DC Motor

Creating your project

In this tutorial, you will learn how to implement position and speed control of a DC motor on an FPGA using the ChipInventor platform. We will use several preconfigured blocks (Verilog modules) to build a feedback system based on an encoder (measuring the motor shaft position) and pulse counters for defining position/velocity setpoints. Then, a controller generates a PWM signal to adjust the motor speed and rotation direction.

Steps to get started:

1. Open **ChipInventor**.

2. Click on **New Project**.

3. Fill out the fields:

   - **Name:** DC Motor Control
   - **Description:** Project for DC motor position and speed control with an encoder and PWM
   - **Type:** FPGA

4. Click **Create** to create the project.

Project Assembly Steps

2.1 System Inputs

In this step, we identify the essential input signals that enable the motor control system’s operation.

a) Synchronization Clock (clk)

- **Description:** Clock signal that synchronizes all operations in the digital system, ensuring all blocks operate in a coordinated manner.
- **Connection:** The clk signal must be connected to the main blocks that require precise timing, such as encoder, debouncer, pulse_count16, and pwm_control.

b) Encoder Inputs (IO57 and IO56)

- **Description:** Quadrature signals from the encoder used to determine the motor’s current position.
  - IO57 → quadA signal
  - IO56 → quadB signal

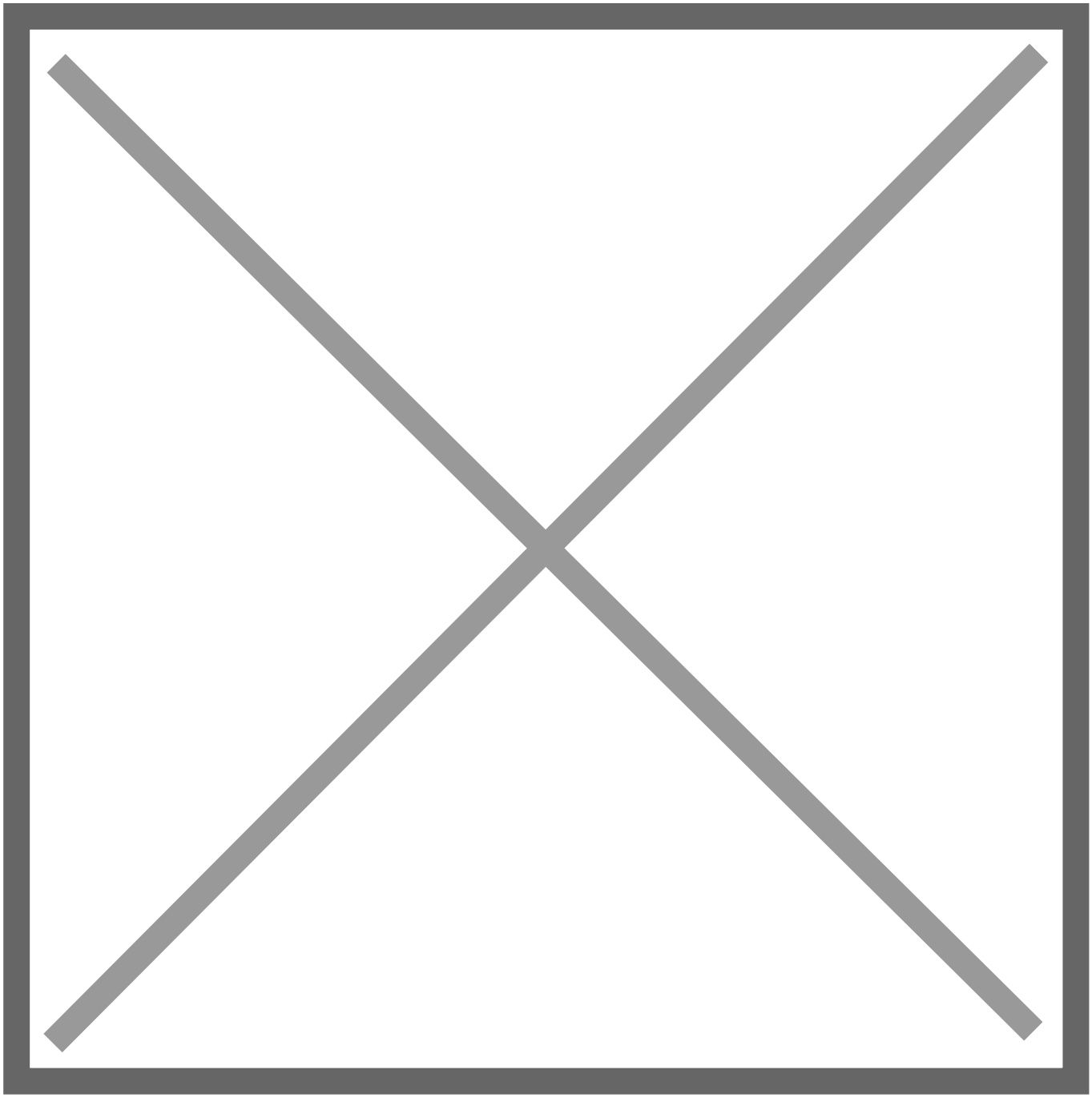
- **Function:** These signals are processed by the encoder block to calculate the rotation and direction of the motor shaft.
- **Connection:** Directly connected to the encoder block.

c) Setpoint Increment Button (key)

- **Description:** Physical button used to increase the desired motor position (setpoint), one increment per click.
- **Connection:** Connected to the debouncer block, which cleans the signal before sending it to the pulse_count16, responsible for incrementing the setpoint count.

d) Reset Button (rst)

- **Description:** Button used to reset or reinitialize a value in the system, such as counters or position reference.
- **Connection:** Passes through a debouncer and then connects to the second pulse_count16, which can reset the system or perform another control function.



2.2 Input Signal Processing (Debouncers)

Ensuring signal integrity is fundamental for system stability. In this step, we eliminate noise and interference from the physical buttons.

