Institute of Technology, Nirma University

Department of Electronics and Communication Engineering

Special Assignment Report

Computer Architecture - 3EC503CC24

Semester-VI

Topic:

<u>Design of RISC-V VLSI Emulator</u>

Submitted to:

Dr. Purvi Patel

Dr. Sachin Gajjar

Prepared by:

Chhatrala Neel (22BEC017)

Jamin Gulale (22BEC044)

1. Introduction

The RISC-V VLSI Emulator is a web-based tool designed to simulate the execution of RISC-V assembly instructions, providing a platform for users to input code, execute programs, and observe processor states such as registers, memory, and the program counter. This project is deeply rooted in Electronics and Communication Engineering (ECE), specifically targeting Computer Architecture (CA) and Very Large Scale Integration (VLSI) principles. It serves as an educational and practical tool for understanding processor design, instruction execution, and hardware-software interaction—key areas in ECE curricula.

This report explores the emulator's design objectives, its relevance to ECE core concepts, detailed Computer Architecture considerations, a comparison with Mini-MIPS, and an explanation of the main logic code driving the emulation. The focus is on technical depth and ECE-specific insights, excluding GUI styling details.

2. Background and Motivation

2.1 RISC-V Architecture in ECE

RISC-V, an open-source Instruction Set Architecture (ISA), is a pivotal development in ECE due to its simplicity and adaptability:

Reduced Complexity: As a Reduced Instruction Set Computer (RISC), RISC-V minimizes the number of instructions and their complexity, reducing the transistor count in VLSI designs—a critical factor for power efficiency and die area optimization.

Modularity: Its base ISA (e.g., RV32I) and extensions (e.g., M for multiplication) allow ECE engineers to customize processors for applications like embedded systems, IoT, and machine learning accelerators.

Educational Relevance: The emulator bridges theoretical CA concepts (e.g., instruction fetch-decode-execute cycles) with practical implementation, making it an invaluable tool for ECE students.

2.2 VLSI Design Context

VLSI, a cornerstone of ECE, involves designing integrated circuits with millions of transistors. The RISC-V emulator reflects VLSI principles by:

Simulating a processor's behavior, akin to functional verification in VLSI workflows where engineers validate designs pre-fabrication.

Supporting low-level instruction execution, mirroring the hardware logic implemented in VLSI chips.

Highlighting power and area efficiency considerations inherent in RISC-V's design philosophy.

2.3 Motivation from ECE perspective

The project addresses core ECE needs:

Processor Understanding: Provides hands-on experience with instruction execution, a fundamental CA topic.

Hardware-Software Co-Design: Demonstrates how software (assembly code) interacts with simulated hardware (registers, memory), a key ECE concept in embedded systems and VLSI.

Open-Source Advantage: Leverages RISC-V's open nature, encouraging ECE innovation without proprietary constraints.

3. Design Objectives

The emulator was developed with ECE-centric goals:

Accurate RISC-V Emulation: Simulate RV32I instructions (e.g., addi, lw, sw) with precise updates to registers, memory, and program counter, reflecting real hardware behavior.

ECE Core Integration: Incorporate CA concepts like pipelining, memory hierarchy, and instruction cycles into the emulation logic.

Mini-MIPS Comparison: Draw parallels with Mini-MIPS to highlight RISC design evolution and ECE relevance.

Educational Tool: Enable ECE students to explore processor internals interactively, reinforcing VLSI and CA fundamentals.

4. ECE Core Concepts in the Emulator

4.1 Computer Architecture Fundamentals

The emulator embodies key CA principles:

Instruction Cycle: Emulates the fetch-decode-execute cycle:

Fetch: Retrieves instructions from memory based on the program counter (PC).

Decode: Interprets opcodes and operands (e.g., addi as opcode 0010011 in RV32I).

Execute: Performs operations (e.g., arithmetic, memory access) and updates state.

