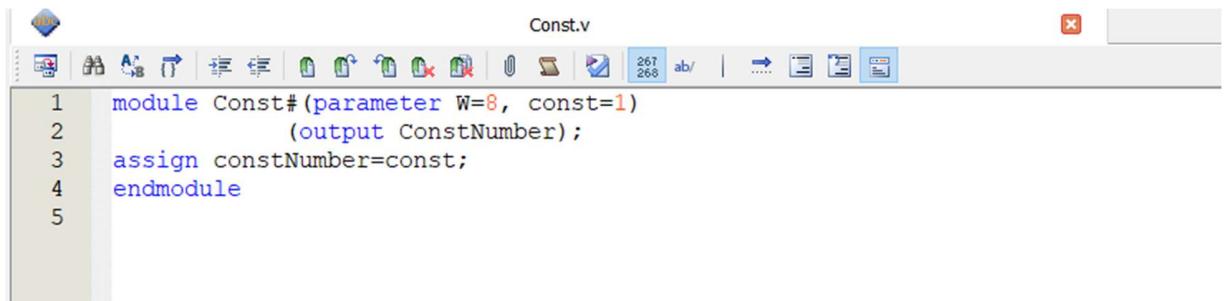


Berkay İPEK

EE446 -2304814

Laboratory Work 1 - Warming Up for Computer Design

1.2.1 Constant Value Generator

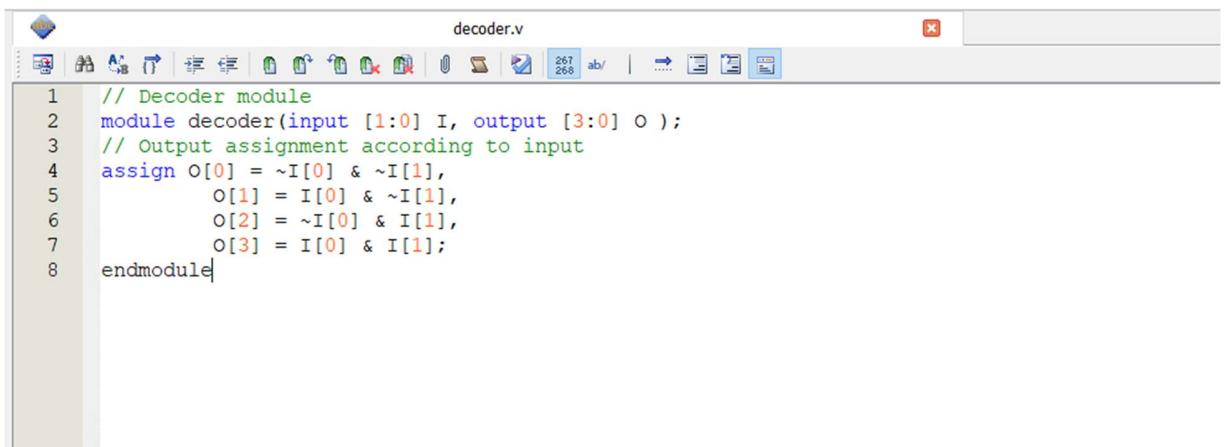


```
Const.v
1 module Const#(parameter W=8, const=1)
2     (output ConstNumber);
3     assign constNumber=const;
4 endmodule
5
```

Figure 1: Constant Value Generator with parametric of W and const

1.2.2 Decoder

1. Implement a 2 to 4 decoder



```
decoder.v
1 // Decoder module
2 module decoder(input [1:0] I, output [3:0] O );
3     // Output assignment according to input
4     assign O[0] = ~I[0] & ~I[1],
5         O[1] = I[0] & ~I[1],
6         O[2] = ~I[0] & I[1],
7         O[3] = I[0] & I[1];
8 endmodule
```

Figure 2: 2x4 Decoder in verilog

2. Write a test bench module to test your implementation

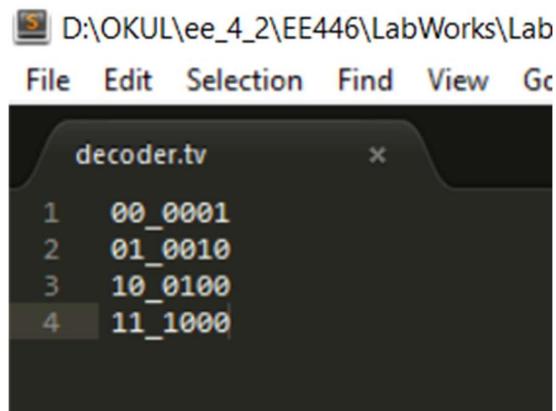
```
// To read vectors
initial
begin
    clk_tb=0;
    $readmem("decoder.tv",testvectors);
    vectornum = 0;
    errors = 0;
end

// Test Vectors are read and operated at pos edges
always @(posedge clk_tb)
begin
    #1;
    {Itb,Oexp}=testvectors[vectornum];
end

// Test Vector Output and Expect are compared at neg edges
always @(negedge clk_tb)
begin
    if(Otb != Oexp)
        begin
            $display("Error: input = %b", {Itb});
            $display(" outputs = %b (%b expected)",Otb,Oexp);
            errors= errors +1;
        end
    vectornum = vectornum +1;
    if(testvectors[vectornum] == 4'bX)
        begin
            $display("%d tests completed with %d errors",vectornum, errors);
            $stop;
        end
    end
end
endmodule
```

Figure 3: 2x4 Decoder Testbench Code in verilog

3. Provide a vector table to be used by your test bench module to test your implementation.



The screenshot shows a text editor window titled "decoder.tv". The menu bar includes "File", "Edit", "Selection", "Find", "View", and "Go". The main content area displays the following test vectors:

```
1 00_0001
2 01_0010
3 10_0100
4 11_1000
```

Figure 4: 2x4 Decoder Test Vector file

4. Verify that your implementation is correct.

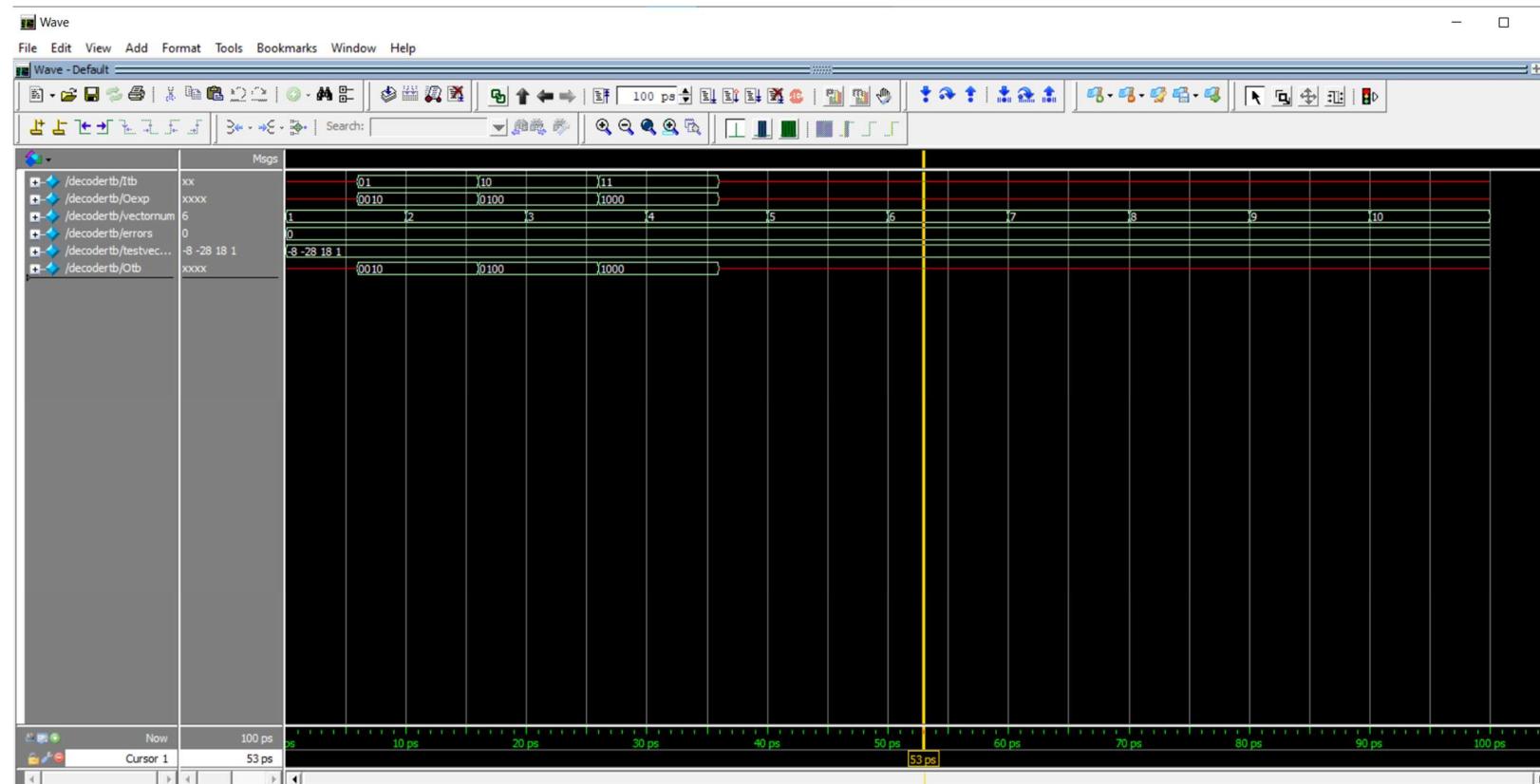
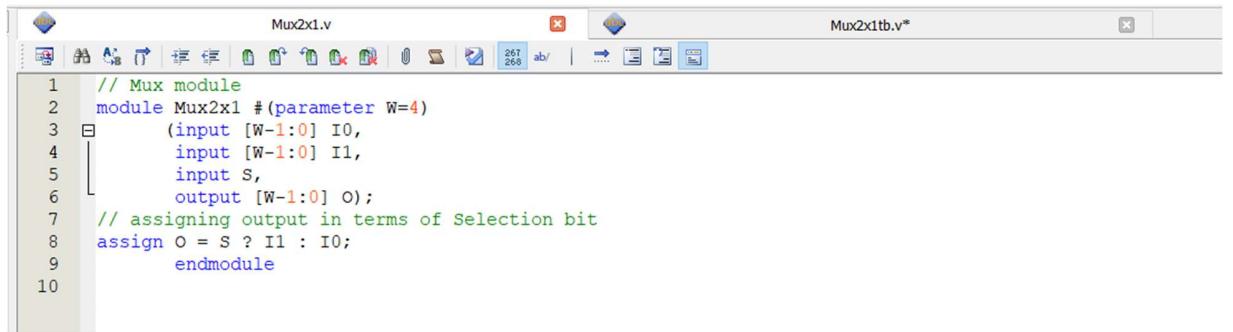


Figure 5: 2x4 Decoder Test Vector Result

As Figure 5 shows, error number is zero during the simulation of test vector, which means my implementation is correct.

1.2.3 Multiplexers

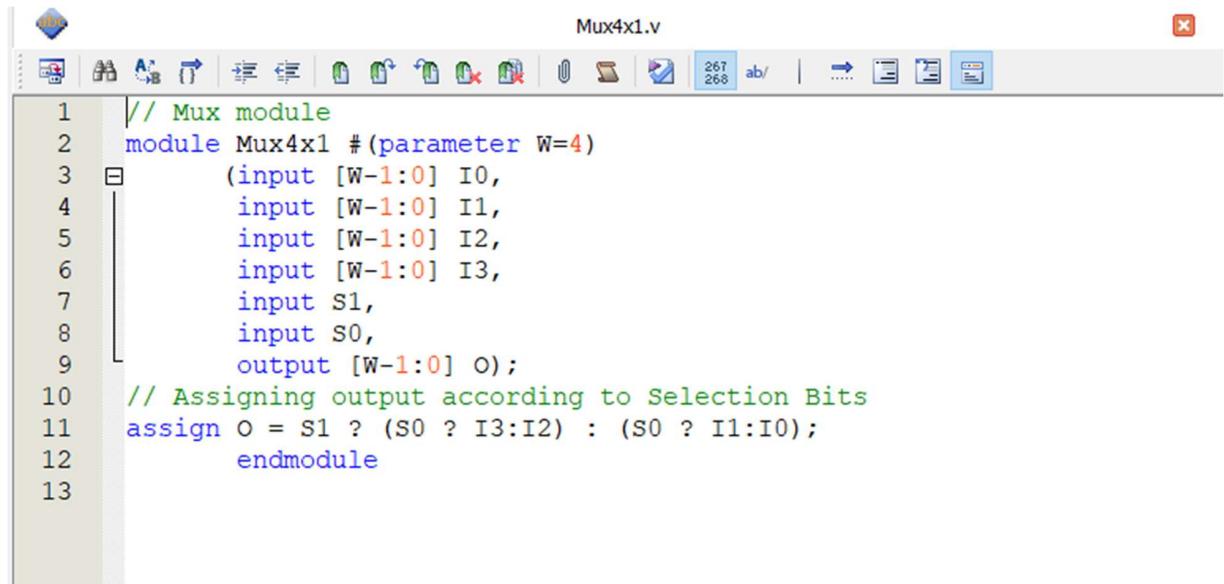
1. Implement a W-bit 2 to 1 and a W-bit 4 to 1 multiplexers, where W is a parameter specifying the data width of the input.



