MIPS – ALU PROCESSOR

Submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering Degree in

Electronics and Communication Engineering

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE TECHNOLOGIES

SRIKAKULAM-532401

Presented by:

M. S. K. Chaitanya (S210063)

K. Ajay Varma (S210763)

K. Lalitha Supriya (O210670)

M. Meghana (S210482)

P. Prabhu Varma (S210953)

A. Vijay Kumar (S210172)

Under the Guidance of:

Mr. D. Chanti, M. Tech

Assistant Professor

CERTIFICATE

This is to certify that the project entitled **“MIPS – ALU Processor”** submitted by: M. S. K. Chaitanya – S210063, M. Meghana – S210482, K. Ajay Varma – S210763, K. Lalitha Supriya – O210670, P. Prabhu Varma – S210963, A. Vijay Kumar – S210172, students of **Electronics and Communication Engineering,** has been carried out under my supervision and guidance in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** by **Rajiv Gandhi University of Knowledge Technologies, Srikakulam.** The work embodied in this project report has not been submitted to any other University or Institution for the award of any degree or diploma.

Project Guide:

Mr. D. Chanti , M. Tech

Assistant Professor, Dept. of ECE

RGUKT Srikakulam

Head of the Department:

Mr. T. S. Gagandeep, M. Tech, (PhD)

Assistant Professor, Dept. of ECE

RGUKT Srikakulam

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our guide, **Mr. D. Chanti**, for his continuous support, valuable guidance, and consistent encouragement throughout every stage of this project. His insights, constructive feedback, and motivation played a crucial role in shaping our work and helping us successfully complete this project.

We are also deeply thankful to the **Department of Electronics and Communication Engineering** and the esteemed faculty of **RGUKT Srikakulam** for providing us with the necessary resources, laboratory facilities, and a supportive technical environment. Their commitment to quality education and practical learning greatly contributed to our understanding and execution of this project.

We extend our appreciation to our **classmates and friends** for their collaboration, timely suggestions, and meaningful discussions that helped us overcome several challenges. Their support created a positive and encouraging atmosphere, enabling us to work more efficiently.

Finally, we are grateful to our **families** for their constant encouragement, patience, and moral support, which motivated us to stay focused and dedicated throughout the project.

DECLARATION

We hereby declare that the project report titled **"MIPS – ALU Processor"** is a bona fide record of the original work carried out by us as part of the requirements for the **Bachelor of Technology (B. Tech)** degree. This work has been completed under the valuable guidance and supervision of **Mr. D. Chanti, M. Tech**, Department of Electronics and Communication Engineering.

We further affirm that the work presented in this report is the result of our own efforts and hasn't been submitted, either in part or full, to any other university or institution for the award of any degree, diploma, or certification. All sources of information used in this project have been duly acknowledged and referenced.

We take full responsibility for the authenticity and originality of the contents of this report and declare that any assistance received during the course of this project has been explicitly mentioned.

Signatures:

M. S. K. Chaitanya	- S210063
M. Meghana	- S210482
K. Ajay Varma	- S210763
K. Lalitha Supriya	- O210670
P. Prabhu Varma	- S210953
A. Vijay Kumar	- S210172

Date: ______

Place: RGUKT Srikakulam

ABSTRACT

The design of processors based on RISC architecture has played a vital role in shaping modern computing systems. Among them, the MIPS architecture is widely used in academic and research environments because of its simplicity and clarity. Understanding how processors execute arithmetic and logical operations at the hardware level is essential.

Despite the availability of full-fledged processors, their complexity often makes them unsuitable for beginner-level study and small-scale implementations. A gap exists in creating simplified, easily understandable processor models that allow learners to focus on fundamental concepts.

The objective of this project is to design and implement **MIPS-ALU Processor (Pipelined)** in Verilog HDL, which supports a subset of MIPS instructions to perform basic arithmetic and logical operations.

To achieve this, a modular approach is adopted where the ALU, control unit, and Datapath are designed and interconnected. The processor executes a limited set of MIPS instructions such as addition, subtraction, logical operations and simple load/store, focusing on the fundamental behaviour of processor execution while keeping the design lightweight.

The expected outcome is a functional MIPS-ALU Processor model that demonstrates the execution of core instructions in simulation. This work will contribute as a practical and educational tool for understanding processor design.

Key words: *MIPS Architecture, ALU, Pipelining*

TABLE OF CONTENTS

APPENDIX

ABBREVATIONS

- **CPU** – Central Processing Unit
- **ISA** – Instruction Set Architecture
- **MIPS** – Million Instructions Per Second
- **RISC** – Reduced Instruction Set Computer
- **HDL -** Hardware Description Language
- **ALU** – Arithmetic Logic Unit
- **RTL** – Register Transfer Level
- **FPGA** – Field Programmable Gate Array
- **FSM** – Finite State Machine
- **IF** – Instruction Fetch
- **ID -** Instruction Decode
- **EX** – Execute
- **MEM** – Memory Access
- **WB** – Write Back
- **PC** – Program Counter
- **IR** – Instruction Register
- **NPC** – Next Program Counter
- **Imm** – Immediate Value
- **RR_ALU** – Register-Register ALU Operation
- **RM_ALU** – Register-Immediate ALU Operation
- **LOAD** – Memory Read Operation
- **STORE** – Memory Write Operation
- **BRANCH** – Branch Instruction Type
- **HALT_T** – Halt Type
- **NOP** – No Operation
- **ADD** – Addition
- **SUB** – Subtraction
- **AND_OP** – Bitwise AND
- **OR_OP** – Bitwise OR
- **SLT** – Set Less than
- **MUL** – Multiply
- **LW** – Load Word
- **SW** – Store Word
- **ADDI** – Add Immediate
- **SUBI** – Subtract Immediate
- **SLTI** – Set Less Than Immediate
- **BEQZ** – Branch if Equal to Zero
- **BNEQZ** – Branch if Not Equal to Zero
- **HLT** – Halt
- **IDE** – Integrated Development Environment

- **GUI** – Graphical User Interface
- **IF/ID** – Pipeline Latch between IF and ID
- **ID/EX -** Pipeline Latch between ID and EX
- **EX/MEM -** Pipeline Latch between EX and MEM
- **MEM/WB -** Pipeline Latch between MEM and WB
- **Op1, Op2** – Inputs to ALU from Pipeline
- **LMD** – Load Memory Data
- **TB** – Testbench
- **UUT** – Unit Under Test
- **CLK** – Clock
- **posedge** – Positive Edge of Clock
- **$monitor** – Runtime monitoring System Task
- **$display** – Print to Console
- **$stop** – Stop Simulation

CHAPTER - 1
INTRODUCTION

1.1 Overview of this Project:

The proposed project focuses on the design and implementation of a custom **MIPS-based ALU Processor** that operates using a five-stage pipelined architecture. Unlike the traditional MIPS processor, which is characterized by its standard 32-bit, 32-register architecture, this project introduces a unique modification by adopting a **20×32 register file**. This tailored design not only reduces hardware resource utilization but also provides a practical and efficient alternative suitable for academic experimentation, FPGA-based testing, and compact embedded processor applications.

At its core, the processor is built around the principles of **RISC** (Reduced Instruction Set Computing), following the classical MIPS philosophy of simplicity, uniformity, and high-speed execution. The processor supports a carefully chosen subset of instructions sufficient for demonstrating pipeline behaviour, ALU operations, control flow, and memory interaction. These instructions collectively allow the processor to perform arithmetic, logical, data transfer, and basic branch operations — establishing it as a fully functional, educational-grade CPU model.

A major emphasis of this project is placed on the integration of **pipelining**, a widely used technique in modern microprocessors to increase throughput by overlapping multiple instructions during execution. The architecture incorporates the five fundamental pipeline stages:

- **Instruction Fetch (IF)**
- **Instruction Decode (ID)**
- **Execution (EX)**
- **Memory Access (MEM)**
- **Write Back (WB)**

By structuring the processor in this manner, each stage works concurrently with others, thereby ensuring that one instruction completes in each clock cycle after the pipeline is filled. This dramatically enhances performance compared to a non-pipelined single-cycle processor.

To enable this pipelined functionality, the processor design includes **pipeline latches** (IF/ID, ID/EX, EX/MEM, and MEM/WB) that hold the intermediate values between stages. These registers ensure smooth flow of data even as multiple instructions are processed simultaneously. The ALU, placed in the EX stage, is responsible for performing arithmetic and logical operations, serving as the computational heart of the processor.

Another important aspect of this project is the handling of **pipeline hazards**, which naturally arise in any parallel instruction execution model. The processor incorporates essential hazard mitigation techniques such as **operand forwarding**, **stall insertion**, and **control hazard handling**, allowing accurate execution even when instructions depend on each other. This enhances the processor's reliability and demonstrates real-world CPU behaviour in a practical, manageable way.

The entire processor is implemented in **Verilog HDL**, ensuring that the design is fully synthesizable and compatible with industry-standard simulation tools. The modular coding style — with separate files for the ALU, control unit, register file, multipliers, pipeline registers, and memory modules — makes the design readable, scalable, and easy to modify. Simulation waveforms generated during testing offer clear visual confirmation of the processor's functionality, correctness, and pipeline timing.

Overall, this project provides a comprehensive demonstration of processor design, pipelined execution, Datapath coordination, hardware–software partitioning, and digital system implementation. The work integrates theoretical concepts from computer organization with practical HDL-based realization, resulting in a fully functional pipelined MIPS processor model. Through the design, coding, simulation, and verification stages, the project showcases the complete workflow involved in developing a processor architecture—from conceptual planning to final hardware-level validation. This implementation highlights the intricacies of microarchitecture, instruction-level parallelism, and hazard management, emphasizing the complexity and precision required to build efficient and reliable digital hardware systems.

1.2 What is MIPS Architecture?

The **MIPS architecture** (Microprocessor without Interlocked Pipeline Stages) is one of the most influential and widely studied **RISC** (Reduced Instruction Set Computing) architectures in computer engineering. Originally developed by MIPS Computer Systems, Inc. in the early 1980s, it was designed with the goal of achieving high performance through simplicity, uniformity, and efficient pipelining. Over the decades, MIPS has become a foundational model used in both academic teaching and commercial embedded systems, due to its clean instruction set, predictable execution model, and modular architectural structure.

At its core, the MIPS architecture is characterized by a **load/store design**, meaning that all arithmetic and logical operations are performed only on registers, while memory is accessed exclusively through dedicated load and store instructions. This separation of computation and memory access simplifies the Datapath and control logic, making MIPS ideal for pipelining. In traditional MIPS, a set of **32 general-purpose registers** is used to hold operands and results, but in this project, the architecture is customized to operate on a **20×32 register file**, while still maintaining the fundamental MIPS principles.

The MIPS instruction set is designed such that all instructions are of **fixed 32-bit length** and follow a small number of instruction formats, typically **R-type, I-type, and J-type**. These formats maintain a consistent structure that simplifies instruction decoding and control signal generation. Each instruction format includes clearly defined fields such as opcode, source registers, destination register, shift amount, function code, and immediate values. This uniformity allows hardware designers to create efficient decoding mechanisms and straightforward data paths.

One of the hallmark features of the MIPS architecture is its deep accommodation for pipelining, a technique used to improve CPU throughput by overlapping the execution of multiple instructions. MIPS was intentionally designed without the need for hardware interlocks in early versions, encouraging compilers to schedule instructions to avoid hazards. Although modern implementations include hazard detection, the inherent simplicity of the instruction set still makes pipeline implementation easier compared to more complex CISC architectures. The classical **five-stage pipeline**—Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB)—is directly inspired by the MIPS architecture and remains a standard teaching model in computer architecture textbooks.

Another key aspect of MIPS is its emphasis on **orthogonality**—most instructions behave consistently in similar ways, and different instruction types use the same underlying Datapath components. This enables designers to reuse ALU, register file, and multiplexers across multiple stages of the pipeline. The architecture also provides a rich set of arithmetic, logical, branch, and memory access instructions, making it versatile enough for both control-oriented and computation-oriented applications.

Due to its clean design, MIPS has been widely adopted in embedded processors, networking hardware, consumer electronics, and academic CPU prototypes. Even though other architectures like ARM dominate the commercial space today, MIPS continues to be a preferred choice for learning processor design concepts because it offers a perfect balance between simplicity and completeness. Its deterministic behaviour, minimal instruction-types, and modular architecture make it easy to extend, modify, and experiment with — as demonstrated in this project through the customization of the register file and pipelined Datapath.

In summary, the MIPS architecture provides an excellent platform for understanding processor structures, Datapath organization, pipelining, instruction execution flow, and control logic. By basing this project on MIPS principles, it becomes possible to implement a practical, efficient, and educationally meaningful pipelined processor that clearly demonstrates how real CPUs execute instructions at the hardware level.

1.3 Motivation for Custom Design:

The motivation behind designing a **custom MIPS-based ALU processor** stems from the need to gain a deeper, hands-on understanding of processor microarchitecture and its internal working principles. While theoretical study provides insight into CPU operations, only through implementation can one truly appreciate the underlying challenges, trade-offs, and design choices involved in building a fully functional processor. This project allows for exploration beyond textbook knowledge, enabling customization, experimentation, and optimization of the Datapath and control logic.

A major motivation for this design is to develop a processor that is lightweight, modular, and adaptable, unlike standard MIPS implementations that follow fixed conventions such as a 32×32 register file and predefined hardware structures. By implementing a **20×32 register file**, this project showcases how architectural components can be optimized for reduced hardware usage while maintaining the capability to execute essential instructions. This customization demonstrates flexibility in hardware architecture and highlights how processor resources can be scaled based on application requirements.

Additionally, the project serves to explore pipelining in a practical setting, a technique central to modern high-performance processors. Studying pipelining solely through diagrams or theory limits understanding of its complexity; however, implementing it in Verilog and verifying it through simulation provides authentic experience with pipeline hazards, forwarding mechanisms, stalls, and the precise timing relationships between stages.

Another key motivation is the opportunity to develop proficiency in **HDL-based hardware design**, simulation, debugging, and verification. Building a processor from scratch enhances familiarity with digital design concepts such as control signal generation, Datapath integration, multiplexing, register synchronization, and memory handling. The experience gained contributes directly to fields such as embedded systems, VLSI design, processor development, and digital system engineering.

Ultimately, this project is motivated by the desire to create a functional and efficient processor that not only reflects the principles of MIPS architecture but also offers the flexibility to modify, expand, and experiment with different architectural components — a valuable experience for any engineer working with processor or hardware design.

1.4 Problem Identification:

Despite the widespread understanding of MIPS architecture, many academic or simplified processor models fail to address the core challenges of processor design, particularly when it comes to implementing pipelining, handling hazards, and integrating Datapath components. The need for a practical architecture that makes these concepts tangible is a significant problem addressed by this project.

One of the primary challenges is the **complexity of pipeline implementation**. While pipelining increases throughput, it introduces hazards such as data dependencies, structural conflicts, and control anomalies. These issues require careful planning, additional hardware units, and intelligent control logic. Many beginner processor designs avoid these complications, but this project tackles them directly, making it necessary to identify, analyse and resolve such hazards.

Another problem is the **hardware resource overhead** associated with traditional MIPS architecture, which mandates a 32×32 register file. For minimal hardware systems or small FPGA implementations, such a large register file may be excessive. This project identifies the need for a more compact register set that can still support the required instruction operations.

Traditional processors also require a substantial amount of hardware to manage the Datapath, ALU operations, memory interactions and control signals. Integrating all these units in an efficient and synchronized manner presents additional challenges. Moreover, designing a scalable and modular architecture requires a thorough understanding of how individual components interact within a full processor pipeline.

Finally, the gap between theoretical instruction execution models and actual hardware behaviour is a major problem. Students or designers often understand the concept of instruction cycles but find it challenging to visualize how values move through registers, how control signals propagate or how pipeline registers store intermediate results. This project identifies this gap and solves it by implementing and simulating a real, functional, pipelined MIPS processor.

1.5 Scope of the Project:

The scope of this project encompasses the complete design, development, simulation, and verification of a **custom pipelined MIPS ALU processor**. The processor features a five-stage pipeline, a simplified instruction set, a custom 20×32 register file, and all necessary control logic to support proper execution flow.

The project covers the following major components:

- **Designing the processor Datapath**, including ALU, register file, instruction memory, data memory, and multiplexers.
- **Implementing the pipeline structure**, including IF, ID, EX, MEM, and WB stages.
- **Constructing pipeline registers** to ensure stable flow of data and control signals across stages.
- **Designing the control unit**, including main control, ALU control, branch logic, and hazard management mechanisms.
- **Developing Verilog HDL modules** for each component and integrating them into a top-level processor module.
- **Simulating the processor** to verify correct instruction execution, timing relationships, and pipeline behaviour.
- **Analysing pipeline hazards**, their detection, and appropriate mitigation strategies such as forwarding and stalling.
- **Testing the processor with sample instructions**, validating ALU operations, load/store functionality, and branch instructions.

Additionally, the project is limited to a defined set of instructions necessary to demonstrate the pipeline's working and the processor's capabilities. While it does not implement the full MIPS instruction set, the modular design allows future extensions such as adding more instructions, implementing branch prediction, expanding the register file, or migrating to a 32-bit full-scale processor

1.6 Objectives:

The primary objectives of the project are:

1. **To design and implement a basic pipelined MIPS** processor using Verilog HDL.
2. **To develop a modified register file** with a 20x32 architecture instead of the standard 32x32.
3. **To create a functional Datapath** that includes instruction fetch, decode, execution, memory access and write back stages.
4. **To integrate an ALU** capable of performing essential arithmetic and logical operations.
5. **To design pipeline latches** that allow concurrent instruction execution.
6. **To understand and manage pipeline hazards** including data, structural and control hazards.
7. **To simulate and verify processor operation** through waveform outputs and timing diagrams.
8. **To demonstrate a complete processor execution cycle** using a sample instruction sequence.
9. **To analyse performance improvements** gained through pipelining compared to non-pipelined architectures.
10. **To document the entire design flow**, from architecture definition to simulation results, providing a detailed understanding of the processor implementation process.

1.7 Methodology:

The development of the custom MIPS-based pipelined processor follows a structured methodology that transforms the architectural concept into a functional hardware model.

1. Architecture Definition

The first step involves defining the processor structure, selecting the instruction subset, designing the 20×32 register file, and outlining the five pipeline stages (IF, ID, EX, MEM, WB).

