Connor Aksama
EE 371
January 19, 2023
Lab 2 Report

Procedure

This lab is comprised of three tasks. The first task involved creating a 32x4 RAM module, simulating inputs using peripherals.

Task 1

I approached this task by first studying the block diagram given in the lab specification for the task. From this schematic, I extrapolated the input and output ports for the RAM module, then began working to define the internal logic. This involved creating the internal memory array, clocking inputs, and defining write logic.

After this module was developed and tested in ModelSim, a separate top-level module was created to instantiate an instance of the previous RAM module, and to define connections between board peripherals, and HEX displays. These modules were made separate as to make the RAM module more flexible – the control lines for the RAM module are not tied to any particular type of input source.

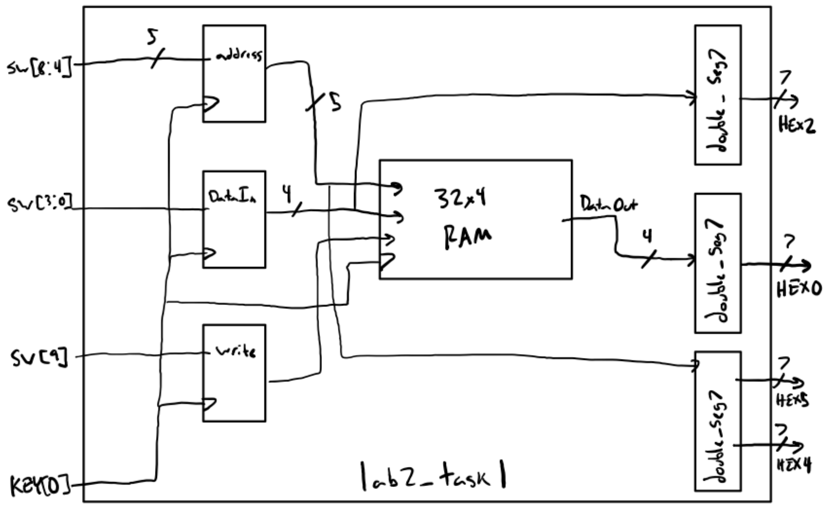

Figure 1. Top-Level Block Diagram for Lab 2 Task 1.

Task 2

I approached this task by first following the directions to create an on-chip, dual-port, 32x4 RAM module in Quartus along with a memory initialization file to provide initial values for instances of this RAM module. I then created a top level module in Verilog to instantiate this RAM module and define connections between board peripherals and displays. Additionally, I included a five-bit counter in this top-level module that would continuously increment on every cycle of a divided clock. This was done to achieve the one second, periodic cycling through of reading each memory address in the RAM module.

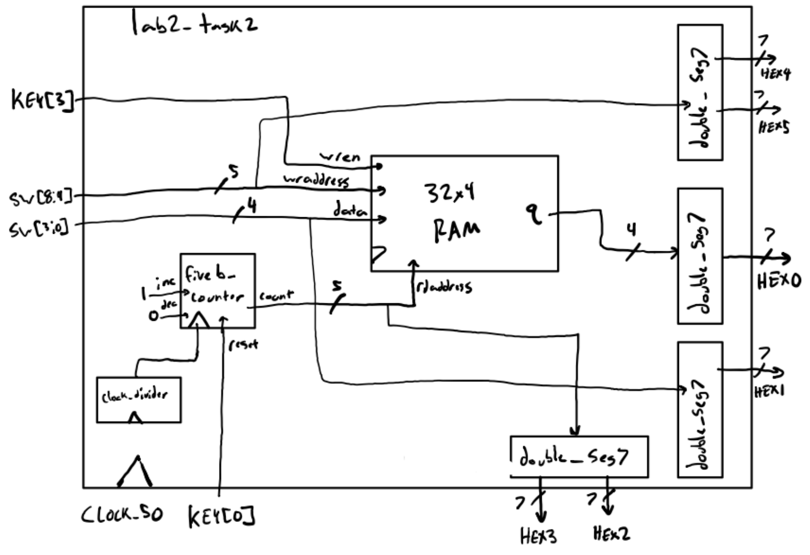

Figure 2. Top-Level Block Diagram for Lab 2 Task 2.

Task 3

I approached this task by first configuring the dual port 16x8 RAM module to use for the buffer. After this, I designed the functionality by which the FIFO modules would keep track of the read and write addresses, which would act as the front and back pointers of the queue. To this end, I implemented both of these as five-bit counters, which would each increment on a read and write operation, respectively. I also decided to include a third five-bit counter, which would keep track of the number of elements in the buffer. This was done to distinguish between whether the buffer was full or empty.

Following this, I created the top level module that would instantiate the FIFO buffer modules and defined connections between peripherals and displays. I also decided to include an extra output signal "error" which would be raised when the user tries to read while empty, or to write while full. This was done to increase the level of feedback given by the system given inputs from the user.

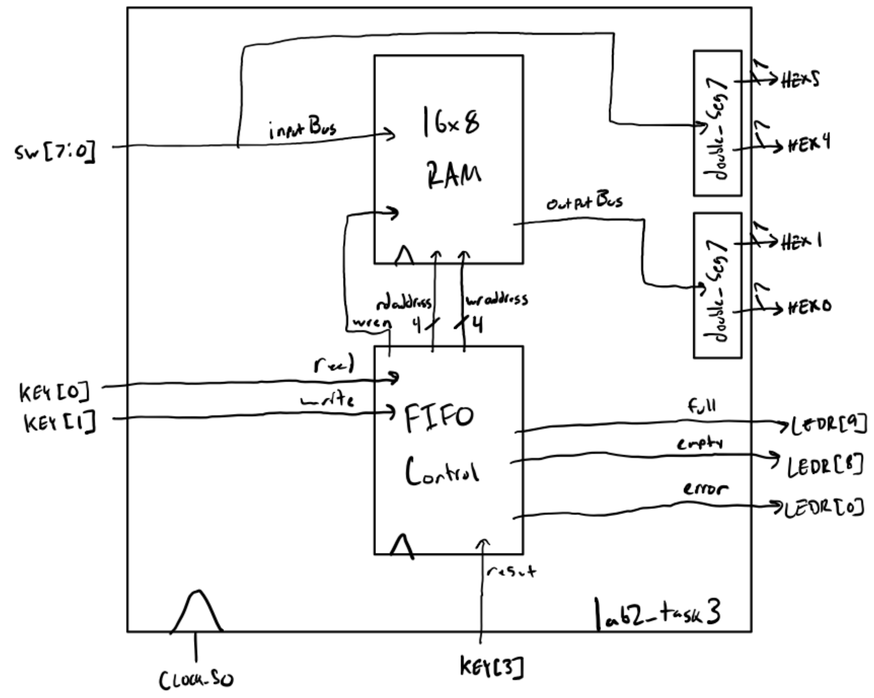

Figure 3. Top-Level Block Diagram for Lab 2 Task 3.

