A Project Report

on

Implementation of 32-Bit 5-Stage Pipelined RISC-V Processor with Hazard Unit

Submitted in the Partial Fulfilment of the Requirements

for the award of

Bachelor of Technology

in

Electronics and Communication Engineering

by

Gubbala Sai Manoj (20215053)

Ayush Singh Rajawat (20215099)

Aman Kumar (20215041)

Under the guidance of

Dr. V. KRISHNA RAO KANDANVLI
Associate Professor

Department of Electronics and Communication Engineering
Motilal Nehru National Institute of Technology Allahabad
Prayagraj - INDIA

Department of Electronics and Communication Engineering
Motilal Nehru National Institute of Technology Allahabad
Prayagraj - INDIA

DECLARATION

We certify that the work contained in the project titled **Implementation of 32-Bit 5-Stage Pipelined RISC-V Processor with Hazard Unit** submitted by **Gubbala Sai Manoj, Ayush Singh Rajawat, Aman Kumar** in the partial fulfilment of the requirement for the award of Bachelor of Technology in Electronics and Communication Engineering to the Electronics and Communication Engineering Department, Motilal Nehru National Institute of Technology, Allahabad is carried out by us and we have not intentionally violated any professional ethics during the work.

Date:

Place:

Gubbala Sai Manoj (20215053)

Ayush Singh Rajawat (20215099)

Aman Kumar (20215041)

Department of Electronics and Communication Engineering
Motilal Nehru National Institute of Technology Allahabad
Prayagraj – INDIA

CERTIFICATE

This is to certify that the work contained in the project titled **Implementation of 32-Bit 5-Stage Pipelined RISC-V Processor with Hazard Unit**, submitted by **Gubbala Sai Manoj, Ayush Singh Rajawat, Aman Kumar** in the partial fulfillment of the requirement for the award of Bachelor of Technology in Electronics and Communication Engineering to the Electronics and Communication Engineering Department, Motilal Nehru National Institute of Technology, Allahabad, is a Bonafide work of the students carried out under my supervision.

Date:

Place:

Dr. V. KRISHNA RAO KANDANVLI

Associate Professor

ECE Department, MNNIT Allahabad

Acknowledgement

It is indeed with delight that we express our sense of sincere gratitude and best regards to our Final Year Project Work Supervisor, **Dr. V. KRISHNA RAO KANDANVLI**, Department of Electronics and Communication Engineering Department, MNNIT Allahabad for his constant support, motivation, valuable advice, inspiration, supervision, and positive criticism coupled with patience during the entire work titled **“Implementation of 32 Bit 5 – Stage Pipelined RISC-V Processor with Hazard Unit”.** His meticulous guidance, constructive suggestions, elaborate discussions and clarification of our doubts made us more aware and helped us to have a deeper insight into the subject under study. We consider ourselves lucky to have been associated with him throughout the Project work activity that has turned out to be a great process of learning. I also express our sincere gratitude to the Faculty Members and Supporting Staff of the Department of Electronics and Communication Engineering, MNNIT Allahabad for providing us this opportunity that has immensely contributed to an enhancement in our awareness, knowledge and understanding of the technical aspects involved. Last but not the least I bow down in reverence before the Almighty GOD and our parents who have been an eternal source of constant encouragement for us.

Date:

Gubbala Sai Manoj (20215053)

Ayush Singh Rajawat (20215099)

Aman Kumar (20215041)

Abstract

In this project, we explore and compare the performance of Single-Cycle and Multi-Cycle RISC-V processors, both enhanced with 5-stage pipelining. Using Verilog, we design and simulate these processors, focusing on optimizing their performance through pipelining techniques. The RISC-V instruction set architecture (ISA), known for its open and flexible nature, serves as an excellent foundation for this analysis.

The main goal is to evaluate how different processor architectures—single-cycle and multi-cycle—impact execution speed and resource efficiency. The design incorporates key components like the ALU, control unit, memory, and registers, with a particular emphasis on integrating the 5-stage pipeline to improve throughput. To ensure smooth and accurate operation, hazard detection and resolution mechanisms are also implemented.

Ultimately, this project highlights how pipelining and architectural choices shape the efficiency of RISC-V processors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Project Overview

Processors, often referred to as the “brains” of computers, are typically built using either Complex Instruction Set Computing (CISC) or Reduced Instruction Set Computing (RISC) architectures, each offering unique benefits. Enhancing these architectures for better performance remains a key focus in computing. Among these, RISC-V—a modular, open-source Instruction Set Architecture (ISA) developed at UC Berkeley in 2010—has garnered widespread attention. Its simplicity, with only 47 base instructions and modular extensibility, allows it to adapt to specific design needs, offering flexibility in areas like power consumption, code size, and memory usage.

An ISA serves as the crucial bridge between hardware and software, defining how the CPU processes instructions, handles data types, uses registers, and interacts with memory. Traditionally, major companies like Intel, IBM, and ARM have relied on proprietary ISAs, which are costly and require licenses, posing challenges for smaller organizations. In contrast, open ISAs like RISC-V promote innovation by levelling the playing field, enabling smaller companies to compete and delivering affordable, high-performing products to consumers.

RISC-V’s open and free standard allows developers to design custom hardware and adapt software without licensing restrictions. As highlighted by the RISC-V organization, its strength lies not in cutting-edge chip technology but in its open standard, fostering collaboration in software and hardware development.

1.2 Problem Statement

This project focuses on implementing a RISC-V architecture based on the official open ISA standard, leveraging its flexibility and open nature to explore innovative processor design.

The Architecture should be able to do basic RV23I Integer set instructions can be able to perform jump and branch type instructions along with the memory instructions which are the load word and store word respectively.

1.3 Objective

1. To understand basics of RISC-V Architecture
2. To implement 5 stage pipelines of RISC-V Processor in Verilog
3. To implement Hazard Unit of RISC-V Processor in Verilog
4. To verify its functionality of design by performing testbench and simulation

Chapter 2: Literature Review

2.1 Overview

RISC-V is an open-source Instruction Set Architecture (ISA) that has gained significant recognition across academia, industry, and research for its straightforward design, modular structure, and adaptability. Unlike proprietary ISAs like x86 or ARM, RISC-V is freely accessible and has been engineered to support a broad spectrum of computing applications, ranging from low-power embedded devices to high-performance servers.

2.2 RISC-V Milestones and Developments

1. **Key Milestones**
   - The first RISC-V ISA specification was released in 2014.
   - In 2015, the RISC-V Foundation (now RISC-V International) was established to oversee its development and adoption.
   - Major companies like SiFive, NVIDIA, and Western Digital have since embraced RISC-V, driving its increasing industrial adoption.
2. **Academic Research and Comparative Studies**
   - Researchers have extensively studied RISC-V, comparing it to established ISAs like ARM and x86 in terms of performance, power efficiency, and area utilization.
   - These studies often highlight RISC-V's competitive or superior performance in embedded and IoT applications, attributed to its simplicity and modular architecture.
3. **Extensions and Customizations**
   - Custom RISC-V cores are being developed for specialized domains, such as machine learning, cryptography, and multimedia processing.
   - Researchers have explored domain-specific extensions like tensor operations to support AI workloads.
   - RISC-V's simplicity makes it a popular platform for formal verification, ensuring correctness in processor designs.
4. **Educational Use**
   - RISC-V's open-source nature makes it an excellent tool for teaching computer architecture.

    - Tools like Ripes and QEMU enable students and educators to explore the ISA interactively, promoting hands-on learning.