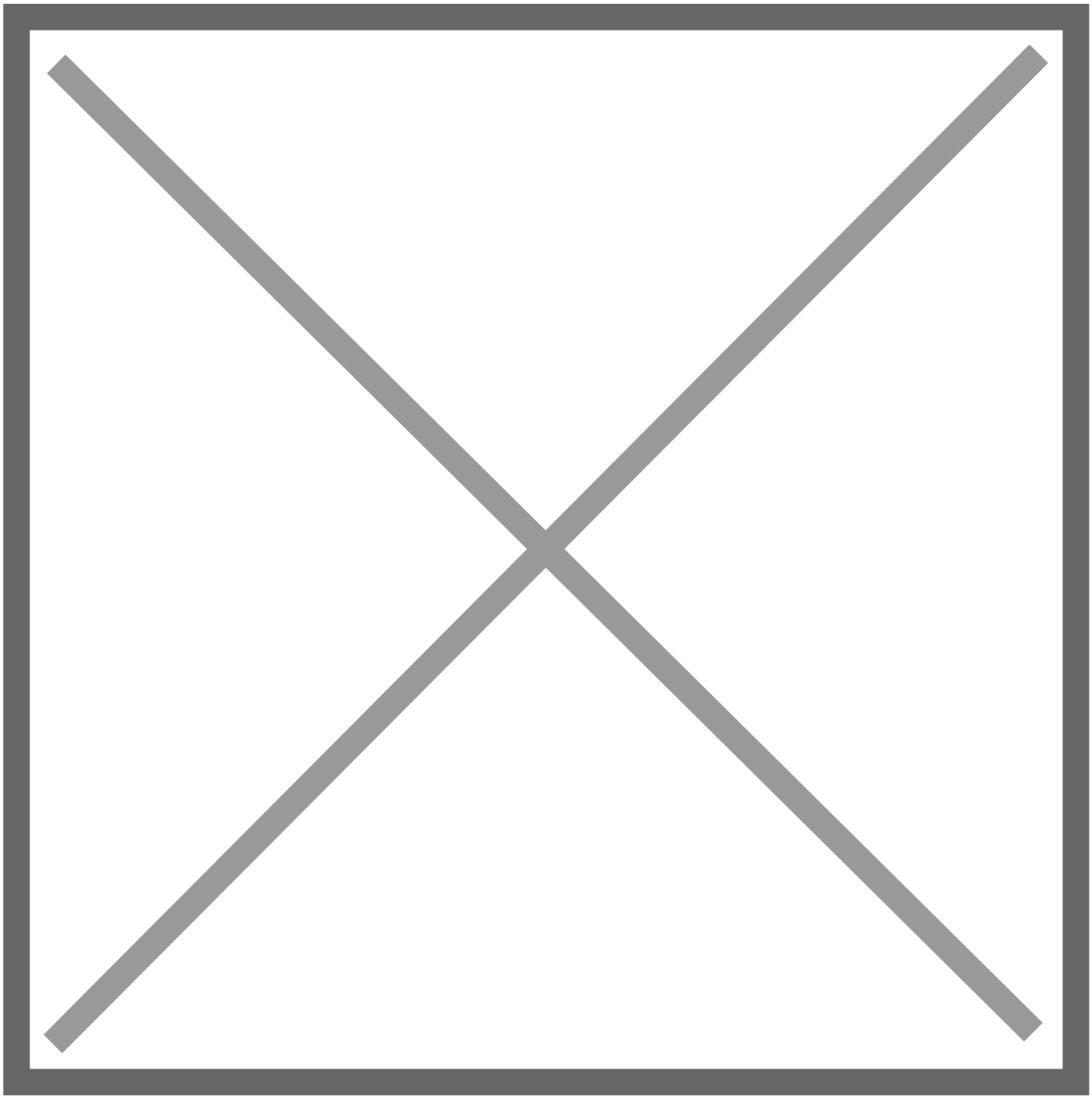
a) Debouncer for the key Button

- **Function:** Eliminates bouncing effects (undesired oscillations) on the key button signal.
- **Inputs:** clk (synchronization signal) and key (increment button).

- **Output:** Clean signal, free of noise, sent to the pulse_count16 responsible for the setpoint.

b) Debouncer for the rst Button

- **Function:** Cleans the reset button signal, removing noise that could trigger multiple unintended resets.
- **Inputs:** clk and rst.
- **Output:** Stable signal connected to the second pulse_count16.



2.3 Defining Setpoint and Reset

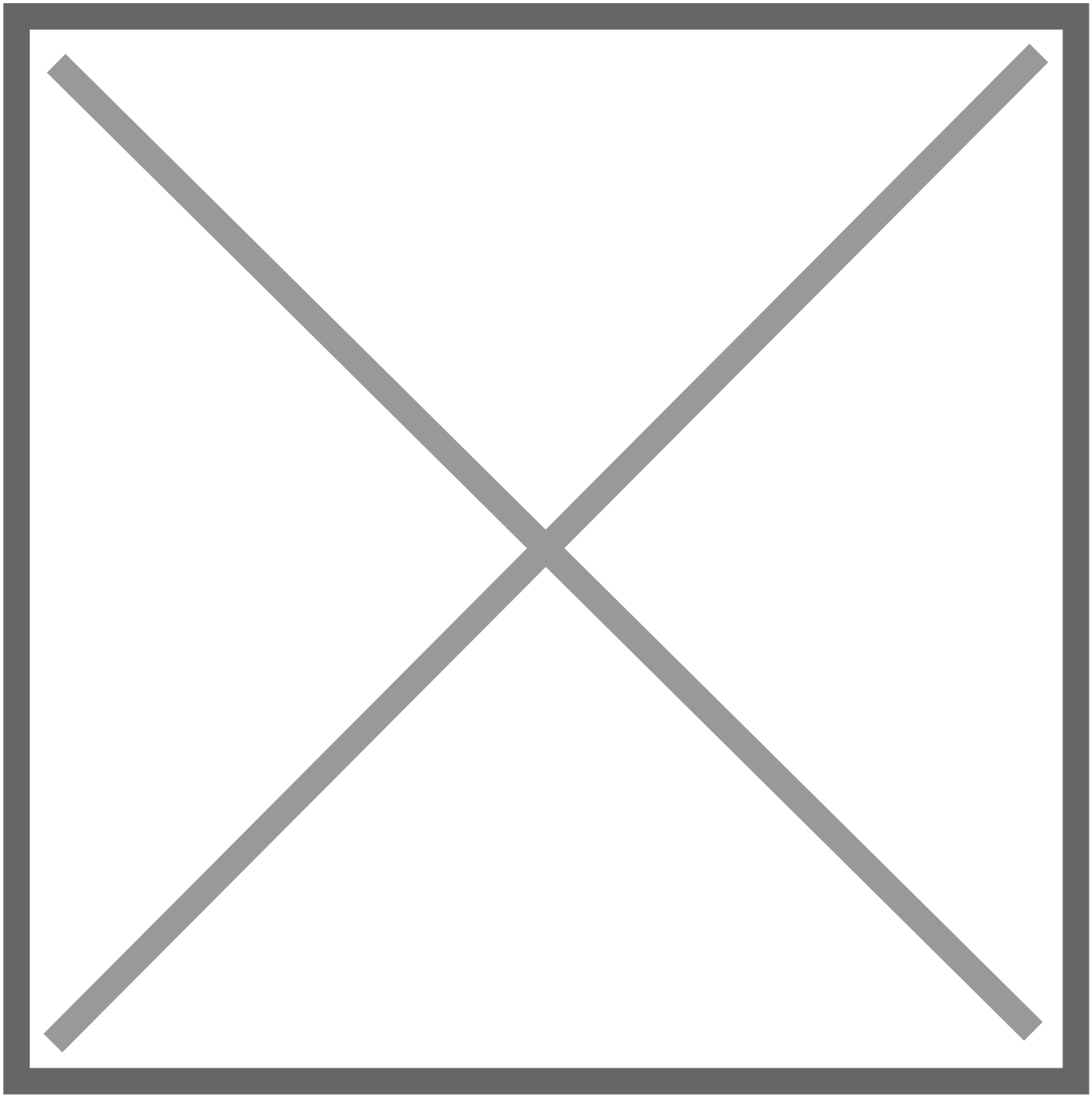
In this step, reference values are defined and managed to guide the motor’s position control.

a) Setpoint Counter (pulse_count16)

- **Function:** Each pulse received (from the key button) increments the setpoint value, representing the new desired motor position.
- **Input:** Directly from the key debouncer.
- **Output:** Reference value (rp[15:0]) used in the error calculation.

b) Reset Counter (pulse_count16)

