Kiana Peterson and Nattapon Oonlmaom
EE 371
January 24, 2023
Lab 2 Report

Procedure
In this lab, the purpose was to implement random access memory (RAM) and display its behavior using the FPGA. To accomplish this, we develop a block diagram and state diagram as necessary. After seeing what component was necessary, we them began writing modules, such as a RAM, counter, seg7, etc. This lab is composed of three tasks with different sets of instruction, so the modules are used accordingly to accomplish each task.

Task 1
In Task 1, we implemented a 32x4 array, which contains 32 words and 4 bits per word in the RAM. SW 3-0 was used to provide input data for the RAM and SW -4 to specify which address the data is stored while SW is a write signal and KEY 0 as the clock to help see the simulation clearly as we can count the number of clock cycle this way. The behavior will then be displayed on the HEXs. HEX5-4 show the address, HEX2 shows data beng put in, and HEX0 shows the read out data from the memory. To accomplish this, we followed the block diagram below.

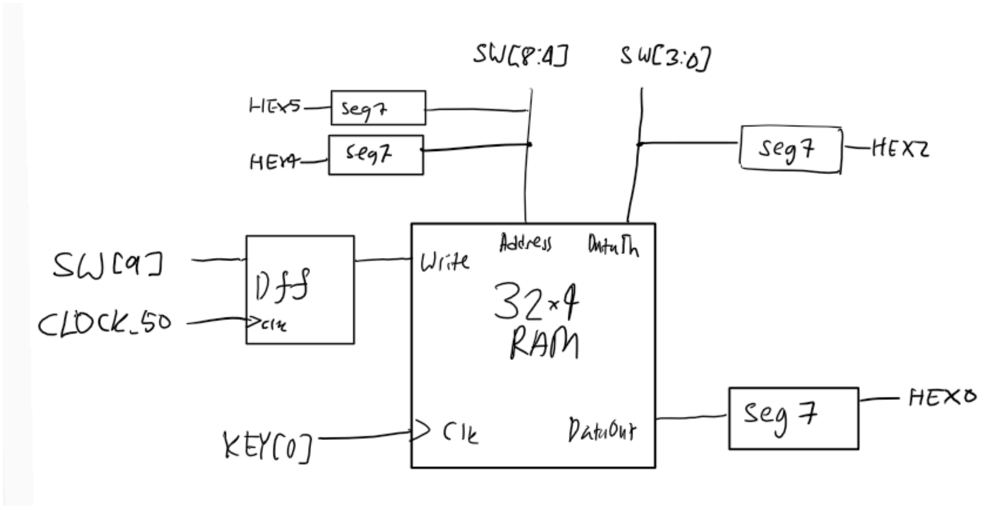


Figure 1.1 Block Diagram of Task 1

The system will be composed of the modules Task1, dflipflop, and seg7, which will be instantiated in the main module DE1_SoC. Inputs are passed through the dflipflop, then through the RAM Task1 to give a 2 clock cycle delay and be stored into memory. Then, the output from the RAM in connected to the seg7 to be displayed on HEXs. We implemented the system this way as is the most logical to solve the given problem.

Task 2
In Task 2, we created the dual-port 32x4 RAM and the ra32x4 mif file as instructed. Then, we implement a RAM for supplying the address for a read operation and a separate port that gives the address for a write operation. SW8-4 is used to write address and SW3-0 is for the corresponding data. The HEXs displays were used to display the content of each four-bit word. HEX0 shuffles through the data stored in mif while HEX3-2

shuffles through its address (from 0-1f in hexadecimal), and HEX1 shows the write data. To accomplish this, we followed the block diagram below.

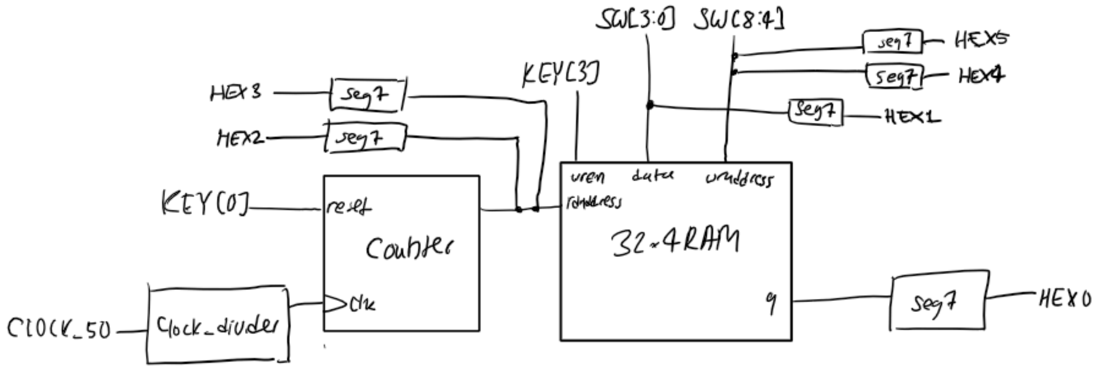


Figure 2.1 Block Diagram of Task 2

The system will be composed of the modules clock_divider, counter, and seg7, and the ram32x4 created using Quartus, which will be instantiated in the main module DE1_SoC. The clock_divider set up the clock, the counter is passed through the ram32x4 adder to keep counting up the address every 1 sec or so. The ram32x4 then outputs the data. The data, address, and write are connected to seg7 to be displayed on HEXs. We implemented the system this way as is the most logical to solve the given problem. After considering how the components will be connected, we determined how each component will work using a state diagram.

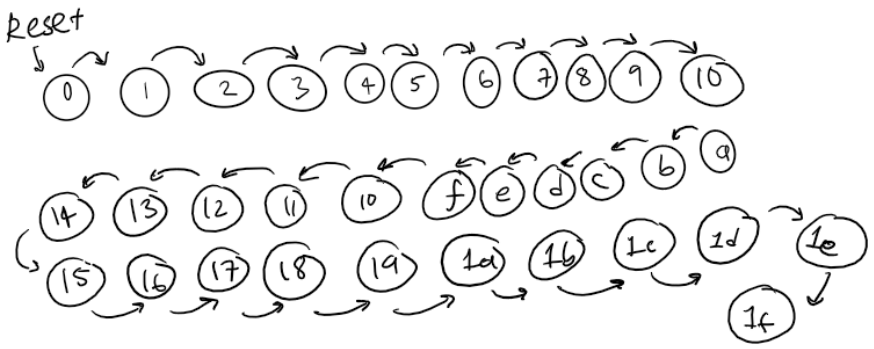


Figure 2.2 State Diagram of Task 2

Task 3

In Task 3, a circular queue was created to store memory to track whether the dual-port RAM from Task 2 is full, empty, or neither. We used the FIFO and its skeleton provided and designed an FSM for the FIFO controller to track the FIFO capacity and update its memory.

For this task, the value of the data input should be shown on HEX5-4. SW7-0 to represent the data input while the current data output is shown on HEX1-0. When “full,” LEDR9 is indicated. When “empty,” LEDR8 is indicated. CLOCK_50 (50MHz) is also used as the input clk to this FIFO. To represent this, the block diagram and the state diagram can be seen below.

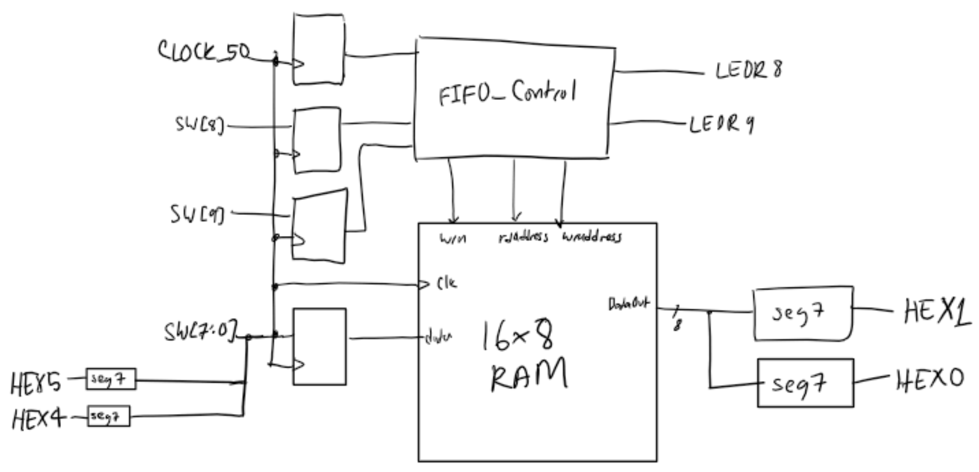


Figure 2.3 Block Diagram of Task 3

After struggling with many attempts in debugging, we implemented the system this way as is the most logical to solve the given problem. After considering how the components will be connected, we determined how each component will work using a state diagram.

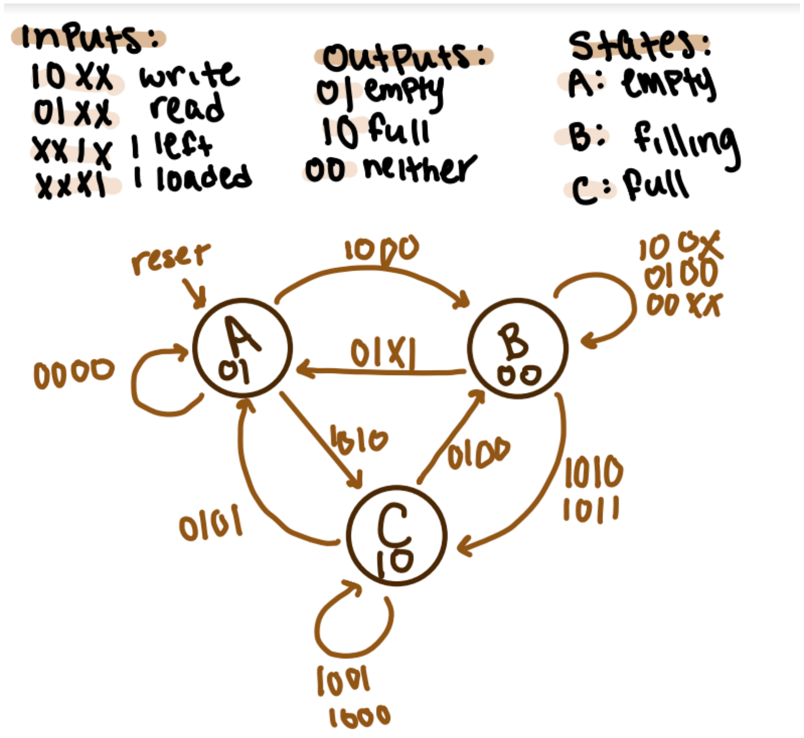


Figure 2.4 State Diagram of Task 3 (made for unknown size)

Results

Task #1:

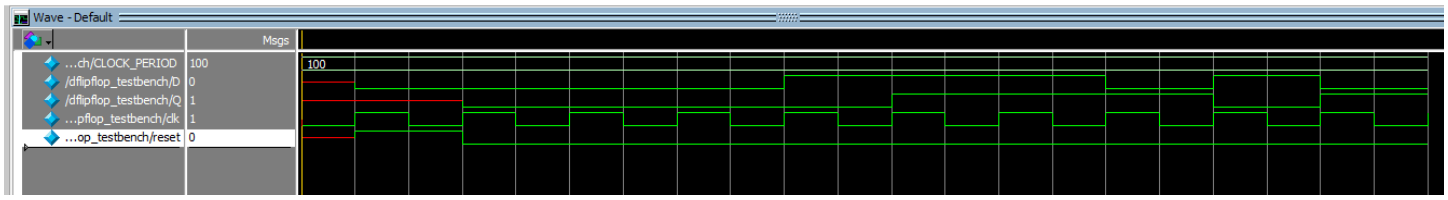


Figure 3.1 dflipflop.sv module Waveform

The dflipflop module works as intended. It takes in an input, set output to 0 when reset and return output delay by a clock cycle.

