Lab Assignment 1: Performance Measurements

1 Objective

The purpose of this assignment is for students to learn how simulation is used in the architectural community to measure processor performance and evaluate the benefits of different architectural features. Specifically, you will be using a functional simulator to perform measurements to quantify the impact of different pipeline hazards.

All work on this lab assignment is to be done in groups of two.

2 Pre-Assignment

Before attempting this assignment, students should complete the pre-assignment. As part of the pre-assignment, you should have already read the document titled *The SimpleScalar Tool Set, Version 2.0* which is available on Blackboard. You may also find other resources by consulting `http://www.simplescalar.com`.

3 Problem Statement

In this assignment, you will investigate the performance of a pipelined processor implementing the SimpleScalar PISA instruction set, which is similar to MIPS. Assume the five-stage pipeline implementation as described in class. The stages of the pipeline are Fetch (F), Decode (D), Execute (EX), Memory (M) and Writeback (WB). Assume that the Writeback stage writes to the register file (RF) in the first half of a cycle, while the Decode stage reads from the register file in the second half of a cycle.

Students should answer the following two questions using the `EIO trace` of the cc1 benchmark: `/cad2/ece552f/benchmarks/gcc.eio`. This EIO trace file runs the compiler portion of gcc (i.e., excluding the preprocessor, assembler, linker, etc.). In answering each of these questions, start with an ideal CPI of 1.0, and report **% drop (slowdown)** in performance due to stalls.

1. What is the slowdown in performance due to read-after-write (RAW) hazards considering our 5-stage pipeline with *NO* forwarding or bypassing? Assume an ideal memory system and ignore control hazards.
2. Now, consider a 6-stage pipeline where Execute is broken into 2 stages, EX1 and EX2, making the full pipeline: F, D, EX1, EX2, M, WB. This design has full forwarding and bypassing. Any inputs needed by Execute must be available at the start of EX1. Both stages (EX1 and EX2) are needed to compute the final result. What is the slowdown in performance due to read-after-write (RAW) hazards for this design? Assume an ideal memory system and ignore control hazards.

For the first question, create a short microbenchmark of 10-20 lines of C code to verify your statistics. More details about microbenchmarks are given in Section 5.

4 Methodology

Use the simulator source code from the pre-assignment (**/cad2/ece552f/simplesim-3.0d-ece552f-assign1.tgz**). Create a modified version of the `sim-safe` simulator to measure the required statistics for each of the problems in Section 3. Use the collected statistics to derive a numerical answer to these questions.

We have already defined two statistics counters for you. These can be found in `sim-safe.c` just after the `#include` statements and are listed below.

```
/* ECE552 Assignment 1 - STATS COUNTERS - BEGIN */

static counter_t sim_num_RAW_hazard_q1;
static counter_t sim_num_RAW_hazard_q2;

/* ECE552 Assignment 1 - STATS COUNTERS - END */
```

We have already registered these counters for you in `sim_reg_stats` (see Section "Modifying sim-safe" from the pre-assignment for more details). You **MUST** create the correct formulas (see `/* ECE552 - MUST ADD YOUR FORMULA */`) to calculate the CPI for the above questions. These will be used by the automarker to check your work. You are free to add additional statistics and formulae but do not change the names of the given statistics and formulas.

Hint: The registered counters should count hazards, i.e., pipeline dependences that result in stall(s). You cannot count more than one hazard per instruction. You might find it convenient to have separate hazard counters for different stall durations (e.g., for one cycle stall versus for two cycles stall).

Think about the two problem statements and how to measure the required statistics. Use Lab 0's "load-to-use hazards" walk-through example as a point of reference for your code modifications. Remember that you can continue to rely on O1, O2, I1, I2 and I3 to identify dependences; i.e., ignore the second corner case from the pre-assignment.

Always identify your code modifications to `sim-safe.c` with the comments `/* ECE552 Assignment 1 - BEGIN CODE */` and `/* ECE552 Assignment 1 - END CODE */`. Any code outside these indicators will NOT be considered during marking.

5 Microbenchmarking

It is quite easy to make a logical or implementation error when modifying `sim-safe` (or any simulator) that causes the gathered statistics to be erroneous. It is critical we verify that we are actually measuring the correct statistics. The best way to do this is to create a short C program of 10-20 lines, i.e., a **microbenchmark**, where we can predict the outcome of the simulation. You are required to create a microbenchmark for problem statement Q1 in Section 3 that verifies the gathered statistics. You are strongly encouraged to do this for all questions.

Recall that you can compile your microbenchmark with the PISA-specific `gcc` compiler located at `/cad2/ece552f/compiler/bin/ssbig-na-sstrix-gcc`. To create an executable called `mbq1` from the `C` file `mbq1.c`, compile with the command:

```
/cad2/ece552f/compiler/bin/ssbig-na-sstrix-gcc mbq1.c -O2 -o mbq1
```

The assembly code contains the instructions that you are passing to `sim-safe`. You may want to use an optimization flag other than `-O2`, such as `-O1` or `-O0`, depending on how your microbenchmark operates. Include the flags that were used to compile your microbenchmark in the report.

