TERM PROJECT REPORT
FOR
ASIC AND FPGA LAB (21VL683)

Name : M. Shanmukha Sri Datta

Roll Number : CB.EN.P2VLD24007

FPGA Based Incremental PID-PWM Controller

Abstract:

Achieving precise motor position control is crucial in applications such as robotics, automation, and industrial systems. However, maintaining accuracy and stability while minimizing overshoot is challenging due to system nonlinearities and external disturbances. Traditional PID controllers, although effective, often require significant memory and computational resources, limiting their efficiency in embedded systems. To address these challenges, this project implements an **Incremental PID-PWM Controller on the Basys 3 FPGA** for closed-loop motor position control, ensuring accurate and responsive actuation with reduced computational complexity. The proposed design consists of three key components: an **Incremental PID Controller** developed in **C++ using Vivado HLS**, which computes the required PWM duty cycle for position correction based on the error between the desired and actual position. The controller operates in incremental mode with **scaled integer computations**, optimizing FPGA resource utilization while maintaining accuracy. The **PWM Generator**, implemented as a **Vivado IP Block**, dynamically generates adjustable PWM signals based on the PID output to drive the motor, ensuring smooth actuation with minimal response delay. The **Control Block**, designed in **Verilog as an RTL module**, synchronizes the PID and PWM blocks by generating appropriate enable signals based on system

conditions, ensuring sequential operation and improving system stability and efficiency. By leveraging the modularity and high-speed processing capabilities of the **Basys 3 FPGA**, this approach optimizes resource utilization, enhances real-time system stability, and provides precise motor position control. The implementation demonstrates the feasibility of FPGA-based incremental PID controllers for embedded motor control applications requiring **high efficiency, reduced computational overhead, and real-time responsiveness**.

Keywords: *Incremental PID Controller, PWM Generator, FPGA, Basys 3, Vivado HLS, Verilog, Motor Position Control, Embedded Systems, Real-time Control, Closed-loop System.*

Introduction:

Precise motor position control is a fundamental requirement in various engineering applications, including robotics, industrial automation, and embedded motion control systems. The ability to accurately control a motor's position while minimizing overshoot, response time, and steady-state error is critical for ensuring optimal system performance. However, real-world challenges such as system nonlinearities, load variations, and external disturbances can significantly affect control accuracy and stability. Traditional control methods, particularly the **Proportional-Integral-Derivative (PID) controller**, are widely used for motor control due to their simplicity and effectiveness.

A conventional PID controller operates using three fundamental components:

- The **Proportional (P) term**, which generates a control output based on the current position error.
- The **Integral (I) term**, which accumulates past errors to reduce steady-state error.
- The **Derivative (D) term**, which predicts future errors and improves transient response.

Despite its advantages, a direct implementation of a **continuous PID controller** requires high precision floating-point calculations, which significantly increase the computational burden. Moreover, tuning PID gains for optimal performance in dynamic environments can be complex, requiring iterative adjustments. Additionally, standard PID implementations suffer from limitations such as overshoot, slow response time, and excessive power consumption in real-time motor control applications.

To overcome these challenges, this project proposes an **Incremental PID-PWM Controller using**

FPGA based Design for closed-loop motor position control. The incremental PID algorithm is designed to reduce computational complexity by performing **scaled integer computations** instead of floating-point operations, making it more efficient for FPGA implementation. Unlike traditional PID controllers, which compute the absolute control signal, an **incremental PID controller** computes only the change in control output based on the variation in error, reducing memory and processing requirements.

Methodology:

The proposed Block -based design consists of three key components:

1. **Incremental PID Controller**: Developed using **Vivado HLS (High-Level Synthesis) in C++**, this block computes the PWM duty cycle required to correct the motor position error. By implementing an incremental update mechanism, it optimizes resource utilization while maintaining accurate position control.
2. **PWM Generator**: Implemented as a **Vivado IP block**, this module generates a pulse-width modulated (PWM) signal based on the duty cycle output from the PID controller. This ensures smooth motor actuation and precise speed control, minimizing torque ripple and response delay.
3. **Control Block**: Designed as a **Register-Transfer Level (RTL) block in Verilog**, this module synchronizes the PID computation and PWM generation processes, ensuring smooth and stable system operation. It manages the sequential execution of tasks by generating appropriate enable signals based on system conditions.

