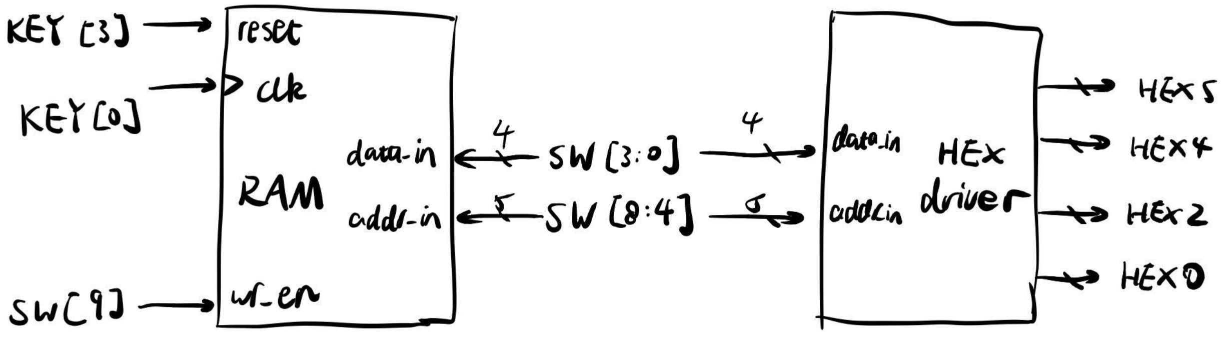
Peter Zhong
EE 371
Apr. 21st, 2021
Lab 2 Report

Procedure

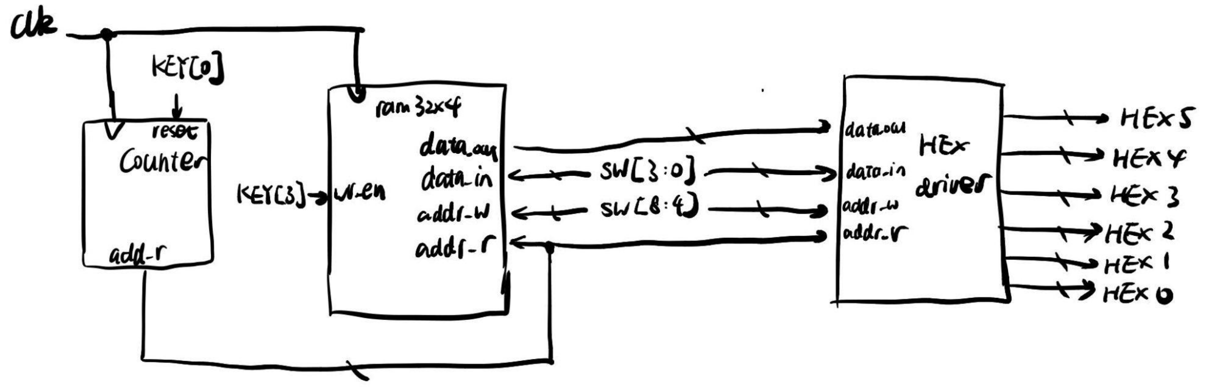
Task #1: RAM on FPGA



(Figure 1: Block Diagram of the RAM Module)

Task #1 asks me to implement 32x4 RAM in SystemVerilog. To implement this RAM, I first implemented a 32x4 RAM module that stores data at certain addresses when the write signal is enabled. I implemented these functionalities with a 32x4 array and an always_ff block for logics. I then implemented a hex display driver to properly display the address, input data, and output data information on the designated hex displays. Lastly, I combined the two modules in my top-level module and mapped the I/Os to the corresponding switches and keys on the DE1_SoC board. Please see the code under "Task #1" in the appendix section for the details regarding my implementation.

Task #2: Pre-Configured RAM with Counter

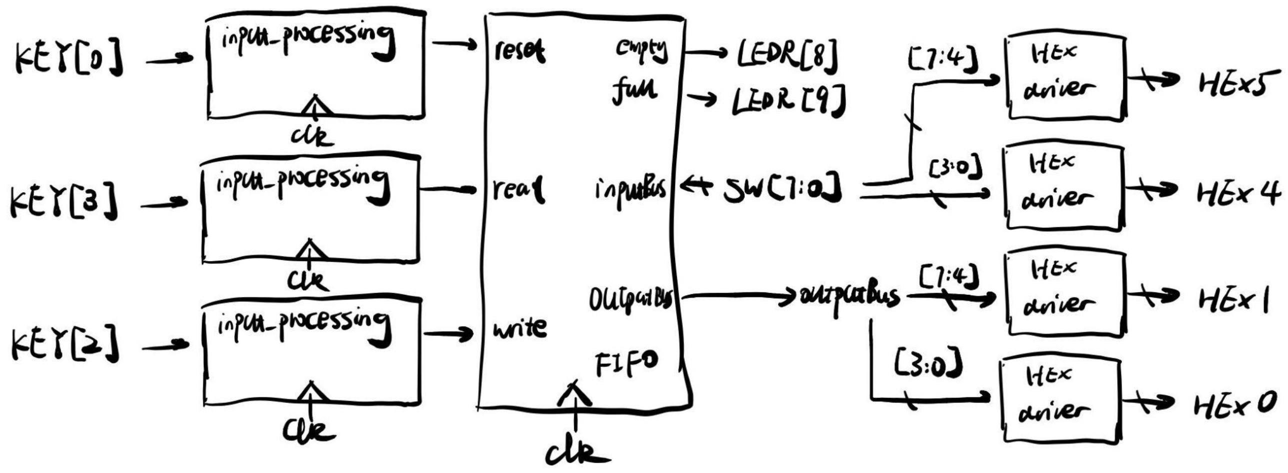


(Figure 2: Block Diagram of the RAM w/ Counter Module)

Task #2 asks me to implement a pre-configured 32x4 RAM, and attach two additional modules to it in SystemVerilog. I need to create a counter module that increments the read address once every second, and I need to create another hex display driver module to output appropriate information to the 6 hex displays on the DE1_SoC board.

To implement the RAM, I followed the instructions listed on the lab manual. To implement the counter, I used an integer to keep count and added conditional statements in an always_ff block to implement the logic for the counter to output true about once every second. To implement the hex_driver, I copied most of the code from the hex_driver module in Task #1, added two more outputs, and adjusted some connections within the module so that all six hex displays can be connected to this driver module. Please see the code under "Task #2" in the appendix section for the details regarding my implementation.

Task #3: FIFO Structure

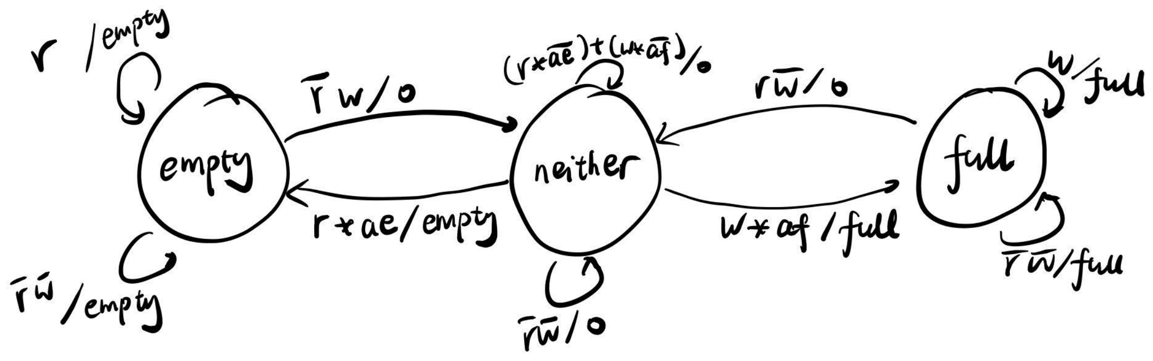


(Figure 3: Block Diagram of the FIFO Module)

Task #3 asks me to implement a FIFO queue-like structure on a 16x8 RAM with a FIFO_controller controlling the I/Os of the entire system. I first started by drawing out the state diagram of the FSM for the FIFO controller. The diagram is as shown below in Figure 1.

r = read , w = write, ae = almost_empty, af = almost_full

$$\begin{cases} \text{almost\_empty} = (\text{wr\_addr} == \text{rd\_addr}+1) \\ \text{almost\_full} = (\text{wr\_addr}+1 == \text{rd\_addr}) \end{cases}$$



(Figure 4: High-Level State Diagram for the FIFO FSM)

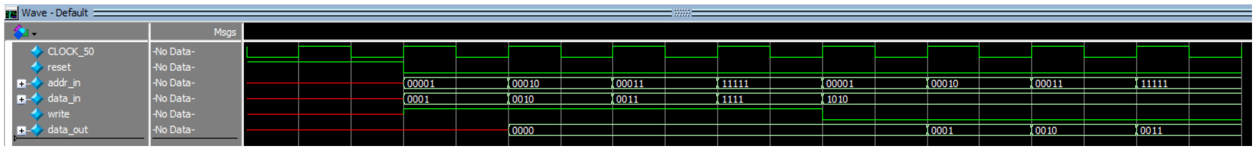
I then noticed that the functionality of this diagram can be more easily implemented with just logic statements inside an always_ff block. Therefore, I used 4 different conditional statements in my always_ff block to implement the functionality of the FIFO controller. I used an integer "n" to keep track of the number of elements in my FIFO structure, and I used another integer "f" to keep track of the address of the front of the FIFO. When a "read" operation goes through, n should be decremented and f should always be updated to (f+1)%(2**depth). When a "write" operation goes through, n should be incremented and the new data should always be stored at address (f+n)%(2**depth). Please see the code under "Task #3" in the appendix section for the details regarding my implementation.

The algorithm behind the indexing actions on read and write comes from what I have learned about ArrayQueue in Java. To arrive at the two algorithms presented in my always_ff block, I consulted my CSE notes and also read through the following webpage on the implementation of ArrayQueue in Java: https://opendatastructures.org/ods-java/2_3_ArrayQueue_Array_Based_.html.

Results

Task #1 - Demonstration Video: https://youtu.be/TNEfibEp9ug

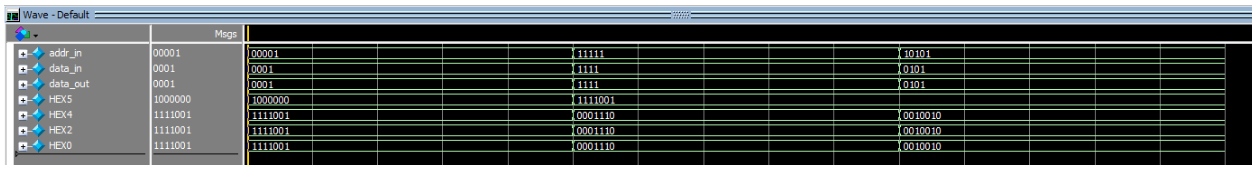
Screenshot for the RAM module simulation:



(Figure 5: Screenshot of RAM Module Simulation)

- For the RAM module, I first wrote 4 random data onto 4 random addresses on the RAM, and then attempted to read them. As can be seen from the simulation above, the RAM module correctly outputs the previously stored data when it is being written to , and the RAM module correctly outputs the currently stored data when it is being read from.