5. **Industrial Adoption**
    - **Embedded Systems**: RISC-V is widely used in IoT and low-power devices.
    - **High-Performance Computing**: Custom RISC-V cores are designed for servers and data center applications.
    - **AI and Machine Learning**: RISC-V has been adapted with vector extensions and accelerators to meet the demands of these fields.

2.3 RISC-V Challenges and Future Developments

1. **Software Ecosystem**
    - While growing, the RISC-V software ecosystem still lags behind mature ISAs like x86 and ARM.
    - Efforts are ongoing to improve support for operating systems, middleware, and development tools.
2. **Standardization of Extensions**
    - Managing and standardizing custom extensions while ensuring interoperability is a significant challenge as the architecture evolves.
3. **Global Competition**
    - Competing with established Instruction Set Architecture like ARM, which benefit from decades of development and a robust ecosystem, remains an uphill task.
4. **Emerging Applications**
    - The future of RISC-V includes potential advancements in quantum computing, neuromorphic systems, and ultra-low-power devices, showcasing its adaptability for cutting-edge technologies.

Chapter 3: Methodology

3.1 Theories

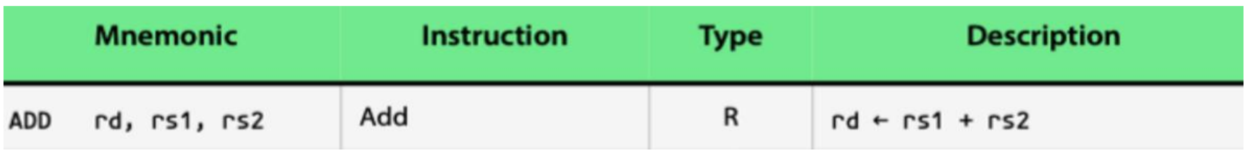
3.1.1 ISA – Instruction Set Architecture

Instruction Set Architecture (ISA) defines the set of instructions that a processor can execute, acting as the interface between software and hardware. It specifies the operations a processor can perform, how data is accessed, and the format of instructions.

There are two main types of ISAs: RISC (Reduced Instruction Set Computing) and CISC (Complex Instruction Set Computing).

- RISC uses a smaller set of simple, highly optimized instructions, designed to execute in a single clock cycle. This approach leads to faster performance and simpler processor design. Modern processors, like those in smartphones and embedded systems, commonly use RISC architectures for their efficiency and low power consumption.
- CISC, on the other hand, uses a larger set of complex instructions that can perform multiple operations in one instruction cycle. This can result in more powerful operations but typically requires more clock cycles to execute, making it less efficient for certain applications.

For this project, we are using RISC architecture, specifically RISC-V, due to its simplicity, flexibility, and open-source nature. RISC-V offers an ideal platform for building efficient and customizable processors, allowing us to focus on performance optimization with a streamlined set of instructions. The below figure 3.1 is an example of a RV32I Integer type instruction.



Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2

Figure 3.1 Example of a RV32I Add Instruction

In this figure, an instruction is shown here **ADD** is the opcode of the instruction meanwhile **rd**, **rs1**, **rs2** are the operand addresses of Register File through which operands are obtained

3.1.2 The Pipeline Architecture (5 Stages)

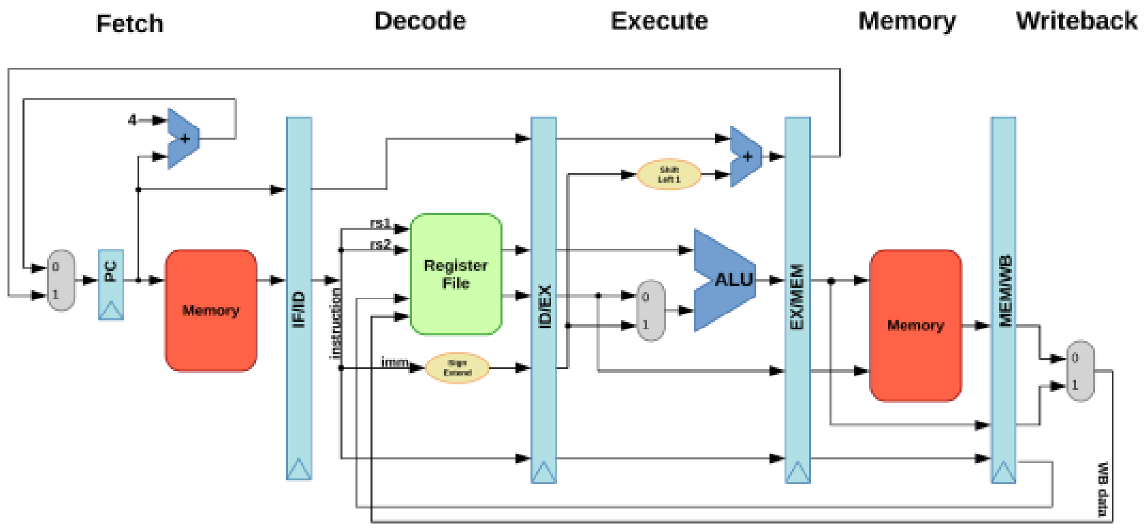


Figure 3.2 The example architecture of a 5-stage pipeline

In this figure 3.2, a pipeline architecture is shown with 5 pipeline segments Fetch, Decode, Execute, Memory access and Writeback respectively. A **5-stage pipeline** is a common design used in modern processors to improve performance by breaking down instruction execution into smaller, more manageable steps. Each step in the pipeline processes a part of an instruction, allowing multiple instructions to be worked on simultaneously at different stages, rather than one after another.

The five stages are typically:

1. **Instruction Fetch (IF)**: The processor retrieves the instruction from memory.
2. **Instruction Decode (ID)**: The fetched instruction is decoded, and the necessary operands are fetched.
3. **Execute (EX)**: The actual operation is performed, such as an arithmetic calculation or address computation.
4. **Memory Access (MEM)**: If the instruction involves memory, data is read from or written to memory.
5. **Write Back (WB)**: The result is written back to the register file, completing the execution of the instruction.

The key advantage of this pipeline is that while one instruction is being executed in one stage, another instruction can be fetched in the next stage, and yet another can be decoded, all happening simultaneously. This parallelism significantly speeds up processing, making the overall execution much faster than if instructions were handled one at a time.

However, managing a pipeline introduces complexities like handling **data hazards** (when instructions depend on each other's results) and **control hazards** (due to branches in the code). To deal with these, additional mechanisms like hazard detection and forwarding are used to ensure the processor operates efficiently. Overall, the 5-stage pipeline is an effective method for increasing throughput and improving processor performance.

