

VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
INTERNATIONAL UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING



DIGITAL SYSTEM DESIGN
Final Project

DESIGN AND SIMULATION OF RISC-V PROCESOR USING VERILOG

Submitted by
Bùi Gia Bảo – ITITIU22019

Date Submitted: 12/01/2026
Course Instructor: M.Eng. Vo Minh Thanh

1. Introduction

1.1 Background of RISC Processors and Pipelining

Modern computer systems require processors that can execute a large number of instructions efficiently while maintaining low hardware complexity and power consumption. To meet these requirements, processor architecture has evolved from complex instruction sets toward more streamlined designs known as **Reduced Instruction Set Computer (RISC)** architectures. RISC processors are based on the principle that a smaller and simpler set of instructions can be executed more efficiently by hardware, resulting in improved performance and easier implementation.

Key characteristics of RISC architectures include a load-store design, a large register file, fixed-length instructions, and simple addressing modes. These features allow instructions to be decoded and executed quickly and consistently, making RISC processors especially suitable for pipelined execution. The **MIPS architecture** is one of the most well-known RISC architectures and is widely used as a teaching and research model due to its clean instruction format and well-structured datapath.

One of the most effective techniques for improving processor performance in RISC systems is **instruction pipelining**. Pipelining divides instruction execution into a sequence of stages, where each stage performs a specific portion of the instruction. Instead of waiting for one instruction to complete before starting the next, multiple instructions are processed concurrently, each at a different pipeline stage. As a result, pipelining significantly increases instruction throughput without requiring additional execution units or higher clock frequencies.

The classic MIPS processor uses a **five-stage pipeline**, consisting of:

1. **Instruction Fetch (IF)** – fetching the instruction from instruction memory,
2. **Instruction Decode (ID)** – decoding the instruction and reading registers,
3. **Execute (EX)** – performing arithmetic or logical operations and address calculations,
4. **Memory Access (MEM)** – accessing data memory for load and store instructions,
5. **Write Back (WB)** – writing results back to the register file.

This five-stage organization represents a balanced trade-off between performance and hardware complexity and serves as a foundation for understanding more advanced processor designs.

1.2 Motivation for Using a 5-Stage MIPS Pipeline

The motivation for implementing a 5-stage pipelined MIPS processor lies in both its **educational value** and **practical relevance**. The MIPS pipeline provides a clear and structured datapath that allows designers to study the interaction between pipeline stages, control logic, and memory systems in a systematic manner. Because each pipeline stage has a well-defined function, the effects of pipelining on performance and correctness can be analyzed in detail.

However, pipelining also introduces several challenges, commonly referred to as **pipeline hazards**. These hazards occur when the normal flow of instructions through the pipeline is disrupted. The three main types of hazards are data hazards, control hazards, and structural hazards. In a MIPS processor, structural hazards are typically avoided through careful hardware design, but data and control hazards must be explicitly handled.

Data hazards arise when an instruction depends on the result of a previous instruction that has not yet completed its execution. Control hazards occur due to branch and jump instructions, which affect the program counter before the correct next instruction is known. Without proper handling, these hazards can lead to incorrect program execution.

By implementing a 5-stage MIPS pipeline, this project provides an opportunity to explore real-world solutions to these problems, such as **pipeline stalling**, **data forwarding**, and **pipeline flushing**. Forwarding techniques reduce performance loss by routing data directly between pipeline stages, while hazard detection units ensure correctness by inserting stalls only when necessary. These mechanisms reflect techniques used in real commercial processors.

Furthermore, implementing the processor on an FPGA allows for hands-on verification and observation of pipeline behavior, bridging the gap between theoretical processor design and actual hardware implementation.

1.3 Objectives of the Project

The main objective of this project is to design, implement, and verify a **five-stage pipelined MIPS processor** using the Verilog hardware description language and an FPGA-based platform.

The specific objectives of the project are detailed as follows:

- **Design and implement a complete 5-stage pipelined MIPS processor**, including the Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back stages, with appropriate pipeline registers to maintain correct data and control flow between stages.
- **Develop and integrate hazard handling mechanisms**, including a hazard detection unit to manage load-use data hazards through pipeline stalling and a forwarding unit to resolve data dependencies by forwarding results from later pipeline stages to earlier stages.
- **Implement control hazard handling**, including branch and jump instructions, through pipeline flushing and program counter control logic to ensure correct instruction sequencing.
- **Verify processor functionality on an FPGA**, using physical switches to control clock, reset, and display selection, and HEX displays and LEDs to observe instruction execution, computation results, stall conditions, and forwarding behavior in real time.

Through achieving these objectives, the project aims to provide a comprehensive understanding of pipelined processor design, hazard management, and hardware verification. The implementation demonstrates how fundamental computer architecture concepts are translated into a working hardware system using modern digital design techniques.

2. Overall System Architecture

2.1 Top-Level Block Diagram Description

The overall system architecture is based on a **five-stage pipelined MIPS processor** integrated with an FPGA interface for control and visualization. The system is organized hierarchically, with a top-level module responsible for FPGA input/output management and a processor core module implementing the pipeline datapath and control logic.

At the highest level, the architecture consists of the following components:

- FPGA input interface (switches for clock, reset, and display selection)
- Top-level control and display logic
- Five-stage pipelined MIPS processor core
- Output visualization through LEDs and seven-segment displays

This modular organization improves design clarity, simplifies debugging, and allows each functional block to be verified independently.

2.2 Main Module Description

2.2.1 project Module (Top Module / FPGA Interface)

The project module serves as the **top-level hardware interface** between the FPGA board and the MIPS processor core. It instantiates the `mips_pipeline` module and connects internal processor signals to physical FPGA peripherals for monitoring and debugging.

Key functions of the project module include:

- Supplying clock and reset signals to the processor
- Selecting which internal processor data is displayed
- Driving LED indicators to show pipeline status
- Converting 32-bit data into hexadecimal form for seven-segment displays

A display multiplexer controlled by a switch allows the user to observe either the current instruction or the write-back result.

FPGA Switch and Output Mapping

FPGA Signal	Function
SW[17]	Manual clock input
SW[16]	Global reset
SW[0]	Display select
LEDG[7]	Stall indicator
LEDG[3:0]	Forwarding status
LEDR[17:0]	Program counter
HEX0-HEX7	Instruction / result display

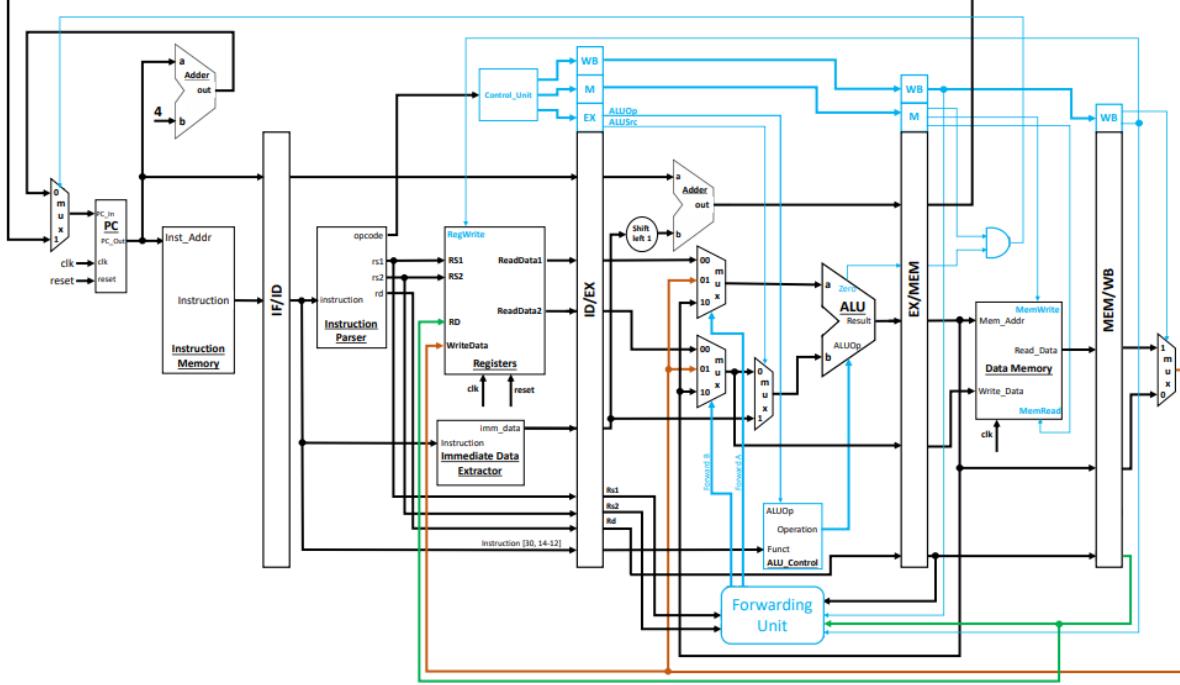
2.2.2 mips_pipeline Module (Core CPU)

The mips_pipeline module implements the complete five-stage MIPS processor. It integrates the datapath, control logic, pipeline registers, hazard detection unit, and forwarding unit.

The processor supports arithmetic, memory, branch, and jump instructions. Each instruction progresses through the pipeline stages concurrently with other instructions, improving overall throughput.

Pipeline registers store both data and control signals to maintain correct execution ordering.

Complete 5-Stage MIPS Pipeline Datapath with Hazard Detection and Forwarding



2.3 Clock and Reset Strategy

A synchronous clocking approach is used, where all sequential elements operate on the rising edge of a single clock signal. For debugging purposes, the clock is manually controlled through an FPGA switch, allowing single-cycle execution observation.

The reset signal initializes all pipeline registers, control signals, and memories, ensuring the processor starts from a known state.

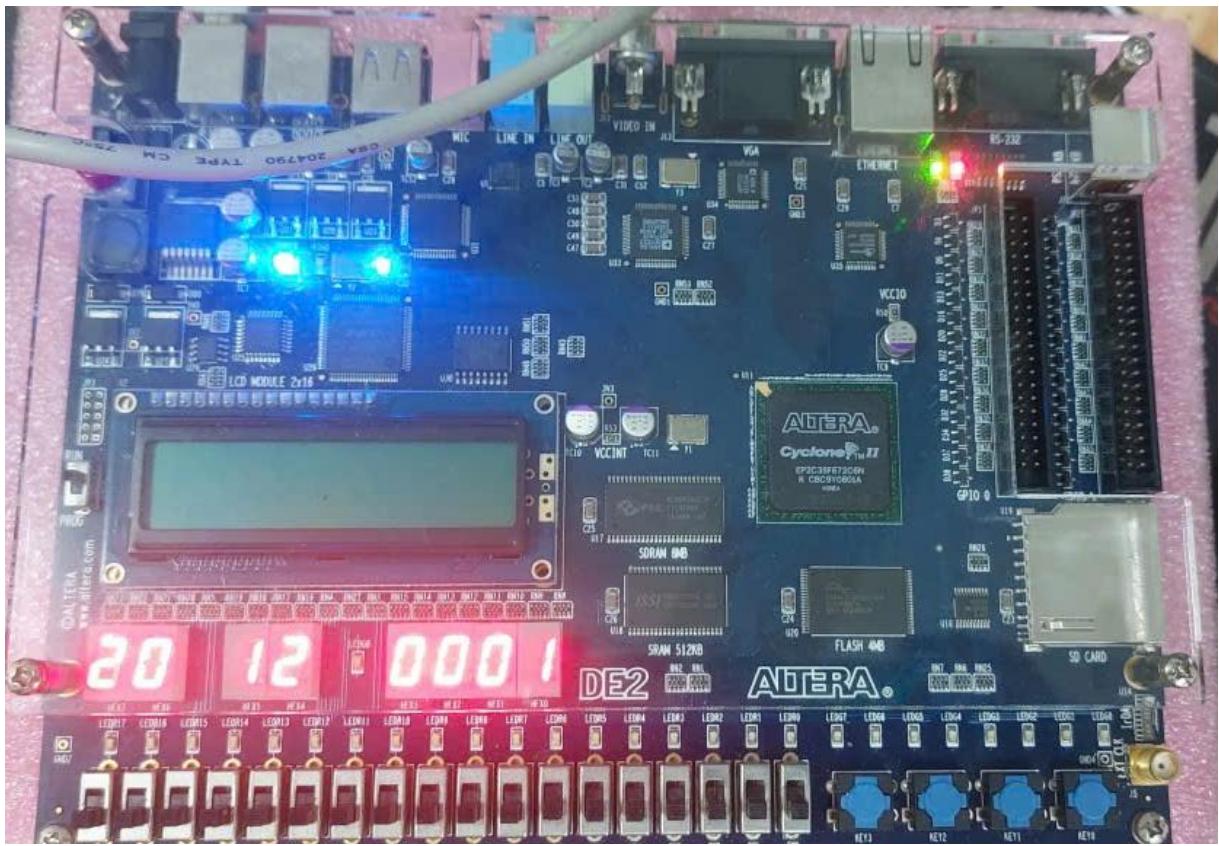
Component	Reset Action
Program Counter	Set to 0
Pipeline Registers	Cleared
Instruction Memory	Initialized
Data Memory	Initialized
Control Signals	Set to default

2.4 FPGA Input and Output Usage

FPGA inputs and outputs are used to provide visibility into the internal operation of the pipeline. This design choice enables real-time verification of pipeline behavior such as stalling, forwarding, and instruction flow.

