CS 6290: High-Performance Computer Architecture

Spring 2015

Project 0
Due: September 2nd 2015 at 5pm EST

This project is intended to help you set up and use the simulator for the other three projects, and also to help you understand the importance of branch prediction a little bit better. To complete this project, you will need to set up VirtualBox and the CS6290 Project virtual machine.

You will fill in the form fields in this PDF document and submit it in T-Square (the submitted file name should still be PRJ0.pdf). Note that you will need to save the filled-out form, and that many PDF readers/viewers cannot save the document. One free online viewer that you can use to do this is http://www.pdfescape.com/. If you are having trouble filling (or saving) this PDF form, fill out and submit the provided PRJ0.txt form. If you do use the PRJ0.txt file, please note that the answers you put into PRJ0.txt should be **exactly** the same as what you would enter in the PDF form, and that the format of the PRJ0.txt file must be preserved.

Additional files to upload are specified in each part of this document. Do **not** archive (zip, rar, or anything else) the files when you submit them – each file should be uploaded separately, with the file name specified in this assignment. You will lose up to 10 points for not following the file submission and naming guidelines. Furthermore, if it is not VERY clear which submitted file matcheswhich requested file, we will treat the submission as missing that file. The same is true if you submit multiple files that appear to match the same requested file (e.g. several files with the same name). In short, if there is any ambiguity about which submitted file(s) should be used for grading, the grading will be done as if those ambiguous files were not submitted at all.

As explained in the course rules, **this is an individual project**: **no collaboration with other students or anyone else is allowed**.

Part 1 [40 points]: Setting up the simulator

Install the Oracle VirtualBox VMM (Virtual Machine Manager) and the CS6290 Project VM (Virtual Machine). Once you start (boot) the CS6290 Project VM from within VirtualBox, you should see a Linux desktop. Click on the the topmost icon on the left-side ribbon and in the “Type Your Command” field type “Terminal”. Now you should get a terminal window – click on it and use it to issue commands we need.

The simulator in the “sesc” directory within our home directory, and we first need to build it. We will go into that directory and build the simulator:

```
cd ~/sesc
```

```
make
```

In the ~/sesc directory you should see (use 'ls') sesc.opt and sesc.debug links to executable files. We only compiled sesc.opt now, so only that link works, but that is enough for now because we will not be modifying the simulator in this project.

Now we need some application to run inside the simulator. The simulator simulates processors that use the MIPS ISA, so we need an application compiled for a MIPS-based Linux machine. Since these are hard to find, we have set up the Splash2 benchmark suite in the 'apps' subdirectory. For now, let's focus on the lu application:

```
cd apps/Splash2/lu
```

There is a Makefile for building the application from its source code, which is in the Source subdirectory:

```
make
```

Now you should see ('ls') a lu.mipseb executable in the application's directory. This is an executable for a big-endian MIPS processor – just the right thing for our simulator. So let's simulate the execution of this program!

Well, the simulator can simulate all sorts of processors– you can change its branch predictor, the number of instructions to execute per cycle, etc. So for each simulation we need to tell it exactly what kind of system to simulate. This is specified in the configuration file. SESC already comes with a variety of configuration files (in the ~/sesc/confs/ directory), and we will use the one described in the cmp4-noc.conf file. So we use the following command line (this is all one line):

```
~/sesc/sesc.opt   -c   ~/sesc/confs/cmp4-noc.conf   -olu.out
-elu.err lu.mipseb -n16 -p1
```

Note that the dashes in the above command line are "minus" signs – pdf conversion of this document may have resulted in changing these to longer dashes, in which case you should change them back to "minus" signs.

In this command line, the –c option tells the simulator which configuration file to use. The –o and –e options tell it to save the standard and error output of the application into lu.out and lu.err files, respectively. This is needed because the simulator itself prints stuff, and we want to be able to look at the output and possible errors from the application to tell it executed OK. Note that there should be a space between "-c" and the configuration file name, but there should be no space between the "-o" and its file name, and there should be no space between "-e" and its file name.

Then we told the simulator which (MIPS) executable to use, and what parameters to pass to that executable. The –n16 parameter tells the LU application (which does LU decomposition of a matrix) to work on a 16x16 matrix – it should finish very quickly because the matrix is so small. The –p1 parameter tells the application to use only one core – we will want to use only one core until Project 3!

