REPORT ON

RISC-V PROCESSOR ON FPGA

SUBMITTED BY

Tejhuram Ravichandran(231EC263)

R Rishadd (231EC241)

Under The Guidance of

Dr. Sushil Kumar Pandey

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY, KARNATAKA(NITK)

APRIL 2025

CONTENTS

ABSTRACT

This project presents the design and implementation of a single-cycle RISC-V microprocessor on the PYNQ-Z2 development board using Verilog HDL. The processor adheres to the RV32I base integer instruction set and is structured to execute one instruction per clock cycle, enabling straightforward control flow and simplified datapath integration. The objective was to create a functional, synthesizable processor core from scratch, providing a foundation for architectural exploration and embedded system development.

Key modules include the program counter, instruction memory, register file, ALU, immediate generator, control unit, and data memory, all interconnected in a single-cycle datapath. The processor supports essential instructions such as arithmetic, logical, load/store, and branch operations, verified through extensive simulation and real hardware testing. Integration with the Vivado design suite facilitated synthesis, timing analysis, and deployment to the FPGA board for validation.

The implementation demonstrates correct instruction execution, register updates, and memory interactions, confirming architectural compliance and functional reliability. This project highlights the practical application of computer architecture concepts in FPGA-based digital design and provides a modular baseline for future enhancements such as pipelining, interrupt handling, and peripheral interfacing.

1.INTRODUCTION

1.1 INTRODUCTION

The rapid growth of embedded systems, IoT devices, and custom accelerators has driven the need for efficient, customizable microprocessor architectures. Among the open-source solutions, the **RISC-V** instruction set architecture (ISA) has emerged as a popular choice due to its simplicity, modularity, and extensibility. Designed with academic, research, and industrial use in mind, RISC-V provides an excellent foundation for understanding processor architecture and digital system design.

This thesis focuses on the design and hardware implementation of a **single-cycle RISC-V microprocessor** using **Verilog HDL** and its deployment on the **PYNQ-Z2 FPGA development board**. The single-cycle architecture was chosen for its pedagogical clarity and ease of implementation, where each instruction is fetched, decoded, and executed within a single clock cycle. Although not the most performance-efficient in commercial settings, this approach is ideal for educational purposes, architectural exploration, and prototyping.

The project encompasses all critical components of a RISC-V processor, including the **program counter (PC)**, **instruction memory**, **register file**, **arithmetic logic unit (ALU)**, **immediate generator**, **control logic**, and **data memory**. These modules are designed from the ground up and interconnected to form a complete and functional processor capable of executing a subset of the RV32I instruction set.

To validate the design, functional simulation and hardware testing were performed using the **Xilinx Vivado Design Suite**. The processor was synthesized and deployed on an FPGA, demonstrating correct behavior for basic instruction sequences and real-time operation.

Through this work, the thesis aims to bridge theoretical computer architecture concepts with practical digital hardware design, offering a strong foundation for future enhancements such as pipelining, exception handling, and custom instruction support. It also demonstrates the capability of FPGAs as a platform for hands-on architectural learning and prototyping of modern computing systems.

1.2 MOTIVATION

Understanding the internal workings of a microprocessor is fundamental to the study of computer architecture and digital design. However, modern commercial processors are highly complex and inaccessible for most students and developers due to their proprietary nature. The emergence of RISC-V, an open-source instruction set architecture, has created a valuable opportunity for learners and researchers to explore processor design in a transparent and flexible manner.

The motivation behind this project stems from the desire to gain hands-on experience in processor architecture by designing and implementing a working microprocessor from the ground up. A single-cycle RISC-V processor provides an ideal starting point: its simplicity makes it easy to understand and implement while still encompassing all the essential components and behaviours of a real CPU.

Furthermore, implementing the processor on an FPGA platform such as the PYNQ-Z2 allows for direct hardware-level validation and real-time interaction, bridging the gap between simulation and actual hardware execution. This not only reinforces theoretical learning but also introduces valuable skills in hardware description languages (HDLs), FPGA toolchains, and digital debugging.

The project is also motivated by the growing relevance of custom processor design in fields like embedded systems, security, and domain-specific accelerators. A deep understanding of how instructions are fetched, decoded, and executed at the hardware level is critical for designing efficient and specialized computing solutions.

By undertaking this project, the goal is to develop both a foundational architectural understanding and the practical skills necessary to work on more advanced processor enhancements in the future, such as pipelining, memory hierarchies, and system-on-chip integrations.

1.3 PROBLEM STATEMENT

Gaining a deep understanding of processor architecture requires more than theoretical knowledge—it demands hands-on experience with actual hardware implementations. However, most commercial processors are too complex or proprietary for educational exploration. Although the **RISC-V ISA** provides an open and modular framework ideal for such learning, implementing it in hardware remains a challenging task for students and beginners.

This project addresses the need for an accessible and educational processor design by **developing a single-cycle RISC-V microprocessor** using **Verilog HDL**, and **deploying it on an FPGA**. The objective is to bridge the gap between architectural theory and digital system design, enabling real-time instruction execution, functional validation, and a foundation for further architectural enhancements.

1.4 OBJECTIVES

The main objectives of this project are:

- To design and implement a single-cycle RISC-V microprocessor using Verilog HDL, adhering to the RV32I base integer instruction set.
- To develop essential processor modules including the program counter, instruction memory, register file, ALU, immediate generator, control unit, and data memory.
- To ensure correct instruction execution, register updates, and memory interactions within a single clock cycle.
- To deploy the processor design on the Nexys 4 FPGA board using the Vivado Design Suite for synthesis and implementation.
- To validate functionality through simulation and hardware testing, verifying correct behavior for a representative set of RISC-V instructions.
- To gain hands-on experience in digital design, computer architecture, and hardware-level system implementation using FPGA platforms.

LITERATURE REVIEW

The design and implementation of microprocessors have been widely studied and documented in the field of computer architecture and digital systems. Foundational texts provide the theoretical and practical frameworks essential for building a RISC-based processor from the ground up. This project draws heavily from the principles outlined in *Computer Architecture: A Quantitative Approach by David A. Patterson and John L. Hennessy*, and *Digital Design and Computer Architecture: RISC-V Edition by David Harris and Sarah Harris.*

Patterson and Hennessy's work emphasizes a quantitative, performance-driven approach to computer architecture. Their book introduces the RISC (Reduced Instruction Set Computer) philosophy, advocating for simplified instruction sets to achieve higher performance and efficiency. The concept of single-cycle execution is discussed as an introductory architectural model, highlighting its simplicity and ease of analysis, albeit with trade-offs in scalability and performance.

The RISC-V Edition of Digital Design and Computer Architecture by Harris and Harris offers a practical and implementation-focused perspective. It introduces the RV32I instruction set and walks through the design of a single-cycle RISC-V processor, providing modular HDL examples and simulation guidance. The book details the development of core components such as the ALU, control unit, and register file, and explains how to handle instruction decoding and immediate generation. These practical examples and diagrams served as a blueprint for the architecture developed in this project.

This literature not only guided the architectural decisions made during the design process but also shaped the development methodology, from initial HDL coding to final hardware validation.

