CSUN | COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

Fall 2024

California State University, Northridge

Department of Electrical & Computer Engineering

Lab 10: Synchronous FIFO

I hereby attest that this lab report is entirely my own work. I have not copied either code or text from anyone, nor have I allowed or will I allow anyone to copy my work.

Name (printed) Zachary Martin

Name (signed) [illegible] Date 12/14/2024

ECE 526

Written By: Zach Martin

December 14, 2024

I. Introduction

A FIFO buffer is a form of memory storage that processes the oldest data first. As an example, if ‘A’, ‘B’, and ‘C’ were written into memory positions 0x1, 0x2, and 0x3 sequentially, the first read operation would process memory position 0x1 and return ‘A’, and the second read operation would process memory position 0x2 and return ‘B’. FIFO as an organization method is often seen outside of computer engineering. Grocery stores feature FIFO as a method of stocking shelves (ensuring that customers retrieve the soonest-to-expire goods first), serving customers in line, and scanning conveyor belts of items that customers have selected to purchase. This process is illustrated in figure 1 below.

The aim of this experiment is to design a First-In-First-Out (FIFO) data buffer in SystemVerilog. A FIFO module will be designed to accept as inputs a variable-width data field, clock signal, write-enable, read-enable, and active-low reset signal, and to return as outputs an equally wide data output, current-depth counter, and a series of status flags. These inputs and outputs can be observed in the high-level block diagram shown in figure 2. The module and its status flags will be tested thoroughly using a test fixture module to verify FIFO functionality.

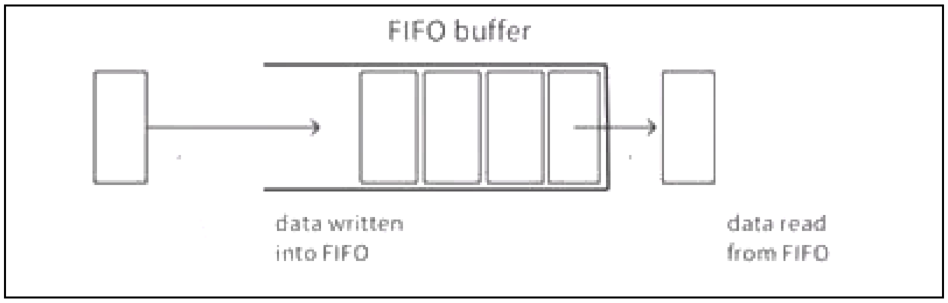


Fig. 1. Example FIFO buffer emphasizing read/write order.

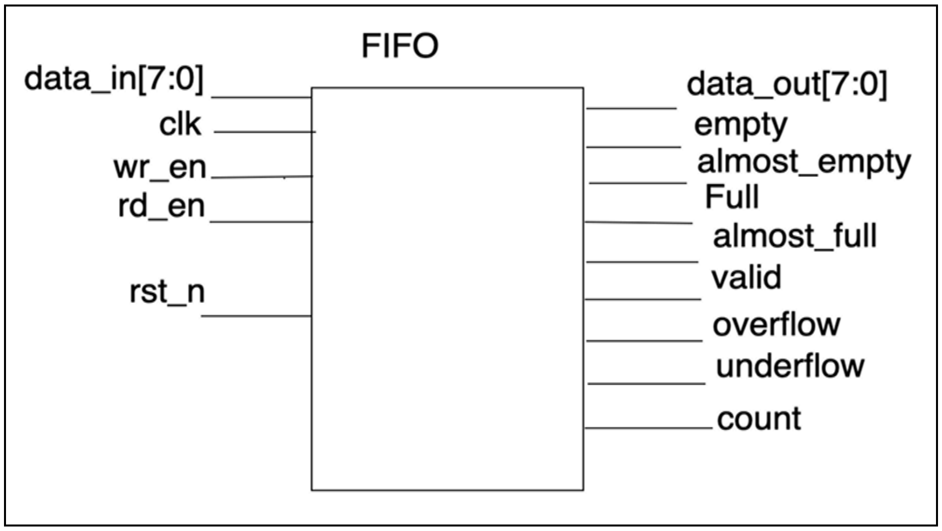


Fig. 2. High level FIFO module block diagram.

II. PROCEDURE

A. *Module Design*

The FIFO.sv file can be partitioned into three sections containing the following:

1. Module, parameter, and internal variable declarations.
2. Continuous flag and counter assignments.
3. A procedural block containing read, write, and reset functionality.

In the first section, the module was created and all inputs and outputs were declared using ANSI style inside of the port declaration list. Parameters were defined for the width of each memory location, the depth of the FIFO, and the size of the near-full and near-empty status flags. Table 1 contains names, widths, and descriptions of all inputs, outputs, and parameters. A two dimensional array named ‘fifo’ held all memory data. Pointers to the current write position and read position were created according to the parameterized FIFO depth. The first section of the module can be seen in figure 3.

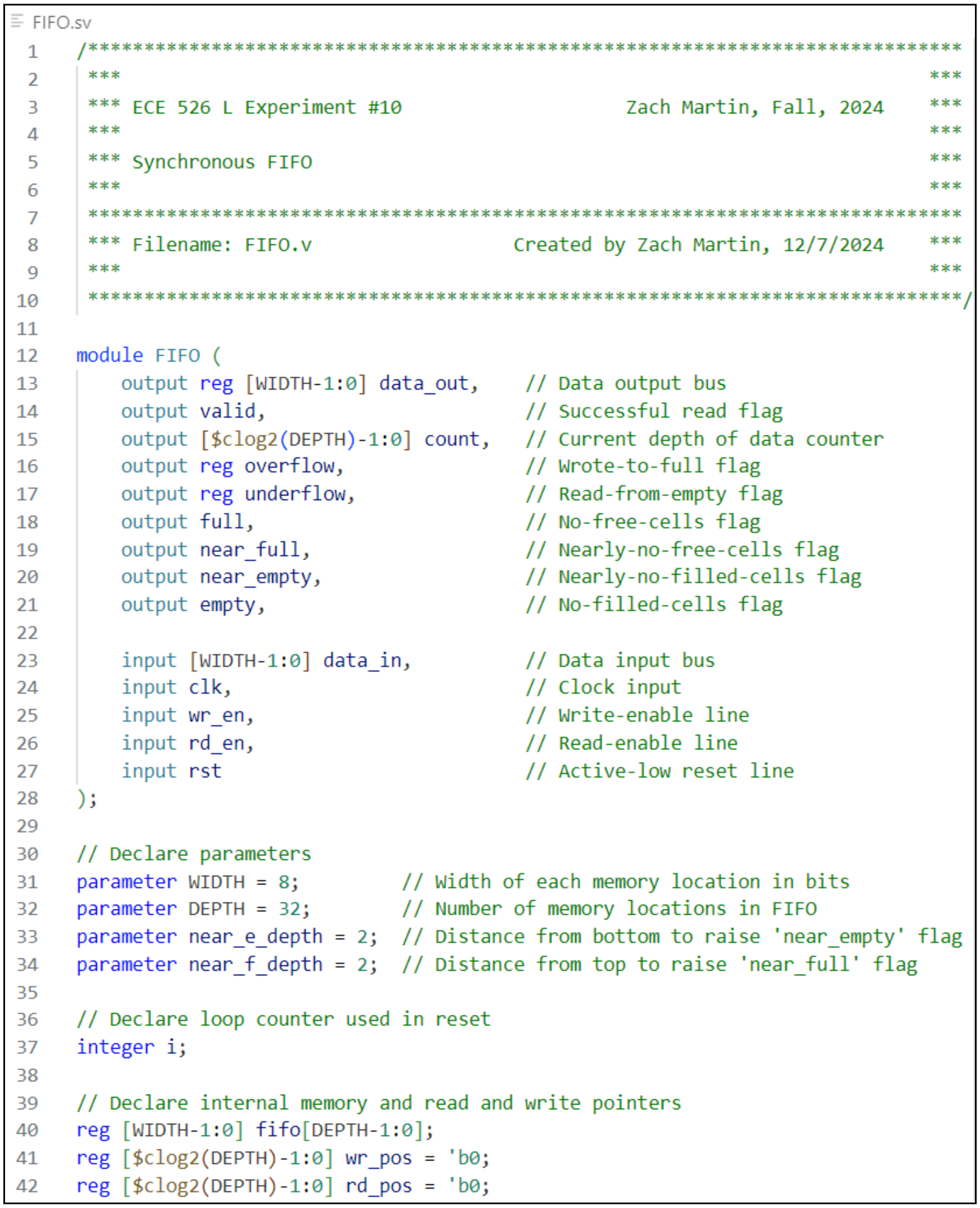
In the second section, the depth of the FIFO counter was assigned to the difference between the write and read position pointers, and all status flags except overflow and underflow were continuously assigned. The second section of the module can be seen in figure 4.

In the third section, an always procedural block sensitive to the positive edge of the clock signal and the negative edge of the reset signal contained the reset, write, and read operations. If reset was asserted, the read and write position pointers, memory values, data output bus, and overflow and underflow flags were cleared. Otherwise, if the write enable signal was asserted and the buffer was not full, the memory location pointed to by the write position pointer was updated with the value on the data input bus, underflow was deasserted, and the write position pointer was incremented. If the write enable signal was asserted but the buffer was full, the overflow flag was asserted. If the read enable signal was asserted and the buffer was not empty, the value held in the memory location pointed to by the read position pointer was passed to the data output, the overflow flag was deasserted, and the read position pointer was incremented. If the read enable signal was asserted but the buffer was empty, the underflow flag was asserted. The third section of the module can be seen in figure 5. The entire FIFO.sv file can be seen in the appendix section of this report.

