Exploring RISC-V CPU for Aerospace Applications

1st Arthur Martins de Souza Barreto
Department of Computer Engineering
Insper
Sao Paulo, Brazil
arthurmsb@al.insper.edu.br

2nd Eduardo Schneider Monteiro de Barros
Department of Computer Science
Insper
Sao Paulo, Brazil
eduardosmb@al.insper.edu.br

3rd Rodrigo Anciães Patelli
Department of Computer Engineering
Insper
Sao Paulo, Brazil
rodrigoap8@al.insper.edu.br

4th Victor Luis Gama de Assis
Department of Computer Engineering
Insper
Sao Paulo, Brazil
victorlga@al.insper.edu.br

5th Rafael Corsi Ferrão
Department of Computer Engineering
Insper
Sao Paulo, Brazil
rafael.corsi@insper.edu.br

***Abstract*—This paper explores the potential of RISC-V CPUs for aerospace applications, focusing on the specific use case of Brushless Direct Current Motor (BLDC) control. The project aims to contribute to Brazil's technological sovereignty by reducing reliance on foreign semiconductor technology. An initial implementation of a basic Six-Step algorithm demonstrated the feasibility of RISC-V for motor control. However, the limitations of this approach led to the investigation of more advanced Field-Oriented Control (FOC). While FOC offers superior performance, its implementation presented challenges related to timing constraints, current measurement accuracy, and rotor position feedback. The project successfully identified hardware requirements and constraints associated with BLDC motor control.**

***Index Terms*—RISC-V, BLDC motor control, aerospace applications, technological sovereignty, FPGA**

I. Introduction

Brazil currently depends on integrated circuits (IC) developed and manufactured by other nations. This dependency exposes the country to the political landscape of the international scenario, reduces the level of computer personalization for national use case scenarios, and exposes what technology projects the Brazilian government is working on to foreign institutions, which may represent a vulnerability to information security. The national industry is also affected by forcing companies to rely on international technological commercial agreements, typically slow-paced, instead of focusing on building their products and solutions.

To address the need for domestic semiconductor technology, the Centro de Tecnologia da Informação Renato Archer (CTI Renato Archer), a research institution under the Ministério de Ciência, Tecnologia e Inovação (MCTI), has partnered with Insper Instituto de Ensino e Pesquisa (Insper) to develop a proprietary computer processor. Established in 1982, CTI Renato Archer is dedicated to promoting technological innovation and strengthening the Brazilian industry through collaborations between government, industry, and academia. Based in Campinas, São Paulo, the center conducts research in various areas, including micro/nanofabrication, electronics, photovoltaic energy, and software development.

The development of a custom intellectual property (IP) for computer processors is a multi-year resource-demanding initiative. Careful planning and prototyping is needed to properly meet requirements. To investigate hardware requirements for an IC useful in critical industries such as aerospace we decided to build a common application for satellites utilizing an open-source implementation of the RISC-V, NEORV32[1]. The selected application was the control algorithm of a Brushless Direct Current Motor (BLDC) motor. This motor is a critical component to several solutions, specially those with low-maintenance constraints, such as drone's helices and reaction wheels and gyroscopes to control the satellite's orientation. Specifically, in the aerospace scenario, BLDC motors are important in reason of the difficulty of accessing motors deployed in space. This makes low to no maintainability a critical requirement.

This paper presents the results of our investigation, what was and wasn't possible to prototype. The open-source NEORV32 implementation and a common application like BLDC motor control allow us to gain valuable insights into the upcoming iterations of the custom RISC-V core, empowering the nation's industry and reducing dependence on foreign nations.

II. Brushless DC Motor

The BLDC motor is an electric motor without brushes, making it more efficient and durable than traditional direct current (DC) motors. It consists of a rotor with permanent magnets and a stator with coils as show on Figure 1. A magnetic field is generated when current flows through the coils, causing the rotor to rotate as it aligns with this field.

The operation of BLDC motors necessitates the utilization of an electronic controller to regulate the current through the coils, thereby guaranteeing uninterrupted rotor movement. This controller is paramount for accurately selecting coils and precisely timing the current to enhance performance. BLDC motors are used in aerospace and other critical applications due to their reliability and low maintenance, and fault-tolerant

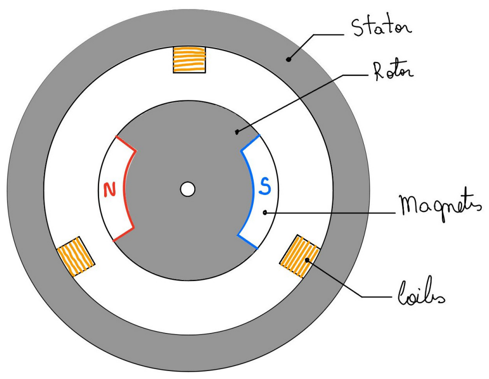


Fig. 1: Basic Structure of a BLDC

control systems are often employed to ensure continued operation.

To maintain rotation, the current direction in the coils must periodically reverse, creating a dynamic magnetic field that keeps the rotor in motion. This requires specialized algorithms for both open and closed-loop control.

A. Operating Principle

In a DC electric motor, five fundamental physical phenomena occur simultaneously, enabling motor operation, as illustrated in Figure 2. The process begins with the armature voltage e_a, which, according to Ohm's law, generates an armature current i_a. This current, in turn, produces a motor torque τ_m through the Lorentz force. The torque causes the motor to rotate with an angular velocity ω_m, which then induces a back EMF (electromotive force) e_b in the armature due to Faraday's law of induction. This induced voltage opposes the armature voltage e_a, creating a feedback loop that regulates motor speed and stabilizes operation. This chain of interactions follows physical laws essential to motor functionality.

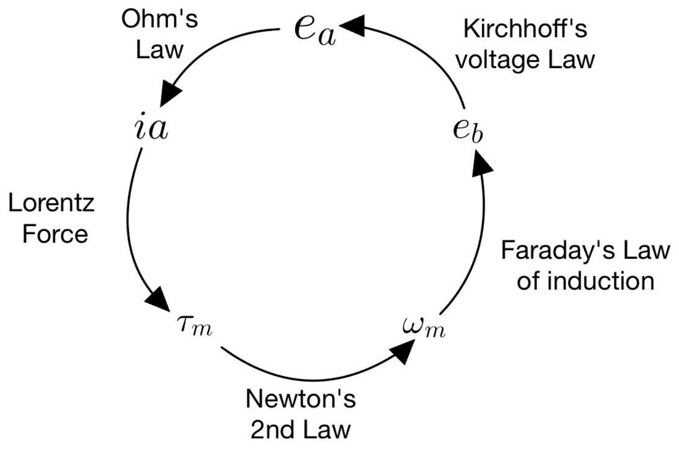


Fig. 2: Electromagnetism Laws

The same fundamental laws govern the operation of a BLDC motor as a conventional DC motor, but initiating movement in a BLDC motor is more complex. As shown in Figure 1, the BLDC motor contains multiple coils, and to generate rotation, these coils must be energized in a specific sequence. This precise control of coil activation enables the BLDC motor to rotate efficiently. There are various methods for driving a BLDC motor, with the most well-known being the **Six-step algorithm** and the **Field Oriented Control (FOC) algorithm**. Each approach offers unique benefits in terms of torque smoothness, precision, and efficiency.

B. Key Characteristics

Accurate motor characterization is essential for designing an effective controller, as it allows precise modeling of the motor's behavior. Understanding the motor's electrical and mechanical parameters provides insights into how it will respond under different operating conditions, which is particularly important for implementing advanced control techniques, such as FOC.

In this project, rather than performing in-depth parameter identification ourselves, we utilized tools from STMicroelectronics [2], that provide precise measurements for all relevant electrical and mechanical variables, as shown in Table I. These parameters are essential for developing an accurate control model and ensuring the stability and efficiency of the motor under FOC, being used to make the control algorithm, which will be explored more in section V-B.

TABLE I: Electrical and Mechanical Specifications

Electrical Model	
Resistance (R_s)	0.1 Ω
Inductance (L_s)	0.05 mH
Electrical Constant (k_e)	0.36 Vrms/kRPM
VBus	11.4 V
Max Current ($I_{\max}$)	10 Apk
Mechanical Model	
Friction	5.22 μN·m·s
Inertia	992.61 nN·m·s^2
Max Speed (mechanical limit)	20,080 RPM

These electrical and mechanical parameters directly influence the motor's performance and control precision. For example, the resistance (R_s) and inductance (L_s) affect the motor's response time to current changes, which is crucial for FOC algorithms that rely on fast, precise current adjustments to maintain optimal torque and speed. Similarly, friction and inertia impact the motor's acceleration and deceleration characteristics, affecting its stability and efficiency under load.

Implementing FOC requires accurate knowledge of these parameters to achieve smooth and efficient control. If the parameter values are not correctly calibrated, the FOC algorithm may perform poorly or even fail, leading to instability or inefficient operation. However, if a simpler control method, such as six-step control, is used, detailed parameter data may not be necessary, as six-step control does not rely as heavily on precise motor characterization.

III. Drive Methods

When controlling a BLDC motor, several algorithms can be applied, ranging from simple to more complex methods.

Unlike a conventional DC motor, even the simplest approach for a BLDC motor requires activating specific phases in a precise order and position; failing to do so prevents the motor from moving.

A. Open-Loop Control

Open-loop control for a BLDC motor lacks feedback, meaning there is no mechanism to optimize efficiency, regulate current, or maintain a constant speed. In this configuration, the motor simply receives an input signal to move, but the resulting output behavior, including speed and position, remains uncontrolled. As shown in Figure 3, the open-loop approach merely initiates motion without maximizing performance or stability.

In open-loop control, an input signal causes an action without monitoring or adjusting the output. This limitation is why closed-loop control, which includes feedback mechanisms, is often preferred in applications requiring consistent motor performance.

B. Closed-Loop Control

As discussed in Chapter III-A, open-loop control lacks the performance and efficiency required for many applications. For this reason, closed-loop control is preferred, despite being more complex to implement. Closed-loop control requires continuous feedback from the motor to adjust its performance in real-time. Figure 4 illustrates an example of a FOC system, a reliable, and efficient closed-loop algorithm commonly used for BLDC motors.

