had to implement a dual-port memory by writing SystemVerilog code and using the IP-catalog tool from the Quartus Prime software to create a dual-port RAM module in Verilog. The first step to approach this task was creating the RAM module in Verilog and observing its inputs and outputs. To do this, I followed the instructions in the spec and created the ram32x4.v module in Verilog. Similar to task 1, some of the switches of the FPGA are the address bits. However, unlike task 1, this task has two address ports. The second address port, which is now the address for the reading operation, has to count up on its own. After creating the Verilog module for the RAM unit, I created a memory initialization file (MIF) to predetermine the values stored in each address. Then I created a counter module that will increment the reading address every clock cycle. The modules so far are relatively easy, especially since one of them was created using the IP catalog.

From here, we have every input and output necessary for the HEX displays, so similar to task 1, I created a module to organize what each HEX display will represent as shown in appendix 2.C. By combining all of the modules together, I created the dual-port RAM unit that works sufficiently in simulation. However, there was one issue when implementing my code onto the FPGA board. CLOCK\_50 is a 50 MHz clock, which is quite fast for the human eye. Thus, the read address will be incrementing too fast to demonstrate the capability of the dual-port RAM unit on the board. To resolve this issue, I implemented another module that slows down the clock only for the counter. To do this, I simply create a large array that fills up with the clock. Every time the array becomes full, it represents one clock cycle. Thus, the smaller the array, the closer it resembles CLOCK\_50, but the larger the array, the slower the clock will become. The code I used for this method is shown in Figure 4. For this simulation, I used an array size of 25 that represents a 0.75 Hz clock, or about one clock cycle every 1.3 seconds. This means the counter for the read address will increment every 1.3 seconds, making it much more manageable for the demonstration of my dual-port RAM unit on the FPGA board, successfully demonstrating the requirements for task 2.

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

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

module clock_divider_testbench();
logic clk, reset;
logic [31:0] divided_clocks = 0;

clock_divider dut(.clk, .reset, .divided_clocks);

parameter CLOCK_PERIOD=100;
initial begin
    clk <= 0;
    forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
end

initial begin
    reset <= 1;
    reset <= 0; repeat(2000)@(posedge clk);
    $stop;
end
endmodule

