Introduction to Processor Architecture

Phase 1 Report

RISC-V Sequential Processor

Name: Shriya Kansal
Roll No: 2024102075

Name: Prathish Ghosh
Roll No: 2024102069

Name: Kavya Veer
Roll No: 2024112002

Contents

1 Introduction

Processor architecture focuses on the design and implementation of processors capable of executing instructions efficiently and reliably. In this project, we implement a **sequential (single-cycle) RISC-V processor** in Verilog that supports a subset of the RV64I base integer instruction set. Each instruction completes in one clock cycle, and no pipelining or hazard handling mechanisms are incorporated in this phase.

RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture designed to provide simplicity, modularity, and extensibility. Its clean instruction encoding, fixed-length instruction format, and load-store architecture make it particularly suitable for academic learning and hardware implementation.

This implementation follows a 64-bit datapath (RV64I), while maintaining a fixed 32-bit instruction format. The processor executes instructions sequentially in the order they appear in memory.

1.1 RISC-V Instruction Set Architecture

RISC-V follows a **load-store architecture**, meaning that only load and store instructions access memory. All arithmetic and logical operations are performed exclusively on register operands. Memory access occurs only through dedicated load (`ld`) and store (`sd`) instructions.

Each instruction in RV64I is 32 bits wide. The instruction format determines how opcode, register operands, function codes, and immediate values are arranged within the 32-bit encoding. Based on the instruction type, different fields are extracted and interpreted by the control unit and datapath components.

In this phase, the processor supports the following instructions:

- **Arithmetic/Logical:** `add, sub, and, or`
- **Immediate Arithmetic:** `addi`
- **Memory Access:** `ld, sd`
- **Control Transfer:** `beq`

1.1.1 Instruction Formats

The supported instruction formats in this design are:

R-Type (Register Type) Used for register-to-register arithmetic and logical operations such as `add`, `sub`, `and`, and `or`. The format contains:

- Opcode
- Destination register (`rd`)
- Source registers (`rs1`, `rs2`)
- Function fields (`funct3`, `funct7`)

All operands are read from the register file, and the result is written back to a destination register.

I-Type (Immediate Type) Used for instructions involving immediate values such as `addi` and `ld`. The format includes:

- Opcode
- Destination register (`rd`)
- Source register (`rs1`)
- 12-bit immediate value (sign-extended to 64 bits)

The immediate value is generated by the Immediate Generator and used as an ALU operand.

S-Type (Store Type) Used for store instructions such as `sd`. The immediate field is split across two parts of the instruction and combined during decoding. The effective memory address is computed by adding the sign-extended immediate to the base register.

B-Type (Branch Type) Used for conditional branch instructions such as `beq`. The immediate field is extracted, sign-extended, and left-shifted appropriately to compute the branch target address. The branch is taken when the ALU zero flag indicates equality between the two source registers.

1.1.2 Instruction Categories

The implemented instructions are classified into the following categories:

- **Arithmetic and Logical Instructions:** These include `add`, `sub`, `and`, and `or`. They operate on 64-bit register operands and produce a 64-bit result through the Arithmetic Logic Unit (ALU).
- **Load and Store Instructions:** The `ld` instruction reads 64-bit data from memory into a register, while `sd` writes 64-bit data from a register into memory. The effective address is computed using base-register plus offset addressing.
- **Control Transfer Instructions:** The `beq` instruction alters program flow conditionally. The branch target address is calculated using the Program Counter (PC) and the sign-extended branch offset. If the branch condition is satisfied, the PC is updated with the branch target; otherwise, execution proceeds sequentially.

This structured instruction organization enables a clean and modular hardware implementation, forming the foundation for the sequential datapath described in the following sections.

2 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic and logical operations on 64-bit operands. It receives two 64-bit inputs along with a 4-bit control signal (`ALUControl`) generated by the ALU Control unit. Based on this control signal, the ALU performs the required operation and produces a 64-bit result.

In addition to the result, the ALU generates status flags:

- **Zero Flag (Z):** Asserted when the result equals 64'd0.
- **Carry Flag (C):** Indicates carry-out in unsigned addition/subtraction.
- **Overflow Flag (O):** Indicates signed arithmetic overflow.

For the sequential processor in Phase 1, the **Zero flag** is used for branch decisions (specifically `beq`). The carry and overflow flags are generated for completeness but are not used in control decisions in this phase.

2.1 Supported Operations

The ALU supports the following 64-bit operations:

- Addition
- Subtraction
- AND
- OR
- XOR

- Shift Left Logical (SLL)
- Shift Right Logical (SRL)
- Shift Right Arithmetic (SRA)
- Set Less Than (SLT)
- Set Less Than Unsigned (SLTU)

Although the ALU supports multiple operations, the sequential processor in this phase utilizes only the following subset: `add`, `sub`, `and`, `or`, and operations required for address computation and branch comparison.

2.2 ALU Control Mapping

The ALU operation is selected using a 4-bit control signal generated by the ALU Control block. The mapping between the control signal and the operation performed is shown in Table 1.

ALUControl	**Operation**
0000	ADD
1000	SUB
0111	AND
0110	OR
0100	XOR
0001	SLL
0101	SRL
1101	SRA
0010	SLT
0011	SLTU

Table 1: ALU Control Decoding

2.3 Subtraction using Two's Complement

Subtraction is implemented using two's complement arithmetic. For two 64-bit operands A and B,

$$A - B = A + (\sim B + 1)$$

where $\sim B$ denotes the bitwise complement of B. Thus, subtraction is performed by inverting operand B and adding 1 before performing addition. This allows subtraction to reuse the same adder hardware used for addition.

Signed overflow occurs when the sign of the result differs unexpectedly from the signs of the operands. The overflow flag is asserted accordingly during arithmetic operations.

The modular ALU design enables clean integration into the sequential datapath and allows extension in later phases without structural modification.

