ECE 586
Simulated RISC-V ISA Final Report

Team 2

Portland State University

Maseeh College of Engineering & Computer Science

Department of Electrical & Computer Engineering

Authors:

Kane-Pardy, Chris

Monti, Violet

March 18, 2025

ECE 586

Computer Architecture

Contents

1 Introduction

This project simulates the RV32I ISA, including instructions and user-visible state. The user-visible state includes the general purpose register file (with the 32 registers defined by RISC-V conventions), the program counter, which is incremented by the word size, and used for our fetch and decode loop, and the data memory (which has an assumed maximum size of 64 KiB). The instructions are broken down properly according to their corresponding instruction format, which is determined by their opcode.

The simulation features two operating modes: Silent Mode (mode 0) and Normal (Verbose) Mode (mode 1). Silent Mode prints only the PC of the final instruction and the value (in hexadecimal) of every register, after a `.mem` file has been read. Normal Mode prints the PC, the instruction (in hexadecimal), and the register file after each instruction's execution. Single-Step Mode (set by a separate runtime argument) provides the option for user input. Depending upon the entered value, the user can print any or all integer or floating point registers, all memory entries, or a single memory address and its contents, as well as the current instruction. By default, the program operates in Silent Mode (`-m 0`) but can be switched to Normal Mode by passing the argument (`-m 1`) or Single-Step Mode (`-step 1`) at runtime. Aside from the three standard modes, functionality for setting breakpoints and "watching" registers or memory locations exists. Breakpoints can be toggled on using the (`-bp 1`) runtime argument, and "watching" with the (`-w 1`) runtime argument, so that they can be used in any mode.

The simulation requires a `.mem` file to operate, which must be of the format `< PC Value/Memory Address >:< Instruction in Hexadecminal >`. It can be specified with the `-f <filename>` argument. If no file is provided or the specified file is invalid, the program uses a default file. Should neither the provided nor the default file be available, the program will terminate with an error. Additional runtime arguments are (`-p`) to declare the starting memory address of the stack pointer (which defaults to 65536), and (`-s`) which is the starting value for the PC (which

defaults to 0).

Outside of RV32I, we opted to attempt all of the extra credit opportunities. We added the RV32M and RV32F / D extensions, and the `ecall` instruction. All necessary additions to our simulation to accommodate these (registers, instruction formats, etc) are also all included within our simulation.

This simulation successfully mimics and recreates the aforementioned aspects of the RISC-V ISA, including the instruction set and user-visible state. Through the base project, added extensions, and additional debugging features we've added, we've gained a strong understanding of the conventions, capability, and functionality of RISC-V.

2 Internal Design

To read our input file, update memory, and execute and decode instructions, we split our code into numerous different functions. The memory array, declared as a static `uint32_t` array, consists of `MemWords` elements, where `MemWords` is computed as `(int) (pow(2, MEMORY_SIZE) / 32)`, where `MEMORY_SIZE` is the desired size of the data portion of memory in bits. Our integer register file is also a static `uint32_t` array, with 32 members, and our floating point register file is a `float` array with 32 members. The only registers we initialize before execution are x0 (`zero`), x1(`ra`), and x2(`sp`), all of which are initialized to 0x0. The program counter (PC) is managed as a `uint32_t` variable, incremented appropriately with each instruction execution.

We implemented runtime arguments to specify filename (`-f <filename>`), operation mode (`-m <mode>` (0 = silent, 1 = verbose)), single-step operation (`-step 1`), register/memory watching (`-w 1`), breakpoints (`-bp 1`), starting stack pointer (`-p`), and starting pc value/starting memory address (`-s`). After processing these arguments, the simulator reads the input `.mem` file using a `fscanf` loop and aligns instructions to 4-byte memory addresses.

