FACULTY OF ENGINEERING

MULTIMEDIA UNIVERSITY

June 2024

The project report is prepared for

Faculty of Engineering

Multimedia University

in partial fulfilment for

Bachelor of Engineering (Hons) Electronics

majoring in Computer

Design a 32-bit MIPS ALU based on RISC

by

Saeb K. H. Naser

1191302764

Session 2023/2024

© 2024 Universiti Telekom Sdn. Bhd. ALL RIGHTS RESERVED.

Copyright of this report belongs to Universiti Telekom Sdn. Bhd. as qualified by Regulation 7.2 (c) of the Multimedia University Intellectual Property and Commercialisation Policy. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Universiti Telekom Sdn. Bhd. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

DECLARATION

I hereby declare that this work has been done by myself and no portion of the work contained in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

I also declare that pursuant to the provisions of the Copyright Act 1987, I have not engaged in any unauthorised act of copying or reproducing or attempt to copy / reproduce or cause to copy / reproduce or permit the copying / reproducing or the sharing and / or downloading of any copyrighted material or an attempt to do so whether by use of the University’s facilities or outside networks / facilities whether in hard copy or soft copy format, of any material protected under the provisions of sections 3 and 7 of the Act whether for payment or otherwise save as specifically provided for therein. This shall include but not be limited to any lecture notes, course packs, thesis, text books, exam questions, any works of authorship fixed in any tangible medium of expression whether provided by the University or otherwise.

I hereby further declare that in the event of any infringement of the provisions of the Act whether knowingly or unknowingly the University shall not be liable for the same in any manner whatsoever and undertakes to indemnify and keep indemnified the University against all such claims and actions.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name: Saeb K. H. Naser

Student ID: 1191302764

Date: 18th June 2024

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my supervisor, Dr. Chinnaiyan Senthilpari, for his invaluable guidance and support throughout this project. His expertise and insights have been instrumental in shaping the direction and execution of my work. This project would not have been possible without his continuous mentorship and constructive feedback. I would also like to thank my moderator, Dr. Lee Chu Liang, for his valuable insights and feedback, which have greatly contributed to the refinement and improvement of this project.

I also extend my gratitude to the faculty and staff of the Faculty of Engineering for providing a solid foundation in engineering principles. Their dedication and encouragement have been pivotal in my academic journey.

A special thank you to my mom and dad for saying I can and my brothers and sister for their unwavering support and encouragement, which have been a constant source of strength. Their belief in me has motivated me and has been crucial in the successful completion of this project.

Lastly, I am grateful to all those who have indirectly contributed to this project. Your support, whether through technical assistance or moral encouragement, has played an essential role in its successful completion. Thank you all for your contributions.

ABSTRACT

In the rapidly advancing field of computer engineering, processor design is critical for optimizing the performance and efficiency of computing systems. This project focuses on designing and implementing a 32-bit Arithmetic Logic Unit (ALU) based on the principles of Reduced Instruction Set Computing (RISC) within the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture.

The ALU, responsible for executing arithmetic and logic operations, is integral to contemporary computing. The project involves the detailed design and implementation of the ALU, celebrated for its simplicity and efficiency. To validate the ALU's functionality and performance, the remaining components of the MIPS architecture were designed, creating a basic MIPS architecture to ensure seamless integration into the CPU datapath. Additionally, a MIPS assembler was implemented in C++ to define and execute custom test programs, facilitating thorough testing and validation.

Significant findings include the successful implementation and optimization of the ALU, ensuring efficient execution of arithmetic and logic operations. The ALU's design adhered to RISC principles, emphasizing a reduced set of instructions executed in a single clock cycle, which established it as a standard in processor design. The project demonstrated the ALU's efficiency and performance through rigorous VHDL coding and simulation, confirming its critical role in processor optimization.

In conclusion, the project successfully developed a 32-bit MIPS ALU embodying the simplicity and efficiency of the RISC instruction set architecture. This comprehensive approach provided valuable insights into processor design and optimization, contributing to the ongoing discourse in computer engineering.

TABLE OF CONTENTS

[DECLARATION iii](#_Toc441237503)

[ACKNOWLEDGEMENT iv](#_Toc441237504)

[ABSTRACT v](#_Toc441237505)

[TABLE OF CONTENTS vi](#_Toc441237506)

[LIST OF FIGURES viii](#_Toc441237507)

[LIST OF TABLES xi](#_Toc441237508)

[LIST OF ABBREVIATIONS xii](#_Toc441237509)

[CHAPTER 1: INTRODUCTION 1](#_Toc441237510)

[1.1 Overview 1](#_Toc441237511)

[1.2 Problem Statements 13](#_Toc441237512)

[1.3 Project Scope 13](#_Toc441237513)

[1.4 Report Outline 14](#_Toc441237514)

[CHAPTER 2: LITERATURE REVIEW 15](#_Toc441237515)

[2.1 Introduction 15](#_Toc441237516)

[2.2 Literature Review 15](#_Toc441237517)

[2.2.1 Synthesis and Simulation of a 32-bit MIPS RISC Processor using VHDL 15](#_Toc441237517)

[2.2.2 Implementation of a 32-bit MIPS Based RISC Processor using Cadence 17](#_Toc441237517)

[2.2.3 Design and Implementation of 32-bit Functional Unit for RISC Architecture Applications 18](#_Toc441237517)

[2.3 Comparison of Results 2](#_Toc441237518)0

[2.4 Identifying Research Gaps and Future Steps 22](#_Toc441237519)

[CHAPTER 3: DETAILS OF THE DESIGN 23](#_Toc441237520)

[3.1 Literature-Based Circuit Design 23](#_Toc441237521)

[3.2 Proposed Circuit Modifications 28](#_Toc441237522)

[3.3 FPGA-Based Circuit Design 35](#_Toc441237523)

[CHAPTER 4: DATA PRESENTATION AND DISCUSSION OF FINDINGS 73](#_Toc441237524)

[4.1 Data Presentation 73](#_Toc441237525)

4.1.1 RTL Flow Map 73

4.1.2 Simulation Waveforms 77

4.1.3 Timing Diagram 87

[4.2 Discussion of Findings 90](#_Toc441237526)

[CHAPTER 5: CONCLUSIONS 92](#_Toc441237527)

[5.1 Summary and Conclusions 92](#_Toc441237528)

[5.2 Areas of Future Research 93](#_Toc441237529)

[REFERENCES 94](#_Toc441237530)

LIST OF FIGURES

[Figure 1.1: MIPS pipelined processor datapath 4](#_Toc441236458)

[Figure 1.2: R-type instruction register format 7](#_Toc441236458)

[Figure 1.3: An example of an R-type instruction register format](#_Toc441236458) 7

[Figure 1.4: I-type instruction register format (ALU immediate) 7](#_Toc441236458)

[Figure 1.5: I-type instruction register format (load or store) 8](#_Toc441236458)

[Figure 1.6: I-type instruction register format (branch) 8](#_Toc441236458)

[Figure 1.7: J-type instruction register format (jump) 8](#_Toc441236458)

[Figure 1.8: Data path for R-type instructions 9](#_Toc441236458)

[Figure 1.9: Data path for I-type instructions 10](#_Toc441236458)

[Figure 1.10: Data path for J-type instructions 11](#_Toc441236458)

[Figure 3.1: MIPS pipelined processor datapath by Topiwala and Saraswathi 24](#_Toc441236458)

[Figure 3.2: Pipelined architecture with improved datapath by Topiwala and Saraswathi 24](#_Toc441236458)

[Figure 3.3: RTL view of proposed MIPS RISC processor by Ritpurkar, Thakare, and Korde 25](#_Toc441236458)

[Figure 3.4: 32-Bit Single-Cycle MIPS Block Diagram 26](#_Toc441236458)

[Figure 3.4: 32-Bit Single-Cycle MIPS Block Diagram 30](#_Toc441236458)

[Figure 3.5: Instruction fetch stage of R-type instruction 31](#_Toc441236458)

[Figure 3.6: Instruction decode stage of R-type instruction 32](#_Toc441236458)

[Figure 3.7: Execute stage of R-type instruction 33](#_Toc441236458)

[Figure 3.8: Write back stage of R-type instruction 34](#_Toc441236458)

[Figure 3.9: Program Counter VHDL code 36](#_Toc441236458)

[Figure 3.10: Adder VHDL code 37](#_Toc441236458)

[Figure 3.11: Instruction Memory VHDL code - part 1 38](#_Toc441236458)

[Figure 3.12: Instruction Memory VHDL code - part 2 39](#_Toc441236458)

[Figure 3.13: Instruction Memory VHDL code - part 3 40](#_Toc441236458)

[Figure 3.14: MIPS Assembler C++ header file code 41](#_Toc441236458)

[Figure 3.15: MIPS Assembler C++ source file code - part 1 42](#_Toc441236458)

[Figure 3.16: MIPS Assembler C++ source file code - part 2 43](#_Toc441236458)

[Figure 3.17: MIPS Assembler C++ source file code - part 3 44](#_Toc441236458)

[Figure 3.18: MIPS Assembler C++ source file code - part 4 45](#_Toc441236458)

[Figure 3.19: MIPS Assembler C++ source file code - part 5 46](#_Toc441236458)

[Figure 3.20: MIPS Assembler C++ source file code - part 6 47](#_Toc441236458)

[Figure 3.21: MIPS Assembler C++ source file code - part 7 48](#_Toc441236458)

[Figure 3.22: MIPS Assembler C++ source file code - part 8 49](#_Toc441236458)

[Figure 3.23: MIPS Assembler entry point (main) code 50](#_Toc441236458)

[Figure 3.24: MIPS Assembler in action 51](#_Toc441236458)

[Figure 3.25: Multiplexer (MUX) VHDL code 52](#_Toc441236458)

[Figure 3.26: Registers VHDL code 53](#_Toc441236458)

[Figure 3.27: Sign Extend VHDL code 54](#_Toc441236458)

[Figure 3.28: Shift Left 2 VHDL code 55](#_Toc441236458)

[Figure 3.29: Data Memory VHDL code 56](#_Toc441236458)

[Figure 3.30: ALU VHDL code - part 1 57](#_Toc441236458)

[Figure 3.31: ALU VHDL code - part 2 58](#_Toc441236458)

[Figure 3.32: Control Unit VHDL code - part 1 59](#_Toc441236458)

[Figure 3.33: Control Unit VHDL code - part 2 60](#_Toc441236458)

[Figure 3.34: Data Path VHDL code - part 1 61](#_Toc441236458)

[Figure 3.35: Data Path VHDL code - part 2 62](#_Toc441236458)

[Figure 3.36: Data Path VHDL code - part 3 63](#_Toc441236458)

[Figure 3.37: Data Path VHDL code - part 4 63](#_Toc441236458)

[Figure 3.38: Top-Level VHDL code - part 1 64](#_Toc441236458)

[Figure 3.39: Top-Level VHDL code - part 2 65](#_Toc441236458)

[Figure 3.40: Top-Level VHDL code - part 3 66](#_Toc441236458)

[Figure 3.41: ALU Testbench VHDL code - part 1 67](#_Toc441236458)

[Figure 3.42: ALU Testbench VHDL code - part 2 68](#_Toc441236458)

[Figure 3.43: ALU Testbench VHDL code - part 3 69](#_Toc441236458)

[Figure 3.44: Top-Level Testbench VHDL code 70](#_Toc441236458)

[Figure 3.45: Top-Level second Testbench VHDL code 72](#_Toc441236458)

[Figure 4.1: RTL Flow Map of the 32-bit Single-Cycle MIPS processor 74](#_Toc441236458)

[Figure 4.2: RTL Flow Map of the control unit 75](#_Toc441236458)

[Figure 4.3: RTL Flow Map of the data path 76](#_Toc441236458)

[Figure 4.4: RTL Flow Map of the instruction memory 77](#_Toc441236458)

[Figure 4.5: Simulation waveform for ALU operations 78](#_Toc441236458)

[Figure 4.6: Simulation waveform for AND operation 79](#_Toc441236458)

[Figure 4.7: Simulation waveform for XOR operation 80](#_Toc441236458)

[Figure 4.8: Simulation waveform for Subtraction operation 81](#_Toc441236458)

[Figure 4.9: Simulation waveform for Top-Level second testbench 82](#_Toc441236458)

[Figure 4.10: Screenshot of the instruction memory showing the first instruction 83](#_Toc441236458)

[Figure 4.11: Simulation waveform for Addi instruction 84](#_Toc441236458)

[Figure 4.12: Screenshot of the control unit showing the 11-bit data signal 85](#_Toc441236458)

[Figure 4.13: Simulation waveform for SW instruction 86](#_Toc441236458)

[Figure 4.14: Simulation waveform for Top-Level first testbench 87](#_Toc441236458)

[Figure 4.15: Nonpipelined execution of three Load Word instructions 88](#_Toc441236458)

[Figure 4.16: Pipelined execution of three Load Word instructions 89](#_Toc441236458)

LIST OF TABLES

[Table 2.1: Literature review table of findings 20](#_Toc441236213)

[Table 4.1: Theoretical MIPS Pipeline Timing Analysis 88](#_Toc441236213)

LIST OF ABBREVIATIONS

ALU Arithmetic Logic Unit

MIPS Microprocessor without Interlocked Pipelined Stages

RISC Reduced Instruction Set Computing

CISC Complex Instruction Set Computing

ISA Instruction Set Architecture

CPU Central Processing Unit

IF Instruction Fetch Stage

ID Instruction Decode Stage

EX Execute Stage

MEM Memory Access Stage

WB Write Back Stage

PC Program Counter

RTL Register Transfer Level

# INTRODUCTION

## Overview

In the ever-evolving landscape of computer engineering, the design and architecture of processors play a pivotal role in shaping the performance and efficiency of modern computing systems. Central to this domain is the Arithmetic Logic Unit (ALU), a critical component that executes arithmetic and logic operations, forming the backbone of processor functionality. This project, titled "Design a 32-bit MIPS ALU based on RISC", aims to contribute to this dynamic field by focusing on the meticulous design and implementation of a 32-bit ALU within the framework of the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture, adhering to the principles of Reduced Instruction Set Computing (RISC).

