6.823
Computer System Architecture
Lab 1

Assigned Feb 14, 2014 Worth 5% of Course Grade ***Due Feb 26, 2014***

http://csg.csail.mit.edu/6.823/

Summary

In lecture, we will shortly discuss the concept of pipelining for parallelism. Recall that the main difficulty in pipelining is pipeline hazards, that is, data dependencies between instructions in the pipeline. We will discuss two main 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 an 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 SPEC2000 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

First, as for Lab 0, set up your environment for Pin. You will need to do this each time you log in to work on the labs.

```
% add 6.823
% source /mit/6.823/Fall12/setup.csh (or setup.sh depending on your shell)
```

To obtain the materials for lab 1, use the following commands, assuming that you start in your individual repository (cd $USER) from the previous lab:

```
% svn export $LAB1ROOT
```

```
% svn add lab1handout
% svn commit -m "Lab 1 Initial Check-in"
```

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

```
% cd lab1handout
% make
```

Make will build the regDeps Pin tool. The regDeps Pin tool can be invoked from the command line in the following manner:

```
% pin -t regDeps – [target_executable]
```

However, attempting to run the Pin tool now will result in a failure message, "Warning: Pin Tool not implemented."

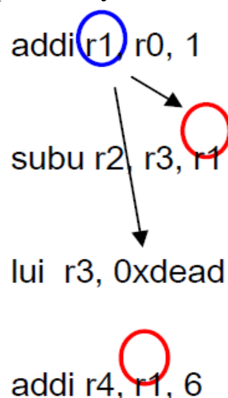
We have provided a test perl script lab1test.pl. The perl script will invoke Pin using the regDeps Pintool on multiple SPEC binaries. To invoke the perl script, type:

```
% ./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 MIPS-like assembly (with the modified register in the leftmost position), r1 has dependency distances of one and three.



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. 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 the corresponding full register. You will need to explicitly convert from the partial register to the corresponding full register in this case. Some instructions may read or write the same register more than once. Be careful not to double count dependency distances. You may find the functions INS_MaxNumRRegs, INS_MaxNumWRegs, INS_RegW, and INS_RegR to be quite helpful in writing your instrumentation function, though you may need other Pin functions as well. 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 updated each time a dependency is detected. The index into the histogram array represents the dependency distance minus one, which is equivalent to the number of other instructions “skipped over" by the dependency. For r1 above, dependencySpacing[0] and dependencySpacing[2] would be updated, although these would not be the only updates to the histogram in this code. 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 importation 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 Teaching Assistants are impatient people. The TA 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. Do not write horrendously inefficient code: it makes kittens sad.

When you have completed the lab to your satisfaction, submit your changes (not results) to the svn repository by running