The core execution loop repeatedly calls the `fetch_and_decode` function to retrieve the next instruction. This function extracts the opcode and returns it to the main loop, where a switch-case structure determines the corresponding instruction type. Based on the decoded instruction, the appropriate handler function is called. Each instruction type follows a structured execution process:

R-type (Register-based operations): These instructions operate between 3 registers and include arithmetic, logical, and shift operations. The R-format breaks instructions into opcode, destination register (rd), function code (func3), two source registers (rs1 and rs2), and an additional function code (func7). The execution involves extracting the operands from registers, performing the operation defined by func3 and func7, which is determined by nested switch cases, and then placing the result back into the destination register in our integer register array. Example in-

structions include ADD, SUB, XOR, OR, AND. The RV32M extension operations like MUL are also register format operations, which we fully added. They have identical function 3 bits to many of the regular register ops, but differ in their function 7 values. The same integer registers are also used for these operations.

I-type (Immediate operations and loads): These instructions use an immediate value instead of a second source register. The function for I-format breaks the instruction into an opcode, destination register (rd), function code (func3), a single source register (rs1), and a 12-bit immediate value (imm). The immediate value is also sign-extended. The execution involves performing the specified operation (such as ADDI, XORI, ORI, ANDI) using rs1 and the immediate, storing the result in rd. Special cases of I-type instructions include:

- **JALR (Jump and Link Register)**: This instruction updates the PC to an address computed as `(rs1 + imm) & 0xFFFFFFFE`, ensuring the least significant bit is zero. It also stores the return address (PC + 4) in the destination register, allowing it to be used for function returns.

- **ECALL (Environment Call)**: This system call instruction triggers an environment call based on the value stored in register x17. Depending on the system call type (read, write, or exit), appropriate system functions are invoked.

- **Load Instructions (LW, LH, LB, LHU, LBU)**: These instructions retrieve values from memory using displacement addressing (computed as `rs1 + imm`). The memory is accessed using `readWord()`, `readHalfWord()`, or `readByte()` functions, which are the only way to read memory in our code, ensuring proper alignment and handling of sign extension for signed loads.

S-type (Store operations): Used for storing data into our memory array, this format computes an opcode, two source registers (rs1, rs2), a function code (func3), and a split immediate value (imm[11:5] and imm[4:0]). The immediate value is combined to form a full 12-bit signed offset. The execution process calculates the memory address using rs1 and the immediate, then stores the value from rs2 into that memory location using `writeWord()`, `writeHalfWord()`, or `writeByte()`, which ensure

proper access alignment and update only the required portion of memory. These functions are the only place in our code that can write to the memory array, ensuring no improper modification takes place.

B-type (Branch operations): These instructions compute branches (PC jumps) based on comparisons between 2 registers. The b-format includes an opcode, two source registers (rs1, rs2), function code (func3), and a split immediate value, which is sign-extended and left-shifted to form the final branch target offset (in instructions). Execution compares rs1 and rs2 based on the condition specified by func3 (BEQ for equality, BNE for inequality, BLT for less than, etc.). If the condition is met, the PC is updated to the branch target address; otherwise, it proceeds sequentially.

U-type (Upper Immediate operations): These two instructions are used to construct large constants (immediates) or to modify the PC. The format consists of an opcode, a destination register (rd), and a 20-bit immediate value that is left-shifted by 12 bits before being used. LUI (Load Upper Immediate) stores this value directly in rd, while AUIPC (Add Upper Immediate to PC) adds it to the current PC before storing it.

J-type (Jump operations): Used for the jump and link (jal) instruction, this format includes an opcode, destination register (rd), and a 20-bit immediate value. The immediate is sign-extended, left-shifted, and added to the PC to compute the jump target. The rd register stores the return address (PC + 4).

Floating-Point Instructions: The simulator fully supports RISC-V floating-point operations, adhering to IEEE-754 single-precision. Instructions are categorized into arithmetic, comparison, type conversion, and memory transfer. We have several functions for each type, which breaks down the instruction depending on the operation that's meant to be performed. Arithmetic operations such as FADD.S, FSUB.S, FMUL.S, and FDIV.S perform computations on floating-point registers, applying the selected rounding mode (RNE, RTZ, RDN, RUP, or RMM) where applicable.

