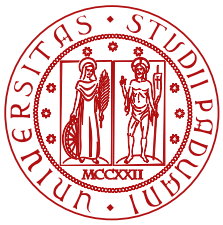


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN ELECTRONIC ENGINEERING

FPGA-Based Stepper Motor Control Board with Automatic Acceleration Management

Supervisor
Prof. Daniele Vogrig

Author
Alberto Ragazzo

ACADEMIC YEAR 2024-2025

Graduation date 08/10/2025

Prima di procedere con la trattazione, desidero dedicare alcune righe a tutti coloro che mi sono stati vicini durante questo percorso di crescita personale e professionale.

Innanzitutto, desidero esprimere la mia sincera gratitudine al Professor Daniele Vogrig, relatore di questo lavoro, per la sua pazienza, i preziosi consigli e la condivisione della sua esperienza durante l'intero processo di redazione di questa tesi.

Un grazie di cuore va ai miei genitori, che non hanno mai smesso di credere in me, sostenendomi con amore e lasciandomi spazio per essere me stesso, rispettando le mie scelte e i miei tempi.

Un caloroso grazie anche ai miei amici, sia a chi ne è consapevole, sia a chi forse ancora non se ne è reso conto, per avermi accompagnato con la loro presenza, le risate e i momenti condivisi, rendendo questi anni più sereni.

Un sincero ringraziamento infine ai miei colleghi Alessandro e Manuel, il cui prezioso supporto è stato fondamentale per il completamento con successo di questo progetto.

Abstract

This master's thesis aims to design and develop an electronic board for the control of stepper motors, with a particular focus on the use of FPGA devices for intelligent acceleration management. Following an introductory overview of stepper motors and their main types, the work analyzes the driving techniques, control methods, and criteria for selecting electronic components.
The project is intended to develop a control board that drives a mechanical system for a probe used in display testing. Unlike traditional solutions based on microcontrollers, the use of an FPGA allows for more precise and faster control of the stepper motors, leveraging the high parallelism and flexibility offered by such devices.
The thesis describes the entire board design process, from the choice of the hardware architecture to the realization of the layout, including the integration of an FPGA unit programmed to execute motion control algorithms. The work also includes the development of VHDL code, the simulation and functional verification of the system, as well as experimental tests conducted on the prototype to evaluate its overall performance.

Sommario

Questa tesi magistrale ha l'obiettivo di progettare e sviluppare una scheda elettronica per il controllo di motori passo-passo, con particolare attenzione all'uso di dispositivi FPGA per una gestione intelligente delle accelerazioni. Dopo una panoramica introduttiva sui motori passo-passo e le loro principali tipologie, il lavoro analizza le tecniche di pilotaggio, i metodi di controllo e i criteri di scelta dei componenti elettronici.
Il progetto prevede lo sviluppo di una scheda di controllo che gestisca un sistema meccanico per una sonda utilizzata nel collaudo di display. A differenza delle soluzioni tradizionali basate su microcontrollori, l'impiego di un FPGA consente un controllo più preciso e veloce dei motori passo-passo, sfruttando l'elevato parallelismo e la flessibilità offerti da tali dispositivi.
La tesi descrive l'intero processo di progettazione della scheda, dalla scelta dell'architettura hardware alla realizzazione del layout, includendo l'integrazione di un'unità FPGA programmata per eseguire algoritmi di controllo del moto. Il lavoro comprende inoltre lo sviluppo del codice VHDL, la simulazione e la verifica funzionale del sistema, oltre a prove sperimentali effettuate sul prototipo per valutarne le prestazioni complessive.

Contents

Chapter 1

Introduction

In almost all applications requiring precise and controlled movements, there is a need to drive electric motors. This need for **precision** is particularly evident in fields such as robotics, industrial automation, and precision mechanics, where even a small error in movement could compromise the entire system. Stepper motors, also known as step motors or stepping motors, are especially valued for their ability to provide accurate movements, making them ideal for a wide range of uses, such as controlling robotic arms. Each type of electric motor construction technology requires a specific type of drive: a brushed DC motor can be rotated simply by supplying sufficient DC voltage to its terminals, while this is not enough for a brushless motor. The device that interfaces between the motor and the control system is called driver, and it plays a crucial role as the interface between the control system and the motor, enabling the control of the high power required by the motor through low-power signals.

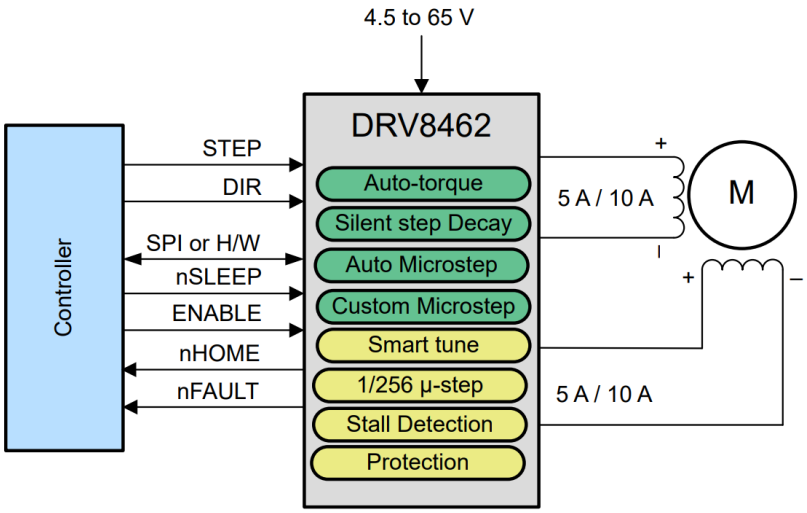


Figure 1.1: Simplified block diagram of a motor driver. [2]

The input interface to a motor driver is responsible for communicating the desired functionalities to the motor, while respecting the limits of the circuitry and the mechanical system.
The internal logic of the driver processes our request by outputting signals suitable to drive the generic amplification stage, which in turn adjusts the electrical quantities from the power supply and appropriately converts them to drive the electric motor connected at its output. Currently, in the world of stepper motor drivers, there are generally two main options:

1. **Pulse input drivers (step and direction):** These drivers receive pulse signals indicating the number of steps and the direction of movement. They are simple to implement, but require the microcontroller to generate high-speed and variable-frequency signals. This can be problematic since microcontrollers typically do not have dedicated peripherals to generate such signals. As a result, generic timers must be used, handling a considerable number of interrupts, which can overload the microcontroller, especially if it has to manage multiple axes (thus many motors) and also handle a high-speed data bus like Ethernet.

2. **Drivers with digital interface:** These drivers allow setting the number of steps and speed through a communication bus. In this case, counting pulses is unnecessary, as they are generated by the driver itself. However, it is necessary to continuously write to the driver's bus to break down a large movement into smaller steps and keep reprogramming the speed. This approach can be inefficient and requires a significant workload for the microcontroller.

A system requiring high speed, further increased by microstepping, demands proper management of accelerations and decelerations. If acceleration management is delegated to the microcontroller, the workload can become particularly heavy. This is especially true if the microcontroller must manage multiple axes and also handle a high-speed data bus. In this scenario, a bus-configurable FPGA at startup with desired acceleration profiles, which then receives as input only the number of steps to execute on each axis, would be particularly useful to relieve the microcontroller from managing accelerations and decelerations. Thanks to their parallel processing capability and flexible configuration, FPGAs can efficiently handle high-speed signals and frequency variations, enabling speeds significantly higher than those achievable by a microcontroller-based control alone. This approach not only improves the overall performance of the system, but also reduces the workload of the microcontroller, allowing it to focus on other critical functions.
The overall system has been designed to control the motors integrated into several **FUYU FSK40J** linear modules, selected for their reliability, precision and compactness. These linear actuators, which use ball screw transmission and linear guide rails, are particularly well suited for applications requiring fast, repetitive motion with high positioning accuracy.

In the context of this project, the FSK40J modules will be used within an automated machine for testing touch displays, developed for use in an industrial production environment. The system will be capable of coordinating the motion of multiple motorized axes in order to precisely probe the entire surface of the panel and verify its functionality through repeatable and cyclic tests.

Figure 1.2: FUYU FSK40J linear actuator used in the testing system. [5]

The designed control board includes:

- **Voltage regulators:** essential to ensure that each component receives the appropriate voltage, preventing malfunctions and optimizing system performance.
- **Microcontroller:** an ARM Cortex M3 microcontroller, the LPC1768, with related communication protocols to manage control and interface operations.
- **FPGA:** a Lattice Semiconductor iCE40 HX1K VQ100, configured via an external flash. This provides crucial flexibility and parallel processing capability for managing motor accelerations.
- **Drivers:** The system employs multiple DRV8462 drivers by Texas Instruments. A minimum of three drivers is required to enable movement along three dimensions; however, up to six drivers have been used in this project to achieve more complex and precise motion.

Chapter 2

Stepper Motors

2.1 Introduction to Stepper Motors

The first modern stepper motors were developed around 1920 and initially found application in the British Navy, where they were used for the remote positioning control of guns and torpedoes. Since then, these electromechanical actuators have undergone continuous evolution, both in terms of structure and applications, due to their ability to convert digital signals into precise mechanical movements through discrete angular increments known as "*steps*". For each control pulse received from the driving system, the motor performs a fixed rotation of the shaft, making it possible to predict the angular displacement based on the number of input pulses. This operating principle enables open-loop position control without the need for feedback sensors.

Figure 2.1: Various versions of stepper motors belonging to the NEMA family.

Contrary to the intuitive notion of a slow and segmented system, modern stepper motors are capable of executing each step in just a few milliseconds. At stepping rates of several thousand pulses per second (pps), the shaft rotation appears smooth and continuous, exhibiting behavior similar to that of a traditional electric motor. This characteristic, combined with the ease of control via microcontrollers or FPGAs, makes stepper motors ideal for applications requiring accurate and repeatable positioning, such as plotters, printers, hard disk drive actuators, small-scale CNC (Computer Numerical Control) systems, and robotic arms.

From a structural standpoint, there are three main types of stepper motors: variable reluctance motors, which operate based on the principle of minimum magnetic reluctance; permanent magnet motors, which rely on the interaction between rotor magnets and the stator-generated magnetic field; and hybrid stepper motors, which combine features of both to maximize torque and precision. Modern stepper motors can achieve very small step angles, down to less than 2°, thanks to multi-stack mechanical designs and the optimization of internal geometry. The available torque values range from a few $\mathrm{mN} \cdot \mathrm{m}$ in miniature models to several tens of $\mathrm{N} \cdot \mathrm{m}$ in industrial units with diameters of up to 15 cm.

Another key advantage of these motors is the absence of cumulative positioning error: each step is independent of the previous one and does not drift over time. Furthermore, the simplicity of control, together with the increasing availability of compact switching devices, has encouraged their use since the 1960s in automated systems such as numerically controlled milling machines. Today, stepper motors are widely used in industrial drives and robotic systems that require a high torque-to-inertia ratio and low electrical response times, ensuring satisfactory dynamic performance even at high speeds.

2.2 Motor Drive

2.2.1 Open-Loop Control

Open-loop control represents the simplest and most cost-effective method for driving a stepper motor and is particularly common in systems where continuous feedback on the motor shaft position is not required. In this control scheme, each pulse sent to the motor driver corresponds to a precise angular displacement of the shaft, determined by the motor's step angle. The operation is therefore based on the direct and unambiguous relationship between the number of pulses received and the motor's rotational angle: for example, a motor with a step angle of $1{,}8^\circ$ will complete a full rotation after 200 pulses. The system is called "open-loop" because it does not include feedback control. No direct measurements of the actual shaft position are performed; instead, the system merely counts the pulses sent, assuming that the motor accurately executes each step. This characteristic makes open-loop control simple to implement and particularly

suited to applications where the loads are known and controllable, such as in small positioning systems, 3D printers, or pick-and-place machines.

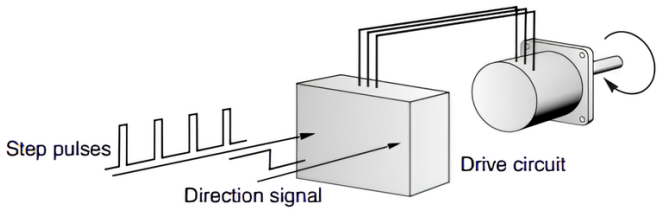


Figure 2.2.1: Open-loop control scheme with a stepper motor. [6]

In a typical scheme (**Figure 2.2.1**), the motor is driven by a driver circuit that receives two low-power digital signals: one represents the step pulse, and the other the rotation direction. For each step pulse, the motor rotates by one step in the direction specified by the direction signal. The direction can be clockwise or counterclockwise, depending on the logic state of the signal. This allows for digital control of the absolute shaft position, provided the system starts from a known reference position. However, the open-loop system has intrinsic limitations: in the absence of feedback, any anomaly, such as a missed step due to overload or overly abrupt acceleration, cannot be detected automatically. In scenarios where absolute precision is critical or where unpredictable load variations occur, it may be necessary to adopt closed-loop control or implement position sensors to monitor the motor's actual movement.
Another important aspect to consider is that, being an incremental system, knowledge of the absolute shaft position depends on the initial position. For this reason, during startup, it is often necessary to perform a homing procedure, that is, returning to a mechanically known position (such as a limit switch), which allows resetting the pulse counter and starting the motion sequence from a reliable baseline.

2.2.2 Step Pulse Generation

The generation of step pulses represents the core operation of an open-loop control system using stepper motors. Each pulse corresponds to a movement command for the motor, which responds by rotating by its step angle. The frequency and timing of these pulses directly determine the motor's speed and acceleration, making their management a critical aspect in every application. Pulses can be generated by a wide range of devices, including simple oscillator circuits, programmable digital controllers, microprocessors, or microcontrollers. In more complex systems, pulse generation is entrusted to microcontrollers or FPGAs, which allow fine and programmable control of both the number of pulses and their frequency. When the system must perform a cer-

tain displacement, the controller generates a sequence of pulses sent to the motor driver. The total number of pulses determines the total rotation angle, while the pulse frequency determines the rotational speed.

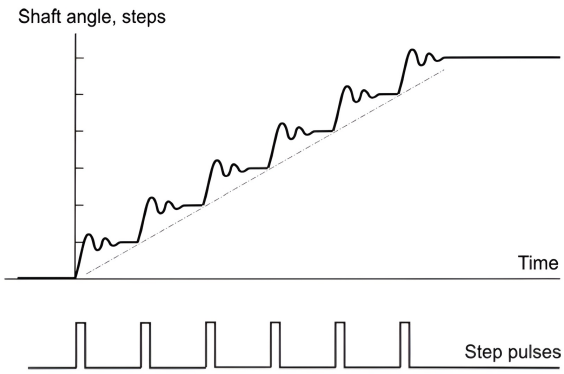


Figure 2.2.2: Typical motor response to a low-frequency pulse sequence. [6]

As illustrated in **Figure 2.2.2**, a typical sequence of six pulses evenly spaced in time causes the motor to execute six consecutive steps. In this example, the average speed of the motor shaft can be visualized as the slope of the dashed line connecting the end points of each step. For the same number of pulses, a higher frequency corresponds to a faster motor movement. Despite the discrete nature of the command, the resulting motion can be either smooth or jerky depending on the pulse frequency and the system's dynamic characteristics. Each step requires a finite time to complete: the rotor moves towards the new position but may slightly overshoot and oscillate before settling. These overshoot and settling phenomena are influenced by the rotor mass, the applied load, and the overall mechanical properties. Typically, the settling time for a single step ranges from a few milliseconds up to about 100 ms.

Another crucial point is that, to ensure system reliability, the initial position of the motor must be known. Since the stepper motor is an incremental actuator, all pulse counting is based on the assumption that the motor starts from a precise position. Therefore, at system power-up or before beginning a new motion sequence, it is essential to perform a homing or calibration procedure, resetting the pulse counter while the motor is in a physically known reference position. From that moment onward, each pulse can be counted to derive the absolute position. The accuracy of the system therefore depends both on the quality of pulse generation and the motor's ability to faithfully follow them. Under normal operating conditions, without excessive loads or overly rapid accelerations, the correspondence between generated pulses and executed steps remains perfect. However, under more severe dynamic conditions, it becomes necessary to introduce speed management strategies, such as *ramping*, to avoid loss of steps and ensure the reliability of the system.

2.2.3 Ramping and High-Speed Operation

The concept of ramping (gradual acceleration and deceleration) is essential for the optimal operation of a stepper motor system, especially when the motor operates at high speeds. Although stepper motors are well known for their ability to execute precise movements through sequences of pulses, their behavior at high speeds requires careful management of the transition timings between the various acceleration and deceleration phases. When a motor is commanded to run at very high speeds, for example 2000 step/s or more, its behavior significantly changes compared to low-speed operation.

At low speeds, the motor easily follows each discrete pulse, resulting in a somewhat "stepped" motion. However, as the pulse frequency increases, the motor enters the slewing mode, in which the response becomes progressively smoother and the rotor movement is no longer perceived as discrete jumps from one position to another. In this mode, the one-to-one correspondence between pulses and motor steps is still maintained, preserving the exact relationship between controller-generated pulses and executed steps. However, acceleration to such high speeds cannot occur instantaneously. If the motor is commanded directly to high speed, it will not be able to ramp up quickly enough and will struggle to keep up with the pulse train. In this case, the motor will fail to follow the pulses, resulting in slip or stall, which causes position loss. To prevent this issue, ramping is employed, which consists of gradually increasing the pulse frequency. Ramping allows the motor to progressively adapt to the desired speed, starting from a low step rate and gradually increasing the pulse frequency incrementally until reaching the maximum speed. This ramping operation is critical because it prevents stalling or slipping phenomena and ensures the motor can perform the requested movements without missed steps. In industrial systems or automation plants, ramping is managed by digital controllers that finely regulate the pulse frequency according to the specific application requirements.

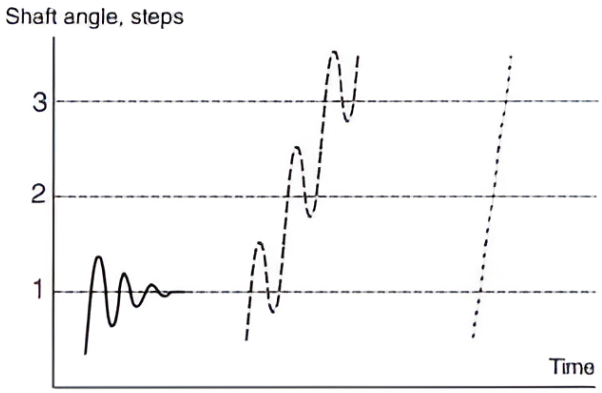


Figure 2.2.3: Position-time responses at low, medium, and high pulse frequencies. [6]

The ability of a stepper motor to operate at high speeds also depends on its construction and mechanical characteristics. For instance, the mechanical resistance of the rotor, bearing stiffness, and stall torque influence how rapidly the motor can be accelerated. Moreover, vibrations and oscillations commonly occur during high-speed operation and can only be minimized through accurate control of speed and position. Applications requiring high movement speeds use ramping to smoothly manage the transition from low to high speed. Even in these cases, when the motor reaches the optimal operating speed, ramping management allows for precise position control and prevents step loss or motion errors.
If ramping is not properly executed, stepper motors can lose steps. Moreover, in the presence of high loads or incorrect ramping programming, there is a risk that the speed increases too rapidly, causing the motor to be unable to respond quickly enough. In such cases, the motor may stall, compromising the overall position accuracy of the system. Therefore, managing acceleration and deceleration speeds through ramping is a fundamental aspect in the design of high-speed control systems using stepper motors.

2.2.4 Positioning Systems

Positioning systems using stepper motors are employed in numerous applications to achieve precise and controlled movements. However, the performance of the motor can vary depending on the operating conditions, particularly with respect to the applied load. Understanding the differences between no-load operation, nominal load, and intermediate torque is essential to comprehend how a stepper motor responds to various positioning demands.

- **No-load operation:** This occurs when the motor is not subjected to any external load. In this situation, the motor can operate at a higher rotational speed and respond more quickly since it does not have to overcome any additional resistance. In other words, the motor can move its rotor freely without the need to generate significant torque to counteract resisting forces. This type of operation is typical in applications where the motor must perform fast and light movements, such as precision positioning devices where the load is minimal or absent, like the head positioning systems in optical drives.

- **Nominal load operation:** The situation changes when the motor operates under nominal load. The nominal load represents the amount of force the motor is designed to continuously withstand without damage. In this condition, the motor must generate enough torque to overcome the load resistance and maintain the desired speed. Operating at nominal load requires precise control of the current and voltage applied to the motor windings, since the motor must produce the torque necessary to move the load without causing overheating or mechanical damage. Industrial applications such as positioning control in automatic

assembly systems or cutting machines are examples where the motor works under nominal load. In these conditions, the ability to maintain constant torque is essential to ensure the motor does not lose steps or compromise positioning accuracy.

- **Intermediate torque operation:** In situations of intermediate torque, the motor is subjected to a load that lies between no load and nominal load. Here, the motor does not need to produce the maximum possible torque, but still faces significant resistance. The performance of the motor under these conditions depends on its ability to adapt to changes in resisting forces and regulate its behavior to ensure precise and controlled movement. Applications involving complex positioning, such as the movement of components in automation processes with variable loads, require accurate management of torque and speed to avoid overloads and guarantee correct positioning. Managing intermediate torque is therefore essential for optimal response in dynamic systems, where the load can vary frequently or rapidly.

The performance of a stepper motor is therefore closely linked to the type of load it must handle. The design of the control system and the selection of the motor must consider these variables to optimize the efficiency and precision of the positioning system. Motors operating under nominal or intermediate load conditions usually require more sophisticated control to manage torque fluctuations and maintain stable, error-free movement, especially when loads change in real time.

2.3 Operating Modes

The stepper motor is a two-phase brushless synchronous motor, consisting of a stator with electromagnetic windings and a rotor magnetized in segments. The direct current supply to the windings generates alternating magnetic fields that attract or repel the rotor, causing it to rotate in discrete steps. By reversing the direction of the current in the windings, it is possible to control the motor's position and speed with high precision, making it particularly suitable for applications requiring accurate motion control. This type of motor has several operating modes, each influencing positioning accuracy, dynamic response, and energy efficiency. The main modes include **full-step**, **half-stepping**, **mini-stepping**, and the more advanced **microstepping**. These modes mainly differ in angular resolution and motion quality, significantly determining the motor's performance in specific applications.

2.3.1 Full Step Operation

In full-step operation, the motor rotates by a fixed angle with each pulse received. The rotor moves sharply and sequentially from one stable position to another, following a precise excitation sequence of the phases. For example, in a motor with 200 step/revolution, each pulse causes a rotation of $1{,}8°$. This type of operation can be implemented in two variants: with single excitation, where only one phase is active at a time, or with double excitation, where two phases are activated simultaneously. The first option helps contain energy consumption, but at the expense of torque; the second instead offers higher torque, at the cost of greater power dissipation. Full-step operation is simple and robust, making it suitable for many applications where the load is predictable and the system operates at low or moderate speeds. However, the motion occurs in discrete jumps, which can produce vibrations, noise, and resonance phenomena, especially at low speeds. Furthermore, the angular resolution is limited to the motor's step angle, which may be insufficient in high-precision applications or where greater smoothness of motion is required.

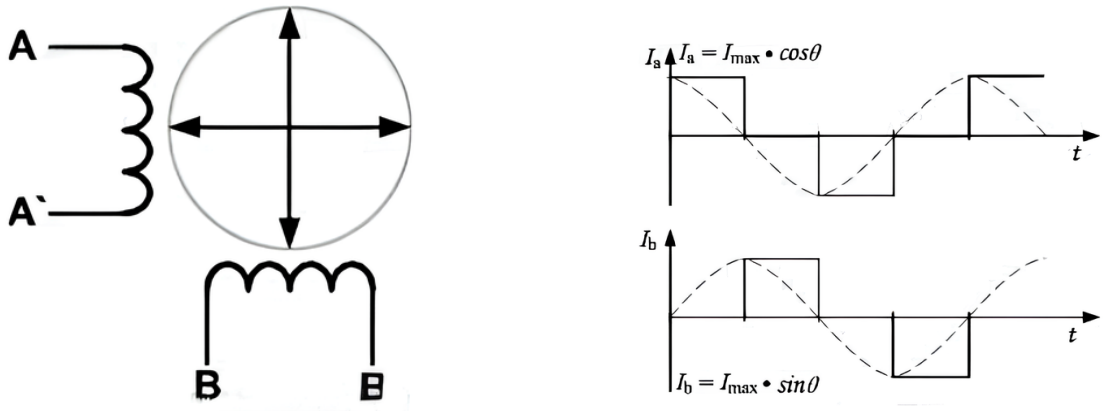


Figure 2.3.1: Vector representation (left) and voltage-current diagram (right) in full-step. [7]

2.3.2 Half Step Operation

Half-stepping represents an intermediate solution between full-step and microstepping. In this mode, the excitation alternates between one and two active phases, thus generating intermediate positions between the full steps. This allows achieving twice the resolution compared to full-step: for example, in a motor with a nominal step angle of $1{,}8°$, half-stepping enables movements of $0{,}9°$. The main benefit of half-stepping is the increased precision, which can be useful in applications requiring higher resolution without overly complicating the control electronics. Additionally, the movement is smoother compared to full-step, as the intermediate steps help dampen the stepping effect. However, the torque produced during the intermediate steps (when

only one phase is active) is lower than that of the steps with two active phases, which can cause slight variations in speed and dynamic behavior of the system, especially in the presence of light loads or undamped mechanics.

.

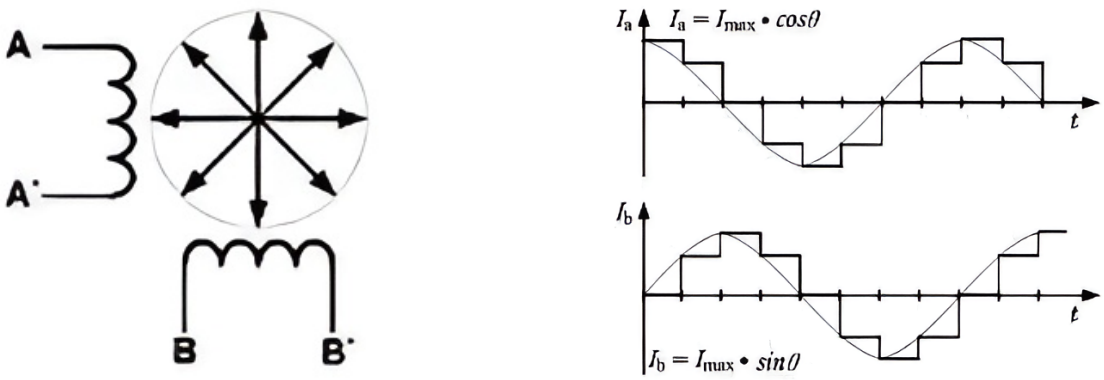


Figure 2.3.2: Vector representation (left) and voltage-current diagram (right) in half-step mode. [7]

2.3.3 Microstep Operation

Microstepping is an advanced technique that allows each full step to be subdivided into a large number of microsteps. This is made possible through precise analog or digital control of the currents supplied to the motor phases, generating a sinusoidal or pseudo-sinusoidal waveform. Modern drivers can achieve extremely high resolutions, even exceeding 1000 microstep/revolution.

The most obvious advantage of microstepping is the extremely smooth motion it produces. Vibrations are greatly reduced, making this mode ideal in contexts where quietness and smoothness are fundamental requirements, such as in precision robotics, optical systems, and biomedical equipment. Additionally, the sinusoidal approximation of the internal magnetic field makes the motor behave similarly to a synchronous motor, with torque distributed more evenly throughout the rotation. However, microstepping has its limitations. As the number of microsteps increases, the available torque per microstep decreases drastically. For example, with 16 microsteps, the useful torque per microstep can drop below 10% of the nominal torque. This means that microstepping should primarily be used to improve resolution and motion quality, not to increase motor force. Moreover, its implementation requires sophisticated drivers capable of generating the correct waveforms via PWM modulation or the use of DACs and current tables managed by microcontrollers or FPGAs. Errors in current control can compromise the linearity of the movement, negating the theoretical advantages of the technique.

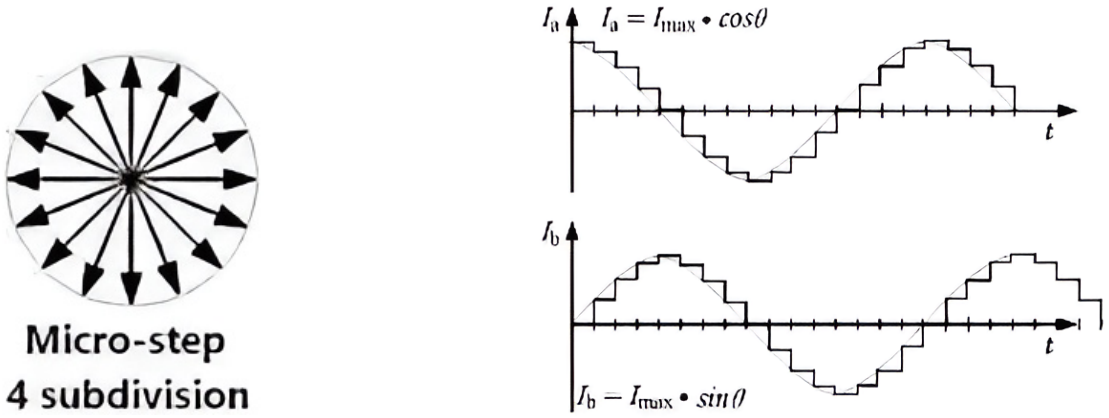


Figure 2.3.3: Vector representation (left) and voltage-current diagram (right) in microstep mode. [7]

2.3.4 Dynamic Effects and Real Behavior

In real-world contexts, the behavior of stepper motors is influenced by numerous factors that alter their theoretical performance. Load characteristics, inertia, mechanical resonances, and power supply conditions largely determine the quality of movement and positioning reliability. One of the most common problems is step loss, which occurs when the required torque exceeds the available torque at a given moment. In open-loop systems, this error goes undetected and can accumulate over time, compromising the system accuracy. Another important phenomenon involves mechanical resonances, which can occur at specific frequencies, amplifying vibrations to the point of causing instability. To counteract these effects, mechanical or electronic damping techniques can be applied, or the pulse frequency can be dynamically adjusted to avoid critical resonance bands.

To ensure reliable and stable operation, it is important to choose a motor with adequate holding torque and adopt acceleration and deceleration profiles that reduce dynamic stresses. These ramping profiles can be linear or variable in shape, implemented through dedicated logic or software algorithms. Furthermore, in systems requiring high precision and secure positioning, the use of closed-loop control is recommended, allowing detection and correction of any errors. In conclusion, the choice of the most suitable operating mode for a stepper motor depends closely on the application context. Full-step and half-step modes are suited for simple and less demanding systems, while microstepping is ideal for high-precision applications, although it requires more careful and sophisticated design. Understanding the limits and potentials of each mode is fundamental to developing reliable, efficient, and tailored solutions for every need.