3.2 Implementing the RISC-V Architecture

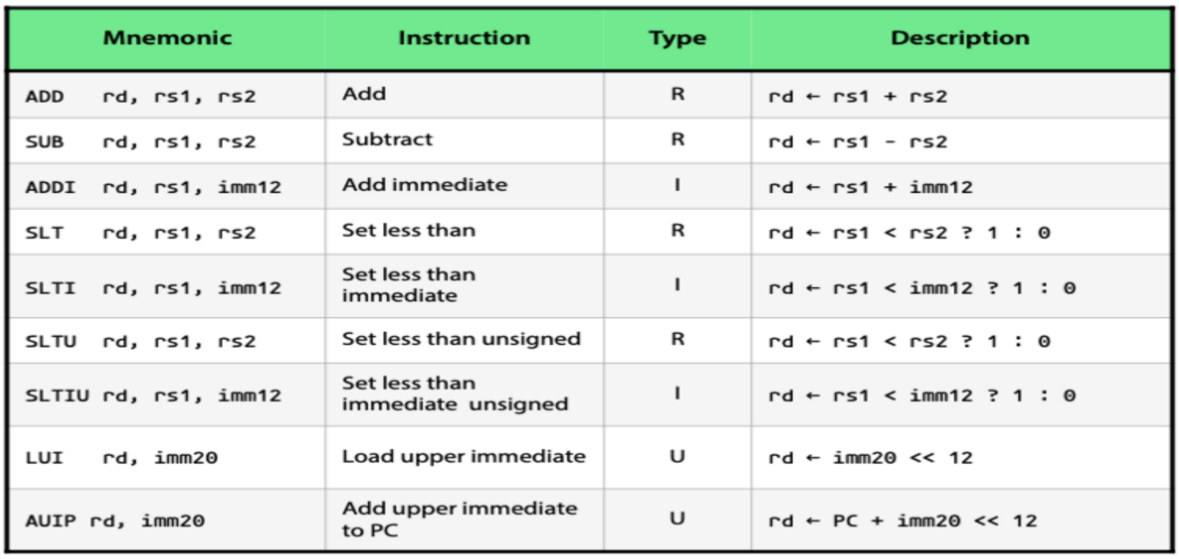
The processor is made up of several key components that work together to execute instructions. These include the Arithmetic Logic Unit, ALU Decoder, Main Decoder, Instruction Memory, Data Memory, Instruction Fetch/Instruction Decode register, Instruction Decode/Execute register, Execute/Instruction Memory register, Instruction Memory/Instruction Write register, Program Counter Multiplexer, Register File, Forwarding Multiplexers.

In the following section, we'll dive deeper into each of these components of the datapath, explaining their functions and how they contribute to the overall operation of the processor.

3.2.1 ALU – Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a critical part of the CPU, designed to handle arithmetic and logical operations on binary data. It takes its input from the pipeline register (Pipeline Register between Decode and Execute stages) and performs the required computations based on control signals provided. The results of these operations are then stored in the output.

In this particular RISC-V processor design, the ALU is set up to handle nine specific instructions. The below table 3.1 shows the all the operations and R Type instructions that the ALU can perform.



Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Table 3.1 The R-Type instructions of RV32I

3.2.2 Decoder for the ALU

The Arithmetic Logic Unit Decoder plays a vital role in interpreting instructions for the ALU. It works with the Main Decoder Unit by receiving signals that help identify the type of operation the ALU needs to execute.

To decode an instruction, aludec combines four inputs: ALU Op, funct3, funct7b5, and opb5. Here's what each represents in the RISC-V instruction format:

- **funct7b5**: 5th index bit of bits [31:25] of the instruction. Instruction [30].
- **funct3**: Refers to bits [14:12].
- **opb5**: Refers to bits [6:0].

These fields provide essential information about the operation's type and specifics. For instance, to differentiate between the ADD and SUB instructions in the R-type format, the subtract signal is derived by performing a logical **AND** operation between funct7b5 and opb5. This distinction is crucial, as both instructions share the same funct3 value but differ in their funct7b5 values. The decoder outputs for all the R-Type instructions is given below in the table 3.2.

ALU Op	*Subtract signal*	*funct3*	Alu decode	Operation
2'b10	0	3'b000	4'b0000	AND
2'b10	1	3'b000	4'b0001	SUB
2'b10	0	3'b111	4'b0010	AND
2'b10	0	3'b110	4'b0011	OR
2'b10	0	3'b001	4'b0100	SLLI
2'b10	0	3'b011	4'b0101	SLTI
2'b10	0	3'b100	4'b0110	XOR
2'b10	0	3'b101	4'b0111	SHR
2'b10	0	3'b101	4'b1000	SLTU
2'b10	0	3'b101	4'b1111	SHL

Table 3.2 The decoder outputs

3.2.3 The Main Decoder of the controller

The Main Decoder is a key component in the RISC-V processor, responsible for interpreting the 7-bit opcode (Instruction [6:0]) to generate control signals. These control signals guide the components, determining how data flows and which operations are performed.

The control signals produced by maindecoder include:

1. **RegWrite**: Enables writing data to a register.
2. **ImmSrc**: Specifies the type of immediate value to be used, based on the instruction format (e.g., I-type, S-type, etc.).
3. **ALUSrcA**: Selects the source for the first operand of the ALU.
4. **ALUSrcB**: Selects the source for the second operand of the ALU (e.g., a register value or an immediate).
5. **MemWrite**: Controls whether data is written to memory.
6. **ResultSrc**: Chooses the source of data to be written back to a register (e.g., ALU result, memory data, etc.).
7. **Branch**: Determines if a branch operation should be performed.
8. **ALUOp**: Indicates the type of ALU operation to be performed, serving as input to the aludec.
9. **Jump**: Specifies if a jump operation is needed.

In the given table 3.3, The control signal values for different types if instructions are shown below. For R-type instructions immediate value is not used and hence the xxx value. Which means a don't care signal.

Control Signal	Instruction					
	L-Type (lw)	S-Type (sw)	R-Type	B-Type	I-Type	J-Type (jal)
RegWrite	1	0	1	0	1	1
ImmSrc	000	001	xxx	010	000	011
ALUSrcA	0	0	0	0	0	0
ALUSrcB	01	01	00	00	01	00
MemWrite	0	1	0	0	00	0
ResultSrc	01	00	00	00	000	10
Branch	0	0	0	1	0	0
ALUOp	00	00	10	01	10	00
Jump	0	0	0	0	0	1

Table 3.3 The different control signals for different instructions

3.2.4 Data Memory

In computer architecture, **data memory** is an essential part of the system used to temporarily store and retrieve data during processing. It plays a critical role in supporting the ALU by holding data that the ALU processes or generates.

Inputs of the Data Memory Module:

1. **write_enable**: A control signal that determines whether the data memory is writable. When write_enable is active (e.g., set to 1), data can be written to the specified address. If inactive, the memory is read-only during that cycle.
2. **data_address**: Specifies the memory location where data will be written to or read from. This address is typically the result of an ALU operation, which calculates the exact location in memory.
3. **write_data**: The actual data to be written into memory. This data usually comes from the **register file** and represents the value that needs to be stored.

Functionality:

- During a **write operation**, if write_enable is set, the write_data is written into the memory location specified by data_address.
- During a **read operation**, when write_enable is not active, the module retrieves the data stored at the specified data_address.

This mechanism ensures efficient and controlled access to data memory, allowing the processor to perform read and write operations as dictated by the instruction flow.

3.2.5 Instruction Memory

In computer architecture, **instruction memory** is a critical component that stores the program's instructions, ensuring the CPU has access to the sequence of operations it needs to perform. This memory is designed specifically for read-only access during program execution.

Functionality of Instruction Memory:

1. **Storage**:
    - It holds the program's instructions in a 32-bit instruction format.
    - Instructions are stored in a memory array, often referred to as the **RAM array**, though it's typically read-only during execution (e.g., ROM or flash memory).
2. **Instruction Fetch**:

- The **Program Counter Fetch (PCF)** signal specifies the memory address of the instruction to be retrieved.
- Since each instruction is 4 bytes (32 bits), the PCF is incremented by 4 after each fetch, pointing to the address of the next instruction in sequence.

3. **Sequence**:
   - During execution, the CPU fetches an instruction from the address provided by the PCF.
   - The fetched instruction is then sent to the decode stage for further processing.

This systematic fetching process ensures that the program's instructions are executed in the correct order unless control flow instructions (e.g., jumps or branches) alter the sequence. By providing a structured and efficient way to retrieve instructions, the **instruction memory** is foundational to the CPU's operation.