Before delving into specific control algorithms, it's essential to understand the feedback variables necessary for implementing closed-loop control. These include:

- **Position**
- **Angular Speed**
- **Current**

At least one of these parameters plays a crucial role in motor control. The system can adjust performance dynamically to meet specific requirements by accurately measuring these variables.

Position and Angular Speed are fundamental for precise control. Position feedback allows the control system to estimate angular speed, typically using timer techniques. Common methods for obtaining position feedback include Hall sensors:

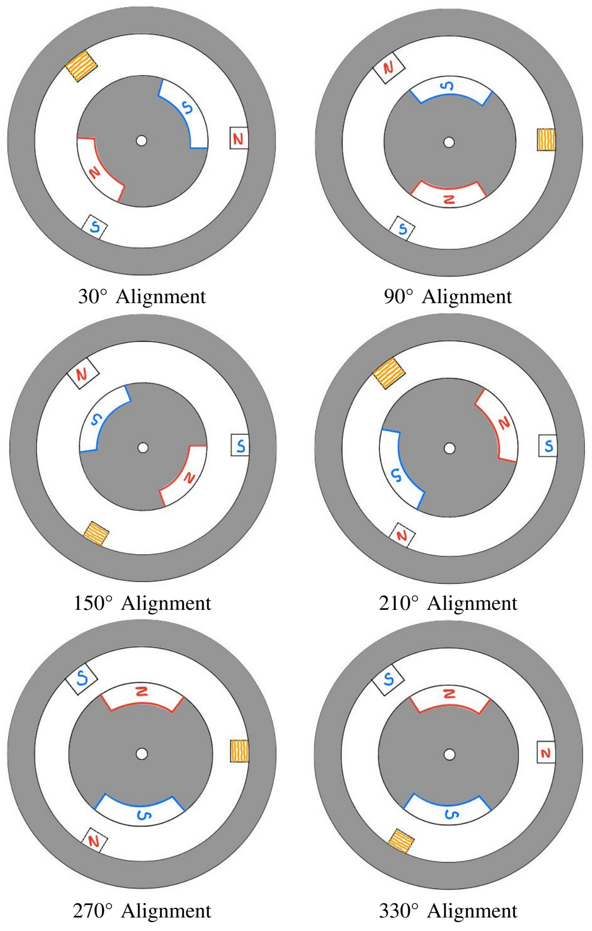
- Hall sensors provide digital signals indicating the rotor's presence within specific sectors.
- In BLDC control, three Hall sensors typically divide the rotor's movement into six distinct positions (or sectors).
- However, Hall sensors offer coarse position data, covering 60-degree sectors. For finer position control, sensorless methods measuring back EMF or using an encoder may be necessary.

Current and Torque Control are important components, depending on the specific requirements of the application. As discussed in Section II-A, torque is proportional to the current supplied to the motor. Therefore, if the application requires precise torque control, implementing a closed-loop algorithm that regulates current is essential. However, for applications where torque control is not critical, current regulation may not be necessary. The choice of control strategy depends on the problem to be solved, allowing for a more tailored and efficient use of resources.

C. Six-step

The Six-Step Algorithm is a common open-loop control method for BLDC motors, consisting of six discrete steps that sequentially activate motor coils to produce rotation. Each step advances the rotor alignment by 60°, ensuring continuous rotation as shown in Table II. Hall sensors, typically three in BLDC motors, detect rotor position by producing high signals when the rotor aligns with a phase, as depicted in Figure 5. Based on the rotor's position, specific phases are energized to generate the necessary torque.

TABLE II: Motor Alignments at Different Angles



The motor's commutation is divided into six sectors, each covering a specific rotor angle range and determining the switching sequence for the motor phases, as summarized in Table III. In this scheme, a phase can be energized as a north or south pole, or set to a high-impedance state. This switching approach provides efficient torque generation, with proper phase transitions occurring at each rotor position for smooth operation.

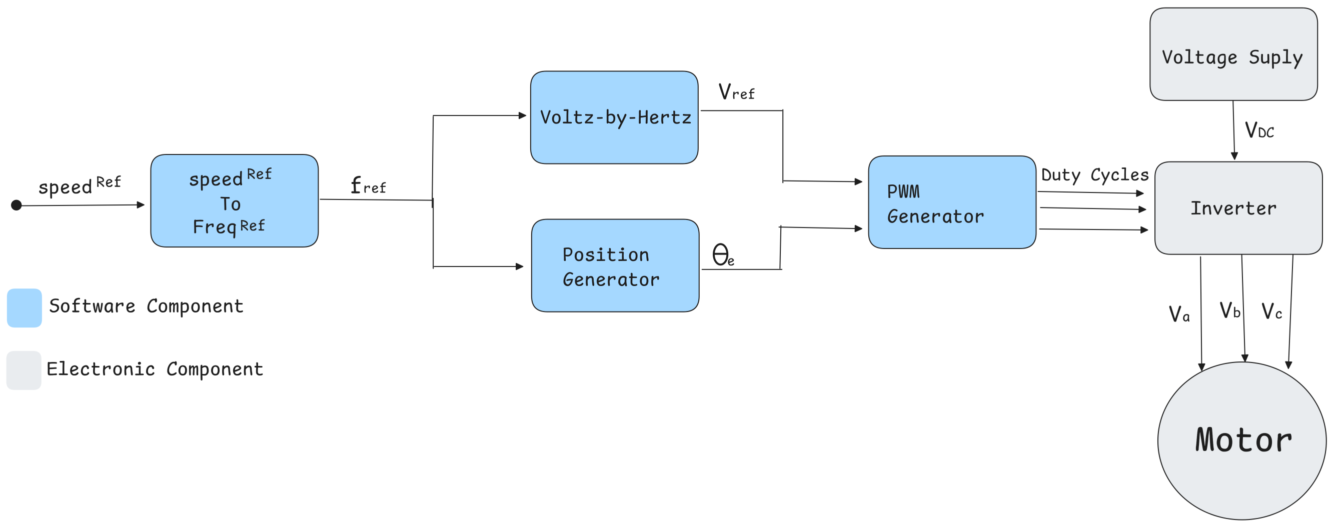


Fig. 3: Open-Loop Motor Control

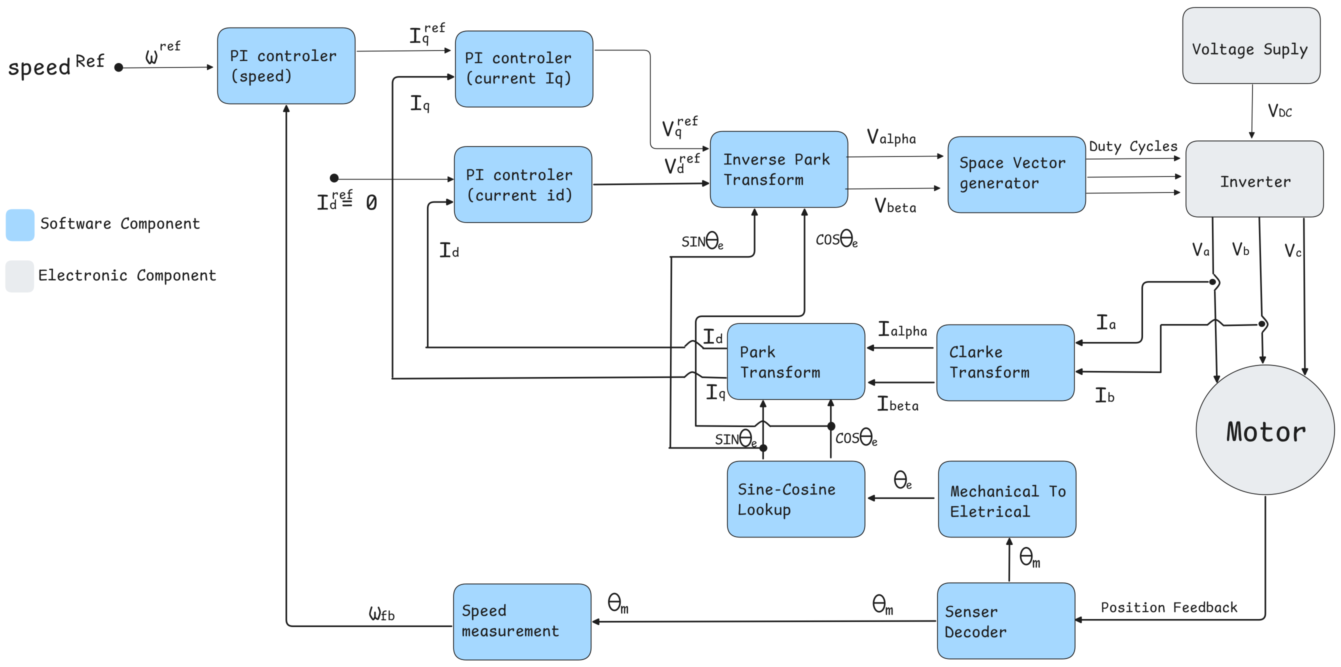


Fig. 4: Closed-Loop Motor Control

TABLE III: Commutation logic based on the Hall sequence to generate switching sequences [3]

Position (θ)	**Sector**	**Switching Sequence (AA'BB'CC')**		
		AA'	**BB'**	**CC'**
(-30°, 30°]	1	00	10	01
(30°, 90°]	2	01	10	00
(90°, 150°]	3	01	00	10
(150°, 210°]	4	00	01	10
(210°, 270°]	5	10	01	00
(270°, 330°]	6	10	00	01

For practical application, the X-NUCLEO-IHM07M1 shield was selected for its compatibility and availability of supporting instructional materials. This hardware platform facilitated testing and validating control code implementation.

Each motor phase has three possible states: high-impedance, -Vcc, and +Vcc, as illustrated in Figures 6a, 6b, and 6c. These states are controlled by two signals, EN (Enable) and IN (Input). Table IV summarizes the EN and IN values across sectors, detailing which switches are activated to drive the motor consistently.

Hall sensors enhance BLDC motor control but are not strictly required. Following the command sequence in Table IV enables counterclockwise rotation, and reversing the sequence achieves clockwise motion.