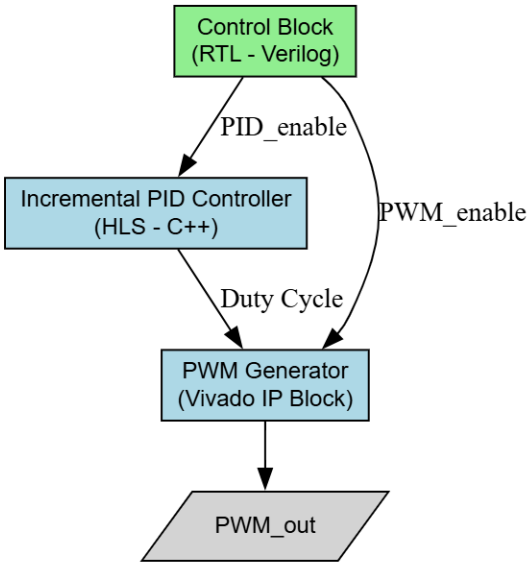
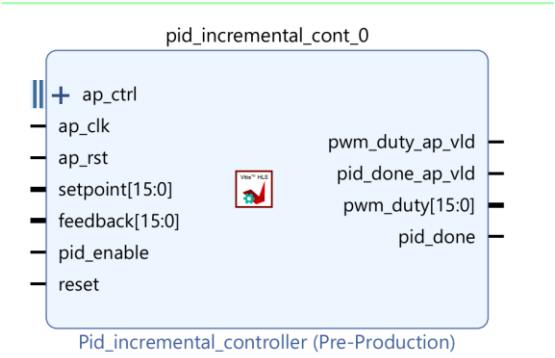


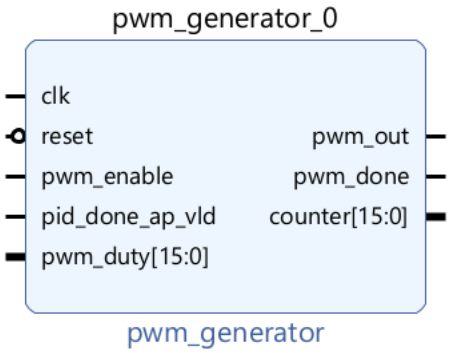
Fig 1 : Flow Chart of the project

HLS Block Design:



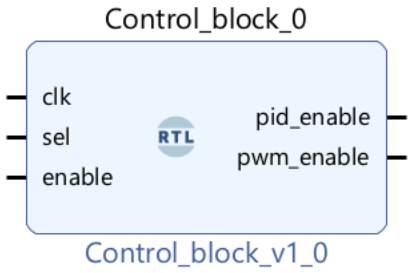
This Incremental PID Controller IP is designed to adjust the speed or position of a system smoothly and efficiently. It monitors the difference between the desired value (setpoint) and the actual value (feedback) and makes small corrections to bring them closer together. Instead of recalculating everything from scratch, it only updates the necessary changes, making it faster and more efficient for real-time control. The controller uses three tuning factors—**proportional (Kp), integral (Ki), and derivative (Kd)**—which determine how aggressively it responds to errors. The computed correction is applied to adjust the PWM duty cycle, which controls a motor or actuator, ensuring the value remains within a safe range (0 to 255). A reset function clears stored values when needed, and an enable flag allows control activation or deactivation. The design ensures smooth adjustments, prevents sudden jumps, and operates efficiently in FPGA-based systems, offering a balance between performance and hardware efficiency for real-time applications.

Vivado IP Block Design:



The **PWM Generator IP** creates a **Pulse Width Modulated (PWM) signal** based on the given duty cycle value. It operates by toggling the **pwm_out** signal, keeping it **high** for the duration specified by **pwm_duty** and **low** for the remaining cycle. A **counter** tracks the duty cycle, ensuring accurate pulse generation. The **pwm_done** signal indicates when a full PWM cycle is completed. The module includes a **reset function** to initialize all values and a **pwm_enable** control to start or stop the operation.

RTL Block Design:



The **Control Block IP** manages the activation of the **PID controller** and **PWM generator** based on input control signals. It continuously monitors the **enable** and **sel** signals to determine when to start or stop these components. When **enable** is active and **sel** is high, the block asserts both **pid_enable** and **pwm_enable**, allowing the system to perform control operations. If **sel** is low or **enable** is inactive, both outputs are deactivated, ensuring the system remains idle.

