Project Summary

Verilog-Based Implementation of a RISC-V Single-Cycle Processor

Introduction

This project involves designing and simulating a single-cycle RISC-V processor using Verilog HDL. The processor supports a core subset of the RISC-V ISA, specifically R-type, I-type, S-type, and B-type instructions. The goal was to deepen understanding of CPU microarchitecture by building key components from scratch.

The processor follows a single-cycle architecture. Each instruction is fetched, decoded, and executed in one clock cycle. All parts, including control logic, datapath, and memory, were built modularly using a structural design approach.

Functionality was validated using custom instruction memory files containing various RISC-V sequences. These test cases covered instruction execution, branching, memory access, and register operations. Simulation and waveform analysis confirmed correct behaviour, including proper memory and register updates.

This project reinforced foundational concepts in computer architecture and digital design while offering hands-on experience with hardware description languages.

Overview of Processor Design and Supported Instruction Set

The implemented processor is a 32-bit single-cycle design based on the RV32I instruction set. The design is structured into **four main modules**:

- PC
- instructionMemory
- instructionDecoder
- RISCV_TOP (top-level integration module)

A testbench module, RISCV_TB, was also developed for simulation.

Supported Instructions:

The processor supports a functional subset of the RV32I instruction set:

- R-type: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT
- I-type: ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI, SRAI
- Load: LW
- Store: SW
- Branch (B-type): BEQ, BNE, BLT, BGE

These instructions enable full arithmetic/logic operations, memory access and branching sufficient for control-heavy logic and algorithms like sorting.

Key Design Characteristics:

- Single-cycle execution: Each instruction completes in one clock cycle.
- No pipelining: No need for hazard detection or forwarding logic.
- PC increment: The program counter increments by 1 (word addressing assumed).
- Immediate support: Immediate values are sign-extended as per instruction type.
- No external modules: General Purpose Registers and Data Memory are managed internally and verified via simulation.

Execution Flow:

1. PC Module: Provides instruction address.
2. Instruction Memory: Fetches a 32-bit instruction.
3. Instruction Decoder: Generates control signals, extracts registers and immediates.
4. RISCV_TOP: Executes instructions using internal logic.
5. Simulation Output: Register and memory changes observed via waveform analysis.

This architecture offers a clear and modular design, supporting essential RISC-V features needed for a wide range of general-purpose and control-oriented tasks.

Instruction Types, Instructions and Their Functions

The processor supports key instruction types from the RV32I base ISA: R-type, I-type, S-type and B-type. Each type has a distinct format and purpose, outlined below:

1. R-Type (Register-Register Instructions)

Purpose: Performs arithmetic and logical operations using two source registers. The result is written to a destination register.

Format: **funct7 | rs2 | rs1 | funct3 | rd | opcode**

Supported Instructions:

- ADD: rd = rs1 + rs2
- SUB: rd = rs1 - rs2
- AND, OR, XOR: Bitwise operations
- SLL, SRL, SRA: Shift operations
- SLTU: Unsigned comparison; rd = 1 if rs1 < rs2, else 0

2. I-Type (Immediate Logic, Arithmetic, Shift, Load)

Purpose: Executes arithmetic/logical operations between a register and an immediate value. Also used for loads and register-based jumps.

Format: **imm[11:0] | rs1 | funct3 | rd | opcode**

Supported Instructions:

- ADDI, ANDI, ORI, XORI: Arithmetic/bitwise ops with immediates
- SLTI: Signed comparison with immediate
- SLLI, SRLI, SRAI: Shift operations by immediate value
  Supported Load Instruction:
- LW: Load a 32-bit word from memory at rs1 + imm into rd
  Supported Jump Instruction:

3. S-Type (Store Instructions)

Purpose: Stores a register’s value into memory at an address computed from a base register and an immediate offset.

Format: **imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode**

Supported Instruction:

- SW: Store 32-bit word from rs2 into memory at rs1 + imm

4. B-Type (Branch Instructions)

Purpose: Performs conditional branching by comparing two registers. If the condition is true, the PC is updated by a signed immediate offset.

