EE108B Prof. Kozyrakis
Winter 2012-13

EE 108B Lab Assignment #3
Pipelining
Due: Tuesday, February 26, 2013

1. Introduction

Pipelining has introduced huge performance gains to the processor. With these performance gains, there has been additional complexity 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.

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. We will be testing your implementation on many forwarding and stalling corner cases. You will be required to:

1. Write an instruction sequence that demonstrates correct forwarding and stalling behavior of your processor. (Your processor should never use stale data in the presence of a RAW hazard, and it should also never stall unless it has to in order to resolve the RAW hazard.)
2. Using your implementation of Pong from lab 1, write an instruction ROM to synthesize with your design on the Xilinx Virtex 5 FPGA and demonstrate it.

3. 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 processor from Lab 2 processor has been divided into 5 pipeline stages, which contain the following modules:

- Instruction Fetch – instruction_fetch.v
- Instruction Decode – decode.v (and regfile.v)
- Execute – alu.v
- Memory – memory_top.v
- Writeback – regfile.v

To make the code more readable, we have used a naming convention, which we encourage you to follow. Appended to each signal name in mips_cpu.v is the name of the stage in which the signal is used (or generated, in cases where stages must communicate with each other). For example, the signal which determines whether the instruction currently in the EX stage will involve writing data to memory is called mem_we_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, there is no alu_src signal, and all forwarding logic that you create will need to appear in decode.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 BEQ and BNE branches, the register comparisons occur in the ID stage.

Verilog Code:
The model we have provided does not yet handle hazards. 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, so you should only have to edit decode.v and the portion of mips_cpu.v where the decode module is instantiated.

As for interlocks (stalls), the original MIPS processor (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) was designed to simplify the hardware design. In order to prevent interlocks on branches and load instructions, a delay slot had to be filled with a non-dependent instruction or a NOP. However, in modern processors where the issue width has increased and pipelines 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 Decode stage. Executing a branch instruction will never be the cause of a stall, but the branch instruction itself may depend on previous instructions.

In the case of load instructions, we want to have no delay slot, so you need to implement a stall when a load instruction is followed by an instruction that depends on it. An example is shown below.

```
lw $t1, 4($t0)
add $t3, $t1, $t2
```

If there were no interlocks, the `add` instruction above would use the old (stale) value of `$t1`. In order to obtain the proper value of `$t1`, interlocks (stalls) need to be added to the model and the proper data needs to be used. (If we wanted a load delay slot, we would simply ignore the hazard and let `add` use the stale value.)

You need to modify the hardware in Verilog to support this situation. (Beware: this part can get more complex than necessary. Try to think of a way to detect stall conditions with as little logic as possible.)

Note that the pipeline registers between the F and D stages have their own special enable signal, en_if, and the registers between the D and X have their own special reset signal, rst_id. In order to allow for stalling, the starter code assigns these signals using the stall signal. Make sure you understand why these signals are set the way they are.

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 your implementation before proceeding with new changes.

4. Simulation Instructions

You should use the provided Makefile to compile and simulate your design:

```
make // this compiles the design into an executable
make test // runs the executable (also runs make)
make wave // opens the waveform file
make text // runs a testbench that displays instruction info as text
make display // opens the display simulator (also runs clean and test)
make clean // removes test output files
```

Notes:
Any of the make targets can be run on their own and will do the correct thing. For simplicity, however, when using `make wave` simply run `make test` again and refresh from within GTKWave to avoid creating a new GTKWave instance after making changes to your code.

When you are ready to synthesize your design (i.e., you have written a thorough irom that covers many test cases and verified correct functionality), you can open the Xilinx synthesis with:

```
ise &
```

From ISE, open the project file `synth/mips.xise`, then generate the programming file and use iMPACT to program the FPGA with your design.

5. Submission Requirements

You will demonstrate your lab simulations and synthesis to the TA during office hours on the due date (or at an alternate time that you must arrange in advance).

You must submit your implementation and lab report to the drop-off box on CCNet by 11:59PM on the due date. Please refer to the lab report guidelines handout for what must be in your report.

In addition to your lab report, you must include the following in your submission:

1. The whole project directory with your modifications (the Makefile should still work).
2. All instruction ROMs you wrote (including Pong, and one for your debugging).

You will be required to demonstrate that your Pong game works on the FPGA. If you implemented the user control extension in lab 1, you can use it on the FPGA with slight modifications. Ask the TA if you are interested in doing this.

6. Additional Questions

There are no extensions for this lab, but please answer the following questions in your lab report (make a clearly labeled additional section in your report):

1. According to the Xilinx synthesis report, what is the maximum frequency for your pipelined processor?
2. What did you need to change about your implementation of Pong from Lab 1 in order for it to run on your processor? Why?
3. Where is the critical path in this processor? (Use the synthesis tools to find it.) How might you reduce the critical path?
4. 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.

7. Grading Rubric

Lab Report:
5 points – following report guidelines (no missing sections, title page, etc.)
20 points – thorough design discussion

Code:
5 points – turning in your code
45 points – passing automated tests

Demonstration:
25 points – synthesizing a design and showing Pong

Total: 100