Computer Systems Engineering and Testing

RISC-V Instruction Set Simulator

|  |  |
| --- | --- |
|  | Berindeiu Sergiu Alexandru |
|  | Simion Alexandru George |
| Universitatea Politehnica Timișoara | Stoicov Luka |
| Facultatea de Automatică și Calculatoare | Stolojan Andrei |
| Secția C (Calculatoare) | Tulbure Alexandru Nicolae |
| Anul III | Vekony Carla |

1. Project Objective

The objective of this project is to design and implement a simulator for executing and testing RISC-V assembly instructions. The simulator aims to provide an intuitive and efficient tool for understanding, debugging, and analyzing RISC-V assembly code.

Goals:

* To create a functional and reliable RISC-V assembly instruction simulator
* To facilitate understanding and testing of RISC-V assembly code
* To provide visual feedback by offering a user-friendly interface
* To simulate a virtual memory environment

1. RISC-V instruction set architecture
2. Registers and memory

RISC-V, like many RISC architectures, uses a set of general-purpose registers and special-purpose registers (of which only pc will be used). The combination of these registers and memory management systems provides the foundation for the instruction set's execution.

RISC-V features 32 general-purpose registers (GPRs), each of which is 32-bits wide (in RV32). These registers are used for storing operands for arithmetic and logical operations, as well as intermediate results. In total, there are 32 general-purpose registers (x0 to x31), and they are accessible by all instructions in the ISA. The description of each register is provided in Table 1.

|  |  |  |
| --- | --- | --- |
| Register | Alias | Description |
| x0 | zero | Always holds the value 0. It is hardwired to 0 is not writeable. |
| x1 | ra | Return address, used to store the return address for function calls. |
| x2 | sp | Stack pointer, used for managing the stack during function calls. |
| x3 | pc | Special-purpose register: Holds the address of the next instruction to be executed. It is automatically updated to point to the next instruction, except in cases where control flow instructions alter it. |
| x4 | tp | Thread pointer, used in multi-threaded environments to store a pointer to thread-specific data. |
| x5 – x7 | t0 –t2 | Temporary registers, used for intermediate computations. These registers saved across function calls. |
| x8 – x9 | s0 – s1 | Saved registers, which preserve their values across function calls. |
| x10 – x17 | a0 – a7 | Argument registers, used to pass function arguments and return values. |
| x18 – x27 | s2 – s11 | Saved registers, which preserve their values across function calls. |
| x28 – x31 | t3 – t6 | Temporary registers, used for intermediate computations. These registers saved across function calls. |
| Table 1 | | |

RISC-V follows a **load/store** architecture, meaning that operations on memory only occur through special load (lw, lb, etc.) and store (sw, sb, etc.) instructions. The processor works with data in registers, and data is moved to and from memory via explicit memory operations. Each memory cell holds 32-bits wide instructions/data.

1. Instruction Formats

The RISC-V ISA consists of a base integer instruction set and extensions (e.g., “M” for multiplication and division, “A” for atomic instructions). The ISA uses fixed 32-bit instruction formats, detailed in Table 2.

|  |  |
| --- | --- |
| Format | Description |
| R-type | Register-register operations (e.g., arithmetic, logic) |
| I-type | Register-immediate operations |
| S-type | Store operations |
| L-type | Load operations, CSR operations |
| B-type | Conditional branch operations |
| U-type | Upper immediate operations (used for large immediate values) |
| J-type | Jump operations for function calls or unconditional control flow changes |
| Table 2 | |

R-type fields:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 -27 | 26 - 25 | 24 - 20 | 19 - 15 | 14 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| instr. identif. | instr. identif. | rs2 /shamt | rs1 | instr. identif. | rd | opcode | alignment |

I- type fields:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 - 20 | 19 - 15 | 14 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| Immediate | rs1 | instr. identif. | rd | opcode | alignment |