To create the intermediate assembly code in the file mbq1.s execute the command:

```
/cad2/ece552f/compiler/bin/ssbig-na-sstrix-gcc mbq1.c -O2 -S
```

You can disassemble the code in mbq1 using the following command:

```
/cad2/ece552f/compiler/bin/ssbig-na-sstrix-objdump -x -d -l mbq1
```

The file generated by objdump can be quite long, so search for `main()` to focus on the code you wrote. Also, you can always run sim-safe with the -v option to see all the assembly instructions that are executed. These should match the ones you see with objdump.

5.1 Good Practices

Crafting a good microbenchmark requires quite a bit of effort and experience, and it is an invaluable tool to help you validate the correctness of your code. This subsection provides some good practices and useful advice for your first microbenchmark.

1. Each microbenchmark should contain a loop that is executed a large number of times so that initialization effects can be discounted. For example, you cannot reach any conclusions for the number of RAW hazards, if your loop only executes ten times. Your measurements will be outnumbered by RAW hazards outside your loop (e.g., in data initialization, library calls, etc).

2. Your code within this large loop should reflect the statistics you are trying to measure. For example, if you are interested in RAW dependences, a code snippet of the form

   ```
   a = b
   k = 3
   ```

   is useless.

3. Always verify that the generated assembly code reflects the dependences you expressed in C. Sometimes the compiler may optimize away instructions. Using a lower optimization level during compilation is one solution. Another alternative is to print out the contents of all used variables, outside the loop, to ensure they are not optimized away. However be aware that calling a library function such as printf will make your `main()` code lengthier.

4. Your microbenchmark should be written in such a way that your measurements undoubtedly prove you are collecting the right statistic. For example, having 1,000,000 control flow instructions, 500,000 of which are taken, and saying that your program reports the correct number of taken control flow instructions is ambiguous. Your program may be counting the not-taken control flow instructions.

5. Your microbenchmark should provide comprehensive coverage of all scenarios, if possible. For example, if you come across hazards with different stall durations (e.g., 1-cycle stall versus 2-cycles stalls), you need to include **both** these scenarios in your microbenchmark. In addition, it is preferable to have different percentages of the different stall types, because otherwise you cannot prove you are counting them correctly.

6. You should be able to provide a short but concise explanation about how your microbenchmark verifies the correctness of your code. For example, assume you have a for-loop that executes N times and contains x 2-cycles stalls and y 1-cycle stalls. The hazard_counter_1cycle should be N*y, while the hazard_counter_2cycles should be N*x. These will be approximate values because there will be other RAW hazards outside your loop (e.g., in library calls). N should be large enough (e.g., millions) so all these initialization effects are just noise.

7. Alternatively, you could run your microbenchmark for two different values of N (e.g., N1 and N2). Since the hazards outside your loop will not change, you can account exactly for the hazards within the loop.

8. Be ready to demonstrate your microbenchmark's coverage. For example, for RAW hazards provide annotated snippets of assembly code (from objdump preferably rather than the intermediate assembly *.s) showing RAW dependencies between instructions. Compare your expected RAW hazard count with the SimpleScalar statistics after executing your microbenchmark. Be explicit. For example, saying that the number of hazards increases with the number of loop iterations is far from sufficient.

5.2 Debugging

If your microbenchmark's measurements differ from what you expect, then it is time to start debugging. First of all, revisit the aforementioned "good practices" to ensure the generated assembly is correct, and that your loop iteration count is large enough. Then, run your benchmark with `-max:inst` set to a small number of instructions (e.g., 10 or so) with the `-v` flag set. The SimpleScalar verbose mode prints out all the executed instructions one-by-one. Look at the instructions with their source and destination operands, and manually recompute the statistics of interest. If your computations differ from the generated statistics, then you have just identified the offending instruction(s). Slowly increase the number of executed instructions until a mismatch is found. Finally, you can always isolate the measurements to your for-loop. All you have to do is identify the PCs of interest from the objdump, and temporarily count hazards only for these PCs. Please note that this requires recompiling sim-safe after any modification to the microbenchmark.

6 Prelab

The prelab is worth 1/6 of the overall lab mark. Please complete the following steps before coming to the lab:

- Complete the pre-assignment, with a special focus in the walk-through example. You are encouraged to do this individually.
- Answer the following questions **(in writing!)**:
  1. What is the difference between a dependence and a hazard?
  2. What is the difference between a data hazard and a structural hazard?
  3. Are WAW hazards possible in a simple 5-stage pipeline? Explain your answer.
  4. What is the significance of the phrase "Assume that the Writeback stage writes to the register file (RF) in the first half of a cycle, while the Decode stage reads from the register file in the second half of a cycle" from Section 3? Give an example of how the register file access order can affect the number of RAW stall cycles.

