MIPS Simulator: Pipelining II – Forwarding Paths

Project 4 – CS 3339

Due Date per Canvas. Friday before 11:59pm, 24-hour late period -10%. After that no submissions accepted.

PROBLEM STATEMENT

In this project, you will enhance your simulator from Project 3 to model a pipeline with full forwarding paths. There is no additional code provided for this assignment. You should modify the CPU and Stats classes from your Project 3 submission.

Exactly as in Project 3, 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). 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 (the register file is "double pumped").

This time 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 is the technique used by processors such as 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. As before, `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.

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 **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

ASSIGNMENT SPECIFICS

Begin by copying all of your Project 3 files into a new Project 4 directory, e.g.:

```
$ cp -r cs3339_project3/ cs3339_project4/
```

Only the CPU and Stats classes should have to be changed from Project 3. You will probably want to change the parameter lists of some Stats member functions, and you will probably want to add some member variables to Stats.

I recommend the following approach:

Modify your `registerSrc` and `registerDest` functions to take a second argument, specifying the earliest pipeline stage in which the source will be needed or the result will become available, respectively. In addition to tracking the destination register of each op in the pipeline, also track the stage in which that instruction's result will become available for forwarding.

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.

You can again check your timing results using the equation instrs = cycles – 7 – bubbles – flushes. Note that your flush count should not change from Project 3, but that you should see significant reduction in bubble count (and, consequently, significant improvement in CPI) due to the addition of forwarding paths.

The reference output file for the sssp.mips is provided on Canvas and your output must match this format verbatim. See below for additional instructions on using the provided script to check the output as well as your submission files. The file `project4_expected.txt` on Canvas has expected bubble/cycle counts for all inputs.

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`) Turn off debugging in your submission.
- Make sure your code's runtime is not excessive
- Make sure your code is correctly indented and uses a consistent coding style
- 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 in the directory containing your source code:

```
$ tar czvf yourNetID_project4.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 submitted in a comment or separate text file (not inside tarfile) on Canvas named `yourNetID_README.txt`.

Use the submit_test script that was provided for Project 3 to confirm the contents of your tar file, ability to compile and execute on zeus, and check your output. You will need to copy over the input sssp.mips and use the new version of sssp.out provided with Project 4. The remaining instructions are in the script itself. ***Submissions that do not untar correctly and compile on zeus with this script will receive zero points, so you are highly encouraged to check your submission.***

As in the prior projects and assignments all files are to be submitted using Canvas. Note that files are only submitted if Canvas indicates a successful submission. Pictures of timestamped files on your local machine or zeus will not be accepted as proof of work completed.

You may submit your file(s) as many times as you would like before the deadline. Only the last submission will be graded. ***Canvas will not allow submission after the deadline***, so I strongly recommend that you do not come down to the final seconds of the assignment window.