Format: **imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode**

Supported Instructions:

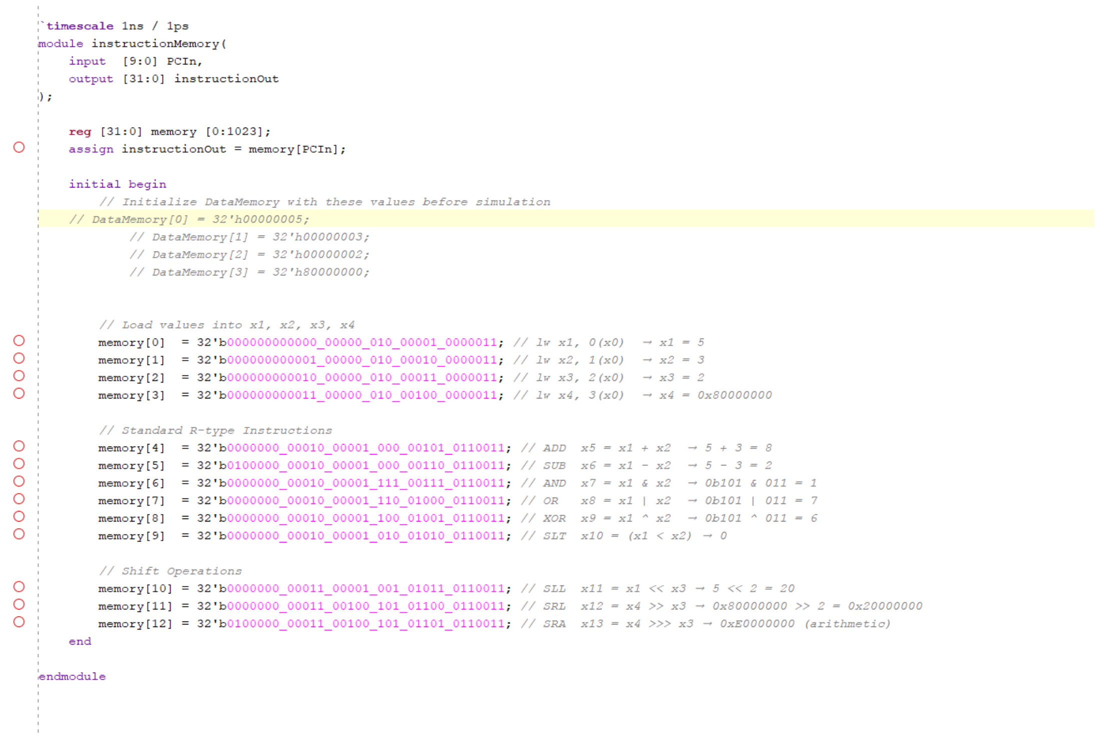
- BEQ: Branch if rs1 == rs2
- BNE: Branch if rs1 != rs2
- BLTU: Branch if rs1 < rs2 (unsigned)
- BGEU: Branch if rs1 >= rs2 (unsigned)

Demonstration of Supported Instructions

To test the correctness and functionality of the processor, I have used three separate instruction memory setups—each designed to evaluate specific instruction types: R-type, I-type, and bubble sort logic using conditional and memory instructions.

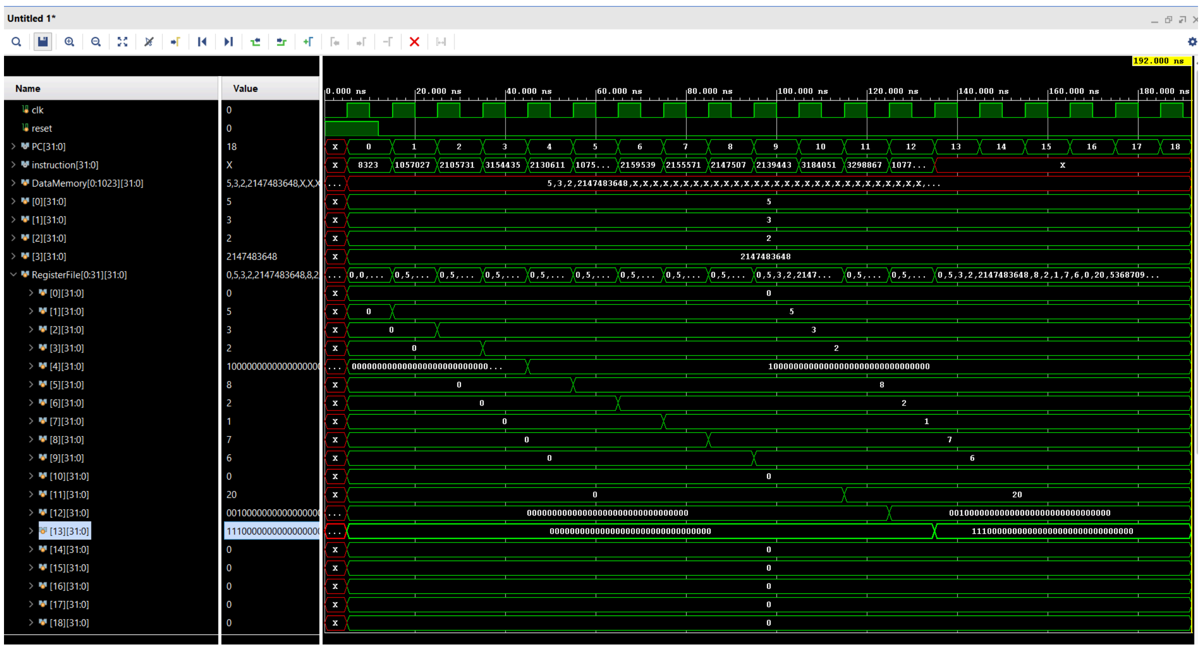
R-Type Instruction Testing:

