DESIGN AND FPGA IMPLEMENTATION OF PID CONTROLLER

A Project Report

Submitted by

GAURAV DEV & MUNAZIR REZA

(20ELB171 & 20ELB172)

In Partial Fulfilment

of the Requirements for the award of the degree

BACHELOR OF TECHNOLOGY

Department of Electronics Engineering,

Zakir Husain College of Egineering & Technology

Aligarh Muslim University,

Aligarh, India-202002

APRIL 2024

CERTIFICATE

This is to certify that the project report entitled "**Design and FPGA implementation of PID Controller**" that is being submitted to the Department of Electronics Engineering, Zakir Husain College of Engineering and Technology, Aligarh Muslim University, Aligarh, in partial fulfillment for the award of the degree of **Bachelor of Technology in Electronics Engineering**, during the session 2023-24, is a record of candidates' own work carried out under my supervision.

Dr. Naushad Alam
Supervisor
Professor
Dept. of Electronics Engineering
AMU, Aligarh, 202002

Place: Aligarh

Date: May 12, 2024

Candidate's Declaration

We hereby declare that our work submitted in the partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in the Department of Electronics Engineering of the Aligarh Muslim University, titled as **DESIGN AND FPGA IMPLEMENTATION OF PID CONTROLLER** is a record of our work carried out during the **VII Semester** from August 2023 to December 2023 and **VIII Semester** from January 2024 to May 2024 under the guidance of **Prof. Naushad Alam** , Professor, Department of Electronics Engineering, AMU.

The matter presented in this report has not been submitted by us for the award of any other degree of this or any other Institute/University.

We have also received a plagiarism report from MA Library and submitted it to our guide, where the similarity index is _ %.

Gaurav Dev - 20ELB171 **Munazir Reza - 20ELB172**

This is to certify that the above statement made by the candidates is true to the best of my knowledge and belief.

Prof. Naushad Alam

Date: May 12, 2024 **Supervisor**

ACKNOWLEDGEMENTS

We would like to start by expressing our sincerest gratitude to our Project supervisor **Prof. Naushad Alam**, Professor, Department of Electronics Engineering at the Aligarh Muslim University, for his expertise, guidance, and enthusiastic involvement during our coursework. We are highly obliged to the faculty members of the Electronics Engineering Department because, without their valuable insights and constructive opinions during evaluations, our project would not have yielded significant results and led us to explore a myriad of use cases that we have put forward in this report. We express our special thanks to our parents for their encouragement, constant moral Support, and to our friends and colleagues for being inquisitive and supportive during the course of this project

Gaurav Dev - 20ELB171

Munazir Reza - 20ELB172

ABSTRACT

KEYWORDS: PID, Controller, FPGA, MATLAB, Simulink, DC Motor, Hardware Implementation, Real-time Speed Measurement. IR Sensor

This project presents the design of a digital Proportional-Integral-Derivative (PID) controller using MATLAB Simulink and implementation on FPGA boards.The PID controller stands out in engineering and automation systems for its adeptness in effectively regulating system behavior, making it a widely adopted control algorithm. This work focuses on developing a digital version of the PID controller, suitable for real-time applications and hardware integration.

In the design process, the PID controller is modeled using MATLAB Simulink, with a focus on optimizing key parameters like the proportional gain (Kp), integral gain (Ki), and derivative gain (Kd) to achieve the desired system response. The Simulink model is then converted into hardware description language (HDL) code compatible with FPGA platforms.

The FPGA implementation of the digital PID controller offers advantages such as high-speed processing, low-latency response, and potential for parallel processing of control tasks. The controller is synthesized and mapped onto the FPGA, allowing for real-time control of physical systems or processes.

In our project, we chose the FPGA approach and employed the Xilinx Artix-7 series FPGA on the Nexys4 DDR board. We controlled the DC motor's speed and programmed the FPGA using Verilog HDL. Our design leveraged FPGA technology, with simulations conducted through the Xilinx Vivado tool. The real-time speed measurement was achieved using an IR Sensor, and the L298N motor driver ensured the efficient operation of the DC motor

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

PID	Proportional Integrator Derivative
FPGA	Field Programmable Gate Array
DC	Direct Current
KI	Integrator Gain
KD	Derivative Gain
KP	Proportional Gain
KB	Back Proportional Gain
IR Sensor	Infrared Sensor
PWM	Pulse Width Modulation
CT PID	Continuous time PID
DT PID	Discrete time PID
RPM	Rotation per minute

NOTATION

J	Moment of inertia of motor
B	Damping ratio of mechanical system
τ_m	Mechanical time constant of armature circuit
τ_a	Time constant of armature circuit
$K_e\phi$	Electromotive force constant
R_a	Armature Resistance
I_a	Armature current

CHAPTER 1

Introduction

1.1 Problem Statement

To develop an efficient design of a digital PID controller. The goal is to implement this controller on an FPGA board to control the speed of the DC motor and enable real-time speed measurement of the DC motor.

1.2 Proposed Solution

This project aims to develop an efficient digital PID controller with features like anti-windup to prevent output saturation and counter high-frequency disturbances. It achieves this by incorporating a low-pass filter in the derivative path and limiting the PID output to protect the load. The PID controller is implemented on an FPGA board, controlling the speed of a DC motor and performing real-time speed measurements using an IR sensor. Through these integrated components, the project targets precise motor control, addresses common challenges in control systems, and advances industrial automation and control technology

1.3 Overview of PID Controller

Controllers are integrated into numerous specialized control systems, where PID control often collaborates with logic, sequential functions, selectors, and basic function blocks to construct intricate automation systems employed in energy production, transportation, and manufacturing. Advanced control strategies like model predictive control are typically organized hierarchically, with PID control serving at the fundamental level; higher-level multivariable controllers assign set points to the lower-level controllers. Consequently, the PID controller is fundamental in control engineering, recognized as a crucial component in every control engineer's toolkit.PID controllers have adapted across various technological shifts, transitioning from mechanical and pneumatic systems to microprocessors through electronic tubes, transistors, and integrated circuits. The introduction of microprocessors has significantly impacted PID controllers, with the majority of modern PID controllers relying on

microprocessor technology. This advancement has enabled the integration of additional functionalities such as automatic tuning, gain scheduling, and continuous adaptation, enhancing the capabilities of PID controllers.

It consists of three control actions: proportional, integral, and derivative, each contributing to the overall control strategy.

The proportional action generates an output signal proportional to the current error, which is the difference between the desired set point and the actual system output. This action provides immediate correction based on the present error, adjusting the control signal proportionally to the error magnitude. However, proportional control alone may lead to steady-state error, especially in systems with constant disturbances.

To address this, the integral action integrates the error signal over time, summing up the past errors and effectively eliminating steady-state errors. It continuously adjusts the control signal based on the accumulated error history, ensuring accurate and stable control performance over time.

Additionally, the derivative action anticipates the future trend of the error signal by calculating the rate of change of the error. This action dampens rapid changes in the system, improving stability and response time while reducing overshoot and oscillations.

We desire control over every process/system to achieve a controlled response/output from the process/system. Particularly, we desire that the system should respond quickly to the input change, the output should be stable, and the system must have high disturbance rejection capability. So, how do we achieve these properties? The answer is we need some kind of controller that can look over the behavior of our system and perform some action to maintain the system in a desired state. But how do we decide what behavior to look for? These in the majority of the cases are: How much is actual response deviated from the set point (desired response), and how quickly this deviated value is changing (increasing or decreasing). The difference between the set point and response is called as error.

These all behaviors are monitored by a controller called a Proportional-Integral-Derivative (PID) Controller. Over the last fifty years, Proportional Integral derivative (PID) controllers have gained extensive use owing to their simplicity, resilience, efficiency, and versatility across various systems. Despite the emergence of numerous control design methods in the literature, PID controllers continue to dominate, finding application in over 95% of industrial processes.

The traditional PID controller operates in an analog system and requires an additional mod-

ule for interfacing with digital systems. The suggested concept of a Discrete PID Controller for DC Motor Speed Control aims to facilitate the seamless integration of digital systems with motors.[1]

Also, Analog or continuous-time PID controllers face challenges due to their reliance on physically changing components, making them susceptible to component drift and temperature sensitivity, which can lead to stability issues in control systems. Moreover, these controllers are prone to noise and interference, affecting their accuracy. On the other hand, digital PID controllers offer distinct advantages. They provide greater flexibility in tuning, allowing for precise adjustments of control parameters. Furthermore, their digital nature makes them immune to noise and interference, enhancing their reliability in noisy environments. Additionally, digital PID controllers are less affected by component variations, resulting in increased stability in control systems. These characteristics collectively position digital PID controllers as a more stable and adaptable choice in various applications compared to their analog counterparts.

1.4 Literature Review

1. The research paper [1] discusses the traditional DC motor and its speed control methods, highlighting the limitations of the conventional controller in interfacing with computer systems. To address this issue, the paper presents a discrete PID controller designed for the speed control of DC motors, emphasizing its compatibility with computer systems and ease of implementation. The dynamics of the system and controller parameters are calculated using the Root Locus method, and the complete simulation is conducted using MATLAB/SIMULINK software. The paper also elaborates on the integration of the DC motor with power electronics circuits to enhance its performance and achieve improved speed control. This upgraded DC motor is proposed for various industrial applications requiring adjustable speed control, such as electric cranes and manipulative vehicles. Furthermore, the paper discusses the mathematical modelling of the DC motor, the conventional PID controller design, and the discrete PID controller design. Results and discussions of different controllers, including the performance comparison between conventional PID and discrete PID controllers, are also presented. A comprehensive analysis of the closed-loop transient response and the use of the root locus technique to design the required compensator for stabilizing the system is included. Overall, the paper underscores the significance of discrete PID controllers in improving the speed control of traditional DC motors, especially for industrial applications. The proposed integration of power electronics circuits with the DC motor shows promise for enhancing performance. The research also suggests the potential for future investigations of DC motors using hybrid soft computing techniques.

2. The paper [2] introduces a closed-loop motion control system utilizing a BP neural network (BPNN) PID controller implemented on a Xilinx field-programmable gate array (FPGA) platform. The design is structured into two main components: the BPNN PID

control algorithm design and the peripheral module design for the closed-loop control system. The control algorithm comprises several sub-modules like the forward propagation module, PID module, main state machine module, and error back-propagation with weight update module. Peripheral modules encompass the speed measurement module and pulse width modulation (PWM) signal generation module.

The results from simulations and experiments validate the system's reliability, high real-time performance, and robust anti-interference capabilities. The proposed system achieves self-tuning of PID control parameters and significantly outperforms microcontroller unit (MCU)-based convergence speeds by more than three orders of magnitude. Furthermore, the FPGA-based approach exhibits superiority over traditional MCU-based control methods. Integration of BPNN and PID control algorithms showcases adaptive closed-loop control abilities, offering insights for intelligent motion control systems.

The study recommends enhancements in FPGA resource utilization and user interaction, such as refining activation function implementation methods for accuracy and resource efficiency, and developing remote wireless touch screens for improved operability.

The PID controller acts based on the value of the error. As understood by the name PID controller has three components, the behavior of each components are discussed below [3].

1.5 PID Algorithm

The **textbook** version of the PID algorithm is described by:

$$u(t) = K(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}) \tag{1.1}$$

The measured process variable is denoted by y, the reference variable by r, the control signal by u, and the control error by e which is calculated as $e = y_{sp} - y$, where y_{sp} is the set point.

The control signal is a combination of three terms: the proportional term (P-term), which is proportional to the error; the integral term (I-term), which is proportional to the integral of the error; and the derivative term (D-term), which is proportional to the derivative of the error. The controller parameters include the proportional gain K, integral time T_i, and derivative time T_d. These terms correspond to control actions based on past, present, and future values, respectively.

1.5.1 Proportional Controller (P)

The proportional term generates an output directly proportional to the present error. Its purpose is to diminish steady-state error by adjusting proportionally to the error's magnitude. When the error is substantial, the corrective response is more pronounced.

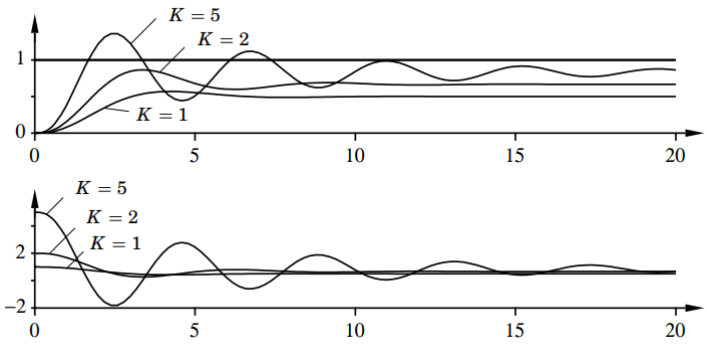

Figure 1.1: closed-loop system response with proportional control

Proportional control is illustrated in Fig. 1.1. The proportional controller is characterized by Fig. 1.1 with $T_i = \infty$ and $T_d = 0$. The figure demonstrates a persistent steady-state error in proportional control. While the error decreases with increasing gain, there is also an escalating tendency toward oscillation.

