# ENGR 3410: MP #1
# MIPS 32-bit Register File

Due: October 13, 2006, beginning of class

## 1 Introduction

The purpose of this machine problem is to create the first large component of our MIPS-style microprocessor—the register file. The register file is where we keep values that our CPU is currently working on before they are committed to memory. My time estimate for the completion of this machine problem is 12 person-hours. Your mileage may vary.

## 2 The Problem

For this machine problem you are to construct a 32 by 32 register file using Verilog. Within the 32 by 32 register file is an array of 32 different 32-bit registers. These registers must be constructed from D flip-flops (Master/Slave, positive edge-triggered, supplied later in this machine problem). Note that for our processor (and a MIPS processor), register zero is hardwired to always output the value zero, regardless of what may or may not be written to it. Figure 1 below shows a block diagram of a register file.

Read Register 1 and Read Register 2 are 5-bit inputs which select the registers whose values are output on the 32-bit Read Data 1 bus and Read Data 2 bus respectively. The 5-bit Write Register input bus designates the register into which the information on the 32-bit Write Data bus is to be written when the RegWrite control signal is high.

## 3 Implementation

A simple implementation of the 32 by 32 MIPS register file can be made using Registers composed of D flip-flops, a 1 to 32 decoder, and two large 32x32 to 32 multiplexors. This is shown in the following block diagram (note that the clock is omitted for clarity), Figure 2.

Each register is simply an array of 32 D flip-flops, the block representation of which is in Figure 3

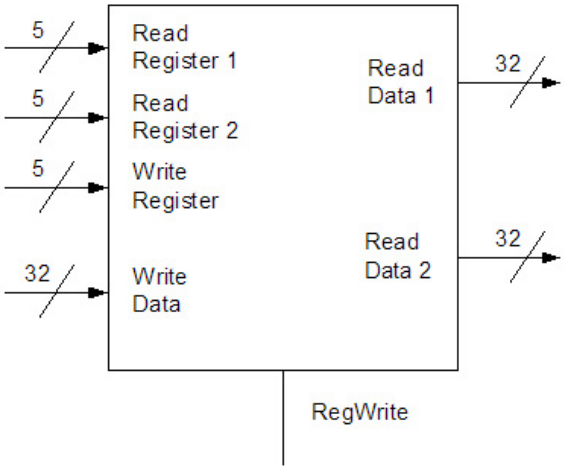


Figure 1: Register file schematic.

In your register file, the D input of each D flip-flop corresponds to a single bit in the 32 bit data input bus and the Q output of each D flip-flop corresponds to the appropriate bit in the 32 bit data output bus. The enable of every D flip-flop is connected to the same write enable input signal. This element will also receive a clock input, which synchronizes the registers. Note that the clock and the write enable are separate signals. The clock is a pure, periodic signal with no glitches or other weird behaviors. The write enable may have glitches and hazards, and thus must be moderated by the clock - make sure that random transitions of the write enable, as long as they are not simultaneous with the activating clock edge (positive edge), do not cause the register to spuriously grab a new value.

The decoder selects which register receives the RegWrite control signal as its enable input. When a register is not selected its enable input should be false. Also note that the least significant output line of the decoder is floating in the block diagram for the register file. This is because the zero register is hardwired to zero so it does not need write enable input. In fact, instead of using a register, you can just hard-code the inputs to the output muxes for register zero to all zeroes.

# 4 Requirements

- Use the file "regstim.v" as your *sample* testbench. You can find this file on the wiki. *This is just an example.* It does not contain nearly enough test cases to consider your register file "tested". You should be making significant changes to it to prove to yourself (and us) that your register file works. I have my own testbench for use during the demos, so you must make sure your register file takes the same inputs and outputs, in

