INTEGRATING RISC-V PROCESSOR with an FPGA

by

MANOJ KUMAR G.SIVAKUMAR

1191101460

Session 2023/2024

The project report is prepared for

Faculty of Engineering

Multimedia University

in partial fulfilment for

Bachelor of Engineering (Hons) Electronics

majoring in Computer

FACULTY OF ENGINEERING

MULTIMEDIA UNIVERSITY

JULY 2024

DECLARATION

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

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

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

Signature: ________________________

Name: MANOJ KUMAR G.SIVAKUMAR

Student ID: 1191101460

Date: 18 JUNE 2024

ACKNOWLEDGEMENT

I would like to thank my supervisor, Assoc. Prof. Ir. Dr. Lee Lini for the knowledge and guidance which helped me during the technical challenges that this thesis presented.

I would also like to thank the people from the r/FPGA and r/RISCV subreddit for their prompt responses, advice and the knowledge shared, which significantly contributed to the theoretical and technical aspects of this project in overcoming many challenges.

I am deeply thankful to my family members and the people that have been close to me all these years for their constant support and patience. Their love and belief in me have been a continuous source of strength.

Finally, I am grateful to the Robotics Club (MMU Cybertron), Faculty of Engineering, MMU Cyberjaya for providing the resources and environment necessary for this research.

ABSTRACT

This report presents the implementation of a RISC-V processor integrated into an FPGA platform. The NEORV32 processor, an open-source and versatile RISC-V core was chosen for its configurability and robustness.

The implementation process involved setting up a development environment on Ubuntu, configuring the core and uploading it to the FPGA. Key steps included are synthesizing the design, assigning pins and verifying the processor functionality through an application.

Results demonstrated the processor’s performance and efficiency by successfully running a holonomic X-drive mobile robot which requires high computational resource application on a single program which includes quadrature encoder, inverse kinematics and PID control which validated the processor’s capabilities to handle complex tasks.

The successful integration of the RISC-V processor on an FPGA offers a valuable resource for education on computer architecture as well as a flexible platform for future research. Further development could expand peripheral support and incorporate advanced features like neural networks for image processing tasks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
CFS	Custom Function Subsystem
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GPIO	General Purpose Input Output
IP Core	Intellectual Property Core
ISA	Instruction Set Architecture
PID	Proportional-Integral-Derivative
PWM	Pulse Width Modulation
RISC-V	Reduced Instruction Set Computing – 5th version
ROM	Read-Only Memory
SoC	System on Chip
TWI	Two-Wire Interface
UART	Universal Asynchronous Receiver and Transmitter
VHDL	VHSIC Hardware Description Language
WSL	Windows Subsystem for Linux

CHAPTER 1 INTRODUCTION

1.1 Overview

The integration of processors on Field-Programmable Gate Arrays (FPGAs) has been a significant area of research and development in the field of electronics and computer engineering. This project explores the integration of the RISC-V processor on an FPGA.

Traditional Instruction Set Architectures (ISAs) were frequently designed to prevent illegal use, which limited their use and adaptability. RISC-V, an open-source ISA, provides a more adaptive and collaborative environment. Despite its benefits, there are significant obstacles and limitations in understanding the implementation approaches for combining RISC-V processors with FPGA technology (Andrew Shell Waterman, 2016).

With the growing availability of open-source RISC-V cores on platforms like GitHub, there is a need for comprehensive literature and guidelines to aid in real-world implementation. This project aims to address these challenges by providing a detailed exploration of RISC-V and FPGA integration.

1.2 Problem Statements

Traditional ISAs, with their proprietary designs aimed at preventing unauthorized use, present significant barriers to developers seeking to innovate and customize these architectures for specific applications. This restrictive nature often necessitates acquiring licenses or permissions, hindering the flexibility and adaptation needed for rapid technological advancements.

Integrating RISC-V processors with FPGAs offers compelling advantages, yet numerous challenges persist. Gaps in understanding implementation methodologies remain, posing hurdles that must be bridged to streamline and optimize the integration process. This complexity limits accessibility and efficiency, potentially delaying the realization of RISC-V's full potential within FPGA environments.

Despite the numerous open-source RISC-V cores available on platforms like GitHub, the vast variety can overwhelm developers. Selecting the most suitable core for specific applications becomes challenging without clear guidelines or criteria, holding back efficient adoption and deployment.

Furthermore, there is an obvious absence of comprehensive literature and practical guidelines for implementing RISC-V cores on FPGAs in real-world scenarios. The lack of detailed documentation and case studies undermines the ability of engineers and researchers to effectively harness these resources, holding back innovation and progress in the field. Addressing these gaps is crucial for unlocking the broader adoption and utilization of RISC-V in FPGA-based systems.

1.3 Project Scope

This project involves the selection and integration of a suitable open-source RISC-V core onto an FPGA platform. The scope includes:

- Evaluating RISC-V cores that are available on GitHub.
- Understanding the implementation methodologies for integrating RISC-V cores with FPGA hardware.
- Developing a framework or set of guidelines to assist in the practical implementation of RISC-V on FPGAs.
- Testing and validating the integrated system to ensure functionality and performance.

1.4 Objective

The primary objective of this project is to integrate a RISC-V processor onto an FPGA platform. This integration aims to show the configurability and robustness of the open-source processor. By successfully implementing this processor on an FPGA, the project will continue to demonstrate its capability to handle both simple and complex tasks to validate its performance and reliability. Additionally, this project also aims to explore the educational and research potential of the use of RISC-V processors within the FGPA environment.

1.5 Report Outline

This report is structured to provide a comprehensive understanding of the integration of RISC-V processors on FPGAs. It begins with the theory behind the RISC-V processor to better understand its overall architecture and available extensions. Following this theoretical foundation, the report transitions into the practical application of integrating a RISC-V processor on an FPGA. Detailed implementation steps are provided, highlighting the challenges encountered and the solutions developed to address these challenges.

This structure ensures that readers gain both the necessary background knowledge and practical insights, enabling a thorough comprehension of the subject matter.

CHAPTER 2 LITERATURE REVIEW

The RISC-V architecture, an open-source instruction set architecture (ISA) that has been gaining significant traction in both academic and in the industry. Its simplicity, modularity, and extensibility make it highly suitable for educational purposes and research initiatives. Unlike proprietary ISAs from Intel, AMD, IBM and ARM, RISC-V allows for extensive customization and without licensing costs. RISC-V stands out in this context due to its flexibility and open-source nature, making it an ideal focus for educational and research applications. This has led to a growing interest in the study of computer architecture, particularly through practical implementations on Field Programmable Gate Arrays (FPGAs).

2.1 Open-Sourced RISC-V Processor IP Cores for FPGA: Overview and Evaluation

Open-sourced RISC-V processor IP cores have gained some popularity due to their flexibility, customizability and the cost-effectiveness. These cores are implemented on FGPAs which provides an excellent platform for configurability as well as educational purposes. The paper provides an overview of various open-sourced RISC-V processor IP cores and evaluates their performance and suitability for different applications (Höller, 2019).

Among the most notable cores are PicoRV32, Rocket and Pulpino where each of the core have unique characteristics and performance.

- PicoRV32: This core is known for its minimal resource usage where it is designed to be compact for efficient resource-constrained environments. It is ideal for application specific which requires low power consumption and small footprints. The PicoRV32 supports the RV32IMC instruction and a configurable.
- Rocket: The Rocket core was developed as part of the Berkeley RISC-V project, which the core offers high performance and more features compared to

PicoRV32. The core also supports a broader range of RISC-V instructions and cache subsystems making it more suitable for complex applications that require high computational power. The downside it is harder to implement for new students who wish to use the core.

- Pulpino: This core is designed for low-power applications like picoRV32. Pulpino is part of the PULP (Parallel Ultra Low Power) where it integrates several power-saving techniques. Pulpino supports the RV32IMC instruction sets similarly to PicoRV32. The targeted area for applications is the Internet of Things (IoT) and wearable devices.

These open-source RISC-V cores are valuable in education and research, where PicoRV32 and Pulpino have simple and energy-efficient designs which are excellent for introductory to learn the fundamental concepts of computer architecture without the complexity of advanced features. The Rocket core on the other hand with its high performance is suitable for research projects that explore new architectural and optimization techniques.

In conclusion, the open-source RISC-V processor provides a rich set of tools for educational and research purposes. Their adaptability and cost-effectiveness make them a very attractive alternative to proprietary cores. The evaluations of several cores highlight these cores' respective strengths and trade-offs according to their optimal use cases.

2.2 FPGA Implementation of Educational RISC-V Processor Suitable for Embedded Applications

In the paper "FPGA implementation of educational RISC-V Processor for Embedded Applications", the authors (Saif et al, 2023) provide a detailed process of designing and implementing a RISC-V Processor that is suitable for educational purposes and can be uploaded to an FPGA board. The goal is to create a processor that is both accessible for teaching use and able to handle embedded applications.

The processor's design and implementation process are thoroughly documented which provides the students with clear guidance step. This documentation includes an explanation of the VHDL code to synthesizing the core. The architecture includes the basic core components such as ALU, register file and a memory interface. The simplicity of the design ensures that the students are able to study and learn the fundamental concepts of a processor while providing enough practical application. While for practical applications, involve the implementation of the RISC-V processor on ans FPGA board which allows students to engage in a range of learning curves from modifying the hardware configuration to running programs on the processor. This hands-on approach not only reinforces theoretical knowledge but also enhances practical skills in digital design and embedded systems.

In conclusion, the FGPA implementation of an educational RISC-V processor provides a valuable resource for teaching computer architecture and embedded applications. the processor's design offers a balance between simplicity and functionality, making it an effective educational tool that can be adapted and extended to meet various instructional needs.

2.3 RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education

The "RVfpga" project, as detailed in the paper "RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education," aims to provide a comprehensive educational platform based on the RISC-V architecture (Harris et al, 2021). This project leverages the open-source nature of RISC-V to create an accessible and configurable environment for teaching computer architecture.

One of the significant advantages that was highlighted in this paper was the configurability of the RISC-V with an FPGA. Unlike fixed hardware, FPGA can be reprogrammed to accommodate different designs and experiments making them an ideal tool for education. This feature allows us to test the core designs in real-time and able to observe the outcomes to make any necessary adjustments. Thus, making it ideal for learning through direct experiments.

The RVfpga project also emphasizes ease of use and accessibility. It includes detailed documentation, tutorials, and educational materials that guide students and instructors through the learning process. These resources cover a range of topics, from basic RISC-V concepts to advanced FPGA programming techniques, ensuring that users at all skill levels can benefit from the platform (Harris et al, 2021).

In conclusion, the RVfpga project represents a significant advancement in computer architecture education. By combining the open-source RISC-V ISA with the flexibility of FPGAs, it provides a powerful and versatile platform for learning.

2.4 The MicroRV32 Framework: An accessible and configurable Open-Source RISC-V Cross-Level Platform for education and research

The MicroRV32 framework presents another contribution to the field of RISC-V-based educational tools. As described in "The MicroRV32 Framework: An Accessible and Configurable Open-Source RISC-V Cross-Level Platform for Education and Research" this framework aims to bridge the gap between high-level theoretical concepts and low-level practical implementation (Ahmadi-Pour, 2022).

The MicroRV32 framework is designed to be highly modular and flexible allowing users to configure the system to their specific needs. The framework includes a RISC-V processor core which is implemented in such a way that it is easier to understand the fundamentals of computer architecture. The framework also supports a broad range of configurations which enables the user to explore various aspects of processor design from basic instruction sets to more complex features like memory hierarchies.

For education, the MicroRV32 framework serves as a powerful teaching tool which allows students to gain hands-on experience with RISC-V architecture. With that, the gap between the theoretical and practical knowledge can be bridged. While for researchers, this framework can provide them with a robust platform to explore new processor designs and architecture. Since it is open source, researchers can freely modify and extend the framework.

The performance of the MicroRV32 framework is evaluated based on several criteria including resource utilization, operational speed, and scalability. The framework is designed to be efficient to be used in FPGA, making it suitable for deployment on a variety of FPGA platforms. Its operational speed is sufficient for most educational and research applications, providing a responsive environment for experimentation and learning.

In conclusion, the MicroRV32 ensures that it can meet the needs of a diverse user base, from students who just beginning their journey in computer architecture to researchers expanding the processor designs for more advanced and efficient computing. By providing practical hands-on tools for learning and experimentation to better understand computer architecture and encourage innovation in this field.

2.5 Implementation Methodologies

The implementation of RISC-V processors on FPGAs involves various methodologies, each with its own set of challenges and benefits. The literature indicates a notable gap in comprehensive guidelines and practical documentation, which hampers the effective utilization of RISC-V cores on FPGAs. This gap is evident in the abundance of open-source RISC-V cores available on platforms like GitHub, making it overwhelming for developers to select the appropriate core for specific applications without proper guidelines.

2.6 Challenges in Integration

Despite the clear advantages of integrating RISC-V with FPGAs, significant challenges exist. These challenges include understanding the implementation methodologies, bridging gaps in practical guidelines, and addressing the lack of detailed documentation and case studies. These obstacles limit the ability of engineers and researchers to effectively utilize RISC-V cores on FPGA platforms.

2.7 Summary

The integration of RISC-V processors into FPGA-based educational tools, as explored in these studies, represents a crucial step in bridging the gap between theoretical knowledge and practical skills. This approach not only enriches the learning experience but also prepares students for real-world applications in embedded systems and beyond. The hands-on experience facilitated by FPGA implementations, combined with the flexibility of RISC-V, offers a robust platform for both education and research in computer architecture.

CHAPTER 3 NEORV32 PROCESSOR (SoC)

The NEORV32 RISC-V Processor is a fully customizable microcontroller-like SoC that was built around the RISC-V architecture. It is made available under an open-sourced license by the author, snotling in GitHub.

3.1 SoC Block Diagram

The NEORV32 block diagram provides a comprehensive overview of the available components and their interactions within the processor. This SoC integrates several peripherals and memory interfaces, offering a full-scale microcontroller-like environment based on the RISC-V architecture (Nolting, 2020).

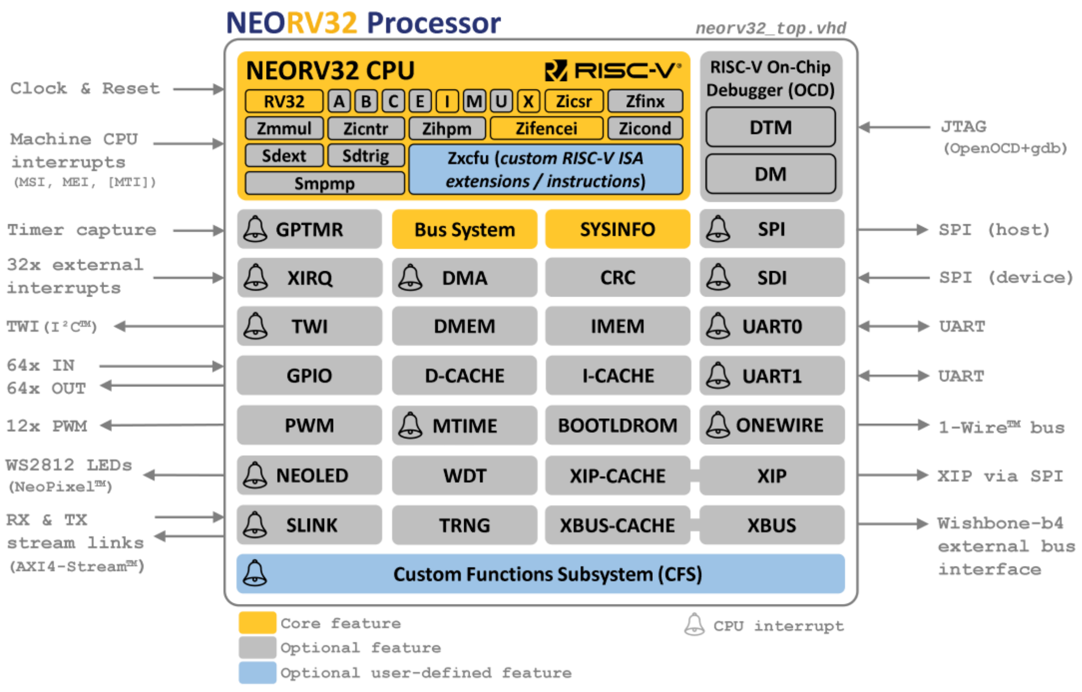


Figure 3. 1: SoC Block Diagram

Some of the key components of the NEORV32 SoC include:

1. CPU Core: the Central Processing Unit (CPU) which handles the primary computational tasks.

2. Internal Memories: The system includes optional processor-internal data (DMEM) and Instruction Memory (IMEM) for storing program code and data directly accessible by the CPU.
3. Caches: Optional Instruction Cache (iCACHE) and Data Cache (dCACHE) to improve performance by reducing the time needed to access the frequently used instructions and data.
4. Boot ROM: An internal bootloader ROM to initialize and provide a UART console.
5. Peripherals Interface: Such as UART, PWM, GPIO etc for communication and digital I/O tasks.

3.2 Processor Top Entity (Signals)

The processor's top entity for signals describes the overall signals with the NEORV32 processor core. This top entity for signals is crucial for proper communication between the processor and the environment modules to ensure the functionality within the system.

The signal can be categorized into 2 environments, Global control, and local variables.

Table 3. 1: Global Processor Entity

Global Control	
Name	Description
Clk_i	Input clock signal, main clock source for the processor where it will trigger on every rising edge.
rstn_i	Global reset, this signal will reset the processor and it is an active-low reset signal where it can be assigned to a button

Table 3. 2: UART Signals

Universal Asynchronous Receiver and Transmitter (UART)	
Name	Description
uart0_txd_0	Serial transmission to send data from the processor to an external device
uart0_rxd_0	Serial receiver to receive data to the processor from an external device
uart1_txd_0	UART1 transmit data output (optional). Like UART0, used for transmitting data
uart1_rxd_0	UART1 receive data input (optional). Like UART0, used for receiving data

The UART0 and UART1 signals provide the communication capability to different external devices which is suitable for various embedded applications. These signals are essential for a larger system and debugging purposes.

Table 3. 3: GPIO Signals

General Purpose Input and Output (GPIO)	
Name	Description
gpio_o	General purpose output signals.
gpio_i	General purpose input signals.

The GPIO ports enable the capabilities to read and write digital signals from and to external devices such as sensors, motor drivers and servos. Overall, GPIO pins are an essential feature in embedded applications.

Table 3. 4: Custom Function Subsystem (CFS) Signals

Custom Function Subsystem (CFS)	
Name	Description
cfs_in_i	Custom function input signal. Used to provide input to user-defined custom functions.
cfs_out_o	Custom function output signal. Used to output results from user-defined custom functions.

The Custom Function Subsystem (CFS) allows the integration of user-defined functionalities, which makes the processor highly versatile for various applications and able to perform parallel processing.

Table 3. 5: PWM Signal

Pulse-Width Modulation Controller (PWM)	
Name	Description
pwm_o	Pulse-width modulation channels

The Pulse-Width Modulation (PWM) allows the control of the amount of power delivered to an electrical device, commonly used in motor control and signal generation.

Table 3. 6: TWI Signals

Two-Wire Serial Interface Controller (TWI)	
Name	Description
twi_sda_i	serial data line sense input
twi_sda_o	serial data line output (pull low only)
twi_scl_i	serial clock line sense input
twi_scl_o	serial clock line output (pull low only)

The TWI signals are used for communication with I2C-compatible devices. These signals ensure the processor can interact with peripherals and external systems.

3.3 Processor Top Entity (Generics)

The processor Top Entity generics section defines all the configurable parameters to configure the system according to the intended specification of application requirement. The generics will control certain CPU extensions and peripherals modules and can be used to optimize the CPU depending on the application based on maximum performance or power consumption.

Table 3. 7: General Generics

General Generics	
Name	Description
CLOCK_FREQUENCY	This generic defines the frequency of the input clock signal (clk_i). It is used to configure various timing-related functionalities within the processor
INT_BOOTLOADER_EN	This generic activates the internal bootloader, which allows the system initialization and boot executable

General generics are the most important and the highest priority parameters that need to be configured. If the clock frequency does not match with the FPGA board's external oscillator, the overall operational speed and functionality of the processor operation will be affected.

Table 3. 8: Data Memory Configuration

Data Memory Configuration	
Name	Description
MEM_INT_DMEM_EN	Implement the internal processor data memory
MEM_INT_DMEM_SIZE	Size of each data internal memory blocks in bytes

The data memory configuration outlines how the internal memory is defined in the processor. It is essential for storing and accessing data efficiently during execution. Enabling and specifying the size will influence the total storage capacity available for program data and variables.

Table 3. 9: Cache Memory Configuration

Cache Configuration	
Name	Description
ICACHE_EN	Implement instruction cache
ICACHE_NUM_BLOCKS	Number of blocks
ICACHE_BLOCK_SIZE	Size of each cache memory blocks in bytes

The cache memory parameter enables the implementation of instruction cache in the core. The instruction cache will store frequently accessed instruction from the main memory which results in reducing the accessing latency, hence increasing the performance of the core.

Table 3. 10: Data Cache Configuration

Data Cache Configuration	
Name	Description
DCACHE_EN	Implement data cache
DCACHE_NUM_BLOCKS	Number of data cache blocks
DCACHE_BLOCK_SIZE	Size of each data cache memory blocks in bytes

The implementation of the data cache feature will improve memory latency access time by storing frequently accessed data closer to the processor core. Configuring these parameters will improve the performance of the core particularly in scenarios where frequent data access is used, such as high-performance computing tasks or real-time processing applications.

Table 3. 11: Tuning Configuration

CPU Architecture Tuning Configuration	
Name	Description
FAST_MUL_EN	Implement parallel multipliers
FAST_SHIFT_EN	Implement full barrel shifters
REGFILE_HW_RST	Implement hardware reset for register to prevent interruption of BRAM

By enabling this generics, overall multiplication operations can be accelerated as well as increasing the efficiency of bit manipulation at the cost of additional logic resources.

3.4 Processor Clocking

The processor is designed to operate as a fully synchronous logic design using a single clock domain from an external clock signal 'clk_i' (Nolting, 2020). This clocking is essential to ensure that all internal modules and components used by memories and internal registers operate in a synchronous manner to enhance the reliability of the processor.

The internal memories and registers are triggered on the rising edge of the input clock signal, 'clk_i'. While the processor reset is triggered on the falling edge of the clock signal 'clk_i' which ensures proper clock management.

Besides the clock domain and triggering components, the clock signal is also used for peripheral clocks. Where the clock frequency defined by the CLOCK_FREQUENCY generic is used to generate many modules like UART and other timer-based operations.

From the main clock, 8 sub-main clock signals can be used by configuring the programmable three-bit prescaler select in their control register. The mapping of the select bits of prescaler to its resulting clocks is shown in the table below.

Table 3. 12: Prescalar Bit Select

Prescalar bits	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting clock	f/2	f/4	f/8	f/64	f/128	f/1024	f/2048	f/4096

It is important to understand the clocking mechanism and management to design and work with the processor, knowing the clocking system allows designers to make decisions on how to implement efficient and optimize code for different applications needs for example parallel processing using the Custom Function Subsystem (CFS).

In summary, the processor clocking management in a synchronous processor is important and ensures all the components within the component work together properly.

3.5 Bus System

The NEORV32 processor incorporates a bus system that facilitates the communication between the processor core and various peripheral I/O devices, memory modules and external bus interfaces (Nolting, 2020).

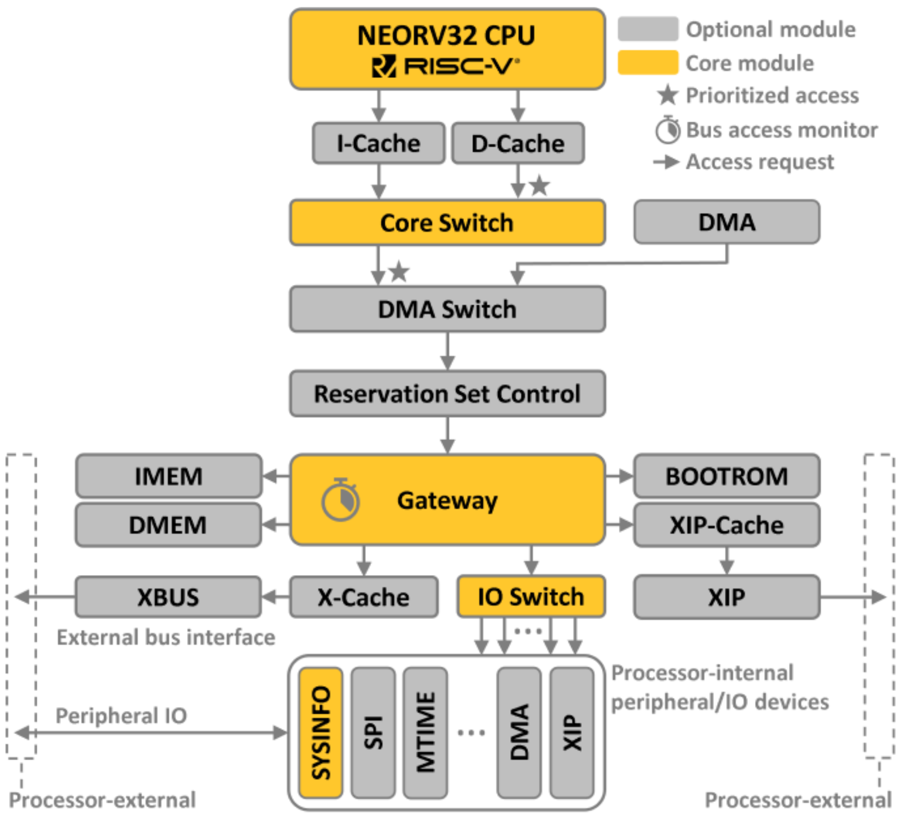


Figure 3. 2: Processor Bus Architecture

Key core components of the Bus System:

1. NEORV32 CPU: The CPU executes the instructions and processes the data. The process data will interact with the memory and peripherals through the bus system
2. Core Switch: This component manages the routing of data between the I-Cache and D-Cache, ensuring that data is prioritized and reaches the correct destination

3. Bus Gateway: the bus gateway serves the communication between the different bus systems and redirects the core accesses according to respective modules as well as monitors all bus transactions. It acts like a bridge between various communication protocols.
4. IO Switch: This switch connects various I/O peripherals and handles the data flow between external devices and the core
5. SYSINFO: The system information module provides critical information on the processor settings that are related to processor configuration. The NEORV32 runtime requires this data to perform the correct operations

3.6 Boot Configuration

The NEORV32 processor provides several boot configurations that can be selected in the top file of the core. Two possible boot scenarios are indirect and direct boot, which can be configured via the ‘INT_BOOTLOADER_EN’ generic in the top file of the core.

