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
    clktb=0;
    $readmemb("decoder.tv",testvectors);
    vectornum = 0;
    errors = 0;
end

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

// Test Vector Output and Expect are compared at neg edges
    always @(negedge clktb)
    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
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.

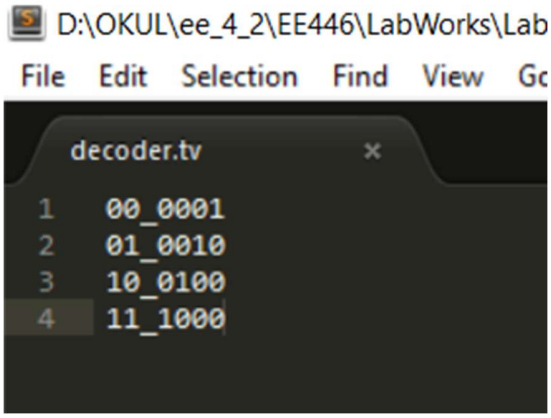


Figure 4: 2x4 Decoder Test Vector file

4. Verify that your implementation is correct.

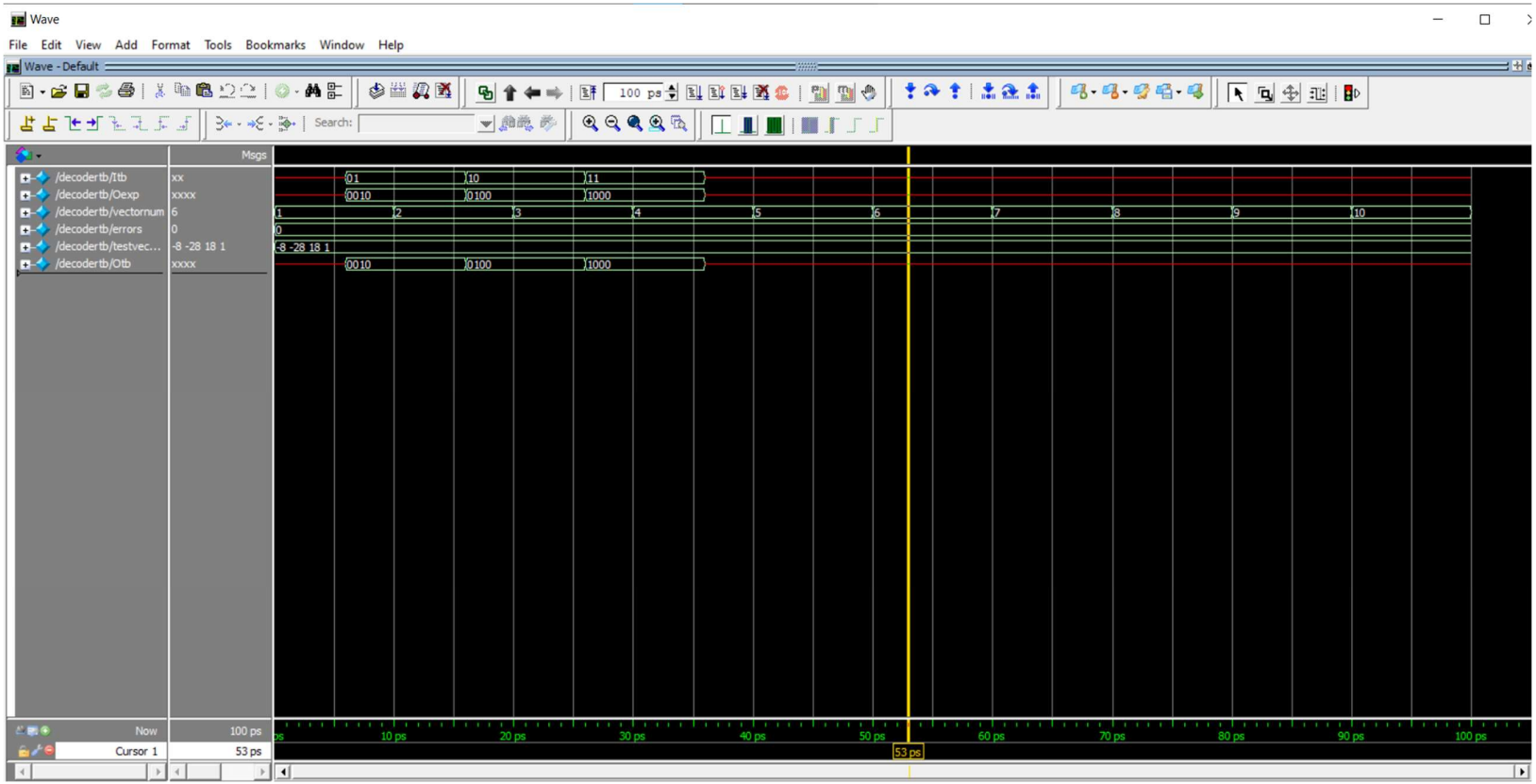


Figure 5: 2x4 Decoder Test Vector Result

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.

Mux2x1.v | Mux2x1tb.v*

```
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

Mux4x1.v

```
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      $readmemb("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
    clktb=1;
    $readmemb("mux2x1tb.tv",testvectors);
    vectornum = 0;
    errors = 0;
end

// Test Vectors are read and operated at pos edges
    always @(posedge clktb)
    begin
        #1;
        {I0tb,I1tb,S0tb,Oexp}=testvectors[vectornum];
    end
// Test Vector Output and Expect are compared at neg edges
    always @(negedge clktb)
    begin
        if(Otb != Oexp)
        begin
            $display("Error: input = %b ", S0tb);
            $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.