Comparison instructions like FEQ.S, FLT.S, and FLE.S set integer registers based on floating-point comparisons, and correctly handle NaN values. Type conversion operations, such as FCVT.S.W and FCVT.W.S, handle integer-float conversions and adhere to sign extension and rounding. Floating-point memory operations FLW and FSW use the simulator's `readWord()` and `writeWord()` functions, properly handle memory alignment and directly manipulate the bit-level representation of floating-point values in memory.

Debugging and Execution Control: The simulator provides extensive debugging features, including:

- **Verbose Mode**: When enabled with `-m 1`, the program prints register and memory states after every instruction.
- **Single-Step Execution**: Enabled with `-step 1`, this allows execution to halt after each instruction, requiring user input to proceed, and providing the option to view memory contents, register contents, and the current instruction.
- **Breakpoints**: When `-bp 1` is set, users can specify PC values where execution will pause, allowing manual inspection of register and memory states.
- **Location Watching (Memory/Register)**: The `-w 1` argument enables monitoring specific registers or memory locations, printing their values after each instruction executes.

The simulator successfully models the RISC-V ISA, ensuring correct execution of all RV32I, RV32M, and RV32F / D instructions, while providing debugging support and monitoring tools to help analyze execution behavior.

3 Simulation Results

Our testing methodology for this project is to use Case Analysis for every single instruction to ensure no bugs are within the program. Our addressing formats are each tested with debug statements when our debug mode is active, which ensures the proper decoding of the instruction based on the opcode. Below is a heavily simplified list of the tests we ran, the full test cases can be found in our ECE 586 RISC-V ISA Test Plan (Team #2) document.

Register Format

Purpose: To ensure that instructions are broken down properly into their components, registers are properly retrieved from the register file, computations are completed as expected, sign extension works, and results are stored in the correct location.

- **add**
  - Positive to Positive (Positive Result)
  - Positive to Negative (Negative Result)
  - Positive to Negative (Positive Result)
  - Negative to Negative (Negative Result)
- **sub**
  - Positive from Positive (Positive Result)
  - Positive from Positive (Negative Result)
  - Positive from Negative (Positive Result)
  - Positive from Negative (Negative Result)
  - Negative from Positive (Positive Result)
  - Negative from Negative (Positive Result)
  - Negative from Negative (Negative Result)
- **xor**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's
  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)

- **or**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's
  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)
- **and**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's
  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)
- **sll**
  - sll on all 0's
  - sll on all 1's
  - sll on 1's and 0's
  - Maximum sll
- **srl**
  - srl on all 0's
  - srl on all 1's
  - srl on 1's and 0's
  - Maximum srl
- **sra**
  - sra on all 0's
  - sra on all 1's
  - sra on 1's and 0's
  - Maximum sra
- **slt**
  - slt negative vs positive (always true)
  - slt negative vs negative (true)
  - slt negative vs negative (false)
  - slt positive vs positive (true)
  - slt positive vs positive (false)
  - slt positive vs negative (always false)
  - slt equal (always false)

- **sltu**
  - sltu negative vs positive (always false)
  - sltu negative vs negative (true)
  - sltu negative vs negative (false)
  - sltu positive vs positive (true)
  - sltu positive vs positive (false)
  - sltu positive vs negative (always true)
  - sltu equal (always false)

Results: Instructions are accurately broken down into their components, both source register values are properly retrieved from the register file, computations are completed as expected and accurately, results are sign extended accurately, and stored in the correct location in our register array.

Immediate Format (Operations)

Purpose: To ensure that instructions are broken down properly into their components, the source register is properly retrieved from the register file, computations are completed as expected, sign extension of the immediate and the results work, and results are stored in the destination register properly.