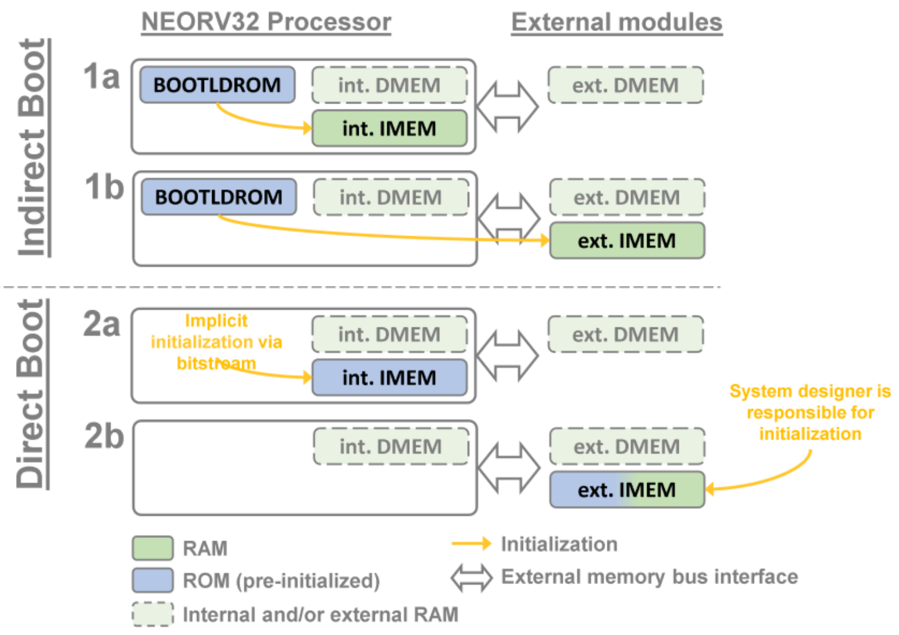


Figure 3. 3: Boot Configurations

3.6.1 Direct boot

When the ‘INT_BOOTLOADER_EN’ is set to false, the direct boot will be selected, and the program hex code will be loaded to the core during the synthesize stage of the FPGA and uploaded via the bitstream. This boot is implemented into the processor’s internal memory, IMEM as ROM and it provides non-volatile storage of the program in the processor (Nolting, 2020). The processor will directly start to execute the code from the beginning of the instruction address space after reset.

3.6.2 Indirect boot

When the ‘INT_BOOTLOADER_EN’ is set to true the indirect boot will be selected. The core will be synthesized with the processor’s internal Bootloader ROM. The bootloader will provide several options to upload the executable binary file either via UART or SPI. The bootloader will be communicating with the computer via primary UART (UART0) using a USB-to-UART Converter.

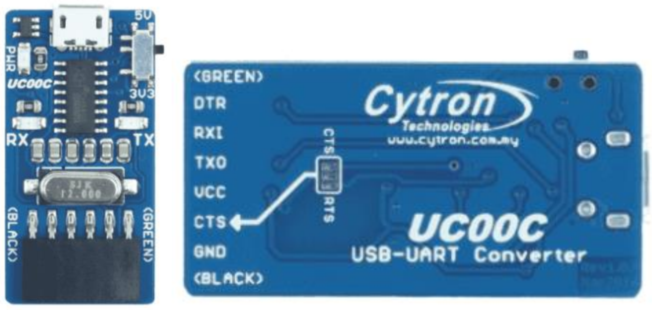
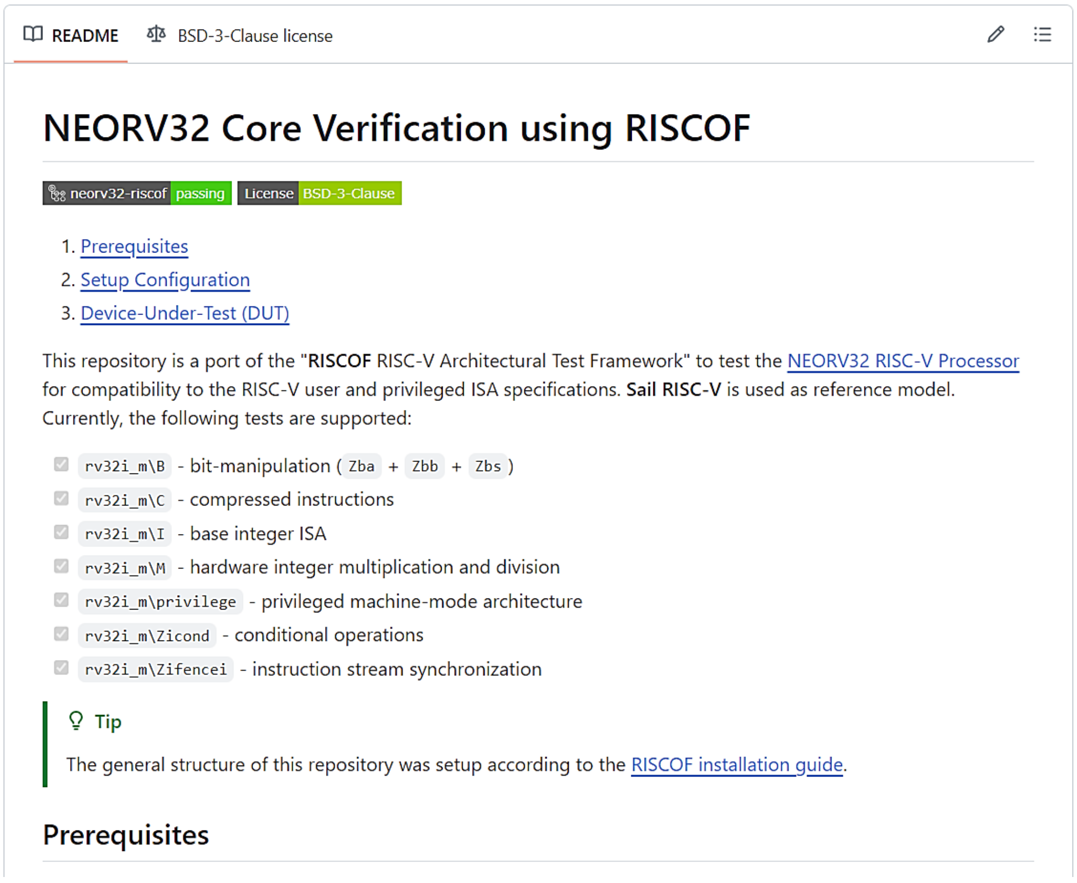


Figure 3. 4: USB-to-UART Converter

3.7 RISC-V Compatibility

The NEORV32 CPU passes the tests of RISCOF, a RISC-V Architecture compatibility/compliance framework that can test the RISC-V implementation against a reference standard RISC-V model using a suite of assembly tests (pawks et al, 2020).



README   BSD-3-Clause license

NEORV32 Core Verification using RISCOF

neorv32-riscof passing License BSD-3-Clause

1. Prerequisites
2. Setup Configuration
3. Device-Under-Test (DUT)

This repository is a port of the "**RISCOF** RISC-V Architectural Test Framework" to test the NEORV32 RISC-V Processor for compatibility to the RISC-V user and privileged ISA specifications. **Sail RISC-V** is used as reference model. Currently, the following tests are supported:

- [x] `rv32i_m\B` - bit-manipulation (`Zba` + `Zbb` + `Zbs`)
- [x] `rv32i_m\C` - compressed instructions
- [x] `rv32i_m\I` - base integer ISA
- [x] `rv32i_m\M` - hardware integer multiplication and division
- [x] `rv32i_m\privilege` - privileged machine-mode architecture
- [x] `rv32i_m\Zicond` - conditional operations
- [x] `rv32i_m\Zifencei` - instruction stream synchronization

Tip

The general structure of this repository was setup according to the RISCOF installation guide.

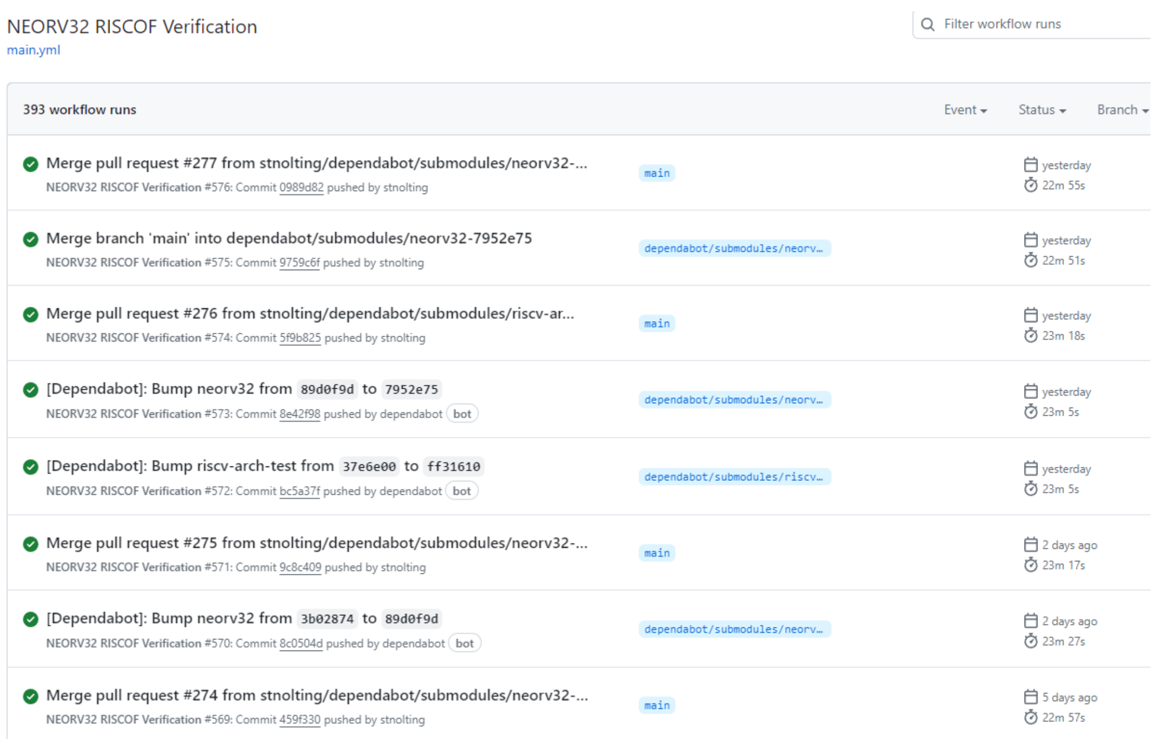
Prerequisites

Figure 3. 5: NEORV32 Core Verification

The NeoRV32 CPU passes several tests from different aspects of the RISC-V ISA. These include bit-manipulation instructions, compressed instructions, base integer instructions, hardware integer multiplication and division as well as conditional operations and instruction stream synchronization.

The process of verification involves a detailed setup as outlined in the RISCOF installation guide. The repository setup, configuration and the Device-Under-Test (DUT) are defined to ensure accurate testing conditions to provide reliable results.

In the verification status as shown in Figure 3.6, the NeoRV32 has undergone several successful workflows runs and it indicate continuous integration and consistent compliance with the RISC-V standards (stnolting, 2020).


NEORV32 RISCOF Verification
main.yml

Filter workflow runs

393 workflow runs | Event | Status | Branch

Merge pull request #277 from stnolting/dependabot/submodules/neorv32-...
NEORV32 RISCOF Verification #576: Commit 0989d82 pushed by stnolting
main
yesterday
22m 55s

Merge branch 'main' into dependabot/submodules/neorv32-7952e75
NEORV32 RISCOF Verification #575: Commit 9759c6f pushed by stnolting
dependabot/submodules/neorv…
yesterday
22m 51s

Merge pull request #276 from stnolting/dependabot/submodules/riscv-ar...
NEORV32 RISCOF Verification #574: Commit 5f9b825 pushed by stnolting
main
yesterday
23m 18s

[Dependabot]: Bump neorv32 from 89d0f9d to 7952e75
NEORV32 RISCOF Verification #573: Commit 8e42f98 pushed by dependabot bot
dependabot/submodules/neorv…
yesterday
23m 5s

[Dependabot]: Bump riscv-arch-test from 37e6e00 to ff31610
NEORV32 RISCOF Verification #572: Commit bc5a37f pushed by dependabot bot
dependabot/submodules/riscv…
yesterday
23m 5s

Merge pull request #275 from stnolting/dependabot/submodules/neorv32-...
NEORV32 RISCOF Verification #571: Commit 9c8c409 pushed by stnolting
main
2 days ago
23m 17s

[Dependabot]: Bump neorv32 from 3b02874 to 89d0f9d
NEORV32 RISCOF Verification #570: Commit 8c0504d pushed by dependabot bot
dependabot/submodules/neorv…
2 days ago
23m 27s

Merge pull request #274 from stnolting/dependabot/submodules/neorv32-...
NEORV32 RISCOF Verification #569: Commit 459f330 pushed by stnolting
main
5 days ago
22m 57s

Figure 3. 6: NEORV32 Verification Status

The consistent passing status of the verification signifies that the core is a reliable and compliant implementation of the RISC-V architecture.

3.8 CPU Architecture

The CPU uses a pipelined multi-cycle architecture where each instruction is broken down into a sequence of micro-operations for execution. To boost performance, the CPU's front-end (which fetches instructions) and back-end (which executes instructions) are separated by a FIFO instruction prefetch buffer. This allows the front end to continue fetching new instructions even while the back-end is still executing the previously fetched ones (stnolting, 2020).

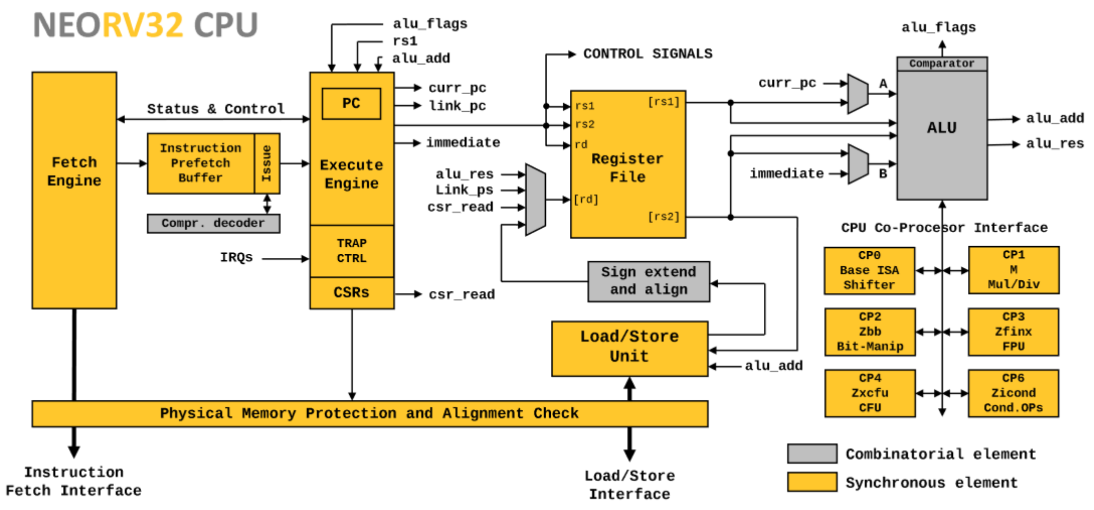


Figure 3. 7: NEORV32 CPU Architecture

The NEORV32 processor fully supports the RV31 base integer instruction set of the RISC-V architecture, which includes 32-bit instructions for basic arithmetic, logic, control flow, and memory access operations. Additionally, it supports several standard RISC-V extensions such as the Multiply (M), Atomic (A), and Compressed (C) instruction sets, enhancing its capabilities.

Besides that, author architecture follows the Harvard Architecture where the buses (signal path) for instruction and do not share the same buses (Nihal Kularatna, 2000). This enables the CPU to perform read instruction and read/write data simultaneously. This separation results in better performance and reduced bottlenecks.

The architecture of the NEORV32 processor is balanced between performance, area and power consumption.

3.9 Instruction Sets and Extensions

The NEORV32 CPU provides multiple optional RISC-V and custom ISA extensions (stnolting, 2020) which can be enabled or disabled via Processor Top Entity Generics (Section 3.3). Each of the extensions adds additional instruction sets and it is important to properly understand every one of them to add functionality (Andrew,

2016) to an intended application or increasing the performance of the processor or reducing the overall architecture size.

- A ISA Extension

The A-extension also known as Atomic Extension in the NEORV32 processor only offers load-reserved/store-conditional instructions, while the read-modify-write instructions are not supported but can be emulated using ‘lr’ and ‘sc’ operations.

Atomic instruction is crucial for notifying an application if a specific memory location has been altered by another process or a DMA access. This notification is essential for the synchronization mechanism.

- B ISA Extension

The B-extension adds the bit-manipulation instructions to the NEORV32 processor, implemented as a multi-cycle Arithmetic Logic Unit (ALU) co-process. This extension include several sub-extensions which includes Zba (address generation instructions), zbb (basic bit-manipulation instructions) and Zbs (single-bit instructions).

- C ISA Extension

The C-extensions or “Compressed” extension offers 16-bit encodings for frequently used instructions, which helps to reduce the overall code size.

- E ISA Extension

The E-extension or Embedded extension reduces the general-purpose register from 32 to 16 entries, decreasing the hardware size. While the overall size is reduced it still provides the same instructions as the base I ISA extension.

Note: Since the register file is reduced, the toolchain needs to be changed to “ilp32e”

- I ISA Extension

The I-extension is the base RISC-V integer ISA that is always enabled by default which runs basic operations like add, subtract, jump, load, store etc.

- M ISA Extension

The M-extension is a hardware-accelerated integer multiplication and division operation. This extension is implemented as a multi-cycle ALU co-processor. The operations can be further accelerated with a cost of additional logic resources by enabling the “FAST_MUL_EN”.

- U ISA Extension

The U-extension also known as the User-mode extension adds a second less-privilege operation mode on top of the highest-privilege machine code. In user-mode the code will have limited access to the CSRs and the address space for instance peripherals devices will be restricted. If there are any violations, an exception will trigger.

- X ISA Extension

The most crucial extension of the NEORV32 is where the CPU provides 16 fast interrupts (FIRQ), which are managed through custom bits in the CSRs. These extensions are mapped to CSR bits, following the RISC-V specs they are available for custom use.

- Zifencei ISA Extension

Zifencei extension includes the Instruction-Fetch Fence instructions which allows explicit synchronization between write and instruction fetch at the same time to an instruction memory.

- Zfinx ISA Extension

The Zfinx extension is a floating-point extension compared to the standard F ISA extension. The extension uses a register file to store and work on the floating-point data instead of a separate register file for the floating-point. This approach will significantly increase the speed of data change and require fewer hardware resources since it uses the same register file.

- Zicntr ISA Extension

Zicntr extension is used for base counters and timers. The extension consists of 2 counters which are CYCLE and INSTRET in the Control and Status Register (CSR). For NeoRV32 the mode TIME is not implemented (stnolting, 2020). All in all, the RISC-V foundation has stated that this extension must be mandatory to comply with the RISC-V specifications.

- Zicond ISA Extension

The Zicond ISA extension implements the branch-less control flows where the extension adds integer conditional, and it is being implemented as a multi-cycle ALU process.

- Zihpm ISA Extension

Zihpm extension provides hardware performance monitoring where it can be used to benchmark applications for example CoreMark.

- Zmnul ISA Extension

Zmnul extension is a sub-extension from the M ISA extension, where it implements only the multiplication. The main objective for this extension is for a minimal-size setup where the hardware does not have to operate divisions. This extension will be useful for applications that do not require division or intended smaller FPGA boards.

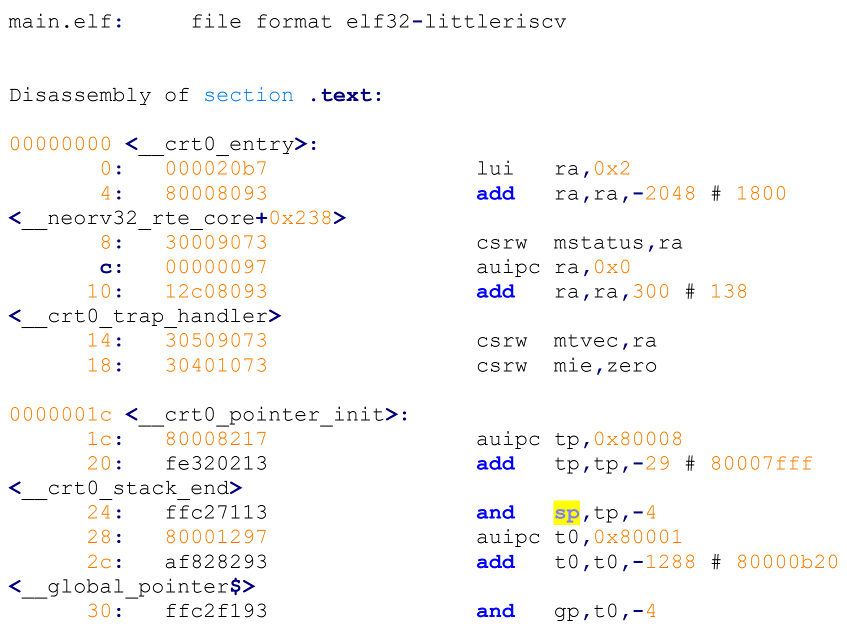
- Zxcfu ISA Extension

The Zxcfu extension is a NEORV32-specific ISA extension where it adds the Custom Function Unit (CFU) to allow the user to add custom RISC-V instructions to the processor (stnolting, 2020). The implemented instructions will run in parallel and tightly coupled to the processor's core

3.10 CPU Register File

The CPU Register file is another crucial component of the NEORV32 processor where it acts as a processor's storage data (stnolting, 2020). The processor can quickly read and write the instructions to the register to execute instructions. The register file contains 31 general-purpose registers (x1 to x31) used in arithmetic operations and storage, and 1 zero register (x0) which always reads as zero and has no effect if there is any write on to it.

"Read-during-write" behaviour should not be considered as the read and write access are mutually exclusive where it happens in separate cycles. Besides the read and write, up to 4 register operands are allowed to be fetched which allows synchronous read ports.



```
main.elf:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <__crt0_entry>:
       0:   000020b7                lui   ra,0x2
       4:   80008093                add   ra,ra,-2048 # 1800
<__neorv32_rte_core+0x238>
       8:   30009073                csrw  mstatus,ra
       c:   00000097                auipc ra,0x0
      10:   12c08093                add   ra,ra,300 # 138
<__crt0_trap_handler>
      14:   30509073                csrw  mtvec,ra
      18:   30401073                csrw  mie,zero

0000001c <__crt0_pointer_init>:
      1c:   80008217                auipc tp,0x80008
      20:   fe320213                add   tp,tp,-29 # 80007fff
<__crt0_stack_end>
      24:   ffc27113                and   sp,tp,-4
      28:   80001297                auipc t0,0x80001
      2c:   af828293                add   t0,t0,-1288 # 80000b20
<__global_pointer$>
      30:   ffc2f193                and   gp,t0,-4
```

Figure 3. 8 Assembly dump file

Figure 3.8 dump file snippet of the assembly language shows some of the operands used after the C-code being converted to assembly language.

3.11 CPU Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) handles both data processing and memory address. Basic computational instructions in the simple I ISA extension, for instance, addition and logical operations are executed using combinational logic that only requires a single cycle to complete. For more complex instructions like shifts and multiplications are handled by the ALU co-processor (stnolting, 2020).

The co-processors are designed as iterative units, which take multiple cycles to complete a task. In addition to handling the base ISA's shift instructions, the co-processors are responsible for executing all other processing-based ISA extensions, for example, M and B extensions.

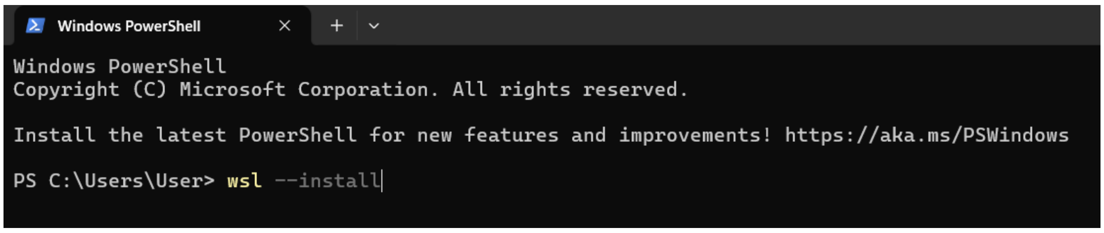
CHAPTER 4 GENERAL SOFTWARE FRAMEWORK

In this chapter, the detailed implementation of the software setup is described. This includes the steps and processes involved in setting up the necessary software frameworks including Linux environment and software design. The goal is to provide a comprehensive guide to ensure that the development environment is properly configured and ready for efficient project development.

4.1 Installing Ubuntu on Windows (WSL)

Windows Subsystem for Linux (WSL) allows users to run a Linux distribution alongside with Windows which provides a seamless environment. WSL will be used to install and run a RISC-V toolchain which is only supported on Linux environments.