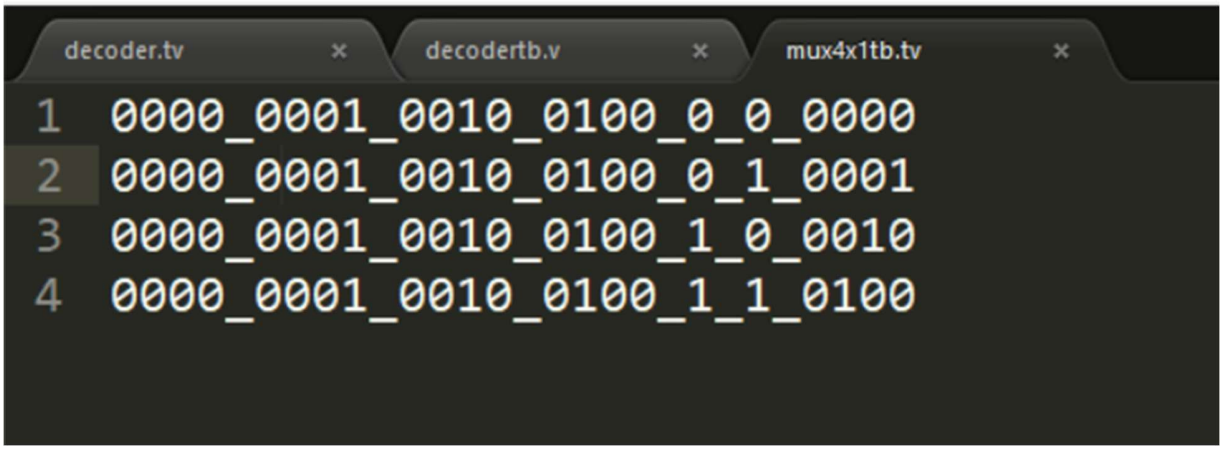


Figure 10: 4x1 MUX Test Vectors

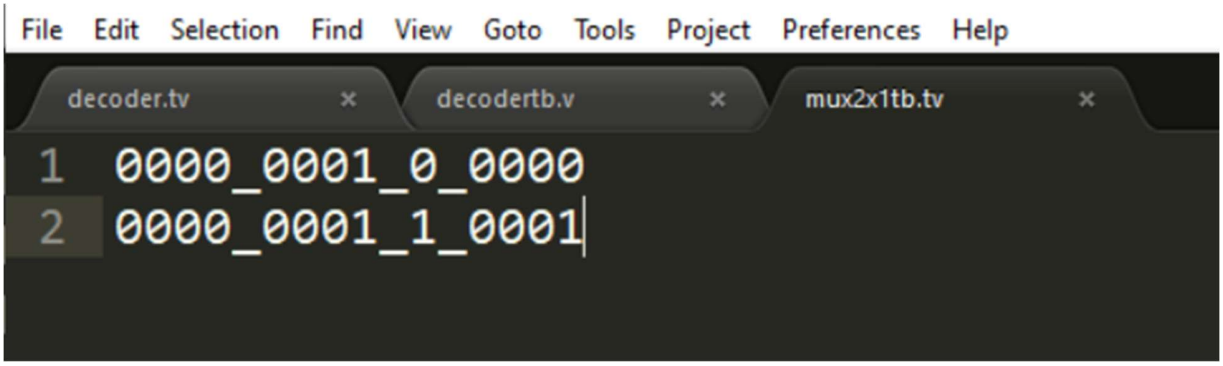


Figure 11: 2x1 MUX Test Vectors

4. Verify that your implementations are correct.

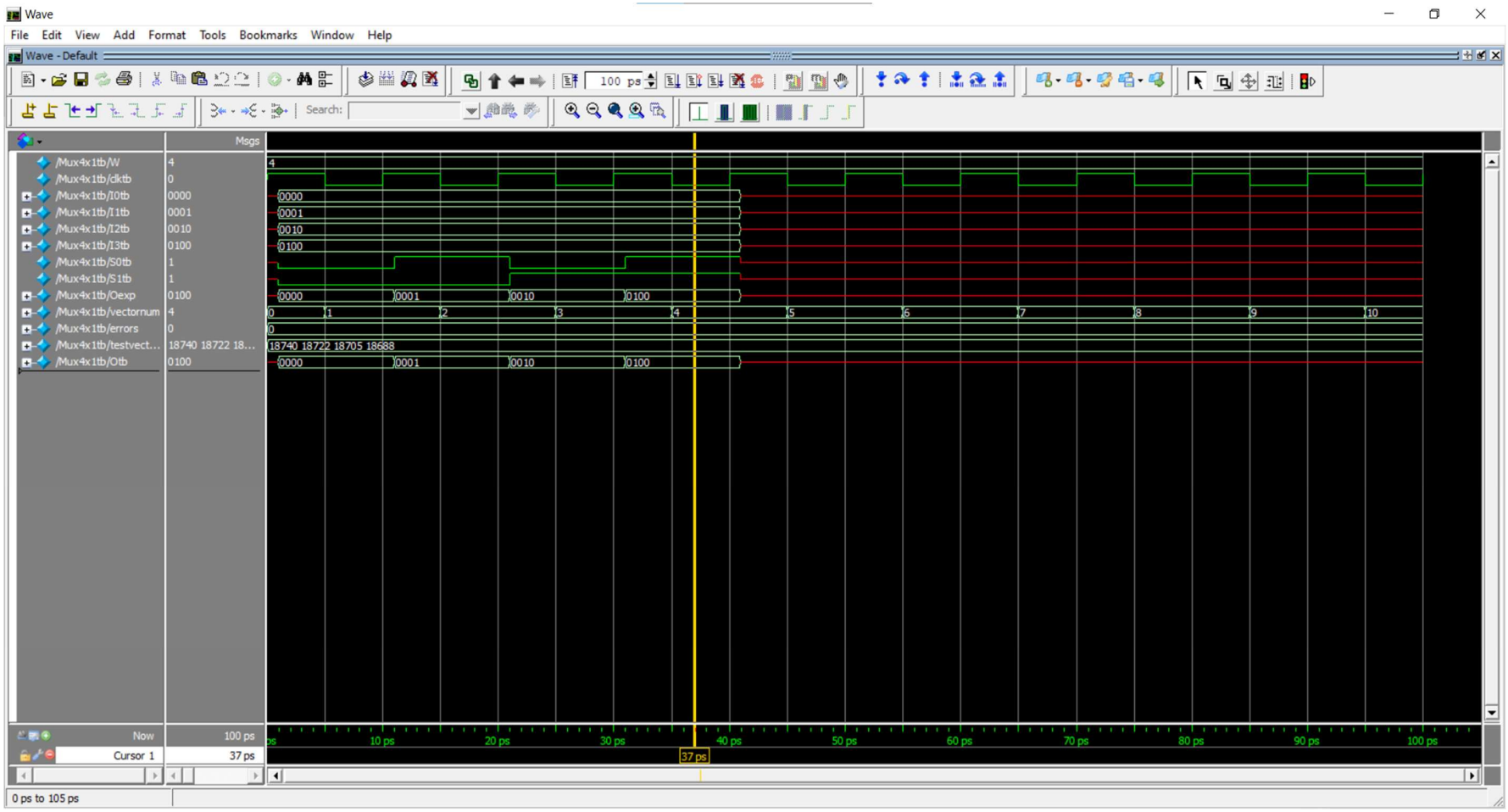


Figure 12: 4x1 MUX Test Vector Result

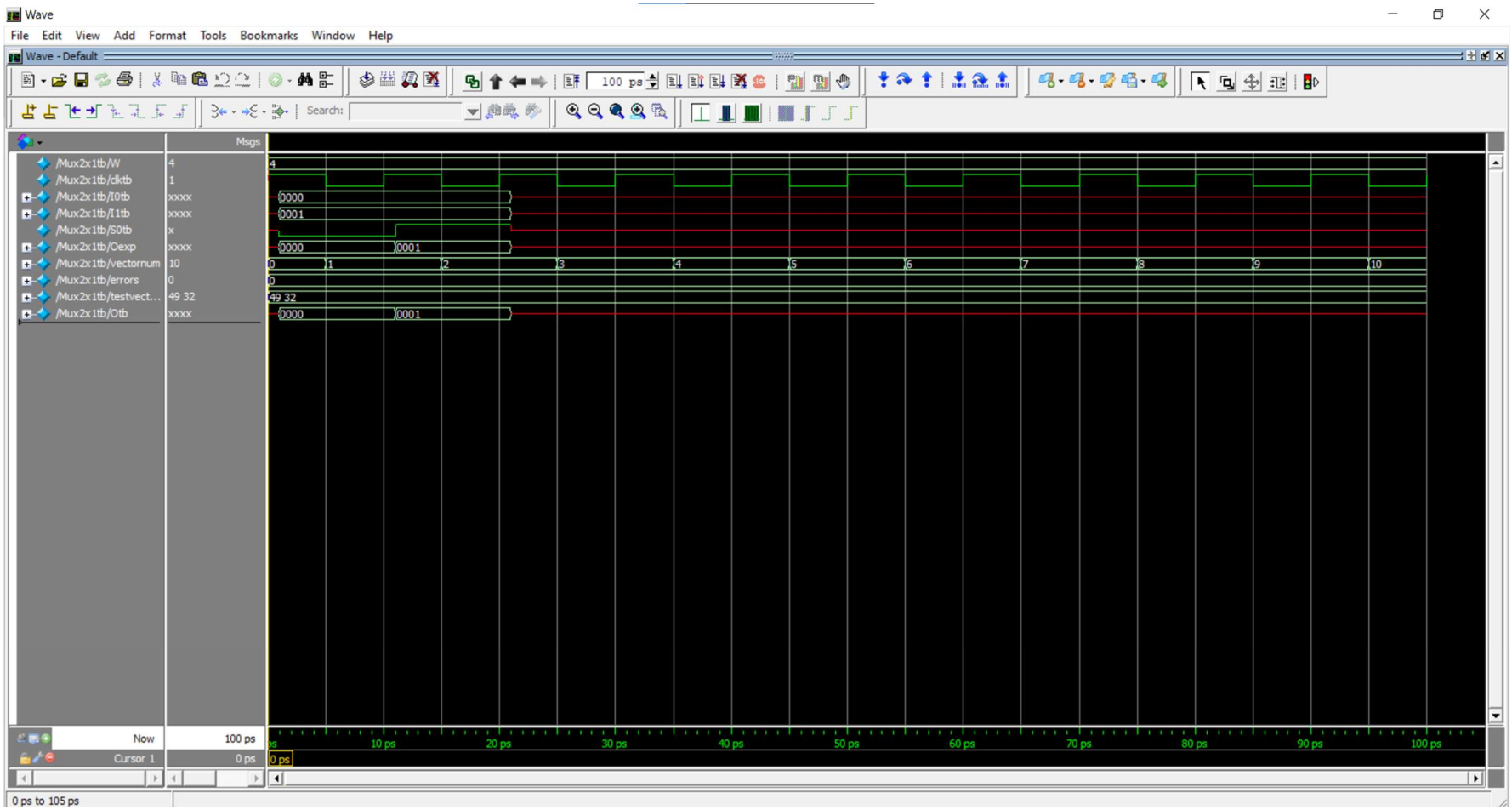


Figure 13: 2x1 MUX Test Vector Result

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.

```
// AlU module
module ALU #(parameter W=4)
      (input [W-1:0] A,
      input [W-1:0] B,
      input [2:0] ALUcntrl,
      output reg [W-1:0] O,
      output reg CO,
      output reg OVF,
      output reg N,
      output reg Z);
// Since switch case block can be implemented only in always block
// I created always block with default sensitivity list
always @(*)
begin
   OVF=0;// Assign default parameters
   CO=0;// These parameters are meaningful in only addition and subtraction
   // Doing operations according to value of control signals
   case(ALUcntrl)
      // Addition
      3'b000:
         begin
         {CO,O} = A+B;
         Z = (O==0) ? 1:0;
         N=O[W-1];
         // Overflow occurs when A and B are positive however result is negative
         if (~A[W-1] & ~B[W-1])
            begin
            if(O[W-1])
               OVF=1;
            end
         // Overflow occurs when A and B are negative however result is positive
         if (A[W-1] & B[W-1])
            begin
            if(~O[W-1])
               OVF=1;
            end
         end
      // Subtraction
      3'b001:
         begin
         {CO,O} = A-B;
         Z = (O==0) ? 1:0;
         // Overflow occurs when A is Negative and B is positive however result is positi
         if (A[W-1] & ~B[W-1])
            begin
            if(~O[W-1])
               OVF=1;
            end
         // Overflow occurs when A is Positive and B is negative however result is negati
         if (~A[W-1] & B[W-1])
            begin
            if(O[W-1])
               OVF=1;
            end
         N=O[W-1];
         end
      // Subtraction
      3'b010:
         begin
         {CO,O} = B-A;
         Z = (O==0) ? 1:0;
         // Overflow occurs when A is positive and B is negative however result is positi
         if (~A[W-1] & B[W-1])
            begin
            if(~O[W-1])
```

```
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

```
module ALUtb#(parameter W=4);

reg clktb;
reg [3:0] Atb;
reg [3:0] Btb;
reg [2:0] ALUcntrltb;
reg [3:0] Oexp;
reg COexp;
reg OVFexp;
reg Nexp;
reg Zexp;
reg [31:0] vectornum,errors;
reg [18:0] testvectors[9:0];
wire [3:0] Otb;
wire COtb;
wire OVFtb;
wire Ntb;
wire Ztb;

ALU dut(.A(Atb),.B(Btb),.ALUcntrl(ALUcntrltb),
        .O(Otb),.CO(COtb),.OVF(OVFtb),.N(Ntb),.Z(Ztb));
// To generate clock cycles
always
begin
   clktb=~clktb;
   #5;
end

// To read vectors
initial
begin
   clktb=1;
   $readmemb("ALUtb.tv",testvectors);
   vectornum = 0;
   errors = 0;
end
// Test Vectors are read and operated at pos edges
   always @(posedge clktb)
   begin
      #1;
      {Atb,Btb,ALUcntrltb,Oexp,COexp,OVFexp,Nexp,Zexp}=testvectors[vectornum];
   end
// Test Vector Output and Expect are compared at neg edges
   always @(negedge clktb)
   begin
      if(Otb != Oexp)
      begin
         $display("Error: input = %b %b", Atb, Btb);
         $display("O outputs = %b (%b expected)",Otb,Oexp);
         errors= errors +1;
      end
      if(COtb != COexp)
      begin
         $display("Error: input = %b %b", Atb, Btb);
         $display("CO outputs = %b (%b expected)",COtb,COexp);
         errors= errors +1;
      end
      if(OVFtb != OVFexp)
      begin
         $display("Error: input = %b %b", Atb, Btb);
         $display("OVF outputs = %b (%b expected)",OVFtb,OVFexp);
         errors= errors +1;
      end
      if(Ntb != Nexp)
      begin
         $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.

