PERFORMANCE ENHANCEMENT IN A PIPELINED RISC-V CPU USING A TWO-BIT BRANCH PREDICTOR

A Project Report

Submitted to the APJ Abdul Kalam Technological University

in partial fulfillment of requirements for the award of degree

Bachelor of Technology

in

Electronics and Communication Engineering

by

VINAYAK PRAKASH(LTVE22EC074)

SHAHAL SIYADH M P(LTVE22EC073)

PRASOON PRADEEP(LTVE22EC072)

DIWA TOMY(TVE22EC024)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

KERALA

April 2025

DEPT. OF ELECTRONICS & COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

2022 - 26

CERTIFICATE

This is to certify that the report entitled **PERFORMANCE ENHANCEMENT IN A PIPELINED RISC-V CPU USING A TWO-BIT BRANCH PREDICTOR** submitted by **VINAYAK PRAKASH** (LTVE22EC074), **SHAHAL SIYADH M P** (LTVE22EC073), **PRASOON PRADEEP** (LTVE22EC072) & **DIWA TOMY** (TVE22EC024) to the APJ Abdul Kalam Technological University in partial fulfillment of the B.Tech. degree in Electronics and Communication Engineering is a bonafide record of the project work carried out by him under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Dr. Nikhil M
(Project Guide)
Assistant Professor
Dept.of ECE
College of Engineering
Trivandrum

Dr. Arun Varghese
(Project Coordinator)
Assistant Professor
Dept.of ECE
College of Engineering
Trivandrum

Dr. Haris P A
Professor and Head
Dept.of ECE
College of Engineering
Trivandrum

DECLARATION

We hereby declare that the project report **PERFORMANCE ENHANCEMENT IN A PIPELINED RISC-V CPU USING A TWO-BIT BRANCH PREDICTOR**, submitted for partial fulfillment of the requirements for the award of degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by us under supervision of Dr. Nikhil M

This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the original sources.

We also declare that I have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

VINAYAK PRAKASH
SHAHAL SIYADH M P
PRASOON PRADEEP
DIWA TOMY

Trivandrum
04-04-2025

Abstract

In modern processor design, pipelining is a fundamental technique to improve instruction throughput. However, control hazards introduced by branch instructions often lead to performance degradation due to frequent pipeline stalls. This project presents the implementation and simulation of a pipelined RISC-V CPU enhanced with a two-bit dynamic branch predictor to mitigate these control hazards. The CPU was designed and tested using Vivado 2023.1, without deployment on physical FPGA hardware. The branch predictor utilizes a four-state finite state machine to accurately predict the outcome of conditional branches, thereby reducing the number of pipeline flushes. Simulation results show a significant performance improvement—reducing the execution time of a sample program from 325 ns to 265 ns. The observed gain demonstrates the effectiveness of the branch predictor, especially in loop-intensive workloads. This work emphasizes the importance of prediction mechanisms in CPU architectures and sets a foundation for more advanced prediction techniques in future designs.

Acknowledgement

We take this opportunity to express my deepest sense of gratitude and sincere thanks to everyone who helped us to complete this work successfully. We express our sincere thanks to Dr. Haris P A, Head of Department, Electronics and Communication Engineering, College of Engineering Trivandrum for providing us with all the necessary facilities and support.

We would like to express my sincere gratitude to the Dr. Arun Varghese, department of Electronics and Communication Engineering, College of Engineering Trivandrum Trivandrum for the support and co-operation.

We would like to place on record my sincere gratitude to our project guide Dr. Nikhil M, Assistant Professor, Electronics and Communication Engineering, College of Engineering Trivandrum for the guidance and mentorship throughout this work.

Finally I thank my family, and friends who contributed to the succesful fulfilment of this seminar work.

VINAYAK PRAKASH

SHAHAL SIYADH M P

PRASOON PRADEEP

DIWA TOMY

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

RISC-V (Reduced Instruction Set Computing – Five) is an open-source instruction set architecture (ISA) that has gained significant traction due to its simplicity, modularity, and extensibility. The RV32I base ISA provides a foundational 32-bit instruction set suitable for educational, embedded, and general-purpose processor designs.

In this project, we have designed and implemented a pipelined RISC-V CPU based on the RV32I instruction set. Pipelining is a fundamental microarchitectural technique that allows overlapping execution of instructions, significantly increasing instruction throughput and improving overall CPU performance. Our CPU design follows a classic 5-stage pipeline architecture: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

To further enhance performance, particularly for control flow instructions such as conditional branches and jumps, we have integrated a branch predictor into our design. Branch prediction is critical in pipelined architectures, as it helps minimize control hazards by speculatively determining the direction of branch instructions before they are resolved. This reduces pipeline stalls and maintains a smooth instruction flow.

The CPU was implemented using Verilog HDL and verified through extensive simulation. The project encompassed designing all major processor modules, handling data and control hazards, and implementing a prediction mechanism for improving branch handling efficiency. The result is a fully functional and efficient pipelined RISC-V CPU capable of executing standard RV32I instructions with improved performance due to branch prediction.

1.1 MOTIVATION AND RELEVANCE

With the rapid advancement of computing systems and the growing demand for open and customizable hardware platforms, RISC-V has emerged as a revolutionary ISA that provides flexibility, openness, and scalability. Unlike proprietary ISAs, RISC-V is free to use and modify, making it highly relevant for academic research, teaching, and industry applications. Designing a processor from scratch not only deepens our understanding of computer architecture but also gives hands-on experience with core hardware concepts such as pipelining, hazard management, and control flow handling.

This project was driven by the motivation to bridge the gap between theoretical knowledge and practical implementation in digital design. Implementing a pipelined CPU provided an opportunity to explore how modern processors achieve high performance through instruction-level parallelism. Moreover, integrating a branch predictor reflects real-world processor optimizations, offering insight into how CPUs handle uncertainty in control flow to maintain efficiency.

The relevance of this project lies in its educational value and its alignment with industry trends. As the demand for RISC-V-based processors grows, the ability to design and optimize such architectures is becoming increasingly valuable. This project not only reinforces foundational concepts in computer architecture but also prepares us for advanced work in processor design and embedded systems development.

1.2 PROBLEM STATEMENT

Modern processors must execute instructions efficiently while minimizing delays caused by pipeline hazards, particularly control hazards introduced by branch instructions. In a pipelined architecture, branch instructions can disrupt the steady flow of instructions, leading to pipeline stalls and performance degradation.

