MJSAT 6 (S2) 80-92

MJSAT

Malaysian Journal of Science and Advanced Technology

journal homepage: https://mjsat.com.my/



Design and FPGA Implementation of a 32-Bit Single-Cycle RISC-V Processor

Jun Wei Siaw[1], and King Lee Chua*[1]

[1]*Faculty of Electrical and Electronic Engineering, Universiti Tun Hussein Onn Malaysia, 86400 Parit Raja, Batu Pahat, Johor, Malaysia.*

KEYWORDS

RISC-V
Single-Cycle Processor
RV32I
FPGA
Verilog HDL

ARTICLE HISTORY

Received 27 July 2026
Accepted 6 October 2026
Available online 20 October 2026

ABSTRACT

The Reduced Instruction Set Computer Five (RISC-V) instruction set architecture (ISA) is open-source and has become increasingly prominent in computer architecture education due to its flexibility and accessibility. However, most educational implementations rely primarily on simulation, with limited emphasis on hardware validation. This paper presents the design, implementation and hardware validation of a 32-bit single-cycle RISC-V processor based on the RV32I instruction set. The processor was developed in Verilog HDL with a complete integration of datapath and control unit. Functional verification was first performed in ModelSim-Altera using six instruction categories, followed by synthesis and deployment on a Terasic DE1-SoC FPGA Development board using Intel Quartus Prime Lite. A total of 31 RV32I instructions were successfully implemented and verified through simulation, while 29 instructions were further validated on the Field-Programmable Gate Array (FPGA) using five representative applications, including a password lock system and a temperature-controlled fan controller. The implementation utilized only 17% of the FPGA logic elements, achieved a maximum operating frequency of 53.31 MHz and consumed 422.07 mW of power. These results confirm the functional correctness and hardware efficiency of the designed processor, demonstrating FPGA-based RISC-V development as a practical platform for bridging the gap between simulation-based learning and hands-on processor architecture education.

© 2026 The Authors. Published by Penteract Technology.
This is an open access article under the CC BY-NC 4.0 license (https://creativecommons.org/licenses/by-nc/4.0/).

1. INTRODUCTION

The RISC-V instruction set architecture (ISA) is an open-source ISA originally developed at the University of California, Berkeley, created to offer a modern, royalty-free alternative to proprietary architectures such as ARM and x86 [1]-[2]. Unlike commercial ISAs, which require licensing fees and restrict access to internal design details, RISC-V is fully open and freely extensible. This openness has driven its rapid adoption in embedded systems, IoT, robotics, and high-performance computing [3]. Within the RISC-V ecosystem, the RV32I base integer instruction set is particularly attractive for education and prototyping: it is simple, modular, and free of the legacy complexity found in commercial ISAs, allowing designers to clearly observe and understand instruction execution, datapath behaviour, and control logic [1], [4]-[5].

Field-Programmable Gate Arrays (FPGAs) complement this openness by providing a reconfigurable hardware platform for implementing, validating, and refining custom processor designs without the high cost and inflexibility of Application-Specific Integrated Circuits (ASICs) [6]-[7]. Unlike microcontrollers, which have fixed internal architectures, FPGAs allow a fully custom ISA such as RISC-V to be realized and observed at the hardware level. Together, RISC-V and FPGAs form a natural pairing for processor design education: one supplies an architecture that is simple enough to fully understand, and the other supplies a platform on which that architecture can be built and physically verified. In particular, the single-cycle architecture, where each instruction completes execution within one clock cycle, provides maximum transparency for tracing the relationship between an instruction and its datapath operation, making it especially suitable for educational processor design [8]-[9].

Despite this potential, most educational RISC-V implementations stop at the simulation stage. Previous studies, including [10]-[12], have demonstrated the correctness of

*Corresponding author:
E-mail address: King Lee Chua < chua@uthm.edu.my >.
https://doi.org/10.56532/mjsat.v6iS2.821
2785-8901/ © 2026 The Authors. Published by Penteract Technology.
This is an open access article under the CC BY-NC 4.0 license (https://creativecommons.org/licenses/by-nc/4.0/).

RV32I datapath and control logic through simulation, but they do not extend their validation to FPGA implementation. Conversely, studies [13]-[15] have implemented RISC-V cores on FPGA platforms, demonstrating hardware feasibility, but with limited instruction coverage or only basic functional demonstration. A separate body of work [16]-[20] emphasizes performance enhancement through techniques such as pipelining and low-power optimization. While these approaches improve processor throughput and energy efficiency, they also increase design complexity, making them less suitable for students who are new to processor architecture. Across the literature, a gap remains: few designs integrate comprehensive RV32I instruction coverage, rigorous simulation-based verification, and FPGA-based hardware validation with realistic, interactive applications rather than isolated test programs.

This paper addresses this gap by presenting the design, simulation, and FPGA implementation of a 32-bit single-cycle RISC-V processor based on the RV32I instruction set. The objectives of this work are to design a 32-bit single-cycle RISC-V processor architecture that supports the RV32I instruction set, verify its functional correctness through simulation with ModelSim-Altera, and implement and validate the processor on a Terasic DE1-SoC FPGA development board. To demonstrate practical functionality beyond isolated instruction tests, the designed processor is tested through five embedded application demonstrations: a password lock security system, a parking lot management system, a game score-ranking system, a temperature monitoring system with fan speed control, and a menu navigation controller. These applications exercise the processor across a range of realistic scenarios using the FPGA board's onboard peripherals to provide a comprehensive hardware-level functional validation. This paper is organized as follows: Section II describes the processor architecture and FPGA implementation methodology, Section III presents the simulation and hardware validation results, and Section IV concludes with key findings and outlines directions for future work.

2. METHODOLOGY

2.1 Development Workflow

The RISC-V processor was developed through a structured design flow to ensure functional correctness before hardware deployment. The process began by defining the processor architecture and the scope of the RV32I instruction set, after which development proceeded along two parallel tracks: the datapath and the control unit. This modular approach enabled each unit to be designed and verified independently. Each module was described in the Verilog hardware description language (HDL) and validated through functional simulation; decision points were included to verify that each unit met its functional requirements, and iterative refinement was used if it failed. While formal verification methods [21] and cross-level testing approaches [22] offer rigorous correctness guarantees for RISC-V designs, simulation-based verification was selected in this work for its direct observability of signal-level behaviour and suitability for an educational context. Once both units were individually verified, they were integrated and tested at the instruction level to ensure correct processor operation. The verified design was then synthesized and implemented on the FPGA, followed by hardware testing using onboard peripherals. Any failures identified during hardware testing were traced back to the corresponding design stage for refinement before revalidation.

2.2 Datapath Design

Fig. 1 illustrates the processor datapath architecture, which integrates all functional units required to execute each instruction within a single clock cycle. The instruction fetch, decode, execute, memory access, and write-back stages are completed in a single cycle instead of multiple cycles. The Program Counter (PC) provides the current instruction address to instruction memory, while a PC+4 adder computes the next sequential address and a separate PC+immediate adder computes branch/jump target address using the sign-extended output of the immediate generator. The fetched instruction is decoded to extract the opcode, *funct3*, *funct7*, and register fields, which drive the register file and control unit. Two multiplexers determine the Arithmetic Logic Unit (ALU) operands: *ALU_A* selects the first operand from the PC, zero, or a register, whereas *ALU_B* selects the second operand from a register or the generated immediate. For branch instructions, a branch comparator evaluates the register operands and produces a branch-decision signal for the PC multiplexer selection. For jump instructions, a dedicated multiplexer selects either the PC-relative target address or the ALU-computed target address. For load/store instructions, the ALU result serves as the data memory address, and a *MemtoReg* multiplexer selects the value written back to the register file from the ALU result, memory output, or the PC+4 value for jump instructions.

