PipSim: A Basic Python-based RISC-V Pipeline Simulator

Prateek P Kulkarni
PES1UG22EC210
Department of Electronics and Communication Engineering,
PES University
(Sem 5, Section D)
pkulkarni2425@gmail.com

Abstract – Due to it's flexibility and modular structure, RISC-V has undoubtedly stood the time of the test, with changing landscapes of technology. As a part of the Computer Organisation and Design course in the 5th semester of my undergraduate engineering, we present a basic, python-based simulator to visualise and simulate one of the most vital features of RISC-V, it's pipelining. PipSim works fully on cloud, and can be deployed and reproduced on any platform that can host python and a few dependent lightweight libraries. While a lot remains to be done, the project accomplishes the spirit behind it. This serves as the project report, summarizing in completion the design, function, capabilities and also demonstrate the results and outputs for a few cases.

Index Terms – *RISC-V, Pipeline, Simulation, Computer Organization, UE22EC352A*

INTRODUCTION

PipSim is a RISC-V pipeline simulator designed to model and visualize the execution of RISC-V instructions through a 5-stage pipeline. The primary goals of PipSim are:

- To provide a detailed simulation of a RISC-V pipeline
- To visualize the flow of instructions through pipeline stages
- To demonstrate key concepts such as hazard detection and handling
- To serve as an educational tool for understanding modern processor design

ARCHITECTURE OVERVIEW

PipSim implements a 5-stage pipeline typical of RISC processors:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

The simulator models key components of a processor, including:

- Instruction parsing and classification
- Pipeline stage progression
- Data hazard detection and forwarding
- Basic branch prediction
- Register file and memory simulation

KEY COMPONENTS

Instruction

The `Instruction` class represents a single RISC-V instruction. It parses the raw instruction string and extracts relevant information such as opcode, source registers, destination register, and immediate values.

Key features:

- Instruction type classification (R-type, I-type, S-type, B-type, U-type, J-type)
- Parsing of instruction fields (op, rd, rs1, rs2, imm)
- Arithmetic operations: add, sub, addi
- Logical operations: and, or, xor, andi, ori, xori
- Shift operations: sll, srl, sra, slli, srli, srai
- Memory operations: lw, sw, lb, sb, lbu, sbu (*Upcoming!*)
- Control flow: beq, bne, blt, bge, bltu, bgeu, jal, jalr
- Upper immediate: lui, auipc

Pipeline

The `Pipeline` class is the core of the simulator. It manages the progression of instructions through the pipeline stages and handles hazard detection and resolution.

Key features:

- Simulation of all pipeline stages
- Stall management
- Hazard detection and handling
- Register file and memory simulation
- Cycle-by-cycle state tracking

Branch Predictor

The `BranchPredictor` class implements a simple branch prediction mechanism. In the current implementation, it uses a basic 1-bit predictor.

Key features:

- Branch outcome prediction
- Prediction table management

Forwarding Unit

The `ForwardingUnit` class is responsible for detecting data hazards and implementing data forwarding logic to mitigate pipeline stalls.

Key features:

- Data hazard detection
- Forwarding logic (to be fully implemented)

PIPELINE STAGE SIMULATIONS

PipSim models each of the five pipeline stages:

1. **Instruction Fetch (IF)**: Simulates fetching the instruction from memory.
2. **Instruction Decode (ID)**: Parses the instruction and prepares operands.
3. **Execute (EX)**: Simulates the ALU operation or address calculation.
4. **Memory Access (MEM)**: Handles memory read/write operations.
5. **Write Back (WB)**: Writes results back to the register file.

Each stage is implemented as a separate method in the `Pipeline` class, allowing for detailed modeling of stage-specific operations.

HAZARD HANDLING

PipSim implements mechanisms for handling both data and control hazards:

- **Data Hazards**: Detected by the `ForwardingUnit`. When a data hazard is detected, the pipeline is stalled for a specified number of cycles.
- **Control Hazards**: Basic support for detecting branches and jumps is implemented. The simulator currently stalls on control hazards.

Future versions may implement more sophisticated hazard resolution techniques, such as full data forwarding and speculative execution.

MEMORY AND REGISTER

PipSim provides basic simulation of memory and registers:

- **Memory**: Simulated using a dictionary in the `Pipeline` class, allowing for read and write operations.
- **Registers**: A dictionary in the `Pipeline` class represents the register file, tracking the state of all 32 RISC-V integer registers.

The memory simulation will soon support various operations:

- Store Word (sw): Stores a 32-bit value to memory
- Store Byte (sb): Stores the least significant byte to memory
- Store Byte Unsigned (sbu): Stores the least significant byte to memory
- Load Word (lw): Loads a 32-bit value from memory
- Load Byte (lb): Loads a byte from memory and sign-extends it to 32 bits
- Load Byte Unsigned (lbu): Loads a byte from memory and zero-extends it to 32 bits

These components allow for a comprehensive simulation of instruction effects and data flow, including precise modeling of byte-level memory accesses.

VISUALISATION

PipSim includes a visualization component that generates a graphical representation of the pipeline execution:

- Uses matplotlib to create a timeline of instruction flow through pipeline stages
- Color-coded stages for easy identification
- Stalls are visually represented

The `plot_pipeline` function handles the creation of this visualization.

USAGE GUIDE

To use PipSim:

1. Define your RISC-V assembly code as a multi-line string.
2. Create a `Pipeline` instance.
3. Run the simulation using the `run` method of the `Pipeline` class.
4. Use `plot_pipeline` to visualize the execution.
5. Examine the printed cycle-by-cycle state, final register state, and memory state.