I created an instruction memory module specifically to test R-type instructions. This memory contains a sequence of LW instructions to first load test data into registers (x1 to x4) from predefined DataMemory locations, followed by a comprehensive set of R-type operations like ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, and SLTU.



```verilog
`timescale 1ns / 1ps
module instructionMemory(
    input  [9:0] PCIn,
    output [31:0] instructionOut
);

    reg [31:0] memory [0:1023];
    assign instructionOut = memory[PCIn];

    initial begin
        // Initialize DataMemory with these values before simulation
    // DataMemory[0] = 32'h00000005;
            // DataMemory[1] = 32'h00000003;
            // DataMemory[2] = 32'h00000002;
            // DataMemory[3] = 32'h80000000;


        // Load values into x1, x2, x3, x4
        memory[0]  = 32'b000000000000_00000_010_00001_0000011; // lw x1, 0(x0)  → x1 = 5
        memory[1]  = 32'b000000000001_00000_010_00010_0000011; // lw x2, 1(x0)  → x2 = 3
        memory[2]  = 32'b000000000010_00000_010_00011_0000011; // lw x3, 2(x0)  → x3 = 2
        memory[3]  = 32'b000000000011_00000_010_00100_0000011; // lw x4, 3(x0)  → x4 = 0x80000000

        // Standard R-type Instructions
        memory[4]  = 32'b0000000_00010_00001_000_00101_0110011; // ADD  x5 = x1 + x2  → 5 + 3 = 8
        memory[5]  = 32'b0100000_00010_00001_000_00110_0110011; // SUB  x6 = x1 - x2  → 5 - 3 = 2
        memory[6]  = 32'b0000000_00010_00001_111_00111_0110011; // AND  x7 = x1 & x2  → 0b101 & 011 = 1
        memory[7]  = 32'b0000000_00010_00001_110_01000_0110011; // OR   x8 = x1 | x2  → 0b101 | 011 = 7
        memory[8]  = 32'b0000000_00010_00001_100_01001_0110011; // XOR  x9 = x1 ^ x2  → 0b101 ^ 011 = 6
        memory[9]  = 32'b0000000_00010_00001_010_01010_0110011; // SLT  x10 = (x1 < x2) → 0

        // Shift Operations
        memory[10] = 32'b0000000_00011_00001_001_01011_0110011; // SLL  x11 = x1 << x3 → 5 << 2 = 20
        memory[11] = 32'b0000000_00011_00100_101_01100_0110011; // SRL  x12 = x4 >> x3 → 0x80000000 >> 2 = 0x20000000
        memory[12] = 32'b0100000_00011_00100_101_01101_0110011; // SRA  x13 = x4 >>> x3 → 0xE0000000 (arithmetic)
    end

endmodule
```