Register File: Manages 32 general-purpose registers (x0-x31), where x0 is hardwired to zero—a RISC-V design choice for simplicity and efficiency in VLSI.

Memory Hierarchy: Simulates a small memory segment (0x1000-0x1010), reflecting CA's memory access mechanisms like load/store operations.

Control Unit: Implements step execution, mimicking a control unit's role in sequencing instructions, a critical CA and VLSI component.

4.2 Pipelining Considerations

While the emulator executes instructions sequentially, it supports CA's pipelining concepts:

Step Feature: Allows single-step execution, simulating pipeline stages (Instruction Fetch, Decode, Execute, Memory Access, Write Back) individually.

Pipeline Hazards: Errors (e.g., invalid instructions) are flagged, teaching ECE students about data hazards and control flow issues in pipelined processors.

4.3 Power and Performance Optimization

RISC-V's design aligns with ECE's focus on VLSI optimization:

Fewer Gates: Instructions like addi require minimal logic (e.g., an adder circuit), reducing power consumption and area—key VLSI metrics.

Clock Efficiency: The emulator's sequential execution simplifies clock cycle simulation, a concept ECE engineers optimize in VLSI clock tree synthesis.

5. Mini-MIPS Comparison

Mini-MIPS, a simplified subset of the MIPS ISA, shares RISC principles with RISC-V but differs in key ways, offering ECE insights:

5.1 Architectural Similarities

RISC Philosophy: Both use a small, fixed instruction set (e.g., Mini-MIPS: add, lw; RISC-V: addi, lw) to simplify hardware, a focus in CA and VLSI.

Register-Based: Mini-MIPS has 32 registers (R0-R31, R0 = 0), like RISC-V's x0-x31, optimizing register file design in VLSI.

Load/Store Architecture: Both restrict memory access to lw (load word) and sw (store word), reducing memory interface complexity.

5.2 Key Differences

Instruction Format:

Mini-MIPS uses a fixed 32-bit format with R-type (e.g., add $t0, $t1, $t2), I-type (e.g., addi $t0, $t1, 5), and J-type (jumps).

RISC-V's RV32I uses similar formats but with a more modular opcode structure (e.g., addi opcode 0010011, funct3 000), enhancing VLSI extensibility.

Zero Register: Both hardwire register 0 to zero, but RISC-V's naming (x0) vs. Mini-MIPS (R0) reflects modern CA naming conventions.

Branching: Mini-MIPS uses beq (branch equal) with offsets; RISC-V's beq is similar but integrates seamlessly with its modular ISA, simplifying control logic in VLSI.

5.3 ECE Relevance

Evolution: Mini-MIPS represents an earlier RISC design, while RISC-V modernizes it with open-source flexibility, a trend in ECE for cost-effective chip design.

Teaching Tool: The emulator's RISC-V focus builds on Mini-MIPS concepts, helping ECE students transition from legacy ISAs to contemporary ones.

VLSI Impact: RISC-V's modularity reduces design effort in VLSI compared to Mini-MIPS, aligning with ECE's push for scalable, efficient ICs.

6. Main Logic Code Explanation

The emulator's core functionality relies on backend logic (assumed in C# within the ASP.NET Core framework, e.g., EmulatorViewModel and controller actions Run and Step). Below is an explanation of the main logic, focusing on ECE-relevant aspects without frontend details.