The screenshot shows a Verilog code editor window titled "Mux2x1.v". The code defines a module "Mux2x1" with a parameter "W=4". It has three inputs: I0, I1, and S, and one output O. The output is assigned based on the selection bit S: if S is 1, then O = I1; otherwise, O = I0. The code is as follows:

```
1 // Mux module
2 module Mux2x1 #(parameter W=4)
3   (input [W-1:0] I0,
4    input [W-1:0] I1,
5    input S,
6    output [W-1:0] O);
7   // assigning output in terms of Selection bit
8   assign O = S ? I1 : I0;
9   endmodule
10
```

Figure 6: 2x1 Mux in Verilog



The screenshot shows a Verilog code editor window titled "Mux4x1.v". The code defines a module "Mux4x1" with a parameter "W=4". It has four inputs: I0, I1, I2, I3, and two selection bits S1 and S0, and one output O. The output is assigned based on the selection bits: if S1 is 1, then O = S0 ? I3:I2; otherwise, O = S0 ? I1:I0. The code is as follows:

```
1 // Mux module
2 module Mux4x1 #(parameter W=4)
3   (input [W-1:0] I0,
4    input [W-1:0] I1,
5    input [W-1:0] I2,
6    input [W-1:0] I3,
7    input S1,
8    input S0,
9    output [W-1:0] O);
10  // Assigning output according to Selection Bits
11  assign O = S1 ? (S0 ? I3:I2) : (S0 ? I1:I0);
12  endmodule
13
```

Figure 7: 4x1 Mux in Verilog

2. Write test bench modules to test your implementations

```
22 initial
23 begin
24     clktb=1;
25     $readmem("mux4x1tb.tv",testvectors);
26     vectornum = 0;
27     errors = 0;
28 end
29
30 // Test Vectors are read and operated at pos edges
31 always @(posedge clktb)
32 begin
33     #1;
34     {I0tb,I1tb,I2tb,I3tb,S1tb,S0tb,Oexp}=testvectors[vectornum];
35 end
36 // Test Vector Output and Expect are compared at neg edges
37 always @(negedge clktb)
38 begin
39     if(Otb != Oexp)
40     begin
41         $display("Error: input = %b %b", S0tb, S1tb);
42         $display(" outputs = %b (%b expected)",Otb,Oexp);
43         errors= errors +1;
44     end
45     vectornum = vectornum +1;
46     if(testvectors[vectornum] == 4'bx)
47     begin
48         $display("%d tests completed with %d errors",vectornum, errors);
49         $stop;
50     end
51 end
52 endmodule
```

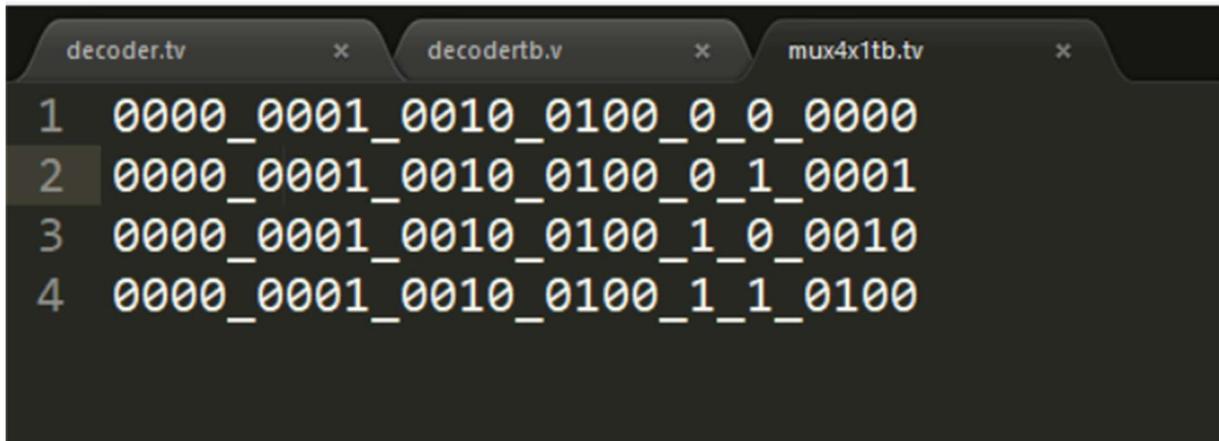
Figure 8: 4x1 MUX Testbench Code in Verilog

```
// To read vectors
initial
begin
    clk_tb=1;
    $readmem("mux2x1tb.tv", testvectors);
    vectornum = 0;
    errors = 0;
end

// Test Vectors are read and operated at pos edges
always @(posedge clk_tb)
begin
    #1;
    {I0_tb,I1_tb,S0_tb,Oexp}=testvectors[vectornum];
end
// Test Vector Output and Expect are compared at neg edges
always @(negedge clk_tb)
begin
    if(Otb != Oexp)
    begin
        $display("Error: input = %b ", S0_tb);
        $display(" outputs = %b (%b expected)", Otb, Oexp);
        errors= errors +1;
    end
    vectornum = vectornum +1;
    if(testvectors[vectornum] == 4'bX)
    begin
        $display("%d tests completed with %d errors",vectornum, errors);
        $stop;
    end
end
endmodule
```

Figure 9: 2x1 MUX Testbench Code in Verilog

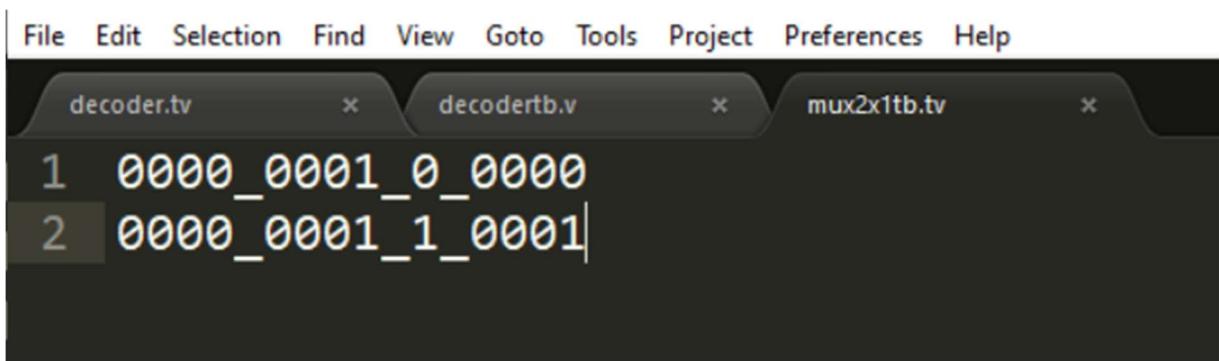
3. Provide vector tables to be used by your test bench modules to test your implementations.



A screenshot of a terminal window with three tabs: "decoder.tv", "decodertb.v", and "mux4x1tb.tv". The "mux4x1tb.tv" tab is active, displaying the following test vectors:

```
1 0000_0001_0010_0100_0_0_0000
2 0000_0001_0010_0100_0_1_0001
3 0000_0001_0010_0100_1_0_0010
4 0000_0001_0010_0100_1_1_0100
```

Figure 10: 4x1 MUX Test Vectors



A screenshot of a terminal window with three tabs: "decoder.tv", "decodertb.v", and "mux2x1tb.tv". The "mux2x1tb.tv" tab is active, displaying the following test vectors:

```
1 0000_0001_0_0000
2 0000_0001_1_0001
```

Figure 11: 2x1 MUX Test Vectors

4. Verify that your implementations are correct.

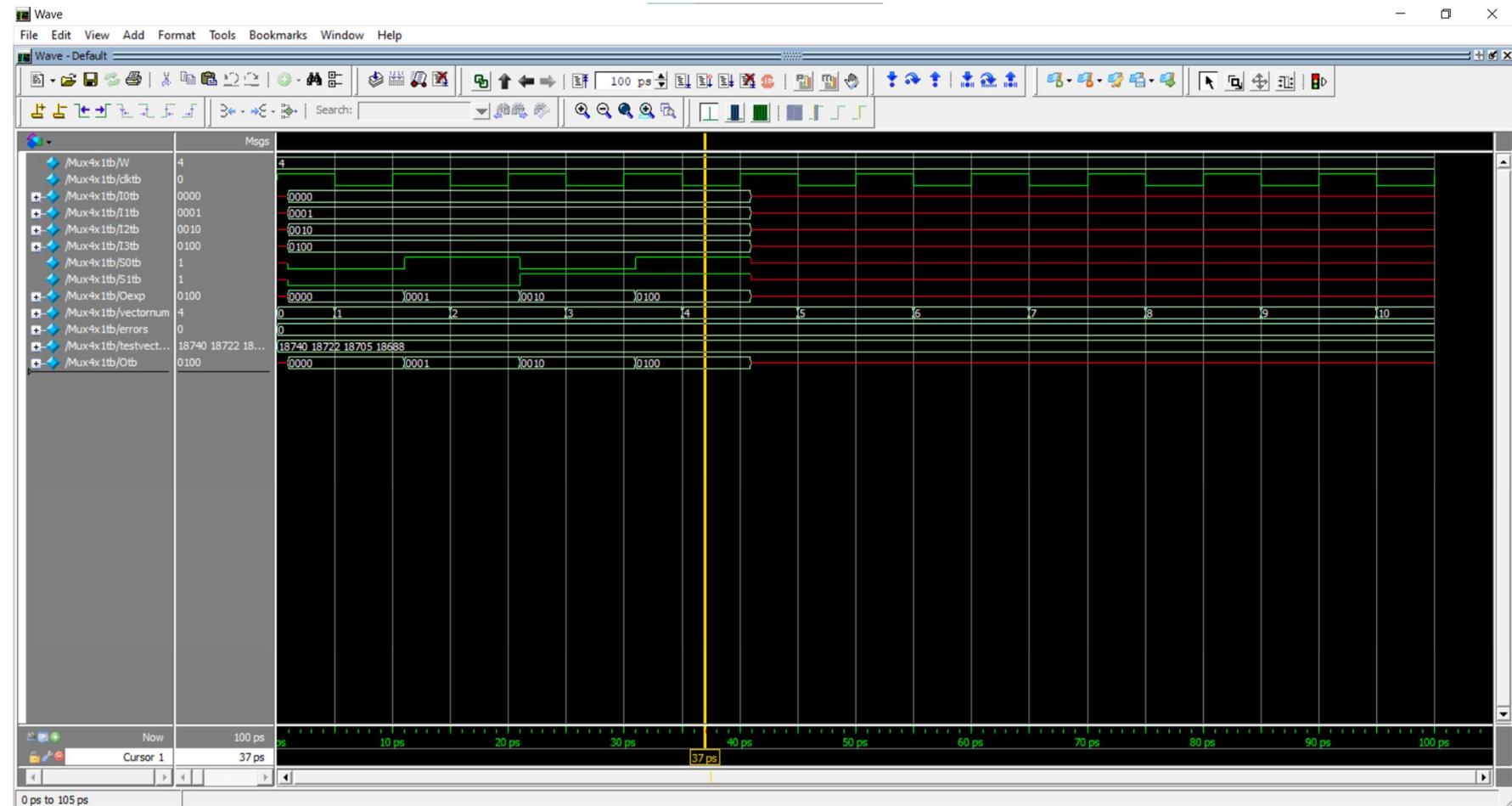


Figure 12: 4x1 MUX Test Vector Result

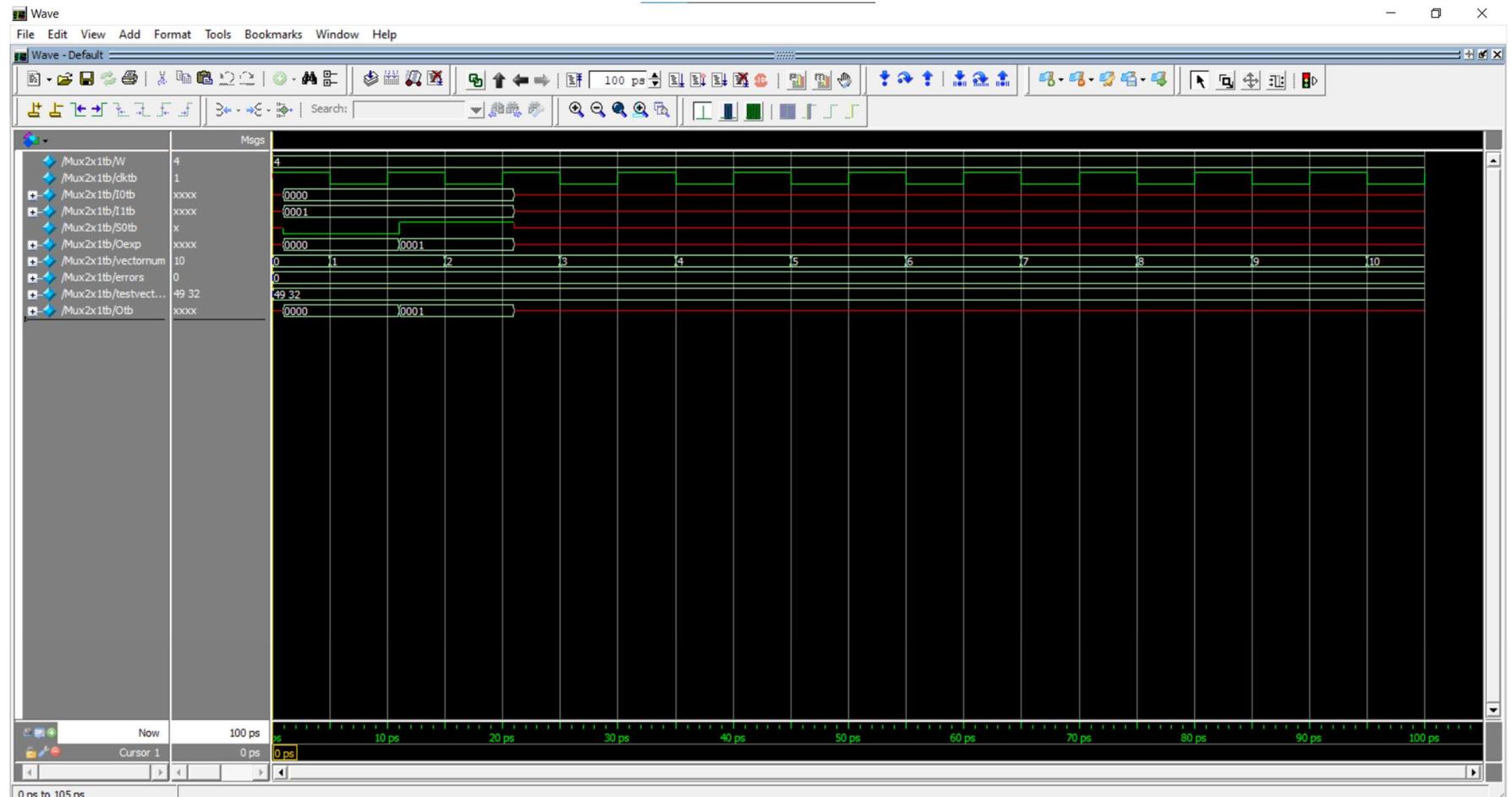


Figure 13: 2x1 MUX Test Vector Result

As Figures 12 and 13 show, error number is zero during the simulation of test vector, which means my implementation is correct

