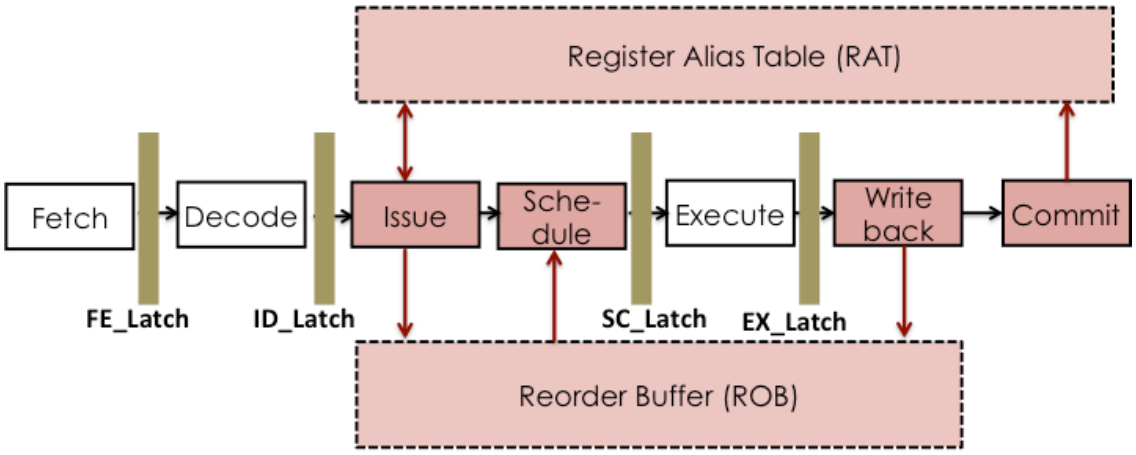
CS 4290 / CS 6290 / ECE 4100 / ECE 6100
Advanced Computer Architecture

Lab 3: Out-of-Order Pipeline with In-Order Commit (10 Pts + 3 extra credit)
Part A Due: Friday, October 13, 2023 (11:55 pm)
Part B(+C) Due: Friday, October 27, 2023 (11:55 pm)



All boxes in Red have to be implemented by students for the Lab.

This is an individual assignment. You can discuss this assignment with other classmates but you should code your assignment individually. *You are NOT allowed to see the code of (or show your code to) other students or of past submissions that may be available online.* We will be using software that detects code similarity. If you think you collaborated closely enough with another student that could lead to more code similarities than would normally be expected, please declare it upfront, at time of submission (as a comment in your Canvas submission). *Please note that such declaration is not granting you an excuse for copying code.*

We already found ourselves in the unpleasant situation of having to report a few students to OSI for suspected Lab 2 code plagiarism, flagged by our similarity check software. Please remember that you are required to adhere to the Georgia Tech Honor Code.

OBJECTIVE
The objective of the third programming assignment is to do a performance evaluation of an out-of-order machine. You will equip the provided code skeleton to perform register renaming, implement different scheduling algorithms (in-order and out-of-order), and use a ROB to maintain a precise state. You will add a small fully associative L1 data cache with LRU replacement policy, which will dictate the latency of load instructions. You will initially implement a scalar pipeline and later extend it to a 2-wide superscalar pipeline. Additionally, you will implement primitive exception handling that will flush the pipeline and pipeline structures when a committing instruction raises an exception.

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. It is strongly recommended that you go through all source code files and the handout carefully and make sure you understand the simulator's structure before you proceed to write any code.*

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 as well). 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 each LD is either fixed to 1 cycle or obtained from the cache[1].

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 are fed to the pipelined machine do not contain any data values, and your pipeline will not track any data values (in Registers, Memory, ROB, PC) either. Furthermore, the traces only contain the committed path of 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. The trace files are different from those used in Lab 2. Your objective is to do the following:

Part A (1 point): Implement the functionality for the two newly added units, RAT and ROB. We have already created the *.h files for these objects, and you should 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 take this part seriously, you may find Part B impossible to finish within a few days!** We will not be providing a sample solution for Part A.

[1] While store instructions (ST) are memory operations as well, from the processor's perspective they always complete in a single cycle (as soon as they are placed in the store queue). The store queue thus ensures that stores are never on the critical path. Committed stores eventually drain from the store queue to the memory system.

Part B.1 (6 points): For a scalar 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 instruction ready first).

