ECE 4100/6100 - Fall 2023
Lab 3: Out-of-Order Pipeline with In-Order Commit

Dr. Callie Hao
Version: 1.1

Part A Due: October 13th 2023 @ 11:59 PM ET
Part B & C Due: October 27th 2023 @ 11:59 PM ET

Changelog

- Version 1.1 (10/03/2023): Clarified the grading breakdown for parts B1 - B4 and C1 - C4. Removed references to fvalidate since it no longer exists.
- Version 1.0 (9/28/2023): Initial release.

1 Rules

- **This is an INDIVIDUAL assignment.** You may discuss this assignment with classmates, but you should code your assignment individually (i.e. no code sharing). **All honor code violations will be reported to the Dean of Students. NO EXCEPTIONS.**
- These lab assignments are very difficult and take a lot of time. It is in your best interest to start early.
- See the "Late Policy for Assignments" Canvas announcement for the course's late policy.
- Please utilize TA office hours, Piazza, and recitation if you have questions. **Do not go to the professor's office hours with lab-specific questions.**
- Read the entire document before starting. Not only is it critical to understanding the assignment, but most questions can be answered by reading the corresponding section.
- Sometimes, mistakes will be found in the PDF and the lab assignment. **It is solely your responsibility to ensure you are using the most up-to-date PDF and lab files.**
- Make sure that your code works with C++11 as this is the version the autograder will run your code with.

2 Objective

The objective of the third programming assignment is to do a performance evaluation of an out-of-order machine. In particular, you will equip the code to do register renaming, implement different scheduling algorithms (in order and out-of-order), and use a ROB to maintain a precise state. You will initially implement a 1-wide pipeline and later extend it to a 2-wide superscalar pipeline.