2. Datapath Development

Individual components such as the ALU, register file, instruction memory, data memory, multiplexers, and PC logic are designed. These blocks are then connected to form the complete Datapath capable of executing instructions.

3. Pipeline Latch Design

Pipeline latches (IF/ID, ID/EX, EX/MEM, MEM/WB) are created to separate stages and store intermediate values. These ensure smooth and synchronized data flow during pipelined execution.

4. Control Unit Implementation

The main control and ALU control units are designed to generate necessary control signals based on instruction types. Branch logic and immediate value handling are also included.

5. Verilog Coding

Each component is implemented as an independent Verilog module. A top-level module integrates all blocks into a single pipelined processor.

6. Simulation & Testing

Testbenches are created to apply sample instructions and observe waveforms. Simulation verifies Datapath correctness, ALU outputs, pipeline timing, and overall instruction flow.

7. Refinement and Documentation

Based on simulation results, the design is refined to correct timing issues or control mismatches. Finally, the entire design process, results, and observations are documented clearly.

1.8 Applications:

1. Educational Use

The processor is highly suitable for learning and teaching computer architecture concepts such as Datapath design, pipelining, ALU operations, and control logic. It helps visualize how instructions move through different stages of execution.

2. Embedded System Prototyping

With its lightweight register file (20×32) and minimal instruction set, this processor can be adapted for small embedded systems where low hardware usage and basic computation are sufficient.

3. FPGA Implementations

The Verilog-based design can be easily deployed onto FPGA boards for hardware demonstrations, laboratory experiments, and real-time testing of pipeline behaviour.

4. Research and Experimentation

The modular architecture makes it a good starting point for extending features such as branch prediction, hazard control, ALU enhancements, or cache integration.

5. Custom Low-Power CPU Designs

This processor structure can be adapted for low-power, application-specific controllers used in robotics, IoT devices, and small digital systems.

CHAPTER - 2
LITERATURE SURVEY

2.1 Background on MIPS RISC Architecture:

The MIPS architecture is a well-known example of **RISC** (Reduced Instruction Set Computing) design. It was developed to achieve high performance through simplicity and efficiency. MIPS uses a **load/store architecture**, where all ALU operations work only on registers, and memory is accessed only through dedicated load and store instructions. This reduces Datapath complexity and makes the design easier to implement in hardware.

One of the main strengths of MIPS is its **fixed 32-bit instruction format**, which simplifies instruction decoding and control signal generation. The architecture uses three basic instruction types—R-type, I-type, and J-type—which follow a consistent structure and are easy to map onto hardware components.

MIPS traditionally includes **32 general-purpose registers**, but many simplified or educational designs modify the register file size depending on hardware constraints, just as this project uses a **20×32 register file**.

MIPS is also widely recognized for its compatibility with **pipelining**. The standard five-stage pipeline (IF, ID, EX, MEM, WB) works naturally with the MIPS instruction set because of its uniform format and simple addressing modes. This is why MIPS is commonly used as a teaching model for processor design and pipeline implementation.

Overall, MIPS provides a clean, easy-to-understand architecture that forms a strong foundation for designing custom processors, especially in academic and experimental environments.

2.2 Evolution of ALU-Based Processors:

The evolution of ALU-based processors reflects the overall growth of computer architecture from simple, single-operation machines to today's high-performance, deeply pipelined CPUs. The **Arithmetic Logic Unit (ALU)** has always been the central component responsible for performing basic arithmetic and logical operations, and its development has directly influenced processor capability, speed, and architecture design.

Early Generations

The earliest processors used **simple, hardwired ALUs** capable of performing only basic operations like addition, subtraction, and logic functions. These systems executed instructions sequentially, taking multiple cycles per instruction due to limited hardware resources and no parallelism.

Microprogrammed Control Era

As computing needs increased, processors adopted **microprogrammed control**, allowing ALUs to support more operations through sequences of micro-instructions. This improved flexibility but did not significantly boost performance because instructions were still executed in a multi-cycle manner.

RISC Architecture Influence

With the rise of **RISC architectures**, ALU design became more streamlined. RISC systems emphasized:

- Simple, fast ALU operations
- Uniform instruction formats
- Load/store principles
- Reduced control complexity

This shift made it easier to build efficient ALUs and helped prepare processors for pipelined execution.

Pipelining and Parallel Execution

The next major evolution was the introduction of **pipelining**, where the ALU became part of a larger execution stage within a multi-stage processor pipeline. Instead of waiting for each instruction to complete fully, processors allowed multiple instructions to pass through the stages simultaneously. The ALU now worked every clock cycle, significantly increasing throughput.

Modern ALU Enhancements

Modern processors include ALUs that support:

- Multiple simultaneous operations
- Fast arithmetic (carry-lookahead, parallel adders)
- Logical operations
- Shift and comparison functions
- Sometimes integrated multiply/divide units

These improvements enhance performance and efficiency across a wide range of applications.

Relevance to This Project

The processor in this project follows this evolutionary pattern by implementing:

- A simplified RISC-style ALU
- A pipelined execution stage

- A compact design suitable for educational and experimental use

It reflects the foundational principles developed over decades of ALU and processor architecture research.

2.3 Existing Pipelined Designs in Research:

Research on pipelined processor designs mainly focuses on implementing simple RISC-based architectures, especially MIPS, due to their clean instruction format and predictable execution flow. Many studies highlight five-stage pipelined processors as standard teaching models, demonstrating how instruction throughput improves when stages operate in parallel.

Most existing designs emphasize basic pipeline components such as IF, ID, EX, MEM, and WB stages, along with essential hazard-handling techniques like stalling, forwarding, and simple branch control. Several academic works also explore reduced-register architectures and lightweight Datapaths to support FPGA implementation or low-resource environments.

These existing pipelined CPU designs serve as practical references and form the foundation for custom or simplified processors—such as the one developed in this project—where certain elements (like register file size or instruction subset) are modified to suit project goals and hardware limitations.

2.4 Key Findings from Literature:

Research strongly highlights that **RISC architectures**, especially MIPS, are preferred for academic processor design because of their simplicity, fixed instruction size, and predictable Datapath behaviour. These characteristics make hardware implementation easier and reduce the overall control complexity.

Several studies point out that the **five-stage pipeline structure** used in MIPS is one of the most effective ways to introduce instruction-level parallelism. The literature consistently shows that splitting execution into IF, ID, EX, MEM, and WB stages improves throughput while maintaining manageable hardware overhead.

Another major finding is the importance of **hazard handling techniques**. Data hazards, control hazards, and structural hazards must be properly addressed for the pipeline to operate correctly. Techniques such as stalling, forwarding, and basic branch handling are commonly discussed in existing designs.

Finally, many academic works emphasize that **custom or simplified versions of MIPS** are useful for teaching and experimentation. Reduced register files, smaller instruction sets, and simplified ALUs are frequently implemented in projects where hardware resources are limited or where the goal is to demonstrate core architecture principles without unnecessary complexity.

2.5 Gap Analysis:

Although many pipelined MIPS processors exist in research, a large number of them either replicate the full 32×32 architecture or focus purely on theoretical aspects without implementing a truly functional pipeline. This leaves a gap for lightweight, practical designs that are easier to understand and implement in academic environments.

Most reference designs also assume access to large register files and abundant hardware resources. This makes them difficult to implement on smaller FPGAs or limited simulation environments. A **20×32 register file**, as used in this project, directly addresses this problem by offering a more compact and resource-efficient architecture.

Another gap identified in the literature is that many simplified designs ignore or minimize pipeline hazards to reduce complexity. While easier to implement, this approach does not accurately represent real processor behaviour. Our design focuses on a clean, functional pipeline structure where hazards can be observed, analysed, and understood clearly.

Overall, the literature suggests the need for a **balanced MIPS model**—not too large, not too simplified—one that maintains realistic pipeline operation while remaining suitable for learning and experimentation. This project fills that space by providing a practical, clear, and efficient pipelined processor design.

CHAPTER - 3

PROPOSED SYSTEM

3.1 Introduction to Proposed Architecture:

The proposed processor architecture is a simplified version of a MIPS-based pipelined CPU, designed to demonstrate core principles of RISC execution and instruction-level parallelism. It follows the traditional five-stage pipeline model while using a reduced **20×32 register file**, making the design lighter and easier to implement.

Key characteristics of the architecture include:

- **Five-stage pipeline**: IF, ID, EX, MEM, and WB
- **Pipeline latches** between every stage
- **A compact Datapath** supporting essential ALU, memory, and branch operations
- **A simplified instruction set** chosen for easy testing and simulation

Overall, this architecture balances simplicity and functionality, making it suitable for academic use, simulation, and small hardware demonstrations.

3.2 Overall Processor Block:

The processor block diagram represents the essential Datapath and control flow. It includes all major components connected in a structured pipeline layout. Each stage is clearly separated, showing how data moves through the processor in a multi-stage execution model.

Main elements in the block diagram:

- Program Counter (PC)
- Instruction Memory
- Register File (20×32)
- ALU and ALU Control
- Data Memory
- Main Control Unit
- Multiplexers (for routing data)
- Immediate Generator
- Pipeline Latches (IF/ID, ID/EX, EX/MEM, MEM/WB)

This diagram helps visualize how instructions pass through different stages and how intermediate values are stored.

3.3 Instruction Set Implemented:

The single cycle Datapath represents the full flow of instruction execution within one clock cycle. All operations—fetching, decoding, executing, memory access, and register update—occur in this single cycle. The diagram includes all major components required to execute basic MIPS-style instructions.

Main Components and Flow

- **Program Counter (PC)**

  Holds the address of the current instruction. After execution, it moves to the next instruction or a branch target.

- **Instruction Memory**

  Provides the instruction to be decoded and executed.

- **Register File (20×32)**

  Supplies the source operands (A and B) and receives the result during write-back.

- **Sign Extend Unit**

  Extends immediate fields for ALU operations and memory addressing.

- **ALU**
  Performs arithmetic or logical operations based on instruction type.

- **Data Memo**

  Used for load (LW) and store (SW) instructions.

- **Multiplexers (MUXes)**

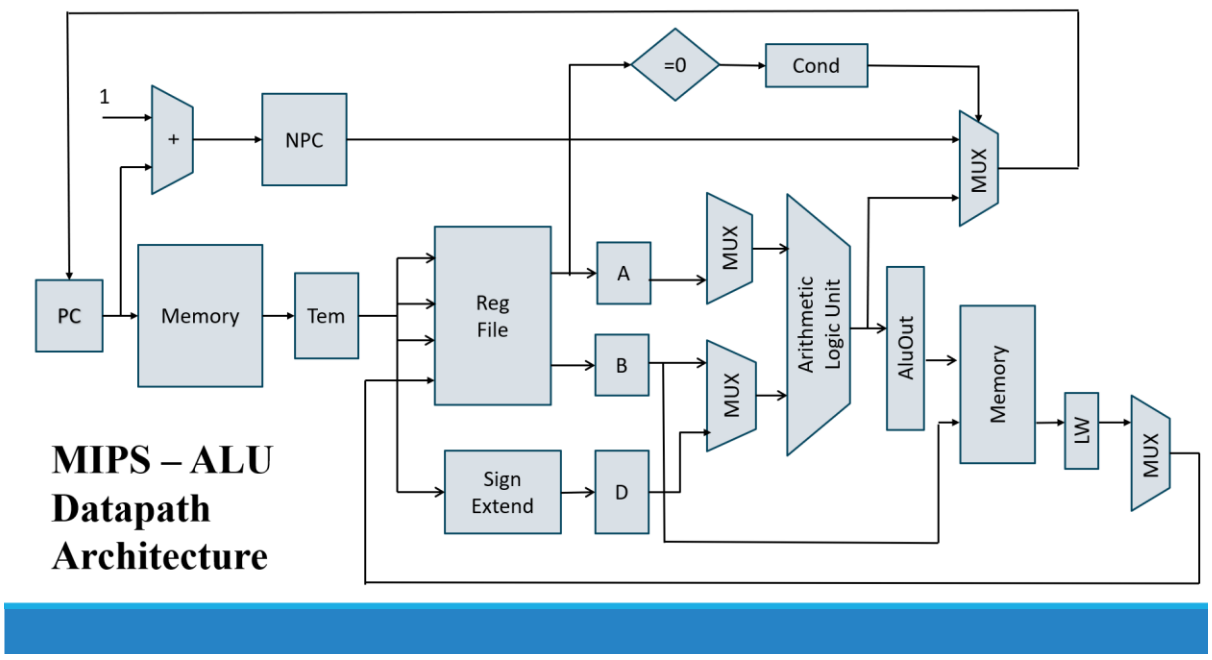
  Select between register operands, immediate values, ALU results, memory outputs, or branch targets.

- **Branch/Condition Logic**

  Checks if branch conditions are met and selects the appropriate next PC value.

Summary

In the single-cycle model, every instruction flows through the entire Datapath in one clock cycle. This representation helps understand the structure before adding pipelining and separating each step.



3.3 Datapath Architecture

3.4 Instruction Formats:

The processor follows the standard MIPS-style classification of instructions into three basic formats: **R-type**, **I-type**, and **J-type**. Each format defines how the 32-bit instruction word is divided into fields such as opcode, register identifiers, function codes, and immediate values. Understanding these formats is essential because they directly influence how instructions are decoded, how control signals are generated, and how the Datapath routes operands through each pipeline stage.

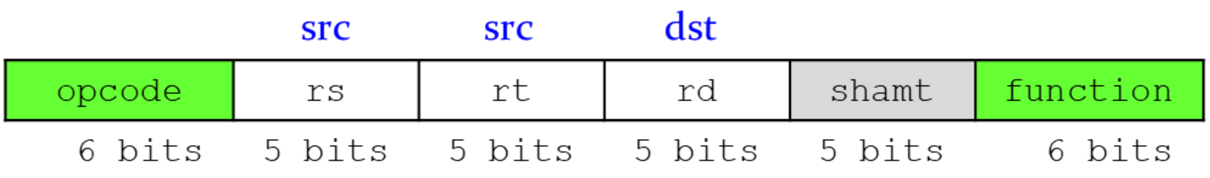
R-Type (Register Type) Format

R-type instructions perform register-to-register operations using the ALU. They contain fields that specify the source registers, destination register, and the ALU operation.

Structure:

- **Opcode (6 bits)** – Always 0 for R-type instructions
- **rs (5 bits)** – First source register
- **rt (5 bits)** – Second source register
- **rd (5 bits)** – Destination register
- **shamt (5 bits)** – Shift amount (used in shift operations)
- **funct (6 bits)** – Specifies the ALU operation (ADD, SUB, AND, OR, etc.)

R-type instructions are used for operations such as arithmetic and logical functions.



3.3 R-Type Format

I-Type (Immediate Type) Format

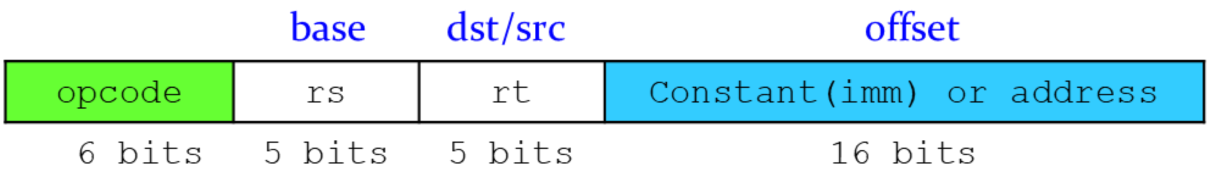
I-type instructions use immediate values or perform memory access. They include load/store instructions and branch instructions with a 16-bit immediate field.

Structure:

- **Opcode (6 bits)** – Identifies instruction type
- **rs (5 bits)** – First operand
- **rt (5 bits)** – Destination register (or source for SW)
- **Immediate (16 bits)** – Constant value or address offset

Common I-type instructions include **LW**, **SW**, and **BEQ**.

The immediate field allows efficient address calculations and quick data manipulation.



3.3 I-Type Format

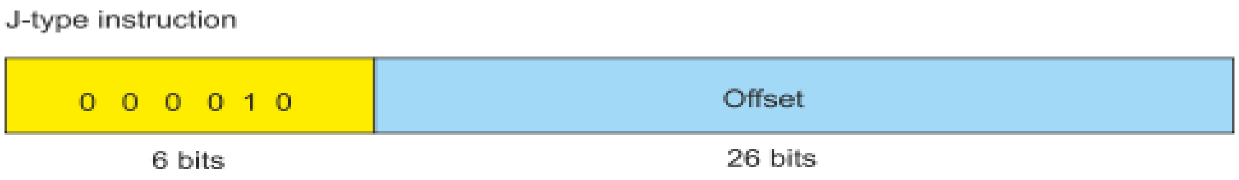
J-Type (Jump Type) Format

J-type instructions are used for long-range jumps. They contain a large 26-bit address field that allows jumping to distant instruction locations.

Structure:

- **Opcode (6 bits)** – Identifies jump instruction
- **Target Address (26 bits)** – Combined with upper bits of PC to form jump target

Although this project uses only basic control instructions, understanding J-type format is useful for extending the processor with full jump support.



3.3 J-Type Format

Summary

The three instruction formats define how the Datapath interprets and processes instructions.

- **R-type → Register operations**
- **I-type → Immediate, load/store, and branch operations**
- **J-type → Jump operations** (optional for future expansion)

These formats create consistency across the architecture and make decoding simpler in each pipeline stage.

3.5 ALU Design:

The Arithmetic Logic Unit (ALU) is the core computational block of the processor. It performs all arithmetic and logical operations required by the instruction set. In this design, the ALU supports essential R-type instructions such as ADD, SUB, AND, and OR, as well as address calculations for load/store instructions.

Functions Performed

- **Arithmetic operations:** Addition, subtraction
- **Logical operations:** AND, OR
- **Address Computation:** For LW and SW instructions
- **Branch Comparison:** Used to evaluate conditions (e.g., BEQZ)

Inputs and Outputs

- **Inputs:**
  - Operand A (from RegFile)
  - Operand B (from RegFile or immediate value)
  - ALU control signals (derived from instruction opcode)
- **Outputs:**
  - ALU Result
  - Zero Flag (used for branch decisions)

The ALU is positioned in the EX stage of the pipeline and plays a key role in both normal instruction execution and branch evaluation.