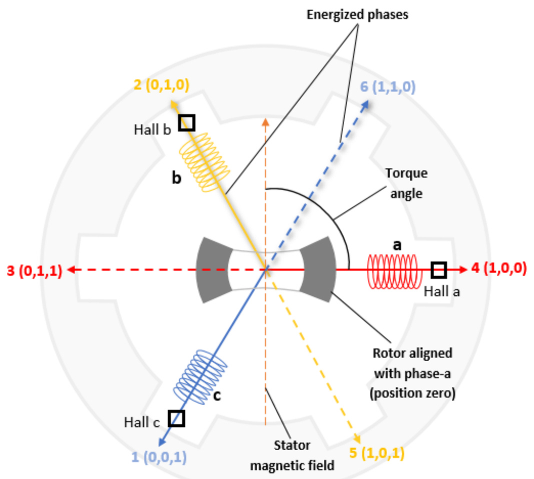
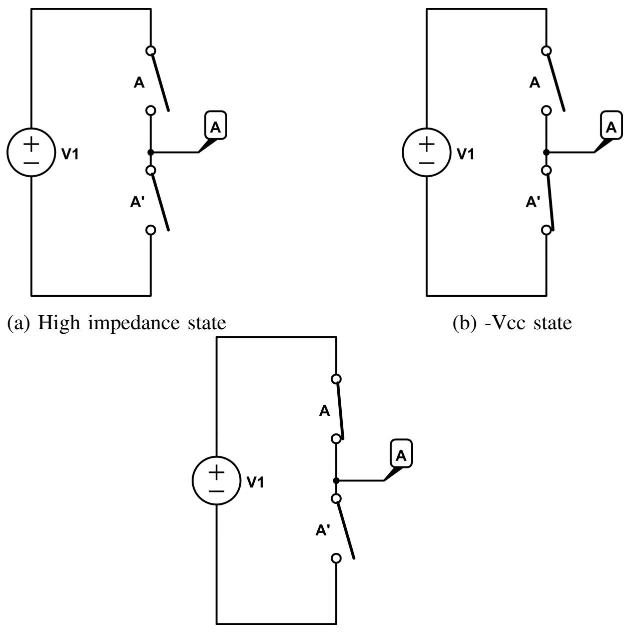


Fig. 5: Stator magnetic field phases along with their default Hall sequence Figure reproduced from MathWorks [3]



(a) High impedance state

(b) -Vcc state

(c) +Vcc state

for

Fig. 6: Illustration of different phase states for motor control

D. FOC

The FOC algorithm is a closed-loop approach for controlling BLDC motors, addressing torque optimization issues present in open-loop methods. In the Six-Step method, torque fluctuates due to varying angles between the rotor's magnetic field and the induced magnetic field, as shown in Figure 7a. This fluctuation results in inefficient torque generation, where maximum torque is only achieved at a 90° angle, as seen in Figure 7b. FOC continuously adjusts the phase currents to maintain this optimal angle, ensuring maximum torque efficiency throughout the rotor's rotation.

Achieving optimal control requires continuous rotor position feedback to adjust the three-phase current and maintain the desired 90° alignment. Precise current control relies on real-time measurements from current sensors in each phase. These sensors enable feedback necessary for FOC's current regulation, ensuring efficient torque generation.

TABLE IV: Activation values for each sector

Sector	EN1	IN1	EN2	IN2	EN3	IN3
1	0	0	1	1	1	0
2	1	0	1	1	0	0
3	1	0	0	0	1	1
4	0	1	0	0	1	1
5	1	1	1	0	0	0
6	1	1	0	0	1	0

In BLDC motors, the rotor produces a constant magnetic field based on its pole pairs, while the stator induces a magnetic field from the powered phases. This interaction generates an induced current, as illustrated in Figure 7c. The current is split into two components: i_d, aligning with the rotor's magnetic field, and i_q, perpendicular to it. Only i_q contributes to torque generation, so FOC aims to minimize i_d to optimize efficiency and torque output.

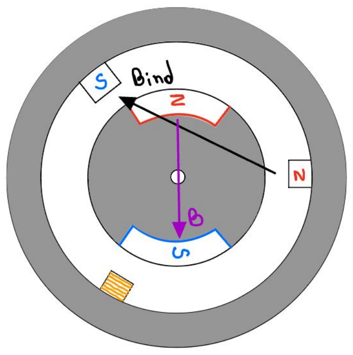
The Clark Transform [4], shown in Figure 8a, plays a crucial role by converting the three-phase current (abc) into the stationary $\alpha\beta$ frame. This transformation simplifies the analysis and control of the current, facilitating efficient motor operation.

In the $\alpha\beta$ system, the direct axis (i_d) and quadrature axis (i_q) currents can be determined. The Clark Transform's matrix form includes a constant factor, k, which ensures either current magnitude or power invariance between the abc and $\alpha\beta$ frames. For balanced systems, the sum of phase currents equals zero, ensuring stability.

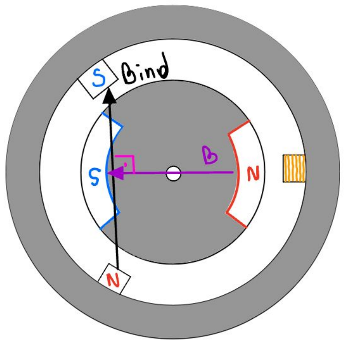
After the Clark Transform, the Park Transform [5] further converts $\alpha\beta$ currents to the rotating dq frame, aligning them with the rotor's magnetic field (Figures 8b and 8c). The Park equations express i_d and i_q as functions of the rotor angle θ, separating the torque-producing i_q component from the non-torque-producing i_d component. This separation enables precise torque control by modulating i_q while minimizing i_d, enhancing motor efficiency.

After the transformations, the FOC algorithm computes the required phase current values. However, since BLDC motors operate with voltage adjustments rather than direct current inputs, these computed current values must be converted into corresponding voltage signals. To achieve this, Pulse Width Modulation (PWM) is used to approximate sinusoidal voltages by rapidly switching the DC supply. Specifically, Space Vector Modulation (SVM) is employed, as illustrated in Figure 9. SVM calculates the precise duty cycle for each phase based on the reference signal from the control block, enabling PWM to adjust the effective voltage. This process creates a smooth, sinusoidal-like current waveform in each phase, ensuring efficient and continuous motor operation.

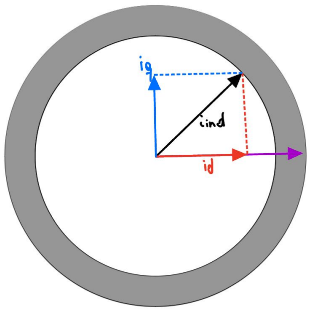
Implementing the Park Transform presents a challenge due to the requirement for continuous rotor angle tracking. In an



(a) Magnetic Fields at the BLDC

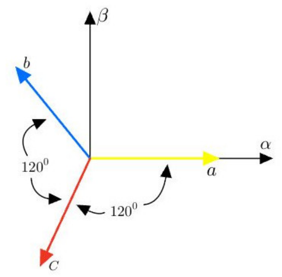


(b) Maximum torque situation

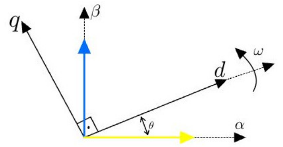


(c) Induced magnetic field related to rotor position

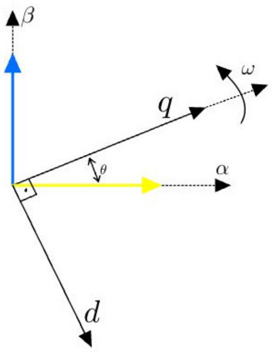
Fig. 7: Various aspects of BLDC motor performance



(a) Direction of magnetic axes in abc reference frame



(b) The d-axis aligns with the α-axis



(c) The q-axis aligns with the α-axis

Fig. 8: Illustration of magnetic axes and reference frame alignment

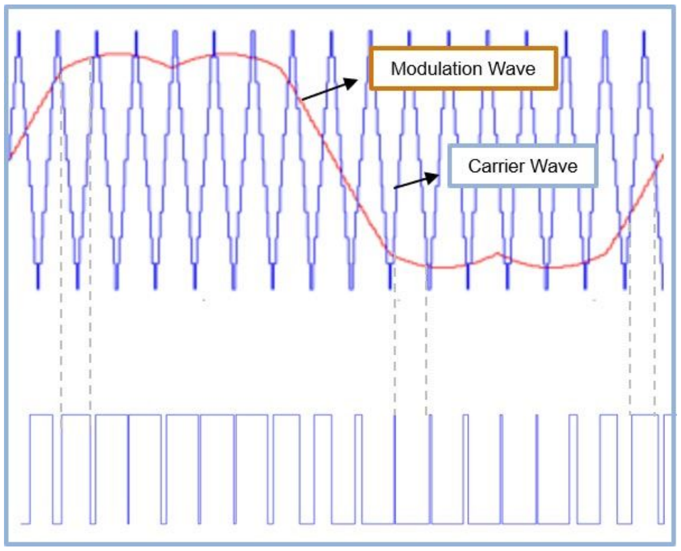


Fig. 9: Gate pulse generation as a result of comparing the modulation wave and the carrier wave
Figure reproduced from MathWorks [6]

earlier stage of this study, a mechanical encoder was used; however, due to noise issues affecting the accuracy of its readings, the focus shifted to Hall sensors. Various methods for angle estimation exist, but Hall sensors were chosen to minimize potential failure points, a critical consideration in aerospace applications.

The *FOC* control system (Figure 4) integrates the Clark and Park transforms, along with a Proportional-Integral (PI) controller for precise current and speed regulation.

The detailed description of how both *Six-Step* and *FOC* control algorithms were implemented in this project can be found in V-B.

IV. Running the closed loop six-step control software in RISC-V core

During the project, some key decisions were made regarding the control algorithm. As detailed in Section V-B, the implementation of FOC faced significant challenges, demanding a shift to the Six-Step algorithm with closed-loop speed control.

This change was driven by the need for a simpler, yet effective, solution to validate the project's goals.