2.4 Categories of Stepper Motors

2.4.1 Variable Reluctance Motors

Figure 2.4.1a schematically shows the internal section of a variable reluctance stepper motor, which has three phases and four teeth on the rotor. Both the stator and the rotor are made of ferromagnetic material, typically soft steel, and exhibit pronounced radial anisotropy (that is, a variation in magnetic reluctance along different radial directions) achieved through the particular geometric design of the rotor and stator. This characteristic is fundamental to the motor's operating principle. Each stator phase consists of multiple windings distributed in diametrically opposite pairs of pole extensions, called pole pairs. The figure illustrates the simplified case of a three-phase winding, where each phase is associated with only one pair of poles. The rotor is equipped with a number D_r of equidistant teeth, separated by a constant angle α_r, defined as the step angle of the rotor.

$$\alpha_r = \frac{2\pi}{D_r} \tag{2.1}$$

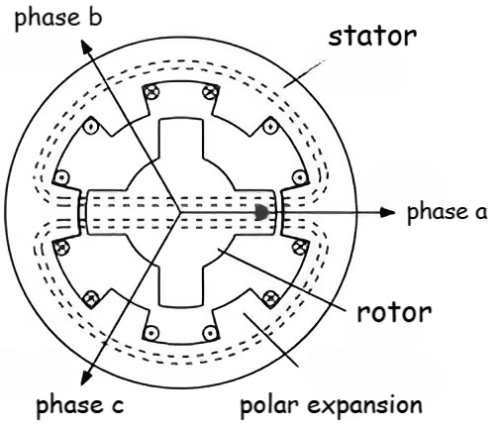


Figure 2.4.1a: Structure of a Variable Reluctance Stepper Motor. [8]

The operating principle of the variable reluctance motor can be described as follows. Consider the initial condition illustrated in **Figure 2.4.1a**, where phase A is supplied with constant direct current and the motor is running unloaded. In this configuration, the rotor is placed so that a pair of its teeth aligns with the magnetic axis of the energized phase, thus reaching a stable equilibrium position characterized by minimal reluctance. If the supply to phase A is interrupted and phase B is energized, an electromagnetic torque is generated on the rotor, causing it to rotate counterclockwise until the pair of teeth nearest to the newly energized phase aligns with its axis. This position also corresponds to a minimum reluctance configuration. The angular rotation

performed during this process is defined as the step angle α_p , and the number of steps per revolution of the motor is given by the relation:

$$N_p = \frac{2\pi}{\alpha P} \tag{2.2}$$

This parameter is fundamental because it determines the angular resolution of the motor, that is, the degree of precision with which a mechanical load directly connected to the shaft can be positioned. Repetition of the process with the energization of phase c means that the rotor takes an additional step forward (still in the counterclockwise direction), as shown in **Figure 2.4.1b**.

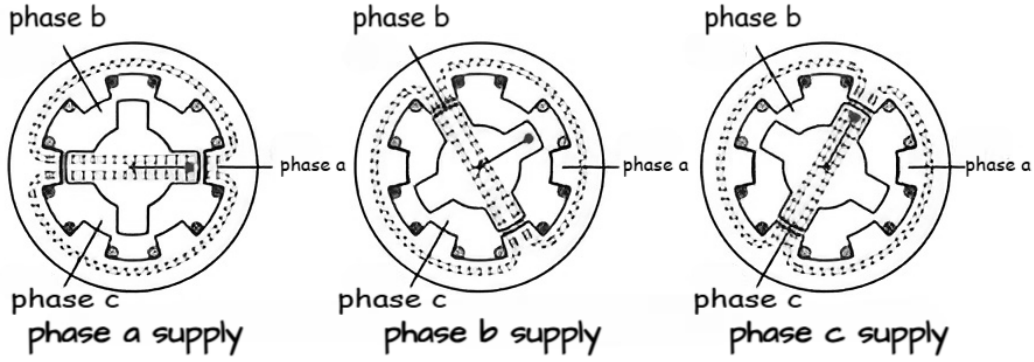


Figure 2.4.1b: Rotor positions in two successive steps. [8]

Another version of the variable reluctance motor is the multi-stack type, which consists of several magnetically independent sections (typically three) within a single housing. In a three-stack motor, the rotor is made up of three separate sections, each with an equal number of teeth spaced evenly but shifted by one-third relative to the teeth of the other sections. The stator also has three separate stacks, each similar to the stator of a hybrid motor. In this type of motor, each stator has a coil that excites all its poles. Each stator is powered sequentially, so the rotor teeth of each stack align with the corresponding teeth of the stator, allowing the motor to perform stepping. The variable reluctance motor is less commonly used due to its relatively poor performance in terms of torque and efficiency. Furthermore, the absence of permanent magnets results in the lack of a holding torque: in case of power loss, the rotor is unable to maintain the load position. These limitations reduce the effectiveness of the motor in applications that require precise holding of position without power supply.

2.4.2 Permanent Magnet Motors

Permanent magnet stepper motors represent an intermediate category between variable reluctance motors and hybrid motors, distinguished by the presence in the rotor of a permanently

magnetized magnetic material, without teeth, which provides the system with torque even in the absence of power supply. The stator has a structure similar to that of variable reluctance motors, with windings arranged on polar expansions, while the rotor consists of one or more permanent magnetic poles, whose polarity is essential to determine the interaction with the magnetic field generated by the stator phases. The operation of these motors is based on the alignment of the rotor with the magnetic field generated by the energized stator winding. For example, if phase A is powered with a positive current, the rotor tends to align its poles with the stator teeth corresponding to phase A. If phase B is subsequently energized with current of opposite polarity, the rotor makes a step in the counterclockwise direction; conversely, if the current has the same polarity, the rotation occurs clockwise. Therefore, the direction of rotation is controlled by the sequence and polarity of the currents applied to the phases, making this type of motor suitable for applications requiring precise positioning and bidirectional motion control.

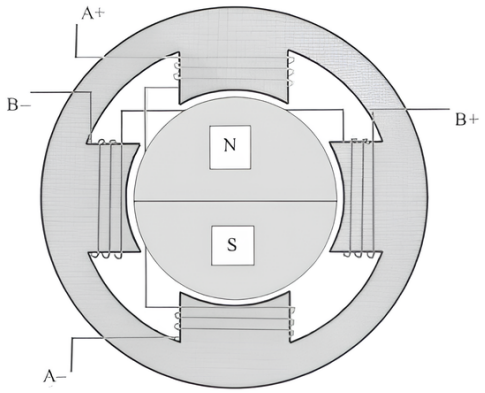


Figure 2.4.2: Structure of a permanent magnet motor. [10]

A distinctive feature of these motors is the detent torque, which is the resisting torque present even when the phases are not energized, tending to hold the rotor in preferred positions of magnetic equilibrium. This characteristic is particularly useful in applications where it is necessary to maintain the position without power supply, avoiding the use of brakes or other mechanical devices. Typically, the detent torque represents between 5% and 20% of the torque generated during active operation. However, in some cases, it can cause undesirable effects such as *cogging*, which is an uneven movement caused by the periodic interaction between the stator and rotor. To reduce this effect, the shape of the stator teeth can be modified to reduce anisotropy and make the movement smoother. Compared to variable reluctance motors, permanent magnet motors generally offer higher torque for the same size and better efficiency, thanks to the presence of the permanent magnetic field that contributes to torque generation without the need

for continuous excitation. However, the use of high-quality magnetic materials, such as rare earth elements, can significantly impact the motor cost, making it more suitable for applications where performance, compactness, and reliability are prioritized over cost.

2.4.3 Linear Motors

Linear stepping motors are a specific type of electromechanical actuators designed to produce direct translational movement without the need for mechanisms to convert rotary motion. Unlike conventional stepping motors, whose rotary motion is often converted into linear motion via external mechanical components such as ball screws, belts, pulleys, or lead screws, linear motors integrate this functionality within their own structure. This results in more compact, precise, and reliable solutions, especially suited for high-speed applications with relatively light moving masses. There are mainly two approaches to achieving linear motion in stepping systems: a mechanical solution, based on internal conversion, and a fully electromagnetic solution. In the first case, a conventional rotary motor is internally coupled to a worm gear or a lead screw. The rotor features an internal thread, while the shaft is replaced by a threaded screw which, prevented from rotating but free to translate, moves linearly in response to the rotation of the rotor. The amount of linear displacement per step depends on the motor's step angle and the mechanical pitch of the screw.

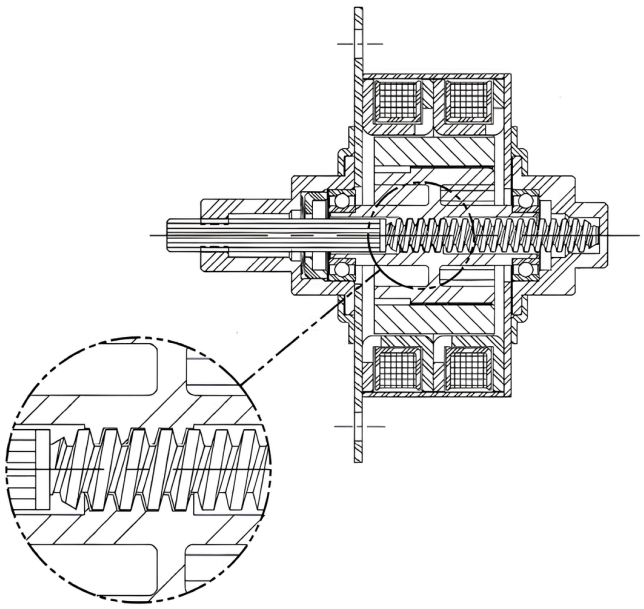


Figure 2.4.3a: Section of a linear motor belonging to the first type. [11]

In the second approach, typical of true linear stepper motors, the operating principle mirrors that of rotary stepper motors but in an “unrolled” configuration. In this case, the moving part of the motor, called the forcer, moves along a guide or toothed roller that acts as the linear stator. The forcer is generally composed of a permanent magnet and toothed electromagnets whose geometry is designed to maximize the concentration and direction of the magnetic flux. The arrangement of the teeth on the forcer’s phases is offset relative to the teeth on the roller, and each cycle of electrical excitation produces a movement corresponding to the magnetic alignment between opposing teeth.

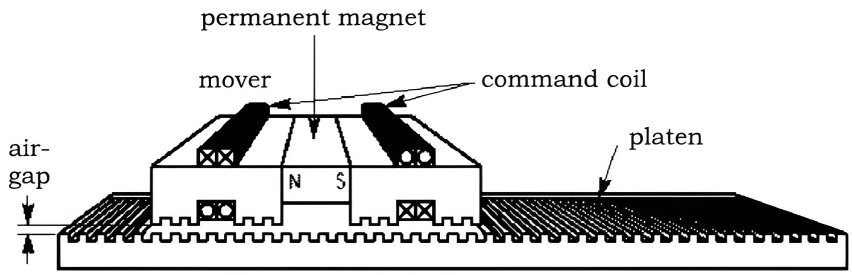


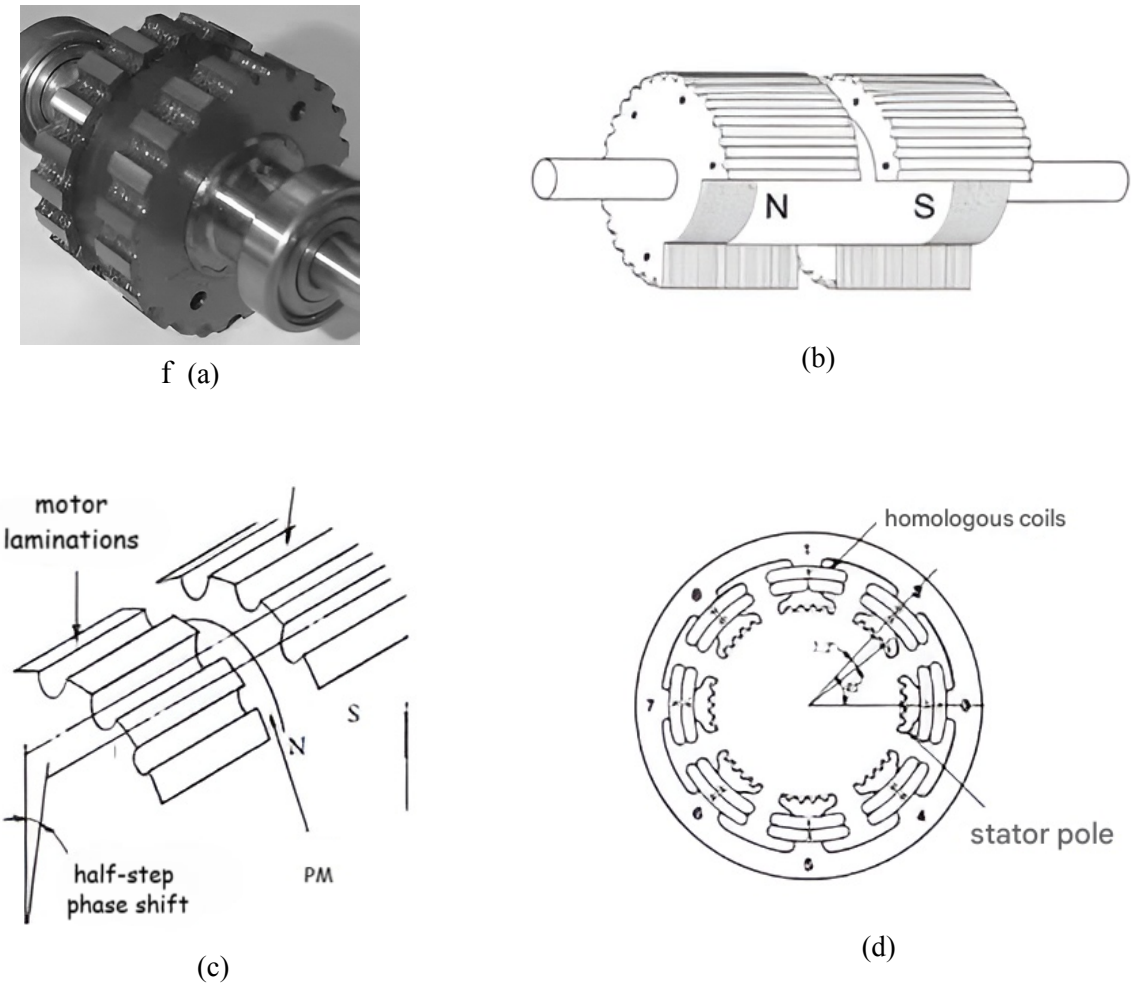
Figure 2.4.3b: Section of a linear motor belonging to the second type. [12]

The force generated by the magnetic field between the forcer and the roller enables the mobile component to advance by one step, and the repetition of the excitation sequence results in continuous linear displacement. Each complete sequence (for example, of four steps) allows a net advancement equal to the pitch of the roller’s teeth. When at rest, the motor maintains its position thanks to a holding force, derived from the magnetic attraction between the two components, which can be significantly higher than the typical detent torque found in other stepper motors. This force is useful to ensure positional stability without continuous energy consumption. Thanks to their simple construction, precise positioning, and the ability to provide constant forces regardless of travel length, linear stepper motors are widely used in numerous industrial and scientific precision applications.

2.4.4 Hybrid Motors

Hybrid stepper motors are designed to improve performance compared to reluctance stepper motors, particularly in terms of torque density. These motors combine two torque-generation principles: those of reluctance systems and electrodynamic systems. The hybrid motor structure is complex and includes both anisotropic stator and rotor, with an axial flux permanent magnet located in the rotor. The first patent for these motors was filed by Feiertag and Donahoo in 1952, and the motor was described as a synchronous induction motor for low-speed applications.

The internal structure of a typical four-phase hybrid stepper motor, with 50 teeth on the rotor (**Figure 3.4.5** ***a***), consists of a rotor with a cylindrical permanent magnet core that produces a unipolar magnetic flux (**Figure 3.4.5** ***b***). Each pole of the permanent magnet features a toothed structure typical of a reluctance motor, but with a half-step offset between the teeth of the two sections (**Figure 3.4.5** ***c and d***).



f (a) (b) (c) (d)

Figure 2.4.4: Various sections of a 4-phase, 50-tooth hybrid stepper motor. [6]

The stator structure in hybrid stepper motors is similar to that of reluctance motors, but with a key difference: each stator tooth is associated with two homologous phases instead of just one. The windings of these two phases are typically bifilar, meaning they are wound in opposite directions, so when energized with the same current, they generate opposite magnetic polarities. In the described hybrid stepper motor, there are four phases, each consisting of a series of coils. Each stator tooth is linked to two coils belonging to two different phases. The phase excitation pattern is designed so that the stator poles produce an alternating magnetic field, causing the rotor to move step by step. The torque generated in this motor type exploits the principle of

reluctance, reinforced by the magnetic field produced by the permanent magnet in the rotor. The operation of the hybrid stepper motor can be understood by observing the rotor's behavior in response to phase energization. When a phase is energized, the magnetic field produced by the stator poles attracts the rotor, aligning it with the field. Thanks to the half-step offset between the rotor sections, the motor advances one step at a time, generating torque that enables controlled and precise movement.

Some hybrid motor configurations allow operation in either bipolar or unipolar mode. In bipolar mode, phases are alternately energized with positive and negative currents to create the necessary magnetic field. In unipolar mode, the coils are arranged so the motor can be driven by a single current polarity, simplifying control but reducing the available power. Overall, hybrid stepper motors effectively combine the principles of variable reluctance and electromagnetic force from permanent magnets, delivering high torque and precision, making them particularly suitable for applications requiring accurate motion control.

2.5 Economic Considerations

In the context of industrial applications, economic considerations regarding stepper motors play a fundamental role. Although these motors offer precise and reliable control, they are not exempt from costs, both direct and indirect, that must be carefully evaluated during the design and implementation of an actuation system. The cost of a stepper motor actuation system depends on several factors, including the type of motor, the control system, the complexity of the positioning system, and the duration of the application. Understanding these costs is essential to making informed choices that balance performance and economic sustainability.

The initial cost of a stepper motor and its associated components, such as the controller, power supply, and feedback system, can be relatively low compared to other types of actuators, such as DC motors or induction motors. This aspect makes stepper motors a popular choice in precision positioning applications where quality position control is essential but the budget is limited. However, although the initial cost may be competitive, long-term operational and maintenance costs must also be considered. In particular, energy consumption is one of the main cost factors for systems using stepper motors. While these motors can be highly efficient in position control, they tend to consume energy continuously, especially when operating under nominal or intermediate load. This can lead to significant operating costs, particularly in applications where the motor runs for extended periods. To optimize energy costs, advanced control techniques can be implemented, such as phase current management or algorithms that reduce consumption during idle periods or when the load is reduced. Another factor affecting the overall cost of an actuation system is the complexity of system design and integration. Stepper motors are generally easy to implement in basic applications, but when advanced performance is required,

such as high-speed control or handling complex loads, the system design becomes more costly. Advanced control systems, which include feedback and compensation for step errors or load variations, add complexity to the project and thus increase costs. These additional costs must be justified by the need for superior performance, so that the overall investment in the system is economically viable relative to the benefits obtained.

Finally, the lifecycle of the actuation system must be taken into account. Although stepper motors are known for their reliability and long life, components such as bearings, windings, and electronic switching systems may require periodic maintenance, especially in intensive-use applications. Maintenance and component replacement costs must be included in the overall economic analysis to provide a clear view of the total cost of ownership of an actuation system.

In summary, although stepper motors offer initial economic advantages due to their low cost and ease of implementation, long-term economic considerations must include energy consumption, maintenance costs, control system complexity, and overall system durability. A thorough evaluation of these aspects is essential to optimize economic efficiency and ensure the sustainability of the actuation system over time.

Chapter 3

Circuit Design and Electrical Schematic

The main objective of this application is the design of a control board for stepper motors, capable of precisely and flexibly managing the acceleration profile through the combined use of a microcontroller and an FPGA. The microcontroller is responsible for providing an external communication interface, receiving commands, and processing the initial parameters; the FPGA, on the other hand, handles the generation of motor control signals and the real-time management of acceleration profiles, leveraging its high parallel processing capability and low latency.

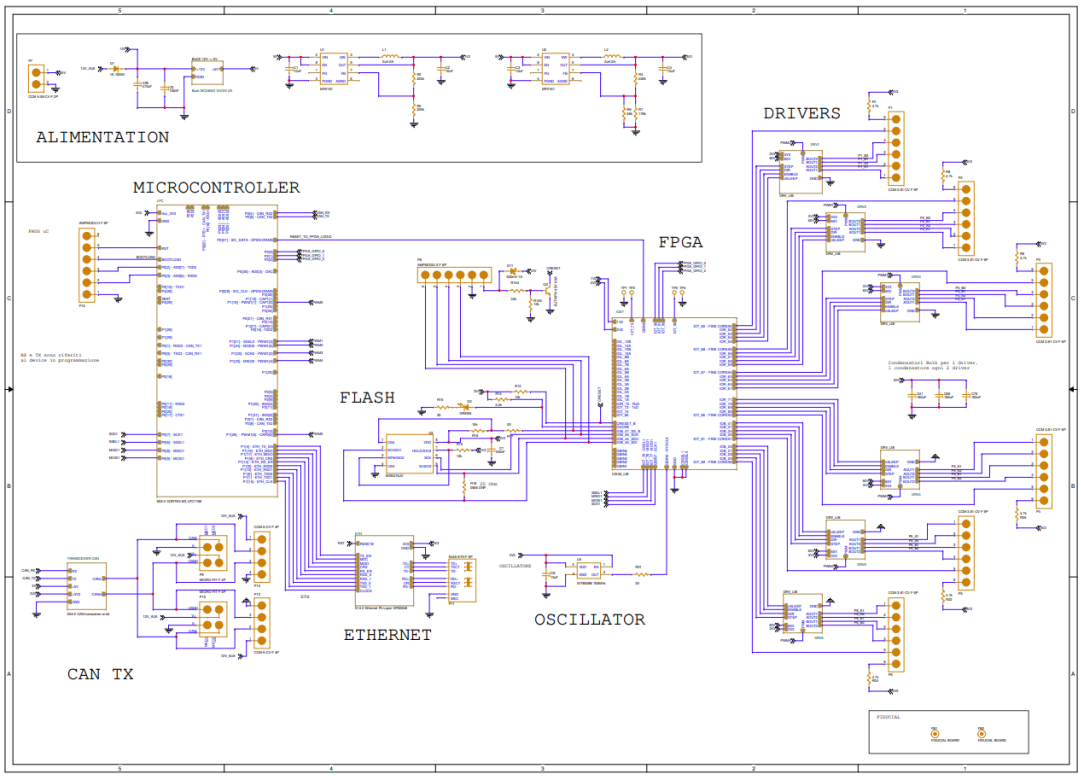


Figure 3.1: Complete electrical schematic of the stepper motor control board.

Figure 3.1 shows the electrical schematic of the system, highlighting the main components used and the functional connections between them. The schematic represents the hardware architecture underlying the project, including both active and passive elements, as well as power supply and communication lines.
The following sections provide a detailed analysis of the individual components and the role each one plays within the overall system, highlighting the design choices made and the technical rationale behind the implementation.

3.1 Microcontroller

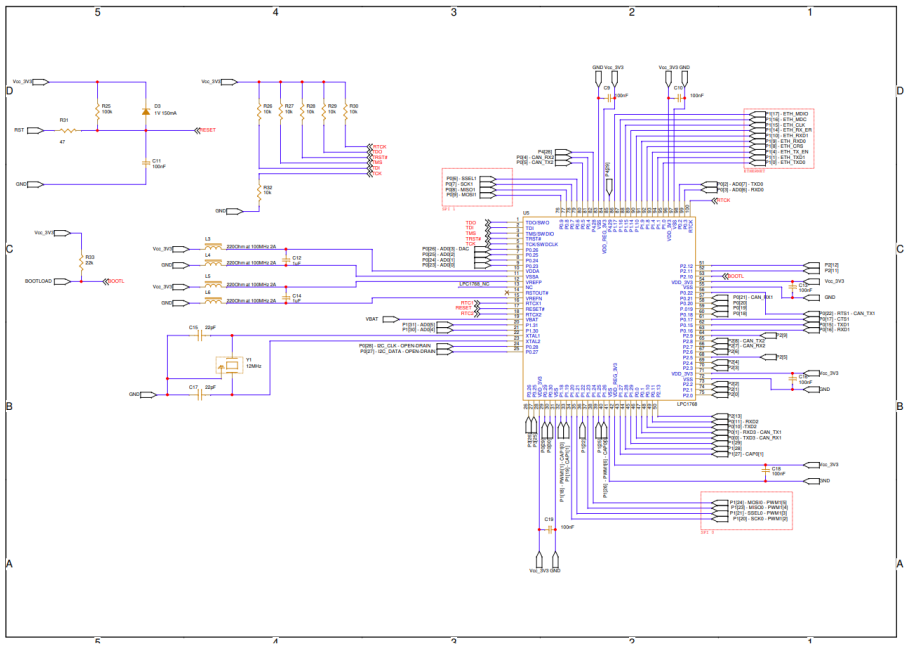


Figure 3.1.1 : Internal architecture of the Microcontroller.

A microcontroller is a type of microprocessor specifically designed to perform control, timing, and supervision tasks for devices, apparatuses, or industrial processes. These devices are crucial for the automation and efficient management of various systems, thanks to their ability to execute precise and repetitive operations. The widespread use of microcontrollers is especially prominent in systems that integrate control devices, which manage the overall operation of the system. Such systems, known as *embedded control systems*, represent a category of applications where microcontrollers are embedded directly within devices to optimize their functioning and ensure efficient and automated management. Their versatility and reliability make them indispensable across multiple sectors, from consumer electronics to the automotive industry, medical devices, and industrial automation.

A microcontroller is characterized by the onboard availability of several essential components that determine its functionality and versatility. Among these components are:

- **Central Processing Unit (CPU):** The core of the microcontroller is responsible for executing instructions and managing logical and arithmetic operations through an Arithmetic Logic Unit (ALU). It also includes registers to store CPU data, a status register (SR) to monitor the state of operations, and additional temporary registers. The CPU contains an instruction decoder and other control logic to manage its operations, handle resets, interrupts, and more.

- **Memory:** It includes various types of memory, which can be either volatile, meaning they lose their content when power is turned off, or non-volatile. Volatile memory, such as RAM (Random Access Memory), is used to quickly store and transfer temporary data between the CPU and the microcontroller's programs. On the other hand, non-volatile memory retains data even after power is removed. Examples include ROM (Read-Only Memory), which is static and cannot be altered during normal operation, and flash memory, which can be electronically programmed and erased, making it suitable for storing the program code.

- **Input/Output (I/O) Peripherals:** They allow the microcontroller to interact with the external environment by connecting to sensors, actuators, and other devices. These peripherals are essential for data acquisition and control of various external components. I/O peripherals may include serial ports, timers, counters, and analog-to-digital converters (ADC), enabling the microcontroller to communicate and handle complex operations in real time.

- **Communication Bus Management Modules:** They enable the management of serial communication protocols such as I²C and SPI, which facilitate connection and data exchange with other electronic devices. These modules are essential for transmitting data between the microcontroller and peripheral components, ensuring fast and efficient communication. For industrial applications, fieldbus modules can also be used, allowing communication in complex industrial environments and the management of networks composed of sensors and actuators.

- **Clock:** To keep pace with the synchronized operation of the entire system, the clock signal can be generated internally, derived from a crystal oscillator, or sourced externally. The clock signal is essential to ensure that all microcontroller operations are carried out in a coordinated and timely manner. Crystal oscillators are especially valued for their pre-

cision and stability, whereas external sources may be employed in applications requiring synchronization with other devices or systems.

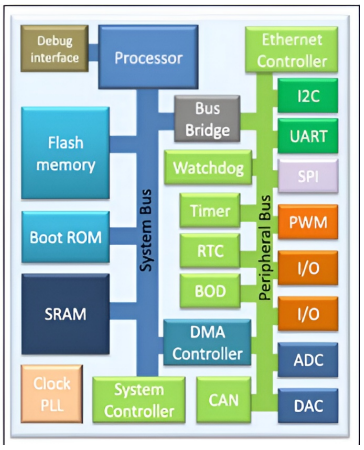


Figure 3.1.2: Typical block diagram of a microcontroller. [18]

The microcontroller used in this project is the **LPC1768**, based on the Cortex-M3 architecture. It is a high-performance 32-bit processor designed specifically for the microcontroller market. It offers excellent processing capabilities combined with fast interrupt handling. Additionally, it features advanced system debugging functionalities with extensive breakpoint and trace capabilities. The processor core, system, and memory architecture are optimized for efficiency, while power consumption is minimized thanks to its integrated sleep mode and an optional deep sleep mode.

This processor is ideal for applications that require intensive computation and precise interrupt handling, an essential requirement for time, critical systems. Its debug and trace functionalities, such as Serial Wire Debug and Serial Wire Trace, reduce the number of pins needed for debugging, tracing, and code profiling.

Moreover, energy efficiency is enhanced through its power-saving modes, which reduce consumption when the processor is idle. It also features an optional Memory Protection Unit (MPU) that enables control over individual memory regions, allowing applications to implement multiple privilege levels. This helps to isolate and protect code, data, and stack on a per-task basis. Such capabilities are increasingly critical in embedded applications, particularly in the automotive industry.

The Cortex-M3 processor supports system interrupts and exceptions, which alter the normal control flow of the software. These are managed, except for the reset, by the Handler mode. During an exception, the processor state is automatically saved onto the stack and restored upon completion of the interrupt service routine.