- **addi**
  - Positive to 0 (Positive Result)
  - Positive to 0 (Negative Result)
  - 0 to 0
- **xori**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's
  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)
- **ori**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's

  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)

- **andi**
  - 0's with 0's
  - 1's with 1's
  - 1's with 0's
  - 1's with 1's and 0's
  - 1's with 1's and 0's with sign extension (MSB 0)
  - 1's with 1's and 0's with sign extension (MSB 1)

- **slli**
  - slli on all 0's
  - slli on all 1's
  - slli on 1's and 0's
  - Maximum slli

- **srli**
  - srli on all 0's
  - srli on all 1's
  - srli on 1's and 0's
  - Maximum srli

- **srai**
  - srai on all 0's
  - srai on all 1's
  - srai on 1's and 0's
  - Maximum srai

- **slti**
  - slti negative vs positive (always true)
  - slti negative vs negative (true)
  - slti negative vs negative (false)
  - slti positive vs positive (true)
  - slti positive vs positive (false)
  - slti positive vs negative (always false)
  - slti equal (always false)

- **sltiu**
  - sltiu negative vs positive (always false)
  - sltiu negative vs negative (true)
  - sltiu negative vs negative (false)

  - sltiu positive vs positive (true)
  - sltiu positive vs positive (false)
  - sltiu positive vs negative (always true)
  - sltiu equal (always false)

Results: The instructions are accurately broken down into their components, the source register value is properly retrieved from the register file, the immediate is properly extracted and sign extended, computations are completed as expected and accurately, results are sign extended properly, and they are stored in the destination register.

Immediate Format (Loads)

Purpose: Ensure that the memory address from the source register properly indexes into our memory array, the offset is computed properly, the memory contents are retrieved and loaded into the destination register. For non-word loads, we also want to make sure misaligned attempted accesses are disallowed, and that the result is accurately sign extended.

- **lb**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 2 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 0 (byte 4 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - MSB 1 (byte 4 of 4)
- **lh**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 2 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 0 (byte 4 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - MSB 1 (byte 4 of 4)

- **lw**
  - MSB (byte 1 of 4)
  - MSB (byte 2 of 4)
  - MSB (byte 3 of 4)
  - MSB (byte 4 of 4)
- **lbu**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 2 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 0 (byte 4 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - MSB 1 (byte 4 of 4)
- **lhu**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 2 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 0 (byte 4 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - MSB 1 (byte 4 of 4)

Results: The memory array is indexed by the source register, including the offset. The destination register receives the correct contents from memory, so long as the address is aligned properly, and results are sign extended when necessary.

Immediate Format (JALR)

Purpose: Ensure the immediate is properly extracted and sign extended, is added to the PC, the (pre-addition) PC (+ 4) is stored in the destination register.

- **JALR**
  - Basic Functionality
  - LSB → Zero Verification
  - Return to Previous PC

Results: The immediate is properly extracted, sign extended, and added to the PC, PC + 4 is stored in the destination register.

Store Format

Purpose: Ensure that the memory address from the destination register properly indexes into our memory array, the offset is computed properly, the memory contents are stored from the source register. For non-word stores, we also want to make sure misaligned attempted accesses are disallowed.

- **sb**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 2 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 0 (byte 4 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - MSB 1 (byte 4 of 4)
- **sh**
  - MSB 0 (byte 1 of 4)
  - MSB 0 (byte 3 of 4)
  - MSB 1 (byte 1 of 4)
  - MSB 1 (byte 3 of 4)
  - Misaligned Reference (byte 2 of 4)
  - MSB 1 (byte 3 of 4)
  - Misaligned Reference (byte 4 of 4)
- **sw**
  - Correct Reference (byte 1 of 4)
  - Misaligned Reference (byte 2 of 4)
  - Misaligned Reference (byte 3 of 4)
  - Misaligned Reference (byte 4 of 4)

Results: The memory address from the destination register properly indexes into our memory array, the offset is computed properly, and the memory contents are stored from the source register. Misaligned accesses are disallowed.

Branch Format

Purpose: To ensure that instructions are broken down properly into their components, branch offsets are properly added to the PC, source registers are retrieved from the register file, and comparisons are properly tested.