METHODOLOGY

3.1 Design Overview

The development of the single-cycle RISC-V microprocessor was carried out using a modular design methodology, where each functional block of the processor was implemented as an independent Verilog module. These modules—including the program counter, ALU, register file, instruction memory, control unit, and immediate generator—were individually simulated and verified to ensure correctness before being integrated into a unified datapath.

The design process followed a structured workflow comprising **requirement analysis, HDL coding, functional simulation, synthesis, implementation**, and **hardware testing**, all conducted using the **Vivado Design Suite**.The **PYNQ-Z2 FPGA development board** was used for deployment and testing. Since the board does not feature a seven-segment display, **hardware verification was achieved by outputting the least significant 4 bits of the write-back data to the 4 onboard LEDs**, allowing for real-time observation of instruction execution and register updates. This method provided a simple yet effective way to confirm the correct behavior of the processor without requiring external peripherals.

3.2 Module Development

3.2.1 Program Counter (PC.v)

The Program Counter (PC) module is a fundamental component of the processor datapath, responsible for storing the address of the current instruction and updating it each clock cycle. It receives a 32-bit input pc_i, which is the next instruction address, and outputs this value as pc_o on the rising edge of the clock. The module includes a synchronous reset mechanism: when the active-low reset signal rst is deasserted (rst = 0), the PC value is reset to 0, ensuring a clean start during system initialization.

3.2.2 Adder (Adder.v)

The Adder module performs 32-bit signed arithmetic addition and is primarily used for computing the next program counter value and branch target addresses in a RISC-V processor datapath. It takes two 32-bit signed inputs, a and b, and produces their sum on the output sum.

3.2.3 Instruction Memory (InstructionMemory.v)

The **InstructionMemory** module implements a read-only memory (ROM) that stores the instruction set for the processor. It takes a 32-bit word-aligned address input (readAddr) from the Program Counter (PC) and outputs the corresponding 32-bit instruction (inst).

The internal memory array insts holds up to 128 instructions (each 32 bits wide). During simulation or synthesis, the memory is initialized to zero and then populated using the $readmemh system task from an external file (TEST_INSTRUCTIONS.dat).

3.2.4 Control Unit (Control.v)

The **Control** module is responsible for generating all control signals required by different components of the processor based on the 7-bit **opcode** from the instruction. It decodes the instruction type and configures signal paths to guide the processor's behavior in the Execute, Memory, and Writeback stages.

This module is designed as a combinational circuit using a case block sensitive to changes in the input opcode. For each supported instruction type (e.g., R-type, I-type, Load, Store, Branch, JAL, AUIPC, LUI), a specific combination of control signals is asserted to properly route data through the datapath.

Signal	Description
branch	Asserted for conditional branch instructions (beq, etc.)
jump	Asserted for unconditional jumps like jal
memRead	Enables reading from data memory (e.g., in lw)
memWrite	Enables writing to data memory (e.g., in sw)
ALUSrc	Chooses between register or immediate for ALU second operand
regWrite	Enables writing to the register file
resultSRC	Selects the source of data written back to registers (ALU, Memory, PC+4, etc.)
ALUOp	Encodes ALU operation type (to be further decoded by ALU Control)

Table 3.2.4.1 Key Control Signals

Opcode	Instruction Type	Behavior Summary
0000011	lw	Load from memory to register
0100011	sw	Store register to memory
0110011	R-type (add, sub, etc.)	Register-register ALU ops
1100011	Branch	Conditional PC update
0010011	I-type ALU	ALU with immediate operand
1101111	jal	Jump and link
0010111	auipc	PC-relative upper immediate
0110111	lui	Load upper immediate

Table 3.2.4.2 Instruction Support Breakdown

3.2.5 Register File (Register.v)

The Register module implements the 32 general-purpose registers defined in the RISC-V ISA, each 32 bits wide. This module supports simultaneous reading from two registers and conditional writing to one register, all synchronized with the system clock.

3.2.6 Immediate Generator (ImmGen.v)

The **Immediate Generator** (ImmGen) module is responsible for extracting and sign-extending immediate values from instruction fields, based on the instruction type.

Slices the instruction fields and sign-extends the extracted immediate according to the format:

- **I-type:**
  Uses bits [31:20]. Sign-extended to 32 bits.
- **S-type:**
  Concatenates bits [31:25] and [11:7]. Sign-extended.
- **B-type:**
  Immediate is constructed from bits [31], [7], [30:25], and [11:8], with a trailing zero (1'b0) for word alignment. Sign-extended.
- **J-type:**
  Immediate is constructed from bits [31], [19:12], [20], and [30:21], followed by 1'b0. Sign-extended.
- **U-type:**
  Takes bits [31:12] and shifts left by 12 (pads with 12 zeros) to form a 32-bit upper immediate.

3.2.7 2-to-1 Multiplexer (Mux2to1.v)

The **Mux2to1** module implements a basic 2-input, 1-output multiplexer. It is a reusable and parameterized combinational logic component used to select between two 32-bit (or variable-width) signed inputs based on a single control signal.

3.2.8 ALU Control Unit (ALUCtrl.v)

The **ALU Control Unit** is responsible for generating a 4-bit control signal (ALUCtl) that directs the Arithmetic Logic Unit (ALU) to perform a specific operation. It interprets the instruction type (ALUOp), function fields (funct3, funct7), and opcode (op) to determine the ALU behavior.

3.2.9 Branch Logic Unit (BranchLogic.v)

The **BranchLogic** module determines whether a branch or jump should be taken by controlling the PCSrc signal. It evaluates branch conditions based on instruction type, ALU output, and flags such as zero.

3.2.10 Data Memory Unit (DataMemory.v)

The **DataMemory** module implements a 128-byte memory for load and store operations, with support for different memory access widths (byte, half-word, word) and both sign and zero extension for loads. It follows the RISC-V memory access conventions.

3.2.11 Single Cycle CPU (SingleCycleCPU.v)

The SingleCycleCPU module implements a top-level integration of a RISC-V single-cycle CPU datapath. It coordinates all components—registers, ALU, memory, control unit, etc.—to fetch, decode, and execute instructions in a single clock cycle.

3.2.12 Arithmetic Logic Unit (ALU.v)

The **Arithmetic Logic Unit (ALU)** executes the core arithmetic and logical operations as directed by the control signal (ALUCtl). It operates on two 32-bit operands and produces a 32-bit result along with a **zero flag**, which is used for conditional branching.

ALUCtl Code	Operation	Description
0000	ADD	Addition: A + B
0001	SUB	Subtraction: A - B
0010	AND	Bitwise AND: A & B
0011	OR	Bitwise OR: A | B
0100	XOR	Bitwise XOR: A ^ B
0101	SLT	Set if less than (signed)
0110	SLTU	Set if less than (unsigned)
0111	SLL	Shift left logical: A << B[4:0]
1000	SRL	Shift right logical: A >> B[4:0]
1001	SRA	Shift right arithmetic: A >>> B[4:0]
1010	LUI	Load upper immediate: B << 12
1011	AUIPC	Add upper immediate to PC: A + (B << 12)

Table 3.2.12.1 Supported Operation

RESULTS:

Simulations:
The following instructions were used to test for different instruction types.