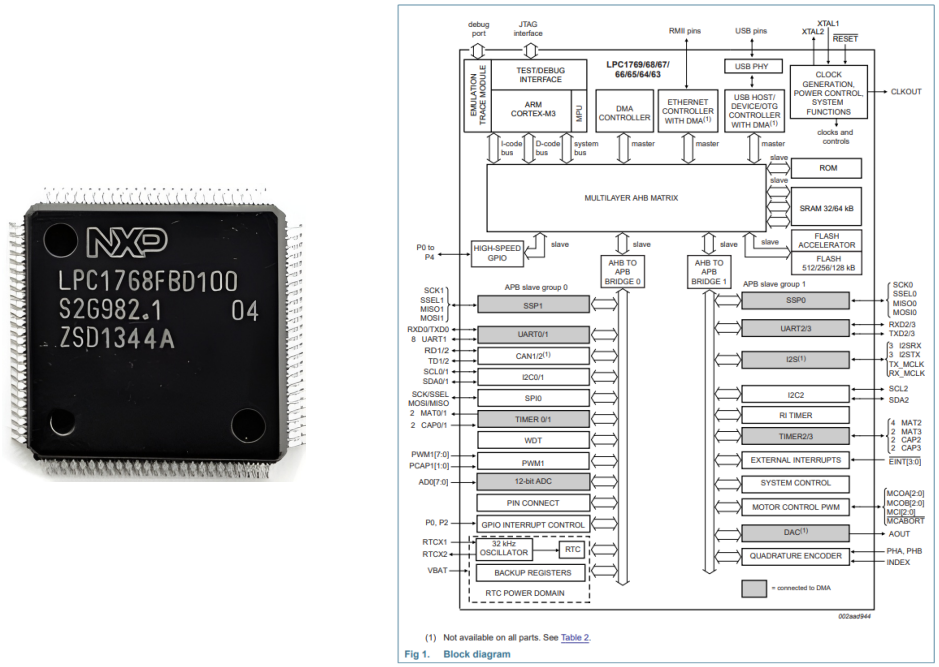


Figure 3.1.3: LPC1768 microprocessor (left) and his internal block diagram (right). [18]

The LPC1768 microcontroller operates at frequencies of up to 100 MHz and is based on a Harvard architecture with a three-stage pipeline. It features separate buses for instructions, data, and peripherals, as well as a prefetch unit with support for speculative branching, which enhances overall system performance. The device integrates 512 kB of Flash memory and up to 64 kB of SRAM, divided into multiple blocks to optimize access by the processor and peripherals, especially those utilizing Direct Memory Access (DMA). Its 8-channel DMA controller allows for fast and efficient data transfers between memory and peripherals, thereby reducing the CPU workload.

From a connectivity standpoint, the LPC1768 is highly versatile: it includes 4 UARTs, 2 CAN interfaces, 2 SSP controllers, 1 SPI interface, 3 I²C interfaces, and one I²S interface for digital audio data transmission. It also provides an Ethernet MAC interface with RMII support and a USB 2.0 Host/Device/OTG port, both equipped with dedicated DMA controllers and integrated PHY.

On the analog side, the microcontroller features a 12-bit Analog-to-Digital Converter (ADC) with eight multiplexed input channels and a conversion rate of up to 200 kHz, as well as a 10-bit Digital-to-Analog Converter (DAC), suitable for signal generation applications. For timing and PWM signal management, the device includes four general-purpose timers, a three-phase motor control PWM module, a six-output general-purpose PWM, and quadrature encoder interfaces. The integrated Real-Time Clock (RTC) has its own power domain and can keep track of time even when the main power supply is off, thanks to a dedicated backup battery input.

Power control is managed by the Power Management Unit (PMU), which supports various low-

power modes: Sleep, Deep-sleep, Power-down, and Deep Power-down. Each peripheral can be assigned a separate clock, allowing for fine-grained power consumption management. The device operates on a single 3.3 V power supply, compatible with a range from 2.4 V to 3.6 V. Debugging is facilitated through JTAG and Serial Wire Debug interfaces, with support for real-time tracing via the Emulation Trace Module. Finally, the LPC1768 is available in several package options, including LQFP100 and TFBGA100, with up to 70 configurable GPIO pins that can be assigned to a variety of digital and analog functions.

3.2 FPGA

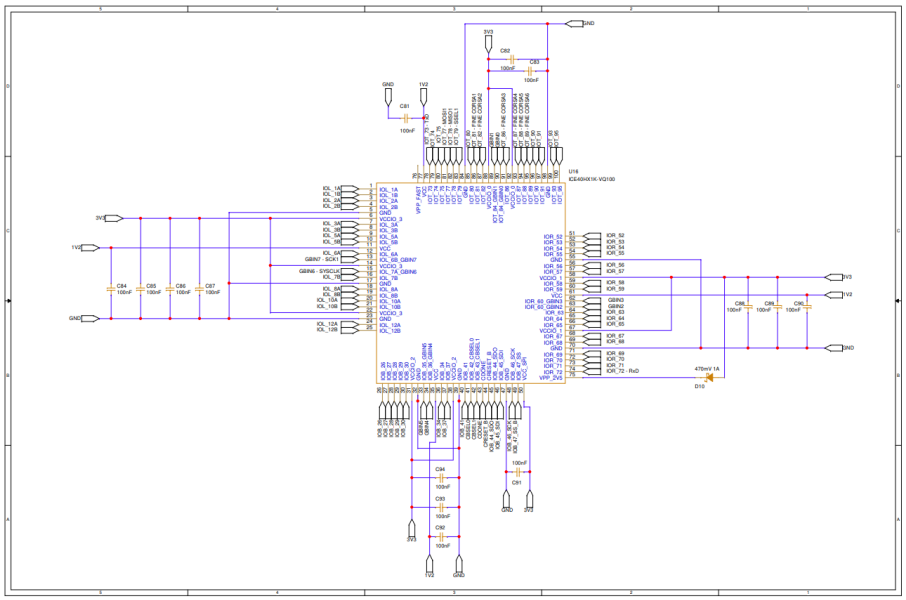


Figure 3.2.1: Internal architecture of the FPGA.

FPGAs (Field Programmable Gate Arrays) represent one of the most versatile and powerful technologies in the field of digital electronics and programmable hardware system design. Unlike traditional integrated circuits such as ASICs (Application-Specific Integrated Circuits), which are designed to perform a fixed and unchangeable function, FPGAs are programmable devices that allow the definition and modification of hardware behavior even after manufacturing. This feature makes them ideal for rapid prototyping, the development of complex embedded systems, and applications that demand high flexibility.

An FPGA is composed of an array of Configurable Logic Blocks (CLBs), interconnected by a programmable routing network, along with I/O blocks for external communication. Each logic block contains fundamental components such as Look-Up Tables (LUTs), flip-flops, and multiplexers, which can be combined to implement any desired logic function. Thanks to this

architecture, both simple combinational circuits and complex digital architectures, such as processors, controllers, and parallel processing systems, can be constructed.
FPGAs are also distinguished by the underlying technology used in their construction, which affects performance, power consumption, cost, and programming methods. The main technologies include:

- **Antifuse:** This technology uses simple logic modules interconnected by antifuses, devices that are normally open and become permanently conductive when exposed to a specific programming current, thus creating stable, non-volatile connections even without power. Antifuse-based devices are characterized by low power dissipation, instant startup times, and reduced propagation delays. However, they are not reprogrammable, offer limited logic capacity, and must be programmed prior to PCB assembly.

- **Flash:** Flash-based FPGAs use floating-gate transistors to implement programmable interconnections. These transistors can be reprogrammed, although they support a limited number of erase/write cycles (typically a few hundred). Flash technology enables non-volatile devices with good logic capacity and the ability to update configurations, but it requires specific programming voltages.

- **SRAM:** The most common FPGA technology today is based on static RAM (SRAM) cells to implement both the Look-Up Tables (LUTs) and the programmable interconnect networks. SRAM-based FPGAs are highly flexible and freely reprogrammable, allowing for rapid development and iterative project updates. However, being volatile devices, they require configuration data to be loaded from external memory at every power-up, which results in longer startup times and the need for additional support circuitry.

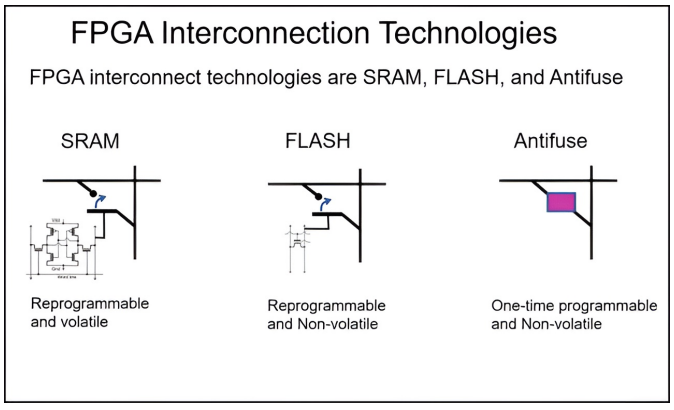


Figure 3.2.2: FPGA interconnection technologies. [20]

The operation of an FPGA is based on programming the matrix of logic blocks and the interconnection lines to implement the desired digital circuit. The Look-Up Tables (LUTs), for example, are memories that directly implement the truth table of a logic function, enabling the execution of any combinational operation on n variables. Flip-flops, on the other hand, allow data storage and the implementation of sequential logic. The ability to reconfigure connections between logic blocks enables the construction of modular and scalable architectures.
In addition to the Configurable Logic Blocks (CLBs), FPGAs include programmable I/O blocks that manage communication between the device and external components such as sensors, motors, microcontrollers, and networks. These blocks support high-speed communication protocols and standards, making FPGAs suitable for complex systems with demanding throughput and synchronization requirements.

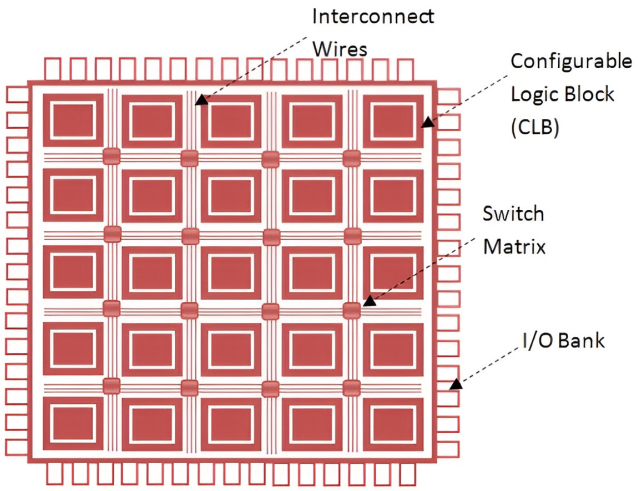


Figure 3.2.3: FPGA architecture diagram. [21]

The adoption of FPGAs offers a highly flexible solution for the implementation of advanced digital systems. One of the distinctive features of these devices is the ability to configure and reconfigure them even after integration into the final system. This allows for updates, modifications, or adaptation of the design to new operational requirements without physically altering the hardware. Such an approach is particularly useful during prototyping, testing phases, and in applications that evolve over time.
Another advantage lies in their capacity to perform parallel operations. Unlike microcontrollers and microprocessors, which operate sequentially, FPGAs can manage multiple processes simultaneously by fully exploiting their hardware architecture. This property makes them especially suitable for tasks requiring speed and efficiency, such as signal processing, real-time control, and the management of stepper motors with dynamic profiles. Compared to ASICs (Application-Specific Integrated Circuits), these technologies significantly reduce development

time. The absence of complex manufacturing steps allows the system to be designed and validated in shorter timeframes, facilitating the rapid introduction of new solutions. Although the cost per device may be higher than simpler solutions, the trade-off between performance, logic capacity, and adaptability is advantageous, especially in low- and medium-volume production runs where investment in custom circuits is not justifiable.

Finally, FPGAs stand out for their wide adaptability across different application domains, ranging from telecommunications to automotive, industrial sectors, and research. In each of these fields, they enable the implementation of sophisticated algorithms, specific protocols, and highly customized control systems.

Despite these benefits, using FPGAs requires attention to several factors. Power consumption is generally higher than that of equivalent microcontrollers, and the complexity of the hardware design and interface management can increase development time and costs. Therefore, it is important to carefully evaluate the technological choice based on the specific needs of the project. For this reason, the choice of an FPGA should be guided by the necessity for flexibility, parallelism, and dedicated hardware processing capability. In stepper motor control systems, for example, FPGAs efficiently implement complex and customized acceleration profiles, while also managing real-time feedback signals and multiple communication interfaces.

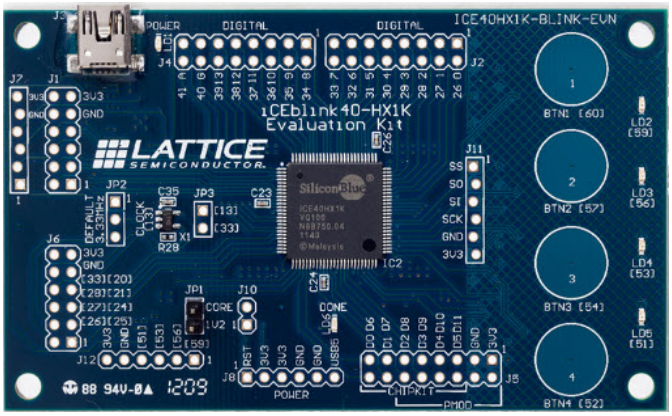


Figure 3.2.4: iCE40 evaluation board. [19]

The device used for this project is the **iCE40HX1K-VQ100** produced by Lattice Semiconductor, particularly suited for embedded applications requiring low power consumption and a compact footprint. This device is based on low-power CMOS technology and utilizes a volatile SRAM configuration memory, which allows the FPGA to be reprogrammed by loading a configuration file at power-up from an external memory, such as an SPI Flash.

From a hardware perspective, the iCE40HX1K integrates approximately 1,280 configurable logic blocks, including LUTs and flip-flops, enabling the implementation of both combinational

and sequential logic functions with great flexibility. These elements, together with internal memory blocks usable as buffers, make the FPGA particularly suitable for managing real-time data streams and control tasks, such as those required for driving stepper motors. The device also features a wide range of programmable Input/Output pins with multi-voltage capability, facilitating direct interfacing with external components such as microcontrollers, motor drivers, and sensors. The presence of an integrated PLL allows the generation of stable and configurable clocks, ensuring precise synchronization of digital operations.

The 100-pin VQFP package offers a compact solution well-suited for PCBs with limited space while still providing a sufficient number of pins for the project's requirements. These features make the iCE40HX1K-VQ100 particularly well suited for the stepper motor control project developed in this thesis. In particular, its low power consumption, reprogrammability, and flexibility in managing logic resources allow for the creation of an efficient and easily updatable control system.

The specific advantages of the iCE40HX1K-VQ100 for stepper motor control include:

- **Low power consumption,** essential for embedded applications and low-power systems.
- **Programming flexibility,** allowing easy adaptation of the control algorithm to different motors or requirements.
- **Parallel processing capability** due to the hardware nature of the FPGA, which allows simultaneous control of multiple motors without slowdowns.
- **Support for high-precision clocks** via the integrated PLL, necessary to ensure accurate timing synchronization during acceleration and deceleration phases.
- **A large number of configurable I/O pins,** useful for direct interface with motor drivers, position sensors or limit switches, and other peripheral components.
- **Compact package size,** which supports a compact design of the control board.

Finally, the availability of a comprehensive and accessible development environment, such as Lattice Diamond or open-source solutions, facilitates writing, debugging, and optimizing the FPGA firmware, thereby reducing the overall project development time.

3.3 Drivers

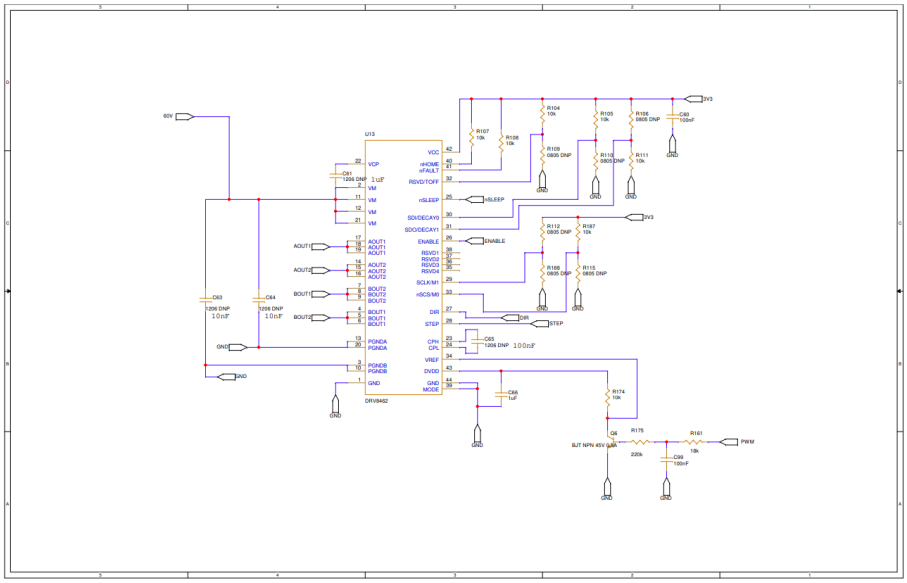


Figure 3.3.1: Internal architecture of a driver.

Motor drivers are a fundamental component in electric motor control systems, especially when dealing with stepper motors, which require precise management of current and coil activation sequences. Their primary function is to act as an interface between the control system, such as a microcontroller, digital processor, or FPGA, and the motor itself. The driver converts low-level digital commands, typically pulse signals or low-power control signals, into high-power electrical signals that drive the motor coils, ensuring the correct current and voltage for the desired motion.

Specifically, drivers are responsible for regulating the current in each motor phase to control torque, speed, and position with high precision. This control is crucial to prevent problems such as overheating, stalling, or missing steps, which would degrade the performance and safety of the system. Additionally, drivers often include built-in protections against overcurrent, overtemperature, short circuits, and other fault conditions, safeguarding both the motor and the control electronics. These devices support various driving modes, ranging from simple step/direction operation, where the driver executes one step per received pulse, to more advanced methods like microstepping, which subdivides each step into multiple microsteps for smoother and more precise motion.

From an implementation standpoint, modern drivers interface with control systems through digital buses or discrete signals and offer programmable configurations to adapt to different motor types and operating conditions. Using drivers with digital interfaces and integrated functions

significantly reduces the computational load on the microcontroller or FPGA, offloading complex tasks such as current modulation and acceleration management, which would otherwise require substantial processing resources.

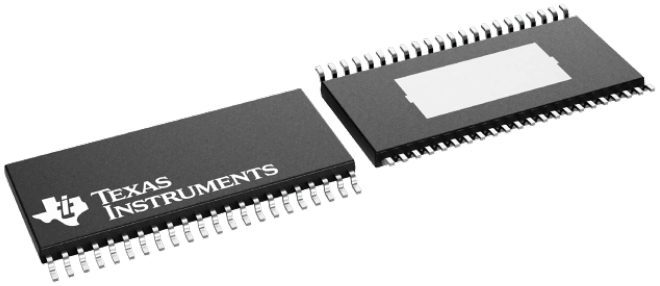


Figure 3.3.2: DRV8462 with top and bottom thermal pad. [2]

An example of an advanced stepper motor driver is the **DRV8462** produced by Texas Instruments. This device integrates full motor control, offering a simple yet powerful interface to manage motion. The DRV8462 supports up to 6 control phases and allows various levels of microstepping, up to 1/256 of a full step, significantly enhancing motion precision and smoothness. This is especially useful in robotics and industrial automation applications, where high positioning resolution is required to minimize vibration and unwanted noise.

The driver autonomously manages current modulation in the motor windings, ensuring constant torque output and dynamically adapting to load variations. It also includes advanced protection features, such as on-chip temperature monitoring, overcurrent detection, and short-circuit protection, ensuring safe operation even under demanding conditions.

In terms of integration, the DRV8462 supports both step/direction control and more complex digital interfaces, making it easy to interface with microcontrollers or FPGAs. Its programmable internal registers allow flexible configuration of operating parameters, making the device suitable for a wide range of applications, from simple stepper motor actuation to multi-axis advanced control systems.

Moreover, the design is optimized to minimize power losses and improve the overall efficiency of the drive system, an essential aspect in industrial applications where power consumption and thermal dissipation must be kept under control. Its compact package and the availability of modules with different pin configurations also facilitate integration into space-constrained projects.

3.4 Auxiliary Components

3.4.1 CAN bus

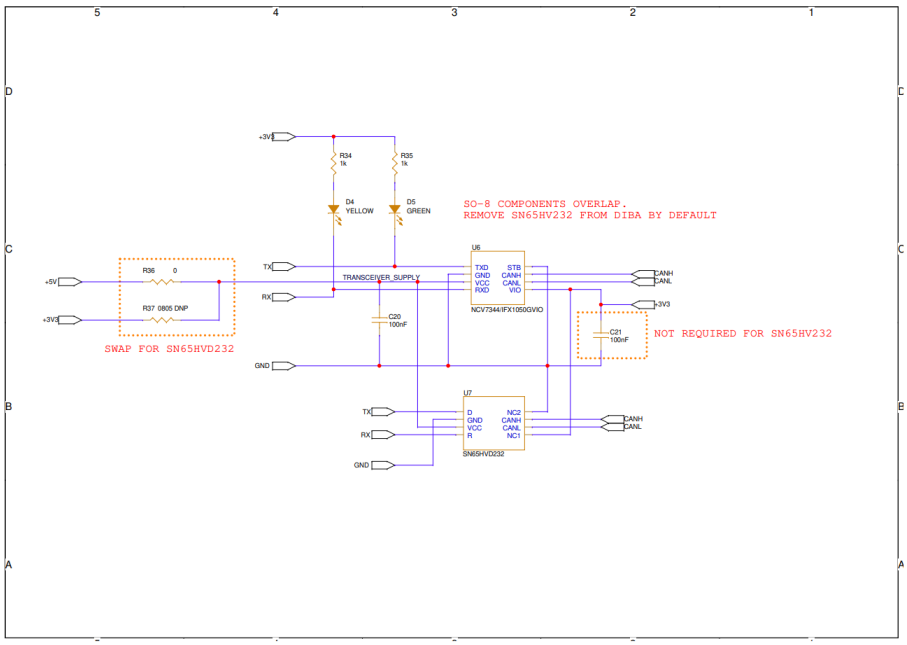


Figure 3.4.1: Internal architecture of CAN bus.

The CAN bus (Controller Area Network) is a standardized asynchronous serial communication protocol developed for reliable, real-time data transmission in industrial and automotive environments. Thanks to its high noise immunity, error detection and correction mechanisms, and the ability to manage message priority through non-destructive arbitration, CAN remains widely used in embedded and industrial systems.

In this project, the CAN bus is used as a communication channel for the microcontroller, enabling the exchange of system commands, diagnostic data, and status information with other devices. This choice ensures robust and deterministic local communication, ideal for system control and supervision.

At the same time, the system also integrates an Ethernet interface, used for connection to larger networks, remote access, and management through advanced protocols. The combination of CAN and Ethernet thus allows leveraging the benefits of real-time, reliable communication offered by CAN at the local level, alongside the versatility and network integration capabilities provided by Ethernet. This approach is particularly effective in meeting both the operational and management requirements of the system.

The transmission speed of the CAN bus depends on the length of the connection, as shown in **Table 3.1**, which lists typical values of 1 Mbit/s for lengths up to 40 meters, with progressively lower speeds for longer distances.

Table 3.1: CAN bus speed as a function of link length.

Bus length [m]	Transmission speed [Mbit/s]
40	1.00
100	0.50
200	0.25
500	0.125
6000	0.010

3.4.2 Ethernet

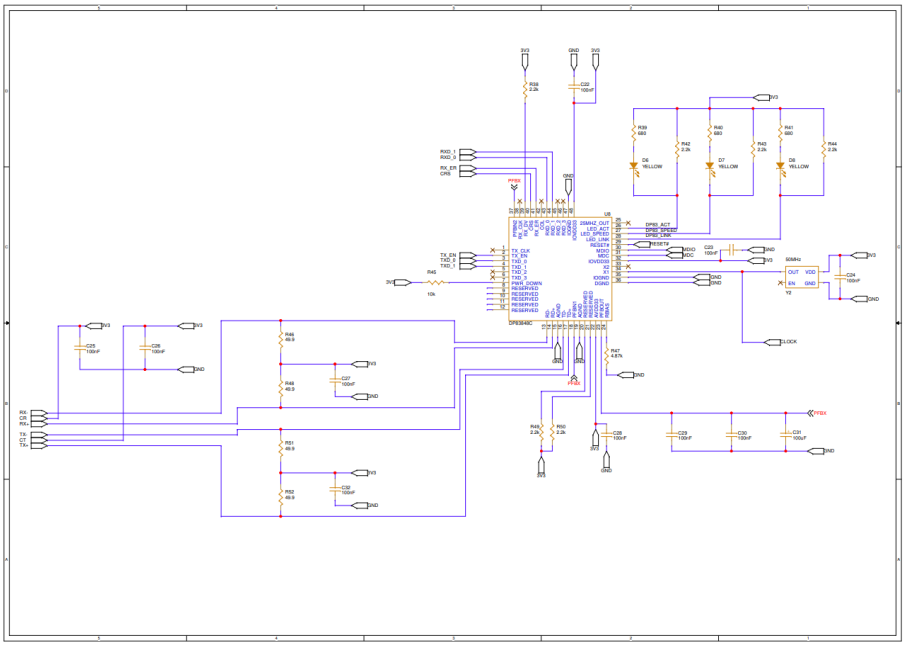


Figure 3.4.2: Internal architecture of Ethernet.

In this project, the Ethernet interface is used to provide the system with connection to larger local networks, enabling remote access, monitoring, and firmware updates from PCs or servers. Thanks to this standardized technology, the device can exchange data, commands, and diagnostic messages with external tools via TCP/IP protocols, offering a versatile control interface, for example accessible through a web browser.

From a hardware perspective, the Ethernet module consists of two main parts: the MAC (Media Access Control), integrated into the microcontroller, which manages the logic for data transmission and reception, and the PHY (Physical Layer), an external component that converts logical signals into electrical signals transmitted over the cable. In this project, the PHY is connected to the microcontroller via an RMII interface, chosen to optimize resources and ensure efficient communication.

One of Ethernet's key advantages is its high bandwidth, typically 10/100 Mbps in common microcontroller implementations, which allows managing large amounts of data with low latency. This feature is essential to ensure fast updates, real-time monitoring, and reliable communication with external systems.

Within the context of the project, Ethernet integrates with the CAN bus, providing an optimal combination: CAN handles local, robust, and deterministic communication between the microcontroller and internal devices, while Ethernet enables connectivity to wider networks and remote supervision services. Additionally, the microcontrollers used include dedicated buffers, DMA, and hardware support for efficient packet processing, minimizing CPU load and improving overall system stability.

3.4.3 Flash memory

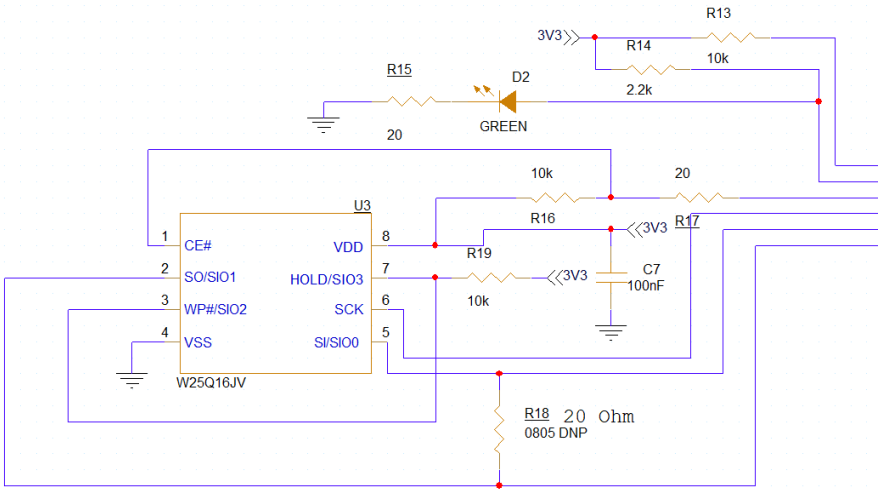


Figure 3.4.3: Architecture of Flash Memory.

In this project, flash memory was primarily chosen for its ability to permanently store configuration data and motion profiles necessary for the proper operation of the system. Unlike volatile memories, such as RAM, which lose their content when power is removed, flash ensures that

all essential settings remain saved even when the system is powered off. This feature is essential because the programmable device used lacks internal non-volatile memory capable of retaining parameters or state between power cycles. To guarantee a consistent and functional startup, the data required for initialization, such as acceleration, speed, and position profiles for the stepper motors, are saved in the external flash memory. During the power-up phase, this information is automatically read and transferred into the internal logic of the system, enabling a fast and reliable start with the configuration already ready for use. This approach avoids the need to manually re-enter parameters at every power-up, reducing both initialization times and the overall complexity of the architecture. Furthermore, the ability to update the flash contents through a simple programming procedure allows flexible management, adaptable to different operating conditions or subsequent project modifications.

3.4.4 Quartz crystall oscillator

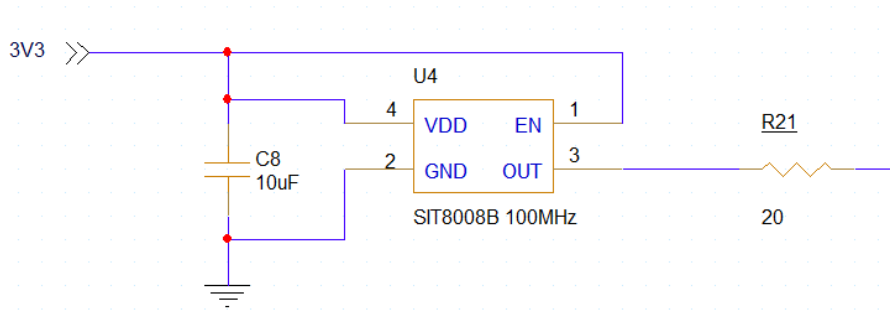
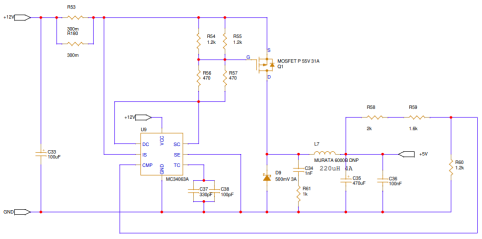


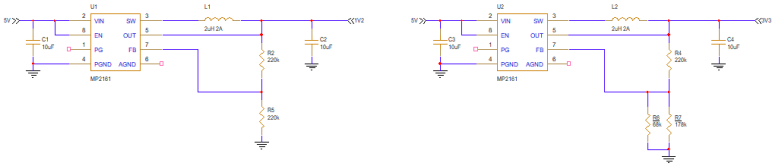
Figure 3.4.4: Architecture of quartz oscillator

In the project, a quartz crystal oscillator was used to provide the main clock signal to the FPGA. The quartz oscillator exploits the piezoelectric properties of the quartz crystal, which, when subjected to an electric voltage, vibrates at a very stable and precise frequency. This characteristic makes the quartz oscillator ideal for generating a reliable clock signal, essential for the correct operation of the digital logic implemented in the FPGA. Thanks to this precise clock source, the device can perform stepper motor control operations with strict timing, ensuring smooth and synchronized movements. Moreover, the stability of the oscillator minimizes timing errors in SPI communications between the microcontroller and the FPGA, thus improving the overall reliability of the system. Finally, the quartz oscillator maintains a constant frequency even in the presence of environmental variations, guaranteeing stable performance throughout all operational phases.

