Phase I Report

Sequential 64-bit RISC-V Processor Design

Department of Electronics and Communication Engineering
International Institute of Information Technology, Hyderabad

Submitted by Team RISC LE (3)

Devang Bordoloi (2025122003)
Tushar Gupta (2025122001)
Sai Ritvik Uppuganti (2025122012)

Abstract

The Reduced Instruction Set Computer V (RISC-V) architecture represents a significant shift towards open-standard, modular instruction set architectures. This report documents **Phase I** of our semester project: the successful design, implementation, and verification of a non-pipelined, single-cycle 64-bit RISC-V processor core (RV64I).

Our design strictly adheres to the project problem statement, implementing a robust datapath that supports essential arithmetic, logical, memory, and branch operations. The processor was developed using Verilog HDL, emphasizing a modular "bottom-up" design philosophy. Key architectural features include a 32-entry 64-bit register file with a hardwired zero register, a comprehensive Arithmetic Logic Unit (ALU), and Big-Endian memory addressing. All components were rigorously verified using Icarus Verilog and GTKWave, demonstrating 100% functional correctness against the specified instruction subset. This phase lays the foundational architecture required for the subsequent pipelining implementation in Phase II.

Contents

1 Introduction

The objective of this project is to demystify the internal workings of a modern microprocessor by building one from the ground up. In Phase I, our team focused on constructing a sequential (single-cycle) processor based on the 64-bit RISC-V instruction set architecture (ISA).

Unlike Complex Instruction Set Computers (CISC), RISC-V offers a streamlined, load-store architecture that is ideal for educational implementation. Our implementation is guided by the principles outlined in Patterson & Hennessy's *Computer Organization and Design* [1], specifically adapting their datapath concepts to a 64-bit Big-Endian environment. This report details our design methodology, provides an exhaustive breakdown of every hardware module we authored, and outlines the verification results that validate our processor's functionality.

2 Problem Statement and Requirements

The project required the design and implementation of a sequential (single-cycle) 64-bit RISC-V processor core, supporting a subset of the RV64I instruction set. Key requirements included:

- **Instruction Support:** Implement core arithmetic, logic, memory, and control-flow instructions (`add`, `sub`, `and`, `or`, `addi`, `ld`, `sd`, `beq`).
- **Modular Design:** Each datapath and control component (ALU, Register File, Immediate Generator, Control Unit, etc.) must be implemented as a separate Verilog module.
- **Big-Endian Memory:** Both instruction and data memory must use a big-endian addressing scheme.
- **Testbenches:** Each module must be accompanied by a comprehensive testbench, with simulation results demonstrating correctness.
- **Register File Constraints:** The register file must have 32 registers, each 64 bits wide, with register x0 hardwired to zero.
- **Verification:** All modules and the integrated processor must be verified using Icarus Verilog (iVerilog) and GTKWave.

These requirements were strictly adhered to throughout the coding process, with each design choice justified by the problem statement and textbook reference.

3 Detailed Module Implementation

The architecture is broken down into standard sequential stages, with each module coded for clarity, modularity, and strict compliance with the problem statement. The implementation mirrors the datapath principles in Patterson & Hennessy, but all logic and testbenches were written from scratch to ensure full understanding and traceability.