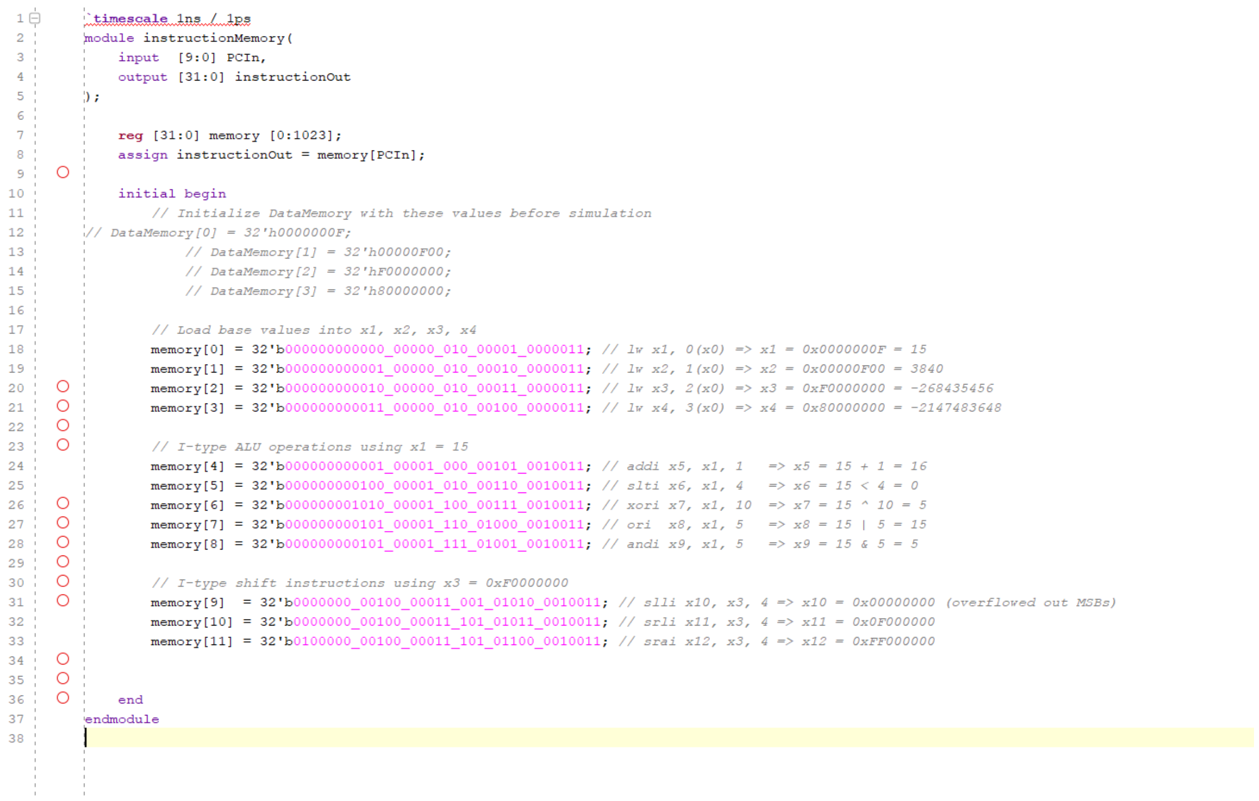
After loading the instruction memory and running the simulation, the values in the general-purpose registers (x5 to x13) are updated based on the ALU operations. This confirms that all R-type instructions perform as expected.



```
Untitled 1*

Name                      Value
clk                       0
reset                     0
PC[31:0]                  18
instruction[31:0]         X
DataMemory[0:1023][31:0]  5,3,2,2147483648,X,X,X
[0][31:0]                 5
[1][31:0]                 3
[2][31:0]                 2
[3][31:0]                 2147483648
RegisterFile[0:31][31:0]  0,5,3,2,2147483648,8,2,
[0][31:0]                 0
[1][31:0]                 5
[2][31:0]                 3
[3][31:0]                 2
[4][31:0]                 1000000000000000000
[5][31:0]                 8
[6][31:0]                 2
[7][31:0]                 1
[8][31:0]                 7
[9][31:0]                 6
[10][31:0]                0
[11][31:0]                20
[12][31:0]                0010000000000000000
[13][31:0]                1110000000000000000
[14][31:0]                0
[15][31:0]                0
[16][31:0]                0
[17][31:0]                0
[18][31:0]                0
```