Executing the “wsl – – install” command on Windows PowerShell allows to install all the WSL dependencies and features necessary to run Linux as well as install Ubuntu on the system by default (Craigloewen, 2023).



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\User> wsl --install
```

Figure 4. 1: WSL Installation

After the successful installation of WSL and Ubuntu, users will be prompted to create a new user account. Upon successful installation, it is essential to update and upgrade the Ubuntu package.

4.2 Software Toolchain Setup

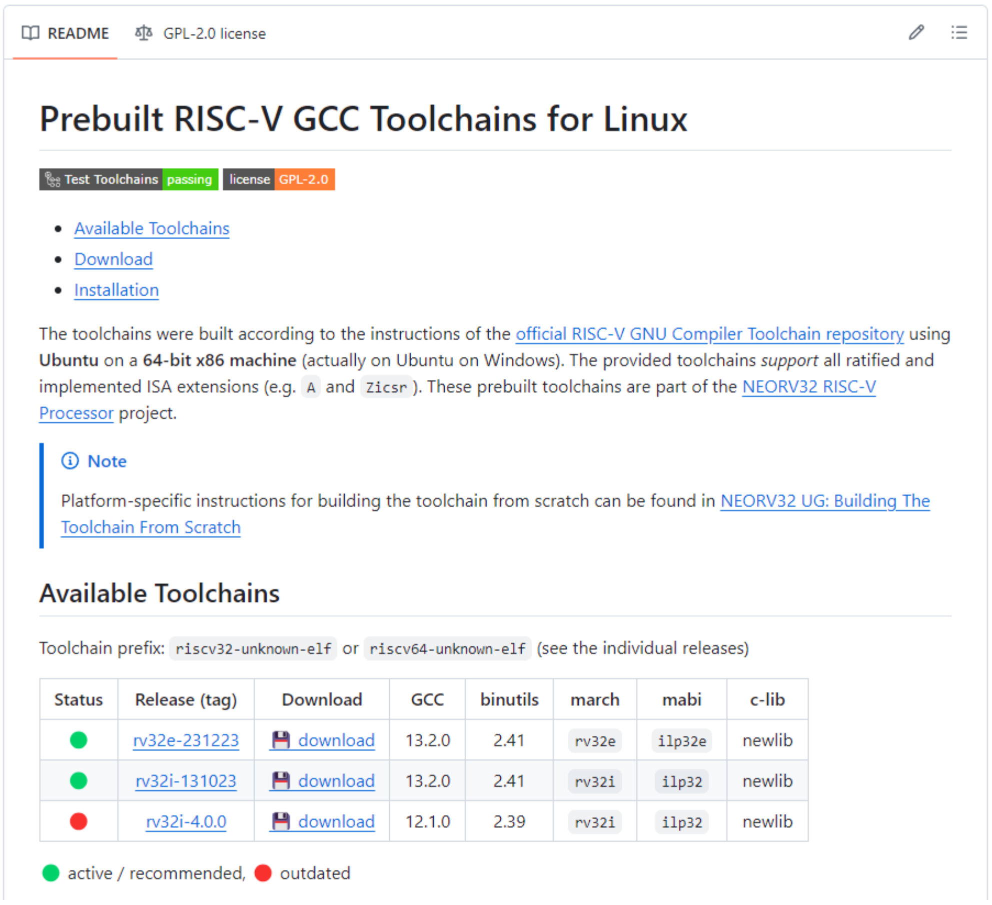
A RISC-V toolchain is needed to create the assembly instructions and executable files for the RISC-V core based on the C/C++ code. The toolchain consists of an assembler, compiler, linker as well as a debugger. This simplifies the development tasks.

The toolchain can be downloaded and built from scratch from the official GitHub page. The only downside it takes 6.65GB to clone the entire toolchain and additional time to build it (stnolting, 2020).

Alternatively, a pre-built RISC-V GCC toolchain can be downloaded as a single zip file and can be extracted. This approach can save up a lot of resources as well as time. Another advantage of using a pre-build toolchain is that the developer links all the libraries and external dependencies used by the selected core through a linker.

The compiler will translate the high-level C-code to a RISC-V assembly and optimize as well as improve the output assembly code. After compiling, the toolchain will start the assembler which takes the RISC-V assembly file and further translates into machine code. Finally, a linker will be used to combine the assembler file into a single executable file and at the same time incorporating the libraries used. The final output file can be loaded into the RISC-V core (Andrew, 2016).

It is important to understand the toolchain to better understand the components that work together to write an efficient code.



Prebuilt RISC-V GCC Toolchains for Linux

- Available Toolchains
- Download
- Installation

The toolchains were built according to the instructions of the official RISC-V GNU Compiler Toolchain repository using **Ubuntu** on a **64-bit x86 machine** (actually on Ubuntu on Windows). The provided toolchains *support* all ratified and implemented ISA extensions (e.g. `A` and `Zicsr`). These prebuilt toolchains are part of the NEORV32 RISC-V Processor project.

Note

Platform-specific instructions for building the toolchain from scratch can be found in NEORV32 UG: Building The Toolchain From Scratch

Available Toolchains

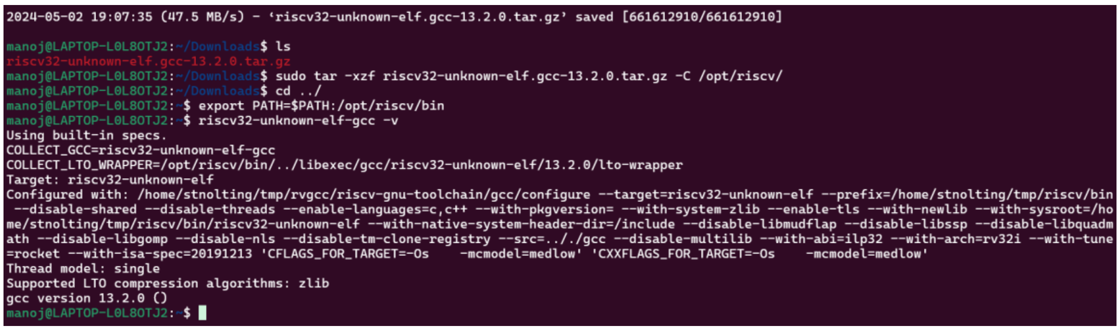
Toolchain prefix: `riscv32-unknown-elf` or `riscv64-unknown-elf` (see the individual releases)

Status	Release (tag)	Download	GCC	binutils	march	mabi	c-lib
🟢	rv32e-231223	download	13.2.0	2.41	`rv32e`	`ilp32e`	newlib
🟢	rv32i-131023	download	13.2.0	2.41	`rv32i`	`ilp32`	newlib
🔴	rv32i-4.0.0	download	12.1.0	2.39	`rv32i`	`ilp32`	newlib

🟢 active / recommended, 🔴 outdated

Figure 4. 2: RISC-V GCC Toolchain

To begin, it's essential to set up the compilation environment correctly. This involves ensuring that the RISC-V toolchain is installed and properly configured. The toolchain can be downloaded from the GitHub page (Stephen, 2024) as a single file and extracted into the desired path in the Linux environment. Before using the toolchain, adding the toolchain's binaries to the PATH environment variable is necessary to make the tools accessible from the command line variable by using ‘export PATH=$PATH (followed by the binaries folder of the toolchain)”. The toolchain can be tested using the ‘riscv32-unknown-elf-gcc -v’ command.



```
2024-05-02 19:07:35 (47.5 MB/s) - ‘riscv32-unknown-elf.gcc-13.2.0.tar.gz’ saved [661612910/661612910]

manoj@LAPTOP-L0L8OTJ2:~/Downloads$ ls
riscv32-unknown-elf.gcc-13.2.0.tar.gz
manoj@LAPTOP-L0L8OTJ2:~/Downloads$ sudo tar -xzf riscv32-unknown-elf.gcc-13.2.0.tar.gz -C /opt/riscv/
manoj@LAPTOP-L0L8OTJ2:~/Downloads$ cd ../
manoj@LAPTOP-L0L8OTJ2:~$ export PATH=$PATH:/opt/riscv/bin
manoj@LAPTOP-L0L8OTJ2:~$ riscv32-unknown-elf-gcc -v
Using built-in specs.
COLLECT_GCC=riscv32-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/opt/riscv/bin/../libexec/gcc/riscv32-unknown-elf/13.2.0/lto-wrapper
Target: riscv32-unknown-elf
Configured with: /home/stnolting/tmp/rvgcc/riscv-gnu-toolchain/gcc/configure --target=riscv32-unknown-elf --prefix=/home/stnolting/tmp/riscv/bin
 --disable-shared --disable-threads --enable-languages=c,c++ --with-pkgversion= --with-system-zlib --enable-tls --with-newlib --with-sysroot=/ho
me/stnolting/tmp/riscv/bin/riscv32-unknown-elf --with-native-system-header-dir=/include --disable-libmudflap --disable-libssp --disable-libquadm
ath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=.././gcc --disable-multilib --with-abi=ilp32 --with-arch=rv32i --with-tune
=rocket --with-isa-spec=20191213 'CFLAGS_FOR_TARGET=-Os    -mcmodel=medlow' 'CXXFLAGS_FOR_TARGET=-Os    -mcmodel=medlow'
Thread model: single
Supported LTO compression algorithms: zlib
gcc version 13.2.0 ()
manoj@LAPTOP-L0L8OTJ2:~$ █
```

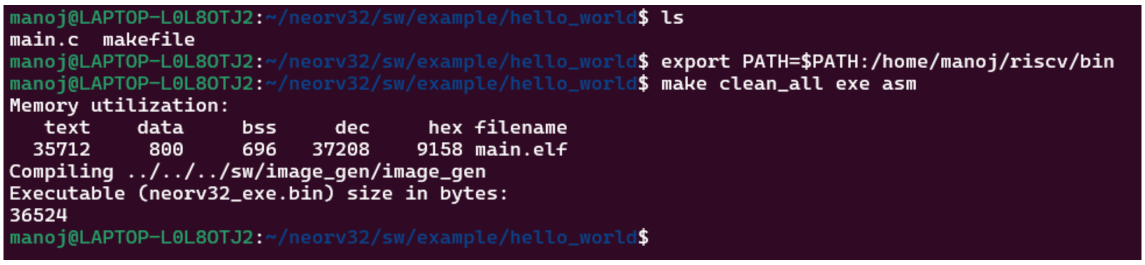
Figure 4. 3: Toolchain installation

If the installation was successful, the following commands will return the GCC compiler version and the targeted 32-bit RIS-V environment and the supported ISA extensions.

4.3 Application Program Compilation

This section of the guide shows the compilation of an example C code application to a RISC-V executable and assembly code. Application compilation is done on a single GNU makefile that is located at “sw/common/common.mk”

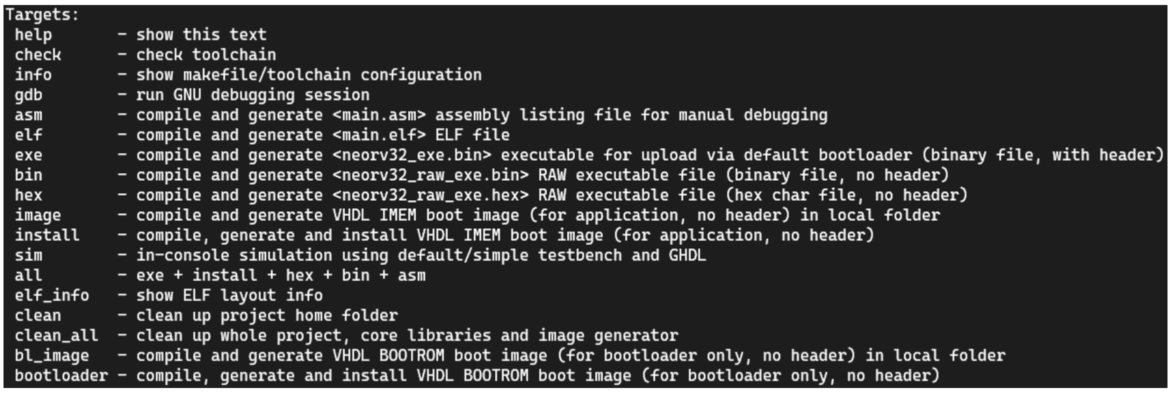
Navigate to the example file in the cloned repository, export the RISC-V bin PATH and execute the ‘make clean_all exe asm’ command. This command will clean any previous builds and compile the C source code into an executable.



```
manoj@LAPTOP-L0L8OTJ2:~/neorv32/sw/example/hello_world$ ls
main.c  makefile
manoj@LAPTOP-L0L8OTJ2:~/neorv32/sw/example/hello_world$ export PATH=$PATH:/home/manoj/riscv/bin
manoj@LAPTOP-L0L8OTJ2:~/neorv32/sw/example/hello_world$ make clean_all exe asm
Memory utilization:
   text    data     bss     dec     hex filename
  35712     800     696   37208    9158 main.elf
Compiling ../../../sw/image_gen/image_gen
Executable (neorv32_exe.bin) size in bytes:
36524
manoj@LAPTOP-L0L8OTJ2:~/neorv32/sw/example/hello_world$
```

Figure 4. 4: C Code compilation

Besides ‘asm’ there are executions for the make which can be displayed by executing ‘make help’.



```
Targets:
 help       - show this text
 check      - check toolchain
 info       - show makefile/toolchain configuration
 gdb        - run GNU debugging session
 asm        - compile and generate <main.asm> assembly listing file for manual debugging
 elf        - compile and generate <main.elf> ELF file
 exe        - compile and generate <neorv32_exe.bin> executable for upload via default bootloader (binary file, with header)
 bin        - compile and generate <neorv32_raw_exe.bin> RAW executable file (binary file, no header)
 hex        - compile and generate <neorv32_raw_exe.hex> RAW executable file (hex char file, no header)
 image      - compile and generate VHDL IMEM boot image (for application, no header) in local folder
 install    - compile, generate and install VHDL IMEM boot image (for application, no header)
 sim        - in-console simulation using default/simple testbench and GHDL
 all        - exe + install + hex + bin + asm
 elf_info   - show ELF layout info
 clean      - clean up project home folder
 clean_all  - clean up whole project, core libraries and image generator
 bl_image   - compile and generate VHDL BOOTROM boot image (for bootloader only, no header) in local folder
 bootloader - compile, generate and install VHDL BOOTROM boot image (for bootloader only, no header)
```

Figure 4. 5: Makefile target commands

Upon execution, the memory utilization details are displayed and the executable, main.elf along with the assembly file is generated in the current directory, as shown in the figure 4.4. This method ensures that both the executable and assembly files are correctly produced for further deployment on the FPGA.

By following these detailed steps, users can effectively compile application programs for the RISC-V processor, ensuring a smooth and efficient development process.

CHAPTER 5 UPLOADING CORE INTO FPGA

5.1 Board Choice

The FPGA board choice is dependable on the language used to describe the RISC-V processor. Since the RISC-V processor chosen is written in VHDL, a vast number of boards can be used. The FPGA board used in this project was the Intel Altera De2i-150 Development Board. Where it supports both VHDL and Verilog language, a well-documented board user manual (Altera, 2016) and a user-friendly interface IDE (Quartus II).

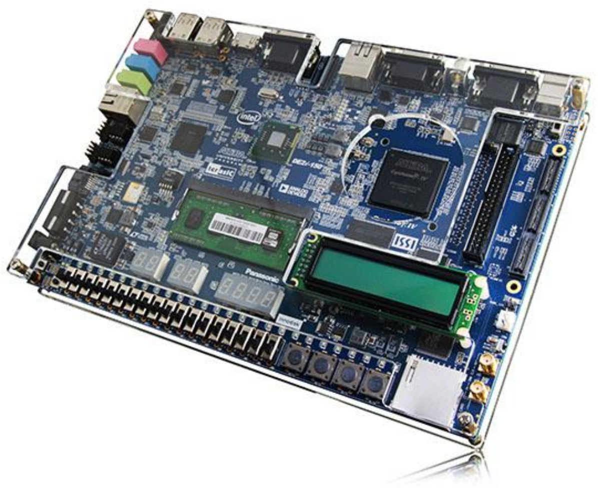


Figure 5. 1: Intel Altera De2i-150 FPGA Board

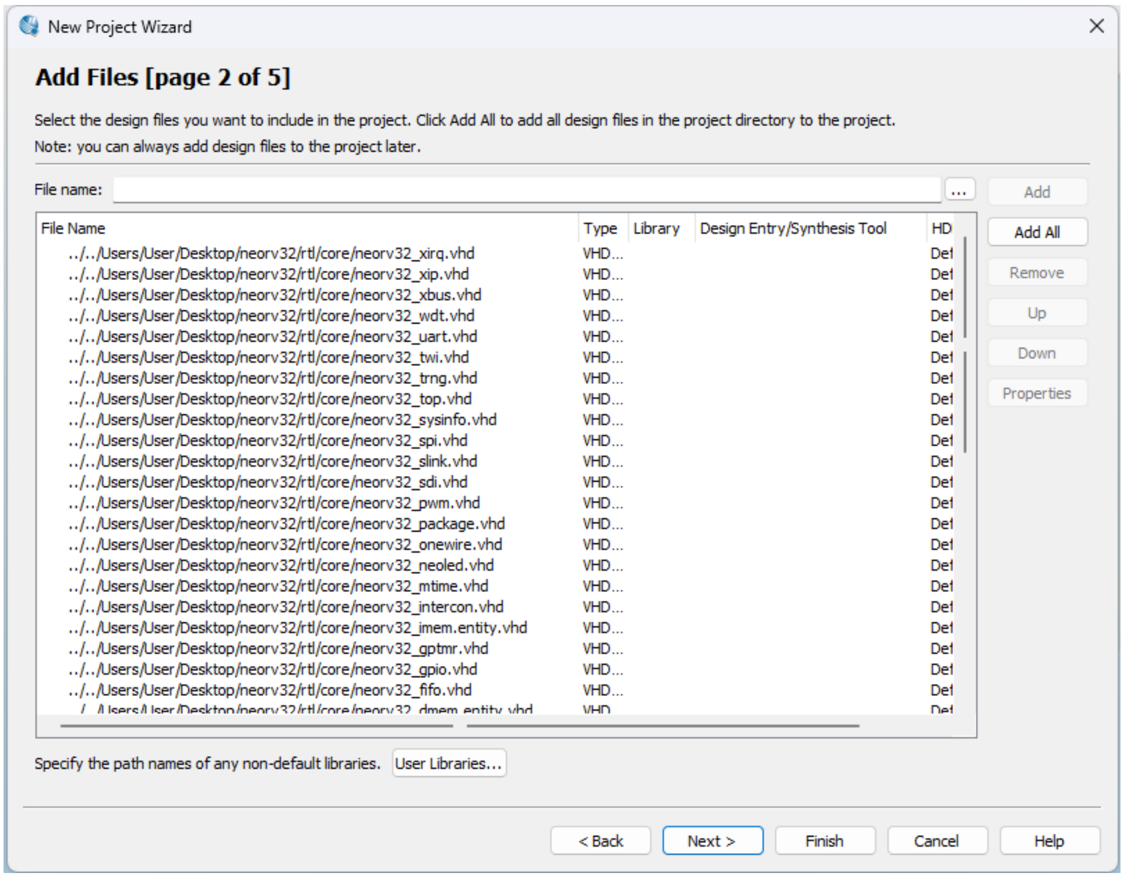
Some details of the board:

- FPGA: Cyclone IV EP4CGX150DF31 device
- Logic elements: 149,760 Les
- Phase-locked loop (PLL): 8 PLLs
- Embedded Memory: 6,480 Kbits
- Memory Blocks: 720 M9K
- Programmer: On-board USB Blaster circuitry
- Header: 40-pin expansion port (voltage levels: 3.3V)
- Clock: 50MHz

5.2 Hardware Setup

The pre-requisite for this part is to download the Quartus II software which is free from the Intel webpage and cloned NEORV32 RISC-V core repository.

Load Intel Quartus II software and choose “New Project Wizard” and place the project working directory into the cloned repository folder that contains the RICS-V core. Navigate to the RTL folder where all the VHDL files that make up the core are located. Import these files into the software by clicking "Add All" to include all design files in the project directory. This step is crucial for ensuring that all necessary VHDL files are available for synthesis.



Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.
Note: you can always add design files to the project later.

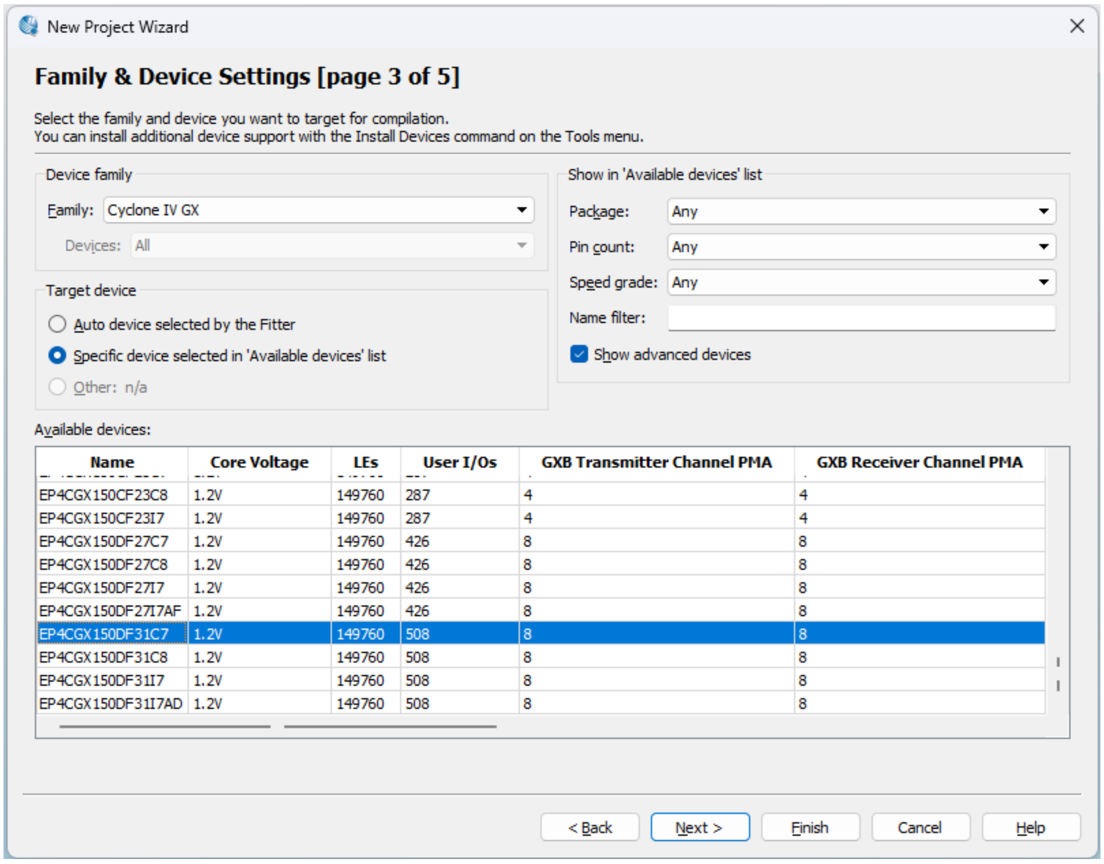
File name:

File Name	Type	Library	Design Entry/Synthesis Tool	HD
../../Users/User/Desktop/neorv32/rtl/core/neorv32_xirq.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_xip.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_xbus.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_wdt.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_uart.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_twi.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_trng.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_top.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_sysinfo.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_spi.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_slink.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_sdi.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_pwm.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_package.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_onewire.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_neoled.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_mtime.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_intercon.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_imem.entity.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_gptmr.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_gpio.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_fifo.vhd	VHD...			Def
../../Users/User/Desktop/neorv32/rtl/core/neorv32_dmem.entity.vhd	VHD			Def

Specify the path names of any non-default libraries.

Figure 5. 2: Adding RTL cores into project files

Select the appropriate FPGA board to ensure that the design is compatible with the hardware that will be used.



New Project Wizard

Family & Device Settings [page 3 of 5]

Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

Device family

Family: Cyclone IV GX

Devices: All

Target device

( ) Auto device selected by the Fitter

[✓ Specific device selected in 'Available devices' list]

( ) Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Speed grade: Any

Name filter:

[✓] Show advanced devices

Available devices:

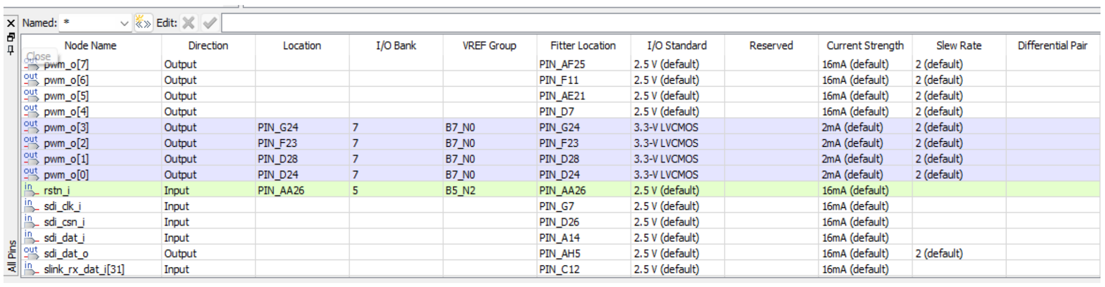
Name	Core Voltage	LEs	User I/Os	GXB Transmitter Channel PMA	GXB Receiver Channel PMA
EP4CGX150CF23C8	1.2V	149760	287	4	4
EP4CGX150CF23I7	1.2V	149760	287	4	4
EP4CGX150DF27C7	1.2V	149760	426	8	8
EP4CGX150DF27C8	1.2V	149760	426	8	8
EP4CGX150DF27I7	1.2V	149760	426	8	8
EP4CGX150DF27I7AF	1.2V	149760	426	8	8
EP4CGX150DF31C7	1.2V	149760	508	8	8
EP4CGX150DF31C8	1.2V	149760	508	8	8
EP4CGX150DF31I7	1.2V	149760	508	8	8
EP4CGX150DF31I7AD	1.2V	149760	508	8	8

Figure 5. 3: Selecting the appropriate FPGA board

With the core files loaded and the correct FPGA board selected into Quartus II, the core is ready to be modified. For default setup, the core can be left as it is but the only generic needed to be changed is the CLOCK_FREQUENCY. As mentioned in section 3.1, since the FPGA uses a 50Mhz external clock signal, the clock generic must be matched accordingly. The next step will be performing “analysis and synthesis”, this step will estimate the resource utilization for partial design which will be needed for the next step which is on pin assignments.

5.3 Pin Assignments

Upon successful completion of setting up the Quartus II Ide and the core loaded, the next step is to assign the pins. The pin assignments can be done via the pin planner button located on the ribbon bar.



Named: * Edit:

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair
pwm_o[7]	Output				PIN_AF25	2.5 V (default)		16mA (default)	2 (default)	
pwm_o[6]	Output				PIN_F11	2.5 V (default)		16mA (default)	2 (default)	
pwm_o[5]	Output				PIN_AE21	2.5 V (default)		16mA (default)	2 (default)	
pwm_o[4]	Output				PIN_D7	2.5 V (default)		16mA (default)	2 (default)	
pwm_o[3]	Output	PIN_G24	7	B7_N0	PIN_G24	3.3-V LVCMOS		2mA (default)	2 (default)	
pwm_o[2]	Output	PIN_F23	7	B7_N0	PIN_F23	3.3-V LVCMOS		2mA (default)	2 (default)	
pwm_o[1]	Output	PIN_D28	7	B7_N0	PIN_D28	3.3-V LVCMOS		2mA (default)	2 (default)	
pwm_o[0]	Output	PIN_D24	7	B7_N0	PIN_D24	3.3-V LVCMOS		2mA (default)	2 (default)	
rstn_i	Input	PIN_AA26	5	B5_N2	PIN_AA26	2.5 V (default)		16mA (default)		
sdi_clk_i	Input				PIN_G7	2.5 V (default)		16mA (default)		
sdi_csn_i	Input				PIN_D26	2.5 V (default)		16mA (default)		
sdi_dat_i	Input				PIN_A14	2.5 V (default)		16mA (default)		
sdi_dat_o	Output				PIN_AH5	2.5 V (default)		16mA (default)	2 (default)	
slink_rx_dat_i[31]	Input				PIN_C12	2.5 V (default)		16mA (default)		

Figure 5. 4: FPGA pin assignments

The pins can be assigned accordingly by following the FPGA board system manual on the FPGA chip pins that are hard wired to the specific components for example the clock source and the buttons for the reset.

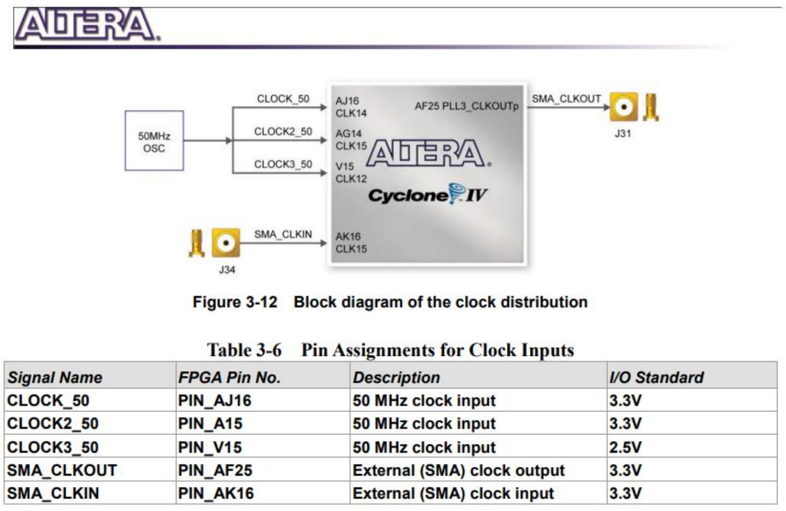
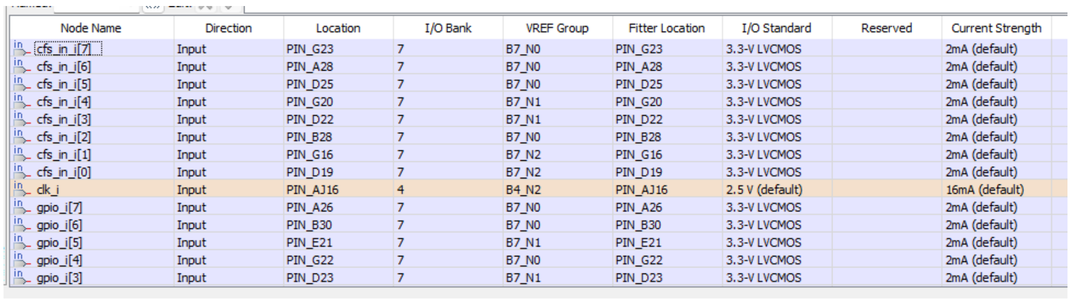

ALTERA
50MHz OSC
CLOCK_50
CLOCK2_50
CLOCK3_50
AJ16 CLK14
AG14 CLK15
V15 CLK12
AF25 PLL3_CLKOUTp
SMA_CLKOUT
J31
SMA_CLKIN
AK16 CLK15
J34
ALTERA
Cyclone IV

Figure 3-12 Block diagram of the clock distribution

Table 3-6 Pin Assignments for Clock Inputs

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
CLOCK_50	PIN_AJ16	50 MHz clock input	3.3V
CLOCK2_50	PIN_A15	50 MHz clock input	3.3V
CLOCK3_50	PIN_V15	50 MHz clock input	2.5V
SMA_CLKOUT	PIN_AF25	External (SMA) clock output	3.3V
SMA_CLKIN	PIN_AK16	External (SMA) clock input	3.3V

Figure 5. 5: FPGA Board user manual on oscillator clock pins



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength
cfs_in_i[7]	Input	PIN_G23	7	B7_N0	PIN_G23	3.3-V LVCMOS		2mA (default)
cfs_in_i[6]	Input	PIN_A28	7	B7_N0	PIN_A28	3.3-V LVCMOS		2mA (default)
cfs_in_i[5]	Input	PIN_D25	7	B7_N0	PIN_D25	3.3-V LVCMOS		2mA (default)
cfs_in_i[4]	Input	PIN_G20	7	B7_N1	PIN_G20	3.3-V LVCMOS		2mA (default)
cfs_in_i[3]	Input	PIN_D22	7	B7_N1	PIN_D22	3.3-V LVCMOS		2mA (default)
cfs_in_i[2]	Input	PIN_B28	7	B7_N0	PIN_B28	3.3-V LVCMOS		2mA (default)
cfs_in_i[1]	Input	PIN_G16	7	B7_N2	PIN_G16	3.3-V LVCMOS		2mA (default)
cfs_in_i[0]	Input	PIN_D19	7	B7_N2	PIN_D19	3.3-V LVCMOS		2mA (default)
clk_i	Input	PIN_AJ16	4	B4_N2	PIN_AJ16	2.5 V (default)		16mA (default)
gpio_i[7]	Input	PIN_A26	7	B7_N0	PIN_A26	3.3-V LVCMOS		2mA (default)
gpio_i[6]	Input	PIN_B30	7	B7_N0	PIN_B30	3.3-V LVCMOS		2mA (default)
gpio_i[5]	Input	PIN_E21	7	B7_N1	PIN_E21	3.3-V LVCMOS		2mA (default)
gpio_i[4]	Input	PIN_G22	7	B7_N0	PIN_G22	3.3-V LVCMOS		2mA (default)
gpio_i[3]	Input	PIN_D23	7	B7_N1	PIN_D23	3.3-V LVCMOS		2mA (default)

Figure 5. 6: I/O standard selection

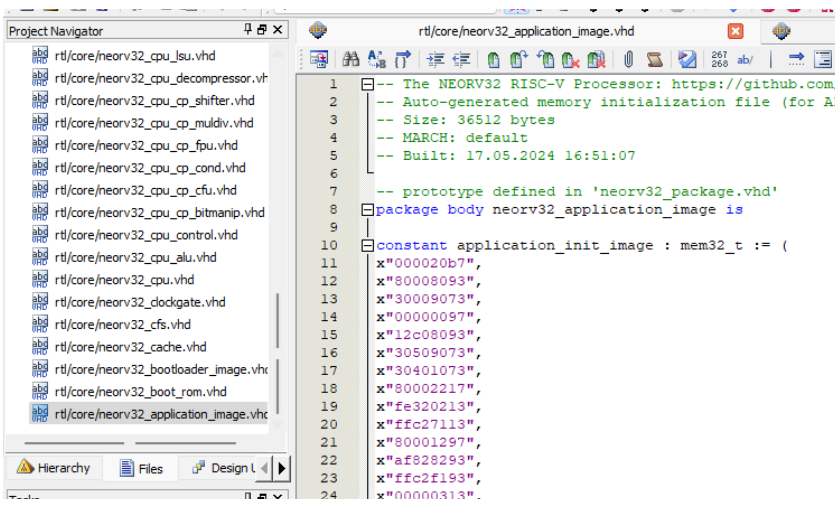
The I/O standard of the pins should be selected to 3.3V LVCMOS to produce and receive 3.3V devices and sensors like motor drivers or UART to operate in 3.3V logic to interface with a vast variety of microcontrollers and sensors.

Upon completion of the Pin Assignments, the next 2 subsequent topics will be the process of uploading the Code into the core followed by the synthesizing and uploading bitstream. As mentioned previously, there are 2 ways, direct and indirect. Optionally, the user can choose either one of the methods.

5.4 Uploading Executable Directly into Memory

If the user wishes not to use the bootloader to upload the executable file, alternatively the executable can be directly installed into the embedded memory. As mentioned previously in section 3.6 on boot configurations, this concept uses the ‘Direct Boot’ where the source code application hex file will be synthesized together with the core and programmed the FPGA via the bitstream.

Execute ‘make clean_all exe install’ to generate the application image with the hex file and include the “application image file” into Quartus II project files together with the core and its modules.



rtl/core/neorv32_cpu_lsu.vhd
rtl/core/neorv32_cpu_decompressor.vh
rtl/core/neorv32_cpu_cp_shifter.vhd
rtl/core/neorv32_cpu_cp_muldiv.vhd
rtl/core/neorv32_cpu_cp_fpu.vhd
rtl/core/neorv32_cpu_cp_cond.vhd
rtl/core/neorv32_cpu_cp_cfu.vhd
rtl/core/neorv32_cpu_cp_bitmanip.vhd
rtl/core/neorv32_cpu_control.vhd
rtl/core/neorv32_cpu_alu.vhd
rtl/core/neorv32_cpu.vhd
rtl/core/neorv32_clockgate.vhd
rtl/core/neorv32_cfs.vhd
rtl/core/neorv32_cache.vhd
rtl/core/neorv32_bootloader_image.vhc
rtl/core/neorv32_boot_rom.vhd
rtl/core/neorv32_application_image.vhc

