

11/03/2019

EE446 LABORATORY

EXPERIMENT 1

PRELIMINARY REPORT

Muttalip Caner TOL

2031466

Tuesday Afternoon

1.2. Module Design with Verilog HDL

1.2.1. Constant Value Generator

```
1 module Constant_Value_Generator #(parameter W = 4, V = 4'b0000) (bus_in);
2
3     output wire [(W-1):0] bus_in;
4
5     assign bus_in = V;
6
7 endmodule
```

1.2.2. Decoder

1. Implementation

```
1 module decoder #(parameter IN_WIDTH = 3) (
2     in, out
3 );
4     input [(IN_WIDTH-1):0] in;
5     output reg [(1<<IN_WIDTH)-1):0] out;
6
7     always @(in)
8     begin
9         out = {(1<<IN_WIDTH){1'b0}};
10        out[in] = 1'b1;
11    end
12
13 endmodule
```

2. Test bench module

```
1 module decoder_tb();
2     reg [1:0] in;
3     wire [3:0] out;
4     reg[1:0] testvectors[3:0]; // array of testvectors
5     reg [1:0] vectornum;
6
7     decoder #2 DUT(in, out);
8
9     // load vectors
10    initial
11    begin
12        $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/decoder_tb.tv",testvectors);
13        vectornum = 0;
14    end
15
16    //apply test vectors
17    always
18    begin
19        #1;
20        in = testvectors[vectornum];
21        vectornum = vectornum +1;
22    end
23
24 endmodule
```

3. Vector table

1	00
2	01
3	10
4	11

4. Verification

Wave - Default		Msgs			
+	/decoder_tb/in	00	00	01	10
	/decoder_tb/out	0001	0010	0100	1000

1.2.3. Multiplexers

1. Implementations

```
1 module mux2 #(parameter W = 4) ( select, mux_in1, mux_in2, mux_out );
2
3 input      select;
4 input[W-1:0] mux_in1, mux_in2;
5 output wire [W-1:0] mux_out;
6
7 assign mux_out = select ? mux_in2 : mux_in1;
8
9 endmodule
```

```
1 module mux4 #(parameter W = 3) ( select, mux_in1,mux_in2,mux_in3,mux_in4, mux_out );
2
3 input[1:0] select;
4 input[W:0] mux_in1,mux_in2,mux_in3,mux_in4;
5 output reg [W:0] mux_out;
6 always @(select, mux_in1,mux_in2,mux_in3,mux_in4)
7 begin
8     case(select)
9         2'b00: mux_out = mux_in1;
10        2'b01: mux_out = mux_in2;
11        2'b10: mux_out = mux_in3;
12        2'b11: mux_out = mux_in4;
13    endcase
14 end
15
16 endmodule
```

2. Test bench modules

```
1 module mux2_tb();
2     reg select;
3     reg[3:0] mux_in1, mux_in2;
4     wire [3:0] mux_out;
5     reg[8:0] testvectors[31:0]; // array of testvectors
6     reg [4:0] vectornum;
7
8     mux2 #4 DUT(select, mux_in1, mux_in2, mux_out);
9
10    // load vectors
11    initial
12    begin
13        $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/mux2_tb.tv",testvectors);
14        vectornum = 0;
15    end
16
17    //apply test vectors
18    always
19    begin
20        #5;
21        {mux_in1, mux_in2,select} = testvectors[vectornum];
22        vectornum = vectornum +1;
23    end
24
25 endmodule
```

3. Vector tables

Test vectors of mux2 and mux4:

1	0000_0000_0	1	0000_0000_0000_0000_00	33	0000_0000_0000_0000_10
2	0001_0000_0	2	0001_0000_0000_0000_00	34	0000_0000_0001_0000_10
3	0010_0000_0	3	0010_0000_0000_0000_00	35	0000_0000_0010_0000_10
4	0011_0000_0	4	0011_0000_0000_0000_00	36	0000_0000_0011_0000_10
5	0100_0000_0	5	0100_0000_0000_0000_00	37	0000_0000_0100_0000_10
6	0101_0000_0	6	0101_0000_0000_0000_00	38	0000_0000_0101_0000_10
7	0110_0000_0	7	0110_0000_0000_0000_00	39	0000_0000_0110_0000_10
8	0111_0000_0	8	0111_0000_0000_0000_00	40	0000_0000_0111_0000_10
9	1000_0000_0	9	1000_0000_0000_0000_00	41	0000_0000_1000_0000_10
10	1001_0000_0	10	1001_0000_0000_0000_00	42	0000_0000_1001_0000_10
11	1010_0000_0	11	1010_0000_0000_0000_00	43	0000_0000_1010_0000_10
12	1011_0000_0	12	1011_0000_0000_0000_00	44	0000_0000_1011_0000_10
13	1100_0000_0	13	1100_0000_0000_0000_00	45	0000_0000_1100_0000_10
14	1101_0000_0	14	1101_0000_0000_0000_00	46	0000_0000_1101_0000_10
15	1110_0000_0	15	1110_0000_0000_0000_00	47	0000_0000_1110_0000_10
16	1111_0000_0	16	1111_0000_0000_0000_00	48	0000_0000_1111_0000_10
17	0000_0000_1	17	0000_0000_0000_0000_01	49	0000_0000_0000_0000_11
18	0000_0001_1	18	0000_0001_0000_0000_01	50	0000_0000_0000_0001_11
19	0000_0010_1	19	0000_0010_0000_0000_01	51	0000_0000_0000_0010_11
20	0000_0011_1	20	0000_0011_0000_0000_01	52	0000_0000_0000_0011_11
21	0000_0100_1	21	0000_0100_0000_0000_01	53	0000_0000_0000_0100_11
22	0000_0101_1	22	0000_0101_0000_0000_01	54	0000_0000_0000_0101_11
23	0000_0110_1	23	0000_0110_0000_0000_01	55	0000_0000_0000_0110_11
24	0000_0111_1	24	0000_0111_0000_0000_01	56	0000_0000_0000_0111_11
25	0000_1000_1	25	0000_1000_0000_0000_01	57	0000_0000_0000_1000_11
26	0000_1001_1	26	0000_1001_0000_0000_01	58	0000_0000_0000_1001_11
27	0000_1010_1	27	0000_1010_0000_0000_01	59	0000_0000_0000_1010_11
28	0000_1011_1	28	0000_1011_0000_0000_01	60	0000_0000_0000_1011_11
29	0000_1100_1	29	0000_1100_0000_0000_01	61	0000_0000_0000_1100_11
30	0000_1101_1	30	0000_1101_0000_0000_01	62	0000_0000_0000_1101_11
31	0000_1110_1	31	0000_1110_0000_0000_01	63	0000_0000_0000_1110_11
32	0000_1111_1	32	0000_1111_0000_0000_01	64	0000_0000_0000_1111_11

4. Verification

Simulation results for mux2:

	Page	
/mux2_b/select	0	
/mux2_b/mux_in1	0000	
/mux2_b/mux_in2	0000	
/mux2_b/mux_out	0000	

Simulation results for mux4:

/mux4_b/select	00	
/mux4_b/mux_in1	0111	
/mux4_b/mux_in2	0000	
/mux4_b/mux_in3	0000	
/mux4_b/mux_in4	0000	
/mux4_b/mux_out	0111	