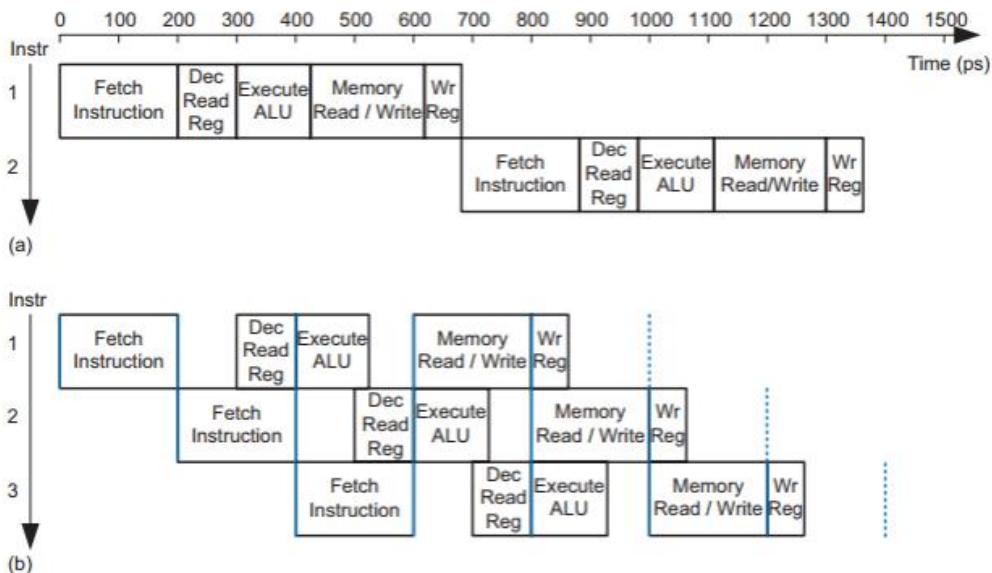
Switches control execution and display modes, LEDs indicate hazard-related events, and HEX displays show processor data in hexadecimal format.

FPGA Board I/O



3. Five-Stage Pipeline Overview

The processor is implemented using a **five-stage pipelined MIPS architecture**, where each instruction is divided into distinct stages that operate concurrently on different instructions. This approach increases instruction throughput while maintaining a simple and well-structured datapath. Pipeline registers are inserted between stages to store both data and control signals, ensuring correct execution across clock cycles.

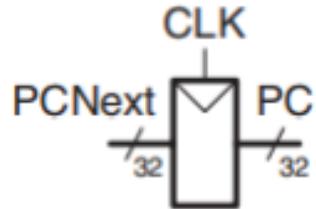


3.1 Instruction Fetch (IF)

The Instruction Fetch stage is responsible for retrieving the next instruction from instruction memory and computing the address of the subsequent instruction.

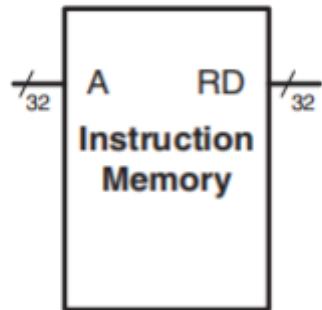
Program Counter (Program_Counter)

The Program Counter (PC) holds the address of the current instruction. It is updated on every rising clock edge unless a stall condition is detected. When a reset is asserted, the PC is initialized to zero, ensuring execution begins from the start of the program.



Instruction Memory (Instruction_Memory)

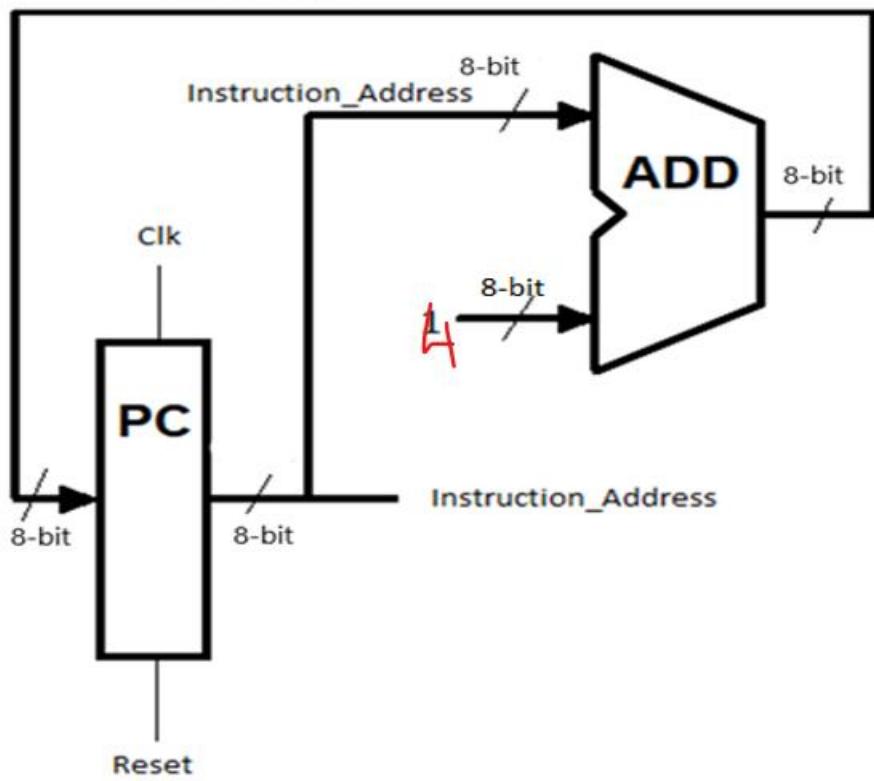
Instruction memory stores the program instructions in word-addressed format. The PC value, after removing the two least significant bits, is used as the memory index. Instructions are fetched combinationallly, allowing the pipeline to receive a new instruction every cycle under normal operation.



PC + 4 Calculation

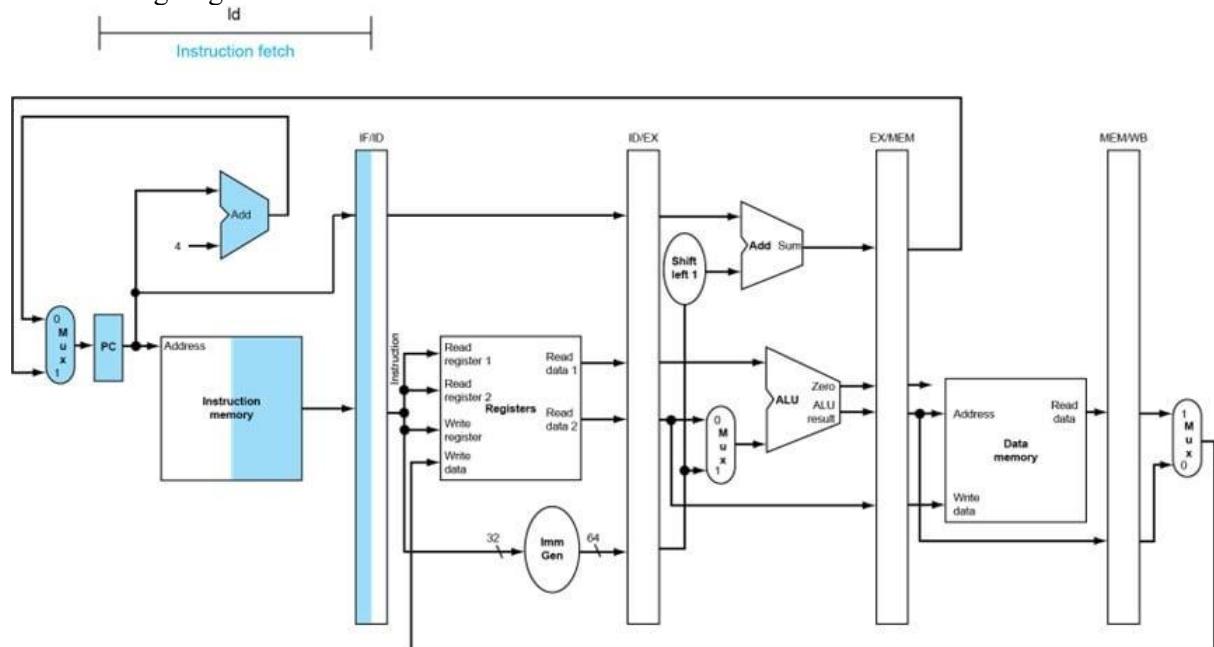
To support sequential execution, the next instruction address is computed by adding 4 to the current PC value. This value represents the address of the next instruction and is passed to the IF/ID pipeline register for use in branch and jump calculations.

W_PC_add_1



IF/ID Pipeline Register

The IF/ID pipeline register stores the fetched instruction and the PC+4 value. This register ensures that instruction decoding occurs with the correct instruction and corresponding program counter value in the following stage.



Stall and Flush Behavior

When a data hazard is detected, the IF stage is stalled by preventing the PC from updating and freezing

the IF/ID pipeline register. In the event of a taken branch or jump instruction, a flush mechanism clears the IF/ID register to discard incorrectly fetched instructions.

3.2 Instruction Decode (ID)

The Instruction Decode stage interprets the fetched instruction, reads operands from the register file, and generates control signals for subsequent stages.

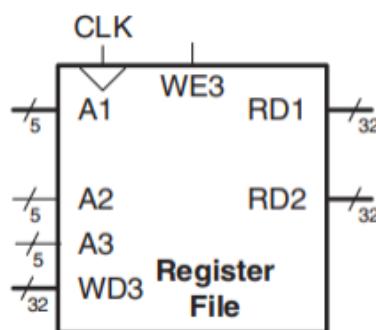
Control Unit (control)

The control unit decodes the opcode field of the instruction and generates control signals such as RegDst, ALUSrc, MemRead, MemWrite, MemtoReg, RegWrite, Branch, and Jump. These signals determine how the instruction is executed in later stages.



Register File (Register_File)

The register file contains 32 general-purpose registers. Two registers are read simultaneously based on the source register fields of the instruction, while a write operation occurs in the Write Back stage. Register zero is hardwired to zero to comply with MIPS conventions.



Immediate Generation (Sign Extension)

For immediate-type instructions, the 16-bit immediate field is sign-extended to 32 bits. This extended value is used as an ALU operand or branch offset in later stages.

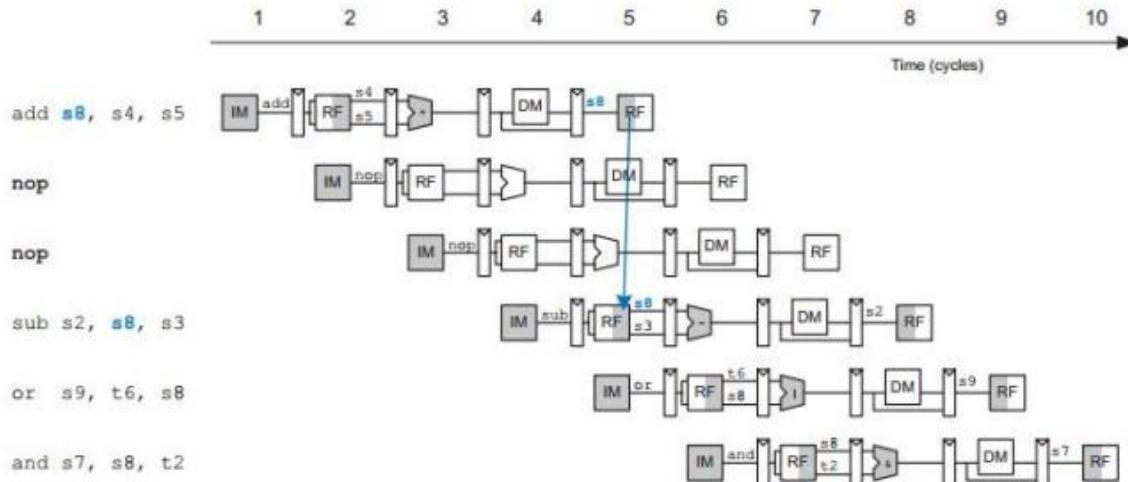


Jump Address Calculation

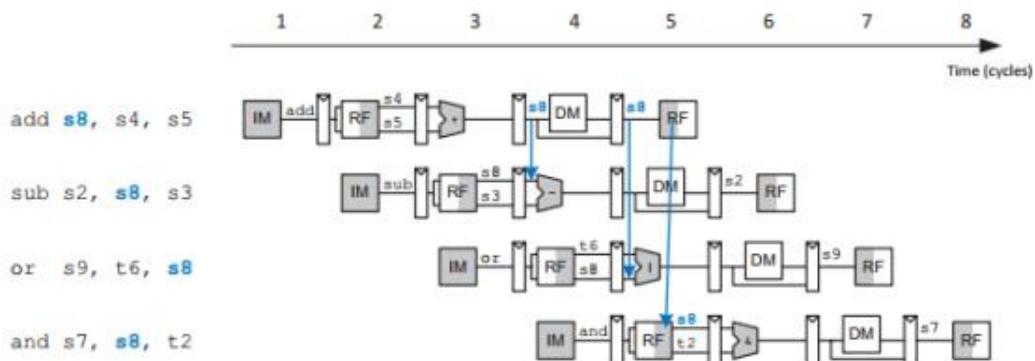
For jump instructions, the jump target address is formed by combining the upper bits of the PC+4 value with the instruction's target field, followed by a left shift of two bits.