This project aims to validate the hardware requirements necessary to guide the development of a custom RISC-V core. To achieve this, we leveraged the NEORV32, an open-source and highly customizable RISC-V implementation in VHDL, to simulate the core's performance and compatibility with essential control algorithms. The NEORV32 was implemented on a DE0-CV development board equipped with a Cyclone V Altera FPGA [7], chosen for its compatibility with the project memory requirements and availability at Insper. By integrating specific hardware extensions and developing a custom peripheral, *HallSector*, for hall sensor signal processing, we created a setup that approximates the environment of the envisioned custom core, shown in figure 10. The following subsections detail the hardware configurations and key insights obtained from this implementation, providing a foundation for the proprietary RISC-V core's future design.

A. FPGA

An FPGA is a device that can be programmed to emulate any custom digital circuit that fits its hardware limitations. It can be used for prototyping and developing application-specific integrated circuits (ASICs), deploying hyperspecialized hardware-software systems for high-performance computing, and creating hardware that does not require mass production, as in industries like financial market trading.

In this project, we used the DE0-CV board from Terasic Technologies [7], which features a Cyclone V FPGA chip from Altera. This development kit was selected due to its high availability at Insper, low cost, and sufficient capacity to accommodate 64 KB of instruction memory (IMEM) directly on-chip. Furthermore, a GitHub repository [8] was identified, which contained a wrapped version of NEORV32 with a top-level module compatible with the DE0-CV board, significantly accelerating the integration process.

After synthesis and placement, resource utilization on the Cyclone V FPGA was analyzed, revealing the following key metrics:

- **Logic Utilization:** 2,287 Adaptive Logic Modules (ALMs) out of 18,480 available (12%). A dense packing estimation suggests that 217 ALMs (1%) could be recoverable with further optimization.
- **Registers Used:** 2,919 dedicated logic registers, representing 7% of the device's 36,960 registers.
- **Memory Utilization:** The design utilized 106 M10K memory blocks (34%) and consumed 822,272 block memory bits (26% of total).
- **I/O Pins:** 55 out of 224 available I/O pins were used (25%), including 3 clock pins.
- **DSP Blocks:** Only 1 out of 66 DSP blocks was used (2%), reflecting limited reliance on high-performance arithmetic hardware.
- **LAB Utilization:** 319 logic array blocks (LABs) were partially or completely used (17% of total LABs).

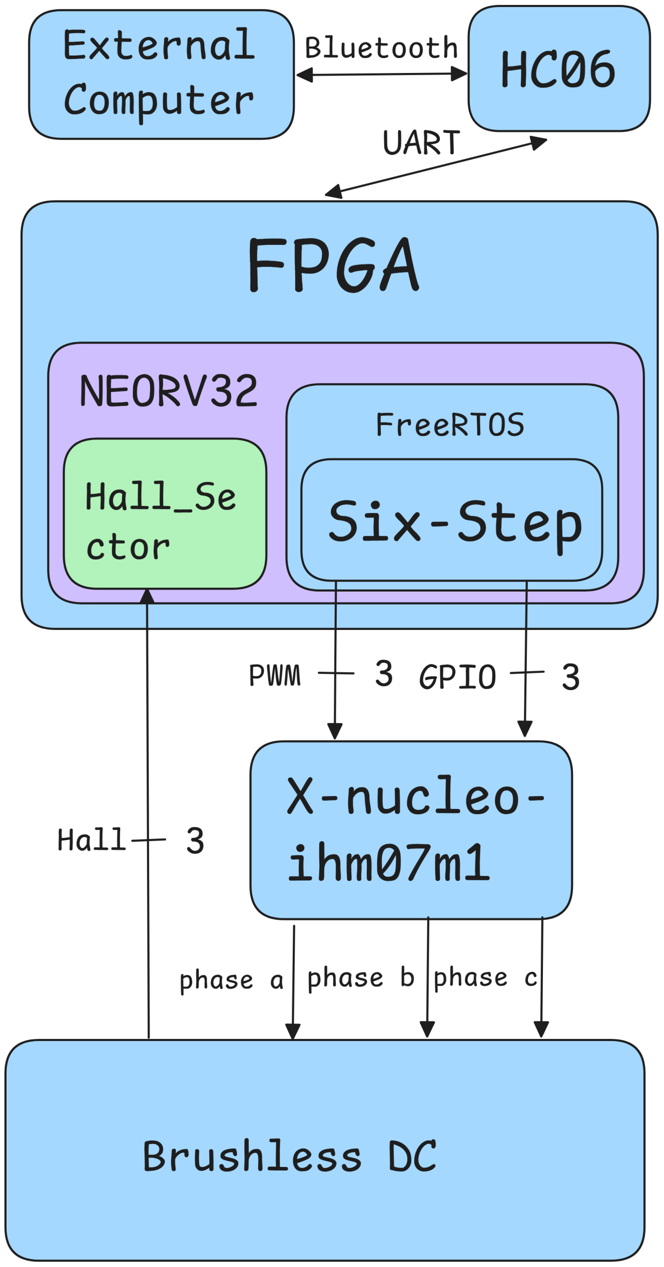


Fig. 10: Project Diagram

- **Combinational Logic:** The design required 3,651 adaptive look-up tables (ALUTs), with a breakdown ranging from 7-input to simpler $\leq$3-input functions.
- **Packing Difficulty:** Reported as low by the Fitter.
- **Interconnect Usage:** Average interconnect usage was 6.1%, with a peak usage of 22.7%.

B. RISC-V

The decision to focus on RISC-V was mainly motivated by its being the industry's most widely used open-standard ISA. RISC-V is distinguished by its modularity and simplicity, allowing the processor to specialize in different use cases. However, the technology gained fame because of the open and free license that allows any engineer or designer to implement the ISA freely. It can be used for any purpose, and the implementations may or may not be open at the developer's discretion.

Initially, the RV32I instruction set was selected as the base. This is a 32-bit instruction set that supports basic operations such as integer addition and subtraction, which serve as the foundation for compilation and assembly by RISC-V tool-chains. Regarding extensions, the M, F and Zicntr or similar extensions were mapped as essential. These extensions respectively implement multiplication and division, floating-point operations, and basic system counters. These are necessary for executing the control algorithm, as they enable the hardware to perform the required mathematical operations with decimal precision and at the required frequency and timing requirements, which is critical for the proper functioning of the system.

While not essential for the project's core functionality, the inclusion of the C extension is desirable. This extension enables compressed instructions, allowing the compiler to optimize code by compressing frequently used 32-bit instructions into 16-bit instructions, thus reducing overall memory usage. This optimization is particularly valuable given the limited memory available on FPGAs, which are not only constrained in storage but also share programmable logic resources with memory. These resources could otherwise be allocated for additional functions, and any increase in memory capacity or functionality would add to fabrication costs. By reducing memory demands, the C extension helps avoid reliance on external memory, which would require further configuration and a communication bus to interface with the processor core, increasing both the project's complexity and its dependency on external peripherals.

C. NEORV32

The NEORV32 [1] processor is a flexible, microcontroller-like SoC built around the RISC-V CPU, designed entirely in VHDL. It is highly customizable, enabling users to activate specific features with minimal code change, mapping and activating the desired features by using its port and flags in the top-level entity of the processor.

It is licensed under the BSD 3-Clause License, which allows redistribution and use in both source and binary forms, with or without modification, provided that the same license is retained for redistribution and that the authors' names are not used for promotional purposes without prior written permission [9].

In this project, the NEORV32 processor was customized with specific peripherals to fulfill both general operational requirements and the unique demands of our application. The final setup includes general purpose input output pins (GPIO), pulse width modulation pins (PWM), universal asynchronous receiver/transmitter pins (UART), and communication bus peripherals, as detailed in subsubsection IV-C1 and shown in Figure 11. Moreover, a custom peripheral called *HallSector* was implemented to address the specific challenge of accurately reading hall signals. *HallSector* will be discussed in detail in subsection IV-E.

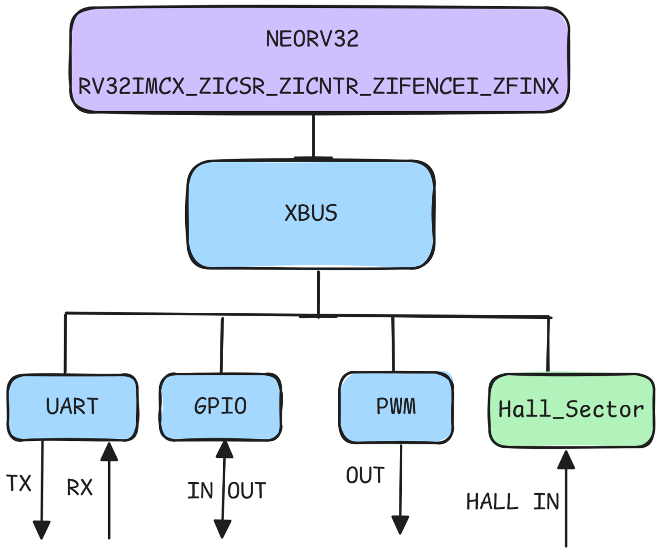


Fig. 11: Peripherals Diagram

The configuration also incorporates additional extensions compared to the original plan described in subsection IV-B. The changes were needed since NEORV32 requires the use of some extensions by default that cannot be deactivated, resulting in a core with RV32IMCX_Zicsr_Zicntr_Zifencei_Zfinx base architecture and extensions, which are described in Subsection IV-C2.

1) Utilized Peripherals: To develop the project and execute its functionalities, several peripherals were utilized, which can be categorized as follows:

- Communication
- Motor control
- Internal processor needs

To establish communication between a host machine, we used the UART communication protocol, which we configured within the NEORV32 structure using its pre-implemented Bluetooth module. This setup allowed for seamless data exchange, enabling remote control and monitoring capabilities through telecommands. The UART-USB adapter facilitated initial boot code loading, while the additional UART port provided dedicated support for Bluetooth-based communication, ensuring reliable and efficient wireless connectivity essential for system flexibility and ease of use in various operational scenarios.

For motor control, two main peripherals were necessary: the GPIO and PWM modules. The GPIO module provided a straightforward way to interface with the motor's sensors, allowing for real-time monitoring of its velocity. Meanwhile, the PWM module enabled fine control over the motor's speed by adjusting the duty cycle of the signal, effectively managing power output to the motor.

To enable the processor to manage communication between the core, peripherals, and extensions, a bus controller was added. Specifically, XBUS was used, which is the NEORV32 implementation of the Wishbone bus [10] and is responsible for handling all internal processor communication.