1.2.4. Arithmetic Logic Unit (ALU)

1. Implementation

```
1 module alu #(parameter W = 4) (  
2     // Inputs  
3     A, B,  
4     // Outputs  
5     out, CO, OVF, Z ,N,  
6     // Control Signal  
7     ALU_control);  
8     // Inputs  
9     input [W-1:0] A;  
10    input [W-1:0] B;  
11    // Outputs  
12    output reg[W-1:0] out;  
13    output reg CO, OVF, Z ,N;  
14    // Signal  
15    input[2:0] ALU_control;  
16  
17    always@(*)  
18    begin  
19        case (ALU_control)  
20            // Arithmetic operations  
21            3'b000: {CO,out} = A + B;  
22            3'b001: {CO,out} = A - B;  
23            3'b010: {CO,out} = B - A;  
24  
25            // Logic operations  
26            3'b011: out = A & ~B;  
27            3'b100: out = A & B;  
28            3'b101: out = A | B;  
29            3'b110: out = A ^ B;  
30            3'b111: out = A ~^ B;  
31  
32            endcase  
33  
34            OVF = ({CO,out[W-1]} == 2'b01);  
35  
36            if((ALU_control != 3'b000) && (ALU_control != 3'b001) && (ALU_control != 3'b010))  
37                begin  
38                    CO = 1'b0;  
39                    OVF = 1'b0;  
40                end  
41  
42            N = out[W-1];  
43            Z = out=={W{1'b0}};  
44        end  
45    endmodule
```

2. Overflow detection

I have put checkpoints on the arithmetic operations. Overflow flag is set when:

- adding two positives yields a negative
- or, adding two negatives gives a positive
- or, subtract a negative from a positive gives a negative
- or, subtract a positive from a negative gives a positive

3. Test bench module

```

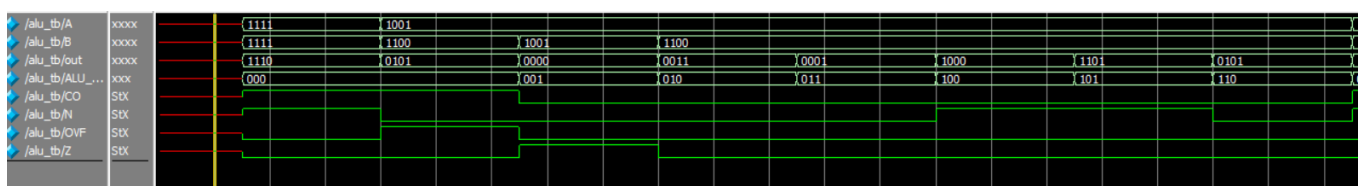
1  module alu_tb();
2      reg [3:0] A;
3      reg [3:0] B;
4      // Outputs
5      wire [3:0] out;
6      wire CO, OVF, Z ,N;
7      // Signal
8      reg [2:0] ALU_control;
9      reg[10:0] testvectors[7:0]; // array of testvectors
10     reg [2:0] vectornum;
11
12     alu #4 DUT(A, B,      out, CO, OVF, Z ,N, ALU_control);
13
14     // load vectors
15     initial
16     begin
17         $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/alu_tb.tv",testvectors);
18         vectornum = 0;
19     end
20
21     //apply test vectors
22     always
23     begin
24         #5;
25         {A,B,ALU_control} = testvectors[vectornum];
26         vectornum = vectornum +1;
27     end
28
29 endmodule

```

4. Vector table

1	1111_1111_000
2	1001_1100_000
3	1001_1001_001
4	1001_1100_010
5	1001_1100_011
6	1001_1100_100
7	1001_1100_101
8	1001_1100_110
9	1001_1100_111

5. Verification



1.2.5. Registers

1. Simple register with synchronous reset

```
1 module register_A #(parameter W = 8) (  
2     clk,  
3     DATA, A,  
4     reset  
5 );  
6     input clk;  
7     input [W-1:0] DATA;  
8     output reg [W-1:0] A;  
9     input reset;  
10  
11     initial  
12     begin  
13         A = {W{1'b0}};  
14     end  
15  
16     always @(posedge clk)  
17     begin  
18         if (!reset)  
19             A <= DATA;  
20         else  
21             begin  
22                 A <= {W{1'b0}};  
23             end  
24     end  
25 endmodule
```

