MIPS Simulator: Pipelining I
Project 3 – CS 3339 – Spring 2019

Due Date per TRACS. Friday before 11:55pm, late until Saturday noon -10pts, after that no submissions accepted.

PROBLEM STATEMENT

Cycle-accurate simulators are often used to study the impact of microarchitectural enhancements before they are designed in hardware. In this project, you will turn your emulator from Project 2 into a cycle-accurate simulator of a pipelined MIPS CPU.

Most cycle-accurate simulators are composed of two parts: a ***functional model***, which emulates the functional behavior of each instruction (e.g., your Project 2 emulator), and a ***timing model***, which simulates enough of the cycle-level behavior of the hardware to allow it to determine the runtime of an instruction sequence. The simulator may also track and report other events of interest, such as the number of branches taken vs. not-taken. These counts are called ***performance counters***.

I have provided a class skeleton for a Stats class intended to be instantiated inside your existing CPU class. The Stats class has functions (some of which are defined already, some of which are blank and need to be filled in) that allow it to collect statistics from the CPU during the simulation of a program, and to model the timing behavior of an 8-stage MIPS pipeline similar to the one we have studied in class.

The pipeline has the following stages: **IF1**, **IF2**, **ID**, **EXE1**, **EXE2**, **MEM1**, **MEM2**, **WB**.

Branches are resolved in the ID stage. There is no branch delay (in other words, the instruction words immediately following a taken branch in memory should *not* be executed). There is also no load delay slot (an instruction that reads a register written by an immediately-preceding load should receive the loaded value). Just like the 5-stage MIPS pipeline, there are no structural hazards, and data is written to the register file in WB in the first half of the clock cycle and can be read in ID in the second half of that same clock cycle.

For now, assume there are no forwarding paths: all sources must be read in ID, and a value must always be written back to the register file before it can be used as a source. The pipeline must inject the appropriate number of bubbles to ensure this for all read-after-write (RAW) data hazards.

Note that `trap 0x01` reads register Rs and `trap 0x05` writes register Rt. Note that `mfhi` and `mflo` read the hi/lo registers, and `mult` and `div` write them.

Note that the *$zero* register cannot be written and is therefore always immediately available.

Your simulator will report the following statistics at the end of the program:

- The exact **number of clock cycles** it would take to execute the program on a CPU with the hardware parameters described above. (Remember that there's a 7-cycle startup penalty before the first instruction is complete)
- The **CPI** (cycle count / instruction count)
- The **number of bubble cycles** injected due to data dependencies
- The **number of flush cycles** in the shadows of jumps and taken branches
- The percentage of instructions that are **memory accesses** (memory-op count / instr count)
- The percentage of instructions that are **branches** (branch count / instruction count)
- The percentage of **branch instructions that are taken** (taken branches / total branches)

Note that the last two bullet points above refer to ***conditional branch instructions only*** (i.e., `beq` and `bne` instructions), not jump instructions.

You are provided the following new files:

- A `Stats.h` class specification file, which you should read carefully to understand the functions which should be implemented
- A `Stats.cpp` class implementation file, which you should enhance with code to track the pipeline state in order to identify data dependencies and to model bubbles and flushes
- A new Makefile

In addition to enhancing the `Stats.cpp/.h` skeleton, you will need to modify your existing `CPU.cpp` in order to call the necessary Stats class functions. Don't forget to `#include "Stats.h"` in the CPU class header file and instantiate a Stats class object in CPU.

When modifying your existing `CPU.cpp` please follow this format example (you will have to change it as appropriate for each instruction). If you keep the calls to `Stats` on the same line as the register use it will be much easier to find mistakes. Additionally, by keeping the order rd, rs, rt it will help me debug your code if needed.

```
writeDest = true; destReg = rd; stats.registerDest(rd);
aluOp = ADD;
aluSrc1 = regFile[rs]; stats.registerSrc(rs);
aluSrc2 = regFile[rt]; stats.registerSrc(rt);
```

You'll also need to change `CPU::printFinalStats()` to match the expected output format (see below).

