Mehmet Ataş & 2304020 27.03.2022

EE446 PRELIMINARY WORK #1

Laboratory Work 1 - Warming Up for Computer Design

The goal of the first laboratory activity is to create a Verilog library comprised of the essential modules that will be utilized throughout the computer design process. Furthermore, simple data path design will be explored by creating simple architectures using the modules in the built library to do certain simple tasks.

This laboratory activity will familiarize you with developing modules with Verilog HDL and constructing architectures with schematic design. This laboratory activity is intended to familiarize students with the software—Quartus and Modelsim or cocotb—that will be used throughout the semester. Finally, the designs will be practiced on a development board, DE0-Nano, which is equipped with a field programmable gate array (FPGA) and many peripheral components such as switch inputs, general purposed I/O pins, LED outputs and so on.

1.2.2. Decoder:

```
//Mehmet Ataş 2304020
//2x4 decoder
module decoder
(
    input [1:0] X,
    output [3:0] Y
);
assign Y[0] = ~X[0] & ~X[1];
assign Y[1] = X[0] & ~X[1];
assign Y[2] = ~X[0] & X[1];
assign Y[3] = X[0] & X[1];
endmodule

```

Figure 1. Implementation of a 2 to 4 decoder with Verilog

```
//Mehmet Ataş 2304020
module tb();
wire [3:0] Y;
reg [1:0] X;
decoder dut(.X(X), .Y(Y));
initial begin
    X = 2'b00;
    #20;
    X = 2'b01;
    #20;
    X = 2'b10;
    #20;
    X = 2'b11;
    #20;
end

endmodule

```

Figure 2. Test bench module to test my implementation of the decoder

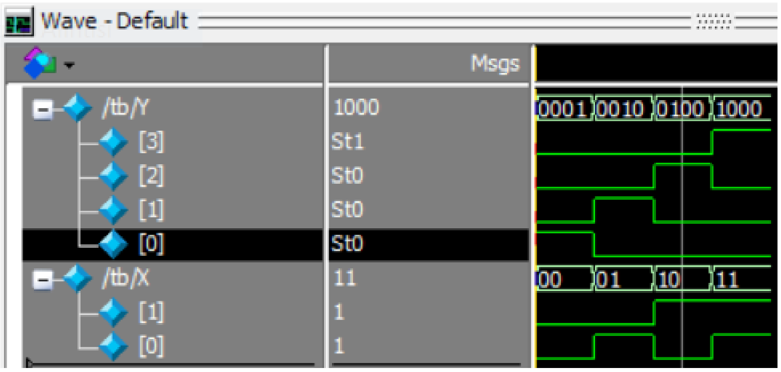


Figure 3: Simulation of the decoder

Table 1: Decoder Vector Table

A[1]	A[0]	D[0]	D[1]	D[2]	D[3]
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The results are the same as in the vector table as expected.

1.2.3. Multiplexers:

W-bit 2 to 1 multiplexer, (W=32);

```
//Mehmet Ataş 2304020
module mux2x1
#(parameter W = 32)
(
    input [W-1:0] I0, I1,
    input S,
    output [W-1:0] Q
);
    assign Q = S ? I1 : I0;
endmodule

```

Figure 4. Implementation of a 32 bit 2 to 1 multiplexer with Verilog

```
//Mehmet Ataş 2304020
module mux2x1TB #(parameter W = 32)();
reg [W-1:0] I0, I1;
reg S;
wire [W-1:0] Q;
mux2x1 dut(.I0(I0), .I1(I1), .S(S), .Q(Q));
initial begin
    I0 = 32'h00000000;
    I1 = 32'hFFFFFFFF;
    S = 1'b0;
    #10;
    S = 1'b1;
    #10;
end
endmodule
```

Figure 5. Test bench module to test my implementation of the 32 bit 2 to 1 multiplexer

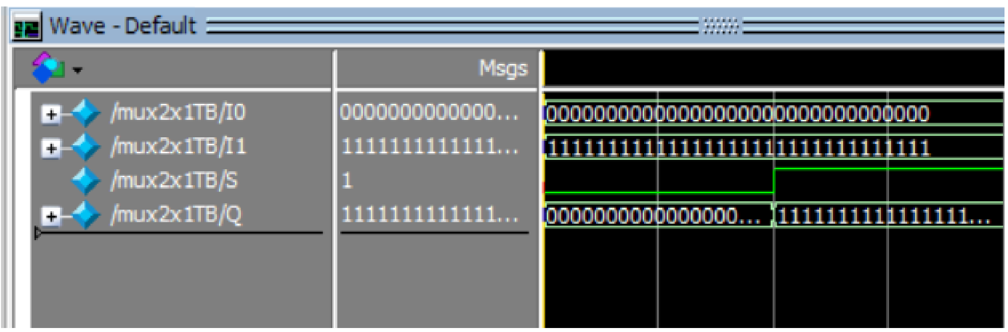


Figure 6: Simulation of the 32 bit 2 to 1 multiplexer

Table 2: 2x1 Mux Vector Table

I0	*I1*	*S*	*Q*
32'h00000000	*32'h11111111*	*0*	*32'h00000000*
32'h00000000	*32'h11111111*	*1*	*32'h11111111*

The results are the same as in the vector table as expected.

W-bit 4 to 1 multiplexer, (W=32);

```
//Mehmet Ataş 2304020
module mux4x1
#(parameter W = 32)
(
    input [1:0] S,
    input [W-1:0] I0, I1, I2, I3,
    output [W-1:0] Q
);
assign Q = S[1] ? (S[0] ? I3 : I2) : (S[0] ? I1 : I0);
endmodule
```

Figure 7. Implementation of a 32 bit 4 to 1 multiplexer with Verilog

```verilog
//Mehmet Ataş 2304020
module mux4x1TB #(parameter W = 32)();
reg [W-1:0] I0, I1, I2, I3;
reg [1:0] S;
wire [W-1:0] Q;
mux4x1 dut(.I0(I0), .I1(I1), .I2(I2), .I3(I3), .S(S), .Q(Q));
initial begin
    I0 = 32'h11111111;
    I1 = 32'h22222222;
    I2 = 32'h44444444;
    I3 = 32'h88888888;
    S = 2'b00;
    #10;
    S = 2'b01;
    #10;
    S = 2'b10;
    #10;
    S = 2'b11;
    #10;
end
endmodule
```

Figure 8. Test bench module to test my implementation of the 32 bit 4 to 1 multiplexer

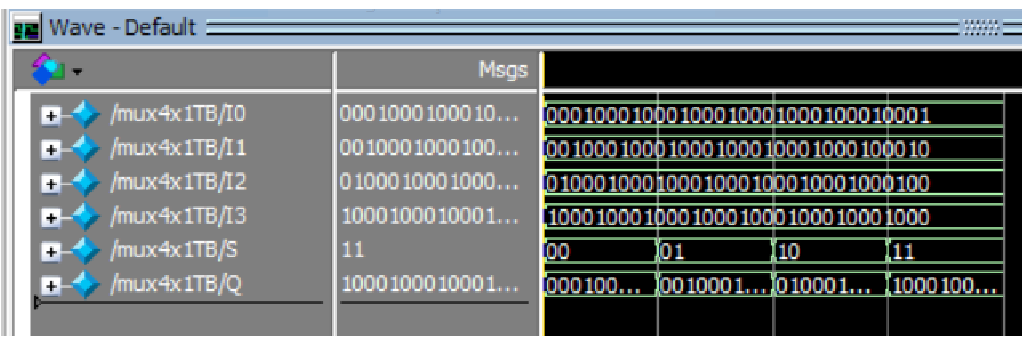


Figure 9: Simulation of the 32 bit 4 to 1 multiplexer

I3	I2	I1	I0	S[1]	S[0]	Q
32’h88888888	32’h44444444	32’h22222222	32’h11111111	0	0	32’h11111111
32’h88888888	32’h44444444	32’h22222222	32’h11111111	0	1	32’h22222222
32’h88888888	32’h44444444	32’h22222222	32’h11111111	1	0	32’h44444444
32’h88888888	32’h44444444	32’h22222222	32’h11111111	1	1	32’h88888888

The results are the same as in the vector table as expected.