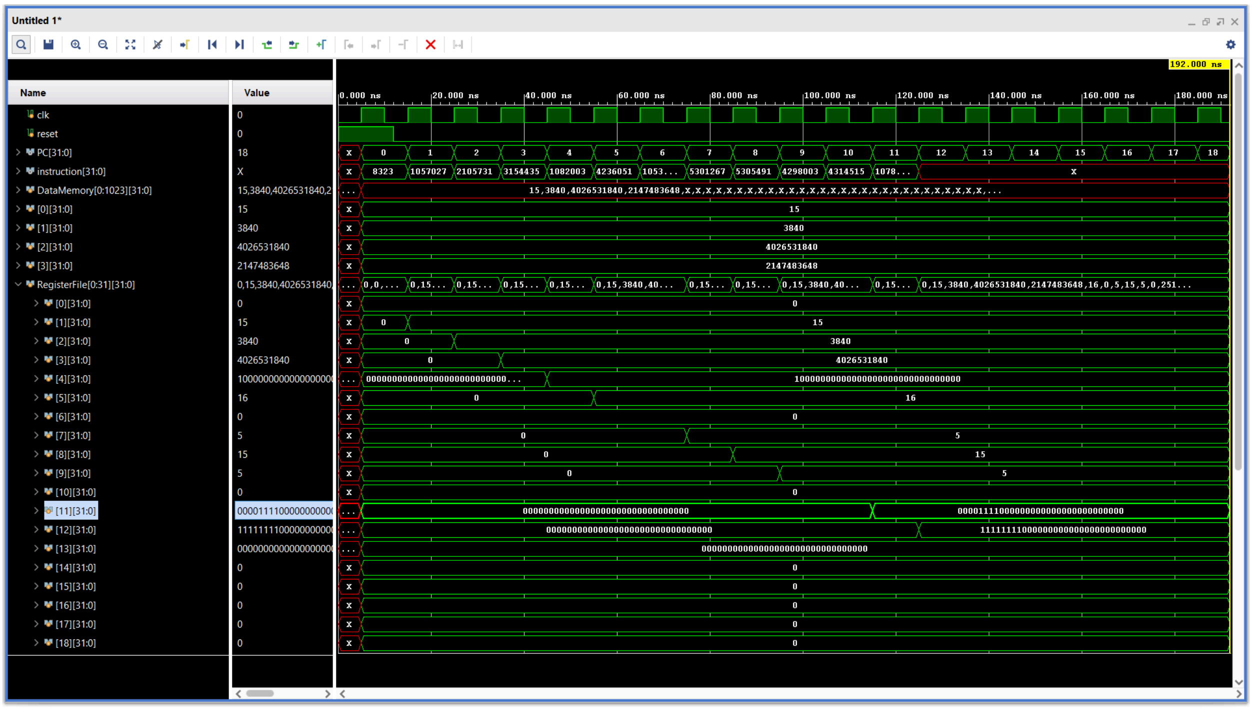
I-Type Instruction Testing:

Similar to the R-type, I prepared a separate instruction memory for testing I-type instructions. The test begins by loading a range of test values into registers x1 to x4 using LW. Then, immediate-based instructions like ADDI, SLTI, ANDI, ORI, XORI, SLLI, SRLI, and SRAI are executed. These cover arithmetic, logic, and shift operations involving immediate values.



```verilog
`timescale 1ns / 1ps
module instructionMemory(
    input  [9:0] PCIn,
    output [31:0] instructionOut
);

    reg [31:0] memory [0:1023];
    assign instructionOut = memory[PCIn];

    initial begin
        // Initialize DataMemory with these values before simulation
// DataMemory[0] = 32'h0000000F;
            // DataMemory[1] = 32'h00000F00;
            // DataMemory[2] = 32'hF0000000;
            // DataMemory[3] = 32'h80000000;

        // Load base values into x1, x2, x3, x4
        memory[0] = 32'b000000000000_00000_010_00001_0000011; // lw x1, 0(x0) => x1 = 0x0000000F = 15
        memory[1] = 32'b000000000001_00000_010_00010_0000011; // lw x2, 1(x0) => x2 = 0x00000F00 = 3840
        memory[2] = 32'b000000000010_00000_010_00011_0000011; // lw x3, 2(x0) => x3 = 0xF0000000 = -268435456
        memory[3] = 32'b000000000011_00000_010_00100_0000011; // lw x4, 3(x0) => x4 = 0x80000000 = -2147483648

        // I-type ALU operations using x1 = 15
        memory[4] = 32'b000000000001_00001_000_00101_0010011; // addi x5, x1, 1   => x5 = 15 + 1 = 16
        memory[5] = 32'b000000000100_00001_010_00110_0010011; // slti x6, x1, 4   => x6 = 15 < 4 = 0
        memory[6] = 32'b000000001010_00001_100_00111_0010011; // xori x7, x1, 10  => x7 = 15 ^ 10 = 5
        memory[7] = 32'b000000000101_00001_110_01000_0010011; // ori  x8, x1, 5   => x8 = 15 | 5 = 15
        memory[8] = 32'b000000000101_00001_111_01001_0010011; // andi x9, x1, 5   => x9 = 15 & 5 = 5

        // I-type shift instructions using x3 = 0xF0000000
        memory[9]  = 32'b0000000_00100_00011_001_01010_0010011; // slli x10, x3, 4 => x10 = 0x00000000 (overflowed out MSBs)
        memory[10] = 32'b0000000_00100_00011_101_01011_0010011; // srli x11, x3, 4 => x11 = 0x0F000000
        memory[11] = 32'b0100000_00100_00011_101_01100_0010011; // srai x12, x3, 4 => x12 = 0xFF000000


    end
endmodule
```