What experiments to run:
B.1.1 Schedule in-order, load latency (1)
B.1.2 Schedule OoO-oldest first, load latency (1)

For both of these experiments the load latency is fixed.
Look in runall.sh for the commands.

Part B.2 (3 points): Implement a Fully Associative 512B cache with a block size of 64B and an LRU (Least Recently Used) replacement policy. The latency of a load instruction that misses the cache should lead to a load latency of 2 cycles, while a load that misses in the cache leads to a 20 cycle latency. The only purpose of this cache is to simulate variable load latency. As mentioned earlier, the latency of store instructions will always be a single cycle. For simplicity, we assume that an access (load or store) that misses in the cache immediately installs its corresponding cache block in the cache. Thus,

(i) a store will always return a 1-cycle latency and update the LRU state (if a hit) or insert the corresponding cache block (if a miss).
(ii) a load will return the HIT latency and update the LRU state, if a hit, or return the MISS latency and install the corresponding cache block, if a miss.

Make sure to read the comments in cache.cpp and cache.h. You ARE allowed to change the cache.h file.

What experiments to run:
B.2.1 Schedule in-order, cache on (use the *-cache* command line argument, as can be seen in the runall.sh script)
B.2.2 Schedule OoO-oldest first, cache on

The only difference between parts B2 and B1 is the cache. Make sure that the pipeline is using the cache by adding the cache flag in the command.

Part B.3 (Extra Credit: 1 point): Implement primitive exception handling to flush the pipeline, handle the exception, and replay previously fetched instructions when an instruction raises an exception upon commit. Note that the implementation of exception handling is 'primitive' because of two simplifying assumptions: (i) it does not need to support nested exceptions, and (ii) no new exception may occur while you replay instructions that were previously flushed from the pipeline because of an exception.
At the pipeline's commit stage, you should check if an instruction that is about to commit raises an exception using the is_exception and exception_handler_cost fields from the trace. The is_exception parameter is set to true if the instruction about to commit raises an exception. If is_exception==true, then the exception_handler_cost field is a positive integer value N, which

represents the number of cycles the hypothetical exception handler takes to execute. Remember that your simulator only models performance, not the functional execution of instructions or exception handlers. After *N* cycles have passed, your pipeline should start fetching again the instructions that were flushed because of the exception, starting from the instruction that raised the exception. See under the 'Exception Handling' comment in the Pipeline struct for potentially helpful variables in implementing exception handling. Ensure that once the exception is handled, the same instruction should not raise an exception again.

For Part B.3, you will run two experiments:
B.3.1 Schedule in-order, exceptions on, cache on (by using the command line argument *-exceptions*, as can be seen in the runall.sh script)
B.3.2 Schedule OoO-oldest first, exceptions on, cache on

Part C (Extra Credit: 2 points): Extend your OoO machine to be 2-wide superscalar. Run the same two experiments as in part B.2, but for a 2-wide machine.

WHAT to SUBMIT:

A. For Part A, submit rat.cpp and rob.cpp (note that you are allowed to change only the *.cpp files and not the *.h files).
B. For Parts B and C, you need to submit all the files in the src folder (don't submit *.o files)

Reference output for testing:
We provide an additional small trace (sml) in the traces directory, along with a sample expected output for it in the reference directory, as a point of comparison for the outputs produced by your solution. The reference directory also includes a reference output for the gcc trace.

Gradescope autograder:
A simple autograder on Gradescope generates a score based on the difference of the reference outputs and your code's outputs for the small and gcc traces runs a similar script after you submit the required files. Please note that the autograder is not comprehensive and your code will be tested against additional traces.

You can use the virtual machine **oortcloud.cc.gatech.edu** to develop and test your code. Your submission must compile, run, and produce the correct results on Gradescope. Please ensure that your submission on Gradescope produces the desired output (without any extra printf statements).

FAQ:

1. What are the convention changes from Lab2 to Lab3?

We will use a struct called inst_info to track both the ISA state (src1_reg, src2_reg, dest_reg) as well as the microarchitectural state (src1_tag, src2_tag, dr_tag, src1_ready, src2_ready), the simulation metadata (inst_num), and a few additional fields required for part B.3 (exceptions) and cache lookup. You can use inst_num to deduce 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 reduces the number of variables the inst_info needs to carry through the pipeline.

2. Why are we not simulating condition codes?

It would 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).