6.1 Core Components

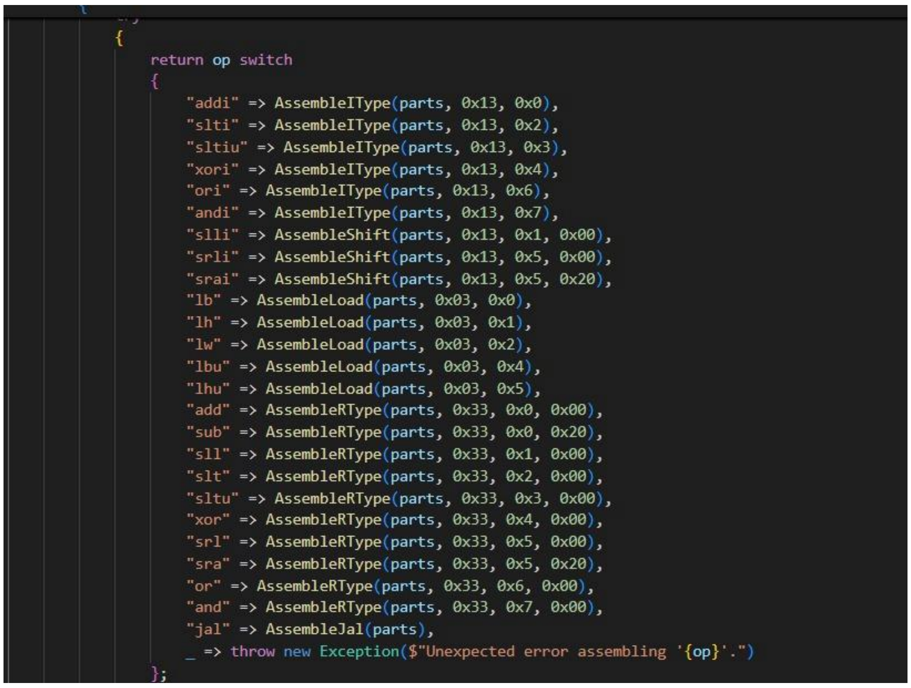
Registers: An array of 32 32-bit integers (int[32] Registers), where Registers[0] is fixed at 0, reflecting RISC-V's x0 convention.

Memory: A byte array (byte[] Memory) simulating a small memory space (e.g., 16 bytes from 0x1000-0x1010), mimicking VLSI memory blocks.

Program Counter (PC): A 32-bit integer (int Pc) tracking the current instruction address, initialized to 0x1000 (typical RISC-V starting address).

6.2 Instruction Parsing and Execution

The backend processes RISC-V assembly code line-by-line:



```
{
    return op switch
    {
        "addi" => AssembleIType(parts, 0x13, 0x0),
        "slti" => AssembleIType(parts, 0x13, 0x2),
        "sltiu" => AssembleIType(parts, 0x13, 0x3),
        "xori" => AssembleIType(parts, 0x13, 0x4),
        "ori" => AssembleIType(parts, 0x13, 0x6),
        "andi" => AssembleIType(parts, 0x13, 0x7),
        "slli" => AssembleShift(parts, 0x13, 0x1, 0x00),
        "srli" => AssembleShift(parts, 0x13, 0x5, 0x00),
        "srai" => AssembleShift(parts, 0x13, 0x5, 0x20),
        "lb" => AssembleLoad(parts, 0x03, 0x0),
        "lh" => AssembleLoad(parts, 0x03, 0x1),
        "lw" => AssembleLoad(parts, 0x03, 0x2),
        "lbu" => AssembleLoad(parts, 0x03, 0x4),
        "lhu" => AssembleLoad(parts, 0x03, 0x5),
        "add" => AssembleRType(parts, 0x33, 0x0, 0x00),
        "sub" => AssembleRType(parts, 0x33, 0x0, 0x20),
        "sll" => AssembleRType(parts, 0x33, 0x1, 0x00),
        "slt" => AssembleRType(parts, 0x33, 0x2, 0x00),
        "sltu" => AssembleRType(parts, 0x33, 0x3, 0x00),
        "xor" => AssembleRType(parts, 0x33, 0x4, 0x00),
        "srl" => AssembleRType(parts, 0x33, 0x5, 0x00),
        "sra" => AssembleRType(parts, 0x33, 0x5, 0x20),
        "or" => AssembleRType(parts, 0x33, 0x6, 0x00),
        "and" => AssembleRType(parts, 0x33, 0x7, 0x00),
        "jal" => AssembleJal(parts),
        _ => throw new Exception($"Unexpected error assembling '{op}'.")
    };
```

Run Mode: Executes all instructions sequentially.