1.5.2 Integral Controller (I)

The integral term accounts for the accumulation of past errors over time. It is employed to eradicate any residual steady-state error that the proportional term may not rectify. Additionally, the integral term aids in managing system biases or long-term trends.

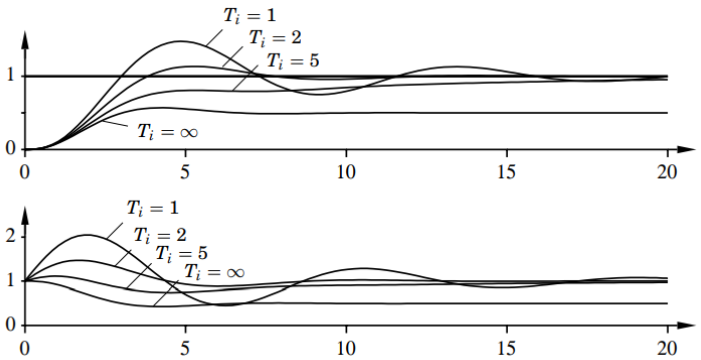

Figure 1.2: Closed-loop system response with proportional & integral control

Fig 1.2 illustrates the impact of adding integral action. As discussed in Fig 1.1, the influ-

ence of integral action strengthens as the integral time T_i decreases. The figure demonstrates the elimination of steady-state error with the use of integral action. This contrasts with the concept of the "magic of integral action." However, reducing T_i also tends to increase

1.5.3 Derivative Controller (D)

The derivative term anticipates future errors by analyzing the error's rate of change. It aids in damping the system's response, thereby preventing overshooting or oscillations. This predictive element of the derivative term helps counteract abrupt changes in the error.

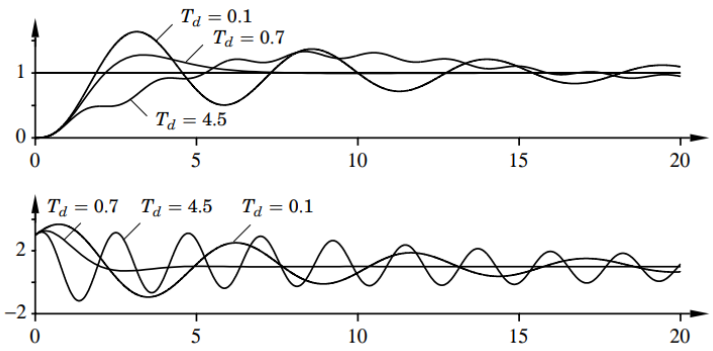

Figure 1.3: Closed-loop system response with proportional, integral & derivative control

Fig 1.3 showcases the impact of incorporating derivative action. The parameters K and T_i are chosen to induce oscillations in the closed-loop system. Damping increases with the derivative time T_d until it starts decreasing with excessively large T_d values. It's crucial to recognize that derivative action provides prediction through linear extrapolation over T_d. However, if T_d is overly large, derivative action loses effectiveness. In Fig 1.3, the oscillation period is approximately 6 seconds.

1.6 Continuous Time PID Controller

A continuous-time PID controller is a widely used control mechanism for continuously adjusting a system's output to match a desired set-point. It works by analyzing the difference (error) between the current measurement and the target value. This error is then fed into three functions: proportional, integral, and derivative. The proportional term responds immediately to the error size, the integral term considers the accumulated error over time to eliminate steady-state errors, and the derivative term anticipates future changes based on the error's rate of change. By combining these actions with adjustable gains, a continuous-time

PID controller offers precise and adaptable control for various industrial processes and automation tasks. These controllers are typically designed in the mathematical framework of Laplace transforms (s-domain), making them ideal for analog electronic implementations. The block diagram representing the Continuous-time PID controller can be seen in fig 1.4.

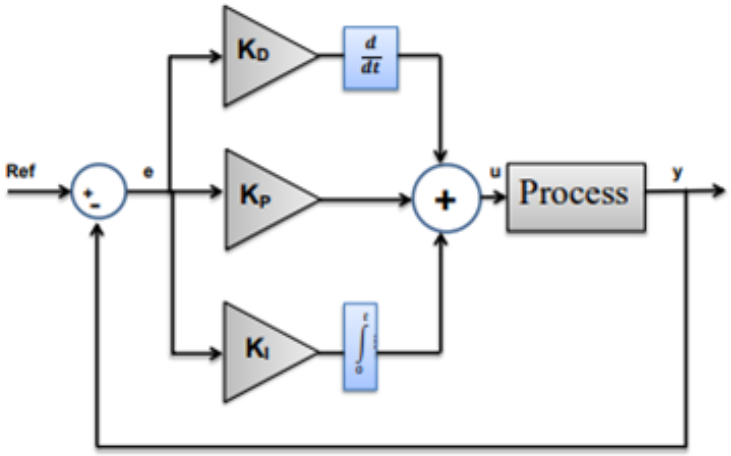

Figure 1.4: Continuous time PID Controller

1.7 Discrete PID Controller

A discrete PID controller is a digital workhorse for controlling physical systems like motors or temperature regulation. Unlike its continuous counterpart, it operates based on discrete time intervals. Imagine a movie – a continuous system captures every tiny movement, but a discrete system only captures frames at specific intervals.

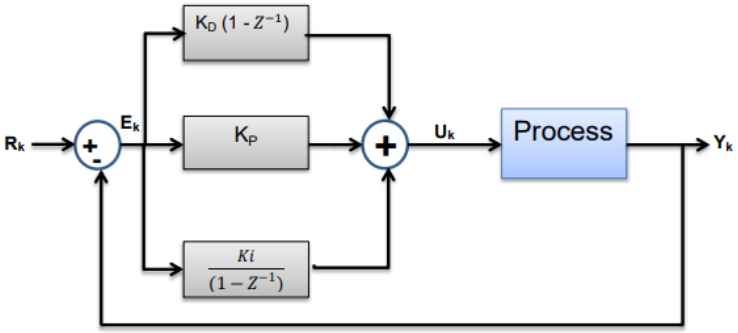

Figure 1.5: Discrete PID Diagram

The PID stands for Proportional, Integral, and Derivative – three terms that work together to adjust the system's output based on the difference (error) between the desired value (set-

point) and the actual reading. The proportional term reacts immediately to the error, the integral term considers the accumulated error over time, and the derivative term anticipates future errors based on the rate of change.

By combining these elements in a digital algorithm, the discrete PID controller calculates adjustments to be applied to the system at each time interval. This approach makes it ideal for use with micro-controllers and digital circuits, enabling precise control in various applications. The block diagram representing the discrete PID controller can be seen in Fig 1.5.

1.8 PID Tuning

All general methods for control design can be applied to PID control. Additionally, several specialized methods tailored for PID control, known as tuning methods, have been developed. Regardless of the method used, it is crucial to always consider key control elements such as load disturbances, sensor noise, process uncertainty, and reference signals.

Among the most well-known tuning methods are those developed by Ziegler and Nichols. These methods have significantly influenced PID control practice for over half a century. They rely on characterizing process dynamics with a few parameters and simple equations for controller parameters. Despite their widespread reference, these methods provide only moderately good tuning in specific scenarios. This could be attributed to their simplicity and their suitability for basic control exercises in educational settings.

Step Response Method

Ziegler and Nichols introduced a tuning method that utilizes process information obtained from an open-loop step response, often acquired through a bump test.

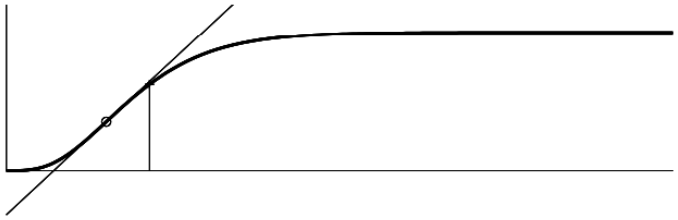

Figure 1.6: Ziegler-Nichols step response method involves analyzing and characterizing the step response behavior

This method falls under the conventional category, relying on modeling and control prin-

ciples with a simple process model. The step response is defined by two parameters, labeled as a and L, as shown in Fig 1.6.

Table 1.1: The Ziegler-Nichols step response method yields PID controller parameters

Controller	**K**	T_i	T_d	T_p
P	1/a			4L
PI	0.9/a	3L		5.7L
PID	1.2/a	2L	L/2	4L

Initially, the point where the slope of the step response reaches its maximum is identified, and a tangent is drawn at this point. The parameters a and L are determined by the intersections between this tangent and the coordinate axes. Subsequently, the controller parameters are derived from Table 1.1. Additionally, an approximation of the closed-loop system's period Tp is provided in the table.

CHAPTER 2

Software Simulation

2.1 MATLAB/ SIMULINK

It is a powerful computational and simulation environment widely used in engineering and scientific applications. It provides an integrated platform for modeling, simulating, and analyzing complex systems, making it a cornerstone tool for control system design, signal processing, and algorithm development. With its intuitive graphical interface, MATLAB/Simulink enables engineers to seamlessly translate mathematical models into simulations, facilitating the exploration and refinement of various control strategies before implementation. The icon of MATLAB/Simulink is shown in fig 2.1.

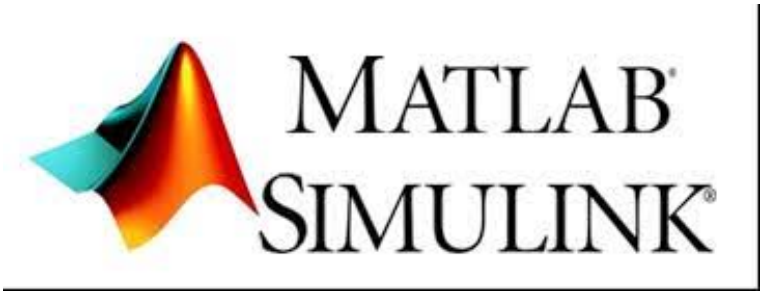

Figure 2.1: MATLAB/Simulink Icon

2.2 DC MOTOR MODELLING

The DC motor serves as a power actuator by converting electrical energy into mechanical energy. It consists of a rotating armature winding, a non-rotating armature magnetic field, and a permanent magnet that generates a magnetic field. The armature connections play a role in regulating torque, and the winding contributes to the development of varying intrinsic speeds. Traditional methods for controlling the speed of DC motors involve adjusting the armature current, modifying variable resistance in the armature or field circuit, or employing a combination of these techniques. The equivalent circuit of a DC motor is shown in fig 2.2.

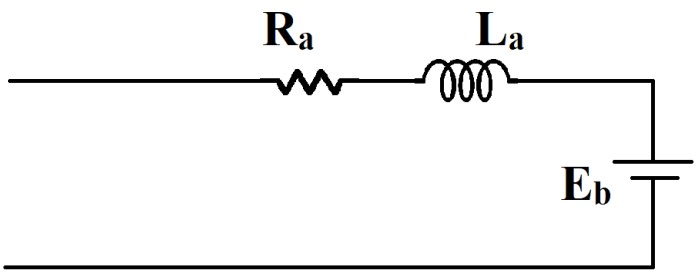

Figure 2.2: Equivalent Electrical Circuit diagram of a DC motor with constant magnetic field

Table 2.1: Parameters for DC motor model

Parameter	***Value***
Armature Resistance (R_a)	0.5 Ω
Armature Inductance (L_a)	0.1 H
Electromotive force constant $(K_e\phi)$	$1.6 N_m/A$
Moment of inertia of rotor (J)	5 $kg * m^2/s^2$
Damping Ratio of mechanical system (B)	0.01 Nms
Input voltage (V_a)	200V

The governing equations are as follows:

$$V_a = E_b + R_a \cdot I_a + L_a \cdot \frac{dI_a}{dt} \tag{2.1}$$

$$E_b = (k_e.\Phi)\omega_m \tag{2.2}$$

$$T_m = (k_e.\Phi).I_m = T_L + J.\frac{d\omega_m}{dt} + B\omega_m \tag{2.3}$$

Taking Laplace Transform of 2.4, We get

$$V_a(s) - E_b(s) = R_a.I_s(s) + s.L_a \tag{2.4}$$

$$I_a(s) = \frac{V_a(s) - E_b(s)}{R_a(1 + s.\frac{L_a}{R_a})} \tag{2.5}$$

$$I_a(s) = \frac{V_a(s) - (k_e\Phi)\omega_m}{R_a(1 + s.\tau_a)} \tag{2.6}$$

Where $\tau_a = \frac{L_a}{R_a}$ is the time constant of the armature circuit.