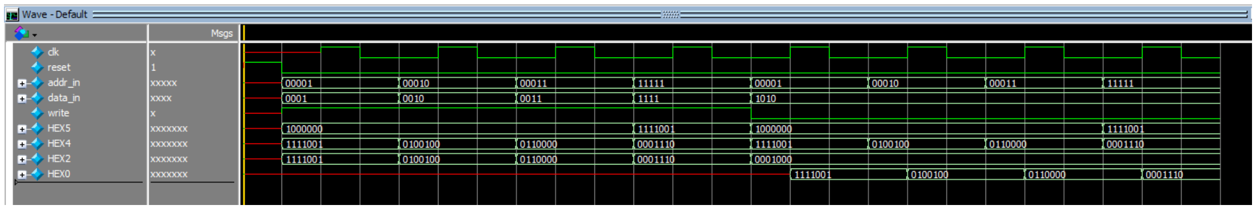
Screenshot for the hex_driver module simulation:



(Figure 6: Screenshot of hex_driver Module Simulation)

- For the hex_driver module, I tested three random combinations of addr_in, data_in, and data_out. As can be seen from the simulation above, the hex_driver module correctly outputs corresponding addresses in hexadecimal value to the corresponding hex display.

Screenshot for the top level simulation:

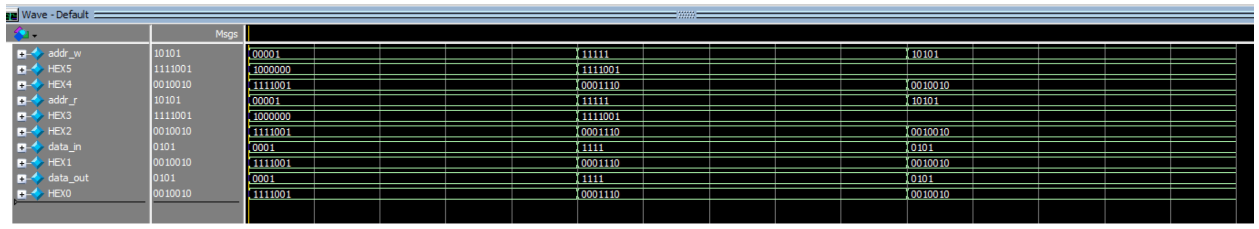


(Figure 7: Screenshot of sTop Level Module Simulation)

- For the top level module, I simulated it the same way as the RAM module: I wrote 4 random data onto 4 random addresses, and then attempted to read them. As can be seen from the simulation above, the hex displays correctly outputs the previously stored data when it is being written to, and they also correctly outputs the currently stored data when it is being read from.

Task #2 - Demonstration Video: https://youtu.be/OkuSoVpD4qw

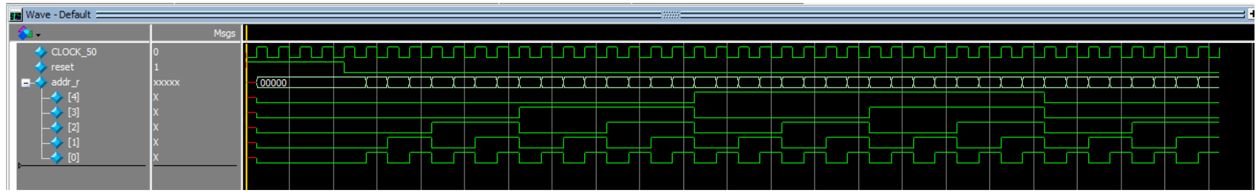
Screenshot for the hex_driver simulation:



(Figure 8: Screenshot of hex_driver Module Simulation)

- For the hex_driver module, I tested three random combinations of addr_w, addr_r, data_in, and data_out. As can be seen from the simulation above, the hex_driver module correctly outputs corresponding addresses in hexadecimal value to the corresponding hex display.

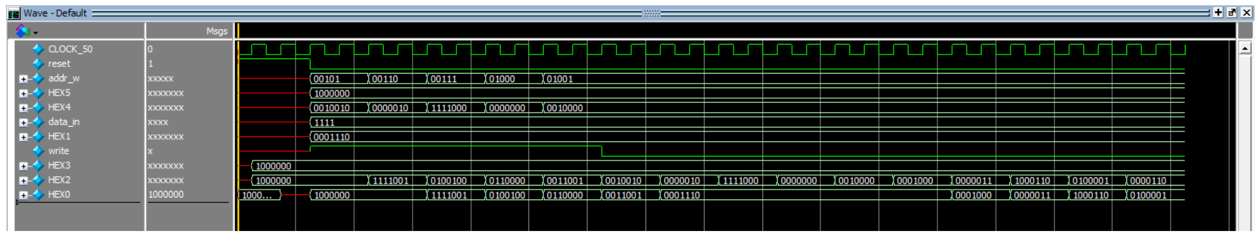
Screenshot for the hex_driver simulation:



(Figure 9: Screenshot of counter Module Simulation)

- For the counter module, I set the parameter of the counter to 0 so that it increments on every clock cycle. I then resetted the counter and let it run for 40 clock cycles. As can be seen from the simulation above, the counter goes through every number from 0 - 5'b11111, and cycles back to 0 after reaching the maximum value.

Screenshot for the top level simulation:



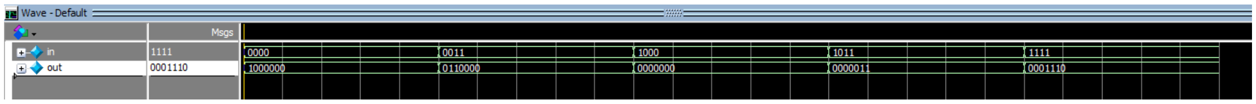
(Figure 10: Screenshot of counter Module Simulation)

- For the top level module, I performed the following steps in order:
    - 1. Reset the entire module.
    - 2. Read address 0-4, while also writing hexadecimal f to address 5-9

    - 3. Read address 5-9.
    - 4. Read address 10-14, which are unmodified.
- As can be seen from the simulation above, the RAM correctly outputs the addr_w and write value to HEX 5-4 and HEX0. Additionally, it also correctly performs the write action, since the stored values at address 5-9 are all updated to hexadecimal f. All of these values are also properly displayed on the hex displays.

Task #3 - Demonstration Video: https://youtu.be/9uKttPRFil4

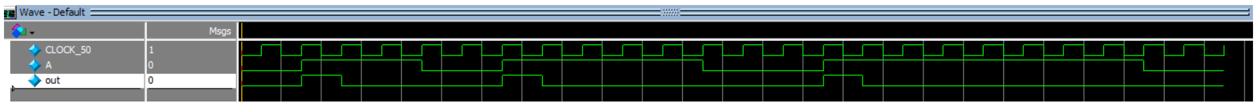
Screenshot for hex_driver simulation:



(Figure 11: Screenshot of hex_driver Module Simulation)

- To test the functionality of the hex_driver, I picked 4 random hexadecimal values (0, 3, 8, b, f) and tested their output. As can be seen from the simulation above, all 5 instances gave the correct hex display output.

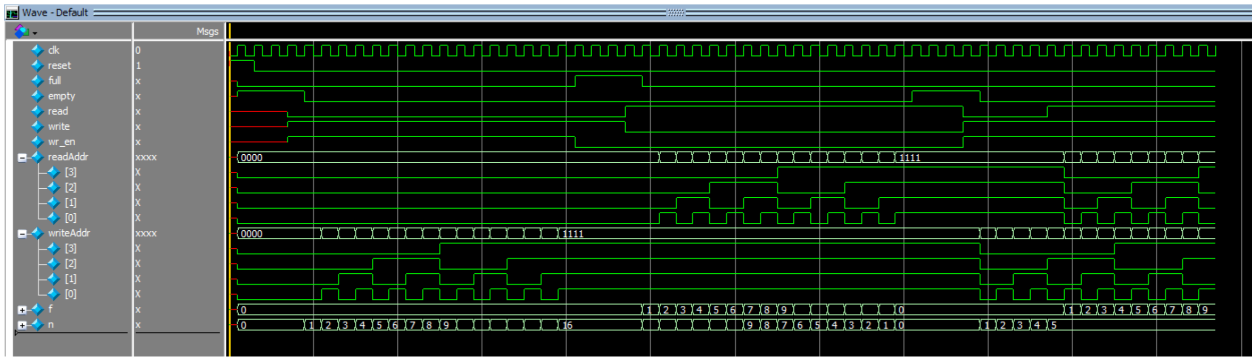
Screenshot for input_processing simulation:



(Figure 12: Screenshot of input_processing Module Simulation)

- To test the functionality of input_processing, I pressed and released the virtual "input" button for random amounts of durations. As can be seen from the simulation above, the input processing module correctly reduced the user input to 1 clock cycle long for all instances of user input.

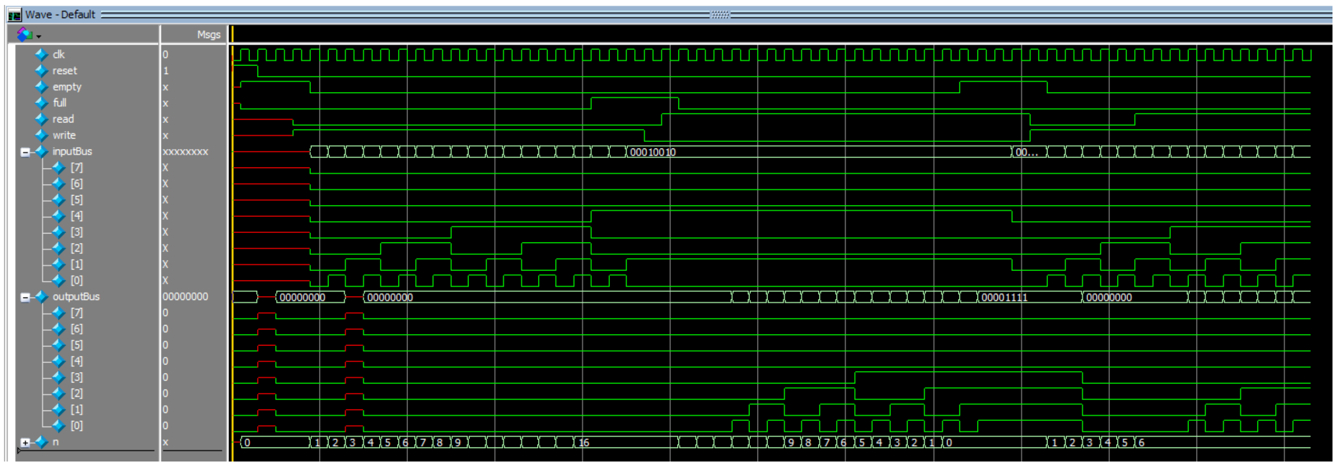
Screenshot for the FIFO_Control simulation:



(Figure 13: Screenshot of FIFO_control Module Simulation)

- To test the functionality of my FIFO control module, I tested three scenarios: writing over the size of the RAM, reading over the size of the RAM, and reading and writing simultaneously. As can be seen from the simulation above, the FIFO controller gives the correct output for all three scenarios. For the first scenario, it outputs “full”, disables wr_en, and holds the write address to 1111. For the second scenario, it outputs “empty” and holds the read address to 0000. For the third scenario, it performs both read and write operations correctly, and it does not update the number of elements in the RAM (since one element is added and another one is discarded).