2. Register with synchronous reset and write enable

```
1 module register_B #(parameter W = 8) (  
2     clk,  
3     DATA, A,  
4     reset, WE  
5 );  
6     input clk;  
7     input [W-1:0] DATA;  
8     output reg [W-1:0] A;  
9     input reset, WE;  
10  
11     initial  
12     begin  
13         A = {W{1'b0}};  
14     end  
15  
16     always @(posedge clk)  
17     begin  
18         if (reset)  
19             A <= {W{1'b0}};  
20         else if (WE==1)  
21             A <= DATA;  
22         else  
23             A <= A;  
24     end  
25 end  
26 endmodule
```

3. Shift register with parallel and serial load

```
1  module shift_reg #(parameter W = 4) (  
2      clk,  
3      parallel, right,  
4      DATA, A, R, L,  
5      reset  
6  );  
7      input clk;  
8      input [W-1:0] DATA;  
9      output reg [W-1:0] A;  
10     input reset, R, L, parallel, right;  
11  
12     initial  
13     begin  
14         A = {W{1'b0}};  
15     end  
16  
17     always @(posedge clk)  
18     begin  
19         if (reset)  
20             A = {W{1'b0}};  
21         else  
22             begin  
23                 if(parallel==1)  
24                     A = DATA;  
25                 else  
26                     begin  
27                         if(right)  
28                         begin  
29                             A = A >> 1;  
30                             A[W-1] = L;  
31                         end  
32                         else  
33                         begin  
34                             A = A << 1;  
35                             A[0] = R;  
36                         end  
37                     end  
38                 end  
39             end  
40         end  
41     endmodule
```


4. Test bench modules

For Simple Register with synchronous reset:

```
1 module register_A_tb();
2
3     reg clk;
4     reg [3:0] DATA;
5     wire [3:0] A;
6     reg reset;
7
8     reg[4:0] testvectors[31:0]; // array of testvectors
9     reg [4:0] vectornum;
10
11     register_A #4 DUT(clk, DATA, A, reset);
12     // generate clock
13     always // no sensitivity list, so it always executes
14     begin
15         clk <= 1;
16         #5;
17         clk <= 0;
18         #5;
19     end
20     // load vectors
21     initial
22     begin
23         $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/register_A_tb.tv",testvectors);
24         vectornum = 0;
25     end
26
27     //apply test vectors
28     always
29     begin
30         #13;
31         {DATA, reset} = testvectors[vectornum];
32         vectornum = vectornum +1;
33     end
34
35 endmodule
```

For Register with synchronous reset and write enable:

```
1 module register_B_tb();
2
3     reg clk;
4     reg [3:0] DATA;
5     wire [3:0] A;
6     reg reset, WE;
7
8     reg[5:0] testvectors[31:0]; // array of testvectors
9     reg [4:0] vectornum;
10
11     register_B #4 DUT(clk, DATA, A, reset, WE);
12     // generate clock
13     always // no sensitivity list, so it always executes
14     begin
15         clk <= 1;
16         #5;
17         clk <= 0;
18         #5;
19     end
20     // load vectors
21     initial
22     begin
23         $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/register_B_tb.tv",testvectors);
24         vectornum = 0;
25     end
26
27     //apply test vectors
28     always
29     begin
30         #13;
31         {DATA, reset, WE} = testvectors[vectornum];
32         vectornum = vectornum +1;
33     end
34
35 endmodule
```

