OBJECTIVE

The primary objective of this project is to design and implement a real-time digital speedometer system using FPGA technology. The system aims to:

- **Accurately measure vehicle speed** by interfacing with a Hall effect sensor to detect wheel rotations
- **Provide real-time speed display** on three 7-segment displays showing speed in km/h with a range of 0-255 km/h
- **Implement safety features** through an over-speed warning system using a red LED indicator
- **Demonstrate practical VLSI design principles** including modular architecture, synchronous digital design, and real-world sensor interfacing
- **Create a cost-effective and reliable speed measurement solution** suitable for automotive applications
- **Validate the design** through comprehensive simulation and testing methodologies

SOFTWARE AND HARDWARE USED

Software Tools:

- **Xilinx Vivado Design Suite 2020.1**
    - HDL synthesis and implementation
    - Timing analysis and optimization
    - Simulation and verification
- **ModelSim**
    - RTL simulation and waveform analysis
    - Testbench development and verification
- **Programming Languages:**
    - Verilog HDL for RTL design
    - Behavioural and structural modelling techniques
    - Testbench development and verification

Hardware Components:

- **FPGA Development Board:**
    - Digilent Basys 3 Artix-7 FPGA Board
    - 100MHz on-board system clock (divided to 2Hz for project)
    - Three integrated 7-segment displays (AN[2:0], CA-CG segments)
    - 16 user switches (SW[15:0]) and 16 user LEDs (LED[15:0])

- **External Components:**
    - Hall Effect Sensor (A3144 or equivalent)
    - Red LED for over-speed indication
    - Connecting wires and breadboard
    - Power supply (3.3V/9V)

OVERVIEW OF THE SYSTEM

The FPGA-based Digital Speedometer is a comprehensive embedded system that transforms mechanical wheel rotation into accurate digital speed measurements. The system operates on the fundamental principle of measuring rotational frequency and converting it to linear velocity.

The system continuously monitors pulses generated by a Hall effect sensor mounted near a rotating wheel with embedded magnets. Each pulse represents one complete rotation, and by counting these pulses over a fixed time interval (0.5 seconds), the system calculates the rotational frequency and subsequently the vehicle speed.

Key Features:

- **Real-time Processing:** Instantaneous speed calculation and display updates
- **High Accuracy:** Precise pulse counting with edge detection algorithms
- **User-Friendly Display:** Three-digit 7-segment display showing speed in km/h
- **Safety Integration:** Configurable over-speed warning system
- **Robust Design:** Noise-immune digital processing with proper synchronization

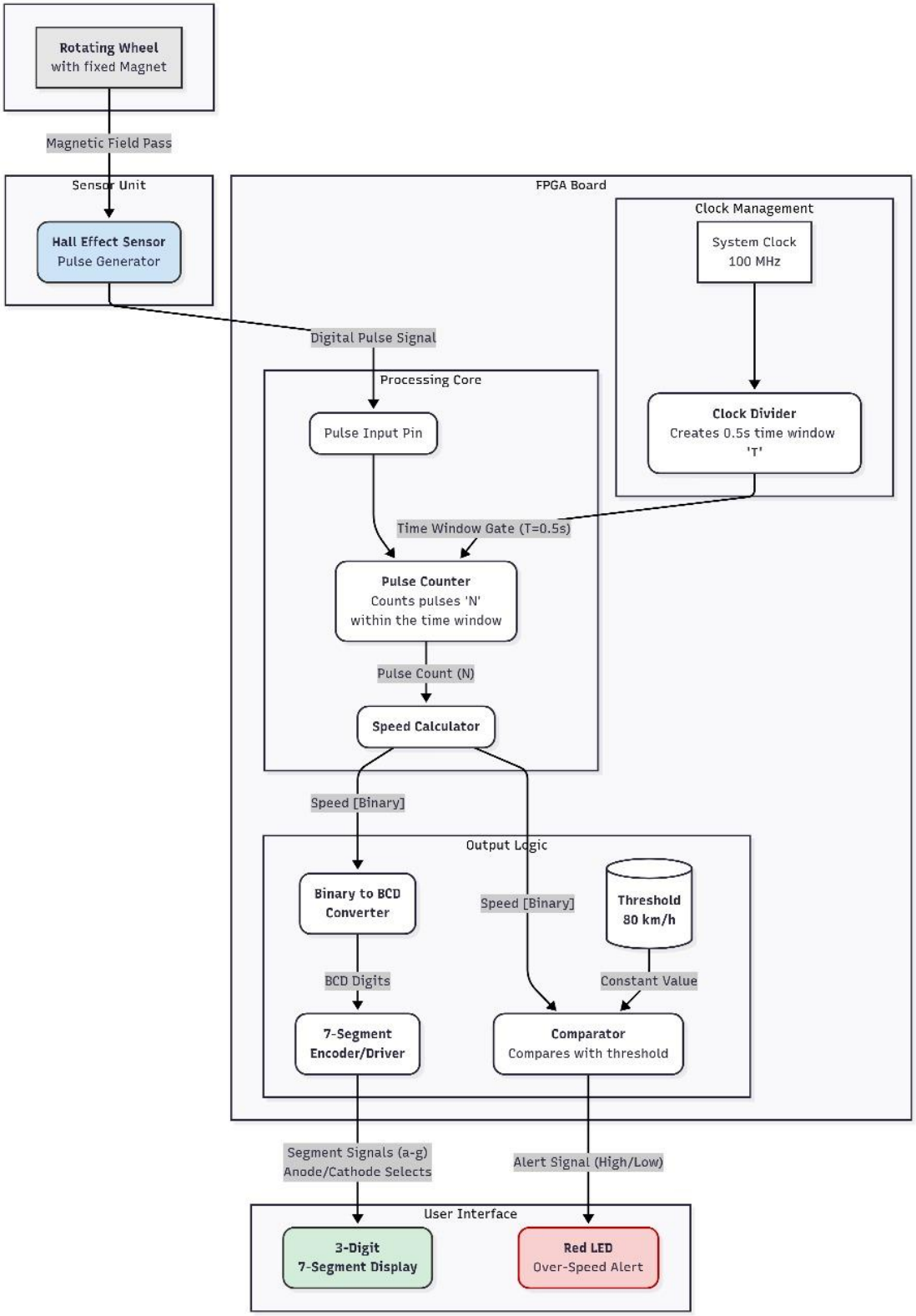
System Workflow:

1. **Signal Acquisition:** Hall sensor detects magnetic field changes as magnets pass
2. **Pulse Conditioning:** Digital processing eliminates noise and detects valid edges
3. **Time-based Counting:** Accumulates pulses over precisely timed intervals
4. **Speed Calculation:** Applies calibrated mathematical conversion
5. **Display Processing:** Converts binary speed to BCD format for 7-segment displays
6. **Safety Monitoring:** Continuously compares speed against predefined thresholds

Technical Specifications:

- **Measurement Range:** 0 to 255 km/h
- **Update Rate:** 2 Hz (every 0.5 seconds)
- **Display Resolution:** 1 km/h
- **Response Time:** < 0.5 seconds
- **Power Consumption:** < 2W (FPGA dependent)

BLOCK DIAGRAM



INTRODUCTION

In the rapidly evolving automotive industry, the integration of digital technologies has revolutionized traditional mechanical systems, leading to enhanced accuracy, reliability, and functionality. The speedometer, a critical instrument in vehicle safety and navigation, has undergone significant transformation from mechanical cable-driven systems to sophisticated electronic solutions.

Traditional mechanical speedometers rely on flexible cables connected to the vehicle's transmission, converting rotational motion through gear mechanisms to display speed on analog dials. While functional, these systems suffer from several limitations:

- **Mechanical wear and tear** leading to accuracy degradation over time
- **Limited integration capabilities** with modern digital dashboard systems
- **Susceptibility to vibrations** and environmental factors
- **Difficulty in implementing advanced features** such as digital displays and warning systems

The advent of Field-Programmable Gate Array (FPGA) technology presents an unprecedented opportunity to address these challenges while introducing advanced features previously unattainable with mechanical systems.

FPGAs offer unique advantages for automotive speedometer applications:

- **Real-time Processing:** Hardware-level parallel processing ensures instantaneous response to sensor inputs
- **Reconfigurability:** Software-defined functionality allows for easy updates and feature additions
- **Reliability:** No mechanical components reduce failure points and maintenance requirements
- **Integration:** Single-chip solution combining sensor interface, processing, and display control
- **Cost-effectiveness:** Eliminates complex mechanical assemblies while providing enhanced functionality

This project demonstrates the practical implementation of digital signal processing principles in real-world automotive applications. By leveraging Hall effect sensors—solid-state devices that detect magnetic field variations—the system achieves:

- **Non-contact measurement:** Eliminates mechanical wear and improves reliability
- **High precision:** Digital processing enables accurate pulse counting and speed calculation
- **Environmental robustness:** Solid-state components withstand vibration, temperature variations, and electromagnetic interference