2) ISA Extensions: The RISC-V extensions are additional units that add new features to the base core of RISC-V. These units often require the implementations of another extensions as dependency and also can be modified or be entirely new to the ISA, as long as they met the RISC-V standards.

For this project, we are using the base 32 bits RISC-V implementation RV32I along with the extensions *M*, *C*, *Zicsr*, *Zicntr*, *Zifencei* and *Zfinx*. All these extensions are implemented as described in the RISC-V ISA specifications [11] as well as the X extension, original to the NEORV32, defining an architecture of a RV32IMCX_Zicsr_Zicntr_Zifencei_Zfinx core, which provides the mathematical operations and interrupt capabilities required for the control code. However, each extension within the NEORV32 implementation presents certain specifics that created challenges and required special handling during development.

The *Zfinx* extension is responsible for floating-point operations, but its current implementation is incomplete and contains errors in certain corner cases. For example, floating-point division is not supported, requiring emulation of this operation within the code. Additionally, the conversion between floating-point and signed integers is not correctly implemented [12], limiting the flexibility of the code. The *Zicsr* is mandatory to the NEORV32 and implements instructions to access the control and status registers (CSR), described in the RISC-V ISA [11], and is a requirement for the Zicntr extension. The *Zicntr* adds the basic cycle counter, machine cycle counter, instructions-retired counter and machine instructions-retired counter CSRs and is mandatory by the RISC-V spec. The *Zifencei* extension is mandatory and allows manual synchronization of the instruction stream. This instruction is not actively utilized, and explaining it in depth in not in the project's scope. The *X* extension [13] is always active and represents all the NEORV32 specific ISA extensions. These include 16 fast interrupts and custom trap codes for the machine trap cause, that maps the exact cause of a trap serving as a debug tool.

D. Customizing the hardware

To configure the NEORV32 implementation on an FPGA, it is essential to create a custom top-level entity. This entity invokes the NEORV32 top-level component, passing in the desired configuration settings for NEORV32 extensions, peripherals, and hardware-specific requirements, including clock settings and resource availability.

Integrating NEORV32 with the FPGA clock involves synchronizing the FPGA clock of 50 MHz with the NEORV32 system clock of 100 MHz using a phase-locked loop (PLL). This ensures that the clock frequency aligns with the system's requirements.

E. Implementing HallSector IP for reliable HALL sensor signal processing

An important challenge in the implementation of external interrupts on the NEORV32 processor is that it uses a custom external interrupt controller (XIRQ), which is incompatible with the standard RISC-V ISA. Additionally, FreeRTOS 10.4.1 only supports RISC-V timer interrupts and external interrupts, meaning that using NEORV32-specific interrupts as such the external interrupts for the XIRQ module would cause the FreeRTOS kernel to stall, requiring modifications on the FreeRTOS kernel that are out of the project scope [14].

External interrupts were utilized via software to evaluate changes in the hall sensor input. Our team developed a custom hardware component, written entirely in VHDL, to solve this issue, *HallSector* (hall value to motor sector processing). *HallSector* was designed to read the inputs from the hall sensor and evaluate in what sector the motor is currently, also avoiding readings in the transition step, using a debounce component. This approach avoids the need for external interrupts, which would conflict with the FreeRTOS kernel, and provides a simple, efficient way to track the motor's current sector.

The *HallSector* module works by reading the value of the three hall sensor wires and applying debouncing techniques to ensure that only valid states are counted. The count is then stored in the base registers of NEORV32. Software can access these counts through bus reads in the hardware abstraction layer, making retrieving the data easily without complex interrupt handling. This process is straightforward: the software simply reads the relevant register from the base register address space, which is managed by the bus system.

The VHDL code for the *Hall_Sector* module, which implements the debouncing logic and the hall sensor value processing, is shown in **Listing 1**. This code highlights the logic behind it and how it is integrated into the system to ensure proper signal handling without relying on external interrupts.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hall_sector is
Port (
clk : in std_logic;
rst : in std_logic;
signal_in : in std_logic_vector(2 downto 0);
sector : out std_logic_vector(2 downto 0)
);
end hall_sector;

architecture Behavioral of hall_sector is

signal prev_signal : std_logic_vector(2 downto
    0) := "111";
signal stable_signal : std_logic_vector(2
    downto 0) := "111";
signal debounce_counter : integer := 0;
constant debounce_limit : integer := 1000;

begin

```

```vhdl
process(clk, rst)
begin

if rst = '1' then
debounce_counter <= 0;
stable_signal <= "111";

else

if signal_in /= prev_signal then
debounce_counter <= 0;

else

if debounce_counter < debounce_limit then
debounce_counter <= debounce_counter + 1;

else
stable_signal <= signal_in;

end if;

end if;

prev_signal <= signal_in;
end if;
end process;

with stable_signal select
sector <= "100" when "001", -- 001 to 4
010" when "010", -- 010 to 2
"011" when "011", -- 011 to 3
"000" when "100", -- 100 to 0
"101" when "101", -- 101 to 5
"001" when "110", -- 110 to 1
-- others undefined
"111" when others; -- 111 to 7

end Behavioral;
```

Listing 1: VHDL code for the HallSector module

1) Exposing Sector Value via Memory Address in NEORV32: To allow the software to access the value of the sector, we mapped the *HallSector* module to a specific memory address in the NEORV32's address space. This memory-mapped access allows software to read the sector value without the need for complex interrupt handling, thus simplifying the interface between the hardware and the FreeRTOS kernel.

In the NEORV32 top-level design, we use a simple bus access process to expose the sector value through the memory-mapped address at '0xFFFFE100U'. The code responsible for this is shown in **Listing 2**. This process listens for read access requests on the bus and responds by returning the current sector value stored in the *HallSector* module passed to NEORV32 top-level via port map. If a read request is made to the mapped address, the sector value is placed on the data bus for the software to read.

The relevant VHDL code for the bus access logic is as follows:

```vhdl
bus_hall_sector: process(rstn_sys, clk_i)
begin
  if (rstn_sys = '0') then
    iodev_rsp(IODEV_SECTOR) <=
        rsp_terminate_c;

  elsif rising_edge(clk_i) then
    -- bus handshake --
    iodev_rsp(IODEV_SECTOR).ack  <=
        iodev_req(IODEV_SECTOR).stb;
    iodev_rsp(IODEV_SECTOR).err  <= '0';
    iodev_rsp(IODEV_SECTOR).data <= (others
        => '0');

    if (iodev_req(IODEV_SECTOR).stb = '1')
        then
      if (iodev_req(IODEV_SECTOR).rw = '0')
          then -- read access
          iodev_rsp(IODEV_SECTOR).data <=
              to_stdulogicvector("
              00000000000000000000000000000"
               & sector);
      end if;
    end if;