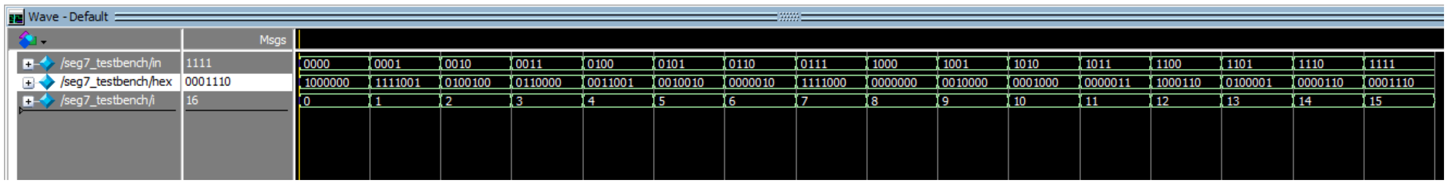


Figure 3.2 seg7.sv module Waveform

The seg7 module functions correctly as expected. The HEX display shows numbers correctly according to their assigned binary.

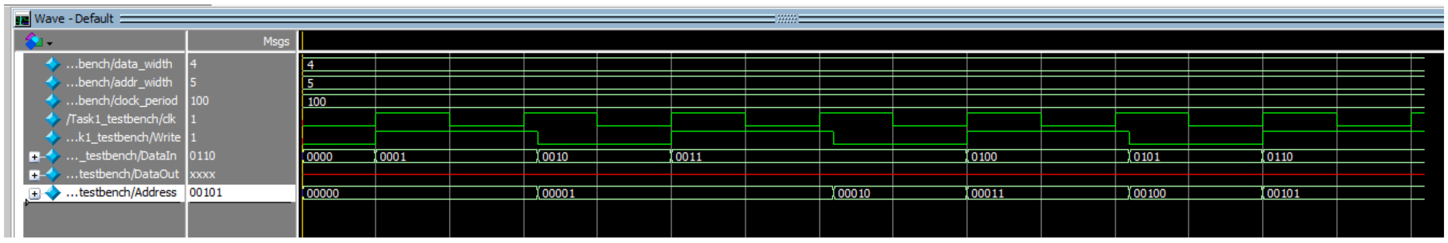


Figure 3.3 Task1.sv module Waveform (RAM1)

The Task1 (the RAM) works correctly. It shows the data values are stored in the correct assigned address in the testbench. When write, the read out data is shown accordingly.

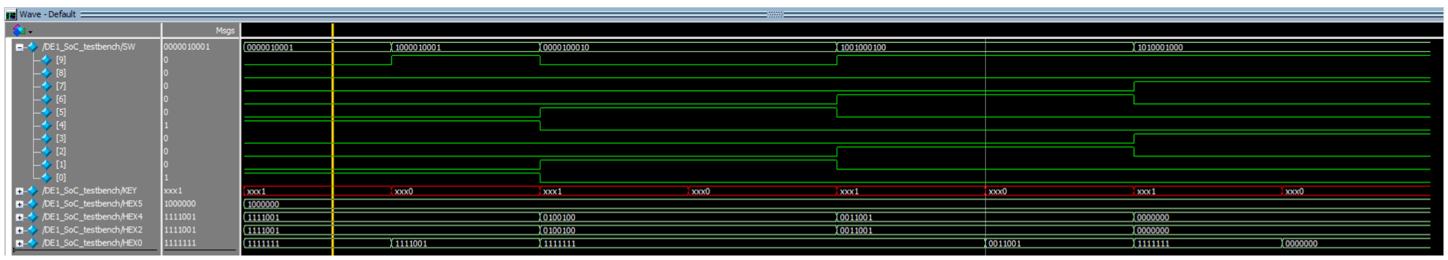


Figure 3.4 DE1_SoC.sv module Waveform Task1

The DE1_SoC module is functioning correctly as it is displaying the correct output from the inputs given on the testbench and aligned with the truth table. Pressing KEY[0] is a clock cycle. HEXs are displaying the values accordingly as it is supposed to.

Task #2:

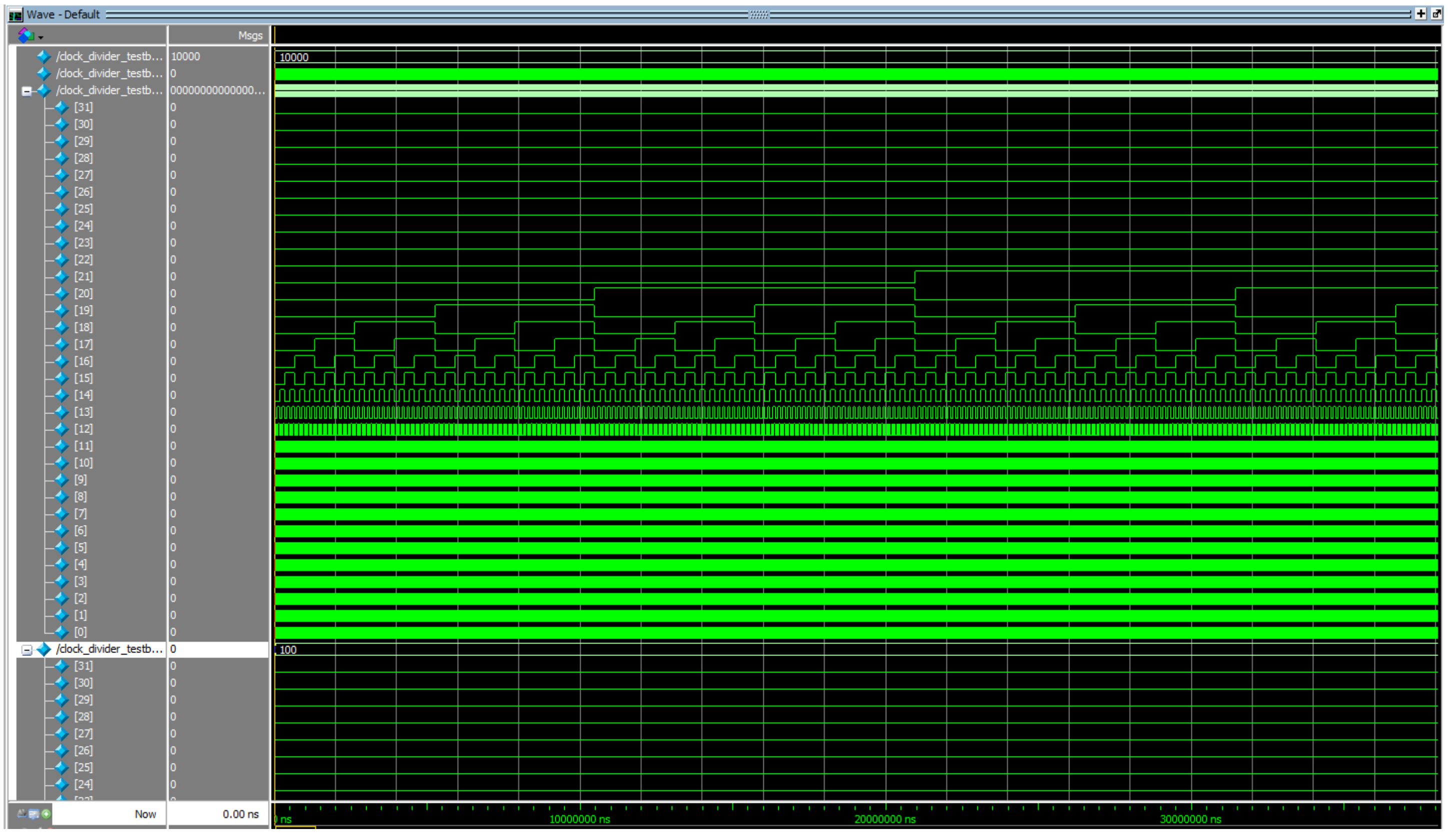


Figure 4.1 clock_divider.sv module Waveform

The clock divider works as expected. It is slowing down the clock by dividing as named.

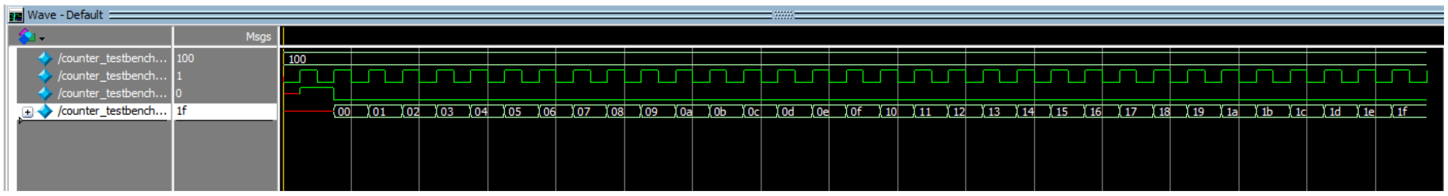

Wave - Default
Msgs
/counter_testbench... 100
/counter_testbench... 1
/counter_testbench... 0
/counter_testbench... 1f
100
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

Figure 4.2 counter.sv module Waveform

The counter module is working as expected. The counter counts up by 1 (from 0 to 1f) and reset when reset is input or when the max is reached as designed by the state diagram.

