Design and Implementation of a Single-Cycle RV32I Processor on FPGA Board

Design, Verification, and Functional Testing of a Custom RISC-V Core

Rhythm Dahiya

Indraprastha Institute of Information Technology Delhi (IIIT Delhi)

I. Introduction

The RISC (Reduced Instruction Set Computer) architecture has become the leading model in modern processor design. It focuses on simplicity by using a streamlined instruction set where each instruction executes predictably. This method enables easier hardware implementation, lower power consumption, and improved performance compared to complex instruction set architectures. Among RISC architectures, RISC-V has come forward as a groundbreaking open-source instruction set architecture developed at UC Berkeley. It is free from proprietary licensing restrictions. Its modular and flexible design lets users customise it for specific applications while keeping software compatibility. This has fostered a vibrant community of academic research and commercial innovation, making RISC-V increasingly popular for both educational and industrial applications.

Single-cycle execution is the simplest method for implementing a processor. Here, each instruction completes all operations, fetch, decode, execute, memory access, and write back, within a single clock cycle. While this approach requires a longer clock period to accommodate the slowest instruction, it prevents pipeline hazards and control complexities. This makes the architecture clear and easier to debug. For educational projects and proof-of-concept designs, the single-cycle model serves as a solid foundation for understanding basic processor organisation before moving on to more complex architectures.

Field-Programmable Gate Arrays (FPGAs) combine software flexibility with hardware performance by utilising reconfigurable logic blocks that can create nearly any digital circuit. Unlike ASICs, which require costly fabrication, FPGAs support rapid development cycles and design changes without incurring manufacturing costs. They also come with integrated debugging tools for real-time signal observation. In processor design projects, FPGAs enable the actual implementation and testing of hardware, rather than relying solely on simulations. This provides hands-on experience with timing issues, resource utilisation, and the integration of hardware and software.

This project implements a single-cycle RV32I processor on an FPGA board to gain practical experience in the entire hardware design process, from architectural specification to on-board verification. The RV32I instruction set offers a clear and manageable range, with a limited set of instructions that cover all essential types, showcasing all key processor components. By working on real FPGA hardware, the project addresses real-world challenges such as timing constraints and resource optimisation, while developing practical skills in hardware description languages and FPGA development tools that are applicable in both academic research and industry settings.

II. Design Objectives

The primary objectives of this project are:

- Instruction Set Architecture - Defining and encoding instructions
- Block Diagram Design - Architecture planning
- Control Unit Encoding
- Verilog Coding and Hazard Prevention
- Simulation Testing - Behavioural verification
- Hardware Synthesis - To check whether Verilog code is synthesizable.
- FPGA Testing and Results

III. Instruction Set Architecture

For this single-cycle RV32I processor implementation, a carefully selected subset of instructions has been chosen to demonstrate all fundamental instruction types and processor operations while maintaining managing implementation complexity. The selected instructions cover arithmetic operations, logical operations, memory access, and control flow, providing a representative implementation of the RV32I base integer instruction set.

A. R-Type Instructions

R-Type instructions perform operations between two source registers (rs1, rs2) and store the result in the destination register (rd). The funct7 and funct3 fields will help distinguish between different R-type operations.

The R-Type instructions implemented in this processor are:

- **ADD**: Addition of two register values (rd = rs1 + rs2)
- **SUB**: Subtraction of two register values (rd = rs1 - rs2)
- **AND**: Bitwise AND operation (rd = rs1 & rs2)
- **OR**: Bitwise OR operation (rd = rs1|rs2)
- **SLT**: Set Less Than comparison (rd = (rs1 $<$ rs2) ? 1 : 0)

TABLE I
R-TYPE INSTRUCTION ENCODING (PART 1)

Instruction	[31:25]	[24:20]	[19:15]
	funct7	rs2	rs1
add	0000000	rs2	rs1
sub	0100000	rs2	rs1
slt	0000000	rs2	rs1
or	0000000	rs2	rs1
and	0000000	rs2	rs1

TABLE II
R-TYPE INSTRUCTION ENCODING (PART 2)

[14:12]	[11:7]	[6:0]	Instruction
funct3	rd	opcode	
000	rd	0110011	add
000	rd	0110011	sub
010	rd	0110011	slt
110	rd	0110011	or
111	rd	0110011	and

B. I-Type Instructions

I-Type instructions use a 12-bit immediate value that is sign-extended to 32 bits. The immediate is combined with the rs1 register value, with results stored in rd.

