ENGR xD52: MP b001

Due October 14th 5PM EST

This lab assignment stresses your ability to test complex systems effectively and reinforces your Verilog skills. Teams will play a game of tester vs test case.

You may work in groups of up to 3. One representative from each team must submit all deliverables electronically to comparch13@gmail.com as a single compressed archive. Please format the email's Header as "[CA] [MP1] Name1 Name2 Name3"

The Work Plan

This subdeliverable is due within 36 hours as an email to comparch13@gmail.com with the subject line formatted as [CA][MP1 Work Plan] Name1 Name2 Name3". Any questions regarding this section should be directed to Molly Farison or Eric VanWyk.

Create a work plan for this lab. Break down the lab in to small portions, and for each portion predict how long it will take (in hours) and when it will be done by (date). You will be comparing your predictions to reality later.

The Test Bench

Construct a test bench for a 32-bit 32-entry register file in Verilog. This test bench will approximate the type of testing done on the manufacturing line, and will either accept or reject sample register files. It **does not** need to diagnose the location or type of fault in the DUT (Device Under Test), only approve or reject.

The Good Register File

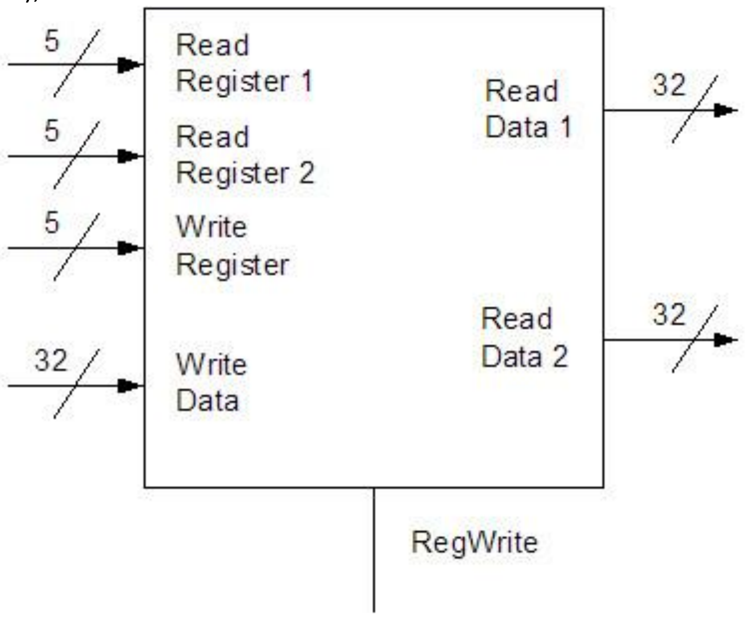
Construct a 32-bit 32-entry register file in Verilog. This register file will read two registers and optionally write one register on the positive edge of each clock cycle.

Since this register file will mimic the MIPS architecture, address zero will always return the value zero regardless of attempts to write to it.

Signature

Your register must have this signature so that all the test benches are cross compatible.

```
module regfile(ReadData1,       // Contents of first register read
               ReadData2,       // Contents of second register read
               WriteData,       // Contents to write to register
               ReadRegister1,   // Address of first register to read
               ReadRegister2,   // Address of second register to read
               WriteRegister,    // Address of register to write
               RegWrite,         // Enable writing of register when High
               clk               // Clock (Positive Edge Triggered)
               );
```