Taking Laplace Transform of 2.3, We get

$$T_m(s) - T_L(s) = \omega_m(s)[B + s.J] \tag{2.7}$$

$$\omega_m(s) = \frac{T_m(s) - T_L(s)}{B(1 + s.\frac{J}{B})} \tag{2.8}$$

$$\omega_m(s) = \frac{(K_e.\phi)I_a(s) - T_L(s)}{B(1 + s.\tau_m)} \tag{2.9}$$

Where $\tau_m = \frac{J}{B}$ is the mechanical time constant of the armature circuit.

Using equation 2.6 and 2.9 , We can derive the following model as shown in fig 2.4.

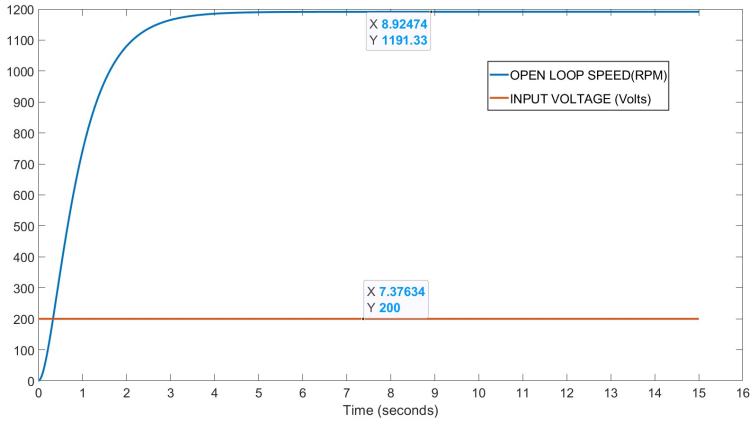

Figure 2.3: Response of DC motor model

Fig 2.3 shows the open-loop response of the DC motor model revealing that at an applied voltage of 200V, the steady-state speed stabilizes at approximately 1200 RPM.

The model of the DC motor, designed in Simulink, is depicted in fig 2.4.

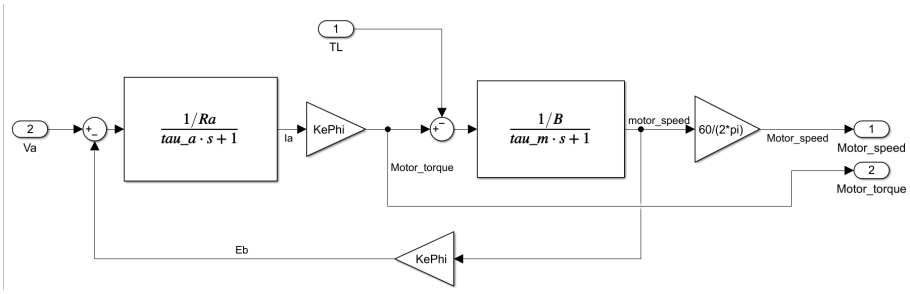

Figure 2.4: Simulink model of DC motor

2.3 Design of Continuous-time PID Controller in SIMULINK

By Combining the Proportional, Integrator & Derivative component we get the equation for Continuous-time PID equation 2.10.

$$u(t) = K_p.e(t) + K_i \int_0^t e(\tau)d\tau + \frac{K_d(de(t))}{dt} \tag{2.10}$$

Here,

e(t): Error signal

K_p : Proportional gain

K_i: Integral gain

K_d: Derivative gain

u(t): Control Signal

The Simulink model of the Continuous-time PID controller is shown in fig2.5.

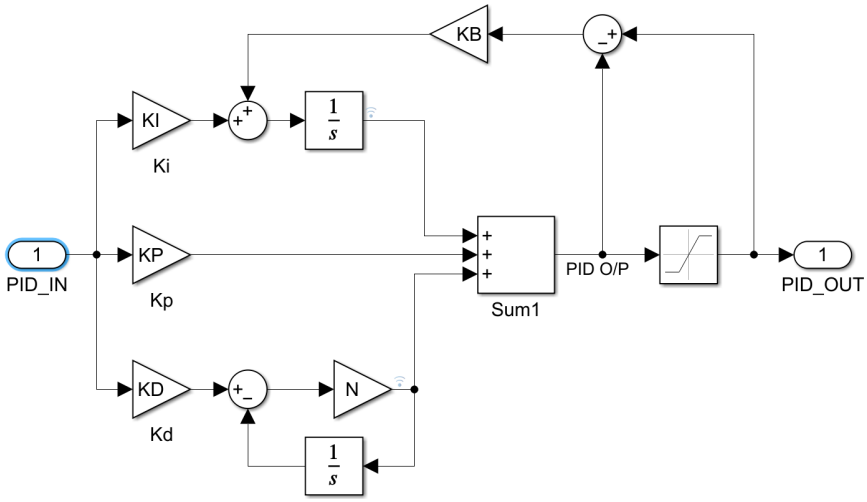

Figure 2.5: Simulink model of CT PID Controller

2.4 Design of Discrete-time PID Controller in SIMULINK

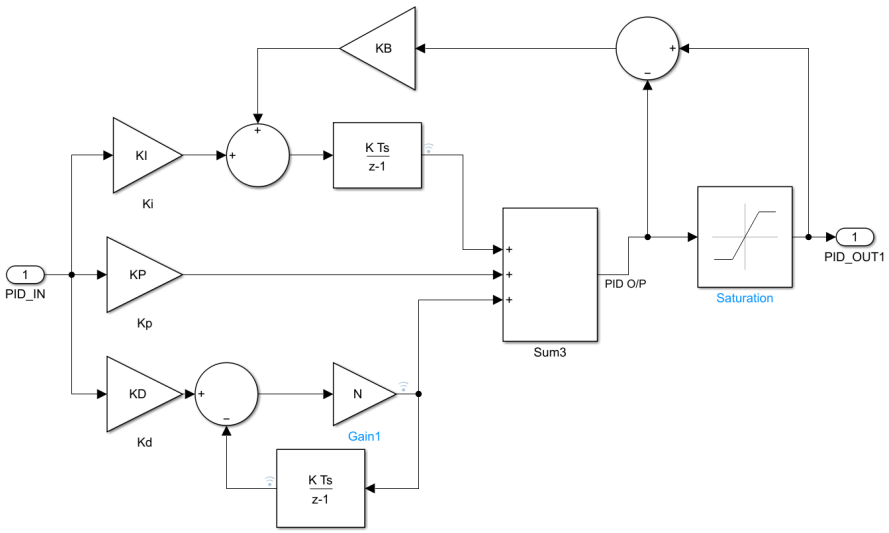

Figure 2.6: Simulink Model of Discrete PID controller with practical considerations

Taking Laplace transform of 2.10, we get

$$U(s) = K_p.E(s) - E_i\frac{E(s)}{s} + k_d.s.E(s) \tag{2.11}$$

Using Forward Euler Transformation [4]

$$s = \frac{z-1}{T_s} \tag{2.12}$$

Where T_s is the sampling time. We can write,

$$U(z) = K_p.E(z) - E_i\frac{T_s}{z-1}.E(z) + k_d.\frac{z-1}{T_s}.E(z) \tag{2.13}$$

2.4.1 PRACTICAL PID CONSIDERATIONS

Output Saturation

Output saturation in a PID controller refers to the situation where the manipulated variable, which is the output of the controller is constrained or limited within a certain range. This limitation prevents the controller from driving the system beyond predefined bounds. This can occur in various systems, such as motor control, temperature regulation, or any application where the control output must operate within specific limits.

Low Pass Filter in Derivative block

In a PID (Proportional-Integral-Derivative) controller, the derivative term is responsible for providing damping to the system response and improving stability. However, the derivative action is sensitive to high-frequency noise or rapid changes in the system input. To address this sensitivity and prevent the amplification of high-frequency noise, a low-pass filter is often introduced in the derivative block.

With a first-order low pass filter with filter coefficient N, introduced the derivative term can be given as:

In s-domain,

$$U_d(s) = s.\frac{N}{s+N}.E(s) \tag{2.14}$$

$$U_d(s) = \frac{N}{1+N.\frac{1}{s}}.E(s) \tag{2.15}$$

and in z-domain,

$$U_d(z) = \frac{N}{1 + N.\frac{T_s}{z-1}}.E(z) \tag{2.16}$$

Anti-Windup Mechanisms

Integral windup is a phenomenon that can occur in a PID (Proportional-Integral-Derivative) controller when the integral term accumulates excessively due to sustained errors in the system [3]. This typically happens when the system is subject to long-lasting disturbances or is operating at or near its limits. Integral windup can lead to performance issues and overshooting when the system eventually responds to correct the error. To overcome this problem, Anti-Windup mechanisms are used. In our case, we have used the back-calculation method to avoid integral windup.

After including the above-mentioned methods for a practical PID controller, we can derive the following Simulink Model of a PID as shown in fig 2.6.

We used the following values for PID design:

Table 2.2: Parameters for PID controller model

Parameter	***Value***
KP	10
KD	2
KI	15
N	50
KB	20
Saturation	0-200

These values are just an initial guess with some manual tweaking to get the desired result.

2.4.2 SIMULATION RESULTS OF DISCRETE PID CONTROLLER

We tested our DC motor model for an input voltage of 200V, which would give a speed of about 1200 RPM. But while considering a variable load torque, it can be seen in Fig. 6 that the open loop speed of the motor drops from 1190 RPM to around 1000 RPM and then when the load is removed it comes back to 1200 RPM. It is expected because the speed of a motor is inversely related to load torque, i.e. as load torque increases the motor speed decreases. The simulation response of the dc motor is shown in fig 2.7.

But this effect is undesirable because we desired an RPM of 1200, to overcome this we utilized PID to control the motor. As can be seen in fig 2.7 the closed-loop response is much

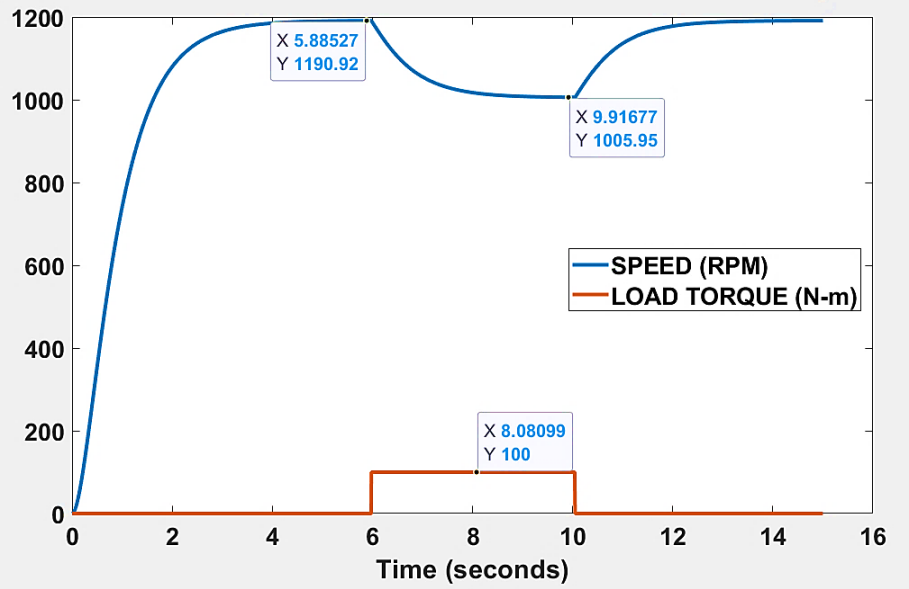

Figure 2.7: open loop response of DC motor model with load torque of 100 N-m

better as compared to the open-loop response. On comparison of both the responses (at a load of 100 N-m), we can see that, the speed of the Open-loop motor varies by 15.5% whereas for the closed-loop motor with PID, the change is only about 1.5%. This means that our PID is working fine and is able to reject disturbances and maintain the output at the desired level. The comparison of the open loop response with the closed loop response of the DC motor model with a load torque of 100 N-m is shown in fig 2.9.

The block diagram of the PID controller with the dc motor module is shown in fig 2.8.

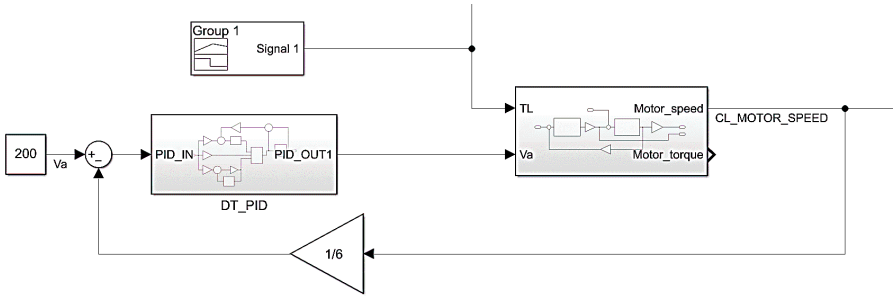

Figure 2.8: Block diagram for simulation tests

2.4.3 OTHER APPLICATIONS OF PID CONTROLLER

PID controllers play a crucial role in various applications, ensuring precise and stable control in different electronic systems. One prominent use lies in voltage regulation, where PID controllers help maintain consistent voltage levels in power supplies, ensuring the stable

and reliable operation of electronic devices. In motor speed control, PID controllers govern the rotational speed of motors, optimizing efficiency and accuracy in diverse applications such as robotics, automated machinery, and electric vehicles. Additionally, PID controllers find application in temperature control systems, where they regulate the heating or cooling elements in electronic devices or systems, preventing overheating and ensuring optimal operating conditions. Their ability to offer precise and adaptable control makes PID controllers indispensable in electronics engineering, enhancing the performance, efficiency, and reliability of various electronic systems and components.