TABLE I

FIFO BUFFER VARIABLES

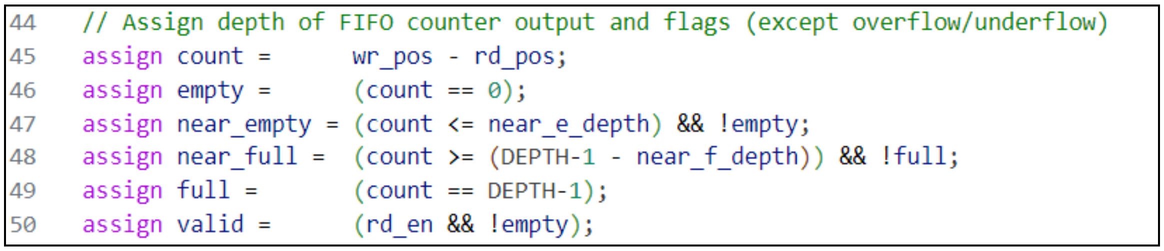
Name	Type	Size [bits]	Description
WIDTH	parameter	8 (default)	Width of each memory location
DEPTH	parameter	32 (default)	Number of memory locations in FIFO
near_e_depth	parameter	2 (default)	Distance from bottom to raise ‘near_empty’ flag
near_f_depth	parameter	2 (default)	Distance from top to raise ‘near full’ flag
data_in	input	WIDTH	Data input bus
clk	input	1	Clock signal
wr_en	input	1	Active-high write-enable signal
rd_en	input	1	Active-high read-enable signal
rst	input	1	Active-low reset signal
data_out	output	WIDTH	Data output bus
valid	output	1	Flag activated by successful read operation
count	output	$\log_2$(DEPTH)	Current FIFO depth counter
overflow	output	1	Flag activated by write ops. when FIFO is full
underflow	output	1	Flag activated by read ops. when FIFO is empty
full	output	1	Flag activated when FIFO is full
near_full	output	1	Flag activated when FIFO is nearly full
near_empty	output	1	Flag activated when FIFO is nearly empty
empty	output	1	Flag activated when FIFO is empty



FIFO.sv

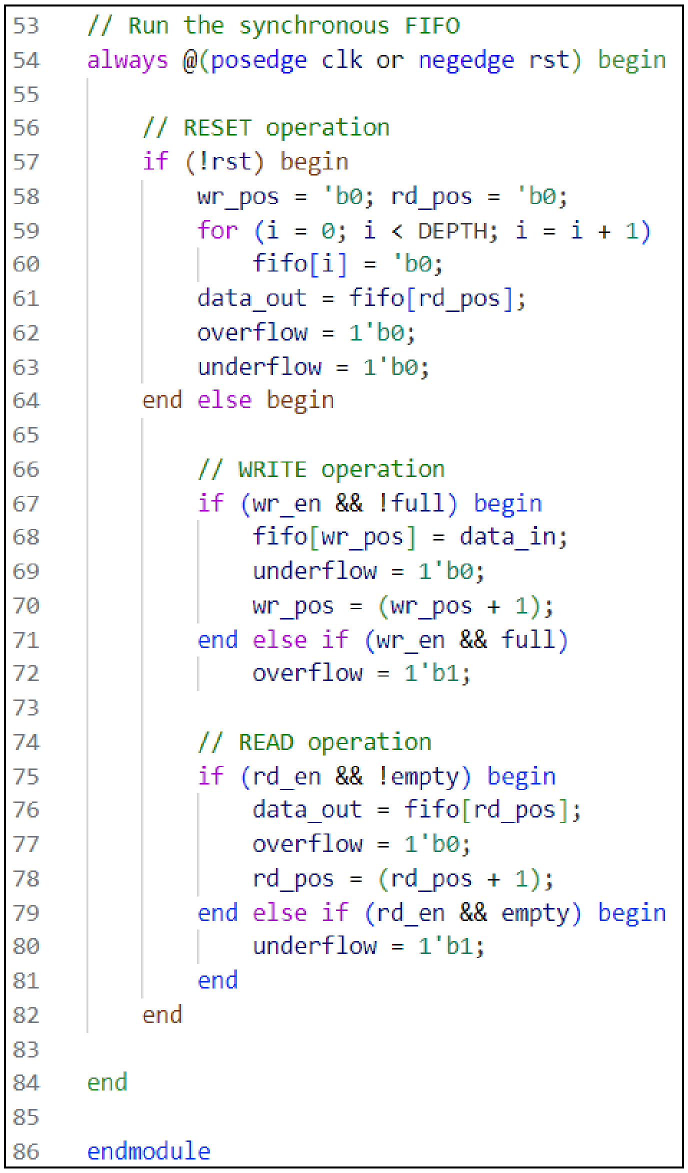
```
/***********************************************************************
 ***                                                                ***
 *** ECE 526 L Experiment #10                 Zach Martin, Fall, 2024   ***
 ***                                                                ***
 *** Synchronous FIFO                                               ***
 ***                                                                ***
 ***********************************************************************
 *** Filename: FIFO.v                 Created by Zach Martin, 12/7/2024   ***
 ***                                                                ***
 ***********************************************************************/

module FIFO (
    output reg [WIDTH-1:0] data_out,    // Data output bus
    output valid,                       // Successful read flag
    output [$clog2(DEPTH)-1:0] count,   // Current depth of data counter
    output reg overflow,                // Wrote-to-full flag
    output reg underflow,               // Read-from-empty flag
    output full,                        // No-free-cells flag
    output near_full,                   // Nearly-no-free-cells flag
    output near_empty,                  // Nearly-no-filled-cells flag
    output empty,                       // No-filled-cells flag

    input [WIDTH-1:0] data_in,          // Data input bus
    input clk,                          // Clock input
    input wr_en,                        // Write-enable line
    input rd_en,                        // Read-enable line
    input rst                           // Active-low reset line
);

// Declare parameters
parameter WIDTH = 8;          // Width of each memory location in bits
parameter DEPTH = 32;         // Number of memory locations in FIFO
parameter near_e_depth = 2;   // Distance from bottom to raise 'near_empty' flag
parameter near_f_depth = 2;   // Distance from top to raise 'near_full' flag

// Declare loop counter used in reset
integer i;

// Declare internal memory and read and write pointers
reg [WIDTH-1:0] fifo[DEPTH-1:0];
reg [$clog2(DEPTH)-1:0] wr_pos = 'b0;
reg [$clog2(DEPTH)-1:0] rd_pos = 'b0;
```

Fig. 3. FIFO.sv module, parameter, and internal variable declarations.



```
// Assign depth of FIFO counter output and flags (except overflow/underflow)
assign count =      wr_pos - rd_pos;
assign empty =      (count == 0);
assign near_empty = (count <= near_e_depth) && !empty;
assign near_full =  (count >= (DEPTH-1 - near_f_depth)) && !full;
assign full =       (count == DEPTH-1);
assign valid =      (rd_en && !empty);
```

Fig. 4. FIFO.sv continuous flag and counter assignments.



```
// Run the synchronous FIFO
always @(posedge clk or negedge rst) begin

    // RESET operation
    if (!rst) begin
        wr_pos = 'b0; rd_pos = 'b0;
        for (i = 0; i < DEPTH; i = i + 1)
            fifo[i] = 'b0;
        data_out = fifo[rd_pos];
        overflow = 1'b0;
        underflow = 1'b0;
    end else begin

        // WRITE operation
        if (wr_en && !full) begin
            fifo[wr_pos] = data_in;
            underflow = 1'b0;
            wr_pos = (wr_pos + 1);
        end else if (wr_en && full)
            overflow = 1'b1;

        // READ operation
        if (rd_en && !empty) begin
            data_out = fifo[rd_pos];
            overflow = 1'b0;
            rd_pos = (rd_pos + 1);
        end else if (rd_en && empty) begin
            underflow = 1'b1;
        end
    end

end

endmodule
```

Fig. 5. FIFO.sv procedural block.

B. *Test Fixture Design*

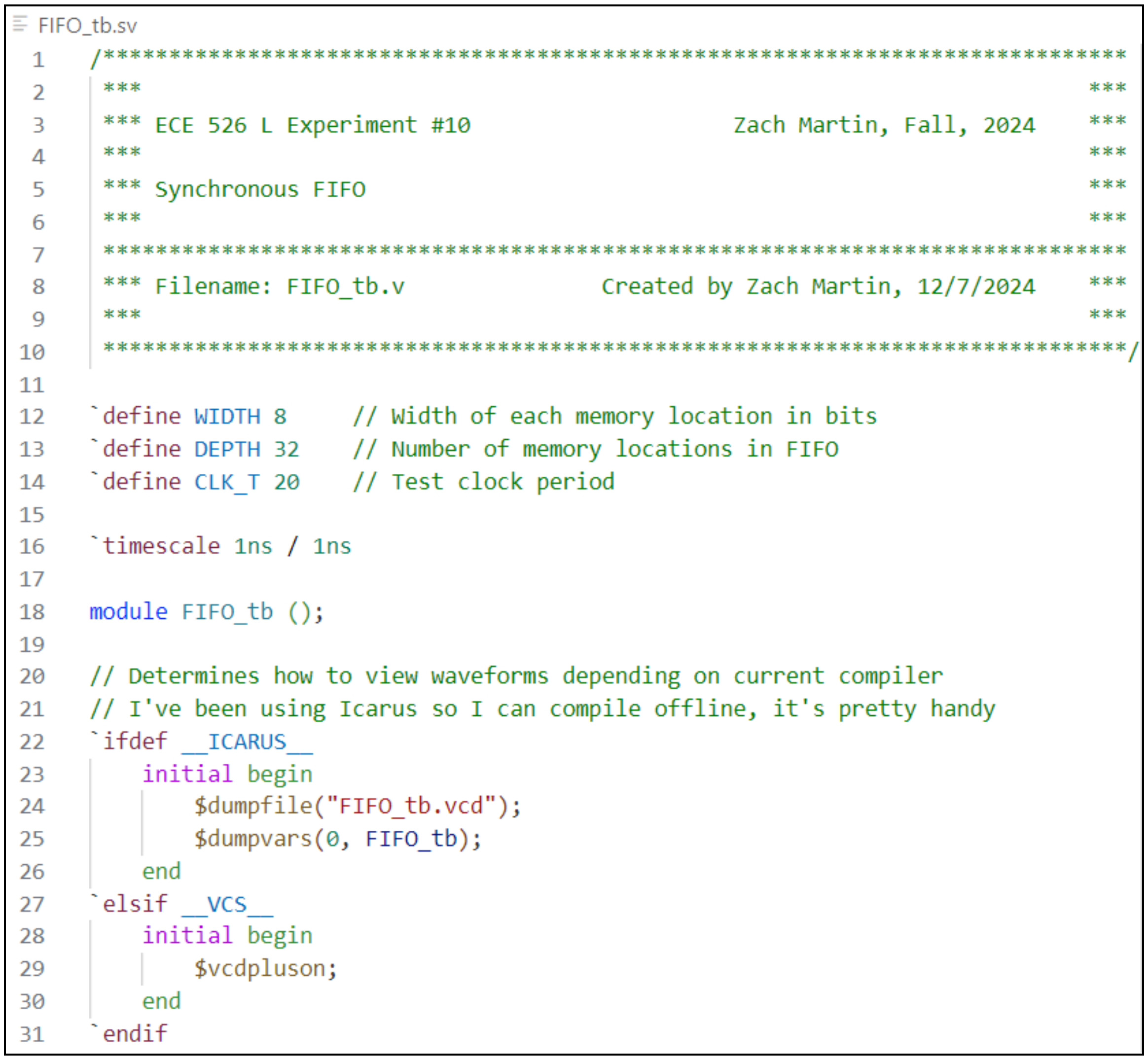
The FIFO_tb.sv test fixture file contained three sections as follows:

1. A file header, preprocessor directives and compiler-specific instructions.
2. Declarations, instantiations, and monitoring statements.
3. A procedural block containing test vectors.