S-type fields:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 -25 | 24 - 20 | 19 - 15 | 14 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| offset[11:5] | rs2 | rs1 | instr. identif. | offset[4:0] | opcode | alignment |

L-type fields:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 -20 | 19 - 15 | 14 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| offset[11:0] | rs1 | instr. identif. | rd | opcode | alignment |

B-type fields:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 -25 | 24 - 20 | 19 - 15 | 14 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| offset[12|10:5] | rs2 | rs1 | instr. identif. | offset[4:1|11] | opcode | alignment |

U-type fields:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 -12 | 11 - 7 | 6 - 2 | 1-0 |
| immediate[31:12] | rd | opcode | alignment |

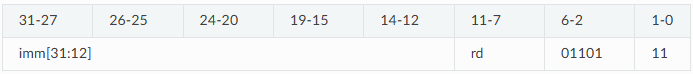
J-type fields:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 -20 | 19 – 12 | 11 - 7 | 6 - 2 | 1-0 |
| offset[11:0]/offset[20|10:1|11] | rs1/ offset[19:12] | rd | opcode | alignment |

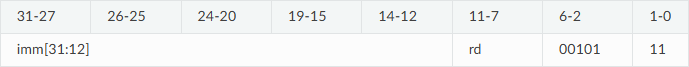
1. Instruction Descriptions
2. RV32I

The **RV32I** base integer instruction set is the fundamental component of the RISC-V architecture for 32-bit systems. It provides the core functionality necessary for building software systems, including arithmetic, control flow, memory access, and system management operations.

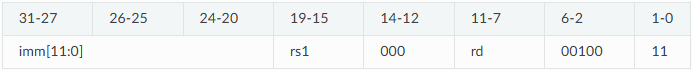
* **lui**: Format: lui rd, imm; Description: places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.



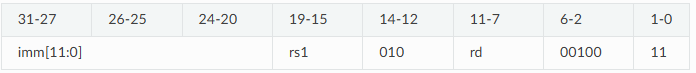
* **auipc**: Format: auipc rd,imm; Description: builds pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.



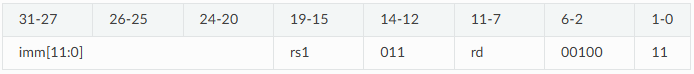
* **addi**: Format: addi rd,rs1,imm; Description: Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.



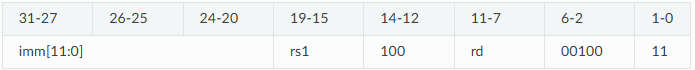
* **slti**: Format: slti rd,rs1,imm; Description: Place the value 1 in register rd if register rs1 is less than the signextended immediate when both are treated as signed numbers, else 0 is written to rd.



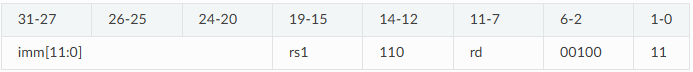
* **sltiu**: Format: sltiu rd,rs1,imm; Description: Place the value 1 in register rd if register rs1 is less than the immediate when both are treated as unsigned numbers, else 0 is written to rd.



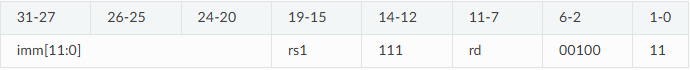
* **xori**: Format: xori rd,rs1,imm; Description: Performs bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd



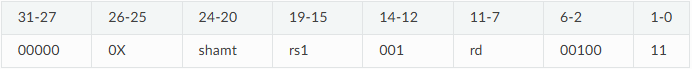
* **ori**: Format: ori rd,rs1,imm; Description: Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd



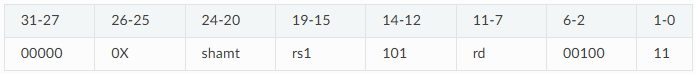
* **andi**: Format: andi rd,rs1,imm; Description: Performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd



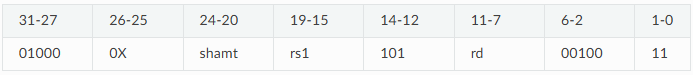
* **slli**: Format: slli rd,rs1,shamt; Description: Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate



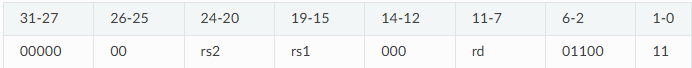
* **srli**: Format: srli rd,rs1,shamt; Description: Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate



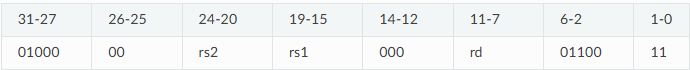
* **srai**: Format: srai rd,rs1,shamt; Description: Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate



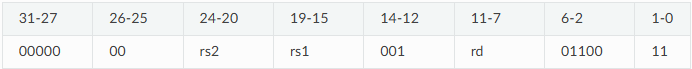
* **add**: Format: add rd,rs1,rs2; Description: Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.



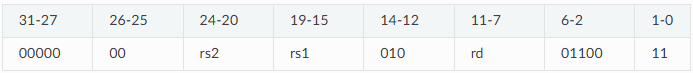
* **sub**: Format: sub rd,rs1,rs2; Description: Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.



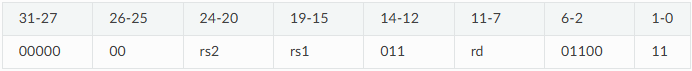
* **sll**: Format: sll rd,rs1,rs2; Description: Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.



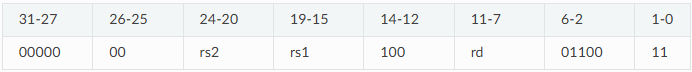
* **slt**: Format: slt rd,rs1,rs2; Description: Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as signed numbers, else 0 is written to rd.



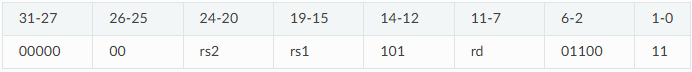
* **sltu**: Format: sltu rd,rs1,rs2; Description: Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as unsigned numbers, else 0 is written to rd.



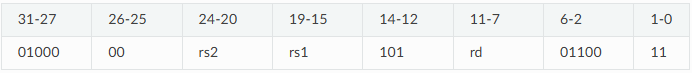
* **xor**: Format: xor rd,rs1,rs2; Description: Performs bitwise XOR on registers rs1 and rs2 and place the result in rd.



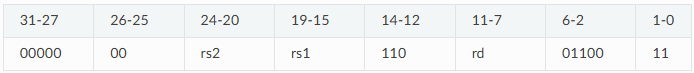
* **srl**: Format: srl rd,rs1,rs2; Description: Logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.



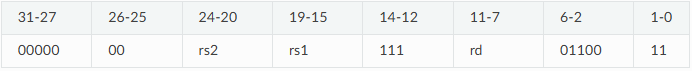
* **sra**: Format: sra rd,rs1,rs2; Description: Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.



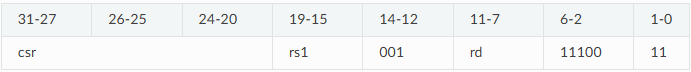
* **or**: Format: or rd,rs1,rs2; Description: Performs bitwise OR on registers rs1 and rs2 and place the result in rd.



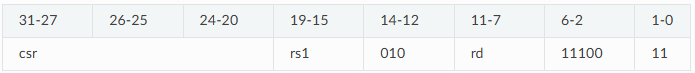
* **and**: Format: and rd,rs1,rs2; Description: Performs bitwise AND on registers rs1 and rs2 and place the result in rd.



