Computer Architecture
Lab 4: Out-of-Order RISC-V Processor - Reorder Buffer

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

revision: 11-4-2025

In this lab, your task is to design and implement a reorder buffer (ROB) for an out-of-order, single-issue RISC-V processor. We will provide a complete, functional processor framework, minus the reorder buffer, which you will develop. Your objectives include building the reorder buffer, creating appropriate test cases, and submitting a report summarizing your approach and results.

This lab will provide you with experience in:

- The fundamentals of reorder buffers and their role in out-of-order execution.
- How to design and implement in-order commitment logic within an out-of-order processor.

1 Reminder

- **Submission Deadline:** 11/17 (Mon) 11:59 p.m.
- Please refer to Lab 0 for the academic integrity requirements.
- **No sharing or distribution of lab materials is allowed.**

2 Setup

2.1 Getting Started

After obtaining the Lab 4 materials, extract them using the following commands:

```
% tar -xf lab4.tar
% cd lab4
% export LAB4_ROOT=$PWD
```

Inside the lab root directory, you'll find these subdirectories, each serving a specific purpose:

- `build`: Makefile and compiled code.
- `riscvooo`: Out-of-Order RISC-V processor source code (includes a placeholder reorder buffer).
- `tests`: Assembly test build system.
  - `riscv`: RISC-V assembly tests.
  - `scripts`: Utility scripts for the build system.
- `ubmark`: Benchmarks for evaluation.
- `vc`: Additional Verilog components.

Most directories remain unchanged from the previous lab. The new directories are `riscvooo`. It contains a working OoO processor, except that it contains a dummy implementation of the reorder buffer itself.

2.2 Building the Project

As usual, we'll start by compiling the reference processors and running the RISC-V assembly tests:

```
% cd $LAB4_ROOT/tests
% mkdir build && cd build
% ../configure --host=riscv32-unknown-elf
% make && ../convert

% cd $LAB4_ROOT/build
% make
% make check
% make check-asm-riscvooo
% make check-asm-rand-riscvooo
```

Then run the benchmark:

```
% cd $LAB4_ROOT/ubmark
% mkdir build && cd build
% ../configure --host=riscv32-unknown-elf
% make && ../convert

% cd $LAB4_ROOT/build
% make
% make check
% make run-bmark-riscvooo
% make run-bmark-rand-riscvooo
```

3 Out-of-Order Processor Description

The out-of-order (OoO) processor in Lab 4 is derived from the `riscvlong` microarchitecture from Lab 2, so most of the design should feel familiar. As provided, it functions as a simplified I2O2 processor: fetch, decode, and issue occur in order, while execution, writeback, and commit are out of order. Your task is to convert this into an I2OI processor. Key differences from the lecture model include:

- The pipeline includes three pipes: The MUL pipe handles muldiv operations and takes four cycles to execute. The MEM pipe handles memory accesses and takes two cycles to execute. The ALU pipe handles all other operations and takes one cycle to execute.
- Memory writes occur directly in the M stage without the need for a store buffer. This simplifies the design in this lab by ensuring that memory operations execute sequentially and without exception handling.
- Unlike some configurations discussed in lecture, this processor includes a **single** (architectural) register file. Instead of using a physical register file or storing uncommitted data directly in the ROB, we have a dedicated buffer within the datapath to hold data that has been written back but not yet committed.
- The reorder buffer in this lab does not include speculative bits, as speculative execution is not needed.

In this lab, you will work with a reorder buffer containing 16 entries, sufficient for the maximum expected in-flight instructions. The structure of the ROB is illustrated in Table 1.

Each entry in the ROB uses a **valid bit** to indicate if it is active; this bit is set when the entry is allocated and cleared when the entry is committed. The **pending bit** is set at allocation and cleared when the result

Table 1: Reorder Buffer Contents

Entry	Valid	Pending	Physical Register	Data
0	1	1	0	0x00000000
1	1	0	3	0xDEADBEEF
...	...	...	...	...
15	1	1	0	0x00000000

is written back to the ROB. The **physical register bits** store the destination register for the entry in the architectural register file.

