Design & Development of a High-speed Performance ALU by Using Execution Modes and Multi-Operand Operation

Abdullah Saad Albishri

Supervisor by

Dr.Nazar EL Fadel

A Thesis submitted for the requirements for the

of Master degree in Computer Engineering

At the Faculty of Graduate Studies

Fahd Bin Sultan University

June 2023

Design & Development of a High-speed Performance ALU by Using Execution Modes and Multi-Operand Operation

By

Abdullah Saad Albishri

Signature of Author ……

Committee Member		Signature and Date
Dr. Nazar El Fadel	(Chairman)	……
Prof. Samir Bataineh	(Member)	……
Dr. Mutsam Jarajreh	(Member)	……

ABSTRACT

Design and Development of a High-Performance ALU Using Execution Modes and Multi-Operand Operation

The ALU is a critical component in contemporary computer design, responsible for executing arithmetic and logical operations on data. In this thesis, the researcher explores a new type of ALU, known as the triple arithmetic logic unit (Tri-ALU), which extends the capabilities of traditional ALUs by integrating multiple ALU architectures and introducing four compact operations with execution modes. These enhancements aim to improve data processing efficiency and overall system performance. The objective of this study is to enhance data processing performance by reducing memory access, minimizing software size, and utilizing operator fusion and routing fusion within the Tri-ALU, serving as the processing unit within the central processing unit (CPU). This enables efficient execution of complex and adaptable operations. The methodology employed in this research involves utilizing hybrid architecture techniques, including operator fusion, routing fusion, and execution modes. Operator fusion enables the Tri-ALU to accommodate three operands and perform two arithmetic and logic operations, as well as an index operation for loop processes and shift operations. Routing fusion facilitates the routing of operation results to multiple register destinations. The execution mode automates and controls the Tri-ALU for specific tasks. The Tri-ALU is constructed using a hybrid architecture to minimize software size. The research findings demonstrate a significant improvement in performance, as evidenced by reduced CPU operation time, minimized power consumption, and decreased software size.

Keywords: Register transfer level (RTL), Hardware description language (HDL), Arithmetic logic unit (ALU), Field programmable gate array (FPGA), Processing unit (PE), Artificial neural networks (ANNs), Multiplier–accumulator (MAC unit), Vector processing (VP), Instruction-level parallelism (ILP).

TABLE OF CONTENTS

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Adaptive Logic Module (ALM)
(ALM), 20
Arithmetic Logic Unit (ALU)
(ALU), 1
central processing unit (CPU)
(CPU), 1
Digital Signal Processors
(DSP), 23
Field-Programmable Gate Array (FPGA)
(FPGA), 64
functional unit for arithmetic operations
(VFU)
(VFU), 28
high performance computing
HPC, 27
Microprocessor without Interlocked Pipeline Stages
MIPS, 29
Multiple instruction, multiple data (MIMD)
(MIMD), 42
partial run-time reconfiguration
(PR), 101
Proportional–integral–derivative controller (PID)
(PID), 85
Scalable Vector Extension (SVE)
(SVE), 27
Vector Length Agnostic (VLA)
(VLA), 27
VHDL (very high-speed integrated circuit hardware description language)
(very high-speed integrated circuit hardware), 64

CHAPTER I.

INTRODUCTION

This chapter aims to introduce the triple arithmetic logic unit (Tri-ALU), followed by stating and explaining the problem statement of this research. Finally, it discusses the research provided and this work's contribution.

1.1 Basic ALU Architectures

In a computer system, the arithmetic logic unit (ALU) is a critical component in the central processing unit (CPU). ALUs in contemporary CPUs are quite strong and sophisticated. CPUs also include ALUs and a control unit (CU), which uses control signals to run the ALU. These signals direct the ALU's actions, and the ALU records the outcomes of those operations in output registers. Moreover, the CU uses the control signals to transfer data between these registers, the ALU, and the memory [1]. The ALU is employed for executing the arithmetic and logic operations specified in a program. Additionally, it is utilized by the CPU when it needs to perform its own arithmetic tasks, such as adding two values to calculate a memory address [2].

The ALU, a crucial part of a computer's CPU, is in charge of executing arithmetic and logical operations on data. These operations comprise addition, subtraction, multiplication, division, AND, OR, and NOT logical operations. Conventional ALUs are built to perform these tasks effectively, enabling quick and precise data processing.

It should be noted that the majority of the ALU is entirely asynchronous, which means it has no direct impact on the primary system clock. Any modifications made to the ALU's

data, instruction, or carry-in inputs will start to affect the logic gates right away and eventually manifest themselves in the data and status output [3].

However, there is a growing need for more advanced ALUs that can handle more complex operations. This is where the Tri-ALU comes in. By adding three operations' execution Modes with hybrid architecture, the Tri-ALU can improve data processing efficiency and ultimately improve a computer system's overall performance.

The ALU's architecture can vary depending on the specific implementation and design choices of the CPU manufacturer. However, most ALUs have a basic structure that includes an arithmetic unit, a logical unit, and a CU.

The arithmetic unit performs arithmetic operations, such as addition and subtraction. The logical unit performs logical operations, such as AND, OR, and NOT. The CU manages the flow of data between the input and output registers and the arithmetic and logical units.

Overall, the ALU architecture is a critical component of the CPU, and its design and implementation can significantly impact the CPU's performance and efficiency (Figure 1).

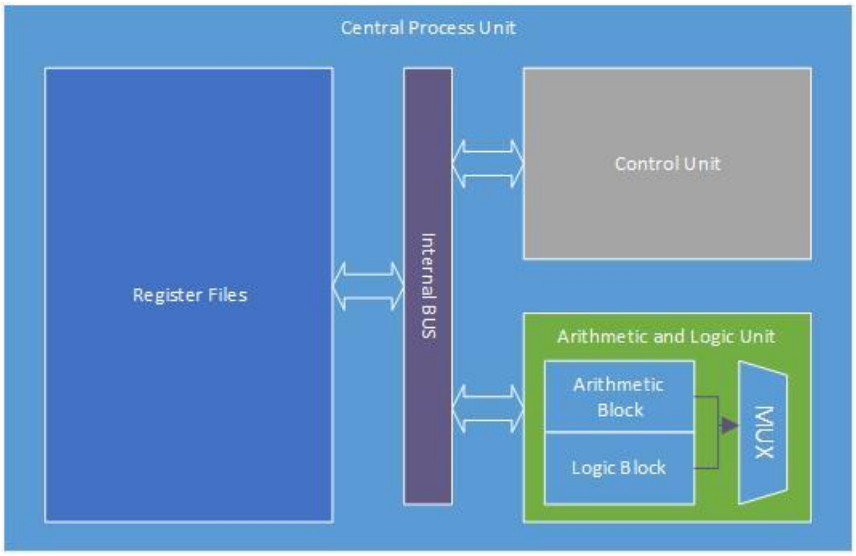


Figure 1: The ALU and other CPU components [4]

1.2 High-Speed ALU Architecture

High-speed ALUs can handle more than two operands and perform multiple operations in the same machine cycle. In the high-speed ALU architecture, the CPU can fetch fewer instructions from memory and complete the required task. In contrast, basic ALU architecture needs to push the CU to fetch more instructions from memory, thus needing more time to complete the required task. Therefore, the ALU needs to address this problem by compressing instructions into a few compact instructions. However, this will not be accomplished until the high-speed ALU architecture handles more operands and executes multiple operations in a single run.

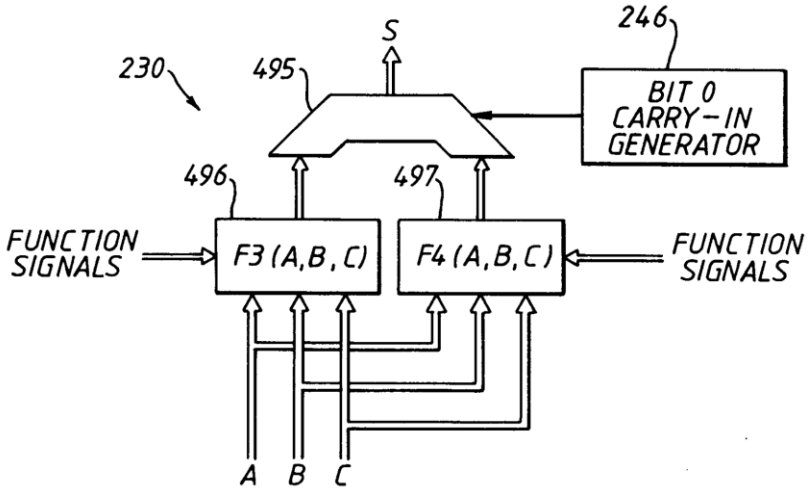


Figure 2: High-speed arithmetic logic units [5]

High-speed ALUs are used in a wide variety of applications, including:

1. Scientific computing: High-speed ALUs are essential for scientific computing applications, such as weather forecasting, fluid dynamics simulations, and medical imaging. These applications require the ability to perform complex calculations rapidly, and a high-speed ALU can provide a significant performance boost.
2. Graphics processing: High-speed ALUs are also used in graphics processing applications, such as video games and 3D rendering. These applications require the ability to perform many calculations regularly, and a high-speed ALU can help to improve the frame rate and overall performance.
3. Machine learning: High-speed ALUs are becoming increasingly important for machine learning applications. Machine learning algorithms often require the ability to perform many calculations on large datasets, and a high-speed ALU can help to improve the performance of these algorithms.

In addition to these specific applications, high-speed ALUs can also be used in diverse other applications that require fast and efficient arithmetic and logic operations.

Here are some of the benefits of using a high-speed ALU:

1. Improved performance: A high-speed ALU can significantly improve the performance of applications that require fast and efficient arithmetic and logic operations.
2. Reduced power consumption: A high-speed ALU can help to reduce power consumption by performing calculations more quickly. This can be a significant benefit for battery-powered devices, such as laptops and smartphones.

Overall, high-speed ALUs offer several benefits that make them a valuable tool for a wide variety of applications.

1.3 Purpose of the Study

This study aims to compare the effectiveness of the Tri-ALU compared to the basic ALU as an architecture for increasing data processing speed and efficiency.

1.4 Problem Statement

This study is concerned with the ability to perform multiple operations simultaneously. Current methods of using ALUs do not reduce access operations to memory due to their internal architecture, and performing one operation with one or two operands is inefficient for today's applications such as machine learning and AI algorithms.

In general, when accessing random memory, one should communicate the address of the memory one needs to read from or write to before receiving or reading any data. In our research, we introduce a new method to deal with tasks and operations effectively using the Tri-ALU and control signal to reduce access to the random-access memory (RAM).

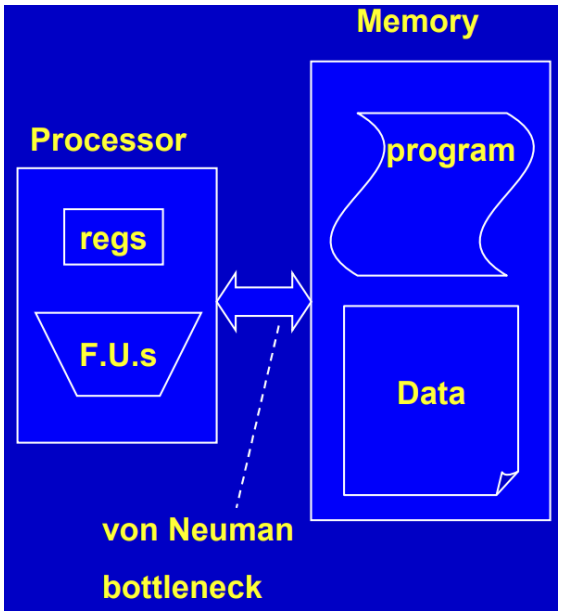


Figure 3: von Neuman bottleneck [6]

The importance of reducing access to memory has increased with advances in the performance of processors. Figure 4 plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency). The situation in a uniprocessor is somewhat worse since the peak memory access rate is faster than the average rate, which is what is plotted [7].

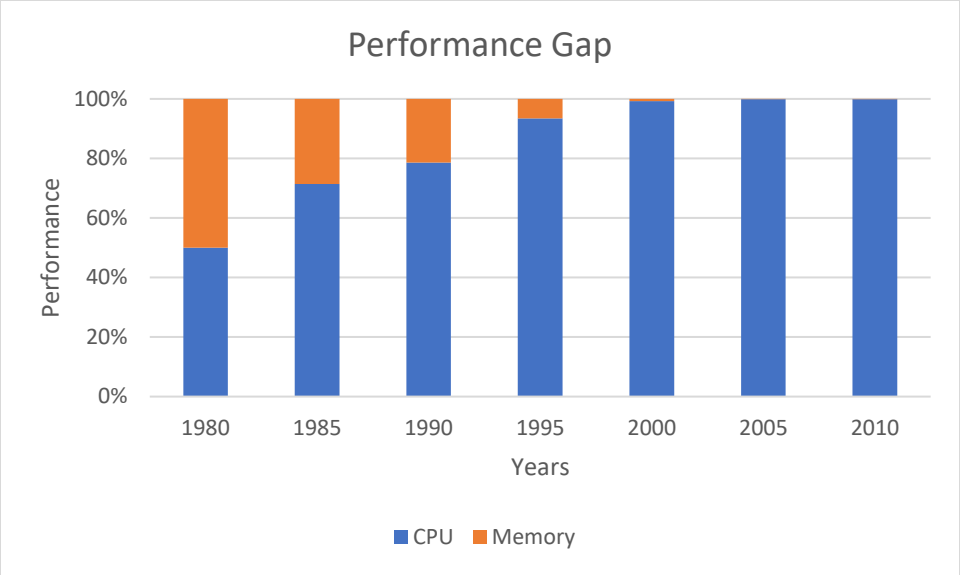


Figure 4: The performance gap, calculated as the difference in time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is depicted across time starting with the performance in 1980 as a baseline [2].

Therefore, reducing access to memory will enhance the system performance and increase computation throughput.

Another critical factor is power consumption and cost. Low-power microcontrollers are well-liked for intelligent edge node applications, especially those based on TinyML, because of their adaptability. Several voice assistant-based programs execute interference using constant cloud connectivity, only locally detecting a trigger word or brief phrase. However, this method creates delay and increases the possibility of device vulnerability. A top priority is the requirement for local, deterministic decision-making. Koedel pointed to Infineon MCUs that incorporate accelerators for automotive applications, such as graphical displays and ADAS radar processing, and noted that most microcontroller suppliers concentrate on integrating neural network accelerators into their MCUs.

Microchip's Thomsen called AI a game changer for MCUs involved in real-time closed-loop control. "I think AI is probably the big revolutionary change for the MCU, and in a lot of cases, it's going to be a revolutionary change for our customers' applications," Thomsen said [8]. The microcontrollers need new ALU architectures that address new AI application requirements with low-cost and simplest circuits, not like vector processors or complex AI accelerators. The embedded system and IoT device platform are limited in parallel processing and memory size in addition to the demand for privacy and low latency; therefore, cloud AI is not an option in some applications, such as real-time defect detection [9]. Hence, The embedded system and IoT needs something in the middle that can perform and execute neural network operations better than conventional ALUs with low cost and consume less power than accelerators.

In summary, the background problem of this research lies in the increasing computing demands, memory access bottlenecks, software size and complexity, limited ALU capabilities, challenges in performance optimization, and the growing importance of specialized applications. Addressing these challenges is crucial to enhance computing systems' efficiency, performance, and versatility in the face of evolving technological requirements.

1.5 Research Motivation

The motivation behind this research stems from the fact that current ALUs do not provide high-speed data processing. The software complexity and large computation process impact computer performance and throughput. The old ALU design does not tackle future computing problems, such as neural networks and machine learning applications.

Therefore, a new design and approach must be developed to meet system requirements. By handling multiple operands and performing multiple arithmetic and logic operations with execution modes besides hybrid architecture, the Tri-ALU is hoped to increase system efficiency.

1.6 Proposed Solution

The Tri-ALU extends the capabilities of conventional ALUs by integrating multiple ALU architectures and incorporating three additional operations with execution Modes. By introducing this novel ALU, we aim to achieve more efficient data processing and ultimately improve computer systems' overall performance.

1.6 Research Questions

This thesis provides a systematic review of the ALU architecture. The areas of interest are the current state of ALU design and identifying the most significant gaps and limitations in the reviewed studies. The three objectives of this research are broken down into three research questions:

1- What is the current state of ALU architectures?

2- What are the most significant gaps and limitations in the reviewed studies?

3- How can the Tri-ALU reduce access to memory, increase operation efficiency, and decrease power consumption?

1.7 Research Hypothesis

1- The current state of ALU architectures is not efficient for today's applications, such as machine learning and AI algorithms. This is because the current ALU architecture is not designed to handle the increasing computing demands of these applications.

2- The most significant gaps and limitations in the reviewed studies are that they do not address the following:

1. The need for ALUs to perform multiple operations simultaneously.
2. The need for ALUs to reduce access operations to memory.
3. The need for ALUs to handle complex operations with fewer instructions.

3- The Tri-ALU can reduce access to memory, increase operation efficiency, and decrease power consumption by:

1. Performing multiple operations simultaneously.
2. Reducing access operations to memory.
3. Performing complex operations with fewer instructions.

1.8 Research Contribution

The Tri-ALU has several benefits over traditional ALUs. The addition of the three operations (two arithmetic, one index) allows for more efficient data processing. For example, in matrix multiplication, the Tri-ALU can perform two arithmetic operations (multiplication and addition) and produce the result without the need for additional instructions. This can significantly speed up the matrix multiplication and improve the

overall performance of the algorithms or machine learning models, such as neural networks.

Another benefit of the Tri-ALU is its ability to handle data routing automatically using Switching Modes. The Tri-ALU can perform multiple routings in a single instruction, which can improve a computer system's efficiency. This contributes to answering the previous questions and leads to better performance.

The Tri-ALU is based on a hybrid architecture to employ and utilize the advantage of (stack, accumulator, register, and vector) architectures.

The Tri-ALU can be used in a variety of applications, such as scientific computing, AI, and big data analysis. Its improved efficiency can also be beneficial in embedded systems and mobile devices, where power consumption is a critical factor.

The contributions of this work are listed as follows:

1. Enhanced Performance and Efficiency: The Tri-ALU allows for the simultaneous execution of multiple operations.
2. Multiple Operation Types: The Tri-ALU can perform arithmetic, logic, indexing, and shifting operations simultaneously.
3. Execution Modes: The Tri-ALU utilizes execution modes for automatic data routing.
4. Improved Data Processing: The Tri-ALU improves efficiency in tasks such as matrix multiplication by performing multiple operations without additional instructions.

5. Automatic Data Routing: The Tri-ALU handles data routing automatically, reducing the need for explicit routing instructions.
6. Hybrid Architecture: The Tri-ALU is based on a hybrid architecture, combining the advantages of different ALU types.

1.9 Thesis Layout

The rest of the thesis is organized as follows. Chapter two reviews the works proposed previously in the domain of ALU architecture. Chapter three provides the methodology of the proposed Tri-ALU in detail. Chapter four describes the system design and Tri-ALU application and explains the Tri-ALU performance through applications and benchmarking the Tri-ALU with other ALU architectures. Chapter five concludes the thesis, explains the limitations of this work, and proposes future work.

CHAPTER II.

LITERATURE REVIEW

This chapter presents an overview of some of the approaches proposed previously in the domain of ALU design. The chapter is organized so that it depends on the taxonomy of approaches as described below.

2.1 Related Works

The ALU architecture field has been a topic of interest for computer engineers and researchers for several decades. Researchers have worked on improving the performance and efficiency of ALUs to meet the ever-increasing demands of modern computing. The related work in ALU architecture covers a broad range of topics, including design techniques, optimization strategies, power management, and reliability issues.

Overall, the related work in ALU architecture is a continuously evolving field, and the research conducted in this area has made significant contributions to improving CPUs' performance and efficiency.

2.1.1 Conventional ALU

Related to the Tri-ALU is a Tri-ALU designed using four multiplexers to select multiple operations. Usually, in a conventional ALU, there is one multiplexer to select an operation, and the basic ALU can retrieve only two operand registers, A and B.

An instruction word, also known as a machine instruction word, makes up the input. It includes one or more operands, a format code on occasion, and an operation code, sometimes called operation code or "opcode." The operation uses operands, and the

operation code specifies to the ALU the operation to execute. Two operands might, for instance, be logically compared or added together. It identifies things like whether an instruction is fixed point or floating point and can be blended with the format and opcode. The result is written to a storage register along with settings that indicate whether the operation was successful. If not, the so-called machine status word will be permanently updated with some sort of status [10] (Figure 5).

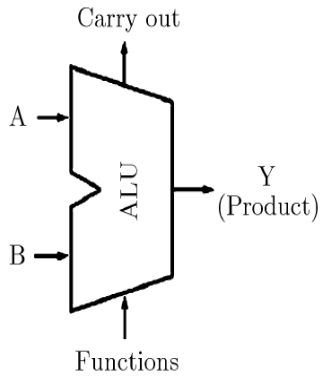


Figure 5: General form of an ALU

The following configurations exist in each ALU:

ACCUMULATOR: The accumulator consists of the intermediate results of each operation. The instruction set architecture (ISA) is less complex since only one bit has to be saved (two bits on other devices). It would not be required to save the destination as well. Although they are simpler and frequently execute extremely rapidly, extra algorithms must be created to fill the accumulator with the proper values and boost stability (Figure 6).

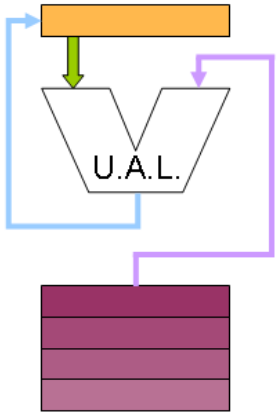


Figure 6: ALU accumulator structure [11]

STACK: The stack always contains the most recently finished action. It is a minor register. The programs it stores are arranged top-down. When fresh instructions are received, they compact to insert the old ones (Figure 7).

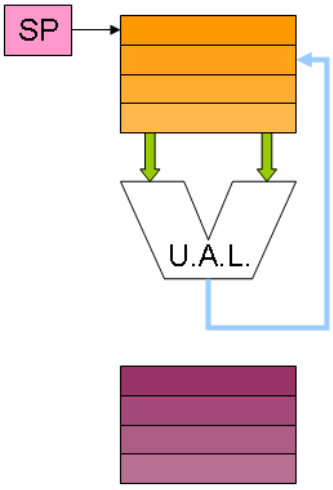


Figure 7: ALU stack structure [11]

REGISTER-REGISTER ARCHITECTURE: A three-register operating machine is another name for it. It has two source instructions and one destination instruction. This ISA must be longer to contain three operands (two sources and one destination). When the procedures are finished, it would be difficult to write the findings back into the registers; thus, the word

count should be raised. The write-back rule would then be implemented here, increasing the synchronization issues (Figure 8).

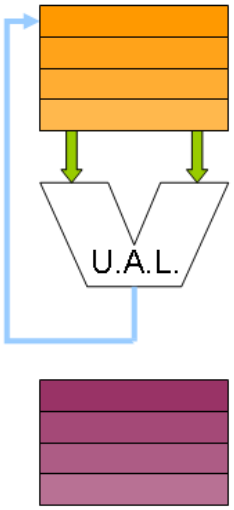


Figure 8: ALU register structure [11]

REGISTER–STACK ARCHITECTURE: Usually, it combines operations from the accumulator and register. In the register-stack design, the necessary actions are pushed to the top of the stack, where they are also stored.

REGISTER AND MEMORY: One of the trickiest architectural concepts. This has two operands, one from the register and the other from external memory. Because each instruction must be saved in its entirety in a memory area that could be rather vast, it is difficult. However, this technology cannot be used alone; it is linked to three-register technology. CICC architecture stores instruction words in this way rather than employing whole memory sections for that purpose [12], [13], [14], [11] (Figure 9).

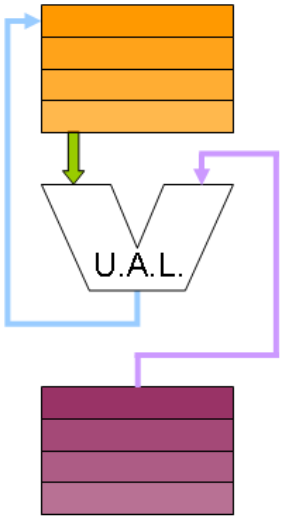


Figure 9: Register and memory structure [11]

The main disadvantages of a conventional ALU compared to a Tri-ALU are as follows:

Performance: A conventional ALU can only perform one operation at a time, while a Tri-ALU can perform four operations simultaneously. This can lead to a significant performance improvement for applications that require a high degree of parallelism.

Flexibility: A conventional ALU can only perform a limited number of operations, while a Tri-ALU can perform a wider variety of operations. This can make the Tri-ALU more versatile for different applications.

Efficiency: A conventional ALU may not be as efficient in terms of power consumption and performance as a Tri-ALU. This is because a conventional ALU must perform a series of steps to perform each operation, while a Tri-ALU can perform multiple operations simultaneously.

The advantages of the Tri-ALU design are as follows:

Parallelism: The Tri-ALU can perform four operations simultaneously, which can improve performance for certain workloads.

Efficiency: The Tri-ALU can be more efficient in terms of power consumption and performance than other ALU designs. This is because it can perform multiple operations simultaneously, which reduces the amount of time it takes to complete a task.

Flexibility: The Tri-ALU can perform a wider variety of operations than other ALU designs.

2.2.1 Non-Conventional ALU

Researchers have explored novel applications of ALUs, such as in machine learning and AI. The use of specialized ALUs designed specifically for neural network processing has shown promising results in improving the performance and efficiency of deep learning algorithms. Moreover, researchers have explored various design techniques, such as pipelining, parallelism, and vector processing, to enhance the ALU's performance. Furthermore, optimization strategies, such as algorithmic improvements and compiler optimizations, have been proposed to reduce the power consumption of ALUs.

2.2.2 Neural Network of Ternary Arithmetic Logic Unit

In Ref. [15], the neural network ternary arithmetic logic unit (NN-TALU) uses multiple cascades of ALU. 1-trit TALU can accept two operands and perform one operation at a time. The TALU performs an operation, and the result of this operation should be transferred to a destination ternary register (Figure 10).

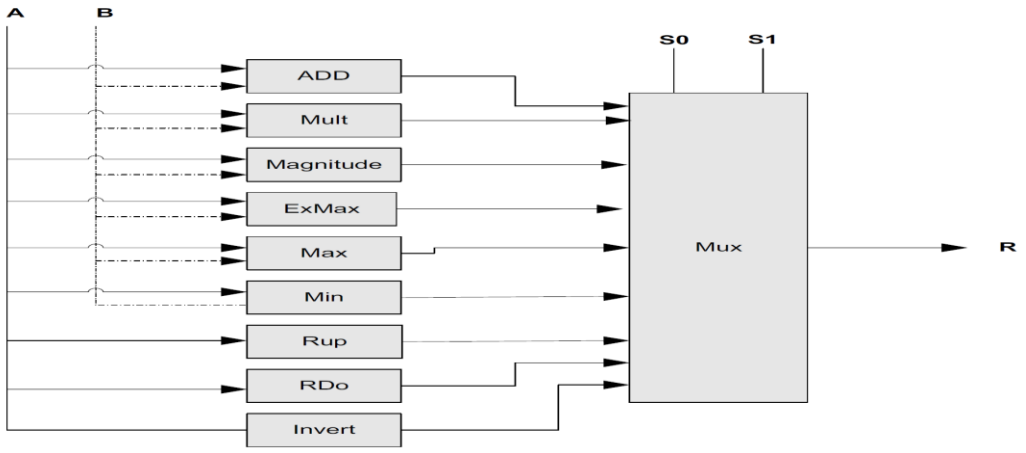


Figure 10: Block diagram of the TALU

The main disadvantage of the NN-TALU design is that it uses multiple cascades of ALUs, which can lead to increased latency and power consumption. Each cascade of ALUs must perform a series of operations before the result can be transferred to a destination register. This can add significant overhead, especially for complex operations. Additionally, the multiple cascades of ALUs can increase the overall size and complexity of the NN-TALU design.

On the other hand, the Tri-ALU design uses a single ALU that can perform multiple operations simultaneously. This can significantly reduce latency and power consumption, as well as the design's overall size and complexity. Additionally, the Tri-ALU design is more flexible than the NN-TALU design, as it can be used to perform a wider variety of operations.

Ultimately, the best design for a particular application will depend on the specific requirements of that application. For applications that require high performance and low power consumption, the Tri-ALU design may be a better choice. For applications that require a wider range of functionality, the NN-TALU design may be a better choice.

2.2.3 The Arithmetic Logic Unit Based on Adaptive Logic Module (ALM) Architecture

In Ref. [16], the ALU is based on adaptive logic module (ALM) architecture. The ALM, which is intended to maximize performance and resource use, is the fundamental component of the supported device families (Arria® series, Cyclone® V, Stratix® IV, and Stratix® V). Each ALM has two or four register logic cells (lc ff) and two combinational logic cells (lc comb), two dedicated full adders, a carry chain, a register chain, and a 64-bit LUT mask. It can accommodate up to eight inputs and eight outputs [17]. The ALU comprises a two-stage pipelined design. The first stage is to complete the instruction operations and register the corresponding results, and the second stage is to output the results according to the instruction opcode (Figure 11).

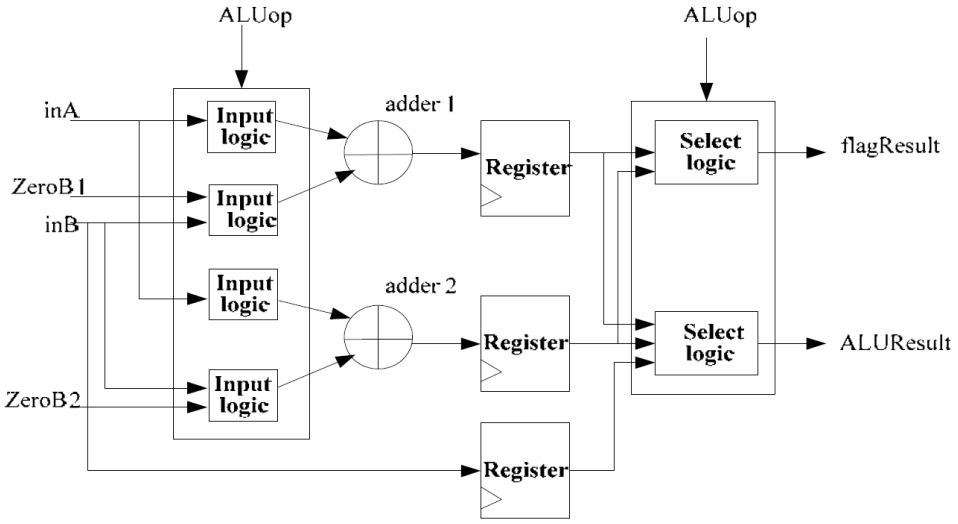


Figure 11: The architecture of ALU design

The main disadvantage of the ALM architecture is that it is more complex than the Tri-ALU architecture. This complexity can make the design and implementation of the ALM

architecture more difficult and expensive. The ALM architecture also has a lower throughput than the Tri-ALU architecture. This is because the ALM architecture has a two-stage pipelined design, which can introduce some latency.

The advantages of the Tri-ALU architecture are as follows:

Simplicity: The Tri-ALU architecture is simpler than the ALM architecture. This simplicity can make the design and implementation of the Tri-ALU architecture easier and less expensive.

Throughput: The Tri-ALU architecture has a higher throughput than the ALM architecture. This is because the Tri-ALU architecture does not have a pipelined design, which can reduce latency.

Flexibility: The Tri-ALU architecture is less flexible than the ALM architecture. This is because the Tri-ALU architecture can only perform a limited number of operations.

The disadvantages of the ALM architecture are as follows:

Complexity: The ALM architecture is more complex than the Tri-ALU architecture. This complexity can make the design and implementation of the ALM architecture more difficult and expensive.

Latency: The ALM architecture has a lower throughput than the Tri-ALU architecture. This is because the ALM architecture has a pipelined design, which can introduce some latency.

2.2.4 Pipelining of Floating-point ALU

Ref. [18] aimed to implement pipelining in the design of the floating-point ALU using VHDL. Designing a 16-bit floating-point ALU that adheres to IEEE 754 was one of the sub-objectives. The four fundamental arithmetic operations of addition, subtraction, multiplication, and division are supported by floating-point representations (Figure 12).

The main disadvantage of pipelining a floating-point ALU is that it can increase the ALU's latency. This is because the pipeline stages must be synchronized, and any delay in one stage can delay all the stages. Additionally, pipelining can increase the complexity of the ALU design, as it must be designed to handle the different stages of the pipeline.

On the other hand, the Tri-ALU design does not use pipelining. This means that the Tri-ALU can perform operations more quickly than a pipelined ALU.

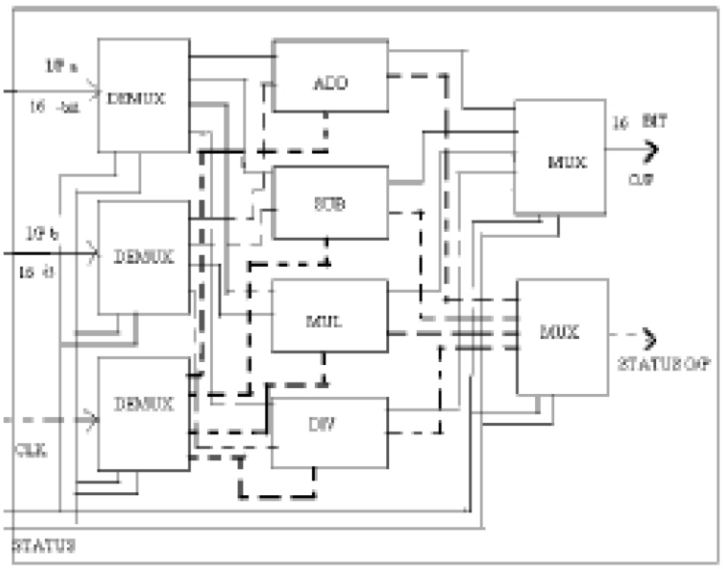


Figure 12: Top-level view of the ALU design

2.2.5 Multiplier and Accumulator Unit (MAC)

A crucial component of computing equipment, particularly digital signal processors, is the multiplier and accumulator (MAC) unit, which performs multiplication and accumulation processes. Multiplier, adder, and accumulator make up the basic MAC unit [19] (Figure 13).

