ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	6
EXPERIMENT DATE	:	30.04.2021
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G17

GROUP MEMBERS:

150180024 : ŞULE BEYZA KARADAĞ

150190710 : SENİHA SERRA BOZKURT

150190024 : AHMET FURKAN KAVRAZ

SPRING 2021

Contents

1 INTRODUCTION [10 points]

In the following experiment, we are going to implement data buses and basic memory by using three-state buffers in Verilog. We will learn:

- How to analyze a given circuit with data bus,
- How to implement an 8-bit bus by using buffers / readers / drivers,
- How to implement an 8-bit memory line module,
- How to implement an 8 byte memory module using 8-bit memory line module,
- How to implement a 32 byte memory module using 8 byte memory module,
- How to implement a 128 byte memory module using 32 byte memory module.

2 MATERIALS AND METHODS [40 points]

2.1 Experiment

2.1.1 Modules we Used:

```
module not_gate(
    input wire a,
    output wire not_a
    );
    assign not_a = ~a;
endmodule
module and_gate_2input(
    input wire a,
    input wire b,
    output wire o
    );
    assign o = a && b;
endmodule

module and_gate_3input(
    input wire a,
    input wire b,
    input wire c,
    output wire o
    );
    assign o = a && b && c;
endmodule
```

(a) NOT gate and AND gate modules with different number of inputs

```
module decoder_3input(
    input wire [2:0] s,
    output wire [7:0] O
    );
    wire not_s0, not_s1, not_s2;

    not_gate not0(s[0], not_s0);
    not_gate not1(s[1], not_s1);
    not_gate not2(s[2], not_s2);

    and_gate_3input and0(not_s2, not_s1, not_s0, O[0]); // 000
    and_gate_3input and1(not_s2, not_s1, s[0]  , O[1]); // 001
    and_gate_3input and2(not_s2, s[1]  , not_s0, O[2]); // 010
    and_gate_3input and3(not_s2, s[1]  , s[0]  , O[3]); // 011
    and_gate_3input and4(s[2]  , not_s1, not_s0, O[4]); // 100
    and_gate_3input and5(s[2]  , not_s1, s[0]  , O[5]); // 101
    and_gate_3input and6(s[2]  , s[1]  , not_s0, O[6]); // 110
    and_gate_3input and7(s[2]  , s[1]  , s[0]  , O[7]); // 111
endmodule

module decoder_2input(
    input wire [1:0] s,
    output wire [3:0] O
    );
    wire not_s0, not_s1;

    not_gate not0(s[0], not_s0);
    not_gate not1(s[1], not_s1);

    and_gate_2input and0(not_s1, not_s0, O[0]); // 00
    and_gate_2input and1(not_s1, s[0]  , O[1]); // 01
    and_gate_2input and2(s[1]  , not_s0, O[2]); // 10
    and_gate_2input and3(s[1]  , s[0]  , O[3]); // 11
endmodule
```

(b) Decoder with 3 inputs

2.1.2 Part 1

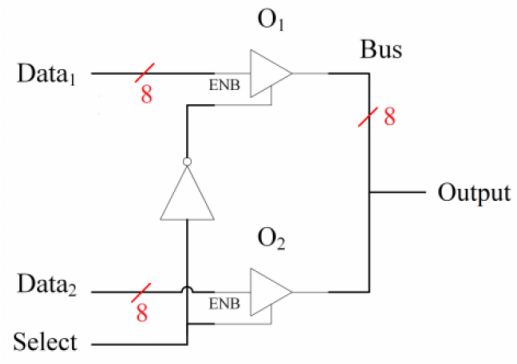


Figure 2: Circuit of 8-bit data bus with 2 drivers with 3-state buffers

```
/*      Buffer      */
module buffer(
    input wire [7:0] Input,
    input wire enable,
    output wire [7:0] Output
    );
   assign Output = enable ? Input : 8'bZ;

endmodule
```

(a) Buffer

```
/*          part1          */
module part1(
    input wire [7:0] Input1,
    input wire [7:0] Input2,
    input wire enable,
    output wire [7:0] Output
    );
    wire o1, o2;
    buffer b0(Input1, ~enable, Output);
    buffer b1(Input2, enable, Output);

endmodule
```

(b) Module code of the circuit given in Figure 2

Figure 3: Implemented 8-bit data bus with 2 drivers with 3-state buffers

In this part, we implemented an 8-bit bus in Figure 2 given to us in the pdf by using Verilog. We used 2 drivers with 3-state buffers. As we learned from what is mentioned in the homework pdf, there has to be only one active 3-state buffer which runs the bus as an output. In accordance with this purpose, we use Buffer module and assign the inputs to the output according to the value of enable. If enable is 1, the input value will be assigned to Output; if enable is 0, ??? (see Figure 3)