3.4.5 Voltage regulators



(a)



(b)

Figure 3.4.5: Architecture of (a) Buck regulator and (b) switching regulators.

To properly power the various components of the system, voltage regulators were used to convert a primary 12 V source into the lower voltages required for the operation of individual devices. Specifically, three voltage levels were generated: 5 V, 3.3 V, and 2.5 V. The 5 V supply is used, for example, to power communication modules, while the 3.3 V level is required by almost all the main devices, including drivers and microcontroller, which typically operate with low-voltage logic. The 2.5 V supply, instead, is necessary for some internal FPGA power rails, particularly for the logic core.

To obtain these voltages, a buck converter was used to generate the 5 V rail directly from the 12 V input. Then, two identical switching regulators were employed to derive the 3.3 V and 1.2 V outputs, by appropriately adjusting the feedback resistor divider in each circuit to set the desired output voltage. The use of switching voltage regulators allows for a stable and safe power distribution, protecting components from potential overvoltages, and ensuring the correct operation of the system. Furthermore, starting from a single input voltage simplifies the external power supply and enables a more organized and compact PCB architecture.

3.5 Connections

The connections between the various components of the board have been carefully designed to ensure high operational reliability, with redundancy implemented where necessary in critical areas. Special attention was also given to the layout and routing of PCB traces, with the aim of achieving a clean and well-organized schematic that minimizes interference and facilitates debugging and maintenance tasks.

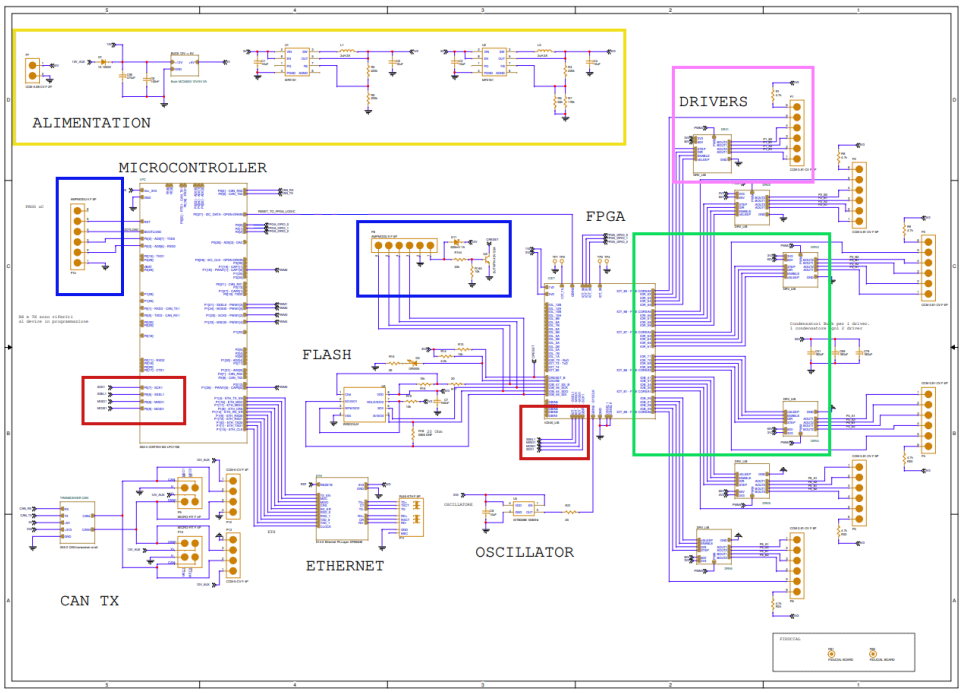


Figure 3.5.1: Main connections within the schematic: in red the connection between the microcontroller and the FPGA, in green between the FPGA and the drivers, in pink between the drivers and the motors, in yellow the power supplies, and in blue the programming and debugging connectors.

Microcontroller ↔ FPGA Connection

The communication between the microcontroller and the FPGA is primarily handled through a dedicated SPI interface, composed of the following signals:

- **SPI_MOSI** (Master Out Slave In): carries data from the microcontroller to the FPGA;
- **SPI_MISO** (Master In Slave Out): transfers data from the FPGA to the microcontroller;
- **SPI_CLK**: the clock signal provided by the microcontroller;

- **SPI_CS** (Chip Select): activates the FPGA during transmission.

This interface enables reliable and fast communication for sending control commands and data to the FPGA-based motor control system.

FPGA ↔ Stepper Driver Connection

The FPGA controls each of the six stepper motor drivers using two primary signals per motor:

- **STEP** for the step pulse,
- **DIR** for direction.

Additional signals such as **ENABLE**, if supported by the drivers, and **SLEEP** inputs for diagnostics can also be included. All signals are buffered to ensure sufficient drive current and resistance to noise. In addition, a limit switch signal is present for each motor, allowing precise detection of the maximum position reached. This prevents mechanical damage or undesired movements beyond the predefined limits.

Driver ↔ Motor Connection

The drivers are connected to the motors through specific screw terminals (CCM), commonly used for electrical connections on printed circuit boards. Each connector directly drives the A/B phases of a bipolar stepper motor.
In particular, the **CCM-3.81-CV** model used in this project features six inputs, employed for connecting the four phases of the stepper motor, a 3.3 V power supply line, and the limit switch signal. This compact and organized configuration allows all essential lines for the system's operation to be managed through a single connector, greatly simplifying wiring and maintenance tasks.

Power Supply Connections

The system power supply initially provides a voltage up to 60 V, which is subsequently reduced to 40 V during the operational phase to power the motors. Meanwhile, a 12 V source is converted to 5 V using a buck converter, from which voltages of 1.2 V and 3.3 V are obtained through dedicated switching regulators. These lower voltages are necessary for the proper operation of digital logic, drivers, and communication circuits.

Programming and Debug Connectors

Two **AMPmodu** connectors are used in the project to implement the programming interfaces for the microcontroller and the flash memory connected to FPGA. Thanks to their standard 2.54 mm pitch, they are compatible with most commercial programmers and adapters, simplifying development, debugging, and firmware or bitstream updating operations. Furthermore, their modular design allows for a stable yet easily removable connection, ideal for prototyping and frequent testing environments where rapid system reconfiguration is needed without resorting to soldering.

Chapter 4

PCB Layout and Board Design

The layout of the Printed Circuit Board (PCB) represents a critical phase in the design of complex electronic systems. Once the electrical schematic is finalized, it is essential to translate the functional logic into an optimal physical arrangement of components and traces to ensure reliable performance, signal integrity, and manufacturability.
In the specific case of the board developed to control six stepper motors, the layout had to take into account a range of electrical, thermal, and mechanical constraints. The presence of distinct subsystems necessitated a careful and functional placement of components on the board.
Specifically, several design measures were implemented to:

- **Logical and physical separation of power and signal circuits:**
  Power circuits, which handle high voltages and currents (such as motor drivers and voltage regulators), were separated both logically and physically from signal circuits (microcontroller, FPGA, etc.). This division is essential to reduce electromagnetic interference and preserve the quality and integrity of sensitive digital signals.

- **Optimization of power distribution:**
  Given the high current drawn by the motor drivers, power distribution was optimized by employing significantly wider and more robust copper traces. This approach reduces voltage drops, minimizes trace heating, and ensures a stable and reliable current supply to high-demand loads.

- **Minimization of critical trace lengths:**
  The length of critical traces was reduced particularly for modular blocks such as voltage regulators, which were compactly grouped with very short signal paths. Conversely, managing connections between the FPGA and the six motor drivers was more challenging, since the separation between power and signal circuits and the physical distribution of the drivers across the board required relatively long traces for step and direction signals.

To mitigate issues from these lengths, the routing alternates communication signals with different types of signal traces, minimizing crosstalk and electromagnetic interference.

- **Accessibility and ease of assembly, testing, and maintenance:**
  Connectors were positioned along the perimeter of the board, maintaining sufficient spacing to ensure easy access. This choice greatly facilitates wiring, assembly, testing, and maintenance operations, improving the ergonomics and usability of the board throughout the project lifecycle.

4.1 Design Tools

The software eployed for the PCB design was OrCAD PCB Designer, a professional software by Cadence, widely recognized for its effectiveness in developing complex and highly reliable projects. This integrated environment enables managing the entire design process, from the electrical schematic to layout generation, ensuring netlist consistency and an efficient, reliable workflow.

OrCAD provides automatic design rule checks, both electrical and geometric, allowing timely detection of critical errors for manufacturing and operation. Multilayer management was used to organize digital signals, power supplies, grounds, and reference planes in separate layers, enhancing signal integrity and reducing electromagnetic noise.

The extensive component library, combined with the ability to import custom footprints, enabled adapting the design to non-standard components and specific mechanical requirements. Routing was facilitated by both the assisted autorouter, which automates trace placement, and powerful manual routing tools, essential for critical traces such as differential pairs or high-current lines. The strong integration between schematic and layout allowed for an iterative and dynamic workflow, enabling quick identification and correction of connection errors or footprint mismatches, keeping schematic and layout always synchronized.

Finally, OrCAD enabled the export of all the files necessary for production and assembly, including Gerber files, detailed BOM (Bills of materials), and Pick & Place files for automatic SMT assembly. Thanks to these features, the design was precise, efficient, and compliant with the highest standards of quality and reliability.

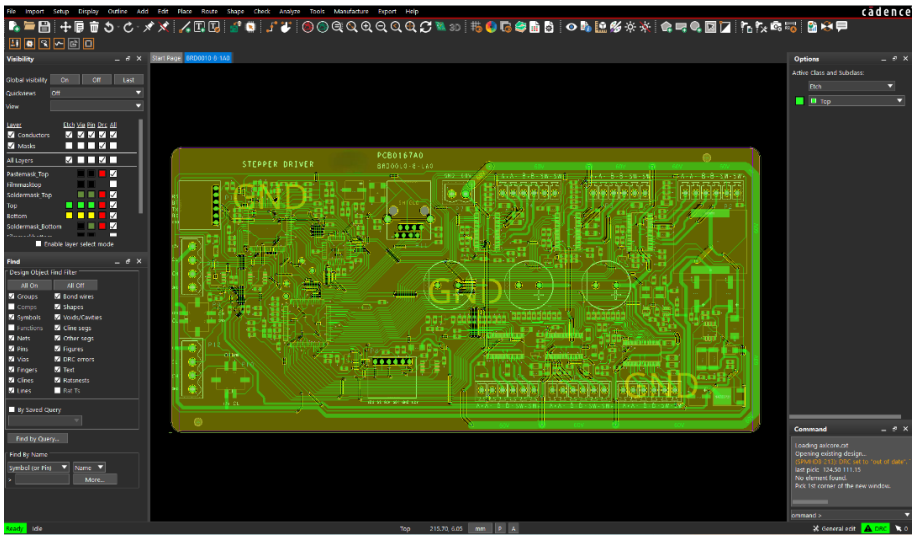


Figure 4.1.1 : PCB layout overview of the developed motor control board designed in OrCAD PCB Designer.

4.2 Component Organization

The arrangement of functional blocks was carefully designed to optimize trace lengths and facilitate wiring, as well as to improve the thermal management of the board.

The FPGA is positioned at a central height, slightly shifted towards the left side, to reduce the length of communication traces both to the microcontroller and to the motor drivers, thereby improving digital signal integrity and minimizing interference.

The microcontroller and Ethernet module are located near the FPGA, creating a compact functional area that enables direct and fast interfacing between devices. This proximity helps to reduce delays and noise in control signals, which is essential for efficient and synchronized operation.

The motor drivers are arranged along the right edge of the board, close to the output connectors leading to the external motors. This placement facilitates neat wiring and better thermal dissipation by leveraging direct contact with the external environment and heat sinks mounted on each driver. The voltage regulators are placed in the same area to effectively manage the heat generated.

To stabilize the power supply for the drivers, the original design included three high-capacity bulk capacitors, each capable of supporting two drivers simultaneously. This choice, preferred over using one capacitor per driver as suggested by datasheets, optimized board space without compromising power distribution quality. However, during production, due to the unavailability of the specified capacitors in stock, seven lower-capacity capacitors connected in parallel were used instead, maintaining the desired electrical characteristics.

Connectors are placed along the edges of the board to facilitate external wiring and ensure easy access during assembly, testing, and maintenance, thus contributing to an orderly and clear layout.

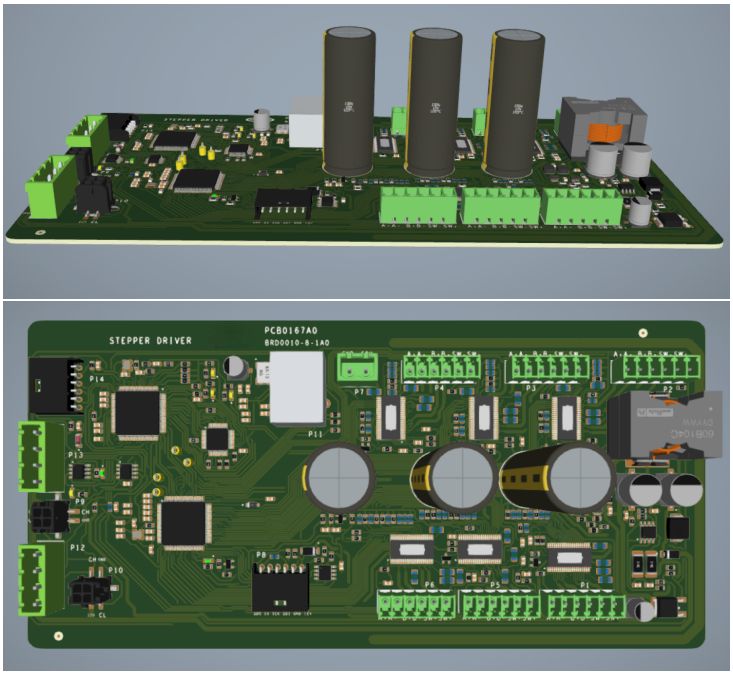


Figure 4.3.1: 3D rendering of the PCB layout illustrating component placement and mechanical structure, shown from front and top views.

The board measures 105 mm in width and has a variable length up to 217 mm. It is built on a double-layer substrate, with the top layer dedicated to signal routing and the bottom layer primarily used as a ground plane, with a few traces dedicated to signal return paths to minimize interference and ensure signal integrity. This layout choice enhances electromagnetic compatibility (EMC) and reduces potential cross-talk between critical traces.

The PCB thickness is 1.6 mm, a standard value that ensures mechanical robustness and compatibility with automated assembly processes such as pick-and-place and wave soldering. Vias are of a standard type, sized to guarantee proper signal and current passage without compromising electrical integrity or mechanical strength, and are strategically placed to minimize impedance discontinuities and thermal hotspots.

For aesthetic and protective finishing, the board features a green soldermask that protects against oxidation, moisture, and short circuits. Additionally, a white silkscreen layer is applied to identify components, pin headers, test points, and polarity marks, facilitating not only assembly and testing, but also long-term maintenance and troubleshooting.

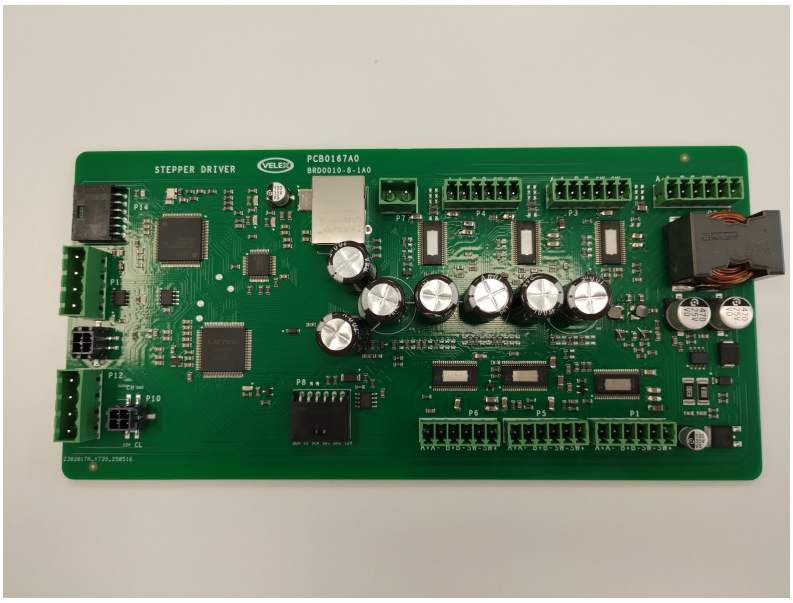


Figure 4.3.2: Top view of the stepper driver board after assembly.

4.3 Trace Sizing

Particular attention was given to the trace sizing on the PCB, a critical aspect to ensure that tracks can safely carry the circuit's required current, preventing excessive overheating or voltage drops that could jeopardize the proper functioning and reliability of the entire system. The trace width is not an arbitrary value; it depends on several key factors, such as the maximum current to be carried, the copper thickness used in PCB manufacturing, typically around 35 microns (1 oz/ft^2)and the allowable temperature rise during operation.
For an accurate estimation of the required trace width, reference was made to the international standard IPC-2221 ("Generic Standard on Printed Board Design"), which provides guidelines and specific formulas for calculating trace dimensions based on operational conditions and electrical and thermal requirements of the design. This standard takes into account not only the operating current but also the ambient temperature and copper thickness, which are essential to ensure the durability and safety of the printed circuit board.
Specifically, the following widely adopted industrial formulas were used for the calculation:

$$A = \left(\frac{I}{k \cdot T_{\text{rise}}^{b}} \right)^{\frac{1}{c}} \qquad W = \frac{A}{t \times 1.378} \tag{4.1}$$

where:

- A is the trace area in mils^2,
- I is the current in amperes,

- T_{rise} is the allowable temperature rise in degrees Celsius,
- t is the copper thickness in mils,
- k, b, and c are empirical constants with values 0.024, 0.44, and 0.725 respectively.

By applying these calculations, the optimal trace widths for all sections of the board were accurately determined. Most standard traces were sized between 0.25 and 0.5 mm in width, ensuring safe and stable operation for typical signal circuit currents. For the most critical traces, particularly those carrying higher currents or power voltages, such as the power supply lines to motor drivers and certain motor output lines, wider traces ranging approximately from 1 to 3 mm were adopted, as can be seen in **Figure 4.3.1**. This approach reduced electrical resistance and improved thermal dissipation, thus maintaining high system reliability even under heavy load conditions.

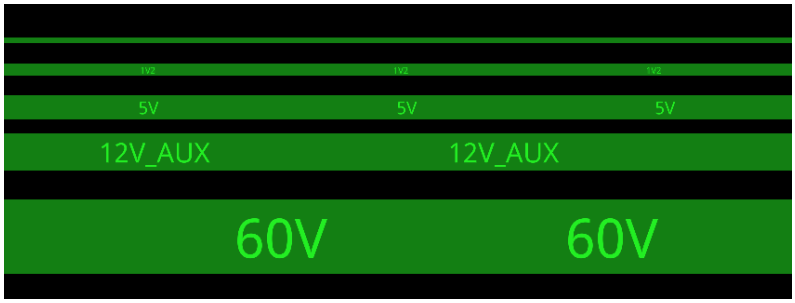


Figure 4.3.1 : Trace width variation for carrying voltages from 60V to 1.2V and logic signals.

Chapter 5

Acceleration Profile Design

Unlike DC motors, whose speed can be controlled by varying the supply voltage or the duty cycle in the case of PWM modulation, stepper motors operate according to a different principle. Their speed is exclusively determined by the frequency of the command pulses sent to the driver, as each pulse corresponds to the execution of a single step or microstep. Therefore, increasing the pulse frequency proportionally increases the angular speed of the motor.

This principle requires careful management of the stepper motor's start-up and stop phases. In fact, a direct start at high frequencies may result in missed steps or even motor stall, caused by the rotor's inability to instantly follow the variation of the magnetic field generated by the windings, due to its mechanical inertia and the available torque.

Each stepper motor is characterized by a maximum no-load starting frequency, generally referred to as the starting frequency (f_s), beyond which direct starting is unreliable. If the motor is commanded to start from standstill at a frequency higher than this limit, synchronization may be lost, compromising the correct execution of the stepping sequence. This phenomenon, especially in open-loop control systems without feedback, leads to a loss of positioning accuracy.

To mitigate these issues, it is essential to adopt an acceleration profile, that is, a phase in which the frequency of the command pulses is gradually increased. This allows the rotor to accelerate in a controlled manner, preventing loss of synchronism. Similarly, during the stopping phase, a progressive deceleration must be implemented, reducing the pulse frequency to allow a smooth stop and prevent undesired residual motion.

In the context of an FPGA implementation, the acceleration and deceleration profile is typically managed through a finite state machine (FSM), which controls the generation of the step pulses and dynamically computes the waiting time between consecutive pulses. Depending on the required complexity, the profile can be designed using different approaches: linear profile, exponential profile, precomputed lookup tables, or mathematical formulas derived from uniformly accelerated motion, such as square root functions.

In the absence of feedback sensors (open-loop control), the accuracy of the velocity profile

implementation is crucial to ensure that the motor performs exactly the intended number of steps without positional errors. Therefore, an optimal management of the acceleration and deceleration profile is an essential requirement for achieving reliable and precise performance, particularly in applications where positioning control is critical.

5.1 Trapezoidal Profile

To accelerate a stepper motor from an initial speed to a steady-state speed, it is necessary to progressively modify the frequency of the pulses sent to the driver. Since, as previously discussed, the motor speed depends solely on the frequency of the `STEP` signals, acceleration can be achieved by increasing this frequency at regular intervals.

The most common implementation uses a microcontroller to handle these dynamics through two separate timers: one to generate the step pulses at the desired frequency (SPS timer, *Steps Per Second*), and another to manage the temporal variation of the speed (acceleration timer). Each time the acceleration timer expires, the value of the *SPS* timer is updated, thus modifying the frequency of the `STEP` signal. This behavior, which represents a variation of angular velocity over time ($\mathrm{d}v/\mathrm{d}t$), corresponds to a true acceleration, measured in steps per second squared (*SPS/s*).

Figure 5.1.1 shows a close-up view of a typical acceleration profile implemented on a microcontroller, highlighting the system behavior during the phase of increasing the stepper motor's speed until the target velocity is reached.

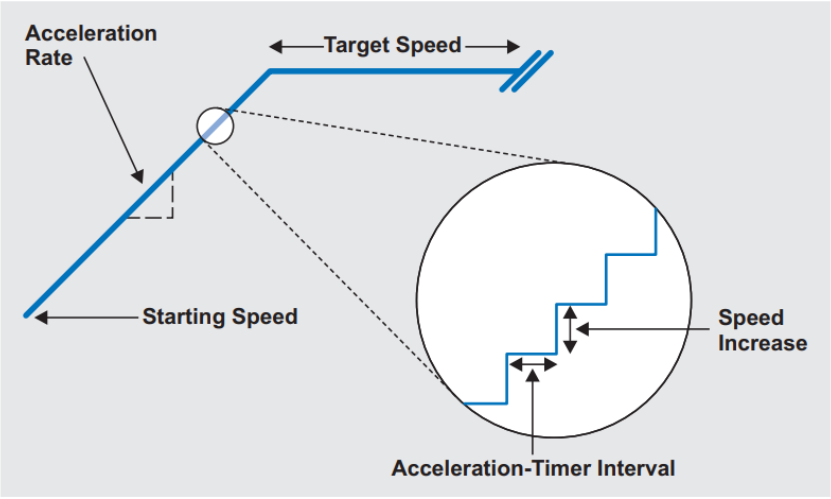


Figure 5.1.1: Close-up on a typical acceleration profile. [33]

The value of the *SPS* timer depends on the system clock frequency and the desired speed. The equation generally used to determine it is as follows:

$$\text{SPS\_timer\_register} = \frac{f_{\text{clk}}}{\text{SPS}} \tag{5.1}$$

where f_{clk} represents the timer clock frequency (e.g., 100 MHz), and *SPS* is the desired number of steps (or microsteps) per second.
In a microcontroller, this value configures the hardware timer that dictates the pace of the step pulses. In an FPGA implementation, the principle is analogous: a counter is used to generate a `STEP` pulse every `STEP_DELAY` clock cycles. This `STEP_DELAY` is directly related to the desired frequency and is regularly updated during the acceleration and deceleration phases.
Two fundamental parameters define the acceleration profile:

- **SPS update frequency**: indicates how often the speed value should be modified;
- **Increment (or decrement) per update**: specifies the amount by which the frequency should change at each interval.

The product of these two values defines the slope of the acceleration profile. For example, if the speed is updated every millisecond and increases by 10 SPS at each update, the resulting acceleration will be $10\,000$ SPS/s.
A practical example can help clarify the concept: suppose we want to accelerate a motor from 1 SPS to 1000 SPS in one second. This entails a total increase of 1000 SPS over a time interval of one second. This result can be achieved in several ways:

- updating the speed every 1 millisecond and increasing by 1 SPS at each update, for a total of 1000 updates;
- updating every 2 milliseconds and increasing by 2 SPS, for a total of 500 updates;
- updating every 5 milliseconds and increasing by 5 SPS, for a total of 200 updates.

The choice among these solutions represents a trade-off between velocity profile resolution and system computational load. A higher update frequency allows for a smoother profile and better approximation of the desired acceleration, but also requires more operations to modify registers or counters. This trade-off is generally negligible in microcontrollers, which typically have ample memory resources and computational capabilities.
Conversely, in FPGAs, where logical resources and internal memory are significantly more limited, this choice plays a crucial role in the overall design.

Once the target speed is reached, the system enters the constant-speed phase (*run*), during which the frequency value remains unchanged. Subsequently, to stop the motor in a controlled manner, a deceleration profile is applied, either symmetric to the acceleration phase (with identical times and increments but of opposite sign), or asymmetric, if a slower or faster stop is desired.

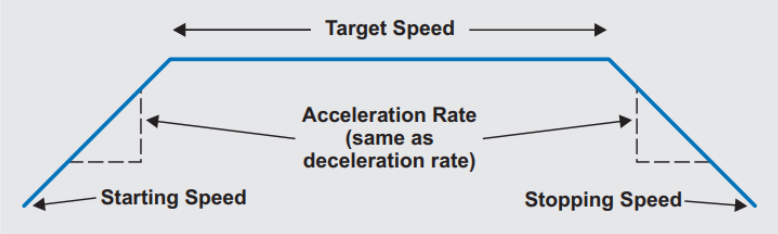


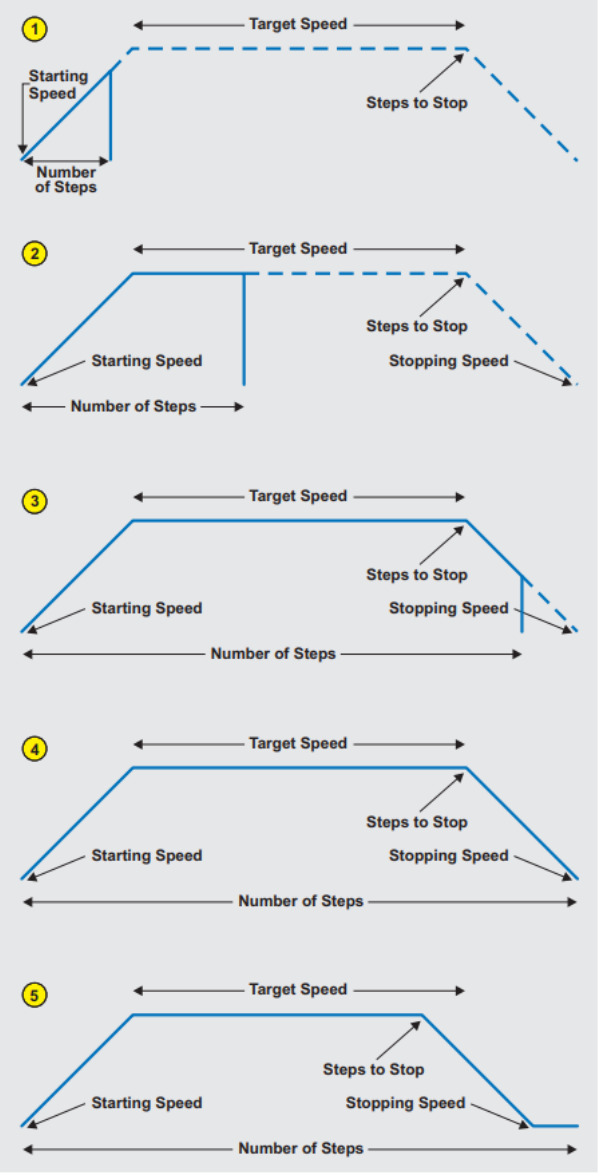
Figure 5.1.2: Complete trapezoidal acceleration profile. [33]

Finally, in an *open-loop* control system, such as those commonly used with stepper motors, it is essential that the number of generated pulses exactly matches the desired number of steps. Any loss of synchronization, caused, for example, by overly abrupt acceleration or sudden stops, results in a positioning error that cannot be corrected, since the system lacks feedback. For this reason, the quality and precision of the acceleration and deceleration profiles are essential for the correct operation and reliability of the control system.
Figure 5.1.2 illustrates a typical acceleration and deceleration profile, highlighting the initial speed, target speed, acceleration rate (assumed equal to the deceleration rate), and final stopping speed.

5.2 Position Control

So far, motor control has been analyzed in the context of a speed control system, in which the motor is accelerated to a target speed and then stopped via an external command. However, in many practical applications, it is necessary to perform a predetermined number of steps within a defined time frame. In this scenario, the acceleration and deceleration profiles play a fundamental role, as it is not possible to wait for an external command to start deceleration: the motor must stop exactly at the end of the expected steps.
The parameter that defines the total number of steps to be executed is commonly referred to as `NUMBER_OF_STEPS`. The motion profile must therefore be designed to ensure that the motor stops precisely upon completing this number of steps. A well-established technique involves introducing an auxiliary variable, such as `STEPS_TO_STOP`, which represents the remaining number of steps required to complete the deceleration profile. When the remaining step count falls below this threshold, the system automatically initiates the deceleration phase.

Depending on the system configuration, five main scenarios may occur, conceptually illustrated in **Figure 5.2.1**:



- **Scenario 1** – All steps are completed before reaching the target speed.
- **Scenario 2** – All steps are executed during the constant-speed phase.
- **Scenario 3** – All steps are completed before the deceleration phase ends.
- **Scenario 4** – All steps are completed exactly at the end of deceleration (ideal case).
- **Scenario 5** – All steps are completed after the deceleration phase has ended.

