Project Report

ECE 586 - WINTER 2025

RISC-V Instruction Set Architecture Simulator

By:
Karthik Bandla
Harsha Vardhan Duvvuru
Dhanush Savaram
Pavan Gaddam

Under the supervision of:
Professor Mark Faust

1. Introduction

The RISC-V Instruction Set Architecture (ISA) has gained significant popularity in recent years due to its open-source nature, modularity, and scalability. Designed with simplicity and efficiency in mind, RISC-V provides a flexible foundation for developing modern processors and embedded systems. This project focuses on the development of a **RISC-V Simple Simulator** that implements the **RV32I Base Integer Instruction Set**.

2. Testing Objectives

- Verify the correct loading of the memory image.
- Validate instruction fetching and decoding.
- Ensure correct execution of RISC-V instructions.
- Test different instruction types (R, I, S, B, U, J).
- Verify the handling of edge cases and invalid instructions.
- Ensure correct updating of registers and program counter (PC).
- Validate memory read/write operations.
- Test verbose and silent modes for output correctness.

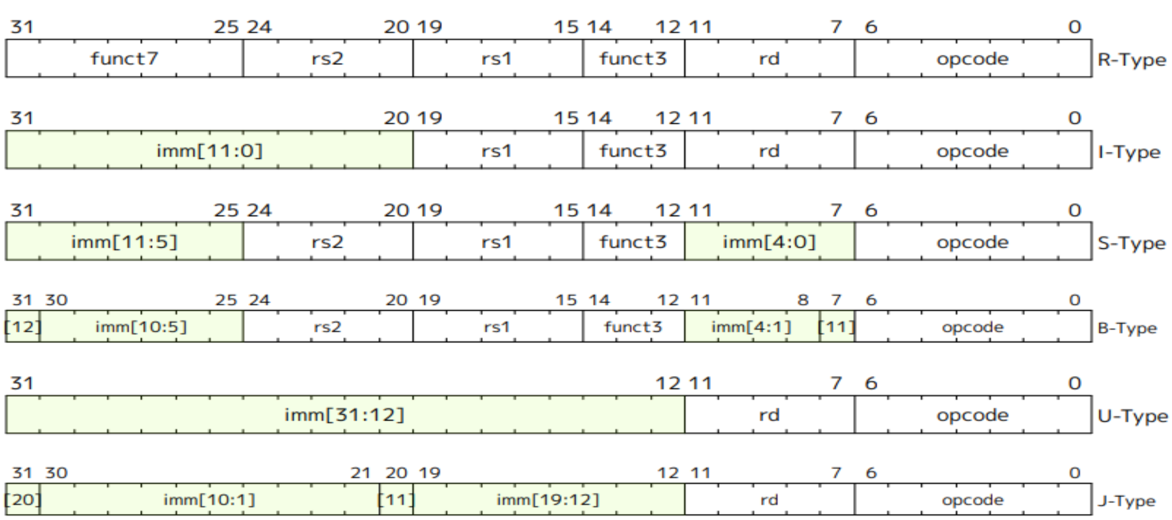
3. Test Environment

- **Hardware:** x86_64
- **Software:**
  - GCC/G++ compiler for C++
  - RISC-V GNU toolchain (riscv32-unknown-elf-as, riscv32-unknown-elf-objdump)
  - Linux/macOS/Windows (WSL for Windows users)
- **Test Files:**
  - C programs (.c files)
  - Assembly test programs (.s files)
  - Corresponding memory image files (.mem)

Command to run the RISC-V simulator.