In the first section, memory width, FIFO depth, and test clock period values were assigned using preprocessor directives. The module timescale was set to 1ns with a resolution of 1 ns. Inside the module, the ifdef preprocessor directive was used to identify the compiler being used to compile the program and enable waveform analysis in accordance with each compiler. This was implemented so that the programs could be analyzed using Icarus Verilog, an offline compiler which was used throughout all stages of development, but remained compatible with the MobaXTerm software used in previous labs. This section of the test fixture can be seen in figure 6.

In the second section, a FIFO module was created and connected to previously declared registers and wires by name. Additionally, an initial procedural block was written to enable signal monitoring using the $monitor system task. This section can be seen in figure 7.

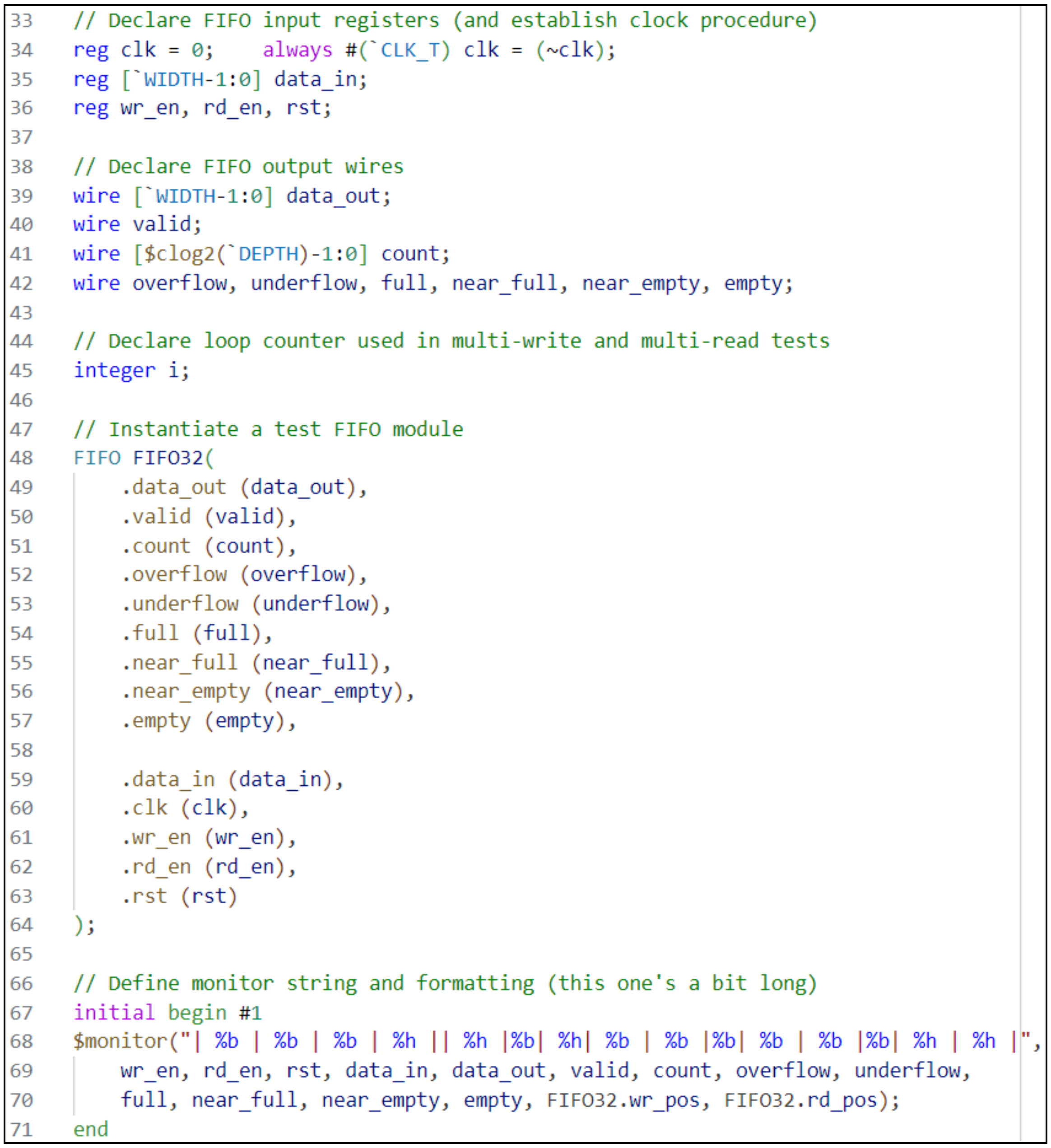
The procedural block in the third section began by pulsing the reset input to ensure that no initial values were indeterminate. Then a for loop was used to iteratively write to each memory location in the FIFO buffer, and to attempt to write to the buffer once full. Without resetting the FIFO, another for loop was used to iteratively read from each memory location, and to attempt to read from the buffer once empty. The FIFO was then reset and a simultaneous read/write test was written. A series of five write operations were performed, after which the read-enable input was activated. Three more write operations were performed with the read-enable input active to demonstrate that the FIFO was capable of writing to and reading from distinct memory locations simultaneously. The procedural block can be seen in figure 8.



FIFO_tb.sv

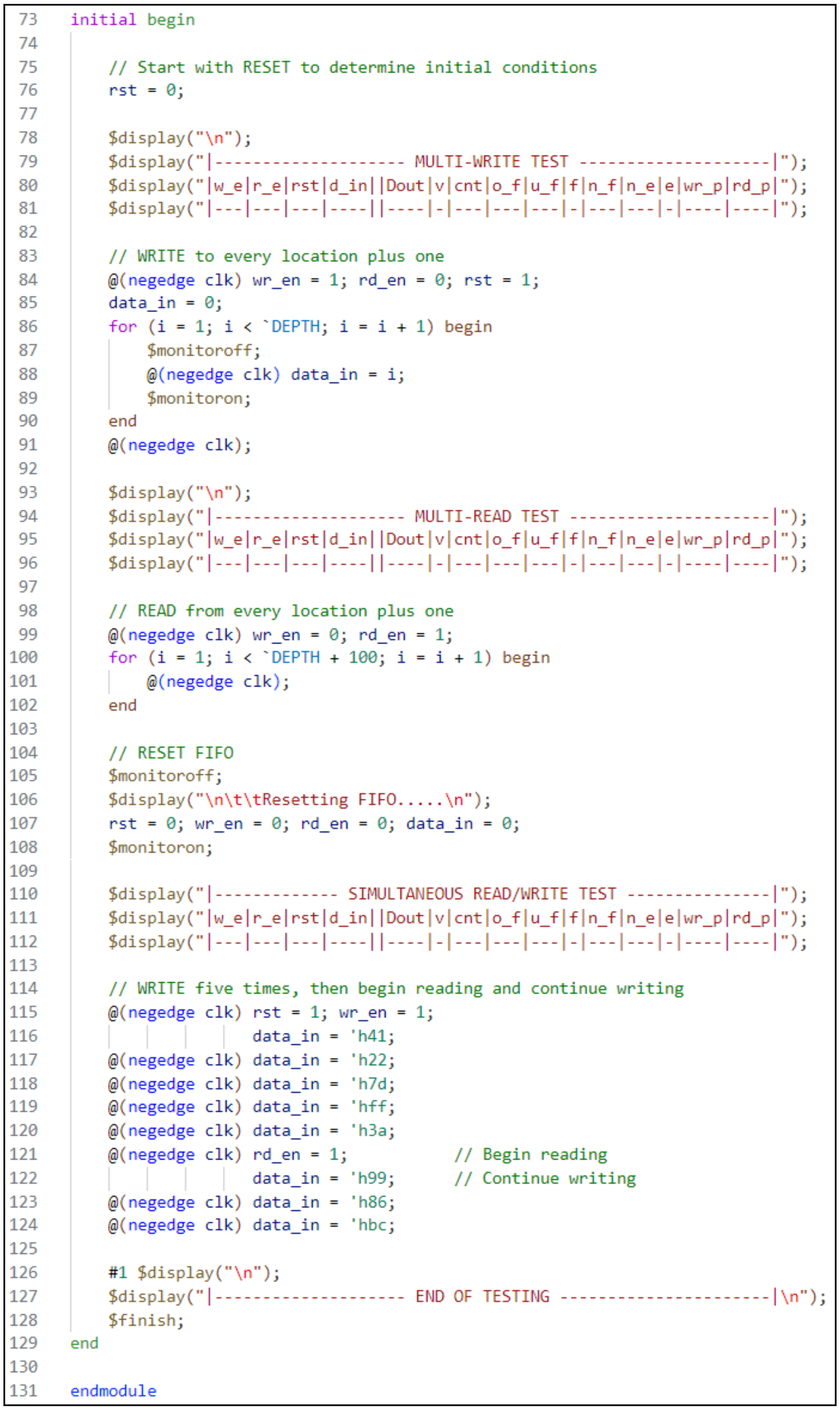
```
/*****************************************************************************
 ***                                                                       ***
 *** ECE 526 L Experiment #10                    Zach Martin, Fall, 2024   ***
 ***                                                                       ***
 *** Synchronous FIFO                                                      ***
 ***                                                                       ***
 *****************************************************************************
 *** Filename: FIFO_tb.v               Created by Zach Martin, 12/7/2024   ***
 ***                                                                       ***
 *****************************************************************************/

`define WIDTH 8     // Width of each memory location in bits
`define DEPTH 32    // Number of memory locations in FIFO
`define CLK_T 20    // Test clock period

`timescale 1ns / 1ns

module FIFO_tb ();

// Determines how to view waveforms depending on current compiler
// I've been using Icarus so I can compile offline, it's pretty handy
`ifdef __ICARUS__
    initial begin
        $dumpfile("FIFO_tb.vcd");
        $dumpvars(0, FIFO_tb);
    end
`elsif __VCS__
    initial begin
        $vcdpluson;
    end
`endif
```