1.2.4 Arithmetic Logic Unit (ALU)

1. Implement a W-bit ALU for 2's complement arithmetic, where W is a parameter specifying the data width of its inputs. The ALU should be equipped with 8 operations controlled by 3 control signals. In addition to the N-bit result output, the ALU should have 4 other status output bits: Carry out (CO), overflow (OVF), negative (N) and zero (Z). Negative and zero bits are affected by all the ALU operations; whereas carry out and overflow can only be affected by arithmetic operations. The specifications of the ALU operations and the ALU status outputs are provided in Table 1 and Table 2, respectively.

```
1 // ALU module
2 module ALU #(parameter W=4)
3   (input [W-1:0] A,
4   input [W-1:0] B,
5   input [2:0] ALUcntrl,
6   output reg [W-1:0] O,
7   output reg CO,
8   output reg OVF,
9   output reg N,
10  output reg Z);
11 // Since switch case block can be implemented only in always block
12 // I created always block with default sensitivity list
13 always @(*)
14 begin
15   OVF=0;// Assign default parameters
16   CO=0;// These parameters are meaningful in only addition and subtraction
17   // Doing operations according to value of control signals
18   case(ALUcntrl)
19     // Addition
20     3'b000:
21       begin
22         (CO,O) = A+B;
23         Z = (O==0) ? 1:0;
24         N=O[W-1];
25         // Overflow occurs when A and B are positive however result is negative
26         if (~A[W-1] & ~B[W-1])
27           begin
28             if(O[W-1])
29               OVF=1;
30           end
31         // Overflow occurs when A and B are negative however result is positive
32         if (A[W-1] & B[W-1])
33           begin
34             if(~O[W-1])
35               OVF=1;
36           end
37       end
38     // Subtraction
39     3'b001:
40       begin
41         (CO,O) = A-B;
42         Z = (O==0) ? 1:0;
43         // Overflow occurs when A is Negative and B is positive however result is positive
44         if (A[W-1] & ~B[W-1])
45           begin
46             if(~O[W-1])
47               OVF=1;
48           end
49         // Overflow occurs when A is Positive and B is negative however result is negative
50         if (~A[W-1] & B[W-1])
51           begin
52             if(O[W-1])
53               OVF=1;
54           end
55         N=O[W-1];
56       end
57     // Subtraction
58     3'b010:
59       begin
60         (CO,O) = B-A;
61         Z = (O==0) ? 1:0;
62         // Overflow occurs when A is positive and B is negative however result is positive
63         if (~A[W-1] & B[W-1])
64           begin
65             if(~O[W-1])
66               OVF=1;
67           end
68       end
69   endcase
70 end
```

```

67         end
68     // Overflow occurs when A is negative and B is positive however result is nega
69     if (A[W-1] & ~B[W-1])
70     begin
71         if(O[W-1])
72             OVF=1;
73         end
74     N=O[W-1];
75     end
76 //Bit Clear
77 3'b011:
78 begin
79     O = A & ~B;
80     Z = (O==0) ? 1:0;
81     N=O[W-1];
82     end
83 //AND operator
84 3'b100:
85 begin
86     O = A & B;
87     Z = (O==0) ? 1:0;
88     N=O[W-1];
89     end
90 //OR operator
91 3'b101:
92 begin
93     O = A | B;
94     Z = (O==0) ? 1:0;
95     N=O[W-1];
96     end
97 //XOR operator
98 3'b110:
99 begin
100    O = A ^ B;
101    Z = (O==0) ? 1:0;
102    N=O[W-1];
103    end
104 //XNOR operator
105 3'b111:
106 begin
107    O = ~(A ^ B);
108    Z = (O==0) ? 1:0;
109    N=O[W-1];
110    end
111 endcase
112 end
113 endmodule

```

Figure 14: ALU in Verilog

2. Explain your method to detect overflow

In addition, and subtraction operation, there can be overflow. Otherwise, all operations expect them must give OVF as 0 always. Therefore, I checked it in only 000, 001, and 010 ALU controls. To detect OVF, I just checked the sign bits of operators and result. If two negative numbers are added, then result should be negative also. It can not be positive. I simply checked it. The same procedure is applied in subtraction, as explained in detail in figure 14.

3. Write a test bench module to test your implementation

```
1  module ALUtb#(parameter N=4);
2
3  reg clk;
4  reg [3:0] Atb;
5  reg [3:0] Btb;
6  reg [2:0] ALUcntrltb;
7  reg [3:0] Oexp;
8  reg COexp;
9  reg OVExp;
10 reg Nexp;
11 reg Zexp;
12 reg [31:0] vectornum,errors;
13 reg [18:0] testvectors[9:0];
14 wire [3:0] Otb;
15 wire COTb;
16 wire OVFTb;
17 wire Ntb;
18 wire Ztb;
19
20 ALU dut(.A(Atb),.B(Btb),.ALUcntrl(ALUcntrltb),
21           .O(Otb),.CO(COTb),.OVF(OVFTb),.N(Ntb),.Z(Ztb));
22 // To generate clock cycles
23 always
24 begin
25   clk=-clk;
26   #5;
27 end
28
29 // To read vectors
30 initial
31 begin
32   clk=1;
33   $readmem("ALUtb.tv",testvectors);
34   vectornum = 0;
35   errors = 0;
36 end
37 // Test Vectors are read and operated at pos edges
38 always @(posedge clk)
39 begin
40   #1;
41   (Atb,Btb,ALUcntrltb,Oexp,COexp,OVExp,Nexp,Zexp)=testvectors[vectornum];
42 end
43 // Test Vector Output and Expect are compared at neg edges
44 always @(negedge clk)
45 begin
46   if(Otb != Oexp)
47     begin
48       $display("Error: input = %b %b", Atb, Btb);
49       $display("O outputs = %b (%b expected)",Otbo,Oexp);
50       errors= errors +1;
51     end
52   if(COTb != COexp)
53     begin
54       $display("Error: input = %b %b", Atb, Btb);
55       $display("CO outputs = %b (%b expected)",COTb,COexp);
56       errors= errors +1;
57     end
58   if(OVFTb != OVExp)
59     begin
60       $display("Error: input = %b %b", Atb, Btb);
61       $display("OVF outputs = %b (%b expected)",OVFTb,OVExp);
62       errors= errors +1;
63     end
64   if(Ntb != Nexp)
65     begin
66       $display("Error: input = %b %b", Atb, Btb);
```

Figure 15: ALU Test Bench in Verilog

4. (If using ModelSim) Provide a vector table to be used by your test bench module to test your implementation. For this step, explicitly state to what operation a vector corresponds. In other words, if you are testing addition of the two numbers that will cause overflow, explicitly state which vector in the vector table is dedicated to that test.

	A	B	$ALU O_{C0}_O_{N2}$	
1	0000_0000_000_0000_0_0_0_1			$O+P$
2	0111_0111_000_1110_0_1_1_0			$7+7$
3	1110_1110_000_1100_1_0_1_0			$-2 + -2$
4	0111_0111_001_0000_0_0_0_1			$7 - 7$
5	1101_0110_001_0111_0_1_0_0			$-3 - 6$
6	1111_1111_011_0000_0_0_0_1			BC
7	1111_0001_100_0001_0_0_0_0			AND
8	0000_0001_101_0001_0_0_0_0			OR
9	1111_0000_110_1111_0_0_1_0			XOR
10	1111_0000_111_0000_0_0_0_1			XNOR

Figure 16: ALU Test Vector with explanation

6. Verify that your implementation is correct.

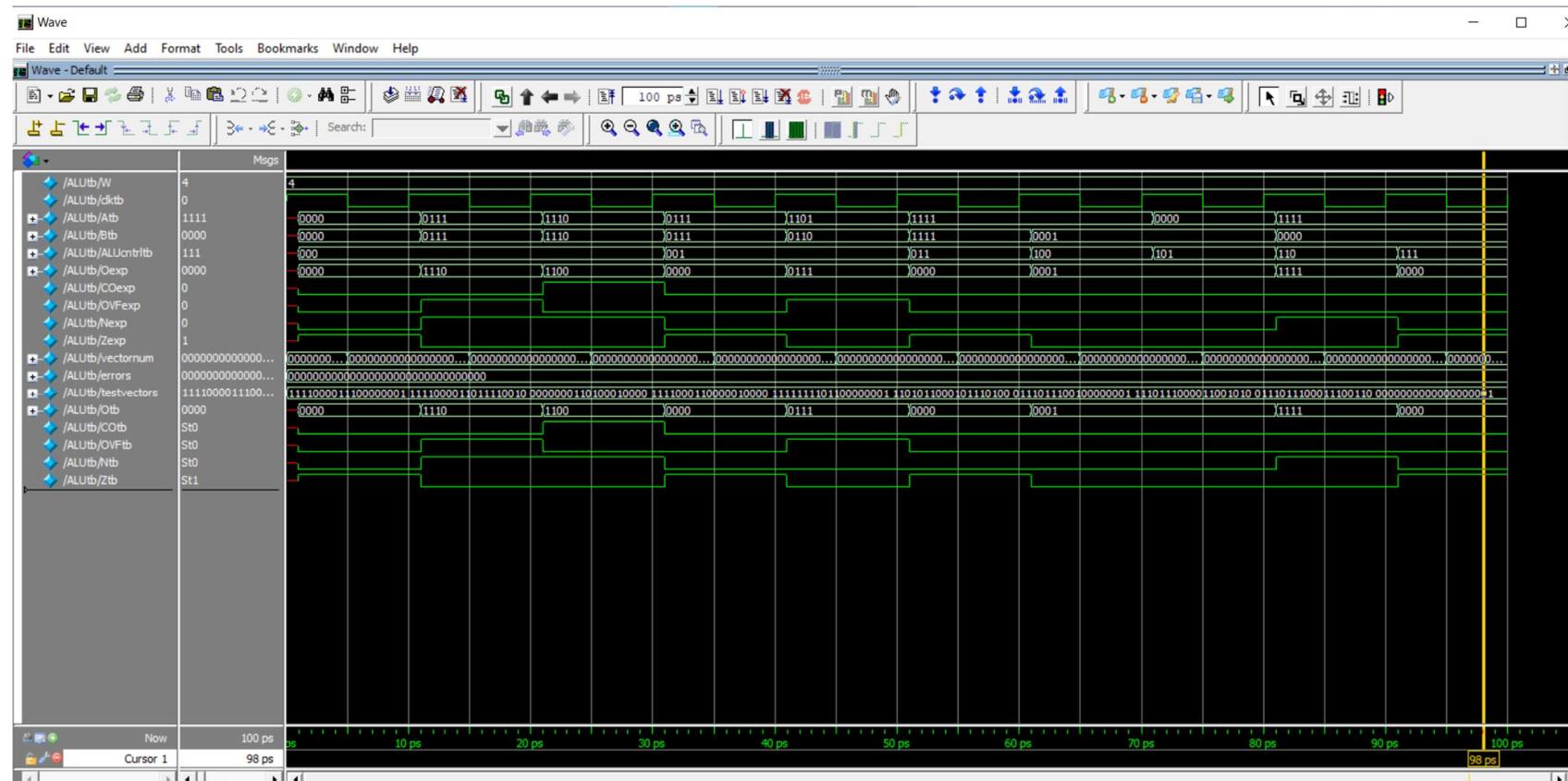
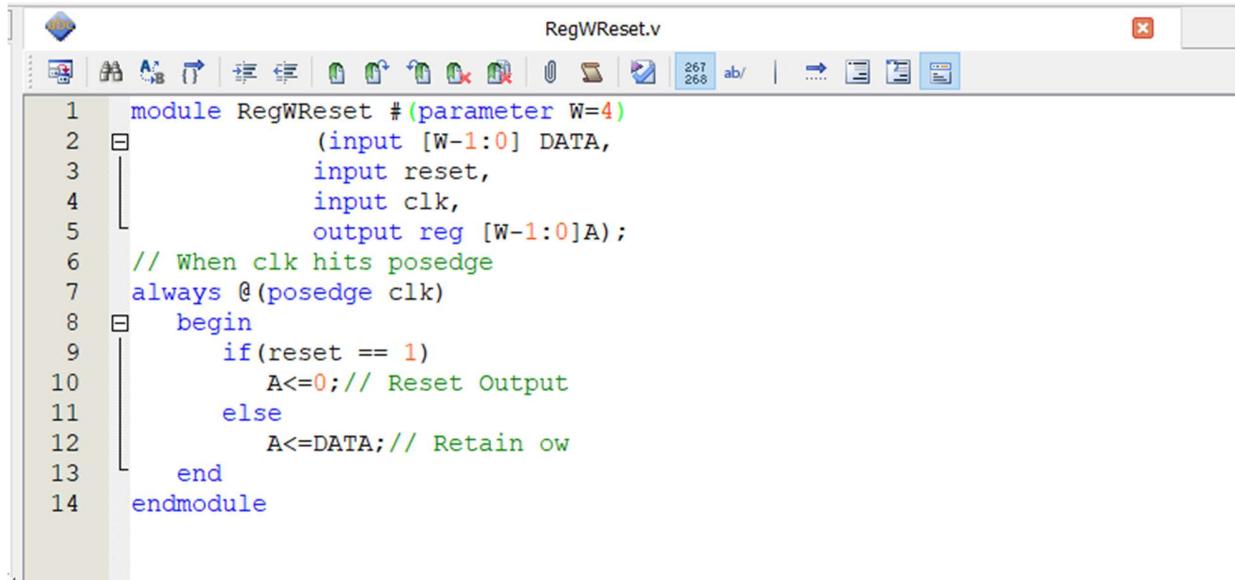


Figure 17: ALU Test Vector Result

As Figures 17 shows, error number is zero during the simulation of test vector, which means my implementation is correct.