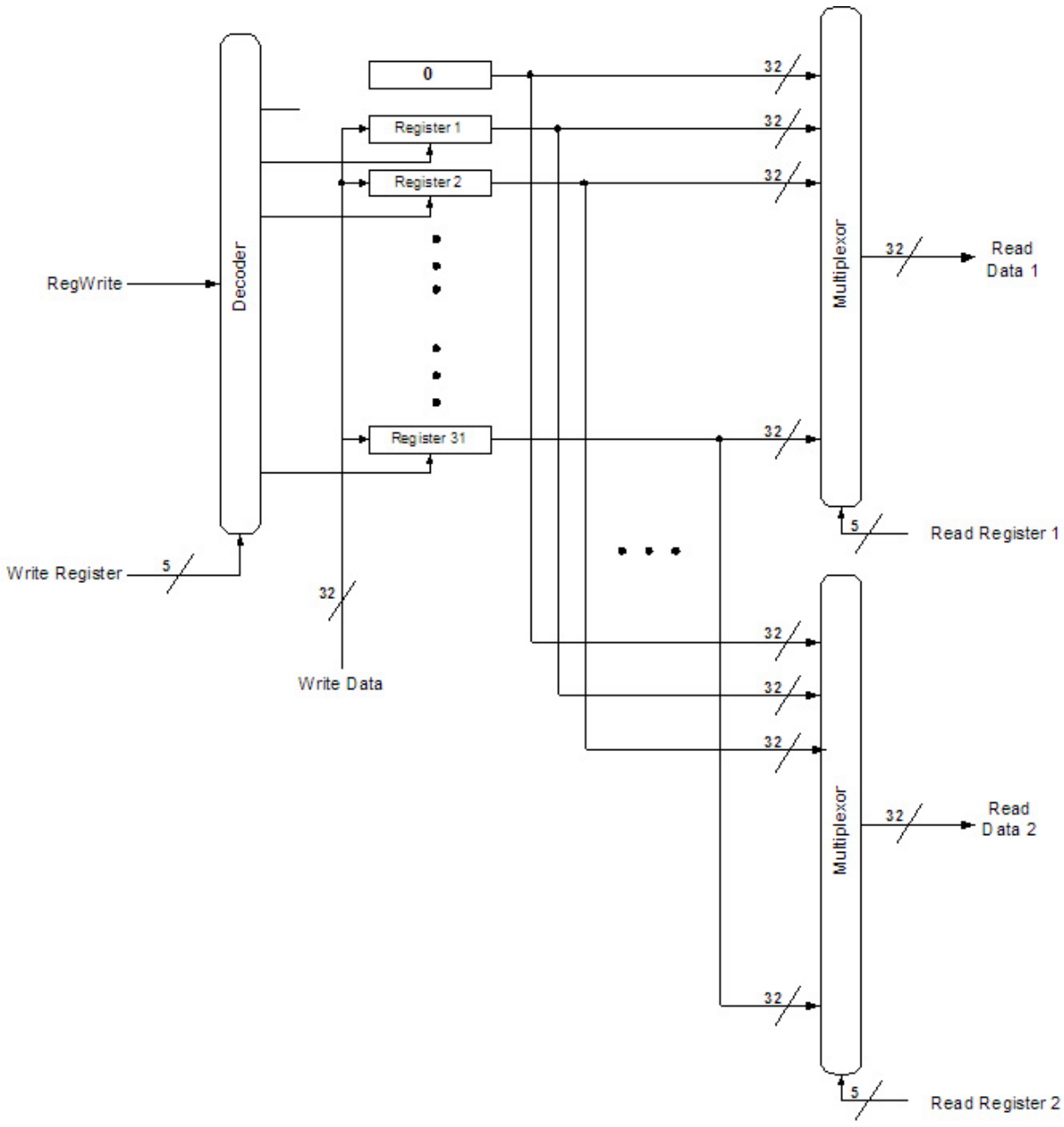


Figure 2: Block diagram of register file internals.

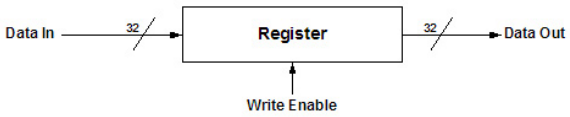


Figure 3: D-type Flip Flop block diagram.

the same order, as is presented in the provided testbench. Write a bunch of tests!

- All logic must be gate level, structural. That is, built from explicit AND, OR, NAND, NOR, XOR, etc. gates. No assign statements (except an assign to set a wire to a constant value), CASE statements, etc.
- You may use behavioral Verilog for your test benches.
- All gates have a delay of 50 units. Processor performance won't be a grading criteria for the class (unless you do really ridiculous things), but you need delay to show how things behave.