assign clkSelect = div_clk[25]; // for board
```

Figure 4. Code for clock\_divider

### Task #3

For the third task we were tasked to design a FIFO from the ground up. A FIFO is a memory mechanism that uses a memory module to store data when written and outputs data when read from. A control module is used to organize this process. The main idea of a FIFO is that it is like a “queue” where when there is nothing written, it will not read, and when a certain number of words are written, it can only read those words in the order they were written and cannot read more than the number of words written into the queue. To approach this task, I had to split the work into two main modules. One for the FIFO itself, and one for the FIFO controller. The FIFO itself was quite simple since it is essentially just calling the FIFO controller module and the dual-port RAM module. The FIFO controller module was already called in the skeleton we received, so to call the RAM module, I had to use the IP catalog tool to create a 16x8 RAM in Verilog as requested by the spec. All I had to do was connect everything in the FIFO module between the RAM and the FIFO controller, and the FIFO module was done. The more difficult part of this task was to create the FIFO controller. To do this, I had to create an FSM that represents the overall process of the FIFO as shown in Figure 5.

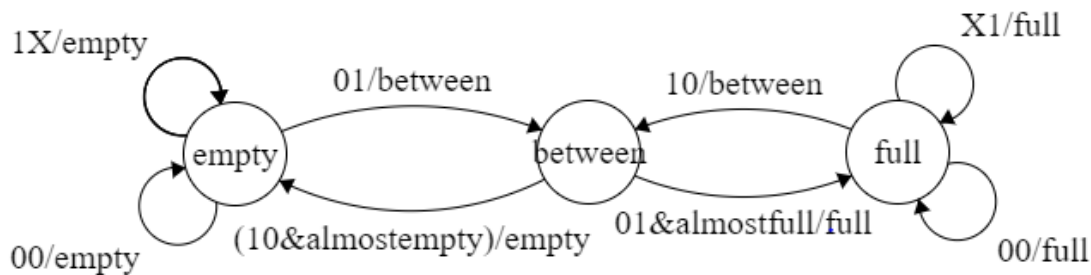


Figure 5: Finite State Machine for Task 3. Inputs are read and write, respectively.

There are three main states in the FIFO. Empty, Full, and somewhere in between. When it is empty, the read operation should not do anything and the write operation must be done to leave this state. When it is full, the write operation should not do anything and the read operation must be done to leave this state. In the designing process of the FSM, I had to add extra signals to determine when the in between state should go to full or empty states. When the read address is one bit below the write address, it is “almost empty” and when the write address is one bit below the read address, it is “almost full”. These signals will help transition between the in between states and the empty and full states. To implement this FSM in the FIFO controller module, I used a case statement. Similar to task 2, I used a counter to increment the addresses, but unlike task 2, I used the counter for both addresses. Also, I had to add a condition for the counter where it will only increment the respective addresses if they successfully read from or wrote into the RAM. The counters were called within the FIFO controller since the signals “almost full” and “almost empty” are determined by the addresses.

Now that both the FIFO and FIFO controller modules are done, it was time to put them together and display the necessary information onto the HEX displays as required by the lab spec. I used the same seg7hex module from Figure 3 to determine the hexadecimal value that should be displayed. After combining the FIFO, FIFO controller, and the displays, the simulation worked perfectly. However, similar to task 2, I ran into issues when attempting to demonstrate the FIFO behavior on the board. One issue was that the keys used to determine the write and read operations were being pressed more than once every key press. This is because CLOCK\_50 is too fast and will register many presses from one key press. To prevent this, I quickly added an input buffer module for each key. The input buffer is a great module for this task, since even if you press the key down and hold it down, it will only output true one time. This means you must press the key multiple times to register multiple outputs. A better look at this can be seen in the results task 3 section in Figure 20. After implementing the input buffer, the FIFO worked perfectly on the board, and I had no other issues. This task was completed and both the ModelSim simulation and FPGA board behavior were working as the lab spec requested.

## Top-Level Diagrams

Here are all of the top-level diagrams for each task.

Task 1 Top Level Diagram:

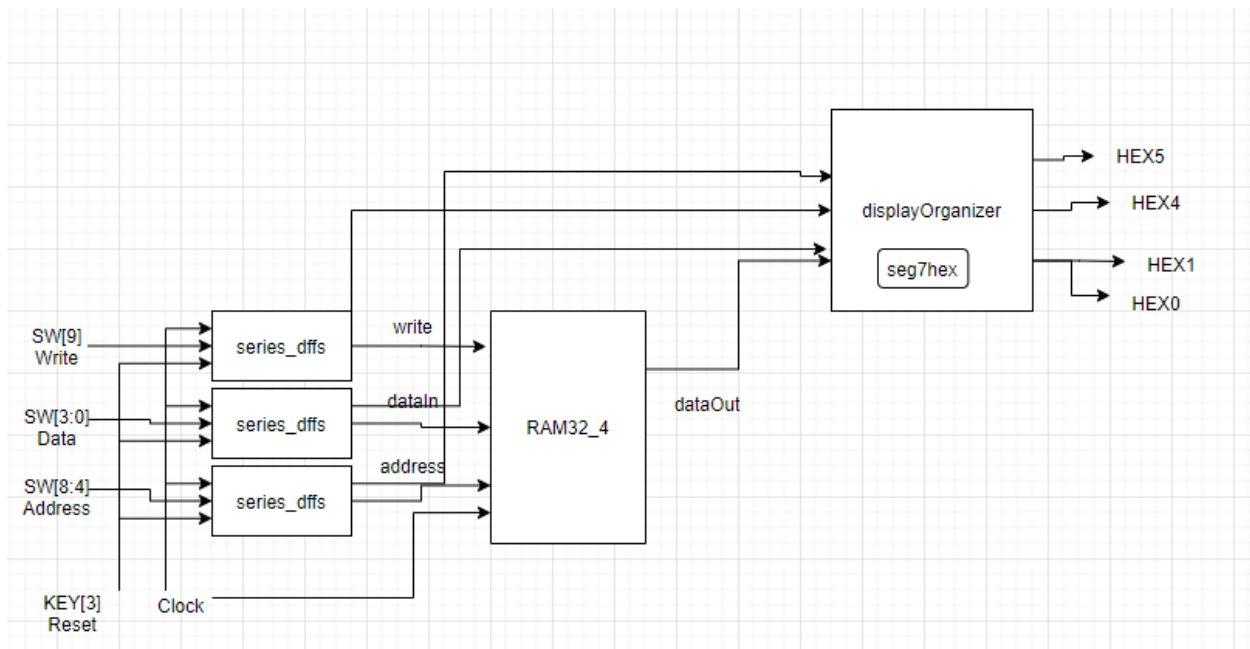


Figure 6. Top Level Block Diagram for Task 1 DE1\_SoC

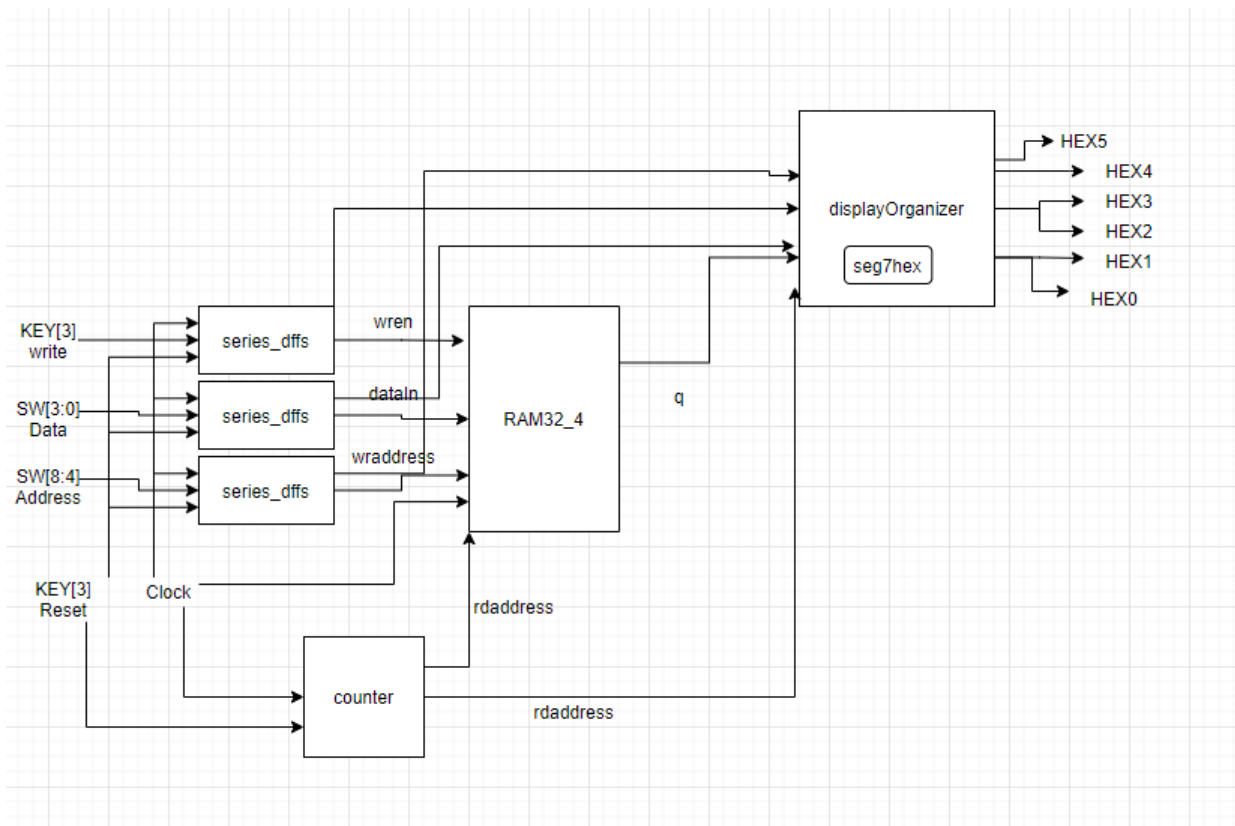


Figure 7. Top Level Block Diagram for Task 2 DE1\_SoC

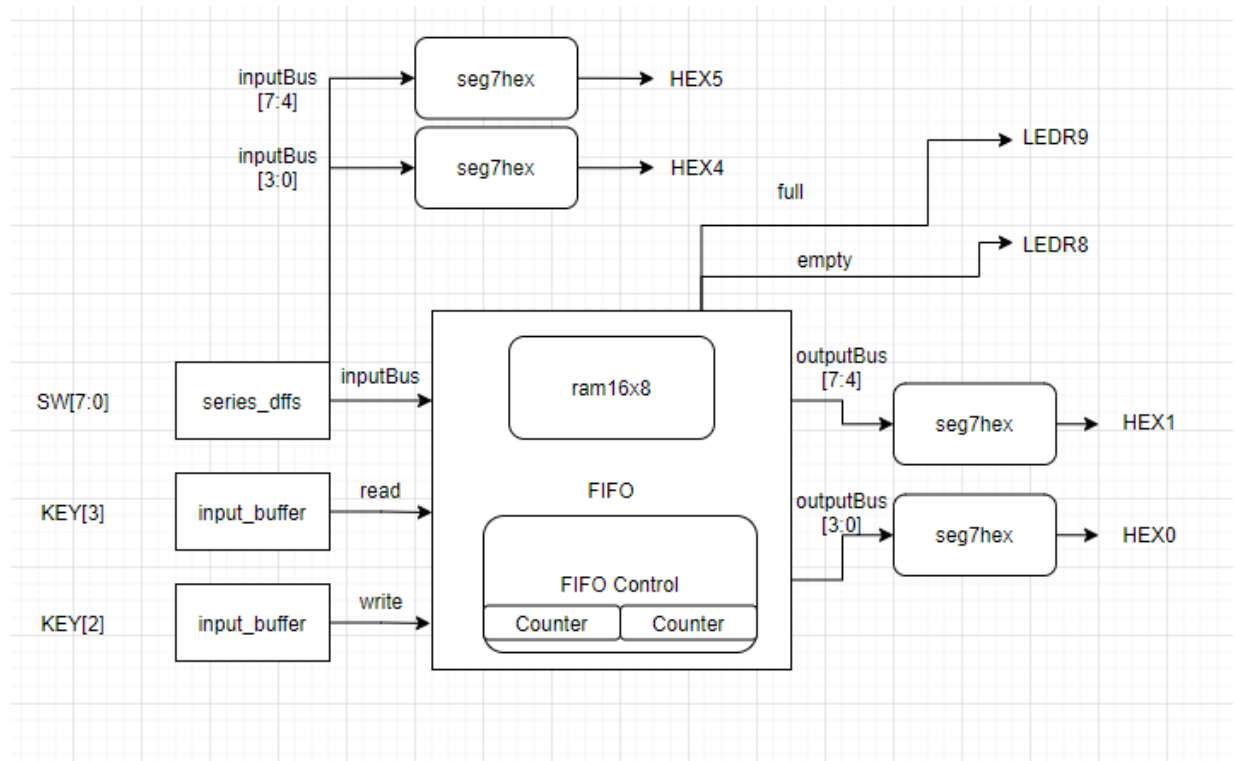


Figure 8. Top Level Block Diagram for Task 3 DE1\_SoC

## Results

### Task #1

For the first task, I created a test bench for the RAM32\_4 module to test if the memory unit is working as intended. I tested a few cases where I write into an address and read from it later to see if it functioned. I also rewrote into the addresses to make sure it can be written into multiple times, and I also read addresses I did not write into to see if it was empty as it should be. These behaviors were all successful as shown in Figure 9. As seen, the contents written into each address while “write” was 1 are properly read when “write” is 0 and the address was revisited. Any locations without values written into outputted zeros as expected.

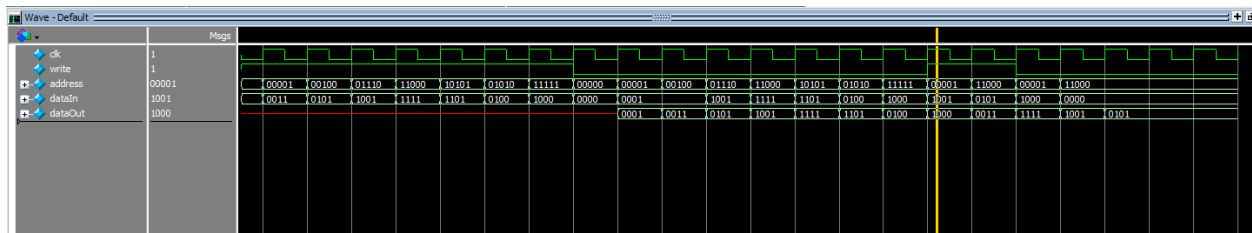


Figure 9. ModelSim Simulation for RAM32\_4 Module in Task 1

Then I created a module for the seg7hex module, to see if each 4-bit input corresponds to the correct hexadecimal value in the display as shown in Figure 10. Displays 0-F from 0000 to 1111, respectively. Simulation responded as expected.

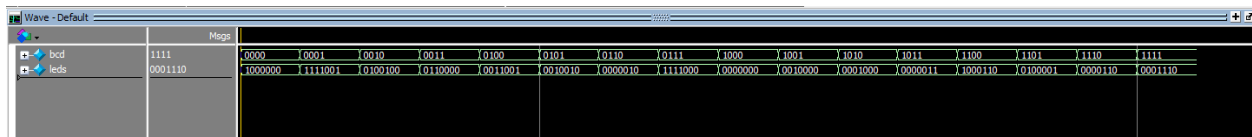


Figure 10. ModelSim Simulation for seg7hex Module used in all tasks

I also created a testbench for the displayOrganizer module, to ensure the values are being displayed by the correct HEX displays as shown in Figure 11. The simulation corresponded to the expected results.

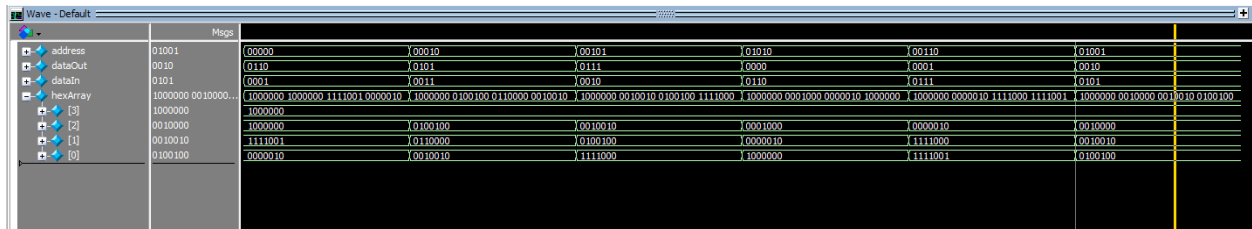


Figure 11. ModelSim Simulation for displayOrgniazer module in Task 1

In series\_dffs, I simulated the effects of two flip flops in series, and this module was used to prevent metastability from mechanical switches and keys. The simulation can be seen in figure 12.

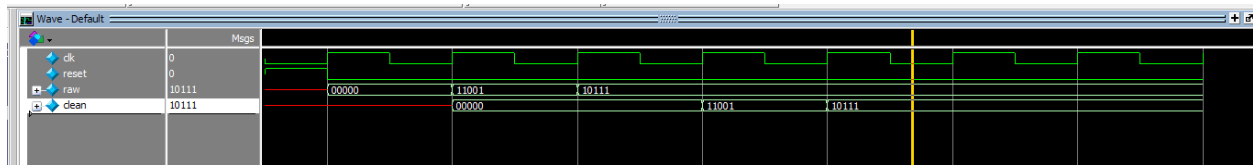


Figure 12. ModelSim Simulation for series\_dffs module used in all tasks

Lastly, I simulated the top-level entity, DE1\_SoC. This was a combination of all of the modules, and the RAM unit functioned as expected, similar to the testbench for the RAM32\_4 module. This time however, the outputs for the HEX displays are also visible and meet the expected results. For example, if the data read from an address was 1111, the HEX display should display an F. The simulation can be seen in Figure 13.

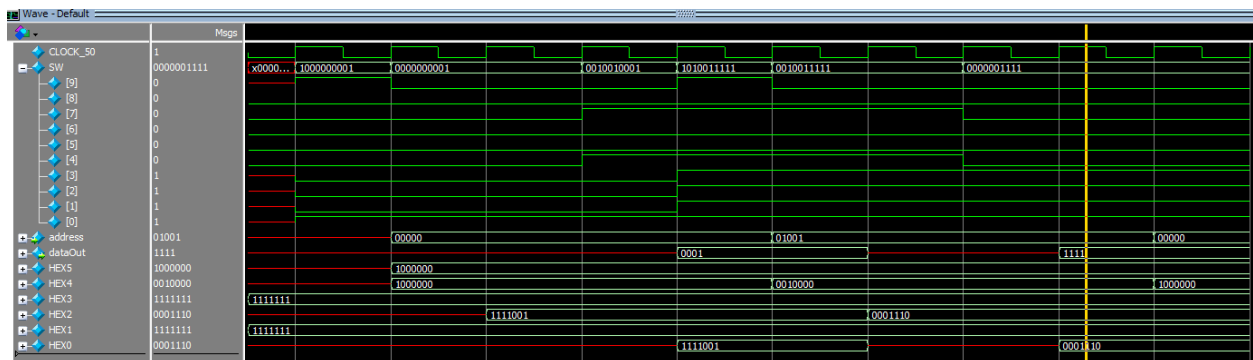


Figure 13. ModelSim Simulation for DE1\_SoC module for task 1

## Task #2

In this task, I used the exact same seg7hex and series\_dffs module as task 1, so refer to Figure 10 and 12 for those simulations from this task.

The first simulation I would like to show is the simulation for the counter module. In this simulation, the most important thing to test for was what happened after the counter reached all 1's for each bit of the output. The module functioned as expected and reset to 0 after maximum capacity, which means the read address will increment from 0 to 1111 and then back to 0 again and start counting again. The total results of the simulation can be seen in Figure 14.

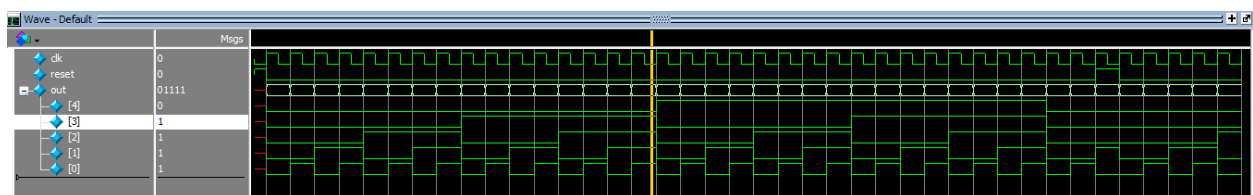


Figure 14. ModelSim Simulation for counter module in task 2

The next simulation I want to show is the clock divider module. The clock divider module is essentially just a counter, but I used the large array to “divide” the CLOCK\_50 50MHz clock. Every time the entire array is full, it will count as one clock cycle, so the larger the array, the slower the clock will be. This was necessary for my on-board demonstration, since CLOCK\_50



will increment the counter for the read address too quickly. The simulation for clock divider can be seen in Figure 15.

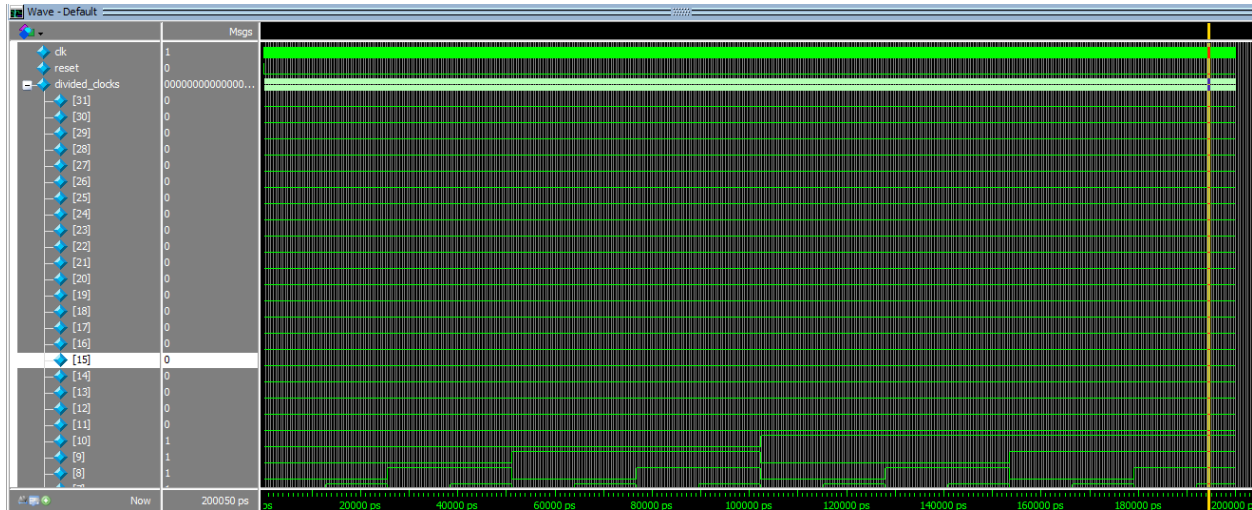


Figure 15. ModelSim Simulation for clock\_divider module in task 2

Similar to task 1, I made a display organizer module to organize all of my HEX displays. Figure 16 showcases how my inputs and outputs of the RAM are being represented with the correct HEX displays in the correct hexadecimal values.

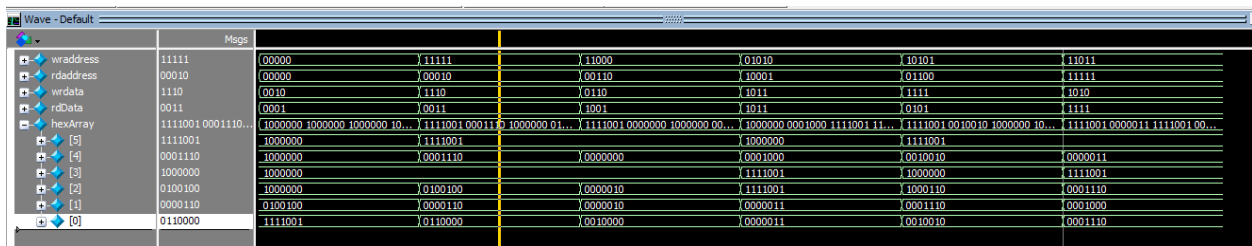


Figure 16. ModelSim Simulation for displayOrganizer module in task 2

The next simulation was the for ram32x4 module that was generated by the IP catalog. Because it was in Verilog, it took some getting used to, but I was able to successfully test the functions of the dual-port RAM. I wrote a lot of data into many addresses, then read them all to make sure a read and write operation work properly. Then, I rewrote new values into a couple of addresses and read them to make sure the RAM can be rewritten in the same addresses. Then, I read random addresses I did not write into to see if the MIF file I made for this task was being read from properly. The simulation produced results I expected and they can be seen in Figure 17

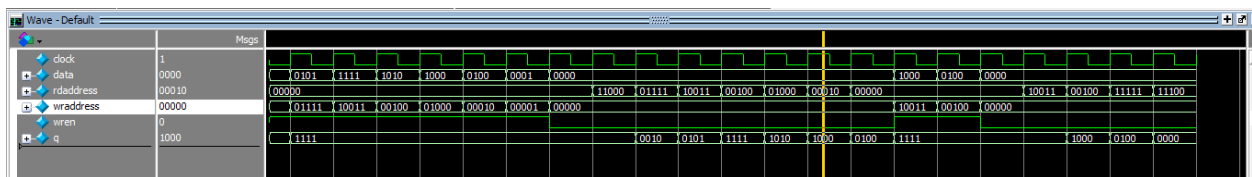


Figure 17. ModelSim Simulation for ram32x4 Verilog module in task 2

Lastly, the DE1\_SoC top-level entity was simulated. For this simulation, I wanted to see if all of the MIF values would be read properly, then I rewrote a few of the values in a few of the addresses and reread the values from these addresses. Figure 18 shows the overall simulation as well as the zoom-ins of both of these cases.

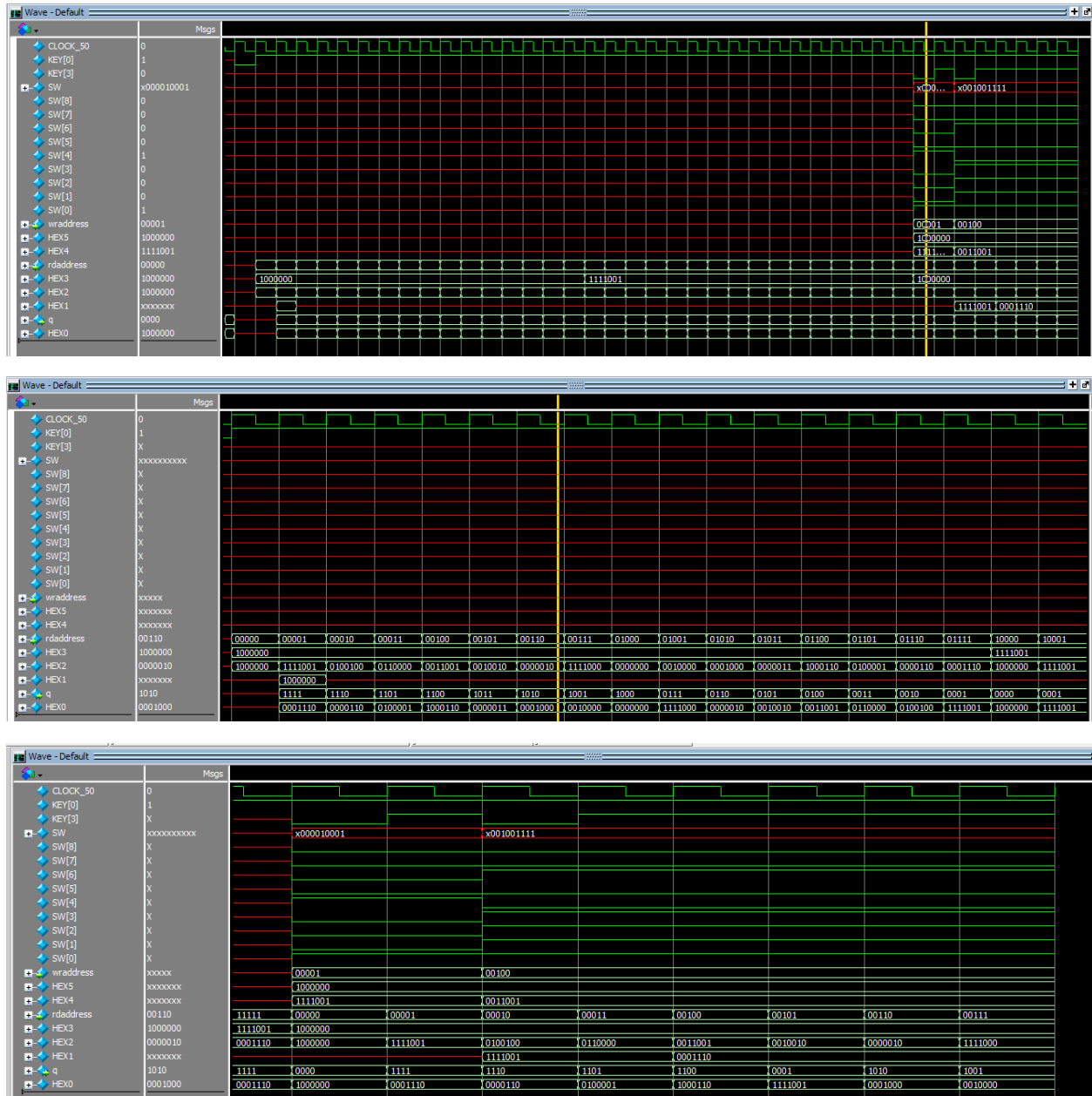


Figure 18. ModelSim Simulation for DE1\_SoC module in task 2. Top is overall waveform, middle is zoom-in for reading the MIF file, and bottom is zoom-in for rewriting a few of the address values.

### Task #3

In this task, I used the exact same seg7hex and series\_dffs module as task 1, so refer to Figure 10 and 12 for those simulations from this task.

The first simulation I would like to show is the counter module for task 3. Unlike task 2, the write and read addresses only increment when the FIFO control module tells it to, so an increment condition was added to the counter. The simulation went as expected and can be seen in Figure 19.

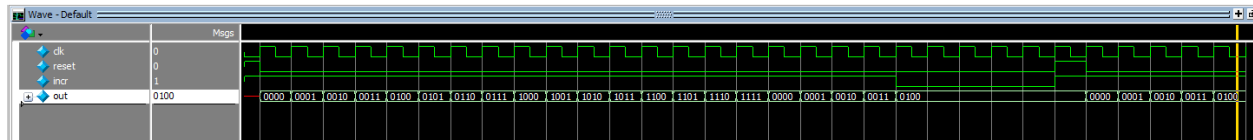


Figure 19. ModelSim Simulation for counter module in task 3

Next, I would like to show how the input buffer module works in simulation. The input buffer is designed so that when a key is pressed for longer than one clock cycle, it will only output true for one clock cycle. This behavior can be seen in Figure 20 with my successful simulation. You can see how when press is true for much longer than one clock cycle, it will only output true for one clock cycle. You can also see that when press is pressed for one clock cycle, it will still output true for one clock cycle.

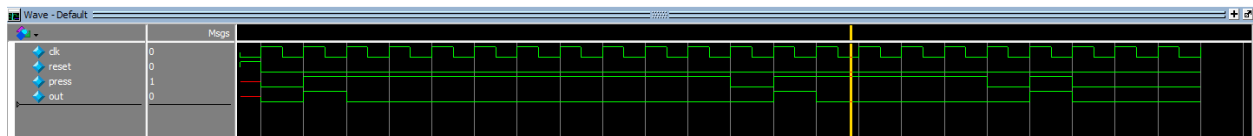


Figure 20. ModelSim Simulation for input\_buffer module in task 3

The simulation for the FIFO Control was tricky, since I had to test situations when the FIFO is full, empty, when read and write are both one, and general behavior when the FIFO is neither full nor empty. In Figure 21, you can see the full simulation I did for the FIFO Control module. You can see that the simulation works properly and changes to full and empty when it should. Also, the state returns to empty when reset is true.

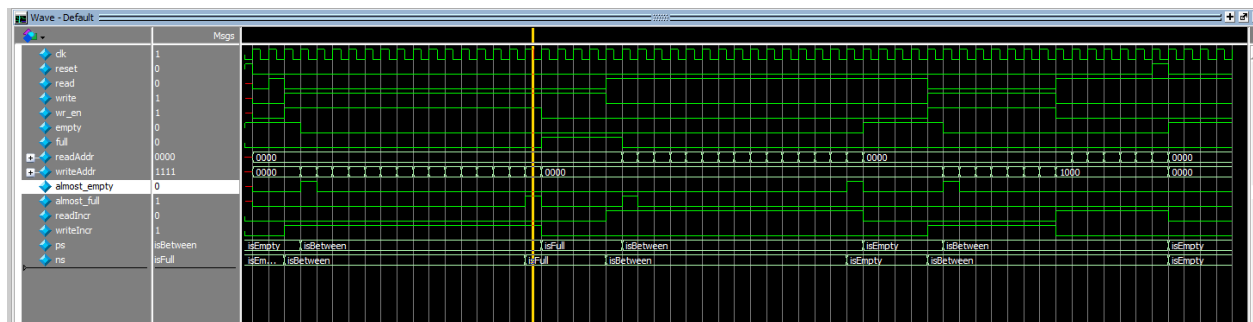


Figure 21. ModelSim Simulation for FIFO\_Control module in task 3

Similar to task 2, I had to create a dual-port RAM module using the IP catalog tool for a 16x8 RAM. This was also written in Verilog, but the simulation is nearly identical to the simulation from task 2. Instead of 32x4, this ram is 16x8, so I simply rewrote the data line so that there are 8 bits instead of 4, and the address line so that there are 4 bits instead of 5. The tests I did are the exact same as the ones I used in Figure 17. The overall simulation for the ram16x8 module can be seen in Figure 22 below.

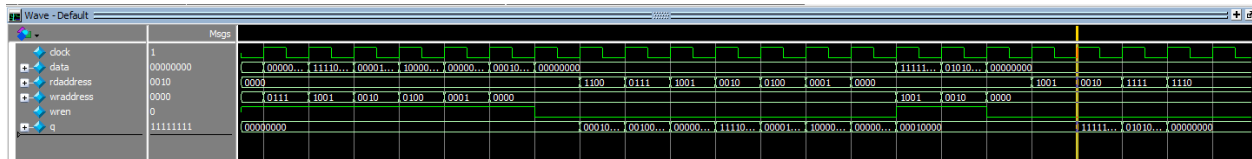


Figure 22. ModelSim Simulation for ram16x8 Verilog module for task 3

The next simulation I would like to show is naturally the FIFO module. This combines the FIFO control module with the RAM I created using the IP catalog. A few variables from FIFO control module are shown for convenience, but the main point of this simulation was to see if the outputs of the FIFO module, empty, full, and the output data, output properly in the states I described in the FIFO control module (Figure 20). The overall simulation can be seen in Figure 23.

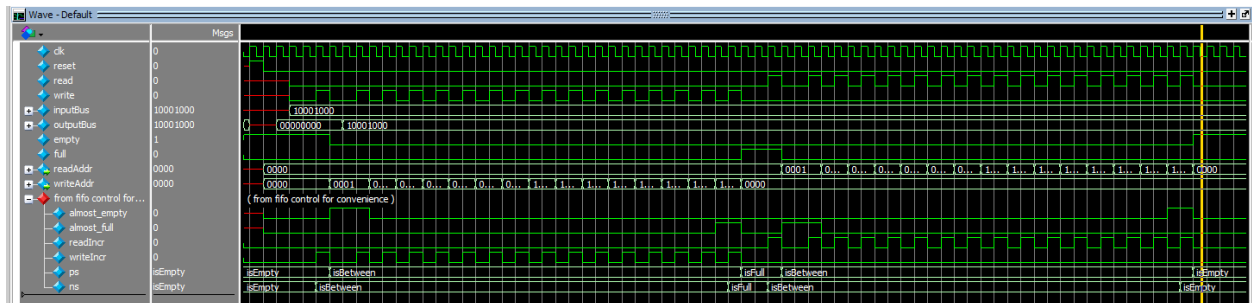


Figure 23. ModelSim Simulation for FIFO module in task 3

Lastly, the DE1\_SoC module combines all of the modules I created for this task. For this simulation, I tested the same situations as the FIFO, but observed if the HEX displays displayed the correct hexadecimal values. The simulation was successful and can be seen in Figure 24.

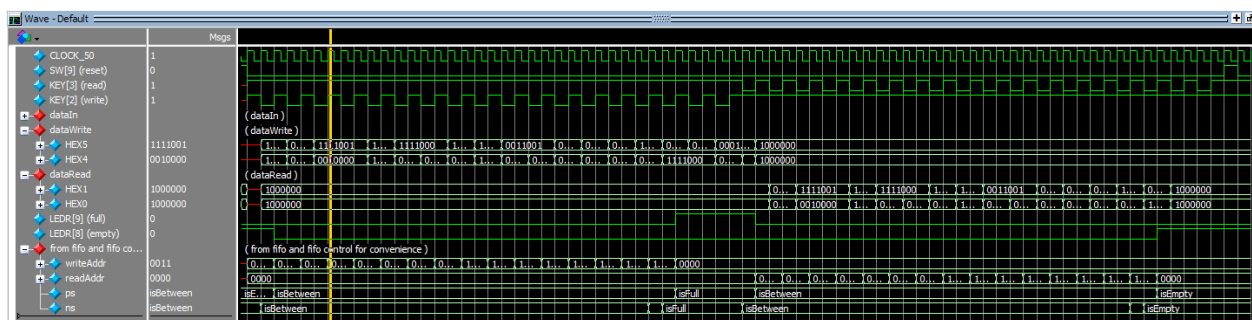


Figure 24. ModelSim Simulation for DE1\_SoC Module in task 3

## **Summary Conclusion**

The main goal of this project was to learn more about RAM units and implementing them into an FPGA board by coding them in SystemVerilog. We learned the difference between single and dual-port RAM units, and how a FIFO utilizes a dual-port RAM to make functions like a “queue” or controlling the dual-port RAM possible. This lab proved to be a challenge for me because of the new tool using the IP Catalog to create modules for the RAM units in tasks 2 and 3 in Verilog. I never coded in Verilog before, so creating the testbenches or even understanding the code that was generated was difficult. I was very excited when my task 3 project finally was working on the board. It took a lot of problem solving and looking at older material from both 271 and 371 to help me get through this lab.

Overall, my lab provided the results I wanted and I believe it is sufficient in covering the requirements of this lab. The special cases were covered for and the primary functions work perfectly.

## Appendix

### Used in ALL Tasks

#### 0.A) seg7hex.sv (code)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 3

// seg7hex takes a 4-bit input bcd and outputs a 7-bit output leds. This module takes the 4-bit input that refers to
// either the data or address that is supposed to be displayed onto a 7-segment HEX display. The name seg7hex means
// that the purpose of this module is to convert the 4-bit input into a hexadecimal value on the HEX displays.

module seg7hex (bcd, leds);
    input logic [3:0] bcd;
    output logic [6:0] leds;

    // this always_comb block uses the 4-bit input bcd as a case, and returns the value of leds in a way that
    // displays the hexadecimal value of the 4-bit input. It is clearly visible that all of the one digit
    // hexadecimal values are covered from 0 - F

    always_comb begin
        case (bcd)
            4'b0000: leds = 7'b1000000; // 0
            4'b0001: leds = 7'b1111001; // 1
            4'b0010: leds = 7'b0100100; // 2
            4'b0011: leds = 7'b0110000; // 3
            4'b0100: leds = 7'b0011001; // 4
            4'b0101: leds = 7'b0010010; // 5
            4'b0110: leds = 7'b0000010; // 6
            4'b0111: leds = 7'b1111000; // 7
            4'b1000: leds = 7'b0000000; // 8
            4'b1001: leds = 7'b0010000; // 9
            4'b1010: leds = 7'b0001000; // A
            4'b1011: leds = 7'b0000011; // B
            4'b1100: leds = 7'b1000110; // C
            4'b1101: leds = 7'b0100001; // D
            4'b1110: leds = 7'b0000110; // E
            4'b1111: leds = 7'b0001110; // F
            default: leds = 7'bx;
        endcase
    end
endmodule
```