1.2.5 Registers

1. Simple register with synchronous reset

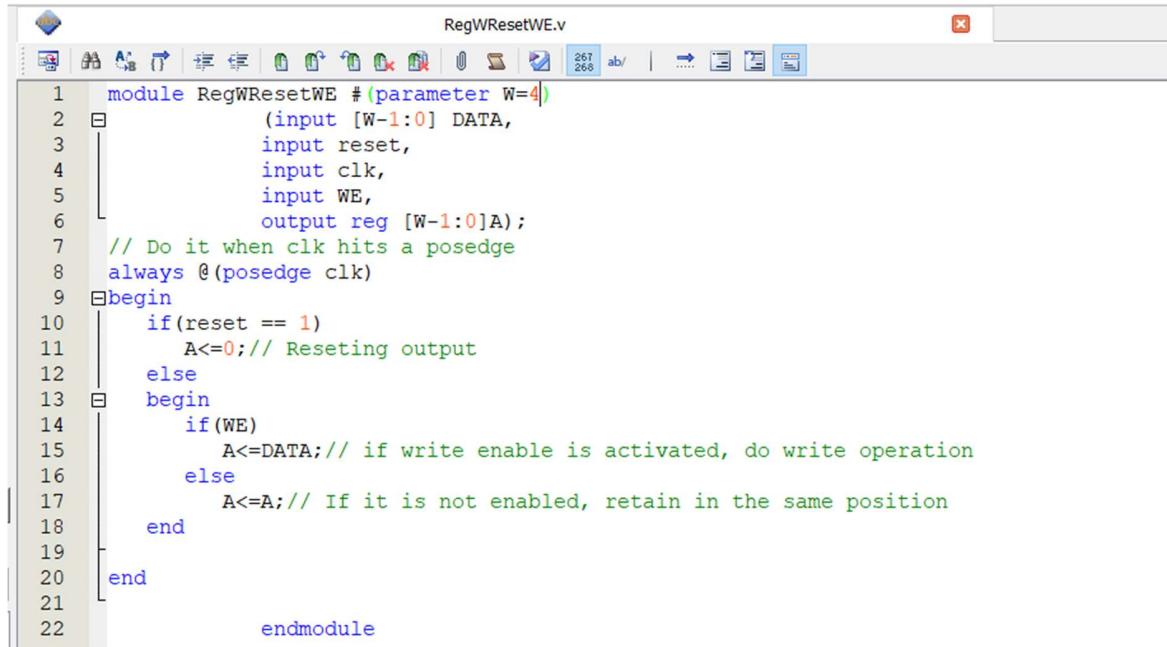


The screenshot shows a Verilog code editor window titled "RegWReset.v". The code defines a module "RegWReset" with a parameter "W=4". The module has four inputs: "DATA" (W-1:0), "reset", "clk", and "WE" (Write Enable). It has one output: "A" (W-1:0). The logic is controlled by an always block that triggers on the rising edge of "clk". If "reset" is high, the output "A" is set to 0. Otherwise, it retains its previous value. The code is well-formatted with color-coded keywords and syntax.

```
1 module RegWReset #(parameter W=4)
2   input [W-1:0] DATA,
3   input reset,
4   input clk,
5   output reg [W-1:0] A;
6   // When clk hits posedge
7   always @(posedge clk)
8   begin
9     if(reset == 1)
10       A<=0;// Reset Output
11     else
12       A<=DATA;// Retain ow
13   end
14 endmodule
```

Figure 18: Simple register with synchronous reset in verilog

2. Register with synchronous reset and write enable

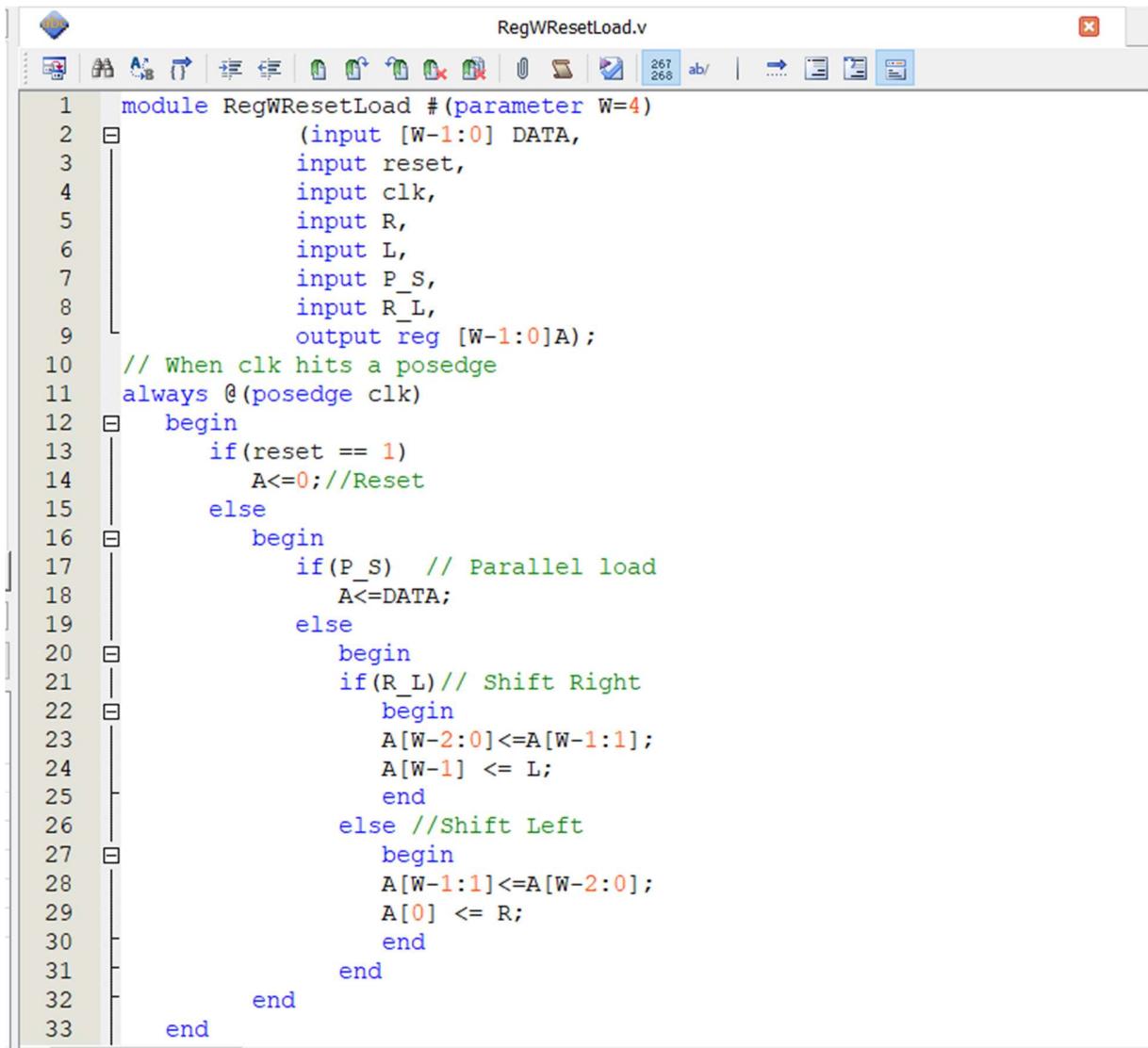


The screenshot shows a Verilog code editor window titled "RegWResetWE.v". The code defines a module "RegWResetWE" with a parameter "W=4". The module has five inputs: "DATA" (W-1:0), "reset", "clk", "WE" (Write Enable), and "WE" (Write Enable). It has one output: "A" (W-1:0). The logic is controlled by an always block that triggers on the rising edge of "clk". If "reset" is high, the output "A" is set to 0. If "WE" is high, the output "A" is updated to the value of "DATA". Otherwise, it retains its previous value. The code uses nested begin-end blocks to handle the different conditions.

```
1 module RegWResetWE #(parameter W=4)
2   input [W-1:0] DATA,
3   input reset,
4   input clk,
5   input WE,
6   output reg [W-1:0] A;
7   // Do it when clk hits a posedge
8   always @(posedge clk)
9   begin
10     if(reset == 1)
11       A<=0;// Resetting output
12     else
13     begin
14       if(WE)
15         A<=DATA;// if write enable is activated, do write operation
16       else
17         A<=A;// If it is not enabled, retain in the same position
18     end
19   end
20 end
21
22 endmodule
```

Figure 19: Register with synchronous reset and write enable in verilog

3. Shift register with parallel and serial load

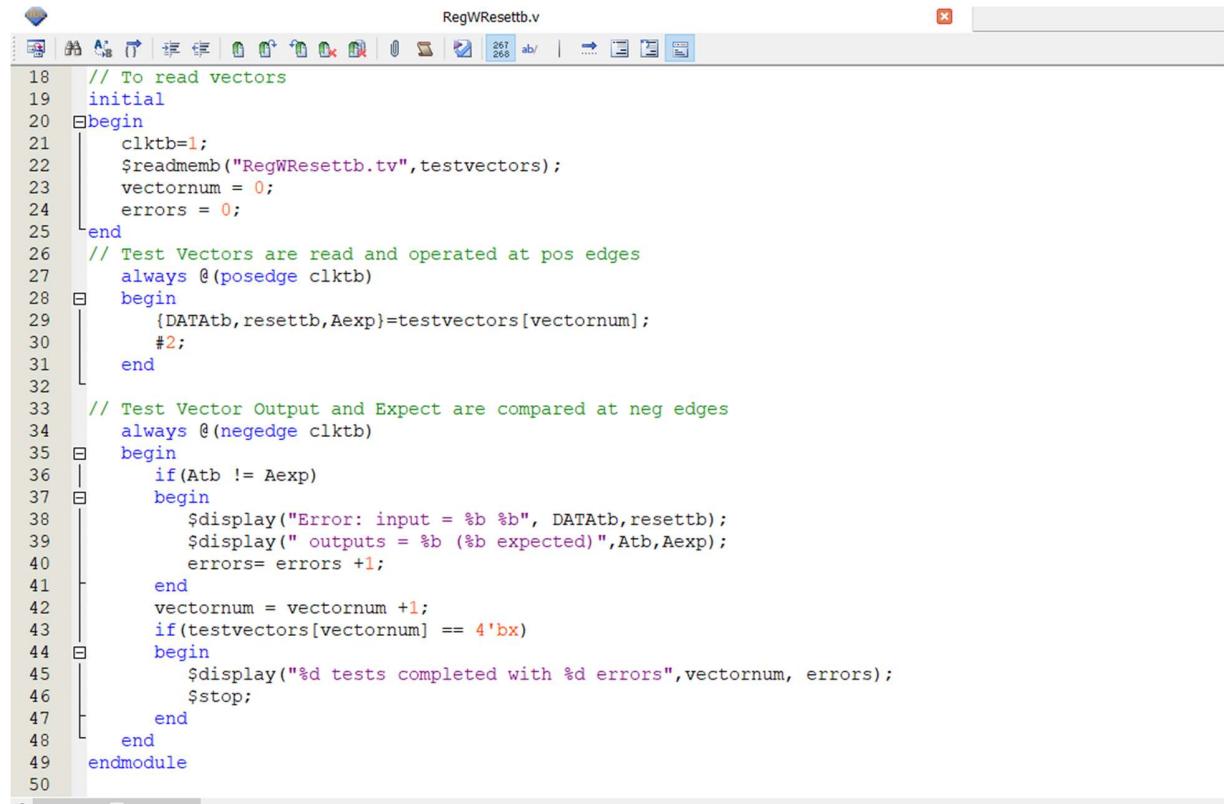


The screenshot shows a Verilog code editor window titled "RegWResetLoad.v". The code defines a module "RegWResetLoad" with a parameter "W=4". The module has inputs: DATA [W-1:0], reset, clk, R, L, P_S, and R_L. It has one output: reg [W-1:0] A. The logic inside the module handles a parallel load if P_S is asserted, or a shift operation (right or left) if R_L is asserted. The shift operation is controlled by the clock edge (posedge clk) and the direction (L or R). The code uses nested begin-end blocks and if-else statements to implement the logic.

```
1  module RegWResetLoad #(parameter W=4)
2      input [W-1:0] DATA,
3          input reset,
4          input clk,
5          input R,
6          input L,
7          input P_S,
8          input R_L,
9          output reg [W-1:0]A;
10 // When clk hits a posedge
11 always @(posedge clk)
12 begin
13     if(reset == 1)
14         A<=0;//Reset
15     else
16         begin
17             if(P_S) // Parallel load
18                 A<=DATA;
19             else
20                 begin
21                     if(R_L)// Shift Right
22                         begin
23                             A[W-2:0]<=A[W-1:1];
24                             A[W-1] <= L;
25                         end
26                     else //Shift Left
27                         begin
28                             A[W-1:1]<=A[W-2:0];
29                             A[0] <= R;
30                         end
31                 end
32             end
33         end
34     end
```

Figure 20: Shift register with parallel and serial load in verilog

4. For the aforementioned 3 registers, write test bench modules to test your implementation.



```
RegWResettb.v
18 // To read vectors
19 initial
20 begin
21     clk_tb=1;
22     $readmemb("RegWResettb.tv",testvectors);
23     vectornum = 0;
24     errors = 0;
25 end
26 // Test Vectors are read and operated at pos edges
27 always @(posedge clk_tb)
28 begin
29     {DATAtb,resettb,Aexp}=testvectors[vectornum];
30     #2;
31 end
32
33 // Test Vector Output and Expect are compared at neg edges
34 always @(negedge clk_tb)
35 begin
36     if(Atb != Aexp)
37         begin
38             $display("Error: input = %b %b", DATAtb,resettb);
39             $display(" outputs = %b (%b expected)",Atb,Aexp);
40             errors= errors +1;
41         end
42     vectornum = vectornum +1;
43     if(testvectors[vectornum] == 4'bxx)
44         begin
45             $display("%d tests completed with %d errors",vectornum, errors);
46             $stop;
47         end
48     end
49 endmodule
50
```