1.2.4. Arithmetic Logic Unit (ALU):

My method to detect overflow is simple . If the operands signs are the same but the result sign is different, there is an overflow.

```
//Mehmet Ataş
module ALU
#(parameter W = 32)
(
    input [2:0] ctrl,
    input [W-1:0] A, B,
    output reg [W-1:0] Q,
    output reg CO, OVF, N, Z
);
    always @(*)
    begin
        case(ctrl)
            3'b000:  begin
                        //Addition operation
                        {CO, Q} = A + B;
                        N = Q[W-1];
                        //MSB is the sign bit
                        if (Q == 0)
                            Z = 1;
                        else
                            Z = 0;
                        // Check overflow
                        if ((A[W-1] == 1 && B[W-1] == 1 && Q[W-1] == 0) || (A[W-1] == 0 && B[W-1] == 0 && Q[W-1] == 1))
                            OVF = 1;
                        else
                            OVF = 0;
                    end
            3'b001:  begin
                        //Subtraction operation A-B
                        {CO, Q} = A - B;
                        N = Q[W-1];
                        //MSB is the sign bit
                        if (Q == 0)
                            Z = 1;
                        else
                            Z = 0;
                        //Check overflow
                        if ((A[W-1] == 1 && B[W-1] == 1 && Q[W-1] == 0) || (A[W-1] == 0 && B[W-1] == 0 && Q[W-1] == 1))
                            OVF = 1;
                        else
                            OVF = 0;
                    end
            3'b010:  begin
                        //Subtraction operation B-A
                        {CO, Q} = B - A;
                        N = Q[W-1];
                        //MSB is the sign bit
                        if (Q == 0)
                            Z = 1;
                        else
                            Z = 0;
                        // check overflow
                        if ((A[W-1] == 1 && B[W-1] == 1 && Q[W-1] == 0) || (A[W-1] == 0 && B[W-1] == 0 && Q[W-1] == 1))
                            OVF = 1;
                        else
                            OVF = 0;
                    end
            3'b011:  begin
                        //Bit Clear
                        Q = A & ~B;
                        N = Q[W-1];
                        //MSB is the sign bit
                        if (Q == 0)
                            Z = 1;
                        else
                            Z = 0;
                        CO = 0;
                        OVF = 0;
                    end
```

```
                end
        3'b100: begin
                    //AND
                    Q = A & B;
                    N = Q[W-1];
                    //MSB is the sign bit
                    if (Q == 0)
                        Z = 1;
                    else
                        Z = 0;
                    CO = 0;
                    OVF = 0;
                end
        3'b101: begin
                    //OR
                    Q = A | B;
                    N = Q[W-1];
                    //MSB is the sign bit
                    if (Q == 0)
                        Z = 1;
                    else
                        Z = 0;
                    CO = 0;
                    OVF = 0;
                end
        3'b110: begin
                    //EXOR
                    Q = A ^ B;
                    N = Q[W-1];
                    //MSB is the sign bit
                    if (Q == 0)
                        Z = 1;
                    else
                        Z = 0;
                    CO = 0;
                    OVF = 0;
                end
        3'b111: begin //exnor
                    Q = ~(A ^ B);
                    N = Q[W-1];
                    //MSB is the sign bit
                    if (Q == 0)
                        Z = 1;
                    else
                        Z = 0;
                    CO = 0;
                    OVF = 0;
                end
    endcase
  end
endmodule
```

Figure 10. Implementation of ALU with Verilog

```verilog
//Mehmet Ataş
module ALUTB #(parameter W = 32)();
   reg [2:0] ctrl;
   reg [W-1:0] A, B, Qexp;
   wire [W-1:0] Q;
   wire CO, OVF, N, Z;
   reg [W-1:0] vectornum, errors;
   reg [3*W-1:0] testvectors [32:0];
   reg [2:0] ctrltv [32:0];
   ALU dut(.A(A), .B(B), .ctrl(ctrl), .CO(CO), .OVF(OVF), .N(N), .Z(Z), .Q(Q));
   initial begin
      $readmemh("ALU.tv", testvectors);
      $readmemb("ctrl.tv",ctrltv);
      vectornum = 0; errors = 0;
      #10;
      while (ctrltv[vectornum] !== 3'bxxx)
      begin
         {A, B, Qexp} = testvectors[vectornum];
         ctrl = ctrltv[vectornum];
         vectornum = vectornum + 1;
         #10;
         if (Qexp != Q)
         begin
            $display("Error: inputs = %h, %h", A, B);
            $display(" outputs = %h (%h expected)", Q, Qexp);
            errors = errors + 1;
         end
         #10;
      end
      $display("%d tests completed with %d errors", vectornum, errors);
   end

endmodule
```

Figure 11. Test bench module to test my implementation of ALU

Table 4: ALU Vector Table

Ctrl	A	B	CO	OVF	N	Z	Q[31:0]
ADD							
000	32’h00000001	32’h00000001	0	0	0	0	32’h00000002
000	32’h00000001	32’h00000002	0	0	0	0	32’h00000003
000	32’h0000FFFF	32’h00000001	0	0	0	0	32’h00010000
000	32’hFFFFFFFF	32’hFFFFFFFF	1	0	1	0	32’hFFFFFFFE
SUBAB							
001	32’h00000001	32’h00000001	0	0	0	1	32’h00000000
001	32’h00000002	32’h00000001	0	0	0	0	32’h00000001
001	32’h00000001	32’h00000002	1	1	1	0	32’hFFFFFFFF
001	32’h0000000A	32’h0000006E	1	1	1	0	32’hFFFFFF9C

SUBBA

010	32’h00000001	32’h00000001	0	0	0	1	32’h00000000
010	32’h00000001	32’h00000002	0	0	0	0	32’h00000001
010	32’h00000002	32’h00000001	1	1	1	0	32’hFFFFFFFF
010	32’h0000006E	32’h0000000A	1	1	1	0	32’hFFFFFF9C

BITC

011	32’hFFFFFFFF	32’h55555555	0	0	1	0	32’hAAAAAAAA
011	32’h00000001	32’h00000001	0	0	0	1	32’h00000000
011	32’hFFFFFFFF	32’h00000000	0	0	1	0	32’hFFFFFFFF
011	32’hFFFF0000	32’h0000FFFF	0	0	1	0	32’hFFFF0000

AND

100	32’hFFFFFFFF	32’h55555555	0	0	0	0	32’h55555555
100	32’h00000001	32’h00000001	0	0	0	0	32’h00000001
100	32’hFFFFFFFF	32’h00000000	0	0	0	1	32’h00000000
100	32’hFFFF0000	32’h0000FFFF	0	0	0	1	32’h00000000

OR

101	32’hFFFFFFFF	32’h55555555	0	0	1	0	32’hFFFFFFFF
101	32’h00000001	32’h00000001	0	0	0	0	32’h00000001
101	32’hFFFFFFFF	32’h00000000	0	0	1	0	32’hFFFFFFFF
101	32’hFFFF0000	32’h0000FFFF	0	0	1	0	32’Hffffffff