#### 0.B) seg7hex.sv (testbench)

```
// seg7hex_testbench tests all of the cases of the 4-bit input bcd and tests to see if the output
// leds correspond to the hexadecimal value in the HEX display. The only thing to test for in this
// testbench is to go through all of the cases.

module seg7hex_testbench();
    logic [3:0] bcd;
    logic [6:0] leds;

    seg7hex dut (.bcd, .leds);

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

## 0.C) series\_dffs (code)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 3
5
6 // series_dffs takes the 1-bit inputs clk and reset and variable-bit input raw, and outputs the variable-bit output clean.
7 // The purpose of this module is that it represents two DFFs in series. When a signal goes through two DFFs, it is very
8 // difficult for it to reach metastability with other signals. I used a global parameter in this module so that multiple
9 // bit inputs are possible without having to call this module more than once.
10
11 module series_dffs # (parameter BITS = 1) (clk, reset, raw, clean);
12
13     input logic clk, reset;
14     input logic [BITS-1 : 0] raw;
15     output logic [BITS-1 : 0] clean;
16     logic [BITS-1 : 0] n1;
17
18     // always_ff block that shows the DFFs displacing the signal. The input signal goes into the first DFF and
19     // the output of the first DFF goes in as the input to the second DFF. The output of the second DFF is the
20     // output clean
21
22     always_ff @(posedge clk) begin
23         if(reset) begin
24             n1 <= '0;
25             clean <= '0;
26         end else
27             n1 <= raw;
28             clean <= n1;
29     end
30
31 endmodule
```

