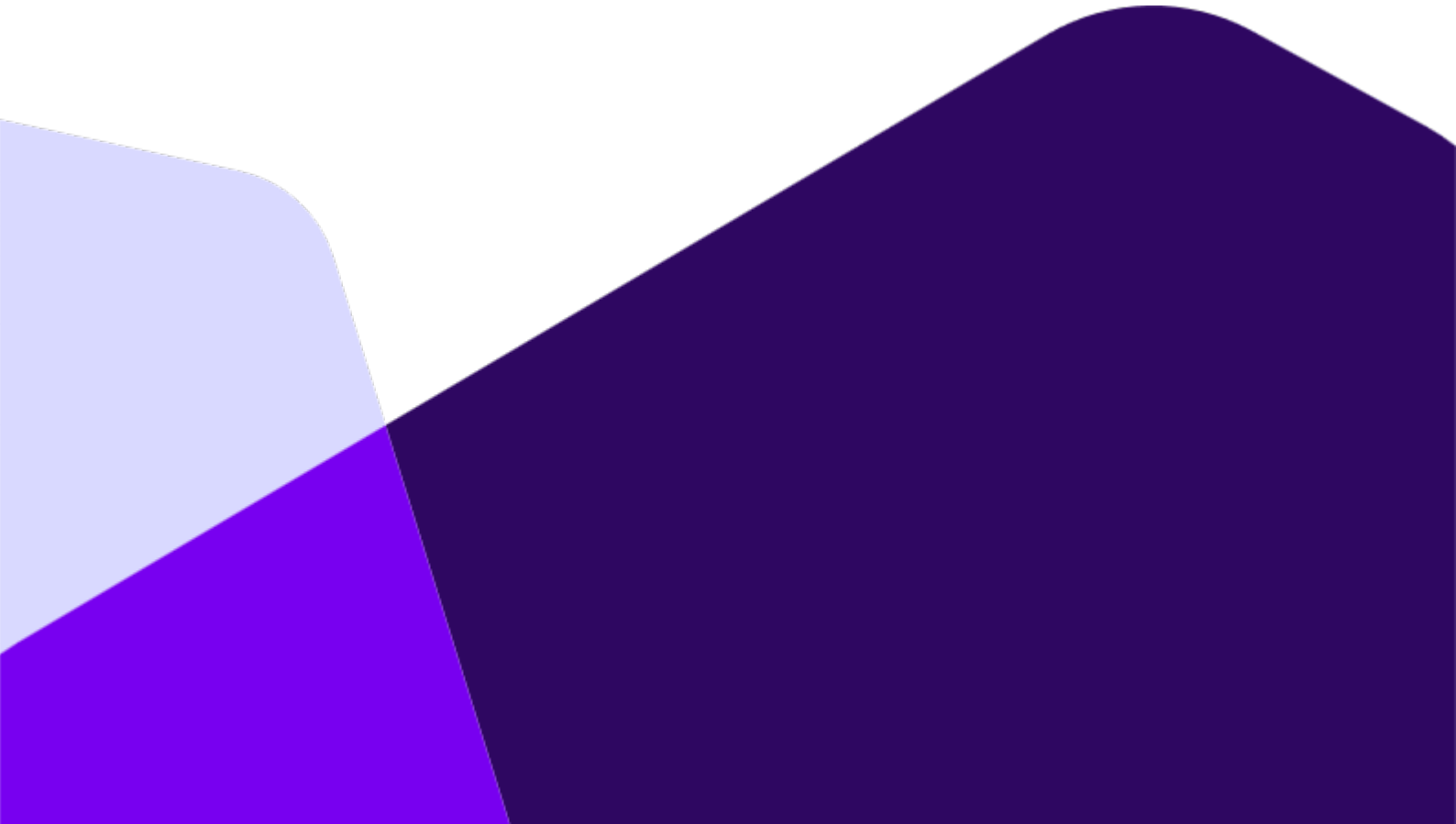
University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
—
Master of Science in Computer Science
Department of Science and Industry Systems
May, 2024

Ole Christian Moholth

Exploring Ternary Computing: Design of a Superscalar CPU and Carry-Lookahead Adder



University of South-Eastern Norway
Faculty of Technology, Natural Sciences, and Maritime Sciences
Department of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

`http://www.usn.no`

This thesis is worth 60 study points

University of
South-Eastern Norway

www.usn.no

A MSc thesis in Computer Science

Exploring Ternary Computing: Design of a Superscalar CPU and Carry-Lookahead Adder

**Thesis submitted to the University of South-Eastern Norway
for the degree of Master of Science (MSc)**

Ole Christian Moholth

Supervisors

Henning Gundersen

Steven Bos

Faculty of Technology, Natural Sciences and Maritime Sciences

Campus Kongsberg

Preface

This thesis is being submitted to partially fulfil the requirements for the Master of Science in Computer Science from the Department of Science and Industry Systems at the University of South-East Norway (USN). The thesis work was carried out between August 2023 and May 2024 under the guidance of Henning Gundersen and Steven Bos. Their expertise and mentorship were instrumental in this academic endeavour.

Acknowledgements

I want to express my heartfelt gratitude to the University of South-Eastern Norway for allowing me to complete this master's thesis.

I especially wish to thank my thesis advisors, Dr. Henning Gundersen and Dr. Steven Bos, for their patience, guidance, and support throughout the research process. Your help has been invaluable and inspiring.

Daniel Larsson and Arvid Siqveland have been enormously patient maths professors throughout my studies. This thesis involved figuring out the notation for ternary mathematics, and their encouragement and input have enabled me to understand how I wanted to express my ideas.

I am grateful to Vetle Bodahl, Ståle Rudin and Lars Rustand for editing help, moral support, and laughter. Sharing the workspace with you has been inspiring and motivating.

Another person I would like to mention is my uncle, Emil Moholth. I had the advantage of growing up among computer scientists, academics, and engineers. However, Emil was the one who had the patience to sit down with an inquisitive 15-year-old and put him on a programming path. I am so grateful for the week you spent with us in Preston that summer.

I am deeply grateful to Emil, my sister Hilde Marie, and her partner Joakim Fauskrud for their endless patience and invaluable assistance. Their meticulous reviews and constructive feedback on form, style, structure, and LaTeX were essential in polishing this thesis. Family is everything!

Lastly, I would like to mention the rest of my family, who have shown endless patience, understanding, nagging, and support. Your role in ensuring I remained focused on my task and did not starve to death in front of my laptop was crucial. I would not have survived the last weeks without you.

Ole Christian Moholth
Kongsberg, May, 2024

Abstract

Abstract

This thesis explores the design and architecture of a balanced ternary *RISC-V-like Energy efficient Balanced tErnary Logic* (REBEL)-2 Central Processing Unit (CPU). There is a growing interest in ternary computing due to the potential performance and efficiency benefits. The development of this type of system has its own challenges and pitfalls due to the differences and complications that can arise from its use. It continues the MSc work of Erica Fegri [1] on ternary counter design and the PhD work of Steven Bos [2] on the REBEL-2 Instruction Set Architecture (ISA) and ternary CPU design.

The research examines several important topics:

- The current implementations of fast adder structures in balanced ternary logic
- The implementation of a carry-lookahead adder for balanced ternary
- The design of a balanced ternary CPU implementing the REBEL-2 ISA
- The required additions to support a superscalar architecture, allowing for parallel out-of-order process execution with in-order completion

This research proposes a solution for a balanced ternary carry-lookahead adder and explains the method of finding the solution. It also proposes a CPU design that implements REBEL-2 and shows a general design for a superscalar implementation. Some key components that should be present in the superscalar design have been identified, and a path forward to further develop the design is presented.

This research not only enhances the understanding of fast balanced ternary adders but also demonstrates its practical application. The implementation of the adder serves as a verification of the logic, showcasing the relevance of this research.

Keywords
Adders, Ternary Arithmetic, Microprocessors, Multivalued logic, Parallel Processing

List of Figures

List of Tables

Contents

Glossary

Glossary

ALU-* the ALU we designed for the REBEL-2 architecture in a ternary system

barrel shifter a digital circuit used in computer processors and other digital systems to perform bitwise shifting operations efficiently. Unlike a simple shift register, which shifts bits one position at a time, a barrel shifter can shift data by a specified number of positions in a single operation, making it much faster for certain applications.

radix-n a numbering system or numeral system based on a given base, where "n" represents the radix or base of the system. In other words, radix-n numeral systems are those in which the position of each digit represents an increasing power of "n".
In the **binary** system (radix 2), the base is 2, and the position of each digit represents a power of 2.
In the **ternary** system (radix 3), the base is 3, and the position of each digit represents a power of 3.
In the **septenary** system (radix 7), the base is 2, and the position of each digit represents a power of 7.
In the **decimal** system (radix-10), the base is 10, and each digit's position represents a power of 10.

ternary A ternary computer uses the ternary logic (base 3) instead of the more common binary system (base 2) in its calculations. Ternary computers use trits instead of binary bits.

trit (plural trits) The ternary equivalent of a bit. A fundamental unit of information that can take any of three distinct states.

word size the number of bits/trits processed by the computer's CPU in one operation. It is a fundamental characteristic of computer architecture and has a significant impact on the performance and capabilities of the system.

Acronyms

Acronyms

ABC The Atanasoff–Berry computer

ASIP Application-Specific Instruction Processor

ALU Arithmetic Logic Unit

CDB Common Data Bus

CMOS Complementary Metal-Oxide Semiconductor

CNTFET Carbon NanoTube Field-Effect Transistor

CPU Central Processing Unit

IMRaD Introduction, Methods, Results, and Discussion

ISA Instruction Set Architecture

MRCS Mixed Radix Circuit Synthesizer

MVL Multi Value Logic

MVP Minimum Viable Product

OoO Out-of-Order

RAW Read After Write

REBEL RISC-V-like Energy efficient Balanced tErnary Logic

RISC Reduced Instruction Set Computer

RQ Research Question

RTL Register-Transfer Level

SFG Semi-Floating Gate (transistors)

SPICE Simulation Program with Integrated Circuit Emphasis

VLSI Very-Large-Scale Integration

WAW Write After Write

WAR Write After Read

—"Omnibus ex nihilo ducendis sufficit unum" - "In order to produce everything out of nothing, one is sufficient"

Gottfried Wilhelm Leibniz

1 Introduction

The CPU is at the heart of every computing system, orchestrating the execution of countless instructions that drive modern technology. At the CPU's core lies the Arithmetic Logic Unit (ALU), responsible for the heavy lifting of computational tasks. With addition operations constituting a significant portion of these tasks, optimising the speed and efficiency of addition is paramount. This research focuses on developing faster and more efficient ternary adders to help computing processes run faster and more effectively.

This thesis explores the realm of superscalar balanced ternary computing and addresses a critical knowledge gap in the development of fast ternary adders. By enhancing the performance of ternary adders, this research aims to help advance the knowledge of ternary processing, pushing the boundaries of what is achievable in high-speed and low-power processing.

This research continues the work in Bos [2] using the two new tools Bos introduced: *Mixed Radix Circuit Synthesizer* (MRCS) and REBEL-2, described in Section 2.4. MRCS is a circuit synthesis tool and REBEL-2 is a modern ternary-first CPU and ISA with reusable open-source building blocks.

1.1 Thesis Objectives

This thesis aims to identify and illuminate fundamental challenges and research gaps in ternary computing and address these challenges, paving the way for the development of

innovative ternary computing systems that push the boundaries of computational capabilities.

1.1.1 Problem Statement

We want to investigate how to design and implement a ternary superscalar CPU that implements REBEL-2 ISA using MRCS presented in Bos [2]

1.1.2 Technical Research Questions

The technical focus of this thesis has been to answer the following research questions:

- **RQ1:** What are the functional requirements of the ALU for the REBEL-2 ISA in a ternary system?
- **RQ2:** How can a balanced ternary CPU be implemented based on the REBEL-2 ISA?

  RQ2 is a complex topic that requires two separate discussions to explore fully:

  - **RQ2.1:** What existing designs of balanced ternary adders can be adapted for high-speed operations, and what are their comparative advantages and disadvantages?
  - **RQ2.2:** How can one design a carry-lookahead adder for a computing system that uses balanced ternary?
- **RQ3:** What modifications are necessary for the core architecture of the balanced ternary CPU to support superscalar processing?

To achieve this, technical research questions and a set of goals have been identified;

1.1.3 Goals for this thesis

- Understand the state of the art for ternary computing through reading related work.
- Create a process to conduct the research.
- Develop a ternary CPU that implements the REBEL 2 CPU design and ISA.
- Evaluate the CPU to identify gaps in current research and design of ternary computing.

- Enhance the CPU.
- Develop a carry-lookahead adder using balanced ternary.
- Provide documentation to facilitate further research on ternary computing.

1.2 Thesis Outline

This document is organised into four main sections, following the IMRaD format: Introduction, Methods, Results, and Discussion. This layout allows readers to find specific information without reading the entire thesis.

Mapping the thesis to the IMRaD format

- The Introduction section of the IMRaD format of the thesis comprises the two first chapters.
- The Method section is addressed in Chapter 3,
- Research is discussed in Chapter 4,
- Discussion needs two chapters and can be found in Chapter 5 and Chapter 6.

Chapter 1 - Introduction

This chapter lays the foundation by presenting the motivation and problem statement, the research questions and approach, the assumptions and limitations, contributions, and finally, the document's outline.

Chapter 2 - Background

This chapter traces computing's evolution from ancient systems to modern CPU design. The goal is to investigate ternary computing as a solution to the limitations of binary-based systems. This thesis explores research and theory to identify gaps and suggest future investigations.

Chapter 3 - Research Methodology

This chapter focuses on research methodology. It describes the model, methods, and techniques for answering research questions. The thesis approach is clearly explained, and the limitations, assumptions, and constraints of the chosen methods are discussed.

Chapter 4 - Design and Results

This chapter explains in detail the efforts to achieve the main objective of designing a ternary CPU, and of designing a fast ternary adder. It explains all the details and concepts of balanced ternary carry-lookahead, and the expressions to make it work for 3 trits. It introduces the ALU—* design, a ternary ALU specifically tailored to support REBEL-2. The chapter then discusses the essential components to design and implement a ternary CPU. Finally, it explores the requirements for scaling up to a superscalar architecture.

Chapter 5 - Discussion

This chapter summarises the findings of the previous chapter and limitations of the project.

Chapter 6 - Conclusion

The final chapter answers the research questions from Chapter 1, highlights the outcomes and contributions, gives recommendations for future work, and concludes the document.

Appendices

In addition to these chapters, the thesis contains the following appendices.

- **Appendix A** - DSR Implementation
- **Appendix B** - Workflow
- **Appendix C** - CPU Design Stages
- **Appendix D** - Testing
- **Appendix E** - Standard Notation of Lookahead Logic
- **Appendix F** - About Balanced Carry and Addition
- **Appendix G** - Lookahead Expressions With More Trits
- **Appendix H** - Components Implemented in MRCS

—"In the history of science, we often find that the study of some natural phenomenon has been the starting point in the development of a new branch of knowledge."

C. V. Raman

2 Background

This chapter briefly explores the historical evolution of computing, from ancient numerical systems to the present-day challenges of CPU design. The ultimate goal is to examine the potential of ternary computing as a solution to the limitations of binary-based systems. By exploring existing research and theoretical frameworks, this thesis will identify gaps in current understanding and suggest avenues for future research.

2.1 History

One of the most basic "decisions" man has made in history is which number base represents numbers. The most common choice for humans is decimal, using the number 10 as a base. Buchholz [3] gives us examples that some ancient civilisations had already used two as a base and suggests that using your fingers (digits) or fists when counting is a choice. The digital system has advantages for humans since the larger "alphabet" makes reading and interpreting numbers easier.

The binary number system is the basis of today's computing. This discovery is often credited to Gottfried Wilhelm Leibniz in the 17th century. He wrote a paper, 'Explication de l'Arithmétique Binaire', or 'Explanation of Binary Arithmetic,' Leibniz [4] in which he proposed a binary digit system. However, earlier civilisations used something that resembled a binary system. In China, "The I Ching", or "Book of Changes", possibly published 3-400 BC, used a binary-like system to develop religious symbols Ifrah [5].

The Entscheidungsproblem, German for the "decision problem", was formulated by David Hilbert and Wilhelm Ackermann in 1928 [6]. This problem involves searching for an algorithm capable of taking a statement as input and determining whether it is universally valid, meaning true in every possible structure, by providing a "yes" or "no" answer. To determine if this problem can be solved, Alan Turing published a paper in 1936, "On Computable Numbers, with an Application to the Entscheidungsproblem" [7]. In the paper, Turing spent much of the time describing what could be a real machine.

Konrad Ernst Otto Zuse designed the Z1 [8]. This electrically driven mechanical calculator read instructions from a punched 35mm celluloid film. Zuse built it at his parents' home from 1936 to 1938, entirely financed by private funds. This was the world's first known programmable computer with Boolean logic and binary floating-point numbers, but it was unreliable. This computer was destroyed in the bombardment of Berlin in December 1943, during World War II, together with all construction plans.

The Atanasoff–Berry Computer (ABC) was the first automatic electronic digital computer. Due to the limitations of its time, it remains somewhat obscure. Historians debate its importance because it was not programmable or Turing-complete [1]. However, it is considered the first electronic ALU in modern processors. Designed in 1937 by Iowa State College professor John Vincent Atanasoff and graduate student Clifford Berry, the ABC was built to solve systems of linear equations and was successfully solved in 1942. It introduced key elements of modern computing, such as binary arithmetic and electronic switching. However, its use of an imperfect paper card writer/reader for storing intermediate results and the halt of its development when Atanasoff left for World War II assignments limited its impact. Atanasoff and Berry's computer work was not widely known until it was rediscovered in the 1960s, amid patent disputes over the first instance of an electronic computer [9]. The ABC was recognised as an IEEE Milestone in 1990.

John von Neumann was a mathematician who developed the basic architecture for modern computers. He met Alan Turing several times, and Turing's paper [7] is quoted as having been the ispiration Randell [10, p. 10] when he created the Von Neumann machine. This machine remains the foundational blueprint for the modern computer, known as the "classical computer". The concept was developed by three key figures of the Institute for Advanced Study Electronic Computer Project: Arthur Burks, Herman Goldstine, and John von Neumann, in their groundbreaking paper "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" [11]. This is often considered the official introduction document to computer science. Several researchers contributed ideas to the paper, but von Neumann emerged as its primary author. This paper defined *the principles*

[1] A system is Turing-complete if it can simulate any computation or algorithm that a Turing machine can, given sufficient time and resources.

of modern computers, but the one crucial principle for this research is the ”application of the binary system”.

2.2 Situation today

Today, computer design has hit a wall. The CPUs' clock rate and power consumption are correlated, and we have reached the practical power limit to cool the system. A major reason for this is the breakdown of the Dennard scaling in 2005 Kanduri, Rahmani, Liljeberg *et al.* [12]. Dennard, Gaensslen, Yu *et al.* [13] states that the power consumption of a device is in proportion to the area. This was true because the supply voltage of the CPU was scaled down with the size of the transistors. The problem is that a lower voltage increases the leakage current, leading to power consumption and cooling problems. Patterson and Hennessy [14] state that about 40% of power consumption in server chips today is due to leakage current.

Despite the potential efficiency gains offered by ternary systems, research on comprehensive ternary CPU architectures and fast balanced ternary components is limited. Earlier efforts have focused on specific aspects rather than holistic CPU architectures. Addressing these gaps is essential for the advancement of energy-efficient and high-performance computing systems.

2.3 Related work

This investigation builds on the MSc work by Erica Fegri [1] on ternary counter design and the PhD work by Steven Bos [2] on ternary CPU design by creating MRCS and some of the components required for a ternary microprocessor. As is apparent from reviewing previous work on this study area, much work remains to be done.

So far, there has been limited research on ternary and superscalar CPUs. Furthermore, there has been limited research on the implementation of REBEL-2 and fast ternary adders. Understanding the theory of ternary computing and finding useful related literature on the relevant topics has been an integral step in the iterations of this work, and the findings throughout the work have been summarised in this section.

The following papers provided a broad perspective and understanding of ternary systems' viability and potential benefits. They are organised into sections corresponding to the

timeline of this investigation. Although some of these papers could be relevant to multiple sections, they are included based on their initial usefulness.

2.3.1 Ternary vs Binary

This section covers various aspects of ternary versus binary computing, including theoretical efficiency, practical applications, and design challenges.

The integration of binary and ternary logic systems is discussed in Schoenbaum and Al-Assadi [15]. This paper explores their efficiency and applications in reversible computing and system programming and gives us a multifaceted view of binary/ternary computing systems.

Alexander [16] theorises that a ternary counting system could require less equipment than a binary system for the same number handling capacity. The challenge of developing circuits with three stable states is discussed.

Obiniyi, Ezugwu and Kwanashie [17] introduce a novel method of representing ternary states using colour codes. This paper proposes logic design models for ternary adders and explores the advantages of ternary systems over binary ones. This includes reduced hardware complexity and power consumption.

A new binary and ternary coding comparison is discussed in Kak [18]. It looks at their efficiency in number representation and decision trees, while highlighting potential advantages in ternary coding classifications relevant to AI and medical applications.

2.3.2 Work Related to MRCS and REBEL-2

The initial problem statement (subsection 1.1.1) of this thesis is based on the tools created by Bos [2]. This part of *Related Work* covers the reading available on these tools.

Bos, Nilsen and Gundersen [19] introduces a hardware-based ternary memory controller for memristors, the controller stores ternary sign as ”trits” to control a robotic actuator in real-time, building upon previous research.

Bos, Gundersen and Sanfilippo [20] presents an open-source framework implemented in Unity (C#) that allows for control and interaction with memristors that store and retrieve ternary states. The framework includes a closed-loop ternary memory controller, making it suitable for both PC and real-time embedded applications that require ternary operation.

Bos, Risto and Gundersen [21] presents a new tool, Mixed Radix Circuit Synhesizer (MRCS) for Electronic Design Automation that supports Multi Value Logic (MVL) and facilitates the synthesis of circuits using Carbon NanoTube Field-Effect Transistors (CNTFET). The tool supports binary, ternary, and hybrid formats and features various design, simulation, and verification tasks.

In Bos and Gundersen [22], IoT devices' advantages and future impact in seven categories are discussed: computational power, communication, compression, comprehension, cybersecurity, design complexity, and energy consumption.

Risto, Bos and Gundersen [23] explores the automatic generation of netlists for ternary-valued n-ary logic functions in CNTFET circuits using a static ternary gate design approach. A C++ implementation that generates Simulation Program with Integrated Circuit Emphasis (SPICE) subcircuit netlists for ternary-valued n-ary function circuits is presented, as well as a circuit schematic for synthesising a 3-operand carry. The study also examines the design of balanced full adders using three gate-level design approaches (compound, noncompound, and hybrid) and compares them to each other and existing methodologies. Additionally, the study proposes standardising ternary functions through indexing to address the lack of semantic names for most ternary functions and the exponential growth of unique 3-valued functions with higher parity.

Bos, Risto and Gundersen [24] presents a method for high-speed conversion between binary and ternary to allow binary devices to work with non-binary ones. The method uses CNTFETs and a balanced ternary full adder and was simulated using HSPICE 2020. The simulation showed that nibble word conversion speeds of more than 25 GB / s with power consumption of 97.8 W are achievable with 1760 CNTFETs switching at 5 GHz.

2.3.3 CPU Design

Understanding ternary thinking and design is not covered in standard computer science education. Solving the initial problem and creating ternary designs required studies on this topic.

A significant portion of existing research has focused on lower-level hardware design, with little research on the design or synthesis of a ternary microprocessor. Previous research has proposed a 5-stage pipelined microprocessor. Kam, Min, Yoon *et al.* [25] introduced a design and verification framework for developing a fully functional emerging ternary processor. They implemented their proposed microprocessor on an FPGA-based platform using a binary-encoded ternary number system. They found that there may be benefits to energy consumption and memory cell count when a ternary design is used.