Figure 21: Simple register with synchronous reset Test Bench in Verilog

```

19  begin
20    clktb=1;
21    $readmem("RegWResetWEtb.tv",testvectors);
22    vectornum = 0;
23    errors = 0;
24  end
25  // Test Vectors are read and operated at pos edges
26  always @(posedge clktb)
27  begin
28    {DATAtb,resettb,WEtb,Aexp}=testvectors[vectornum];
29    #2;
30  end
31  // Test Vector Output and Expect are compared at neg edges
32  always @(negedge clktb)
33  begin
34    if(Atb != Aexp)
35    begin
36      $display("Error: input = %b %b %b", DATAtb,resettb,WEtb);
37      $display(" outputs = %b (%b expected)",Atb,Aexp);
38      errors= errors +1;
39    end
40    vectornum = vectornum +1;
41    if(testvectors[vectornum] == 4'bx)
42    begin
43      $display("%d tests completed with %d errors",vectornum, errors);
44      $stop;
45    end
46  end
47 endmodule
48
49

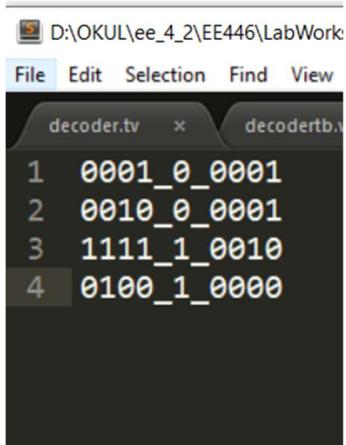
```

Figure 22: Register with synchronous reset and write enable Test Bench in Verilog

```
19 // To read vectors
20 initial
21 begin
22     clktb=1;
23     $readmem("RegWResetLoadtb.tv",testvectors);
24     vectornum = 0;
25     errors = 0;
26 end
27 // Test Vectors are read and operated at pos edges
28 always @(posedge clktb)
29 begin
30     {DATAtb,resettb,Rtb,Ltb,P_Stb,R_Ltb,Aexp}=testvectors[vectornum];
31     #2;
32 end
33 // Test Vector Output and Expect are compared at neg edges
34 always @(negedge clktb)
35 begin
36     if(Atb != Aexp)
37     begin
38         $display("Error: input = %b %b %b %b %b", DATAtb,resettb,Rtb,Ltb,P_Stb,R_Ltb);
39         $display(" outputs = %b (%b expected)",Atb,Aexp);
40         errors= errors +1;
41     end
42     vectornum = vectornum +1;
43     if(testvectors[vectornum] == 4'bx)
44     begin
45         $display("%d tests completed with %d errors",vectornum, errors);
46         $stop;
47     end
48 end
49 endmodule
50
51
```

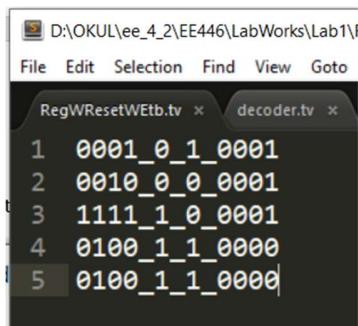
Figure 23: Shift register with parallel and serial load Test Bench in Verilog

5. (If using ModelSim) Provide a vector table to be used by your test bench module to test your implementation.



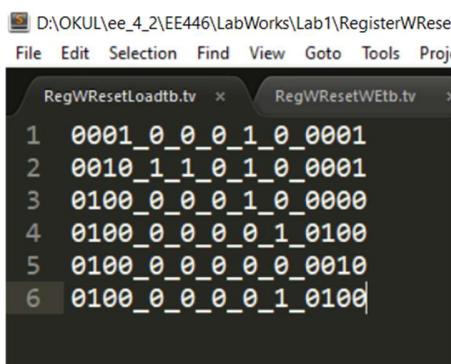
```
D:\OKUL\ee_4_2\EE446\LabWork>
File Edit Selection Find View
decoder.tv * decodertb.v
1 0001_0_0001
2 0010_0_0001
3 1111_1_0010
4 0100_1_0000
```

Figure 24: Simple register with synchronous reset Test Vectors in Verilog



```
D:\OKUL\ee_4_2\EE446\LabWorks\Lab1>
File Edit Selection Find View Goto
RegWResetWEtb.tv * decoder.tv *
1 0001_0_1_0001
2 0010_0_0_0001
3 1111_1_0_0001
4 0100_1_1_0000
5 0100_1_1_0000
```

Figure 25: Register with synchronous reset and write enable Test Vectors in Verilog



```
D:\OKUL\ee_4_2\EE446\LabWorks\Lab1\RegisterWResetWEtv>
File Edit Selection Find View Goto Tools Project
RegWResetLoadtb.tv * RegWResetWEtv.tv *
1 0001_0_0_0_1_0_0001
2 0010_1_1_0_1_0_0001
3 0100_0_0_0_1_0_0000
4 0100_0_0_0_0_1_0100
5 0100_0_0_0_0_0_0010
6 0100_0_0_0_0_1_0100
```

Figure 26: Shift register with parallel and serial load Test Vectors in Verilog

7. Verify that your implementation is correct.

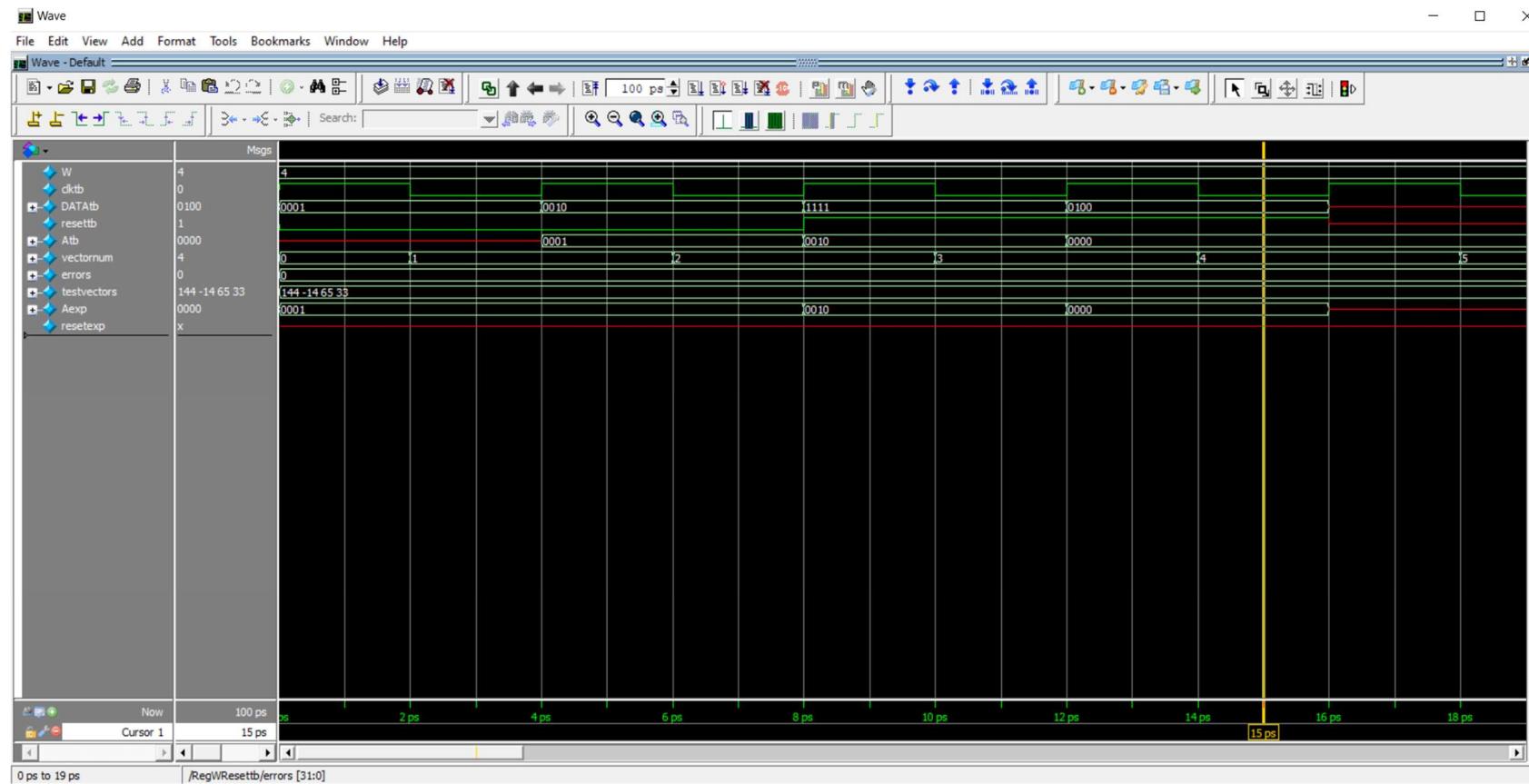


Figure 27: Simulation Results for Simple register with synchronous reset Test Vectors in Verilog

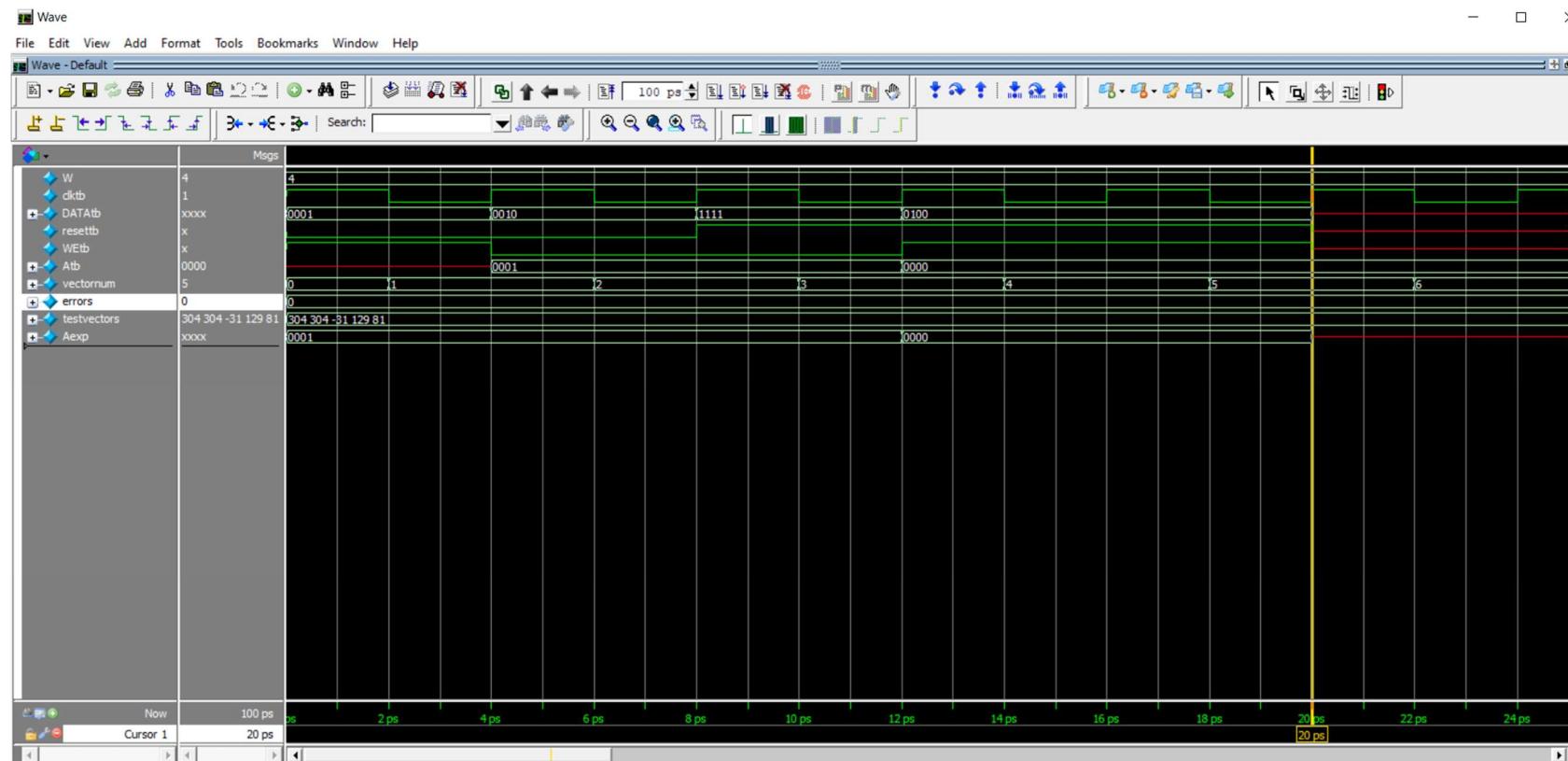


Figure 28: Test Results for Register with synchronous reset and write enable Test Vectors in Verilog

