Design Document: Functional Simulator for RISC-V Instruction Set

Introduction

This document describes the design of a functional simulator for the RISC-V instruction set. The simulator implements the five-stage instruction execution process: Fetch, Decode, Execute, Memory Access, and Write-back. It models the CPU state including registers, memory, and control flow.

Input/Output

Input

The simulator takes a machine code file (.mc) as input. Each line of the file contains an address and the corresponding 32-bit instruction or data value, separated by space. For example:

```
0x0 0x00a00093   # addi    x1, x0, 10
0x4 0x00000113   # addi    x2, x0, 0
0x8 0x00100193   # addi    x3, x0, 1
0xc 0x00008c63   # beq     x1, x0, fibo_done
0x10 0x00310233  # add     x4, x2, x3
0x14 0x00018113  # addi    x2, x3, 0
0x18 0x00020193  # addi    x3, x4, 0
0x1c 0xfff08093  # addi    x1, x1, -1
0x20 0xfedff06f  # jal     x0, fibo_loop
0x24 0x00000013  # addi    x0, x0, 0
0x28 0x00000000  # TERM
```

Lines beginning with '#' are treated as comments.

Functional Behavior and Output

The simulator reads instructions from instruction memory, decodes each instruction, executes the operation, accesses memory if needed, and writes back to the register file. The simulator supports the core RISC-V RV32I instruction set with extensions for multiplication and division operations.

The execution continues until one of the following occurs:

- A termination instruction (all zeros) is encountered
- No instruction exists at the PC
- The user chooses to exit the simulation

During execution, the simulator prints detailed messages for each stage. For example:

```
[Fetch] PC = 0x8, IR = 0x00100193
[Decode] opcode=0x13 rd=3 rs1=0 imm=1
[Execute] ADDI: 0 + 1 = 1
[WB] Updated R[3] = 1
```

After simulation completes, the updated data segment is written back to the original input file.

Design of Simulator

Data Structures

The simulator implements the following key data structures:

1. Register File: A 32-element array representing the 32 integer registers.

```
int32_t registers[NUM_REGS];
```

2. Program Counter (PC): Holds the address of the current instruction.

```
uint32_t programCounter;
```

3. Instruction Register (IR): Holds the current instruction being executed.

```
uint32_t instructionRegister;
```

4. Operand and ALU Registers: Intermediate registers for instruction execution.

```
int32_t operandA, operandB, operandM, aluResult,
   writeBackResult, memoryDataRegister;
```

5. Instruction Memory: Maps addresses to 32-bit instructions.

```
std::map<uint32_t, uint32_t> instructionMemory;
```

6. Data Segment: A byte-addressable memory for program data.

```
class DataSegment {
    std::map<uint32_t, uint8_t> storage;
    // Methods for reading/writing bytes and words
};
```

7. Decoded Instruction Structure: Holds the components of a decoded instruction.

```
struct DecodedInstruction {
    uint32_t opcode, rd, rs1, rs2, funct3, funct7;
    int32_t immediate;
};
```

Simulator Flow

The simulator operates in two main steps:

1. Initialization:

- Load machine code into instruction memory and data segment
- Initialize registers to zero
- Display initial register state

2. Execution Loop:

- Fetch instruction from memory at current PC
- Decode instruction into components
- Execute operation based on opcode and function codes
- Access memory if required (load/store)
- Write back result to destination register
- Update PC and cycle count
- Display register state

The simulator provides two modes of operation:

- Step-by-step: Execute one instruction at a time with user prompts
- Run-to-end: Execute all instructions continuously until termination

Implementation of Five-Stage Pipeline

1. Fetch Stage

```
instructionRegister = instructionMemory[programCounter];
```

The fetch stage retrieves the instruction at the current PC from instruction memory. If no instruction exists at the PC, the simulation ends.

2. Decode Stage

```
DecodedInstruction decoded = decodeInstruction(
    instructionRegister);
```

The decode stage extracts the components of the instruction:

- Opcode: Identifies the instruction category
- Register specifiers: rd, rs1, rs2
- Function codes: funct3, funct7
- Immediate value: Used for many instruction types

3. Execute Stage

The execute stage performs different operations based on the opcode and function codes:

- For R-type instructions (opcode 0x33): ALU operations like ADD, SUB, XOR, AND, etc.
- For I-type ALU instructions (opcode 0x13): ADDI, XORI, ANDI, etc.
- For load/store instructions: Address calculation
- For branch instructions: Branch condition evaluation
- For jump instructions: Target address calculation

4. Memory Access Stage

For load instructions:

```
memoryDataRegister = dataSegment.readWord(effectiveAddr);
```

For store instructions:

```
dataSegment.writeWord(effectiveAddr, operandM);
```

5. Write-back Stage

```
if (decoded.rd != 0) {
    registers[decoded.rd] = writeBackResult;
}
```

The write-back stage updates the destination register with the computation result. Register 0 is hardwired to zero in RISC-V, so it's never modified.

Instruction Set Support

The simulator supports the following RISC-V instruction categories:
1. R-type Instructions (ALU operations with two register sources):

- ADD, SUB, XOR, OR, AND, SLL, SLT, SRL, SRA
- Extension: MUL, DIV, REM

2. I-type Instructions (ALU operations with immediate):

- ADDI, XORI, ORI, ANDI, SLTI, SLLI, SRLI, SRAI

3. Load Instructions (Memory to register):

- LB, LH, LW

4. Store Instructions (Register to memory):

- SB, SH, SW

5. Branch Instructions (Conditional branches):

- BEQ, BNE, BLT, BGE

6. Jump Instructions (Unconditional jumps):

- JAL, JALR

7. Upper Immediate Instructions:

- LUI, AUIPC

Test Plan

The simulator should be tested with the following RISC-V programs:

- Fibonacci sequence calculation
- Factorial calculation
- Bubble sort algorithm

These test programs exercise various aspects of the instruction set including arithmetic operations, branching, memory access, and subroutine calls.

Additional Features

1. Interactive Simulation: Users can step through execution one instruction at a time or run the entire program.

2. Register Display: After each instruction, the simulator displays the state of all registers.

3. Termination Instruction: The simulation ends when an all-zero instruction is encountered.

4. Data Segment Persistence: On exit, the simulator updates the original input file with the new data segment values.

5. Cycle Counter: A global cycle counter tracks the number of instructions executed.