Now you want to look at the output of the application – the lu.err should be empty and the lu.out should have the "Total time without initialization: 0" line at the end. Now we know that the application actually did all the work it was supposed to! Applications not really finishing is a very common pitfall in benchmarking – we are running the applications just to see how the hardware performs, so we tend to not look at the output. However, if a run got aborted half-way through because of a problem (e.g. mistyped parameter, running out of memory, etc.), the hardware might look really fast for that run because execution time will be much shorter! So we need to check for each run whether it actually completed!

After the run completes, we also get a simulation report file in the current directory. This report will have a name that consists of "sesc_", the name of the executable (lu.mipseb in our case), a dot, and a unique identifier (random string). The identifier makes the report file name unique, so you can do several simulations and get different reports for them. Let's say our report file name is "sesc_lu.mipseb.V7BFX9". This report file contains the command line we used, a copy of the hardware configuration we used, and then lots of information about the simulated execution we have just completed. You can read the report file yourself, but since the processor we are simulating is very sophisticated, there are many statistics about its various components in the report file. To get the most relevant statistics out of the report file, you can use the report script that comes with SESC:

```
~/sesc/scripts/report.pl -last
```

The "-last" option tells the script to use the latest (by file time) report in the current directory. Instead of "-last", you can specify the name of the report file you want to use, or use "-a" which makes the script process all reports in the current directory.

The printout of the report script gives us the command line that was used, then tells us how fast the simulation was going (Exe Speed), how much real time elapsed while the simulator was running (Exe Time), and how much simulated time on the simulated machine it took to run the benchmark (Sim Time). Note that we have simulated an execution that would take only about 260 microseconds on the simulated machine, but it probably took a second or two for the simulator to simulate that execution.

The printout of the report script also tells us how accurate the simulated processor's branch predictor was overall, and how its components fared (RAS, direction predictor, and BTB). Next, it tells us how many instructions were executed and the breakdown of these instructions. Finally, it tells us the overall IPC achieved by the simulated processor on this application, how many cycles the processor spent trying to execute instructions, and what those cycles were spent on. This last part is broken down into issuing useful instructions (Busy), waiting for a load queue entry (LDQ) to become free, then the same for store queue (STQ), reservation stations (IWin), ROB entries, and physical registers (Regs, the simulator uses a Pentium 4-like separate physical register file for register renaming). Some of the cycles are wasted on TLB misses (TLB), on issuing wrong-path instructions due to branch mispredictions (MisBr), and some other more obscure reasons (ports, maxBr, Br4Clk, other). Note that each of these "how the cycles were spent" items is a percentage although they do not have the percentage sign. Also note that the simulator models a multiple-issue processor, so the breakdown of where the cycles went is actually attributing partial cycles when needed. For example, in a four-issue processor we may have a cycle where one useful instruction is issued and then the processor could not issue another one because the LDW became full. In this cycle, 0.25 cycles would be counted as "Busy" and 0.75 cycles toward "LDQ".

Because simulation executes on one machine but is trying to model another, when reporting results there can be some confusion about which one we are referring to. To avoid this confusion, the simulation itself is called "native" execution and the simulated execution is called "target" execution.

To complete Part 1 of the homework, you should run another simulation, this time using 128 as the matrix size for lu (this simulation will take a few minutes even on a reasonably powerful laptop computer):

```
~/sesc/sesc.opt   -c   ~/sesc/confs/cmp4-noc.conf   -olu.out
-elu.err lu.mipseb -n128 -p1
```

Then fill in the blanks in the following statements:

A) In this simulation, the simulated processor executed ________ ("nInst" in the report) instructions, which corresponds to ________ milliseconds of simulated time ("Sim Time" in the report), but the simulator itself took ________ seconds of actual native execution time to do this simulation ("Exe Time" in the report).

B) The accuracy of the simulated branch direction predictor is ______ percent in this simulation. This is the number below BPred.

C) Branches represent ________ percent of all executed instruction. This is the BJ number that follows nInst.

D) On average, ________ instructions are executed between one branch misprediction and the next. You need to compute this using other numbers in the simulation report..