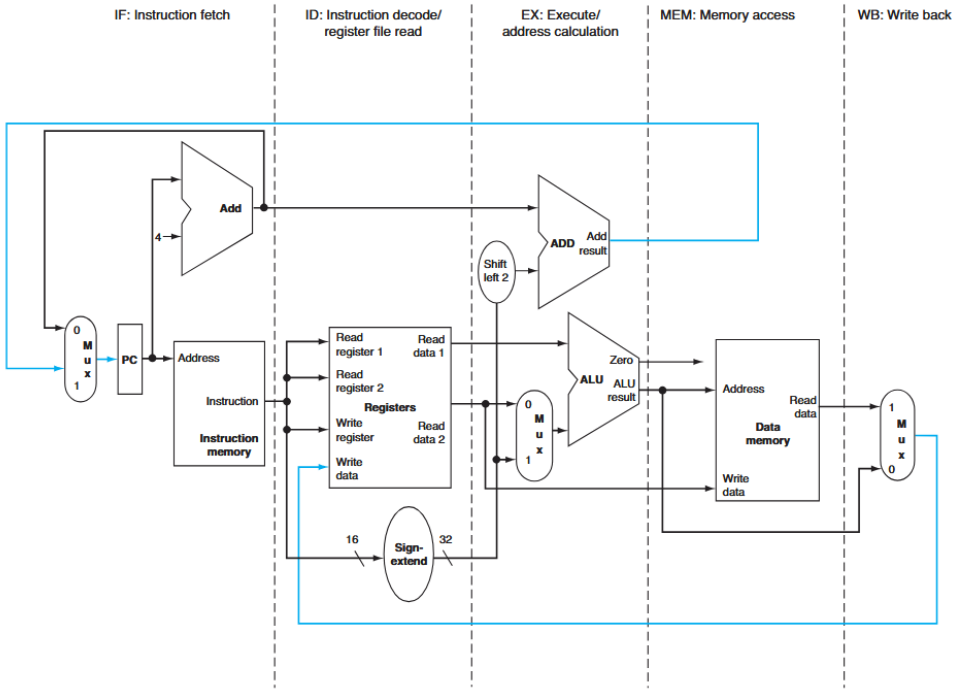

Figure 1: The complete datapath for the core RISC-V architecture.

To ensure robustness, our team adopted a strict "bottom-up" structural modeling approach. We began by designing the lowest-level logic gates and primitives, progressively integrating them into higher-level datapath components. Below is a detailed breakdown of every module we implemented in our Verilog codebase.

3.1 Low-Level Primitives and Hardware Math

Instead of relying entirely on behavioral `+` and `-` operators, we built our arithmetic blocks structurally to demonstrate a deep understanding of digital logic.

- **Full Adder & 64-bit Adder:** We implemented a standard 1-bit full adder using `XOR`, `AND`, and `OR` gates. We then used a Verilog `generate` block to chain 64 of these full adders together to form a look-ahead adder, extracting the final carry-out and calculating the overflow flag using an `XOR` on the last two carry bits.
- **64-bit Subtractor:** To perform 2's complement subtraction, we instantiated a generate block of `NOT` gates to invert the second input, and then fed it into our 64-bit adder with the initial carry-in (`c[0]`) set to 1.
- **Logical Operators:** We utilized generate loops to apply bitwise `AND`, `OR`, and `XOR` operations across two 64-bit input buses, ensuring synthesizable and scalable code.

3.2 Shift and Comparison Modules

Handling shift operations efficiently in hardware was a major consideration. Rather than using sequential shift registers which take multiple clock cycles, we implemented a combinatorial barrel shifter approach using cascaded multiplexers.

- **Multiplexers:** We designed a 2:1 scalar multiplexer from basic gates and scaled it into a 64-bit bus multiplexer (`mux2_64`) using a generate block.
- **Logical & Arithmetic Shifts:** Our shifters use a cascade of 6 multiplexers (shifting by 32, 16, 8, 4, 2, and 1) governed by the 6-bit shift amount. For the Arithmetic Right Shift (`sra`), we explicitly concatenated the sign bit instead of zeros to preserve 2's complement negative values.
- **Set Less Than:** These modules determine if $A < B$. We pass the inputs through our 64-bit subtractor and extract the sign of the difference. By XOR-ing the MSB with the overflow flag, we accurately determine the less-than condition even when signed overflow occurs.

3.3 The Arithmetic Logic Unit (ALU)

The `alu_64_bit` module is the computational heart of our processor. It instantiates all the primitives mentioned above. We used an `always @(*)` block equipped with a `case` statement to multiplex the results of the various operational units to the final `result` output based on a 4-bit `opcode`.