Fig. 6. FIFO_tb.sv file header, preprocessor directives and compiler-specific instructions.



```
// Declare FIFO input registers (and establish clock procedure)
reg clk = 0;    always #(`CLK_T) clk = (~clk);
reg [`WIDTH-1:0] data_in;
reg wr_en, rd_en, rst;

// Declare FIFO output wires
wire [`WIDTH-1:0] data_out;
wire valid;
wire [$clog2(`DEPTH)-1:0] count;
wire overflow, underflow, full, near_full, near_empty, empty;

// Declare loop counter used in multi-write and multi-read tests
integer i;

// Instantiate a test FIFO module
FIFO FIFO32(
    .data_out (data_out),
    .valid (valid),
    .count (count),
    .overflow (overflow),
    .underflow (underflow),
    .full (full),
    .near_full (near_full),
    .near_empty (near_empty),
    .empty (empty),

    .data_in (data_in),
    .clk (clk),
    .wr_en (wr_en),
    .rd_en (rd_en),
    .rst (rst)
);

// Define monitor string and formatting (this one's a bit long)
initial begin #1
$monitor("| %b | %b | %b | %h || %h |%b| %h| %b | %b |%b| %b | %b |%b| %h | %h |",
    wr_en, rd_en, rst, data_in, data_out, valid, count, overflow, underflow,
    full, near_full, near_empty, empty, FIFO32.wr_pos, FIFO32.rd_pos);
end
```

Fig. 7. FIFO_tb.sv declarations, instantiations, and monitoring statements.



```
initial begin

    // Start with RESET to determine initial conditions
    rst = 0;

    $display("\n");
    $display("|-------------------- MULTI-WRITE TEST --------------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // WRITE to every location plus one
    @(negedge clk) wr_en = 1; rd_en = 0; rst = 1;
    data_in = 0;
    for (i = 1; i < `DEPTH; i = i + 1) begin
        $monitoroff;
        @(negedge clk) data_in = i;
        $monitoron;
    end
    @(negedge clk);

    $display("\n");
    $display("|-------------------- MULTI-READ TEST ---------------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // READ from every location plus one
    @(negedge clk) wr_en = 0; rd_en = 1;
    for (i = 1; i < `DEPTH + 100; i = i + 1) begin
        @(negedge clk);
    end

    // RESET FIFO
    $monitoroff;
    $display("\n\t\tResetting FIFO.....\n");
    rst = 0; wr_en = 0; rd_en = 0; data_in = 0;
    $monitoron;

    $display("|------------- SIMULTANEOUS READ/WRITE TEST ---------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // WRITE five times, then begin reading and continue writing
    @(negedge clk) rst = 1; wr_en = 1;
                   data_in = 'h41;
    @(negedge clk) data_in = 'h22;
    @(negedge clk) data_in = 'h7d;
    @(negedge clk) data_in = 'hff;
    @(negedge clk) data_in = 'h3a;
    @(negedge clk) rd_en = 1;          // Begin reading
                   data_in = 'h99;     // Continue writing
    @(negedge clk) data_in = 'h86;
    @(negedge clk) data_in = 'hbc;

    #1 $display("\n");
    $display("|-------------------- END OF TESTING ---------------------|\n");
    $finish;
end

endmodule
```

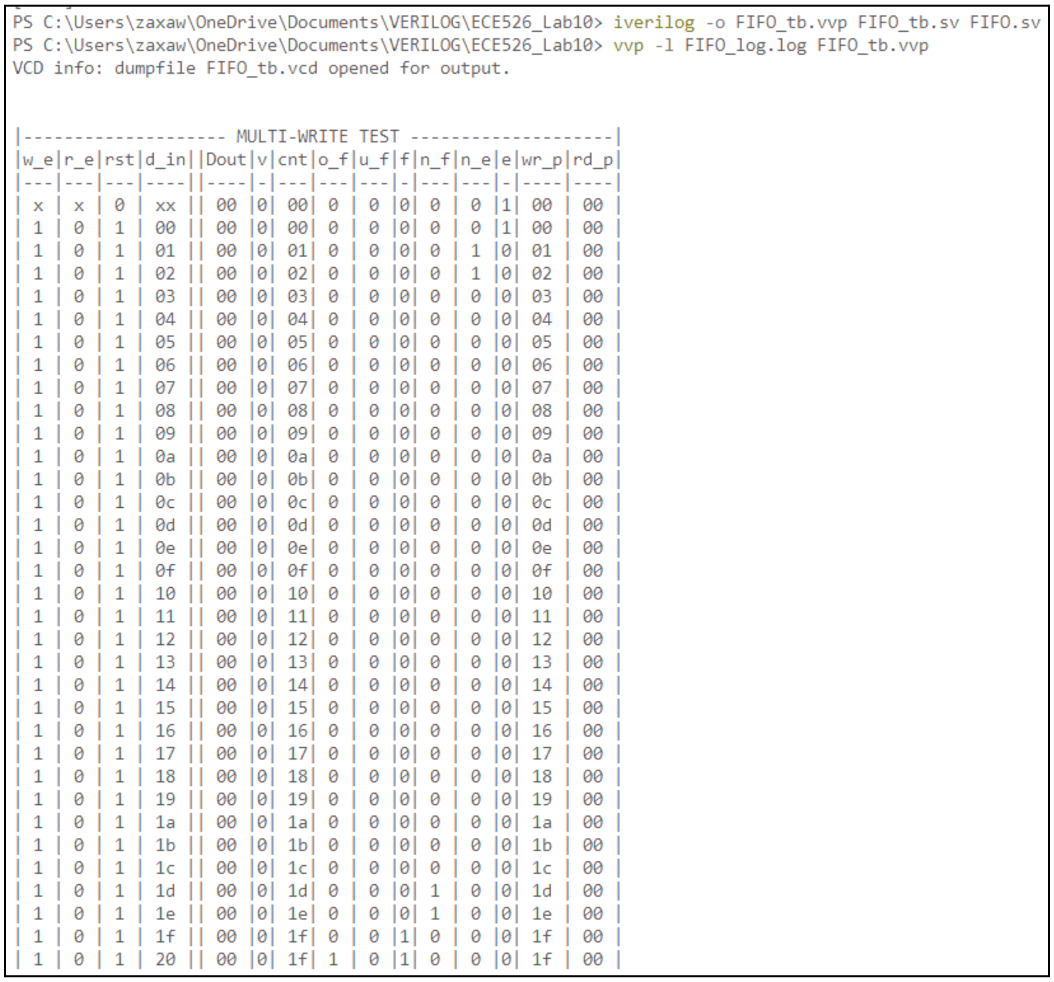
Fig. 8. FIFO_tb.sv procedural block containing test vectors.

III. RESULTS

The output of the test fixture displayed three sections of results as follows:

1. Multiple-write test results.
2. Multiple-read test results.
3. Simultaneous read/write results.

In the multiple-write test, every memory address was written to with the hexadecimal value of the write position pointer. This cannot be directly viewed due to the nature of the $monitor system task, but the value can be seen on the data input bus (d_in) and will be shown to be true in the read test results. Before the first write operation, the empty flag (e) can be observed to be asserted. For the next two write operations, the empty flag was deasserted and the near-empty flag (n_e) was asserted. Upon performing the 30th write operation, the near-full flag (n_f) was asserted, and upon performing the 32nd write operation, the near-full flag was deasserted and the full flag (f) was asserted. When trying to perform a 33rd write operation, the overflow flag (o_f) was asserted. The Icarus Verilog command line output of the multiple-write test can be seen in figure 9. GTKWave simulation waveforms of the test can be seen in figure 10.



```
PS C:\Users\zaxaw\OneDrive\Documents\VERILOG\ECE526_Lab10> iverilog -o FIFO_tb.vvp FIFO_tb.sv FIFO.sv
PS C:\Users\zaxaw\OneDrive\Documents\VERILOG\ECE526_Lab10> vvp -l FIFO_log.log FIFO_tb.vvp
VCD info: dumpfile FIFO_tb.vcd opened for output.


|------------------- MULTI-WRITE TEST -------------------|
|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|
|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|
| x | x | 0 | xx || 00 |0| 00| 0 | 0 |0| 0 | 0 |1| 00 | 00 |
| 1 | 0 | 1 | 00 || 00 |0| 00| 0 | 0 |0| 0 | 0 |1| 00 | 00 |
| 1 | 0 | 1 | 01 || 00 |0| 01| 0 | 0 |0| 0 | 1 |0| 01 | 00 |
| 1 | 0 | 1 | 02 || 00 |0| 02| 0 | 0 |0| 0 | 1 |0| 02 | 00 |
| 1 | 0 | 1 | 03 || 00 |0| 03| 0 | 0 |0| 0 | 0 |0| 03 | 00 |
| 1 | 0 | 1 | 04 || 00 |0| 04| 0 | 0 |0| 0 | 0 |0| 04 | 00 |
| 1 | 0 | 1 | 05 || 00 |0| 05| 0 | 0 |0| 0 | 0 |0| 05 | 00 |
| 1 | 0 | 1 | 06 || 00 |0| 06| 0 | 0 |0| 0 | 0 |0| 06 | 00 |
| 1 | 0 | 1 | 07 || 00 |0| 07| 0 | 0 |0| 0 | 0 |0| 07 | 00 |
| 1 | 0 | 1 | 08 || 00 |0| 08| 0 | 0 |0| 0 | 0 |0| 08 | 00 |
| 1 | 0 | 1 | 09 || 00 |0| 09| 0 | 0 |0| 0 | 0 |0| 09 | 00 |
| 1 | 0 | 1 | 0a || 00 |0| 0a| 0 | 0 |0| 0 | 0 |0| 0a | 00 |
| 1 | 0 | 1 | 0b || 00 |0| 0b| 0 | 0 |0| 0 | 0 |0| 0b | 00 |
| 1 | 0 | 1 | 0c || 00 |0| 0c| 0 | 0 |0| 0 | 0 |0| 0c | 00 |
| 1 | 0 | 1 | 0d || 00 |0| 0d| 0 | 0 |0| 0 | 0 |0| 0d | 00 |
| 1 | 0 | 1 | 0e || 00 |0| 0e| 0 | 0 |0| 0 | 0 |0| 0e | 00 |
| 1 | 0 | 1 | 0f || 00 |0| 0f| 0 | 0 |0| 0 | 0 |0| 0f | 00 |
| 1 | 0 | 1 | 10 || 00 |0| 10| 0 | 0 |0| 0 | 0 |0| 10 | 00 |
| 1 | 0 | 1 | 11 || 00 |0| 11| 0 | 0 |0| 0 | 0 |0| 11 | 00 |
| 1 | 0 | 1 | 12 || 00 |0| 12| 0 | 0 |0| 0 | 0 |0| 12 | 00 |
| 1 | 0 | 1 | 13 || 00 |0| 13| 0 | 0 |0| 0 | 0 |0| 13 | 00 |
| 1 | 0 | 1 | 14 || 00 |0| 14| 0 | 0 |0| 0 | 0 |0| 14 | 00 |
| 1 | 0 | 1 | 15 || 00 |0| 15| 0 | 0 |0| 0 | 0 |0| 15 | 00 |
| 1 | 0 | 1 | 16 || 00 |0| 16| 0 | 0 |0| 0 | 0 |0| 16 | 00 |
| 1 | 0 | 1 | 17 || 00 |0| 17| 0 | 0 |0| 0 | 0 |0| 17 | 00 |
| 1 | 0 | 1 | 18 || 00 |0| 18| 0 | 0 |0| 0 | 0 |0| 18 | 00 |
| 1 | 0 | 1 | 19 || 00 |0| 19| 0 | 0 |0| 0 | 0 |0| 19 | 00 |
| 1 | 0 | 1 | 1a || 00 |0| 1a| 0 | 0 |0| 0 | 0 |0| 1a | 00 |
| 1 | 0 | 1 | 1b || 00 |0| 1b| 0 | 0 |0| 0 | 0 |0| 1b | 00 |
| 1 | 0 | 1 | 1c || 00 |0| 1c| 0 | 0 |0| 0 | 0 |0| 1c | 00 |
| 1 | 0 | 1 | 1d || 00 |0| 1d| 0 | 0 |0| 1 | 0 |0| 1d | 00 |
| 1 | 0 | 1 | 1e || 00 |0| 1e| 0 | 0 |0| 1 | 0 |0| 1e | 00 |
| 1 | 0 | 1 | 1f || 00 |0| 1f| 0 | 0 |1| 0 | 0 |0| 1f | 00 |
| 1 | 0 | 1 | 20 || 00 |0| 1f| 1 | 0 |1| 0 | 0 |0| 1f | 00 |
```

Fig. 9. Icarus Verilog multiple-write test command line output.

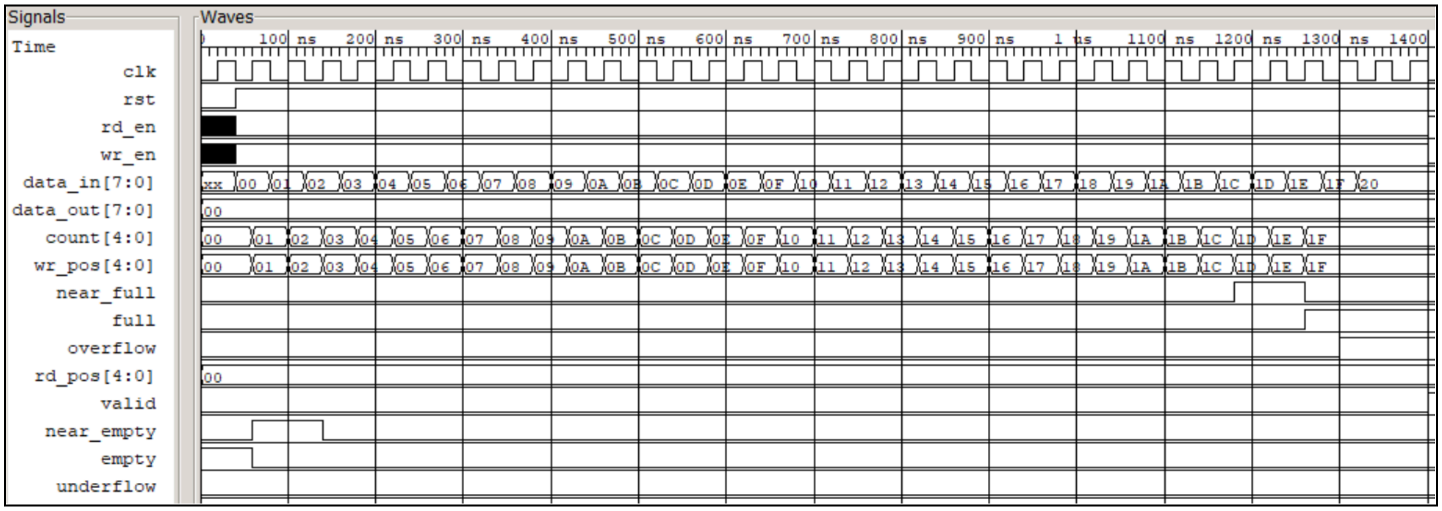
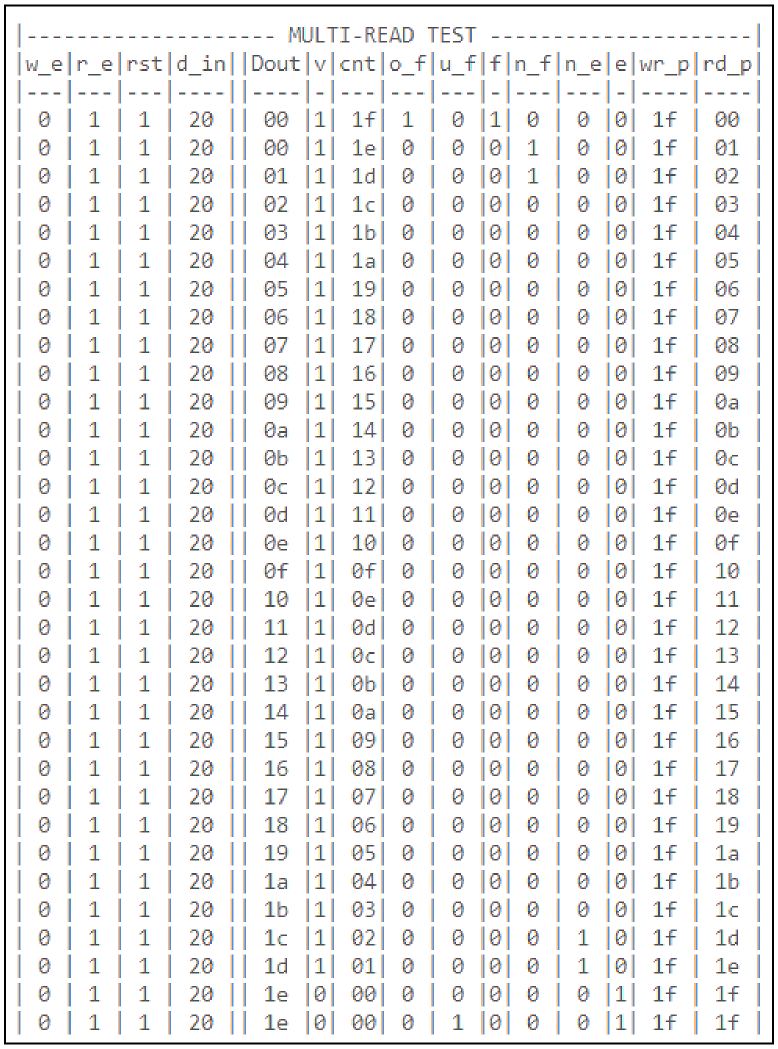


Fig. 10. GTKWave simulation waveforms of multiple-write test.