3.2.6 The Pipelining Blocks in the architecture

The pipelining the RISC-V Architecture is for increase in the parallel processing of the instructions which in return increases the trough-put of the processor. To implement a 5-stage pipeline, we need 4 Register segments.

- **Program Counter –** Pipeline between Fetch and Decode stages
- **ID IEX** – Pipeline between Decode and Execute stages
- **IEX IMEM –** Pipeline between Execute and Memory stages
- **IMEM IW –** Pipeline between Memory and Writeback stages

Basing on the type of the instruction the stages between them are used

- **Fetch Stage –** All the instructions use this stage
- **Decode Stage –** All the instructions use this stage
- **Execute Stage –** All the instructions use this stage
- **Memory Access stage –** Only Load and Store Word uses this stage
- **Write Back Stage –** Except Load, Store and Branch most of all uses this stage

These pipeline registers are nothing but registers for temporary storage of data for one clock cycle

3.2.7 The Result Mux (Write Data Selection)

The ALU is designed to perform both arithmetic operations, such as addition (A + B), and logical operations, such as equality comparison (A = B). The output of the ALU can serve different purposes depending on the instruction being executed. It could represent:

1. **A memory address**: Used to access or store data in memory.
2. **An operation result**: Used as a computational result to be written back to the register file.
3. **PC Plus-4 Value:** Used in jump instructions where Next instruction address needs to be stored

Role of the Multiplexer (MUX):

To manage these different outputs efficiently, a **multiplexer** is utilized. The MUX acts as a decision-making switch, selecting one of two inputs based on the control signal ResultSrc:

- **Input 1**: The computational result from the ALU.
- **Input 2**: The memory address derived from the ALU output.

The below table 3.4 shows the different values the registers are getting written by, for R,I-Type instruction we need to store ALU Result where as in Load Instruction we need to store Read Data and in Jump instructions we need to store Program counter value.

ResultSrc	output
00	*ALU Result*
01	*Read Data*
10	*PC Plus 4*

Table 3.4 The corresponding selection to output in the result mux

3.2.8 Program Counter Multiplexer

The Program Counter does work for keeping track of the current instruction being executed and determining the address of the next instruction.

In standard execution, the PC increments by a fixed value of 4 for each clock cycle. This increment corresponds to the size of a single 32-bit instruction in memory, ensuring the PC always points to the address of the next sequential instruction.

The PC can be modified by a jump signal (jump) from the control unit. When specific conditions are met (e.g., branch or jump instructions), the control unit instructs the PC to load a new value, the jump address, instead of incrementing by 4.

Role of pc_mux :

A multiplexer is employed to decide the next value of the PC:

- PCPlus4F—the address of the next sequential instruction.

- JumpTargetE—the target address for a jump instruction.

The selection between these two inputs is controlled by the PCSrcE signal:

1. If PCSrcE is high, the pc_mux selects JumpTargetE, and the PC will point to the jump address in the next clock cycle.
2. If PCSrcE is low, the pc_mux selects PCPlus4F, and the PC will continue with the sequential instruction flow.

This mechanism ensures efficient program flow control, allowing the processor to handle both normal sequential execution and deviations like jumps or branches.

3.2.9 Reg File or Register File

The register file (regfile) in a CPU is essential for storing and manipulating data during program execution. It acts as a high-speed storage unit that contains a set of registers, each capable of holding a fixed width of data. The RegWrite control signal is responsible for managing the write operation on the regfile. When RegWrite is activated, the data from WriteData is written into the regfile. Additionally, Instruction [19:15] and Instruction [24:20] function as inputs from the pipeline register (IF/ID) and outputs to the pipeline register (ID/Iex) for the ALU during the execution stage.

3.2.10 Hazard Detection using Hazard Unit

The Hazard Unit is a part of a CPU's architecture that plays a crucial role in identifying and managing hazards that may arise during instruction execution. Hazards are situations where the sequential execution of instructions could result in incorrect or unintended outcomes due to dependencies or conflicts among them. The hazard unit identifies these issues and implements suitable measures to reduce their impact. This allows the dependencies of write data to start from a pipeline register instead of waiting for the WB stage to update the register file. As a result, the necessary data is available in time for subsequent instructions, with the pipeline registers storing the data for forwarding. There are two main strategies to address hazards: forwarding and stalling.

In a CPU pipeline, forwarding and stalling are methods used to handle hazards, which arise from conflicts or dependencies between instructions. These hazards can interrupt the flow of data and their execution. The hazard unit plays a crucial role in identifying and resolving these issues to maintain efficient processor operation.

Forwarding (also referred to as Data Forwarding or Bypassing):

Forwarding is a strategy employed to address data hazards, which occur when an instruction relies on the result of a prior instruction that hasn't finished executing.

Instead of waiting for the instruction to complete and write its result back to the register file, the result is sent directly to the next instruction that requires it.

How Forwarding Works:

When an instruction generates a result that a following instruction needs, the result is forwarded straight from the output of one pipeline stage to the input of another, eliminating the need to access the register file.

For instance, if an instruction adds two numbers and the result is needed by the next instruction, instead waiting for the ALU result to be written back to the register file, the result is forwarded when needed.

Where Forwarding Occurs:

Forwarding can take place between different stages of the pipeline, such as:

EX-to-EX forwarding refers to the process of passing data directly from the execution stage of one instruction to the execution stage of the next instruction.

MEM-to-EX forwarding, on the other hand, involves sending data from the memory stage to the execution stage.

Stalling:

Stalling is another technique used to manage hazards, especially when forwarding cannot resolve a dependency. It entails pausing the pipeline for one or more cycles to give time for the necessary data to become available. A stall introduces a delay, allowing the instruction that requires the data to wait until it is ready.

How stalling works: When the hazard unit detects that a required value is not yet available (for instance, because a prior instruction is still in progress), it introduces a stall cycle. During this cycle, the instruction remains on hold.

Chapter 4: Implementation

4.1 Development Process

RISC-V is an open standard instruction set architecture (ISA) that is becoming increasingly popular due to its simplicity, modularity, and flexibility. Unlike proprietary ISAs, RISC-V is freely accessible for academic research and industrial applications, making it a favored option for developing modern processors. This project aims to design and implement a RISC-V processor that can execute a subset of the RISC-V instruction set.

The processor includes essential components such as an Arithmetic Logic Unit (ALU), Control Unit, Register File, and Memory. It incorporates pipelining to improve performance and utilizes tools like Verilog for coding, simulation tools for debugging, and synthesis tools for visualization. By adhering to a systematic development process, the project seeks to create a functional and efficient design for a RISC-V processor.

3. **Requirement Gathering**
   The initial step in this RISC-V project was to clearly outline the objectives and requirements.
   The primary goals included:
   Creating a functional RISC-V processor with key components like the ALU, Control Unit, Register File, and Memory. Supporting a specific set of RISC-V instructions for arithmetic, logical, and control flow operations. Ensuring compatibility with simulation and visualization tools to validate both functionality and performance.

   Tools identified during this phase included:
   Icarus Verilog for coding and simulation.GTKWave for waveform analysis.Visual Studio Code for writing and managing Verilog code.Xilinx ISE for synthesizing and visualizing the RTL (Register Transfer Level).

4. **Design Phase**
   During the design phase, the architecture and components of the RISC-V processor were meticulously planned.
   Key tasks involved:
   Defining the pipeline stages: Instruction Fetch, Decode, Execute, Memory Access, and Write Back. Designing the ALU to perform arithmetic and logical operations. Creating the Control Unit to manage instruction flow and address hazards.
   Specifying memory components: Instruction Memory and Data Memory for program execution.