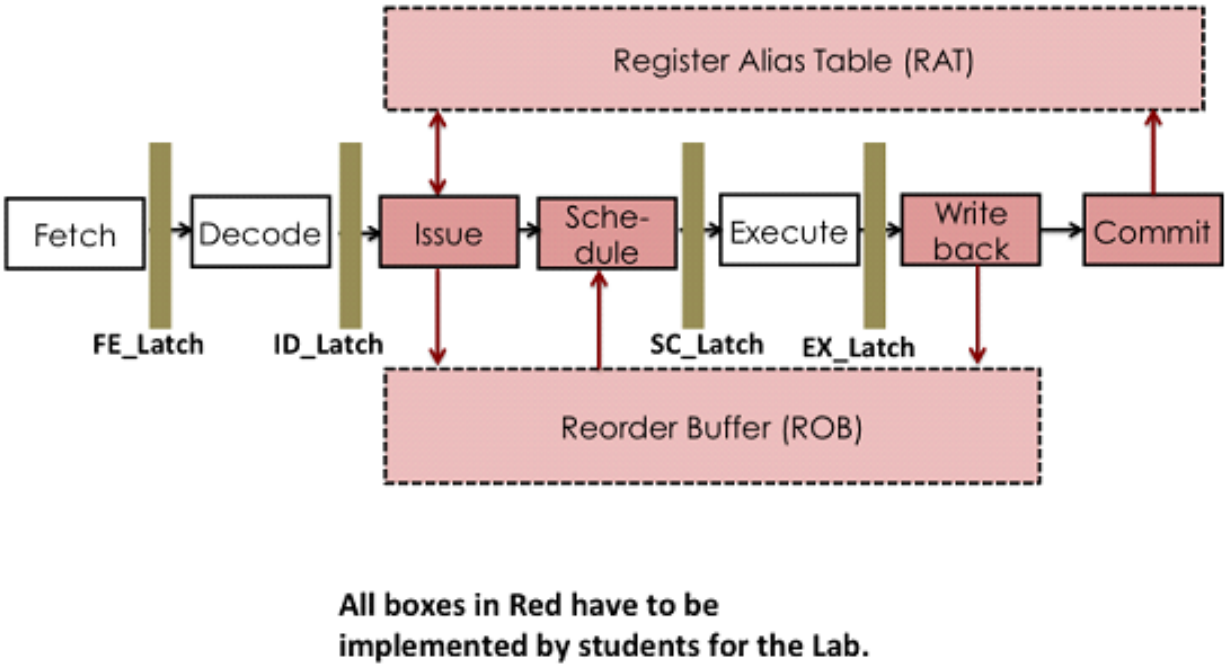

Figure 1: Seven stage out-of-order pipeline

3 Problem Description

A seven stage out-of-order pipeline is shown in the figure above. It consists of `Fetch, Decode, Issue, Schedule, Execute, Writeback,` and `Commit` stages. The newly added units include the **Register Alias Table (RAT)** and the **Reorder Buffer (ROB)**. More details about these individual units and newly added stages can be found in the handout for this lab. To simplify this assignment, we will assume perfect branch prediction. We will ignore the `cc_read` and `cc_write` operations generated by the instructions (otherwise, you would need to rename the cc register also). We will also assume that memory instructions (`LD/ST`) in the pipeline do not conflict with each other, to avoid the complexity of the hardware associated with memory disambiguation. We will assume that all instructions, except `LD`, incur a latency of 1 cycle to execute. The latency of the `LD` instructions can be varied using configuration parameters.

We will use a trace driven simulator that is strictly meant for doing timing simulation. To

keep the framework simple, we will not be doing any functional simulation –which means the trace records that is fed to the pipelined machine does not contain any data values, and your pipeline will not track any data values (in REG, Memory, ROB, PC) either. Furthermore, the traces only contain the committed path instructions. The purpose of our simulation is to figure out how many clock cycles it takes to execute the given instruction stream, for a variety of different machines such as different scheduling policies, LD latencies, and pipeline width.

You will be provided with a trace reader, as well as a pipeline machine for which the `fetch()`, `decode()`, and `exe()` stage are already filled for you.

4 Design

Part A: Create the functionality for the two newly added units RAT and ROB. We have already created the `*.h` files for these objects, and your job is to fill the corresponding functions in the `*.cpp` file. The objective of this part is simply to create the three objects and compile them independently (using `g++ -c filename.cpp`). Note that you will not be able to debug these structures just yet, as you will need to have a working pipeline for testing. So, for this part, as long as you fill the `*.cpp` and submit the two files that can compile without error you will receive 1 point. You will be able to change these files for Part B. Note that if you do not do this part with seriousness, you may find Part B impossible to finish within a few days!

Part B: For a 1-wide machine,integrate the created objects in your pipeline and populate the four functions of the pipeline: `issue()`, `schedule()`, `writeback()`, and `commit()`. You will implement two scheduling policies: in-order and out-of-order (oldest ready first)

Part C (Extra Credit): Extend your OoO machine to be 2-wide superscalar. (Add `-pipewidth 2` to command line.)

5 Implementation Details

You have been provided the following files:

- `pipeline.cpp` - **(Part B & C)** The functions `pipe_cycle_issue()`, `pipe_cycle_schedule()`, `pipe_cycle_writeback()`, and `pipe_cycle_commit()` need to be implemented for providing pipeline functionality.
- `pipeline.h` - **Do not modify.** Header file containing structs and definitions necessary for the simulator.
- `sim.cpp` - **Do not modify.** Responsible for opening the trace, initialization, and instating and executing the pipeline.
- `trace.h` - **Do not modify.** Header file containing structs and definitions pertaining to instructions from the traces.