3.6 Registe File Description:

The register file in this processor contains **20 registers**, each 32 bits wide. This is a deliberate reduction from the traditional MIPS register file of 32 registers × 32 bits.

Why 20×32 Instead of 32×32?

- **Hardware Resource Reduction:**

  Smaller register files require fewer flip-flops and less area, which is useful for FPGA or small-scale hardware implementations.

- **Project Requirements:**

  Since the goal is to demonstrate pipeline behaviour and ALU operations, only a limited number of registers are needed.

- **Simplified Design:**

  A smaller register file makes decoding logic and addressing easier, speeding up simulation and reducing clutter in the Datapath.

Register File Features

- Two read ports (A and B outputs)
- One write port (Write Data input)
- Synchronous write operation on clock edge
- Works seamlessly with all supported instructions

Despite the reduced size, the 20×20 register file provides full functionality for the implemented instruction set and the pipelined architecture.

3.7 Control Unit:

The control unit determines how each instruction behaves as it moves through the pipeline. In this processor, the control logic is simplified and does not use separate Main Control or ALU Control blocks. Instead, instruction control is handled through the decoded **opcode** and the instruction **type field**.

Types of Control Signals Generated

- **ID_EX_Type**
  Determine instruction category

  RR_ALU; RM_ALU; LOAD; STORE; BRANCH; HALT_T; NOP_T

- **Opcode (ID_ID_IR[31:26]**

  Selects the exact ALU operation (ADD, SUB, AND_OP, OR_OP, SLT, MUL, ADDI, SUBI, SLTI)

- **EX_MEM_Cond**
  Branch condition flag for BEQZ / BNEQZ
- **TAKEN_BRANCH**

  Controls whether the next instruction is fetched from the branch target.
- **HALTED**

  Stops pipeline execution when an HLT instruction is reached.

Control Unit Structure

- **Instruction Type Decoder:**

  Uses opcode to assign ID_EX_Type, which determines how each stage behaves.
- **Integrated ALU Operation Logic:**

  No separate ALU Control Block. The ALU operation is selected directly in the EX stage **case statement** using opcode from ID_EX_IR[31:26].

The control unit ensures proper execution by assigning the correct instruction type and using opcode-based logic to activate ALU, memory, branch, and write-back behaviour during each pipeline stage.

3.8 Pipeline Architecture:

The proposed processor uses a **five-stage pipelined architecture**, where multiple instructions are processed simultaneously at different stages. Pipeline latches are placed between each stage to hold intermediate values and maintain smooth data flow. This improves instruction throughput and helps demonstrate instruction-level parallelism.

3.8.1 Instruction Fetch (IF) Stage

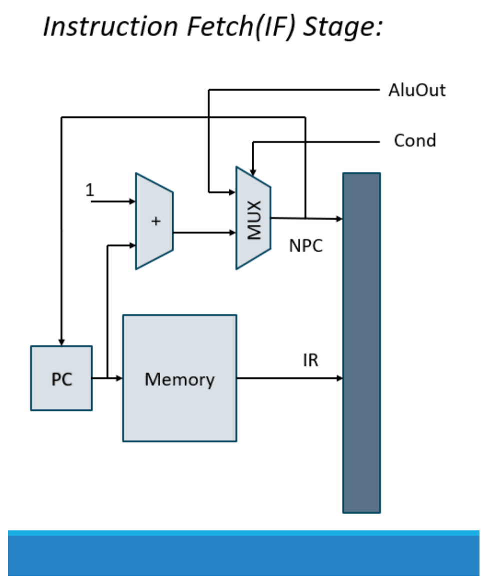
The IF stage retrieves the next instruction from memory and updates the Program Counter (PC).

Key Operations

- Reads instruction from Instruction Memory
- Computes next PC (PC + 1 or branch target)
- Sends instruction to IF/ID latch

Components Used

- Program Counter
- Instruction Memory
- PC Increment Logic
- IF/ID Pipeline Latch



3.8 IF Stage

3.8.2 Instruction Decode (ID) Stage

The ID stage decodes the instruction and prepares operands for execution.

Key Operations

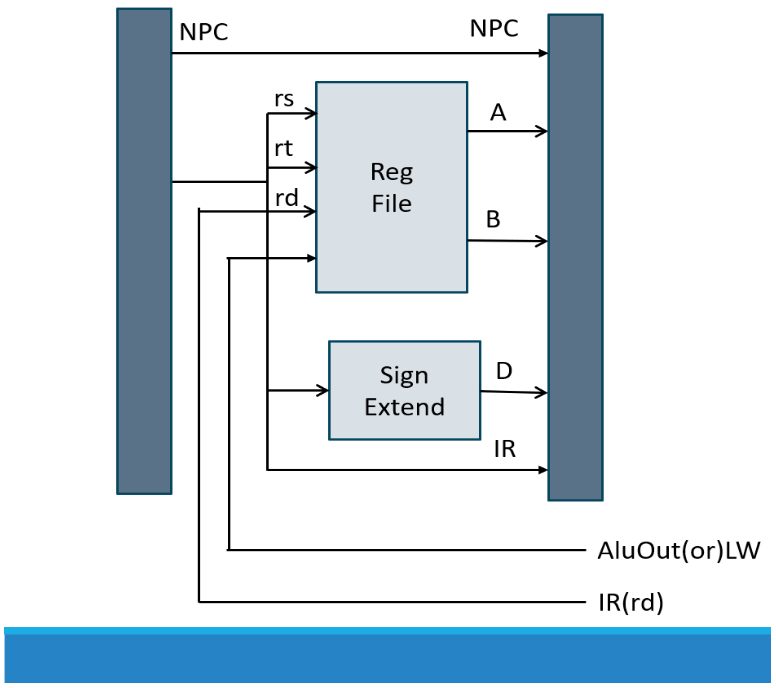
- Extracts opcode, rs, rt, rd, immediate
- Reads source registers from Register File
- Generates control signals
- Extends immediate values

Outputs

- Operand A, Operand B
- Immediate value
- Control signals

The results are stored in the **ID/EX latch**.

Instruction Decode(ID) Stage:



3.8 ID Stage

3.8.3 Execution (EX) Stage

This stage performs ALU operations and computes branch decisions.

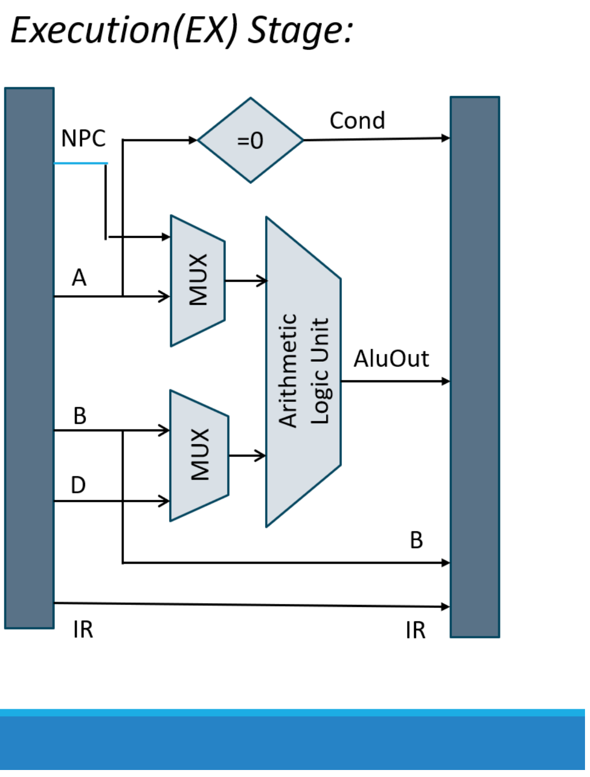
Key Operations

- ALU performs arithmetic/logical operation
- Immediate or register value selected via MUX
- Branch condition evaluated
- Effective address computed for LW/SW

Outputs

- ALU Result
- Branch decision flag
- Target address

Stored in **EX/MEM latch** for next stage.



3.8 EX Stage

3.8.4 Memory Access (MEM) Stage

This stage handles data memory operations.

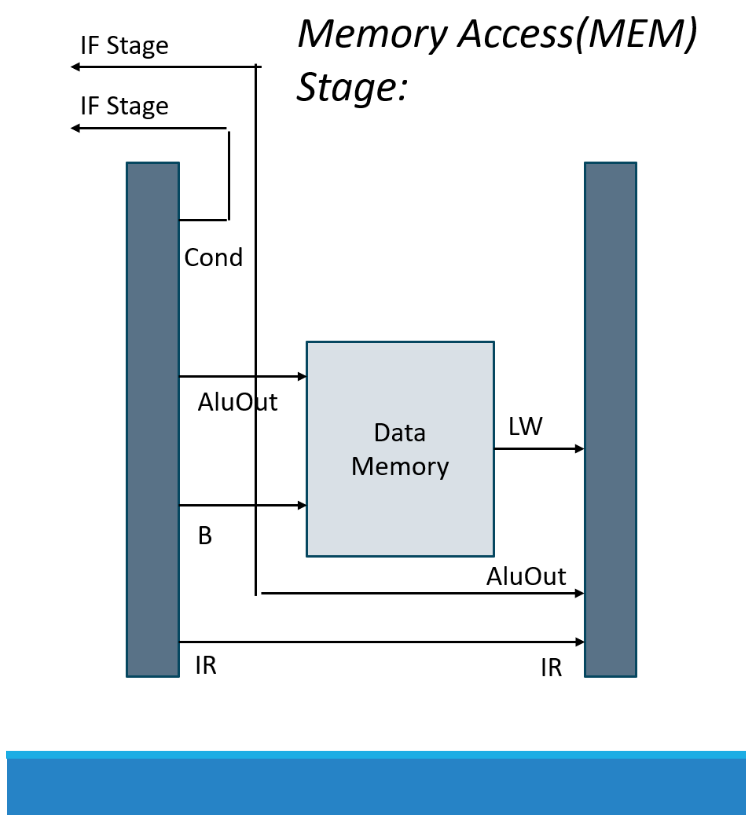
Key Operations

- For LW: read data from memory
- For SW: write data to memory
- For R-type: simply forward ALU result

Components

- Data Memory
- Branch control logic

Results go to the **MEM/WB latch**.



3.8 MEM Stage

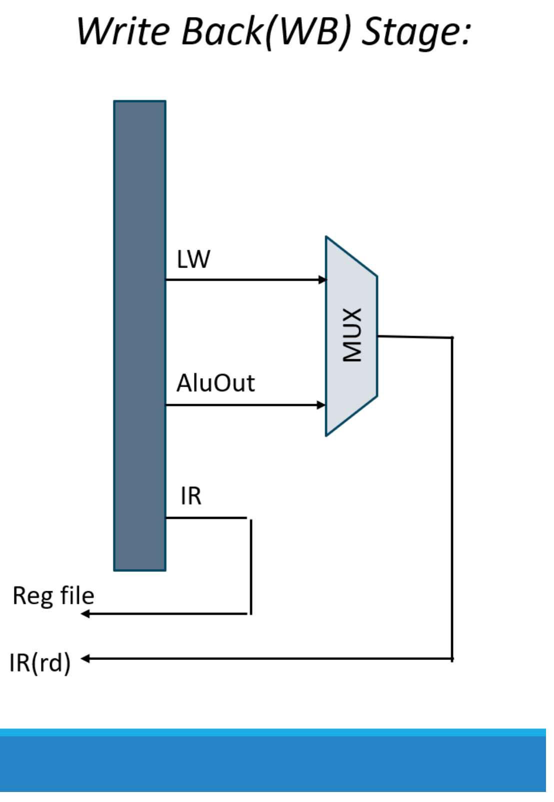
3.8.5 Write Back (WB) Stage

The Write Back (WB) stage is the final stage of the pipeline, where the results of an instruction are written back into the register file. This stage ensures that the outcome of either an ALU operation or a memory access becomes permanently stored for use by future instructions. Completing the write-back step finalizes the instruction lifecycle.

Key Operations

- The processor chooses between **ALU result** and **memory-read data** using the **MemtoReg MUX**.
- The selected value is written to the destination register specified earlier in the instruction.
- The **RegWrite** control signal ensures that only valid instructions update the register file.

This mechanism guarantees data correctness by placing the final result in the appropriate register.



3.8 WB Stage

3.9 Hazards:

Pipeline hazards occur when overlapping instructions interfere with each other. The processor must detect and handle these hazards to ensure correct execution.

3.9.1 Data Hazards

Data hazards occur when an instruction depends on the result of a previous instruction.

Example:

ADD R1, R2, R3

SUB R4, R1, R5 ← depends on result of ADD

Types

- Read After Write (RAW)
- Write After Read (WAR)
- Write After Write (WAW)

RAW is the most common in a simple pipeline like this.

3.9.2 Structural Hazards

Structural hazards occur when two pipeline stages need the **same hardware resource** at the same time.

Example:

- If both IF and MEM need access to memory simultaneously.

In this design, structural hazards are minimized due to separate instruction and data memory blocks.

3.9.3 Control Hazards

Control hazards arise from branch instructions.

Cause

- The pipeline does not know the next PC value until the branch condition is evaluated in EX stage.

Impact

- Wrong instructions may enter the pipeline after a branch.

3.9.4 Hazard Mitigation Techniques

To ensure correctness, hazards are managed using:

• Forwarding (Data Hazards)

- Sends ALU result directly to next instruction without waiting for WB.
- Reduces RAW hazards.

• Stalling (Data/Control Hazards)

- Inserts a bubble (NOP) into the pipeline.
- Used when forwarding cannot resolve the dependency.

• Simple Branch Handling

- Branch decision taken in EX stage.
- PC updated accordingly.

These techniques maintain correct execution and stable pipeline flow.

3.10 Advantages of Proposed Architecture:

- **Improved Performance:**
  Pipelining increases throughput by executing multiple instructions in parallel.
- **Reduced Hardware Cost:**
  A 20×32 register file lowers resource usage without affecting functional demonstration.
- **Modular and Easy to Understand:**
  Each stage is clearly separated, making debugging and testing easier.
- **Educational Value:**
  Demonstrates real pipelined processor behaviour in a simplified form.
- **Flexible for Extensions:**
  ALU, control unit, and registers can be expanded easily for future enhancements.

3.11 Limitations:

- **Limited Instruction Set:**
  Only a subset of MIPS instructions is implemented.
- **Basic Hazard Handling:**
  No advanced branch prediction or complex forwarding matrix.
- **Reduced Register File Size:**
  20 registers may restrict certain complex programs.
- **Single-cycle Memory Access Assumption:**
  Real processors use multi-cycle or cached memory systems.

CHAPTER - 4
SYSTEM REQUIREMENTS

4.1 Hardware Requirements:

The hardware required for developing and simulating the proposed pipelined MIPS-ALU processor is minimal, since most of the design and testing is performed through HDL simulation tools. The following components may be used:

Essential Hardware

- **Personal Computer (PC)**

  Used for coding, compiling, and running simulations.

  - Minimum: Dual-core processor, 4 GB RAM
  - Recommended: Quad-core processor, 8 GB+ RAM

Optional Hardware (If hardware implementation is attempted)

- **FPGA Development Board**

  For implementing the processor in real hardware. Example boards:

  - Xilinx Spartan-6 / Artix-7
  - Intel (Altera) Cyclone series

- **USB Programmer / JTAG Cable**

  Needed to upload bitstreams onto the FPGA.

- **Power Supply and Supporting Peripherals**

  Based on the FPGA development environment.

These hardware components allow efficient testing and optional real-time execution of the processor.

4.2 Software Requirements:

Software tools play a major role in designing, coding, and testing the processor. The following software is required:

HDL Design and Simulation Tools

- **ModelSim / Questasim**

  Industry-standard simulators used for Verilog code compilation and waveform analysis.

- **Icarus Verilog (iverilog)**

  Lightweight, open-source simulator for testing and debugging Verilog code.

- **GTKWave**
  Used for viewing waveform outputs generated during simulation.

HDL Coding Environment

- **Any Verilog-Compatible Text Editor or IDE**, such as:
  - VS Code
  - Sublime Text
  - Notepad++
  - Vivado

FPGA Toolchain

- **Xilinx Vivado / ISE** or **Intel Quartus**

  Required only if the processor is implemented on FPGA hardware.

These tools together support coding, compilation, simulation, debugging, and optional hardware synthesis of the pipelined processor.

CHAPTER - 5

DESIGN & IMPLEMENTATION

5.1 Datapath Architecture Explanation:

The Datapath forms the core functional structure of the processor. It connects all major hardware components such as the Program Counter, Register File, ALU, Memory units, and Control blocks. Each instruction travels through this Datapath as it moves across the five pipeline stages: IF, ID, EX, MEM, and WB. The architecture is built to support a simplified MIPS-style processor with a **20×32 register file**, a compact ALU, and minimal memory logic, making it an efficient and educational Datapath model.

Overall Flow of Data

Instruction execution begins at the **Program Counter (PC)**, which holds the address of the current instruction. In the **IF stage** (on clk1), the instruction is fetched from memory using Mem[PC] and stored into **IF_ID_IR**, while the next address (PC + 1) is saved in **IF_ID_NPC**. If a branch was taken in the previous cycle (TAKEN_BRANCH = 1), the instruction is instead fetched from the branch target stored in **EX_MEM_ALUOut**, ensuring correct control flow.

During the **ID stage** (on clk2), the instruction in IF_ID_IR is decoded into opcode, rs, rt, rd, and immediate fields. The register file supplies operands into **ID_EX_A** and **ID_EX_B**, while the immediate value is sign-extended and stored in **ID_EX_Imm**. The instruction type (ID_EX_type) is determined from the opcode, classifying it as RR_ALU, RM_ALU, LOAD, STORE, BRANCH, HALT, or NOP. All decoded values, including the next PC and the original instruction, are forwarded into the **ID/EX latch**.

In the **EX stage** (on clk1), the ALU performs the required computation using ID_EX_A, ID_EX_B, or ID_EX_Imm depending on ID_EX_type. The exact operation (ADD, SUB, MUL, AND, OR, SLT, ADDI, SUBI, SLTI) is chosen directly from the opcode (ID_EX_IR[31:26]). For memory instructions, the ALU computes the effective address; for branches, the condition is evaluated and stored in **EX_MEM_cond**, and if true, the PC is updated and TAKEN_BRANCH is asserted. All results and necessary values are placed in the **EX/MEM latch**.