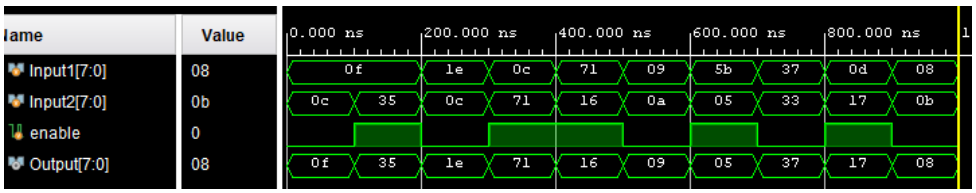
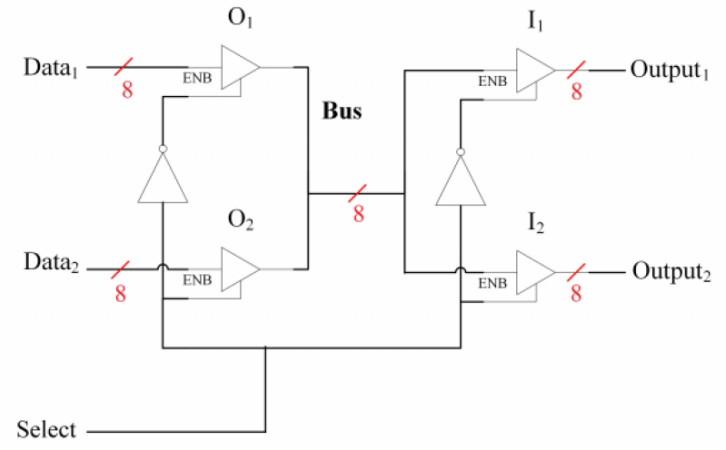


Figure 4: Simulation result of the circuit given in Part 1

2.1.3 Part 2



(a) Circuit of 8-bit data bus with 2 drivers and 2 readers

```
/*          part2          */
module part2(
    input wire [7:0] Input1,
    input wire [7:0] Input2,
    input wire enable,
    output wire [7:0] Output1,
    output wire [7:0] Output2
    );

    wire [7:0] o;
    buffer b0(Input1, ~enable, o);
    buffer b1(Input2, enable, o);

    buffer b2(o, ~enable, Output1);
    buffer b3(o, enable, Output2);

endmodule
```

(b) Module code of the circuit given in Figure 3.a

Figure 5: Implemented 8-bit data bus with 2 drivers and 2 readers

In this part, we implemented 8-bit data bus with 2 drivers and 2 readers in Figure 5.a given to us in the pdf by using Verilog. We used 3-state buffers for both drivers and readers. According to the value of enable, one of the 2 buffers on the left hand side will determine the value of *wire o*. Then, enable will also determine which one of the buffers on the right hand side will on. If enable is 0, b0 and b2 will be on; *Output1* will be *Input1*. If enable is 1, b1 and b3 will be on; *Output2* will be *Input2*. Buffers which are off will make the outputs high impedance.

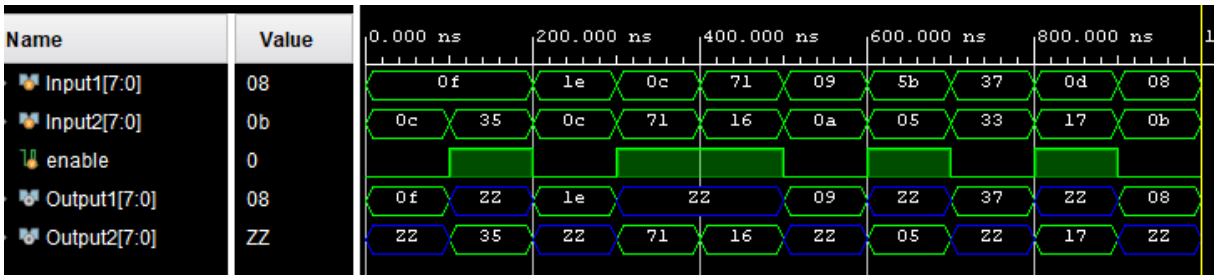


Figure 6: Simulation result of the circuit given in Part 2

2.1.4 Part 3

In this part, we implemented an 8-bit memory line module. This module took 8-bit data as input and give 8-bit data as output. Also, the module took reset, line select, read enable, write enable, and clock inputs for required operations. Required operations are below.

• At the rising edge of the clock signal, the module stored the data value which is given as input when the write enable and line select inputs are high.

• The module cleared the stored data at the falling edge of the reset signal.

• If read enable and line select inputs are high, the output of the module should be the stored data. Otherwise, the output should be high impedance.

```
/*          part3          */
module part3(
    input wire [7:0] Input,
    input wire reset,
    input wire line_select,
    input wire read_enable,
    input wire write_enable,
    input wire clock,
    output wire [7:0] Output
    );
    reg [7:0] O;

    initial begin O = 8'dZ; end
    always @(posedge clock) begin
      if (write_enable && line_select) begin
          O <= Input;
      end
    end

    always @(negedge reset) begin
        O = 8'dZ;
    end
    assign Output = (read_enable && line_select) ? O : 8'dZ;

endmodule
```

Figure 7: 8-bit memory line module

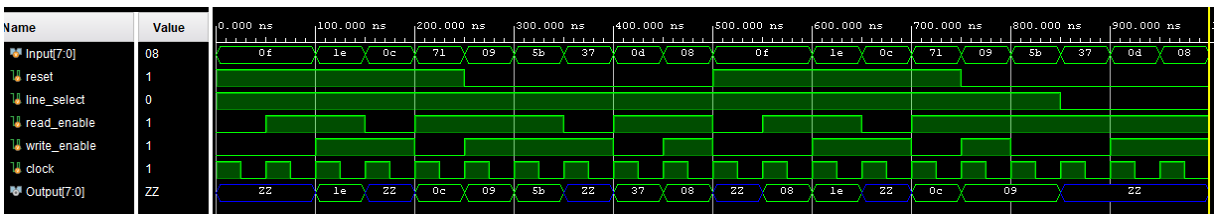


Figure 8: 8-bit memory line module simulation results

2.1.5 Part 4

In this part, we implemented an 8 byte memory module using 8-bit memory line module. 8 byte memory module should take 8-bit data as input and give 8-bit data as output. Also, the module should take 3-bit address, chip select, reset, read enable, write enable, and clock inputs for required operations. Required operations are below.

