Ayush Tibrewal & Anton Melnychuk

Lab 4: Out of Order PARCv2 Processor

Abstract

In this lab, we implement a ROB for an I2O2 processor to make it an I2OI processor. We also modify this processor to allow for speculative processing of the instructions immediately following a branch instruction. The speculative processor uses the ROB which is modified to account for speculatively allocated instructions and a scoreboard which we modified to take speculative instructions in the issue stage only after they have been resolved. The ROB makes the commit stage strictly in order, which ensures that WAW hazards are not an issue for the OoO processor, including when high latency writes occur immediately before a low latency write. The assembly and benchmark tests demonstrate correctness, while our custom tests also show resilience to specific edge cases. The OoO processor gave incredible speedup from having a 3-stage muldiv unit compared to byp's iterative unit, but the actual speedup from out of order execution is minimal. In the two benchmarks with branching, speculative processing gives a 5.5% speedup, but there is a performance hit with the new issue stage since consecutive instructions with a dependency cannot immediately bypass from Issue.

Design

Part 1

We first implemented an ROB to make commits in-order for the OoO processor. This ensures that the write-after-write hazards from the test we designed is no longer an issue. The ROB keeps a circular queue of capacity 16, with each entry containing information about the allocated instruction. When an alloc request is made, the ROB gives the processor the slot that was allocated, which allows the processor to refer to it for later parts of the process. Namely, once the instruction corresponding to a slot has been executed and is in the writeback stage, the processor sends a fill request to the ROB which lets the ROB mark the slot as no longer pending. Throughout this time, whenever the ROB head is at a non-pending but valid entry, it can direct the processor to commit the data to the regfile. We did not differ from the given ROB specs as we did not feel there was a better choice and the design is fairly simple.

Part 2

We add a spec column to the ROB and an additional set of inputs to allow the control unit to resolve speculative slots. Since at most one instruction is speculative in our processor, only one such input is needed. Then, the ROB adjusts the appropriate entry to either be no longer speculative or to now be invalid, and the commit logic now also skips any invalid instructions. We only skip a single entry in a cycle since there are no capacity concerns in the ROB and to keep commit logic simple. We believe that this does not stray from the intended modifications.

We also had to make changes to the scoreboard, despite the spec stating that this would not be necessary. When a branch is mispredicted, allowing speculative instructions into the scoreboard creates incorrect bypassing, whereas correct predictions need to be added to the scoreboard to ensure stale values are not bypassed in favor of the no-longer speculative value. To solve this, we added an interface for the I stage to add speculative instructions in the scoreboard once the branch is confirmed to be not taken. In theory, the entire scoreboard logic could have been moved to the Issue stage, but this felt like a quick solution for the purpose of this lab.

Besides these two changes, and any additional wires in the control side to connect to the new interfaces in the ROB and scoreboard, we did not change anything from the pv2spec starter code given. A clear extension of this project would be to move the issuing logic from Decode to Issue, which would also improve a key slowdown noted later and remove the added interface from the scoreboard.

Testing Methodology

We start by creating a simple **parcv2-ooo.S** test that writes to the same destination twice: a 4-cycle MUL stores 0 to $1, then a 1-cycle ADDIU stores 5 to $1; after several NOPs we read $1. Thus, in the supplied I2O2 core, results are written to the register file as soon as each pipe finishes, so the fast ADDIU commits first, then the slower MUL commits later and overwrites $1; the final read returns 0 and the TEST_CHECK_EQ fails. With a new reorder buffer we should expect MUL results to wait in the ROB until earlier instructions are committed, making the final value 5 and the test passes. Note, all original PARCv2 tests still pass on I2O2 because they never read a register after multiple pending writes to that same register, so out-of-order commits cannot change their observed behavior.

Next, **parcv2-rob-byp.S** is a test that bypasses out of the ROB before commit rather than waiting for commit. After a load (LH) writes to register but before the instruction commits, a dependent instruction reads it again. In Figure 1, we correctly observe the src0_byp_mux_sel to be 5, which is the expected value to bypass from the ROB.