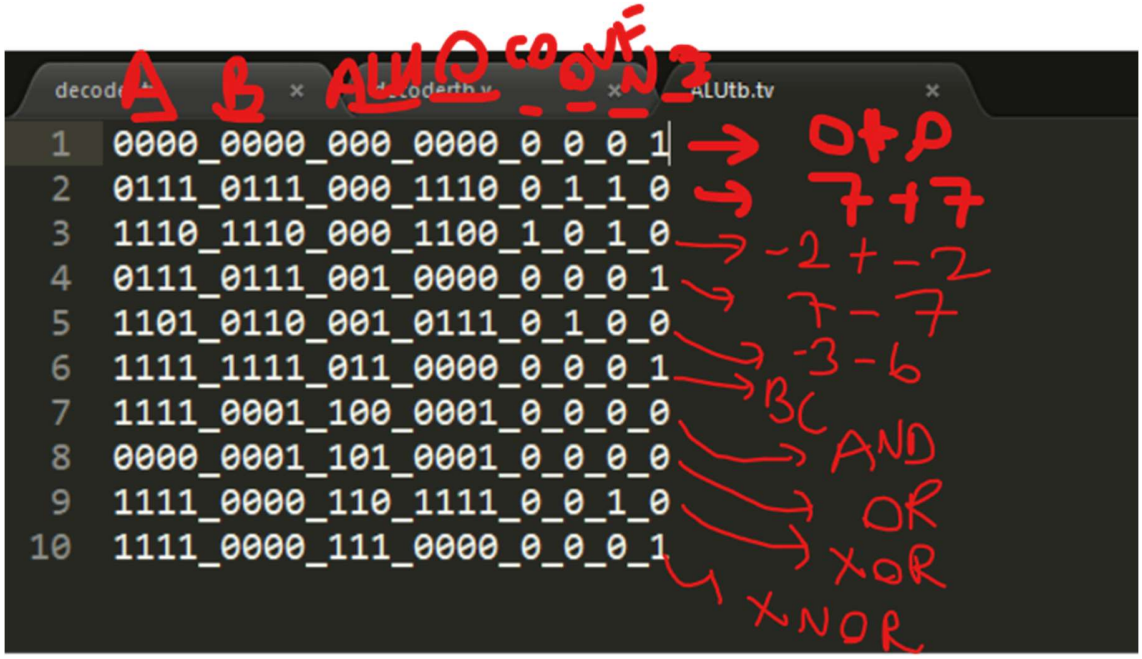


Figure 16: ALU Test Vector with explanation

6. Verify that your implementation is correct.

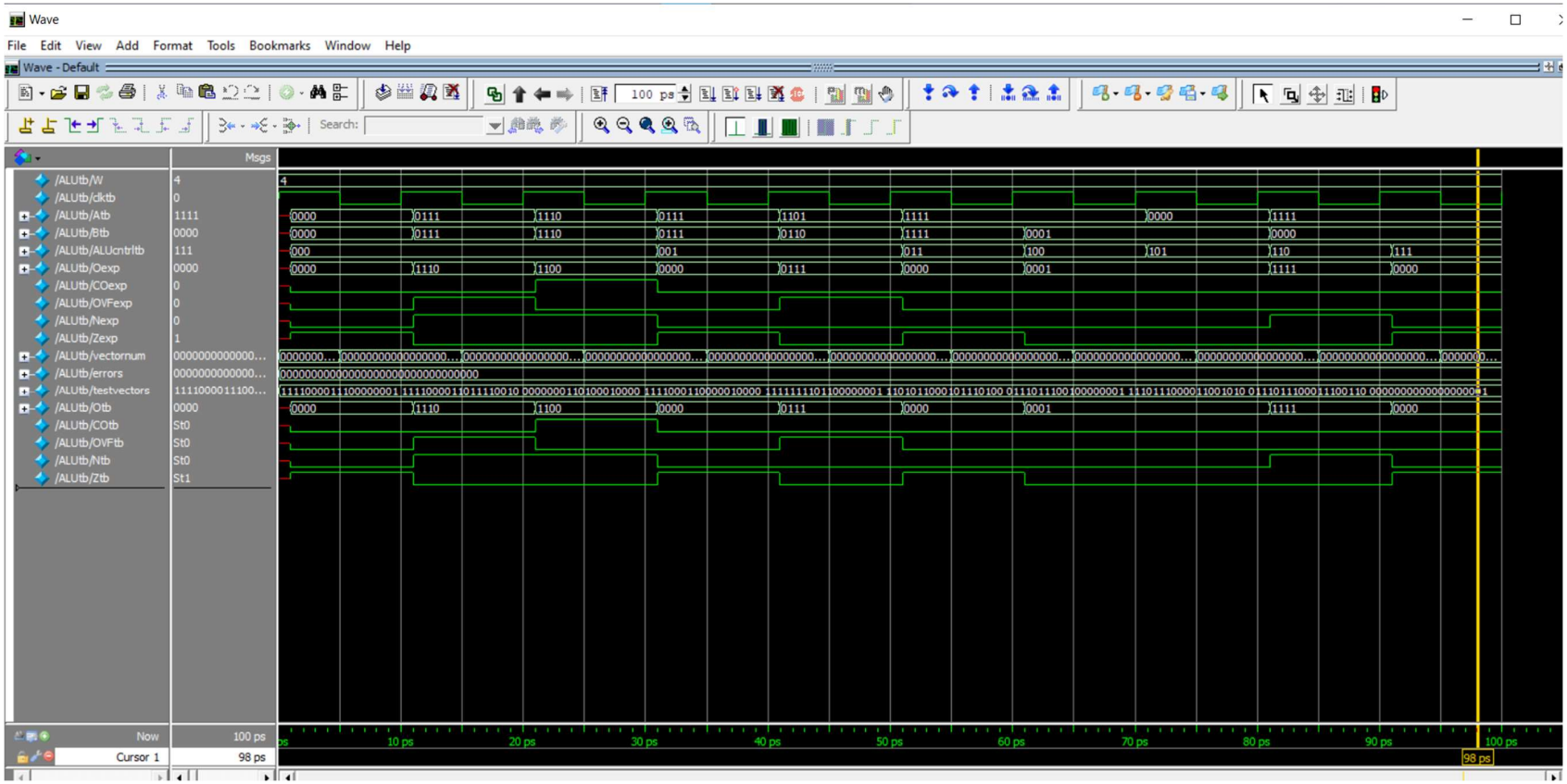


Figure 17: ALU Test Vector Result

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

RegWReset.v

```
module RegWReset #(parameter W=4)
            (input [W-1:0] DATA,
            input reset,
            input clk,
            output reg [W-1:0]A);
// When clk hits posedge
always @(posedge clk)
   begin
      if(reset == 1)
         A<=0;// Reset Output
      else
         A<=DATA;// Retain ow
   end
endmodule
```

Figure 18: Simple register with synchronous reset in verilog

2. Register with synchronous reset and write enable

RegWResetWE.v

```
module RegWResetWE #(parameter W=4)
            (input [W-1:0] DATA,
            input reset,
            input clk,
            input WE,
            output reg [W-1:0]A);
// Do it when clk hits a posedge
always @(posedge clk)
begin
   if(reset == 1)
      A<=0;// Reseting output
   else
   begin
      if(WE)
         A<=DATA;// if write enable is activated, do write operation
      else
         A<=A;// If it is not enabled, retain in the same position
   end

end

            endmodule
```

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

3. Shift register with parallel and serial load

RegWResetLoad.v

```
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
```

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

```
// To read vectors
initial
begin
    clktb=1;
    $readmemb("RegWResettb.tv",testvectors);
    vectornum = 0;
    errors = 0;
end
// Test Vectors are read and operated at pos edges
    always @(posedge clktb)
    begin
        {DATAtb,resettb,Aexp}=testvectors[vectornum];
        #2;
    end

// Test Vector Output and Expect are compared at neg edges
    always @(negedge clktb)
    begin
        if(Atb != Aexp)
        begin
            $display("Error: input = %b %b", DATAtb,resettb);
            $display(" outputs = %b (%b expected)",Atb,Aexp);
            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 21: Simple register with synchronous reset Test Bench in Verilog

```
19  begin
20      clktb=1;
21      $readmemb("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     $readmemb("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 %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.