The objective of this project is to design and implement a pipelined 32-bit RISC-V CPU based on the RV32I instruction set, capable of executing a wide range of instructions efficiently. A key challenge addressed in this project is the reduction of control hazards by integrating a branch prediction mechanism that can anticipate the outcome of branch instructions and maintain pipeline flow with minimal disruption.

This project aims to solve the following core problems:

1. How to design and implement a functional pipelined RISC-V processor with support for RV32I instructions.
2. How to handle data and control hazards in a pipelined architecture.
3. To analyze and resolve data and control hazards within the pipeline.
4. How to implement a branch predictor that improves performance by reducing stalls due to control dependencies.

By addressing these challenges, the project enhances our understanding of pipelined CPU design and introduces techniques used in modern processor optimization.

1.3 ORGANISATION OF THE PROJECT REPORT

This project report is organized into the following chapters:

- **Chapter 1: Introduction**
  Provides an overview of the RISC-V architecture and the motivation behind designing a pipelined CPU with branch prediction. It highlights the importance of the RV32I instruction set and pipelining in modern processor design.
- **Chapter 2: Motivation and Relevance**
  Explains the motivation for undertaking this project and its relevance in the context of current trends in processor design and education.
- **Chapter 3: Problem Statement**
  Clearly defines the problem being addressed, emphasizing the need for efficient pipeline execution and control hazard mitigation through branch prediction.
- **Chapter 4: Literature Review**
  Existing CPU architectures and their features,Basics of pipelining and ,branch-prediction strategies,Prior work, if any, on similar topics.

- **Chapter 5: System Architecture and Design**

  Describes the architectural design of the pipelined RISC-V CPU, the pipeline stages, hazard detection units, forwarding logic, and the branch prediction mechanism.

- **Chapter 6: Implementation**

  Details the implementation of the CPU in Verilog HDL, module-wise functionality, and integration of components.

- **Chapter 7: Simulation and Testing**

  Discusses the testing methodology, simulation results, and verification of the CPU using test programs and waveform analysis.

- **Chapter 8: Results and Discussion**

  Presents the performance outcomes, benefits of the branch predictor, and comparison of execution with and without prediction.

- **Chapter 9: Conclusion and Future Work**

  Summarizes the project, key learnings, and proposes directions for future improvements or extensions to the CPU design.

Chapter 2

LITERATURE REVIEW

The design of microprocessors has evolved significantly over the decades, with a major focus on enhancing performance through techniques like pipelining and branch prediction. The Reduced Instruction Set Computing (RISC) philosophy, first proposed in the 1980s, led to simplified instruction sets that are easier to pipeline and optimize. RISC-V, an open standard ISA developed at the University of California, Berkeley, follows these principles while offering modularity and extensibility, making it a popular choice for academic and industrial processor design.

Pipelining is a well-established technique that increases instruction throughput by overlapping the execution of multiple instructions. Hennessy and Patterson's work on computer architecture laid the foundation for understanding pipeline stages, hazards (data, structural, and control), and techniques like forwarding and hazard detection units that allow efficient pipeline execution. In a 5-stage pipeline — comprising Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) — each instruction passes through stages sequentially, allowing multiple instructions to be processed simultaneously

2.1 Branch Prediction Techniques

Branch prediction is a fundamental technique used in pipelined processor architectures to improve performance by guessing the outcome of branch instructions before they are resolved. An accurate branch predictor minimizes control hazards, reduces pipeline stalls, and increases instruction throughput. Various strategies have been developed

over time, ranging from simple static approaches to complex hybrid mechanisms. This section provides a detailed overview of these strategies.

2.1.1 1. Static Branch Prediction

Static branch prediction does not rely on runtime behavior. The decision to predict a branch as taken or not taken is fixed and does not change with execution history. Common static strategies include:

- **Always Taken**: This method assumes that every branch will be taken. It works well in loop-heavy code where backward branches are common but performs poorly for conditional branches that are often not taken.
- **Always Not Taken**: This predictor assumes that branches will not be taken. It is simple but generally inaccurate in loops and branching structures.
- **Backward Taken, Forward Not Taken (BTFNT)**: Based on the idea that backward branches (loops) are usually taken and forward branches (conditionals) are not taken. This is slightly more accurate than purely taken/not taken strategies.

While static methods are easy to implement and cost-effective in hardware, their prediction accuracy is typically low compared to dynamic approaches.

2.1.2 2. Dynamic Branch Prediction

Dynamic prediction strategies use the history of previous branch outcomes to make predictions. These methods are more accurate and widely used in modern CPUs.

2.1 One-Bit Predictor

The one-bit predictor stores the outcome of the last execution of a branch (taken or not taken). Each branch is associated with a single bit in a Branch History Table (BHT).

Working: - If the last time the branch was taken, predict taken. - If the last time it was not taken, predict not taken.

Limitation: Fails in loop-exit scenarios, where the branch is mostly taken but not taken at the end. The prediction bit flips after a single misprediction, causing inaccuracy.

2.2 Two-Bit Saturating Counter Predictor

This predictor improves upon the one-bit scheme by requiring two consecutive mispredictions before changing the prediction. It uses a 2-bit finite state machine (FSM) with four states:

- Strongly Taken
- Weakly Taken
- Weakly Not Taken
- Strongly Not Taken

Working: - If a branch is taken, the FSM moves toward the ”strongly taken” state. - If not taken, it moves toward ”strongly not taken.” - The prediction changes only when the FSM crosses the center threshold (i.e., two wrong predictions in a row).

Advantage: Reduces the effect of temporary changes in branch behavior, improving accuracy for loop-heavy code.

2.3 Branch History Table (BHT)

A BHT is a memory table that keeps track of recent outcomes of branches, typically indexed by the lower bits of the Program Counter (PC).

Working: - The BHT stores prediction bits for each indexed branch. - It can use either 1-bit or 2-bit predictors.

Note: Collisions can occur if different branches map to the same index (aliasing), which may reduce accuracy.

2.4 Global Branch Predictor

This predictor uses a single **Global History Register (GHR)** to record the outcomes of recent branches (e.g., last 8 taken/not taken outcomes). This GHR is then used to index into a **Pattern History Table (PHT)** containing the actual prediction bits.

Working: - Captures correlations between different branches. - Example: if one branch is often taken only after another is taken, this relationship can be learned.

Advantage: High accuracy in structured code with repeated patterns.

2.5 Local Branch Predictor

Unlike global predictors, local predictors use the past history of a specific branch (not all branches). A table stores local histories for each branch (based on PC), and each history is used to index into its own pattern table.

Advantage: Better for branches with their own unique behavior not related to other branches.