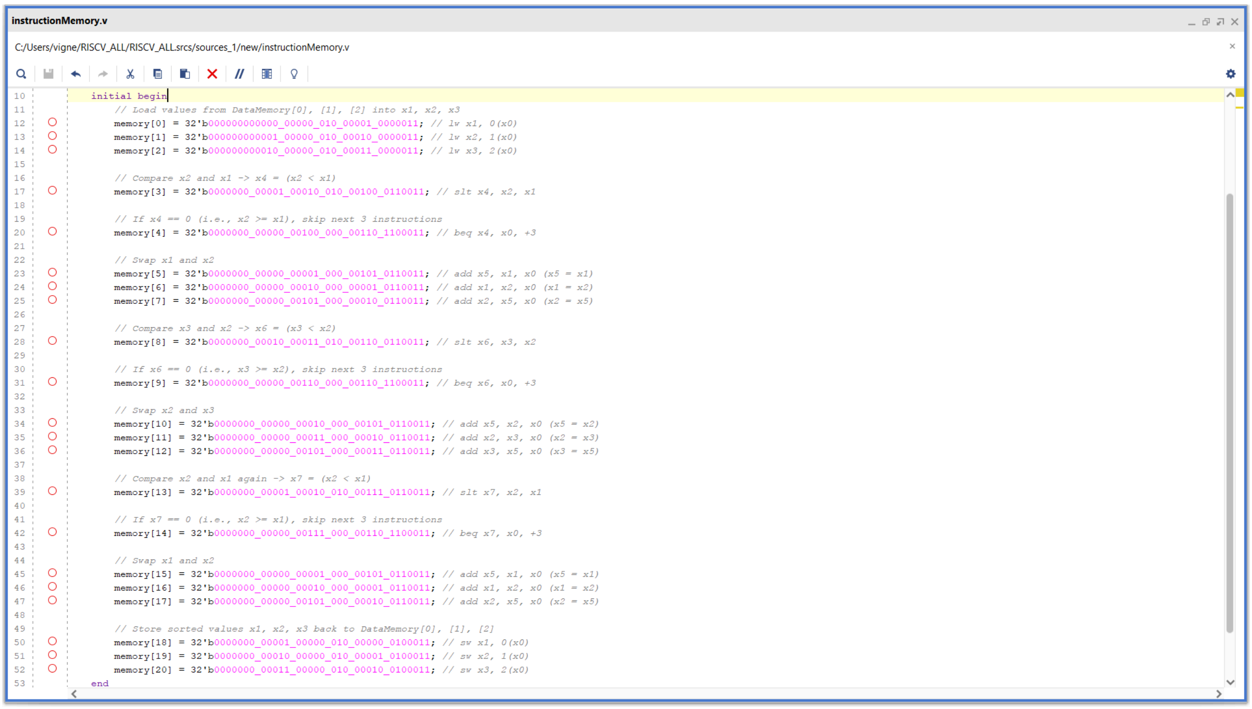
Simulation shows correct updates to the destination registers (x5 to x12), verifying the handling of signed/unsigned values and immediate shift limits.


Untitled 1*

Name	Value
clk	0
reset	0
PC[31:0]	18
instruction[31:0]	X
DataMemory[0:1023][31:0]	15,3840,4026531840,2
[0][31:0]	15
[1][31:0]	3840
[2][31:0]	4026531840
[3][31:0]	2147483648
RegisterFile[0:31][31:0]	0,15,3840,4026531840,
[0][31:0]	0
[1][31:0]	15
[2][31:0]	3840
[3][31:0]	4026531840
[4][31:0]	100000000000000000
[5][31:0]	16
[6][31:0]	0
[7][31:0]	5
[8][31:0]	15
[9][31:0]	5
[10][31:0]	0
[11][31:0]	000011110000000000
[12][31:0]	111111110000000000
[13][31:0]	000000000000000000
[14][31:0]	0
[15][31:0]	0
[16][31:0]	0
[17][31:0]	0
[18][31:0]	0

Sorting Algorithm Using RISC-V Assembly:

The sorting algorithm implemented is a hardcoded version of the bubble sort technique using RISC-V assembly instructions. It operates on three values loaded from memory into registers and performs pairwise comparisons using the slt (set less than) instruction. If a value is out of order, conditional branching with beq triggers a manual swap using add instructions to copy and exchange register values. This process is repeated to ensure all elements are sorted, mimicking the two-pass nature of bubble sort. Finally, the sorted values are stored back into memory using sw instructions. This demonstrates control flow, conditional logic, and register manipulation in the processor.