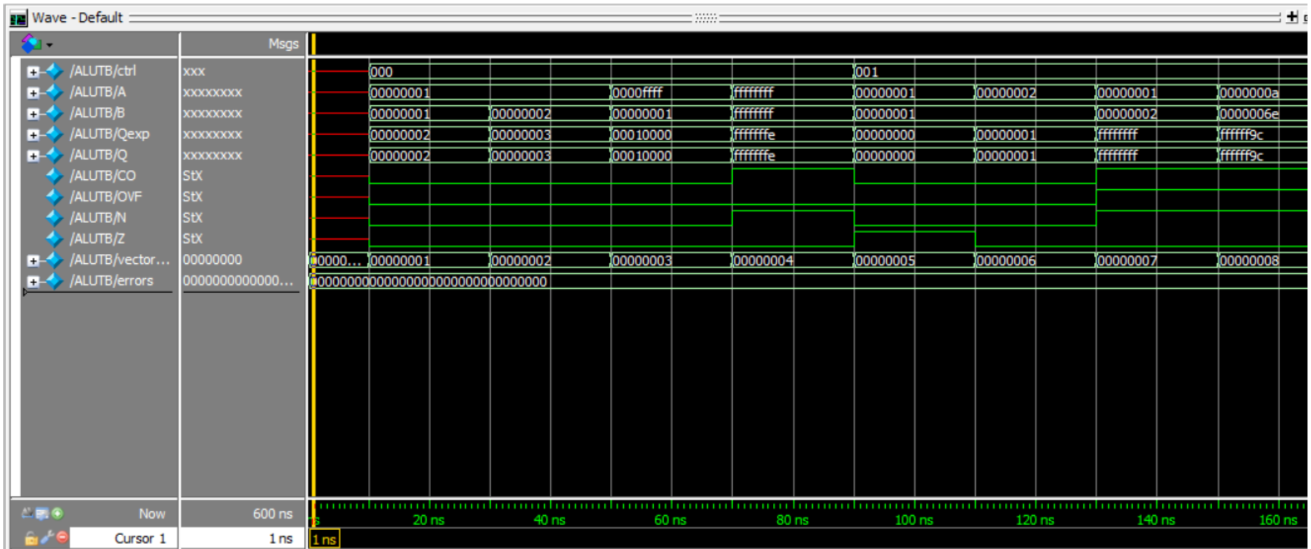
EXOR

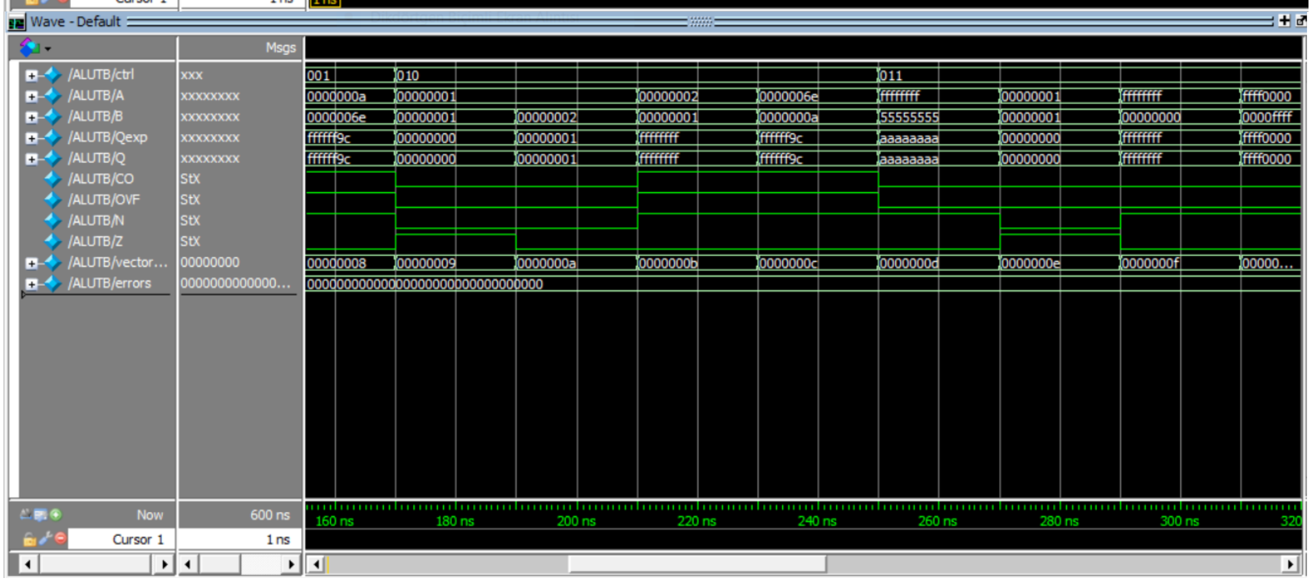
110	32’hFFFFFFFF	32’h55555555	0	0	1	0	32’hAAAAAAAA
110	32’h00000001	32’h00000001	0	0	0	1	32’h00000000
110	32’hFFFFFFFF	32’h00000000	0	0	1	0	32’hFFFFFFFF
110	32’hFFFF0000	32’h0000FFFF	0	0	1	0	32’hFFFFFFFF

EXNOR

111	32’hFFFFFFFF	32’h55555555	0	0	0	0	32’h55555555
111	32’h00000001	32’h00000001	0	0	1	0	32’hFFFFFFFF
111	32’hFFFFFFFF	32’h00000000	0	0	0	1	32’h00000000
111	32’hFFFF0000	32’h0000FFFF	0	0	0	1	32’h00000000