TABLE III
I-TYPE INSTRUCTION ENCODING (PART 1)

Instruction	[31:20]	[19:15]
	imm[11:0]	rs1
addi	imm[11:0]	rs1
lw	imm[11:0]	rs1

TABLE IV
I-TYPE INSTRUCTION ENCODING (PART 2)

[14:12]	[11:7]	[6:0]	Instruction
funct3	rd	opcode	
000	rd	0010011	addi
010	rd	0000011	lw

The I-Type instructions implemented in this processor are:

- **ADDI**: Add immediate value to register (rd = rs1 + imm)
- **LW**: Load Word from memory to register (rd = MEM[rs1 + imm])

C. S-Type Instructions

S-Type instructions store data from register rs2 to memory at an address calculated from rs1 and the sign-extended immediate value. The immediate is split across two fields.

TABLE V
S-TYPE INSTRUCTION ENCODING (PART 1)

Instruction	[31:25]	[24:20]	[19:15]
	imm[11:5]	rs2	rs1
sw	imm[11:5]	rs2	rs1

TABLE VI
S-TYPE INSTRUCTION ENCODING (PART 2)

[14:12]	[11:7]	[6:0]	Instruction
funct3	imm[4:0]	opcode	
010	imm[4:0]	0100011	sw

The S-Type instruction implemented in this processor is:

- **SW**: Store Word from register to memory (MEM[rs1 + imm] = rs2)

D. B-Type Instructions

B-Type instructions compare two registers (rs1, rs2) and conditionally branch to a target address calculated using the sign-extended immediate offset. The immediate represents a 13-bit signed offset in multiples of 2 bytes and is split across multiple fields.

TABLE VII
B-TYPE INSTRUCTION ENCODING (PART 1)

Instruction	[31]	[30:25]	[24:20]	[19:15]
	imm[12]	imm[10:5]	rs2	rs1
beq	imm[12]	imm[10:5]	rs2	rs1

TABLE VIII
B-TYPE INSTRUCTION ENCODING (PART 2)

[14:12]	[11:8]	[7]	[6:0]	Instruction
funct3	imm[4:1]	imm[11]	opcode	
000	imm[4:1]	imm[11]	1100011	beq

The B-Type instruction implemented in this processor is:

- **BEQ**: Branch if Equal (if (rs1 == rs2) PC = PC + imm)

IV. BLOCK DIAGRAM

In this section, all of the individual modules that make up the block diagram will be introduced individually first along with their module diagram, followed by the final block diagram connecting all smaller modules.

A. Program Counter

The Program Counter is a 32-bit sequential register that maintains the address of the current instruction being executed. It operates on the rising edge of the clock signal, updating its output (CurrentPC) with the input value (NextPC) at each clock cycle. The PC increments in multiples of 4 bytes (0, 4, 8, 12, ...), aligning with the 32-bit instruction word size in the RV32I architecture. As a sequential circuit implemented using an always @(posedge clk) block, it ensures synchronised instruction fetch operations throughout the processor pipeline.

The module diagram is shown in Figure 1.

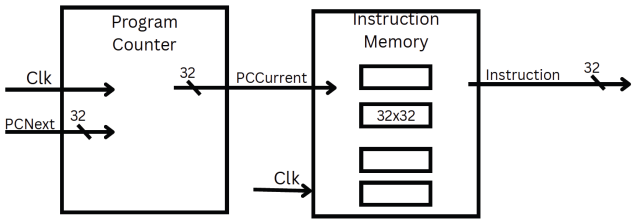

FIGURE 1. Program Counter and Instruction Memory

B. Instruction Memory

The Instruction Memory is a 32-bit × 32-bit memory module that stores program instructions. It takes the CurrentPC and Clk as input and outputs a 32-bit Instruction signal. The memory features asynchronous read and synchronous write operations, with the clock signal used for write synchronisation. Due to the asynchronous read operation, Vivado synthesises this design using Look-Up Tables (LUTs) rather than BRAM. Instructions are preloaded into the memory before execution. The 32-instruction capacity is sufficient for the project's testing requirements and can be easily expanded by adjusting the address width.

The module diagram is shown in Figure 1.

C. Register File

The Register File is a 32×32-bit memory module containing 32 general-purpose registers. It takes as inputs a clock signal, a 32-bit Write Data signal, three 5-bit Read Address signals (RA1, RA2, RA3), a Write Enable signal, and a Reset signal. The module outputs two 32-bit Read Data signals (RD1, RD2) corresponding to the first two read addresses. The register file supports simultaneous dual-port asynchronous read and single-port synchronous write operations. The x0 register is hardwired to zero and remains constant regardless of any write operations to it. The Reset signal initialises all registers to zero, providing a known state at the start of execution.