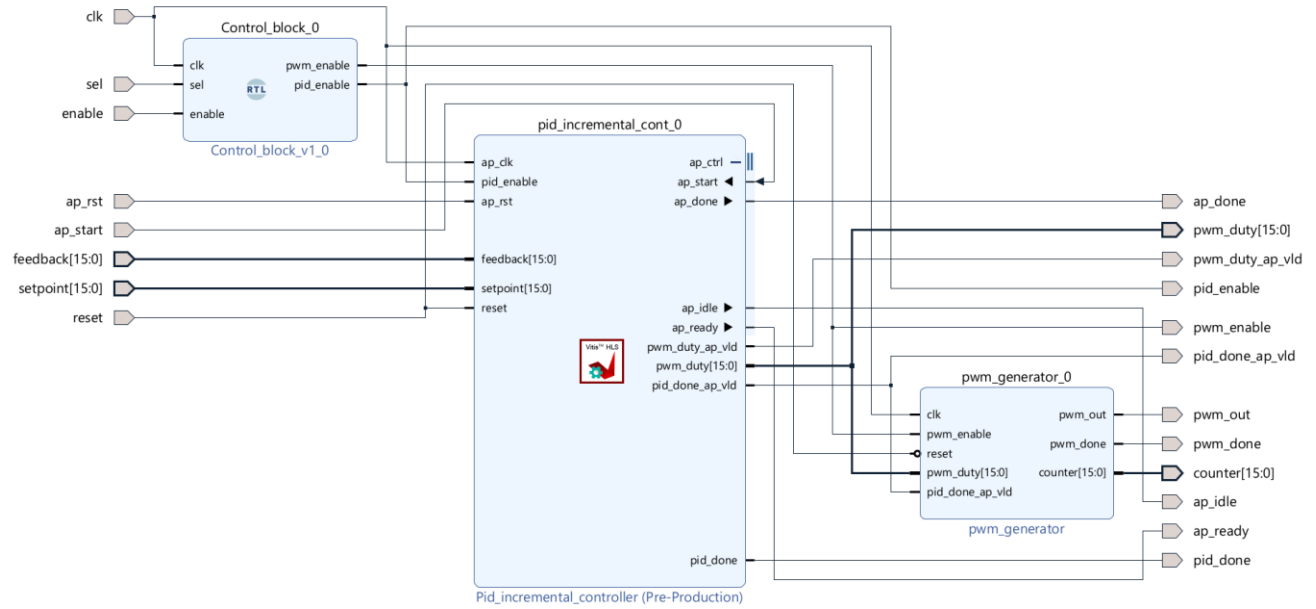
RTL Code of the Control Block (RTL Block):

```
module Control_block (
   input  clk,         // Clock signal
   input  sel,         // Select input
   input  enable,      // Enable signal
   output reg pid_enable,   // PID enable signal
   output reg pwm_enable    // PWM enable signal
);

always @(posedge clk) begin
   if (enable) begin
      if (sel) begin
         pid_enable <= 1;
         pwm_enable <= 1;
      end
      else begin
         pid_enable <= 0;
         pwm_enable <= 0;
      end
   end
   else begin
      pid_enable <= 0;
      pwm_enable <= 0;
   end
end

endmodule
```

Architecture for Incremental PID-PWM Controller Block Design:



Formula of Incremental PID is given below:

$$\Delta u(n) = K_p \cdot [e(n) - e(n-1)] + K_i \cdot e(n) + K_d \cdot [e(n) - 2e(n-1) + e(n-2)]$$

Where:

- $\Delta u(n)$ → Incremental change in control output.
- K_p → Proportional gain.
- K_i → Integral gain.
- K_d → Derivative gain.
- $e(n)$ → Error at current step:

$$e(n) = \text{setpoint} - \text{feedback}(n)$$

- $u(n)$ → Updated control output:

$$u(n) = u(n-1) + \Delta u(n)$$

HLS Code for PID Incrmental Controller:

```
#include <ap_int.h>
```

```
#define Kp 100  // Scaled Proportional Gain (1.0 * 100)
#define Ki 1    // Scaled Integral Gain (0.01 * 100)
#define Kd 10   // Scaled Derivative Gain (0.1 * 100)

typedef ap_int<16> data_t;

void pid_incremental_controller(
    data_t setpoint,   // Desired speed/position
    data_t feedback,   // Measured speed/position
    bool pid_enable,   // Enable PID computation
    bool reset,        // Reset signal
    data_t &pwm_duty,  // PID output mapped to PWM duty cycle
    bool &pid_done     // Signals completion of PID computation
) {
    static data_t prev_error1 = 0;  // Error at n-1
    static data_t prev_error2 = 0;  // Error at n-2
    static data_t prev_pwm_duty = 0; // Previous PWM duty value

    if (reset) {
        prev_error1 = 0;
        prev_error2 = 0;
        prev_pwm_duty = 0;
        pwm_duty = 0;
        pid_done = 0;
    }
    else if (pid_enable) {
        // Calculating the current error
        data_t error = setpoint - feedback;

        // Incremental PID computation with integer scaling (divide by 100 to restore scale)
```

```
        data_t delta_u = ((Kp * (error - prev_error1)) +
                          (Ki * error) +
                          (Kd * (error - 2 * prev_error1 + prev_error2))) / 100;

        // Updating the PWM duty cycle incrementally
        pwm_duty = prev_pwm_duty + delta_u;

        // Clamping the PWM duty cycle to prevent overflow or underflow
        if (pwm_duty > 255) pwm_duty = 255; // Max limit (8-bit PWM resolution)
        else if (pwm_duty < 0) pwm_duty = 0; // Min limit

        // Updating previous values for next iteration
        prev_error2 = prev_error1;
        prev_error1 = error;
        prev_pwm_duty = pwm_duty;

        pid_done = 1;  // Indicating PID computation completion
    }
    else {
        pid_done = 0;
    }
}
```

Test Bench:

```
#include <iostream>
#include <ap_int.h>

typedef ap_int<16> data_t;

void pid_incremental_controller(
    data_t setpoint,   // Desired position/speed
```

```
    data_t feedback,   // Measured position/speed
    bool pid_enable,   // Enable PID computation
    bool reset,        // Reset signal
    data_t &pwm_duty,  // PID output mapped to PWM duty cycle
    bool &pid_done     // Signals completion of PID computation
);

int main() {
    // Variables initialization
    data_t setpoint = 100;   // Desired target position
    data_t feedback = 90;    // Initial measured position
    data_t pwm_duty = 0;     // Initial PWM duty cycle
    bool pid_done = false;   // PID done flag

    std::cout << "=== Incremental PID Controller Test for Two Cases ===" << std::endl;

    // 1. Reset the PID Controller
    std::cout << "\n[RESET] Initializing PID..." << std::endl;
    pid_incremental_controller(setpoint, feedback, false, true, pwm_duty, pid_done);
    std::cout << "After reset -> PWM Duty: " << pwm_duty << " PID Done: " << pid_done <<
std::endl;

    // 2. First Test Case: setpoint = 100, feedback = 90
    std::cout << "\n[Cycle 1] Running Incremental PID Controller (Setpoint: 100, Feedback: 90)..."
<< std::endl;
    pid_incremental_controller(setpoint, feedback, true, false, pwm_duty, pid_done);
    std::cout << "Cycle 1 -> PWM Duty: " << pwm_duty << " PID Done: " << pid_done <<
std::endl;

    // 3. Second Test Case: setpoint = 100, feedback = 95
    feedback = 95; // Update feedback value
```

```
    pwm_duty = 0;  // Reset PWM duty cycle
    pid_done = false; // Reset PID done flag

    std::cout << "\n[Cycle 2] Running Incremental PID Controller (Setpoint: 100, Feedback: 95)..."
<< std::endl;
    pid_incremental_controller(setpoint, feedback, true, false, pwm_duty, pid_done);
    std::cout << "Cycle 2 -> PWM Duty: " << pwm_duty << " PID Done: " << pid_done <<
std::endl;

    return 0;
}
```

RTL Code of the Architecture:

```
module Final_Controller_wrapper
   (ap_done,
    ap_idle,
    ap_ready,
    ap_rst,
    ap_start,
    clk,
    counter,
    enable,
    feedback,
    pid_done,
    pid_done_ap_vld,
    pid_enable,
    pwm_done,
    pwm_duty,
    pwm_duty_ap_vld,
    pwm_enable,
```

```
  pwm_out,
  reset,
  sel,
  setpoint);
output ap_done;
output ap_idle;
output ap_ready;
input ap_rst;
input ap_start;
input clk;
output [15:0]counter;
input enable;
input [15:0]feedback;
output pid_done;
output pid_done_ap_vld;
output pid_enable;
output pwm_done;
output [15:0]pwm_duty;
output pwm_duty_ap_vld;
output pwm_enable;
output pwm_out;
input reset;
input sel;
input [15:0]setpoint;

wire ap_done;
wire ap_idle;
wire ap_ready;
wire ap_rst;
wire ap_start;
wire clk;
```

```
wire [15:0]counter;
wire enable;
wire [15:0]feedback;
wire pid_done;
wire pid_done_ap_vld;
wire pid_enable;
wire pwm_done;
wire [15:0]pwm_duty;
wire pwm_duty_ap_vld;
wire pwm_enable;
wire pwm_out;
wire reset;
wire sel;
wire [15:0]setpoint;

Final_Controller Final_Controller_i
    (.ap_done(ap_done),
     .ap_idle(ap_idle),
     .ap_ready(ap_ready),
     .ap_rst(ap_rst),
     .ap_start(ap_start),
     .clk(clk),
     .counter(counter),
     .enable(enable),
     .feedback(feedback),
     .pid_done(pid_done),
     .pid_done_ap_vld(pid_done_ap_vld),
     .pid_enable(pid_enable),
     .pwm_done(pwm_done),
     .pwm_duty(pwm_duty),
     .pwm_duty_ap_vld(pwm_duty_ap_vld),
```

```
        .pwm_enable(pwm_enable),
        .pwm_out(pwm_out),
        .reset(reset),
        .sel(sel),
        .setpoint(setpoint));
endmodule
```