In digital signal processing (DSP), a MAC unit is a functional unit in a processor that performs a multiplication operation between two numbers and adds the result to an accumulated value in a single clock cycle. This operation is fundamental to many DSP algorithms and is widely used in applications such as filtering, convolution, and signal analysis.

The MAC unit is designed to perform this operation efficiently and with a high degree of parallelism, allowing it to execute many computations in parallel. This is crucial in many DSP applications where large amounts of data need to be processed in real time, such as in audio and video processing.

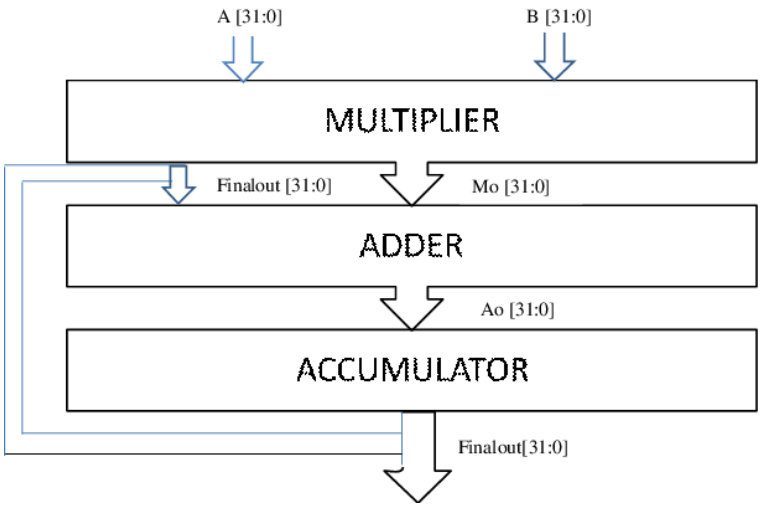


Figure 13: Structure of a Mac unit [15]

A Tri-ALU and a MAC unit serve different purposes and have distinct advantages in their respective domains.

Advantages of a Tri-ALU:

Versatility: A Tri-ALU can handle a broader range of operations compared to a MAC unit. It can perform not only multiplication and accumulation but also various arithmetic, logic, indexing, and shifting operations. This versatility makes it suitable for applications beyond DSP.

Parallel processing: A Tri-ALU allows for the execution of multiple operations simultaneously, which enables parallel processing and can significantly enhance overall computational performance. This is particularly advantageous when dealing with complex algorithms or large-scale computations.

Customizability: The architecture of a Tri-ALU can be tailored to specific software requirements. It supports different ALU types, such as stack, accumulator, registers, and vector architectures, offering flexibility and adaptability to different computational needs.

2.2.6 Three-Input Arithmetic Logic Unit Forming the Sum of a First and Second Boolean Combination of the Inputs

A three-input ALU forms a mixed arithmetic and Boolean combination of three multibit input signals. The ALU first forms a Boolean combination and then forms an arithmetic combination. The current instruction drives an instruction decoder that generates the function signals that control the combination formed. The three-input ALU preferably employs a set of bit circuits, each forming carry, propagate, generate, and kill signals. These signals may be employed with a multilevel logic tree circuit and a carry input to produce a bit resultant and a carry output to the next bit circuit. This structure permits the formation of selected arithmetic, Boolean, or mixed arithmetic and Boolean functions of the three input signals based on the current instruction. Selecting the function signals enables the combination to be insensitive to one of the input signals, thus performing a two-input function of the remaining input signals. The instruction itself may include the function signals and function modification bits, or the function signals and function modification signals may be stored in a special data register. Function modification signals cause the modification of the function signals prior to use. The three-input ALU includes a least significant bit carry-in generator supplying a carry input to the least significant bit. This carry input is determined by the combination being formed and is generally "1" only during subtraction. The carry input may be specified in the special purpose data register

(DO) for CM certain instructions. The combination formed is optionally modified dependent on the sign bit of one of the inputs [5] (Figure 14).

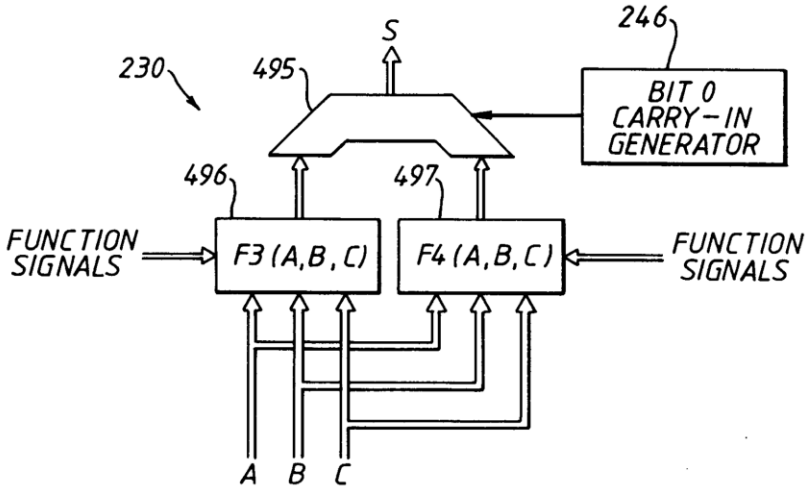


Figure 14: Three-input arithmetic logic unit

Advantages of a Tri-ALU over a three-input ALU:

Hybrid design and performance enhancement: A Tri-ALU combines the advantages of multiple ALU types (e.g., stack, accumulator, registers, and vector) to enhance performance and efficiency based on software requirements.

Simultaneous operations: A Tri-ALU can perform four operations simultaneously, including arithmetic, logic, counter, and shift operations.

Multiple operands and registers: A Tri-ALU receives three operands and stores them in registers (A, B, feedback) for efficient processing. The feedback register is dedicated to indexing and loop operations.

Efficiency and speed: By allowing multiple operations with different data to be performed simultaneously, a Tri-ALU aims to increase a computer system’s efficiency and processing speed.

Execution Modes: A TRI-ALU can perform operations automatically for specific tasks, such as automatic switch results between registers, and iterate operations until the feedback register reaches zero.

2.2.7 Scalable Vector Extension (SVE)

The scalable vector extension (SVE) is the next-generation single instruction, multiple data (SIMD) extension of the Arm®v8-A AArch64 instruction set. The SVE is a new set of vector instructions created with HPC workloads in mind; it is not an expansion of Neon. The SVE makes it feasible to vectorize loops that Neon® either could not or would not be useful in vectorizing.

SVE can be vector length agnostic(VLA), unlike other SIMD designs. The vector registers' sizes are not fixed by the SVE, allowing hardware developers to select the size that best suits their workloads [20] (Figure 15).

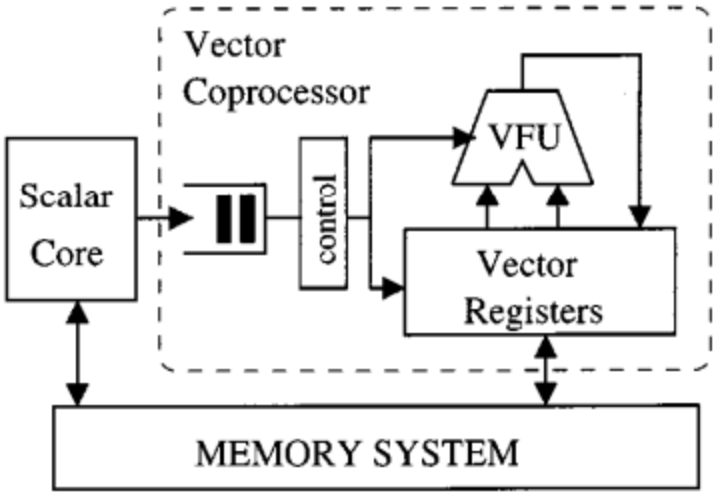


Figure 15: Simplified view of a vector processor with one functional unit for arithmetic operations (VFU). The functional unit contains a single execution datapath [18].

The disadvantages of the SVE are as follows:

Limited support: The SVE is not as widely supported as other SIMD designs, such as Neon. This means that applications that use the SVE may not be able to run on all hardware platforms.

Complexity: The SVE can be more complex to program than other SIMD designs. This is because the SVE uses a different instruction set than Neon.

The advantages of the Tri-ALU design:

Parallelism: The Tri-ALU can perform four operations simultaneously, which can improve performance for certain workloads.

Efficiency: The Tri-ALU can be more efficient in terms of power consumption and performance than other ALU designs. This is because it can perform multiple operations simultaneously, which reduces the time it takes to complete a task.

Flexibility: The Tri-ALU can be more flexible than other ALU designs. This is because it can perform a wider variety of operations.

2.2.8 Run-Time Customization of a soft-core CPU on an FPGA

This report discusses the increasing use of customized soft-core processors integrated into field-programmable gate arrays (FPGAs) and the benefits of partial run-time reconfiguration in specialized processors. The focus is on customizing a soft-core MIPS processor using an FPGA and partial reconfiguration to optimize resource utilization. This report proposes a design flow that enables the design to fit into a smaller device and reduce static power consumption through run-time reconfiguration. The project aimed to investigate the use of partial reconfiguration in FPGAs for adapting the instruction set of a soft-core CPU, including integrating custom instructions and exploring the potential of the MultiBoot feature in Xilinx FPGAs for this purpose. The system was evaluated on a Nexus 3 development board with a Xilinx Spartan-6 FPGA. The system allows for the dynamic loading of reconfigurable custom instructions into user programs using the trap handler when the custom instruction is called by the MIPS CPU. The experiment demonstrates that custom instructions implemented in hardware can accelerate specific functions and reduce the number of instructions compared to software implementations. The report concluded that implementing custom instructions in hardware is feasible and worthwhile [21].

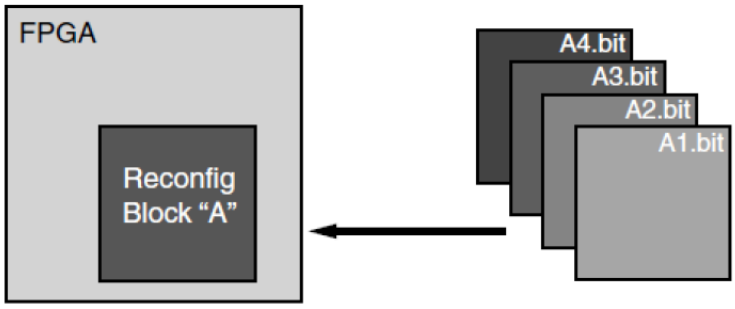


Figure 16: System overview of the reconfigurable region and static region

Run-time reconfiguration of an ALU can be disadvantageous because it can lead to the following:

Increased latency: When the ALU is reconfigured, there is a period during which it cannot perform any operations. This can lead to increased latency for applications that require a high degree of parallelism.

Reduced throughput: When the ALU is reconfigured, it cannot operate at its full capacity. This can lead to reduced throughput for applications that require many operations to be performed.

Increased complexity: Run-time reconfiguration of an ALU can add complexity to the design of the ALU and the system that it is used in. This can make the system's design and implementation more difficult and expensive.

The Tri-ALU design can overcome these disadvantages by:

Performing multiple operations simultaneously: The Tri-ALU can perform multiple operations simultaneously, which can help to mitigate the effects of increased latency and reduced throughput caused by run-time reconfiguration.

Being more efficient: The Tri-ALU is more efficient than a reconfigurable ALU, which can help to offset the design's increased complexity.

2.3 Integrating Tri-ALU Inside Pipeline and Superscalar Processor

Pipelining is an implementation technique that enables the overlapping execution of multiple instructions [22].

In a digital computer, executing an instruction involves four steps:

1. Instruction fetch (IF): The instruction is fetched from memory.
2. Instruction decoding (ID): The instruction is decoded to determine what operation needs to be performed.
3. Operand fetch (OF): If needed, the operands for the operation are fetched from memory.
4. Execution (EX): The operation is executed, and the result is stored in a register.

In a non-pipelined computer, these four steps must be completed before the next instruction can be executed. This means that the computer can only execute one instruction at a time. In a pipelined computer, the four steps are overlapped so that multiple instructions can be executed simultaneously. This allows the computer to execute instructions much faster than a non-pipelined computer [23].

2.3.1 The MIPS R4000 Instruction Pipeline

The MIPS R4000, a 64-bit processor, utilized separate instruction and data caches and featured an eight-stage pipeline to execute register-based instructions. The pipeline design was specifically aimed at achieving a high execution rate of nearly one instruction per cycle [24].

However, the Tri-ALU can replace the basic ALU in R4000 to increase data processing throughput and add another level of instruction-level parallelism ILP (Figure 17).

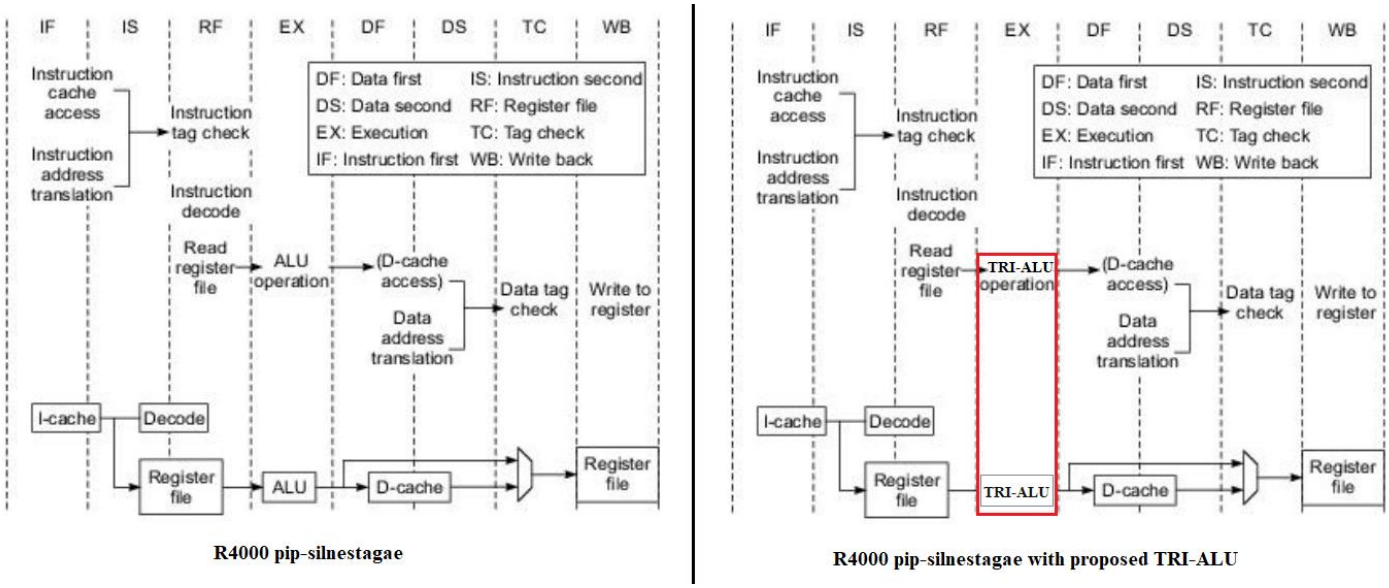


Figure 17: R4000 pipeline stage

2.3.2 RISC-V Vector Processing Pipelining

The vector processing unit can configurr several functional units, such as the load/store unit, multiplier, and adder (ALU), to be chained together [25].

Consider these simple operations:

```
# RISC-V Vector processing
VLE32.V   v1, (x1)     # v1 ← memory[x1]
VFMUL.VV  v3, v1, v2   # v3 ← v1 * v2
VFADD.VV  v5, v3, v4   # v5 ← v3 + v4
```

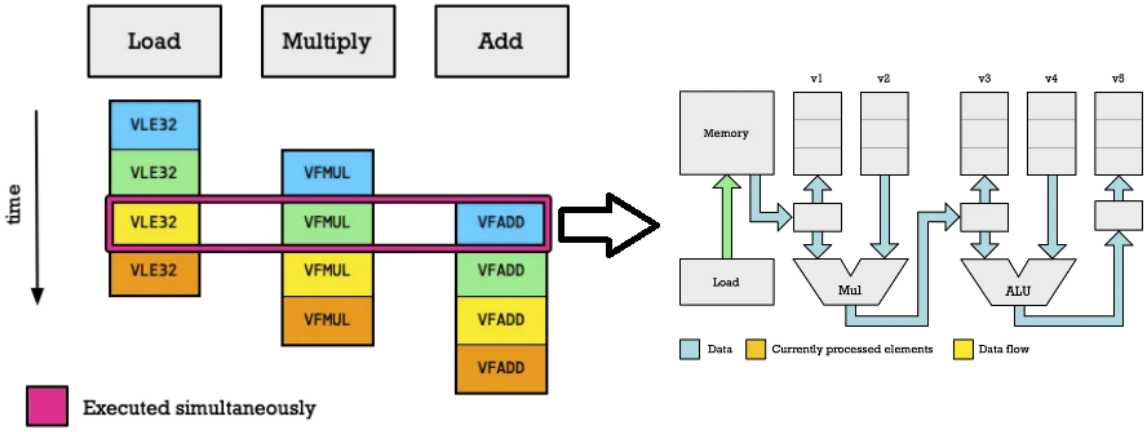


Figure 18: Integrating a Tri-ALU instead of two ALUs that perform MUL and ADD operations separately

```
# TRI-ALU
VLE32.V   v1, (x1)     # v1 ← memory[x1]
TRI   v1 * v2 + v4 , v5
```

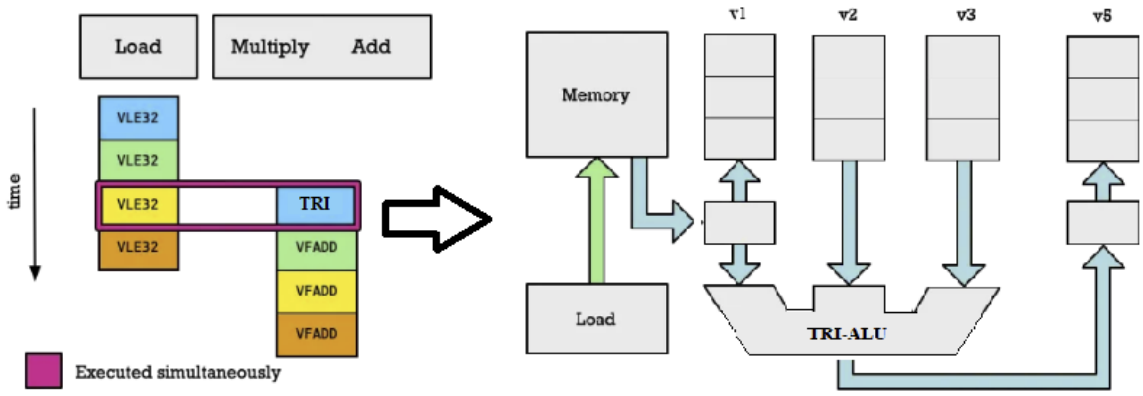


Figure 19: TRI-ALU will reduce clock requirements and instructions for operations and increase data processing throughput.

Note: TRI-ALU internal operations are asynchronies with system clock.

2.3.3 Superscalar Processor

A superscalar processor is a type of processor that can execute multiple instructions simultaneously. This is done by exploiting ILP, which is the ability of a program to have multiple instructions that can be executed independently. The instruction issue degree (m) of a superscalar processor is the maximum number of instructions that it can execute at the same time. The formula for m is $2 < m < 5$.

This means that a superscalar processor can issue at least two instructions per cycle and, at most, five instructions per cycle. In practice, most superscalar processors have an instruction issue degree of 3 or 4 [26]. We show a superscalar processor of degree $m = 3$ in Figure 20.

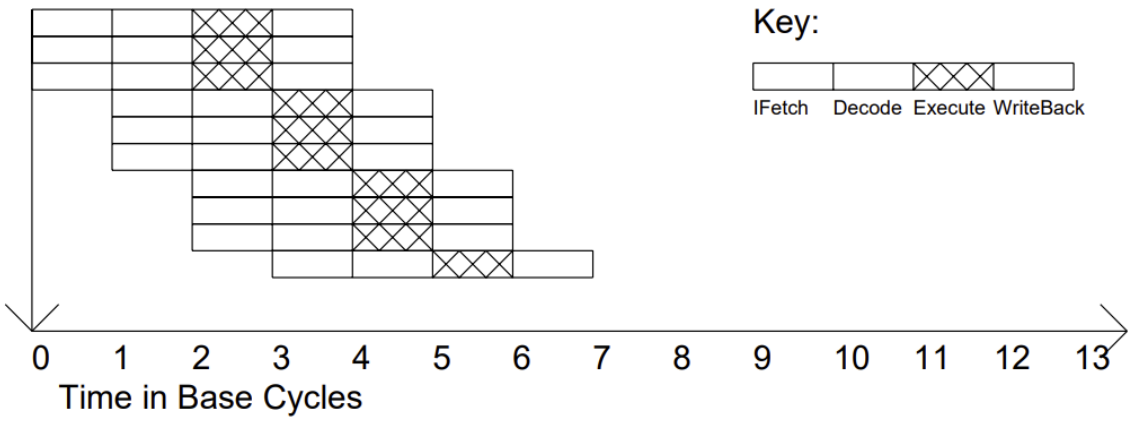


Figure 20: Execution in a superscalar machine (n = 3) (Adopted) [27]

The Tri-ALU can replace a simple ALU to increase ILP in a superscalar processor (Figure 21).

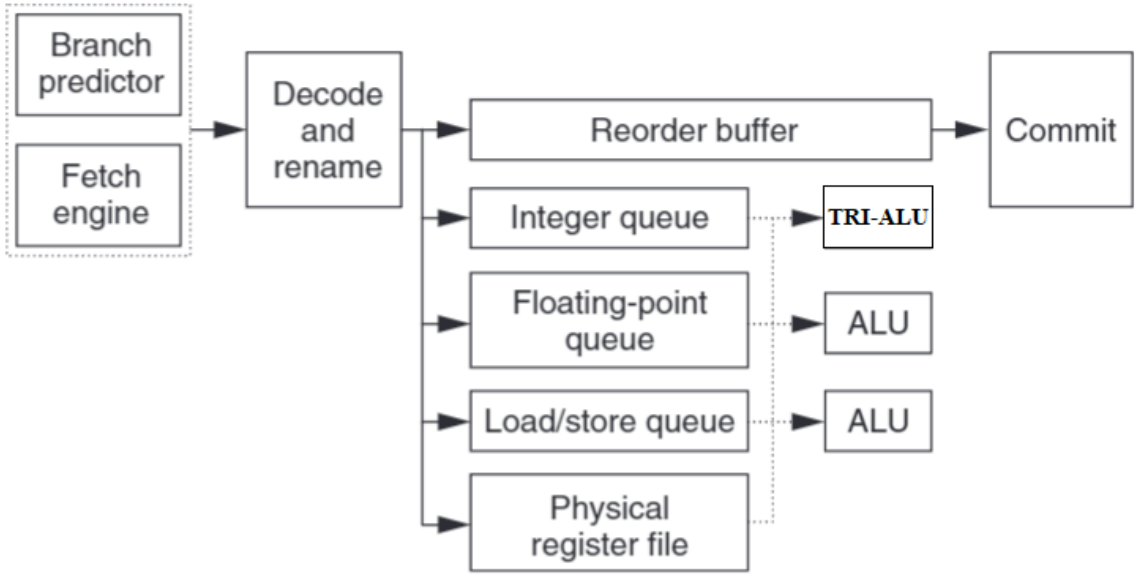


Figure 21: Tri-ALU inside a superscalar processor (Adopted) [28]

2.3.4 Very Long Instruction Word (VLIW) Processor and Superscalar Processors

In the very long instruction word (VLIW) design, multiple instructions are grouped together within a single word. The compiler plays a crucial role in constructing the VLIW by organizing operations that can be executed in parallel and placing them within the same word. In instances where it is not feasible to completely fill the word with instructions to be issued in parallel, no-ops are utilized. This ensures that the word remains occupied and the VLIW machine functions properly, even if some instructions are not actively executed [29].

The main challenge in designing superscalar processors is that it is difficult to dynamically schedule instructions to reduce pipeline delays. This is because the hardware can only look at a small window of instructions at a time, and it is difficult to consider all the dependencies between instructions.

On the other hand, compilers can take a more global view of the program and rearrange code to better utilize the resources and reduce pipeline delays. This is done by exposing a sequence of instructions that have no dependency and require different resources of the processor.

One way to do this is to use VLIWs. In a VLIW architecture, a single instruction word can contain multiple operations. For example, a VLIW could contain two integer operations, two floating-point operations, two load/store operations, and a branch.

The processor must have enough resources to execute all the operations specified in a VLIW simultaneously. This means that it must have multiple integer units, multiple floating-point units, multiple data memories, and a branch arithmetic unit.

The challenge with VLIW architectures is that it is difficult to find enough parallelism in a sequence of instructions to keep all the resources busy. This is because most programs do not have enough instructions that are independent and require different resources [30].

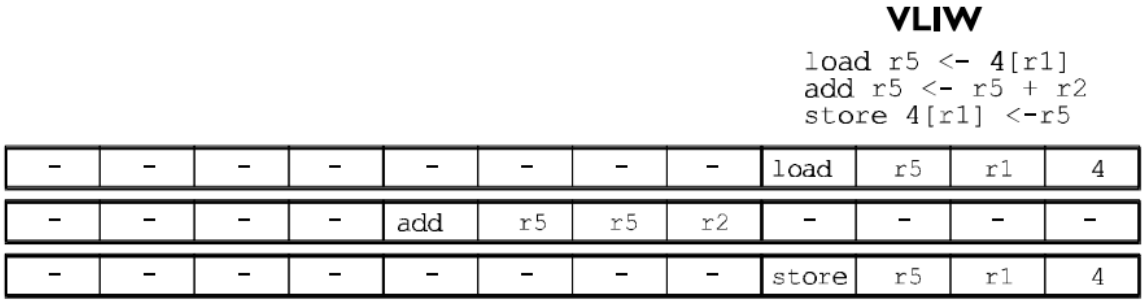


Figure 22: VLIW slots (Adopted) [29].

Given the code fragment presented, the VLIW exhibits a discouraging level of inefficiency in terms of code utilization. In a practical program scenario, the VLIW compiler would employ various optimization techniques to maximize the usage of all three slots in each of the three instructions. It is enlightening to consider the performance that each machine could potentially achieve when executing this code [31].

The Tri-ALU offers advantages over a VLIW processor in certain scenarios:

Dynamic adaptability: The Tri-ALU can dynamically adapt its execution modes and architectures based on software requirements, allowing for optimized performance and efficient hardware resource utilization. However, VLIW processors lack this adaptability and rely on explicit ILP extraction during compilation.

Reduced compiler complexity: The Tri-ALU simplifies the compilation process by inherently executing multiple operations simultaneously, eliminating the need for complex instruction scheduling and packing required by VLIW processors.

Hardware efficiency: The Tri-ALU achieves efficient hardware utilization by fully utilizing resources without explicit instruction scheduling, potentially leading to lower power consumption. However, VLIW processors may introduce additional hardware complexity and power inefficiencies due to explicit instruction scheduling requirements.

Improved performance in dynamic workloads: The Tri-ALU excels in dynamic workloads with varying execution behavior and dependencies between operations. Its ability to adapt and execute different operations simultaneously enables efficient processing of diverse computational requirements. VLIW processors are better suited for static workloads with predictable dependencies.

Flexibility in operation mix: The Tri-ALU can simultaneously execute a mix of arithmetic, logic, indexing, and shift operations, providing greater flexibility in handling different types of computations. However, VLIW processors have fixed instruction formats and limited flexibility in the variety of operations that can be executed simultaneously (Figure 23).

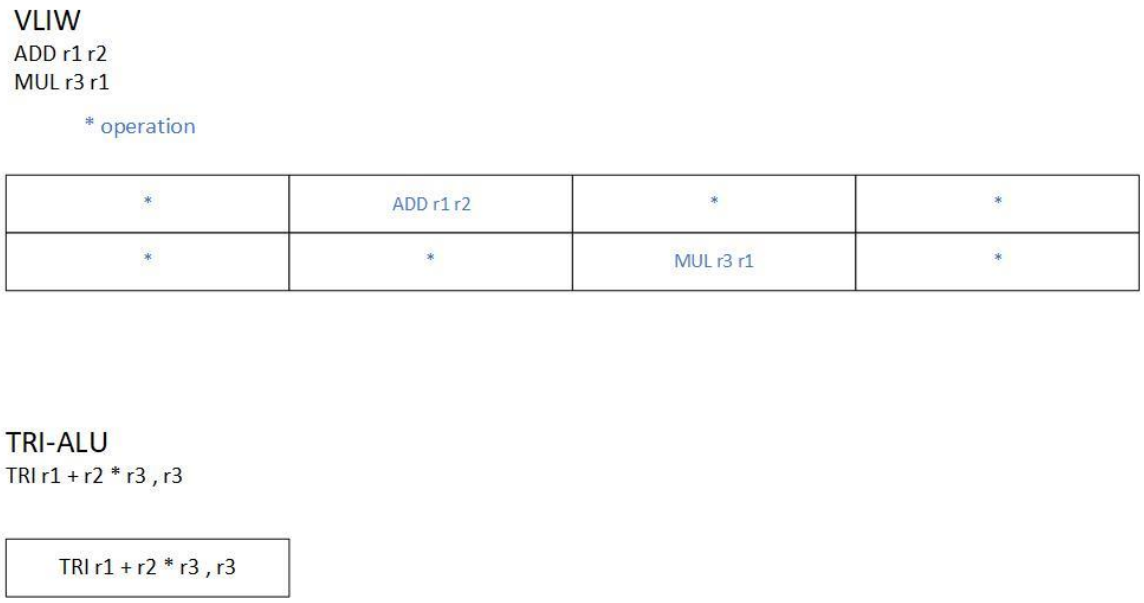


Figure 23: VLIW and Tri-ALU MAC operation

2.4 Current Design's Drawbacks and Limitations

The previous design architecture had several limitations. First, there is a lack of support for multiple operands, which is crucial for many computer programs that require more than two operands to perform a task. Additionally, it cannot combine arithmetic, logic, or shift operations within the same instruction. This means that operations such as addition and subtraction or addition and multiplication, among others, cannot be performed simultaneously. For instance, the expression A×B+C would be impossible to execute efficiently.

Second, there is a need for an index operation. Many algorithms heavily rely on looping operations, but the traditional ALU design lacks an inner loop operation.

Third, there are no execution modes available. Execution modes allow for automatic data routing through the ALU register and enable looping operations to perform specific tasks without the need for explicit instructions.

Furthermore, advanced algorithms require a resilient architecture, such as a hybrid architecture, which the current ALU cannot provide. Partial run-time reconfiguration adds to the logic overhead and increases the system's footprint.

Moreover, ILP scheduling in superscalar processors introduces overhead and necessitates complex hardware. On the other hand, VLIW processors have fixed instruction formats and limited flexibility, further constraining their capabilities.

2.5 Summary

The researcher listed and discussed the recent technology in ALU architectures. The current design’s drawbacks motivated the researchers to develop a new design. The next chapter explains our proposed ALU architecture, the Tri-ALU, in detail.

CHAPTER III.

METHODOLOGY

This chapter presents the main thesis contribution, which includes the methodology used for operator fusion and routing fusion, and hybrid architecture. In other words, the chapter starts by elaborating on the proposed approaches.

3.1 Basic ALU Limitation

Current ALU design limitations are as follows:

Limited parallelism: It performs only one operation at a time.

Lack of specialized operations: It has a limited set of arithmetic and logic operations.

Reduced flexibility: Its fixed structure and functionality limits its adaptability to different instruction sets and ALU architectures.

Limited operand handling: Basic ALU typically handles two operands and produces a single result.

Therefore, the researcher proposed a Tri-ALU to overcome these limitations.

3.2 Proposed System (Tri-ALU)

A Tri-ALU is a hybrid ALU that enhances performance and efficiency by combining the advantages of several ALU types. It acts as stack, accumulator, register, or vector architectures depending on software requirements. A Tri-ALU can perform four operations simultaneously. These operations typically include two arithmetic or logic operations (such as addition, subtraction, AND, and XOR), one indexing or counter operation (such as

decrement -1 or increment +1), and one shift operation employing execution modes to accomplish a specific function. A Tri-ALU receives three operands and saves them in registers (A, B, feedback). The feedback is a special register for indexing and loop operations. It can be used with the other two registers (A, B) for arithmetic operations. In contrast, a conventional ALU takes only two operands (A, B) and performs one operation at a time (Figure 13). The purpose of a Tri-ALU is to increase the efficiency and speed of a computer's processing by allowing multiple operations with different data to be performed simultaneously. Multiple instruction, multiple data (MIMD) and execution modes are used to automatically route data throw operations (Figure 24).