FUTURE WORK

The current version is hosted in limited scope, with only a part of the entire ISA supported. Subsequent version roll-outs will focus on making it more full-fledged and realistic. It lacks in it's capacity to simulate entirely the intricacies of branching and memory. Furthermore, it is at present microarchitecture agnostic. It would be desirable to make it hardware-aware, for better performance and efficiency. Making it "intelligent" also lies in the scope, with the advent of ML–based microarchitecture techniques. Prefetching techniques, custom memory hierarchies stand desirable in the present age of rapid growth in tech, to stay relevant.

ACKNOWLEDGEMENTS

The author is grateful to Prof. Rajeshwari B for not only introducing the various aspects of computer architecture in such a lucid manner, but also for letting me take up and present this project for the course requirements.

CODE DEPENDENCIES AND AVAILABILITY

The code used to extract results in this study can be found here. The dependencies can also be found in the same file. The usability is granted subject to the MIT license.

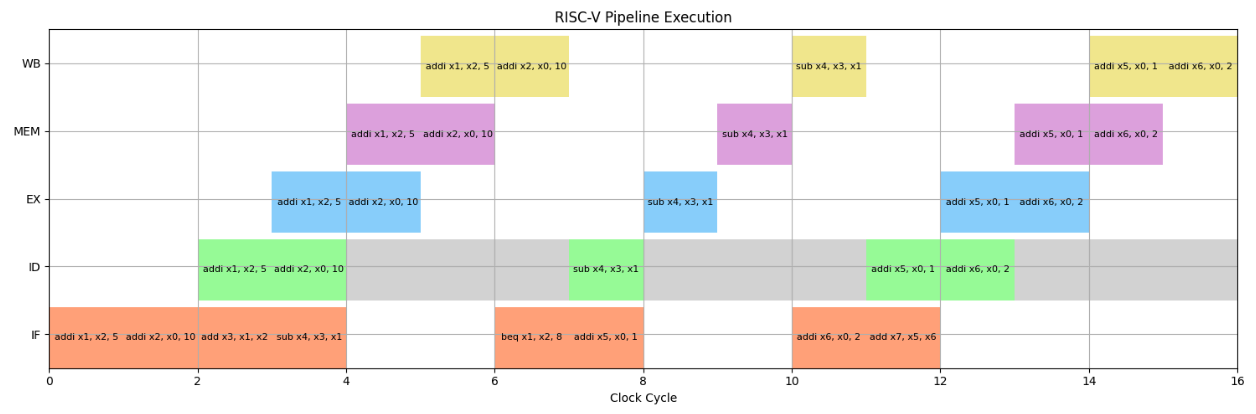
Appendix A: Test Codes and Results

A1:

a) Code 1:

```
addi x1, x2, 5

addi x2, x0, 10

add x3, x1, x2

sub x4, x3, x1

beq x1, x2, 8

addi x5, x0, 1

addi x6, x0, 2

add x7, x5, x6
```

b) Pipeline Visuals from PipSim:

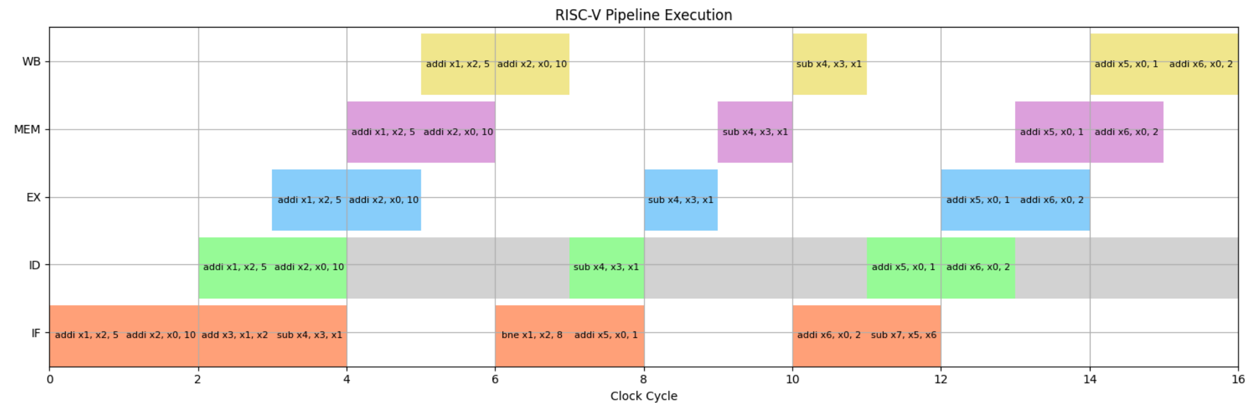


A2:

c) Code 2:

```
addi x1, x2, 5

addi x2, x0, 10

add x3, x1, x2

sub x4, x3, x1

bne x1, x2, 8

addi x5, x0, 1

xor x6, x0, 2

sub x7, x5, x6
```

d) Pipeline Visuals from PipSim:


RISC-V Pipeline Execution
WB
MEM
EX
ID
IF
addi x1, x2, 5
addi x2, x0, 10
add x3, x1, x2
sub x4, x3, x1
bne x1, x2, 8
addi x5, x0, 1
addi x6, x0, 2
sub x7, x5, x6
Clock Cycle