Mehmet Ataş & 2304020 27.03.2022





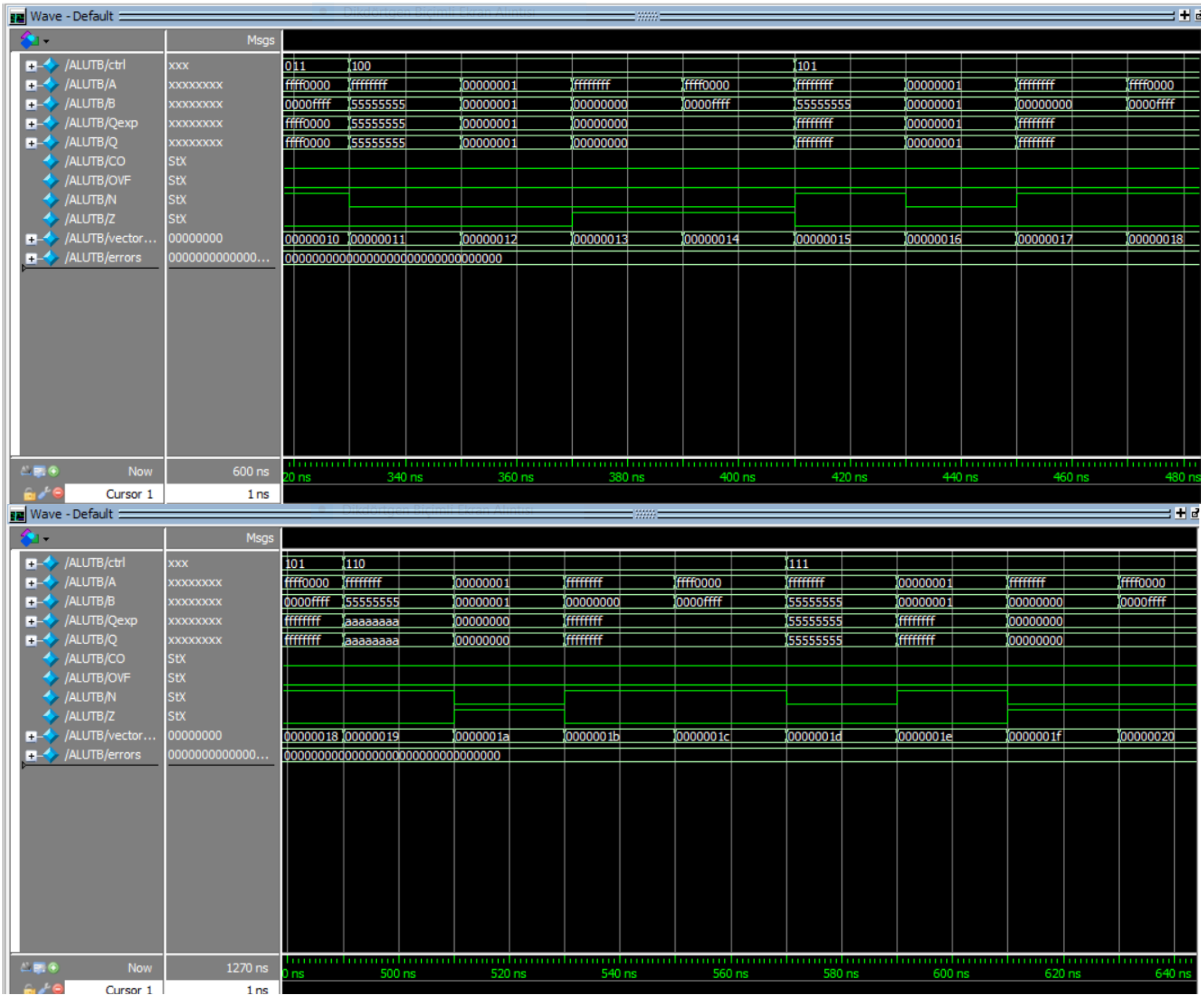


Figure 12: Simulation of the ALU

1.2.5. Registers:

1.2.5.1. Simple Register with Synchronous Reset;

```
//Mehmet Ataş 2304020
module regSR
#(parameter W = 32)
(
    input reset,
    input clk,
    input [W-1:0] D,
    output reg [W-1:0] Q
);
//Register with synchronous reset
always @ (posedge clk)
    if (reset)
        Q <= 0;
    else
        Q <= D;
endmodule
```

Figure 13. Implementation of the simple register with synchronous reset with Verilog

```
//Mehmet Ataş 2304020
module regSRTB #(parameter W = 32)();
reg [W-1:0] D, Qexp;
reg clk, reset;
wire [W-1:0] Q;
reg [2*W-1:0] testvectors [15:0];
reg [W-1:0] vectornum, errors;


regSR dut(.clk(clk), .reset(reset), .D(D), .Q(Q));

always begin
    //creation of the clock
    clk = 0; #10; clk = 1; #10;
end

initial begin
    //reading the testvectors
    $readmemh("regSR.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #50; reset = 0;
    #50;
    reset = 1;
    //reset is done here.
    #100;
    reset = 0;
end

always @(posedge clk)   //write a testvector into input and expected output at posedge + 1ns
begin
    #1; {D, Qexp} = testvectors[vectornum];
end

always @(negedge clk)   //compare the results at negedge
begin
```

```verilog
always @(negedge clk)    //compare the results at negedge
begin
    if (~reset)
    begin
        if (Q != Qexp) //if output is not equal to expected output
        begin
            $display("Error: inputs = %h", D);
            $display(" outputs = %h (%h expected)", Q, Qexp);
            errors = errors + 1;
        end
        vectornum = vectornum + 1;
        if (testvectors[vectornum] === 64'hxxxxxxxxxxxxxxxx)  //if finished
        begin
            $display("%d tests completed with %d errors", vectornum, errors);
            $stop();
        end
    end
end
endmodule
```

Figure 14. Test bench module to test my implementation of the simple register with synchronous reset

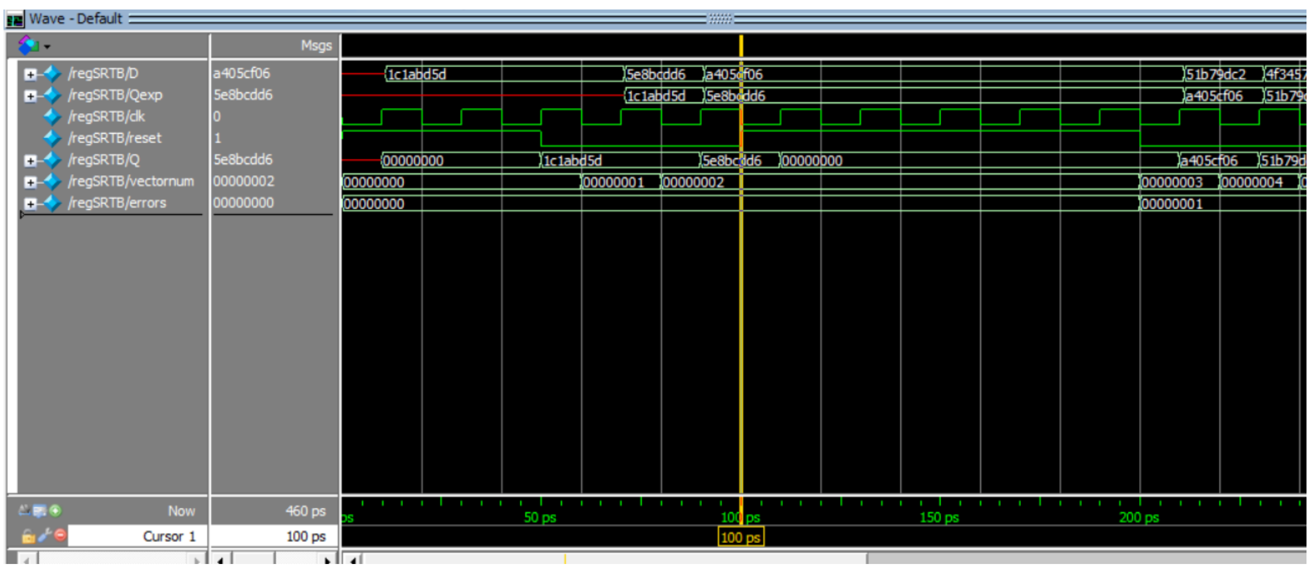


Figure 15: Simulation of the simple register with synchronous reset

I tested my concept with random numbers and added a reset in the midst of the simulation. I'm simply providing the simulation excerpt where I put the reset. As predicted, the output of the register is dragged down to 0 at the very first posedge of clock when reset is entered. Basically, I implement a positive edge triggered register with parallel load and synchronous reset. If the reset signal is 1, the contents of the register is cleared at the next rising edge of the clock. If the reset signal is 0, the contents of the register is loaded with the input data at the next rising edge of the clock.

1.2.5.2. Register with synchronous reset and write enable;

```
//Mehmet Ataş 2304020
module regSRWEN
#(parameter W = 32)
(
    input reset,
    input write,
    input clk,
    input [W-1:0] D,
    output reg [W-1:0] Q
);
//Register with synchronous reset and write enable
    always @ (posedge clk)
        if (reset)
            Q <= 0;
        else if (write)
            Q <= D;
endmodule

```

Figure 16. Implementation of the Register with synchronous reset and write enable with Verilog

```
module regSRWENTB #(parameter W = 32)();
   reg [W-1:0] D, Qexp;
   reg clk, reset, write;
   wire [W-1:0] Q;
   reg [W-1:0] vectornum, errors;
   reg [2*W-1:0] testvectors [15:0];

   regSRWEN dut(.clk(clk), .reset(reset), .write(write), .D(D), .Q(Q));

   always begin
      clk = 0; #10; clk = 1; #10;
   end

   initial begin  //read the testvectors
      $readmemh("regSRWEN.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #20; reset = 0;
      write = 0; # 20; write = 1;
      #40;
      write = 0;
      //write disabled
      #20;
      reset = 1;
      //reset enabled
      #20;
      write = 1;
      //write enabled
      #20;
      reset = 0;
      //reset disabled
   end

   always @(posedge clk)
   //write a testvector into input and expected output at posedge + 1ns
   begin
```

```verilog
    begin
       #1; {D, Qexp} = testvectors[vectornum];
    end

    always @(negedge clk)
    //compare the results at negedge
    begin
       if (~reset)
       begin
          if (Q != Qexp)
          //if output is not equal to expected output
          begin
             $display("Error: inputs = %h", D);
             $display(" outputs = %h (%h expected)", Q, Qexp);
             errors = errors + 1;
          end
          vectornum = vectornum + 1;
          if (testvectors[vectornum] === 64'hxxxxxxxxxxxxxxxx)
          begin
             $display("%d tests completed with %d errors", vectornum, errors);
             $stop();
          end
       end
    end
endmodule
```