- g++ RISCV.cpp -o RISCV
- ./RISCV <Mem File with .mem extension> <start address> <stack address> <mode>
- Mode Can be Verbose, Silent, debug

Encoding Formats:



Specifications:

1	Instruction	Opcode	funct3	funct7	Notes		Decode	verification
2	add	011_0011	000	0000000	Addition	**R-format**		done
3	sub	0110011	000	0100000	Subtraction			done
4	sll	0110011	001	0000000	Shift left logical			done
5	slt	0110011	010	0000000	Set less than			done
6	sltu	0110011	011	0000000	Set less than unsigned			done
7	xor	0110011	100	0000000	XOR			done
8	srl	0110011	101	0000000	Shift right logical			done
9	sra	0110011	101	0100000	Arithmetic shift right			done
10	or	0110011	110	0000000	OR			done
11	and	0110011	111	0000000	AND			done
12								
13	addi	001_0011	000	-	Add immediate	**I-format**		done
14	slti	0010011	010	-	Set less than (signed)			done
15	sltiu	0010011	011	-	Set less than (unsigned)			done
16	xori	0010011	100	-	XOR immediate			done
17	ori	0010011	110	-	OR immediate			done
18	andi	0010011	111	-	AND immediate			done
19	slli	0010011	001	-	Shift left logical			done
20	srli	0010011	101	0000000	Shift right logical			done
21	srai	0010011	101	0100000	Arithmetic shift right			done
22	jalr	1100111	000	-	Jump and link register			done
23	ecall/ebre	1110011	-	-	System calls			in progress
24								
25	sb	010_0011	000	-	Store byte	**S-format**		done
26	sh	0100011	001	-	Store halfword			done
27	sw	0100011	010	-	Store word			done
28								
29	beq	110_0011	000	-	Branch if equal	**B-format**		done
30	bne	1100011	001	-	Branch if not equal			done
31	blt	1100011	100	-	Branch if less than			done
32	bge	1100011	101	-	Branch if greater or equal			done
33	bltu	1100011	110	-	Branch if less than (unsigned)			done
34	bgeu	1100011	111	-	Branch if greater or equal (unsigned)			done
35								
36	lui	011_0111	-	-	Load upper immediate	**U-format**		done
37	auipc	0010111	-	-	Add upper immediate to PC			done
38	jal	110_1111	-	-	Jump and link	**J-format**		done
39	lb	000_0011	000	-	Load byte (signed)	**I-format**		done
40	lh	0000011	001	-	Load halfword (signed)			done
41	lw	0000011	010	-	Load word			done
42	lbu	0000011	100	-	Load byte (unsigned)			done
43	lhu	0000011	101	-	Load halfword (unsigned)			done
44								
45	mul	110011	0x0	0x1	Mul	**R-Format**		done
46	mulh	110011	0x1	0x1	Mul High			done
47	mulsu	110011	0x2	0x1	Mul High Signed & Unsigned			done
48	mulu	110011	0x3	0x1	Mul High			done
49	div	110011	0x4	0x1	Div			done
50	divu	110011	0x5	0x1	Div U			done
51	rem	110011	0x6	0x1	remainder			done
52	remu	110011	0x7	0x1	remainder			done

TEST PLAN

R-FORMAT

ADD

1. **Addition of a negative and a positive number**
   - Input: `t0 = -20`, `t1 = 4`
   - Expected Output: `a0 = -16`
2. **Addition of a positive and a negative number**
   - Input: `t0 = 1`, `t1 = -1`
   - Expected Output: `a1 = 0`
3. **Addition of two negative numbers**
   - Input: `t0 = -1`, `t1 = -1`
   - Expected Output: `a2 = -2`
4. **Addition of a positive number and zero**
   - Input: `t0 = 1`, `t1 = 0`
   - Expected Output: `a3 = 1`
5. **Addition causing overflow**
   - Input: `t0 = 0x7FFFFFFF` (largest positive 32-bit signed integer), `t1 = 1`
   - Expected Output: Overflow occurs, result wraps around to `a4 = 0x80000000` (smallest negative 32-bit signed integer in two's complement representation)

