Lab 4:
Re-Order Buffer and Tomasulo's Algorithm

NAME: YUHAO ZHANG
STUDENT NUMBER: 2021533141
EMAIL: ZHANGYH7@SHANGHAITECH.EDU.CN

1 INTRODUCTION

The Reorder Buffer can manage out-of-order instruction execution while maintaining program order. As for Tomasulo's Algorithm, using reservation stations and register renaming, improves instruction-level parallelism and uses the ROB for precise program execution.

In this lab, we add a reorder buffer structure in this simulator, and implement the Tomasulo's algorithm.

- Description of my implementation.
- quantitative evaluation and analysis.
- whether the implementation is optimal.

2 IMPLEMENTATION

Using the hints in the document, we implement the ROB and tomasulo's algorithm by adding three extra data structures. They are reorder buffer, reservation station and register status table.

The tomasulo's algorithm is characterized by out of order issue, out of order execution and in order commit, which can ensure good data consistency and handle exceptions well.

2.1 issue

When we issue an instruction, we should assign a corresponding FU and a ROB line for it, so if there is no available FU or ROB, we will not issue it. If FU and the ROB line can be assigned successfully, then we initialize the status of them. We will check the register status table, if the register is not busy, we can straightly use the value in the register and mark Q as -1, which means the oprand is ready. If the register is busy, which means the register is the destination of some other previous instruction, then we mark Q as the index of the register, waiting the computing result.

2.2 execution

When both of the oprands are ready, i.e. Q_j and Q_k equal to -1, FU will execute the instruction with the given oprands. After computing, we mark this reservation station line as finished.

2.3 write back

After execution done, we write it back. This stage does not mean that write back to the register, but pass the result to other waiting instructions, like bypassing in pipeline. At this point, we haven't made any changes to registers or memory.

2.4 commit

The ROB follows the FIFO policy, and we always commit the head of the ROB, and that is why we say the Tomasulo's algorithm is in order commit.

3 EVALUATION AND ANALYSIS

We choose *quicksort.riscv* as our test case here, and we use the pipelined simulator, scoreboard simulator and tomasulo's algorithm simulator to run the test case, and here is the result.

time of rendering(ms)

Simulator	CPI	Data Hazard	Mem Hazard
Pipeline	1.394	93856	23398
Scoreboard	3.434	0	450
Tomasulo	1.929	0	0

Table 1. test case: quick sort

We can see that the Tomasulo's algorithm run faster than the scoreboard algorithm, Since the FU are not pipelined, so the CPI may be larger than the pipelined simulator. As for the branch prediction part, score board seems hard to do the branch prediction and pre-running, but because of the in order commit policy, tomasulo's algorithm can run the predicted branch in advance, if mis-predicted, we can just flush the ROB and keep going.

4 IF THE IMPLEMENTATION OPTIMAL?

I think it is not, obviously. Just for the store and load instruction, if we do not consider the trap handling part, the load and store can be executed in the execute stage, what we need to do is to ensure the data consistency, which is not difficult to check: just traverse the ROB list and we will know it.

5 TEST

There are three test cases for my implementation.

You can test it with the command below:

- ./Simulator ../riscv-elf/helloworld.riscv
- ./Simulator ../riscv-elf/ackermann.riscv
- ./Simulator ../riscv-elf/quicksort.riscv

Or you can get more information in the README.md.