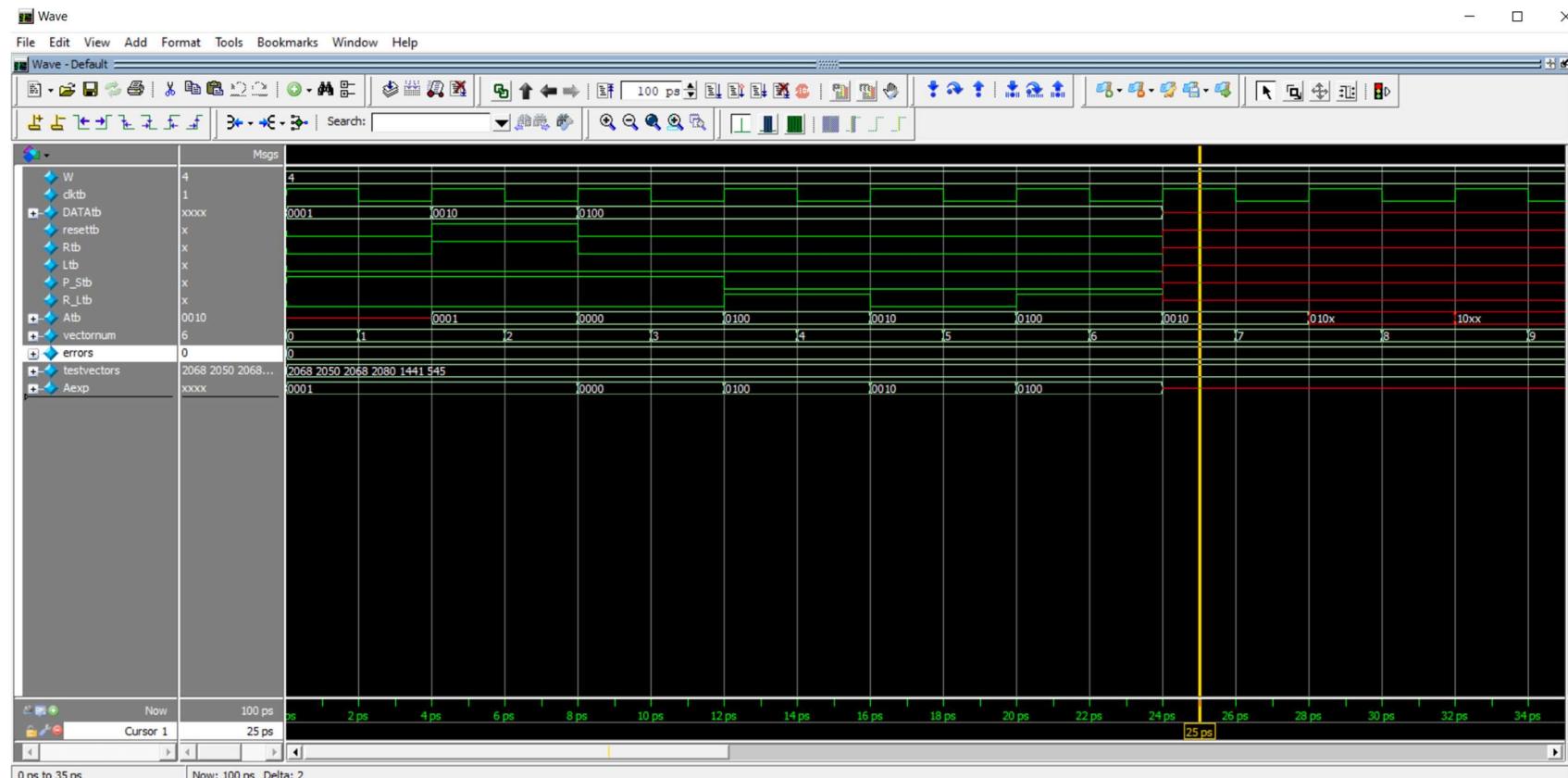
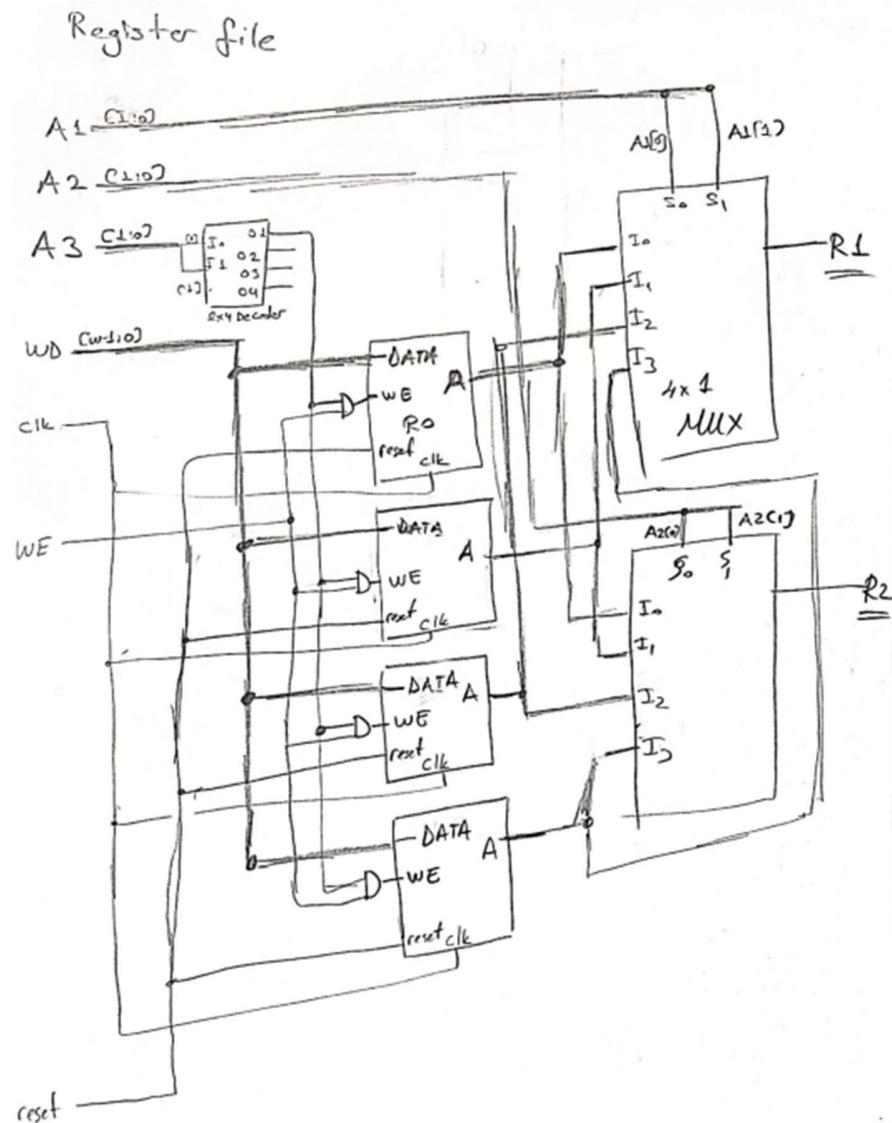


Figure 29: Test Results for Shift register with parallel and serial load Test Vectors in Verilog

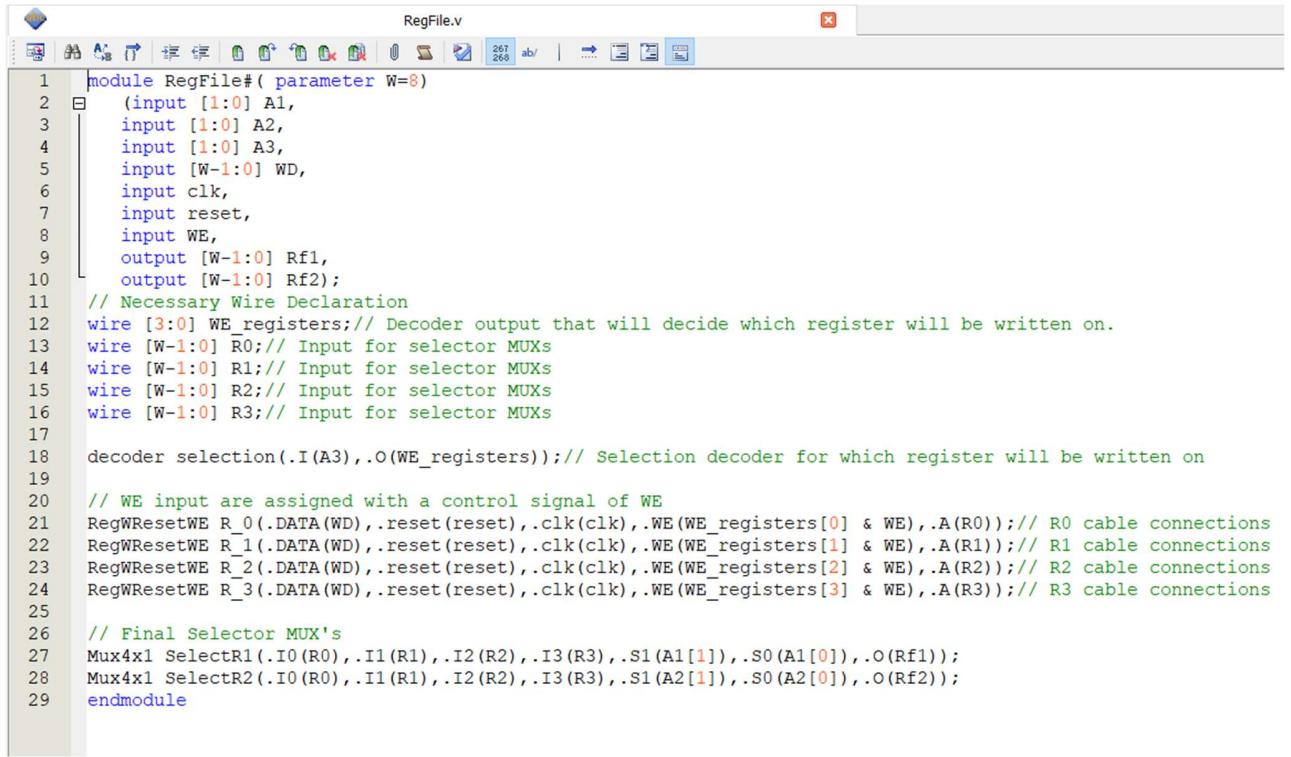
As Figures 27,28 and 29 show, error number is zero during the simulation of test vector, which means my implementation is correct

1.3 Register File

1. Design and sketch (on a paper) a datapath for a register file design using your decoder, multiplexer and register modules. You may use additional gates wherever necessary. For your sketch, you may present your modules with boxes.



2. Implement your design in Verilog HDL.



The screenshot shows a Verilog HDL editor window titled "RegFile.v". The code implements a Register File (RegFile) module. It includes input ports for address A1 (1-bit), A2 (1-bit), A3 (1-bit), and data WD (W-bit). It also includes control inputs for clock (clk), reset, and write enable (WE). The module outputs two data words Rf1 and Rf2. The code uses a selection decoder to determine which register will be written based on A3 and WE. It also includes four MUXes (R0, R1, R2, R3) to select the appropriate data from the four registers. The code is annotated with comments explaining the logic flow and connections.

```
1 module RegFile#( parameter W=8)
2   (input [1:0] A1,
3   input [1:0] A2,
4   input [1:0] A3,
5   input [W-1:0] WD,
6   input clk,
7   input reset,
8   input WE,
9   output [W-1:0] Rf1,
10  output [W-1:0] Rf2);
11 // Necessary Wire Declaration
12 wire [3:0] WE_registers;// Decoder output that will decide which register will be written on.
13 wire [W-1:0] R0;// Input for selector MUXs
14 wire [W-1:0] R1;// Input for selector MUXs
15 wire [W-1:0] R2;// Input for selector MUXs
16 wire [W-1:0] R3;// Input for selector MUXs
17
18 decoder selection(.I(A3),.O(WE_registers));// Selection decoder for which register will be written on
19
20 // WE input are assigned with a control signal of WE
21 RegWResetWE R_0(.DATA(WD),.reset(reset),.clk(clk),.WE(WE_registers[0] & WE),.A(R0));// R0 cable connections
22 RegWResetWE R_1(.DATA(WD),.reset(reset),.clk(clk),.WE(WE_registers[1] & WE),.A(R1));// R1 cable connections
23 RegWResetWE R_2(.DATA(WD),.reset(reset),.clk(clk),.WE(WE_registers[2] & WE),.A(R2));// R2 cable connections
24 RegWResetWE R_3(.DATA(WD),.reset(reset),.clk(clk),.WE(WE_registers[3] & WE),.A(R3));// R3 cable connections
25
26 // Final Selector MUX's
27 Mux4x1 SelectR1(.IO(R0),.I1(R1),.I2(R2),.I3(R3),.S1(A1[1]),.S0(A1[0]),.O(Rf1));
28 Mux4x1 SelectR2(.IO(R0),.I1(R1),.I2(R2),.I3(R3),.S1(A2[1]),.S0(A2[0]),.O(Rf2));
29 endmodule
```

Figure 30: RegFile in verilog

4. (If using ModelSim) Provide a vector table to be used by your test bench module to test your implementation.

```

D:\OKUL\ee_4_2\EE446\LabWorks\Lab1\Try\RegFile\simulation\modelsim

File Edit Selection Find View Goto Tools Project Preferences

RegFiletb.tv — RegFile\simulation\modelsim × RegWResetLoadtb.tv

1 00_01_00_0000_1_0_0000_0000
2 00_01_00_1111_0_1_0000_0000
3 00_01_01_1111_0_1_1111_0000
4 00_01_10_1111_1_1_1111_1111
5 00_01_10_0100_0_1_0000_0000
6 00_10_11_0010_0_1_0000_0100
7 11_10_11_0100_0_0_0010_0100
8 11_10_11_0100_0_0_0010_0100

```

Figure 31: Test Vectors for RegFile

```

30     vectornum = 0;
31     errors = 0;
32   end
33   // Test Vectors are read and operated at pos edges
34   always @(posedge clk)
35   begin
36     {A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rflexp,Rf2exp}=testvectors[vectornum];
37     #2;
38   end
39   // Test Vector Output and Expect are compared at neg edges
40   always @(negedge clk)
41   begin
42     if(Rfltb != Rflexp)
43     begin
44       $display("Error: input = %b %b %b %b %b %b %b", A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rfltb,Rf2exp);
45       $display(" outputs = %b (%b expected)",Rfltb,Rflexp);
46       errors= errors +1;
47     end
48     if(Rf2tb != Rf2exp)
49     begin
50       $display("Error: input = %b %b %b %b %b %b %b", A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rfltb,Rf2tb);
51       $display(" outputs = %b (%b expected)",Rf2tb,Rf2exp);
52       errors= errors +1;
53     end
54     vectornum = vectornum +1;
55     if(testvectors[vectornum] == 4'bxx)
56     begin
57       $display("%d tests completed with %d errors",vectornum, errors);
58       $stop;
59     end
60   end
61 endmodule