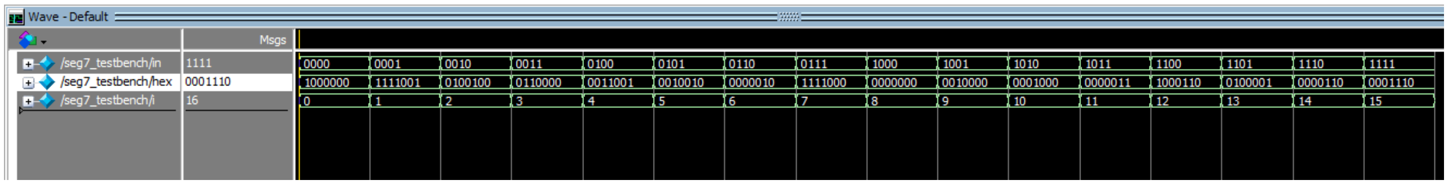
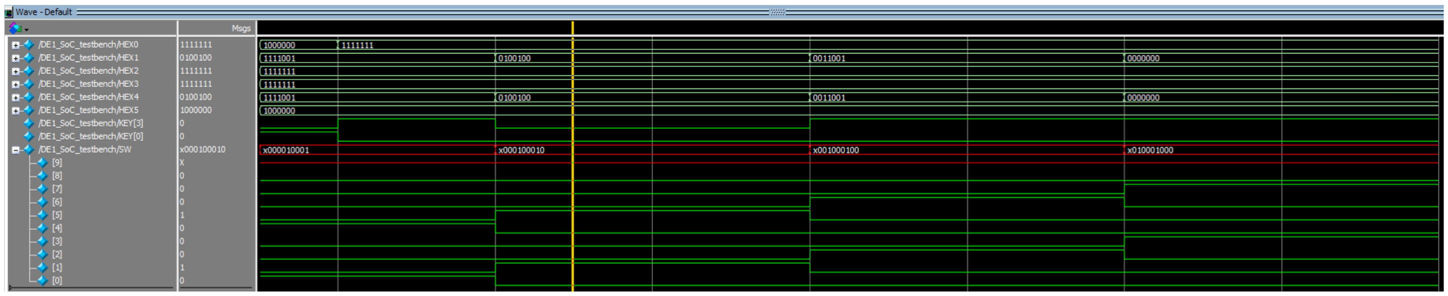


Figure 4.3 seg7.sv module Waveform

The seg7 module functions correctly as expected. The HEX display shows numbers correctly according to their assigned binary.


Wave - Default
Msgs
/DE1_SoC_testbench/HEX0 1111111
/DE1_SoC_testbench/HEX1 0100100
/DE1_SoC_testbench/HEX2 1111111
/DE1_SoC_testbench/HEX3 1111111
/DE1_SoC_testbench/HEX4 0100100
/DE1_SoC_testbench/HEX5 1000000
/DE1_SoC_testbench/KEY[3] 0
/DE1_SoC_testbench/KEY[0] 0
/DE1_SoC_testbench/SW x000100010
[9] X
[8] 0
[7] 0
[6] 0
[5] 1
[4] 0
[3] 0
[2] 0
[1] 1
[0] 0
1000000
1111111
1111001
0100100
0011001
0000000
x000010001
x000100010
x001000100
x010001000
Figure 4.4 DE1_SoC.sv module Waveform Task2

The DE1_SoC module is functioning correctly as it is displaying the correct output from the inputs given on the testbench and aligned with the truth table. The HEXs are displaying counter, write, read, and addresses correctly accordingly to the inputs as it supposed to.

Task #3:

Figure 5.1 FIFO.sv module Waveform

The FIFO module worked as expected according to the waveform as it uses a memory module to store data when written to and ejects data when read from as it is supposed to.

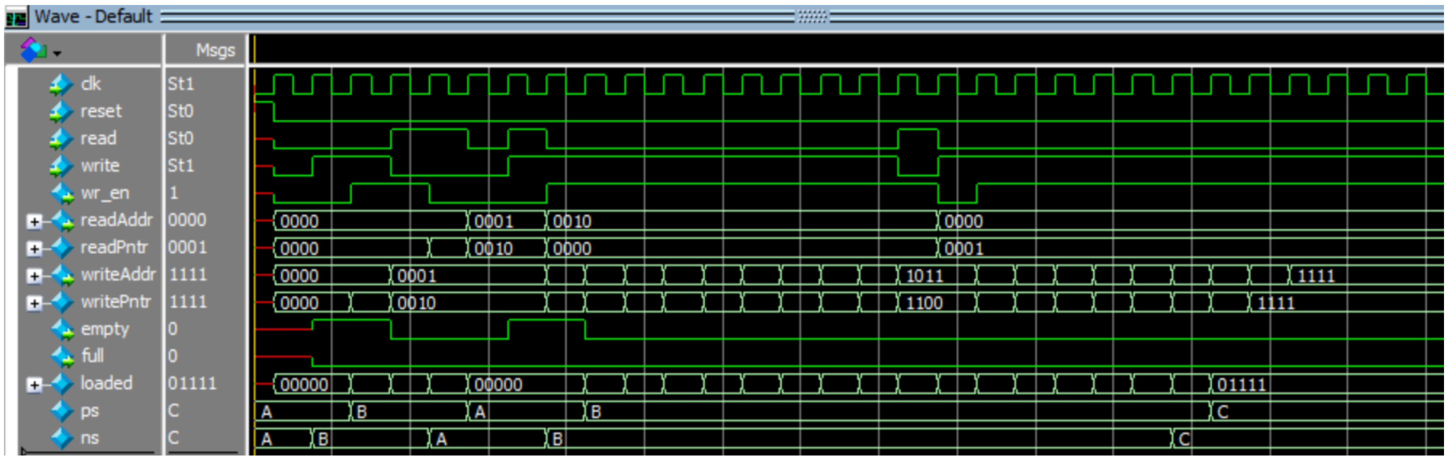


Figure 5.2 FIFO_Control.sv Waveform

The FIFO_control module is working as expected as it keeping track of the RAM's addresse and its memory. It shows empty when there is no queue and full when the memory input is full as it supposed to.

Figure 5.3 DE1_SoC.sv module Waveform Task3

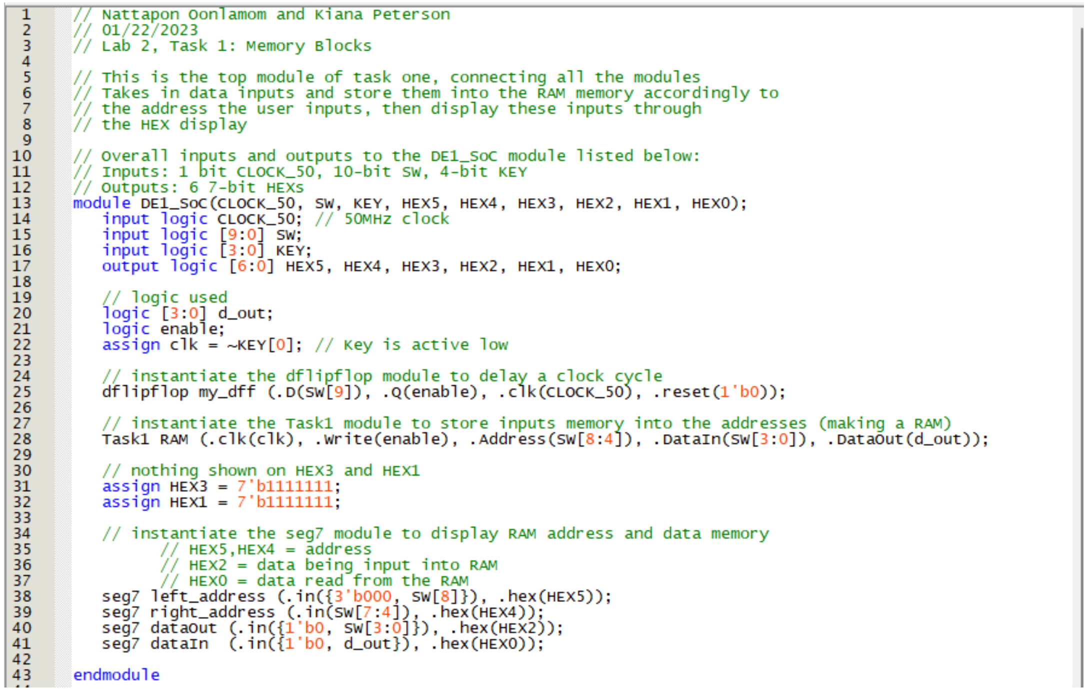
The DE1_SoC module is functioning correctly as it is displaying the correct output from the inputs given on the testbench and aligned with the truth table. LEDR8 light up when the queue is empty and LEDR9 lights up when the queue is full while HEXs are displaying the values correctly as assigned.

Final Product

Demonstrate Link:
Task1 and Task2: https://drive.google.com/file/d/1-gNmK_42Nm4zY4kdKmXbrJshkUZpl2Lk/view
Task3: https://drive.google.com/file/d/1qWkCMBE5rnCgPX7uw83cFmrJVltv89V8/view?usp=sharing

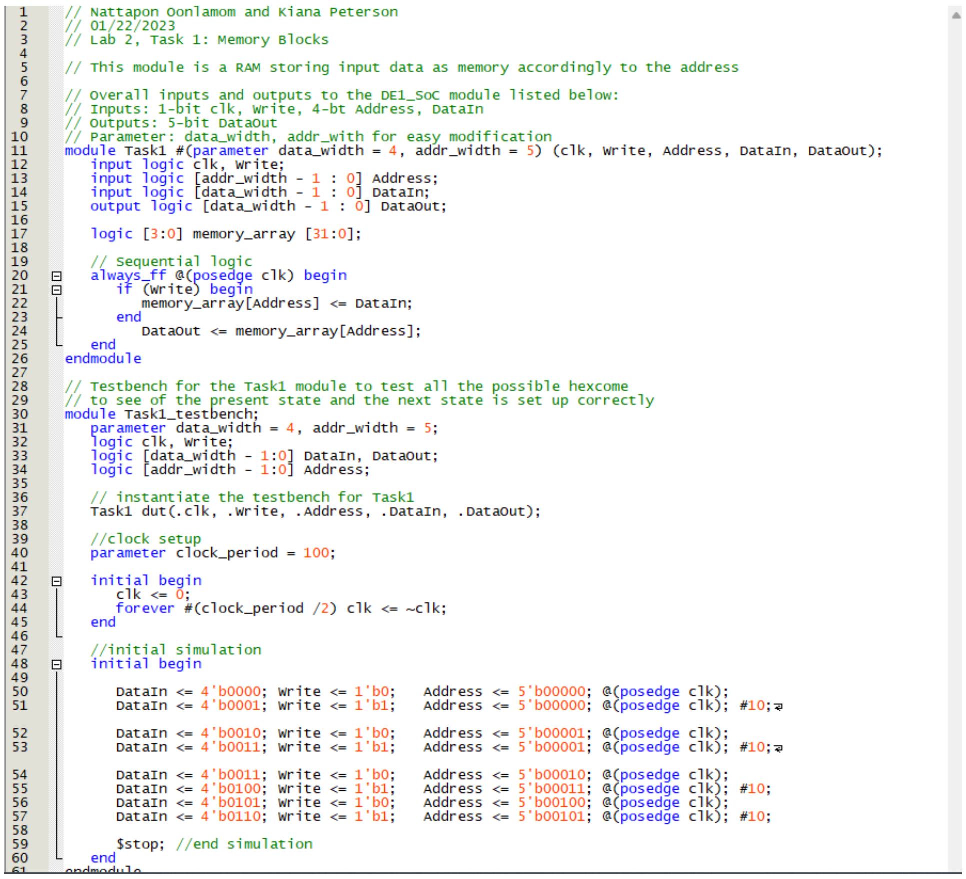
The overall goal of this lab was to be able to understand how RAM works and how to implement and use them. In Task 1, we built the memory block RAM from scratch. As a result, we made a 32x4 RAM that stores the user data memory input into the address provided by the user. In Task 2, we used the built-in memory block from Quartus to investigate the implementation of a dual-port RAM. In doing this, we implemented a RAM for supplying the address for a read operation and a separate port that gives the address for a write operation. In Task 3, we added a controller onto the memory block we learned about. The final product gives a memory module that only writes in information if there is room and will output whatever data is stored in the least recent stored address. To conclude, we learned how RAMs operate and how to implement them.