Figure 17. Test bench module to test my implementation of the Register with synchronous reset and write enable

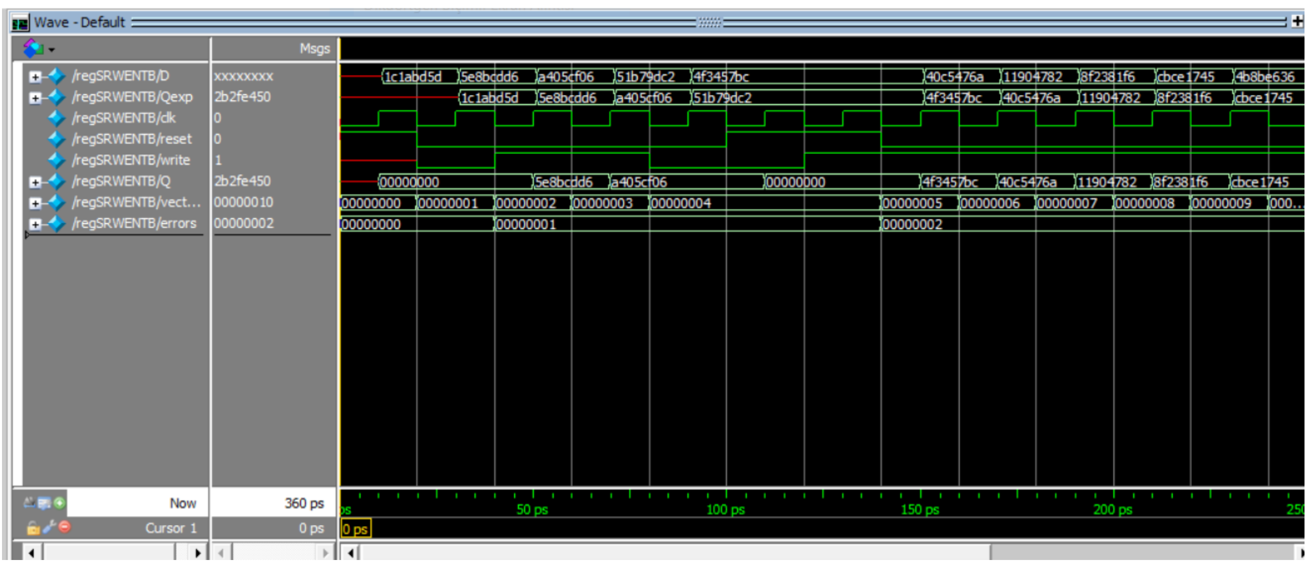


Figure 18: Simulation of the Register with synchronous reset and write enable

When we bring write enable down to zero, we find that the output does not change until we pull reset up. When the reset button is pressed, the output remains at 0. The fact that the write enable signal has no impact when the reset is pushed up is also proven. When the reset button was pressed, the regular load procedure resumed. As a result, the design has been validated. Basically, I implement a positive edge triggered register with parallel load, write enable and synchronous reset. If the reset signal is 1, the contents of the register is cleared at the next rising edge of the clock. If the reset signal is 0 and write enable signal is 1, the contents of the register is loaded with the input data at the next rising edge of the clock. Finally, if the reset signal is 0 and write enable signal is 0, the register retains its content.

1.2.5.3. Shift Register with Parallel and Serial Load;

```verilog
//Mehmet Ataş
module shiftregister
#(parameter W = 32)
(
   input shift,
   input clk,
   input reset,
   input load,
   input left,
   input right,
   input [W-1:0] D,
   output reg [W-1:0] Q
);

   always @ (posedge clk)
      if (reset)
         Q <= 0;
      else if (load)
      //if load = 1, then parallel load.
         Q <= D;
      else if (shift)
      // if shift = 1, then shift right.  A[W - 1] ← Serial Input Left
      begin
         Q[W-2:0] <= Q[W-1:1];
         Q[W-1] <= left;
      end
      else if (~shift)
      // if shift = 0, then shift left.  A[0] ← Serial Input Right
      begin
         Q[W-1:1] <= Q[W-2:0];
         Q[0] <= right;
      end
endmodule
```

Figure 19. Implementation of the Shift Register with Parallel and Serial Load with Verilog

```verilog
//Mehmet Ataş
module shiftregisterTB #(parameter W = 32)();
   wire [W-1:0] Q;
   reg [W-1:0] D, Qexp;
   reg clk, load, shift, reset, left, right;
   reg [W-1:0] vectornum, errors;
   reg [2*W-1:0] testvectors [15:0];
   shiftregister dut(.clk(clk), .load(load), .shift(shift), .reset(reset), .D(D), .left(left), .right(right), .Q(Q));
   always begin
   //creation of the clock
      clk = 0; #10; clk = 1; #10;
   end
   initial begin
      //reading the testvector
      $readmemh("shiftregister.tv", testvectors);
      vectornum = 0; errors = 0;
      load = 1; shift = 0;
      //parallel load
      left = 1; right = 1;
      reset = 1; #20; reset = 0;
      #200;
      load = 0; shift = 1;
      //shift right 1
      #20;
      left = 0;
      //shift left 0
      #20;
      shift = 0;
      //shift left 1
      #20;
      right = 0;
      //shift left 0
      #20;
      load = 1;
      //parallel load
      #100;
      load = 0;
      right = 0; left = 0;
      //shift left 0
      #20;
      shift = 1;
      //shift right 0
      left = 0;
   end
   always @(posedge clk)
   //write a testvector into input and expected output
   begin
      #1; {D, Qexp} = testvectors[vectornum];
   end
   always @(negedge clk)
   //compare the results
   begin
      if (~reset)
      begin
         if (Q != Qexp)
         begin
            $display("Error: inputs = %h", D);
            $display(" outputs = %h (%h expected)", Q, Qexp);
            errors = errors + 1;
         end
         vectornum = vectornum + 1;
         if (testvectors[vectornum] === 64'hxxxxxxxxxxxxxxxx)
         begin
            $display("%d tests completed with %d errors", vectornum, errors);
            $stop();
         end
      end
   end
endmodule
```

Figure 20. Test bench module to test my implementation of the Shift Register with Parallel and Serial Load