Figure 5.2.1: Main possible scenarios. [33]

The optimal scenario is clearly number 4, where the motor stops exactly at the final step, at a safe speed. Scenarios 3 and 5 are generally acceptable if the deviation is minimal. However, in the case of scenario 3, overshooting the motion profile can lead to position loss due to the rotor's inertia. In contrast, in Scenario 5, the motor operates through a greater number of steps at its minimum speed, resulting in an increased overall execution time, which may become unacceptably long.

Scenarios 1 and 2, on the other hand, must be avoided: the system must be designed so that `STEPS_TO_STOP` is always less than `NUMBER_OF_STEPS`.

A simpler alternative to dynamically managing `STEPS_TO_STOP` is to segment the total number of steps into fixed percentages assigned to each phase of the motion profile (**Figure 5.2.2**). A common approach includes the following subdivision:

- 20% of the steps for the acceleration phase;
- 60% of the steps for the constant-speed phase;
- 20% of the steps for the deceleration phase.

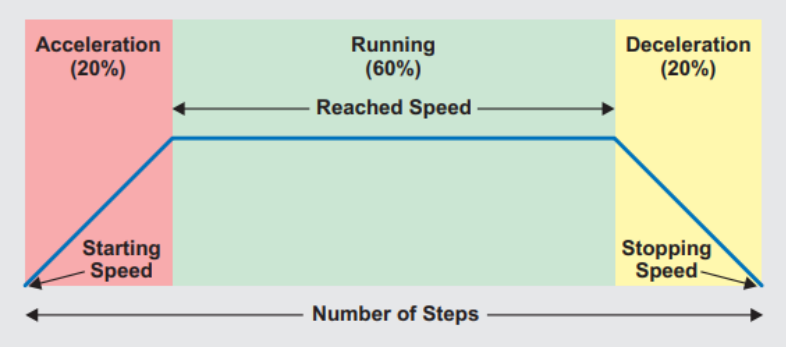


Figure 5.2.2: Percentage-based acceleration profile. [33]

If `NUMBER_OF_STEPS` = 1000, the motor accelerates for 200 steps, runs at constant speed for 600 steps, and finally decelerates during the remaining 200 steps. This approach has the advantage of ensuring that the motor stops in a safe position without the risk of running out of steps before deceleration begins. However, it also presents a significant limitation: the target speed is not deterministic. It depends on the acceleration rate and the number of steps assigned to the initial phase.

If the achieved speed is insufficient for the application, it is necessary to either increase the acceleration rate (as long as it is within the available torque limit) or allocate a higher percentage of steps to the acceleration phase.

In both cases, it is essential to avoid exceeding the physical limits of the system, such as the maximum torque deliverable by the motor or the maximum speed supported by the load's torque/speed curve. The balance of dynamic performance and system reliability, therefore, represents one of the key design challenges.

5.3 Critical Aspects of FPGA Implementation

The traditional implementation of a trapezoidal acceleration profile for stepper motors is commonly carried out using microcontrollers. Due to their flexibility and availability of computational resources, microcontrollers can directly handle relatively complex mathematical operations, such as divisions or floating-point calculations.
However, when an FPGA is adopted as the control platform, the scenario changes significantly, especially in contexts where it is necessary to manage multiple motors simultaneously within a single device. In such situations, the availability of hardware resources becomes highly constrained, and the optimization of each architectural component becomes crucial.
In particular, implementing complex arithmetic functions such as division requires special attention. Unlike multiplication, which can be efficiently handled by leveraging DSP (Digital Signal Processing) blocks available in most modern FPGAs, division typically lacks dedicated hardware resources. As a result, it must be implemented using combinational or sequential logic, with a significant impact in terms of the utilization of logic resources.
Furthermore, during synthesis, the FPGA tends to implement divisions as combinational operations, attempting to complete them within a single clock cycle. This approach results in extremely high use of Look-Up Tables (LUTs),the fundamental structures used to represent logic functions within the device, negatively affecting both area occupation and overall timing.
This implies that direct implementation of division consumes a significant portion of the available logic area on the FPGA, thereby increasing the overall complexity of the design. This increase in complexity may also adversely affect the maximum achievable clock frequency, thus limiting the system's temporal performance. Finally, circuits dedicated to direct division typically lead to increased latency, elongating the overall response time of the device.
For this reason, FPGA implementations commonly adopt various strategies to minimize the impact of division operations. These include:

- **Iterative division:** algorithms that perform division over multiple clock cycles, enabling a trade-off between execution speed and hardware resource usage. Although such methods are more efficient in terms of logic area, they come with higher latency compared to direct combinational division.

- **Use of dedicated IP cores:** FPGA development tools often offer optimized modules for arithmetic operations, including pipelined division modules. These IP cores enable high performance while maintaining control over resource consumption.

- **Mathematical simplifications:** in many cases, direct division is avoided by using alternative methods such as mathematical approximations, multiplication by precomputed

inverse values stored in memory, or conversion of the division into shift and subtraction operations, particularly effective when the divisor is a power of two or close to it.

These techniques allow for high operational frequency and reduced resource usage, key characteristics in systems that must simultaneously control multiple stepper motors, each with its own acceleration and deceleration profile.
An example of an algorithm initially adopted in this project, but later replaced by a more efficient solution, is the **restoring division** algorithm (**Figure 5.3.1**).

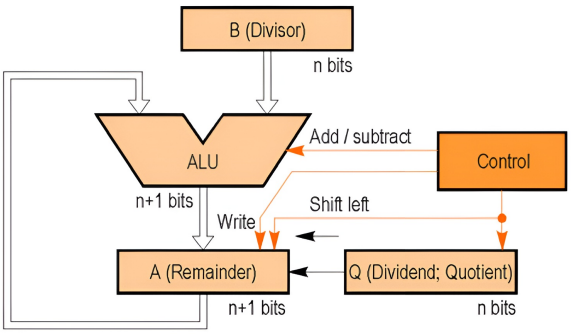


Figure 5.3.1: Circuit for restoring division. [36]

The purpose of this algorithm is to perform integer division using only elementary operations. Its general behavior consists of producing one quotient bit per cycle, proceeding from left to right through a sequence of shifts and subtractions.
The circuit uses three main registers: the `Q` register, which initially contains the dividend and eventually stores the quotient; the `R` register, which holds the partial remainder during processing; and the `D` register, which contains the divisor.
The entire procedure is repeated for a number of cycles equal to the number of bits in the dividend. In each cycle, the system performs the following operations:

- **Combined left shift:** the contents of the `R` and `Q` registers are shifted left by one bit. The most significant bit of `Q` enters the least significant bit of `R`, as if the two registers were concatenated into a single value to manipulate. This shift makes "room" to calculate the next quotient bit.

- **Subtraction of the divisor:** an attempt is made to subtract `D` from the new value of `R`. The result of this subtraction is used to determine whether the divisor fits into the partial remainder.

- **Result check:**
  - If the value of R after subtraction is positive or zero, it means the divisor fits into the remainder. In this case, the new value of R is retained, and the current bit of the quotient (i.e., the least significant bit of Q) is set to 1.
  - If the value of R is negative, it means the divisor is too large to subtract from the remainder. Therefore, the previous value of R is restored (hence the term *restoring*), and the current quotient bit is set to 0.

This process continues cyclically, with one shift and one subtraction for each bit of the quotient. At the end of the final iteration, the Q register contains the final quotient, while the R register holds the division remainder.

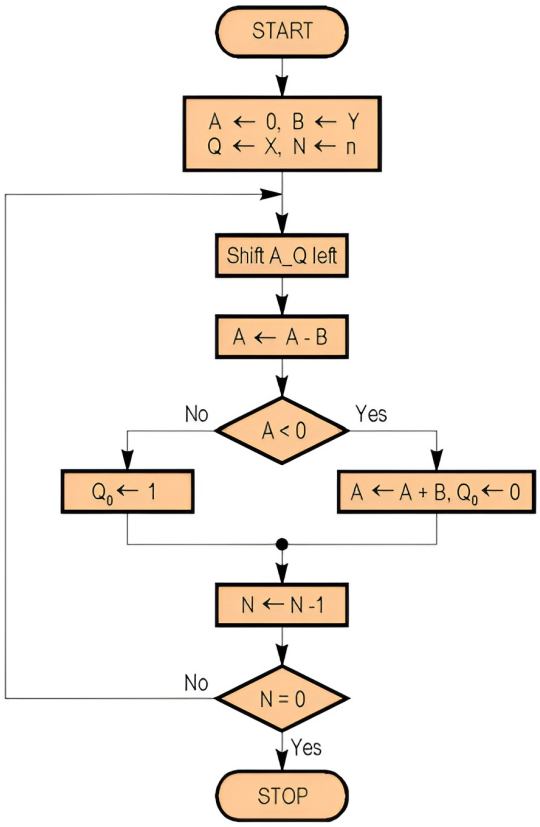


Figure 5.3.2: Algorithm for restoring division. [36]

5.4 Implementation of Linear Speed Control

An efficient algorithm for implementing a simplified acceleration profile for stepper motors is described in an application note by **Atmel**, which has become a well-established reference in many embedded applications.

To achieve controlled rotary motion in a stepper motor, the current flowing through the windings must vary according to a correct and precise sequence. This sequence is generated by a dedicated driver, which responds to command pulses and a direction signal.
To rotate the motor at a constant speed, the command pulses must be generated at regular intervals, as illustrated in **Figure 5.4.1**.

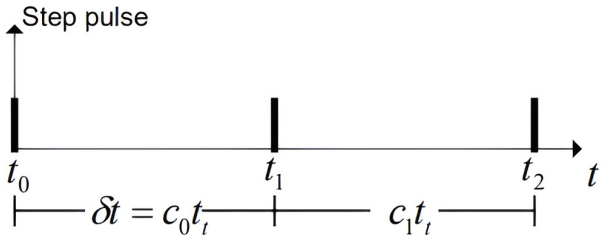


Figure 5.4.1: Stepper motor pulses. [34]

The generation of these pulses is handled by a counter operating at a frequency f_t [Hz]. The delay δt, programmed by the counter and corresponding to the time interval between two consecutive pulses, is defined as:

$$\delta t = \frac{c_t}{f_t} \quad [s] \tag{5.2}$$

where c_t is the timer count value.
The kinematic parameters of the motor are:

- **Step angle** $\alpha = \frac{2\pi}{spr}$ [rad], where *spr* is the number of steps per full revolution (200);
- **Angular position** $\theta = n\alpha$ [rad], where n is the number of executed steps;
- **Angular velocity** $\omega = \frac{\alpha}{\delta t}$ [rad/s].

Additionally, the following conversion applies:

$$1\,\text{rad/s} = 9.55\,\text{rpm}.$$

To smoothly start and stop the motor, it is necessary to control the acceleration and deceleration profiles, maintaining a constant change in speed over time, i.e. constant acceleration. This results in a linear speed profile over time, as shown in **Figure 5.4.2**.
The motor speed is directly controlled by the delay δt between pulses; therefore, accurately calculating these delays is essential to ensure the motor faithfully follows the desired speed profile.

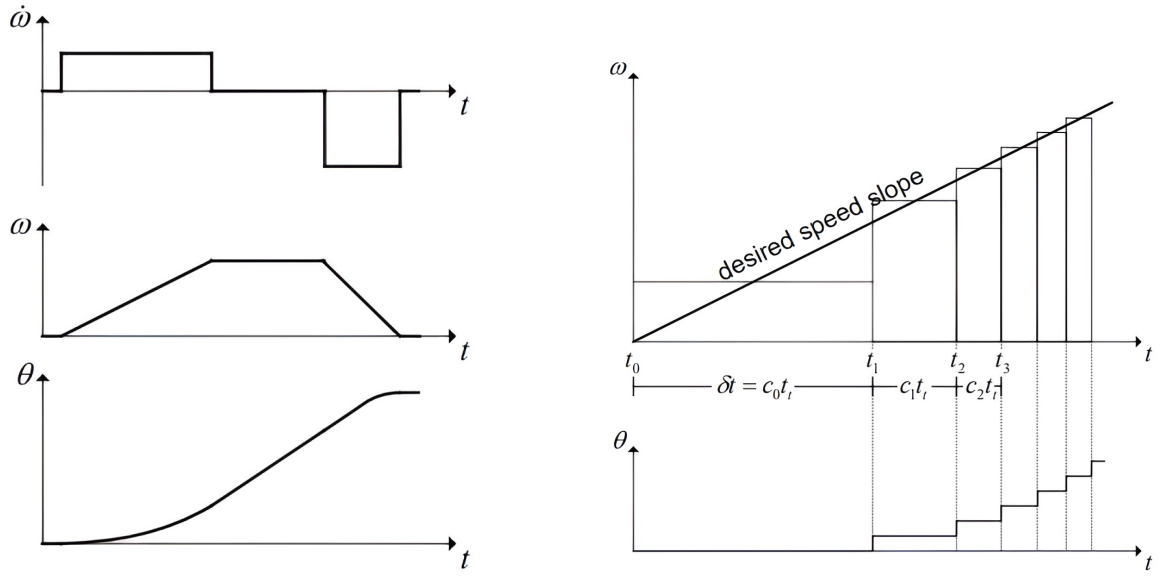


Figure 5.4.2: (a) Acceleration ($\dot{\omega}$), speed (ω) and position (θ); (b) speed profile vs. stepper motor pulses/speed. [34]

The initial delay counter c_0, as well as the subsequent c_n, are given by:

$$c_0 = f_{\text{timer}} \cdot \sqrt{\frac{2\alpha}{\dot{\omega}}} \qquad c_n = c_0 \left(\sqrt{n+1} - \sqrt{n} \right) \tag{5.3}$$

This formula involves computing two square roots, which are computationally expensive operations, especially in the context of an FPGA.

To reduce computational complexity, the algorithm adopts a Taylor series approximation of the delay formula. This approximation allows for the recursive calculation of c_n using a simpler formula:

$$c_n = c_{n-1} - \frac{2c_{n-1}}{4n+1} \tag{5.4}$$

This expression is much more efficient to compute, requiring only subtractions and divisions by integers, but introduces a significant initial error, approximately 0.44 at step $n = 1$. To compensate for this error, the initial value c_0 can be multiplied by an empirical correction factor, for example 0.676, thus improving the overall accuracy of the profile.

As shown, acceleration is defined in terms of the parameters c_0 and n. When the acceleration (or deceleration) value changes, it is necessary to recalculate the value of n to properly adapt the speed profile.

The time t_n and parameter n can be expressed as functions of the angular acceleration $\dot{\omega}$, angular velocity ω, and step angle α of the motor, according to the following relations:

$$t_n = \frac{\omega_n}{\dot{\omega}} \qquad n = \frac{\dot{\omega} t_n^2}{2\alpha} \tag{5.5}$$

Combining these two equations yields the following relationship:

$$n\dot{\omega} = \frac{\omega^2}{2\alpha} \tag{5.6}$$

From this expression, an important result emerges: the number of steps required to reach a given speed is inversely proportional to the acceleration.

$$n_1\dot{\omega}_1 = n_2\dot{\omega}_2 \tag{5.7}$$

This means that changing the acceleration from a value $\dot{\omega}_1$ to a value $\dot{\omega}_2$ is equivalent to modifying the corresponding parameter n in the control profile.
The implementation of the acceleration and deceleration profile control algorithm requires five fundamental input parameters:

-Acceleration ($\dot{\omega}_{accel}$)

-Deceleration ($\dot{\omega}_{decel}$)

-Minimum speed (ω_{min})

-Target maximum speed (ω_{max})

-Total number of steps to execute (N_{steps})

Based on these parameters, two main operating scenarios may occur:

- **Trapezoidal profile**: acceleration continues until the target speed ω_{max} is reached, which is maintained until deceleration begins.
- **Triangular profile**: deceleration starts before the target speed is reached, so the motor never achieves the programmed maximum speed.

Trapezoidal profile: reaching the target speed before deceleration begins

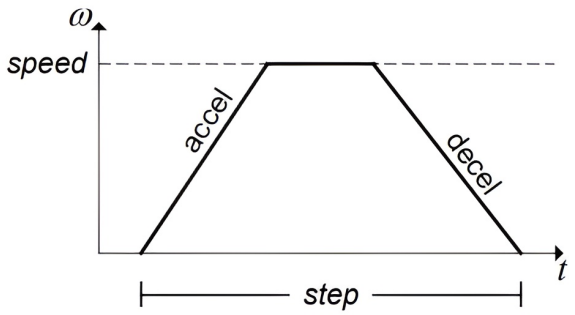


Figure 5.4.3: Trapezoidal acceleration profile. [34]

As shown in **Figure 5.4.3**, in this scenario the speed ramp reaches the target value ω_{max} before the deceleration phase begins.
`max_s_lim` represents the number of steps required to reach the target speed, calculated as:

$$\texttt{max\_s\_lim} = \frac{\omega_{max}^2}{2\alpha\dot{\omega}_{accel}}$$

where α is the step angle of the motor.
`accel_lim` is the total number of steps available before deceleration must begin, defined as:

$$\texttt{accel\_lim} = N_{steps} \cdot \frac{\dot{\omega}_{decel}}{\dot{\omega}_{accel} + \dot{\omega}_{decel}}$$

If the following condition is met:

$$\texttt{max\_s\_lim} < \texttt{accel\_lim}$$

the acceleration is limited by the attainment of the desired speed. In this case, the number of steps at the end of the deceleration phase (`decel_val`) is computed based on this condition (the exact formula may depend on the specific algorithm or implementation used).

Triangular profile: deceleration begins before the desired speed is reached

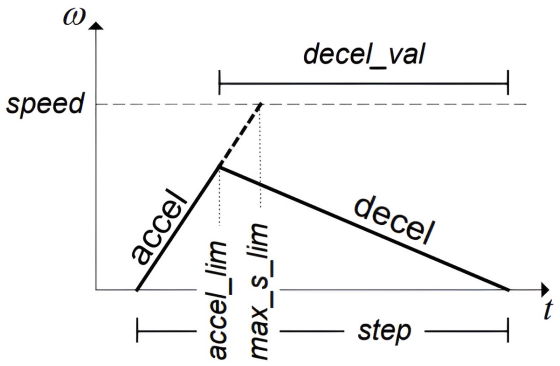


Figure 5.4.4: Triangular acceleration profile. [34]

In the scenario illustrated in **Figure 5.4.4**, deceleration must begin before the motor reaches the target speed. This implies that the number of available steps for acceleration is limited, and the actual maximum speed reached will be lower than ω_{max}. In this context, the velocity profile takes on a triangular shape, with a peak speed reached at the midpoint if acceleration and deceleration are equal, or earlier if deceleration is greater than acceleration (the opposite case is rare). Then follows the deceleration phase, which stops the motor exactly at the end of the total number of steps N_{steps}.

The two examined scenarios allow the control algorithm to adapt, ensuring compliance with the total step constraint. This approach enables optimization of the transition between acceleration, constant speed (if any), and deceleration phases.
To reduce resource usage and improve FPGA implementation performance, a strategy was adopted that completely eliminates division operations during execution. To address this issue, critical values necessary for speed profile control, particularly inter-pulse timings and associated frequencies, are precomputed. These values are stored in tables within embedded memory blocks on the FPGA. During operation, the motor control module directly accesses these tables, retrieving the required values based on the current step or desired speed, thus avoiding complex real-time calculations.

The implementation of the stepper motor movement profile is performed through a **finite state machine** (FSM), composed of four main states: Stop, Accel, Run, Decel (**Figure 5.4.5**).

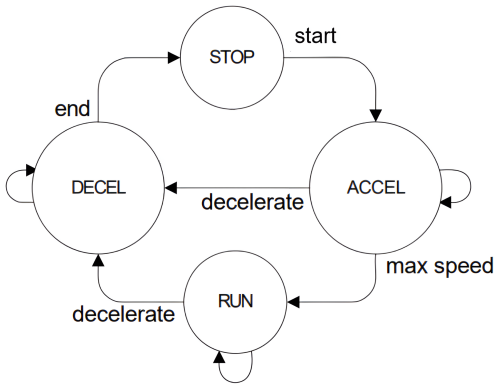


Figure 5.4.5: Finite state machine. [34]

- **STOP state**: when the application starts, or whenever the stepper motor is stopped, the FSM is in the STOP state. In this phase, the necessary setup calculations for the speed profile are executed, such as the computation of initial parameters or the preparation of precalculated tables. The timer or counter that generates pulses is disabled or kept inactive. Once the initial setup is complete and a movement command is received, the FSM transitions to the acceleration state.
- **ACCEL state**: during the ACCEL state, the speed profile progressively increases the frequency of the motor pulses, accelerating it in a controlled manner. The FSM remains in this state until:
  - The motor reaches the desired speed, at which point the state transitions to RUN to maintain constant speed; or
  - Deceleration must begin before reaching the maximum speed (triangular profile case), in which case the state changes to DECEL.
- **RUN state**: in this phase, the motor speed is kept constant, with pulses generated at a fixed frequency. The FSM remains in the RUN state until, after a certain number of steps, deceleration must begin, and the state transitions to DECEL.
- **DECEL state**: during deceleration, the pulse frequency is progressively reduced until the motor stops precisely at the desired number of steps. At the end of the deceleration, when the speed reaches zero, the FSM returns to the STOP state, waiting for a new command.

Chapter 6

Logic Design and Modular Implementation

6.1 General Architecture of the Project

The core of the project is the generation of precise motion profiles for each motor, including the phases of acceleration, constant speed, and deceleration. This approach enables smooth motion, reducing mechanical wear and improving operational efficiency.
The overall architecture comprises integrated hardware and software components, robust communication protocols, and optimized motor control. The FPGA implementation was designed considering technological constraints and synthesis parameters to ensure high performance and reliability.
The system is structured around five main VHDL entities, each with specific tasks, all gathered into a single *top entity* named `STEPPER_CONTROL`. The essential components are:

- `SPI_SLAVE`, responsible for serial communication with the microcontroller;
- `CONTROLLER`, which interprets received commands and coordinates data flow;
- `RAM`, used as temporary memory for configuration and control data;
- `MOTOR`, implementing the logic to generate the control signal for a single stepper motor;
- `MOTORS_CONTROL_UNIT`, tasked with managing the six motors simultaneously, synchronizing their actions with the rest of the system.

The primary objective is to guarantee effective and reliable communication between the microcontroller, FPGA, and stepper motor drivers. Through the SPI protocol, the microcontroller sends commands to the FPGA, which uses them to precisely and dynamically regulate the behavior of the motors. This scheme delegates the more complex temporal and logical management to the FPGA, reducing the microcontroller's load, and ensuring low-latency control.

The system operates in two well-defined phases: configuration phase and run phase. During configuration, the microcontroller sets the fundamental parameters for each motor. Motor selection is performed via commands ranging from `0x31` to `0x36`, corresponding to the six available motors. Subsequently, parameters defining the motion profile, such as initial speed, maximum speed, acceleration, and deceleration, are transmitted in byte form to allow detailed and flexible configuration.
Once configuration is complete, the system can switch to execution mode using the `0x2F` command, while returning to configuration mode is possible with the `0x2E` command. In run mode, the motors use the preloaded parameters and wait for the number of steps to follow (command `0x71`) to be loaded. From these data, the FPGA generates the pulse sequence necessary for precise and synchronized control.
In addition to operational functions, the system provides diagnostic and monitoring commands, essential to verify status and safety during operation. These include querying active parameters or the state of motor limit switches, critical signals to prevent movements beyond limits, or potentially damaging ones. This information allows the microcontroller to make timely decisions, improving overall reliability.
The combination of modules and functionalities described here composes a modular and flexible system capable of effectively interfacing with the microcontroller and managing simultaneous control of multiple stepper motors, precisely regulating speed profiles via the FPGA.

For the project synthesis, the native development environment for the selected FPGA device, namely *Lattice IceCube2*, was employed. This software provides a comprehensive and integrated toolchain that supports all stages of the hardware project lifecycle: from writing VHDL code, through simulation and synthesis, to generating the device configuration file.
IceCube2 proved essential in translating the VHDL code into an efficient and optimized hardware representation, respecting the specific constraints of the target FPGA. During synthesis, the software provides a detailed report on the usage of available logical resources, such as lookup tables (LUTs), programmable logic blocks (PLBs), registers (flip-flops), internal RAM memories, and I/O resources. These data were analyzed in tabular form (**Table 6.1**), allowing evaluation and balance of resource occupation, a fundamental aspect to maintain expansion margins and adequate performance.

Table 6.1: Numerical representation of total resources utilized.

Final Design Statistics	Used Devices	Available Devices
Logic Cells	1064	1280
Flip Flops	566	1280
Carrys	195	1280
RAMs	12	16
IOs	27	73
GBIOs	3	8
PLBs	156	160

Simultaneously, IceCube2 offers a graphical representation of FPGA resource usage through placement maps of logical resources on the chip. This visualization (**Figure 6.1.1**) helps to understand the spatial distribution of the designed modules, avoiding congestion and facilitating possible floor planning interventions.

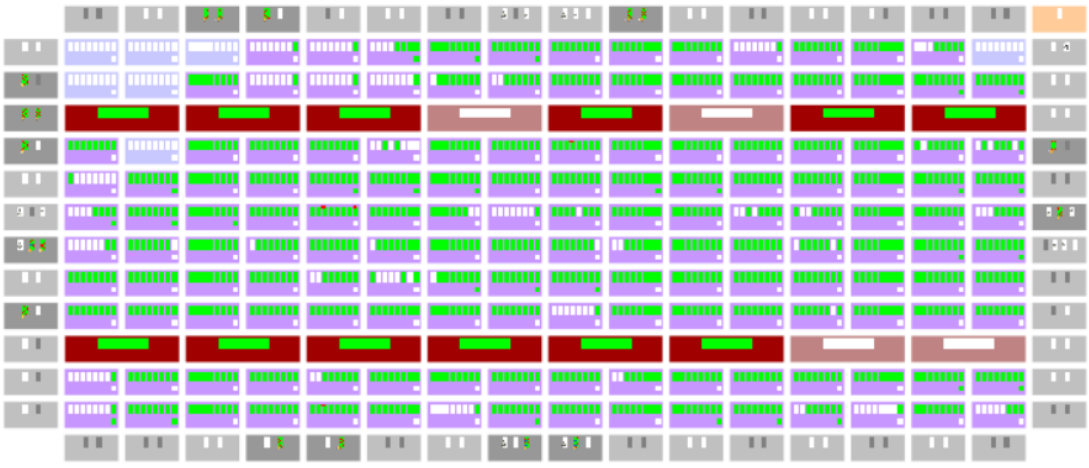


Figure 6.1.1: Visual representation of the total resources used. LUTs in green, PLBs in blue, RAM in red.

Another critical aspect managed by the development environment is timing analysis. IceCube2 enables the verification that clock signals and critical paths comply with the required temporal constraints for proper operation. Accurate timing management is indispensable in a project like this, where multiple modules must communicate synchronously and multiple clock frequencies are involved.

Additionally, IceCube2 provides tools to estimate the power consumption of the project, an important feature for evaluating energy efficiency and thermal dissipation of the device, especially in embedded or industrial applications.
Finally, a valuable aid to understanding and verifying the project is the automatic generation of the block diagram (**Figure 6.1.2**), which graphically represents the implemented entities and their interconnections. This visual scheme is particularly useful for illustrating the overall architecture, facilitating communication among team members, and documenting the project clearly and immediately.

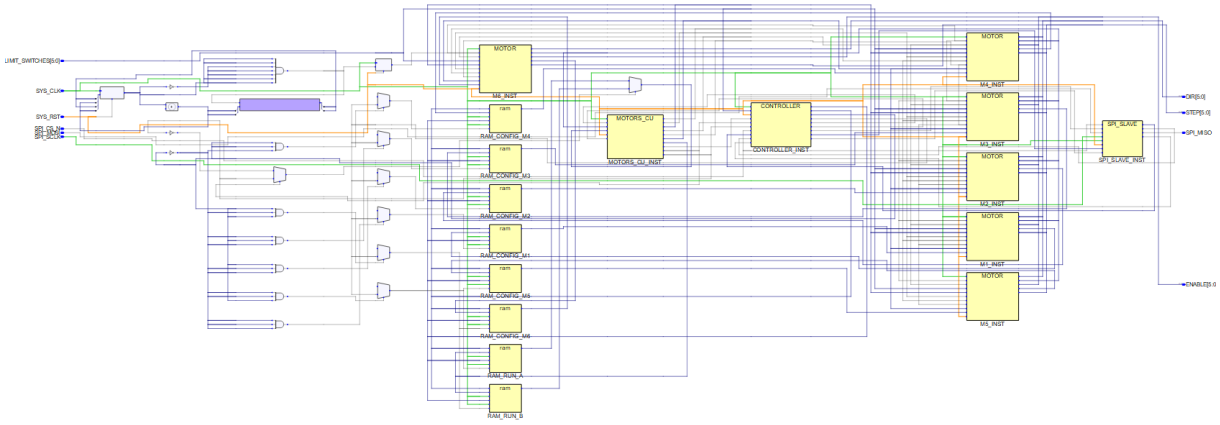


Figure 6.1.2: Hierarchical visualization of the utilized architecture.

All these tools provided by IceCube2 enabled an iterative design and optimization process, allowing an effective balance among performance, hardware resources, and power consumption, while ensuring the correct functionality of the system as a whole.

6.2 Functional Block Analysis

6.2.1 SPI_SLAVE

The module under examination implements the *SPI* (Serial Peripheral Interface) communication protocol in *slave* mode, allowing the FPGA device to receive commands and data from the external microcontroller. This communication channel plays a crucial role in the dynamic configuration of the system, as it enables the microcontroller to send operational instructions such as the desired speed, the number of steps to execute, and parameters related to acceleration and deceleration.
The module manages the main signals of the SPI protocol: *clock (SCLK)*, *chip select (CS)*, and *data lines (MOSI/MISO)*. Internally, the logic correctly decodes the data stream according to

the SPI specification, ensuring synchronous and reliable transfer of information to the system's main control module.

```
-- Tx data
P_TX_DATA_REG : process(SYS_CLK, SYS_RST)
begin
    if(SYS_RST = '1' ) then
        TX_DATA_REG <= (others => '0' );
    elsif SYS_CLK'event and SYS_CLK = '1' then
        if (TX_DATA_WR = '1' and TX_READY_IN = '1' and CS_N = '0' ) then
            TX_DATA_REG <= TX_DATA(7 downto 0);
        end if;
    end if;
end process P_TX_DATA_REG;

-- Rx data
P_RX_DATA_REG : process(SYS_CLK, SYS_RST)
begin
    if(SYS_RST = '1' ) then
        RX_DATA <= (others => '0' );
    elsif SYS_CLK'event and SYS_CLK = '1' then
        if RX_DONE_SYNC = '1' and CS_N = '0' then
            RX_DATA <= RX_DATA_REG;
        end if;
    end if;
end process P_RX_DATA_REG;
```

Listing 6.1: SPI transmission and reception registers.

The following portion of code implements two main processes dedicated to handling data transmission (Tx) and reception (Rx) within the SPI module operating in *slave* mode.

