16-bit RISC Processor Design

A report on the Summer Project

Bachelor of Technology
in
Electrical Engineering

Submitted by

Roll No	Name of Student
U23EE036	Sujay Bhati

Under the guidance of
Dr. Zuber M. Patel

Department of Electrical Engineering
National Institute of Technology Surat
Surat, Gujarat, India – 395 007

Summer Semester 2025

Department of Electrical Engineering

National Institute of Technology Surat

Certificate

This is to certify that this is a bonafide record of the project presented by the student whose name is given below during Summer 2025 as part of their Bachelor of Technology in Electrical Engineering program..

Roll No	Names of Students
U23EE036	Sujay Bhati

Dr. Zuber M. Patel
(Project Guide)

Date:

Abstract

This report presents the design and implementation of a 16-bit Reduced Instruction Set Computer (RISC) processor as part of a summer project. The primary objective of this project was to develop a simplified yet functional processor architecture capable of executing a basic set of instructions efficiently. The processor was designed using modular principles, with distinct components for the Arithmetic Logic Unit (ALU), control unit, registers, and memory interface, enabling easier understanding and future scalability. The instruction set includes arithmetic, logical, and data transfer operations, and the processor employs a multi-cycle execution model, requiring a minimum of 4 and a maximum of 8 clock cycles to execute a instruction. The implementation was carried out using hardware description language (HDL) and simulated on a digital design platform, allowing verification of correct operation through comprehensive test benches. Results demonstrate that the processor correctly executes all implemented instructions and produces the expected outputs, validating the functional correctness of the design. This project provides practical insights into processor architecture, instruction execution, and hardware design methodology, serving as a foundation for more complex processor designs in future work.

Contents

List of Figures

Chapter 1

Problem Definition

The purpose of this project is to design and implement a simplified 16-bit RISC processor that demonstrates the essential principles of computer architecture, instruction execution, and hardware design. The project aims to provide a practical, hands-on understanding of how a processor operates internally—covering datapath design, control logic, instruction flow, and timing behavior.

By building this processor from scratch, the objective is to strengthen core concepts such as instruction decoding, register operations, ALU functionality, memory access, and pipeline/control sequencing. The project also intends to highlight how architectural decisions—like instruction formats, cycle count, control signals, and datapath organization—affect the performance, complexity, and efficiency of a processor.

Another purpose of this work is to expose students to industry-relevant design tools and methodologies. By simulating the processor, verifying each instruction through waveforms, and testing the overall operation, the project aims to bridge the gap between theoretical knowledge and real-world digital hardware implementation workflows.

Ultimately, this project serves as a foundational step toward understanding more advanced processor concepts, enabling further exploration into pipelining, hazard handling, optimization techniques, and modern CPU architectures.

Chapter 2

Introduction

2.1 Background

A Reduced Instruction Set Computer (RISC) architecture is based on the principle of using a small, optimized set of simple instructions that can execute quickly and efficiently. RISC processors typically follow a uniform instruction format, simple addressing modes, and a streamlined datapath that enables faster execution compared to traditional CISC processors.

With advancements in embedded systems, IoT devices, and low-power applications, compact processors with predictable execution behavior are widely used. Designing a custom 16-bit RISC processor allows exploration of fundamental computer architecture concepts such as datapath design, control unit implementation, instruction set design, and multi-cycle execution. This project focuses on understanding the working of a basic processor by building it from the ground up using HDL, simulation, and verification.

2.2 Motivation

The motivation behind this project is to gain practical experience in digital system design and to bridge the gap between theoretical processor architecture concepts and real hardware implementation. While studying computer architecture, it is often difficult to fully understand how instructions move through the datapath and how control signals operate internally. Building a processor provides hands-on understanding of:

- How instructions are fetched, decoded, and executed
- How ALU, register file, and memory interact

- How multi-cycle control logic is designed
- How to verify each instruction using simulations and waveforms

Additionally, designing a custom ISA and implementing all instructions strengthens understanding of low-level programming, hardware–software interaction, and embedded system design.

2.3 Objectives of the Project

1. To design and implement a 16-bit RISC processor with a efficient instruction set.
2. To develop a complete datapath including ALU, register file, control unit, program counter, and memory interface.
3. To design the multi-cycle control logic allowing each instruction to execute in 4–8 clock cycles.
4. To implement and test all instructions through behavioral and timing simulations.
5. To verify correctness using waveform analysis for every instruction in the ISA.
6. To document the architecture, design methodology, and simulation results in a structured report.

2.4 Scope of the Work

The scope of the project includes the following:

- Designing a 16-bit RISC processor using HDL (Verilog/VHDL).
- Creating a datapath with ALU, register file, instruction register, memory interface, and PC logic.
- Developing the control unit using a finite state machine (FSM) to support multi-cycle execution.
- Defining a custom 16-bit instruction set architecture with fixed instruction formats.
- Simulating every instruction and capturing waveforms for ADD, SUB, LOAD, STORE, AND, OR, JUMP, BRANCH, etc.

Chapter 3

Litrature-Review

3.1 Introduction

The purpose of this literature review is to understand the background concepts and previous work related to processor design, particularly RISC-based architectures. Before designing a custom processor, it is important to study how existing architectures are structured, how instruction sets are defined, and how execution cycles are managed. This chapter reviews the key ideas behind RISC processors, compares them with CISC systems, and examines examples of earlier 16-bit designs. The information collected here builds the foundation for the design decisions made in this project.

3.2 RISC Architecture Overview

The Reduced Instruction Set Computer (RISC) paradigm was developed in the early 1980s to address the increasing complexity found in CISC architectures. Early RISC designs such as MIPS, SPARC, and ARM1 demonstrated that simplifying the instruction set and datapath significantly improves processing speed and power efficiency. RISC architectures are characterized by:

- Simple, fixed-length instructions
- Uniform instruction formats
- Load–store design
- Minimal addressing modes
- Highly optimized datapaths

- Reduced control logic complexity

These principles reduce hardware complexity and allow faster execution per instruction. Research shows that RISC processors benefit educational and experimental environments because their architectures are predictable, modular, and easier to implement using HDL.
For undergraduate processor design projects, RISC is especially important because:

- It simplifies datapath design (ALU, register file, multiplexers).
- Control logic becomes easier to derive systematically.
- Waveform analysis becomes clear due to predictable instruction flow.
- The architecture can be extended or modified without breaking design consistency.

This background is directly relevant to this project, where a custom 16-bit RISC architecture was developed using a simple multi-cycle execution model.

3.3 RISC VS CISC

RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) represent two different processor design philosophies. RISC architectures use a small set of simple, fixed-length instructions that execute quickly and rely on a streamlined datapath, making them efficient and easier to implement. In contrast, CISC processors use large, complex instructions that can perform multiple operations but require more hardware and often rely on microcoding. While CISC aims to reduce the number of instructions per program, RISC focuses on reducing the number of cycles per instruction. Due to their simplicity, predictable timing, and lower hardware complexity, RISC architectures are preferred for educational designs, embedded systems, and efficient hardware implementations like the processor developed in this project.