2.1.3 3. Advanced Prediction Techniques

3.1 Tournament (Hybrid) Predictors

Tournament predictors combine multiple predictors (e.g., local and global) and select the best-performing one using a **meta-predictor**.

Working: - Both local and global predictions are computed. - A selector chooses the more accurate predictor based on recent outcomes.

Advantage: - Offers the highest accuracy by leveraging multiple strategies. - Used in many modern high-performance CPUs (e.g., Intel, AMD).

3.2 Branch Target Buffer (BTB)

A BTB is used in conjunction with branch prediction to store the **target addresses** of previously taken branches.

Working: - When a branch is predicted taken, the BTB provides the next instruction address (target). - Reduces the penalty of misfetching instructions.

3.3 Loop Predictors

Specialized predictors designed to detect loop patterns, particularly useful for predicting loop exits. These maintain loop iteration counts and predict not taken when the loop is likely to end.

3.4 Neural Predictors

Recent research has introduced neural network-based predictors (like perceptron predictors) that can learn complex branch behavior patterns over longer histories.

Limitation: - High hardware complexity - Still largely experimental or limited to research CPUs

2.1.4 Summary of Prediction Strategies

Prediction Method	Type	Accuracy Level	Estimated Accuracy (%)
Always Taken / Not Taken	Static	Low	40–60%
1-Bit Predictor	Dynamic	Moderate	70–80%
2-Bit Predictor	Dynamic	Good	80–90%
Branch History Table (BHT)	Dynamic	Good	85–90%
Global Predictor	Dynamic	High	90–94%
Local Predictor	Dynamic	High	90–94%
Tournament Predictor	Hybrid	Very High	94–98%
Neural Predictor	Advanced	Research-level	95–99%

Table 2.1: Comparison of Branch Prediction Techniques with Estimated Accuracy

Chapter 3

SYSTEM ARCHITECTURE AND DESIGN

This chapter provides a comprehensive overview of the architectural design of the pipelined RISC-V CPU developed in this project. It covers the pipeline stages, control logic, datapath components, and the integration of the branch prediction unit. The goal is to achieve high performance with efficient hazard handling and accurate branch prediction.

3.1 Overview of the CPU Design

The core of our design is a high-performance **5-stage pipelined RISC-V processor** based on the RV32I instruction set architecture. This classic pipeline structure is adopted to increase instruction throughput by allowing multiple instructions to be processed simultaneously at different stages.

Pipeline Stages

- ➤ **Instruction Fetch (IF):** Fetches the instruction from memory using the program counter (PC).
- ➤ **Instruction Decode (ID):** Decodes the instruction and reads operands from the register file.
- ➤ **Execution (EX):** Performs arithmetic or logic operations, calculates memory addresses, or determines branch targets.
- ➤ **Memory Access (MEM):** Accesses data memory for load/store instructions.
- ➤ **Write Back (WB):** Writes the result back to the register file.

This pipelined architecture allows one instruction to be completed every clock cycle (ideal case), significantly improving overall CPU throughput compared to non-pipelined designs.

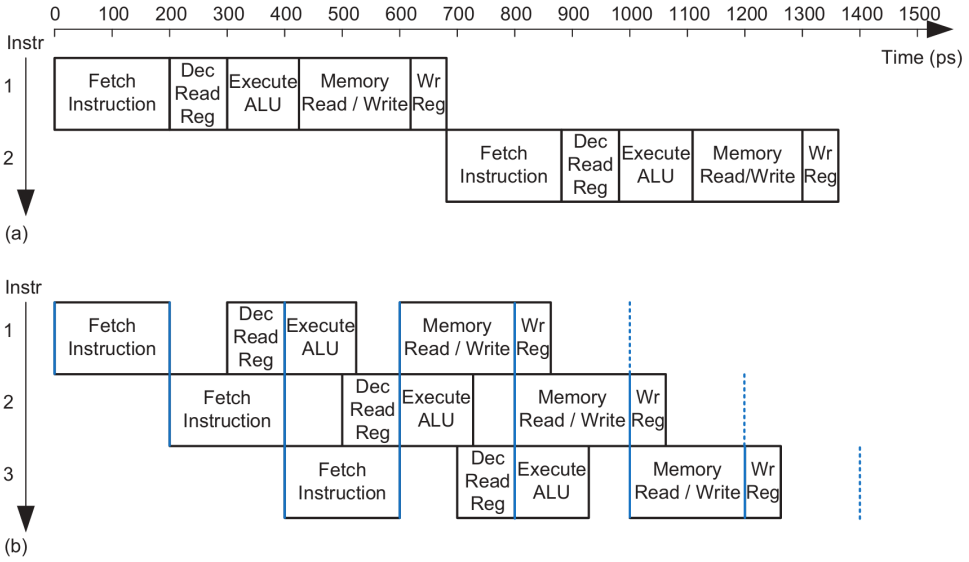

Figure 3.1: Comparison of Non-Pipelined (a) vs. Pipelined (b) Execution

Figure 3.1 illustrates the execution of instructions in a non-pipelined processor (a) and a pipelined processor (b).

- In the **non-pipelined execution** (a), each instruction must complete all its stages before the next instruction begins, leading to longer execution time.

- In the **pipelined execution** (b), multiple instructions are executed in parallel, with each instruction at a different stage of execution, significantly improving throughput.
- The pipelined approach results in better resource utilization and increased instruction throughput per clock cycle, reducing the total execution time.

3.2 Instruction Set Architecture (ISA)

The implemented processor follows the **RV32I** (32-bit base integer) instruction set of the RISC-V architecture. RV32I is a reduced instruction set (RISC) architecture that supports simple and efficient instruction execution.

3.2.1 Instruction Formats

RISC-V instructions are encoded in a **fixed 32-bit format** with different layouts based on the operation. The main instruction formats in RV32I are:

- **R-Type:** Used for register-to-register operations.
- **I-Type:** Used for immediate operations, load instructions, and some control instructions.
- **S-Type:** Used for store instructions.
- **B-Type:** Used for conditional branch instructions.
- **U-Type:** Used for upper immediate instructions.
- **J-Type:** Used for jump instructions.