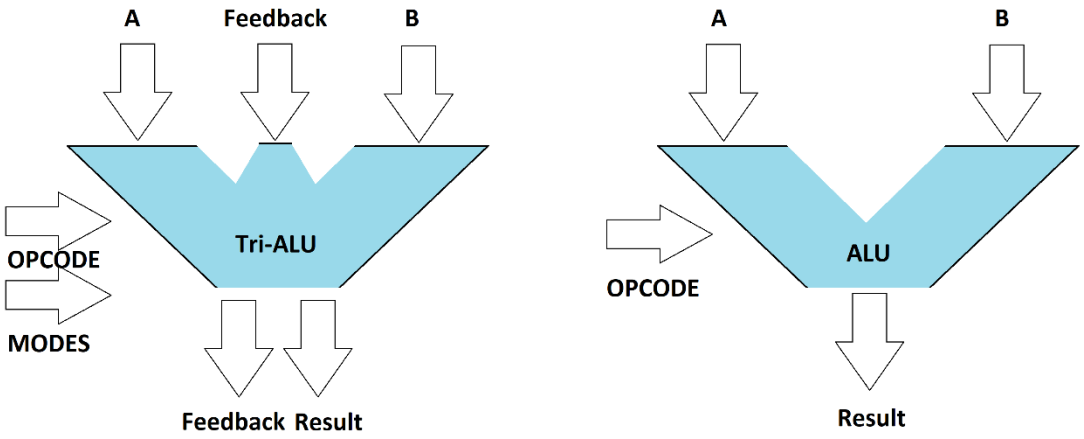


Figure 24: Tri-ALU vs. ALU [11]

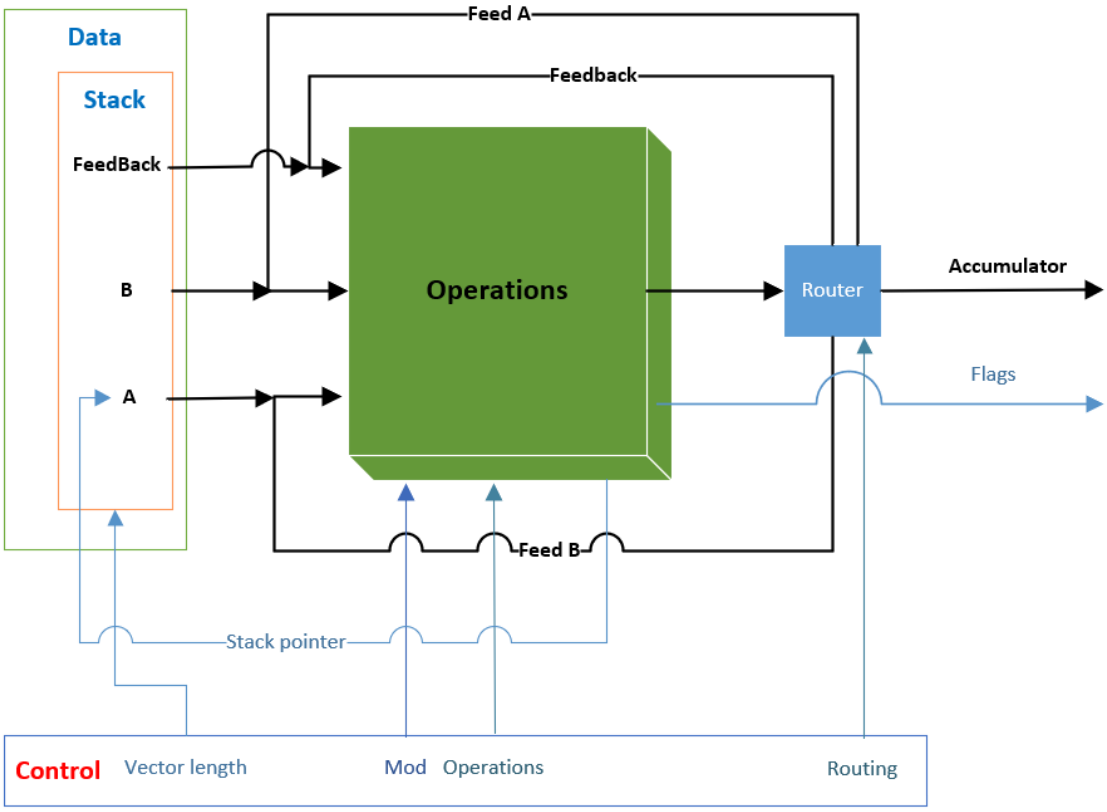


Figure 25: Tri-ALU top-level design

A, B, and feedback registers can act like a stack (Figure 25). Moreover, these registers support vector operations. A Tri-ALU enhances computing efficiency using several techniques. First, operator fusion is the primary optimization technique for the Tri-ALU. Operations are fused to calculate many functions in a single run rather than writing each intermediate result to memory to decrease memory reads and writes. The second technique is routing fusion, used to route operations result to multiple registers. Furthermore, the routing can be done automatically by modes. Operator dispatch, memory bandwidth, and memory size costs are all improved via operator fusion and routing fusion [32].

Figure 26 illustrates an example of operator fusion.

```
for(i=0;i<1000;i++)
        {
            tmp1[i]=S*B[i];      // MUL A, B
        }
        for(i=0;i<1000;i++)
        {
            Result[i]=tmp1+C[i];  //ADD A, C
        }
```

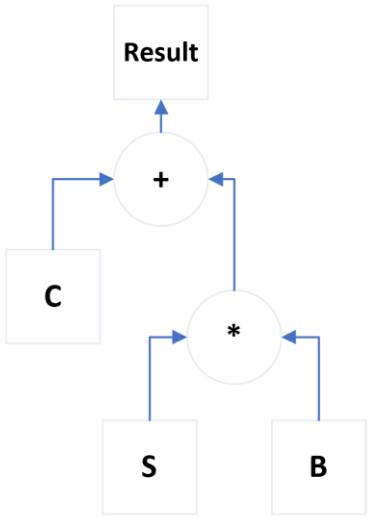


Figure 26: Data flow in the above code

This code can be reduced by Tri-ALU architecture in the assembly language level to the one in Figure 27.

```
for(i=0;i<1000;i++)
      {
          Result[i]=S*B[i]+C[i];     // TRI A * B + F, ACC
      }
```

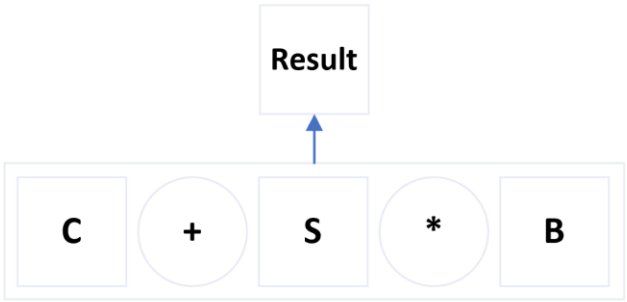


Figure 27 Data flow in the above code

Example Routing fusion: Switching mode (Figure 28)

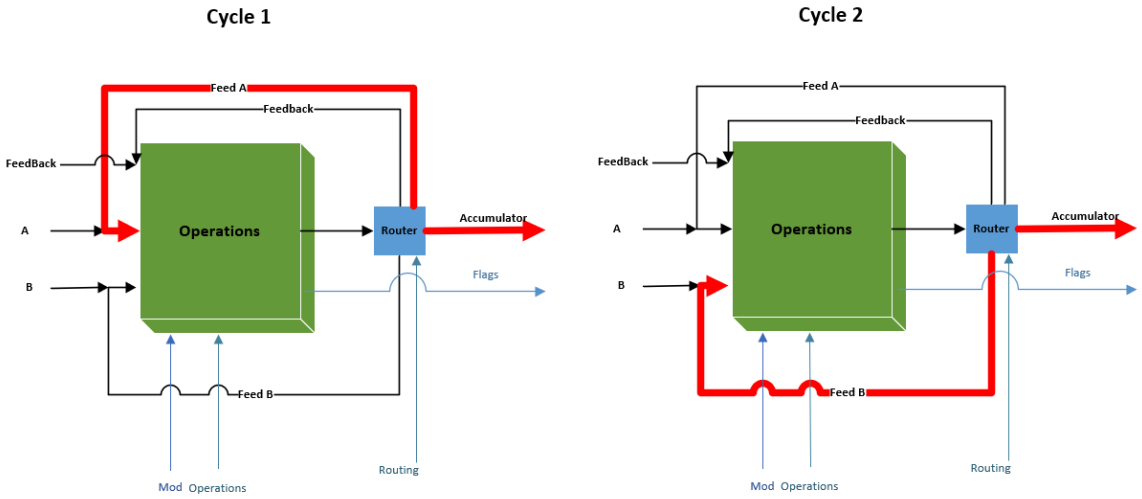


Figure 28: Switching mode

The operation result automatically switches between registers A and B every clock cycle without any MOV instruction.

The benefit of these techniques is to reduce the program size by reducing instructions and minimizing access to memory.

The basic working principle of a Tri-ALU is that it receives three input operands and an operations control signal and then performs the specified operations on the operands and produces the result.

The input operands are typically stored in registers (A, B, and feedback) and are fed into the Tri-ALU through data buses (Data_in_out_controlUnit). Data buses also carry information that operands use in operations (Figure 18). The operation control (ALU_Operation) signal is used to specify which operations the Tri-ALU should perform (Figure 29).

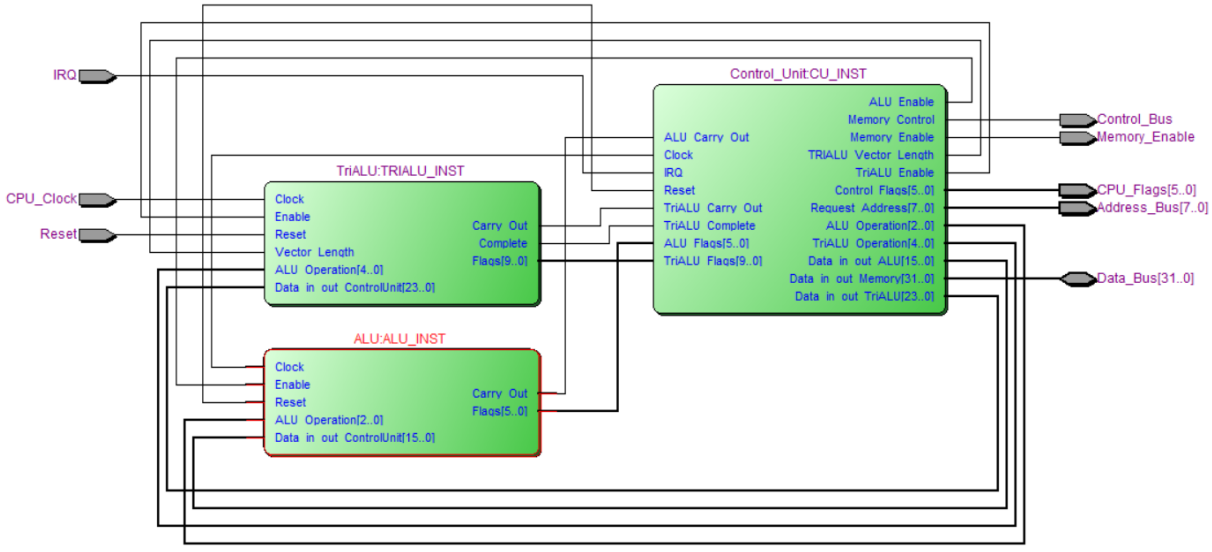


Figure 29: Tri-ALU interfacing with the control unit

The Tri-ALU then performs the specified operation on the input operands using the appropriate arithmetic and logic circuits. The result of the operation is then routed to a

specific register or registers according to the control signal and can be used by other parts of the computer (Figure 31).

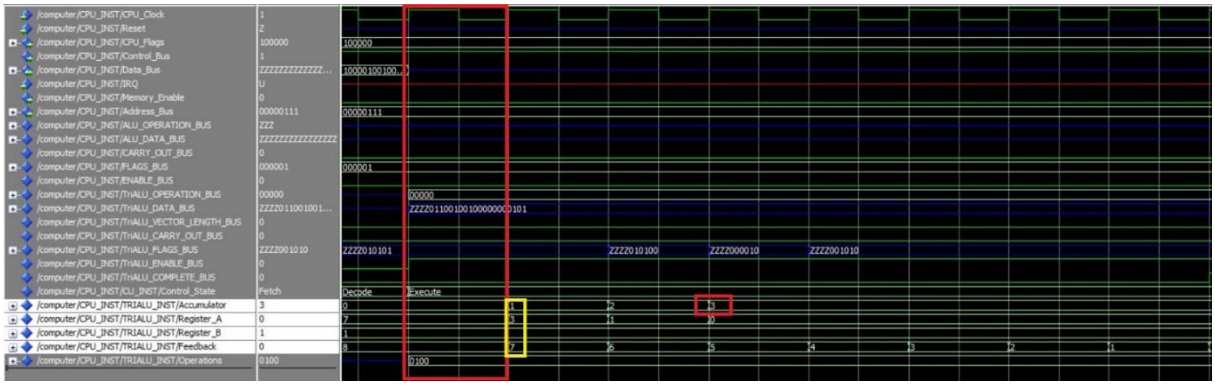


Figure 30: Tri-ALU internal operations are asynchrony with the system clock. All operations inside a Tri-ALU are complete in one cycle.

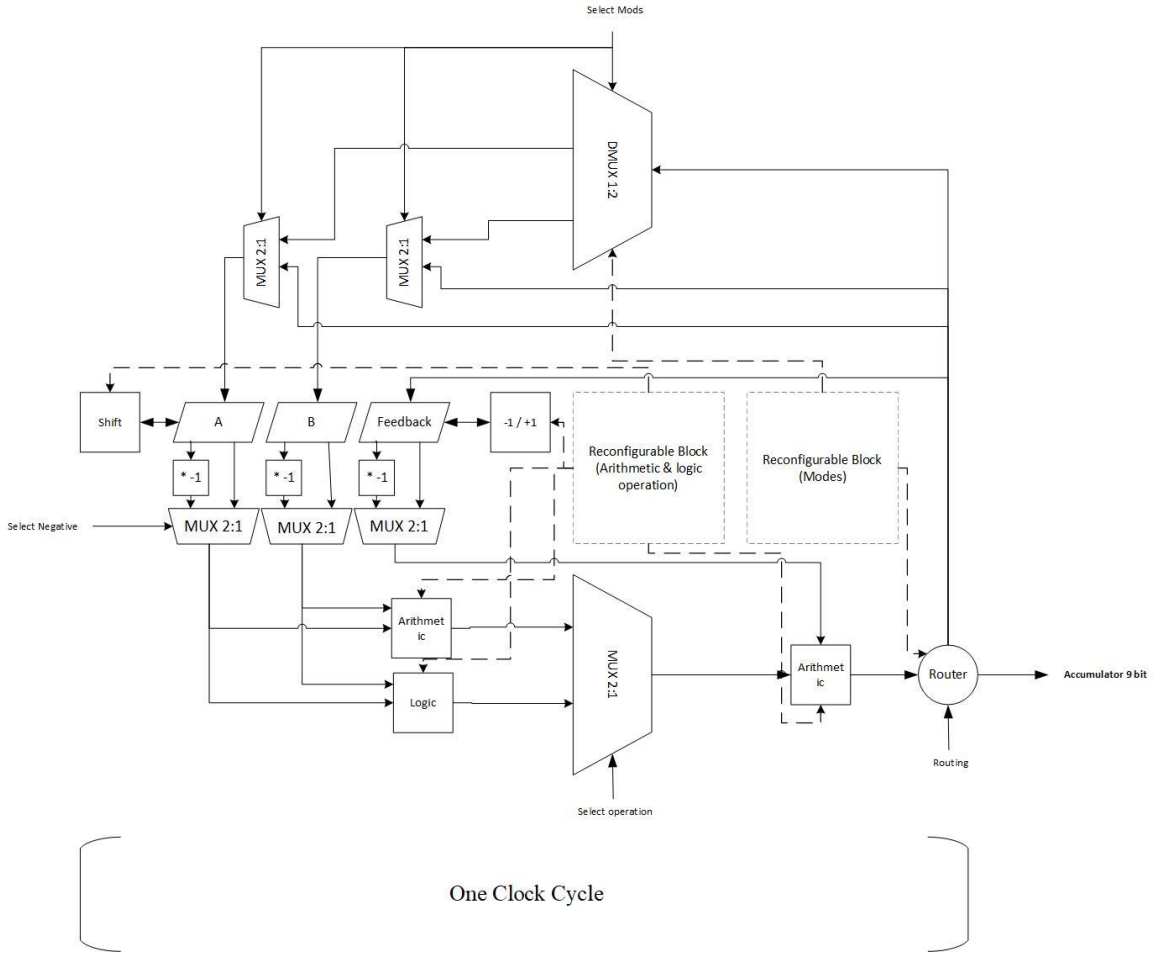


Figure 31: Tri-ALU datapath

It is important to note that a Tri-ALU's specific design and number of operations can vary depending on the application and the resources available on the target platform.

3.2.1 Tri-ALU modes

A Tri-ALU typically has several different modes of routing operations that it can perform. These modes are determined by the specific control signals applied to the Tri-ALU. Each mode corresponds to a different routing that the Tri-ALU can perform on the result.

Tri-ALU modes include:

1- Normal mode: The result routes to a specific register based on the control signal.

2- Switcher mode: The result routes to registers A and B depending on the clock signal (Figure 32).

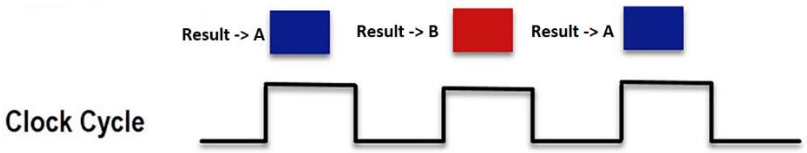


Figure 32: Switcher mode

3- Continuous mode: In this mode, the Tri-ALU will perform looping operations until the feedback register becomes zero (Figure 33).

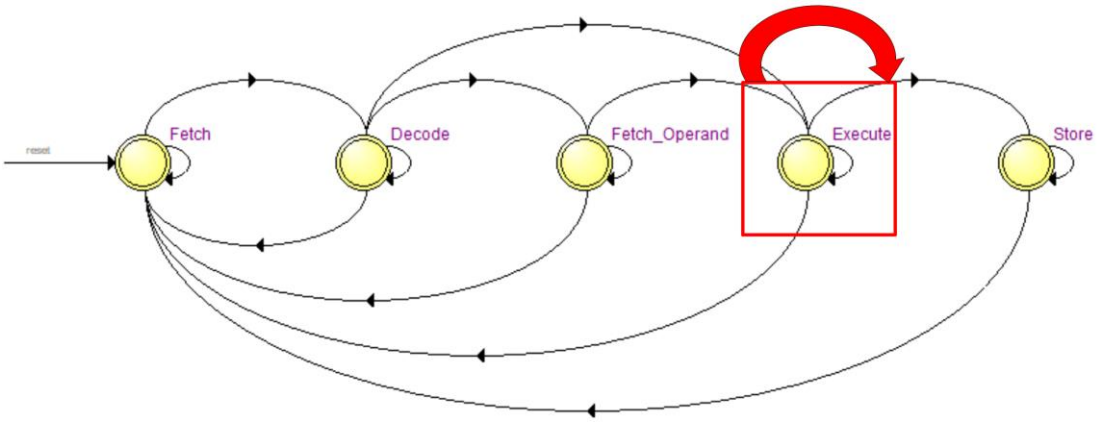


Figure 33: Continuous mode CPU performs the execute state until the feedback register reaches zero.

4- Switcher continuous mode: This mode switches the result between A and B registers and executes looping operations until the feedback register becomes zero.

Most computer programs need looping and indexing; therefore, the Tri-ALU uses the feedback data slot for this purpose to increase CPU performance and decrease access to register files or memory. Furthermore, the Tri-ALU can perform math operations on three data slots (A, B, and feedback) registers with independent math or logic operators between pair data slots. It can also redirect the result data to multiple locations simultaneously. For example, the result can redirect to the accumulator and feedback register.

A Tri-ALU hybrid architecture comprises multiple sub-ALUs, each optimized for a specific type of operation. There are sub-ALUs (stack, accumulator, registers, and vector) based. By combining these sub-ALUs, the Tri-ALU can mix those architectures techniques into a single program. Performing stack zero address instruction operations like (push, pop) with Accumulator one address instructions and vector operations, performing a wide range of operations with high efficiency and low power consumption, and executing a variety of mathematical and logical operations quickly and efficiently (Figure 34).

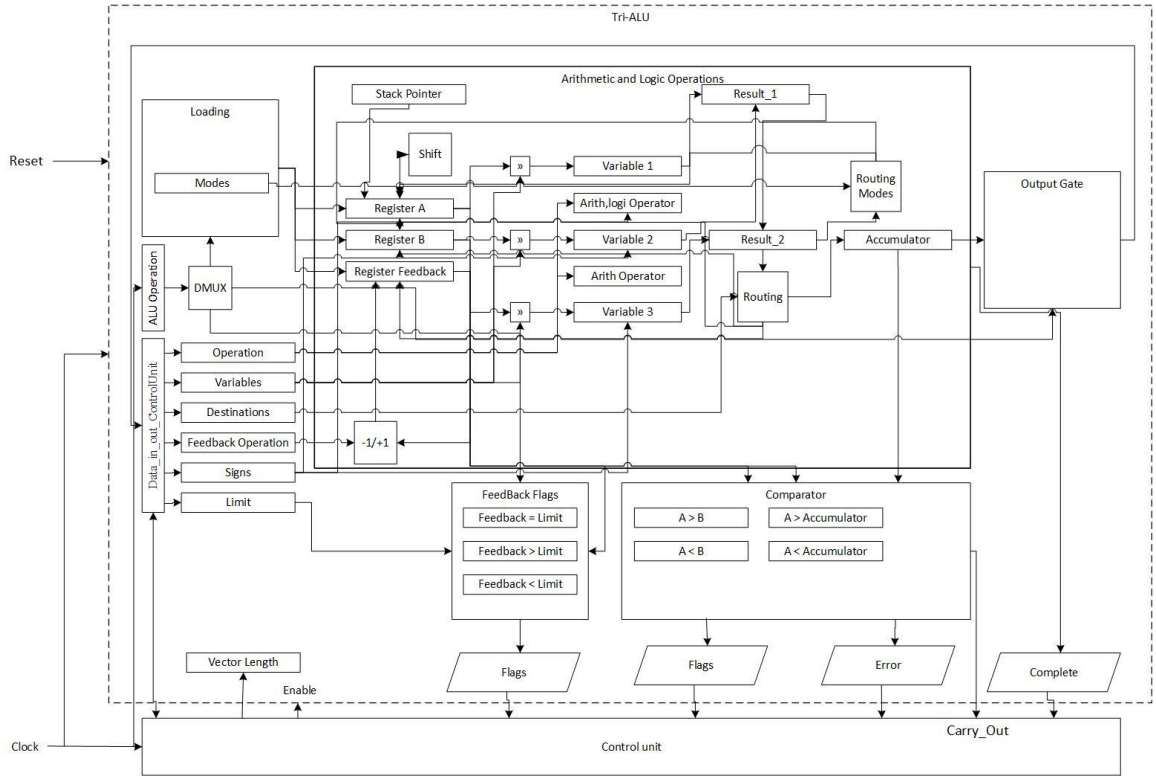


Figure 34: Tri-ALU block diagram

3.3 The Tri-ALU's Internal Structure

A Tri-ALU internal block comprises the following:

1- Tri-ALU arithmetic logic with shifting block (MIMD): Performs two operations (arithmetic or logic) on three registers and a shift register on the left or right simultaneously.

2- Tri-ALU Vector structure block (SIMD): Implements vector processing using SIMD architecture.

3- Tri-ALU Accumulator structure block: Executes one address instruction.

4- Tri-ALU Stack structure block: Executes zero address instruction.

5- Tri-ALU Comparison block: Generates flags depending on the value of registers A and B
6- Tri-ALU Indexing block: Performs increment and decrement operations on the feedback register in parallel with the triple arithmetic block.
7- Tri-ALU Loading block: Loads data from the control data bus to registers.
8- Tri-ALU Output block: Produces the result throw register accumulator.
9- Tri-ALU Feedback flags block: Generates flags depending on the register's value of feedback and limit registers.

3.3.1 Tri-ALU Ports and signals

1. "Clock" signal is an input signal that represents the clock signal.
2. "ALU_Operation" signal is an input signal that represents the five-bit ALU operation code.
3. "Data_in_out_ControlUnit" signal is an input signal that represents data going to and from the CU.
4. "Carry_Out" signal is an output signal that represents the carry-out bit from the ALU.
5. "Flags" signal is an output signal that represents the status flags generated by the ALU.
6. "Enable" signal is an input signal that enables the entity when it is asserted. The signal is initialized with "Z," which represents an undefined state.
7. "Reset" signal is an input signal that resets the entity when it is asserted. The signal is initialized with "Z."

8. "Complete" signal is a buffer output signal that indicates when the ALU operation is complete. The signal is initialized with "0."
9. "Accumulator" signal is a 9-bit signal that represents the output of the ALU.
10. "Register_A" and "Register_B" signals are 8-bit signals that represent the two inputs to the ALU.
11. "Feedback" signal is an 8-bit signal used for feedback operations.
12. "Limit" signal is an 8-bit signal used for limit operations.
13. "Error" signal is a single-bit signal used to indicate whether an error has occurred during the ALU operation.
14. "Operations" signal is a 4-bit signal that represents the ALU operation code.
15. "Variables" signal is a 3-bit signal used to select the input variables for the ALU operation.
16. "Signs" signal is a 3-bit signal used to select the signs for the input variables.
17. "Destinations" signal is a 4-bit signal used to select the destination register for the ALU operation.
18. "Feedback_Operation" signal is a 2-bit signal used to select the feedback operation.
19. "Modes" signal is a 2-bit signal used to select the data transfer mode.
20. "Variable1," "Variable2," and "Variable3" variables are defined as 8-bit. These variables are initialized to all zeros.
21. "Result" variable is defined as 9-bit. This variable is also initialized to all zeros.
22. "Vector_length" is used to detriment the vector length operation.

Overall, these variables are used for storing values used in calculations performed by the Tri-ALU. "Variable1," "Variable2," and “Variable3" represent input values for an arithmetic operation, while "Result" represents the output value of that operation. Since these variables are defined as "variable" types, their values can be modified during the execution of a process or subprogram.

3.3.2 Tri-ALU Arithmetic Logic with Shifting Block

The Tri-ALU takes three 8-bit inputs (Register_A, Register_B, and Feedback), performs a specified arithmetic or logic operation on them (addition, subtraction, AND, XOR), and outputs an 8-bit result. The Tri-ALU also has a switch input and can output its result to various destinations (Accumulator, Feedback, Register_A, or Register_B).

The block checks the value of an input signal called ALU_Operation. Depending on its value, the block performs different operations on the inputs and outputs the result to the specified destination.

The first part of the block handles the signs and variables of the inputs. The signs are represented by a 3-bit signal called signs, where the first bit represents the sign of Register_A, the second bit represents the sign of Register_B, and the third bit represents the sign of the Feedback input. If the sign bit is "1," the corresponding input is multiplied by -1 before the operation is performed. The variables are represented by a 3-bit signal called Variables, where each bit represents whether the corresponding input should be included in the operation. If the variable bit is "0," the corresponding input is set to zero before the operation is performed.

The second part of the block performs the specified arithmetic or logic operation on the inputs. Depending on the value of the Operations signal (which specifies the operation to

be performed), the block performs addition, subtraction, multiplication, or division on the inputs. The result of this operation is stored in a variable called Result1.

The third part of the block performs another arithmetic operation on the result of the previous operation (Result1) and the feedback input. Depending on the value of the Operations signal (which specifies the operation to be performed), the block performs addition, multiplication, or division on the inputs. The result of this operation is stored in a variable called Result2.

The fourth part of the block handles the destination of the output. Depending on the value of the Destinations signal, the output is stored in one of four locations: Accumulator, Feedback, Register_A, or Register_B.

The fifth part of the block handles the shift of operation on Register_A.

The sixth part of the block handles the mode of operation. Depending on the value of the Modes signal, the Tri-ALU can operate in one of three modes: Switch the result between Register_A and Register_B in every math operation, continue in normal mode, or switch the result between Register_A and Register_B in every math operation and continue.

The final part of the block handles the feedback operation. Depending on the value of the Feedback_Operation signal, the feedback input can be decremented, incremented, set to zero, or left unchanged. Finally, the signal "Complete" is set to "1," indicating that the action specified by this ALU_Operation has been completed.

3.3.3 Tri-ALU Vector Structure Block

The Tri-ALU vector structure block is used for vector processing using the SIMD approach. The block contains a series of cases with different opcode values that determine the operation to be performed on the input data.

In this section of the block, the following operations are defined:

Add values: The values in the input data are added to the registers. The first IF statement checks the value of the Vector_Length bit, which determines the vector length. If it equals zero, then only the first two registers are used (A, B). If it is 1, then the third register (Feedback) is also used (Figure 35).

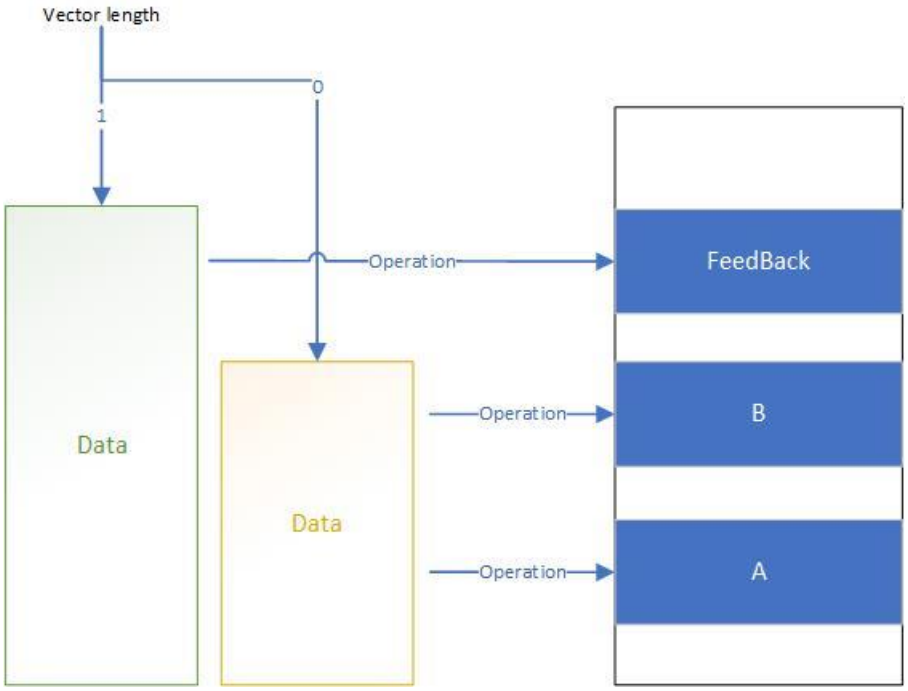


Figure 35: Vector length detriments the registers involved in the operation.

Add value: Similar to the previous case, but the same value is added to both registers.

Greater values: This case compares the values in the three registers and outputs the largest value to an accumulator.

Smaller value: Similar to the previous case, but outputs the smallest value to the accumulator.

XOR values: The values in the input data are XORed with the values in the registers.

XOR value: Similar to the previous case, but the same value is XORed with both registers.

Load values: The values in the input data are loaded into the registers.

Load value: Similar to the previous case, but the same value is loaded into both registers.

Overall, this block defines operations that can be applied to vectors of data using a single instruction. It is designed to improve efficiency and reduce the amount of code required to perform these operations.

3.3.4 Tri-ALU Loading Block

When ALU_Operation equals "00001," it is the loading operation to load data from Data_in_out_ControlUnit to specific registers.

The specific block checks the value of the 2-bit field "Data_in_out_ControlUnit(9 downto 8)" and executes different actions based on its value:

If "Data_in_out_ControlUnit(9 downto 8)" is "00," then the value in the 8-bit field "Data_in_out_ControlUnit(7 downto 0)" is loaded into "Register_A."

If "Data_in_out_ControlUnit(9 downto 8)" is "01," then the value in the 8-bit field "Data_in_out_ControlUnit(7 downto 0)" is loaded into "Register_B."

If "Data_in_out_ControlUnit(9 downto 8)" is "10," then the value in the 8-bit field "Data_in_out_ControlUnit(7 downto 0)" is loaded into a register named "Feedback."

If "Data_in_out_ControlUnit(9 downto 8)" is "11," then the program just sets a mode specified in the 2-bit field "Data_in_out_ControlUnit(11 downto 10)" and does not load any value into any register.

Regardless of which case is executed, the 2-bit field "Data_in_out_ControlUnit(11 downto 10)" is also loaded into a register named "Modes."

Finally, the signal "Complete" is set to "1," indicating that the action specified by this code has been completed.

Overall, this block handles the input data, deciding which register to load with the input value and setting certain modes in the circuit, then indicating that the operation has been completed.

3.3.5 Tri-ALU Feedback Flags Block

When ALU_Operation equals "00010," it checks whether the value in the signal Feedback is less than, greater than, or equal to a certain limit and sets specific bits in the signal Flags accordingly.

Here is a line-by-line explanation of the block:

If Feedback = Limit, then: If the value of the signal Feedback is equal to the limit (presumably defined elsewhere in the code), then execute the following block of circuit.

Flags(6) <= "1": Set the 6th bit (counting from 0) of the signal Flags to 1.

else: If the previous condition is not met, execute the following block of circuit instead.

Flags(6) <= "0": Set the 6th bit of the signal Flags to 0.

end if: End the conditional statement.

The next two blocks of circuits are similar but set the seventh and eighth bits of Flags based on whether Feedback is less than or greater than the limit, respectively. These blocks work in the same way as the first block, setting the relevant bit to 1 if the condition is true, or 0 if it is false.

Set the signal Complete to 1, indicating that the operation has been completed successfully.

This block is used to set certain flags in response to the value of Feedback relative to a certain limit, which is used to control the behavior of the overall system**.** Finally, the signal "Complete" is set to "1," indicating that the action specified by this ALU_Operation has been completed.

3.3.6 Tri-ALU Comparator Block

The Tri-ALU comparator block performs a series of comparisons and sets the appropriate flags based on the values of various registers.

Flags(5) is set to Error if the Error signal is high, indicating that a division by zero has occurred.

Carry_Out is set to the most significant bit of the Accumulator register.

Flags(4) is set to 1 if Register_A is greater than Register_B, and 0 otherwise.

Flags(3) is set to 1 if Register_A is less than Register_B, and 0 otherwise.

Flags(2) is set to 1 if Register_A is greater than the Accumulator register, and 0 otherwise.

Flags(1) is set to 1 if Register_A is less than the Accumulator register, and 0 otherwise.

Flags(0) is set to 1 if the lower 8 bits of the Accumulator register are all 0, indicating that the result of the previous operation was 0.

Finally, the process sets the Complete signal to 1, indicating that the operation is complete.

3.3.7 Tri-ALU Accumulator Structure Block

The Tri-ALU accumulator structure block performs arithmetic and logic operations on two registers, Register_A and Register_B, and stores the result in an 8-bit accumulator called Accumulator.

For example, when ALU_Operation equals "00100," the circuit adds the contents of Register_A to the least significant eight bits of Accumulator, and the result is stored back in Accumulator. The ext function extends the 8-bit result to a 9-bit value, with the most significant bit being the carry-out bit from the addition operation.

3.3.8 Tri-ALU Stack Structure Block

The Tri-ALU stack structure block implements stack-based. The input code is a binary value, and when ALU_Operation equals "10100," the circuit performs a push operation; when ALU_Operation equals "10101," it performs a pop operation.

When ALU_Operation equals "10100" (push operation), the circuit stores the value from Data_in_out_ControlUnit (which is an 8-bit register that serves as an input and output port to the CU) into one of the three registers, Register_A, Register_B, or Feedback, depending on the current value of the Stack_Pointer. The Stack_Pointer is a 2-bit register that points to the top of the stack. After the value is stored, the Stack_Pointer is incremented by 1 (Figure 36).