In the **MEM stage** (on clk2), load instructions read data from memory into **MEM_WB_LMD**, while store instructions write the value in **EX_MEM_B** to memory at the computed address. Non-memory instructions simply pass their ALU result through to **MEM_WB_ALUOut**. All results are stored into the **MEM/WB latch** for final processing.

Finally, during the **WB stage** (on clk1), the processor writes the result back to the register file. RR_ALU instructions write to the rd field, while RM_ALU and LOAD instructions write to the rt field. Store and branch instructions do not perform write-back. If the instruction is HALT, the signal **HALTED** is raised, stopping further pipeline execution.

Major Components of the Datapath

To support this flow, the Datapath includes the following essential units:

1. Program Counter (PC)

- Holds the current instruction address
- Updates every cycle based on sequential PC+1 or branch target

2. Instruction Memory

- Stores the program
- Outputs a 32-bit instruction on every fetch cycle

3. Register File (20×32)

- Two read ports and one write port
- Supplies operands for ALU operations
- Receives results in the WB stage

4. ALU (Arithmetic Logic Unit)

- Executes arithmetic/logical operations
- Computes memory addresses
- Evaluates branch conditions

5. Data Memory

- Stores and retrieves data for LW and SW instructions

6. Multiplexers

Used for:

- Selecting between register or immediate operand
- Choosing ALU result or memory data for WB
- Controlling branch and PC update logic

7. Control Unit

- Generates control signals based on opcode
- Coordinates all Datapath operations
- Ensures correct routing of data

8. Pipeline Latches

- IF/ID, ID/EX, EX/MEM, and MEM/WB

- Preserve instruction flow and intermediate values
- Essential for pipelined execution

Purpose and Importance of the Datapath

The Datapath is the backbone of the processor. It determines:

- How data moves between hardware units
- How instructions are executed and updated
- How pipeline parallelism is maintained
- How control signals influence operation

A properly designed Datapath ensures that every instruction progresses smoothly, all hazards are minimized, and the processor performs efficiently.

5.2 ALU Internal Design:

The Arithmetic Logic Unit (ALU) in this processor performs all arithmetic, logical, and comparison operations during the **EX stage**, which runs on the **posedge of clk1**.

In the Verilog implementation, ALU behaviour is embedded directly inside the EX-stage always block, and its result is written to the pipeline register **EX_MEM_ALUOut**. Because all operations are computed inside a combinational case structure, the ALU behaves like a **pure combinational unit**, producing output immediately based on its inputs (ID_EX_A, ID_EX_B, or ID_EX_Imm).

Functional Operations

The ALU supports a selected set of operations aligned with the instruction set implemented in the processor. These include:

- **Arithmetic Operations:**
    - Addition (ADD)
    - Subtraction (SUB)
- **Logical Operations:**
    - Bitwise AND
    - Bitwise OR
- **Comparison Operations:**
    - Zero detection for branch evaluation (used by BEQZ)
- **Address Calculation:**
    - Adding base register + immediate offset for LW and SW

These operations ensure that the ALU can handle both R-type and I-type instructions efficiently.

Inputs and Control Signals

The ALU receives two main operands and a control code that determines which operation to perform.

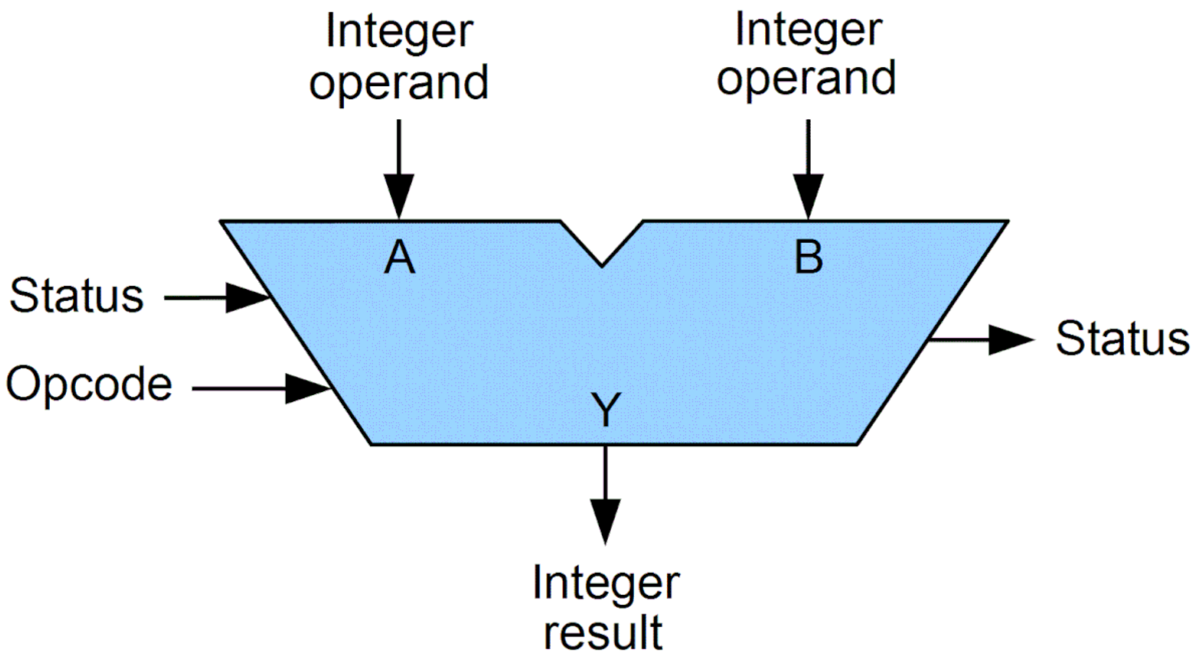
Inputs:

- **Operand A:** Value from registers rs, stored in ID_EX_A.
- **Operand B:** Value from rt (ID_EX_B) or the sign extended immediate (ID_EX_Imm)selected based on ID_EX_TYPE.
- **Operand Selection:** Determined directly from ID_EX_Type and the opcode ID_EX_IR[31:26].

Outputs:

- **ALU Result:** The computed value (arithmetic/logical/address result)
- **Zero Flag:** Indicates if the result is zero, used mainly for branch decisions

These outputs are stored into the **EX/MEM pipeline latch** for use in memory or write-back stages.



5.2 ALU Internal Diagram

Internal Structure

Internally, the ALU is built using:

- **Arithmetic circuits** (adder/subtractor implemented using ripple-carry or simple adder logic)
- **Logic circuits** (basic AND/OR gates)
- **Multiplexers** to select the required operation based on the ALU control signal

This modular design keeps the ALU simple but fully functional for the reduced instruction set.

Role in Pipeline Execution

Within the EX stage, the ALU:

- Computes register results for R-type instructions
- Calculates effective memory addresses for I-type instructions
- Evaluates branch conditions using subtraction and zero flag
- Forwards result to the next stage to enable pipelined execution

Since the ALU performs critical computations, its correct functioning directly affects the accuracy and speed of overall instruction execution.

5.3 Control Unit FSM:

The Control Unit is responsible for directing the flow of data inside the processor by generating the appropriate control signals for each stage of the Datapath. It acts like the “brain” of the processor, interpreting the instruction’s and activating the necessary components to execute that instruction correctly.

In this design, the control logic behaves like a **Finite State Machine (FSM)**, even though the MIPS pipeline generally uses a single-cycle control approach. By treating it as an FSM, the control signals are logically grouped based on instruction type, allowing a structured and clear implementation.

Main Responsibilities of the Control Unit

The control unit performs four key tasks:

1. **Instruction Classification**
    - Identifies whether the instruction is R-type, I-type (LW, SW, BEQZ), or J-type.
    - Determines which functional blocks are needed for execution.

2. **Signal Generation**

   Produces signals that control multiplexers, register file, ALU, memory units, and write-back paths.

   Key control signals include:

   - **ID_EX_type** – Selects instruction category (RR_ALU, RM_ALU, LOAD, STORE, BRANCH, HALT_T)
   - **Opcode (ID_EX_IR[31:26])** – Selects the exact ALU operation inside each instruction type.
   - **EX_MEM_cond** – Branch condition flag (used for BEQZ and BNEQZ)
   - **TAKEN_BRANCH** – Controls whether PC and pipeline fetch should follow a branch target .
   - **HALTED** – Stops all pipeline activity when HLT instruction executes

3. **ALU Operation Selection**

   - The ALU operation is selected directly using the **instruction opcode** found in **ID_EX_IR[31:26]**.
   - The **instruction type** stored in **ID_EX_type** (RR_ALU, RM_ALU, LOAD, STORE, BRANCH) determines which group of operations the ALU performs.
   - No separate ALUOp or ALU Control Unit is used; the EX-stage case statement directly computes ADD, SUB, AND, OR, SLT, MUL, etc.

4. **Coordinating Pipeline Flow**

   - Ensures correct control signal propagation across pipeline stages
   - Works with pipeline latches to pass signals into ID/EX, EX/MEM, and MEM/WB stages

Internal Structure of the Control Unit

In this design, the control unit is implemented in a simplified form without separate Main Control or ALU Control blocks.

1. Instruction Type Decoder

- Takes **opcode** as input
- Produces the instruction category stored in **ID_EX_type** (Value can be: RR_ALU, RM_ALU, LOAD, STORE, BRANCH, HALT_T, NOP_T)
- This classification determines how operands, immediates, and ALU logic behave in later stages

2. Integrated ALU Operation Selection

- The exact ALU operation is selected directly using the **opcode** inside the EX stage

- No ALUOp or funct-based ALU control block is used
- Operations such as ADD, SUB, AND_OP, OR_OP, SLT, MUL, ADDI, SUBI, SLTI are selected in the EX-stage case statement

How the FSM Behaviour Works

Each instruction type corresponds to a "state" represented by the value stored in:

ID_EX_type.
This determines how the EX, MEM, and WB stages behave.

1. **RR_ALU State (R-Type Instructions)**
    - ALU uses ID_EX_A and ID_EX_B.
    - Operation selected directly from opcode (ID_EX_IR[31:26])
    - Result written back to register MEM_WB_IR[15:11].
2. **LOAD State (LW)**
    - Effective address = ID_EX_A + ID_EX_Imm.
    - Memory data loaded into MEM_WB_LWD.
    - Register write-back goes to rt (MEM_WB_IR[20:16])
3. **STORE State (SW)**
    - Effective address = ID_EX_A + ID_EX_Imm.
    - Value in EX_MEM_B written to memory
    - No register write-back
4. **Branch State (BEQZ/BNEQZ)**
    - Condition evaluated in EX_MEM_Cond.
    - If true → PC updated to target branch, TAKEN_BRANCH = 1.
    - No write-back when branch is taken
5. **HALT State**
    - Halted is set to 1
    - All pipeline activity stops

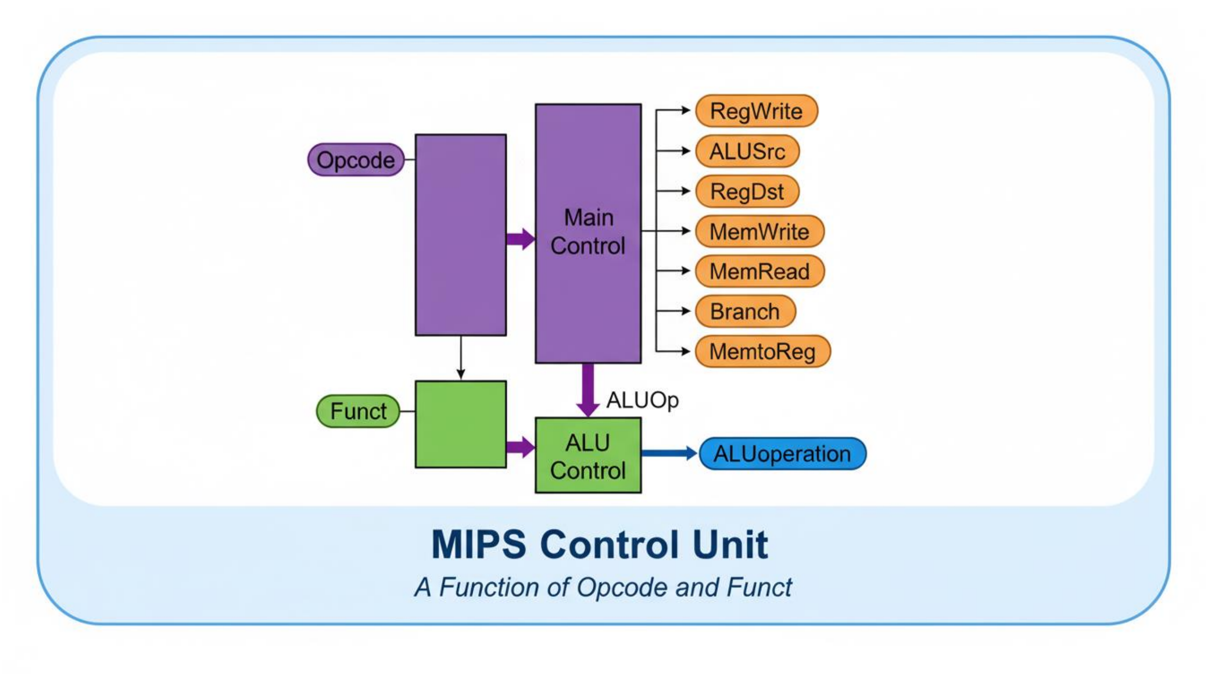
Role in Pipeline Execution

The control unit must ensure that:

- Correct control signals are stored in the **ID/EX latch**
- Memory-related signals flow into **EX/MEM and MEM/WB** latches

- Instructions do not interfere with each other
- Branch decisions are passed correctly to the EX stage

Because each stage depends on the previous one, the control unit's accuracy is essential for stable and error-free pipelined execution.



5.3 Control Unit

5.4 Pipeline Latch Design:

Pipeline latches separate each pipeline stage and hold the intermediate values required for correct execution. Each latch stores instruction fields, control signals, operand values, ALU results, memory data, or PC values depending on the stage.

IF/ID Latch

- Stores: Fetched instruction, PC+1
- Purpose: Passes instruction to the decode stage

ID/EX Latch

- Stores: Register values (A, B), immediate, control signals
- Purpose: Provides operands and control info to EX stage

EX/MEM Latch

- Stores: ALU result, branch flags, target address, control signals

- Purpose: Supports memory access and branch operations

MEM/WB Latch

- Stores: Memory output or ALU result
- Purpose: Supplies result for register write-back

These latches ensure correct timing and prevent interference between overlapping instructions.

5.5 Instruction Execution Flow:

The following describes how typical instructions pass through the pipeline:

RR_ALU Type (ADD/SUB/AND/OR/SLT/MUL)

1. **IF:** Fetch instruction into IF_ID_IR, next PC stored in IF_ID_NPC.
2. **ID:** Registers rs and rt read into ID_EX_A and ID_EX_B.
3. **EX:** Operation selected by opcode ID_EX_IR[31:26] → result written to EX_MEM_ALUOut.
4. **MEM:** No memory access, values move to MEM_WB_ALUOut.
5. **WB:** Write ALU result to rd (MEM_WB_IR[15:11]).

LW (LOAD, LOAD Type)

1. **IF:** Fetch instruction into IF_ID_IR.
2. **ID:** Base register read into ID_EX_A; Offset sign-extended into ID_EX_Imm.
3. **EX:** Address computed → EX_MEM_ALUOut = ID_EX_A = ID_EX_Imm.
4. **MEM:** Memory read → MEM_WB_LMD = Mem[EX_MEM_ALUOut]
5. **WB:** Loaded value written to rt (MEM_WB_IR[20:16])

SW (STORE, STORE Type)

1. **IF:** Instruction loaded into IF_ID_IR
2. **ID:** Base register into ID_EX_A, store data into ID_EX_B
3. **EX:** Address computed → EX_MEM_ALUOut = ID_EX_A + ID_EX_Imm.
4. **MEM:** Memory Write → MEM[EX_MEM_ALUOut] = EX_MEM_B (if branch not taken)
5. **WB:** No register write-back

These flows illustrate correct data movement across pipeline stages.