The ALU's significance in computing cannot be overstated, as it is responsible for performing essential operations that enable the execution of instructions within a processor. The MIPS architecture, renowned for its simplicity and efficiency, has been a benchmark in processor design due to its commitment to RISC principles. RISC architecture emphasizes a streamlined set of instructions that can be executed in a single clock cycle, enhancing processing speed and efficiency. This approach contrasts with Complex Instruction Set Computing (CISC), which utilizes a more extensive set of instructions, often resulting in more complex and slower execution.

Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) represent two fundamentally different approaches to processor design, with significant differences in their Instruction Set Architectures (ISAs). MIPS, a classic example of RISC architecture, contrasts sharply with CISC architectures like x86, influencing performance, complexity, and efficiency.

RISC architectures use a small, highly optimized set of instructions, each designed to execute in a single clock cycle. This simplicity allows for faster instruction execution and easier pipelining. MIPS instructions are uniform in size and format, making them straightforward to decode and execute. In contrast, CISC architectures like x86 have a larger and more complex instruction set, with many instructions performing multiple operations. This complexity can lead to more powerful instructions but often requires multiple clock cycles to execute. RISC processors are designed with pipelining in mind, enabling them to execute multiple instructions simultaneously by breaking down the execution process into several stages. The MIPS architecture, with its five-stage pipeline (fetch, decode, execute, memory access, writeback), exemplifies this approach, enhancing throughput and efficiency. CISC processors, due to their more complex instructions, face greater challenges in achieving efficient pipelining. Instructions that vary in length and complexity make it difficult to maintain a steady flow through the pipeline.

CISC architectures often use microcode to implement complex instructions. Microcode is a layer of low-level instructions that translate high-level CISC instructions into sequences of simpler operations. While this approach allows for more versatile and powerful instructions, it adds another layer of complexity and can slow down instruction execution. RISC architectures, including MIPS, avoid microcode altogether. Each instruction is executed directly by the hardware, simplifying the control unit and speeding up execution.

RISC architectures make extensive use of registers, relying on a large number of general-purpose registers to hold operands and intermediate results. MIPS, for example, provides 32 general-purpose registers, minimizing the need for frequent memory accesses. This contrasts with CISC architectures, which often have fewer registers and rely more heavily on memory operations. The heavy use of registers in RISC designs contributes to faster and more efficient execution of instructions. In RISC architectures, only load and store instructions access memory, adhering to a load/store model. This means that all data processing operations are performed on register values. This separation simplifies the instruction set and enhances execution speed. In CISC architectures, memory access can be part of any instruction, leading to more flexible but also more complex instruction execution.

The design philosophy of RISC emphasizes simplicity and efficiency, focusing on executing a small set of instructions extremely well. This results in a simpler control unit and faster clock speeds. In contrast, the design philosophy of CISC aims to reduce the number of instructions per program by using more complex instructions, potentially reducing the program's size but increasing the complexity of the processor design. By comparing RISC and CISC, it becomes evident why MIPS, as a RISC architecture, is valued for its simplicity, efficiency, and educational utility. The streamlined design of MIPS facilitates understanding of fundamental processor operations while offering robust performance suitable for a variety of applications.

One of the key features of MIPS is its "without Interlocked Pipeline Stages" design, which emphasizes a crucial aspect of its architecture. In a pipelined processor, instructions progress through different stages (e.g., fetch, decode, execute) in a pipeline fashion, with each stage handling a specific part of instruction execution. Interlocked pipeline stages imply that these stages are closely connected or dependent on each other, meaning the progress of one stage may affect the progress of others. MIPS avoids this interdependence, allowing each stage of the pipeline to operate independently and efficiently.

The MIPS architecture is renowned for its streamlined design, which includes a five-stage execution pipeline, as shown in Figure 1.1, consisting of fetch, decode, execute, memory access, and write result stages. This pipeline structure allows instructions to be processed efficiently, with each stage dedicated to a specific aspect of instruction execution. This design choice ensures that instructions move through the processor smoothly, enhancing performance.

A computer screen shot of different colored rectangular shapes

Description automatically generated

Figure 1.1: MIPS pipelined processor datapath

This figure before, illustrates the MIPS pipeline implementation. It clearly depicts the flow of instructions through the five stages of the pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The figure highlights the concurrent processing of multiple instructions, showcasing how each stage operates in parallel with others to maximize efficiency and throughput. By visualizing the pipeline, it becomes evident how MIPS architecture achieves its performance advantages through instruction overlap and stage-specific processing. The explanation of the MIPS pipeline implementation is based on concepts discussed in "Computer Architecture" by Dr. Ranjani Parthasarathi [2].

In the Instruction Fetch (IF) stage, the processor retrieves the next instruction to be executed from memory. The Program Counter (PC) holds the address of the current instruction, which is used to fetch the instruction from the Instruction Memory. Once fetched, the instruction is stored in the Instruction Register (IR), and the PC is incremented to point to the next instruction. This stage sets the foundation for the subsequent stages by ensuring that the correct instruction is available for decoding.

The Instruction Decode (ID) stage involves interpreting the fetched instruction to determine the required operation and operands. During this stage, the instruction is divided into its constituent fields, such as the opcode, source and destination registers, and immediate values. The control signals necessary for executing the instruction are generated based on the opcode. Additionally, the source registers specified in the instruction are read from the Register File, and the operands are prepared for the execution stage. This stage is crucial for translating the instruction into specific control signals and data needed for execution.

In the Execute (EX) stage, the actual computation specified by the instruction takes place. For arithmetic and logical operations, the operands read during the ID stage are fed into the Arithmetic Logic Unit (ALU), which performs the required operation (such as addition, subtraction, or bitwise operations). For branch instructions, the ALU calculates the target address, and for shift instructions, the data is shifted accordingly. This stage is the core of the instruction execution process, where the main computational work is carried out.

The Memory Access (MEM) stage is where the instruction interacts with memory if necessary. For load instructions, data is fetched from the Data Memory and placed into a register, while for store instructions, data from a register is written into the Data Memory. This stage ensures that data is correctly transferred between the processor and memory, facilitating the execution of memory-related operations. In cases where the instruction does not require memory access, this stage is bypassed and finally, the Write Back (WB) stage is responsible for writing the results of the computation back into the Register File. For most instructions, the result computed in the EX stage, or the data fetched in the MEM stage is written into the destination register specified by the instruction. This stage completes the instruction execution process by updating the processor's state with the new data, making it available for subsequent instructions.

The MIPS pipeline achieves efficient instruction execution by dividing the process into these five stages, allowing multiple instructions to be processed simultaneously. Each stage is carefully designed to perform a specific function, ensuring that the instruction flows smoothly through the pipeline. This pipelining approach significantly improves the throughput of the processor, as new instructions can be fetched and processed while previous instructions are still in the pipeline, leading to better utilization of the processor's resources and higher overall performance.

A fundamental aspect of MIPS is its regular instruction set, where all instructions are 32 bits long. This uniformity simplifies the decoding process, making it easier for the processor to execute instructions quickly and accurately. Additionally, MIPS instructions typically follow a three-operand format, involving source1, source2, and destination operands. This design enables the processor to perform a wide range of arithmetic and logical operations efficiently.

MIPS provides 32 general-purpose registers, each 32 bits in size, for storing data and operands. These registers play a crucial role in data manipulation and processing within the processor. Furthermore, MIPS avoids complex instructions, focusing on simplicity and efficiency. Instead, it utilizes a load/store architecture, where only load and store instructions access memory directly. This architecture simplifies memory access and management, contributing to the overall efficiency of the processor. MIPS instructions are organized into three main types: R-type, I-type, and J-type. R-type instructions are used for arithmetic and logical operations, involving three registers. I-type instructions handle immediate values in addition to registers, while J-type instructions are used for jump and branch operations. In the R-type instruction format, as shown in Figure 1.2, there are a total of six fields. The Opcode field is 6 bits long and is used to select the type of instruction format. The Source register (Rs), Target register (Rt), and Destination register (Rd) fields are each 5 bits long and are used for data storage and manipulation within the processor. The Shamt field, which stands for Shift amount, is 5 bits long and is used for shifting data within the registers. Finally, the Function field is 6 bits long and is used to select which type of function to perform, such as addition, subtraction, or logical operations.

A screenshot of a computer

Description automatically generated

Figure 1.2: R-type instruction register format

Figure 1.3 shows an example of an R-type instruction, specifically Add $17, $2, $5. This instruction adds the contents of registers $2 and $5, and stores the result in register $17. This type of instruction involves three registers, adhering to the format where the opcode specifies the operation (addition in this case), and the registers specify the operands and the destination.

A diagram of a graph

Description automatically generated with medium confidence

Figure 1.3: An example of an R-type instruction register format

In the I-type instruction formats, as shown in Figures 1.4, 1.5, and 1.6, there are a total of four fields. Similar to the R-type format, the Opcode field is 6 bits long and is used to select the type of instruction format. The Source register (Rs) and Target register (Rt) fields are each 5 bits long and are used for data storage and manipulation. Additionally, there is another field called the Address/Immediate Value field, which is 16 bits long. This field is used for storing immediate data values that are used in operations such as data transfer and immediate arithmetic/logical operations.

A white rectangular box with black text

Description automatically generated

Figure 1.4: I-type instruction register format (ALU immediate)

A screenshot of a computer

Description automatically generated

Figure 1.5: I-type instruction register format (load or store)

A screenshot of a computer

Description automatically generated

Figure 1.6: I-type instruction register format (branch)

An example of an I-type instruction is Addi $17, $2, 1. In this case, the instruction adds the immediate value 1 to the contents of register $2 and stores the result in register $17. This demonstrates how I-type instructions handle immediate values, with the opcode indicating the operation, and the immediate field providing a constant value.

In the J-type instruction format, as shown in Figure 1.7, there are only two fields. The Opcode field is 6 bits long and is used to select the type of instruction format. The Target address field is 26 bits long and is used to specify the address to which the processor should branch. This format is primarily used for jump and branch operations within the processor, allowing for efficient control flow management.



Figure 1.7: J-type instruction register format (jump)

An example of a J-type instruction is j 64. This instruction causes the program to jump to address 64. J-type instructions are primarily used for jump and branch operations, where the opcode indicates the jump, and the target address specifies where to jump within the memory.

Each instruction type has distinct characteristics and interacts with the datapath components differently, reflecting the architecture's streamlined and modular approach.

For R-type instructions, which are used for arithmetic and logical operations, the datapath involves several key components, as shown in Figure 1.8. During the instruction fetch (IF) stage, the instruction is retrieved from memory and stored in the Instruction Register (IR). In the instruction decode (ID) stage, the control unit decodes the instruction to generate the necessary control signals. The source registers (Rs and Rt) are read from the Register File, and the operands are prepared for execution. In the execute (EX) stage, the operands are fed into the Arithmetic Logic Unit (ALU), where the specified operation (e.g., addition, subtraction) is performed. The result of the ALU operation is then passed to the memory access (MEM) stage, although R-type instructions typically do not require memory access. Finally, in the write-back (WB) stage, the result is written back to the destination register (Rd) in the Register File, completing the instruction execution process.

