CS 6290: High-Performance Computer Architecture

Fall 2014

Project 0
Due: September 3rd 2014 at 5pm EST

This project is intended to help you set up the simulator for the other three projects. To complete this project, you will need to set up VirtualBox and the UD233Project virtual machine.

You will submit a report for this project in T-Square. Each part of this project assignment specifies what should be included in your report and which additional files to submit in T-Square. Do not archive the report files when you submit them – each file should be uploaded separately, with the file name specified in this assignment.

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

Part 1 [50 points]: Setting up the simulator

Install the Oracle VirtualBox VMM (Virtual Machine Manager) and the UD233Porject VM (Virtual Machine). Once you start (boot) the UD233Project 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 is already set up in the "sesc" directory within our home directory. We will go into that directory:

```
cd ~/sesc
```

The simulator is already compiled, so in this directory you should see (use the Linux command 'ls') sesc.opt and sesc.debug links to executable files. Now we need some applications 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:

```
~/sesc/sesc.opt -c ~/sesc/confs/cmp16-noc.conf -olu.out -elu.err
 lu.mipseb -n32 -p1
```

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 information, and we want to be able to look at the output and possible errors from the application to ensure it executed correctly.

Then we told the simulator which (MIPS) executable to use, and what parameters to pass to that executable. The –n32 parameter tells the LU application (which does LU decomposition of a matrix) to work on a 32x32 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 (Exe Time) for the simulation, and how much time on the simulated machine we simulated (Sim Time). Note that we have simulated an execution that would take only about 460 microseconds on the simulated machine, but it probably a second or two to do that simulation.

The printout of the report script also tells us how accurate the simulated processor's branch predictor was overall, and how its components did (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 instructions were executed, and how the processor's issue potential was spent – issuing instructions (Busy), waiting for load queue entry (LDQ), store queue entry (STQ), reservation station (IWin), ROB entry, physical registers (Regs, the simulator uses a Pentium 4-like separate physical register file for register renaming), TLB misses (TLB), issuing wrong-path instructions due to branch mispredictions (MisBr), and some other more obscure reasons (ports, maxBr, Br4Clk, other).

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 512 as the matrix size for lu (this simulation will take about half an hour or more even on a reasonably powerful laptop computer):

```
~/sesc/sesc.opt -c ~/sesc/confs/cmp16-noc.conf -olu.out -elu.err
 lu.mipseb –n512 -p1
```

Then answer the following in your report:

A) What is the accuracy of the simulated branch direction predictor (BPred) in this simulation?
B) What percentage of all executed instruction do branches represent? This is the BJ number that follows nInst. How many instructions are executed between consecutive branch mispredictions?
C) How much of the processor's performance potential was wasted because of branch mispredictions (MisBr)? Why is this number so high, especially if you take into account the predictor's accuracy.
D) How much target time was simulated ("Sim Time" in the report), how many instructions were executed in the simulated processor ("nInst" in the report), and how much real time was used for the simulation ("Exe Time" in the report)?
E) 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 report in T-Square.

Part 2 [50 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;

}
```

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    -O3    -g
-static -mabi=32 -fno-delayed-branch -fno-optimize-sibling-calls
-msplit-addresses -march=mips4 -o hello.mipseb hello.c
```

We use maximum optimization (-O3), 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:

```
~/sesc/sesc.opt -c ~/sesc/confs/cmp16-noc.conf -ohello.out hello.mipseb
```

To complete this part of the project, get the simulation report summary (using report.pl) and briefly answer the following questions:

F) How many instructions were executed on the target (simulated) machine?

G) 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.
How many instructions are now executed on the simulated machine?

H) 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.

I) Why is the branch predictor so much less accurate here than it was on crafty?

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