Hazard Detection Interaction

The Hazard Detection Unit monitors instructions in the ID and EX stages to identify load-use data hazards. When such a hazard is detected, a stall signal is asserted, preventing incorrect data usage by subsequent instructions.



Solving data hazard with NOP instruction



Solving data hazard with forwarding

ID/EX Pipeline Register

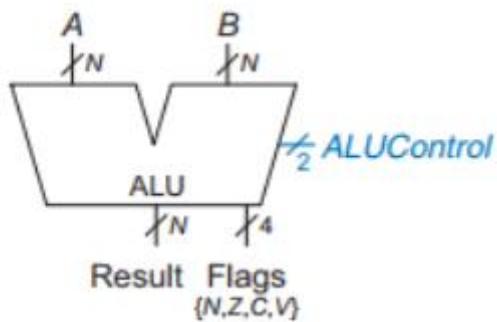
The ID/EX pipeline register stores decoded register values, immediate data, destination register fields, and control signals, passing them safely to the Execute stage.

3.3 Execute (EX)

The Execute stage performs arithmetic and logical operations, evaluates branch conditions, and resolves data hazards using forwarding.

ALU Operations (alu)

The Arithmetic Logic Unit (ALU) performs operations such as addition, subtraction, logical AND, and logical OR. The specific operation is selected based on the ALU control signal derived from the instruction type.



ALU Control via ALUop

The ALUop control signal determines the operation executed by the ALU. For example, addition is used for arithmetic operations and address calculations, while subtraction is used for branch comparisons.

Operand Selection (Forwarded or Register)

Operands supplied to the ALU may come from the ID/EX register or be forwarded from later pipeline stages. Multiplexers controlled by forwarding signals select the correct operand source, ensuring data hazards are resolved without unnecessary stalls.

Branch Target Computation

The branch target address is calculated by adding the sign-extended immediate value (shifted left by two bits) to the PC+4 value from the ID/EX register.

Forwarding Unit Operation

The Forwarding Unit detects data dependencies between instructions in the EX, MEM, and WB stages. It generates control signals that forward the most recent data directly to the ALU inputs, minimizing pipeline stalls.

EX/MEM Pipeline Register

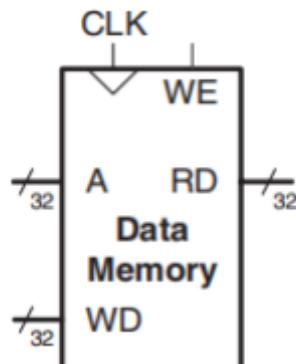
The EX/MEM pipeline register stores the ALU result, branch target address, zero flag, destination register number, and relevant control signals for use in the Memory Access stage.

3.4 Memory Access (MEM)

The Memory Access stage handles data memory operations and determines the next program counter value.

Data Memory (Data_Memory)

Data memory is accessed during load and store instructions. Load instructions read data from memory, while store instructions write data to memory using values forwarded or stored from previous stages.



Load and Store Handling

For load instructions, the data read from memory is forwarded to the Write Back stage. For store instructions, the data to be written is selected using forwarding logic to ensure correctness.

Branch Decision Logic

The branch decision is finalized in this stage using the branch control signal and the zero flag generated in the Execute stage. If a branch is taken, the PC is updated with the branch target address.

PC Update Logic

The next PC value is selected based on jump, branch, or sequential execution. This logic ensures correct instruction flow and triggers pipeline flushing when necessary.

3.5 Write Back (WB)

The Write Back stage completes instruction execution by updating the register file with the final result.

Write-Back Data Selection

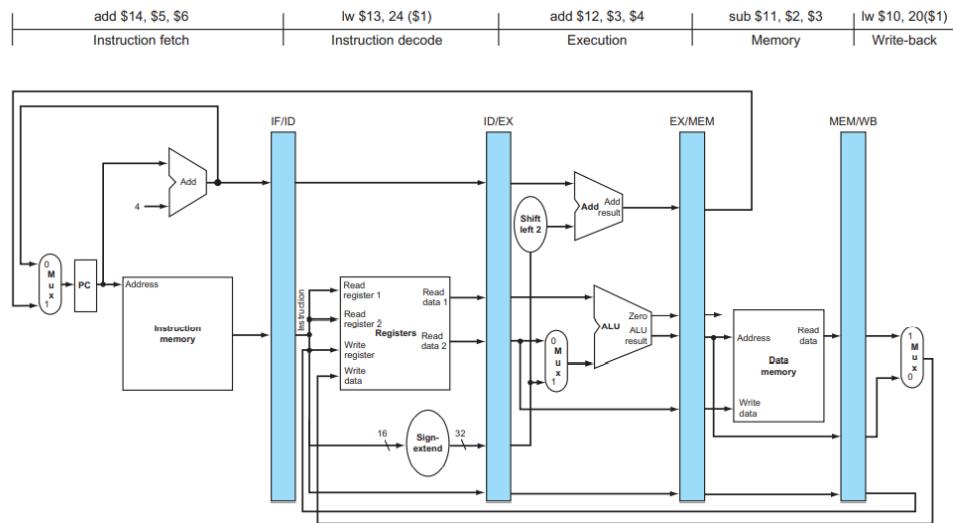
A multiplexer selects between the ALU result and memory data based on the MemtoReg control signal. The selected value is written back to the register file.

Register Write Enable Control

The RegWrite control signal determines whether a register write operation occurs. Only instructions that produce a result enable this signal.

MEM/WB Pipeline Register

The MEM/WB pipeline register stores the final data and control signals required for register write-back, ensuring correct synchronization with the clock.



4. Pipeline Registers Design

Pipeline registers are a fundamental component of the pipelined processor architecture. They are used to separate pipeline stages and to store both **data values** and **control signals** as instructions progress through the pipeline. In this design, four pipeline registers are implemented: **IF/ID**, **ID/EX**, **EX/MEM**, and **MEM/WB**. Each register operates synchronously with the system clock and ensures correct data propagation between stages.

4.1 IF/ID Pipeline Register

The IF/ID pipeline register stores the instruction fetched from instruction memory and the

corresponding PC+4 value. These values are required in the Instruction Decode stage for instruction decoding, register access, and jump address calculation.

The IF/ID register is updated on every rising clock edge when no stall condition is present. When a reset or flush signal is asserted, the register contents are cleared to prevent incorrect instruction execution. This flushing mechanism is essential for handling control hazards caused by branch and jump instructions.

Key characteristics of the IF/ID register include:

- Storage of the fetched instruction
- Storage of the PC+4 value
- Stall control to freeze contents during data hazards
- Flush control to discard incorrectly fetched instructions

4.2 ID/EX Pipeline Register

The ID/EX pipeline register transfers decoded instruction information from the Instruction Decode stage to the Execute stage. This register holds both **operand data** and **control signals**, allowing the Execute stage to operate independently of earlier stages.

Stored information includes:

- Register operands (RD1 and RD2)
- Sign-extended immediate value
- Source and destination register numbers
- Control signals such as RegDst, ALUSrc, MemRead, MemWrite, MemtoReg, RegWrite, Branch, and ALUop

When a load-use hazard is detected, the ID/EX register is cleared, effectively inserting a **NOP (bubble)** into the pipeline. This prevents incorrect data usage while allowing earlier instructions to continue execution.

ID/EX Pipeline Register Contents

Signal Category	Signals Stored
Data	RD1, RD2, Immediate, PC+4
Register IDs	rs, rt, rd
Control	RegDst, ALUSrc, MemRead, MemWrite, MemtoReg, RegWrite, Branch, ALUop

4.3 EX/MEM Pipeline Register

The EX/MEM pipeline register stores the results produced by the Execute stage and passes them to the Memory Access stage. These include both computation results and branch-related information.

Stored values include:

- ALU result
- Forwarded store data
- Branch target address
- Zero flag for branch evaluation
- Destination register number
- Memory and write-back control signals

This register plays a crucial role in enabling branch decision logic and memory access while maintaining correct timing between stages.

4.4 MEM/WB Pipeline Register

The MEM/WB pipeline register stores data required for the Write Back stage. It ensures that the correct value is written back to the register file after memory access or ALU computation.

Stored values include:

- Data read from memory
- ALU computation result

- Destination register number
- Write-back control signals

The separation provided by this register allows memory operations to complete before register updates occur, preserving correct instruction semantics.

Signal	Description
MEMWB_Mem	Data read from memory
MEMWB_ALU	ALU result
MEMWB_rd	Destination register
MEMWB_MemtoReg	Write-back data select
MEMWB_RegWrite	Register write enable

MEM/WB Pipeline Register Contents

4.5 Pipeline Register Control Behavior

All pipeline registers operate on the rising edge of the clock and are affected by reset, stall, and flush signals. Reset initializes all registers to zero, ensuring a known starting state. Stall signals freeze selected pipeline registers, while flush signals clear instructions that should not be executed.

This coordinated control ensures both **correctness** and **efficient pipeline operation** in the presence of hazards.

5. Control Unit Design

The control unit is responsible for interpreting the instruction opcode and generating the appropriate control signals that govern datapath behavior across all pipeline stages. In this design, control signal generation is centralized in the Instruction Decode stage and then propagated through the pipeline using pipeline registers.

5.1 Role of the Control Unit

The control unit decodes the 6-bit opcode field of the instruction and determines the instruction type, such as R-type, immediate-type, memory access, branch, or jump. Based on this decoding, it produces control signals that specify:

- Register destination selection
- ALU operand source
- ALU operation type
- Memory read and write operations
- Write-back behavior
- Branch and jump control

These signals ensure that each instruction is executed correctly as it moves through the pipeline.

5.2 Instruction Decoding and Supported Instructions

The processor supports a subset of the MIPS instruction set, including R-type arithmetic instructions, immediate arithmetic, load and store operations, branch instructions, and jump instructions. Each instruction type activates a unique combination of control signals.

Supported instruction categories include:

- R-type instructions (add, sub, and, or)
- addi
- lw and sw
- beq
- j

The control logic is implemented using combinational logic that maps each opcode to a predefined control signal pattern.

Instruction	RegDst	ALUSrc	MemRead	MemWrite	MemtoReg	RegWrite	Branch	Jump	ALUop
R-type	1	0	0	0	0	1	0	0	ADD
addi	0	1	0	0	0	1	0	0	ADD
lw	0	1	1	0	1	1	0	0	ADD
sw	X	1	0	1	X	0	0	0	ADD
beq	X	0	0	0	X	0	1	0	SUB
j	X	X	0	0	X	0	0	1	X

Control Signal Settings for Supported Instructions

5.3 Control Signal Propagation through the Pipeline

Once generated, control signals are stored in the ID/EX pipeline register and carried forward to later stages. This ensures that each pipeline stage receives the correct control information corresponding to the instruction currently being executed.

Control signals are grouped by function:

- Execute-stage control (RegDst, ALUSrc, ALUop)
- Memory-stage control (MemRead, MemWrite, Branch)
- Write-back control (MemtoReg, RegWrite)

This grouping simplifies pipeline register design and improves clarity.

5.4 ALU Control Strategy

The ALU control logic determines which arithmetic or logical operation the ALU performs. In this design, the ALU operation is directly derived from the ALUop signal produced by the control unit. Addition is used for arithmetic operations, load/store address calculation, and PC updates, while subtraction is used for branch comparison. Logical operations are supported for R-type instructions. This simplified ALU control approach reduces hardware complexity while maintaining correct functionality for the supported instruction set.

5.5 Summary of Control Unit Operation

The control unit ensures correct instruction execution by coordinating datapath operations across all pipeline stages. By generating control signals early in the pipeline and propagating them forward, the processor achieves both high performance and correct operation in a pipelined environment.

6. Hazard Handling Mechanisms

In a pipelined processor, hazards occur when the normal flow of instructions through the pipeline could lead to incorrect execution. To ensure correctness while maintaining high performance, this design incorporates dedicated hardware mechanisms to handle **data hazards** and **control hazards**. These mechanisms include a **Hazard Detection Unit**, a **Forwarding Unit**, and pipeline **stalling** and **flushing** logic.

6.1 Data Hazards

Data hazards arise when an instruction depends on the result of a previous instruction that has not yet completed its execution. In a five-stage pipeline, this situation commonly occurs because results are written back to the register file only in the Write Back stage, while subsequent instructions may require the data earlier.

The most critical data hazard addressed in this design is the **load-use hazard**, which occurs when an instruction immediately following a load instruction attempts to use the loaded value. Since the data from memory is not available until the MEM stage, forwarding alone is insufficient, and the pipeline must be stalled.

To reduce performance degradation, this processor uses **data forwarding** whenever possible and resorts to stalling only when forwarding cannot resolve the dependency.

6.2 Hazard Detection Unit (Stalling Mechanism)

The Hazard Detection Unit monitors the instructions in the ID and EX stages to detect load-use

hazards. Specifically, it checks whether the instruction in the EX stage is performing a memory read and whether its destination register matches either source register of the instruction in the ID stage.

When such a hazard is detected:

- The Program Counter is prevented from updating.
- The IF/ID pipeline register is frozen.
- The ID/EX pipeline register is cleared, inserting a NOP into the pipeline.

This ensures that the dependent instruction is delayed until the required data becomes available.

6.3 Data Forwarding Technique

Data forwarding is used to resolve most data hazards without stalling the pipeline. Instead of waiting for data to be written back to the register file, results are forwarded directly from later pipeline stages to the ALU inputs in the Execute stage.

This design supports forwarding from:

- The EX/MEM stage (most recent result)
- The MEM/WB stage (write-back result)

Multiplexers controlled by forwarding signals select the appropriate operand source.