```

Figure 32: Test Bench for RegFile

6. Verify that your implementation is correct.

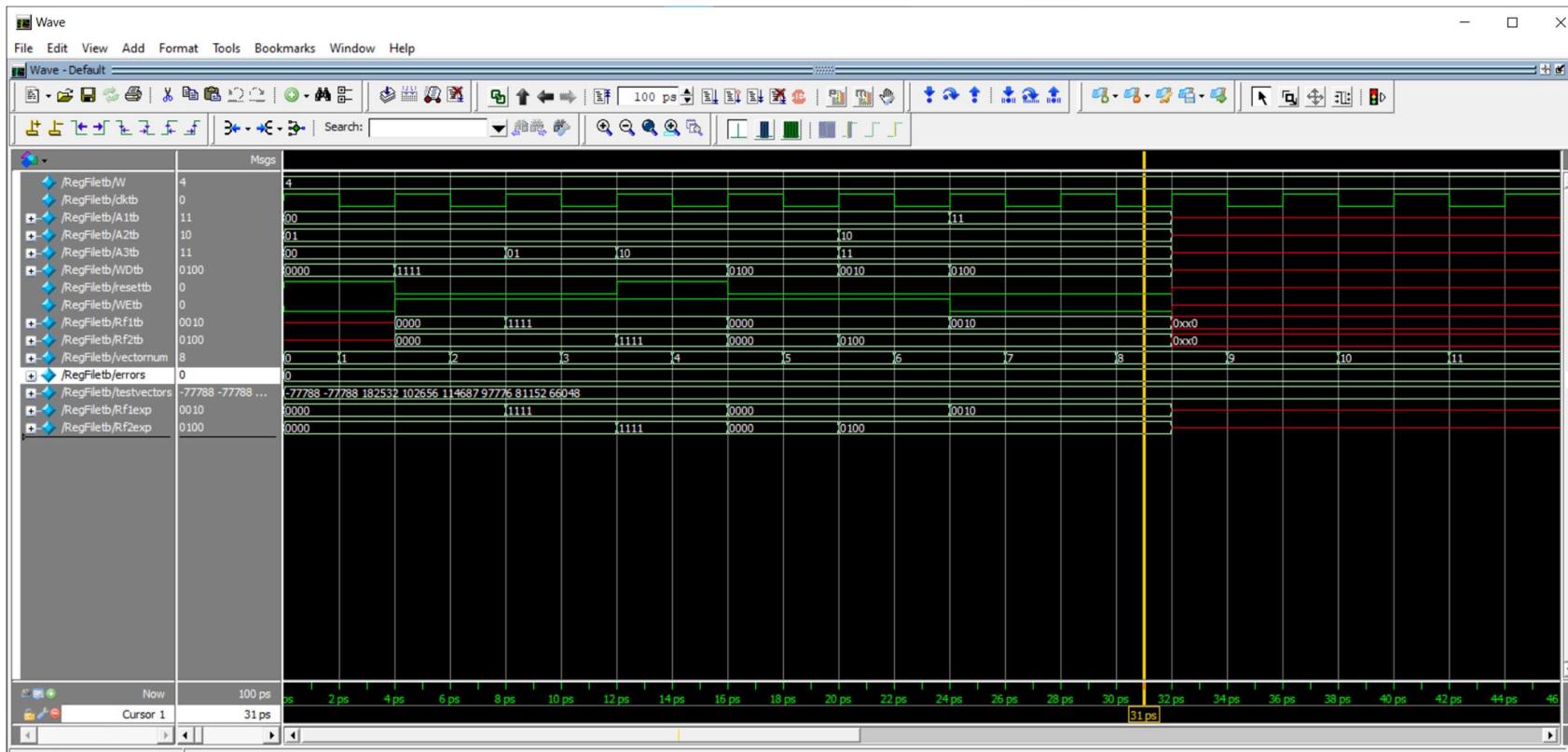


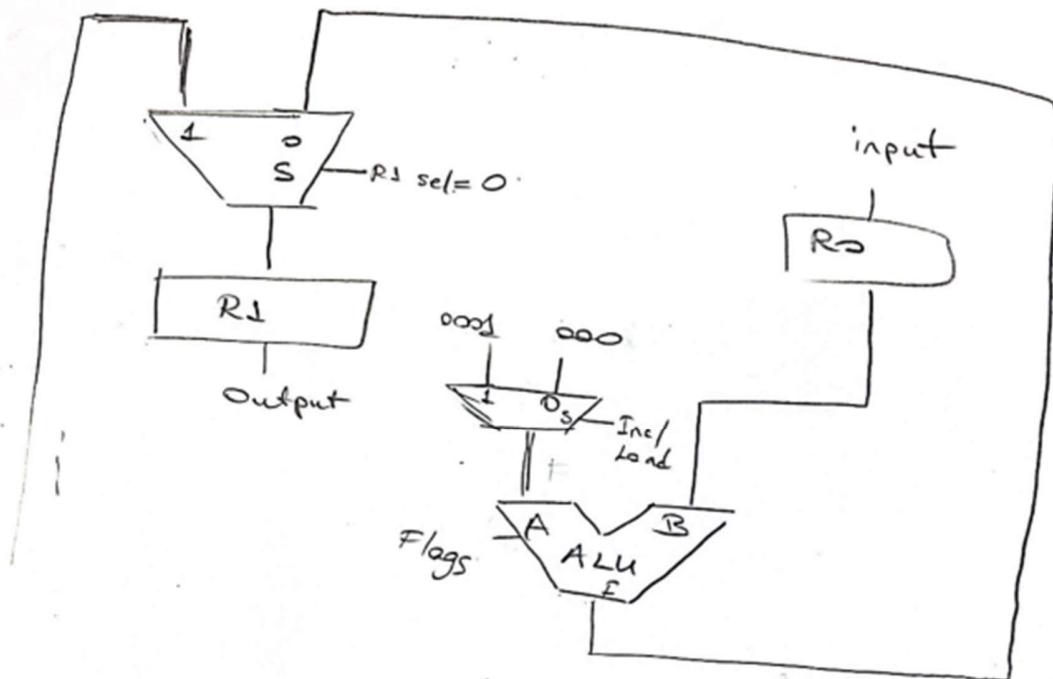
Figure 33: Test Result for RegFile

As Figures 33 shows, error number is zero during the simulation of test vector, which means my implementation is correct.

1.4 Datapath Design for an Architecture

For Only Load & INC:

For only LOAD & Increment Load

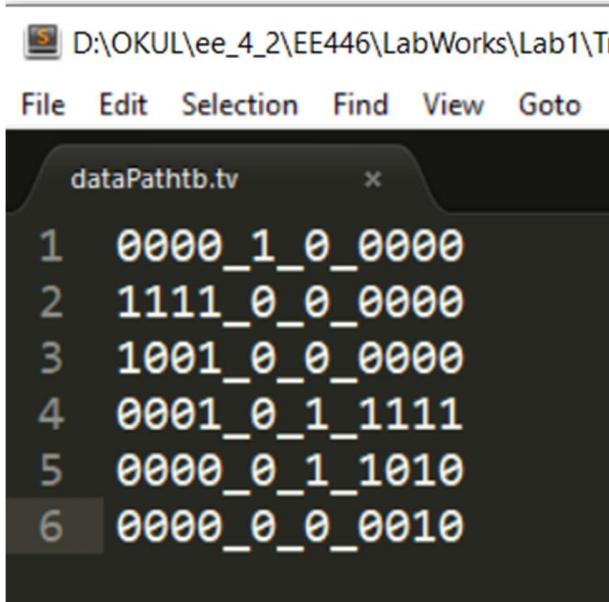


- According the Inc/Load signal we can
Inc / Load: $1 \Rightarrow R_1 \leftarrow R_0 + 1$
 $0 \Rightarrow R_1 \leftarrow R_0$

```
dataPathtb.v

16      #2;
17  end
18 // To read vectors
19 initial
20 begin
21     clk_tb=1;
22     $readmemb("dataPathtb.tv",testvectors);
23     vectornum = 0;
24     errors = 0;
25 end
26 // Test Vectors are read and operated at pos edges
27 always @(posedge clk_tb)
28 begin
29     {R0_tb,reset_tb,Inc_Load_tb,Rlexp}=testvectors[vectornum];
30     #2;
31 end
32 // Test Vector Output and Expect are compared at neg edges
33 always @(negedge clk_tb)
34 begin
35     if(Rltb != Rlexp)
36     begin
37         $display("Error: input = %b %b %b ", R0_tb,Inc_Load_tb,Rltb);
38         $display(" outputs = %b (%b expected)",Rltb,Rlexp);
39         errors= errors +1;
40     end
41     vectornum = vectornum +1;
42     if(testvectors[vectornum] == 4'bxx)
43     begin
44         $display("%d tests completed with %d errors",vectornum, errors);
45         $stop;
46     end
47 end
48 endmodule
```

Figure 34: Test Bench for Data path



D:\OKUL\ee_4_2\EE446\LabWorks\Lab1\T

```
File Edit Selection Find View Goto
dataPathtb.tv ×
1 0000_1_0_0000
2 1111_0_0_0000
3 1001_0_0_0000
4 0001_0_1_1111
5 0000_0_1_1010
6 0000_0_0_0010
```

Figure 35: Test Vectors for Data path

Note: Although it seems like it is operating in 2 cycles, 1 cycle is taking for loading R0.