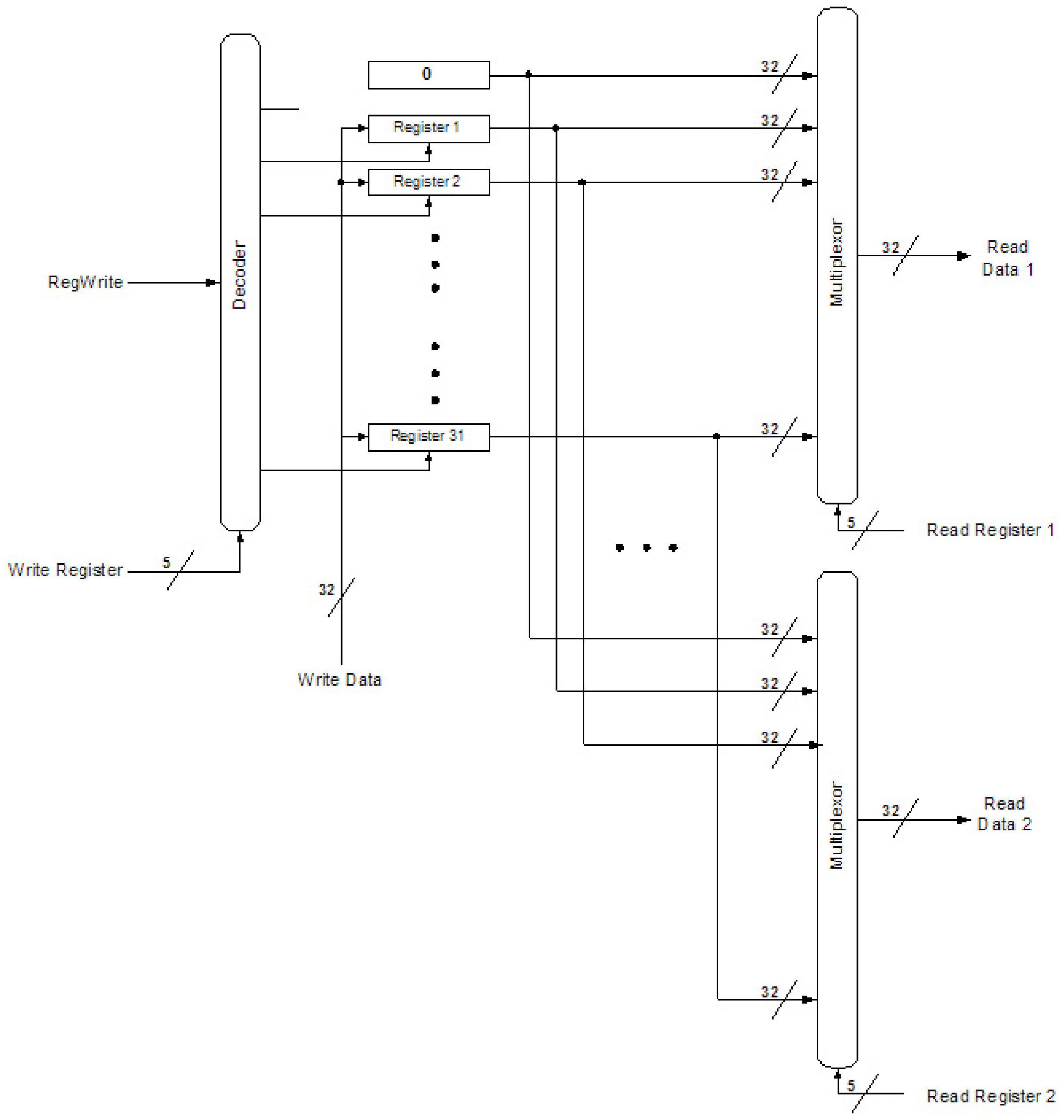


Recommended Implementation

Construct a 32-bit register with 32 D flip flop. This 32-bit register should have a write enable input. DO NOT GATE THE CLOCK (see notes/hints).

Construct a 32-bit decoder (with enable) to select the appropriate register to write to.

Construct a 32-bit mux as a module, and re-use this to create a 32x32-bit mux. Use two of these for reading.



The Busted Register File

Using your 'good' register file, create a register file that exhibits a single fault. This fault should assume that the design was good, but the manufacturing process was flawed.

You should be able to trace the nature of your active fault to a single misbehaving gate. For example, one of your bits in your register file could be "stuck at high" or "stuck at low". Be creative, but be realistic.

Clearly comment and document the active fault within your code.

KEEP YOUR FAULT A SECRET UNTIL DEMO DAY.

The Write Up

Create a semi-formal lab write up detailing the modules, their expected behavior, and their tested behavior. Include the waveform outputs of your test benches.

Explain the active fault in your busted register file, and provide a test case that highlights this fault.

This should be a single document bundled into the submitted archive.

The Demo Day

We will devote a day in class to Demo this lab. Each team will run their test bench against all of the other teams' good register files and busted register files. A "winning" test protocol will be chosen based on how few test cases are used to catch all of the active faults.

Hints / Notes

The D Flip Flop

Here is the Verilog code for the edge triggered D Flip Flop. Note that it is Behavioral Verilog code. You ~~may~~ should use this as-is in your labs.

```
module D_FF (q, d, clk);
    output q;
    input d, clk;
    reg q; // Indicate that q is stateholding
    always @(posedge clk) // Hold value except at edge
    q = d;
endmodule
```

Never Ever Gate The Clock Ever

It is bad form to "gate the clock". What this means is that the clock that goes around to all your state-holding elements (i.e. DFF or registers) should be unencumbered, and not pass through any gates. This is critical in complex designs because it puts a delay on the clock signal for items which need to fire at the same time. The whole reason we put in clocks is to "sample" the outputs and inputs only when we know there will be no glitching. If we instead put gates in the way of the clock, the DFF's will fire at different times, completely defeating our global synchronization. This in turn will cause glitching that is darned near impossible to find. Also, people will laugh at you and point fingers.

Big Elements

There are several very wide elements in this lab. Consider constructing them from smaller re-usable blocks as appropriate. You have flexibility in your implementation, explore it.

Input and Output

You need to declare all your inputs and outputs and all the intermediate signals you use in your designs. This was not told to you explicitly, but it is just good coding convention that makes your code more 'portable' between compilers, simulators, and synthesizers. Thus, if you have the statement:

```
and (out, in1, in2)
```

you need to have previously declared out, in1, and in2, to be some sort of physical entity (wire, reg).

Behavioral vs Structural

The previous lab required all of your modules to be written in structural Verilog, which requires you to explicitly write out each and every gate in your design. This lab allows the use of behavioral Verilog, which is higher level and will create gates to suit your purposes. The advantage is that this may reduce the time it takes you to write the code. The disadvantage is that you have less visibility in to exactly what gates it is using.

Only use behavioral Verilog where you are already confident in the structural equivalent. When you do this, include a brief description of the structural equivalent as a comment. This is only to save some typing.

I explicitly do not want to see complex or compound behavioral statements

Do Files

ModelSim "Do" files are a way of scripting repetitive tasks. You may want to learn more about them.

Final bits of wisdom

- Test EACH MODULE you make. There is literally 0% chance that you will write all these pieces without testing them, then slap them together into a register file and it will just work. Add to the fact that this is now a conglomeration of hundreds if not thousands of gates, it is hard to debug when it inevitably does not work. Trust me, it will help.
- Check for typos. Verilog won't really tell you when you've used a signal that doesn't exist.
- Use the concatenation operation. Check the verilog tutorial.
- You may want to consider writing a preprocessor (only if you already know what one is).