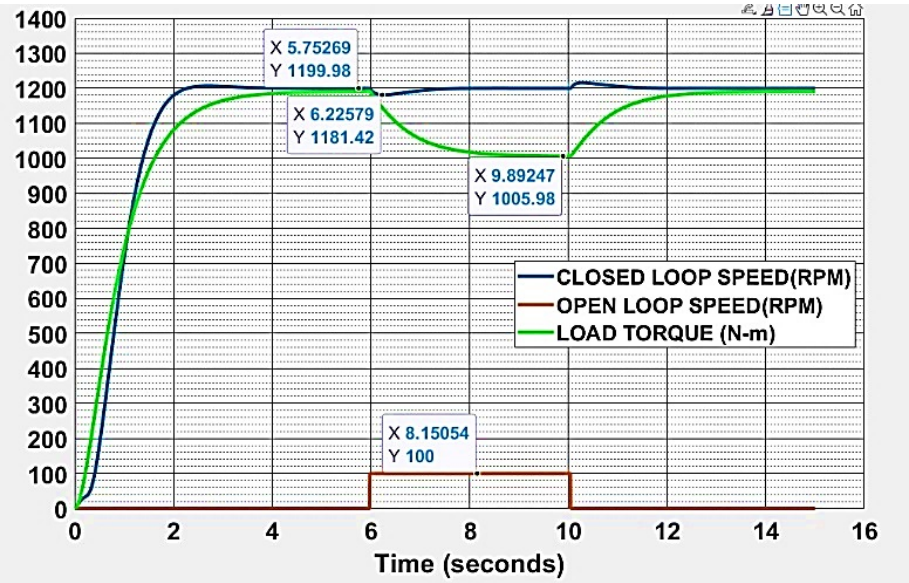

Figure 2.9: Comparison of open loop response with closed loop response of DC motor model with load torque of 100 N-m

CHAPTER 3

Hardware

3.1 Introduction of Xilinx Vivado

It stands as a leading tool in the realm of FPGA (Field-Programmable Gate Array) development. This sophisticated software suite offers comprehensive capabilities for the design, synthesis, and implementation of digital systems on FPGA platforms. Vivado supports high-level hardware description languages and enables efficient utilization of FPGA resources, making it a go-to tool for developing cutting-edge digital circuits, embedded systems, and customizable hardware accelerators.

Figure 3.1: Vivado Icon

Xilinx offers the Vivado Design Suite, a tool set for analyzing, designing, and synthesizing hardware description language (HDL) systems.

Vivado's High-Level Synthesis (HLS) compiler presents a programming environment akin to that found in general-purpose and specialized CPUs for software development. Leveraging processor compiler technologies, Vivado HLS comprehends, analyzes, and optimizes C/C++ programs.

By eliminating the need for manual RTL generation, the Vivado High-Level Synthesis compiler allows direct targeting of Xilinx devices with programs written in C, C++, and System C.

3.1.1 FPGA-Based Design Flow

The flow navigator illustrates the entire flow of the FPGA-based design, which is divided into the following six phases:

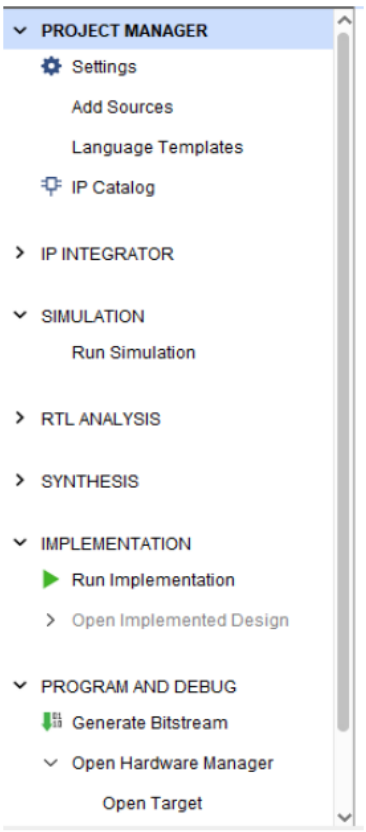

Figure 3.2: FPGA-Based Design Flow

1. **Adding Sources:** This initial step involves creating or adding essential source files, including Design source, Simulation source, and Constraints. These files are crucial for designing, simulating, and implementing the desired system on the FPGA.

2. **Simulation:** This phase allows for simulation after completing design and simulation files. It enables checking and verifying the design's functionality. If the functionality is not met, Verilog code can be edited, and the simulation can be rerun.

3. **RTL Analysis:** Following successful simulation and verification, RTL analysis provides the register-level implementation.

4. **Synthesis:** The RTL design proceeds to synthesis after RTL analysis, where the run synthesis option is selected. After synthesis completion, I/O planning is done using VIVADO's GUI in the I/O Ports section.

5. **Implementation:** Synthesis completion and I/O planning lead to implementing the design, achieved by running implementation under the Implementation menu.

6. **Program and Debug:** Following successful implementation, a Bit-stream is produced, and the Hardware Manager is initiated. The FPGA board is connected to the computer via a micro USB adapter, and within the Hardware Manager, the FPGA board is designated by selecting the target option. Once the board is chosen, the Program Device option is activated to configure the device, as signified by an inbuilt LED on the FPGA board emitting a "done" signal.

3.2 Hardware Description Language (HDL)

Hardware Description Language (HDL) serves as a programming language tailored for electronic devices, digital circuits, and electronics. It streamlines the programming of design, layout, and functionalities, covering operators, expressions, statements, inputs, and outputs. HDL compilers produce a gate map instead of a computer-executable file, which is then loaded onto programming devices to ensure the circuit operates as intended. This language is especially suited for Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs), allowing for the expression of structural, behavioral, and gate-level aspects. Verilog and VHDL stand out as the most prevalent HDLs, with Verilog being our preferred choice for project work.

3.2.1 Verilog HDL

Verilog is a hardware description language (HDL) utilized for defining digital system designs such as microprocessors, ROM, RAM, flip-flops, and various other digital hardware components. HDL is employed to describe digital hardware at any level, making it technology-independent, easy to construct, and debug. This attribute is particularly beneficial for complex circuits where HDL often proves more useful than schematics.

Verilog was specifically designed to enhance the resilience and adaptability of HDL by streamlining the design process. Consequently, it has become the most extensively used and practiced HDL in the semiconductor industry.

3.2.2 Abstraction Levels of Verilog

Verilog supports design at multiple levels of abstraction, which include:

- **Behavioral level:** This top level of abstraction in Verilog HDL allows the implementation of modules based on desired design algorithms without concerning the hardware specifics. While it accurately describes circuit functioning, it can be challenging to synthesize.
- **Data flow level:** This level defines circuit features using data flow between hardware registers, akin to a Boolean expression. It offers a structured approach to implementation.
- **Gate level:** At this level, logic gates and their connections implement the module. Components are linked via signals, resembling a schematic diagram. Component activation occurs when input signals change, with simultaneous activation of triggered components.

3.2.3 Features of Verilog Language

- Verilog is case-sensitive.
- Verilog does not require packages.
- Verilog syntax is largely based on the "C" programming language.
- Verilog allows the creation of Behavioral, Gate, and Switch-level models for digital circuits.

3.3 DC Motor

DC (Direct Current) motors are electromechanical devices that convert electrical energy into mechanical motion. In industrial settings, DC motors are utilized in machinery, conveyor systems, and manufacturing processes due to their ability to provide precise speed and position control. Overall, the versatility and reliability of DC motors make them indispensable across a diverse range of applications in both industrial and consumer contexts.

In our project, we'll provide the voltage of a 12V from a DC adapter to run a DC motor at a predetermined speed.

Figure 3.3: DC Motor

3.4 FPGA Board

An FPGA Development Board serves as a vital hardware platform for prototyping and deploying digital systems designed using FPGA technology. These boards typically integrate FPGAs, memory, input/output interfaces, and other essential components. FPGA Development Boards enable engineers and researchers to implement and test complex digital designs, accelerating the development cycle of applications ranging from signal processing and communication to image processing and embedded systems. These boards offer a versatile environment for exploring the capabilities of FPGAs and developing innovative ones. The Picture of Artix-7 Nexys 4 FPGA board is shown in fig 3.4.

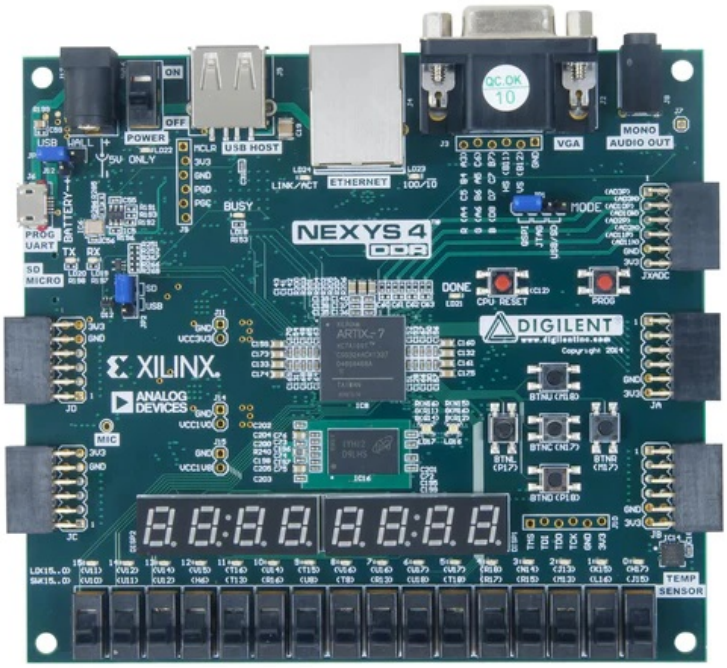

Figure 3.4: FPGA Board

3.5 L298N Motor Module

The L298N motor driver module is a workhorse for controlling DC motors in various projects. It's an integrated circuit (IC) that acts like a translator between a micro-controller and two DC motors. You provide low-voltage control signals from your micro-controller, and the L298N amplifies them to handle the higher current demands of the motors. This allows you to control the direction (forward or backward) and speed (through pulse-width modulation) of the motors independently. The L298N can handle DC motors with voltages between 5 and 35V and currents up to 2A, making it suitable for a wide range of applications from small robots to hobbyist projects. It's a popular choice due to its ease of use, affordability, and ability to drive two motors simultaneously. The image of the L298N motor module is shown in fig 3.5.

3.6 IR Sensor Module

The infrared (IR) sensor module is a workhorse for many projects involving object detection and line following. It acts like an electronic eye, emitting and detecting infrared light invisible to the human eye. When an object comes within the sensor's range, it disrupts the infrared beam. The sensor then detects this change and outputs a signal, indicating the pres-

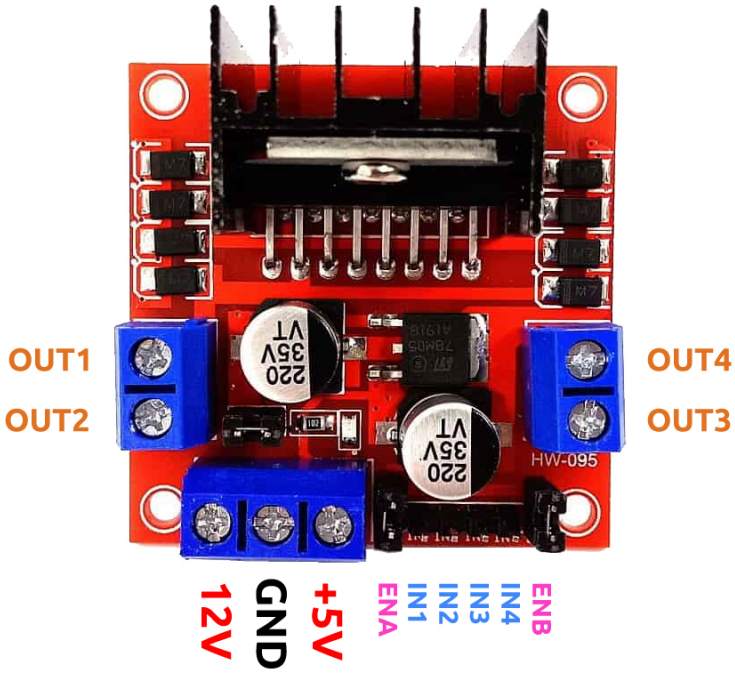

Figure 3.5: L298N Motor module

ence of an object. IR sensor modules come in various forms, but most have a transmitter and a receiver. The transmitter sends out the infrared beam, and the receiver picks it up under normal conditions. Different types of IR sensors offer diverse ranges, from a few centimeters to several meters. These versatile sensors find applications in robots for obstacle avoidance and line tracking, automated lighting systems for motion detection, and even security alarms. Their ease of use, affordability, and compact size make them popular choices for hobbyists and professionals alike. The Block diagram of the IR Sensor is shown in fig 3.6.