SUB

1. **Subtraction of a positive number from a negative number**
   - Input: `t0 = -20`, `t1 = 4`
   - Expected Output: `a0 = -24`
2. **Subtraction of a negative number from a positive number**
   - Input: `t0 = 1`, `t1 = -1`
   - Expected Output: `a1 = 2`
3. **Subtraction of a negative number from itself**
   - Input: `t0 = -1`, `t1 = -1`
   - Expected Output: `a2 = 0`
4. **Subtraction of zero from a positive number**
   - Input: `t0 = 1`, `t1 = 0`
   - Expected Output: `a3 = 1`
5. **Subtraction causing overflow**
   - Input: `t0 = 0x80000000` (smallest negative 32-bit signed integer), `t1 = 1`

- Expected Output: Overflow occurs, and the result wraps around to `a4 = 0x7FFFFFFF` (largest positive 32-bit signed integer in two's complement representation)

XOR

1. **XOR with all bits set (flips bits of t1)**
   - Input: `t0 = 0xFFFFFFFF`, `t1 = 0x12345678`
   - Expected Output: `a0 = 0xEDCBA987`
2. **XOR with alternating bit pattern**
   - Input: `t0 = 0xAAAAAAAA`, `t1 = 0x55555555`
   - Expected Output: `a1 = 0xFFFFFFFF` (all bits set to 1)
3. **XOR with sign bit flip**
   - Input: `t0 = 0x80000000`, `t1 = 0x7FFFFFFF`
   - Expected Output: `a2 = 0xFFFFFFFF` (all bits set to 1)
4. **XOR with zero and all ones (inverts all bits of t1)**
   - Input: `t0 = 0x00000000`, `t1 = 0xFFFFFFFF`
   - Expected Output: `a3 = 0xFFFFFFFF`
5. **XOR of two random large values**
   - Input: `t0 = 0xDEADBEEF`, `t1 = 0xCAFEBABE`
   - Expected Output: `a4 = 0x14760551`

OR

1. **OR with all bits set (should remain all 1s)**
   - Input: `t0 = 0xFFFFFFFF`, `t1 = 0x12345678`
   - Expected Output: `a0 = 0xFFFFFFFF`
2. **OR with alternating bit pattern (results in all 1s)**
   - Input: `t0 = 0xAAAAAAAA`, `t1 = 0x55555555`
   - Expected Output: `a1 = 0xFFFFFFFF`
3. **OR operation setting both highest and lowest bits**
   - Input: `t0 = 0x80000000`, `t1 = 0x00000001`
   - Expected Output: `a2 = 0x80000001`
4. **OR with zero and all ones (remains all 1s)**
   - Input: `t0 = 0x00000000`, `t1 = 0xFFFFFFFF`
   - Expected Output: `a3 = 0xFFFFFFFF`
5. **OR of two random large values (merging their bits)**
   - Input: `t0 = 0xDEADBEEF`, `t1 = 0xCAFEBABE`
   - Expected Output: `a4 = 0xDEFFBEFF`

AND

1. **AND with all bits set (result is unchanged `t1`)**
   - Input: `t0 = 0xFFFFFFFF`, `t1 = 0x12345678`
   - Expected Output: `a0 = 0x12345678`
2. **AND with alternating bit pattern (no overlapping bits, results in 0)**
   - Input: `t0 = 0xAAAAAAAA`, `t1 = 0x55555555`
   - Expected Output: `a1 = 0x00000000`
3. **AND operation where no bits overlap (results in 0)**
   - Input: `t0 = 0x80000000`, `t1 = 0x00000001`
   - Expected Output: `a2 = 0x00000000`
4. **AND with zero (always results in zero)**
   - Input: `t0 = 0x00000000`, `t1 = 0xFFFFFFFF`
   - Expected Output: `a3 = 0x00000000`
5. **AND of two random large values (bitwise AND result of both values)**
   - Input: `t0 = 0xDEADBEEF`, `t1 = 0xCAFEBABE`
   - Expected Output: `a4 = 0xCAACBAAE`

