MIPS Simulator: Pipelining with Forwarding Paths
Project 3 – CS 3339 – Fall 2017

Due: Tuesday October 24, 2017 @ 11:55pm (late until 10/25 at noon -10pts)
40 points

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.

Assume full data forwarding is possible and therefore bubbles are only required for data hazards that cannot be resolved by the addition of a forwarding path. In the case of such a data hazard, the processor stalls the instruction with the read-after-write (RAW) hazard in the ID stage by inserting bubbles for the minimum number of cycles until a forwarding path can get source data to the instruction.

To do this, ID must track the destination registers and cycles-until-available information for instructions later in the pipeline, so that it can detect hazards and insert the correct number of bubbles. (ID also uses that information to create forwarding logic control signals that are then flopped down the pipeline with the consuming instruction, though you do not have to model that part for this project). This is called ***static scoreboarding***, and it's the technique used by one of the processors we've studied in class, the ARM Cortex-A8.

All instruction inputs are needed at the beginning of the respective stage. Most instructions need their inputs in the EXE1 stage, except for jr, beq, and bne, which need their inputs in ID, and the sw instruction's store data, which is needed in MEM1 (the base register is still needed in EXE1).

Instruction results become available for forwarding at the beginning of the stage *after* they are produced (e.g., the ALU produces data at the end of EXE2, but that data is not forwardable until the beginning of MEM1, with the data forwarded from the EXE2/MEM1 flop). All instruction results are produced at the end of the EXE2 stage, except for lw, mult, and div, which produce results at the end of MEM2, and jal, whose result becomes available at the end of ID.

For simplicity, assume that trap instructions follow the same timing as add instructions. 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)
- The **total number of RAW (read-after-write) hazards** detected, including those that result in bubbles and those that can be immediately addressed by forwarding. Note that an op in ID cannot have a RAW hazard with an op in WB.
- The **ratio of instructions to RAW hazards**
- The **number and percentage of RAW hazards** identified on the instruction in each of the stages between ID and WB

Note that the branches and percentage of branch instructions that are taken above refer to *conditional branch instructions only* (i.e., beq and bne instructions), not jump instructions.

I have provided the following new files:

- A Stats.h class specification file, which you should modify to add any functions or function parameters that you need
- 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. You'll also need to change CPU::printFinalStats() to match my 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.

All provided files are on TRACS: Stats.cpp, Stats.h, a new Makefile, and project3_expected.txt with partial results.

Begin by copying all of your Project 2 files into a new Project 3 directory, e.g.:

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

Then add the provided files to your project3 directory. You can compile and run the simulator program identically to Project 2, and test it using the same *.mips inputs.

You can implement your Stats class any way you'd like, including adding variables, functions, or function parameters. I recommend the following approach:

Notify the Stats object whenever an instruction in the ID stage uses a register as a destination and specify the earliest pipeline stage in which the result will become available. Also notify stats whenever an instruction in ID uses a register as a source and specify the earliest pipeline stage in which the source will be needed. I've provided function definitions for this (registerSrc/Dest).

Whenever a register is used as a source, the Stats class should look for instructions in later pipeline stages (older instructions) that will write result data to that register in a future cycle, i.e. RAW hazards. The Stats class should then determine, based on what stage the source is needed and when the matching destination will be produced, whether the data dependency can be handled by a forwarding path without a bubble, or whether one or more bubbles must be injected.

Note that it's possible for multiple instructions later in the pipeline to all be writing the same destination register, and that if the instruction in ID reads that register, the hazard exists only on the most recent (youngest) producing instruction.

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. 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: 983220
CPI: 2.19
```

```
Bubbles: 481710
Flushes: 51990

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

RAW hazards: 349295 (1 per every 1.29 instructions)
  On EXE1 op: 225878 (65%)
  On EXE2 op: 79203 (23%)
  On MEM1 op: 37697 (11%)
  On MEM2 op: 6517 (2%)
```

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

Submit all of the code necessary to compile your simulator (all of the .cpp and .h files as well as the Makefile) as a compressed tarball. You can do this using the following Linux command:

```
$ tar czvf 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 README.txt.

All project files are to be submitted using TRACS. Please follow the submission instructions here:
http://tracsfacts.its.txstate.edu/trainingvideos/submitassignment/submitassignment.htm
Note that files are only submitted if TRACS indicates a successful submission.

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 not be accepted.