Project Report: RISC-V Pipelined CPU Simulation

1. Introduction

In this project, a pipelined RISC-V processor is designed and simulated using C++. By simulating the flow of instructions via the five main pipeline stages—Fetch, Decode, Execute, Memory Access, and Write Back—the simulator simulates the internal operations of a CPU. Branch prediction, data forwarding, hazard detection, and the flexibility to execute in pipelined or single-cycle modes are among the features it supports.

2. Objective

The simulator's primary goal is to accurately handle a variety of risks and control flows, including branches and jumps, while simulating how a pipelined CPU handles several instructions at once. It is meant to serve as a teaching tool to help students comprehend how an actual CPU manages the execution of instructions within.

3. Key Components

a. Pipeline Registers

Four pipeline registers are used in the simulation:
Between Instruction Fetch and Decode, values are stored using IF/ID.
Between Decode and Execute, or ID/EX.
EX/MEM stands for Execute/Memory.
MEM/WB stands for "Memory and Write Back."
Control signals, register data, immediate values, and other fields required for the following step are all carried by each register.

b. Instruction Types Supported

The CPU simulator can handle various instruction formats of the RISC-V ISA:

- **R-type** (e.g., ADD, SUB, MUL)
- **I-type** (e.g., ADDI, LW, JALR)
- **S-type** (e.g., SW)
- **B-type** (e.g., BEQ, BNE)
- **U-type** (e.g., LUI, AUIPC)
- **J-type** (e.g., JAL)

c. Branch Prediction

Included is a basic branch prediction unit (BPU). To anticipate if a branch is taken and where it will go, it makes use of a Branch Target Buffer (BTB) and a 1-bit predictor. In the Execute step, errors are identified and fixed**.**

d. Hazard Detection and Forwarding

When the simulator identifies data dangers, it either halts the pipeline or, if enabled, forwards data from later stages. If forwarding is turned off, load-use hazards—a specific case—cause the pipeline to pause.

4. Simulation Process

a. Initialization

The simulator starts by reading a machine code file, loading instructions into memory, and initializing registers. The stack pointer is also set.

b. Execution Flow

Depending on the configuration, the processor runs in one of two modes:

- **Pipelined Mode**: Instructions are executed across different stages concurrently.
- **Single-Cycle Mode**: One instruction is completed in all stages before the next begins.

Each clock cycle involves the following:

1. **Fetch**: Get the instruction from memory using the program counter (PC).
2. **Decode**: Break down the instruction and read required operands.
3. **Execute**: Perform ALU operations or calculate branch targets.
4. **Memory**: Access data memory for load/store operations.
5. **Write Back**: Save results to the destination register.

5. Features and Debugging Options

• Pipeline and Register Print: Capable of showing the register file and pipeline register state at each cycle.
• Branch Prediction Monitoring: Displays statistics such as hits and misses along with the BPU's judgments.
• Instruction Tracing: Monitors a particular instruction at every turn.
• Stall and Flush Warnings: These alert users when the pipeline is flushed because of an error in forecasting or stalled because of potential dangers.

6. Customization and Flags

The simulator can be customized through various boolean flags:

- pipeliningEnabled: Turns pipelining on or off.
- dataForwardingEnabled: Enables or disables data forwarding logic.
- printRegistersEnabled: Shows register values each cycle.

- printPipelineRegsEnabled: Displays intermediate pipeline states.
- printBPUEnabled: Shows internal state of the branch predictor.
- traceInstructionNum: Traces a specific instruction through the pipeline.

7. Output

At the end of execution, the simulator prints:

- Total number of clock cycles used.
- Final contents of all 32 registers.
- Non-zero memory contents.
- Branch predictor statistics (if enabled).

8. Conclusion

This simulator is a helpful resource for understanding the internal operations of contemporary pipelined processors. It takes into consideration important CPU features like branching and memory access while effectively modeling control and data flow. Additionally, it is simple to expand or modify for more complex subjects like cache, superscalar execution, or more sophisticated branch prediction algorithms due to its modular nature.