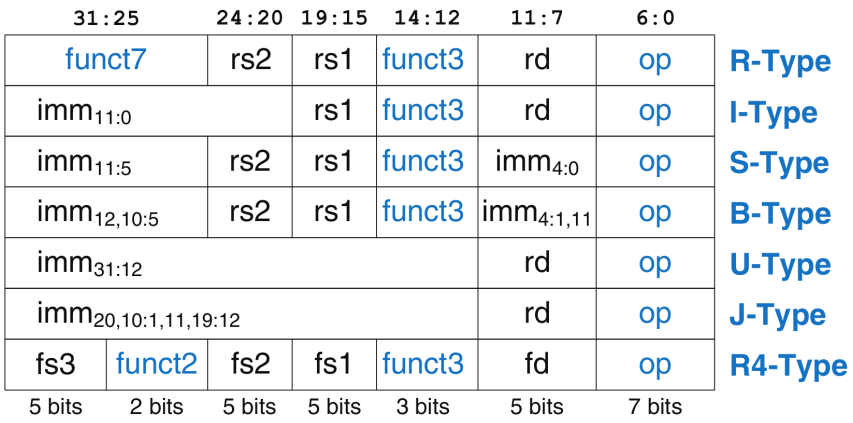

Figure 3.2: RISC-V RV32I Instruction Formats

3.2.2 Instruction Categories

The RV32I instruction set consists of six primary instruction categories:

1. **Arithmetic and Logical Instructions**
2. **Load and Store Instructions**
3. **Branch Instructions**
4. **Jump Instructions**
5. **Immediate Instructions**
6. **System Instructions**

3.2.3 RV32I Instructions

Table 3.1 lists the primary instructions of the RV32I instruction set, categorized based on their type and operation.

Category	Instruction	Format	Description
Arithmetic	ADD	R-Type	Adds two registers
	SUB	R-Type	Subtracts two registers
	AND	R-Type	Bitwise AND of two registers
	OR	R-Type	Bitwise OR of two registers
	XOR	R-Type	Bitwise XOR of two registers
	SLL	R-Type	Shift left logical
	SRL	R-Type	Shift right logical
	SRA	R-Type	Shift right arithmetic
Load	LW	I-Type	Load word from memory
	LH	I-Type	Load halfword from memory
	LB	I-Type	Load byte from memory
	LHU	I-Type	Load halfword unsigned
	LBU	I-Type	Load byte unsigned
	LUI	U-Type	Load upper immediate
	AUIPC	U-Type	Add upper immediate to PC
Store	SW	S-Type	Store word to memory
	SH	S-Type	Store halfword to memory
	SB	S-Type	Store byte to memory
Branch	BEQ	B-Type	Branch if equal
	BNE	B-Type	Branch if not equal
	BLT	B-Type	Branch if less than
	BGE	B-Type	Branch if greater or equal
	BLTU	B-Type	Branch if less than (unsigned)
	BGEU	B-Type	Branch if greater or equal (unsigned)
Jump	JAL	J-Type	Jump and link
	JALR	I-Type	Jump and link register
Immediate	ADDI	I-Type	Add immediate
	ANDI	I-Type	Bitwise AND with immediate
	ORI	I-Type	Bitwise OR with immediate
	XORI	I-Type	Bitwise XOR with immediate
	SLTI	I-Type	Set less than immediate

Table 3.1: Extended RV32I Instruction Set Summary

3.2.4 Instruction Execution Pipeline

The execution of RV32I instructions follows the 5-stage pipeline discussed earlier. Arithmetic and logical instructions typically complete execution in a single cycle, while load/store and branch instructions may introduce pipeline hazards.

3.3 CPU Architecture Diagrams

To provide a visual overview of the design evolution of our RISC-V processor, this section presents the different architectural stages — from a simple single-cycle CPU to an advanced pipelined CPU with hazard detection and branch prediction mechanisms.

3.3.1 Single-Cycle CPU Architecture

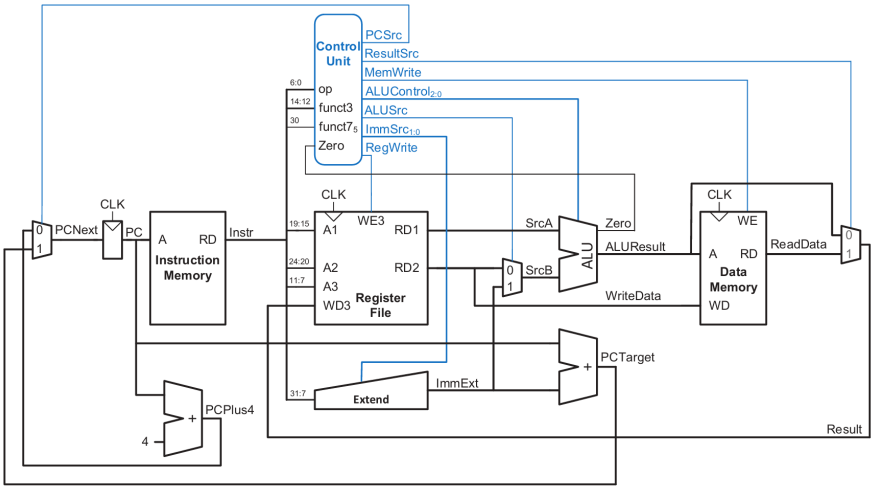

Figure 3.3: Single-Cycle CPU Architecture

In a single-cycle CPU, every instruction is executed in one clock cycle. While simple and easy to implement, it suffers from low clock speed due to the need to accommodate the longest instruction path in one cycle.

3.3.2 Pipelined CPU Architecture

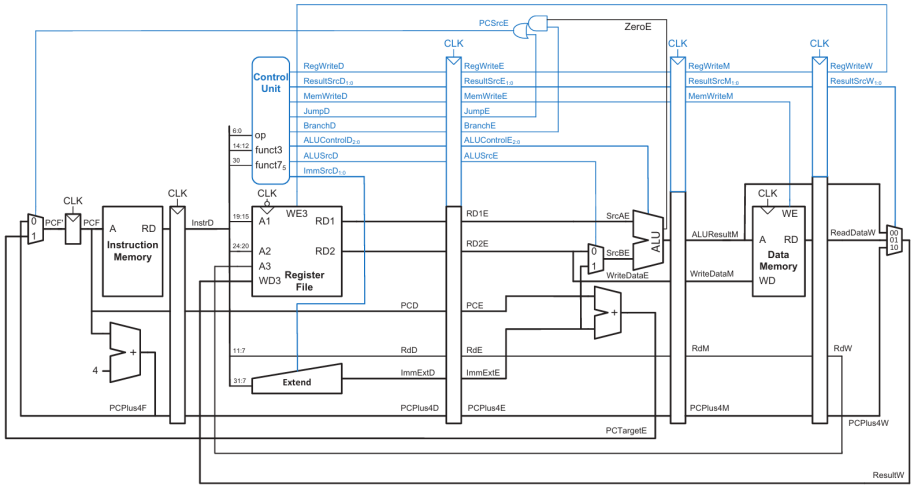

Figure 3.4: Pipelined CPU Architecture

Pipelining divides instruction execution into stages, allowing multiple instructions to be processed concurrently. This significantly improves throughput compared to the single-cycle approach.

