**ELC 5313: Advanced Computer Architecture**

**Fall 2025**

**Lab 3: Register File and Instruction Queue**

**Introduction and Scope**

In this lab, you will begin implementing the front-end of an out-of-order execution pipeline. Specifically, you will complete two key components:

1. **Instruction Queue (IQ)** – manages instruction fetching, issuing, execution tracking, and completion.
2. **Register File (Registers)** – provides register read and write functionality while supporting register renaming for out-of-order execution.

Together, these modules form the foundation for Tomasulo’s Algorithm — allowing instructions to issue and execute dynamically while preserving data dependencies.

In a basic in-order processor, each instruction must wait for previous instructions to complete before executing. Tomasulo’s algorithm removes this bottleneck by introducing:

* **Reservation stations (RS):** temporary storage for instructions waiting for operands or execution units.
* **Register renaming:** eliminates false data dependencies by mapping architectural registers to reservation stations.
* **Instruction queue:** holds instructions as they move from fetch to issue and beyond.

The **instruction queue** manages when instructions are fetched, when they are ready to issue to available reservation stations, and when they can be removed after execution completes.  
The **register file** is responsible for tracking the latest producer (reservation station) for each register and providing operands when they are ready.

**Schedule**

* **Week 1 (today):**
  + Register module completion
* **Week 2:**
  + Instruction Queue module completion

You are welcome to work ahead.

**Provided Files**

* registers.sv
* resgisters\_tb.sv
* .instruction\_queue.sv
* Instruction\_queue\_tb.sv

**Module 1: Registers**

**Big Picture**

This module is similar to the *register file* from Computer Organization but extended for out-of-order execution.

1. **Reading operands:**  
   Each instruction can read two source registers (A\_address and B\_address). If a register’s value is ready, it is output directly. If it’s not ready because a reservation station is still computing it, the module outputs the reservation station number instead and marks the output as invalid.
2. **Deferring writes:**  
   Unlike the in-order register file, the destination register is not updated immediately. Instead, the register file stores the *reservation station number* that will eventually produce that register’s new value.
3. **Updating after execution:**  
   When a reservation station completes, it writes back the produced value to any registers that were waiting on it.

**Tasks**

**1. Operand Fetch (Negative Clock Edge)**

**2. Register Update (Positive Clock Edge)**

**Expected Behavior**

When simulated:

* Instructions that read from ready registers output their actual values.
* Instructions waiting on data from reservation stations output the RS identifiers.
* Once the producing RS completes and writes back, the register file automatically updates with the new value.

A screenshot of a computer

AI-generated content may be incorrect.

**Module 2: Instruction Queue**

**Big Picture**

The **instruction queue** controls instruction flow between the fetch stage and the reservation stations.  
It tracks instruction status through multiple stages:

| **Stage** | **Description** |
| --- | --- |
| Fetch | Instruction is loaded from memory into the queue |
| Issue | Instruction is sent to a reservation station |
| Execute | Instruction is actively running in an execution unit |
| Write Back | Instruction has completed and can retire |

**Tasks**

**1. Fetch Block**

**2. Issue Block**

**3. Issue Completion Block**

**4. Execute Start Block**

**5. Write Back Block**

**6. Instruction Removal Block**

**Expected Behavior**

When simulated:

* Instructions will continuously fetch from memory into the queue.
* When the adder is available, one instruction will issue each cycle.
* As reservation stations confirm execution, instructions update their status and are eventually removed.
* The queue contents should dynamically reflect instruction flow through fetch → issue → execute → write-back → retire.

A screenshot of a computer program

AI-generated content may be incorrect.

**Deliverables**

1. registers.sv – completed module.
2. Instruction\_queue.sv – completed module.
3. Screenshots of each waveform simulation showing correct register redirection and instruction flow, and status updates. Separate waveforms for each module.

**Grading Rubric**

* **Register Module Functional** (35 pts)
  + Correct implementation of operand fetch, redirection logic, and register updates.
* **Instruction Queue Module** (35 pts)
  + Correct implementation of fetch, issue, execute, write-back, and instruction removal.
* **Successful simulation & waveform** (20 pts)
  + Demonstrates proper behavior of both modules with clear waveform evidence; shows correct data flow, redirection, and instruction status update.
* **Code clarity & comments** (10 pts)
  + Descriptive names and comments explaining design choices.

**Due Date**

**10/22/2025** (Submit individually via Canvas). You can work in pairs for this lab.