Results

Task 1

The following are screenshots from ModelSim simulations for each module used in the simulated RAM module.

For conciseness, HEX displays are encoded in hexadecimal. Use the following table to convert from hexadecimal code to the corresponding seven segment display:

ModelSim Hexadecimal Code	Seven Segment Display Character
40	0
79	1
24	2
30	3
19	4
12	5
02	6
78	7
00	8
10	9
08	A
03	b
27	c
21	d
06	E
0e	F

Table 1. Conversion between hexadecimal values shown in ModelSim and the character shown on a seven segment display.

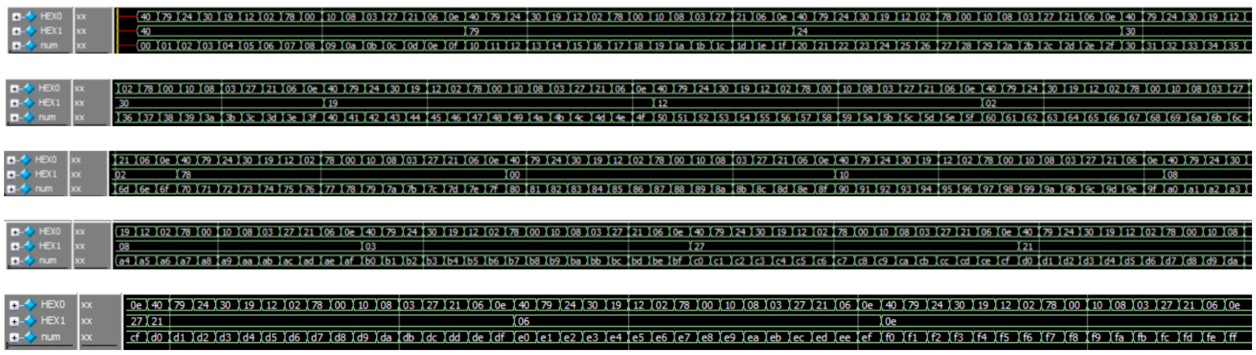

Figure 4. ModelSim Waveform for the Double Seven Segment Display module testbench.

This waveform demonstrates the two digit, hexadecimal display given any number from 0x00 to 0xff. Refer to Table 1 to decode the hexadecimal values to their corresponding characters.

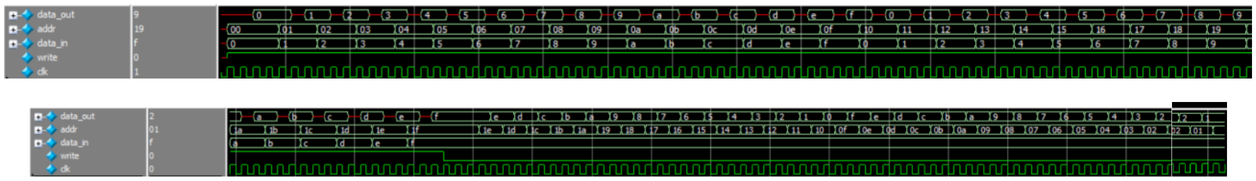

Figure 5. ModelSim Waveform for the 32x4 RAM module testbench.

These screenshots first show the process of writing data to every address in the module. Notice that it takes two clock cycles for the data on “data_in” to appear on the “data_out” line. Next, the waveform shows the process of reading from every address in the module. Notice that the value read from each address matches with the one previously written.

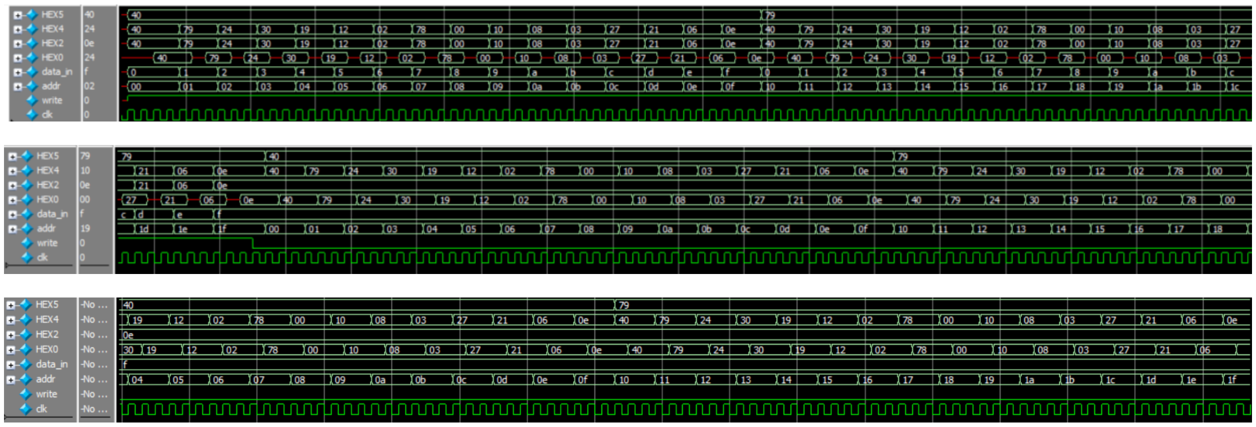

Figure 6. ModelSim Waveform for the Task 1 top-level module testbench.

Similar to Figure 5, these screenshots demonstrate the process of writing to every address in the RAM module, followed by reading from every address in the module. Additionally, the screenshots demonstrate that the correct values for the correct input and output values are shown on the HEX displays. (Refer to Table 1.)