1. R-Type Instructions:

```
1  li x5, 10                                                    IF
2  li x6, 5
3  add x7, x5, x6
4  sub x8, x5, x6
5  and x9, x5, x6
6  or x10, x5, x6
7  xor x11, x5, x6
8  sll x12, x5, x6
9  srl x13, x5, x6
10 sra x14, x5, x6
11 slt x15, x5, x6
12 sltu x16, x5, x6
13
```

Fig 1.1 R-Type Instructions

```
0:      00a00293        addi x5 x0 10                           IF
4:      00500313        addi x6 x0 5
8:      006283b3        add x7 x5 x6
c:      40628433        sub x8 x5 x6
10:      0062f4b3        and x9 x5 x6
14:      0062e533        or x10 x5 x6
18:      0062c5b3        xor x11 x5 x6
1c:      00629633        sll x12 x5 x6
20:      0062d6b3        srl x13 x5 x6
24:      4062d733        sra x14 x5 x6
28:      0062a7b3        slt x15 x5 x6
2c:      0062b833        sltu x16 x5 x6
```

Fig 1.2 R-Type Instructions Disassembled

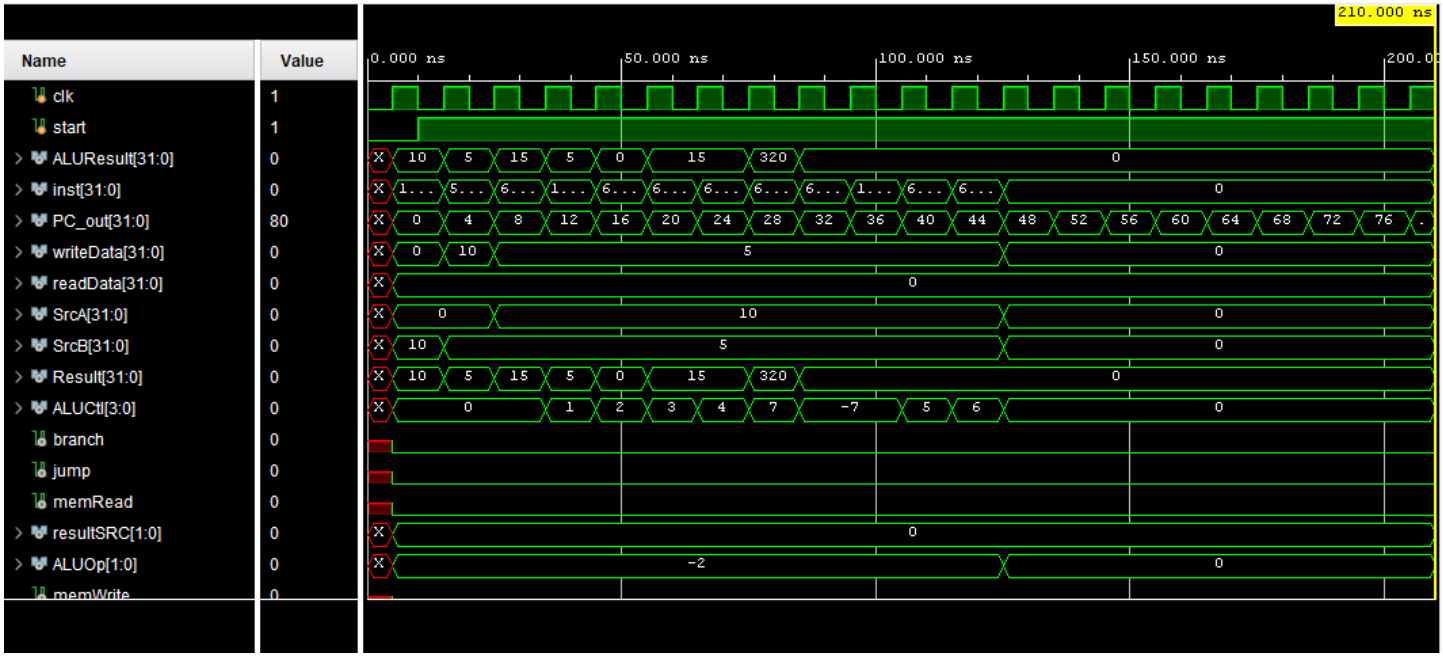

Fig 1.3 R-Type Instructions Simulation

regs[0:31][31:0]	00000000,00000000,00000080,00000000,00000000,0000000a,00000005,0000000f,00000005,00000000,0000000f,0000000f,00000140,00000000,00000000,00000000,0
[0][31:0]	00000000
[1][31:0]	00000000
[2][31:0]	00000080
[3][31:0]	00000000
[4][31:0]	00000000
[5][31:0]	0000000a
[6][31:0]	00000005
[7][31:0]	0000000f
[8][31:0]	00000005
[9][31:0]	00000000
[10][31:0]	0000000f
[11][31:0]	0000000f
[12][31:0]	00000140
[13][31:0]	00000000
[14][31:0]	00000000

Fig 1.3 R-Type Instructions Register Window

2. I-Type Instructions:

```
1  li x5, 10                                                    IF
2  addi x6, x5, -5
3  andi x7, x5, 3
4  ori x8, x5, 4
5  xori x9, x5, 6
6  slli x10, x5, 2
7  srli x11, x5, 1
8  srai x12, x5, 1
9  slti x13, x5, 20
10 sltiu x14, x5, -1
```

Fig 2.1 I-Type Instructions

```
0:      00a00293        addi x5 x0 10                            IF
4:      ffb28313        addi x6 x5 -5
8:      0032f393        andi x7 x5 3
c:      0042e413        ori x8 x5 4
10:      0062c493        xori x9 x5 6
14:      00229513        slli x10 x5 2
18:      0012d593        srli x11 x5 1
1c:      4012d613        srai x12 x5 1
20:      0142a693        slti x13 x5 20
24:      fff2b713        sltiu x14 x5 -1
```