For Shift register with parallel and serial load:

```

1 module shift_reg_tb();
2
3     reg clk;
4     reg [3:0] DATA;
5     wire [3:0] A;
6     reg reset, R, L, parallel, right;
7
8     reg[8:0] testvectors[31:0]; // array of testvectors
9     reg [4:0] vectornum;
10
11     shift_reg #4 DUT(clk, parallel, right, DATA, A, R, L, reset);
12     // generate clock
13     always // no sensitivity list, so it always executes
14     begin
15         clk <= 1;
16         #5;
17         clk <= 0;
18         #5;
19     end
20     // load vectors
21     initial
22     begin
23         $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/shift_reg_tb.tv",testvectors);
24         vectornum = 0;
25     end
26
27     //apply test vectors
28     always
29     begin
30         #13;
31         {DATA, parallel,right, R, L,reset} = testvectors[vectornum];
32         vectornum = vectornum +1;
33     end
34
35 endmodule

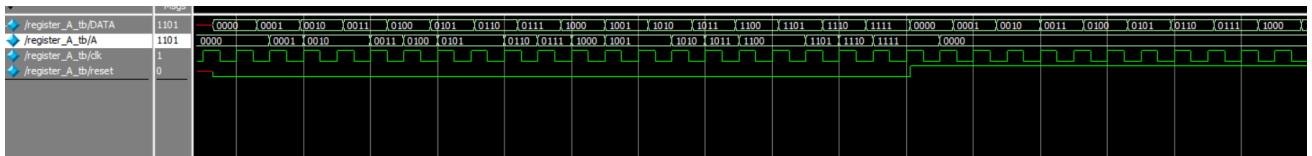
```

5. Vector tables

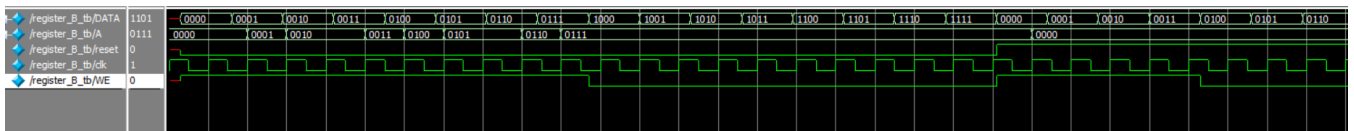
1	0000_0	1	0000_0_1	1	0000_1_0_0_0_0
2	0001_0	2	0001_0_1	2	0001_1_0_0_0_0
3	0010_0	3	0010_0_1	3	0010_1_0_0_0_0
4	0011_0	4	0011_0_1	4	0011_1_0_0_0_0
5	0100_0	5	0100_0_1	5	0100_1_0_0_0_0
6	0101_0	6	0101_0_1	6	0101_1_0_0_0_0
7	0110_0	7	0110_0_1	7	0110_1_0_0_0_0
8	0111_0	8	0111_0_1	8	0111_1_0_0_0_0
9	1000_0	9	1000_0_0	9	1000_0_1_0_1_0
10	1001_0	10	1001_0_0	10	1001_0_1_1_0_0
11	1010_0	11	1010_0_0	11	1010_0_1_0_1_0
12	1011_0	12	1011_0_0	12	1011_0_1_1_0_0
13	1100_0	13	1100_0_0	13	1100_0_0_0_1_0
14	1101_0	14	1101_0_0	14	1101_0_0_1_0_0
15	1110_0	15	1110_0_0	15	1110_0_0_0_1_0
16	1111_0	16	1111_0_0	16	1111_0_0_1_0_0
17	0000_1	17	0000_1_1	17	0000_1_1_1_0_1
18	0001_1	18	0001_1_1	18	0001_1_1_1_0_1
19	0010_1	19	0010_1_1	19	0010_1_1_1_0_1
20	0011_1	20	0011_1_1	20	0011_1_1_1_0_1
21	0100_1	21	0100_1_0	21	0100_1_0_1_0_1
22	0101_1	22	0101_1_0	22	0101_1_0_1_0_1
23	0110_1	23	0110_1_0	23	0110_1_0_1_0_1
24	0111_1	24	0111_1_0	24	0111_1_0_1_0_1
25	1000_1	25	1000_1_1	25	1000_1_1_1_0_1
26	1001_1	26	1001_1_1	26	1001_1_1_1_0_1
27	1010_1	27	1010_1_1	27	1010_1_1_1_0_1
28	1011_1	28	1011_1_1	28	1011_1_1_1_0_1
29	1100_1	29	1100_1_0	29	1100_1_0_1_0_1
30	1101_1	30	1101_1_0	30	1101_1_0_1_0_1
31	1110_1	31	1110_1_0	31	1110_1_0_1_0_1
32	1111_1	32	1111_1_0	32	1111_1_0_1_0_1