Task 2

Task 2 uses the same Double Seven Segment Display module from Task 1. Refer to Figure 4 for waveform screenshots.

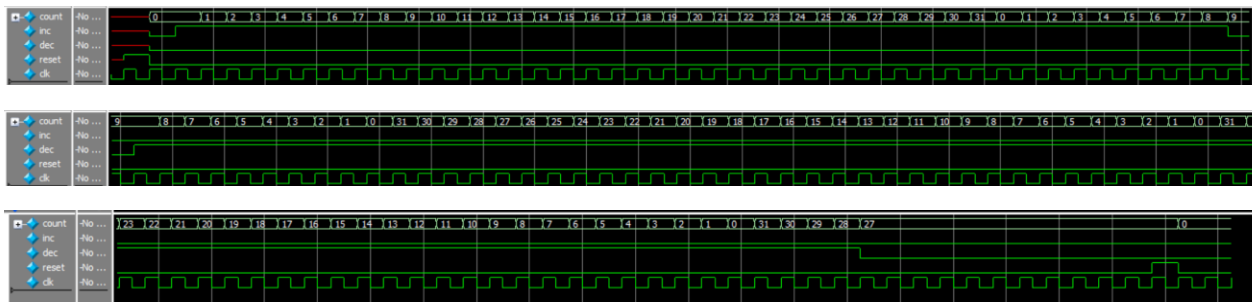

Figure 7. ModelSim Waveform for the five-bit counter module testbench.

These screenshots demonstrate incrementing the counter and overflowing, decrementing the counter and underflowing, holding the counter constant, and resetting the counter to zero.

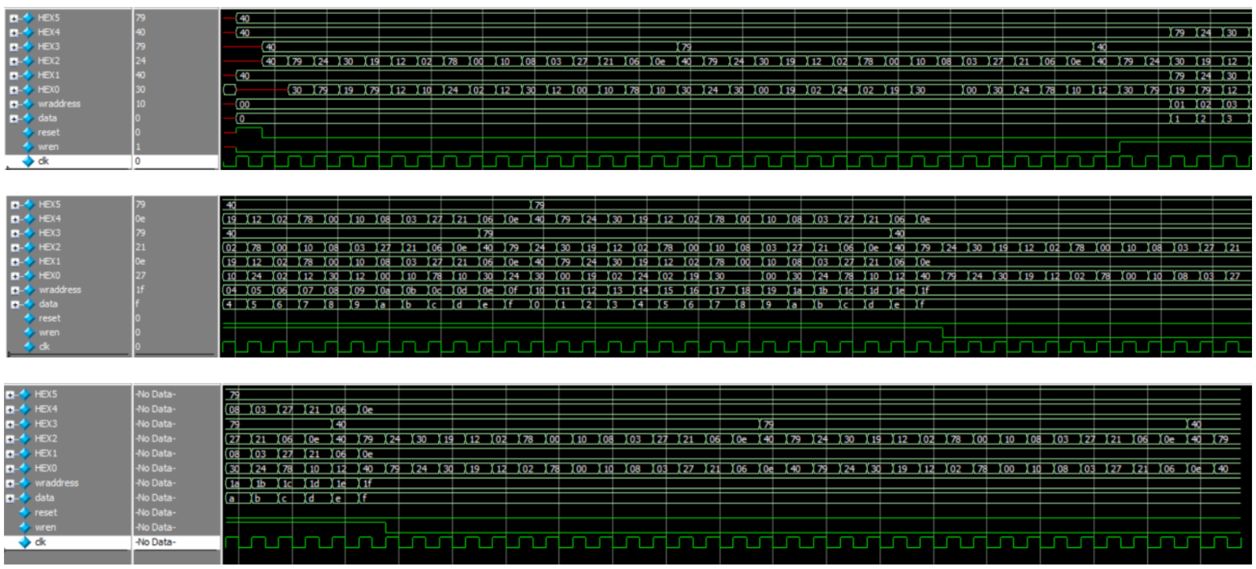

Figure 8. ModelSim Waveform for the Task 2 top-level module testbench.

These screenshots demonstrate the process of reading from every address in the RAM to ensure the correct initial values were loaded, then the process of writing a new value to every address, then reading every value to ensure the correct values were written. Note: some signals are given aliases for better clarity.

Task 3

Task 3 uses the same Double Seven Segment Display module and five-bit counter module from Task 2. Refer to Figure 4 and Figure 7 for waveform screenshots.

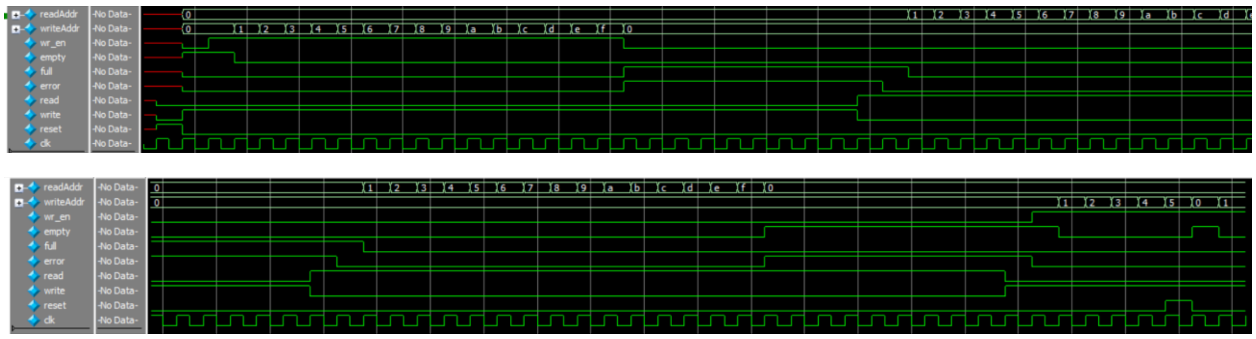

Figure 9. ModelSim Waveform for the FIFO Control module testbench.

These screenshots demonstrate the process of writing into the buffer until full, then attempting to write while the buffer is full. Then the process of reading values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.).