Fig 2.2 I-Type Instructions Disassembled

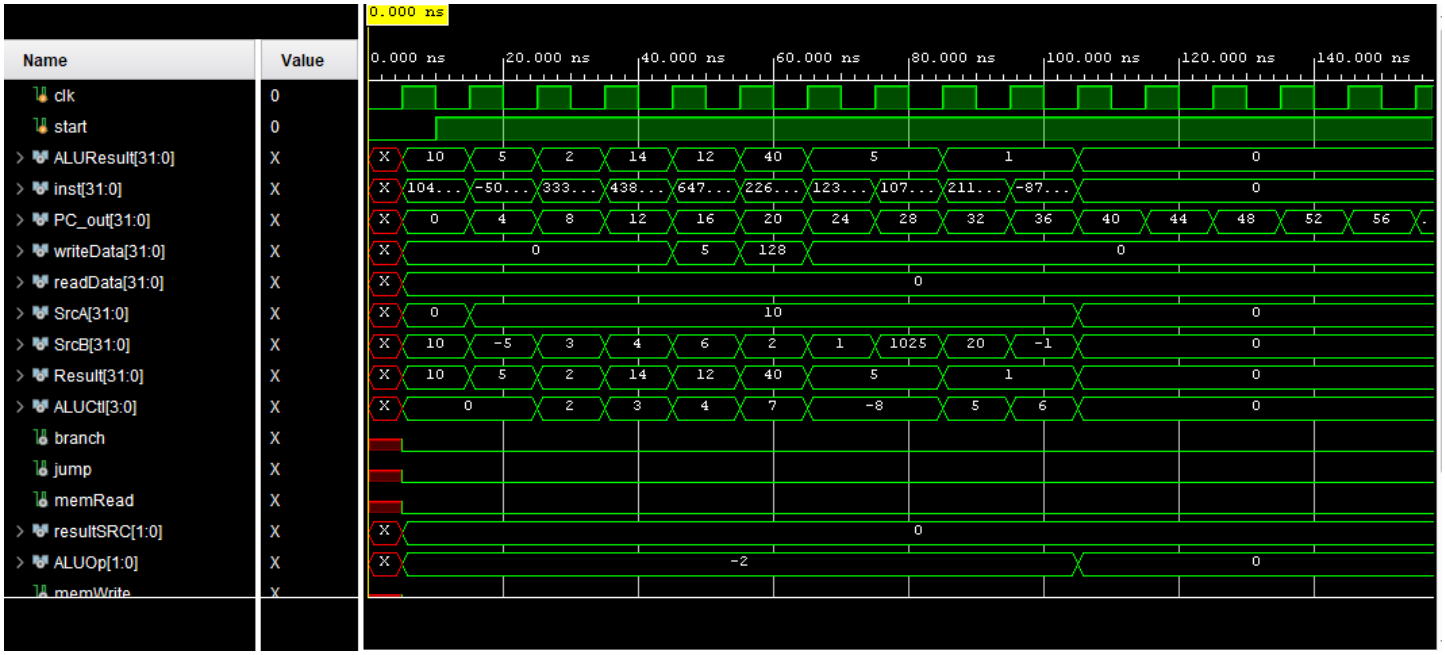

Fig 2.3 I-Type Instructions Simulation

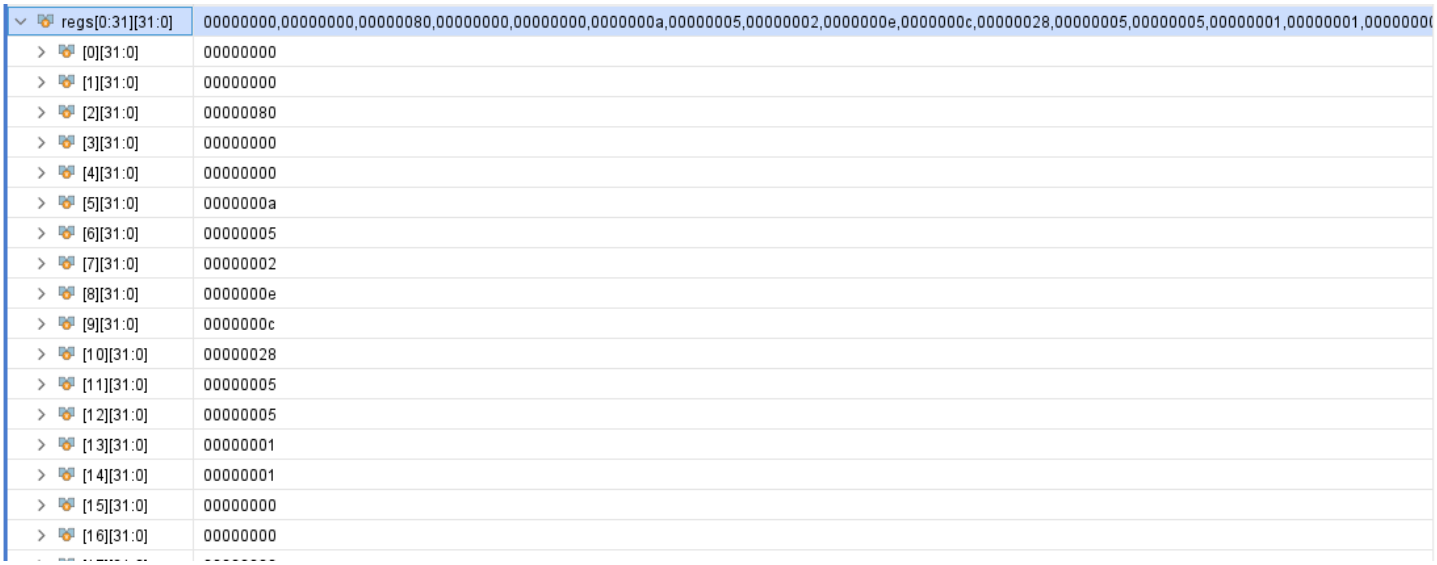

Fig 2.4 I-Type Instructions Register Window

3. B-Type Instructions:

```
1     li x5, 10                                                    IF
2     li x6, 10
3     li x7, 5
4     beq x5, x6, equal_label
5     li x8, 1
6  equal_label:
7     li x8, 2
8     bne x5, x7, not_equal_label
9     li x9, 3
10 not_equal_label:
11    li x9, 4
12    blt x7, x5, less_label
13    li x10, 5
14 less_label:
15    li x10, 6
16    bge x5, x7, greater_equal_label
17    li x11, 7
18 greater_equal_label:
19    li x11, 8
```

Fig 3.1 B-Type Instructions

```
    0:        00a00293        addi x5 x0 10                        IF
    4:        00a00313        addi x6 x0 10
    8:        00500393        addi x7 x0 5
    c:        00628463        beq x5 x6 8 <equal_label>
    10:        00100413        addi x8 x0 1

00000014 <equal_label>:
    14:        00200413        addi x8 x0 2
    18:        00729463        bne x5 x7 8 <not_equal_label>
    1c:        00300493        addi x9 x0 3

00000020 <not_equal_label>:
    20:        00400493        addi x9 x0 4
    24:        0053c463        blt x7 x5 8 <less_label>
    28:        00500513        addi x10 x0 5

0000002c <less_label>:
    2c:        00600513        addi x10 x0 6
    30:        0072d463        bge x5 x7 8 <greater_equal_label>
    34:        00700593        addi x11 x0 7

00000038 <greater_equal_label>:
    38:        00800593        addi x11 x0 8
```