6. Verification

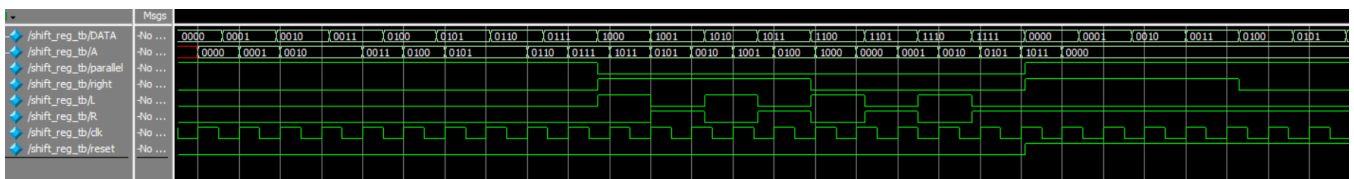
Simulation of Simple register with synchronous reset



Simulation of Register with synchronous reset and write enable

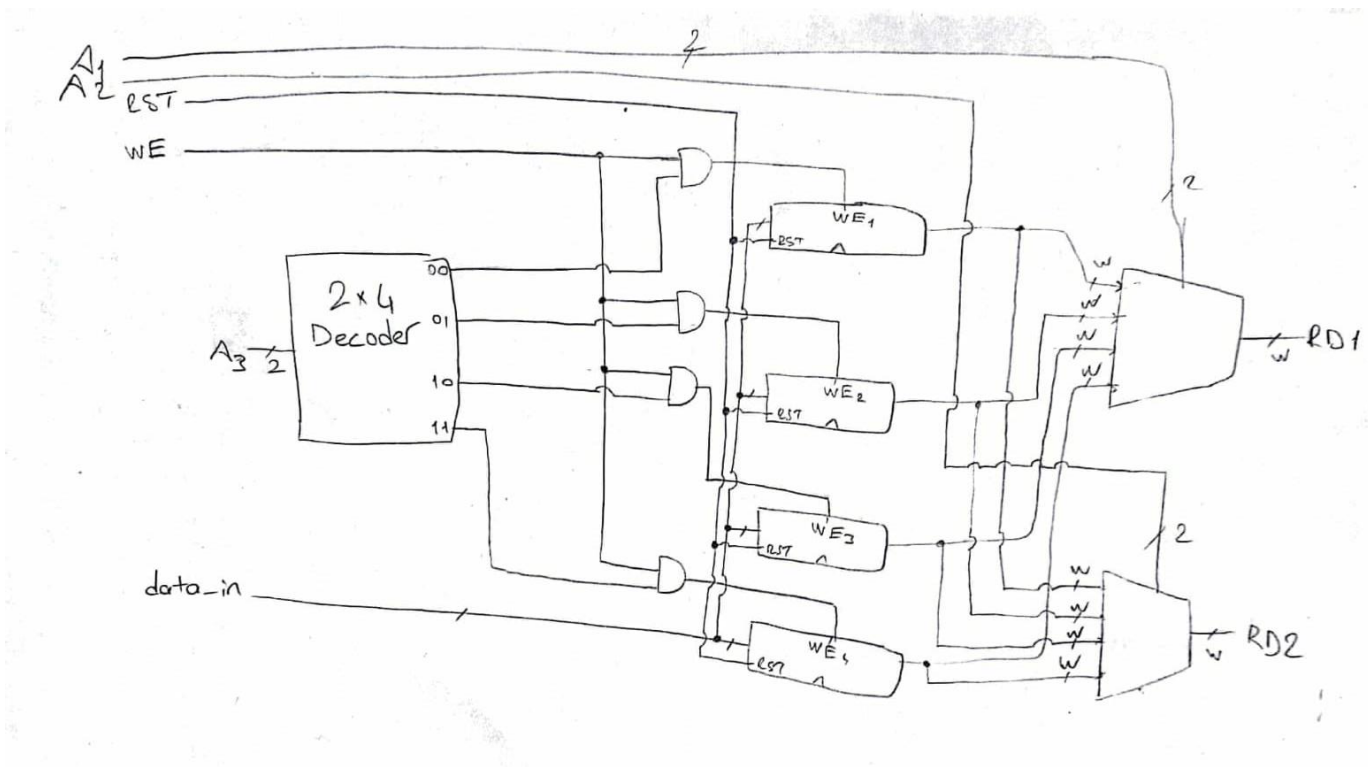


Simulation of Shift register with parallel and serial load



1.3. Register File

1. Design



2. Implementation

```
1 module register_file #(parameter W = 4) (
2     clk,
3     data_in,
4     WE3, rst,
5     A1, A2, A3,
6     RD1, RD2
7 );
8
9     // inputs
10    input clk;
11    input [W-1:0] data_in;
12    input [1:0] A1, A2, A3;
13    input WE3, rst;
14    // outputs
15    output wire [W-1:0] RD1, RD2;
16    // wires
17    wire [3:0] d; // output of decoder
18    wire [W-1:0] r_out1, r_out2, r_out3, r_out4; // register outputs
19    wire EN1, EN2, EN3, EN4; // write enables
20
21    decoder #(2) decoder1 (
22        .in(A3),
23        .out(d)
24    );
25
26    assign EN1 = WE3 & d[0];
27    assign EN2 = WE3 & d[1];
28    assign EN3 = WE3 & d[2];
29    assign EN4 = WE3 & d[3];
30