SLL

1. **Left shift a small value by 1**
   - Input: `t0 = 0x00000001`, `t1 = 1`
   - Expected Output: `a0 = 0x00000002`
2. **Left shift a byte-sized value by 8 (moves it to the next byte)**
   - Input: `t0 = 0x000000FF`, `t1 = 8`
   - Expected Output: `a1 = 0x0000FF00`
3. **Left shift a 16-bit value by 4**
   - Input: `t0 = 0x0000FFFF`, `t1 = 4`
   - Expected Output: `a2 = 0x000FFFF0`
4. **Left shift the largest signed positive 32-bit integer by 1 (causes overflow into sign bit)**
   - Input: `t0 = 0x7FFFFFFF`, `t1 = 1`
   - Expected Output: `a3 = 0xFFFFFFFE`
5. **Left shift a value with only the highest bit set by 2 (moves sign bit left, result becomes 0)**
   - Input: `t0 = 0x80000000`, `t1 = 2`
   - Expected Output: `a4 = 0x00000000` (shifting the highest bit left causes it to be lost)

SRL

1. **Logical right shift of a large value by 4 (preserving unsigned behavior)**
   - Input: `t0 = 0xF0000000`, `t1 = 4`
   - Expected Output: `a0 = 0x0F000000`
2. **Logical right shift of a middle-byte value by 8 (moves it down by one byte)**
   - Input: `t0 = 0x00FF0000`, `t1 = 8`
   - Expected Output: `a1 = 0x0000FF00`
3. **Logical right shift of a random value by 2**
   - Input: `t0 = 0x12345678`, `t1 = 2`
   - Expected Output: `a2 = 0x048D159E`
4. **Logical right shift of all bits set (ensuring zero-fill behavior)**
   - Input: `t0 = 0xFFFFFFFF`, `t1 = 16`
   - Expected Output: `a3 = 0x0000FFFF`
5. **Logical right shift of the highest bit set by 31 (moves only bit 31 to position 0)**
   - Input: `t0 = 0x80000000`, `t1 = 31`
   - Expected Output: `a4 = 0x00000001`

SRA

1. **Arithmetic right shift of a negative value (sign extension preserved)**
   - Input: `t0 = 0xF0000000`, `t1 = 4`
   - Expected Output: `a0 = 0xFF000000`
2. **Arithmetic right shift of a large negative value (sign bit propagates)**
   - Input: `t0 = 0x80000000`, `t1 = 8`
   - Expected Output: `a1 = 0xFF800000`
3. **Arithmetic right shift of the largest positive 32-bit integer by 1**
   - Input: `t0 = 0x7FFFFFFF`, `t1 = 1`
   - Expected Output: `a2 = 0x3FFFFFFF`
4. **Arithmetic right shift of -1 (all bits set, remains -1)**
   - Input: `t0 = -1` (0xFFFFFFFF in two's complement), `t1 = 4`
   - Expected Output: `a3 = 0xFFFFFFFF`
5. **Arithmetic right shift of a positive number with the highest bit unset**
   - Input: `t0 = 0x40000000`, `t1 = 2`
   - Expected Output: `a4 = 0x10000000`

SLT

1. **Set less than when `t0` is less than `t1` (positive numbers)**
   - Input: `t0 = 5`, `t1 = 10`
   - Expected Output: `a0 = 1` (since 5 < 10)
2. **Set less than when `t0` is a negative number and `t1` is positive**
   - Input: `t0 = -10`, `t1 = 10`
   - Expected Output: a1 = 1 (since -10 < 10)
3. **Set less than when `t0` is positive and `t1` is negative**
   - Input: `t0 = 10`, `t1 = -10`
   - Expected Output: a2 = `0` (since 10 is not less than -10)
4. **Set less than when `t0` is negative and `t1` is zero**
   - Input: `t0 = -1`, `t1 = 0`
   - Expected Output: a3 = 1 (since -1 < 0)
5. **Set less than when `t0` and `t1` are equal**
   - Input: `t0 = 100`, `t1 = 100`
   - Expected Output: a4 = `0` (since 100 is not less than 100)