Fig 3.2 B-Type Instructions Disassembled

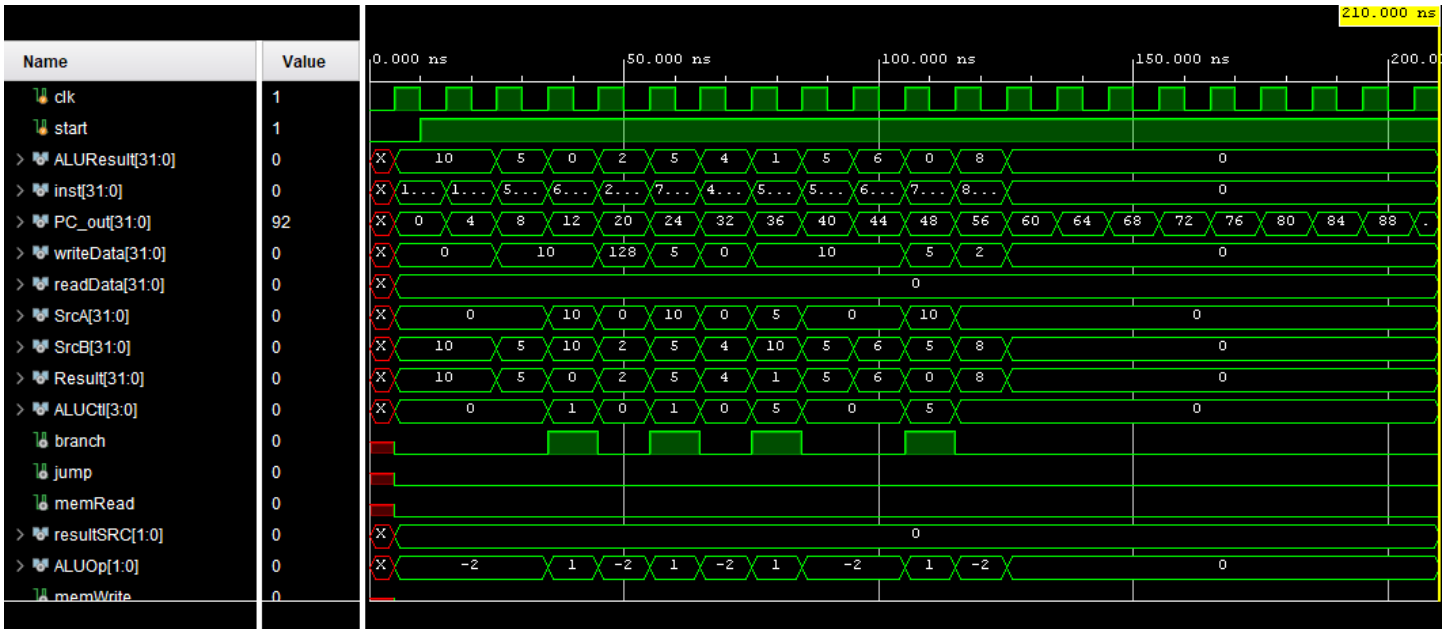

Fig 3.3 B-Type Instructions Simulation

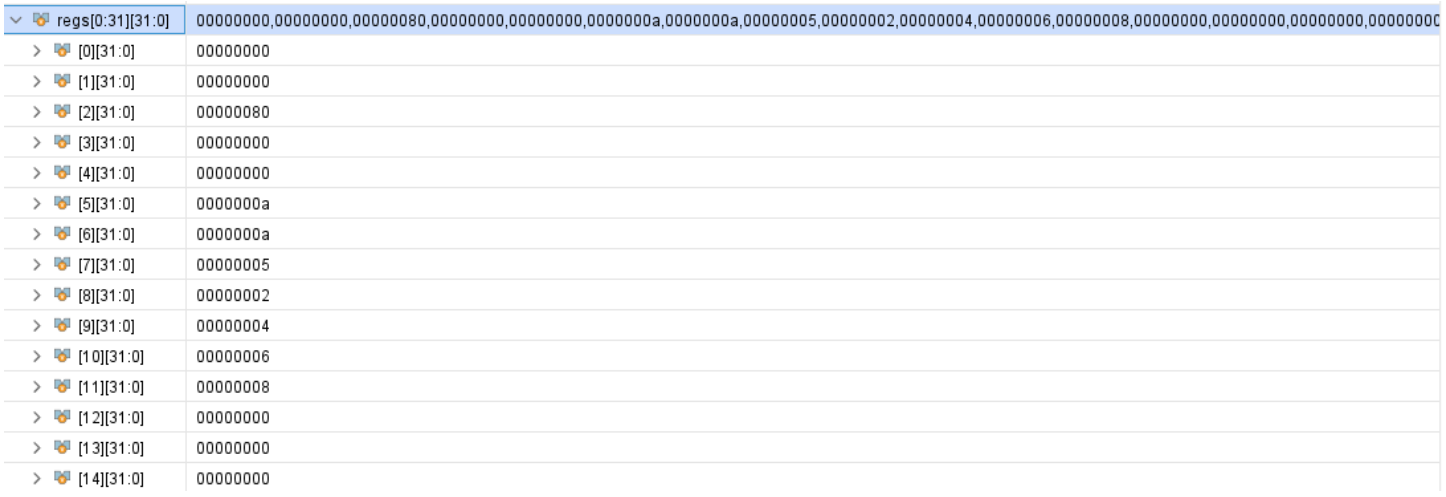

Fig 3.4 B-Type Instructions Register Window

4. J-Type Instructions:

```
1   addi x5, x0, 0                                  IF
2   jal x1, target_jal
3   addi x5, x0, 1
4
5 target_jal:
6   addi x5, x0, 2
7   j target_j
8   addi x6, x0, 0
9
10 target_j:
11  addi x6, x0, 3
12  addi x7, x0, 4
13  jalr x1, x7, 0
14  addi x6, x0, 5
15  addi x6, x0, 6
16
```

Fig 4.1 J-Type Instructions

```
    0:        00000293        addi x5 x0 0                IF
    4:        008000ef        jal x1 8 <target_jal>
    8:        00100293        addi x5 x0 1

0000000c <target_jal>:
    c:        00200293        addi x5 x0 2
    10:        0080006f        jal x0 8 <target_j>
    14:        00000313        addi x6 x0 0

00000018 <target_j>:
    18:        00300313        addi x6 x0 3
    1c:        00400393        addi x7 x0 4
    20:        000380e7        jalr x1 x7 0
    24:        00500313        addi x6 x0 5
    28:        00600313        addi x6 x0 6
```

