ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E
LOGIC CIRCUITS LABORATORY
DATA BUS IMPLEMENTATION AND MEMORY DESIGN

HOMEWORK NO	:	3
GROUP NO	:	G08

GROUP MEMBERS:

150200081	:	KAAN KARATAŞ
150210719	:	NACİ TOYGUN GÖRMÜŞ

SPRING 2023

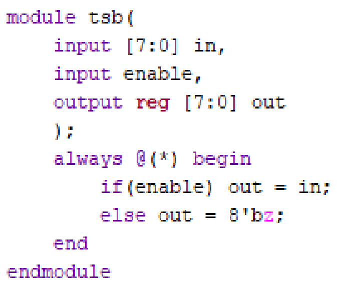
Contents

1 INTRODUCTION

In this experiment, we have used three-state buffers to implement data buses and basic memory. For simulating the corresponding modules, Vivado is used. Lastly, Verilog is used to code and implement all the modules.

2 HOMEWORK

2.1 Three State Buffer



```
module tsb(
    input [7:0] in,
    input enable,
    output reg [7:0] out
    );
    always @(*) begin
        if(enable) out = in;
        else out = 8'bz;
    end
endmodule
```

Figure 1: Three State Buffer Code

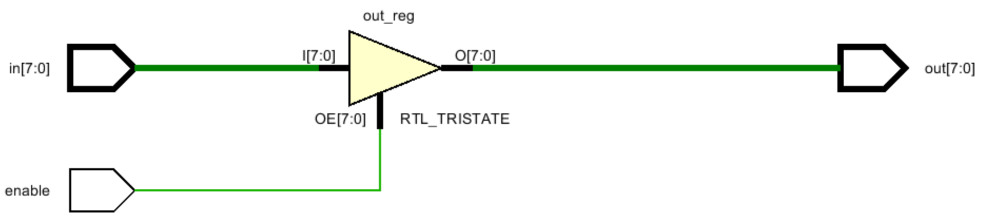


Figure 2: Three State Buffer Schematic

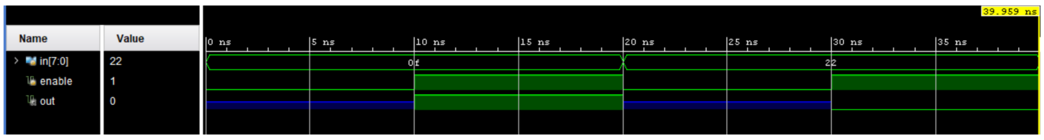


Figure 3: Three State Buffer Simulation

2.2 Part 1

Three state buffer is used to implement an 8-bit bus. Changes to the selection input affect what is assigned to the output.

```
module part1(
    input [7:0] data1,
    input [7:0] data2,
    input select,
    output [7:0] out
);
    tsb b1(data1, !select, out);
    tsb b2(data2, select, out);
endmodule
```

Figure 4: Part 1 Code

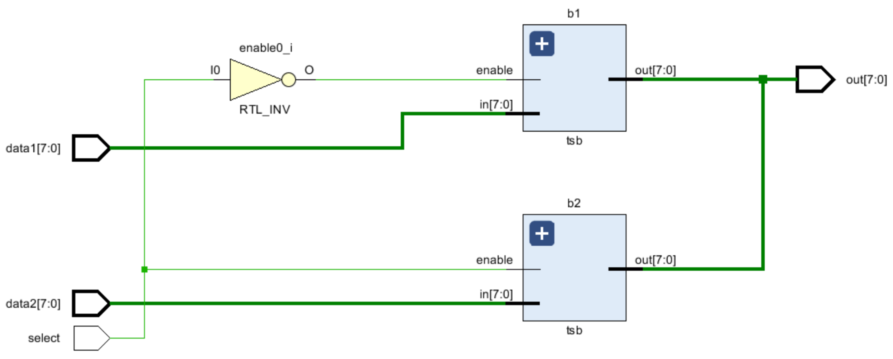


Figure 5: Part 1 Schematic

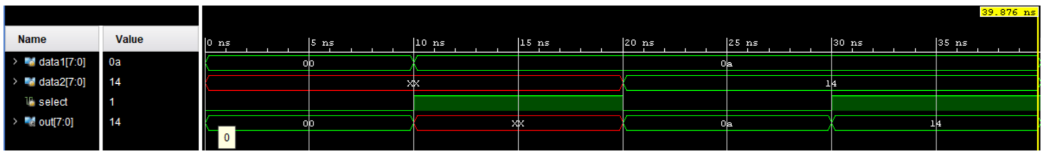


Figure 6: Part 1 Simulation

2.3 Part 2

Specified Memory design containing a bus and two outputs with inputs is implemented in this part.

```
module part2(
    input [7:0] data1,
    input [7:0] data2,
    input select,
    output [7:0] out1,
    output [7:0] out2
);
    wire [7:0]bus;
    part1 p1(data1, data2, select, bus);
    tsb b0(bus, !select, out1);
    tsb b1(bus, select, out2);
endmodule
```