5. **Testing Phase**
   Testing was a crucial stage to ensure the processor met all functional requirements.
   The following steps were carried out:
   Simulation with Icarus Verilog: The code was simulated to confirm the accuracy of each module and the overall pipeline.
   Waveform Analysis with GTKWave: Signal outputs were examined to identify timing issues and ensure smooth data flow.
   Debugging: Errors were pinpointed and fixed using simulation results and debugging tools. Iterative testing guaranteed that the design conformed to the intended specifications and addressed any hazards or stalls within the pipeline.

6. **Deployment**
   During the deployment phase, the synthesized design was prepared for visualization and future implementation.
   Key tasks included:
   Utilizing Xilinx ISE to synthesize the Verilog code and produce the RTL schematic. RTL design corrects the connection of components. Preparing documentation and reports for future reference or hardware implementation. The project's completion signifies a functional RISC-V processor design that is ready for further enhancement or deployment in hardware environments.

3.3 Testing and Debugging

During the development and testing phases of the RISC-V project, we faced and resolved several challenges

1. Data Hazards in Pipelining:

   Issue: Incorrect data forwarding led to stalls in the pipeline. Resolution: We implemented a hazard detection unit and forwarding logic to address data dependencies

2. ALU Operation Mismatch:
   Issue: Errors in ALU operations were caused by incorrect instruction decoding. Resolution: We refined the ALU decoder logic to ensure it aligned with the RISC-V specification.

3. Simulation Timing Errors:
   Issue: We encountered timing issues in signal propagation during simulation. Resolution: We adjusted the clock cycle timing and improved the signal propagation paths. These solutions played a crucial role in the successful

implementation of the RISC-V processor, ensuring both robust performance and accuracy.

Chapter 5: Results and Analysis

5.1 The final implemented design

The final architecture that is implemented is illustrated below in the figure 5.1. It's a schematic of all the blocks involved

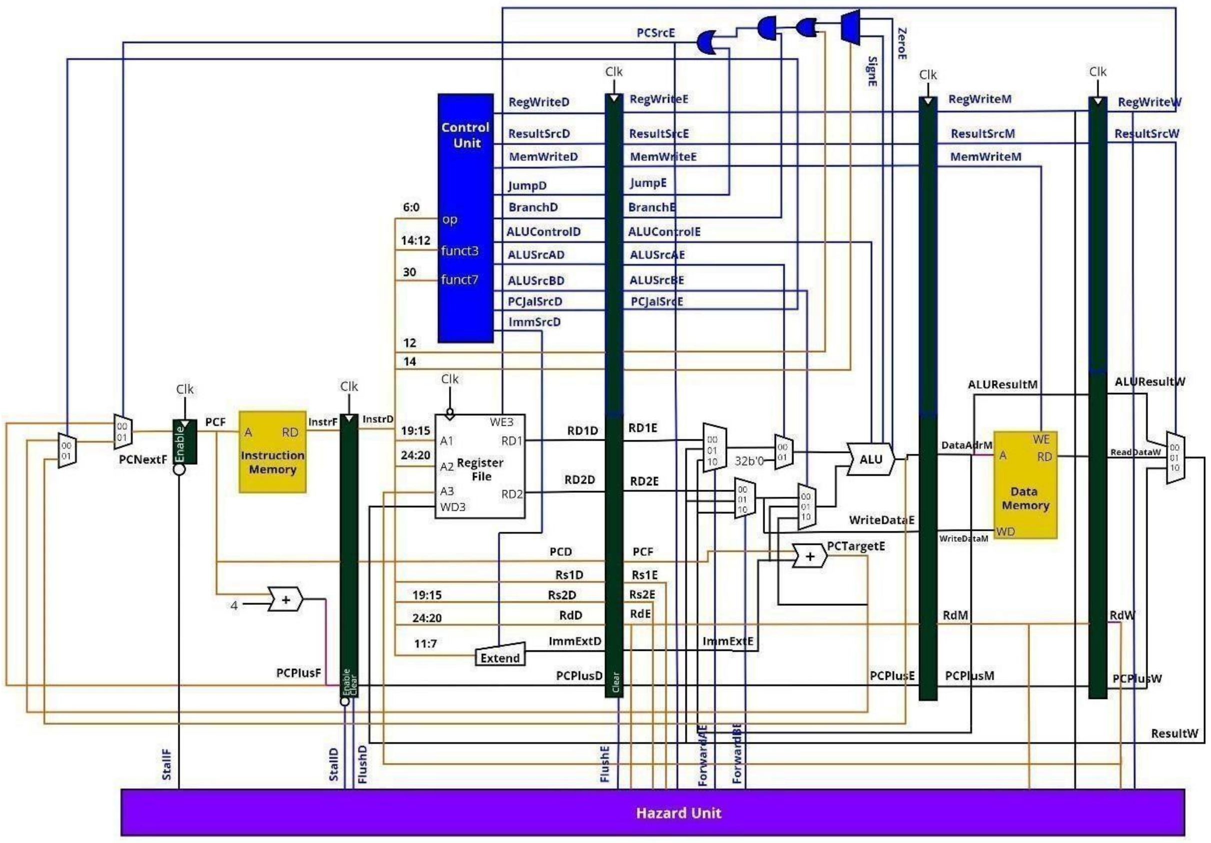


Figure 5.1: The implemented final architecture

The architecture is implemented using the module hierarchy with Instruction memory and Data memory being added the last because they are not added into the central processing unit. And the implementation is divided into 2 blocks basing on the functionality Datapath and Control unit.

5.2 Testbenches and Verifications:

Waveforms were produced from the simulations, and the results were thoroughly analysed and discussed. Additionally, these results were utilized to confirm the functionality of each design module. In the simulation, the functionality of each module was assessed by executing a series of test sequences with various instructions. The test sequences were incorporated into the instruction memory as a program.

The first 3 instructions that were added into the instruction memory are shown in the table 5.2.

Instruction	Description	ALU Result
0x00500113	R2 ← R0 + 5	5
0x00C00193	R3 ← R0 + C	C
0xFF718393	R7 ← R3 - 9	3

Figure 5.2 Instructions and their ALU Results

and these ALU result is verified through the simulations waveforms through the vcd file which is obtained through the testbench. The above 3 instructions should provide a result which is specified in the above table. And these are now verified through the gtkwave waveform simulation and the simulations are given below in figure 5.3.

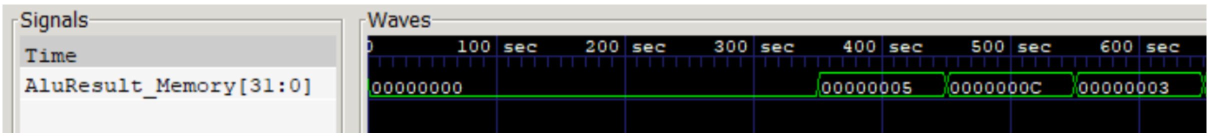


Figure 5.3 The ALU Result for given instructions

As there are no branch instructions involved in these 3 instructions the program counters value should go on by incrementing 4 from its previous value. The simulations of the program counter is given in below in the figure 5.4.

