Design Document

Functional Simulator

Subset of RISC-V instruction set

The document describes the design aspect of a functional simulator for subset of RISC-V instruction set, inspired from Venus RISC-V simulator, as part of CS204 under the mentorship of Dr. T. V. Kalyan, IIT Ropar.  The simulator executes machine code (.mc files) generated by a RISC-V assembler, modeling the 5-stage pipeline: ****Fetch**, **Decode**, **Execute**, **Memory**, and **Writeback****.. You may visit the GitHub repo [here](https://github.com/AVTG-1/RV32I_-Subset-_Simulator).

# Input/Output

## Input

Input to the simulator is the assembled output of a RISC-V RV32I assembly program, assembled by the assembler designed in phase-1 ([Link to Assembler](https://github.com/AVTG-1/RISCV_Assembler)).

InputFile.mc

For example:

0x0 0x00400093 # addi x1, x0, 4

0x4 0x00208133 # add x2, x1, x2

## Output

1. **Execution Trace**: Messages for each pipeline stage (e.g., FETCH: 0x00400093 from 0x0).
2. **Updated State**:
   * data\_mem.mc: Final data memory dump (hex).
   * data\_r.mc: Register file state (decimal).

## Functional Behavior

|  |  |
| --- | --- |
| **Stage** | **Description** |
| Fetch | Loads instruction from memory into **IR**. Increments **PC** by 4. |
| Decode | Extracts **opcode**, **funct3**, registers, and immediates. Logs operands. |
| Execute | Performs ALU operations or computes branch targets. |
| Memory | Handles **lw**/**sw** or bypasses for non-memory ops. Logs address/data accessed. |
| Writeback | Updates registers (except **x0**) or memory. Logs written values. |

The simulator reads the instruction from instruction memory, decodes the instruction, read the register, execute the operation, and write back to the register file. The instruction set supported is same as given in the lecture notes.

The simulator also prints messages for each stage, for example for the third instruction above following messages are printed.

* Fetch prints:
  + “FETCH:Fetch instruction 0xE3A0200A from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1”
  + “DECODE: Read registers R2 = 10, R3 = 2”
* Execute
  + “EXECUTE: ADD 10 and 2”
* Memory
  + “MEMORY:No memory operation”
* Writeback
  + “WRITEBACK: write 12 to R1”

Example Trace for **add x2, x1, x2**:

FETCH: 0x00208133 from 0x4

DECODE: ADD | x1 (4), x2 (0) → x2

EXECUTE: 4 + 0 = 4

MEMORY: No memory operation

WRITEBACK: x2 = 4

# Design of Simulator

## Data structure

The simulator uses the following core data structures to model the RISC-V processor state:

#### **1. Processor State**

// Register File (x0-x31)

uint32\_t reg[32] = {0}; // x0 hardwired to 0

// Byte-addressable memory (4KB)

uint8\_t memory[4096]; // Covers 0x00000000 to 0x00000FFF

// Pipeline Registers

struct PipelineState {

uint32\_t PC; // Program Counter

uint32\_t IR; // Instruction Register

int32\_t ALUResult; // Output of Execute stage

int32\_t MemData; // Data read/written in Memory stage

int32\_t Imm; // Decoded immediate value

bool BranchTaken; // Branch prediction flag

};

* ****Encapsulation****: All variables are declared as global within **PHASE2.cpp** to avoid external dependencies while maintaining modularity between functions.
* ****Initialization****:
  + **reg[2]** (stack pointer) is initialized to **0x10000000** (aligned to 16 bytes).
  + **PC** starts at **0x00000000**.

### ****Simulator flow:****

The simulator operates in two phases:

#### ****Phase 1: Memory Initialization****

1. ****Input****: Reads the .mc file (e.g., fibonacci.mc).
2. ****Loading****:
   * Instructions are stored consecutively starting at 0x00000000.
   * Data (if any) is loaded at 0x10000000 (heap/stack region).
3. **Example**:

0x0 0x00400093 // addi x1, x0, 4 → loaded at memory[0..3]

0x4 0x00208133 // add x2, x1, x2 → loaded at memory[4..7]

#### ****Phase 2: Instruction Execution****

An infinite loop processes instructions until encountering the **exit instruction** (0x0000007F):

while (true) {

fetch();

decode();

execute();

memory\_access();

writeback();

if (IR == 0x0000007F) break; // Exit condition

}

### ****Pipeline Stage Implementation****

#### **1. **Fetch****

* ****Logic****:
  + Reads 4 bytes from memory[PC] into IR.
  + Increments PC += 4 (unless branch/jump).
* ****Output****:

FETCH: Instruction 0x00400093 from 0x00000000

#### **2. Decode**

* ****Tasks****:
  + Splits IR into opcode, funct3, rd, rs1, rs2, and immediates.
  + Reads register values (reg[rs1], reg[rs2]).
* ****Output****:

DECODE: addi | x0 (0), imm=4 → x1

DECODE: addi | x0 (0), imm=4 → x1

#### **3. **Execute****

* ****Operations****:
  + ALU computes results (e.g., reg[rs1] + Imm for addi).
  + For branches, calculates PC = PC + Imm.
* **Output**:

EXECUTE: addi 0 + 4 = 4

#### **4. **Memory Access****

* ****Handling****:
  + lw: Reads memory[ALUResult] into MemData.
  + sw: Writes reg[rs2] to memory[ALUResult].
* ****Output****:

MEMORY: Read 0x42 from 0x10000000 (lw)

#### **5. **Writeback****

* ****Actions****:
  + Updates reg[rd] (except x0).
  + Skips for sw/beq instructions.
* ****Output****:

WRITEBACK: x1 = 4

# Flowchart LR

A[Load .mc File] --> B[Initialize PC=0]

B --> C{Fetch}

C --> D[Decode]

D --> E[Execute]

E --> F[Memory]

F --> G[Writeback]

G --> H{Exit?}

H -- No --> C

H -- Yes --> I[Save data\_mem.mc]

# Test plan

We test the simulator with following assembly programs:

* Fibonacci Program
* Factorial
* Bubble Sort on an array