OoO RISC-V Processor Project Specifications
Fall 2024 (Honors 189 Section)

Overview

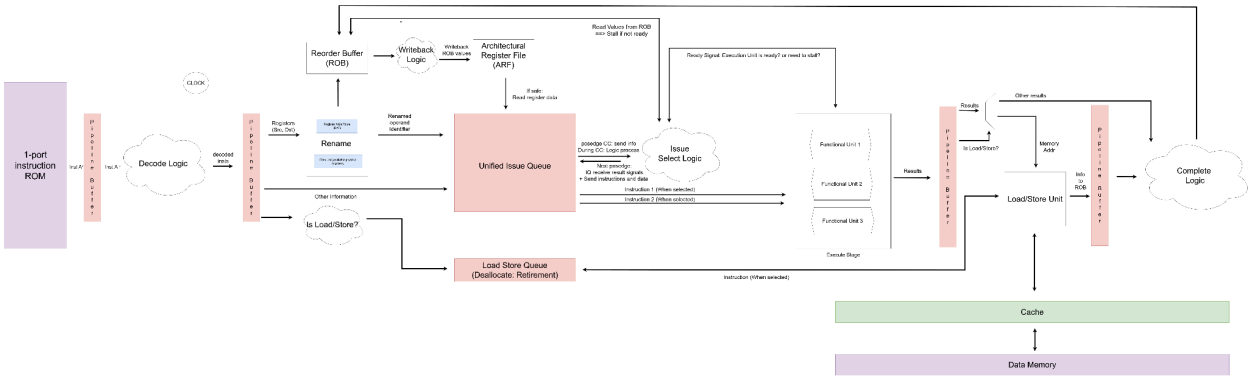
The goal of this project is to design a simplified 2-issue out-of-order (OoO) processor in RTL. Preferably, the code should be synthesizable but we will only check your design through simulation (Questa/ModelSim). Verilog is the preferred language for the design but you are free to use any other HDL including VHDL and/or SystemVerilog.

The following will provide details of the microarchitecture and configurations. The project can be done in groups (a separate Google Sheet has already been provided for creating the groups).

The due date is the Friday of Finals Week. You will have one design checkpoint in Week 8 where you need to show some basic functionality of your code (details will be discussed in the class).

Microarchitecture

The architecture may vary, but your design should closely follow the microarchitecture shown below. This document will be updated as changes to descriptions and requirements are made.



Link to the detailed version:
https://drive.google.com/file/d/1tEQZ8QjoPTSgt0RAT1iJprvz4u0qWs8C/view?usp=sharing

ECE M116C - CS M151B Computer Architecture Systems - UCLA

Specifications

- Specifications
  - Instruction Size: 32-bit
  - Standard ALU instructions (No fp operations)
  - 2-Issue Processor

- Components

Components	Parameter/Standard	Name	(Default) Specs
Issue Queue (IQ)	Parameter	Size	64 instructions
		Issue Ports	2 issues
Reorder Buffer (ROB)	Parameter	Size	64 values
		Commit Ports	1 commit
Architectural Register File	Standard	Number of Regs	32 registers (each 32 bit)
Load-Store Queue (LSQ)	Parameter	Size	16 instructions
Rename Table and Free Pool	Parameter	No.Entries	32
Cache (L1d)	Parameter	Size	32kB
		Cache Hit	1 CC
		Cache Write	1 CC
	Standard	Type	4-Way Set Assoc /Direct Mapped
		Eviction Policy	Random
		Write Policy	Writethrough
Memory	Parameter	Read Latency	10 CC
		Write Latency	10 CC
	Standard	Type	Always Hit

Functional Units	Parameter	Number of Units	3

- Notes on current arch:
  - No BPU
  - Only 1 Fetch per CC, but multiple issues per CC for superscalar
  - Assume: no stalling at the dispatch-rename stage
    - Stalling can happen at the Issue stage assuming the issue queue is small enough
  - No compressed instructions

Instructions to be implemented:

A small set of R-type, I-type, and Memory instructions.

```
ADD, ADDI, LUI, ORI, XOR, SRAI
LB, LW, SB, SW
```

Your design will not need to handle branch or jump instructions hence you don't need to support flushing and/or stalling.