6.4 Forwarding Unit Operation

The Forwarding Unit compares the destination registers of instructions in the EX/MEM and MEM/WB stages with the source registers of the instruction in the ID/EX stage. Based on these comparisons, it generates two control signals, ForwardA and ForwardB, which determine the source of the ALU operands.

Priority is given to data from the EX/MEM stage, as it contains the most recent result. If no forwarding is required, operands are taken directly from the ID/EX register.

Forward Signal	Selected Source
00	ID/EX register
10	EX/MEM ALU result
01	MEM/WB write-back data

Forwarding Control Signal Encoding

6.5 Control Hazards

Control hazards occur due to branch and jump instructions, which affect the program counter before the correct next instruction is known. In this design, branch decisions are resolved in the MEM stage, while jump decisions are identified in the ID stage.

When a branch is taken or a jump instruction is executed:

- Incorrectly fetched instructions are flushed from the pipeline.
- The program counter is updated with the correct target address.

This approach ensures correct instruction sequencing, at the cost of discarding a small number of prefetched instructions.

7. Instruction Memory and Test Program

To verify the correct operation of the pipelined MIPS processor, a dedicated test program is stored in the instruction memory. This program is designed to exercise **all major features of the processor**, including arithmetic operations, memory access, data hazards, control hazards, forwarding, stalling, branching, and jumping. By executing this program on the FPGA, the internal behavior of the pipeline can be observed and validated.

7.1 Instruction Memory Organization

Instruction memory is implemented as a **word-addressed memory** that stores 32-bit MIPS instructions. The Program Counter (PC) provides the instruction address, with the two least significant bits removed to ensure word alignment. Instructions are fetched combinationallly, allowing one instruction to be read per clock cycle.

The instruction memory is initialized during reset, ensuring a known and repeatable execution

sequence every time the processor is restarted. This approach simplifies debugging and verification.

7.2 Structure of the Test Program

The test program is carefully structured to validate different processor functionalities in a progressive manner. Instructions are grouped logically to test specific pipeline behaviors.

The program includes the following instruction categories:

- Immediate arithmetic instructions (addi)
- Register-to-register arithmetic instructions (R-type)
- Memory access instructions (lw, sw)
- Branch instructions (beq)
- Jump instructions (j)
- No-operation (NOP) instructions for pipeline stabilization

7.3 Arithmetic Instruction Testing

The initial portion of the program uses immediate and R-type arithmetic instructions to verify:

- Register file read and write operations
- ALU functionality
- Write-back correctness

These instructions confirm that arithmetic results propagate correctly through all pipeline stages and are written back to the appropriate registers.

Arithmetic Instruction Test Sequence

Instruction Type	Purpose
addi	Initialize registers
add	Test ALU addition
sub	Test ALU subtraction
and	Test logical AND
or	Test logical OR

7.4 Memory Access and Load-Use Hazard Testing

The next portion of the test program focuses on memory access instructions. A store instruction writes data to memory, followed immediately by a load instruction that retrieves the stored value. This sequence intentionally introduces a **load-use data hazard**, where the loaded value is required by the next instruction.

This scenario verifies:

- Correct operation of the Data Memory module
- Hazard Detection Unit functionality
- Proper insertion of pipeline stalls
- Correct resumption of execution after the stall

The presence of a stall can be observed using the FPGA LED indicator dedicated to stall detection.

7.5 Forwarding Verification

Following the load-use hazard test, arithmetic instructions that depend on recently computed results are executed. These instructions are designed such that forwarding can resolve the data dependency without stalling the pipeline.

This confirms:

- Correct operation of the Forwarding Unit
- Proper selection of forwarded operands
- Priority handling between EX/MEM and MEM/WB forwarding paths

Forwarding activity can be observed using FPGA LEDs that display the ForwardA and ForwardB control signals.

Forwarding Scenarios Tested

Dependency Type	Resolution Method

EX → EX	Forwarding from EX/MEM
MEM → EX	Forwarding from MEM/WB
Load-use	Stall + forwarding

7.6 Branch Instruction Testing

Branch instructions are included to test control hazard handling. A conditional branch compares two registers and, if the condition is satisfied, redirects program execution to a new address.

This portion of the program verifies:

- Correct branch condition evaluation
- Branch target address calculation
- Pipeline flushing of incorrectly fetched instructions
- Proper PC update after branch resolution

7.7 Jump Instruction Testing

The test program also includes an unconditional jump instruction to verify jump address calculation and immediate redirection of program flow. Instructions following the jump are intentionally placed to confirm that they are correctly skipped.

This confirms:

- Jump control signal generation
- Jump address computation
- Correct pipeline flush behavior

7.8 Summary of Test Program Coverage

The instruction memory test program provides comprehensive coverage of the processor's functionality. By executing this program, all major pipeline features—including hazard detection, forwarding, branching, and jumping—are validated both functionally and visually using FPGA outputs.

8. FPGA Integration and Output Display

To facilitate real-time verification and debugging of the pipelined MIPS processor, the design is integrated with FPGA input and output peripherals. Switches, LEDs, and seven-segment displays are used to control processor operation and to visualize internal pipeline behavior. This hardware-based observation complements simulation results and provides strong evidence of correct processor functionality.

8.1 FPGA-Based Execution Control

The FPGA switches are used to manually control the execution of the processor. This approach allows the user to advance the pipeline **one clock cycle at a time**, making it easier to observe instruction flow, pipeline stalls, forwarding behavior, and control hazard resolution.

The manual clock control is especially useful for educational and debugging purposes, as it allows precise inspection of pipeline state changes at each cycle.

8.2 Switch (SW) Utilization

The FPGA switches provide control signals to the processor and the display logic.

- **SW[17]** is used as the clock input, allowing single-step execution.
- **SW[16]** serves as the global reset signal, initializing the processor state.
- **SW[0]** selects the data displayed on the seven-segment displays.

This configuration provides flexible control over both execution and output visualization.

FPGA Switch Functionality

Switch	Function
SW[17]	Manual clock
SW[16]	Processor reset

SW[0]	Display select (instruction/result)
-------	-------------------------------------

8.3 LED Indicators for Pipeline Status

LED indicators are used to display key pipeline control signals, allowing immediate identification of hazard-related events during execution.

- **LEDG[7]** indicates when a pipeline stall occurs due to a load-use data hazard.
- **LEDG[1:0]** display the ForwardA control signal.
- **LEDG[3:2]** display the ForwardB control signal.

These indicators make it possible to visually confirm that the Hazard Detection Unit and Forwarding Unit are operating as expected.

8.4 Program Counter Monitoring Using LEDs

The lower bits of the Program Counter (PC) are mapped to the red LEDs on the FPGA. This allows the user to observe instruction sequencing and control flow changes caused by branches and jumps. Monitoring the PC value in real time helps verify:

- Sequential instruction execution
- Branch target redirection
- Jump instruction behavior

8.5 Seven-Segment Display (HEX) Output

Eight seven-segment displays are used to show a 32-bit value in hexadecimal format. A display multiplexer selects between two internal processor signals:

- The currently fetched instruction
- The value written back to the register file

Each HEX display represents one hexadecimal digit, allowing complete visualization of 32-bit values. This output mechanism provides clear insight into instruction encoding and execution results.

8.6 Display Selection Logic

A combinational selection mechanism controlled by **SW[0]** determines which internal value is shown on the HEX displays. This allows the user to dynamically switch between observing instruction words and execution results without interrupting processor operation.

This feature enhances debugging capability and demonstrates the flexibility of the FPGA-based interface.

9. Simulation and Verification

To ensure the correctness and reliability of the proposed pipelined MIPS processor, the design is verified through a combination of **functional simulation** and **FPGA-based testing**. This two-level verification approach allows both theoretical validation of the design logic and practical observation of real hardware behavior.

9.1 Verification Methodology

The verification process follows a structured methodology:

1. Functional correctness is first validated through simulation by observing signal behavior across pipeline stages.
2. Hardware verification is then performed on the FPGA using switches, LEDs, and seven-segment displays to observe real-time execution.

This approach ensures that the processor operates correctly under both controlled simulation conditions and actual hardware constraints.

9.2 Functional Simulation

Functional simulation is used to verify:

- Correct instruction execution
- Proper pipeline stage operation
- Correct propagation of data and control signals
- Hazard detection and resolution behavior

Simulation allows internal signals such as pipeline registers, ALU outputs, forwarding paths, and stall

signals to be monitored over time.

9.3 Verification of Data Hazard Handling

Data hazards are verified by simulating instruction sequences that introduce register dependencies. Load-use hazards are specifically tested to confirm that the Hazard Detection Unit correctly inserts pipeline stalls.

Simulation results show that:

- The stall signal is asserted for exactly one cycle during a load-use hazard.
- The dependent instruction is delayed without corrupting data.
- Execution resumes correctly once the data becomes available.

9.4 Verification of Forwarding Mechanism

Forwarding behavior is verified using instruction sequences that require results from previous instructions without introducing a load-use hazard. Simulation confirms that:

- Forwarding paths from EX/MEM and MEM/WB stages are correctly selected.
- ForwardA and ForwardB signals change as expected.
- No unnecessary stalls are introduced.

9.5 Verification of Control Hazards

Control hazards caused by branch and jump instructions are verified by monitoring program counter updates and pipeline flushing behavior. Simulation results demonstrate that:

- Incorrectly fetched instructions are flushed when a branch is taken.
- The PC is updated to the correct target address.
- Jump instructions immediately redirect execution flow.

9.6 FPGA-Based Verification

After successful simulation, the design is implemented on an FPGA for hardware verification. Manual clock stepping allows precise observation of pipeline behavior at each cycle.

The following behaviors are confirmed on hardware:

- Instruction sequencing using PC LEDs
- Stall detection using LED indicators
- Forwarding activity using forwarding LEDs
- Correct instruction and result display using HEX displays

10. Results and Discussion

This section analyzes the observed results obtained from simulation and FPGA-based testing of the pipelined MIPS processor. The discussion focuses on functional correctness, pipeline behavior, hazard handling effectiveness, and overall system performance.

10.1 Functional Correctness

The execution of the test program confirms that all supported instruction types operate correctly. Arithmetic, memory access, branch, and jump instructions produce the expected results and update the processor state appropriately. Values written back to the register file match expected outcomes, and memory operations correctly store and retrieve data.

The correctness of execution is validated through:

- Simulation waveforms showing accurate data propagation
- FPGA output displays reflecting correct instruction execution
- Consistent program counter progression

These results indicate that the datapath, control logic, and pipeline registers are properly designed and integrated.

10.2 Pipeline Operation and Instruction Throughput

Observations from simulation and FPGA testing demonstrate that multiple instructions are active in different pipeline stages simultaneously, confirming correct pipelined execution. Once the pipeline is

filled, the processor completes approximately one instruction per clock cycle under hazard-free conditions.

Pipeline stalls occur only when required, such as during load-use data hazards. In all other cases, forwarding allows the pipeline to continue execution without interruption, minimizing performance loss.

10.3 Effectiveness of Data Hazard Handling

The combination of forwarding and stalling mechanisms effectively resolves data hazards.

Forwarding successfully eliminates most read-after-write dependencies without stalling the pipeline. Load-use hazards, which cannot be resolved by forwarding alone, are correctly handled by inserting a single-cycle stall.

The stall indicator on the FPGA provides visible confirmation that the Hazard Detection Unit is activated only when necessary. This demonstrates an efficient balance between correctness and performance.

Hazard Type	Resolution Method	Stall Required
EX → EX	Forwarding	No
MEM → EX	Forwarding	No
Load-use	Stall + Forwarding	Yes

Data Hazard Resolution Summary

10.4 Control Hazard Handling

Branch and jump instructions are handled correctly through pipeline flushing and program counter redirection. When a branch is taken or a jump instruction is executed, incorrectly fetched instructions are flushed, preventing erroneous execution.

While this approach introduces a small performance penalty due to flushed instructions, it guarantees correct control flow. The behavior observed in both simulation and FPGA testing matches the expected MIPS pipeline operation.

10.5 FPGA-Based Observation and Debugging

FPGA integration significantly enhances the ability to observe and debug pipeline behavior. Manual clock stepping allows detailed inspection of each pipeline stage, while LED and HEX outputs provide immediate visual feedback.

This hardware-based verification approach confirms that:

- Pipeline stalls occur at the correct time
- Forwarding paths are activated appropriately
- Program flow changes are accurately reflected

10.6 Design Limitations and Trade-Offs

Although the processor functions correctly, several design trade-offs are identified:

- Branch resolution occurs in a later pipeline stage, resulting in control hazard penalties.
- No branch prediction is implemented, limiting performance for branch-heavy programs.
- The supported instruction set is limited to a subset of MIPS instructions.

These trade-offs were made to maintain design simplicity and clarity, which are appropriate for an educational pipeline implementation.