Chapter 4

System Design

4.1 Instruction Set Architecture (ISA)

The instruction design of the processor follows a multi-word instruction format, where the first word contains the opcode, addressing mode, and register-related information, while the second word is used only when required—such as for jump instructions or operations involving a 16-bit immediate value. In this architecture, the opcode and addressing mode extracted from the Instruction Register (IR) are sent to the Control Unit, while the opcode and register addresses are simultaneously forwarded to the ALU/Shifter and Register File, respectively.

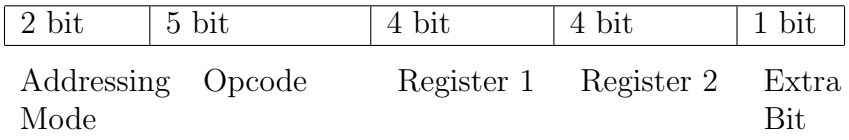
When an instruction requires a second word, the Control Unit activates a dedicated control signal that suppresses normal IR decoding and prevents the opcode/register details from being forwarded. Instead, the second word is routed directly into Datapath Multiplexer 1, ensuring that its entire 16-bit value is made available for immediate or jump operations as needed.

4.1.1 Instruction Format

As shown in the table below, the 16-bit instruction format is divided into multiple fields based on their functionality. The first 2 bits represent the addressing mode, followed by 5 bits that specify the opcode. The next 4 bits store the address of Register 1, and the subsequent 4 bits store the address of Register 2. The final 1 bit is reserved as an extra control or flag bit that may be used depending on the instruction type.

This structured layout ensures that each field has a fixed position, allowing the control unit to easily decode instructions and activate the appropri-

ate datapath components. The separation of addressing mode, opcode, and register fields simplifies decoding logic, improves modularity, and supports flexible instruction types such as immediate, register, and jump instructions.



4.1.2 Classification of Instructions & Addressing Mode

This processor includes three addressing modes and 23 opcodes, with several opcode values intentionally left unused to allow future expansion or addition of new instructions.
The addressing modes include:

Addressing mode	**Bits**
Direct	00
Indirect	01
Immediate	10

1. **Direct:** The instruction contains the actual memory address of the operand, allowing the processor to directly access the data from that location.

2. **Indirect:** The instruction provides a memory location that holds the address of the operand, requiring an extra memory fetch to retrieve the actual data.

3. **Immediate:** The operand value is specified directly within the instruction itself, so no additional memory access is required to fetch it. In this format, the second register field plays a dual role: it provides the value of the second operand and also indicates the destination register where the final result will be stored.

Now Coming to the Opcode they are as follows:

Opcode	Instruction	Opcode	Instruction
00000	XOR	10000	ROR
00001	ADD	10001	ROL
00010	SUB	10010	SHR
00011	INC	10011	SHL
00100	DEC	10100	AHR
00101	AND	10101	AHL
00110	OR	10110	
00111	NOT	10111	

The table above represents the ALU and Shifter instructions. Opcodes 10110 and 10111 are left empty and reserved for future upgrades.

Opcode	Instruction	Opcode	Instruction
01000	BEQ	11000	LOD
01001	BNE	11001	STR
01010	CMP	11010	MOV
01011	CLR	11011	JMP
01100	SET	11100	
01101		11101	
01110		11110	
01111		11111	

The table above represents the Data Transfer and Branch/Jump instructions. The opcodes 01101, 01110, 01111, 11100, 11101, 11110, and 11111 are left empty and reserved for future upgrades.

An example of the instruction is:

00 00001 0000 0001 0

This instruction represents a direct addition operation, where the value stored in register 0000 is added to the value stored in register 0001.

Another example includes:

10 00010 0000 0010 0
00 00000 0000 0110 1

Here, the first word represents the immediate subtraction operation, where the value stored in register 0010 is subtracted from the immediate value provided in the second word. The result of this operation is then stored back

into register 0010, meaning the same register is used both as an operand and as the destination.

Mnemonic	Meaning
ADD	Adds the values of two registers and stores the result in the destination register.
SUB	Subtracts the value of one register from another and stores the result.
INC	Increments the value of a register by 1.
DEC	Decrements the value of a register by 1.
AND	Performs bitwise AND between two registers and stores the result.
OR	Performs bitwise OR between two registers and stores the result.
NOT	Performs bitwise NOT (complement) on a register.
XOR	Performs bitwise XOR between two registers and stores the result.
BEQ	Branches to a specified address if two registers are equal.
BNE	Branches to a specified address if two registers are not equal.
CMP	Compares two registers and sets the Zero flag accordingly.
CLR	Clears a register by setting its value to 0.
SET	Sets all bits of a register to 1.
ROR	Rotates the bits of a register right by one or more positions.
ROL	Rotates the bits of a register left by one or more positions.
SHR	Shifts the bits of a register right (logical shift).
SHL	Shifts the bits of a register left (logical shift).
AHR	Arithmetic shift right: shifts bits right while preserving sign.
AHL	Arithmetic shift left: shifts bits left, preserving sign if needed.
LOD	Loads data from memory into a register.
STR	Stores data from a register into memory.
MOV	Copies data from one register to another.
JMP	Unconditionally jumps to a specified memory address.

4.1.3 Flag Definitions

- **Zero Flag (Z):** Set to 1 when the result of an operation is zero; otherwise 0.
- **Carry Flag (C):** Set to 1 when an arithmetic operation produces a carry-out (in addition) or requires a borrow (in subtraction).
- **Negative Flag (N):** Set to 1 when the result of an operation is negative in signed arithmetic, typically indicated by the MSB being 1.

- **Overflow Flag (V):** Set to 1 when a signed arithmetic operation exceeds the representable range, causing an incorrect change in the sign bit.

To keep the processor architecture simple and reduce hardware complexity, this RISC processor implements only the Zero flag (Z) and the carry flag (C).

4.2 Processor Architecture Overview

The proposed processor is a 16-bit multi-cycle RISC architecture designed with simplicity, modularity, and efficiency in mind. It executes each instruction in 4 to 8 clock cycles, allowing the hardware to reuse functional units across different stages and thus reducing circuit complexity. The instruction set follows the RISC design philosophy, using fixed 16-bit instruction formats, a small number of addressing modes, and a streamlined set of operations.

The core architectural components include an Arithmetic Logic Unit (ALU), a register file, a Program Counter (PC), an Instruction Register (IR), a memory unit, and a control unit that orchestrates data movement through a set of multiplexers. The processor supports Direct, Indirect, and Immediate addressing modes for flexible operand access. To maintain simplicity and minimize hardware overhead, the processor uses only the Zero (Z) flag and the carry flag, which is sufficient for branch decisions and basic conditional operations. Overall, the architecture is optimized for educational purposes and low-complexity hardware implementations.