- **beq**
  - Equal (Taken Forwards)
  - Equal (Taken Backwards)
  - Not Equal
  - Equal but Opposite Signs (Not Equal)
- **bne**
  - Equal
  - Equal but Opposite Signs (Not Equal)
  - Not Equal (Taken Forwards)
  - Not Equal (Taken Backwards)
- **blt**
  - less than (positive & positive)
  - less than (negative & positive)
  - less than (negative & negative)
  - Equal (both positive)
  - Equal (both negative)
  - greater than (positive & positive)
  - greater than (negative & positive)
  - greater than (negative & negative)
- **bge**
  - less than (positive & positive)
  - less than (negative & positive)
  - less than (negative & negative)
  - Equal (both positive)
  - Equal (both negative)
  - greater than (positive & positive)
  - greater than (negative & positive)
  - greater than (negative & negative)
- **bltu**
  - less than (positive & positive)
  - less than (negative & positive)

  - less than (negative & negative)
  - Equal (both positive)
  - Equal (both negative)
  - greater than (positive & positive)
  - greater than (negative & positive)
  - greater than (negative & negative)

- **bgeu**
  - less than (positive & positive)
  - less than (negative & positive)
  - less than (negative & negative)
  - Equal (both positive)
  - Equal (both negative)
  - greater than (positive & positive)
  - greater than (negative & positive)
  - greater than (negative & negative)

Results: Branch offsets are properly added to the PC for branches taken both forwards and backwards, source registers are retrieved from the register file properly, and comparisons are accurate.

Upper Format

Purpose: Ensure that the correct pc is used, instructions are properly broken down, and the immediate is accurately constructed.

- **auipc**
  - Positive Immediate
  - Negative Immediate
  - Immediate Value of 0

- **lui**
  - Positive Immediate
  - Negative Immediate
  - Immediate Value of 0

Results: The PC value grabbed is accurate, instructions are properly broken down, the immediate is accurately constructed, the PC and the destination register both receive results accurately.

Jump Format

Purpose: Ensure the immediate is properly extracted, assembled, and sign extended. Make sure the correct PC is grabbed, and the immediate is properly added. Make sure the (pre-addition) PC (+ 4) is stored in the destination register.

- **jal**
  - Positive Jump
  - Negative Jump
  - Loop
  - Max Forward Jump
  - Max Backward Jump

Results: The immediate is properly rearranged, sign extended, and added to the PC correctly. PC + 4 is stored in the destination register.

Multiply Extension

Purpose: Using the register format, the correct multiply functions should be selected. Instructions should be broken down properly into their respective components, registers should be properly retrieved from the register file, computational results should be completed as expected, sign extension works, and results are stored in the correct location in the register file.

- **mul**
  - Positive by Positive (small #'s) (positive sign extension)
  - Positive by Negative (small #'s) (negative sign extension)
  - Negative by Positive (small #'s) (negative sign extension)
  - Negative by Negative (small #'s) (positive sign extension)
  - Multiplication by 0
  - "Overflow"
- **mulh**
  - Positive by Positive (positive sign extension)
  - Positive by Negative (negative sign extension)
  - Negative by Positive (negative sign extension)
  - Negative by Negative (positive sign extension)
  - Multiplication by 0

- **mulhsu**
  - Positive by Positive (actually positive) (positive sign extension)
  - Positive by Positive (actually negative) (positive sign extension)
  - Negative by Positive (actually positive) (negative sign extension)
  - Negative by Positive (actually negative) (negative sign extension)
  - Multiplication by 0
- **mulhu**
  - Positive (actually positive) by Positive (actually positive)
  - Positive (actually positive) by Positive (actually negative)
  - Positive (actually negative) by Positive (actually positive)
  - Positive (actually negative) by Positive (actually negative)
  - Multiplication by 0
- **div**
  - Positive by Positive
  - Positive by Negative
  - Negative by Positive
  - Negative by Negative
  - Divide by 1
  - Divide by -1
  - Divide by 0
  - Rounding
- **divu**
  - Positive by Positive
  - Positive by Negative
  - Negative by Positive
  - Negative by Negative
  - Divide by 1
  - Divide by -1
  - Divide by 0
  - Rounding
- **rem**
  - Positive by Positive
  - Positive by Negative
  - Negative by Positive
  - Negative by Negative
  - Rem 1
  - Rem -1
  - Rem 0