Assumptions:

You can safely assume that your PC is always PC+4. You are not responsible for the correct implementation of other instructions, however, you should do your best that other instructions are harmless (converted to NOP). You are not responsible for memory hazards in this design. Please refer to the table shown above about the size of each unit (number of registers, issue queue, etc.).

You will be given a HEX file (similar to CA1) that should be loaded into your instruction memory. Each line in the HEX file is one byte thus four lines become one instruction. We will provide debug traces but it is strongly recommended to create your own testbenches and HEX files.

As mentioned earlier, your code doesn't need to be synthesizable but do your best to follow the best practices when writing an HDL code.

Setup:

You should use ModelSim/Questasim (with Quartus) to simulate your code. Here are a few helpful links to install and learn the tools:

1. Windows Quartus Installation
2. macOS Quartus Installation

Alternatively, you could also use other tools such as Xilinx Vivado.

Modules:

Top Level:

This module should instantiate all of your pipeline buffers and contain modules for your pipeline stages, along with any additional logic you find helpful. Importantly, your top-level is responsible for communication between stages such as freeing up the free pool and waking up/forwarding to registers in the reservation station.

While the diagram contains each pipeline's stage logic cleanly in one box for organization, you may find it helpful to contain some stages' logic in the top-level directly.

Instruction Fetch:

IF module should use the program counter to read one instruction per cycle from the one-port instruction ROM (READ-ONLY MEMORY) sequentially. If there are no more instructions left, you should have some sort of signal or logic to stop fetching instructions.

Decode:

Decode logic should take one instruction per cycle and using the RISC-V ISA specification, store all important information from the bits into signals representing source registers, destination registers, ALU OP, lw/sw flags, and any control signals your design needs.

Rename:

In this architecture of out-of-order execution, we make use of register renaming to avoid data hazards. We should keep a register alias table (RAT) that maps instruction registers to tags – we will update this as we rename registers. We must also keep a free pool/list.

Unified Issue Queue:

This can be thought of as a very large table with each row corresponding to an instruction. It contains all the decoded information such as control signals, renamed source and destination registers, as well as register source values (you must read from register files during this pipeline stage) in this architecture. It should also have bits corresponding to whether the source registers and ALUs that the instruction depends on are ready or not. Every cycle, up to three instructions can be issued from RS if they map to different ALUs and all the RSs are ready. Note that from the complete stage, we will have outputs ready from certain instructions and can update register data values in the RS (forwarding/wake-up) rather than waiting for those instructions to retire and be written to registers. You should deal with this forwarding logic before determining which instructions to issue every cycle.

Issue/Fire:

The selector logic should used to select (wake-up) instructions that can be issued. The issue logic should read information from the queue and ROB to decide which instructions to wake up. Up to three instructions can be scheduled.

Complete Stage:

When an operation is finished with the issue stage, we have the outputs from ALU ready. This means we can update the values of register data that the results will be stored in the Reservation Station and "wake-up" that row by marking those register sources as ready. However, in OoO, we want to avoid data and memory hazards, so we use something called a Reorder Buffer (ROB) to retire our instructions in order and cannot write back to register files or memory yet.

Retire:

We can retire two instructions per cycle from the ROB. We must retire instructions in order that a programmer intended for them to be executed–we do this by ensuring the ordering in the ROB is enforced by the dispatch stage reserving rows for instructions and then retiring in order from the ROB. The retire involves memory and register writes and clearing the ROB row corresponding to the instruction. Lastly, a retire should free up the physical register that was used as the destination for the rename stage's free pool.