- `rob.cpp` - **(Part A)** Where your function for providing reorder buffer (ROB) functionality will go.
- `rob.h` - **(Part A)** Header file containing structs and defintions necessary for the reorder buffer. You should read this to have good understanding for part A.
- `rat.cpp` - **(Part A)** Where your function for providing register alias table (RAT) functionality will go.
- `rat.h` - **(Part A)** Header file containing structs and defintions necessary for the register alias table. You should read this to have good understanding for part A.
- `execq.cpp` - **Do not modify.** Implements simulated execution functionality for your simulator.
- `execq.h` - **Do not modify.** Header file containing structs and defintions necessary for `execq.cpp`.
- `Makefile` - **Do not modify.** A Makefile for compiling your code. Can be used to invoked the following commands...
  - `make` and `make all`: Compiles your code and creates an output file so you can run your code.
  - `make clean`: Cleans out compiled files.
  - `make fast`: Compiles your code with the `-O2` flag. See this for details.
  - `make debug`: Compiles your code with the preprocessor definition DEBUG defined. This causes code blocks of `#ifdef DEBUG ... #endif` to compile. **Highly recommend using for debugging purposes!**
  - `make profile`: Compiles your code for use with `gprof`. Helps diagnose bottlenecks if your code is running slowly.
  - `make submit`: Creates a tarball for your submission. Note: DO NOT change the name of the "src" directory, or your submission will not run with the autograder!
  - `make validate`: Compiles your code and runs `runtests.sh`.
  - `make runall`: Compiles your code and runs `runall.sh`
- `traces/` - **Do not modify.** A directory containing the execution traces for this project.
- `scripts/` - **Do not modify.** A directory containing the following scripts...
  - `runall.sh`: Runs your code on `bzip2`, `gcc`, `libq`, and `mcf`. The results of each run is placed in the corresponding `.res` file in `results/`. A summary of all the results can be found in `report.txt`, which is placed in whichever directory `runall.sh` is ran from (e.g. if you call `make runall` the report will be put in `src/`).
  - `runtests.sh`: Runs your code on `gcc` and `sml` and verifies that your `NUM_INST`, `NUM_CYCLES` and `CPI` match the reference statistics for each test.

- `results/` - **Do not modify.** A directory containing the output of your code from `runall.sh`.
- `ref/` - **Do not modify.** A directory containing the reference outputs for `gcc` and `sml`.
  - `results/`: A directory containing the reference outputs for `gcc` and `sml` for each individual part of the lab.
  - `refoutput_gcc.pdf`: A report summarizing the reference outputs for `gcc`.
  - `refoutput_sml.pdf`: A report summarizing the reference outputs for `sml`.

5.1 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-pipewidth <width>`: The number of pipelines to simulate (1 by default)
- `-loadlatency <num>`: Sets the number of cycles it takes for LD to execute (4 by default)
- `-schedpolicy <num>`: The scheduling policy your simulator will use. There are 2 options:
  - `0`: In-order
  - `1`: Out-of-order (default)
- `-h`: Print usage information

Here's an example of running your code on a single trace using these simulator parameters: `./sim -pipewidth 2 -schedpolicy 1 -loadlatency 4 ../traces/sml.ptr.gz`

5.2 Simulator Statistics

The simulator keeps track of a variety of statistics, some of which you may have to update:

- `stat_retired_inst`: Counts the number of committed instructions.
- `stat_num_cycle`: Counts the number of simulated CPU cycles.
- `cpi`: Defined as `stat_num_cycle / stat_retired_inst`.

6 Debugging & Debug Outs

For debugging purposes, the Makefile has an option called `make debug`. This allows code enclosed in `#ifdef DEBUG ... #endif` blocks to compile alongisde your other code. For instance, say I have the following...

```
int x = 2;
#ifdef DEBUG
    x = 4;
#endif
printf(%d, x);
```

If you ran this with `make` by itself the output would be `2`. However, if you run it with `make debug` the output would be `4`. If we were to replace this example with `printf` statements, then we would only ever output when we call `make debug`!

Additionally, since debugging can be cumbersome we have provided debug outputs for you on Canvas. These debug outs are made by collecting the first 100,000 lines of the solution running `gcc` and `sml`, then putting the output into a text file. In order to match these `debug_outs`, you **must** must add the `printf` statements provided in `debug_out_printfs.txt` to your code. The parameters for these statements have been purposefully left blank; you have to fill them in on your own depending on your implementation. When adding the `printf`'s, be sure to wrap them in `#ifdef DEBUG` blocks so they only print when you call `make debug`.