Transmission Data Handling:

The `P_TX_DATA_REG` process is sensitive to both the system clock signal `SYS_CLK` and the asynchronous reset signal `SYS_RST`. When the reset is active (`SYS_RST = '1'`), the register `TX_DATA_REG` is initialized by clearing all its bits.

On the rising edge of the clock (`SYS_CLK'event and SYS_CLK = '1'`), the process checks several conditions to enable writing: the write request signal `TX_DATA_WR` must be active, the `TX_READY_IN` signal (indicating readiness to transmit) must be high, and the chip select signal `CS_N` must be active low (`'0'`). Only when all these conditions are met, the content of the input data bus `TX_DATA(7 downto 0)` is stored in the `TX_DATA_REG` register.

This mechanism ensures that the data to be transmitted are loaded only when the device is properly selected and ready to transmit, thus avoiding unintended writes or error conditions.

Reception Data Handling:

The second process manages the reception phase and is also sensitive to the SYS_CLK and SYS_RST signals. Under reset conditions, the output register RX_DATA is cleared.

On the rising edge of the clock, if the RX_DONE_SYNC signal indicates that a complete reception has occurred and the CS_N signal is active low, then the contents of the intermediate register RX_DATA_REG are copied into the RX_DATA register.

This logic ensures that received data are updated only at the end of a valid transmission and only when the SPI device is actively selected, thereby ensuring consistency and synchronization between SPI communication and the internal control module.

```
-- SPI slave receiver ------

-- sample MOSI on rising edge of SPI_SCLK
P_RX_DATA_SAMPLE : process(SPI_SCLK, SYS_RST)
begin
    if (SYS_RST = '1' ) then
        RX_DATA_REG <= (others => '0' );
    elsif SPI_SCLK'event and SPI_SCLK = '1' then
        RX_DATA_REG <= RX_DATA_REG(6 downto 0) & SPI_MOSI;
    end if;
end process P_RX_DATA_SAMPLE;

-- count rising edges on SPI_SCLK
P_RX_DATA_COUNT: process(SPI_SCLK, SYS_RST)
begin
    if (SYS_RST = '1' ) then
        RX_DATA_COUNT <= (others => '0' );
        RX_DONE <= '0' ;
    elsif SPI_SCLK'event and SPI_SCLK = '1' then
        if RX_DATA_COUNT = "111" then
            RX_DATA_COUNT <= (others => '0' );
            RX_DONE <= '1';
        else
            RX_DATA_COUNT <= RX_DATA_COUNT + 1;
            RX_DONE <= '0' ;
        end if;
    end if;
end process P_RX_DATA_COUNT;

P_RX_DONE_SYNC : process(SYS_CLK, SYS_RST)
begin
    if (SYS_RST = '1' ) then
        RX_DONE_REG1 <= '0' ;
        RX_DONE_REG2 <= '0' ;
        RX_DONE_REG3 <= '0' ;
    elsif SYS_CLK'event and SYS_CLK = '1' then
        RX_DONE_REG1 <= RX_DONE;
        RX_DONE_REG2 <= RX_DONE_REG1;
        RX_DONE_REG3 <= RX_DONE_REG2;
    end if;
end process P_RX_DONE_SYNC;
```

```
42
43 RX_DONE_SYNC <= RX_DONE_REG2 and not RX_DONE_REG3;
44
45 P_RX_DATA_VLD: process(SYS_CLK, SYS_RST)
46 begin
47     if (SYS_RST = '1' ) then
48         RX_DATA_VLD <= '0' ;
49     elsif SYS_CLK'event and SYS_CLK = '1' then
50         if RX_DONE_SYNC = '1' and CS_N = '0'  then
51             RX_DATA_VLD <= '1';
52         elsif RX_DATA_RD = '1' and CS_N = '0'  then
53             RX_DATA_VLD <= '0' ;
54         end if;
55     end if;
56 end process P_RX_DATA_VLD;
```

Listing 6.2: Implementation of SPI reception in slave mode.

In this section, the logic required for data reception via SPI is implemented, synchronizing the `MOSI` (Master Out Slave In) signal with the SPI clock (`SPI_SCLK`) and signaling when a complete byte has been received.

Sampling of the MOSI data:

The process `P_RX_DATA_SAMPLE` is triggered on the rising edge of the `SPI_SCLK` signal and on the reset signal `SYS_RST`. When an active reset is detected, the `RX_DATA_REG` register is cleared. On each rising edge of `SPI_SCLK`, the register performs a left shift concatenating the new bit present on the `SPI_MOSI` signal into the least significant bit. This mechanism enables the reconstruction, bit by bit, of the byte received serially.

Counting of the received bits:

The `P_RX_DATA_COUNT` process keeps track of the number of bits received by counting the rising edges of `SPI_SCLK`. Under reset conditions, the `RX_DATA_COUNT` counter is cleared and the `RX_DONE` signal is deactivated. At each rising edge, the counter increments until it reaches the binary value `"111"` (7 in decimal), indicating the reception of the eighth bit. When this condition occurs, the counter is reset and the `RX_DONE` signal is set to '1', signaling the completion of byte reception.

Synchronization of the `RX_DONE` signal with the system clock:

Since the `RX_DONE` signal is generated based on the SPI clock, which may be asynchronous with respect to the system clock `SYS_CLK`, a multi-stage synchronization circuit named `P_RX_DONE_SYNC` is implemented. This process transfers the `RX_DONE` signal through three registers synchronized to the system clock, in order to prevent metastability phenomena and

generate a synchronization pulse RX_DONE_SYNC that detects the rising edge of the reception completion signal.

Management of the data validity signal:

The P_RX_DATA_VLD process manages the RX_DATA_VLD signal, which indicates when the received data is valid and ready to be read. When RX_DONE_SYNC is active and the device is selected (CS_N = '0'), the RX_DATA_VLD signal is set to '1', indicating the availability of a complete byte in the register. Subsequently, when the microcontroller reads the data (RX_DATA_RD = '1' with CS_N = '0'), the RX_DATA_VLD signal is reset to '0'.

```
-- SPI slave transmitter ------

-- count falling edges on SPI_SCLK
P_TX_DATA_COUNT : process(SPI_SCLK, SYS_RST)
begin
    if (SYS_RST = '1' ) then
        TX_DATA_COUNT     <= "000";
        TX_DONE           <= '0' ;
    elsif SPI_SCLK'event and SPI_SCLK = '0'  then
        if TX_DATA_COUNT = "111" then
            TX_DATA_COUNT <= "000";
            TX_DONE       <= '1' ;
        else
            TX_DATA_COUNT <= TX_DATA_COUNT + 1;
            TX_DONE <= '0' ;
        end if;
    end if;
end process P_TX_DATA_COUNT;

-- data must be present on MISO before the first rising edge of SPI_SCLK
-- and must be updated at every falling edge of SPI_SCLK
SPI_MISO <= TX_DATA_REG(to_integer(7 - RX_DATA_COUNT));

P_TX_DONE_SYNC : process(SYS_CLK, SYS_RST)
begin
    if (SYS_RST = '1' ) then
        TX_DONE_REG1     <= '0' ;
        TX_DONE_REG2     <= '0' ;
        TX_DONE_REG3     <= '0' ;
    elsif SYS_CLK'event and SYS_CLK = '1' then
        TX_DONE_REG1 <= TX_DONE;
        TX_DONE_REG2 <= TX_DONE_REG1;
        TX_DONE_REG3 <= TX_DONE_REG2;
    end if;
end process P_TX_DONE_SYNC;

TX_DONE_SYNC <= TX_DONE_REG2 and not TX_DONE_REG3;

TX_READY_IN <= (not TX_DATA_REG_BUSY) or TX_DONE_SYNC;

P_TX_DATA_REG_BUSY : process (SYS_CLK, SYS_RST)
```

```
42 begin
43     if (SYS_RST = '1' ) then
44         TX_DATA_REG_BUSY <= '0' ;
45     elsif SYS_CLK'event and SYS_CLK = '1' then
46         if (TX_DATA_WR = '1' and TX_READY_IN = '1' and CS_N = '0' ) then
47             TX_DATA_REG_BUSY <= '1';
48         elsif (TX_DONE_SYNC = '1' or CS_N = '1' ) then
49             TX_DATA_REG_BUSY <= '0' ;
50         end if;
51     end if;
52 end process P_TX_DATA_REG_BUSY;
53
54 TX_READY <= TX_READY_IN;
```

Listing 6.3: SPI slave transmitter.

This part of the code implements the logic for transmitting data from the FPGA device to the SPI master (microcontroller), synchronizing the MISO (Master In Slave Out) signal with the SPI clock and indicating the transmission completion status.

Bit Transmission Counter:
The process `P_TX_DATA_COUNT` counts the falling edges of the `SPI_SCLK` signal. During reset, the `TX_DATA_COUNT` counter is cleared, and the `TX_DONE` signal is deactivated. At each falling edge, the counter increments until it reaches the value `"111"` (i.e., 7 in binary), indicating that 8 bits have been transmitted. At that point, the counter is reset and the `TX_DONE` signal is asserted (`'1'`) to indicate the completion of a byte transmission.

Management of the MISO Data Signal:
The `SPI_MISO` signal is assigned in such a way as to transmit the correct bit from the data register `TX_DATA_REG`. Specifically, the bit at index `7 - RX_DATA_COUNT` is transmitted, aligning correctly with the SPI clock sequence. This value must be present on the `SPI_MISO` line before the first rising edge of `SPI_SCLK` and must be updated at each falling edge to ensure proper timing of the transmitted data.

Synchronization of the TX_DONE Signal with the System Clock:
Since the `TX_DONE` signal is generated in the `SPI_SCLK` clock domain, which is asynchronous with respect to the system clock `SYS_CLK`, synchronization is required. For this purpose, a three-stage synchronizer circuit (`P_TX_DONE_SYNC`) is used to prevent metastability issues. The synchronized signal `TX_DONE_SYNC` is used to detect the rising edge of transmission completion in the system clock domain.

Management of Data Register Availability for Writing:
The TX_READY_IN signal indicates when the transmission data register is ready to receive a new byte. It is active when the register is not busy (TX_DATA_REG_BUSY = '0') or when the previous transmission has completed (TX_DONE_SYNC = '1'). The process P_TX_DATA_REG_BUSY updates this state: it sets the register as busy ('1') when a valid write occurs (TX_DATA_WR = '1', with CS_N = '0' and register ready), and clears it ('0') at the end of transmission or if the device is deselected.

External Signaling of Data Readiness:
The output signal TX_READY reflects the internal state of TX_READY_IN and informs the SPI master of the availability of the device to receive a new byte for transmission.

6.2.2 CONTROLLER

The *CONTROLLER* module is responsible for orchestrating the execution of requested operations, interpreting data received via SPI, and translating it into operational commands for the motor. It manages the control flow to the *MOTOR* module, enabling or disabling motion, setting speed parameters, and coordinating the start, execution and stop phases of the motor profile. The logic of this module includes a state machine that ensures correct synchronization and sequencing of operations, guaranteeing that each phase of the motion is completed properly before transitioning to the next.

```
P_RAM_ACCESS: process(CLK, RST)
begin
  if RST = '1' then
    RAM_ACCESS_STATE <= IDLE;
    RAM_ACCESS_BUSY <= '0' ;
    RAM_ACCESS_DONE <= '0' ;
    CTRL_RAM_IN <= (others => '0' );
    CTRL_RAM_ADDR <= (others => '0' );
    CTRL_RAM_WR_EN <= '0' ;
  elsif CLK'event and CLK = '1' then

    case RAM_ACCESS_STATE is

      when IDLE =>

        CTRL_RAM_WR_EN  <= '0' ;
        RAM_ACCESS_DONE <= '0' ;
        RAM_ACCESS_BUSY <= '0' ;
        if WRITE_REG = '1' then
          RAM_ACCESS_BUSY <= '1';
          CTRL_RAM_ADDR   <= REG_ADDRESS;
          CTRL_RAM_IN     <= REG_DATA_WR;
          CTRL_RAM_WR_EN  <= '1';
```

```
          RAM_ACCESS_STATE <= WR_REG;
        end if;

      when WR_REG =>

        CTRL_RAM_WR_EN  <= '0' ;
        RAM_ACCESS_BUSY <= '0' ;
        RAM_ACCESS_DONE <= '1';
        RAM_ACCESS_STATE <= IDLE;

    end case;
  end if;
end process P_RAM_ACCESS;
```

Listing 6.4: Process for managing RAM write operations.

RAM Access:

The `P_RAM_ACCESS` process manages write access to the control RAM. Implements a state machine with two main states: `IDLE` and `WR_REG`. When a write request is activated (`WRITE_REG = '1'`), the process loads the address and data to be written into the RAM control signals, enables the write signal for one clock cycle, and transitions to the write state. Once the operation is complete, it signals completion through the `RAM_ACCESS_DONE` signal and returns to the idle state. The signals `RAM_ACCESS_BUSY` and `RAM_ACCESS_DONE` respectively indicate ongoing activity and completion of the access, ensuring proper synchronization of the write operation.

```
  P_CONTROL: process(CLK, RST)
  begin
    if RST = '1' then
      All signals set to starting conditions
    elsif CLK'event and CLK = '1' then
        case CTRL_STATE is
          when IDLE =>
            All signals set to starting conditions
              CTRL_STATE <= CMD_RX;
            end if;

          when CMD_RX =>
            SPI_TX_DATA <= X"FF";
            SPI_TX_DATA_WR <= '1';
            Counters set to 0
            if SPI_CS_N = '1' then
              CTRL_STATE <= IDLE;
            elsif SPI_RX_DATA_VLD = '1' then
              SPI_TX_DATA_WR <= '0' ;
              CMD <= SPI_RX_DATA;
              SPI_RX_DATA_RD_INT <= '1';
              CTRL_STATE <= CMD_PARSE;
            end if;

          when CMD_PARSE =>
```

```
26
27              SPI_RX_DATA_RD_INT <= '0' ;
28              if SPI_CS_N = '1' then
29                CTRL_STATE <= IDLE;
30              else
31                case to_integer(unsigned(CMD)) is
32                  when 49 to 54 =>  -- 0x31 write RAM_CONFIG_M1 - 0x36 write RAM_CONFIG_M6
33                    COUNT <= (others => '0' );
34                    TOT_BYTES <= to_unsigned(256, TOT_BYTES'length);
35                    SPI_TX_DATA <= X"FF";
36                    SPI_TX_DATA_WR <= '1';
37                    REG_INDEX <= (others => '0' );
38                    CTRL_RAM_SEL_INT <= std_logic_vector(to_unsigned(to_integer(unsigned(CMD)) -
                          49, CTRL_RAM_SEL_INT'length));
39                    CTRL_STATE <= STORE_RAM;
40                    if RUN_INT = '1' then
41                      CTRL_STATE <= CMD_IGNORE;
42                    end if;
43
44                  when 113 => -- 0x71 write RAM_RUN
45                    COUNT <= (others => '0' );
46                    TOT_BYTES <= to_unsigned(64, TOT_BYTES'length);
47                    SPI_TX_DATA <= X"FF";
48                    SPI_TX_DATA_WR <= '1';
49                    REG_INDEX <= (others => '0' );
50                    CTRL_RAM_SEL_INT <= "10" & RUN_RAM_TO_WRITE;
51                    LOADING_RUN <= '1';
52                    CTRL_STATE <= STORE_RAM;
53                    if RUN_INT = '0'  or RUN_RAM_FULL = '1' then
54                      LOADING_RUN <= '0' ;
55                      CTRL_STATE <= CMD_IGNORE;
56                    end if;
57
58                  when 46  => -- 0x2E enter CONFIG state
59                    RUN_INT <= '0' ;
60                    if SPI_CS_N = '1' then
61                      CTRL_STATE <= IDLE;
62                    end if;
63
64                  when 47  => -- 0x2F enter RUN state
65                    RUN_INT <= '1';
66                    if SPI_CS_N = '1' then
67                      CTRL_STATE <= IDLE;
68                    end if;
69
70                  when 144  => -- 0x90 get STATUS
71                    SPI_TX_DATA_WR <= '0' ;
72                    CTRL_STATE <= GET_STATUS;
73
74                  when others =>
75                    CTRL_STATE <= IDLE;
76                end case;
77              end if;
78
79            when STORE_RAM =>
```

```
80
81              SPI_RX_DATA_RD_INT <= '0' ;
82              if SPI_CS_N = '1' then
83                COUNT <= (others => '0' );
84                CTRL_STATE <= IDLE;
85              elsif RAM_ACCESS_BUSY = '1' then
86                WRITE_REG <= '0' ;
87              elsif RAM_ACCESS_DONE = '1' then
88                if COUNT = TOT_BYTES then
89                  COUNT <= (others => '0' );
90                  CTRL_STATE <= IDLE;
91                end if;
92              elsif SPI_RX_DATA_VLD = '1' and SPI_RX_DATA_RD_INT = '0'  then
93                if COUNT(0) = '0'  then
94                  REG_DATA_WR(15 downto 8) <= SPI_RX_DATA;
95                else
96                  REG_DATA_WR(7 downto 0) <= SPI_RX_DATA;
97                  REG_ADDRESS <= REG_INDEX;
98                  REG_INDEX <= REG_INDEX + 1;
99                  WRITE_REG <= '1';
100               end if;
101               -- acknowledge received data
102               SPI_RX_DATA_RD_INT <= '1';
103               COUNT <= COUNT + 1;
104             end if;
105
106           when CMD_IGNORE =>
107
108             SPI_RX_DATA_RD_INT <= '0' ;
109             SPI_TX_DATA <= X"FF";
110             SPI_TX_DATA_WR <= '1';
111             if SPI_CS_N = '1' then
112               COUNT <= (others => '0' );
113               CTRL_STATE <= IDLE;
114             elsif SPI_RX_DATA_VLD = '1' and SPI_RX_DATA_RD_INT = '0'  then
115               -- acknowledge received data
116               SPI_RX_DATA_RD_INT <= '1';
117             end if;
118
119           when GET_STATUS =>
120
121             if SPI_CS_N = '1' then
122               COUNT <= (others => '0' );
123               CTRL_STATE <= IDLE;
124             elsif SPI_TX_READY = '1' and SPI_TX_DATA_LOADED = '0'  then
125               --SPI_TX_DATA <= STATUS;
126               SPI_TX_DATA <= "00" & RUN_RAM_STATUS & "0" & LIMIT_SWITCHES;
127               SPI_TX_DATA_WR <= '1';
128               SPI_TX_DATA_LOADED <= '1';
129             end if;
130         end case;
131     end if;
132   end process P_CONTROL;
```

Listing 6.5: Process for command management.

SPI Control and Command Handling:

The P_CONTROL process manages SPI communication for system control, implementing a finite state machine with several functional states: IDLE, CMD_RX, CMD_PARSE, STORE_RAM, CMD_IGNORE, and GET_STATUS.

At startup or reset, it initializes control signals and internal variables. In the IDLE state, it waits for the activation of the SPI chip select line (SPI_CS_N = '0') and begins transmitting a default value (0xFF) to indicate availability. It then transitions to the CMD_RX state to receive the SPI command.

In CMD_RX, it receives the command byte from SPI when available (SPI_RX_DATA_VLD = '1') and stores it in the CMD register. If the chip select line is deactivated, it returns to IDLE. Otherwise, it proceeds to the CMD_PARSE state to interpret the received command.

In the CMD_PARSE state, it analyzes the received command value and selects the appropriate action:

- Commands from 0x31 to 0x36 (decimal 49 to 54) trigger the writing of speed (maximum and minimum), acceleration, or deceleration values into the configuration RAMs of the six motors, respectively. For each command, the selector CTRL_RAM_SEL_INT is set to correctly address the corresponding RAM.
- Command 0x71 (decimal 113) enables the writing of the number of steps to be executed into the execution RAM RAM_RUN, provided that the system is in the RUN state and the RAM is not full.
- Commands 0x2E (46) and 0x2F (47) manage entry into the CONFIG and RUN states, enabling or disabling the RUN_INT signal.
- Command 0x90 (144) requests the transmission of the system status, transitioning to the GET_STATUS state.
- Other commands cause the process to return to IDLE.

In STORE_RAM, the process handles the sequential writing of data received via SPI into the selected RAM, using handshake signals with the RAM access process. Data are written as 16-bit values (two consecutive bytes) to incremental addresses, and the write operation is confirmed with the WRITE_REG signal asserted. At the end of the transfer or upon chip select deactivation, the process returns to IDLE.

The CMD_IGNORE state is used to disregard SPI data when the system is not ready to process them, while still providing acknowledgement of reception.

In GET_STATUS, a status byte containing information about the execution RAM and limit switches is transmitted until the chip select line is deactivated.

The process also updates various external signals such as SPI_RX_DATA_RD, RUN, and CTRL_RAM_SEL to correctly synchronize write operations to the control RAM and manage the operational state of the system.

6.2.3 RAM

The RAM module provides an internal memory area within the FPGA where operational data received from the CONTROLLER module are temporarily stored. This design choice significantly reduces the use of dedicated registers, delegating to the RAM the task of storing configuration, command, or parameter information destined for other subsystems.
The memory is structured as a dual-port synchronous RAM, capable of independently supporting read and write operations, each synchronized on its own clock signal. In this way, the CONTROLLER can write new data without interfering with reading operations by downstream modules, such as MOTOR or other motor control blocks.
Subsequent modules access the stored data directly from this RAM, using the assigned addresses to retrieve them and using them as references for speed profile execution or other motion control logics. The use of RAM as a central data exchange element thus represents an efficient, flexible, and scalable solution for system configuration management.

```
process (WCLK) -- Write Memory.
  begin
    if (WCLK'event and WCLK = '1' ) then
      if (WRITE_EN = '1' ) then
        MEM(to_integer(WADDR)) <= DIN;
      end if;
    end if;
  end process;

  process (RCLK) -- Read Memory.
  begin
    if (RCLK'event and RCLK = '1' ) then
      DOUT <= MEM(to_integer(RADDR)); -- Using read address bus.
    end if;
  end process;
```

Listing 6.6: Processes of RAM Read and Write.

RAM Write Process:
This process is sensitive to the clock signal WCLK and handles data writing into the RAM. On the rising edge of the clock (WCLK'event and WCLK = '1'), the process checks whether the write enable signal WRITE_EN is active ('1'). If so, the value present on the input data bus DIN is stored in the memory location specified by the address WADDR.
This mechanism allows the system to store information originating from external modules (such

as the *CONTROLLER*), using RAM as a temporary storage area without employing registers, thereby reducing the usage of logic resources within the FPGA.

RAM Read Process:
The second process is sensitive to the clock signal `RCLK` and manages the RAM read phase. On the rising edge of `RCLK`, the data located at the address `RADDR` is read from memory and assigned to the output `DOUT`.
This logic allows downstream modules to synchronously access data previously stored in RAM, enabling the propagation of information within the system according to the expected timing. Separate read and write access, each with its own clock, enables independent and parallel operation of the two channels.

6.2.4 MOTOR

The `MOTOR` module is responsible for generating and controlling the stepper motor's motion profile. It receives as input the fundamental operating parameters, such as initial speed, maximum speed, acceleration, deceleration, and the total number of steps to execute, provided by the `CONTROLLER` module.
Based on these parameters, it implements a speed profile that can assume either a trapezoidal or triangular shape, typical in stepper motor control systems to ensure smooth start-up and stopping. Specifically, the profile defines three distinct phases: progressive acceleration up to cruising speed, maintenance of constant speed, and finally deceleration until the motor comes to a complete stop.
The module dynamically calculates the time interval between step signals, thereby generating properly timed PWM pulses to drive the motor precisely and smoothly. This logic optimizes movement, minimizing vibrations and mechanical wear, while ensuring compliance with the time and speed specifications required by the application.
To optimize resource usage within the FPGA, instead of performing real-time calculations to determine the appropriate values for frequency variation intervals or step pulse durations (as suggested by the application note in the previous chapter), a precomputed table approach has been adopted. These values are calculated in advance and provided by the microcontroller directly to the FPGA, thereby simplifying the control logic and reducing the computational load on the programmable device.
The internal structure of the module includes a state machine that governs the transitions between the different phases of the profile, as well as counters and timers that manage the emission of pulses based on the calculated frequency or values read from the tables. It also provides status signals useful for monitoring motion progress and synchronizing with other system blocks.

```
P_RAM_ACCESS: process(CLK, RST)
begin
    if RST = '1' then
        RAM_ACCESS_STATE <= IDLE;
        RAM_ACCESS_BUSY <= '0' ;
        RAM_ACCESS_DONE <= '0' ;
        RAM_ADDR <= (others => '0' );
        REG_DATA_RD <= (others => '0' );
    elsif CLK'event and CLK = '1' then
        case RAM_ACCESS_STATE is

            when IDLE =>
                RAM_ACCESS_DONE <= '0' ;
                RAM_ACCESS_BUSY <= '0' ;
                if READ_REG = '1' then
                    RAM_ACCESS_BUSY   <= '1';
                    RAM_ACCESS_DONE   <= '0' ;
                    RAM_ADDR          <= REG_ADDRESS;
                    RAM_ACCESS_STATE  <= RD_REG_EN;
                end if;

            when RD_REG_EN =>
                RAM_ACCESS_BUSY <= '1';
                RAM_ACCESS_STATE <= RD_REG;

            when RD_REG =>
                REG_DATA_RD    <= RAM_OUT;
                RAM_ACCESS_BUSY <= '0' ;
                RAM_ACCESS_DONE <= '1';
                RAM_ACCESS_STATE <= IDLE;
        end case;
    end if;
end process P_RAM_ACCESS;
```

Listing 6.7: Process of RAM Access.

RAM Access Management:

The `P_RAM_ACCESS` process implements a state machine that manages reading from the internal RAM. During the reset condition, the state machine is set to the `IDLE` state, and the `RAM_ACCESS_BUSY` and `RAM_ACCESS_DONE` signals are cleared. Additionally, the read address `RAM_ADDR` and the data register `REG_DATA_RD` are reset to zero.

Under normal operating conditions, the state machine behaves as follows: it initially resides in the `IDLE` state, where it remains inactive and waits for a read request indicated by the `READ_REG` signal. When such a request is detected, the process asserts the busy signal by setting it to `'1'`, loads the address of the register to be read into the `RAM_ADDR` register, and then transitions to the next state `RD_REG_EN`.

In the `RD_REG_EN` state, the machine keeps the busy signal active to indicate that access is ongoing and prepares for the actual data read. In the next clock cycle, it transitions to the `RD_REG` state,

where the data present on the RAM output (RAM_OUT) is acquired and stored in the REG_DATA_RD register. At the same time, the busy signal is deasserted and the done signal (RAM_ACCESS_DONE) is asserted to indicate the completion of the operation. At the end of this state, the machine returns to the IDLE state, waiting for new requests.

```
P_ACC: process(CLK, RST)
  begin
    if RST = '1' then
      All signals set to starting conditions;
    elsif CLK'event and CLK = '1' then
      case MOTOR_STATE is

        when IDLE =>

          MAX_SPEED_REACHED <= '0' ;
          DECEL_STARTED <= '0' ;
          REG_INDEX <= (others => '0' );
          EN_OUT <= '0' ;;
          if MOTOR_ENABLE = '1' then
            if END_MOVE_INT = '0'  then
              EN_OUT <= '1';
              REG_ADDRESS <= REG_INDEX;
              READ_REG <= '1';
              MOTOR_STATE <= READ_CONFIG;
            end if;
          else
            END_MOVE_INT <= '0' ;
          end if;

        when READ_CONFIG =>

          if RAM_ACCESS_BUSY = '1' then
            READ_REG <= '0' ;
          elsif RAM_ACCESS_DONE = '1' then
            READ_REG_VALID <= '1';
          elsif READ_REG_VALID = '1' then
            READ_REG_VALID <= '0' ;
            if REG_ADDRESS(0) = '0'  then
              CYCLES_COUNT <= unsigned(REG_DATA_RD);
              REG_ADDRESS  <= REG_INDEX + 1;
              READ_REG    <= '1';
            else
              STEP_PERIOD      <= unsigned(REG_DATA_RD);
              STEP_PERIOD_COUNT <= unsigned(REG_DATA_RD);
              if CYCLES_COUNT = to_unsigned(0, CYCLES_COUNT'length) then
                MAX_SPEED_REACHED <= '1';
              end if;
              MOTOR_STATE <= MOVE;
            end if;
          end if;

        when MOVE =>

```

```
49          if CLK_EN = '1' then
50            if MOTOR_REGS(0) = to_unsigned(0, MOTOR_REGS(0)'length) or LIMIT_SWITCH = '1' or
                  MOTOR_ENABLE = '0' then
51              --end of movement
52              END_MOVE_INT <= '1';
53              EN_OUT <= '0' ;
54              MOTOR_STATE <= IDLE;
55            else
56              --step output generation
57              STEP_PERIOD_COUNT <= STEP_PERIOD_COUNT - 1;
58              if STEP_PERIOD_COUNT = to_unsigned(0, STEP_PERIOD_COUNT'length) then
59                STEP_INT <= not STEP_INT;
60                STEP_PERIOD_COUNT <= STEP_PERIOD - 1;
61                if STEP_INT = '0' then
62                  MOTOR_REGS(0) <= MOTOR_REGS(0) - 1;
63                end if;
64              end if;
65              if MAX_SPEED_REACHED = '0' and DECEL_STARTED = '0' then
66                CYCLES_COUNT <= CYCLES_COUNT - 1;
67                if CYCLES_COUNT = to_unsigned(0, CYCLES_COUNT'length) then
68                  REG_INDEX   <= REG_INDEX + 2;
69                  REG_ADDRESS <= REG_INDEX + 2;
70                  READ_REG <= '1';
71                  MOTOR_STATE <= READ_CONFIG;
72                end if;
73              elsif DECEL_STARTED = '1' then
74                CYCLES_COUNT <= CYCLES_COUNT - 1;
75                if CYCLES_COUNT = to_unsigned(0, CYCLES_COUNT'length) then
76                  if REG_INDEX >= to_unsigned(2, REG_INDEX'length) then
77                    REG_INDEX   <= REG_INDEX - 2;
78                    REG_ADDRESS <= REG_INDEX - 2;
79                    READ_REG <= '1';
80                    MOTOR_STATE <= READ_CONFIG;
81                  else
82                    CYCLES_COUNT <= (others => '0' );
83                  end if;
84                end if;
85              end if;
86
87              if MOTOR_REGS(0) <= MOTOR_REGS(1) and DECEL_STARTED = '0' then
88                if REG_INDEX >= to_unsigned(2, REG_INDEX'length) then
89                  REG_INDEX <= REG_INDEX - 2;
90                  REG_ADDRESS <= REG_INDEX -2;
91                end if;
92                READ_REG <= '1';
93                MOTOR_STATE <= READ_CONFIG;
94                DECEL_STARTED <= '1';
95              end if;
96            end if;
97          end if;
98        end case;
99      end if;
100   end process P_ACC;
```

Listing 6.8: Motor movement process.

