Computer Architecture
Lab 3: Superscalar RISC-V Processor

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

revision: 10-13-2025

In this lab, you will design and implement a two-wide superscalar in-order processor. We provide a functional datapath, and your task will be to add the necessary control logic, instruction steering mechanism, and a scoreboard to enable the processor to execute two instructions per cycle efficiently.

This lab will provide you with experience in:

- The principles of superscalar processor design and multi-issue pipelines.
- How to implement control logic and instruction steering for in-order execution.
- The use of a scoreboard to manage dependencies and maximize instruction throughput.

1 Reminder

- **Submission Deadline:** 11/3 (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

Once you have the Lab 3 materials, extract them by entering the following commands:

```
% tar -xf lab3.tar
% cd lab3
% export LAB3_ROOT=$PWD
```

Within the lab root directory, you will find eight subdirectories, each serving a specific purpose:

- `build`: Contains the Makefile and compiled code.
- `imuldiv`: Integer multiply/divide unit
- `riscvdualfetch`: Superscalar RISC-V processor source code with dual-fetch, but single instruction issue
- `riscvssc`: Superscalar RISC-V processor source code
- `tests`: Assembly test build system
  - `riscv`: RISC-V assembly tests
  - `scripts`: miscellaneous scripts for build system

- `ubmark`: Benchmarks for evaluation
- `vc`: Contains additional Verilog components.

Most directories remain unchanged from the previous lab. The new additions are `riscvdualfetch` and `riscvssc`. The `riscvdualfetch` directory contains the initial datapath for a two-wide superscalar processor, lacking only the scoreboard, steering logic, and control logic.

2.2 Building the Project

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

```
% cd $LAB3_ROOT/tests
% mkdir build && cd build
% ../configure --host=riscv32-unknown-elf
% make
% ../convert
% cd $LAB3_ROOT/build
% make
% make check
% make check-asm-riscvdualfetch
% make check-asm-riscvssc
```

The methodology remains the same as in Lab 2. For details on the test suite, refer to the Lab 2 handout.

Note: When you first run `make` in `$LAB3_ROOT/build`, you'll encounter errors because the control unit in `riscvdualfetch` is incomplete. You'll need to make modifications before compiling and running tests. However, the checks can be performed on other designs.

3 2-Wide Superscalar In-order Processor Overview

In this lab, you are provided with a 7-stage, two-wide superscalar in-order processor, lacking the control logic, scoreboard, and instruction steering logic. Each stage of the processor can handle up to two instructions simultaneously. The execute stage consists of two heterogeneous functional units:

- **Pipeline A**: Capable of executing all ALU, memory, control flow, and multiply/divide operations. It includes a four-stage multiply/divide unit.
- **Pipeline B**: Limited to simple ALU operations and cannot handle multiply, branch, jump, divide, or memory instructions.

Below is a high-level description of the processor pipeline, followed by instructions for the two parts of this lab.

Fetch Stage

The fetch stage should retrieve two instructions per cycle when not stalled. We provide a triple-ported memory unit, replacing the dual-ported one from the previous lab. This allows up to three memory accesses (two instructions, one data) per cycle, preventing structural hazards.

For simplicity, we assume all memory accesses are properly aligned, and the memory unit will handle all requests accordingly. However, real systems would need to account for misaligned instructions, which could lead to issues, as discussed in class.

Decode/Issue Stage

In lecture, we discussed the I4 and I2OI processor architectures, which separate the decode and issue stages. For continuity with the previous lab, however, we will combine them into a single decode/issue stage in this lab. This simplifies your task by not requiring a rewrite of the logic from the previous lab. Keep in mind that, in a real processor, these stages would likely be separated to maintain reasonable cycle times.

Register File

To minimize structural hazards, we provide a six-ported register file, allowing four reads (two for each pipeline) and two writes per cycle. The register file itself does not detect data hazards, and writing to the same register has undefined behavior. Your stalling and bypassing logic must ensure correct register values are maintained and avoid overwriting errors.

Steering Logic

The decode logic will be similar to the previous lab, but since the two execute pipelines are functionally heterogeneous, you'll need to add steering logic to correctly assign instructions to each pipeline. Only simple-ALU instructions can be issued to the second (B) pipeline. If two instructions that cannot be paired due to pipeline constraints are decoded together, the second instruction must stall for one cycle before being sent down the correct pipeline.