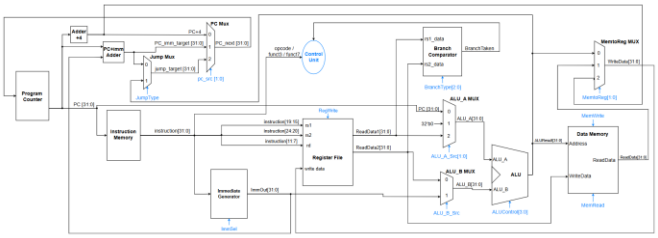


Fig. 1. The processor Datapath Architecture

2.3 Control Unit Design

The control unit, as shown in Fig. 2, decodes the instruction opcode and the *funct3* field to generate the control signals required for datapath operation, including register write enable, memory access control, ALU operand selection, and program counter control. A hierarchical structure is adopted in which the main control unit produces a high-level *ALUOp* signal and other datapath control signals including *RegWrite*, *MemRead*, *MemWrite*, *MemtoReg*, *ALU_A_Src*, *ALU_B_Src*, *BranchType*, *JumpType*, *ImmSel*, and *pc_src*. These signals are then complemented by a dedicated ALU control unit, which combines *ALUOp* with the *funct3* and *funct7* fields to produce the *ALUControl* signal that specifies the precise arithmetic or logic operations to be performed. This hierarchical organization simplifies the main control logic while enabling exact ALU operation decoding for all the supported RV32I instructions.

For a conditional branch instruction, a branch comparator generates equality and signed/unsigned relational signals (eq, lt, ltu). The branch decision logic combines the *BranchType* control signal to produce a *BranchTaken* signal. *BranchTaken* is then logically ANDed with the control unit's *Branch* signal

to generate *pc_src[0]*, which selects the branch target address as the next PC value rather than the sequential PC+4.

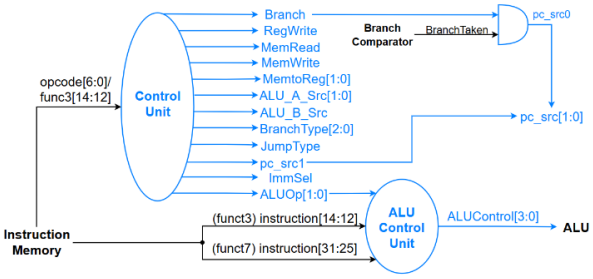


Fig. 2. The processor Control Unit and ALU Control Unit Architecture

2.4 Instruction Classification for Functional Verification

To streamline the verification process, the 31 supported RV32I instructions were classified into six functional categories: R-type, I-type, Load/Store, Branch, Jump, and Upper Immediate, based on instruction format and operational behaviour, as shown in Table 1. Each category was verified using a single representative simulation waveform that exercised multiple instructions in sequence, rather than evaluating each instruction individually. This approach reduced verification complexity while ensuring comprehensive functional coverage of the ALU, register file, memory interface, and PC control logic.

Table 1. Classification of Supported RV32I Instructions for Functional Verification

Instruction Category	Supported Instructions
R-Type	ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
I-Type	ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
Load / Store	LW
Branch	BEQ, BNE, BLT, BGE, BLTU, BGEU
Jump	JAL, JALR
Upper Immediate	LUI, AUIPC

2.5 FPGA Implementation

Following functional verification in ModelSim-Altera, the Verilog HDL design was synthesized and implemented on a Terasic DE1-SoC development board [23] using Intel Quartus Prime Lite, targeting the board's Cyclone V SoC FPGA. A top-level module was developed to interface the processor core with the onboard peripherals, including toggle switches, push buttons, red LEDs, and six seven-segment displays. The processor was driven by the board's 50 MHz system clock, which was divided to a lower frequency to enable step-by-step observation of program execution during hardware validation.

To evaluate the processor under realistic operating conditions, five embedded application demonstrations were implemented: a password lock security system, a parking lot management system, a game score-ranking system, a temperature monitoring with fan speed control, and a menu navigation controller. In each demonstration, an RV32I assembly program is compiled and pre-loaded into the processor's instruction memory as machine code. The processor executes the program in a continuous loop, reading the states of the on board toggle switches and push buttons as inputs through the top-level wrapper module, and writing the corresponding results to the red LEDs and seven-segment displays as outputs. This enables direct observation of the execution program code and system behaviour on real hardware. These demonstrations validate the correct operation of the datapath, control unit, memory interface and selected RV32I instructions in a practical hardware environment.

As the DE1-SoC board's seven-segment displays are active-low, a binary-coded decimal (BCD)-to-seven-segment decoder was incorporated into the top-level module to convert 4-bit data values into the corresponding active-low segment control signals for numerical display.

3. Results and Analysis

3.1 RTL Design in Quartus Prime Lite

The completed processor was first examined at the Register Transfer Level (RTL) in Intel Quartus Prime Lite to verify its structural integrity before functional validation. The top-level RTL, as shown in Fig. 3, illustrates the integration of the datapath and control unit through a grouped control bus that carries the control signals generated from the decoded instruction fields. In turn, feedback signals, such as *opcode*, *funct3*, *funct7*, and *branch_taken*, are routed from the datapath back to the control unit. As shown in Fig. 4, the datapath RTL includes the correct interconnection of the program counter, instruction memory, register file, immediate generator, ALU, data memory, and supporting multiplexers. The RTL control unit shown in Fig. 5 further verifies that the ALU control submodule and the branch decision logic are properly nested within the hierarchical control structure described in Section 2.3. Overall, these RTL representations verify that the intended processor architecture was correctly realized in the hardware description prior to functional simulation.

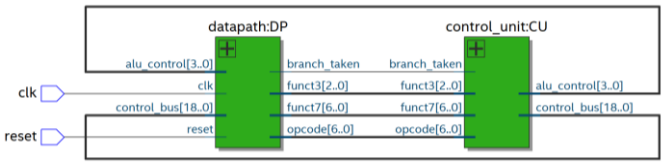


Fig. 3. Top-Level RTL for the 32-bit RISC-V Processor

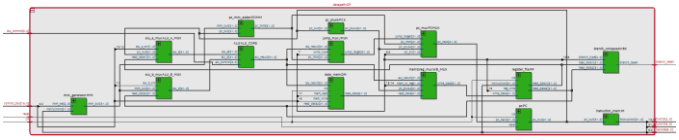


Fig. 4. RTL Representation of the Datapath Module

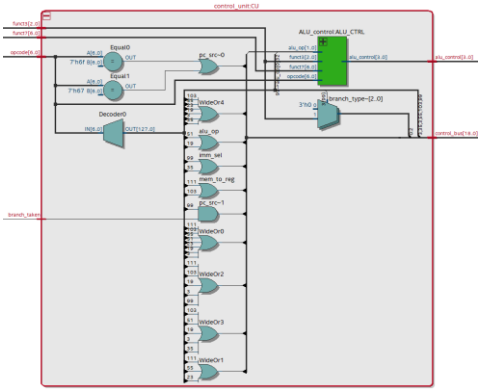


Fig. 5. RTL Representation of the Control Unit

3.2 Functional Simulation Using ModelSim-Altera

The functional correctness of the instruction categories listed in Table 1 was verified using one representative simulation waveform for each category, thereby validating the execution of all 31 selected RV32I instructions.

The simulation waveform for the R-type instruction category, shown in Fig. 6, verifies the sequential execution of all ten R-type instructions (ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND) using register operands of 10 and 5. Each instruction produced the expected result (e.g., ADD = 15, SUB = 5, SLL = 320).

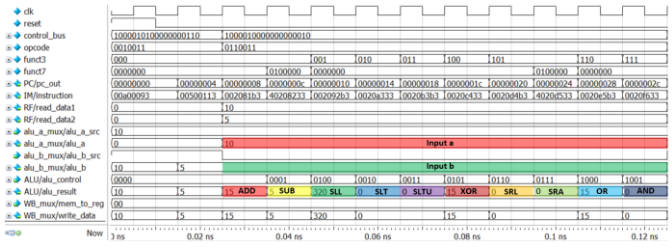


Fig. 6. Simulation Waveform for R-Type Instructions

As shown in Fig. 7, additional test cases using a negative operand further confirmed the correct distinction between signed and unsigned comparisons for the SLT and SLTU instructions, as well as the correct zero-fill and sign-extension behaviour of the SRL and SRA shift operations.