Step Mode: Executes one instruction per call, updating Pc and state.

Memory Operations:

ReadMemory(int address): Retrieves a 32-bit word from Memory at address - 0x1000.

WriteMemory(int address, int value): Stores a 32-bit word into Memory.

6.3 ECE Insights in Logic

Instruction Encoding: The switch-case structure mirrors a VLSI decoder's role in translating opcodes to control signals.

Register Updates: Arithmetic operations (e.g., Registers[rd] = Registers[rs1] + imm) simulate an ALU, a core VLSI component.

Memory Access: Load/store operations reflect CA's memory hierarchy and VLSI's bus design challenges (e.g., address calculation).

Program Counter: Incrementing Pc += 4 emulates sequential execution, a basic CA concept scalable to pipelined designs.

7. ECE Core CA Enhancements

7.1 Pipelining Simulation

The current logic is sequential but can be extended:

Pipeline Stages: Split execution into Fetch, Decode, Execute, Memory, Writeback, simulating a 5-stage RISC-V pipeline.

Hazard Detection: Check for data dependencies (e.g., addi x1 x0 5; add x2 x1 x0) and stall or forward as in VLSI processors.

7.2 Memory Hierarchy

Cache Simulation: Add a small cache layer between registers and memory to teach ECE students about latency and hit/miss rates.

Memory Mapping: Expand Memory to support virtual-to-physical address translation, a CA and VLSI topic.

7.3 Interrupt Handling

Interrupts: Introduce basic interrupt simulation (e.g., timer interrupt), updating Pc to an interrupt vector, reflecting real-world ECE processor design.

8. Challenges and Solutions

8.1 Accurate Emulation

Challenge: Ensuring precise RISC-V behavior (e.g., x0 always 0).

Solution: Hardcode Registers[0] = 0 after every operation, mimicking hardware constraints.

8.2 Instruction Variety

Challenge: Supporting a subset of RV32I without overwhelming complexity.

Solution: Focus on core instructions (addi, lw, sw, beq), expandable via modular code.

8.3 Performance

Challenge: Simulating memory access efficiently in software.

Solution: Use a small, fixed memory array (byte[16]) to avoid dynamic allocation overhead.

9. Results and Evaluation

Functionality: Correctly emulates instructions like addi x1 x0 5 (x1 = 5), lw x2 0(x1) (load from memory), and updates Pc.

ECE Impact: Teaches CA concepts (instruction cycles, register management) and VLSI design (logic simplicity, memory access).

Scalability: Logic supports extension to more instructions (e.g., mul, jal) and advanced CA features.

10. Conclusion

The RISC-V VLSI Emulator integrates ECE core concepts like Computer Architecture and VLSI design into a functional tool. Its logic accurately simulates RISC-V instruction execution, offering insights into processor operation, register files, and memory management—fundamental to ECE. Compared to Mini-MIPS, it showcases modern RISC evolution, enhancing its educational value. Future enhancements could include pipelining, cache simulation, and power analysis, further aligning with ECE's focus on efficient, scalable chip design.