SLTU

1. **Unsigned comparison where `t0` is -1 (interpreted as 0xFFFFFFFF) and `t1` is 1**
   - Input: `t0 = -1 (0xFFFFFFFF)`, `t1 = 1 (0x00000001)`
   - Expected Output: `a0 = 0` (since 0xFFFFFFFF > 0x00000001 in unsigned comparison)
2. **Unsigned comparison where `t0` is 1 and `t1` is -1 (interpreted as 0xFFFFFFFF)**
   - Input: `t0 = 1 (0x00000001)`, `t1 = -1 (0xFFFFFFFF)`
   - Expected Output: a1 = 1 (since 0x00000001 < 0xFFFFFFFF in unsigned comparison)
3. **Unsigned comparison of zero and a small positive number**
   - Input: `t0 = 0x00000000`, `t1 = 0x00000001`
   - Expected Output: a2 = 1 (since 0 < 1 in unsigned comparison)
4. **Unsigned comparison where `t0` is greater than `t1`**
   - Input: `t0 = 100 (0x00000064)`, `t1 = 50 (0x00000032)`
   - Expected Output: a3 = `0` (since 100 > 50 in unsigned comparison)
5. **Unsigned comparison where `t0` is 0x80000000 (large positive value in unsigned) and `t1` is 0x7FFFFFFF (smaller positive value)**
   - Input: `t0 = 0x80000000`, `t1 = 0x7FFFFFFF`
   - Expected Output: a4 = `0` (since 0x80000000 > 0x7FFFFFFF in unsigned comparison)

I-FORMAT

ADDI (Add Immediate)

1. **Addition of a negative and a positive number**
   - Input: t0 = -20, Immediate = 4
   - Expected Output: a0 = -16
2. **Addition of a positive and a negative number**
   - Input: t0 = 1, Immediate = -1
   - Expected Output: a1 = 0
3. **Addition of two negative numbers**
   - Input: t0 = -1, Immediate = -1
   - Expected Output: a2 = -2
4. **Addition of a positive number and zero**
   - Input: t0 = 1, Immediate = 0
   - Expected Output: a3 = 1
5. **Addition causing overflow**
   - Input: t0 = 0x7FFFFFFF, Immediate = 1
   - Expected Output: Overflow occurs, result wraps around to a4 = 0x80000000

SLTI (Set Less Than Immediate - Signed)

1. **t0 is less than Immediate (positive comparison)**
   - Input: t0 = 5, Immediate = 10
   - Expected Output: a0 = 1
2. **t0 is a negative number and Immediate is positive**
   - Input: t0 = -10, Immediate = 10
   - Expected Output: a1 = 1
3. **t0 is positive and Immediate is negative**
   - Input: t0 = 10, Immediate = -10
   - Expected Output: a2 = 0
4. **t0 is negative and Immediate is zero**
   - Input: t0 = -1, Immediate = 0
   - Expected Output: a3 = 1
5. **t0 and Immediate are equal**
   - Input: t0 = 100, Immediate = 100
   - Expected Output: a4 = 0

SLTIU (Set Less Than Immediate - Unsigned)

1. **Unsigned comparison where t0 is -1 (0xFFFFFFFF) and Immediate is 1**
   - Input: t0 = -1 (0xFFFFFFFF), Immediate = 1 (0x00000001)
   - Expected Output: a0 = 0
2. **Unsigned comparison where t0 is 1 and Immediate is -1 (0xFFFFFFFF)**
   - Input: t0 = 1 (0x00000001), Immediate = -1 (0xFFFFFFFF)
   - Expected Output: a1 = 1
3. **Unsigned comparison of zero and a small positive number**
   - Input: t0 = 0x00000000, Immediate = 0x00000001
   - Expected Output: a2 = 1
4. **Unsigned comparison where t0 is greater than Immediate**
   - Input: t0 = 100 (0x00000064), Immediate = 50 (0x00000032)
   - Expected Output: a3 = 0
5. **Unsigned comparison where t0 is equal to Immediate**
   - Input: t0 = 5000, Immediate = 5000
   - Expected Output: a4 = 0