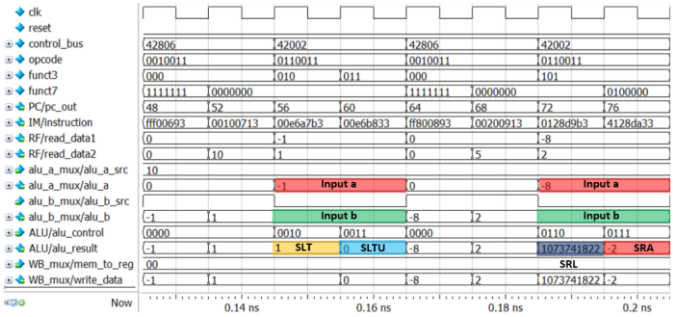


Fig. 7. Simulation Waveform for Signed and Unsigned Cases of R-Type Instructions

Fig. 8 presents the simulation waveform for the I-type instruction category, where all nine instructions (ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI) were verified using an immediate value of 5 and a register value of 10. Each instruction produced the expected arithmetic, logical, or shift result.

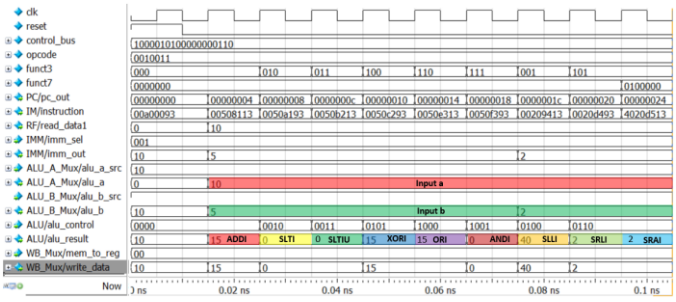


Fig. 8. Simulation Waveform for I-Type Instructions

Fig. 9 demonstrates boundary tests with negative operand (−1) and positive operand (1) further verifying the correct distinction between signed and unsigned comparison and shift operations.

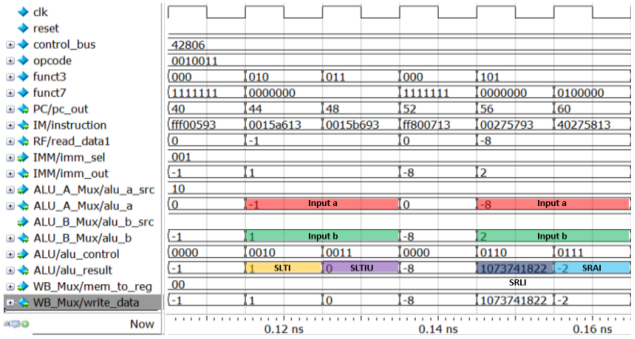


Fig. 9. Simulation Waveform for Signed and Unsigned Cases of I-Type Instructions

Load/Store functionality was verified using SW and LW instructions with both positive and negative address offsets, as shown in Fig. 10. The SW instruction correctly stored data to word-aligned addresses (e.g., sw x2, 0(x1) writing the value 10 to address 16), while the LW instruction successfully retrieved both the original and subsequently updated values, including accesses using negative-offset addressing. These results confirm the correct implementation of effective address computation, memory read/write operations, and data write-back.

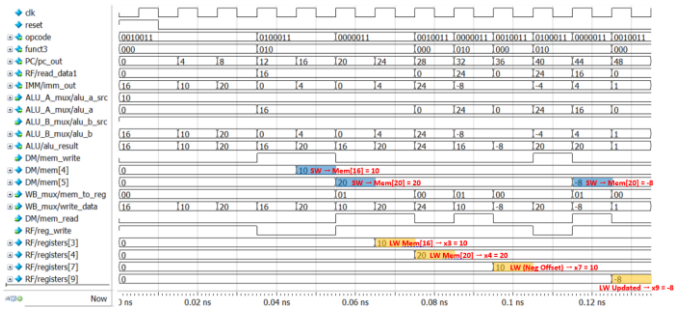


Fig. 10. Simulation Waveform for Load/Store Instructions

Fig. 11 and Fig. 12 present the functional verification for all six branch instructions (BEQ, BNE, BLT, BGE, BLTU, and BGEU) under both taken and not-taken execution conditions. For taken branches, the PC was correctly redirected to the target address (e.g., BEQ: PC 12 → 20), whereas not-taken branches advanced sequentially to the next instruction.

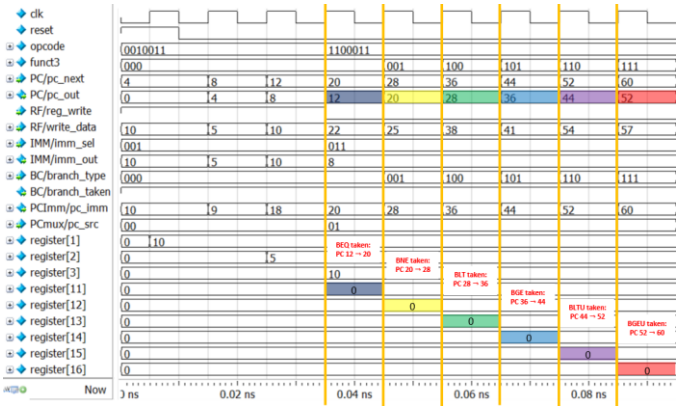


Fig. 11. Simulation Waveform for Branch Taken Cases of Branch Instructions

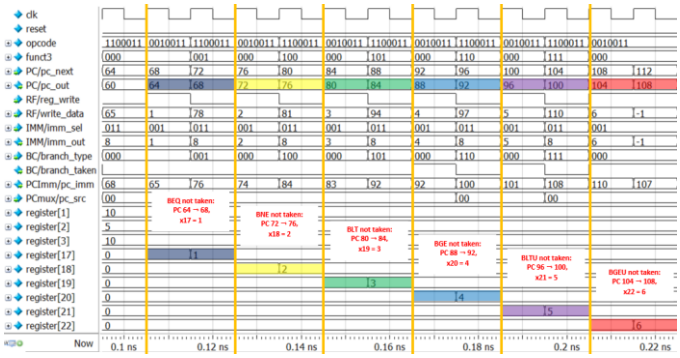


Fig. 12. Simulation Waveform for Branch Not Taken Cases of Branch Instructions

Fig. 13 shows test cases using signed and unsigned operands (−1 and 1), demonstrating BLT was taken while BLTU was not, with the opposite behaviour observed for BGE and BGEU. These results confirm the correct implementation of signed and unsigned comparison within the branch decision unit.

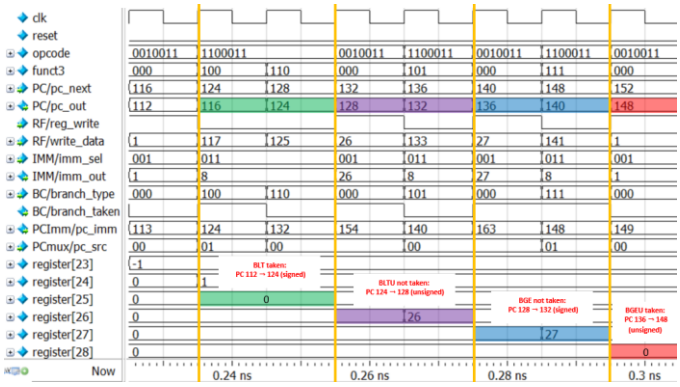


Fig. 13. Simulation Waveform for Signed and Unsigned Cases of Branch Instructions

Fig. 14 presents the functional verification of the JAL and JALR instructions across forward-jump, base-register, and immediate-offset scenarios. The JAL instruction correctly redirected the PC (8 → 16), while storing the return address (PC+4 = 12) in the link register. The JALR instruction correctly computed the target address from a base register, both with and without an immediate offset, and properly enforced address alignment by clearing the least significant bit of an odd target address (PC 68 → 76, with a link register value of 72). These results verify the successful implementation of jump target generation, return address storage, and address alignment logic.

