

6.823
Computer System Architecture
Lab 0

Assigned Feb. 10, 2017

Worth Something

Due Feb. 17, 2017

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

Summary:

This term in 6.823 we will be using Pin, a binary instrumentation tool provided by Intel. The purpose of this lab is to familiarize you with compiling and running Pin tools, as well as to make sure that you are able to use the class infrastructure and computers. To test the infrastructure, you will build and run two small Pin tools.

In order to develop architectures that run programs efficiently, architects must thoroughly understand the properties of representative programs. Architects experiment with different programs to help them make design decisions about features that will be included in processors. There are a few approaches that can be used to evaluate the behavior a program. One is simulation. In this case, a software model of a processor is built, and the program executed on the model. Simulators have the advantage of being arbitrarily detailed – in theory one could build a SPICE processor simulator. Typically, architectural simulators give cycle-accurate timing estimates. The penalty for this level of detail is simulation speed: the more detailed a software simulator is, the slower it executes programs; advanced software simulators can simulate at rates of tens of KIPS (kilo instructions per second). A second option is code instrumentation. Code instrumentation collects information about program characteristics. In general, code instrumentation is less detailed than simulation, but code instrumentation is faster to implement and enjoys faster program execution. Thus code instrumentation can be very useful in guiding architectural decisions early in the development process, before detailed simulators are available. The simplest form of code instrumentation, terminal display statements (e.g. printf), is used by almost every programmer, computer architect, or otherwise. However, such manual instrumentation is time-consuming, both in terms of writing code and collecting execution results. A better choice is to use a meta language to describe the code instrumentation and develop a tool that will efficiently instrument the target program at compile or runtime.

Pin is a free (though not open), industrial grade binary instrumentation tool produced by Intel and used widely in industry and academia. Pin accepts as inputs a compiled Pin tool and a generic binary executable. A Pin tool is a C++ program that makes a series of calls to an instrumentation API and provides code for Pin to execute at instrumented locations. The executable itself is just-in-time compiled by Pin and the instruments are inserted. The code is then executed natively on the host machine. A major advantage of Pin is that it can instrument programs without requiring recompilation. Thus, even legacy binaries can be analyzed. Since Pin executes large portions of the target program natively, it can be very fast, however, this constrains the programs analyzed to the host architecture, namely x86. Yet, Pin is surprisingly versatile: new instructions can be emulated by hijacking unused x86 opcodes.

If you would like, you may install Pin on your home machine. Pin can be downloaded from <http://www.pintool.org/> and run on any Linux or Windows platform. Please note that Pin is under active development. Thus, it is frequently updated. If you install your own version of Pin, take care to update it periodically.

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

Setting up:

Although you can develop Pin tools on your home machine or any Athena computer, we will have eight dedicated machines for class use. These machines are

```
vlsifarm-01.mit.edu
vlsifarm-02.mit.edu
...
...
vlsifarm-08.mit.edu
```

The course machines use Athena passwords, but only class members may log into them. If you have trouble logging in to the course machines, you may not have an account. In this case, email the TA to obtain an account. If you choose to use your own machine or an Athena environment other than the lab machines, you are responsible to set up your environment.

We will use Subversion for starter code distribution and lab submission. If you are not familiar with Subversion source control, `svn help` is a good command to remember. In general, we will provide skeleton code that you will edit and check in before the submission deadline. As specified in the syllabus, no late hand-ins will be accepted, so submit early and often.

The lab handout repositories and your individual repositories are located on Athena afs. Therefore, you need afs permissions to access them. **[The TA will send out an email once the permissions are set]**. If you are unable to checkout your repository or the lab materials, please contact your TA so that permissions can be set up for you.

To obtain the materials for lab 0, use the following commands:

```
% ssh <athena username>@vlsifarm-01.mit.edu
% add 6.823
% source /mit/6.823/Spring17/setup.sh
% mkdir ~/6.823
% cd ~/6.823
% svn co $SVNROOT
% cd $USER
% svn export $LAB0ROOT
% svn add lab0handout
% svn commit -m "Lab 0 Initial Check-in"
```

The `svn co` will create a local directory with your user name. The `svn export` will create a `lab0handout` directory in your individual repository. In the `lab0handout` directory that just created, you should find two sub-directories `part1` and `part2`.

