Design of a dual-issue out-of-order execution RISC-V CPU with branch predictor

Members: Shuqi Xu, Andris Huang

Repository: asic-project-fa24-phhxh (Design included)

Introduction

In this project, we designed a CPU with out-of-order execution (OoOE) using the Tomasulo algorithm adapted from the LEN5 architecture [1]. The main motivation behind the choice of OoOE is to increase the functional units' utilization rate to reduce the pipeline's latency. This scheme is especially appealing when complex operations are executed, where one instruction that takes multiple cycles to complete can unnecessarily stall the entire pipeline for an in-order processor. In our OoOE processor, the instructions are dynamically scheduled so that when, for instance, a cache miss occurs, the following instructions that do not depend on the data can be executed first.

Design brief

Our design has an issue block FIFO to store fetched instructions, a reorder buffer (ROB) FIFO to store the instruction order and commit the result when it is ready, a register status table to store which ROB entry will commit the result, and three reservation stations (RS) for branching instructions (B-type, jal, jalr), ALU instructions (R-type, I-arithmetic-type, auipc, csrrw if rs1 is not ready), and load/store instructions (I-load-type, S-type). Some instructions will just issue the ready result to ROB (lui, csrrwi, csrrw if rs1 is ready). Each RS is connected to its dedicated execution unit, i.e. branch unit, ALU unit, and memory unit. When the execution result is ready from the execution unit, its RS will broadcast the result to other RS and ROB through the common data bus (CDB). RS will choose an instruction that has all operands ready to execute. With CDB to transmit data among RS, RS to choose ready instructions to execute, and ROB to commit in program order, our CPU is able to execute instructions out-of-order. For more details, please see our GitHub repo!

Store challenge and our solution

One tricky part to design is the load and store execution unit. Store instruction can only execute only when it is at the top of ROB, otherwise the program order will be lost. Thus the memory unit should store the data to memory only when ROB commits. Load/store RS is a FIFO that executes in load store instruction order. If a naive design lets store instructions wait for the ROB commit signal to execute, then the memory unit is also sitting there idling and wasted. To circumvent this, we came up with a store buffer that temporarily stores the data in it and issues an execution completion signal to load/store RS, then load/store RS treat it as completed, and send the result to ROB. When ROB commits the store instruction, it will send the commit signal to trigger the real memory store. A priority encoder is used to give store higher priority than load to use the memory to reduce the stall when ROB waits for this action to initiate. Latter load instructions will also check the store buffer to get the latest result.

Early branch improvement

An optimization we made is early branching. If the branch unit finds out an issued branch instruction has a wrong prediction, or it is a jal/jalr that must branch to the calculated address, it will directly ask the frontend to fetch instructions from its new address, and the issue block will empty its entries for correct instructions as well as pause issue. Then the branch unit will save the misprediction flag to branch RS, and branch RS will clean out its following branch instructions and only wait to broadcast the previous branching verification results to ROB one-by-one. Once the branch instruction reaches the top of the ROB table, it will reset all RS, register status table, and signal the issue block to resume issues. This early branch design could hide instruction fetch latency which in turn can reduce the branch misprediction penalty.

Dual issue

Because our Tomasulo design has 3 execution units, it is natural to design a dual-issue CPU to utilize them more efficiently. We designed the dual issue strategy to issue at most two instructions to the execution units at a time. To achieve this, we doubled the read/write ports for the issue block, register file, register status table, ROB, and CDB. The issue logic will issue two instructions only if the two required RS are empty and ROB has two ready entries. So the best scenario is that two adjacent instructions are from different types. Unfortunately, the benchmark is not designed in this way, so the dual issue rate is only about 10-20%. And frequent jal/jalr will also undermine the gain from the dual issue. On top of that, the frontend can also be optimized for dual instructions fetch. This requires a cache with two read and output ports which we run out of time to implement. As a consequence, the ceiling of our CPU design is not fully achieved in this report.

Branch predictor

In our design, we noticed that the number of cycles stalled in the OoOE pipeline due to a branch prediction miss is quite large, and a lot of issued instructions will just be washed out. We thus decided to implement a branch predictor. We implemented a predictor using a branch history table. Particularly, the PC address is used as the index and tag in a history table to read out the previous prediction result, and a 2-bit counter is used to indicate the confidence of the prediction. Using this design, we were able to reduce the total cycle counts by about 15%. After storing the branch address in the history table, the critical path is reduced by about 40%.

Performance

Using our speculative dual issue Tomasulo OoOE CPU, we run the benchmarks either with a real memory system integrated with a direct-mapped cache that needs multiple cycles to load and store data (minimum 2 for load, 3 for store, if cache hit), or with an ideal memory system (1 for both load and store in all situations). The results in Fig. 1 and Fig. 2 show that the cycle ratio of a real memory system to an ideal memory system on average is only **1.535**, which shows the success of OoOE to accelerate tasks with high latencies. We also tested a 2-way set associative cache, but the improvement is negligible compared to the direct-mapped cache for the benchmarks (third decimal of the ratio). After place and route we get a clock period of 9.05ns with a direct-mapped cache. See Fig. 3 for timing and floorplan. The total time is 0.524s

$$T_{sum} = N_{total\ cycles} \times T_{period} = 57929714 \times 9.05ns = 0.524s$$

```
[ PASSED ] /home/tmp/eecs151-ace/ASIC/project/bmark_output/cachetest.out       () after 4611826 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/ASIC/project/bmark_output/final.out    () after 8917 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/ASIC/project/bmark_output/fib.out      () after 8126 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/ASIC/project/bmark_output/sum.out      () after 26666753 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/ASIC/project/bmark_output/replace.out          () after 26634092 simulation cycles
```

Fig. 1 Number of cycles with a high **latency memory system** and a direct-mapped cache. 2 cycles for cache hit read, 3 cycles for cache hit write, 6 cycles for external memory load, 15 cycles for write back.

```
[ PASSED ] /home/tmp/eecs151-ace/asic-project-fa24-phhxh-5/bmark_output/cachetest.out  () after 2981828 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/asic-project-fa24-phhxh-5/bmark_output/final.out      () after 5819 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/asic-project-fa24-phhxh-5/bmark_output/fib.out        () after 5408 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/asic-project-fa24-phhxh-5/bmark_output/sum.out        () after 17372765 simulation cycles
[ PASSED ] /home/tmp/eecs151-ace/asic-project-fa24-phhxh-5/bmark_output/replace.out    () after 17372822 simulation cycles
```

Fig. 2 Number of cycles with an ideal memory system. It always takes 1 cycle for all loads and stores.

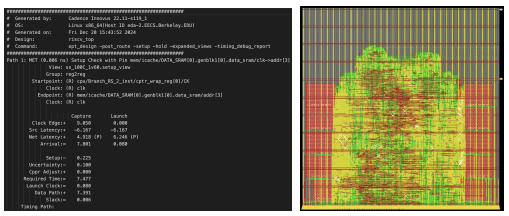

Fig. 3 Timing of critical path and floorplan after place and route.

[1] Caon, Michele. "Design of the execution pipeline for LEN5, an out-of-order RISC-V processor." Master Thesis., Politecnico di Torino, 2019.