```
-- The NEORV32 RISC-V Processor: https://github.com
-- Auto-generated memory initialization file (for A
-- Size: 36512 bytes
-- MARCH: default
-- Built: 17.05.2024 16:51:07

-- prototype defined in 'neorv32_package.vhd'
package body neorv32_application_image is

constant application_init_image : mem32_t := (
x"000020b7",
x"80008093",
x"30009073",
x"00000097",
x"12c08093",
x"30509073",
x"30401073",
x"80002217",
x"fe320213",
x"ffc27113",
x"80001297",
x"af828293",
x"ffc2f193",
x"00000313",
```

Figure 5. 7: Uploading Hex File into Application Image

Disable the internal bootloader so that the IMEM can be initialized with the executable during the synthesis.

```
INT_BOOTLOADER_EN : boolean := false; -- boot configuration: true = boot
explicit bootloader; false = boot from int/ext (I)MEM
```

Figure 5. 8: Disabling Bootloader Configuration

Perform synthesis and upload the bitstream and the program will start running.

5.5 Uploading Binary Executable via UART

To upload executables via UART, the “INT_BOOTLOADER_EN” and UART0 must be enabled, via the top file.

```
INT_BOOTLOADER_EN : boolean := true; -- boot configuration: true = boot
explicit bootloader; false = boot from int/ext (I)MEM
```

Figure 5. 9: Enabling Bootloader Configuration

Connect the primary UART0 following the pin assigned for uart0_tx and uart0_rx from the pin planner from the FPGA to the USB-to-UART converter. For this project, the connection diagram as shown in figure 5.10.

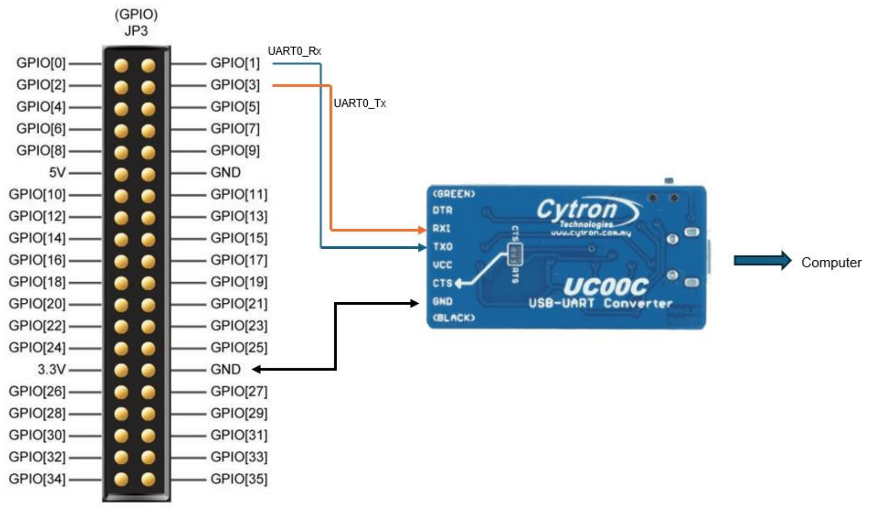


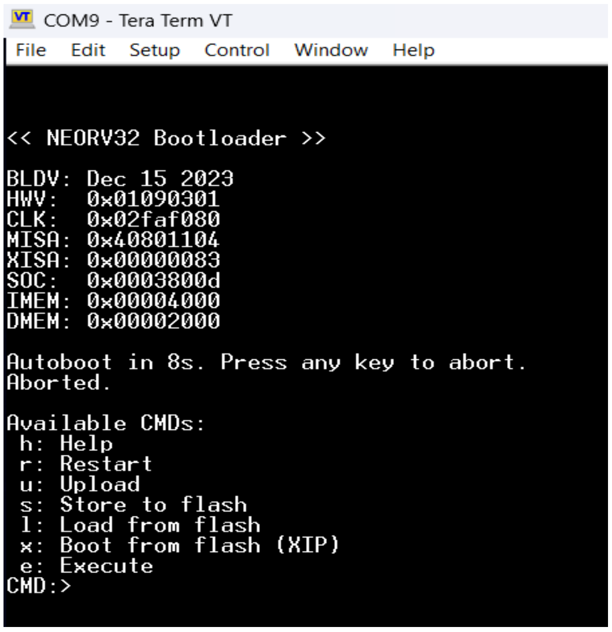
Figure 5. 10: USB to UART Connection

TeraTerm 5 for Windows will be used as the terminal program as the program is able to transfer raw data bytes without any overhead protocol. TeraTerm 5 can be downloaded from the TeraTermProject official GitHub.

Start a terminal console program with the following settings is used to initialize the serial port.

- 19200 Baud rate
- 8 data bits
- No parity bit
- 1 stop bit
- Newline on \r\n

Upon successful configuration of the the terminal program will printout the bootloader console program.

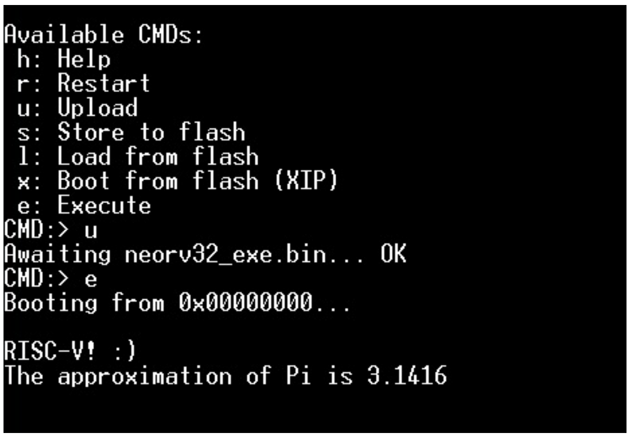


```
<< NEORV32 Bootloader >>

BLDV: Dec 15 2023
HWV:  0x01090301
CLK:  0x02faf080
MISA: 0x40801104
XISA: 0x00000083
SOC:  0x0003800d
IMEM: 0x00004000
DMEM: 0x00002000

Autoboot in 8s. Press any key to abort.
Aborted.

Available CMDs:
 h: Help
 r: Restart
 u: Upload
 s: Store to flash
 l: Load from flash
 x: Boot from flash (XIP)
 e: Execute
CMD:>
```

Figure 5. 11: NEORV32 Bootloader

Executing the "Upload" command by typing "u" in the terminal, it will return "Awaiting neorv32_exe.bin…". Locate the "send file" on the terminal program and upload the binary file. If the file is successfully uploaded the terminal will return a "OK".

Now, the binary file is in the Instruction Memory of the processor. To execute the code, type "e" and it will boot from the start of the address.

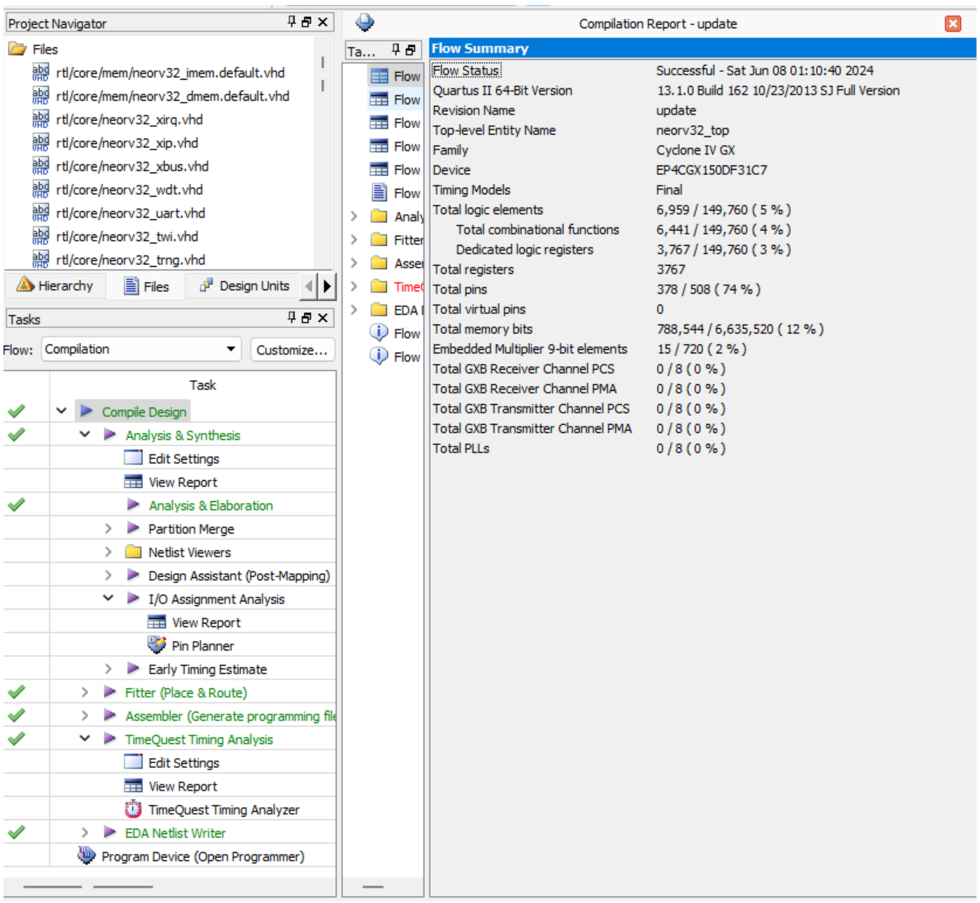


```
Available CMDs:
 h: Help
 r: Restart
 u: Upload
 s: Store to flash
 l: Load from flash
 x: Boot from flash (XIP)
 e: Execute
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> e
Booting from 0x00000000...