• Nth memory line is selected when the chip select input high. Here, N is address number.

• At the rising edge of the clock signal, the selected memory line stores the data, which is given as input, if the write enable input is high.

• The module clears all stored data in the memory lines at the falling edge of the reset signal.

• The output of the module is the data of the selected memory line, if read enable input is high.

```
/*          part4           */
module part4(
    input wire [7:0] Input,
    input wire [2:0] address,
    input wire chip_select,
    input wire reset,
    input wire read_enable,
    input wire write_enable,
    input wire clock,
    output wire [7:0] Output
    );
    wire [7:0] line_select;
    decoder_3input d(address, line_select);

    wire [7:0] temp0, temp1, temp2, temp3, temp4, temp5, temp6, temp7 ;

    part3 p0(Input, reset, chip_select && line_select[0], read_enable, write_enable, clock, temp0);
    part3 p1(Input, reset, chip_select && line_select[1], read_enable, write_enable, clock, temp1);
    part3 p2(Input, reset, chip_select && line_select[2], read_enable, write_enable, clock, temp2);
    part3 p3(Input, reset, chip_select && line_select[3], read_enable, write_enable, clock, temp3);
    part3 p4(Input, reset, chip_select && line_select[4], read_enable, write_enable, clock, temp4);
    part3 p5(Input, reset, chip_select && line_select[5], read_enable, write_enable, clock, temp5);
    part3 p6(Input, reset, chip_select && line_select[6], read_enable, write_enable, clock, temp6);
    part3 p7(Input, reset, chip_select && line_select[7], read_enable, write_enable, clock, temp7);

    buffer b0(temp0, line_select[0], Output);
    buffer b1(temp1, line_select[1], Output);
    buffer b2(temp2, line_select[2], Output);
    buffer b3(temp3, line_select[3], Output);
    buffer b4(temp4, line_select[4], Output);
    buffer b5(temp5, line_select[5], Output);
    buffer b6(temp6, line_select[6], Output);
    buffer b7(temp7, line_select[7], Output);

endmodule
```

Figure 9: 8 byte memory module using 8-bit memory line module

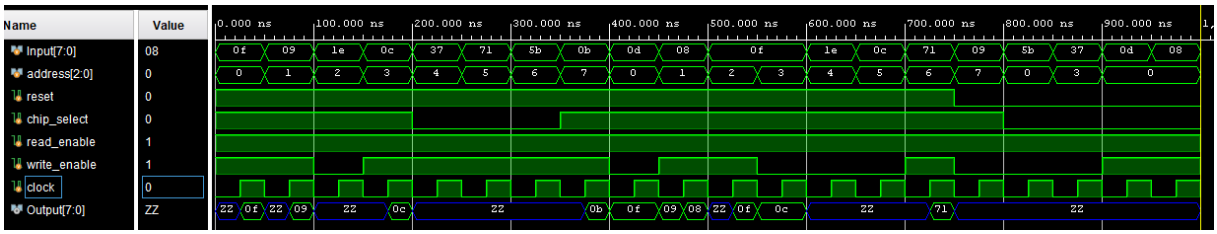


Figure 10: 8 byte memory module using 8-bit memory line module simulation results

2.1.6 Part 5

In this part, we implemented a 32 byte memory module using 8 byte memory module. 32 byte memory module took 8-bit data as input and gave 8-bit data as output. Also, this module took 5-bit address, reset, read enable, write enable, and clock inputs. 2 bits of the address input are used for chip selection and rest of 3 bits are used for selecting line. For instance, if the address input is 00111, zeroth chip (8-byte memory) and its 7th line (0-indexed) are selected. Our module performed the operations below.

• At the rising edge of the clock signal, the selected memory line stores the data value, which is given as input, if the write enable is high.

• The module clears the all stored data in the memory modules at the falling edge of the reset signal.

• If read enable is high, the output of the module is the stored data of the selected memory line.

```
/*          part5          */
module part5(
    input wire [7:0] Input,
    input wire [4:0] address,
    input wire chip_select,
    input wire reset,
    input wire read_enable,
    input wire write_enable,
    input wire clock,
    output wire [7:0] Output
    );
    wire [3:0] line_select;
    decoder_2input d(address[4:3], line_select);

    wire [7:0] temp0, temp1, temp2, temp3;

    part4 p0(Input, address[2:0], chip_select && line_select[0], reset, read_enable, write_enable, clock, temp0);
    part4 p1(Input, address[2:0], chip_select && line_select[1], reset, read_enable, write_enable, clock, temp1);
    part4 p2(Input, address[2:0], chip_select && line_select[2], reset, read_enable, write_enable, clock, temp2);
    part4 p3(Input, address[2:0], chip_select && line_select[3], reset, read_enable, write_enable, clock, temp3);

    buffer b0(temp0, line_select[0], Output);
    buffer b1(temp1, line_select[1], Output);
    buffer b2(temp2, line_select[2], Output);
    buffer b3(temp3, line_select[3], Output);

endmodule
```