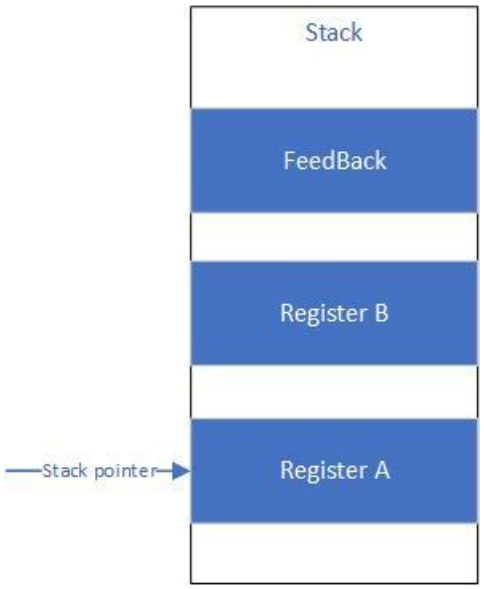


Figure 36: Tri-ALU stack

If Stack_Pointer is "00," the value is stored in Register_A.

If Stack_Pointer is "01," the value is stored in Register_B.

If Stack_Pointer is "10," the value is stored in Feedback.

If Stack_Pointer is any other value, it means that the stack has overflowed, and the Flags register is set with a high value on its ninth bit (Tri-ALU Flags is a 10-bit register).

When ALU_Operation equals "10101" (pop operation), the circuit reads the value from the top of the stack and stores it in the Accumulator. If Stack_Pointer is "01," the value is read from Register_A; if it is "10," the value is read from Register_B; and if it is "11," the value is read from Feedback. After the value is read, the Stack_Pointer is decremented by 1.

The Stack_Pointer is used to keep track of the top of the stack, and the Register_A, Register_B, and Feedback registers are used to store the values in the stack. The block handles stack overflow and underflow conditions and generates error flags accordingly.

Finally, the stack block can perform arithmetic and logic operations on the stack.

3.3.9 Tri-ALU Output Block

When the ALU_Operation equals "00011" is detected, the output signal Data_in_out_ControlUnit to the value of the Accumulator signal's lower, 8 bits, which are represented by the range (7 downto 0).

In addition, the block sets the signal Complete to logic level "1," which indicates that the operation is complete.

3.4 Summary

In this chapter, the researcher explained the Tri-ALU and how the multi-operation and execution Modes increase the system performance. The next chapter gives an overview of our computer system design with Tri-ALU and shows Tri-ALU applications, finally benchmarking our proposed system with other architecture, and then discusses the result.

CHAPTER IV.

DESIGN AND IMPLEMENTATION

This chapter is arranged so that it first presents the design and then the application of the Tri-ALU and benchmarking for comparison. Then, the actual results and the corresponding discussions are provided.

4.1 System Design

A Tri-ALU is designed to have three main components: Arithmetic and logic, indexing, and comparison. The arithmetic and logic aspect performs traditional arithmetic and logic operations, such as addition, subtraction, AND, XOR, and shift operations. The indexing aspect performs index operations, such as increment and decrement. The comparison aspect performs comparison operations, such as greater than, less than, and equal to.

The Tri-ALU is implemented using digital logic circuits, which comprise transistors, diodes, and other electronic components (hard IP core). These circuits are designed to perform the necessary arithmetic, logical, and comparison operations quickly and accurately.

Moreover, the Tri-ALU can be implemented using an FPGA, which is a type of digital logic circuit that can be customized to perform a wide range of arithmetic, logical, and comparison operations (soft IP core).

An FPGA is a type of programmable logic device that can be configured to implement different digital circuits. The FPGA can be programmed using a hardware description

language (HDL), such as VHDL or Verilog, which allows for the design and implementation of a Tri-ALU.

The Tri-ALU built on FPGA can be highly customizable and tailored to specific application requirements. It also allows for more efficient use of resources, as it can be implemented using only the resources required. However, it also requires a more complex design and implementation process, as well as specialized knowledge of FPGA design.

4.2 Research Design

The researcher built a soft-core CPU using VHDL (very high-speed integrated circuit hardware description language) on an Altera DE2 FPGA development board with Intel Quartus Prime logic device design software. The researcher built two ALU versions: One is a typical ALU design, and one is a Tri-ALU. And The researcher will test the performance and power consumption for both designs by applying a Fibonacci function using Modelsim software for simulation. The CPU will integrate with other computer parts like the IO controller and memory, components referred to as a computer system. The computer system is a configuration that includes all functional components of a computer and its associated hardware. A basic microcomputer system includes a console, or system unit, with one or more disk drives, a monitor, and a keyboard. Additional hardware, called peripherals, can include such devices as a printer, a modem, and a mouse. Software is usually not considered part of a computer system, although the operating system that runs the hardware is known as system software [33].

4.2.1 Computer Architecture

The computer architecture is based on von Neumann. John von Neumann developed the stored-program concept, which is the foundation of modern computers. Programs and data are stored in a separate storage device called memory and are treated equally in this stored-program model. A computer made using this design would be considerably simpler to program thanks to this innovative concept. The basic components of a computer are the CPU, main memory unit, and input/output device [34] (Figure 37).

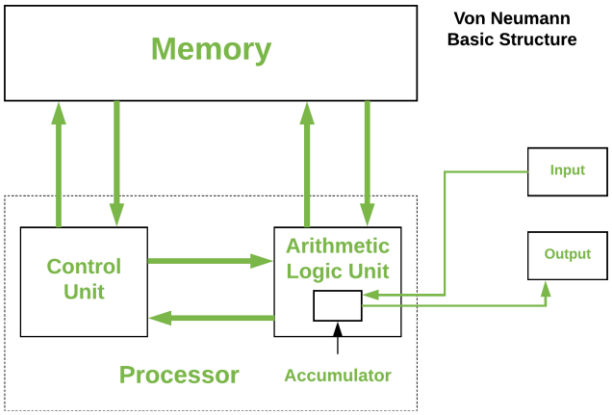


Figure 37 von Neumann architecture

The computer is 8-bit, meaning it can process 8-bit data at a time. It can also add two 8-bit numbers together in one CPU cycle.

There are three main components of computer:

1- Central processing unit

2- Memory

3- IO controller

The memory stores the instruction and data, and the CPU fetches and executes the instruction and writes back the result to the memory or the output device through the IO controller. The CPU has two main components: the CU and the ALU. The CPU is the most critical part of a computer system since it performs and controls all the computer system. The IO controller governs the data destination (input or output) by the CPU control signals. There are three major buses:

1- Address bus
2- Data bus
3- Control bus

Computer system buses are used to connect the system component by transferring the data and control signal between each part.

LEDs in the board are used as an output unit, and switches are used as an input unit. CPU flags are indicated by LEDs (Figure 38).

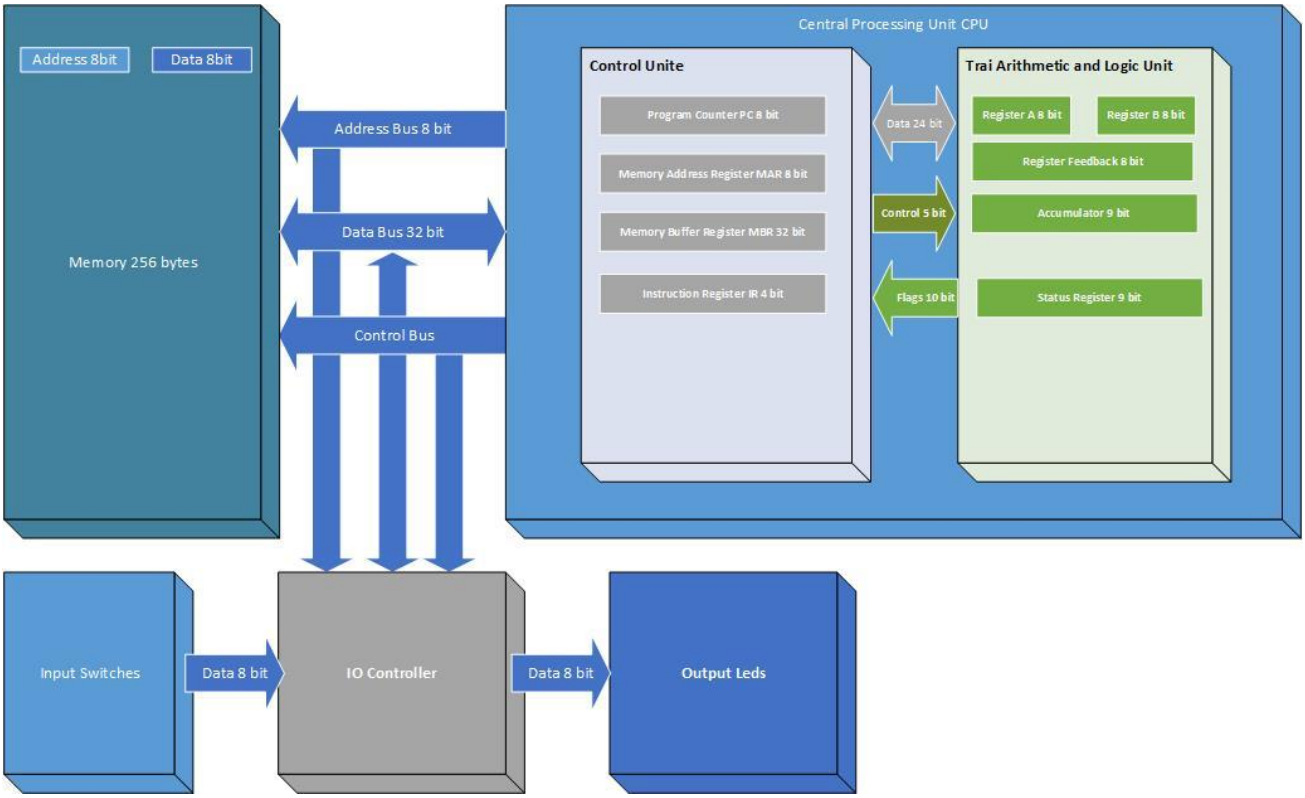


Figure 38: Top-level design of an 8-bit computer

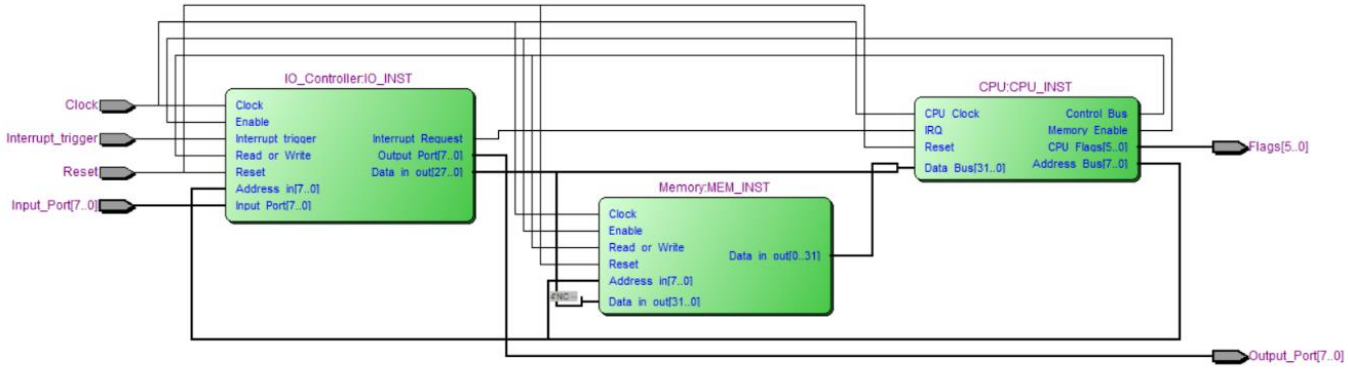


Figure 39: RTL view of a computer

Computer VHDL entity:

```
  Entity Computer is
  port(
  Clock : in std_logic; -- System Clock
  Reset : in std_logic:='Z';
  Flags : out std_logic_vector(5 downto 0); -- Flags from control
unit
  Interrupt_trigger : in std_logic;
  Input_Port : in std_logic_vector(7 downto 0);
  Output_Port : out std_logic_vector(7 downto 0)
  );
  end entity;
```

The system clock will come from the DE2 Altera board through the PIN_N2 50 MHz clock input. An internal clock is used by computers to synchronize all their computations. The clock ensures that all a computer's circuits are active at once [35].

The reset signal will be assigned to the PIN_G26 switch. A reset returns a system to its initial state or to normal operation, usually in a controlled way. It also clears any pending errors or events [36].

Flags signals will be assigned to the PIN_U17, PIN_U18, PIN_V18, PIN_W19, PIN_AF22, and PIN_AE22 LEDs, respectively. Flags are a signal indicating the existence or status of a particular condition inside the CPU.

The interrupt trigger will be assigned to the PIN_B13 switch. An interrupt I/O is a data transfer mechanism in which a peripheral or external device notifies the CPU that it is ready for communication and seeks the CPU's attention. The interrupt signal is used when user input is ready to transfer to the CPU [37].

Input port signals will be assigned to the PIN_C13, PIN_AC13, PIN_AD13, PIN_AF14, PIN_AE14, PIN_P25, PIN_N26, PIN_N25, and PIN_B13 switches, respectively.

Output port signals will be assigned to the PIN_AC21, PIN_AD21, PIN_AD23, PIN_AD22, PIN_AC22, PIN_AB21, PIN_AF23, and PIN_AE23 LEDs, respectively (Figure 28).

Figure 40: IO unit on a board

4.2.2 Memory Design

The memory is a device where information (instruction, data) can be stored and retrieved. It is directly connected to the processor [33]. The memory is broken up into many tiny pieces called cells. A unique address is assigned to each piece or cell, and it can range from zero to memory size minus one [38].

Memories are made up of an array of registers, each of which has a distinct address. The memory has a size of *N*x*M*, where *N* is the number of registers, and *M* is the number of bits in a word [39]. In this computer, the memory size = 2^8 = 256 location, and each word size = 8 bits; therefore, the total memory size is 256 bytes from address x"00" to x"FA" for memory and from address x"FB" to x"FF" for IO devices. The memory type is RAM, so the CPU can access the memory randomly (Figure 41).

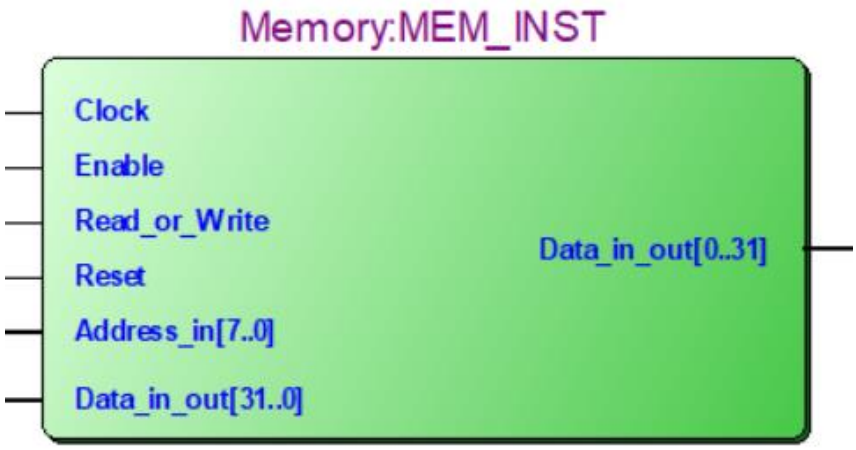


Figure 41: RTL view of the memory

4.2.3 IO Controller Design

A central processor unit's bus system is connected to input and output (I/O) devices by an I/O controller. It operates on a variety of devices and often communicates across the system bus with the CPU and system memory. Control is typically started by the CPU, which instructs the I/O controller on how to manage the peripheral devices and signals connected to it. Peripherals are mapped onto system address ranges via the I/O controller. I/O controllers can be set up to use interrupt-based special signaling to notify the CPU when events or transfers occur [40] (Figure 42).

In our computer system, the I/O memory mapped from address x"FB" to x"FF" in hexadecimal.

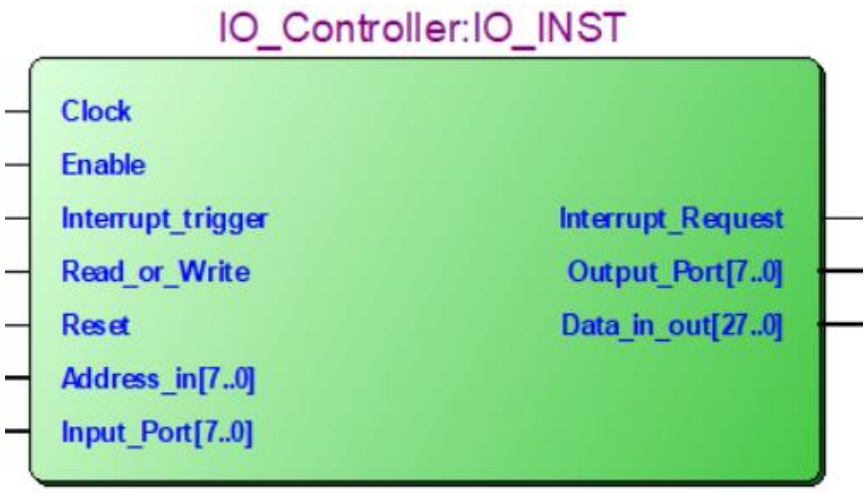


Figure 42: RTL view of IO controller

4.2.4 System Bus Design

A system bus is a set of electrical signals with a group function. For example, the data bus is a set of *n* pins used by a processor to read and write *n*-bit data values or opcodes from a memory device [41]. The system bus operates by merging the duties of the data, address, and control buses, which are the three primary buses. Each of the three buses is unique in terms of its attributes and duties [42].

The computer utilizes three bus types:

1- ADDRESS BUS: The address bus refers to the buses that the CPU utilizes to transmit a memory location's address. The CPU features an 8-bit address bus with 256 memory addressable places. Data can only move in one direction on the address buses, from the microprocessor to the peripheral devices. When a microprocessor writes data to a port, the address bus can also be used to send the port address.

2- DATA BUS: The data bus of the CPU contains 32 parallel lines. The data bus is a bidirectional bus since it can transfer data from the CPU to the memory and vice versa. The size of the read or written data word determines how many data lines are used in the data bus. Data is also read from or written to the RAM or I/O ports via the data buses.

3- CONTROL BUS: Timing and control signals travel on lines known as control buses of 2-bit sizes. To enable the outputs of targeted memory devices or I/O port devices, the CPU transmits signals on the control bus. The control signals are memory read, memory write, and enable [43].

4.2.5 Central Processing Design

The CPU is the heart of a computer system responsible for processing data and controlling the computer peripherals (Figure 43). It is the part of a computer that obtains and executes instructions and comprises a CU, a number of registers, and an ALU. The term "processor" is frequently used to refer to the CPU. According to the program's instructions, the ALU executes arithmetic operations, logic operations, and associated operations [44].

In our system, the microprocessor is a "subscalar CPU," which is a form of processor design where each instruction can only work with one operand or piece of data at a time, and the CPU can only execute one instruction at a time.

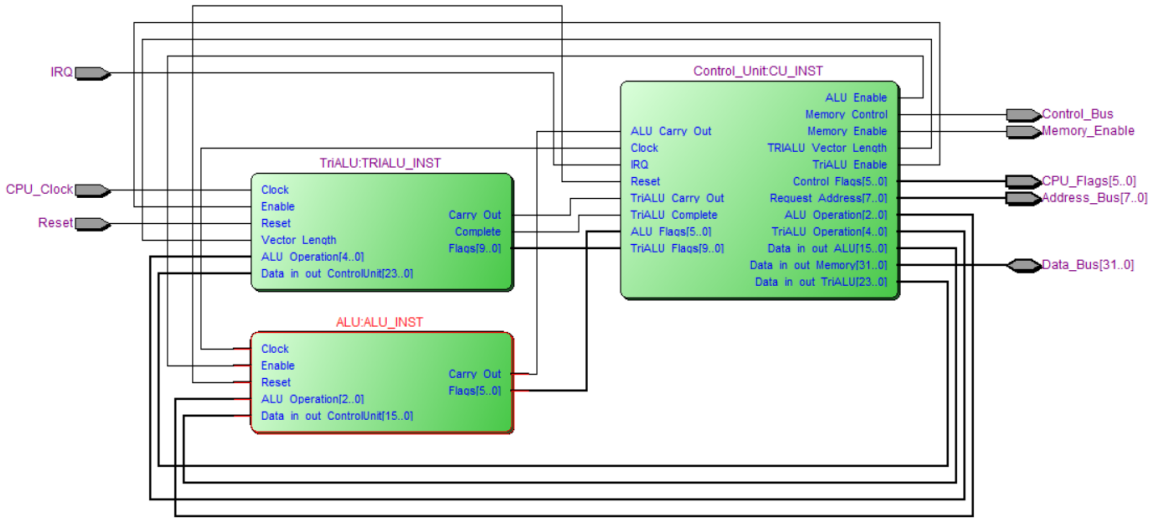


Figure 43: RTL view of the CPU

The processor in this design is multicycle, which means each instruction is divided into a set of stages, each of which takes one clock cycle to complete.

4.2.6 Micro-architecture

Control unit: The control unit is a fundamental component of a CPU in a computer system. The CU is responsible for controlling the operation of the processor, including the execution of instructions and the management of data flow within the CPU. The CU is based on a hardwired logic design approach. To transfer data between registers in a computer system, specific signals need to be generated. For instance, to move the contents of one register to another, signals are required to load the source register's contents onto a bus and then load the bus's contents into the destination register. This hardware implementation typically involves one signal for each source register and one signal for each destination register. Once the signals have been generated, the logic progresses to the next state and repeats the process [45] (Figure 44).

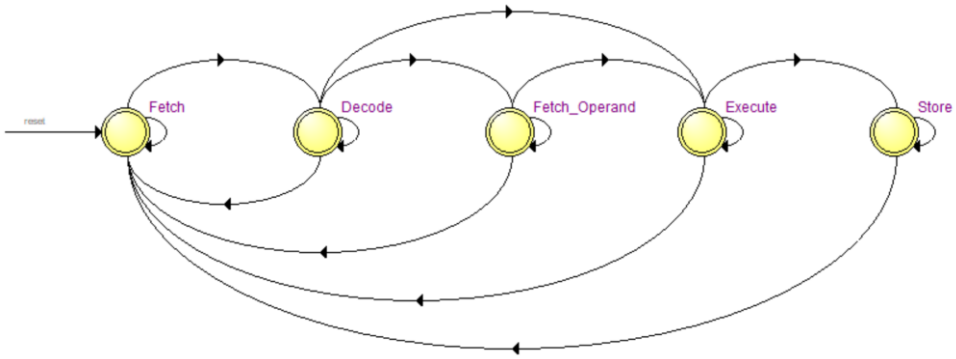


Figure 44: Control unit state machine

ALU: The ALU is responsible for performing arithmetic and logical operations on data, including addition, subtraction, multiplication, division, and bitwise operations. The ALU contains three registers: A, B with an 8-bit size for each register, and Accumulator with a 9-bit size.

4.2.7 Tri-ALU

The proposed component in the research is the Tri-ALU depicted in Figure 45.

Figure 45: RTL view of a Tri-ALU

System clock: This comes from the DE2 Altera board through the PIN_N2 50 MHz clock input. A system clock is used by computers to synchronize all their computations. The clock ensures that all a computer's circuits are active at once [35].

Enable: When data and control signals are sent to the Tri-ALU, the CU sends an enable signal to the Tri-ALU to perform the operation. This signal ensures the data are received and the Tri-ALU is ready to execute the commands.

Reset: The reset signal is assigned to the PIN_G26 switch. A reset returns a system to its initial state or to normal operation, usually in a controlled way. It also clears any pending errors or events [36].

Data in out ControlUnit: Data and command signals (inner operation, modes, signs, routing, variables) from the CU.

ALU Operation: Operation (math, load, compare) command signal from the CU.

Vector length: This sets the registers included in the vector operation.

Carry out: This is the overflow bit signal from math operations to CPU flags.

Flags: FeedBack > limit bit[8], FeedBack < limit bit[7], FeedBack = limit bit[6], division by zero bit[5], A>B bit[4], A<B bit[3], A > Acc bit[2], A < Acc bit[1], and zero flags bit[0].

Tri-ALU Instruction:

Table 1: Assembly instruction: TRI

OP Code 31:28	Operation1 27:26	Operation2 25:24	Variables 23:21	Signs 20:18	Destinations 17:14	Feedback operation 13:12	Division over operation 11	Logic operation 10 Depend on operation 1	Shift operation On Register A 9:8
1000	00 no operation	00 no operation	111 A,B,F	0 +	0001 Accumulator	00 no operation	1 $\frac{x}{y+z}$		00 no shift
	01 A+B	01 Feedback + B	011 B,F	1 -	0010 Feedback	01 Decrement -1	0 $\frac{y+z}{x}$	00 AND	01 shift right
	10 A*B	10 Feedback * B	101 A,F		0100 A	10 Increment +1		01 XOR	10 shift left
	11 A/B	11 Feedback / B	110 A,B		1000 B	11 Set Zeros		10 OR	
			100 A,B,ACC					11 NAND	

This command is used to send two math operations between Registers A and B and Feedback. Moreover, the command can choose which variable is used in the operation and can choose A and Feedback or A and B, depending on the operation requirement. The signs are used to change the variables signs. Destination is used for routing the result. Feedback operation is used to apply decrement or increment operations on the feedback register. Division over operation detriments the division operation sequence. For example, when Operation 1 addition and Operation 2 division if division over operation = 1, then the result will equal

Result = $\frac{Feedback}{A+B}$

Else

Result = $\frac{A+B}{Feedback}$

Table 2: Assembly instruction name: TLDA

OP Code 31:28	27:24	Modes 23:22	Register select 21:20	Data 19:12
1000	0000	00 Normal	00 A	
		01 Switcher	01 B	
		10 Continues Normal	10 Feedback	
		11 Continues Switcher	11 Accumulator	

This command used to load data to Registers A or B or Feedback and set the Tri-ALU Modes.

Table 3: Assembly instruction name: TJMP

OP Code 31:28	Limit 27:20	Address 19:12	Jump Modes 11:8
1010			1101 Jump if FeedBack = limit
			1110 Jump if FeedBack < limit
			1111 Jump if FeedBack > limit

This command is used to jump to a specific memory address if the condition is true. Limit is a value used to compare it with a feedback register.

Table 4: Assembly instruction name: TSTA

OP Code 31:28	Address 27:20	Data 19:12
0011		

This command is used to store the Tri-ALU Accumulator value to a specific memory address.

Table 5: Assembly instructions for Tri-ALU Accumulator operations

OP Code 31:28	Operations 27:24	Assembly	Description
1001	0100	TADD A	ACC= A+ACC
	0101	TSUB A	ACC= A-ACC
	0110	TAND A	ACC= A AND ACC
	0111	TOR A	ACC= A OR ACC
	1100	TADD B	ACC= B+ACC
	1101	TSUB B	ACC= B-ACC
	1101	TAND B	ACC= B AND ACC
	1111	TOR B	ACC= B OR ACC

This command is used to perform Tri-ALU Accumulator structure one address operation.

Table 6: Assembly instruction for Tri-ALU stack operations

OP Code 31:28	Operations 27:24	Assembly	Description
1100	0100	TPUSH []	Push value in Tri-ALU stack
	0101	TPOP	Pop value from Tri-ALU stack
	0110	TADD	Perform Add on Tri-ALU stack
	0111	TSUB	Perform Sub on Tri-ALU stack
	1100	TMUL	Perform Multiple on Tri-ALU stack
	1101	TDIV	Perform Division on Tri-ALU stack
	1101	TAND	Perform AND on Tri-ALU stack
	1111	TOR	Perform OR on Tri-ALU stack

This command used to perform Tri-ALU Accumulator structure one address operation.

Table 7: Assembly instruction for Tri-ALU vector operations

OP Code 31:28	Operations 27:24	Assembly	description
0111	000	TVADD [,,]	Add values to registers A, B, Feedback
	001	TVADD []	Add value to registers A, B, Feedback
	010	TVG	Store greater value between A,B,Feedback in Accumulator
	011	TVS	Store smaller value between A,B,Feedback in Accumulator
	100	TVXOR [,,]	Vector XOR values with registers A, B, Feedback
	101	TVXOR []	Vector XOR value with registers A, B, Feedback
	110	TVLDA [,,]	Load values to registers A, B, Feedback
	111	TOR	Load value to registers A, B, Feedback

This command is used to perform Tri-ALU vector operations.

4.3 Tri-ALU Application

4.3.1 Machine Learning (Neural Network)

A neural network is a collection of algorithms that aims to identify underlying links in a set of data using a method that imitates how the human brain functions [46] (Figure 46). The Tri-ALU can increase neural network learning performance by enhancing matrix multiplication operations. A neural network depends on matrix multiplication in the feedforward mechanism. Most machine learning models are based on matrix multiplication (Figure 47).

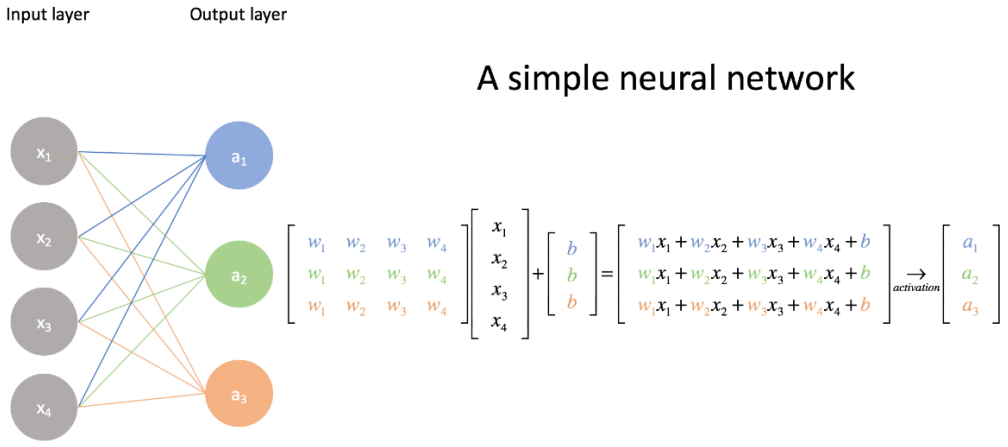


Figure 46: Neural network [29]

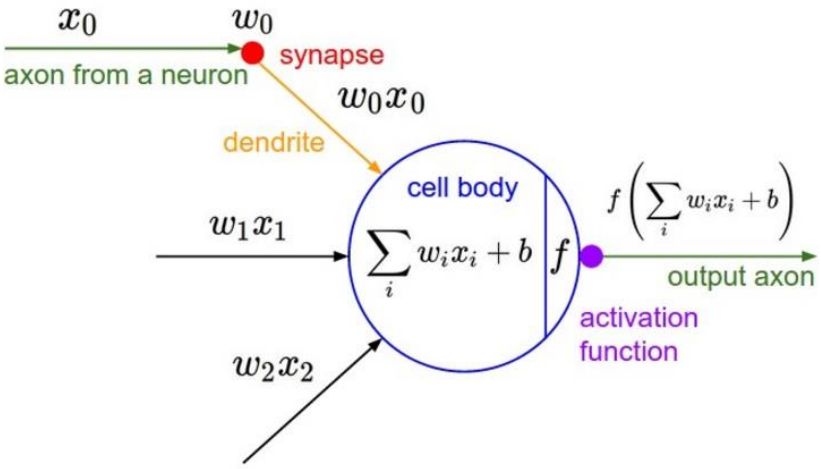


Figure 47: Feedforward mechanism [29]

We will take the first row of the matrix:

$$w1x1 + w2x2 + w3x3 + w4x4 + b$$

Table 8: Tri-ALU can take the inputs X and weights and bias and calculate the result.

Assembly language	Description
TLDA A, w1	Load wight1 to register A
TLDA B, x1	Load x1 to register B
TRI A * B , FB	Feedback = w1*x1
TLDA A, w2	Load wight2 to register A
TLDA B, x2	Load x2 to register B
TRI A*B + FB , FB	Feedback = w2*x2+Feedback
TLDA A, w3	Load wight3 to register A
TLDA B, x3	Load x3 to register B
TRI A*B + FB , FB	Feedback = w3*x3+Feedback
TLDA A, w4	Load wight4 to register A
TLDA B, x4	Load x4 to register B
TRI A*B + FB , FB	Feedback = w4*x4+Feedback
TLDA A, Bias	Load Bias to register A
TRI A+FB, Acc	Accumulator = Bias + Feedback

From the above assembly code, it is clear that the Tri-ALU can execute two mathematical operations in a single run and route the result to Feedback. This method speeds up the neural learning operation.

4.3.2 Mathematical Sequences and Series (Triangular Number Series)

The universe of mathematical series and sequences is intriguing. These series are excellent tools for mathematical reconstruction. In addition to several branches of mathematics, the sequences are also found in many other disciplines, such as physics, chemistry, and computer science [47]. The Tri-ALU can calculate and find a mathematical series term more efficiently than conventional ALUs by applying switcher and continuous mode. These Modes can reroute data between registers without any extra instruction to calculate mathematical series, such as triangular number series (Figure 36). The things that can be arranged into an equilateral triangle are tallied by a triangular number or triangle number.

The sum of the *n* natural numbers from 1 to *n* determines the *n*th triangle number, which is the quantity of dots or balls in a triangle with *n* sides (Figure 48).

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

Figure 48: Pictorially, the triangular numbers can be represented as above.

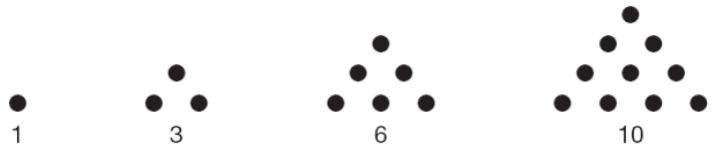


Figure 49: The sequence of triangular numbers is 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, …

Table 9: Finding the fourth term in a triangular number series

Assembly language	Description
TLDA A 4	Load value 4 to register A
TLDA B 1	Load value 1 to register B
TLDA Feedback 2	Load value 2 to register Feedback
TRI A + B, B	A + B routing the result to B $(n+1)$
TRI A * B / Feedback, ACC	A * B / Feedback routing the result to Accumulator ($\frac{n(n+1)}{2}$)
TSTA x“1C”	Send Accumulator value to output leds
HLT	Halt

The Tri-ALU can perform multiplication and division in a single command. This feature makes the Tri-ALU faster than conventional ALUs.

4.3.3 Factorial and Power of Number

Factorial**:** In mathematics, the term "factorial" refers to the product of all positive integers that are less than or equal to a certain positive integer, which is followed by an exclamation point. Thus, factorial seven is denoted by the symbol 7!, which stands for 1 × 2 × 3 × 4 × 5 × 6 × 7. Factorial 0 is equivalent to 1 by definition [48].