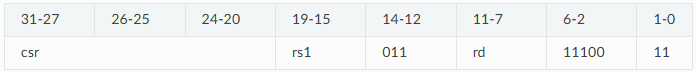
* **csrrw**: Format: csrrw rd,offset,rs1; Description: Atomically swaps values in the CSRs and integer registers.CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR.



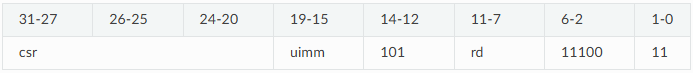
* **csrrs**: Format: csrrs rd,offset,rs1; Description: Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable.



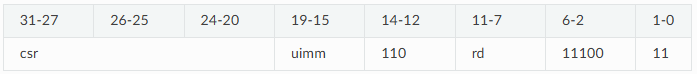
* **csrrc**: Format: csrrc rd,offset,rs1; Description: Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable.



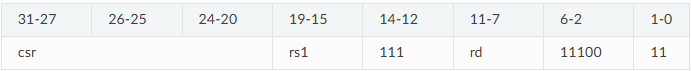
* **csrrwi**: Format: csrrwi rd,offset,rs1; Description: Update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.



* **csrrsi**: Format: csrrsi rd,offset,uimm; Description: Set CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.



* **csrrci**: Format: csrrci rd,offset,uimm; Description: Clear CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.



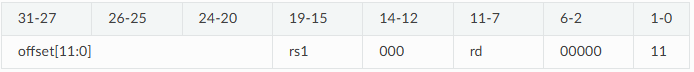
* **ecall**: Format: ecall; Description: Ends the execution of the program



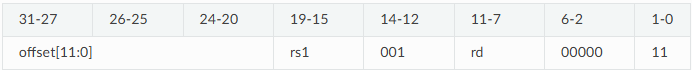
* **ret**: Format: ret; Description: PC is set to the value of RA, all temporary registers are lost.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| 10000 | 00 | 00010 | 00000 | 000 | 00000 | 11100 | 11 |

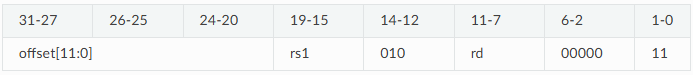
* **lb**: Format: lb rd,offset(rs1); Description: Loads a 8-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.



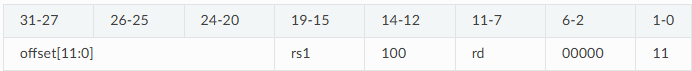
* **lh**: Format: lh rd,offset(rs1);Description: Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.



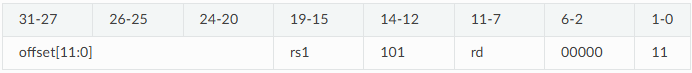
* **lw**: Format: lw rd,offset(rs1);Description: Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.



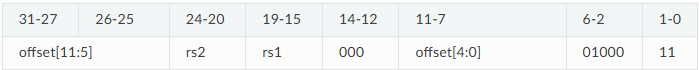
* **lbu**: Format: lbu rd,offset(rs1);Description: Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.



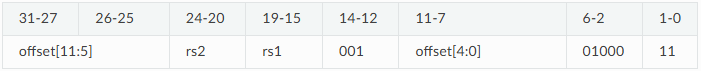
* **lhu**: Format: lhu rd,offset(rs1);Description: Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.



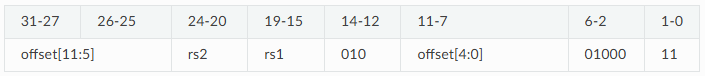
* **sb**: Format: sb rs2,offset(rs1); Description: Store 8-bit, values from the low bits of register rs2 to memory.



* **sh**: Format: sh rs2,offset(rs1); Description: Store 16-bit, values from the low bits of register rs2 to memory.



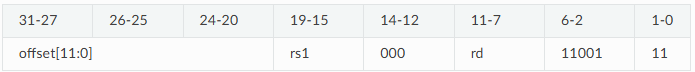
* **sw**: Format: sw rs2,offset(rs1); Description: Store 32-bit, values from the low bits of register rs2 to memory.