XORI (Bitwise XOR Immediate)

1. **XOR with all bits set (flips bits of t0)**
   - Input: t0 = 0xFFFFFFFF, Immediate = 0x12345678
   - Expected Output: a0 = 0xEDCBA987
2. **XOR with alternating bit pattern**
   - Input: t0 = 0xAAAAAAAA, Immediate = 0x55555555
   - Expected Output: a1 = 0xFFFFFFFF
3. **XOR with sign bit flip**
   - Input: t0 = 0x80000000, Immediate = 0x7FFFFFFF
   - Expected Output: a2 = 0xFFFFFFFF
4. **XOR with zero (unchanged value)**
   - Input: t0 = 0x12345678, Immediate = 0x00000000
   - Expected Output: a3 = 0x12345678
5. **XOR of two random values**
   - Input: t0 = 0xDEADBEEF, Immediate = 0xCAFEBABE
   - Expected Output: a4 = 0x14760551

ORI (Bitwise OR Immediate)

1. **OR with all bits set (remains all 1s)**
   - Input: t0 = 0xFFFFFFFF, Immediate = 0x12345678
   - Expected Output: a0 = 0xFFFFFFFF
2. **OR with alternating bit pattern**

- Input: t0 = 0xAAAAAAAA, Immediate = 0x55555555
- Expected Output: a1 = 0xFFFFFFFF

3. **OR operation setting both highest and lowest bits**
   - Input: t0 = 0x80000000, Immediate = 0x00000001
   - Expected Output: a2 = 0x80000001
4. **OR with zero (unchanged value)**
   - Input: t0 = 0x12345678, Immediate = 0x00000000
   - Expected Output: a3 = 0x12345678
5. **OR of two random values**
   - Input: t0 = 0xDEADBEEF, Immediate = 0xCAFEBABE
   - Expected Output: a4 = 0xDEFFBEFF

ANDI (Bitwise AND Immediate)

1. **AND with all bits set (result is t0)**
   - Input: t0 = 0xFFFFFFFF, Immediate = 0x12345678
   - Expected Output: a0 = 0x12345678
2. **AND with alternating bit pattern (no overlapping bits, results in 0)**
   - Input: t0 = 0xAAAAAAAA, Immediate = 0x55555555
   - Expected Output: a1 = 0x00000000
3. **AND operation where no bits overlap (results in 0)**
   - Input: t0 = 0x80000000, Immediate = 0x00000001
   - Expected Output: a2 = 0x00000000
4. **AND with zero (always results in zero)**
   - Input: t0 = 0x12345678, Immediate = 0x00000000
   - Expected Output: a3 = 0x00000000
5. **AND of two random values**
   - Input: t0 = 0xDEADBEEF, Immediate = 0xCAFEBABE
   - Expected Output: a4 = 0xCAACBAAE

SLLI (Shift Left Logical Immediate)

1. **Left shift by 1**
   - Input: t0 = 0x00000001, Immediate = 1
   - Expected Output: a0 = 0x00000002
2. **Left shift by 31 (largest allowed shift)**
   - Input: t0 = 0x00000001, Immediate = 31
   - Expected Output: a1 = 0x80000000
3. **Left shift a negative number**
   - Input: t0 = -1 (0xFFFFFFFF), Immediate = 4
   - Expected Output: a2 = 0xFFFFFFF0

SRLI (Shift Right Logical Immediate)

1. **Right shift by 1**
    - Input: t0 = 0x00000008, Immediate = 1
    - Expected Output: a0 = 0x00000004
2. **Right shift a large number**
    - Input: t0 = 0x80000000, Immediate = 31
    - Expected Output: a1 = 0x00000001
3. **Right shift a negative number (treated as unsigned)**
    - Input: t0 = -1 (0xFFFFFFFF), Immediate = 4
    - Expected Output: a2 = 0x0FFFFFFF

SRAI (Shift Right Arithmetic Immediate)

1. **Arithmetic right shift of a negative number**
    - Input: `t0 = -1 (0xFFFFFFFF)`, Immediate = 1
    - Expected Output: `a0 = 0xFFFFFFFF` (preserves sign)
2. **Arithmetic right shift of a positive number**
    - Input: `t0 = 0x80000000`, Immediate = 1
    - Expected Output: `a1 = 0xC0000000`
3. **Right shift by 31 (largest shift)**
    - Input: `t0 = 0x80000000`, Immediate = 31
    - Expected Output: `a2 = 0xFFFFFFFF`

LOADS

We have written a ReadMem function to load values from memory to registers.