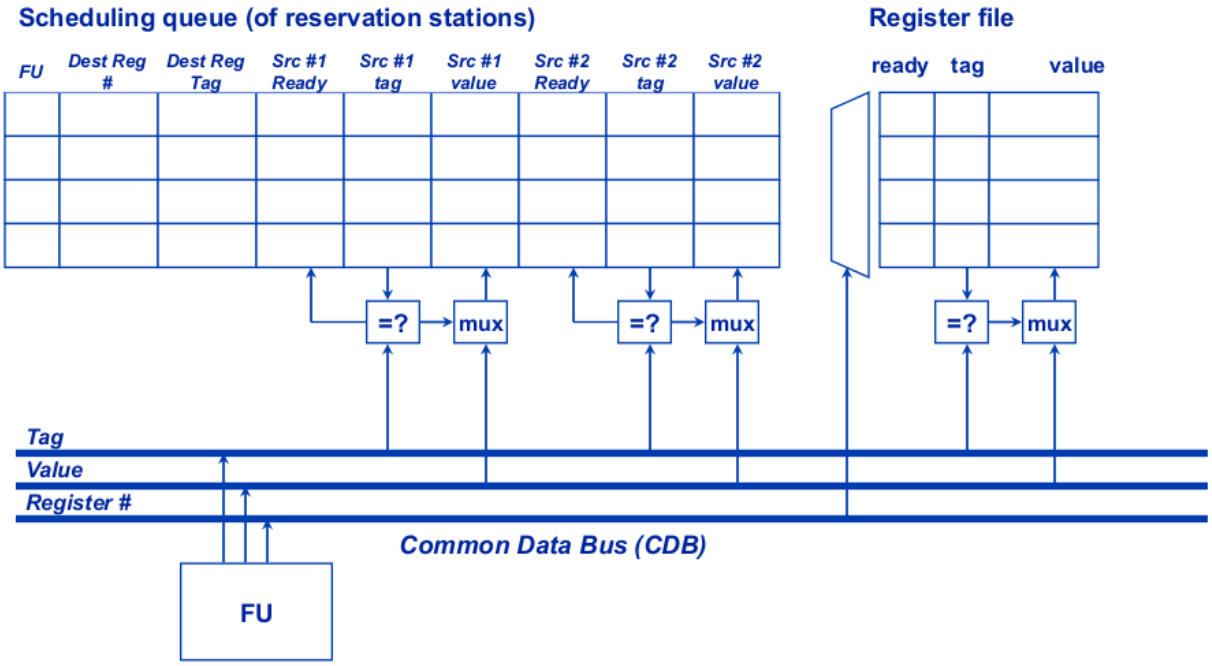
Cache:

Optionally, your design can support cache as well. The details are provided in the table above. However, this part is considered a bonus. You can also assume that no cache exists so all your requests go directly to the memory.

More Details:

This part will be completed as we move forward. Start early and ask questions. We will add details based on the discussions in the class.

Here is an example of how to implement your reservation station and forwarding.



What to turn in

In W8, you should turn in a 30-50% completed project. This should include several components of your design, and you should be able to demo it using your test benches (e.g., you can successfully load a file, fetch and decode it, rename it, and read correct registers. Alternatively, you can show your reservation station and ROB).

On Friday of Exam week, you should turn in your full design. We will provide a test file that you should use to test your design (see below). The expectation is that your simulator prints the correct results.

You should turn in all of your Verilog files (only send us your .v files) and a short video explaining your design and showing that it is outputting the correct values.

ECE M116C - CS M151B Computer Architecture Systems - UCLA

Traces:

The link to r-type and test traces can be found here:
https://drive.google.com/drive/folders/17I15p0aRZhZZHpXLzvgPBvp7FgGfaim1?usp=sharing

Start with r-type first! Further, if you need to create more unique test benches, use this script:
https://drive.google.com/file/d/1-TZuWEsjBLTpviLVKNiQ_iXwlYfozuoB/view?usp=sharing
(credits to Tyler Price for creating this!)