**EE108B – Kyoto Spring 2011-2012**

**Lab 3**

**Due Thursday May 17th**

**Introduction:**

Pipeling has introduced huge performance gains to the processor. With these performance gains, there have been additional complexities in the design and implementation of these processors. In this lab, you are going to have a chance to explore some of these challenging design issues as you pipeline the processor from Lab 2.

**Requirements:**

To get you started we will provide a processor with pipeline registers already inserted. You will need to add hazard detection, provide forwarding, and stall when data cannot be forwarded. The TAs will run a test script on your processor to verify that it handles all cases properly.

We will be testing the forwarding corner cases and decucting points for the cases that do not work. Thus, when you come to us ready to demo, make sure everything works. Lastly, using ModelSim will be very instrumental in debugging.

**Implementation Details:**

*Processor Model*

As with Lab 2, it is important that you study the model we have provided before you begin coding. This model is slightly different from the approach in the textbook. The Lab 2 processor has been divided into 5 pipeline stages, which contain the following modules:

* Instruction Fetch – IF.v
* Instruction Decode – RegFile.v and Decode.v
* Execute – ALU.v
* Memory – MemStage.v
* Writeback – no explicit module

To make the code more readable, we have used a naming convention which we encourage but do not require you to follow. Appended to each signal name in MIPS.v is the name of the stage in which the signal is used. For example, the signal which determines whether the instruction currently in the EX stage will involve writing data to memory is called MemWrite\_ex.

There are two important changes to note. First, unlike in the textbook, the ALU operands are chosen in the ID stage, not in the EX stage. As a result, the ALUSrce signal has disappeared, and all forwarding logic which you create will need to appear in Decode.v or RegFile.v. The second change arises from the fact that jumps and branches need to be resolved in the ID stage in order to avoid multiple stalls in the pipeline. Instead of using the ALU to resolve branches through the use of the ALUZero or ALUNeg wires, the register comparisons should occur in the ID stage.

*Verilog Code*

The model we have provided does not yet handle hazards, and a few signals need to be assigned before it will run a program (for instance, the reset signals in the top level module need to be assigned). A good approach to doing this lab is to write in the basic necessary logic and then start your testing with a program that has sufficient nops between dependent instructions. Verify that the program runs correctly on the processor. As you add support for forwarding and stalling, remove the appropriate nops and check that your program still works.

Unlike Lab 2, we have not provided all of the signals that you will need. It is up to you to decide what signals are required for forwarding. Remember that forwarding should occur in the ID stage. You should not have to edit any modules except for RegFile and Decode, and depending on how you choose to implement forwarding, you may not even need to edit both of these modules. Regardless of how you choose to approach this lab, be sure to write clean, commented code.

As for stalls, the original MIPS processor was designed to simplify the hardware design. In order to prevent stalls on branches and load instructions, delay slot(s) had to be filled with an independent instruction or a NOP. However, in modern processors where the issue width has increased and pipeline stages have become longer and more complex, it has become more difficult to fill these delay slots.

For this lab assignment, we will have one branch delay slot with the branch condition being resolved in the ID (second) stage. Thus no stalls will be required on branch instructions. However, you need to determine the slight modification that is required to account for the fact that there is a branch delay slot.

In the case of load instructions, unfortunately there is no load delay slot. Thus, you need to consider the stall when a load instruction is followed by an instruction that depends on it. An example is shown below:

lw r3, 4(r7)

add r5, r4, r3

If there are no stalls, the add instruction above will use the old (stale) value of r3 register. In order to obtain the proper value of r3, the proper interlocks need to be added to the model and the proper data needs to be used.

You need to modify the hardware in Verilog to support this situation. (HINT: this part can get more complex than necessary. Think of an easy way of detecting this stall situation without with out too much complexity.). Note that the pipeline registers for each of the stages have both an enable and reset signal. In order to allow for stalling, make the proper assignments to the enable and reset signals for each stage.

You will be making very few additions in this lab, but they require much thought. Before jumping in and making changes, make sure you know what you are doing. When you make changes, thoroughly test it before proceeding with new changes.

*Simulation and Hardware Implementation*

Please refer to the Lab 2 handout for details on using the ModelSim and the hardware. To guarantee that your model properly handles pipelining, hazards, and stalls properly, make sure the cycle count is appropriate – i.e. unnecessary interlocks or stalls will force the machine to take a longer number of cycles to execute.

**Write-up and Submission:**

You must submit a write-up via the website.

In your submission, you must include the following:

1. All your lab code (only the files you modified)
2. All testbenches that you wrote
3. Screenshots of testbenches that demonstrate correct functionality in the following cases:
   1. General ALU op
   2. General lw
   3. General sw
   4. General branch
   5. General jump
   6. All forwarding cases (from X to D, X, and M; from M to D, X, M)
   7. Stall on a load followed by a dependent ALU op
   8. Stall on a load followed by a dependent store
   9. Stall on a load followed by a dependent branch

You will be required to show the screenshots above during your demo.

Please submit a .zip or .tar file via the website. Please keep your report and your files separate (e.g. do *NOT* paste your code at the end of the report). Also, remember, please submit your report in PDF form.

There are no extensions for this lab.

Please answer the following questions in your report:

1. What do you believe to be the critical path in the processor? Explain, and then give some suggestions for reducing the critical path (no need to implement them).
2. Is there any way to eliminate the need for stalls following a load instruction in all cases? If so, how would you do this, and what problems would this cause? If not, explain what prevents this particular hazard from being eliminated?