A diagram of a computer code

Description automatically generated

Figure 1.8: Data path for R-type instructions

I-type instructions, which handle immediate values and data transfer operations, have a slightly different datapath, which is shown in Figure 1.9. After the instruction is fetched and stored in the IR during the IF stage, it is decoded in the ID stage to generate control signals. The source register (Rs) is read from the Register File, and the immediate value is extracted from the instruction. In the EX stage, the ALU performs the required operation, such as adding the immediate value to the contents of the source register. For load and store instructions, the result of the ALU operation is an effective address used in the MEM stage. For load instructions, data is fetched from memory and stored in the target register (Rt). For store instructions, data from the target register is written to memory. The WB stage is only used for load instructions, where the fetched data is written back to the target register.

A diagram of a program

Description automatically generated

Figure 1.9: Data path for I-type instructions

J-type instructions, used for jump operations, involve a more streamlined datapath, as shown in Figure 1.10. After fetching the instruction in the IF stage, the instruction is decoded in the ID stage to identify it as a jump instruction. The target address is extracted from the instruction and used to update the Program Counter (PC) in the EX stage. This immediate change in the PC causes the processor to fetch the next instruction from the specified address, effectively implementing the jump. The MEM and WB stages are typically bypassed for J-type instructions, as they do not involve data processing or memory access.

A diagram of a shift

Description automatically generated

Figure 1.10: Data path for J-type instructions

In the figure before, the functionality of the J-type instruction is illustrated. The last four bits of PC+4 are appended to the 26-bit instruction value (shifted left by 2) obtained from the Instruction Memory to form a 32-bit jump address. For instance, in the instruction "j trgt", where "j" denotes the jump instruction and "trgt" is the target address, the processor skips intermediate instructions and jumps directly to the specified target. This mechanism allows for efficient and straightforward control flow changes, which are essential for implementing functions and conditional execution.

The choice of the MIPS architecture for this project is driven by its historical impact and continued relevance in processor development. The MIPS architecture's elegance and simplicity have inspired numerous subsequent architectures, making it a foundational element in the study and practice of computer engineering. By focusing on the 32-bit variant of the MIPS ALU, this project aims to explore the design intricacies and challenges associated with implementing a high-performance ALU within a well-established architectural framework.

A key objective is to ensure that the ALU design adheres strictly to RISC principles by utilizing a simplified set of instructions that can be executed in a single clock cycle. This approach is intended to enhance processing speed and reduce complexity, aligning with the goals of achieving high-performance computing. The designed ALU will cover nine fundamental operations, including addition, subtraction, logical AND, OR, NOR, NAND, XOR, left shift (SLL), and right shift (SRL). These operations are essential for the MIPS instruction set [1] and provide a comprehensive functionality required for efficient processing within the architecture.

To validate the designed ALU's functionality and performance, the project also involves the design of the remaining MIPS components, ensuring a seamless integration of the ALU within the CPU datapath. Additionally, a MIPS assembler is implemented in C++ to enable users to define and execute custom test programs, facilitating thorough testing and validation of the ALU and other MIPS components. This holistic approach provides a comprehensive understanding of how the ALU fits into the broader MIPS architecture, ensuring that each component aligns perfectly for optimal system performance.

The designed MIPS processor will be capable of executing a range of instructions from the MIPS instruction set, including register and immediate arithmetic-logical operations like ADD, SUB, AND, OR, NOR, XOR, ADDI, ANDI, and ORI. It also handles shift operations such as SLL and SRL, set operations like SLT and SLTI, branch and jump commands including BEQ, BNE, and J, as well as memory operations with LW and SW. In a single-cycle CPU, each instruction must finish in one clock cycle, so the cycle's length is based on the time it takes for the slowest instruction. This is usually determined by the longest path an instruction takes, often the load word (LW) instruction.

As technology advances and the demand for more efficient computing solutions grows, the need for processors capable of handling complex operations with minimal latency becomes increasingly critical. This project addresses this need by concentrating on the design and optimization of the ALU, a core component that significantly impacts the overall performance of a processor. The project's primary objective is to develop a 32-bit MIPS ALU that not only adheres to the principles of RISC but also enhances the capabilities and efficiency of the processor within the MIPS architecture.

Furthermore, the project aims to optimize the ALU design for efficiency in terms of speed, power consumption, and resource utilization, striving to achieve a balance between high performance and minimal resource overhead. By doing so, the research contributes to the ongoing discourse on processor design, offering a comprehensive analysis and documentation of the design process, challenges encountered, and solutions implemented. This contribution is intended to enhance the understanding and knowledge base in the field of processor design, providing valuable insights for future research and development.

## Problem Statements

This project addresses several key problems in the realm of processor design. First, there is a need for a simplified yet efficient processor architecture that can execute instructions swiftly and accurately. Existing architectures often struggle with balancing complexity and performance, particularly in executing arithmetic and logic operations. The MIPS architecture, despite its advantages, presents challenges in integrating various components to achieve seamless performance. Specific problems addressed by this project include the design and implementation of a high-performance 32-bit ALU, the integration of this ALU within the MIPS processor, and the validation of its performance through comprehensive testing. Moreover, developing a MIPS assembler in C++ is necessary to streamline the testing process by converting human-readable assembly language into machine code, ensuring accurate execution of test programs.

## Project Scope

The project encompasses the design, implementation, and testing of a 32-bit ALU based on the MIPS architecture. Activities include detailed design work, coding in VHDL, and extensive simulation to ensure the ALU functions correctly and efficiently. The scope extends to designing other necessary MIPS components to integrate the ALU seamlessly within the CPU datapath. Furthermore, the development of a MIPS assembler in C++ is included to facilitate the testing and validation of the ALU and the overall MIPS architecture. This project does not cover the development of a complete MIPS processor beyond the necessary components for ALU integration, nor does it delve into advanced optimization techniques beyond basic enhancements for performance and efficiency. Limitations include the focus on a single-cycle implementation and the exclusion of more complex pipelining and parallel processing techniques.

## Report Outline

This report is structured to provide a clear and comprehensive overview of the project's objectives, methodologies, and outcomes. The introduction sets the stage by highlighting the significance of the ALU and the MIPS architecture. The literature review follows, summarizing recent research and providing a comparative analysis of various approaches to ALU and processor design. The design chapter details the methodology and design choices made during the project. Subsequent chapters cover the implementation and testing phases, presenting simulation results and discussing the performance of the ALU within the MIPS architecture. The inclusion of the MIPS assembler and its role in the testing process is also elaborated. Finally, the report concludes with a summary of findings, reflections on the project's impact, and suggestions for future work.

# LITERATURE REVIEW

## Introduction

The study of processor architecture, particularly in the context of the MIPS (Microprocessor without Interlocked Pipelined Stages) design based on Reduced Instruction Set Computing (RISC) principles, is a crucial area of interest in computer engineering. This project, focusing on the design and implementation of a 32-bit ALU within the MIPS framework, aims to address key challenges and gaps identified in existing literature.

The scope of this literature review includes an examination of recent research and developments related to ALU design and MIPS architecture. This review will cover scholarly articles, technical reports, and industry standards from the past decade to provide a comprehensive understanding of current advancements and persisting issues in the field. The review will also consider industrial applications and commercially available products to highlight practical implementations and identify areas for further improvement.

## Literature Review

### Synthesis and Simulation of a 32-bit MIPS RISC Processor using VHDL

The paper titled "Synthesis and Simulation of a 32Bit MIPS RISC Processor using VHDL" [3] by Mr. S. P. Ritpurkar, Prof. M. N. Thakare, and Prof. G. D. Korde explores the design and simulation of a 32-bit MIPS RISC Processor using Very High-Speed Integrated Circuit Hardware Description Language (VHDL). The primary objective is to analyze the components of the processor, including the Instruction fetch module, Decoder module, and Execution module, which encompasses a 32-bit Floating-point ALU, Flag register, MIPS Instruction Set, and 32-bit general-purpose registers. The design theory is based on the 32-bit MIPS RISC Processor, incorporating a pipeline concept for improved efficiency.

The paper begins by distinguishing between Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) architectures. It provides insights into the characteristics of CISC, exemplifying processors like System 360–IBM, VAX–Digital Equipment Corporation, and Intel x86 architecture. The focus then shifts to RISC, emphasizing its aim for simplicity and speed, with examples such as Apple iPods, iPhones, and MIPS Technologies' prevalence in home electronics.

The MIPS Instruction Set Architecture is introduced, highlighting four formats: R-Type, I-Type, J-Type, and I/O-Type. Each format is explained, depicting their structure and purpose. The paper underscores the advantages of having a smaller number of instructions, uniform instruction sizes, and an organized register file, contributing to the ease of design and cost-effectiveness.

The authors delve into the pipeline design, explaining the five stages: Instruction Fetch (IF), Instruction Decoder (ID), Execution (EXE), Memory and I/O (MEM), and Write-Back (WB). Each stage's role is detailed, emphasizing the decomposition of the pipeline to enhance instruction throughput.

The experimental section presents the Register Transfer Logic (RTL) view of the 32-bit MIPSRISC Processor, showcasing units such as Instruction Fetch, Instruction Decoder, Execution, and Memory. Simulation results for different instruction formats, including R-Type, I-Type, J-Type, and I/O-Type, are provided, demonstrating the successful execution of instructions and the efficiency of the proposed design.

A performance table is presented, comparing the proposed MIPS RISC processor with various latest RISC processors, showcasing its efficiency and competitiveness in terms of delay and maximum operating frequency.

The paper concludes by summarizing the design methodology, VHDL implementation, and successful synthesis and simulation results. The proposed 32-bit MIPS RISC Processor is lauded for its high performance, achieving a delay of 0.741ns and a maximum operating frequency of 1.350 GHz. The clear hierarchy of the design, ease of editing, and successful implementation contributes to the paper's overall conclusion.

### Implementation of a 32-bit MIPS Based RISC Processor using Cadence

The paper titled “Implementation of a 32-bit MIPS Based RISC Processor using Cadence” [4] by Topiwala and Saraswathi, presented at the 2014 IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), focuses on the implementation of a 32-bit MIPS-based RISC processor. RISC architecture, known for its streamlined instruction set and emphasis on performance, has been a significant area of research in computer architecture.

The authors highlight the importance of RISC processors in achieving higher speeds through a reduced set of instructions. The chosen approach involves a 5-stage pipelined architecture, comprising Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB) stages. This structure aligns with the MIPS philosophy, emphasizing efficiency in instruction execution.

The study underscores the significance of load/store architecture in MIPS design, emphasizing the advantages of on-chip registers for faster operations compared to memory locations. To enhance the efficiency of the pipeline, the authors introduce a Hazard Detection Unit and a Data Forwarding Unit. These units play a crucial role in identifying and mitigating data hazards, ensuring a smooth execution flow through the pipeline.

The paper acknowledges the growing relevance of RISC processors in embedded systems, particularly in the "system-on-a-chip" paradigm. The authors discuss the trade-offs involved in integrated circuits, considering power consumption, area utilization, and propagation delay. The implementation is done using Verilog-HDL and Cadence tools, with a focus on optimizing power usage.

In terms of instruction types, the authors delve into the MIPS architecture, covering Register Type (R-Type), Immediate Type (I-Type), and Jump Type (J-Type). The discussion provides a comprehensive overview of the instruction set and its impact on the overall design.

The paper addresses power reduction challenges, emphasizing the importance of minimizing unnecessary switching activity to achieve lower dynamic power consumption. Simulation results are presented, and performance metrics such as area, power dissipation, and propagation delay are analyzed using Cadence RTL Compiler with TSMC 0.18μm technology libraries.

In conclusion, Topiwala and Saraswathi's work significantly contributes to the understanding of 32-bit MIPS-based RISC processor design. The incorporation of pipeline architecture, hazard detection, and data forwarding reflects contemporary trends in computer architecture. The study serves as a valuable resource for researchers and practitioners in the fields of VLSI design and computer architecture, offering insights into optimizing performance and power efficiency.

### Design and Implementation of 32-bit Functional Unit for RISC Architecture Applications

The paper titled "Design and Implementation of 32-bit Functional Unit for RISC architecture applications" [5] by Rashmi Samanth, Ashwini Amin, and Subramanya G. Nayak presents the design and implementation of a 32-bit functional unit for RISC-based processors. The functional unit includes modules such as Arithmetic and Logic Unit (ALU) that performs operations like addition, subtraction, multiplication, shifting, and code conversion. These operations are controlled by suitable control units and data paths, utilizing multiplexers for selecting various operations based on control inputs. The design is carried out using Hardware Description Language (HDL), and simulation and synthesis are performed using Xilinx ISE to analyze the results.

The introduction outlines the significance of high-speed data path logic system design in VLSI. The authors specifically target the MIPS architecture, a RISC-based processor. The paper aims to enhance the efficiency of the processor by designing a 32-bit ALU using behavioral and structural modelling.

The RISC architecture adopted follows a simple load/store architecture. Operations are performed based on operand registers, and memory access is restricted to load/store instructions. The architecture comprises an ALU, multiplexers, multipliers, and register files. Functional units play a crucial role in manipulating data in the processor's data path.