Figure 7: Part 2 Code

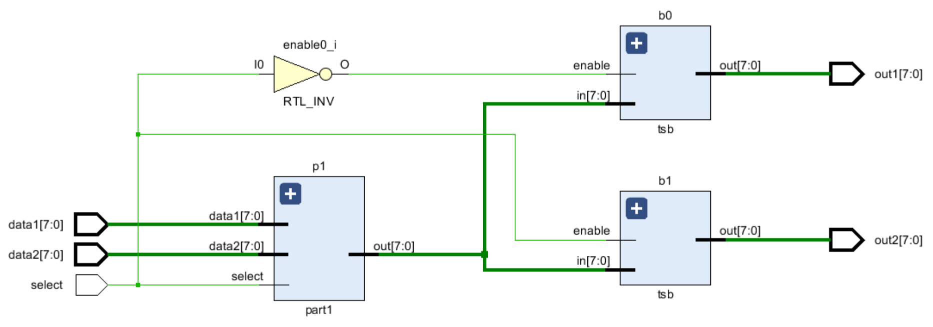


Figure 8: Part 2 Schematic

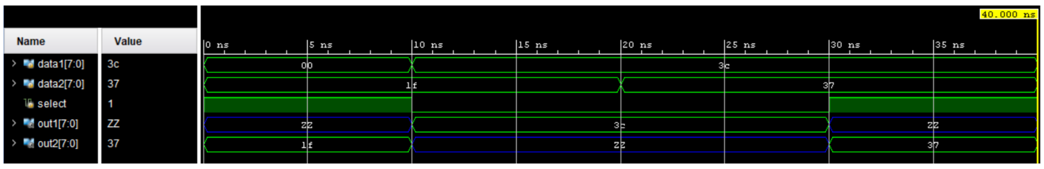


Figure 9: Part 2 Simulation

2.4 Part 3

We have implemented an 8-bit memory line module. This module accepts 8-bit inputs and outputs 8-bit data. Additionally, the module accepts inputs for line selection, reset , write enable, read enable and clock for necessary operations. The module stores the data value that is provided as input when the write enable and line select inputs are high, at the rising edge of the clock signal. At the falling edge of the reset signal, the module erases the data that has been stored. The output of the module is the stored data if the inputs for read enable and line select are both high. The output has a high impedance if not.

```
module part3(
    input [7:0] in,
    input reset,
    input l_select,
    input read_e,
    input write_e,
    input clk,
    output reg [7:0] out
);
    reg [7:0] memory;
    always @(posedge clk)begin
        if(write_e & l_select) memory <= in;
    end
    always @(negedge reset) begin
        memory <= 8'd0;
    end
    always @(*) begin
        if(read_e & l_select) out <= memory;
        else out <= 8'bz;
    end
endmodule
```

Figure 10: Part 3 Code

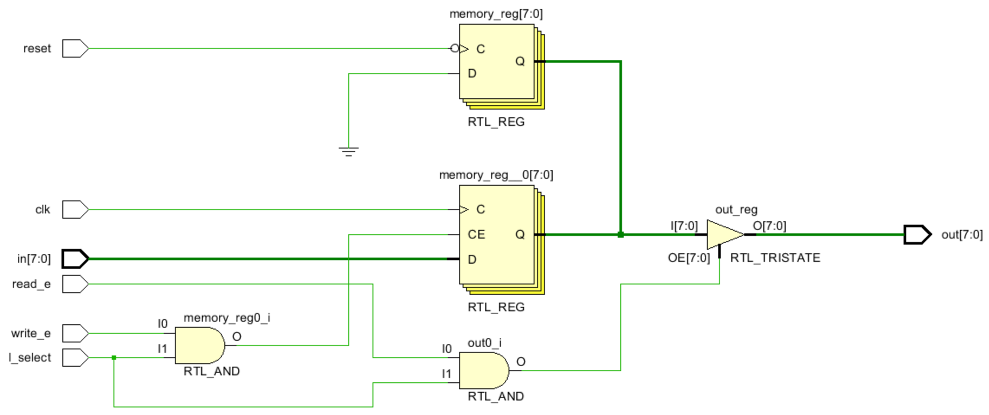
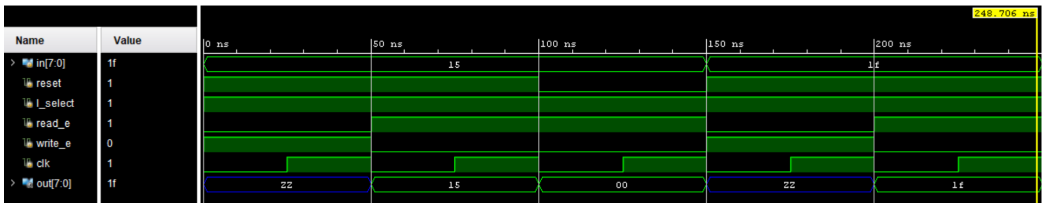