## 0.D) series\_dffs (testbench)

```
// series_dffs_testbench tests to see if the flip flops in series works properly even with multiple bit inputs
// I tested to see if different values will bug and output incorrectly
module series_dffs_testbench();
    logic clk, reset;
    logic [4:0] raw;
    logic [4:0] clean;

    series_dffs #(.BITS(5)) dut (.clk, .reset, .raw, .clean);

    parameter CLOCK_PERIOD=100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    initial begin
        reset <= 1;
        reset <= 0;
        raw <= 5'b00000;
        raw <= 5'b11001;
        raw <= 5'b10111;
        repeat(4) @(posedge clk);
        $stop;
    end
endmodule
```

## Task #1 Modules

### 1.A) RAM32\_4.sv (code)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 1

// RAM32_4 takes the 1-bit inputs clk and write, 4-bit input dataIn, 5-bit input address, and outputs the
// 4-bit output dataOut. The main purpose of this module is to represent a 32x4 single-port RAM using
// SystemVerilog code.

module RAM32_4 (clk, write, dataIn, address, dataOut);

    input logic clk, write;
    input logic [3:0] dataIn;
    input logic [4:0] address;
    output logic [3:0] dataOut;

    logic [3:0] memory_array [31:0];

    // this always_ff block uses the intermediate logic memory_array to
    // implement the 32x4 RAM unit. Every clock cycle, it will "read"
    // the value at each address by assigning dataOut the 4-bit data
    // inside the 32-bit array, where the location of each data is determined
    // by the address. If write is true, it will also rewrite a new data
    // determined by dataIn into the address of the memory_array.
    always_ff @(posedge clk) begin
        dataOut <= memory_array[address];
        if(write)
            memory_array[address] <= dataIn;
    end

endmodule
```

## 1.B) RAM32\_4.sv (testbench)

```
// RAM32_4_testbench tests both the expected and unexpected situations of this module. In this simulation,
// I do a writing operation on many addresses; then read all of the values written into those addresses
// with a reading operation. Then, I try rewriting a few of those addresses to test if the RAM can be rewritten
// more than once in a specific address.

module RAM32_4_testbench();
    logic clk, write;
    logic [3:0] dataIn;
    logic [4:0] address;
    logic [3:0] dataOut;

    RAM32_4 dut (.clk, .write, .dataIn, .address, .dataOut);

    parameter CLOCK_PERIOD=100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    initial begin
        //writing operation
        address <= 5'b00000; dataIn <= 4'b0001; write <= 1; @(posedge clk);
        address <= 5'b00001; dataIn <= 4'b0011; write <= 1; @(posedge clk);
        address <= 5'b00100; dataIn <= 4'b0101; write <= 1; @(posedge clk);
        address <= 5'b01110; dataIn <= 4'b1001; write <= 1; @(posedge clk);
        address <= 5'b11000; dataIn <= 4'b1111; write <= 1; @(posedge clk);
        address <= 5'b10101; dataIn <= 4'b1101; write <= 1; @(posedge clk);
        address <= 5'b01010; dataIn <= 4'b0100; write <= 1; @(posedge clk);
        address <= 5'b11111; dataIn <= 4'b1000; write <= 1; @(posedge clk);

        //reading operation
        address <= 5'b00000; dataIn <= 4'b0000; write <= 0; @(posedge clk);
        address <= 5'b00001; dataIn <= 4'b0001; write <= 0; @(posedge clk);
        address <= 5'b00100; dataIn <= 4'b0001; write <= 0; @(posedge clk);
        address <= 5'b01110; dataIn <= 4'b1001; write <= 0; @(posedge clk);
        address <= 5'b11000; dataIn <= 4'b1111; write <= 0; @(posedge clk);
        address <= 5'b10101; dataIn <= 4'b1101; write <= 0; @(posedge clk);
        address <= 5'b01010; dataIn <= 4'b0100; write <= 0; @(posedge clk);
        address <= 5'b11111; dataIn <= 4'b1000; write <= 0; @(posedge clk);

        //rewriting operation
        address <= 5'b00001; dataIn <= 4'b1001; write <= 1; @(posedge clk);
        address <= 5'b11000; dataIn <= 4'b0101; write <= 1; @(posedge clk);

        //rereading operation
        address <= 5'b00001; dataIn <= 4'b1000; write <= 0; @(posedge clk);
        address <= 5'b11000; dataIn <= 4'b0000; write <= 0; @(posedge clk);
        repeat(3)@(posedge clk);