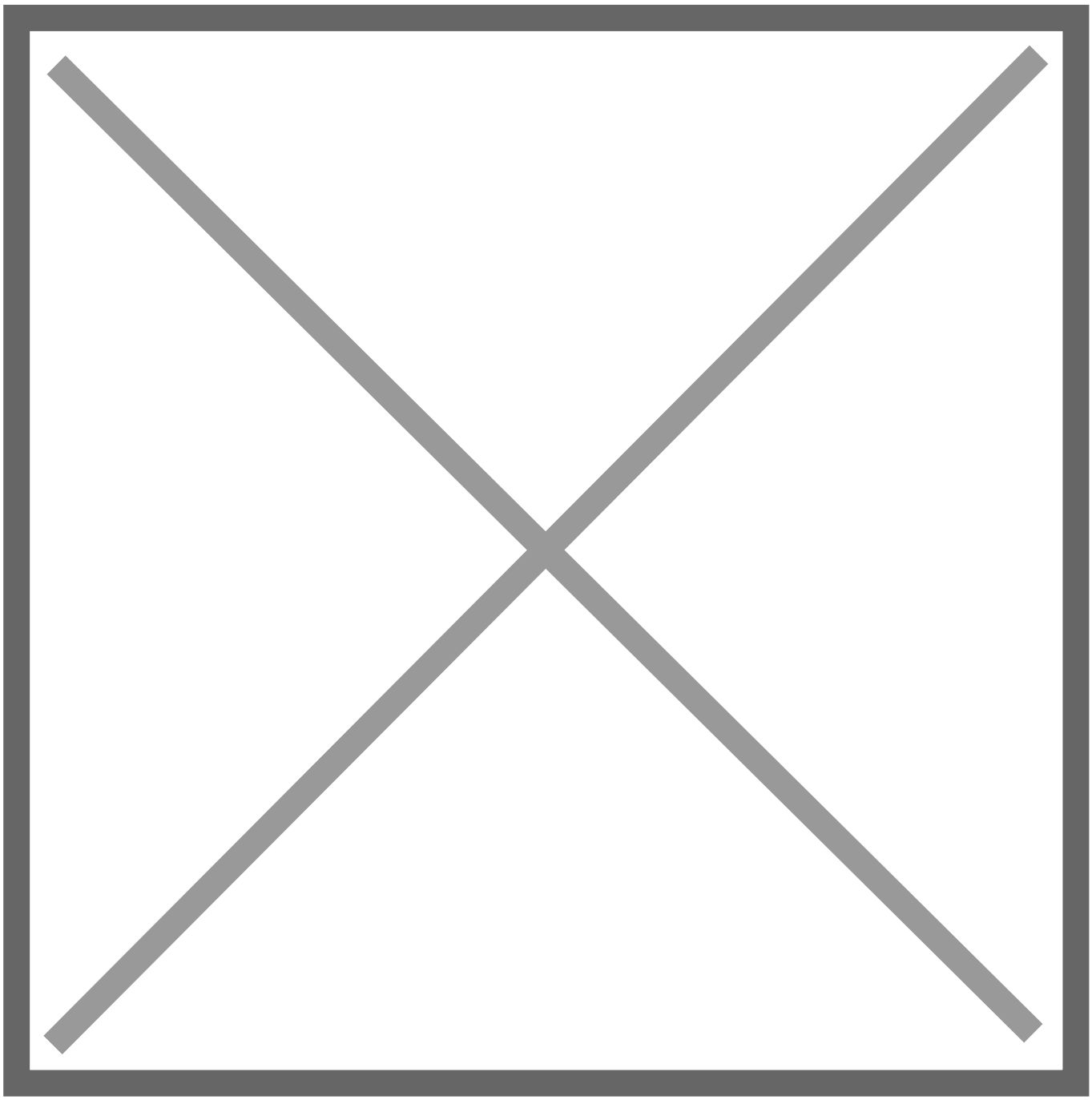
- **Function:** Controls reset or adjustment of a second reference parameter, possibly used to reset position values or counters.
- **Input:** From the rst button debouncer.
- **Output:** Provides an auxiliary value for error calculation or internal resets.



2.4 Reading the Current Position (Encoder)

a) encoder Block

- **Function:** Reads the quadA and quadB signals from the incremental encoder and converts them into the motor’s current position.
- **Inputs:** clk, IO57 (quadA), and IO56 (quadB).
- **Output:** counter[15:0] representing the accumulated pulse count, i.e., the motor’s current position relative to the starting point.

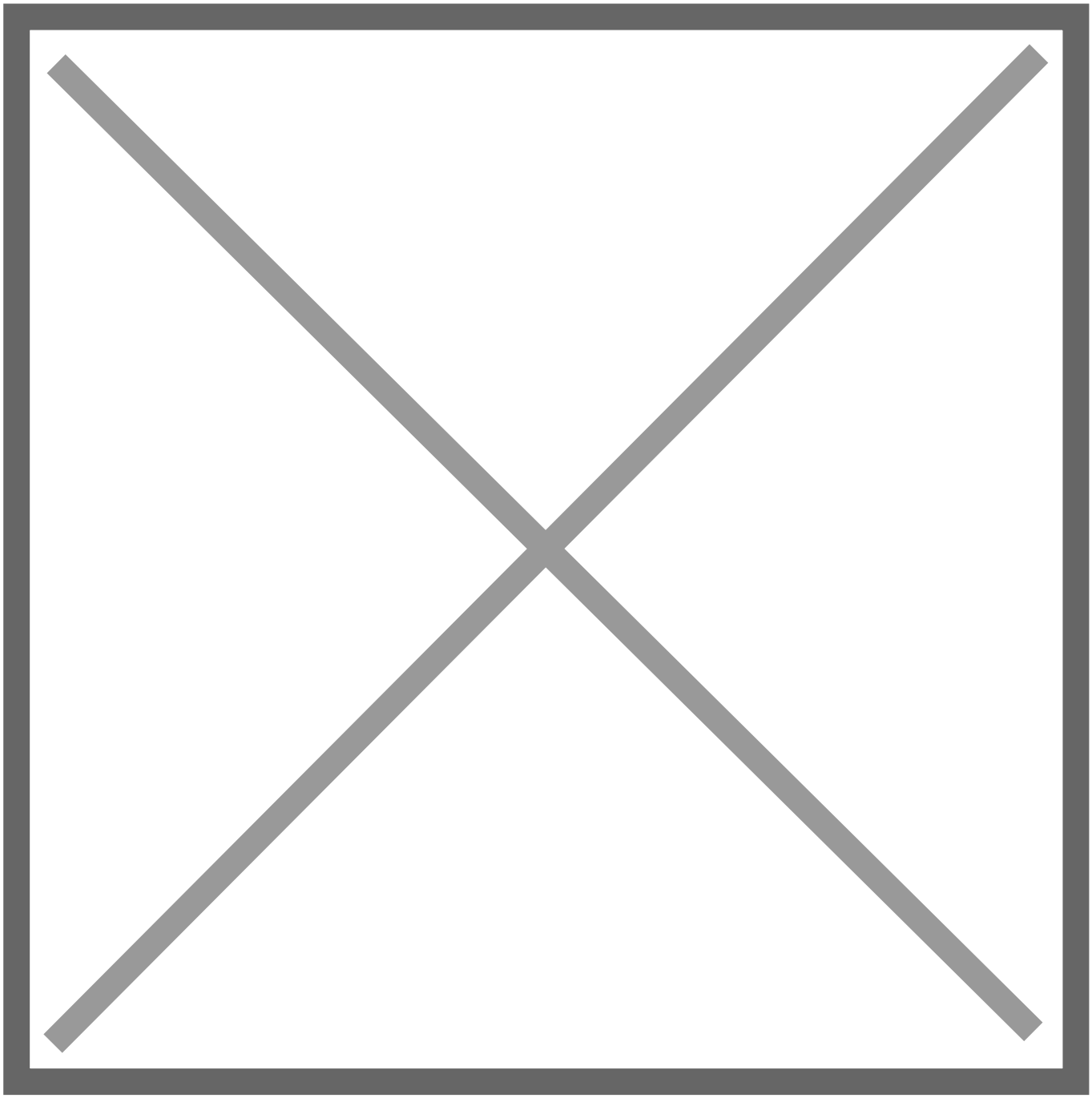


2.5 Position Error Calculation

a) Subtractor (subtractor16signed)

- **Function:** Determines the error between the desired position (setpoint) and the motor’s current position.
- **Inputs:**
  - rp[15:0]: Reference value from the first pulse_count16.
  - rp[15:0]: Reference value from the second pulse_count16.

- **Output:** n[15:0], representing the position error. This value indicates the distance remaining for the motor to reach the target position.