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

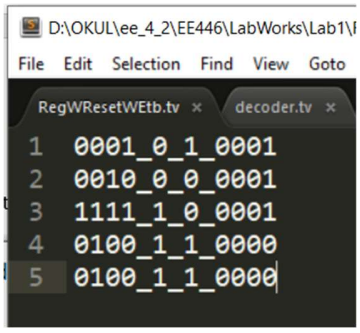


Figure 25: Register with synchronous reset and write enable Test Vectors in Verilog

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

7. Verify that your implementation is correct.

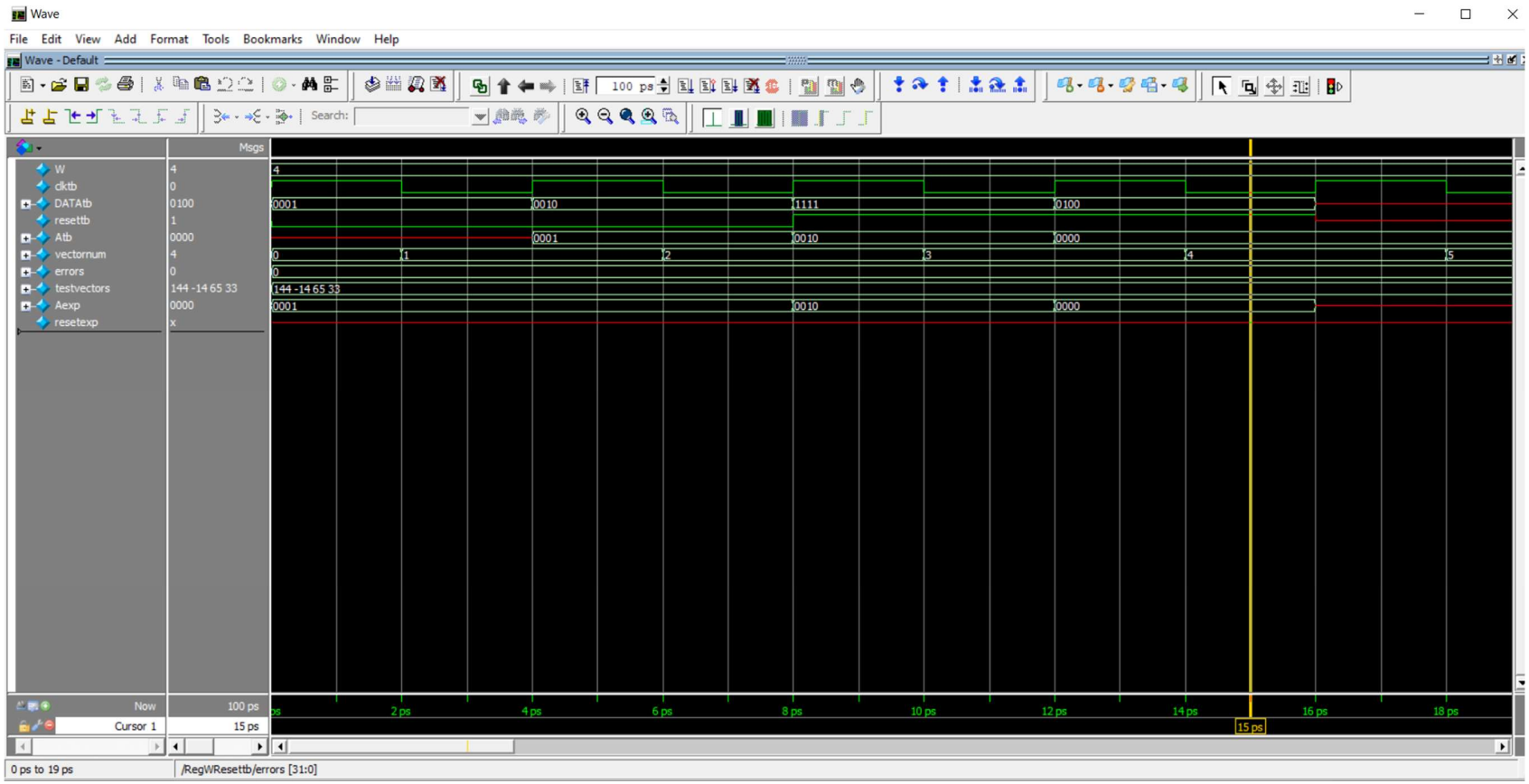


Figure 27: Simulation Results for Simple register with synchronous reset Test Vectors in Verilog

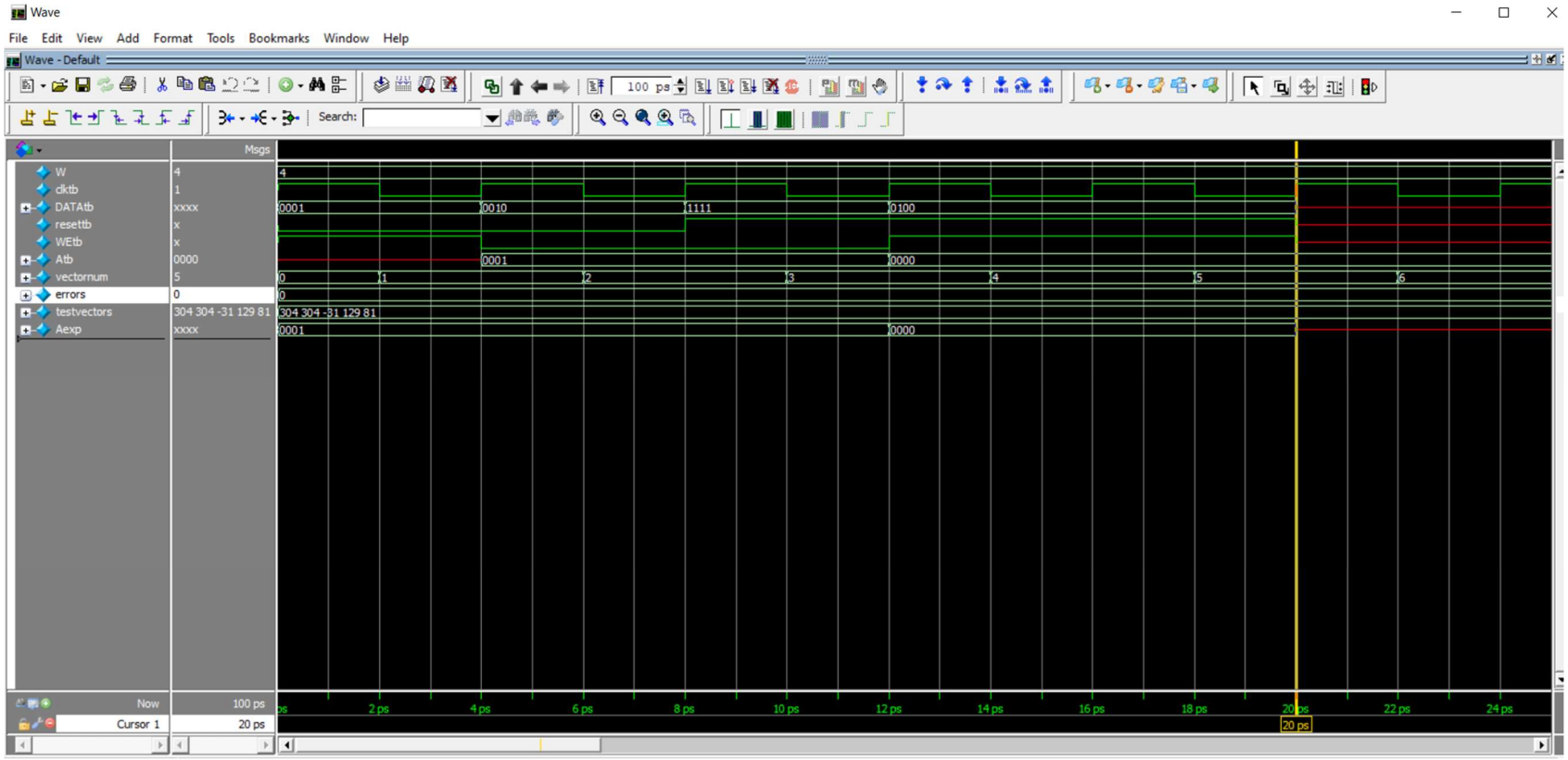


Figure 28: Test Results for Register with synchronous reset and write enable Test Vectors in Verilog