        $stop;
    end
endmodule
```

## 1.C) displayOrganizer.sv (code)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 1

// displayOrganizer takes the 4-bit inputs dataIn and dataOut and the 5-bit input address and returns
// the 4, 7-bit outputs hexArray. The purpose of this module is to organize everything that needs to
// be displayed onto the HEX displays.

module displayOrganizer (dataIn, address, dataOut, hexArray);

    input logic [3:0] dataIn, dataOut;
    input logic [4:0] address;
    output logic [3:0][6:0] hexArray;

    // seg7hex takes the 4-bit input with 3 bits of zeros + the 4th bit of the address
    // to display the hexadecimal value onto the 4th value of hexArray, which corresponds
    // to HEX3
    seg7hex addr_displ (.bcd({3'b000, address[4]}), .leds(hexArray[3]));

    // seg7hex takes the 4-bit input which is the first 4 bits of address, and outputs
    // the third value of the hexArray which corresponds to the 7-segment display HEX4
    seg7hex addr_dispr (.bcd(address[3:0]), .leds(hexArray[2]));

    // seg7hex takes the 4-bit input dataIn and outputs the second value of hexArray
    // which corresponds to the 7-segment display HEX1
    seg7hex dataIn_disp (.bcd(dataIn), .leds(hexArray[1]));

    // seg7hex takes the 4-bit input dataOut and outputs the first value of the hexArray
    // which corresponds to the 7-segment display HEX0
    seg7hex dataOut_disp (.bcd(dataOut), .leds(hexArray[0]));

endmodule
```



### 1.D) displayOrganizer.sv (testbench)

```
// displayOrganizer_testbench is a test to see if the 4-bit values dataIn and dataOut and
// the 5-bit value address all display onto the correct HEX display and display the correct
// hexadecimal value on those displays.

module displayOrganizer_testbench();
    logic [3:0] dataIn, dataOut;
    logic [4:0] address;
    logic [3:0][6:0] hexArray;

    displayOrganizer dut (.dataIn, .address, .dataOut, .hexArray);

    initial begin
        dataIn <= 3'b001; dataOut <= 3'b110; address <= 4'b0000; #10;
        dataIn <= 3'b011; dataOut <= 3'b101; address <= 4'b0010; #10;
        dataIn <= 3'b010; dataOut <= 3'b111; address <= 4'b0101; #10;
        dataIn <= 3'b110; dataOut <= 3'b000; address <= 4'b1010; #10;
        dataIn <= 3'b111; dataOut <= 3'b001; address <= 4'b0110; #10;
        dataIn <= 3'b101; dataOut <= 3'b010; address <= 4'b1001; #10;
        $stop;
    end
endmodule
```

### 1.E) DE1\_SoC.sv (code)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 1

// DE1_SoC takes a 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW and returns the six, 7-bit outputs
// HEX5-HEX0. DE1_SoC combines all of the modules in Task 1 and uses HEX5, HEX4, HEX1, and HEX0 to display the
// values of the input and output data as well as the address.

module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, CLOCK_50);
    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    input logic [3:0] KEY;
    input logic [9:0] SW;
    input logic CLOCK_50;

    assign HEX3 = 7'b1111111;
    assign HEX1 = 7'b1111111;

    logic write;
    logic [3:0] dataIn;
    logic [4:0] address;
    logic [3:0] dataOut;

    logic clkSelect;
    // assign clkSelect = ~KEY[0]; // for on-board simulation
    assign clkSelect = CLOCK_50; // for testbench

    logic reset;
    assign reset = ~KEY[3]; // reset key added only for the dffs

    // series_dffs has a 1-bit input clkSelect, reset, and SW[9] and outputs the 1-bit output
    // write. This ensures SW[9] outputs a clean signal and prevents metastability using
    // two flip-flops in series
    series_dffs #(.BITS(1)) cleanwrite (.clk(clkSelect), .reset, .raw(SW[9]), .clean(write));

    // series_dffs has a 1-bit input clkSelect, reset, and 4-bit input SW[3:0] and outputs the 4-bit output
    // dataIn. This ensures SW[3:0] outputs a clean signal and prevents metastability using
    // two flip-flops in series
    series_dffs #(.BITS(4)) cleanData (.clk(clkSelect), .reset, .raw(SW[3:0]), .clean(dataIn));

    // series_dffs has a 1-bit input clkSelect, reset, and 5-bit input SW[8:4] and outputs the 5-bit output
    // address. This ensures SW[8:4] outputs a clean signal and prevents metastability using
    // two flip-flops in series
    series_dffs #(.BITS(5)) cleanAddr (.clk(clkSelect), .reset, .raw(SW[8:4]), .clean(address));

    // RAM32_4 has a 1-bit input clkSelect and write and 4-bit input dataIn and 5-bit input address and
    // outputs a 4-bit output dataOut
    RAM32_4 ram (.clk(clkSelect), .write, .dataIn, .address, .dataOut);

    // displayOrganizer takes the 4-bit inputs dataIn and dataOut and the 5-bit input address and
    // outputs 4, 7-bit outputs from HEX5, HEX4, HEX2, and HEX0. This organizes the displays so
    // that the respective HEX displays display the values of dataIn, dataOut, and address in
    // hexadecimal values.
    displayOrganizer displays (.dataIn, .address, .dataOut, .hexArray({HEX5, HEX4, HEX2, HEX0}));
endmodule
```

## 1.F) DE1\_SoC.sv (testbench)

```
// DE1_SoC_testbench tests all of the expected and unexpected behavior of the single-port RAM unit
// In this simulation, I wrote into many address values, and read from those addresses. Then, I rewrote
// in a few of those addresses to test if it can be rewritten. Then, I read addresses that I did not write
// anything in

module DE1_SoC_testbench();

    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    logic [3:0] KEY;
    logic [9:0] SW;
    logic CLOCK_50;

    DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);

    parameter CLOCK_PERIOD=100;
    initial begin
        CLOCK_50 <= 0;
        forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
    end
    initial begin
        SW[8:4] <= 5'b00000; @ (posedge CLOCK_50);
        SW[9] <= 1; SW[3:0] <= 4'b0001; @ (posedge CLOCK_50);
        SW[9] <= 0; repeat(2) @ (posedge CLOCK_50);
        SW[8:4] <= 5'b01001; @ (posedge CLOCK_50);
        SW[9] <= 1; SW[3:0] <= 4'b1111; @ (posedge CLOCK_50);
        SW[9] <= 0; repeat(2) @ (posedge CLOCK_50);
        SW[8:4] <= 5'b00000; repeat(3) @ (posedge CLOCK_50);

        $stop;
    end
endmodule
```

## Task #2 Modules

### 2.A) clock\_divider.sv (code and testbench)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 2

// clock_divider takes the 1-bit inputs clk and reset and outputs the 32 bit output
// divided_clocks. The purpose of this module is to slow down the clock that is inputted
// by a specific amount. If the user calls this module and uses divided_clocks[25] for example,
// the outputted clock frequency would be 0.75Hz instead of the 50MHz clock from CLOCK_50.

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

    // always_ff block that increments divided_clocks by 1. Once divided clocks is full,
    // it will output as one clock cycle. Thus, the bigger the array the slower the clock
    always_ff @(posedge clk) begin
        divided_clocks <= divided_clocks + 1;
    end
endmodule

// clock_divider_testbench tests the functionality of divided_clocks. It showcases how
// every increment divided clocks becomes closer to full.

module clock_divider_testbench();
    logic clk, reset;
    logic [31:0] divided_clocks = 0;

    clock_divider dut(.clk, .reset, .divided_clocks);

    parameter CLOCK_PERIOD=100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    initial begin
        reset <= 1; @ (posedge clk);
        reset <= 0; repeat(2000) @ (posedge clk);
        $stop;
    end
endmodule
```

## 2.B) counter.sv (code and testbench)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 2
5
6 // counter takes the 1-bit inputs clk and reset and outputs the 5-bit output out. The
7 // main purpose of this module is to increment the output variable by 1 every clock cycle
8
9
10 module counter (clk, reset, out);
11
12     input logic clk, reset;
13     output logic [4:0] out;
14
15     // always_ff block where on reset the output will reset to zero,
16     // but otherwise increment by 1 every positive edge of the clock.
17     // when output reaches max capacity, it will naturally return to zero
18     // the next increment.
19     always_ff @(posedge clk) begin
20         if(reset)
21             out <= '0;
22         else
23             out <= out + 1;
24     end
25 endmodule
26
27 // counter_testbench tests the expected and unexpected cases of this module. For this simulation,
28 // I tested what happens when the output "overflows" to see if it returns to zero and keeps counting.
29 // I also tested what happens when reset is true in the middle of counting.
30
31 module counter_testbench();
32     logic clk, reset;
33     logic [4:0] out;
34
35     counter dut (.clk, .reset, .out);
36
37     parameter CLOCK_PERIOD=100;
38     initial begin
39         clk <= 0;
40         forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
41     end
42
43     initial begin
44         //test to see if counter resets to zero after overflowing
45         reset <= 1; @ (posedge clk);
46         reset <= 0; repeat(34) @ (posedge clk);
47
48         //test to see if resetting in the middle of a count affects behavior
49         reset <= 1; @ (posedge clk);
50         reset <= 0; repeat(5) @ (posedge clk);
51         $stop;
52     end
53 endmodule
```

## 2.C) displayOrganizer.sv (code and testbench)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 2

// displayOrganizer takes the 4-bit inputs rdData and wrdata and the 5-bit inputs rdaddress and wraddress
// and returns 6, 7-bit outputs called hexArray. The purpose of this module is to organize everything that
// needs to be displayed onto the HEX displays.

module displayOrganizer (rdData, rdaddress, wrdata, wraddress, hexArray);

    input logic [3:0] rdData, wrdata;
    input logic [4:0] rdaddress, wraddress;
    output logic [5:0][6:0] hexArray;

    logic [3:0] rdaddrL, rdaddrR, wraddrL, wraddrR;

    // seg7hex takes the 4-bit input rdData and outputs the first value of hexArray
    // which corresponds to the 7-segment display HEX0
    seg7hex rdData_disp (.bcd(rdData), .leds(hexArray[0]));

    // seg7hex takes the 4-bit input wrdata and outputs the second value of hexArray
    // which corresponds to the 7-segment display HEX1
    seg7hex wrdata_disp (.bcd(wrdata), .leds(hexArray[1]));

    // seg7hex takes the 4-bit input which is the first 4 bits of rdaddress, and outputs
    // the third value of the hexArray which corresponds to the 7-segment display HEX2
    seg7hex rdaddrR_disp (.bcd(rdaddress[3:0]), .leds(hexArray[2]));

    // seg7hex takes the 4-bit input with 3 bits of zeros + the 5th bit of rdaddress
    // to display the hexadecimal value onto the fourth value of hexArray, which corresponds
    // to HEX3
    seg7hex rdaddrL_disp (.bcd({3'b000, rdaddress[4]}), .leds(hexArray[3]));

    // seg7hex takes the 4-bit input which is the first 4 bits of wraddress, and outputs
    // the fifth value of the hexArray which corresponds to the 7-segment display HEX4
    seg7hex wraddrR_disp (.bcd(wraddress[3:0]), .leds(hexArray[4]));

    // seg7hex takes the 4-bit input with 3 bits of zeros + the 5th bit of wraddress
    // to display the hexadecimal value onto the fourth value of hexArray, which corresponds
    // to HEX5
    seg7hex wraddrL_disp (.bcd({3'b000, wraddress[4]}), .leds(hexArray[5]));

endmodule

// displayOrganizer_testbench is a test to see if the 4-bit values rdData and wrdata and
// the 5-bit values rdaddress and wraddress all display onto the correct HEX display and
// display the correct hexadecimal value on those displays.

module displayOrganizer_testbench();
    logic [3:0] rdData, wrdata;
    logic [4:0] rdaddress, wraddress;
    logic [5:0][6:0] hexArray;

    displayOrganizer dut (.rdData, .rdaddress, .wrdata, .wraddress, .hexArray);

    initial begin
        rdData <= 4'b0001; wrdata <= 4'b0010; rdaddress <= 5'b00000; wraddress <= 5'b00000; #10;
        rdData <= 4'b0011; wrdata <= 4'b1110; rdaddress <= 5'b00010; wraddress <= 5'b11111; #10;
        rdData <= 4'b1001; wrdata <= 4'b0110; rdaddress <= 5'b00110; wraddress <= 5'b11000; #10;
        rdData <= 4'b1011; wrdata <= 4'b1011; rdaddress <= 5'b10001; wraddress <= 5'b01010; #10;
        rdData <= 4'b0101; wrdata <= 4'b1111; rdaddress <= 5'b01100; wraddress <= 5'b10101; #10;
        rdData <= 4'b1111; wrdata <= 4'b1010; rdaddress <= 5'b11111; wraddress <= 5'b11011; #10;
        $stop;
    end
endmodule
```

