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`:

```
// ...

always @(addrA)
    data_out_A <= register[addrA];

always @(addrB)
    data_out_B <= register[addrB];

// ...
```

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

```
// ...

always @(posedge clk) begin
    if (write_enable) begin
        // ...

        register[addrC] <= data_in_C;

        // After the update, if the updated adresses are
        // supposed to be read, update the outputs accordingly.
        if (addrC == addrA)
            data_out_A <= data_in_C;

        if (addrC == addrB)
            data_out_B <= data_in_C;

        // ...
    end
end

// ...
```

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

```
// ...

always @(posedge clk) begin
    if (write_enable) begin
        if (addrC != 0) begin // This check makes
        // sure that only those addresses which
        // are not 0 get updated.
            // ...
        end
    end
end

// ...
```

5. The whole testbench:

```
module gpr_tb();

reg clk;
reg write_enable;
reg [4:0] addrA;
reg [4:0] addrB;
reg [4:0] addrC;
wire [31:0] data_out_A;
wire [31:0] data_out_B;
reg [31:0] data_in_C;

gpr UUT(
    .clk(clk),
    .write_enable(write_enable),
```

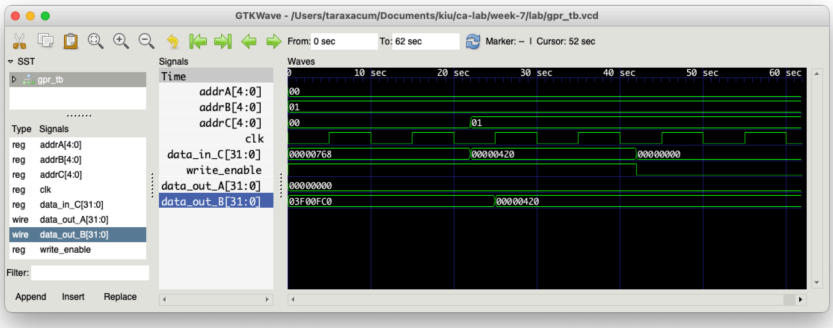
```
    .addrA(addrA),
    .addrB(addrB),
    .addrC(addrC),
    .data_out_A(data_out_A),
    .data_out_B(data_out_B),
    .data_in_C(data_in_C)
);

always #5 clk = ~clk;

initial begin
    $dumpfile("gpr_tb.vcd");
    $dumpvars(0, gpr_tb);
    clk = 0;
    write_enable = 1;
    addrA = 0; addrB = 1; addrC = 0;
    data_in_C = 3'h768;
    #22;
    addrC = 1;
    data_in_C = 3'h420;
    #20;
    write_enable = 0;
    data_in_C = 1'h0;
    #20;
    addrA = 9;
    #20;
    $finish;
end

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:

```
module gpr (
    input clk,
    input write_enable,
    input [4:0] addrA,
    input [4:0] addrB,
    input [4:0] addrC,
    output reg [31:0] data_out_A,
    output reg [31:0] data_out_B,
    input [31:0] data_in_C
);

reg [31:0] register [0:31];
integer i;

initial begin
    $readmemb("values.txt", register);
end

always @(posedge clk) begin
    if (write_enable) begin
        if (addrC != 0) begin
            register[addrC] <= data_in_C;

            if (addrC == addrA)
                data_out_A <= data_in_C;

            if (addrC == addrB)
                data_out_B <= data_in_C;
        end
    end
end

always @(addrA)
    data_out_A <= register[addrA];

always @(addrB)
    data_out_B <= register[addrB];

endmodule
```

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

```
00000000000000000000000000000000
00000011111100000000111111000000
00000100000010000001000000100000
00001000000001000010000000010000
00001000000001000010000000010000
00001000000001000010000000010000
00001000000001000010000000010000
00000100000010000001000000100000
00000011111110000001111111000000
00000000000010000001000000000000
00000000000010000011000000000000
00000000000010000101000000000000
00000000000010000111000000000000
00000000000010001001000000000000
00000000000010001011000000000000
```

```
00000000000010001101000000000000
00000000000010001111000000000000
00000000000010010001000000000000
00000000000010010011000000000000
00000000000010010101000000000000
00000000000010010111000000000000
00000000000010011001000000000000
00000000000010011011000000000000
00000000000011111111000000000000
00000000000011000000110000000000
00000000000010000000100000000000
00000000000010000001000000000000
00000000000001111110000000000000
10000000000000000000000000000001
01000000000000000000000000000010
00100000000000000000000000000100
00010000000000000000000000001000
```

Reference

- me.