5. Provide a high-level algorithmic solution for each question of Section 3, along with pipeline diagrams. Think of a series of instructions (e.g., i1 to ix), and specify the necessary conditions for a hazard to occur (e.g., data dependence, distance of instructions in the pipeline). Provide examples for all possible stall scenarios; for example, the conditions will be different for a one-cycle stall and a two-cycles stall. We provide empty pipeline diagrams in the appendix. Feel free to print multiple copies of that page and fill-in the tables as needed. This is an important step. Not only will it help you brainstorm, but it will also help the TA identify any issues with your logic.

- Write most of your code, along with some microbenchmarks, before coming to the lab.
- Finally, be prepared to answer any high-level questions about the simulator, your code, and the lab material (e.g., dependences, hazards, stalls).

7 Lab Deliverables

At the end of this assigment you should submit the following three files using the `submitece552f` (`/local/bin/`) command:

1. **sim-safe.c**: the modified simulator file. Identify all modifications to `sim-safe.c` with the comments `/* ECE552 Assignment 1 - BEGIN CODE */` and `/* ECE552 Assignment 1 - END CODE */`. Any code outside these indicators will NOT be considered during marking.
2. **mbq1.c**: your microbenchmark code for Question 1. Your microbenchmark must include comments and snippets of generated (or inline) assembly code that explain how it verifies the collected statistics. Follow the guidelines from Section 5. Failure to include appropriate explanation will result in a grade of 0 for the microbenchmark portion of the assignment.
3. **report.pdf**: a brief one-page pdf report (single-spaced with 12-point font size). Make sure your report can be viewed on the ug machines through `xpdf` or `acroread`.
   - State the % performance drop versus an ideal pipeline with CPI of 1.0 for the two questions. Briefly describe the mathematical derivations used to arrive at these answers.
   - Briefly explain how your microbenchmark collected statistics validate the correctness of your code for the first problem statement. Feel free to refer to comments within the mbq1.c file, as needed. Specify which compilation flags you used.

The submit command should be similar to the following:

```
submitece552f 1 sim-safe.c report.pdf mbq1.c
```

Please adhere to the provided naming conventions; misnamed files will not be marked. You can view the files that you have submitted via the command `submitece552f -l 1`. Finally, while developing, remember not to leave the Unix permissions to your code open.

8 Due Date and Marking Scheme

This assignment is due on **Monday October 9, 2017 at 5:00pm**. It is worth **6%** of the total course mark. The pre-lab will constitute 1/6 of the overall lab mark. An automarker will be used to compile and run your implementation and verify the statistics generated. Therefore, your implementation is required to follow some strict rules.

To begin with, use the simulator package specifically provided for this assignment as described in Section 4. Things to watch for:

- run the benchmarks as specified in the handout
- properly initialize all data structures
- do not change the existing stats; you can add more stats if you are interested in studying something in particular, but do not modify existing ones
- make sure to use all provided stats counters to collect the stats for RAW hazards (Q1, Q2); the counters can be found in the region marked with the following comments:

  ```
  /* ECE552 Assignment 1 - STATS COUNTERS - BEGIN */

  ... stats counters to be used are here ...

  /* ECE552 Assignment 1 - STATS COUNTERS - END */.
  ```

- the only file to modify is `sim-safe.c`
- do not add additional files, they will not be taken into account; your simulator should compile with the provided `Makefile` by typing `make sim-safe`
- CAUTION! When you are redirecting the output of the simulated program, use the `-redir:prog` flag of `sim-safe`. Do NOT redirect the output using the pipe character `'|'` or the redirect character `'>'` as this causes variation in the simulated instruction count. To redirect the output of the simulator, use the `-redir:sim` flag of `sim-safe`.
- an automatic comparison of your submissions will be done to check for copied code; changing variable names and formatting will not defeat the checker.

9 Questions

Please plan ahead and start early! We recommend students start working on the assignments as soon as possible. This will allow you time to learn `sim-safe.c` at a steady pace. Also by starting early, you will have an opportunity to ask better questions during the tutorial sessions.

Please post clarification questions on Piazza. DO NOT post code snippets; this constitutes cheating and you will be penalized. For a code-specific question send a private message to a TA.

A Appendix - Empty Pipeline Diagrams

We provide you with empty pipeline diagrams for your convenience. Print multiple copies of the following page and fill-in the tables as needed. A few pieces of advice:

- Enumerate the instructions as i1, i2 etc. Fill in each cycle with the appropriate stage (e.g., F, D) or the stall type.

- Draw the dependence and any forwarding/bypassing (add arrows between the appropriate pipeline stages). Briefly explain the number of stall cycles in the comments.
- Add a comment if this hazard is only possible for a specific instruction type (e.g., load).

	Cycles															
Instr.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	**Comments**

Table 1: Pipeline Diagram 1

	Cycles															
Instr.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	**Comments**

Table 2: Pipeline Diagram 2

	Cycles															
Instr.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	**Comments**

Table 3: Pipeline Diagram 3

	Cycles															
Instr.	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	**Comments**

Table 4: Pipeline Diagram 4