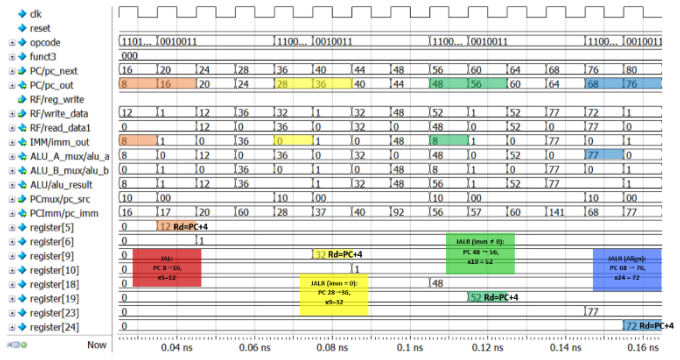


Fig. 14. Simulation Waveform for Jump Instructions

Fig. 15 presents the functional verification of the upper immediate instructions LUI and AUIPC. The LUI instruction correctly loaded the 20-bit immediate into the upper 20-bits of the destination register while clearing the lower 12 bits (e.g., lui x3, 0x12345 → 0x12345000). The AUIPC instruction correctly added the upper immediate value, shifted value to the current PC (e.g., auipc x5, 0x00001 at PC = 16 → 4112). These results confirm the correct implementation of upper-immediate generation and PC-relative address computation.

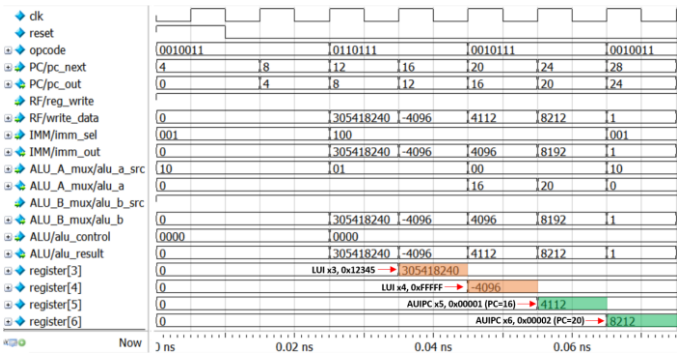


Fig. 15. Simulation Waveform for Upper Immediate Instructions

Overall, the simulation results across all six instruction categories verify the correct operation of the datapath and control unit in executing all 31 implemented RV32I instructions. These results further demonstrate the correct implementation of signed and unsigned arithmetic and comparison, memory address computation and access, and control-flow operations, including branch and jump redirection.

3.3 FPGA Prototyping Validation

To validate the processor under realistic operating conditions, five embedded application demonstrations were implemented on the Terasic DE1-SoC board, each exercising a different subset of instructions through onboard switches, push buttons, LEDs, and seven-segment displays.

(a) Password Lock Security System

The password lock security system operates in three states: Locked, Cool-down, and Unlocked. As shown in Fig. 16, the system is initialized in the Locked state, where password authentication is performed.

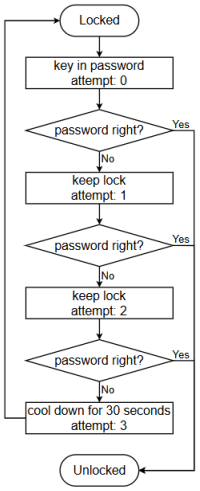


Fig. 16. Flowchart of FPGA Password Lock Security System

A correct password entry immediately transitions the system to the Unlocked state. Conversely, each incorrect password increments the failed-attempt counter, with a maximum of three consecutive attempts. Upon reaching the maximum number of failed attempts, the system enters the Cool-down state, during which password input is disabled for 30 seconds. After the cool-down state expires, the failed-attempt counter is reset, and the system automatically returns to the Locked state to resume password authentication. The corresponding peripheral layout implemented on the DE1-SoC board is shown in Fig. 17.

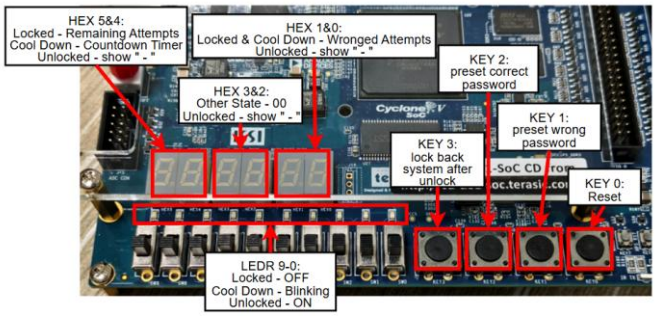


Fig. 17. FPGA Peripheral Layout for Password Lock Security System

Table 2 summarises the expected peripheral responses for each system state during hardware validation. Push buttons KEY1 and KEY2 represent incorrect and correct password entries, respectively. The HEX displays show the remaining password attempts and the cumulative number of failed attempts, while the red LEDs visually indicate the system's current operating state.

Table 2. Peripheral Responses for the Password Lock Security System at Each State

Peripheral	Locked State	Cool Down State	Unlocked State
KEY0	Reset the system to the initial state		
KEY1	Incorrect password entry	No effect	No effect
KEY2	Correct password entry	No effect	No effect
KEY3	No effect	No effect	Reset to Locked state
HEX5 & HEX4	Remaining attempts (e.g. 03, 02, 01)	Countdown timer (starting from 30)	–
HEX3 & HEX2	Displays 00	Displays 00	–
HEX1 & HEX0	Cumulative failed attempt (e.g. 01)	Cumulative failed attempt (e.g. 03)	–
LEDR9 – LEDR0	All OFF	Continuously blinking	All ON

Table 3 presents the hardware test results, confirming the correct operation of password verification, failed-attempt monitoring, and temporary lockout functionality under all tested conditions. The RV32I instructions exercised by this application are summarised in Table 4. The demonstration covers 9 of the 31 implemented instructions, namely ADDI, ANDI, XOR, LW, SW, BEQ, BNE, BGE, and JAL, for validating arithmetic, logical, memory access, and control-flow operations within a practical embedded application.

Table 3. Results of the FPGA Password Lock Security System

Conditions	System Response
1st Incorrect Authentication	 First incorrect password entered, leaving two remaining attempts and recording one failed attempt.
2nd Incorrect Authentication	 Second incorrect password entered, leaving one remaining attempt and two failed attempts recorded.
3rd Incorrect Authentication	 Third incorrect password entered, no attempt remaining and cumulative three failed attempts. The system immediately enters the cool-down state.
Cool Down	 HEX5 and HEX4 act as a countdown timer starting from 30, while HEX3-HEX0 display 0the failed-attempt count 0003. All red LEDs blink continuously to indicate the temporary lockout condition.
Correct Authentication - Unlocked	 A correct password entry transitions the processor to the unlocked state. All red LEDs are turned ON, while the HEX displays show dash symbols to indicate successful access.

Table 4. Instructions Covered in the FPGA Password Lock Security System

Instruction Type	Instruction Used	Purpose in Demo
I-type	ADDI	Set password value, counters, LED pattern, and countdown value.
	ANDI	Check KEY input bits.
R-type	XOR	Compare the entered password with the correct password.
Load	LW	Read KEY input and 1-second tick event.
Store	SW	Write to LEDR, HEX display, and blink control.
Branch	BEQ	Check password match and wait-loop condition.
	BNE	Detect key presses and cool down countdown loop.
	BGE	Check whether wrong attempts reach maximum limit.
Jump	JAL	Jump back to locked state, wait state, or initialization state.

(b) Parking Lot Management System

The parking lot management system models vehicle entry and exit with capacity enforcement across three states: Normal, Full, and Warning. As shown in Fig. 18, the system initializes in the Normal state, where car entry and exit are freely permitted while the remaining capacity is tracked. Once the lot reaches maximum capacity, the system transitions to the Full state and blocks further entries. An attempted entry when the lot is full, or an attempted exit when the lot is already empty,

triggers the corresponding Invalid Enter or Invalid Exit Warning state, during which all LEDs blink as an alert.