3 Sequential Implementation

The processor is implemented as a **single-cycle sequential architecture**, where each instruction completes all stages of execution within one clock cycle. No instruction overlaps with another, and the next instruction begins execution only after the current instruction has fully completed.

In this design, all datapath components operate combinationally within a cycle, while state elements such as the Program Counter (PC) and Register File update on the clock edge. This architecture prioritizes simplicity and clarity over performance and serves as a baseline for future pipelined implementations.

3.1 Key Components of the Sequential Processor

The sequential datapath consists of the following major components:

- **Program Counter (PC):** A 64-bit register that stores the address of the current instruction. The PC is updated every clock cycle. Under normal execution, the PC increments by 4. For branch instructions, it updates to the computed branch target address.
- **Instruction Memory:** A byte-addressable memory that stores program instructions in Big-Endian format. The instruction memory outputs a 32-bit instruction corresponding to the address provided by the PC.
- **Register File:** Contains 32 registers (x0–x31), each 64 bits wide. It supports two simultaneous read ports and one write port. Register x0 is hardwired to zero. The register file writes data on the rising edge of the clock when `RegWrite` is asserted.
- **Immediate Generator:** Extracts and sign-extends immediate fields from the 32-bit instruction according to instruction type (I-type, S-type, or B-type). The output is a 64-bit signed immediate used in ALU operations and address computations.
- **Control Unit:** Decodes the opcode field of the instruction and generates the control signals: `Branch`, `MemRead`, `MemWrite`, `MemtoReg`, `ALUSrc`, `RegWrite`, and `ALUOp`. These signals determine datapath behavior for each instruction.
- **ALU Control Unit:** Receives the 2-bit `ALUOp` signal along with instruction fields (`funct3`, `funct7`) and produces a 4-bit `ALUControl` signal that selects the exact ALU operation.
- **Arithmetic Logic Unit (ALU):** Performs 64-bit arithmetic and logical operations. It generates the result and the zero flag, which is used for branch decisions.
- **Data Memory:** A 1024-byte, byte-addressable memory used for load and store operations. It supports:
  - 64-bit reads when `MemRead` is asserted
  - 64-bit writes when `MemWrite` is asserted

  Memory operations occur synchronously with the clock.
- **Write Back Multiplexer:** Selects between ALU result and memory read data based on the `MemtoReg` control signal. The selected value is written back to the register file.

3.2 Instruction Execution Flow

Each instruction proceeds through the following stages within a single clock cycle:

Step 1: Instruction Fetch

The Program Counter provides the instruction address to the Instruction Memory. The corresponding 32-bit instruction is fetched and supplied to the datapath. Simultaneously, the value `PC + 4` is computed for potential sequential execution.

Step 2: Instruction Decode and Register Read

The Control Unit decodes the opcode field and generates the required control signals. The register file reads the source operands (`rs1`, `rs2`). If the instruction contains an immediate value, it is extracted and sign-extended by the Immediate Generator.

Step 3: Execute

The ALU performs the required operation based on the `ALUControl` signal. For:

- Arithmetic instructions: The ALU computes the result.
- Load/Store instructions: The ALU computes the effective memory address.

- Branch instructions: The ALU compares register operands and sets the zero flag.

Step 4: Memory Access
If the instruction is:

- **Load (ld):** Data is read from the computed memory address.
- **Store (sd):** Data from the register file is written to memory.

For arithmetic and branch instructions, this stage performs no memory operation.

Step 5: Write Back
If `RegWrite` is asserted, the result is written back into the destination register. The data written depends on `MemtoReg`:

- $0 \rightarrow$ ALU result
- $1 \rightarrow$ Memory read data

Step 6: PC Update
The next value of the Program Counter is determined as follows:

- For normal execution: `PC = PC + 4`
- For branch instructions (`beq`): If the zero flag is asserted and `Branch` is active, `PC = PC + immediate offset`

The updated PC value is loaded on the next clock edge, completing the instruction cycle.

4 Test for Sequential Implementation

4.1 Verification Strategy

The sequential processor was verified using a hierarchical testing approach. Each module was first tested independently using dedicated testbenches to ensure correctness in isolation. After validating individual components, the full processor was integrated and tested using the top-level testbench `seq_tb.v`.

All simulations were performed using `iverilog`, and outputs were verified through terminal logs and generated output files.

4.2 Module-Level Testing

Each major module was verified independently before integration. All simulations were performed using `iverilog` and verified through terminal outputs.

Program Counter (PC)

The PC module was verified using both terminal-based simulation output and waveform inspection.

Terminal Verification

```
iverilog PC_tb.v
vvp a.out
```

```
[shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog PC_tb.v
[shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
 VCD info: dumpfile PC_TB.vcd opened for output.

Test Case 1:
Test: pc_in = deadbeefdeadbeef
      reset = 1
      Expected pc_out = 0000000000000000
      Actual pc_out = 0000000000000000
      Status: PASS

Test Case 2:
Test: pc_in = 0000000000000004
      reset = 0
      Expected pc_out = 0000000000000004
      Actual pc_out = 0000000000000004
      Status: PASS

Test Case 3:
Test: pc_in = 0000000080000000
      reset = 0
      Expected pc_out = 0000000080000000
      Actual pc_out = 0000000080000000
      Status: PASS

Test Case 4:
Test: pc_in = fffffffffffffffc
      reset = 0
      Expected pc_out = fffffffffffffffc
      Actual pc_out = fffffffffffffffc
      Status: PASS

Test Case 5:
Test: pc_in = 0000000000001000
      reset = 1
      Expected pc_out = 0000000000000000
      Actual pc_out = 0000000000000000
      Status: PASS

Test Case 6:
Test: pc_in = 0000000000000008
      reset = 0
      Expected pc_out = 0000000000000008
      Actual pc_out = 0000000000000008
      Status: PASS

Test Case 7:
Test: pc_in = 000000000000000c
      reset = 0
      Expected pc_out = 000000000000000c
      Actual pc_out = 000000000000000c
      Status: PASS

Test Case 8:
Test: pc_in = 0000000000000010
      reset = 0
      Expected pc_out = 0000000000000010
      Actual pc_out = 0000000000000010
      Status: PASS

Test Case 9:
Test: pc_in = 0000000000002000
      reset = 0
      Expected pc_out = 0000000000002000
      Actual pc_out = 0000000000002000
      Status: PASS

Test Case 10:
Test: pc_in = 0000000000002004
      reset = 1
      Expected pc_out = 0000000000000000
      Actual pc_out = 0000000000000000
      Status: PASS

=== Test Summary ===
Total Tests: 10
Passed: 10
Failed: 0
==================

PC_tb.v:99: $finish called at 117000 (1ps)
shriyakansal@Shriyas-MacBook-Pro ipa-project %
```