Motor State Management:
The P_ACC process implements a finite state machine for the control of the stepper motor. At reset, all signals are initialized to their default starting conditions. When the clock is active, the process evaluates the current MOTOR_STATE and behaves accordingly based on its value.

In the IDLE state, the flags MAX_SPEED_REACHED and DECEL_STARTED are cleared, and the register index REG_INDEX is reset. The output enable signal EN_OUT is deactivated. If the motor is enabled via MOTOR_ENABLE and movement end is not signaled (END_MOVE_INT = '0'), then EN_OUT is activated, the register address REG_ADDRESS is set, and a read operation is initiated using READ_REG. The state then transitions to READ_CONFIG. If the motor is not enabled, END_MOVE_INT is cleared.

In the READ_CONFIG state, the process waits for the RAM read operation to complete. While RAM_ACCESS_BUSY is high, the READ_REG signal is deactivated. Once the read operation is finished (RAM_ACCESS_DONE = '1'), the READ_REG_VALID signal is asserted to confirm data validity. Upon validation, the least significant bit of the address is checked: if it is zero, the read data is stored in CYCLES_COUNT, and the next register is read by incrementing REG_ADDRESS. If it is one, the data is stored in STEP_PERIOD and STEP_PERIOD_COUNT; and if CYCLES_COUNT is zero, it indicates that the maximum speed has been reached, after which the state transitions to MOVE.

In the MOVE state, on each activation of the internal clock CLK_EN, the process checks whether the step counter MOTOR_REGS(0) is zero, the limit switch LIMIT_SWITCH is active, or the motor has been disabled. In any of these cases, movement is considered complete, END_MOVE_INT is asserted, EN_OUT is deactivated, and the state returns to IDLE.
Otherwise, a step pulse is generated by decrementing STEP_PERIOD_COUNT until it reaches zero, at which point STEP_INT is toggled, the counter is reloaded, and on the falling edge of the signal, the step counter is decremented.
If the maximum speed has not been reached and deceleration has not started, CYCLES_COUNT is decremented. Upon expiration, REG_INDEX is incremented, REG_ADDRESS is updated, and a new read operation is requested, returning to the READ_CONFIG state. If deceleration has started, CYCLES_COUNT is decremented, and upon reaching zero, it is checked whether REG_INDEX can be decremented to read previous registers; if so, REG_INDEX and REG_ADDRESS are updated accordingly, otherwise CYCLES_COUNT is cleared.
Finally, if the step counter reaches or drops below the comparison value MOTOR_REGS and deceleration has not yet begun, deceleration is initiated by updating the address to read the previous registers, activating READ_REG, and transitioning to the READ_CONFIG state.

6.2.5 MOTORS_CONTROL_UNIT

This block fully manages the control of stepper motors, coordinating access to RAM for loading motion parameters and supervising the phases of start-up, execution, and completion of command sequences.

A finite state machine governs the operations of data reading from RAM, updating of internal registers, motor enablement, and monitoring of movement completion. Interaction with memory is handled by a dedicated process that synchronizes read operations, managing *BUSY* and *DONE* signals to ensure correct data acquisition.

Step counts and direction parameters are thus loaded into internal registers, ready to be used by the motor control system. At the same time, control logic detects signals from the microcontroller, identifying when new data has been loaded and orchestrating the transition between operating phases: preparation, execution, and completion verification.

The management of RAM banks enables efficient and synchronized use of resources, ensuring that updates and sequences are executed without conflict. This structure allows for precise and reliable motor control, ensuring that motion parameters are always available and correctly applied in real time.

```
P_RAM_ACCESS: process(CLK, RST)
begin
    if RST = '1' then
        RAM_ACCESS_STATE <= IDLE;
        RAM_ACCESS_BUSY <= '0' ;
        RAM_ACCESS_DONE <= '0' ;
        RAM_ADDR <= (others => '0' );
        REG_DATA_RD <= (others => '0' );
    elsif CLK'event and CLK = '1' then
        case RAM_ACCESS_STATE is

            when IDLE =>
                RAM_ACCESS_DONE <= '0' ;
                RAM_ACCESS_BUSY <= '0' ;
                if READ_REG = '1' then
                    RAM_ACCESS_BUSY   <= '1';
                    RAM_ACCESS_DONE   <= '0' ;
                    RAM_ADDR          <= REG_ADDRESS;
                    RAM_ACCESS_STATE  <= RD_REG_EN;
                end if;

            when RD_REG_EN =>
                RAM_ACCESS_BUSY <= '1';
                RAM_ACCESS_STATE <= RD_REG;

            when RD_REG =>
                REG_DATA_RD    <= RAM_OUT;
                RAM_ACCESS_BUSY <= '0' ;
                RAM_ACCESS_DONE <= '1';
```

```
                RAM_ACCESS_STATE <= IDLE;
        end case;
    end if;
end process P_RAM_ACCESS;

P_LOADING_RUN_D : process (CLK, RST)
  begin
    if RST = '1' then
      LOADING_RUN_D <= '0' ;
    elsif (rising_edge(CLK)) then
      LOADING_RUN_D <= LOADING_RUN;
    end if;
  end process P_LOADING_RUN_D;

  -- Falling edge of busy signal
  LOADING_RUN_FE <= '1' when LOADING_RUN = '0'  and LOADING_RUN_D = '1' else '0' ;
```

Listing 6.9: Processes for RAM Access and motion preparation.

RAM Access:

The P_RAM_ACCESS process manages read access to the RAM through a simple state machine. When a read signal (READ_REG) is requested, it transitions from the idle state (IDLE) to a sequence that enables the read operation and waits for the data from the RAM. Once the value has been acquired, it is stored in the REG_DATA_RD register, the end of the operation is signaled (RAM_ACCESS_DONE), and the process returns to its initial state. This process coordinates the control and timing signals required to ensure correct and synchronized access to memory data.

Synchronization of the LOADING_RUN Signal:

The P_LOADING_RUN_D process is responsible for synchronizing the LOADING_RUN signal with respect to the system clock, storing its previous value in the LOADING_RUN_D register at each rising edge of the clock. This dual representation allows the generation of a falling edge signal, LOADING_RUN_FE, which is active for a single clock cycle when LOADING_RUN transitions from '1' to '0'. This signal is used to reliably detect the exact moment the data loading process ends, facilitating synchronization and control of the subsequent system phases.

```
P_MOTORS_CU: process(CLK, RST)
begin
  if RST = '1' then
    All signals set to starting conditions;
  elsif CLK'event and CLK = '1' then

    case MOTORS_CU_STATE is

      when IDLE =>

```

```
11          REG_INDEX <= (others => '0' );
12          MOTORS_EN <= '0' ;
13          MOTOR_DATA_LOADED <= '0' ;
14
15          if LOADING_RUN_FE = '1' then
16            --controller wrote inactive RUN_RAM bank
17            RUN_RAM_TO_WRITE_INT <= not RUN_RAM_TO_WRITE_INT;
18            if RUN_RAM_EMPTY_INT = '0'  then
19              RUN_RAM_FULL_INT <= '1';
20            end if;
21            RUN_RAM_EMPTY_INT <= '0' ;
22          end if;
23
24          if RUN = '1' and RUN_RAM_EMPTY_INT = '0'  then
25            --load motors counters
26            MOTOR_SEL_INT   <= (others => '0' );
27            REG_ADDRESS <= REG_INDEX;
28            READ_REG <= '1';
29            MOTORS_CU_STATE <= LOAD_COUNTERS;
30          end if;
31
32        when LOAD_COUNTERS =>
33
34          MOTORS_EN <= '0' ;
35
36          if LOADING_RUN_FE = '1' then
37            --controller wrote inactive RUN_RAM bank
38            RUN_RAM_TO_WRITE_INT <= not RUN_RAM_TO_WRITE_INT;
39            if RUN_RAM_EMPTY_INT = '0'  then
40              RUN_RAM_FULL_INT <= '1';
41            end if;
42            RUN_RAM_EMPTY_INT <= '0' ;
43          end if;
44
45          MOTOR_DATA_LOAD <= '0' ;
46          if RUN = '0'  then
47            MOTORS_EN <= '0' ;
48            MOTORS_CU_STATE <= IDLE;
49            RUN_RAM_EMPTY_INT <= '1';
50            RUN_RAM_FULL_INT <= '0' ;
51            RUN_RAM_TO_WRITE_INT <= '0' ;
52          else
53            if RAM_ACCESS_BUSY = '1' then
54              READ_REG <= '0' ;
55            elsif RAM_ACCESS_DONE = '1' then
56              READ_REG_VALID <= '1';
57            elsif READ_REG_VALID = '1' and MOTOR_DATA_LOADED = '0'  then
58              READ_REG_VALID    <= '0' ;
59              MOTOR_DATA_LOADED   <= '1';
60
61              if REG_INDEX(0) = '0'  then
62                MOTOR_DATA_OUT(19 downto 16) <= REG_DATA_RD(3 downto 0);
63                MOTOR_DIR <= REG_DATA_RD(4);
64              else
65                MOTOR_DATA_OUT(15 downto 0)  <= REG_DATA_RD;
```

```
66              MOTOR_REG_SEL <= REG_INDEX(1);
67              MOTOR_DATA_LOAD <= '1';
68            end if;
69
70            if REG_INDEX(2 downto 0) = "000" then
71              MOTOR_SEL_INT   <= MOTOR_SEL_INT + 1;
72              if MOTOR_SEL_INT = MOTORS_COUNT then
73                MOTOR_SEL_INT   <= (others => '0' );
74                MOTORS_EN <= '1';
75                MOTOR_DATA_LOADED <= '0' ;
76                REG_INDEX <= (others => '0' );
77                MOTORS_CU_STATE <= WAIT_MOVE;
78              end if;
79            end if;
80
81            REG_INDEX <= REG_INDEX + 1;
82          elsif MOTOR_DATA_LOADED = '1' then
83            MOTOR_DATA_LOADED <= '0' ;
84            REG_ADDRESS <= REG_INDEX;
85            READ_REG <= '1';
86          end if;
87        end if;
88
89      when WAIT_MOVE =>
90
91        if LOADING_RUN_FE = '1' then
92          --controller wrote inactive RUN_RAM bank
93          RUN_RAM_TO_WRITE_INT <= not RUN_RAM_TO_WRITE_INT;
94          if RUN_RAM_EMPTY_INT = '0'  then
95            RUN_RAM_FULL_INT <= '1';
96          end if;
97          RUN_RAM_EMPTY_INT <= '0' ;
98        end if;
99
100       MOTOR_DATA_LOAD <= '0' ;
101       if RUN = '0'  then
102         --to do: exit from RUN state while loading motor registers
103         MOTORS_EN <= '0' ;
104         MOTORS_CU_STATE <= IDLE;
105         RUN_RAM_EMPTY_INT <= '1';
106         RUN_RAM_FULL_INT <= '0' ;
107         RUN_RAM_TO_WRITE_INT <= '0' ;
108       elsif MOTORS_END = "111" then
109         --all motors end the movement
110         MOTORS_EN <= '0' ;
111         if RUN_RAM_FULL_INT = '1' or LOADING_RUN_FE = '1' then
112           --load motors counters
113           MOTOR_SEL_INT   <= (others => '0' );
114           REG_INDEX <= (others => '0' );
115           REG_ADDRESS <= (others => '0' );
116           READ_REG <= '1';
117           MOTORS_CU_STATE <= LOAD_COUNTERS;
118           RUN_RAM_FULL_INT <= '0' ;
119         else
120           RUN_RAM_EMPTY_INT <= '1';
```

```
121             MOTORS_CU_STATE <= IDLE;
122           end if;
123         end if;
124     end case;
125   end if;
126 end process P_MOTORS_CU;
```

Listing 6.10: Motion control unit.

Sequential Management of Data Loading and Motor Control: The process implements a finite state machine composed of three main states: `IDLE`, `LOAD_COUNTERS`, and `WAIT_MOVE`, which coordinate the loading of motor control data from RAM, motor activation, and monitoring of motion completion. In the `IDLE` state, the system waits for the execution signal (`RUN`) and checks for valid data in RAM, also managing any bank switching in the event of new write operations from the controller. Once the loading is triggered, the machine transitions to the `LOAD_COUNTERS` state, in which it sequentially reads the RAM registers, combining the data to reconstruct the necessary parameters for each motor (direction, counters, and register selection). The machine manages read requests and waits for BUSY and DONE signals from the RAM subsystem, synchronizing data acquisition with the correct timing. Upon completion of loading for all motors, it enables their operation and transitions to the `WAIT_MOVE` state, where it monitors the end-of-motion status of all motors simultaneously. If all motors have completed their movements or a new load request is received, the process updates the RAM status flags and, depending on the condition, either restarts the loading sequence or returns to the `IDLE` waiting state. This cyclic control ensures synchronization between data management and motor activity, maintaining a continuous and orderly flow of control operations.

6.3 Critical Areas, Optimizations, and Reuse

During the development of the project, numerous challenges and critical issues emerged, primarily due to the complexity of managing real-time simultaneous control of six stepper motors, each with independent and configurable motion profiles. In particular, one of the critical nodes was the implementation of the calculations necessary for the precise management of timing parameters, such as the duration of step pulses, the intervals between frequency changes, and the coordination of the acceleration, constant speed, and deceleration phases.
Implementing such calculations in real-time would have required a significant amount of logical and timing resources, as the computational processes would heavily engage the *lookup tables* (LUTs) and registers, potentially negatively impacting the achievable clock frequency and overall power consumption. Moreover, the intensive use of registers to store command and configuration data risked quickly saturating flip-flop and LUT resources, jeopardizing the possibility

of integrating all required functionalities within the selected FPGA.

To overcome these limitations, a decision was made to adopt an approach based on the use of **precomputed tables**, stored in *internal RAM blocks* (the red blocks in **Figure 6.1.2**). These tables contain predefined values indicating how often to update the frequency, the duration of the step signals, and the transition points between different motion phases. Access to this data occurs rapidly and synchronously, with minimal impact on logical resources, since BRAMs are optimized for storing large amounts of data with parallel access. This approach allowed offloading the burden of calculations from combinational logic, improving system stability and operational speed.

In addition to optimized calculation management, another significant source of resource utilization was represented by the **registers** dedicated to storing commands, particularly those used for diagnostic and debugging functions. During the development phases, numerous commands were implemented to query the current system state, read the configurations of each motor, or monitor limit switch signals, useful for ensuring operational safety and reliability. Although fundamental for validation, these registers occupied many LUTs and flip-flops, pushing logic utilization beyond 100% in some preliminary simulations.

To contain resource usage, many of these debug commands were progressively removed in the final version of the project, once verification and functional testing were completed. Thanks to this targeted reduction, combined with the use of tables and BRAMs, a significant decrease in resource utilization was achieved, bringing LUT usage to approximately 80% and PLB blocks to 97%, thus remaining within the technical limits imposed by the FPGA.

It is important to highlight that, although the adoption of precomputed tables imposes some rigidity in the system, this limitation is largely compensated by the nature of the application. Indeed, what truly affects motor control quality is the **accuracy** of the movement and its smoothness, rather than the absolute mathematical precision of frequencies. A variation of a few hertz in step frequencies does not compromise functionality or movement quality in any way, thereby ensuring an excellent compromise between resource optimization and operational performance.

Alongside these technical choices, the project was developed with strong attention to **modularity** and reuse, strategic elements fundamental in industrial environments. Each system module, from SPI communication management to single motor control, was designed as an independent block with a well-defined and documented interface. This subdivision facilitated development and testing work, allowing each component to be individually verified in isolation before integration into the complete system.

Modularity also promoted greater clarity in the code and project structure, simplifying problem identification and future maintenance. Furthermore, thanks to the extensive use of *configurable parameters*, the project proved highly flexible and easily adaptable to different hardware con-

figurations or application specifications.

This flexibility was an explicit requirement from the commissioning company, which desired not only a functioning device for the current application but also a scalable and reusable platform for future products and applications still under development. The presence of configurable parameters such as the number of controlled motors, maximum operating frequencies, or motion profiles allows rapid adaptation of the system to new needs without rewriting the VHDL code from scratch.

Ultimately, the project achieved an optimal balance between resource optimization, operational performance, and future evolvability, thanks to careful modular design, efficient use of BRAM, and conscious management of debugging features. This approach allowed meeting the precision, reliability, and scalability requirements, laying a solid foundation for the implementation of increasingly advanced and adaptable stepper motor control systems.

Chapter 7

Simulation and Experimental Evaluation

The verification phase represents a crucial step in the development of complex embedded systems, particularly in motor control applications that require high precision and reliability. Ensuring correct functionality through simulation methods and experimental testing is essential to reduce the risk of field malfunctions and to optimize the overall system performance.
Simulation, carried out using dedicated tools such as *ModelSim* and the *ICEcube2* Simulator, made it possible to analyze the behavior of the control logic under ideal conditions, verifying the correct sequence of states, the generation of step pulses, and compliance with the programmed acceleration and deceleration profiles. These simulations served as an initial filter to identify and correct design errors quickly and cost-effectively.
Subsequently, experimental tests were conducted on the implemented system to evaluate performance under real operating conditions, taking into account physical phenomena such as signal delays, electrical noise, and component tolerances. The comparative analysis between simulated results and experimental measurements made it possible to validate the model and identify any discrepancies, providing valuable insights for further optimizations.
This integrated approach of simulation and experimentation is essential to ensure that the final system meets the requirements of precision, reliability, and robustness demanded by industrial motor control applications.

7.1 Simulations

Simulations played a crucial role in the functional and timing verification phase of the system, allowing accurate testing of expected behaviors prior to physical implementation on the FPGA. In particular, several scenarios were developed and tested, focusing on three fundamental aspects: SPI communication control, correct command reception and transmission, and dynamic management of stepper motor motion.
Established development environments such as ModelSim and ICEcube2 were used for simula-

tion. These tools allowed both functional analysis, without delays, and timing analysis, including propagation delays and adhering to the timing constraints of the hardware platform. This dual approach ensured comprehensive control of both logic and real timing, which is essential for systems with high clock frequencies and temporal precision.
Regarding SPI communication control (**Figure 7.1.1**), targeted simulation tests were conducted to verify the correct transmission and reception of data between the microcontroller and the FPGA. Specifically, it was verified that every value inserted by the microcontroller was correctly transferred and in the expected order through the MOSI (Master Out Slave In) line.
During these simulations, particular attention was paid to the management of control and synchronization signals, such as the SPI clock and chip select signal, to ensure that data was acquired and sampled correctly on the active edges. Furthermore, the proper implementation of status flags, used as control signals to indicate, for instance, the presence of available data, read completion, or transmission completion, was verified. The correct updating of these flags proved to be fundamental to prevent data loss or overwriting during communication operations, thus ensuring the integrity and synchronization of the information flow.

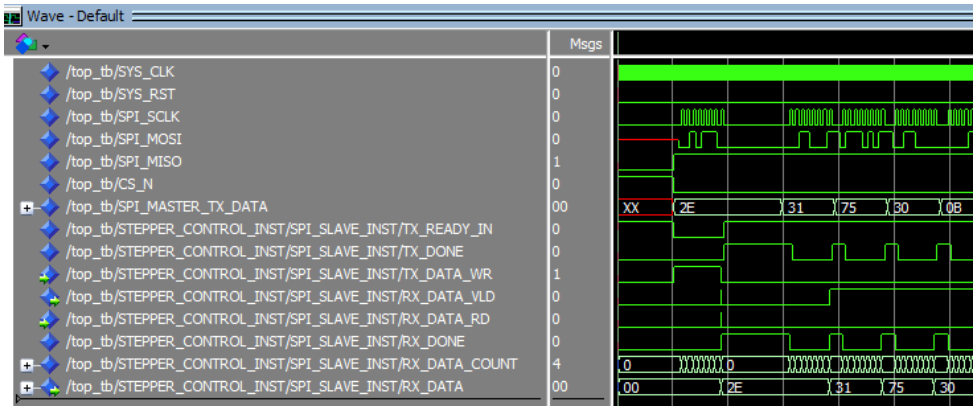


Figure 7.1.1: SPI module simulation.

The simulation of command receiving and sending processes involved the analysis of the bidirectional data flow, with particular attention to the correct management of command sequences. Long and complex command sequences were simulated, with the goal of ensuring that each individual command was recognized and processed in an orderly manner and without loss. This included verifying that each command triggered the correct transition of the finite state machine responsible for control logic, thus ensuring consistent and predictable system behavior even under high load conditions. Additionally, malformed or invalid commands were introduced to test the robustness of the protocol: the system had to promptly detect such anomalies, prevent inconsistent states or unwanted operations, and respond with appropriate error or acknowledg-

ment signals. This error control mechanism ensured reliable and safe interaction between the FPGA and the control host, contributing to maintaining operational integrity and stability even in the presence of unexpected situations or command-side malfunctions.

To model the input signals, system clocks with a fixed frequency were generated, reset signals were applied at specific moments to test initialization and recovery capabilities, and realistic command sequences were used with timings consistent with the specifications of the protocol and the state machine.

An example of a command sequence is shown in **Figure 7.1.2**. The `0x2E` command activates the configuration mode, with the `CHIP_SELECT` signal returning high at the end of the command. Subsequently, the `0x31` command selects the first motor, allowing the insertion of its configuration parameters. Once the writing is completed, `CS` is again set high to confirm the operation. Finally, the `0x32` command is used to select the second motor and proceed with its configuration.

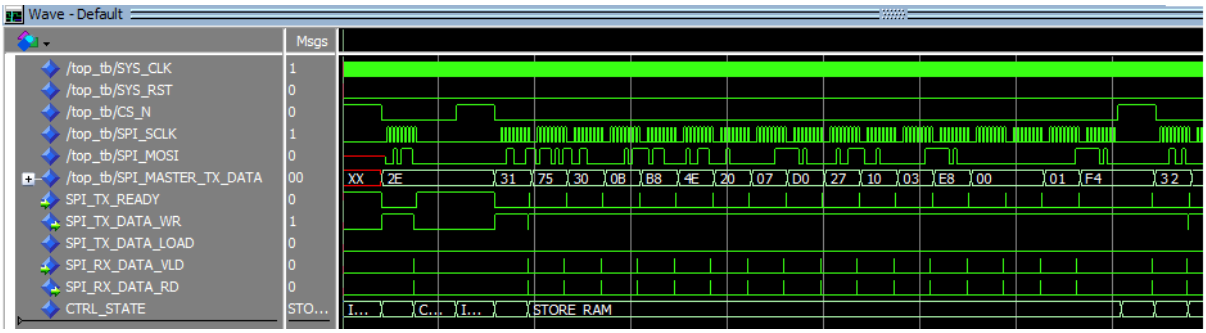


Figure 7.1.2: Command insertion simulation.

Finally, the motor movement tests (**Figure 7.1.3**) focused on the implementation of acceleration, constant speed, and deceleration profiles. The simulations highlighted the correct generation of step pulses with variable frequencies, the maintenance of the required timing sequence, and the synchronous management of the internal states of the finite state machine. Special attention was given to verifying the proper functioning of all timers involved in control, especially those responsible for periodically updating the step frequency and varying the pulse duration. This ensured that, during each real-time dynamic change, all data were updated correctly and synchronized, avoiding misalignments or timing errors in the command signal generation.

Through the analysis of waveforms, it was possible to verify the consistency of the step signals, the correctness of the transitions between motion phases, and the response to dynamic variations in control parameters. In error scenarios or sudden resets, the system demonstrated the ability to return to a stable state without loss of synchronization.

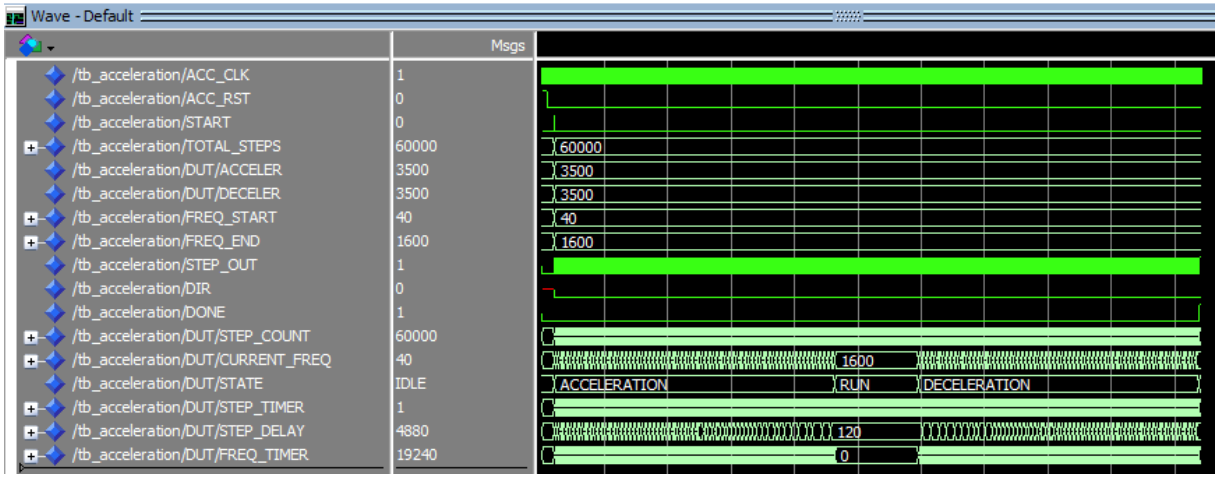


Figure 7.1.3: Motor movement simulation.

These tests formed the basis for promptly identifying and correcting anomalies and inconsistencies in the implemented logic, enabling the design to be optimized and significantly reducing the risk of malfunctions in the hardware implementation.

7.2 Experimental Tests

The experimental tests were conducted in several phases, with the aim of validating both the designed hardware and the correct functioning of the system as a whole, up to its actual use on the machine.

Initial setup and board verification:

The first testing phase focused exclusively on the designed electronic board. A couple of PCB prototypes were initially produced to carry out preliminary checks before starting production of the final version.

The tests concentrated on verifying the correct power supply to the components and the overall functionality of the layout. Some minor issues emerged, such as an incorrectly chosen microcontroller pin, which consequently could not output the expected signal, and a few suboptimal connections in a couple of integrated circuits. These defects were easily resolved on the test versions through simple manual modifications (e.g., wire bridges and soldering), and were subsequently corrected in the final layout of the board.

The verification was carried out by connecting the board to a 12 V power supply, which was then internally distributed through voltage regulators to 5 V, 3.3 V, and 1.2 V. Subsequently, the FPGA was tested through simple configurations to ensure its operability.

Communication protocol verification:
Once the integrity of the board and main components was verified, the communication protocol between the microcontroller and FPGA was tested. Configuration and execution commands were sent to verify that values were correctly written and read. No critical issues emerged during this phase either.

Single motor test:
After validating communication, a single stepper motor was connected to its driver, and the motion profile was tested at various speeds. Configurations with different minimum and maximum frequencies were tried to identify the system's practical limits, both in terms of driver and motor capabilities.
At this stage, it was possible to observe the system's response to acceleration and deceleration commands and verify that the profile was correctly executed also in physical terms. To support these tests, specific measurements and observations were carried out to validate the proper functioning of the STEP signal and the system's dynamic behavior:

- **Step pulse verification:** using an oscilloscope and logic analyzer, the STEP signals generated by the FPGA were observed, verifying their frequency and duty cycle during the different phases (acceleration, steady-state, deceleration).
- **Real frequency measurement:** using the oscilloscope's automatic functions, the frequency of the pulses during motion was measured and compared with theoretical and simulated values, showing excellent correspondence.
- **Dynamic response testing:** by changing motion parameters in real-time, the system demonstrated good responsiveness and stability, even during sudden changes in speed or acceleration.

Test on real machine:
The final testing phase involved mounting the board on a real test machine equipped with multiple motorized axes, capable of simultaneous multi-directional movement. The motors were tested both without load and under applied load to evaluate system behavior in different mechanical conditions.
The axes were connected gradually, one at a time, allowing for the individual validation of each motor and driver before integrating the full system. This incremental approach made it easier to isolate potential issues and verify proper functioning step by step.
Special attention was given to the vertical axis, which is subject to greater mechanical stress due to the weight of the moving components and the effect of gravity. In this case, the system was

tested to ensure consistent performance under load, with particular focus on the motor's ability to maintain position and follow motion profiles without losing steps or generating excessive vibration.
Additionally, the limit switches were thoroughly tested. Each axis was initialized from its home position (zero), and then moved along its full range until reaching the opposite limit. These tests confirmed that the system was able to automatically stop when reaching a limit, thereby preventing mechanical damage or loss of synchronization.
This functionality was validated both in manual control mode and during the execution of predefined motion trajectories. The correct integration of limit switches into the firmware contributed to the overall reliability, precision, and safety of the motion control system.

Issues encountered and solutions adopted:
During the testing phases, some practical issues were encountered and resolved with targeted interventions:

- **Step loss at high speeds:** Initially, step losses occurred when high speeds were requested. After careful analysis, it was concluded that the driver could not handle such a high number of microsteps at high frequency. The issue was resolved by reducing the number of microsteps from 256 to 64, maintaining good motion smoothness and thereby allowing the maximum speed to be quadrupled.
- **Driver overheating:** In some conditions, a significant increase in driver temperature was observed, especially during long movements or under load. The issue was resolved by installing larger heat sinks and improving airflow around the components.
- **Motor acoustic noise:** Another issue, more annoying than functional, was the noise produced by the motors during operation. Although not excessive, the continuous sound could become disturbing over time. The solution was found by modifying the driver's current regulation decay mode, choosing an option with a slower decay that was acoustically quieter.

The entire experimental phase confirmed the effectiveness of the project and allowed the system behavior to be refined to ensure stability, reliability, and readiness for real-world deployment.

7.3 Comparison between Simulation and Experimental Results

The comparison between simulations performed during development and real experimental tests showed a high degree of consistency between expected behaviors and those observed in practice. In particular, the implemented motion profiles, including acceleration, constant speed, and deceleration phases, were accurately reproduced both in the control logic and in the generation of step pulses. This confirmed that the theoretical assumptions made during the design phase were correctly translated into functional hardware behavior.
The simulated waveforms accurately represented the timing of frequencies and transitions between the states of the finite state machine (FSM), which was confirmed by experimental data collected using high-resolution measurement tools such as oscilloscopes and logic analyzers. The real signals exhibited frequencies, duty cycles, and switching times fully aligned with the predicted values, not only in static conditions but also during rapid state transitions and frequency changes.
Another key aspect was verifying the system's response to dynamic changes in control parameters such as speed and acceleration. The results showed that the implemented logic maintains stability, responsiveness, and precision even under abrupt or non-linear changes, which is crucial for motion control applications where timing and synchronization are critical.

