EE 108B Lab 3: Pipelining

Darren Hau

2/26/13

Introduction:

In this lab, I implemented forwarding and stalling in a MIPS 5-stage pipelined processor. I developed the proper conditions for both forwarding and stalling and inserted them in the decode stage. This eliminated all stalls except for dependencies following a lw. I tested the correctness of the design by running it through a test sequence and analyzing the resulting waveforms. Then, I converted my pong.s file from Lab 1 into the irom.v format and demoed the synthesis.

Design:

1. **Forwarding**: I implemented forwarding by checking for dependencies between the decode stage and the execute and memory stages. By reading through the starter code and reviewing lecture notes, I created a set of conditional statements that determined two 2-bit signals, forwardA and forwardB. The values of these signals indicate whether data should be forwarded from the execute or memory stage. To make my code more readable, I broke the final signals into smaller signals - forwardA_ex and forwardB_ex for a hazard from the execution stage, and forwardA_mem and forwardB_mem for a hazard from the memory stage. The basic concept is that we need to forward if one of the two previous instructions writes back to a register, and the write-back register is one of the two source registers currently in the decode stage. In order to ensure that we forward the most recent value in the case of a double-dependency, I enabled the forward_mem signals only if the forward_ex signals were 0.
2. **Stalling**: We only needed to stall for RAW hazards starting with a lw. A stall is generated if you have a lw, and the destination register of the lw is either of the source registers for the instruction currently in the decode stage. These RAW hazards therefore can be distinguished as “rs_RAW” or “rt_RAW.” However, it is important to note that every instruction has data in the rs and rt fields, although the data may be invalid because the instruction does not use rs or rt. Therefore, I added rs_enabled and rt_enabled signals - we only check for rs_RAW or rt_RAW the corresponding “enabled” signal is true. The trickiest part was figuring out all the cases that did not use either rs or rt.
3. **Verification**: To verify my design, I created a test sequence in irom.v. I first created a test sequence with sufficient NOPs spaced between dependent instructions as a sanity check. It was through this check that I realized I had to store to a register other than 0xffff000c if I wanted to load the value back out later. When I had a good test sequence, I eliminated the NOPs and checked the results again. The only issue was an odd unknown signal in forwardB, which I discovered was the result of a typo. I was ultimately able to get the entire sequence operating perfectly.
4. **Pong**: To implement pong, I used a classmate’s MipsAssembler.jar file. I had to make significant changes to pong.s from Lab 1. First of all, no pseudoinstructions were accepted, so I converted pseudoinstructions to multiple real instructions. Next, the translator couldn’t interpret hex values, so I converted all hex values to decimal values. $sp also had to be initialized to 0. Instead of using the write_byte function to write color, x-coord, and y-coord separately, I had to load all the values into one register and store

that register into 0xffff000c. After translating the assembly into MIPS instructions, I need to insert NOPs after branches and jumps, and change the corresponding offsets and destinations accordingly.

Results:

My design resulted in seemingly perfect waveforms from my test sequence (hope it works well with the automatic tester!). However, I wasn't quite able to make Pong work during my demo with Chris. The paddle was half-drawn and the ball was drawn, but neither moved. I believe it was an error in the translated irom.v program. Fortunately, the design likely works on an accurate program.

Conclusions:

This project was really fun in understanding forwarding and stalling. I was impressed that so many nuances could be captured by around 10 lines of conditional code. This was a great way to test my understanding of the details of the MIPS instruction set and the pipelined processor. Debugging the forwarding and stalling with an irom.v was also educational. However, translating the pong file was more trouble than it was worth. I think there was little gained in terms of educational experience that I couldn't get from coding a test sequence. Seeing the MipsAssembler.jar file work was pretty cool though!

Unfortunately, my partner, Krista Fryauff, did not contribute to this lab at all. I tried throughout the week to get in contact with her via text and email, with no results. She responded once saying she would come into OH but never showed up. Although the work has been split 70-30 in my direction for all the past assignments, Krista's been a helpful and fun partner, and I would like to keep working productively with her. Unfortunately, I'm starting to get concerned about the lack of communication and unbalanced distribution of work.

Additional Questions:

1. According to the Xilinx synthesis report, the maximum frequency for my pipelined processor is 68.090 MHz.
2. To implement pong, I used a classmate's MipsAssembler.jar file. I had to make significant changes to pong.s from Lab 1. First of all, no pseudoinstructions were accepted, so I converted pseudoinstructions to multiple real instructions. Next, the translator couldn't interpret hex values, so I converted all hex values to decimal values. $sp also had to be initialized to 0. Instead of using the write_byte function to write color, x-coord, and y-coord separately, I had to load all the values into one register and store that register into 0xffff000c. After translating the assembly into MIPS instructions, I need to insert NOPs after branches and jumps because of the branch delay slot, and change the corresponding offsets and destinations accordingly.
3. The critical path in this processor appears to be from ctrl/gen_sync/x_9 to ctrl/make_chip_data/pixel_21, which consumes 7.473 ns out of 7.544 ns. To reduce this critical path, we would have to pipeline the display code further.
4. The only way to eliminate all stalls following a lw instruction is to place the memory stage

before the execution stage. However, this would cause its own slew of stall cases, such as a dependent EX-MEM sequence (add $t0, $t0, $t1 followed by sw $t0, 0($t1)). The fundamental problem is that either the EX or MEM stage has to “forward backward in time” to the other stage, which is impossible.