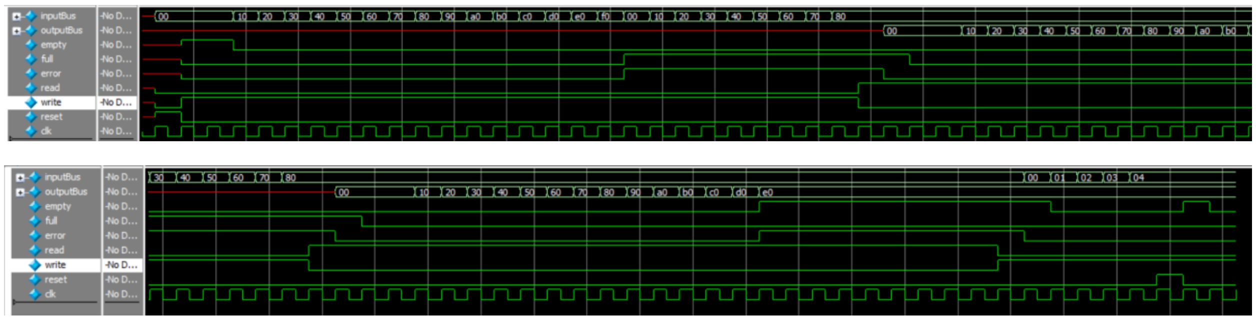

Figure 10. ModelSim Waveform for the FIFO module testbench.

These screenshots demonstrate the process of writing specific values into the buffer until full, then attempting to write while the buffer is full. Then the process of reading those same values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.).

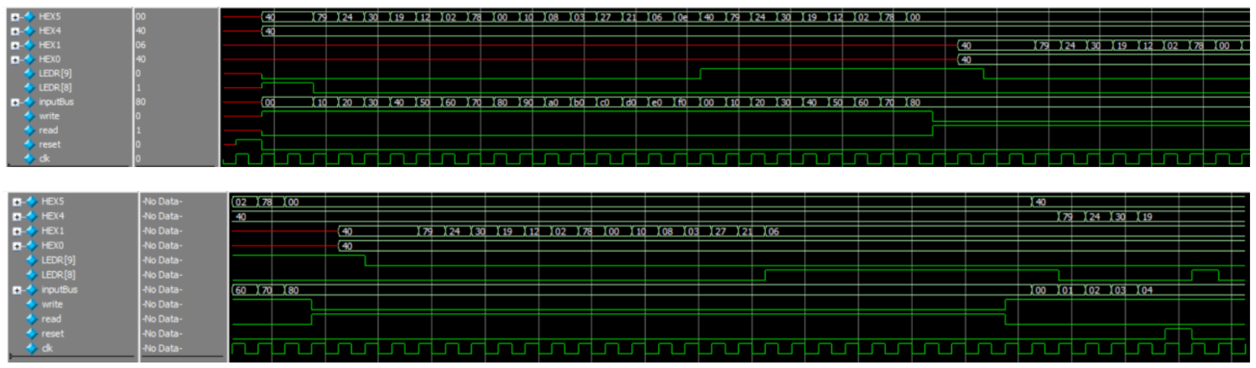

Figure 11. ModelSim Waveform for the Task 3 top-level module testbench.

These screenshots demonstrate the process of writing values into the buffer until full, then attempting to write while the buffer is full. Then the process of reading values from the buffer until empty is shown, followed by attempting to read while the buffer is empty. These screenshots additionally demonstrate that the correct output signals are raised in the correct conditions (i.e. “full” when the buffer is full, “empty” when the buffer is empty, etc.), as well as that the correct displays are shown in the correct conditions (i.e. HEX displays, LEDs, etc.). Note: some input signals are given aliases in these screenshots for better clarity of signal function.

Final Product

The overarching goal of this lab was to gain practice with interacting with RAM modules in various ways. Task 1 necessitated building the RAM module mostly from scratch while still utilizing on chip memory. This task was completed fully in line with the specification. In Task 2, I used a RAM module configured from the IP catalog, then built a lightweight system around it to cycle through its contents and be able to write to a specified address. This was also completed in line with the specification. And in Task 3, I again used a configured RAM module from the IP catalog, but this time to build a more involved FIFO buffer. The results of this task were more open ended, so there were more individual decisions made about timing constraints, edge case behavior, and overall data flow.