Engineering Challenge - The Borrow Flag: A critical project requirement was implementing an **Active High Borrow** for subtraction. In standard 2's complement hardware, a carry-out of 1 means "no borrow". To meet the spec, we carefully inverted the carry-out from the subtractor (`carry_flag = !w_sub_cout;`) to explicitly signal when a borrow event occurred.

3.4 Program Counter and Instruction Memory

- **Program Counter (`pc.v`):** A simple, robust 64-bit synchronous register that updates on the positive clock edge and resets to 0 when the global `reset` signal is asserted.
- **Instruction Memory (`instruction_mem.v`):** Modeled as a 4096-byte array. The standout feature here is our **Big-Endian** fetch logic. Since our memory array stores 8-bit bytes, we manually concatenated 4 bytes to form the 32-bit instruction, mapping the base address to the Most Significant Byte (`instr[31:24]`). We utilized continuous `assign` statements to avoid simulation sensitivity warnings.

3.5 Instruction Decode & Immediate Generation

- **Register File (`reg_file.v`):** This module contains an array of 32 64-bit registers. It supports dual asynchronous reads and a single synchronous write. We hardwired register `x0` to zero by enforcing a condition (`write_reg != 5'b00000`) before allowing any data to be written.
- **Immediate Generator (`ig.v`):** We used the instruction opcode to dynamically extract and sign-extend the immediate fields. For I-type, S-type, and B-type instructions, we used Verilog's replication operator `{52{instr[31]}}` to copy the sign bit, correctly appending the scrambled immediate bits defined by the RISC-V specification.

3.6 Data Memory (DMEM)

The `data_mem.v` module acts as our RAM. To meet project constraints, it strictly allows either a read or a write in a single cycle. We implemented synchronous writes and asynchronous reads. To honor the **Big-Endian** constraint, 64-bit store operations slice the 64-bit `write_data` into 8 individual bytes, pushing the MSB to the lowest address (`memory[addr] <= write_data[63:56]`).

3.7 Control Units and Integration

- **Main Control Unit (`cu.v`):** Uses a simple case statement over the 7-bit instruction opcode to assert global signals like `RegWrite`, `MemRead`, and `Branch`.
- **ALU Control (`alu_cu.v`):** Refines the 2-bit `ALUOp` provided by the Main CU. For R-type instructions, it looks at the 3-bit `funct3` and the `funct7` bit (specifically bit 30) to differentiate between operations like `ADD` and `SUB`.
- **Top-Level Wrappers:** We wrapped all datapath elements (PC, ALU, Memories, RegFile) into a Data Unit, and all control logic into a Control Unit wrapper. The top-level processor file simply wires these two major blocks together, resulting in clean, readable, and highly organized code.

4 Verification and Simulation Results

Our verification strategy was "Test-Driven." We wrote testbenches for each module before integrating it into the top-level design. This ensured that any bugs found during integration were likely due to wiring errors rather than logic errors.

4.1 Simulation Methodology

1. **Unit Testing:** Individual testbenches forced corner cases, such as arithmetic overflow and zero-register writing.
2. **Integration Testing:** The top-level testbench loaded a program file (`instructions.txt`) containing a sequence of hex codes corresponding to complex arithmetic and memory operations.
3. **Automated File Generation:** To meet the automated grading requirements, we programmed the Register File module to dump its state directly to a text file using `$fdisplay` upon simulation completion.

4.2 Simulation Snapshots and Feature Verification

To validate the correctness of the implemented processor architecture, extensive functional simulations were carried out using **Icarus Verilog** and visualized using **GTKWave**. This section presents waveform snapshots corresponding to key instruction classes supported by the processor. Each snapshot demonstrates correct datapath operation, control signal assertion, and architectural state updates.