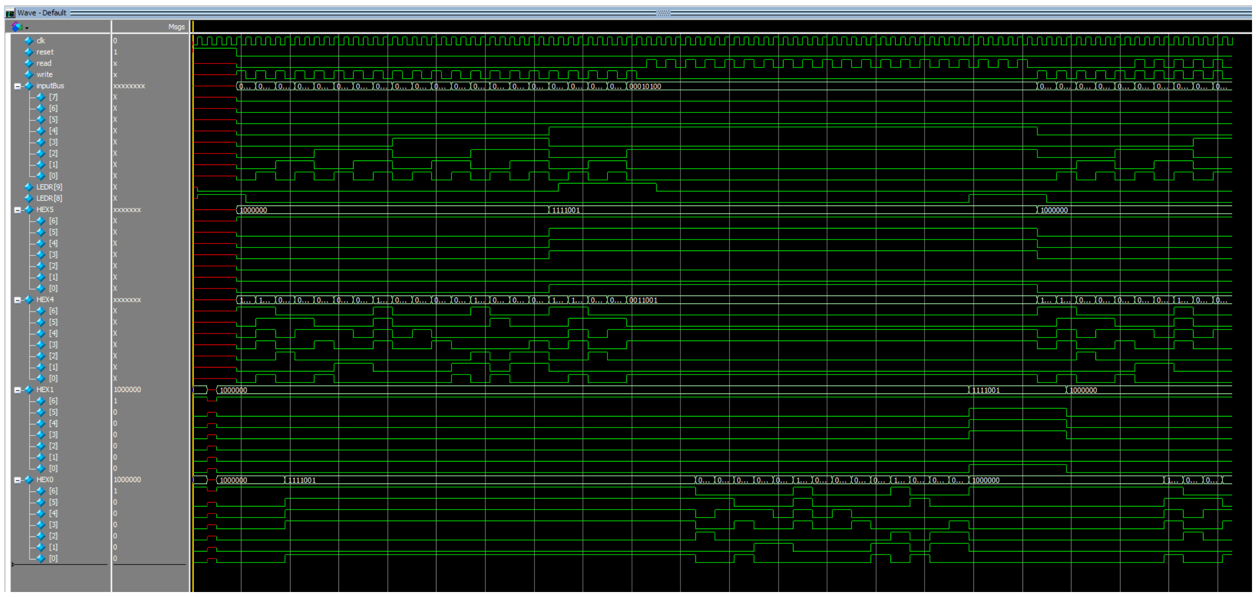
Screenshot for the FIFO simulation:



(Figure 14: Screenshot of FIFO Module Simulation)

- To test the functionality of my FIFO module, I first wrote data from 8’h00 through 8’h12 to the FIFO over 19 clock cycles. As can be seen from the simulation above, the FIFO correctly outputs “full” after 8’h0f has been stored.
- Then, I disabled the write signal, enabled the read signal, and allowed the FIFO to run for 20 clock cycles. As can be seen from the simulation above, the FIFO correctly outputs data from 8’h00 through 8’h0f. After outputting 8’h0f, it also correctly outputs the empty signal and stops outputting further values (since 8’h0f is the last value stored in the FIFO).
- After these actions have been completed, I tested the simultaneous read/write functionality of the FIFO structure. I first wrote 8’h00 - 8’h06 to the RAM, and then enabled both read and write signals. As can be seen from the simulation above, the FIFO correctly stores the data and also outputs the previously stored data in order. The parameter named “n” on the bottom, which is the parameter used to keep track of the number of elements in the FIFO, also stays constant after read/write operations have been enabled at the same time.

Screenshot for the top level simulation:



(Figure 15: Screenshot of Top Level Simulation)

- To test the functionality of the top level module, I used a procedure that is very similar to the testing procedure of the FIFO module.
- It can be seen from the simulation above that HEX5 and HEX4 correctly displays in real time the information on the input data bus.
- I first wrote data from 8’h00 through 8’h14 to the RAM over 42 clock cycles. As can be seen from the simulation above, the LEDR[9] correctly outputs “full” after 8’h0f has been stored.
- Then, I disabled the write signal, enabled the read signal, and allowed the FIFO to run for 40 clock cycles. As can be seen from the simulation above, the RAM correctly outputs data from 8’h00 through 8’h0f on HEX0 and HEX1. After outputting 8’h0f, it also correctly outputs the empty signal on LEDR[8] and stops outputting further values (since 8’h0f is the last value stored in the FIFO).
- After these actions have been completed, I tested the simultaneous read/write functionality of the FIFO structure. I first wrote 8’h00 - 8’h04 to the RAM, and then enabled both read and write signals. As can be seen from the simulation above, the FIFO correctly stores the data and also outputs the previously stored data in order.