## 2.D) ram32x4.v (code)

```

41 // synopsys translate_off
42 timescale 1 ps / 1 ps
43 // synopsys translate_on
44 module ram32x4 (
45     clock,
46     data,
47     rdaddress,
48     wraddress,
49     wren,
50     q);
51
52     input    clock;
53     input [3:0] data;
54     input [4:0] rdaddress;
55     input [4:0] wraddress;
56     input    wren;
57     output [3:0] q;
58
59 `ifndef ALTERA_RESERVED_QIS
60 // synopsys translate_off
61 endif
62     tri1    clock;
63     tri0    wren;
64 `ifndef ALTERA_RESERVED_QIS
65 // synopsys translate_on
66 endif
67
68     wire [3:0] sub_wire0;
69     wire [3:0] q = sub_wire0[3:0];
70
71     altsyncram altsyncram_component (
72         .address_a (wraddress),
73         .address_b (rdaddress),
74         .clock0 (clock),
75         .data_a (data),
76         .wren_a (wren),
77         .q_b (sub_wire0),
78         .aclr0 (1'b0),
79         .aclr1 (1'b0),
80         .addressstall_a (1'b0),
81         .addressstall_b (1'b0),
82         .byteena_a (1'b1),
83         .byteena_b (1'b1),
84         .clock1 (1'b1),
85         .clocken0 (1'b1),
86         .clocken1 (1'b1),
87         .clocken2 (1'b1),
88         .clocken3 (1'b1),
89         .data_b ({4{1'b1}}),
90         .eccstatus (),
91         .q_a (),
92         .rden_a (1'b1),
93         .rden_b (1'b1),
94         .wren_b (1'b0));
95
96     defparam
97         altsyncram_component.address_aclr_b = "NONE",
98         altsyncram_component.address_reg_b = "CLOCK0",
99         altsyncram_component.clock_enable_input_a = "BYPASS",
100         altsyncram_component.clock_enable_input_b = "BYPASS",
101         altsyncram_component.clock_enable_output_b = "BYPASS",
102         altsyncram_component.init_file = "ram32x4.mif",
103         altsyncram_component.intended_device_family = "cyclone v",
104         altsyncram_component.lpm_type = "altsyncram",
105         altsyncram_component.numwords_a = 32,
106         altsyncram_component.numwords_b = 32,
107         altsyncram_component.operation_mode = "DUAL_PORT",
108         altsyncram_component.outdata_aclr_b = "NONE",
109         altsyncram_component.outdata_reg_b = "UNREGISTERED",
110         altsyncram_component.power_up_uninitialized = "FALSE",
111         altsyncram_component.ram_block_type = "M10K",
112         altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
113         altsyncram_component.widthad_a = 5,
114         altsyncram_component.widthad_b = 5,
115         altsyncram_component.width_a = 4,
116         altsyncram_component.width_b = 4,
117         altsyncram_component.width_byteena_a = 1;
118
119 endmodule

```

## 2.E) ram32x4.v (testbench)

```
// ram32x4_testbench tests both the expected and unexpected cases for this module. In this simulation,
// I tested writing into many modules and then reading from the same addresses I wrote into. Then,
// I tried rewriting into some of those addresses and rereading to see if they can be overwritten more
// than one time. Then, I tried reading addresses I did not touch to see if it reads from the MIF file or
// is initialized to zero when unwritten.

`timescale 1 ps / 1 ps
module ram32x4_testbench();
    reg clock;
    reg [3:0] data;
    reg [4:0] rdaddress;
    reg [4:0] wraddress;
    reg wren;
    wire [3:0] q;

    ram32x4 dut (.clock(clock), .data(data), .rdaddress(rdaddress), .wraddress(wraddress), .wren(wren), .q(q));

    parameter CLOCK_PERIOD=100;
    initial begin
        clock <= 0;
        forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
    end

    initial begin
        //test writing operation
        rdaddress <= 5'b00000;
        data <= 4'b0010; wraddress <= 5'b11000; wren <= 1; @(posedge clock);
        data <= 4'b0101; wraddress <= 5'b01111; wren <= 1; @(posedge clock);
        data <= 4'b1111; wraddress <= 5'b10011; wren <= 1; @(posedge clock);
        data <= 4'b1010; wraddress <= 5'b00100; wren <= 1; @(posedge clock);
        data <= 4'b1000; wraddress <= 5'b01000; wren <= 1; @(posedge clock);
        data <= 4'b0100; wraddress <= 5'b00010; wren <= 1; @(posedge clock);
        data <= 4'b0001; wraddress <= 5'b00001; wren <= 1; @(posedge clock);
        data <= 4'b0000; wraddress <= 5'b00000; wren <= 0; @(posedge clock);

        //test reading operation
        rdaddress <= 5'b11000; @(posedge clock);
        rdaddress <= 5'b01111; @(posedge clock);
        rdaddress <= 5'b10011; @(posedge clock);
        rdaddress <= 5'b00100; @(posedge clock);
        rdaddress <= 5'b01000; @(posedge clock);
        rdaddress <= 5'b00010; @(posedge clock);
        rdaddress <= 5'b00000; @(posedge clock);

        //test rewriting, rereading, and reading empty spaces
        data <= 4'b1000; wraddress <= 5'b10011; wren <= 1; @(posedge clock);
        data <= 4'b0100; wraddress <= 5'b00100; wren <= 1; @(posedge clock);
        data <= 4'b0000; wraddress <= 5'b00000; wren <= 0; @(posedge clock);
        rdaddress <= 5'b10011; @(posedge clock);
        rdaddress <= 5'b00100; @(posedge clock);

        rdaddress <= 5'b11111; @(posedge clock);
        rdaddress <= 5'b11100; @(posedge clock);

        $stop;
    end
endmodule
```



## 2.F) DE1\_SoC.sv (code)

```

1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 2
5
6 // DE1_SoC takes a 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW and returns the six, 7-bit outputs
7 // HEX5-HEX0. DE1_SoC combines all of the modules in Task 2 and uses HEX5-HEX0 to display the values of the
8 // read and write data and read and write addresses in hexadecimal values. A counter is implemented to increment
9 // the read address.
10
11
12 module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, CLOCK_50);
13
14     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
15     input logic [3:0] KEY;
16     input logic [9:0] SW;
17     input logic CLOCK_50;
18
19     logic [31:0] div_clk;
20     // clock_divider takes the 1-bit inputs clk and reset and outputs the 32 bit output
21     // div_clk. The purpose of this module is to slow down CLOCK_50 for on-board demonstration.
22     // using the value div_clk[25] will make the clock 0.75 Hz instead of 50MHz like CLOCK_50
23     clock_divider cdiv (.clk(CLOCK_50),
24                        .reset(~KEY[0]),
25                        .divided_clocks(div_clk));
26
27     logic clkselect; // for easy switching between clocks
28     assign clkselect = CLOCK_50; // for simulation
29     //assign clkselect = div_clk[25]; // for board
30
31     logic [4:0] rdaddress;
32     logic [3:0] rdData;
33     logic [4:0] wraddress;
34     logic [3:0] dataIn;
35     logic wren;
36
37     // series_dffs has a 1-bit input CLOCK_50, KEY[0], and 5-bit input SW[8:4] and outputs the 5-bit output
38     // writeAddr. This ensures SW[8:4] outputs a clean signal and prevents metastability using two flip-flops in series
39     series_dffs #(0.BITS(5)) cleanAddr (.clk(CLOCK_50), .reset(~KEY[0]), .raw(SW[8:4]), .clean(writeAddr));
40
41     // series_dffs has a 1-bit input CLOCK_50, KEY[0], and 4-bit input SW[3:0] and outputs the 4-bit output
42     // dataIn. This ensures SW[3:0] outputs a clean signal and prevents metastability using
43     // two flip-flops in series
44     series_dffs #(0.BITS(4)) cleanData (.clk(CLOCK_50), .reset(~KEY[0]), .raw(SW[3:0]), .clean(dataIn));
45
46     // series_dffs has a 1-bit input CLOCK_50, KEY[0], and KEY[3] and outputs the 1-bit output
47     // wren. This ensures SW[9] outputs a clean signal and prevents metastability using two flip-flops in series
48     series_dffs #(0.BITS(1)) cleanwren (.clk(CLOCK_50), .reset(~KEY[0]), .raw(~KEY[3]), .clean(wren));
49
50
51     // counter uses the 1-bit inputs clkselect and KEY[0] and outputs the 5-bit output rdaddress.
52     // the purpose of this module is to increment the read address
53     counter readAddr (.clk(clkselect), .reset(~KEY[0]), .out(rdaddress));
54
55     // ram32x4 is a verilog file that uses 1-bit input CLOCK_50, 4-bit input dataIn, 5-bit inputs rdaddress and wraddress,
56     // and 1-bit input wren to output the 4-bit output rdData. The purpose of this module is that it is the 32x4 dual-port
57     // RAM that the IP catalog generated and is the core of this task
58     ram32x4 ram (.clock(CLOCK_50), .data(dataIn), .rdaddress, .wraddress(SW[8:4]), .wren, .q(rdData));
59
60     // displayorganizer takes the 4-bit inputs rdData and wrData, and the 5-bit inputs rdaddress and wraddress and outputs
61     // 6, 7-bit outputs HEX5-HEX0. The purpose of this module is to organize the displays so that the inputs will be displayed
62     // in their respective HEX displays and the proper hexadecimal values will be displayed
63     displayorganizer displays (.rdData, .rdaddress, .wrData(dataIn), .wraddress(SW[8:4]), .hexArray({HEX5, HEX4, HEX3, HEX2, HEX1, HEX0}));
64
65 endmodule

```

## 2.G) DE1\_SoC.sv (testbench)

```

// DE1_SoC_testbench tests the expected and unexpected situations for this task. In this simulation,
// I tested to see if I write nothing the MIF values will be read properly, then I rewrote values into
// some of the addresses to see if their values can be overwritten

`timescale 1 ps / 1 ps
module DE1_SoC_testbench();

    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    logic [3:0] KEY;
    logic [9:0] SW;
    logic CLOCK_50;

    DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);
    parameter CLOCK_PERIOD=100;
    initial begin
        CLOCK_50 <= 0;
        forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
    end
    initial begin
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50);
        KEY[0] <= 1; repeat(1) @(posedge CLOCK_50); // Always reset FSMs at start
        KEY[3] <= 0; SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; repeat(1) @(posedge CLOCK_50);
        KEY[3] <= 1; repeat(1) @(posedge CLOCK_50);
        KEY[3] <= 0; SW[8:4] <= 5'b00100; SW[3:0] <= 4'b1111; repeat(1) @(posedge CLOCK_50);
        KEY[3] <= 1; repeat(1) @(posedge CLOCK_50);
        repeat(4) @(posedge CLOCK_50);

        $stop;
    end
endmodule