Appendix: SystemVerilog Code

(See next page)

```
/***********************************

double_seg7.sv

***********************************/
/*
 * Connor Aksama
 * 01/18/2023
 * CSE 371
 * Lab 2
*/

/**
 * Defines data for 2-digit hexadecimal HEX display given a 8-bit unsigned integer

 * Inputs:
 *      num [8 bit] - An unsigned integer value [0x0-0x7F] to display

 * Outputs:
 *  HEX0 [7 bit] - A HEX display for the least significant digit of the input num,
 formatted in standard seven segment display format
 *  HEX1 [7 bit] - A HEX display for the most significant digit of the input num,
 formatted in standard seven segment display format
*/
module double_seg7(
    output logic [6:0] HEX0, HEX1
    ,input logic [7:0] num
    );

    // Drive HEX output signals given num
    always_comb begin
        // Light HEX0 using LSD of num
        case (num[3:0])
            //      Light: 6543210
             0: HEX0 = ~7'b0111111;  // 0
             1: HEX0 = ~7'b0000110;  // 1
             2: HEX0 = ~7'b1011011;  // 2
             3: HEX0 = ~7'b1001111;  // 3
             4: HEX0 = ~7'b1100110;  // 4
             5: HEX0 = ~7'b1101101;  // 5
             6: HEX0 = ~7'b1111101;  // 6
             7: HEX0 = ~7'b0000111;  // 7
             8: HEX0 = ~7'b1111111;  // 8
             9: HEX0 = ~7'b1101111;  // 9
            10: HEX0 = ~7'b1110111;  // A
            11: HEX0 = ~7'b1111100;  // b
            12: HEX0 = ~7'b1011000;  // c
            13: HEX0 = ~7'b1011110;  // d
            14: HEX0 = ~7'b1111001;  // E
            15: HEX0 = ~7'b1110001;  // F
            default: HEX0 = 7'bX;
        endcase

        // Light HEX1 using MSD of num
        case (num >> 4)
            //      Light: 6543210
             0: HEX1 = ~7'b0111111;  // 0
             1: HEX1 = ~7'b0000110;  // 1
             2: HEX1 = ~7'b1011011;  // 2
             3: HEX1 = ~7'b1001111;  // 3
             4: HEX1 = ~7'b1100110;  // 4
             5: HEX1 = ~7'b1101101;  // 5
             6: HEX1 = ~7'b1111101;  // 6
             7: HEX1 = ~7'b0000111;  // 7
             8: HEX1 = ~7'b1111111;  // 8
             9: HEX1 = ~7'b1101111;  // 9
            10: HEX1 = ~7'b1110111;  // A
            11: HEX1 = ~7'b1111100;  // b
            12: HEX1 = ~7'b1011000;  // c
            13: HEX1 = ~7'b1011110;  // d
            14: HEX1 = ~7'b1111001;  // E
            15: HEX1 = ~7'b1110001;  // F
            default: HEX1 = 7'bX;
        endcase

    end

endmodule  // double_seg7

/*
 * Tests the functionality of the double_seg7 module.
*/
module double_seg7_testbench();

    logic [6:0] HEX0, HEX1;
    logic [7:0] num;

    double_seg7 dut (.HEX0, .HEX1, .num);

    initial begin

        integer i;

        // Check HEX displays for integers 0x0-0xff
        for (i = 0; i <= 255; i++) begin
            #10 num = i;
        end
        #50;
    end

endmodule  // double_seg7_testbench

/************************************

ram.sv

************************************/

/*
 * Connor Aksama
 * 01/18/2023
 * CSE 371
 * Lab 2
*/

/**
 * 32x4 RAM Module

 * Inputs:
 *  addr    [5 bit] - Address to manipulate in the given module.
 *  data_in [4 bit] - Data to store at the specified address in the module.
 *  write   [1 bit] - Signal to enable writing the given data_in to the specified addr. If raised, the value at addr is updated to data_in; o.w. the value is unchanged.
 *  clk     [1 bit] - The clock to use for this module.

 * Outputs:
 *  data_out [4 bit] - The data stored at the location specified by addr.
*/
module ram(
    output logic [3:0] data_out
    ,input logic [4:0] addr
    ,input logic [3:0] data_in
    ,input logic write, clk
    );

    // Flip-flop outputs
    logic [4:0] addr_reg;
    logic [3:0] data_in_reg;
    logic write_reg;

    // Memory cells
    logic [3:0] memory_array [31:0];

    // Use the registered address, output data at location
    assign data_out = memory_array[addr_reg];

    always_ff @(posedge clk) begin

        // Synchronize inputs
        addr_reg <= addr;
        data_in_reg <= data_in;
        write_reg <= write;

        // Write to addr if write enabled
        if (write_reg) memory_array[addr_reg] <= data_in_reg;

    end

endmodule  // ram

// Module to test the functionality of the ram module
module ram_testbench();

    logic [3:0] data_out;
    logic [4:0] addr;
    logic [3:0] data_in;
    logic write, clk;

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

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

    initial begin
        integer i;

        // Write to every address
        @(posedge clk) write <= 1; addr <= 0; data_in <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk) addr <= i; data_in <= i;
                @(posedge clk);
                @(posedge clk);
        end

        // Read from every address
        @(posedge clk) write <= 0;
        for (i = 31; i >= 0; i--) begin
            @(posedge clk) addr <= i;
                @(posedge clk);
        end
            $stop;
    end

endmodule  // ram_testbench

/***********************************

lab2_task1.sv

***********************************/

/*
 * Connor Aksama
 * 01/18/2023
 * CSE 371
 * Lab 2
*/

/**
 * Top-level module for Lab 2 Task 1. Instantiates a 32x4 RAM module and connects I/O to
 board peripherals

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
 display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment
 display format.
 *  HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment
 display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
 display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
 display format.
*/
module lab2_task1(
    output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
    ,input logic [9:0] SW
    ,input logic [3:0] KEY
    );

    // Turn off HEX3 and HEX1
    assign HEX3 = ~7'b0;
    assign HEX1 = ~7'b0;

    // Use KEY0 as system clock
    logic SYS_CLOCK;
    assign SYS_CLOCK = KEY[0];

    logic [3:0] data_in, data_out;
    logic [4:0] addr;
    logic write;

    assign data_in = SW[3:0];  // SW3-0 specifies input data
    assign addr = SW[8:4];     // SW8-4 specifies adddress
    assign write = SW[9];      // SW9 is write signal

    // Connect RAM input ports to SW inputs
    // Store RAM output in local logic
    ram r (.data_out, .addr, .data_in, .write(write), .clk(SYS_CLOCK));

    // Display address on HEX4-5
    double_seg7 addr_display (.HEX0(HEX4), .HEX1(HEX5), .num({3'b0, addr}));

    // Display data_in on HEX2
    double_seg7 data_in_display (.HEX0(HEX2), .HEX1(), .num({4'b0, data_in}));

    // Display data_out on HEX0
    double_seg7 data_out_display(.HEX0(HEX0), .HEX1(), .num({4'b0, data_out}));

endmodule

// Module to test the functionality of the lab2_task1 module
module lab2_task1_testbench();

    logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
    logic [9:0] SW;
    logic [3:0] KEY;
    lab2_task1 dut (.HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0, .SW, .KEY);