The module diagram is shown in Figure 2.

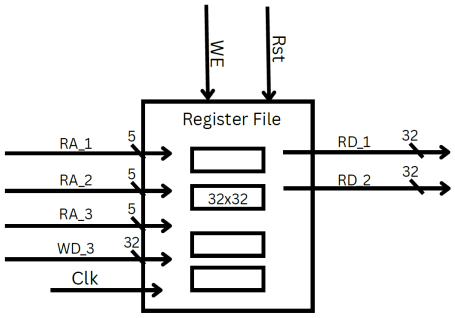

FIGURE 2. Register File

D. ALU

The Arithmetic Logic Unit is a purely combinational module that performs arithmetic and logical operations. It takes as inputs two 32-bit data signals (Src1 and Src2) and a 3-bit ALU Control signal that selects the specific operation to be executed. The module outputs a 32-bit Result signal containing the operation outcome and a 1-bit Zero flag signal used for branch decision logic. The Zero flag is asserted when the ALU result equals zero, enabling the processor to determine whether conditional branch instructions should be taken. As a combinational circuit, the ALU output updates immediately based on the input values without requiring a clock signal.

The module diagram is shown in Figure 3.

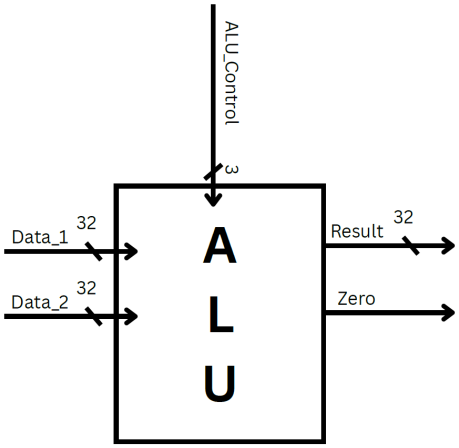

FIGURE 3. ALU

E. Data Memory

The Data Memory is a 32×128-bit memory module used for load and store operations. It takes as inputs a 32-bit Address signal, a clock signal, a 32-bit Write Data signal, and a Write Enable signal. The module outputs a 32-bit Read Data signal. Data Memory operates with synchronous write, where data is written to the specified address on the rising edge of the clock when Write Enable is asserted, and asynchronous read, allowing immediate data retrieval based on the input address. This memory module provides the processor with the ability to access and modify data during program execution.

The module diagram is shown in Figure 4.

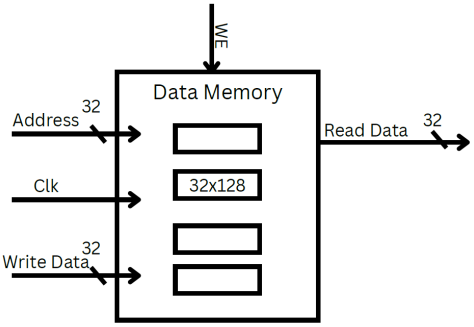

FIGURE 4. Data Memory

F. Multiplexers

The processor design incorporates three key multiplexers for data path selection:

The ResultSrc Multiplexer selects between the 32-bit ALU Result and the 32-bit Data Memory Read Data. The ResultSrc control signal determines which input is forwarded to the register file's Write Data input, enabling both arithmetic/logical results and loaded data to be written back to registers. Shown in Figure 5.

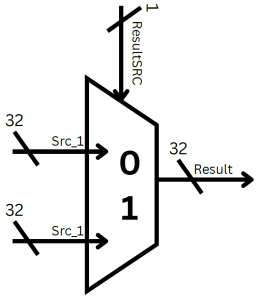

FIGURE 5. ResultSrc Mux

The ALUSrc Multiplexer chooses between the second register read data (RD2) from the register file and the 32-bit sign-extended immediate value. The ALUSrc control signal selects which input is forwarded to the second operand of the ALU, supporting both register-register and register-immediate operations. Shown in Figure 6.

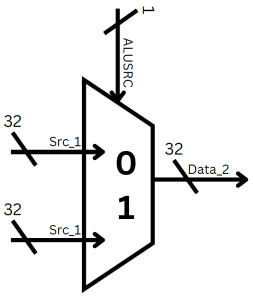

FIGURE 6. ALUSrc Mux

The PCSrc Multiplexer selects between PC+4 (sequential execution) and PC+4+sign-extended immediate (branch target address). The PCSrc control signal, generated by the AND gate that combines the Branch signal and Zero flag, determines which address is forwarded to PCNext, thereby controlling program flow for branch instructions. Shown in Figure 7.