* **jal**: Format: jal rd,offset; Description: Jump to address and place return address in rd. The value of all temporary registers is lost.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[19:0] | | | | | rd | 11011 | 11 |

* **jalr**: Format: jalr rd,rs1,offset; Description: Jump to address and place return address in rd. The value of all temporary registers is lost.



* **beq**: Format: beq rs1,rs2,offset; Description: Take the branch if registers rs1 and rs2 are equal.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 000 | offset[4:0] | 11000 | 11 |

* **bne**: Format: bne rs1,rs2,offset; Description: Take the branch if registers rs1 and rs2 are not equal.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 001 | offset[4:0] | 11000 | 11 |

* **blt**: Format: blt rs1,rs2,offset; Description: Take the branch if registers rs1 is less than rs2, using signed comparison.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 100 | offset[4:0] | 11000 | 11 |

* **bge**: Format: bge rs1,rs2,offset; Description: Take the branch if registers rs1 is greater than or equal to rs2, using signed comparison.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 101 | offset[4:0] | 11000 | 11 |

* **bltu**: Format: bltu rs1,rs2,offset; Description: Take the branch if registers rs1 is less than rs2, using unsigned comparison.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 110 | offset[4:0] | 11000 | 11 |

* **bgeu**: Format: bgeu rs1,rs2,offset; Description: Take the branch if registers rs1 is greater than or equal to rs2, using unsigned comparison.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| offset[11:5] | | rs2 | rs1 | 111 | offset[4:0] | 11000 | 11 |

* **push**: Format: push rd; Description: Stores rd in memory at the current location of SP and updates SP.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| 01000 | 00 | 00000 | 00000 | 110 | rd | 11100 | 11 |

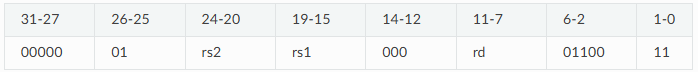
* **pop**: Format: pop rd; Description: Stores in rd what is in memory at the current location of SP and updates SP.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31-27 | 26-25 | 24-20 | 19-15 | 14-12 | 11-7 | 6-2 | 1-0 |
| 01000 | 00 | 00000 | 00000 | 111 | rd | 11100 | 11 |

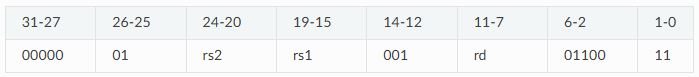
1. RV32M

The **RV32M** extension adds integer multiplication and division operations to the base **RV32I** instruction set. This extension is optional and is typically used in applications that require high-performance numerical computation.

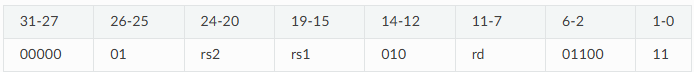
* **mul**: Format: mul rd,rs1,rs2; Description: performs an XLEN-bit × XLEN-bit multiplication of signed rs1 by signed rs2 and places the lower XLEN bits in the destination register.



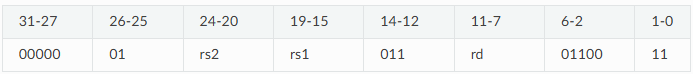
* **mulh**: Format: mulh rd,rs1,rs2; Description: performs an XLEN-bit × XLEN-bit multiplication of signed rs1 by signed rs2 and places the upper XLEN bits in the destination register.



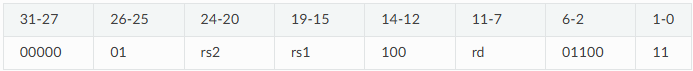
* **mulhsu**: Format: mulhsu rd,rs1,rs2; Description: performs an XLEN-bit × XLEN-bit multiplication of signed rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.