Part 1: Instruction Count

In the first part of this lab, you will develop a simple Pin tool to count the number of instructions executed by an application binary. In sub-directory `part1`, you should find a makefile, some sample source code, and a test script. Type the following at the command prompt:

```
% cd lab0handout
% cd part1
% chmod a+x part1test.pl
% make
```

You will notice that `make` fails to compile due to a parse error. While your TA is an expert in Python, his C is not so good. Fix his silly parse error and run the perl script. The perl script will invoke Pin using the `inscount0` Pin tool on multiple SPEC binaries. Pin can be invoked from the command line in the following manner:

```
% pin -t toolname -- target_executable [args...]
```

We have provided a test perl script `part1test.pl`. The perl script will invoke Pin using the `inscount0` Pin tool on multiple SPEC binaries. To invoke the perl script, type:

```
% ./part1test.pl
```

Executing the script will produce a results directory which contains a set of files each of which is named in the following manner:

```
ExecName_Arg1_Arg2_..._.out
```

As you might imagine, the naming scheme is based on the executable and arguments that are instrumented and analyzed by Pin tool. Since the `inscount0` Pin tool counts the number of instructions in the instrumented program, these output files will contain the number of instructions executed. If you are interested in the inner workings of a real Pin tool, examine `inscount0.cpp`. We will cover this tool in detail in recitation.

Part 2: Instruction Dependency Distance

For the second part, you will develop a Pintool that generates 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.

```

addi  r1, r0, 1
subu  r2, r3, r1
lui   r3, 0xde04
addi  r4, r1, 6

```

Dependency distance distributions provide useful insights to computer architects that inform design choices. For example, it captures the tradeoffs in stalling vs data-forwarding for handling pipeline hazards, which we will cover in lecture shortly.

To get started, go to the part2 sub-directory. You should find a makefile, sample source code and a test script. The code that we have provided does most of the heavy-lifting. It instruments every instruction, and captures the registers read and written. You will have to edit the `updateDependencyDistanceInfo()` function suitably. The code also provides 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 an output file for easy importation into a spreadsheet program. Do not modify the `Fini` function, or you will not pass our test benches.

To build your pin tool, do:

```
% 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 [-o outputFile -s maxSpace] -- <target_executable>
```

We have provided a test perl script `part2test.pl`. The perl script will invoke Pin using the `regDeps` Pintool on multiple SPEC binaries. To invoke the perl script, type:

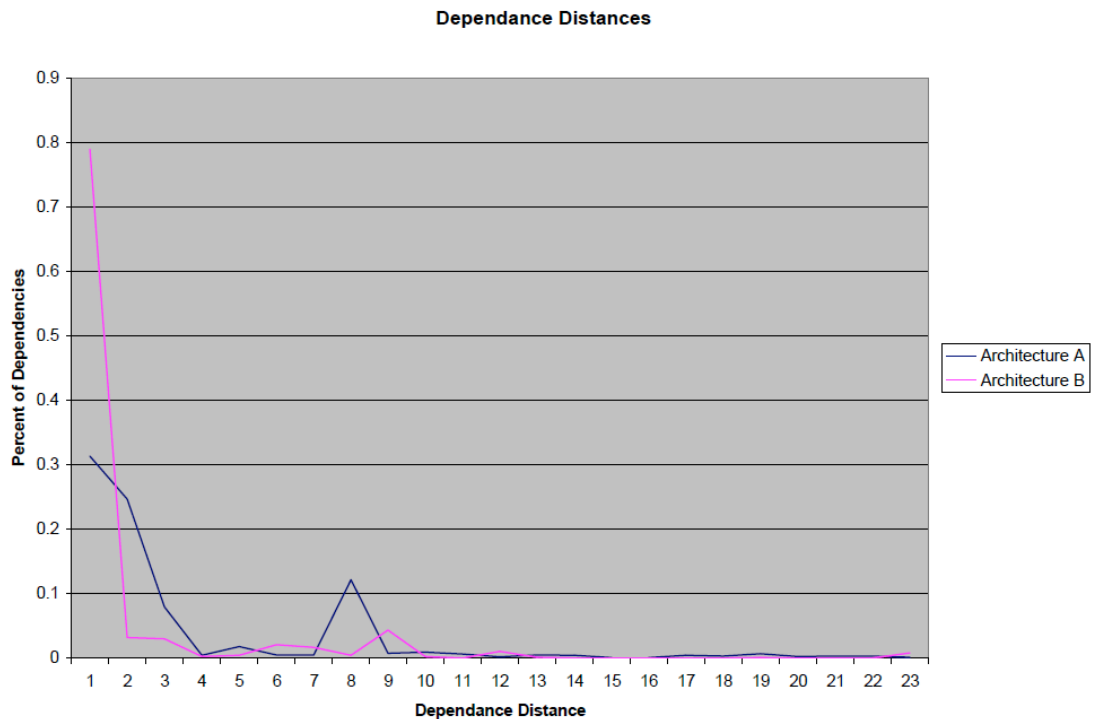
```
% ./part2test.pl
```

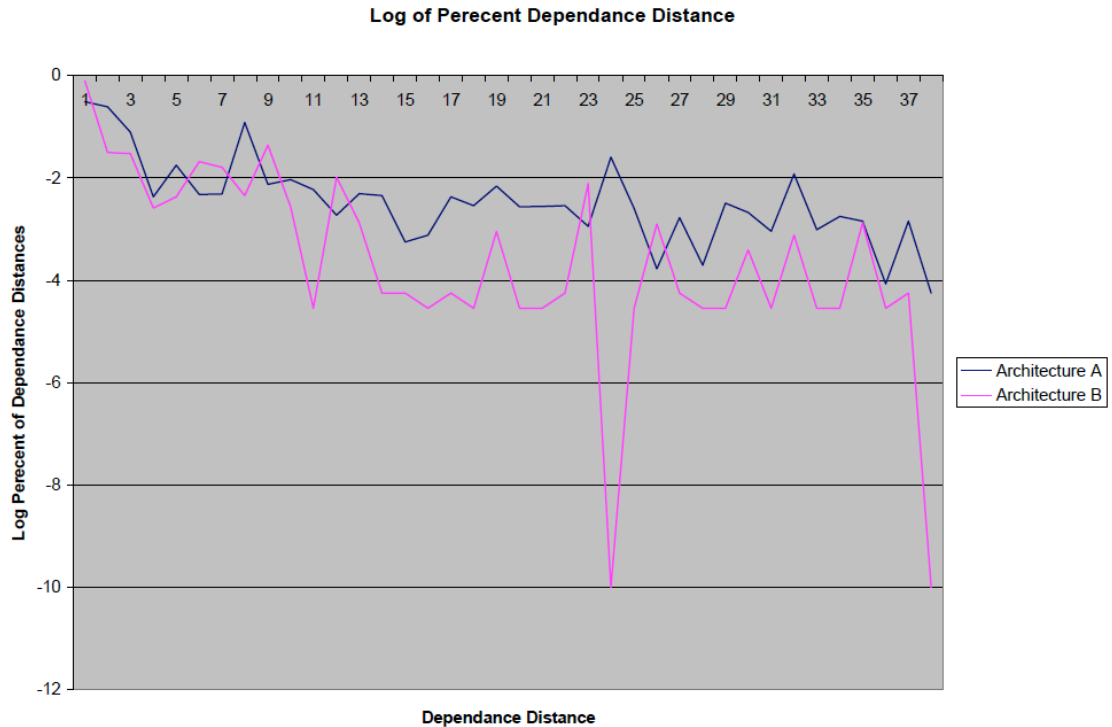
Although we will release a script that will test the lab Pin tools, these scripts will not verify your Pin tools, and we will not release the expected results of the test cases. Further, we reserve the right to run test cases not included in the released test script. You are encouraged to compare your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

Lab Questions:

Your response to the lab questions should be typed in lab0questions.pdf (or lab0questions.doc) in the lab0handout 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. Plot the dependency distance histogram (Number of dependences vs dependency distance) for the applications.
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.
3. 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 completed the lab to your satisfaction, submit your changes (not results) to the svn repository by running

```
% svn commit -m "Lab 0 Final Check-in"
```

The deadline for submission is 23:59:59 EST 17 February 2017. No late submissions will be accepted!

Advice on Mine Sweeping:

These labs are quite new. There will probably be bugs in either our code or infrastructure. You can help us give our lab infrastructure a shakedown. If you notice any 'interesting' or 'unexpected' behavior it could be a problem in the code or infrastructure that we have reported. Report these bugs immediately to the TA, preferably in an email with header *6.823 Bug Report*. This header 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