4.2.1 Execution Flow

- **Fetch:** The instruction is fetched from memory and stored in the Instruction Register (IR).
- **Decode:** The instruction begins decoding. After a wait cycle, the instruction is fully decoded and its fields are distributed to the Control Unit and the ALU.
- **Wait Cycle:** The processor waits briefly to allow the instruction to be fully decoded and control signals to be generated.
- **Execution:**
  - **Cycle 1:** For direct addressing mode, full execution occurs. For immediate and indirect addressing modes, the first half of execution is performed.

  - **Cycle 2:** The remaining half of execution for immediate and indirect modes is completed.

- **Memory Wait Cycle:** Inserted when instructions involve memory access (LOAD/STORE) to give the memory unit sufficient time to complete read or write operations before write-back.
- **Write-Back:** The final result is written back to the destination register.

Fetch → **Decode** → **Wait Cycles** → **Execute (Cycle 1)** →

Execute (Cycle 2) → **Memory Wait Cycles** → **Write Back**

4.2.2 Datapath & Control Interaction

The Control Unit generates signals that manage the flow of data and operations in the datapath.

- **ALUSrc:** Selects whether the ALU's second operand comes from a register or an immediate value.
- **RegWrite:** Enables writing the ALU or memory result back to the destination register.
- **MemRead / MemWrite:** Controls memory access. *MemRead* is asserted during LOAD; *MemWrite* is asserted during STORE.
- **PCSrc:** Determines the next value of the Program Counter (PC), choosing between sequential execution or a branch/jump target address.
- **Multiplexer Control:** Directs multiplexers to route the correct data to the ALU, registers, or memory.

By coordinating these signals across the multi-cycle execution, the control unit ensures:

- Each instruction progresses correctly through fetch, decode, execute, memory, and write-back stages.
- Only required resources are active during each cycle.

- Efficient usage of the ALU, registers, and memory without conflicts.

In this processor, only the Zero flag is used for branching decisions, simplifying the control logic.

4.3 Datapath

The datapath of the proposed 16-bit RISC processor is the hardware subsystem responsible for executing instructions by transporting and transforming data between the Program Counter, instruction memory, register file, ALU, shifter, accumulator, and data memory. It is composed of a tightly-coupled set of modules that work together under the control of the Control Unit to perform the fetch, decode, execute, memory, and write-back phases of every instruction.

In this design, the datapath integrates the following structural components:

4.3.1 Program Counter (PC)

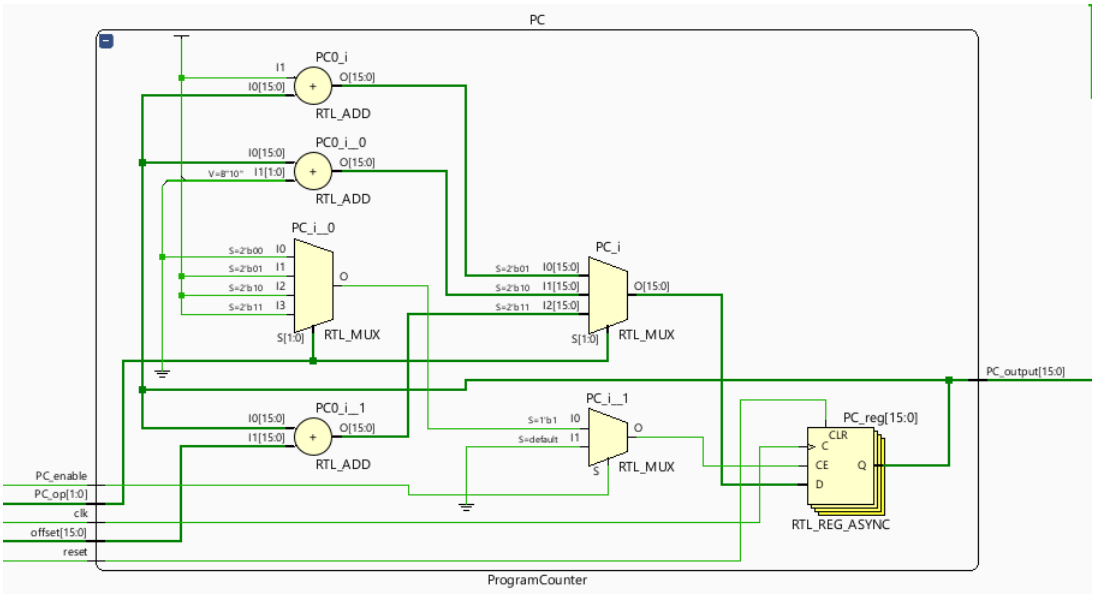


Figure 4.1: Program Counter (PC)

The Program Counter (PC) is a 16-bit register that holds the address of the next instruction to fetch. The PC input is selected by a multiplexer under control of the Control Unit: when PC_op selects sequential execution

the PC receives PC + 1, and when a branch or jump is taken the PC receives PC + offset. The PC output drives the instruction memory (IM) and is used by the Control Unit to compute branch targets and manage control flow.

4.3.2 Register File (Registers)

The Register File is a multi-ported 16-bit storage block that holds the general-purpose registers used by the processor during instruction execution. It provides two simultaneous read ports and one synchronous write port, allowing the datapath to fetch both source operands in a single cycle.

During the Decode stage, the Instruction Register (IR) provides the register indices required for the operation. These indices are fed directly into the Register File, which outputs the corresponding 16-bit values on its two read buses. These values are then routed to the ALU, Shifter, Accumulator (AC), or other datapath modules depending on the current instruction and the control signals.