Appendix



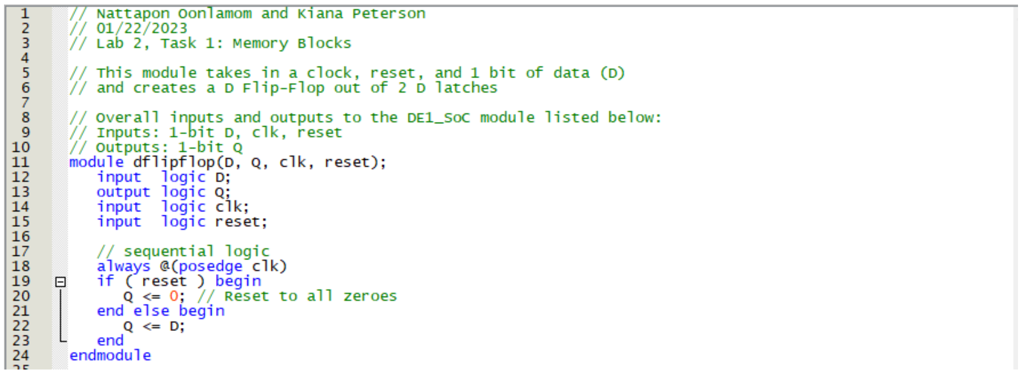
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2, Task 1: Memory Blocks

// This is the top module of task one, connecting all the modules
// Takes in data inputs and store them into the RAM memory accordingly to
// the address the user inputs, then display these inputs through
// the HEX display

// Overall inputs and outputs to the DE1_SoC module listed below:
// Inputs: 1 bit CLOCK_50, 10-bit SW, 4-bit KEY
// Outputs: 6 7-bit HEXs
module DE1_SoC(CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input logic CLOCK_50; // 50MHz clock
   input logic [9:0] SW;
   input logic [3:0] KEY;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   // logic used
   logic [3:0] d_out;
   logic enable;
   assign clk = ~KEY[0]; // Key is active low

   // instantiate the dflipflop module to delay a clock cycle
   dflipflop my_dff (.D(SW[9]), .Q(enable), .clk(CLOCK_50), .reset(1'b0));

   // instantiate the Task1 module to store inputs memory into the addresses (making a RAM)
   Task1 RAM (.clk(clk), .Write(enable), .Address(SW[8:4]), .DataIn(SW[3:0]), .DataOut(d_out));

   // nothing shown on HEX3 and HEX1
   assign HEX3 = 7'b1111111;
   assign HEX1 = 7'b1111111;

   // instantiate the seg7 module to display RAM address and data memory
         // HEX5,HEX4 = address
         // HEX2 = data being input into RAM
         // HEX0 = data read from the RAM
   seg7 left_address (.in({3'b000, SW[8]}), .hex(HEX5));
   seg7 right_address (.in(SW[7:4]), .hex(HEX4));
   seg7 dataOut (.in({1'b0, SW[3:0]}), .hex(HEX2));
   seg7 dataIn  (.in({1'b0, d_out}), .hex(HEX0));

endmodule
```

Figure 6 DE1_SoC.sv module for Task 1



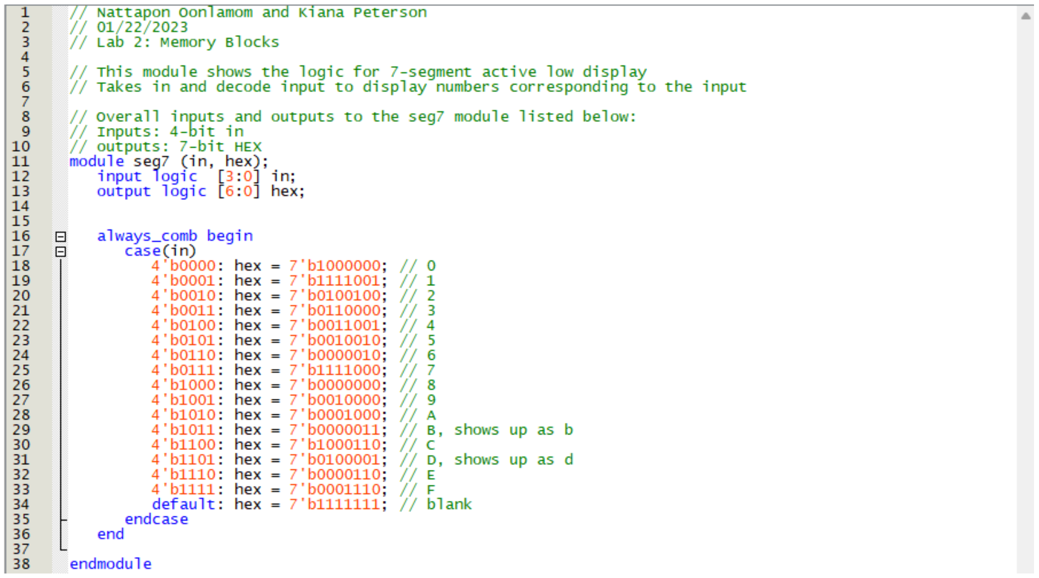
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2, Task 1: Memory Blocks

// This module is a RAM storing input data as memory accordingly to the address

// Overall inputs and outputs to the DE1_SoC module listed below:
// Inputs: 1-bit clk, Write, 4-bt Address, DataIn
// Outputs: 5-bit DataOut
// Parameter: data_width, addr_with for easy modification
module Task1 #(parameter data_width = 4, addr_width = 5) (clk, Write, Address, DataIn, DataOut);
   input logic clk, Write;
   input logic [addr_width - 1 : 0] Address;
   input logic [data_width - 1 : 0] DataIn;
   output logic [data_width - 1 : 0] DataOut;

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

   // Sequential logic
   always_ff @(posedge clk) begin
      if (Write) begin
         memory_array[Address] <= DataIn;
      end
         DataOut <= memory_array[Address];
   end
endmodule

// Testbench for the Task1 module to test all the possible hexcome
// to see of the present state and the next state is set up correctly
module Task1_testbench;
   parameter data_width = 4, addr_width = 5;
   logic clk, Write;
   logic [data_width - 1:0] DataIn, DataOut;
   logic [addr_width - 1:0] Address;

   // instantiate the testbench for Task1
   Task1 dut(.clk, .Write, .Address, .DataIn, .DataOut);

   //clock setup
   parameter clock_period = 100;

   initial begin
      clk <= 0;
      forever #(clock_period /2) clk <= ~clk;
   end

   //initial simulation
   initial begin

      DataIn <= 4'b0000; Write <= 1'b0;   Address <= 5'b00000; @(posedge clk);
      DataIn <= 4'b0001; Write <= 1'b1;   Address <= 5'b00000; @(posedge clk); #10;

      DataIn <= 4'b0010; Write <= 1'b0;   Address <= 5'b00001; @(posedge clk);
      DataIn <= 4'b0011; Write <= 1'b1;   Address <= 5'b00001; @(posedge clk); #10;

      DataIn <= 4'b0011; Write <= 1'b0;   Address <= 5'b00010; @(posedge clk);
      DataIn <= 4'b0100; Write <= 1'b1;   Address <= 5'b00011; @(posedge clk); #10;
      DataIn <= 4'b0101; Write <= 1'b0;   Address <= 5'b00100; @(posedge clk);
      DataIn <= 4'b0110; Write <= 1'b1;   Address <= 5'b00101; @(posedge clk); #10;

      $stop; //end simulation
   end
endmodule
```

Figure 7 Task1.sv module (RAM1)



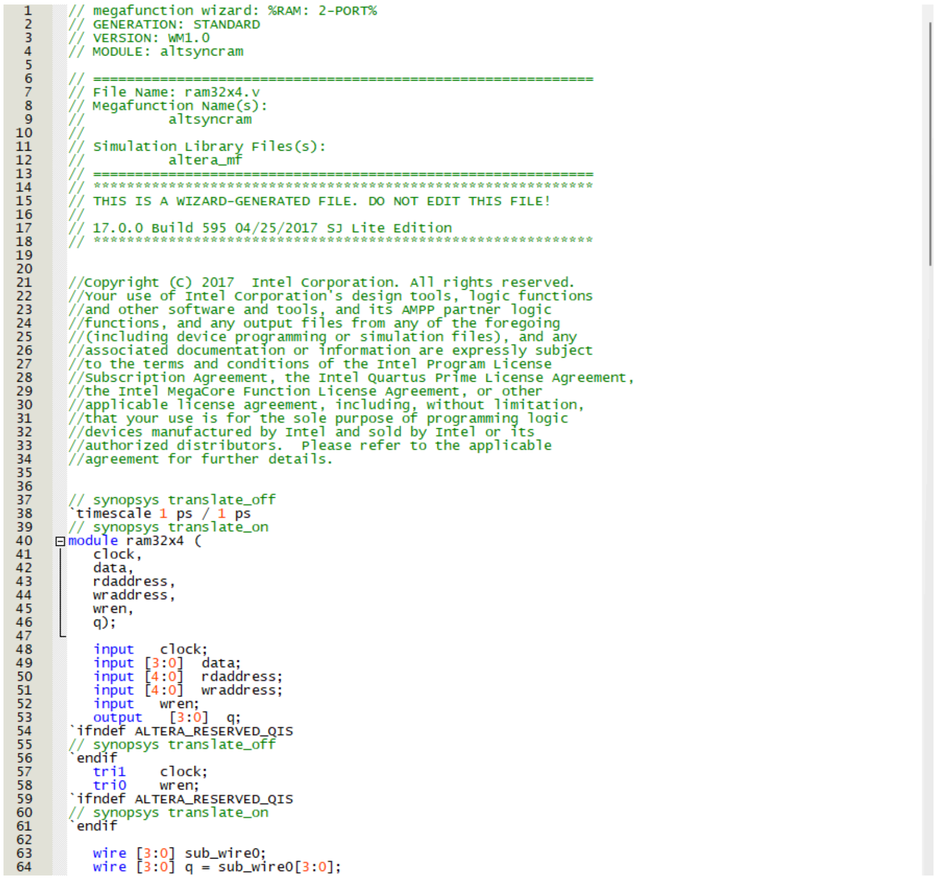
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2, Task 1: Memory Blocks