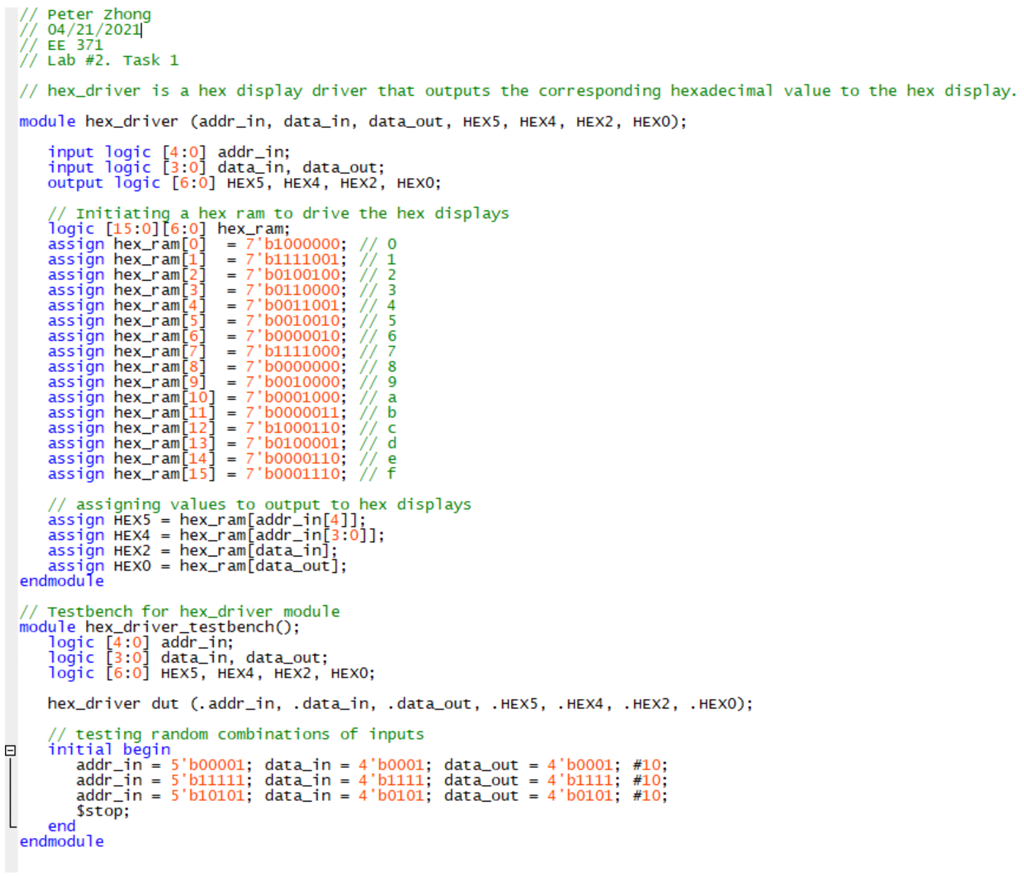
Appendix

Task #1

1) RAM.sv

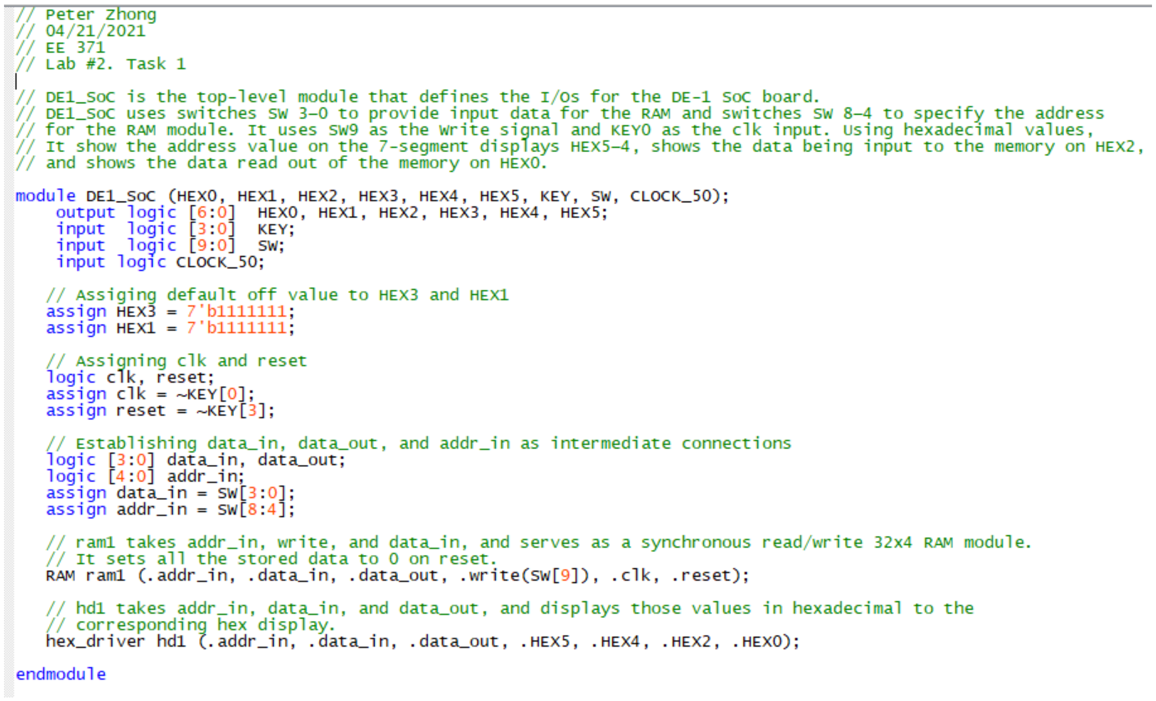
```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 1

// RAM is by default a 32x4 RAM that implements synchronous read/write functionalities as specified in the
// lab specifications.

module RAM #(parameter ADDR_WIDTH=5, parameter DATA_WIDTH=4) (addr_in, data_in, data_out, write, clk, reset);

   input logic [ADDR_WIDTH-1:0] addr_in;
   input logic [DATA_WIDTH-1:0] data_in;
   input logic write, clk, reset;
   output logic [DATA_WIDTH-1:0] data_out;

   // Initiating the 2D array for the memory module
   logic [2**ADDR_WIDTH-1:0][DATA_WIDTH-1:0] memory_array;

   always_ff @(posedge clk) begin
      // Implementing reset logic
      if (reset)
         memory_array <= '0;
      // Implementing write logic
      else if (write) begin
         memory_array[addr_in] <= data_in;
         data_out <= memory_array[addr_in];
      end
      // Implementing read logic
      else
         data_out <= memory_array[addr_in];
   end

endmodule

// Testbench for the RAM module
module RAM_testbench();
   logic [4:0] addr_in;
   logic [3:0] data_in;
   logic write, reset;
   logic [3:0] data_out;
   logic CLOCK_50;

   RAM dut (.addr_in, .data_in, .data_out, .write, .clk(CLOCK_50), .reset);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   initial begin
      // resetting the module
      reset <= 1; repeat(5) @(posedge CLOCK_50);

      // writing some random data into the RAM
      reset <= 0; addr_in <= 5'b00001; data_in <= 4'b0001; write <= 1'b1; @(posedge CLOCK_50);
                  addr_in <= 5'b00010; data_in <= 4'b0010; write <= 1'b1; @(posedge CLOCK_50);
                  addr_in <= 5'b00011; data_in <= 4'b0011; write <= 1'b1; @(posedge CLOCK_50);
                  addr_in <= 5'b11111; data_in <= 4'b1111; write <= 1'b1; @(posedge CLOCK_50);

      // reading the previously written data from the RAM
                  addr_in <= 5'b00001; data_in <= 4'b1010; write <= 1'b0; @(posedge CLOCK_50);
                  addr_in <= 5'b00010; data_in <= 4'b1010; write <= 1'b0; @(posedge CLOCK_50);
                  addr_in <= 5'b00011; data_in <= 4'b1010; write <= 1'b0; @(posedge CLOCK_50);
                  addr_in <= 5'b11111; data_in <= 4'b1010; write <= 1'b0; @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

2) hex_driver.sv



```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 1

// hex_driver is a hex display driver that outputs the corresponding hexadecimal value to the hex display.

module hex_driver (addr_in, data_in, data_out, HEX5, HEX4, HEX2, HEX0);

   input logic [4:0] addr_in;
   input logic [3:0] data_in, data_out;
   output logic [6:0] HEX5, HEX4, HEX2, HEX0;

   // Initiating a hex ram to drive the hex displays
   logic [15:0][6:0] hex_ram;
   assign hex_ram[0]  = 7'b1000000; // 0
   assign hex_ram[1]  = 7'b1111001; // 1
   assign hex_ram[2]  = 7'b0100100; // 2
   assign hex_ram[3]  = 7'b0110000; // 3
   assign hex_ram[4]  = 7'b0011001; // 4
   assign hex_ram[5]  = 7'b0010010; // 5
   assign hex_ram[6]  = 7'b0000010; // 6
   assign hex_ram[7]  = 7'b1111000; // 7
   assign hex_ram[8]  = 7'b0000000; // 8
   assign hex_ram[9]  = 7'b0010000; // 9
   assign hex_ram[10] = 7'b0001000; // a
   assign hex_ram[11] = 7'b0000011; // b
   assign hex_ram[12] = 7'b1000110; // c
   assign hex_ram[13] = 7'b0100001; // d
   assign hex_ram[14] = 7'b0000110; // e
   assign hex_ram[15] = 7'b0001110; // f

   // assigning values to output to hex displays
   assign HEX5 = hex_ram[addr_in[4]];
   assign HEX4 = hex_ram[addr_in[3:0]];
   assign HEX2 = hex_ram[data_in];
   assign HEX0 = hex_ram[data_out];
endmodule

// Testbench for hex_driver module
module hex_driver_testbench();
   logic [4:0] addr_in;
   logic [3:0] data_in, data_out;
   logic [6:0] HEX5, HEX4, HEX2, HEX0;

   hex_driver dut (.addr_in, .data_in, .data_out, .HEX5, .HEX4, .HEX2, .HEX0);

   // testing random combinations of inputs
   initial begin
      addr_in = 5'b00001; data_in = 4'b0001; data_out = 4'b0001; #10;
      addr_in = 5'b11111; data_in = 4'b1111; data_out = 4'b1111; #10;
      addr_in = 5'b10101; data_in = 4'b0101; data_out = 4'b0101; #10;
      $stop;
   end
endmodule
```

3) DE1_SoC.sv

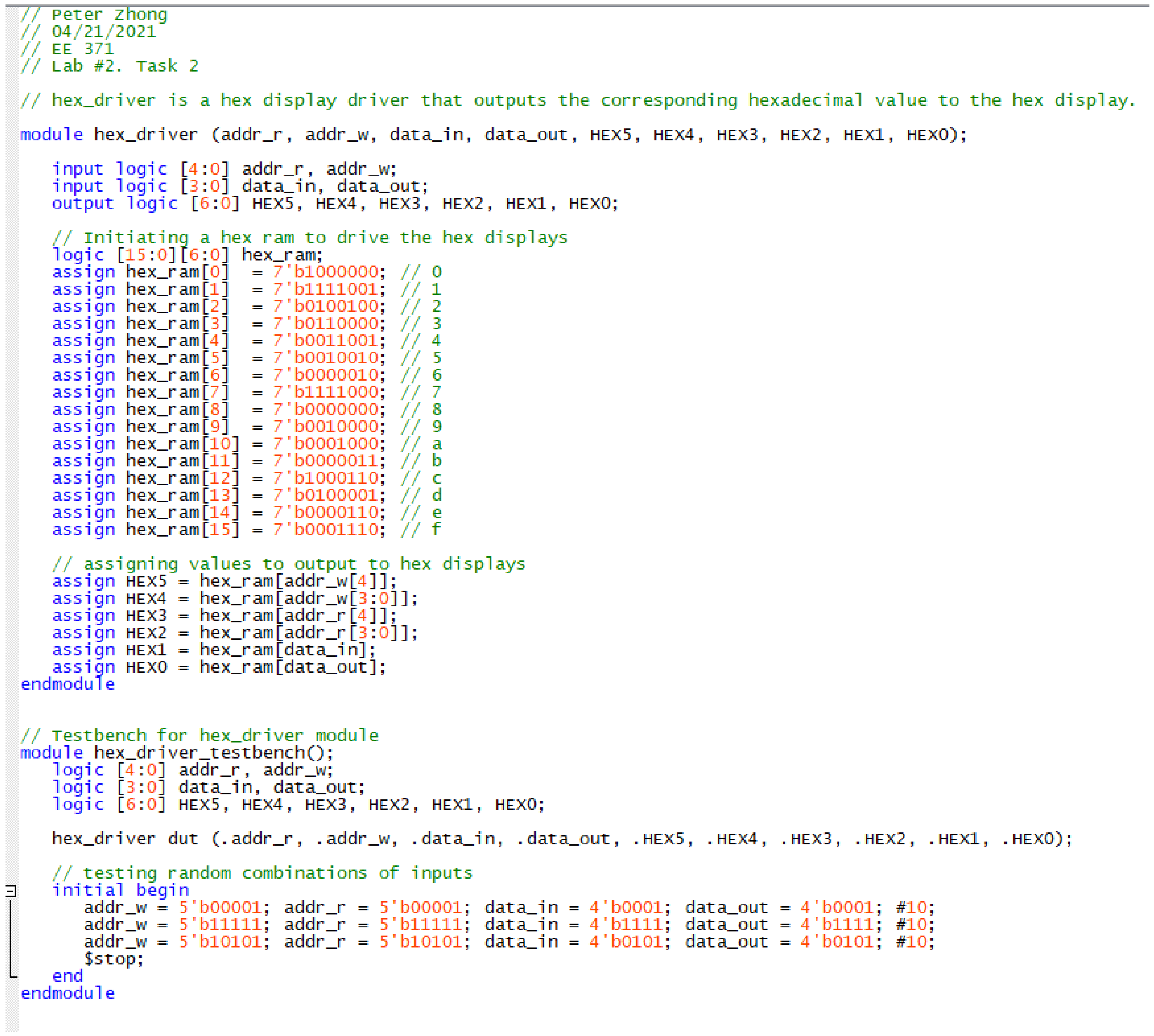


```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 1

// DE1_SoC is the top-level module that defines the I/Os for the DE-1 SoC board.
// DE1_SoC uses switches SW 3-0 to provide input data for the RAM and switches SW 8-4 to specify the address
// for the RAM module. It uses SW9 as the write signal and KEY0 as the clk input. Using hexadecimal values,
// It show the address value on the 7-segment displays HEX5-4, shows the data being input to the memory on HEX2,
// and shows the data read out of the memory on HEX0.