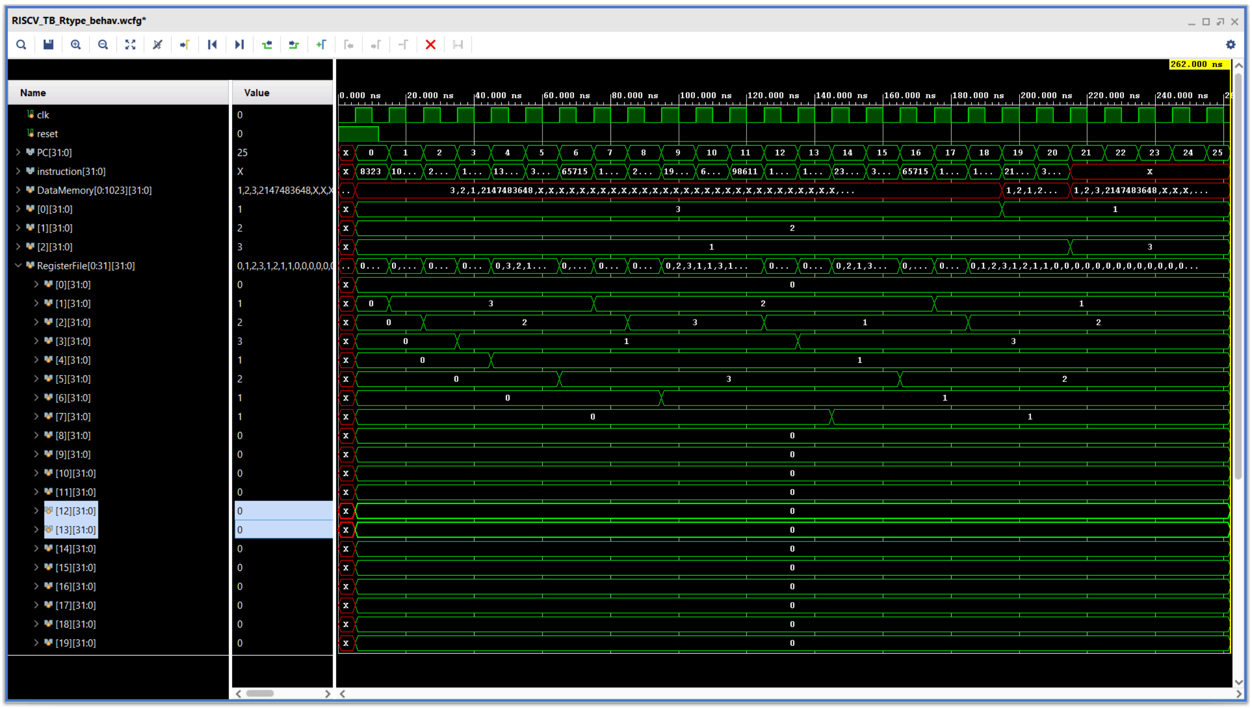


instructionMemory.v

C:/Users/vigne/RISCV_ALL/RISCV_ALL.srcs/sources_1/new/instructionMemory.v

```verilog
    initial begin
        // Load values from DataMemory[0], [1], [2] into x1, x2, x3
        memory[0] = 32'b000000000000_00000_010_00001_0000011; // lw x1, 0(x0)
        memory[1] = 32'b000000000001_00000_010_00010_0000011; // lw x2, 1(x0)
        memory[2] = 32'b000000000010_00000_010_00011_0000011; // lw x3, 2(x0)

        // Compare x2 and x1 -> x4 = (x2 < x1)
        memory[3] = 32'b0000000_00001_00010_010_00100_0110011; // slt x4, x2, x1

        // If x4 == 0 (i.e., x2 >= x1), skip next 3 instructions
        memory[4] = 32'b0000000_00000_00100_000_00110_1100011; // beq x4, x0, +3

        // Swap x1 and x2
        memory[5] = 32'b0000000_00000_00001_000_00101_0110011; // add x5, x1, x0 (x5 = x1)
        memory[6] = 32'b0000000_00000_00010_000_00001_0110011; // add x1, x2, x0 (x1 = x2)
        memory[7] = 32'b0000000_00000_00101_000_00010_0110011; // add x2, x5, x0 (x2 = x5)

        // Compare x3 and x2 -> x6 = (x3 < x2)
        memory[8] = 32'b0000000_00010_00011_010_00110_0110011; // slt x6, x3, x2

        // If x6 == 0 (i.e., x3 >= x2), skip next 3 instructions
        memory[9] = 32'b0000000_00000_00110_000_00110_1100011; // beq x6, x0, +3

        // Swap x2 and x3
        memory[10] = 32'b0000000_00000_00010_000_00101_0110011; // add x5, x2, x0 (x5 = x2)
        memory[11] = 32'b0000000_00000_00011_000_00010_0110011; // add x2, x3, x0 (x2 = x3)
        memory[12] = 32'b0000000_00000_00101_000_00011_0110011; // add x3, x5, x0 (x3 = x5)

        // Compare x2 and x1 again -> x7 = (x2 < x1)
        memory[13] = 32'b0000000_00001_00010_010_00111_0110011; // slt x7, x2, x1

        // If x7 == 0 (i.e., x2 >= x1), skip next 3 instructions
        memory[14] = 32'b0000000_00000_00111_000_00110_1100011; // beq x7, x0, +3

        // Swap x1 and x2
        memory[15] = 32'b0000000_00000_00001_000_00101_0110011; // add x5, x1, x0 (x5 = x1)
        memory[16] = 32'b0000000_00000_00010_000_00001_0110011; // add x1, x2, x0 (x1 = x2)
        memory[17] = 32'b0000000_00000_00101_000_00010_0110011; // add x2, x5, x0 (x2 = x5)

        // Store sorted values x1, x2, x3 back to DataMemory[0], [1], [2]
        memory[18] = 32'b0000000_00001_00000_010_00000_0100011; // sw x1, 0(x0)