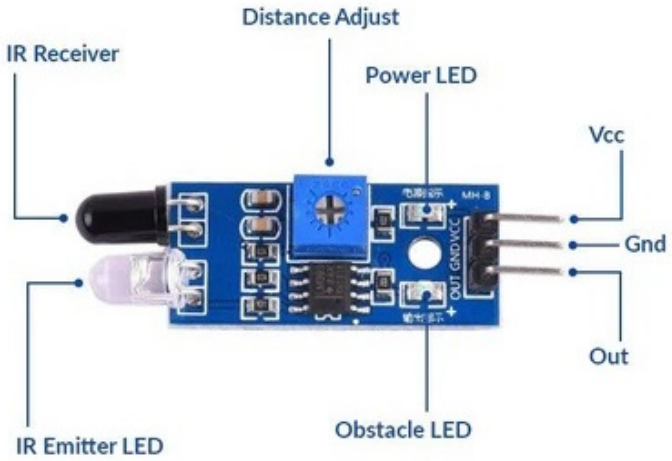

Figure 3.6: IR Sensor Module

3.7 PID Controller Module

A PID controller implemented on an FPGA is a digital control system that uses three control actions—proportional, integral, and derivative—to regulate a process or system. In FPGA implementation, the PID controller is designed using Verilog HDL. The proportional action provides an immediate response to the current error between the desired set-point and the actual system output, adjusting the control signal proportionally to the error. The integral action integrates the error signal over time, correcting for accumulated errors and eliminating steady-state errors. The derivative action anticipates the future trend of the error signal, damping oscillations, and improving system stability. FPGA-based PID controllers offer advantages such as fast processing speeds, real-time control capabilities, and flexibility for parameter tuning and customization, making them suitable for a wide range of control applications in industries such as robotics, automation, and mechatronics.

The Complete PID Controller Module RTL Schematic is shown in fig 3.7. The Complete Verilog Code has been attached in Appendix A. PID Module component proportional, Integrator, Derivative along with summation and Saturation Block. The details code has been attached in Appendix A

3.7.1 Proportional Block

The Verilog module for a proportional module involves defining the behavior of the proportional control action based on the error signal and the proportional gain. Typically, this is implemented as a multiplication operation. The proportional block is shown in fig 3.8.

3.7.2 Integrator Block

The Verilog code for an integrator module involves defining the behavior of the integration operation using discrete time steps. This can be achieved using techniques such as a discrete-time accumulator or a summing register. The Integrator block is shown in fig 3.9. The details code has been attached in Appendix A.

3.7.3 Derivative Block

The Verilog code for a derivative module involves defining the behavior of the differentiation operation using discrete time steps. This can be achieved using techniques such as finite

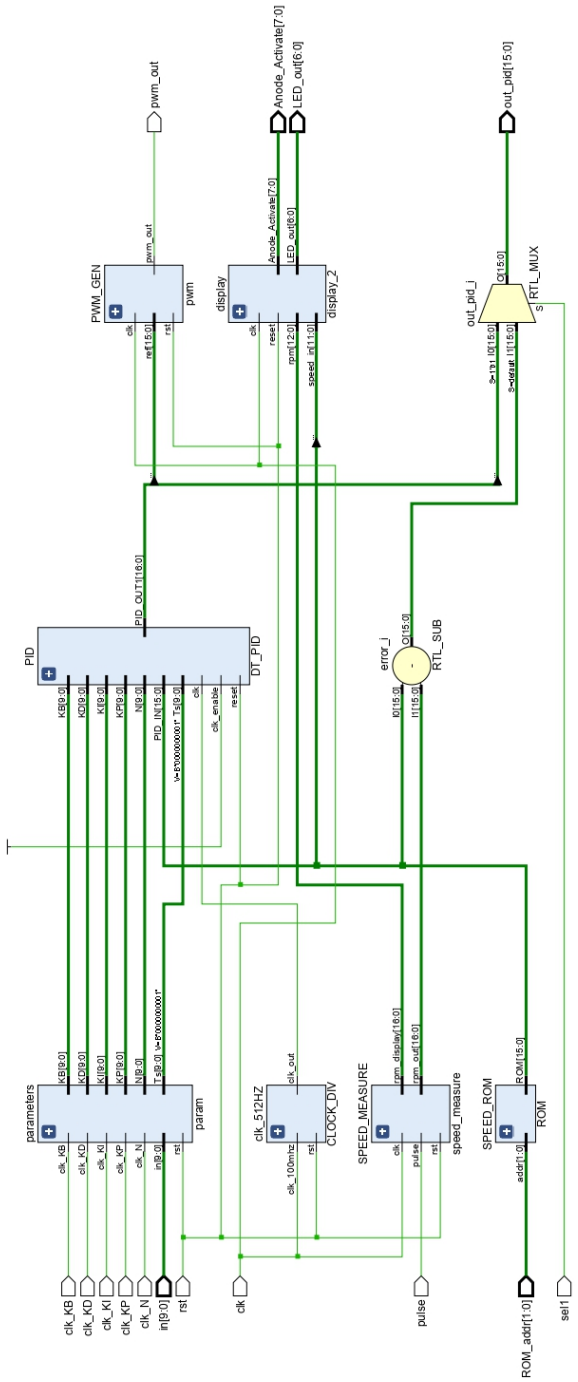

Figure 3.7: PID Controller Verilog Module

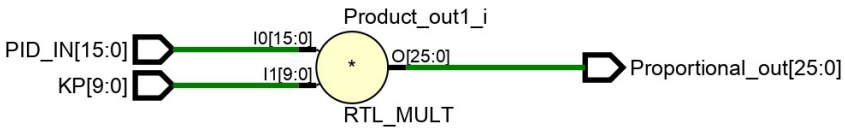

Figure 3.8: Proportional Verilog Module

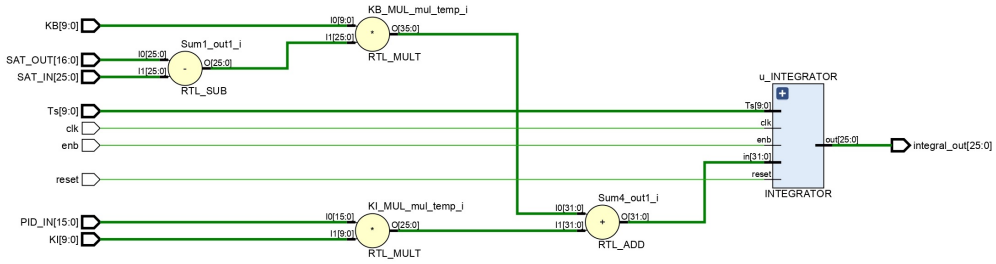

Figure 3.9: Integrator Verilog Module

difference methods or using shift registers and subtractors. In our project, we have implemented a derivative block using an integrator module with feedback. We also incorporated a low-pass filter for removing high-frequency noise. The Derivative block is shown in fig 3.10. The details code has been attached in Appendix A.

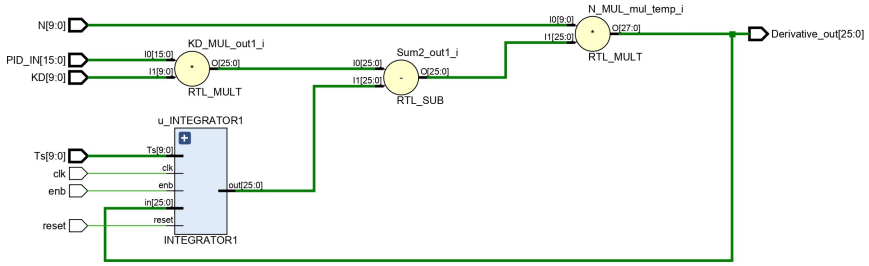

Figure 3.10: Derivative Verilog Module

3.8 PWM Generation Module

The PWM module utilizes an increasing counter to generate a saw-tooth waveform, a repetitive signal that rises linearly before resetting. This saw-tooth waveform is then compared with a reference signal, determining the PWM output. The PWM signal's frequency is precisely set at 2 kHz, ensuring a consistent and predictable modulation pattern. This method

of PWM generation is commonly employed in various applications such as motor control, power regulation, and signal modulation, where precise timing and control of pulse width are essential for system functionality and performance optimization. fig 3.11 shows the PWM generation RTL schematic. The details code has been attached in Appendix A.

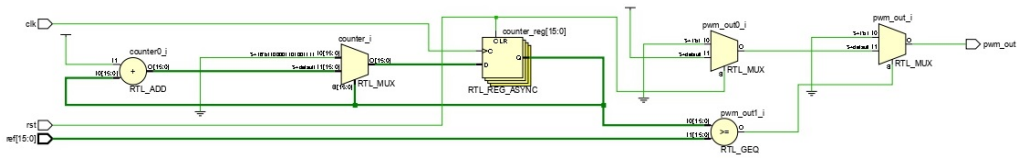

Figure 3.11: PWM Verilog Module

3.9 Speedometer

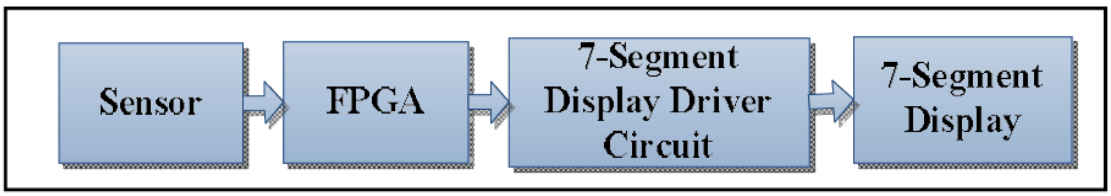

Figure 3.12: Speedometer Block Diagram

An FPGA-based speedometer is designed and implemented to display the speed of a motor vehicle in rpm. The digital speedometer operates by sensing pulses from a sensor and then calculates and displays the vehicle's speed on a 7-segment display. Key features of this speedometer include digital readout and rpm speed display. Notably, its linear response to speed changes resembles that of an analog speedometer [5]. The speedometer's measurement range spans from 0 rpm to 9999 rpm. An additional noteworthy characteristic is its ability to automatically adjust for reduced speeds, unlike other speedometers that may maintain the same reading if new pulses arrive at a slower pace over time. The Verilog RTL schematic of the speedometer is depicted in fig 3.12, and the complete Verilog code is provided in Appendix B.

3.9.1 Methodology

The proposed methodology can be summarized as follows:

1. Step 1: Design the Speedometer with two counters, labeled A and B. Counter A represents past count values, while counter B represents current count values.
2. Step 2: At the initiation of pulse counting with $x(0)$, counter A begins counting until the subsequent pulse, at which juncture it halts. Counter A denotes the duration between consecutive pulses, commencing from the first pulse's arrival: $A = x(1) - x(0)$. Counter B is initially set to 0.
3. Step 3: Upon the arrival of the first pulse, counter B begins counting and persists until the second pulse arrives. Counter B signifies the duration between the first and second pulses: $B = x(2) - x(1)$.
4. Step 4: If $A > B$, the speed exhibits a gradual decrease. Conversely, if $B > A$, the speed experiences a significant increase. Upon the arrival of the third pulse, the value of counter B is transferred to A, and the duration between the third and second pulses is recorded in B. This iterative process persists for subsequent pulses, with the nth pulse calculations unfolding as follows:
   - Counter A: $A = x(n-1) - x(n-2)$.
   - Counter B: $B = x(n) - x(n-1)$.
5. Step 5: Record the calculated values on the 7-segment display.

3.9.2 Architecture of speedometer

The Speed Measurement Flowchart visually illustrates all the steps outlined in Section 3.9.1 through a series of detailed flowcharts. Each step is elaborated upon and depicted graphically in fig 3.13 to provide a comprehensive understanding of the speed measurement process.

Fig 3.12 shows the proposed architecture of a real-time digital speedometer. The whole architecture has a sensor which used to sense the moving of the shaft of the motor, in FPGA which is the core part of the design, and a 7-segment display to view the speed in rpm.