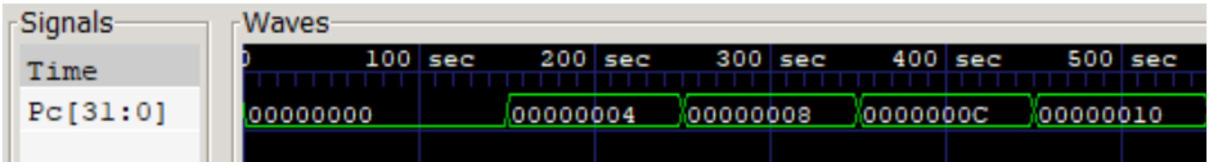


Figure 5.4 The Program Counter (PC) simulation

The instructions should come out from the instruction memory with the given Program Counter address that is provided. The instructions obtained from the instruction memory is shown below in the figure 5.5.

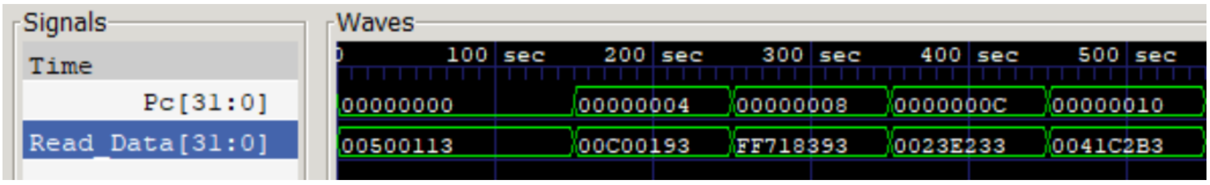


Figure 5.5 The Instructions from the Instruction Memory

Since all the 3 instructions are R-Type instructions and these are writing into the registers at R2, R3, R7 respectively the write address (rd) and the alu result are verified at the same clock cycle and in the given figure 5.6 rd corresponds to the signal Rd_W.

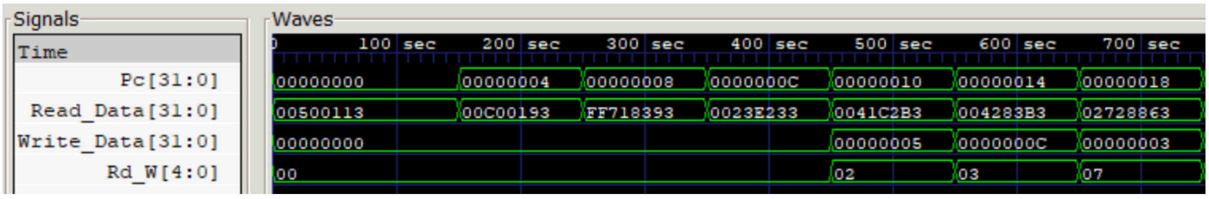


Figure 5.6 The written address for given instructions

The additional instructions and all the results can be verified using the simulation waveforms in the gtkwave and the results for some instructions are given in the figure 5.7 below

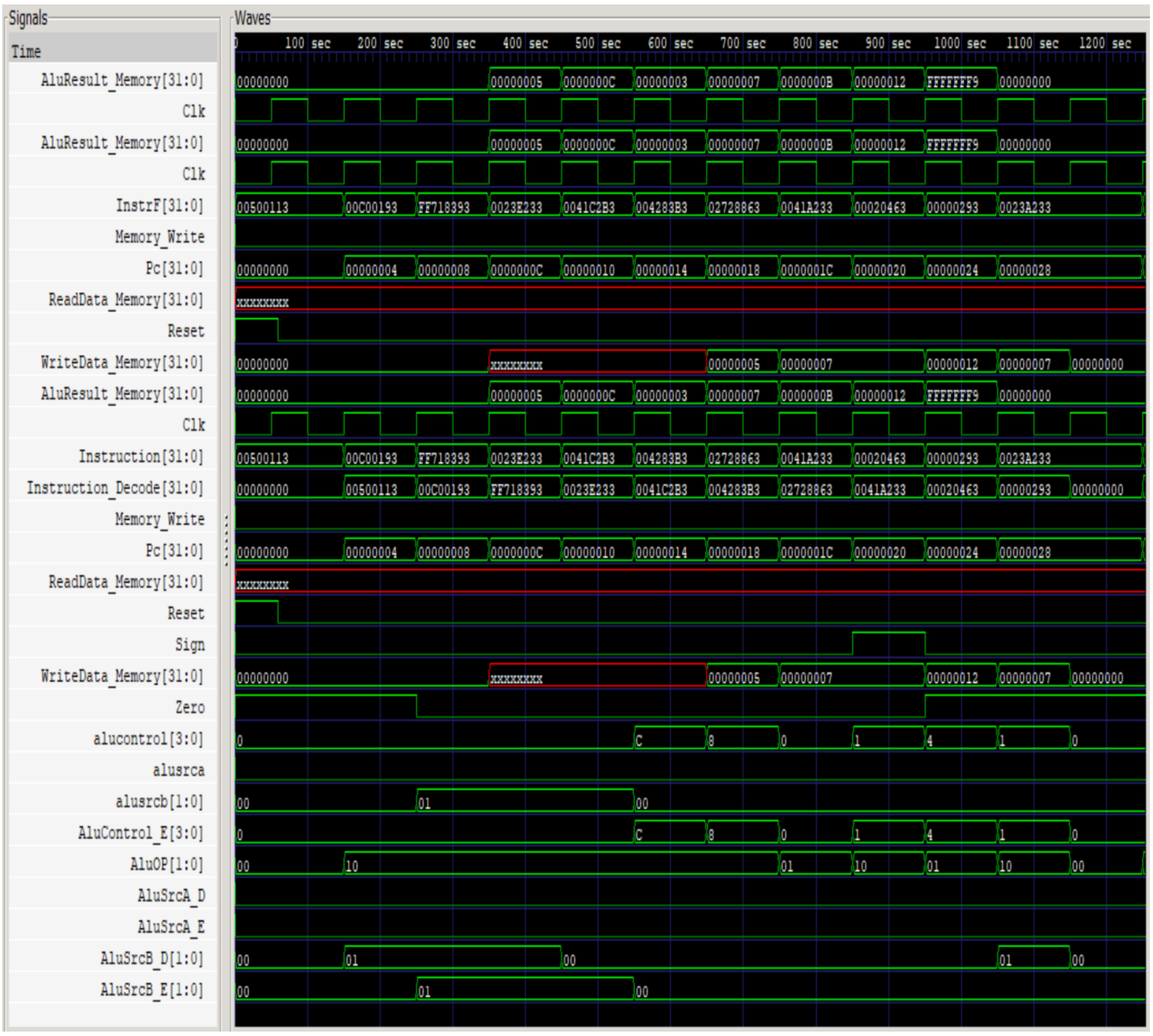
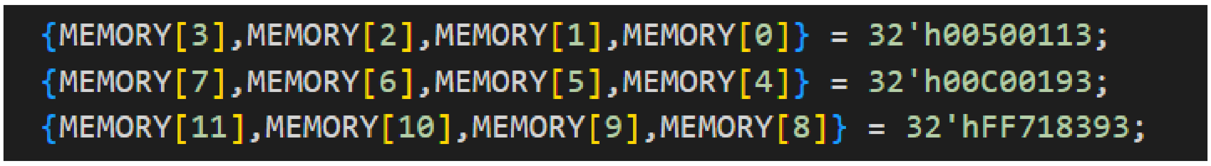
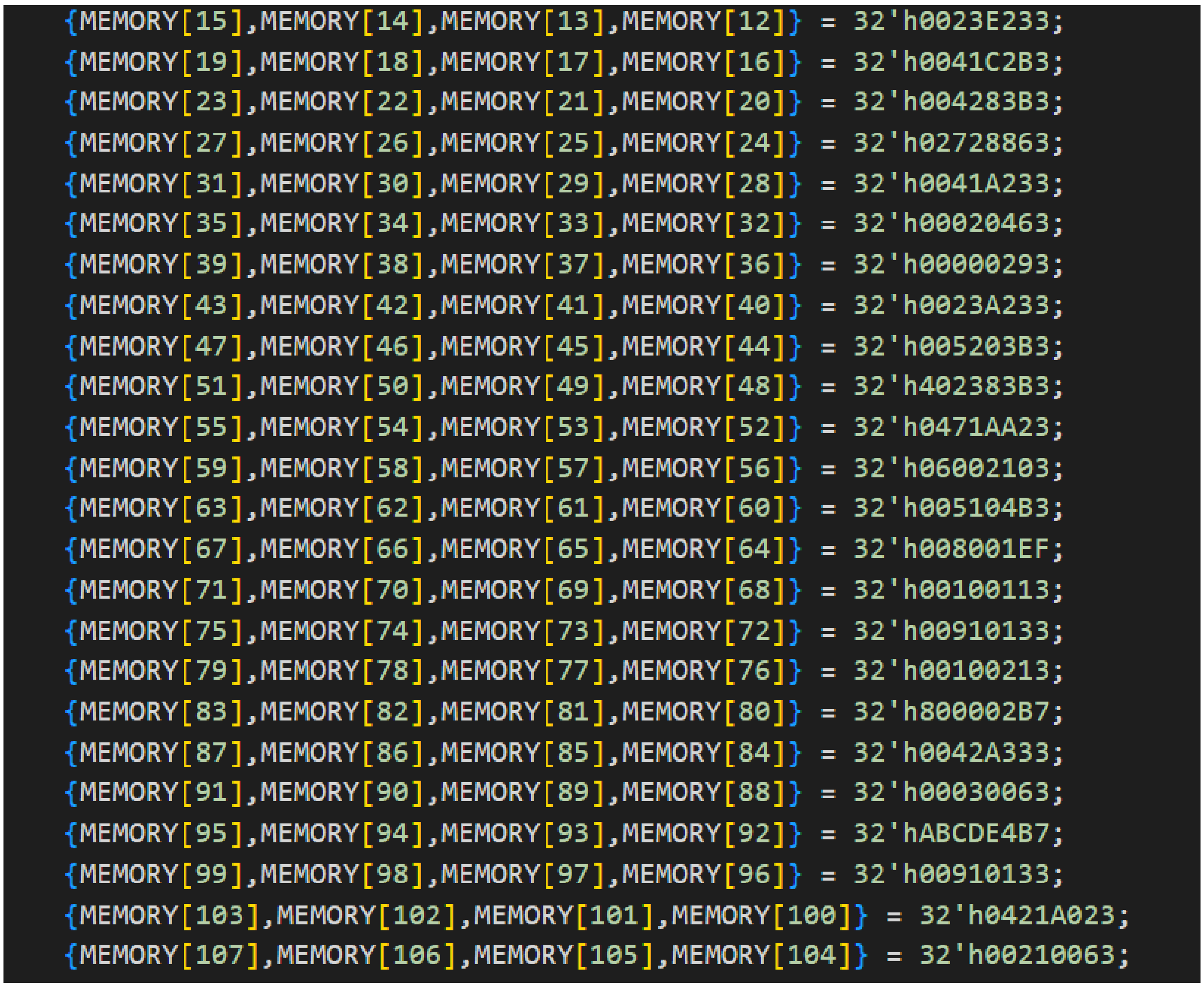