- **remu**
  - Positive by Positive
  - Positive by Negative
  - Negative by Positive
  - Negative by Negative
  - Rem 1
  - Rem -1
  - Rem 0

Results: The correct multiply functions are selected and broken down. Registers are retrieved and stored properly, with the expected result.

Floating Point Extension

Purpose: Floating point uses several of its own different instruction formats based upon the address, each with a different breakdown. We want to ensure that our functions created to implement those properly break down the addresses, index the floating point registers, integer registers, memory array, and compute and store results accurately.

- **flw**
  - Correct reference (byte 1 of 4)
  - Misaligned reference (byte 2 of 4)
  - Misaligned reference (byte 3 of 4)
  - Misaligned reference (byte 4 of 4)
- **fsw**
  - Correct Reference (byte 1 of 4)
  - Misaligned Reference (byte 2 of 4)
  - Misaligned Reference (byte 3 of 4)
  - Misaligned Reference (byte 4 of 4)

- **fmadd.s**
  - All Positive
  - Multiply Positive Add Negative
  - Multiply Positive & Negative (way 1) Add Negative
  - Multiply Positive & Negative (way 1) Add Positive
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Positive
  - Multiply Negative Add Negative
  - Multiply Negative Add Positive
  - Multiply Zero Add Positive
  - Multiply Zero Add Negative
  - Multiply Zero Add Zero
- **fmsub.s**
  - All Positive
  - Multiply Positive Sub Negative
  - Multiply Positive & Negative (way 1) Sub Negative
  - Multiply Positive & Negative (way 1) Sub Positive
  - Multiply Positive & Negative (way 2) Sub Negative
  - Multiply Positive & Negative (way 2) Sub Negative
  - Multiply Positive & Negative (way 2) Sub Positive
  - Multiply Negative Sub Negative
  - Multiply Negative Sub Positive
  - Multiply Zero Sub Positive
  - Multiply Zero Sub Negative
  - Multiply Zero Sub Zero
- **fnmadd.s**
  - All Positive
  - Multiply Positive Add Negative
  - Multiply Positive & Negative (way 1) Add Negative
  - Multiply Positive & Negative (way 1) Add Positive
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Positive
  - Multiply Negative Add Negative
  - Multiply Negative Add Positive
  - Multiply Zero Add Positive
  - Multiply Zero Add Negative
  - Multiply Zero Add Zero

- **fnmsub.s**
  - All Positive
  - Multiply Positive Add Negative
  - Multiply Positive & Negative (way 1) Add Negative
  - Multiply Positive & Negative (way 1) Add Positive
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Negative
  - Multiply Positive & Negative (way 2) Add Positive
  - Multiply Negative Add Negative
  - Multiply Negative Add Positive
  - Multiply Zero Add Positive
  - Multiply Zero Add Negative
  - Multiply Zero Add Zero
- **fadd.s**
  - Positive to Positive
  - Positive to Negative (Way 1)
  - Positive to Negative (Way 2)
  - Negative to Negative
  - Add 0
- **fsub.s**
  - Positive from Positive
  - Positive from Negative (Way 1)
  - Positive from Negative (Way 2)
  - Negative from Negative
  - Sub 0
- **fmul.s**
  - Positive by Positive
  - Positive by Negative (Way 1)
  - Positive by Negative (Way 2)
  - Negative by Negative
  - Mul 0