Fig 4.2 J-Type Instructions Disassembled

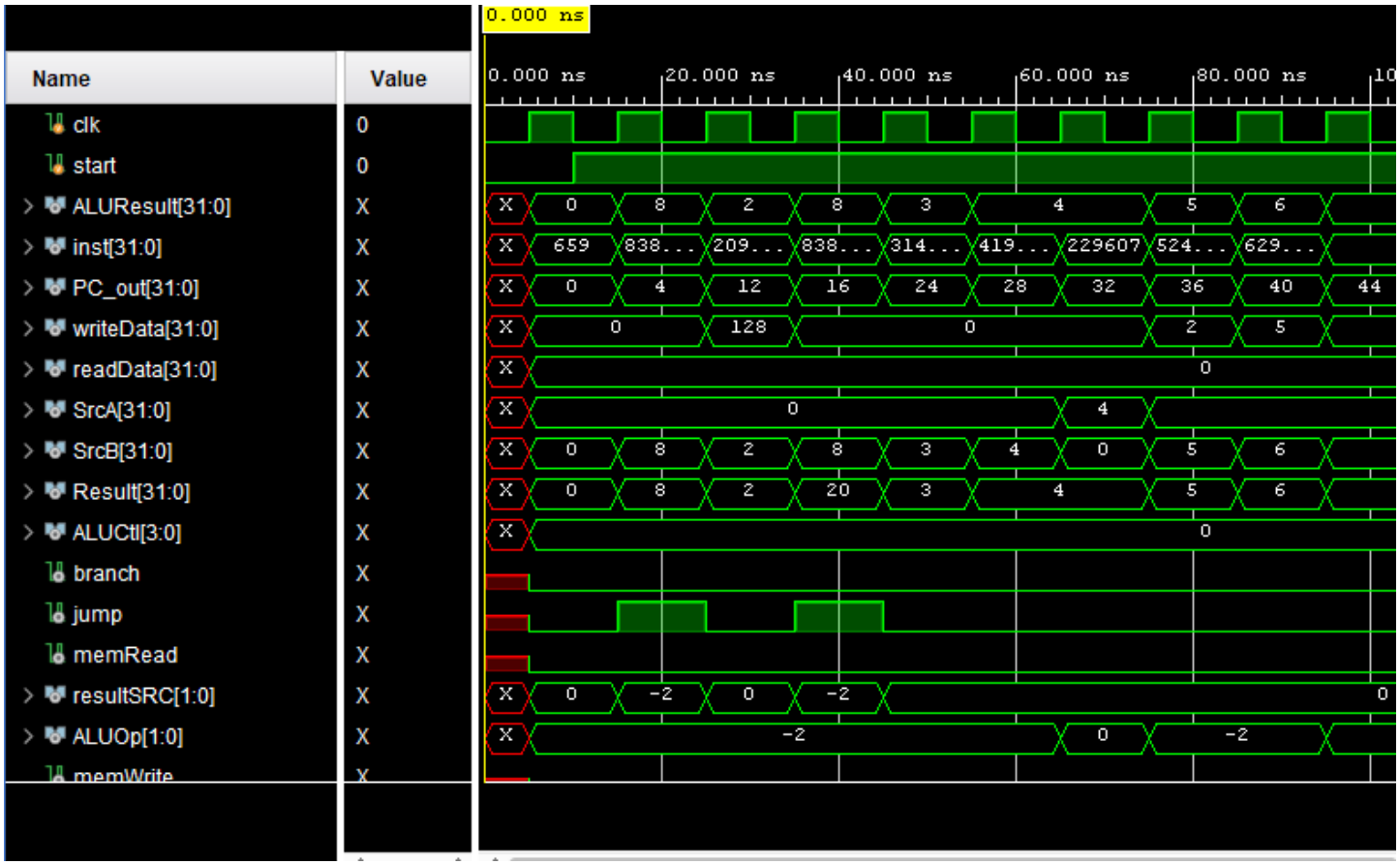

Fig 4.3 J-Type Instructions Simulation

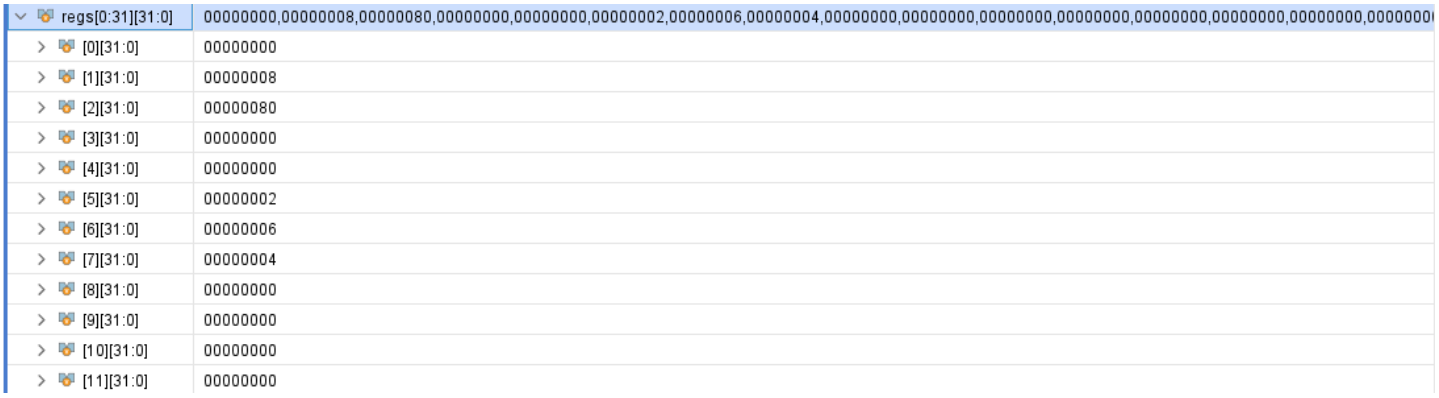

Fig 4.4 J-Type Instructions Register Window

5. U-Type Instructions:

```
1 lui x5, 0x12345
2 lui x6, 0xFFFFF
3 auipc x7, 0x1
4 auipc x8, 0xABCDE
```

Fig 5.1 U-Type Instructions

```
0:      123452b7        lui x5 0x12345
4:      fffff337        lui x6 0xfffff
8:      00001397        auipc x7 0x1
c:      abcde417        auipc x8 0xabcde
```