3.3.3 Pipelined CPU with Hazard Detection Unit

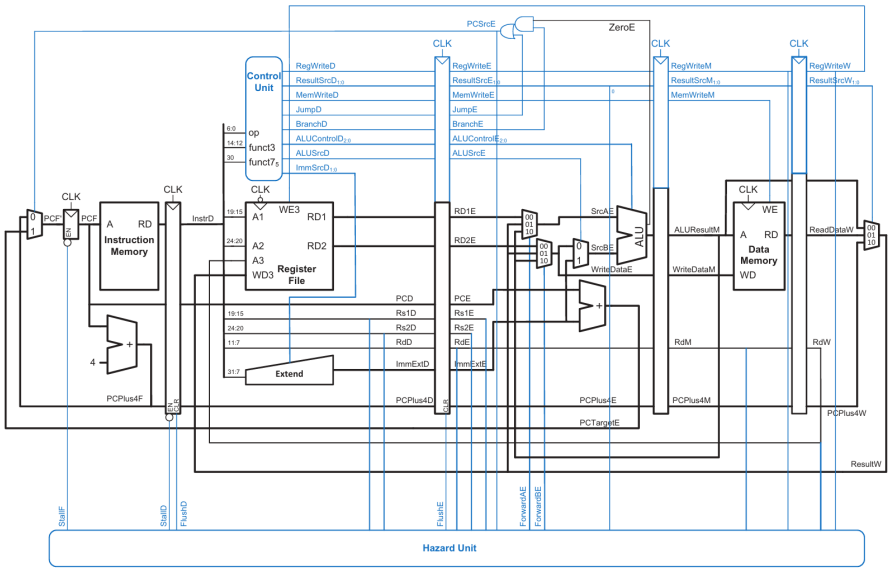

Figure 3.5: Pipelined CPU with Hazard Detection Unit

To ensure correct execution, a hazard detection unit is introduced to manage data and control hazards. It detects dependencies and introduces stalls or forwarding to maintain instruction correctness.

3.3.4 Pipelined CPU with Branch Predictor

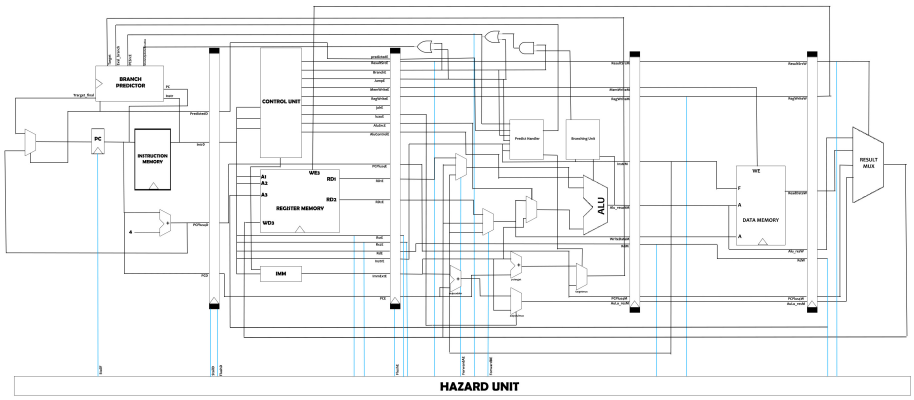

Figure 3.6: Pipelined CPU with Branch Predictor

The final version of our CPU integrates a branch predictor. This module speculatively executes instructions based on predicted branch outcomes, reducing control stalls and enhancing pipeline efficiency.

3.4 Pipeline Hazards and Their Solutions

In pipelined processors, multiple instructions execute simultaneously across different stages. While this improves performance, it also introduces several types of hazards that can disrupt normal instruction flow. These hazards and their solutions are outlined below:

1. Data Hazards

Data hazards occur when instructions depend on the results of previous instructions that have not yet completed.

Example: `add s8, s1, s2`

```
sub s3, s8, s4
```

Here, sub needs the result from add, which may not be ready.

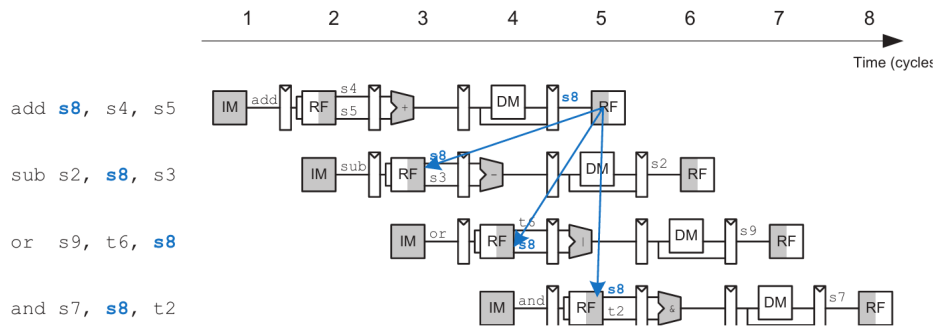

Figure 3.7: Illustration of hazards

Solutions:

- **Forwarding (Bypassing):** Sends results from later stages (e.g., MEM or WB) directly to the EX stage, avoiding the need to wait.
- **Stalling:** Introduces a bubble (nop) if data is not ready, especially in load-use scenarios.
- **Software Interlocks:** Manually insert `nop` instructions (less preferred in hardware design).

2. Control Hazards

Control hazards arise from branch instructions, where the next instruction to fetch depends on the outcome of the branch.

Solutions:

- **Stalling:** Delay instruction fetch until the branch decision is known (typically from EX stage).
- **Branch Prediction:** Predict the branch direction. If mispredicted, flush the incorrect instructions.

Hazard Handling Logic

- **Load-use detection:** `lwStall = ResultSrcE[0] && ((Rs1D == RdE) || (Rs2D == RdE))` `StallF = StallD = FlushE = lwStall`

- **Branch flushing:** `FlushD = PCSrcE`

  `FlushE = lwStall || PCSrcE`

3.5 Branch Prediction

In a pipelined processor, **control hazards** arise when the processor encounters a branch instruction but has not yet determined whether the branch should be taken or not. Since the outcome of the branch is only known in the Execute stage, subsequent instructions fetched in the meantime might be invalid if the branch is taken. To minimize performance loss due to stalling or flushing, modern processors use **branch prediction**.