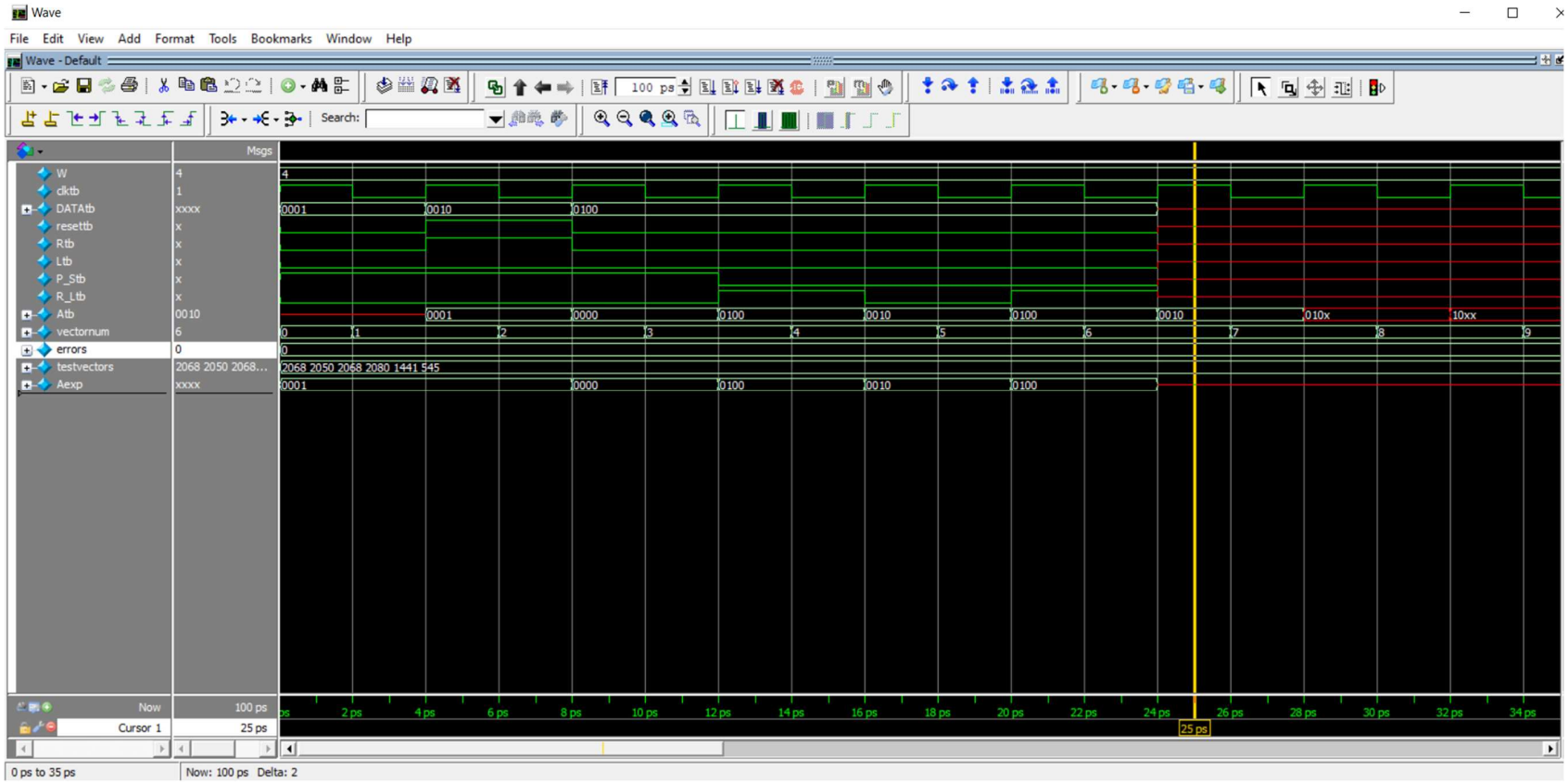
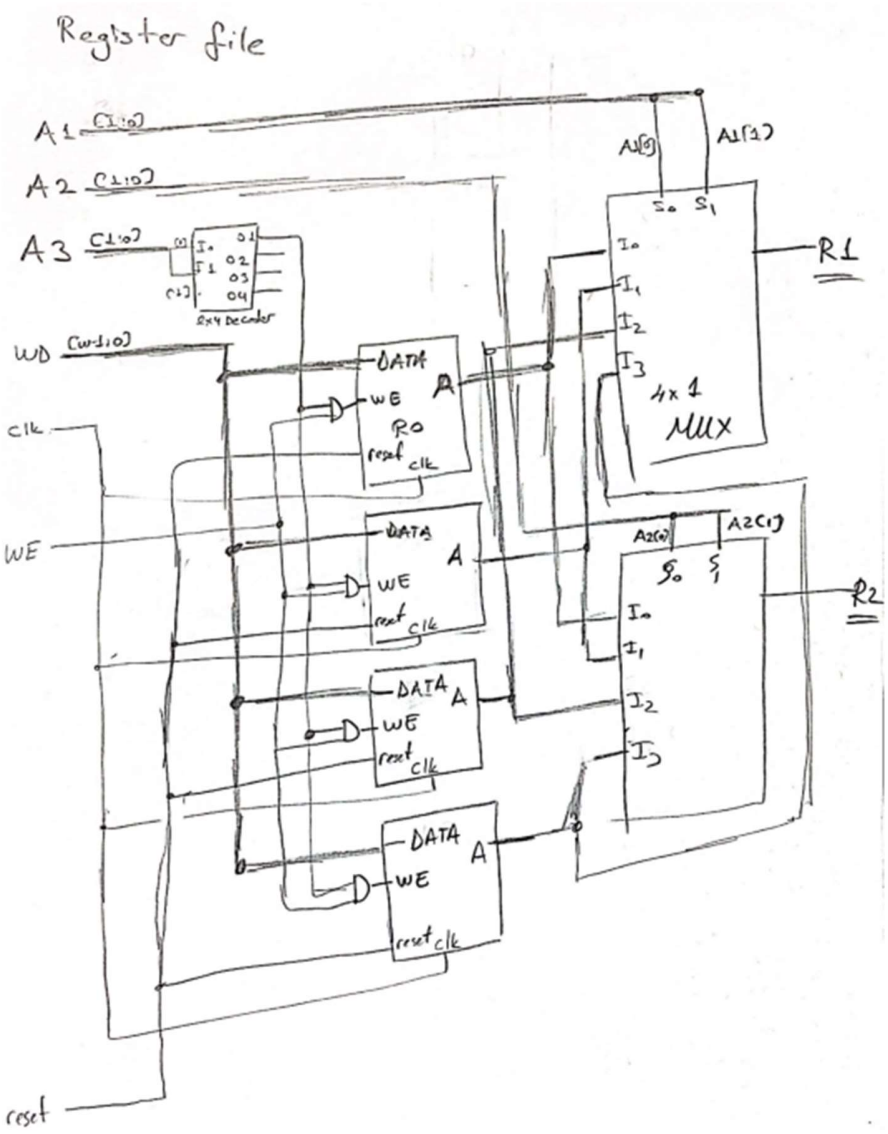


Figure 29: Test Results for Shift register with parallel and serial load Test Vectors in Verilog

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.

RegFile.v

```
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(.I0(R0),.I1(R1),.I2(R2),.I3(R3),.S1(A1[1]),.S0(A1[0]),.O(Rf1));
28 Mux4x1 SelectR2(.I0(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.

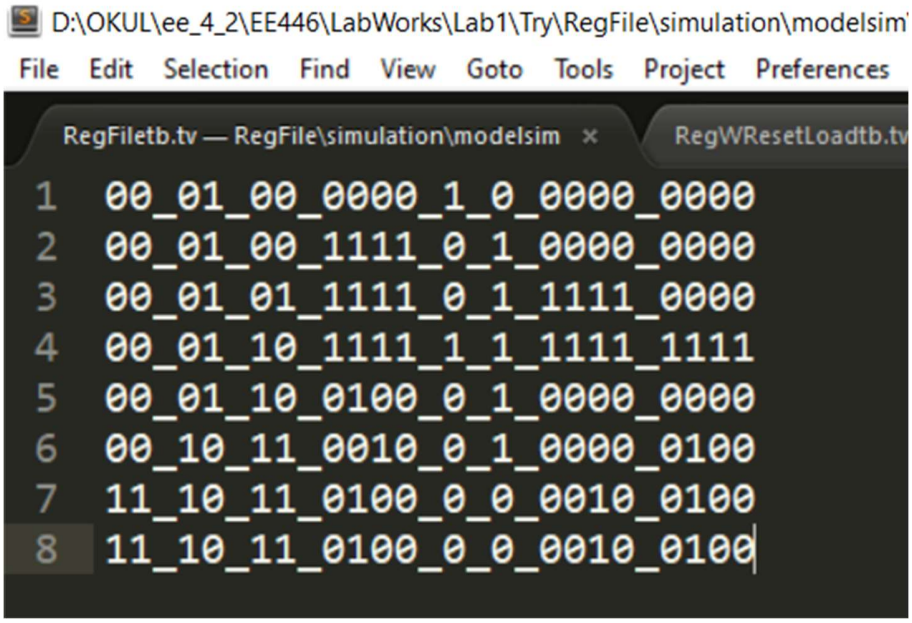


Figure 31: Test Vectors for RegFile

```
30      vectornum = 0;
31      errors = 0;
32    end
33    // Test Vectors are read and operated at pos edges
34      always @(posedge clktb)
35      begin
36          {A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rf1exp,Rf2exp}=testvectors[vectornum];
37          #2;
38      end
39    // Test Vector Output and Expect are compared at neg edges
40      always @(negedge clktb)
41      begin
42          if(Rf1tb != Rf1exp)
43          begin
44              $display("Error: input = %b %b %b %b %b %b %b %b", A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rf1tb,Rf2tb);
45              $display(" outputs = %b (%b expected)",Rf1tb,Rf1exp);
46              errors= errors +1;
47          end
48          if(Rf2tb != Rf2exp)
49          begin
50              $display("Error: input = %b %b %b %b %b %b %b %b", A1tb,A2tb,A3tb,WDtb,resettb,WEtb,Rf1tb,Rf2tb);
51              $display(" outputs = %b (%b expected)",Rf2tb,Rf2exp);
52              errors= errors +1;
53          end
54          vectornum = vectornum +1;
55          if(testvectors[vectornum] == 4'bx)
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.