RISC-V! :)
The approximation of Pi is 3.1416
```

Figure 5. 12: Executing program files

5.6 Synthesizing The Design

Upon successful completion of hardware, the user may choose to further configure the core by enabling the ISA extensions or modifying the code. The next step will be synthesizing by compiling using the “Compile Design”. After the compilation is done the software provides a compilation report.

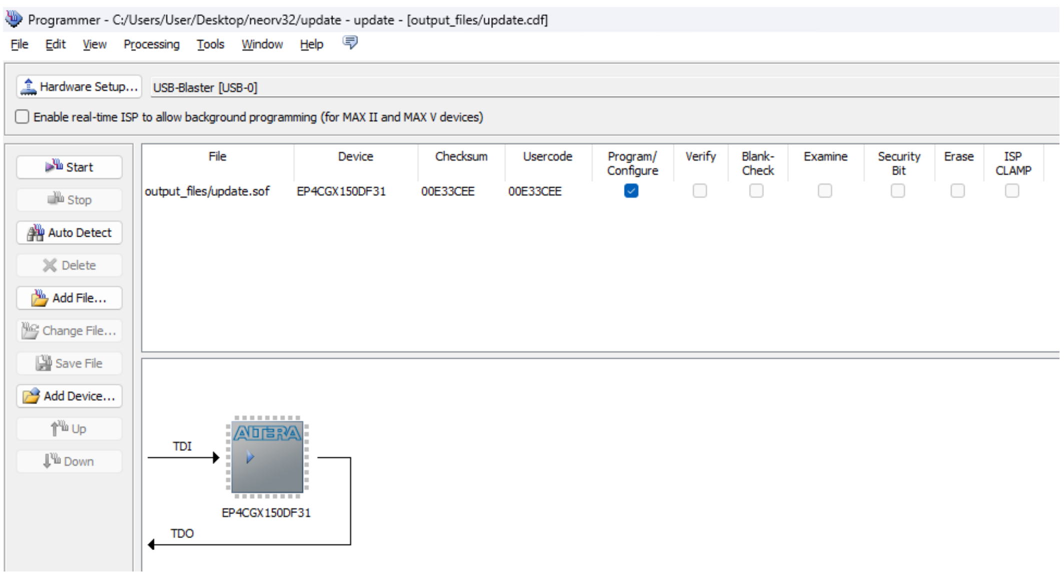


Flow Status	Successful - Sat Jun 08 01:10:40 2024
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Full Version
Revision Name	update
Top-level Entity Name	neorv32_top
Family	Cyclone IV GX
Device	EP4CGX150DF31C7
Timing Models	Final
Total logic elements	6,959 / 149,760 (5 %)
Total combinational functions	6,441 / 149,760 (4 %)
Dedicated logic registers	3,767 / 149,760 (3 %)
Total registers	3767
Total pins	378 / 508 (74 %)
Total virtual pins	0
Total memory bits	788,544 / 6,635,520 (12 %)
Embedded Multiplier 9-bit elements	15 / 720 (2 %)
Total GXB Receiver Channel PCS	0 / 8 (0 %)
Total GXB Receiver Channel PMA	0 / 8 (0 %)
Total GXB Transmitter Channel PCS	0 / 8 (0 %)
Total GXB Transmitter Channel PMA	0 / 8 (0 %)
Total PLLs	0 / 8 (0 %)

Figure 5. 13: Synthesizing the RTL files

5.7 Uploading the Bitstream Via Programmer

After compiling the design, the FPGA can be programmed. The SOF file will be automatically loaded and ready to be downloaded into the FPGA using the USB-Blaster circuitry. Open the “Tools” tab and choose “Programmer”, a programmer window will pop up.


Programmer - C:/Users/User/Desktop/neorv32/update - update - [output_files/update.cdf]

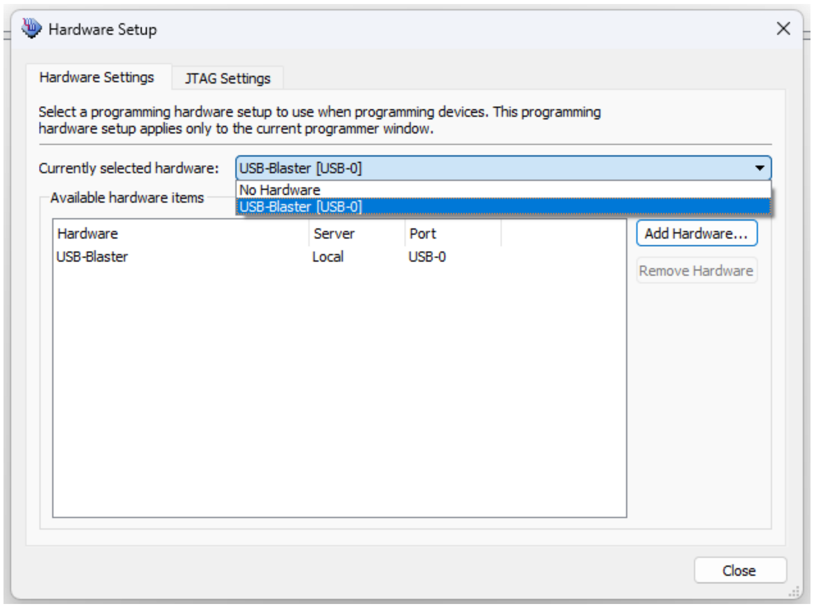
Hardware Setup... USB-Blaster [USB-0]

Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase	ISP CLAMP
output_files/update.sof	EP4CGX150DF31	00E33CEE	00E33CEE	✓	☐	☐	☐	☐	☐	☐

Figure 5. 14: Programmer Tab

Click on the Hardware Setup and select “USB-Blaster” option under currently selected hardware.


Hardware Setup

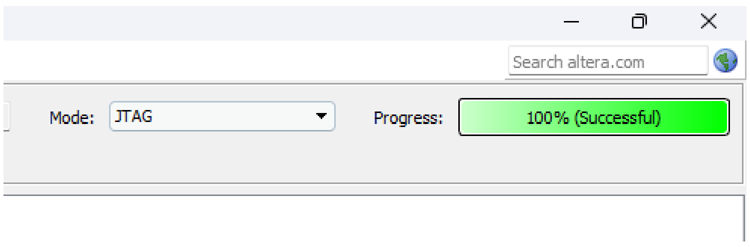
Select a programming hardware setup to use when programming devices. This programming hardware setup applies only to the current programmer window.

Currently selected hardware: USB-Blaster [USB-0]

Hardware	Server	Port
USB-Blaster	Local	USB-0

Figure 5. 15: Selecting Programmer Hardware

After selecting the programming hardware, the SFO file can be uploaded into the FPGA by clicking the “Start”. Upon successful uploading the program, the progress bar will return 100%. Now the compiled core is loaded into the FPGA and the design should be running



Mode: JTAG

Progress: 100% (Successful)

Figure 5. 16: Progress Bar

5.8 Summary

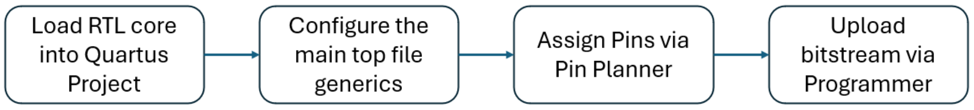

Load RTL core into Quartus Project
Configure the main top file generics
Assign Pins via Pin Planner
Upload bitstream via Programmer

Figure 5. 17: Summary on Uploading RISC-V Core

- Load RTL core into Quartus Project: This step involves importing the RTL (Register-Transfer Level) description of the RISC-V core into a Quartus project. This description defines the hardware design.
- Configure the main top file generics: In this step, configure the top-level file of the project with specific parameters and generics that are required for the design.
- Assign Pins via Pin Planner: Using the Pin Planner tool in Quartus, assign the FPGA pins to the appropriate signals. This step is crucial for mapping the pins to the physical FPGA.
- Upload bitstream via Programmer: Finally, generate a bitstream file from the design and use the Quartus Programmer tool to upload this bitstream to the FPGA board.

CHAPTER 6 APPLICATIONS AND RESULTS

6.1 Overview

Shifting from theoretical to application, a 4-wheel holonomic X-drive will be developed using the RISC-V core as the main processor where high computational demands such as PID control and inverse kinematics. In addition, this application will be an ideal platform to leverage and show the capabilities of the RISC-V processor.

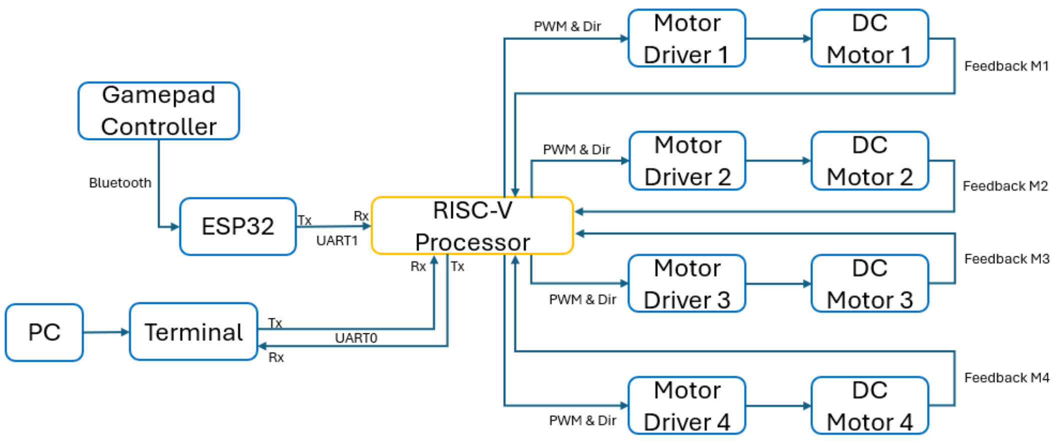


Figure 6. 1: Overview of the Application

This application not only demonstrates the practical use of RISC-V processors in robotics but also addresses several key challenges in the field, such as real-time processing, efficient motor control, and seamless wireless communication. By leveraging the capabilities of the RISC-V architecture, this project aims to achieve high performance and flexibility in robotic control systems.

The development of the 4-wheel holonomic X-drive consists of several key components and technologies:

- RISC-V Processor: Serving as the central processing unit, the RISC-V processor is chosen for its modular and extensible architecture, which supports custom instructions and specialized extensions tailored for specific applications. This processor will handle complex computations required for motion control and kinematics.

- Feedback Mechanisms: Each motor will provide feedback to the processor, ensuring accurate control and synchronization. This feedback loop is crucial for implementing effective PID (Proportional-Integral-Derivative) control, which is essential for maintaining stability and precision in the robot's movements.
- ESP32 Module: The ESP32 module will facilitate wireless communication with the gamepad controller via Bluetooth and with a PC terminal through UART. This setup allows for remote control and real-time monitoring of the robot's status and performance.
- Gamepad Controller: The gamepad controller will provide a user-friendly interface for manually operating the robot. Commands from the controller will be transmitted to the RISC-V processor through the ESP32 module, enabling intuitive control over the robot's movements.

In conclusion, the development of the 4-wheel holonomic X-drive using the RISC-V core showcases the processor's potential to handle complex computational tasks in real-time applications while providing a comprehensive demonstration of its practical applications.

6.2 GPIO Pinout Diagram

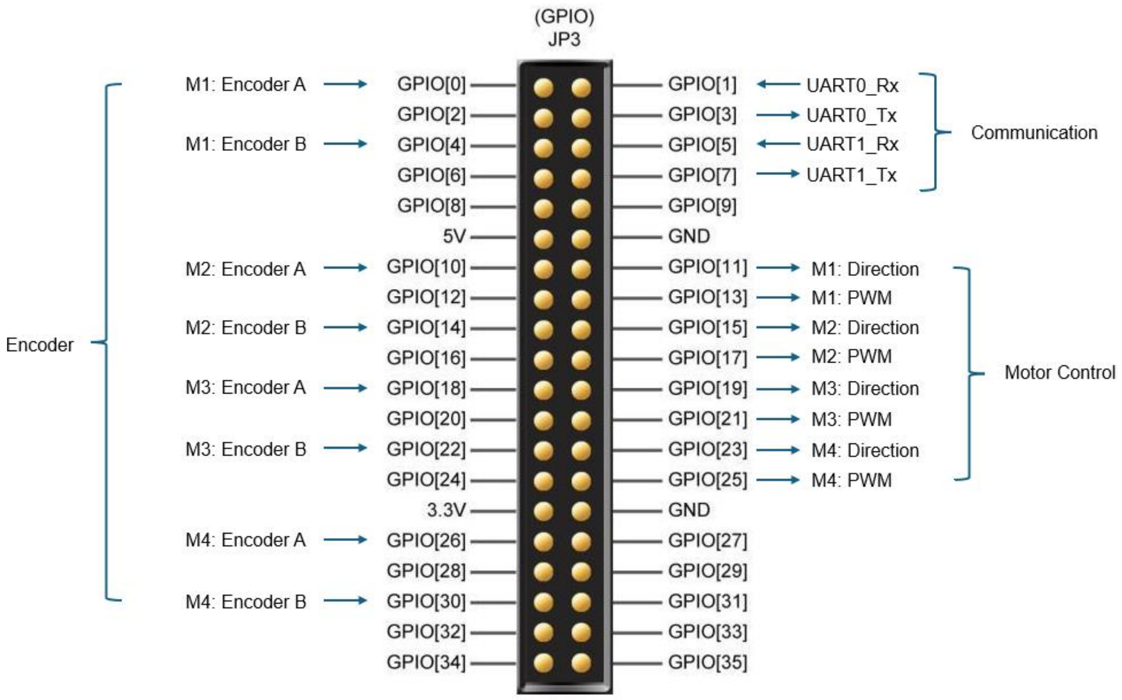


Figure 6. 2: GPIO Pinout

A simple expansion board is created to hard wire the connections for the UART communication, encoders for the individual DC motors as well as PWM and direction outputs.

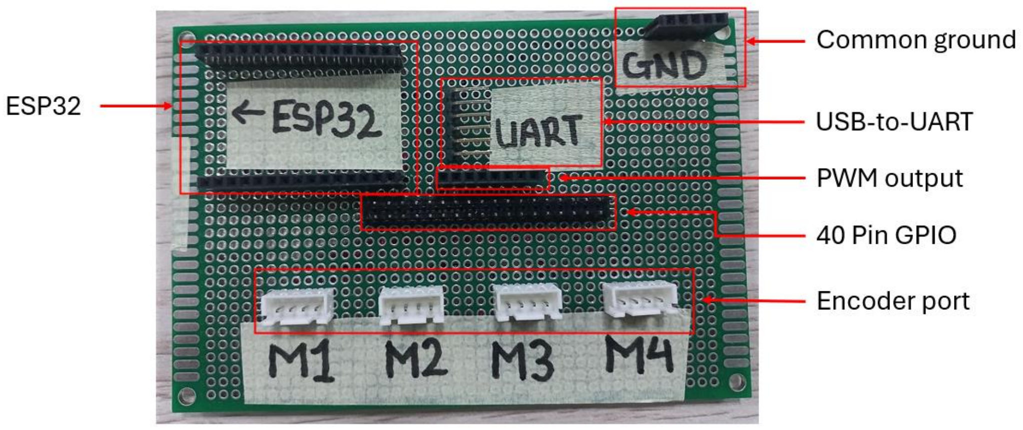


Figure 6. 3: Expension Board Connections

6.3 Quadrature Encoder Implementation using VHDL

A quadrature encoder is used to measure the position and speed of a rotating object. As the encoder rotates, it will generate 2 square wave output signals, output A and output B which are 90 degrees out of phase. There will be 2 scenarios which are output A leading output B and output B leading output A (S. Sarkar, 2018).

- Output A leading output B: The rotation will be clockwise (CW)
- Output B leading output C: The rotation will be counterclockwise (CCW)

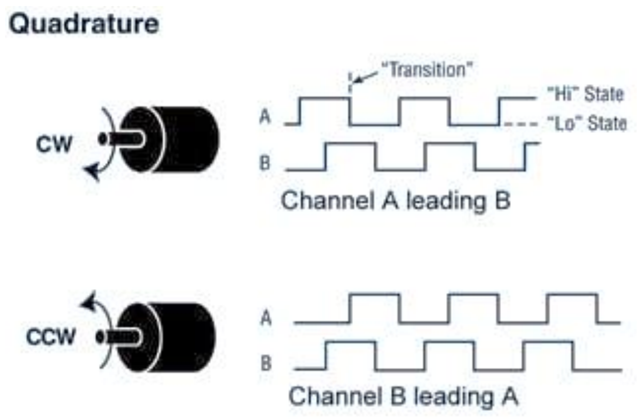
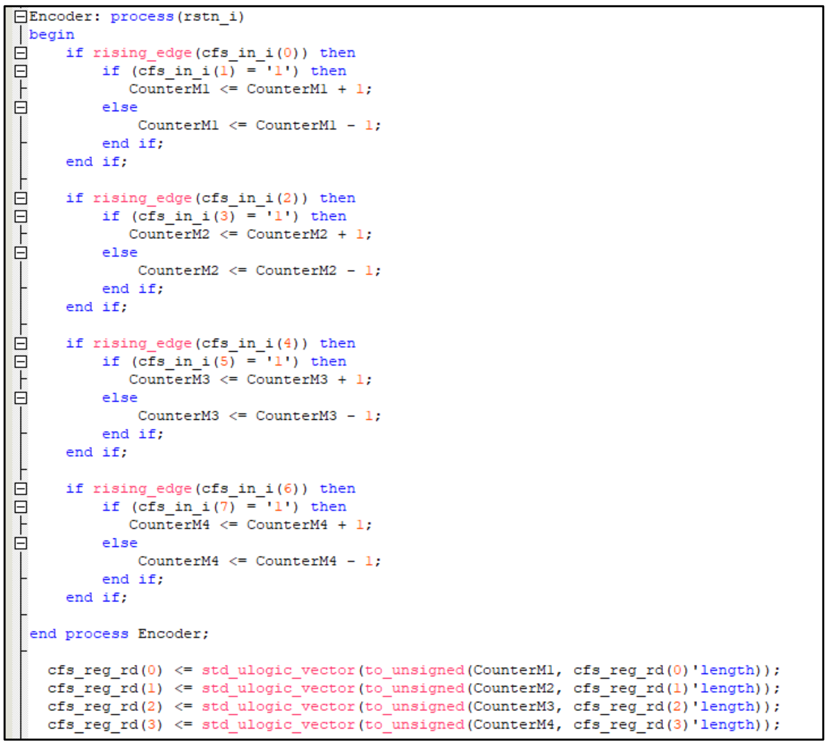


Figure 6. 4: Quadrature Encoder Output Wave

It is possible to get data from the Quadrature Encoder using an external interrupt of the processor alongside with an interrupt service routine to read the encoder counter and it was being tested. Due to the high-speed inputs of 4 DC motors, the program will encounter interrupt overflow and the processor will only run the interrupt service routine and not continue to the main loop of the firmware.

To overcome this issue, the quadrature encoder logic will be implemented at the hardware level using VHDL and be synthesized together with the processor core. The Custom Subsystem Functions (CFS) will be tightly coupled to the processor and can operate independently of the CPU which provides true parallel processing. The implementation will be done in rtl/core/neorv32_cfs.vhd

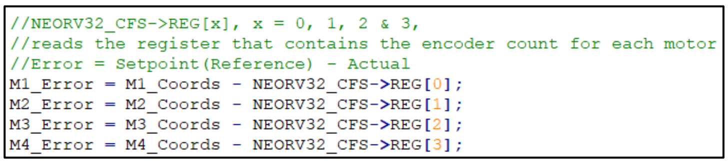


```
Encoder: process(rstn_i)
begin
    if rising_edge(cfs_in_i(0)) then
        if (cfs_in_i(1) = '1') then
           CounterM1 <= CounterM1 + 1;
        else
            CounterM1 <= CounterM1 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(2)) then
        if (cfs_in_i(3) = '1') then
           CounterM2 <= CounterM2 + 1;
        else
            CounterM2 <= CounterM2 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(4)) then
        if (cfs_in_i(5) = '1') then
           CounterM3 <= CounterM3 + 1;
        else
            CounterM3 <= CounterM3 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(6)) then
        if (cfs_in_i(7) = '1') then
           CounterM4 <= CounterM4 + 1;
        else
            CounterM4 <= CounterM4 - 1;
        end if;
    end if;

end process Encoder;

  cfs_reg_rd(0) <= std_ulogic_vector(to_unsigned(CounterM1, cfs_reg_rd(0)'length));
  cfs_reg_rd(1) <= std_ulogic_vector(to_unsigned(CounterM2, cfs_reg_rd(1)'length));
  cfs_reg_rd(2) <= std_ulogic_vector(to_unsigned(CounterM3, cfs_reg_rd(2)'length));
  cfs_reg_rd(3) <= std_ulogic_vector(to_unsigned(CounterM4, cfs_reg_rd(3)'length));
```

Figure 6. 5: Implementation of Encoder Reading using VHDL

The generic cfs_in_i (x), x=0, 1, 2, 3, … takes in data values of the encoder, it behaves and can be treated exactly as a GPIO pin. For example, motor 1, cfs_in_i (0) takes in data from encoder output A and cfs_in_i (1) takes in data from encoder output B of motor 1. From the 2 inputs, the comparison will be made to determine the encoder shaft turning clockwise or counterclockwise and at the same time incrementing or decrementing the encoder pulse value. The same logic goes for motors 2, 3 and 4.

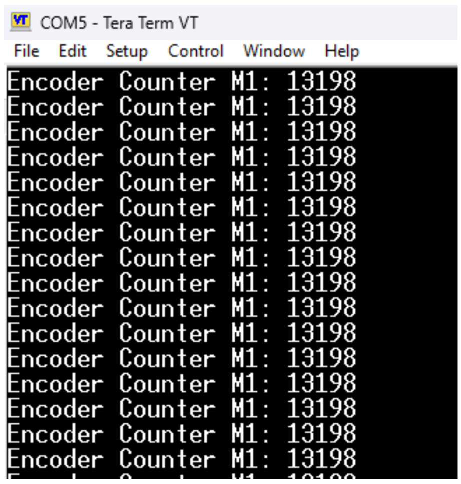
The processed values will be passed to memory read/write registers which can be accessed by the CPU in the C code. With the use of CFS, no external interrupt will be utilized, instead, the memory registers holding the encoder pulses will be called in the Code.



```
//NEORV32_CFS->REG[x], x = 0, 1, 2 & 3,
//reads the register that contains the encoder count for each motor
//Error = Setpoint(Reference) - Actual
M1_Error = M1_Coords - NEORV32_CFS->REG[0];
M2_Error = M2_Coords - NEORV32_CFS->REG[1];
M3_Error = M3_Coords - NEORV32_CFS->REG[2];
M4_Error = M4_Coords - NEORV32_CFS->REG[3];
```

Figure 6. 6: Reading CFS Register in C

The results obtained from the implementation is very satisfactory where an encoder rated for 13000 pulse per revolution is used and the output result after testing is 13198 pulse per revolution.



```
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
Encoder Counter M1: 13198
```

Figure 6. 7: Results from Encoder Reading

6.4 Inverse Kinematics Derivation for Holonomic Drivetrain

Deriving the inverse Kinematics for a holonomic X-drive drivetrain has several benefits, especially in terms of optimization for crucial real-time control and troubleshooting where it gives a deep understanding of the underlying mathematics and mechanics which will be valuable for troubleshooting and enhancing the system. In addition, it is easier to integrate control algorithms.

Several considerations must be done before deriving the equations for the inverse kinematics which include:

- The separation angle of the wheels must be 45 degrees apart
- The wheels must be rotating the same direction if voltage applied to the motor

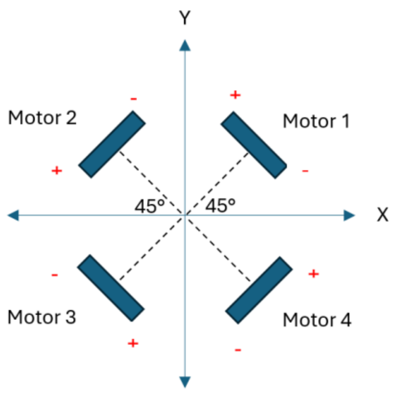


Figure 6. 8: X-drive Configuration

The X-drive wheelbase in figure 6.9 will be used for implementing and testing the firmware for the RISC-V core.

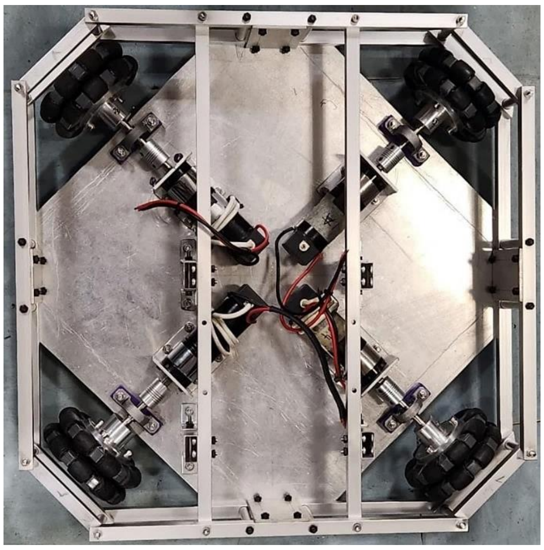


Figure 6. 9: Wheelbase Used in Project

Based on the joystick position and corresponding direction the robot base will move, 4 waves can be plotted based the wheel expected direction of ration for all 4 individual DC motors.

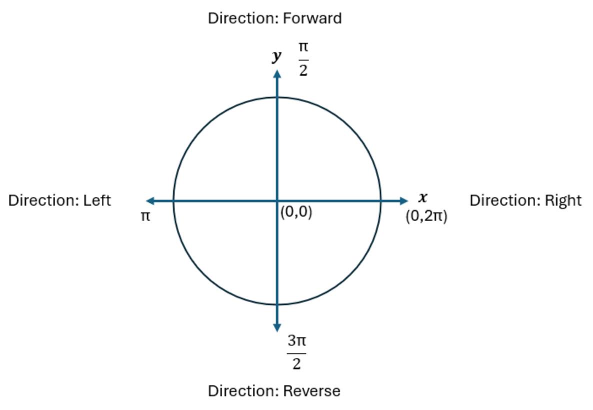


Figure 6. 10: Motion based on Joytick Position

The following behaviour graph can be plotted which describes the wheel's rotation based on the position of the gamepad joystick. From the graph, a pattern is formed which can be formed into equations (1), (2), (3), (4)

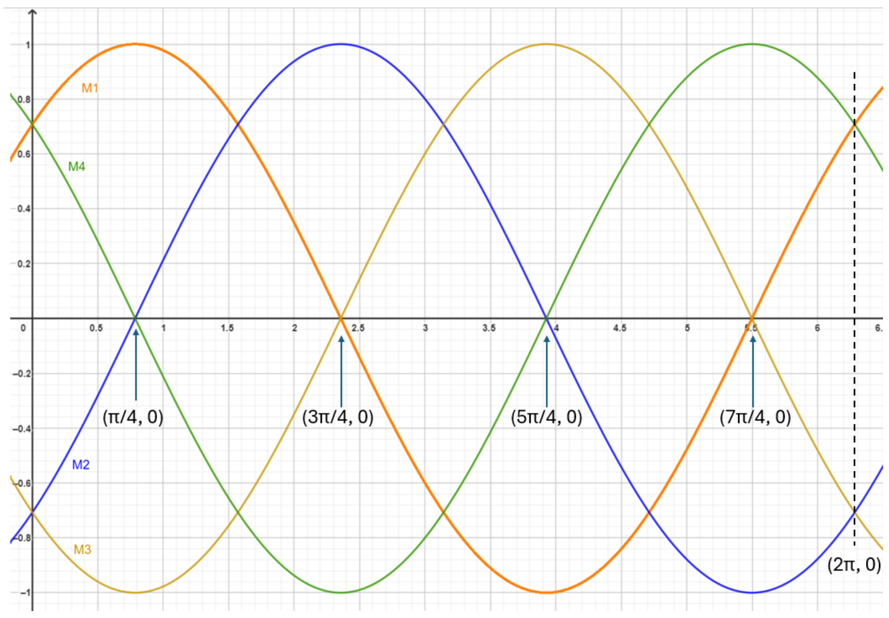


Figure 6. 11: Output Waveform Based on the Predicted Motion

$$M_1 = \sin\left(\theta + \frac{\pi}{4}\right) \qquad (1)$$

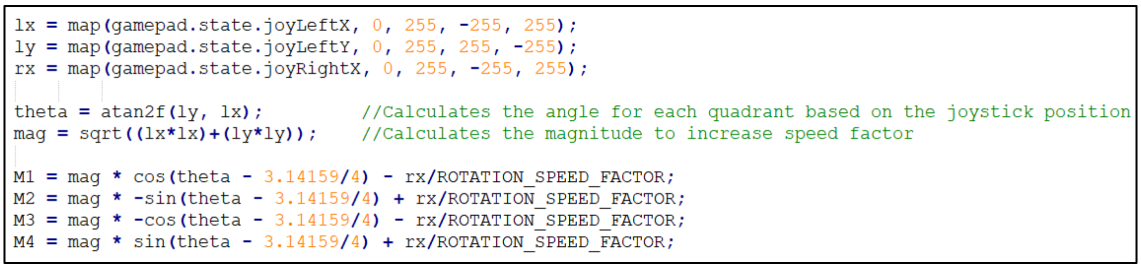
$$M_2 = -\sin\left(\theta + \frac{\pi}{4}\right) \qquad (2)$$

$$M_3 = -\cos\left(\theta + \frac{\pi}{4}\right) \qquad (3)$$

$$M_4 = \cos\left(\theta + \frac{\pi}{4}\right) \qquad (4)$$

Using the kinematics obtained, the angle theta can be obtained by calculating the x and y position of the joystick on each quadrant in radians. It can be done using the function atan2() from C <math.h> header file. This function will take in 2 arguments to calculate the angle in radians.

Based on the equation formed, the implementation in C shown below



```
lx = map(gamepad.state.joyLeftX, 0, 255, -255, 255);
ly = map(gamepad.state.joyLeftY, 0, 255, 255, -255);
rx = map(gamepad.state.joyRightX, 0, 255, -255, 255);

theta = atan2f(ly, lx);          //Calculates the angle for each quadrant based on the joystick position
mag = sqrt((lx*lx)+(ly*ly));     //Calculates the magnitude to increase speed factor

M1 = mag * cos(theta - 3.14159/4) - rx/ROTATION_SPEED_FACTOR;
M2 = mag * -sin(theta - 3.14159/4) + rx/ROTATION_SPEED_FACTOR;
M3 = mag * -cos(theta - 3.14159/4) - rx/ROTATION_SPEED_FACTOR;
M4 = mag * sin(theta - 3.14159/4) + rx/ROTATION_SPEED_FACTOR;
```

Figure 6. 12: Code Snippet on the Implementation of the Inverse Kinematics

6.5 Interfacing DualShock 5 Controller using ESP32

For interfacing the DualShock 5 to the RISC-V core, an ESP32 will be used as the Bluetooth module to connect with the gamepad wirelessly and to avoid any Bluetooth protocol if a typical Bluetooth module is used. PS5 library by rodneybakishan will be also utilized in ESP32 to receive the data from the gamepad.

Since the ESP32 is used to obtain the data, there will be 2 parts in the program, which are the transmission program and the receiver program.

- Transmission: Input from the DualShock5 controller will be read by ESP32 and sent to the RISC-V core
- Receiver: The RISC-V core will receive the data from the EPS32, and the processing will be done

6.5.1 Transmission from ESP32

Considering there are many buttons on the gamepad, an 8-byte data array will be initialized. Using bit manipulation, each byte of data will be processed accordingly of the bits based on the buttons available on the gamepad. When a button from the gamepad is pressed, the bit which was assigned will be masked to "1". When the button is released, the specific bit will be masked to "0". The code snippet shows the implementation.