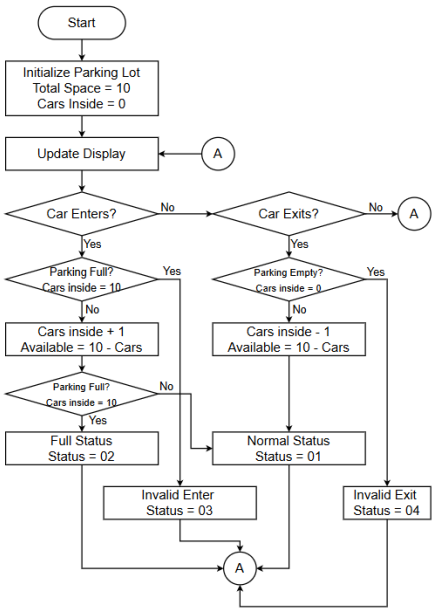


Fig. 18. Flowchart of FPGA Parking Lot Management System

The corresponding peripheral layout implemented on the DE1-SoC board is shown in Fig. 19.

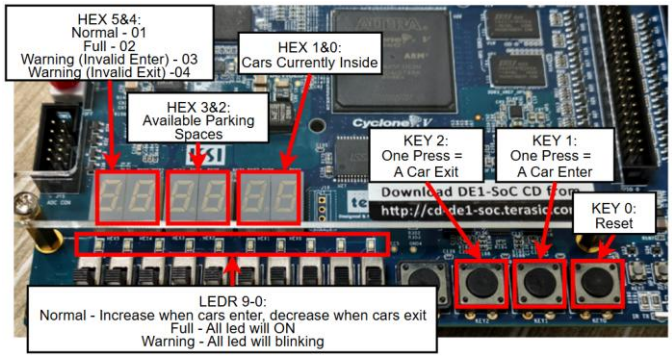


Fig. 19. FPGA Peripheral Layout for Parking Lot Management System Demonstration

Table 5 summarises the peripheral responses for each system state during hardware validation. Push buttons KEY1 and KEY2 simulate car entry and exit events, respectively, while KEY0 resets the system to its initial state. The HEX displays show the current status code, the number of available parking spaces, and the total number of cars currently inside, while the red LEDs indicate the current occupancy level and blink continuously during warning states.

Table 6 presents the test results on the FPGA development board, confirming that the processor correctly incremented and decremented occupancy counts, blocked invalid entries once the lot was full, blocked invalid exits when the lot was already empty, and updated all HEX display and LED outputs accordingly across all four tested conditions.

Table 5. Peripheral Functions of the Parking Lot Management System at Each State

Peripheral	State			
	Normal	**Full**	**Warning: Invalid Entry**	**Warning: Invalid Exit**
KEY0	Reset the system to the initial state.			
KEY1	Simulates one car entry	Triggers an invalid enter warning	No effect	No effect
KEY2	Simulates one car exit	Simulates one car exit; returns to normal	No effect	No effect
HEX5 & HEX4	Code 01: Normal	Code 02: Parking full	Code 03: Invalid entry	Code 04: Invalid exit
HEX3 & HEX2	Available parking	Code 00: No spaces	Currently available spaces	Currently available spaces
HEX1 & HEX0	Number of cars inside	Maximum car count	Current car count	00: Empty car count
LEDR9 – LEDR0	Increases on enter; decreases on exit	All LEDs ON	All LEDs blink	All LEDs blink

Table 6. Results of the FPGA Parking Lot Management System

Conditions		System Response
Normal		 Status 01 indicates a normal condition. HEX3-HEX2 display 05 available spaces, while HEX1-HEX0 display 05 cars inside. LEDs indicate the current number of cars
Full		 Status 02 shows the parking lot is full. HEX3-HEX2 display no available spaces, and HEX1-HEX0 display 10 cars inside. All LEDs are ON.
Warning	Invalid Entry	 Status 03 indicates a car attempting to enter the parking lot when it is full. Available spaces remain is zero, cars inside remain 10, and LEDs blink as a warning.
	Invalid Exit	 Status 04 shows a car attempting to exit when the parking lot is empty. There are 10 available spaces, and no cars are inside the parking lot. The LEDs blink as a warning.

The RV32I instructions used by this application are listed in Table 7. The demonstration covers 13 of the 31 implemented instructions, including signed comparison (SLT, SLTI), conditional branching (BEQ, BNE, BLT, BGE), arithmetic operations (ADD, SUB, ADDI), and memory access (LW, SW), validating both arithmetic correctness and boundary-condition enforcement within a practical capacity-management scenario.

Table 7. Instructions Covered in FPGA Parking Lot Management System

Instruction Type	Instruction Used	Purpose in Demo
I-type	ADDI	Set total parking spaces, counters, status codes, and display values.
	ANDI	Check KEY input bits for car entry and car exit events.
R-type	ADD	Update or transfer calculated parking values.
	SUB	Calculate the available spaces from the total spaces and the number of cars.
	SLT	Compare the car count with the parking limit.
I-type	SLTI	Check the warning threshold or the parking count condition.
Load	LW	Read the KEY input event from memory-mapped I/O.
Store	SW	Write parking status, car count, available spaces, LED pattern, and blink control.
Branch	BEQ	Check full, empty, or normal condition
	BNE	Detect KEY press events
	BLT	Check whether the car count is below the parking capacity.
	BGE	Check whether the car count reaches or exceeds the parking capacity.
Jump	JAL	Jump between wait, enter, exit, warning, and display states.

(c) Game Score Ranking System

The game score ranking system compares two player scores in real time across three states: Player A Leading, Player B Leading, and Tie. As shown in Fig. 20, the system continuously evaluates the two player scores and transitions between states based on signed and unsigned comparisons. When both scores are equal, the system enters the Tie state. As either player's score changes, the system updates the displayed state accordingly. The corresponding peripheral layout implemented on the DE1-SoC board is shown in Fig. 21.

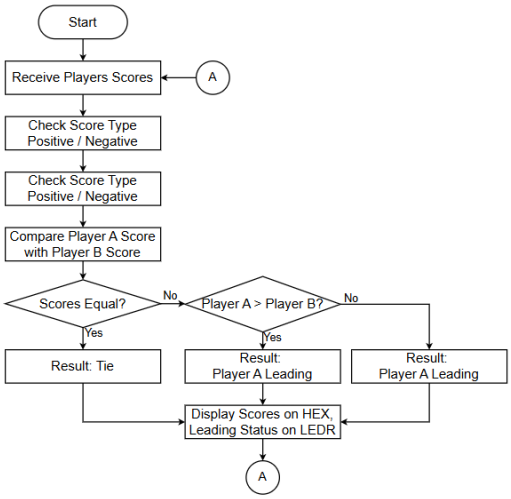


Fig. 20. Flowchart of Game Score Ranking System

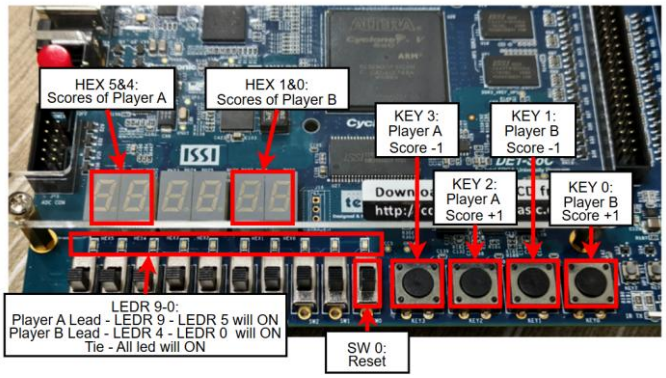


Fig. 21. FPGA Peripheral Layout for Game Score Ranking System

Table 8 summarises the expected peripheral responses for each system state during hardware validation. Push buttons KEY0 to KEY3 control score increments and decrements for both players, while KEY0 also resets the system to its initial state. The HEX displays show the current scores of Player A and Player B, and the red LED pattern indicates the leading player or a tie; all LEDs turn on when the scores are equal.

Table 8. Peripheral Functions of the Game Score Ranking System in Each State

Peripheral	Player A Leading State	Player B Leading State	Tie State
KEY0	Increases Player B score by 1		
KEY1	Decreases Player B score by 1		
KEY2	Increases Player A score by 1		
KEY3	Decreases Player A score by 1		
HEX5 & HEX4	Displays Player A score from -9 to 99		
HEX1 & HEX0	Displays Player B score from -9 to 99		
LEDR9 – LEDR0	LEDR9 to LEDR5 ON, indicating Player A leads	LEDR4 to LEDR0 ON, indicating Player B leads	All LEDs ON, indicating both players have the same score

Table 9 presents the hardware test results across six test cases, including equal-score, single-leading, and negative-score scenarios. The results confirm the correct identification of the leading player or tie condition in every case, verifying the correct execution of both signed and unsigned branch-comparison instructions.

