ENGR 3410: MP #2
MIPS 32-bit ALU

Due before class on 1 November 2010

1 The Problem

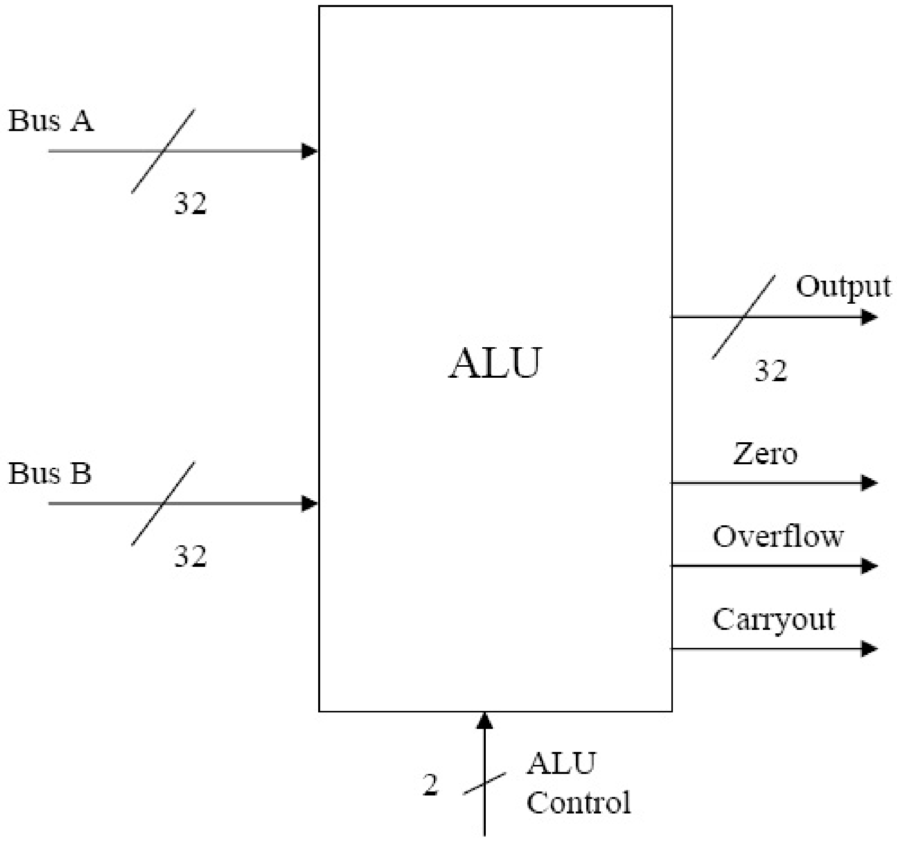

Figure 1: A block diagram of the ALU.

The purpose of this machine problem is to create the arithmetic-logic unit of your MIPS-style microprocessor. You will be designing and implementing a simple 32-bit MIPS ALU. The ALU functions to implement are ADD, SUB, XOR, and SLT. Examples of this type of architecture are shown in our textbook. The overall block diagram of our design will look a little like the Figure 1.

2 Implementation

The ALU has 7 ports. These ports are the two input ports A and B, the output port, ALU control, zero detect output, overflow detect output, and the carryout output. The ALU control line assignments are given below. You must use these inputs to select the ALU function.

ALU CONTROL LINES	FUNCTION
00	ADD
01	SUB
10	XOR
11	SLT

My estimate for completion of this machine problem is approximately 20-30 person-hours. This machine problem is in some ways harder, and some ways easier than the last.

One way to easily break it up is to develop the ADD/SUB first, the XOR/SLT next, and put it all together in the last several days.

3 Lab requirements

- Use the file `alustim.v` as your test bench. You can find this file on the wiki. You should alter the testing as necessary to make sure your unit works. I have my own test bench for use during the demos, so you must make sure your ALU takes the same inputs and outputs, in the same order, as is presented in the provided test bench. 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.

4 Deliverables

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

4.1 Write-Up

I expect a semi-formal lab write-up of this machine problem. The goal of this write-up is to demonstrate and explain your testing methodology and find out how long the machine problem took you. This is the one area we can't see in the demos and isn't apparent in the code.

Please describe how you knew you were done testing and ready to turn in the machine problem. This should include testing approaches to all submodules, if you do that.

You do not need to provide waveforms and test benches in the write-up. The test frameworks should be in the code you submit. You should *explain* what your tests do, in english, and how those tests verify the operation of your circuits.

Designate one member of your team as the submission vehicle. In that person's SVN directory, create a directory called `mp2`. Put your write-up (PDF) and supporting Verilog code in that `mp2` directory and commit the files to the repository.

Other notes:

- Turn in one deliverable for all group members

- All group members must participate in every aspect of this machine problem
- Please check out the tutorials on the class wiki. They are actually useful, I promise.

4.2 Demos

DEMOS ARE REQUIRED, WHETHER YOUR CODE WORKS OR NOT

The Demo is when your team convinces us that your implementation does what it was supposed to do. This is accomplished by your team running our test bench file. I will distribute this file about 12-24 hours before the assignment is due so you have a chance to plug your register file in and make sure it works.

The Demo time is also a time for us to gauge the level of involvement of each of the group members. Demos will be done *in class on the due date.*

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.

HONOR CODE INFORMATION HERE: Your team must complete the assignment **before** you try the official test bench. Your team will **not change the code after this time** except to correct errors. If you have to change the code, you must let the grader know that you have made modifications to the code. During the demo you must demonstrate your broken code first, then any modifications you did to fix it.

5 Hints and Tips

Some of these are repeated from the last machine problem because they are so important.

- 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 an ALU 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.
- As with the last machine problem, the provided test bench is really just a skeleton for something **you** should be writing to test each module you make. The testing here is **far** from exhaustive, and **far** from acceptable. Heck, it doesn't even try and test the XOR or SLT instructions. This is *deliberate.* You're given enough examples to see how to construct the tests, so you can figure out the test cases yourself.
- 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.