```
if (ps5.Square()) key_B5 = key_B5 | 0x10;
else  key_B5 = key_B5 & 0xEF;

if (ps5.Cross()) key_B5 = key_B5 | 0x20;
else  key_B5 = key_B5 & 0xDF;

if (ps5.Circle()) key_B5 = key_B5 | 0x40;
else  key_B5 = key_B5 & 0xBF;

if (ps5.Triangle()) key_B5 = key_B5 | 0x80;
else  key_B5 = key_B5 & 0x7F;
```

Figure 6. 13: Code Snippet on the Bit Manipulation of Gamepad Data

The 8-bytes of data will be sent via UART to the RISC-V core.

```
gamepad_data[0] = 0x01;   //unchanged
gamepad_data[1] = Lx_abs; //byte1 -> LX
gamepad_data[2] = Ly_abs; //byte2 -> LY
gamepad_data[3] = Rx_abs; //byte3 -> Rx
gamepad_data[4] = Ry_abs; //byte4 -> Ry
gamepad_data[5] = key_B5;
gamepad_data[6] = key_B6;
gamepad_data[7] = key_B7; //checksum + touchpad + L3, R3

Serial.write(gamepad_data, sizeof(gamepad_data));
```

Figure 6. 14: Code Snippet on the Sending the Data via UART

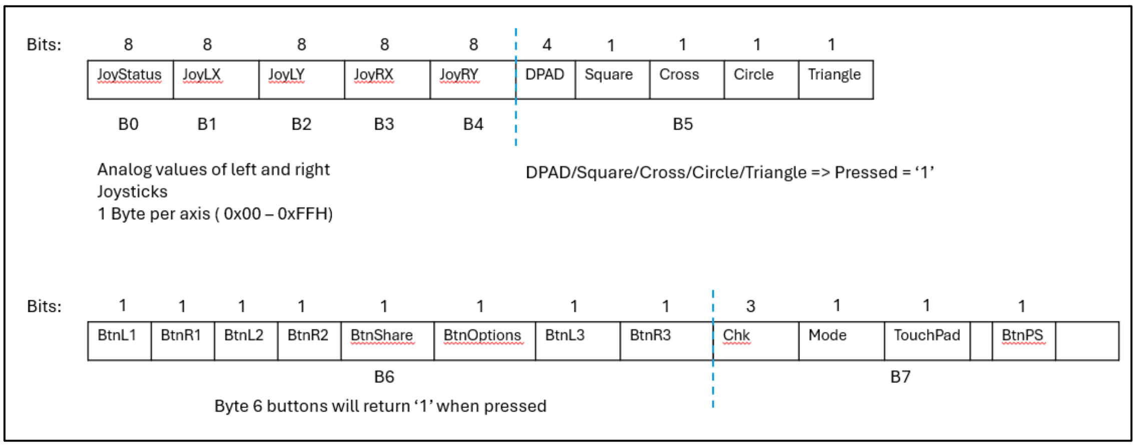
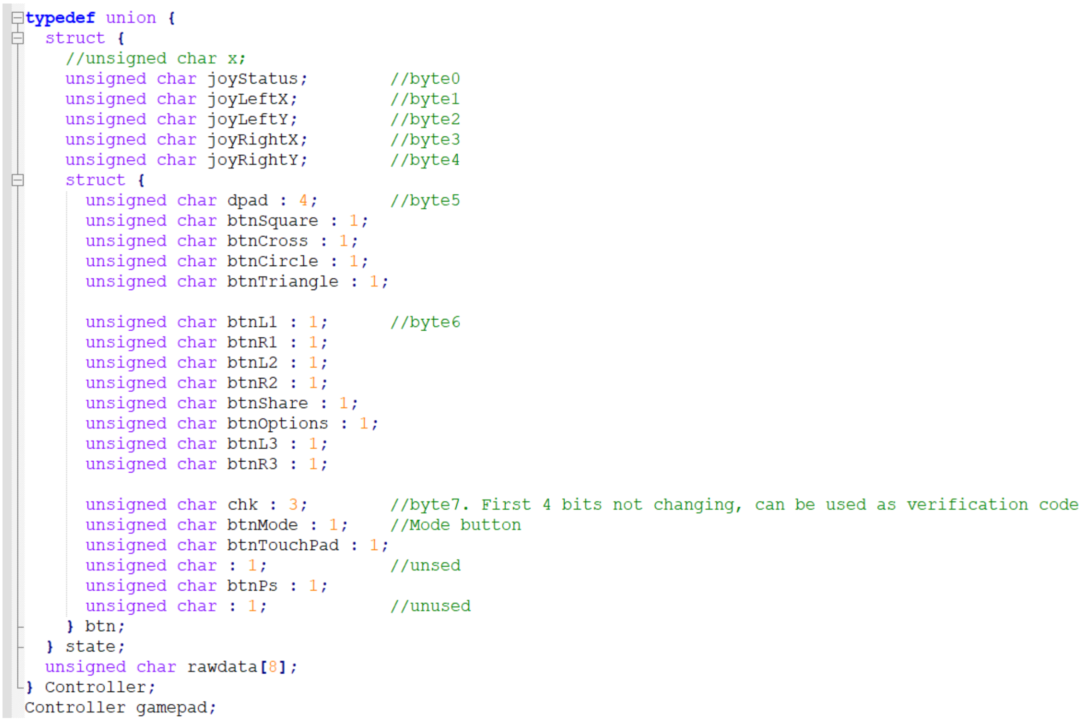

Bits: 8 8 8 8 8 4 1 1 1 1
JoyStatus
JoyLX
JoyLY
JoyRX
JoyRY
DPAD
Square
Cross
Circle
Triangle
B0 B1 B2 B3 B4 B5
Analog values of left and right Joysticks
1 Byte per axis (0x00 – 0xFFH)
DPAD/Square/Cross/Circle/Triangle => Pressed = '1'
Bits: 1 1 1 1 1 1 1 1 3 1 1 1
BtnL1
BtnR1
BtnL2
BtnR2
BtnShare
BtnOptions
BtnL3
BtnR3
Chk
Mode
TouchPad
BtnPS
B6
B7
Byte 6 buttons will return '1' when pressed

Figure 6. 15 Data Bytes that will be sent via UART

6.5.2 Processor receiving data bytes

The incoming bytes will be processed using First-In-First-Out (FIFO) method into a structure group.



```
typedef union {
  struct {
    //unsigned char x;
    unsigned char joyStatus;        //byte0
    unsigned char joyLeftX;         //byte1
    unsigned char joyLeftY;         //byte2
    unsigned char joyRightX;        //byte3
    unsigned char joyRightY;        //byte4
    struct {
      unsigned char dpad : 4;       //byte5
      unsigned char btnSquare : 1;
      unsigned char btnCross : 1;
      unsigned char btnCircle : 1;
      unsigned char btnTriangle : 1;

      unsigned char btnL1 : 1;      //byte6
      unsigned char btnR1 : 1;
      unsigned char btnL2 : 1;
      unsigned char btnR2 : 1;
      unsigned char btnShare : 1;
      unsigned char btnOptions : 1;
      unsigned char btnL3 : 1;
      unsigned char btnR3 : 1;

      unsigned char chk : 3;        //byte7. First 4 bits not changing, can be used as verification code
      unsigned char btnMode : 1;    //Mode button
      unsigned char btnTouchPad : 1;
      unsigned char : 1;            //unsed
      unsigned char btnPs : 1;
      unsigned char : 1;            //unused
    } btn;
  } state;
  unsigned char rawdata[8];
} Controller;
Controller gamepad;
```

Figure 6. 16: Code Snippet on the Data Structure to Receive Data

Since the maximum data can be transferred through UART is 128 using char variable, data will be sent between [0, 128] from the transmitter side. The data received data will be translated back into [0, 255] by the main controller.

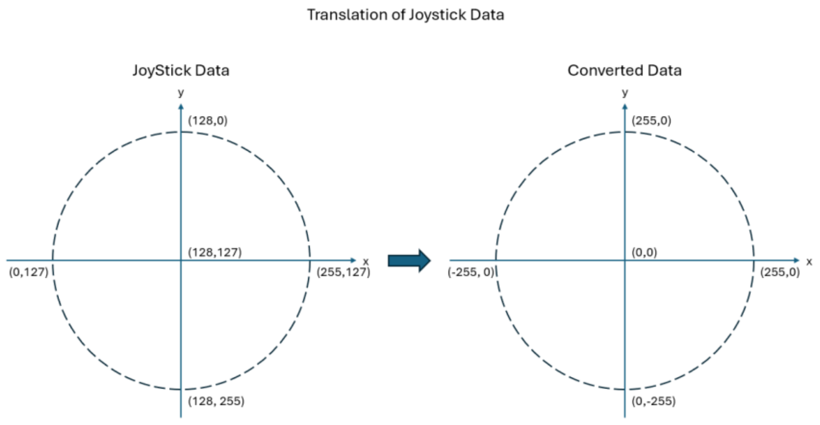
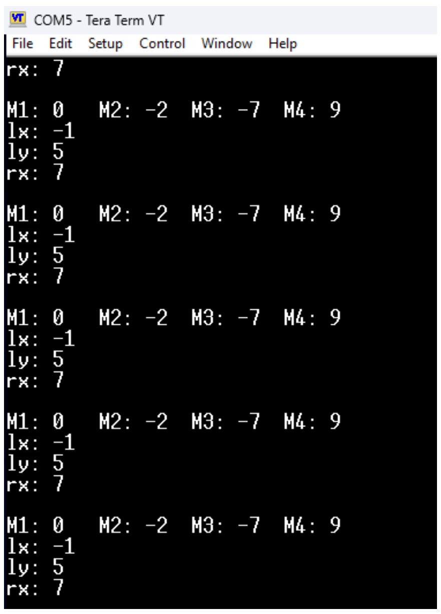


Figure 6. 17: Translation of Joystick Data

Results displayed in figure 6.12 show that the core is able to receive the inputs from the ESP32 via UART. Thus, the implementation of interfacing the DualShock 5 gamepad is successful.



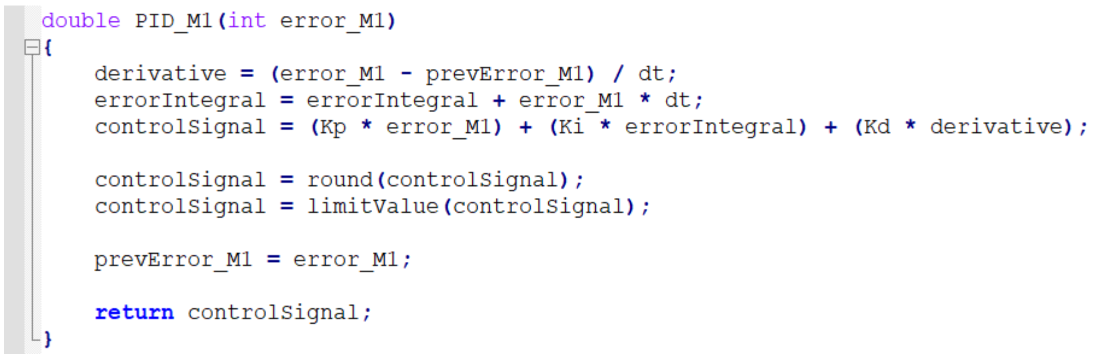
```
rx: 7

M1: 0   M2: -2  M3: -7  M4: 9
lx: -1
ly: 5
rx: 7

M1: 0   M2: -2  M3: -7  M4: 9
lx: -1
ly: 5
rx: 7

M1: 0   M2: -2  M3: -7  M4: 9
lx: -1
ly: 5
rx: 7

M1: 0   M2: -2  M3: -7  M4: 9
lx: -1
ly: 5
rx: 7

M1: 0   M2: -2  M3: -7  M4: 9
lx: -1
ly: 5
rx: 7
```

Figure 6. 18: Results obtained from Interfacing Gamepad

6.6 PID Control

The Proportional-Integral-Derivative (PID) controller implemented in the firmware is used for a feedback loop to reach a desired setpoint by minimizing the error provided by the actual measured values (Gabriel, 2016). The control signal is being computed using three components:

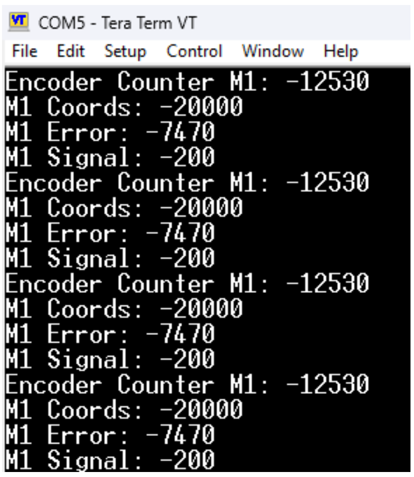
- Proportional Term (P): Produces an output that is proportional to the current error value. The proportional gain constant, Kp determines the response strength of the error difference, where the larger the error, the stronger the control action
- Integral Term (I): This term sums the error over time where it accumulated past errors which help to reduce the steady-state error. The drawbacks will be that the overall system will be sluggish.
- Derivative Term (D): This factor is applied to minimize the overshooting by having the ability to predict the error.



```
double PID_M1(int error_M1)
{
    derivative = (error_M1 - prevError_M1) / dt;
    errorIntegral = errorIntegral + error_M1 * dt;
    controlSignal = (Kp * error_M1) + (Ki * errorIntegral) + (Kd * derivative);

    controlSignal = round(controlSignal);
    controlSignal = limitValue(controlSignal);

    prevError_M1 = error_M1;

    return controlSignal;
}
```

Figure 6. 19: PID Control Implementation

The PID controller implemented in the firmware is used to create a close loop using the encoder pulse generated by the movement of the DC motor act as a position control like a servo.



```
Encoder Counter M1: -12530
M1 Coords: -20000
M1 Error: -7470
M1 Signal: -200
Encoder Counter M1: -12530
M1 Coords: -20000
M1 Error: -7470
M1 Signal: -200
Encoder Counter M1: -12530
M1 Coords: -20000
M1 Error: -7470
M1 Signal: -200
Encoder Counter M1: -12530
M1 Coords: -20000
M1 Error: -7470
M1 Signal: -200
```

Figure 6. 20: Results from Implementing PID Controller

The overall results obtained from the PID control are successful where the target goal for encoder count is set to -20000 pulse, the resulting error produced and the output signal generated are suitable to rotate a DC motor shaft to exactly where the intended goal.

6.7 Error Checking Mechanisms

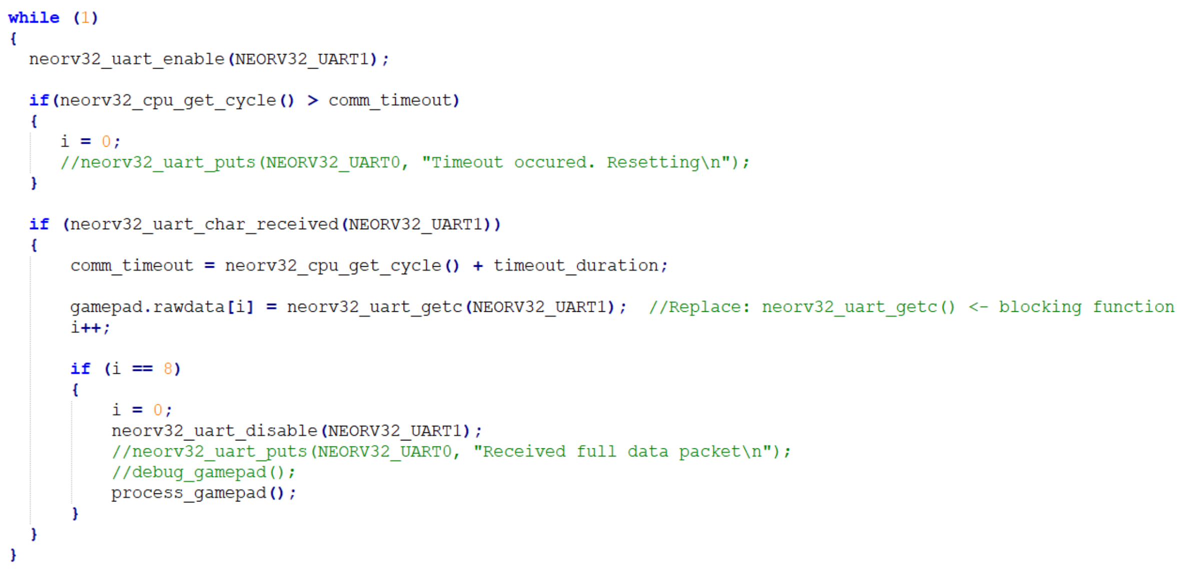
Error-checking mechanisms are added to the firmware to increase the communication reliability between two processors, the main core and ESP32. Without the implemented modules, false data was detected for example, when the controller was untouched, but the main core received random key presses.

6.7.1 Timeout Mechanism

A timeout mechanism is developed to prevent the UART from waiting for the data to wait indefinitely which will result in a slow response from the controller. The timeout mechanism uses the CPU clock cycles to count time elapsed and reset the data buffer if necessary.

The code checks the current CPU cycle count using the "neorv32_cpu_get_cycle()" and if the current cycle count exceeds the "comm_timeout" which it indicates the timeout period has occurred without receiving any data and the index buffer will resets to zero.

The “comm_timeout” is updated to the current cycle count plus the “timeout_duration” extending the timeout period if data is incoming from UART.



```
while (1)
{
  neorv32_uart_enable(NEORV32_UART1);

  if(neorv32_cpu_get_cycle() > comm_timeout)
  {
     i = 0;
     //neorv32_uart_puts(NEORV32_UART0, "Timeout occured. Resetting\n");
  }

  if (neorv32_uart_char_received(NEORV32_UART1))
  {
      comm_timeout = neorv32_cpu_get_cycle() + timeout_duration;

      gamepad.rawdata[i] = neorv32_uart_getc(NEORV32_UART1);  //Replace: neorv32_uart_getc() <- blocking function
      i++;

      if (i == 8)
      {
          i = 0;
          neorv32_uart_disable(NEORV32_UART1);
          //neorv32_uart_puts(NEORV32_UART0, "Received full data packet\n");
          //debug_gamepad();
          process_gamepad();
      }
  }
}
```

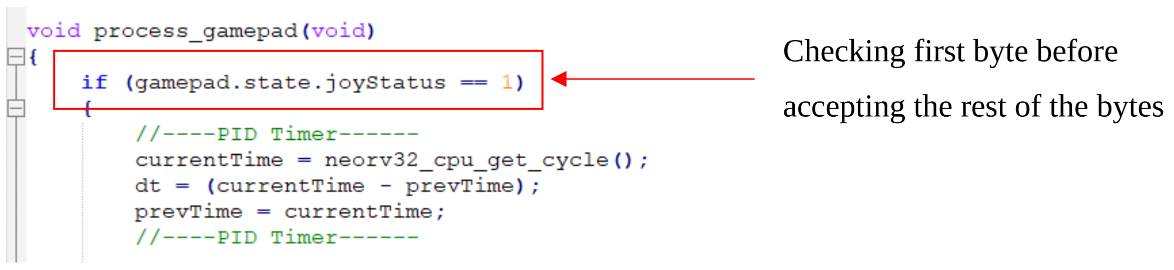
Figure 6. 21: Implementation of Timeout Mechanism

When all the 8 Bytes of data received from UART1, the program will start to process the application function, else it will wait until all 8 complete bytes of data.

6.7.2 Error Handling

To verify the integrity of the data packet received from the UART, a checksum is added. The checksum is included in the first byte of the data packet, which will be used for double-checking if the checksum is matched with the one from the transmitter. A comparison is performed and if the checksum received does not align, it will be terminated and it checks for new data packets.

The checksum is checked before accessing into the main functions where all the algorithms and data processing are being done.



```
void process_gamepad(void)
{
    if (gamepad.state.joyStatus == 1)
    {
        //----PID Timer------
        currentTime = neorv32_cpu_get_cycle();
        dt = (currentTime - prevTime);
        prevTime = currentTime;
        //----PID Timer------
```

Figure 6. 22: Implemetation of Error Handling in Code

After checksum have been implemented, the data received is very much more reliable and no faulty or missing data is received which will disrupt the overall functionality of the application

6.8 Performance Tuning

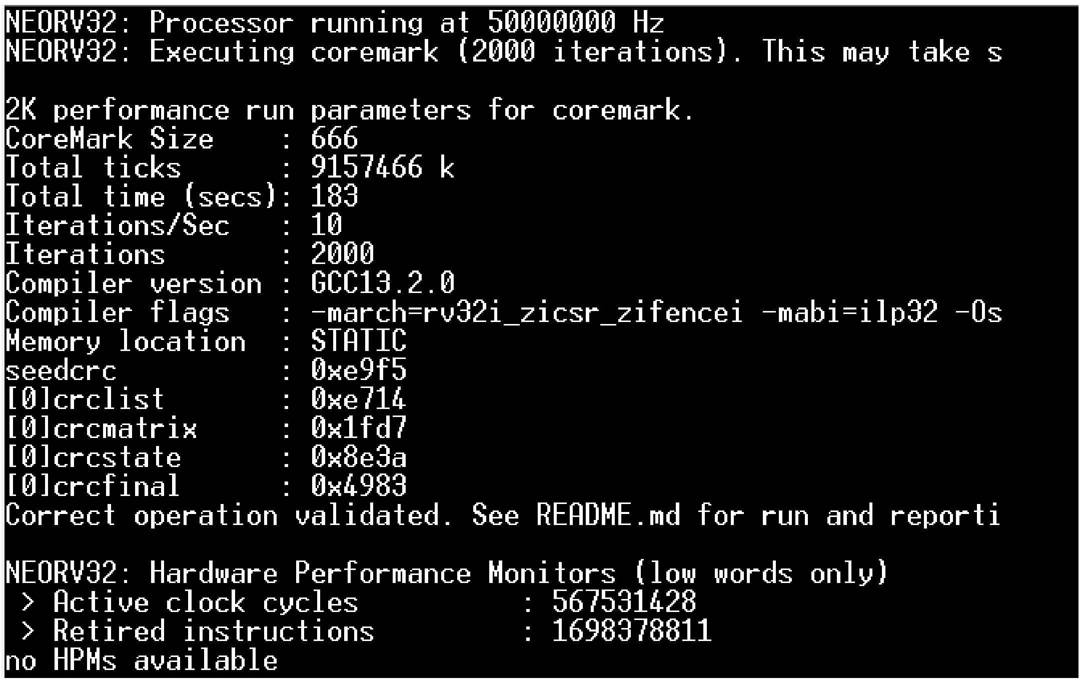
Since the application is heavily dependent on the CPU performance, it is essential to optimise the core for maximum performance. Performance can be improved by enabling dedicated hardware accelerators from the RISC-V CPU extensions which are M, C and Zfinx extensions. Besides the CPU extensions, CPU operations can be improved by enabling FAST_MUL_EN and FAST_SHIFT_EN.

Enabling the cache memory (ICACHE_EN) can further improve the performance as well as the internal memory MEM_INT_IMEM_EN, MEM_INT_DMEM_EN where it reduces the memory access latency.

The performance can be evaluated using the CoreMark, which is a benchmarking tool to test the overall processor core functionality. Setup:

- Hardware: 32kB int IMEM, 16kB int DMEM, 64kB int CACHE, 50MHz clock
- Compiler: RISC-V32-GCC 13.2.0 (march=rv32i and mabi=ilp32)

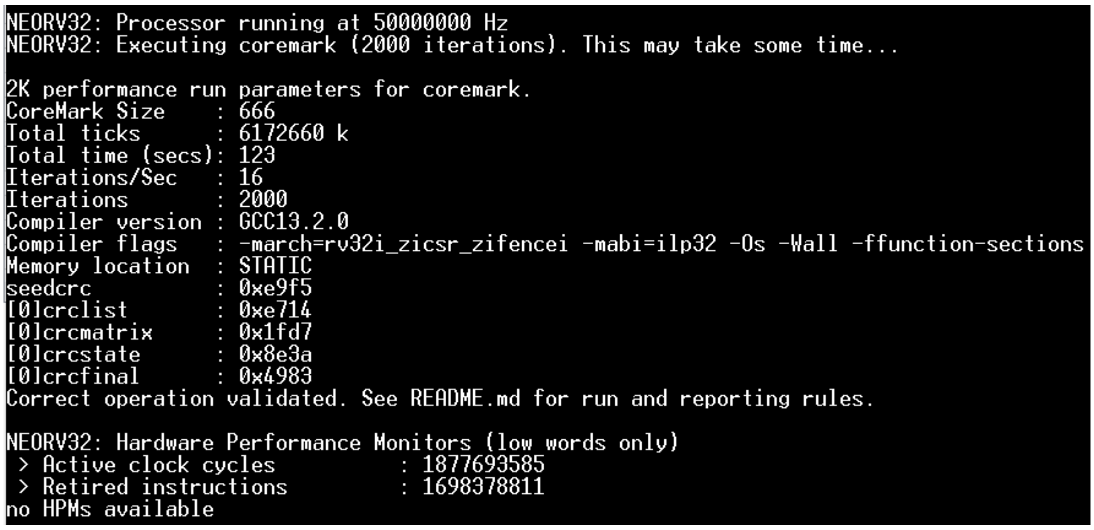
6.8.1 CoreMark results before tuning



```
NEORV32: Processor running at 50000000 Hz
NEORV32: Executing coremark (2000 iterations). This may take s

2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 9157466 k
Total time (secs): 183
Iterations/Sec   : 10
Iterations       : 2000
Compiler version : GCC13.2.0
Compiler flags   : -march=rv32i_zicsr_zifencei -mabi=ilp32 -Os
Memory location  : STATIC
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x4983
Correct operation validated. See README.md for run and reporti

NEORV32: Hardware Performance Monitors (low words only)
 > Active clock cycles          : 567531428
 > Retired instructions         : 1698378811
no HPMs available
```

Figure 6. 23: CoreMark Performance before Tuning

6.8.2 CoreMark results after tuning



```
NEORV32: Processor running at 50000000 Hz
NEORV32: Executing coremark (2000 iterations). This may take some time...