E) Branch mispredictions result in wasting ________ percent of the processor's performance potential. Why is this number so high, especially considering your answer for Part 1D?

Hello!
a
b

F) Submit the lu.out file and the simulator's report file (sesc_lu.mipseb.<something>, rename it to sesc_lu.mipseb.Part1) together with this filled-out form in T-Square.

Part 2 [60 points]: Compiling and simulating a small application

Now let's see how to compile our own code so we can run it within the simulator. For that purpose, create a small application `hello.c` (you can modify the one in the ~/sesc/tests/ directory) that will print a greeting with your first and last name. For example, the application for the instructor would be:

```
#include <stdio.h>

int main(int argc, char *argv[]){

  printf("Hello, my name is Milos Prvulovic\n");

  return 0;

}
```

Make sure that the tring you use in your printf is **exactly** like the one shown, except for your name. You should try to compile and run your code natively, to see if it's working:

```
gcc -o hello hello.c

./hello
```

This should result in printing out the "Hello, my name is Milos Prvulovic" message, with your own name of course. Once your code is working natively, you can try compiling it for the simulator. Remember that, because the simulator uses the MIPS ISA, we cannot use the native compiler. Instead, we use a **cross-compiler** – it is a compiler that runs on one machine (the native machine) but produces code for a different machine (the target machine, i.e. MIPS). Installing and setting up the cross-compiler is a time-consuming and tricky process, so we have already installed one and made it ready for you to use. It can be found in /mipsroot/cross-tools/bin with a name "mips-unknown-linux-gnu-gcc". In fact, there a number of other cross-compilation tools there, all of them with a prefix "mips-unknown-linux-gnu-" followed by the usual name of a GNU/GCC tool. Examples of these tools include gcc (a C compiler), g++ (a C++ compiler), gas (assembler), objdump (tool for displaying information about object files), addr2line (returns source file and line for a given instruction address), etc.

To compile our hello.c application, we can use the cross-compiler directly:

```
/mipsroot/cross-tools/bin/mips-unknown-linux-gnu-gcc -O2 -g
-static -mabi=32 -fno-delayed-branch -fno-optimize-sibling-
calls   -msplit-addresses   -march=mips4   -o   hello.mipseb
hello.c
```

We use some optimization (-O2), include debugging information in the executable file (-g), link with static libraries (-static) to avoid dynamic linking which requires additional setting up in the simulator, tell the compiler to avoid using branch delay slots (-fno-delayed-branch), regular call/return for all procedure calls (-fno-optimize-sibling-calls), use some optimizations that allow faster addressing (-msplit-addresses), and use a 32-bit implementation of the MIPS IV ISA (-mabi=32 –march=mips4).

After the compiler generates the MIPS executable file hello.mipseb for us, we can execute it in the simulator. Run this command line **exactly** as shown below – you should not change the command line options, run it from a different directory, or specify the path for hello.mipseb – that would change the simulated execution and prevent us from checking if you did it correctly and if everything is working.

```
~/sesc/sesc.opt  -c  ~/sesc/confs/cmp4-noc.conf  -ohello.out
hello.mipseb
```

To complete this part of the project, get the simulation report summary (using report.pl) and fill in the blanks in the following statements:

G) In this simulation, the simulated processor executed __________ ("nInst" in the report) instructions

H) Edit your hello.c to add an exclamation mark ("!") to the string that is printed: `printf("Hello, my name is Milos Prvulovic!\n");`
then cross-compile and simulate your hello.mipseb program again.
Now the simulated processor has executed _________ instructions. Why do you think the number of executed (simulated) instruction has changed this way? Note that *it is possible* for the number of instructions to go down even though we have made the string longer!

I) Why does it take so many instructions to execute even this small program? Hint 1: you can use mips-unknown-linux-gnu-objdump with a "-d" option to see the assembler listing of the hello.mipseb executable.
Hint 2: Not all of the code you see in the listing from Hint 1 is actually executed.

J) Why is the branch predictor so much less accurate here than it was on the lu benchmark?

K) Submit your hello.c source code from part H) (the one with the exclamation mark) and the two report files (sesc_hello.mipseb.<blah>, rename to sesc_hello.PartG and sesc_hello.PartH) in T-Square.