Factorial formal by product:

$$n! = \prod_{K=1}^{n} K$$

Table 10: The Tri-ALU can calculate factorial numbers using continuous mode (Figure 50).

Assembly language	Description
TLDA Feedback 5	Load value 5 to register feedback
TLDA A 1 MOD C	Load value 1 to register A set mod to continues
TRI Feedback * A , ACC ,A	Accumulator = A = Feedback * A
TSTA x“1C”	Send Accumulator value to output leds
HLT	Halt

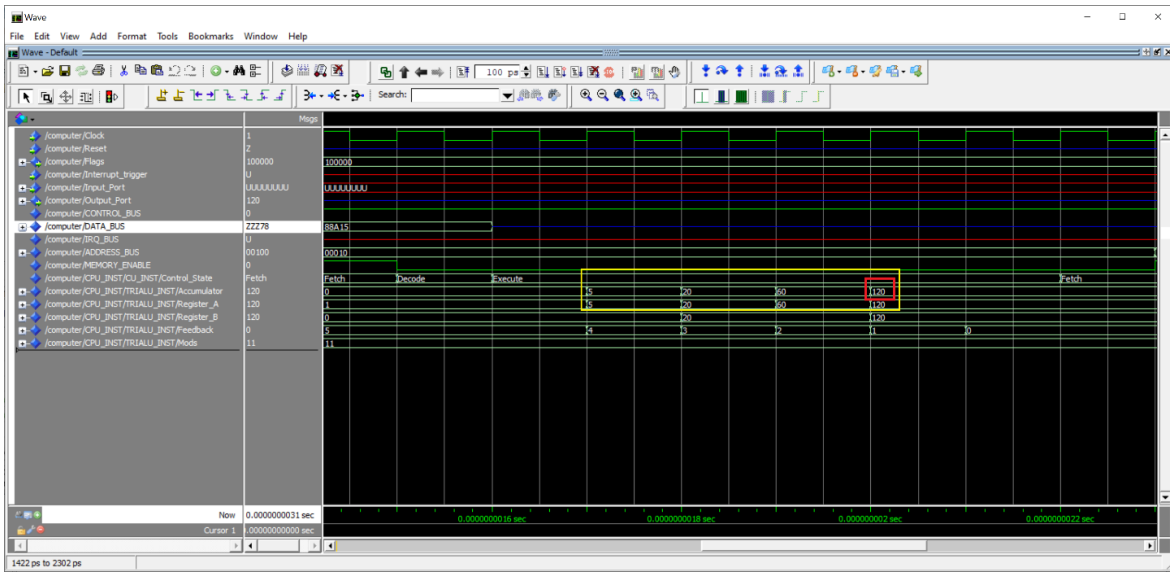


Figure 50: Continuous operation: Every clock cycle, the Tri-ALU executes the instruction 5! until it reaches the desired output.

Power of: The number's power indicates how many times the original number can be multiplied. An exponent is a symbol used to express a number's power.

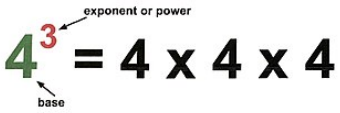


Table 11: The Tri-ALU can calculate a number’s power using continuous mode.

Assembly language	Description
TLDA Feedback 5	Load value 5 to register feedback
TLDA B 2	Load value 2 to register B
TLDA A 1 MOD C	Load value 1 to register A set mod to continues
TRI A * B , ACC ,A	Accumulator = A = A * B
TSTA x“1C”	Send Accumulator value to output leds
HLT	Halt

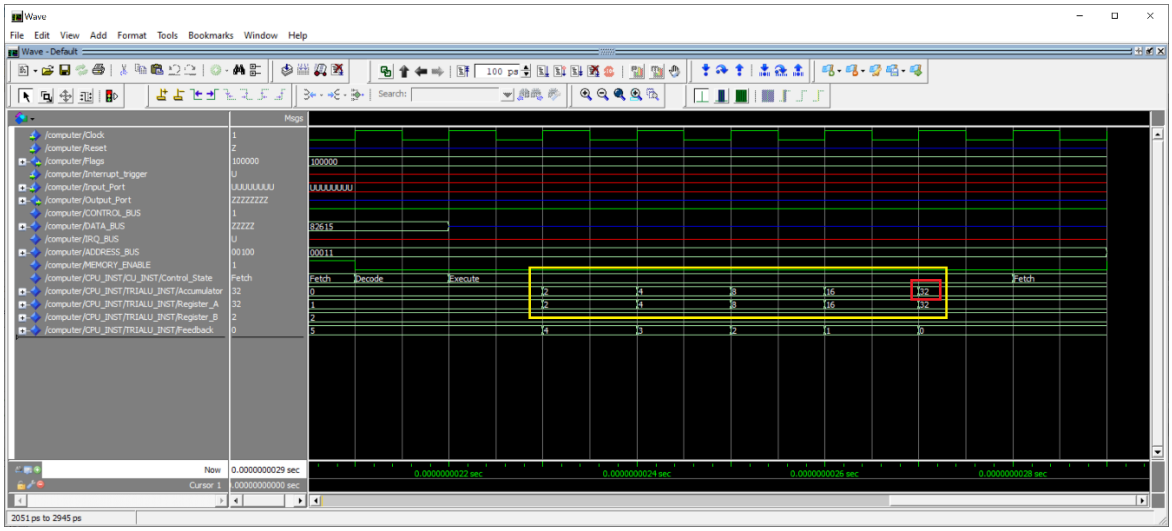


Figure 51: 2^5 calculated by Tri-ALU using continuous modes

4.3.4 MAC Operations (For Filters)

The MAC class of dual-operand DSP instructions are commonly used to perform the main Dot Product operation in finite impulse response filters and in several other DSP algorithms [49] (Figure 52).

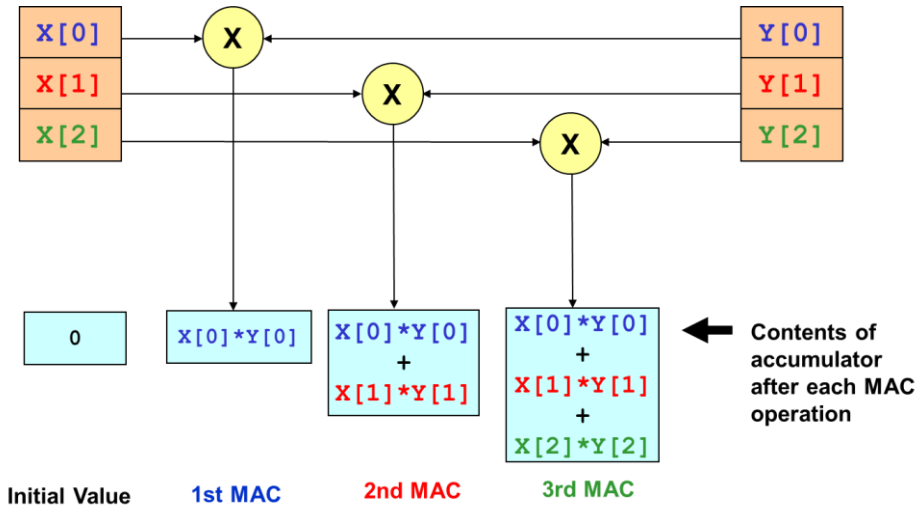


Figure 52: Multiply and Accumulate operations

The Tri-ALU can perform MAC operations in a single instruction. However, the CPU can perform this operation more efficiently and with a higher degree of parallelism, improving the overall performance of the computer for DSP applications.

Table 12: Tri-ALU can calculate MAC operations as follows.

Assembly language	Description
TLDA B 2	Load value 2 to register B
TLDA A 1	Load value 1 to register A
TRI A * B + FD , ACC ,FD	Accumulator = Feedback = A * B + Feedback
TSTA x“1C”	Send Accumulator value to output leds
HLT	Halt

4.3.5 PID Control

The proportional–integral–derivative (PID) controller is by far the most used control algorithm. This algorithm, or slight modifications of it, controls many feedback loops. It is put into practice in a variety of ways, such as a standalone controller, a component of a direct digital control package, or an element of a hierarchical distributed process control system [50].

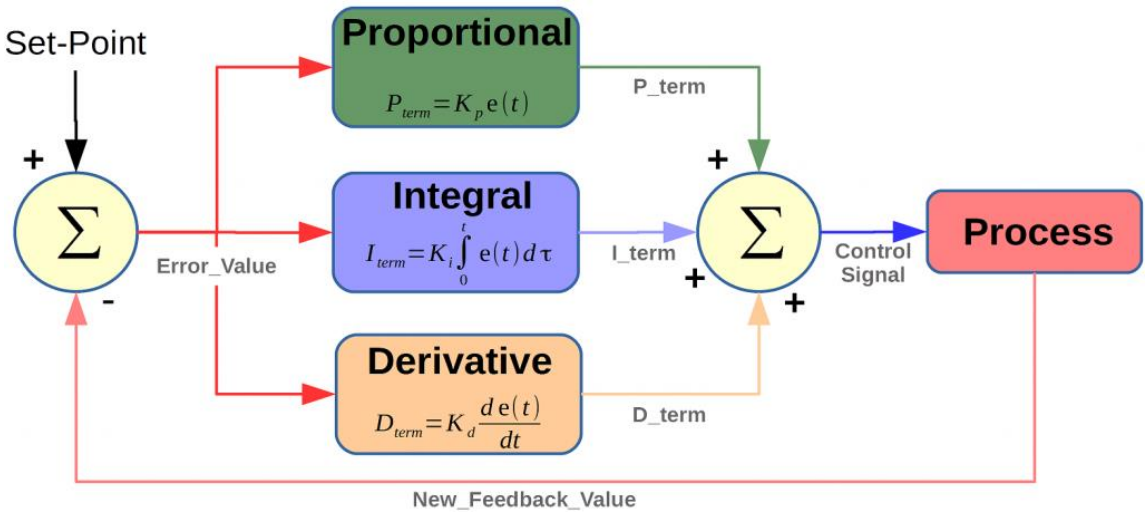


Figure 53: PID controller [36]

The Tri-ALU hybrid architecture calculates the control signal efficiently (software size, perform multiple operations in parallel) by mixing multiple architecture instructions.

Control signal in C language:

```
Control_Signal = Error_Value*Kp + Error_SumValues*Ki + (Error_Value - PreError_Value)*Kd;
```

Table 13: The Tri-ALU can calculate the control signal as follows.

Assembly language	Description	Operation Type
TPUSH [Error_Value]	Push Error value in stack	Stack
TPUSH [Kp]	Push Kp in stack	
MUL	Stack multiplication	
TPUSH [Error_SumValues]	Push Error_SumValues in stack	
TLDA Ki , Accumulator	Load Ki value to Accumulator register	Accumulator
MUL B	Accumulator = B * Accumulator	
ADD A	Accumulator = A + Accumulator	
TSTA [x'F1']	Store Accumulator value to memory location x'F1'	
VLDA 3	Vector load values Error_Value, PreError_Value, Kd from consecutive memory location	Vector
TRI A - B * Feedback	B = A – B * Feedback	Parallel
TLDA [x'F1'], Accumulator	Load vale from x'F1' to Accumulator	
TADD B	Accumulator = B + Accumulator	Accumulator
TSTA [x'F2']	Store Accumulator value to memory location x'F2'	

4.3.6 Hamming Weight

We require various definitions to examine the error detection and error repair capabilities of block codes in communication. The quantity of 1s in a binary block is what we refer to as the block's Hamming weight [51]. We write the Hamming weight of a binary block A as $\omega H(A)$.

The Hamming weight of [1001], for instance, is 2.

The Tri-ALU calculates the Hamming weight in a single instruction using continuous mode alongside shift and logic AND beside arithmetic Add operations.

Table 14: Tri-ALU Hamming weight software $\omega H(d7)$

Assembly language	Description
TPUSH 7	Load value 2 to register B
TPUSH 1	Load value 1 to register A
TLDA Feedback 7 MOD C	Load value 7 to register Feedback, set mod to continues
TRI A AND B + ACC, ACC - FD , >>A	Accumulator = A AND B + Accumulator Feedback -1, shift to right register A
TSTA x“1C”	Send Accumulator value to output leds
HLT	Halt

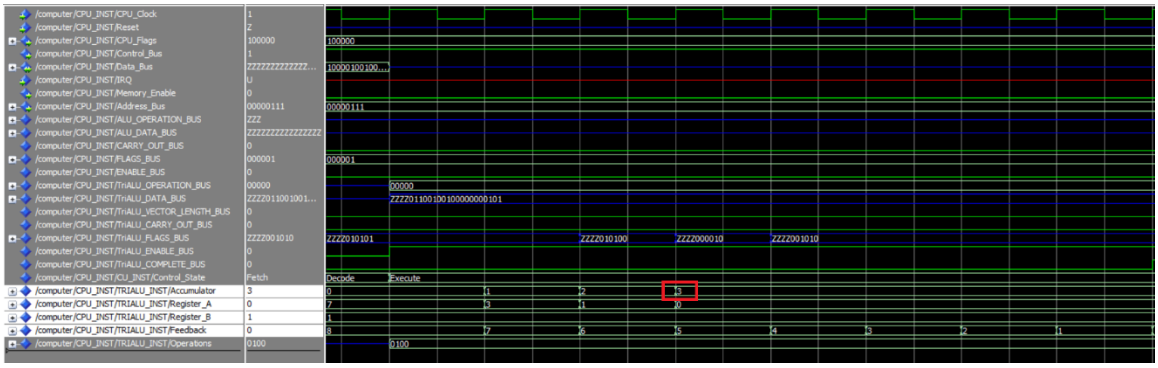


Figure 54: Hamming weight software result

4.4 Benchmark

In this section, we will apply a Fibonacci series function on X86 and RISC-V processors and the two ALU versions (conventional ALU and Tri-ALU) to find the ninth term of the series.

The Fibonacci sequence was created in the 13th century by the Italian mathematician Leonardo Fibonacci. The Fibonacci sequence's numbers do not correspond to any particular formula, although they do frequently have certain associations with one another. Starting with zero and one, the series of numbers increase slowly, with each number being

equal to the sum of the two numbers before it. For example, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, and 377 [52].

Fibonacci sequence rule:

$x_n = x_{n-1} + x_{n-2}$

where:

x_n is term number "n"

x_{n-1} is the previous term (n−1)

x_{n-2} is the term before that (n−2)

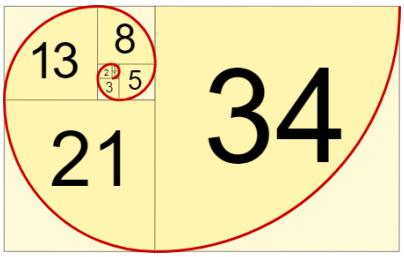


Figure 55: Fibonacci spiral

4.4.1 Conventional ALU Fibonacci Software

The ALU takes two arguments from Registers A and B and applies a mathematical operation to them. The result value goes to the Accumulator register, and the data is retrieved from memory or the register file (Figure 56).

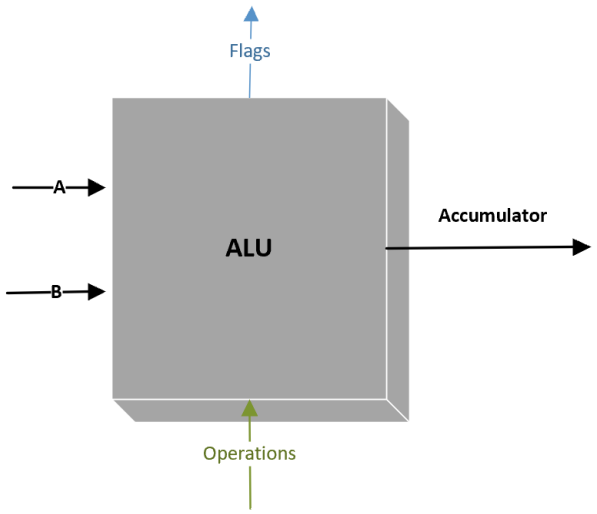


Figure 56: Conventional ALU structure

Table 15: Fibonacci function software

Machine language in HEX	Assembly language	Description
x"50100"	LDA A “1”	Load value from address 1 to register A
x"50301"	LDA B “3”	Load value from address 3 to register B
x"60000"	ADD A B	A + B
x"30100"	STA “1”	Store Accumulator value in address 1
x"50700"	LDA A “7”	Load value from address 7 to register A
x"50901"	LDA B “9”	Load value from address 9 to register B
x"6000d"	DEC B	Decrement B by 1
x"30900"	STA “9”	Store Accumulator value in address 9
x"a1907"	JMP A = ACC “19”	Jump to address 19 if A = Accumulator
x"50100"	LDA A “1”	Load value from address 1 to register A
x"50301"	LDA B “3”	Load value from address 3 to register B
x"30300"	STA “3”	Store Accumulator value in address 3
x"60000"	ADD A B	A + B
x"50700"	LDA A “7”	Load value from address 7 to register A
x"50901"	LDA B “9”	Load value from address 9 to register B
x"6000d"	DEC B	Decrement B by 1
x"30900"	STA “9”	Store Accumulator value in address 9
x"a0006"	JMP A < ACC “0”	Jump to address 19 if A < Accumulator
x"50302"	LDA ACC “3”	Load value from address 3 to Accumulator
x"31C00"	STA x“1C”	Send Accumulator value to output leds
x"f0000"	HLT	Halt
x"50102"	LDA ACC “1”	Load value from address 1 to Accumulator
x"31C00"	STA “1C”	Send Accumulator value to output leds

The TRI-ALU can perform those instructions in just one single instruction.

Table 16: Conventional ALU result

Number of instructions	20
Cycles	535
CPU time = instruction count * clock cycles per instructions * clock cycle time $= 20 \times 6 \times 2 \times 10^{-8}$	2.4 micro sec
Total power consumption = CPU time * Power = 2.76 microsec * 0.184 W	4.416×10^{-7} W/Sec

Note: For the Altera DE2 board, the clock cycle time is $\frac{1}{50\ MHz} = 2 \times 10^{-8}$ sec.

Based on the Altera Cyclone II datasheet, we can find the rated voltage as 4.6 v and rated current as 40 mA [53].

Then the power = voltage * current = 4.6V * 40 mA = 0.184 W.

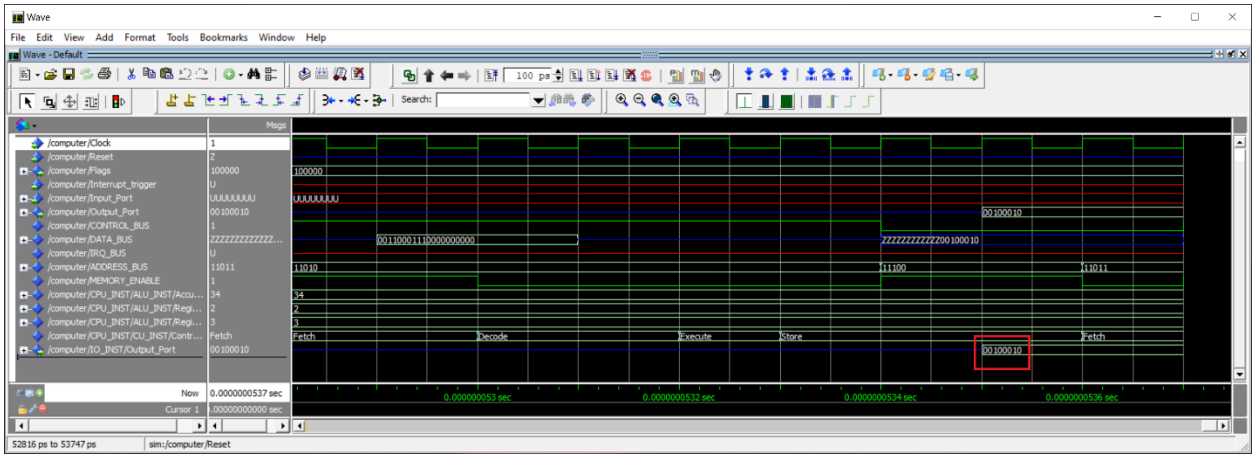


Figure 57: ALU RTL simulation

4.4.2 Tri-ALU Fibonacci Software

The Tri-ALU takes three arguments (A, B, and Feedback) and applies three independent mathematical operations to the arguments. The user can choose which variable they need in the operation and can select just one or two, or three variables.

The Feedback operation result automatically goes to the Feedback register. In this case, we will apply the decrement by 1 operation on the Feedback register, which is commonly used as a counter or index.

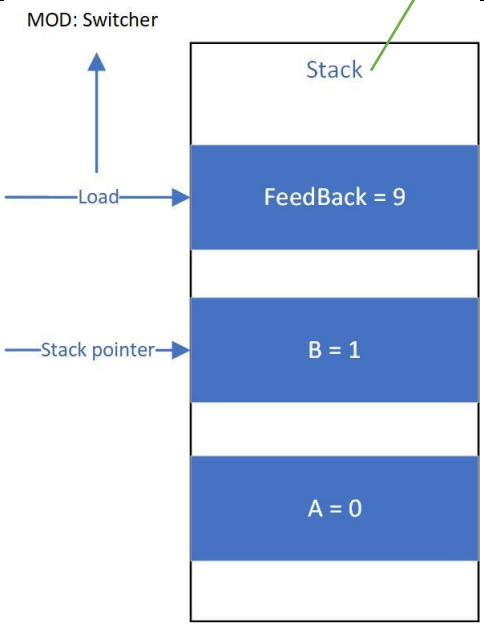
The result can go either to the Accumulator or the registers (A, B, and Feedback), but the Feedback operation result can overwrite the Feedback register. The data is retrieved from the memory or register files. The Tri-ALU can operate on different modes; in this program, we set the Tri-ALU to the switcher mode. The switcher mode can automatically save the operation result between Registers A and B, depending on the clock cycle.

The above technique is useful for calculating mathematical series or sequences, as it reduces access to memory or register files, which increases operation performance.

Table 17: Fibonacci function software (Tri-ALU)

The Tri-ALU performs stack operations to reduce software size.

Machine language in HEX	Assembly language	Description
x"C400"	TPUSH 0	Push 0 to register A
x"C401"	TPUSH 1	Push 1 to register B
x"80609"	TLDA Feedback 9 MOD S	Load value 9 to register Feedback and set mod to switcher
x"81605"	TRI A*B - 1	A + B and decrement Feedback by 1
x"A013F"	TJMP 1 < ACC “3”	Jump if Accumulator greater than 1 to address 3
x"3fe01"	TSTA x“FE”	Send Accumulator value to output leds
x"f0000"	HLT	Halt



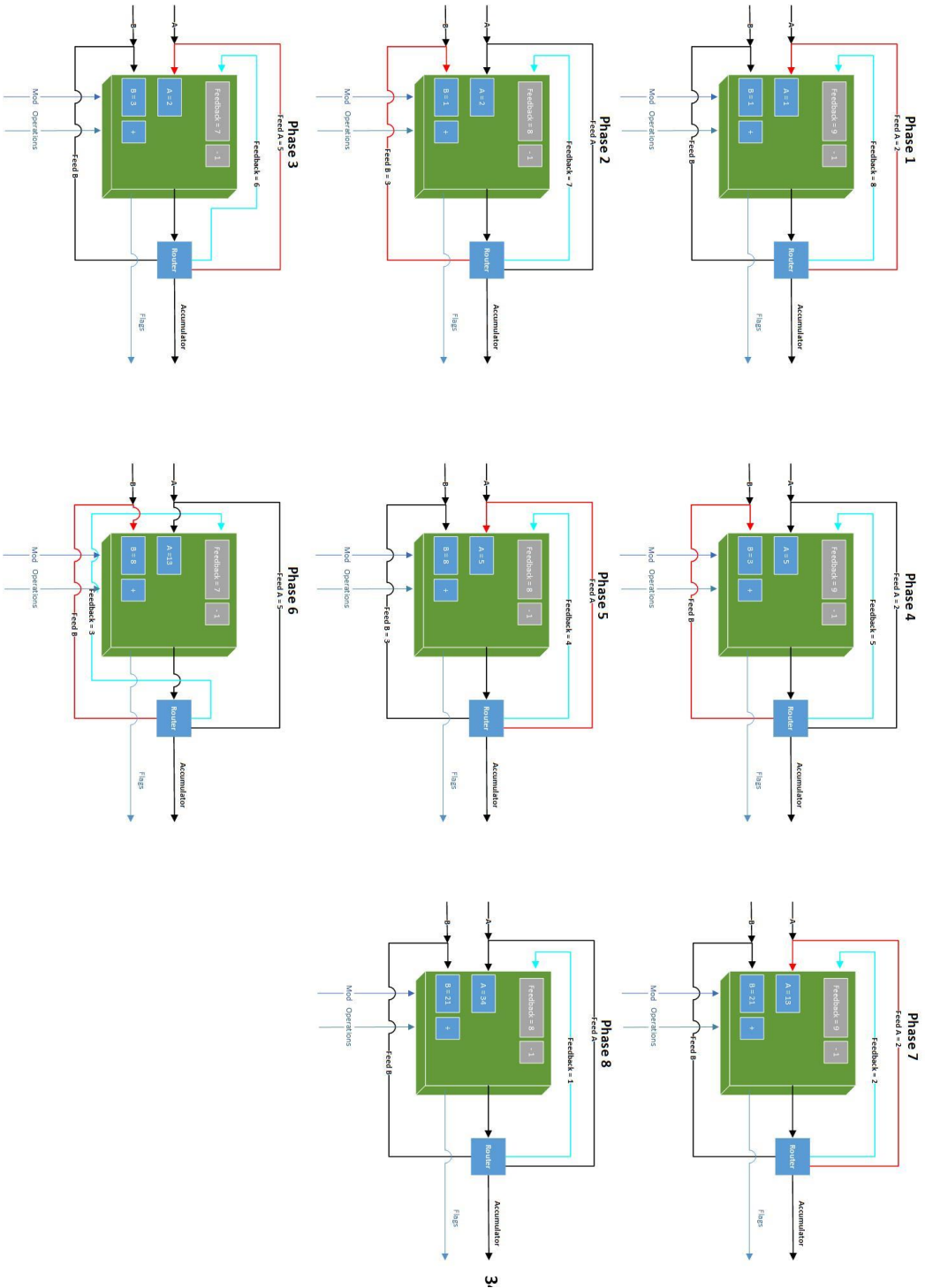


Figure 58: Tri-ALU operation phase

Table 18: Tri-ALU result

Number of instructions	6
Cycles	100
CPU time = instruction count * clock cycles per instructions * clock cycle time $= 6 \times 6 \times 2 \times 10^{-8}$	0.72 micro sec
Total power consumption = CPU time * Power = 0.72 microsec * 0.184 W	1.3248×10^{-7} W/Sec

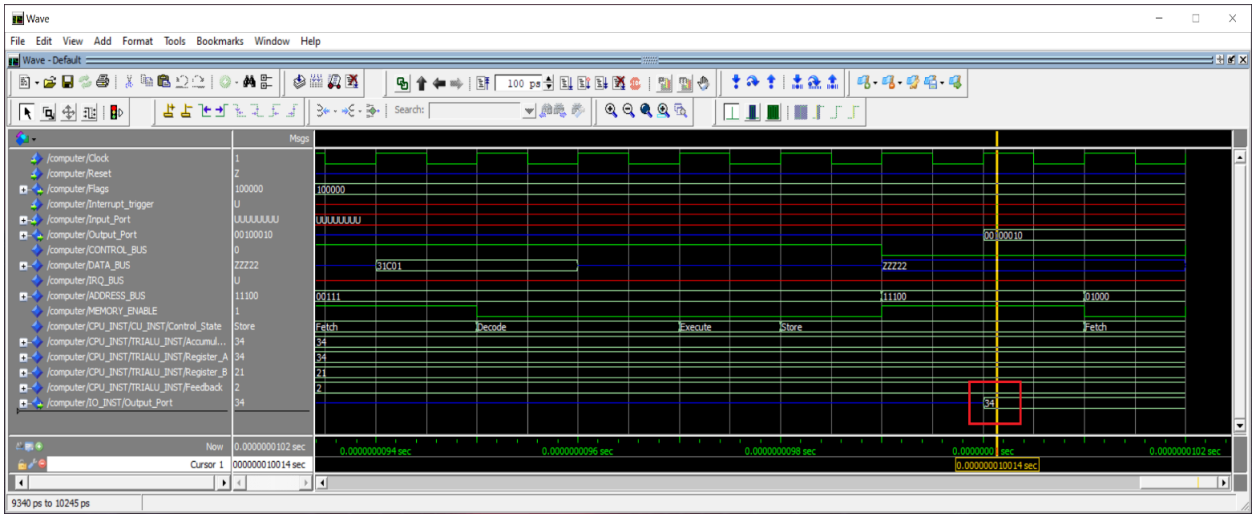


Figure 59: Tri-ALU RTL simulation

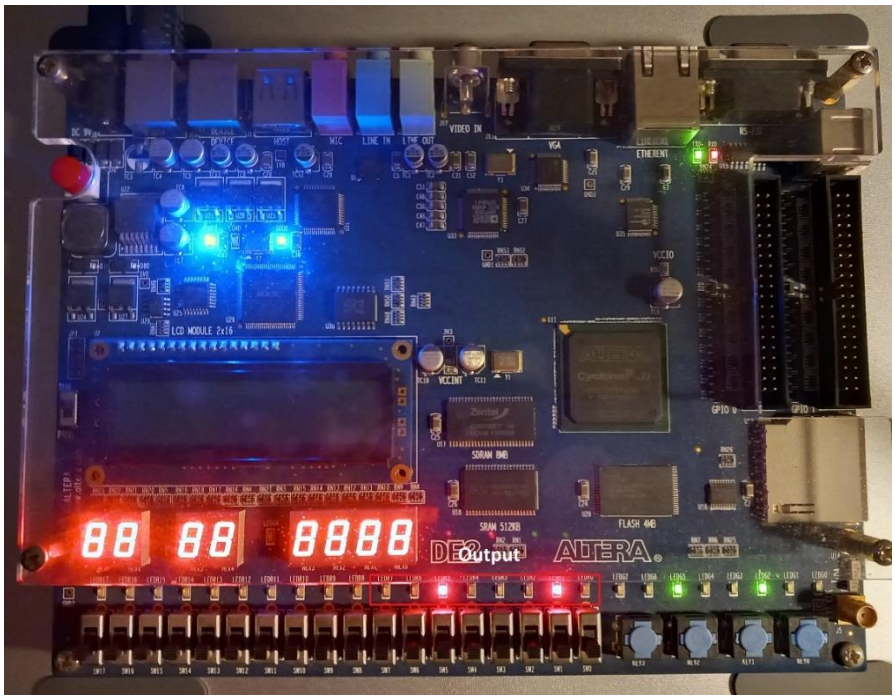
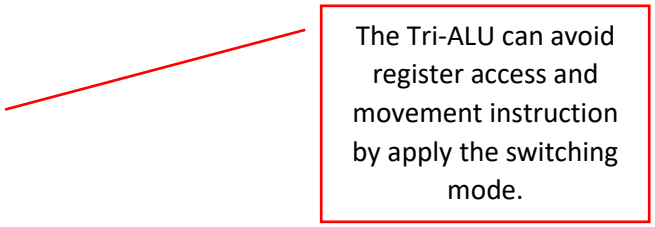


Figure 60: Output = 100010 = 34

4.4.3 X86 Fibonacci Software

```
org 100h
.DATA
ANS DW ?
.CODE
MAIN PROC
   MOV AX,@DATA
   MOV DS,AX
   MOV BX,0
   MOV DX,1
   MOV CX,8
   LABEL:
   ADD BX,DX
   MOV AX,BX
   MOV BX,DX
   MOV DX,AX
   LOOP LABEL
   MOV ANS,DX
   END MAIN
ret
```



X86 processor

Note: The researcher assumes the x86 instruction takes six cycles, and we use the Altera DE2 board's clock cycle time as $\frac{1}{50\ MHz} = 2 \times 10^{-8}$.

Table 19: X86 processor result

Number of instructions	9
Cycles	222
CPU time = instruction count * clock cycles per instructions * clock cycle time $= 9 \times 6 \times 2 \times 10^{-8}$	1.08 micro sec
Total power consumption = CPU time * Power = 1.08 microsec * 0.184 W	1.9872×10^{-7} W/Sec

4.4.4 RISC-V Fibonacci Software

```
.data
   n:  .word 47
.text
.globl main

main:

        li x2, 0            # Used to determine if n (x7) equals 0
        li x3, 1            # Used to determine if n (x7) equals 1
        li x5, 0            # First number
        li x6, 1            # Second number
        lw x7, 9             # Limit
        li x8, 1            # Counter
        beq x7, x2, DO        # If n == 0 then jump to DO (Which should print 0). Implements f(0) = 0
        beq x7, x3, WRITE      # if n == 1 then jump to WRITE (Which should print 1). Implements f(1) = 1

   LOOP:  beq x8, x7, EXIT       # Comparse the counter x8 which starts with 1 to n (limit). If x8 == x7 jump to EXIT
        add x4, x5, x6        # Add x5 to x6 and store in x4
        ori x5, x6, 0         # Assign the second number to my first number
        ori x6, x4, 0         # Assign the sum of x5 and x6 to my second number
        addi x8, x8, 1        # Add 1 to my counter
        j LOOP                # Jump to loop

   EXIT:
        li x17, 1            # Load constant 1 to x17
        add x10,x4,x0          # Add x4 (which contains the result after the above coe) to x10
        ecall               # Issue an SystemCall which prints an integer (Because of the 1 in x17)
        li x17, 5
        ecall
        li x17, 10
        ecall               # Reads an int from input console (Because of the 10 in x17)
   DO:
        li x4, 0            # load 0 in x10 (x10 will be used by the SysCall to print) and print
        add x10,x4,x0
        li x17, 1
        ecall
        li x17, 5
        ecall
        li x17, 10
        ecall

   WRITE:  li x4, 1             # load 1 in x10 and print
        add x10,x4,x0
        li x17,1
        ecall
        li x17, 5
        ecall
        li x17, 10
        ecall
```

The Tri-ALU can avoid register access and movement instruction by apply the switching mode.

The Tri-ALU can perform indexing operations alongside math operations.

Note: The researcher assumes the RISC-V instruction takes six cycles and use the Altera DE2 board's clock cycle time as $\frac{1}{50\ MHz} = 2 \times 10^{-8}$.