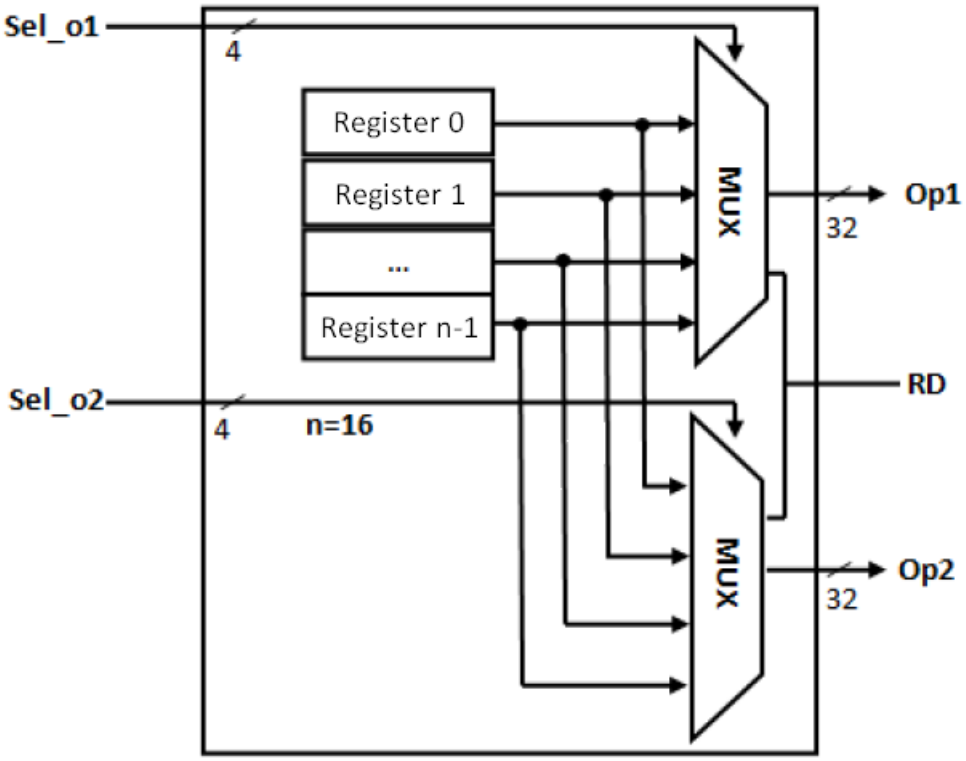


Figure 4.2: Register File Design

4.3.3 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is the central computational element of the processor, responsible for executing all arithmetic, logical, and comparison operations defined in the ISA. In your architecture, the ALU is implemented as a two-stage subsystem consisting of the ALU_pro module and the ALU_out_pro Temporary storing module. One of the value received by the ALU is form the MUX_1 and the Other is received from accumalator,The operation executed by the ALU is determined by the control word issued by the Control Unit during the Execute stage. This control word selects one of the supported operations, which include:

- **Arithmetic:** ADD, SUB, INC, DEC
- **Logical:** AND, OR, XOR, NOT
- **Compare:** CMP (used to set flags without modifying destination registers)

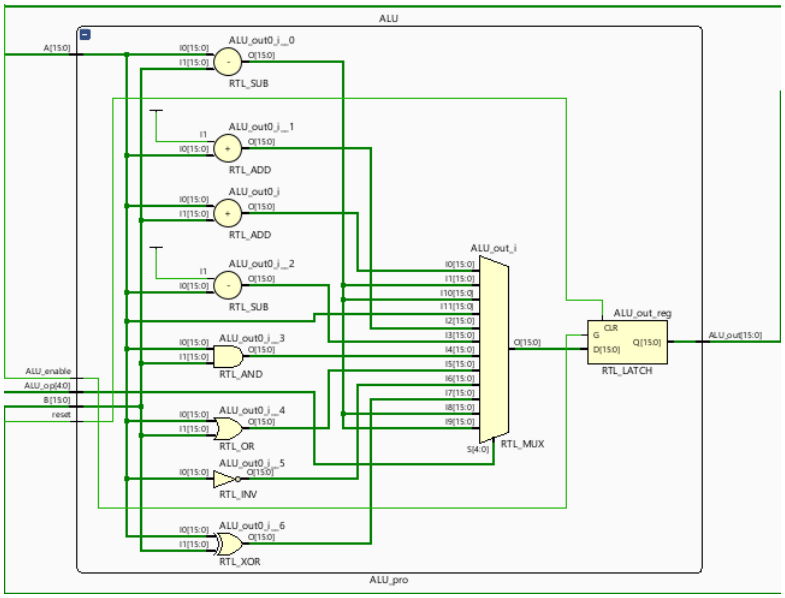


Figure 4.3: Register File Design

During computation, the ALU generates several condition flags (Zero, Carry, Negative, Overflow), but these are not updated internally. Instead, the raw flag values are forwarded to the dedicated Flags module, which latches them only when the Control Unit enables flag updates.

The output of the ALU at this point is a "raw" 16-bit result that may require temporary storage hence is stored in ALU_out_pro before being used by later datapath modules.

4.3.4 Instruction Memory (IM)

The Instruction Memory in this processor is a dedicated 16-bit wide memory unit that stores the program code. It is directly addressed by the Program Counter (PC), which outputs the instruction address every cycle. Since the processor follows a Von Neumann model, the IM shares the same global memory space as data memory, but in this design it is treated as a logically separate module for clarity. During the Fetch stage, the IM provides one 16-bit instruction word to the Instruction Register (IR). The memory is read-only during execution, meaning the datapath never modifies instruction contents. All instructions are stored in fixed 16-bit format, allowing single-cycle access without alignment overhead. The IM outputs remain stable for the entire Fetch cycle, ensuring that the IR captures the correct instruction before decoding and multi-cycle execution begin.

4.3.5 Data Memory (DM)

The Data Memory module is responsible for storing and retrieving 16-bit data values during program execution. Unlike the Instruction Memory, this module supports both read and write operations, allowing instructions such as LOD and STR to access or update memory contents. The effective memory address is provided either directly by the ALU output—typically after computing register + offset or direct register addressing—or via the dedicated Address Register (AR), which temporarily holds intermediate addresses when multi-cycle load/store operations are performed. During a memory read (LOD), the DM outputs a stable 16-bit word that is forwarded to the Write-Back stage through the MemToReg multiplexer. During a memory write (STR), the DM stores the 16-bit data from the selected register into the specified address during the appropriate cycle, controlled by the MemWrite signal.

The Data Memory operates synchronously with the processor clock and only responds when the control unit asserts MemRead or MemWrite, preventing unintended memory access. Because the architecture follows a Von Neumann model, DM and IM share a unified address space conceptually, but are implemented as separate modules in this design for clarity and modularity.

4.3.6 Accumulator (AC)

The AC is a dedicated 16-bit register used to temporarily store the first operand during ALU operations. In this processor, certain instructions require one operand to be pre-loaded before the second operand is selected through the ALUSrc multiplexer. During the Decode or early Execute cycle, the value from the register file (or immediate value, depending on the addressing mode) is transferred into the AC, allowing the Control Unit to stage arithmetic and logical operations over multiple cycles. When the next operand becomes available—either from another register, immediate field, or memory—the ALU receives its two inputs: AC as the first operand and the selected source as the second operand. This staged operand loading mechanism ensures correct execution of multi-cycle ALU instructions such as ADD, SUB, AND, OR, CMP, and shift operations. The AC thus serves as a stable operand-holding register that prevents data hazards inside the datapath and allows the ALU to operate even when the second operand is not yet ready during earlier cycles.

4.4 Control Unit Design

4.4.1 Control Unit Architecture

The Control Unit (CU) of the 16-bit Multi-Word RISC processor is responsible for interpreting the instruction word and generating all the control signals required to coordinate the datapath. Since the processor supports three addressing modes, 23 defined opcodes, and a multi-word instruction format.

The CU receives the 5-bit opcode, the 2-bit addressing mode, and various ALU status flags (Zero, Carry, Sign), and produces a set of control outputs that activate different parts of the datapath. It ensures that each stage—PC update, instruction fetch, operand selection, ALU execution, memory access, and register write-back—occurs in the correct order.

The CU is divided into the following logical blocks:

- **Instruction Decoder** – Extracts opcode and addressing mode bits from the fetched instruction and maps them to control signal patterns.
- **Addressing Mode Handler** – Generates specific control sequences for immediate, register, and memory-based operations, such as selecting the operand source or enabling AC/RF output.

