Introduction

This was a difficult project. With so many intricacies and working in a new unfamiliar language, I was (disappointingly) unable to complete a working implementation. As such, I would like to lay out my partially complete solution alongside theoretical implementation for the missing pieces.

Codebase Overview

I attempted my implementation in c++20.
The file breakdown is as follows:
main.cpp - main execution file handling user arguments,instantiating the cpu and stepping it
cpu.cpp - representation of our cpu
instruction.cpp - representation of the 9 instructions used in this project
registers.cpp - representation of our virtual registers and physical registers including physical registers used by the ROB
ooo.cpp - crux of our out-of-order execution. hold the data structures necessary to facilitate ooo execution including our reservation stations, ROB, register status table, and
termcolor.hpp - header only library for nice colored output

Technical Details

CPU

Our cpu class consists of a few main parts which are in turn their own structs. Most importantly, it contains a step function which simulates one clock cycle of the cpu. This function consists of two main parts. First we execute all of the stages of the pipeline based upon the output of stages from the previous cycle. This means we must fetch, decode, and issue instructions in order, then for each of our functional units we must obtain an instruction from the reservation station if a ready one is available, or otherwise execute the instruction currently in the functional unit. If any of our functional units complete execution on this cycle, their results should be put on a CDB buffer. Finally, if there are any remaining CDBs we look at the front of our ROB and any instructions which have completed can have their result written to the CDB and their ROB entry freed. To handle pipe-lining, we keep buffers of instructions that hold the instructions fetched or decoded on this cycle, so that in the next cycle these buffers can be read to simulate reading from the previous in-order phase.

Registers

We must represent a total of 64 architected registers, as well as 32 general purpose physical registers and 16 rename registers. We represent these as enums, so that they may be cast to floats to handle and interchange easily. Additionally, since every register can hold either an int or a float of width 32, we represent these as a union of an int and a float. We must also handle register renaming in the decode stage, so we need a map table as well as a free list. The map table is represented as a map of architected regs to vec of physical reg. Whatever physical reg is as the rear of the vector is the current location of the architected reg. When a register is commited from the ROB it is removed from the physical reg vector of the map table and returned to the free list. Correspondingly, the free list is a simple vector of physical registers that is popped from when we allocate a reg and pushed to when we free one from the map table.

Instructions

Instructions represented the largest initial challenge to represent. I initially took an object oriented approach, attempting to use a parent instruction class from which three intermediate classes (memory, arithmetic, and branch) inhereted from, and which in turn had child classes to represent each individual instruction. This approach created several issues most likely stemming from my lack of knowledge of c++. Facing this, I

instead pivoted to a single struct to represent every instruction. This consisted of simply its address, its operation (an enum of all 9 possible operations) and a vector of ints representing its field. In conjunction with our enum to represent register approach, this allows us to represent any field of any operation, whether it be an architected register, physical register, immediate field, or address. This decision made it extremely hard to deal with instructions throughout the cpu, as every time we dealt with an instruction, we would need to switch on the instruction opcode to figure out what kind of instruction it was and interpret the fields accordingly.

Reservation Stations

Reservation stations are the second most important part of an out of order execution algorithm. We represent the entirety of our cpu's reservation stations in a struct we call our reservation station file. This file has a field for each functional unit consisting of a vector of individual reservation stations. A single reservation station runs very close to the reservation status table covered in our slides. We have a name, boolean to represent if the field is busy, two value fields that are a union of an int and a float, two fields that are physical registers (so they may either come from the register file or from the ROB's renaming registers) a physical register representing the destination, and finally an address field for memory operations.

ReOrder Buffer

The Reorder Buffer is the most important part of our out of order execution algorithm. We represent it in a struct as a vector of ROB entries. In conjunction with a head pointer, this allows us to simulate the circular behavior of the buffer. A single ROB entry consists of a boolean indicating if the entry is being used, a pointer to the instruction its keeping track of, a field representing its state (executing, wrote result, and commit), a physical register representing its write destination, and its value as a union of int and float.

Other OOO Data Structures

We need two more minor data structures to handle OOO execution. First, we need a simple table of architected register to location, where location can either be the register file, ROB, or CDB. Similarly, we need a small vector of int float unions representing our CDBs. By keeping these in their own structure owned by the cpu, they can be read from or written to by any other parts of our main cpu.

Speculation

Speculation requires only two additional data structures, a branch predictor and a BTB. Because the branch predictor is a single bit predictor, we can represent it as a boolean. Our BTB is a map of ints to ints. This allows us to index it using the address of our branch instruction to reference a target address.

Conclusion

All in all, I came away from this project feeling much more confident about OOO execution architecture and algorithm. I think my issues mostly stemmed from a lack of knowledge of C++, as well as the data structure I used for my instructions, which allowed too much ambiguity in encoding our fields, forcing us to use a large switch statement every time we wanted to deal with an instruction. Given the opportunity to start again, I would start earlier and attempt to get my initial object oriented approach to instructions working.