LB - Load Byte

- It loads a byte from the specified location to the specified register and is sign extended.
- We checked by giving 0xff and 0x7f to verify whether it was a sign extending or not.

LHW - Load Half Word

- It loads a Half word from the specified location to the specified register and is sign extended.

- We checked by giving 0xffff and 0x7fff to verify whether it is sign extending or not.

LW - Load Word

- It loads a word from the specified location to the specified register and is sign extended.

LBU - Load Byte Unsigned

- It loads a Byte from the specified location to the specified register and is 0 extended.
- We checked by giving 0xff and 0x7f to verify whether it is 0 extending or not.

LHWU - Load Half Word Unsigned

- It loads a word from the specified location to the specified register and is 0 extented.
- We checked by giving 0xffff and 0x7fff to verify whether it is 0 extending or not.

JALR

- It loads PC + 4 into the destination register and jumps to the value of source register + imm value specified in the instruction.
- We verified it by writing a C program starting from the main function and calling another function inside it and it jumps to the callee using JAL and it returns to the main function using the JALR instruction

S-FORMAT

We have written a StoreMem Function to store register values into Memory.

SB - Store Byte

- It stores a byte from the specified register to specified location.
- We have verified it by loading from the same location to a register.

SB - Store HalfWord

- It stores a Half Word from the specified register to the specified location.
- We have verified it by loading it from the same location to a register.

SB - Store Word

- It stores a Word from the specified register to the specified location.
- We have verified it by loading it from the same location to a register.

B-FORMAT

BEQ

- Load two values into two different registers.

- It will branch if the values are equal.
- Will not branch if they are not equal.

BNE

- Load two values into two different registers.
- It will branch if the values are not equal.
- Will not branch if they are equal.

BLT

- Load two values into registers a4, a5.
- It will branch if a4 is less than a5. Otherwise it will not branch.

BGE

- Load two values into registers a4, a5.
- It will branch if a4 is greater than or equal to a5. Otherwise it will not branch.

BLTU

- Load two values into registers a4, a5.
- It will branch if a4(unsigned) is less than a5(unsigned). Otherwise it will not branch.

BGEU

- Load two values into registers a4, a5.
- It will branch if a4(unsigned) is greater than a5(unsigned). Otherwise it will not branch.

U-FORMAT

LUI

- Taking 20 bit value and it will load them with the upper 20 bits of the specified register.
- We have verified it by loading a 32-bit immediate value to a register. It will first load upper 20 bits into the register and in the next instruction using load imm it loads remaining 12 bits into the register.

AUIPC

- Add imm value and PC and store into the specified register.

J-FORMAT

JAL

- Stores the PC + 4 into the destination register and stores PC + imm into PC.
- We have written a main function and called another function inside it and it uses JAL instruction to jump to the callee.

Modes Used To Verify the Simulator:

Verbose Mode:

In verbose mode it should print the PC and hexadecimal value of each instruction as it's fetched along with the contents (in hexadecimal) of each register after the instruction's execution.

```
---------------------------------------------
Program Counter : 0x1c
Current Instruction : 0x20000593
x[0] (zero) = 0x0
x[1] (ra) = 0x0
x[2] (sp) = 0x10000
x[3] (gp) = 0x0
x[4] (tp) = 0x0
x[5] (t0) = 0x0
x[6] (t1) = 0x0
x[7] (t2) = 0x0
x[8] (s0) = 0x0
x[9] (s1) = 0x0
x[10] (a0) = 0x1
x[11] (a1) = 0x200
x[12] (a2) = 0xa
x[13] (a3) = 0x0
x[14] (a4) = 0x0
x[15] (a5) = 0x0
x[16] (a6) = 0x0
x[17] (a7) = 0x40
x[18] (s2) = 0x0
x[19] (s3) = 0x0
x[20] (s4) = 0x0
x[21] (s5) = 0x0
x[22] (s6) = 0x0
x[23] (s7) = 0x0
x[24] (s8) = 0x0
x[25] (s9) = 0x0
x[26] (s10) = 0x0
x[27] (s11) = 0x0
x[28] (t3) = 0x0
x[29] (t4) = 0x0
x[30] (t5) = 0x0
x[31] (t6) = 0x0
---------------------------------------------
Program Counter : 0x20
Current Instruction : 0xa00613
x[0] (zero) = 0x0
x[1] (ra) = 0x0
x[2] (sp) = 0x10000
x[3] (gp) = 0x0
```

Silent Mode:

In silent mode it should print the PC of the final instruction and hexadecimal value of each register only at the end of the simulation.

```
-----------------RISC-V Simulator---------------------------
Program Counter : 0x40
Current Instruction : 0x8067
x[0] (zero) = 0x0
x[1] (ra) = 0x0
x[2] (sp) = 0x10000
x[3] (gp) = 0x0
x[4] (tp) = 0x0
x[5] (t0) = 0x0
x[6] (t1) = 0x0
x[7] (t2) = 0x0
x[8] (s0) = 0x0
x[9] (s1) = 0x0
x[10] (a0) = 0x0
x[11] (a1) = 0x0
x[12] (a2) = 0x0
x[13] (a3) = 0x0
x[14] (a4) = 0x765
x[15] (a5) = 0x765
x[16] (a6) = 0x0
x[17] (a7) = 0x0
x[18] (s2) = 0x0
x[19] (s3) = 0x0
x[20] (s4) = 0x0
x[21] (s5) = 0x0
x[22] (s6) = 0x0
x[23] (s7) = 0x0
x[24] (s8) = 0x0
x[25] (s9) = 0x0
x[26] (s10) = 0x0
x[27] (s11) = 0x0
x[28] (t3) = 0x0
x[29] (t4) = 0x0
x[30] (t5) = 0x0
x[31] (t6) = 0x0
---------------------------------------------
------------------Simulation Ended here--------------------
```

Debug Mode:
All the memory contents, decoded results, the PC and hexadecimal value of each instruction as it's fetched along with the contents (in hexadecimal) of each register after the instruction's execution.

```
---------------------------------------------
Program Counter : 0x8
funct3: 0
funct7: 16
rs1: 0
rs2: 0
rd: 11
imm: 200
ADDI Detected
Current Instruction : 0x20000593
x[0] (zero) = 0x0
x[1] (ra) = 0x0
x[2] (sp) = 0x10000
x[3] (gp) = 0x0
x[4] (tp) = 0x0
x[5] (t0) = 0x0
x[6] (t1) = 0x0
x[7] (t2) = 0x0
x[8] (s0) = 0x0
x[9] (s1) = 0x0
x[10] (a0) = 0x0
x[11] (a1) = 0x200
x[12] (a2) = 0x0
x[13] (a3) = 0x0
x[14] (a4) = 0x0
x[15] (a5) = 0x0
x[16] (a6) = 0x0
x[17] (a7) = 0x3f
x[18] (s2) = 0x0
x[19] (s3) = 0x0
x[20] (s4) = 0x0
x[21] (s5) = 0x0
x[22] (s6) = 0x0
x[23] (s7) = 0x0
x[24] (s8) = 0x0
x[25] (s9) = 0x0
x[26] (s10) = 0x0
x[27] (s11) = 0x0
x[28] (t3) = 0x0
x[29] (t4) = 0x0
x[30] (t5) = 0x0
x[31] (t6) = 0x0
```