COMPLETE CODE WITH COMMENTS

```
`timescale 1ns / 1ps

//=============================================================================
// Module: 7-Segment Display Multiplexer
// Implements time-division multiplexing to drive 3 digits on a 4-digit display.
//=============================================================================
module seven_seg_mux (
    input  wire       clk,
    input  wire       reset_n,
    input  wire [6:0] hex2_in, // Hundreds digit
    input  wire [6:0] hex1_in, // Tens digit
    input  wire [6:0] hex0_in, // Ones digit
    output reg  [6:0] seg,     // 7-segment cathodes (active-low)
    output reg  [3:0] an       // Anode enables (active-high, an[0] is rightmost)
);
    // Counter to generate the refresh rate (controls which digit is active)
    reg [17:0] refresh_counter;
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            refresh_counter <= 0;
        end else begin
            refresh_counter <= refresh_counter + 1;
        end
    end
    // Sequential logic to switch between the three digits (0, 1, 2)
    always @(*) begin
        // Using the top two bits of the counter for a 4-state sequence
        case (refresh_counter[17:16])
            2'b00: begin an = 4'b0001; seg = hex0_in; end // Display Ones digit (an[0])
            2'b01: begin an = 4'b0010; seg = hex1_in; end // Display Tens digit (an[1])
            2'b10: begin an = 4'b0100; seg = hex2_in; end // Display Hundreds digit
(an[2])
            default: begin an = 4'b0000; seg = 7'b1111111; end // Off state (all segments
off)
        endcase
    end
endmodule


//=============================================================================
// Top Module: speedometer_top
```

```
// Instantiates all sub-modules for the full speedometer function.
//=============================================================
==================
module speedometer_top (
    input  wire       clk,
    input  wire       reset_n,
    input  wire       hall_effect_input, // Raw, asynchronous input
    output wire [6:0] seg,
    output wire [3:0] an,
    output wire       red_led            // Over-speed indicator
);
    wire timer_done;
    wire timer_reset;
    wire [8:0] pulse_count;
    wire [7:0] speed_kmh;
    wire [3:0] hundreds_digit;
    wire [3:0] tens_digit;
    wire [3:0] ones_digit;
    wire [6:0] seven_seg_hex0_wire;
    wire [6:0] seven_seg_hex1_wire;
    wire [6:0] seven_seg_hex2_wire;

    // The timer is reset when it is done, creating a periodic measurement interval.
    assign timer_reset = timer_done;

    // Creates a measurement interval (200,000,000 clock cycles = 2 seconds for
100MHz clock)
    clock_divider #( .MAX_COUNT(200_000_000 - 1) ) clk_div_inst ( .clk(clk),
.reset_n(reset_n), .timer_done(timer_done) );

    // Counts the hall sensor pulses during the measurement interval.
    pulse_counter pulse_cnt_inst ( .clk(clk), .reset_n(reset_n),
.hall_effect_input(hall_effect_input), .timer_done(timer_done),
.timer_reset(timer_reset), .pulse_count_out(pulse_count) );

    // Converts pulse count to speed in km/h.
    speed_calculator speed_calc_inst ( .pulse_count_in(pulse_count),
.speed_kmh_out(speed_kmh) );

    // Decomposes the 8-bit speed into 3 BCD digits.
    bcd_to_digits bcd_conv_inst ( .speed_in(speed_kmh),
.hundreds_out(hundreds_digit), .tens_out(tens_digit), .ones_out(ones_digit) );

    // Decode BCD to 7-segment format for each digit.
    seven_segment_decoder decoder_hex0 ( .bcd_in(ones_digit),
.seg_out(seven_seg_hex0_wire) );
```

```
    seven_segment_decoder decoder_hex1 ( .bcd_in(tens_digit),
.seg_out(seven_seg_hex1_wire) );
    seven_segment_decoder decoder_hex2 ( .bcd_in(hundreds_digit),
.seg_out(seven_seg_hex2_wire) );

    // Multiplexes the 7-segment data onto the physical display pins.
    seven_seg_mux display_mux_inst ( .clk(clk), .reset_n(reset_n),
.hex2_in(seven_seg_hex2_wire), .hex1_in(seven_seg_hex1_wire),
.hex0_in(seven_seg_hex0_wire), .seg(seg), .an(an) );

    // Controls the red LED based on a speed limit (currently 0 km/h).
    red_led_controller led_ctrl_inst ( .speed_in(speed_kmh), .red_led_out(red_led) );
endmodule


//=============================================================
==================
// Sub-module: clock_divider
// Generates a single-cycle pulse (timer_done) after MAX_COUNT cycles.
//=============================================================
==================
module clock_divider #( parameter MAX_COUNT = 25_000_000 - 1 ) (
    input wire clk,
    input wire reset_n,
    output reg timer_done // Pulse high for one clock cycle when done counting
);
    // 26 bits for a 200M count from a 100MHz clock is sufficient.
    reg [25:0] count;
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            count <= 0; timer_done <= 0;
        end else begin
            if (count == MAX_COUNT) begin
                count <= 0; timer_done <= 1; // Assert done and reset count
            end else begin
                count <= count + 1; timer_done <= 0;
            end
        end
    end
endmodule

//=============================================================
==================
// NEW Sub-module: Input Synchronizer
// 2-Flip-Flop synchronizer to mitigate metastability for asynchronous input.
```

```
//=============================================================
==================
module synchronizer (
   input  wire clk,
   input  wire reset_n,
   input  wire async_in, // The raw, external signal (hall_effect_input)
   output wire sync_out  // The synchronized signal for internal logic
);
   // Two registers in series to allow an asynchronous signal to settle
   reg sync_reg1;
   reg sync_reg2;
   always @(posedge clk or negedge reset_n) begin
      if (!reset_n) begin
         sync_reg1 <= 1'b0;
         sync_reg2 <= 1'b0;
      end else begin
         sync_reg1 <= async_in; // First stage
         sync_reg2 <= sync_reg1; // Second stage (output)
      end
   end
   assign sync_out = sync_reg2;
endmodule


//=============================================================
==================
// UPDATED Sub-module: pulse_counter
// Counts rising edges of the synchronized hall sensor signal.
//=============================================================
==================
module pulse_counter (
   input wire clk,
   input wire reset_n,
   input wire hall_effect_input, // Raw input from the pin
   input wire timer_done,
   input wire timer_reset,
   output reg [8:0] pulse_count_out // Output count is 9 bits for up to 511 pulses
);
   wire hall_effect_sync; // The synchronized signal
   reg  hall_effect_prev;

   // Synchronizes the external sensor signal to the FPGA clock domain
   synchronizer hall_sensor_sync_inst (
      .clk       (clk),
      .reset_n   (reset_n),
      .async_in  (hall_effect_input), // Raw signal
```

```
        .sync_out  (hall_effect_sync)   // Clean signal
    );

    // Rising edge detection and counting logic
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            pulse_count_out <= 0;
            hall_effect_prev <= 0;
        end else begin
            if (timer_reset) begin
                pulse_count_out <= 0; // Reset count for the next measurement interval
            end
            else if (!timer_done) begin
                // Detect a rising edge on the synchronized signal
                if (hall_effect_sync == 1'b1 && hall_effect_prev == 1'b0) begin
                    pulse_count_out <= pulse_count_out + 1;
                end
            end
            // Store current state for next cycle's edge detection
            hall_effect_prev <= hall_effect_sync;
        end
    end
endmodule

//================================================================
==================
// Sub-module: speed_calculator
// Converts pulse count (per 2 seconds) to speed in km/h.
// Formula: Speed = (Pulse_Count * 23) / 160
// (This specific formula is based on wheel circumference, magnets, and interval
time).
//================================================================
==================
module speed_calculator (
    input wire [8:0] pulse_count_in,
    output reg [7:0] speed_kmh_out
);
    always @(*) begin
        speed_kmh_out = (pulse_count_in * 23) / 160;
    end
endmodule


//================================================================
==================
// Sub-module: bcd_to_digits
```