Figure 11: 32 byte memory module using 8 byte memory module

Example Memory Simulation:

- Reset all lines
- Write 25 to Address 30
- Write 15 to Address 20
- Read Address 12
- Write 18 to Address 10
- Read Address 15
- Read Address 30
- Read Address 10

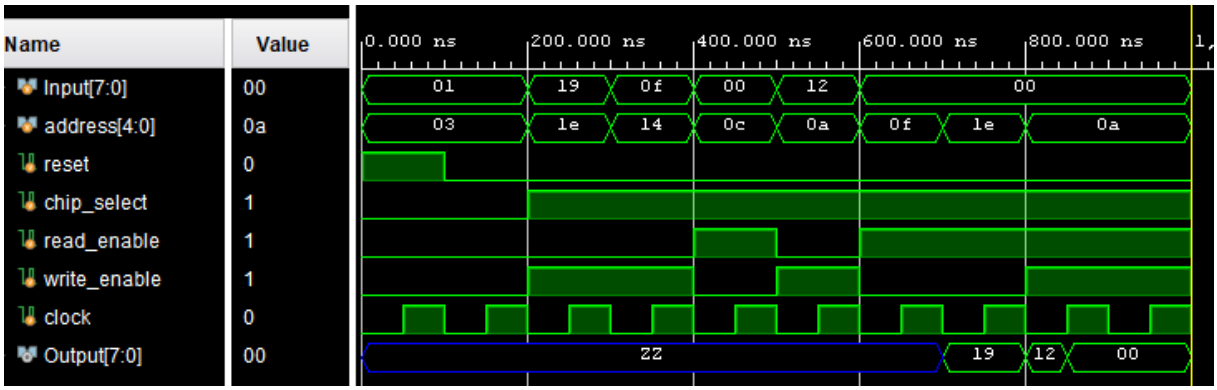


Figure 12: 32 byte memory module using 8 byte memory module simulation results

2.1.7 Part 6

In this part, we implemented a 128 byte memory module using 32 byte memory module. 128 byte memory module took 32-bit data as input and give 32-bit data as output. Also, this module took address, reset, read enable, write enable and clock inputs. Our module is able to perform the operations below.

• At the rising edge of the clock signal, the selected memory line stores the data value, which is given as input, if the write enable is high.

• The module clears the all stored data in the memory modules at the falling edge of the reset signal.

• If read enable is high, the output of the module is the stored data of the selected memory line.

By considering the given hint, 32 byte memory modules have 8-bit inputs and 8-bit outputs. We used concatenate operation to implement 128 byte memory.

This concluded the fact we stated above in the **Introduction** section, that nearly all the parts can be formed by concatenating the above smaller implementation into a larger version. Step by step, an 8 bit memory unit became a 128 byte memory module.

```
/*          part6          */
module part6(
    input wire [7:0] Input,
    input wire [6:0] address,
    input wire chip_select,
    input wire reset,
    input wire read_enable,
    input wire write_enable,
    input wire clock,
    output wire [7:0] Output
    );
    wire [3:0] line_select;
    decoder_2input d(address[6:5], line_select);

    wire [7:0] temp0, temp1, temp2, temp3;

    part5 p0(Input, address[4:0], chip_select && line_select[0], reset, read_enable, write_enable, clock, temp0);
    part5 p1(Input, address[4:0], chip_select && line_select[1], reset, read_enable, write_enable, clock, temp1);
    part5 p2(Input, address[4:0], chip_select && line_select[2], reset, read_enable, write_enable, clock, temp2);
    part5 p3(Input, address[4:0], chip_select && line_select[3], reset, read_enable, write_enable, clock, temp3);

    buffer b0(temp0, line_select[0], Output);
    buffer b1(temp1, line_select[1], Output);
    buffer b2(temp2, line_select[2], Output);
    buffer b3(temp3, line_select[3], Output);

endmodule
```

Figure 13: 128 byte memory module using 32 byte memory module

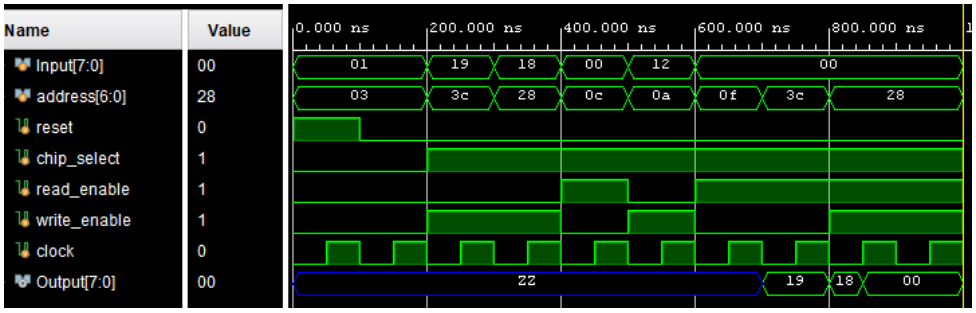


Figure 14: 128 byte memory module using 32 byte memory module simulation results

3 RESULTS [15 points]

You can see the simulation codes we used for Experiment parts and their simulation results here. To carry out each different implementation of the experiment, we arranged inputs of each simulation according to part it belongs. The simulation results we obtained after completing each step of the experiments resulted IN THE WAY we expected. This concludes the fact that our experiment is coherent for each individual part.