// This module takes in a clock, reset, and 1 bit of data (D)
// and creates a D Flip-Flop out of 2 D latches

// Overall inputs and outputs to the DE1_SoC module listed below:
// Inputs: 1-bit D, clk, reset
// Outputs: 1-bit Q
module dflipflop(D, Q, clk, reset);
   input  logic D;
   output logic Q;
   input  logic clk;
   input  logic reset;

   // sequential logic
   always @(posedge clk)
   if ( reset ) begin
      Q <= 0; // Reset to all zeroes
   end else begin
      Q <= D;
   end
endmodule
```

Figure 8 dflipflop.sv module

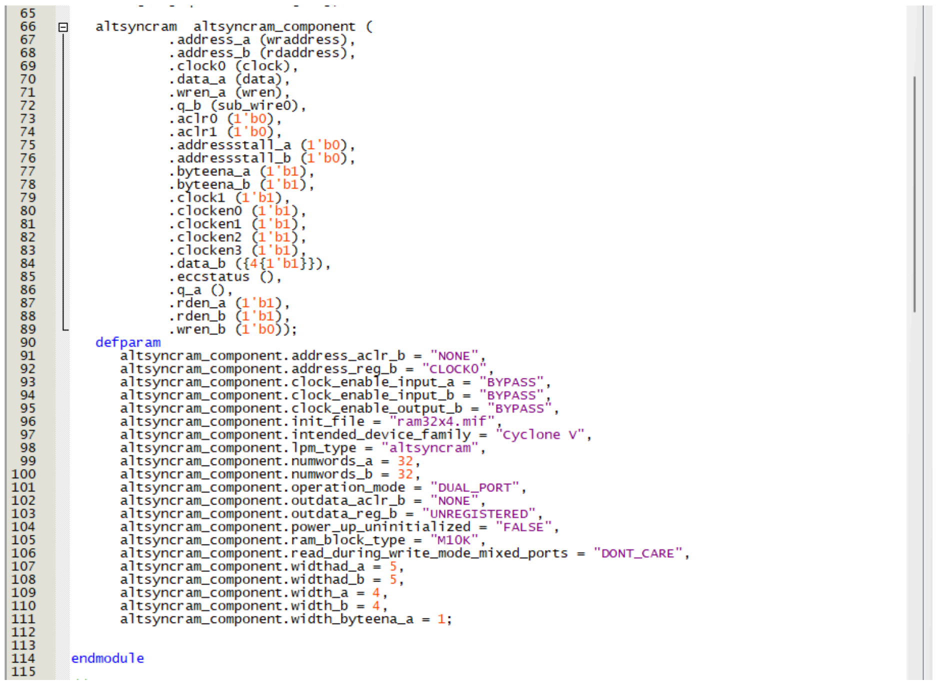


```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2: Memory Blocks

// This module shows the logic for 7-segment active low display
// Takes in and decode input to display numbers corresponding to the input

// Overall inputs and outputs to the seg7 module listed below:
// Inputs: 4-bit in
// outputs: 7-bit HEX
module seg7 (in, hex);
   input logic  [3:0] in;
   output logic [6:0] hex;


   always_comb begin
      case(in)
         4'b0000: hex = 7'b1000000; // 0
         4'b0001: hex = 7'b1111001; // 1
         4'b0010: hex = 7'b0100100; // 2
         4'b0011: hex = 7'b0110000; // 3
         4'b0100: hex = 7'b0011001; // 4
         4'b0101: hex = 7'b0010010; // 5
         4'b0110: hex = 7'b0000010; // 6
         4'b0111: hex = 7'b1111000; // 7
         4'b1000: hex = 7'b0000000; // 8
         4'b1001: hex = 7'b0010000; // 9
         4'b1010: hex = 7'b0001000; // A
         4'b1011: hex = 7'b0000011; // B, shows up as b
         4'b1100: hex = 7'b1000110; // C
         4'b1101: hex = 7'b0100001; // D, shows up as d
         4'b1110: hex = 7'b0000110; // E
         4'b1111: hex = 7'b0001110; // F
         default: hex = 7'b1111111; // blank
      endcase
   end

endmodule
```

Figure 9 seg7.sv module

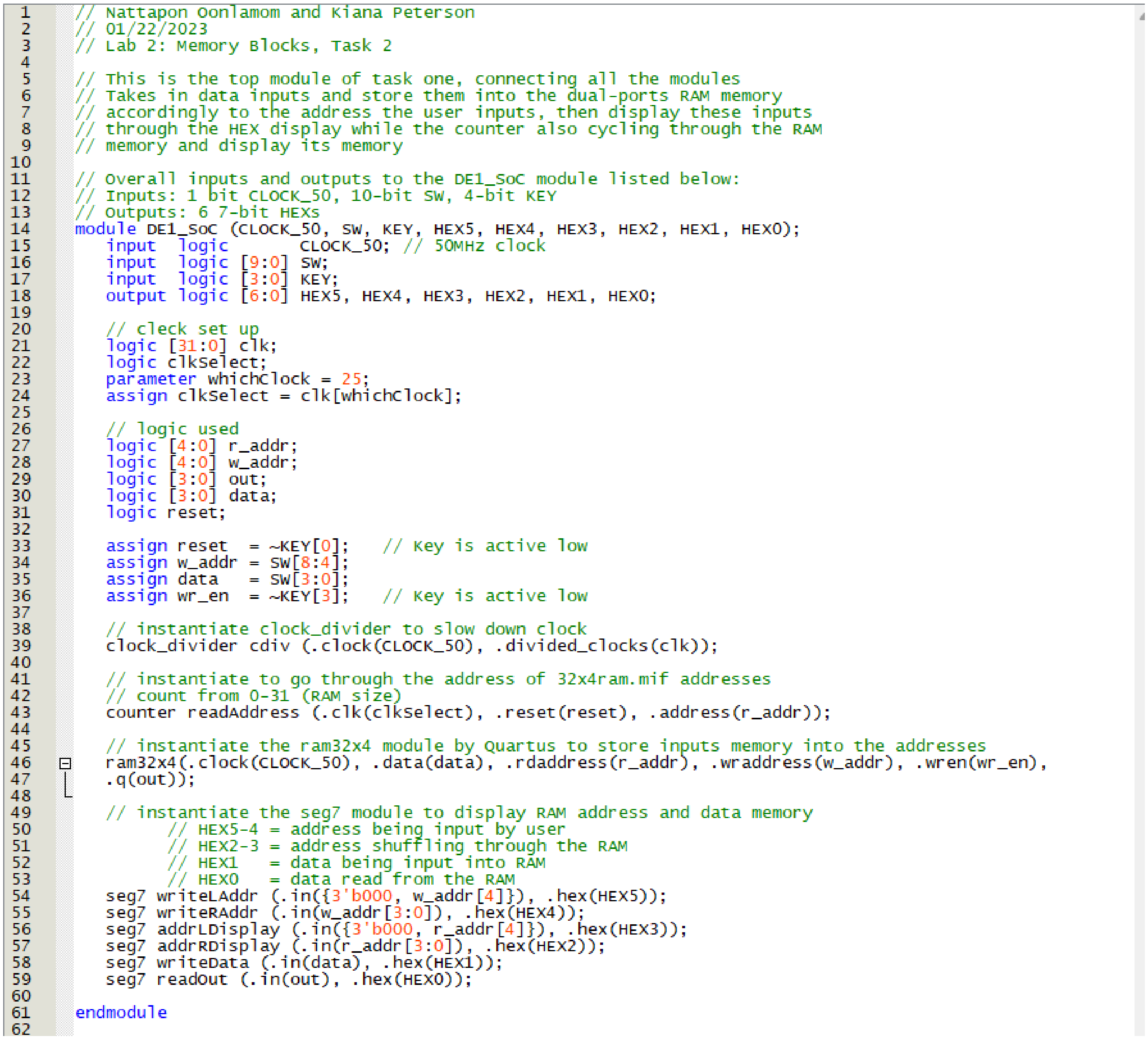


```verilog
// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: ram32x4.v
// Megafunction Name(s):
//          altsyncram
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 17.0.0 Build 595 04/25/2017 SJ Lite Edition
// ************************************************************


//Copyright (C) 2017  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Intel and sold by Intel or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.


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

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

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



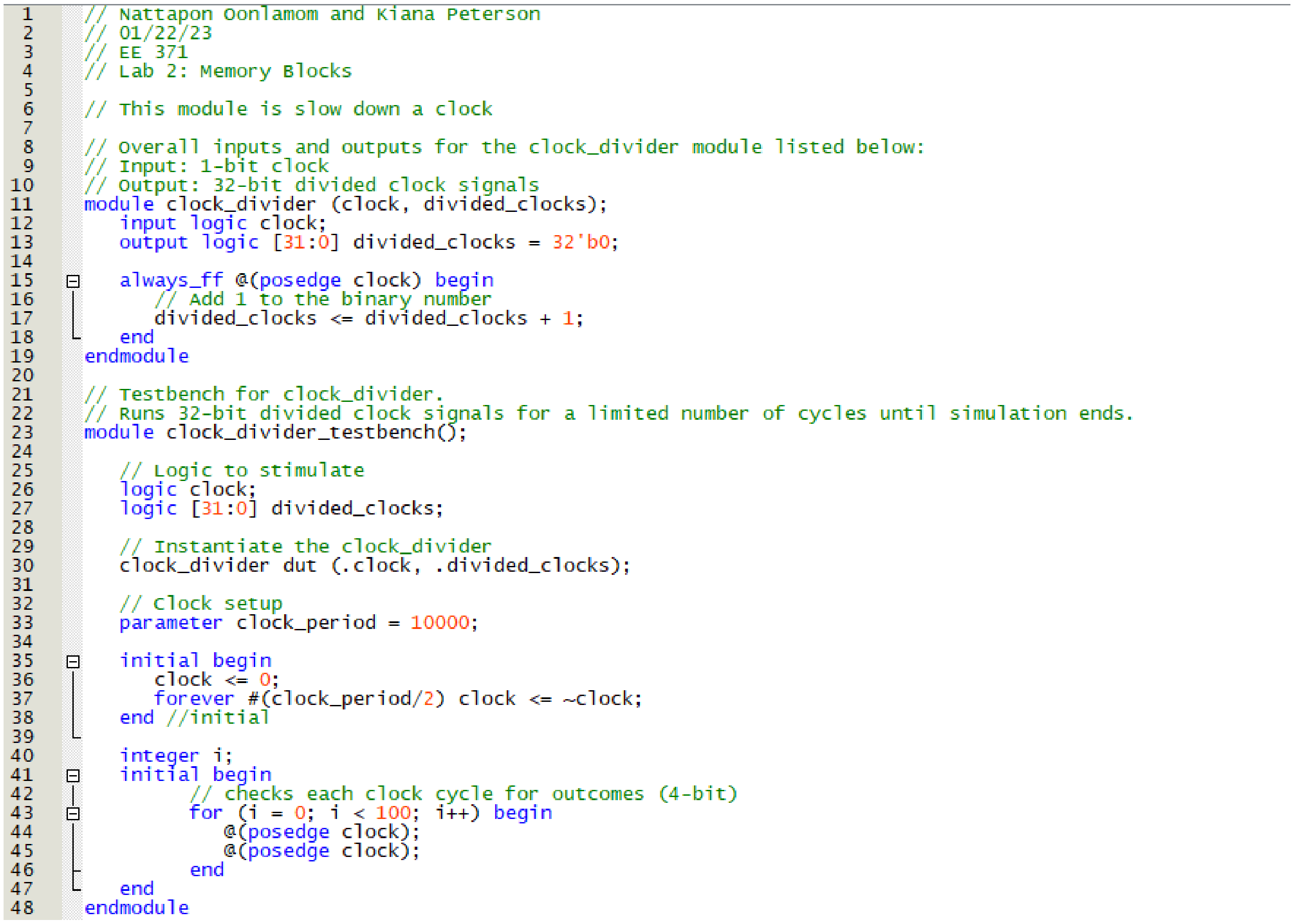
```verilog

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


endmodule
```