Table 20: RISC-V processor result

Number of instructions	11
Cycles	318
CPU time = instruction count * clock cycles per instructions * clock cycle time $= 11 \times 6 \times 2 \times 10^{-8}$	1.32 micro sec
Total power consumption = CPU time * Power = 1.32 microsec * 0.184 W	2.4288×10^{-7} W/Sec

4.4.5 Final Result

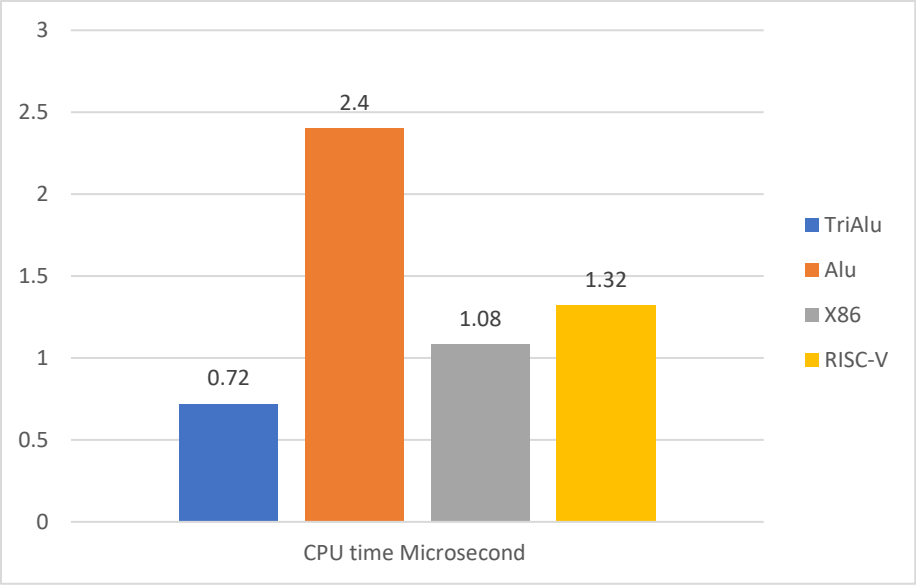


Figure 61: CPU time microsecond

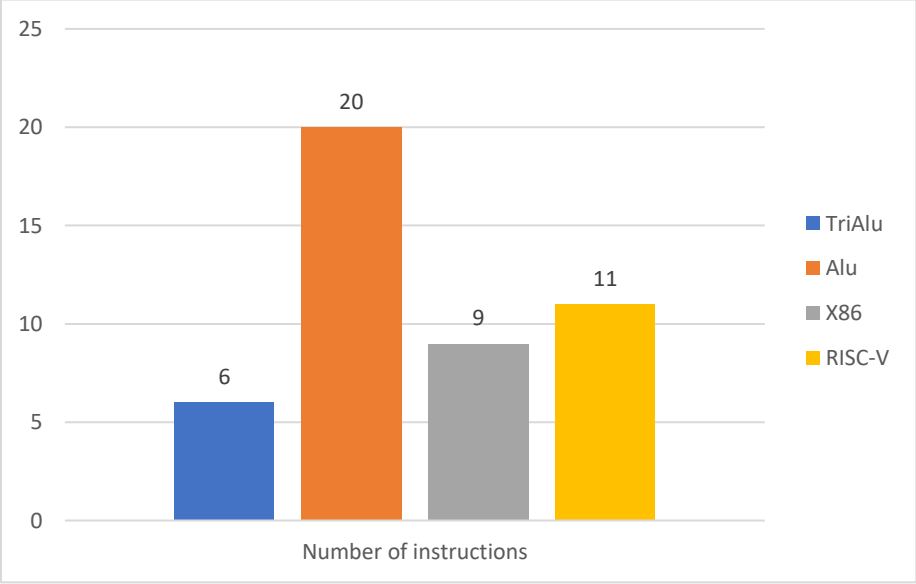


Figure 62: Number of instructions

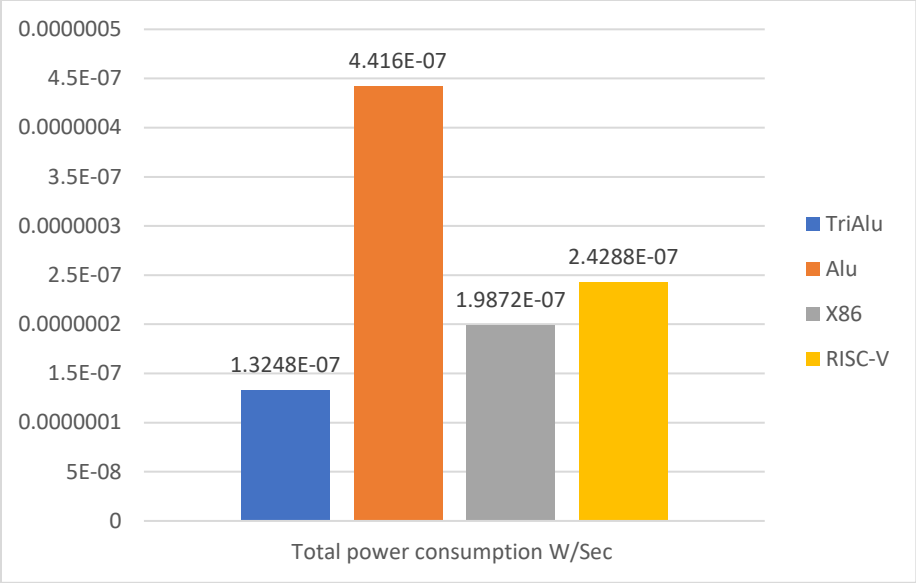


Figure 63: Total power consumption W/Sec

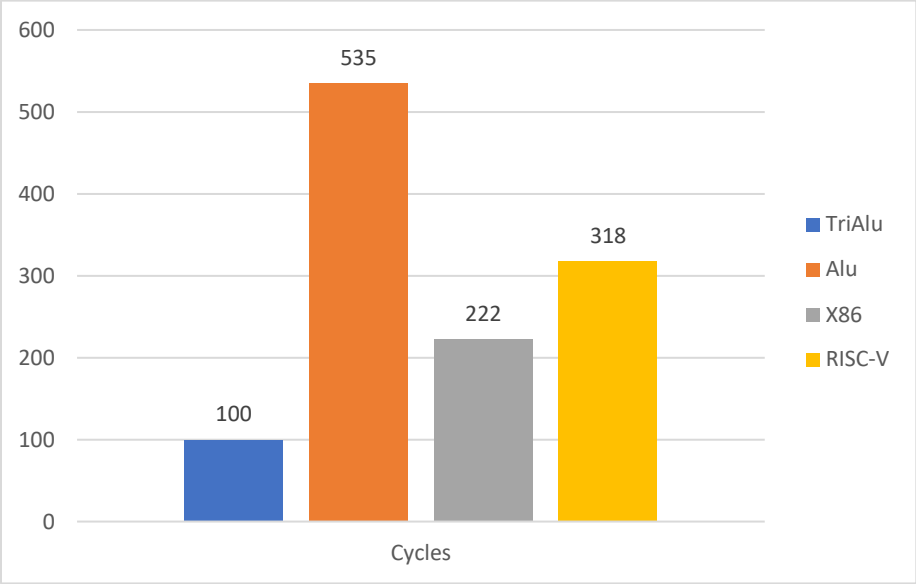


Figure 64: Total operation cycles

The figures above illustrate the significant performance difference result between the basic ALU and Tri-ALU. The basic ALU requires more instruction and memory access than the Tri-ALU, which impacts its performance. On the other hand, the Tri-ALU reduces access to memory by applying the switcher mode to transfer the operation result between Registers A and B.

A + B -> A First clock cycle

A + B -> B Second clock cycle

. . . and so on. This becomes useful when calculating mathematical series.

In this case, the Tri-ALU performs two operations simultaneously: Add A + B and decrement Feedback by 1.

The X86 processor needs to transfer data between registers manually to find the tenth Fibonacci term, which can reduce its performance. The Tri-ALU avoids this problem using the switcher technique. The RISC-V processor needs index operation instructions addi x8, x8, and 1, while the Tri-ALU performs math and indexing operations in single instruction TRI A*B -1. The benefit of using hybrid architecture arises here when the Tri-ALU uses the stack operation “PUSH” zero address instruction to reduce software size.

4.5 Summary

The application demonstrated how the Tri-ALU enhances the speed of computations. Moreover, the final benchmarking result revealed how the Tri-ALU could reduce power consumption and increase system performance among other architectures. The next chapter concludes our research, exposes its limitations, and discusses future work.

CHAPTER V.

CONCLUSION

This chapter summarizes the results obtained from this work. In addition, it lists some of the limitations and mentions features that could be upgraded in the Tri-ALU component in future work.

5.1 Conclusion

A hybrid Tri-ALU architecture design was proposed with unique design paradigms. The proposed Tri-ALU designs were verified using Altera Quartus II and ModelSim software. Both the proposed design and conventional ALUs were analyzed and compared in terms of the number of instructions, cycle counts, and CPU processing time. The simulation results illustrate that the proposed Tri-ALU design outperforms the X86 and RISC-V ALUs and conventional ALU designs. The Tri-ALU is a significant advancement in computer architecture. With its crossbred structures, the addition of four operations, and execution modes (switching, continuous), the Tri-ALU can improve data processing efficiency and ultimately improve a computer system's overall performance. Its benefits and applications in various fields make it a promising technology for future computing systems.

5.2 Limitations

As research is a never-ending process, this section explains the study limitations and constraints that might affect the findings and motivate other researchers to pursue further research. First, the study covered recent years by reviewing published papers in databases

and search engines, such as Scopus, Google Scholar, and ResearchGate, so this criterion could also possibly limit the inclusion of potentially valuable studies despite being older. Another limitation is that this study was limited to articles published in English; it is possible that helpful studies were overlooked due to the language barrier. Finally, the Tri-ALU requires longer instruction word lengths, as well as complex and expensive hardware to build.

5.3 Future Work

The scope of this work could be expanded by adding partial run-time reconfiguration (PR) aspects inside the Tri-ALU to adjust the inner execution Modes and operation. This enhances the Tri-ALU's capabilities to handle versatile domain applications. Most applications require floating-point operations, so upgrading the Tri-ALU to handle floating-point numbers will enhance computer computation. Moreover, Prefetch data and more data register makes a great impact on Tri-ALU performance.

Every computer architecture relies on compiler and compiler optimization to produce high-efficiency software that utilizes features in specific architecture. The researcher will develop a toolchain (Adoptive GCC cross compiler, linker, library) or assembler for the proposed architecture.

Through open collaboration, the open standard ISA known as RISC-V paves the way for a new era of processor innovation [54]. Integrating the Tri-ALU inside the RISC-V CPU and meager multiple RISC-V instructions in a single Tri-ALU instruction will increase performance. However, several modifications in the fetch instruction unit inside the RISC-

V CPU are needed. Hence, the researcher will build a softcore RISC-V processor with Tri-ALU.

We anticipate that Tri-ALU computation and its execution Modes will extend in the future with greater performance for DSP and machine learning applications, especially in embedded systems and IoT devices due to the performance's wide range of operations in parallelism, exceptional energy efficiency, and reducing software instruction and memory access.

References

[1] K. F. Sağlam Bedir N, "Design and Simulation of 64 Bit FPGA Based Arithmetic Logic Unit. Electrica," 2019.

[2] b. R. G. Plantz, INTRODUCTION TO COMPUTER ORGANIZATION, San Francisco: no strach press.

[3] A. B. Clive Max, Bebop BYTES Back An Unconventional Guide to Computers, Madison,AL: Doone, 1997.

[4] S. R. Sarangi, Basic Computer Architecture.

[5] K. M. ,. R. ,. W. B. : Guttag, "Three input arithmetic logic unit forming the sum of a first and a second boolean combination of the". North Central Expressway Patent 94308888.0 , 30 11 94 .

[6] C.-F. Kuo, Computer Architecture.

[7] D. A. P. John L. Hennessy, Computer Architecture: A Quantitative Approach, Elsevier, 2012.

[8] R. Huntley, "whats-next-for-the-microcontroller," eetimes, [Online]. Available: https://www.eetimes.eu/whats-next-for-the-microcontroller/.

[9] A. TEAM, "from-embedded-sensors-to-advanced-intelligence-driving-industry-4-0-innovation-with-tinyml," arduino, [Online]. Available: https://blog.arduino.cc/2022/02/10/from-embedded-sensors-to-advanced-intelligence-driving-industry-4-0-innovation-with-tinyml/.

[10] K. T. Hanna, "arithmetic-logic-unit-ALU," [Online]. Available: https://www.techtarget.com/whatis/definition/arithmetic-logic-unit-ALU#:~:text=What%20is%20an%20arithmetic%2Dlogic%20unit%20(ALU)%3F,a%20logic%20unit%20(LU)..

[11] "en.wikibooks.org," [Online]. Available: https://en.wikibooks.org/wiki/Microprocessor_Design/ALU.

[12] "teachcomputerscience," [Online]. Available: https://teachcomputerscience.com/arithmetic-logic-unit/.

[13] "study.com," [Online]. Available: https://study.com/academy/lesson/arithmetic-logic-unit-alu-definition-design-function.html.

[14] "computerhope," [Online]. Available: https://www.computerhope.com/jargon/a/alu.htm.

[15] M. J. H. M. B. R. H. H. A. I. A. K. Ali Haidar, "A Novel Neural Network Ternary Arithmetic Logic Unit," 2008.

[16] Z.-Q. W. W. L. T. Hui zhanga*, "The Design of Arithmetic Logic Unit Based on ALM," 2012.

[17] [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alm.htm.

[18] A. T. V. K. S. G. S. T. Rajit Ram Singh, "VHDL environment for floating point Arithmetic Logic Unit," 2011.

[19] D. K. ,. D. Maroju SaiKumar, "Design and Performance Analysis of Multiply-Accumulate (MAC) Unit," 2014.

[20] "arm.com," ARM, [Online]. Available: https://developer.arm.com/documentation/101726/4-0/Learn-about-the-Scalable-Vector-Extension--SVE-/What-is-the-Scalable-Vector-Extension-.

[21] R. A. Shendi, "RUN-TIME CUSTOMIZATION OF A SOFT-CORE CPU ON AN FPGA," UNIVERSITY OF MANCHESTER, 2015.

[22] J. L. H. David A. Patterson, Computer Organization and Design T H E H A R D W A R E / S O F T W A R E I N T E R F A C E F I F T H E D I T I O N, Oxford: ELSEVIER, 2014.

[23] F. A. B. Kai Hwang, Computer Architecture and parallel processing, McGraw hill.

[24] N. J. Kai Hwang, ADVANCED IIDHPIITIEII ARCHITECTURE Parallelism, Scalability, mgranunamuw Second Edition, NEW DELHI: Tata McC-iraw Hill Education Private Limited, 2011.

[25] E. Engheim, "itnext.io," ITNEXT, 2022. [Online]. Available: https://itnext.io/advantages-of-risc-v-vector-processing-over-x86-simd-c1b72f3a3e82?gi=52e3d51e2167.

[26] R. Chopra, Advanced Computer Architecture (A Practical Approach), Delhi: S. CHAND, 2009.

[27] N. P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," in *IEEE Transactions on Computers*, January 1990.

[28] A. &. S. O. &. C. F. &. G. M. &. R. T. &. P. M. &. V. M. Cristal, "Kilo-Instruction Processors: Overcoming the Memory Wall.," in *IEEE Micro 25(3):48 - 57*, 2005.

[29] W. Stallings, Computer Organization and Architecture Designing for Performance Tenth edition., Boston: pearson, 2016.

[30] S. S. Jadhav, Advanced Computer Architecture & Computing, Technical Publications Pune, 2008.

[31] "An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture," Philips Semiconductors.

[32] "nvidiaopenaitritonpytorch," 2023. [Online]. Available: https://www.semianalysis.com/p/nvidiaopenaitritonpytorch.

[33] P. Aiken, Microsoft Computer Dictionary, Fifth Edition, Washington: Microsoft Press, 2002.

[34] "computer organization von neumann architecture," 2022. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/.

[35] "aekt," [Online]. Available: https://kb.iu.edu/d/aekt.

[36] "Reset_(computing)," [Online]. Available: https://en.wikipedia.org/wiki/Reset_(computing)#:~:text=In%20a%20computer%20or%20data,usually%20in%20a%20controlled%20manner..

[37] "interrupt," [Online]. Available: https://www.tutorialspoint.com/what-is-interrupt-i-o-process#:~:text=An%20interrupt%20I%2FO%20is,the%20attention%20of%20the%20CPU..

[38] "computer_memory," [Online]. Available: https://www.tutorialspoint.com/computer_fundamentals/computer_memory.htm.

[39] D. A. G. Atul P. Godes, Microprocessors, Technical Publications Pune, 2003.

[40] "io-controllers," [Online]. Available: https://www.arrow.com/en/categories/controllers/peripheral-controllers/io-controllers#:~:text=An%20I%2FO%20controller%20connects,and%20can%20control%20many%20devices..

[41] J. G. a. M. Barr, Embedded Systems Dictionary.

[42] "system-bus," techopedia, [Online]. Available: https://www.techopedia.com/definition/2307/system-bus.

[43] A. J. Dr A.K. GAUTAM, Microprocessors and Applications, S.K. KATARIA & SONS, 2007.

[44] D. R. Dominick Rosato, COMPUTER-AIDED DESIGN.

[45] B. Wilkinson, Computer architecture design and preformance, Prentice Hall, 1992.

[46] J. CHEN, "neuralnetwork," [Online]. Available: https://www.investopedia.com/terms/n/neuralnetwork.asp.

[47] edublognss, "famous-mathematical-sequences-and-series," [Online]. Available: https://edublognss.wordpress.com/2013/04/16/famous-mathematical-sequences-and-series/#:~:text=(1)%20Fibonacci%20Series%3A%20Probably,%2C34%2C55%2C89%E2%80%A6.

[48] "factorial," [Online]. Available: https://www.britannica.com/science/factorial.

[49] "microchipdeveloper," [Online]. Available: https://microchipdeveloper.com/dsp0201:mac-overview#:~:text=The%20MAC%2C%20or%20%E2%80%9CMultiply%20And,in%20several%20other%20DSP%20algorithms..

[50] T. H. Karl J. Astrom, PID Controllers: Theory, Design, and Tuning, NC: Instrument Society of America, 1995.

[51] M. m. Simon haykin, Introduction to Analog & Digital Communications, Ontario: JOHN WILEY & SONS, INC, 2007.

[52] "fibonaccilines," [Online]. Available: https://www.investopedia.com/terms/f/fibonaccilines.asp.

[53] Altera, Cyclone II Device Family Data Sheet.

[54] "riscv.org," [Online]. Available: https://riscv.org/about/.

[55] Intel® Quartus® Prime Pro Edition User Guide: Design Optimization, Intel.

[56] C. Dubois. [Online]. Available: https://www.allaboutcircuits.com/news/intel-to-introduce-new-cpu-fpga-hybrid-chip-supported-by-acceleration-stack/.

[57] [Online]. Available: https://en.wikipedia.org/wiki/Tensor_Processing_Unit.

[58] "https://www.apple.com/sa/newsroom/2020/11/apple-unleashes-m1/," 2022. [Online].

[59] J. C. RIGDON, "Dicitionary of computer and internet terms," AUG 2016, p. 220.

[60] "semiengineering," [Online]. Available: https://semiengineering.com/knowledge_centers/integrated-circuit/ic-types/fpga/embedded-fpga-efpga/#:~:text=An%20eFPGA%20is%20an%20IP,without%20the%20cost%20of%20FPGAs..

[61] P. A. Laplante, DICTIONARY OF COMPUTER SCIENCE, ENGINEERING, and TECHNOLOGY, Washington, D.C.: CRC press, 2001.

[62] "https://www.investopedia.com/terms/f/fibonaccilines.asp," [Online].

[63] J. JORDAN, "intro-to-neural-networks," [Online]. Available: https://www.jeremyjordan.me/intro-to-neural-networks/.

[64] S. S, "Design of low power floating point multiplier with reduced switching activity in deep submicron technology".

[65] D. J. J. G. S. W. D. P. K. Y. CHRISTOFOROS KOZYRAKIS, "Hardware/Compiler Codevelopment for an Embedded Media Processor," *IEEE Xplore,* 2012.

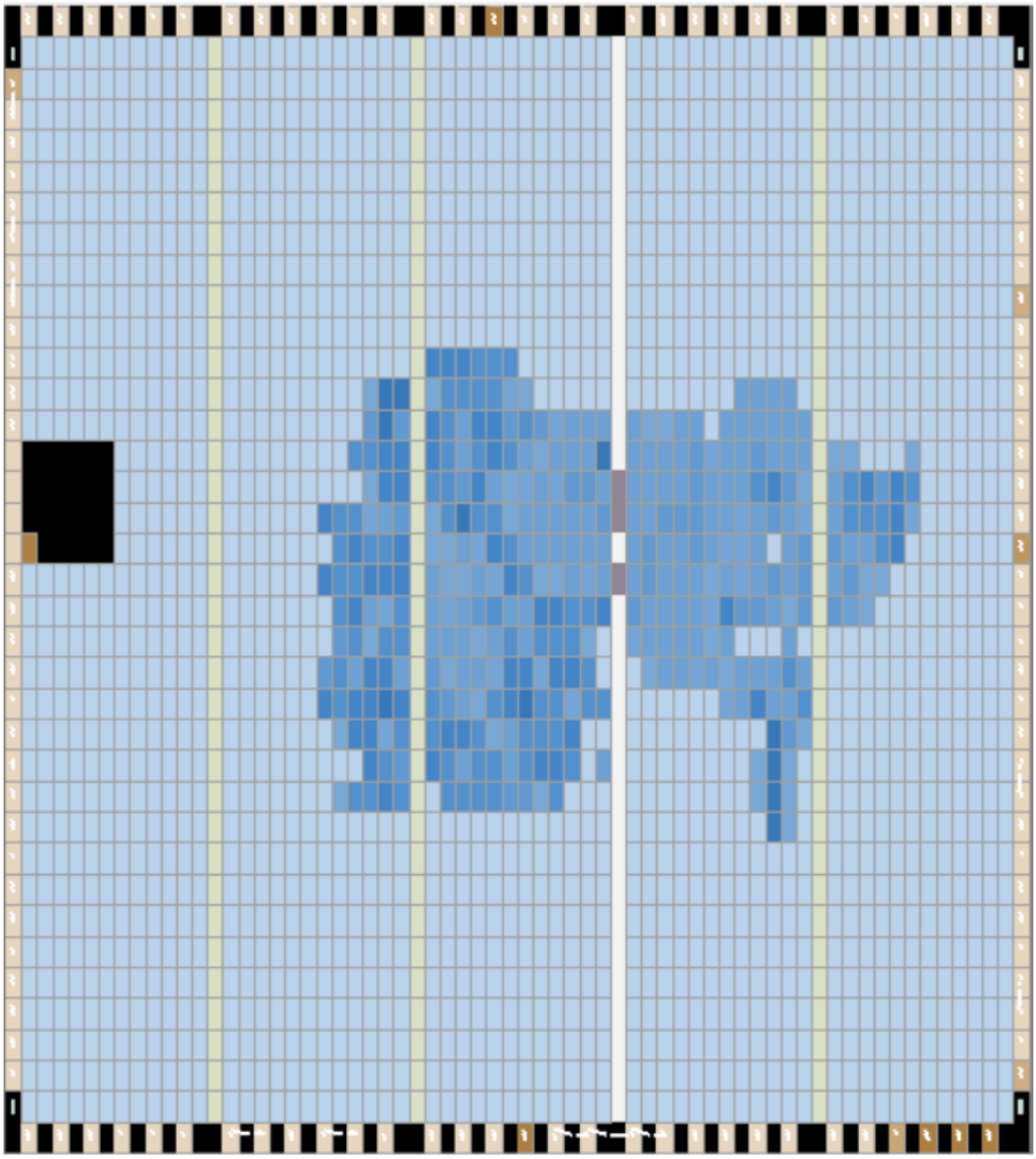
[66] "microcontrollerslab," [Online]. Available: https://microcontrollerslab.com/pid-controller-implementation-using-arduino/.

APPENDIX A.

Project report

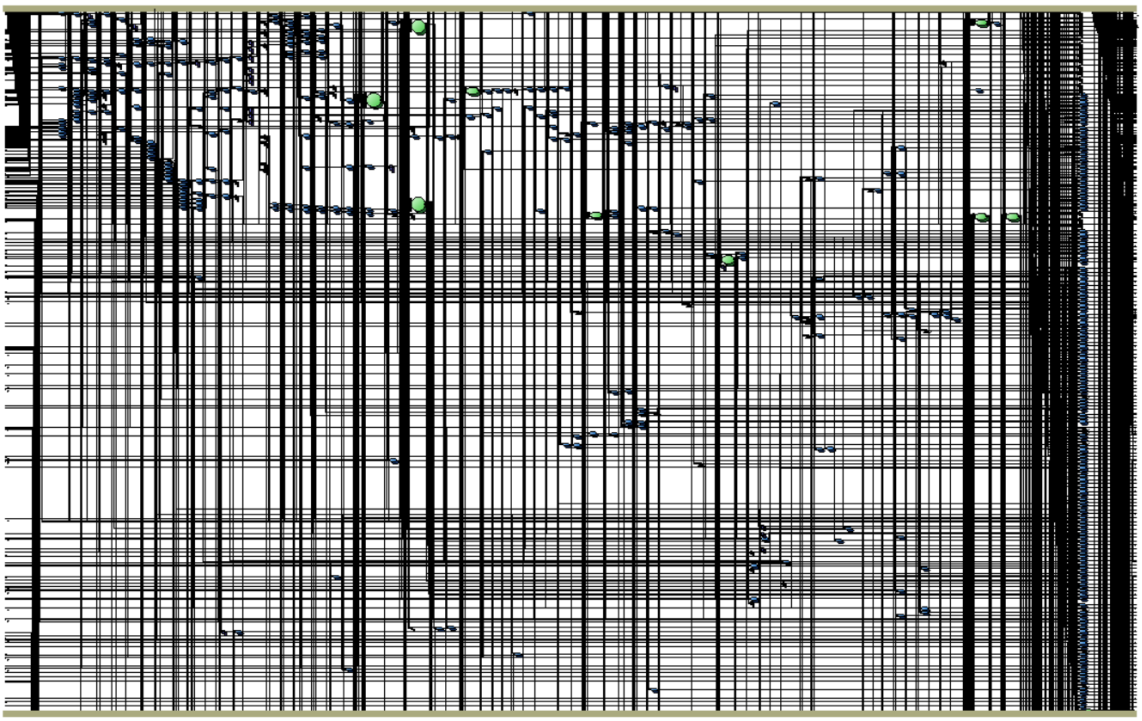
Flow Summary	
Flow Status	Successful - Fri May 26 18:56:42 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	CPU
Top-level Entity Name	Computer
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	5,161 / 33,216 (16 %)
Total combinational functions	4,299 / 33,216 (13 %)
Dedicated logic registers	2,630 / 33,216 (8 %)
Total registers	2630
Total pins	25 / 475 (5 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	6 / 70 (9 %)
Total PLLs	0 / 4 (0 %)

Chip planner



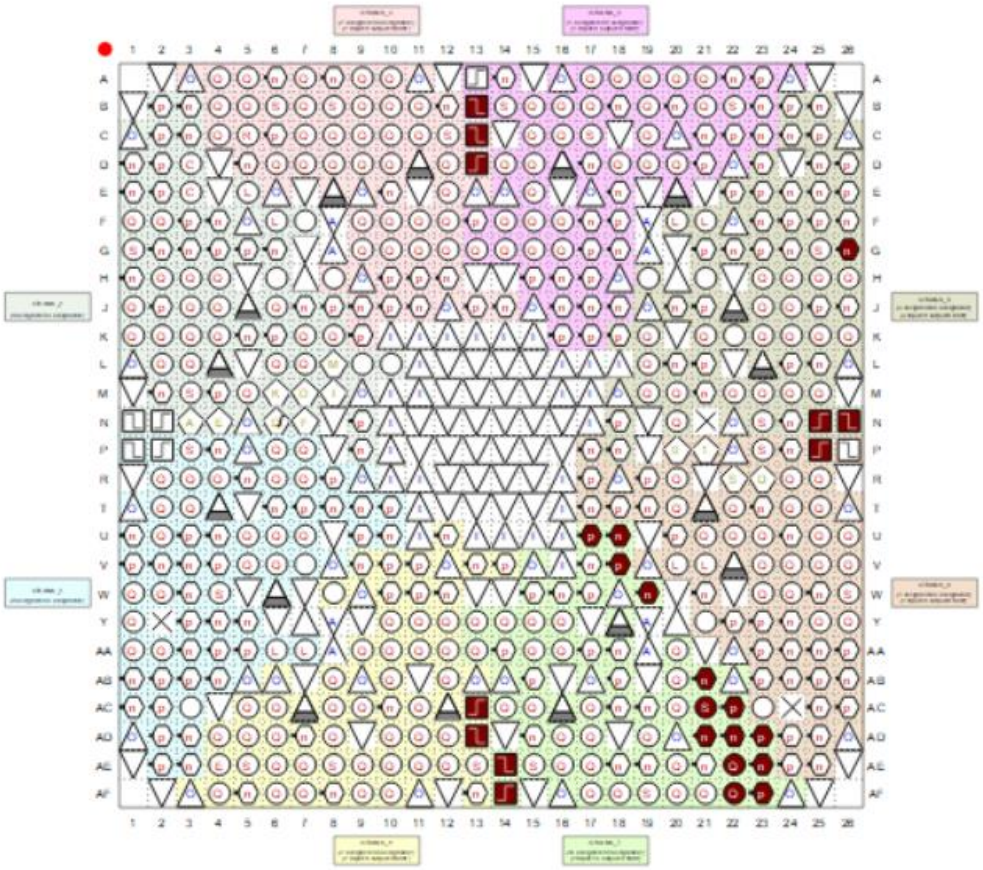
Technology Map

The Technology Map Viewer provides a visual representation of the hierarchical structure of atom primitives within a design, including device logic cells, I/O ports, and other components. In addition to displaying these elements, it allows you to explore the internal registers and look-up tables (LUTs) contained within logic cells (LCELLs), as well as the registers present in I/O atom primitives. This tool is particularly useful for examining the organization and composition of supported device families [55].



Tri-ALU technology map viewer post fitting

Pin Planner



Top view-Wire Bond Cyclone II – EP2C35F672C6

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Differential Pair
Clock	Input	PIN_D13	3	B3_N0	PIN_D13	3.3-V LV...default)		24mA (default)	
Flags[5]	Output	PIN_U17	7	B7_N0	PIN_U17	3.3-V LV...default)		24mA (default)	
Flags[4]	Output	PIN_U18	7	B7_N0	PIN_U18	3.3-V LV...default)		24mA (default)	
Flags[3]	Output	PIN_V18	7	B7_N0	PIN_V18	3.3-V LV...default)		24mA (default)	
Flags[2]	Output	PIN_W19	7	B7_N0	PIN_W19	3.3-V LV...default)		24mA (default)	
Flags[1]	Output	PIN_AF22	7	B7_N0	PIN_AF22	3.3-V LV...default)		24mA (default)	
Flags[0]	Output	PIN_AE22	7	B7_N0	PIN_AE22	3.3-V LV...default)		24mA (default)	
Input_Port[7]	Input	PIN_C13	3	B3_N0	PIN_C13	3.3-V LV...default)		24mA (default)	
Input_Port[6]	Input	PIN_AC13	8	B8_N0	PIN_AC13	3.3-V LV...default)		24mA (default)	
Input_Port[5]	Input	PIN_AD13	8	B8_N0	PIN_AD13	3.3-V LV...default)		24mA (default)	
Input_Port[4]	Input	PIN_AF14	7	B7_N1	PIN_AF14	3.3-V LV...default)		24mA (default)	
Input_Port[3]	Input	PIN_AE14	7	B7_N1	PIN_AE14	3.3-V LV...default)		24mA (default)	
Input_Port[2]	Input	PIN_P25	6	B6_N0	PIN_P25	3.3-V LV...default)		24mA (default)	
Input_Port[1]	Input	PIN_N26	5	B5_N1	PIN_N26	3.3-V LV...default)		24mA (default)	
Input_Port[0]	Input	PIN_N25	5	B5_N1	PIN_N25	3.3-V LV...default)		24mA (default)	
Interrupt_trigger	Input	PIN_B13	4	B4_N1	PIN_B13	3.3-V LV...default)		24mA (default)	
Output_Port[7]	Output	PIN_AC21	7	B7_N0	PIN_AC21	3.3-V LV...default)		24mA (default)	
Output_Port[6]	Output	PIN_AD21	7	B7_N0	PIN_AD21	3.3-V LV...default)		24mA (default)	
Output_Port[5]	Output	PIN_AD23	7	B7_N0	PIN_AD23	3.3-V LV...default)		24mA (default)	
Output_Port[4]	Output	PIN_AD22	7	B7_N0	PIN_AD22	3.3-V LV...default)		24mA (default)	
Output_Port[3]	Output	PIN_AC22	7	B7_N0	PIN_AC22	3.3-V LV...default)		24mA (default)	
Output_Port[2]	Output	PIN_AB21	7	B7_N0	PIN_AB21	3.3-V LV...default)		24mA (default)	
Output_Port[1]	Output	PIN_AF23	7	B7_N0	PIN_AF23	3.3-V LV...default)		24mA (default)	
Output_Port[0]	Output	PIN_AE23	7	B7_N0	PIN_AE23	3.3-V LV...default)		24mA (default)	
Reset	Input	PIN_G26	5	B5_N0	PIN_G26	3.3-V LV...default)		24mA (default)	

Pins Assigned

APPENDIX B.