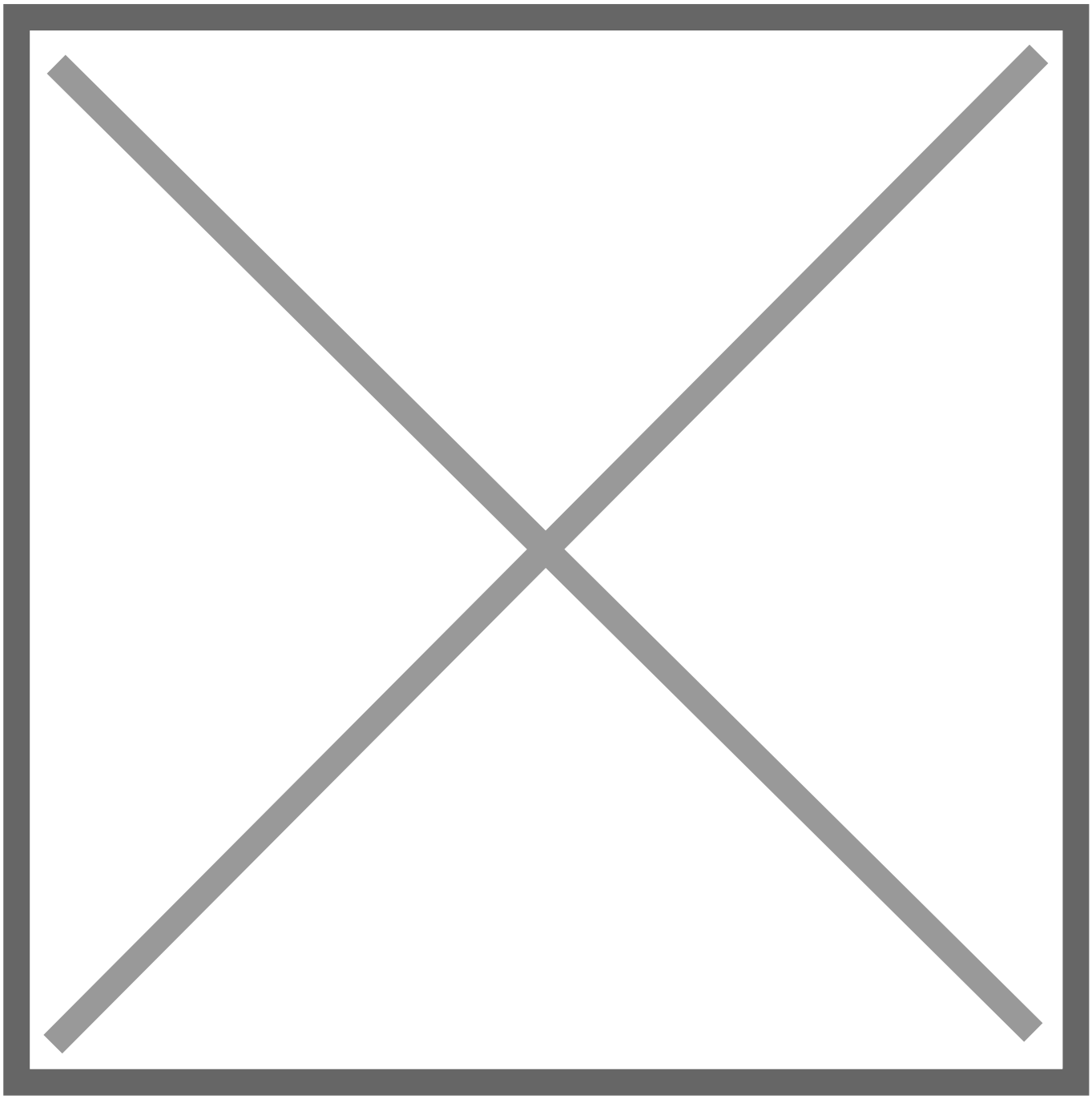
2.6 Proportional Gain Application

a) Proportional Gain (gain16signed)

- **Function:** Amplifies the position error, adjusting the control action intensity accordingly.
- **Inputs:**

- n[15:0]: Position error from the subtractor.
- nb[15:0]: Gain value (e.g., 100).

- **Output:** n[15:0], representing the control signal proportional to the error magnitude.

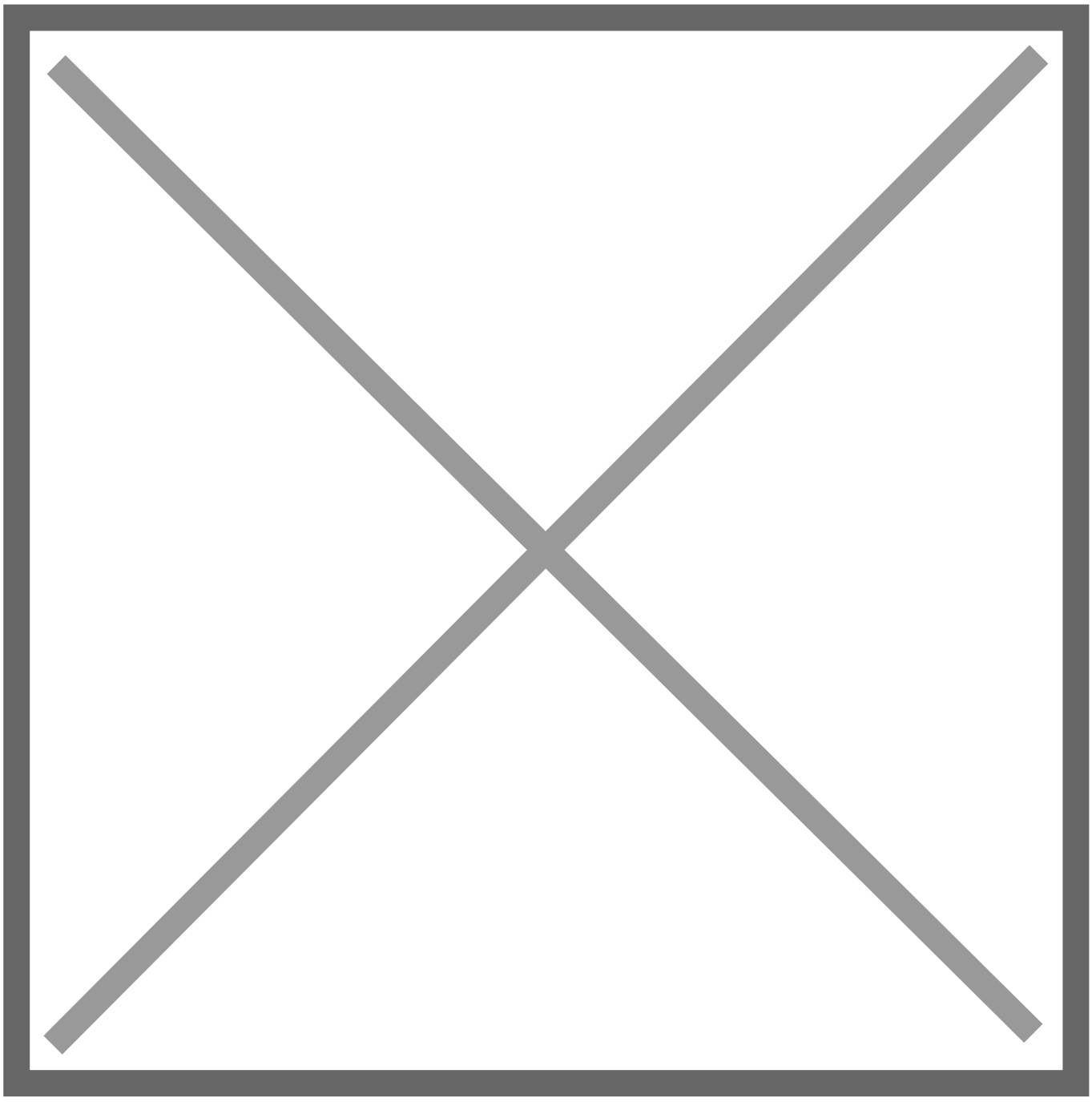


2.7 Summing with Additional Corrections (Optional)

a) Adder (adder16signed)

- **Function:** Adds the proportional control value to a fixed correction or to an integral control result (if implemented).