The proposed 32-bit ALU architecture includes logical units, arithmetic units, multipliers, shifters, and code converters. Operations are controlled by multiplexers that govern memory address computations, arithmetic, and logical operations. The design is implemented in Verilog HDL for RISC-based applications, and simulation is carried out using Xilinx ISE. The comparison with a conventional MIPS ALU demonstrates a reduction in power dissipation by 30.44%, area by 6%, and delay by 34.49%.

In conclusion, the paper successfully implements the functional unit using behavioral and structural modelling. The design, simulated using Xilinx ISE and Cadence RTL Compiler, showcases reduced power, area, and delay compared to a conventional MIPS design. The focus on an energy-efficient ALU suggests potential improvements using low-power techniques. The references cited in the paper cover various aspects of processor design, providing a comprehensive background for the proposed work.

## Comparison of Results

The reviewed papers collectively provide a comprehensive overview of 32-bit processor design within the MIPS architecture, focusing on RISC principles. While each paper approaches the topic from a slightly different angle, they all contribute valuable insights into processor efficiency, performance, and design optimization.

Table 2.1 illustrates the findings of the literature review for the three main papers that were focused on.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Paper Title | Year | Idea | Technology | Application | Advantage | Finding |
| Synthesis and Simulation of a 32Bit MIPS RISC Processor using VHDL | 2014 | Design and simulation of a 32-bit MIPS RISC Processor | VHDL | Processor design and implementation | Efficient pipelined architecture | Successful execution and efficiency demonstrated, with a delay of 0.741ns and a maximum operating frequency of 1.350 GHz. |
| Implementation of a 32-bit MIPS Based RISC Processor using Cadence | 2014 | Implementation of a 32-bit MIPS-based RISC processor | Verilog, and Cadence tools | Processor design and implementation | Hazard Detection and Data Forwarding Units | Optimized performance through the incorporation of pipeline architecture, hazard detection, and data forwarding. |
| Design and Implementation of 32-bit Functional Unit for RISC architecture applications | 2020 | Design and implementation of a 32-bit functional unit for RISC-based processors | Verilog | Functional unit design and implementation | Reduced power dissipation, area, and delay | Practical implementation aspects discussed, addressing power reduction challenges and optimizing performance and power efficiency. |

Table 2.1: Literature review table of findings

The paper by Ritpurkar, Thakare, and Korde emphasizes the efficiency gains achieved through a pipelined architecture but lacks discussion on power consumption and area utilization. Topiwala and Saraswathi's work addresses these aspects by focusing on power reduction and practical implementation using Cadence tools. Samanth, Amin, and Nayak's paper concentrates on the design of an energy-efficient ALU, showcasing significant improvements in power, area, and delay.

## Identifying Research Gaps and Future Steps

The literature review identifies several key areas for further research. One significant aspect is the effectiveness of pipelined architecture in improving processor efficiency, as demonstrated by Ritpurkar, Thakare, and Korde. Further optimization and exploration of pipeline stages could lead to greater efficiency gains. Additionally, integrating hazard detection and data forwarding units, as discussed by Topiwala and Saraswathi, could significantly improve pipeline efficiency.

The importance of designing energy-efficient functional units, particularly ALUs, is highlighted by Samanth, Amin, and Nayak. Further research in this area could result in substantial energy savings without compromising performance. This project will focus on implementing and optimizing these identified enhancements, including pipeline optimization techniques, integration of hazard detection and data forwarding units, and designing energy-efficient functional units. Simulation and synthesis using tools like Intel Quartus Prime and ModelSim will validate these enhancements' efficiency and performance improvements.

# DETAILS OF THE DESIGN

This project involves the design and implementation of a 32-bit MIPS ALU based on RISC principles. It also includes the creation of the remaining MIPS processor components to validate and ensure the ALU's seamless integration into the CPU datapath. These components include the Control Unit and every functional unit of the datapath: Program Counter, Adders, Instruction Memory, Data Memory, Multiplexers, Registers, Shifters, and a Sign Extender. Each component is designed to work in harmony with the ALU, ensuring the overall system operates efficiently.

## Literature-Based Circuit Design

Each of the reviewed papers on MIPS processor design provided valuable insights and methodologies, yet none seemed to encompass all aspects comprehensively. The first paper focused heavily on the VHDL implementation and simulation aspects, showcasing the efficiency gains of a pipelined architecture. However, it lacked detailed discussions on hazard detection and data forwarding, which are crucial for optimizing pipeline efficiency, as discussed in the second paper.

The second paper, while addressing practical implementation aspects and the importance of load/store architecture, did not delve deeply into energy-efficient functional unit design, a key focus of the third paper. The third paper, which emphasized the design of an energy-efficient ALU, did not extensively cover pipelined architecture or hazard detection mechanisms. Figures 3.1 and 3.2 show the proposed MIPS pipelined processor without and with improved datapath, respectively, as designed by Mohit N. Topiwala and N. Saraswathi, the authors of "Implementation of a 32-bit MIPS Based RISC Processor using Cadence" [4]. Figure 3.3 shows the RTL view of the proposed MIPS RISC processor by Ritpurkar, Thakare, and Korde, the authors of "Synthesis and Simulation of a 32-bit MIPS RISC Processor using VHDL" [3].

A diagram of a computer hardware

Description automatically generated

Figure 3.1: MIPS pipelined processor datapath by Topiwala and Saraswathi

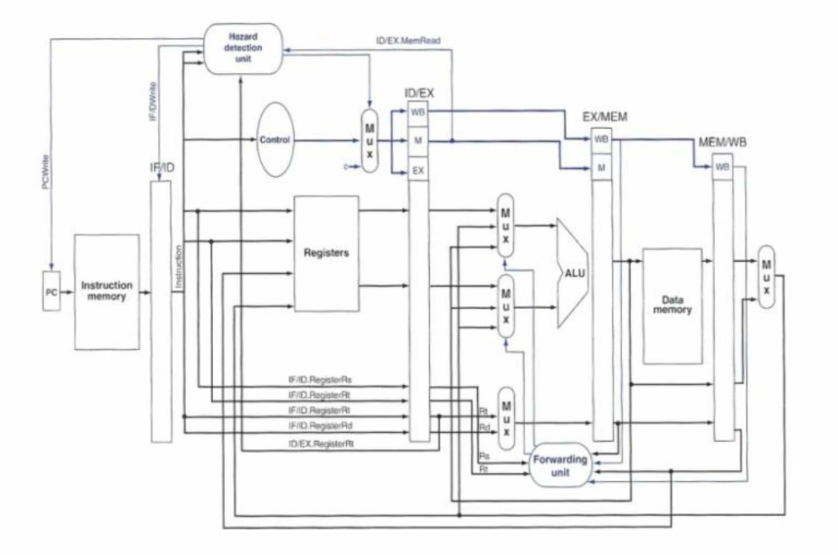


Figure 3.2: Pipelined architecture with improved datapath by Topiwala and Saraswathi

A computer screen shot of a circuit board

Description automatically generated

Figure 3.3: RTL view of proposed MIPS RISC processor by Ritpurkar, Thakare, and Korde

In my proposed design of the 32-bit Single-Cycle MIPS CPU, I aim to incorporate the strengths of each paper while addressing their respective shortcomings. The design will feature a pipelined architecture for improved efficiency, robust hazard detection, and data forwarding mechanisms for optimized pipeline operation.

A diagram of a machine

Description automatically generated

Figure 3.4: 32-Bit Single-Cycle MIPS Block Diagram

At the core of the block diagram, as shown in Figure 3.4, lies the ALU, depicted as the central processing unit for arithmetic and logic operations. The ALU receives input operands from the Register File and the ALU control signal from the Control Unit. This control signal, generated by the Control Unit based on the current instruction being executed, determines the type of operation the ALU will perform. The ALU then executes the specified operation on the input operands, producing the result.

Surrounding the ALU are various supporting components that facilitate its operation within the MIPS architecture. The Control Unit is a critical component of the MIPS processor, responsible for generating control signals that coordinate the operation of other components based on the current instruction being executed. It receives the instruction opcode as input and decodes it to determine the type of instruction and the corresponding control signals needed for its execution.

For R-type instructions such as addition, subtraction, logical AND, OR, NOR, XOR, left shift, and right shift, the Control Unit generates control signals to select the appropriate ALU operation and set other necessary flags. It handles I-type instructions like load word (LW), store word (SW), AND immediate (ANDI), OR immediate (ORI), set less than immediate (SLTI), branch on equal (BEQ), and branch on not equal (BNE) by generating control signals to facilitate memory operations, ALU operations, and branching based on comparison results. Additionally, the Control Unit manages J-type instructions like jump (J) by setting the appropriate control signals to modify the Program Counter (PC) value, enabling the processor to jump to a different instruction address in memory.

The Program Counter (PC) is a register in the MIPS processor that holds the address of the current instruction being executed. It is responsible for guiding the sequential fetching of subsequent instructions from memory. The PC is incremented after each instruction fetch, ensuring that the processor fetches instructions in the correct sequence. The Instruction Memory is a storage component in the MIPS processor that holds the program instructions. It supplies instructions to the Instruction Fetch (IF) unit for decoding and execution. The Instruction Memory is organized as a sequential memory array, with each memory location storing a single instruction.

The Register File is a set of registers in the MIPS processor that provides storage for the processor's general-purpose registers. These registers are used to store operands for ALU operations, temporary data values, and other essential information during program execution. The Register File enables efficient access to operands, improving the processor's overall performance. The Sign Extender is a component in the MIPS processor that extends immediate values to a full 32-bit size. It ensures compatibility with ALU operations and other components within the processor. The Sign Extender plays a crucial role in processing immediate values used in instructions like add immediate (ADDI) and load word (LW).

The Data Memory is a component in the MIPS processor that serves as a repository for storing data values that need to be accessed by load and store instructions. It is organized as a sequential memory array, similar to the Instruction Memory, but dedicated to data storage. The Data Memory facilitates efficient data access and manipulation within the processor. Adders are components within the MIPS processor that are instrumental in address calculation for various instructions, particularly branch and jump instructions. They perform addition operations on two input operands to calculate the target address for the branch or jump. Adders play a crucial role in facilitating efficient branching and jumping within the processor.

Multiplexers (MUXs) are components used throughout the MIPS processor to select between different sources of input data or control signals based on the specific requirements of each instruction. They enhance the processor's adaptability and versatility by enabling it to choose the appropriate data or control signals for each operation. Shifters are components within the MIPS processor that are necessary for instructions that involve shifting the bits of a data operand, such as left shift and right shift operations. They perform logical or arithmetic shift operations on data values, enabling the processor to execute shift instructions efficiently.

The project also involves the development of a MIPS assembler in C++ to encode MIPS instructions into binary 32-bit format for storage in the instruction memory. This assembler allows users to define and execute custom test programs, facilitating thorough testing and validation of the ALU and other MIPS components.

## Proposed Circuit Modifications

Building on the early MIPS processor designs, my proposed 32-bit Single-Cycle MIPS CPU, as shown in Figure 3.4, integrates the strengths of each paper while addressing their respective shortcomings. The first paper's focus on VHDL implementation and simulation efficiency serves as a solid foundation. However, it lacked in-depth discussions on critical aspects like hazard detection and data forwarding, as highlighted in the second paper. The practical implementation insights from the second paper, especially regarding load/store architecture, further enrich the design.

Incorporating the third paper's emphasis on energy-efficient ALU design, my CPU aims to achieve a balance between performance and power consumption. By blending these approaches, the CPU design strives to overcome the limitations of individual papers, creating a holistic processor architecture. The design's key features include a pipelined architecture for enhanced efficiency, robust hazard detection mechanisms, and energy-efficient functional units, reflecting the best practices and innovations from the literature review.

The design of the 32-bit Single-Cycle MIPS CPU will incorporate key elements such as efficient pipelined architecture, robust hazard detection, data forwarding mechanisms, and energy-efficient functional units. The primary objective is to create a circuit that integrates these elements to achieve optimal performance and efficiency. The circuit design adheres to the principles of a single-cycle MIPS processor, which executes each instruction in a single clock cycle. In a single-cycle CPU, each instruction must finish in one clock cycle, so the cycle's length is based on the time it takes for the slowest instruction. This is usually determined by the longest path an instruction takes, often the load word (LW) instruction. This design is informed by the methodologies discussed in the reviewed papers, ensuring a comprehensive and efficient processor architecture.