# 5 Deliverables

We have two deliverables. The write-up with code, and a demo.

## 5.1 Write-Up

*The requirements are modified from MP#0. Please read carefully.* I expect a semi-formal "lab write-up" of this machine problem. It does not need to be as rigorous as lab notebooks in other, more experimental classes. It should include, at a minimum:

- A brief write-up of the experiments
- Files of all Verilog code — modules *and* test benches
- Simulation output (textual or waveform) for each circuit
- A full schematic at the gate level. It will likely be multi-level (i.e. boxes on an upper level have a lower-level sheet with the details). Do not use the Cadence tool to generate your schematic. Photocopies or scans of your pictures are acceptable.

We would prefer this packaged as a single document (Word, LaTeX, PDF) and supporting Verilog code, in a single, well-named archive file (ZIP or TAR). Please name this file after your team. So if you are *Team Smack*, your directory would be "teamsmack", and you would ZIP that up into a file "teamsmack.zip".

Other notes:

- You may turn in one deliverable for all group members
- Please email your documents to myself and Joe College
- We expect all group members to participate in every aspect of this machine problem
- Please check out the tutorials on the class wiki. They are actually useful, I promise.

### 5.2 Demos

**DEMOS ARE REQUIRED, WHETHER YOUR CODE WORKS OR NOT**

We will be putting together demo dates that start *after* the due date of this machine problem. We will be using the wiki to coordinate times. Your deliverables are still due on the date at the top of this assignment, in class.

The Demo is when your team convinces us that your implementation does what it was supposed to do. This is accomplished using a combination of your test benches and our custom test benches. It is also a time for us to gauge the level of involvement of each of the group members.

If you do not demo your assignment, your team will automatically get a zero. Missing your demo slot without prior approval will impose a late penalty on the entire assignment. All team members should be present for the demonstration unless a prior arrangement has been made.

## 6 Hints and Tips

### 6.1 Multiplexors

The most difficult part of this machine problem is constructing the large 32x32 to 32 multiplexors. One way to do this is to first construct a 32 to 1 multiplexor and use 32 of these 32 to 1 multiplexors to construct the larger 32x32 to 32 mux.

### 6.2 The D Flip Flop

Here is the Verilog code for the D Flip Flop. Note that it is Behavioral Verilog code. You may use this as-is in your code. Do not modify it.

```
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
```

## 6.3 Enabled D Flip Flop

There are lots of ways to "enable" your D Flip-Flop. As shown above, the D_FF module takes in a new value on every rising clock edge. Somehow, your team needs to make this not happen unless it is enabled. Not hard. But, you can't "gate the clock" ... see below.

## 6.4 Gating the clock

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, ie. 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, making you very, very sad.

## 6.5 Gate delays

In order to model some sort of delay for our gates, simply put these statements at the top of your Verilog source:

```
// define gates with delays
‘define AND and #50
‘define OR or #50
‘define NOT not #50
```

Then, when you go to instantiate an AND, for instance, instead of using just "and", use ‘AND. That is, back-tick followed by the define you specified. Think of the back-tick as a macro definition.

That means that the gate, ‘AND, has a delay of 50 units. Then, in your simulation, you should wait between transitions of the input long enough to allow the signals to propagate to the output of your circuit.

## 6.6 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).

## 6.7 Balanced circuits

Balanced circuits are nice, however if you have to add twice the circuitry to get a balanced circuit, you might want to rethink it. Speed is important. Balanced circuits are important. But you want to get an overall good circuit that is speedy. Do not sacrifice one in the face of another. Also, balanced circuits are nice, but with the use of clocked state-holding elements, we can deal with glitches because we ignore the outputs of combinational logic in between clock transitions. So, just go about it sensibly.

## 6.8 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. Write a test for the MUX2, write one for the enabled DFF, write one for the larger muxes and the decoder. Trust me, it will help.
- Consider using code generators. For repetitive Verilog statements that vary by only a few digits, it is trivial to make a loop in Python to generate lots of Verilog programmatically. This is a fantastic short-cut.
- 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.