Design Document: Functional Simulator for Subset of ARM instruction set

# Introduction

# This document outlines the design and implementation details of myRISCVSim, a functional simulator developed to execute a subset of the ARM instruction set. The simulator is designed to provide a structured and efficient way to process machine instructions while mimicking the fundamental stages of a processor pipeline.

# Input/Output

## Input Format

The simulator takes a MEM file as input, which contains encoded RISCV instructions along with their respective memory addresses. The instructions are stored sequentially, with each entry specifying an address-instruction pair.

Example:

## Functional Behavior and output

# The simulator processes instructions through the following five-stage pipeline:

# Fetch: Reads the instruction from memory.

# Decode: Interprets the instruction and extracts necessary operands.

# Execute: Performs arithmetic, logic, or branch operations.

# Memory Access: Handles load/store instructions.

# Writeback: Updates register values.

# Execution continues until the SWI 0x11 instruction (0xEF000011) is encountered. At this point, the simulator halts and writes the updated memory contents to an output file.

# For better traceability, the simulator prints execution logs for each stage. For instance, for instruction 0xE0821003, the log might look like this:

# Design of Simulator

## Data structure

## The key data structures used in the simulator include:

## Registers: A global array representing 32 general-purpose registers.

## Memory: An unordered map that stores instruction and data memory dynamically.

## Program Counter (PC): Keeps track of the address of the current instruction.

## Intermediate Data Registers: Used to temporarily hold values during different pipeline stages.

## Instruction Lookup Table: A structured mapping of opcodes, function codes, and instruction types for efficient decoding.

## Simulator flow:

# The simulator operates in two primary steps:

# Memory Initialization: The MEM file is read into memory, storing the encoded instructions.

# Instruction Processing: The simulator runs an instruction cycle loop until a termination condition is met.

# Implementation of Pipeline Stages

## Fetch Stage

The fetch stage retrieves the next instruction from memory at the address specified by the **Program Counter (PC).** The instruction is fetched as a 32-bit value stored in four consecutive memory locations (little-endian format). The simulator constructs the instruction word using these memory bytes.

FETCH: Fetch instruction 0xE3A0200A from address 0x0

After fetching, the **PC is incremented by 4** to point to the next instruction unless a branch or jump alters the flow of execution.

## Decode Stage

The instruction word is broken down into different fields using **bitwise operations** and a structured lookup table. The simulator extracts:

* **Opcode**: Determines the instruction type.
* **Source and Destination Registers**: Extracted based on instruction format (R-type, I-type, etc.).
* **Immediate Values or Offsets**: Processed and sign-extended if needed.

Example output:

DECODE: ADD operation detected, operands: R2, R3, destination: R1

DECODE: Read registers R2 = 10, R3 = 2

The instruction is now prepared for execution.

## Execute Stage

The execution stage performs the actual computation based on the decoded instruction. Depending on the **operation type**, the simulator selects an arithmetic, logical, or control operation:

* ALU Operations (ADD, SUB, AND, OR, XOR, etc.)
* Shift Operations (SLL, SRL, SRA)
* Branch Comparisons (BEQ, BNE, BGE, BLT)
* Jump Instructions (JAL, JALR)

Optimized helper functions process these operations efficiently:

EXECUTE: Performing ADD operation: 10 + 2

For branch instructions, if the condition is met, the **PC is updated** to the target address.

## Memory Stage

For load/store instructions, memory is accessed in this stage. The simulator:

* Computes the **effective memory address** using base + offset.
* **Loads data** from memory into a register (for load instructions like LW, LH, LB, etc.).
* **Stores data** from a register into memory (for store instructions like SW, SH, SB, etc.).

If no memory access is required, the stage is skipped:

MEMORY: No memory access required

## Writeback Stage

In this final stage, the result computed in the **Execute** or **Memory** stage is written back to the register file. The destination register is updated with the computed value.

WRITEBACK: Stored result 12 in R1

If the instruction is a store operation, no register is updated in this stage.

## Termination

The simulator halts execution when it encounters the instruction:

0xEF000011 (SWI 0x11)

At this point, the final state of memory and registers is written to output files:

* **Final memory contents** to a file.
* **Register states** to another file.

# Test plan

To ensure correctness and robustness, the simulator is tested with the following programs, each implemented in both **iterative and recursive** approaches:

1. **Bubble Sort** – Sorting an array using nested loops and comparisons.
2. **Fibonacci Sequence** – Generating Fibonacci numbers up to a given index.
3. **Factorial Calculation** – Computing the factorial of a number.

These test cases validate the simulator’s handling of:

* Arithmetic operations
* Branching and control flow
* Memory access and data handling
* Function calls and recursion

By structuring and optimizing execution through lookup tables, modular function calls, and efficient memory management, **myRISCVSim** provides a reliable and scalable framework for RISCV instruction simulation.