  end if;
end process bus_hall_sector;
```

Listing 2: Bus access process to expose the sector value via memory-mapped address

In this code, the bus access process listens for requests at the memory-mapped address corresponding to the *IODEV_SECTOR* device. When the software sends a read request (*rw = '0'*), the current value of the sector is returned on the data bus. The response includes the 32-bit sector value from the *HallSector* module, which is appended to the addressable data bus width, ensuring that the full 32-bit sector value is accessible for reading.

This approach simplifies the interaction between the software and the hardware, eliminating the need for complex interrupt management and providing efficient access to the sector data. The sector value can now be easily read via standard bus operations, allowing the software to track the motor's velocity in real-time.

F. Graphical Interface

For this project, a graphical interface was developed with the aim of enabling motor control and parameter adjustment in a visual and user-friendly manner for the end user. The interface is capable of sending and receiving communications via UART, including Bluetooth communication, allowing short-distance telecommands to control the motor. Figure 12 illustrates the current application.

The interface was implemented in Python, leveraging the *PyQt5* library for creating the graphical user interface. Python was chosen for its simplicity and the team's expertise with the language. For UART communication, including data transmission and reception, the Python standard library for serial communications was employed.

Delving into the details of the constructed graphical interface, in Figure 12, region 1, it is possible to connect to different ports of the host machine for communication. In Figure 12, region 2, the desired motor speed can be defined

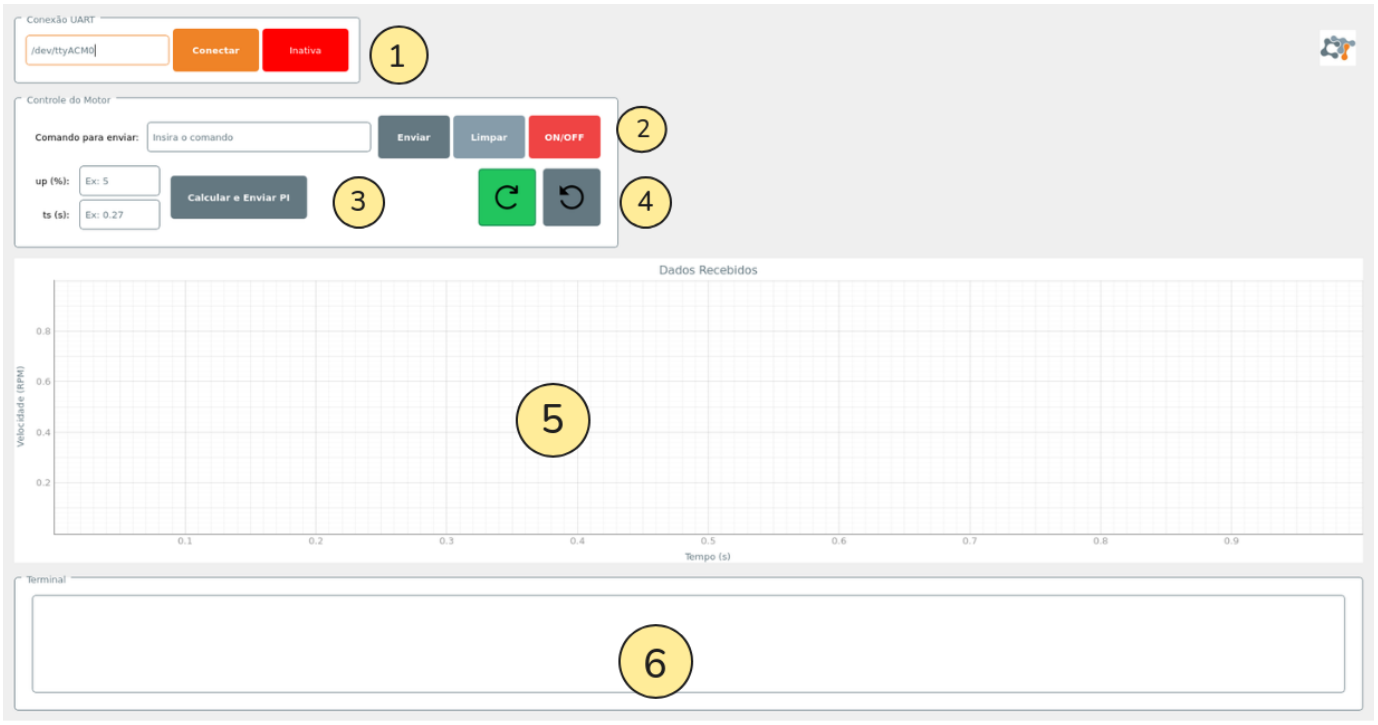


Fig. 12: Graphical interface

and sent via UART communication. This region also includes a button to turn the motor on and off, as well as another button to clear the terminal. Similarly, in Figure 12, region 3, the overshoot (u_p) and settling time (t_s) values for the PI controller can be configured and transmitted. These values are used by the interface to calculate the k_i and the k_p parameters, which are then sent via UART. In Figure 12, region 4, buttons are available to change the motor's rotation direction. Finally, in Figure 12, region 5, a graph displaying the motor's actual speed is presented, constructed from the information sent by the control code to the graphical interface via UART communication. Additionally, in Figure 12, region 6, a terminal is provided to display information received via UART along with other debugging logs.

G. Continuous Integration and Deployment

To ensure consistency, stability, and efficiency in the development and validation of the motor control system and the NEORV32 core, a Continuous Integration (CI) and Continuous Delivery (CD) pipeline was implemented. This pipeline enables an automated approach for testing, integrating new features, and deploying updates. Through CI, each code change is verified in real-time, quickly identifying errors and ensuring the code is always in optimal condition for integration. CD, in turn, allows approved updates to be automatically deployed to the production environment after verification, accelerating development and reducing the risk of failures.

GitHub Actions was used to implement the CI/CD, a tool that enables the creation of workflows triggered automatically whenever specified files are modified. These workflows perform a series of tests to ensure that the code maintains its functionality. Figure 13 illustrates the flow of actions that occurs in the repository when CI/CD is applied; if the tests are successful, the changes are ready to be integrated into the project. Otherwise, the modifications receive a warning indicating an issue, requiring a manual review before integration.

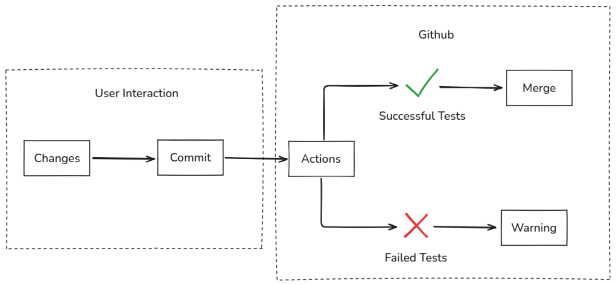


Fig. 13: Github Actions Workflow Diagram

For this project, two separate CI/CD pipelines were developed: one for hardware development and another for software development. In the hardware pipeline, a workflow was created to ensure that the code compiles in Quartus without errors, ensuring that no changes compromise hardware functionality. Additionally, the workflow generates a release of the *.sof* file, making it easily accessible on the repository's main page, eliminating the need for local recompilation to obtain the updated file.

For the software pipeline, a separate CI/CD was created.

This workflow ensures that all software files are correctly compiled through their respective Makefiles, providing an efficient and agile verification of the system's functionality.

H. Printed Circuit Board (PCB) Design

To enhance system robustness and reliability, a custom PCB was designed to replace all the jumper cables previously used. The connections for this PCB are shown in Figure 14, and its implementation is displayed in Figure 15.

I. Memory usage monitoring

In order to keep the memory usage of the project under the limit of the resources available in the FPGA, we restricted the code to using a minimum number of libraries, limiting it to the FreeRTOS library, the standard C library, and the NEORV32 hardware abstraction library [1]

Additionally, to monitor the resources utilized by the libraries and to select functions in a way that reduces memory usage, we used the tool `elf-size-analyze`, available in the Git repository[1]. This tool allowed us to investigate the RAM and ROM usages, the results were condensed in the graphics displayed in figure 16 and 17.

Monitoring memory usage was crucial in our decision to avoid the use of character-to-float conversion functions during UART readings and as these required importing the `unistd.h` library or other separate functions and the use of char or string formats was mandatory during the sending or reception of information since the NEORV32 library could only send and receive characters in char or string formats. These additions increased RAM and ROM usage by at least 20 kB to 30 kB.

V. Results

This section presents the results of the hardware-software co-design process, focusing on the integration of the NEORV32 RISC-V core with Cyclone V FPGA hardware and the proprietary *HallSector* module for precise motor control. The results highlight the successful implementation and testing of the Six-Step control algorithm in both open-loop and closed-loop configurations, including the tuning of a PI controller for speed regulation. Although the initial plan included a full implementation of the FOC algorithm, time constraints and the complexity of the approach required a shift to the Six-Step algorithm. This decision ensured that key project milestones were achieved while maintaining the focus on validating hardware requirements. Critical challenges, optimizations, and enhancements achieved during the development process are also discussed, providing valuable information for future custom RISC-V processor development.

A. Hardware-Software Interactions

As shown in Figure 18, the integration of the Cyclone V FPGA, NEORV32 core, selected peripherals, and the custom *HallSector* module has allowed us to emulate and test essential hardware functionalities for the proprietary RISC-V core. The FPGA served as a versatile platform for configuring and validating the NEORV32's functionalities, while the core's modularity allowed for targeted customization aligned with our control algorithm requirements. The peripherals—*GPIO*, *PWM*, *UART*, and *XBUS*—supported communication, motor control, and internal operations, enabling the six-step algorithm to operate within our FPGA. Additionally, the *HallSector* module specifically addressed the need for precise HALL sensor signal processing, providing reliable motor position measurements without interrupt dependencies.

To facilitate motor control and parameter adjustment, a graphical interface was developed. This interface enables visual and user-friendly motor management, including UART communication with Bluetooth support. The interface provides features for connecting to different ports, setting motor speed, configuring PI controller parameters, changing the motor's rotation direction, and visualizing motor speed in real-time. A terminal within the interface also displays UART communication logs and debugging information.