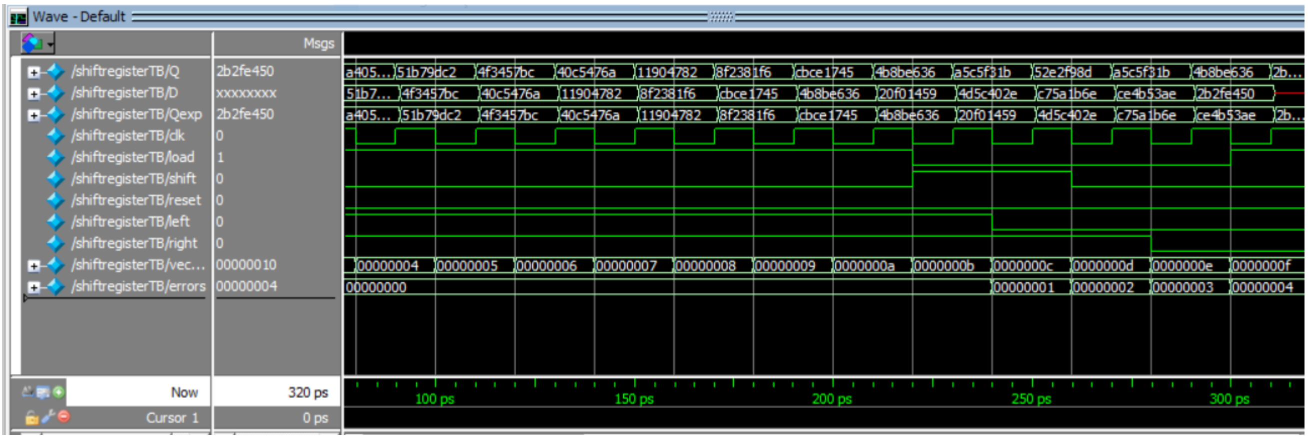


Figure 21. Simulation of the Shift Register with Parallel and Serial Load

When load is pulled down to 0, I am shifting right (shift = 1) and left input is 1. Output was 32'h 4b8be636 before shift, after shift it is 32'h a5c5f31b which is correct (can be calculated with calculator easily). After that, shift is pulled down to 0, and right input is 1. Output was 32'h a5c5f31b before shift, after shift it is 32'h 4b8be637 which is correct. Basically, I implement a positive edge triggered shift register with parallel and serial load. The register has three control signals: Synchronous reset, parallel/serial load select, shift left/right select. There are 3 input sources to the register: W-bit parallel input, 1-bit serial input left and 1-bit serial input right. If the reset signal is 1, the contents of the register is cleared at the next rising edge of the clock. If the reset signal is 0, the operation of the register is determined according to the parallel/serial and shift right/left select signals. If the parallel/serial select signal is 1, the contents of the register is loaded with the parallel input at the next rising edge of the clock. If the parallel/serial select signal is 0, the contents of the register is shifted and the left most or right most bit of the register is loaded with the corresponding serial input at the next rising edge of the clock. For the serial input operation, if the shift right/left select signal is 1, the contents are shifted right and the most significant bit is loaded with the serial input left; if the shift right/left select signal is 0, the contents are shifted left and the least significant bit is loaded with the serial input right.

My vector table was the same for 3 different 32-bit registers which is…

1c1abd5d_xxxxxxxx	5e8bcdd6_1c1abd5d	a405cf06_5e8bcdd6	51b79dc2_a405cf06
4f3457bc_51b79dc2	40c5476a_4f3457bc	11904782_40c5476a	8f2381f6_11904782
cbce1745_8f2381f6	4b8be636_cbce1745	20f01459_4b8be636	4d5c402e_20f01459
c75a1b6e_4d5c402e	ce4b53ae_c75a1b6e	2b2fe450_ce4b53ae	xxxxxxxx_2b2fe450

1.3. Register File

I design and sketch a datapath for a register file design using your decoder, multiplexer and register modules. I used RTL viewer for convenience.

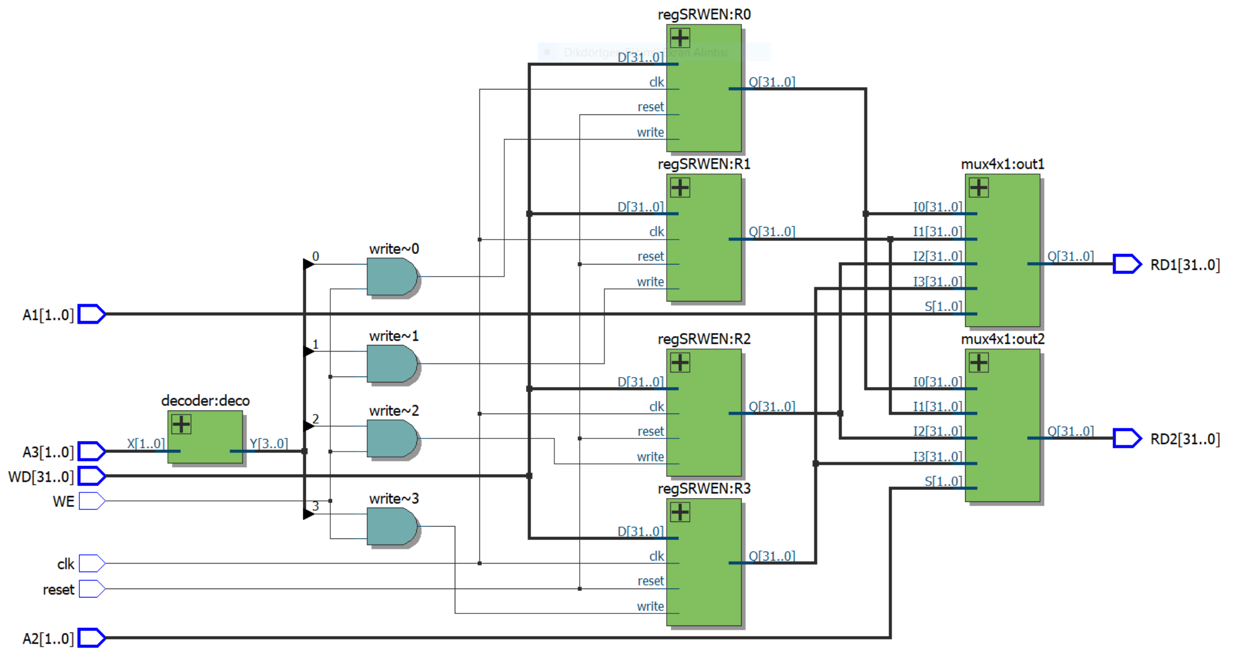


Figure 22. Sketch of the datapath for a register file

```
//Mehmet Ataş
module registerfile
#(parameter W = 32)
(
    input clk,
    input WE,
    input reset,
    input [W-1:0] WD,
    input [1:0] A1,A2,A3,
    output [W-1:0] RD1, RD2
);

    wire [W-1:0] R0_out, R1_out, R2_out, R3_out;
    wire [3:0] write;
    wire [3:0] write_deco;

    regSRWEN R0(.clk(clk), .reset(reset), .write(write[0]), .D(WD), .Q(R0_out));
    regSRWEN R1(.clk(clk), .reset(reset), .write(write[1]), .D(WD), .Q(R1_out));
    regSRWEN R2(.clk(clk), .reset(reset), .write(write[2]), .D(WD), .Q(R2_out));
    regSRWEN R3(.clk(clk), .reset(reset), .write(write[3]), .D(WD), .Q(R3_out));

    decoder deco(.X(A3), .Y(write_deco));
    assign write[0] = write_deco[0] & WE;
    assign write[1] = write_deco[1] & WE;
    assign write[2] = write_deco[2] & WE;
    assign write[3] = write_deco[3] & WE;

    mux4x1 out1(.I0(R0_out), .I1(R1_out), .I2(R2_out), .I3(R3_out), .S(A1), .Q(RD1));
    mux4x1 out2(.I0(R0_out), .I1(R1_out), .I2(R2_out), .I3(R3_out), .S(A2), .Q(RD2));

endmodule
```

Figure 23. Implementation of the Register file with Verilog

```verilog
//Mehmet Ataş
module registerfileTB #(parameter W = 32)();
   reg reset;
   reg clk;
   reg WE;
   reg [W-1:0] WD;
   reg [1:0] A1,A2,A3;
   wire [W-1:0] RD1, RD2;

   reg [W-1:0] vectornum, errors;
   reg testWE [499:0];
   reg [W-1:0] testWD [499:0];
   reg [5:0] testaddress [499:0];
   reg [2*W-1:0] testexp [499:0];
   reg [W-1:0] RD1_exp, RD2_exp;

   registerfile dut(.clk(clk), .WE(WE), .reset(reset), .WD(WD), .A1(A1), .A2(A2), .A3(A3), .RD1(RD1), .RD2(RD2));

   always begin   //create a clock
      clk = 0; #10; clk = 1; #10;
   end

   //4 different testvectors for different signals
   initial begin
      $readmemb("WE.tv", testWE);
      //test write enable signal
      $readmemh("testWD.tv", testWD);
      //test write data values
      $readmemb("testaddress.tv", testaddress);
      //test address values
      $readmemh("testexp.tv", testexp);
```


```verilog
//Mehmet Ataş
module registerfileTB #(parameter W = 32)();
   reg reset;
   reg clk;
   reg WE;
   reg [W-1:0] WD;
   reg [1:0] A1,A2,A3;
   wire [W-1:0] RD1, RD2;

   reg [W-1:0] vectornum, errors;
   reg testWE [499:0];
   reg [W-1:0] testWD [499:0];
   reg [5:0] testaddress [499:0];
   reg [2*W-1:0] testexp [499:0];
   reg [W-1:0] RD1_exp, RD2_exp;

   registerfile dut(.clk(clk), .WE(WE), .reset(reset), .WD(WD), .A1(A1), .A2(A2), .A3(A3), .RD1(RD1), .RD2(RD2));

   always begin   //create a clock
      clk = 0; #10; clk = 1; #10;
   end

   //4 different testvectors for different signals
   initial begin
      $readmemb("WE.tv", testWE);
      //test write enable signal
      $readmemh("testWD.tv", testWD);
      //test write data values
      $readmemb("testaddress.tv", testaddress);
      //test address values
      $readmemb("testexp.tv", testexp);
```