Figure 11: Part 3 Schematic

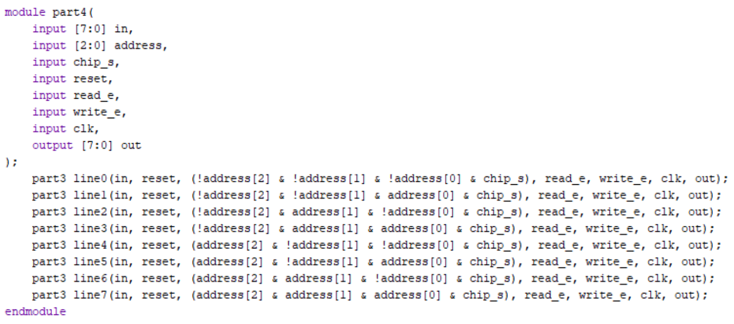


Name	Value
in[7:0]	1f
reset	1
I_select	1
read_e	1
write_e	0
clk	1
out[7:0]	1f

Figure 12: Part 3 Simulation

2.5 Part 4

Utilizing an 8-bit memory line module, an 8 byte memory module is implemented. The input and output of an 8-byte memory module are both 8-bit data. Inputs for the module's required operations include a 3-bit address, chip select, reset, read enable, write enable, and clock. When the chip select input is high, the Nth memory line is chosen. Address number N is shown here. If the write enable input is high, the chosen memory line stores the data that is input at the rising edge of the clock signal. At the falling edge of the reset signal, the module erases all data that has been stored in the memory lines. If the read enable input is high, the module's output is the data from the selected memory line.



```
module part4(
    input [7:0] in,
    input [2:0] address,
    input chip_s,
    input reset,
    input read_e,
    input write_e,
    input clk,
    output [7:0] out
);
    part3 line0(in, reset, (!address[2] & !address[1] & !address[0] & chip_s), read_e, write_e, clk, out);
    part3 line1(in, reset, (!address[2] & !address[1] & address[0] & chip_s), read_e, write_e, clk, out);
    part3 line2(in, reset, (!address[2] & address[1] & !address[0] & chip_s), read_e, write_e, clk, out);
    part3 line3(in, reset, (!address[2] & address[1] & address[0] & chip_s), read_e, write_e, clk, out);
    part3 line4(in, reset, (address[2] & !address[1] & !address[0] & chip_s), read_e, write_e, clk, out);
    part3 line5(in, reset, (address[2] & !address[1] & address[0] & chip_s), read_e, write_e, clk, out);
    part3 line6(in, reset, (address[2] & address[1] & !address[0] & chip_s), read_e, write_e, clk, out);
    part3 line7(in, reset, (address[2] & address[1] & address[0] & chip_s), read_e, write_e, clk, out);
endmodule
```