- **Branch/Jump Logic Unit** – Uses opcode and flag bits (Zero, Carry, Sign) to determine PC branching and produces `PC_op` and `PC_enable`.
- **Memory Control Logic** – Enables the data memory for load/store instructions and coordinates read or write cycles.
- **Register File Control Logic** – Controls register selection, write-enable, and multiplexer select lines for routing ALU or memory results back into the RF.

Through these blocks, the CU ensures that for every instruction, the correct combination of multiplexers, registers, and functional units are enabled to perform one complete execution cycle.

4.4.2 Control Signal Generation

Based on the decoded instruction fields and internal state, the Control Unit generates a set of control signals that directly drive the datapath. Each control signal governs a specific hardware action; together, they coordinate the movement of data through the processor.

The major control signals generated are:

1. PC-related Control Signals

- **PC_op** – Selects whether the next PC value comes from `PC + 1` (sequential execution) or `PC + offset` (branch/jump).
- **PC_enable** – Enables updating of the Program Counter during fetch or branch instructions.

2. Instruction Fetch Signals

- **IR_enable** – Loads the fetched instruction into the Instruction Register..
- **IR_control_enable** – when 1 Loads the instruction into the control unit to be further processed and when its 0 it sends the instruction into the datapath to be used as a immediate value.

3. Operand Selection and Routing

- **AC_enable** – Controls when the Accumulator captures the first operand during multi-word or memory instructions.
- **mux1_sel** – Selects the source of the second operand: immediate, register, or memory output.

4. ALU Control Signals

- **ALU_Enable** – It enables the ALU and allow it to perform the speific task on the two operands.
- **Zero_flag_enable** – Allows ALU flags (Zero, Carry, Sign) to update only for arithmetic/logical operations.

5. Memory Control Signals

- **Datamem_enable_read** – Enables data memory read for load instructions.
- **Datamem_enable_write** – Enables data memory write for store instructions.

6. Register File Control Signals

- **Reg_Write_enable** – Determines whether the register file should store the result.
- **mux2_sel** – Selects whether the value written to RF comes from the ALU or Data Memory.

List of Control signal used and there function is given in the table below:

Table 4.1: Control Signals and Their Functions

Control Signal	**Description**
PC_enable	Enables the Program Counter to update its value during the fetch or branch cycle.
PC_op[1:0]	Selects the source for the next PC value (PC+1, PC+offset, jump address, or hold).
mux1_sel[1:0]	Selects the first operand source for the ALU (e.g., register file output, immediate, AC, or memory).
mux2_sel[1:0]	Selects the second operand input for the ALU based on instruction type or addressing mode.
Reg_Write_enable	Enables writing data into the Register File during the write-back stage.
IR_enable	Loads the fetched instruction word into the Instruction Register.
IR_control_enable	It allows the IR to send the data between Control unit and Datapath depending on weather the signal is 1 or 0.

Control Signal	Description
IR_control[6:0]	Input coming from the extended instruction format used for decoding addressing modes and operands.
ALU_enable	Activates the ALU to perform arithmetic or logical operations for the current instruction.
Datamem_enable_read	Enables reading from Data Memory for LOAD or memory-based operand fetch.
Datamem_enable_write	Activates writing to Data Memory for STORE instructions.
AC_enable	Enables the Accumulator register to store the first operand for multi-word or memory instructions.
Zero_flag_enable	Allows updating of the Zero flag after an ALU operation.
Zero_flag	Input flag used by the Control Unit for conditional branch decisions.
Shifter_enable	Activates the barrel shifter for shift or rotate instructions.
ALU_out_enable	Enables the ALU output register to capture the final ALU result for write-back.
AR_enable	Enables the Address Register to store memory addresses for LOAD/STORE instructions.

4.4.3 Control Unit Summary

The Control Unit of the proposed 16-bit RISC processor is implemented as a multi-cycle, FSM-driven controller that supervises every stage of program execution. It operates across eight well-defined states (Fetch, Decode, Decode2, Execute1, Execute2, Wait1, Wait2, and Wait3), each responsible for a specific subset of operations such as instruction loading, operand fetching, ALU execution, memory access, or write-back.

The unit generates all necessary control signals required to orchestrate the datapath, including `PC_enable`, `PC_op`, multiplexer select signals, register file write controls, memory read/write enables, accumulator update control, ALU and shifter activation, and all intermediate register enables. By decoding the 7-bit IR control field, it determines both the addressing mode (Direct, Indirect, or Immediate) and the instruction opcode, allowing the FSM to choose the correct execution path.

Each instruction class follows a unique micro-sequence of states, enabling the processor to execute arithmetic operations, memory access instructions, shifts/rotates, comparisons, and control-flow instructions without pipelining. Indirect instructions use the accumulator and address register to fetch the

effective address; immediate instructions insert an additional fetch cycle for the immediate operand; and direct instructions execute using register-file operands. Branch and jump operations are handled through `PC_op`, allowing the PC to switch between `PC + 1`, `PC + offset`, or a jump target.

Through this finely structured FSM and precisely coordinated control signals, the Control Unit ensures that every datapath component—ALU, shifter, multiplexers, register file, PC, memory interface, and temporary registers—activates only when required. This minimizes hardware complexity while preserving correct sequencing and timing for all instruction types. Overall, the Control Unit provides reliable, deterministic, and fully synchronized execution control, completing the multi-cycle design and ensuring seamless integration of all modules within the processor.

Chapter 5

System-Testing

5.1 Immediate Mode Testing

5.1.1 LOAD Test

Loading an immediate value into a register using instruction code.

10 11000 0000 0000 0
00 00000 0000 0110 1

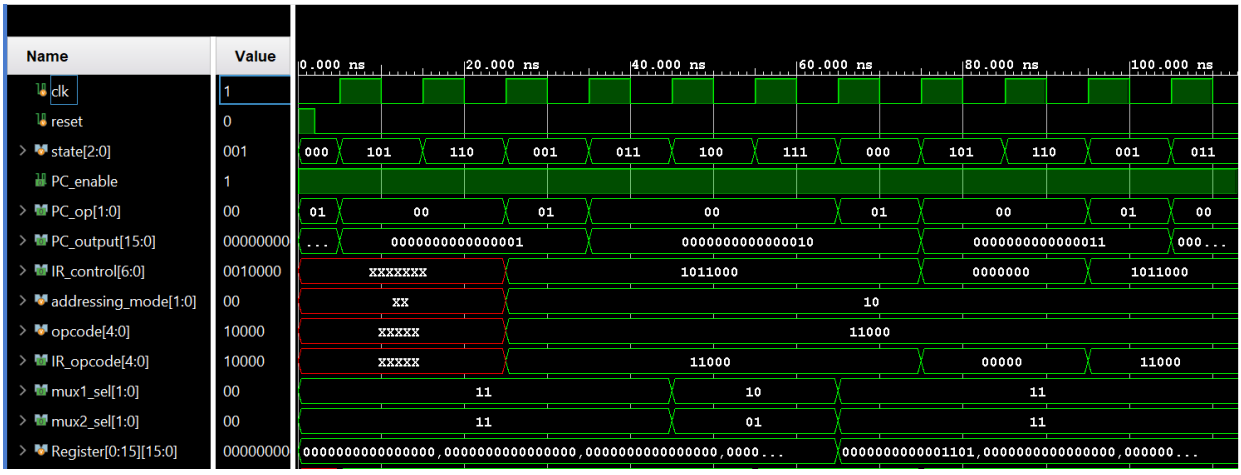


Figure 5.1: Immediate Load

5.1.2 ALU Test

First Storing a value into the register-1 using the instruction:

10 11000 0000 0000 0
00 00000 0000 0110 1

ALU operation between a register value and an immediate value getting stored into a register using instruction code:

10 11000 0000 0000 0
00 00000 0000 0110 1

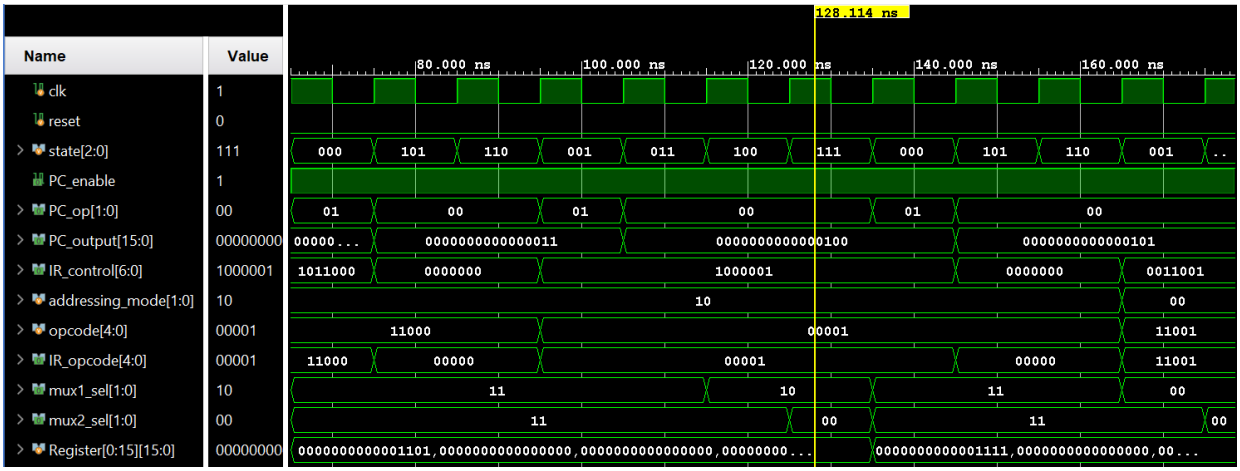


Figure 5.2: Immediate ALU

5.1.3 Compare Test

First Storing a value into the register-1 using the instruction:

10 11000 0000 0000 0
00 00000 0000 0110 1

The compare instruction is used to evaluate the relationship between the data stored in a register and a given immediate value.
In this test, the contents of the selected register are fetched and internally passed to the ALU's comparison unit.

The instruction code shown below represents the exact binary format used for performing this compare operation.

10 11000 0000 0000 0
00 00000 0000 0110 1

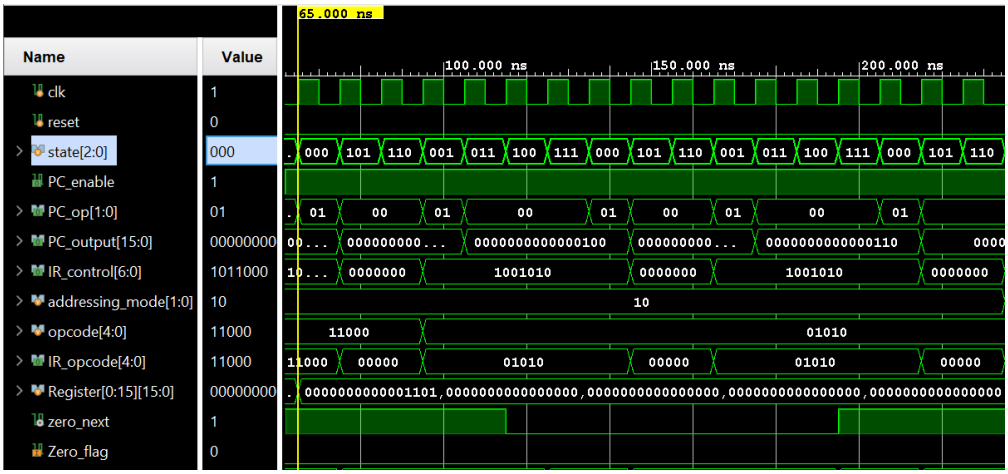


Figure 5.3: Immediate Compare

5.1.4 Jump Test

This test verifies the correct operation of the immediate jump instruction, which updates the Program Counter using a directly encoded address. It ensures that the processor can redirect execution flow accurately without relying on register-stored operands.

10 11011 0000 0000 0
00 00000 0000 0010 0

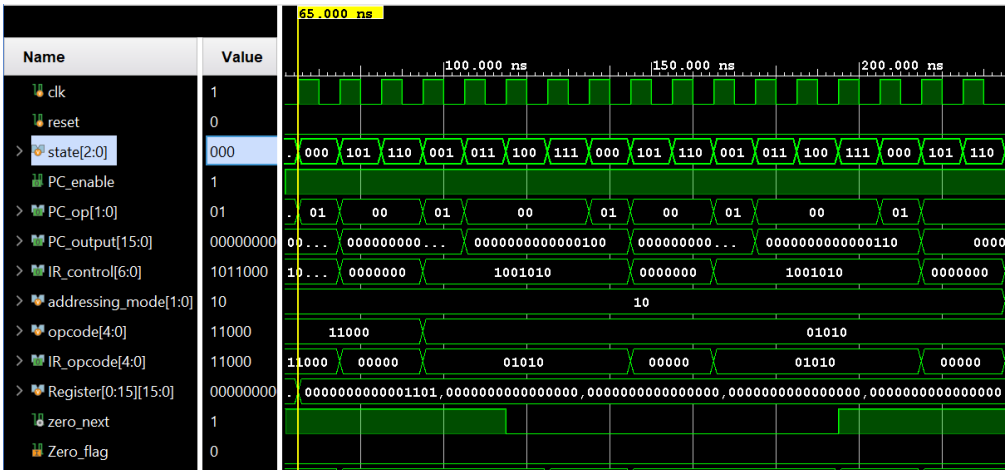


Figure 5.4: Immediate Jump

5.2 Direct Mode Testing

Before performing the Direct Mode tests, it is necessary to preload specific values into the registers for later operations. These values are initialized using the Immediate Load instruction, represented by the following format.