CHAPTER - 6
VERILOG CODE

6.1 ALU Verilog Code:

```
module pipe_MIPS20(
    input  wire clk1,
    input  wire clk2,

    // OUTPUTS (Visible in Waveform & Netlist)
    output wire [31:0] pc_out,
    output wire [31:0] alu_result,
    output wire        halted_out,

    // NEW: DEBUG OUTPUTS FOR INPUT OPERANDS
    output wire [31:0] debug_operand1,
    output wire [31:0] debug_operand2
);

    // ---------- Pipeline Registers ----------
    reg [31:0] PC;
    reg [31:0] IF_ID_IR, IF_ID_NPC;
    reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
    reg [2:0]  ID_EX_type, EX_MEM_type, MEM_WB_type;
    reg [31:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_B;
    reg        EX_MEM_cond;
    reg [31:0] MEM_WB_IR, MEM_WB_ALUOut, MEM_WB_LMD;

    // Register Bank (20x32) & Memory
```

```
reg [31:0] Reg [0:19];
reg [31:0] Mem [0:1023];

// Flags
reg HALTED;
reg TAKEN_BRANCH;

// ---------- OUTPUT CONNECTIONS ----------
assign pc_out     = PC;
assign alu_result = EX_MEM_ALUOut;
assign halted_out = HALTED;

// LOGIC TO SHOW WHAT IS ENTERING THE ALU
// Operand 1 is always A
assign debug_operand1 = ID_EX_A;

// Operand 2 depends on type:
// If RR_ALU (like ADD), use Register B. Otherwise (ADDI, LW, STORE), use Immediate.
assign debug_operand2 = (ID_EX_type == 3'b000) ? ID_EX_B : ID_EX_Imm;

// ---------- OPCODES ----------
parameter ADD    = 6'b000000, SUB    = 6'b000001, AND_OP = 6'b000010,
          OR_OP  = 6'b000011, SLT    = 6'b000100, MUL    = 6'b000101,
          HLT    = 6'b111111, LW     = 6'b001000, SW     = 6'b001001,
          ADDI   = 6'b001010, SUBI   = 6'b001011, SLTI   = 6'b001100,
          BNEQZ  = 6'b001101, BEQZ   = 6'b001110;

parameter RR_ALU = 3'b000, RM_ALU = 3'b001, LOAD   = 3'b010,
          STORE  = 3'b011, BRANCH = 3'b100, HALT_T = 3'b101, NOP_T  = 3'b110;
```

```
// Helper wires
wire [5:0] opcode = IF_ID_IR[31:26];
wire [4:0] rs     = IF_ID_IR[25:21];
wire [4:0] rt     = IF_ID_IR[20:16];
wire [4:0] rd     = IF_ID_IR[15:11];

integer i;

// ---------- Initialization ----------
initial begin
   PC = 0; HALTED = 0; TAKEN_BRANCH = 0;
   IF_ID_IR = 0; IF_ID_NPC = 0; ID_EX_IR = 0; ID_EX_NPC = 0;
   EX_MEM_IR = 0; MEM_WB_IR = 0;
   for (i=0; i<20; i=i+1) Reg[i]=0;
   for (i=0; i<1024; i=i+1) Mem[i]=0;
end

// ---------- IF Stage ----------
always @(posedge clk1) begin
   if (!HALTED) begin
      if ((EX_MEM_IR[31:26] == BEQZ && EX_MEM_cond == 1) ||
         (EX_MEM_IR[31:26] == BNEQZ && EX_MEM_cond == 1)) begin
         IF_ID_IR  <= Mem[EX_MEM_ALUOut];
         IF_ID_NPC <= EX_MEM_ALUOut + 1;
         PC        <= EX_MEM_ALUOut + 1;
         TAKEN_BRANCH <= 1;
      end else begin
         IF_ID_IR  <= Mem[PC];
         IF_ID_NPC <= PC + 1;
         PC       <= PC + 1;
      end
   end
end

// ---------- ID Stage ----------
always @(posedge clk2) begin
   if (!HALTED) begin
      ID_EX_Imm <= {{16{IF_ID_IR[15]}}, IF_ID_IR[15:0]};

      if (rs == 0) ID_EX_A <= 0;
      else if (rs < 20) ID_EX_A <= Reg[rs];
      else ID_EX_A <= 0;

      if (rt == 0) ID_EX_B <= 0;
      else if (rt < 20) ID_EX_B <= Reg[rt];
      else ID_EX_B <= 0;

      ID_EX_NPC <= IF_ID_NPC;
      ID_EX_IR  <= IF_ID_IR;

      case (opcode)
         ADD, SUB, AND_OP, OR_OP, SLT, MUL: ID_EX_type <= RR_ALU;
         ADDI, SUBI, SLTI:              ID_EX_type <= RM_ALU;
         LW:                         ID_EX_type <= LOAD;
         SW:                         ID_EX_type <= STORE;
         BNEQZ, BEQZ:                   ID_EX_type <= BRANCH;
         HLT:                         ID_EX_type <= HALT_T;
         default:                     ID_EX_type <= NOP_T;
```

```
        endcase
    end
end

// ---------- EX Stage ----------
always @(posedge clk1) begin
    if (!HALTED) begin
        EX_MEM_type <= ID_EX_type;
        EX_MEM_IR   <= ID_EX_IR;
        TAKEN_BRANCH <= 0;
        EX_MEM_ALUOut <= 0; EX_MEM_B <= ID_EX_B; EX_MEM_cond <= 0;

        case (ID_EX_type)
            RR_ALU: begin
                case (ID_EX_IR[31:26])
                    ADD:    EX_MEM_ALUOut <= ID_EX_A + ID_EX_B;
                    SUB:    EX_MEM_ALUOut <= ID_EX_A - ID_EX_B;
                    AND_OP: EX_MEM_ALUOut <= ID_EX_A & ID_EX_B;
                    OR_OP:  EX_MEM_ALUOut <= ID_EX_A | ID_EX_B;
                    SLT:    EX_MEM_ALUOut <= (ID_EX_A < ID_EX_B) ? 1 : 0;
                    MUL:    EX_MEM_ALUOut <= ID_EX_A * ID_EX_B;
                endcase
            end
            RM_ALU: begin
                case (ID_EX_IR[31:26])
                    ADDI: EX_MEM_ALUOut <= ID_EX_A + ID_EX_Imm;
                    SUBI: EX_MEM_ALUOut <= ID_EX_A - ID_EX_Imm;
                    SLTI: EX_MEM_ALUOut <= (ID_EX_A < ID_EX_Imm) ? 1 : 0;
                endcase
```

```
          end
          LOAD, STORE: EX_MEM_ALUOut <= ID_EX_A + ID_EX_Imm;
          BRANCH: begin
            EX_MEM_ALUOut <= ID_EX_NPC + ID_EX_Imm;
            if (ID_EX_IR[31:26] == BEQZ) EX_MEM_cond <= (ID_EX_A == 0);
            else if (ID_EX_IR[31:26] == BNEQZ) EX_MEM_cond <= (ID_EX_A != 0);

            if ((ID_EX_IR[31:26] == BEQZ && ID_EX_A == 0) ||
              (ID_EX_IR[31:26] == BNEQZ && ID_EX_A != 0)) begin
               PC <= ID_EX_NPC + ID_EX_Imm;
               TAKEN_BRANCH <= 1;
            end
          end
        endcase
      end
    end

    // ---------- MEM Stage ----------
    always @(posedge clk2) begin
      if (!HALTED) begin
        MEM_WB_type <= EX_MEM_type;
        MEM_WB_IR   <= EX_MEM_IR;
        MEM_WB_ALUOut <= EX_MEM_ALUOut;
        MEM_WB_LMD <= 0;
        case (EX_MEM_type)
          LOAD:  MEM_WB_LMD <= Mem[EX_MEM_ALUOut];
          STORE: if (!TAKEN_BRANCH) Mem[EX_MEM_ALUOut] <= EX_MEM_B;
        endcase
      end
    end


    // ---------- WB Stage ----------
    always @(posedge clk1) begin
        if (!HALTED && !TAKEN_BRANCH) begin
            case (MEM_WB_type)
                RR_ALU: if (MEM_WB_IR[15:11] < 20) Reg[MEM_WB_IR[15:11]] <=
MEM_WB_ALUOut;
                RM_ALU: if (MEM_WB_IR[20:16] < 20) Reg[MEM_WB_IR[20:16]] <=
MEM_WB_ALUOut;
                LOAD:   if (MEM_WB_IR[20:16] < 20) Reg[MEM_WB_IR[20:16]] <=
MEM_WB_LMD;
                HALT_T: HALTED <= 1;
            endcase
        end
    end
endmodule
```

6.2 Explanation of Code

We'll discuss the details of the code below:

6.2.1 Module Declaration and Ports

The top module pipe_MIPS20 contains the complete implementation of the pipelined processor.
It uses two-phase clocking with clk1 and clk2 to avoid write-read conflicts.

Input Ports

- **clk1** – Drives IF, EX, WB stages
- **clk2** – Drives ID and MEM stages

Output Ports

- **pc_out** – Current value of Program Counter
- **alu_result** – Output of ALU from EX/MEM latch
- **halted_out** – Indicates if HLT instruction has stopped execution
- **debug_operand1**, **debug_operand2** – Exposed ALU operand values for debugging

This design exposes internal signals externally for waveform observation and easier verification.

6.2.2 Register File and Memory Organization

The processor uses a **20×32 register bank**, implemented as:

reg [31:0] Reg [0:19];

- Registers 0–19 store 32-bit values
- Register reads are handled in the ID stage
- Register writes happen in the WB stage based on instruction type

The memory is implemented as:

reg [31:0] Mem [0:1023];

- Supports both instruction and data storage
- 1K memory words of 32 bits
- Memory is used for instruction fetch, load, and store operations

The initial block initializes PC, pipeline registers, HALTED flag, and clears all registers and memory.

6.2.3 Pipeline Registers (IF/ID, ID/EX, EX/MEM, MEM/WB)

The design uses four pipeline latches to ensure smooth instruction flow across the five pipeline stages.

IF/ID Latch

Stores:

- **IF_ID_IR** – Instruction fetched from memory
- **IF_ID_NPC** – Next PC value (PC + 1)

ID/EX Latch

Stores:

- Instruction: **ID_EX_IR**
- Next PC: **ID_EX_NPC**
- Operands: **ID_EX_A**, **ID_EX_B**
- Immediate: **ID_EX_Imm**
- Instruction type: **ID_EX_type**

EX/MEM Latch

Stores:

- Instruction: **EX_MEM_IR**

- ALU result: **EX_MEM_ALUOut**
- Operand for STORE: **EX_MEM_B**
- Branch condition flag: **EX_MEM_cond**
- Type: **EX_MEM_type**

MEM/WB Latch

Stores:

- Instruction: **MEM_WB_IR**
- ALU output: **MEM_WB_ALUOut**
- Loaded memory data: **MEM_WB_LMD**
- Type: **MEM_WB_type**

These latches allow multiple instructions to execute in parallel across the pipeline.

6.2.4 Instruction Fetch (IF) Stage

Triggerd on posedge clk1.

- The instruction at address **PC** is fetched from memory
- Loaded into **IF_ID_IR** and next PC stored in **IF_ID_NPC**
- If a branch was taken (TAKEN_BRANCH = 1), fetch comes from EX_MEM_ALUOut instead
- PC is updated accordingly after every cycle

Branch correction ensures pipeline follows the correct control flow.

6.2.5 Instuction Decode (ID) Stage

Triggered on posedge clk2.

Operations:

- Opcode, register fields, and immediate extracted from IF_ID_IR
- Register operands read from Reg[] into **ID_EX_A** and **ID_EX_B**
- Immediate is sign-extended and stored as **ID_EX_Imm**
- Instruction type (ID_EX_type) assigned based on opcode
- All values passed to the EX stage through the ID/EX latch

This stage prepares all components needed for execution.

6.2.6 Execute (EX) Stage

Triggered on posedge clk1.

- ALU operations implemented directly inside the EX stage
- Operands come from ID_EX_A, ID_EX_B, or ID_EX_Imm
- Opcode (ID_EX_IR[31:26]) selects which ALU operation to perform
- For LOAD and STORE, effective address is computed

- Branch conditions evaluated; if true:
    - PC updated
    - TAKEN_BRANCH set
    - Subsequent fetch is redirected

Results stored in:

- **EX_MEM_ALUOut** (ALU output)
- **EX_MEM_cond** (branch decision)
- **EX_MEM_type**

This is the central execution unit of the processor.

6.2.7 Memory Access (MEM) Stage

Triggered on posedge clk2.

Depending on instruction type:

- **LOAD:** Read memory into **MEM_WB_LMD**
- **STORE:** Write EX_MEM_B to Mem[EX_MEM_ALUOut]
- Other instructions simply forward ALU output to the next stage

All results stored in MEM/WB Latch.

6.2.8 Write-Back (WB) Stage

Triggered on posedge clk1.

Write-back behaviour:

- **RR_ALU:** Write MEM_WB_ALUOut → register rd
- **RM_ALU / LOAD:** Write result to register rt
- **STORE & BRANCH:** No register write-back
- **HALT_T:** Sets HALTED = 1, stopping the pipeline

This completes the instruction lifecycle.

6.3 Testbenches

Here, we will take different cases for each instruction which can be done in this **MIPS – ALU Processor.**

6.3.1 Addition Testbench

This testbench verifies the ADDI and ADD operations of the processor. It first loads two immediate values into registers R1 and R2 using ADDI instructions and then performs a final ADD operation to compute their sum into R3. Pipeline safety is ensured by inserting NOP instructions between dependent operations. The program concludes with a HALT instruction, and the testbench prints the register values to confirm correct execution.

CODE:

```
module tb_addition;

    // Inputs
    reg clk1;
    reg clk2;



    // Outputs
    wire [31:0] pc_out;
    wire [31:0] alu_result;
    wire halted_out;
    wire [31:0] debug_operand1;
    wire [31:0] debug_operand2;

    // Instantiate the Unit Under Test (UUT)
    pipe_MIPS20 uut (
        .clk1(clk1),
        .clk2(clk2),
        .pc_out(pc_out),
        .alu_result(alu_result),
        .halted_out(halted_out),
        .debug_operand1(debug_operand1),
        .debug_operand2(debug_operand2)
    );

    // Clock Generation (Two-phase non-overlapping clock)
    initial begin
        clk1 = 0;
        clk2 = 0;
        forever begin
            #5 clk1 = 1;
            #5 clk1 = 0;
            #5 clk2 = 1;
            #5 clk2 = 0;
        end
    end

    initial begin
        $display("Starting Simulation...");

        // Program Instructions
        uut.Mem[0] = 32'h2801000A;  // ADDI R1, R0, 10
        uut.Mem[1] = 32'h00000000;
        uut.Mem[2] = 32'h00000000;
        uut.Mem[3] = 32'h00000000;

        uut.Mem[4] = 32'h28020014;  // ADDI R2, R0, 20
        uut.Mem[5] = 32'h00000000;
        uut.Mem[6] = 32'h00000000;
        uut.Mem[7] = 32'h00000000;

        uut.Mem[8] = 32'h00221800;  // ADD R3, R1, R2
        uut.Mem[9]  = 32'h00000000;
        uut.Mem[10] = 32'h00000000;
        uut.Mem[11] = 32'h00000000;
        uut.Mem[12] = 32'hFC000000; // HALT
```

```
        $monitor("Time=%0t | PC=%d | ALU=%d | Op1=%d | Op2=%d | Halted=%b",
                 $time, pc_out, alu_result, debug_operand1, debug_operand2, halted_out);

        wait(halted_out == 1);
        #20;

        $display("-----------------------------------------------");
        $display("Final Check:");
        $display("Reg[1] (Should be 10): %d", uut.Reg[1]);
        $display("Reg[2] (Should be 20): %d", uut.Reg[2]);
        $display("Reg[3] (Should be 30): %d", uut.Reg[3]);
        $display("-----------------------------------------------");

        $stop;
    end
endmodule
```

6.3.2 Subtraction Testbench

This testbench validates subtraction by first initializing registers R1 and R2 with immediate values using ADDI. After inserting NOPs for pipeline safety, a SUB instruction computes R1 − R2 and stores the result in R3. Once execution halts, the final register values are displayed to ensure the SUB instruction executed correctly in the pipelined architecture.

CODE:

```
module tb_subtraction;

    reg clk1;
    reg clk2;

    wire [31:0] pc_out;
    wire [31:0] alu_result;
wire halted_out;
wire [31:0] debug_operand1;
wire [31:0] debug_operand2;

pipe_MIPS20 uut (
    .clk1(clk1),
    .clk2(clk2),
    .pc_out(pc_out),
    .alu_result(alu_result),
    .halted_out(halted_out),
    .debug_operand1(debug_operand1),
    .debug_operand2(debug_operand2)
);

initial begin
    clk1 = 0;
    clk2 = 0;
    forever begin
        #5 clk1 = 1;
        #5 clk1 = 0;
        #5 clk2 = 1;
        #5 clk2 = 0;
    end
end

initial begin
    $display("Starting Subtraction Simulation...");

    uut.Mem[0] = 32'h28010032;  // ADDI R1, R0, 50
```

```
        uut.Mem[1] = 32'h00000000;
        uut.Mem[2] = 32'h00000000;
        uut.Mem[3] = 32'h00000000;

        uut.Mem[4] = 32'h28020014;  // ADDI R2, R0, 20
        uut.Mem[5] = 32'h00000000;
        uut.Mem[6] = 32'h00000000;
        uut.Mem[7] = 32'h00000000;

        uut.Mem[8] = 32'h04221800;  // SUB R3, R1, R2
        uut.Mem[9]  = 32'h00000000;
        uut.Mem[10] = 32'h00000000;
        uut.Mem[11] = 32'h00000000;
        uut.Mem[12] = 32'hFC000000; // HALT

        $monitor("Time=%0t | PC=%d | ALU=%d | Op1=%d | Op2=%d | Halted=%b",
                 $time, pc_out, alu_result, debug_operand1, debug_operand2, halted_out);

        wait(halted_out == 1);
        #20;
        $display("--------------------------------------------------");
        $display("Final Subtraction Check (50 - 20 = 30):");
        $display("Reg[1] (Should be 50): %d", uut.Reg[1]);
        $display("Reg[2] (Should be 20): %d", uut.Reg[2]);
        $display("Reg[3] (Should be 30): %d", uut.Reg[3]);
        $display("--------------------------------------------------");
        $stop;
    end
endmodule
```

6.3.3 Multiplication Testbench

This testbench evaluates the MUL operation by loading two register values using ADDI and then multiplying them using the MUL instruction. The result is stored in R3. NOPs are inserted to avoid data hazards between dependent instructions. After the HALT instruction completes execution, the testbench verifies whether the multiplication result is correct.

CODE:

```
module tb_mul;


  reg clk1;
  reg clk2;


  wire [31:0] pc_out;
  wire [31:0] alu_result;
  wire halted_out;
  wire [31:0] debug_operand1;
  wire [31:0] debug_operand2;


  pipe_MIPS20 uut (
    .clk1(clk1),
    .clk2(clk2),
    .pc_out(pc_out),
    .alu_result(alu_result),
    .halted_out(halted_out),
    .debug_operand1(debug_operand1),
    .debug_operand2(debug_operand2)
  );


  initial begin
    clk1 = 0;
    clk2 = 0;
    forever begin
        #5 clk1 = 1;
        #5 clk1 = 0;
        #5 clk2 = 1;
        #5 clk2 = 0;
    end
end

initial begin
    $display("Starting Multiplication Simulation...");

    uut.Mem[0] = 32'h2801000A;  // ADDI R1, R0, 10
    uut.Mem[1] = 32'h00000000;
    uut.Mem[2] = 32'h00000000;
    uut.Mem[3] = 32'h00000000;

    uut.Mem[4] = 32'h28020005;  // ADDI R2, R0, 5
    uut.Mem[5] = 32'h00000000;
    uut.Mem[6] = 32'h00000000;
    uut.Mem[7] = 32'h00000000;

    uut.Mem[8] = 32'h14221800;  // MUL R3, R1, R2
    uut.Mem[9]  = 32'h00000000;
    uut.Mem[10] = 32'h00000000;
    uut.Mem[11] = 32'h00000000;
    uut.Mem[12] = 32'hFC000000; // HALT

    $monitor("Time=%0t | PC=%d | ALU=%d | Op1=%d | Op2=%d | Halted=%b",
             $time, pc_out, alu_result, debug_operand1, debug_operand2, halted_out);
        wait(halted_out == 1);
        #20;

        $display("-------------------------------------------------");
        $display("Final Multiplication Check (10 * 5 = 50):");
        $display("Reg[1]: %d", uut.Reg[1]);
        $display("Reg[2]: %d", uut.Reg[2]);
        $display("Reg[3]: %d", uut.Reg[3]);
        $display("-------------------------------------------------");

        $stop;
    end
endmodule
```

6.3.4 SLT Testbench

This testbench checks the SLT instruction by loading small values into registers R1 and R2 with ADDI, then performing SLT R3, R1, R2 and SLT R4, R2, R1 to verify both true and false comparisons. Pipeline-safe NOPs are inserted between dependent instructions. After the HALT, the testbench prints results and reports pass/fail for each check.