CPU instruction set

Instruction (Assembly code)	Instruction code (OP-CODE)	Operation
NOP	0000	No operation
ADDI	0001	ADD immediate value
SUBI	0010	SUB immediate value
STA	0011	Store accumulator value in memory
MOV	0100	Load value to specific register
LDA	0101	Load value from address in the memory to specific register
ADR	01100000	Add current registers values A+B
SUR	01100001	Sub current registers values A-B
MUL	01100010	MUL multiple current registers values A*B
AND	01100100	AND current registers values A and B
OR	01100101	OR current registers values A or B
XOR	01100110	XOR current registers values A xor B
NOT	01100111	NOT current register value A
INC []	01101000	Increment specific register by one
DEC []	01101001	Decrement specific register by one
SHR []	01101010	Shift to right specific register
SHL []	01101011	Shift to lift specific register
JMP	1010	Jump to specific register
RTS	1011	Return
INM	1101	Set interrupts mask 1 enable interrupt, 0 disable interrupt
HLT	1111	HALT
TVADD []	0111000	Vector addition
TVG []	0111010	Mov greater value from A,B,Feedback to Accumulator
TVS []	0111011	Mov smaller value from A,B,Feedback to Accumulator
TVXOR	0111100	Vector XOR
TVLDA []	0111110	Vector Load
TRI {,,} [,,]	1000	MIMD
TADD []	10010100	Add specific register to Accumulator
TSUB []	10010101	Sub specific register to Accumulator
TAND []	10010110	AND specific register to Accumulator
TOR []	10010111	OR specific register to Accumulator
TPUSH ()	11000100	Push value in Tri-ALU stack
TPOP	11000101	Pop value from Tri-ALU stack
TADD	11000110	Add stack values

TSUB	11000111	Sub stack values
TMUL	11001100	Multiplication stack values
TDIV	11001101	Division stack values
TAND	11001110	AND stack values
TOR	11001111	OR stack values

Project VHDL Code

Memory component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;

use IEEE.STD_LOGIC_TEXTIO.ALL;

library STD;
use STD.TEXTIO.ALL;

entity Memory is
generic (
Address_Bus_Size : integer  := 7
);
port(
Clock : in std_logic;
Reset: in std_logic:='Z';
Read_or_Write    : in std_logic;
Data_in_out : inout std_logic_vector(31 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- 32 bit data bus
Enable : in std_logic:='Z';
Address_in : in std_logic_vector(Address_Bus_Size downto 0)
);
end entity;


architecture behave of Memory is

type Memory_Architecture is array(0 to 256) of std_logic_vector(7
downto 0); -- 250 Address one byte word 7 Address for IO device

signal Memory_Data:Memory_Architecture :=(others=>(others=>'0'));

begin

process(Clock)
begin
```

```
if rising_edge(Clock) then
if Reset='0' then
-- TRIALU Fibonacci using TriALU  with Switcher Mode and stack
push
--Memory_Data(0) <= x"C4"; -- Push 0
--Memory_Data(1) <= x"00";
--Memory_Data(2) <= x"C4"; -- Push 1
--Memory_Data(3) <= x"01";
--Memory_Data(4) <= x"80"; -- load
--Memory_Data(5) <= x"60";
--Memory_Data(6) <= x"90";
--Memory_Data(7) <= x"81"; -- Add a+b  --feedback
--Memory_Data(8) <= x"60";
--Memory_Data(9) <= x"50";
--Memory_Data(10) <= x"A0"; -- compare
--Memory_Data(11) <= x"10";
--Memory_Data(12) <= x"7f";
--Memory_Data(13) <= x"3f"; -- store output
--Memory_Data(14) <= x"e0";
--Memory_Data(15) <= x"10";
--Memory_Data(16) <= x"f0"; -- hlt
--Memory_Data(17) <= x"00";
--Memory_Data(18) <= x"00";
---------------------------------

--Counter-----------------------------------------------------
Memory_Data(0) <= x"51"; -- load from location 19
Memory_Data(1) <= x"30";
Memory_Data(2) <= x"00";
Memory_Data(3) <= x"00"; -- 00 gap
Memory_Data(4) <= x"60"; -- A++
Memory_Data(5) <= x"00";
Memory_Data(6) <= x"80";
Memory_Data(7) <= x"31"; -- store in location 19
Memory_Data(8) <= x"30";
Memory_Data(9) <= x"00";
Memory_Data(10) <= x"3f"; -- output
Memory_Data(11) <= x"e0";
Memory_Data(12) <= x"00";
Memory_Data(13) <= x"f0"; -- sleep
Memory_Data(14) <= x"00";
Memory_Data(15) <= x"70";
Memory_Data(16) <= x"a0"; -- jump
Memory_Data(17) <= x"00";
Memory_Data(18) <= x"00";
Memory_Data(19) <= x"01";
-----------------------------------------
Data_in_out <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
```

```
elsif unsigned(Address_in) <= 250 and Enable = '1' then -- enable
-> memory mapped >= 256 IO

if Read_or_Write = '0' then -- write
Memory_Data(to_integer(unsigned(Address_in)))<=Data_in_out(7
downto 0);

elsif Read_or_Write = '1' then -- read
Data_in_out <=
Memory_Data(to_integer(unsigned(Address_in)))&Memory_Data(to_inte
ger(unsigned(Address_in)+1))&Memory_Data(to_integer(unsigned(Addr
ess_in)+2))&Memory_Data(to_integer(unsigned(Address_in)+3));
else
Data_in_out<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end if;
else
Data_in_out<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end if;
end if;
end process;

end behave;
```

Control unit component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--use IEEE.STD_LOGIC_ARITH.ALL;

entity Control_Unit is
generic (
Address_Bus_Size : integer  := 7;
Board_frequency : integer :=  27e6     -- 50MHz oscillator for
altera DE2 board
);
Port (
Clock: in std_logic;
Reset: in std_logic:='Z';
Control_Flags : out std_logic_vector(5 downto 0):="101000";
--control flags
-- Empty Stack bit[5],carry bit[4], program counter =0 bit[3],
Halt bit[2], stackoverflow error bit[1], division by zero bit[0]

--Memory Handling
Request_Address: out std_logic_vector(Address_Bus_Size downto
0):="ZZZZZZZZ";-- Memory Address
Data_in_out_Memory : inout std_logic_vector(31 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";-- from and to Memory
Memory_Control : out std_logic:='Z';  -- 1 for read 0 for write
Memory_Enable : out std_logic:='Z';
--IO Handling
IRQ : in std_logic;

--ALU Handling
ALU_Operation: out std_logic_vector(2 downto 0):="ZZZ";-- ALU
Operations
Data_in_out_ALU : inout std_logic_vector(15 downto
0):="ZZZZZZZZZZZZZZZZ";-- from and to ALU
ALU_Enable : out std_logic:='Z';
ALU_Carry_Out : in std_logic;
ALU_Flags : in std_logic_vector(5 downto 0);
-- ALU flags
--  division by zero bit[5], A>B bit4, A<B bit3, A > Acc bit2, A
< Acc bit1, zero flags bit0
```

```
--TriALU Handling
TriALU_Operation: out std_logic_vector(4 downto 0):="ZZZZZ";--
ALU Operations
Data_in_out_TriALU : inout std_logic_vector(23 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZ";-- from and to ALU
TRIALU_Vector_Length : out std_logic:='0';
TriALU_Enable : out std_logic:='Z';
TriALU_Carry_Out : in std_logic;
TriALU_Flags : in std_logic_vector(9 downto 0);
TriALU_Complete : in std_logic
-- TriALU flags
--Sackoverflow bit[9], FeedBack > limit bit[8], FeedBack < limit
bit[7], FeedBack = limit bit[6], division by zero bit[5], A>B
bit4, A<B bit3, A > Acc bit2, A < Acc bit1, zero flags bit0
);
end Control_Unit;

architecture Behavioral of Control_Unit is
-----------------------------------------------STACK-------------
---------------------------------
type Stack is array(0 to 8) of
std_logic_vector(Address_Bus_Size+8 downto 0);  -- stck contains
address and accumulator value before CALL
signal Call_Stack : Stack := (others=>(others=>'0'));-- use it
when jump to save current program counter location and
accumulator value
signal Returned_Address_Accumulator :
std_logic_vector(Address_Bus_Size+8 downto 0) :=(others => '0');
signal Stack_Pointer : integer range 0 to 8 := 0;
-----------------------------------------------------------------
---------------------------------
-----------------------------------------------CONTROL-----------
---------------------------------
signal Program_Counter: std_logic_vector(Address_Bus_Size downto
0):=(others => '0');
signal Memory_Address_Register: std_logic_vector(Address_Bus_Size
downto 0):=(others => '0');     --          8bit     256 Addresss
signal Memory_Buffer_Register: std_logic_vector(27 downto
0):="000000000000000000000000000";
signal Instruction_Register: std_logic_vector(3 downto
0):="0000";

type t_State is (Fetch, Decode, Fetch_Operand, Execute, Store);
signal Control_State : t_State:=Fetch;

--Control State counter
signal Fetch_Counter: unsigned(1 downto 0) := (others => '0');
```

```
--signal Decode_Counter: unsigned(1 downto 0) := (others => '0');
signal Store_Counter: unsigned(1 downto 0) := (others => '0');
signal Execute_Counter: unsigned(1 downto 0) := (others => '0');
signal Fetch_Operand_Counter: unsigned(1 downto 0) := (others =>
'0');
signal ALU_Operation_Counter: unsigned(1 downto 0) := (others =>
'0');
signal TriALU_Operation_Counter: unsigned(1 downto 0) := (others
=> '0');

signal HALT_Counter: unsigned(63 downto 0) := (others => '0');--
64 bits to hold big frequancy value ---------------------------!
signal Total_Halt_Time: unsigned(63 downto 0) := (others => '0');

signal Register_Select : std_logic_vector(1 downto 0):="ZZ";
-------------------------------------------------------------------
---------------------------------

---------------------------------------Interrupts  H/S----------
-----------------------------------
signal Software_INT_Mask : std_logic_vector(5 downto
0):="000000"; -- Software Interrupts mask  which can be disabled
or enabled. "Hardware interrupt INTR can't be disable"
--TriALU stackoverflow bit[5], carry bit[4], program counter =0
bit[3], Halt bit[2], stackoverflow error bit[1], division by zero
bit[0]
type Vector_Table is array(0 to 6) of
std_logic_vector(Address_Bus_Size downto 0);  -- Vector table
signal Interrupts_Vector_Table : Vector_Table :=
("01100000","01011000","01010000","01001000","00111000","10000000
","11000000");-- change the value when you change the memory
size-----------------------------------!!!!!!!!
----division by zero 0----stackoverflow 1----Halt 2----program
counter 3----carry bit 4 Non vectored----5 Hardware interrupt
INTR vectored 6----TriALU stackoverflow ---------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--
begin

Control_Process:process(Clock)-----------------------------------
COONTROL PROCESS-------------------------------------------------
-----------------------------------

procedure Emergency_Push(signal Address_Accumulator : in
std_logic_vector(Address_Bus_Size+8 downto 0)) is  -- Push
Address inside address stack in Emergency situation!!!
begin
```

```
Call_Stack(8)(Address_Bus_Size+8 downto 0) <=
Address_Accumulator;
Stack_Pointer <= 8;
end procedure Emergency_Push;
-----------------------------------------------------------------
-----------------------------------------------------------------
-------------------------
procedure Push_Address_Accumulator(signal Address_Accumulator :
in std_logic_vector(Address_Bus_Size+8 downto 0)) is  -- Push
Address inside address stack
begin
if stack_Pointer <= 7 then
Call_Stack(Stack_Pointer)(Address_Bus_Size+8 downto 0) <=
Address_Accumulator;
Stack_Pointer <= Stack_Pointer + 1;
Control_Flags(1) <= '0'; --Stackoverflow flag
Control_Flags(5) <= '0'; --Empty stack flag
else -- ERROR stackoverflow
Control_Flags(1) <= '1';-- stackoverflow error flag
if Software_INT_Mask(1) = '1' then
Emergency_Push(Address_Accumulator);
program_Counter <= Interrupts_Vector_Table(1);
Control_Flags(1)<='0';-- Clear the flag
end if;
end if;
end procedure Push_Address_Accumulator;
-----------------------------------------------------------------
-----------------------------------------------------------------
procedure Pop_Address_Accumulator  is  -- Pop Address from
address stack
begin
if stack_Pointer > 0 then
Stack_Pointer <= Stack_Pointer - 1;
Returned_Address_Accumulator <= Call_Stack(Stack_Pointer-1);
if((Stack_Pointer - 1) =0) then
Control_Flags(5) <= '1';-- Empty stack flag
end if;
elsif stack_Pointer = 0 then
Control_Flags(5) <= '1';-- Empty stack flag
Returned_Address_Accumulator <= Call_Stack(Stack_Pointer);
end if;
end procedure Pop_Address_Accumulator;
-----------------------------------------------------------
procedure Carry_Interrupt (signal Address_Accumulator : in
std_logic_vector(Address_Bus_Size+8 downto 0)) is    -- Carry
interrupt mechanism
begin
```

```
if ALU_Carry_Out = '1'  then
Control_Flags(4)<='1';
if Software_INT_Mask(4) = '1' then
Push_Address_Accumulator(Address_Accumulator);
program_Counter <= Interrupts_Vector_Table(4);
Control_Flags(4)<='0';-- Clear the flag
end if;
else
Control_Flags(4)<='0';
end if;
end procedure Carry_Interrupt;
-------------------------------------------------------------
procedure division_by_zero_Interrupt (signal Address_Accumulator
: in std_logic_vector(Address_Bus_Size+8 downto 0)) is --
division by zero error mechanism
begin

if ALU_Flags(5) = '1'  then
Control_Flags(0)<='1';
if Software_INT_Mask(0) = '1' then
Push_Address_Accumulator(Address_Accumulator);
program_Counter <= Interrupts_Vector_Table(0);
Control_Flags(0)<='0';-- Clear the flag
end if;
else
Control_Flags(0)<='0';
end if;
end procedure division_by_zero_Interrupt;
-------------------------------------------------------------------
-----------
-------------------------------------------------------------
procedure Hardware_IRQ (signal Address_Accumulator : in
std_logic_vector(Address_Bus_Size+8 downto 0)) is -- hardware
interrupt   --------TODO!!!!!
begin
if IRQ = '1'  then
Push_Address_Accumulator(Address_Accumulator);
program_Counter <= Interrupts_Vector_Table(5);
end if;
end procedure Hardware_IRQ;
-------------------------------------------------------------------
-----------
begin


if rising_edge(Clock) then
-----------------------RESET----------
if Reset='0' then
```

```
Memory_Enable <= '0';
Program_Counter<=(others => '0');-- Pc=0  "reset"
Call_Stack <= (others=>(others=>'0'));
Stack_Pointer <= 0;
Control_Flags<= "101000";
Fetch_Counter<= (others => '0');
--   Decode_Counter<= (others => '0');
Store_Counter<= (others => '0');
Execute_Counter<= (others => '0');
Fetch_Operand_Counter<= (others => '0');
HALT_Counter<= (others => '0');
Control_State   <= Fetch;
Request_Address <= (others => '0');
Memory_Control <= 'Z';
ALU_Operation <="000";
Data_in_out_ALU <= "0000000000000000";
ALU_Enable <= '0';
TriALU_Operation <="00000";
Data_in_out_TriALU <= "000000000000000000000000";
TriALU_Enable <= '0';
if Software_INT_Mask(3)= '1' then
program_Counter <= Interrupts_Vector_Table(3);
Control_State   <= Fetch;
Control_Flags<= "100000";  -- Clear the flag
end if;
----------------------------------------
else
Control_Flags(3)<='0';
Control_Flags(4)<=ALU_Carry_Out;
-- Start counter 10
case Control_State is----------------------Control finite state
machine
when Fetch => -- Fetch instruction from memory ------------------
--------------------------------
if unsigned(Program_Counter) <= 250 then -- read just memory not
IO
Request_Address <= Program_Counter;
Memory_Control <= '1'; -- Read Memory
Memory_Enable <= '1';
Data_in_out_Memory <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; --Apply
tri-state on data bus

Memory_Buffer_Register <= Data_in_out_Memory(27 downto 0); --
data or address part  -- last instruct bit 12
Instruction_Register <= Data_in_out_Memory(31 downto 28); --
instruction part
Memory_Address_Register <= Data_in_out_Memory(27 downto 20); --
used if we want store value in specific memory location or to
```

```
fetch operand from memory

--Clear alu buss
ALU_Operation <= "ZZZ";
Data_in_out_ALU <= "ZZZZZZZZZZZZZZZ";
ALU_Enable <= '0';
--Clear trialu buss
TriALU_Operation <= "ZZZZZ";
Data_in_out_TriALU <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
TriALU_Enable <= '0';

if Fetch_Counter = 2 then
Memory_Enable <= '0';
Control_State   <= Decode;
Fetch_Counter  <= (others => '0');
else
Fetch_Counter <= Fetch_Counter + 1;
end if;
end if;


when Decode => -- Decode instruction ----------------------------
-----------------------------------

if ALU_Operation_Counter = 0 and TriALU_Operation_Counter = 0
then
case Instruction_Register is -- increamnt PC based on instruction
leangth
when "0000" =>
Program_Counter <= Program_Counter+1;
when "1001" =>
Program_Counter <= Program_Counter+1;
when "1100" =>
if Memory_Buffer_Register(27 downto 24) = "0100" then
Program_Counter <= Program_Counter+2;
else
Program_Counter <= Program_Counter+1;
end if;
when "0111" =>
case Memory_Buffer_Register(27 downto 25) is
when "000" => -- Add values to register    PC + 4
if Memory_Buffer_Register(24) = '1' then
Program_Counter <= Program_Counter+4;
else
Program_Counter <= Program_Counter+3;
end if;
when "001" => -- Add value to register     PC + 2
Program_Counter <= Program_Counter+2;
```

```
when "010" => -- Greater values            PC + 1
Program_Counter <= Program_Counter+1;
when "011" => -- Smaller values
Program_Counter <= Program_Counter+1;
when "100" => -- Xor values
if Memory_Buffer_Register(24) = '1' then
Program_Counter <= Program_Counter+4;
else
Program_Counter <= Program_Counter+3;
end if;
when "101" => -- Xor value
Program_Counter <= Program_Counter+2;
when "110" => -- Load values
if Memory_Buffer_Register(24) = '1' then
Program_Counter <= Program_Counter+4;
else
Program_Counter <= Program_Counter+3;
end if;
when "111" => -- Load value
Program_Counter <= Program_Counter+2;
when others =>
end case;
when others =>
Program_Counter <= Program_Counter+3;
end case;
end if;

case Instruction_Register is

when "0000" =>     -- NOP instructions   split memory instruction
and data in memory data section and code section
Control_State <= Fetch;
when "0001" =>     -- ADDI operation with absolute values
ALU_Enable <= '1';
ALU_Operation <="000";
Data_in_out_ALU <= Memory_Buffer_Register(27 downto 12);
Control_State <= Execute;

when "0010" =>     -- SUBI operation with absolute values
ALU_Enable <= '1';
ALU_Operation <="001";
Data_in_out_ALU <= Memory_Buffer_Register(27 downto 12);
Control_State <= Execute;

when "0011" =>     -- STA Store accumulator value in memory
case Memory_Buffer_Register(12) is
when '0' => -- Store from Alu
ALU_Enable <= '1';
```

```
ALU_Operation <= "101";
Data_in_out_ALU <= "ZZZZZZZZZZZZZZZ";
when '1' => -- Store from TriAlu
TriALU_Enable <= '1';
TriALU_Operation <= "00011";
Data_in_out_TriALU <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
when others =>
end case;
if ALU_Operation_Counter = 1 then
Control_State <= Execute;
ALU_Operation_Counter  <= (others => '0');
else
ALU_Operation_Counter <= ALU_Operation_Counter + 1;
end if;

when "0100" =>     --  MOV load value to specific register
ALU_Enable <= '1';
ALU_Operation <="010";
Data_in_out_ALU <= Memory_Buffer_Register(27 downto 12);
Control_State <= Execute;

when "0101" =>     -- LDA load value from address in the memory
to specific register
Memory_Address_Register <= Memory_Buffer_Register(27 downto 20);-
-                        Memory address register
Register_Select <= Memory_Buffer_Register(13 downto 12);
Control_State <= Fetch_Operand;

--ALU inner operation-----------------------------------------------
--------------------------
when "0110" =>
ALU_Enable <= '1';
Data_in_out_ALU(3 downto 0) <= Memory_Buffer_Register(15 downto
12);-- Inner ALU operation modes
ALU_Operation <="100";
Control_State <= Execute;
------------------------------------------------------------------
--------------------------

when "0111" => -- TRIALU Vector processing SIMD
TriALU_Enable <= '1';
TRIALU_Vector_Length <= Memory_Buffer_Register(24);
Data_in_out_TriALU <= Memory_Buffer_Register(23 downto 0);
Control_State <= Execute;
case Memory_Buffer_Register(27 downto 25) is
when "000" => -- Add values to register    PC + 4
TriALU_Operation <="10000";
when "001" => -- Add value to register     PC + 2
```

```
TriALU_Operation <="10001";
when "010" => -- Greater values            PC + 1
TriALU_Operation <="10010";
when "011" => -- Smaller values
TriALU_Operation <="10011";
when "100" => -- Xor values
TriALU_Operation <="11000";
when "101" => -- Xor value
TriALU_Operation <="11001";
when "110" => -- Load values
TriALU_Operation <="11010";
when "111" => -- Load value
TriALU_Operation <="11011";
when others =>
end case;

when "1000" => -- TRIALU Parallel processing MIMD
TriALU_Enable <= '1';
Data_in_out_TriALU(15 downto 0) <= Memory_Buffer_Register(27
downto 12);
Data_in_out_TriALU(16) <= Memory_Buffer_Register(11); -- Division
over operation  1    x/y+c   , 0 x/y +c
Data_in_out_TriALU(17) <= Memory_Buffer_Register(10); -- Logic
operations
Data_in_out_TriALU(19 downto 18) <= Memory_Buffer_Register(9
downto 8); -- shift a operation
Control_State <= Execute;
if Memory_Buffer_Register(27 downto 24) = "0000" then -- No math
operations
TriALU_Operation <="00001"; -- load data to registers and set
Modes
else
TriALU_Operation <="00000"; -- Math operations
end if;

when "1001" => -- TRIALU Accumulator structure
A           A       A            A            B         B
B            B
TriALU_Enable <= '1';
TriALU_Operation <= '0'&Memory_Buffer_Register(27 downto 24); --
from 00100 + , 00101 - , 00110 AND , 00111 OR   , 01100 + , 01101
-  , 01110 AND , 01111 OR
Control_State <= Execute;

when "1010" => -- jump with modes
if Memory_Buffer_Register(11 downto 8) = "0001" then
ALU_Enable <= '1';
ALU_Operation <= "101";
```

```
Returned_Address_Accumulator <= Program_Counter&Data_in_out_ALU(7
downto 0);
elsif Memory_Buffer_Register(11 downto 8) > "1100" then
TriALU_Enable <= '1';
TriALU_Operation <= "00010";
Data_in_out_TriALU(15 downto 0) <= Memory_Buffer_Register(27
downto 12);
if TriALU_Operation_Counter = 1 and  Memory_Buffer_Register(11
downto 8) > "1100" then
Control_State <= Execute;
TriALU_Operation_Counter  <= (others => '0');
else
TriALU_Operation_Counter <= TriALU_Operation_Counter + 1;
end if;
end if;

if Memory_Buffer_Register(11 downto 8) < "1100" then
Control_State <= Execute;
end if;

when "1011" => -- RTS
Pop_Address_Accumulator;
Control_State <= Execute;

when "1100" => -- TRIALU Stack structure
Push       Pop     +         -         *         /        And
or
TriALU_Enable <= '1';
TriALU_Operation <= '1'&Memory_Buffer_Register(27 downto 24); --
from 10100  , 10101  , 10110  , 10111    , 11100  , 11101   ,
11110  , 11111
Data_in_out_TriALU(7 downto 0) <= Memory_Buffer_Register(23
downto 16);
Control_State <= Execute;

when "1101" => -- INM   interrupts mask 1 enable interrupt , 0
disable interrupt
Software_INT_Mask <= Memory_Buffer_Register(17 downto 12); --Set
interrupts mask
Control_State <= Execute;

when "1110" => -- INTISR

when "1111" => -- HLT
if(Memory_Buffer_Register(12) = '1') then-- HALT for specific
time
if(Memory_Buffer_Register(13) = '1') then
Total_Halt_Time <= unsigned(Memory_Buffer_Register(27 downto
```

```
14))*to_unsigned(board_frequency,50);
else
-- hint for assmbler coder don't write big number! write between
1 and 10
Total_Halt_Time <=
shift_right(to_unsigned(board_frequency,64),to_integer(unsigned(M
emory_Buffer_Register(27 downto 14))+1));
end if;
end if;

Control_State <= Execute;
when others =>

end case;



when Fetch_Operand => -- Fetch Operand form memory     ----------
----------------------------------------
Request_Address <= Memory_Address_Register(Address_Bus_Size
downto 0);-----------------------------------------------------
-change it later!!!!!!!!
Memory_Control <= '1'; -- Read Memory
Memory_Enable <= '1';
Data_in_out_Memory <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; --Apply
tri-state on data bus

Memory_Buffer_Register(27 downto 20) <= Data_in_out_Memory(31
downto 24); -- Operand value

if Fetch_Operand_Counter = 2 then
Memory_Enable <= '0';
ALU_Enable <= '1';
Control_State   <= Execute;
Fetch_Operand_Counter  <= (others => '0');
else
Fetch_Operand_Counter <= Fetch_Operand_Counter + 1;
end if;

when Execute => -- Excute instruction --------------------------
---------------------------------------

case Instruction_Register is
when "0000" =>     -- Data no instructions

when "0001" =>     -- ADDI operation with absolute values
ALU_Enable <= '0';
```

```
when "0010" =>     -- SUBI operation with absolute values
ALU_Enable <= '0';

when "0011" =>     -- STA Store accumulator value in memory
case Memory_Buffer_Register(12) is
when '0' =>
ALU_Enable <= '0';
Memory_Buffer_Register(7 downto 0) <= Data_in_out_ALU(7 downto
0);
when '1' =>
TriALU_Enable <= '0';
Memory_Buffer_Register(7 downto 0) <= Data_in_out_TriALU(7 downto
0);
when others =>
end case;

when "0100" =>     -- MOV load value to specific register
ALU_Enable <= '0';

when "0101" =>     -- LDA load value from address to specific
register -------------
Data_in_out_ALU(15 downto 8) <= Memory_Buffer_Register(27 downto
20);
Data_in_out_ALU(1 downto 0) <= Register_Select;
ALU_Operation <= "011";
------------------------------------------------------------------
--------------------

--ALU inner operation-------------------------------------
when "0110" =>
ALU_Enable <= '0';
------------------------------------------------------------

when "0111" =>    -- TRIALU SIMD
TriALU_Enable <= '0';


when "1000" =>    --TRIALU MIMD
if TriALU_Complete = '1' then
TriALU_Enable <= '0';
Control_State <= Fetch;
end if;

when "1001" =>   -- TRIALU Accumulator structure
TriALU_Enable <= '0';

when "1010" =>    --  jump command with modes
case Memory_Buffer_Register(11 downto 8)  is -- modes
```

```
when "0000"  =>-- JMP  unconditional jump to specific address
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);

when "0001"  =>-- CALL  unconditional CALL to specific subroutine
Push_Address_Accumulator(Returned_Address_Accumulator);-- this is
used when we CALL subroutine ,jump -> CALL Subroutine mode
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
ALU_Enable <= '0';
when "0010"  =>-- JMAGB  conditional jump to specific address
when register a > register b
if(ALU_Flags(4)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "0011"  =>-- JMASB  conditional jump to specific address
when register a < register b
if(ALU_Flags(3)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "0100"  =>-- JMAEB  conditional jump to specific address
when register a = register b
if(ALU_Flags(4) ='0' and ALU_Flags(3) ='0') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "0101"  =>-- JMAGC  conditional jump to specific address
when register a > Accumulator
if(ALU_Flags(2)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "0110"  =>-- JMASC  conditional jump to specific address
when register a < Accumulator
if(ALU_Flags(1)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "0111"  =>-- JMAEC  conditional jump to specific address
when register a = Accumulator
if(ALU_Flags(2) ='0' and ALU_Flags(1) ='0') then
```

```
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "1000"  =>-- JMANB  conditional jump to specific address
when register a /= register b
if((ALU_Flags(4) XOR ALU_Flags(3)) ='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "1001"  =>-- JMANC  conditional jump to specific address
when register a /= Accumulator
if((ALU_Flags(2) XOR ALU_Flags(1)) ='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "1010"  =>-- JMZ  conditional jump to specific address when
Accumulator = zero
if(ALU_Flags(0)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "1011"  =>-- JMSE  conditional jump to specific address when
Stack is empty
if(ALU_Flags(5)='1') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;

when "1100"  =>-- JIRQ  conditional jump to specific address when
IRQ = 1 ----------------------------------------used for Input
if(IRQ='0') then
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+20
downto 20);
end if;
-- TRIALU flags check ------------------------------------------
--------------------
when "1101"  =>
TriALU_Enable <= '0';
if(TriALU_Flags(6)='1') then --FeedBack = limit
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+12
downto 12);
end if;

when "1110"  => -- TRIALU check
```

```
TriALU_Enable <= '0';
if(TriALU_Flags(7)='1') then --FeedBack < limit
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+12
downto 12);
end if;

when "1111"  => -- TRIALU check
TriALU_Enable <= '0';
if(TriALU_Flags(8)='1') then --FeedBack > limit
Program_Counter <= Memory_Buffer_Register(Address_Bus_Size+12
downto 12);
end if;

when others =>
end case;


when "1011" =>    -- RTS returen to previous address and
accumulator value before jumping to subroutines
Program_Counter <=
Returned_Address_Accumulator(Address_Bus_Size+8 downto 8); --
return address
ALU_Enable <= '1';
ALU_Operation <="010";
Data_in_out_ALU(15 downto 8) <= Returned_Address_Accumulator(7
downto 0); -- return accumulator value
Data_in_out_ALU(1 downto 0) <=  "10";


when "1100" => -- TRIALU Stack structure
TriALU_Enable <= '0';

-- when "1101" =>    -- INM interrupts mask

-- when "1110" =>    -- INTISR
when "1111" =>    -- HLT
if(Memory_Buffer_Register(12) = '1') then-- HALT for specific
time
if(Memory_Buffer_Register(13) = '1') then
if(HALT_Counter = Total_Halt_Time) then -- more than one second
HALT_Counter <= (others => '0');
Control_State <= Fetch;
else
HALT_Counter<=HALT_Counter+1;
end if;
else
-- don't write big number! write between 1 and 10
if Halt_Counter = Total_Halt_Time then -- less than one second
```

```
HALT_Counter <= (others => '0');
Control_State <= Fetch;
else
HALT_Counter<=HALT_Counter+1;
end if;
end if;
else-- HALT forever "it can be interrupt by hardware interrupt!"
Control_Flags(2) <= '1';
end if;
when others =>

end case;
--   div by zero         carry                                inner
operation                     SUB
ADD
if ((Alu_Flags(5) = '1' or ALU_Carry_Out = '1') and
(Instruction_Register ="0110" or Instruction_Register ="0010"  or
Instruction_Register ="0001")) then

if Execute_Counter = 0 then
ALU_Enable <= '1';
ALU_Operation <="111";
end if;

if Execute_Counter = 1 then
Returned_Address_Accumulator <= Program_Counter&Data_in_out_ALU(7
downto 0);
end if;

if Execute_Counter = 2 then
ALU_Enable <= '0';
Carry_Interrupt(Returned_Address_Accumulator);-----------------
can't happen at the same time
division_by_zero_Interrupt(Returned_Address_Accumulator);------
!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Execute_Counter  <= (others => '0');
Control_State   <= Fetch;
else
Execute_Counter <= Execute_Counter + 1;
end if;

elsif (Instruction_Register = "0011") then
Control_State <= Store;
elsif (Instruction_Register /= "1111" and Instruction_Register /=
"1000")  then -- don't fetch anything if HALT or if there is
continues operation inside TriALU
Control_State <= Fetch;
end if;
```

```
when Store => -- Store result in memory or io controller buffer -
-----------------------------------------------------------------
Data_in_out_Memory(7 downto 0) <= Memory_Buffer_Register(7 downto
0);
Request_Address <= Memory_Address_Register;
Memory_Control <= '0';
Memory_Enable <= '1';

if Store_Counter = 2 then                  -- IO need 2 cycles
Memory_Enable <= '0';          -- ------------------------------
---------------!!!!!!!!!!!!!!!!!!!!!!!!
Control_State   <= Fetch;
Request_Address <= Program_Counter;
Store_Counter  <= (others => '0');
else
Store_Counter <= Store_Counter + 1;
end if;

end case;
end if;
end if;
end process;

end Behavioral;
```

ALU component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity ALU is
port(
Clock : in std_logic;
ALU_Operation: in std_logic_vector(2 downto 0);-- ALU Operations
Data_in_out_ControlUnit : inout std_logic_vector(15 downto
0):="ZZZZZZZZZZZZZZZZ";-- from and to Control unit
Carry_Out : out std_logic:='0';
Flags : out std_logic_vector(5 downto 0):="ZZZZZZ";
Enable : in std_logic:='Z';
Reset : in std_logic:='Z'
);
end entity;

architecture Behave of ALU is

signal Accumulator : std_logic_vector(8 downto 0):="000000000";
signal Register_A : std_logic_vector(7 downto 0):="00000000";
signal Register_B : std_logic_vector(7 downto 0):="00000000";
signal Error : std_logic:='0';
begin


process(Clock)
begin

if rising_edge(Clock) then
if Reset = '0' then
Data_in_out_ControlUnit <= "ZZZZZZZZZZZZZZZZ";
Register_A <= "00000000";
Register_B <= "00000000";
Accumulator <= "000000000";
Error <= '0';
elsif Enable = '1' then
Error <= '0';
```

```
case ALU_Operation is
when "000" => -- ADDI
Accumulator <= ext(Data_in_out_ControlUnit(15 downto 8),9) +
ext(Data_in_out_ControlUnit(7 downto 0),9); -- BUG !!!

when "001" => -- SUBI
Accumulator <= ext(Data_in_out_ControlUnit(15 downto 8),9) -
ext(Data_in_out_ControlUnit(7 downto 0),9);

when "010" => -- MOV load value to corresponding register
if Data_in_out_ControlUnit(1 downto 0) = "00" then
Register_A <= Data_in_out_ControlUnit(15 downto 8);
elsif Data_in_out_ControlUnit(1 downto 0) = "01" then
Register_B <= Data_in_out_ControlUnit(15 downto 8);
elsif Data_in_out_ControlUnit(1 downto 0) = "10" then
Accumulator <= ext(Data_in_out_ControlUnit(15 downto 8),9);
end if;

when "011" => -- LDA load value from address to specific register
if Data_in_out_ControlUnit(1 downto 0) = "00" then
Register_A <= Data_in_out_ControlUnit(15 downto 8);
elsif Data_in_out_ControlUnit(1 downto 0) = "01" then
Register_B <= Data_in_out_ControlUnit(15 downto 8);
elsif Data_in_out_ControlUnit(1 downto 0) = "10" then
Accumulator <= ext(Data_in_out_ControlUnit(15 downto 8),9);
end if;

when "100" =>

case Data_in_out_ControlUnit(3 downto 0) is

--Arithmetic operations-----------------------------------------
-------
when "0000" => -- ADR add current registers values A+B
Accumulator <= ext(Register_A,9) + ext(Register_B,9);

when "0001" => -- SUR Sub current registers values A-B
Accumulator <= ext(Register_A,9) - ext(Register_B,9);

when "0010" => -- MUL multiple current registers values A*B
Accumulator <= ext(ext(Register_A,9) * ext(Register_B,9),9);

when "0011" => -- DIV Sub Divide registers values A/B using shift
to right  , check if b is zero to send error flag
if Register_B = "00000000" then
Error <= '1';
else
```