10.7 FPGA CODE

```
module
project(CLOCK_50,SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);
    input CLOCK_50;
    input [17:0] SW;
    output [17:0] LEDR;
    output [7:0] LEDG;
```

```

output [0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7;
reg [31:0] w_hex;
wire [31:0] result_out, instr_out;
assign LEDG[7] = Stall;

mips_pipeline CPU(
    .clk(SW[17]),
    .reset(SW[16]),
    .Stall(Stall),
    .instr_out(instr_out),
    .pc_out_check(LEDR[17:0]),
    .forwardA_out(LEDG[1:0]),
    .forwardB_out(LEDG[3:2]),
    .result_out(result_out)
);

always @(*) begin
    case (SW[0])
        1'b0: w_hex = instr_out;
        1'b1: w_hex = result_out;
        default: w_hex = 32'b0;
    endcase
end

HEX_7SEG_DECODE H0(.BIN(w_hex[3:0]), .SSD(HEX0));
HEX_7SEG_DECODE H1(.BIN(w_hex[7:4]), .SSD(HEX1));
HEX_7SEG_DECODE H2(.BIN(w_hex[11:8]), .SSD(HEX2));
HEX_7SEG_DECODE H3(.BIN(w_hex[15:12]), .SSD(HEX3));
HEX_7SEG_DECODE H4(.BIN(w_hex[19:16]), .SSD(HEX4));
HEX_7SEG_DECODE H5(.BIN(w_hex[23:20]), .SSD(HEX5));
HEX_7SEG_DECODE H6(.BIN(w_hex[27:24]), .SSD(HEX6));
HEX_7SEG_DECODE H7(.BIN(w_hex[31:28]), .SSD(HEX7));

endmodule

module mips_pipeline(
    input clk,
    input reset,
    output Stall,
    output [31:0] instr_out,
    output [17:0] pc_out_check,
    output [1:0] forwardA_out,
    output [1:0] forwardB_out,
    output [31:0] result_out
);
    assign result_out = WB_data;
    assign forwardA_out = ForwardA;
    assign forwardB_out = ForwardB;

    wire Flush;
    assign Flush = Jump || (EXMEM_Branch && EXMEM_Zero);

```

```

assign pc_out_check = PC;
assign instr_out = IFID_instr;
wire [1:0] ForwardA, ForwardB;
wire [31:0] ALU_A, ALU_B_pre;
assign ALU_A =
(ForwardA == 2'b00) ? IDEX_RD1 :
(ForwardA == 2'b10) ? EXMEM_ALU :
(ForwardA == 2'b01) ? WB_data :
IDEX_RD1;

assign ALU_B_pre =
(ForwardB == 2'b00) ? IDEX_RD2 :
(ForwardB == 2'b10) ? EXMEM_ALU :
(ForwardB == 2'b01) ? WB_data :
IDEX_RD2;

wire [31:0] PC;
wire [31:0] PC_plus4, PC_next, PC_write;
wire [31:0] instr;
assign PC_plus4 = PC + 4;
assign PC_write = Stall ? PC : PC_next;

wire [5:0] opcode = IFID_instr[31:26];
wire [4:0] rs = IFID_instr[25:21];
wire [4:0] rt = IFID_instr[20:16];
wire [4:0] rd = IFID_instr[15:11];
wire [15:0] imm = IFID_instr[15:0];

wire RegDst, ALUSrc, MemRead, MemWrite, MemtoReg, RegWrite, Branch;
wire [2:0] ALUop;

wire [31:0] RD1, RD2;
wire [31:0] WB_data;

wire [4:0] EX_WriteReg = IDEX_RegDst ? IDEX_rd : IDEX_rt;
wire [31:0] ALU_B = IDEX_ALUSrc ? IDEX_Imm : ALU_B_pre;
wire [31:0] ALU_out;
wire Zero;

wire [31:0] SignExtImm = {{16{imm[15]}}, imm};
wire Jump;
wire [31:0] JumpAddr;

/* ===== IF ===== */
Program_Counter PCU(
    .clk(clk),
    .reset(reset),
    .PC_in(PC_write),
    .PC_out(PC)
);

/* Instruction Memory */
Instruction_Memory IM(
    .read_address(PC[31:2]),
    .instruction(instr),

```

```

.reset(reset));

/* IF/ID */
reg [31:0] IFID_PC4, IFID_instr;
    always @(posedge clk or posedge reset) begin
if (reset) begin
    IFID_instr <= 32'b0;
    IFID_PC4 <= 32'b0;
end
else if (Flush) begin
    IFID_instr <= 32'b0;
    IFID_PC4 <= 32'b0;
end
else if (!Stall) begin
    IFID_instr <= instr;
    IFID_PC4 <= PC_plus4;
    end
end

/* ===== ID ===== */
control CU(
opcode,
RegDst, Branch, MemRead, MemtoReg,
ALUop, MemWrite, ALUSrc, RegWrite, Jump
);

Register_File RF(
    clk, rs, rt,
    MEMWB_rd, RD1, RD2,
    WB_data, MEMWB_RegWrite
);

assign JumpAddr = { IFID_PC4[31:28], IFID_instr[25:0], 2'b00 };

/* ID/EX */
reg [31:0] IDEX_RD1, IDEX_RD2, IDEX_Imm, IDEX_PC4;
reg [4:0] IDEX_rs, IDEX_rt, IDEX_rd;
reg IDEX_RegDst, IDEX_ALUSrc, IDEX_MemRead,
    IDEX_MemWrite, IDEX_MemtoReg, IDEX_RegWrite, IDEX_Branch;
reg [2:0] IDEX_ALUop;
always @(posedge clk or posedge reset) begin
if (reset) begin
    IDEX_RegDst <= 0;
    IDEX_ALUSrc <= 0;
    IDEX_MemRead <= 0;
    IDEX_MemWrite <= 0;
    IDEX_MemtoReg <= 0;
    IDEX_RegWrite <= 0;
    IDEX_Branch <= 0;
    IDEX_ALUop <= 0;

    IDEX_RD1 <= 0;
    IDEX_RD2 <= 0;
    IDEX_Imm <= 0;
    IDEX_PC4 <= 0;
    IDEX_rs <= 0;

```

```

    IDEX_rt <= 0;
    IDEX_rd <= 0;
end
else if (Stall) begin
    IDEX_RegDst <= 0;
    IDEX_ALUSrc <= 0;
    IDEX_MemRead <= 0;
    IDEX_MemWrite <= 0;
    IDEX_MemtoReg <= 0;
    IDEX_RegWrite <= 0;
    IDEX_Branch <= 0;
    IDEX_ALUop <= 0;
end
else begin
    IDEX_RD1 <= RD1;
    IDEX_RD2 <= RD2;
    IDEX_Imm <= SignExtImm;
    IDEX_PC4 <= IFID_PC4;
    IDEX_rs <= rs;
    IDEX_rt <= rt;
    IDEX_rd <= rd;

    IDEX_RegDst <= RegDst;
    IDEX_ALUSrc <= ALUSrc;
    IDEX_MemRead <= MemRead;
    IDEX_MemWrite <= MemWrite;
    IDEX_MemtoReg <= MemtoReg;
    IDEX_RegWrite <= RegWrite;
    IDEX_Branch <= Branch;
    IDEX_ALUop <= ALUop;
end
end

/* ===== EX ===== */
alu ALU(
INDEX_ALUop,
ALU_A,
ALU_B,
ALU_out,
Zero);

ForwardingUnit FU(
EXMEM_RegWrite,
MEMWB_RegWrite,
EXMEM_rd,
MEMWB_rd,
INDEX_rs,
INDEX_rt,
ForwardA,
ForwardB
);

wire [31:0] BranchTarget = INDEX_PC4 + (INDEX_Imm << 2);

```

```

/* EX/MEM */
wire [31:0] StoreData;

assign StoreData = (ForwardB == 2'b10) ? EXMEM_ALU : (ForwardB == 2'b01) ?
WB_data : IDEX_RD2;

reg [31:0] EXMEM_ALU, EXMEM_RD2, EXMEM_BranchTarget;
reg [4:0] EXMEM_rd;
reg EXMEM_MemRead, EXMEM_MemWrite,
      EXMEM_MemtoReg, EXMEM_RegWrite, EXMEM_Branch;
reg EXMEM_Zero;

always @(posedge clk) begin
    EXMEM_ALU <= ALU_out;
    EXMEM_RD2 <= StoreData;
    EXMEM_rd <= EX_WriteReg;
    EXMEM_Zero <= Zero;
    EXMEM_BranchTarget <= BranchTarget;

    EXMEM_MemRead <= IDEX_MemRead;
    EXMEM_MemWrite <= IDEX_MemWrite;
    EXMEM_MemtoReg <= IDEX_MemtoReg;
    EXMEM_RegWrite <= IDEX_RegWrite;
    EXMEM_Branch <= IDEX_Branch;
end

/* ===== MEM ===== */
wire [31:0] MemData;

Data_Memory DM(
    EXMEM_ALU[31:2],
    EXMEM_RD2,
    MemData,
    EXMEM_MemRead,
    EXMEM_MemWrite
);

assign PC_next =
Jump ? JumpAddr :
(EXMEM_Branch & EXMEM_Zero) ? EXMEM_BranchTarget :
PC_plus4;

/* MEM/WB */
reg [31:0] MEMWB_Mem, MEMWB_ALU;
reg [4:0] MEMWB_rd;
reg MEMWB_MemtoReg, MEMWB_RegWrite;

always @(posedge clk) begin
    MEMWB_Mem <= MemData;
    MEMWB_ALU <= EXMEM_ALU;
    MEMWB_rd <= EXMEM_rd;
    MEMWB_MemtoReg <= EXMEM_MemtoReg;

```

```

    MEMWB_RegWrite <= EXMEM_RegWrite;
end

assign WB_data =
    MEMWB_MemtoReg ? MEMWB_Mem : MEMWB_ALU;

HazardDetectionUnit HDU(
    IDEX_MemRead,
    IDEX_rt,
    IFID_instr[25:21],
    IFID_instr[20:16],
    reset,
    Stall
);
endmodule

```

```

module HEX_7SEG_DECODE(BIN, SSD);
    input [3:0] BIN;
    output reg [0:6] SSD;
    always begin
        case(BIN)
            0:SSD=7'b0000001;
            1:SSD=7'b1001111;
            2:SSD=7'b0010010;
            3:SSD=7'b0000110;
            4:SSD=7'b1001100;
            5:SSD=7'b0100100;
            6:SSD=7'b0100000;
            7:SSD=7'b0001111;
            8:SSD=7'b0000000;
            9:SSD=7'b0001100;
            10:SSD=7'b0001000;
            11:SSD=7'b1100000;
            12:SSD=7'b0110001;
            13:SSD=7'b1000010;
            14:SSD=7'b0110000;
            15:SSD=7'b0111000;
        endcase
    end
endmodule

```

```

module Program_Counter (clk, reset, PC_in, PC_out);
    input clk, reset;
    input [31:0] PC_in;
    output reg [31:0] PC_out;
    always @ (posedge clk or posedge reset)
    begin
        if(reset==1'b1)
            PC_out<=0;
        else

```

```

        PC_out<=PC_in;
    end
endmodule

module Adder32Bit(input1, input2, out);
    input [31:0] input1, input2;
    output [31:0] out;
    reg [31:0]out;
    always@( input1 or input2)
        begin
            out <= input1 + input2;
        end
endmodule

module alu(
    input [2:0] alufn,
    input [31:0] ra,
    input [31:0] rb_or_imm,
    output reg [31:0] aluout,
    output reg zero);
    parameter      ALU_OP_ADD  = 3'b000,
                   ALU_OP_SUB   = 3'b001,
                   ALU_OP_AND    = 3'b010,
                   ALU_OP_OR     = 3'b011,
                   ALU_OP_XOR    = 3'b100,
                   ALU_OP_LW     = 3'b101,
                   ALU_OP_SW     = 3'b110,
                   ALU_OP_BEQ    = 3'b111;

    always @(*) begin
        aluout = 0;
        zero = (ra == rb_or_imm);
        case(alufn)
            3'b000: aluout = ra + rb_or_imm;
            3'b001: aluout = ra - rb_or_imm;
            3'b010: aluout = ra & rb_or_imm;
            3'b011: aluout = ra | rb_or_imm;
        endcase
    end
endmodule

module Register_File (
    clk,
    read_addr_1,
    read_addr_2,
    write_addr,
    read_data_1,
    read_data_2,
    write_data,
    RegWrite
);
    input clk;
    input RegWrite;
    input [4:0] read_addr_1, read_addr_2, write_addr;
    input [31:0] write_data;

```

```

output [31:0] read_data_1, read_data_2;

reg [31:0] Regfile [31:0];
integer k;

// ===== INIT =====
initial begin
    for (k = 0; k < 32; k = k + 1)
        Regfile[k] = 32'b0;
end

// ===== READ (COMBINATIONAL) =====
assign read_data_1 = (read_addr_1 == 0) ? 32'b0 : Regfile[read_addr_1];
assign read_data_2 = (read_addr_2 == 0) ? 32'b0 : Regfile[read_addr_2];

// ===== WRITE (SEQUENTIAL) =====
always @(posedge clk) begin
    if (RegWrite && (write_addr != 0))
        Regfile[write_addr] <= write_data;
end

endmodule

module Data_Memory (addr, write_data, read_data, MemRead, MemWrite);
    input [31:0] addr;
    input [31:0] write_data;
    output [31:0] read_data;
    input MemRead, MemWrite;
    reg [31:0] DMemory [255:0];
    integer k;
    assign read_data = (MemRead) ? DMemory[addr] : 32'bx;
    initial begin
        for (k=0; k<64; k=k+1)
begin
            DMemory[k] = 32'b0;
            end
            DMemory[11] = 99;
        end
    always @(*)
begin
    if (MemWrite) DMemory[addr] = write_data;
end
endmodule

module Sign_Extension (sign_in, sign_out);
    input [15:0] sign_in;
    output [31:0] sign_out;
    assign sign_out[15:0]=sign_in[15:0];
    assign sign_out[31:16]=sign_in[15]?16'b1111_1111_1111_1111:16'b0;
endmodule

module Mux_32_bit (in0, in1, mux_out, select);
    parameter N = 32;