Figure 1: Terminal output of PC testbench execution

The terminal output confirms:

- Correct reset behavior
- Proper PC update on each clock cycle
- Sequential increment by 4

Waveform Verification (GTKWave)

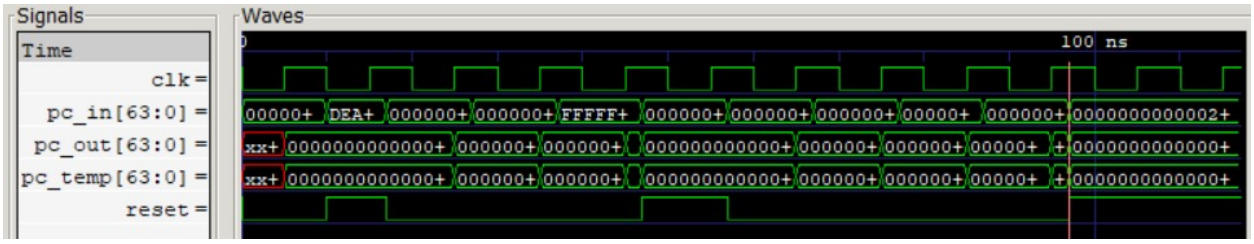

Figure 2: GTKWave timing diagram for Program Counter

Waveform analysis confirms:

- `pc_out` updates only on the rising clock edge
- Reset forces `pc_out = 0`
- No asynchronous glitches are observed

ALU Control Unit

The ALU Control unit was verified to ensure correct mapping between the 2-bit ALUOp and the 4-bit ALU instruction codes.

```
iverilog alu_control_tb.v
vvp a.out
```

```
[shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog alu_control_tb.v
[shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
ALUOp   | Funct3| Funct7        | ALUControl    | Expected      | Type
-------------------------------------------------------------------------------
00      | 000   | 0             | 0000          | 0000          | LD/SD/ADDI
01      | 000   | 0             | 1000          | 1000          | BEQ (SUB)
10      | 000   | 0             | 0000          | 0000          | R-ADD
10      | 000   | 1             | 1000          | 1000          | R-SUB
10      | 111   | 0             | 0111          | 0111          | R-AND
10      | 110   | 0             | 0110          | 0110          | R-OR
11      | xxx   | x             | 0000          | 0000          | Default
-------------------------------------------------------------------------------
alu_control_tb.v:49: $finish called at 70 (1s)
shriyakansal@Shriyas-MacBook-Pro ipa-project %
```

Figure 3: Terminal output of ALU Control testbench execution

Register File

The Register File was tested for dual-read functionality, correct write-back, and enforcement of register x0 being hardwired to zero.

```
iverilog register_file_tb.v
vvp a.out
```

```
[shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog register_file_tb.v
[shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
 VCD info: dumpfile register_file_tb.vcd opened for output.

Test Case 1: 0 hardwired to 0
Inputs:
  read_reg1 =  0
  read_reg2 =  0
  write_reg =  0
  write_data = ffffffffffffffff
  reg_write_en = 1
Status: PASS
  read_data1 = 0000000000000000 (Expected: 0000000000000000)
  read_data2 = 0000000000000000 (Expected: 0000000000000000)

Test Case 2:    Write to reg1
Inputs:
  read_reg1 =  1
  read_reg2 =  0
  write_reg =  1
  write_data = deadbeefdeadbeef
  reg_write_en = 1
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = 0000000000000000 (Expected: 0000000000000000)

Test Case 3:  reg2, read reg1
Inputs:
  read_reg1 =  1
  read_reg2 =  2
  write_reg =  2
  write_data = cafebabecafebabe
  reg_write_en = 1
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)

Test Case 4:   Write disabled
Inputs:
  read_reg1 =  1
  read_reg2 =  2
  write_reg =  1
  write_data = 1111111111111111
  reg_write_en = 0
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)

Test Case 5: ould be ignored)
Inputs:
  read_reg1 =  0
  read_reg2 =  1
  write_reg =  0
  write_data = ffffffffffffffff
  reg_write_en = 1
Status: PASS
  read_data1 = 0000000000000000 (Expected: 0000000000000000)
  read_data2 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)

Test Case 6:     Write to x31
Inputs:
  read_reg1 = 31
  read_reg2 =  1
  write_reg = 31
  write_data = 1234567890abcdef
  reg_write_en = 1
Status: PASS
  read_data1 = 1234567890abcdef (Expected: 1234567890abcdef)
  read_data2 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)

Test Case 7: of same register
Inputs:
  read_reg1 =  2
  read_reg2 =  2
  write_reg =  3
  write_data = aaaaaaaaaaaaaaaa
  reg_write_en = 1
Status: PASS
  read_data1 = cafebabecafebabe (Expected: cafebabecafebabe)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)

=== Test Summary ===
Total Tests: 7
Passed: 7
Failed: 0
==================

register_file_tb.v:148: $finish called at 150000 (1ps)
```

Figure 4: Terminal output of Register File testbench execution

All read and write operations were verified successfully.