Figure 10 ram32x4.v module



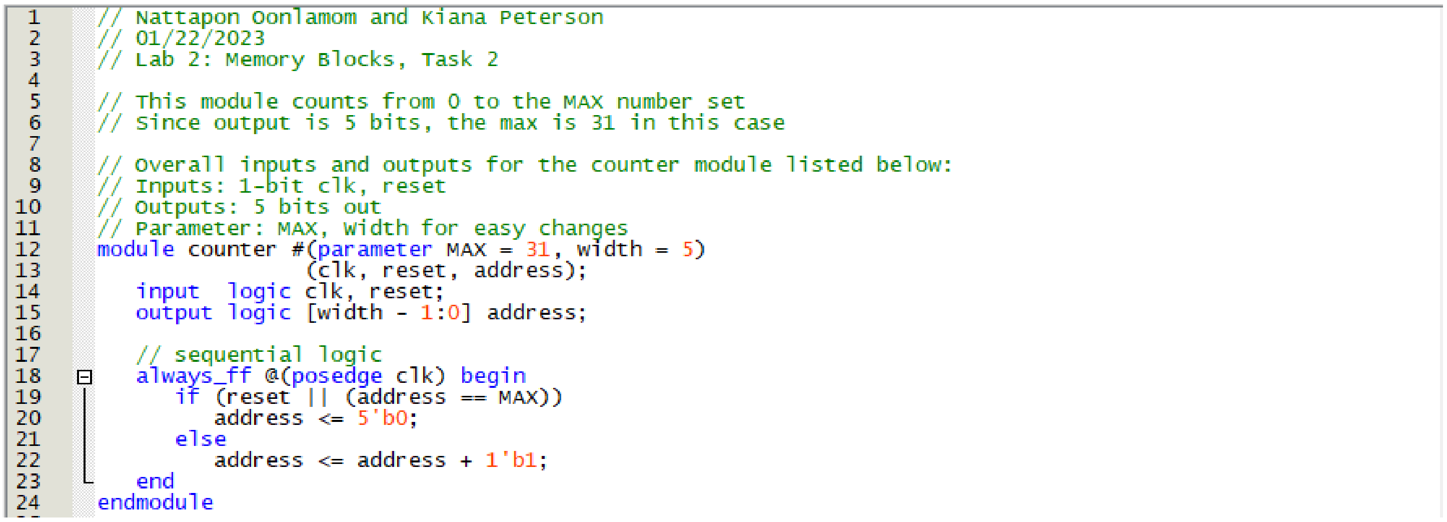
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2: Memory Blocks, Task 2

// This is the top module of task one, connecting all the modules
// Takes in data inputs and store them into the dual-ports RAM memory
// accordingly to the address the user inputs, then display these inputs
// through the HEX display while the counter also cycling through the RAM
// memory and display its memory

// Overall inputs and outputs to the DE1_SoC module listed below:
// Inputs: 1 bit CLOCK_50, 10-bit SW, 4-bit KEY
// Outputs: 6 7-bit HEXs
module DE1_SoC (CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input  logic       CLOCK_50; // 50MHz clock
   input  logic [9:0] SW;
   input  logic [3:0] KEY;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   // cleck set up
   logic [31:0] clk;
   logic clkSelect;
   parameter whichClock = 25;
   assign clkSelect = clk[whichClock];

   // logic used
   logic [4:0] r_addr;
   logic [4:0] w_addr;
   logic [3:0] out;
   logic [3:0] data;
   logic reset;

   assign reset  = ~KEY[0];    // Key is active low
   assign w_addr = SW[8:4];
   assign data   = SW[3:0];
   assign wr_en  = ~KEY[3];    // Key is active low

   // instantiate clock_divider to slow down clock
   clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(clk));

   // instantiate to go through the address of 32x4ram.mif addresses
   // count from 0-31 (RAM size)
   counter readAddress (.clk(clkSelect), .reset(reset), .address(r_addr));

   // instantiate the ram32x4 module by Quartus to store inputs memory into the addresses
   ram32x4(.clock(CLOCK_50), .data(data), .rdaddress(r_addr), .wraddress(w_addr), .wren(wr_en),
   .q(out));

   // instantiate the seg7 module to display RAM address and data memory
         // HEX5-4 = address being input by user
         // HEX2-3 = address shuffling through the RAM
         // HEX1   = data being input into RAM
         // HEX0   = data read from the RAM
   seg7 writeLAddr (.in({3'b000, w_addr[4]}), .hex(HEX5));
   seg7 writeRAddr (.in(w_addr[3:0]), .hex(HEX4));
   seg7 addrLDisplay (.in({3'b000, r_addr[4]}), .hex(HEX3));
   seg7 addrRDisplay (.in(r_addr[3:0]), .hex(HEX2));
   seg7 writeData (.in(data), .hex(HEX1));
   seg7 readOut (.in(out), .hex(HEX0));

endmodule
```

Figure 11 DE1_SoC.sv module for Task 2



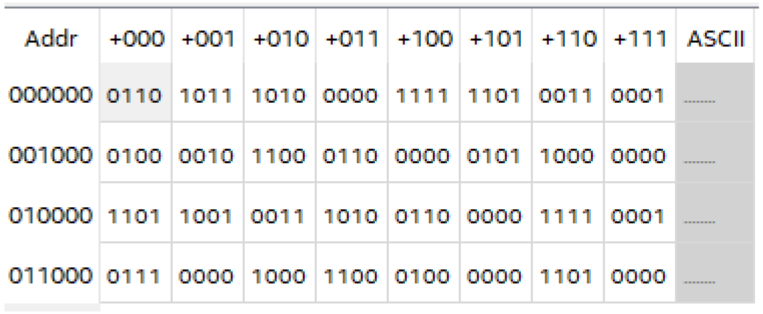
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/23
// EE 371
// Lab 2: Memory Blocks

// This module is slow down a clock

// Overall inputs and outputs for the clock_divider module listed below:
// Input: 1-bit clock
// Output: 32-bit divided clock signals
module clock_divider (clock, divided_clocks);
   input logic clock;
   output logic [31:0] divided_clocks = 32'b0;

   always_ff @(posedge clock) begin
      // Add 1 to the binary number
      divided_clocks <= divided_clocks + 1;
   end
endmodule

// Testbench for clock_divider.
// Runs 32-bit divided clock signals for a limited number of cycles until simulation ends.
module clock_divider_testbench();

   // Logic to stimulate
   logic clock;
   logic [31:0] divided_clocks;

   // Instantiate the clock_divider
   clock_divider dut (.clock, .divided_clocks);

   // Clock setup
   parameter clock_period = 10000;

   initial begin
      clock <= 0;
      forever #(clock_period/2) clock <= ~clock;
   end //initial

   integer i;
   initial begin
        // checks each clock cycle for outcomes (4-bit)
        for (i = 0; i < 100; i++) begin
           @(posedge clock);
           @(posedge clock);
        end
   end
endmodule
```

Figure 12 clock_divider.sv module



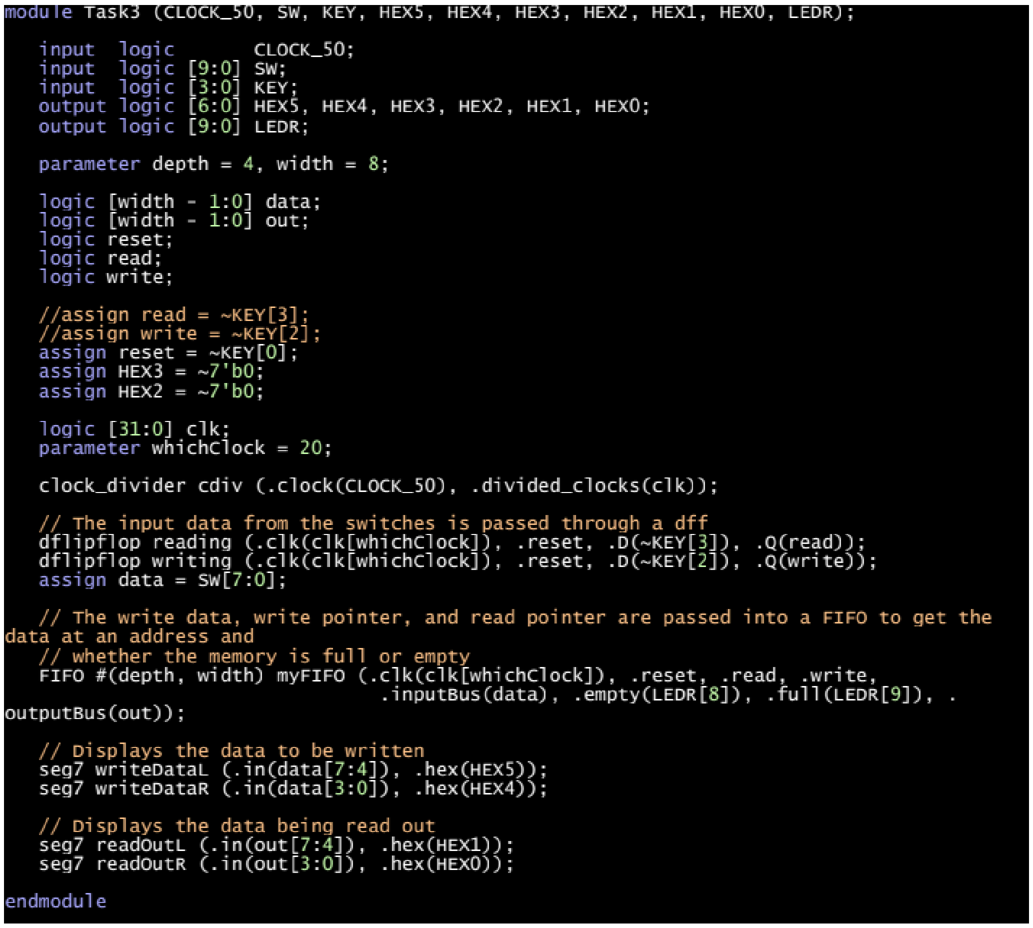
```
// Nattapon Oonlamom and Kiana Peterson
// 01/22/2023
// Lab 2: Memory Blocks, Task 2

// This module counts from 0 to the MAX number set
// Since output is 5 bits, the max is 31 in this case

// Overall inputs and outputs for the counter module listed below:
// Inputs: 1-bit clk, reset
// Outputs: 5 bits out
// Parameter: MAX, width for easy changes
module counter #(parameter MAX = 31, width = 5)
                (clk, reset, address);
   input  logic clk, reset;
   output logic [width - 1:0] address;

   // sequential logic
   always_ff @(posedge clk) begin
      if (reset || (address == MAX))
         address <= 5'b0;
      else
         address <= address + 1'b1;
   end
endmodule
```