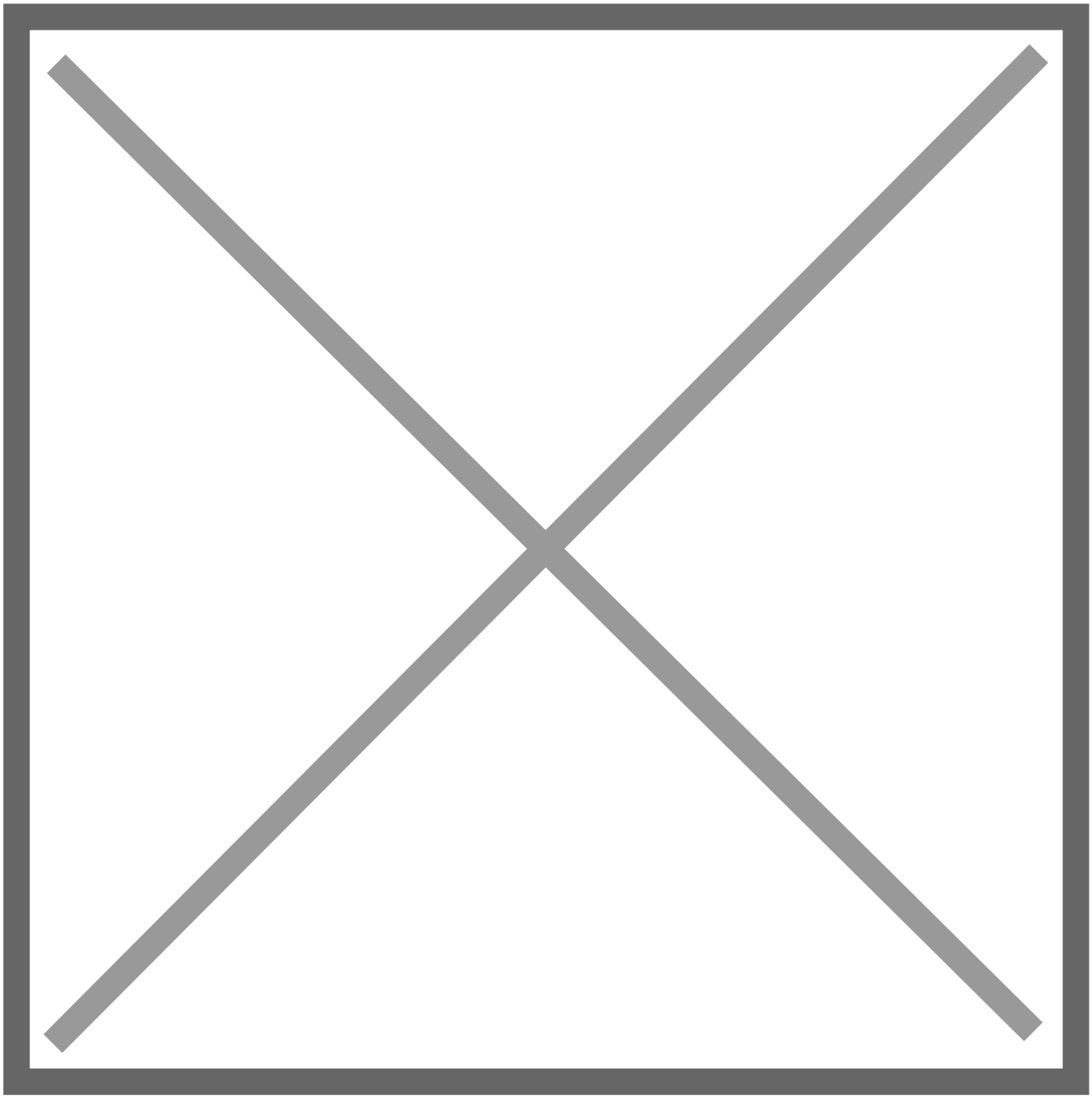
- **Inputs:**
  - Proportional control value.
  - A constant (e.g., 0 in this project).
- **Output:** Refined control signal for later use.



2.8 Determining the Rotation Direction

a) Comparator (greater16signed)

- **Function:** Compares the control signal with zero to determine the motor’s rotation direction.
- **Output:** out → 1 (rotates clockwise) or 0 (rotates counterclockwise).



2.9 PWM Duty Cycle Calculation

a) Absolute Module (number_module16)

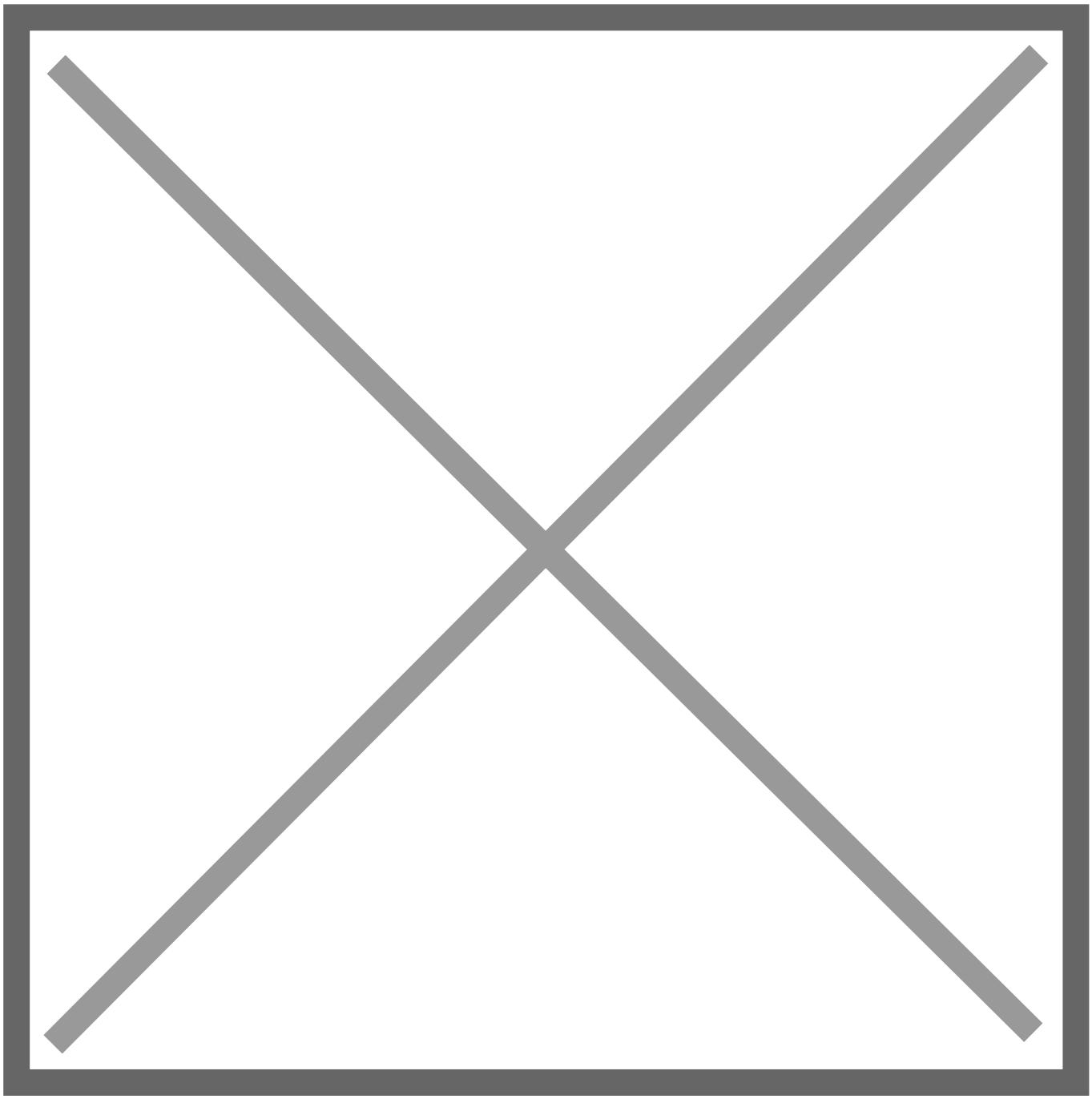
- **Function:** Converts the control signal to its absolute value, removing the sign information since the PWM is always positive.

b) Additional Gain (gain16signed)

- **Function:** Adjusts the absolute value with a specific gain to determine the duty cycle’s magnitude.
- **Constant:** Example: 10.

c) Saturation (saturation16)