G. Immediate Extension Unit

The Immediate Extend Unit extracts and sign-extends immediate values from instructions. It takes the complete 32-bit instruction as input along with a 2-bit ImmSrc control signal. The ImmSrc signal determines which bits of the instruction

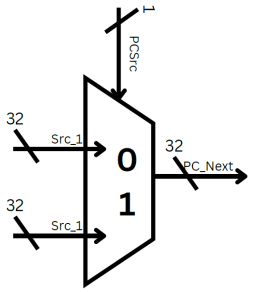

FIGURE 7. PCSrc Mux

correspond to the immediate field based on the instruction format (I-type, S-type, B-type, etc.). The unit then performs sign extension on the extracted immediate value to produce a 32-bit output, preserving the sign of the original immediate for proper arithmetic operations.

The module diagram is shown in Figure 8.

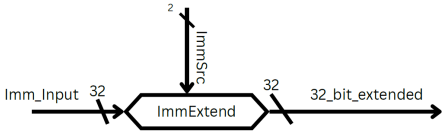

FIGURE 8. Immediate Extender

H. Next PC Calculation Modules

The processor includes two dedicated modules for calculating the next Program Counter value: The PC + 4 Module performs a simple addition to compute the address of the next sequential instruction. It takes the CurrentPC as input and adds 4 to it, producing the PC + 4 output.

The PC Target Module calculates the branch target address by adding the output of PC + 4 module to the sign-extended immediate value. The result represents the potential branch destination address that will be selected if the branch condition is satisfied.

The module diagram is shown in Figure 9.

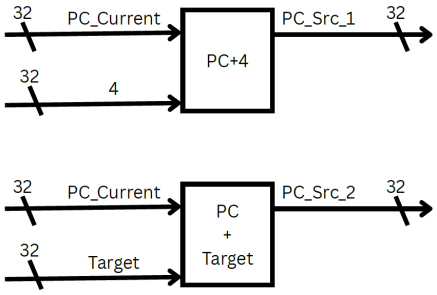

FIGURE 9. Next PC Calculation

I. Control Unit Module

The Control Unit is the most critical module in the processor, responsible for generating all control signals that coordinate the operation of other modules. It takes as inputs the 7-bit OpCode, 3-bit Funct3, and 7-bit Funct7 fields from the instruction. Based on these inputs, the Control Unit decodes the instruction type and generates the appropriate control signals: ImmSrc (immediate format selection), Rst (reset), MemtoReg (memory to register), MemWrite (memory write enable), Branch (branch enable), ALUSrc (ALU source selection), RegWrite (register write enable), and ALUControl (ALU operation selection). The Control Unit is implemented as a combinational logic circuit, consisting of a Main Decoder that processes the OpCode and an ALU Decoder that determines the specific ALU operation based on Funct3, Funct7, and ALUOp signals. This hierarchical design ensures efficient instruction decoding and proper control signal generation for all RV32I instructions.

The module diagram is shown in Figure 10.

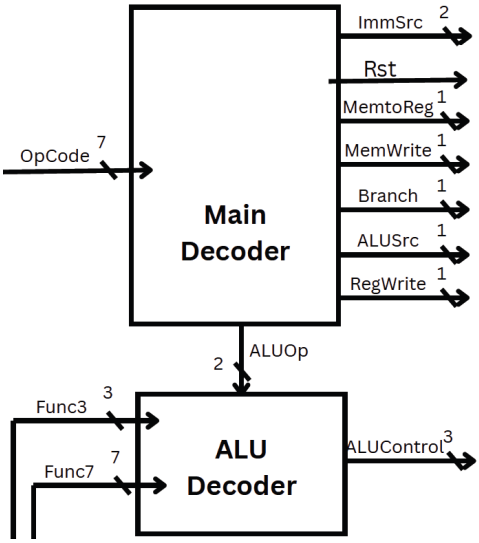

FIGURE 10. Control Unit

J. Top Module

The Top Module serves as the highest level of hierarchy in the single-cycle RV32I processor design. It instantiates and interconnects all individual modules including the Program Counter, Instruction Memory, Register File, ALU, Data Memory, Immediate Extend Unit, Control Unit, multiplexers, and Next PC calculation modules.

The module diagram is shown in Figure 11.