Gadgil, Sandesh and Kumar V [26] presented a proposal to design and implement a CNTFET-based Ternary logic processor. They covered the processors' fundamental blocks like the instruction fetches, register files, data memory, and control unit designs; they performed simulations for the processor and verified the functionality using a few benchmark programs.

Patterson and Hennessy [14] write that a ternary system could be a step towards more efficient computing. By increasing the radix, the number of transistors and memory cells required could be reduced by up to log (3) / log (2), or about 58.5%.

Shen and Lipasti [27] describe techniques and concepts for instructional parallelism and the fundamentals of superscalar processors. They cover various techniques to implement and improve superscalar processors, memory management, and how to handle branch predictions to avoid having to flush the pipelines in the case of mispredictions. This book also discusses how these techniques have been used in earlier processors.

SonicBOOM, the third generation of the Berkeley Out-of-Order (OoO) Machine (BOOM)Zhao, Korpan, Gonzalez *et al.* [28]. SonicBOOM is an open-source Register-Transfer Level (RTL) implementation of a RISC-V superscalar out-of-order core that provides a state-of-the-art platform for research in high-performance OoO design.

Large FPGA systems need powerful soft processors for complex tasks such as network software stacks. Out-of-order (OoO) designs can boost performance, but do not fit in FPGAs. Mashimo, Fujita, Matsuo *et al.* [29] introduce the RSD processor: a RISC-V soft processor tailored for FPGAs. It is efficient and outperforms other soft processors by using fewer FPGA resources.

2.3.4 Superscalar Design

The design of a superscalar CPU was one of the initial criteria for this investigation. The following papers cover the theory on this topic.

Tomasulo [30] described as early as 1967 the methods employed in the floating-point area of the System/360 Model 91 to exploit the existence of multiple execution units. His algorithm is discussed in a later section. He presented several concepts that helps limit stalling and hazards, and is a good starting point for a superscalar CPU design.

Antonov [31] introduces a new method for designing Application-Specific Instruction Processors (ASIPs) using RISC-V architecture. It involves using a programmable hardware

generator to create customized superscalar, OoO processors, enhancing flexibility and efficiency. This approach separates computations and flow control, leading to improved performance for specific applications.

2.3.5 Ternary Logic

Designing a ternary CPU requires a deeper understanding of ternary logic and ternary logic notation, which the following papers cover.

Considerable progress has been made in the implementation of ternary logic, but a universally accepted notation system has not been established. The proposed notation in "Standard Ternary Logic" Jones [32] extends the conventional Boolean notation to accommodate ternary logic. It introduces new symbols where necessary and allows for a combination of Boolean and ternary logic. The final part of the discussion focuses on optimising these logic functions.

The potential of ternary logic in digital computers is the subject of Merrill [33]. The paper provides a historical perspective on ternary systems and evaluates advantages such as cost, reliability, and operating speeds. The technical information is focused on threshold logic, which this thesis does not explore.

Halvor Risto [34] discusses ternary logic and CNTFETs, covering concepts such as ternary notation, logic functions, and the advantages of CNTFETs over MOSFETs. The thesis describes methods for designing and optimising ternary CNTFET circuits, using a logic synthesiser to create and simulate full adder circuits. Data conversion between binary and ternary systems is explored, showing high-speed conversion with low power consumption using CNTFET circuits. Finally, the thesis compares ternary and binary full adder circuits, concluding that ternary adders do not outperform binary ones in power and efficiency with the CNTFET circuits used.

2.3.6 Ternary Adders

While designing a ternary CPU, it became apparent that finding or designing a fast adder was important. This was a substantial knowledge gap, which led to a rework of the research questions in cooperation with the supervisors. Most existing research has focused on binary or unsigned ternary, where the logic for carry differs significantly. The papers in this section represent some of the previous work on fast adders.

In unsigned and 3's-complement ternary arithmetic, digits are represented by the values 0, 1, and 2. However, balanced ternary employs -1, 0, and +1. Balanced ternary is similar to biased unsigned ternary with a half-range bias. "Fast Ternary Addition" by Jones [35] outlines implementations of ternary arithmetic, following the conventional sequence used in presenting binary arithmetic. The progression starts from a half adder to a full adder and subsequently to an ALU with a carry-lookahead logic.

Yoon, Baek, Kim *et al.* [36] introduce ternary Wallace tree multipliers to enhance performance and reduce transistor count compared to existing designs. The paper suggests using ternary Wallace tree multipliers with 4-input ternary adders and proposes a ternary carry-select adder to decrease carry propagation delay. These improvements are evaluated using a custom ternary standard cell library and HSPICE simulation. The results show significant power-delay product improvements for the proposed ternary multipliers compared to previous designs.

Binary adders, including microprocessors and digital signal processors, are essential in digital circuits. Increasing efficiency is crucial to improve the performance of a system. In Very-Large-Scale Integration (VLSI) implementations, parallel prefix adders are highly effective. Vallabhuni [37] explores the architecture and performance of various types of adders, such as carry-select adders, ripple-carry adders and carry-skip adders, using the Vivado Design tool for execution, simulation, verification, and synthesis.

Soliman, Fouda, Said *et al.* [38] have introduced innovative methods to design lookahead adders for ternary carry and provided information on trade-offs between different implementation techniques, which could advance digital circuit design.

Regarding high-speed adders, Ling [39] presents a promising advancement in carry propagation techniques for binary adder circuits, offering potential efficiency, performance, and scalability improvements compared to traditional approaches.

Gundersen and Berg [40] presents a design using Complementary Metal-Oxide Semiconductor (CMOS) semi-floating gates to implement a version of a balanced ternary adder based on threshold logic. This paper describes the potential benefits of this design, and compares it to a binary implementation.

Øverås' [41] master thesis uses semi-floating gate (SFG) transistors to generate MVL signals. The thesis proposes a design using SFG to create the signals used in lookahead adders for unbalanced addition, then correctly determines that binary is sufficient for determining the carry-output based on the P and G signals. The paper attempts to combine multiple carry-lookahead adders into a larger adder and proposes a lookahead scheme to handle this.

Mechanical Advantage [42] proposes an architecture that ensures efficient ternary addition while minimising the need for multiplexers and facilitating the cascading of carry candidates between full select adders. Centralised control of multiplexers with a single carry-in signal also streamlines the adder's operation, contributing to its effectiveness in ternary arithmetic operations. Although this covers some of the topics explored in this thesis, the validity of the information is doubtful. It is discussed more in Section subsection 5.1.1, which discusses *Ternary Adders*.

2.4 REBEL-2 and MRCS

Steven Bos' PhD dissertation Bos [2] is the background for the research in this thesis.

Bos introduced a workflow for designing ternary VLSI with MRCS [21]. This is the first open-source browser-based Electronic Design Automation tool for creating and verifying mixed-radix circuits with CMOS and CNTFET.

REBEL-2 is shown in Figure 2.1 from Bos [2]. This is a modern ternary-first CPU and ISA with reusable open-source building blocks. It embraces the ternary way of thinking, which includes 3 data paths which are great for comparisons and jumps. It also allows for more compact opcodes and instruction formats. Bos makes REBEL-2 in the dissertation and defines the instruction set in the designs presented in Chapter 4. REBEL-2 directly impacts the requirements and design of ALU-* (RQ1) as it determines which components must be present by defining the arithmetic and logical operations of the instruction set. ALU-* must be capable of executing the required mathematical and logical operations, and the output is then stored at the specified location.

2.5 Balanced Ternary Adders

Little research details the design of balanced ternary lookahead adders. Existing research and projects do not design or develop $O(\log n)$ time complexity adders and instead use ripple carry adders. This leaves some space for $O(\log n)$ adders to develop. The use of balanced logic creates additional challenges for designing adders.

Understanding the state of Ternary CPUs is crucial as it forms the basis of this research. This section will delve into the existing implementations of ternary CPUs, explaining what they are and are not.

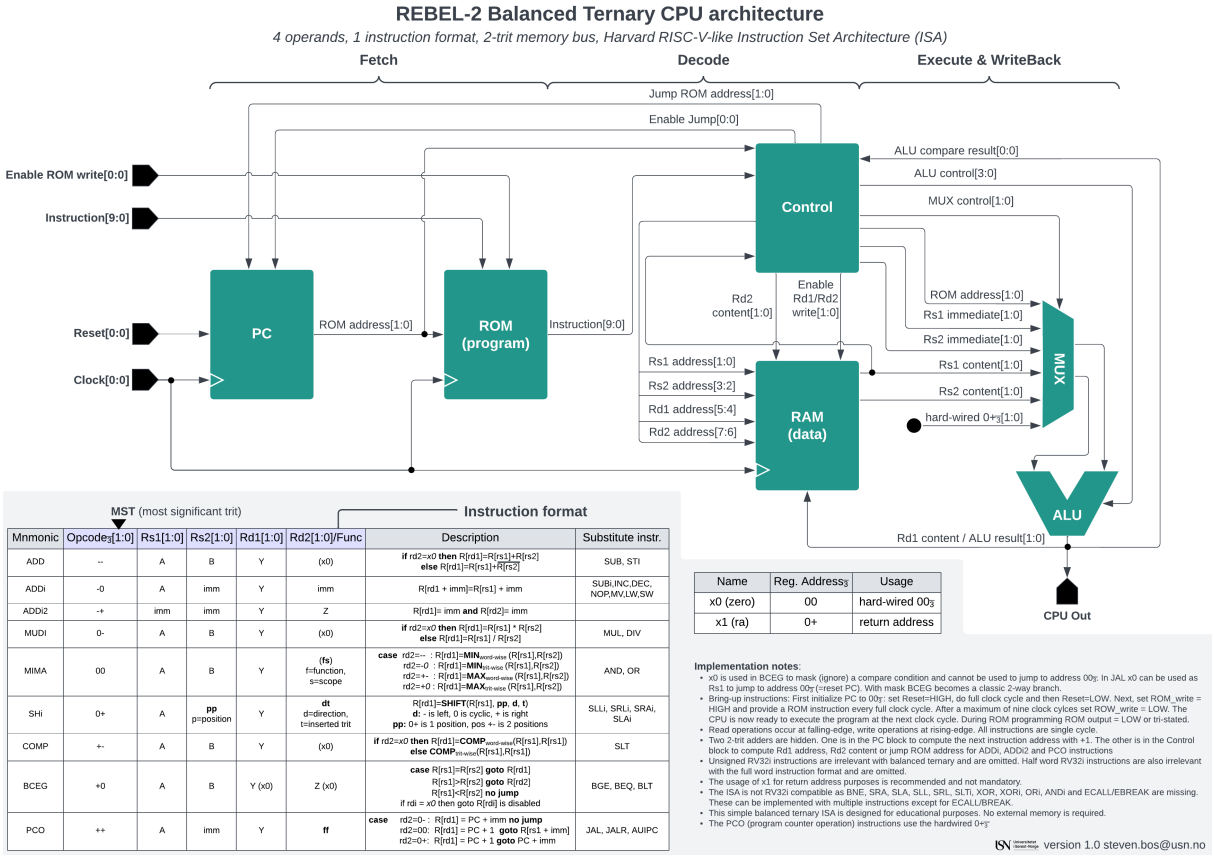


Figure 2.1: REBEL-2 Bos [2]

The variations in ternary computing, namely balanced and unbalanced, are not just different approaches but significant ones because they offer diverse ways of representing and handling ternary logic, which, unlike binary logic, deals with three states such as (0, 1, and -1). Two main types of ternary are explored in the literature:

- Balanced Ternary:
  Balanced ternary uses the digits -1, 0, and 1 to represent the values. Each digit carries an equal weight. With a balanced ternary number, positive and negative numbers are represented similarly.

- Unbalanced Ternary:
  Unbalanced ternary uses the digits 0, 1, and 2 to represent possible values. Negative and positive numbers are represented differently, as there is no way to represent negative numbers with the available digits.

These variations of ternary are represented by three separate states, each representation with different advantages and disadvantages. For ternary arithmetic, balanced ternary allows you to treat all numbers equally, whether they are negative or not. For addition, this means that the carry encompasses one additional state over unbalanced addition. This causes some additional complications that have not been explored in the literature.

2.6 Types of adders

An adder is one of the most commonly used components in an ALU. This section provides a basic overview of a few different adder types available, paving the way for describing and designing the logic for a balanced ternary lookahead adder later Section 4.1.

2.6.1 Ripple Carry Adders

The ripple-carry adder is the simplest adder to implement and functions similarly to how one usually adds numbers using pen and paper. The least significant digits are added together, and the carry signal is propagated to the next digit and added together. This could lead to a carry propagating through every full adder in the circuit like a ripple. The propagation of the signal is high and increases with the number of digits in the adder.

2.6.2 Carry-save Adders

Carry-save adders are a type of adder structure frequently used when adding 3 or more numbers at a time. This type of adder works in multiple steps. Every step, the inputs are two numbers and the carry from an earlier addition. This is repeated until only numbers to add remain, a sum and the previous carry. These can be added using a normal 2 input adder such as a ripple-carry adder.

Yoon, Baek, Kim *et al.* [36] show a design of one such adder for use in a Wallace tree multiplier and highlights the strength of balanced ternary to add 3 numbers and a carry together without ending up with a carry exceeding 1 in size. This quality of balanced ternary is discussed more in Appendix F.

2.6.3 Carry-Select Adders

Another option for a fast adder is carry-select adders. A carry-select adder function calculates the addition result for every carry value and then selects the valid addition based on the carry input. This allows you to break the addition into smaller chunks that you can calculate in parallel with simpler ripple-carrying adders. The implementation is more straightforward than a lookahead adder, especially in the case of balanced ternary, where the complexity of lookahead adders is significantly increased.

The ripple-carry chain is duplicated once for each possible carry value. Once the carry is known, the correct calculation is selected. For binary, this means two; for ternary, three different carry values exist.

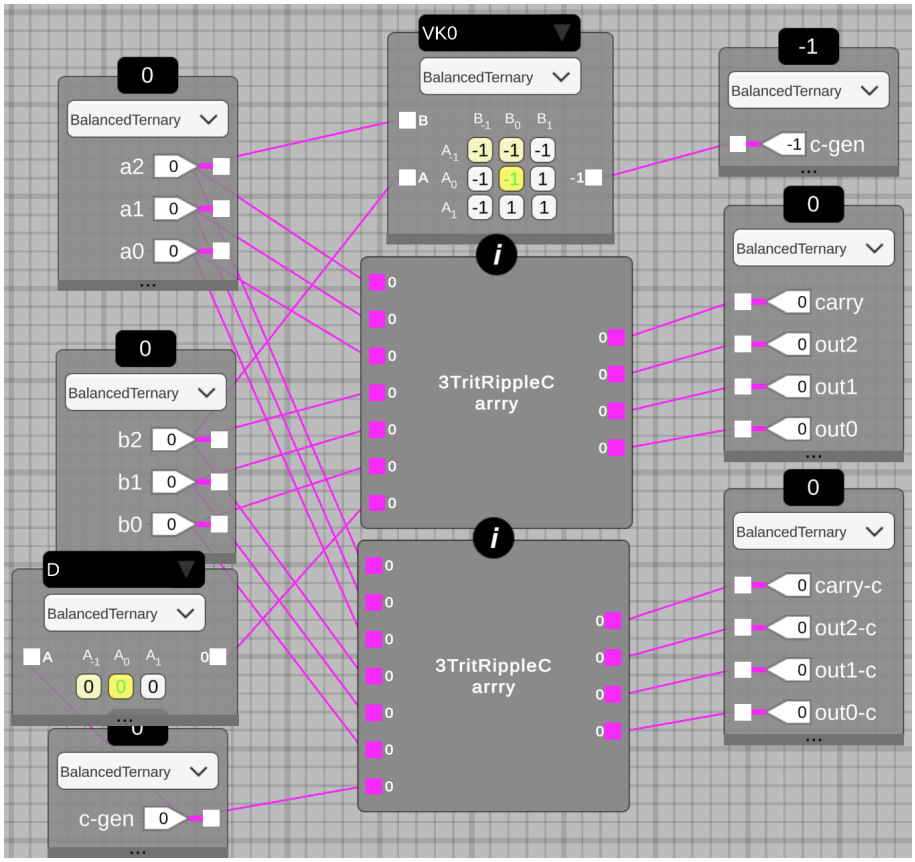


Figure 2.2: Implementation of the carry-select group using two ripple-carry chains

As proposed in Yoon, Baek, Kim *et al.* [36], this can be reduced to only two ripple-carrying chains instead of 3 for balanced ternary. This is possible because the final carry output can be partially determined without knowing the carry input. As will be shown

Section 4.1, positive and negative propagate signals are mutually exclusive and can be determined without knowing the signals. Learning only the most significant bits lets you decide if -1 or +1 could be propagated. By exploiting this, it is only necessary to have one ripple-carry chain for a 0 carry and another ripple-carry chain for a non-0 carry. This will cut the transistor count by almost $\frac{1}{3}$, assuming that the full adders add the most transistors to the circuit. Figure 2.2 and Figure 2.3 show an implementation of a carry-select adder in MRCS. The leftmost logic gate in Figure 2.3 calculates possible carry, if any. The rightmost logic gate checks the carry output from the first three trits and uses the result as the select signal for the multiplexer, which selects the output for whether it is carried or not. Figure 2.2 shows the implementation of the carry-select group. The bottom component calculates the addition for a carry using the generated carry value. The upper component always calculates the sum for a carry of 0. The logic gate determines what carry it can propagate, if any.

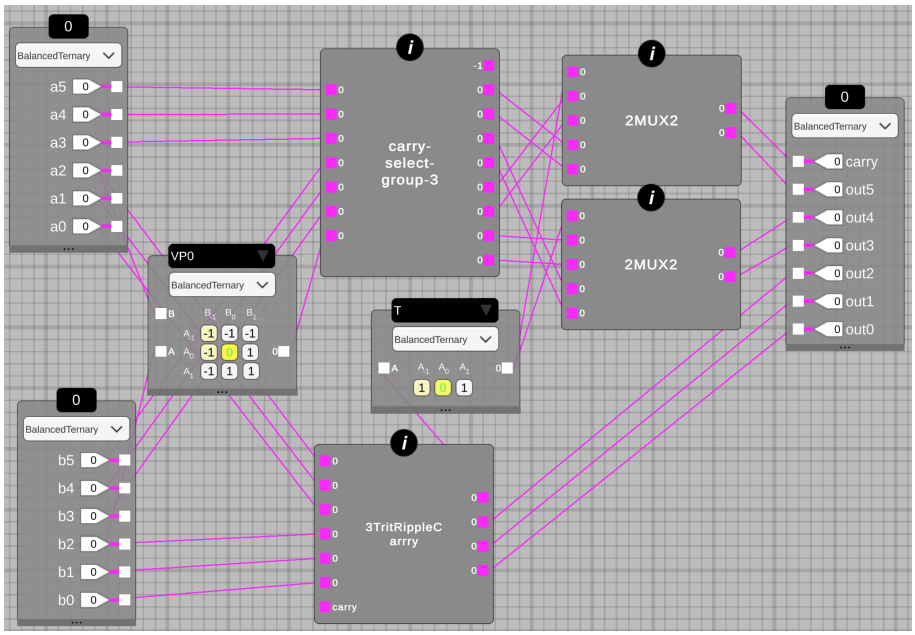


Figure 2.3: Implentation of 6-bit carry-select adder

—"The worse thing that contemporary qualitative research can imply is that, in this postmodern age, anything goes. The trick is to produce intelligent, disciplined work on the very edge of the abyss."

David Silverman, in: [43]

3

Research Methodology

This chapter presents the research model, methods, and techniques to address the research questions. It also elaborates on the rationale behind the approach choices and discusses the limitations, assumptions, and constraints associated with the selected approach.

In a project such as this, it is essential to define a systematic and structured approach to guide the process to ensure success and answer the research questions defined for the thesis. The thesis was developed using the **Design Science Research** (DSR) process Brocke, Hevner and Maedche [44]. DSR was chosen for its iterative process, which allows for ongoing refinement and alignment with evolving project requirements.

This choice was made after considering other methodologies, such as *Experimental Research* and *Case Study Research*, less suitable for the objectives of this project. **Experimental Research** [45] is useful for hypothesis testing, but the focus here was on developing a prototype and understanding its wider impact rather than solely validating the results. **Case Study Research** [46] was unsuitable as the project's goal expanded beyond examining a single event to developing a prototype with wide-ranging applications.

DSR typically follows a methodology that involves several phases. Although there are variations and the methodology must be adapted to fit each project depending on specific contexts, here is a general outline of the common phases.

- **Research** - Problem Identification, Data Collection/Preparation/Analysis
- **Development** - Feature Engineering, Model Development/Evaluation/Deployment

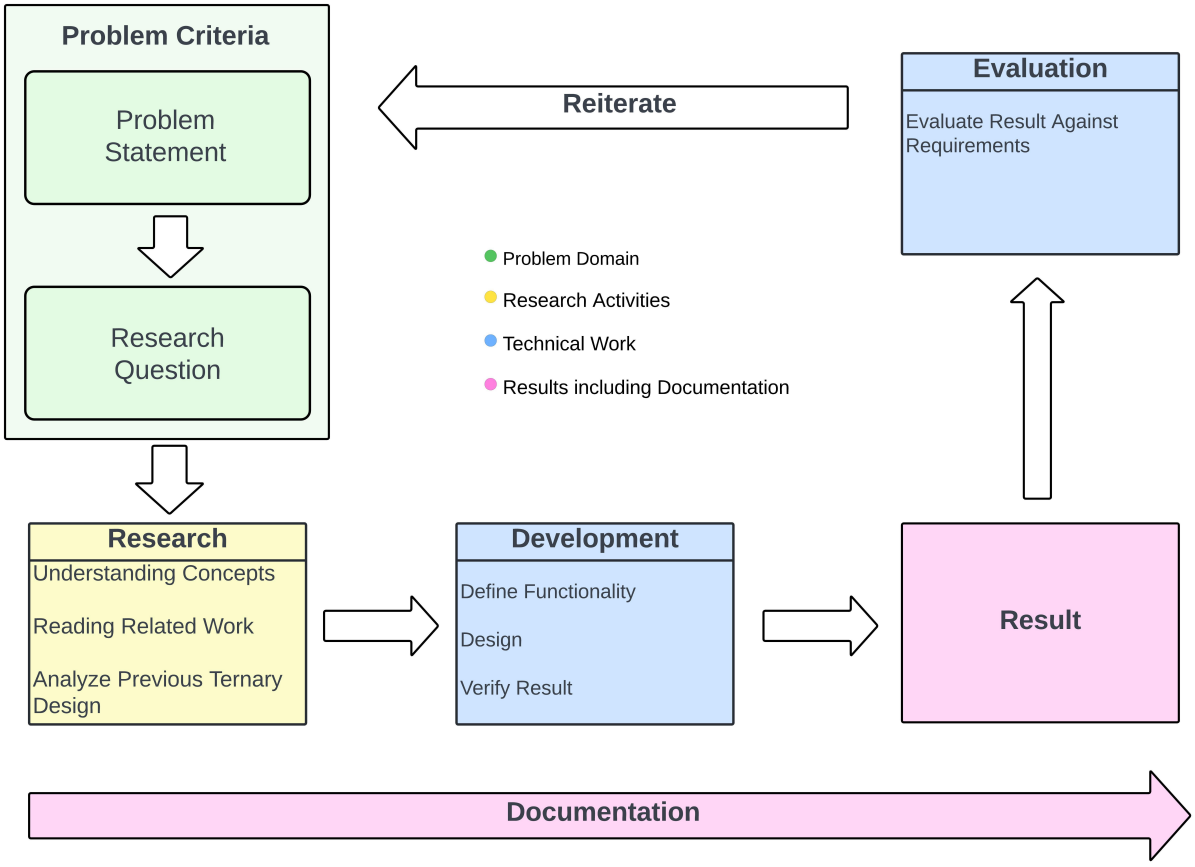


Figure 3.1: DSR Research Flow as applied in this paper

- **Evaluation** - Communication and Visualization, Feedback and Iteration

Following these steps, data scientists can systematically solve research problems, extract meaningful insights from results, and drive decision-making processes. Figure 3.1 illustrates how DSR is adapted to this project by showing the iterative workflow from research goals and questions to results.