31    register_B #(W) reg1 (
32        .clk(clk),
33        .DATA(data_in),
34        .A(r_out1),
35        .reset(rst),
36        .WE(EN1)
37    );
38
39    register_B #(W) reg2 (
40        .clk(clk),
41        .DATA(data_in),
42        .A(r_out2),
43        .reset(rst),
44        .WE(EN2)
45    );
46
47    register_B #(W) reg3 (
48        .clk(clk),
49        .DATA(data_in),
50        .A(r_out3),
51        .reset(rst),
52        .WE(EN3)
53    );
54
55    register_B #(W) reg4 (
56        .clk(clk),
57        .DATA(data_in),
58        .A(r_out4),
59        .reset(rst),
60        .WE(EN4)
61    );
62
```

```

63     mux4 #(W) mux_1(
64         .select(A1),
65         .mux_in1(r_out1),
66         .mux_in2(r_out2),
67         .mux_in3(r_out3),
68         .mux_in4(r_out4),
69         .mux_out(RD1)
70     );
71
72     mux4 #(W) mux_2(
73         .select(A2),
74         .mux_in1(r_out1),
75         .mux_in2(r_out2),
76         .mux_in3(r_out3),
77         .mux_in4(r_out4),
78         .mux_out(RD2)
79     );
80
81 endmodule

