ENGR 3410: MP #2
MIPS 32-bit ALU

Assigned: October 12, 2007
Due: October 23, 2007

1 The Problem

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 below (Fig. 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. Please 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

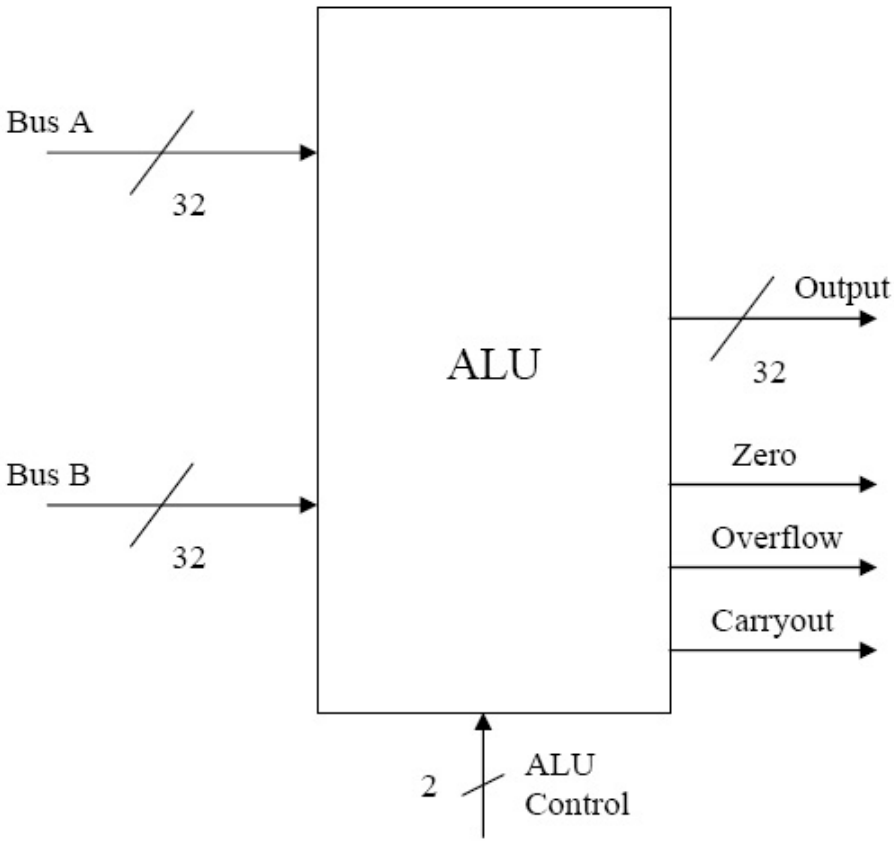

Figure 1: A block diagram of the ALU.

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 *three* deliverables. A coding check in, a write-up with code, and a demo.

4.1 Check-in

In order to keep you on track, your team will be working on this lab in class on October 18th, and *must demonstrate what you have done, in person, on October 25th, in class.* It will be a quick assessment, and you will be graded based upon your overall progress toward the final deliverable.

4.2 Write-Up

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. You do not need to make completely detailed diagrams of your register file.

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 John Morgan
- 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.

4.3 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.

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*. I give you enough examples to see how to construct the tests, while having you 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.