The green and yellow boxes represent the first bullet point in the above DSR outline. The green boxes are the starting point for every iteration. The problem statement and the research questions from Chapter 1 are the most important decision criteria to choose the way forward. They are assessed in collaboration with the supervisors, considering current knowledge to determine the next goals and the most promising research path.

Once these decisions are made, the research enters the yellow box, *Research*. This stage has three steps in this project: understanding concepts, reading related work, and analysing previous designs, both from others and the work in earlier iterations.

Development, the first blue box in the figure, is the next stage in the iteration. Here, requirements for the objectives are found, designs are made, the theory is documented, and these results are verified. Verification would typically involve fabricating and testing a chip if this research had a longer time frame. However, the lead time for chip production is so long that the ALU-* design sent for production[1] at the start of the work has not yet been returned. This means that the verification for the designs presented is theoretical and simulated.

Next blue box, *Evaluation* is the last stage in the iterative process. In cooperation with the supervisors, the results are evaluated against the requirements developed in the *Research* stage and against the decision criteria in the green box to determine the new path forward in the research.

This circle was repeated throughout the work. Each cycle had a specific task on the Gantt chart in Section 3.1. The thesis workflow is shown in the Appendix B.

Appendix A documents the steps of the iterative process created using DSR in detail.

3.1 Gantt Chart

Supervisors use the Gantt chart in Figure 3.2 to follow the progress of the work. It is updated throughout the project, and the following is the latest version showing the

[1] https://tinytapeout.com/runs/tt05/107/

activities as they were:

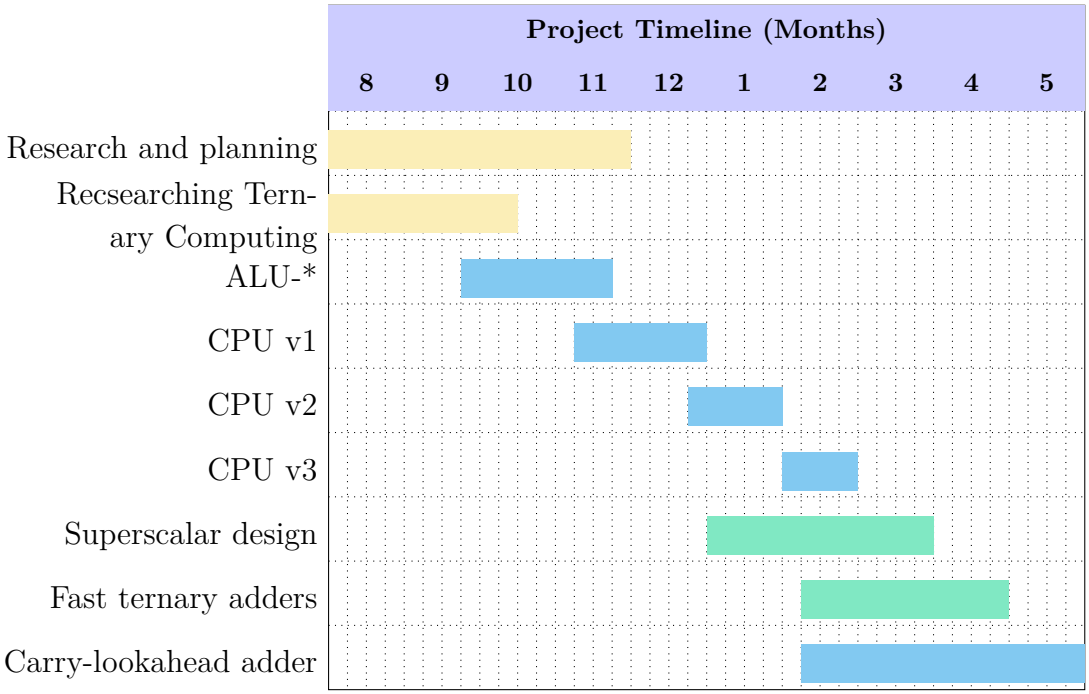


Figure 3.2: Project timeline

In this figure, yellow represents preliminary studies and preparation, blue indicates technical work that results in documented designs, and green signifies technical work that results in documentation only. Understanding this Gantt chart in the context of the thesis journey, as described in Section B.1, is crucial. The graph in Figure 3.2 may give the impression of a predetermined sequential approach. However, this research follows an agile methodology according to the implementation of DSR, as detailed in Appendix A. The colours in both these diagrams are coordinated to help the reader visualise the connection between the stages of the project and their respective outcomes.

3.2 Finding Related Work

This is an activity that takes place in the *Research* stage of the project's DSR implementation. The research is focused on an area that needs more work. It is based on the work done by Steven Bos in his PhD dissertation, Bos [2]. Finding previous work on a complete system was difficult, but smaller individual parts of the problem domain have been studied in isolation.

The most important websites used to find previous work:

- IEEE Xplore IEEE xplore [47] a digital library providing access to a vast collection of research articles, conference papers, and standards in engineering, technology, and related fields
- Google Scholar [48] a freely accessible search engine that indexes scholarly articles, theses, books, and conference papers across various disciplines
- ACM Digital Library [49] a comprehensive database of articles, conference proceedings, and journals published by the Association for Computing Machinery, covering all aspects of computing and information technology
- Scopus by Elsevier [50] a comprehensive abstract and citation database covering peer-reviewed literature, including scientific journals, books, and conference proceedings across diverse research fields
- Web of Science by Clarivate [51] a multidisciplinary research database providing access to scholarly articles, citation indexes, and conference proceedings across various academic discipline

During the search process on these websites, a variety of specific search terms were used, such as *"ternary"*, *"ternary adder"*, *"balanced"*, *"lookahead"* and *"fast adder"*, *"superscalar"*, *"CPU"*, *"microprocessor"*. These terms were chosen to align with key concepts and areas of interest in the investigation to ensure the relevance of the literature review.

In addition to the above-mentioned tools, Google, library databases, and Wikipedia have all helped to find the academic references used in this document.

3.3 Requirements and Design

This is part of the *Developmet* stage. An incremental approach was used to define the requirements for each iteration. This approach makes learning possible throughout the process and makes it possible to pivot if taking advantage of discoveries improves the outcome. The following is a step-by-step explanation of each iteration in the work process:

- Develop a detailed list of project requirements based on supervisor feedback and industry best practices.
- Iteratively refine designs based on the feedback of the supervisor.

- Continually evaluate requirements with the supervisors to see if they have changed as discoveries are made.

The process of finding the specific design and validation are closely tied together. The circuits were designed by defining the functionality as a logic table and implementing it in MRCS. For more complex algorithms that require additional information, the problem needed to be divided into smaller parts for implementation. This was the case for the shift operation, the multiply operation, and the functionality of the carry-lookahead adder.

The CPU architecture presented in Section 4.3 is inspired by the design presented in Patterson and Hennessy [14].

The required logic tables in Section 4.1 were initially designed without sources. The resulting tables and outputs were compared with expected outcomes and mathematical definitions. If applicable, binary functionality was used as a reference for the design. This leads to an empirical process where the functionality is defined, implemented, and verified in that order from the ground up.

3.4 Validating the Results

The next step in the *Development* stage is to validate the results. This is done using MRCS, and the design can be verified using a CSV file. In MRCS, you can define all input values that you wish to test in order and the expected output. This is more described in Appendix D, including the specific tests that were created and run.

In addition to this, the carry-select adders from [42] and Yoon, Baek, Kim *et al.* [36] have also been implemented and tested in MRCS. This could be done with the same tests used for the adder logic presented in Section 4.1 since the results are expected to be the same.

3.5 Technologies

The technologies used to create the designs are heavily influenced by a lack of experience with LaTeX and the absence of previous work in the problem domain.

- MRCS - Mixed Radix Circuit Synthesizer - is one of the major outcomes of Bos [2] and is the tool used to develop the ternary circuits in this thesis.

- the LaTeX project [52] is used to create documents, especially in academia. Latex offers precise control over formatting. It is particularly popular because it handles complex mathematical equations and citations seamlessly.
- Overleaf [53] has made Latex more intuitive than expected. Not having to learn everything from scratch has saved time.
- Lucidchart [54] is a flexible way to create the complex diagrams needed in the designs.
- ChatGPT OpenAI [55] has proven itself to be a useful discussion partner for learning Overleaf and Latex, with questions like *"How do you change the normal font size with the fontspec package?"*. It has also helped to check how clear the technical information and the explanations of the sections in the document are. For example, asking questions like ***Explain this:*** *(Document paragraph explaining theory)* demonstrated how understandable sentences are.
- Grammarly [56] was used to overcome the linguistic challenges encountered in this work. Grammar check has helped eliminate many annoying mistakes with is/are, has/have, and split infinitives. Grammarly was also used to suggest synonyms to improve the flow of the text.

3.6 Challenges

The main goal of the thesis is to design and implement a ternary CPU and investigate a superscalar design, with some additional challenges to overcome.

- There is a lack of research on ternary computing, and some existing studies are incomplete or inaccurate.
- The notation for ternary logic and logic for fast adders is not well developed, so expressing ideas is challenging as discussed in Jones [32].
- REBEL-2 is designed to be a teaching tool and has a limited set of instructions for educational purposes. This means that it has a deliberately small instruction set, without some common instructions such as load and store. In addition, it was still under development in parallel with the work performed in this thesis.

—"Two are better than one and three than two."

Afghan Proverb

4

Design and Results

This chapter proposes a comprehensive design for a ternary CPU and examines its critical components. Key areas of focus include the design of the ternary ALU, the adaptation of the CPU for superscalar operations, and the exploration of fast adders for balanced ternary logic. A design for a balanced ternary carry-lookahead adder is introduced; this type of fast adder details the underlying carry-lookahead logic.

The requirements summarised in the bullets below have impacted the design process of a ternary CPU.

- The design must be developed on MRCS [57]
- The design must support the ternary ISA Rebel 2 shown in Figure 2.1
- The balanced ternary carry-lookahead adder is explained in Section 4.1.
- The proposed ALU-* is explained in Section 4.2
- The proposed ternary CPU is explained in Section 4.3
- Scaling the proposed CPU to superscalar is discussed in Section 4.4

All diagrams showing the various circuits and components are screenshots of the circuit design of MRCS, the circuit synthesis tool presented by Bos, Risto and Gundersen [21].

4.1 Design for a carry-lookahead adder for balanced ternary

A carry-lookahead adder is a popular type of fast adder in the binary context Patterson and Hennessy [14]. It is a fast digital circuit for adding numbers. Unlike other adders, it calculates the carry signals for each stage simultaneously rather than one after the other. This parallel computation speeds up addition. This type of adder is excellent for fast calculations, but is more complex. While working on this, it was found that minimal research has been conducted on this somewhat open problem so far within balanced ternary.

4.1.1 Carry-lookahead logic in balanced ternary

For binary lookahead, two types of signals are defined as follows:

- Propagate - A carry is propagated to the output
- Generate - A carry is unconditionally generated

The carry output is essentially a function of propagating and generating signals, as well as the carry input.

For balanced ternary, this can quickly be shown not to be the same. When looking at the truth table for carry output Table 4.1 and Table 4.3, it is clear that for balanced ternary there is no combination of two digits added that unconditionally generates a carry. This leads to another state proposed by Jones [35], maybe generate.

Looking at the Table 4.1, Table 4.2, and Table 4.3, it is apparent that the behaviour is symmetric for positive and negative numbers. The lookahead logic as a product of adding two trits is shown in Table 4.4. By combining two of these and analysing the resulting logic, it is clear that there are combinations that can unconditionally generate a carry. This can also be seen in Table 4.5. *Maybe Generate* (M) will *Generate* (G) if the

B	A		
	-	**0**	+
-	-	-	0
0	-	0	0
+	0	0	0

Table 4.1: Carry out for negative carry in

B	A		
	-	**0**	+
-	-	0	0
0	0	0	0
+	0	0	+

Table 4.2: Carry out for 0 carry in

B	A		
	-	**0**	+
-	0	0	0
0	0	0	+
+	0	+	+

Table 4.3: Carry out for positive carry in

B	A		
	-	**0**	+
-	-M	-P	0
0	-P	0	P
+	0	P	M

Table 4.4: Lookahead information from 1 trit addition

trit addition before cannot *Negative Propagate* (-P), so all 3 states must be represented. This leads to 7 different possibilities for the output carry depending on the input carry represented by the following symbols later in this chapter:

- -G : **Generate negative**, negative carry no matter what
- -M : **Maybe generate negative**, negative carry if input carry is not positive
- -P : **Propagate negative**, negative carry if input carry is negative
- **No carry**
- P : **Propagate positive**, positive carry if input carry is positive
- M : **Maybe generate positive**, positive carry if input carry is not negative
- G : **Generate positive**, positive carry no matter what

B	A						
	$-g_0$	$-m_0$	$-p_0$	0	p_0	m_0	g_0
$-g_1$	-G	-G	-G	-G	-G	-G	-G
$-m_1$	-G	-G	-G	-G	-M	-P	0
$-p_1$	-G	-M	-P	0	0	0	0
0_1	0	0	0	0	0	0	0
p_1	0	0	0	0	P	M	G
m_1	0	P	M	G	G	G	G
g_1	G	G	G	G	G	G	G

Table 4.5: Lookahead information from two sub-signals in a ternary adder, highlighting combinations that cannot result from trit addition.

These states effectively represent a septenary value that gives information about the lookeahead. This is later encoded to ternary values shown in Table 4.7. The resulting lookahead logic when doing 2-trit addition are shown in Table 4.5 The highlighted section

are the values possible from 2 trit addition, and the gray areas are required for a full tree node. This difference between a full tree node, and what can be output from the addition of balanced ternary trits is used to simplify the resulting design in subsection 4.1.5

Jones [35] reaches the same conclusion, and further states that this is reminiscent of threshold logic. This is discussed in Appendix E

As for binary, the septenary states in Table 4.5 describe the carry output as a function of the carry input. This is shown in Table 4.6. An encoding method is required since the focus is on ternary rather than septenary numbers. Jones chose to encode the values as 2 trits; however, encoding them as 3 was more beneficial. This makes it possible to see if the septenary value is equal to, less than, or greater than a specific value with a single comparison without additional decoding. This encoding is shown in Table 4.7. Any combination of g, m, p not in the table shall be considered invalid and is undefined.

Lookahead Signal	Carry **-C**	**0**	**C**
G-	-	-	-
M-	-	-	0
P-	-	0	0
0	0	0	0
P+	0	0	+
M+	0	+	+
G+	+	+	+

Table 4.6: The states for a lookahead circuit, and how it responds to a carry input

Lookahead Signal	Encoding **p**	**m**	**g**
G-	-	-	-
M-	-	-	0
P-	-	0	0
0	0	0	0
P+	+	0	0
M+	+	+	0
G+	+	+	+

Table 4.7: The encoding of 7 states onto 3 trits

4.1.2 Implementation of a full tree node

Since lookahead information can be represented as a septenary value, only $7^2 = 49$ different defined states exists and must be considered for 2-trit addition. This can for the purpose of logic analysis be cut in half again because of symmetry, leading to only 25 states that need to be considered. The remaining values after this reduction are listed in Table 4.8. The carry out information shows that the resulting signal is a representation of how the carry output is affected by the carry input.

This allows the determination of which combinations of inputs are of interest for creating the final expression. The important rows are where the response to a carry input changes

Lookahead 1			Lookahead 0			Carry in, out			Lookahead out
p1	m1	g1	p0	m0	g0	-	0	+	
0	0	0	0	0	0	0	0	0	0
0	0	0	+	0	0	0	0	0	0
0	0	0	+	+	0	0	0	0	0
0	0	0	+	+	+	0	0	0	0
+	0	0	-	-	-	0	0	0	0
+	0	0	-	-	0	0	0	0	0
+	0	0	-	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0
+	0	0	+	0	0	0	0	+	P
+	0	0	+	+	0	0	+	+	M
+	0	0	+	+	+	+	+	+	G
+	+	0	-	-	-	0	0	0	0
+	+	0	-	-	0	0	0	+	P
+	+	0	-	0	0	0	+	+	M
+	+	0	0	0	0	+	+	+	G
+	+	0	+	0	0	+	+	+	G
+	+	0	+	+	0	+	+	+	G
+	+	0	+	+	+	+	+	+	G
+	+	+	-	-	-	+	+	+	G
+	+	+	-	-	0	+	+	+	G
+	+	+	-	0	0	+	+	+	G
+	+	+	0	0	0	+	+	+	G
+	+	+	+	0	0	+	+	+	G
+	+	+	+	+	0	+	+	+	G
+	+	+	+	+	+	+	+	+	G

Table 4.8: All necessary states to evaluate for 2 trit lookahead, and the resulting propagate values for the full node

to a non-zero value and one of the septenary input values overflows. The rows where these happen are highlighted.

This information provides the minimum number of expressions required to generate the lookahead signals for the selected encoding. Written in a non-standard format, Equation (4.1) shows the connection between the encoded input, and the output.

$$\begin{aligned} P = c &\overset{def}{\Longleftrightarrow} ((p_1 = c) \wedge (p_0 = c)) \vee ((m_1 = c) \wedge (g_0 \neq -c)) \vee (g_1 = c) \\ M = c &\overset{def}{\Longleftrightarrow} ((p_1 = c) \wedge (m_0 = c)) \vee ((m_1 = c) \wedge (m_0 \neq -c)) \vee (g_1 = c) \\ G = c &\overset{def}{\Longleftrightarrow} ((p_1 = c) \wedge (g_0 = c)) \vee ((m_1 = c) \wedge (p_0 \neq -c)) \vee (g_1 = c) \\ & c \in \{-1, 1\} \end{aligned} \tag{4.1}$$

This can be written using the notation proposed by Jones [32] as done in Equation (4.2), but this increases the complexity as the input variables need to be reused:

$$\begin{aligned} P &= (p_1 \boxtimes p_0) \boxplus ((m_1 \boxplus g_0) \boxtimes m_1) \boxplus g_1 \\ M &= (p_1 \boxtimes m_0) \boxplus ((m_1 \boxplus m_0) \boxtimes m_1) \boxplus g_1 \\ G &= (p_1 \boxtimes g_0) \boxplus ((m_1 \boxplus p_0) \boxtimes m_1) \boxplus g_1 \end{aligned} \tag{4.2}$$

In Equation (4.1), $\vee$ represents a logical OR operation and $\wedge$ represents a logical AND operation. The results of the two equations will be the same, with the exception that Equation (4.2) will tell you what the values are, while Equation (4.1) will tell you what the value is if it is not 0.

These expressions will cover all the rows necessary to satisfy the encoding specified in Table 4.7, and generate the correct output.

4.1.3 Adding trits to the node

It is possible to express the equations needed to create 3 and 4 trit lookahead units. It is important to note that the number of expressions for balanced ternary grows exponentially, and you have $2^t - 1$ expressions, where t is the number of trits in the look-ahead unit. This means that the complexity and size will rapidly increase as you add more trits to the unit. Of these $2^t - 1$ expressions, $2^{t-1} - 1$ are the signals generated that are common for all expressions. The remaining 2^{t-1} expressions are unique but share a common first part where only the final value is different. By grouping these, it is possible to

reduce the circuit size at the cost of adding some propagation delay. This is explored in subsection 4.1.4

The expressions for 3 trit lookahead are shown in Equations 4.3, 4.4, and 4.5. The expression for 4 trits is longer and is shown in Appendix G.

By creating those expressions, some patterns started to appear:

- The least significant trit is alternating between checking p_0 and g_0 in the case of P, always checking m_0 in the case of M, and alternating between g_0 and p_0 in the case of G.
- The rest of the trits follow the same model by repeating the pattern $p \to m \to g \to m$. These values are repeated depending on the trit position. Each value is repeated only once for the second trit from the right. For the third trit value, each value is repeated twice. For the fourth trit value, each value is repeated 4 times. The amount of repetition doubles every step left towards the most significant trit.
- Whether the carry and signal should have the same sign, or not be the opposite sign follows the bits in the binary representation of the row number from the top
- The common parts of each expression are equal to the *generate* (G) expression for one less trit

This pattern can be represented as a formula that generates the expressions added for each trit, which is what Figure 4.2 does. This expression generates the signal for row n, in trit a. The outputs will be in the range $\{-1,0,1\}$, which is mapped to $\{g,m,p\}$. It does not include the common parts for the previous g expressions, which must be added to the final expression.

After identifying which signal to use, it can be determined how it should be checked by using the binary expansion of n and the bit position corresponding to the trit position a. This is done through Equation (4.12). This will give all expressions for evaluating P for any number of trits. To obtain the expression for the values M and G, the least significant trit signal must be swapped to the corresponding signal for that type of expression, as described earlier in this section. For example, if you wanted to find the expression corresponding to row 4 for a 3 trit number, you need to calculate $C_2(3)$,$C_1(3)$,$C_0(3)$, $S_2(3)$, $S_1(3)$, $S_0(3)$. This will give the signals 0, 0, -1, which is mapped to m m g. It will give the values 0, 1, 1 for the binary expansion, so the first trit must match the sign of a c value, and the second and third trit must not be the opposite of a c value. This gives the expression $(m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)$, which is the fourth line for P shown in Equation (4.3). Once you have unique expressions for a signal, you need to add the

[ht!]

$$
\begin{aligned}
P = c \overset{def}{\Longleftrightarrow} &((p_2 = c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((p_2 = c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((m_2 = c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((p_2 = c) \wedge (g_1 = c)) \vee \\
&((m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&(g_2 = c)
\end{aligned} \tag{4.3}
$$

$$
\begin{aligned}
M = c \overset{def}{\Longleftrightarrow} &((p_2 = c) \wedge (p_1 = c) \wedge (m_0 = c)) \vee \\
&((p_2 = c) \wedge (m_1 = c) \wedge (m_0 \neq -c)) \vee \\
&((m_2 = c) \wedge (g_1 \neq -c) \wedge (m_0 = c)) \vee \\
&((m_2 = c) \wedge (m_1 \neq -c) \wedge (m_0 \neq -c)) \vee \\
&((p_2 = c) \wedge (g_1 = c)) \vee \\
&((m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&(g_2 = c)
\end{aligned} \tag{4.4}
$$

$$
\begin{aligned}
G = c \overset{def}{\Longleftrightarrow} &((p_2 = c) \wedge (p_1 = c) \wedge (g_0 = c)) \vee \\
&((p_2 = c) \wedge (m_1 = c) \wedge (p_0 \neq -c)) \vee \\
&((m_2 = c) \wedge (g_1 \neq -c) \wedge (g_0 = c)) \vee \\
&((m_2 = c) \wedge (m_1 \neq -c) \wedge (p_0 \neq -c)) \vee \\
&((p_2 = c) \wedge (g_1 = c)) \vee \\
&((m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&(g_2 = c)
\end{aligned} \tag{4.5}
$$

Figure 4.1: Lookahead expressions for 3 trits

$$f : \{-1,0,1\} \to \{g,m,p\} \tag{4.6}$$

where

$$f(x) = \begin{cases} g & \text{if } x = -1, \\ m & \text{if } x = 0, \\ p & \text{if } x = 1, \end{cases} \tag{4.7}$$

$$C_a(n) = \cos(\frac{\pi}{2} \cdot \lfloor \frac{n}{2^{a-1}} \rfloor) \tag{4.8}$$

$$n \in \mathbb{Z}, \quad 0 \leq n < 2^{t-1} \tag{4.9}$$

$$a \in \mathbb{Z}, \quad 0 \leq a < t \tag{4.10}$$

$$f(C_a(n)) \text{ maps } C_a \text{ to } \{g,m,p\} \tag{4.11}$$

Figure 4.2: Function to generate expressions for lookahead

$$S_a(n) = \lfloor \frac{n}{2^a} \rfloor \mod 2 \tag{4.12}$$

$$
\begin{aligned}
P = c \overset{def}{\iff} & ((p_2 = c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
& ((p_2 = c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
& ((m_2 = c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
& ((m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
& ((p_2 = c) \wedge (g_1 = c)) \vee \\
& ((m_2 = c) \wedge (p_1 \neq -c)) \vee \\
& (g_2 = c)
\end{aligned}
\tag{4.13}
$$

Figure 4.3: Breakdown of the pattern

expressions for G with one less trit, shifted once to the left. This is repeated down to 1 trit.