Figure 5.7 The different output waveforms for different instructions

The total instruction memory used for the verification is given in the image below the above 3 instructions are the first 3 instructions that are provided into the instruction memory. This figure 5.8 shows the Verilog initialisation of instruction memory



```
{MEMORY[3],MEMORY[2],MEMORY[1],MEMORY[0]} = 32'h00500113;
{MEMORY[7],MEMORY[6],MEMORY[5],MEMORY[4]} = 32'h00C00193;
{MEMORY[11],MEMORY[10],MEMORY[9],MEMORY[8]} = 32'hFF718393;
```



```
    {MEMORY[15],MEMORY[14],MEMORY[13],MEMORY[12]} = 32'h0023E233;
    {MEMORY[19],MEMORY[18],MEMORY[17],MEMORY[16]} = 32'h0041C2B3;
    {MEMORY[23],MEMORY[22],MEMORY[21],MEMORY[20]} = 32'h004283B3;
    {MEMORY[27],MEMORY[26],MEMORY[25],MEMORY[24]} = 32'h02728863;
    {MEMORY[31],MEMORY[30],MEMORY[29],MEMORY[28]} = 32'h0041A233;
    {MEMORY[35],MEMORY[34],MEMORY[33],MEMORY[32]} = 32'h00020463;
    {MEMORY[39],MEMORY[38],MEMORY[37],MEMORY[36]} = 32'h00000293;
    {MEMORY[43],MEMORY[42],MEMORY[41],MEMORY[40]} = 32'h0023A233;
    {MEMORY[47],MEMORY[46],MEMORY[45],MEMORY[44]} = 32'h005203B3;
    {MEMORY[51],MEMORY[50],MEMORY[49],MEMORY[48]} = 32'h402383B3;
    {MEMORY[55],MEMORY[54],MEMORY[53],MEMORY[52]} = 32'h0471AA23;
    {MEMORY[59],MEMORY[58],MEMORY[57],MEMORY[56]} = 32'h06002103;
    {MEMORY[63],MEMORY[62],MEMORY[61],MEMORY[60]} = 32'h005104B3;
    {MEMORY[67],MEMORY[66],MEMORY[65],MEMORY[64]} = 32'h008001EF;
    {MEMORY[71],MEMORY[70],MEMORY[69],MEMORY[68]} = 32'h00100113;
    {MEMORY[75],MEMORY[74],MEMORY[73],MEMORY[72]} = 32'h00910133;
    {MEMORY[79],MEMORY[78],MEMORY[77],MEMORY[76]} = 32'h00100213;
    {MEMORY[83],MEMORY[82],MEMORY[81],MEMORY[80]} = 32'h800002B7;
    {MEMORY[87],MEMORY[86],MEMORY[85],MEMORY[84]} = 32'h0042A333;
    {MEMORY[91],MEMORY[90],MEMORY[89],MEMORY[88]} = 32'h00030063;
    {MEMORY[95],MEMORY[94],MEMORY[93],MEMORY[92]} = 32'hABCDE4B7;
    {MEMORY[99],MEMORY[98],MEMORY[97],MEMORY[96]} = 32'h00910133;
    {MEMORY[103],MEMORY[102],MEMORY[101],MEMORY[100]} = 32'h0421A023;
    {MEMORY[107],MEMORY[106],MEMORY[105],MEMORY[104]} = 32'h00210063;
```

Figure 5.8 Program in the instruction memory

5.3 The RTL (Register Transfer Level) Design

This processor is divided into 2 parts which are CPU and Memory respectively the memory is divided into 2 types Instruction Memory and Data Memory respectively and in the CPU part there are divisions which are data path, control unit, and hazard unit respectively

This is implemented by using the module level hierarchy in the XILINX ISE software through which the RTL or the Register Transfer level of the CPU is generated