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;

   // Assiging default off value to HEX3 and HEX1
   assign HEX3 = 7'b1111111;
   assign HEX1 = 7'b1111111;

   // Assigning clk and reset
   logic clk, reset;
   assign clk = ~KEY[0];
   assign reset = ~KEY[3];

   // Establishing data_in, data_out, and addr_in as intermediate connections
   logic [3:0] data_in, data_out;
   logic [4:0] addr_in;
   assign data_in = SW[3:0];
   assign addr_in = SW[8:4];

   // ram1 takes addr_in, write, and data_in, and serves as a synchronous read/write 32x4 RAM module.
   // It sets all the stored data to 0 on reset.
   RAM ram1 (.addr_in, .data_in, .data_out, .write(SW[9]), .clk, .reset);

   // hd1 takes addr_in, data_in, and data_out, and displays those values in hexadecimal to the
   // corresponding hex display.
   hex_driver hd1 (.addr_in, .data_in, .data_out, .HEX5, .HEX4, .HEX2, .HEX0);

endmodule
```

```
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);

   // Setting up logic to make testbench cleaner
   logic [4:0] addr_in;
   logic [3:0] data_in;
   logic write, clk, reset;
   assign SW[3:0] = data_in;
   assign SW[8:4] = addr_in;
   assign SW[9] = write;
   assign KEY[0] = ~clk;
   assign KEY[3] = ~reset;

   // Setting up a simulated clk.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 = 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 = ~CLOCK_50; // Forever toggle the clk
   end

   // Trying all combinations of inputs (x and y).
   initial begin
      // resetting the module
      reset = 1; #10;

      // writing some random data into the RAM
      reset = 0; addr_in = 5'b00001; data_in = 4'b0001; write = 1'b1; #10;
                 clk = 1; #10; clk = 0; #10; // simulated clk pulse
                 addr_in = 5'b00010; data_in = 4'b0010; write = 1'b1; #10;
                 clk = 1; #10; clk = 0; #10;
                 addr_in = 5'b00011; data_in = 4'b0011; write = 1'b1; #10;
                 clk = 1; #10; clk = 0; #10;
                 addr_in = 5'b11111; data_in = 4'b1111; write = 1'b1; #10;
                 clk = 1; #10; clk = 0; #10;

      // reading the previously written data from the RAM
                 addr_in = 5'b00001; data_in = 4'b1010; write = 1'b0; #10;
                 clk = 1; #10; clk = 0; #10;
                 addr_in = 5'b00010; data_in = 4'b1010; write = 1'b0; #10;
                 clk = 1; #10; clk = 0; #10;
                 addr_in = 5'b00011; data_in = 4'b1010; write = 1'b0; #10;
                 clk = 1; #10; clk = 0; #10;
                 addr_in = 5'b11111; data_in = 4'b1010; write = 1'b0; #10;
                 clk = 1; #10; clk = 0; #10;
      $stop;
   end
endmodule
```

Task #2

1) hex_driver.sv

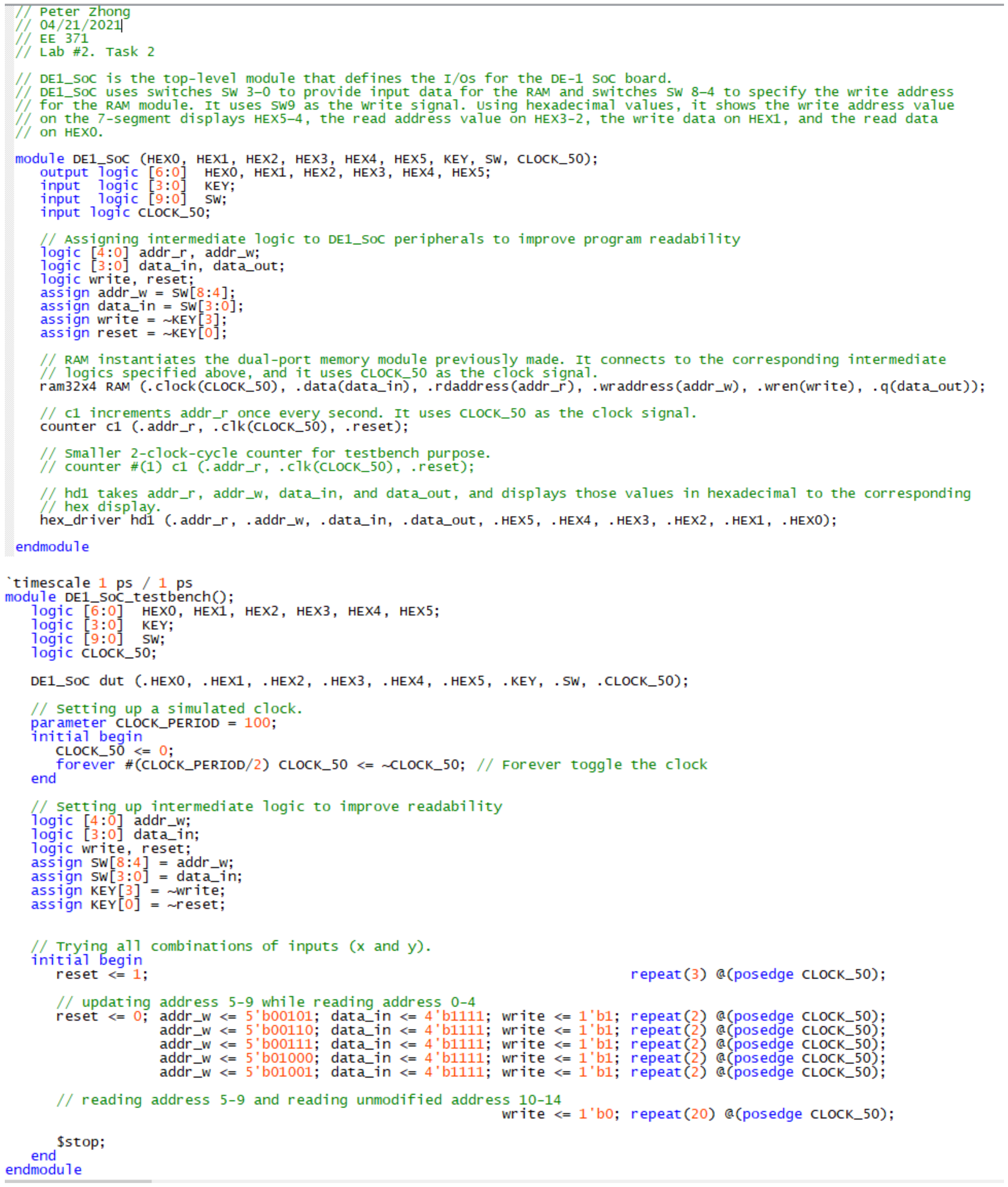


```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 2

// hex_driver is a hex display driver that outputs the corresponding hexadecimal value to the hex display.

module hex_driver (addr_r, addr_w, data_in, data_out, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

   input logic [4:0] addr_r, addr_w;
   input logic [3:0] data_in, data_out;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   // Initiating a hex ram to drive the hex displays
   logic [15:0][6:0] hex_ram;
   assign hex_ram[0]  = 7'b1000000; // 0
   assign hex_ram[1]  = 7'b1111001; // 1
   assign hex_ram[2]  = 7'b0100100; // 2
   assign hex_ram[3]  = 7'b0110000; // 3
   assign hex_ram[4]  = 7'b0011001; // 4
   assign hex_ram[5]  = 7'b0010010; // 5
   assign hex_ram[6]  = 7'b0000010; // 6
   assign hex_ram[7]  = 7'b1111000; // 7
   assign hex_ram[8]  = 7'b0000000; // 8
   assign hex_ram[9]  = 7'b0010000; // 9
   assign hex_ram[10] = 7'b0001000; // a
   assign hex_ram[11] = 7'b0000011; // b
   assign hex_ram[12] = 7'b1000110; // c
   assign hex_ram[13] = 7'b0100001; // d
   assign hex_ram[14] = 7'b0000110; // e
   assign hex_ram[15] = 7'b0001110; // f

   // assigning values to output to hex displays
   assign HEX5 = hex_ram[addr_w[4]];
   assign HEX4 = hex_ram[addr_w[3:0]];
   assign HEX3 = hex_ram[addr_r[4]];
   assign HEX2 = hex_ram[addr_r[3:0]];
   assign HEX1 = hex_ram[data_in];
   assign HEX0 = hex_ram[data_out];
endmodule


// Testbench for hex_driver module
module hex_driver_testbench();
   logic [4:0] addr_r, addr_w;
   logic [3:0] data_in, data_out;
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   hex_driver dut (.addr_r, .addr_w, .data_in, .data_out, .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);

   // testing random combinations of inputs
   initial begin
      addr_w = 5'b00001; addr_r = 5'b00001; data_in = 4'b0001; data_out = 4'b0001; #10;
      addr_w = 5'b11111; addr_r = 5'b11111; data_in = 4'b1111; data_out = 4'b1111; #10;
      addr_w = 5'b10101; addr_r = 5'b10101; data_in = 4'b0101; data_out = 4'b0101; #10;
      $stop;
   end
endmodule
```

2) counter.sv

```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 2

// counter scrolls through the 5-bit address one by one, stopping for approximately one second at each address.
// The parameter FREQ specifies the number of clock cycles for the counter to increment by 1. It is set to
// 50M by default for the 50MHz clock on the DE1_SoC board.

module counter #(parameter FREQ=50000000)(addr_r, clk, reset);

   input logic clk, reset;
   output logic [4:0] addr_r;

   integer count;

   always_ff @(posedge clk) begin
      // Implementing reset
      if (reset) begin
         addr_r <= '0;
         count <= 0;
      end
      // Update addr_r when 1 second has passed
      else if (count == FREQ) begin
         if (addr_r == 5'b11111)
            addr_r <= '0;
         else
            addr_r <= addr_r + 5'b00001;
         count <= 0;
      end
      // Increment count when less than 1 second has passed
      else
         count <= count + 1;
   end

endmodule

// Testbench for counter module
module counter_testbench();
   logic clk, reset;
   logic [4:0] addr_r;
   logic CLOCK_50;

   // Instantiating a 1-clock-cycle counter for testbench purpose
   counter #(0) dut (.addr_r, .clk(CLOCK_50), .reset);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Reset the module, then let the counter run
   initial begin
      reset <= 1; repeat(5) @(posedge CLOCK_50);
      reset <= 0; repeat(40) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