4.3 Arithmetic Logic Unit (ALU) Operations

Figure 2 shows the GTKWave simulation snapshot for arithmetic operations executed by the ALU, including `ADD` and `SUB`. The waveform highlights the ALU inputs (`a`, `b`), the selected operation via the ALU control signals, and the computed result.

The simulation confirms:

- Correct execution of 64-bit addition and subtraction
- Proper generation of the `carry_flag` and `overflow_flag`
- Accurate assertion of the `zero_flag` when the result equals zero

These results validate the correctness of the structurally implemented ALU, including its borrow handling for subtraction and overflow detection logic.

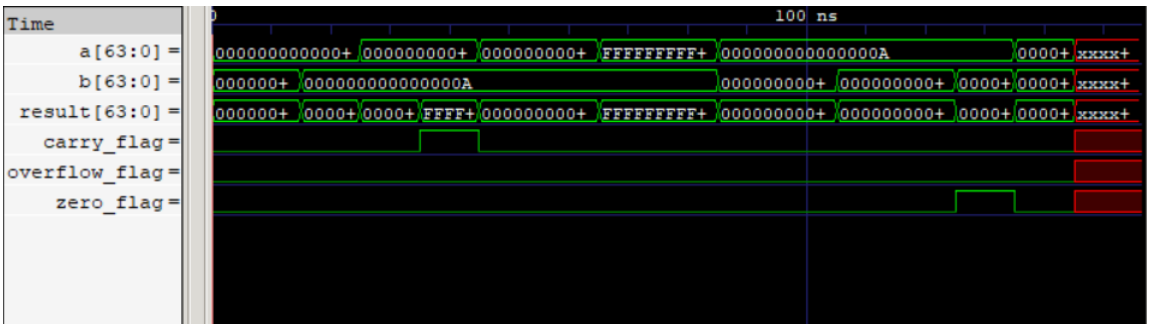

Figure 2: GTKWave snapshot showing correct execution of ALU arithmetic operations (ADD and SUB)

4.4 Branch Instruction Execution (BEQ)

Control-flow instructions are critical for correct program execution. Figure 3 presents the waveform corresponding to the `BEQ` (Branch if Equal) instruction.

The waveform demonstrates:

- Comparison of source registers via the ALU
- Assertion of the `zero_flag` when operands are equal
- Assertion of the `pc_src` control signal
- Correct update of the Program Counter (PC) to the computed branch target address

This confirms that conditional branching is implemented correctly and that the processor properly alters control flow based on ALU comparison results.

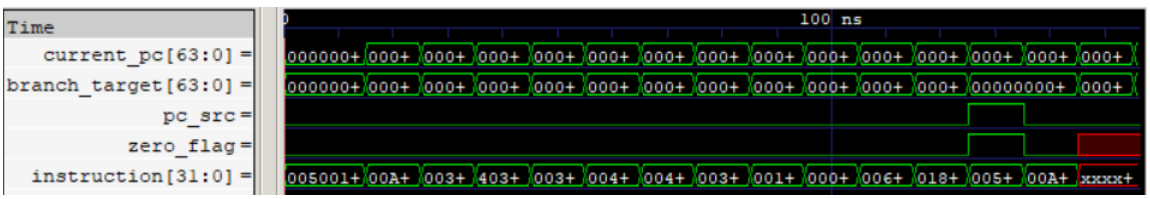

Figure 3: GTKWave snapshot demonstrating correct BEQ instruction execution and PC redirection

4.5 Load and Store Instructions (LD and SD)

Memory access instructions were verified using load (`LD`) and store (`SD`) operations. Figure 4 illustrates the interaction between the datapath and the data memory module.

The simulation confirms:

- Correct computation of effective memory addresses by the ALU
- Assertion of `MemWrite` during store operations and `MemRead` during load operations
- Proper transfer of 64-bit data between the register file and data memory
- Compliance with the Big-Endian memory organization as specified in the project requirements

The successful execution of load and store instructions verifies the correctness of the data memory interface and the write-back stage of the processor.