Steering Logic			
Instruction 0 Type	Instruction 1 Type	Instruction 0 Dest.	Instruction 1 Dest.
ALU	ALU	A	B
ALU	Non-ALU	B	A
Non-ALU	ALU	A	B
Non-ALU	Non-ALU	A	Stall, then A

The steering logic should decode and issue two instructions per cycle if no structural hazards exist. If the second instruction depends on the first, they cannot issue simultaneously. **Additionally, if two fetched instructions target the same destination register, the second instruction must stall to prevent simultaneous writes to the register file.**

Scoreboard

You will need to implement a scoreboard, which will be read during the decode/issue stage. Its purpose is to ensure that all input registers for instructions at this stage are ready to be read, and to determine where to bypass data values from (pipeline and stage). Since all instructions write to the register file at the end of the pipeline, the scoreboard does not need to track writeback hazards. Instead, it will handle bypassing and data stall information, providing a more compact solution than the stall logic used in Lab 2.

	Scoreboard						
Dest. Reg.	Pending (Y/N)	Functional Unit (Pipeline A/B)	4	3	2	1	0
x0	1	0	0	1	0	0	0
x1	0	1	0	0	0	0	0
...	...		.	.	.	.	.

Execute Stage

The execute stage is divided into two parallel pipelines. Pipeline A is fully functional and identical to the `riscvlong` execute stages, while Pipeline B only handles simple ALU operations, without mul/div or memory units.

Memory (X1) Stage

The X1 stage serves as the memory stage, similar to previous labs. For this lab, we provide a triple-ported memory unit, allowing multiple accesses per cycle.

X2/X3 Stage

Stages X2 and X3 are included for timing purposes. Although Pipeline B does not have a mul/div unit, both pipelines contain these stages since we are still implementing an in-order processor.

Writeback Stage

The writeback stage remains the same as in the previous lab. As before, the register file is equipped with two write ports, preventing structural hazards related to write port availability.

3.1 Part 1: Two-Wide Superscalar Processor with Dual Fetch, but Single Issue

In `riscvdualfetch`, we provide a datapath with most of the `riscvlong` datapath logic duplicated. For **Part 1**, much of the control unit logic has been removed. **Your task in Part 1 is to implement a processor that fetches two instructions per cycle and steers them down the pipeline one at a time.** The processor should alternate between the two fetched instructions. Start by getting basic instructions flowing through the pipeline.

You do not need to handle steering issues caused by pipeline heterogeneity or instruction dependencies yet. These topics will be covered in **Part 2**. Test the pipeline incrementally. In the dual-fetch, single-issue design, branches and jumps must kill both instructions, including the second fetched instruction.

You may use the bypassing and stalling logic from previous labs, or employ a scoreboard to detect data dependencies. A scoreboard is only required in **Part 2**. We have provided `riscvdualfetch-CoreScoreboard.v` as a starting spoint for your scoreboard implementation. **Unlike other modules, you may modify its port declarations if needed.**

To avoid executing the same instruction twice, consider injecting an invalid instruction after the first instruction is issued. Be careful when using the `nop` instruction, as it is implemented as an `addi` in RISC-V.

Key changes to the processor:

- The `pc_plus4_*hl` signals are replaced with `pc_plus8_*hl` to account for fetching two instructions per cycle.
- The register file now has six ports.
- The memory unit has an additional read port.
- The `br_targ`, `j_targ`, and `jr` signals must be de-multiplexed from the two decoded instructions as they are issued.
- If the first decoded instruction is a taken branch or jump, the second instruction (stalling in the decode stage) must be killed.
- The pipeline diagrams for debugging are drawn by code at the end of `riscvdualfetch-sim.v` and `riscvdualfetch-randdelay-sim.v`. These have been updated to display all seven stages of both pipelines in parallel. Ensure your signal names match the debugger to avoid compilation errors, or update the debugger to match your signals.

For this part, you're only allowed to modify the following files:

- `riscvdualfetch-CoreCtrl.v`
- `riscvdualfetch-CoreScoreboard.v` (optional)

3.2 Part 2: Two-Wide Superscalar Processor

In this section, the main objective is to enable the processor to issue and execute two instructions simultaneously. This requires implementing full instruction steering logic and a scoreboard. The steering logic should be integrated into the decode/issue stage to ensure that instructions are issued to the appropriate pipelines.