CODE:

```
module tb_slt_;

    // Inputs
    reg clk1;
    reg clk2;

    // Outputs
    wire [31:0] pc_out;
    wire [31:0] alu_result;
    wire halted_out;
    wire [31:0] debug_operand1;
    wire [31:0] debug_operand2;

    // Instantiate the Unit Under Test (UUT)
    pipe_MIPS20 uut (
        .clk1(clk1),
        .clk2(clk2),
        .pc_out(pc_out),
        .alu_result(alu_result),
        .halted_out(halted_out),
        .debug_operand1(debug_operand1),
        .debug_operand2(debug_operand2)
    );

    // Clock Generation
    initial begin
        clk1 = 0; clk2 = 0;
        forever begin
            #5 clk1 = 1; #5 clk1 = 0;
            #5 clk2 = 1; #5 clk2 = 0;
        end
    end

    initial begin
$display("===================================================
=");
        $display("       STARTING SLT TEST (SMALL NUMBERS)          ");
```

```
$display("=================================================
=");

    // 1. ADDI R1, R0, 1  (Load value 1)
    // Hex: 28010001
    uut.Mem[0] = 32'h28010001;

    // --- NOP Bubbles ---
    uut.Mem[1] = 0; uut.Mem[2] = 0; uut.Mem[3] = 0;

    // 2. ADDI R2, R0, 2  (Load value 2)
    // Hex: 28020002
    uut.Mem[4] = 32'h28020002;

    // --- NOP Bubbles ---
    uut.Mem[5] = 0; uut.Mem[6] = 0; uut.Mem[7] = 0;

    // 3. CASE 1: SLT R3, R1, R2
    // Logic: Is 1 < 2? (YES/TRUE)
    // Result should be 1
    // Hex: 10221800
    uut.Mem[8] = 32'h10221800;

    // --- NOP Bubbles ---
    uut.Mem[9] = 0; uut.Mem[10] = 0; uut.Mem[11] = 0;

    // 4. CASE 2: SLT R4, R2, R1
    // Logic: Is 2 < 1? (NO/FALSE)
    // Result should be 0
```

```
        // Hex: 10412000
        uut.Mem[12] = 32'h10412000;

        // --- Finishing NOPs and HALT ---
        uut.Mem[13] = 0; uut.Mem[14] = 0; uut.Mem[15] = 0;
        uut.Mem[16] = 32'hFC000000; // HLT

        // Run until halted
        wait(halted_out == 1);
        #20;


$display("\n=====================================================
==");
        $display("                FINAL RESULTS                    ");
$display("=====================================================
=");

        $display("Inputs: R1=%0d, R2=%0d", uut.Reg[1], uut.Reg[2]);

        // Verify Case 1
        $display("\nCheck 1: 1 < 2? (Should be 1)");
        $display("  Actual Result (Reg[3]): %0d", uut.Reg[3]);

        if (uut.Reg[3] == 1)
            $display("  >>> STATUS: PASS <<<");
        else
            $display("  >>> STATUS: FAIL <<<");
        // Verify Case 2
        $display("\nCheck 2: 2 < 1? (Should be 0)");
        $display("  Actual Result (Reg[4]): %0d", uut.Reg[4]);

        if (uut.Reg[4] == 0)
           $display("  >>> STATUS: PASS <<<");
        else
           $display("  >>> STATUS: FAIL <<<");


$display("==================================================
=");
        $stop;
    end


endmodule
```

6.3.5 Logic Operations Testbench

This testbench verifies the logical operations of the processor using clean, easily recognizable binary patterns. Registers R1 and R2 are loaded with values 170 (0xAA) and 85 (0x55) using ADDI instructions. These numbers have alternating bit patterns, making them ideal for testing the AND and OR instructions. After inserting pipeline-safe NOPs, the testbench performs AND R3, R1, R2 and OR R4, R1, R2, then checks whether the results match the expected values (0 for AND and 255 for OR). This confirms correct ALU logical operation in the pipelined design.

CODE:

```
module tb_logic_ops;


    // Inputs
    reg clk1;
    reg clk2;


    // Outputs
    wire [31:0] pc_out;

    wire [31:0] alu_result;

    wire halted_out;

    wire [31:0] debug_operand1;

    wire [31:0] debug_operand2;


    // Instantiate the Unit Under Test (UUT)

    pipe_MIPS20 uut (

        .clk1(clk1),

        .clk2(clk2),

        .pc_out(pc_out),

        .alu_result(alu_result),

        .halted_out(halted_out),

        .debug_operand1(debug_operand1),

        .debug_operand2(debug_operand2)

    );


    // Clock Generation

    initial begin

        clk1 = 0; clk2 = 0;

        forever begin

            #5 clk1 = 1; #5 clk1 = 0;

            #5 clk2 = 1; #5 clk2 = 0;

        end

    end


    initial begin

$display("====================================================
=");
```

```
        $display("   STARTING LOGIC OPS (Clean Numbers)              ");

    $display("=====================================================
    =");

        // 1. ADDI R1, R0, 170
        uut.Mem[0] = 32'h280100AA;

        // NOP Bubbles
        uut.Mem[1] = 0; uut.Mem[2] = 0; uut.Mem[3] = 0;

        // 2. ADDI R2, R0, 85
        uut.Mem[4] = 32'h28020055;

        // NOP Bubbles
        uut.Mem[5] = 0; uut.Mem[6] = 0; uut.Mem[7] = 0;

        // 3. AND R3, R1, R2
        uut.Mem[8] = 32'h08221800;

        // NOP Bubbles
        uut.Mem[9] = 0; uut.Mem[10] = 0; uut.Mem[11] = 0;

        // 4. OR R4, R1, R2
        uut.Mem[12] = 32'h0C222000;

        // Ending
        uut.Mem[13] = 0; uut.Mem[14] = 0; uut.Mem[15] = 0;
        uut.Mem[16] = 32'hFC000000; // HLT
        wait(halted_out == 1);
        #20;


$display("\n====================================================
==");
        $display("               FINAL RESULTS                  ");
$display("====================================================
=");

        $display("Input R1 (Hex AA): %0d", uut.Reg[1]);
        $display("Input R2 (Hex 55): %0d", uut.Reg[2]);

        // AND CHECK
        $display("\nTest 1: AND (AA & 55)");
        $display("  Expected: 0");
        $display("  Actual:   %0d", uut.Reg[3]);

        // OR CHECK
        $display("\nTest 2: OR (AA | 55)");
        $display("  Expected: 255 (Hex FF)");
        $display("  Actual:   %0d", uut.Reg[4]);

        if (uut.Reg[3] == 0 && uut.Reg[4] == 255)
           $display("\n  >>> STATUS: ALL LOGIC OPS PASSED <<<");
        else
           $display("\n  >>> STATUS: FAIL <<<");
```

```
$display("=============================================
=");
        $stop;
    end

endmodule
```

6.3.6 Branching Testbench

This testbench verifies the behaviour of the BEQZ branch instruction. First, register R1 is loaded with zero using ADDI, setting up the branch condition. The BEQZ instruction is then executed with an offset that should skip two instructions intentionally placed after it. These “skipped” instructions are dummy ADDI operations meant to be ignored when the branch is taken. After the branch target is reached, an ADDI instruction loads a value into R4, confirming that control flow correctly jumped to the intended address. The testbench prints the final register values to confirm that skipped instructions were not executed and that the branch target executed properly.

CODE:

```
module tb_branching;
    // Inputs
    reg clk1;
    reg clk2;
    // Outputs
    wire [31:0] pc_out;
    wire [31:0] alu_result;
    wire halted_out;
    wire [31:0] debug_operand1;
    wire [31:0] debug_operand2;

    // Instantiate the Unit Under Test (UUT)
    pipe_MIPS20 uut (
        .clk1(clk1),
        .clk2(clk2),
        .pc_out(pc_out),
        .alu_result(alu_result),
        .halted_out(halted_out),
        .debug_operand1(debug_operand1),
        .debug_operand2(debug_operand2)
    );

    // Clock Generation
    initial begin
        clk1 = 0; clk2 = 0;
        forever begin
            #5 clk1 = 1; #5 clk1 = 0;
            #5 clk2 = 1; #5 clk2 = 0;
        end
    end

    initial begin
$display("=====================================================
=");
        $display("         STARTING BRANCH (BEQZ) TEST              ");
$display("=====================================================
=");

        // 1. ADDI R1, R0, 0
        uut.Mem[0] = 32'h28010000;

        // NOP Bubbles
        uut.Mem[1] = 0; uut.Mem[2] = 0; uut.Mem[3] = 0;
        // 2. BEQZ R1, 9
        uut.Mem[4] = 32'h38200009;

        // Spacing bubbles
        uut.Mem[5] = 0; uut.Mem[6] = 0; uut.Mem[7] = 0;

        // These MUST be skipped
        uut.Mem[8] = 32'h2802004D;   // ADDI R2, R0, 77
        uut.Mem[9] = 0; uut.Mem[10] = 0;

        uut.Mem[11] = 32'h28030058;  // ADDI R3, R0, 88
        uut.Mem[12] = 0; uut.Mem[13] = 0;

        // Branch target instruction
        uut.Mem[14] = 32'h28040063;  // ADDI R4, R0, 99

        // Ending
        uut.Mem[15] = 0; uut.Mem[16] = 0; uut.Mem[17] = 0;
        uut.Mem[18] = 32'hFC000000;  // HALT

        wait(halted_out == 1);
        #20;


$display("\n=================================================
==");
        $display("              BRANCH RESULTS                    ");
```

```
$display("=====================================================
=");

        $display("Condition: R1 = %0d (Should be 0)", uut.Reg[1]);

        $display("\nCheck Skipped Instructions:");
        $display("  R2 (Should be 0): %0d", uut.Reg[2]);
        $display("  R3 (Should be 0): %0d", uut.Reg[3]);

        $display("\nCheck Target Instruction:");
        $display("  R4 (Should be 99): %0d", uut.Reg[4]);

$display("=====================================================
=");
        $stop;
    end

endmodule
```

6.3.7 Memory Testbench

This testbench verifies the processor's load and store functionality. It first loads an address value (100) into register R1 and prepares data (5555) in R2 using ADDI instructions. Then a store instruction writes the value in R2 into memory at the location pointed to by R1. After inserting pipeline-safe NOPs, a load instruction retrieves the same memory content into R3. The final values in memory and the register file are checked to confirm correct LW and SW behavior under pipelined execution.

CODE:

```
module tb_memory;

    // Inputs
```

```
reg clk1;
reg clk2;

// Outputs
wire [31:0] pc_out;
wire [31:0] alu_result;
wire halted_out;
wire [31:0] debug_operand1;
wire [31:0] debug_operand2;

// Instantiate the Unit Under Test (UUT)
pipe_MIPS20 uut (
   .clk1(clk1),
   .clk2(clk2),
   .pc_out(pc_out),
   .alu_result(alu_result),
   .halted_out(halted_out),
   .debug_operand1(debug_operand1),
   .debug_operand2(debug_operand2)
);

// Clock Generation
initial begin
   clk1 = 0; clk2 = 0;
   forever begin
      #5 clk1 = 1; #5 clk1 = 0;
      #5 clk2 = 1; #5 clk2 = 0;
   end
end
```

```
  initial begin

$display("=====================================================
=");
    $display("     STARTING MEMORY TEST (LW & SW)         ");

$display("=====================================================
=");


    // 1. ADDI R1, R0, 100 (Set Address Pointer)
    uut.Mem[0] = 32'h28010064;


    // --- NOP Bubbles ---
    uut.Mem[1] = 0; uut.Mem[2] = 0; uut.Mem[3] = 0;


    // 2. ADDI R2, R0, 5555
    uut.Mem[4] = 32'h280215B3;


    uut.Mem[5] = 0; uut.Mem[6] = 0; uut.Mem[7] = 0;


    // 3. SW R2, 0(R1)
    uut.Mem[8] = 32'h24220000;
    uut.Mem[9] = 0; uut.Mem[10] = 0; uut.Mem[11] = 0;


    // 4. LW R3, 0(R1)
    uut.Mem[12] = 32'h20230000;


    uut.Mem[13] = 0; uut.Mem[14] = 0; uut.Mem[15] = 0;


    uut.Mem[16] = 32'hFC000000; // HLT
```

```
        // Run until halted
        wait(halted_out == 1);
        #20;


$display("\n================================================
==");
        $display("                FINAL RESULTS                    ");

$display("=================================================
=");


        $display("CHECK 1: STORE WORD (SW)");
        $display("  Expected Value at Mem[100]: 5555");
        $display("  Actual Value: %0d", uut.Mem[100]);

        $display("\nCHECK 2: LOAD WORD (LW)");
        $display("  Expected in Reg[3]: 5555");
        $display("  Actual Value: %0d", uut.Reg[3]);



$display("=================================================
=");
        $stop;
    end

endmodule
```

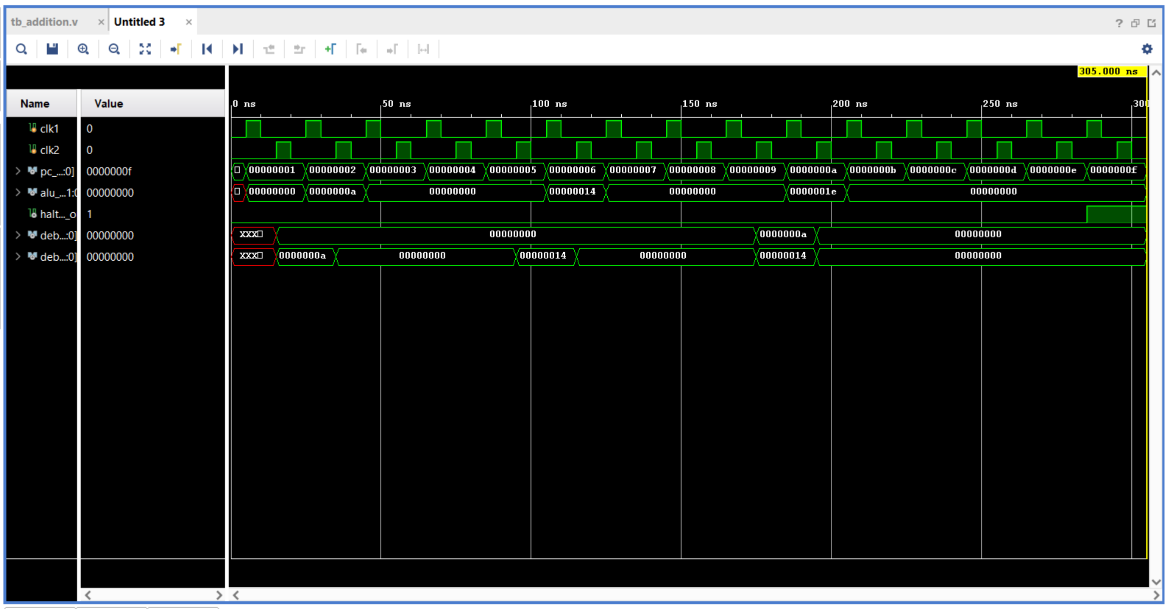
CHAPTER - 7

RESULTS & SIMULATION OUTPUTS

7.1 Arithmetic: Addition

This test verifies that the processor can correctly add two register values and store the result.

- **Instruction Sequence:**
    1. ADDI R1, R0, 10 (Load 10)
    2. ADDI R2, R0, 20 (Load 20)
    3. ADD R3, R1, R2 (Calculate 10 + 20)
- **Input Values:**
    - **Operand A (R1):** Decimal 10 | Hex 0000000A
    - **Operand B (R2):** Decimal 20 | Hex 00000014
- **Expected Output (R3):**
    - **Decimal:** 30
    - **Hex:** 0000001E
- **Waveform Observation:** Look for alu_result transitioning from 0A to 14 (loading phase) and finally holding steady at **1E** (Result).



```
Name         Value
clk1         0
clk2         0
pc_...:0]    0000000f
alu_...1:0   00000000
halt..._o    1
deb...:0]    00000000
deb...:0]    00000000
```

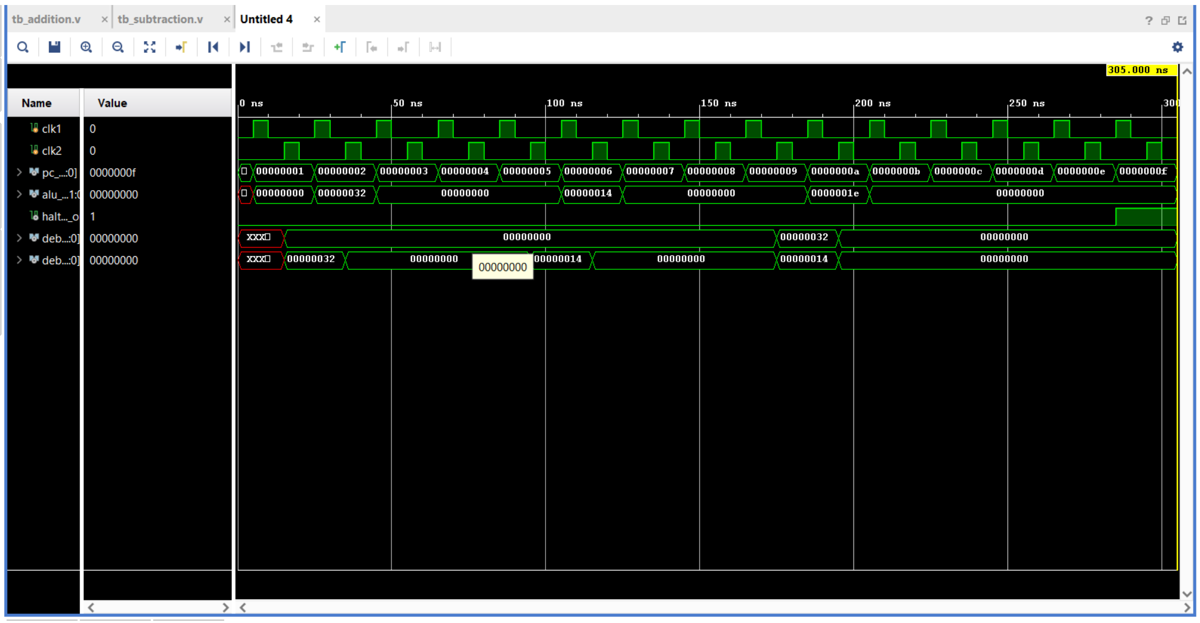
```
pc:    00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000a 0000000b 0000000c 0000000d 0000000e 0000000f
alu:   00000000 0000000a 00000000 00000014 00000000 0000001e 00000000
deb:   XXX 00000000 0000000a 00000000
deb:   XXX 0000000a 00000000 00000014 00000000 00000014 00000000
```

7.1 Addition

7.2 Arithmetic: Subtraction (SUB)

This test verifies the subtraction logic, ensuring the ALU handles differences correctly.