4.1.4 Implementing the design

The expressions in the previous section have been implemented and tested in MRCS. Similar logic in the expressions could be grouped together to increase simplicity in design, but it may increase the propagation delay. As a continuation of the earlier observation about the pattern in subsection 4.1.3, all the output signals P, M, and G are mostly the same except for the final part of the expression which changes between the three signals. This can be seen in Equation (4.13), where the common generate is coloured blue, the common parts of the expressions are coloured red, and the unique parts are coloured green.

The change pattern is the same for all expressions, which means that the same logical circuit could be used to implement it.

This means three different types of standard cells are enough to find the output carry:

- The common expression for all signals, coloured red
- The unique expression for all signals, coloured green
- The common generate for all signals, coloured blue

For 3 trits, this means that Equation (4.14) can be implemented in logic to determine the carry anticipation.

$$
\begin{aligned}
common_p = c &\overset{def}{\iff} ((p_2 = c) \wedge (p_1 = c) \vee (m_2 = c) \wedge (g_1 \neq -c)) \\
common_g = c &\overset{def}{\iff} ((p_2 = c) \wedge (m_1 = c) \vee (m_2 = c) \wedge (m_1 \neq -c)) \\
partial_g = c &\overset{def}{\iff} ((p_2 = c) \wedge (g_1 = c)) \vee ((m_2 = c) \wedge (p_1 \neq -c)) \vee (g2 = c) \\
unique_P = c &\overset{def}{\iff} (common_p \wedge (p_0 = c)) \vee (common_g \wedge (g_0 \neq -c)) \\
unique_M = c &\overset{def}{\iff} (common_p \wedge (m_0 = c)) \vee (common_g \wedge (m_0 \neq -c)) \\
unique_G = c &\overset{def}{\iff} (common_p \wedge (g_0 = c)) \vee (common_g \wedge (p_0 \neq -c))
\end{aligned}
\tag{4.14}
$$

Figure 4.4: Simplified expression grouping the common parts

These are simpler to implement since they reuse components at the cost of a larger propagation delay. These simplified expressions results in the circuits shown in Figures 4.5, 4.6, 4.7, 4.8

If logic gates allow an arbitrary number of signals, the expressions can instead be implemented in. Figure 4.1 directly in only two gate depths.

4.1.5 Simplifying the first lookahead nodes

The first lookahead nodes do not need to account for the *generate signal*, as two trits added together cannot *generate* as seen in Table 4.4. This enables the simplification of the expressions to Equations 4.15, 4.16, and 4.17 by assuming that g is always 0

This means that the initial layer in the carry-lookahead tree can use significantly simplified expressions to achieve the same output. This only applies to balanced ternary and is not true for a higher radix.

$$\begin{aligned}(P=c) \overset{def}{\iff} ((p_2=c)\wedge(p_1=c)\wedge(p_0=c))\vee\\ ((p_2=c)\wedge(m_1=c)\vee\\ ((m_2=c)\wedge(p_0=c))\vee\\ ((m_2=c)\wedge(m_1\neq -c)\vee\\ (((m_2=c)\wedge(p_1\neq -c))\end{aligned} \tag{4.15}$$

$$\begin{aligned}(M=c) \overset{def}{\iff} ((p_2=c)\wedge(p_1=c)\wedge(m_0=c))\vee\\ ((p_2=c)\wedge(m_1=c)\wedge(m_0\neq -c))\vee\\ ((m_2=c)\wedge(m_0=c))\vee\\ ((m_2=c)\wedge(m_1\neq -c)\wedge(m_0\neq -c))\vee\\ ((m_2=c)\wedge(p_1\neq -c))\end{aligned} \tag{4.16}$$

$$\begin{aligned}(G=c) \overset{def}{\iff} ((p_2=c)\wedge(m_1=c)\wedge(p_0\neq -c))\vee\\ ((m_2=c)\wedge(m_1\neq -c)\wedge(p_0\neq -c))\vee\\ ((m_2=c)\wedge(p_1\neq -c))\end{aligned} \tag{4.17}$$

4.1.6 Carry-lookahead in a higher radix

When creating a carry-lookahead adder in a higher radix may seem like a daunting task, but the logic for calculating the carry will not change at all. This applies to balanced and unbalanced lookahead trees, as long as only two digits are added together. This can be seen when looking at the maximum value by adding two digits together. Given radix r, the highest digit that can be represented equals $r-1$. Adding two max value digits together will result in a value of $2(r-1) = 2r-2$. The value needed to achieve a carry that is greater than 1 is equal to $2r$. This holds for any radix. For an unbalanced number system, these values are instead $2(\lfloor \frac{r}{2} \rfloor) = 2(\frac{r-1}{2}) = r-1 < \frac{r-1}{2} + r$. This is explored more in Section 2.5.

The effect of this is that the carry value can never exceed 0. This means that the carry-lookahead tree determines how the carry output changes based on the carry input. This means that all unbalanced lookahead tree nodes will be identical and all balanced lookahead tree nodes will be identical. For any balanced addition higher than radix 3 the optimisation in subsection 4.1.5 is no longer possible. Unlike balanced ternary, adding two digits in a higher radix can unconditionally generate a carry. Table 4.9 shows the

	-2	-1	0	1	2
-2	-g	-m	-p	0	0
-1	-m	-p	0	0	0
0	-p	0	0	0	p
1	0	0	0	p	m
2	0	0	p	m	g

Table 4.9: Carry information when adding two balanced quinary numbers

resulting carry information when adding two quinary numbers. For quinary and higher, all signals must always be included.

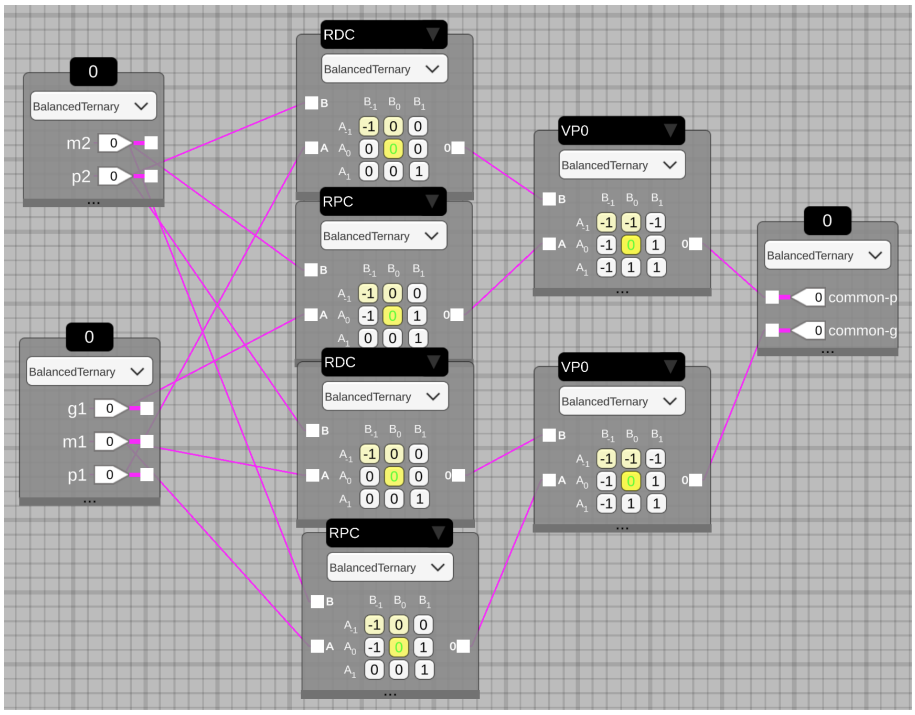


Figure 4.5: The logic common to all carry information signals

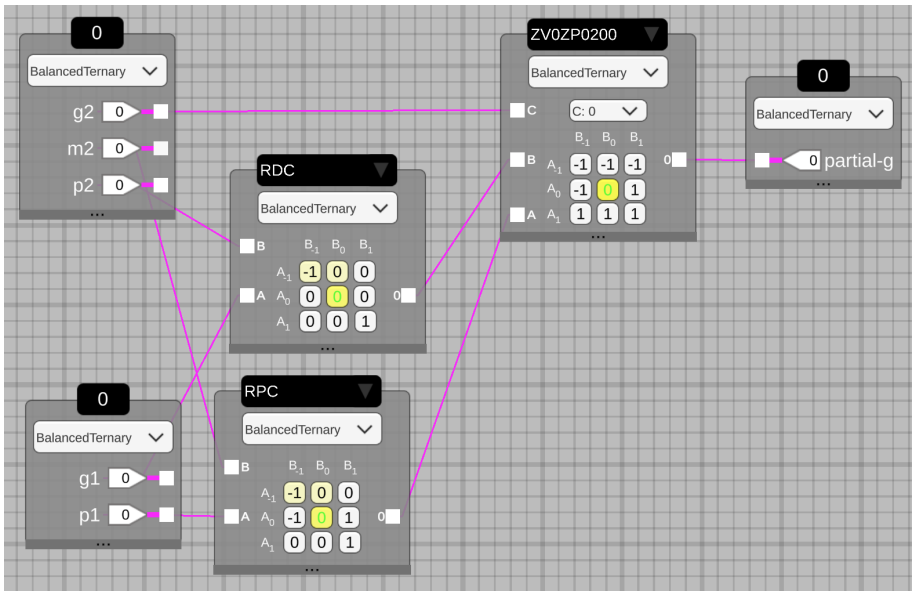


Figure 4.6: The logic to determine if a carry is generated, common to all signals

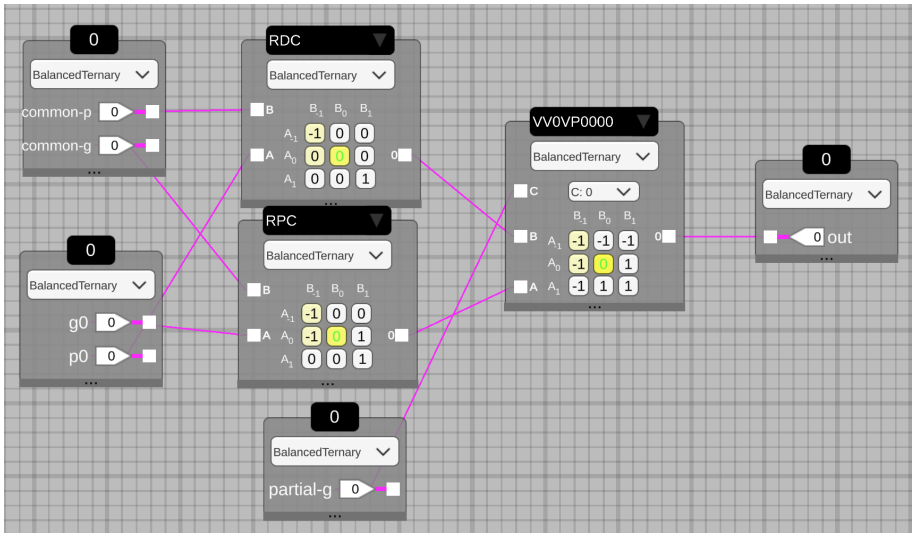


Figure 4.7: The circuit that combines the partial generate, the common signals, and the signals from least significant trit

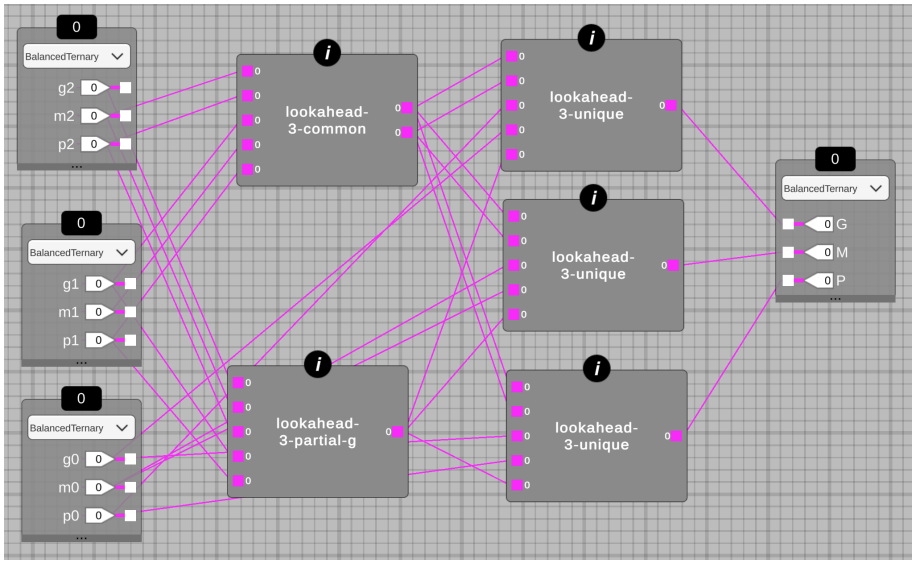


Figure 4.8: Combining everything to create the G, M, P signals that encode the carry information

4.2 Design of ALU-*, a ternary ALU supporting REBEL-2

An ALU is a central part of any processor, responsible for all arithmetic and mathematical operations. This section describes ALU-*, the ternary ALU designed for this thesis, and the components it is built from.

ALU-* is designed as part of this thesis using MRCS [58], Bos' open-source tool to design binary and ternary circuits. ALU-* implements the REBEL-2 instruction set defined in Bos' dissertation as seen in Figure 2.1. The ALU-* design is open source and is available on Github [59].

The design of ALU-* contains almost everything needed to support the functions described in the REBEL 2 ISA. The design is depicted in Figure 4.10 and consists of two sections: **Logical Components** and **Signal Selection**. The logical components consist of:

- Adder - Supports the addition operation - ADD, ADDi
- Multiplier - Supports the multiplication operation - MUDI
- Wordwise comparator - Supports the wordwise mode of COMP, as well as MIMA (min/max)
- Tritwise comparator - Supports the tritwise mode of COMP
- Shifter - Supports all the shift operations, SHi

func2	func1	func0	Function description
-	+	X	Compare Word
-	0	X	Compare Trit, min/max
0	0	X	Add/subtract
0	+	X	Multiply
0	-	X	Divide
+	X	X	Shift operation

Table 4.10: Function signals for ALU-*

Division is not included. Without more time, a simple implementation of division in the ALU would not contribute significantly beyond a logic table for the 2x2 inputs to the 2 output trits. Instead, the time was focused on higher-level CPU design and specific implementations of adders in balanced ternary.

Signal selection consists of multiplexers that play a vital role in determining the type of instruction to be executed by ALU-*. The input signals func0, func1 and func2 select which result to output, and modify the calculation. The result to output is defined in Table 4.10 [1], and how they modify the calculation is specified in the respective tables. Despite not being implemented in the ALU, divide has been assigned a code for later implementations that may include it.

All designs defined in MRCS consist of **four** types of building blocks shown in Figure 4.9. **Input** and **output** can be set to ternary or binary and do not have logic or truth tables. They only function as an interface to the circuit design. A **logic gate** is a building block that accepts between 1 and 3 inputs and output data as set in the truth table.

A binary logic gate with two inputs can create $2^{2 \cdot 2} = 16$ unique logic gates. A ternary logic gate with two inputs gives $3^{3 \cdot 3} = 19683$ possible unique logic gates. A **component** can consist of inputs, outputs, logic gates, and other components. It is possible to create the desired combinational circuits using these four elements. The proposed design for ALU-* has been implemented and tested in MRCS and is part of Tiny Tapeout 5 [59]. This means that the circuit will run on a chip using binary-coded ternary.

4.2.1 Adder

The adder is a ternary implementation of a ripple carry adder. It consists of a ternary half adder followed by a chain of ternary full adders, the difference being that a full adder

[1] In this and following tables **X** designates a *DON'T CARE* state

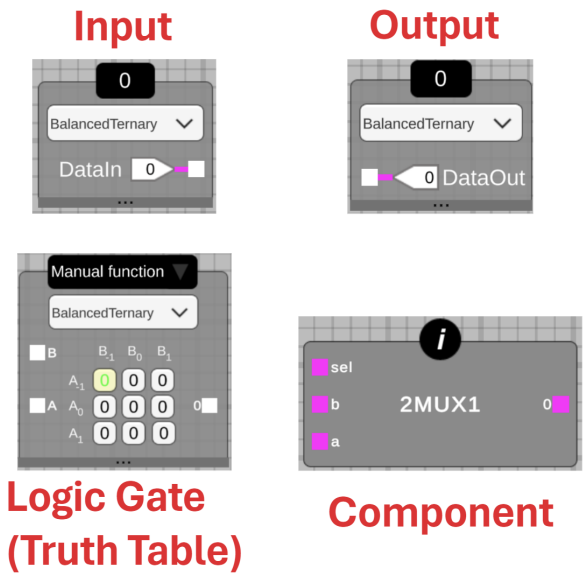


Figure 4.9: The different types of components present in MRCS

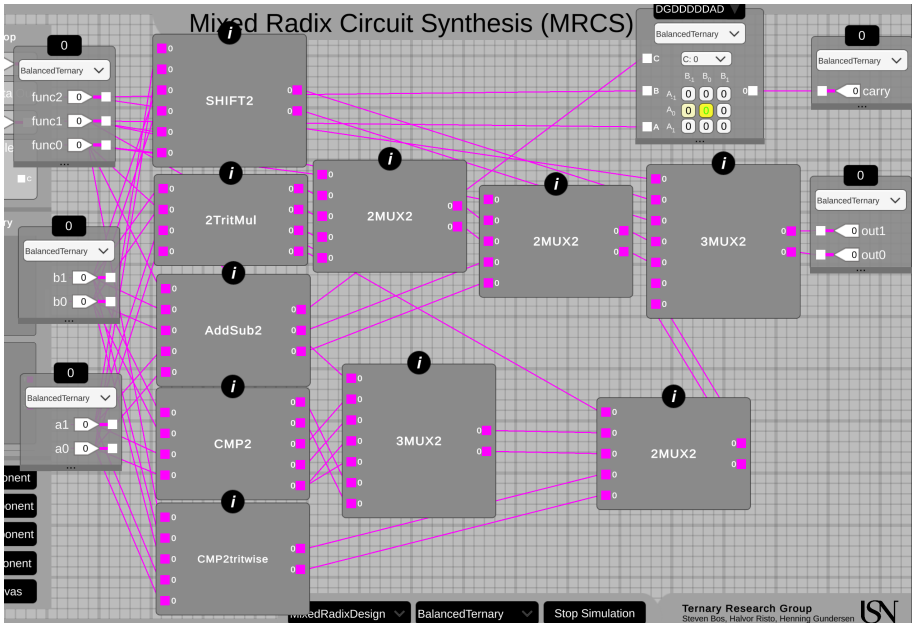


Figure 4.10: Complete design for ALU-*

can also accept the carry from the previous adder. The downside becomes apparent with more trits, as the propagation delay will increase. With only two trits, this is negligible. With only two trit numbers used, only a single half adder and full adder are needed as seen in Figure 4.12. The same downsides and problems exist in binary implementations, and the ripple-carry adder implementation on a high level is the same for ternary and binary.

The designed adder has a mode input to switch from subtraction mode to subtraction. This will cause the second inverted value to be added to the first, as shown in Figure 4.11. A multiplexer can select the inverted signal.

The func0 signal determines whether the adder should subtract or add the two numbers Table 4.11

func2	**func1**	**func0**	**Function description**
0	0	0	Add
0	0	+	Subtract

Table 4.11: Addition table

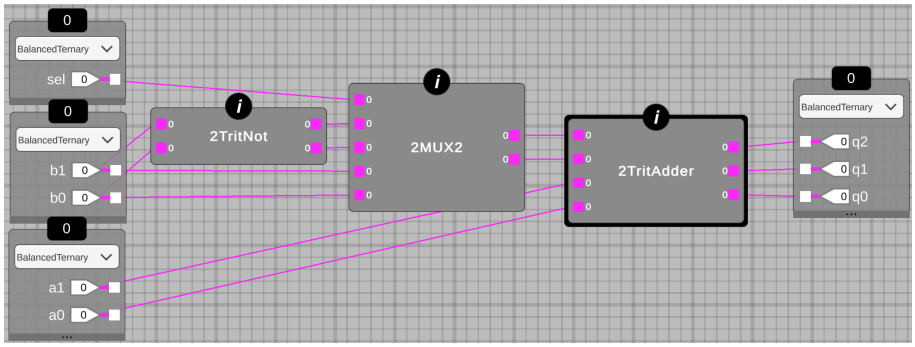


Figure 4.11: Addition and subtraction

4.2.2 Multiplier

A hardware implementation of lattice multiplication performs multiplication. Since the result of two trits multiplied in a balanced ternary can always be represented with a single trit, there is no carry from the multiplication itself. For a 2 trit multiplier, this in the multiplications and additions shown in Figure 4.13.

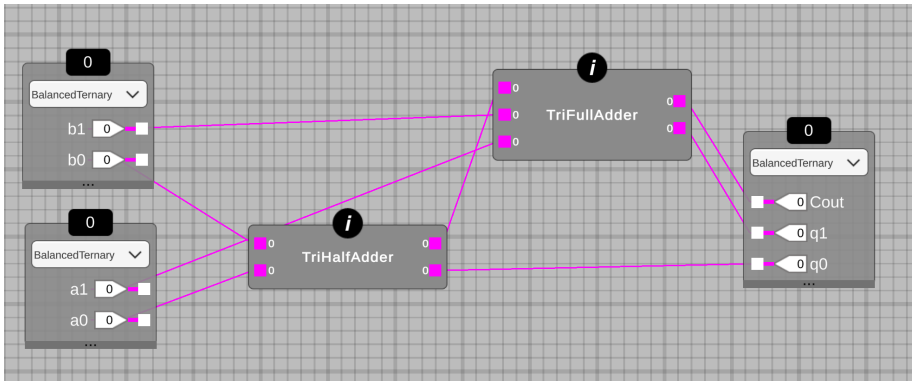


Figure 4.12: MRCS implemetation of 2 trit adder

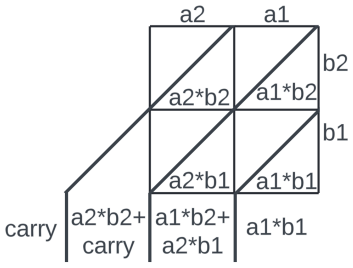


Figure 4.13: 2 trit lattice multiplication

Each trit is multiplied by every other trit. The sums are added together following the diagonals in the lattice multiplication. This is shown in Figure 4.14

When multiplying two numbers together, you may need up to twice as many trits to represent the result. Although this is not handled in the REBEL-2 ISA, the option to obtain the upper and lower half of the multiplication in ALU-* is still added. The selected output portion is determined by func0, as shown in Table 4.12.