Why Branch Prediction?

Without prediction, the pipeline must stall until the branch decision is resolved, leading to a significant performance penalty. By predicting whether a branch will be taken or not, the processor can speculatively fetch and execute instructions. If the prediction is correct, the pipeline proceeds without interruption. If incorrect, the mispredicted instructions are flushed, and execution resumes from the correct address. This method significantly improves overall performance when branches are frequent.

2-Bit Saturating Counter Predictor

We use a **2-bit branch predictor** which provides a good balance between accuracy and simplicity. This predictor uses a finite state machine with four states:

- **Strongly Taken (ST)**
- **Weakly Taken (WT)**
- **Weakly Not Taken (WNT)**
- **Strongly Not Taken (SNT)**

The prediction is determined by the current state:

- ST and WT predict the branch **will be taken**.

- SNT and WNT predict the branch **will not be taken**.

Transitions between states occur based on the actual outcome of the branch:

- If a branch is taken, the state moves toward ST.
- If a branch is not taken, the state moves toward SNT.

State Transition Diagram

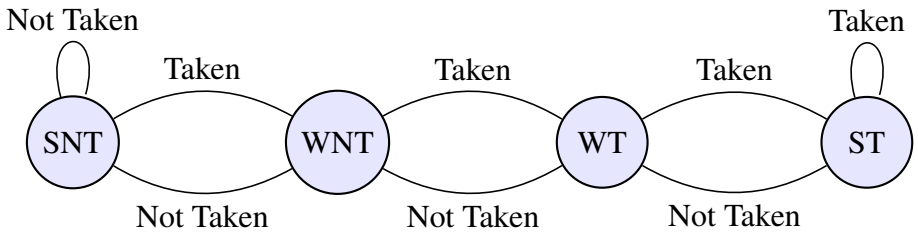

Figure 3.8: 2-bit Branch Predictor State Machine

Chapter 4

IMPLEMENTATION

4.1 Development Environment

The design and simulation of the pipelined RISC-V CPU were carried out using **Vivado 2023.1**, a comprehensive tool for digital design and simulation by Xilinx. The CPU was implemented entirely in Verilog and tested using Vivado's built-in simulator.

- **Software Used:** Vivado Design Suite 2023.1
- **Target:** Simulation-based design (no physical FPGA deployment)
- **Language:** Verilog HDL
- **Operating System:** Windows 11

4.2 Design Modules

- **Instruction Fetch (fetch_cycle.v)**: Retrieves instructions from memory.
- **Instruction Decode (decode_cycle.v)**: Decodes the fetched instruction to determine control signals.
- **Execution (execute_cycle.v)**: Performs arithmetic, logic, and branching operations.
- **Memory Access (memory_cycle.v)**: Reads and writes data from/to memory.

- **Write-back (writeback_cycle.v)**: Updates the register file with computed results.
- **Branch Predictor (branch_predictor.v)**: Implements a 2-bit predictor for branch prediction.
- **Hazard Unit**: Detects and resolves data and control hazards in the pipeline.
- **Top-level Module (riscv_top.v)**: Integrates all components into a functional CPU.

4.2.1 Hazard Unit

The hazard unit detects and resolves hazards using forwarding and stalls. The Verilog implementation is given below:

```
module hazard_unit (
    input rst,RegWriteM,RegWriteW,
    input [4:0]RdM,RdW,
    input [4:0] rs1_addr_E,rs2_addr_E,
    output [1:0] ForwardAE,ForwardBE,
    input [1:0] ResultSrcE,
    input [4:0] Rs1D,Rs2D,RdE,
    output StallF,StallD,FlushE,
    input Eval_branch,
    output FlushD
);

wire lwstall;

assign ForwardAE = (rst==1'b1)? 2'b00 :
                   ((RegWriteM ==1'b1) && (RdM != 5'b00000) && (RdM
                      == rs1_addr_E)) ? 2'b10 :
                   ((RegWriteW ==1'b1) && (RdW != 5'b00000) && (RdW
                      == rs1_addr_E)) ? 2'b01 : 2'b00;
assign ForwardBE = (rst==1'b1)? 2'b00 :
                   ((RegWriteM ==1'b1) && (RdM != 5'b00000) && (RdM
                      == rs2_addr_E)) ? 2'b10 :
```

```
                    ((RegWriteW ==1'b1) && (RdW != 5'b00000) && (RdW
                        == rs2_addr_E)) ? 2'b01 : 2'b00;

assign lwstall = (ResultSrcE ==2'b01) & ((Rs1D == RdE) | (Rs2D ==
    RdE));

assign StallF = lwstall;
assign StallD = lwstall;
assign FlushE = lwstall | Eval_branch;
assign FlushD = Eval_branch;

endmodule
```

Listing 4.1: Verilog Code Hazard unit

4.2.2 Branch Predictor

The branch predictor is based on a 2-bit saturating counter. It maintains four states: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken.

```
module branch_predictor (
    input clk,
    input reset,
    input Eval_branch,
    input PCSrcE,
    input [31:0] Act_Target,
    input [31:0] instr,
    input [31:0] PC,
    output  predict_branch,
    output  [31:0] Target_final,
    input StateUpdateEnable

);

    reg [1:0] state; // 2-bit saturating counter

    // State encoding
    localparam STRONG_NOT_TAKEN = 2'b00;
    localparam WEAK_NOT_TAKEN   = 2'b01;
```

```
    localparam WEAK_TAKEN       = 2'b10;
    localparam STRONG_TAKEN     = 2'b11;

wire [31:0] imm,Target;
reg predicted;
    // Handle final prediction logic (correct if mismatch occurs)
assign predict_branch = Eval_branch | (predicted & (7'b1100011 ==
   instr[6:0] || 7'b1101111 == instr[6:0]));

assign imm = (7'b1100011 == instr[6:0]) ? {{20{instr[31]}}, instr
   [7], instr[30:25], instr[11:8], 1'b0}: //b-type
             (7'b1101111 == instr[6:0]) ? {{12{instr[31]}}, instr
                [19:12], instr[20], instr[30:21], 1'b0}: //j-type
             32'bx;

assign Target = PC + imm; //target_address

mux2 targetmux(Target,Act_Target,Eval_branch,Target_final);

 always @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= STRONG_TAKEN;
        end else if(StateUpdateEnable) begin
            case (state)
                STRONG_NOT_TAKEN: state <= PCSrcE ? WEAK_NOT_TAKEN :
                    STRONG_NOT_TAKEN;
                WEAK_NOT_TAKEN:   state <= PCSrcE ? WEAK_TAKEN :
                   STRONG_NOT_TAKEN;
                WEAK_TAKEN:       state <= PCSrcE ? STRONG_TAKEN :
                   WEAK_NOT_TAKEN;
                STRONG_TAKEN:     state <= PCSrcE ? STRONG_TAKEN :
                   WEAK_TAKEN;
                default:          state <= state; // Default case to
                    handle unexpected states
            endcase
        end
    end

```

```
50      always @(*) begin
51          case (state)
52              STRONG_NOT_TAKEN, WEAK_NOT_TAKEN: predicted = 0;
53              WEAK_TAKEN, STRONG_TAKEN:         predicted = 1;
54          endcase
55      end
56
57  endmodule
```

Listing 4.2: Verilog Code for Branch Predictor

4.3 Final Schematic

The complete schematic of the single-cycle RISC-V CPU, as generated in Vivado 2023.1, is shown below. It includes all pipeline stages, the hazard detection unit, and the branch predictor.