Figure 13 counter.sv module



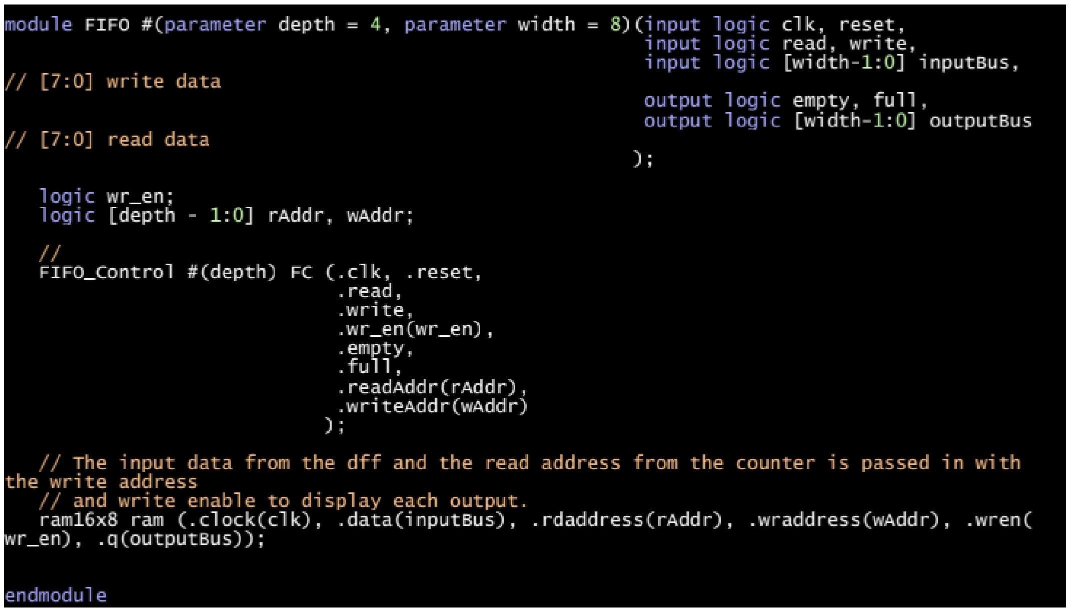
Addr	+000	+001	+010	+011	+100	+101	+110	+111	ASCII
000000	0110	1011	1010	0000	1111	1101	0011	0001	....
001000	0100	0010	1100	0110	0000	0101	1000	0000	....
010000	1101	1001	0011	1010	0110	0000	1111	0001	....
011000	0111	0000	1000	1100	0100	0000	1101	0000	....

Figure 14 32x4ram.mif



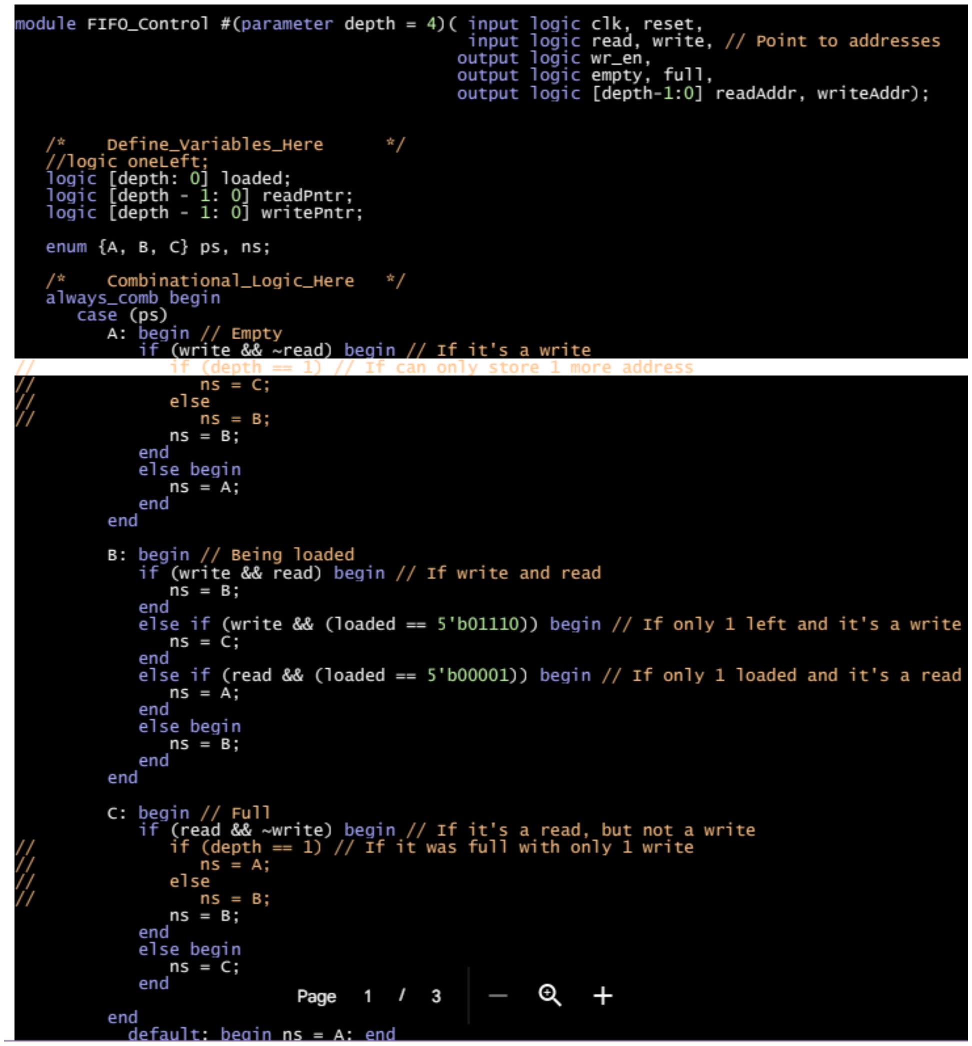
```
module Task3 (CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDR);

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

   parameter depth = 4, width = 8;

   logic [width - 1:0] data;
   logic [width - 1:0] out;
   logic reset;
   logic read;
   logic write;

   //assign read = ~KEY[3];
   //assign write = ~KEY[2];
   assign reset = ~KEY[0];
   assign HEX3 = ~7'b0;
   assign HEX2 = ~7'b0;

   logic [31:0] clk;
   parameter whichClock = 20;

   clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(clk));

   // The input data from the switches is passed through a dff
   dflipflop reading (.clk(clk[whichClock]), .reset, .D(~KEY[3]), .Q(read));
   dflipflop writing (.clk(clk[whichClock]), .reset, .D(~KEY[2]), .Q(write));
   assign data = SW[7:0];

   // The write data, write pointer, and read pointer are passed into a FIFO to get the
data at an address and
   // whether the memory is full or empty
   FIFO #(depth, width) myFIFO (.clk(clk[whichClock]), .reset, .read, .write,
                                 .inputBus(data), .empty(LEDR[8]), .full(LEDR[9]), .
outputBus(out));

   // Displays the data to be written
   seg7 writeDataL (.in(data[7:4]), .hex(HEX5));
   seg7 writeDataR (.in(data[3:0]), .hex(HEX4));

   // Displays the data being read out
   seg7 readOutL (.in(out[7:4]), .hex(HEX1));
   seg7 readOutR (.in(out[3:0]), .hex(HEX0));

endmodule
```

Figure 15 Task3.sv module

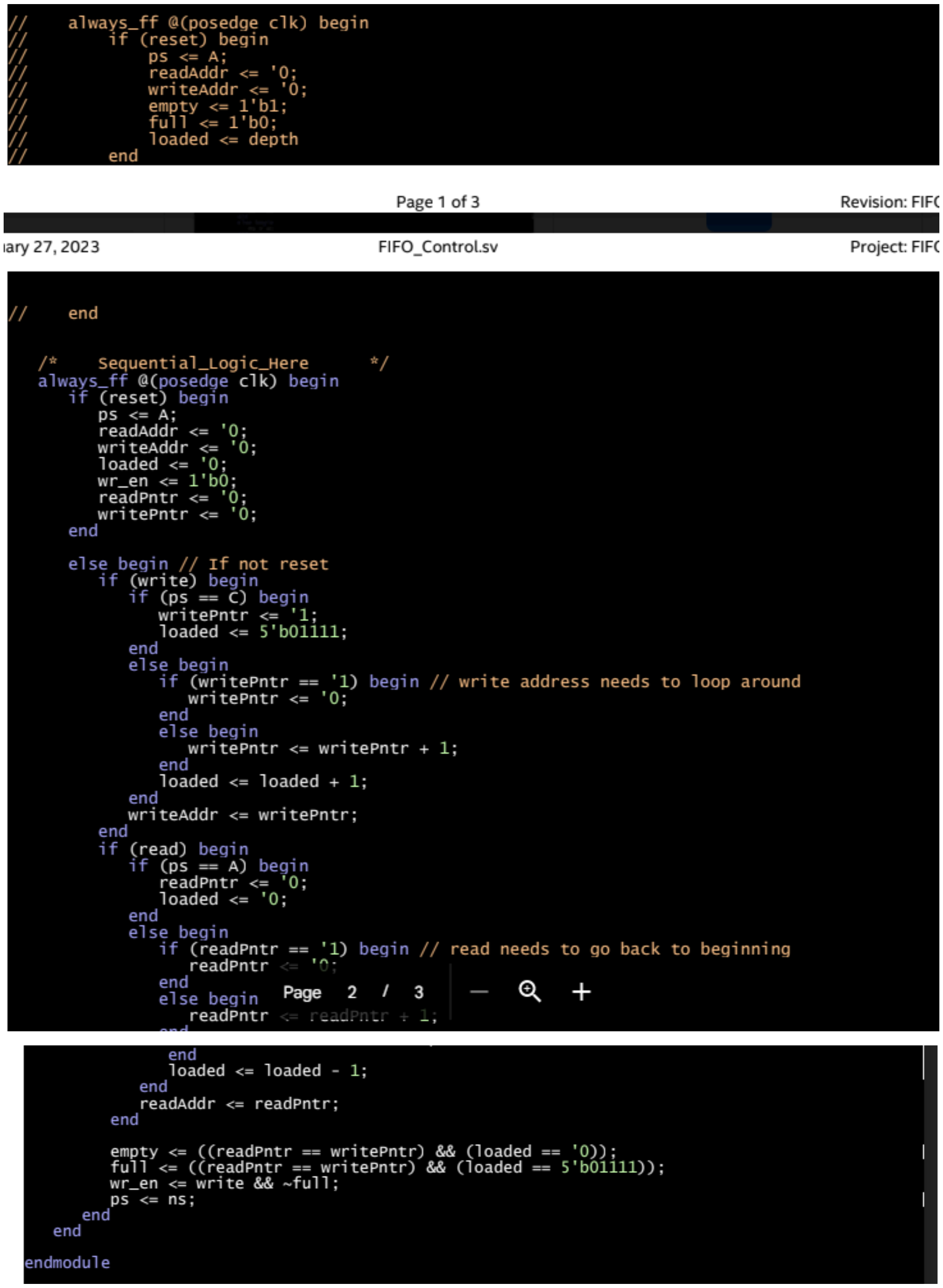


```
module FIFO #(parameter depth = 4, parameter width = 8)(input logic clk, reset,
                                                         input logic read, write,
                                                         input logic [width-1:0] inputBus,
// [7:0] write data
                                                         output logic empty, full,
                                                         output logic [width-1:0] outputBus
// [7:0] read data
                                                        );

   logic wr_en;
   logic [depth - 1:0] rAddr, wAddr;

   //
   FIFO_Control #(depth) FC (.clk, .reset,
                             .read,
                             .write,
                             .wr_en(wr_en),
                             .empty,
                             .full,
                             .readAddr(rAddr),
                             .writeAddr(wAddr)
                            );

   // The input data from the dff and the read address from the counter is passed in with
the write address
   // and write enable to display each output.
   ram16x8 ram (.clock(clk), .data(inputBus), .rdaddress(rAddr), .wraddress(wAddr), .wren(
wr_en), .q(outputBus));

endmodule
```