The development process leveraged CI/CD pipelines implemented through GitHub Actions. Two distinct workflows were created: one for hardware and another for software development. The hardware CI/CD workflow ensured Quartus project compilation and generated updated *.sof* files for FPGA programming. The software workflow verified correct compilation of the system using Makefiles. These automated processes enhanced development consistency and efficiency, reducing the risk of errors during updates.

For increased system reliability, a custom PCB was designed to replace jumper cables, establishing stable connections between the FPGA and the X-NUCLEO-IHM07M1. The PCB streamlined hardware integration, ensuring robustness and ease of use.

Memory usage monitoring was critical in keeping the project within the Cyclone V FPGA's resource limits. By restricting the use of libraries to only FreeRTOS, the standard C library, and the NEORV32 hardware abstraction library, memory consumption was optimized. Tools such as `elf-size-analyze` were employed to monitor RAM and ROM usage, helping identify unnecessary functions and avoiding resource-intensive operations like character-to-float conversions. These optimizations ensured efficient resource utilization for the six-step motor control algorithm.

Through these efforts, the project successfully combined hardware and software elements to implement a robust motor control system, validated on an FPGA platform with customized RISC-V functionalities.

B. Control Algorithms Implementation

With the theoretical groundwork established, the next step was to implement the BLDC motor control algorithm. The initial approach involved using a basic Six-Step algorithm without position feedback, relying uniquely on timing intervals for phase transitions. After completing this preliminary version, the focus shifted to implementing Field-Oriented Control

[1]https://github.com/jedrzejboczar/elf-size-analyze

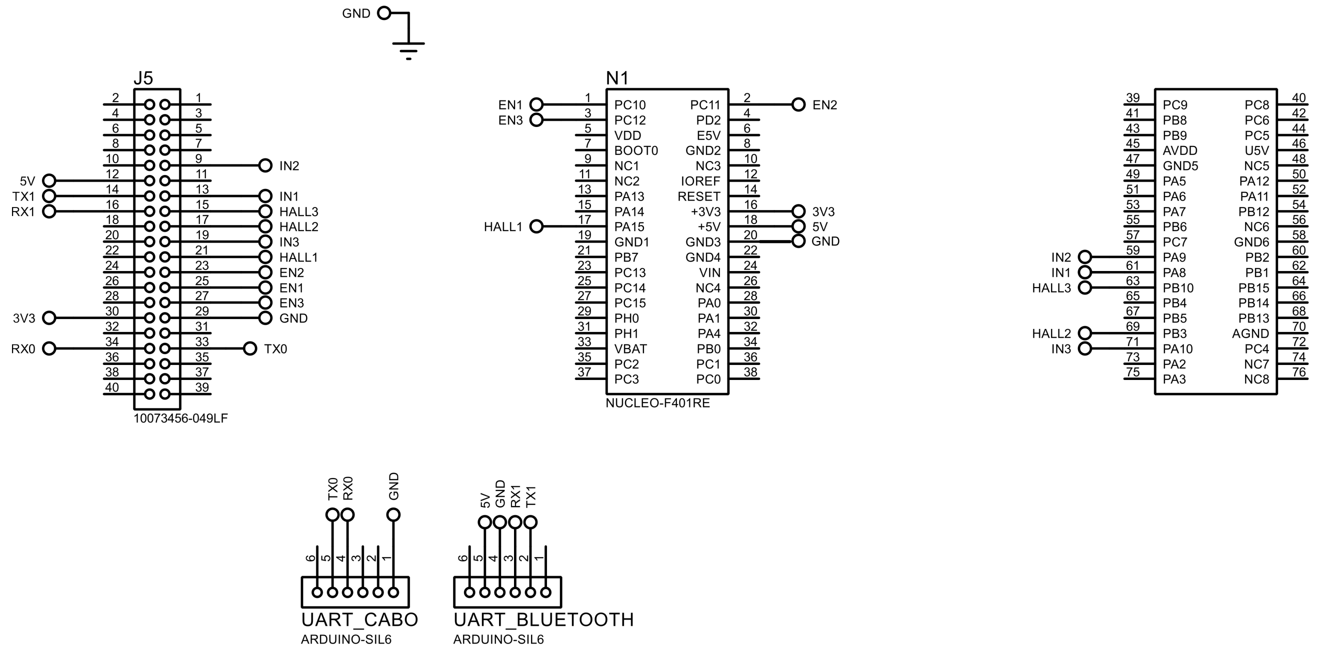


Fig. 14: Connections between FPGA and X-NUCLEO-IHM07M1

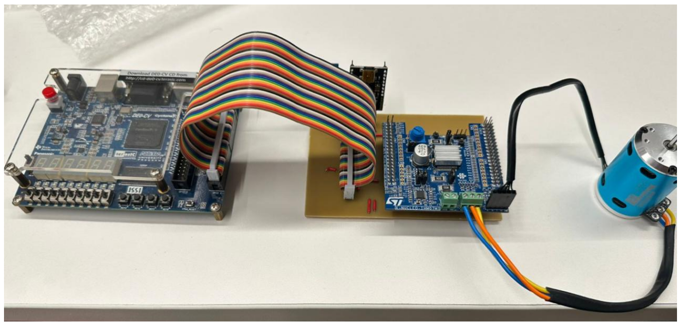


Fig. 15: Final setup with PCB

(FOC) due to its superior control precision and efficiency for BLDC motors.

An incomplete implementation of the FOC algorithm can be accessed in the GitHub Repository[2]. The following sections detail the challenges encountered during FOC implementation. First, it is essential to understand the FOC architecture, as shown in Figure 19 where the FOC control system features two feedback loops:

- **Current Loop**: Controls i_q and i_d currents, which are critical for torque and flux control.
- **Velocity Loop**: While optional, it is essential for applications requiring precise speed regulation.

Each feedback loop operates on a distinct timescale. The velocity loop typically updates at around 100 Hz, whereas the current loop requires a significantly higher frequency—around 10 kHz—for effective current regulation. Additionally, the

[2]https://github.com/Arthur-Barreto/pico-foc

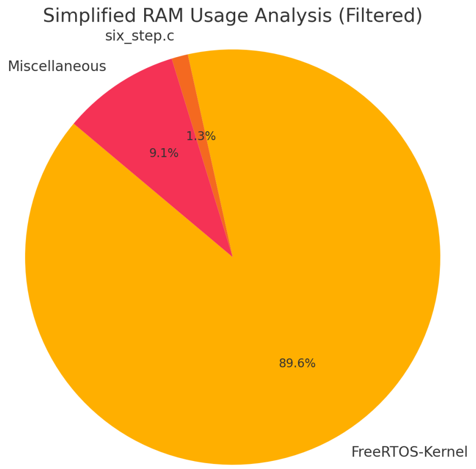


Fig. 16: RAM usage by closed loop six step control code

PWM frequency must be in the range of 20 kHz to 40 kHz to approximate a sinusoidal voltage waveform effectively.

In initial tests, incorrect frequency settings resulted in control issues. For instance, the PWM frequency was inadvertently set to 500 MHz, well above the necessary range, which led to a 0V output from the motor driver.

Another critical aspect was the timing of current measurements. Accurate readings necessitate an interrupt triggered immediately after each PWM cycle to account for transistor switching delays. This delay prevents short circuits that could occur if both the high and low transistors in a phase are activated simultaneously. Failing to manage this delay

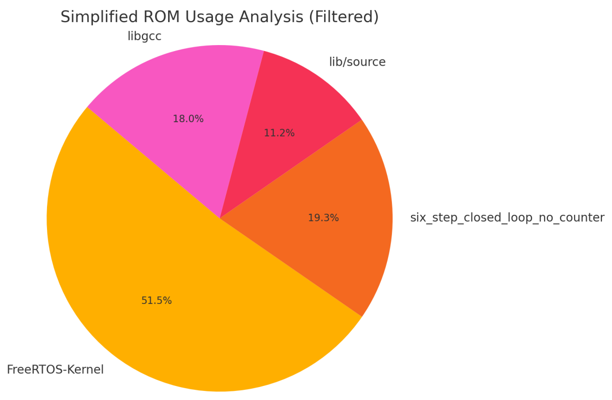


Fig. 17: ROM usage by closed loop six step control code

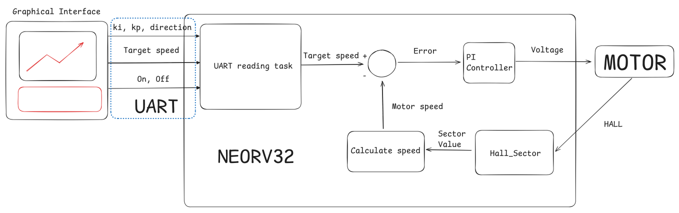


Fig. 18: General view of final hardware-software solution

can create an unintended state not represented in Figure 6, where both switches are closed, leading to erroneous current measurements in the subsequent steps.

The use of the X-NUCLEO-IHM07M1 shield resolved these electronic challenges by handling hardware-specific issues that were outside the primary scope of this project.

One key limitation was the reliance on Hall sensors, which only provide position feedback every 60 degrees. As mentioned in Chapter III-D, FOC requires continuous angular feedback to compute sine and cosine values for the Park transform. To address this, an encoder with a 1.8-degree resolution was added. Implementing a sensorless method using back EMF sensing to estimate the rotor's position could eliminate the encoder, but it would add significant complexity to the project.

Execution time for each function within the FOC pipeline presented another major challenge. Due to the high execution times, the control update frequency could not reach the target of 10 kHz, instead capping at around 1.8 kHz. The challenge was primarily in functions involving trigonometric calculations, such as the Park transform, inverse Park transform, and space vector modulation.

To address this, a hash table was implemented to store precomputed sine and cosine values for known angles, as the encoder provides position updates every 1.8 degrees. This optimization reduced trigonometric calculations to constant-time lookups ($O(1)$ complexity), significantly improving performance. Table V summarizes the execution times for each function before and after applying this optimization. To measure these values, an oscilloscope was connected to an unused GPIO pin, which was set to high before executing the function and set to low after its completion. The elapsed time recorded by the oscilloscope provided precise execution times for each function.

As shown in Table V, the execution time for the SVM function remains relatively high due to the reliance on arctan2 calculations, which are computationally intensive and depend on dynamic inputs from previous stages, complicating the implementation of a lookup table for this function.

While the FOC algorithm offers a sophisticated solution for BLDC motor control, it is not yet functioning as expected, even after addressing the timing and computational issues previously identified. Given that the primary scope of this project is to implement an application on RISC-V, rather than fully developing the FOC algorithm, the decision was made to proceed with the Six-Step algorithm using position feedback. This approach provides a reliable control solution for the BLDC motor, meeting the immediate application requirements. Moving forward, the Six-Step algorithm will be enhanced with closed-loop speed control to further improve performance.

1) Open-Loop Six-Step Control: The open-loop Six-Step control method for BLDC motors can be implemented in two ways. The first option involves using a fixed delay between each switch in the commutation sequence, allowing the motor to advance through each sector at a steady rate. This method is simple but does not account for variations in motor speed due to load changes or external factors.

The second option leverages position feedback from an encoder to determine when to switch sectors. This approach provides more accurate commutation timing, as it adapts to the motor's actual position rather than relying on a fixed delay. However, this method is still considered ”open-loop” despite using position feedback because it does not actively control the motor's speed. Instead, it only uses feedback to ensure correct sector switching based on position.

2) Closed-Loop Six-Step Control: Closed-loop Six-Step control builds upon the position feedback utilized in open-loop control but introduces an additional velocity feedback loop. In this approach, the encoder provides position data to switch sectors and calculate the motor's speed. This velocity information is then used to adjust the motor's input dynamically, enabling precise speed control.

Initially, the encoder was used for position feedback due to limitations in the NEORV32, particularly regarding the simultaneous use of external interrupts and an RTOS. However, during implementation, high noise levels were observed in the encoder readings, which compromised the accuracy of the position feedback. This noise was attributed to the imperfect physical connection between the encoder and the motor, resulting in vibrations that sometimes triggered false encoder readings. To address this issue, the approach was changed to utilize Hall sensors. This method provided the necessary position feedback to implement the six-step algorithm. External hardware was used to map the rotor's sector based on the Hall sensors' output, as detailed in Section IV-E.

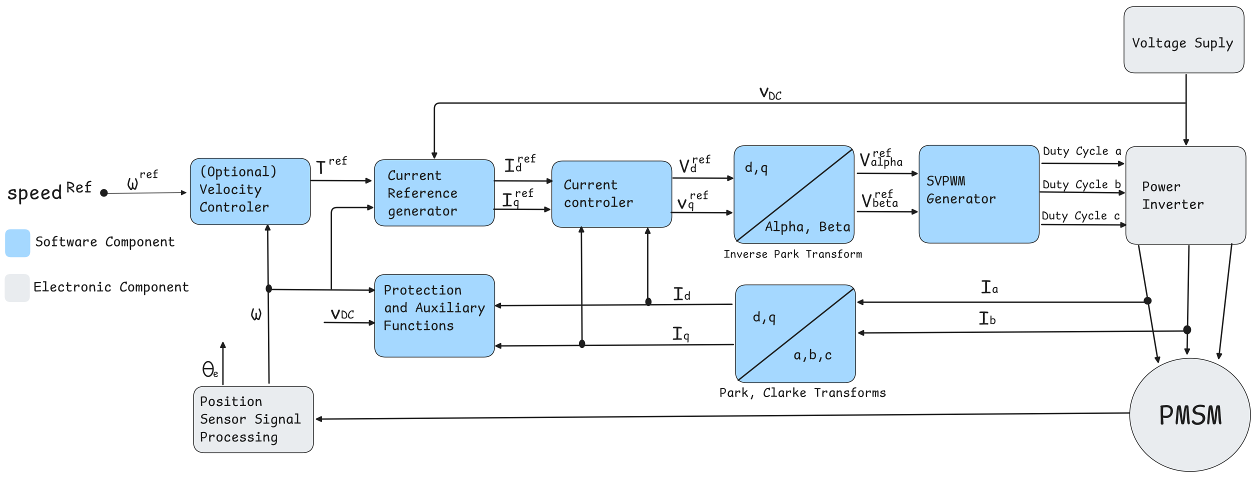


Fig. 19: FOC Architecture with Space Vector Modulation

TABLE V: Execution Time Comparison for Each Function in the FOC Algorithm (Before and After Optimization)