10 11000 0000 0000 0
00 00000 0000 0010 0
and
10 11000 0000 0000 0
00 00000 0000 0010 0

5.2.1 Shift Test

To validate the rotate-right operation, a dedicated test was performed to ensure that the bits circulate correctly within the register.
The Instruction used is as follow:

00 10000 0000 0000 0

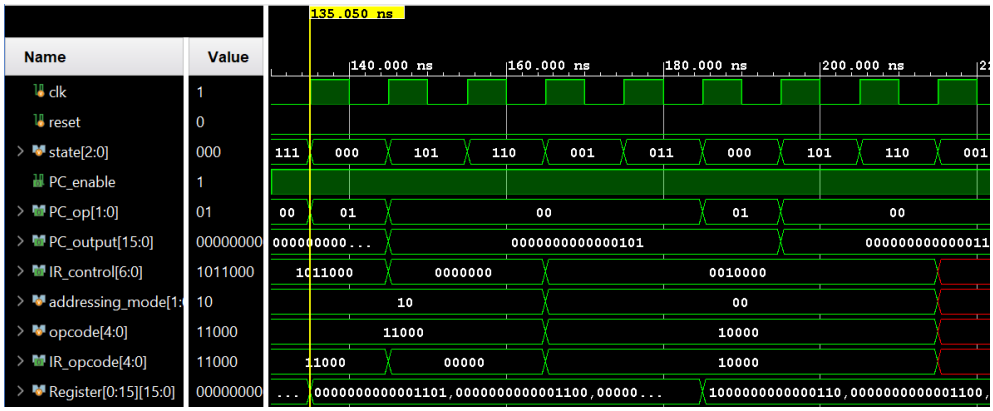


Figure 5.5: Direct Shift

5.2.2 ALU Test

To validate the ALU operation, a dedicated test was performed to ensure that the operation is performed correctly and result is stored back in the register.

The Instruction used is as follow:

00 00001 0000 0001 0

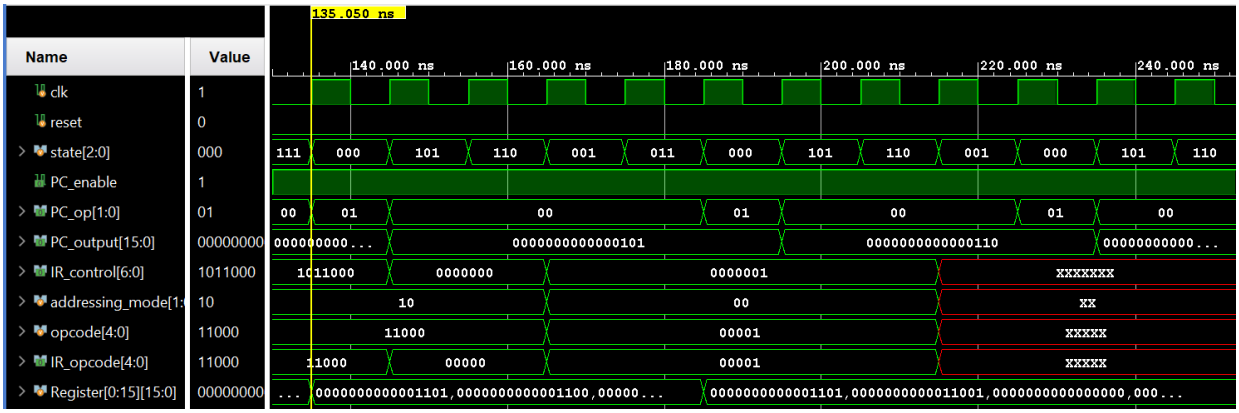


Figure 5.6: Direct ALU

5.2.3 Compare Test

To validate the Compare operation, a dedicated test was performed to ensure that the Zero flag update. The Instruction used is as follow:

00 01010 0001 0000 0

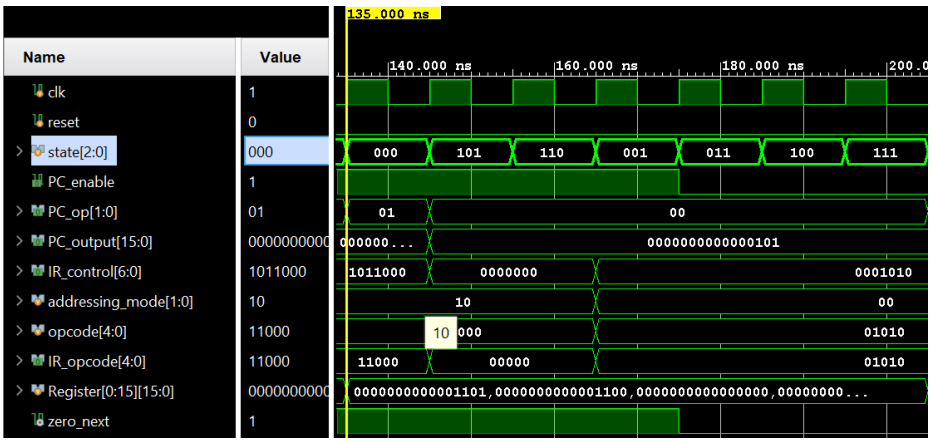


Figure 5.7: Direct Compare

5.2.4 Jump Test

To validate the Jump operation, a dedicated test was performed to ensure that the Jump operation works. The Instruction used is as follow:

00 11011 0000 0000 0

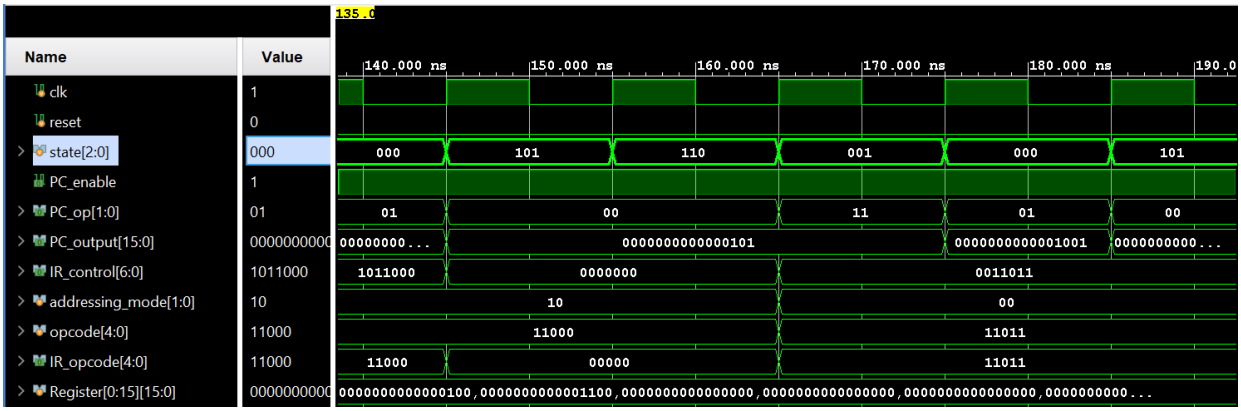


Figure 5.8: Direct Jump

5.2.5 Move Test

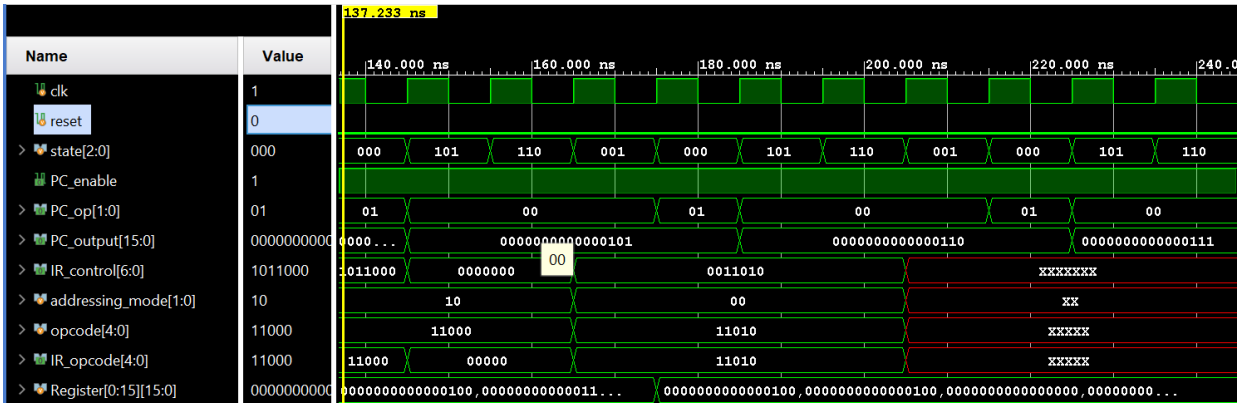


Figure 5.9: Direct Move

To validate the Move operation, a dedicated test was performed to ensure that the move operation works. In which the value of register 1 (0000) is stored in register 2 (0001).

The Instruction used is as follow:

00 11010 0000 0001 0

5.2.6 Store Test

To validate the Store operation, a dedicated test was performed to ensure that the Store operation works. In which the value of register 2 (0001) is stored in data memory index represented by register 1 (0000).
The Instruction used is as follow:

00 11001 0000 0001 0

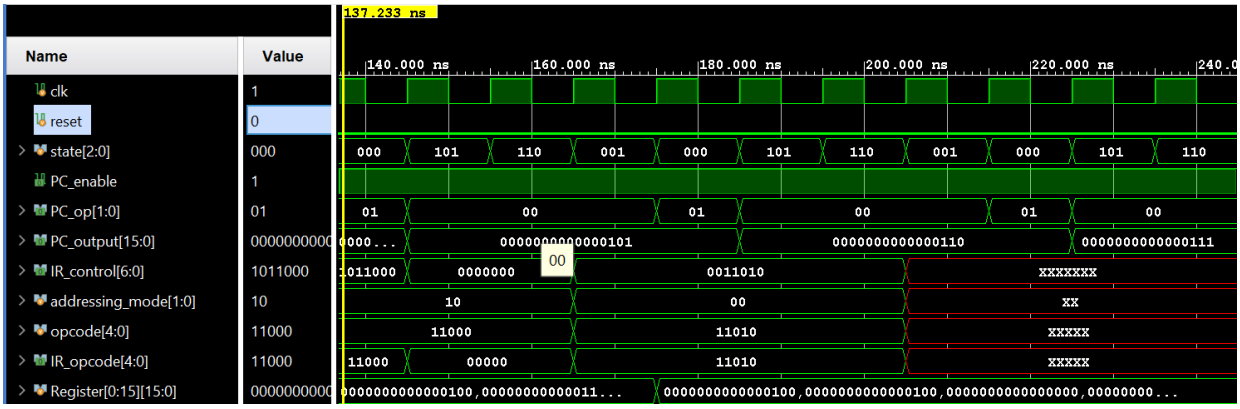


Figure 5.10: Direct Store

5.3 Indirect Mode Testing

Indirect mode is same as Direct mode except in this case one of the value comes from data memory instead of another register, To use the Indirect mode the first two bits of the instruction need to be 01 instead of 00 rest is same as Direct mode.

5.3.1 ALU Test

To validate the ALU operation, a dedicated test was performed to ensure that the operation is performed correctly and the data is stored.

The Instruction used is as follow:

01 00001 0000 0001 0

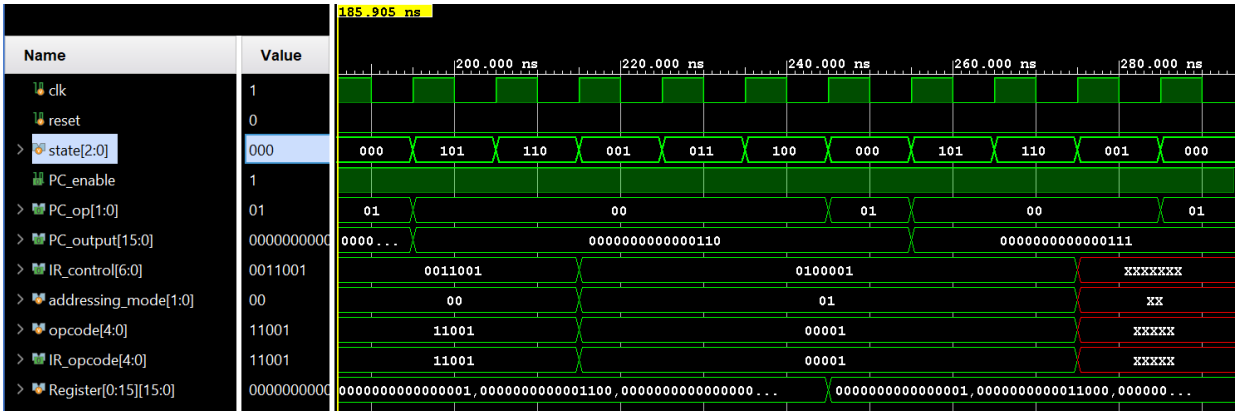


Figure 5.11: Indirect ALU

Acknowledgments

I would like to express my sincere gratitude to my project guide, Dr. Zuber M. Patel, for his continuous guidance, valuable suggestions, and technical insights throughout the design and implementation of this 16-bit RISC processor. His mentorship and encouragement were instrumental in the successful completion of this project.

I am also thankful to the Department of Electronics and Communication Engineering for providing the academic framework and resources necessary to carry out this work. Finally, I gratefully acknowledge the support and encouragement of my family throughout the duration of this project.

Sujay Bhati

May-July 2025
National Institute of Technology Surat

References

[1] Computer System Architecture by M.Morris Mano

[2] Digital System Design using verilog by Charles H. Roth

[3] Advanced Digital Design with the Verilog HDL by Michael D. Ciletti

[4] Verilog HDL:A guide to Digital Design and Synthesis by Samir Palnitkar