Experiment 4

Pipelined RISC-V Processor

Objective

The objective of this lab is to implement a three-stage pipelined RISC-V processor based on the single-cycle RV32I architecture. By the end of this lab, you will:

- Understand the differences between single-cycle and pipelined processors.
- Modify a single-cycle RV32I datapath into a three-stage pipelined datapath.
- Identify and resolve data and control hazards in the pipeline.
- Evaluate the performance improvement achieved by pipelining.

Introduction

Pipelining is a technique used to improve the throughput of a processor by overlapping the execution of multiple instructions. While it does not reduce the time-to-completion for a single instruction, it allows the processor to handle multiple instructions simultaneously, increasing overall efficiency. In this lab, you will transform a single-cycle RV32I processor into a three-stage pipelined processor, consisting of the following stages:

- **Fetch**: Instruction fetch from memory.
- **Decode and Execute**: Decode the instruction and perform the required computation.
- **Memory and Writeback**: Access memory (if needed) and write results back to the register file.

Single-Cycle to Three-Stage Pipeline

To convert a single-cycle processor into a three-stage pipelined architecture, the single-cycle datapath must be divided into three distinct stages. Each stage is separated by pipeline registers, which hold intermediate results and control signals for the next stage. The key modifications include:

- Introducing pipeline registers between stages to store intermediate data and control signals.
- Splitting the single-cycle datapath into three stages:

  - **Fetch Stage**: Fetch the instruction from memory and update the Program Counter (PC).
  - **Decode and Execute Stage**: Decode the instruction, read registers, and perform the ALU operation.
  - **Memory and Writeback Stage**: Access memory (for load/store instructions) and write results back to the register file.
- Ensuring that all signals required by subsequent stages are passed through pipeline registers.

Pipeline Registers

Pipeline registers are critical for maintaining the state of the processor between stages. The following registers are required:

- **PC Register**: Stores the updated Program Counter value after the fetch stage.
- **Instruction Register (IR)**: Stores the fetched instruction for the decode stage.
- **Control Signals Register**: Stores control signals generated during the decode stage for use in later stages.
- **ALU Result Register**: Stores the result of the ALU operation for the memory/writeback stage.
- **Memory Data Register**: Stores data read from memory for the writeback stage.

Resolving Hazards

Pipelining introduces the possibility of hazards, which occur when instructions depend on the results of previous instructions that are still in the pipeline. Hazards can be classified into two types:

- **Data Hazards**: Occur when an instruction depends on the result of a previous instruction that has not yet been written back to the register file.
- **Control Hazards**: Occur due to branch or jump instructions, where the target address is not known until the execute stage.

Hazard Resolution Techniques

To resolve hazards, the following techniques can be used:

- **Forwarding (Bypassing)**: Forward the result of an instruction directly from the pipeline register to the dependent instruction, bypassing the writeback stage.
- **Stalling**: Halt the pipeline until the required data is available. This is done by inserting ”bubbles” into the pipeline.

- **Flushing**: Discard instructions fetched after a branch or jump instruction until the target address is known.

Implementation Steps

Follow these steps to implement the three-stage pipelined processor:

Step 1: Modify the Datapath

- Split the single-cycle datapath into three stages: Fetch, Decode/Execute, and Memory/Writeback.
- Insert pipeline registers between each stage to store intermediate results and control signals.
- Ensure that all signals required by subsequent stages are passed through the pipeline registers.

Step 2: Implement Hazard Detection and Resolution

- Add logic to detect data hazards by comparing the source registers of the current instruction with the destination registers of previous instructions.
- Implement forwarding logic to resolve data hazards by forwarding results from the pipeline registers.
- Add stall logic to handle cases where forwarding is not possible.
- Implement flushing logic to handle control hazards caused by branch and jump instructions.

Step 3: Test the Pipeline

- Write test programs to verify the correctness of the pipeline.
- Test for data hazards by creating sequences of dependent instructions.
- Test for control hazards by including branch and jump instructions in the test program.
- Verify that the pipeline produces the correct results and handles hazards appropriately.

Deliverables

- System-verilog implementation of the three-stage pipelined processor.
- Drawio diagram of the pipelined processor along with control instructions.
- Testbench and test programs to verify the functionality of the pipeline.
- FPGA implementation of the pipeline. Display PC and final register and memory data (whichever got updated with the result) on the seven segment.