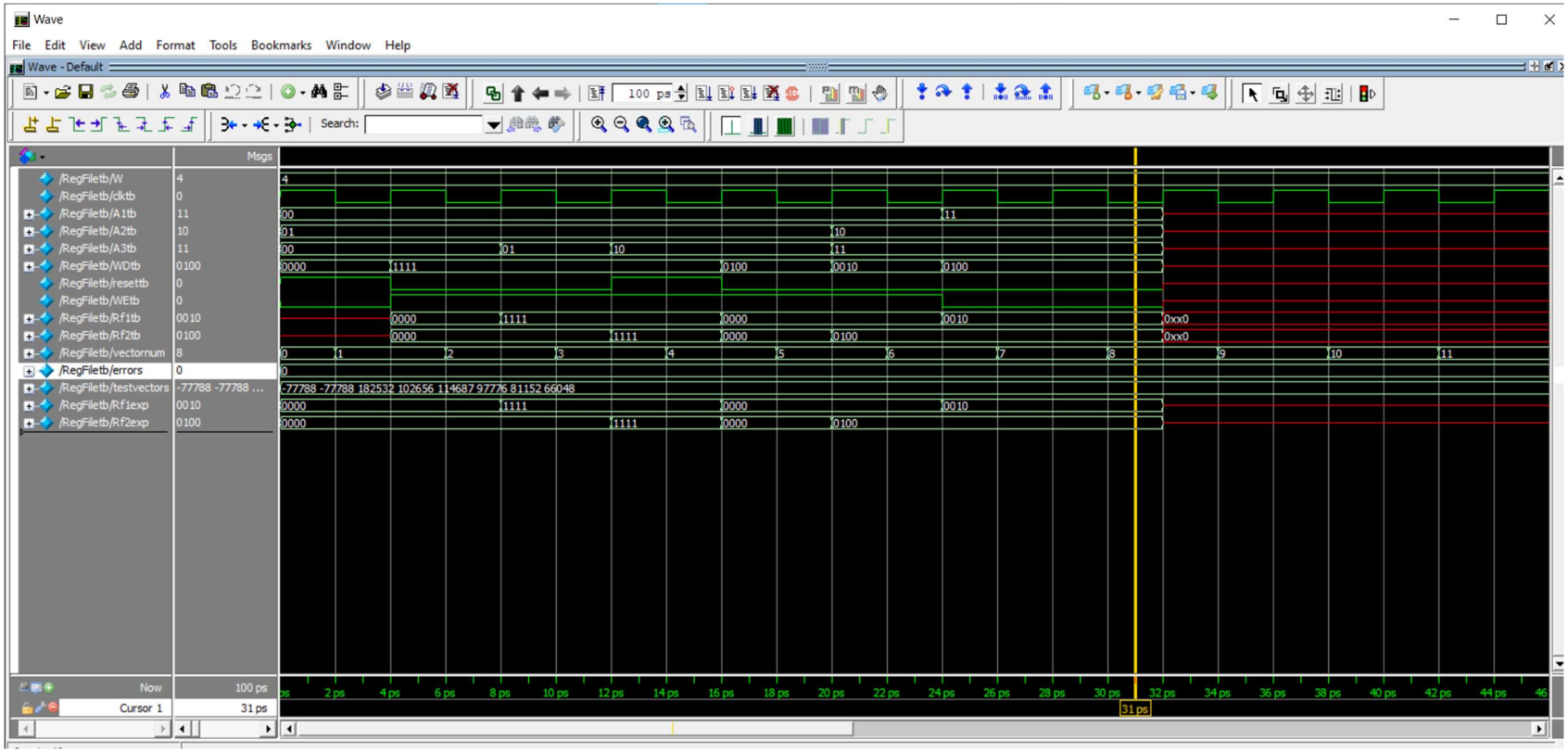
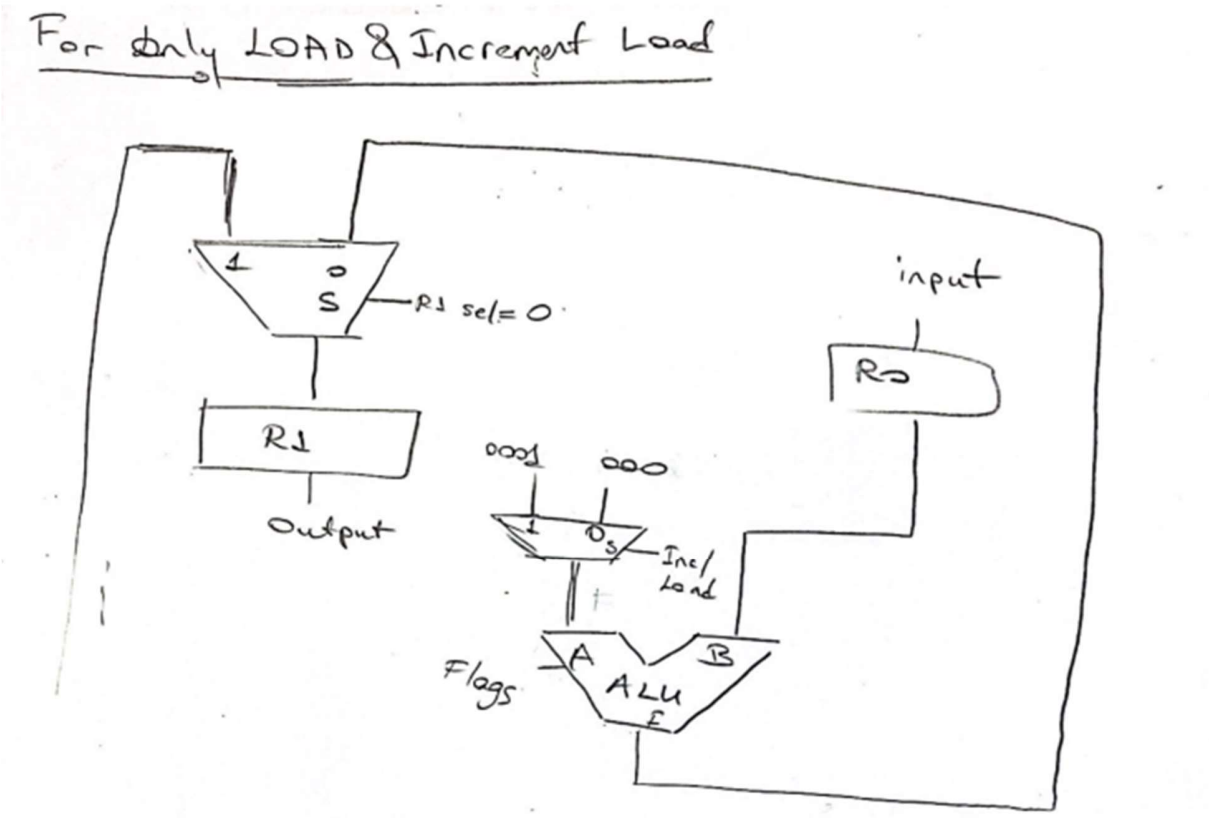


Figure 33: Test Result for RegFile

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:



- According the Inc/Load signal we can

  Inc / Load : 1 ⇒ R1 ← R0+1

  0 ⇒ R1 ← R0

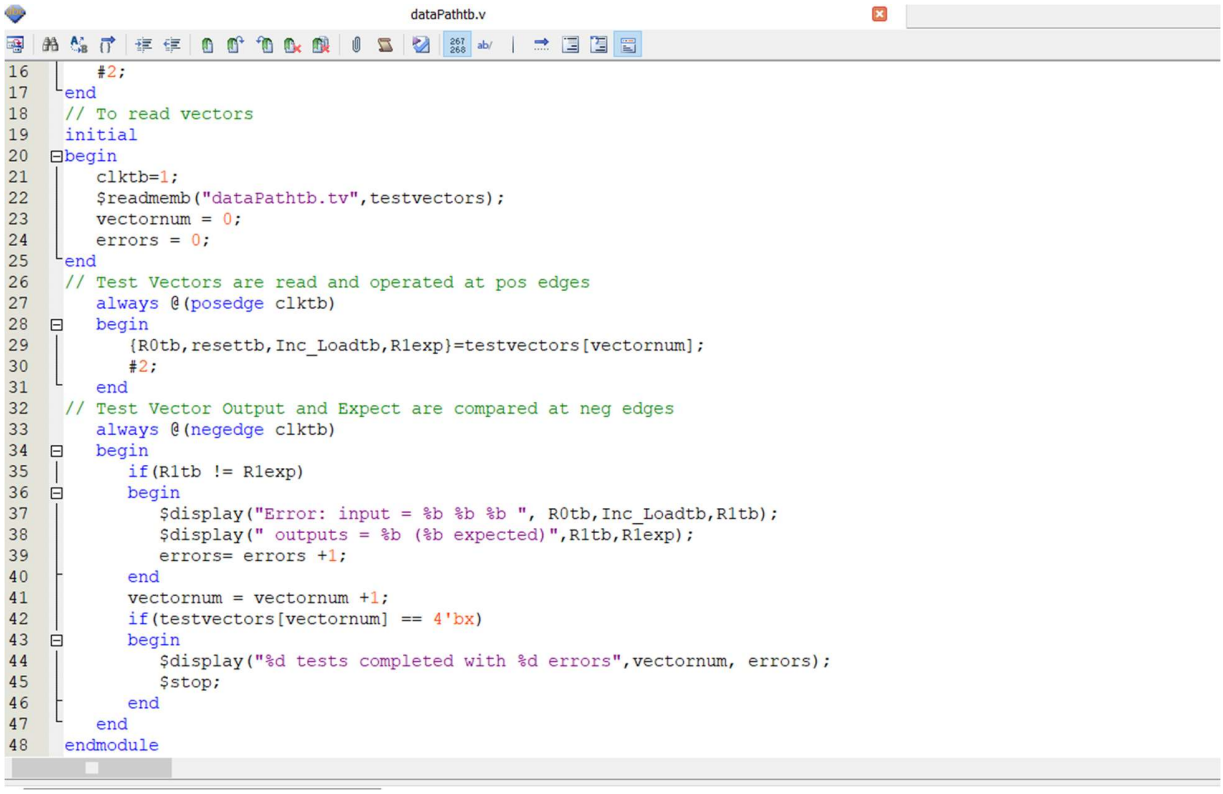


Figure 34: Test Bench for Data path

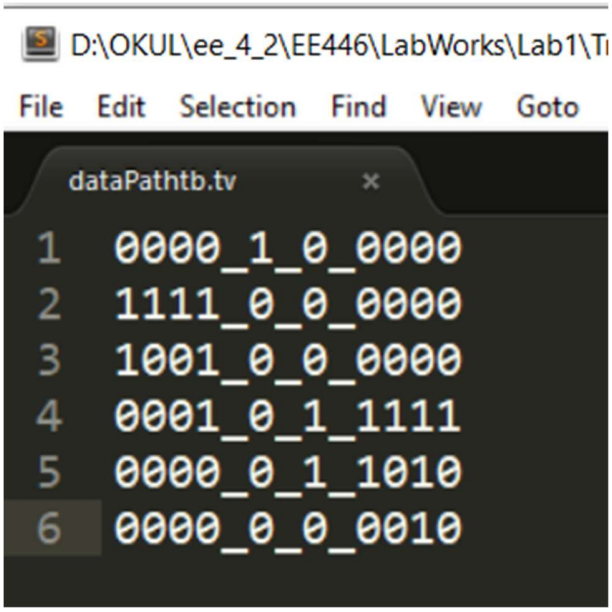


Figure 35: Test Vectors for Data path

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

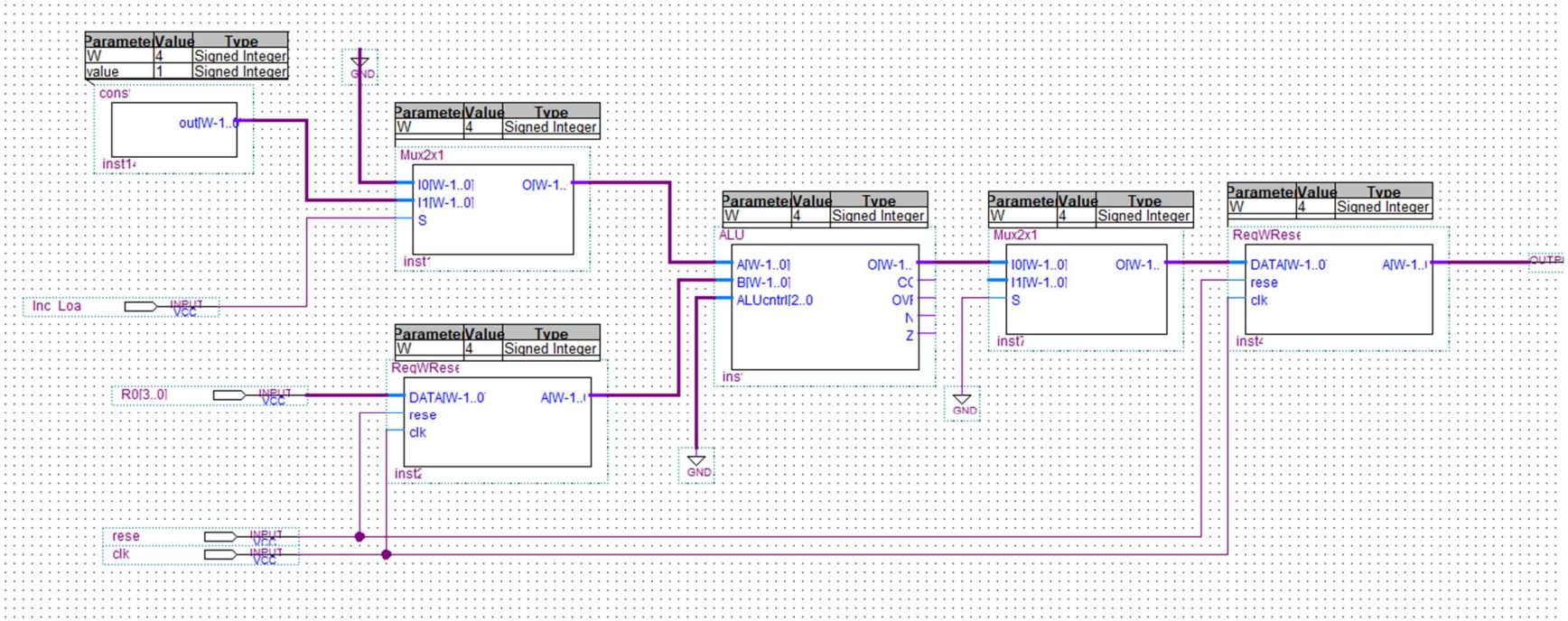


Figure 36: Schematic for Data path

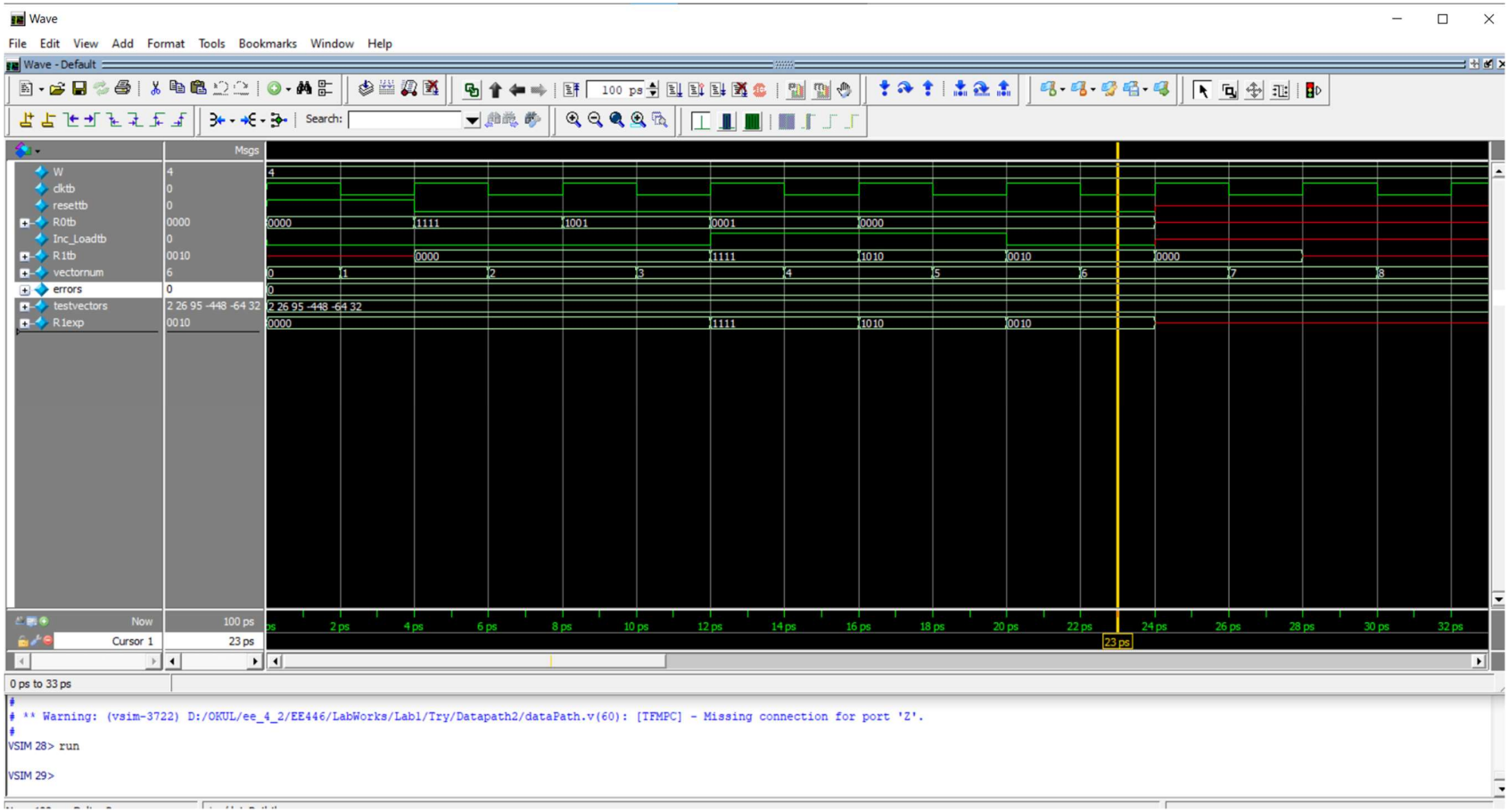
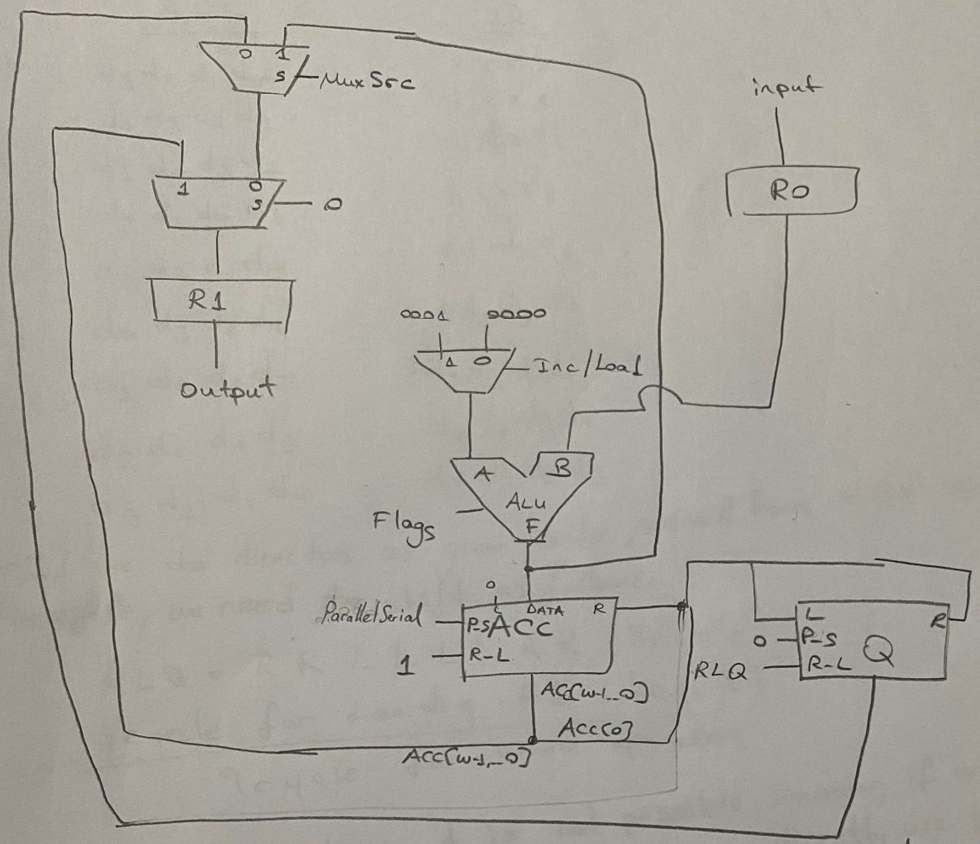


Figure 37: Result Test Vectors for Data path

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

For Load, 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 begging so that we can take R0 directly (Inc/Load=0). O.w it will be zero.