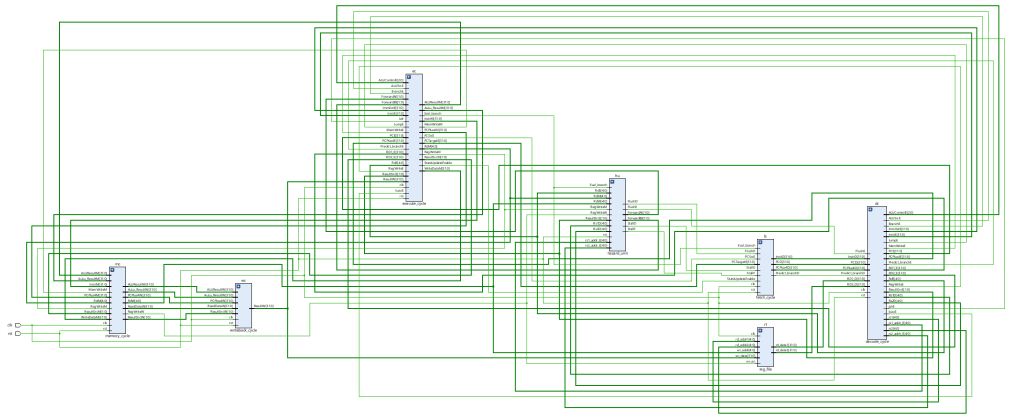

Figure 4.1: Final schematic of the single-cycle RISC-V CPU

Chapter 5

RESULTS AND SIMULATION

5.1 Simulation Setup

The design was implemented and simulated using **Vivado 2023.1**. A custom testbench was written in Verilog to validate the functionality of the pipeline stages, hazard unit, and branch prediction logic. Simulation was carried out using behavioral simulation, and waveforms were analyzed using Vivado's integrated waveform viewer.

5.2 Branch Prediction Impact

To demonstrate the effectiveness of the 2-bit branch predictor, we used a sample loop in RISC-V assembly as test input.

5.2.1 Sample Assembly Program

The following assembly code snippet contains a conditional branch. The performance of this code is analyzed both with and without branch prediction.

```
0:      00300113        addi x2 x0 3
4:      00300193        addi x3 x0 3
8:      00310663        beq x2 x3 12
c:      01400213        addi x4 x0 20
10:      00c00293        addi x5 x0 12
14:      00000313        addi x6 x0 0
18:      00200393        addi x7 x0 2
1c:      00507393        andi x7 x0 5
20:      00020313        addi x6 x4 0
24:      00028413        addi x8 x5 0
28:      00000213        addi x4 x0 0
2c:      00000213        addi x4 x0 0
30:      00000213        addi x4 x0 0
34:      00000213        addi x4 x0 0
38:      00000213        addi x4 x0 0
3c:      00000213        addi x4 x0 0
40:      00310663        beq x2 x3 12
44:      00000113        addi x2 x0 0
48:      00000113        addi x2 x0 0
4c:      00000113        addi x2 x0 0
50:      00000113        addi x2 x0 0
54:      00000113        addi x2 x0 0
58:      00000113        addi x2 x0 0
5c:      00000113        addi x2 x0 0
60:      00400113        addi x2 x0 4
```

Figure 5.1: Sample input: RISC-V assembly loop used for testing branch prediction

5.2.2 Without Branch Prediction

When the CPU is run without a branch predictor, every branch causes a pipeline flush unless perfectly predicted. The waveform below illustrates how the control hazard leads to frequent stalls and wasted cycles.

Observations:

- Multiple stalls occur after each branch.
- Instructions fetched speculatively are flushed due to mispredictions.
- Execution time is increased due to frequent pipeline flushes.
- **Final result is obtained at `325 ns`, as shown in the waveform.**

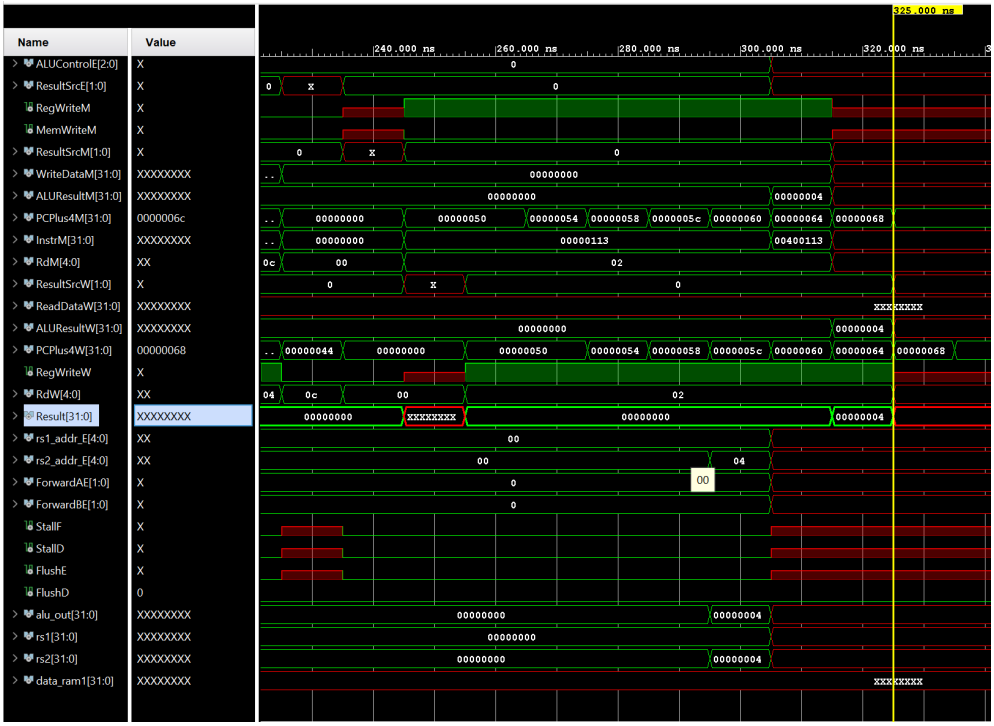

Figure 5.2: Waveform of CPU execution without branch predictor

5.2.3 With Branch Prediction

When the 2-bit branch predictor is enabled, the CPU is able to speculate on branch outcomes and reduce pipeline flushes significantly. The following waveform demonstrates more efficient execution with fewer stalls.