    logic write, clk;
     logic [3:0] data_in;
     logic [4:0] addr;
    assign KEY[0] = clk;
    assign SW[9] = write;
    assign SW[3:0] = data_in;
    assign SW[8:4] = addr;

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

    initial begin
        integer i;

        // Write to every address in the RAM
        @(posedge clk) write <= 1; addr <= 0; data_in <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk) addr <= i; data_in <= i;
                @(posedge clk);
                @(posedge clk);
        end

        // Read from every address in the RAM
        @(posedge clk) write <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk) addr <= i;
                @(posedge clk);
                @(posedge clk);
        end

          $stop;
    end

endmodule

/************************************

fiveb_counter.sv

************************************/

/*
 * Connor Aksama
 * 01/19/2023
 * CSE 371
 * Lab 2
*/

/**
 * A five-bit increment/decrement counter. Overflows to 5'b00000; underflows to 5'11111.
 * reset takes precedence; if inc and dec are simultaneously raised, count is unchanged.
 * Inputs:
 *  inc [1 bit] - Increments the count by 1 when raised. Does nothing if signal is low.
 *  dec [1 bit] - Decrements the count by 1 when raised. Does nothing if signal is low.
 *  clk [1 bit] - The clock to use for this module.
 *  reset [1 bit] - Resets the counter to 0 when raised. Does nothing if signal is low.

 * Outputs:
 *      count [5 bit] - The count of this counter.
 */
module fiveb_counter(
    output logic [4:0] count
    ,input logic inc, dec, clk, reset
    );

    // Handles increment/decrement/reset operations
    always_ff @(posedge clk) begin

        if (reset)
            count <= 5'b0;
        else if (inc & ~dec)
            count <= count + 5'b1;
        else if (dec & ~inc)
            count <= count - 5'b1;

    end

endmodule  // fiveb_counter

/*
 * Tests the functionality of the fiveb_counter module.
*/
module fiveb_counter_testbench();

    logic [4:0] count;
    logic inc, dec, clk, reset;

    fiveb_counter dut (.count, .inc, .dec, .clk, .reset);

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

    initial begin
        integer i;

        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0; inc <= 0; dec <= 0;

        // Increment counter from 0 to max, then past max
        @(posedge clk) inc <= 1;
        for (i = 0; i < 40; i++) begin
            @(posedge clk);
        end

        // Hold counter constant
        @(posedge clk) inc <= 0;
        for (i = 0; i < 10; i++) begin
            @(posedge clk);
        end

        // Decrement counter below 0, then max to 0, etc.
        @(posedge clk) dec <= 1;
        for (i = 0; i < 45; i++) begin
            @(posedge clk);
        end

        // Hold counter constant
        @(posedge clk) dec <= 0;
        for (i = 0; i < 10; i++) begin
            @(posedge clk);
        end

        // reset to 0
        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0;

        @(posedge clk)
        @(posedge clk)

        $stop;
    end

endmodule  // fiveb_counter_testbench

/***********************************

lab2_task2.sv

***********************************/

/*
 * Connor Aksama
 * 01/19/2023
 * CSE 371
 * Lab 2
*/

/**
 * Top-level module for Lab 2 Task 2. Instantiates a 32x4 RAM module and connects I/O to board peripherals

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment display format.
 *  HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment display format.
 *  HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment display format.
*/
module lab2_task2 #(
    parameter which_clock = 25
    )
    (
    output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
    ,input logic [9:0] SW
    ,input logic [3:0] KEY
    ,input logic CLOCK_50
    );

    // Clock divider to cycle through reading RAM addresses
    // Divides the 50 MHz clock into an array of divided clocks (clk)
    logic [31:0] clk;
    clock_divider clk_div (.clock(CLOCK_50), .divided_clocks(clk));

    // Aliases for peripherals and RAM output
    logic   [3:0] data;  // Data_In
    logic   [4:0] rdaddress;  // Read Address
    logic   [4:0] wraddress;  // Write Address
    logic wren;  // Write Enable
    logic [3:0] q;  // RAM Data_Out
    logic reset;

    // Assign peripherals to aliases
    assign reset = ~KEY[0];
    assign wren = ~KEY[3];
    assign wraddress = SW[8:4];
    assign data = SW[3:0];

    // 32x4 RAM Module
    // Inputs: 50 MHz sys. clock, Data_In, Read Address, Write Address, Write Enable
    // Outputs: q -> RAM Data_Out
    ram32x4 ram (.clock(CLOCK_50), .data, .rdaddress, .wraddress, .wren, .q);

    // HEX Display for RAM Data_Out
    double_seg7 q_display (.HEX0(HEX0), .HEX1(), .num(q));

    // HEX Display for Read Address
    double_seg7 rdaddress_display (.HEX0(HEX2), .HEX1(HEX3), .num(rdaddress));

    // HEX Display for Write Address
    double_seg7 wraddress_display (.HEX0(HEX4), .HEX1(HEX5), .num(wraddress));

    // HEX Display for Data_In
    double_seg7 data_display (.HEX0(HEX1), .HEX1(), .num(data));

    // Counter to cycle through RAM addresses, runs on divided clock
    // Output count to Read Address
    fiveb_counter counter (.count(rdaddress), .inc(1), .dec(0), .clk(clk[which_clock]), .
    reset);

endmodule  // lab2_task2

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

  initial begin
    divided_clocks = '0;
  end

  always_ff @(posedge clock) begin
    divided_clocks <= divided_clocks + 'd1;
  end

endmodule  // clock_divider

/**
 * Identical to lab2_task2 module, without clock division. Used for testing purposes.

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
 display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX2 [7 bit] - Data to show on the HEX2 display, formatted in standard 7 segment
 display format.
 *  HEX3 [7 bit] - Data to show on the HEX3 display, formatted in standard 7 segment
 display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
 display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
 display format.
*/
`timescale 1 ps / 1 ps
module lab2_task2_testmodule
    (
    output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
    ,input logic [9:0] SW
    ,input logic [3:0] KEY
    ,input logic CLOCK_50
    );

    // Aliases for peripherals and RAM output
    logic   [3:0] data;  // Data_In
    logic   [4:0] rdaddress;  // Read Address
    logic   [4:0] wraddress;  // Write Address
    logic wren;  // Write Enable
    logic [3:0] q;  // RAM Data_Out
    logic reset;

    // Assign peripherals to aliases
    assign reset = ~KEY[0];
    assign wren = ~KEY[3];
    assign wraddress = SW[8:4];
    assign data = SW[3:0];

    // 32x4 RAM Module
    // Inputs: 50 MHz sys. clock, Data_In, Read Address, Write Address, Write Enable
    // Outputs: q -> RAM Data_Out
    ram32x4 ram (.clock(CLOCK_50), .data, .rdaddress, .wraddress, .wren, .q);