Figure 13: Part 4 Code

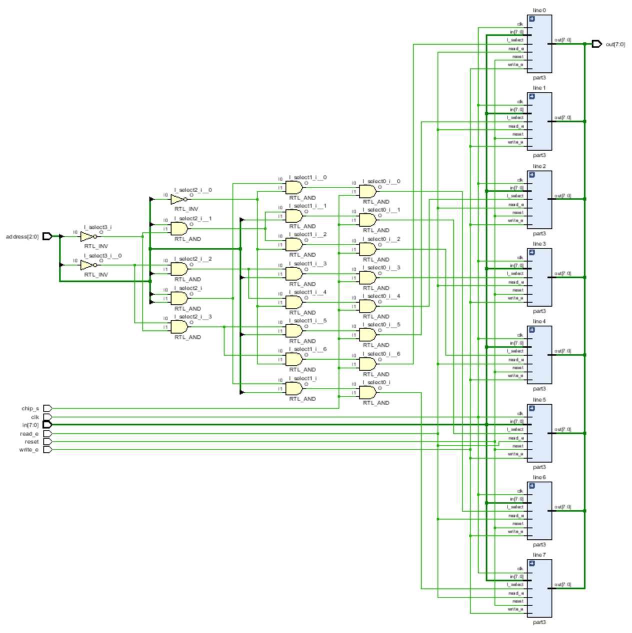
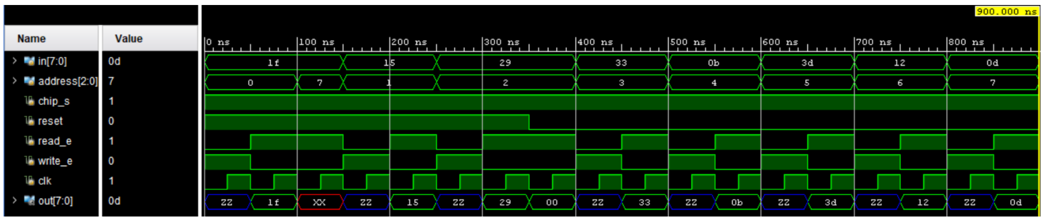


Figure 14: Part 4 Schematic



Name	Value
in[7:0]	0d
address[2:0]	7
chip_s	1
reset	0
read_e	1
write_e	0
clk	1
out[7:0]	0d

Figure 15: Part 4 Simulation

2.6 Part 5

Implemented design is a 32 byte memory module using an 8 byte memory module. It accepts 8-bit data as input and outputs 8-bit data. This module also accepts inputs for a 5-bit address, reset, read enable, write enable, and clock. The remaining 3 bits of the address input are used to select the line, while the remaining 2 bits are used to select the chip. The same procedures as in Part 4 apply.

```
module part5(
    input [7:0] in,
    input [4:0] address,
    input reset,
    input read_e,
    input write_e,
    input clk,
    output [7:0] out
);
    part4 mem0(in, address[2:0], (!address[4] & !address[3]), reset, read_e, write_e, clk, out);
    part4 mem1(in, address[2:0], (!address[4] & address[3]), reset, read_e, write_e, clk, out);
    part4 mem2(in, address[2:0], (address[4] & !address[3]), reset, read_e, write_e, clk, out);
    part4 mem3(in, address[2:0], (address[4] & address[3]), reset, read_e, write_e, clk, out);
endmodule
```