Figure 24. Test bench module to test my implementation of the Register File

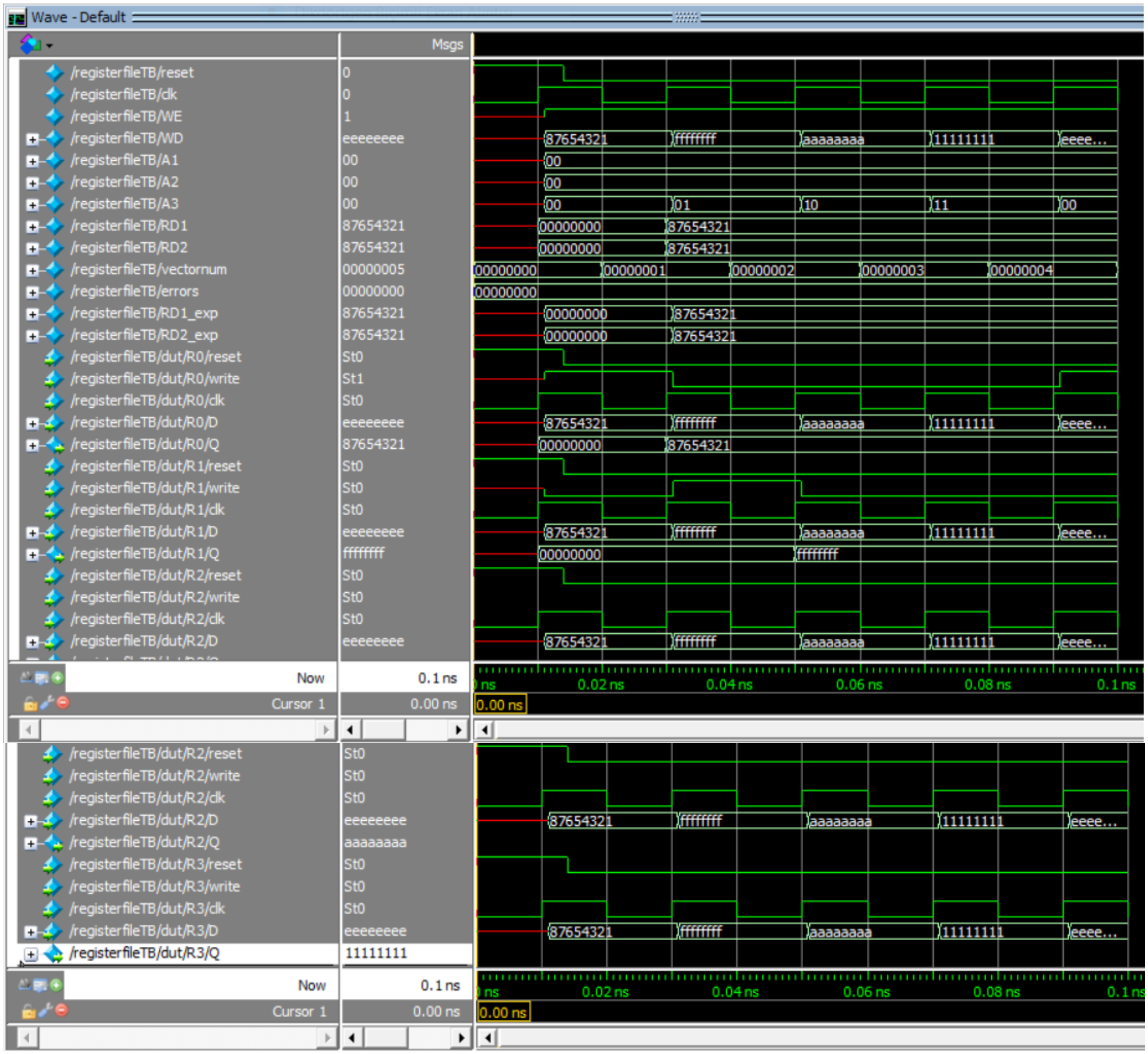


Figure 25. Simulation of the Register File

My Testvectors:

testaddress.tv

00_00_00	00_00_01	00_00_10	00_00_11	00_00_00
00_01_00	10_11_00	11_10_00	11_11_11	11_00_00

testexp.tv

00000000_00000000	87654321_87654321	87654321_87654321
87654321_87654321	87654321_87654321	EEEEEEEE_FFFFFFFF
AAAAAAAA_11111111	11111111_AAAAAAAA	11111111_11111111
12345678_EEEEEEEE		

testWD.vt

87654321	FFFFFFFF	AAAAAAAA	11111111	EEEEEEEE
00000000	00000000	00000000	12345678	00000000

WE.vt

1	1	1	1	1	0	0	0	1	0

Table 5: Register File Vector Table

WE A1 [1:0] A2 [1:0] A3 [1:0] WD [31:0] RD1 [31:0] RD2 [31:0]

1 1 2’b00 2’b00 2’b00 32’h87654321 32’h00000000 32’h00000000

2 1 2’b00 2’b00 2’b01 32’hFFFFFFFF 32’h87654321 32’h87654321

3 1 2’b00 2’b00 2’b10 32’hAAAAAAAA 32’h87654321 32’h87654321

4 1 2’b00 2’b00 2’b11 32’h11111111 32’h87654321 32’h87654321

5 1 2’b00 2’b00 2’b00 32’hEEEEEEEE 32’h87654321 32’h87654321

6 0 2’b00 2’b01 2’b00 32’h00000000 32’hEEEEEEEE 32’hFFFFFFFF

7 0 2’b10 2’b11 2’b00 32’h00000000 32’hAAAAAAAA 32’h11111111

8 0 2’b11 2’b10 2’b00 32’h00000000 32’h11111111 32’hAAAAAAAA

9 1 2’b11 2’b11 2’b11 32’h12345678 32’h11111111 32’h11111111

10 0 2’b11 2’b00 2’b00 32’h00000000 32’h12345678 32’hEEEEEEEE

R0 Q[31:0] R1 Q[31:0] R2 Q[31:0] R3 Q[31:0]