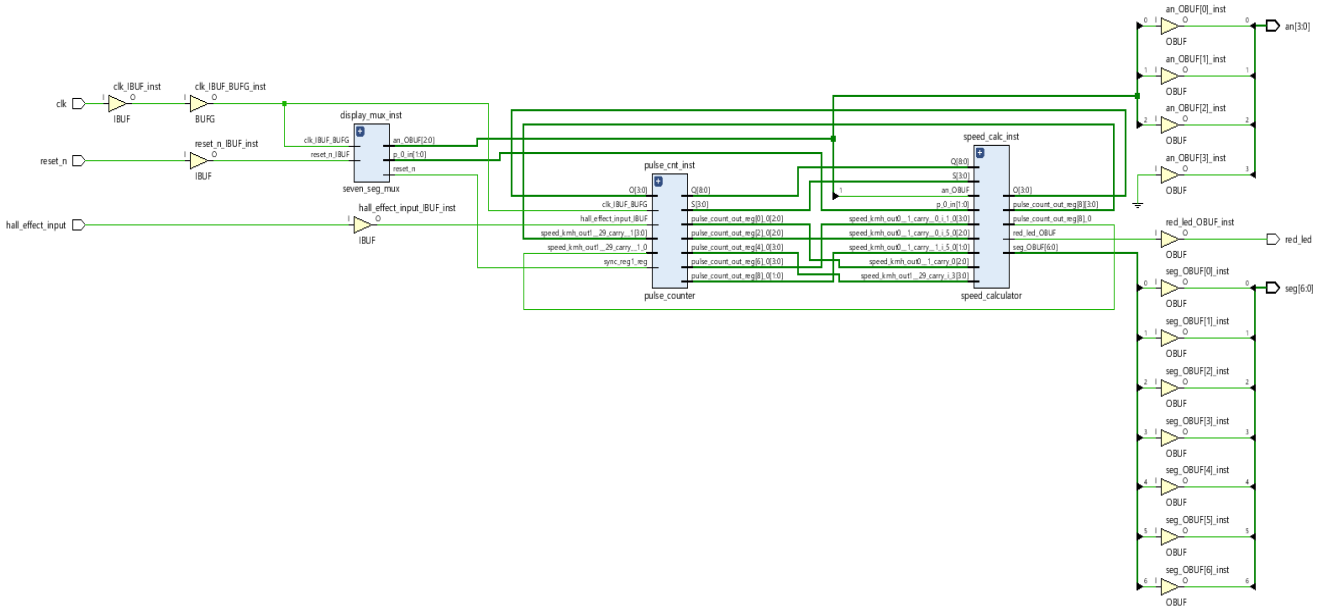
```
// Performs decimal decomposition of the speed value.
//=============================================================
==================
module bcd_to_digits (
   input wire [7:0] speed_in, // Max value is 255
   output reg [3:0] hundreds_out,
   output reg [3:0] tens_out,
   output reg [3:0] ones_out
);
   always @(*) begin
      // Integer arithmetic for digit extraction
      hundreds_out = speed_in / 100;
      tens_out     = (speed_in % 100) / 10;
      ones_out     = speed_in % 10;
   end
endmodule


//=============================================================
==================
// Sub-module: seven_segment_decoder
// Maps a 4-bit BCD number to the 7-segment segment pattern (active-low).
//=============================================================
==================
module seven_segment_decoder (
   input wire [3:0] bcd_in,
   output reg [6:0] seg_out // a, b, c, d, e, f, g (seg[6:0])
);
   always @(*) begin
      case (bcd_in)
         4'd0: seg_out = 7'b1000000; // 0
         4'd1: seg_out = 7'b1111001; // 1
         4'd2: seg_out = 7'b0100100; // 2
         4'd3: seg_out = 7'b0110000; // 3
         4'd4: seg_out = 7'b0011001; // 4
         4'd5: seg_out = 7'b0010010; // 5
         4'd6: seg_out = 7'b0000010; // 6
         4'd7: seg_out = 7'b1111000; // 7
         4'd8: seg_out = 7'b0000000; // 8
         4'd9: seg_out = 7'b0010000; // 9
         default: seg_out = 7'b1111111; // Off/Blank
      endcase
   end
endmodule
```

```
//=============================================================================
// Sub-module: red_led_controller
// Compares the calculated speed to a user-defined speed limit.
//=============================================================================
module red_led_controller (
    input wire [7:0] speed_in,
    output reg red_led_out
);
    // Placeholder for a configurable speed limit. Set to 0 for initial testing.
    parameter SPEED_LIMIT = 40;
    always @(*) begin
        if (speed_in > SPEED_LIMIT) begin
            red_led_out = 1'b1;
        end else begin
            red_led_out = 1'b0;
        end
    end
endmodule

//=============================================================================
// Testbench: test_speedometer_tb
// Simulates the clock and hall sensor pulses to verify functionality.
//=============================================================================
`timescale 1ns / 1ps
module test_speedometer_tb;

    // --- Signal Declarations ---
    reg clk;
    reg reset_n;
    reg hall_effect_input;
    wire [6:0] seg;
    wire [3:0] an;
    wire red_led;
    integer errors = 0;

    // Internal signal access is necessary for direct speed verification in the TB
    wire [7:0] DUT_speed_kmh = uut.speed_kmh;

    // --- UUT Instantiation ---
    speedometer_top uut (
        .clk(clk),
        .reset_n(reset_n),