To check your implementation using the debug outs, edit your code according to the type you're using. Then, run your code with the trace and the simulator parameters you want. Then, set your output to only go to 100,000 lines and set the destination of your output to some file. For example:

```
./sim <necessary flags> ../traces/gcc.ptr.gz | head -n100000 > gcc.my_debug_out.res
```

After running the above, you can compare your output to the provided debug out's output using `diff`...

```
diff gcc.my_debug_out.res <provided gcc trace> > diff.txt
```

This will create a file `diff.txt` and place the output of the `diff` command inside it. At this point, you can go to the line numbers of your `.res` files indicated in `diff.txt` to see what is going wrong and fix it. If you run the `diff` command above and `diff.txt` is empty, you should call `make validate` to see if your results match!

7 Deliverables

To prepare your submission, run `make submit` to generate a tarball (`tar.gz`) with your code. **Do NOT change the name of the "src" directory**, as the autograder will not be able to test your submission. Please avoid including unnecessary files like traces, the assignment's PDF, etc. Make sure to untar your submission to ensure all the required files are there before submitting. **You are solely responsible for what you submit.**

To see if you can receive full credit, you should run `make validate` and ensure you are passing. Note that just because you pass `make validate`, this does **NOT** mean you get a

100%. Your submission will be checked using traces that you are not given. Furthermore, we reserve the right to examine any submission if we suspect plagiarism or attempts to game the simulator in any way.

8 Grading

This lab is worth 10 points in total. Note that this project is 10% of your final grade, meaning that each point equates to one point towards your final grade. There are 2 extra points of extra credit available by doing part C, meaning you can get 2 bonus points towards your final grade.

While there is no strict efficiency requirement, please ensure your simulator finishes each trace in less than a minute. This is to ensure expedient grading, since verification would be difficult and tedious otherwise. If your simulator is taking too long, we recommend you profile your code to see where most your code spends its time running. You can use `make profile` to allow your code to be used with `gprof`.

8.1 Part A (1 point)

Your submission for part A is evaluated based on whether or not your code compiles successfully, regardless of whether or not your implementation is wholly correct. If your code compiles without error *in a reasonable amount of time*, you will get 1 point. Otherwise, you will get 0 points.

8.2 Part B (9 points)

For part B, you submission will be checked against 4 different simulator configurations. For each configuration, you will be evaluated based on how your simulator's CPI compares to the reference CPI for `bzip2, gcc, libq,` and `mcf`. If you are within 1% of the reference CPI, you will get full credit for that calculation. If it is within 5%, you will get half credit. The following is how many points you get **per trace**:

- B.1: Schedule in-order, 1-cycle load latency, 1-wide pipeline (2.25 points)
  - **CPI**: 0.5625 or $\frac{9}{16}$ for within 1%, half-credit for within 5%, 0 otherwise
- B.2: Schedule OoO oldest first, 1-cycle load latency, 1-wide pipeline (2.25 points)
  - **CPI**: 0.5625 or $\frac{9}{16}$ for within 1%, half-credit for within 5%, 0 otherwise
- B.3: Schedule in-order, 4-cycle load latency, 1-wide pipeline (2.25 points)
  - **CPI**: 0.5625 or $\frac{9}{16}$ for within 1%, half-credit for within 5%, 0 otherwise
- B.4: Schedule OoO oldest first, 4-cycle load latency, 1-wide pipeline (2.25 points)
  - **CPI**: 0.5625 or $\frac{9}{16}$ for within 1%, half-credit for within 5%, 0 otherwise

8.3 Part C (2 points)

For part C, your submission will be checked against 4 simulator configurations. These configurations are the same as part B **except now the width of your pipeline is 2**. For each configuration, you will be evaluated based on how your simulator's CPI compares to the reference CPI for `bzip2, gcc, libq,` and `mcf`. If you are within 1% of the reference CPI, you will get full credit for that calculation. If it is within 5%, you will get half credit.