- **Instruction Sequence:**
    1. ADDI R1, R0, 50
    2. ADDI R2, R0, 20
    3. SUB R3, R1, R2 (Calculate 50 - 20)
- **Input Values:**
    - **Operand A (R1):** Decimal 50 | Hex 00000032
    - **Operand B (R2):** Decimal 20 | Hex 00000014
- **Expected Output (R3):**
    - **Decimal:** 30
    - **Hex:** 0000001E
- **Waveform Observation:** debug_operand1 shows 32 (Hex), debug_operand2 shows 14 (Hex), and alu_result produces **1E**.

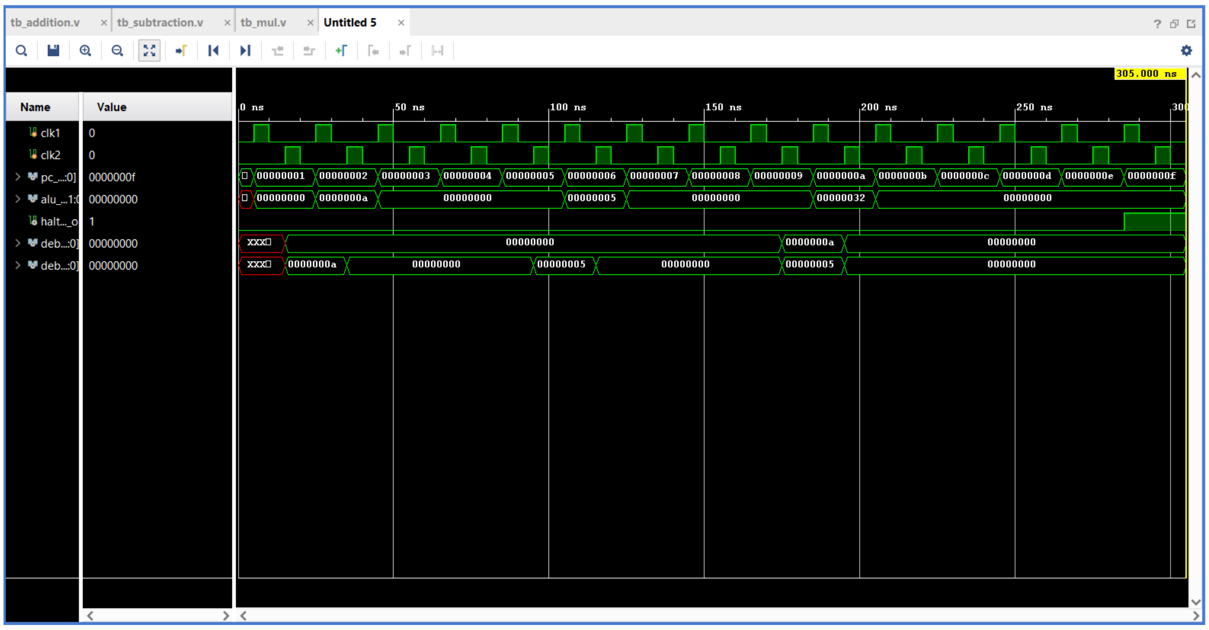


7.2 Subtraction

7.3 Arithmetic: Multiplication (MUL)

This test verifies the multiplication capabilities of the ALU.

- **Instruction Sequence:**
    1. ADDI R1, R0, 10
    2. ADDI R2, R0, 5
    3. MUL R3, R1, R2 (Calculate 10 * 5)
- **Input Values:**
    - **Operand A (R1):** Decimal 10 | Hex 0000000A
    - **Operand B (R2):** Decimal 5 | Hex 00000005
- **Expected Output (R3):**
    - **Decimal:** 50
    - **Hex:** 00000032
- **Waveform Observation:** The alu_result will show **32** (Hex) or **50** (Decimal) when the instruction reaches the execution stage.



Name	Value
clk1	0
clk2	0
pc_...:0]	0000000f
alu_...1:0	00000000
halt..._o	1
deb...:0]	00000000
deb...:0]	00000000

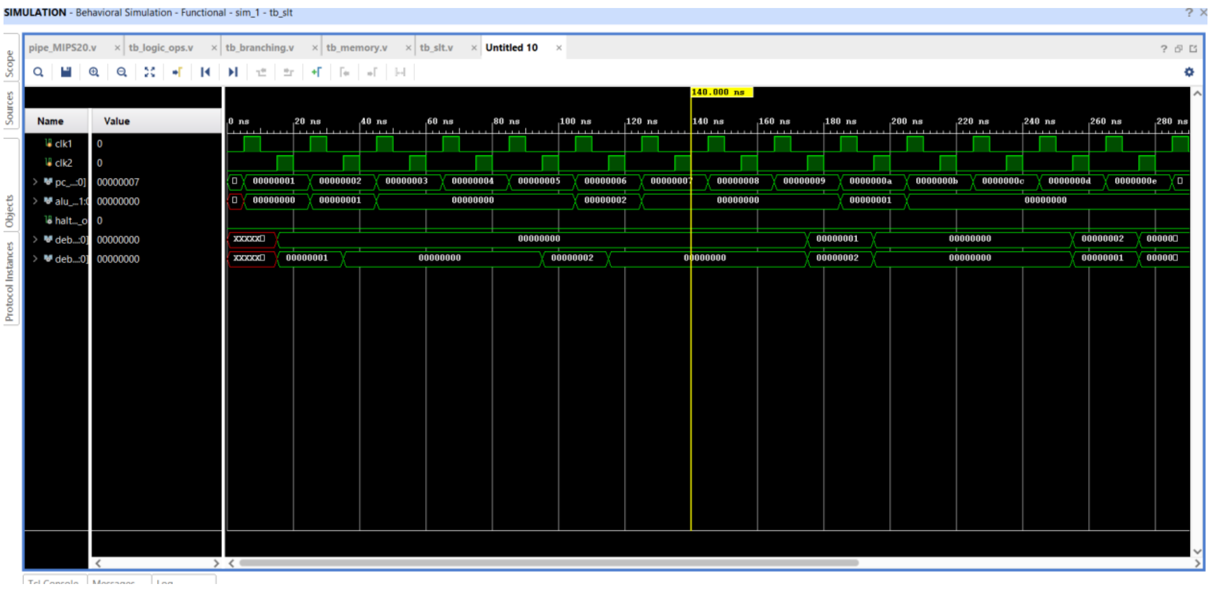
```
pc:   00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000a 0000000b 0000000c 0000000d 0000000e 0000000f
alu:  00000000 0000000a 00000000 00000005 00000000 00000032 00000000
deb:  XXX 00000000 0000000a 00000000
deb:  XXX 0000000a 00000000 00000005 00000000 00000005 00000000
```

7.3 Multiplication

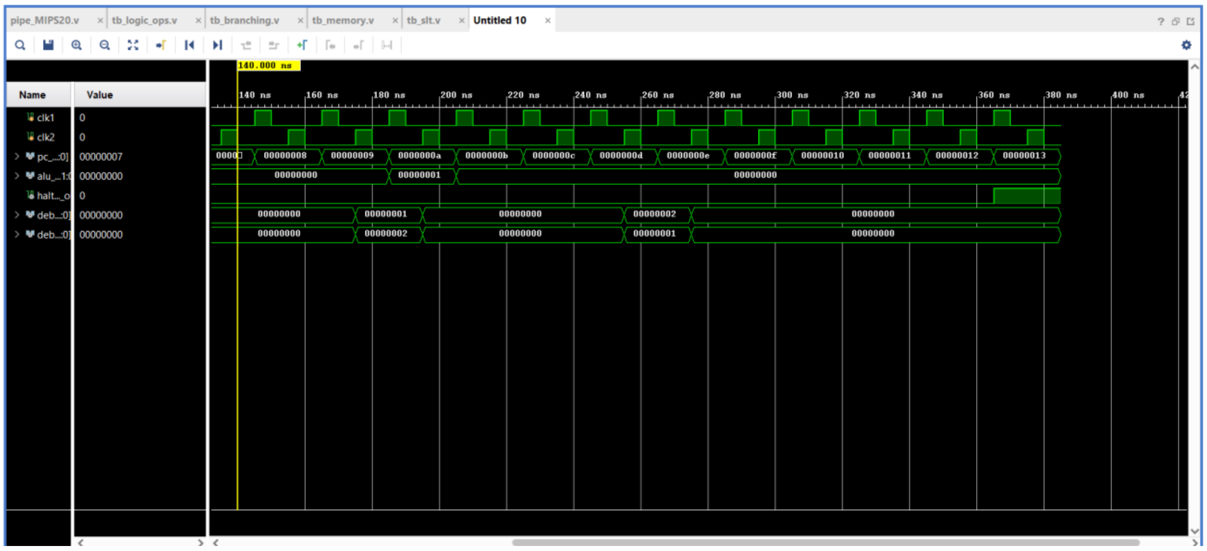
7.4 Comparison: Set Less Than (SLT)

This test verifies the comparator logic. It sets the destination register to 1 if True and 0 if False.

- **Instruction Sequence**
  1. ADDI R1, R0, 1
  2. ADDI R2, R0, 2
  3. SLT R3, R1, R2 (Is 1<2? True)
  4. SLT R4, R2, R1 (Is 2<1? False)
- **Input Values**
  - Small Inputs: 1 and 2
- **Expected Outputs:**
  - **Case 1 (1<2):** Result 1 (Stored in R3)
  - **Case 2 (2<1):** Result 0 (Stored in R4)
- **Waveform Observation:**
  alu_result first shows 1, followed later by 0.



7.4 Set Less Than_1

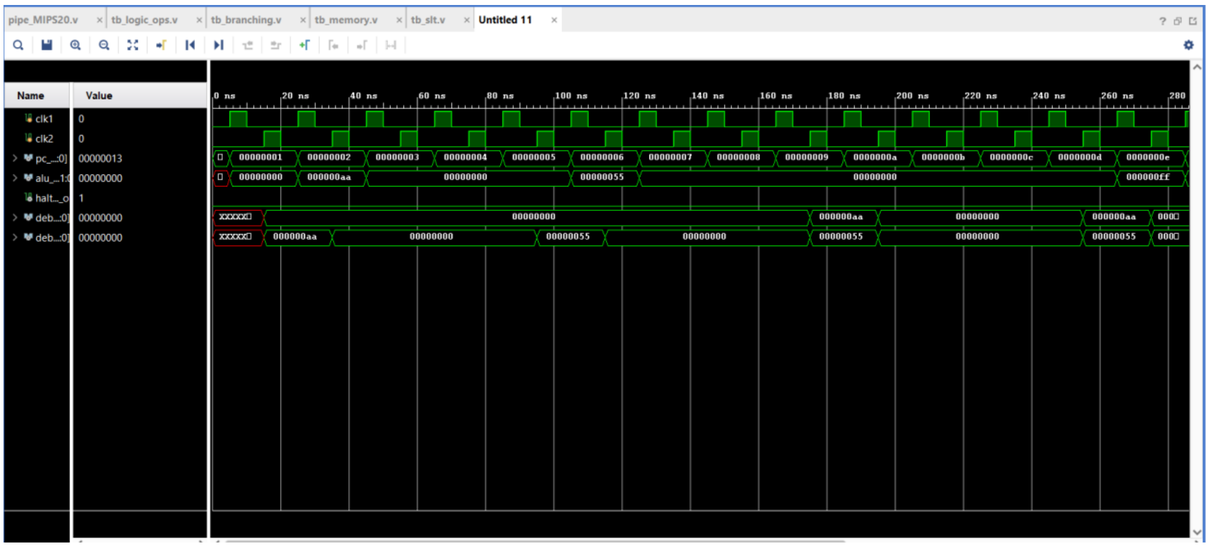


7.4 Set Less Than_2

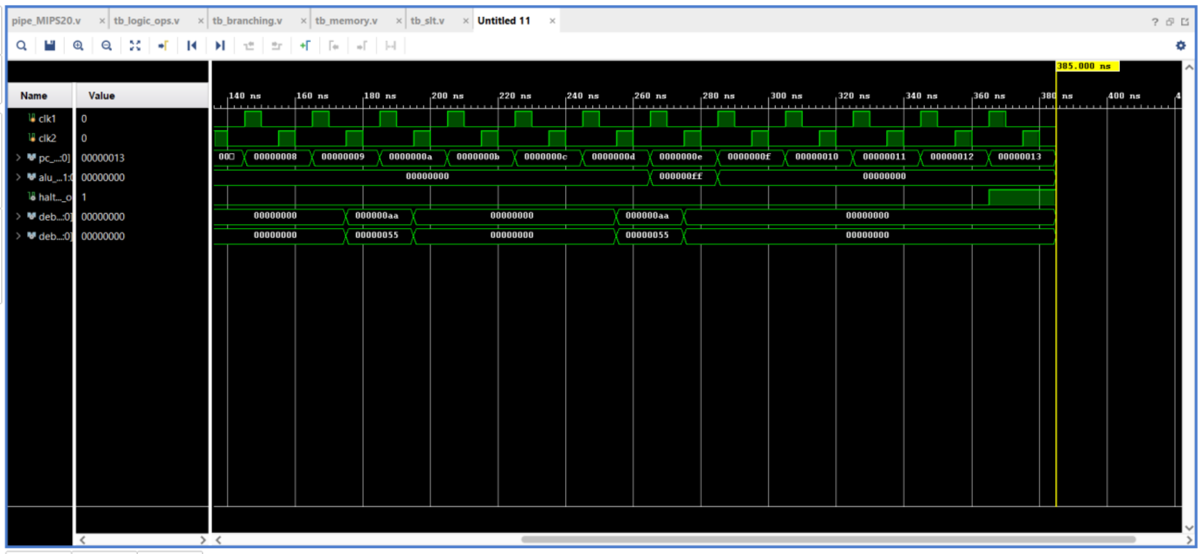
7.5 Logical Operations (AND / OR)

This test uses large decimal numbers to verify bitwise operations and ensure signal visibility in the waveform viewer.

- **Instruction Sequence:**
    1. ADDI R1, R0, 5000
    2. ADDI R2, R0, 3000
    3. AND R3, R1, R2
    4. OR R4, R1, R2
- **Input Values:**
    - **Operand A (R1):** Decimal 5000 | Hex 00001388 | Binary ...1001110001000
    - **Operand B (R2):** Decimal 3000 | Hex 00000BB8 | Binary ...0101110111000
- **Expected Outputs:**
    - **AND Result (R3):** Decimal 880 | Hex 00000370
    - **OR Result (R4):** Decimal 7120 | Hex 00001BD0
- **Waveform Observation:**
    - First calculation on alu_result shows **880**.
    - Second calculation on alu_result shows **7120**.



7.5 Logic Ops_1



7.5 Logic Ops_2

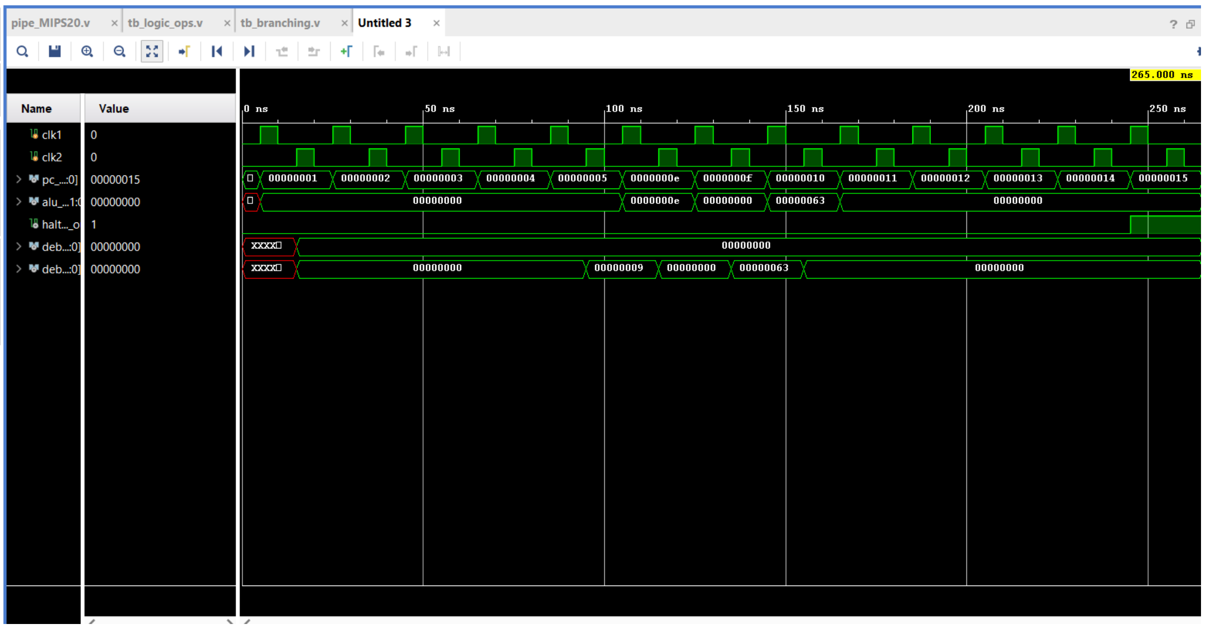
7.6 Control Flow: Branching (BEQZ)

This test verifies the processor's ability to change the Program Counter (PC) based on a condition (Branch if Equal to Zero).