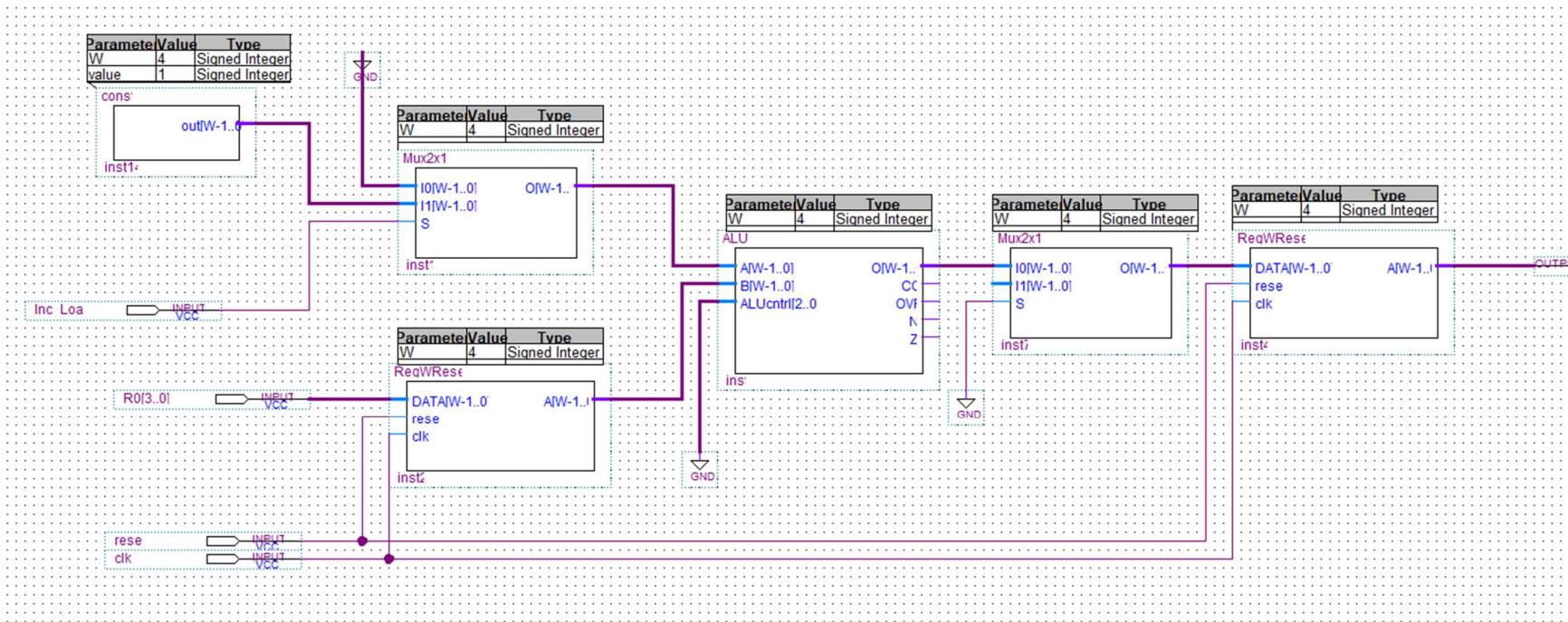


Figure 36: Schematic for Data path

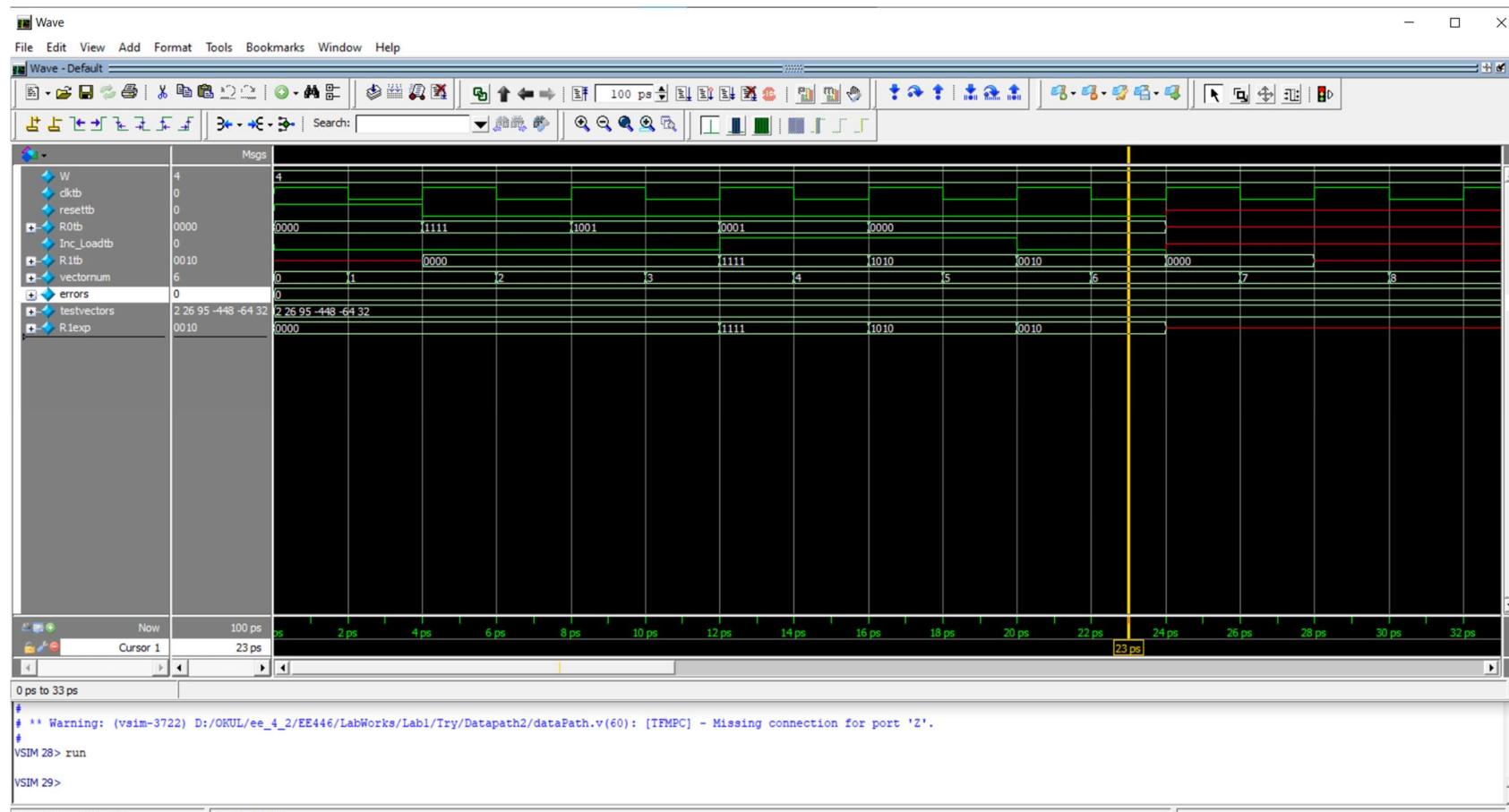
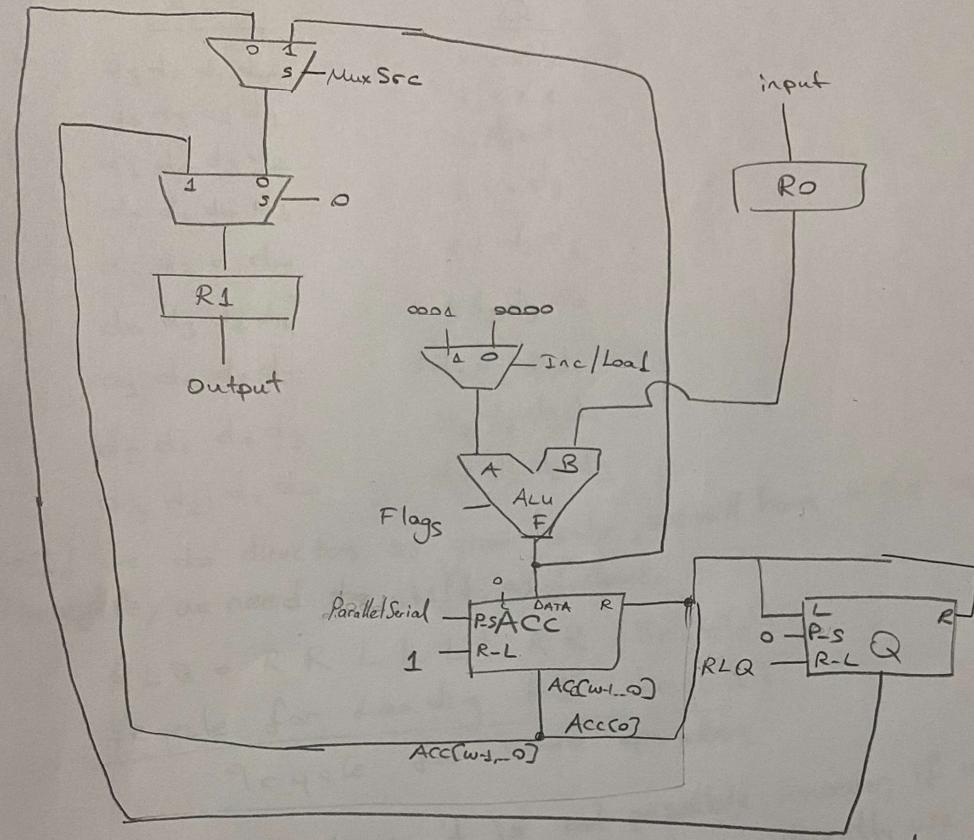


Figure 37: Result Test Vectors for Data path

As Figures 37 shows, error number is zero during the simulation of test vector, which means my implementation is correct.

For Lead, Inc, & Mirror



• For the MUX selection, there is no need for ACC output. Therefore, I connected it as GND (0).

• # of Control pins is four.

1) Mux SRC: it will be one, when there is a operation of LOAD or INC.

2) Inc/Load: It will decide whether inc or direct load. (If it is 1, then inc, o.w direct load.)

3) Parallel Serial: It will be 1 for the Mirror operation at the beginning so that

we can take RO directly ($\text{Inc/Load} = 0$). O.w it will be zero.

4) RLQ: It will

4) RLQ: It is the essential part of mirror operation, since it will hold the number. Let's say,

<u>Acc</u>	<u>Q</u>
$d_3 d_2 d_1 d_0$	$\times \times \times \times$
$d_0 d_3 d_2 d_1$	$d_0 \times \times \times$
$d_1 d_0 d_3 d_2$	$d_1 \cancel{d}_0 \times \times$
$d_2 d_1 d_0 d_3$	$d_0 \times \times d_2$
$d_3 d_2 d_1 d_0$	$\times \times d_2 d_3$
$d_0 d_3 d_2 d_1$	$\times d_2 d_3 d_0$
$d_1 d_0 d_3 d_2$	$d_2 d_3 d_0 d_1$
$d_2 d_1 d_0 d_3$	$d_2 d_2 d_3 d_0$
$d_3 d_2 d_1 d_0$	$d_3 d_2 d_2 d_3$

If we do directors as given order, we will have mirror operation.
Therefore, we need to shift as follows:

$$\begin{aligned} RLQ &= R R L L L L R R \text{ (8cycle)} \\ &\quad \text{1 cycle for loading R1 (1cycle)} \\ &\quad \text{+ 9 cycle for whole operation} \end{aligned}$$

As discussed above, it is not possible; however, if we were not restricted in this condition, we can directly use bits for RO and we can get result of " $d_3 d_2 d_2 d_3$ ".

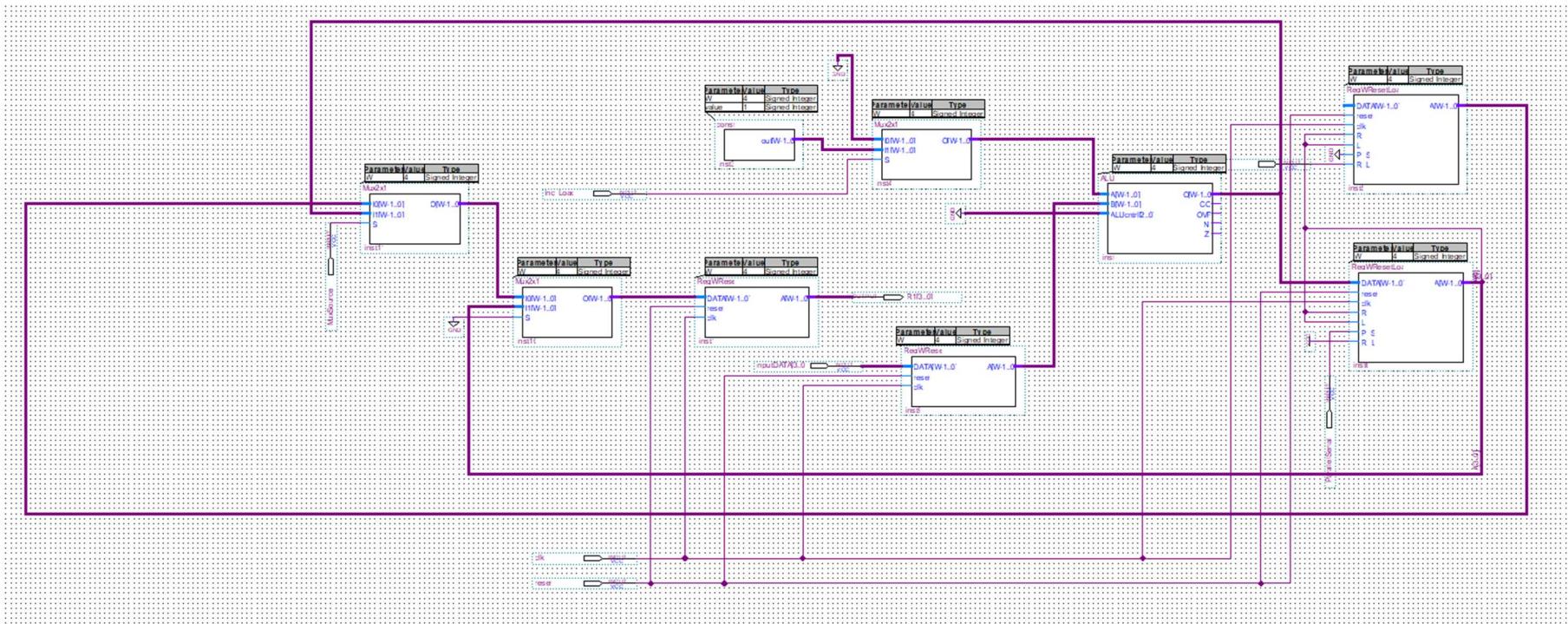


Figure 38: Schematic for full datapath

It is little bit complicated sorry :(It is 1:16. Therefore, I do not have to explain all structure.

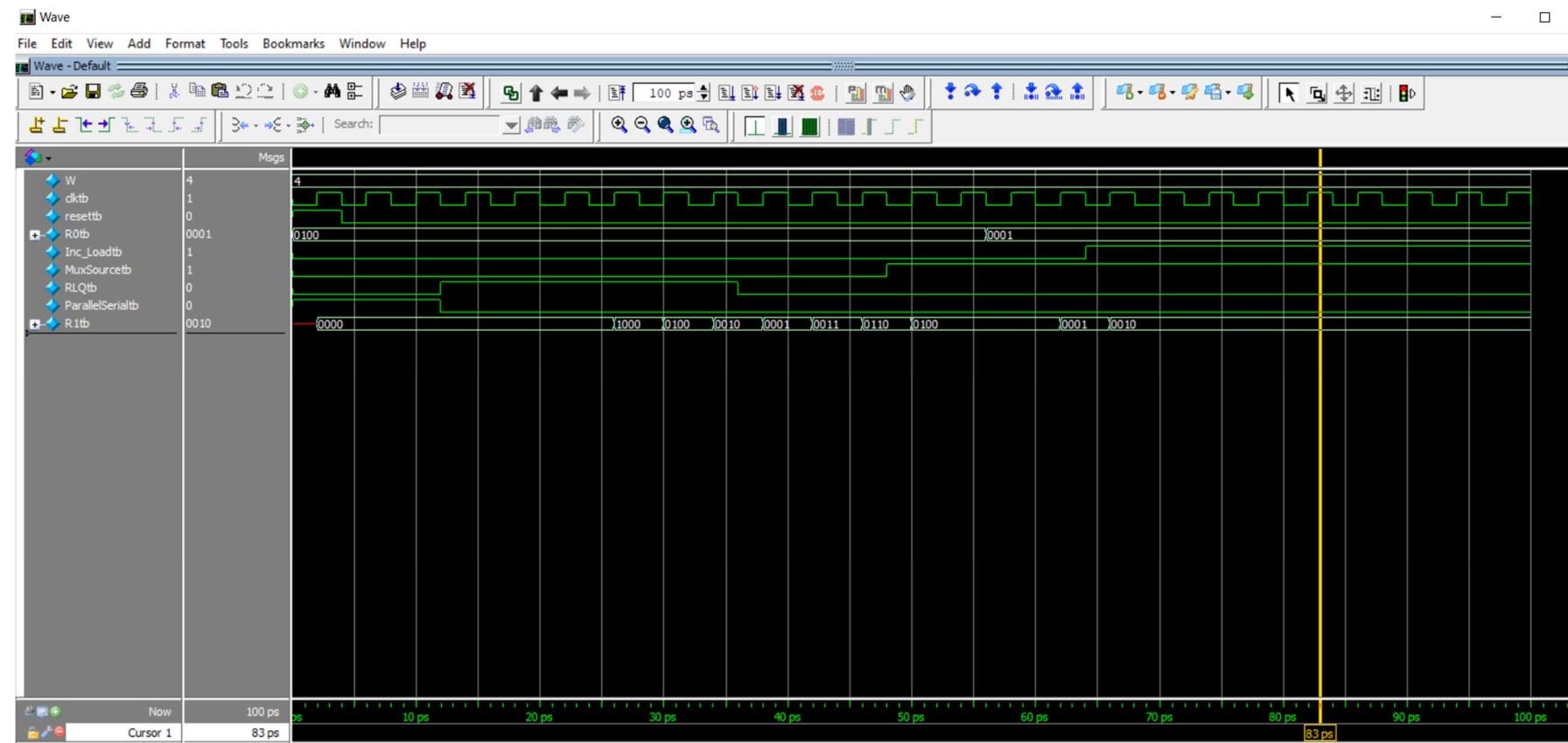


Figure 39: Simulation for full datapath

```

module dataPath tb #(parameter W=4)();
//Necessary initialization
reg clktb,resettb;
reg [W-1:0] R0tb;
reg Inc_Loadtb;
reg MuxSourcetb,RLQtb,ParallelSerialtb;
wire [W-1:0] Rltb;

dataPath DUT(.clk(clktb),.reset(resettb),.Inc_Load(Inc_Loadtb),
    .inputDATA(R0tb),.MuxSource(MuxSourcetb),
    .RLQ(RLQtb),
    .ParallelSerial(ParallelSerialtb),.R1(Rltb));

//Clock Cycles
always
begin
    #2;
    clk_tb = ~clk_tb;
end

initial begin
//To do mirror operation
    clk_tb = 0; // initial clock value
    Inc_Loadtb=0;//Increment is disabled since it is mirror operation
    resettb = 1;//Reset all sync registers
    R0tb= 4'b0100;// Input Data
    MuxSourcetb=0; // Since it is mirror operation R1 = 0
    RLQtb=0; // I will shift first R
    ParallelSerialtb=1;// Took parallel input from output of ALU
    #4;// Wait one cycle to reset
    resettb = 0; // Disabling reset
    RLQtb=0;
    #8;
    ParallelSerialtb=0;
    RLQtb=1;
    // 1st Right
    #4;
    // 2nd Right
    RLQtb= 1;
    #4;
    // 3rd step is Left
    RLQtb= 1;
    #4;
    // 4th step is Left
    RLQtb= 1;
    #4;
    // 5th step is Left
    RLQtb= 1;
    #4;
    // 6th step is Left
    RLQtb= 1;
    #4;
    // 7th step is Right
    RLQtb= 0;
    #4;
    // 8th step is Right
    RLQtb= 0;
    #8;
// To do Direct Load Mode
    MuxSourcetb = 1;
    #8;
    R0tb=4'b0001;
    #8;
// To do Incremented Load Mode
    Inc_Loadtb = 1;

```

Figure 40: Test Vector for full datapath