- C.1: Schedule in-order, 1-cycle load latency, 2-wide superscalar (0.5 points)
  - **CPI**: 0.125 or $\frac{1}{8}$ for within 1%, half-credit for within 5%, 0 otherwise
- C.2: Schedule OoO oldest first, 1-cycle load latency, 2-wide superscalar (0.5 points)
  - **CPI**: 0.125 or $\frac{1}{8}$ for within 1%, half-credit for within 5%, 0 otherwise
- C.3: Schedule in-order, 4-cycle load latency, 2-wide superscalar (0.5 points)
  - **CPI**: 0.125 or $\frac{1}{8}$ for within 1%, half-credit for within 5%, 0 otherwise
- C.4: Schedule OoO oldest first, 4-cycle load latency, 2-wide superscalar (0.5 points)
  - **CPI**: 0.125 or $\frac{1}{8}$ for within 1%, half-credit for within 5%, 0 otherwise

9 Reference Machines

There are two reference machines you can use for the course:

- **ECE**: ece-linlabsrv01.ece.gatech.edu (email)
- **CS**: shuttle3.cc.gatech.edu (help form)

We highly recommend that both CS and ECE students obtain access to each other's servers as these servers have been shown to go down at times. You can contact the respective help desks to request access. Since the servers can go down, it is crucial that you do not exclusively work on the servers while working on these projects. We recommend working on your code locally using services like WSL that will simulate a Linux environment for you to work and verify your implementation in.

10 FAQ

1. **What are the convention changes from Lab 2 to Lab 3?**
   We will use a struct called `inst_info` to track both the ISA state (`src1_reg, src2_reg, dest_reg`) as well as microarchitectural state (`src1_tag, src2_tag, dr_tag, src1_ready, src2_ready`) as well as simulation metadata (`inst_num`). You can use `inst_num` to estimate the age of the instruction. Furthermore, we will use "-1" to denote invalid values or when a value is not needed. For example, if `src1_reg = -1` it means `src1_reg` is not needed. This simplifies the number of variables the `inst_info` needs to carry throughout the pipeline.

2. **Why are we not simulating condition code?** It will significantly increase the complexity of the pipeline as you would need to rename the cc for each instruction that does a `cc_write`. So, in essence an instruction would have two sources and two destinations (`destreg` and `cc`). To keep the simulation model tractable (so that students can finish this assignment in a couple of weeks), we will ignore `cc_read` and `cc_write` (in fact, we are ignoring branch instructions altogether, and they are treated as `OP_OTHER` instructions).

3. **How should I get started?**
   We strongly recommend that you read through the header files first to get a sense of what data structures are available to you and what you must implement. The header files should provide documentation for everything you need to know to complete this lab.

4. **Why aren't instruction addresses unique in the trace?**
   During trace generation, the complex x86 instructions having multiple operations at a particular address were converted to simpler operations having the types provided in the trace header file. These simpler instructions would then have the same instruction address. The instruction address is thus not a unique identifier for an operation. (op_id is supposed to be used for that)

5. **What is `pipe_print_state()`?**
   This function allows you to print the state of the pipeline at any time for debugging purposes. If you use it when debugging, please remove any calls to it before you submit your work. Submitting your code with calls to `pipe_print_state()` will result in extraneous output, which may result in you not receiving full credit.

6. **How can I test my code?**
   Reference outputs for `gcc.ptr.gz` and `sml.ptr.gz` are provided in the ref directory as `refoutput_gcc.pdf` and `refoutput_sml.pdf`, respectively. You can run the script `runtests.sh` located in the scripts directory to compare your implementation's output with these reference outputs.

7. **I get an error when I try to execute `runall.sh` or `runtests.sh`**
   Check that you have execute permissions on both scripts. Add execute permissions to these files using this command: `chmod +x runall.sh runtests.sh`

8. **What do I do if I get a `no space left on device` error?**
   You can run `du -sh *` to see what is taking up space in your current directory and use `rm` and `rm -r` to remove files and directories respectively. If you are running `./runall.sh`, ensure that you have cleared out the `results/` directory as the files here can be very large if you ran `./runall.sh` with your print statements.

Appendix A - Plagiarism

Preamble: The goal of all assignments in this course is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives

an unfair advantage over students who do the work and follow the rules.

1. As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code. The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."

2. You must submit an assignment or project as your own work. Absolutely no collaboration on answers is permitted. Absolutely no code or answers may be copied from others — such copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.

3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person*").

4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.

6. Students suspected of Academic Misconduct are informed at the end of the semester. Suspects receive an Incomplete final grade until the issue is resolved.

7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.

8. If you are not sure about any aspect of this policy, please ask Dr. Hao