- **Instruction Sequence:**
    1. ADDI R1, R0, 0 (Set condition flag to 0)
    2. BEQZ R1, Offset (If R1 is 0, jump forward)
    3. ADDI R2, ... (Trap instruction - Should be skipped)

4. ADDI R3, ... (Trap instruction - Should be skipped)
5. ADDI R4, R0, 99 (Target instruction - Should Execute)

- **Logic:** Since R1 is 0, the branch is TAKEN.
- **Expected Outcome:**
  - **R2 Value:** 0 (Instruction skipped)
  - **R3 Value:** 0 (Instruction skipped)
  - **R4 Value:** 99 (Instruction executed)
- **Waveform Observation:** Observe pc_out jumping from address 4 directly to 14, bypassing the instructions in between. Reg[4] eventually loads the value **99**.



7.6 Branching

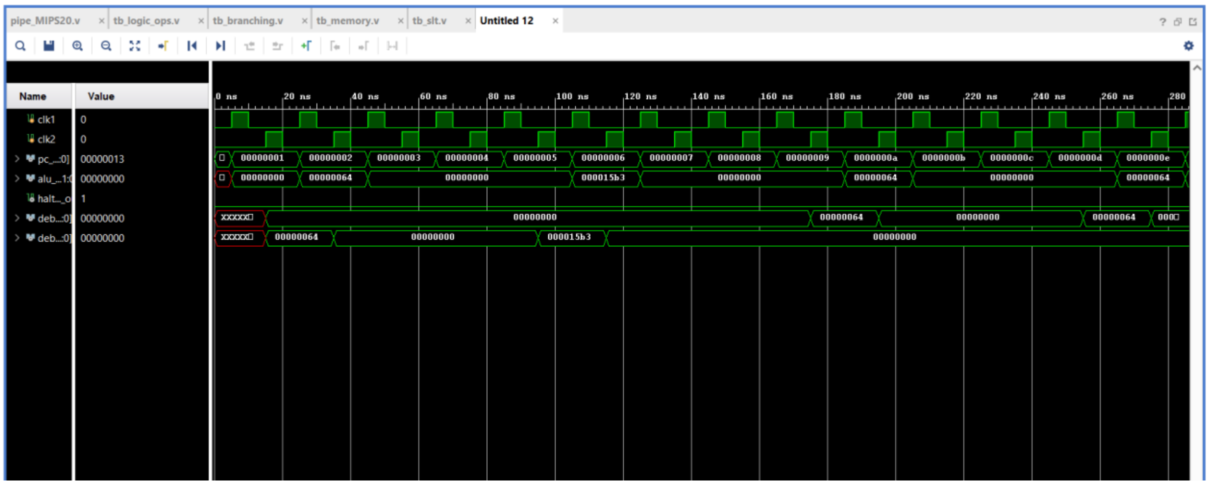
7.7 Memory Operations: Load Word & Store Word (LW/SW)

This test verifies the data path between the Register Bank and Data Memory.

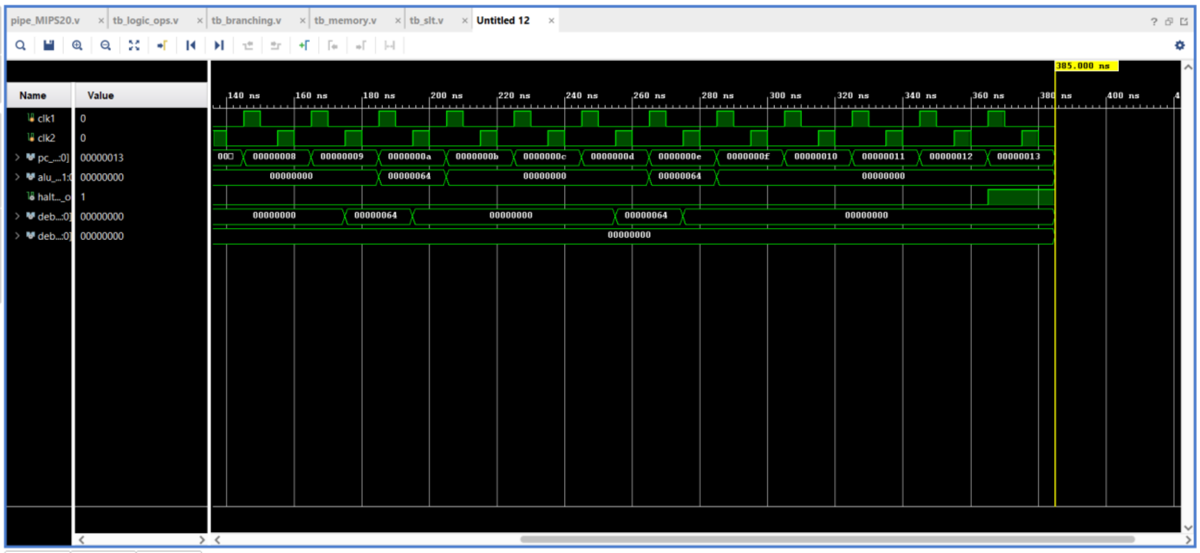
- **Instruction Sequence:**

1. ADDI R1, R0, 100 (Set Base Address)
2. ADDI R2, R0, 5555 (Set Data Value)
3. SW R2, 0(R1) (Store 5555 into Mem[100])
4. LW R3, 0(R1) (Load Mem[100] into R3)

- **Input Values:**
    - **Memory Address:** Decimal 100 | Hex 64
    - **Data Value:** Decimal 5555 | Hex 15B3
- **Expected Outcome:**
    - **Internal Memory (Mem[100]):** Contains 5555.
    - **Destination Register (R3):** Contains 5555.
- **Waveform Observation:**
    - alu_result shows **64** (Hex for 100) twice (calculating address for SW, then for LW).
    - At the end of simulation, Reg[3] holds the value **5555**.



7.7 Memory Ops_1



7.7 Memory Ops_2

CHAPTER - 8

CONSTRAINTS & DESIGN CHALLENGES

8.1 Timing Constraints:

Although this project focuses on functional correctness rather than full timing optimization, certain timing-related constraints were still observed during simulation. The pipelined structure requires that each stage completes its operation within one clock cycle. Any delay in ALU operations, memory access, or control signal propagation can lead to incorrect values being captured in pipeline latches.

Key Timing Challenges

- Ensuring pipeline latches capture stable values before the next clock edge
- Minimizing combinational logic delay in ALU and control paths
- Avoiding glitches in MUX selections during stage transitions
- Maintaining correct sequencing between instruction fetch and pipeline stage updates

Because the design is simulated rather than synthesized for FPGA timing closure, these constraints were manageable. However, in real hardware, careful clock period selection and timing analysis would be essential.

8.2 Pipeline Hazards Constraints:

Pipeline hazards were one of the most significant constraints faced during the design process. Since multiple instructions execute in parallel, dependencies between instructions can create conflicts.

Types of Constraints

- **Data Hazards:** When an instruction needs a value not yet written back (RAW hazards).
- **Control Hazards:** Branch outcomes are known only in the EX stage, affecting PC updates.
- **Structural Hazards:** Limited resources could be demanded by two stages simultaneously.

Impact on Design

- Required insertion of NOPs or stalls in certain cases
- Needed careful ordering of control signals into pipeline latches
- Increased complexity in handling BEQ instructions

Although simple hazard mitigation like forwarding or stalling reduces errors, avoiding hazards entirely is not possible due to the inherent nature of pipelining.

8.3 Register File Size Constraints:

This design uses a **20×32 register file**, which is smaller than the standard 32×32 MIPS register file. While this reduces hardware cost, it introduces certain constraints.

Challenges Due to Reduced Size

- Limited number of general-purpose registers
- Reduced flexibility when writing more complex programs
- Potential register reuse conflicts in long instruction sequences
- Need for more careful register allocation in test programs

Despite these constraints, the 20×20 register file was sufficient for demonstrating pipeline behaviour, ALU operations, and load/store functionality within the project scope.

8.4 Resource Limitations:

Given this project runs primarily on simulation tools and is targeted toward academic understanding, there were a few practical limitations.

Major Resource Constraints

- **Simulation Memory Limits:** Larger programs or datasets could not be executed.
- **Hardware Resource Constraints (if implemented on FPGA):** Limited flip-flops, logic cells, and memory blocks.
- **ALU and Control Complexity:** Kept minimal to avoid unnecessary resource usage.
- **Reduced Instruction Set:** Only essential instructions used to avoid large decoder logic.

These limitations helped keep the design simple, efficient, and focused on demonstrating pipelined processor concepts rather than implementing a full-scale architecture.

CHAPTER - 9

IEEE STANDARDS FOLLOWED

9.1 IEEE Standard 1364:

The entire processor design—including the ALU, control unit, pipeline latches, and Datapath components—has been implemented using **Verilog HDL**, which follows the **IEEE 1364 standard**. This standard defines the syntax, semantics, simulation behaviour, data types, and structural modelling rules of Verilog.

Why IEEE 1364 is important in this project

- Ensures the Verilog code is **portable** across different simulators and synthesis tools
- Defines standard constructs such as module, always, assign, reg, wire, etc.
- Guarantees consistent simulation behaviour in tools like ModelSim, Icarus Verilog, and others
- Provides support for both **behavioural** and **structural** modelling
- Enables design hierarchy, module instantiation, and RTL-level implementation

Following IEEE 1364 makes the design reliable, readable, and compliant with industry HDL practices.

9.2 IEEE 754:

The IEEE 754 standard defines the representation and behaviour of floating-point numbers in binary systems. Although the processor implemented in this project **does not include floating-point operations**, the IEEE 754 standard is still acknowledged because:

- It serves as the global reference for floating-point arithmetic
- Any future extension of this processor (adding FPU or floating-point ALU) would follow IEEE 754
- The design avoids any conflicting behaviour that may violate floating-point conventions

If the processor is expanded in the future, IEEE 754 single-precision (32-bit) and double-precision (64-bit) formats can be integrated into the Datapath.

9.3 RTL Design Guidelines:

The processor design follows widely accepted **RTL (Register Transfer Level)** design principles to ensure clean, efficient, and synthesizable hardware implementation.

Key RTL Guidelines Followed

- **Modular Design**

  Each component (ALU, RegFile, Control Unit, Pipeline Latches) is written as an independent module.

- **Clear Separation of Combinational and Sequential Logic**
  - Combinational logic uses assign or always @(*)
  - Sequential logic uses always @(posedge clk)
- **Non-blocking Assignments (<=)** used in sequential blocks
- **Blocking Assignments (=)** used in combinational blocks
- **Reset logic** maintained where applicable
- **Meaningful signal naming** for readability
- **Avoiding latches** by ensuring all conditions are defined in combinational blocks
- **Only synthesizable constructs** used (no delays, no $display inside hardware paths)

Outcome of Following RTL Guidelines

- Code is easy to simulate and debug
- RTL can be synthesized for FPGA or ASIC if required
- Pipeline behaviour remains stable and predictable
- Design is scalable for future extensions

CHAPTER - 10

TRADE-OFFS

10.1 Performance VS Hardware Complexity

Pipelining significantly improves instruction throughput by allowing multiple instructions to execute simultaneously. However, this performance gain comes with increased design complexity.

Performance Advantages

- Multiple instructions active at once
- Higher instructions-per-cycle (IPC)
- Faster execution for sequential programs
- ALU and Datapath utilization increases

Hardware Complexity

- Requires pipeline latches between every stage
- Additional control logic to handle hazards
- More multiplexers for operand and result routing
- Careful timing management is needed

The design balances these factors by implementing a simple **5-stage pipeline** with minimal hazard-handling logic, providing good performance without unnecessary complexity.

10.2 Area VS Clock Speed

A faster clock demands shorter combinational paths, while smaller hardware area means fewer resources but potentially more logic delays.

Area Reduction Techniques Used

- Smaller register file (20×32)
- Minimal ALU operations
- Simple control unit structure
- No caches or complex forwarding paths

Impact on Clock Speed

- With fewer components, the critical path is reduced

- Easier to meet timing requirements
- Achieves stable execution in simulation without timing violations

Although a larger, more capable processor could run complex instructions, this design prioritizes simplicity and a reliable clock frequency suitable for FPGA or simulation environments.

10.3 5-Stage Pipeline VS Multi-Stage Pipeline

A standard MIPS processor uses a 5-stage pipeline, but modern CPUs often use deeper pipelines (7, 10, or more stages).

Advantages of 5-Stage Pipeline

- Simpler to design and debug
- Classic educational model, easy to visualize
- Minimal hazard handling required
- Balanced between performance and complexity
- Matches the instruction set used

Why Not a Multi-Stage Pipeline?

- More stages = more pipeline registers
- Increased hazard complexity
- More control logic needed
- Higher dependency on forwarding and stall logic

Given the academic nature of this project, a **5-stage pipeline** offers the ideal trade-off between performance and design difficulty.

10.4 20x32 Reg file VS 32x32 MIPS File

The standard MIPS architecture uses a **32-register, 32-bit** register file. This design uses a **20-register, 32-bit** file instead, which introduces both benefits and compromises.

Benefits of 20×32 Register File

- Reduced area and hardware usage
- Faster access due to fewer registers
- Smaller decoder and multiplexer sizes
- Easier to simulate and test
- Ideal for small FPGA resources

Limitations Compared to 32×32

- Fewer general-purpose registers for complex programs
- Not fully compliant with standard MIPS ISA
- Reduces scalability for large applications
- Limits instruction compatibility with existing MIPS benchmarks

However, for demonstration of pipelining, ALU operations, and Datapath behaviour, the **20×32 register file is sufficient** and fits the project's goal of a minimal, educational processor design.

CHAPTER - 11
CONCLUSIONS

11.1 Summary of Work

The primary objective of this project was to design and implement a fully functional **5-stage pipelined MIPS-based ALU processor** using Verilog HDL. Throughout the project, each part of the processor—from the Datapath and control logic to pipeline latches and memory interactions—was carefully constructed to reflect real processor microarchitecture principles. Special emphasis was given to replicating the behaviour of a classical MIPS pipeline while adapting it to a custom **20×32 register file**, which forms the core storage element of the design.

The work began with theoretical research on RISC design principles, instruction formats, ALU operations, and pipeline behaviour. Based on this foundation, the processor architecture was designed stage by stage. A complete Datapath was mapped, covering instruction fetch, decoding, operand read, ALU execution, memory interface, and final write-back. Each stage was separated using explicit pipeline latches (IF/ID, ID/EX, EX/MEM, MEM/WB), allowing smooth propagation of control and data signals across cycles.

The ALU was implemented as a combinational module integrated directly within the EX stage, supporting operations such as addition, subtraction, multiplication, logical AND/OR, and SLT. The processor also incorporated immediate arithmetic, load/store addressing, and conditional branching. The control logic followed a simplified, opcode-driven decoding method consistent with the instruction set supported by the design.

Implementation was followed by extensive simulation and verification. A structured set of testbenches was developed to validate every instruction type individually—arithmetic (ADD, SUB), multiplication (MUL), logical operations (AND, OR), comparison (SLT), branching (BEQZ), and memory operations (LW, SW). Each testbench included carefully spaced NOPs to account for pipeline hazards and ensured that data

moved correctly through all five pipeline stages. Debug signals were also incorporated to visualize operand values entering the ALU, which significantly aided waveform analysis and debugging.

Overall, the project delivered a complete, modular, and functioning pipelined processor. It not only demonstrates the correctness of the implemented instruction set but also showcases a deep understanding of digital system design, pipelining principles, control signal generation, hazard behaviour, and practical Verilog implementation. The processor serves as a strong academic framework for learning microarchitecture and can be extended into more advanced CPU designs in future development.

11.2 Final Outcomes

The final design meets the intended objectives of creating a functional, pipelined, instruction-level processor capable of executing a subset of the MIPS instruction set. The processor correctly handles instruction sequencing, data flow, control signalling, and hazard conditions inherent in pipelined execution. Simulation waveforms confirmed correct ALU operation, memory access, register file updates, and successful branching behaviour.

The project achieved the following outcomes:

- A fully operational **5-stage pipelined CPU** implemented in Verilog
- A working **20×32 register file** with correct read-write behaviour
- Accurate execution of arithmetic, logical, comparison, memory, and branch instructions
- Clean **debug signals** for operand tracing at the EX stage
- Verified functionality through **multiple structured testbenches**
- A modular and extendable architecture suitable for further enhancement

These results demonstrate a robust and reliable pipelined processor capable of serving as a strong foundation for more advanced CPU features

11.3 Future Scopes

There are several ways this processor can be expanded and improved in future work. First, support for additional instructions—such as MULT/DIV variants, shift operations, or advanced branching mechanisms—can be added to extend the ISA. Hazard detection and forwarding units can be implemented to eliminate the manual insertion of NOPs, thereby improving the performance and reducing pipeline stalls.

Structural enhancements such as a dedicated instruction memory and data memory, improved control logic, and separation of combinational/sequential submodules can make the design more scalable. Implementing features like pipelined multiplier units, cache memory, reordered execution, or even superscalar extensions can transform this design into a more modern CPU pipeline. With further optimization, the processor can also be synthesized onto FPGA hardware for real-time demonstration.

REFERENCES

1. David A. Patterson and John L. Hennessy, *Computer Organisation and Design: The Hardware/Software Interface,* Morgan Kaufmann Publishers.
2. NPTEL Online Certification Course (IIT), *Computer Organisation and Architecture* ~ Video Lectures
3. NPTEL Online Certification Course (IIT), *Hardware Modelling Through Verilog HDL* ~ Video Lectures and Notes.
4. Xilinx Vivado Design Suite Documentation, Xilinx Inc.
5. OpenAI, *ChatGPT* ~ Technical Assistance and Concept Clarification.
6. Google DeepMind, *Gemini AI* ~ Conceptual support and explanation.
7. Anthropic, Claude AI ~ Code Assistance and Verilog Implementation and Support.