The design of the 32-bit Single-Cycle MIPS CPU will include several key components, each playing a crucial role in the processor's operation. The Control Unit will manage the control signals required to coordinate the operation of the processor, ensuring the correct sequencing of operations and data flow. The Program Counter (PC) will hold the address of the next instruction to be executed, facilitating instruction fetch operations. Adders will be used in various stages of the pipeline to perform arithmetic operations, including address calculations for the PC and memory access. The Instruction Memory will store the instructions to be executed by the CPU, accessed during the instruction fetch stage. The Data Memory will store data that is read or written during the execution of load and store instructions. Multiplexers will direct the flow of data between different components based on control signals, enabling flexible data handling. Registers will provide storage for intermediate data and operands used during instruction execution. Shifters will perform shift operations required by certain instructions. The Sign Extender will extend the sign of immediate values to ensure correct arithmetic operations with varying operand sizes.

A diagram of a machine

Description automatically generated

Figure 3.4: 32-Bit Single-Cycle MIPS Block Diagram

The design will incorporate a pipelined architecture as discussed in the "Synthesis and Simulation of a 32Bit MIPS RISC Processor using VHDL" paper. The pipelined stages include Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). This structure enhances instruction throughput by allowing multiple instructions to be processed simultaneously at different stages of completion. To handle data hazards and ensure efficient pipeline operation, the design will integrate hazard detection and data forwarding units as highlighted in the "Implementation of a 32-bit MIPS Based RISC Processor using Cadence" paper. These units will detect potential hazards and forward data between pipeline stages to avoid stalls and maintain smooth execution flow.

A key focus of the design will be the Arithmetic and Logic Unit (ALU), which performs arithmetic and logical operations required by instructions. Based on the insights from the "Design and Implementation of 32-bit Functional Unit for RISC architecture applications" paper, the ALU will be designed with energy efficiency in mind. Techniques such as low-power design methodologies and efficient control logic will be employed to minimize power consumption while maintaining high performance.

Another critical aspect of the design is the implementation of instruction pipelining [6] to enhance overall processor efficiency. Instruction pipelining allows for the simultaneous execution of multiple instructions by overlapping their execution phases, significantly improving the throughput of the processor. The R-type instruction in a MIPS architecture, which performs operations such as addition, subtraction, and logical operations, goes through five distinct stages in the pipeline. In the Instruction Fetch (IF) stage, the R-type instruction is fetched from memory. The Program Counter (PC), which is byte-addressed, holds the address of the next instruction to be executed, which is sent to the Instruction Memory. The fetched instruction is then stored in the Instruction Register. The PC is incremented by 4 bytes to point to the next instruction in sequence, ensuring that the instruction flow continues smoothly (as shown in Figure 3.5).

A computer screen shot of a diagram

Description automatically generated

Figure 3.5: Instruction fetch stage of R-type instruction

During the Instruction Decode (ID) stage, the fetched instruction is decoded to determine the operation to be performed and the operands involved. The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. The inputs (RS and RT) carrying the register number to the register file are all 5-bit wide, whereas the lines carrying data values are 32-bit wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which is 4-bit wide (as shown in Figure 3.6).

A diagram of a computer

Description automatically generated

Figure 3.6: Instruction decode stage of R-type instruction

In the Execution (EX) stage, the actual operation specified by the R-type instruction is performed by the ALU. The ALU takes the operands from the source registers and executes the operation (such as addition, subtraction, or a logical operation). The result of this operation is produced and temporarily stored. This stage is crucial as it determines the outcome of the instruction based on the ALU’s performance and the operands provided (as shown in Figure 3.7).

A computer screen shot of a diagram

Description automatically generated

Figure 3.7: Execute stage of R-type instruction

The Memory Access (MEM) stage typically involves accessing memory for load and store instructions. However, for R-type instructions, this stage does not perform any memory operations as they involve only register-to-register operations. The data generated by the ALU in the Execution stage is passed directly to the next stage. This ensures that no unnecessary memory operations are performed, maintaining the efficiency of the pipeline.

Finally, in the Write Back (WB) stage, the result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register. This stage completes the execution of the R-type instruction, ensuring that the computed data is correctly recorded and accessible for future operations. The destination register is updated with the result, making it available for subsequent instructions that may require it. This stage plays a crucial role in maintaining the state of the processor and ensuring that the results of operations are properly stored and utilized (as shown in Figure 3.8).

A computer screen shot of a diagram

Description automatically generated

Figure 3.8: Write back stage of R-type instruction

It's important to note that while the R-type instruction was used as an example, other instruction types in the MIPS architecture also go through the same five stages in the pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). Each stage is responsible for a specific set of operations, and by overlapping these stages, the processor can achieve higher throughput and improved performance.

The design process will involve detailed implementation and simulation using Intel Quartus Prime and ModelSim. These tools will facilitate the creation, verification, and optimization of the processor architecture. VHDL will be used to describe the entire processor architecture, ensuring a modular and maintainable codebase. ModelSim will be employed for functional and timing simulations to verify the correctness and performance of the design. Quartus Prime will be used to synthesize the VHDL code into a gate-level netlist, optimizing the design for the target FPGA platform. The synthesized design will be mapped onto the FPGA, followed by placement and routing to meet design constraints. Post-synthesis and post-implementation simulations will be conducted to ensure the design meets performance metrics. Testing on actual FPGA hardware will validate functionality in a real-world environment.

## FPGA-Based Circuit Design

Learning to code for an FPGA presents a unique journey, markedly different from typical CPU programming. Writing VHDL code for an FPGA essentially involves describing logic to build a CPU, whereas traditional programming runs directly on a CPU. Recognizing this distinction is crucial to avoid misconceptions. In FPGA programming, all operations are concurrent, as circuits with multiple interconnected wires are defined. In VHDL, the logical behavior of a circuit is specified, relying on FPGA tools to synthesize the design into the appropriate gates on the FPGA.

Following the proposed design outlined in the previous section.

1. **Program Counter:**

Figure 3.9 shows the VHDL code implementation for the Program Counter (PC).

**A screenshot of a computer program

Description automatically generated**

Figure 3.9: Program Counter VHDL code

The program counter synchronizes with the clock’s rising edge (transition from a low state 0 to a high state 1), always pointing to the next instruction. It includes an asynchronous reset for starting fresh. During normal operation, it follows the next instruction address, ensuring smooth execution and allowing jumps within the program. Asynchronous signals act right away without waiting for a clock, like an instant reset. Synchronous signals sync up with a clock’s rhythm, like a program counter updating on every tick.

1. **Adder:**

Figure 3.10 shows the VHDL code implementation for the Adder.

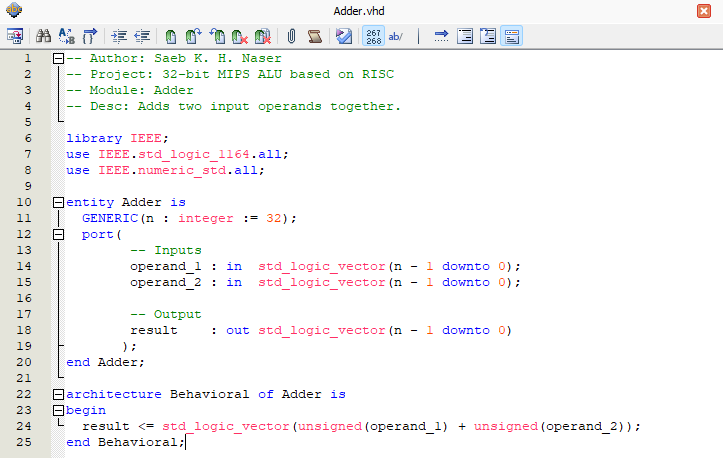
****

Figure 3.10: Adder VHDL code

The Adder adds two input operands together. It’ll be used in several places, like calculating the next PC value and in the ALU for arithmetic operations.

1. **Instruction Memory:**

Figures 3.11, 3.12, and 3.13 show the VHDL code implementation for the Instruction Memory.

**A screenshot of a computer program

Description automatically generated**

Figure 3.11: Instruction Memory VHDL code - part 1

A screenshot of a computer

Description automatically generated

Figure 3.12: Instruction Memory VHDL code - part 2

A screenshot of a computer

Description automatically generated

Figure 3.13: Instruction Memory VHDL code - part 3

This instruction memory is byte-addressable, meaning that each address points to an individual byte of data. Each MIPS instruction is 32 bits long, equivalent to 4 bytes. To accommodate this, instructions are stored across four consecutive memory addresses. For instance, if an instruction starts at address 0, it will occupy the bytes at addresses 0, 1, 2, and 3. To simplify it, the instruction memory will retrieve a full instruction by doing these steps:

1. A 32-bit address is sent to the instruction memory.
2. The memory will fetch four successive bytes starting from this address.
3. These bytes are then concatenated in sequences to reconstruct the full 32-bit instruction.

The instruction memory contains a test program, but we can define our own program instructions using the Assembler I implemented in C++, which was originally written in Python [7].

This porting effort resulted in faster execution and the capability to define any MIPS program for testing purposes. The decision to port the assembler was motivated by the need for improved performance and flexibility in defining and testing MIPS programs. This enhancement allows for a more efficient development process and better testing of the MIPS CPU design.

Figure 3.14 shows the C++ header file code for the MIPS Assembler.

A screen shot of a computer program

Description automatically generated

Figure 3.14: MIPS Assembler C++ header file code

Figures 3.15, 3.16, 3.17, 3.18, 3.19, 3.20, 3.21, and 3.22 show the C++ source file code for the MIPS Assembler.

A screenshot of a computer program

Description automatically generated

Figure 3.15: MIPS Assembler C++ source file code - part 1

A screen shot of a computer program

Description automatically generated

Figure 3.16: MIPS Assembler C++ source file code - part 2

A screenshot of a computer program

Description automatically generated

Figure 3.17: MIPS Assembler C++ source file code - part 3

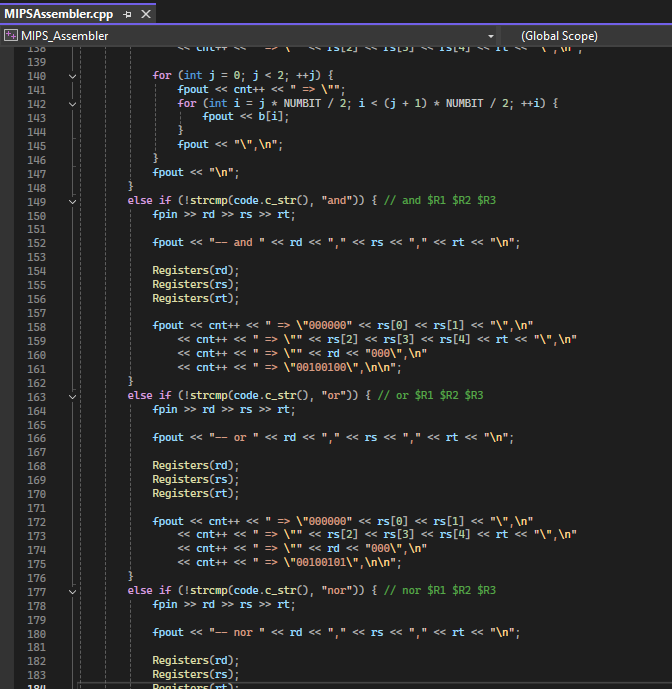


Figure 3.18: MIPS Assembler C++ source file code - part 4

A screen shot of a computer program

Description automatically generated

Figure 3.19: MIPS Assembler C++ source file code - part 5

A screenshot of a computer program

Description automatically generated

Figure 3.20: MIPS Assembler C++ source file code - part 6

A computer screen shot of a program

Description automatically generated

Figure 3.21: MIPS Assembler C++ source file code - part 7

A screenshot of a computer program

Description automatically generated

Figure 3.22: MIPS Assembler C++ source file code - part 8

Figure 3.23 shows the C++ entry point (main) code for the MIPS Assembler.

A screen shot of a computer program

Description automatically generated

Figure 3.23: MIPS Assembler entry point (main) code

The assembler operates by reading instructions from an input file, mapping register names to their binary representations using a lookup table, encoding each instruction into binary format, and writing the encoded instructions to an output file. The assembler is customized via command line arguments, which specify the input file containing the assembly instructions and the output file for the machine code, allowing to easily change input and output files without modifying the code.

For example, running ./MIPSAssembler input.asm output.bin, as shown in Figure 3.24, processes the instructions in input.asm and writes the machine code to output.bin.

A screenshot of a computer

Description automatically generated

Figure 3.24: MIPS Assembler in action

To ensure the correctness of the assembler, I extensively tested it with various MIPS assembly programs, verifying that the generated machine code matched the expected binary representation for each instruction. This involved comparing the output of the assembler with the binary machine code generated by other tools and simulators. Now, we can easily write any program in MIPS Assembly and feed it to the instruction memory and test our processor. The code for this project, including the Assembler C++ source code and the executable binary, is available on my GitHub repository [8].

1. **Multiplexer:**

Figure 3.25 shows the VHDL code implementation for the Multiplexer.