```

```
        .hall_effect_input(hall_effect_input),
        .seg(seg),
        .an(an),
        .red_led(red_led)
    );

    // --- Clock Generation ---
    initial begin
        clk = 0;
        forever #10 clk = ~clk; // 100MHz clock (period 20ns)
    end

    // --- TASK to run a single speed test ---
    task run_speed_test(input integer target_speed);
        // Task-local variables
        integer required_pulses;
        integer pulse_half_period_ns;

        begin
            $display("\n--- Testing Target Speed: %0d km/h ---", target_speed);
            // Inverse of calculation: Pulses = (Speed * 160) / 23 (target_speed is an
integer)
            required_pulses = (target_speed * 160) / 23;

            if (required_pulses > 0) begin
                // Calculate half-period (time between rising/falling edges) in nanoseconds
                // 500M ns = 0.5s. Pulses are counted over a 2s interval, so 4 times as many
pulses.
                pulse_half_period_ns = (500_000_000 / required_pulses) / 2;
                repeat (required_pulses) begin
                    // Generates the pulse train
                    #(pulse_half_period_ns) hall_effect_input = 1'b1;
                    #(pulse_half_period_ns) hall_effect_input = 1'b0;
                end
            end else begin
                 #500_000_000; // Wait for half of the measurement period if speed is 0
            end

            #100_000; // Margin for DUT to finish the calculation and update output

            // Verification (allowing +/- 1 km/h tolerance due to integer math)
            if (uut.speed_kmh >= target_speed - 1 && uut.speed_kmh <= target_speed +
1) begin
                $display("  -> RESULT: PASS. Target: %0d km/h, Actual: %0d km/h",
target_speed, uut.speed_kmh);
            end else begin
```

```
            $display("  -> RESULT: FAIL. Target: %0d km/h, Actual: %0d km/h",
target_speed, uut.speed_kmh);
            errors = errors + 1;
          end
        end
    endtask

    // --- Main Test Sequence ---
    initial begin
        // 1. Reset the device
        reset_n = 1'b0;
        hall_effect_input = 1'b0;
        #200; // Wait for reset to assert
        reset_n = 1'b1;
        $display("--- Starting Spot-Check Simulation ---");

        // 2. Run tests for specific speeds
        run_speed_test(0);
        run_speed_test(10);
        run_speed_test(88);
        run_speed_test(110);
        run_speed_test(112);
        run_speed_test(150);

        // 3. Final Report
        $display("\n-------------------------------------");
        $display("        SPOT-CHECK SUMMARY");
        if (errors == 0) $display("  All spot-checks passed! âœ…");
        else $display("  Finished with %0d errors. â Œ", errors);
        $display("-------------------------------------");

        $finish;
    end

endmodule
```