Arithmetic Logic Unit (ALU)

The ALU was tested across all supported 64-bit operations including edge cases.

```
iverilog ALU_testcases.v
vvp a.out
```

```
[shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog ALU_testcases.v
[shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
 VCD info: dumpfile alu_tb.vcd opened for output.
 Test 1:
   A: 0000000000000000  B: 0000000000000000  Opcode: 0000
   Result: 0000000000000000  Flags: C=0, O=0, Z=1
 Test 2:
   A: 7fffffffffffffff  B: 0000000000000001  Opcode: 0000
   Result: 8000000000000000  Flags: C=0, O=1, Z=0
 Test 3:
   A: 8000000000000000  B: 8000000000000000  Opcode: 0000
   Result: 0000000000000000  Flags: C=1, O=1, Z=1
 Test 4:
   A: ffffffffffffffff  B: 0000000000000001  Opcode: 0000
   Result: 0000000000000000  Flags: C=1, O=0, Z=1
 Test 5:
   A: ffffffffffffffff  B: ffffffffffffffff  Opcode: 0000
   Result: fffffffffffffffe  Flags: C=1, O=0, Z=0
 Test 6:
   A: 7fffffffffffffff  B: 7fffffffffffffff  Opcode: 0000
   Result: fffffffffffffffe  Flags: C=0, O=1, Z=0
 Test 7:
   A: 8000000000000000  B: ffffffffffffffff  Opcode: 0000
   Result: 7fffffffffffffff  Flags: C=1, O=1, Z=0
 Test 8:
   A: 0000000000000001  B: ffffffffffffffff  Opcode: 0000
   Result: 0000000000000000  Flags: C=1, O=0, Z=1
 Test 9:
   A: 00000000ffffffff  B: 0000000000000001  Opcode: 0000
   Result: 0000000100000000  Flags: C=0, O=0, Z=0
 Test 10:
   A: 06eae7cd9408d55f  B: 0000000aa221d37b  Opcode: 0000
   Result: 06eae7d8362aa8da  Flags: C=0, O=0, Z=0
 Test 11:
   A: 0023185ddfbf101b  B: fffd288475fde3b9  Opcode: 0000
   Result: 002040e255bcf3d4  Flags: C=1, O=0, Z=0
 Test 12:
   A: 0000000100000000  B: ffffffff00000000  Opcode: 0000
   Result: 0000000000000000  Flags: C=1, O=0, Z=1
 Test 13:
   A: 0000000000000000  B: 0000000000000000  Opcode: 1000
   Result: 0000000000000000  Flags: O=0, Z=1
 Test 14:
   A: 0000000000000001  B: 0000000000000001  Opcode: 1000
   Result: 0000000000000000  Flags: O=0, Z=1
 Test 15:
   A: 0000000000000000  B: 0000000000000001  Opcode: 1000
   Result: ffffffffffffffff  Flags: O=0, Z=0
 Test 16:
   A: 8000000000000000  B: 0000000000000001  Opcode: 1000
   Result: 7fffffffffffffff  Flags: O=1, Z=0
 Test 17:
   A: 7fffffffffffffff  B: ffffffffffffffff  Opcode: 1000
   Result: 8000000000000000  Flags: O=1, Z=0
 Test 18:
   A: 0000000000000000  B: 8000000000000000  Opcode: 1000
   Result: 8000000000000000  Flags: O=1, Z=0
 Test 19:
   A: ffffffffffffffff  B: ffffffffffffffff  Opcode: 1000
   Result: 0000000000000000  Flags: O=0, Z=1
 Test 20:
   A: 06eae7cd9408d55f  B: 0000000aa221d37b  Opcode: 1000
   Result: 06eae7c2f1e701e4  Flags: O=0, Z=0
 Test 21:
   A: 0023185ddfbf101b  B: fffd288475fde3b9  Opcode: 1000
   Result: 0025efd969c12c62  Flags: O=0, Z=0
 Test 22:
   A: ffffffffffffffff  B: 0000000000000001  Opcode: 1000
   Result: fffffffffffffffe  Flags: O=0, Z=0
 Test 23:
   A: 0000000000000001  B: ffffffffffffffff  Opcode: 1000
   Result: 0000000000000002  Flags: O=0, Z=0
 Test 24:
   A: ffffffffffffffff  B: 0000000000000000  Opcode: 0111
   Result: 0000000000000000  Zero=1
 Test 25:
   A: 00002c84c4d54177  B: 011c2d636e06d380  Opcode: 0111
   Result: 00002c0044044100  Zero=0
 Test 26:
   A: 5555555555555555  B: aaaaaaaaaaaaaaaa  Opcode: 0111
   Result: 0000000000000000  Zero=1
 Test 27:
   A: 123456789abcdef0  B: ffffffff00000000  Opcode: 0111
   Result: 1234567800000000  Zero=0
 Test 28:
   A: 0000000000000000  B: 0000000000000000  Opcode: 0110
   Result: 0000000000000000  Zero=1
 Test 29:
   A: ffffffffffffffff  B: 0000000000000000  Opcode: 0110
   Result: ffffffffffffffff  Zero=0
 Test 30:
   A: 5555555555555555  B: aaaaaaaaaaaaaaaa  Opcode: 0110
   Result: ffffffffffffffff  Zero=0
```