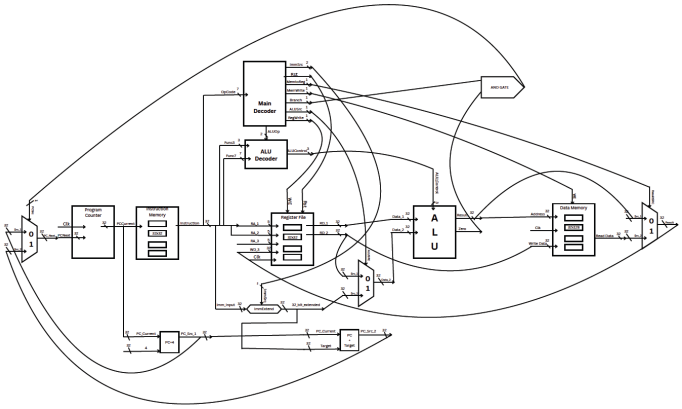

FIGURE 11. Top Module

V. Control Unit Encoding

This section presents the control signal encodings for the implemented RV32I instructions (R-type, lw, sw, beq, and addi). The tables detail how each instruction's opcode maps to specific control signals that coordinate the processor's data path.

TABLE IX shows the basic control signals (RegWrite, ALUSRC, Branch, MemWrite), while TABLE X displays the remaining signals (ResultSRC, ImmSRC, ALUControl, PCSrc).

TABLE IX
Control Signals - RegWrite, ALUSRC, Branch, MemWrite

Instruction	Opcode	RegWrite	ALUSRC	Branch	MemWrite
R-type	0110011	1	0	0	0
lw	0000011	1	1	0	0
sw	0100011	0	1	0	1
beq	1100011	0	0	1	0
addi	0010011	1	1	0	0

TABLE X
Control Signals - ResultSRC, ImmSRC, ALUControl,PCSrc

Instruction	ResultSRC	ImmSRC	ALUControl	PCSrc
R-type	0	X	func	0
lw	1	00	010	0
sw	X	01	010	0
beq	X	10	110	Branch & Zero
addi	0	00	010	0

VI. Verilog Coding and Hazard Prevention

The complete Verilog implementation of all processor modules is available on the GitHub repository and can be accessed for detailed code review.

The Verilog code has been specifically written to be synthesizable and not just simulation-ready, as some codes can be. The implementation adheres to common hardware coding practices to ensure proper synthesis and avoid potential issues:

- When modeling sequential logic, non-blocking assignments (<=) are used.
- When modeling combinational logic with an always block, blocking assignments (=) are used.
- Blocking and non-blocking assignments are never mixed in the same always block.
- The @* sensitivity list is used to automatically include all desired identifiers for combinational logic.
- All branches of if and case statements are included to prevent unintended latch inference.
- Outputs are assigned in all branches of conditional statements.
- Default values for outputs are assigned at the beginning of always blocks to ensure complete signal assignment

Preventing Hazards

In a single-cycle processor with synchronous write operations, a critical Read-After-Write (RAW) hazard can occur. Since the Register File performs synchronous writes, data written in cycle N is only updated in the register at the start of cycle N+1. If a write instruction is immediately followed by a read instruction that accesses the same register, the read operation will retrieve stale data from cycle N instead of the updated value, causing a data discrepancy.

To prevent this hazard, several solutions exist, including data forwarding and pipeline stalls. For this implementation, the chosen approach is to insert a NOP (no operation) instruction between the write and the subsequent read instruction. This ensures that the register write completes before the dependent read occurs, eliminating the data hazard at the cost of one additional clock cycle. While this solution reduces performance compared to hardware forwarding techniques, it provides a straightforward and effective method to maintain data consistency in the single-cycle design.

VII. Simulation Testing

Simulation testing is essential for verifying the logical correctness of the processor design, though it does not guarantee synthesizability. For testing purposes, the top module exposes key internal signals including the 32×32-bit Register File contents, the current Instruction, and the Program Counter (PC).

Three test programs have been pre-loaded in our instruction memory to verify different categories of RV32I instructions. By observing the simulation waveforms generated during execution, we can verify that each instruction executes correctly, registers update with expected values, and the control flow behaves as designed.

A. Branch Instructions,Addi and Reset

For this test program, the assembly instructions have been provided in TABLE XI, and the binary instruction codes are available in the GitHub repository.

Figure 12 shows the execution of the 5th instruction at mem[4], which is a branch equal (beq) instruction comparing registers x1 and x2. At this point, both x1 and x2 contain the value 5, satisfying the branch condition. As a result, the branch is taken, and the Program Counter updates to PC + 4 + 8, jumping forward to mem[7].