RTL



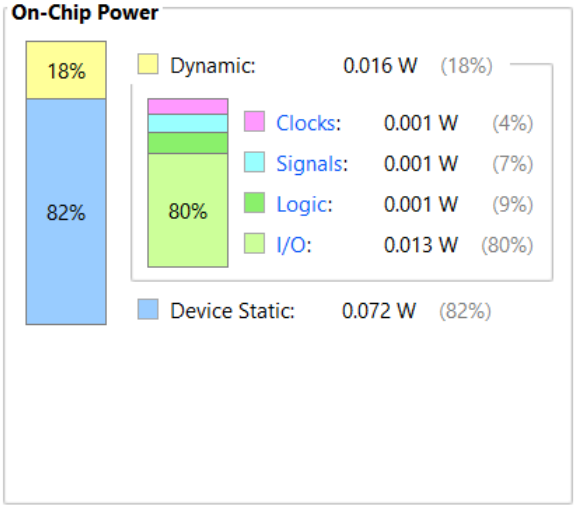
RESOURCE UTILIZATION SUMMARY TABLE

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)
N speedometer_top	134	30	48	134	15	1
display_mux_inst (seven_seg_mux)	4	18	8	4	0	0
pulse_cnt_inst (pulse_counter)	28	12	9	28	0	0
speed_calc_inst (speed_calculator)	103	0	39	103	0	0

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.088 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (11.8 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



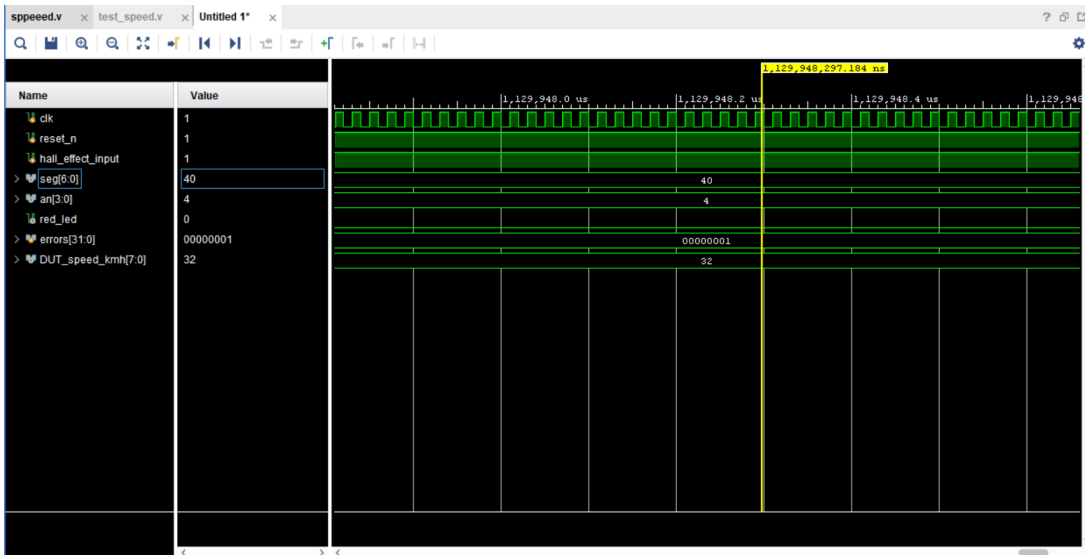
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.622 ns	Worst Hold Slack (WHS):	0.208 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	38	Total Number of Endpoints:	38	Total Number of Endpoints:	31

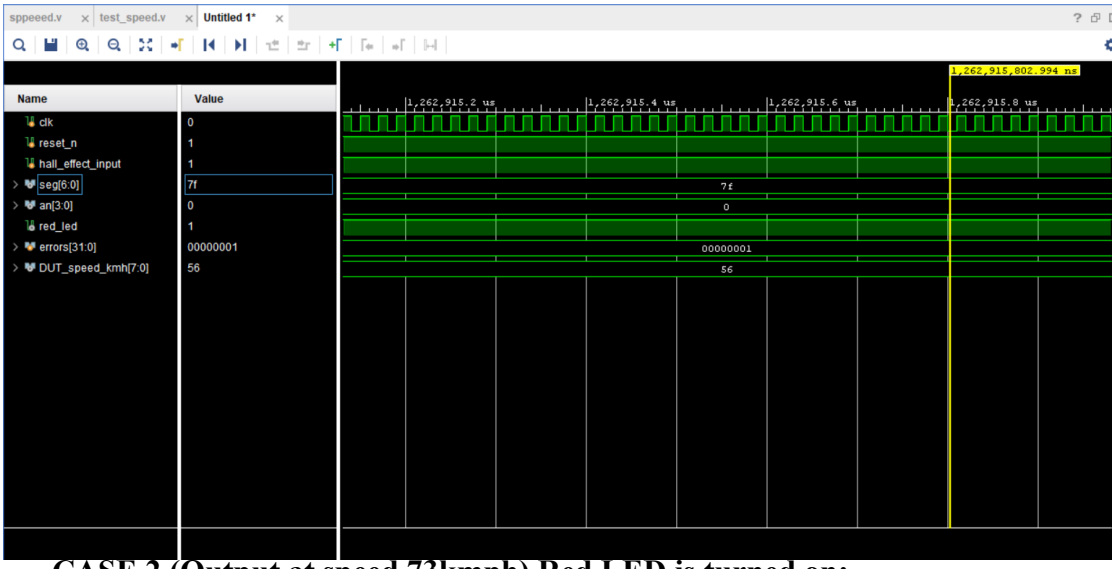
All user specified timing constraints are met.