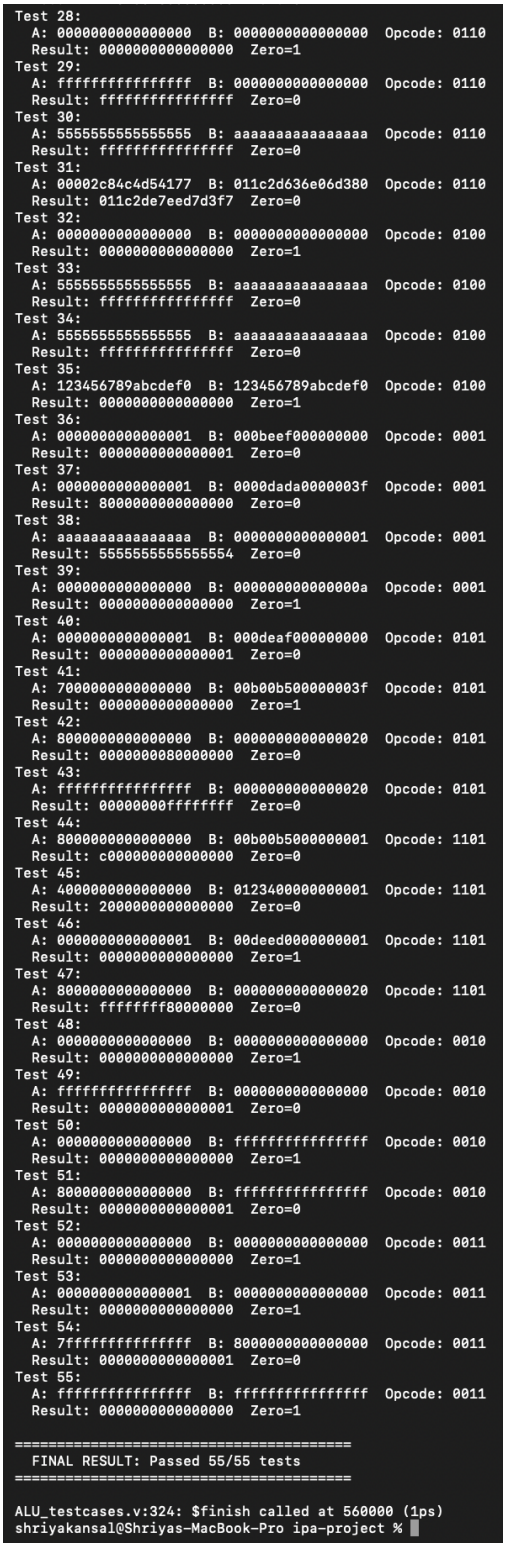

Figure 5: Terminal output of ALU testbench execution

All 55 test cases passed successfully, confirming correct arithmetic and logical functionality.

Data Memory

The Data Memory module was tested using terminal simulation output and GTKWave timing verification.

Terminal Verification

```
iverilog data_mem.v data_mem_tb.v
vvp a.out
```

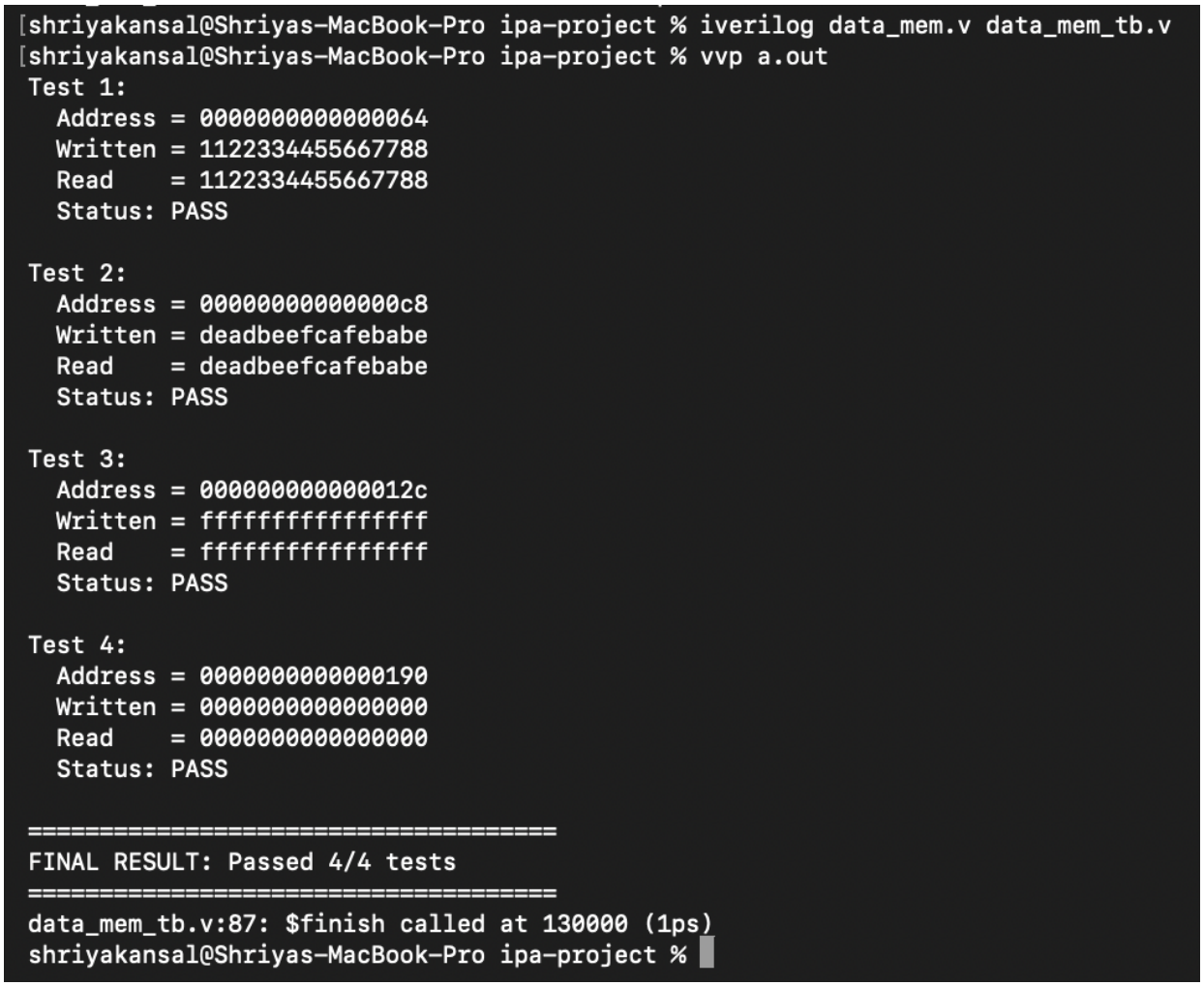

Figure 6: Terminal output of Data Memory testbench execution

Terminal results confirm:

- Correct 64-bit write operations
- Correct 64-bit read operations
- Proper Big-Endian reconstruction of multi-byte data

Waveform Verification (GTKWave)

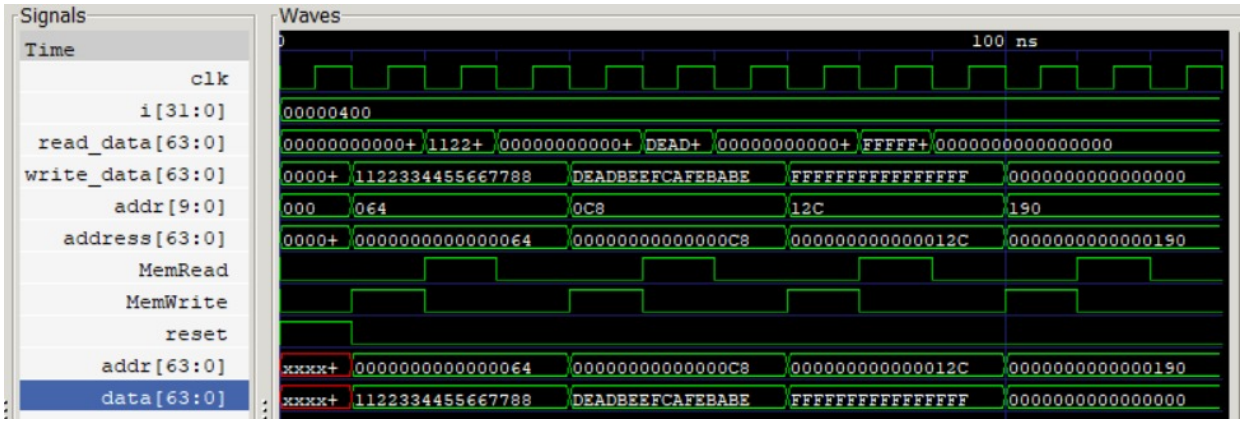

Figure 7: GTKWave timing diagram for Data Memory

Waveform observations:

- `MemWrite` triggers synchronous memory update on clock edge
- `MemRead` produces correct combinational read data
- Address transitions align with expected test cases

Immediate Generator

The Immediate Generator was tested for correct extraction and sign-extension of I-type, S-type, and B-type immediates.

```
iverilog imm_gen_tb.v
vvp a.out
```

```
shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog imm_gen_tb.v
shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
Instruction     | Type  | 64-bit Hex Result     | Value
-----------------------------------------------------------------------
00500093        | I     | 0000000000000005      | 5
fff00093        | I     | ffffffffffffffff      | -1
00113423        | S     | 0000000000000008      | 8
fe000ce3        | B     | fffffffffffffff8      | -8
00000033        | R     | 0000000000000000      | 0
-----------------------------------------------------------------------
imm_gen_tb.v:33: $finish called at 50 (1s)
shriyakansal@Shriyas-MacBook-Pro ipa-project %
```

Figure 8: Terminal output of Immediate Generator testbench execution

All immediate formats were correctly sign-extended to 64 bits.

Control Unit

The Control Unit was verified for correct generation of control signals for all supported instruction types.

```
iverilog control_unit_tb.v
vvp a.out
```

```
shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog control_unit_tb.v
shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
Opcode  | Type  | RegW  | ASrc  | MtoR  | MRead | MWrite| Brch  | ALUOp
--------------------------------------------------------------------------------
0110011 | R     | 1     | 0     | 0     | 0     | 0     | 0     | 10
0010011 | I     | 1     | 1     | 0     | 0     | 0     | 0     | 00
0000011 | LD    | 1     | 1     | 1     | 1     | 0     | 0     | 00
0100011 | SD    | 0     | 1     | 0     | 0     | 1     | 0     | 00
1100011 | BEQ   | 0     | 0     | 0     | 0     | 0     | 1     | 01
0000000 | NULL  | 0     | 0     | 0     | 0     | 0     | 0     | 00
--------------------------------------------------------------------------------
control_unit_tb.v:48: $finish called at 60 (1s)
```

Figure 9: Terminal output of Control Unit testbench execution

Control signals matched expected outputs for each opcode.

4.3 Integrated Processor Testing

Terminal Verification of Integrated Execution

After verifying individual modules, the complete processor was tested using the top-level testbench `seq_tb.v`.