**A screenshot of a computer program

Description automatically generated**

Figure 3.25: Multiplexer (MUX) VHDL code

The multiplexer is a data selector, it selects one of two inputs signals based on a control signal.

1. **Registers:**

Figure 3.26 shows the VHDL code implementation for Registers.

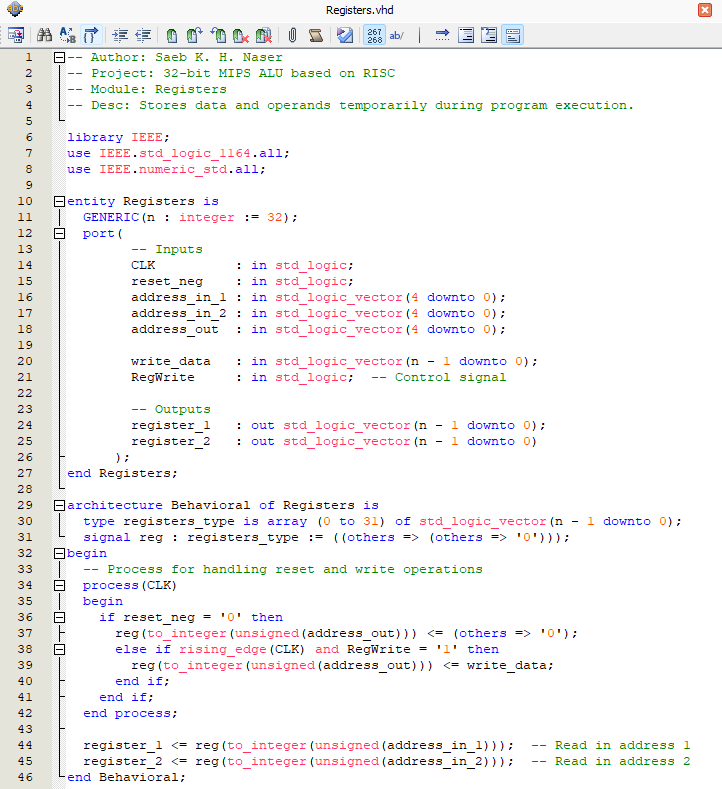
****

Figure 3.26: Registers VHDL code

Registers store data and operands temporarily during program execution.

The main operations are performed on changes to the CLK signal. When the reset\_neg signal is low 0, indicating an active-low reset condition, it sets the specified register (indicated by address\_out) to zero. This is a focused reset, affecting only one register.

On the rising edge of the clock (CLK), if the RegWrite signal is high 1, the data from write\_data is written into the register specified by address\_out. This operation allows updating of register contents during normal operation.

The module continuously assigns values from the register file to the outputs register\_1 and register\_2. These assignments use address\_in\_1 and address\_in\_2 to select which registers’ contents are read, making these outputs always reflect the current contents of the specified registers.

1. **Sign Extend:**

Figure 3.27 shows the VHDL code implementation for the Sign Extend module.

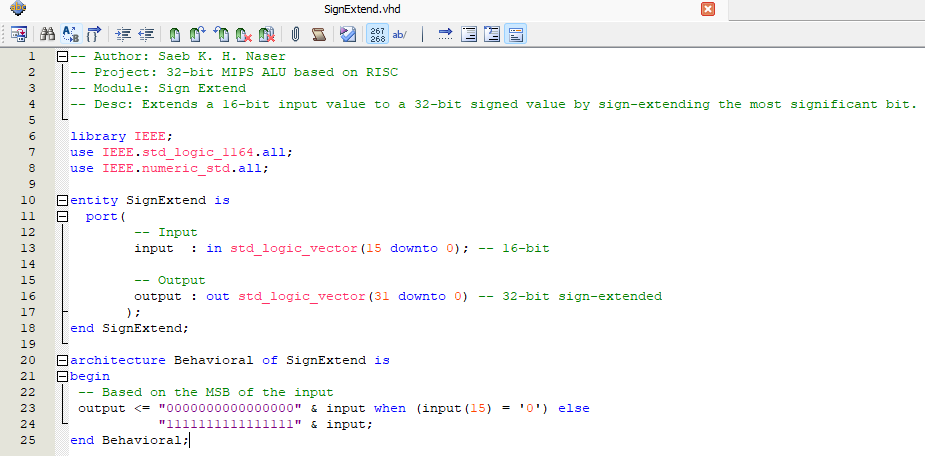
****

Figure 3.27: Sign Extend VHDL code

Sign Extend module will extend a 16-bit input value to a 32-bit signed value by sign-extending the MSB. If the 16-bit input has an MSB of 0 (positive), it prepends 16 zero bits to it, maintaining the value as positive in a 32-bit signed format. On the other hand, if the MSB is 1 (negative), it prepends 16 one bits to it, effectively extending the sign and preserving the negative value in two’s complement form.

1. **Shift Left 2:**

Figure 3.28 shows the VHDL code implementation for the Shift Left 2 module.

**A screenshot of a computer program

Description automatically generated**

Figure 3.28: Shift Left 2 VHDL code

Shift Left 2 module shifts the input value to the left by two positions, effectively multiplying it by 4. Left shifting multiplies the number by 2 for each shift, while right shifting divides it by 2.

1. **Data Memory:**

Figure 3.29 shows the VHDL code implementation for the Data Memory.

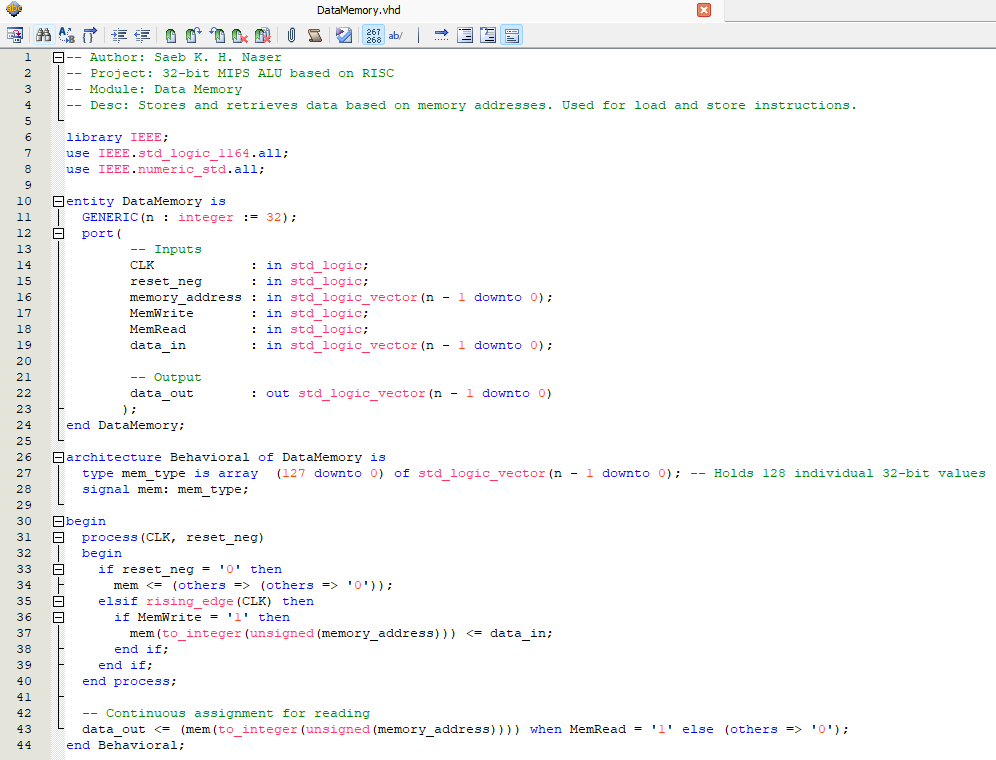
****

Figure 3.29: Data Memory VHDL code

The data memory stores and retrieves data based on memory addresses. Used only for load and store instructions.

1. **ALU:**

Figures 3.30 and 3.31 show the VHDL code implementation for the ALU.

**A screenshot of a computer

Description automatically generated**

Figure 3.30: ALU VHDL code - part 1

A screenshot of a computer screen

Description automatically generated

Figure 3.31: ALU VHDL code - part 2

The ALU, our focal point in the processor's datapath, is responsible for executing arithmetic and logical operations crucial to instruction execution. The architecture Behavioral specifies the operations based on the ALU\_control signal, which selects from nine operations including addition, subtraction, logical AND, OR, NOR, NAND, XOR, left shift, and right shift. The result of the operation is stored in temp, and a process is used to detect if the result is zero, updating the is\_zero signal accordingly. Finally, the result is assigned to result and the zero flag to zero.

1. **Control Unit:**

Figures 3.32, and 3.33 show the VHDL code implementation for the Control Unit.

**A screenshot of a computer

Description automatically generated**

Figure 3.32: Control Unit VHDL code - part 1

A screenshot of a computer

Description automatically generated

Figure 3.33: Control Unit VHDL code - part 2

The control unit has the most complicated behavior, it generates control signals for various components within the process based on the input instruction and other control signals.

The control unit decodes each instruction to generate control signals that direct the processor’s operation. It identities the operation type, such as arithmetic or branching, and activates the necessary components.

All MIPS instructions are encoded in binary, and this is what the Assembler handles. All instructions have an opcode (op) that specifies the operation, which occupies the first 6 bits of the instruction. For R-type instructions, such as addition, subtraction, and logical operations, it is also necessary to examine the last 6 bits (function field) to determine the specific operation being performed.

1. **Data Path:**

Figures 3.34, 3.35, 3.36, and 3.37 show the VHDL code implementation for the Data Path.

**A screenshot of a computer

Description automatically generated**

Figure 3.34: Data Path VHDL code - part 1

A screenshot of a computer program

Description automatically generated

Figure 3.35: Data Path VHDL code - part 2

A screenshot of a computer

Description automatically generated

Figure 3.36: Data Path VHDL code - part 3

A screenshot of a computer

Description automatically generated

Figure 3.37: Data Path VHDL code - part 4

This module executes instructions by performing arithmetic and logical operations, accessing memory, and updating registers. This data path ties together various components like ALU, Program Counter, Registers, Memory, and Multiplexers to handle instruction fetching, decoding, execution, and the writing back of results. It represents the entire interface of the CPU.

1. **Top-Level:**

Figures 3.38, 3.39, and 3.40 show the VHDL code implementation for the Top-Level module.

**A screenshot of a computer

Description automatically generated**

Figure 3.38: Top-Level VHDL code - part 1

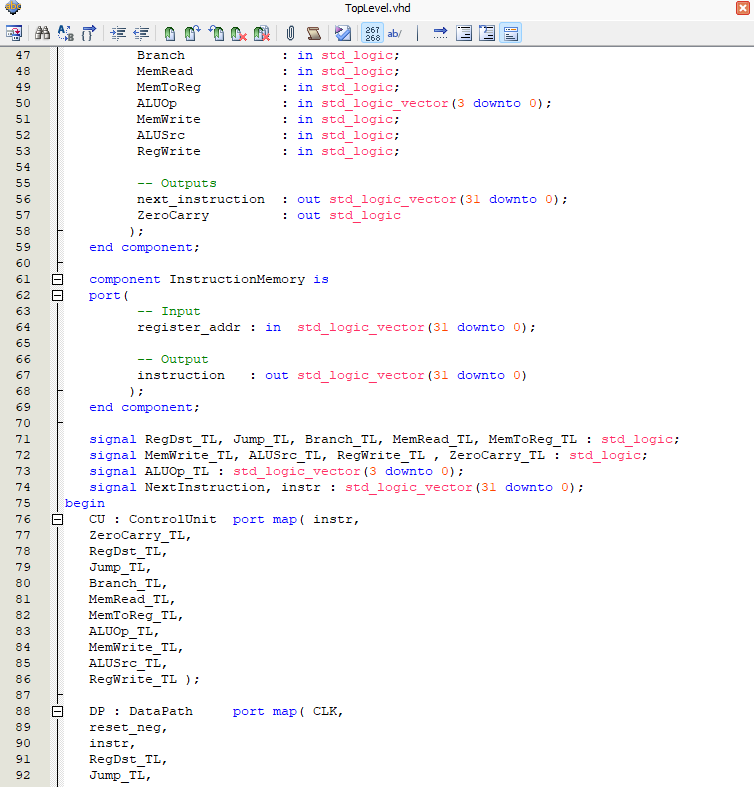


Figure 3.39: Top-Level VHDL code - part 2

A screenshot of a computer

Description automatically generated

Figure 3.40: Top-Level VHDL code - part 3

The Top-Level module coordinates the flow of instructions, control signals, and data between different components to execute instructions and perform desired operations.

Testbenches are essential in VHDL Design, serving as simulation environments, providing input stimuli to the design under test (DUT) and evaluating its responses. In order to validate the functionality and ensure everything works fine in the circuit, writing testbenches is essential.