3.10 Results and Discussion

Fig 3.15 illustrates the complete setup of the PID controller, with the main processing unit being the FPGA board. The L298N motor module is utilized for driving a 12V DC motor, with additional power supplied to the motor via a 12V DC adapter. An IR sensor is employed for speed measurement, and the results are displayed on the 7-segment display of the FPGA board. The interconnection between various components is established using jumper wires. Pin C17 is configured as an output port for PWM output signals, while pin G17 of the FPGA board is designated as an input port for pulses from the IR sensor. Switches V10 and U11 are used for addressing the ROM, where speed settings such as 800, 1000, 1200, and 1500

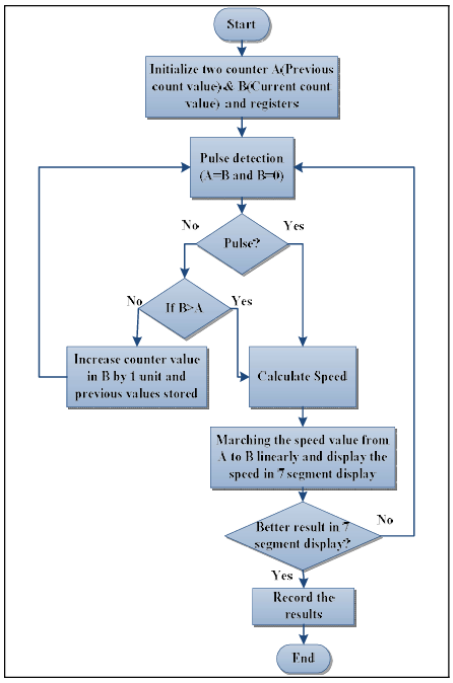

Figure 3.13: Flowchart of Speedometer

RPM are selected. Rest 10 switches are utilized for inputting values like K_P, K_D, K_I, and K_B for the PID controller. Pulsar M18 serves as the reset pin, and pulsars P17, N17, M17, and P18 are assigned for clocking purposes for K_P, K_I, K_D, and K_B, respectively.

Fig 3.16 illustrates the PWM output signal on a Digital Storage Oscilloscope (DSO). The fundamental frequency of the signal measures 2 kHz, with a duty cycle of 37.58%. This signal is generated by the PWM generation module, with detailed code provided in Appendix A.

Fig 3.17 displays the results of our project, where the power supply is provided by a 12V DC adapter. We have selected a speed of 1500 rotation per minute (RPM) from the read-only memory (ROM), and all the PID values (K_P, K_I, K_D, and K_B) have been applied through switches. The motor is operating at 1500 RPM. The left 4 LEDs indicate the reference rotation per minute (RPM), while the remaining 4 LEDs show the real-time motor speed which is the desired speed. Due to slight accuracy errors in various components, the motor is running at 1504 RPM, which is close to the reference speed set in the PID controller. So we are getting around 0.26% error in the desired speed. This error occurs because of the imperfect reading of the IR sensor, which can be attributed to its susceptibility to environmental light interference during its operation.

The values of K_p, K_d, K_i, and K_b used in this project are presented in the table 3.1.

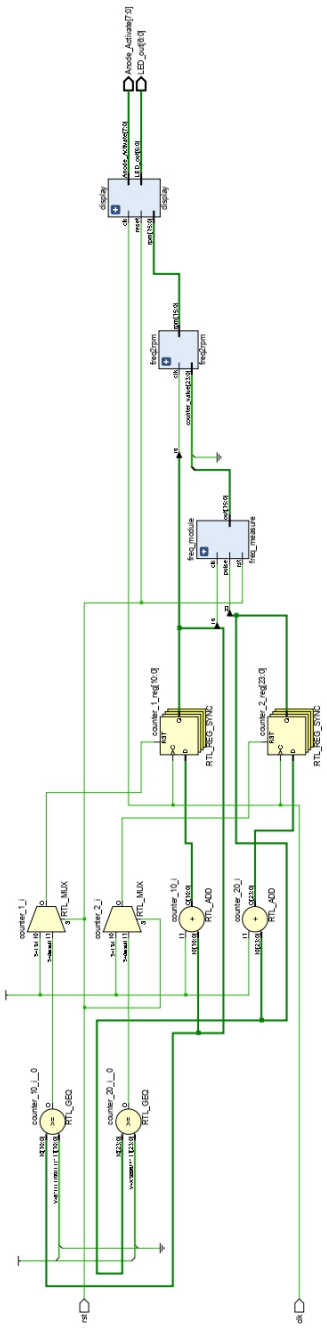

Figure 3.14: Block Diagram of Speedometer

These values can be adjusted in real-time to achieve the desired output.

Table 3.1: Value for PID controller model

Parameter	***Value***
KP	0.01171875
KD	0.009765625
KI	0.001953125
KB	0.01758125

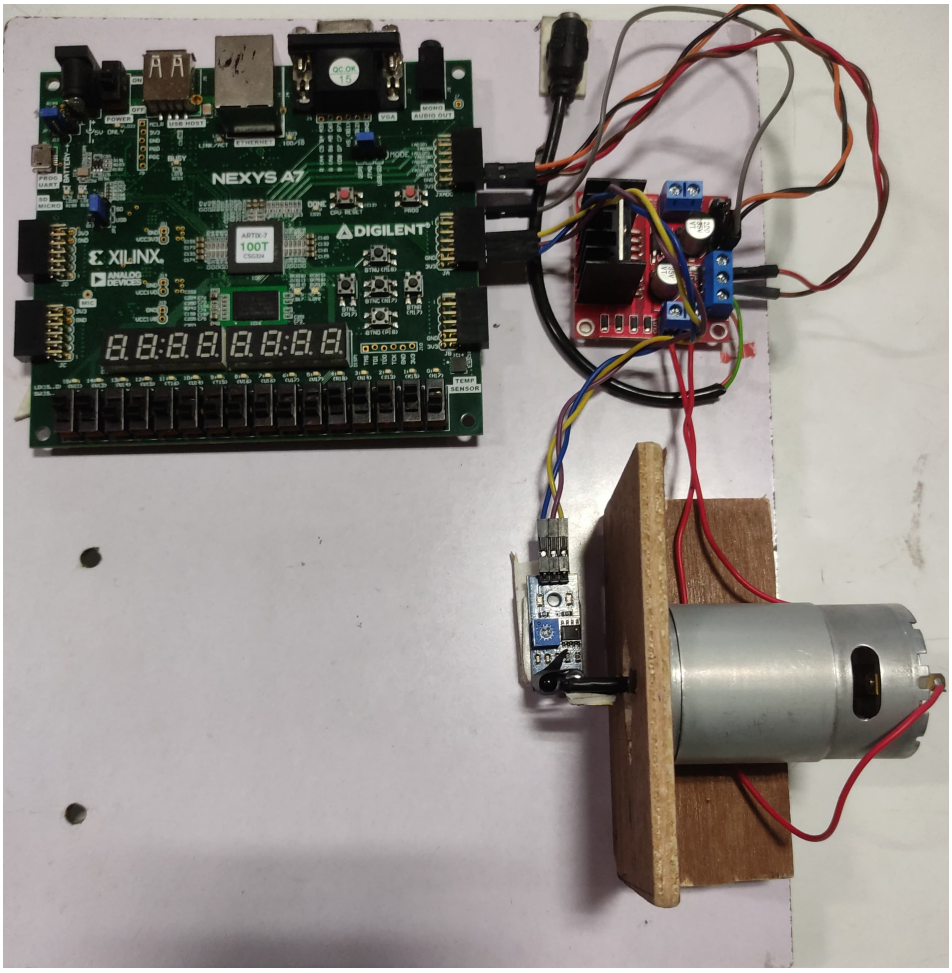

Figure 3.15: Complete set up of PID

3.11 Utilisation and Power Report

The utilization report displayed in Fig 3.18 presents the hardware usage in our project, including the number of resources utilized such as look-up tables (LUTs), registers, Digital

Figure 3.16: PWM output from FPGA

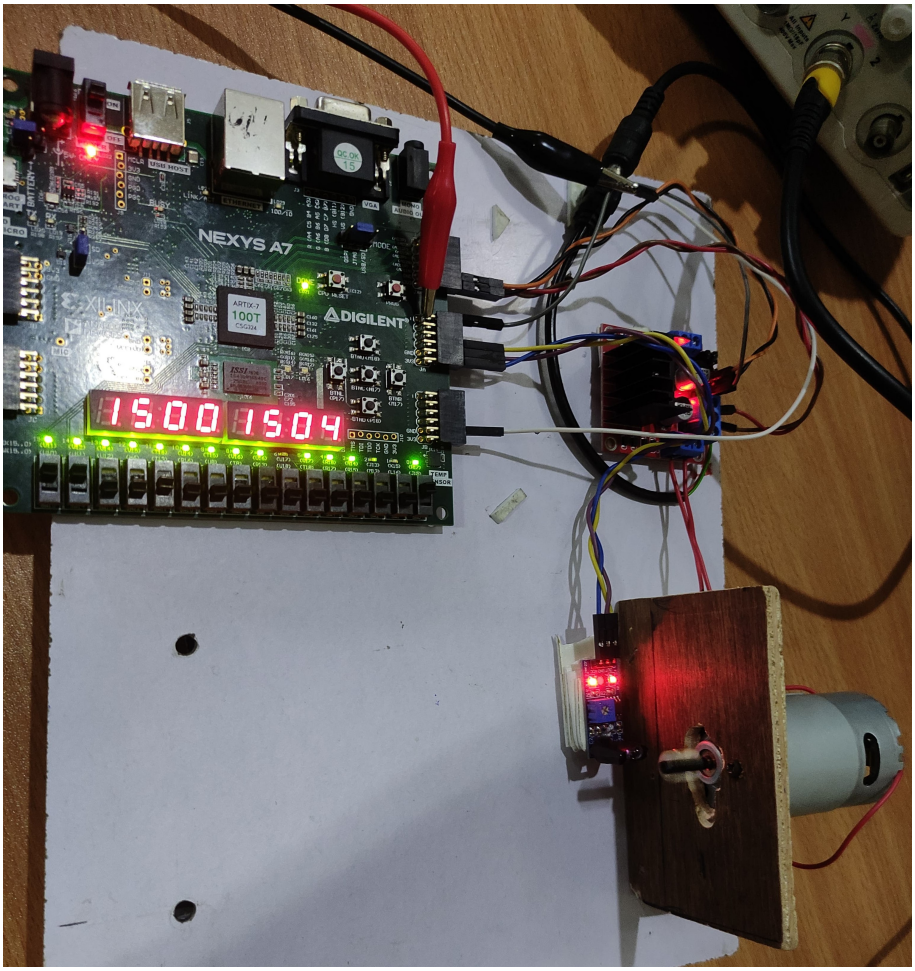

Figure 3.17: Reference (set point) and desired speed in RPM displayed on 7 segments display

signal processing block (DSPs), and input output buffer (IOBs) for the PID controller implementation.

Fig 3.19 depicts the power report of our project. It's important to note that this power analysis does not reflect actual power consumption as we have not utilized the SAIF file. The actual power consumption is expected to be significantly lower. However, from this report, we can discern the power consumption by various components: signals account for 7%, logic for 6%, DSPs for 12%, and I/O for 75%.

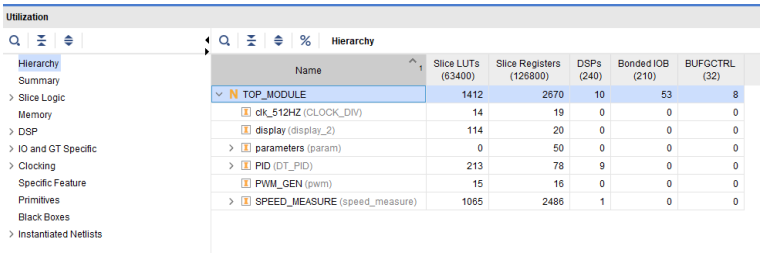

Figure 3.18: Utilization Report

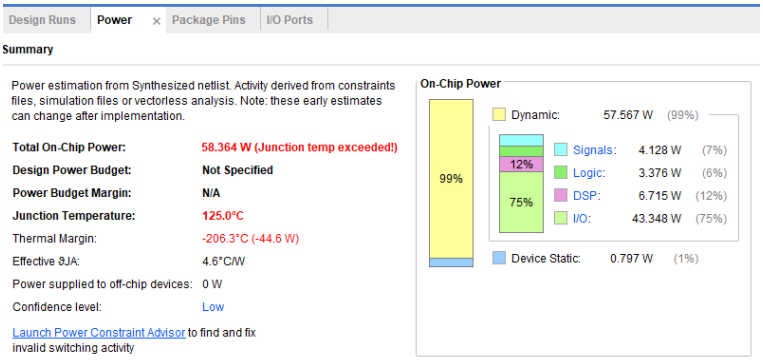

Figure 3.19: Power report

CHAPTER 4

Conclusion

This project report serves as a comprehensive overview of our efforts in digital control systems, with a specific focus on leveraging FPGA technology for implementing PID control, PWM generation, and developing a digital speedometer. Our exploration has revealed significant advancements in the realm of controlling traditional DC motors through the precise application of PID controllers and the accurate generation of PWM signals. The culmination of our work is the successful development of a digital speedometer capable of displaying the speed of a vehicle with high precision.