The following command sequence was used:

```
iverilog seq_tb.v
vvp a.out
```

```
shriyakansal@Shriyas-MacBook-Pro ipa-project % iverilog seq_tb.v
shriyakansal@Shriyas-MacBook-Pro ipa-project % vvp a.out
WARNING: ./instruction_mem.v:16: $readmemh(instructions.txt): Not enough words in the file for the requested range [0:4095].
Cycle 1 | PC=0 | Instr=00500113
Cycle 2 | PC=4 | Instr=00a00193
Cycle 3 | PC=8 | Instr=003100b3
Cycle 4 | PC=12 | Instr=40310133
Cycle 5 | PC=16 | Instr=0031f233
Cycle 6 | PC=20 | Instr=0041f2b3
Cycle 7 | PC=24 | Instr=00416333
Cycle 8 | PC=28 | Instr=003163b3
Cycle 9 | PC=32 | Instr=0012b023
Cycle 10 | PC=36 | Instr=0002b503
Cycle 11 | PC=40 | Instr=0062bc23
Cycle 12 | PC=44 | Instr=0182b583
Cycle 13 | PC=48 | Instr=00520463
Cycle 14 | PC=56 | Instr=00a086b3
Cycle 15 | PC=60 | Instr=xxxxxxxx
---- Program Completed ----
Total Cycles = 15
Register file dumped to register_file.txt
seq_tb.v:72: $finish called at 155000 (1ps)
shriyakansal@Shriyas-MacBook-Pro ipa-project %
```

Figure 10: Terminal output of top-level sequential processor testbench (seq_tb.v)

The instruction memory was initialized using `instructions.txt`, containing byte-addressed hexadecimal instructions in Big-Endian format.

During simulation:

- Instructions were fetched sequentially.
- Control signals were generated correctly.
- ALU operations were performed as expected.
- Memory operations were validated for load and store instructions.
- Branch behavior was verified using `beq`.

Terminal outputs confirmed correct execution flow without runtime errors.

Waveform Verification of Integrated Execution

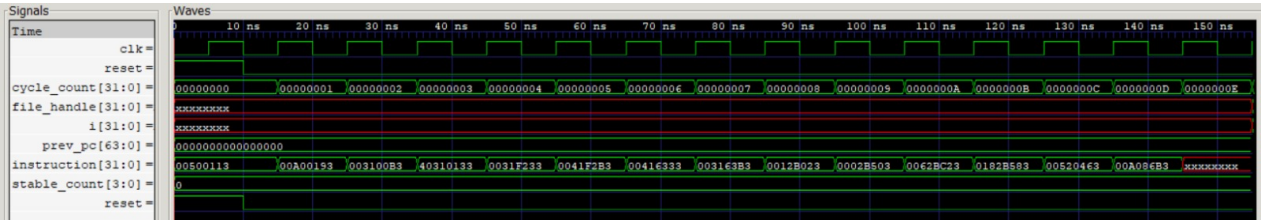

Figure 11: Integrated sequential processor execution waveform

The waveform confirms:

- PC progression in increments of 4
- Correct instruction fetch at each cycle
- Branch decision correctly modifies PC when zero flag is asserted

4.4 Register File Output Verification

At the end of execution, the testbench generated a file named `register_file.txt`, containing:

- 32 lines of 64-bit hexadecimal register values
- Followed by the total number of clock cycles executed

The generated register values were manually verified against the expected results derived from the instruction sequence. The final register state matched the expected output, confirming correct sequential processor functionality.

Screenshots of terminal execution logs and the generated `register_file.txt` file are included to demonstrate successful verification.