The minimal discrepancies observed, attributable to physical delays in electronic components, signal noise, and construction tolerances, were negligible and had no significant impact on the system's overall performance. These variations fall within the normal behavior of real-world digital systems and were considered physiological, requiring no modifications or recalibrations of the original design.
Overall, the excellent agreement between simulation and experimental results confirmed the validity of the adopted design methodology and demonstrated the reliability of the simulation tools in predicting the actual behavior of the system. This alignment strengthens confidence in the system's robustness and suitability for real-time embedded applications, particularly in domains where accurate control of motion and timing is essential.

Below is a representative test case comparing design parameters, simulation, and experimental measurements. Note that the indicated values are theoretical calculations based on ideal formulas, while the FPGA implementation uses approximations due to rounding and truncation. Therefore, a perfect match between theoretical data and practical results is not possible.

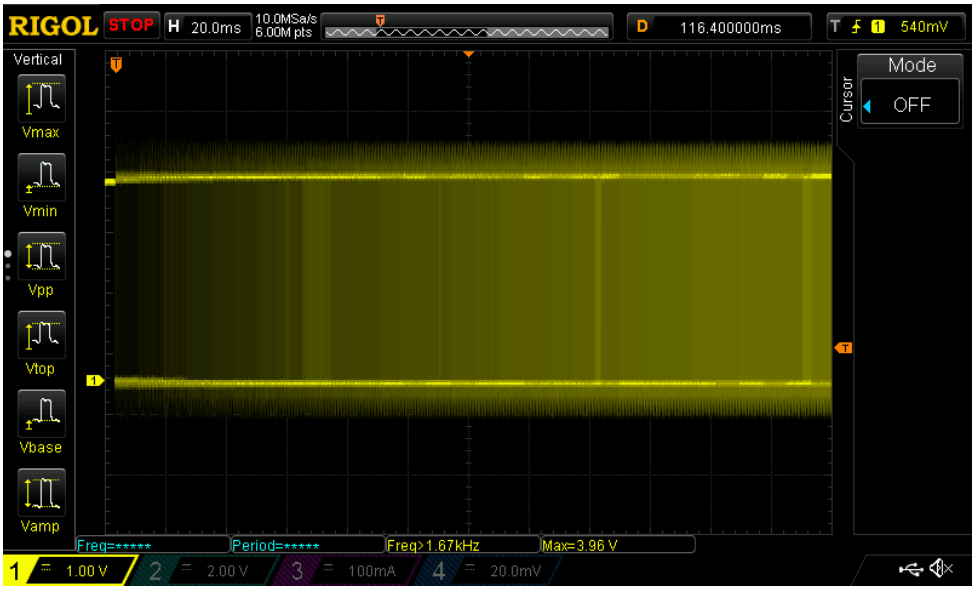


Figure 7.3.1: Full acceleration phase, showing a gradually more evident frequency increase.

- **Input parameters:** initial speed $V_{\text{min}} = 40$ step/s, maximum speed $V_{\text{max}} = 1600$ step/s, acceleration $= 3500$ step/s^2 (equivalent to 224000 microstep/s^2).
  With a resolution of 64 microsteps per full step, the resulting frequency will be equal to the speed multiplied by 64.

- **Number of increments:** to go from 40 step/s to 1600 step/s, the motor passes through 64 speed levels. The increment per step is therefore approximately 24 step/s.

- **Theoretical acceleration time:** using the formula $t = \Delta v / a$, with $\Delta v = 1560$ step/s and $a = 3500$ step/s^2, the total acceleration time is approximately 445 ms.

- **Duration of each increment:** Since the increase in speed is linear, the time between two frequency changes is approximately 445 ms$/64 \approx 7$ ms.

Figure 7.3.2 compares the waveform obtained from the numerical simulation with the experimentally acquired signal at the lowest considered frequency of 2560 Hz (40 steps/s). The two signals show strong agreement, especially in terms of the period, which is identical in both cases. This confirms the temporal accuracy of the simulated model.

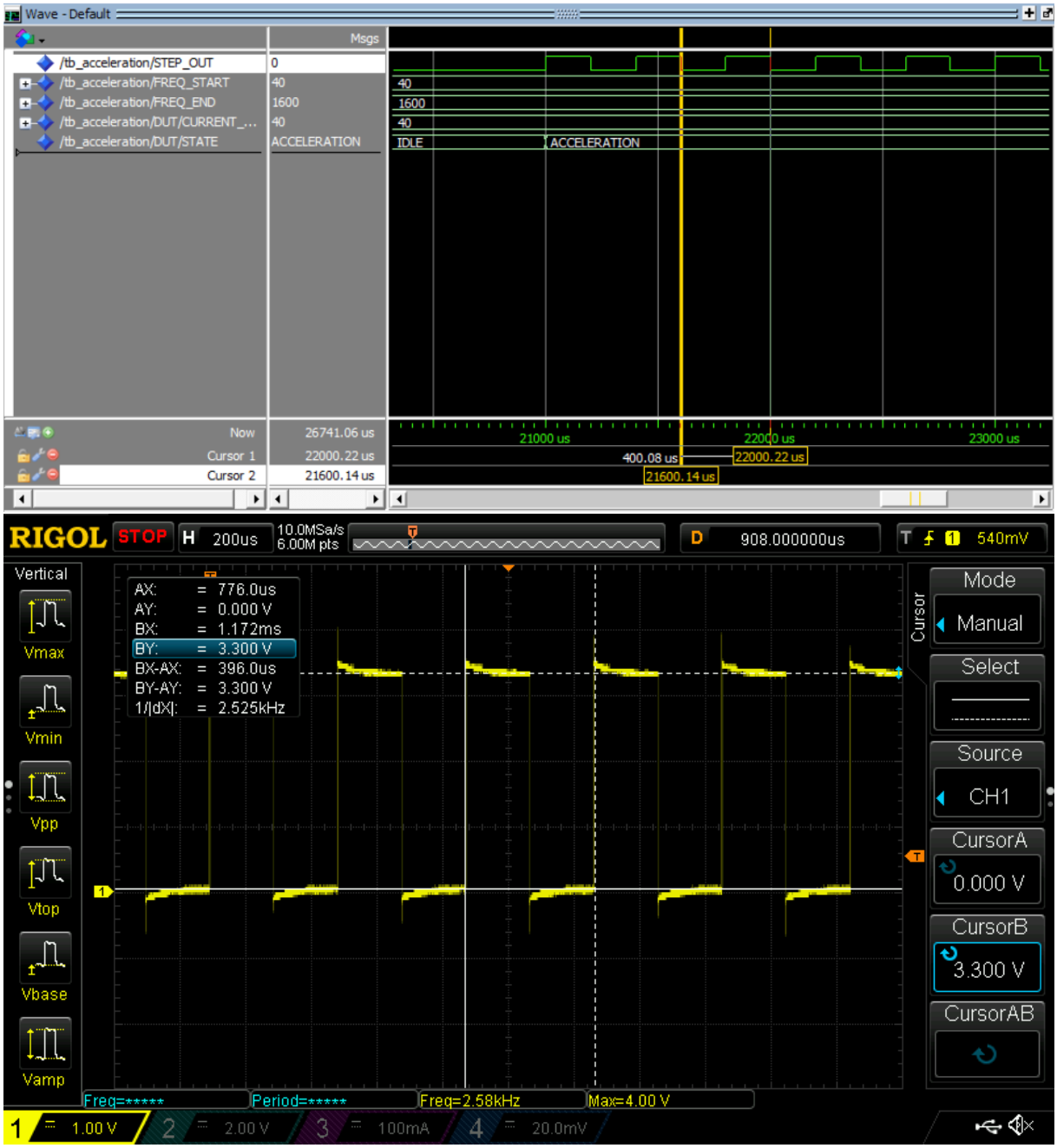


Figure 7.3.2: Comparison between simulation and experimental results at the minimum frequency.

Figure 7.3.3 illustrates the frequency transition from 2560 Hz (40 steps/s) to 4096 Hz (64 steps/s). The transition occurs at 7.7 ms, as clearly visible in both the simulation output (bottom center of the first image) and the oscilloscope capture (top right of the second image). This precise alignment demonstrates the model's ability to replicate the timing of dynamic changes in the real system.

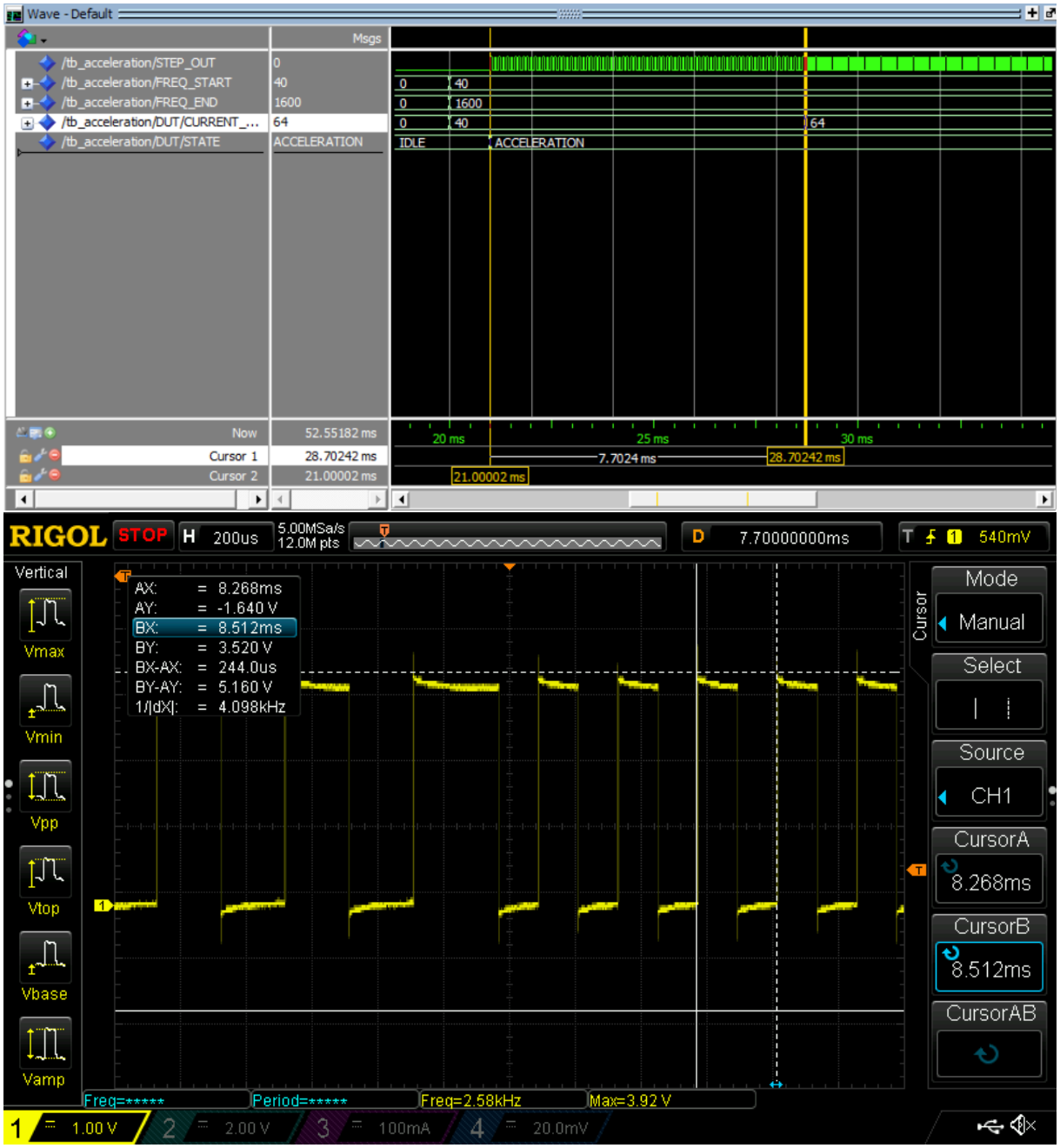


Figure 7.3.3: First frequency transition, from 2560 Hz to 4096 Hz.

Figure 7.3.4 presents the second frequency transition, from 4096 Hz (64 steps/s) to 5632 Hz (88 steps/s), occurring at 15.5 ms. The transition point is well-defined in both the simulated and experimental waveforms, further validating the model's temporal fidelity.

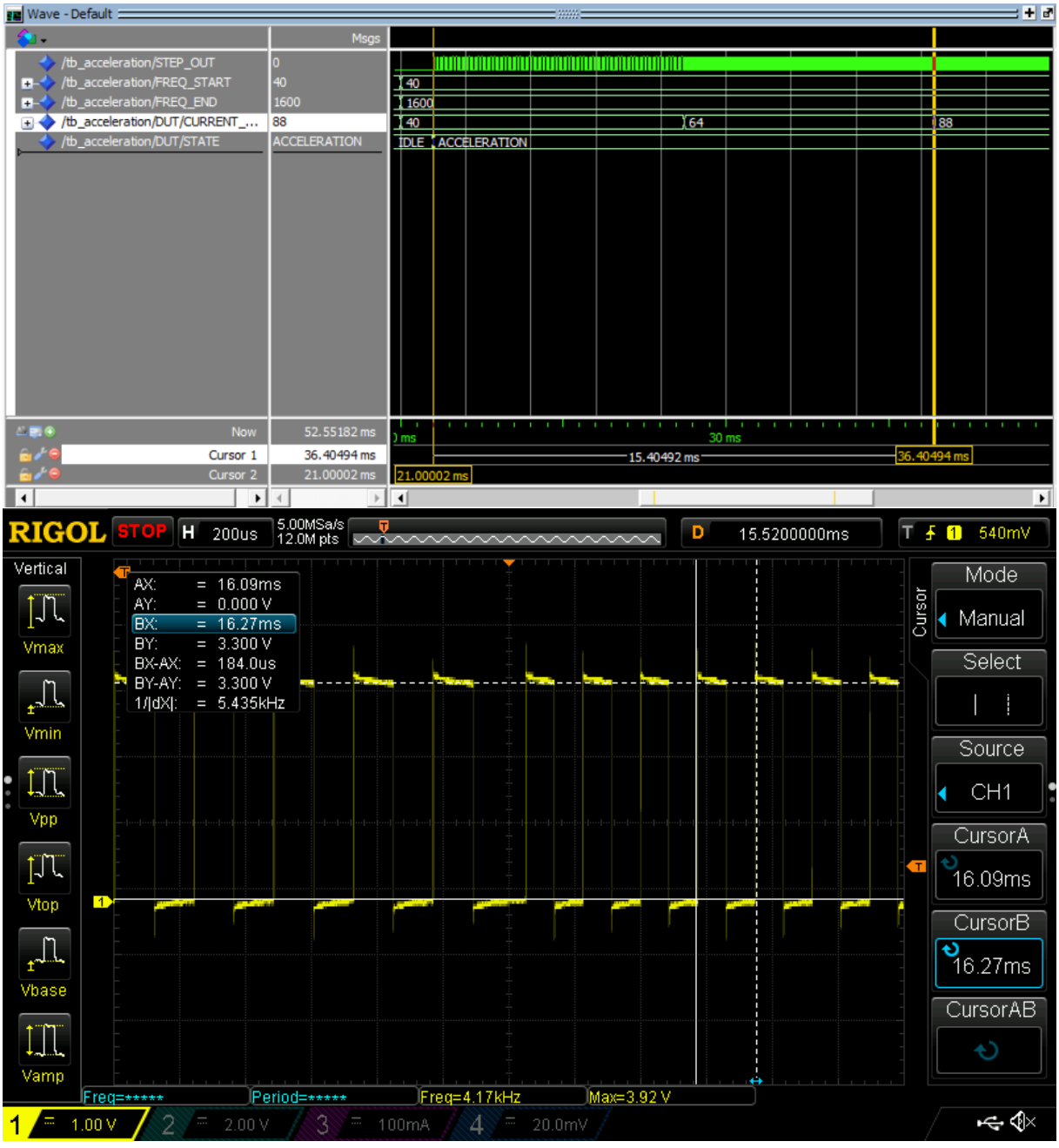


Figure 7.3.4: Second frequency transition, from 2560 Hz to 4096 Hz.

Figure 7.3.5 shows the final frequency change, reaching the maximum value of 10240 Hz (1600 steps/s) at 442 ms. Once again, the simulation closely matches the experimental signal, confirming the model's consistency in capturing the system's timing under varying operational conditions.

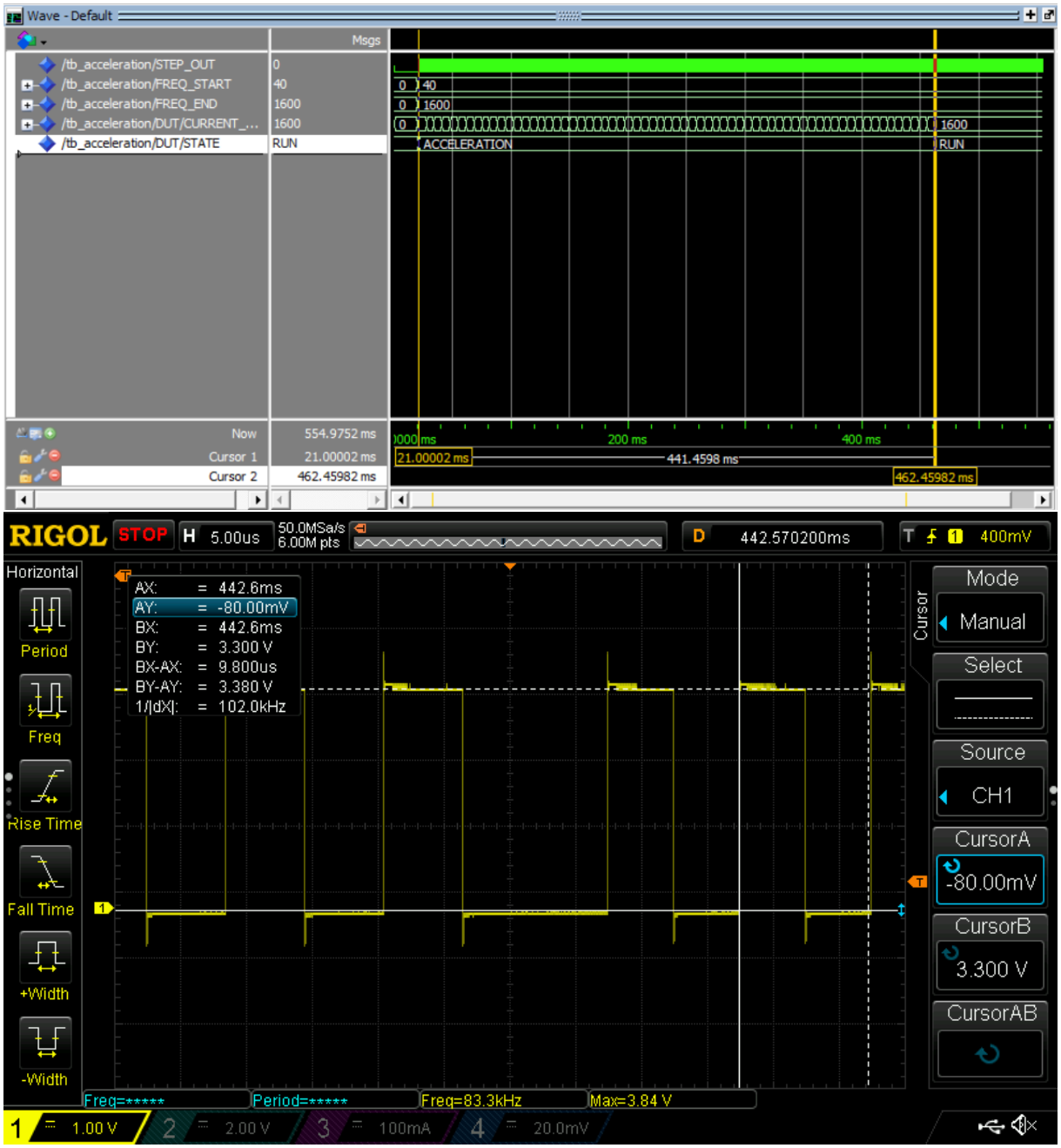


Figure 7.3.5: Maximum speed reached, 102.4 KHz.

The measurements carried out with the oscilloscope confirmed that all frequency transitions occurred with precise timing, fully matching the numerical simulations. The complete acceleration profile developed within the expected time frame, and the stepping frequency followed the intended linear progression without deviation. These results validate not only the functional correctness of the VHDL control logic but also the reliability of its physical implementation on the FPGA board.

The excellent agreement between simulated and experimental signals demonstrates that the system behaves exactly as designed, both in terms of timing and frequency control. This confirms the effectiveness of the adopted design methodology and marks the successful validation of the implemented acceleration profile. The results obtained in this chapter provide definitive proof that the system is ready for integration into more complex motion control applications, with a solid foundation of temporal precision and functional robustness.

Chapter 8

Conclusions and Future Developments

This work has addressed the design and implementation of a control system for stepper motors based on FPGA technology, with a particular focus on the generation of customizable acceleration profiles and the modular design of digital logic. The use of an FPGA-based solution allowed for a high degree of flexibility and parallelism, fundamental aspects for precise and timely management of stepper motor dynamics, especially in applications where speed and reliability are critical requirements.

During development, numerous complex technical aspects were tackled and resolved, such as signal synchronization, accurate step pulse generation, and precise speed control during acceleration and deceleration phases. The adopted modular structure facilitated the isolation and optimization of individual functionalities, promoting easy scalability of the system. Although more stepper motors than necessary were used in the current implementation for demonstration purposes, the designed architecture allows the number of motors to be easily expanded simply by adopting FPGAs with greater logic and memory resources, thus making the solution highly adaptable to different application contexts and project sizes.

Preliminary simulations, confirmed by experimental tests on a physical prototype, validated the effectiveness of the system, showing that the generated speed profiles closely matched the theoretical expectations. This confirms the suitability of the proposed approach for industrial applications, where accurate motor control is essential to ensure quality and repeatability of the processes.

In addition to technical aspects, attention was also given to the documentation and organization of the project, which features a clear and modular structure, facilitating future revisions and updates. The adoption of standard technologies such as the VHDL hardware description language, along with the use of open-source tools for simulation and testing, further improves the accessibility and reproducibility of the work carried out.

Future Developments

Among the possible future development directions, a first aspect concerns the integration of feedback systems for closed-loop control. The addition of position sensors, such as optical or magnetic encoders, would allow the implementation of more sophisticated and robust control strategies, such as PID or adaptive control. This would improve the accuracy and stability of the system, enabling real-time correction of errors or disturbances and ensuring high performance even under varying conditions.

Another development could involve the optimization of acceleration and deceleration algorithms, through the adoption of more advanced models such as S-curve profiles, which help reduce mechanical stress and improve the smoothness of motion. This would require a refinement of the control logic and greater computational capacity, but would bring significant benefits in terms of component longevity and motion quality.

From a hardware perspective, migrating to next-generation FPGAs, with larger resources and superior performance, would enable the implementation of even more complex multi-motor control systems, supporting a higher number of axes and introducing additional functionalities such as integrated communication management or advanced monitoring of operating conditions.

Finally, the possibility of integrating more advanced communication modules (e.g., advanced Ethernet or industrial protocols such as EtherCAT) would make the system more easily integrable into complex industrial automation contexts, enabling remote control, data collection, and predictive analysis.

Conclusions

In conclusion, this thesis represents a significant contribution in the field of embedded control systems for stepper motors, demonstrating how FPGA-based solutions can offer effective, scalable, and customizable control architectures, capable of meeting the growing demands of modern industrial automation. The modularity of the project and the choice of open technologies facilitate further developments and adaptations, making the system a solid starting point for increasingly complex and high-performance future implementations.

Bibliography

[3] B. C. Kuo, *Automatic control systems*, 6th. Prentice Hall, 1991.

[4] P. Acarnley, *Stepping Motors: A guide to theory and practice*, 4th. The Institution of Engineering and Technology, 2002.

[6] A. Huges and B. Drury, *Electric Motors and Drives: Fundamentals, Types and Applications*, 5th. Newnes, 2019.

[8] M. Zigliotto, *Macchine e azionamenti elettrici*, Appunti di corso universitario, 2023.

[13] C. Donaldson, T. McMillan, and W. Nichols, *Stepper Motors: Fundamentals, Applications and Design*, 1st. Institute of Electrical Engineers (IEE), 2003.

[14] M. A. Mazidi, R. D. McKinlay, and D. Causey, *The 8051 Microcontroller and Embedded Systems: Using Assembly and C*, 2nd. Pearson, 2008.

[15] J. B. Peatman, *Designing with Microcontrollers*, 2nd. McGraw-Hill, 1997.

[16] S. Buso, *Introduzione alle applicazioni industriali di microcontrollori e DSP*, 2nd. Bologna: Società Editrice Esculapio, 2018.

[17] F. Zappa, *Microcontrollers: Hardware and Firmware for 8-bit and 32-bit Devices*. Bologna: Società Editrice Esculapio, 2017.

[21] C. Ünsalan and B. Tar, *Digital System Design with FPGA: Implementation Using Verilog and VHDL*. McGraw-Hill Education, 2017.

[27] C. D. Systems, *OrCAD PCB Designer User Guide*. Cadence, 2021.

[28] H. W. Johnson and M. Graham, *High-Speed Digital Design: A Handbook of Black Magic*. Prentice Hall, 2003.

[29] S. W. M. Steeghs, *Design Automation of PCB Layout: for Power Electronics with Reinforcement Learning*. 2023, Master's thesis, Mathematics and Computer Science, Eindhoven University of Technology.

[32] E. Bogatin, *Signal and Power Integrity - Simplified*, 3rd. Pearson, 2018.

[35] Y. K. Chow, J. F. Poliakoff, and P. D. Thomas, “Interpolation and acceleration algorithms for stepper motors – a parametric approach,” in *Methods and Models in Automation and Robotics*, 2002, pp. 1179–1184.

[37] M. Scarpino, *Motors for Makers: A guide to steppers, servos, and other electrical machines*, 1st. Que Publishing, 2016.

[41] D. Harris and S. Harris, *Digital Design and Computer Architecture*, 1st. Morgan Kaufmann, 2012.

[42] P. A. Laplante, *Real-Time Systems Design and Analysis: Tools for the Practitioner*, 3rd. Wiley-Interscience, 2004.

[43] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design Using VHDL: A Systems Approach*. Cambridge University Press, 2015.

[44] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, 1st. Wiley, 2018.

Sitography

[1] EMCElettronica, *L'importanza dei driver motori*, https://it.emcelettronica.com/limportanza-dei-driver-motori, 2023.

[2] Texas Instruments, *Drv8462 low-voltage stepper motor driver datasheet*, Retrieved from https://www.ti.com/lit/ds/symlink/drv8462.pdf.

[5] FUYU Technology Co., Ltd., *Fuyu technology: Linear motion system intelligent products*, Retrieved from https://www.fuyumotion.com/.

[7] Automate, *What is the difference between full stepping, half-stepping and the micro drive?* https://www.automate.org/motion-control/case-studies/what-is-the-difference-between-full-stepping-the-half-stepping-and-the-micro-drive, 2018.

[9] ResearchGate, *Mixed-mode pwm for high-performance stepping motors*, https://www.researchgate.net/publication/3219650_Mixed-Mode_PWM_for_High-Performance_Stepping_Motors, 2008.

[10] ResearchGate, *Development of a simulation environment to investigate the control behaviour of a permanent magnet synchronous motor*, https://www.researchgate.net/publication/309727869_Development_of_a_simulation_environment_to_investigate_the_control_behaviour_of_a_permanent_magnet_synchronous_motor, 2015.

[11] Haydon Kerk Pittman, *Stepper motor theory*, https://www.haydonkerkpittman.jp/learningzone/technicaldocuments/stepper-motor-theory, 2015.

[12] ResearchGate, *On the optimal teeth geometry of a hybrid linear stepper motor*, https://www.researchgate.net/publication/292077858_On_the_Optimal_Teeth_Geometry_of_a_Hybrid_Linear_Stepper_Motor, 2015.

[18] NXP Semiconductors, *Lpc1768/66/64/60/61/62/63 user manual*, Retrieved from https://www.nxp.com/docs/en/data-sheet/LPC1769_68_67_66_65_64_63.pdf.

[19] Lattice Semiconductor, *Ice40hx1k fpga datasheet*, Retrieved from https://www.latticesemi.com/ice40.

[20] HACARUS, *Latest trends in non volatile fpgas*, https://hacarus.com/information/20210212-tech-5-latest-fpga/, Accessed: 2025-08-01, 2021.

[22] onsemi, *Ncv7344: High-speed can transceiver datasheet*, Retrieved from https://www.onsemi.com/pdf/datasheet/ncv7344-d.pdf.

[23] Texas Instruments, *Dp83848c single port 10/100mb/s ethernet phy datasheet*, Retrieved from https://www.ti.com/product/DP83848C.

[24] Winbond Electronics, *W25q16jv: 16mbit serial nor flash memory datasheet*, Retrieved from https://www.winbond.com/hq/product/code-storage-flash-memory/serial-nor-flash/?__locale=en&partNo=W25Q16JV.

[25] SiTime, *Sit8008: Lowpower programmable mems oscillator datasheet*, Retrieved from https://alcom.eu/uploads/SiTime-SiT8008-Datasheet.pdf.

[26] Monolithic Power Systems, Inc., *Mp2161gj:step down converter datasheet*, Retrieved from https://www.monolithicpower.com/en/documentview/productdocument/index/version/2/document_type/Datasheet/lang/en/sku/MP2161GJ/document_id/450/.

[30] NextPCB, *Ipc-2221: The standard for printed circuit board design*, https://www.nextpcb.com/blog/ipc-2221, 2023.

[31] Digi-Key Electronics, *Pcb trace width calculator*, https://www.digikey.it/en/resources/conversion-calculators/conversion-calculator-pcb-trace-width, 2025.

[33] T. Instruments, *Applying acceleration and deceleration profiles to bipolar stepper motors*, https://www.ti.com/lit/an/slyt482/slyt482.pdf, 2011.

[34] M. T. Inc., *Avr446: Linear speed control of stepper motor*, https://ww1.microchip.com/downloads/en/Appnotes/doc8017.pdf, 2006.

[36] S. Baruch, *Basic division in sequential systems and computer engineering*, https://users.utcluj.ro/~baruch/book_ssce/SSCE-Basic-Division.pdf, 2010.

[38] Lattice Semiconductor, *Icecube2 user guide*, Retrieved from https://www.latticesemi.com/products/designsoftwareandip/fpgaandlds/icecube2.

[39] Lattice Semiconductor, *Memory usage guide for ice40 devices*, Retrieved from https://www.latticesemi.com/products/designsoftwareandip/fpgaandlds/icecube2.

[40] Lattice Semiconductor, *Using mentor modelsim simulator with lattice icecube2*, Retrieved from https://www.latticesemi.com/products/designsoftwareandip/fpgaandlds/icecube2.