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

Acc	Q
$d_3 d_2 d_1 d_0$	$x x x x$
$d_0 d_3 d_2 d_1$	$d_0 x x x$
$d_1 d_0 d_3 d_2$	$d_1 d_0 x x$
$d_2 d_1 d_0 d_3$	$d_0 x x d_2$
$d_3 d_2 d_1 d_0$	$x x d_2 d_3$
$d_0 d_3 d_2 d_1$	$x d_2 d_3 d_0$
$d_1 d_0 d_3 d_2$	$d_2 d_3 d_0 d_1$
$d_2 d_1 d_0 d_3$	$d_2 d_2 d_3 d_0$
$d_3 d_2 d_1 d_0$	$d_3 d_2 d_2 d_3$

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

RLQ = R R L L L L R R (8cycle)
+ 1cycle for loading R1 (1cycle)
9cycle for whole operation

- As discussed above, it is not possible; however, if we were not restricted in this condition, we can directly use bits for R0 and we can get result of "$d_3 d_2 d_2 d_3$".

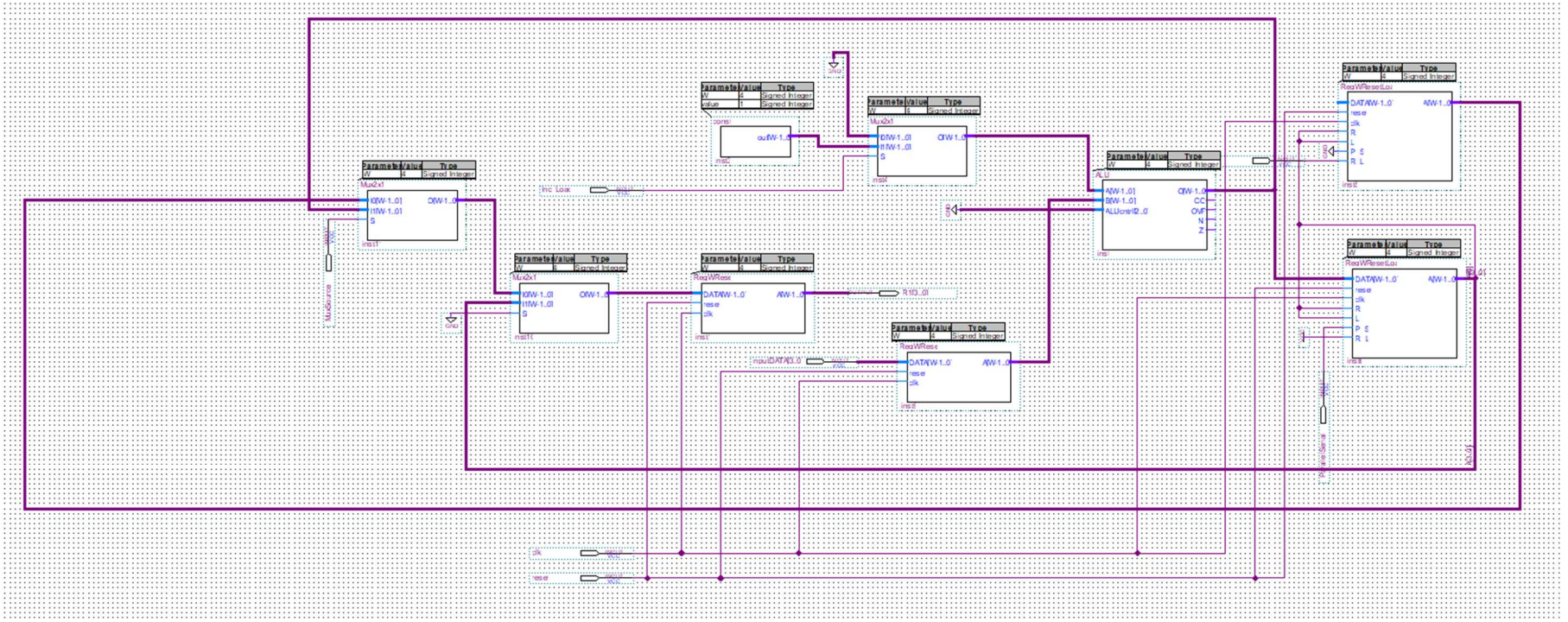


Figure 38: Schematic for full datapath

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

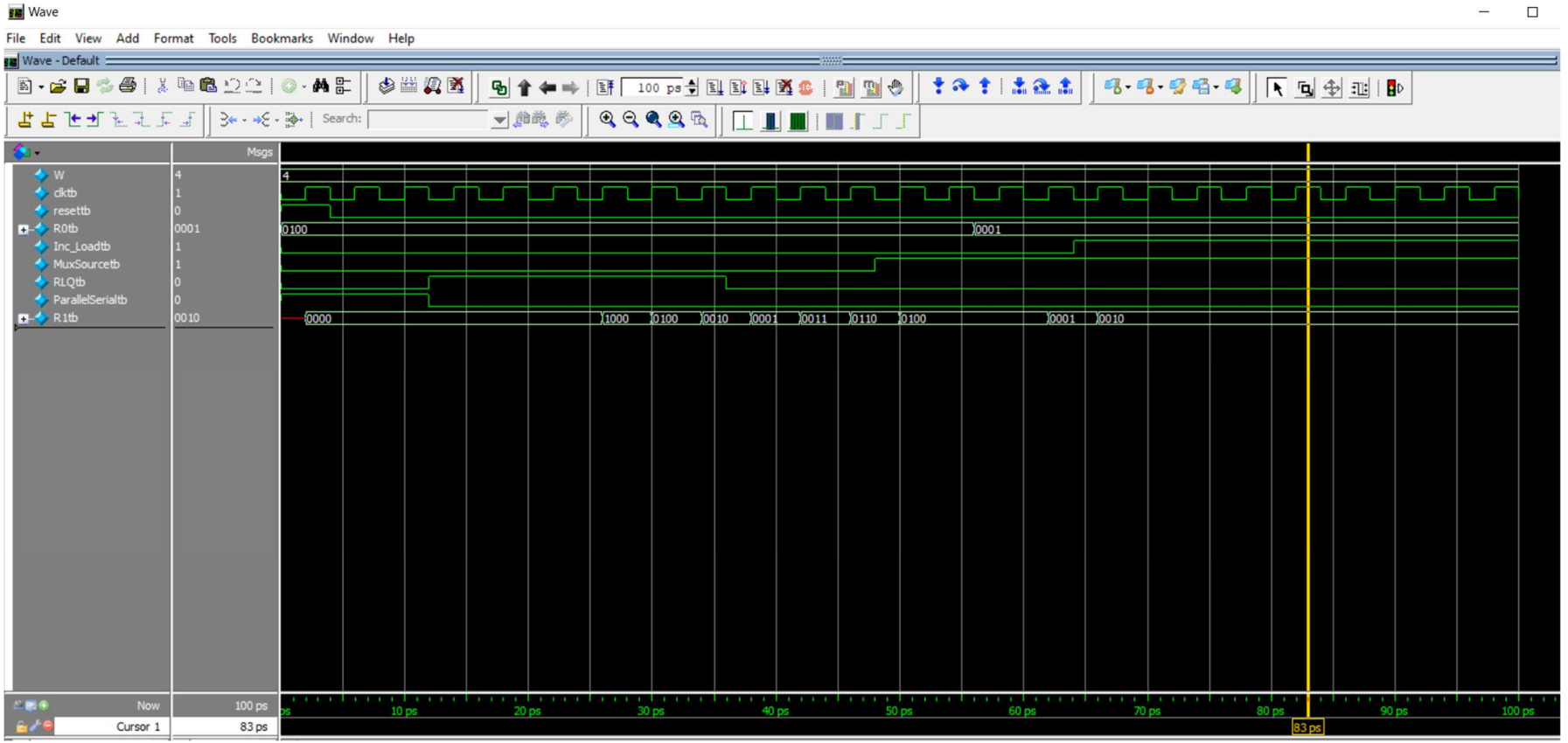


Figure 39: Simulation for full datapath

```
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] R1tb;

dataPath DUT(.clk(clktb),.reset(resettb),.Inc_Load(Inc_Loadtb),
             .inputDATA(R0tb),.MuxSource(MuxSourcetb),
             .RLQ(RLQtb),
             .ParallelSerial(ParallelSerialtb),.R1(R1tb));

//Clock Cycles
always
begin
   #2;
   clktb = ~clktb;

end

initial begin
//To do mirror operation
   clktb = 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 = Q
   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