Table 9. Results of the FPGA Game Score Ranking System

Conditions	System Response
Player A Leading	 Player A score is higher than Player B. HEX5-HEX4 show Player A score, HEX1-HEX0 show Player B score, and LEDR9-LEDR5 turn ON.
Player B Leading	 Player B score is higher than Player A. HEX displays both scores, and LEDR4-LEDR0 turn ON to indicate Player B is leading.

Tie	 Both players have the same score. The same value is shown for Player A and Player B, and all LEDs turn ON to indicate a tie.
Player B Leading with Negative Score	 Player A has a negative score while Player B has a positive score. Player B is leading, so LEDR4-LEDR0 turn ON
Player B Leading with both Negative Score	 Both players have negative scores, but Player B has the higher value. Therefore, LEDR4-LEDR0 turn on to indicate that Player B is leading.
Player A Leading with Zero Score	 Player A has a zero score while Player B has a negative score. Player A is leading, so LEDR9-LEDR5 turn ON.

The RV32I instructions exercised by this application are summarised in Table 10. This demonstration was the most instruction-intensive of the five, covering 17 of the 31 implemented instructions, including signed and unsigned comparison variants (SLT, SLTU, SLTI, SLTIU), shift instructions (SLLI), unsigned branch comparison (BGEU), and arithmetic and memory-access instructions, validating a wide range of the implemented instruction set within a real-time comparison application.

Table 10. Instructions of the FPGA Gaming Score Ranking System

Instruction Type	Instruction Used	Purpose in Demo
I-type	ADDI	Set score limits, initial scores, LED patterns, and update score values.
	ANDI	Check KEY input bits for Player A and Player B score control.
	SLLI	Generate an LED pattern for Player A leading status.
	SLTI	Check the signed score condition during the comparison setup.
	SLTIU	Check the unsigned score condition during comparison setup.
R-type	ADD	Generate LED patterns and transfer selected leading status.
	SUB	Calculate the score difference between Player A and Player B.
	SLT	Perform a signed comparison between score limits.
	SLTU	Perform an unsigned comparison between score values.
Load	LW	Read KEY input event from memory-mapped I/O.
Store	SW	Write Player A score, Player B score, and LED leading status.
Branch	BEQ	Check whether both players have the same score.
	BNE	Detect score control button press.
	BLT	Check whether Player A score is lower than Player B score.
	BGE	Prevent scores from exceeding the allowed range of -9 to 99.
	BGEU	Perform an unsigned branch comparison during. comparison setup
Jump	JAL	Jump between display, input waiting, score update, and LED update states

(d) Temperature Monitor with Fan Speed Control

The temperature monitor with fan speed control classifies temperature readings into five discrete states: Extra Low, Low, Medium, High, and Extra High. As shown in Fig. 22, the system initializes in the Extra Low state and transitions between states based on the current temperature reading. Each state maps to a specific fan speed level. In the Extra High state, all red LEDs blink continuously to indicate a high-temperature alarm. The corresponding peripheral layout implemented on the DE1-SoC board is shown in Fig. 23.

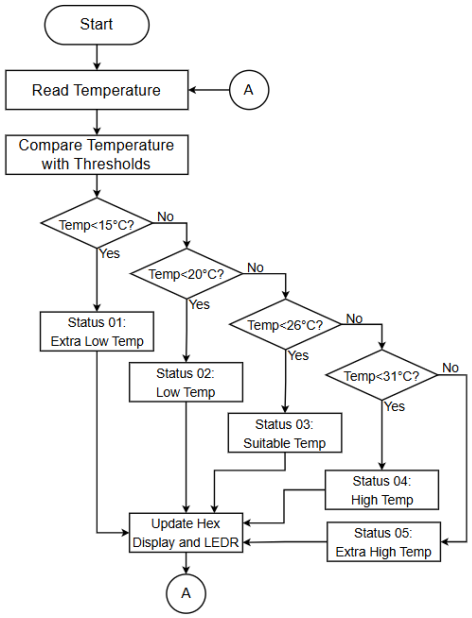


Fig. 22. Flowchart of Temperature Monitor with Fan Speed Control System

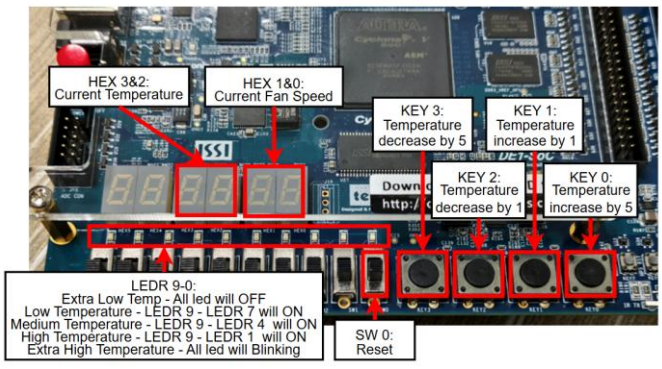


Fig. 23. FPGA Peripheral Layout for Temperature Monitor with Fan Speed Control System

Table 11 summarises the expected peripheral responses for each system state during hardware validation. Push buttons KEY0 to KEY3 adjust the simulated temperature in increments of 1°C or 5°C, either upward or downward, while KEY0 also resets the system to its initial state. The HEX displays show the

current temperature value and the corresponding fan speed level, while the red LEDs indicate the active temperature band and blink continuously at the highest severity state.

Table 11. Peripheral Functions of the Temperature Monitor with Fan Speed Control System State

Peripheral	Extra Low Temp State	Low Temp State	Medium Temp State	High Temp State	Extra High Temp State
SW0	Reset the system to the initial state				
KEY0	Increases temperature by 5°C				
KEY1	Increases temperature by 1°C				
KEY2	Decreases temperature by 1°C				
KEY2	Decreases temperature by 1°C				
HEX3 & HEX2	Displays current temperature				
HEX1 & HEX0	Display fan speed level 00	Display fan speed level 01	Display fan speed level 02	Display fan speed level 03	Display fan speed level 03
LEDR9 – LEDR0	All LEDs OFF	LEDR9 to LEDR7 ON	LEDR9 to LEDR4 ON	LEDR9 to LEDR1 ON	All LEDs blink continuously

Table 12 presents the hardware test results, confirming correct temperature classification, fan speed mapping, and LED behaviour across all five temperature states, including the blinking-LED alarm condition at the Extra High state.

Table 12. Results of FPGA Temperature Monitor with Fan Speed Control System

Conditions	System Response
Extra Low Temperature State	 The temperature is below 15°C, so the fan is stopped at level 00 and all LEDs are OFF.
Low Temperature State	 The temperature is between 15°C and 19°C, so the fan runs at level 01, and LEDR9-LEDR7 turn ON.
Medium Temperature State	 The temperature is between 20°C and 25°C, so the fan runs at level 02, and LEDR9-LEDR4 turn ON.
High Temperature State	 The temperature is between 26°C and 30°C, so the fan runs at level 03, and LEDR9-LEDR1 turn ON.
Extra High Temperature State	 The temperature is above 30°C, so the fan remains at level 03, and all LEDs blink as a warning.

The RV32I instructions exercised by this application are summarised in Table 13. This demonstration achieved the highest instruction coverage among the five applications, covering 19 of the 31 implemented instructions, including arithmetic shift instructions (SLLI, SRLI, SRAI, SRL, SRA), bitwise operations (ORI, XORI, AND, OR), and unsigned branching (BGEU), validating the full range of shift and bitwise logic operations within a practical sensor-based control application.

