CS 231 – Lab 8 – Simple Pipeline Simulation

In class we saw how an instruction's execution is broken up into various *stages*, such as fetch, decode, execute, memory and write back. Today we will be running and making small modifications to a high-level language program that simulates these steps by printing a pipeline diagram.

The basic skill for today is being able to detect structural and data hazards. As you know, a structural hazard occurs when the previous instruction is taking extra time in the EX stage. A data hazard generally occurs if the previous instruction is a load and one of our source operands is the same as the destination of the load (and not the zero register). There are other examples of hazards, but for the purpose of this lab, just focus on these two. We will not be addressing floating-point instructions or branches today.

Create a new directory called lab8 on your USB drive or account, and do all of today's work in there. On the class Web site you should see a folder called lab 8. Please download the files 1.txt, 2.txt and 3.txt. There is also the source code of an unfinished pipeline simulator for you to download and then modify during the lab. If you want to work in Python, then you should use pipe.py. If you prefer Java, then you will need Pipe.java and Instruction.java.

1. I recommend you print out the source code so that you can get acquainted with the structure of the program. Notice the big while loop in the main program that handles one cycle at a time. Within this loop, we work on each stage one at a time starting with WB and finishing with IF. If the pipeline happens to be empty after moving the instructions over one stage, then we break from the loop and quit.

2. I have also included some sample input files, 1.txt, 2.txt and 3.txt. All three files give a list of assembly instructions. The purpose of your program is to depict the pipeline diagram for a given input file. The first example input case, 1.txt, is straightforward and should feature no stalls. But the other two input files illustrate common hazard situations that your program must handle. The file 2.txt should have a structural hazard because of the multiply, and file 3.txt should have a data hazard because of the load. But right now, our pipeline simulator is not designed to handle the hazards. It assumes exactly 1 cycle per stage no matter what.

   In the code, notice that I've put comments in the approximate places where you should add statements that detect the hazards.

   First, focus on input file 2.txt and the structural hazard. Assume that multiplies take 13 cycles in the EX stage and divides take 36. There is more than one way to do this, but here is one approach. You can use the ex_available variable to keep track of which cycle an instruction's time in the EX stage will complete, and then refer to this time to see if the instruction in ID can move over to EX. Verify your code works by checking the result for input file 2.txt. Instruction 1 should spend 13 cycles in the EX stage, and the other two instruction should be stalled behind it. √

3. Next, we can handle a data hazard situation. For the instruction in the ID stage, you need to see if its predecessor (now in EX) is a load instruction, and if there is a data dependence between these two instructions. Add your code near where I've placed the comments for this task.

   Hint: You probably need to keep track of when a register is "available" with my reg_available array. When you are executing a load instruction, you can set this time, and then when detecting a data hazard, you can look up the "available" times of the source operands to see if you need to stall the instruction (i.e. not move it to EX).

   Check your solution by running the program with input file 3.txt (and make sure cases 1.txt and 2.txt still work). You should notice a bubble in the pipeline because the 2nd instruction had to wait for the MEM stage of the 1st instruction. √

4. To test the simulator a little more thoroughly, create some more input files. They should illustrate certain situations of hazards – for example, trying a divide instruction, checking the first source operand versus the second operand or both. Also make a test case to make sure you don't incur a stall if the load instruction is loading into register $0. √