```
Accumulator <=
std_logic_vector(shift_right(IEEE.NUMERIC_STD.unsigned(ext(Regist
er_A,9)),IEEE.NUMERIC_STD.to_integer(IEEE.NUMERIC_STD.unsigned(Re
gister_B)-1)));
end if;
------------------------------------------------------------------
------------

--Logic operations-------------------------------------------------
------------
when "0100" => -- AND  current registers values A and B
Accumulator <= ext(Register_A,9) AND ext(Register_B,9);

when "0101" => -- OR  current registers values A or B
Accumulator <= ext(Register_A,9) OR ext(Register_B,9);

when "0110" => -- XOR  current registers values A xor B
Accumulator <= ext(Register_A,9) XOR ext(Register_B,9);

when "0111" => -- NOT  current register value A
Accumulator <= NOT ext(Register_A,9);
------------------------------------------------------------------
-------------

--Register A manipulations-----------------------------------------
-------------
when "1000" => -- Increment  current register value A
Accumulator <= ext(Register_A,9) + 1;

when "1001" => -- Decrement  current register value A
Accumulator <= ext(Register_A,9) - 1;

when "1010" => -- Shift to right  current register value A
Accumulator <= ext('0' & Register_A(7 downto 1),9);

when "1011" => -- Shift to lift  current register value A
Accumulator <= ext(Register_A(6 downto 0) & '0',9);

--Register B manipulations-----------------------------------------
-------------
when "1100" => -- Increment  current register value B
Accumulator <= ext(Register_B,9) + 1;

when "1101" => -- Decrement  current register value B
Accumulator <= ext(Register_B,9) - 1;

when "1110" => -- Shift to right  current register value B
Accumulator <= ext('0' & Register_B(7 downto 1),9);
```

```
when "1111" => -- Shift to lift  current register value B
Accumulator <= ext(Register_B(6 downto 0) & '0',9);
when others =>
end case;

when "101" =>
Data_in_out_ControlUnit(7 downto 0) <= Accumulator(7 downto 0);
when others =>
end case;

else
Data_in_out_ControlUnit <= (others=>'Z');
end if;
end if;
end process;

process(Clock)   -- Flags process
begin

if rising_edge(Clock) then
if Reset = '0' then
Carry_Out <= '0';
Flags <= "000000";
else

Flags(5) <= Error; -- division by zero
Carry_Out <= Accumulator(8); -- Carry
if Register_A > Register_B then
Flags(4) <= '1';
else
Flags(4) <= '0';
end if;

if Register_A < Register_B then
Flags(3) <= '1';
else
Flags(3) <= '0';
end if;

if Register_A > Accumulator then
Flags(2) <= '1';
else
Flags(2) <= '0';
end if;

if Register_A < Accumulator then
Flags(1) <= '1';
```

```
else
Flags(1) <= '0';
end if;

if Accumulator(7 downto 0) = "00000000" then
Flags(0) <= '1'; -- Zero flag
else
Flags(0) <= '0';
end if;
end if;
end if;
end process;


end Behave;
```

Tri-ALU component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity TriALU is
port(
Clock : in std_logic;
ALU_Operation: in std_logic_vector(4 downto 0);-- ALU Operations
Data_in_out_ControlUnit : inout std_logic_vector(23 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZ";-- from and to Control unit
Vector_Length : in std_logic:='0';
Carry_Out : out std_logic:='0';
Flags : out std_logic_vector(9 downto 0):="ZZZZZZZZZZ";
Enable : in std_logic:='Z';
Reset : in std_logic:='Z';
Complete : Buffer std_logic:='0'
);
end entity;

architecture Behave of TriALU is

signal Accumulator : std_logic_vector(8 downto 0):="000000000"; -
- out
signal Register_A : std_logic_vector(7 downto 0):="00000000";
signal Register_B : std_logic_vector(7 downto 0):="00000000";
signal Feedback : std_logic_vector(7 downto 0):="00000000";
signal Limit : std_logic_vector(7 downto 0):="00000000";
signal Error : std_logic:='0';

signal Operations : std_logic_vector(3 downto 0):="0000";
signal Logic_Operations : std_logic:='0';
signal Shift_Operations : std_logic_vector(1 downto 0):="00";
signal Variables : std_logic_vector(2 downto 0):="000";
signal Signs : std_logic_vector(2 downto 0):="000";
signal Division_Over_Operation : std_logic := '0';
signal Destinations : std_logic_vector(3 downto 0):="0000";
signal Feedback_Operation : std_logic_vector(1 downto 0):="00";
signal Modes : std_logic_vector(1 downto 0):="00"; -- 00 normal
dtat tramsfer , 01 flip flop (A,B)
```

```
signal Minus : std_logic_vector(7 downto 0):="11111111";

signal Stack_Pointer: IEEE.NUMERIC_STD.unsigned(1 downto 0) :=
(others => '0');
begin

Operations <= Data_in_out_ControlUnit(15 downto 12);
Logic_Operations <= Data_in_out_ControlUnit(17);
Shift_Operations <= Data_in_out_ControlUnit(19 downto 18);
Variables <= Data_in_out_ControlUnit(11 downto 9);
Signs <= Data_in_out_ControlUnit(8 downto 6);
Division_Over_Operation <= Data_in_out_ControlUnit(16);
Destinations <= Data_in_out_ControlUnit(5 downto 2);
Feedback_Operation <= Data_in_out_ControlUnit(1 downto 0);
Limit <= Data_in_out_ControlUnit(15 downto 8);


process(Clock)
variable Variable1 : std_logic_vector(7 downto 0):="00000000";
variable Variable2 : std_logic_vector(7 downto 0):="00000000";
variable Variable3 : std_logic_vector(7 downto 0):="00000000";
variable Result1 : std_logic_vector(8 downto 0):="000000000";
variable Result2 : std_logic_vector(8 downto 0):="000000000";
variable Switcher : std_logic :='0';
begin

if rising_edge(Clock) then
if Reset = '0' then
Data_in_out_ControlUnit <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
Register_A <= "00000000";
Register_B <= "00000000";
Accumulator <= "000000000";
Feedback <= "00000000";
Error <= '0';-----------------------------------
Complete <= '0';
Flags(8 downto 6) <= "000";
Stack_Pointer <= "00";
elsif Enable = '1' and Complete = '0' then
Error <= '0';

case ALU_Operation is
when "00000" => -- Operation  Parallel processing MIMD
if signs(0)='1' then -- Signs
variable1 := ext((Register_A*Minus),8);
else
variable1 := Register_A;
end if;
if signs(1)='1' then
```

```
variable2 := ext((Register_B*Minus),8);
else
variable2 := Register_B;
end if;
if signs(2)='1' then
if Variables = "100" then -- use Accumulator with a and b
registers
variable3 := ext((Accumulator*Minus),8);
else
variable3 := ext((Feedback*Minus),8);
end if;
else
if Variables = "100" then -- use Accumulator with a and b
registers
variable3 := ext(Accumulator,8);
else
variable3 := Feedback;
end if;
end if;

if Variables /= "100" then -- use Accumulator with a and b
registers
if Variables(0)='0' then -- Variables
variable1 := "00000000";
end if;
if Variables(1)='0' then
variable2 := "00000000";
end if;
if Variables(2)='0' then
variable3 := "00000000";
end if;
end if;


if Logic_Operations = '0' then
if Operations(1 downto 0) = "00" then -- Operations 1 Arithmetic
--------------------------------
if Variables(0)='0' then
Result1 := ext(variable2,9);
end if;
if Variables(1)='0' then
Result1 := ext(variable1,9);
end if;
end if;

if Operations(1 downto 0) =  "01" then  -- Addition 1
Result1 := ext(variable1 + variable2,9) ;
end if;
```

```
if Operations(1 downto 0) =  "10" then -- multiplication 1
Result1 := ext(variable1 * variable2,9);
end if;

if Operations(1 downto 0) = "11" then  -- division 1
if variable2 /= "00000000" then
Result1 :=
ext(std_logic_vector(IEEE.NUMERIC_STD.unsigned(variable1) /
IEEE.NUMERIC_STD.unsigned(variable2)),9);
else
Error <= '1';
end if;
end if;

elsif Logic_Operations = '1' then
if Operations(1 downto 0) = "00" then -- Operations 1 logic -----
-----------------------------
Result1 := ext(variable1 AND variable2,9); --AND

elsif Operations(1 downto 0) =  "01" then
Result1 := ext(variable1 XOR variable2,9); -- XOR


elsif Operations(1 downto 0) =  "10" then
Result1 := ext(variable1 OR variable2,9); -- OR


elsif Operations(1 downto 0) = "11" then
Result1 := ext(variable1 NAND variable2,9); -- NOT
end if;
end if;


if Operations(3 downto 2) = "00" then-- Operations 2 ------------
------------------------- no v3
Result2 := Result1;
end if;

if Operations(3 downto 2) =  "01" then  -- Addition 2
Result2 := Result1 + variable3 ;
end if;

if Operations(3 downto 2) = "10" then -- multiplication 2
Result2 := ext(Result1 * variable3,9);
end if;

if Operations(3 downto 2) = "11" then -- division 2
```

```
if Division_Over_Operation = '0' then
if variable3 /= "00000000" then
Result2 :=
ext(std_logic_vector(IEEE.NUMERIC_STD.unsigned(Result1) /
IEEE.NUMERIC_STD.unsigned(variable3)),9);
else
Error <= '1';
end if;
elsif Division_Over_Operation = '1' then
if Result1 /= "00000000" then
Result2 :=
ext(std_logic_vector(IEEE.NUMERIC_STD.unsigned(variable3) /
IEEE.NUMERIC_STD.unsigned(Result1)),9);
else
Error <= '1';
end if;
end if;

end if;


-- Destination----------------------------------------
if Destinations(0) = '1' then
Accumulator <= Result2;
end if;
if Destinations(1) = '1' then
Feedback <= ext(Result2,8);
end if;
if Destinations(2) = '1' then
Register_A <= ext(Result2,8);
end if;
if Destinations(3) = '1' then
Register_B <= ext(Result2,8);
end if;

case Shift_Operations is
when "00" => -- no shift
when "01" =>
Register_A <=
std_logic_vector(shift_right(IEEE.NUMERIC_STD.unsigned(Register_A
),1)); -- shift right
when "10" =>
Register_A <=
std_logic_vector(shift_left(IEEE.NUMERIC_STD.unsigned(Register_A)
,1)); -- shift left
when others =>
end case;
```

```
case Modes is
when "01" => -- switch the result between A and B in every math
operation
if Switcher = '0' then
Register_A <= ext(Result2,8);
elsif Switcher = '1' then
Register_B <= ext(Result2,8);
end if;
Switcher := not Switcher;
Complete <= '1';
when "10" => -- continues, normal mode
if Feedback = "00000001" then
Complete <= '1';
else
Complete <= '0';
end if;
Switcher := '0';
when "11" => -- continues, switch the result between A and B in
every math operation
if Switcher = '0' then
Register_A <= ext(Result2,8);
elsif Switcher = '1' then
Register_B <= ext(Result2,8);
end if;
Switcher := not Switcher;
if Feedback = "00000001" then
Complete <= '1';
else
Complete <= '0';
end if;
when others => -- reseting
Switcher := '0';
Complete <= '1';
end case;

case Feedback_Operation(1 downto 0) is -- Feedback Operations ---
-----------------------------------------------
when "00" => -- No oprations
when "01" => -- --
Feedback <= Feedback - 1;
when "10" => -- ++
Feedback <= Feedback + 1;
when "11" => -- Zeros
Feedback <= "00000000";
when others =>
end case;
```

```
when "00001" => -- Load value to corresponding register and set
Modes-------------------------------------------------------------
----------------------
if Data_in_out_ControlUnit(9 downto 8) = "00" then
Register_A <= Data_in_out_ControlUnit(7 downto 0);
elsif Data_in_out_ControlUnit(9 downto 8) = "01" then
Register_B <= Data_in_out_ControlUnit(7 downto 0);
elsif Data_in_out_ControlUnit(9 downto 8) = "10" then
Feedback <= Data_in_out_ControlUnit(7 downto 0);
elsif Data_in_out_ControlUnit(9 downto 8) = "11" then
Accumulator(7 downto 0) <= Data_in_out_ControlUnit(7 downto 0);
end if;
Modes <= Data_in_out_ControlUnit(11 downto 10);
Complete <= '1';
when "00010" => -- Feedback flags
if Feedback = Limit then
Flags(6) <= '1';
else
Flags(6) <= '0';
end if;

if Feedback < Limit then
Flags(7) <= '1';
else
Flags(7) <= '0';
end if;

if Feedback > Limit then
Flags(8) <= '1';
else
Flags(8) <= '0';
end if;
Complete <= '1';
when "00011" =>
Data_in_out_ControlUnit(7 downto 0) <= Accumulator(7 downto 0);
Complete <= '1';

--TRIALU Accumulator structure ---------------------------------
-------------------------------------
when "00100" =>
Accumulator <= ext(Accumulator(7 downto 0) + Register_A,9);
when "00101" =>
Accumulator <= ext(Accumulator(7 downto 0) - Register_A,9);
when "00110" =>
Accumulator <= ext(Accumulator(7 downto 0) and Register_A,9);
when "00111" =>
Accumulator <= ext(Accumulator(7 downto 0) or Register_A,9);
```

```
when "01100" =>
Accumulator <= ext(Accumulator(7 downto 0) + Register_B,9);
when "01101" =>
Accumulator <= ext(Accumulator(7 downto 0) - Register_B,9);
when "01110" =>
Accumulator <= ext(Accumulator(7 downto 0) and Register_B,9);
when "01111" =>
Accumulator <= ext(Accumulator(7 downto 0) or Register_B,9);

--Vector processing SIMD ----------------------------------------
------------------------------
when "10000" => -- Add values
if Vector_Length = '0' then
Register_A <= Register_A + Data_in_out_ControlUnit(23 downto 16);
Register_B <= Register_B + Data_in_out_ControlUnit(15 downto 8);
else
Register_A <= Register_A + Data_in_out_ControlUnit(23 downto 16);
Register_B <= Register_B + Data_in_out_ControlUnit(15 downto 8);
Feedback   <= Feedback + Data_in_out_ControlUnit(7 downto 0);
end if;

when "10001" =>  -- Add value
if Vector_Length = '0' then
Register_A <= Register_A + Data_in_out_ControlUnit(23 downto 16);
Register_B <= Register_B + Data_in_out_ControlUnit(23 downto 16);
else
Register_A <= Register_A + Data_in_out_ControlUnit(23 downto 16);
Register_B <= Register_B + Data_in_out_ControlUnit(23 downto 16);
Feedback   <= Feedback + Data_in_out_ControlUnit(23 downto 16);
end if;

when "10010" => -- Greater values
if Register_A >= Register_B and Register_A >= Feedback then
Accumulator <= ext(Register_A,9);
elsif Register_B >= Register_A and Register_B >= Feedback then
Accumulator <= ext(Register_B,9);
else
Accumulator <= ext(Feedback,9);
end if;

when "10011" => -- Smaller value
if Register_A <= Register_B and Register_A <= Feedback then
Accumulator <= ext(Register_A,9);
elsif Register_B <= Register_A and Register_B <= Feedback then
Accumulator <= ext(Register_B,9);
else
Accumulator <= ext(Feedback,9);
end if;
```

```
when "11000" => -- Xor values
if Vector_Length = '0' then
Register_A <= Register_A xor Data_in_out_ControlUnit(23 downto
16);
Register_B <= Register_B xor Data_in_out_ControlUnit(15 downto
8);
else
Register_A <= Register_A xor Data_in_out_ControlUnit(23 downto
16);
Register_B <= Register_B xor Data_in_out_ControlUnit(15 downto
8);
Feedback   <= Feedback xor Data_in_out_ControlUnit(7 downto 0);
end if;
when "11001" => -- Xor value
if Vector_Length = '0' then
Register_A <= Register_A xor Data_in_out_ControlUnit(23 downto
16);
Register_B <= Register_B xor Data_in_out_ControlUnit(23 downto
16);
else
Register_A <= Register_A xor Data_in_out_ControlUnit(23 downto
16);
Register_B <= Register_B xor Data_in_out_ControlUnit(23 downto
16);
Feedback   <= Feedback xor Data_in_out_ControlUnit(23 downto 16);
end if;

when "11010" => -- load values
if Vector_Length = '0' then
Register_A <= Data_in_out_ControlUnit(23 downto 16);
Register_B <= Data_in_out_ControlUnit(15 downto 8);
else
Register_A <= Data_in_out_ControlUnit(23 downto 16);
Register_B <= Data_in_out_ControlUnit(15 downto 8);
Feedback   <= Data_in_out_ControlUnit(7 downto 0);
end if;

when "11011" => -- load value
if Vector_Length = '0' then
Register_A <= Data_in_out_ControlUnit(23 downto 16);
Register_B <= Data_in_out_ControlUnit(23 downto 16);
else
Register_A <= Data_in_out_ControlUnit(23 downto 16);
Register_B <= Data_in_out_ControlUnit(23 downto 16);
Feedback   <= Data_in_out_ControlUnit(23 downto 16);
end if;
```

```
--TRIALU Stack structure ----------------------------------------
--------------------------------
when "10100" => -- Push
case Stack_Pointer is
when "00" =>
Register_A <= Data_in_out_ControlUnit(7 downto 0);
Stack_Pointer <= Stack_Pointer + 1;
when "01" =>
Register_B <= Data_in_out_ControlUnit(7 downto 0);
Stack_Pointer <= Stack_Pointer + 1;
when "10" =>
Feedback <= Data_in_out_ControlUnit(7 downto 0);
Stack_Pointer <= Stack_Pointer + 1;
when others =>
report "Stack overflow" severity failure;
Flags(9) <= '1';
end case;

when "10101" => -- Pop
if Stack_Pointer >= 0 then
case Stack_Pointer is
when "01" =>
Stack_Pointer <= Stack_Pointer - 1;
Accumulator(7 downto 0 ) <= Register_A;
when "10" =>
Stack_Pointer <= Stack_Pointer - 1;
Accumulator(7 downto 0 ) <= Register_B;
when "11" =>
Stack_Pointer <= Stack_Pointer - 1;
Accumulator(7 downto 0 ) <= Feedback;
when others =>
end case;
else
report "Stack underflow" severity failure;
end if;

when "10110" =>  -- +
case Stack_Pointer is
when "01" =>
Register_A <= Register_A + Register_A;
when "10" =>
Register_A <= Register_A + Register_B;
Stack_Pointer <= Stack_Pointer - 1;
when "11" =>
Register_B <= Register_B + Feedback;
Stack_Pointer <= Stack_Pointer - 1;
when others =>
report "Stack underflow" severity failure;
```

```
end case;
when "10111" => -- -
case Stack_Pointer is
when "01" =>
Register_A <= Register_A - Register_A;
when "10" =>
Register_A <= Register_A - Register_B;
Stack_Pointer <= Stack_Pointer - 1;
when "11" =>
Register_B <= Register_B - Feedback;
Stack_Pointer <= Stack_Pointer - 1;
when others =>
report "Stack underflow" severity failure;
end case;
when "11100" => -- *
case Stack_Pointer is
when "01" =>
Register_A <= ext(Register_A * Register_A,8);
when "10" =>
Register_A <= ext(Register_A * Register_B,8);
Stack_Pointer <= Stack_Pointer - 1;
when "11" =>
Register_B <= ext(Register_B * Feedback,8);
Stack_Pointer <= Stack_Pointer - 1;
when others =>
report "Stack underflow" severity failure;
end case;
when "11101" => -- /
case Stack_Pointer is
when "01" =>
if Register_A /= "00000000" then
Register_A <=
std_logic_vector(IEEE.NUMERIC_STD.unsigned(Register_A) /
IEEE.NUMERIC_STD.unsigned(Register_A));
else
Error <= '1';
end if;

when "10" =>
if Register_B /= "00000000" then
Register_A <=
std_logic_vector(IEEE.NUMERIC_STD.unsigned(Register_A) /
IEEE.NUMERIC_STD.unsigned(Register_B));
Stack_Pointer <= Stack_Pointer - 1;
else
Error <= '1';
end if;
```

```
when "11" =>
if Feedback /= "00000000" then
Register_B <=
std_logic_vector(IEEE.NUMERIC_STD.unsigned(Register_B) /
IEEE.NUMERIC_STD.unsigned(Feedback));
Stack_Pointer <= Stack_Pointer - 1;
else
Error <= '1';
end if;

when others =>
report "Stack underflow" severity failure;
end case;
when "11110" =>  -- AND
case Stack_Pointer is
when "01" =>
Register_A <= Register_A and Register_A;
when "10" =>
Register_A <= Register_A and Register_B;
Stack_Pointer <= Stack_Pointer - 1;
when "11" =>
Register_B <= Register_B and Feedback;
Stack_Pointer <= Stack_Pointer - 1;
when others =>
report "Stack underflow" severity failure;
end case;
when "11111" =>  -- OR
case Stack_Pointer is
when "01" =>
Register_A <= Register_A or Register_A;
when "10" =>
Register_A <= Register_A or Register_B;
Stack_Pointer <= Stack_Pointer - 1;
when "11" =>
Register_B <= Register_B or Feedback;
Stack_Pointer <= Stack_Pointer - 1;
when others =>
report "Stack underflow" severity failure;
end case;

when others =>
end case;

else
Data_in_out_ControlUnit <= (others=>'Z');
Complete <= '0';
end if;
end if;
```

```
end process;

process(Clock)    -- Flags process
begin

if rising_edge(Clock) then
if Reset = '0' then
Carry_Out <= '0';
Flags(5 downto 0) <= "000000";
else

Flags(5) <= Error; -- division by zero
Carry_Out <= Accumulator(8); -- Carry
if Register_A > Register_B then
Flags(4) <= '1';
else
Flags(4) <= '0';
end if;

if Register_A < Register_B then
Flags(3) <= '1';
else
Flags(3) <= '0';
end if;

if Register_A > Accumulator then
Flags(2) <= '1';
else
Flags(2) <= '0';
end if;

if Register_A < Accumulator then
Flags(1) <= '1';
else
Flags(1) <= '0';
end if;

if Accumulator(7 downto 0) = "00000000" then
Flags(0) <= '1'; -- Zero flag
else
Flags(0) <= '0';
end if;
end if;
end if;
end process;


end Behave;
```

IO Controller component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;

use IEEE.STD_LOGIC_TEXTIO.ALL;

library STD;
use STD.TEXTIO.ALL;

entity IO_Controller is
generic (
Address_Bus_Size : integer  := 7
);
port(
Clock : in std_logic;
Interrupt_Request : out std_logic:='Z';-- to cpu
Interrupt_trigger : in std_logic:='Z';-- from user
Reset : in std_logic:='Z';
Read_or_Write    : in std_logic;
Data_in_out : inout std_logic_vector(27 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
Address_in : in std_logic_vector(Address_Bus_Size downto 0);
Input_Port : in std_logic_vector(7 downto 0):="ZZZZZZZZ";
Enable : in std_logic:='Z';
Output_Port : out std_logic_vector(7 downto 0):="ZZZZZZZZ"
);
end entity;


architecture behave of IO_Controller is

signal Input_Buffer : std_logic_vector(7 downto 0):="ZZZZZZZZ";
signal Output_Buffer : std_logic_vector(7 downto 0):="ZZZZZZZZ";

begin

Input_Buffer <= Input_Port; -- todo
Output_Port  <= Output_Buffer; -- done

Interrupt_Request <= Interrupt_trigger;   -- user -> io
```

```
controller -> cpu
process(Clock)
begin

if rising_edge(Clock) then
if Reset = '0' then
Output_Buffer <= "00000000";
Data_in_out <="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
else
if unsigned(Address_in) >= 251 and Enable = '1' then -- IO memory
mapped start from address 256
if Read_or_Write = '0' then--write  output <--- STR
Output_Buffer <= Data_in_out(7 downto 0);--
!!!

-- case Data_in_out(15 downto 8) is  -- change it to address
--  when "00000000" =>  -- output device

--   when others =>

--  end case;

elsif Read_or_Write = '1' then--read   input  ---> LDA
Data_in_out(27 downto 20) <= Input_Buffer;
else
Data_in_out<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end if;
else
Data_in_out<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
end if;
end if;
end if;
end process;

end behave;
```

CPU component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity CPU is
generic (
Address_Bus_Size : integer:= 7;
Board_frequency : integer:=  5 -- Altera board 50e6
);
port(
CPU_Clock : in std_logic;
Reset : in std_logic:='0';
CPU_Flags : out std_logic_vector(5 downto 0);

signal Control_Bus : out std_logic;-- control memory and IO
Controller

--buss between CPU and memory and IO Controller
signal Data_Bus: inout std_logic_vector(31 downto 0);
signal IRQ : in std_logic;
signal Memory_Enable : out std_logic;
signal Address_Bus: out std_logic_vector(Address_Bus_Size downto
0)
);
end entity;



architecture behave of CPU is

component Control_Unit is
generic (
Address_Bus_Size : integer;
Board_frequency : integer
);
Port (
Clock: in std_logic;
Reset: in std_logic;
Control_Flags : out std_logic_vector(5 downto 0);
--Memory Handling
Request_Address: out std_logic_vector(Address_Bus_Size downto
```

```
0);-- Memory Address
Data_in_out_Memory : inout std_logic_vector(31 downto 0);-- from
and to Memory
Memory_Control : out std_logic;  -- 1 for read 0 for write
Memory_Enable : out std_logic;
--IO Handling
IRQ : in std_logic;
--ALU Handling
ALU_Operation: out std_logic_vector(2 downto 0);-- ALU Operations
Data_in_out_ALU : inout std_logic_vector(15 downto 0);-- from and
to ALU
ALU_Enable : out std_logic;
ALU_Carry_Out : in std_logic;
ALU_Flags : in std_logic_vector(5 downto 0);
--TriALU Handling
TriALU_Operation: out std_logic_vector(4 downto 0):="ZZZZZ";--
ALU Operations
Data_in_out_TriALU : inout std_logic_vector(23 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZ";-- from and to ALU
TRIALU_Vector_Length : out std_logic:='0';
TriALU_Enable : out std_logic:='Z';
TriALU_Carry_Out : in std_logic;
TriALU_Flags : in std_logic_vector(9 downto 0);
TriALU_Complete : in std_logic
);
end component;

component ALU is
port(
Clock : in std_logic;
ALU_Operation: in std_logic_vector(2 downto 0);-- ALU Operations
Data_in_out_ControlUnit : inout std_logic_vector(15 downto 0);--
from and to Control unit
Carry_Out : out std_logic;
Flags : out std_logic_vector(5 downto 0);
Enable : in std_logic; --from control unit
Reset: in std_logic:='Z'
);
end component;

component TriALU is
port(
Clock : in std_logic;
ALU_Operation: in std_logic_vector(4 downto 0);-- TriALU
Operations
Data_in_out_ControlUnit : inout std_logic_vector(23 downto
0):="ZZZZZZZZZZZZZZZZZZZZZZZZ";-- from and to Control unit
Vector_Length : in std_logic:='0';
```

```
Carry_Out : out std_logic:='0';
Flags : out std_logic_vector(9 downto 0):="ZZZZZZZZZZ";
Enable : in std_logic:='Z';
Reset : in std_logic:='Z';
Complete : Buffer std_logic
);
end component;

--buss between control uint and alu
signal ALU_OPERATION_BUS : std_logic_vector(2 downto 0);
signal ALU_DATA_BUS: std_logic_vector(15 downto 0);
signal CARRY_OUT_BUS: std_logic;
signal FLAGS_BUS: std_logic_vector(5 downto 0);
signal ENABLE_BUS: std_logic;

--buss between control uint and trialu
signal TriALU_OPERATION_BUS : std_logic_vector(4 downto 0);
signal TriALU_DATA_BUS: std_logic_vector(23 downto 0);
signal TriALU_VECTOR_LENGTH_BUS: std_logic;
signal TriALU_CARRY_OUT_BUS: std_logic;
signal TriALU_FLAGS_BUS: std_logic_vector(9 downto 0);
signal TriALU_ENABLE_BUS: std_logic;
signal TriALU_COMPLETE_BUS: std_logic;

begin

ALU_INST : ALU port map(
Clock =>CPU_Clock,
ALU_Operation => ALU_OPERATION_BUS,-- ALU Operations
Data_in_out_ControlUnit => ALU_DATA_BUS,-- from and to Control
unit
Carry_Out => CARRY_OUT_BUS, -- to control unit
Flags => FLAGS_BUS, -- to control unit
Enable => ENABLE_BUS, --from control unit
Reset => Reset
);

TRIALU_INST : TriALU port map(
Clock =>CPU_Clock,
ALU_Operation => TriALU_OPERATION_BUS,-- TriALU Operations
Data_in_out_ControlUnit => TriALU_DATA_BUS,-- from and to Control
unit
Vector_Length => TRIALU_VECTOR_LENGTH_BUS,
Carry_Out => TriALU_CARRY_OUT_BUS, -- to control unit
Flags => TriALU_FLAGS_BUS, -- to control unit
Enable => TriALU_ENABLE_BUS, --from control unit
Reset => Reset,
Complete => TriALU_COMPLETE_BUS
```

```
);

CU_INST : Control_Unit
generic map(
Address_Bus_Size => Address_Bus_Size,
Board_frequency => Board_frequency
)
port map(
Clock =>CPU_Clock,
Reset =>Reset,
Control_Flags =>CPU_Flags,
--Memory Handling
Request_Address => Address_Bus,-- Memory Address and io
Data_in_out_Memory => Data_Bus,-- from and to Memory
Memory_Control => control_Bus,  -- 1 for read 0 for write
Memory_Enable => Memory_Enable,
--IO Handling
IRQ => IRQ,
--ALU Handling
ALU_Operation =>ALU_OPERATION_BUS,-- ALU Operations
Data_in_out_ALU => ALU_DATA_BUS,-- from and to ALU
ALU_Enable => ENABLE_BUS,
ALU_Carry_Out => CARRY_OUT_BUS, -- from alu
ALU_Flags => FLAGS_BUS,          -- from alu
--TriALU Handling
TriALU_Operation => TriALU_OPERATION_BUS,-- TriALU Operations
Data_in_out_TriALU => TriALU_DATA_BUS,
TRIALU_vector_Length => TRIALU_VECTOR_LENGTH_BUS,
TriALU_Enable => TriALU_ENABLE_BUS,
TriALU_Carry_Out => TriALU_CARRY_OUT_BUS,
TriALU_Flags => TriALU_FLAGS_BUS,
TriALU_Complete => TriALU_COMPLETE_BUS
);


end behave;
```

Computer component:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Computer is
port(
Clock : in std_logic; -- System Clock
Reset : in std_logic:='Z';
Flags : out std_logic_vector(5 downto 0); -- Flags from control
unit
Interrupt_trigger : in std_logic;
Input_Port : in std_logic_vector(7 downto 0);
Output_Port : out std_logic_vector(7 downto 0)
);
end entity;



architecture behave of Computer is
constant Address_Bus_Size : integer := 7;
constant Board_frequency : integer := 5; -- Altera board 50e6
--> 5 for halt test

component CPU is
generic (
Address_Bus_Size : integer;
Board_frequency : integer -- Altera board 50e6
);
port(
CPU_Clock : in std_logic;
Reset : in std_logic:='Z';
CPU_Flags : out std_logic_vector(5 downto 0);

signal Control_Bus : out std_logic;-- control memory and IO
Controller

--buss between CPU and memory and IO Controller
signal Data_Bus: inout std_logic_vector(31 downto 0);
signal IRQ : in std_logic;
signal Memory_Enable : out std_logic;
signal Address_Bus: out std_logic_vector(Address_Bus_Size downto
0)
);
```

```
end component;

component Memory is
generic (Address_Bus_Size : integer );
port(
Clock : in std_logic;
Reset: in std_logic:='Z';
Read_or_Write    : in std_logic;
Data_in_out : inout std_logic_vector(31 downto 0);
Enable : in std_logic;
Address_in : in std_logic_vector(Address_Bus_Size downto 0)
);
end component;

component IO_Controller is
generic (Address_Bus_Size : integer );
port(
Clock : in std_logic;
Interrupt_Request : out std_logic:='Z';
Interrupt_trigger : in std_logic:='Z';
Reset : in std_logic:='Z';
Read_or_Write    : in std_logic;
Data_in_out : inout std_logic_vector(27 downto 0);
Address_in : in std_logic_vector(Address_Bus_Size downto 0);
Input_Port : in std_logic_vector(7 downto 0);
Enable : in std_logic:='Z';
Output_Port : out std_logic_vector(7 downto 0)
);
end component;

--buss between CPU and memory and IO_Controller
signal CONTROL_BUS : std_logic;
signal DATA_BUS: std_logic_vector(31 downto 0);
signal IRQ_BUS : std_logic;
signal ADDRESS_BUS: std_logic_vector(Address_Bus_Size downto 0);
signal MEMORY_ENABLE : std_logic;

begin


CPU_INST : CPU
generic map(
Address_Bus_Size => Address_Bus_Size,
Board_frequency => Board_frequency
)
port map(
CPU_Clock =>Clock,
Reset =>Reset,
```

```
CPU_Flags =>Flags,
--Memory Handling
Address_Bus => ADDRESS_BUS,-- Memory Address
Data_Bus => DATA_BUS,-- from and to Memory
Control_Bus => CONTROL_BUS,  -- 1 for read 0 for write
IRQ => IRQ_BUS,
Memory_Enable => MEMORY_ENABLE
);

MEM_INST : Memory
generic map(Address_Bus_Size => Address_Bus_Size)
port map(
Clock => Clock,
Reset => Reset,
Read_or_Write => CONTROL_BUS,
Data_in_out => DATA_BUS,
Enable => MEMORY_ENABLE,
Address_in => ADDRESS_BUS
);

IO_INST : IO_Controller
generic map(Address_Bus_Size => Address_Bus_Size)
port map(
Clock => Clock,
Reset =>Reset,
Interrupt_Request => IRQ_BUS,
Interrupt_trigger => Interrupt_trigger,
Read_or_Write    => CONTROL_BUS,
Data_in_out  => DATA_BUS(27 downto 0),
Address_in => ADDRESS_BUS,
Input_Port => Input_Port,
Enable => MEMORY_ENABLE,
Output_Port => Output_Port
);

end behave;
```