```

```

input [N-1:0] in0, in1;
output [N-1:0] mux_out;
input select;
assign mux_out = select? in1: in0 ;
endmodule

module Mux_5_bit (in0, in1, mux_out, select);
parameter N = 5;
input [N-1:0] in0, in1;
output [N-1:0] mux_out;
input select;
assign mux_out = select? in1: in0 ;
endmodule

module Instruction_Memory (read_address, instruction, reset);
input reset;
input [31:0] read_address;
output [31:0] instruction;

reg [31:0] Imemory [63:0];
integer k;

// word-addressed memory
assign instruction = Imemory[read_address];

always @(posedge reset) begin
    // clear memory
    for (k = 0; k < 64; k = k + 1)
        Imemory[k] = 32'b0;

    // =====
    // I-TYPE (addi)
    // =====

    // addi $s0, $zero, 5
    Imemory[0] = 32'b0001000_00000_10000_000000000000101;

    // addi $s1, $zero, 10
    Imemory[1] = 32'b0001000_00000_10001_0000000000001010;

    // =====
    // R-TYPE (add, sub, and, or)
    // =====

    // add $s2, $s0, $s1 ; 5 + 10 = 15
    Imemory[2] = 32'b0000000_10000_10001_10010_00000_100000;

    // sub $s3, $s1, $s0 ; 10 - 5 = 5
    Imemory[3] = 32'b0000000_10001_10000_10011_00000_100010;

    // and $s4, $s0, $s1
    Imemory[4] = 32'b0000000_10000_10001_10100_00000_100100;

    // or $s5, $s0, $s1
    Imemory[5] = 32'b0000000_10000_10001_10101_00000_100101;

```

```

// =====
// MEMORY (sw / lw)
// =====

// sw $s2, 0($zero)
Imemory[6] = 32'b101011_00000_10010_0000000000000000;

// lw $s3, 0($zero) ; load-use hazard
Imemory[7] = 32'b100011_00000_10011_0000000000000000;

// add $s4, $s3, $s0 ; needs forwarding or stall
Imemory[8] = 32'b000000_10011_10000_10100_00000_10000;

// =====
// BRANCH (beq)
// =====

// beq $s4, $s2, +2
Imemory[9] = 32'b000100_10100_10010_0000000000000010;

// addi $s0, $zero, 99 ; skipped if branch taken
Imemory[10] = 32'b001000_00000_10000_000000001100011;

// addi $s1, $zero, 77 ; skipped if branch taken
Imemory[11] = 32'b001000_00000_10001_000000001001101;

// =====
// JUMP
// =====

// j 15
Imemory[12] = 32'b000010_000000000000000000001111;

// addi $s2, $zero, 123 ; skipped by jump
Imemory[13] = 32'b001000_00000_10010_000000001111011;

// =====
// TARGET OF JUMP
// =====

// addi $s3, $zero, 55
Imemory[15] = 32'b001000_00000_10011_0000000000110111;

// NOPs
Imemory[16] = 32'b0;
Imemory[17] = 32'b0;
end
endmodule

module Sign_Extension_26 (sign_in, sign_out);
    input [25:0] sign_in;
    output [31:0] sign_out;
    assign sign_out[25:0]=sign_in[25:0];

```

```

assign sign_out[31:26]=sign_in[25]?6'b111111:6'b0;
endmodule

module control(
    input [5:0] op_code,
    output reg RegDst,
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg [2:0] ALUop,
    output reg MemWrite,
    output reg ALUsrc,
    output reg RegWrite,
    output reg Jump
);

always @(*) begin
    RegDst = 0; Branch = 0; MemRead = 0; MemtoReg = 0;
    ALUop = 3'b000; MemWrite = 0; ALUsrc = 0; RegWrite = 0; Jump = 0;

    case(op_code)

        6'b000000: begin // R-type
            RegDst = 1;
            RegWrite = 1;
            ALUop = 3'b000;
            Jump = 0;
        end

        6'b001000: begin // addi
            ALUsrc = 1;
            RegWrite = 1;
            ALUop = 3'b000;
        end

        6'b100011: begin // lw
            ALUsrc = 1;
            MemRead = 1;
            MemtoReg = 1;
            RegWrite = 1;
            ALUop = 3'b000;
        end

        6'b101011: begin // sw
            ALUsrc = 1;
            MemWrite = 1;
            ALUop = 3'b000;
        end

        6'b000100: begin // beq
            Branch = 1;
            ALUop = 3'b001;
        end

        6'b000010: begin // j

```

```

        Jump = 1;
    end

    endcase
end
endmodule

module ForwardingUnit(
    input EXMEM_RegWrite,
    input MEMWB_RegWrite,
    input [4:0] EXMEM_rd,
    input [4:0] MEMWB_rd,
    input [4:0] IDEX_rs,
    input [4:0] IDEX_rt,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB
);

always @(*) begin
    ForwardA = 2'b00;
    ForwardB = 2'b00;

    // EX hazard
    if (EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == IDEX_rs))
        ForwardA = 2'b10;

    if (EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == IDEX_rt))
        ForwardB = 2'b10;

    // MEM hazard
    if (MEMWB_RegWrite && (MEMWB_rd != 0) &&
        !(EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == IDEX_rs)) &&
        (MEMWB_rd == IDEX_rs))
        ForwardA = 2'b01;

    if (MEMWB_RegWrite && (MEMWB_rd != 0) &&
        !(EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == IDEX_rt)) &&
        (MEMWB_rd == IDEX_rt))
        ForwardB = 2'b01;
end
endmodule

module HazardDetectionUnit(
    input IDEX_MemRead,
    input [4:0] IDEX_rt,
    input [4:0] IFID_rs,
    input [4:0] IFID_rt,
    input reset,
    output reg Stall
);

always @(*) begin
    if (reset)
        Stall = 1'b0;
    else if (IDEX_MemRead &&

```

```

((INDEX_rt == IFID_rs) || (INDEX_rt == IFID_rt)) &&
(INDEX_rt != 0))
Stall = 1'b1;
else
    Stall = 1'b0;
end

endmodule

```

10.8 TESTBENCH

```

`timescale 1ns/1ps

module tb_project;

// =====
// DUT Signals
// =====
reg      CLOCK_50;
reg [17:0] SW;
wire [17:0] LEDR;
wire [7:0]  LEDG;
wire [0:6]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7;

// =====
// Instantiate DUT
// =====
project dut (
    .CLOCK_50(CLOCK_50),
    .SW(SW),
    .LEDG(LEDG),
    .LEDR(LEDR),
    .HEX0(HEX0),
    .HEX1(HEX1),
    .HEX2(HEX2),
    .HEX3(HEX3),
    .HEX4(HEX4),
    .HEX5(HEX5),
    .HEX6(HEX6),
    .HEX7(HEX7)
);

// =====
// Clock Generation (50 MHz)
// =====
initial begin
    CLOCK_50 = 0;
    forever #10 CLOCK_50 = ~CLOCK_50; // 20 ns period
end

// =====
// Drive CPU Clock from SW[17]
// =====
always @(*) begin
    SW[17] = CLOCK_50;

```

```

end

// =====
// Test Sequence
// =====
initial begin
    // Default switch values
    SW = 18'b0;

    // -----
    // Apply reset
    // -----
    SW[16] = 1'b1;
    #50;
    SW[16] = 1'b0;

    // -----
    // Show instruction on HEX
    // -----
    SW[0] = 1'b0;

    // Run for several cycles
    repeat (30) begin
        @(posedge CLOCK_50);
        display_pipeline_state();
    end

    // -----
    // Switch to result display
    // -----
    SW[0] = 1'b1;

    repeat (30) begin
        @(posedge CLOCK_50);
        display_pipeline_state();
    end

    // -----
    // End simulation
    // -----
    #100;
    $finish;
end

// =====
// Display Task
// =====
task display_pipeline_state;
begin
    $display("-----");
    $display("Time : %0t ns", $time);
    $display("PC : %h", LEDR);
    $display("Instr HEX : %b %b %b %b %b %b %b",
            HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
    $display("Stall : %b", LEDG[7]);

```

```

$display("ForwardA : %b", LEDG[1:0]);
$display("ForwardB : %b", LEDG[3:2]);
end
endtask

endmodule

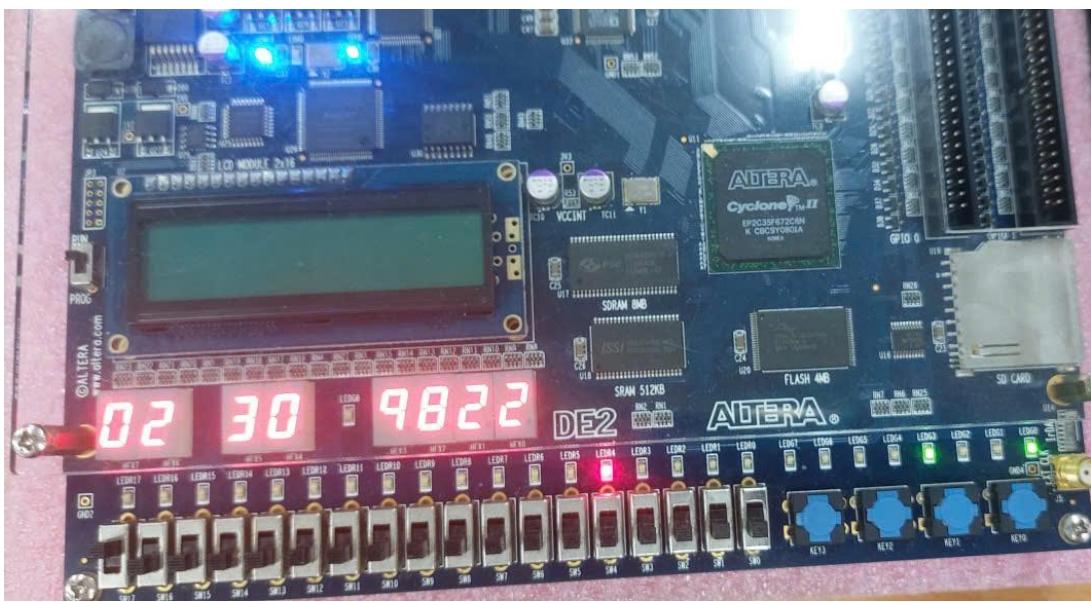
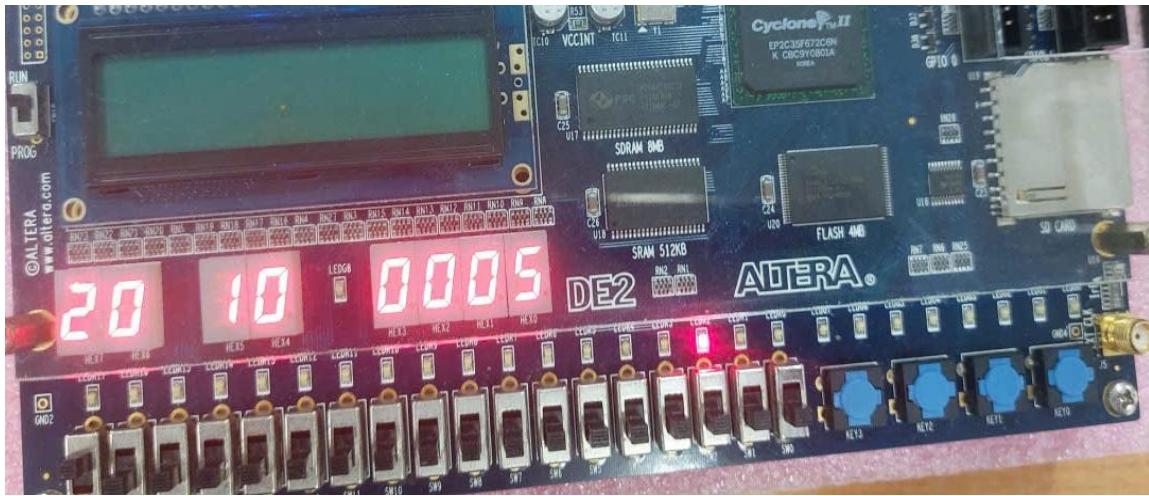
```