```

## Task #3 Modules

### 3.A) counter.sv (code and testbench)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 3
5
6 // counter takes the 1-bit inputs clk, reset and incr and outputs the 5-bit output out. The
7 // main purpose of this module is to increment the output variable by 1 every clock cycle only
8 // when incr is true. If incr is not true, the count will not increment the output.
9
10 module counter #(parameter DEPTH = 4) (clk, reset, incr, out);
11
12     input logic clk, reset, incr;
13     output logic [DEPTH-1:0] out;
14
15     // always_ff block where on reset the output will reset to zero,
16     // but otherwise increment by 1 every positive edge of the clock so
17     // long as incr is true. If incr is not true, out will not increment
18     // when output reaches max capacity, it will naturally return to zero
19     // the next increment.
20     always_ff @(posedge clk) begin
21         if(reset)
22             out <= '0;
23         else if(incr)
24             out <= out + 1'b1;
25         else
26             out <= out;
27     end
28 endmodule
29
30 // counter_testbench tests the expected and unexpected cases of this module. For this simulation,
31 // I tested what happens when the output "overflows" to see if it returns to zero and keeps counting.
32 // I also tested what happens when reset is true in the middle of counting. I also tested if counter
33 // does not count when incr is not true.
34
35 module counter_testbench();
36     logic clk, reset, incr;
37     logic [3:0] out;
38
39     counter dut (.clk, .reset, .incr, .out);
40
41     parameter CLOCK_PERIOD=100;
42     initial begin
43         clk <= 0;
44         forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
45     end
46
47     initial begin
48         //test to see if counter resets to zero after overflowing
49         reset <= 1; incr <= 1; @ (posedge clk);
50         reset <= 0; repeat(20) @ (posedge clk);
51
52         //test to see if counter does not move when incr is 0;
53         incr <= 0; repeat(5) @ (posedge clk);
54
55         //test to see if resetting in the middle of a count affects behavior
56         reset <= 1; incr <= 1; @ (posedge clk);
57         reset <= 0; repeat(5) @ (posedge clk);
58         $stop;
59     end
60 endmodule
61
```



### 3.B) input\_buffer.sv (code and testbench)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 3

// input_buffer takes the 1-bit input clk, reset, and press and outputs the 1-bit output
// out. The purpose of this module is to prevent keys on the FPGA board from outputting
// more than once in one press.

module input_buffer(clk, reset, press, out);
    input logic clk, reset, press;
    output logic out;
    logic PS, NS;

    // this always_comb block is used to determine when out should be true.
    // the next state is determined by press, so only the first time when
    // ps is not press will out be true when press is true. So long as
    // press stays true, the present state from then on will be press,
    // making out = 0
    always_comb begin
        NS = press;
        out = (~PS & press);
    end

    // This always_ff block is used to reset the present state of the FSM to 0 when reset is true,
    // and to update the present state to the next state every positive edge of the clock otherwise.
    always_ff @(posedge clk) begin
        if(reset)
            PS <= 0;
        else
            PS <= NS;
        end

    end
endmodule

// input_buffer_testbench tests every case this module can face. When press is true for over 10 clock cycles,
// when press is true for 5 clock cycles, and when press is true for 1 clock cycle. For every case, out should
// only be true for one clock cycle.

module input_buffer_testbench();
    logic clk, reset, press;
    logic out;

    input_buffer dut (.clk, .reset, .press, .out);

    parameter CLOCK_PERIOD=100;
    initial begin
        clk <= 1'b0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk;
    end

    initial begin
        reset <= 1; @ (posedge clk);
        reset <= 0; press <= 0; @ (posedge clk);
        press <= 1; repeat(10) @ (posedge clk);
        press <= 0; @ (posedge clk);
        press <= 1; repeat(5) @ (posedge clk);
        press <= 0; @ (posedge clk);
        press <= 1; @ (posedge clk);
        press <= 0; repeat(3) @ (posedge clk);

        $stop;
    end
endmodule
```

### 3.C) ram16x8.v (code)

```

43 `timescale 1 ps / 1 ps
44 // synopsys translate_on
45 module ram16x8 (
46     clock,
47     data,
48     rdaddress,
49     wraddress,
50     wren,
51     q);
52
53     input clock;
54     input [7:0] data;
55     input [3:0] rdaddress;
56     input [3:0] wraddress;
57     input wren;
58     output [7:0] q;
59 `ifndef ALTERA_RESERVED_QIS
60 // synopsys translate_off
61 `endif
62     tri1 clock;
63     tri0 wren;
64 `ifndef ALTERA_RESERVED_QIS
65 // synopsys translate_on
66 `endif
67
68     wire [7:0] sub_wire0;
69     wire [7:0] q = sub_wire0[7:0];
70
71     altsyncram altsyncram_component (
72         .address_a (wraddress),
73         .address_b (rdaddress),
74         .clock0 (clock),
75         .data_a (data),
76         .wren_a (wren),
77         .q_b (sub_wire0),
78         .aclr0 (1'b0),
79         .aclr1 (1'b0),
80         .addressstall_a (1'b0),
81         .addressstall_b (1'b0),
82         .byteena_a (1'b1),
83         .byteena_b (1'b1),
84         .clock1 (1'b1),
85         .clocken0 (1'b1),
86         .clocken1 (1'b1),
87         .clocken2 (1'b1),
88         .clocken3 (1'b1),
89         .data_b ({8{1'b1}}),
90         .eccstatus (),
91         .q_a (),
92         .rden_a (1'b1),
93         .rden_b (1'b1),
94         .wren_b (1'b0));
95
96     defparam
97         altsyncram_component.address_aclr_b = "NONE",
98         altsyncram_component.address_reg_b = "CLOCK0",
99         altsyncram_component.clock_enable_input_a = "BYPASS",
100         altsyncram_component.clock_enable_input_b = "BYPASS",
101         altsyncram_component.clock_enable_output_b = "BYPASS",
102         altsyncram_component.init_file = "ram16x8.mif",
103         altsyncram_component.intended_device_family = "Cyclone V",
104         altsyncram_component.lpm_type = "altsyncram",
105         altsyncram_component.numwords_a = 16,
106         altsyncram_component.numwords_b = 16,
107         altsyncram_component.operation_mode = "DUAL_PORT",
108         altsyncram_component.outdata_aclr_b = "NONE",
109         altsyncram_component.operation_mode = "DUAL_PORT",
110         altsyncram_component.outdata_aclr_b = "NONE",
111         altsyncram_component.outdata_reg_b = "UNREGISTERED",
112         altsyncram_component.power_up_uninitialized = "FALSE",
113         altsyncram_component.ram_block_type = "M10K",
114         altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
115         altsyncram_component.width_a = 4,
116         altsyncram_component.width_b = 4,
117         altsyncram_component.width_a = 8,
118         altsyncram_component.width_b = 8,
119         altsyncram_component.width_byteena_a = 1;
120
121 endmodule

```

### 3.D) ram16x8.v (testbench)

```
// ram16x8_testbench tests both the expected and unexpected cases for this module. In this simulation,
// I tested writing into many modules and then reading from the same addresses I wrote into. Then,
// I tried rewriting into some of those addresses and rereading to see if they can be overwritten more
// than one time. Then, I tried reading addresses I did not touch to see if it reads from the MIF file or
// is initialized to zero when unwritten.

`timescale 1 ps / 1 ps
module ram16x8_testbench();
    reg clock;
    reg [7:0] data;
    reg [3:0] rdaddress;
    reg [3:0] wraddress;
    reg wren;
    wire [7:0] q;

    ram16x8 dut (.clock(clock), .data(data), .rdaddress(rdaddress), .wraddress(wraddress), .wren(wren), .q(q));

    parameter CLOCK_PERIOD=100;
    initial begin
        clock <= 0;
        forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
    end

    initial begin
        //test writing operation
        rdaddress <= 4'b0000;
        data <= 8'b00100000; wraddress <= 4'b1100; wren <= 1; @(posedge clock);
        data <= 8'b00000101; wraddress <= 4'b0111; wren <= 1; @(posedge clock);
        data <= 8'b11110000; wraddress <= 4'b1001; wren <= 1; @(posedge clock);
        data <= 8'b00001010; wraddress <= 4'b0010; wren <= 1; @(posedge clock);
        data <= 8'b10000000; wraddress <= 4'b0100; wren <= 1; @(posedge clock);
        data <= 8'b00000100; wraddress <= 4'b0001; wren <= 1; @(posedge clock);
        data <= 8'b00010000; wraddress <= 4'b0000; wren <= 1; @(posedge clock);
        data <= 8'b00000000; wraddress <= 4'b0000; wren <= 0; @(posedge clock);

        //test reading operation
        rdaddress <= 4'b1100; @(posedge clock);
        rdaddress <= 4'b0111; @(posedge clock);
        rdaddress <= 4'b1001; @(posedge clock);
        rdaddress <= 4'b0010; @(posedge clock);
        rdaddress <= 4'b0100; @(posedge clock);
        rdaddress <= 4'b0001; @(posedge clock);
        rdaddress <= 4'b0000; @(posedge clock);

        //test rewriting, rereading, and reading empty spaces
        data <= 8'b11111111; wraddress <= 4'b1001; wren <= 1; @(posedge clock);
        data <= 8'b01010011; wraddress <= 4'b0010; wren <= 1; @(posedge clock);
        data <= 8'b00000000; wraddress <= 4'b0000; wren <= 0; @(posedge clock);
        rdaddress <= 4'b1001; @(posedge clock);
        rdaddress <= 4'b0010; @(posedge clock);

        rdaddress <= 4'b1111; @(posedge clock);
        rdaddress <= 4'b1110; @(posedge clock);
        rdaddress <= 4'b1110; @(posedge clock);

        $stop;
    end
endmodule
```

### 3.E) FIFO\_Control.sv (code)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 3
5
6 // FIFO_Control takes the 1-bit inputs clk, reset, read, and write and outputs the 1-bit outputs wr_en, empty, and full
7 // and outputs the variable-bit outputs readAddr and writeAddr. In this lab, these values are set to 4 bits since the
8 // global parameter depth is set to 4. The purpose of this module is to control the overall behavior of the FIFO and
9 // keep things organized. The FIFO_Control makes sure the FIFO behaves a certain way when it is full or empty, and allows
10 // for data to be written into or read from the RAM.
11
12 module FIFO_Control #(parameter depth = 4)(clk, reset, read, write, wr_en, empty, full, readAddr, writeAddr);
13
14 input logic clk, reset;
15 input logic read, write;
16 output logic wr_en;
17 output logic empty, full;
18 output logic [depth-1:0] readAddr, writeAddr;
19
20 logic almost_empty, almost_full;
21 logic readIncr, writeIncr;
22
23 assign almost_empty = (writeAddr == readAddr + 1'b1);
24 assign almost_full = (readAddr == writeAddr + 1'b1);
25 assign wr_en = (write & ~full);
26
27 counter rdaddr (.clk, .reset, .incr(readIncr), .out(readAddr));
28 counter wraddr (.clk, .reset, .incr(writeIncr), .out(writeAddr));
29
30 // this always_comb block implements the finite state machine I made for the FIFO_Control module
31 // Depending on the values of read, write, almost_empty, and almost_full, the states will transition
32 // between isEmpty, isBetween, and isFull accordingly. Depending on the situation, the increment conditions
33 // for the read and write addresses will also change. The 1-bit outputs empty and full depend on the current state
34 // the FIFO is in
35 enum {isEmpty, isFull, isBetween} ps, ns;
36 always_comb begin
37     case(ps)
38     isEmpty : begin
39         writeIncr = 0;
40         readIncr = 0;
41         empty = 1;
42         full = 0;
43         if(!read & write) begin
44             writeIncr = 1;
45             ns = isBetween;
46         end else
47             ns = isEmpty;
48     end
49     isBetween : begin
50         writeIncr = 0;
51         readIncr = 0;
52         empty = 0;
53         full = 0;
54         if(read & !write & almost_empty) begin
55             readIncr = 1;
56             ns = isEmpty;
57         end else if(!read & write & almost_full) begin
58             writeIncr = 1;
59             ns = isFull;
60         end else if(read & !write) begin
61             readIncr = 1;
62             ns = isBetween;
63         end
64     end
65     isFull : begin
66         writeIncr = 0;
67         readIncr = 0;
68         empty = 0;
69         full = 1;
70         if(read & !write) begin
71             readIncr = 1;
72             ns = isBetween;
73         end else begin
74             ns = isFull;
75         end
76     end
77     endcase
78 end
79
80 // this always_ff block sets the present state to the "isEmpty" state when reset,
81 // and will progress the present state to the next state otherwise every positive
82 // edge of the clk.
83 always_ff @(posedge clk) begin
84     if(reset) begin
85         ps <= isEmpty;
86     end else begin
87         ps <= ns;
88     end
89 end
90
91 endmodule
```

### 3.F) FIFO\_Control.sv (testbench)

```
// FIFO_Control_testbench tests the expected and unexpected situations of this module. In this simulation,
// I tested to see what happens when the FIFO is completely full, and if we read everything will it be completely
// empty. Then, I also tested to see if there are any issues in the "isBetween" state by reading and writing in
// those states a few times.