ASSIGNMENT SPECIFICS

This project is to be submitted individually, and you should be able to explain all code that you submit. You are encouraged to discuss, plan, design, and debug with fellow students. Unlike the prior assignments you do not have a complete set of outputs to compare, however you can check your results with the information provide in the project3_expected.txt file which is included in the tarball. You are encouraged to check your results with other students as well, but never share your source code.

You must have a working copy of Project 2 to start this project. Begin by copying all of your Project 2 files into a new Project 3 directory, e.g.:

```
$ cp -r cs3339_project2/ cs3339_project3/
```

Download and extract the additional files in provided cs3339_project3.tgz and add them to your project3 directory. You can compile and run the simulator program identically to Project 2, and test it using the same `*.mips` inputs.

The following approach is recommended for your Stats class.

Notify the Stats object whenever an instruction in the ID stage uses a register as a destination, and whenever an instruction in ID uses a register as a source. I've provided function definitions for this (`registerSrc/Dest`). When a register is used as a source, the Stats class should look for instructions in later pipeline stages (older instructions) that will write that register. This will allow Stats to identify data dependencies (RAW hazards). A bubble injects a NOP into the pipeline (an operation that doesn't write any register). You'll also need to keep track of flushes due to jumps and taken branches. A flush overwrites an existing pipeline entry with a NOP.

You can check your timing results using the equation instrs = cycles – 7 – bubbles – flushes.

The following is the expected result for sssp.mips which is included as sssp.out in the tarball. Your output must match this format verbatim:

```
CS 3339 MIPS Simulator
Running: sssp.mips

 7 1

Program finished at pc = 0x400440  (449513 instructions executed)

Cycles: 1627234
CPI: 3.62

Bubbles: 1125724
Flushes: 51990

Mem ops: 43.9% of instructions
Branches: 9.6% of instructions
  % Taken: 60.5
```

Note that you can print the ratio a/b with 1 place after the decimal and a percentage sign using the following C++ code:

```
cout << fixed << setprecision(1) << 100.0 * a / b << "%" << endl;
```

Additional Requirements:

- **Your code must compile with the given `Makefile` and run on zeus.cs.txstate.edu**
- Your code must be well-commented, sufficient to prove you understand its operation
- Make sure your code doesn't produce unwanted output such as debugging messages. (You can accomplish this by using the `D(x)` macro defined in `Debug.h`)
- Make sure your code's runtime is not excessive
- Make sure your code is correctly indented and uses a consistent coding style (and don't use any TAB characters!)
- Clean up your code before submitting: i.e., make sure there are no unused variables, unreachable code, etc.

SUBMISSION INSTRUCTIONS

Verify that you have included your name in the header of the CPU.cpp and Stats.cpp files (and Stats.h if you have modified it). Submit all of the code necessary to compile your simulator (all of the `.cpp` and `.h` files not just the ones you have modified, and the `Makefile`) as a compressed tarball with your netID at the beginning of the filename. You can do this using the following Linux command:

```
$ tar czvf <your NetID>_project3.tgz *.cpp *.h Makefile
```

Do not submit the executables (`*.mips` files). Any special instructions or comments to the grader, including notes about features you know do not work, should be included in a separate text file (not inside the tarball) named `<your_netID>_README.txt`. The submission on TRACS should contain one or two files:

```
lbh31_project3.tgz  (using your netID)
lbh31_README.txt  (optional but must be a text file)
```

Failure to follow these instructions will result in point deductions including a zero score if the submission does not run on zeus. To verify you have packaged everything correctly I have also provided a bash script called submit_test. Download this script and follow the directions to confirm your submission will un-tar and compile properly on zeus. Common mistakes include forgetting the Makefile or using .zip format.

All project files are to be submitted using TRACS. Please follow the submission instructions here:
https://tracsfacts.its.txstate.edu/user-guides/tutorials-students/coretools/assignments.html
Note that files are only submitted if TRACS indicates a successful submission. Be sure you receive the verification email to confirm.

You may submit your file(s) as many times as you'd like before the deadline. Only the last submission will be graded. ***TRACS will not allow submission after the deadline***, so I strongly recommend that you don't come down to the final seconds of the assignment window. Late assignments will be assessed a 10 point penalty until noon following the due date. After that submissions will not be accepted.