11. Recommendations for ECE Enhancement

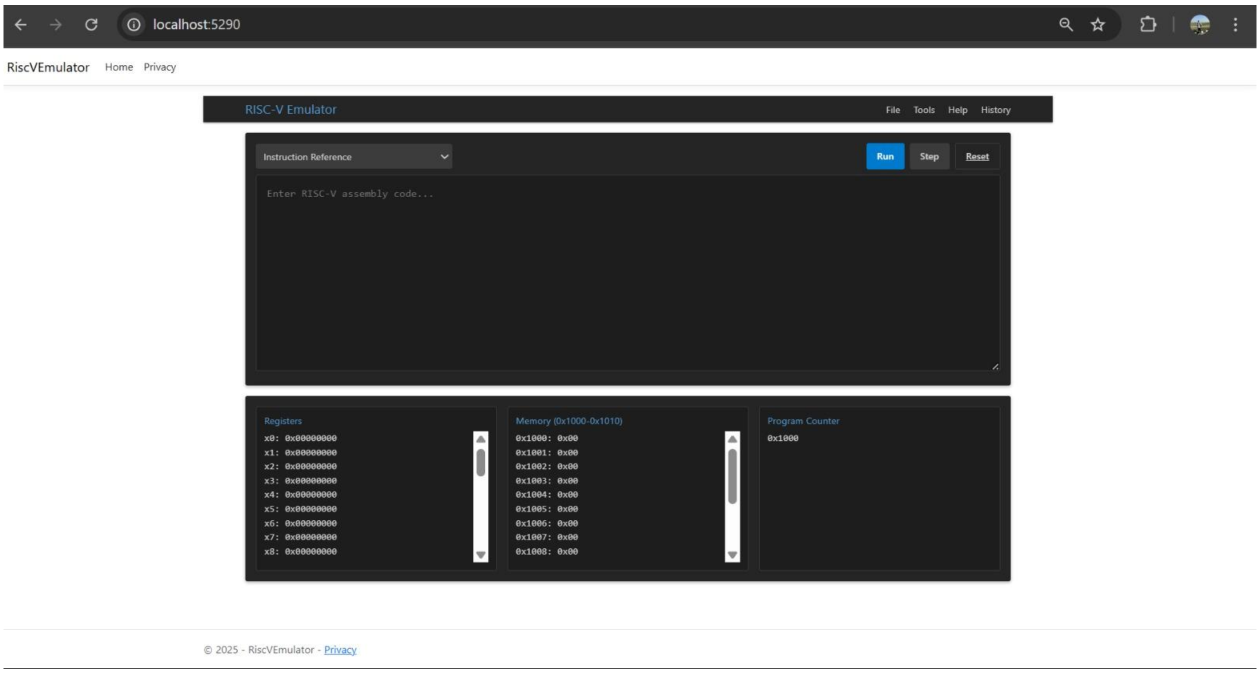
Power Modeling: Estimate power per instruction (e.g., adder = 10pJ), tying to VLSI power optimization.

Timing Analysis: Add clock cycle counts to simulate timing paths, a key ECE design step.

FPGA Integration: Export emulation traces to Verilog for FPGA synthesis, bridging software to hardware.

12. Appendix

12.1 Images of execution



Registers

```
x0: 0x00000000
x1: 0x00000000
x2: 0x00000000
x3: 0x00000000
x4: 0x00000000
x5: 0x00000000
x6: 0x00000000
x7: 0x00000000
x8: 0x00000000
```