3) DE1_SoC.sv



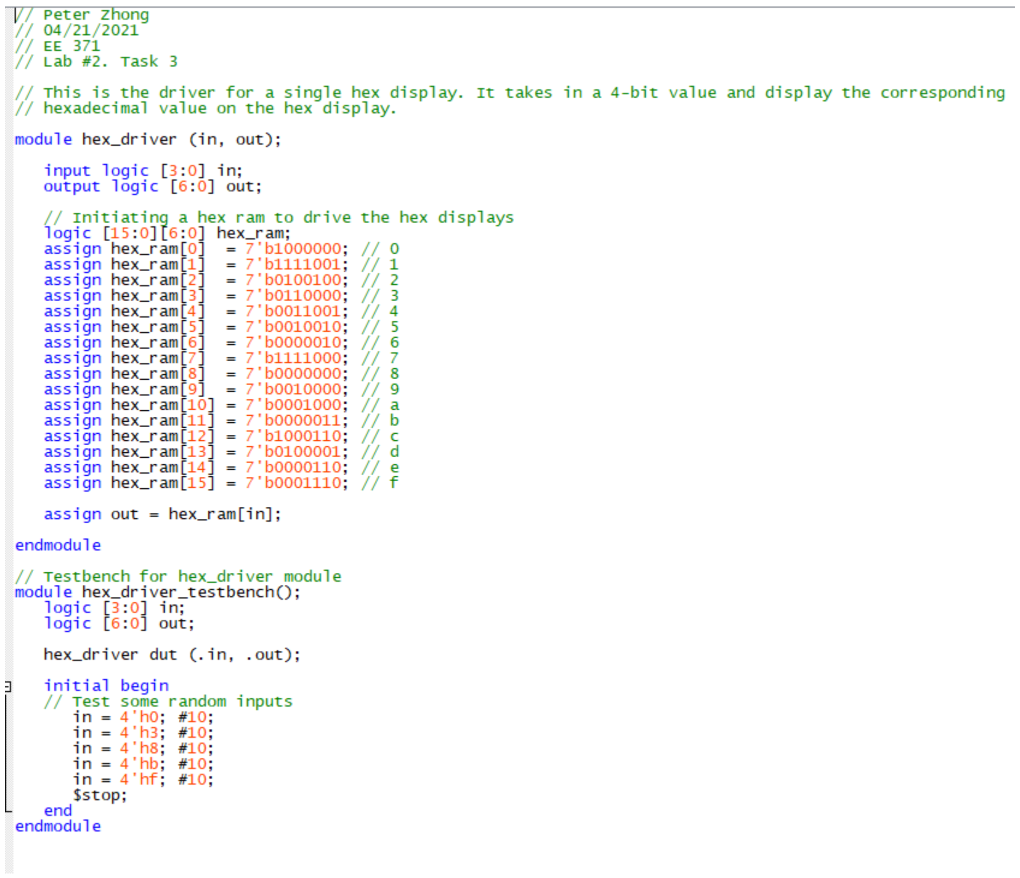
```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 2

// DE1_SoC is the top-level module that defines the I/Os for the DE-1 SoC board.
// DE1_SoC uses switches SW 3-0 to provide input data for the RAM and switches SW 8-4 to specify the write address
// for the RAM module. It uses SW9 as the write signal. Using hexadecimal values, it shows the write address value
// on the 7-segment displays HEX5-4, the read address value on HEX3-2, the write data on HEX1, and the read data
// on HEX0.

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;

   // Assigning intermediate logic to DE1_SoC peripherals to improve program readability
   logic [4:0] addr_r, addr_w;
   logic [3:0] data_in, data_out;
   logic write, reset;
   assign addr_w = SW[8:4];
   assign data_in = SW[3:0];
   assign write = ~KEY[3];
   assign reset = ~KEY[0];

   // RAM instantiates the dual-port memory module previously made. It connects to the corresponding intermediate
   // logics specified above, and it uses CLOCK_50 as the clock signal.
   ram32x4 RAM (.clock(CLOCK_50), .data(data_in), .rdaddress(addr_r), .wraddress(addr_w), .wren(write), .q(data_out));

   // c1 increments addr_r once every second. It uses CLOCK_50 as the clock signal.
   counter c1 (.addr_r, .clk(CLOCK_50), .reset);

   // Smaller 2-clock-cycle counter for testbench purpose.
   // counter #(1) c1 (.addr_r, .clk(CLOCK_50), .reset);

   // hd1 takes addr_r, addr_w, data_in, and data_out, and displays those values in hexadecimal to the corresponding
   // hex display.
   hex_driver hd1 (.addr_r, .addr_w, .data_in, .data_out, .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);

endmodule

`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);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Setting up intermediate logic to improve readability
   logic [4:0] addr_w;
   logic [3:0] data_in;
   logic write, reset;
   assign SW[8:4] = addr_w;
   assign SW[3:0] = data_in;
   assign KEY[3] = ~write;
   assign KEY[0] = ~reset;


   // Trying all combinations of inputs (x and y).
   initial begin
      reset <= 1;                                                        repeat(3) @(posedge CLOCK_50);

      // updating address 5-9 while reading address 0-4
      reset <= 0; addr_w <= 5'b00101; data_in <= 4'b1111; write <= 1'b1; repeat(2) @(posedge CLOCK_50);
                  addr_w <= 5'b00110; data_in <= 4'b1111; write <= 1'b1; repeat(2) @(posedge CLOCK_50);
                  addr_w <= 5'b00111; data_in <= 4'b1111; write <= 1'b1; repeat(2) @(posedge CLOCK_50);
                  addr_w <= 5'b01000; data_in <= 4'b1111; write <= 1'b1; repeat(2) @(posedge CLOCK_50);
                  addr_w <= 5'b01001; data_in <= 4'b1111; write <= 1'b1; repeat(2) @(posedge CLOCK_50);

      // reading address 5-9 and reading unmodified address 10-14
                                                         write <= 1'b0; repeat(20) @(posedge CLOCK_50);

      $stop;
   end
endmodule
```

Task #3

1) input_processing.sv

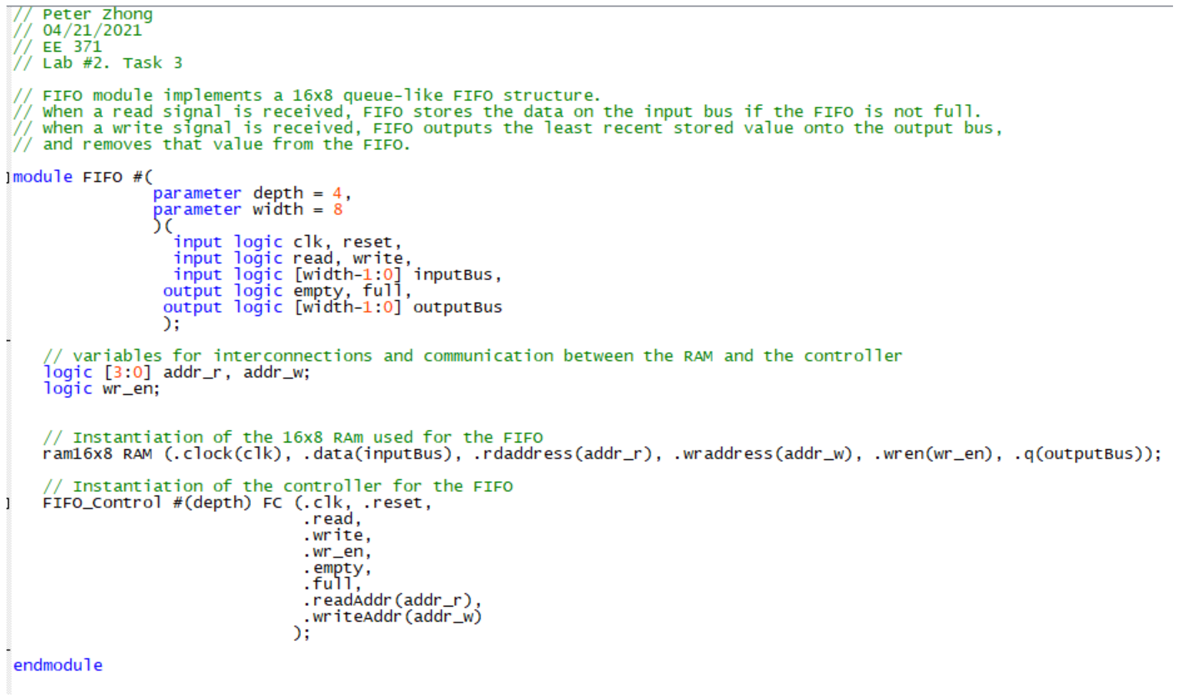
```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 3

// This input processor make user's button push 1 clock cycle long

// clean up button press and make user's button push 1 clock cycle long

module input_processing (A, out, clk);

   input logic A, clk;
   output logic out;

   logic buffer, in;

   // put the input logic through 2 D_FFs to clean up
   always_ff @(posedge clk) begin
      buffer <= A;
      in <= buffer;
   end

   enum {none, hold} ps, ns;

   // Next state logic
   always_comb begin
      case(ps)
         none: if (in) ns = hold;
               else ns = none;
         hold: if (in) ns = hold;
               else ns = none;
      endcase
   end

   // Output logic
   always_comb begin
      case (ps)
         none: if (in) out = 1'b1;
               else out = 1'b0;
         hold: out = 1'b0;
      endcase
   end

   // DFFs
   always_ff @(posedge clk) begin
         ps <= ns;
   end
endmodule

// Testbench for input_processing module
module input_processing_testbench();
   logic A, clk, out;
   logic CLOCK_50;

   input_processing dut (.A, .clk(CLOCK_50), .out);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   initial begin
      // press and release for random duration of time
      A <= 0; repeat(2) @(posedge CLOCK_50);
      A <= 1; repeat(3) @(posedge CLOCK_50);
      A <= 0; repeat(2) @(posedge CLOCK_50);
      A <= 1; repeat(5) @(posedge CLOCK_50);
      A <= 0; repeat(3) @(posedge CLOCK_50);
      A <= 1; repeat(8) @(posedge CLOCK_50);
      A <= 0; repeat(2) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

2) hex_driver.sv



```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 3

// This is the driver for a single hex display. It takes in a 4-bit value and display the corresponding
// hexadecimal value on the hex display.

module hex_driver (in, out);

   input logic [3:0] in;
   output logic [6:0] out;

   // Initiating a hex ram to drive the hex displays
   logic [15:0][6:0] hex_ram;
   assign hex_ram[0]  = 7'b1000000; // 0
   assign hex_ram[1]  = 7'b1111001; // 1
   assign hex_ram[2]  = 7'b0100100; // 2
   assign hex_ram[3]  = 7'b0110000; // 3
   assign hex_ram[4]  = 7'b0011001; // 4
   assign hex_ram[5]  = 7'b0010010; // 5
   assign hex_ram[6]  = 7'b0000010; // 6
   assign hex_ram[7]  = 7'b1111000; // 7
   assign hex_ram[8]  = 7'b0000000; // 8
   assign hex_ram[9]  = 7'b0010000; // 9
   assign hex_ram[10] = 7'b0001000; // a
   assign hex_ram[11] = 7'b0000011; // b
   assign hex_ram[12] = 7'b1000110; // c
   assign hex_ram[13] = 7'b0100001; // d
   assign hex_ram[14] = 7'b0000110; // e
   assign hex_ram[15] = 7'b0001110; // f

   assign out = hex_ram[in];

endmodule

// Testbench for hex_driver module
module hex_driver_testbench();
   logic [3:0] in;
   logic [6:0] out;

   hex_driver dut (.in, .out);