These accomplishments highlight the immense potential of FPGA technology in enhancing the intelligence and efficiency of control systems, particularly in sectors such as manufacturing and automotive engineering. The versatility and adaptability of FPGA-based digital control systems offer immense value in optimizing processes, improving accuracy, and enhancing overall performance.

We will focus on further refining FPGA resource utilization to maximize efficiency and exploring innovative ways to enhance user interaction with these systems. These ongoing efforts aim to push the boundaries of digital control technology, opening up new avenues for advanced applications in robotics, automation, and mechatronics.

This project focused on the design and FPGA implementation of a PID controller for the regulation of a DC motor's speed. The PID controller, which combines proportional, integral, and derivative actions, was chosen for its effectiveness in achieving optimal performance in dynamic systems. The literature review highlighted the significance of discrete PID controllers in enhancing the speed control of traditional DC motors, particularly in industrial applications. The project's progress included the mathematical modeling of the DC motor system, which involved understanding its equivalent circuit and deriving governing equations. The Simulink model of the DC motor was developed, and its response was simulated, showcasing the impact of load torque on the motor's speed. The implementation of a discrete PID controller was discussed, and practical considerations such as output saturation, low-pass filters in the derivative block, and anti-windup mechanisms were addressed. The Simulink model of the discrete PID controller, incorporating these practical aspects were

presented. Simulation results demonstrated the effectiveness of the PID controller in maintaining the desired motor speed, even in the presence of variable load torque. A comparison between open-loop and closed-loop responses showed that the PID controller significantly reduced the variation in the motor speed, indicating its ability to reject disturbances. Furthermore, the rise time of the motor was observed to increase with the implementation of the PID controller, providing an additional benefit yet to be explored. The project's findings contribute to the understanding and application of PID controllers in dynamic systems, particularly in the context of DC motor speed control.

We encountered some significant challenges, including implementing real-time speed measurement, determining the parameters of available PMDC motors, and converting the Simulink PID module to fixed-point representation due to the difficulty in determining variable ranges.

4.1 FUTURE GOALS

in this phase of the project, we have successfully developed a Simulink model for DC motor as well as a Discrete PID. For the next phase our future goals include,

1. Efficient use of hardware and power utilisation.
2. To enhance the PID controller, we can either develop a Neural Network-based PID controller or employ machine learning techniques to optimize its performance.

APPENDIX A

Verilog HDL Code for PID Controller

A.1 PID Module

```
‘timescale 1ns / 1ps
    module PID_Top_Module(
    input clk,rst,pulse,sel1,clk_KP,clk_KI,clk_KD,clk_KB,clk_N,
    input [9:0]in,
    input [1:0] ROM_addr,
    output [7:0] Anode_on,
    output [6:0] LED_glow,
    output pwm_out,
    output [15:0] out_pid   );
    wire [15:0] PID_IN,SPEED_IN,error;
    wire [16:0] PID_OUT;
    wire [9:0] KI, KP, KD, KB, Ts, N;
    wire clk_enable,ce_out,sel;
    wire [16:0] RPM,RPM_DISP;
    wire clk_512;
    ROM SPEED_ROM(ROM_addr,SPEED_IN);
    assign error = SPEED_IN - RPM;
    CLOCK_DIV #(’d97655,18)clk_512HZ(clk,rst,clk_512);
    param parameters(in,clk_KP,clk_KI,clk_KD,clk_KB,clk_N,rst,KI,
    KP,KD,KB,Ts,N);
    DT_PID PID(clk_512,rst,1,error,KI,KP,KD,KB,Ts,N,ce_out,PID_OUT);
    pwm PWM_GEN(clk,rst,PID_OUT[15:0],pwm_out);
    speed_measure SPEED_MEASURE(clk,rst,pulse,RPM,RPM_DISP);
    display_2 display(clk,rst,SPEED_IN[15:4],{RPM_DISP[16:4]},Anode_on,LED_glow);
    assign out_pid =  sel1 ? PID_OUT[16:1] : error;
    endmodule
```

A.1.1 ROM MODULE

```
‘timescale 1ns / 1ps
module ROM (input [1:0] addr,
output reg [15:0] ROM   );
always @(addr)
case(addr)
2’b00: ROM <= {12’d800,4’b0000};
2’b01: ROM <= {12’d1000,4’b0000};
2’b10: ROM <= {12’d1200,4’b0000};
2’b11: ROM <= {12’d1500,4’b0000};
endcase
endmodule
```

A.1.2 CLOCK DIVIDER MODULE

```
‘timescale 1ns / 1ps
module CLOCK_DIV #(parameter DIVIDE_VALUE = ’d999,DATA_WIDTH=26)(
input wire clk_100mhz,
input wire rst,
output reg clk_output  );
//localparam DIVIDE_VALUE = (100000000 / DESIRED_FREQ) / 2 - 1;
// Divide by half the period   // Counter and divide value
reg [DATA_WIDTH-1:0] counter_1;  // Clock divider logic
always @(posedge clk_100mhz or posedge rst) begin
    if (rst) begin
        counter_1 <= 0;
        clk_out <= 0;
    end
    else begin
        if (counter_1 == DIVIDE_VALUE) begin
            counter_1 <= 0;
            clk_output <= ~clk_output;
        end
        else begin
            counter_1 <= counter_1 + 1;
        end
    end
end
endmodule
```

A.1.3 PARAMETER MODULE

```
‘timescale 1ns / 1ps
module param(
input [9:0]in,
input clk_KP,clk_KI,clk_KD,clk_KB,clk_N,rst,
output [9:0] KI,KP,KD,KB,Ts,N    );
        register #(10)reg_KP(clk_KP,rst,1,in,KP);
        register #(10)reg_KI(clk_KI,rst,1,in,KI);
        register #(10)reg_KD(clk_KD,rst,1,in,KD);
        register #(10)reg_KB(clk_KB,rst,1,in,KB);
        register #(10)reg_N(clk_N,rst,1,in,N);
        assign Ts = 10’h001;
endmodule
```

A.1.4 Register module

```
‘timescale 1ns / 1ps
module register #(DATA_WIDTH = 8)(
input clk,rst,en,
input [DATA_WIDTH-1:0] Din,
output reg [DATA_WIDTH-1:0] Dout    );
  always @(posedge clk or posedge rst)
   if(rst)     Dout<=0;
   else if(en) Dout <=Din;
endmodule
```

A.1.5 DT PID Module

```
‘timescale 1 ns / 1 ns
module DT_PID (clk,reset, clk_enable, PID_IN, KI, KP, KD,
               KB, Ts, N, ce_out, PID_OUT1);
  input   clk;
  input   reset;
  input   clk_enable;
  input   signed [15:0] PID_IN;  // sfix16_En4
  input   signed [9:0] KI;  // sfix10_En9
  input   signed [9:0] KP;  // sfix10_En9
  input   signed [9:0] KD;  // sfix10_En9
  input   signed [9:0] KB;  // sfix10_En9
  input   signed [9:0] Ts;  // sfix10_En9
  input   signed [9:0] N;  // sfix10_En5
  output  ce_out;
  output  signed [16:0] PID_OUT1;  // sfix17_En8
  wire signed [25:0] Proportional_Module_out1;  // sfix26_En8
  wire signed [25:0] Derivative_Module_out1;  // sfix26_En8
  wire signed [25:0] PID_O_P;  // sfix26_En8
  wire signed [16:0] Saturation_out1;  // sfix17_En8
  wire signed [25:0] Integral_Module_out1;  // sfix26_En8
  wire signed [25:0] Sum3_stage2_add_temp;  // sfix26_En8
  wire signed [26:0] Sum3_op_stage1;  // sfix27_En8
  wire signed [25:0] Sum3_stage3_add_cast;  // sfix26_En8
  wire signed [16:0] dtc_out;  // sfix17_En8
  Proportional_Module u_Proportional_Module
  (.PID_IN(PID_IN), // sfix16_En4
     .KP(KP),  // sfix10_En9
     .Proportional_out(Proportional_Module_out1)  // sfix26_En8  );
  Derivative_Module u_Derivative_Module
  (.clk(clk),
    .reset(reset),
    .enb(clk_enable),
    .PID_IN(PID_IN),  // sfix16_En4
    .KD(KD),  // sfix10_En9
    .N(N),  // sfix10_En5
    .Ts(Ts),  // sfix10_En9
    .Derivative_out(Derivative_Module_out1)  // sfix26_En8   );
  Integral_Module u_Integral_Module
  (.clk(clk),
    .reset(reset),
    .enb(clk_enable),
    .SAT_IN(PID_O_P),  // sfix26_En8
    .SAT_OUT(Saturation_out1),  // sfix17_En8
    .PID_IN(PID_IN),  // sfix16_En4
     .KI(KI),  // sfix10_En9
     .KB(KB),  // sfix10_En9
     .Ts(Ts),  // sfix10_En9
     .integral_out(Integral_Module_out1)  // sfix26_En8     );
  assign Sum3_stage2_add_temp = Integral_Module_out1 + Proportional_Module_out1;
  assign Sum3_op_stage1 = {Sum3_stage2_add_temp[25],
  Sum3_stage2_add_temp};
  assign Sum3_stage3_add_cast = Sum3_op_stage1[25:0];
```

```
  assign PID_O_P = Sum3_stage3_add_cast + Derivative_Module_out1;
  assign dtc_out = ((PID_O_P[25] == 1'b0) && (PID_O_P[24:16]
  != 9'b000000000) ? 17'sb01111111111111111 :
              ((PID_O_P[25] == 1'b1) && (PID_O_P[24:16]
              != 9'b111111111) ? 17'sb10000000000000000 :
              $signed(PID_O_P[16:0])));
assign Saturation_out1 = (dtc_out >
  17'sb01111101000000000 ?
  17'sb01111101000000000 :
  (dtc_out < 17'sb00000000000000000 17'sb00000000000000000:dtc_out));
  assign PID_OUT1 = Saturation_out1;
  assign ce_out = clk_enable;
endmodule
```

A.1.6 Proportional Module

```
`timescale 1 ns / 1 ns
module Proportional_Module (PID_IN, KP, Proportional_out );
  input   signed [15:0] PID_IN;  // sfix16_En4
  input   signed [9:0] KP;  // sfix10_En4
  output  signed [25:0] Proportional_out;  // sfix26_En8
  wire signed [25:0] Product_out1;  // sfix26_En8
  assign Product_out1 = PID_IN * KP;
  assign Proportional_out = Product_out1;
endmodule
```

A.1.7 Integrator Module

```
`timescale 1 ns / 1 ns
module Integral_Module
          (clk,  reset, enb, SAT_IN, SAT_OUT,  PID_IN, KI, KB, Ts, integral_out  );
  input   clk;
  input   reset;
  input   enb;
  input   signed [25:0] SAT_IN;  // sfix26_En8
  input   signed [16:0] SAT_OUT;  // sfix17_En8
  input   signed [15:0] PID_IN;  // sfix16_En4
  input   signed [9:0] KI;  // sfix10_En4
  input   signed [9:0] KB;  // sfix10_En4
  input   signed [9:0] Ts;  // sfix10_En9
  output  signed [25:0] integral_out;  // sfix26_En8
  wire signed [25:0] Sum1_sub_cast;  // sfix26_En8
  wire signed [25:0] Sum1_out1;  // sfix26_En8
  wire signed [35:0] KB_MUL_mul_temp;  // sfix36_En12
  wire signed [31:0] KB_MUL_out1;  // sfix32_En8
  wire signed [25:0] KI_MUL_mul_temp;  // sfix26_En8
  wire signed [31:0] KI_MUL_out1;  // sfix32_En8
  wire signed [31:0] Sum4_out1;  // sfix32_En8
  wire signed [25:0] INTEGRATOR_out1;  // sfix26_En8
  assign Sum1_sub_cast = {{9{SAT_OUT[16]}}, SAT_OUT};
  assign Sum1_out1 = Sum1_sub_cast - SAT_IN;
  assign KB_MUL_mul_temp = KB * Sum1_out1;
  assign KB_MUL_out1 = KB_MUL_mul_temp[35:4];
```

```
  assign KI_MUL_mul_temp = PID_IN * KI;
  assign KI_MUL_out1 = {{6{KI_MUL_mul_temp[25]}}, KI_MUL_mul_temp};
  assign Sum4_out1 = KB_MUL_out1 + KI_MUL_out1;
  INTEGRATOR u_INTEGRATOR (.clk(clk),
                           .reset(reset),
                           .enb(enb),
                           .in(Sum4_out1),  // sfix32_En8
                           .Ts(Ts),  // sfix10_En9
                           .out(INTEGRATOR_out1)  // sfix26_En8 );
  assign integral_out = INTEGRATOR_out1;
endmodule

`timescale 1 ns / 1 ns
module INTEGRATOR (clk, reset, enb, in, Ts, out   );
  input   clk;
  input   reset;
  input   enb;
  input   signed [31:0] in;  // sfix32_En8
  input   signed [9:0] Ts;  // sfix10_En9
  output  signed [25:0] out;  // sfix26_En8
  reg signed [31:0] Delay_out1;  // sfix32_En8
  wire signed [41:0] Ts_MUL_mul_temp;  // sfix42_En17
  wire signed [31:0] Ts_MUL_out1;  // sfix32_En8
  wire signed [25:0] Sum_out1;  // sfix26_En8
  reg signed [25:0] Delay1_out1;  // sfix26_En8
  wire signed [25:0] Sum_add_cast;  // sfix26_En8
  always @(posedge clk or posedge reset)
    begin : Delay_process
      if (reset == 1'b1) begin
        Delay_out1 <= 32'sb00000000000000000000000000000000;
      end
      else begin
        if (enb) begin
          Delay_out1 <= in;
        end
      end
    end
  assign Ts_MUL_mul_temp = Ts * Delay_out1;
  assign Ts_MUL_out1 = Ts_MUL_mul_temp[40:9];
  always @(posedge clk or posedge reset)
    begin : Delay1_process
      if (reset == 1'b1) begin
        Delay1_out1 <= 26'sb00000000000000000000000000;
      end
      else begin
        if (enb) begin
          Delay1_out1 <= Sum_out1;
        end
      end
    end
  assign Sum_add_cast = Ts_MUL_out1[25:0];
  assign Sum_out1 = Sum_add_cast + Delay1_out1;
  assign out = Sum_out1;
```

```
endmodule
```