Two 4-bit pointers manage the ROB: the **head pointer**, which indicates the next entry to be committed, and the **tail pointer**, which indicates the next available entry for allocation. The tail pointer increments with each new allocation, and the head pointer increments upon commitment. Although capacity issues are unlikely in this lab, ensure boundary checks are in place to stall the processor if the ROB becomes full.

Instruction data is stored in the provided array, `rob_data`, located in the datapath. The ROB generates the control signals for this array.

Finally, speculative bits are not needed in this lab's simplified design. In cases where a conditional branch in the execute stage (X) could impact an instruction in decode (D), the ROB simply skips allocation if the instruction in D is squashed. While this would increase cycle time in a real processor, it is acceptable for this lab's simulation.

4 Building the ROB

First, build the provided I2O2 processor and run the included RISC-V test suites. What do you observe? **Interestingly, even though the I2O2 processor may commit values out of order, it still passes all tests!** This suggests that the test suite is not exhaustive, as you may have noticed in previous labs.

Your first task is to design an assembly test that reveals the flaw in the I2O2 processor's commit behavior. Specifically, construct a scenario where out-of-order commits lead to incorrect program execution, and verify this behavior on the I2O2 processor. (Hint: such a test should be straightforward to create.)

In your report, explain what happens in your test and why it fails. Additionally, discuss why the original I2O2 test suite still passes despite the out-of-order commit issue. (Hint: inspect the assembly code of the provided tests and consider how their structure allows them to succeed.)

Once you've confirmed the need for in-order commit and have created a suitable assembly test (or, ideally, multiple tests), it's time to build the reorder buffer (ROB).

The ROB uses two integers defined in `riscvooo-InstMsg.v`:

- `` `SLOTS `` - The number of ROB slots.
- `` `LOG_S `` - The ceiling of log(`` `SLOTS ``).

The ROB consists of three main port groups, each serving a distinct function:

- **Alloc** - Instructions in the decode stage must be assigned a new ROB entry before they execute.
- **Fill** - Instructions in the writeback stage write their data into the ROB through the fill ports.
- **Commit** - One instruction may be committed per cycle, provided all necessary values are ready.

The ports of the ROB are described below:

- `rob_alloc_req_val`: Indicates if an allocation request is valid in the current cycle.
- `rob_alloc_req_rdy`: Indicates if the ROB is ready to accept an allocation request.

- `rob_alloc_req_preg`: Specifies the physical destination register for the incoming instruction.
- `rob_alloc_resp_slot`: The slot number in the ROB assigned to the new instruction.
- `rob_fill_val`: Indicates if a value is being written back into the ROB during the current cycle.
- `rob_fill_slot`: Specifies the slot being written back during the current cycle.
- `rob_commit_wen`: Indicates if an entry is being committed to the register file in this cycle.
- `rob_commit_slot`: Specifies which slot is committed, used as an index for the `rob_data` array.
- `rob_commit_rf_waddr`: Specifies the physical register address to commit.

Implement the ROB logic by modifying only the `riscvooo-CoreReorderBuffer.v` file. All other files are set up for you, including a scoreboard designed to be compatible with the ROB and bypass values as needed, provided your ROB logic is correctly implemented.

After completing the ROB, update the register file module in `riscvooo-CoreDpath.v` to retrieve values from the ROB instead of directly from the writeback stage. **Then, enable (uncomment) the lines in `riscvooo-CoreScoreboard.v` that allow for bypassing from the ROB.** Finally, rerun the provided test suites to ensure they pass, and confirm that your custom test case also now passes correctly.

5 Testing Methodology

We have created the blank test file in `/lab4/tests/riscv`, you need to fill it and modify `/lab4/tests/riscv/riscv.mk` and `/lab4/build/Makefile`, then, execute, and include in your report and submission at least the following tests:

(Please make sure that each of your tests is less than 30 instructions)

- `riscv-test1.S`: A test that fails due to out-of-order commits but passes with the reorder buffer (ROB) implemented (see Section 4).
- `riscv-test2.S`: A test case requiring a value to be bypassed from the ROB (i.e., bypassing a value that has been written back but not yet committed).
- `riscv-test3.S`: A write-after-write (WAW) scenario that executes correctly on both the original and the final processor. Explain what conditions allow this to occur. (Hint: Consider the time elapsed between the two writes.)
- `riscv-test4.S`: A scenario in which the `riscvlong` processor achieves a higher IPC than the completed `riscvooo` processor. Why might this happen? (Hint: What features are present in `riscvooo` but not in `riscvlong`?)
- `riscv-test5.S`: A test which will use as many ROB slots as your design can.
- Any additional tests you find interesting.