   initial begin
   // Test some random inputs
      in = 4'h0; #10;
      in = 4'h3; #10;
      in = 4'h8; #10;
      in = 4'hb; #10;
      in = 4'hf; #10;
      $stop;
   end
endmodule
```

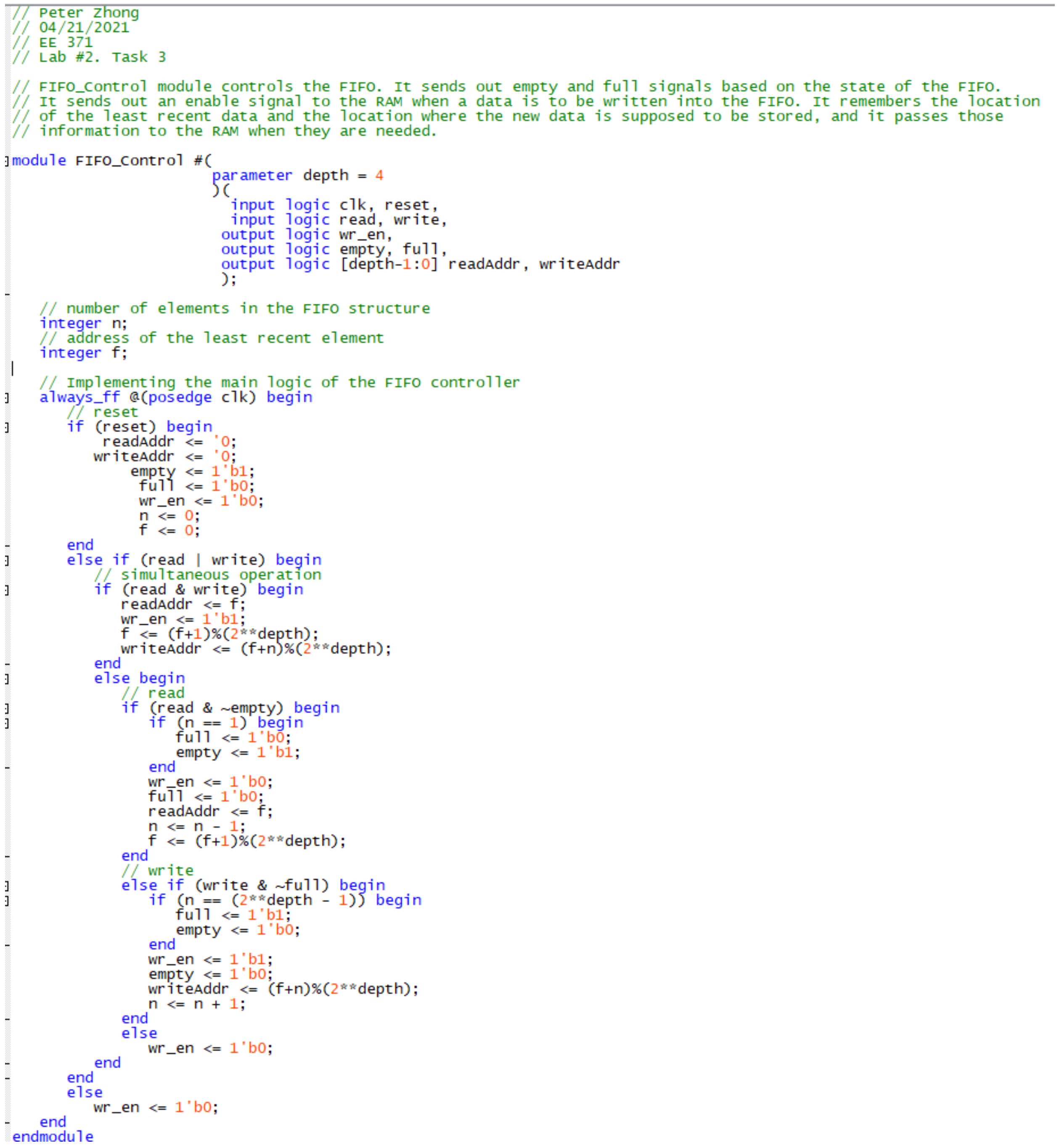
3) FIFO.sv



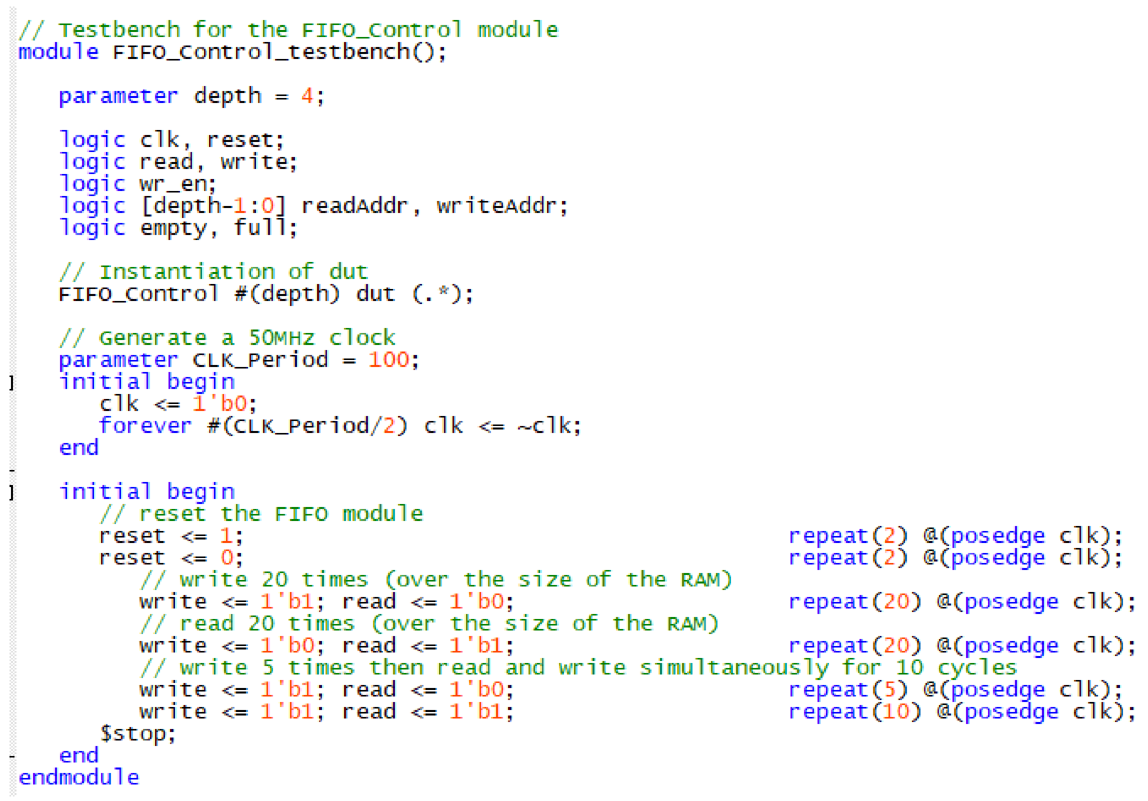
```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 3

// FIFO module implements a 16x8 queue-like FIFO structure.
// When a read signal is received, FIFO stores the data on the input bus if the FIFO is not full.
// When a write signal is received, FIFO outputs the least recent stored value onto the output bus,
// and removes that value from the FIFO.

module FIFO #(
             parameter depth = 4,
             parameter width = 8
             )(
               input logic clk, reset,
               input logic read, write,
               input logic [width-1:0] inputBus,
              output logic empty, full,
              output logic [width-1:0] outputBus
              );

   // variables for interconnections and communication between the RAM and the controller
   logic [3:0] addr_r, addr_w;
   logic wr_en;


   // Instantiation of the 16x8 RAm used for the FIFO
   ram16x8 RAM (.clock(clk), .data(inputBus), .rdaddress(addr_r), .wraddress(addr_w), .wren(wr_en), .q(outputBus));

   // Instantiation of the controller for the FIFO
   FIFO_Control #(depth) FC (.clk, .reset,
                             .read,
                             .write,
                             .wr_en,
                             .empty,
                             .full,
                             .readAddr(addr_r),
                             .writeAddr(addr_w)
                            );

endmodule
```

```
// Testbench for the FIFO module
`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 empty, full;
   logic [width-1:0] outputBus;

   // Instantiation of dut
   FIFO #(depth, width) dut (.*);

   // Generate a 50MHz clock
   parameter CLK_Period = 100;
   initial begin
      clk <= 1'b0;
      forever #(CLK_Period/2) clk <= ~clk;
   end

   initial begin
      // reset the FIFO module
      reset <= 1;                                      repeat(2) @(posedge clk);
      reset <= 0;                                      repeat(2) @(posedge clk);
         write <= 1'b1; read <= 1'b0;                            @(posedge clk);
      // fill up the entire memory w/ 8'h00 through 8'h0f
                                     inputBus <= 8'h00;         @(posedge clk);
                                     inputBus <= 8'h01;         @(posedge clk);
                                     inputBus <= 8'h02;         @(posedge clk);
                                     inputBus <= 8'h03;         @(posedge clk);
                                     inputBus <= 8'h04;         @(posedge clk);
                                     inputBus <= 8'h05;         @(posedge clk);
                                     inputBus <= 8'h06;         @(posedge clk);
                                     inputBus <= 8'h07;         @(posedge clk);
                                     inputBus <= 8'h08;         @(posedge clk);
                                     inputBus <= 8'h09;         @(posedge clk);
                                     inputBus <= 8'h0a;         @(posedge clk);
                                     inputBus <= 8'h0b;         @(posedge clk);
                                     inputBus <= 8'h0c;         @(posedge clk);
                                     inputBus <= 8'h0d;         @(posedge clk);
                                     inputBus <= 8'h0e;         @(posedge clk);
                                     inputBus <= 8'h0f;         @(posedge clk);
      // attempt to put in more data after the memory is full
                                     inputBus <= 8'h10;         @(posedge clk);
                                     inputBus <= 8'h11;         @(posedge clk);
                                     inputBus <= 8'h12;         @(posedge clk);
         write <= 1'b0;                                         @(posedge clk);
      // clear out the entire memory & attempt to read more data
                         read <= 1'b1;                repeat(20) @(posedge clk);
      // simultaneous read and write operations after writing
                                      inputBus <= 8'h00;         @(posedge clk);
         write <= 1'b1; read <= 1'b0; inputBus <= 8'h00;         @(posedge clk);
                                      inputBus <= 8'h01;         @(posedge clk);
                                      inputBus <= 8'h02;         @(posedge clk);
                                      inputBus <= 8'h03;         @(posedge clk);
                                      inputBus <= 8'h04;         @(posedge clk);
                                      inputBus <= 8'h05;         @(posedge clk);
         write <= 1'b1; read <= 1'b1; inputBus <= 8'h06;         @(posedge clk);
                                      inputBus <= 8'h07;         @(posedge clk);
                                      inputBus <= 8'h08;         @(posedge clk);
                                      inputBus <= 8'h09;         @(posedge clk);
                                      inputBus <= 8'h0a;         @(posedge clk);
                                      inputBus <= 8'h0b;         @(posedge clk);
                                      inputBus <= 8'h0c;         @(posedge clk);
                                      inputBus <= 8'h0d;         @(posedge clk);
                                      inputBus <= 8'h0e;         @(posedge clk);
                                      inputBus <= 8'h0f;         @(posedge clk);