About part-1, see **Part 1** section above. In this part, we implemented an 8-bit bus which corresponds to the figure given to us in the pdf by using Verilog. We used 2 drivers with 3-state buffers. For simulation codes and results, see figures below:

```
/* part1     */
reg [7:0] Input1;
reg [7:0] Input2;
reg enable;
wire [7:0] Output;

part1 uut(Input1, Input2 , enable, Output);

initial begin
    Input1 = 8'd15;   Input2 = 8'd12;   enable = 0;   #100;
    Input1 = 8'd15;   Input2 = 8'd53;   enable = 1;   #100;
    Input1 = 8'd30;   Input2 = 8'd12;   enable = 0;   #100;
    Input1 = 8'd12;   Input2 = 8'd113;  enable = 1;   #100;
    Input1 = 8'd113;  Input2 = 8'd22;   enable = 1;   #100;
    Input1 = 8'd9;    Input2 = 8'd10;   enable = 0;   #100;
    Input1 = 8'd91;   Input2 = 8'd5;    enable = 1;   #100;
    Input1 = 8'd55;   Input2 = 8'd51;   enable = 0;   #100;
    Input1 = 8'd13;   Input2 = 8'd23;   enable = 1;   #100;
    Input1 = 8'd8;    Input2 = 8'd11;   enable = 0;   #100;
end
```

Figure 15: Simulation code of the circuit given in Part 1

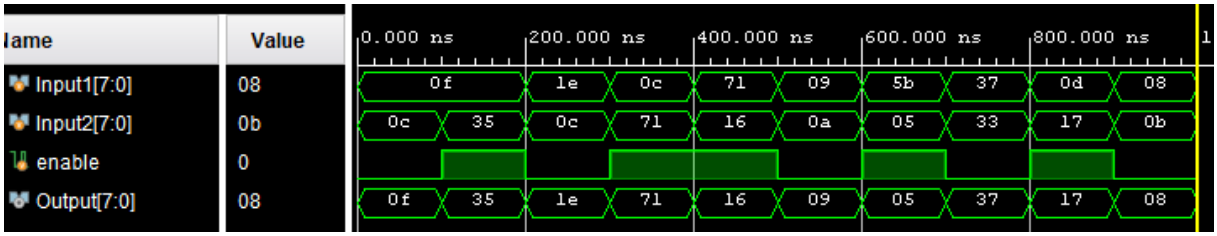


Figure 16: Simulation result of the circuit given in Part 1

About part-2, see **Part 2** section above. In this part, we implemented an 8-bit data bus with 2 drivers and 2 readers which corresponds to the figure given to us in the pdf by using Verilog. We used 3-state buffers for both drivers and readers. For simulation codes and results, see figures below:

```
/*    part2    */
reg [7:0] Input1;
reg [7:0] Input2;
reg enable;
wire [7:0] Output1;
wire [7:0] Output2;

part2 uut(Input1, Input2 , enable, Output1, Output2);

initial begin
    Input1 = 8'd15;   Input2 = 8'd12;   enable = 0;   #100;
    Input1 = 8'd15;   Input2 = 8'd53;   enable = 1;   #100;
    Input1 = 8'd30;   Input2 = 8'd12;   enable = 0;   #100;
    Input1 = 8'd12;   Input2 = 8'd113;  enable = 1;   #100;
    Input1 = 8'd113;  Input2 = 8'd22;   enable = 1;   #100;
    Input1 = 8'd9;    Input2 = 8'd10;   enable = 0;   #100;
    Input1 = 8'd91;   Input2 = 8'd5;    enable = 1;   #100;
    Input1 = 8'd55;   Input2 = 8'd51;   enable = 0;   #100;
    Input1 = 8'd13;   Input2 = 8'd23;   enable = 1;   #100;
    Input1 = 8'd8;    Input2 = 8'd11;   enable = 0;   #100;
end
```