Figure 16 FIFO.sv module



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


   /*    Define_Variables_Here      */
   //logic oneLeft;
   logic [depth: 0] loaded;
   logic [depth - 1: 0] readPntr;
   logic [depth - 1: 0] writePntr;

   enum {A, B, C} ps, ns;

   /*    Combinational_Logic_Here   */
   always_comb begin
      case (ps)
         A: begin // Empty
            if (write && ~read) begin // If it's a write
//             if (depth == 1) // If can only store 1 more address
//                ns = C;
//             else
//                ns = B;
               ns = B;
            end
            else begin
               ns = A;
            end
         end

         B: begin // Being loaded
            if (write && read) begin // If write and read
               ns = B;
            end
            else if (write && (loaded == 5'b01110)) begin // If only 1 left and it's a write
               ns = C;
            end
            else if (read && (loaded == 5'b00001)) begin // If only 1 loaded and it's a read
               ns = A;
            end
            else begin
               ns = B;
            end
         end

         C: begin // Full
            if (read && ~write) begin // If it's a read, but not a write
//             if (depth == 1) // If it was full with only 1 write
//                ns = A;
//             else
//                ns = B;
               ns = B;
            end
            else begin
               ns = C;
            end

         end
          default: begin ns = A; end
```



```
//    always_ff @(posedge clk) begin
//        if (reset) begin
//            ps <= A;
//            readAddr <= '0;
//            writeAddr <= '0;
//            empty <= 1'b1;
//            full <= 1'b0;
//            loaded <= depth
//        end
```

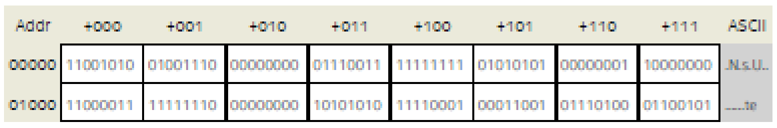
Page 1 of 3 Revision: FIFC

ıary 27, 2023 FIFO_Control.sv Project: FIFC

```
//    end

   /*    Sequential_Logic_Here      */
   always_ff @(posedge clk) begin
      if (reset) begin
         ps <= A;
         readAddr <= '0;
         writeAddr <= '0;
         loaded <= '0;
         wr_en <= 1'b0;
         readPntr <= '0;
         writePntr <= '0;
      end

      else begin // If not reset
         if (write) begin
            if (ps == C) begin
               writePntr <= '1;
               loaded <= 5'b01111;
            end
            else begin
               if (writePntr == '1) begin // write address needs to loop around
                  writePntr <= '0;
               end
               else begin
                  writePntr <= writePntr + 1;
               end
               loaded <= loaded + 1;
            end
            writeAddr <= writePntr;
         end
         if (read) begin
            if (ps == A) begin
               readPntr <= '0;
               loaded <= '0;
            end
            else begin
               if (readPntr == '1) begin // read needs to go back to beginning
                  readPntr <= '0;
               end
               else begin
                  readPntr <= readPntr + 1;
               end
```

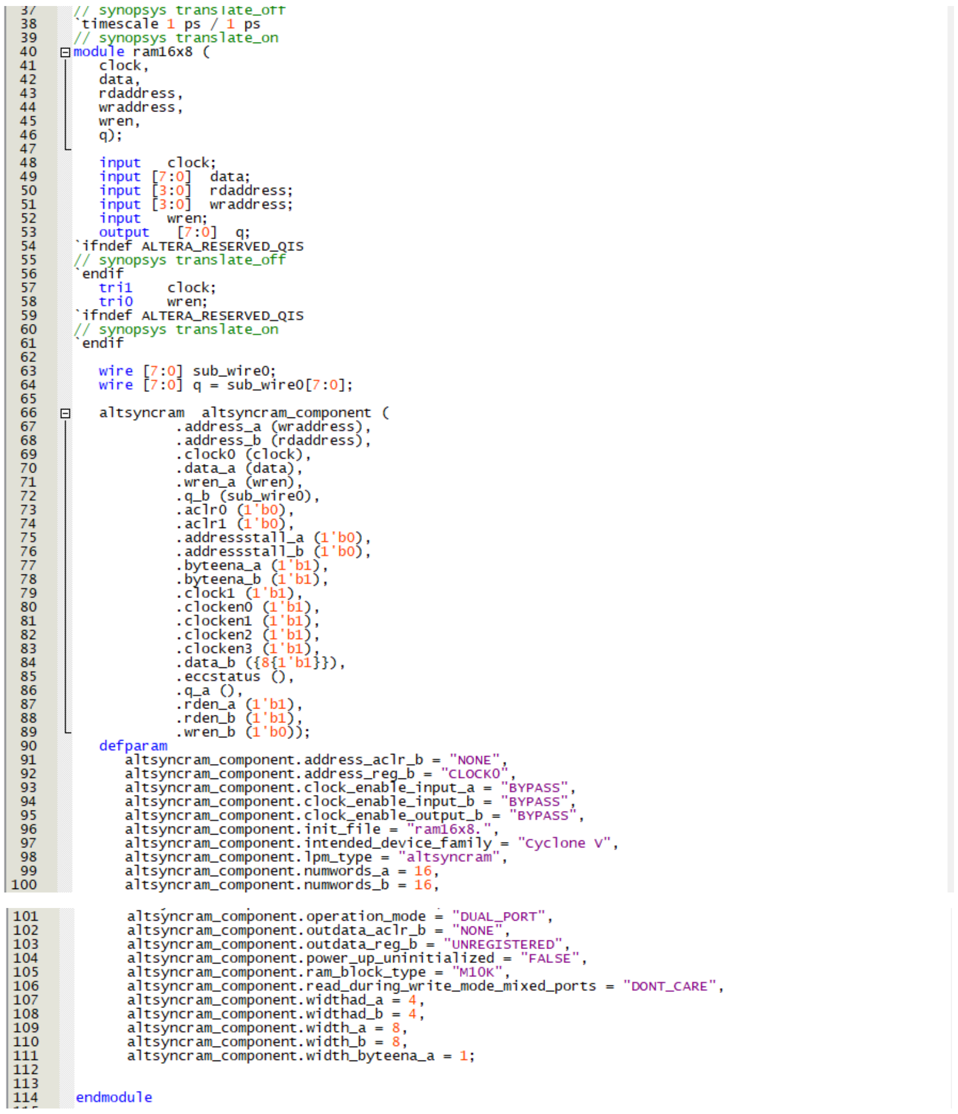
```
               end
               loaded <= loaded - 1;
            end
            readAddr <= readPntr;
         end

         empty <= ((readPntr == writePntr) && (loaded == '0));
         full <= ((readPntr == writePntr) && (loaded == 5'b01111));
         wr_en <= write && ~full;
         ps <= ns;
      end
   end

endmodule
```

Figure 17 FIFO_Control.sv



Addr	+000	+001	+010	+011	+100	+101	+110	+111	ASCII
00000	11001010	01001110	00000000	01110011	11111111	01010101	00000001	10000000	.N.s.U..
01000	11000011	11111110	00000000	10101010	11110001	00011001	01110100	01100101	.....te

Figure 18 16x8ram.mif



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

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

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

   altsyncram  altsyncram_component (
            .address_a (wraddress),
            .address_b (rdaddress),
            .clock0 (clock),
            .data_a (data),
            .wren_a (wren),
            .q_b (sub_wire0),
            .aclr0 (1'b0),
            .aclr1 (1'b0),
            .addressstall_a (1'b0),
            .addressstall_b (1'b0),
            .byteena_a (1'b1),
            .byteena_b (1'b1),
            .clock1 (1'b1),
            .clocken0 (1'b1),
            .clocken1 (1'b1),
            .clocken2 (1'b1),
            .clocken3 (1'b1),
            .data_b ({8{1'b1}}),
            .eccstatus (),
            .q_a (),
            .rden_a (1'b1),
            .rden_b (1'b1),
            .wren_b (1'b0));
   defparam
      altsyncram_component.address_aclr_b = "NONE",
      altsyncram_component.address_reg_b = "CLOCK0",
      altsyncram_component.clock_enable_input_a = "BYPASS",
      altsyncram_component.clock_enable_input_b = "BYPASS",
      altsyncram_component.clock_enable_output_b = "BYPASS",
      altsyncram_component.init_file = "ram16x8.",
      altsyncram_component.intended_device_family = "Cyclone V",
      altsyncram_component.lpm_type = "altsyncram",
      altsyncram_component.numwords_a = 16,
      altsyncram_component.numwords_b = 16,
      altsyncram_component.operation_mode = "DUAL_PORT",
      altsyncram_component.outdata_aclr_b = "NONE",
      altsyncram_component.outdata_reg_b = "UNREGISTERED",
      altsyncram_component.power_up_uninitialized = "FALSE",
      altsyncram_component.ram_block_type = "M10K",
      altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
      altsyncram_component.widthad_a = 4,
      altsyncram_component.widthad_b = 4,
      altsyncram_component.width_a = 8,
      altsyncram_component.width_b = 8,
      altsyncram_component.width_byteena_a = 1;


endmodule
```

Figure 19 16x8ram.v module