OUTPUT WAVEFORM (SPEED LIMIT = 40kmph) :

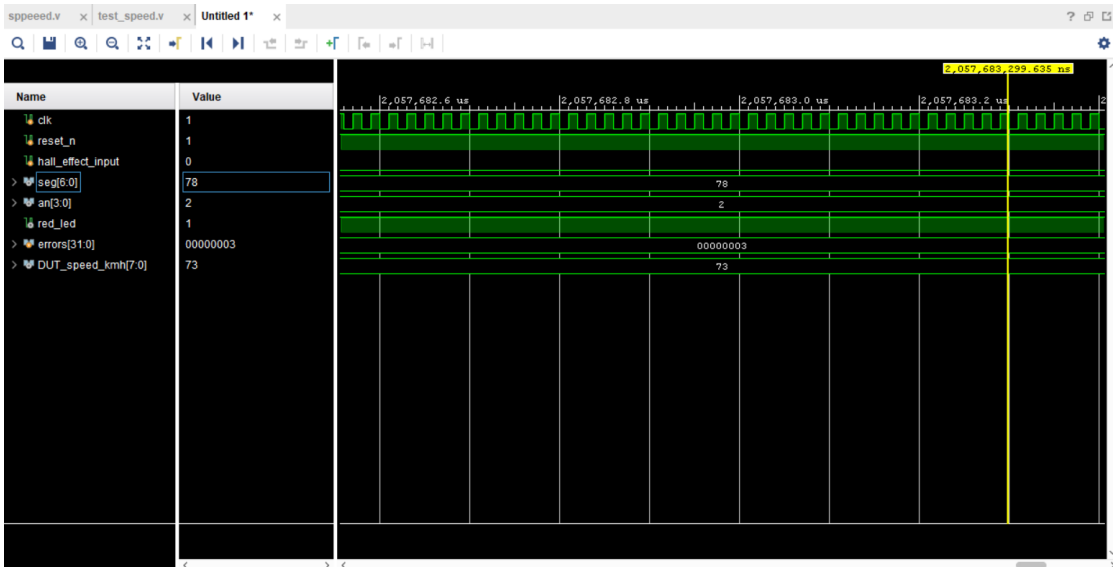
CASE 1 (Output at speed 32kmph) Red LED is turned off:



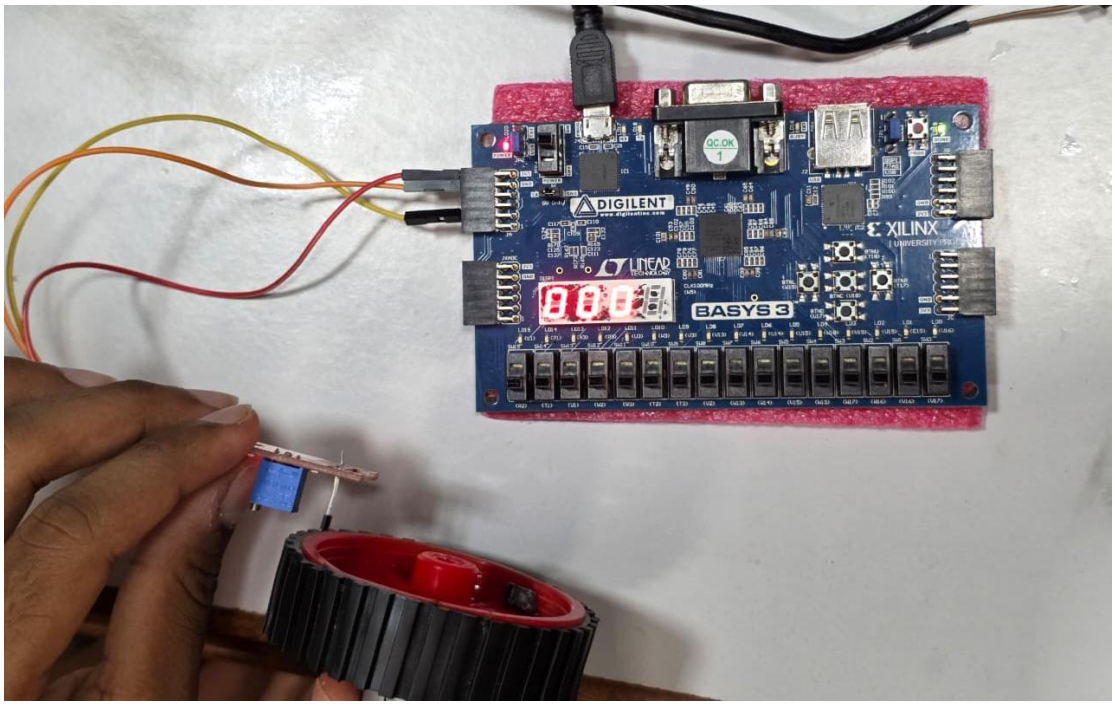
CASE 2 (Output at speed 50kmph) Red LED is turned on:

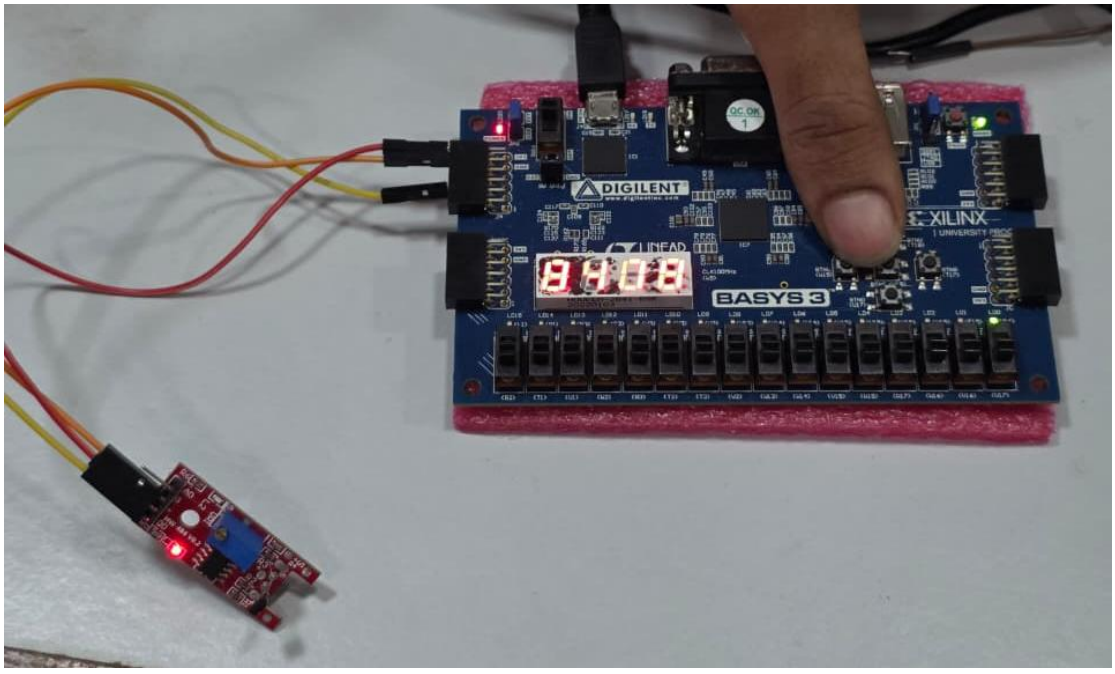


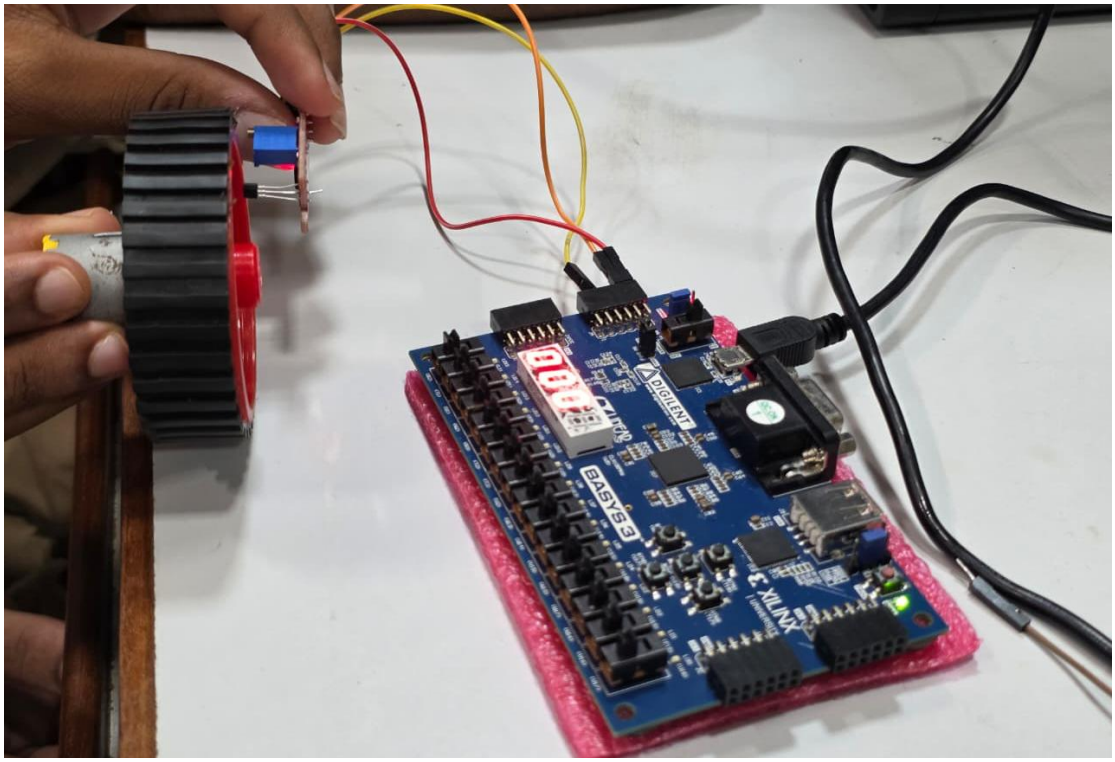
CASE 2 (Output at speed 73kmph) Red LED is turned on:



HARDWARE IMAGES






DIGILENT
BASYS 3
XILINX