Additionally, the steering logic must handle RAW dependencies between the two fetched instructions. If two instructions attempt to write to the same register, the control logic should stall the second instruction.

To begin, run the setup.sh in riscvssc:

```
% cd $LAB3_ROOT/riscvssc
% ./setup.sh
```

The setup.sh script copies files from `../riscvdualfetch`, renames any filenames (and relevant build entries) replacing the riscvdualfetch prefix with riscvssc, and performs basic build updates.

Rebuild your project to verify everything has been copied correctly:

```
% cd $LAB3_ROOT/build
% make
```

Next, implement the steering logic, ensuring that your processor properly handles two non-ALU instructions decoded simultaneously. Run all the tests to ensure correct execution.

To simplify the design, if two instructions in a pair write to the same register, stall the second instruction. This avoids conflicts and simplifies bypass calculations.

Lastly, modify the non-synthesizable code at the bottom of the `*-CoreCtrl.v` files, which tracks the number of executed instructions. Update the condition `num_inst = num_inst + 1;` to accurately reflect the superscalar width.

For the report, create at least one new assembly test to demonstrate that your processor correctly detects a WAW hazard in simultaneously decoded instructions and continues to issue and execute correctly. Provide a detailed explanation of your test.

Additionally, create another assembly test to show that the processor handles two non-ALU instructions decoded at the same time. Explain how your test demonstrates this functionality.

For this part, you're only allowed to modify the following files:

- `riscvssc-CoreCtrl.v`
- `riscvssc-CoreScoreboard.v`

Once your processor passes all existing and custom tests, congratulations!

4 Testing Methodology

We will provide the RISC-V assembly tests and benchmarks from the previous lab. **You will need to create additional assembly tests to verify the correctness of your processor, as outlined in the previous sections.** For guidance on creating these tests, refer to the handout from the previous lab.

5 Evaluation

In this lab, you will compare the performance of the `riscvdualfetch` and `riscvssc` processors against the `riscvlong` processor. Report both the cycle count and the IPC (Instructions Per Cycle) for each benchmark across all three microarchitectures. In your report, analyze the performance differences and discuss the scenarios where each modification is beneficial, harmful, or has a limited impact. Also, consider which parts of the Iron Law of Processor Performance are affected.

6 Submission

6.1 Lab Report

In addition to the source code, you must 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 simulation results and cycle counts for `riscvdualfetch`, `riscvssc`, and `riscvlong`.
- **Discussion**: Compare and analyze benchmark results, and discuss tradeoffs of dual-fetch and dual-issue designs.
- **Figures**:
  - Datapath diagram of heterogeneous dual-issue superscalar I4 processor
  - Diagram of the implemented scoreboard

Avoid scanning hand-drawn figures or using digital photos of them. 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 `student_id-lab3-report.pdf`.

6.2 Deliverables

Submit a `.tar.gz` file of your working directory, keeping the original structure intact. All source files should be in `$LAB3_ROOT/riscvdualfetch` and `$LAB3_ROOT/riscvssc`. Be sure to clean up generated waveforms or compiled code by running the following commands:

```
% cd $LAB3_ROOT/build
% make clean
% cd $LAB3_ROOT/tests
% rm -rf build
% cd $LAB3_ROOT/ubmark
% rm -rf build
```

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

```
% cd $LAB3_ROOT
% cd ..
% tar -cvzf student_id-lab3.tar.gz lab3
```

6.3 Submission Instructions

- Keep your code in the `lab3` folder. If the code is not in this tarball, we cannot grade it.
- Submit the tarball and lab report separately via e3:
  - student_id-lab3-report.pdf
  - student_id-lab3.tar.gz

7 Grading Rubric

- **Report (30%)**
  - Datapath diagram of heterogeneous dual-issue superscalar I4 processor (10%)
  - Scoreboard diagram (5%)
  - Others (15%)
- **Code (70%)**
  - Part 1 (30%)
    - Functional correctness
  - Part 2 (40%)
    - Functional correctness (30%)
    - Performance score (10%)

8 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.