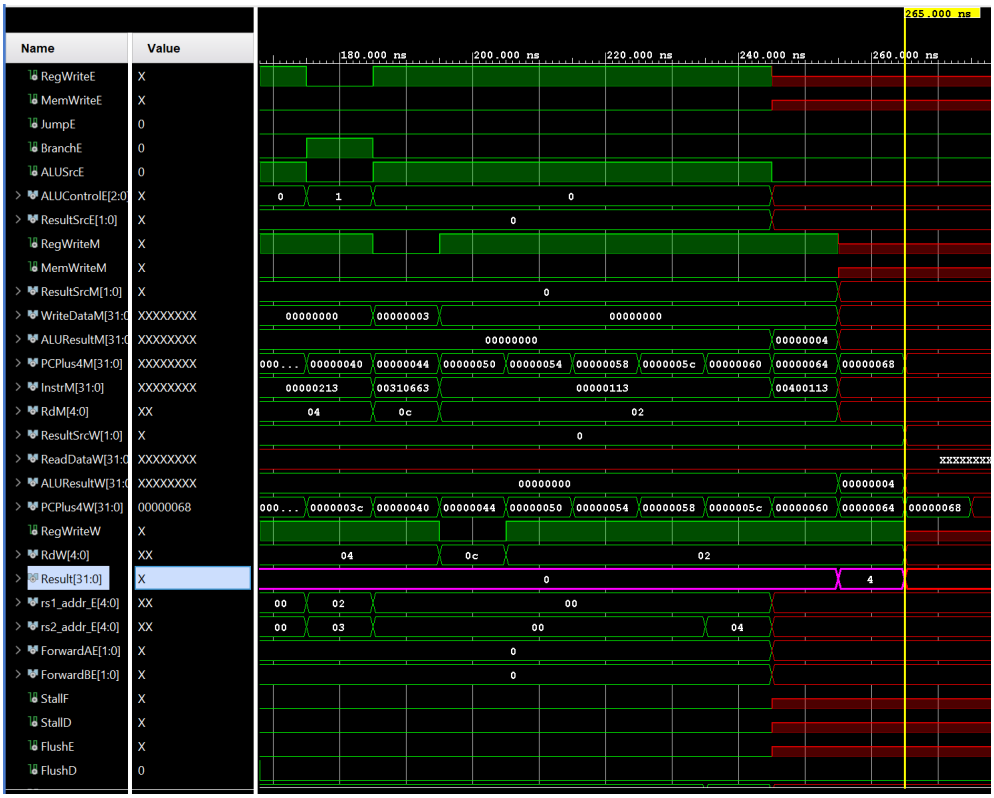

Figure 5.3: Waveform of CPU execution with 2-bit branch predictor

Observations:

- Branch instructions are predicted accurately after initial learning, preventing pipeline flushes.
- As a result, fewer control hazards occur, and performance improves significantly.
- **Final result is obtained at `265 ns`, compared to `325 ns` without prediction.**
- This improvement is because each correct prediction saves 3 cycles.
- With 2 branch instructions in the code, and a clock period of `10 ns`, the total time saved is:

$$3 \times 10\,\text{ns (first branch)} + 3 \times 10\,\text{ns (second branch)} = \textbf{60 ns}$$

- Therefore, the predicted execution time becomes:

$$325\,\text{ns} - 60\,\text{ns} = \textbf{265 ns}$$

5.2.4 Comparison

Performance Comparison: With vs Without Branch Prediction

Parameter	Without Predictor	With 2-bit Predictor
Clock Period	10 ns	10 ns
Execution Time	325 ns	**265 ns**
Pipeline Flushes	Multiple (on every branch)	Very Few (after training)
Branch Accuracy	Not Applicable	High (after initial learning)
Cycles Saved	0	**6 cycles (60 ns)**
Performance	Baseline	**Improved by 18.5%**

Chapter 6

Conclusion

In this project, we successfully designed and simulated a single-cycle RISC-V CPU in Vivado 2023.1, incorporating a 2-bit branch predictor to enhance performance. The processor was modeled using Verilog HDL and tested using realistic instruction sequences. A dedicated hazard detection unit was also implemented to manage pipeline hazards and ensure correct program execution.

The key objective was to reduce performance loss due to control hazards introduced by branch instructions. We achieved this by integrating a finite-state 2-bit saturating branch predictor. The results clearly demonstrate the performance gain: the processor without prediction completed execution in 325 ns, whereas the one with branch prediction completed in just 265 ns. This improvement reflects a reduction of six clock cycles, achieved through correct predictions that avoided unnecessary pipeline flushes.

Overall, the inclusion of branch prediction significantly boosts the efficiency of the pipeline, especially in branch-heavy workloads. The design can be further extended in future by supporting multi-cycle instructions, incorporating a forwarding unit.

References

[1] OpenAI, *ChatGPT*, Available at: https://openai.com/chatgpt, Accessed: April 2025.

[2] Google, *Gemini AI*, Available at: https://gemini.google.com, Accessed: April 2025.

[3] RISC-V: From Transistors to AI, *YouTube Channel*, Available at: https://www.youtube.com/@riscvtransistors2ai, Accessed: April 2025.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 6th Edition, 2017.

[5] D. A. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon LLC, 2017.

[6] Xilinx, *Vivado Design Suite User Guide*, Available at: https://www.xilinx.com, Accessed: April 2025.

[7] T. Yeh and Y. N. Patt, *Alternative Implementations of Two-Level Adaptive Branch Prediction*, In Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA), 1992.

[8] RISC-V Foundation, *Official RISC-V Specifications*, Available at: https://riscv.org/specifications/, Accessed: April 2025.

[9] P. Chu, *FPGA Prototyping by Verilog Examples*, Wiley, 2008.

[10] M. Morris Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, Pearson, 5th Edition, 2012.