ECE4100/6100 Advanced Computer Architecture

Programming Assignment 1
Instruction Level Parallelism (ILP)

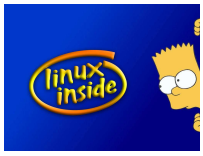
Assigned: Aug. 27, 2010 **Due: 12:35pm Sept 10, 2010**

(By handbook, the due day of distant learning students has an automatic one-week delay)

Score: _____ Date:_____ Name:_____

TA Signoff: _____

The purpose of this assignment is to help you understand, use, and modify an execution-driven processor simulator at the functional level. You will use Simplescalar, in particular, **sim-safe**, to carry out this project. The simulation platforms this toolset supports are all Unix-based, such as Linux, Cygwin on Windows, Solaris, etc. You have to download the `simplecalar3v0d.tgz` (gnu-zipped tar-ball) from Tsquare. The simulator is written in C with extensive use of Macro. (This project has to be done on Unix-like OS such as Linux, Solaris or Cygwin on Windows. You cannot do it with Microsoft Visual Studio.)



Part (I): (0%) Warm-up

This part is the first step to help you set up the toolset and run `sim-safe`. It will not account for any credit of this assignment but just for you to warm up. The second part of this assignment accounts for the entire 10% of your final grade. For further details, please read the install guide, users' guide and tutorial at http://www.simplescalar.com.

After you download `simplesim-3v0d.tgz`, untar the tar-ball into a directory called simplesim-3.0/. The first step is to compile the tool using its `Makefile`. The toolset provides several simulators including `sim-safe`, `sim-cache`, `sim-bpred`, `sim-outorder`, etc., we will only use `sim-safe` for this project. These simulators are capable of executing two types of binaries: PISA and Alpha. We will provide SPEC CPU2000 Alpha binaries (in Tsquare) for you to run your simulations. Also note that, we are not going to use PISA, therefore, you do *not* need to install the PISA `binutils` and cross-compiler instructed in the users_guide_v2.pdf file. To compile your processor simulator for Alpha binaries, do

```
% make config-alpha
```

This command will link the necessary files for building an Alpha processor Simplescalar simulator. Following that, do

```
% make clean
% make sim-safe
```

Depending on what platforms you use, you may experience several warnings or even errors during the compilation. You can largely ignore those warnings. However, if you encounter errors, in most of the cases, you need to fix some header files in the source codes based on the

complaints. For Solaris, you may need to change a syscall name in the `syscall.c` file. (Solaris contains some system calls which function the same but with slightly different names.)

Once you finish the make's, you should have a binary built called `sim-safe` (or `sim-safe.exe` if using Cygwin) in the same directory. This is the simulator that you will use to simulate given binaries. The toolkit directory provides several toy benchmark programs in the following directory: `tests-alpha/bin`. To run your first simulation, do

```
% ./sim-safe ./tests-alpha/bin/test-math
```

You should see the simulation output printed on the screen. The first few lines showed the copyright information followed by your command line and the standard simulator knobs you can use in `sim-safe`. (One useful knobs we will use in the second part is `-max:inst <# of max instructions>` Then the output of the program (i.e. `test-math`) is printed. This program prints out something like `pow(12.0, 2.0) == 144.00000` ,etc. At the last section, your simulation output statistics are printed. The interesting ones are `sim_num_insn` and `sim_num_refs`. They are printed with self-explanatory comments next to them. You can try to run other toy benchmarks or the SPEC benchmark programs we provide and experiment with the knob `–max:insn <# of max instructions>`.

The next step for warm-up is to learn how to execute the selected 11 SPEC2000 benchmark programs provided. I have written a Makefile called `SPEC2000.make` and `redir.sh` to make this task easier. These benchmark programs are compressed (gnu-tarballs) into 2 different files: `CPU2000int.tgz` and `CPU2000fp.tgz` for integer and floating-point programs, respectively. You need to download these three files and uncompress them into your working directory.

```
% tar xvfz CPU2000int.tgz
% tar xvfz CPU2000fp.tgz
```

Now you should have 2 new directories that contain the needed SPEC benchmark binaries and input files. You need to make minor changes for the first few lines in the `SPEC2000.make` file to run the simulations. Here are the quick tips. First of all, change the full directory aliased by `SIM_DIR` into the directory you place your simplesim-3.0. Then do the same for the directories of `CPU2000int` and `CPU2000fp`. Other aliases you need making changes to facilitate your working directories are `MASTER_RESULT_DIR` and `RESULT_DIR`. Note that you have to create these directories (e.g., `results` and `Naïve` from `SPEC2000.make`) by explicit `mkdir` commands. Without creating these directories a priori, when you run the simulation, it will stop and complain "No such file or directory". You can place these output files anywhere you want so long as you point to the locations in your Makefile.