    // HEX Display for RAM Data_Out
    double_seg7 q_display (.HEX0(HEX0), .HEX1(), .num(q));

    // HEX Display for Read Address
    double_seg7 rdaddress_display (.HEX0(HEX2), .HEX1(HEX3), .num(rdaddress));

    // HEX Display for Write Address
    double_seg7 wraddress_display (.HEX0(HEX4), .HEX1(HEX5), .num(wraddress));

    // HEX Display for Data_In
    double_seg7 data_display (.HEX0(HEX1), .HEX1(), .num(data));

    // Counter to cycle through RAM addresses, runs on sys clock
    // Output count to Read Address
    fiveb_counter counter (.count(rdaddress), .inc(1), .dec(0), .clk(CLOCK_50), .reset);

endmodule  // lab2_task2_testmodule

// Module to test the functionality of the lab2_task2 module
module lab2_task2_testbench();

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

    logic [4:0] wraddress;
    logic [3:0] data;
    logic reset, wren;

    assign SW[8:4] = wraddress;
    assign SW[3:0] = data;
    assign KEY[0] = ~reset;
    assign KEY[3] = ~wren;

    lab2_task2_testmodule dut (.HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0, .SW, .KEY, .
    CLOCK_50(clk));

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

    initial begin
        integer i;

        // Read every address
        @(posedge clk) reset <= 1; wraddress <= 0; data <= 0; wren <= 0;
        @(posedge clk) reset <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk);
        end

        // Write to every address
        @(posedge clk) wren <= 1; wraddress <= 0; data <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk) wraddress <= i; data <= i;
        end

        // Read every address
        @(posedge clk) wren <= 0;
        for (i = 0; i < 32; i++) begin
            @(posedge clk);
        end
        $stop;
    end

endmodule  // lab2_task2_testbench

/***********************************

FIFO.sv

***********************************/

/*
 * Connor Aksama
 * 01/19/2023
 * CSE 371
 * Lab 2
*/

/**
 * FIFO Module to handle logic between the FIFO Controller and the physical RAM.

 * Inputs:
 *  inputBus [8 (width) bit] - Data to enqueue in the buffer.
 *  read     [1 bit] - Signal to enable reading an element from the buffer. If raised and buffer is not empty, the least recent word is output on the outputBus, o.w. the outputBus is unchanged.
 *  write    [1 bit] - Signal to enable writing an element to the buffer. If raised and buffer is not full, the word on the inputBus is enqueued, o.w. the buffer is unchanged.
 *  reset    [1 bit] - Resets the FIFO buffer - removes all elements from the buffer.
 *  clk      [1 bit] - The clock to use for this module.

 * Outputs:
 *  outputBus [8 (width) bit] - The data most recently dequeued from the buffer.
 *  empty     [1 bit] - This signal is raised when there are no elements in the buffer, and lowered otherwise.
 *  full      [1 bit] - This signal is raised when the buffer is at capacity, and lowered otherwise.
 *  error     [1 bit] - This signal is raised when reading while the buffer is empty, or when writing while the buffer is full
*/
module FIFO #(
                  parameter depth = 4,
                  parameter width = 8
                  )(
                    output logic empty, full, error
                    ,output logic [width-1:0] outputBus
                    ,input logic clk, reset
                    ,input logic read, write
                    ,input logic [width-1:0] inputBus
                   );

    logic [3:0] rdaddress;  // Read Address from Control module
    logic [3:0] wraddress;  // Write Address from Control module
    logic wren;  // Write enable from Control Module
    logic [width-1:0] outputBus_temp;  // Temporary read output from RAM

    // 16x8 RAM module instantiation
    // Inputs: sys clock, Data_In from user, Read Address, Write Address, Write Enable
    // Outputs: Data at Read Address -> outputBus_temp
    ram16x8 ram (.clock(clk), .data(inputBus), .rdaddress, .wraddress, .wren, .q(
    outputBus_temp));


    // FIFO Control Module
    // Inputs: sys clock, read signal from user, write from user
    // Outputs: Write Enable, Empty/Full/Error, Read Address, Write Address
    FIFO_Control #(depth) FC (.clk, .reset,
                                  .read,
                                  .write,
                                  .wr_en(wren),
                                  .empty,
                                  .full,
                                  .error,
                                  .readAddr(rdaddress),
                                  .writeAddr(wraddress)
                                 );

    always_ff @(posedge clk) begin
        // Change outputBus only if valid read is raised
        if (read & ~empty) outputBus <= outputBus_temp;
        else          outputBus <= outputBus;
    end

endmodule  // FIFO

// Module to test the functionality of 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, error;
    logic [width-1:0] outputBus;

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

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

    initial begin
        integer i;

        @(posedge clk) reset <= 1; write <= 0; read <= 0; inputBus <= 0;

        // Write to queue until full, try to write while full
        @(posedge clk) write <= 1; reset <= 0;
        for (i = 0; i < 25; i++) begin
            @(posedge clk) inputBus <= (i << 4);
        end

        // Read from queue until empty, try to read while empty
        @(posedge clk) write <= 0; read <= 1;
        for (i = 24; i >= 0; i--) begin
```

```systemverilog
            @(posedge clk);
        end

        // Write some elements
        @(posedge clk) write <= 1; read <= 0;
        for (i = 0; i < 5; i++) begin
            @(posedge clk) inputBus <= i;
        end

        // Reset
        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0;
        @(posedge clk);
        @(posedge clk);

        $stop;
    end

endmodule  // FIFO_testbench

/***********************************

FIFO_Control.sv

***********************************/

/*
 * Connor Aksama
 * 01/19/2023
 * CSE 371
 * Lab 2
*/

/**
 * FIFO Control Module to handle logic of what and when to read/write from the RAM given user inputs.

 * Inputs:
 *  read  [1 bit] - Signal to enable reading an element from the buffer. If raised and buffer is not empty, the next address to read is output on readAddr
 *  write [1 bit] - Signal to enable writing an element to the buffer. If raised and buffer is not full, the next address to write is output on writeAddr
 *  reset [1 bit] - Resets the FIFO buffer - removes all elements from the buffer.
 *  clk   [1 bit] - The clock to use for this module.

 * Outputs:
 *  readAddr  [4 (depth) bit] - The next address to read from in the RAM.
 *  writeAddr [4 (depth) bit] - The next address to write to in the RAM.
 *  empty     [1 bit] - This signal is raised when there are no elements in the buffer, and lowered otherwise.