For a larger number of trits, this design may be less useful given how the numbers are added together. Using a similar approach, a ternary implementation of a Wallace multiplier, would be more appropriate, as shown in Yoon, Baek, Kim *et al.* [36].

func2	**func1**	**func0**	**Function description**
0	+	0	Multiply lower
0	+	+	Multiply upper

Table 4.12: Multiply functions

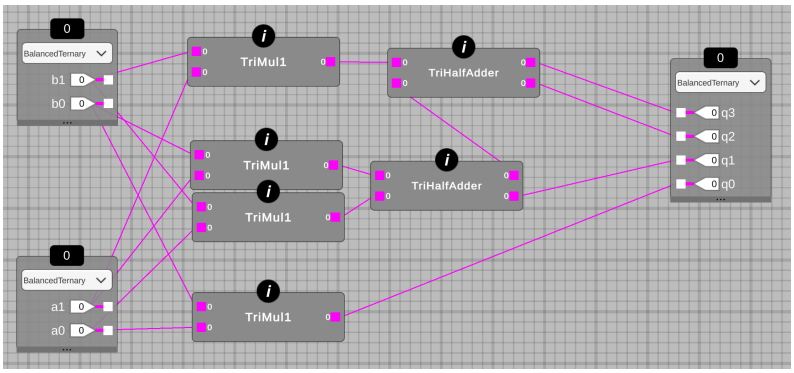


Figure 4.14: MRCS implementation of 2 trit multiplier

4.2.3 Compare

ALU-* has two modes of operation for comparison: **Tritwise** or **full word**. These are implemented as two separate components within the design as the functions differ. The two components must be able to compare and select the min and max values. For tritwise comparison, all trits can be done independently. This can be implemented with a simple logic gate. The truth table for this is shown in Table 4.13, and the MRCS implementation is shown in Figure 4.15

B	A		
	-	**0**	+
-	0	+	+
0	-	0	+
+	-	-	0

Table 4.13: Comparison between two inputs A and B. + if A is greater, - if B is greater

LST	MST		
	-	**0**	+
-	-	-	+
0	-	0	+
+	-	+	+

Table 4.14: Combining groups of trit comparison to output result. MST is most significant trit, LST is least significant trit

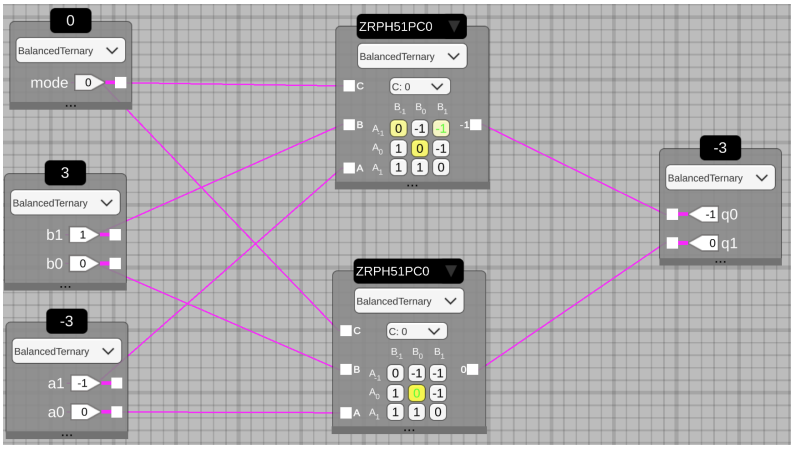


Figure 4.15: Tritwise comparison/min/max

Comparing full words is more complicated, since more significant trit positions take priority for comparison. The implemented design accepts two trits, and an override signal as shown in Figure 4.18. This is not optimal, leading to an implementation with a high propagation delay as the comparison ripples through just like in a ripple-carry adder. This can be seen in Figure 4.16. An alternative implementation could be to create a component that can check comparisons instead, implementing the truth table shown in Table 4.14. This would have allowed for parallel comparisons, and grouping results. The resulting design will be similar to Figure 4.17

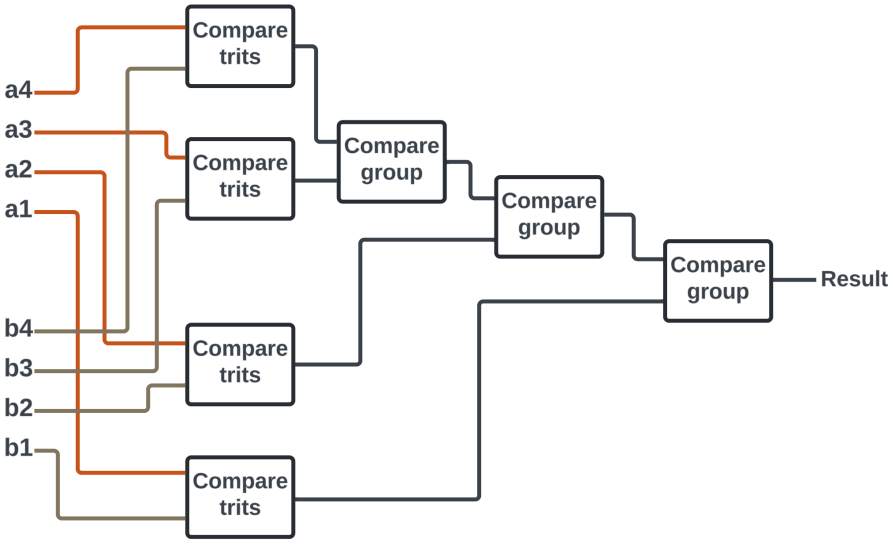


Figure 4.16: Ripple comparison

The comparison result determines the input for full word max and min values. This is done using the comparison result as the select value for multiplexers, which effectively sorts the two values as seen in Figure 4.19.

func1 determines whether the operation should be for the whole word or individual trits. *func0* determines whether the min, compare, or max operation should be performed. This can be seen in Table 4.15.

4.2.4 Trit shift

The ALU-* shift block can shift a two-trit value in either direction. This works similarly to a barrel shifter. A multiplexer selects the value to be output. This is a0, a1, or insert. The direction to shift is decided by dir, and the amount to shift is decided by imm0 and

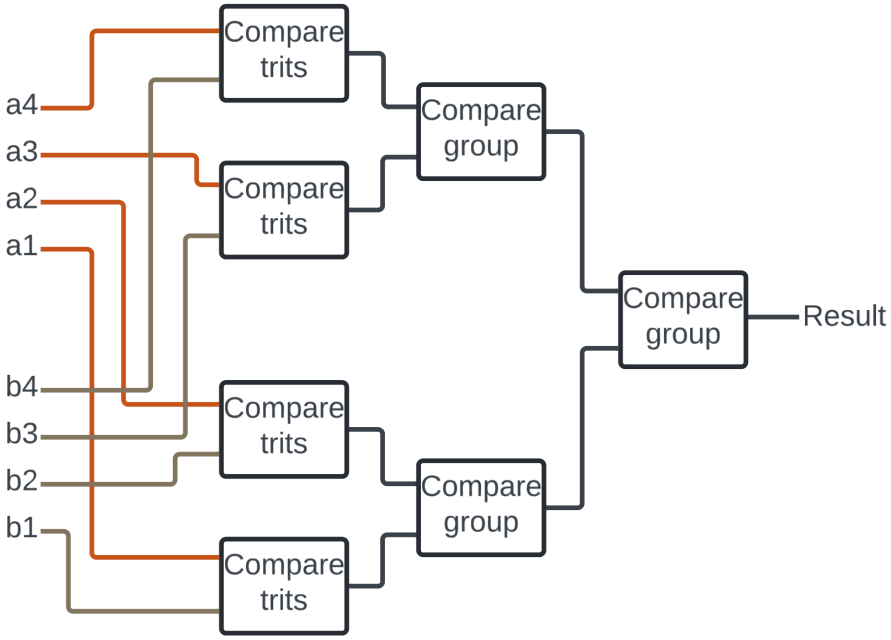


Figure 4.17: Parallel comparison

func2	func1	func0	Function description
-	0	-	Tritwise min
-	0	0	Tritwise comparison
-	0	+	Tritwise max
-	+	-	Wordwise min
-	+	0	Wordwise comparison
-	+	+	Wordwise max

Table 4.15: Compare functions

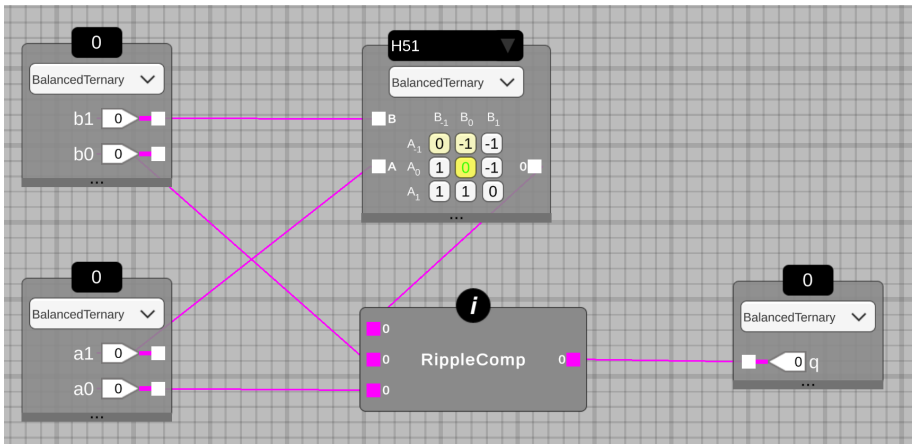


Figure 4.18: MRCS implementation of 2 trits wordwise comparator

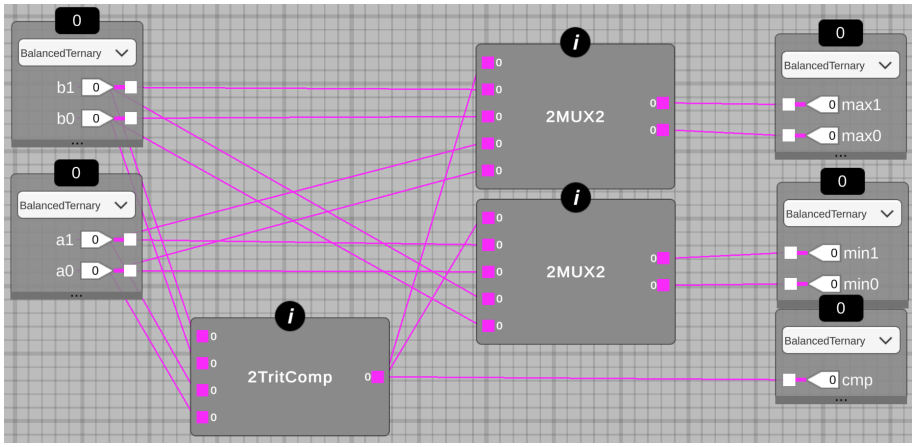


Figure 4.19: Wordwise min/max/comparison

imm1 [2]. If the rotary shift is not selected, that is, $dir \neq 0$, then the insert value will be shifted instead of rotating the output. A description of control signals and modes is shown in Table 4.16, and the implementation in MRCS is shown in Figure 4.20.

Func2	Func1 (dir)	Func0 (insert)	Description
+	-	A	Shift left, insert A
+	0	X	Rotary shift right, the shifted value is inserted at the start of the value
+	+	A	Shift right, insert A

Table 4.16: Shift functions and the required control signals

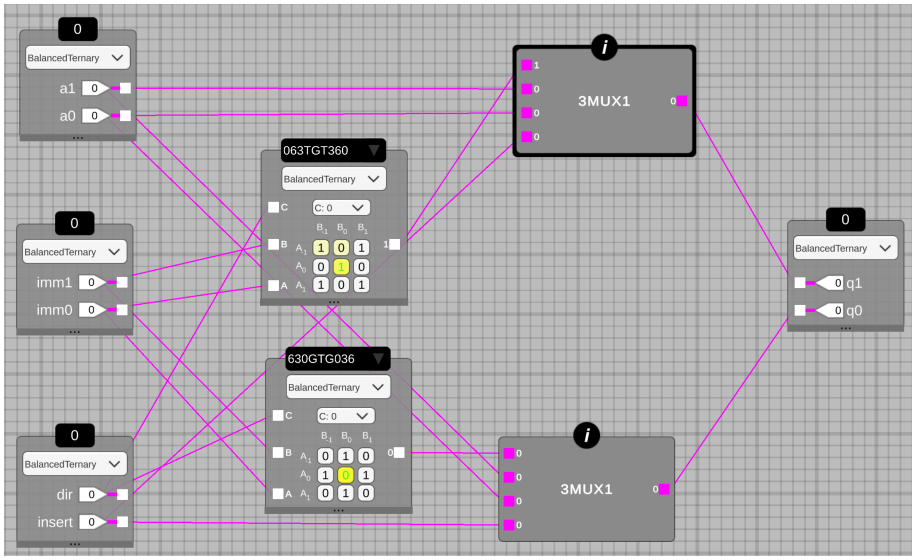


Figure 4.20: MRCS implementation of 2 trit shifter

4.3 Design of a ternary CPU

This section compiles the design choices that reflect the latest version, ternary CPUv3, which has evolved through the three stages discussed in Appendix C.

This latest design consists of several parts handled separately in the sections shown in Figure 4.21.

[2]imm0 and imm1 are parameters defined in REBEL-2

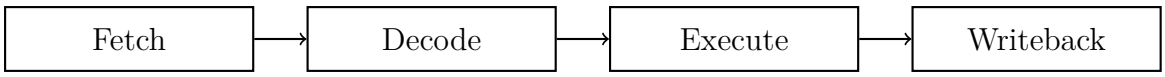


Figure 4.21: The 4 stages in this CPU design

4.3.1 Pipeline

The CPU design is pipelined and helps to separate the functionality into logical blocks. The stages are based on the classic RISC pipeline described in Chapter 4 of Patterson and Hennessy [14] but without the memory stage, as REBEL-2 has not yet implemented memory access instructions. The stages are structured as in Figure 4.21

Instruction pipelining, a vital feature of the CPU design, offers several benefits. It allows for higher frequency operation, as the slowest stage determines the frequency. Moreover, it is instrumental in supporting instruction-level parallelism, a crucial aspect of modern computing. This is important as it can be a cheap way to increase the throughput of a processor without duplicating all the hardware. It can partially hide the latency from fetching data from slower memory modules. Figure 4.22 shows the entire architecture with the stages needed to implement REBEL-2, but the control signals have been omitted for readability.

The CPU is designed to use both edges of the clock cycle. On the rising edge, the data are latched to the next stage in the pipeline. On the falling edge, data are written to the registers and buffers within the stage.

4.3.2 Fetch stage

The fetch stage is the first step in the pipeline and is responsible for obtaining instructions. Figure 4.23 shows this stage in more detail. A specific design decision was to use a data latch and an increment circuit to increment the program counter every cycle. This is done instead of using a ternary counter with a parallel load (Fegri [1]) to achieve the same functionality because the components are simpler to design and test. The data stored in the data latch is either loaded from the end of a pipeline, which could be from a jump or branch instruction, or the previous program counter value plus a constant. The program counter is sent to the instruction register, which fetches the next instruction. The program counter is also forwarded to the decode stage as some instructions use it.

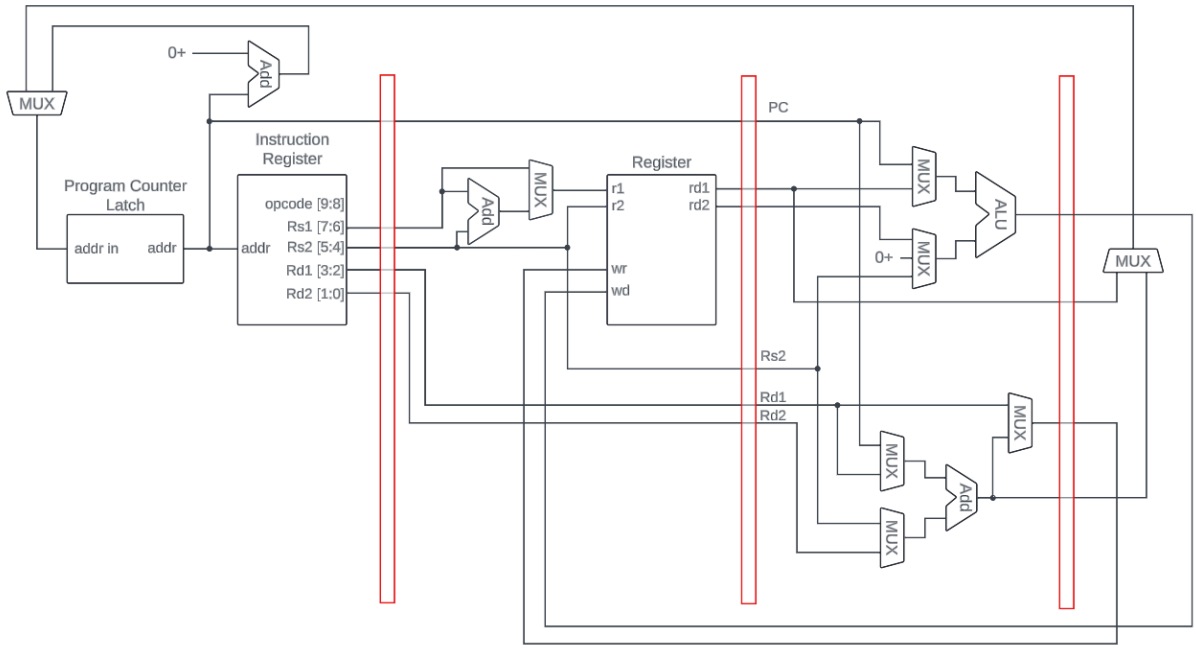


Figure 4.22: Pipelined REBEL 2 CPU architecture (CPUv3)

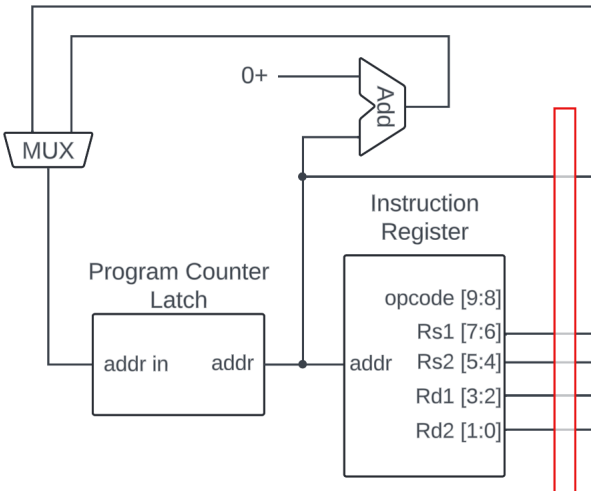


Figure 4.23: Instruction fetch stage

4.3.3 Decode stage

In the decode stage, the data is fetched from the register file and prepared for later execution. Most of the control logic for later stages could be determined at this point in execution and forwarded to later stages. This stage is shown in Figure 4.24. The version depicted does not fully support the REBEL-2 ISA. Two instructions cannot be executed in a single instruction with the register file depicted. This is because the ADDi2 instruction writes to two addresses in a single instruction, and the BCEG instruction reads four values simultaneously. The architecture expanded to include this support in subsection C.1.4. The adder and multiplexer depicted are required for the PCO instruction in the case RD2 is 00. The CPU is instructed to jump to R[rs1 + imm] The values are read from the registers and forwarded to the execute stage in the pipeline.

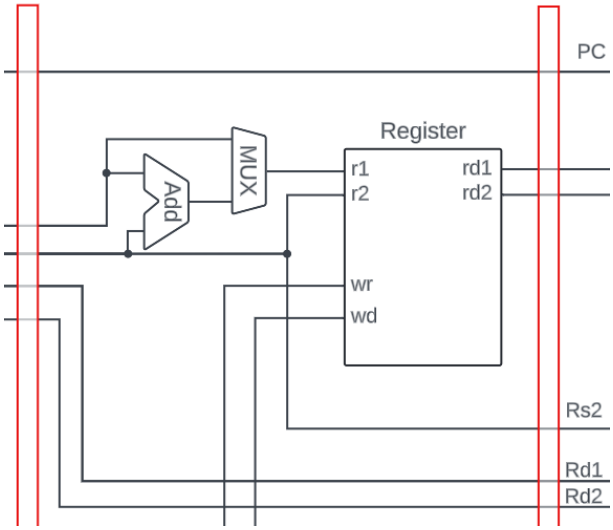


Figure 4.24: Decode stage

Control signals are generated in the decode stage of the CPU. These are not depicted in the diagrams, but connect to every multiplexer and the ALU. This is needed to select the correct signals for the multiplexers, and to use the right operation in ALU-*. Table 4.17 shows how the ISA opcode and RD values relate to the control signals for ALU-* as defined in Table 4.10. The instructions AddI2 and Div have no associated ALU-* instruction as adding 2x2 numbers is not supported, and division is not implemented in ALU-*. BCEG is not implemented as it requires reading 4 values from the register, when the current register only supports up to two at a time.

Instruction	$Opcode_2$	$Opcode_1$	$RD2_2$	$RD2_1$	func2	func1	func0
Add	-	-	0	0	0	0	0
Sub	-	-	X	X	0	0	+
Addi	-	0	X	X	0	0	0
Addi2	-	+	Z	Z	X	X	X
Mul	0	-	0	0	0	+	0
Div	0	-	X	X	0	-	0
Word min	0	0	-	-	-	+	-
Trit min	0	0	-	0	-	0	-
Word max	0	0	+	-	-	+	+
Trit max	0	0	+	0	-	0	+
Shi	0	+	d	t	+	d	t
Comp word	+	-	0	0	-	+	0
Comp	+	-	X	X	-	0	0
BCEG	+	0	X	X	-	+	0
PCO	+	+	X	X	0	0	0

Table 4.17: Instruction types and corresponding ALU-* instruction

4.3.4 Execute stage

The execute stage is where the actual execution of the instruction takes place. ALU-* executes the specified instruction, and the eventual destination address for a jump and write is calculated. The execute stage is shown in Figure 4.25. It consists of two parts: the top part is the ALU-* execution, and the bottom is where the destination address and jump destination are calculated. The control signal sets the multiplexers to perform the right calculation, and the ALU instruction signals must be set according to Table 4.17. The result of the ALU execution and any other necessary signals are forwarded to the execute stage.

4.3.5 Writeback stage

The REBEL-2 ISA does not include any instructions for memory access, and all data is stored to the register file.

The write-back stage is responsible for writing the data back to the register using the result of the execution stage and passing the jump destination to the program counter. This is depicted in Figure 4.22 with the signals travelling through the stage buffers to the

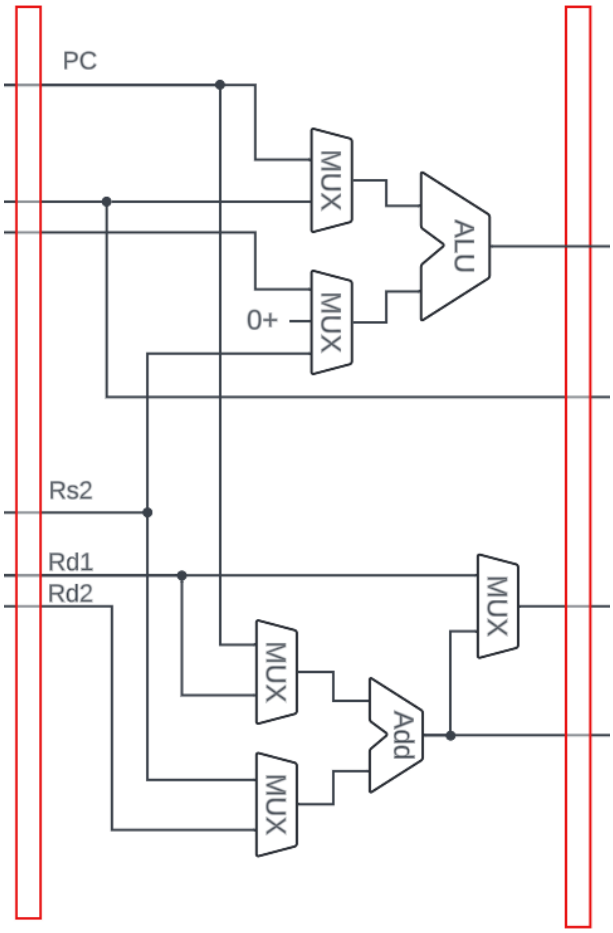


Figure 4.25: Execute stage

register and multiplexer. Except for ADDi2, the result written in the register is always the output of ALU-*. The jump destination could be either the output from the register file directly or could have been calculated in the previous stage. If a jump instruction is executed, the jump destination will be loaded into the program counter, and the previous address will be saved in the register as in the case of the PCO command.