Figure 13 demonstrates the instruction at mem[8], where x1 and x4 are compared (containing values 5 and 0, respectively). Since the values are not equal, the branch condition is not satisfied, and the processor continues sequential execution with PC+4.

Figure 14 shows the final branch instruction at mem[11], which compares x1 and x2 (both now containing the value 5). The branch condition is satisfied, causing the processor to jump backwards with PC + 4 - 12. This creates an infinite loop as the processor repeatedly jumps back to mem[9], continuously executing the same sequence of instructions.

Figure 15 demonstrates the addi instruction at mem[1], which adds the immediate value 5 to register x0 and stores the result in register x1, updating x1 to the value 5.

Figure 16 shows the reset instruction at mem[0]. When executed, the reset signal causes all registers in the Register File to transition from undefined values (xxxxx) to zero (00000), initializing the processor to a known state before program execution begins.

TABLE XI
Test Program Part 1: Branch Taken, Branch Not Taken, Branch Taken with Negative Immediate, Add Immediate and Reset

Address	Instruction
mem[0]	rst (custom opcode)
mem[1]	addi x1, x0, 5
mem[2]	addi x2, x0, 5
mem[3]	nop (RAW before beq)
mem[4]	beq x1, x2, +8 (taken forward)
mem[5]	addi x9, x0, 99 (skipped)
mem[6]	addi x2, x0, 7
mem[7]	nop (RAW before beq)
mem[8]	beq x1, x4, +8 (not taken)
mem[9]	addi x2, x1, 0
mem[10]	nop (RAW before backward beq)
mem[11]	beq x1, x2, -12 (taken backward)
mem[12]	addi x3, x0, 1
mem[13]	addi x4, x0, 2

B. R-Type Instructions

For this test program, the assembly instructions have been provided in TABLE XII, and the binary instruction codes are available in the GitHub repository.

Figure 17 shows the state of the registers x0,x1,x2,x3,x4,x5,x6 and x7 after the last instruction has been executed. Expected values of these registers are also written in the provided table.