Fig 5.2 U-Type Instructions Disassembled

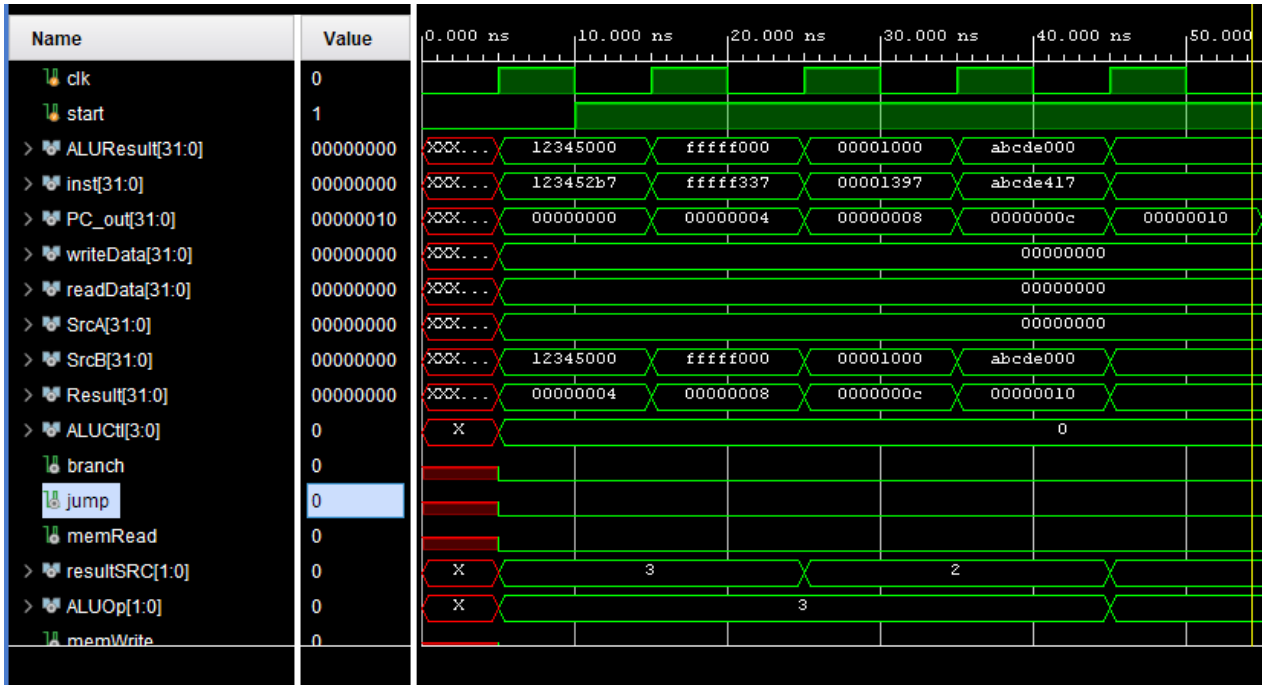

Fig 5.3 U-Type Instructions Simulation

6. General Test-Case Instructions:

```
main:
    addi x2, x0, 5 # x2 = 5 0 00500113
    addi x3, x0, 12 # x3 = 12 4 00C00193
    addi x7, x3, -9 # x7 = (12 - 9) = 3 8 FF718393
    or x4, x7, x2 # x4 = (3 OR 5) = 7 C 0023E233
    and x5, x3, x4 # x5 = (12 AND 7) = 4 10 0041F2B3
    add x5, x5, x4 # x5 = 4 + 7 = 11 14 004282B3
    beq x5, x7, end # shouldn't be taken 18 02728863
    slt x4, x3, x4 # x4 = (12 < 7) = 0 1C 0041A233
    beq x4, x0, around # should be taken 20 00020463
    addi x5, x0, 0 # shouldn't execute 24 00000293
around:
    slt x4, x7, x2 # x4 = (3 < 5) = 1 28 0023A233
    add x7, x4, x5 # x7 = (1 + 11) = 12 2C 005203B3
    sub x7, x7, x2 # x7 = (12 - 5) = 7 30 402383B3
    sw x7, 84(x3) # [96] = 7 34 0471AA23
    lw x2, 96(x0) # x2 = [96] = 7 38 06002103
    add x9, x2, x5 # x9 = (7 + 11) = 18 3C 005104B3
    jal x3, end # jump to end, x3 = 0x44 40 008001EF
    addi x2, x0, 1 # shouldn't execute 44 00100113
end:
    add x2, x2, x9 # x2 = (7 + 18) = 25 48
    sw x2, 0x20(x3) # [100] = 25 0221A023
done:
    beq x2, x2, done # infinite loop 50 00210063
```