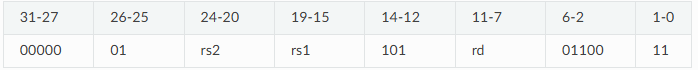
* **mulhu**: Format: mulhu rd,rs1,rs2; Description: performs an XLEN-bit × XLEN-bit multiplication of unsigned rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.



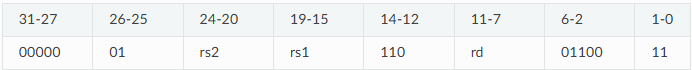
* **div**: Format: div rd,rs1,rs2; Description: perform an XLEN bits by XLEN bits signed integer division of rs1 by rs2, rounding towards zero.



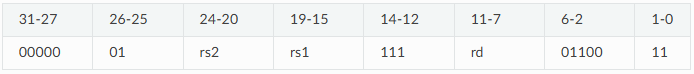
* **divu**: Format: divu rd,rs1,rs2; Description: perform an XLEN bits by XLEN bits unsigned integer division of rs1 by rs2, rounding towards zero.



* **rem**: Format: rem rd,rs1,rs2; Description: perform an XLEN bits by XLEN bits signed integer reminder of rs1 by rs2.



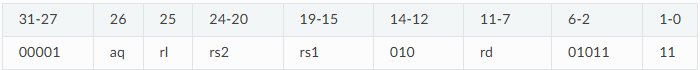
* **remu**: Format: remu rd,rs1,rs2; Description: perform an XLEN bits by XLEN bits unsigned integer reminder of rs1 by rs2.



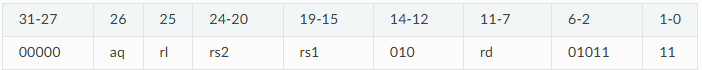
1. RV32A

The **RV32A** extension, also known as the Atomic extension, introduces instructions to support atomic memory operations. These operations are crucial for implementing synchronization mechanisms in multi-threaded or multi-core systems, allowing safe and efficient communication between concurrent processes.

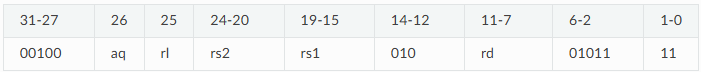
* **amoswap.w**: Format: amoswap.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, swap the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



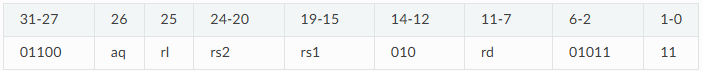
* **amoadd.w**: Format: amoadd.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply add the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



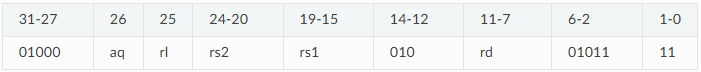
* **amoxor.w**: Format: amoxor.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply exclusive or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



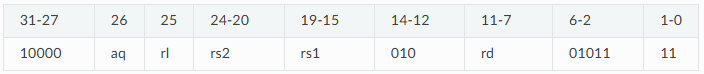
* **amoand.w**: Format: amoand.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply and the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



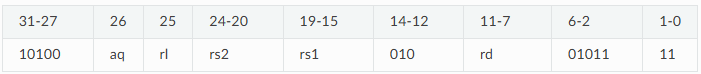
* **amoor.w**: Format: amoor.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



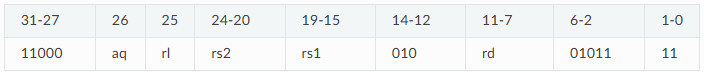
* **amomin.w**: Format: amomin.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply min operator the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



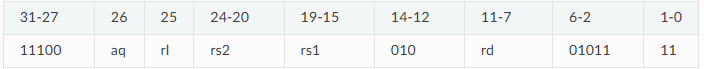
* **amomax.w**: Format: amomax.w rd,rs2,(rs1); Description: atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply max operator the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.