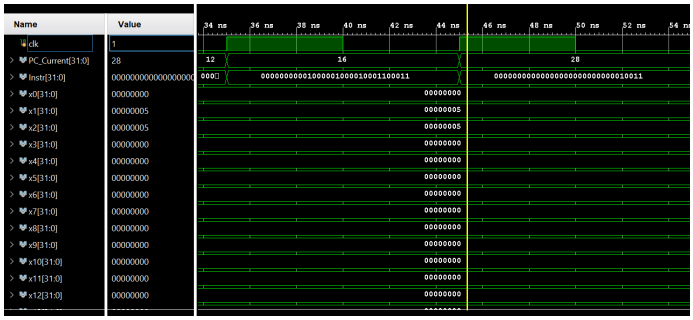

FIGURE 12. Branch Taken with Positive Immediate

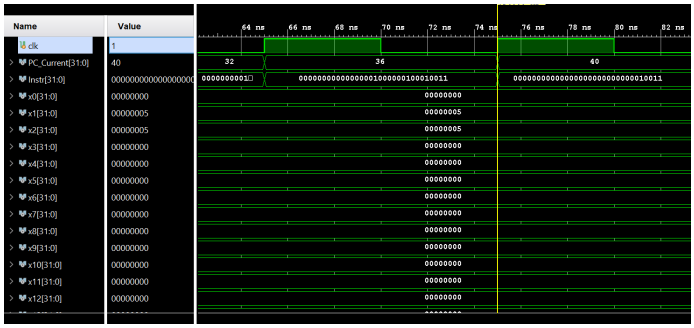

FIGURE 13. Branch Not Taken

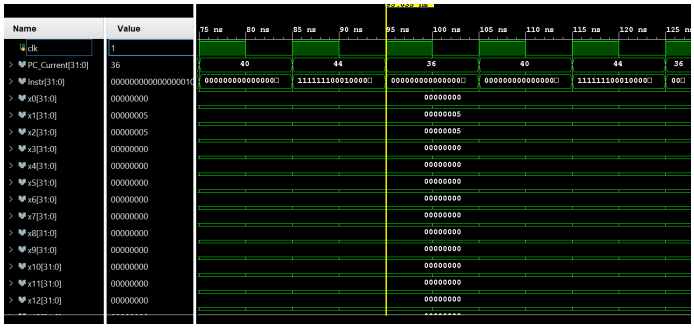

FIGURE 14. Branch Taken with Negative Immediate

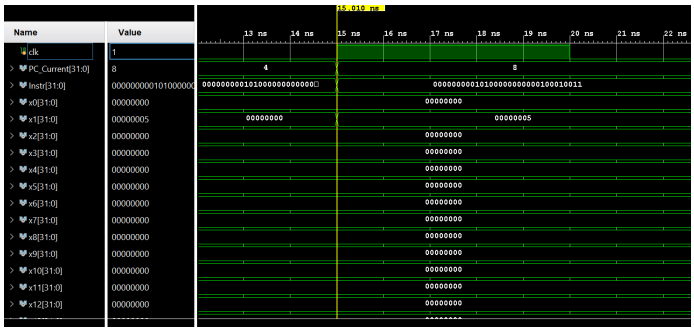

FIGURE 15. Add Immediate Instruction

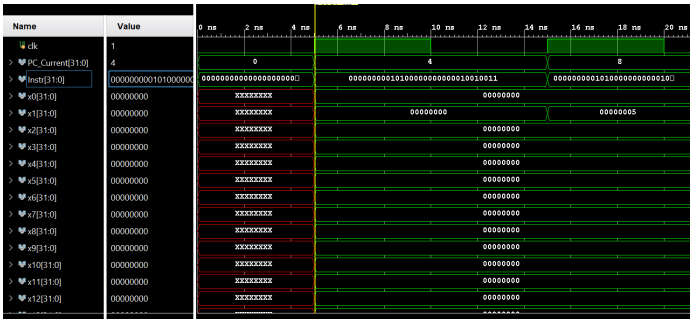

FIGURE 16. Reset Registers Instruction

TABLE XII
TEST PROGRAM PART 1: R-TYPE INSTRUCTIONS

Address	Instruction
mem[0]	rst (custom opcode)
mem[1]	addi x1, x0, 10
mem[2]	nop (RAW: x1)
mem[3]	addi x2, x0, 5
mem[4]	nop (RAW: x2)
mem[5]	add x3, x1, x2 (10 + 5 = 15)
mem[6]	nop (RAW: x3)
mem[7]	sub x4, x1, x2 (10 - 5 = 5)
mem[8]	nop (RAW: x4)
mem[9]	and x5, x1, x2 (10 & 5 = 0)
mem[10]	nop (RAW: x5)
mem[11]	or x6, x1, x2 (10 | 5 = 15)
mem[12]	nop (RAW: x6)
mem[13]	slt x7, x2, x1 (5 < 10 = 1)

C. Load and Store Instructions

For this test program, the assembly instructions have been provided in TABLE XIII, and the binary instruction codes are available in the GitHub repository.

Instead of exposing the Data Memory signals externally like the Register File, a more effective verification approach is to validate memory operations through register values. This test program demonstrates this methodology by storing data from one register into memory and then loading it back into a different register.

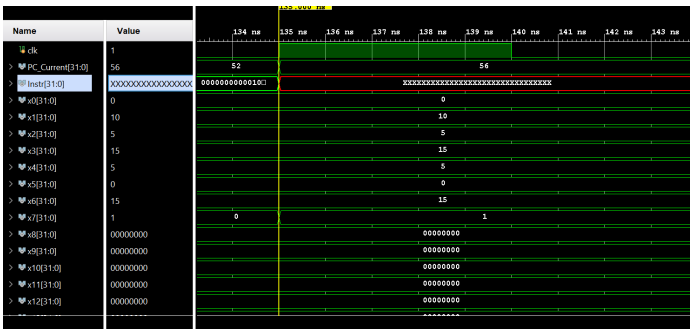

FIGURE 17. R-Type Instructions

By seeing Figure 18, we can verify that x3 successfully receives the value 42, confirming that both the store and load operations executed correctly. This approach validates memory functionality without requiring direct visibility into the Data Memory contents.

TABLE XIII
TEST PROGRAM: LOAD AND STORE INSTRUCTIONS

Address	Instruction
mem[0]	rst (custom opcode)
mem[1]	addi x1, x0, 42 (x1 = 42, data)
mem[2]	addi x2, x0, 16 (x2 = 16, base address)
mem[3]	nop (RAW safety)
mem[4]	sw x1, 0(x2) (MEM[16] = 42)
mem[5]	nop (RAW safety)
mem[6]	lw x3, 0(x2) (x3 = MEM[16] = 42)

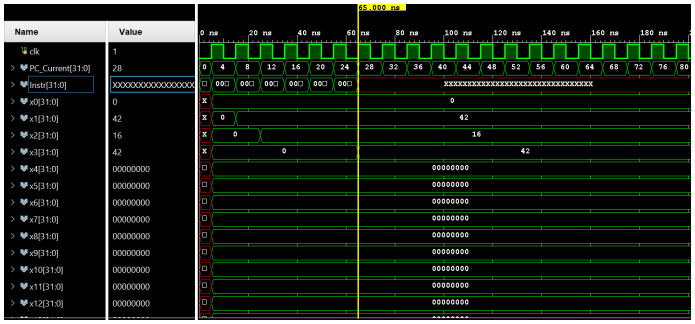

FIGURE 18. Load and Store Instructions

With these three programs, the processor's logic has been successfully verified.