Function	**Execution Time (μs)**		**Speedup**
	Before Optimization	**After Optimization**	
Read Current	7.912	7.912	1.00
Clark Transform	6.368	6.368	1.00
Park Transform	65.208	7.020	9.29
Control Block	8.044	8.044	1.00
Inverse Park Transform	62.704	6.652	9.43
Space Vector Modulation	74.144	39.792	1.86
Motor Activation	5.632	5.632	1.00
Maximum Frequency (kHz)	**4.35**	**12.28**	**2.83**

After successfully obtaining accurate position feedback, the next step involved determining the motor's step response to tune the PI controller for the system. UART communication was established to receive speed feedback, as elaborated in Section IV-B. Once this communication was operational, step tests were performed by applying incremental duty cycles from 0% to 100%. The results are depicted in Figure 20.

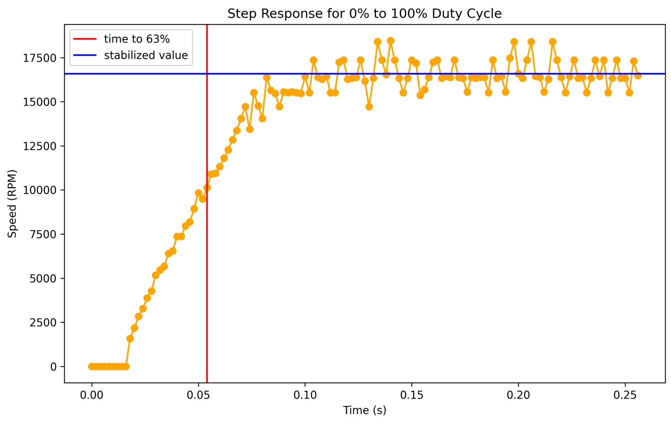


Fig. 20: Step Response

As shown in Figure 20, the motor reached 63% of its stabilized value, which averaged 16589 RPM, within 0.054 s. This time constant enabled the evaluation of the PI controller, which was designed with a 5% overshoot and a settling time of 0.27 s (five times the motor's time constant).

Table VI summarizes the calculation of Kp and Ki values. The implementation of this design is illustrated in Figure 21, which demonstrates the motor maintaining a constant speed of 1000 RPM before transitioning to 5000 RPM, confirming the PI controller's functionality. A video demonstration of this test is available on YouTube[3].

In Table VI, the variable a represents the open-loop pole of the system under study, while K denotes the proportional gain of the open-loop response. These parameters were critical in designing the PI controller to achieve the desired dynamic performance.

VI. Conclusion

VII. Conclusion

This paper delves into the exploration of RISC-V CPUs for aerospace applications, focusing on the specific use case of BLDC motor control. The open-standard nature of RISC-V ISA offers a promising avenue for developing custom solutions tailored to the stringent requirements of the aerospace industry.

The initial implementation of a basic open-loop Six-Step algorithm demonstrated the feasibility of BLDC motor control using RISC-V. However, the limitations of this approach, particularly in terms of torque efficiency and precision, led to the investigation of more advanced control techniques like FOC.

[3]https://youtu.be/mKg5MloLQb8?si=juSUi8IVlXAgQppC

a (s^{-1})	K (RPM/%duty cycle)	UP (%)	T_s (s)	K_i (RPM/s)	K_p (RPM/%)
18.519	16589	5	0.27	0.001500	0.000036

TABLE VI: Calculated K_i and K_p values for the PI controller.

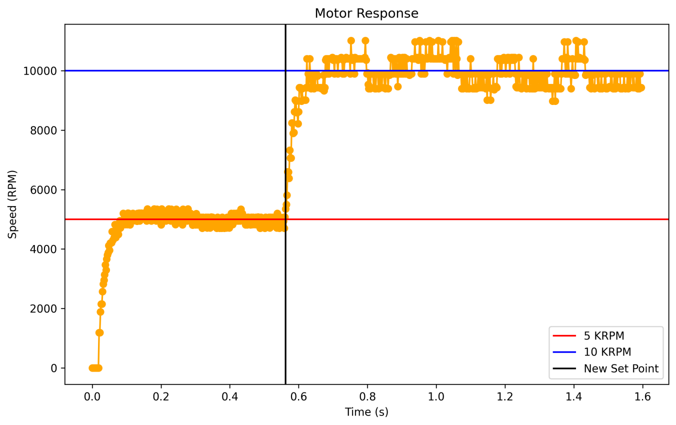


Fig. 21: PI Controller Feedback

The implementation of FOC presented several challenges, including timing constraints, current measurement accuracy, and the need for precise rotor position feedback. Although the use of encoders addressed some of these issues, further optimizations are necessary to achieve the desired performance and reliability. Considering that the main objective of the project was to map hardware requirements and constraints, the team decided not to continue working on the FOC algorithm. The final software delivered was a closed-loop Six-Step algorithm based on velocity calculations from the Hall sensor's position feedback.

Despite successfully implementing a control algorithm on an open-source RISC-V distribution, this work highlights the limitations of current open-source RISC-V distributions. These distributions are often not mature enough, presenting significant issues such as incompatibility with the FreeRTOS kernel, suboptimal implementation of components, and deviations from the original RISC-V design. These shortcomings make them unsuitable for immediate adoption by the Brazilian government, especially for critical applications like aerospace.

This research is part of a larger initiative aimed at reducing Brazil's dependence on foreign semiconductor technology and promoting the development of a domestic semiconductor industry. By exploring the potential of RISC-V, we aim to contribute to the country's technological sovereignty and enhance its capabilities in critical sectors like aerospace.

Overall, this research demonstrates the potential of RISC-V CPUs for aerospace applications, particularly in the domain of BLDC motor control. By addressing the challenges and limitations identified in this work, we can pave the way for the development of reliable and efficient RISC-V-based solutions for critical aerospace systems, contributing to Brazil's technological advancement and self-reliance.

Acknowledgment

We would like to express our sincere gratitude to all the professors who supported us throughout the development of this project. Without their contributions and guidance, this work would not have been possible. Our special thanks go to Professor Silvio Safir for his tremendous availability in assisting us with all electronic challenges, providing us with motor control shields, and offering a solid foundation through his classes to help us initiate motor control. We also extend our thanks to Professor Fabio Bobrow, whose extensive knowledge of brushless motors was fundamental at various stages. He guided us from the physical principles to the switching logic, enabling motor rotation, and his help was essential for us to understand the entire process in detail.

We are also grateful to Professors Carlos Novaes and Cauê Menegaldo, who played an essential role in our understanding of closed-loop control, providing the necessary support to implement a speed control system for the BLDC motor. In particular, Professor Novais was invaluable in guiding us through the stages of FOC, helping us identify and fix errors and offering valuable tips that saved us many hours in debugging.

Our recognition also goes to Professor Paulo Santos, a specialist in FPGA, who, in addition to clarifying numerous hardware questions, contributed insights on sensing and execution times for some critical parts of the system. We cannot overlook the laboratory technicians, who frequently assisted us with implementation questions, provided essential equipment, and shared in our joy with each significant progress.

Finally, our heartfelt thanks to Professor Rafael Corsi, our advisor, whose support was invaluable at every stage of this project. Always available—whether responding to messages or offering guidance at any time—he was more than a mentor; he became a friend and an inspiration that we will undoubtedly carry with us for life.

References

[1] S. Nolting, "Neorv32: A small, customizable and extensible mcu-class 32-bit risc-v soft-core cpu and microcontroller-like soc," 2024, accessed: 2024-11-11. [Online]. Available: https://github.com/stnolting/neorv32

[2] STMicroelectronics, *X-CUBE-MCSDK: Motor Control Software Development Kit*, 2024, accessed: 2024-11-11. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-mcsdk.html

[3] MathWorks, *Six-Step Commutation*, 2024, accessed: 2024-11-11. [Online]. Available: https://www.mathworks.com/help/mcb/ref/sixstepcommutation.html

[4] ——, *Clarke Transform*, 2024, accessed: 2024-11-11. [Online]. Available: https://www.mathworks.com/help/sps/ref/clarketransform.html

[5] ——, *Park Transform*, 2024, accessed: 2024-11-11. [Online]. Available: https://www.mathworks.com/help/mcb/ref/parktransform.html

[6] ——, *Space Vector Modulation*, 2024, accessed: 2024-11-11. [Online]. Available: https://www.mathworks.com/discovery/space-vector-modulation.html

[7] Terasic Technologies, *Terasic - DE Boards - DE0-CV Board*, 2024, accessed: 2024-12-09. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163&No=921

[8] emb4fun, "neorv32-examples," 2023, accessed: 2024-11-12. [Online]. Available: https://github.com/emb4fun/neorv32-examples/tree/master

[9] Open Source Initiative, *BSD 3-Clause License*, 2024, accessed: 2024-11-12. [Online]. Available: https://opensource.org/license/bsd-3-clause

[10] Wishbone Interconnect Project, *Wishbone Interconnect Documentation: Introduction*, 2024, accessed: 2024-11-12. [Online]. Available: https://wishbone-interconnect.readthedocs.io/en/latest/01_introduction.html

[11] RISC-V International, *Specifications*, 2024, accessed: 2024-11-11. [Online]. Available: https://riscv.org/technical/specifications/

[12] Quma78, "FPU Float to Signed Integer," 2024, accessed: 2024-11-12. [Online]. Available: https://github.com/stnolting/neorv32/issues/942

[13] S. Nolting, *NEORV32 Documentation*, 2024, accessed: 2024-11-11. [Online]. Available: https://stnolting.github.io/neorv32/

[14] thibautgravey, "XIRQ on FreeRTOS," 2022, accessed: 2024-12-07. [Online]. Available: https://github.com/stnolting/neorv32/discussions/347