4.4 Scaling up to superscalar

When transitioning from a traditional pipelined CPU to a superscalar CPU, the ternary design does not pose any additional barriers compared to the binary design, and architectural concepts similar to those in binary can be used.

The implementation of ternary logic operates at a higher level of data flow and information management compared to low-level logic. As a result, the impact of ternary logic on design at this level is limited, and the concepts and structures used for binary logic can be reused. Ternary logic could be an improvement in this aspect by increasing the information density.

The idea behind superscalar CPUs is that not all instructions depend on each other, so some instructions can be run out of order. The CPU output must be deterministic and the instructions must be completed in order. Internally, however, the instructions are not required to run in order, and any order is fine as long as the required data is available. Executing instructions out of order now enables the computation of multiple results simultaneously. To achieve this, several steps must be taken.

- The CPU needs to load two or more instructions per clock cycle
- The CPU needs to be aware of data dependencies, and ensure that all instructions use the right data
- The CPU needs multiple functional units to be able to execute instructions in parallel
- The CPU needs to reorder the completed instructions, so the resulting system state is indistinguishable from if every instruction was executed in order

Tomasulo's algorithm

Tomasulo's algorithm [30] improves the performance of superscalar processors by allowing instructions to be executed out of order based on the availability of operands and execution units. The algoritm enables instructions to execute independently and out of order

based on data availability, reducing stalls caused by data dependencies. This allows for a more efficient use of hardware resources and achieves higher levels of instruction-level parallelism.

Reservation Stations - Different types of data dependencies can occur between instructions. These can be classified as Read After Write (RAW), Write After Read (WAW), and Write After Write (WAR). The dependencies can affect the order in which instructions are executed and must be carefully managed to ensure correct program behaviour. In Tomasulo's algorithm, reservation stations avoid waiting time for instructions by buffering them until their operands are available. These stations are linked to each functional unit and independently select the required results.

Common Data Bus - All reservation stations are connected to the Common Data Bus (CDB). When an instruction completes execution, the result is broadcast on the CDB, which must be wide enough to carry all the results.

Register Renaming - Logical registers are specified in the program code, while physical registers are the hardware registers available on the CPU. Register renaming separates the two, allowing the CPU to use a larger pool of physical registers to handle more instructions simultaneously and efficiently.

Dynamic Instruction Scheduling - Instructions can be executed as soon as their operands are available, rather than strictly following their original program order.

Speculative Execution - Tomasulo's algorithm supports speculative execution, allowing the processor to execute instructions ahead of a conditional branch without knowing whether the branch will be taken. If the branch prediction is incorrect, the processor discards the results of the speculatively executed instructions. Tomasulo's algorithm is one potential way of designing an out-of-order CPU with in-order completion. Figure 4.26 shows a rough design to implement such a design. The key elements of the design are the renaming stage, the reservation station, and the reorder buffers. When the instructions are loaded, the order they arrived must be saved in the reorder buffer. A destination register must be assigned, and the registers from which the instruction reads must be mapped from the architectural register (defined in the ISA) to the physical register (the registers available on the chip). These instructions are distributed to reservation stations that keep track of which instructions are in a queue and which are ready to run. Once an instruction has all the required data, the reservation station can process it in the functional unit, and the result is transmitted on the result bus. The result is stored in the register from the result bus, but it is also stored by any instruction in the reservation stations that await that result. When the reorder buffer has received the results, it will commit them to the store buffer.

The blue boxes in Figure 4.26 indicate the main pipeline. The register is not a central part of the pipeline, but is used to store and fetch data when available. The red boxes indicate the functional units, which could be an ALU or a component to access memory, and the red arrows indicate the result bus. Data should be accessed from the result bus to reduce register access.

The number of logical registers does not need to match and will preferably exceed the number of architectural registers.

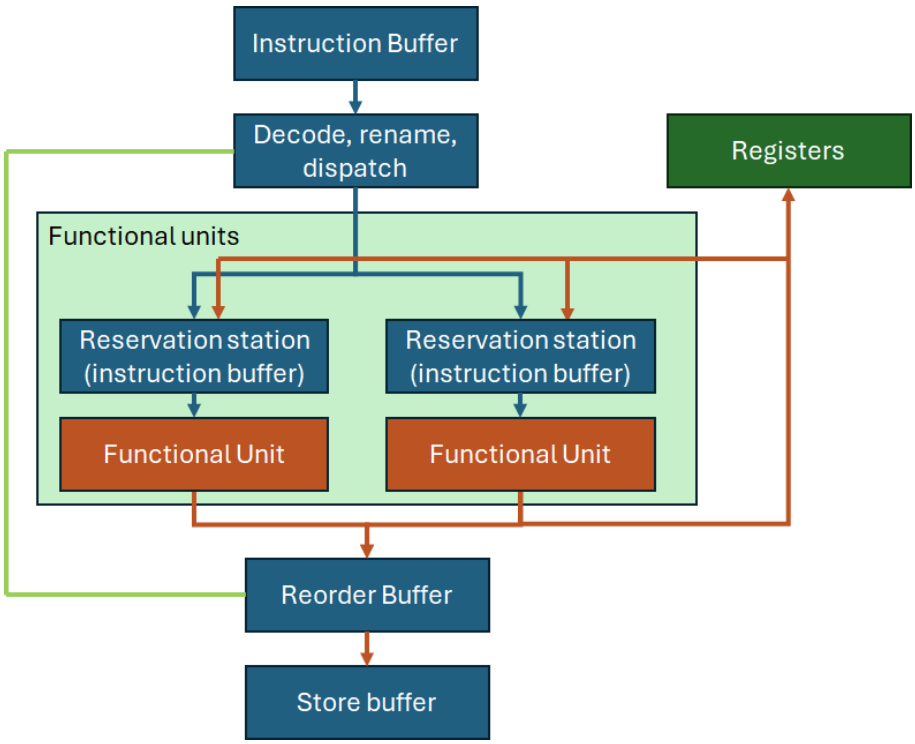


Figure 4.26: Simple design of a superscalar CPU.

4.5 Summary

This chapter presents the culmination of significant technical work in this thesis, with a specific focus on the completed design, algorithmic and architectural work. This work is of paramount importance as it forms the basis of the thesis and its findings.

The necessary expressions were identified to create a fully balanced carry-lookahead node in Section 4.1. This section demonstrates that these expressions are the same for any radix, with the only difference being whether the lookahead is balanced or unbalanced.

Additionally, a method for simplifying the initial nodes in a balanced look-ahead was suggested.

ALU-* was presented in Section 4.2. This is a functional ALU based on a ternary ISA called REBEL-2 shown in Figure 2.1, defined by Bos in his PhD dissertation [2],

A ternary CPU was developed as described in Section 4.3. The final version uses ALU-* and implements the REBEL-2 instruction set.

When transitioning from a traditional pipelined CPU to a superscalar CPU, the ternary design poses no additional barriers compared to the binary design. Similar architectural concepts to binary can be used because the changes are at data flow and information management rather than at the low-level logic of ternary implementation, as discussed in Section 4.4.

—"The aim of argument, or of discussion, should not be victory, but progress."

Joseph Joubert

5 Discussion

This chapter interprets the findings reported in the previous chapter. The first sections discuss the development findings related to the research questions in Chapter 1. Limitations and validity are considered throughout the discussion and consolidated into individual subsections at the end of the chapter.

5.1 Findings

In computing, the adoption of ternary systems presents unique challenges, particularly in low-level circuit design and instruction set architecture (ISA) development. Despite the similarities between ternary and binary circuits at the architectural level, research on balanced ternary adder design beyond basic implementations is scarce.

This thesis aims to clearly understand the challenges and opportunities in ternary CPU design, providing insight into the advancements in high-performance computing.

5.1.1 Ternary adders

Limited research is available on balanced ternary adder design beyond the implementations of full adders and half adders. Yoon, Baek, Kim *et al.* [36] proposes a balanced ternary carry-select adder design, and finds an optimisation to cut the amount of ripple-carry chains from 3 to 2.

Previous research on implementing fast adders has led to the exploration of ideas for designing faster adders. However, most of this research has focused on unbalanced ternary systems or has not presented a complete design for the adders.

In their paper, Ganesan, Mohan and Soman [60] reviewed various designs of ternary unbalanced and ternary full adders and presented simulation results for each selected design. The paper highlighted the shortcomings in existing designs, specifically pointing out a common misconception among researchers about handling a carry input of 2. To address this, the paper simplified the designs by eliminating the full range of carry values before simulation.

Jones [35] presents a detailed carry-look-ahead description for balanced ternary, but does not determine any equations or logic for the full tree node with the selected ternary encoding.

Yoon, Baek, Kim *et al.* [36] presents a design of a carry-save adder with a carry-select adder for use in a balanced ternary multiplier, and proposes an optimisation of the carry-select adder to reduce the number of ripple-carry chains from 3 to 2. They also argue that a ternary adder could have up to 3 inputs, without needing additional carry values. This matches the information presented in Appendix F, and can be generalised to any balanced number system.

Mechanical Advantage [42] presents a possible design for a carry-select adder. However, the implementation does not appear to add input values properly, and implementation of their design in MRCS did not produce correct results. This conclusion can be reached by observing that the sum of any two input trits is not affected by the carry of the previous. Figure 5.1 shows the result of 1+4 using the adders described. None of the results on the right-hand side shows the correct result, which should have been 5. As the design in the project also shows, any change will only affect the final carry and the immediate sum, which means that it cannot add numbers correctly.

5.1.2 An Expression for Carry-Lookahead

The research discovered and tested expressions and logic to implement a balanced-ternary lookahead adder for 3 and 4 trits, shown in subsection 4.1.4 and Appendix G. An equation was derived to create logic for any number of input trits in subsection 4.1.3.

The proposal introduces a new design option for fast adders in balanced ternary. This is significant because the literature review in Section 2.3 found no published design or description of a full tree node for a balanced ternary carry-lookahead adder.

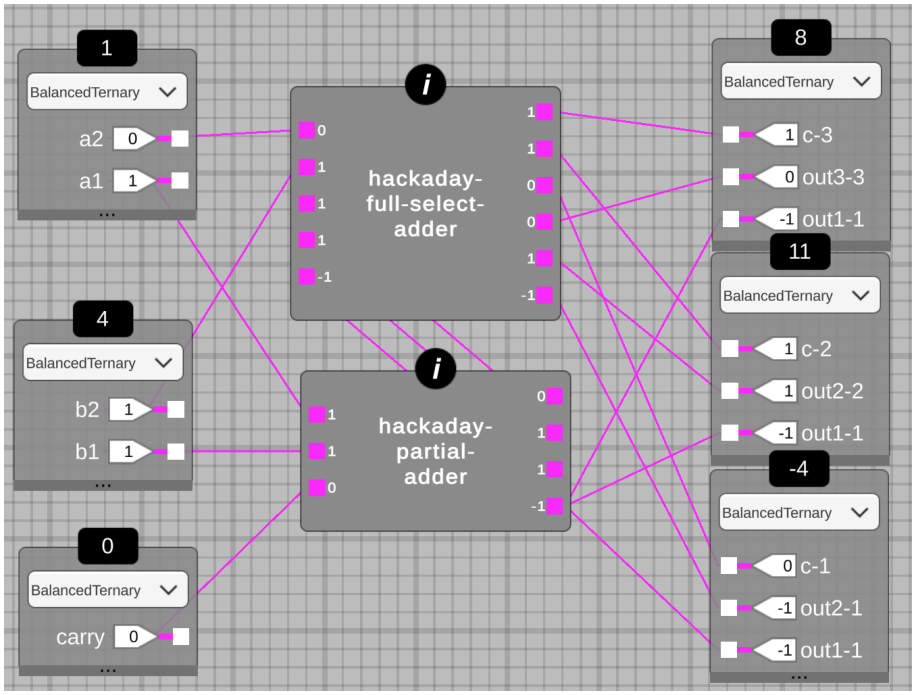


Figure 5.1: 1+4 with the adders described by Mechanical Advantage

All unbalanced adders have been demonstrated to use the same logic in carry-lookahead circuits, regardless of the radix used. Similarly, all balanced adders use the same carry-lookahead logic, regardless of the radix. The same carry-lookahead logic applies to balanced addition with three input digits and one additional carry input. For any balanced number system, the carry information for adding 3 numbers can be calculated similarly for 2. The only change required for a lookahead or carry-select adder is designing an adder that accepts 3 inputs and a carry input.

5.1.3 ALU - binary vs ternary

At the ALU level, ternary differs significantly from binary. This is because the ALU needs to implement specific logical and mathematical functions. The description of these functions in binary and ternary will differ significantly, as will the implementation. Ternary is a design with more than 2 logic levels to consider. The greatest difference in the functions explored for the ALU was found to be in the design of adders, and in the design of comparators. For adders ternary is different because it introduces an extra negative carry, which significantly changes logic implementation. For comparators, it is different because it is now possible to represent the size relations (greater than, equal to, and less

than) between two numbers with a single value. The change in comparators may provide additional advantages that could be exploited, and streamline some instructions.

5.1.4 Ternary CPUs and superscalar

The work indicates that the primary challenges for ternary CPUs are low-level and ISA design. At the architectural circuit design level, ternary does not differ significantly from binary. This results from observing the design of the full pipeline from the REBEL-2-based architecture, where architectural ideas similar to those currently in binary could be employed. This was also supported when looking at superscalar design and not finding any significant changes required for the high-level functionality when using ternary logic.

5.2 Assumptions and Limitations

It is important to note the limitations and assumptions that can affect the validity or generalisability of the findings. The methodology may have introduced biases and resources that might have restricted the investigation's depth. These factors highlight areas for future research to address and enhance the robustness of the conclusions.

- A full CPU design has not been verified or simulated. This also applies to the superscalar presented in Section 4.4, where it lacks a detailed design of a few required components. Some functionality has not yet been implemented in MRCS, making large scale design and verification challenging.
- The method used to discover the expressions for the lookahead signals in subsection 4.1.2 is a heuristic approach and scales poorly as the number of trits increases. Although the method and reasoning behind it are explained, it does not have a rigorous proof. It has been tested and shown to be true for 4 trits.
- Equation (4.8) to find expressions for the ternary lookahead has not been proved, and is only a conjecture based on a heuristic approach. The result appears to be valid, as a few possible rows were sampled, the resulting logic was examined, and it correctly generated the expression for up to 4 trits.

—"There will come a time when you believe everything is finished. Yet, that will be the beginning."

Louis L'Amour

6 Conclusion

This chapter marks the end of the exploration into designing a novel ternary CPU that can be implemented. The research delved into balanced ternary fast adders, and discovered a knowledge gap. Throughout the preceding chapters,ALU-* and a ternary CPU implementing the instruction set from REBEL-2 have been designed. Finally the research explored what is required to implement a ternary superscalar CPU.

As the study is completed, it becomes important to reflect on the insights gained and consider the implications of the findings for future research efforts and practical applications. This chapter offers recommendations based on findings, outlines potential avenues for future research, and summarises the conclusions.

The recommendations presented here are based on empirical evidence and theoretical insights from the investigation.

6.1 Research Questions

The technical focus of this thesis has been to answer the following research questions:

- **RQ1:** What are the functional requirements of the ALU for the REBEL-2 ISA in a ternary system?
- **RQ2:** How can a balanced ternary CPU be implemented based on the REBEL-2 ISA?

RQ2 is a complex topic that requires two separate discussions to explore fully:

  - **RQ2.1:** What existing designs of balanced ternary adders can be adapted for high-speed operations, and what are their comparative advantages and disadvantages?
  - **RQ2.2:** How can one design a carry-lookahead adder for a computing system that uses balanced ternary?

- **RQ3:** What modifications are necessary for the core architecture of the balanced ternary CPU to support superscalar processing?

6.1.1 Results

The results for each of the research questions are summarised below.

- **RQ1:** Functional requirements for ALU-* as described in Section 4.2 have been found designed and tested. This open-source design is available on GitHub[1]. It has been sent to production[2], but the prototype has not yet arrived.
- **RQ2:** A CPU based on the REBEL-2 ISA using balanced ternary has been developed. The finalised design is discussed in Section 4.3, and the steps to get there are described in Appendix C in Sections C.1.1, C.1.2, and C.1.3.
  - **RQ2.1:** The main types of adders currently explored for balanced ternary adders are ripple-carry adders, carry-select adders, and carry-save adders. As in binary, ripple-carry adders have a simple design, but carry calculation propagates slowly. Carry-save adders abandon the carry calculation and quickly add multiple numbers for each step. The carry must be added to the sum later. A carry-select adder aims to reduce the delay by calculating the outcome of all potential carries and then selecting the correct outcome. However, this increases the number of full adders required in the design. This is discussed in Section 2.6.
  - **RQ2.2:** This research question has taken the most effort. It is discussed in Section 4.1. The standard balanced ternary logical notation is not well suited to represent a lookahead adder since not all operators are commutative or distributive. Instead, an alternative notation was used to describe the connection between the carry lookahead output information, and the input signals. This

[1] https://github.com/aiunderstand/tt05-REBEL2-balanced-ternary-ALU
[2] https://tinytapeout.com/runs/tt05/107/

notation shows how the input values combine to form the output value, leading to less repeated terms showing the same logic in this application. Appendix E shows how to get an expression using more common operators in balanced ternary.

- **RQ3:** This is discussed in Section 4.4. The conclusion is that similar architectural ideas to what exists in binary can be employed.

6.1.2 Outcomes of the Thesis

The objectives defined in Chapter 1 have all been achieved with the following outcomes for the thesis:

- Developed a theory on balanced ternary carry-lookahead adders. The research took an unexpected turn upon the realisation that this topic had not been previously explored. This is documented in Section 2.5.
- ALU-* chip has been designed and the design documented in Section 4.2, available as open source on Github [3], tested as documented in Appendix D and sent to production [4].
- A ternary CPU has been designed as documented in Section 4.3.
- Researched ternary superscalar CPU design, documented in Section 4.4.
- A comprehensive documentation has been written.
- A work process has been designed and documented in Appendices A and B.

The resulting designs and documentation have addressed all the research questions.

6.2 Main Contributions of the Research

The research has not found any work closely related to the topic of this thesis. This makes the document a significant contribution to the body of knowledge for ternary computing, especially in the theory of balanced carry-lookahead adders.

[3] https://github.com/aiunderstand/tt05-REBEL2-balanced-ternary-ALU
[4] https://tinytapeout.com/runs/tt05/107/

6.2.1 A novel design for a balanced ternary carry-lookahead adder

This project has defined the logic required to implement a balanced carry-lookahead adder. It proposes simplifying initial nodes in a full carry-lookahead tree for balanced ternary computing. This provides a better design for a fast ternary adder, ensuring that the addition of larger numbers can be done with a lower propagation delay. This is essential for fast balanced ternary processing since addition is one of the most common operations in CPUs.

6.2.2 Design of ALU-*

Throughout this project, ALU- * has been designed. It is a ternary ALU that implements most REBEL-2 instructions (excluding *division*). ALU-* is open source and is available on GitHub [5]. This can act as a starting point for further development of a REBEL-2 based processor, or other processors by reusing ideas presented in the thesis.

6.2.3 A pipelined implementation of REBEL-2

This project has designed CPUv4, a ternary processor that incorporates ALU-* and integrates the remaining functionalities for REBEL-2. This pipelined design, modified with the changes mentioned for superscalar, can help with future development of testing balanced ternary CPUs with the functionality needed for full superscalar design. The basis for this has also been presented in Section 4.4

6.3 Recommendations and Future work

A potential design for a CPU based on the REBEL-2 ISA has been proposed, and the logic for carry anticipation when creating a balanced ternary lookahead adder has been determined. The CPU designs still need to be fully implemented and verified. This was not done as part of this research as the latest design is so complex that it is not feasible to expand with the current version of MRCS. Individual components have been made, but the control logic has not been verified and hazard detection has been omitted. The complete stage buffers have not been implemented due to the resulting size when drawn in MRCS.

[5] https://github.com/aiunderstand/tt05-REBEL2-balanced-ternary-ALU

6.3.1 Lookahead Logic

The lookahead logic still needs to be simulated and compared with other versions of adder circuits, including the carry-select adder. At this point, it is unclear what the speed and power-delay product of the lookahead adder will be in full implementation, and some potential optimisations to the circuit may also be possible.

Other versions of parallel prefix adders exist, such as Kogge-Stone adder [61] and Brent-Kung adder [62] and exhibit different properties regarding speed, fan-out, and size requirements. These can be evaluated with simulations to see how balanced ternary affects performance and scaling.

It is worth examining the usefulness of the addition of 3 input + 1 carry for balanced ternary, as this is possible without complicating the carry logic. Yoon, Baek, Kim *et al.* [36] has used this to design a Wallace tree multiplier with fewer addition steps. It may be good to examine if the CPU has any benefit in having instructions to add 3 numbers and the hardware cost to support it.

The specific implementation of the comparison component of the ALU-* is highly inefficient. It is a problem that can be solved in $O(\log(n))$ time, but the implementation presented is completed in $O(n)$. This can be easily implemented by rearranging the inputs to form a balanced tree for comparison instead. Compare two and two inputs, then compare the results with the same circuits. This was not done in this work, since editing existing components means recreating the entire component from a clear canvas in the current version of MRCS. The repercussions are minimal here, since it is a 2-trit CPU, but it is likely significant for larger values.

The design proposed in subsection C.1.3 cannot fully support REBEL-2. There are 2 missing instructions: ADDi2, and BCEG. These were intentionally omitted from the current design due to the additional connections added to the design to support them. Figure C.3 shows the additional connections required for the last two instructions.

6.3.2 Implementing the full REBEL-2 instruction set on the CPU

The design proposed in Section 4.3 cannot fully support REBEL-2, as two instructions are missing: ADDi2, and BCEG. These were intentionally omitted from the design due to the additional connections needed to support them. Section C.1.4 in Appendix C shows the additional connections required for the last two instructions. The next step to achieve a complete ternary CPU is to design the register shown in Figure C.3.

6.3.3 Superscalar Design

The superscalar design of a ternary CPU is a work in progress. Lower-level components have yet to be designed; full implementation with bus connections requires a new release of MRSC, not yet feasible, but future enhancements and added functionality are expected to enable its realisation.

6.4 Conclusion

This thesis explored a specific implementation of the REBEL-2 ISA, and determined how to implement carry-lookahead logic for balanced ternary.

Reviewing the relevant literature revealed a knowledge gap in implementing the balanced ternary carry-lookahead logic. The lack of a complete carry-lookahead tree node description for balanced ternary revealed a gap that needed to be addressed. Building on previous work on balanced carry-lookahead logic by Jones [35], expressions were developed to describe a full tree-node for carry-lookahead. A way to generate these expressions was also shown, but the size of the expressions is doubled for each additional trit.

This work demonstrates the expressions required to perform the carry-lookahead calculation in a way that can be implemented as a full carry-lookahead tree. This method improves the scaling of propagation delay from $O(n)$ for a ripple-carry adder, to $O(\log(n))$ for a carry-lookahead adder.