```
% svn commit -m "Lab 1 Final Check-in"
```

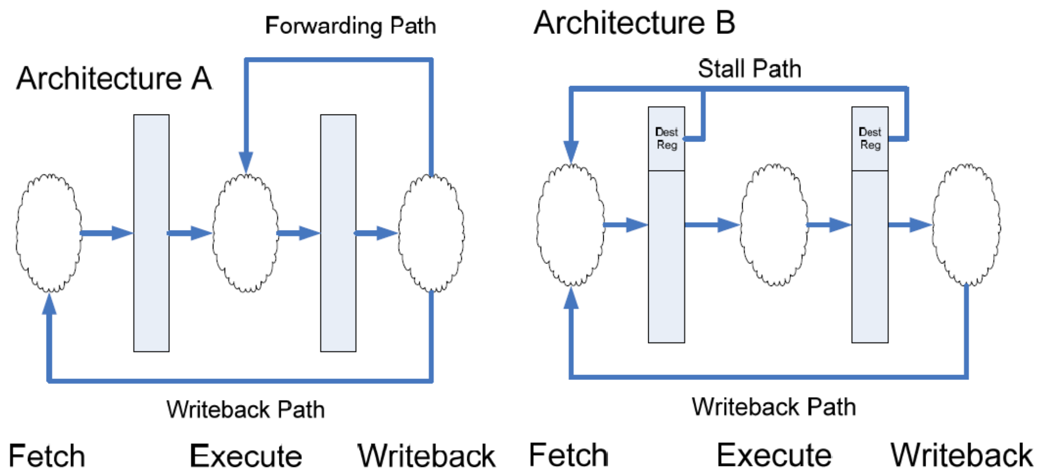
The deadline for submission is 23:59:59 EDT 24 September 2012. We'll grade whatever code you have checked in by the deadline. No Late Submissions will be accepted!

Lab Questions

Your response to the lab questions should be typed in lab1questions.pdf (or lab1questions.doc) 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. 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 your choice. You may include a graph to support your argument.

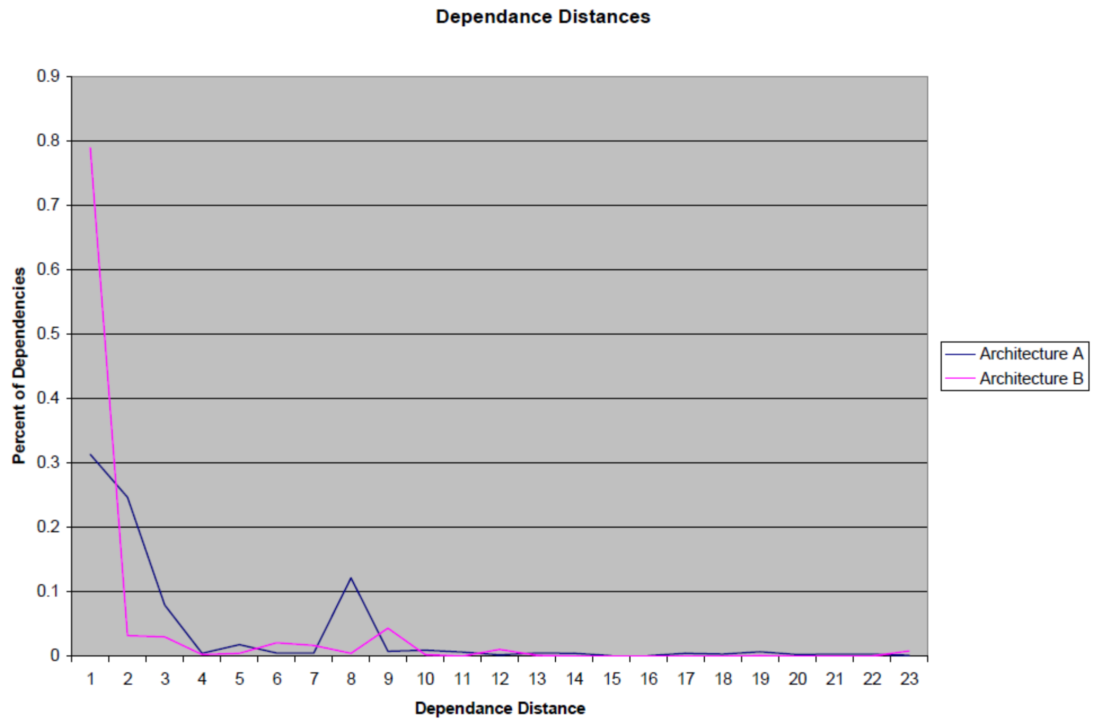


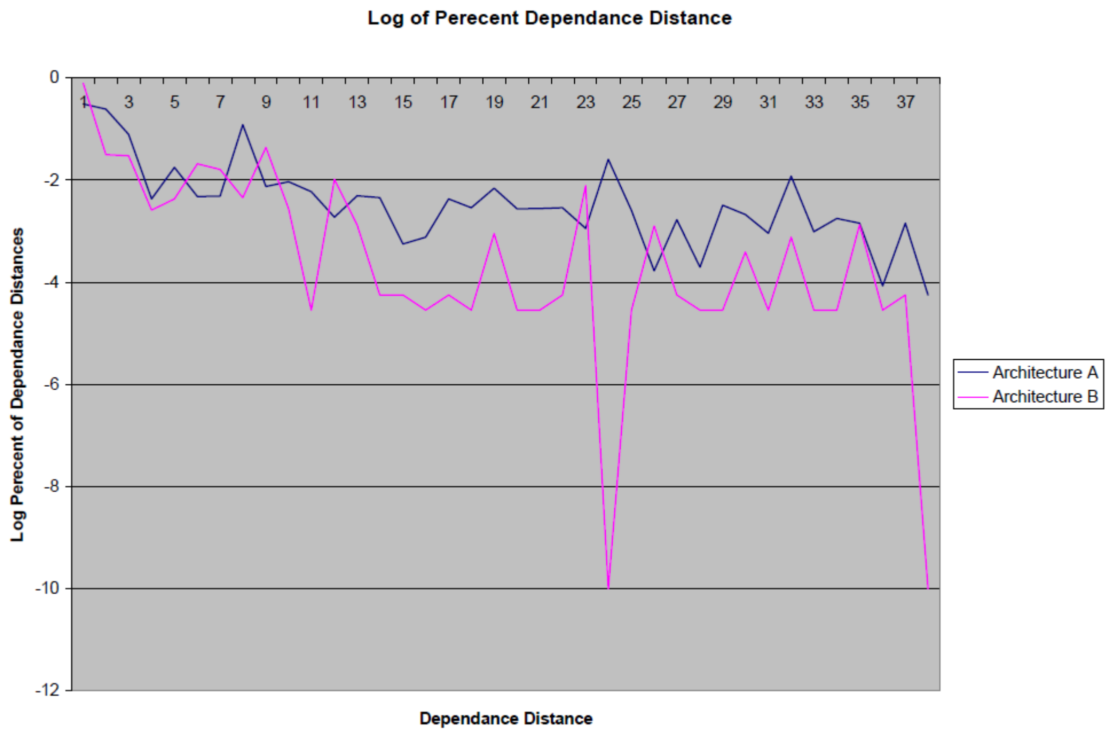
2. You should notice that the dependency histograms have fairly long tails. Which registers account for most of the weight in the tail? How can this behavior be explained? Suggest some architectural changes that we might make to take advantage of the difference in lifetime lengths of the architectural 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 histograms were obtained by running a benchmark on two machines. 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.





When you have answered these questions to your satisfaction, put them in a file called lab1questions.pdf (or lab1questions.doc) in your lab1handout directory, then run the following to add them and commit them.

```
% svn add lab1questions.pdf
% svn commit -m "Lab 1 Questions Check-In"
```

As with the lab code, we'll grade whatever you have checked in by the deadline.

Lab Grading

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

Advice on Mine Sweeping

There may be bugs in either our code or infrastructure. If you notice any `interesting' or `unexpected' behavior it could be a problem in the code or infrastructure that we provided. Report these bugs immediately to the TA, preferably in an email with the subject 6.823 Bug Report. This will help to ensure prompt fixing of any issues that may arise.

Guides for the perplexed

http://www.pintool.org/ - Pin home page
http://tig.csail.mit.edu/twiki/bin/view/TIG/UsingSubversionAtCSAIL – an SVN tutorial