      $stop;
   end
endmodule
```

4) FIFO_Control.sv

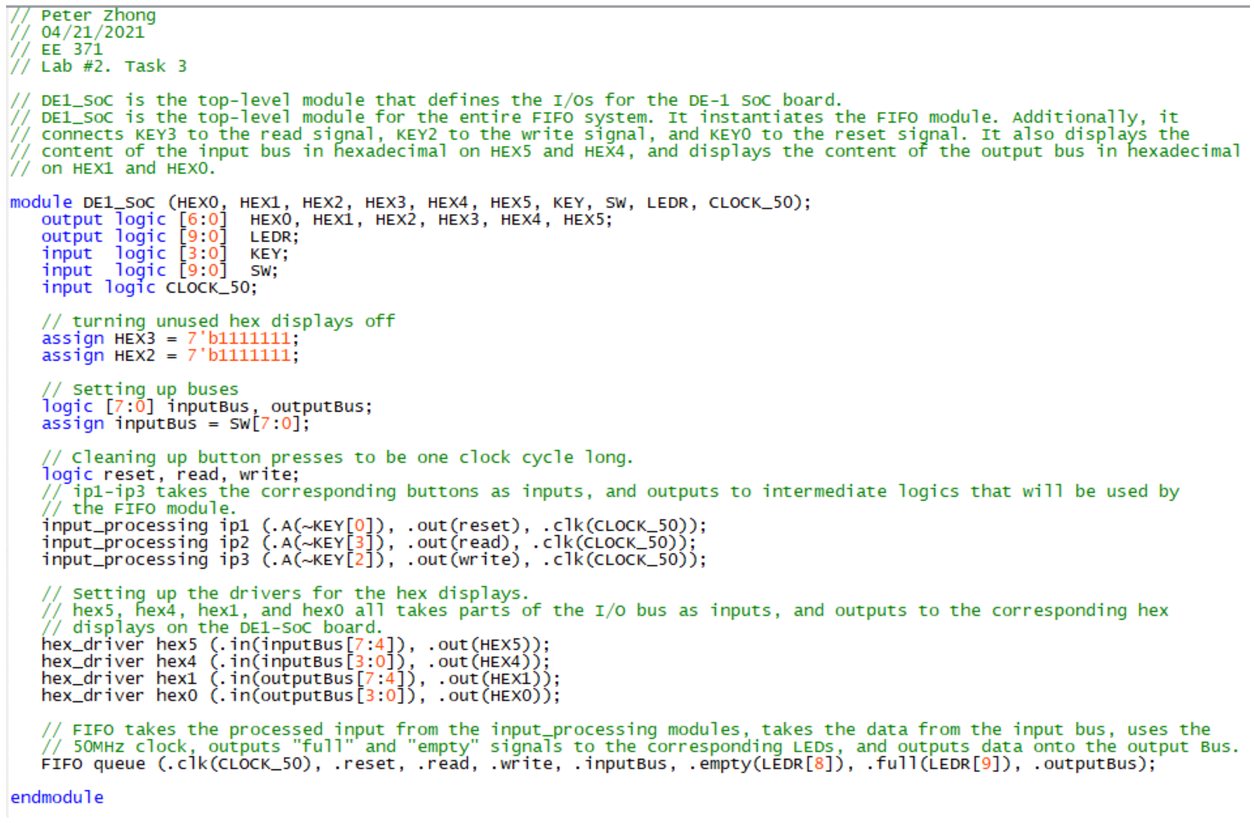


```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 3

// FIFO_Control module controls the FIFO. It sends out empty and full signals based on the state of the FIFO.
// It sends out an enable signal to the RAM when a data is to be written into the FIFO. It remembers the location
// of the least recent data and the location where the new data is supposed to be stored, and it passes those
// information to the RAM when they are needed.

module FIFO_Control #(
                       parameter depth = 4
                       )(
                         input logic clk, reset,
                         input logic read, write,
                        output logic wr_en,
                        output logic empty, full,
                        output logic [depth-1:0] readAddr, writeAddr
                        );

   // number of elements in the FIFO structure
   integer n;
   // address of the least recent element
   integer f;

   // Implementing the main logic of the FIFO controller
   always_ff @(posedge clk) begin
      // reset
      if (reset) begin
          readAddr <= '0;
         writeAddr <= '0;
             empty <= 1'b1;
              full <= 1'b0;
              wr_en <= 1'b0;
              n <= 0;
              f <= 0;
      end
      else if (read | write) begin
         // simultaneous operation
         if (read & write) begin
            readAddr <= f;
            wr_en <= 1'b1;
            f <= (f+1)%(2**depth);
            writeAddr <= (f+n)%(2**depth);
         end
         else begin
            // read
            if (read & ~empty) begin
               if (n == 1) begin
                  full <= 1'b0;
                  empty <= 1'b1;
               end
               wr_en <= 1'b0;
               full <= 1'b0;
               readAddr <= f;
               n <= n - 1;
               f <= (f+1)%(2**depth);
            end
            // write
            else if (write & ~full) begin
               if (n == (2**depth - 1)) begin
                  full <= 1'b1;
                  empty <= 1'b0;
               end
               wr_en <= 1'b1;
               empty <= 1'b0;
               writeAddr <= (f+n)%(2**depth);
               n <= n + 1;
            end
            else
               wr_en <= 1'b0;
         end
      end
      else
         wr_en <= 1'b0;
   end
endmodule
```



```
// Testbench for the FIFO_Control module
module FIFO_Control_testbench();

   parameter depth = 4;

   logic clk, reset;
   logic read, write;
   logic wr_en;
   logic [depth-1:0] readAddr, writeAddr;
   logic empty, full;

   // Instantiation of dut
   FIFO_Control #(depth) dut (.*);

   // Generate a 50MHz clock
   parameter CLK_Period = 100;
   initial begin
      clk <= 1'b0;
      forever #(CLK_Period/2) clk <= ~clk;
   end

   initial begin
      // reset the FIFO module
      reset <= 1;                                       repeat(2) @(posedge clk);
      reset <= 0;                                       repeat(2) @(posedge clk);
         // write 20 times (over the size of the RAM)
         write <= 1'b1; read <= 1'b0;                   repeat(20) @(posedge clk);
         // read 20 times (over the size of the RAM)
         write <= 1'b0; read <= 1'b1;                   repeat(20) @(posedge clk);
         // write 5 times then read and write simultaneously for 10 cycles
         write <= 1'b1; read <= 1'b0;                   repeat(5) @(posedge clk);
         write <= 1'b1; read <= 1'b1;                   repeat(10) @(posedge clk);
      $stop;
   end
endmodule
```

5)DE1_SoC.sv



```
// Peter Zhong
// 04/21/2021
// EE 371
// Lab #2. Task 3

// DE1_SoC is the top-level module that defines the I/Os for the DE-1 SoC board.
// DE1_SoC is the top-level module for the entire FIFO system. It instantiates the FIFO module. Additionally, it
// connects KEY3 to the read signal, KEY2 to the write signal, and KEY0 to the reset signal. It also displays the
// content of the input bus in hexadecimal on HEX5 and HEX4, and displays the content of the output bus in hexadecimal
// on HEX1 and HEX0.

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

   // turning unused hex displays off
   assign HEX3 = 7'b1111111;
   assign HEX2 = 7'b1111111;

   // Setting up buses
   logic [7:0] inputBus, outputBus;
   assign inputBus = SW[7:0];

   // Cleaning up button presses to be one clock cycle long.
   logic reset, read, write;
   // ip1-ip3 takes the corresponding buttons as inputs, and outputs to intermediate logics that will be used by
   // the FIFO module.
   input_processing ip1 (.A(~KEY[0]), .out(reset), .clk(CLOCK_50));
   input_processing ip2 (.A(~KEY[3]), .out(read), .clk(CLOCK_50));
   input_processing ip3 (.A(~KEY[2]), .out(write), .clk(CLOCK_50));

   // Setting up the drivers for the hex displays.
   // hex5, hex4, hex1, and hex0 all takes parts of the I/O bus as inputs, and outputs to the corresponding hex
   // displays on the DE1-SoC board.
   hex_driver hex5 (.in(inputBus[7:4]), .out(HEX5));
   hex_driver hex4 (.in(inputBus[3:0]), .out(HEX4));
   hex_driver hex1 (.in(outputBus[7:4]), .out(HEX1));
   hex_driver hex0 (.in(outputBus[3:0]), .out(HEX0));

   // FIFO takes the processed input from the input_processing modules, takes the data from the input bus, uses the
   // 50MHz clock, outputs "full" and "empty" signals to the corresponding LEDs, and outputs data onto the output Bus.
   FIFO queue (.clk(CLOCK_50), .reset, .read, .write, .inputBus, .empty(LEDR[8]), .full(LEDR[9]), .outputBus);

endmodule
```

```
`timescale 1 ps / 1 ps
// Testbench for DE1_SoC module
module DE1_SoC_testbench();
   logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0]  LEDR;
   logic [3:0]  KEY;
   logic [9:0]  SW;
   logic clk;

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

   // Setting up logics to make the code more readable.
   logic reset, read, write;
   logic [7:0] inputBus;
   assign KEY[0] = ~reset;
   assign KEY[3] = ~read;
   assign KEY[2] = ~write;
   assign SW[7:0] = inputBus;

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

   initial begin
      reset <= 1; repeat(5) @(posedge clk);
      // put in excessive data;
      reset <= 0; read <= 0; write <= 1; inputBus <= 8'h00;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h01;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h02;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h03;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h04;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h05;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h06;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h07;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h08;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h09;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0a;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0b;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0c;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0d;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0e;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h0f;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             // attempt to write more data
                             write <= 1; inputBus <= 8'h10;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h11;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h12;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h13;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
                             write <= 1; inputBus <= 8'h14;           @(posedge clk);
                             write <= 0;                              @(posedge clk);
      // [illegible]
```

```
                             write <= 0;                                      @(posedge clk);
      // attempt to read off 20 sets of data
                  read  <= 1; write <= 0;                                     @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);
                  read  <= 1;                                                 @(posedge clk);
                  read  <= 0;                                                 @(posedge clk);

      // simultaneous read/write
      reset <= 0; read  <= 0; write <= 1; inputBus <= 8'h00;                  @(posedge clk);
                              write <= 0;                                     @(posedge clk);
                              write <= 1; inputBus <= 8'h01;                  @(posedge clk);
                              write <= 0;                                     @(posedge clk);
                              write <= 1; inputBus <= 8'h02;                  @(posedge clk);
                              write <= 0;                                     @(posedge clk);
                              write <= 1; inputBus <= 8'h03;                  @(posedge clk);
                              write <= 0;                                     @(posedge clk);
                              write <= 1; inputBus <= 8'h04;                  @(posedge clk);
                              write <= 0;                                     @(posedge clk);
      reset <= 0; read  <= 1; write <= 1; inputBus <= 8'h05;                  @(posedge clk);
                  read  <= 0; write <= 0;                                     @(posedge clk);
                  read  <= 1; write <= 1; inputBus <= 8'h06;                  @(posedge clk);
                  read  <= 0; write <= 0;                                     @(posedge clk);
                  read  <= 1; write <= 1; inputBus <= 8'h07;                  @(posedge clk);
                  read  <= 0; write <= 0;                                     @(posedge clk);
                  read  <= 1; write <= 1; inputBus <= 8'h08;                  @(posedge clk);
                  read  <= 0; write <= 0;                                     @(posedge clk);
                  read  <= 1; write <= 1; inputBus <= 8'h09;                  @(posedge clk);
                  read  <= 0; write <= 0;                                     @(posedge clk);
      $stop;
   end
endmodule
```