module FIFO_Control_testbench();
    //parameter depth = 4;
    logic clk, reset;
    logic read, write;
    logic wr_en;
    logic empty, full;
    logic [3:0] readAddr, writeAddr;

    FIFO_Control #(depth(4)) dut (.clk, .reset, .read, .write, .wr_en, .empty, .full, .readAddr, .writeAddr);

    parameter CLK_Period = 100;

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

    initial begin
        reset <= 1;
        reset <= 0; read <= 0; write <= 0; @ (posedge clk);
        read <= 1; @ (posedge clk);
        read <= 0; write <= 1; repeat(20) @ (posedge clk);
        read <= 1; write <= 0; repeat(20) @ (posedge clk);
        read <= 0; write <= 1; repeat(8) @ (posedge clk);
        read <= 1; write <= 0; repeat(6) @ (posedge clk);

        reset <= 1; @ (posedge clk);
        reset <= 0; repeat(4) @ (posedge clk);
        $stop;
    end
endmodule
```

### 3.G) FIFO.sv (code)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 3
5
6 // FIFO takes the 1-bit inputs clk, reset, read, and write and the variable-bit inputBus input and outputs/
7 // the 1-bit outputs empty and full and the variable-bit output outputBus. inputBus and outputBus are variable
8 // length determined by the global parameter width. The purpose of this module is to connect the 16x8 dual-port
9 // RAM module and the FIFO control module to represent the overall FIFO.
10
11 module FIFO #(parameter depth = 4, parameter width = 8)
12     (clk, reset, read, write, inputBus, empty, full, outputBus);
13
14     input logic clk, reset;
15     input logic read, write;
16     input logic [width-1:0] inputBus;
17     output logic empty, full;
18     output logic [width-1:0] outputBus;
19
20     /* Define Variables Here */
21     logic wr_en;
22     logic [depth-1:0] writeAddr, readAddr;
23
24     // ram32x4 is a verilog file that uses 1-bit input clk, 8-bit input inputBus, 4-bit inputs rdaddress and wraddress,
25     // and 1-bit input wren to output the 8-bit output outputBus. The purpose of this module is that it is the 16x8 dual-port
26     // RAM that the IP catalog generated and is the core of the FIFO
27     ram16x8 ram (.clock(clk), .data(inputBus), .rdaddress(readAddr), .wraddress(writeAddr),
28         .wren(wr_en), .q(outputBus));
29
30     // FIFO_control uses the 1-bit inputs clk, reset, read and write and outputs the 1-bit outputs wr_en, empty, and full and
31     // outputs the variable-bit outputs readAddr and writeAddr. In this lab, these outputs are 4 bits since global parameter
32     // depth is set to 4. The purpose of this module is to control the overall behavior of the FIFO and keep things organized
33     // The FIFO_control makes sure the FIFO behaves a certain way when it is full or empty, and allows for data to be written
34     // into or read from the RAM.
35     FIFO_Control #(depth) FC (.clk, .reset, .read, .write, .wr_en, .empty, .full, .readAddr, .writeAddr);
36
37 endmodule
```

### 3.H) FIFO.sv (testbench)

```
// FIFO_testbench tests both expected and unexpected situations. In this simulation, I tested if the FIFO can reach full
// or empty properly, and if it outputs the proper 1-bit empty and full outputs or the 4-bit outputBus value.
`timescale 1 ps / 1 ps
module FIFO_testbench();

    parameter depth = 4, width = 8;
    logic clk, reset;
    logic read, write;
    logic [width-1:0] inputBus;
    logic resetState;
    logic empty, full;
    logic [width-1:0] outputBus;

    FIFO #(depth, width) dut (.*);

    parameter CLK_Period = 100;

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

    initial begin
        reset <= 1;                                     @(posedge clk);
        reset <= 0;                                     @(posedge clk);
        inputBus <= 8'b10001000; read <= 0; write <= 0; @(posedge clk);
        write <= 1;                                     @(posedge clk);
        write <= 0;                                     @(posedge clk);
        repeat (15) begin                               @(posedge clk);
            write <= 1;                                 @(posedge clk);
            write <= 0;                                 @(posedge clk);
        end
        read <= 1;                                     @(posedge clk);
        read <= 0;                                     @(posedge clk);
        repeat (15) begin                               @(posedge clk);
            read <= 1;                                 @(posedge clk);
            read <= 0;                                 @(posedge clk);
        end
        $stop;
    end
endmodule
```

### 3.I) DE1\_SoC.sv (code)

```
1 // Brian Dallaire
2 // 04/23/2021
3 // EE 371
4 // Lab #2, Task 3
5
6 // DE1_SoC takes a 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW and returns the six, 7-bit outputs
7 // HEX5-HEX0 and 10-bit output LEDR. DE1_SoC combines all of the modules in Task 3 and uses HEX5-HEX0 to display
8 // the values of the read and write data and read and write addresses in hexadecimal values. A counter is implemented
9 // to increment the read address. It also outputs the full and empty signals into LEDR[9] and LEDR[8], respectively.
10
11 module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR, CLOCK_50);
12
13     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
14     output logic [9:0] LEDR;
15     input logic [3:0] KEY;
16     input logic [9:0] SW;
17     input logic CLOCK_50;
18
19     assign HEX3 = 7'b1111111;
20     assign HEX2 = 7'b1111111;
21
22     logic read, write;
23     logic full, empty;
24     logic reset;
25     logic [7:0] inputBus, outputBus;
26
27     assign reset = SW[0];
28     assign LEDR[9] = full;
29     assign LEDR[8] = empty;
30
31     // series_dffs has a 1-bit input CLOCK_50, SW[9], and 8-bit input SW[7:0] and outputs the 8-bit output
32     // inputBus. This ensures SW[7:0] outputs a clean signal and prevents metastability using two flip-flops in series
33     series_dffs #(BITS(8)) cleanData (.clk(CLOCK_50), .reset(SW[9]), .raw(SW[7:0]), .clean(inputBus));
34
35     // input_buffer takes a 1-bit input CLOCK_50, SW[9], and KEY[3] and outputs the 1-bit output read.
36     // this ensures that the key press will only result in one clock cycle where read is true.
37     input_buffer readBuffer (.clk(CLOCK_50), .reset, .press(~KEY[3]), .out(read));
38
39     // input_buffer takes a 1-bit input CLOCK_50, SW[9], and KEY[2] and outputs the 1-bit output write.
40     // this ensures that the key press will only result in one clock cycle where write is true.
41     input_buffer writeBuffer (.clk(CLOCK_50), .reset, .press(~KEY[2]), .out(write));
42
43     // FIFO takes the 1-bit inputs clk, reset, read, and write and the variable-bit inputBus input and outputs/
44     // the 1-bit outputs empty and full and the variable-bit output outputBus. This module is the core of this
45     // task, and represents both the 16x8 RAM and FIFO_Control modules.
46     FIFO f (.clk(CLOCK_50), .reset, .read, .write, .inputBus, .empty, .full, .outputBus);
47
48     // seg7hex takes the 4-bit input which is the last 4 bits of inputBus, and outputs
49     // the corresponding hexadecimal value to the 7-segment display HEX5
50     seg7hex inputBus_L (.bcd(inputBus[7:4]), .leds(HEX5));
51
52     // seg7hex takes the 4-bit input which is the first 4 bits of inputBus, and outputs
53     // the corresponding hexadecimal value to the 7-segment display HEX4
54     seg7hex inputBus_R (.bcd(inputBus[3:0]), .leds(HEX4));
55
56     // seg7hex takes the 4-bit input which is the last 4 bits of outputBus, and outputs
57     // the corresponding hexadecimal value to the 7-segment display HEX1
58     seg7hex outputBus_L (.bcd(outputBus[7:4]), .leds(HEX1));
59
60     // seg7hex takes the 4-bit input which is the first 4 bits of outputBus, and outputs
61     // the corresponding hexadecimal value to the 7-segment display HEX0
62     seg7hex outputBus_R (.bcd(outputBus[3:0]), .leds(HEX0));
63
64 endmodule
65
```

### 3.J) DE1\_SoC.sv (testbench)

```

65
66 // DE1_SoC_testbench tests the expected and unexpected situations of this task. For this simulation,
67 // I tested what happens if I write into a lot of addresses, more than enough to make the FIFO full,
68 // and tested if when full the FIFO will behave normally. Then, I read all of these values more than
69 // enough times to see if the FIFO will behave normally when trying to read an empty FIFO.
70
71 `timescale 1 ps / 1 ps
72 module DE1_SoC_testbench();
73     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
74     logic [9:0] LEDR;
75     logic [3:0] KEY;
76     logic [9:0] SW;
77     logic CLOCK_50;
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     initial begin
87         SW[9] <= 1;
88         SW[9] <= 0; SW[7:0] <= 8'b00000000; KEY[3] <= 1; KEY[2] <= 1; @(posedge CLOCK_50);
89         SW[7:0] <= 8'b01101010; KEY[2] <= 0; @(posedge CLOCK_50);
90         KEY[2] <= 1; @(posedge CLOCK_50);
91         SW[7:0] <= 8'b00011001; KEY[2] <= 0; @(posedge CLOCK_50);
92         KEY[2] <= 1; @(posedge CLOCK_50);
93         SW[7:0] <= 8'b00011001; KEY[2] <= 0; @(posedge CLOCK_50);
94         KEY[2] <= 1; @(posedge CLOCK_50);
95         SW[7:0] <= 8'b11000001; KEY[2] <= 0; @(posedge CLOCK_50);
96         KEY[2] <= 1; @(posedge CLOCK_50);
97         SW[7:0] <= 8'b01111001; KEY[2] <= 0; @(posedge CLOCK_50);
98         KEY[2] <= 1; @(posedge CLOCK_50);
99         SW[7:0] <= 8'b01111000; KEY[2] <= 0; @(posedge CLOCK_50);
100        KEY[2] <= 1; @(posedge CLOCK_50);
101        SW[7:0] <= 8'b00010101; KEY[2] <= 0; @(posedge CLOCK_50);
102        KEY[2] <= 1; @(posedge CLOCK_50);
103        SW[7:0] <= 8'b11000001; KEY[2] <= 0; @(posedge CLOCK_50);
104        KEY[2] <= 1; @(posedge CLOCK_50);
105        SW[7:0] <= 8'b01001001; KEY[2] <= 0; @(posedge CLOCK_50);
106        KEY[2] <= 1; @(posedge CLOCK_50);
107        SW[7:0] <= 8'b01000010; KEY[2] <= 0; @(posedge CLOCK_50);
108        KEY[2] <= 1; @(posedge CLOCK_50);
109        SW[7:0] <= 8'b00010101; KEY[2] <= 0; @(posedge CLOCK_50);
110        KEY[2] <= 1; @(posedge CLOCK_50);
111        SW[7:0] <= 8'b10100010; KEY[2] <= 0; @(posedge CLOCK_50);
112        KEY[2] <= 1; @(posedge CLOCK_50);
113        SW[7:0] <= 8'b00010101; KEY[2] <= 0; @(posedge CLOCK_50);
114        KEY[2] <= 1; @(posedge CLOCK_50);
115        SW[7:0] <= 8'b11001001; KEY[2] <= 0; @(posedge CLOCK_50);
116        KEY[2] <= 1; @(posedge CLOCK_50);
117        SW[7:0] <= 8'b11010111; KEY[2] <= 0; @(posedge CLOCK_50);
118        KEY[2] <= 1; @(posedge CLOCK_50);
119        SW[7:0] <= 8'b11100111; KEY[2] <= 0; @(posedge CLOCK_50);
120        KEY[2] <= 1; @(posedge CLOCK_50);
121        SW[7:0] <= 8'b11111001; KEY[2] <= 0; @(posedge CLOCK_50);
122        KEY[2] <= 1; @(posedge CLOCK_50);
123        SW[7:0] <= 8'b11111111; KEY[2] <= 0; @(posedge CLOCK_50);
124        SW[7:0] <= 8'b00000000; KEY[2] <= 1; @(posedge CLOCK_50);
125        repeat(18) begin
126            KEY[3] <= 0; @(posedge CLOCK_50);
127            KEY[3] <= 1; @(posedge CLOCK_50);
128        end
129        SW[9] <= 1;
130        SW[9] <= 0;
131
132        $stop;
133    end
134 endmodule
135

```