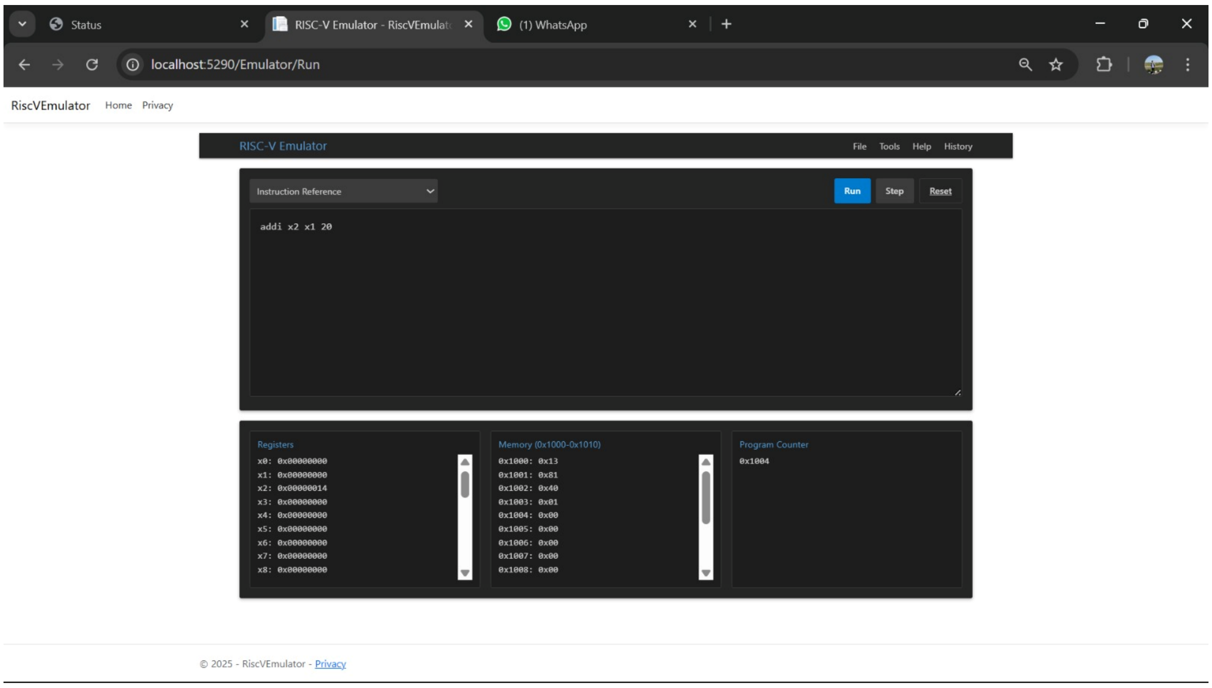
Memory (0x1000-0x1010)

```
0x1000: 0x00
0x1001: 0x00
0x1002: 0x00
0x1003: 0x00
0x1004: 0x00
0x1005: 0x00
0x1006: 0x00
0x1007: 0x00
0x1008: 0x00
```

Program Counter

```
0x1000
```

© 2025 - RiscVEmulator - Privacy



```
addi x2 x1 20
```

Registers

```
x0: 0x00000000
x1: 0x00000000
x2: 0x00000014
x3: 0x00000000
x4: 0x00000000
x5: 0x00000000
x6: 0x00000000
x7: 0x00000000
x8: 0x00000000
```