* **Testbench for the ALU module:**

Figures 3.41, 3.42, and 3.43 show the VHDL code implementation for the ALU module testbench.

**A screenshot of a computer program

Description automatically generated**

Figure 3.41: ALU Testbench VHDL code - part 1

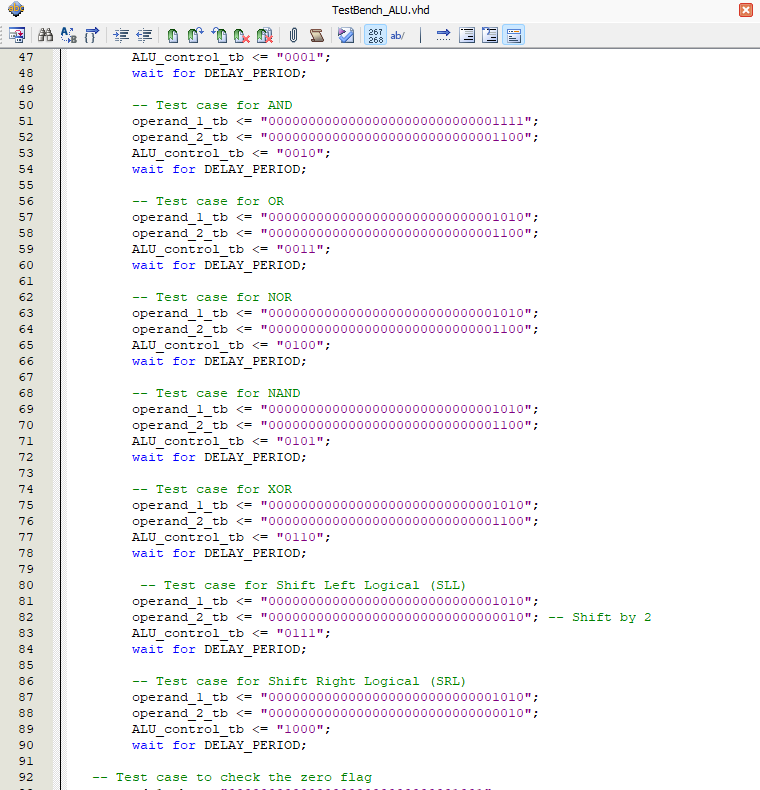


Figure 3.42: ALU Testbench VHDL code - part 2

A screenshot of a computer program

Description automatically generated

Figure 3.43: ALU Testbench VHDL code - part 3

The testbench is designed to verify the functionality of the ALU by applying various test cases for different operations. The testbench includes signals for the operands, the ALU control signal, the result, and the zero flag. Each test case sets the input operands and the ALU control signal to specific values to perform a particular operation such as addition, subtraction, logical AND, logical OR, NOR, NAND, XOR, shift left logical (SLL), and shift right logical (SRL).

The testbench uses a stimulus process to apply the test cases sequentially, allowing for observation of the ALU's output and the zero flag after each operation.

* **Testbench for the Top-Level module:**

Figure 3.44 shows the VHDL code implementation for the Top-Level module testbench.

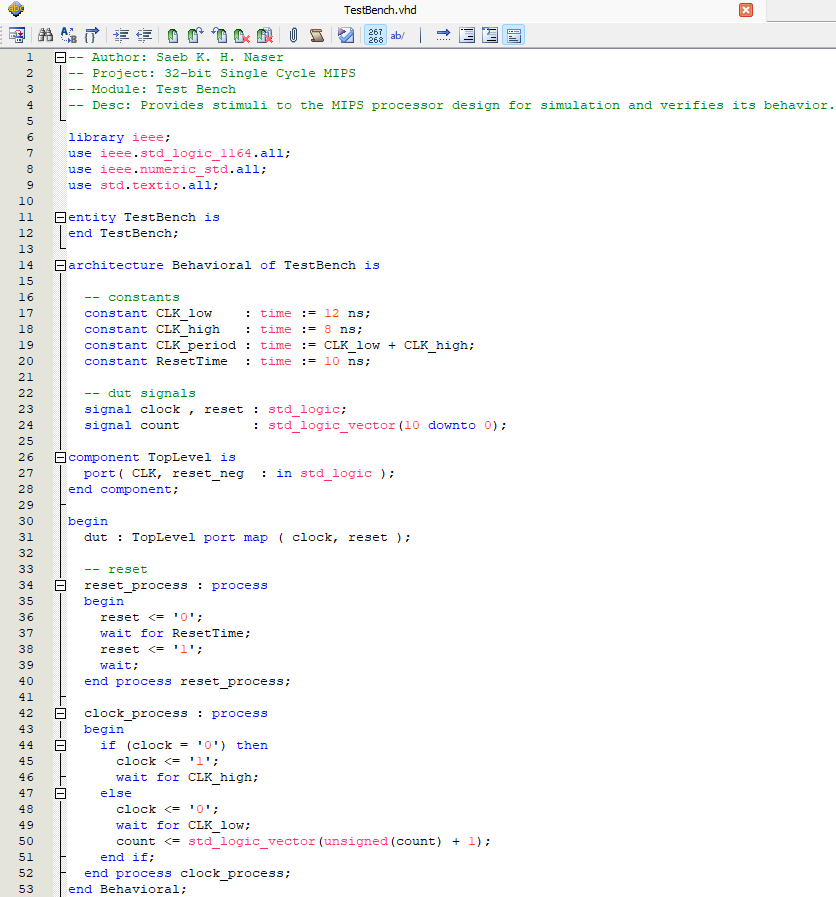
****

Figure 3.44: Top-Level Testbench VHDL code

This testbench is responsible for providing stimuli to the MIPS processor design for simulation and verifying its behavior. The testbench begins by declaring constants for the clock period (CLK\_low, CLK\_high, and CLK\_period) and the reset time (ResetTime). These constants define the timing characteristics of the clock signal and the duration of the reset signal.

The testbench then declares signals for the clock (clock), reset (reset), and a count signal (count) used for counting clock cycles. Next, the testbench instantiates the top-level module of the MIPS processor design (TopLevel) and maps its ports to the signals declared earlier.

The testbench includes two processes. The first process (reset\_process) is responsible for generating the reset signal. It sets the reset signal to '0', waits for a specified reset time (ResetTime), and then sets the reset signal to '1', indicating the end of the reset period.

The second process (clock\_process) generates the clock signal. It toggles the clock signal between '0' and '1' with the specified clock periods. Additionally, it increments the count signal by 1 on each clock cycle, allowing for counting the number of clock cycles elapsed.

The value 12 ns specifies the duration of the clock signal when it is low, while 8 ns specifies the duration of the clock signal when it is high. Together, these values define the period of the clock signal. The value 10 ns (ResetTime) specifies the duration for which the reset signal is asserted before being de-asserted. These timing parameters are essential for ensuring that the testbench generates the correct clock and reset signals to simulate the behavior of the MIPS processor design accurately.

Figure 3.45 shows the VHDL code implementation for the Top-Level module second testbench.

A screenshot of a computer

Description automatically generated

Figure 2.45: Top-Level second Testbench VHDL code

Overall, these testbenches provide a structured environment for simulating the behavior of the MIPS processor design and verifying its functionality under different stimuli.

# DATA PRESENTATION AND DISCUSSION OF FINDINGS

This section, the performance and functionality of the 32-bit Single-Cycle MIPS design based on the implemented circuit and VHDL code are analyzed. The results are presented in terms of simulation outputs and behavioral observations during testing. This analysis is performed using Quartus II 64-Bit Version 13.1.0 Web Edition [9] and ModelSim [10].

## Data Presentation

### RTL Flow Map:

The Register Transfer Level (RTL) flow map illustrates the logical structure of the designed and implemented 32-bit Single-Cycle MIPS processor. This map showcases the interconnections between the Control Unit, Data Path, and Instruction Memory. The RTL flow map generated by Quartus provides a visual representation of the architecture, showing how these components coordinate to execute instructions and perform desired operations. The Control Unit generates control signals based on the instruction, the Data Path executes the operations and handles data flow, and the Instruction Memory supplies the required instructions. This map is crucial for verifying that the design meets the intended structural specifications and helps in identifying any potential issues in the circuit's layout.

Figure 4.1 shows the RTL Flow Map of the designed and implemented 32-bit Single-Cycle MIPS processor.

A screenshot of a computer

Description automatically generated

Figure 4.1: RTL Flow Map of the 32-bit Single-Cycle MIPS processor

To provide a deeper insight into the architecture, RTL flow maps for the individual components of the design are presented as well. These diagrams show the internal structure and how each component interacts with the others within the processor.

Figure 4.2 shows the RTL Flow Map of the Control Unit.

A screenshot of a computer

Description automatically generated

Figure 4.2: RTL Flow Map of the control unit

The Control Unit is responsible for generating the necessary control signals based on the fetched instruction. This unit interprets the opcode and other fields of the instruction to control the operations of the Data Path and other components.

Figure 4.3 shows the RTL Flow Map of the Data Path.

A screenshot of a computer

Description automatically generated

Figure 4.3: RTL Flow Map of the data path

The Data Path is the core of the processor where actual data processing occurs. It includes elements such as the ALU, registers, and multiplexers. This unit performs arithmetic, logical, and memory operations as directed by the Control Unit.

Figure 4.4 shows the RTL Flow Map of the Instruction Memory.

A screenshot of a computer

Description automatically generated

Figure 4.4: RTL Flow Map of the instruction memory

The Instruction Memory holds the program instructions that the processor executes. This memory unit fetches the instructions based on the current program counter and feeds them to the Control Unit for decoding and execution. By using the implemented Assembler shown in earlier sections, we can feed it any MIPS program and test our processor against it.

### Simulation Waveforms:

Using ModelSim, I performed simulations to validate the functionality of the 32-bit Single-Cycle MIPS design. The waveforms generated from these simulations provide a dynamic view of how the processor handles different instructions over time. I wrote two test benches: one focused on the ALU to test the nine operations (addition, subtraction, AND, OR, NOR, NAND,XOR, shift left, and shift right), and another for the Top-Level module to verify the entire processor's behavior. The simulation waveforms below demonstrate the ALU's operations and the overall processor's response to a series of instructions.

* **ALU Simulation Waveforms:**

Figure 4.5 shows the simulation waveform for ALU operations.

**A screenshot of a computer

Description automatically generated**

Figure 4.5: Simulation waveform for ALU operations

The testbench for the ALU module is designed to verify the functionality of the ALU by applying different test cases for each operation. The testbench utilizes a clock signal and defines a constant DELAY\_PERIOD of 10ns to control the timing of the test cases. Each testcase is executed by setting the input operands (operand\_1\_tb and operand\_2\_tb) and the ALU control signal (ALU\_control\_tb), and then waiting for the DELAY\_PERIOD to simulate the processing time of the ALU.

For the simulation waveform analysis of the ALU operations, we focus on four key operations: AND, XOR, SLL (Shift Left Logical), and the zero flag check operation. These operations are crucial as they demonstrate the core functionality and behavior of the ALU. The simulation waveforms provide a visual representation of how the ALU processes inputs and generates outputs for each operation.

* AND Operation: The AND operation performs a bitwise AND between two input operands. In the waveform, we observe that the result (out) is the logical AND of operand\_1 and operand\_2 at each bit position.

Figure 4.6 shows the simulation waveform for AND operation.

A screenshot of a computer

Description automatically generated

Figure 4.6: Simulation waveform for AND operation

The result 1001 is correct for the AND operation, as 0000 indeed indicates the AND operation according to the ALU module implementation.

* XOR Operation: The XOR operation performs a bitwise XOR between two input operands. The waveform illustrates that the result (out) is the logical XOR of operand\_1 and operand\_2 at each bit position.

Figure 4.7 shows the simulation waveform for XOR operation.

A screenshot of a computer

Description automatically generated

Figure 4.7: Simulation waveform for XOR operation

The result 0110 is correct for the XOR operation, as 0110 indeed indicates the XOR operation according to the ALU module implementation.

* Zero Flag Check Operation: This operation checks if the result of a subtraction operation is zero. In the waveform, we observe that the zero flag is set to '1' when operand\_1 and operand\_2 are equal, indicating that the result is zero.

Figure 4.8 shows the simulation waveform for Subtraction operation.

A screenshot of a computer

Description automatically generated

Figure 4.8: Simulation waveform for Subtraction operation

The other six operations of the ALU work similarly well, demonstrating the robustness and correctness of the ALU's design and implementation.

* **Top-Level Simulation Waveforms:**

Figure 4.9 shows the simulation waveform for the Top-Level module second testbench.