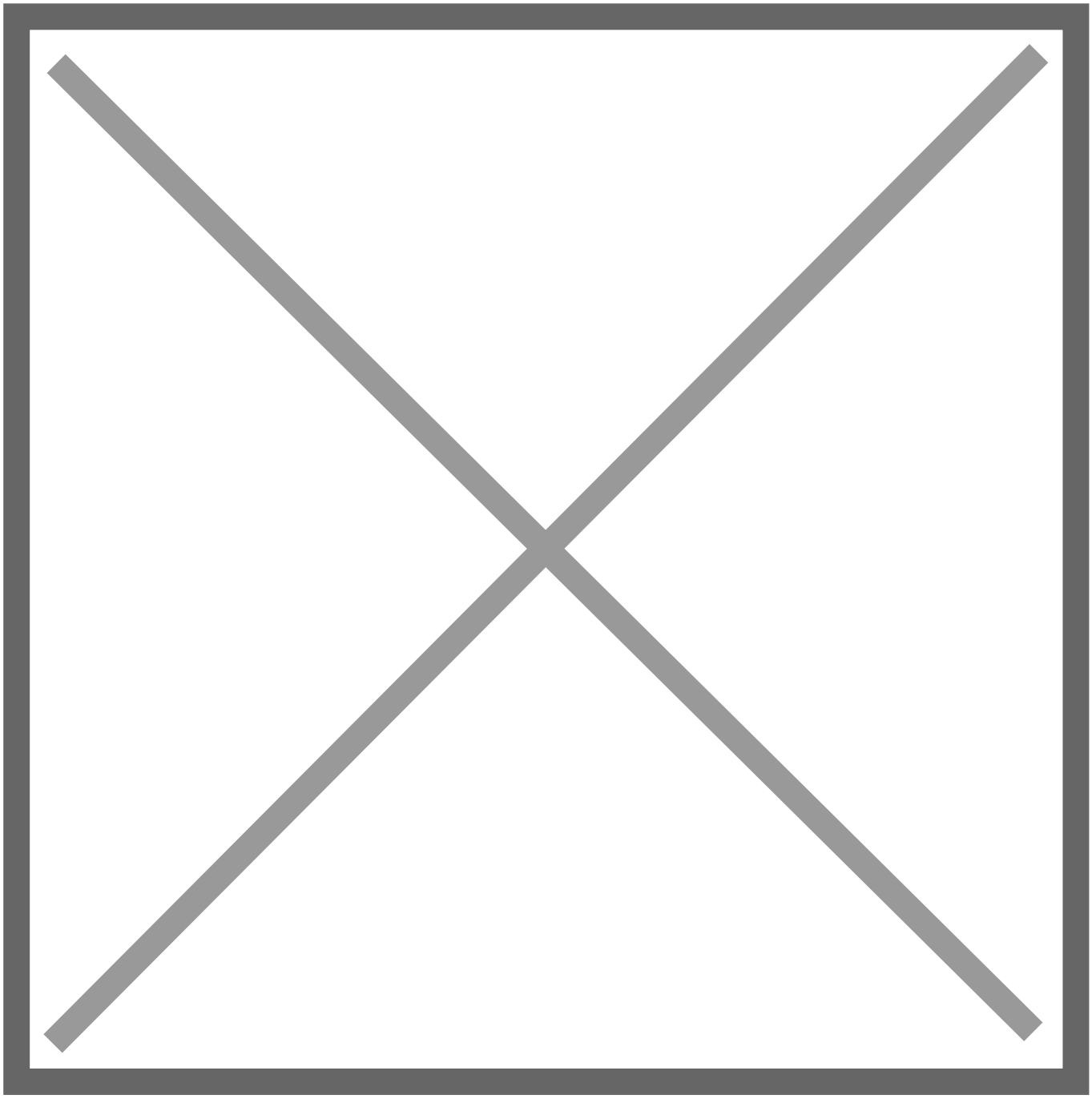
- **Function:** Limits the duty cycle value to a maximum of 255 (8 bits), ensuring the pulse width does not exceed the permitted range.
- **Output:** n[7:0], which feeds the PWM generator.



2.10 PWM Signal Generation

a) PWM Control (pwm_control8)

- **Function:** Generates the PWM signal responsible for motor speed control, based on the previously calculated duty cycle.
- **Inputs:** clk and duty_cycle[7:0].
- **Output:** pwm, a signal proportional to the desired intensity.