2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 6172660 k
Total time (secs): 123
Iterations/Sec   : 16
Iterations       : 2000
Compiler version : GCC13.2.0
Compiler flags   : -march=rv32i_zicsr_zifencei -mabi=ilp32 -Os -Wall -ffunction-sections
Memory location  : STATIC
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x4983
Correct operation validated. See README.md for run and reporting rules.

NEORV32: Hardware Performance Monitors (low words only)
 > Active clock cycles          : 1877693585
 > Retired instructions         : 1698378811
no HPMs available
```

Figure 6. 24: CoreMark Performance after Tuning

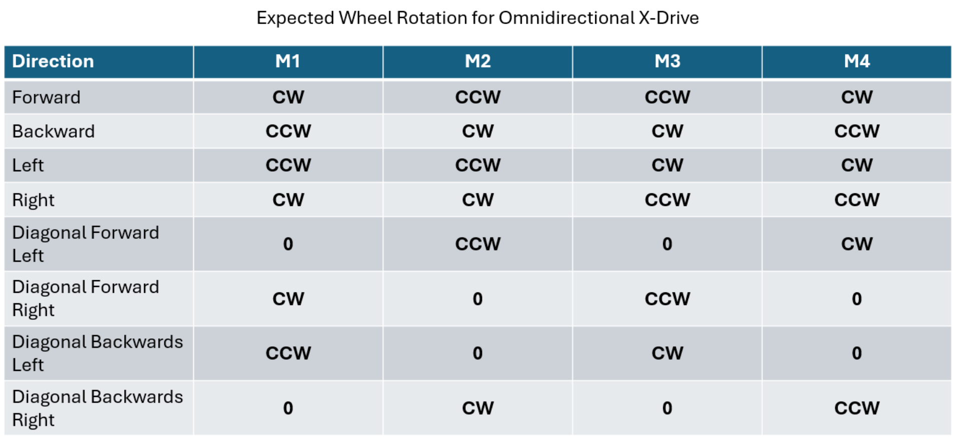
Based on the comparison, the performance difference has increased by 62.5% which will help the overall performance of the system.

6.9 Results

All the modules and functionalities implemented in the firmware are able to execute successfully without any issues. The program can be separated into 2 sections which are manual movement which the mobile robot able to move freely using the gamepad, and predefined motion where the mobile robot will move according to the predefined setpoint based on the encoder pulses. The processor also capable of switching between manual control and pre-defined path control with a press of a gamepad button without any issues or latency which makes the processor very reliable and robust.

6.9.1 Manual Movement

Based on section 6.4 and section 6.5, the expected outcomes are achieved, where the data from the gamepad able to produce accurate wheel rotation based on the inverse kinematics derived. Which makes the processor capable of running the calculations without any errors or freezing of the application.



Expected Wheel Rotation for Omnidirectional X-Drive

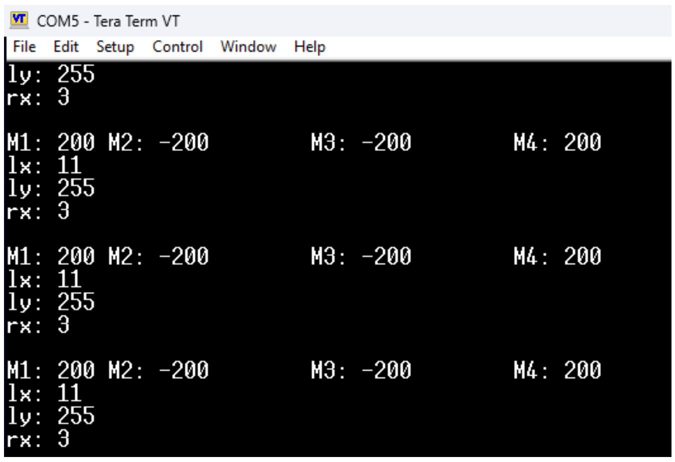
Direction	M1	M2	M3	M4
Forward	CW	CCW	CCW	CW
Backward	CCW	CW	CW	CCW
Left	CCW	CCW	CW	CW
Right	CW	CW	CCW	CCW
Diagonal Forward Left	0	CCW	0	CW
Diagonal Forward Right	CW	0	CCW	0
Diagonal Backwards Left	CCW	0	CW	0
Diagonal Backwards Right	0	CW	0	CCW

Figure 6. 25 Expected Wheel Rotation for X-drive drivetrain

CW: Wheels moving clockwise and will return positive value after calculating kinematics

CCW: Wheels moving counterclockwise and will return negative value after calculating kinematics

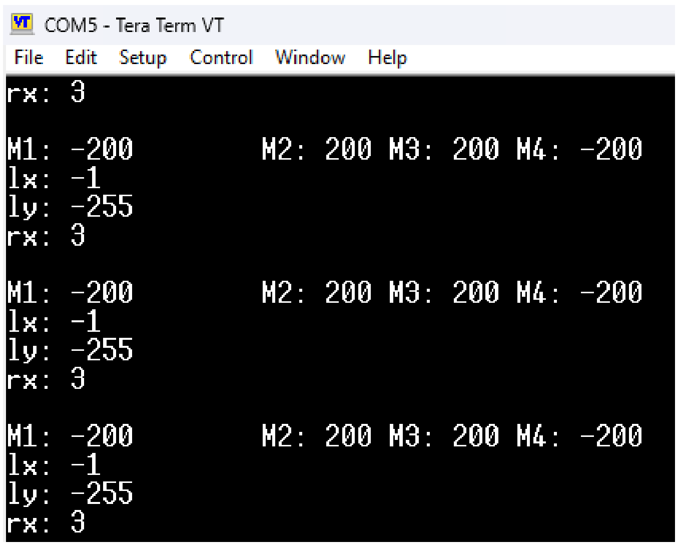
Case 1: when the joystick is pushed forward



```
ly: 255
rx: 3

M1: 200 M2: -200        M3: -200        M4: 200
lx: 11
ly: 255
rx: 3

M1: 200 M2: -200        M3: -200        M4: 200
lx: 11
ly: 255
rx: 3

M1: 200 M2: -200        M3: -200        M4: 200
lx: 11
ly: 255
rx: 3
```

Figure 6. 26: Result when joystick pushed forward

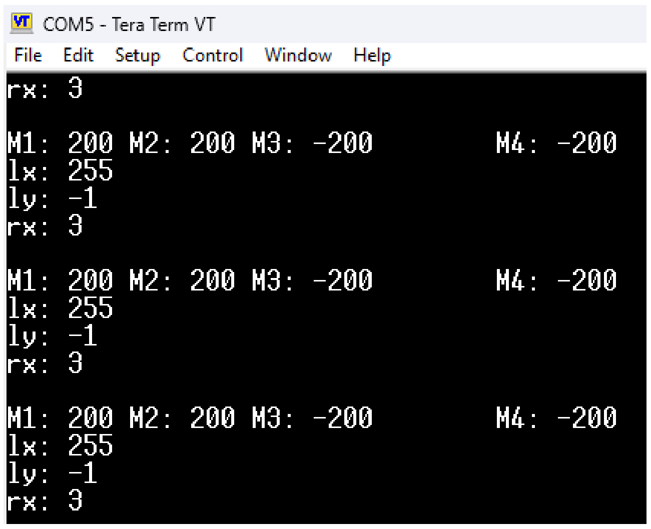
Case 2: when the joystick is pushed backwards



```
rx: 3

M1: -200        M2: 200 M3: 200 M4: -200
lx: -1
ly: -255
rx: 3

M1: -200        M2: 200 M3: 200 M4: -200
lx: -1
ly: -255
rx: 3

M1: -200        M2: 200 M3: 200 M4: -200
lx: -1
ly: -255
rx: 3
```

Figure 6. 27: Result when joystick pushed backwards

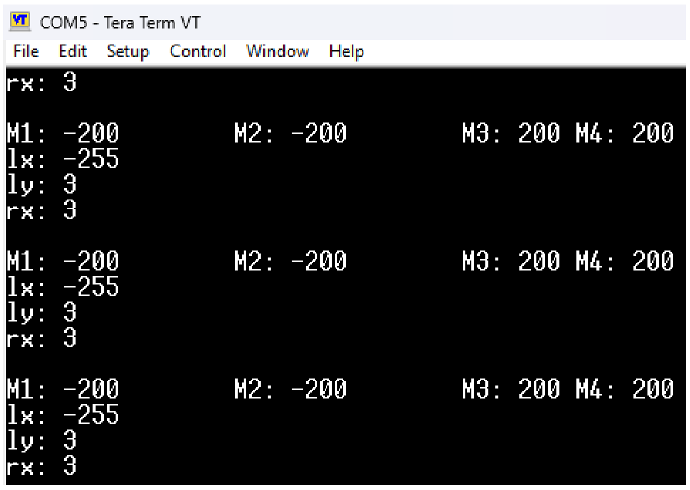
Case 3: when the joystick is pushed right



```
rx: 3

M1: 200 M2: 200 M3: -200        M4: -200
lx: 255
ly: -1
rx: 3

M1: 200 M2: 200 M3: -200        M4: -200
lx: 255
ly: -1
rx: 3

M1: 200 M2: 200 M3: -200        M4: -200
lx: 255
ly: -1
rx: 3
```

Figure 6. 28: Result when joystick pushed right

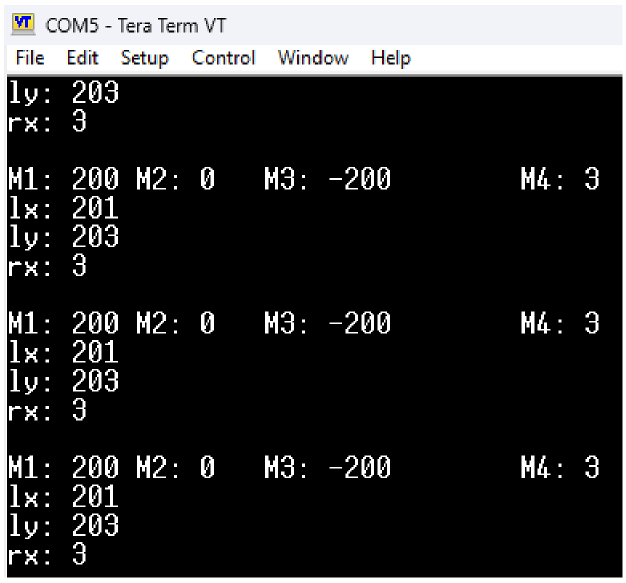
Case 4: when the joystick is pushed left



```
rx: 3

M1: -200        M2: -200        M3: 200 M4: 200
lx: -255
ly: 3
rx: 3

M1: -200        M2: -200        M3: 200 M4: 200
lx: -255
ly: 3
rx: 3

M1: -200        M2: -200        M3: 200 M4: 200
lx: -255
ly: 3
rx: 3
```

Figure 6. 29: Result when joystick pushed left

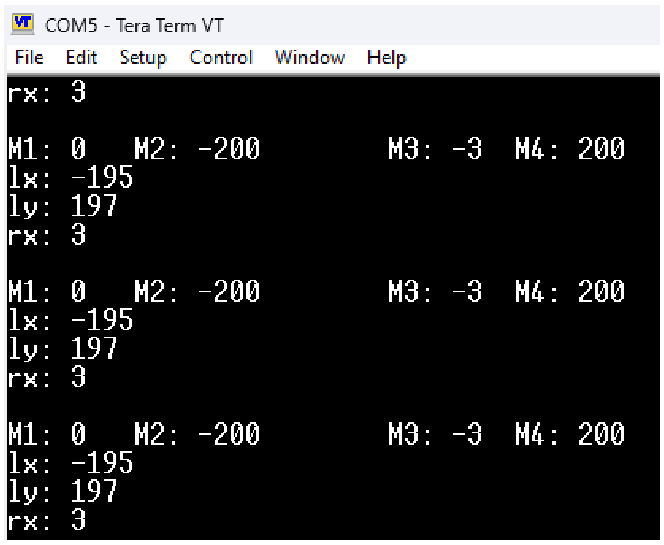
Case 5: when the joystick is pushed forward right



```
ly: 203
rx: 3

M1: 200 M2: 0   M3: -200        M4: 3
lx: 201
ly: 203
rx: 3

M1: 200 M2: 0   M3: -200        M4: 3
lx: 201
ly: 203
rx: 3

M1: 200 M2: 0   M3: -200        M4: 3
lx: 201
ly: 203
rx: 3
```

Figure 6. 30: Result when joystick pushed forward right

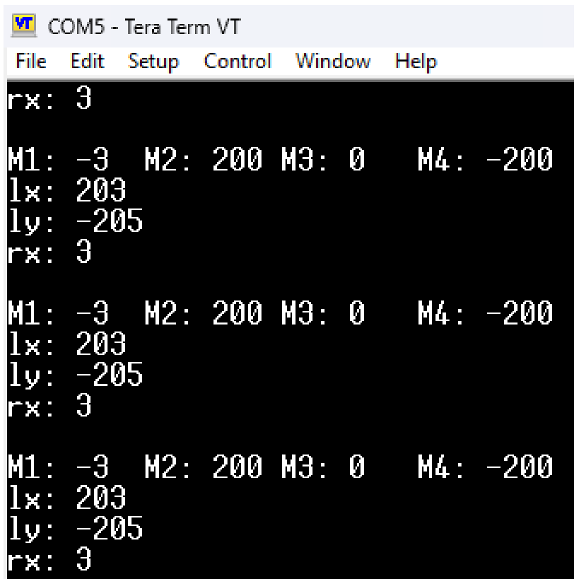
Case 6: when the joystick is pushed forward left



```
rx: 3

M1: 0   M2: -200        M3: -3  M4: 200
lx: -195
ly: 197
rx: 3

M1: 0   M2: -200        M3: -3  M4: 200
lx: -195
ly: 197
rx: 3

M1: 0   M2: -200        M3: -3  M4: 200
lx: -195
ly: 197
rx: 3
```

Figure 6. 31: Result when joystick pushed forward left

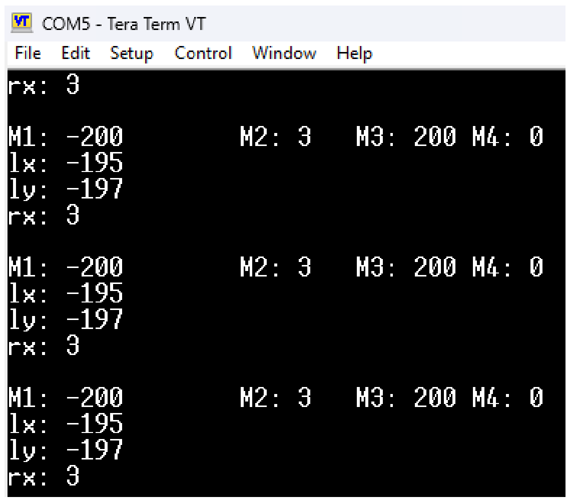
Case 7: when the joystick is pushed backwards right



```
rx: 3

M1: -3  M2: 200 M3: 0   M4: -200
lx: 203
ly: -205
rx: 3

M1: -3  M2: 200 M3: 0   M4: -200
lx: 203
ly: -205
rx: 3

M1: -3  M2: 200 M3: 0   M4: -200
lx: 203
ly: -205
rx: 3
```

Figure 6. 32: Result when joystick pushed backwards right

Case 8: when the joystick is pushed backwards left



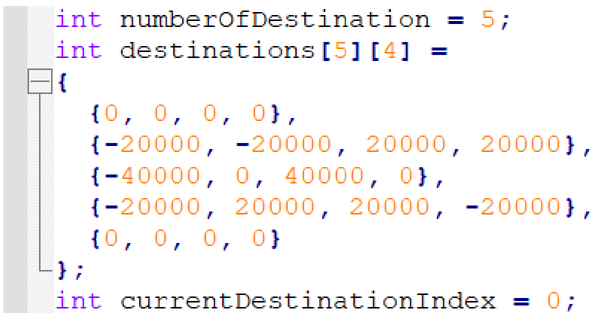
```
rx: 3

M1: -200        M2: 3   M3: 200 M4: 0
lx: -195
ly: -197
rx: 3

M1: -200        M2: 3   M3: 200 M4: 0
lx: -195
ly: -197
rx: 3

M1: -200        M2: 3   M3: 200 M4: 0
lx: -195
ly: -197
rx: 3
```

Figure 6. 33: Result when joystick pushed backwards left

All the data presented are aligned with the expected wheel rotation for the X-drive drivetrain. The mobile robot is able to move in all directions without turning its axis, which makes it a true holonomic robot. This shows that the inverse kinematics can be deployed into the software and guaranteed to work.

6.9.2 Pre-defined Movement using Encoder

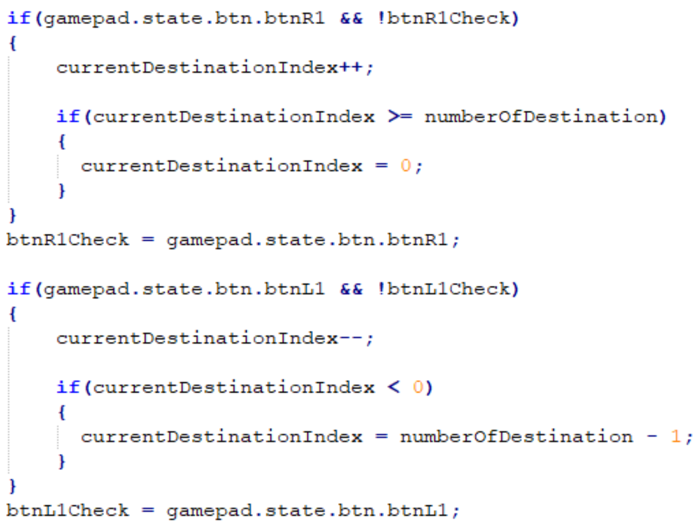
The pre-defined movement is possible by utilizing the encoder from the DC motor. The pulse generated by the encoder will be used as feedback to the PID control. The current test pattern will be a square motion. The target encoder count will be stored in a 2D array for each destination and the required counts need to be moved.



```
int numberOfDestination = 5;
int destinations[5][4] =
{
    {0, 0, 0, 0},
    {-20000, -20000, 20000, 20000},
    {-40000, 0, 40000, 0},
    {-20000, 20000, 20000, -20000},
    {0, 0, 0, 0}
};
int currentDestinationIndex = 0;
```

Figure 6. 34: Pre-defined setpoints for each motor encoder

The firmware also enables to cycle back and forth from its current location to next or previous setpoint since the data of the encoder stored in an array. The implementation will be in the code snippet in figure 6.27.

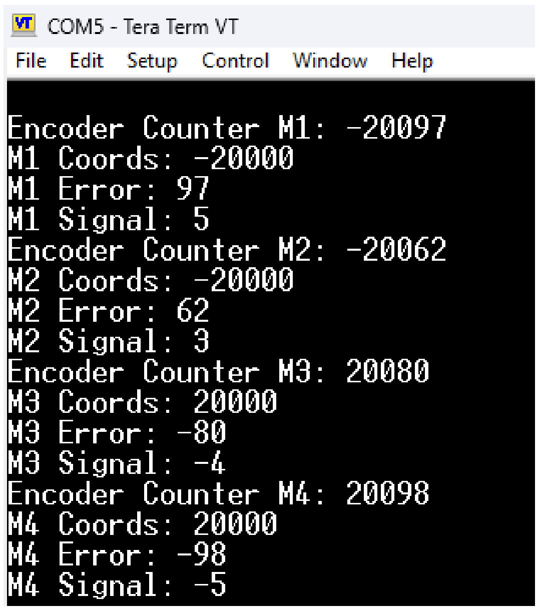


```
if(gamepad.state.btn.btnR1 && !btnR1Check)
{
    currentDestinationIndex++;

    if(currentDestinationIndex >= numberOfDestination)
    {
      currentDestinationIndex = 0;
    }
}
btnR1Check = gamepad.state.btn.btnR1;

if(gamepad.state.btn.btnL1 && !btnL1Check)
{
    currentDestinationIndex--;

    if(currentDestinationIndex < 0)
    {
      currentDestinationIndex = numberOfDestination - 1;
    }
}
btnL1Check = gamepad.state.btn.btnL1;
```

Figure 6. 35: Code implementation on cycling through setpoints

Two buttons from the controller which is right button, R1 and left button, L1 to move to next or previous location respectively.

The mobile robot able to execute the pre-defined movement without any issues and smoothly. The RISC-V core able to obtain the encoders data and compute all 4 respective PID control functions for each motor as well as generating and applying the control signal to the motor drivers.



```
Encoder Counter M1: -20097
M1 Coords: -20000
M1 Error: 97
M1 Signal: 5
Encoder Counter M2: -20062
M2 Coords: -20000
M2 Error: 62
M2 Signal: 3
Encoder Counter M3: 20080
M3 Coords: 20000
M3 Error: -80
M3 Signal: -4
Encoder Counter M4: 20098
M4 Coords: 20000
M4 Error: -98
M4 Signal: -5
```

Figure 6. 36: Generated output signals from PID control

CHAPTER 7 CONCLUSIONS

7.1 Summary and Conclusions

This project successfully integrated the NEORV32 RISC-V processor onto an FPGA platform, aiming to enhance educational and research capabilities in computer architecture. The implementation process involved setting up a development environment on Ubuntu, configuring the FPGA, and verifying the processor’s functionality through various applications. The NEORV32 processor was selected for its open-source nature, configurability, and robustness.

Key features of the implemented processor include efficient resource utilization, scalability, and robust performance. Applications such as a quadrature encoder and PID control for a holonomic drivetrain were used to validate the processor's capabilities, demonstrating its effectiveness in handling complex tasks.

1. Processor Integration and Functionality
   The NEORV32 processor was successfully integrated into the FPGA, with all functionalities performing as expected. The processor effectively handled both simple and complex tasks, confirming its operational reliability.
2. Application Performance
   The processor’s performance was validated through applications such as the quadrature encoder and PID control for the holonomic drivetrain. These applications ran smoothly, showcasing the processor’s robustness and reliability.
3. Scalability and Configurability
   The NEORV32 processor's high configurability allows for future enhancements and scalability. It can accommodate additional peripherals and functionalities, making it a versatile tool for ongoing developments and research.

4. Educational and Research Potential

   The implementation serves as a valuable educational platform, practical tool for studying computer architecture and FPGA-based processor design. Its open-source nature facilitates extensive customization and experimentation.

The project successfully demonstrated the integration of the NEORV32 RISC-V processor on an FPGA platform, highlighting its potential as a powerful and flexible system for educational and research applications. The processor's efficient resource utilization, robust performance, and scalability make it suitable for a wide range of applications. This project highlights the advantages of using open-source RISC-V processors in FPGA environments, setting the stage for future developments and research.

7.2 Areas of Future Research

The RISC-V core can be further developed to accommodate more peripherals by expanding the available GPIO ports. For example, currently the core has 2 UART interfaces but it can be further increased which will be helpful for multiple applications like sensor nodes since the designer has full control of the core.

Besides that, integrating a neural network for example Convolutional Neural Network (CNN) would be possible using the custom function subsystem to perform image recognition and processing.

REFERENCES

Ahmadi-Pour, S., Herdt, V., & Drechsler, R. (2022). The MicroRV32 framework: An accessible and configurable open source RISC-V cross-level platform for education and research. *J. Syst. Archit.*, *133*. https://doi.org/10.1016/j.sysarc.2022.102757

Altera. (2016). De2i-150 FPGA Getting Started Guide. Manual. https://www.secs.oakland.edu/~llamocca/Tutorials/Emb_Int

Andrew Shell Waterman., (2016). Design of the RISC-V Instruction Set Architecture. University of California, Berkeley https://escholarship.org/uc/item/7zj0b3m7

Andrew Waterman. (2015). RISC-V Software Ecosystem [Software]. https://riscv.org/wp-content/uploads/2015/01/riscv-software-toolchain-workshop-jan2015.pdf

Craigloewen. (2023). How to install Linux on Windows with WSL. Microsoft. https://learn.microsoft.com/en-us/windows/wsl/install

Gabriel Găşpăresc. (2016). PID control of a DC motor using Labview Interface for Embedded Platforms. 2016 12th IEEE International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania. 145-148. https://doi.org/10.1109/ISETC.2016.7781078

Harris, Sarah & Chaver, Daniel & Pinuel, Luis & Gomez, J.I. & Liaqat, M. & Kakakhel, Zubair & Kindgren, Olof & Owen, Robert. (2021). RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. 145-150. https://ieeexplore.ieee.org/document/9556406

Höller, R., Haselberger, D., Ballek, D., Rössler, P., Krapfenbauer, M., & Linauer, M. (2019). Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation. 2019 8th Mediterranean Conference on Embedded Computing (MECO), 1-6. https://ieeexplore.ieee.org/document/8760205

Nihal Kularatna. (2000). Modern Component Families and Circuit Block Design. 197-239. https://doi.org/10.1016/B978-075069992-1/50006-1

Pawks., Neelgala., bilalsakhawat., lavanyajagan. (2019). RISCOF [Software]. https://github.com/riscv-software-src/riscof

S. Sarkar and K. Sivayazi. (2018). A New Decoding Logic for Quadrature Encoder Interfacing for Software Implementation on FPGA. *2018 3rd International Conference for Convergence in Technology (I2CT)*, Pune, India, 2018, pp. 1-4. https://doi.org/10.1109/I2CT.2018.8529379

Saif, Md & Sadad, Nahin & Mondal, Md. (2023). FPGA Implementation of Educational RISC- V Processor Suitable for Embedded Applications. 1-5. https://ieeexplore.ieee.org/document/10101508

Stephan Nolting. (2020). The NEORV32 RISC-V PROCESSOR [Software]. https://stnolting.github.io/neorv32/

Stephen Nolting. (2024). Prebuilt RISC-V GCC Toolchain for Linux. GitHub. https://github.com/stnolting/riscv-gcc-prebuilt

stnolting. (2020). NEORV32 Core Verification using RISCOF [Software]. https://github.com/stnolting/neorv32-riscof

APPENDIX

Code