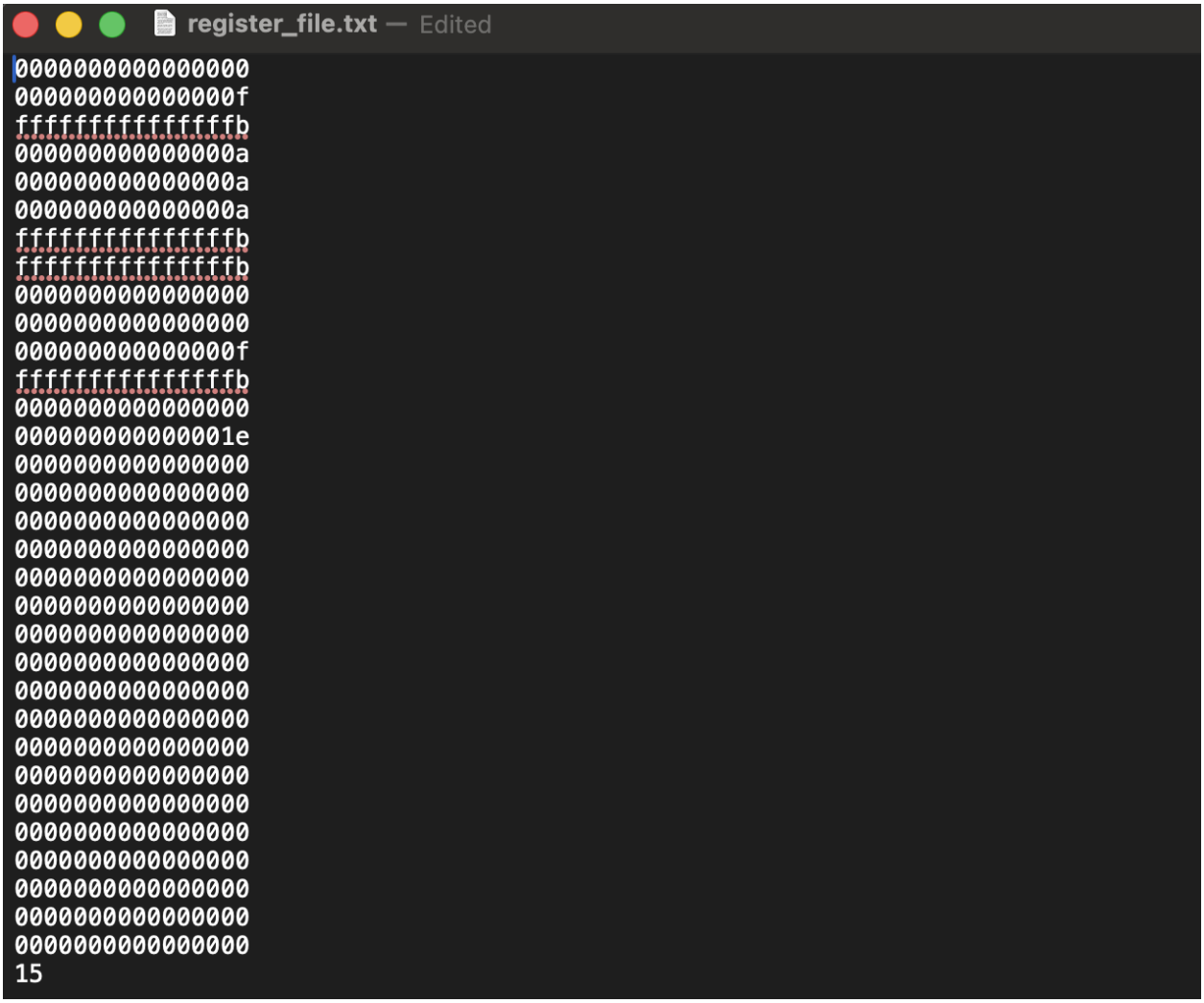

Figure 12: Final register state generated by the sequential processor

5 Design Decisions and Assumptions

64-bit Datapath (RV64I)

The processor was implemented using a 64-bit datapath to align with the RV64I base integer specification. All registers, ALU operations, and memory accesses operate on 64-bit values, ensuring consistency across modules.

Single-Cycle Architecture

A single-cycle (sequential) execution model was chosen for Phase 1 to prioritize simplicity and clarity. All instruction stages (fetch, decode, execute, memory access, and write-back) complete within one clock cycle. While this increases cycle time compared to pipelined architectures, it significantly simplifies control logic and eliminates hazards.

Big-Endian Memory Organization

Both instruction and data memory were implemented using a Big-Endian convention. Instructions were assembled from byte-addressed memory in Big-Endian format, ensuring consistency across modules and alignment with project specifications.

Byte-Addressable Memory

Instruction and data memories were implemented as byte-addressable arrays. Multi-byte values (32-bit instructions and 64-bit data words) were constructed by combining individual bytes. This design provides flexibility and accurately models real hardware memory behavior.

Combinational Read, Synchronous Write Memory

Data memory was implemented with:

- Combinational read logic
- Synchronous write on clock edge

This decision simplifies load execution within a single cycle and avoids introducing additional memory latency in Phase 1.

Branch Resolution in Same Cycle

Branch decisions (`beq`) were resolved within the same clock cycle using the ALU zero flag. The PC update logic directly selects between `PC + 4` and `PC + immediate` based on the `Branch` signal and zero flag condition.

Strict Module Isolation Before Integration

Each module was verified independently using dedicated testbenches before system integration. This modular verification strategy minimized debugging complexity during full integration.

6 Challenges Faced

Data Memory Timing Behavior

One challenge involved determining whether data memory read operations should be synchronous or combinational. Initial implementations caused incorrect load behavior due to timing misalignment. The issue was resolved by implementing combinational read logic while keeping writes synchronous to the clock.

Immediate Field Extraction

Correctly extracting and sign-extending I-type, S-type, and B-type immediates required careful bit-field alignment. Errors in immediate reconstruction initially resulted in incorrect branch offsets and memory addresses.

Branch Offset Computation

For B-type instructions, the immediate field is distributed across non-contiguous bit positions. Ensuring correct bit concatenation, sign-extension, and alignment was critical to proper branch execution.

ALU Control Decoding Consistency

Aligning the 2-bit `ALUOp` from the Control Unit with the final 4-bit `ALUControl` required careful mapping of `funct3` and `funct7` fields. Early mismatches caused incorrect operation selection for R-type instructions.

Instruction Memory Formatting

Instruction memory required strict adherence to byte-level Big-Endian formatting. Improper formatting of `instructions.txt` initially caused incorrect instruction assembly and execution errors.

Full-System Integration Debugging

During integration, interactions between PC update logic, branch conditions, and memory operations required iterative debugging. Independent module correctness did not guarantee correct integrated behavior, necessitating systematic validation using cycle-by-cycle inspection.

7 Contribution

The project was implemented using a structured module-level division of responsibilities. Each team member was assigned specific components of the sequential processor to design, implement, and verify independently before integration.

7.1 Prathish Ghosh

- Implemented `pc.v`
  - 64-bit Program Counter with synchronous update
  - Reset handling
  - PC + 4 computation
- Implemented `instruction_mem.v`
  - Byte-addressable instruction memory (4096 bytes)
  - Reading instructions from `instructions.txt`
  - Big-endian assembly of 32-bit instructions
- Implemented `register_file.v`
  - 32 registers $\times$ 64-bit
  - Dual-read and single-write functionality
  - Register `x0` hardwired to zero
  - Reset clears all registers
- Performed unit testing for `addi` and basic register read/write operations.

7.2 Kavya Veer

- Implemented `control_unit.v`
  - Generated control signals: `Branch`, `MemRead`, `MemWrite`, `MemtoReg`, `ALUSrc`, `RegWrite`, and `ALUOp`
- Implemented `imm_gen.v`
  - I-type, S-type, and B-type immediate extraction
  - 64-bit sign extension
- Implemented `alu_control.v`
  - Decoding `ALUOp` to generate 4-bit `ALUControl`
- Implemented `alu.v`

  - 64-bit arithmetic and logical operations
  - Zero flag generation
- Implemented branch condition logic for `beq`.
- Performed independent testing of ALU and immediate generator modules.

7.3 Shriya Kansal

- Implemented `data_mem.v`
  - 1024-byte, byte-addressable memory
  - Big-endian implementation for `ld` and `sd`
  - 1-cycle memory access behavior
- Implemented `top_cpu.v`
  - Instantiated and interconnected all datapath modules
  - Integrated multiplexers for `ALUSrc`, `MemtoReg`, and PC selection
- Implemented `seq_tb.v`
  - Clock and reset generation
  - Program execution using `instructions.txt`
  - Cycle counting mechanism
  - Automatic register file dump to `register_file.txt`
- Performed full-system integration testing and verified final register outputs.