**A screenshot of a computer

Description automatically generated**

Figure 4.9: Simulation waveform for Top-Level second testbench

This test bench simulates the operation of the designed single-cycle MIPS, focusing on the sequential execution of 10 instructions across 10 full clock cycles, totaling 200ns. It initializes the clock and active-low reset signals, toggling the clock every 10ns to maintain a 50MHz frequency, thereby establishing a 20ns clock period. The reset is initially asserted and then deasserted to ensure the processor starts from a known state.

Let’s take the first full cycle of our execution, which is according to our test program inside our Instruction Memory is an I-type instruction addi $R1, $R0, 30.

Figure 4.10 shows a screenshot of the instruction memory showing the first instruction (addi).

A screenshot of a computer program

Description automatically generated

Figure 4.10: Screenshot of the instruction memory showing the first instruction

Each MIPS instruction spans 32 bits, which is equivalent to 4 bytes. To accommodate this in the Instruction Memory, I allocated four consecutive memory addresses for each individual instruction. Similarly, the Assembler operates by encoding each instruction into 4 bytes to align with our design. It’s important to note that if an instruction begins at address 0, it will occupy the bytes at addresses 0, 1, 2, and 3. Hence, to move to the next instruction, we increment the program counter by 4.

Figure 4.11 shows the simulation waveform for Addi instruction.

A screenshot of a computer program

Description automatically generated

Figure 4.11: Simulation waveform for Addi instruction

Now we can clearly see that our complete 32-bit instruction is the concatenation of the 4 bytes occupying the first 4 addresses: 0, 1, 2, and 3, resulting in the full instruction 00100000000000010000000000011110.

According to the way I designed the Control Unit, which generates control signals for various components within the processor based on our input instruction (addi in this case), we should expect an 11-bit data signal to set the control signals. In our case, the data signal is 000000000011, which perfectly aligns with our implementation.

Figure 4.12 shows a screenshot of the control unit showing the 11-bit data signal for Addi instruction.

A screenshot of a computer

Description automatically generated

Figure 4.12: Screenshot of the control unit showing the 11-bit data signal for Addi instruction

Returning to our primary focus, the ALU, we observe that the ALUOp, indicating the ALU operation to be performed, is set to 0000. This precisely matches our design, where 0000 signifies addition operation.

Moving to the next instruction sw $R1,0($R0). Figure 4.13 shows the simulation waveform for SW instruction.

A screenshot of a computer

Description automatically generated

Figure 4.13: Simulation waveform for SW instruction

The full instruction, 10101100000000010000000000000000, perfectly aligns with our layout in the Instruction Memory. Additionally, the data signal, 000000000110, matches the control signal settings in the Control Unit. Regarding the ALUOp, it remains at 0000 because this instruction is a Memory write (store word), not an ALU operation. However, if you review the waveform, you’ll notice that the ALUOp changes multiple times, indicating that we encountered 5 ALU operations in our test program.

For simulation with the Altera DE2 board [11], we can supply a 50MHz clock, achieving the same results.

Figure 4.14 shows the simulation waveform for the Top-Level module first testbench.

A screenshot of a computer screen

Description automatically generated

Figure 4.14: Simulation waveform for Top-Level first testbench

### Timing Diagram:

As seen in the earlier simulation waveforms for both testbenches, the designed single-cycle MIPS, focusing on the sequential execution of 10 instructions across 10 full clock cycles, totaling 200ns. It initializes the clock and active-low reset signals, toggling the clock every 10ns to maintain a 50MHz frequency, thereby establishing a 20ns clock period. The reset is initially asserted and then deasserted to ensure the processor starts from a known state.

In a pipelined processor, the execution time of an instruction is determined by the stages it goes through. Assuming a simplified pipeline with stages for instruction fetch, register read, ALU operation, memory access, and register write, each with their respective times, we can calculate the total execution time for different types of instructions.

Let’s assume that the time for stages is as follows:

* 100ps for register read or write.
* 200ps for other stages.

Table 4.1 presents theoretical MIPS pipeline timing analysis based on the assumptions made earlier.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | Instruction Fetch | Register Read | ALU Operation | Memory Access | Register Write | Total Time |
| LW | 200ps | 100ps | 200ps | 200ps | 100ps | 800ps |
| SW | 200ps | 100ps | 200ps | 200ps | - | 700ps |
| R-type | 200ps | 100ps | 200ps | - | 100ps | 600ps |
| BEQ | 200ps | 100ps | 200ps | - | - | 500ps |

Table 4.1: Theoretical MIPS Pipeline Timing Analysis

Figure 4.15 shows the timing analysis of the nonpipelined execution of three Load Word (LW) instructions.

A diagram of a data flow

Description automatically generated

Figure 4.15: Nonpipelined execution of three Load Word instructions

Figure 4.16 shows the timing analysis of the pipelined execution of three Load Word (LW) instructions.

A diagram of a diagram

Description automatically generated

Figure 4.16: Pipelined execution of three Load Word instructions

These figures are sourced from [6]. When comparing the non-pipelined and pipelined execution of three load word instructions, we observe significant differences in their total execution times. In the non-pipelined execution, each instruction takes a single cycle of 800ps. Therefore, the total time to execute three load instructions in the non-pipelined design is calculated as 3 × 800ps, resulting in 2400ps. Conversely, in the pipelined execution, although some stages take only 100ps, the clock cycle is set to 200ps to accommodate the worst-case scenario. The total time to execute three load instructions in the pipelined design is calculated as 200ps × 5 + 200ps × 2, totaling 1400ps. This comparison clearly demonstrates that the pipelined execution, with a total time of 1400ps, outperforms the non-pipelined execution, which requires 2400ps.

In my design, which is a pipelined architecture, I set the clock cycle to 20ns to accommodate the worst-case scenario, allowing for a total time of 200ns for executing 10 instructions. However, it's important to note that real-life implementations often differ from theoretical values due to various factors such as overhead from control logic, data dependencies, and memory access latencies. These factors can introduce additional delays and complexities that are not accounted for in theoretical calculations. As a result, while theoretical analysis provides valuable insights, actual performance in real-life scenarios may vary.

## Discussion of Findings

The performance and functionality of the 32-bit Single-Cycle MIPS design were evaluated based on the implemented circuit and VHDL code. The results, presented through simulation outputs and behavioral observations during testing, provide a comprehensive understanding of the processor's operation. Using Quartus II 64-Bit Version 13.1.0 Web Edition and ModelSim, we validated the design's correctness and efficiency.

The RTL flow maps generated by Quartus illustrate the logical structure of the 32-bit Single-Cycle MIPS processor, showcasing the interconnections between the Control Unit, Data Path, and Instruction Memory. These maps are crucial for verifying that the design meets structural specifications and identifying potential issues in the circuit layout. The Control Unit generates control signals based on the instruction, the Data Path executes operations and handles data flow, and the Instruction Memory supplies the required instructions. The detailed RTL flow maps for individual components further highlight the internal structure and interactions within the processor.

Simulation waveforms, generated using ModelSim, provide a dynamic view of how the processor handles different instructions over time. Two test benches were written: one for the ALU to test nine operations (addition, subtraction, AND, OR, NOR, NAND, XOR, shift left, and shift right), and another for the Top-Level module to verify the entire processor's behavior. The ALU simulation waveforms confirm the correctness of each operation, demonstrating the robustness of the ALU's design. The Top-Level simulation waveforms show the processor's response to a series of instructions, validating the overall functionality.

The first full cycle of the execution, starting with the instruction "addi $R1, $R0, 30", illustrates the correct generation of control signals and ALU operations. The subsequent instruction, "sw $R1, 0($R0)", further validates the instruction memory layout and control signal settings. These simulations confirm that the processor operates correctly across different instruction types, ensuring reliable performance.

Timing diagrams highlight the execution times of different instructions. In a single-cycle processor, each instruction is completed in one clock cycle. The comparison between non-pipelined and pipelined execution of load word instructions demonstrates the efficiency of pipelined architectures, where the total execution time is significantly reduced. In our single-cycle design, the clock cycle was set to 20ns to accommodate the worst-case scenario, allowing for the execution of 10 instructions within 200ns.

While theoretical analysis provides valuable insights, real-life implementations may vary due to factors like control logic overhead, data dependencies, and memory access latencies. These factors can introduce additional delays and complexities not accounted for in theoretical calculations. Despite these limitations, the findings of this study contribute valuable knowledge to the field of processor design, demonstrating the efficiency and functionality of a 32-bit Single-Cycle MIPS processor.

In conclusion, the discussion of findings confirms the successful implementation and validation of the 32-bit MIPS ALU and remaining Single-Cycle MIPS design. The detailed analysis of simulation waveforms and timing diagrams provides a thorough understanding of the processor's operation. The study's approach, limitations, and implications are critically evaluated, highlighting the significance of this project for future developments in processor design and optimization.

# CONCLUSIONS

## Summary and Conclusions

In summary, this project has been a compelling exploration into the complexities of designing and implementing a 32-bit MIPS ALU based on RISC principles, integrated into a single-cycle MIPS processor. The primary focus on the ALU provided a comprehensive examination of its functionality, including various arithmetic and logical operations. Integration into the single-cycle MIPS processor demonstrated a practical application of the ALU within a complete processor architecture.

Key milestones included the design and implementation of the ALU's core logic, the development of a complete single-cycle MIPS processor architecture, successful integration and testing of the ALU within the processor, and the creation of a C++ MIPS Assembler for testing against any MIPS program written in the MIPS instruction set. These milestones underscore a solid understanding of digital design principles and the application of these principles in real-world computing systems.

Insights gained into the complexities of designing and implementing processor components, especially in timing, data flow, and control signals, were significant. The use of tools like Quartus Prime and ModelSim was essential for simulating and validating the design, ensuring correctness and functionality.

In conclusion, this project has significantly advanced technical skills in digital design and processor architecture. The hands-on experience of designing and implementing a 32-bit MIPS ALU integrated into a single-cycle MIPS processor has deepened understanding of fundamental concepts in computer engineering. The challenges faced and overcome during this project, such as timing constraints, data path design, and control signal management, have equipped with valuable problem-solving skills applicable in real-world scenarios. This project has laid a robust foundation for future exploration and innovation in computer engineering.

## Areas of Future Research

Future research in this area could focus on several aspects to further enhance the design and performance of the MIPS ALU and processor. One area of interest could be more optimization of the pipelined architecture for the MIPS processor, which could significantly improve its efficiency and performance by allowing multiple instructions to be processed simultaneously.

Another area of research could be the exploration of advanced optimization techniques for the ALU and processor design. This could include techniques such as parallel processing, which could further improve the processor's performance by allowing it to execute multiple instructions in parallel.

Additionally, future research could also explore the integration of the MIPS processor into larger systems, such as system-on-chip (SoC) designs. This could involve integrating the MIPS processor with other components, such as memory and I/O controllers, to create a complete computing system on a single chip.

Overall, there are many exciting avenues for future research in the field of processor design and optimization, and this project has laid a solid foundation for further exploration in this area.

REFERENCES

[1] MIPS Technologies, Inc. "MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set". Revision 6.03, 2013.

[2] Ranjani Parthasarathi, Computer Architecture. Available at: <https://www.cs.umd.edu/~meesh/411/CA-online/chapter/pipelining-mips-implementation/index.html>.

[3] S. P. Ritpurkar, M. N. Thakare and G. D. Korde, "Synthesis and Simulation of a 32Bit MIPS RISC Processor using VHDL", 2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014), Unnao, India, 2014, pp. 1-6, doi:10.1109/ICAETR.2014.7012843.

[4] M. N. Topiwala and N. Saraswathi, "Implementation of a 32-bit MIPS based RISC processor using Cadence", 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, Ramanathapuram, India, 2014, pp.979-983, doi: 10.1109/ICACCCT.2014.7019240.

[5] R. Samanth, A. Amin and S. G. Nayak, "Design and Implementation of 32-bit Functional Unit for RISC architecture applications", 2020 5th International Conference on Devices, Circuits and Systems (ICDCS), Coimbatore, India, 2020, pp. 46-48, doi:10.1109/ICDCS48716.2020.243545.

[6] Sukjin Lee, Computer Organization. Available at: <https://alg.manifoldapp.org/read/computer-organization/section/4e3aaa0a-6539-4d2b-a515-16e7efd99fc3>.

[7] Python MIPS Assembler (PyMIPS). Available at: <https://github.com/kashyapakshay/PyMIPS>.

[8] My GitHub repository for this Final Year Project “Design a 32-bit MIPS ALU based on RISC”. Available at: <https://github.com/saeb0x/mips32-alu>.

[9] Intel Quartus II 64-Bit Version 13.1.0 Web Edition. Available at: <https://www.intel.com/content/www/us/en/software-kit/666221/intel-quartus-ii-web-edition-design-software-version-13-1-for-windows.html>.

[10] ModelSim. Available at: <https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html>.

[11] Altera DE2 Board. Available at: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=30>.