EC513 Computer Architecture
Lab 1
Worth 10%

Assigned Jan 30, 2018 ***Due Feb. 13, 2018***

http://ascslab.org/courses/ec513/index.html

Summary:

In lecture, we will soon introduce the concept of pipelining. We will observe that the main difficulty in pipelining are pipeline hazards, that is, data dependencies between instructions in the pipeline. We will discuss two possibilities for dealing with pipeline hazards in class: stalling and data forwarding. Stalling means that we stop issuing instructions when we detect that the next instruction that we will issue depends on the result of in-flight instructions. We resume issuing instructions once all the data dependencies have been resolved by the completion of the in-flight instructions. Although it is simple to implement and has little impact on the critical path and area of the processor in question, stalling cuts the throughput of the processor by introducing 'bubble' cycles in which no useful work is accomplished. An alternative to stalling is data forwarding. Rather than waiting for the instruction to complete and commit data to the register file, we forward the data from a earlier instructions in the pipeline directly to later instructions. Data forwarding requires more hardware to implement than stalling. However, data forwarding has higher throughput since many stall cycles are avoided. Given this choice of how to handle hazards, which one should the architect choose?

When investigating design tradeoffs, architects typically run simulation experiments to determine the merits of the proposed architectures. To answer the question posed in the preceding paragraph, you will analyze the read after write data dependencies of a series of benchmarks from the PARSEC and SPLASH2 test suite using Pin. Armed with this information you will decide which architecture should be implemented to handle pipeline hazards.

As always, this lab is to be completed individually. You are encouraged to discuss lab concepts with fellow classmates.

Setting up:

To obtain the materials for lab 1, download the file at:

http://bit.ly/2nog415

Extract the file in your **labhandouts/** directory, next to lab 0.

In the lab1handount directory that just got extracted, you should find a make file, some sample source code, a PDF file and a test script. Type the following at the command prompt:

```
% cd lab1handout
% chmod a+x lab1test.pl
% make
```

Make will build the regDeps Pin tool. Attempting to run the Pin tool will result in a failure message, stating that no Pin tool has been implemented. The regDeps Pin tool can be invoked from the command line in the following manner:

```
% pin -t regDeps.so -- target_executable
```

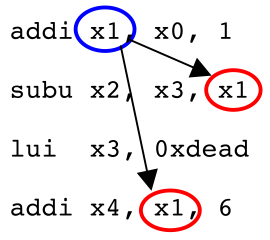
We have provided a test perl script `/lab1test.pl`. The perl script will invoke Pin using the regDeps Pin tool on multiple PARSEC binaries. To invoke the perl script, type:

```
% ./lab1test.pl
```

Although we provide a script that will test your Pin tool, the script will not verify your Pin tool, and we will not release the expected results of the test cases. Further, we reserve the right to run test cases not included in the released test script. You are encouraged to compare your results with your classmates.

Lab Task:

The purpose of this lab is to generate a histogram of the distances between instruction dependencies in a set of benchmarks. We define instruction dependency distance to be the number of instructions between when a register is written by instruction and when the register is read by a subsequent instruction. In the following toy example written in RISC-V like assembly, x1 has dependency distances of one and three.



```
addi x1, x0, 1
subu x2, x3, x1
lui  x3, 0xdead
addi x4, x1, 6
```

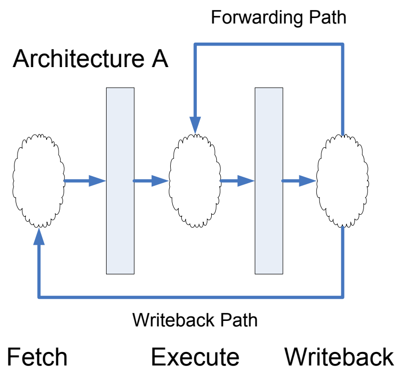
You should instrument each instruction with a function that will determine the registers read and written by the instruction and update the register histogram accordingly. You may find the functions INS_MaxNumRRegs, INS_MaxNumWRegs, INS_RegW, and INS_RegR to be quite helpful in writing your instrumentation function. Since we are using the x86 architecture, some instructions may read from or write to partial registers (e.g %al). You should treat these partial reads and writes as being reads and writes to full registers. You may find the function REG_FullRegName useful. Some instructions may read or write the same register more than once. Be careful not to double count dependency distances. The code we have provided you contains both a dependency histogram array (dependencySpacing) and a Fini function which dumps the dependency histogram data into a file. The dependency histogram should be update each time a dependency is detected. The index into the histogram array represents the dependency distance. For r1 above, dependencySpacing[1] and dependencySpacing[3] would be updated, although these would not be the only updates to the histogram. For initialization purposes, assume that all machine registers have been modified at time 0. The Fini function will be called at the end of program execution and will produce a CSV file for easy importing into a spreadsheet program. Do not modify the Fini function, or you will not pass our test benches.