Choosing between balanced ternary or binary when designing a CPU implementation does not strongly impact the high-level design. A concept or structure made for binary can be implemented in balanced ternary by replacing the low-level implementation with a balanced ternary version. The main differences appear in specific logic gate design and algorithm implementation, as well as in the ISA.

The initial work required to make the CPU design superscalar has been completed. Future work must focus on full implementation as it becomes feasible with a new update of MRCS.

References

[1] E. Fegri, *Design of a Balanced Ternary Tridirectional Loadable Counter Using CNTFETs*. University of South-Eastern Norway, 2022.

[2] S. Bos, *Beyond 0 and 1: A mixed radix design and verification workflow for modern ternary computers*. University of South-Eastern Norway, 2024.

[3] W. Buchholz, 'Fingers or fists? (the choice of decimal or binary representation),' *Commun. ACM*, vol. 2, no. 12, pp. 3–11, 1959, ISSN: 0001-0782. DOI: 10.1145/368518.368529. [Online]. Available: https://doi.org/10.1145/368518.368529.

[4] G. W. Leibniz, 'Explication de l'arithmétique binaire,' *Journal des Sçavans*, 1703, Published as part of a journal article.

[5] G. Ifrah, *The Universal History of Computing: From the Abacus to the Quantum Computer*. 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:208796589.

[6] W. A. D. Hilbert, *Grundzüge der theoretischen Logik*. Berlin: Springer, 1938.

[7] A. M. Turing, 'On computable numbers, with an application to the entscheidungsproblem,' *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. DOI: https://doi.org/10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230. [Online]. Available: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[8] F. L. Bauer, *Origins and Foundations of Computing In Cooperation with Heinz Nixdorf MuseumsForum*. Springer Berlin, Heidelberg, 2009.

[9] J. V. Atanasoff, 'Advent of electronic digital computing,' *Annals of the History of Computing*, vol. 6, pp. 229–282, 1984. [Online]. Available: https://api.semanticscholar.org/CorpusID:34553374.

[10] B. Randell, 'On alan turing and the origins of digital computers,' *Machine intelligence*, 1972. [Online]. Available: https://api.semanticscholar.org/CorpusID:64461042.

[11] A. W. Burks, H. H. Goldstine and J. von Neumann, 'Preliminary discussion of the logical design of an electronic computing instrument (1946),' in *Perspectives on the Computer Revolution*. USA: Ablex Publishing Corp., 1989, pp. 39–48, ISBN: 0893913693.

[12] A. Kanduri, A. M. Rahmani, P. Liljeberg, A. Hemani, A. Jantsch and H. Tenhunen, 'A perspective on dark silicon,' in *The Dark Side of Silicon: Energy Efficient Computing in the Dark Silicon Era*. Cham: Springer International Publishing, 2017, pp. 3–20, ISBN: 978-3-319-31596-6. DOI: 10.1007/978-3-319-31596-6_1. [Online]. Available: https://doi.org/10.1007/978-3-319-31596-6_1.

[13] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous and A. LeBlanc, 'Design of ion-implanted mosfet's with very small physical dimensions,' *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/JSSC.1974.1050511.

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V edition: The Hardware Software Interface*, 2nd ed. Morgan Kaufmann, 2020, ISBN: 978-0-12-820331-6.

[15] L. T. Schoenbaum and W. K. Al-Assadi, 'Binary/ternary logic applications for systems programming and reversible computing,' in *2019 SoutheastCon*, 2019, pp. 1–5. DOI: 10.1109/SoutheastCon42311.2019.9020479.

[16] W. Alexander, 'The ternary computer,' *Electronics and Power*, vol. 10, no. 2, pp. 36–39, 1964. DOI: 10.1049/ep.1964.0037.

[17] A. Obiniyi, A. Ezugwu and A. Kwanashie, 'Arithmetic logic design with colorcoded ternary for ternary computing,' *International Journal of Computer Applications*, vol. 26, Jul. 2011. DOI: 10.5120/3162-2929.

[18] S. Kak, *On ternary coding and three-valued logic*, 2018. arXiv: 1807.06419 [cs.AI].

[19] S. Bos, J. B. Nilsen and H. Gundersen, 'Post-binary robotics: Using memristors with ternary states for robotics control,' in *2020 IEEE 8th Electronics System-Integration Technology Conference (ESTC)*, 2020, pp. 1–6. DOI: 10.1109/ESTC48849.2020.9229820.

[20] S. Bos, H. Gundersen and F. Sanfilippo, 'Umemristortoolbox: Open source framework to control memristors in unity for ternary applications,' in *2020 IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL)*, 2020, pp. 212–217. DOI: 10.1109/ISMVL49045.2020.000-3.

[21] S. Bos, H. N. Risto and H. Gundersen, 'Beyond cmos: Ternary and mixed radix cntfet circuit design, simulation and verification,' in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 80–85. DOI: 10.1109/ISCAS48785.2022.9937259.

[22] S. Bos and H. Gundersen, 'Ternary computing; the future of iot?' *Society for Design and Process Science*, pp. 43–47, 2021.

[23] H. N. Risto, S. Bos and H. Gundersen, 'Automated synthesis of netlists for ternary-valued n-ary logic functions in cntfet circuits,' in *Proceedings of The 61st SIMS Conference on Simulation and Modelling SIMS 2020*, 2020, pp. 483–485. DOI: 10.3384/ecp20176483.

[24] S. Bos, H. N. Risto and H. Gundersen, 'High speed bi-directional binary-ternary interface with cntfets,' *Society for Design and Process Science*, pp. 38–42, 2021.

[25] D. Kam, J. G. Min, J. Yoon, S. Kim, S. Kang and Y. Lee, 'Design and Evaluation Frameworks for Advanced RISC-based Ternary Processor,' in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2022, pp. 1077–1082. DOI: 10.23919/DATE54114.2022.9774584.

[26] S. Gadgil, G. N. Sandesh and c. Kumar V, 'Design and implementation of a cntfet-based ternary logic processor,' Mar. 2023. DOI: 10.36227/techrxiv.22259437.v1. [Online]. Available: http://dx.doi.org/10.36227/techrxiv.22259437.v1.

[27] J. P. Shen and M. H. Lipasti, *Modern Processor Design Fundamentals of Superscalar Processors*. Waveland Press, Inc., 2005.

[28] J. Zhao, B. Korpan, A. Gonzalez and K. Asanovic, 'SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine,' en, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221212196.

[29] S. Mashimo, A. Fujita, R. Matsuo *et al.*, 'An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor,' in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2019, pp. 63–71. DOI: 10.1109/ICFPT47387.2019.00016. [Online]. Available: https://ieeexplore.ieee.org/document/8977924 (visited on 23/10/2023).

[30] R. M. Tomasulo, 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units,' *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967, Conference Name: IBM Journal of Research and Development, ISSN: 0018-8646. DOI: 10.1147/rd.111.0025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5392028 (visited on 08/11/2023).

[31] A. Antonov, 'Superscalar Out-of-Order RISC-V ASIP Based on Programmable Hardware Generator with Decoupled Computations and Flow Control,' in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, ISSN: 2637-9511, Jun. 2022, pp. 1–4. DOI: 10.1109/MECO55406.2022.9797182. [Online]. Available: https://ieeexplore.ieee.org/document/9797182 (visited on 23/10/2023).

[32] D. W. Jones, *Standard ternary logic*, Personal Website, 2012. [Online]. Available: https://homepage.cs.uiowa.edu/~dwjones/ternary/logic.shtml.

[33] R. D. Merrill, 'Ternary logic in digital computers,' in *Proceedings of the SHARE Design Automation Project*, ser. DAC '65, New York, NY, USA: Association for Computing Machinery, 1965, pp. 6.1–6.17, ISBN: 9781450379359. DOI: 10.1145/800266.810759. [Online]. Available: https://doi.org/10.1145/800266.810759.

[34] H. N. Risto, *A study of CNTFET implementations for ternary logic and data radix conversion*. University of South-Eastern Norway, 2020.

[35] D. W. Jones, *Arithmetic in ternary computers*, Personal Website, 2013. [Online]. Available: https://homepage.cs.uiowa.edu/~dwjones/ternary/arith.shtml.

[36] J. Yoon, S. Baek, S. Kim and S. Kang, 'Optimizing ternary multiplier design with fast ternary adder,' *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 2, pp. 766–770, 2023. DOI: 10.1109/TCSII.2022.3210282.

[37] V. Vallabhuni, 'A review on n-bit ripple-carry adder, carry-select adder and carry-skip adder,' *Journal of VLSI circuits and systems*, vol. 4, no. 01, pp. 27–32, Mar. 2022. DOI: 10.31838/jvcs/04.01.05. [Online]. Available: https://vlsijournal.com/index.php/vlsi/article/view/40.

[38] N. S. Soliman, M. E. Fouda, L. A. Said, A. H. Madian and A. G. Radwan, 'N-digits ternary carry lookahead adder design,' in *2019 31st International Conference on Microelectronics (ICM)*, 2019, pp. 142–145. DOI: 10.1109/ICM48031.2019.9021861.

[39] H. Ling, 'High-speed binary adder,' *IBM Journal of Research and Development*, vol. 25, no. 3, pp. 156–166, 1981. DOI: 10.1147/rd.252.0156.

[40] H. Gundersen and Y. Berg, 'Fast addition using balanced ternary counters designed with cmos semi-floating gate devices,' in *37th International Symposium on Multiple-Valued Logic (ISMVL'07)*, 2007, pp. 30–30. DOI: 10.1109/ISMVL.2007.23.

[41] V. Øverås, *Carry-Look-Ahead Adder in Multiple-Valued Recharge Logic*. University of Oslo, 2005.

[42] M. Advantage. 'A fast balanced ternary adder/subtracter.' (2019), [Online]. Available: https://hackaday.io/project/164907-ternary-computing-menagerie/log/169827-a-fast-balanced-ternary-addersubtracter.

[43] D. Silverman, *Interpreting Qualitative Data*. Sage Publications, 2006, ISBN: 9780857024213.

[44] J. v. Brocke, A. Hevner and A. Maedche, 'Introduction to design science research,' in Sep. 2020, pp. 1–13, ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

[45] Hassan, Muhammad, *Experimental Design – Types, Methods, Guide*, https://researchmethod.net/experimental-design/, 2024.

[46] R. E. Stake, *The Art of Case Study Research*. Thousand Oaks, CA: Sage Publications, 1995, ISBN: 9780803957671.

[47] IEEE xplore, *IEEE*, https://www.ieee.org/, Institute of Electrical and Electronics Engineers, 2024.

[48] Google Scholar, *Google Scholar*, https://scholar.google.com/, 2024.

[49] ACM Digital Library, *ACM Digital Library*, https://clarivate.com/products/scientific-and-academic-research/research-discovery-and-workflow-solutions/webofscience-platform/, 2024.

[50] Elsevier, *Scopus: Comprehensive, multidisciplinary, trusted abstract and citation database*, https://www.elsevier.com/products/scopus, 2024.

[51] Clarivate, *Web of Science platform*, https://clarivate.com/products/scientific-and-academic-research/research-discovery-and-workflow-solutions/webofscience-platform/, 2024.

[52] the LaTeX project, *The LATEX project*, https://www.latex-project.org/, 2024.

[53] Overleaf, *Overleaf*, https://www.overleaf.com/, 2024.

[54] Lucidchart, *Lucidchart*, https://www.lucidchart.com/pages, 2024.

[55] OpenAI, *ChatGPT*, https://openai.com/gpt-3/, 2022.

[56] Grammarly, *Grammarly*, https://www.grammarly.com/, 2024.

[57] S. Bos, *Mixedradixcircuitsynthesis*, https://github.com/aiunderstand/MixedRadixCircuitSyn 2023.

[58] S. Bos, *Tt05-rebel2-balanced-ternary-alu*, https://github.com/aiunderstand/tt05-REBEL2-balanced-ternary-ALU, 2023.

[59] O. C. Moholth and S. Bos, *107 : Rebel-2 balanced ternary alu*, https://tinytapeout.com/runs/tt05/107/, This 2-trit balanced ternary ALU is part of the REBEL-2 balanced ternary logic CPU, 2023.

[60] K. Ganesan, N. Mohan and K. P. Soman, 'Superscalar out-of-order risc-v asip based on programmable hardware generator with decoupled computations and flow control,' *IET Circuits, Devices & Systems*, 2023. DOI: 10.1049/cds2.12152. [Online]. Available: https://doi.org/10.1049/cds2.12152.

[61] P. M. Kogge and H. S. Stone, 'A parallel algorithm for the efficient solution of a general class of recurrence equations,' *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973. DOI: 10.1109/TC.1973.5009159.

[62] Brent and Kung, 'A regular layout for parallel adders,' *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982. DOI: 10.1109/TC.1982.1675982.

[63] R. A. Day and B. Gastel, *How to Write and Publish a Scientific Paper*, 7th. Cambridge University Press, 2012.

Appendices

List of Appendices

A

DSR Implementation

The documentation is the main outcome of this thesis and is structured according to the IMRaD model (Introduction, Methods, Results and Discussion) by Day & Gastel Day and Gastel [63]. IMRaD provides a logical flow and a consistent format to assist the reader in following the research progress. Separate sections improve readability and make it easier for readers to find relevant information. This structure ensures a systematic and digestible presentation of scientific ideas, enhancing comprehension by offering a complete picture with context, methodology, outcomes, and implications step-by-step. Furthermore, IMRaD encourages precision, prompts thoroughness, and clear articulation of each research aspect, which promotes detailed and precise reporting and is well suited for a project using DSR as a methodology.

A.1 Research Approach

The project was structured using DSR as the research methodology chosen. This meant having a backlog of tasks/activities/functionality and choosing the next one in cooperation with my supervisors. All tasks can be put into one of these categories, but using this methodology means that any insight might require reworking the results. This helped to keep track of the remaining tasks and progress towards the final result. It also helped maintain a focused and transparent workflow that was easy for the supervisors to understand. This was especially important because my thesis initially focused on the technical task of designing a ternary CPU for the REBEL-2 ISA in a ternary system.

The project includes the following stages:

- Propose a design for a REBEL-2 based microprosessor
- Propose a design for a superscalar microprocessor
- Define gaps in current research on ternary computing

Our chosen methodology has not only helped us evaluate and focus on project challenges but has also, at times, opened up new avenues for research. These unforeseen opportunities and challenges have allowed us to go beyond what we expected when we started this process.

Iterative methodology has several advantages, especially for a project like this one, where we do something new and learn as we go. The most important thing for this project has been that it is flexible, results-driven and enables quick decision-making.

Flexibility allows for change, which means that there is always room for mistakes and an opportunity to iterate and learn. Our implementation of DSR has short and regular deadlines, which makes it easier to avoid procrastination. Short deadlines also apply to decision making, and enabled quick decisions about research direction. Thankfully, flexibility and our methodology have encouraged trying and failing since they promote learning. Instead of focusing on the process, we have been driven to achieve milestones and results.

A.2 Practical use of DSR as a methodolgy

- Supervisors give initial problem
- Divide the project into activities with specific deliverables.
- Iteratively improve the design and the architectural models based on the supervisors' feedback and test results.
- The goal in the execution of the research is to address any issues that arise during the implementation phase while keeping an open mind to ideas that materialise throughout this phase. This means that the problem evolves while working on it.

A.2.1 Planned tasks and subtasks

- Research the instruction set for REBEL-2
- Design an ALU for the REBEL-2 ISA in a ternary system
- CPU Architecture - version 1 - the Minimum Viable Product (MVP)
  - Planning
  - Creating register file

  - Routing the signals and planning control logic
- CPU Architecture - version (n) to be repeated until satisfied
  - Improving the design
- Research scaling to superscalar
  - research Tomasulo's algorithm
  - Research ternary adders

B
Workflow

B.1 Introduction

The thesis takes me on a conceptual journey, as illustrated in Figure B.1. Appendix B details all the tasks on this journey. This section should be viewed in relation to the Gantt diagram in Figure 3.2. The colours in these two diagrams are coordinated to enable the reader to see the connection between them. Yellow represents preliminary studies and preparation, blue indicates technical work that results in documented designs, and green signifies technical work resulting in documentation only.

The first two steps focus on preparatory work and reading as described in Chapter 1 and Chapter 2.

The third step is documented in Section 4.2. Step 3 gives the first technical milestone, ALU-*. The work involves designing a functional ALU based on a ternary ISA called REBEL-2 shown in Figure 2.1 found in Steven Bos' doctoral dissertation Bos [2]. ALU-* is a processor component that uses two trits. The concept for ALU-* was designed using MRCS, an open-source design software available on Github Bos [57]. You can also find the ALU-* source code on Github Bos [58]. Although ALU-* has been sent for production, it has yet to arrive at the time of writing.

Steps 4-6 involve incremental development of the remaining parts of a ternary CPU after designing the ALU. Section 4.3 describes the details in the latest design, CPUv3 and Appendix C documents the status at the end of each step in the process.

Step 7 is a theoretical step during which this project examines the necessary steps to scale and implement a superscalar design. This is documented in Section 4.4.

The last two working steps have been dedicated to examining, planning and designing a fast ternary lookahead adder. Step 8 looked at different types of ternary adders and is documented in Section 2.6. Section 4.1 contains the discussion and design for a fast ternary lookahead adder.

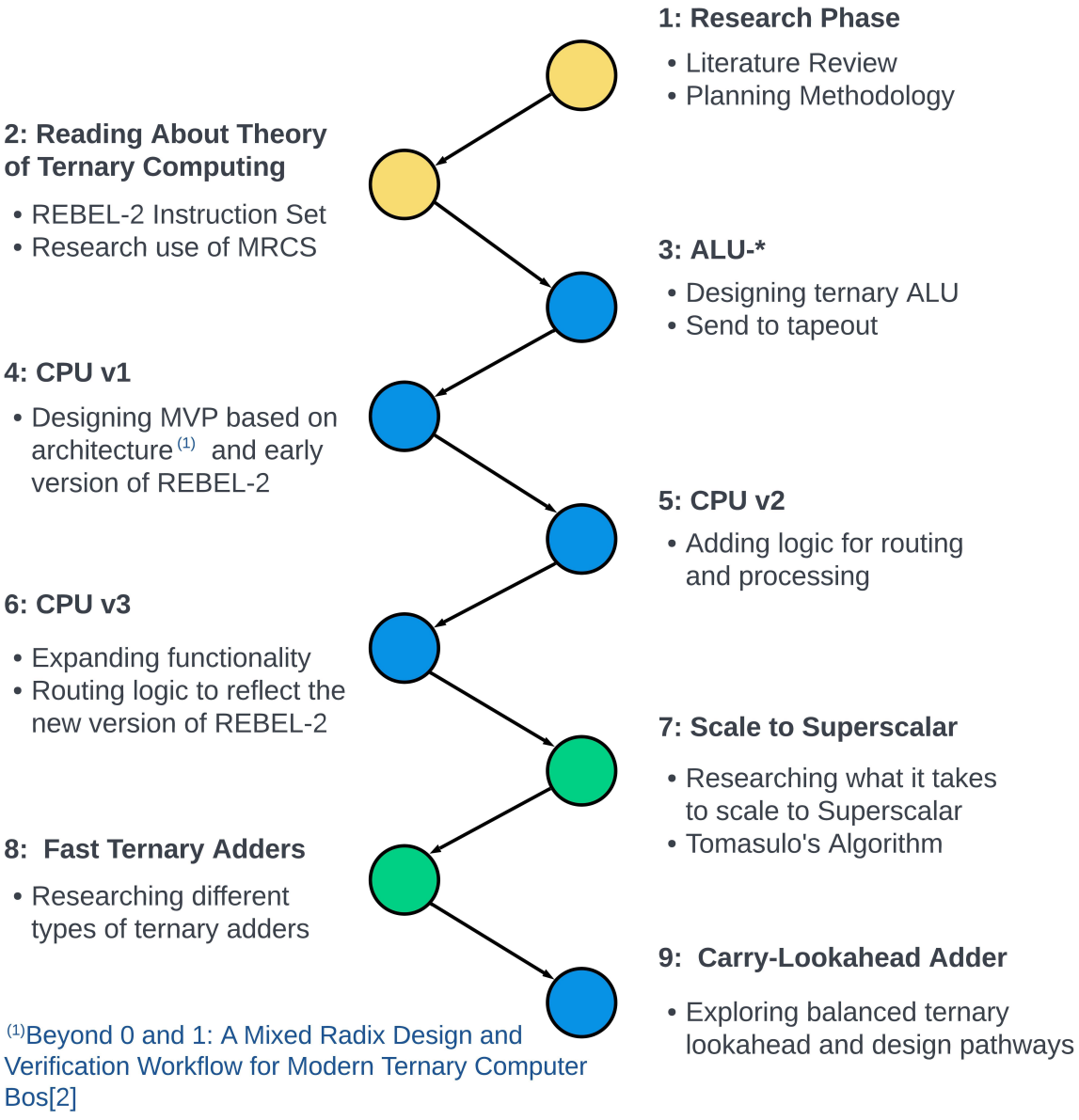


Figure B.1: Workflows through the thesis

B.2 Detailed Workflow

Figure B.1 shows our steps when writing this thesis. For an approximate timeline for these activities, refer to the Gantt chart in Section 3.1, Here is a comprehensive summary of the steps we performed throughout our work on this thesis:

1. **Research phase**
    - Researching methodologies
    - Planning thesis structure, setting up Latex
    - Searching for and reading related work
2. **Studying tools to**
    - MRCS
    - Rebel-2
3. **ALU-***
    - Went through all the instructions for REBEL-2, the ISA defined in Bos [2]. Identified the required instructions for ALU-* to satisfy the ISA. We started planning the required designs. Identified main functions required to satisfy the ISA (in order from top to bottom of the instruction set):
        - Add
        - Multiply
        - Divide
        - Min
        - Max
        - Shift
        - Compare
    - We discovered that compare could be used to determine min and max with a multiplexer and the compare result instead of separate circuitry to determine it.
    - Made a full adder and half adder using ternary logic
    - Made a simple multiplication circuit for 2x2 trits

- Made the shift operator, based on barrel shift implemented with a multiplexer
- Implemented word-wise comparator
- We found that Bos had only planned for trit-wise compare/min/max, while we assumed it was for whole words.
- Implemented trit-wise comparator
- Decided on control signals that could control the ALU-* output, set up the multiplexers so the correct signal is routed to the output based on that.

4. **CPUv1**
    - Started planning the surrounding components and architecture.
    - Created a simple register file based on Steven's design
    - Deviated from what Steven asked for when making the program counter
    - Steven wanted me to use the ternary counter that could load a value - we found it easier to store the value in a clocked latch instead, and to have a component that incremented by 1, that looped back to the latch
    - Spent time routing the correct signals and planning control logic. What signals get routed where, etc.
    - Current design was made not accounting for instructions reading multiple values from register (BCEG), not accounting for instructions that write multiple values at once (ADDi2)
    - Added buffers to make it a staged processor.
5. **CPUv2**
    - Without creating the register, assumed the component existed. Routed the remaining signals and fixed errors made in the previous step for signal routing/accounted for changes in ISA. Current design, pipelined and not superscalar, hazards not accounted for
6. **CPUv3**
    - MRCS
    - Rebel-2
7. **Scale to superscalar**

- Tomasulo's Algorithm

8. **Ternary Adders**
    - Research and reading related works
9. **Fast ternary adders**
    - Implementations of fast balanced ternary adders
    - Attempts at designing it
    - Ternary manifesto - exploring balanced ternary lookahead
    - How to design balanced ternary lookahead adders

C

CPU Design Stages

C.1 Introduction

This describes the steps involved in creating the design for the ternary CPU described in Section 4.3.

subsection C.1.1, subsection C.1.2, and subsection C.1.3 describe the progress made at the end of each of Steps 4-6, which represents the current design process for our ternary CPU. subsection C.1.4 shows future improvements to the CPU to be able to support REBEL-2 fully.