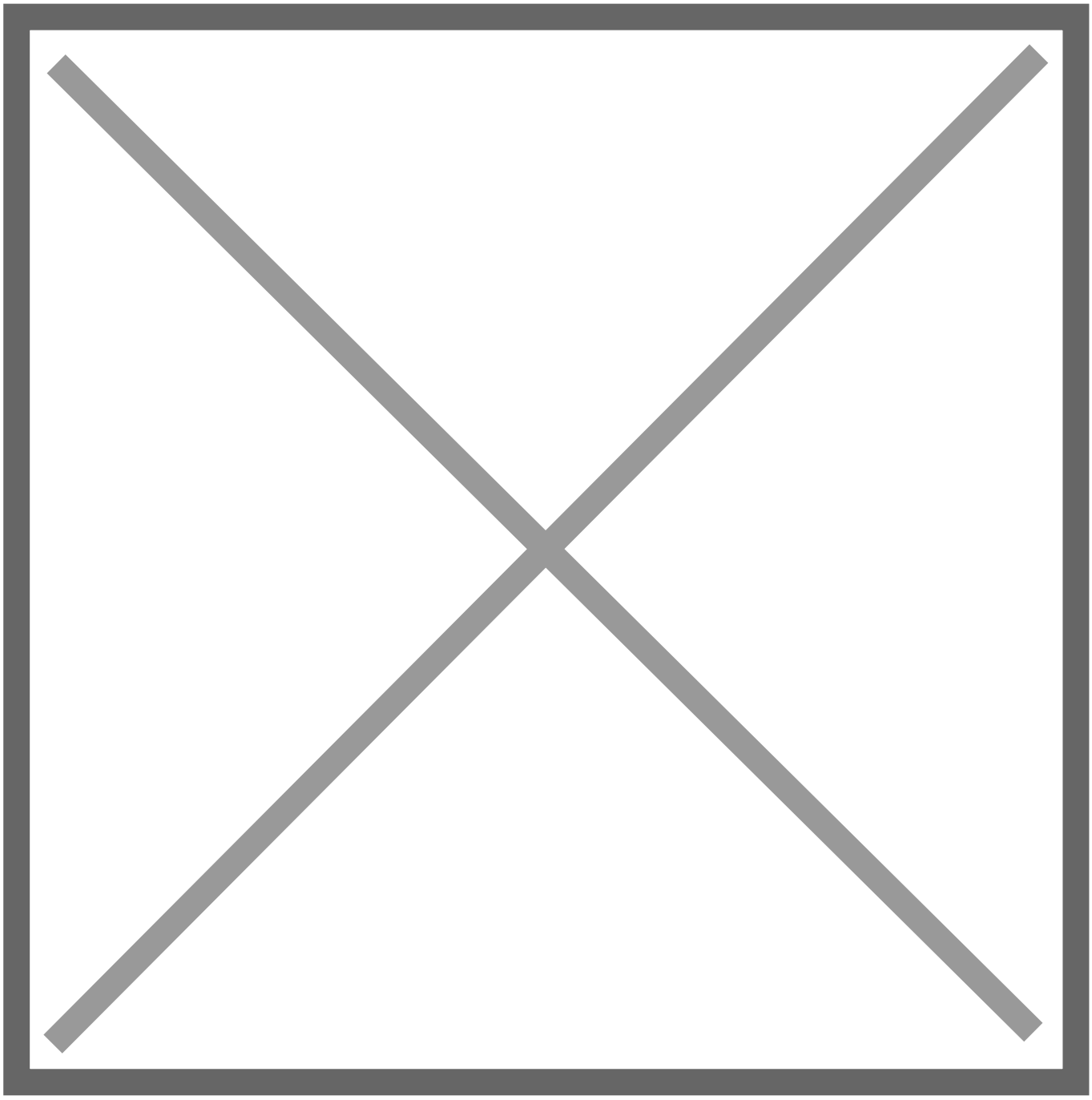


2.11 Direction Control and PWM Application

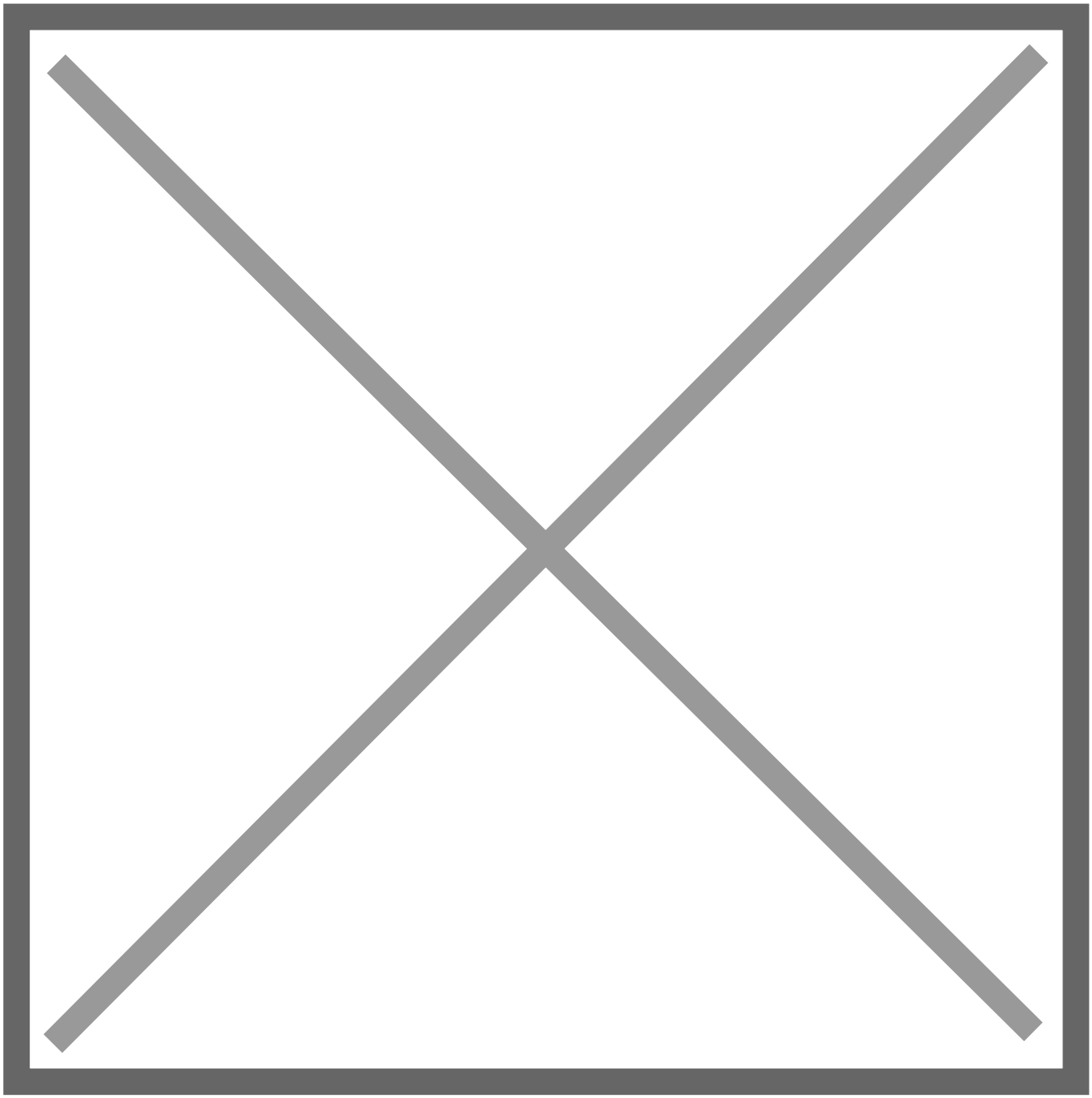
a) Demux (demux2)

- **Function:** Routes the PWM signal to one of two outputs, depending on the calculated rotation direction.
- **Inputs:**
  - in: PWM signal from pwm_control.
  - select: Direction signal from greater16signed.

- **Outputs:**
  - outa: IO69 → PWM signal for clockwise rotation.
  - outb: IO68 → PWM signal for counterclockwise rotation.



And, at the end, your project should be something like this:



Simulation Stage

Before programming the FPGA, it's essential to verify that your project behaves as expected in the simulation environment. This step helps catch potential logic or connection errors, saving time and avoiding issues in the physical implementation.

3.1 Accessing the Simulation Environment

1. Go to the Simulate tab in the top menu of ChipInventor.
2. Choose the simulation type based on your analysis needs:

- Advanced Simulation (with VCD generation and detailed signal monitoring).
- Dynamic Simulation (for real-time testing of inputs/outputs in a simplified interface).

3.2 Running the Simulation

- If using Advanced Simulation, click Run Iverilog to compile and simulate the design.
- Confirm that the simulation compiles without errors.
- If there are any compilation issues, review your connections and block configurations before proceeding.

3.3 Signals to Monitor

Focus on analyzing the key signals that represent the critical functionalities of your system. Here are the main ones you should observe during simulation (especially when generating a VCD file for waveform analysis):

Signal	Description
encoder counter (w_7)	Should increment or decrement based on changes in quadA and quadB.
setpoint (w_3)	Should increment with each press of the key button.
error (w_5)	Difference between setpoint and encoder position; check if it makes sense.
gain outputs (w_6, w_13)	Scaled values of the error and control signals; verify proportionality.
control result (w_8)	Combined control output used to define motor behavior.
duty cycle (w_14)	8-bit signal defining PWM intensity (0-255 range).
PWM signal (w_11)	The actual pulse-width modulated signal controlling motor speed.

Signal	Description
motor direction (w_10)	Binary signal indicating rotation direction (forward/reverse).
Outputs (IO69, IO68)	PWM signals routed according to direction (one active at a time).

3.4 Interpreting the Results

- **Confirm that:**
  - The encoder counter (w_7) updates properly when you simulate encoder signals (quadA and quadB).
  - The setpoint counter (w_3) increases with each key press.
  - The error signal (w_5) responds logically as the setpoint and encoder values change.
  - The duty cycle (w_14) remains within 0 to 255 and adjusts as the error varies.
  - The direction control (w_10) changes according to whether the motor needs to move forward or backward.
  - Only one of the outputs (IO69 or IO68) is active at any time, consistent with the rotation direction.

3.5 Troubleshooting

- **If you encounter compilation errors:**
  - Check all block connections.
  - Ensure no data type mismatches (e.g., 16-bit vs. 8-bit signals).

- **If simulation behavior is unexpected:**
  - Review the control logic (error calculation, gain factors).
  - Validate the encoder signals (quadA and quadB) timing and transitions.
  - Confirm that debouncer outputs behave correctly on button presses.

Run the simulation as many times as necessary until the system performs as expected.