        memory[19] = 32'b0000000_00010_00000_010_00001_0100011; // sw x2, 1(x0)
        memory[20] = 32'b0000000_00011_00000_010_00010_0100011; // sw x3, 2(x0)
    end
```

The simulation plot confirms correct comparisons, conditional execution of swaps, and correct ordering of the values in DataMemory after execution.



RISCV_TB_Rtype_behav.wcfg*

Name	Value
clk	0
reset	0
PC[31:0]	25
instruction[31:0]	X
DataMemory[0:1023][31:0]	1,2,3,2147483648,X,X,X
[0][31:0]	1
[1][31:0]	2
[2][31:0]	3
RegisterFile[0:31][31:0]	0,1,2,3,1,2,1,1,0,0,0,0,0,0
[0][31:0]	0
[1][31:0]	1
[2][31:0]	2
[3][31:0]	3
[4][31:0]	1
[5][31:0]	2
[6][31:0]	1
[7][31:0]	1
[8][31:0]	0
[9][31:0]	0
[10][31:0]	0
[11][31:0]	0
[12][31:0]	0
[13][31:0]	0
[14][31:0]	0
[15][31:0]	0
[16][31:0]	0
[17][31:0]	0
[18][31:0]	0
[19][31:0]	0

This methodical instruction testing verifies that each instruction type behaves as intended in the single-cycle RISC-V processor.

Improvements and Future Work

Proposed Improvements:

- Modular ALU & Control Unit: Improve readability and testability by separating into standalone modules.
- Dedicated Memory & Register File: Promote modular design by isolating memory and register components.
- Byte-Addressable Memory: Align with RISC-V specs for more flexible data handling.
- Improved Immediate Decoder: Ensure accurate decoding across all instruction types and edge cases.
- Enhanced PC & Branch Logic: Refine control flow and jump handling mechanisms.

Future Goals

- Pipelined Architecture: Add pipelining with hazard detection for performance gains.
- Hazard Unit: Reduce stalls and improve execution flow.
- Instruction Set Expansion: Support full RV32I and RV32M for arithmetic extensions.

Conclusion

This project successfully demonstrates the design and simulation of a single-cycle RISC-V processor using Verilog HDL, capable of executing a functional subset of the RV32I instruction set. Through modular implementation and waveform-based testing, core processor concepts such as instruction decoding, ALU operations, memory access, and control flow were effectively realised. The processor serves as a strong foundation for understanding microarchitectural design and offers ample scope for future enhancements, including pipelining, full ISA support, and system-level integration.