Table 13. Instructions Covered in the FPGA Temperature Monitor with Fan Speed Control System

Instruction Type	Instruction Used	Purpose in Demo
I-type	ADDI	Set temperature thresholds, initial temperature, fan levels, and temperature update values.
	ANDI	Check KEY input bits for temperature increase and decrease control.
	ORI	Generate LED patterns for different fan speed levels.
	XORI	Generate a blink enable signal for an extra high temperature warning.
	SLLI	Shift the temperature value to the HEX3-HEX2 display position.
	SRLI	Perform an immediate logical right shift check in the evaluation engine.
	SRAI	Perform an immediate arithmetic right shift check in the evaluation engine.
R-type	ADD	Combine temperature and fan level values for the seven-segment display output.
	AND	Mask the LED pattern to keep the output within the LEDR9-LEDR0 range.
	OR	Generate a full LED pattern for the extra high temperature state.
	SRL	Perform a register-based logical right shift check in the evaluation engine.
	SRA	Perform register-based arithmetic right shift check in the evaluation engine.
Load	LW	Read KEY input event from memory-mapped I/O.
Store	SW	Write the temperature value, fan level, LED pattern, and blink control output.
Branch	BNE	Detect temperature control button press.
	BLT	Compare the current temperature with the threshold values.
	BGE	Prevent the temperature from exceeding the allowed range.
	BGEU	Perform unsigned branch comparison during setup.
Jump	JAL	Jump between evaluation, display, wait, and temperature update states.

(e) Menu Controller System

The menu controller implements a four-page navigation system comprising four display pages: Home, Temperature, Fan, and Alarm. As shown in Fig. 24, the system initializes on the Home page, which displays a summary of all sensor readings. Navigation between pages is performed using dedicated push buttons; each page displays its corresponding sensor data on the HEX displays and indicates the active page via the red LED pattern. Indirect jump instructions (JALR) are used to implement dynamic page switching, where the target page address is computed at runtime using AUIPC-based base address calculation.

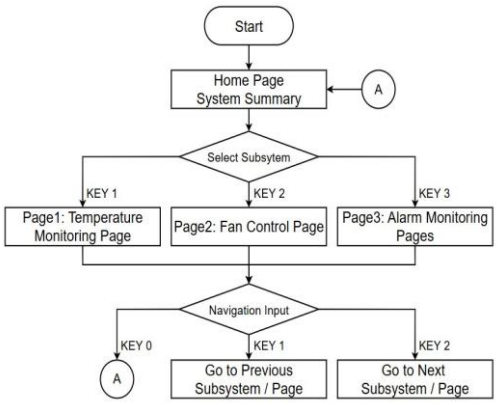


Fig. 24. Flowchart of FPGA Menu Controller

The corresponding peripheral layout implemented on the DE1-SoC board is shown in Fig. 25.

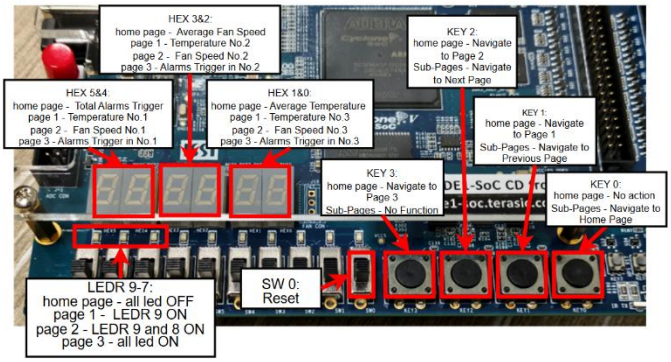


Fig. 25. FPGA Peripheral Layout for Menu Controller

Table 14. Peripheral Functions of the Menu Controller System at Each System Page

Peripheral	Home Page	Page 1: Temperature Page	Page 2: Fan Speed Page	Page 3: Alarm Page
SW0	Reset the system to the initial home page			
KEY0	No action	Navigate to Home Page	Navigate to Home Page	Navigate to Home Page
KEY1	Navigate to Page 1	Navigate to the previous page, Page 3	Navigate to the previous page, Page 1	Navigate to the previous page, Page 2
KEY2	Navigate to Page 2	Navigate to the next page, Page 2	Navigate to the next page, Page 3	Navigate to next page, Page 1
KEY3	Navigate to Page 3	No action	No action	No action
HEX5 & HEX4	Displays total alarm triggers	Displays temperature sensor No.1	Displays fan speed No.1	Displays alarm trigger No.1
HEX3 & HEX2	Displays average fan speed	Displays temperature sensor No.2	Displays fan speed No.2	Displays alarm trigger No.2
HEX1 & HEX0	Displays average temperature	Displays temperature sensor No.3	Displays fan speed No.3	Displays alarm trigger No.3
LEDR9 – LEDR7	All LEDs OFF	LEDR9 ON	LEDR9 and LEDR8 ON	LEDR9 to LEDR7 ON

Table 14 summarises the expected peripheral responses for each page during hardware validation. Push buttons KEY1 to KEY3 to navigate between the four pages in both forward and backward directions, while KEY0 resets the system to the Home page. The HEX displays show either a full sensor summary on the Home page or individual sensor readings on the Temperature, Fan, and Alarm pages, while the red LEDs indicate the currently active page using a distinct LED pattern for each page.

Table 15 presents the hardware test results, confirming correct page navigation and accurate display of page-specific sensor data across all four pages. The results verify that each push button correctly triggers the corresponding page transition, and that the HEX displays and LEDs update as expected on every tested navigation path.

Table 15. Results of FPGA for Menu Controller

Conditions	Results
Home Page	 The home page shows the farm summary. HEX5-HEX4 display total alarms, HEX3-HEX2 display average fan speed, and HEX1-HEX0 display average temperature.
Page 1: Temperature Page	 This page shows the temperature readings from three sensors. HEX5-HEX4, HEX3-HEX2, and HEX1-HEX0 display sensor 1, sensor 2, and sensor 3 respectively.
Page 2: Fans Speed Page	 This page shows the fan speed levels for three fans. HEX5-HEX4, HEX3-HEX2, and HEX1-HEX0 display fan 1, fan 2, and fan 3 speed respectively.
Page 3: Alarms Page	 This page shows the alarm trigger count for three sensor-fan pairs. HEX5-HEX4, HEX3-HEX2, and HEX1-HEX0 display alarm for sets 1, 2, and 3 respectively.

The RV32I instructions exercised by this application are summarised in Table 16. The demonstration covers 12 of the 31 implemented instructions, with particular emphasis on indirect jump instructions (JALR combined with AUIPC-based base address computation) for dynamic page switching, LUI for generating upper immediate base addresses, and SRLI for shifting display data into the correct HEX display positions, validating the upper-immediate and jump instruction categories within a practical multi-page navigation application.

Table 16. Instructions Covered in the FPGA Menu Controller

Instruction Type	Instruction Used	Purpose in Demo
I-type	ADDI	Set display values, page data, LED patterns, and navigation target offsets
	ANDI	Check KEY input bits for page navigation
	SLLI	Shift display values into the correct HEX display position
	SRLI	Generate LED page indicator pattern from upper immediate value
R-type	ADD	Combine display values or transfer page indicator values

Load	LW	Read KEY input event from memory-mapped I/O
Store	SW	Write page data, LED page indicator, and display output
Branch	BNE	Detect page navigation button press
Jump	JAL	Jump to page display or wait states
	JALR	Perform indirect jump to selected menu page
Upper Immediate	LUI	Generate upper immediate value for LED page indicator pattern
	AUIPC	Generate PC-relative address for indirect page navigation

Combining all five demonstrations, 29 of the 31 implemented RV32I instructions were exercised on real hardware; the remaining two (SLL and BLTU) were not triggered by these particular application programs but had already been verified independently during ModelSim simulation (Section 3.2).

3.4 Resource Utilization and Performance

Table 17 summarizes the post-fitting implementation results of the designed RISC-V processor. The processor occupies 17% of the available logic elements (5,528 of 32,070), utilizes 15% of the I/O pins (67 of 457), uses 9,594 dedicated logic registers on the Cyclone V SoC FPGA, achieves a maximum operating frequency of 53.31 MHz, and consumes 422.07 mW of total power. The results indicate that the single-cycle RV32I core occupies a relatively small footprint on the DE1-SoC, leaving substantial headroom for future extensions such as additional instructions or pipelining.