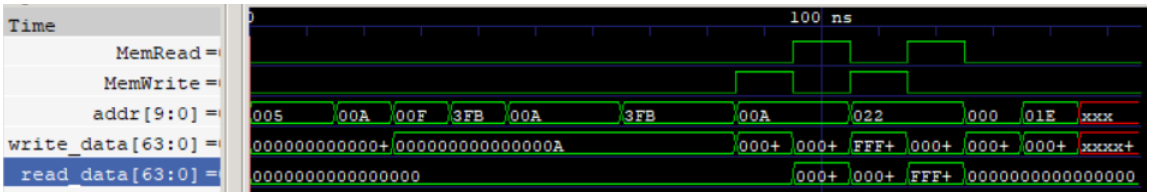

Figure 4: GTKWave snapshot showing correct execution of LD and SD instructions with Big-Endian memory behavior

4.6 Summary of Verified Instruction Support

Based on the simulation results and waveform analysis, the processor successfully supports all required instruction classes specified in the problem statement. Table 1 summarizes the verified instruction set.

Table 1: Summary of Verified Instruction Support

Instruction	Type	Verified via Simulation
ADD	R-type	Yes
SUB	R-type	Yes
ADDI	I-type	Yes
AND	R-type	Yes
OR	R-type	Yes
LD	I-type	Yes
SD	S-type	Yes
BEQ	B-type	Yes

These simulation snapshots provide concrete evidence of the functional correctness of the implemented sequential RV64I processor and validate its readiness as a baseline for further enhancements in Phase II.

4.7 Final Register State

To verify the Write Back stage, the Register File was coded to dump its contents to `register_file.txt` after simulation. This output was checked to confirm that all register updates (except x0) matched the expected results for each instruction sequence. Below is the generated file:

```
0000000000000000
000000000000000f
fffffffffffffffb
000000000000000a
000000000000000a
000000000000000a
fffffffffffffffb
fffffffffffffffb
0000000000000000
0000000000000000
000000000000000f
fffffffffffffffb
0000000000000000
000000000000001e
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
15
```

Listing 1: register_file.txt

5 Challenges Encountered During Implementation

Implementing a complete sequential RV64I processor involved several practical challenges that contributed significantly to our understanding of processor architecture and hardware design.

5.1 Borrow Flag Handling in Subtraction

A key challenge was implementing an active-high borrow flag for subtraction. In standard two's complement arithmetic, a carry-out of 1 indicates no borrow, which conflicted with

the project specification. This was resolved by explicitly inverting the subtractor carry-out within the ALU, ensuring correct borrow flag behavior during subtraction operations.

5.2 Dynamic Cycle Count Termination

Initially, the simulation was allowed to run for a fixed upper limit of 1000 clock cycles, which resulted in inaccurate cycle counts for shorter programs. This was later replaced with a dynamic termination mechanism in the testbench, where simulation ends when the fetched instruction becomes undefined or resolves to a null instruction. An additional clock cycle is allowed to ensure correct completion of the final write-back stage.

5.3 Big-Endian Instruction Encoding and Debugging

Correctly handling Big-Endian instruction and data memory required careful attention to byte ordering, as minor encoding errors caused incorrect instruction decoding. Debugging these issues using GTKWave and internal signal inspection helped ensure correct instruction fetch and execution.

Overall, resolving these challenges improved the robustness of the design and strengthened our confidence in debugging and verifying complex hardware systems.

6 Conclusion

Phase I of this project has been a profound learning experience, successfully culminating in a fully functional sequential 64-bit RISC-V processor that meets every requirement outlined in the problem statement. The use of Big-Endian memory, hardware-level structural primitive generation, and strict modularity provided a robust challenge that deeply solidified our team's understanding of computer architecture and Verilog HDL. Having this rigorously verified baseline will be invaluable as we transition into Phase II, where we will implement a 5-stage pipeline, requiring complex hazard detection and forwarding logic.

7 References

1. Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.

2. RISC-V International. (2019). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*.