ENGR 3410: MP#3
Full MIPS Single-Cycle CPU

Due in class (with demo), 15 November 2010

1 The Problem

The purpose of this machine problem is to complete your simple 32-bit MIPS single-cycle CPU. The CPU instructions to be implemented are: `LW`, `SW`, `J`, `JR`, `BNE`, `XORI`, `ADD`, `SUB`, and `SLT`.

The book will give some examples of how architectures are put together, and will be useful as you design your own CPU. For this CPU, you will use your two previous machine problems (the register file and the ALU) so you will need to have these fully functional before proceeding to work on your CPU. *Contact us immediately if you do not have working code you can use!*

You are given some fake memory Verilog modules, an assembler, some assembly code source, and some compiled machine code to get you started. All of this is available on the wiki.

You need to test more thoroughly! You should very much be writing your own assembly code to test your CPU.

2 Implementation Details

The data memory and instruction memory modules are in the files `datamem.v` and `instrmem.v`, respectively. There are also a bunch of test programs to help you test the functionality of your CPU. You can change the program loaded by editing the `instr.dat` string in `instrmem.v`. You are responsible for coming up with the top-level testbench for this assignment — use previous machine problems' testbenches guides.

You will be given an assembler that will allow you to assemble your own assembly-language test benches into machine language that can be loaded into your system for testing. The tools include a Windows command-line executable on the wiki. If you have trouble executing the assembler, please let me know immediately. *Please note, this is not a commercial assembler. Be gentle.*

The control logic for your CPU can (and should) be done in behavioral Verilog.

This machine problem is significantly more complicated from a system integration perspective than the previous ones. Certainly, there are many more failure points in this problem than in the previous as there is much more integration of parts. Additionally, if your previous machine problems do not work correctly, you will need to make sure they are functional before you can test your CPU.

My estimate for completion of this machine problem is approximately 30 person-hours.

3 Requirements

There is no top-level test bench! You are designing your CPU, you must design a way to reliably test it. These tests will be used to prove to us that your CPU works. We have our own tests. These will take the form of **machine code** that gets loaded into memories.

As before, you should be writing test benches for **every** module you build. You are getting good at Verilog, I understand, however, this is a big machine problem, and you will get something wrong in putting it all together. Tests are your friends.

One good way to structure your test bench is to make a generic CPU test bench that instantiates your entire CPU (control and datapath) as well as both memories. These memories are simply Verilog files that fake memories and fill them with values from another file. **Look at the code for examples of using memories.** Then, you just change the contents of the memories and voila, your CPU executes a different piece of code. Observe the results as generated in the register file or in the input to the data memory write line. You have to instrument these "probes" for any useful data to come out.

Write some assembly code! Assemble it with the provided assembler and make sure your CPUs work!

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. It does not need to be as rigorous as lab notebooks in other, more experimental classes.

Designate one member of your team as the submission vehicle. In that person's SVN directory, create a directory called `mp3`. Put your write-up (PDF) and supporting Verilog code in that `mp3` directory and commit the files to the repository.

Other notes:

- Turn in one deliverable for all group members.
- 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.

4.2 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 demo time so you have a chance to plug everything together and see if 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 *during class time on the due date*, or before the demo time, offline, with results submitted.

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

- Did you notice that there were tons of demo assembly codes and no real CPU test harness? You have to write it. It literally instantiates the CPU parts and ticks a clock. Easy. Now build the CPU.
- You must write some assembly code and test it to make this complete!