1 32’h00000000 32’h00000000 32’h00000000 32’h00000000

2 32’h87654321 32’h00000000 32’h00000000 32’h00000000

3 32’h87654321 32’hFFFFFFFF 32’h00000000 32’h00000000

4 32’h87654321 32’hFFFFFFFF 32’hAAAAAAAA 32’h00000000

5 32’h87654321 32’hFFFFFFFF 32’hAAAAAAAA 32’h11111111

6 32’hEEEEEEEE 32’hFFFFFFFF 32’hAAAAAAAA 32’h11111111

7 32’hEEEEEEEE 32’hFFFFFFFF 32’hAAAAAAAA 32’h11111111

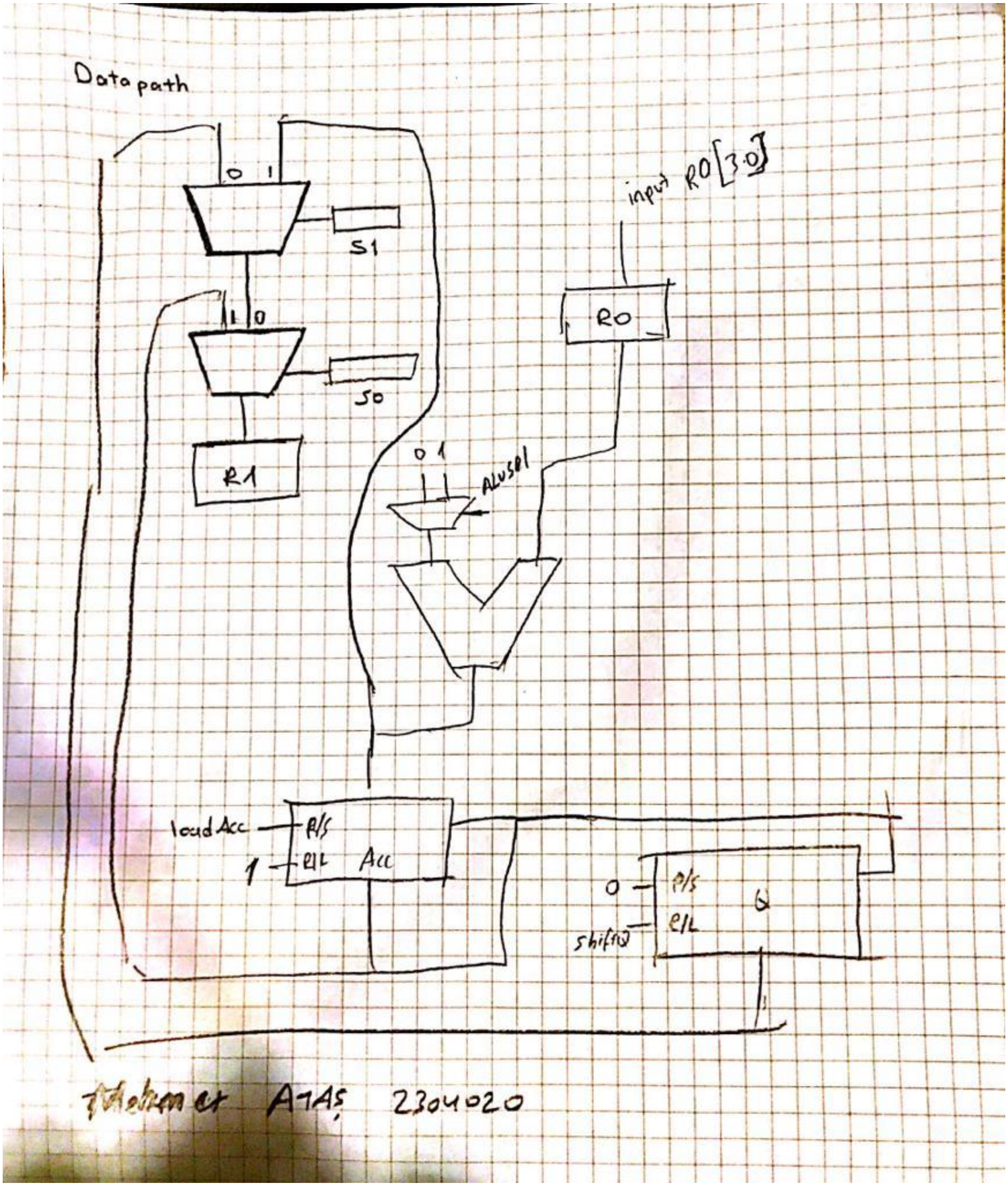
8 32’hEEEEEEEE 32’hFFFFFFFF 32’hAAAAAAAA 32’h11111111

9 32’hEEEEEEEE 32’hFFFFFFFF 32’hAAAAAAAA 32’h11111111

10 32’hEEEEEEEE 32’hFFFFFFFF 32’hAAAAAAAA 32’h12345678

The results are the same as in the vector table as expected.

1.4. Datapath Design



There are 5 control signals.

There are 3 different control signals.

Mux-selects, shift for Q register, load for Acc register.

In my datapath, I used 1 ALU, 2 Simple register with synchronous reset, 2 Shift register with parallel and serial load and 3 2to1 MUX.

Every different control signals in my architecture is use to perform the desired tasks.

Since there are signals, I can demonstrate these signals by 2 (log2(4) = 2) pins by applying some necessary Gates. However, in my design, I did not do that for the sake of clearness of the design.

First, Acc should be parallel loaded Acc-load=1

Then, Acc should be in serial loaded Acc-load=0

Total 10 cycles.

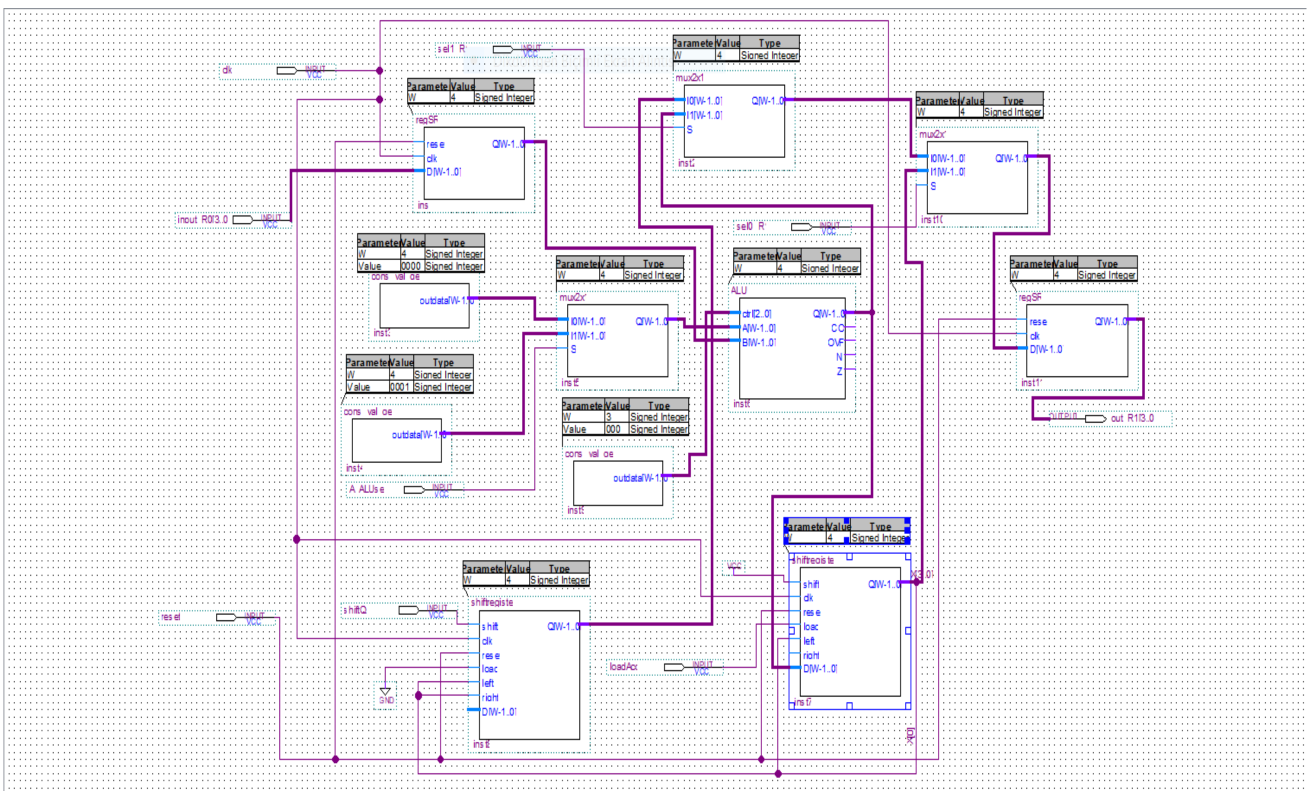


Figure 26. Implementation of my design in Schematic Editor