* **amominu.w**: Format: amominu.w rd,rs2,(rs1); Description: atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned min the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.



* **amomaxu.w**: Format: amomaxu.w rd,rs2,(rs1); Description: atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned max the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.



1. Code format
2. Instruction structure

[<label>:] [<instruction / operation> [<operands>]] [; <comments>]

* Label: marks a specific point in the program and is often used as a reference for branching or jumping. The label is followed by a colon (: ).
* Instruction/operation: This is represented by the mnemonic of the operation or command to be executed, such as add, sub and jal.
* Operands: These are the arguments or data on which the instruction operates. Operands can be: registers, memory addresses or immediate values.
* Comments: provide explanations for the code and are ignored by the assembler. Comments start with a semicolon (;) and last for the rest of the row.

1. Code structure

Mock code example:

.data ; 0

num1: .word 0x5 ; 1

num2: .word 0x10 ; 2

result: .word 0x0 ; 3

.code ; 4

\_start: lw t1, 0x0(t0) ; 5

lw t2, 0x0(t0) ; 6

add t3, t1, t2 ; 7

sw t3, 0x0(t0) ; 8

loop: beq t3, a0, end ; 9

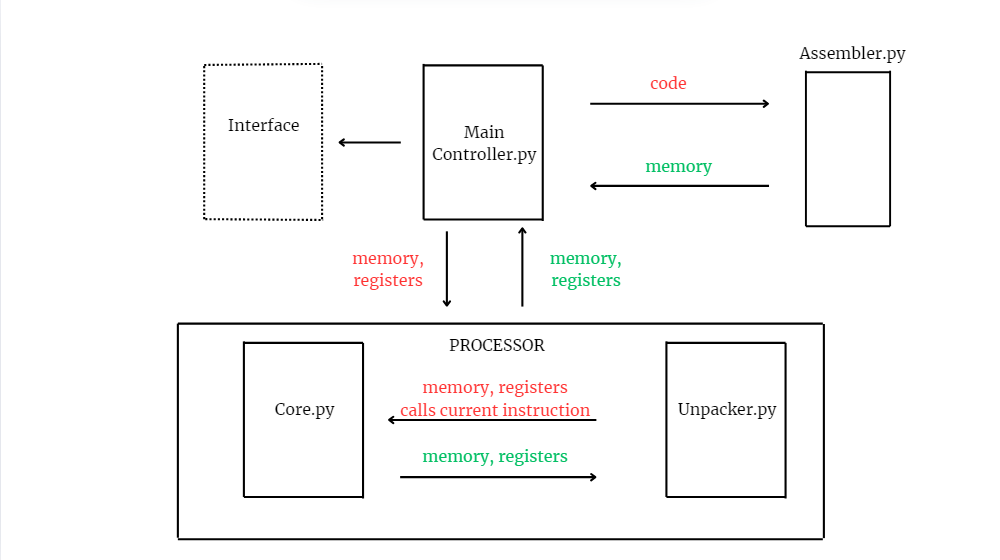
jal a0, loop ; 10

end: addi a7, zero, num1 ; 11

ecall ; 12

Rules and conventions:

1. All immediate values will be specified in hexadecimal format, prefixed with '0x';
2. The simulator is not case-sensitive;
3. The code is divided into two segments: .data and .code. The .data section initializes constants stored in memory and only the .word instruction can be used. The .code section contains the executable instructions of the program;
4. All labels must be followed by an instruction;
5. Program flow



1. Implementation
2. MainController.py

This component serves as the graphical user interface (GUI) for the simulator and is implemented using Python's tkinter library. It is responsible for generating the interface and acting as the primary driver for the processor's execution.

The GUI provides users with an intuitive way to interact with the simulator, allowing them to input RISC-V assembly code, execute it step-by-step, and visualize the results in real time, as it dynamically displays the processor's state, including registers and memory, and highlights the currently executed instruction for better debugging and understanding.

