Project Scope and Deliverables' Specifications
Pipelining with Hazard Detection
CS2323 Computer Architecture

Anant Maheshwary (CS24BTECH11006)

October, 2025

1 Introduction

The topic chosen for the final project is "**Extend the in-house simulator to support pipelining and handle various hazards**," built for assembling a **RISCV-64I** assembly program. The following report discusses the scope of the project, the implementation steps and the verification plan for the project.

2 Project Scope

The primary objective of this project is to extend the existing in-house single-cycle processor simulator to support five-stage pipelining.

2.1 Processor Modes

The simulator will be configurable, allowing the user to enable or disable specific features related to pipelining and hazard management via a `.config` file or using command-line options.

The project aims to create a simulator capable of operating in the following modes:

1. **Mode 0: Single-Cycle (Baseline)**
   The existing implementation, which will serve as the reference for verifying the functional correctness of all other modes.

2. **Mode 1: Pipelining without Hazard Detection**
   A five-stage pipeline (IF, ID, EX, MEM, WB) will be implemented. This mode will not include any hazard detection or forwarding logic. It is expected to produce incorrect results for programs with data or control hazards.

3. **Mode 2: Pipelining with Hazard Detection (managed using only Stalling)**
   This mode will add a Hazard Detection Unit to detect data and control hazards. The hazards will be handled by flushing the pipeline by inserting bubbles (`nops`) until the dependency is resolved or the branch outcome is determined.

4. **Mode 3: Pipelining with Forwarding and Stalling**
   This mode will introduce a Forwarding/Bypassing Unit. It will try to reduce stall cycles in data hazards by forwarding results directly from earlier stages.

5. **Mode 4: Pipelining with Forwarding and Static Branch Prediction**
   This mode builds on the previous one by adding a simple static branch prediction scheme. The default strategy will be "predict not-taken." If the prediction is wrong, the pipeline will be flushed and the correct path will be fetched.

Optional (if time permits):

5. **Mode 5: Pipelining with Forwarding and 1-Bit Dynamic Branch Prediction**
   This mode implements a simple dynamic branch predictor using a **Branch History Table (BHT)** with 1-bit counters.

6. **Mode 6: Pipelining with Forwarding and 2-Bit Dynamic Branch Prediction**

2.2 Performance Evaluation

After the execution of a program with any of the modes mentioned above, the simulator will print a summary report including:

1. The number of data hazards mitigated.
2. The number of control hazards mitigated.
3. The number of stall cycles introduced.
4. Total clock cycles taken and the CPI (Cycles per Instruction).

2.3 Step-by-Step Debugging + Visualization - *Optional (if time permits):*

The debugging part of the simulator can be extended to show which instruction is in which pipelining stage at a given point in time as the user steps through the instructions one-by-one.
Note: *The above can be partially implemented for the earlier, simpler modes as per time constraints.*

3 Implementation Steps

3.1 Refactoring the Existing Simulator

Currently, the `main.cpp` file is the entry point for the simulator. It parses CLI arguments, initializes the CPU and memory, loads the program and starts the simulation loop.

Changes that need to be made:

- `main.cpp` should have CLI checks for the different configuration options and to assemble the program accordingly.
- An additional config file needs to be created to store the config information for all the modes mentioned.
- The main simulation loop has to be changed to support independent clock cycles and simultaneous instruction execution instead of the usual sequential execution after instruction completion.

3.2 Building the Multi-Cycle Data Path

- A separate data path has to be built for each of the 5 pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Write Back (WB).
- This involves creating files for the four pipeline registers along with read/write functionality and the ability to store information like the Program Counter address and Instruction fetched.
- For the load instruction, an additional write control port is required in the register file which passes the destination register address to the Register File along with the write data after the WB stage.

3.3 Adding the Pipeline Control

A file needs to be created for simulating the control unit that provides control signals for the correct corresponding instructions to the pipeline registers and propagates data through them.

3.4 Implement Hazard Detection and Forwarding

- **Hazard Detection Unit:** Add logic to detect data and control hazards. This unit will monitor pipeline registers and stall or flush the pipeline as needed.
- **Forwarding Unit:** Implement data forwarding/bypassing logic to resolve data hazards without stalling, by routing results from later stages back to earlier ones when possible. Some sort of code is needed to simulate the role of the multiplexers deciding which input will be received by the ALU in the EX stage.

3.5 Branch Prediction Logic

- **Static Prediction:** Modify the IF stage to fetch the next instruction immediately after fetching a branch instruction. This simulates the role of an earlier comparator. Execute normally if branch taken and flush otherwise.
- **Dynamic Branch Prediction:** Implement the Branch History Table (BHT) as a hash table with the address of the branch instructions as the keys. Maintain states accordingly for 1-bit or 2-bit branch prediction.

4 Verification Plan

For all testing purposes, the base (Mode-0) simulator will be used as the reference and the final register, program counter and memory states will be compared.

The autotests will fall into 3 main categories:

1. **Unit Tests:** Will check one instruction at a time. A separate test will be created for every instruction, covering all possible outcomes (e.g., both true and false for branch instructions).

2. **Integration Tests:** Will check how multiple instructions work together. Simulator will be tested over a large set of sequences of two to four instructions. This will verify most hazard detection and forwarding cases.

3. **Edge Cases:** Tricky tests built specifically for unusual scenarios. Examples include using `x0` as register for hazard detection, double data hazards, data hazard for branches, etc.

5 Conclusion

The simulator will be used to run different assembly programs written throughout the course and the performance metrics across different modes will be compared.