Figure 16: Part 5 Code

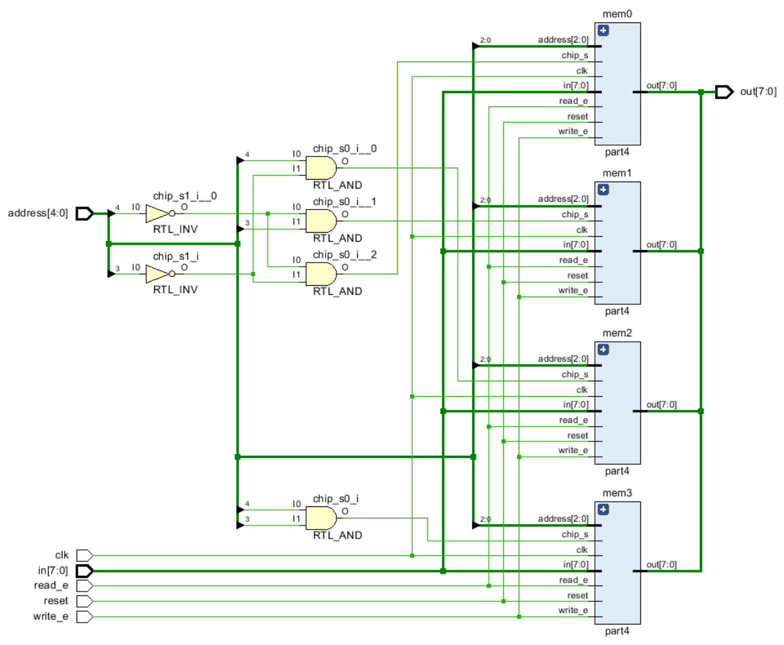


Figure 17: Part 5 Schematic

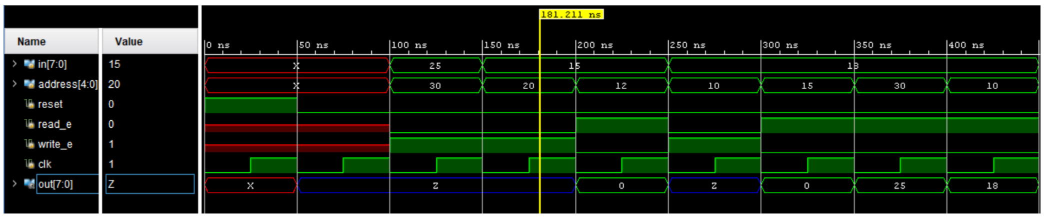
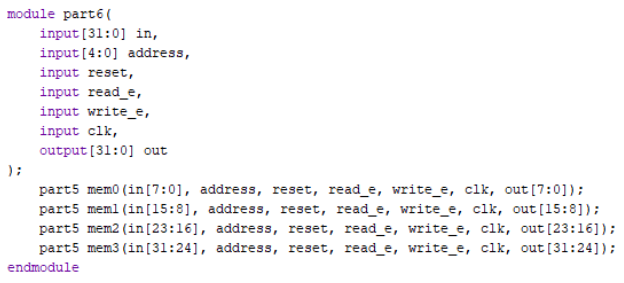


Figure 18: Part 5 Simulation

2.7 Part 6

32-byte memory modules are used to implement 128-byte memory modules. 32-bit data is inputted into the 128-byte memory module and outputted in the same format. This module also accepts clock, address, reset, read enable, and write enable inputs. The same procedures as in Parts 4 and 5 apply. In order to implement 128 bytes of memory, 4 32-byte memories are utilized.



```
module part6(
    input[31:0] in,
    input[4:0] address,
    input reset,
    input read_e,
    input write_e,
    input clk,
    output[31:0] out
);
    part5 mem0(in[7:0], address, reset, read_e, write_e, clk, out[7:0]);
    part5 mem1(in[15:8], address, reset, read_e, write_e, clk, out[15:8]);
    part5 mem2(in[23:16], address, reset, read_e, write_e, clk, out[23:16]);
    part5 mem3(in[31:24], address, reset, read_e, write_e, clk, out[31:24]);
endmodule
```

Figure 19: Part 6 Code

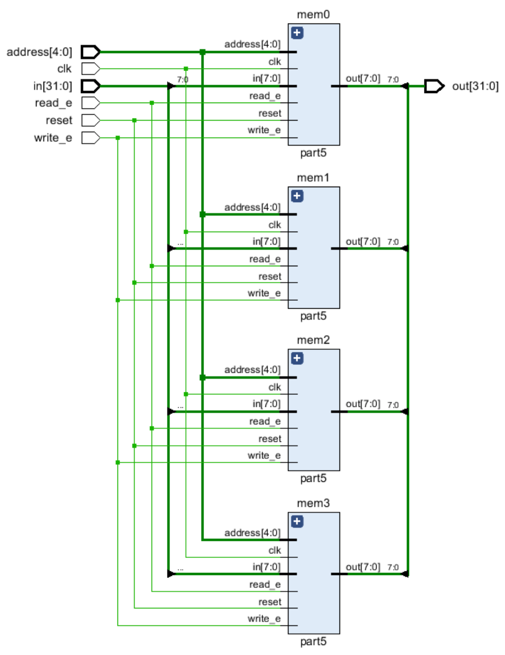


Figure 20: Part 6 Schematic

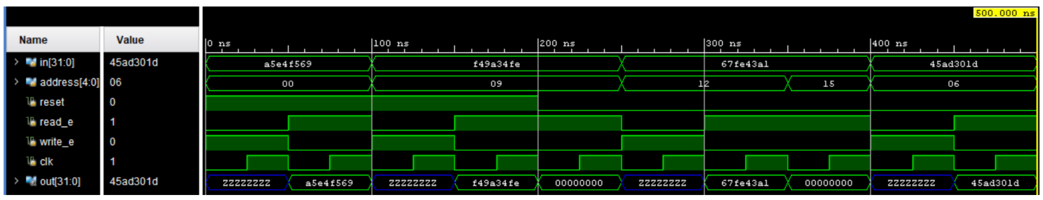


Figure 21: Part 6 Simulation

3 RESULTS

Every memory and buss design in the homework assignment is implemented using Verilog. The test results are the same with what we had predicted and expected before implementing modules. All test results are given on the relevant section.

4 DISCUSSION

Using Verilog programming language, we learned how complex memories work, implementing and using complex memory designs and busses through-out this homework. Our outcomes were fully in line with what we had anticipated.

5 CONCLUSION

With the help of the Verilog programming language, we were able to simulate data busses and basic memory components, succeeding in implementing them through this homework. To sum up, this was not a rather challenging homework for us compered to other assignments, and we were able to successfully complete all the parts.