1. Assembler.py

The Assembler is responsible for parsing and processing the RISC-V code and mapping it to a memory structure. It handles the extraction of labels and operations from each line of code, identifying instructions, and parsing their parameters. It also processes each instruction and stores the corresponding values in memory.

1. Unpacker.py

This module contains a set of functions that play a critical role in decoding and interpreting the instructions for the processor simulation. It takes raw instruction data from memory, decodes it into a more manageable format, and extracts relevant fields such as opcodes, registers, immediate values, and function codes. These functions help parse the instruction format and prepare the necessary information for execution, enabling the processor to correctly interpret and execute different types of operations such as arithmetic, branching, and memory access.

1. Core.py

The core module implements the actual behavior of the processor by executing decoded instructions and updating the processor’s state. This includes performing arithmetic operations (addition, subtraction, multiplication, division), managing memory (loading and storing data), and handling control flow (branches, jumps). It manipulates the processor’s registers, memory, and program counter as needed, ensuring the correct execution of each instruction. Additionally, it supports complex operations such as atomic memory operations and system calls. The core module simulates the essential functionality of a processor.

1. Testing
2. Test\_branch.py

This script implements unit tests for RISC-V-inspired branch and jump instructions using Python's built-in unit test library. It simulates 32 registers and a simple memory structure. The tested functions include jal, jalr, beq, bne, blt, bge, bltu, and bgeu, which handle conditional and unconditional jumps based on register values and offsets. Helper functions like extend\_sign and signed2unsigned are used for handling signed and unsigned values.

1. Test\_LoadStore.py

This script focuses on testing RISC-V-inspired load and store instructions using Python’s unittest framework. It simulates a set of 32 registers alongside a basic memory structure. The test cases cover operations such as lb, lh, lw, lbu, lhu, sb, sh, and sw, which manage the transfer of data between registers and memory. To facilitate proper handling of data sizes, helper functions like sign\_extend and bit\_truncate are used, ensuring accurate sign extension and bit-width manipulation for byte, halfword, and word operations across the instructions.

1. Testing Process After Unit Testing

After completing unit testing for the RISC instruction simulator, additional testing stages were conducted to ensure system reliability and functionality. Integration testing focused on verifying how individual modules interact when combined. Various test scenarios were designed to confirm that data is packed and unpacked correctly, values and opcodes are processed as expected, and the overall sequence of functions works seamlessly. This step helped identify and resolve issues in the communication between components. System testing was then performed to evaluate the simulator as a complete system. Full instruction sets were simulated, including edge cases and uncommon scenarios, to verify that the simulator produced correct outputs and met all functional and performance requirements. Finally, black-box testing validated the system externally, without focusing on internal implementation details. Input instructions were provided to the simulator, and the outputs were compared against expected results to ensure correct execution and behavior under various conditions. These steps ensured that the RISC instruction simulator performs as intended, both in individual functions and as an integrated system.

1. External Resources
2. Bibliography

* Instruction set: “<https://msyksphinz-self.github.io/riscv-isadoc/html/rvm.html#mulh>”
* Instruction set reference for more detailed explanations : <https://marks.page/riscv/>
* Specific questions regarding RISC-V ISA: <https://stackoverflow.com/questions/71374202/why-mov-instruction-is-replaced-by-add-instruction/>
* .code section implementation: “https://hackmd.io/@hauhsu/rJ89ghF6n?utm\_source=preview-mode&utm\_medium=rec”

1. Inspiration

* LMC Simulator: “<https://peterhigginson.co.uk/LMC>/”
* RISC Simulator: “<https://peterhigginson.co.uk/RISC/>”
* RISC-V Simulator: “https://webriscv.dii.unisi.it/index.php/”

1. GitHub Repository

* “https://github.com/AlexSimionGeorge/FinalProj\_ITSC”