 *  full      [1 bit] - This signal is raised when the buffer is at capacity, and lowered otherwise.
 *  error     [1 bit] - This signal is raised when reading while the buffer is empty, or when writing while the buffer is full
 *  wr_en     [1 bit] - This signal is raised when data should be written to the location specified by writeAddr, and lowered otherwise.
*/
module FIFO_Control #(
                             parameter depth = 4
                             )(
                                output  logic wr_en
                                ,output logic empty, full, error
                                ,output logic [depth-1:0] readAddr, writeAddr
                                ,input  logic clk, reset
                                ,input  logic read, write

                              );

    logic [4:0] fifo_size;  // Number of elements in the buffer
    logic read_reg, write_reg;  // Registered read/write inputs
    logic dead;  // Temporary variable for five bit counter output

    // Counter to store current Read Address in RAM
    // Inputs: Increment signal -> read detected and not empty
    // Outputs: Count -> Read Address
    fiveb_counter read_ptr  (.count({dead, readAddr}), .inc(read_reg & ~empty), .dec(0),
    .clk, .reset);

    // Counter to store current Write Address in RAM
    // Inputs: Increment signal -> write detected and not full
    // Outputs: Count -> Write Address
    fiveb_counter write_ptr (.count({dead, writeAddr}), .inc(write_reg & ~full), .dec(0),
     .clk, .reset);

    // Counter to store number of elements in buffer
    // Inputs: Increment signal -> write detected and not full; Decrement signal -> read
    detected and not empty
    // Outputs: Count -> FIFO Size
    fiveb_counter num_elts (.count(fifo_size), .inc(write_reg & ~full), .dec(read_reg & ~
    empty), .clk, .reset);

    // Compare against FIFO size
    assign empty = fifo_size == '0;
    assign full  = fifo_size == (depth ** 2);
    // Check for invalid read/writes
    assign error = (read_reg & empty) | (write_reg & full);

    // Enable write if write detected and not full
    assign wr_en = write_reg & ~full;

    always_ff @(posedge clk) begin
        // Register read/write inputs
        read_reg <= read;
        write_reg <= write;
    end

endmodule  // FIFO_Control

// Module to test the functionality of the FIFO_Control module
`timescale 1 ps / 1 ps
module FIFO_Control_testbench();

    parameter depth = 4;

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

    FIFO_Control #(depth) dut (.*);

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

    initial begin
        integer i;

        @(posedge clk) reset <= 1; write <= 0; read <= 0;

        // Write until full, try writing while full
        @(posedge clk) write <= 1; reset <= 0;
        for (i = 0; i < 25; i++) begin
            @(posedge clk);
        end

        // Read until empty, try reading while empty
        @(posedge clk) read <= 1; write <= 0;
        for (i = 0; i < 25; i++) begin
            @(posedge clk);
        end

        // Write some elements
        @(posedge clk) write <= 1; read <= 0;
        for (i = 0; i < 5; i++) begin
            @(posedge clk);
        end

        // Reset
        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0;
        @(posedge clk);
        @(posedge clk);

        $stop;
    end

endmodule  // FIFO_Control_testbench

/***********************************

lab2_task3.sv

***********************************/

/*
 * Connor Aksama
 * 01/19/2023
 * CSE 371
 * Lab 2
*/

/**
 * Top-level module for Lab 2 Task 3. Instantiates a 16x8 RAM module, FIFO Controller,
 and connects I/O to board peripherals

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
 display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
 display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
 display format.
*/
module lab2_task3  #(
    parameter depth = 4
    ,parameter width = 8
    )
    (
    output logic [6:0] HEX5, HEX4, HEX1, HEX0
    ,output logic [9:0] LEDR
    ,input logic [9:0] SW
    ,input logic [3:0] KEY
    ,input logic CLOCK_50
    );

    logic [width-1:0] output_data;

    // HEX display for FIFO output
    double_seg7 output_data_display (.HEX0(HEX0), .HEX1(HEX1), .num(output_data));

    // HEX display for input data
    double_seg7 input_data_display (.HEX0(HEX4), .HEX1(HEX5), .num(SW[7:0]));

    // FIFO Module
    // Inputs: read/write from KEYs, input data from SW[7:0]
    // Outputs: output element line from outputBus; empty/full/error signals -> LEDRs
    FIFO #(depth, width) fifo  (
                                    .empty(LEDR[8])
                                    , .full(LEDR[9])
                                    , .error(LEDR[0])
                                    , .outputBus(output_data)
                                    , .clk(CLOCK_50)
                                    , .reset(~KEY[3])
                                    , .read(~KEY[0])
                                    , .write(~KEY[1])
                                    , .inputBus(SW[7:0])
                                    );

endmodule  // lab3_task3

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

  initial begin
    divided_clocks = '0;
  end

  always_ff @(posedge clock) begin
    divided_clocks <= divided_clocks + 'd1;
  end

endmodule  // clock_divider

// Module to test the functionality of the lab2_task3 module
`timescale 1 ps / 1 ps
module lab2_task3_testbench();
    parameter depth = 4;
    parameter width = 8;

    logic [6:0] HEX5, HEX4, HEX1, HEX0;
    logic [9:0] LEDR;
    logic [9:0] SW;
    logic [3:0] KEY;
    logic clk;

    logic write, read, reset;
    logic [width-1:0] inputBus;

    assign KEY[1] = ~write;
    assign KEY[0] = ~read;
    assign KEY[3] = ~reset;
    assign SW[7:0] = inputBus;

    lab2_task3 #(depth, width) dut (.HEX5, .HEX4, .HEX1, .HEX0, .LEDR, .SW, .KEY, .
    CLOCK_50(clk));

    parameter CLOCK_PERIOD = 100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD / 2) clk <= ~clk;
    end
```

```verilog

    initial begin
        integer i;

        @(posedge clk) reset <= 1;

        // Write to queue until full, try to write while full
        @(posedge clk) write <= 1; read <= 0; reset <= 0; inputBus <= 0;
        for (i = 0; i < 25; i++) begin
            @(posedge clk) inputBus <= (i << 4);
        end

        // Read from queue until empty, try to read while empty
        @(posedge clk) write <= 0; read <= 1;
        for (i = 24; i >= 0; i--) begin
            @(posedge clk);
        end

        // Write some elements
        @(posedge clk) write <= 1; read <= 0;
        for (i = 0; i < 5; i++) begin
            @(posedge clk) inputBus <= i;
        end

        // Reset
        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0;
        @(posedge clk);
        @(posedge clk);

        $stop;
    end

endmodule  // lab2_task3_testbench
```