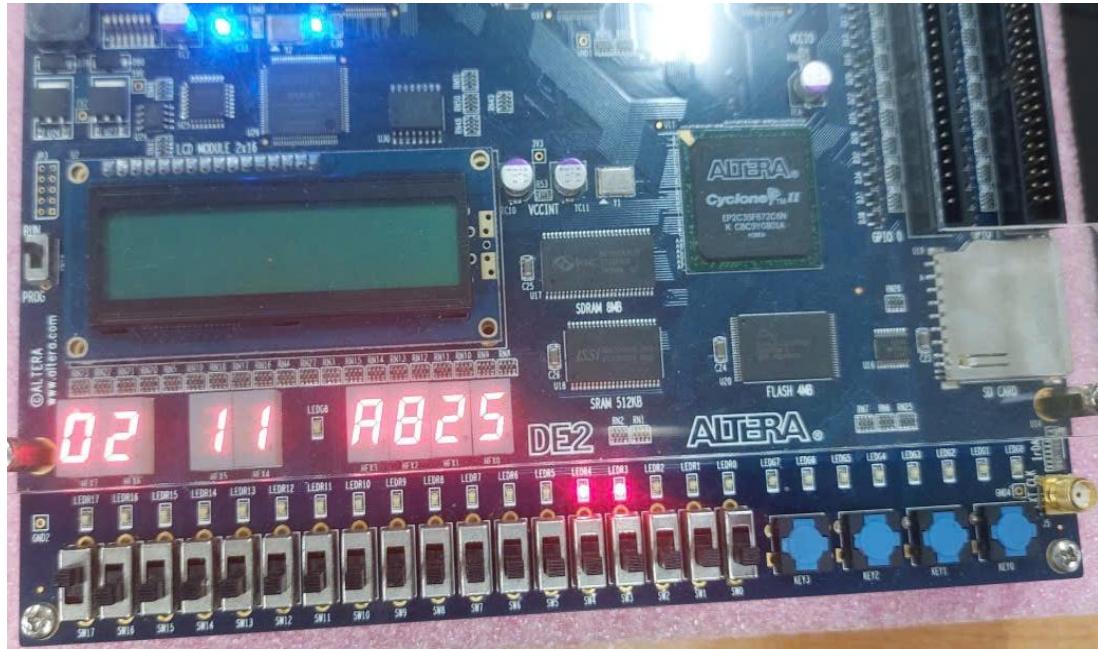
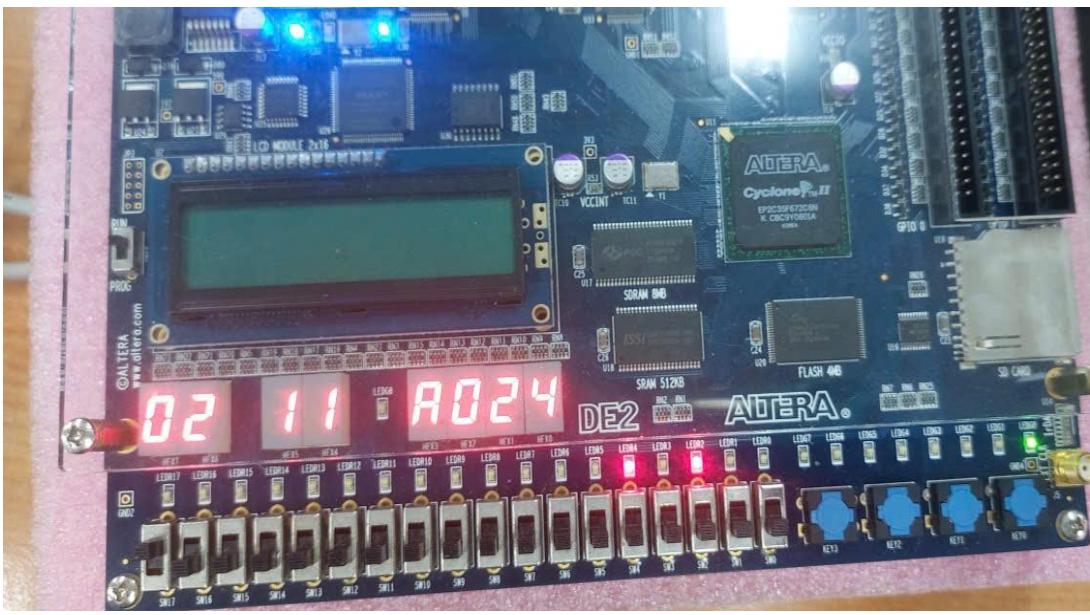
10.9 Results

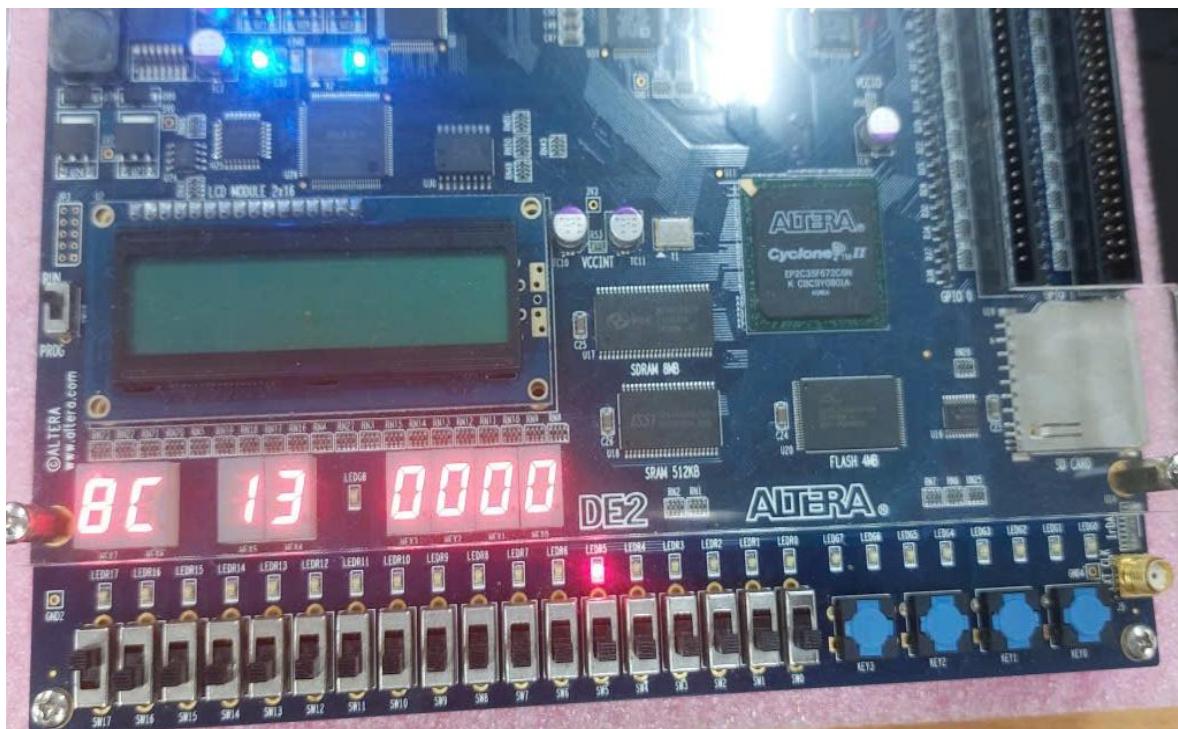
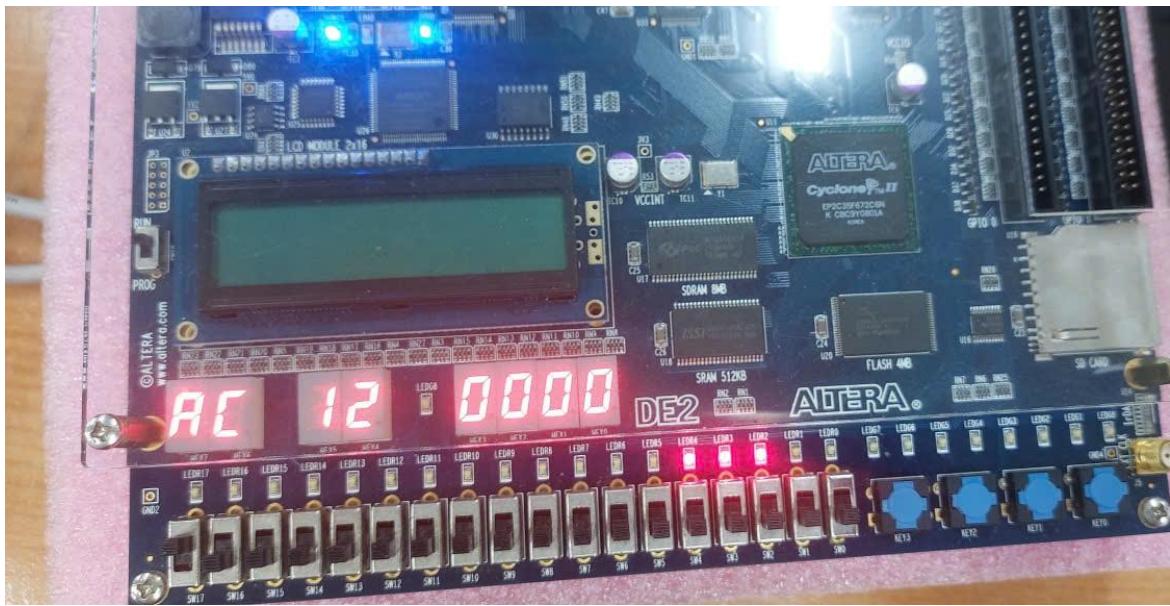
Addr	Assembly	Binary (summary)	**HEX**
0	` addi \$s0, \$zero, 5`	I-type	**0x20100005**
1	` addi \$s1, \$zero, 10`	I-type	**0x2011000A**
2	` add \$s2, \$s0, \$s1`	R-type	**0x02119020**
3	` sub \$s3, \$s1, \$s0`	R-type	**0x02309822**
4	` and \$s4, \$s0, \$s1`	R-type	**0x0211A024**
5	` or \$s5, \$s0, \$s1`	R-type	**0x0211A825**
6	` sw \$s2, 0(\$zero)`	I-type	**0xAC120000**
7	` lw \$s3, 0(\$zero)`	I-type	**0x8C130000**
8	` add \$s4, \$s3, \$s0`	R-type	**0x0270A020**
9	` beq \$s4, \$s2, +2`	I-type	**0x12920002**
10	` addi \$s0, \$zero, 99`	I-type	**0x20100063**
11	` addi \$s1, \$zero, 77`	I-type	**0x2011004D**
12	` j 15`	J-type	**0x0800000F**
13	` addi \$s2, \$zero, 123`	I-type	**0x2012007B**
15	` addi \$s3, \$zero, 55`	I-type	**0x20130037**
16+	` nop`	-	**0x00000000**