C.1.1 Step 4 - Designing CPU v1

The first step in designing a ternary CPU is defining a minimum viable product (MVP), which is the cornerstone of this part of the thesis.

In the initial step, the design was intended to incorporate any working instructions. The decision was made to start with the most straightforward instructions, which formed the largest group of instructions that shared a format. These instructions were first implemented: ADD, MIMA, SHI, and COMP. However, due to the limitations of the design, functions such as modifying the destination address and jumping for the program counter could not be supported. Version 1 consists primarily of the fundamental building blocks, including ALU-* and the register.

The instructions that could be supported at this stage of the design process are highlighted in green in Table C.1. In this table, green means implemented, while red means not yet implemented.

Instruction name
ADD
ADDi
ADDi2
MUDI
MIMA
SHI
COMP
BCEG
PCO

Table C.1: Instructions supported by v1 of the CPU design

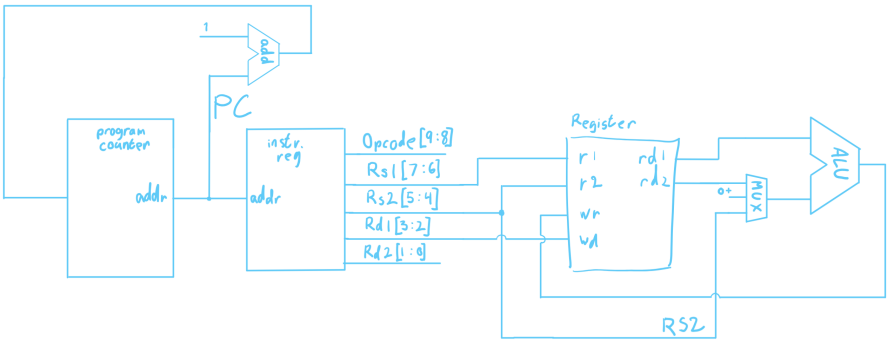


Figure C.1: CPUv1 after Step 4

C.1.2 Step 5 - Designing CPU v2

The following version (Figure C.2) was developed in Step 5, but not all instructions are supported. As discovered later, the description of the instruction set was misunderstood. This was due to an early version of REBEL-2 that contained an imprecise description, leaving room for interpretation. As a result, the PCO instruction had partial support. ADDi now works as addressing a new location can now be done, and one of the modes of PCO is supported.

Table C.2 shows the instructions supported in version 2. In this table, green means implemented, red means not yet implemented, and yellow means partially implemented.

Instruction name
ADD
ADDi
ADDi2
MUDI
MIMA
SHI
COMP
BCEG
PCO

Table C.2: Instructions supported by v2 of the CPU design

C.1.3 Step 6 - Designing CPU v3

While designing the ternary CPU, consultations were held with Bos, who was completing his PhD dissertation at the time. As a result, REBEL-2 underwent some alterations, and the final design draft in the section reflects these changes. The error causing incorrect PCO behaviour in the previous version has been corrected, so the command should be fully supported. The CPU can now jump to the correct address specified by the PCO instruction. The final two instructions BCEG and ADDi2 are still not supported by this design, as they require modifications to the register.

Table C.3 shows which instructions in REBEL-2 the latest version of the ternary CPU supports. In this table, green means implemented, while red means not yet implemented.

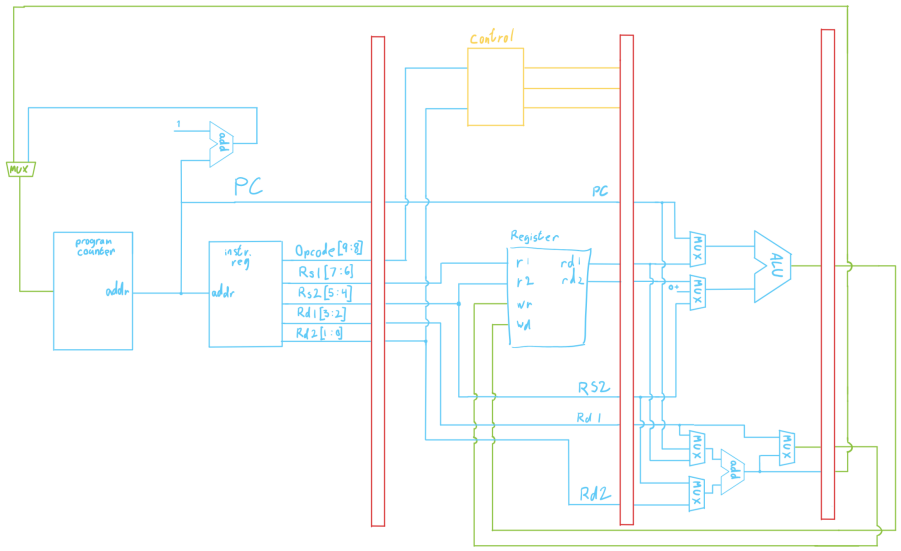


Figure C.2: CPUv2 after Step 5

Instruction name
ADD
ADDi
ADDi2
MUDI
MIMA
SHI
COMP
BCEG
PCO

Table C.3: Instructions supported by v3 of the CPU design

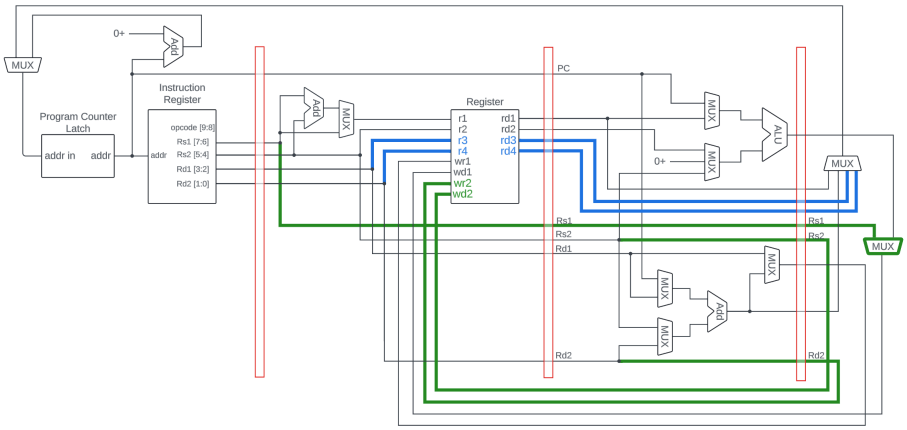


Figure C.3: Future CPUv4 design supporting the final two instructions

C.1.4 Further work on the ternary CPU - CPUv4

C.1.5 CPU Design

The design proposed in subsection C.1.3 cannot fully support REBEL-2. There are 2 missing instructions: ADDi2, and BCEG. These were intentionally omitted from the design due to the additional connections added to the design to support them. Figure C.3 shows the additional connections required for the last two instructions. For ADDi2, green connections need to be added. This is because the instructions are written to two locations in the register instead of one as the rest. This adds complexity to the design of the register so it can support simultaneous writes. BCEG needs to add the blue connections because the instruction compares two values and conditionally jumps to one of two addresses from the register. As a result, up to four values must be read from the register for a single instruction.

The next step to achieve CPUv4 is to design the register shown in the figure. As described earlier, this CPU needs to be able to read 4 values and write 2 values simultaneously.

D

Test of designs in MRCS

MRCS has some support for testing within the application. This is done by defining a sequential set of inputs, and the expected output for each input. This allows for testing both combinational and sequential circuits. This has been used throughout the project to verify various components, as the complexity quickly grew to a point where manual testing became error prone, and could miss problems. The tests for these have been manually created with the help of Excel formulas to ensure that the output is correct. The spreadsheet was exported to a CSV, and cleaned up to be used in MRCS

D.1 Tests performed

Two sets of test files have been created and performed on the components in MRCS. These are tests for the ALU, and tests for adders. There are 2 tests for adders. One that works on the addition level, and one specifically for the carry-lookahead signals. Testing only the carry-lookahead component directly for 3 trits helped prove the functionality of the logic, and is faster than testing the full addition. This is because all possible inputs for an adder without carrying input give $3^6 = 726$ combinations. The lookahead tests contain half the number of inputs in the form of ternary-encoded septenary numbers. This gives $7^3 = 343$ possible combinations. It still adequately tests the functionality of the carry-lookahead, and the adder functionality can be tested separately. The ALU tests have been broken into individual files to test each functionality and combined into a single file test_alu.csv. These files test every combination that could be expected to occur with the ALU and were used to verify the logic.

The following tests have been created and performed. The input files are available as a *.zip* folder and have been uploaded to wiseflow as an appendix.

D.1.1 adder_test

This test can be used to test 3-bit adders. It does not account for carry input, but does check carry output.

D.1.2 lookahead_test

This test has been set up to verify the functionality of the 3rd trit of a carry-lookahead node. In the interest of time, the latest version did not extensively test the cases where the most significant trit generated a carry. These cases were left out due to the simplicity in logic and the number of tests it includes. The remaining cases are fully included, and have been verified with the 3 trit lookahead equations.

D.1.3 test_alu

This test extensively tests the defined function codes of the ALU with all the available inputs. However, not all tests will be run due to the size, as MRCS limits the number of tests to 999. Each test was used to verify the ALU design as an individual design, so all tests were performed and passed.

D.1.4 test_alu_add

The test cases were included in test_alu, and tests addition.

Conducted without errors.

D.1.5 test_alu_mul_max

The test cases were included in test_alu, and test the high values of multiplication.

Conducted without errors.

D.1.6 test_alu_mul_min

The test cases were included in test_alu, and test the low values from multiplication.

Conducted without errors.

D.1.7 test_alu_shift_circular

The test cases were included in test_alu, and test circular shift.

Conducted without errors.

D.1.8 test_alu_shift_left

The test cases were included in test_alu, and test left shift.

Conducted without errors.

D.1.9 test_alu_shift_right

The test cases were included in test_alu, and test right shift. Conducted without errors.

D.1.10 test_alu_sub

The test cases were included in test_alu, and test subtraction.

Conducted without errors.

D.1.11 test_alu_triteq

The test cases were included in test_alu, and test the trit-wise compare.

Conducted without errors.

D.1.12 test_alu_tritmax

The test cases were included in test_alu, and test the trit-wise max.

Conducted without errors.

D.1.13 test_alu_tritmin

The test cases were included in test_alu, and tests the trit-wise min.

Conducted without errors.

D.1.14 test_alu_word_eq

The test cases were included in test_alu, and test the word-wise compare.

Conducted without errors.

D.1.15 test_alu_word_max

The test cases were included in test_alu, and test the word-wise max.

Conducted without errors.

D.1.16 test_alu_word_min

The test cases were included in test_alu, and test the word-wise min.

Conducted without errors.

E

Standard Notation of Lookahead Logic

The lookahead logic, as presented in subsection 4.1.2, does not use any form of standard ternary algebra but was presented that way for the simplicity of the notation. The notation can be rewritten to use standard 2 input gates by following a few steps. This is the same method used to produce the equation shown earlier.

Looking at the original equation for P set up from the table, shown in Equation (E.1), we can classify the separate parts of the equation. We can see that it is made up of 3 checks, where the statement is true if any one of them is true. This matches with the ternary any operator $\boxplus$. Looking at the first of the three statements, it is true if both of them have the same value, and both of them caries. This matches the ternary consensus operator $\boxtimes$. The middle equation is the part without direct parallel with basic logic gates. The truth table for this expression as a product of $m_1$and g_0 is shown in Table E.1. We can, however, see that it appears to be somewhere between any and consensus. If we look at the expressions in more detail, we can see that they all consist of expressions where some variables must all be the same value, and some must all be the same value or be 0. We can get part of this product by using consensus between all values that must be the same value and using the result of that with any value that must be 0 or match. By taking that product and retaking the consensus with the values that must all be the same, you will end up with the same value. That means that $(m_1 = c) \wedge (g_0 \neq -c)$ becomes $(m_1 \boxplus g_0) \boxtimes m_1$. The full expression for the 2 trit propagate sum is shown in Equation (E.1). The full pattern is clearer if a part of the 3 trit equations is used. $(m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)$ becomes $(m_2 \boxplus m_1) \boxtimes (m_2 \boxplus g_0) \boxtimes m_1$.

The full expression for converting P for 3 trits to e can be seen in Equation (E.3). This expression has been implemented and verified in MRCS.

$$P = c \overset{def}{\iff} ((p_1 = c) \wedge (p_0 = c)) \vee ((m_1 = c) \wedge (g_0 \neq -c)) \vee (g_1 = c) \qquad c \in \{-1, 1\} \tag{E.1}$$

$$P_1 = (p_1 \boxtimes p_0) \boxplus ((m_1 \boxplus g_0) \boxtimes m_1) \boxplus g_1 \tag{E.2}$$

m_1	g_0		
	-	**0**	+
-	-	-	0
0	0	0	0
+	0	+	+

Table E.1: Part of definition of P for 2 trits, as a truth table

$$\begin{aligned} P_2 = {} & (p_2 \boxtimes p_1 \boxtimes p_0) \boxplus \\ & (((p_2 \boxtimes m_1) \boxplus g_0) \boxtimes (p_2 \boxtimes m_1)) \boxplus \\ & (((m_2 \boxtimes p_0) \boxplus g_1) \boxtimes (m_2 \boxtimes p_0)) \boxplus \\ & ((m_2 \boxplus m_1) \boxtimes (m_2 \boxplus g_0) \boxtimes m_2) \boxplus \\ & (p_2 \boxtimes g_1) \boxplus \\ & ((m_2 \boxplus p_1) \boxtimes m_2) \boxplus \\ & g_2 \end{aligned} \tag{E.3}$$

Jones [35] mentions the possibility of representing expressions with threshold logic as well. In fact, this is possible. Assuming the encoding described in Section 4.1, the output signal can also be described by thresholds and weights as set up in Equations (E.4), (E.5), (E.6)

$$G(g_2, m_2, p_2, g_1, m_1, p_1) = \begin{cases} -1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \leq -t_3 \\ 1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \geq t_3 \\ 0 & \textit{otherwise} \end{cases} \tag{E.4}$$

$$M(g_2, m_2, p_2, g_1, m_1, p_1) = \begin{cases} -1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \leq -t_2 \\ 1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \geq t_2 \\ 0 & \textit{otherwise} \end{cases} \tag{E.5}$$

$$P(g_2, m_2, p_2, g_1, m_1, p_1) = \begin{cases} -1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \leq -t_1 \\ 1 & w_{2,3} \cdot g_2 + w_{2,2} \cdot m_2 + w_{2,1} \cdot p_2 + w_{1,3} \cdot g_1 + w_{1,2} \cdot m_1 + w_{1,1} \cdot p_1 \geq t_1 \\ 0 & \textit{otherwise} \end{cases} \tag{E.6}$$

Although multiple solutions exist, one possible solution with weights and thresholds that produce the correct values is presented in Equations (E.7), (E.8), (E.9)

$$
\begin{aligned}
w_{1,1} &= 3 \\
w_{2,1} &= 10 \\
t_1 &= 13
\end{aligned}
\tag{E.7}
$$

$$
\begin{aligned}
w_{1,2} &= 4 \\
w_{2,2} &= 12 \\
t_2 &= 17
\end{aligned}
\tag{E.8}
$$

$$
\begin{aligned}
w_{1,3} &= 5 \\
w_{2,3} &= 14 \\
t_3 &= 21
\end{aligned}
\tag{E.9}
$$

F

About Balanced Carry and Addition

In subsection 4.1.6 it was stated that the carry can never exceed 1 for every input digit with or without the carry. For balanced ternary, this is true even when adding 3 numbers.

We stated that the highest value digit in any balanced number system with radix r is $\frac{r-1}{2}$. For a carry to exceed 1, we need the sum to exceed $r+\frac{r-1}{2}$. In radix 5, this would be the number 12_5 (7_{10}). Building upon this, we can determine the difference between the sum of 3 max digits, and the highest value possible before the carry exceeds 2.

$$3 \cdot \frac{r-1}{2} + x = r + \frac{r-1}{2} \tag{F.1}$$

$$3 \cdot (r-1) + 2 \cdot x = r + r - 1 \tag{F.2}$$

$$3 \cdot (r-1) + 2 \cdot x = 2r + r - 1 \tag{F.3}$$

$$2 \cdot x = 3r - 1 - 3r - (-3) \tag{F.4}$$

$$2 \cdot x = 2 \tag{F.5}$$

$$x = 1 \tag{F.6}$$

Equation (F.1) shows the difference X between adding 3 max digit numbers in radix r, compared to the highest allowed digit before carry is 2. Solving for x gives us that the difference is 1. This means that in a balanced number system with radix r, any 3 numbers added together with a carry input still cannot exceed a carry output of 1. This leads to the conclusion that the carry anticipation logic for 2 and 3 input adders can be implemented the same way using balanced numbers, and the only added complexity would be in the initial logic combining the inputs.

F.1 Balanced addition through unbalanced adders

Balanced addition can also be done through unbalanced addition with a bias. This means that a balanced addition can be performed with two unbalanced addititons. The first

addition adds the two unbalanced numbers. The second addition adds the bias. This is explained in more detail by Jones [35]

G

Lookahead Expressions With More Trits

For completeness, additional expressions for lookahead information with 4 trits and 5 trits have been included. The expressions expression for 4 trit was found manually in the same way as for 3 trits and 2 trits. The expression for 5 trits was found using Equation (4.8). It may be possible to group various parts of the equation to require less logic gates, at the cost of slower propagation in the unique stage of the expression. More and more of the expressions will be reused as the trit count increases, more and more parts of the expressions are repeated and reused.

For each of these expressions, you can modify it to give M by mapping $(p_0, g_0) \rightarrow (m_0, m_0)$, and to G by mapping $(p_0, g_0) \rightarrow (g_0, p_0)$

G.1 4 Trit Lookahead

$$
\begin{aligned}
P = c \overset{def}{\Longleftrightarrow} &((p_3 = c) \wedge (p_2 = c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((p_3 = c) \wedge (p_2 = c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((p_3 = c) \wedge (m_2 = c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((p_3 = c) \wedge (m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((m_3 = c) \wedge (g_2 \neq -c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((m_3 = c) \wedge (g_2 \neq -c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((m_3 = c) \wedge (m_2 \neq -c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((m_3 = c) \wedge (m_2 \neq -c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((p_3 = c) \wedge (p_2 = c) \wedge (g_1 = c)) \vee \\
&((p_3 = c) \wedge (m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&((m_3 = c) \wedge (g_2 \neq -c) \wedge (g_1 = c)) \vee \\
&((m_3 = c) \wedge (m_2 \neq -c) \wedge (p_1 \neq -c)) \vee \\
&((p_3 = c) \wedge (g_2 = c)) \vee \\
&((m_3 = c) \wedge (p_2 \neq -c)) \vee \\
&(g_3 = c)
\end{aligned}
\tag{G.1}
$$

G.2 5 Trit Lookahead

$$
\begin{aligned}
P = c \overset{def}{\iff} &((p_4 = c) \wedge (p_3 = c) \wedge (p_2 = c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((p_4 = c) \wedge (p_3 = c) \wedge (p_2 = c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((p_4 = c) \wedge (p_3 = c) \wedge (m_2 = c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((p_4 = c) \wedge (p_3 = c) \wedge (m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (g_2 \neq -c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (g_2 \neq -c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (m_2 \neq -c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (m_2 \neq -c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (p_2 = c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (p_2 = c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (m_2 = c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (m_2 = c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (g_2 \neq -c) \wedge (p_1 = c) \wedge (p_0 = c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (g_2 \neq -c) \wedge (m_1 = c) \wedge (g_0 \neq -c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (m_2 \neq -c) \wedge (g_1 \neq -c) \wedge (p_0 = c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (m_2 \neq -c) \wedge (m_1 \neq -c) \wedge (g_0 \neq -c)) \vee \\
&((p_4 = c) \wedge (p_3 = c) \wedge (p_2 = c) \wedge (g_1 = c)) \vee \\
&((p_4 = c) \wedge (p_3 = c) \wedge (m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (g_2 \neq -c) \wedge (g_1 = c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (m_2 \neq -c) \wedge (p_1 \neq -c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (p_2 = c) \wedge (g_1 = c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (m_2 = c) \wedge (p_1 \neq -c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (g_2 \neq -c) \wedge (g_1 = c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (m_2 \neq -c) \wedge (p_1 \neq -c)) \vee \\
&(p_4 = c) \wedge (p_3 = c) \wedge (g_2 = c)) \vee \\
&((p_4 = c) \wedge (m_3 = c) \wedge (p_2 \neq -c)) \vee \\
&((m_4 = c) \wedge (g_3 \neq -c) \wedge (g_2 = c)) \vee \\
&((m_4 = c) \wedge (m_3 \neq -c) \wedge (p_2 \neq -c)) \vee \\
&((p_4 = c) \wedge (g_3 = c)) \vee \\
&((m_4 = c) \wedge (p_3 \neq -c)) \vee \\
&(g_4 = c)
\end{aligned}
\tag{G.2}
$$

H

Components Implemented in MRCS

This chapter contains a list highlighting some of the components built as a part of this project. These components also contain some subcomponents that have not been explicitly mentioned, but are part of the files delivered with the project.

H.1 add-3-lookahead

This component is an implementation of a full 3 trit lookahead adder. It can not be connected to further nodes, as the P, M, G signals are not output.

H.2 add-3-3-lookahead

This component implements a 3 input 1 carry addition, with the same logic for lookahead as with 2 inputs. This is shown to be possible in Appendix F.

H.3 lookahead-unit-3-2

This is the full 3 trit lookahead node, implemented using the proposed in subsection 4.1.4. It can be connected to other nodes, to perform larger additions.

H.4 LookaheadUnit3-Simple

This component is an implementation of a full tree node with the assumption that g is always 0. This will be the case in the first node in a full carry-lookahead tree, where the sum of two numbers cannot be generated unconditionally, as explained in subsection 4.1.5.

H.5 partial-lookahead-3

This component implements the p, m, and g signals for the 3 inputs, as well as the sum for the 3 inputs and the carry input.

H.6 PartialFullAdder

This component implements the p and m signals for 2 inputs, as well as the sum for the 2 inputs and the carry input.

H.7 balanced-unbalanced-lookahead

This component implements balanced addition through two unbalanced lookahead adders. Because the adders are unbalanced, the logic for carry anticipation can be implemented with binary components and simpler logic. Jones [35] mentions how to perform balanced addition with unbalanced adders.

H.8 CarrySelect6

This component implements a 6 trit carry-select adder based on the optimisation proposed by Yoon, Baek, Kim *et al.* [36]. It consists of one 3 trit ripple-carry chain, two ripple-carry chains with different carry inputs.

H.9 hackaday-2-trit-adder

This component implements a 2 trit adder as proposed in [42]. It was unclear how the multiplexers were intended to be connected from the article, but it would still be non-functional no sums are affected by the addition of less significant trits.

H.10 CircularBufferAddr2

This component implements two 2-trit counters for use in a circular buffer. It is intended for use within the reorder buffer of a superscalar CPU.

H.11 CPUctrl

This component produces the required control signals to be sent to all multiplexers within the CPU to control the signals. These need to be routed through the buffers with the data, to ensure that the data and instructions move through the pipelines together.

H.12 ALUctrl

This component produces the control signals required for the ALU. It reads the relevant data from the instruction, and produces the signals that were shown in Section 4.2.

H.13 ALU2

This component is the full ALU that was shown in Section 4.2. It contains all the sub-components to perform the functionality of the ALU, and is intended to work together with ALUctrl to implement the REBEL-2 ISA.

H.14 Fetch

This component implements the first fetch stage, which includes a program counter and an instruction register. The register must be filled with instructions before the code can begin to execute.

H.15 Decode

This component contains an incomplete version of the decode stage. The register is not edge triggered, but the CPU control is included.