Figure 17: Simulation code of the circuit given in Part 2

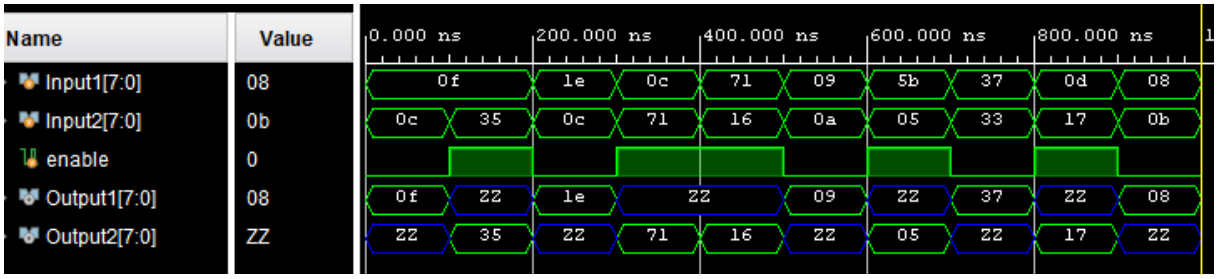


Figure 18: Simulation result of the circuit given in Part 2

About part-3, see **Part 3** section above. In this part, we implemented an 8-bit memory line module. This module took 8-bit data as input and give 8-bit data as output. Also, the module took reset, line select, read enable, write enable, and clock inputs for required operations. For simulation codes and results, see figures below:

```
/*    part3   */
reg [7:0] Input;
reg reset,line_select, read_enable, write_enable, clock;
wire [7:0] Output;

part3 uut(Input, reset, line_select, read_enable, write_enable, clock, Output);

initial begin
    clock = 0; reset = 0;
    Input = 8'd15;   read_enable = 0;   write_enable = 0;   line_select = 1;  #50;
    Input = 8'd15;   read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd30;   read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd12;   read_enable = 0;   write_enable = 1;   #50;
    Input = 8'd113;  read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd9;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd91;   read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd55;   read_enable = 0;   write_enable = 1;   #50;
    Input = 8'd13;   read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd8;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd15;   read_enable = 0;   write_enable = 0;   #50;
    Input = 8'd15;   read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd30;   read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd12;   read_enable = 0;   write_enable = 1;   #50;
    Input = 8'd113;  read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd9;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd91;   read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd55;   read_enable = 1;   write_enable = 0;   line_select = 0;  #50;
    Input = 8'd13;   read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd8;    read_enable = 1;   write_enable = 1;   #50;
end
always begin
    clock = ~clock; #25;
end
always begin
    reset = ~reset; #250;
end
```

Figure 19: Simulation code Part 3

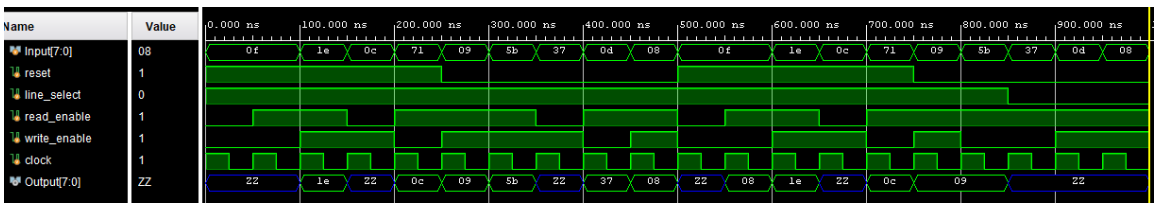


Figure 20: Simulation result of Part 3

About part-4, see **Part 4** section above. In this part, we implemented an 8 byte memory module using 8-bit memory line module. 8 byte memory module should take 8-bit data as input and give 8-bit data as output. Also, the module should take 3-bit address, chip select, reset, read enable, write enable, and clock inputs for required operations. For simulation codes and results, see figures below:

```
/*  part4   */
reg [7:0] Input;
reg [2:0] address;
reg reset,chip_select, read_enable, write_enable, clock;
wire [7:0] Output;
part4 uut(Input, address, chip_select, reset, read_enable, write_enable, clock, Output);

initial begin
    clock = 1; reset = 0;
    Input = 8'd15;   address = 3'd0;    read_enable = 1;   write_enable = 1;   chip_select = 1;  #50;
    Input = 8'd9;    address = 3'd1;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd30;   address = 3'd2;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd12;   address = 3'd3;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd55;   address = 3'd4;    read_enable = 1;   write_enable = 1;   chip_select = 0;  #50;
    Input = 8'd113;  address = 3'd5;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd91;   address = 3'd6;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd11;   address = 3'd7;    read_enable = 1;   write_enable = 1;   chip_select = 1;  #50;
    Input = 8'd13;   address = 3'd0;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd8;    address = 3'd1;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd15;   address = 3'd2;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd15;   address = 3'd3;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd30;   address = 3'd4;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd12;   address = 3'd5;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd113;  address = 3'd6;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd9;    address = 3'd7;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd91;   address = 3'd0;    read_enable = 1;   write_enable = 0;   chip_select = 0;  #50;
    Input = 8'd55;   address = 3'd3;    read_enable = 1;   write_enable = 0;   #50;
    Input = 8'd13;   address = 3'd0;    read_enable = 1;   write_enable = 1;   #50;
    Input = 8'd8;    address = 3'd0;    read_enable = 1;   write_enable = 1;   #50;

end
always begin
    clock = ~clock; #25;
end
always begin
    reset = ~reset; #750;
end
```

Figure 21: Simulation code Part 4

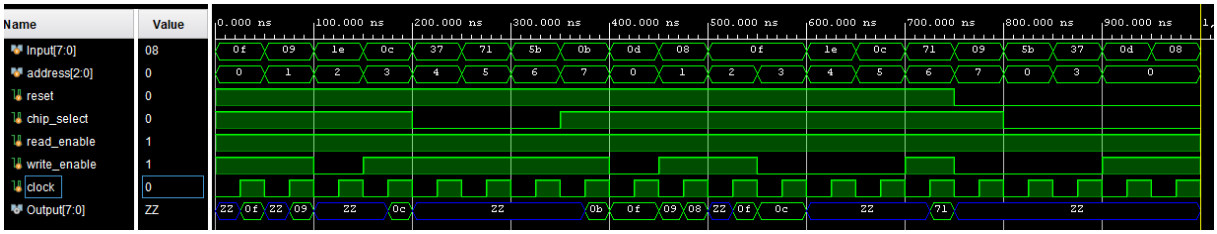


Figure 22: Simulation result of Part 4

About part-5, see **Part 5** section above. In this part, we implemented a 32 byte memory module using 8 byte memory module. 32 byte memory module took 8-bit data as input and gave 8-bit data as output. Also, this module took 5-bit address, reset, read enable, write enable, and clock inputs. 2 bits of the address input are used for chip selection and rest of 3 bits are used for selecting line.