After you have done all the modifications to point to the correct locations, you can start simulating these benchmark programs. You can run them all by simply typing:

```
% make –f SPEC2000.make all
```

In which, `-f` is the knob to specify your Makefile, i.e., `SPEC2000.make` in our case. The "`all`" option is coded inside our Makefile if you read the Makefile. You can also execute benchmark by groups. For example, the following command will simulate only the 6 integer benchmark programs provided.

```
% make –f SPEC2000.make int-all
```

Or you can also simulate one benchmark at a time by typing the name of the benchmark specified in the Makefile, for example, the following will only simulate `mcf`.

```
% make –f SPEC2000.make mcf
```

The proper way to debug your program is to use debugging tool such as `gdb`, `ddd`, or using the framework of `eclipse`. My favorite is to run `gdb` within `emacs` for debugging (I am an old-timer.) In this case, you may want to debug your program first using the toy benchmark or running the SPEC directly under UNIX prompt. You can examine the Makefile to obtain the information such as input files and file commands. For example, to execute gcc directly, you can type

```
% sim-safe $(Your own CPU2000int Directory)/176.gcc/exe/cc1_peak.ev6 <
$(Your own CPU2000int Directory)/176.gcc/data/ref/input/166.i
```

After the simulations, your results directory (e.g., `results/Naïve`) will generate 2 files for each benchmark application: `stderr` and `stdout`. The `stdout` file contains the output of the executed binary. For example, `gcc` will place their assembly output to this file (if you simulate long enough to the point `gcc` starts to spit out these lines.) The `stderr` is the file we are interested which shows the output results from the simulator `sim-safe` in this case. This file will contain the same information you see in the last section on screen when you run `sim-safe` directly.

Part (II): (100%) Instruction-Level Parallelism

Problem Statement

In this part, you need to modify the functional simulator, primarily `sim-safe.c` and potentially other relevant `*.[c, h]` files depending on your personal coding style, to perform an Instruction-Level Parallelism (ILP) limit study. For the results, you have to report the ILP (in IPC) using the three different machine models shown below. (Note that the assumptions of the ILP limit in each model are progressive.) Since we are studying ILP, all instructions will be assumed to have unit-latency, i.e., one cycle latency for all instructions. ILP is equal to IPC under this assumption. Also assumed is that you have perfect knowledge of all branch directions for the Machine Model 2 (No CD) and Machine Model 3 (No CD + RR). In this way, you view all the basic blocks of the entire program as one single gigantic scheduling window. Here are the three machine models for generating your ILP results in your code.

1. Machine Model 1 (Naïve): Track all data dependency caused by **architectural registers** and **memory addresses**. Restart the ILP calculation whenever a branch instruction is encountered. In other words, you exploit ILP within each basic block.

2. Machine Model 2 (No CD): Calculate ILP by taking all data dependency caused by **architectural registers** and **memory addresses** into account. But assume a perfect branch predictor is in place, i.e. viewing the entire program as a giant scheduling window. (No CD: No Control Dependency.)

3. Machine Model 3 (No CD + RR): Perfect register renaming with perfect prediction. Namely, calculate ILP by eliminating all false dependency caused by recycling of the limited **registers** in Alpha ISA.

Simulation you need to carry out.
Run your simulation for 1 billion instructions for each machine model using the SPEC benchmark programs provided. There are 6 integer benchmark programs and 5 floating point benchmark programs in the 2 gnu-zipped files. In your report for check-off, You have to plot the ILP for each model for each benchmark program.

Important Tips.

- When you want to print out the statistics you collected at the end of your simulation, you *must* place your printing routing inside the function called "`sim_aux_states(FILE *stream)`" which will be called after the simulation is terminated.

- All the instructions and their corresponding operations were defined in the `machine.def` file with extensive use of Macro. You do not nee to touch this file at all. All you need to modify should be within `sim-safe.c`. This is just my suggestion, you may have your own coding style though.

- To keep track of memory dependency, you need a good data structure. This part requires careful thought, otherwise, your system memory will blow up quickly and generate segmentation fault in your simulation before you finish 1 billion instructions.

- To understand how to account renaming efficiently in your coding, please read the papers suggested for ILP study on our course website. In particular, the 2 papers by Matthew Postiff *et al.* and Hsien-Hsin Lee, *et al.* They described the theoretical background with regard to how to track the ILP with and without register renaming and memory renaming. **You do not need the memory renaming part for this assignment.**

How to receive credit? You need upload your code as well as a report of your finding onto T-Square by due day. I do not accept late turn in, and there is no excuse. Check off by TA is needed. (We will give details later.) You need to prepare a summary spreadsheet of your simulated results and show the TA your implementation

Honor Code. This assignment must be done **individually**. For those who violate the rule, both the originator and the copier will receive zero credit and will be **immediately** reported to the Dean of Students' Affair for further action.