A.1.8 Derivative Module

```
‘timescale 1 ns / 1 ns
module Derivative_Module (clk, reset, enb, PID_IN, KD, N, Ts, Derivative_out );
  input   clk;
  input   reset;
  input   enb;
  input   signed [15:0] PID_IN;  // sfix16_En4
  input   signed [9:0] KD;  // sfix10_En4
  input   signed [9:0] N;  // sfix10_En2
  input   signed [9:0] Ts;  // sfix10_En9
  output  signed [25:0] Derivative_out;  // sfix26_En8
  wire signed [25:0] KD_MUL_out1;  // sfix26_En8
  wire signed [25:0] N_MUL_out1;  // sfix26_En8
  wire signed [25:0] INTEGRATOR1_out1;  // sfix26_En8
  wire signed [25:0] Sum2_out1;  // sfix26_En8
  wire signed [35:0] N_MUL_mul_temp;  // sfix36_En10  assign KD_MUL_out1 = PID_IN * KD;
  INTEGRATOR1 u_INTEGRATOR1 (.clk(clk),
                             .reset(reset),
                             .enb(enb),
                             .in(N_MUL_out1),  // sfix26_En8
                             .Ts(Ts),  // sfix10_En9
                             .out(INTEGRATOR1_out1)  // sfix26_En8
                             );
  assign Sum2_out1 = KD_MUL_out1 - INTEGRATOR1_out1;
  assign N_MUL_mul_temp = N * Sum2_out1;
  assign N_MUL_out1 = N_MUL_mul_temp[27:2];
  assign Derivative_out = N_MUL_out1;
endmodule

‘timescale 1 ns / 1 ns
module INTEGRATOR1 (clk, reset, enb, in, Ts, out  );
  input   clk;
  input   reset;
  input   enb;
  input   signed [25:0] in;  // sfix26_En8
  input   signed [9:0] Ts;  // sfix10_En9
  output  signed [25:0] out;  // sfix26_En8
  reg signed [25:0] Delay_out1;  // sfix26_En8
  wire signed [35:0] Ts_MUL_out1;  // sfix36_En17
  wire signed [25:0] Sum_out1;  // sfix26_En8
  reg signed [25:0] Delay1_out1;  // sfix26_En8
  wire signed [25:0] Sum_add_cast;  // sfix26_En8
  always @(posedge clk or posedge reset)
    begin : Delay_process
      if (reset == 1’b1) begin
        Delay_out1 <= 26’sb00000000000000000000000000;
      end
      else begin
        if (enb) begin
          Delay_out1 <= in;
```

```
        end
      end
    end
  assign Ts_MUL_out1 = Ts * Delay_out1;
  always @(posedge clk or posedge reset)
    begin : Delay1_process
      if (reset == 1'b1) begin
        Delay1_out1 <= 26'sb00000000000000000000000000;
      end
      else begin
        if (enb) begin
          Delay1_out1 <= Sum_out1;
        end
      end
    end
  assign Sum_add_cast = Ts_MUL_out1[34:9];
  assign Sum_out1 = Sum_add_cast + Delay1_out1;
  assign out = Sum_out1;
endmodule
```

A.1.9 PWM Generation Module

```
`timescale 1ns / 1ps
module pwm( input clk,rst, input [15:0] ref, output pwm_out   );
   wire [15:0]sawtooth ;
   reg[15:0] counter_1 = 0;
   always @(posedge clk or posedge rst)
    begin
        if(rst==1) counter_1 <= 0;
        else begin
            if(counter_1 == 'd49999)  counter_1 <= 0;
            else  counter_1 <= counter_1 + 1;
                end
    end
assign sawtooth = counter_1;
assign pwm_out = (counter_1 >= ref ) ? 0 :(rst?0:1);
endmodule
```

APPENDIX B

Verilog HDL Code for Speed Measurement Module

B.1 Verilog Code for Top Module

```
‘timescale 1ns / 1ps
 module speed_measure ( input clk,rst,output [7:0] Anode_on, output  [6:0] LED_on   );
    wire [15:0] freq_out;
    wire out1;
    wire [15:0] rpm;
    reg [10:0] =0;
    wire clk_50k;
    reg [23:0] counter_2=0;
    wire clk_12;
    always @(posedge clk)
    begin
        if(rst==1)
             <= 0;
        else begin
            if( >= ’d1999)
                  <= 0;
            else
                counter_1 <= counter_1 + 1;
        end
    end
   assign clk_50k = counter_1[10];
    always @(posedge clk)
    begin
        if(rst==1)
            counter_2 <= 0;
        else begin
            if(counter_2 >= ’d9999999)
                 counter_2 <= 0;
            else
                counter_2 <= counter_2 + 1;
        end
    end
    assign clk_12 = counter_2[23];
    freq_measure freq_module(clk_50k,rst,clk_12,freq_out);
    freq2rpm freq2rpm(clk_50k,{8’b0,freq_out},rpm,out1);
    display display(clk,rpm,rst,Anode_on,LED_on);
endmodule
```

B.1.1 frequency measure Module

```
‘timescale 1ns / 1ps
module freq_measure #(DATA_WIDTH = 16)(input clk,rst,pulse,output [DATA_WIDTH-1:0]out);
```

```
    wire enA,enB;
    wire rstA,rstB;
    wire [DATA_WIDTH-1:0] CountA, CountB, count;
    counter #(DATA_WIDTH)
            countA(clk,rstA,enA,CountA);
    counter #(DATA_WIDTH)
            countB(clk,rstB,!enA,CountB);
    control CTRL(clk,rst,pulse,enA,enB,rstA,rstB );
    assign   count = enA ? CountA : CountB;
    register #(DATA_WIDTH)
    regout(pulse,rst,1,count,out);
endmodule
```

B.1.2 Counter Module

```
‘timescale 1ns / 1ps
module counter#(DATA_WIDTH = 12)(input clk, rst, en,output reg [DATA_WIDTH-1:0]count_1);
    always @(posedge clk or posedge rst)
    if(rst)
    count_1<=0;
    else if(en)
     count_1<=count_1+1;
endmodule
```

B.1.3 Frequency to RPM Module

```
‘timescale 1ns / 1ps
module freq2rpm(
    input clk,
    input  [23:0] counter_value,
    output  [15:0] rpm,
    output out1    );
    reg [23:0] f_c =  ’d50000;
    wire  [37:0] ans;
    wire  [31:0] Q;
    div_gen_0 Div_1 (
  .aclk(clk),     // input wire aclk
  .s_axis_divisor_tvalid(1),  // input wire s_axis_divisor_tvalid
  .s_axis_divisor_tdata(counter_value), //[23:0] divisor_tdata
  .s_axis_dividend_tvalid(1), //input wire s_axis_dividend_tvalid
  .s_axis_dividend_tdata(f_c),  //[23:0] s_axis_dividend_tdata
  .m_axis_dout_tvalid(out1),    //output wire m_axis_dout_tvalid
  .m_axis_dout_tdata(Q)  );     //output wire [31:0] dout_tdata
 assign ans = 6’d60 * Q;
 assign rpm = ans[19:4];
endmodule
```

B.1.4 Display Module

```
‘timescale 1ns/1ps
module display_2( input clk,
    input reset, // reset
```

```
input [11:0] speed_in,
input [12:0] rpm,
output reg [7:0] Anode_on,
output reg [6:0] LED_on
);
reg [3:0] LED_BCD;
reg [19:0] refresh_counter_1; // refresh rate 380Hz
wire [2:0] LED_activating_counter_1;
always @(posedge clk or posedge reset)
begin
    if(reset==1) refresh_counter_1 <= 0;
    else  refresh_counter_1 <= refresh_counter_1 + 1;
end
assign LED_activating_counter_1 = refresh_counter_1[19:17];
always @(*)
begin
    case(LED_activating_counter_1)
    3'b000: begin
        Anode_on = 8'b01111111; //activate LED_1 and Deactivate LED_2, LED_3, LED_4
        LED_BCD = speed_in / 1000;  // the first digit of the 16-bit number
          end
    3'b001: begin
        Anode_on = 8'b10111111;  // activate LED_2 and Deactivate LED1, LED_3, LED_4
        LED_BCD = (speed_in % 1000)/100; // the second digit of the 16-bit number
          end
    3'b010: begin
        Anode_on = 8'b11011111;  // activate LED_3 & Deactivate LED_2, LED_1, LED_4
        LED_BCD = ((speed_in % 1000)%100)/10; //the third digit of the 16-bit number
            end
    3'b011: begin
        Anode_on = 8'b11101111; // activate LED_4 and Deactivate LED_2, LED_3, LED1
        LED_BCD = ((speed_in % 1000)%100)%10; //the fourth digit of the 16-bit no.
           end
    3'b100: begin
        Anode_on = 8'b11110111; // activate LED_4 and Deactivate LED_2, LED_3, LED1
        LED_BCD = rpm / 1000; // the fourth digit of the 16-bit number
           end
     3'b101: begin
        Anode_on = 8'b11111011; // activate LED_4 and Deactivate LED_2, LED_3, LED1
       LED_BCD = (rpm % 1000)/100; // the fourth digit of the 16-bit number
           end
    3'b110: begin
        Anode_on = 8'b11111101;  // activate LED_4 and Deactivate LED_2, LED_3, LED1
         LED_BCD = ((rpm % 1000)%100)/10; // the fourth digit of the 16-bit number
           end
    3'b111: begin
        Anode_on = 8'b11111110;  //activate LED_4 and Deactivate LED_2, LED_3, LED1
       LED_BCD = ((rpm % 1000)%100)%10; //the fourth digit of the 16-bit number
           end
    endcase
end
always @(*)
begin
```

```
        case(LED_BCD)
        4'b0000: LED_on = 7'b0000001; // "0"
        4'b0001: LED_on = 7'b1001111; // "1"
        4'b0010: LED_on = 7'b0010010; // "2"
        4'b0011: LED_on = 7'b0000110; // "3"
        4'b0100: LED_on = 7'b1001100; // "4"
        4'b0101: LED_on = 7'b0100100; // "5"
        4'b0110: LED_on = 7'b0100000; // "6"
        4'b0111: LED_on = 7'b0001111; // "7"
        4'b1000: LED_on = 7'b0000000; // "8"
        4'b1001: LED_on = 7'b0000100; // "9"
        default: LED_on = 7'b0000001; // "0"
        endcase
    end
endmodule
```

REFERENCES

[1] S. Balamurugan and A. Umarani, "Study of discrete pid controller for dc motor speed control using matlab," in *2020 International Conference on Computing and Information Technology (ICCIT-1441)*, 2020, pp. 1–6.

[2] J. Wang, M. Li, W. Jiang, Y. Huang, and R. Lin, "A design of fpga-based neural network pid controller for motion control system," *Sensors*, vol. 22, no. 3, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/3/889

[3] A. Visioli, *Practical PID Control*, second edition ed. City, State (optional): Germany, Springer London, 2006.

[4] M. Fadali and A. Visioli, *Digtial Control Engineering: Analysis and design*, second edition ed. City, State (optional): U.S.A, Academic Press, 2012.

[5] A. Adhikary, S. Saha, and R. Sarker, "Real time design & implementation of digital speedometer on fpga," *International journal of innovative research and development*, vol. 2, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:60752693