RTL Test Bench for Architecture:

```
module Final_Controller_wrapper_tb;

  // Inputs
  reg ap_rst;
  reg ap_start;
  reg clk;
  reg reset;
  reg sel;                  // Added sel for controlling pid_enable and pwm_enable
  reg [15:0] setpoint;
  reg [15:0] feedback;

  // Outputs
  wire ap_done;
  wire ap_idle;
  wire ap_ready;
  wire pid_done;
  wire pid_done_ap_vld;
  wire pwm_done;
  wire pwm_duty_ap_vld;
  wire pwm_out;
  wire [15:0] counter;
```

```
wire [15:0] pwm_duty;

// Internal Counter for Counting Cycles
integer cycle_count;

// Instantiate the Final_Controller_wrapper module
Final_Controller_wrapper DUT (
  .ap_done(ap_done),
  .ap_idle(ap_idle),
  .ap_ready(ap_ready),
  .ap_rst(ap_rst),
  .ap_start(ap_start),
  .clk(clk),
  .counter(counter),
  .enable(1'b1),          // Enable always set to 1
  .feedback(feedback),
  .pid_done(pid_done),
  .pid_done_ap_vld(pid_done_ap_vld),
  .pid_enable(),          // Internal
  .pwm_done(pwm_done),
  .pwm_duty(pwm_duty),
  .pwm_duty_ap_vld(pwm_duty_ap_vld),
  .pwm_enable(),          // Internal
  .pwm_out(pwm_out),
  .reset(reset),
  .sel(sel),              // Fixed sel control
  .setpoint(setpoint)
);

// Clock Generation: 10 ns period => 100 MHz clock
always #15 clk = ~clk;  // 30 ns clock period for 33.33 MHz
```

```
// Reset and Initial Setup
initial begin
  clk = 0;
  ap_rst = 1;
  ap_start = 0;
  sel = 1;  // Enable PID and PWM
  reset = 1;
  setpoint = 0;
  feedback = 0;
  cycle_count = 0;

  // Apply reset for 20 ns and release
  #80;
  ap_rst = 0;
  reset = 0;

  // Test Case 1: Setpoint = 100, Feedback = 90
  $display("Starting Case 1: Setpoint = 100, Feedback = 90");
  @(posedge clk);
  #5;  // Small delay for stability
  setpoint = 16'd100;
  feedback = 16'd90;
  ap_start = 1;
  @(posedge clk);
  #10 ap_start = 0;

  // Waiting until pwm_done goes high
  cycle_count = 0;
  while (!pwm_done) begin
    @(posedge clk);
```

```
    cycle_count = cycle_count + 1;
  end
  @(posedge clk);  // Wait for one more clock after pwm_done

  $display("Case 1 Completed. Counter Value: %d, Cycle Count: %d, PWM Done: %b, PID Done: %b, PWM Duty: %d",
           counter, cycle_count, pwm_done, pid_done, pwm_duty);

  $display("Testbench Completed Successfully!");
  $finish;
 end
 initial begin
  $monitor("Time = %0t ns, PWM_Out = %b, PWM_Done = %b, PID_Done = %b, Counter = %d, Setpoint = %d, Feedback = %d, PWM Duty = %d, sel = %b",
           $time, pwm_out, pwm_done, pid_done, counter, setpoint, feedback, pwm_duty, sel);
 End
```

Results:

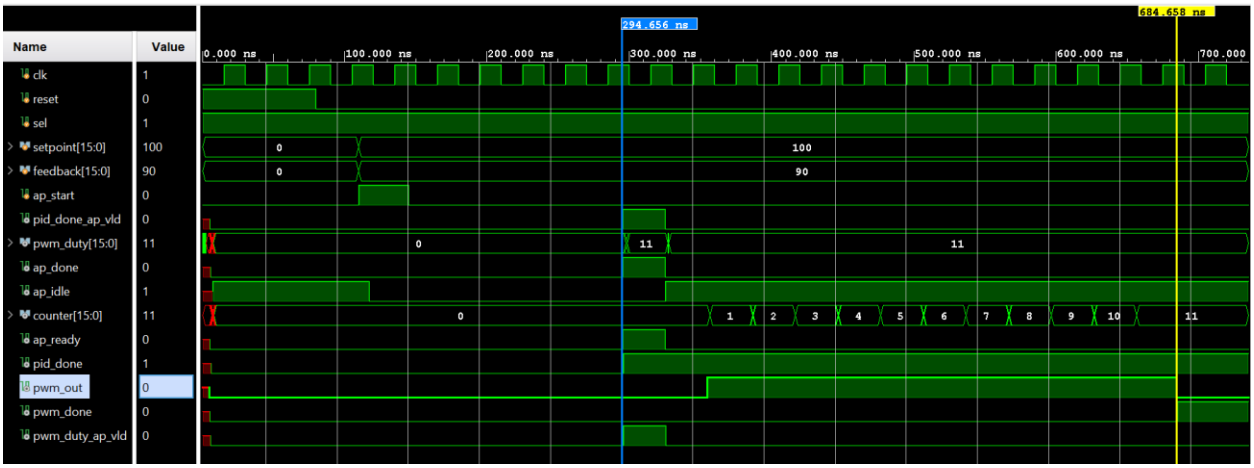
C Simulation Output:

[RESET] Initializing PID...
After reset -> PWM Duty: 0 PID Done: 0

[Cycle 1] Running Incremental PID Controller...
Cycle 1 -> PWM Duty: 11 PID Done: 1

[Cycle 2] Running Incremental PID Controller with Updated Feedback...
Cycle 2 -> PWM Duty: 5 PID Done: 1

Post Implementation Timing Simulation:



FPGA Output:

Conclusion:

The Incremental PID-PWM Controller using FPGA Based Design developed in this project successfully integrates an Incremental PID Controller (implemented using Vivado HLS), a PWM Generator (as a Vivado IP block), and a Control Block (in Verilog RTL) to achieve efficient motor position control. The design focuses on reducing computational complexity by using incremental PID calculations, making it suitable for FPGA-based real-time applications.

References:

[1] M. Liu, H. Zhang, Y. Zhang, and C. Yuan, "Design and Performance Analysis of ZYNQ Based Incremental PID-PWM Controller," *2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*, Changchun, China, 2022, pp. 1123–1126. DOI: 10.1109/EEBDA53927.2022.9744964.

[2] M. Tahmid Wara Ucchas, M. Mustafiz Nuhas, M. Toufiquzzaman, A. Jaber Mahmud, and M. Fokhrul Islam, "Performance and Comparative Analysis of PI and PID Controller-based Single Phase PWM Inverter Using MATLAB Simulink for Variable Voltage," *2022 Second International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, Bhilai, India, 2022, pp. 1–6. DOI: 10.1109/ICAECT54875.2022.9807857.

[3] Xilinx (AMD), *Vivado High-Level Synthesis (HLS) User Guide*, UG902 (v2024.1), January 2024. Available at: https://www.xilinx.com.

[4] Wikipedia, "Proportional–Integral–Derivative Controller," last modified March 30, 2024. Available at: https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller.

[5] Wikipedia, "Pulse-Width Modulation," last modified March 30, 2024. Available at: https://en.wikipedia.org/wiki/Pulse-width_modulation.

[6] Quora, "What Is the Difference Between Pulse-Width Modulation (PWM) and Proportional-Integral-Derivative (PID) Control?" Accessed April 2, 2025. Available at: https://www.quora.com/What-is-the-difference-between-pulse-width-modulation-PWM-and-proportional-integral-derivative-PID-control.

[7] Hackaday, "Generating a Control PWM Signal from a PID Controller," *Night Light Control PID Battery Charger*, January 10, 2023. Available at: https://hackaday.io/project/184690-night-light-control-pid-battery-charger/log/204542-7-generating-a-control-pwm-signal-from-a-pid-controller.

[8] R. Nane, V. M. Sima, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.