VIII. HARDWARE SYNTHESIS

The Hardware Synthesis of the processor does not produce any errors, hence the Verilog code is completely synthesizable. The Hardware Utilisation is shown in the following tables.

TABLE XIV
HARDWARE UTILIZATION AFTER SYNTHESIS ON VIVADO - PART 1

Name	Slice LUTs (53200)	Slice Registers (106400)
top_fpga	646	1028
cpu (Top_Module)	646	1028

TABLE XV
HARDWARE UTILIZATION AFTER SYNTHESIS ON VIVADO - PART 2

Name	F7 Muxes (26600)	F8 Muxes (13300)
top_fpga	96	32
cpu (Top_Module)	96	32

IX. TESTING ON FPGA BOARD

Physical access to an FPGA board is not available to me at this time, so it is not possible to directly display register values or the Program Counter on LEDs. The Xilinx Virtual Input/Output (VIO) and Integrated Logic Analyser (ILA) IP cores are utilised for testing and verification.

A wrapper module, topfpga, is created to contain the CPU module along with the VIO and ILA cores. The VIO provides a 100 MHz clock signal and a clock enable signal, allowing manual control over program execution start time. This requires minimal modifications to the Program Counter and Register File modules to incorporate the clock enable functionality.

The ILA is configured to capture and display waveforms for critical signals, including the Program Counter (PC), the current Instruction, and three registers used in the test program (x1, x2, x3). The same test program used for validating load and store instructions(TABLE XIII) during simulation is employed for FPGA testing, where the expected values are x1 = 42, x2 = 16, and x3 = 42.

Once the topfpga design is completed, a bitstream is generated and the FPGA is programmed remotely using IIITD Cloud Labs Platform. Upon execution, the ILA captures the waveforms, confirming that the output matches expectations: x1 contains 42, x2 contains 16, and x3 successfully loads the value 42 from memory. The program executes in a continuous loop, with the reset signal periodically clearing all registers to zero before execution resumes. Waveform is shown in Figure 19.

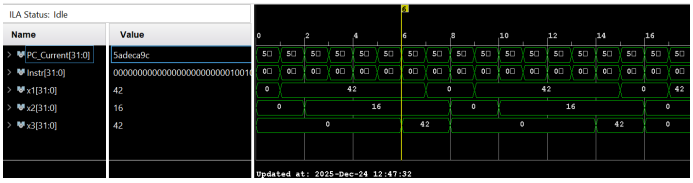

FIGURE 19. ILA Waveform

This successful FPGA implementation confirms that the design is fully synthesizable and operates correctly on hardware, validating both the logical correctness and physical realiability of the single-cycle RV32I processor.

X. CONCLUSION AND FUTURE WORK

In this project, I successfully designed and implemented a fully functional single-cycle RV32I processor on an FPGA board. The implementation covered the complete processor architecture, including the Program Counter, Instruction Memory, Register File, ALU, Data Memory, Control Unit, and all necessary multiplexers and data path components. I coded the design in Verilog, following synthesizable hardware design practices, to ensure compatibility with FPGA implementation.

I conducted comprehensive simulation testing using three test programs covering R-type instructions, load/store operations, and branch instructions. The simulation results verified the correct execution of instructions, proper register updates, and accurate control flow management. Hardware synthesis on Vivado demonstrated efficient resource utilization with 646 Slice LUTs, 1028 Slice Registers, 96 F7 Muxes, and 32 F8 Muxes. I performed remote FPGA testing using VIO and ILA cores, which confirmed that the processor operates correctly on physical hardware, validating both the logical correctness and synthesizability of my design.

In the near future, I plan to upgrade this processor to implement a pipelined architecture with five stages (Fetch, Decode, Execute, Memory, and Write-back) to improve instruction throughput and overall performance. Additional enhancements I intend to implement include data forwarding mechanisms to reduce pipeline stalls, hazard detection units, and expansion of the instruction set to support more RV32I instructions. I will also explore further optimisation of hardware resource utilisation and clock frequency improvement to achieve better performance metrics on the FPGA platform.