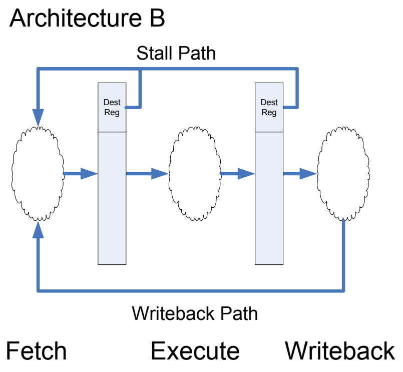
Although your solution will not be graded on its performance in terms of wall clock time, you should note that your instructor is an impatient person. His solution runs the sample testbench in less than one hour on the class machines (with nominal load). For grading purposes, we will allow your pin tool to run for an order of magnitude more time than ours requires (around 10 hours). After ten hours, we will kill it and assign a grade based on progress to that point.

Lab Questions:

Your response to the lab questions should be typed in lab1questions.pdf in the lab1handout directory. The course material necessary to answer some of the questions will be covered after the distribution of this handout, so if some of the material looks foreign, it probably just hasn't been covered yet. Some questions may require coding, and as such should not be put off until the last minute.

1. The following images are rough sketches of two proposed architectures for a three-stage pipelined machine. The three stages of the pipeline are Fetch, Execute, and Writeback. Fetch determines the next instruction to be executed and fetches it from memory. Execute stage decodes the instruction and computes the result of the instruction; it also issues requests to memory (not shown). Writeback stage retrieves requests from memory and writes the results of execution back to memory. Architecture A implements data forwarding and architecture B implements stalling. We assume that the cycle critical path of the machine is located in the instruction fetch logic, and will not be affected by the choice of architecture. Choose one of the proposed architectures, and explain the reasoning behind you choice. You may include a graph to support your argument.

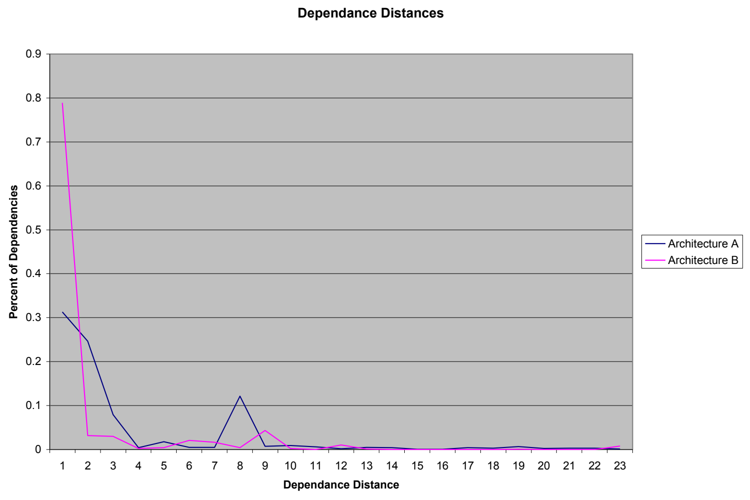


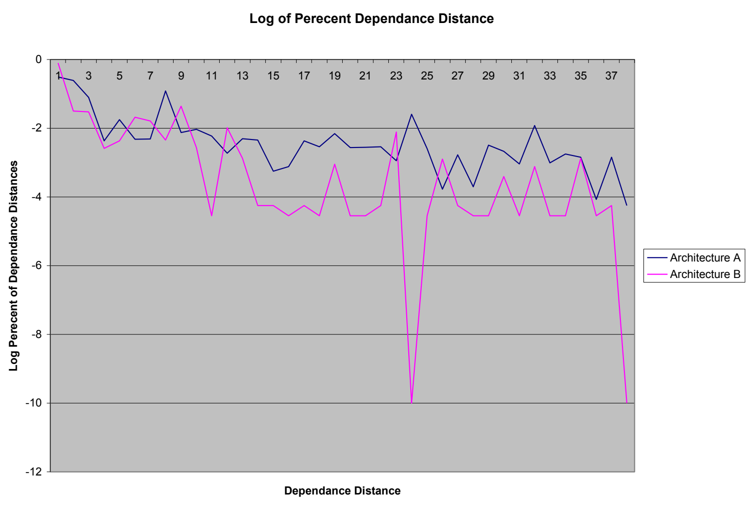


2. You should notice that the dependency histograms have fairly long tails. Which registers account for most of the weight in the tail? How can this behavior be explained? Suggest some architectural changes that we might make to take advantage of the difference in lifetime lengths of the architectural registers, paying particular attention to forwarding paths.

3. In the lab pin tool, we made a simplifying assumption regarding how to handle partial register reads and writes. The lab pin tool assumes that partial reads and writes to a registers touch the entire register. Clearly then, the lab pin tool is not completely accurate, since sometimes it might incorrectly count dependency distances. We have made an engineering tradeoff in the design of our pin tool. Explain the tradeoff and justify it by obtaining simulation results. In your opinion, is the tradeoff acceptable?

4. The following register dependency histogram graphs were obtained by running the same benchmark on different architectures. The same compiler (gcc 3.4) was used to compile both of the benchmarks. Based on the instruction dependency graphs below, can you tell which machine has more general purpose registers? Explain your choice.





Lab Grading:

20% - Submission compiles
20% - Submission passes public test bench
30% - Submission passes private test bench
30% - Quality of lab question responses.