In the multiple-read test, every memory address was read from at the index stored in the read position pointer. This verified that the data written in the multiple-read test was actually being properly stored in the buffer. However, an interesting problem arose: The data being driven onto the output bus was delayed by one clock cycle, and resulted in an inability to display the last address value since the empty flag was asserted before it could be displayed. Due to time constraints (and after several hours of attempting to eliminate this delay), no resolution was found and the flaw remains in the module. Before the first read operation, the overflow and full flags remained set. After the first read operation, the overflow and full flags were deasserted and the near-full flag was asserted. After the second read operation, the near-full flag was deasserted. Upon performing the 30th read operation, the near-empty flag (n_e) was asserted, and upon performing the 32nd read operation, the near-empty flag was deasserted and the empty flag was asserted. When trying to perform a 33rd read operation, the underflow flag (u_f) was asserted. The Icarus Verilog command line output of the multiple-read test can be seen in figure 11. GTKWave simulation waveforms of the test can be seen in figure 12.



```
|------------------- MULTI-READ TEST --------------------|
|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|
|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|
| 0 | 1 | 1 | 20 || 00 |1| 1f| 1 | 0 |1| 0 | 0 |0| 1f | 00 |
| 0 | 1 | 1 | 20 || 00 |1| 1e| 0 | 0 |0| 1 | 0 |0| 1f | 01 |
| 0 | 1 | 1 | 20 || 01 |1| 1d| 0 | 0 |0| 1 | 0 |0| 1f | 02 |
| 0 | 1 | 1 | 20 || 02 |1| 1c| 0 | 0 |0| 0 | 0 |0| 1f | 03 |
| 0 | 1 | 1 | 20 || 03 |1| 1b| 0 | 0 |0| 0 | 0 |0| 1f | 04 |
| 0 | 1 | 1 | 20 || 04 |1| 1a| 0 | 0 |0| 0 | 0 |0| 1f | 05 |
| 0 | 1 | 1 | 20 || 05 |1| 19| 0 | 0 |0| 0 | 0 |0| 1f | 06 |
| 0 | 1 | 1 | 20 || 06 |1| 18| 0 | 0 |0| 0 | 0 |0| 1f | 07 |
| 0 | 1 | 1 | 20 || 07 |1| 17| 0 | 0 |0| 0 | 0 |0| 1f | 08 |
| 0 | 1 | 1 | 20 || 08 |1| 16| 0 | 0 |0| 0 | 0 |0| 1f | 09 |
| 0 | 1 | 1 | 20 || 09 |1| 15| 0 | 0 |0| 0 | 0 |0| 1f | 0a |
| 0 | 1 | 1 | 20 || 0a |1| 14| 0 | 0 |0| 0 | 0 |0| 1f | 0b |
| 0 | 1 | 1 | 20 || 0b |1| 13| 0 | 0 |0| 0 | 0 |0| 1f | 0c |
| 0 | 1 | 1 | 20 || 0c |1| 12| 0 | 0 |0| 0 | 0 |0| 1f | 0d |
| 0 | 1 | 1 | 20 || 0d |1| 11| 0 | 0 |0| 0 | 0 |0| 1f | 0e |
| 0 | 1 | 1 | 20 || 0e |1| 10| 0 | 0 |0| 0 | 0 |0| 1f | 0f |
| 0 | 1 | 1 | 20 || 0f |1| 0f| 0 | 0 |0| 0 | 0 |0| 1f | 10 |
| 0 | 1 | 1 | 20 || 10 |1| 0e| 0 | 0 |0| 0 | 0 |0| 1f | 11 |
| 0 | 1 | 1 | 20 || 11 |1| 0d| 0 | 0 |0| 0 | 0 |0| 1f | 12 |
| 0 | 1 | 1 | 20 || 12 |1| 0c| 0 | 0 |0| 0 | 0 |0| 1f | 13 |
| 0 | 1 | 1 | 20 || 13 |1| 0b| 0 | 0 |0| 0 | 0 |0| 1f | 14 |
| 0 | 1 | 1 | 20 || 14 |1| 0a| 0 | 0 |0| 0 | 0 |0| 1f | 15 |
| 0 | 1 | 1 | 20 || 15 |1| 09| 0 | 0 |0| 0 | 0 |0| 1f | 16 |
| 0 | 1 | 1 | 20 || 16 |1| 08| 0 | 0 |0| 0 | 0 |0| 1f | 17 |
| 0 | 1 | 1 | 20 || 17 |1| 07| 0 | 0 |0| 0 | 0 |0| 1f | 18 |
| 0 | 1 | 1 | 20 || 18 |1| 06| 0 | 0 |0| 0 | 0 |0| 1f | 19 |
| 0 | 1 | 1 | 20 || 19 |1| 05| 0 | 0 |0| 0 | 0 |0| 1f | 1a |
| 0 | 1 | 1 | 20 || 1a |1| 04| 0 | 0 |0| 0 | 0 |0| 1f | 1b |
| 0 | 1 | 1 | 20 || 1b |1| 03| 0 | 0 |0| 0 | 0 |0| 1f | 1c |
| 0 | 1 | 1 | 20 || 1c |1| 02| 0 | 0 |0| 0 | 1 |0| 1f | 1d |
| 0 | 1 | 1 | 20 || 1d |1| 01| 0 | 0 |0| 0 | 1 |0| 1f | 1e |
| 0 | 1 | 1 | 20 || 1e |0| 00| 0 | 0 |0| 0 | 0 |1| 1f | 1f |
| 0 | 1 | 1 | 20 || 1e |0| 00| 0 | 1 |0| 0 | 0 |1| 1f | 1f |
```

Fig. 11. Icarus Verilog multiple-read test command line output.

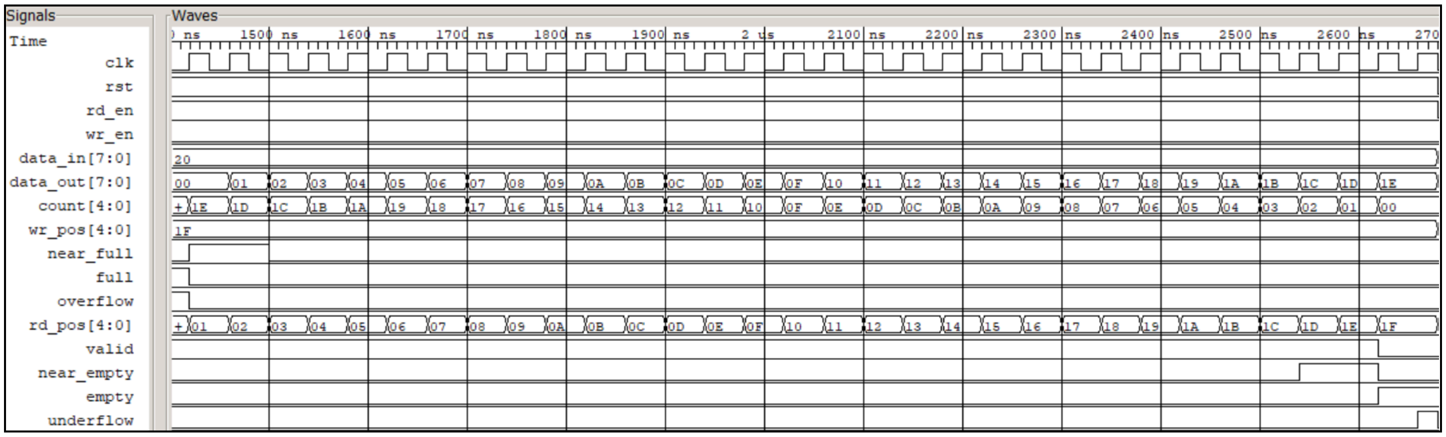
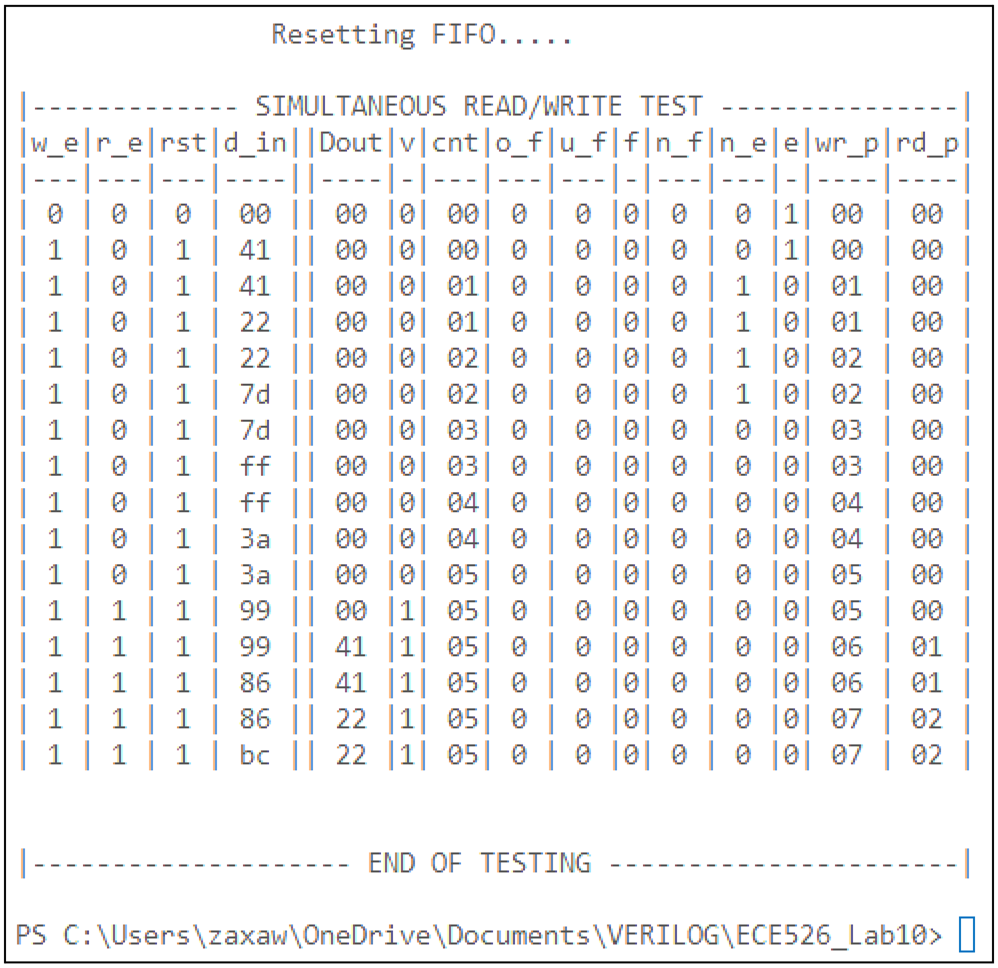


Fig. 12. GTKWave simulation waveforms of multiple-read test.