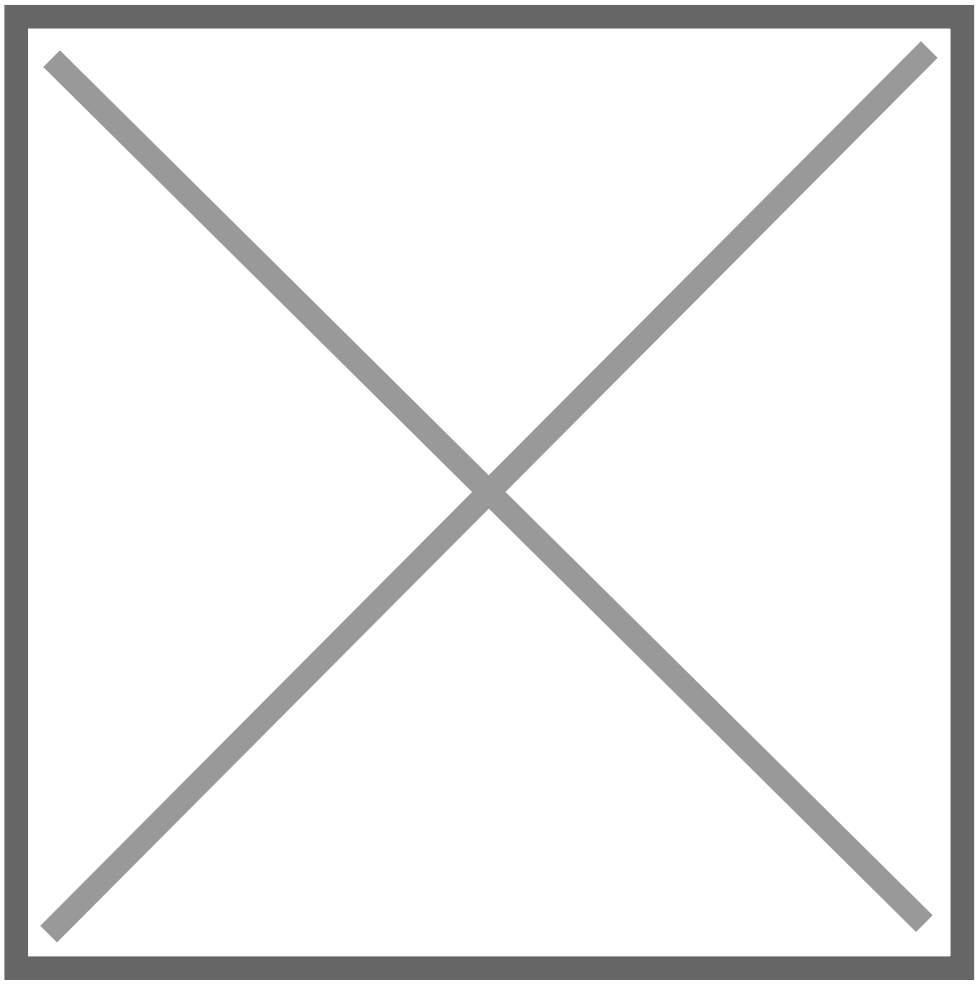
Synthesis and FPGA Programming

After confirming that everything works in simulation:

1. Click on the **Synthesize tab**.

2. Select **Start Synthesis**.

3. Wait until all items **turn green** (successful).



4. **Connect your FPGA board** to the computer via USB.

5. In ChipInventor, choose the correct **serial port** (usually the “Enhanced” port).

6. Click **Flashing** to **program your FPGA** with the generated bitstream.

Hardware Validation

After programming, perform practical validation:

1. Connect the **DC motor and encoder** physically to the FPGA board (using appropriate drivers for motor current/voltage).

2. **Power the system** with the proper power supply.

3. Press the **configured button** in the project (if it is used to increment the setpoint) and observe:

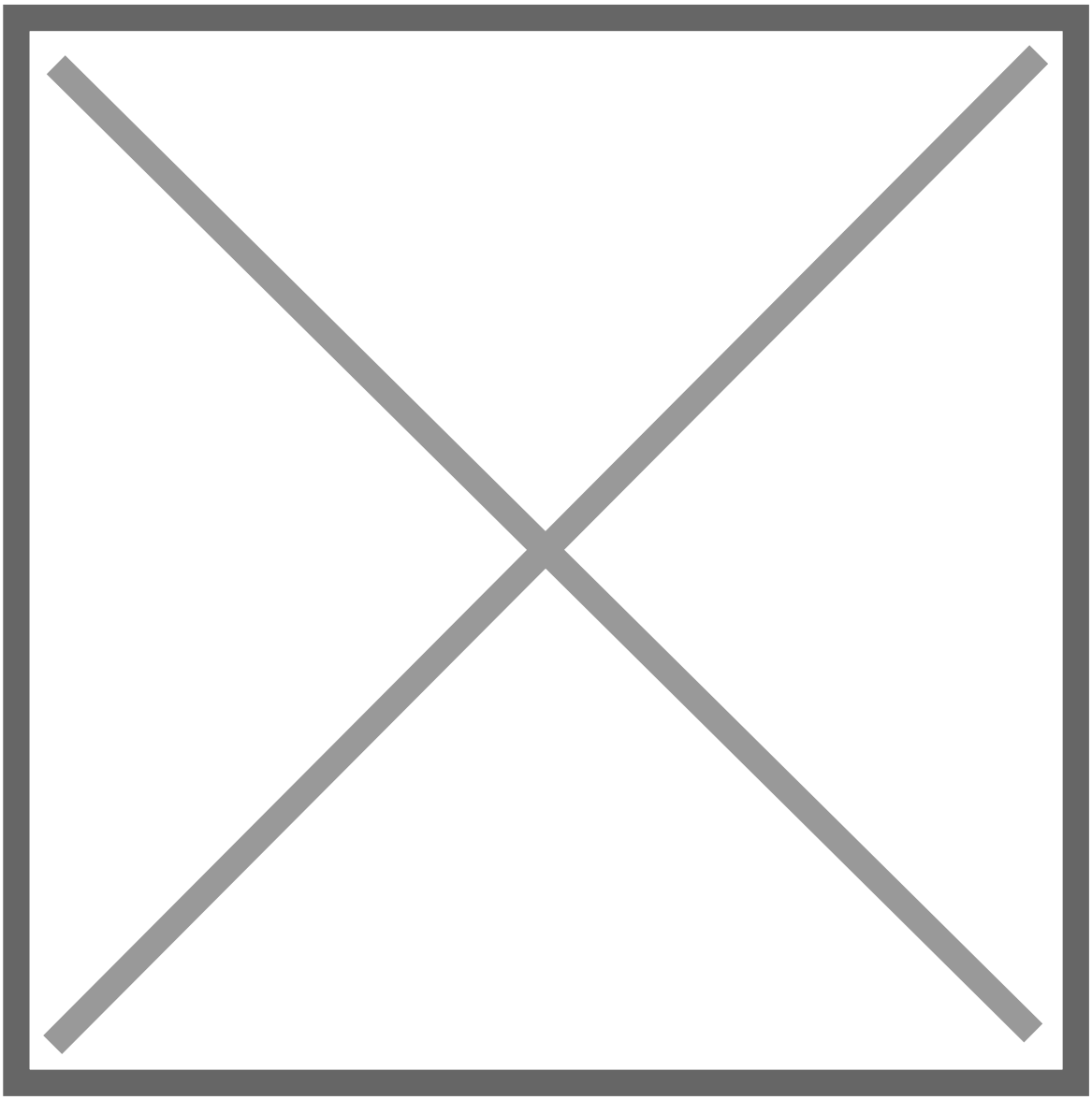
   - Whether the motor rotates in the correct direction as you increase the setpoint.
   - Whether, upon changing the reference or pressing rst, the motor responds correctly (adjusting speed or returning to zero position, depending on the implementation).
   - The PWM should vary to correct the difference between desired and actual position.

> **If you see undesired behavior:**
> - Check if the encoder direction is inverted;
> - Adjust the controller gains (the constants used in gain16signed);
> - Confirm that the PWM and motor speed are not saturating (very high values might keep the motor at high speed continuously).

And at the end, your circuit should look like this:

image.png

DC Motor Circuit



DC Motor Testing

Wrapping Up

Congratulations! You have developed a **DC motor speed and position control system** in ChipInventor. This project covered creating, simulating, synthesizing, and testing in hardware. You used pulse counters, an encoder, arithmetic operations, PWM generation, and direction selection via a demultiplexer.

From here, you can:

- Improve your controller (for example, by adding a derivative term or filters).
- Integrate communication interfaces (UART, SPI, etc.) to monitor position and speed in real time.
- Explore other sensing methods for more sophisticated motor control.

Keep exploring ChipInventor, and take on progressively more complex digital design and control projects!