

CA Lab: Lab 7

student: Dimitri Tabatadze

April 30, 2023

Task Description

Write code for general purpose register file. MIPS processor has a register file that contains 32 registers. Each register is 32-bit long. Lab task is to design General Purpose Register File.

Write Verilog code for 3 port general purpose register file. A port consists of an address and data input/output.

1. Give us the number of bits of **addrA** and **addrB**.
2. Implement logic in your Verilog code that allows us to read values stored in registers.
3. Implement logic in your Verilog code that allows us to update values stored in registers
4. In you code implement logic that makes sure that value stored in register **\$0** stays 0 all the time.
5. Write testbench for your design. Generate Waveforms and explain in your reports why do you think your design works correctly.

Solution

1. The size of the addresses, **addrA**, **addrB** and **addrC** should be $\log_2(32) = 5$
2. The code that allows us to read the values stored at addresses **addrA** and **addrB**:

```
1 // ...
2
3 always @(addrA)
4     data_out_A <= register[addrA];
5
6 always @(addrB)
7     data_out_B <= register[addrB];
8
9 // ...
```

3. The code that allows us to update the value stored at address `addrC`:

```
1 // ...
2
3 always @(posedge clk) begin
4     if (write_enable) begin
5         // ...
6
7         register[addrC] <= data_in_C;
8
9         // After the update, if the updated addresses are
10        // supposed to be read, update the outputs accordingly.
11        if (addrC == addrA)
12            data_out_A <= data_in_C;
13
14        if (addrC == addrB)
15            data_out_B <= data_in_C;
16
17        // ...
18    end
19 end
20
21 // ...
```

4. The code that makes sure the value stored at address `$0` never changes and stays 0:

```
1 // ...
2
3 always @(posedge clk) begin
4     if (write_enable) begin
5         if (addrC != 0) begin // This check makes
6             // sure that only those addresses which
7             // are not 0 get updated.
8                 // ...
9             end
10        end
11    end
12 end
13 // ...
```

5. The whole testbench:

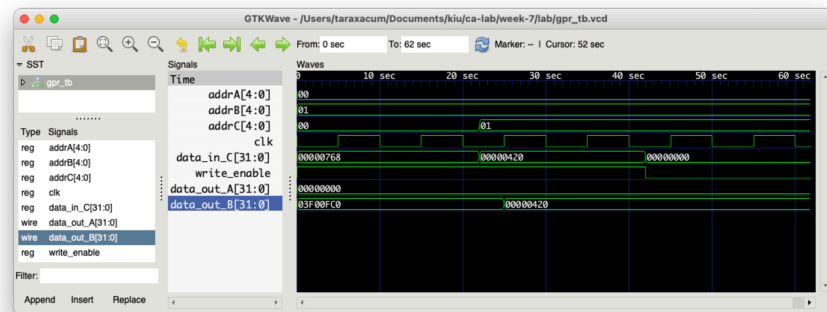
```
1 module gpr_tb();
2
3 reg clk;
4 reg write_enable;
5 reg [4:0] addrA;
6 reg [4:0] addrB;
7 reg [4:0] addrC;
8 wire [31:0] data_out_A;
9 wire [31:0] data_out_B;
10 reg [31:0] data_in_C;
11
12 gpr UUT(
13     .clk(clk),
14     .write_enable(write_enable),
```

```

15     .addrA(addrA),
16     .addrB(addrB),
17     .addrC(addrC),
18     .data_out_A(data_out_A),
19     .data_out_B(data_out_B),
20     .data_in_C(data_in_C)
21 );
22
23 always #5 clk = ~clk;
24
25 initial begin
26     $dumpfile("gpr_tb.vcd");
27     $dumpvars(0, gpr_tb);
28     clk = 0;
29     write_enable = 1;
30     addrA = 0; addrB = 1; addrC = 0;
31     data_in_C = 3'h768;
32     #22;
33     addrC = 1;
34     data_in_C = 3'h420;
35     #20;
36     write_enable = 0;
37     data_in_C = 1'h0;
38     #20;
39     addrA = 9;
40     #20;
41     $finish;
42 end
43
44 endmodule

```

Figure 1: The timing diagram



Conclusion

I think my implementation of the specified GPR is correct, because when given inputs, the outputs are correct (as far as I know).

In case you want to see the full code for the GPR, here it is:

```

1 module gpr (
2     input clk,
3     input write_enable,
4     input [4:0] addrA,
5     input [4:0] addrB,
6     input [4:0] addrC,
7     output reg [31:0] data_out_A,
8     output reg [31:0] data_out_B,
9     input [31:0] data_in_C
10 );
11
12 reg [31:0] register [0:31];
13 integer i;
14
15 initial begin
16     $readmemb("values.txt", register);
17 end
18
19 always @(posedge clk) begin
20     if (write_enable) begin
21         if (addrC != 0) begin
22             register[addrC] <= data_in_C;
23
24             if (addrC == addrA)
25                 data_out_A <= data_in_C;
26
27             if (addrC == addrB)
28                 data_out_B <= data_in_C;
29         end
30     end
31 end
32
33 always @(addrA)
34     data_out_A <= register[addrA];
35
36 always @(addrB)
37     data_out_B <= register[addrB];
38
39 endmodule

```

and the contents of the corresponding "values.txt":

```

1 00000000000000000000000000000000
2 000000111111000000000111111000000
3 0000010000000100000001000000100000
4 000010000000010000010000000010000
5 000010000000010000010000000010000
6 000010000000010000010000000010000
7 000010000000010000010000000010000
8 000001000000010000000100000010000
9 0000001111110000000111111000000
10 000000000000010000000100000000000
11 000000000000010000001100000000000
12 000000000000010000101000000000000
13 000000000000010000111000000000000
14 000000000000010001001000000000000
15 000000000000010001011000000000000

```

```

16 00000000000010001101000000000000
17 00000000000010001111000000000000
18 00000000000010010001000000000000
19 00000000000010010011000000000000
20 00000000000010010101000000000000
21 00000000000010010111000000000000
22 00000000000010011001000000000000
23 00000000000010011011000000000000
24 00000000000011111111000000000000
25 00000000000110000001100000000000
26 00000000000100000001000000000000
27 00000000000010000001000000000000
28 00000000000011111100000000000000
29 10000000000000000000000000000001
30 01000000000000000000000000000010
31 001000000000000000000000000000100
32 000100000000000000000000000001000

```

Reference

- me.