For simulation codes and results, see figures below:

```
/*  part5   */
reg [7:0] Input;
reg [4:0] address;
reg reset,chip_select, read_enable, write_enable, clock;
wire [7:0] Output;
part5 uut(Input, address, chip_select, reset, read_enable, write_enable, clock, Output);

initial begin
    Input = 8'd1;   address = 5'd3;    read_enable = 0;   write_enable = 0;   chip_select = 0;
    clock = 1; reset = 1; #100;
    reset = 0; #100;
    Input = 8'd25;    address = 5'd30;    read_enable = 0;   write_enable = 1;   chip_select = 1;  #100;
    Input = 8'd15;    address = 5'd20;    read_enable = 0;   write_enable = 1;   #100;
    Input = 8'd0;     address = 5'd12;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd18;    address = 5'd10;    read_enable = 0;   write_enable = 1;   #100;
    Input = 8'd0;     address = 5'd15;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd0;     address = 5'd30;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd0;     address = 5'd10;    read_enable = 1;   write_enable = 1;   #100;
end
always begin
    clock = ~clock; #50;
end
```

Figure 23: Simulation code Part 5

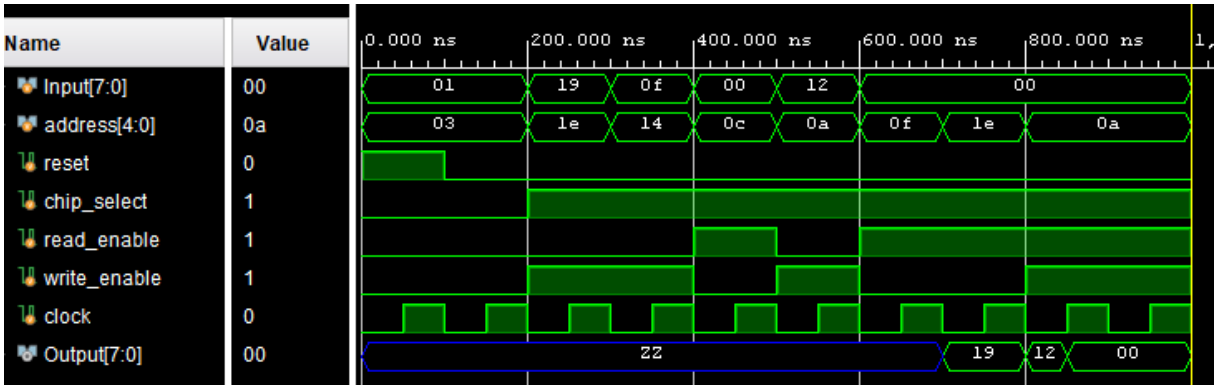


Figure 24: Simulation result of Part 5

About part-6, see **Part 6** section above. In this part, By considering the given hint, 32 byte memory modules have 8-bit inputs and 8-bit outputs. We used concatenate operation to implement 128 byte memory. We implemented a 128 byte memory module using 32 byte memory module. 128 byte memory module took 32-bit data as input and give 32-bit data as output. Also, this module took address, reset, read enable, write enable and clock inputs. Our module is able to perform the given operations:

• At the rising edge of the clock signal, the selected memory line stores the data value,which is given as input, if the write enable is high.

• The module clears the all stored data in the memory modules at the falling edge of the reset signal.

• If read enable is high, the output of the module is the stored data of the selected memory line. For simulation codes and results, see figures below:

```
/*  part6   */
reg [7:0] Input;
reg [6:0] address;
reg reset,chip_select, read_enable, write_enable, clock;
wire [7:0] Output;
part6 uut(Input, address, chip_select, reset, read_enable, write_enable, clock, Output);

initial begin
    Input = 8'd1;   address = 5'd3;    read_enable = 0;   write_enable = 0;   chip_select = 0;
    clock = 1; reset = 1; #100;
    reset = 0; #100;
    Input = 8'd25;    address = 7'd60;    read_enable = 0;   write_enable = 1;   chip_select = 1;  #100;
    Input = 8'd24;    address = 7'd40;    read_enable = 0;   write_enable = 1;   #100;
    Input = 8'd0;     address = 7'd12;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd18;    address = 7'd10;    read_enable = 0;   write_enable = 1;   #100;
    Input = 8'd0;     address = 7'd15;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd0;     address = 7'd60;    read_enable = 1;   write_enable = 0;   #100;
    Input = 8'd0;     address = 7'd40;    read_enable = 1;   write_enable = 1;   #100;
end
always begin
    clock = ~clock; #50;
end
```