Table 17. Summary of the FPGA Implementation

Resource Utilization	Logic Utilization (ALMs)	5,528 / 32,070 (17%)
	I/O Pins	67 / 457 (15%)
	Dedicated Logic Registers	9,594
Maximum Operating Frequency	53.31 MHz	
Total Power Consumption	422.07 mW	

4. CONCLUSION

This paper presented the design, simulation, and FPGA implementation of a 32-bit single-cycle RISC-V processor based on the RV32I instruction set. All three objectives of this work were achieved. First, a complete single-cycle processor architecture was designed in Verilog HDL, comprising a fully integrated datapath and a hierarchical control unit capable of decoding and executing instructions across six functional categories. Second, the functional correctness of the design was verified through ModelSim-Altera simulation, confirming correct execution of all 31 selected RV32I instructions, including proper handling of signed and unsigned arithmetic, memory addressing, and control-flow redirection. Third, the verified design was successfully synthesized and implemented on a Terasic DE1-SoC FPGA board, where it was further validated through five embedded application demonstrations – a password lock security system, a parking lot management system, a gaming score ranking system, a temperature monitor with fan speed control, and a menu navigation controller – collectively exercising 29 of the 31 implemented instructions under realistic operating conditions.

The current design implements 31 of the 40 base RV32I instructions, excluding byte/half-word loads and stores (LB, LH, LBU, LHU, SB, SH) and system-level instructions (FENCE, ECALL, EBREAK). The single-cycle architecture is simple and easy to verify, executing one instruction per clock cycle regardless of instruction complexity, but it limits overall throughput compared to pipelined designs.

Future work recommendations include extending the instruction set design for the remaining RV32I instructions and adding standard extensions such as RV32M (multiplication/division) or RV32F (floating-point), broadening the processor's applicability. Besides, the architecture could be redesigned as a multi-cycle or pipelined implementation to improve instruction throughput and operating frequency, while introducing hazard detection and forwarding logic. This includes additional system-level features such as interrupt handling, exception support, and memory-mapped peripheral controllers, which could be incorporated to move the design closer to a practically deployable embedded processor, while preserving its current value as an accessible teaching platform for RISC-V and computer architecture education.

ACKNOWLEDGEMENT

This research was supported by the Universiti Tun Hussein Onn Malaysia (UTHM) through TIER 1 (vot Q868). The authors would like to thank the Faculty of Electrical and Electronic Engineering, Universiti Tun Hussein Onn Malaysia for its support.

REFERENCES

[1] S. Harris and D. Harris, "Digital Design and Computer Architecture, RISC-V Edition, 1st ed.", *Morgan Kaufmann*, Cambridge, MA, USA, 2021.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA", *EECS Department*, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, 2014.

[3] K. D. Nguyen, T. K. Dang, B. Kieu-Do-Nguyen, C. K. Pham, and T. T. Hoang, "RISC-V-Based System-on-Chips for IoT Applications", *IEEE Hot Chips 36 Symposium (HCS)*, Aug. 2024. doi: https://doi.org/10.1109/HCS61935.2024.10665181.

[4] A. Dörflinger et al., "A comparative survey of open-source application-class RISC-V processor implementations", *Proceedings of the 18th ACM International Conference on Computing Frontiers (CF '21)*, New York, USA, pp. 12–20, 2021. doi: https://doi.org/10.1145/3457388.3458657.

[5] M. S. Poojary and R. H. K, "Design of RV32I Instruction Set Architecture", *3rd International Conference on Smart Systems for Applications in Electrical Sciences (ICSSES)*, pp. 1–6, 2025. doi: https://doi.org/10.1109/ICSSES64899.2025.11009768.

[6] U. Farooq, Z. Marrakchi, and H. Mehrez, "FPGA Architectures: An Overview", *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*, New York, NY: Springer New York, pp. 7–48, 2012. doi: https://doi.org/10.1007/978-1-4614-3594-5_2.

[7] K. Kırali and C. B. Fidan, "Implementation of FPGA based 32-bit RISC-V processor", *Engineering Science and Technology, an International*

Journal, vol. 70, p. 102139, 2025. doi: https://doi.org/10.1016/j.jestch.2025.102139.

[8] M. S. Paliwal and S. Tripathy, "Difference Between Single-Cycle and Multi Cycle Processor", *International Journal of Electrical Engineering and Technology (IJEET)*, vol. 11, no. 10, pp. 240–251, 2020.

[9] J. B. Lazaro et al., "Implementation of a Single Cycle Datapath for the Design of a Low-process Control in XC3S500E", *16th International Conference on Computer and Automation Engineering (ICCAE)*, pp. 730–734, 2024. doi: https://doi.org/10.1109/ICCAE59995.2024.10569250.

[10] J. Y. Lai, C. A. Chen, S. L. Chen, and C. Y. Su, "Implement 32-bit RISC-V Architecture Processor using Verilog HDL", *International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pp. 1–2, 2021. doi: https://doi.org/10.1109/ISPACS51563.2021.9651130.

[11] T. Meeradevi, K. Mohanraj, B. M. Mourissh, V. S. Sivaa, R. Samikannu, and S. Sasikala, "Design of 32 Bit RISC V Processor", *15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–7, 2024. doi: https://doi.org/10.1109/ICCCNT61001.2024.10726132.

[12] E. Jaya, B. Maneesha, G. Sriram, G. Sai, and M. Siddhu, "Design of RISCV processor using Verilog", *i-Manager's Journal on Digital Signal Processing*, vol. 12, no. 1, pp. 15–20, 2024. doi: https://doi.org/10.26634/jdp.12.1.20567.

[13] H. H. Lim and K. L. Chua, "Design of 32-Bit RISC-V Architecture Using Verilog Hardware Description Language", *Evolution in Electrical and Electronic Engineering*, vol. 6, no. 2, pp. 465–476, 2025.

[14] J. K. Goh and C. Uttraphan, "An Enhanced UTHM RISC-V Processor Core Architecture Implemented on FPGA", *Evolution in Electrical and Electronic Engineering*, vol. 5, no. 2, pp. 32–41, 2024.

[15] L. Poli, S. Saha, X. Zhai, and K. D. Mcdonald-Maier, "Design and Implementation of a RISC V Processor on FPGA", *17th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 161–166, 2021. doi: https://doi.org/10.1109/MSN53354.2021.00037.

[16] S. Shukla, P. K. Jha, and K. C. Ray, "An energy-efficient single-cycle RV32I microprocessor for edge computing applications", *Integration*, vol. 88, pp. 233–240, 2023. doi: https://doi.org/10.1016/j.vlsi.2022.09.005.

[17] A. E. Phangestu, I. T. Mujiono, M. I. Kom, and S. A. Zaini, "Five-Stage Pipelined 32-Bit RISC-V Base Integer Instruction Set Architecture Soft Microprocessor Core in VHDL", *International Seminar on Intelligent Technology and Its Applications (ISITIA)*, pp. 304–309, 2022. doi: https://doi.org/10.1109/ISITIA56226.2022.9855292.

[18] K. C. Lau, M. Ab. Rahman, and C. U. Eh Kan, "Design and Implementation of Pipelined RISC Processor for Educational Purpose", *Evolution in Electrical and Electronic Engineering*, vol. 5, no. 2, pp. 369–377, 2024.

[19] F. Hussain and S. Sarkar, "Design and FPGA Implementation of Five Stage Pipelined RISC-V Processor", *IEEE 9th International Conference for Convergence in Technology (I2CT)*, pp. 1–6, 2024. doi: https://doi.org/10.1109/I2CT61223.2024.10544184.

[20] B. V. V. Satyanarayana, N. Madhu, and K. Srinivas, "Design and Implementation of High-Performance RISC-V Processor with Five Stage Pipeline on FPGA", *3rd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, pp. 68–75, 2025. doi: https://doi.org/10.1109/ICSSAS66150.2025.11081376.

[21] L. Weingarten, K. Datta, A. Kole, and R. Drechsler, "Complete and Efficient Verification for a RISC-V Processor Using Formal Verification", *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024. doi: https://doi.org/10.23919/DATE58400.2024.10546693.

[22] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study", *Forum for Specification and Design Languages (FDL)*, pp. 1–7, 2020. doi: https://doi.org/10.1109/FDL50818.2020.9232941.

[23] DE1-SoC User Manual. *Terasic Technologies Inc*, 2013.