```

3. Test bench module

```

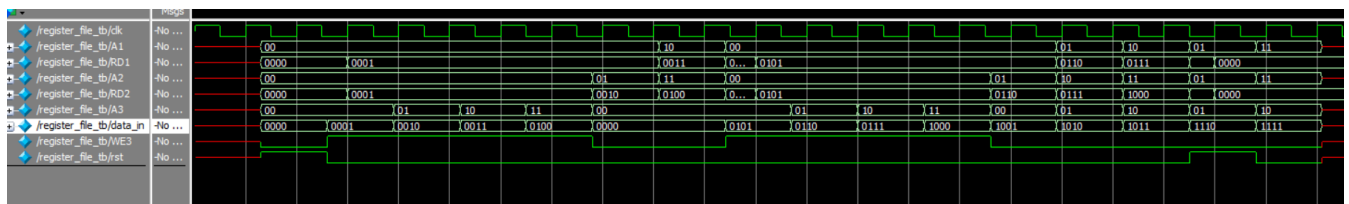
1  module register_file_tb();
2
3      reg clk;
4      reg [3:0] data_in;
5      reg [1:0] A1, A2, A3;
6      reg WE3, rst;
7      // outputs
8      wire [3:0] RD1, RD2;
9
10     reg[11:0] testvectors[15:0]; // array of testvectors
11     reg [4:0] vectornum;
12
13     register_file #4 DUT(clk, data_in, WE3, rst, A1, A2, A3, RD1, RD2);
14     // generate clock
15     always // no sensitivity list, so it always executes
16     begin
17         clk <= 1;
18         #5;
19         clk <= 0;
20         #5;
21     end
22     // load vectors
23     initial
24     begin
25         $readmemb("C:/Users/Caner/Documents/GitHub/Course-Projects/EE446/LAB1/register_file_tb.tv",testvectors);
26         vectornum = 0;
27     end
28
29     //apply test vectors
30     always
31     begin
32         #13;
33         {data_in, A1, A2, A3, WE3, rst} = testvectors[vectornum];
34         vectornum = vectornum + 1;
35     end
36
37 endmodule

```

4. Vector table

1	0000_00_00_00_0_1
2	0001_00_00_00_1_0
3	0010_00_00_01_1_0
4	0011_00_00_10_1_0
5	0100_00_00_11_1_0
6	0000_00_01_00_0_0
7	0000_10_11_00_0_0
8	0101_00_00_00_1_0
9	0110_00_00_01_1_0
10	0111_00_00_10_1_0
11	1000_00_00_11_1_0
12	1001_00_01_00_0_0
13	1010_01_10_01_0_0
14	1011_10_11_10_0_0
15	1110_01_01_01_0_1
16	1111_11_11_10_0_0

5. Verification



1.4. Datapath Design for an Architecture

- How many control pins for the control signals does your architecture have?
R1Src, ASrc, ALUControl[2:0], Acc_parallel, Acc_Lsrc[1..0], Q_right and the reset signals of the four registers.
There are 9 control pins.
- How many different control signals does your architecture use to perform the desired tasks?
R1Src, ASrc, ALUControl[2:0], Acc_parallel, Acc_Lsrc[1..0], Q_right
6 control signals.
- Can you reduce the number of the control pins? Why not or how?
We can ignore the reset pins of the registers.
- Write down the sequence of the control signals for REVERSED LOAD operation.
How many cycle does this operation take?

1st Clock: ASrc = 0, ALUControl = 00

R0 <- DATA

2nd Clock: Acc_parallel = 1

Acc <- DATA + 0x0

3rd Clock: Acc_parallel = 0, Acc_Lsrc[1..0] = 00, Q_right=1

Acc[3] <- Q[0]

Q[3] <- Acc[0]

4th Clock:

Acc[3] <- Q[0]

Q[3] <- Acc[0]

5th Clock:

Acc[3] <- Q[0]

Q[3] <- Acc[0]

6th Clock:

Acc[3] <- Q[0]

Q[3] <- Acc[0]

7th Clock: Q_right = 0, Acc_Lsrc[1..0] = 01

Acc[3] <- Q[3]

Q[0] <- 0

8th Clock:

Acc[3] <- Q[3]

Q[0] <- 0

9th Clock:

Acc[3] <- Q[3]

Q[0] <- 0

10th Clock:

Acc[3] <- Q[3]

Q[0] <- 0

11th Clock: Acc_parallel = 1, R1Src = 1

R1 <- Acc

11 Clock cycles in total.