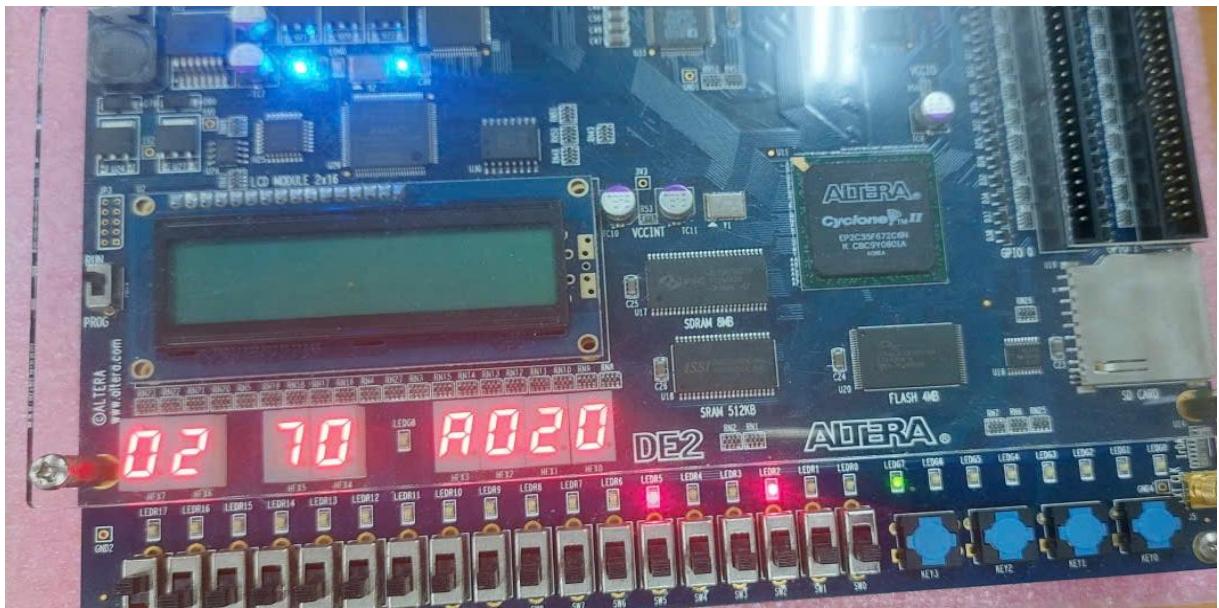
TEST ON FPGA



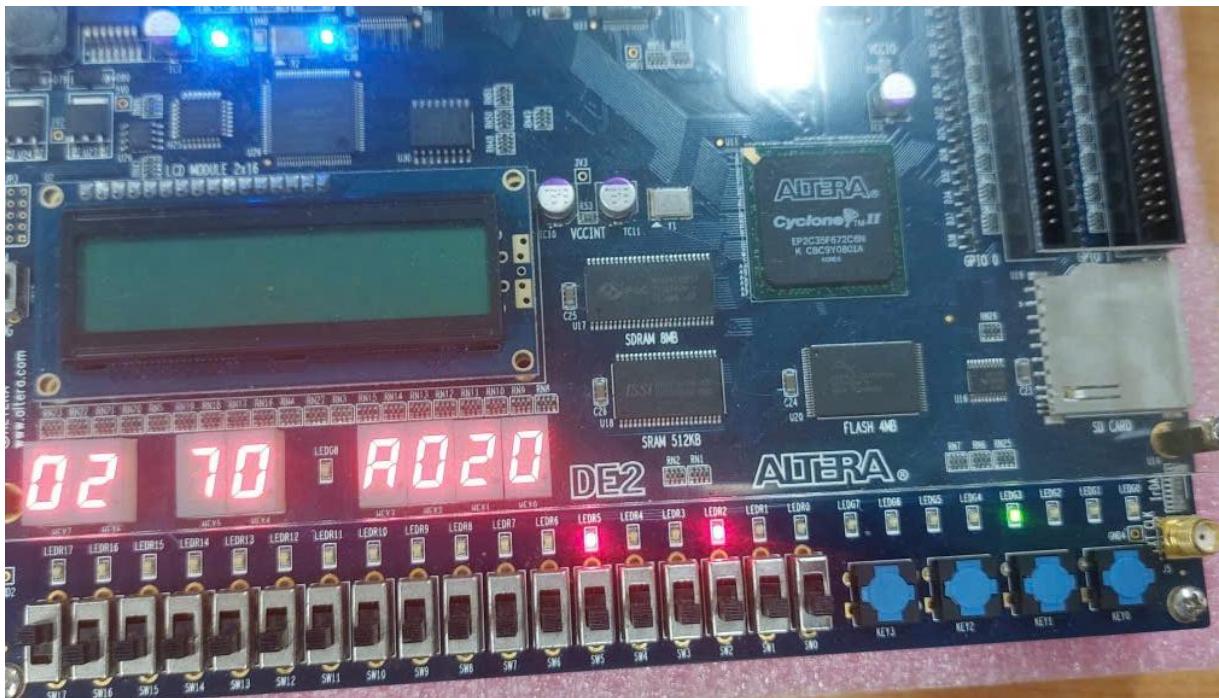


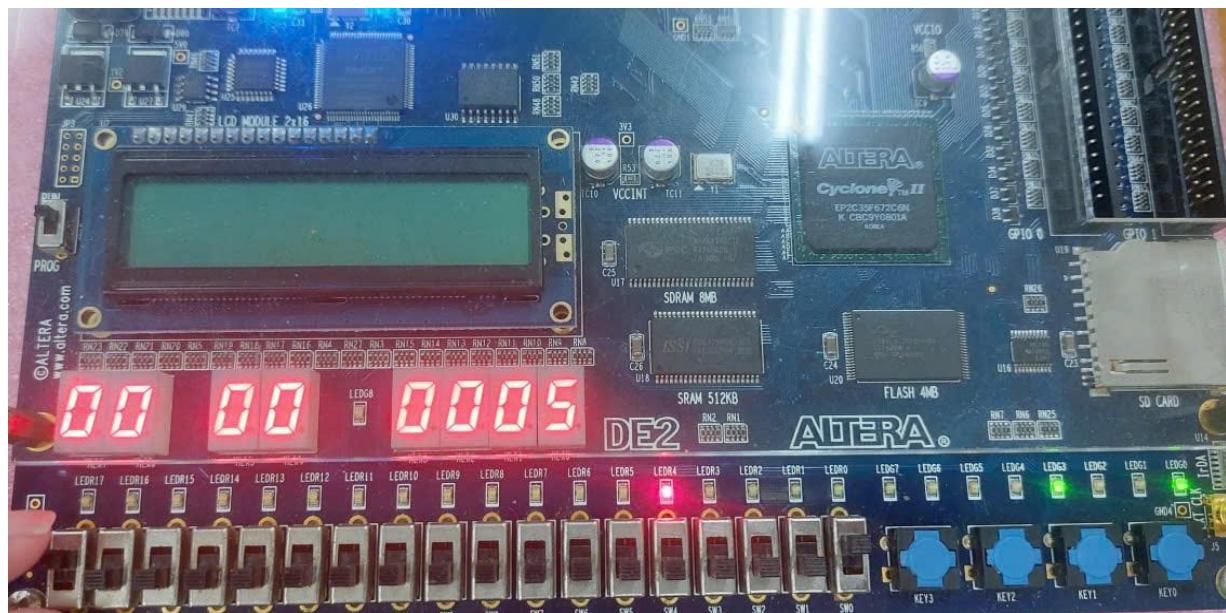
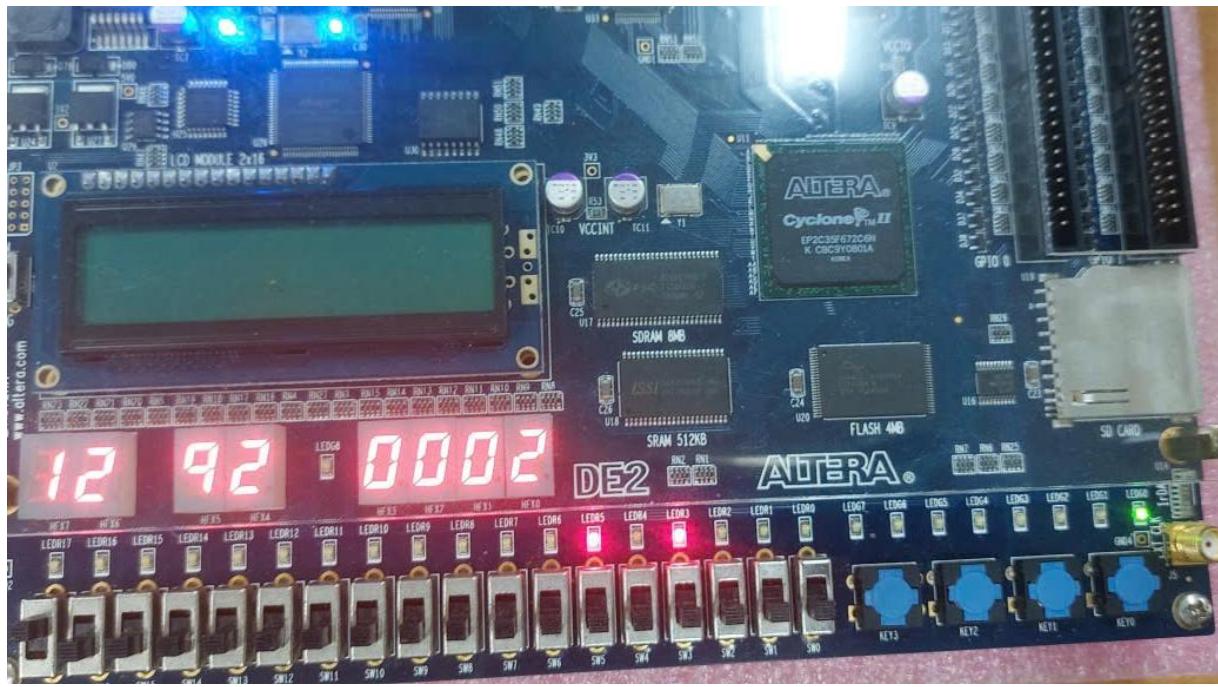


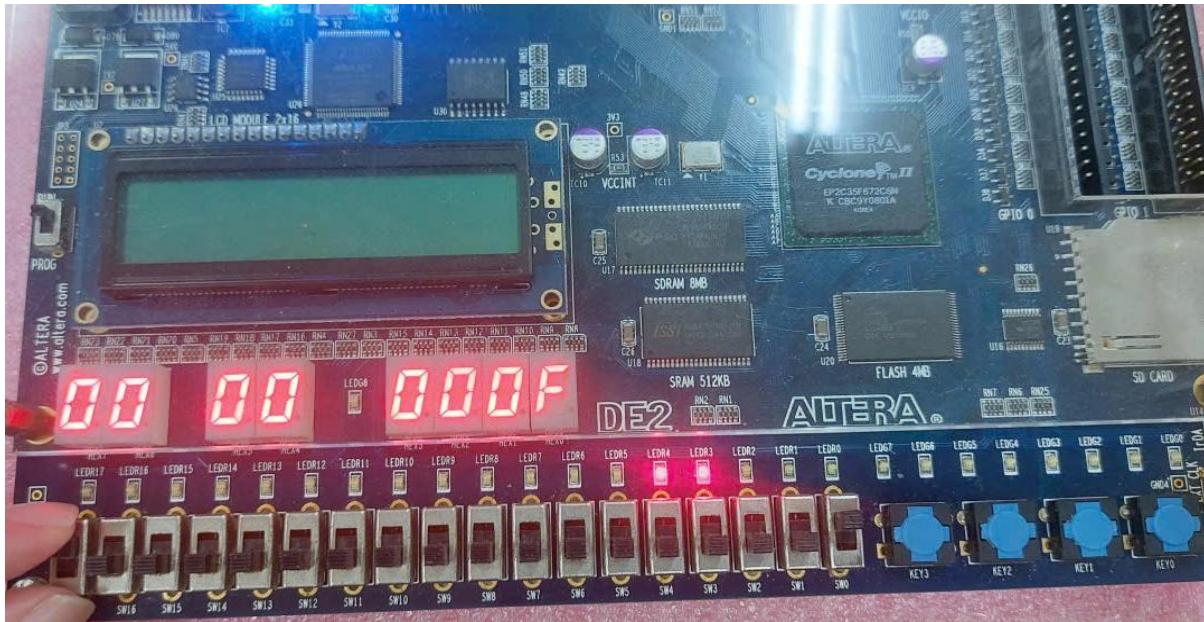
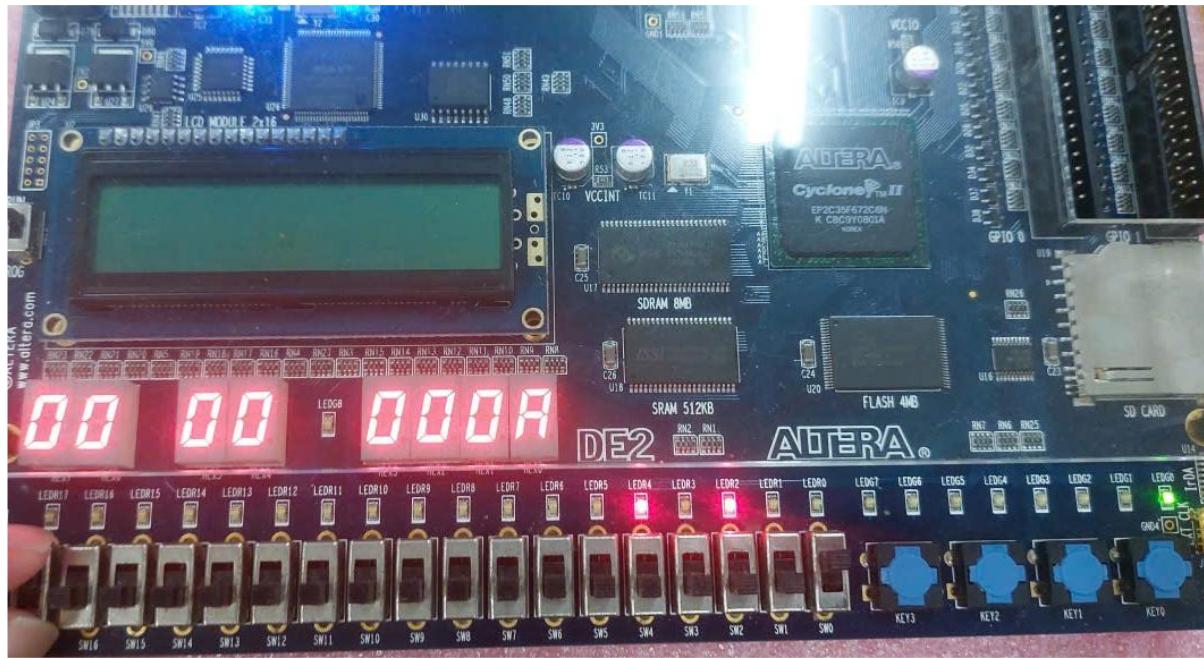




HAZARD HAPPEN







11. Conclusion and Future Work

This project successfully designed, implemented, and verified a **five-stage pipelined MIPS processor** using Verilog and FPGA hardware. The processor follows the standard MIPS pipeline structure and incorporates essential mechanisms such as hazard detection, data forwarding, and pipeline control to ensure correct and efficient instruction execution.

11.1 Project Summary

The main objectives of the project were achieved as follows:

- A complete five-stage pipelined datapath was designed and implemented.
- Data and control hazards were correctly handled using stalling, forwarding, and flushing techniques.
- The processor was successfully integrated with FPGA input and output peripherals.
- Functional correctness was verified through simulation and hardware testing.

Each pipeline stage—Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back—operated as intended, with correct data and control signal propagation.

11.2 Learning Outcomes

Through this project, a deeper understanding of:

- RISC processor architecture and pipelining concepts
- Pipeline hazards and their resolution techniques
- Hardware description using Verilog
- FPGA-based debugging and verification

was developed. Observing pipeline behavior in real time using FPGA LEDs and HEX displays provided valuable insight into the internal operation of a pipelined processor.

11.3 System Reliability and Performance

The implemented hazard handling mechanisms ensured reliable execution under all tested conditions. Forwarding minimized unnecessary pipeline stalls, while stalling and flushing maintained correctness when hazards could not be avoided.

Although the design does not include advanced performance optimizations, it achieves a stable and efficient pipeline suitable for educational and experimental purposes.

11.4 Future Work

Several enhancements can be considered to improve the processor in future iterations:

- **Branch Prediction**
Implementing static or dynamic branch prediction to reduce control hazard penalties.
- **Expanded Instruction Set**
Supporting additional MIPS instructions, such as multiplication, division, and shift operations.
- **Pipeline Optimization**
Resolving branch decisions earlier in the pipeline to improve performance.
- **Cache Integration**
Adding instruction and data caches to reduce memory access latency.
- **Performance Measurement**
Incorporating performance counters to measure CPI and instruction throughput.

11.5 Final Remarks

This project demonstrates a complete and functional implementation of a pipelined MIPS processor, bridging theoretical concepts and practical hardware realization. The design provides a solid foundation for further exploration of advanced processor architectures and optimization techniques.