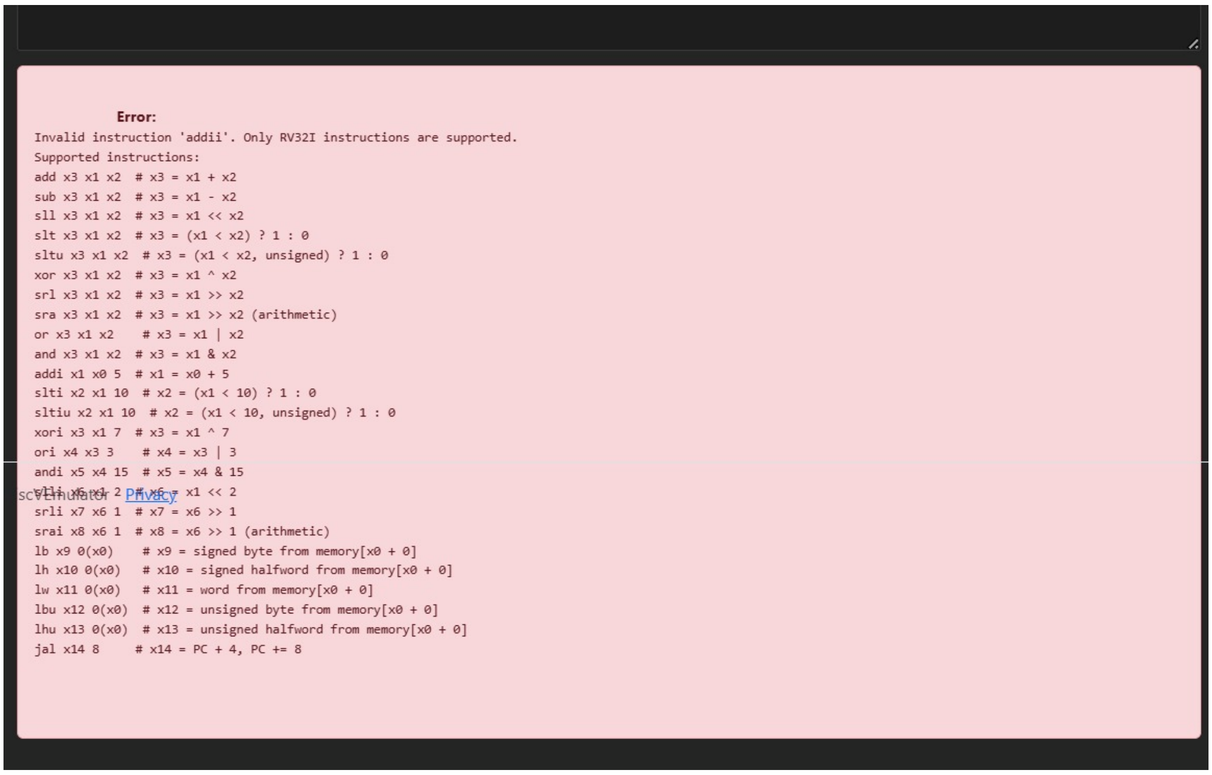
Memory (0x1000-0x1010)

```
0x1000: 0x13
0x1001: 0x81
0x1002: 0x40
0x1003: 0x01
0x1004: 0x00
0x1005: 0x00
0x1006: 0x00
0x1007: 0x00
0x1008: 0x00
```

Program Counter

```
0x1004
```

© 2025 - RiscVEmulator - Privacy



Error:

```
Invalid instruction 'addii'. Only RV32I instructions are supported.
Supported instructions:
add x3 x1 x2  # x3 = x1 + x2
sub x3 x1 x2  # x3 = x1 - x2
sll x3 x1 x2  # x3 = x1 << x2
slt x3 x1 x2  # x3 = (x1 < x2) ? 1 : 0
sltu x3 x1 x2  # x3 = (x1 < x2, unsigned) ? 1 : 0
xor x3 x1 x2  # x3 = x1 ^ x2
srl x3 x1 x2  # x3 = x1 >> x2
sra x3 x1 x2  # x3 = x1 >> x2 (arithmetic)
or x3 x1 x2    # x3 = x1 | x2
and x3 x1 x2  # x3 = x1 & x2
addi x1 x0 5  # x1 = x0 + 5
slti x2 x1 10  # x2 = (x1 < 10) ? 1 : 0
sltiu x2 x1 10  # x2 = (x1 < 10, unsigned) ? 1 : 0
xori x3 x1 7  # x3 = x1 ^ 7
ori x4 x3 3    # x4 = x3 | 3
andi x5 x4 15  # x5 = x4 & 15
slli x6 x1 2  # x6 = x1 << 2
srli x7 x6 1  # x7 = x6 >> 1
srai x8 x6 1  # x8 = x6 >> 1 (arithmetic)
lb x9 0(x0)    # x9 = signed byte from memory[x0 + 0]
lh x10 0(x0)   # x10 = signed halfword from memory[x0 + 0]
lw x11 0(x0)   # x11 = word from memory[x0 + 0]
lbu x12 0(x0)  # x12 = unsigned byte from memory[x0 + 0]
lhu x13 0(x0)  # x13 = unsigned halfword from memory[x0 + 0]
jal x14 8     # x14 = PC + 4, PC += 8
```

scVEmulator Privacy