Figure 25: Simulation code Part 6

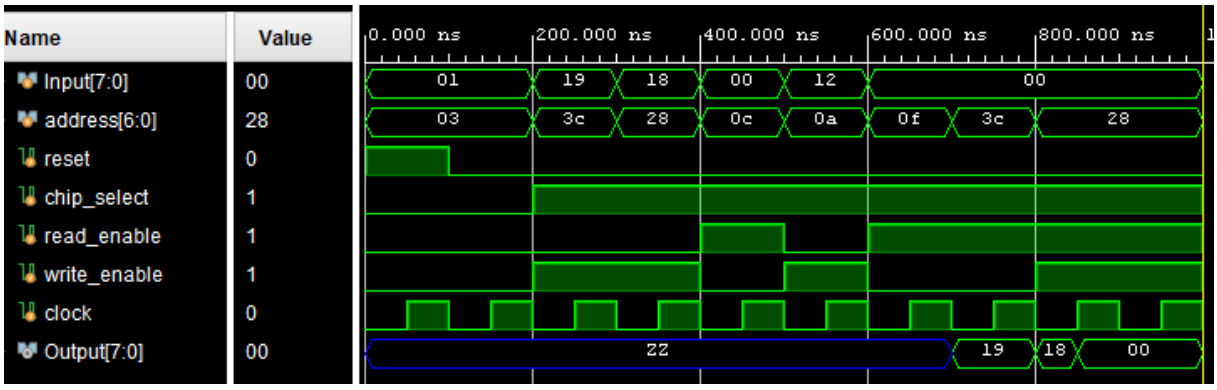


Figure 26: Simulation result of Part 6

4 DISCUSSION [25 points]

Results session above concluded the fact we stated in the **Introduction** section, that nearly all the parts are formed by concatenating the above part's smaller implementation into a larger version. Step by step, an 8 bit memory unit became a 128 byte memory module in this Experiment of Week 6.

In **Part - 1** and **Part - 2** sections we implemented the circuits given to us by using Verilog. Since the circuits are provided in the homework pdf, it was pretty clear about what we are asked to do.

But then, when we arrived to **Part - 3**, it was not really clear what we are asked to do. We are given some commands, but the point was not actually clear, and since the spesific requirements are not self-explanatory, we need to figure them out by looking at the following parts and trying to come up with a unifying approach. But since the commands are not really looking-forward, when we tried to implement the following parts, we see that the commands were just enough to form Part - 3 up, but not adjust it to the following parts. So we wrote Part - 3 like a billion times, and all implementations differently. But eventually and happily, one of the implementations worked. So we deleted the others.

For example, in **Part - 4** we have functions. Each function requires a **reg** in them, but we couldn't know the fact that they all require different **regs** in them. So at first, we thought that there is **ONE** main **reg**, and all of them uses thaat reg one after each other, one by one, like a line formed. We thought that the implementation we're asked to do was each of the functions writing onto that main reg one-by-one. But seems like that's not the case. The true version of the implementation is that each function will call a reg inside them, and write onto them by changing them. We didn't know that at first so we lost hours trying to figure it out. And the Verilog introduction given to us did not consisted that required information, and since Verilog is an old language, documentation on the internet was not nearly good enough to solve this issue. Our approach ended up like Einstein's 1000 different trial of finding the light bulb, trying each different way we came up with, one by one, and it was a tiring experience. (Althoug I know that it was fake news and Tesla was the actual genius there, I couldn't come up with another fitting example here)

We managed to succeed, but we would really appreciate a clearer explanation about this experiment and results.

Another problem we faced with while trying to do **Part - 3** and **Part - 4** was the buffer need. Seems like we should assign the outputs to each of the functions' results, using BUFFER, but figuring out that we should use it took a lot of time.

In addition, while trying to do **Part - 3**, we noticed that when we initialised a reg

at first, it gives a value 0. But it shouldn't, and instead, it should give a value Z (high impedance). So its wrong. Or again, in the same way, while we are going to reset the register in part-3, we were assigning it to 0, but instead, we should have them be high impedance.

After completing them, for **Part - 5** and **Part - 6** there's not much left to do, since all the remaining parts were like the domino tiles starting from Part - 4, coming one after another, so completing part-4 in a clear and true way ended up completing the experiment. Just there was an extra in **Part - 5**, a DECODER with 2 inputs, but that's all. Other than that, they were just copy-pastes of each other.

Another difficulty we faced in all the parts except for part-5 was that we are NOT given any proper example to illustrate the corresponding part's design clearly. So it was just pure difficult to implement the simulation codes (see **Results** part above for simulation codes) And even in the part-5, it was a try-and-fail approach to implement them, so it would be really better if we're given a little more explanatory examples and edge cases, maybe.

5 CONCLUSION [10 points]

In Part 1 and Part 2; we analyzed the given circuits and implemented data buses with buffers, readers and drivers. During these parts of the experiment, we did not have any problems because it was clear what we should do.

In Part 3, we did not have a problem in itself, but we had difficulty in associating it with Part 4. We learned how to implement an 8-bit memory line module in Part 3; then by using this, we implemented an 8-byte memory module in Part 4. We realized that we lost time because we did not know some important details related to Verilog and also we did not find these details in the first lecture of the semester. We think the lesson in which Verilog usage is taught should be more comprehensive. Also, given examples for simulation parts were not enough.

Then, we adapted the methods we used in Part 4 into Part 5 and Part 6. We implemented 32-byte and 128-byte memory modules by using 8-byte and 32-byte memory modules, respectively. As we found the correct method before, we did not face any difficulties during these parts.

In conclusion, we had difficulties figuring out what to do exactly, especially in Part 3 and Part 4. But we tried different methods, then managed to find the method that works properly.