In the simultaneous read/write test, the reset input (rst) was asserted and cleared the buffer. While each proceeding write operation happened only once, they appeared to happen in duplicate due to the $monitor system task detecting a change in the data input bus before a corresponding change in the write position pointer and depth counter (cnt). After the fifth write operation, the read enable signal was asserted and the buffer began simultaneously reading and writing. For the remaining clock cycles, new data is being written to the buffer as it is reading previously written data. The Icarus Verilog command line output of the simultaneous read/write test can be seen in figure 13. GTKWave simulation waveforms of the test can be seen in figure 14.



```
                    Resetting FIFO.....

|------------- SIMULTANEOUS READ/WRITE TEST ---------------|
|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|
|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|
| 0 | 0 | 0 | 00 || 00 |0| 00| 0 | 0 |0| 0 | 0 |1| 00 | 00 |
| 1 | 0 | 1 | 41 || 00 |0| 00| 0 | 0 |0| 0 | 0 |1| 00 | 00 |
| 1 | 0 | 1 | 41 || 00 |0| 01| 0 | 0 |0| 0 | 1 |0| 01 | 00 |
| 1 | 0 | 1 | 22 || 00 |0| 01| 0 | 0 |0| 0 | 1 |0| 01 | 00 |
| 1 | 0 | 1 | 22 || 00 |0| 02| 0 | 0 |0| 0 | 1 |0| 02 | 00 |
| 1 | 0 | 1 | 7d || 00 |0| 02| 0 | 0 |0| 0 | 1 |0| 02 | 00 |
| 1 | 0 | 1 | 7d || 00 |0| 03| 0 | 0 |0| 0 | 0 |0| 03 | 00 |
| 1 | 0 | 1 | ff || 00 |0| 03| 0 | 0 |0| 0 | 0 |0| 03 | 00 |
| 1 | 0 | 1 | ff || 00 |0| 04| 0 | 0 |0| 0 | 0 |0| 04 | 00 |
| 1 | 0 | 1 | 3a || 00 |0| 04| 0 | 0 |0| 0 | 0 |0| 04 | 00 |
| 1 | 0 | 1 | 3a || 00 |0| 05| 0 | 0 |0| 0 | 0 |0| 05 | 00 |
| 1 | 1 | 1 | 99 || 00 |1| 05| 0 | 0 |0| 0 | 0 |0| 05 | 00 |
| 1 | 1 | 1 | 99 || 41 |1| 05| 0 | 0 |0| 0 | 0 |0| 06 | 01 |
| 1 | 1 | 1 | 86 || 41 |1| 05| 0 | 0 |0| 0 | 0 |0| 06 | 01 |
| 1 | 1 | 1 | 86 || 22 |1| 05| 0 | 0 |0| 0 | 0 |0| 07 | 02 |
| 1 | 1 | 1 | bc || 22 |1| 05| 0 | 0 |0| 0 | 0 |0| 07 | 02 |


|------------------- END OF TESTING ---------------------|

PS C:\Users\zaxaw\OneDrive\Documents\VERILOG\ECE526_Lab10>
```

Fig. 13. Icarus Verilog simultaneous read/write test command line output.

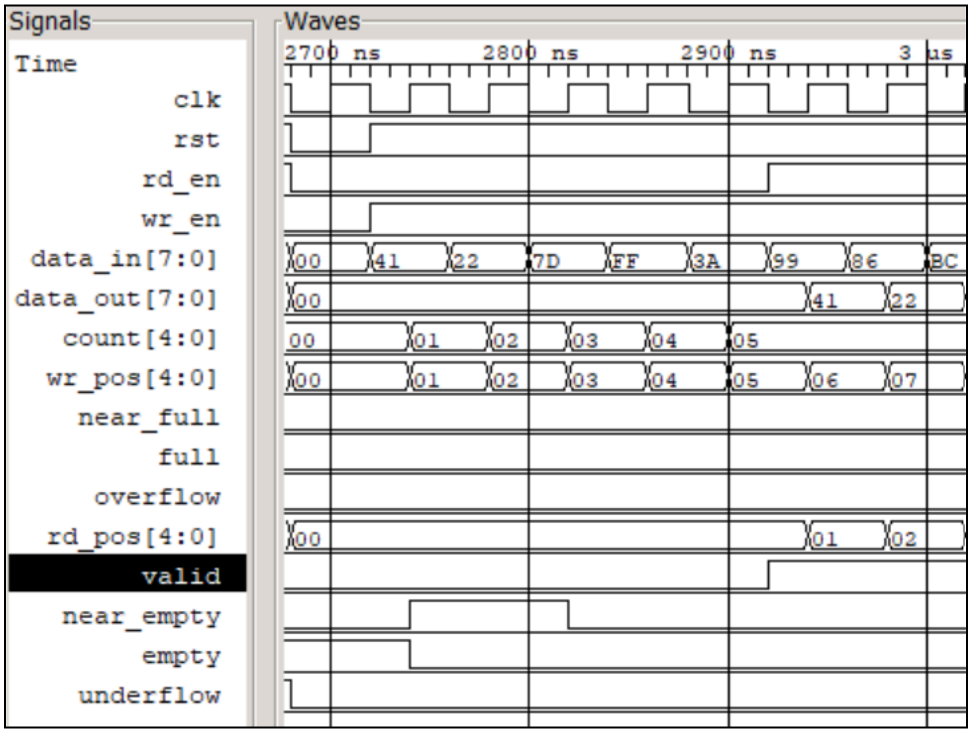


Fig. 14. GTKWave simulation waveforms of simultaneous read/write test.

IV. CONCLUSION

This lab mostly demonstrated the principles of a first-in-first-out data buffer. The functionality of the FIFO module was validated through thorough testing including overflow and underflow. Under standard operations, as well as during simultaneous read/write sequences, the FIFO module's primary logic operated as intended.

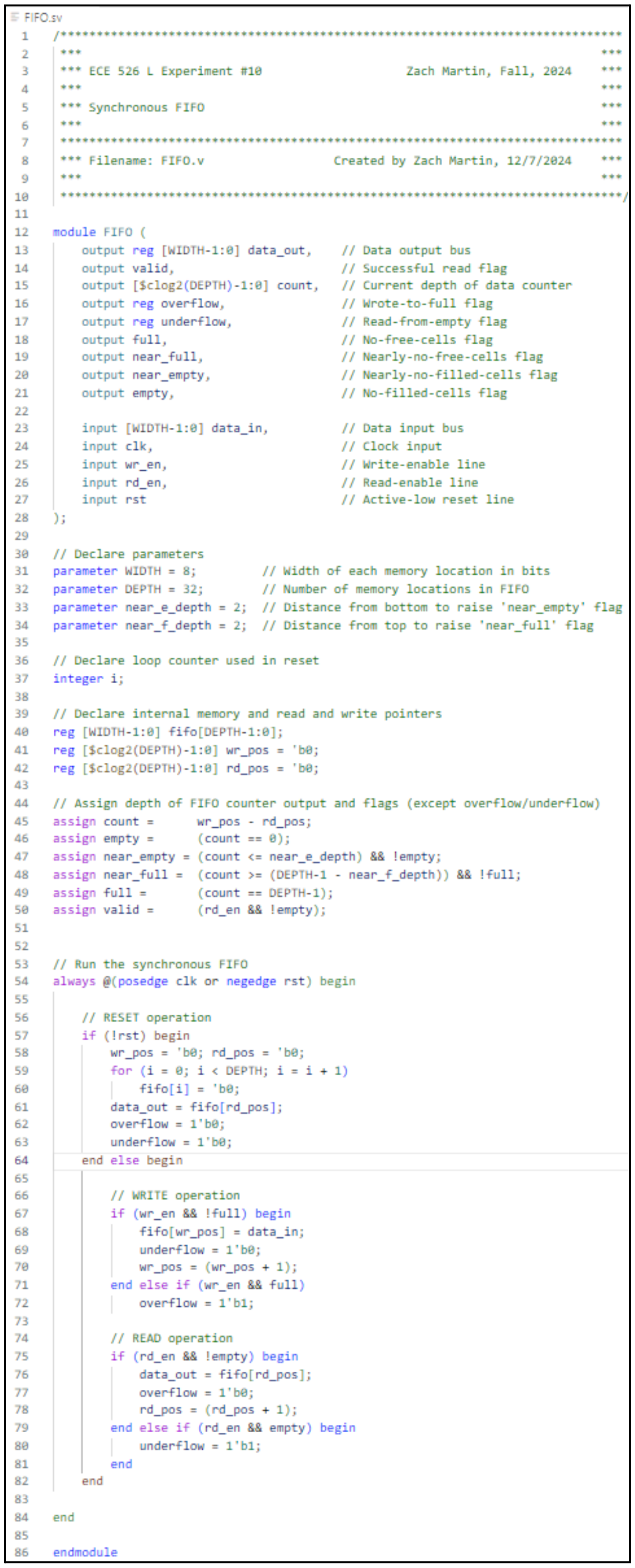
However, the testing process revealed a one-clock cycle delay in the output data during the multiple-read test, which prevented the last address value from being displayed before the empty flag was asserted. Despite several hours devoted to diagnosing and attempting to correct this delay, the issue could not be resolved within the available time. This failure emphasizes the difficulty of managing synchronization and timing alignment in digital circuit design, even when core functionality appears valid. Overall, this lab demonstrated the importance of rigorous testing to identify limitations and provided valuable insights into the complexities of implementing different organization methods.

On a personal note, this one was very frustrating. I probably spent 8 hours on the read delay, and I still don't know how to fix the timing issue. If you have working code that you'd be comfortable sharing with me, I would love to see a proper design. Also, I know that we used MobaXterm as a class, but Icarus Verilog is so much more convenient without having to log in to the VPN and using a shell to remotely compile; it's also on Windows, Mac, and Linux so you can do all the things that bash or powershell or whatnot let you do, which was very nice. The waveform viewer is about on par with VCS, and from what I can tell isn't missing any key features. If you haven't heard of it, check it out:

https://steveicarus.github.io/iverilog/index.html

I learned a lot this semester. Thank you.