- **fdiv.s**
  - Positive by Positive
  - Positive by Negative (Way 1)
  - Positive by Negative (Way 2)
  - Negative by Negative
  - Div 0
  - Div 1
  - Div -1
- **fsqrt.s**
  - Evenly Roots
  - Unevenly Roots
  - 0
  - 1
  - Large Case
  - Negative Number
- **fsgnj.s**
  - Both Positive
  - Positive & Negative (Way 1)
  - Positive & Negative (Way 2)
  - Both Negative
- **fsgnjn.s**
  - Both Positive
  - Positive & Negative (Way 1)
  - Positive & Negative (Way 2)
  - Both Negative
- **fsgnjx.s**
  - Both Positive
  - Positive & Negative (Way 1)
  - Positive & Negative (Way 2)
  - Both Negative

- **fmin.s**
  - Both Positive (rs1 larger)
  - Both Positive (rs2 larger)
  - Positive & Negative (rs1 larger)
  - Positive & Negative (rs2 larger)
  - Both Negative (rs1 larger)
  - Both Negative (rs2 larger)
  - Zero w/ Positive
  - Zero w/ Negative
- **fmax.s**
  - Both Positive (rs1 larger)
  - Both Positive (rs2 larger)
  - Positive & Negative (rs1 larger)
  - Positive & Negative (rs2 larger)
  - Both Negative (rs1 larger)
  - Both Negative (rs2 larger)
  - Zero w/ Positive
  - Zero w/ Negative
- **fcvt.s.w**
  - Zero
  - Negative
  - Positive
  - Rounding
- **fcvt.s.wu**
  - Zero
  - Negative
  - Positive
  - Rounding
- **fcvt.w.s**
  - Zero
  - Negative
  - Positive
  - Rounding

- **fcvt.wu.s**
  - Zero
  - Negative
  - Positive
  - Rounding
- **fmv.x.w**
  - Positive (w/o Rounding)
  - Positive (w/ Rounding)
  - Negative (w/o Rounding)
  - Negative (w/ Rounding)
  - Zero
- **fmv.w.x**
  - Positive (w/o Rounding)
  - Negative (w/o Rounding)
  - Zero
- **feq.s**
  - Equal
  - Greater Than
  - Less Than
  - NAN
- **flt.s**
  - Equal
  - Greater Than
  - Less Than
  - NAN
- **fle.s**
  - Equal
  - Greater Than
  - Less Than
  - NAN

- **fclass.s**
  - -∞
  - -Normal Number
  - -Subnormal Number
  - -0
  - +0
  - +Subnormal Number
  - +Normal Number
  - +∞
  - Signaling NAN
  - Quiet NAN

Results: All of our instructions appear to generate the results we'd expect, store them properly, and thus our instructions are properly broken down as well.

4 Conclusion

In conclusion, our project successfully simulates the RISC-V ISA, RV32I, along with its RV32M and RV32F / D extensions. Through our designs, the implementation of instruction fetching, decoding, and executing, register handling, and memory management, the simulator accurately replicated the behavior of RISC-V programs. We also included several debugging features, such as Single-Step Mode, breakpoints, and a memory watching tool, to enhance usability, making it an effective tool for analyzing assembly programs.

Our testing methodology, which includes case analysis for each instruction, ensures the reliability and correctness of our simulator. Each instruction was verified for proper breakdown, accurate arithmetic or logical computation, and correct memory handling, with special attention to edge cases like sign extension, and misaligned memory accesses. The results signal that the simulator consistently produces the expected outputs across all tested scenarios, validating both the core RV32I functionality and the extended instruction sets.

Beyond functional accuracy, the project furthered our understanding of computer architecture, particularly the workings of an ISA, and low-level program execution. It deepened our knowledge of how modern processors decode, execute, and handle instructions, knowledge that will extend beyond RISC-V into broader concepts of the subject. Furthermore, implementing the extra credit extensions challenged us to adapt our design for more complex operations.

Overall, this project demonstrates our successful simulation of the RISC-V ISA. The completed simulator stands as a comprehensive, reliable tool for exploring RISC-V behavior and offers a foundation for future improvements — whether expanding to more advanced RISC-V extensions or working to design a micro-architecture to support our simulator.