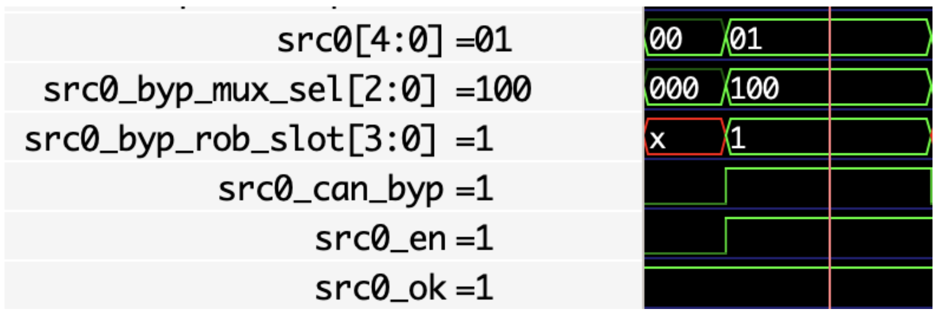


Figure 1: Demonstration of src0_byp_mux_sel = 0b100, indicating bypassing from ROB

We also wrote **parcv2-waw.S**, which tests a WAW hazard scenario that even the original, incorrect OoO processor. A MUL writes to register 2, and later an ADDIU overwrites it. However, there are enough cycles between them that the second write arrives after the first one has completed or is properly tracked in the ROB. This test validates that WAW hazards are correctly resolved when instructions are sufficiently spaced apart, even in the old OoO.

Finally, **parcv2-byp.S** shows a case where the bypass-only design achieves better IPC than both the speculative and OoO processors. This occurs because bypass avoids stalls caused by writeback resource contention or ROB commit delays by relying solely on fast forwarding paths. In this case, even though the OoO has more complex mechanisms, its overhead results in lower throughput under certain conditions compared to a simple pipeline.

Evaluations

	Bin Search		Complex Mult		Masked Filter		Vvadd	
	Cycles	IPC	Cycles	IPC	Cycles	IPC	Cycles	IPC
Byp	1749	0.731275	15312	0.121735	13832	0.325260	473	0.961945
OoO	1695	0.754572	2562	0.727557	6248	0.720070	513	0.886940
Spec Starter	2464	0.519075	3138	0.594009	7693	0.584817	524	0.868321
Spec	2292	0.612129	3138	0.594009	7381	0.635686	524	0.868321

Table 1: Performance Comparison Across Pipeline Strategies for Various Benchmarks