Fig 6.1 General Test Instructions

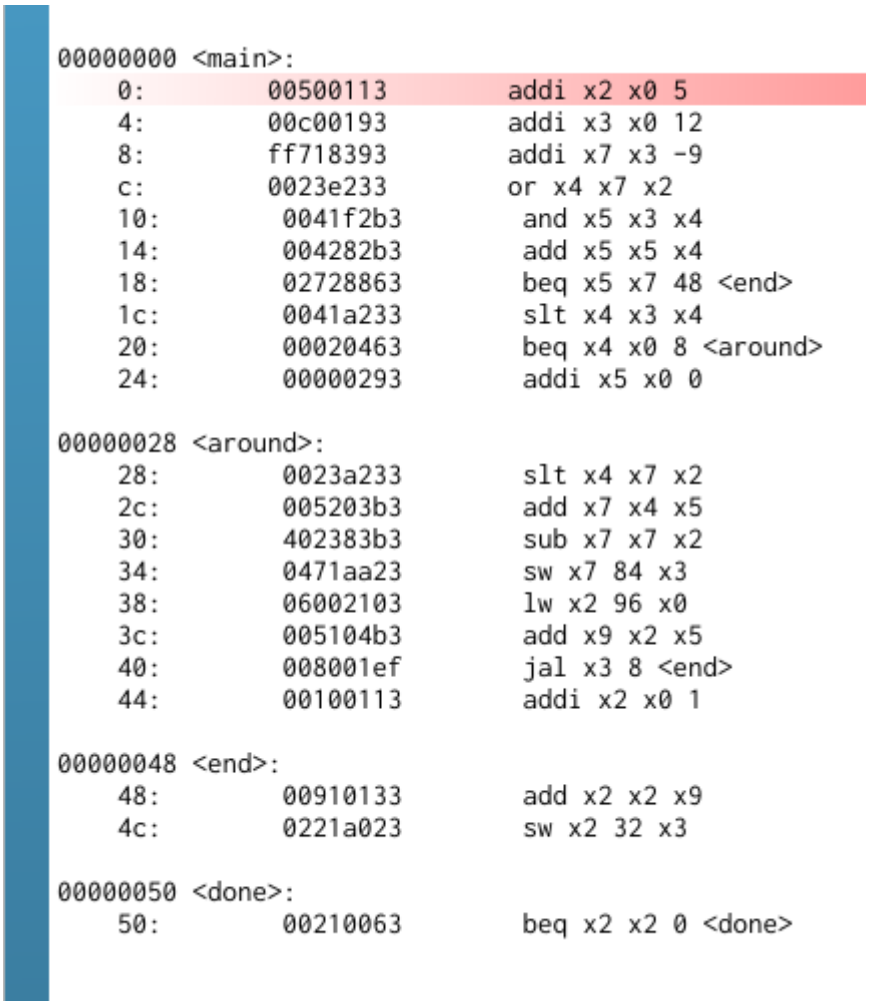

Fig 6.2 General Test Instructions Disassembled

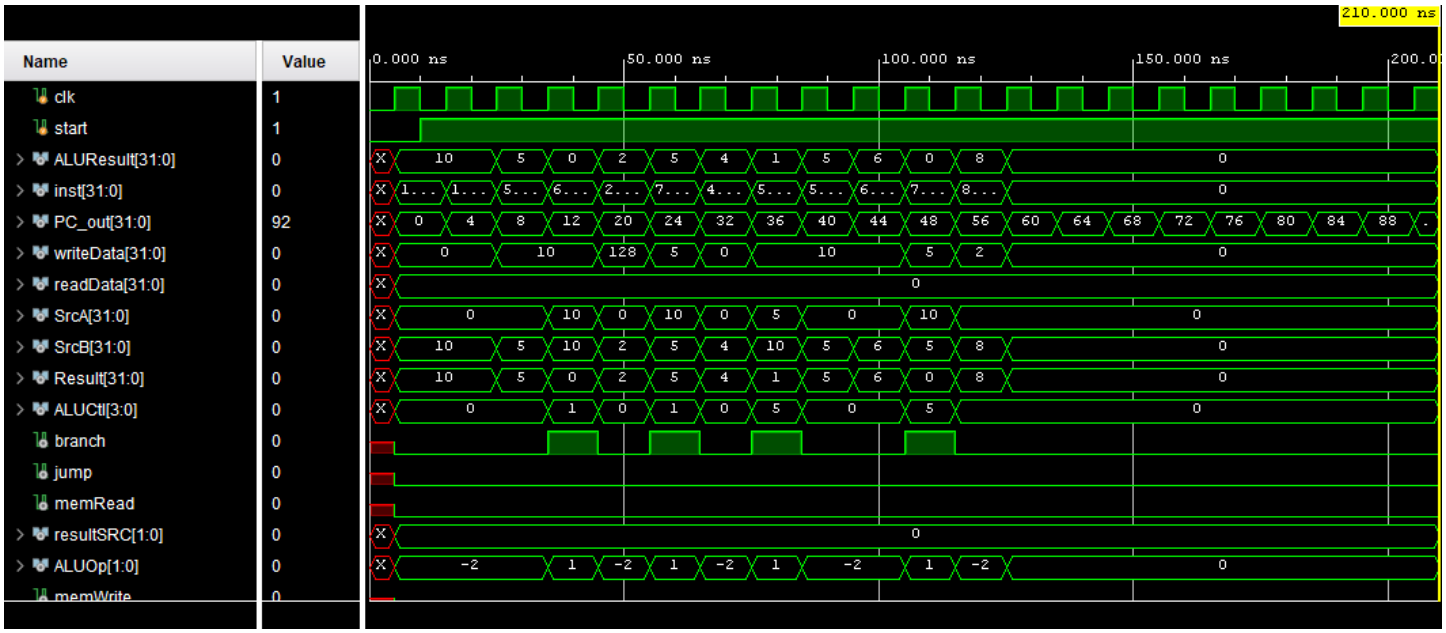

Fig 6.3 General Test Instructions Simulation

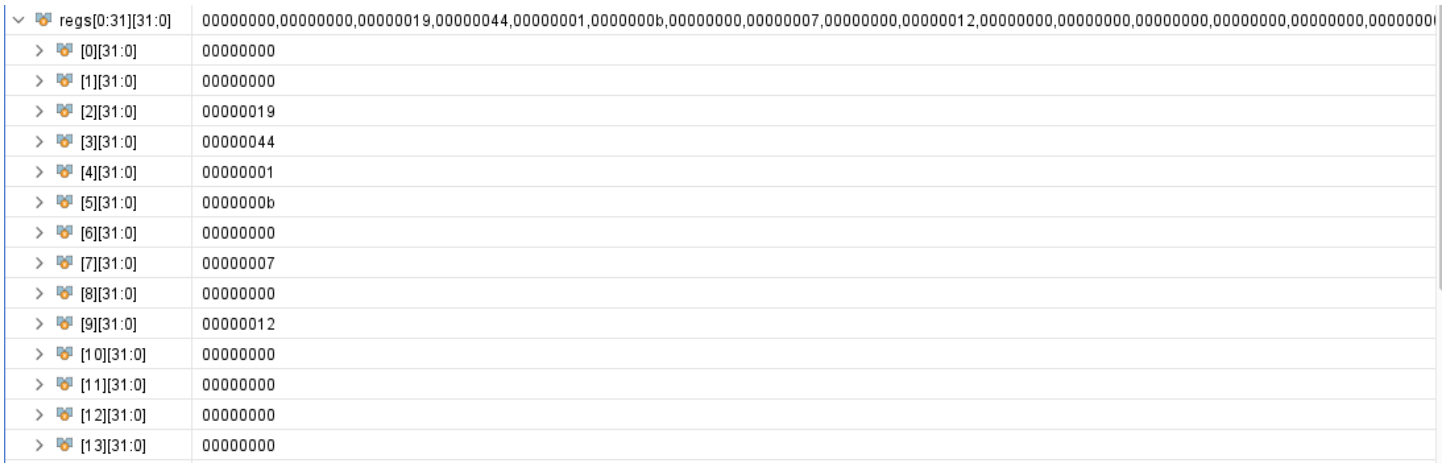

Fig 6.4 General Test Instructions Register Window

Hardware Output:

The following R-Type instructions were used to test the FPGA Implementation on the PYNQ-Z2 board. The video of the working design is linked here.

```
li x5, 10        # Load 10 into x5
li x6, 5         # Load 5 into x6
add x7, x5, x6   # x7 = x5 + x6 = 10 + 5 = 15
sub x8, x5, x6   # x8 = x5 - x6 = 10 - 5 = 5
and x9, x5, x6   # x9 = x5 & x6 = 10 & 5 = 0
or x10, x5, x6   # x10 = x5 | x6 = 10 | 5 = 15
sll x12, x5, x6 # x12 = x5 << (x6 % 32) = 10 << 5 = 0
srl x13, x5, x6 # x13 = x5 >> (x6 % 32) = 10 >> 5 = 0

```

All the source codes and test bench can be found here.

CONCLUSION

The successful implementation of the single-cycle RISC-V microprocessor on an FPGA platform represents a significant achievement in understanding and applying core principles of computer architecture and digital design. By leveraging Verilog HDL, the project demonstrated a complete microprocessor design with critical components such as the ALU, control unit, register file, instruction memory, and data memory, functioning seamlessly together in a single clock cycle.

The processor was tested against a variety of RISC-V instructions, including arithmetic, logic, memory, and branching operations, with successful verification using a range of test programs. The modular design approach allowed for efficient development, debugging, and integration of various functional units, ensuring that each part of the processor was working correctly.

This project lays a solid foundation for future work, such as extending the processor to support multi-cycle and pipelined execution, implementing more advanced RISC-V features, or optimizing performance. Additionally, it serves as a valuable stepping stone for exploring advanced topics in computer architecture, including pipelining, memory hierarchy, and the design of more complex processors. The hands-on experience of working with hardware through FPGA programming deepens understanding and appreciation for the intricacies of embedded system design, and opens up opportunities for further exploration into hardware acceleration and processor optimization techniques.

FUTURE SCOPE

The implementation of the single-cycle RISC-V microprocessor on the FPGA platform serves as a foundational step in exploring more advanced and complex processor designs. Several directions for future development and enhancement can be pursued to expand and refine this project:

1. **Multi-Cycle Processor Design:**
   Moving from a single-cycle to a multi-cycle processor would enhance the processor's efficiency by breaking down the execution of instructions into multiple stages. This approach would reduce the critical path and allow for better timing optimization, leading to faster execution of instructions. A multi-cycle design also provides an opportunity to explore pipeline stages such as instruction fetch, decode, execute, memory access, and write-back.
2. **Pipelining:**
   Implementing a pipelined RISC-V processor is a natural progression from single-cycle design. Pipelining allows for the concurrent execution of multiple instructions, improving throughput and processor performance. The integration of hazards handling (data, control, and structural hazards) and forwarding techniques would be key areas to explore to achieve a fully functional pipeline.
3. **Enhanced Instruction Set Support:**
   Future work could focus on expanding the supported RISC-V instruction set. Currently, the microprocessor supports only a subset of instructions. Adding support for more advanced instructions like floating-point operations, vector instructions, and atomic operations could make the processor more versatile and closer to a real-world implementation.
4. **Memory Hierarchy:**
   Introducing a memory hierarchy with cache memory (L1, L2) or even off-chip memory could significantly improve the performance of the processor. Exploring cache coherency protocols, memory access latency, and efficient data movement would provide insight into real-world processor design challenges.
5. **Integration with Peripherals:**
   The processor can be enhanced by connecting it to various external peripherals like UART, SPI, GPIO, or even an LCD. This would allow the processor to interact with the external environment and enable it to run complex real-world applications. Furthermore, developing peripheral interfaces could lead to the creation of a fully embedded system.

6. **Security Features:**
   Exploring security features such as hardware-based encryption and decryption, secure boot, or memory protection units could be valuable for ensuring secure execution of applications on the RISC-V processor. These features are particularly important in embedded systems and IoT applications.

REFERENCES

[1] C. H. Roth and L. L. Kinney, Fundamentals of Logic Design, 7th ed. Cengage, 2015.

[2] Computer Architecture: A Quantitative Approach, by David A Patterson and John L. Hennessy.

[3]Digital Design and Computer Architecture, RISC-V Edition, by David Harris and Sarah Harris