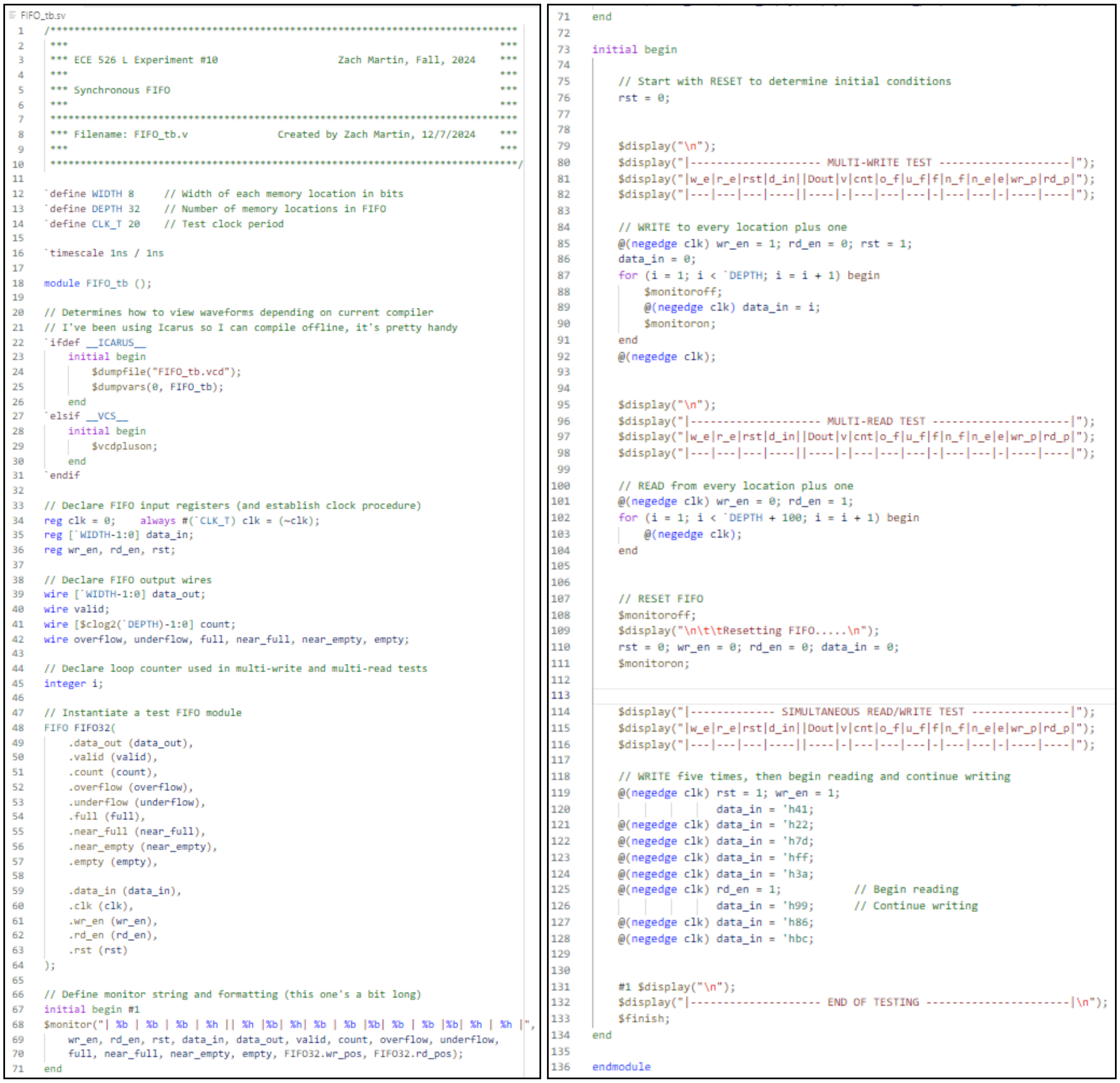
APPENDIX



FIFO.sv

```verilog
/******************************************************************************
 ***                                                                        ***
 *** ECE 526 L Experiment #10                    Zach Martin, Fall, 2024    ***
 ***                                                                        ***
 *** Synchronous FIFO                                                       ***
 ***                                                                        ***
 ******************************************************************************
 *** Filename: FIFO.v                  Created by Zach Martin, 12/7/2024    ***
 ***                                                                        ***
 ******************************************************************************/

module FIFO (
    output reg [WIDTH-1:0] data_out,    // Data output bus
    output valid,                       // Successful read flag
    output [$clog2(DEPTH)-1:0] count,   // Current depth of data counter
    output reg overflow,                // Wrote-to-full flag
    output reg underflow,               // Read-from-empty flag
    output full,                        // No-free-cells flag
    output near_full,                   // Nearly-no-free-cells flag
    output near_empty,                  // Nearly-no-filled-cells flag
    output empty,                       // No-filled-cells flag

    input [WIDTH-1:0] data_in,          // Data input bus
    input clk,                          // Clock input
    input wr_en,                        // Write-enable line
    input rd_en,                        // Read-enable line
    input rst                           // Active-low reset line
);

// Declare parameters
parameter WIDTH = 8;         // Width of each memory location in bits
parameter DEPTH = 32;        // Number of memory locations in FIFO
parameter near_e_depth = 2;  // Distance from bottom to raise 'near_empty' flag
parameter near_f_depth = 2;  // Distance from top to raise 'near_full' flag

// Declare loop counter used in reset
integer i;

// Declare internal memory and read and write pointers
reg [WIDTH-1:0] fifo[DEPTH-1:0];
reg [$clog2(DEPTH)-1:0] wr_pos = 'b0;
reg [$clog2(DEPTH)-1:0] rd_pos = 'b0;

// Assign depth of FIFO counter output and flags (except overflow/underflow)
assign count =      wr_pos - rd_pos;
assign empty =      (count == 0);
assign near_empty = (count <= near_e_depth) && !empty;
assign near_full =  (count >= (DEPTH-1 - near_f_depth)) && !full;
assign full =       (count == DEPTH-1);
assign valid =      (rd_en && !empty);


// Run the synchronous FIFO
always @(posedge clk or negedge rst) begin

    // RESET operation
    if (!rst) begin
        wr_pos = 'b0; rd_pos = 'b0;
        for (i = 0; i < DEPTH; i = i + 1)
            fifo[i] = 'b0;
        data_out = fifo[rd_pos];
        overflow = 1'b0;
        underflow = 1'b0;
    end else begin

        // WRITE operation
        if (wr_en && !full) begin
            fifo[wr_pos] = data_in;
            underflow = 1'b0;
            wr_pos = (wr_pos + 1);
        end else if (wr_en && full)
            overflow = 1'b1;

        // READ operation
        if (rd_en && !empty) begin
            data_out = fifo[rd_pos];
            overflow = 1'b0;
            rd_pos = (rd_pos + 1);
        end else if (rd_en && empty) begin
            underflow = 1'b1;
        end
    end

end

endmodule
```

FIFO.sv file.



FIFO_tb.sv

```verilog
/*****************************************************************************
 ***                                                                       ***
 *** ECE 526 L Experiment #10                    Zach Martin, Fall, 2024   ***
 ***                                                                       ***
 *** Synchronous FIFO                                                      ***
 ***                                                                       ***
 *****************************************************************************
 *** Filename: FIFO_tb.v               Created by Zach Martin, 12/7/2024   ***
 ***                                                                       ***
 *****************************************************************************/

`define WIDTH 8     // Width of each memory location in bits
`define DEPTH 32    // Number of memory locations in FIFO
`define CLK_T 20    // Test clock period

`timescale 1ns / 1ns

module FIFO_tb ();

// Determines how to view waveforms depending on current compiler
// I've been using Icarus so I can compile offline, it's pretty handy
`ifdef __ICARUS__
    initial begin
        $dumpfile("FIFO_tb.vcd");
        $dumpvars(0, FIFO_tb);
    end
`elsif __VCS__
    initial begin
        $vcdpluson;
    end
`endif

// Declare FIFO input registers (and establish clock procedure)
reg clk = 0;    always #(`CLK_T) clk = (~clk);
reg [`WIDTH-1:0] data_in;
reg wr_en, rd_en, rst;

// Declare FIFO output wires
wire [`WIDTH-1:0] data_out;
wire valid;
wire [$clog2(`DEPTH)-1:0] count;
wire overflow, underflow, full, near_full, near_empty, empty;

// Declare loop counter used in multi-write and multi-read tests
integer i;

// Instantiate a test FIFO module
FIFO FIFO32(
    .data_out (data_out),
    .valid (valid),
    .count (count),
    .overflow (overflow),
    .underflow (underflow),
    .full (full),
    .near_full (near_full),
    .near_empty (near_empty),
    .empty (empty),

    .data_in (data_in),
    .clk (clk),
    .wr_en (wr_en),
    .rd_en (rd_en),
    .rst (rst)
);

// Define monitor string and formatting (this one's a bit long)
initial begin #1
$monitor("| %b | %b | %b | %h || %h |%b| %h| %b | %b |%b| %b | %b |%b| %h | %h |",
    wr_en, rd_en, rst, data_in, data_out, valid, count, overflow, underflow,
    full, near_full, near_empty, empty, FIFO32.wr_pos, FIFO32.rd_pos);
end

initial begin

    // Start with RESET to determine initial conditions
    rst = 0;


    $display("\n");
    $display("|-------------------- MULTI-WRITE TEST --------------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // WRITE to every location plus one
    @(negedge clk) wr_en = 1; rd_en = 0; rst = 1;
    data_in = 0;
    for (i = 1; i < `DEPTH; i = i + 1) begin
        $monitoroff;
        @(negedge clk) data_in = i;
        $monitoron;
    end
    @(negedge clk);


    $display("\n");
    $display("|-------------------- MULTI-READ TEST ---------------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // READ from every location plus one
    @(negedge clk) wr_en = 0; rd_en = 1;
    for (i = 1; i < `DEPTH + 100; i = i + 1) begin
        @(negedge clk);
    end


    // RESET FIFO
    $monitoroff;
    $display("\n\t\tResetting FIFO.....\n");
    rst = 0; wr_en = 0; rd_en = 0; data_in = 0;
    $monitoron;


    $display("|------------- SIMULTANEOUS READ/WRITE TEST ---------------|");
    $display("|w_e|r_e|rst|d_in||Dout|v|cnt|o_f|u_f|f|n_f|n_e|e|wr_p|rd_p|");
    $display("|---|---|---|----||----|-|---|---|---|-|---|---|-|----|----|");

    // WRITE five times, then begin reading and continue writing
    @(negedge clk) rst = 1; wr_en = 1;
                   data_in = 'h41;
    @(negedge clk) data_in = 'h22;
    @(negedge clk) data_in = 'h7d;
    @(negedge clk) data_in = 'hff;
    @(negedge clk) data_in = 'h3a;
    @(negedge clk) rd_en = 1;           // Begin reading
                   data_in = 'h99;      // Continue writing
    @(negedge clk) data_in = 'h86;
    @(negedge clk) data_in = 'hbc;


    #1 $display("\n");
    $display("|-------------------- END OF TESTING ---------------------|\n");
    $finish;
end

endmodule
```

FIFO_tb.sv file.