6 Evaluation

This lab includes two parts of evaluation:

- **Performance Comparison:** Compare the `riscvooo` processor with the `riscvlong` processor. For each benchmark, record the total cycle count and IPC (instructions per cycle). In your report, analyze:
  - When and why one design outperforms the other.
  - Situations where performance differences are minimal.
  - Which terms in the Iron Law of Processor Performance are affected.

- **Reorder Buffer (ROB) Analysis:** Explore how performance scales with different ROB sizes. Vary the number of ROB slots (2–16) and measure the cycle count and IPC for `riscv-test5.S`. In your report, discuss:
  - How performance changes as the ROB size increases.
  - The minimum number of ROB slots needed for this lab and your reasoning.

7 Submission

7.1 Lab Report

In addition to the source code for the lab, you would need to submit a lab report that includes the following sections:

- **Introduction/Abstract (1 paragraph max)**: A brief summary of the lab.
- **Design**: Describe your implementation, justify design decisions (if any), note deviations from the prescribed datapath, and provide a balanced discussion on what and why you implemented specific features.
- **Testing Methodology**: Describe your testing strategy and how you handled corner cases.
- **Evaluation**: Report your simulation results and cycle counts comparing `riscvlong` and `riscvooo`, and your simulation results and cycle counts comparing `riscvooo` with different number of slots.
- **Discussion**: Compare and analyze benchmark results
- **Figures**: Include a diagram of your final ROB implementation and any other relevant design illustrations.

Avoid scanning hand-drawn figures or using digital photos of them (if you do this, your report will get zero points). The lab report should be clear and professional. The report must not exceed **4 pages** (excluding figures). Penalties will be imposed for exceeding this limit.

Submit your lab report as a single PDF file named `your_student_id-lab4-report.pdf`.

7.2 Modified Files

You will modify the following files for this lab:

- `riscvooo-CoreDpath.v`, `riscvooo-CoreScoreboard.v`: Make minor edits as indicated by the comments in the source code.
- `riscvooo-CoreReorderBuffer.v`: Implement your reorder buffer design here.
- `riscvooo-InstMsg.v`: Adjust the parameter that specifies the number of ROB slots.
- `riscv-test1.S` to `riscv-test5.S`: See Section 5.

7.3 Deliverables

Submit a `.tar.gz` file of your working directory, keeping the original structure intact. All source files should be in `$LAB4_ROOT/riscvooo/` or in the `$LAB4_ROOT/test/` directory. Be sure to clean up generated waveforms or compiled code by running the following commands:

```
% cd $LAB4_ROOT/build
% make clean
% cd $LAB4_ROOT/tests
% rm -rf build
% cd $LAB4_ROOT/ubmark
% rm -rf build
```

Create a tarball of your completed lab with the following commands:

```
% cd $LAB4_ROOT
% cd ..
% tar -cvzf student_id-lab4.tar.gz lab4
```

7.4 Submission Instructions

- Keep your code in the `lab4` folder. If the code is not in this tarball, we cannot grade it.
- Submit the tarball and lab report separately via e3.
  - `student_id-lab4-report.pdf`
  - `student_id-lab4.tar.gz`

8 Grading Rubric

- **Report (30%)**
  - Evaluation (15%)
  - Figures (5%)
  - Others (10%)
- **Code (65%)**
  - ROB
- **Test case(5%)**
  - `riscv-test1.S` to `riscv-test5.S`

9 Tips

- Develop incrementally—code and test small sections at a time to avoid overwhelming debugging.
- Always draw the hardware design before coding, ensuring clear interaction between control logic and the datapath.
- Modify the control unit as needed—it's a starting point for your own logic and signals.
- Start early and run simulations regularly to catch issues early in the process.
- If things don't fully work, clearly document your progress and debugging efforts in the lab report.