The RTL verification images are given below

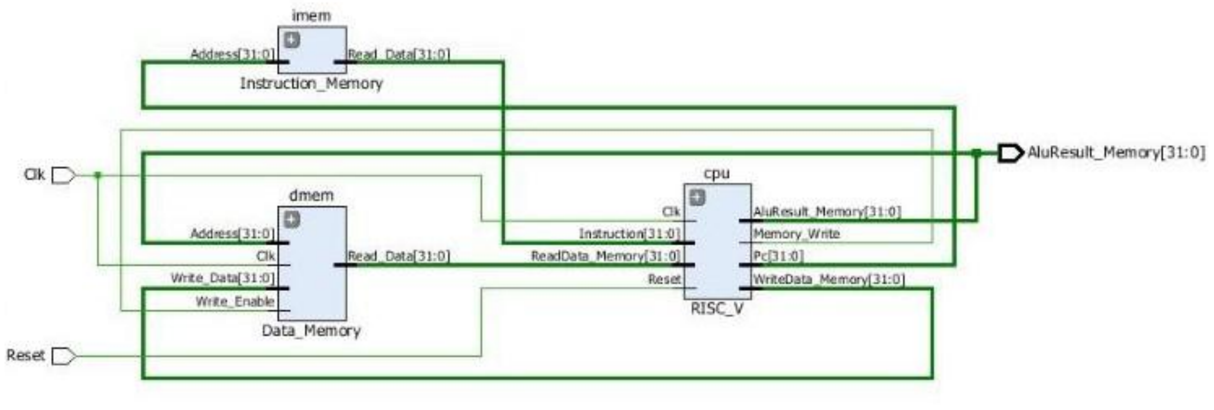


Figure 5.9 The top block with CPU and Memory

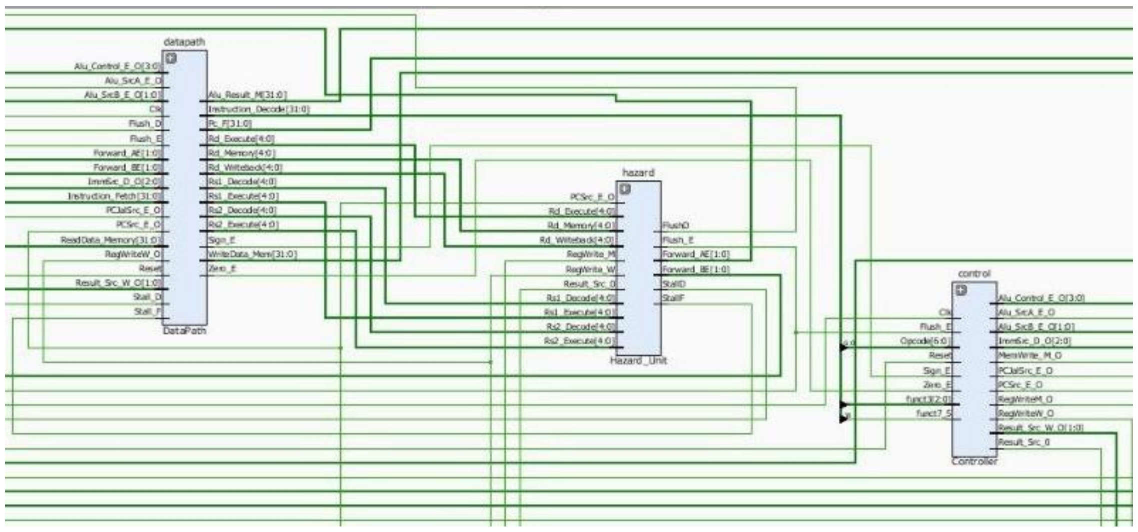


Figure 5.10 The RTL of the CPU with Datapath, Control and Hazard Unit

Chapter 6: Conclusion

The creation of the RISC-V 32-bit processor featuring a 5-stage pipeline and a hazard unit represents a major advancement in contemporary processor design. This project incorporates key pipeline stages—instruction fetch, decode, execute, memory access, and write-back—ensuring a smooth flow of instructions and enhanced performance. The addition of a hazard detection unit effectively manages both data and control hazards, allowing the processor to operate efficiently while maintaining execution accuracy.

Through extensive testing and validation, the processor showcases strong functionality and the capability to process a wide variety of instructions with minimal delays and disruptions. Its modular and open-source RISC-V architecture emphasizes the design's scalability and flexibility, making it ideal for various applications, ranging from educational purposes to practical computational systems.

This project not only highlights the versatility of RISC-V as an effective instruction set architecture but also offers valuable insights into optimizing pipelines and managing hazards. In summary, it serves as a thorough and practical contribution to the realm of computer architecture and processor development.

Chapter 7: Future works and Implementations

The functionality of these modules was verified by analyzing the waveform generated using GTKWave software. Additionally, a testbench for the top design was created to confirm the overall functionality of the 32-bit, 5-stage pipeline RISC-V processor.

However, there are some limitations in our RISC-V processor that could be improved in the future.

Implementing various extensions to the base integer instruction set, RISC-V has standardized a series of extensions that offer additional functionality beyond the base integer instructions, such as floating-point arithmetic, bit manipulation, vector operations, and cryptography. These extensions can be included or excluded based on the design goals and application requirements.

Enhancing branch prediction accuracy and minimizing branch penalties is also crucial. Branch prediction is a method that helps predict the result of a conditional branch instruction before it gets executed. This enables the processor to fetch and execute instructions from the anticipated branch ahead of time, rather than waiting for the actual branch instruction to be determined. However, if the prediction is incorrect, the processor must flush the pipeline and fetch instructions from the correct branch, resulting in a performance penalty. To enhance branch prediction accuracy and reduce branch penalties, techniques such as static branch prediction, dynamic branch prediction, and branch history tables can be employed.

Exploring different cache architectures and memory hierarchies is another area of focus.

Cache is a small, fast memory that stores frequently accessed data from the main memory. Memory hierarchy refers to a system of multiple levels of memory with varying sizes and speeds. The objective of designing cache architecture and memory hierarchies is to minimize the average memory access time and enhance memory bandwidth. To achieve this, various aspects such as cache size, cache organization, and cache mapping can be investigated.

References

1. D. Bhandarkar and D.W. Clark, “Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware

   Organization,”Proceedings of the 4th Int’l. Conference on ASPLOS, Santa Clara, California, April 8-11, 1991.

2. Kulshreshtha, A., Moudgil, A., Chaurasia, A. and Bhushan, B., 2021, March. Analysis of 16-Bit and 32-Bit RISC Processors. In 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS) (Vol. 1, pp. 1318-1324). IEEE.

3. Khairullah, S.S., 2022, June. Realization of a 16-bit MIPS RISC pipeline processor. In 2022 International Congress on Human-Computer.

4. M. N. Topiwala and N. Saraswathi, "Implementation of a 32-bit MIPS based RISC processor using Cadence," 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, 2014.

5. Islam, S., Chattopadhyay, D., Das, M.K., Neelima, V. and Sarkar, R., 2006, September.Design of High-Speed-Pipelined Execution Unit of 32- bit RISC Processor. In 2006 Annual IEEE India Conference (pp. 1-5). IEEE.

6. S. P. 6. Ritpurkar, M. N. Thakare and G. D. Korde, "Design and simulation of 32-Bit RISC architecture based on MIPS using VHDL," 2015 International Conference on Advanced Computing and Communication Systems, 2015.Interaction, Optimization and Robotic Applications (HORA) (pp. 1-6). IEEE.

7. Al-sudany, S.M., Al-Araji, A.S. and Saeed, B.M., 2021. FPGA-Based Multi-Core MIPS Processor Design. IRAQI JOURNAL OF COMPUTERS, COMMUNICATION, CONTROL & SYSTEMS ENGINEERING, 21(2).

8. Wang, W., Han, J., Cheng, X. and Zeng, X., 2021. An energy-efficient cryptoextension design for RISC-V. Microelectronics Journal, 115, p.105165.