```
#include <neorv32.h>
#include <neorv32_rte.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>

/**********************************************************************//**
 * @name User configuration
 **************************************************************************/
/**@{*/
/** UART BAUD rate */
#define BAUD_RATE 115200
/**@}*/

neorv32_uart_t *UART0 = NEORV32_UART0;
neorv32_uart_t *UART1 = NEORV32_UART1;

typedef union {
  struct {
        //unsigned char x;
    unsigned char joyStatus;        //byte0
    unsigned char joyLeftX;         //byte1
    unsigned char joyLeftY;         //byte2
    unsigned char joyRightX;        //byte3
    unsigned char joyRightY;        //byte4
    struct {
      unsigned char dpad : 4;       //byte5
      unsigned char btnSquare : 1;
      unsigned char btnCross : 1;
      unsigned char btnCircle : 1;
      unsigned char btnTriangle : 1;

      unsigned char btnL1 : 1;      //byte6
      unsigned char btnR1 : 1;
      unsigned char btnL2 : 1;
      unsigned char btnR2 : 1;
      unsigned char btnShare : 1;
      unsigned char btnOptions : 1;
      unsigned char btnL3 : 1;
      unsigned char btnR3 : 1;
       //byte7. First 4 bits no changing, can be used as verification code
      unsigned char chk : 3
      unsigned char btnMode : 1;    //Mode button
      unsigned char btnTouchPad : 1;
      unsigned char : 1;            //unsed
      unsigned char btnPs : 1;
      unsigned char : 1;            //unused
    } btn;
  } state;
  unsigned char rawdata[8];
} Controller;
Controller gamepad;

//-------------Function Prototype------------------------
double PID_M1(int error_M1);
double PID_M2(int error_M2);
double PID_M3(int error_M3);
double PID_M4(int error_M4);
void debug_gamepad(void);
void process_gamepad(void);
double limitValue(double value);
int map(int x, int in_min, int in_max, int out_min, int out_max);
//-------------Function Prototype------------------------

//---------Timeout Mechanism------------
unsigned int i = 0;
unsigned long long comm_timeout = 0;
unsigned long long timeout_duration = 0;
//---------Timeout Mechanism------------

//---------Motor Driver-----------------
int MAX_SPEED = 200;
double theta = 0, mag = 0;
int lx = 0, ly = 0, rx = 0;
double M1 = 0, M2 = 0, M3 = 0, M4 = 0;

#define ROTATION_SPEED_FACTOR 2.55
//---------Motor Driver-----------------

//---------PID-------------
double controlSignal = 0;
double Kp = 0.05, Ki = 0.00, Kd = 25.00;
double prevTime = 0, currentTime = 0, dt = 0;
double prevError = 0, errorIntegral = 0, derivative = 0;

int M1_Error = 0, M2_Error = 0, M3_Error = 0, M4_Error = 0;
double M1_Speed = 0, M2_Speed = 0, M3_Speed = 0, M4_Speed = 0;
int M1_Coords = 0, M2_Coords = 0, M3_Coords = 0, M4_Coords = 0;
double prevError_M1 = 0, prevError_M2 = 0, prevError_M3 = 0, prevError_M4 = 0;

int numberOfDestination = 5;
int destinations[5][4] =
{
  {0, 0, 0, 0},
  {-20000, -20000, 20000, 20000},
  {-40000, 0, 40000, 0},
  {-20000, 20000, 20000, -20000},
  {0, 0, 0, 0}
};
int currentDestinationIndex = 0;
//---------PID-------------

/* MOVE IN SQUARE PATTERN
  {0, 0, 0, 0},
  {-20000, -20000, 20000, 20000},
  {-40000, 0, 40000, 0},
  {-20000, 20000, 20000, -20000},
  {0, 0, 0, 0}
*/

//---------Controller----------
bool Speed = 0;
int currentMode = 0;
bool btnL1Check = false;
bool btnR1Check = false;
bool btnPsCheck = false;
bool btnOptionsCheck = false;
//---------Controller----------

//-----DEBUG-----
bool debugAuto = false;
bool debugManual = false;
//-----DEBUG-----

int main()
{
  // capture all exceptions and give debug info via UART
  neorv32_rte_setup();
  // setup UART0 at default baud rate, no interrupts
  neorv32_uart_setup(NEORV32_UART0, BAUD_RATE, 0);
  // setup UART1 at default baud rate, no interrupts
  neorv32_uart_setup(NEORV32_UART1, BAUD_RATE, 0);
  neorv32_uart_rtscts_enable(NEORV32_UART1);

  // check if UART0 is implemented at all
  if (neorv32_uart_available(NEORV32_UART0) == 0)
  {
    neorv32_uart_puts(NEORV32_UART0, "Error! UART0 not synthesized!\n");
    return 1;
  }
```

```
// check if UART1 is implemented at all
  if (neorv32_uart_available(NEORV32_UART1) == 0)
  {
    neorv32_uart_puts(NEORV32_UART0, "Error! UART1 not synthesized!\n");
    return 1;
  }

  // check if PWM unit is implemented at all
  if (neorv32_pwm_available() == 0)
  {
    if (neorv32_uart0_available())
    {
      neorv32_uart0_printf("ERROR: PWM module not implemented!\n");
    }
    return 1;
  }

  int num_pwm_channels = neorv32_pmw_get_num_channels();
  int j;
  for (j=0; j<num_pwm_channels; j++) {
    neorv32_pwm_set(j, 0);
  }

// Configure and enable PWM
// PWM Frequency = fmain(Hz) / 2^8 * clk_prescaler (2, 4, 8, 64, 1024, 2048, 4096)
  neorv32_pwm_setup(CLK_PRSC_8);

  // check if CFS is implemented at all
  if (neorv32_cfs_available() == 0)
  {
    neorv32_uart0_printf("Error! No CFS synthesized!\n");
    return 1;
  }

  //Clears the struct data at start of program
  memset(&gamepad, 0, sizeof(Controller));

  timeout_duration = 10000; //5209

  neorv32_uart_enable(NEORV32_UART0);
  neorv32_uart_disable(NEORV32_UART1);

  while (1)
  {
        neorv32_uart_enable(NEORV32_UART1);

        if(neorv32_cpu_get_cycle() > comm_timeout)
        {
           i = 0;
           //neorv32_uart_puts(NEORV32_UART0, "Timeout occured. Resetting\n");
        }

    if (neorv32_uart_char_received(NEORV32_UART1))
    {
        comm_timeout = neorv32_cpu_get_cycle() + timeout_duration;

        gamepad.rawdata[i] = neorv32_uart_getc(NEORV32_UART1);
        i++;

                if (i == 8)
                {
                 i = 0;
                 neorv32_uart_disable(NEORV32_UART1);

                 //debug_gamepad();
                 process_gamepad();
                }
    }
  }
  return 0;
}

void process_gamepad(void)
{
        if (gamepad.state.joyStatus == 1)
        {
                //----PID Timer------
                currentTime = neorv32_cpu_get_cycle();
                dt = (currentTime - prevTime);
                prevTime = currentTime;
                //----PID Timer------

                if(gamepad.state.btn.btnOptions && !btnOptionsCheck)
                {
                        if(Speed == 0)
                        {
                           MAX_SPEED = 200;
                           Speed = 1;
                        }
                        else
                        {
                            MAX_SPEED = 80;
                            Speed = 0;
                        }
                }
                btnOptionsCheck = gamepad.state.btn.btnOptions;
```

```
//Change between Manual mode or Pre-defined mode
      if(gamepad.state.btn.btnPs && !btnPsCheck)
      {
              if(currentMode == 0)
              {
                  currentMode = 1;
              }
              else
              {
                  currentMode = 0;
              }
      }
      btnPsCheck = gamepad.state.btn.btnPs;

      if(currentMode == 0)
      {
         lx = map(gamepad.state.joyLeftX, 0, 255, -255, 255);
         ly = map(gamepad.state.joyLeftY, 0, 255, 255, -255);
         rx = map(gamepad.state.joyRightX, 0, 255, -255, 255);

         //Calculates the angle for each quadrant
         //based on the joystick position
         theta = atan2f(ly, lx);
        //Calculates the magnitude to increase speed factor
         mag = sqrt((lx*lx)+(ly*ly));

         M1 = mag * cos(theta - 3.14159/4) - rx/ROTATION_SPEED_FACTOR;
         M2 = mag * -sin(theta - 3.14159/4) + rx/ROTATION_SPEED_FACTOR;
         M3 = mag * -cos(theta - 3.14159/4) - rx/ROTATION_SPEED_FACTOR;
         M4 = mag * sin(theta - 3.14159/4) + rx/ROTATION_SPEED_FACTOR;

         //Speed multiplier
         M1 = M1 * 1.25;
         M2 = M2 * 1.25;
         M3 = M3 * 1.25;
         M4 = M4 * 1.25;

         M1 = round(M1);
         M2 = round(M2);
         M3 = round(M3);
         M4 = round(M4);

         M1 = limitValue(M1);
         M2 = limitValue(M2);
         M3 = limitValue(M3);
         M4 = limitValue(M4);
```

```
    if(M1 > 0) neorv32_gpio_pin_clr(1);     // LOW
    else neorv32_gpio_pin_set(1);           // HIGH

    if(M2 > 0) neorv32_gpio_pin_set(2);     // HIGH
    else neorv32_gpio_pin_clr(2);           // LOW

    if(M3 > 0) neorv32_gpio_pin_clr(3);     // LOW
    else neorv32_gpio_pin_set(3);           // HIGH

    if(M4 > 0) neorv32_gpio_pin_set(4);     // HIGH
    else neorv32_gpio_pin_clr(4);           // LOW

    neorv32_pwm_set(0, abs((int)(M1)));
    neorv32_pwm_set(1, abs((int)(M2)));
    neorv32_pwm_set(2, abs((int)(M3)));
    neorv32_pwm_set(3, abs((int)(M4)));

    //-----DEBUG-----
    if(debugManual)
     {
        neorv32_uart_printf(NEORV32_UART0, "lx: %d\n", lx);
        neorv32_uart_printf(NEORV32_UART0, "ly: %d\n", ly);
        neorv32_uart_printf(NEORV32_UART0, "rx: %d\n\n", rx);

        neorv32_uart_printf(NEORV32_UART0, "M1: %d\t", (int)M1);
        neorv32_uart_printf(NEORV32_UART0, "M2: %d\t", (int)M2);
        neorv32_uart_printf(NEORV32_UART0, "M3: %d\t", (int)M3);
        neorv32_uart_printf(NEORV32_UART0, "M4: %d\t\n", (int)M4);
     }
}
else
{

    if(gamepad.state.btn.btnR1 && !btnR1Check)
     {
        currentDestinationIndex++;
        //cycles back if reach final destination

        if(currentDestinationIndex >= numberOfDestination)
        {
          currentDestinationIndex = 0;
        }
     }
     btnR1Check = gamepad.state.btn.btnR1;

     if(gamepad.state.btn.btnL1 && !btnL1Check)
     {
        currentDestinationIndex--;
        //cycles back if reach final destination
        if(currentDestinationIndex < 0)
        {
            currentDestinationIndex = numberOfDestination - 1;
        }
  }
  btnL1Check = gamepad.state.btn.btnL1;

  //Gets the Setpoint data for 4 motors
  M1_Coords = destinations[currentDestinationIndex][0];
  M2_Coords = destinations[currentDestinationIndex][1];
  M3_Coords = destinations[currentDestinationIndex][2];
  M4_Coords = destinations[currentDestinationIndex][3];

//NEORV32_CFS->REG[x], x = 0, 1, 2 & 3
//reads the register that contains the encoder count for each motor
//Error = Setpoint (Reference) - Actual
  M1_Error = M1_Coords - NEORV32_CFS->REG[0];
  M2_Error = M2_Coords - NEORV32_CFS->REG[1];
  M3_Error = M3_Coords - NEORV32_CFS->REG[2];
  M4_Error = M4_Coords - NEORV32_CFS->REG[3];

//Returns Output signal from PID algorithm
  M1_Speed = PID_M1(M1_Error);
  M2_Speed = PID_M2(M2_Error);
  M3_Speed = PID_M3(M3_Error);
  M4_Speed = PID_M4(M4_Error);

  if(M1_Speed >= 0) neorv32_gpio_pin_set(1);      // HIGH
  else neorv32_gpio_pin_clr(1);                           // LOW

  if(M2_Speed >= 0) neorv32_gpio_pin_set(2);      // HIGH
  else neorv32_gpio_pin_clr(2);                           // LOW

  if(M3_Speed >= 0) neorv32_gpio_pin_set(3);      // HIGH
  else neorv32_gpio_pin_clr(3);                           // LOW

  if(M4_Speed >= 0) neorv32_gpio_pin_set(4);      // HIGH
  else neorv32_gpio_pin_clr(4);                           // LOW

  neorv32_pwm_set(0, abs((int)(M1_Speed)));//Generate PWM Motor 1
  neorv32_pwm_set(1, abs((int)(M2_Speed)));//Generate PWM Motor 2
  neorv32_pwm_set(2, abs((int)(M3_Speed)));//Generate PWM Motor 3
  neorv32_pwm_set(3, abs((int)(M4_Speed)));//Generate PWM Motor 4

  //-----DEBUG-----
  if(debugAuto)
  {
```

```
                        neorv32_uart_printf(NEORV32_UART0, "Encoder Counter M1: %d\n",
                        NEORV32_CFS->REG[0]);
                        neorv32_uart_printf(NEORV32_UART0, "M1 Coords: %d\n", M1_Coords);
                        neorv32_uart_printf(NEORV32_UART0, "M1 Error: %d\n", M1_Error);
                        neorv32_uart_printf(NEORV32_UART0, "M1 Signal: %d\n",
                        (int)M1_Speed);

                        neorv32_uart_printf(NEORV32_UART0, "Encoder Counter M2: %d\n",
                        NEORV32_CFS->REG[1]);
                        neorv32_uart_printf(NEORV32_UART0, "M2 Coords: %d\n", M2_Coords);
                        neorv32_uart_printf(NEORV32_UART0, "M2 Error: %d\n", M2_Error);
                        neorv32_uart_printf(NEORV32_UART0, "M2 Signal: %d\n",
                        (int)M2_Speed);

                        neorv32_uart_printf(NEORV32_UART0, "Encoder Counter M3: %d\n",
                        NEORV32_CFS->REG[2]);
                        neorv32_uart_printf(NEORV32_UART0, "M3 Coords: %d\n", M3_Coords);
                        neorv32_uart_printf(NEORV32_UART0, "M3 Error: %d\n", M3_Error);
                        neorv32_uart_printf(NEORV32_UART0, "M3 Signal: %d\n",
                        (int)M3_Speed);

                        neorv32_uart_printf(NEORV32_UART0, "Encoder Counter M4: %d\n",
                        NEORV32_CFS->REG[3]);
                        neorv32_uart_printf(NEORV32_UART0, "M4 Coords: %d\n", M4_Coords);
                        neorv32_uart_printf(NEORV32_UART0, "M4 Error: %d\n", M4_Error);
                        neorv32_uart_printf(NEORV32_UART0, "M4 Signal: %d\n\n",
                        (int)M4_Speed);
                    }
                }
            }
}

double PID_M1(int error_M1)
{
    derivative = (error_M1 - prevError_M1) / dt;
    errorIntegral = errorIntegral + error_M1 * dt;
    controlSignal = (Kp * error_M1) + (Ki * errorIntegral) + (Kd * derivative);

    controlSignal = round(controlSignal);
    controlSignal = limitValue(controlSignal);

    prevError_M1 = error_M1;

    return controlSignal;
}
```

```
double PID_M2(int error_M2)
{
    derivative = (error_M2 - prevError_M2) / dt;
    errorIntegral = errorIntegral + error_M2 * dt;
    controlSignal = (Kp * error_M2) + (Ki * errorIntegral) + (Kd * derivative);

    controlSignal = round(controlSignal);
    controlSignal = limitValue(controlSignal);

    prevError_M2 = error_M2;

    return controlSignal;
}

double PID_M3(int error_M3)
{
    derivative = (error_M3 - prevError_M3) / dt;
    errorIntegral = errorIntegral + error_M3 * dt;
    controlSignal = (Kp * error_M3) + (Ki * errorIntegral) + (Kd * derivative);

    controlSignal = round(controlSignal);
    controlSignal = limitValue(controlSignal);

    prevError_M3 = error_M3;

    return controlSignal;
}

double PID_M4(int error_M4)
{
    derivative = (error_M4 - prevError_M4) / dt;
    errorIntegral = errorIntegral + error_M4 * dt;
    controlSignal = (Kp * error_M4) + (Ki * errorIntegral) + (Kd * derivative);

    controlSignal = round(controlSignal);
    controlSignal = limitValue(controlSignal);

    prevError_M4 = error_M4;

    return controlSignal;
}

int map(int x, int in_min, int in_max, int out_min, int out_max)
{
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

```
double limitValue(double value)
{
   if(value >= MAX_SPEED) return value = MAX_SPEED;
   else if(value <= -MAX_SPEED) return value = -MAX_SPEED;
   else return value;
}

void debug_gamepad(void)
{
    if (gamepad.state.joyStatus == 1)
    {
      neorv32_uart_printf(NEORV32_UART0, "LX=%u\n", gamepad.state.joyLeftX);
      neorv32_uart_printf(NEORV32_UART0, "LY=%u\n", gamepad.state.joyLeftY);
      neorv32_uart_printf(NEORV32_UART0, "RX=%u\n", gamepad.state.joyRightX);
      neorv32_uart_printf(NEORV32_UART0, "RY=%u\n", gamepad.state.joyRightY);
      neorv32_uart_printf(NEORV32_UART0, "dpad=%u\n", gamepad.state.btn.dpad);

      if(gamepad.state.btn.btnSquare)neorv32_uart_puts(NEORV32_UART0, "Square\n");

  if(gamepad.state.btn.btnTriangle)neorv32_uart_puts(NEORV32_UART0, "Triangle\n");

      if (gamepad.state.btn.btnCross) neorv32_uart_puts(NEORV32_UART0, "Cross\n");

      if(gamepad.state.btn.btnCircle)neorv32_uart_puts(NEORV32_UART0, "Circle\n");

      if(gamepad.state.btn.btnL1) neorv32_uart_puts(NEORV32_UART0, "L1\n");

      if (gamepad.state.btn.btnR1)    neorv32_uart_puts(NEORV32_UART0, "R1\n");

      if (gamepad.state.btn.btnL2)    neorv32_uart_puts(NEORV32_UART0, "L2\n");

      if (gamepad.state.btn.btnR2)    neorv32_uart_puts(NEORV32_UART0, "R2\n");

      if (gamepad.state.btn.btnShare) neorv32_uart_puts(NEORV32_UART0, "Share\n");

  if (gamepad.state.btn.btnOptions)neorv32_uart_puts(NEORV32_UART0, "Options\n");

      if (gamepad.state.btn.btnL3)    neorv32_uart_puts(NEORV32_UART0, "L3\n");

      if (gamepad.state.btn.btnR3)    neorv32_uart_puts(NEORV32_UART0, "R3\n");

      if (gamepad.state.btn.btnPs)neorv32_uart_puts(NEORV32_UART0, "PS button\n");

  if(gamepad.state.btn.btnTouchPad)neorv32_uart_puts(NEORV32_UART0, "TouchPad\n");
    }
}
```

Controller Transmission Code

```
#include <ps5Controller.h>

unsigned char Lx_abs = 128;
unsigned char Ly_abs = 127;
unsigned char Rx_abs = 128;
unsigned char Ry_abs = 127;
unsigned char key_B5 = 0;
unsigned char key_B6 = 0;
unsigned char key_B7 = 0;
unsigned char key_DPAD = 0;
byte gamepad_data[8];

void setup()
{
  Serial.begin(115200);
  // 24:A6:FA:50:5D:D4 (1) OR 24:A6:FA:3B:93:0A (2) OR 24:A6:FA:95:1E:9E (3)
  ps5.begin("24:A6:FA:95:1E:9E"); //replace with controller MAC address
}

void loop()
{
  while (ps5.isConnected() == true)
  {

    if (ps5.LStickX()) Lx_abs = ps5.LStickX() + 128;
    if (ps5.LStickY()) Ly_abs = -ps5.LStickY() + 127;
    if (ps5.RStickX()) Rx_abs = ps5.RStickX() + 128;
    if (ps5.RStickY()) Ry_abs = -ps5.RStickY() + 127;

    if (ps5.Up()) key_DPAD = 0;
    else if (ps5.Right()) key_DPAD = 2;
    else if (ps5.Down()) key_DPAD = 4;
    else if (ps5.Left()) key_DPAD = 6;
    else  key_DPAD = 8;

    if (ps5.Square()) key_B5 = key_B5 | 0x10;
    else  key_B5 = key_B5 & 0xEF;

    if (ps5.Cross()) key_B5 = key_B5 | 0x20;
    else  key_B5 = key_B5 & 0xDF;

    if (ps5.Circle()) key_B5 = key_B5 | 0x40;
    else  key_B5 = key_B5 & 0xBF;

    if (ps5.Triangle()) key_B5 = key_B5 | 0x80;
    else  key_B5 = key_B5 & 0x7F;
    key_B5 = key_B5 & 0xF0;
    key_B5 = key_B5 | key_DPAD;

    if (ps5.L1()) key_B6 = key_B6 | 0x01;
    else key_B6 = key_B6 & 0xFE;
    if (ps5.R1())key_B6 = key_B6 | 0x02;
    else key_B6 = key_B6 & 0xFD;

    if (ps5.L2()) key_B6 = key_B6 | 0x04;
    else key_B6 = key_B6 & 0xFB;
    if (ps5.R2()) key_B6 = key_B6 | 0x08;
    else key_B6 = key_B6 & 0xF7;

    if (ps5.Share())key_B6 = key_B6 | 0x10;
    else key_B6 = key_B6 & 0xEF;
    if (ps5.Options()) key_B6 = key_B6 | 0x20;
    else key_B6 = key_B6 & 0xDF;

    if (ps5.L3()) key_B6 = key_B6 | 0x40;
    else key_B6 = key_B6 & 0xBF;
    if (ps5.R3()) key_B6 = key_B6 | 0x80;
    else key_B6 = key_B6 & 0x7F;

    key_B7 = key_B7 | 0x4;

    if (ps5.Touchpad()) key_B7 = key_B7 | 0x10;
    else key_B7 = key_B7 & 0xEF;
    if (ps5.PSButton()) key_B7 = key_B7 | 0x40;
    else key_B7 = key_B7 & 0xBF;

    gamepad_data[0] = 0x01;   //unchanged
    gamepad_data[1] = Lx_abs; //byte1 -> LX
    gamepad_data[2] = Ly_abs; //byte2 -> LY
    gamepad_data[3] = Rx_abs; //byte3 -> Rx
    gamepad_data[4] = Ry_abs; //byte4 -> Ry
    gamepad_data[5] = key_B5;
    gamepad_data[6] = key_B6;
    gamepad_data[7] = key_B7; //checksum + touchpad + L3, R3

    Serial.write(gamepad_data, sizeof(gamepad_data));

    delay(10);
  }
}
```

Quadrature Reading Implementation using VHDL

```
  -- cfs_in_i(0) takes in output A from encoder Motor 1
  -- cfs_in_i(1) takes in output B from encoder Motor 1
  -- cfs_in_i(2) takes in output A from encoder Motor 2
  -- cfs_in_i(3) takes in output B from encoder Motor 2
  -- cfs_in_i(4) takes in output A from encoder Motor 3
  -- cfs_in_i(5) takes in output B from encoder Motor 3
  -- cfs_in_i(6) takes in output A from encoder Motor 4
  -- cfs_in_i(7) takes in output B from encoder Motor 4
Encoder: process(rstn_i)
begin
    if rising_edge(cfs_in_i(0)) then
        if (cfs_in_i(1) = '1') then
           CounterM1 <= CounterM1 + 1;
        else
            CounterM1 <= CounterM1 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(2)) then
        if (cfs_in_i(3) = '1') then
           CounterM2 <= CounterM2 + 1;
        else
            CounterM2 <= CounterM2 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(4)) then
        if (cfs_in_i(5) = '1') then
           CounterM3 <= CounterM3 + 1;
        else
            CounterM3 <= CounterM3 - 1;
        end if;
    end if;

    if rising_edge(cfs_in_i(6)) then
        if (cfs_in_i(7) = '1') then
           CounterM4 <= CounterM4 + 1;
        else
            CounterM4 <= CounterM4 - 1;
        end if;
    end if;

end process Encoder;

  cfs_reg_rd(0) <= std_ulogic_vector(to_unsigned(CounterM1,
cfs_reg_rd(0)'length));
  cfs_reg_rd(1) <= std_ulogic_vector(to_unsigned(CounterM2,
cfs_reg_rd(1)'length));
  cfs_reg_rd(2) <= std_ulogic_vector(to_unsigned(CounterM3,
cfs_reg_rd(2)'length));
  cfs_reg_rd(3) <= std_ulogic_vector(to_unsigned(CounterM4,
cfs_reg_rd(3)'length));
```