Discussion

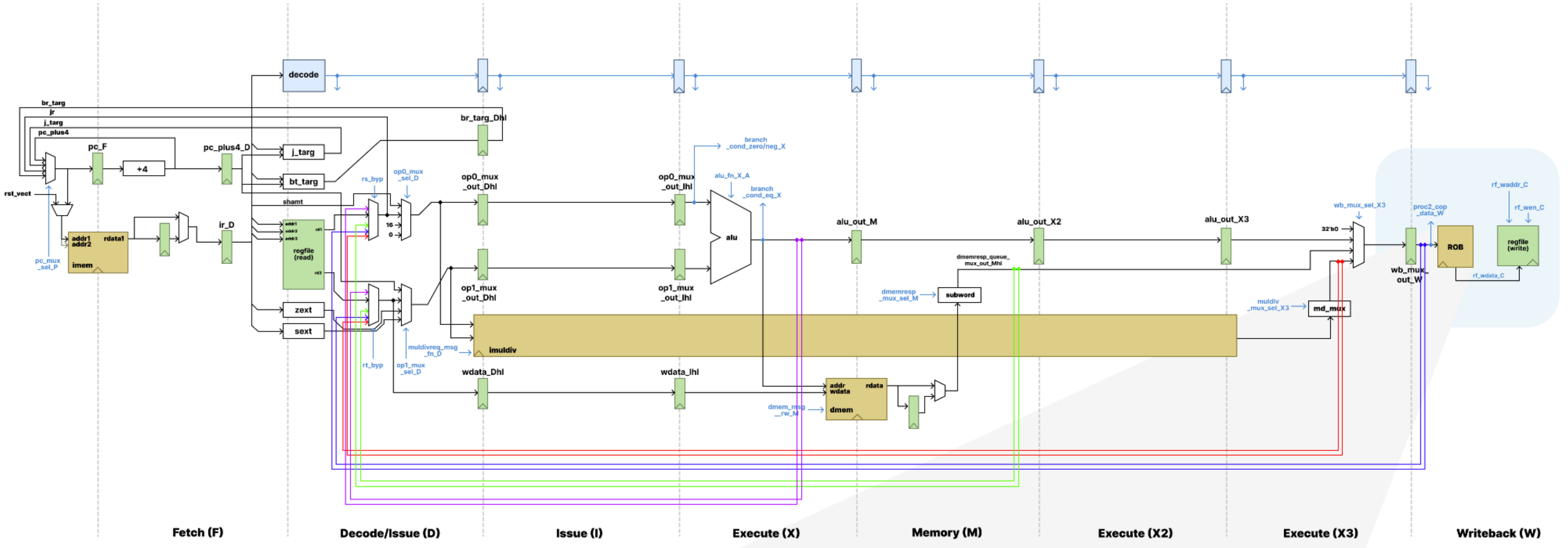
At first glance, the speedup between byp and OoO on Complex Mult and Masked Filter was shocking, especially since the 83% speedup in Complex Mult seems impossible. However, we realized that most of this was the use of an iterative muldiv unit in byp compared to the 3 stage unit in OoO. On the other two benchmarks, the muldiv unit is not used. Then, the speedup from executing multiple instructions at once is much less, and slows down the processor on Vvadd. This occurs due to the writeback hazard demonstrated in parcv2-byp.S. To fix such an issue, it could be possible to have multiple instructions in the writeback stage filling the ROB at the same time, though this would need to involve modifying the ROB further.

Of note is also the new Issue stage that the spec code has compared to the OoO processor. The biggest issue with the Issue stage is that the processor cannot bypass out of the I stage because the result will not yet be available. This causes an extra cycle of stall if a consecutive instruction relies on the result of the previous instruction, which explains the huge slowdown between the OoO processor and the spec processor. However, if the issue stage was fully equipped to do out-of-order issuing, then instructions could be sent from the D stage without yet having all operands available, which would avoid the aforementioned stall. Thus, this slowdown is not a concern given the knowledge that it can be removed with further work.

This lab shows that improving IPC without changing clock speed and instructions per program becomes difficult and narrows to include only certain applications, such as branches, which means not all benchmarks see any improvements. In some cases, such as adding OoO processing, there can even be unintentional slowdowns due to new hazards. However, it is good to note that this could likely be rectified with allowing multiple instructions in writeback, provided the ROB changes appropriately. Of course, this is a good demonstration that increased complexity may not actually yield better results. While out of order execution and parallelizing instructions across multiple pipelines sounds wonderful on paper, this implementation may not have given as strong of an improvement as hoped for.

From this, we see that the Iron Law again holds true. In order to increase IPC without changing clock speed and instructions per program, we must introduce further complexity and logic because this is not usually something that we can get for free. Indeed, the more logic we need to do in a single cycle, now including updating the ROB and maintaining its data, takes more time in a cycle and risks needing to lengthen cycle time as logic complexity increases.

Figure 2: An updated pipeline which adds the ROB after the writeback stage and separates writeback from the commit logic



Lab4: Datapath of 8-stage PARCv2 OoO Processor, including Scoreboard

```
/*
 | Slot | Valid | Pending | Speculative | Dest (PReg)
 ----------------------------------------------------
 |  0   |       |         |             |           <- HEAD
 |  1   |       |         |             |
 | ...  |  ...  |   ...   |     ...     |    ...    <- TAIL
 | 15   |       |         |             |
*/
```